Dynamic Data Structures: The Interplay of Invariants and Algorithm Design

Casper Kejlberg-Rasmussen

PhD Dissertation

Department of Computer Science
Aarhus University
Denmark

Dynamic Data Structures:
The Interplay of Invariants and Algorithm Design

A Dissertation
Presented to the Faculty of Science
of Aarhus University
in Partial Fulfilment of the Requirements for the
PhD Degree

by
Casper Kejlberg-Rasmussen
September 30, 2013

Abstract – English

In this thesis I will address three dynamic data structure problems using the concept of invariants. The first problem is maintaining a dynamically changing set of keys – a *dictionary* – where the queries we can ask are: does it contain a given key? and what is the preceding (or succeeding) key to a given key? The updates we can do are: inserting a new key or deleting a given key. Our dictionary has the *working set* property, which means that the running time of a query depends on the query distribution. Specifically the time to search for a key depends on when we last searched for it. Our data structure is *implicit*, meaning that we do not use any extra space than that of the input keys. Our data structure is implicitly encoded through the permutation of the input keys. Other dictionaries with the working set property have constant factor overhead in the space usage, our dictionary has no overhead and thus has optimal space usage, while still attaining the working set bound. Our result is even cache-oblivious and hence also efficient in external memory.

The second problem is to keep a *first-in-first-out* queue where each element also has a priority – called a *priority queue with attrition*. We delete elements at the front and insert elements at the back. When we insert an element then all elements in the queue with an equal or larger priority are deleted – also called *attrited*. We extend previous solutions by Sundar [Sun89] with the concatenation operation. When we concatenate two queues, all elements in the first queue which are larger or equal to the smallest in the second queue are attrited, and the second queue is appended to the first queue. Our result is also efficient in external memory.

The third problem is to maintain a two-dimensional dynamic point set, where the queries ask for the *skyline* of the points contained within a rectangular query area, i.e., all the points which have no point above them and to the right, within the query area. These points are called the *undominated* points of the query area. Our results are the first dynamic results, that are efficient in external memory.

The central concept we use, throughout this thesis, to solve data structure problems is *invariants*. In the design of *dynamic data structures* we have some set of *query* and *update* operations which we want our data structure to support, but we do not choose the order that they are performed in. Invariants are logical statements about our data structure, which are based on the structure of the underlying problem, that we are trying to solve. We can rely on the properties of the invariants when performing queries, and in return we need to ensure that the invariants remain true after we perform updates. When designing data structures there is an interplay between proposing invariants and checking if they can be efficiently maintained in updates and are sufficient for answering queries efficiently. We have solved our data structure problem when we have found a set of "good" invariants that balances these two sides.

Abstract – Dansk

I denne afhandling vil jeg behandle tre dynamiske datastruktursproblemer ved brug af *invarianter*. Første problem er at vedligeholde en dynamisk mængde af nøgler – kaldt en *ordbog*. Vi kan lave forespørgelserne: er en given nøgle indeholdt? og hvad er den foregående (eller efterfølgende) nøgle til denne nøgle? Vi kan foretage opdateringerne: indsæt en ny nøgle eller slet en given nøgle. Vores ordbog har *arbejdsmængde* egenskaben, hvilket betyder at udførselstiden for en forespørgsel afhænger af distributionen af alle forespørgslerne. Udførselstiden for en forespørgsel afhænger af hvornår vi sidst søgte efter den givne nøgle. Vores datastruktur er *implicit* hvilket betyder at vi ikke bruger yderligere plads end det som inputtet bruger. Vores datastruktur er indkodet implicit i permutationen af inputtet. Andre ordbøger med arbejdsmængde egenskaben bruger en konstant faktor mere plads, vores ordbog bruger ingen ekstra plads og har således det optimale pladsforbrug, men opnår stadig arbejdsmængde egenskaben. Vores resultat er tilmed cache-oblivious og er derfor også effektivt i ekstern hukommelse.

Andet problem er at vedligeholde en *first-in-first-out* kø over elementer med prioriteter – kaldet en *prioritetskø med afskæring*. Vi sletter elementer i fronten og indsætter elementer i enden. Når vi indsætter et element så slettes – *afskæres* – alle elementer med en nøgle som er større eller lig med den indsatte. Vi udvider en tidligere løsning af Sundar [Sun89] med en sammenkædningsoperationen. Når vi sammenkæder to køer så slettes alle elementer i den første kø, der er større eller lig det mindste element i den anden kø, og den anden kø sammenkædes med den første kø. Vores resultat er effektivt i ekstern hukommelse.

Tredje problem er at vedligeholde en todimensionel dynamisk punkmængde, hvor forespørgelserne er at rapportere alle punkter som ikke har nogen punkter over eller til højre for sig, inde i et rektangulært område. Disse punkter kaldes *udominerede* punkter og udgøre *skylinen* i det rektangulære område. Vores resultater er de første dynamiske resultater som er effektive i ekstern hukommelse.

I gennem hele afhandlingen bruger vi *invarianter* til at løse datastruktursproblemer. I designet af dynamiske datastrukturer har vi en mængde af forespørgsler og opdateringer som vi ønsker at vores datastruktur skal understøtte, men vi vælger ikke selv rækkefølgen som de udføres i. Invarianter er logiske udsagn om vores datastrukturer som er baserede på strukturen af det problem vi prøver på at løse. Vi kan afhænge af udsagnene fra invarianterne i vores forespørgsler, men tilgengæld skal vi garantere at de forbliver sande når vi udfører opdateringer. Når vi designer datastrukturer er der et samspil mellem at foreslå nye invarianter og at tjekke at de effektivt kan vedligeholdes ved opdateringer og er tilstrækkelige til effektivt at besvare forespørgsler. Vi har løst vores datastruktursproblem når vi har fundet en mængde "gode" invarianter som balancere de to sider.

Preface

During my undergraduate studies in Computer Science I have always been fascinated by many of the elegant and simple data structures shown to us in our algorithms courses, which have also popped up in other courses as well. I clearly remember, when solving exercises in courses, that coming up with a solution to an algorithmic problem was often very difficult and the reference solutions often occurred as completely black magic to us - a common phrase of ours in those days was "how did they come up with that idea?". It is needless to say that I was intrigued by algorithms design and analysis, and when I decided to take a PhD, it was obvious which topic I would choose to do research in.

Within algorithms and data structures there are static and dynamic structures. The dynamic data structures have always been the most interesting and intriguing to me, since many problems in the real world are naturally dynamic. The deep interplay between inherent problem structure and the formulation of invariants has always fascinated me and I wanted to help contribute to this area of research.

This thesis is divided into four Chapters. The first chapter is an introduction to the thesis and the universal tool of invariants, then I define the computational models that our data structures fall within, the problems that we solve and lastly I describe some of the existing data structures and techniques which we use or build upon. The next three Chapters contain the technical contribution of our work and my PhD, the Chapters are, in order, about: Implicit Working-Set Dictionaries, Catenable Priority Queues with Attrition and Dynamic Planar Skyline Queries.

Bibliography of included work

The bulk of the work in this thesis first appeared in the following three papers of which I am a co-author:

[BKRT10] Gerth Stølting Brodal, Casper Kejlberg-Rasmussen, and Jakob Truelsen. A cache-oblivious implicit dictionary with the working set property. In *International Symposium on Algorithms and Computation (ISAAC)*, volume 6507, pages 37–48. 2010

This paper defines the moveable dictionary in Section 2.1 and is the first paper to present an implicit working set dictionary.

[BKR12] Gerth Stølting Brodal and Casper Kejlberg-Rasmussen. Cache-oblivious implicit predecessor dictionaries with the working-set property. In *Symposium on Theoretical Aspects of Computer Science (STACS)*, volume 14, pages 112–123. 2012

This paper extends the ideas of my first paper and achieves the working set bound not only for searches, but also for predecessor and successor queries. The working set dictionary of this paper appears in Section 2.2.

[KRTT+13] Casper Kejlberg-Rasmussen, Yufei Tao, Konstantinos Tsakalidis, Kostas Tsichlas, and Jeonghun Yoon. I/O-efficient planar range skyline and attrition priority queues. In *Proceedings of ACM Symposium on Principles of Database Systems (PODS)*, pages 103–114. 2013

This paper defines the catenable priority queue with attrition, that appears in Chapter 3, and uses it to give dynamic skyline structures for top-open and 4-sided queries. We also show lower bounds in that paper implying that our 4-sided structure is optimal. The skyline results appear in Chapter 4. This paper also optimally solves skyline queries in the static setting, but those results are not covered in this thesis.

I have only included the parts, from the above three papers, of which I have been a major contributer in the research and writing process. Hence I have left out parts of the papers that I have not been a major contributer to. I have included two small results, as an exception to the previous, as the results complement and complete the other results in the thesis.

Disclaimer

Our last paper [KRTT+13] is a merge of two manuscripts the first by Kostas Tsakalidis, Kostas Tsichlas and me, and the second by Yufei Tao and Jeonghun Yoon. On request from Tao and Yoon, I wish to make it clear that Lemma 4.3 and Section 4.2 in this thesis are written based on Lemma 9 and Subsection 5.2 of the full version of [KRTT+13] which are the work of Yufei Tao and Jeonghun Yoon, and are included in this thesis as the results are heavily based on our dynamic top-open structure from Section 4.1 and the point and query set in Subsection 4.3.1 of this thesis.

Acknowledgments

First of all I want to thank Gerth Stølting Brodal for encouraging and accepting me as a PhD student, and catching my interest for algorithms and data structures and especially invariants. Thanks to Lars Arge, the father of Madalgo, for creating an environment with so many good people and interesting visitors. Also a big thanks to Else Magård, the mother of Madalgo, for creating an atmosphere that always makes you feel welcome and happy. Else makes everything work in our everyday life, she is always happy to help with practical stuff and so is Ellen Kjemtrup, who is always helpful with settling travel expenses.

I also want to thank my two office mates Freek van Walderveen and Jesper Asbjørn Sindahl Nielsen who I have had many interesting discussion about open problems, life and occasionally just plain procrastination. And all the many interesting people at Madalgo, especially Kostas Tsakalidis, Pooya Davoodi, Jakob Truelsen, Lasse Deleuran, Bryan Wilkinson and Jungwoo Yang. We have had many funny and interesting discussions, trips, LAN-parties and BBQs. I also want to thank my co-authors Gerth Stølting Brodal, Jakob Truelsen, Jeonghun Yoon Konstantinos Tsakalidis, Kostas Tsichlas and Yufei Tao for a great collaboration.

During my stay abroad, I went to Thessaloniki in Greece and Tel Aviv in Israel. I will firstly thank Kostas Tsichlas, who invited me to the Aristotle University of Thessaloniki, and his wonderful wife Areti Papathanasiou for helping me find housing and taking so good care of me in Thessaloniki. Their help has been invaluable to me. I also want to thank Kostas Tsakalidis who I spent many many hours with in Thessaloniki and Patras, having fun, doing research and exploring the city and its many interesting people. Also a big thanks to all the people from the office in Thessaloniki. I will especially thank Kostas' family: Athanasios Tsakalidis, Aglaia Liopa-Tsakalidi and Dimitris Tsakalidis, who let me stay with them, in the easter and the summer, and took me in as part of their family. We had so many great experiences together. I will also thank Kostas' cousins Katerina and Dimitris and their families along with Valia, Natalia and Katerina for letting us visit and stay at them, while we where roadtripping Greece.

From Tel Aviv I will firstly thank Orgad Keller, who helped me find housing and showed me all the great places in Tel Aviv. A big thanks to Moshe Lewenstein for inviting me to Bar Ilan University. Thanks to all the people from the office, and all Orgads friends, who he introduced me to. I will also like to thank John Iacono which was attending the Stringology workshop in Safed and invited me for a trip to Jordan, we saw many nice attractions, cultures and people, which I will never forget.

Last and most importantly thanks to my parents Else Marie and Niels for always supporting and encouraging me, and for knowing and telling me that I can achieve anything I want to. This thesis is dedicated to my father, who sadly is not here anymore to see me finish.

Casper Kejlberg-Rasmussen,
Aarhus, September 30, 2013.

In memory of my father, who was taken from this world much too early.

Contents

Chapter 1

Introduction

The theory of data structures considers the compact representation of data in a computer such that questions, also called queries, can be answered about the data efficiently. The data can be anything, but most often it is a collection of strings and numbers, and in many cases the data can be considered as a vector of values. The queries can also be anything, but common queries are "what is the name of the person with this phone number?" or "give me the names of all the persons aged between 25 and 60 years earning between 25.000 and 45.000 DKK a month". The first query is an example of a membership query on a dictionary, which asks for the value that corresponds to a certain key. The second is an example of a *range query*, where we want all the persons who fulfills some range of requirements. If we do not need to update the data, the problem is called *static*. If we need to update the data the problem is called *dynamic*. The data structuring problem is to construct a compact representation, that can answer queries fast. If the problem is dynamic the data structure should also perform updates fast. A data structure gives a solution to a problem within certain time bounds, so we call a data structure an *upper bound*. A natural question now arises: what is the best possible bounds that a data structure for a given problem can have? To answer such a question, we need to make a statement that out of all possible data structures, even those that no one has ever thought of yet. We want to give a proof that no one can do better than a given time bound. Such a proof is called a *lower bound*. Since we need to make a proof that says something about *all* possible data structures, also those we have not thought of yet, we need some assumptions about these data structures to give any non-trivial lower bound. This is where computational models comes into play.

A computational model defines what parameters are available to an algorithm and which operations the algorithm is allowed to use. A good model is very simple, i.e., it has few parameters and operations that the algorithm is allowed to use, yet it models the capabilities of our computers well enough to give good performance predictions. A simple model is easier to reason about in proofs when making logical reasoning, but it might not represent the capabilities of computers very well. On the other hand a very detailed model can represent the computers very well, and give very accurate time bounds, but it will be difficult to argue about it, because of the vast amount of parameters and operations available. As a result of these two naturally contradicting properties of a good model, we have many different models of computation, that model different aspects of computers. The result of this is two-fold. Firstly, since each model is reasonably simple, we can devise lower bounds for it. Secondly since each problem poses different challenges, e.g. some problems are so big that they do not fit in memory which makes disk input/output critical and not CPU time, while others do fit in memory so only CPU time is important, we choose the

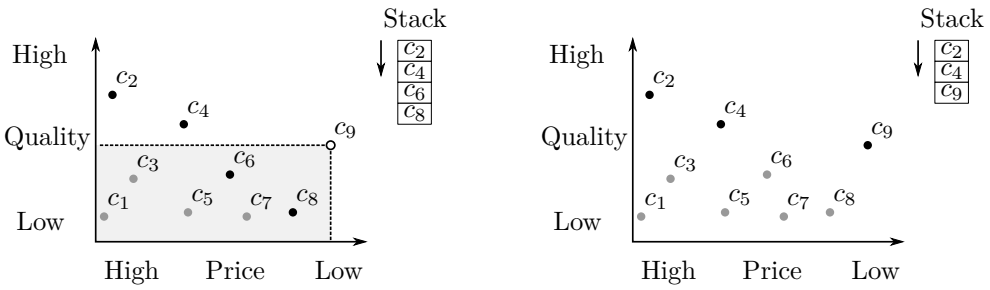

Figure 1.1: Finding the best car offers of a continuous stream of cheaper offers. To the left we receive an offer for car c_9 and to the right we update our stack accordingly.

model that addresses these challenges. So, this means that whenever we look at a problem, we first need to decide which computational model is most relevant for the problem, then using the operations and following the restrictions of the model we can design data structures for the problem. Naturally, one problem can have different complexities in different models, but they are of course related due to the inherent structural complexities of the problem.

The complexity of data structure problems vary greatly from problem to problem, for example solving graph connectivity on a static graph is very simple by numbering each vertex of different connected components using a breadth-first-search. That way we can determine if two vertices are connected by just looking at their component number. On the other hand, if we allow changes to the graph by deleting and adding edges dynamically, then the problem becomes really difficult and solutions are yet to be found that are anywhere near optimal. Another example is the dictionary problem, where in its simplest form we just want to test for membership of elements in a set in the static setting. If we allow insertions and deletions of elements, the problem complexity stays the same in most models. Also many problems which might seem to be different, are in fact just the same basic problem in disguise. We can call these problems the *core problems*. The hardness of a problem comes from the structural complexity of the core problems within it. One way to solve the core problems is to investigate different properties of the problem, and make observations on it that state logical predicates on and implications for the problem. We use these observations to formulate *invariants*, which are conditions of the data structure that we are designing that will always remain true. Now having formulated a set of "good" invariants, proving correctness and time complexities of our data structures is usually easy, as we can rely on the invariants to always be true, and then use their properties to make our proofs based on structural induction. As an illustration of a problem and an invariant which directly gives an elegant solution, we can take the following problem:

> *"We are looking to buy a new car, and of course want the best quality we can get for our money. Since production costs of cars fall all the time, but quality always fluctuates, we know that the price of every new offer will always decrease, but the quality varies. We want to keep the set of all offers which have the best price and quality simultaneously".*

The dynamic data structure problem is to maintain the set of best offers, while we continuously receive new offers. We notice a property of the problem. Given cars c_1 and c_2 with prices $c_i.p$ and qualities $c_i.q$ fulfilling $c_1.p \leq c_2.p$ and $c_1.q \geq c_2.q$, respectively, we can eliminate c_2 from our list of offers to select from, since c_1 is

always better no matter how we value price versus quality. See the left of Figure 1.1. We now keep a stack of best offers, and make the stack fulfill the following invariant: the cars in the stack are sorted simultaneously by quality and price. The car in the bottom of the stack has the highest quality and price and the car in the top has the lowest quality and price. The stack keeps the currently best offers and when we receive a new one, the invariant dictates that we remove all the top cars, which have a lower quality than the new offer, as we already know that the new offer is cheaper than any of our current offers. Then we add the new offer to the top of the stack, see the right of Figure 1.1.

Invariants really are the glue of dynamic data structure design, as they are immensely useful when proving correctness or time bounds of the data structure, because we can just do a structural induction proof of the different cases that the invariants define. The big difference between static and dynamic problems is that for static problems we ourselves can define the order that we process the input data and build our data structures in. Whereas for dynamic problems we have no control over which data is inserted or deleted and when queries are asked, so here the invariants give us a set of rules to follow in designing our procedures to handle the updates and queries.

Now the big question of course comes to mind: "which invariants are good for my problem?". This is where the hard work lies and where the ingenuity, experience and skills of the researcher comes into play, because identifying the right invariants to solve a given problem, requires that one understands the problem and can identify the right structural properties to solve it. We look at the problem and the properties that we have observed and proved about it, then we suggest a set of invariants and then we see if we can maintain them for all the update and query procedures of the problem. If we have a "good" set of invariants, then we can efficiently maintain them for all update procedures and the invariants are strong enough to make us able to answer the queries efficiently. But if we have not found a good set of invariants, then we cannot guarantee that all invariants are true after running some of the update procedures or the invariants are not sufficient to be able to answer the queries. The whole process is very similarly to packing a suitcase with clothes and other usefulness before traveling. We poke something in, that is sticking out of the suitcase, but then something pops out somewhere else. We poke it back in and yet again something pops out. Now if the suitcase is not too filled, i.e., the problem at hand is actually solvable within the desired time bounds, we will eventually succeed in stuffing and closing the suitcase, and hence solving the problem, but it will take a long time of poking. Once the right invariants are found, formulating the algorithms to modify and query the data structure is usually easy or even trivial, and the invariants comprise the heart of the solution, the actual implementation is usually just details. Discovering the right invariants is a long and tedious road of trials and errors, and it can take a long time until a good set of invariants are discovered. Often our failures also gives us insights that we can use in our next attempt at a good set of invariants. The only hope we have while walking the path is that out of all the ideas and all the failures ahead we only need one set of invariants that work, and this is the light ahead the tunnel that drives us!

In the rest of this thesis we will present new dynamic data structures for distribution sensitive dictionaries in Section 1.2 and Chapter 2, priority queues with attrition in Section 1.3 and Chapter 3 and skyline structures along with a few lower bounds for skyline structures in Section 1.4 and Chapter 4. All our structures will be efficient in the external memory model and our dictionaries will also be efficient in the cache-oblivious and implicit models. In Section 1.1 we formally define the computational models, and in Sections 1.2–1.4 we survey previous work and state

our contributions within each problem area. In Section 1.5 we recall previous data structures and techniques which we use or modify in our data structures before presenting our solutions in Chapters 2–4.

1.1 Models

In this section we will formally define the models we work under in our results. We use many different models because they all have benefits and drawbacks as they address different characteristics of our computers in relation to the two naturally contradicting parameters of simplicity and modeling the hardware accurately. The models are the classical *random access machine*, the *implicit* model, the *pointer machine* and the *functional pointer machine* which all are concerned with counting the number of internal memory accesses and CPU instructions used. The last two are the *external memory* model and the *cache-oblivious* model which are concerned with counting the number of external memory accesses, as these operations are a million time slower than internal memory accesses and executing CPU instructions.

1.1.1 Word Random Access Machine

The *Word Random Access Machine* – in short the RAM model – consists of an infinitely large memory divided into *words* that are w bits long and is based on the von Neumann computer architecture [vN45] that all of todays computers use. A data structure is stored in these words and the content of a word can be considered as either an integer, a bit pattern, or an address – commonly called a *pointer* – of another word. The space usage of a data structure storing n elements is $S(n)$ if $S(n)$ words of memory are used. Memory management is not considered in this model, but we implicitly assume that it takes place behind the scene so that the words used have addresses in $[S(n)] = \{0, \ldots, S(n) - 1\}$. See Figure 1.2. It is always assumed that a word can store a pointer to any other word that would be relevant for the data structure in question, and it is commonly assumed that[1] $w = \Omega(\log n)$. This assumption also ensures that any of the words in $[S(n)]$ can point to any other word in $[S(n)]$ as long as $S(n) = n^{\mathcal{O}(1)}$, i.e., the data structure uses at most polynomial space. The *operations* on words are: integer comparisons $(<, =, >)$, arithmetic $(+, -, *, /)$ and bit-wise operations (`AND`, `OR`, `XOR`, `NEG` and shifts), i.e., the standard word operations found in any modern programming language.

All operations can be performed in constant time, and the query time $Q(n)$ and update time $U(n)$ on a data structure are equal to the number of word operations and words accessed – e.g. by following pointers – performed during the query or update.

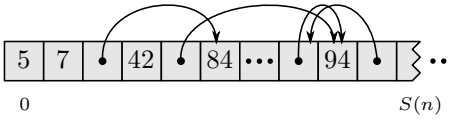

Figure 1.2: Random Access Machine.

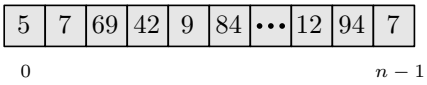

Figure 1.3: Implicit.

[1] Notice that all logarithms written as $\log_b x$ are $\log_b \max(b, x)$ and when we omit the base b then $b = 2$.

1.1.2 Implicit

The *Implicit* model, first defined by [MS79], is a restricted version of the RAM model. The restrictions imposed are that $S(n) = n$, i.e., the data structure only consists of a single array at addresses $[n] = \{0, \ldots, n-1\}$ storing the input elements. Furthermore the data structure is not allowed to change the content of any words, but it is allowed to swap two words. See Figure 1.3. At first this seems like a very limited model, but since there are $n!$ different permutations of the n input elements in the array storing them, we can encode $\log(n!) = \Theta(n \log n)$ bits of information by selecting the appropriate permutation to store our data structure in. To utilize this encoding of information, it is necessary to be able to distinguish elements. So it is often assumed that all the input elements are distinct, which enables us to store $\Theta(n \log n)$ bits of information.

When performing operations on the data structure an additional $\mathcal{O}(1)$ words are allowed to be used to store temporary information. But after the operation ends, these additional words are erased, so only the single array of the input elements is stored between operations. Time usage is counted like in the *RAM* model as the number of word operations and accessed words.

1.1.3 Pointer Machine

The *Pointer Machine* – in short the PM model – introduced by [Tar79] consists of a set of nodes which points to each other. In other words any data structure is considered as a directed graph $G = (V, A)$ where the nodes of V store an input element or a constant amount of information in the form of integers, booleans or pointers to other nodes. The arcs A represent the pointers from nodes to nodes. Each node is constrained to point to at most a constant number of other nodes, and one node $e \in V$ is designated as the *entry node*, which all accesses have to go through. The space usage is the size of G which is $|V|$ as each node contains a constant amount of information. See Figure 1.4.

Any update or query operation always starts at the entry node e and any other node to be visited, has to be found by traversing pointers from e or a node reachable from e. Updates change the information in each node or create new nodes and queries read the information in each node and follow pointers. The time usage is the total number of accessed or created nodes during the query or update operations.

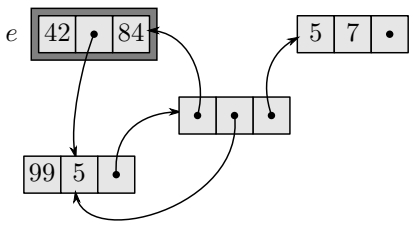

Figure 1.4: Pointer Machine.

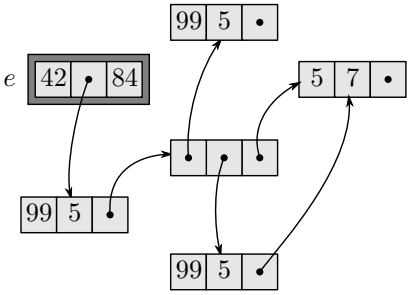

Figure 1.5: Functional Pointer Machine.

1.1.4 Functional Pointer Machine

The *Functional Pointer Machine* – in short the FPM model – is based on the paradigm of functional languages as motivated in [Bac78, Hug89]. The Functional Pointer Machine is a restricted version of the pointer machine model where it is

not allowed to change nodes in V after they are first created, i.e., side-effects are disallowed exactly as in functional languages. This restriction turns out to be of little annoyance but of great benefit in achieving desirable properties of the data structures created which are also highlighted in [KT96, KT99]. See Figure 1.5.

The benefits are that all data structures created are automatically *confluently persistent*, which is the most general form of persistence. The other forms of persistence are *full persistence* and *partial persistence*. A normal *ephemeral* data structure does not keep track of past versions. When an update is made in an ephemeral data structure, the current version is destroyed and only the new version remains. The *version graph* of an ephemeral data structure is just a single node. A partial persistent data structure can query all past versions, but only make updates to the current version. The version graph of a partial persistent data structure is a list, where we can query any version, but only change the latest version, which adds another node to the version list. A full persistent data structure can query and update all past versions. The version graph is a tree, and updating a past version creates a new leaf on the tree. A confluently persistent data structures can query and update any past versions, but moreover, we can also combine any past versions into new versions. The version graph is a DAG.

The only real annoyances of the functional pointer machine model is that it is impossible to make assignments, hence we cannot create cycles in the data structure graph G, i.e., G will always be a DAG, since we can never change a node after it is first created. This restriction implies that some data structures, say e.g. double-linked lists, are not possible to create within the model, but alternatives which are significantly more complex to create, having the same complexities are often possible, e.g. to use catenable deques [KT99] instead.

1.1.5 External Memory

The *External Memory* model also called the *Input/Output* model – in short the EM or I/O model – was first defined in [AV88]. It consists of a *CPU* that works on an *internal memory* of M words and an *external memory* of infinite size divided into consecutive blocks of B words each. The CPU can only work on data in internal memory, but data can be moved back and forth between internal and external memory in I/Os, each I/O moving B words at a time. Since I/Os are significantly slower than CPU instructions, CPU time is considered to be for free and we only count the number of I/Os used. See Figure 1.6.

The space complexity of a data structure is measured as the number of blocks used in external memory. The computational complexity is the number of I/Os performed. An efficient data structure tries to batch words together when doing I/Os so that the average cost of moving a word in or out of external memory is $\mathcal{O}(1/B)$ I/Os instead of the naive $\mathcal{O}(1)$ I/Os.

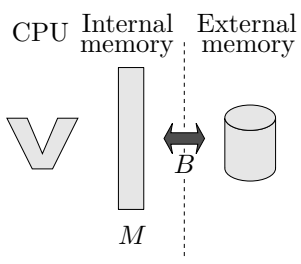

Figure 1.6: External Memory.

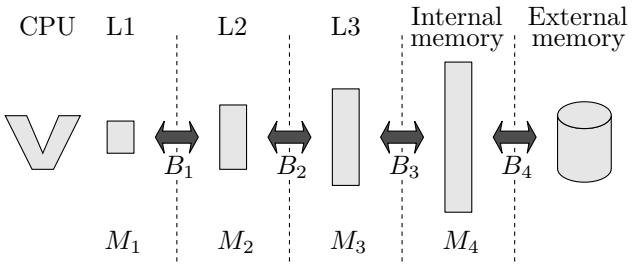

Figure 1.7: Cache-Oblivious.

1.1.6 Cache-Oblivious

The *Cache-Oblivious* model – in short CO model – which first appeared in [FLPR99] defines a hierarchy of memory levels $M_1, \ldots, M_\ell$ and block levels $B_1, \ldots, B_\ell$. See Figure 1.7. The data structure is oblivious to which level $1 \leq i \leq \ell$ we are working on and therefore is not allowed to know neither B_i nor M_i. Hence the data structure is phrased as if it was for the RAM model, but is analyzed in the EM model where B and M are parameters of its asymptotic complexities. So an efficient data structure in the cache-oblivious model is efficient for all values of B and M and hence simultaneously achieves the same asymptotic space and I/O complexities on all levels 1 through ℓ of the memory hierarchy.

Having defined the models of computation used in this thesis, we will now define the problems we are going to solve, survey previous work about each problem area and state our research contributions.

1.2 Distribution Sensitive Dictionaries

In the dictionary[2] problem [CLRS01, Part III, especially Chapter 10] we store a dynamically changing set S of totally ordered elements in the comparison model and want to support search[3], predecessor and successor queries on S. To be specific we want to support the operations:

- $\mathsf{Insert}(e)$ – inserts element e into the dictionary storing S, i.e., change S to become $S \cup \{e\}$.
- $\mathsf{Delete}(e)$ – deletes element e from the dictionary storing S if it exists, i.e., change S to become $S \backslash \{e\}$.
- $\mathsf{Search}(e)$ – returns a boolean telling if e is in the dictionary storing S, i.e., returns true iff $e \in S$.
- $\mathsf{Predecessor}(e)$ – returns the largest element smaller than e if it exits, otherwise $-\infty$ is returned, i.e., returns $\max\{e' \in S \cup \{-\infty\} \mid e' \leq e\}$.
- $\mathsf{Successor}(e)$ – returns the smallest element larger than e if it exits, otherwise ∞ is returned, i.e., returns $\min\{e' \in S \cup \{\infty\} \mid e \leq e'\}$.

There are countless implementations of dictionaries in many different models. In the *PM* model AVL-trees [AVL62], Red-Black-trees [Bay74] and (a, b)-trees [AHU74, sec. 4.9] are the most well known implementations which support all operations in worst-case time $\mathcal{O}(\log n)$.

In the *RAM* model the best solutions solve the dictionary problem when the keys are integers in a universe of size m and we generally assume that $n \leq m$, i.e., we are not in the comparison model in this case. In [vEB75] van Emde Boas give a structure using $\mathcal{O}(m)$ space and supporting all operations in $\mathcal{O}(\log \log m)$ time. This result is optimal as shown by Pătraşcu and Thorup [PT06]. Later Willard [Wil83] improved the space when the dictionary contained significantly less than m elements, i.e., $n \ll m$ to use $\mathcal{O}(n)$ space and still support all operations in $\mathcal{O}(\log \log m)$ time.

In the *Implicit* model there has been a continuous development of implicit dictionaries since the sixties. The first milestone was the implicit AVL-tree [Mun86] having bounds of $\mathcal{O}(\log^2 n)$. The second milestone was the implicit B-tree [FGMP02]

[2]The variation presented here supporting predecessor and successor queries is also often called *the predecessor problem.*

[3]Also often known as membership queries.

having bounds of $\mathcal{O}\left(\frac{\log^2 n}{\log\log n}\right)$ the third was the flat implicit tree [FG06] obtaining $\mathcal{O}(\log n)$ worst-case time for searching and amortized bounds for updates. The fourth milestone is the optimal implicit dictionary [FG03] obtaining worst-case $\mathcal{O}(\log n)$ for search, update, predecessor and successor operations.

In the *EM* model the most famous dictionary implementation is the B-tree [BM72] which support all operations in worst-case $\mathcal{O}(\log_B n)$ I/O's. If we can batch searches together we can get much better amortized bounds by using the Buffter-tree by Arge [Arg03]. The Buffer-tree supports updates and batched searches in $\mathcal{O}(\frac{1}{B}\log_{\frac{M}{B}}\frac{n}{B})$ amortized I/O's per operation.

In the *CO* model there are many different dictionary implementations, [BCR02, FG03] give implementations that support all operations in $\mathcal{O}(\log_B n)$ I/O's worst-case. The data structure of [FG03] is even implicit, which we will use in our contributes to distribution sensitive dictionaries. Like in the EM model if we are satisfied with amortized bounds [BDF$^+$10] supports predecessor, successor and searches in $\mathcal{O}(\frac{1}{\epsilon}\log_B\frac{n}{M})$ worst-case I/O's and updates in $\mathcal{O}(\frac{1}{\epsilon B^{1-\epsilon}}\log_B\frac{n}{M})$ amortized I/O's for $0<\epsilon<1$. Unlike for example the B-tree and the Buffer-tree none of the structures in [BCR02, FG03, BDF$^+$10] support range queries, but there are structures like [BDIW02] which support searches in $\mathcal{O}(\log_B n)$ and range queries in $\mathcal{O}(\log_B n+\frac{k}{B})$, but only updates in $\mathcal{O}(\log_B n+\log^2\frac{n}{B})$. Getting the updates down to optimal $\mathcal{O}(\log_B n)$ I/Os while supporting range queries in $\mathcal{O}(\log_B n+\frac{k}{B})$ I/Os worst-case is a big open problem in the CO model.

If the access sequence does not follow a uniform distribution, we can often give better amortized bounds for search, predecessor and successor queries. Dictionaries that give better amortized bounds for an access sequence are called *distribution sensitive dictionaries*. There a many different distribution sensitive properties. In the following we will define the most popular properties. Before the definitions of the properties we need to define the *working set number* and the *rank distance*.

Definition 1.1 (Working Set Number)**.** *Let $e\in S$ and assume that there have been accesses to ℓ_e distinct elements different from e, since we last accessed e. Then ℓ_e is the working set number of element e.*

The working set number ℓ_e tells when element e was last searched for. See Figure 1.8, where element h has a working set number of 3, then we search for h and it gets a working set number of 0, and the working set number of all elements that had a working set number less than 3 now increase by 1. So the working set number of g, k and a increase by 1.

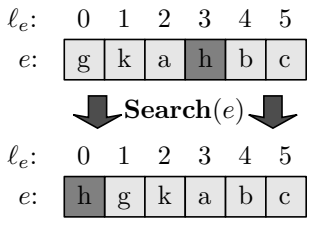

Figure 1.8: Working Set Number.

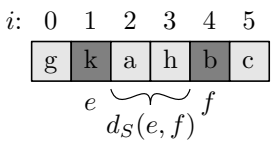

Figure 1.9: Rank Distance.

Definition 1.2 (Rank Distance)**.** *Let $e, f\in S$ then*

$$d_S(e,f)=|\{e'\in S\mid e<e'<f\}|$$

is the rank distance between elements e and f.

The rank distance $d_S(e, f)$ between two elements e and f in S is simply the number of elements between them in rank space, see Figure 1.9, where there are two elements a and h between e and f so $d_S(e, f) = 2$. Now with the working set number and rank distance defined we can define the distribution sensitive properties.

Definition 1.3 (The Working Set Property)**.** *Accessing element e takes $\mathcal{O}(\log \ell_e)$ time.*

An example of an access sequence of length $m \geq n \log n$ that will take total time $\mathcal{O}(m)$ if searching satisfies the working set property, is the sequence $A_1 : 1, 2, 3, \ldots, n, 1, n, 1, n, \ldots, 1, n$. The accesses to 1 and n will take $\mathcal{O}(1)$ time as they have working set number at most 1. But there are also access sequences where a working set dictionary performs bad on, say $A_2 : 1, 2, 3, \ldots, n, 1, 2, 3, \ldots, n, 1, 2, 3, \ldots, n$. Every time we search for an element e in this sequence it will have working set number $\ell_e = n$ as we only accessed it n accesses ago, even though we are accessing an element that is only one position away in key space from the previously accessed element. So the total time to execute A_2 is $\mathcal{O}(m \log n)$.

There are many structures having the working set bound. In the *PM* model the Splay-tree [ST85], the working set structure [Iac01] additionally has worst-case $\mathcal{O}(\log n)$ bounds, the skip-list [BDL08] and B-tree variants [BDL08], which also all have the working set bound. The B-tree is also I/O efficient, giving bounds of $\mathcal{O}(\log_B \ell_e)$ in the EM model. Bose et. al. [BHM09] give randomized bounds that are very space efficient, although not *implicit* as they use $\mathcal{O}(\log \log n)$ additional space.

Definition 1.4 (The Static Finger Property)**.** *Assume that there is some fixed element $f \in S$ that we call the* finger*. Then accessing element e takes time $\mathcal{O}(\log d_S(e, f))$.*

There are not many structures that attain only the static finger property, most achieve the stronger dynamic finger property, defined next, and are in the *PM* model and a few in the *RAM* model. In the *Implicit* model [BNT12] give a structure that is in between, having the dynamic and static finger search property. It supports finger search relative to a static finger f in $\mathcal{O}(\log d_S(e, f))$ time and furthermore supports changing the finger in $\mathcal{O}(n^\epsilon)$ time. They also prove a lower bound showing that this is optimal in the implicit model, but their lower bound breaks down if we allow just one extra word of memory to be stored between operations. Iacono [Iac01] notices that the static optimality property (which will be defined shortly) implies the static finger property and that the working set property implies the static optimality property. So any dictionary with the working set property or the static optimality property also has the static finger property.

Definition 1.5 (The Dynamic Finger Property)**.** *Assume that we accessed element f, which we call the* finger*, just before accessing element e. Then accessing element e takes time $\mathcal{O}(\log d_S(e, f))$, and e now becomes the new finger f.*

The sequence A_2 from above will take $\mathcal{O}(m)$ time to execute if searching satisfies the dynamic finger property as all elements accessed are only one position away from the finger, which is the previously accessed element. The sequence A_1 on the other hand will take $\mathcal{O}(m \log n)$ time as 1 and n have $\mathcal{O}(n)$ positions in difference in key space.

In the *PM* model there are many implementations based on different search tree implementations, [GMPR77] and [HM82] use B-trees as their base structure, [Tsa85] use AVL-trees and all get the finger search bound of $\mathcal{O}(\log d_S(e, f))$ in the worst-case. The Splay-tree [ST85], which is a binary search tree, only gives amortized bounds. Fleischer [Fle93] support updates in $\mathcal{O}(\log^* n)$, this is improved by Brodal [Bro98]

who give a near optimal result of $\mathcal{O}(1)$ insertion and[4] $\mathcal{O}(\log^* n)$ deletion and improves them to both be $\mathcal{O}(1)$ in [BLM+02].

In the *RAM* model [DR94] support finger searches in $\mathcal{O}(\log d_S(e,f))$ and updates in $\mathcal{O}(1)$ amortized time. This was later improved in [AT00] to support finger searches in $\mathcal{O}(\sqrt{\frac{\log d_S(e,f)}{\log\log d_S(e,f)}})$ time and updates in $\mathcal{O}(1)$ time, which is optimal.

In the *Implicit* model there only exist one result for finger search [BNT12] as mentioned before. In the *EM* model the results [GMPR77] and [HM82] use B-trees, and hence also give I/O efficient bounds here. In the *CO* model Bender et. al. [BCR02] give a dictionary supporting finger search in $\mathcal{O}(\log^* d_S(e,f) + \log_B d_S(e,f))$ I/O's and in [BDIW02] they improve the bound to $\mathcal{O}(\log_B d_S(e,f))$.

The unified property, which was first defined by [Iac01], combines the working set property and the dynamic finger search property.

Definition 1.6 (The Unified Property)**.** *Accessing element e takes time*

$$\mathcal{O}(\log \min_{f\in S}(\ell_f + d_S(e,f))).$$

There are not many structures that have the unified property, in the *PM* model [Iac01] give a static structure having the unified bound. This is later extended to be dynamic in [BD04] having the unified bounds for search and insert, but for delete the bound is $\mathcal{O}(\log \min_{f\in S}(\ell_f + d_S(e,f)) + \log\log|S|)$. This extra additive term for delete is later removed in [BCDI07] which attain the unified bound for all operations search, insert and delete. It is also noteworthy to mention that it is not proved that Splay-trees [ST85] have the unified bound as stated in Definition 1.6. The unified theorem in [ST85] is a combination of the working set, static finger and static optimality theorems, and hence does not capture the dynamic finger search property. But in [Iac01, BCDI07] they conjecture that Splay-trees [ST85] do have the unified property stated in Definition 1.6.

In the *EM* model the structures in [BD04, BCDI07] can easily be extended so the logarithm is base B, i.e., if [BCDI07] is extended to the EM model the bounds of all operations would be $\mathcal{O}(\log_B \min_{f\in S}(\ell_f + d_S(e,f)))$. There are currently no structures in the *RAM*, *Implicit* nor *CO* model that attain the unified bound.

Definition 1.7 (The Static Optimality Property)**.** *Let $q(e)$ be the total accesses to element $e \in S$. Then all accesses take a total time of*

$$\mathcal{O}\left(\sum_{e\in S, q(e)\neq 0} q(e)\log\frac{m}{q(e)}\right), \text{ where } m = \sum_{e\in S} q(e).$$

For static optimality there has been a lot of work in the seventies and eighties, the most major results are in [Knu71, AM78, FT83, BST85, ST85]. In the *PM* model Knuth [Knu71] gave a dynamic programming algorithm to construct an optimal binary search tree for S in $\mathcal{O}(n^2)$ time given the access frequencies of each element. This search tree is optimal and not just a constant factor off like later result, but it is only for a static set S. In the *PM* model [AM78] give a dynamic structure attaining the optimal bound within a constant factor without knowing the access frequencies nor estimating them. They use a specific update heuristic, which they prove give the bound of $\mathcal{O}(\sum_{e\in S, q(e)\neq 0} q(e)\log\frac{m}{q(e)})$. In [BST85] they estimate the weights and again get the optimal bound within a constant factor, in [ST85] the famous Splay-tree also get the optimal bound within a constant factor, again with a heuristic, so they do not estimate the access frequencies. In the *EM* model the structure of [BST85] is

[4] Let $\log^{(i)} n = \log(\log^{(i-1)} n)$ for $i \geq 1$ and $\log^{(0)} n = \log n$, then $\log^* n = \min\{i | \log^{(i)} n < 1\}$.

extended to work for B-trees in [FT83] giving bounds of $\mathcal{O}(\sum_{e \in S, q(e) \neq 0} q(e) \log_B \frac{m}{q(e)})$. There are no previous structures in the *Implicit* nor *CO* model that have the static optimality property. But since the working set property implies the static optimality property [Iac01], any working set dictionary will automatically also have the static optimality property.

Contrary to the static optimality property the dynamic optimality property is defined within a model of supported operations and structural constraints. In such a model, we want to obtain bounds that are within a constant factor from the optimal number of operations. The most commonly defined and used search models are:

Definition 1.8 (Search Models)**.** *Before any access a finger is initialized at the root of the structure, which can be considered to be a graph with update operations as defined:*

- *In the* binary search tree *(**BST**) model the unit-cost operations are following pointers and performing rotations.*
- *In the* multi-way branching search tree[5] *(**MWBST**) model the unit-cost operations are following child- or sibling-pointers, and performing splits and joins of nodes.*
- *In the* skip-list *(**SL**) model the unit-cost operations are following forward-, backwards-, parent-, or child-pointers and incrementing or decrementing the height of elements.*

We are now ready to define the dynamic optimality property.

Definition 1.9 (The Dynamic Optimality Property)**.** *Let $OPT_M(X)$ be the minimum number of operations needed to perform the access sequence X in model M. An algorithm A is* dynamically optimal *if it uses $\mathcal{O}(OPT_M(X))$ operations to perform the access sequence X in model M.*

In the search for dynamically optimal structures, the most effort has been put into the BST model. For the MWBST and SL models optimal structures are given in [BDL08]. Optimal structures for the BST model remains to be found. It is conjectured in [ST85] that the Splay-tree is dynamically optimal, and much effort has been put into trying to prove it, but there has been no success yet. But many other interesting properties have been proved for Splay-trees i.e., the dynamic finger conjecture [CMSS00, Col00] the sequential access bound [Tar85, ST85], there are also some lower bounds for binary search trees with rotations by Wilbers [Wil89]. In [DHIP07] Demaine et. al. give a dynamic search tree that is a $\mathcal{O}(\log \log n)$ factor from being dynamically optimal and in [DHI+09] an interesting equivalence is made between binary search trees with rotations and a 2D point set. All these upper and lower bounds are all for the *PM* model and there has not been put much effort into the other models, i.e., *RAM, Implicit* and *CO*. In the *EM* model [BDL08] give a dynamically optimal B-tree which is optimal both in internal and external memory.

1.2.1 Our Contributions

In Chapter 2 we present the first working set dictionaries, in the implicit model, supporting search, predecessor and successor queries within the working set bound. In [BKRT10] we give the first implicit working set dictionary supporting search within the working set bound, but unfortunately predecessor and successor queries have bounds of $\mathcal{O}(\log n)$ time and $\mathcal{O}(\log_B n)$ cache-misses. In [BKR12] we improve

[5] The fanout of a node is unbounded, i.e., there is no bound of the number of siblings.

upon [BKRT10] by also supporting predecessor and successor queries within the working set bound. The moveable dictionary from [BKRT10] appears in Section 2.1 and the working set dictionary from [BKR12] supporting all queries within the working set bound appears in Section 2.2.

In Section 2.1 we show how to make the dictionary of [FG03] moveable, i.e., supporting the operations of move-left (and move-right), which moves the dictionary occupying memory addresses $[i, j]$ to $[i + 1, j + 1]$ ($[i - 1, j - 1]$). We do this by treating the structure of [FG03] as a black box, and hence our construction is general and can be applied with other implicit structures inside, say an implicit binary heap.

Our dictionary in [BKRT10] is in addition to being implicit also cache-oblivious. We support insert, delete, predecessor and successor operations in $\mathcal{O}(\log n)$ time and $\mathcal{O}(\log_B n)$ cache-misses and searches in $\mathcal{O}(\log \ell_e)$ time and $\mathcal{O}(\log_B \ell_e)$ cache-misses. We obtain the results of [BKRT10] by combining many existing ideas; we use the double exponential layout to make it easy for us to code numbers and pointers, we make a moveable version of [FG03] and use this as a black box inside our structure, lastly we implicitly maintain an estimate of the working set number of each element, which enables us to give the bounds for search. We make our estimate by having $m = \lceil \log \log n \rceil$ blocks where the i'th block has size $\mathcal{O}(2^{2^i})$ and within each block we divide the elements into three groups which enables us to estimate the working set number of each element. In block i we have the groups L_i, C_i and R_i. All elements in L_i have a working set number of at least $2^{2^{i-1}}$, all elements in C_i have a working set number of at least $|L_i|$ and all elements in R_i have a working set number of at least 2^{2^i}. For $i < \lceil \log \log n \rceil$ we have that $|L_i| + |R_i| = 2^{2^i}$. The idea is that when we search for an element we find it in block i and have spent $2^i = \mathcal{O}(\log \ell_e)$ time to find it. We remove the element e from block i and insert it in L_0 in block 0 which overflows, then we move an element from R_0 into L_1 which overflows block 1 and so on until we eventually put an element back into block i and fills the hole we made when removing e.

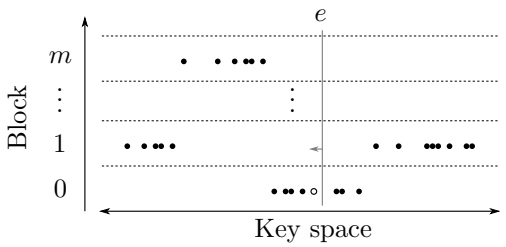

Figure 1.10: Working set estimate of [BKRT10].

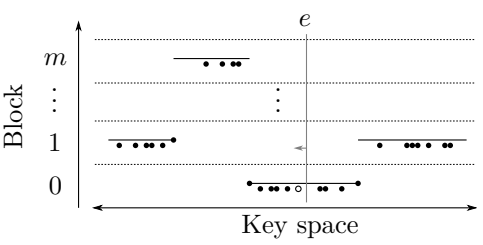

Figure 1.11: Intervals idea of [BKR12].

The problem with the scheme of [BKRT10] for predecessor and successor searches is that if we search for the predecessor of, say e, then we have to look through all blocks $i = 0, \ldots, m$ to be sure that we found the predecessor of e as it might lie in any level i, see Figure 1.10.

Our dictionary in [BKR12] improves upon [BKRT10] by also making the predecessor and successor operations run within the working set bound, i.e., in $\mathcal{O}(\log \ell_{e^*})$ time and $\mathcal{O}(\log_B \ell_{e^*})$ cache-misses, where e^* is the predecessor and successor, respectively, of the element given in the search. The essential idea in getting these better bounds is to view the elements as points in 2D space and then introduce the notion of intervals defined over the points. The horizontal axis is the key space and the vertical axis is the working set estimates of the points, see Figure 1.11. We introduce the notion of *intervals*, and a scheme for maintaining them implicitly. We partition the key space into a set of disjoint intervals of which their union gives the whole key space, we use

some of the elements to encode the intervals, see Figure 1.11. Each interval lies in exactly one block/level i, and hence when we find the block/level i which contains an interval that is intersected by e, then we have found the predecessor of e and are guaranteed that we do not need to look any further in any higher block/levels. The intervals idea is further described in Section 2.2.

1.3 Priority Queues with Attrition

In 1989 Sundar [Sun89] introduced the *priority queue with attrition*, termed a PQA. A PQA Q is a first-in-first-out queue where each element e also has a priority or key[6]. A PQA Q supports the following operations:

- Find-Min(Q) returns $\min(Q)$.
- Delete-Min(Q) returns $\min(Q)$ and removes it from Q. The resulting PQA is $Q' = Q \backslash \{\min(Q)\}$ and the old Q is discarded.
- Insert-and-Attrite(Q, e) inserts element e at the end of Q and *attrites* all elements $e' \in Q$ that have a larger key than the key of e. The resulting PQA is $Q' = \{e' \in Q \mid e' < e\} \cup \{e\}$, and the old Q is discarded. The elements $\{e' \in Q \mid e' \geq e\}$ are said to be *attrited*.

The important difference from a normal FIFO queue and a normal heap is the concept of *attrition*, i.e., that the insertion of a new element e into Q can delete - also called attrite - existing elements in Q which have a higher or equal key to that of e, see Figure 1.12. The black point with white center is the minimum element of the PQA and will be returned by Find-Min and Delete-Min, the black point with gray center is inserted by Insert-and-Attrite and will attrite/delete the gray elements with higher or equal key value in Q.

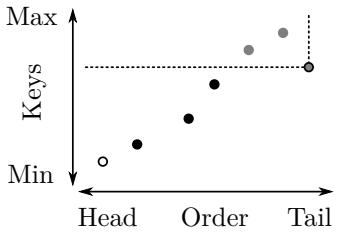

Figure 1.12: PQA-operation effects.

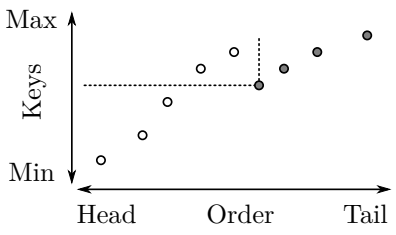

Figure 1.13: Concatenation of PQAs.

It might not be obvious why the attrition property of PQAs is useful but it has many important applications. The attrition property is essential in river routing [CS84], which is a special case of VLSI layout. Attrition also elegantly optimally solves the problem of pagination of scrolls in [LH85], where there have been various initial solutions [McC77, DF84]. Gajewska and Tarjan give deques with heap order in [GT86] which implement a similarly structure to the PQA. All these results are in the *PM* model. Brodal and Tsakalidis [BT11] use the PQA to solve the problem of maintaining dynamically a 2D point set on which they can answer 4-sided and 3-sided top-open orthogonal range queries. Their results are in the *PM* and *RAM* model.

[6] We will not distinguish between an element and its key, and we will use the symbol e of the element to denote both.

PQAs can also be viewed as storing points in 2D. Given two points $p, q \in \mathbb{R}^2$ lets say that p *dominates*[7] q iff $p_x \geq q_x$ and $p_y \leq q_y$. If we only want the undominated points (also called the skyline points) of a point set P then we can just build a PQA over all $p \in P$ where p_x is the insertion order and p_y is the key. Another useful operation on PQAs in relation to computing skyline points is the concatenation operation, see Figure 1.13:

- Catenate-and-Attrite(Q_1, Q_2) appends Q_2 at the end of Q_1 and attrites elements in Q_1. The resulting PQA is $Q' = \{e \in Q_1 \mid e < \min(Q_2)\} \cup Q_2$, and Q_1 and Q_2 are discarded.

If we assume that all the PQA operations are non-destructible[8] then we can build an (a, b)-tree on the x-axis over the point set P and store PQAs over the elements in the leafs, where the key of point (x, y) is $-y$, and store the concatenation of the children for internal nodes. With this simple structure we can answer 3-sided skyline queries $[x_1, x_2] \times [y, \infty[$ by doing as many Catenate-and-Attrite's as the height of the tree times the fanout, plus doing as many Delete-Min's as the size of the skyline. This will be explained in detail in Chapter 4.

1.3.1 Our Contributions

In Chapter 3 we present PQAs extended to support the additional operation of Catenate-and-Attrite. Our extension turns a PQA into a recursively defined tree structure, these results first appeared in [KRTT+13]. Instead of having a PQA store elements, it now stores records, where a record consists of a buffer and a pointer to another PQA. The notion of records with their buffers allows us to make the PQA I/O-efficient and hence in the *EM* model we support the operations of Delete-Min, Find-Min, Insert-and-Attrite and Catenate-and-Attrite in $\mathcal{O}(1)$ I/Os and $\mathcal{O}(\frac{1}{B})$ I/Os amortized assuming $\mathcal{O}(1)$ critical records are pre-loaded for each PQA.

The total space usage for a set of k PQAs where we have performed i insertions and d deletions is $\mathcal{O}\left(\frac{i-d}{B}\right)$ blocks and we require that the memory size M fulfills $M = \Omega(kB)$ to guarantee the amortized bounds.

Assuming that we have spent $\mathcal{O}(1)$ I/Os beforehand on each PQA to maintain some property (to be specified in Chapter 3) we show that we can catenate k PQAs without doing any I/Os, assuming that the critical records of the PQAs are pre-loaded. This is summarized in Lemma 3.1.

1.4 Dynamic Skyline Queries

Let $p, q \in \mathcal{R}^d$ be two d-dimensional points over some totally ordered domain $\mathcal{R}$. We say p *dominates* q if and only if $\forall i = 1, \ldots, d$: $p_i \geq q_i$, i.e., p has larger or equal coordinates than q. For a point set $P \subseteq \mathcal{R}^d$ we define the skyline of P to be all undominated points of P. We want to maintain a dynamic point set $P \subseteq \mathcal{R}^d$, and be able to report the skyline of $Q \cap P$ for a given query $Q = [\alpha_1, \beta_1] \times \cdots \times [\alpha_d, \beta_d] \subseteq \mathcal{R}^d$. The domain $\mathcal{R}$ is most often chosen to be the set of real numbers $\mathbb{R}$, a general integer universe $U = \{1, \ldots, |U|\}$ or rank space $[n] = \{1, \ldots, n\}$. When $\mathcal{R}$ is the real domain $\mathbb{R}$, it costs one time unit to compare two coordinates, where as for integer universe U and for rank space we can use bit manipulation to compare more coordinates at once.

[7] Normally domination is defined as p dominates q iff $p_x \geq q_x$ and $p_y \geq q_y$. We notice from Figure 1.12 that the definition above is symmetric but different to simplify the presentation.

[8] Which can easily be achieved as we will see in Chapter 3.

Skyline points find applications in databases and any kind of optimization where we are interested in finding the vectors/points with maximal values over multiple dimensions. One example is keeping a list of cheap and good quality cars as mentioned in the introduction. A prominent example is the problem of choosing between a range of products based on their price and quality, see Figure 1.14. The skyline points are the products for which there are no other products of a lower price and higher quality, see Figure 1.14, e.g. if we look at points p_1 and q_1 then p_1 dominates q_1 by having a higher quality, and p_1 also dominates q_2 by having a lower price, finally p_1 also dominates q_3 by both having a higher quality and a lower price. If we look at p_1 and p_2 then p_1 has a lower price but p_2 has a higher quality, so we cannot say one is better than the other unless we state how much we value a low price compared to high quality. Skyline queries have applications not only for 2-dimensions but for arbitrary dimensions. Another example is a tourist looking for a hotel in Bahamas. The tourist values a low price and a short distance from the hotel to the beach, but also wants a hotel with a high rating by other customers, see Figure 1.15, here the skyline points or hotels are again those that are undominated by other hotels and hence give some compromise between price, distance and rating.

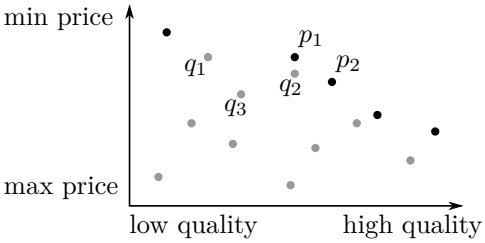

Figure 1.14: Price vs. quality of products.

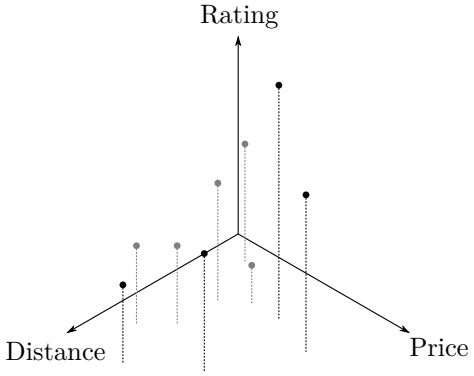

Figure 1.15: The skyline of hotels.

The complexity of finding the skyline of a point set is not completely known. In [KLP75] they show that the total number of needed comparisons $C_d(n)$ to find the skyline of a point set P with n points is determined by the dimensionality of the point set. For $d = 1$ they show that $C_1(n) = n - 1$, for $d = 2, 3$ they show that $C_d(n) = \mathcal{O}(n \log n)$ and for $d \geq 4$ they show that $C_d(n) = \mathcal{O}(n \log^{d-2} n)$. Finally they also show that for $d \geq 2$ there is a lower bound of $C_d(n) \geq \lceil \log n! \rceil = \Omega(n \log n)$.

For $d = 1$ the problem of finding the skyline is trivial as it only requires a scan and a stack or list to keep the points with the maximal value. For $d = 2$ we can use a PQA after we have pre-sorted the points in one dimension. But for $d \geq 3$ the problem is not so easy.

Skyline queries have been studied extensively in the computer science literature due to their wide range of applications to multi-criteria optimization of naturally contradicting attributes, see [BKS01, KLP75, OvL81, FR90, Jan91, dFGT97, Kap00, BT11, DGK+12, KDKS11, SLNX09, SSK09, ST11, BCP08, KRR02, MPJ07, PTFS05, CGGL05] and the references within. Within the database community general high dimensional skyline queries have been of main interest [KLP75, SSK09, ST11, SLNX09, SSK09, ST11, BKS01, BCP08, KRR02, MPJ07, PTFS05], the focus there has been on utilizing characteristics of the input data, e.g. queries on pre-sorted data [BCP08], data sets with low-cardinality domains [MPJ07] or characteristics of the access to the data, e.g. streaming [SLNX09, SSK09] or how fast query results can be returned [KRR02] or extending SQL with a skyline operator [BKS01].

Within theory the skyline problem has mostly been studied in two dimensions [KLP75, OvL81, FR90, Jan91, dFGT97, Kap00, BT11, KDKS11, DGK+12], both for static point sets and dynamic point sets with various update restrictions. Kung et. al. [KLP75] give upper and lower bounds for the number of needed comparisons to find the skyline of a static point set. They can find the skyline in $\mathcal{O}(n \log n)$ time for $d = 2, 3$ and in $\mathcal{O}(n \log^{d-2} n)$ time for $d \geq 4$.

Two Dimensions

For two dimensions there are many variants of the general skyline query $Q = [\alpha_1, \beta_1] \times [\alpha_2, \beta_2]$ depending on which of the boundaries we put at $\pm\infty$, this gives rise to 7 important subcases, see Figure 1.16. Top-open and right-open queries are symmetric and similarly left-open and bottom-open queries are symmetric. Contour and dominance queries are captured within top-open queries and anti-dominance queries are captured within left- and bottom-open queries. It turns out, as we will show in Chapter 4 that anti-dominance queries are harder to answer than top-open queries.

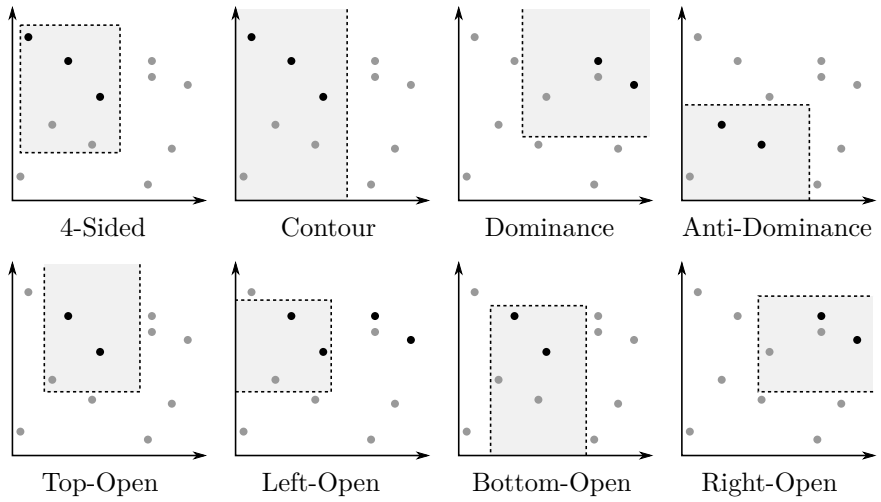

Figure 1.16: All the important subcases of general skyline queries.

In the setting of a dynamic point set in two dimensions using linear $\mathcal{O}(n)$ space, [OvL81] gives a data structure that can maintain the skyline of a point set with a $\mathcal{O}(\log^2 n)$ time cost per update and report it in $\mathcal{O}(t)$ time when the skyline contains t points. Frederickson et. al. [FR90] improves this to $\mathcal{O}(\log n)$ time for insertions while keeping the time for deletion and reporting the skyline unchanged. Janardan [Jan91] matches the bounds of [FR90] but additionally supports contour queries, i.e., reporting the skyline withing $Q \cap P$ where $Q =]-\infty, \beta_1] \times]-\infty, \infty[$. In [dFGT97] d'Amore et. al. maintains the skyline with $\mathcal{O}(\log n)$ updates, but they only allow changes to the point set at the extremes on the x-axis of the point set. Kapoor [Kap00] gives another structure that maintains the skyline with $\mathcal{O}(\log n)$ cost per update without restrictions on which points can be updated, but reporting the skyline takes $\mathcal{O}(t + c \log n)$ time where there are t points in the skyline and c points have been updated since the skyline was last reported. All the mentioned results are in the *PM* model. Finally Brodal and Tsakalidis [BT11] maintain a dynamic point set P with $\mathcal{O}(\log n)$ update costs and can report the skyline for top-open queries, i.e., queries of the form $Q = [\alpha_1, \beta_1] \times [\alpha_2, \infty[$ in $\mathcal{O}(\log n + t)$ time in the *PM* model and $\mathcal{O}(\log n / \log \log n)$ updates and $\mathcal{O}(\log n / \log \log n + t)$ queries in the *RAM* model.

For general two dimensional queries, i.e., queries of the form $Q = [\alpha_1, \beta_1] \times [\alpha_2, \beta_2]$, on a dynamic point set, Brodal et. al. [BT11] can answer these in $\mathcal{O}(\log^2 n + t)$ time and performs updates in $\mathcal{O}(\log^2 n)$ time using $\mathcal{O}(n \log n)$ space in the *PM* model. For a static point set [KDKS11] improves the query to be $\mathcal{O}(\log n + t)$ time using $\mathcal{O}(n \log n)$ space. These bounds are in the *PM* model, in the *RAM* model [DGK+12] give a structure taking $\mathcal{O}(n \frac{\log n}{\log \log n})$ space, supporting queries in $\mathcal{O}(\frac{\log n}{\log \log n} + k)$ time on an $n \times n$ grid, i.e., in rank space.

Higher Dimensions

There are only a few results for higher dimensions. Kung et. al. [KLP75] find the skyline of a d-dimensional static point set in $\mathcal{O}(n \log^{d-2} n)$ for $d \geq 3$. Bentley [Ben80] give the same bound, but in a generalized framework which solves many more related geometric problems. Kirkpatrick et. al. [KS85] give output sensitive versions of [KLP75]. If k is the size of the skyline then [KS85] finds it in $\mathcal{O}(n \log k)$ time for $d = 2, 3$ and in $\mathcal{O}(n \log^{d-2} k)$ time for $d \geq 4$. If the output is small for $d \geq 4$ then they get better bounds of $\mathcal{O}(n \log^{d-3} k)$ when $k^2 \leq n$ and $\mathcal{O}(n \log k)$ when $k^{2+\epsilon} \leq n$. These bounds are in the *PM* model, but in *RAM* Gabow et. al. [GBT84] are able to do better, for $d \geq 4$ they find the skyline in $\mathcal{O}(n \log^{d-3} n \log \log n)$ time. If the point set follows a distribution such that each coordinate is independent from all the others [DZ04] show that the skyline can be found in $\mathcal{O}(n)$ expected time. Sheng et. al. [ST11] later extend the bound of [KLP75] into the *EM* model and finds the skyline in $\mathcal{O}(\frac{n}{B} \log_{m/B}^{d-2} \frac{n}{B})$ I/Os for $d \geq 3$. In [SLNX09] the authors give two multi-pass streaming algorithms that find the skyline in $\mathcal{O}(\log n)$ passes using $\mathcal{O}(m \log n)$ space and another that finds it using only w space in $\mathcal{O}(m \frac{\log n}{w})$ passes, both algorithms are randomized.

1.4.1 Our Contributions

In Chapter 4 we present the first I/O-efficient dynamic data structures being able to answer top-open and 4-sided skyline queries. We use the PQA's extended with the concatenation operation in the *EM* model from Chapter 3 as an essential substructure of our solutions. Our structures use $\mathcal{O}(\frac{n}{B})$ blocks of space, the top-open structure supports updates in $\mathcal{O}(\log_{2B^\epsilon} \frac{n}{B})$ I/Os and queries in $\mathcal{O}(\log_{2B^\epsilon} \frac{n}{B} + \frac{k}{B^{1-\epsilon}})$ I/Os for any $\epsilon \in [0, 1]$. Our 4-sided structure supports updates in $\mathcal{O}(\log \frac{n}{B})$ I/Os amortized and queries in $\mathcal{O}((\frac{n}{B})^\epsilon + \frac{k}{B})$ I/Os. Our top-open structure consist of an $(B^\epsilon, 2B^\epsilon)$-tree with PQAs build on the leaves and on the children of internal nodes. The concatenation operation of the PQAs makes updates and queries easy to perform. Our 4-sided structure uses the dynamic top-open structure as black boxes inside of it. These results first appeared in [KRTT+13], that paper also gives optimal static skyline structures for top-open queries.

In the graph traversal framework of Chazelle [Cha90] we give a point and query set which we use to show that any structure for the anti-dominance reporting problem in the *PM* model requires $\Omega(n \frac{\log n}{\log \log n})$ space if queries are supported in $\mathcal{O}(\log^{\mathcal{O}(1)} n + k)$ time. We furthermore use the point and query set and the indexability theorem to give a lower bound in the indexability model which states that any structure supporting anti-dominance queries using $\mathcal{O}(\frac{n}{B})$ space must incur $\Omega((\frac{n}{B})^\epsilon + \frac{k}{B})$ I/Os when answering a query. Hence our 4-sided dynamic structure is optimal with respect to the query, but not the updates as it uses the top-open structure with $\epsilon = 0$ as a black box which gives updates taking $\mathcal{O}(\log \frac{n}{B})$ I/Os instead of $\mathcal{O}(\log_B \frac{n}{B})$ I/Os.

As an open problem of [KRTT+13] we want to make the reporting term a clean $\mathcal{O}(\frac{k}{B})$ for our top-open structure. The reason why it is only $\mathcal{O}(\frac{k}{B^{1-\epsilon}})$ now, is that

the buffer size of our PQAs is $b = B^{1-\epsilon}$ and the fanout of our $(a, 2a)$-tree which has PQAs as secondary structures, is $a = 2B^{\epsilon}$ to make sure that we can load all the PQAs of the children in $\mathcal{O}(1)$ I/Os. If we increase the buffer size to be $b = \mathcal{O}(B)$ we will only be able to have $\mathcal{O}(1)$ children, hence the update bounds and the log term of the query will be base 2 and not base B. We also notice that if we manage to make an optimal top-open structure supporting queries in $\mathcal{O}(\log_B \frac{n}{B})$ I/Os and queries in $\mathcal{O}(\log_B \frac{n}{B} + \frac{k}{B})$ I/Os then our 4-sided structure will also become optimal as it uses the top-open structure as a black box.

1.5 Preliminaries

In this section we will restate, for completeness, the previously known data structures and techniques which we use or modify in our new contributions. We will describe (a, b)-trees, due to Hopcroft [AHU74, Sec. 4.9], and the augmentation of them with secondary structures, priority queues with attrition due to Sundar [Sun89] of which we will give the details which he left out of his third solution, the double exponential layout and the functional pointer machine trick.

1.5.1 (a, b)-Tree

The (a, b)-tree is a generalization of $(2, 3)$-trees [AHU74, Sec. 4.9] invented by Hopcroft in 1970 and B-trees invented by Bayer and McCreight [BM72]. They provided the basis for the famous Red-Black tree by Bayer [Bay74]. In contrast to Red-Black trees, (a, b)-trees only have very few invariants, and it is obvious how to maintain them.

Let $a, b \in \mathbb{N}$ where $2 \leq a < b$ and $a \leq \lfloor \frac{b+1}{2} \rfloor$, an (a, b)-tree consists of internal nodes and leaf nodes. An internal node contains l keys $k_1, \ldots, k_l$ and $l + 1$ child pointers $p_1, \ldots, p_{l+1}$ which are ordered as $p_1, k_1, p_2, k_2, \ldots, p_l, k_l, p_{l+1}$, i.e., where all keys in child p_i are larger than k_{i-1} for $i = 2, \ldots, l + 1$ and strictly smaller than k_i for $i = 1, \ldots, l$ where $a - 1 \leq l \leq b - 1$. A leaf node just contains a single key k. The invariants for an (a, b)-tree are:

I.1 All internal nodes, except the root have between a and b children.

I.2 The root has at most b children.

I.3 All paths from the root to a leaf have the same length.

Let n the number of leafs of the tree and say the height of the tree, i.e., the length from the root to any leaf, is h. From I.1 and I.3 we have that $a^h \leq n$ hence $h \leq \log_a n$. But we also need the tree to be high enough to contain all n leaves so we have $n \leq b^h$ hence $\log_b n \leq h$. Altogether we have that $\log_b n \leq h \leq \log_a n$. Since $2 \leq a < b$ the height of the tree is $\mathcal{O}(\log_2 n)$ and it is balanced.

Searching and Reporting

It is easy to *search* for a key k in the tree. We start at the root r having keys and child pointers $p_1, k_1, p_2, k_2, \ldots, p_l, k_l, p_{l+1}$, we find the smallest key k_i strictly larger than k and then recursively search the tree pointed to by p_i, if no such key exists we recurse at p_{l+1}. When we end up at a leaf, we simply return true if k is equal to the key of the leaf, else we return false. In Figure 1.17 if we search for 7, we follow the first pointer of the root as 55 is the smallest key strictly larger than 7. Then we follow the second child, as 15 is the smallest key strictly larger than 7, and finally at the leaf 7 we report true as its key is the same as the search key. To search for the *predecessor* of k we do the same as for search, except at a leaf with key k' if $k' \leq k$

we report k', else we report $-\infty$. In Figure 1.17 if we searched for the predecessor of 8 we would find 7, and if we searched for the predecessor of 4, we will end up at 5 and report $-\infty$ as the predecessor. The *successor* searches are symmetric.

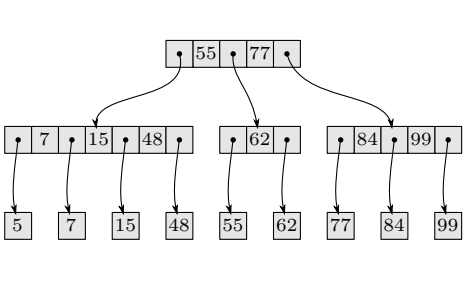

Figure 1.17: Example of an $(2, 4)$-tree.

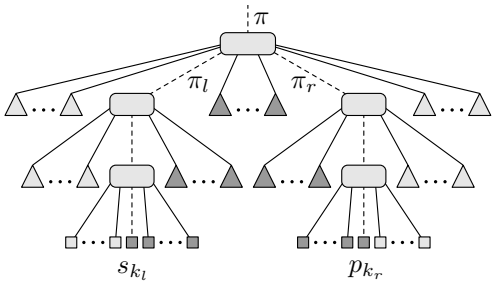

Figure 1.18: Range search in an (a, b)-tree.

If we want to do *range reporting* for all keys in the range $[k_l, k_r]$ we simply find the two search paths $\pi\pi_l$ and $\pi\pi_r$ to the successor leaf s_{k_l} of k_l less than k_r and the predecessor leaf p_{k_r} of k_r greater than k_l, see Figure 1.18. There are $\mathcal{O}(b \log_a n)$ subtrees hanging completely inside the range $[k_l, k_r]$. For each subtree from left to right we now simply do a post-order traversal of it, reporting the key of every leaf that we encounter.

Updates

In order to update the (a, b)-tree the concepts of splitting and joining nodes of the (a, b)-tree are essential. We will now describe the simple updates of inserting and deleting a key k. Afterwards we will describe how to augment the (a, b)-tree with secondary structures.

Insert(k) When inserting a new leaf e with key k we find the leaf of the predecessor k' of k. We insert e into the parent u of k'. This might violate the degree constraint of u so it contains $b + 1$ children. If this happens we will call the *split* procedure for u.

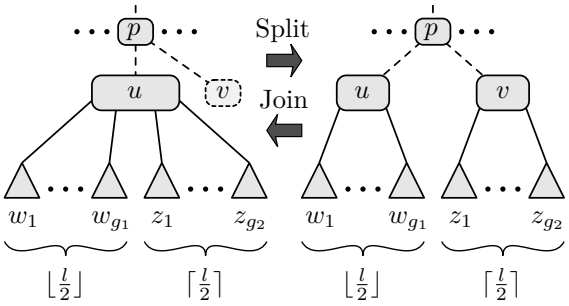

Figure 1.19: Splitting and joining nodes of an (a, b)-tree. Node v in the left side is only relevant for the join procedure.

Split(u) Let u be an internal node with l children. If $a \leq l \leq b$ we do nothing, else if $b + 1 \leq l$ we split u's l children into two groups of size $g_1 = \lfloor \frac{l}{2} \rfloor$ and $g_2 = \lceil \frac{l}{2} \rceil$, assign the first group to u and make a new node u' which we assign the second group to. We now insert u' - using the insert procedure where u' plays the role of e and

the key k is the key before g_2 - as u's right sibling into u's parent p if it exists, else we make a new node r and make u and u' the children of r and r will now be the new root. See Figure 1.19.

Delete(k) We find the leaf e with key k by a search for k. When deleting e from some internal node u, we might violate the degree constraint of u so it contains $l \leq a-1$ children. If u is the root node and $l \geq 2$, this is not a problem. If u is the root and $l = 1$ then we delete u and make its single child the new root. Otherwise we will call the *join* procedure on node u.

Join(u) Let u be an internal node with l children. If $a \leq l \leq b$ we do nothing, else if $l \leq a-1$ we join u's l children with its left or right sibling v, we assume[9] without loss of generality that v is the right sibling of u. We take all of v's children and make them the rightmost children of u. If this makes u have between a and b children, we follow the delete procedure where v plays the role of e, else u has between $b+1$ and $a+b-1$ children, so we delete node v and call split on u. See Figure 1.19.

Time Bounds

We will now analyse the time bounds of the above procedures. Insert does $\mathcal{O}(b)$ work on a leaf or an internal node and might call split on the parent. Likewise split does $\mathcal{O}(b)$ work on an internal node and might call insert on the parent. Hence since the height of the tree is $\mathcal{O}(\log_a n)$ and we always move one level op the tree in both insert and split, the total time usage is $\mathcal{O}(b \log_a n)$.

For delete and join, this is similarly. Delete does $\mathcal{O}(b)$ work and might call join on the parent of the node we delete. Join does $\mathcal{O}(b)$ work and might call delete on a sibling of the node we are joining, or call split on the node we are joining. So in delete we always goes one node up the tree, and in join we either call delete or split, which goes one node up the tree. So the total time usage is again $\mathcal{O}(b \log_a n)$.

Range reporting takes $\mathcal{O}(b \log_a n)$ time. Finding the two search paths $\pi\pi_l$ and $\pi\pi_r$. It takes $\mathcal{O}(t)$ time to report the leafs inside the two search paths. So in total reporting takes $\mathcal{O}(b \log_a n + t)$ time. Searching is a special case of reporting where $t = 1$ so it takes time $\mathcal{O}(b \log_a n)$.

The amortized update times of (a,b)-trees are even better than their worst-case times of $\mathcal{O}(b \log_a n)$. The amortized update time is $\mathcal{O}(1)$ when $b \geq 2a$ starting from an empty tree [BF00].

Augmenting the Search Tree

In Chapter 4 we will augment an (a,b)-tree with the PQA's of Chapter 3 as secondary structures. For completeness I will state the augmentation theorem of [CLRS01, Sec. 14.2] reformulated here for (a,b)-trees.

Theorem 1.1 (Augmenting (a,b)-Trees)**.** *Let f be a field that augments an (a,b)-tree T having n leaf nodes, let $f(u)$ be the value of the field for subtree u and let $s(u)$ be the size of subtree u. Assume that the contents of $f(u)$ for any internal node u can be computed using only the information $f(c_i)$ stored in u's children c_i for $i = 1, \ldots, l$ in $\mathcal{O}(t(s(c_1), \ldots, s(c_l)))$ time, then the augmented (a,b)-tree can be maintained in $\mathcal{O}(t(s(c_1), \ldots, s(c_l)) \log_a n)$ time per update, when $t(s(c_1), \ldots, s(c_l))$ is the time to compute the information $f(u)$ from $f(c_1), \ldots, f(c_l)$.*

[9] In the other case where v is a left sibling of u we simply swap the roles of u and v.

Proof. Since the information stored in $f(u)$ only depends on the information in $f(c_i)$ for $i = 1, \ldots, l$ and can be recomputed in $\mathcal{O}(t(s(c_1), \ldots, s(c_l))$ time, we can update a leaf and then we only need to re-compute the information in the f field on all nodes along the leaf-to-root path, which has length at most $\mathcal{O}(\log_a n)$. □

1.5.2 Priority Queues with Attrition

In 1989 Sundar introduced the *priority queue with attrition* [Sun89] or in short the PQA. He gave three different worst-case implementations where all operations run in $\mathcal{O}(1)$ time. For the third implementation he only sketched the overall details. We give them here for completeness and preparation for Chapter 3 where we extend the structure of this solution to make the PQAs I/O-efficient and support the operation of concatenation.

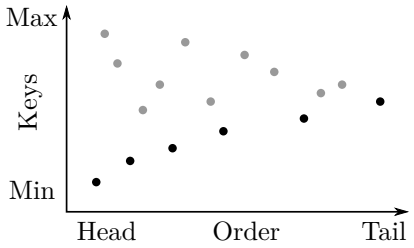

Figure 1.20: Min-PQA.

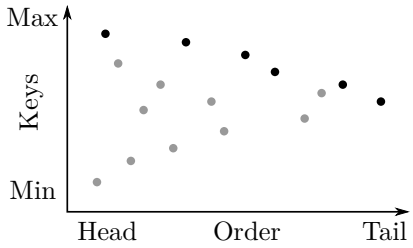

Figure 1.21: Max-PQA.

A priority queue with attrition is a FIFO queue where elements can skip ahead in the queue and kick out other elements. In a Min-PQA the elements e' kicked out are the ones in front of an element e, where $e' \geq e$, these elements e' are said to be *attrited*. See Figure 1.20 where the gray elements are attrited and only the black elements stay. In a Max-PQA the elements e' in front of an element e, $e' \leq e$ are attrited. See Figure 1.21 where the gray elements are again attrited. A PQA[10] Q supports the operations:

- Find-Min(Q) returns $\min(Q)$.
- Delete-Min(Q) returns $\min(Q)$ and removes it from Q. The resulting PQA is $Q' = Q \backslash \{\min(Q)\}$ and the old Q is discarded.
- Insert-and-Attrite(Q, e) inserts e at the end of Q and *attrites* all elements before e that are larger than e. The resulting PQA is $Q' = \{e' \in Q | e' < e\} \cup \{e\}$, the old Q is discarded. The elements $\{e' \in Q | e' \geq e\}$ are attrited.

A PQA consists of $k + 2$ deques of elements; the *clean* deque C, the *buffer* deque B and k *dirty* deques $D_1, \ldots, D_k$. Each deque is sorted in strictly ascending order, and contains no duplicate elements. We have the following invariants for the deques:

I.1) $\max(C) < \min(B) < \min(D_1)$

I.2) $\min(D_1)$ is the minimum element in all the dirty queues $D_1, \ldots, D_k$.

I.3) $|C| \geq \sum_{i=1}^{k} |D_i| + k$

The implementations of the above three operations uses an auxiliary operation Bias which will increase the value $\Delta = |C| - \sum_{i=1}^{k} |D_i| - k$ by one if invariant I.3) is broken.

[10] When we write PQA without stating if it is a Min- or a Max-PQA, it is a Min-PQA.

Find-Min(Q) We just return the element $\min(C)$ which from I.1) and I.2) is the smallest non-attrited element of Q.

Delete-Min(Q) We delete $\min(C)$ from C, if this breaks I.3) then we call Bias(Q) once.

Insert-and-Attrite(Q, e) We have four cases:

1) $e \leq \min(C)$: we delete C, B, $D_1, \ldots, D_k$ and make a new C consisting only of e.

2) $\min(C) < e \leq \max(C)$: we delete B, $D_1, \ldots, D_k$, rename C to B and make a new D_1 which only contains e, put $k = 1$ and then we call Bias(Q) twice.

3) $\max(C) < e < \min(D_1)$: we delete all the dirty queues $D_1, \ldots, D_k$. If furthermore $e \leq \min(B)$ we also delete B. We make a new D_1 which only contains e, put $k = 1$ and call Bias(Q) twice.

4) $\min(D_1) < e$: we add a new dirty queue D_{k+1} only containing e, increase k by one and call Bias(Q) twice.

Bias(Q) We have three cases, in the case where $|B| = 0$ and $k = 0$ we do nothing:

1) $|B| > 0$: we move $\min(B)$ from B onto the end of C, now if $\min(B) \geq \min(D_1)$ we discard B.

2) $|B| = 0$ and $k > 1$: if $\max(D_{k-1}) < \min(D_k)$ we concatenate D_k at the end of D_{k-1} and decrease k by one, else $\max(D_{k-1}) \geq \min(D_k)$ so we delete $\max(D_{k-1})$ from D_{k-1}.

3) $|B| = 0$ and $k = 1$: we concatenate D_1 at the end of C and make k zero.

It is clear that all the operations use $\mathcal{O}(1)$ time. The correctness follows from noticing that we always maintain invariants I.1)–I.3).

1.5.3 Double Exponential Layout

In the implicit model we cannot change the content of the input words or create new words, so we cannot build a data structure using numbers and pointers. But there is a trick that is often used to avoid this. Using two consecutive and distinct elements x and y in memory, we can *pair encode* a 0 bit by putting the smallest of x and y first and the largest second, similarly we can encode a 1 bit by placing the largest first and the smallest second. This way we can encode a pointer or word of $\lceil \log n \rceil$ bits using $2\lceil \log n \rceil$ elements.

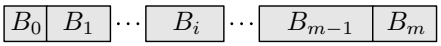

Figure 1.22: Double exponential layout of implicit data structures.

A problem arises when our data structure changes size from n to n' and we have that $\lceil \log n \rceil \neq \lceil \log n' \rceil$. Then we might need to re-encode all pointers and words which we would like to avoid. Frederickson solved this problem by use of the *double exponential layout* [Fre84]. If the data structure in question supports inserts, deletes and searches, then we distribute our n input elements over $m = \lceil \log \log n \rceil$ blocks, where in each block we build our data structure over the elements in that block. Blocks

$i = 0, \ldots, m-1$ have sizes $|B_i| = 2^{2^i}$ and block m has size $|B_m| = n - \sum_{i=0}^{m-1} |B_i|$ see Figure 1.22.

To make an insert of an element, we just insert it in block B_m. To search for an element, we search in each block B_i for $i = 0, \ldots, m$, if we find the element in any of these blocks we report it. Finally, when we want to delete an element we first find it, say it is in block B_i, then we exchange it with an arbitrary element from B_m and then delete the element from B_m.

This way for any block B_i there is an implicit n_i attached to that block which is $n_i = 2^{2^i}$, and we can then represent pointers or numbers within block B_i in pair encoding using $2 \cdot 2^i$ elements.

If an operation of a data structure of size n' uses $\mathcal{O}(\log^c n')$ time for $c = \mathcal{O}(1)$, then the total time to perform that operation on all blocks is at most

$$\sum_{i=0}^{m} \log^c 2^{2^i} = \sum_{i=0}^{m} \left(2^i\right)^c = \mathcal{O}(2^{cm}) = \mathcal{O}\left((2^m)^c\right) = \mathcal{O}(\log^c n)$$

So there is only a constant factor slow-down for operations taking polylogarithmic time, using the double-exponential layout.

1.5.4 Functional Pointer Machine Trick

In the FPM model all data structures are confluently persistent automatically because of the properties of the model. This property enables us to do space savings when augmenting say an (a, b)-tree. Assume that the secondary structure that we augment the (a, b)-tree with uses $\mathcal{O}(t(s(c_1), \ldots, s(c_l)))$ time to compute the augmentation field $f(u)$ for a node u with children $c_1, \ldots, c_l$. Then the extra space usage for augmenting node u is at most $\mathcal{O}(t(s(c_1), \ldots, s(c_l)))$ no matter how large the secondary structure attached to node u actually is. In other words, we only record the changes required to combine the structures of $c_1, \ldots, c_l$ into the structure for u, and this saves us space.

Chapter 2

Implicit Working-Set Dictionaries

In this chapter we present a Cache-Oblivious Implicit Working-Set Dictionary, presented in Section 2.2, this is the first dictionary to attain the working set bound in the implicit model. To obtain our result we also develop a Cache-Oblivious Implicit Moveable Dictionary, in Section 2.1, which will make the memory management of the working-set dictionary easier.

Our moveable dictionary will combine a constant number of previously known non-moveable dictionaries [FG03] into a moveable structure. Our construction only gives a constant worst-case access overhead and preserves the cache-oblivious and implicit properties of [FG03].

Our working-set dictionary will use the moveable dictionary as a black box to ease the memory management. Our working-set dictionary implicitly partitions the elements into $\Theta(\log \log n)$ levels and maintains lower bounds on their working-set numbers. This is enough to give the working-set bound for the search operation, but not for the predecessor and successor operations where we need the idea of intervals. We furthermore partition the key space into intervals, which are spread across the levels. We use these intervals and the lower bounds for each level to support the predecessor and successor operations within the working-set bound.

2.1 Moveable Dictionaries

In this section we describe an implicit moveable dictionary which can be laid out in an array in the range $[i, j]$. We can delete in the left or the right end of the array, such that the structure then lies in the range $[i+1, j]$ or $[i, j-1]$, respectively. Likewise we can insert in the left or right end such that the structure then lies in the range $[i-1, j]$ or $[i, j+1]$, respectively. The structure supports search, predecessor and successor operations. All operations runs in $\mathcal{O}(\log n)$ time and $\mathcal{O}(\log_B n)$ cache-misses where $n = j - i + 1$. The moveable dictionary will be implicit except that it stores $\mathcal{O}(\log n)$ extra bits, this requirement is removed in Subsection 2.1.6.

The amortized solution is easy to obtain. We use two of the dictionaries by Franceschini and Grossi [FG03], from here and onwards we will call such a dictionary a FG dictionary. One dictionary called R located in the range $[r, j]$ growing to the right as normal, and one called L located in the range $[i, r-1]$ growing to the left, that is we have inverted all indices of the original FG dictionary. The Insert-Left operation inserts its element into L, and the Insert-Right operation, inserts its element into R. Analogously for Delete-Left and Delete-Right. In the special case where L or R are

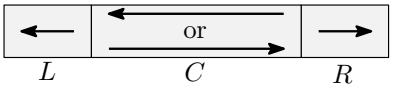

Figure 2.1: We have three FG dictionaries L, C and R, where L will always grow/shrink in the left direction, and R will always grow/shrink in the right direction, and C will change direction during the execution of the jobs to shrink or grow L or R.

empty we rebuild the data structure such that L and R have the same size. To search for an element we search in L and then in R, to find the predecessor/successor of an element we find the predecessor/successor in L and R and return the largest/smallest of the two. All operations run in $\mathcal{O}(\log n)$ amortized time and $\mathcal{O}(\log_B n)$ amortized cache-misses using the potential function $\Phi = |\,|L| - |R|\,|$.

We can de-amortize the construction using incremental rebalancing, and placing an additional FG dictionary C between L and R, see Figure 2.1. The moveable dictionary will support the following operations:

- Insert-Left/Right(e): inserts an element e into the dictionary which grows in the left/right side, respectively.
- Delete-Left/Right(x): deletes the element with key x from the dictionary which shrinks in the left/right side, respectively.
- Search(x): searches for the element e with key x in the dictionary, and returns e if it exists and *nil* otherwise.
- Predecessor(x): finds the predecessor of element x in the dictionary and returns its position, i.e., the position of element $\max\{e' \in P \cup \{-\infty\} \mid e' < e\}$ is returned.
- Successor(x): finds the successor of element x in the dictionary and returns its position, i.e., the position of element $\min\{e' \in P \cup \{\infty\} \mid e < e'\}$ is returned.

We have minimum and maximum invariants on the sizes of L and R. If L or R are close to violating these invariants, say L is getting too big, we start a job to make L smaller. This job is executed incrementally, so each time an Insert/Delete-Left/Right, Search, Predecessor or Successor operation is executed we perform a constant number of steps of the current job. We ensure that there is only one job running at a time, we do this by adding the remaining jobs to a queue. We also ensure that all jobs complete before their deadlines. A job reaches its deadline if one of L or R breaks their size invariants. A job, say for shrinking L, has a deadline which is the time when L grows larger than we allow it to in the invariants. During the execution of a job we have an extra dictionary, which can be one of either L', C' or R', depending on how far we are in the execution of the current job, this will be further clarified in Subsection 2.1.2.

2.1.1 Invariants

We maintain the following invariants for the moveable dictionary:

I.1 $\frac{1}{24}n \leq |L| + |L'|, |R| + |R'| \leq \frac{9}{24}n$.

I.2 The job queue has size at most 2.

I.3 All jobs finish before their deadline

In Subsection 2.1.3 we use these invariants to prove correctness of our dictionary. We observe that from I.1 and the identity $n = |L| + |L'| + |C| + |C'| + |R| + |R'|$ we get $\frac{6}{24}n \leq |C| + |C'| \leq \frac{22}{24}n$.

2.1.2 Operations and Jobs

We will now describe the search and update operations of the moveable dictionary. The Insert- and Delete-left operations and Grow- and Shrink-Left jobs described here have analogous right-versions.

Search(x)

We always have the structures L, C and R, and possibly one of the structures L', C' or R'. So we have at most 4 structures and we search them all, if we find the element we search for we return a pointer to it, otherwise we return *nil*.

Predecessor/Successor(x)

As in Search we search for the predecessor/successor in L, C, R and possibly one of L', C' or R', then we return the largest/smallest of the 4 candidates.

Insert-Left(e)

We insert e into L, then if $|L| \geq \frac{7}{24}n$ we enqueue a Shrink-Left job unless one is already in the queue.

Delete-Left(x)

We delete the element with key x from L. We can do this even though the element we want to delete resides in L', C, C', R or R' by swapping the element we want to delete with one from L. We can swap elements by two deletions and two insertions. Then if $|L| \leq \frac{3}{24}n$ we enqueue a Grow-Left job unless one is already in the queue.

Grow-Left

The Grow-Left job performs the following steps, see Figure 2.2:

1) If C is not growing to the left then turn C around so it grows toward L. We turn C around by creating a new C' at the growing end of C which grows towards C, into which we insert all the elements of C, one element at a time.

2) Construct L' at the beginning of L, growing to the right, of size $\frac{2}{24}n$, by taking elements from C.

3) Turn L' around so it faces L, like we turned C around in 1).

4) Continue taking an element from C putting it into L' so it expands into L. The element overridden in L is moved into the empty place in C where we took the element to place in L'. We do this by splitting L into two pieces by address-mapping, see 3) and 4) in Figure 2.2. When we have moved L completely to the right of L', then swap the names of L and L'.

5) Merge L' back into C, by deleting an element from L' and inserting it into C until L' is empty.

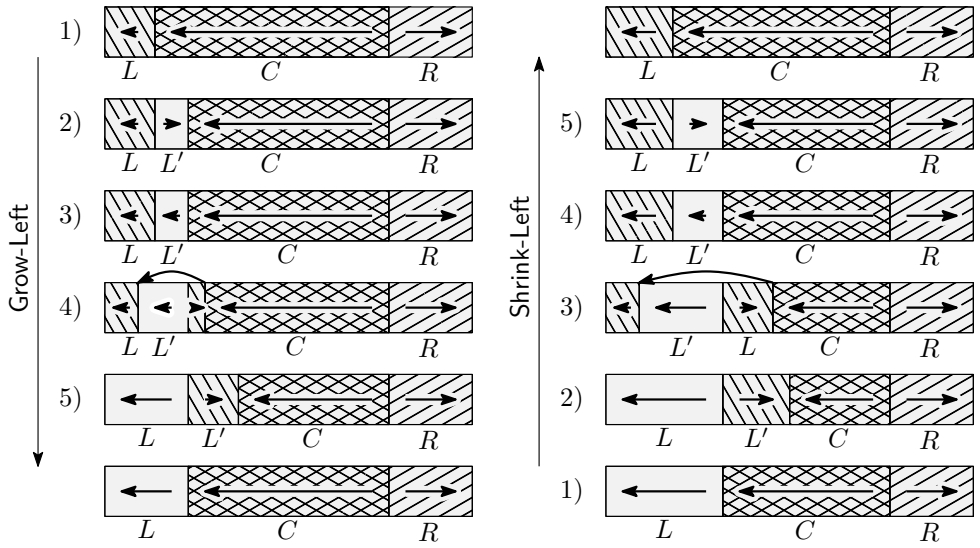

Figure 2.2: The steps of the two jobs Grow- and Shrink-Left, notice that they are almost each other's inverse. (Left) The five steps of the Grow-Left job are shown, notice that in 4) the arrow means that we have split L up into two by use of address-mapping. (Right) The five steps of the Shrink-Left job are shown, in step 3) we have again used address-mapping to split L in two.

Shrink-Left

The Shrink-Left job consists of the following steps, see Figure 2.2 and notice the similarity to the Grow-Left job. The Shrink-Left job is the "inverse" job of Grow-Left:

1) If C is not growing to the left then turn C around so it grows toward L.

2) Create L' by deleting $\frac{5}{24}n$ elements from C, one element at a time and inserting them into L', which we create to the left of C.

3) Swap the names of L and L'. Delete an element from L' and insert it into C so it expands into L, then move the element overridden in L to the empty space to the left of L', do this one element at a time until L is moved completely to the left of L'.

4) Turn L' around so it faces C.

5) Merge L' back into C.

2.1.3 Correctness

The correctness of the Search and Predecessor/Successor operations are clear as we do a predecessor/successor operation on each of the FG dictionaries L, L', C, C', R and R', that exists, and return the resulting/largest/smallest element found, respectively. Similarly granted that L and R exists and contains elements, the Delete-Left/Right and Insert-Left/Right operations are correct. The only thing we need to prove is that the invariants I.1–I.3 are maintained at all times. Before we prove that I.1–I.3 are maintained, we will first prove some simple observations, and then use them to prove that the invariants are maintained.

For a job j we define $n_{\text{init}}(j)$ to be the value of n when j is initialized, $n_0(j)$ to be the value of n when j starts running and $n_{\text{finish}}(j)$ the value of n when j finished. We will also let $0 < \beta \leq 1$ be a constant which we will give an exact value later.

Observation 2.1. *If we perform $\frac{5(1+\beta)}{\beta}$ primitive steps of a job j at each update, then the job will finish within $\beta n_0(j)$ updates.*

Proof. A primitive step is the movement of one element from one dictionary to another. We want the job to finish within $\beta n_0(j)$ updates, from this bound we know that during the execution of a job at most $\beta n_0(j)$ updates can occur, hence the dictionary has size at most $(1+\beta)n_0(j)$. Each of the five steps in a grow or shrink job takes at most $(1+\beta)n_0(j)$ primitive steps, and so the whole job finishes in at most $5(1+\beta)n_0(j)$ primitive steps. If we perform $\frac{5(1+\beta)}{\beta}$ primitive steps per update, then the job finishes within $\beta n_0(j)$ updates. □

Observation 2.2. *Any job j under execution will finish within the next βn updates.*

Proof. If the job in question has been running for d updates then from Observation 2.1 it needs to run for an additional

$$\beta n_0(j) - d \overset{(*)}{\leq} \beta(n_0(j) - d) \overset{(**)}{\leq} \beta n$$

updates to finish, where we in $(*)$ used the fact that $\beta \leq 1$ and in $(**)$ we used the fact that d updates to a dictionary of size $n_0(j)$ will result in a dictionary of size $n \in [n_0(j) - d, n_0(j) + d]$. □

Using Observations 2.1–2.2 we will now prove that I.1–I.3 are always true, which will prove the correctness of the moveable dictionary.

Proof of I.2

We prove that the job queue has size at most 2. If a Grow-Left job is enqueue there needs to be performed at least $\frac{7}{24}n - \frac{3}{24}n = \frac{4}{24}n$ Insert-Left updates before a Shrink-Left job is required, and vice-versa. Likewise if a Shrink-Right job is enqueued there needs to be performed at least $\frac{4}{24}n$ Delete-Right updates before a Grow-Right job is required. So if we ensure that a left and a right job can be performed within $\frac{4}{24}n$ updates, then we will never have more than two jobs queued at a time. From Observation 2.2 it takes at most $\beta n + \beta(n + n\beta)$ updates to run two jobs. So we make the constraint that $\beta n + \beta(n + n\beta) \leq \frac{4}{24}n$ which means that we require $\beta^2 + 2\beta \leq \frac{4}{24}$.

Proof of I.1

We prove that $\frac{1}{24}n \leq |L| + |L'|, |R| + |R'| \leq \frac{9}{24}n$. If no jobs are queued or running then we have from the checks in insert and delete that $\frac{1}{24}n \leq \frac{3}{24}n \leq |L| \leq \frac{7}{24}n \leq \frac{9}{24}n$ (this proof is completely symmetric for R).

Now say $\frac{7}{24}n + d = |L|$ for $d \geq 1$ then we know that a Shrink-Left job is either running or queued. Let j_1 be the job that is currently running and j_2 the queued Shrink-Left job. Then $t_{\text{finish}}(j_2) \leq \beta n - d + \beta(n + \beta n)$ as job j_1 needs to run for at most $\beta n - d$ updates more to finish and j_2 then needs to run for at most $\beta(n + \beta n)$ updates more to finish. We also have that $n_{\text{init}}(j_2) - d \leq n \leq n_{\text{init}}(j_2) + d$ as it is

only d updates since j_2 was created. When j_2 is finished we have

$$\begin{aligned} |L| &\leq \frac{5}{24}n_{\text{init}}(j_2) + t_{\text{finish}}(j_2) \\ &\leq \frac{5}{24}n_{\text{init}}(j_2) + (\beta^2+2\beta)n - d \\ &\leq \frac{5}{24}(n+d) + (\beta^2+2\beta)n - d \\ &\leq \frac{5}{24}n + d + (\beta^2+2\beta)n - d \\ &\overset{(*)}{\leq} \frac{6}{24}n \end{aligned}$$

where we in $(*)$ made the constraint that $\beta^2+2\beta \leq \frac{1}{24}$. We also have $d \leq (\beta^2+2\beta)n \leq \frac{1}{24}n$. So at all times we have that $|L| \leq \frac{8}{24}n \leq \frac{9}{24}n$.

Now say $\frac{3}{24}n = |L| + d$ for $d \geq 1$ then we know that a Grow-Left job is either running or queued. Let j_1 be the job that is currently running and j_2 the queued Grow-Left job. Then $t_{\text{finish}}(j_2) \leq \beta n - d + \beta(n+\beta n)$ as job j_1 needs to run for at most $\beta n - d$ updates more to finish and j_2 then needs to run for at most $\beta(n+\beta n)$ updates more to finish. We also have that $n_{\text{init}}(j_2) - d \leq n \leq n_{\text{init}}(j_2) + d$ as it is only d updates since j_2 was created. When j_2 is finished we have

$$\begin{aligned} |L| &\geq \frac{5}{24}n_{\text{init}}(j_2) - t_{\text{finish}}(j_2) \\ &\geq \frac{5}{24}n_{\text{init}}(j_2) - (\beta^2+2\beta)n + d \\ &\geq \frac{5}{24}(n-d) - (\beta^2+2\beta)n + d \\ &\geq \frac{5}{24}n - d - (\beta^2+2\beta)n + d \\ &\overset{(*)}{\geq} \frac{4}{24}n \end{aligned}$$

where we in $(*)$ again made the constraint that $\beta^2 + 2\beta \leq \frac{1}{24}$. We again have $d \leq (\beta^2+2\beta)n \leq \frac{1}{24}n$. So at all times we have that $|L| \geq \frac{2}{24}n \geq \frac{1}{24}n$.

Proof of I.3

We prove that all jobs finish before their deadlines. When a job j_2 is created, then there might be at most one other job j_1 which is currently running. From Observation 2.2 we know that both jobs finish in at most $\beta n+\beta(n+\beta n)$ updates, and j_2's deadline occurs after $\frac{9-7}{24}n = \frac{3-1}{24}n$ updates. So if we require that $\beta n + \beta(n+\beta n) \leq \frac{2}{24}n$, which means that $\beta^2 + 2\beta \leq \frac{2}{24}$ then all jobs will finish before their deadline.

We have proved all invariants I.1–I.3 under the condition that $\beta^2 + 2\beta \leq \frac{1}{24}$. If we chose $\beta = \frac{1}{49}$ this constraint is satisfied and from Observation 2.1 we need to perform $\frac{5(1+\beta)}{\beta} = 250$ steps per operation on the moveable dictionary to maintain I.1–I.3.

2.1.4 Running Time

All we require of the black box structure we use, that is the FG dictionary, is that it has an insert, delete, search, predecessor and successor operation, so lets assume these use time $i(n)$, $d(n)$, $s(n)$, $pred(n)$ and $succ(n)$, respectively. Then all update operations on our structure run in $\mathcal{O}(i(n)+d(n))$ time, as we use the update

operations on the black box structure a constant number of times for each of our update operations. Using the FG dictionary from [FG03] we get a time bound of $\mathcal{O}(\log n)$ for all our update operations, and the time for Search, Predecessor and Successor are also $\mathcal{O}(s(n))$, $\mathcal{O}(pred(n))$ and $\mathcal{O}(succ(n))$, respectively.

2.1.5 Cache-Oblivious

We will now show that the implicit moveable dictionary as presented in the previous sections is in fact cache-oblivious and has a worst-case running time of $\mathcal{O}(\log_B n)$ cache-misses per operation. We will now go through each operation and look at it from a cache-oblivious viewpoint.

All operations: Search, Predecessor, Insert-Left, Delete-Left, Insert-Right and Delete-Right only use a constant number of calls to operations on the FG dictionary, and as these only incur $\mathcal{O}(\log_B n)$ cache-misses they have the same bound for our moveable dictionary. So we only need to take a closer look at the jobs Grow-Left and Shrink-Left (their right versions are analogously so we ignore these).

Grow-Left and Shrink-Left

The Grow-Left job consists of: 1) we turn C around and we simply do this by removing the right-most element and making a new FG dictionary C' containing it, the rest of the elements in C we also put into C' which grows to the left. As we simply perform a constant number of the FG operations which incurs $\mathcal{O}(\log_B n)$ cache misses per step, so does 1). In 2) we just take out some elements of C, which we call L', and 3) turns L' around like we did with C in 1), so this also incurs $\mathcal{O}(\log_B n)$ cache-misses per step. In 4) we split the FG dictionary L into two parts, but this only incurs twice as many cache-misses when we perform Search, Predecessor and Successor operations running on L while L is split in two (it is the same when we split C or R). Here each step also incurs $\mathcal{O}(\log_B n)$ cache-misses. Finally in 5) we just take all elements of L and insert into C, which again only incurs $\mathcal{O}(\log_B n)$ cache-misses per step.

So all in all the Grow-Left job incurs $\mathcal{O}(\log_B n)$ cache-misses per step, and as the Shrink-Left job is just the Grow-Left job run backwards (seen from a cache-oblivious viewpoint) it will also just incur $\mathcal{O}(\log_B n)$ cache-misses per step.

2.1.6 Making the Dictionary Implicit

We now remove the requirement of the $\mathcal{O}(\log n)$ extra bits of memory between operations, which we used to remember sizes, directions, starting addresses and address-splits of the used FG dictionaries. Doing this makes our moveable dictionary implicit and hence it does not use any extra space between operations. In the moveable dictionary we need to store the size and starting positions of L, C and R, along with the direction of C, and furthermore when a job is running we also need to store the size, starting position and direction of one of L', C' or R' along with the memory-mapping split we make of L or L' (R and R' in the right versions). We can easily pair encode this in $\mathcal{O}(\log n)$ pairs of elements. We will call this list of elements D and we will store them just to the left of L.

Now depending on which of the operations Insert-Left, Delete-Left, Insert-Right and Delete-Right we should perform we have to threat D differently. Lets say that we need to store c words in D then we need $|D| \geq 2c\lceil \log n \rceil$, but we instead put $|D| = \lceil 2c(\log(n) + 1) \rceil \geq 2c\lceil \log n \rceil$ as this will make $|D|$ make smaller jumps in size. Now for each operation we do:

- Insert-Left(e): If $\lceil 2c(\log(n+1)+1)\rceil > \lceil 2c(\log(n)+1)\rceil$ then we insert e into D so it grows to the left, and re-encode D. Else we move the rightmost element of D to the left side and re-encode D, and then we insert e into L as normal.
- Delete-Left(x): If $\lceil 2c(\log(n-1)+1)\rceil < \lceil 2c(\log(n)+1)\rceil$ then we just delete the element with key x that we want to delete[1] from D, so it shrinks to the left. Else we delete the element with key x from L as normal and move the leftmost element of D to the right of D and re-encode D.
- Insert-Right(e): If $\lceil 2c(\log(n+1)+1)\rceil > \lceil 2c(\log(n)+1)\rceil$ then we delete a element g from L and insert it into D and re-encode D. In both cases we then insert e into R like normal.
- Delete-Right(x): If $\lceil 2c(\log(n-1)+1)\rceil < \lceil 2c(\log(n)+1)\rceil$ then we delete a element g from D and insert it into L and re-encode D. In both cases we then delete the element with key x from R as normal.

When performing Search, Predecessor or Successor operations, we also scan through D to determine if the element which we are looking for is located in D.

Notice that having D will only incur an additive overhead of $\mathcal{O}(\log n)$ time for any operation and $\mathcal{O}(\frac{\log n}{B}) = \mathcal{O}(\log_B n)$ cache-misses, so all operations on the moveable dictionary will still take $\mathcal{O}(\log n)$ time and $\mathcal{O}(\log_B n)$ cache-misses.

2.2 Working-Set Dictionaries

In this section we will construct our implicit working-set dictionary using the moveable dictionary from Section 2.1 as a black box. Our dictionary will support the following operations:

- Search(e) determines if e is in the dictionary, if so its working-set number is set to 0.
- Predecessor(e) will find $\max\{e' \in P \cup \{-\infty\} \mid e' < e\}$, without changing any working-set numbers.
- Successor(e) will find $\min\{e' \in P \cup \{\infty\} \mid e < e'\}$, without changing any working-set numbers.
- Insert(e) inserts e into the dictionary with a working-set number of 0, all other working-set numbers are increased by one.
- Delete(e) deletes e from the dictionary, and does not change the working-set number of any element.

Our dictionary will maintain lower bounds on the working set numbers of each element, this will create a partition of the elements into $\Theta(\log\log n)$ levels. These levels are enough to guarantee the working set bound for the Search operation, but not for the Predecessor nor Successor operations. To get the working set bound for these operations we will introduce another idea, we will furthermore partition the key space into disjoint intervals that span the whole key space. Each interval will then be placed within exactly one level, and will give use a termination criteria when doing predecessor and successor searches.

Our dictionary will furthermore be cache-oblivious due to our use of the moveable dictionary as a black box. The dictionary supports the operations Insert and

[1] We assume that the element with key x we want to delete lies in D, if this is not true we can make it so by swapping the element with key x with some element from D.

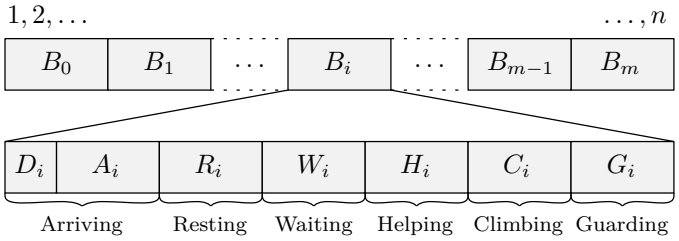

Figure 2.3: Overview of how the working set dictionary is laid out in memory. The dictionary grows and shrinks to the right when elements are inserted and deleted.

Delete in time $\mathcal{O}(\log n)$ and $\mathcal{O}(\log_B n)$ cache-misses, Search, Predecessor and Successor in time $\mathcal{O}(\log \min(\ell_{\mathsf{p}(e)}, \ell_e, \ell_{\mathsf{s}(e)}))$, $\mathcal{O}(\log \ell_{\mathsf{p}(e)})$ and $\mathcal{O}(\log \ell_{\mathsf{s}(e)})$, and cache-misses $\mathcal{O}(\log_B \min(\ell_{\mathsf{p}(e)}, \ell_e, \ell_{\mathsf{s}(e)}))$, $\mathcal{O}(\log_B \ell_{\mathsf{p}(e)})$ and $\mathcal{O}(\log_B \ell_{\mathsf{s}(e)})$, respectively, where $\mathsf{p}(e)$ and $\mathsf{s}(e)$ are the predecessor and successor of e, respectively.

2.2.1 Structure

Our structure consists of $m = \Theta(\log \log n)$ blocks $B_0, \ldots, B_m$, each block B_i is of size $\mathcal{O}(2^{2^{i+k}})$, where k is a constant. Elements in B_i have a working-set number of at least $2^{2^{i+k-1}}$. The block B_i consists of an array D_i of $w_i = d \cdot 2^{i+k}$ elements, where d is a constant, and moveable dictionaries A_i, R_i, W_i, H_i, C_i and G_i, for $i = 0, \ldots, m-1$, see Figure 2.3. For block B_m we only have D_m if $|B_m \backslash \{\min(P), \max(P)\}| \leq w_m$, otherwise we have the same structures as for the other blocks. We use the block D_i to encode the sizes of the movable dictionaries A_i, R_i, W_i, H_i, C_i and G_i so that we can locate them. Discussion of further details of the memory layout is postponed to Subsection 2.2.5.

We call elements in the structures D_i and A_i for *arriving* points, and when making a non-arriving point arriving, we will put it into A_i unless specified otherwise. We call elements in R_i for *resting* points, elements in W_i for *waiting* points, elements in H_i for *helping* points, elements in C_i for *climbing* points and elements in G_i for *guarding* points.

Crucial to our data structure is the partitioning of $[\min(P), \max(P)]$ into *intervals*. Each interval is assigned to a *level* and level i corresponds to block B_i. Consider an interval lying at level i. The endpoints e_1 and e_2 will be guarding points stored at level $0, \ldots, i$. All points inside of this interval will lie in level i and cannot be guarding points, i.e., $]e_1, e_2[\cap (\bigcup_{j \neq i} B_j \cup G_i) = \emptyset$. We do not allow intervals defined by two consecutive guarding points to be empty, they must contain at least one non-guarding point. We also require $\min(P)$ and $\max(P)$ to be guarding points in G_0 at level 0, but they are special as they do not define intervals to their left and right, respectively. A query considers $B_0, B_1, \ldots$ until B_i where the query is found to be in a level i interval where the answer is guaranteed to have been found in blocks $B_0, \ldots, B_i$.

The basic idea of our construction is the following. When searching for an element, it is moved to level 0. This can cause block overflows (see invariants I.5–I.9 in Subsection 2.2.3), which are handled as follows. The arriving points in level i have just entered from level $i-1$, and when there are $2^{2^{i+k}}$ of them in A_i they become resting. The resting points need to charge up their working-set number before they can begin their journey to level $i+1$. They are charged up when $2^{2^{i+k}}$ further arriving points have come to level i, then the resting points become waiting points. Waiting points have a high enough working-set number to begin the journey to level $i+1$,

Figure 2.4: The structure of the levels for a dictionary. The levels are indicated to the left.

but they need to wait for enough points to group up close together in key space so that they can start the journey. When a waiting point is picked to start its journey to level $i+1$ it becomes a helping or climbing point, and every time enough helping points have grouped up, i.e., there are at least $c = 5$ consecutive of them, then they become climbing points and are ready to go to level $i+1$. The climbing points will then incrementally be going to level $i+1$. See Figure 2.4 for an example of the structure of the intervals.

2.2.2 Notation

Before we introduce the invariants we need to define some notation. For a subset $S \subseteq P$, we define $\mathsf{p}_S(e) = \max\{p \in S \cup \{-\infty\} \mid p < e\}$ and $\mathsf{s}_S(e) = \min\{s \in S \cup \{\infty\} \mid e < s\}$. When we write $S_{\leq i}$ we mean $\bigcup_{j=0}^{i} S_j$ where $S_j \subseteq P$ for $j = 0, \ldots, i$.

For $S \subseteq P$, we define $\mathsf{GIL}_S(e) = S \cap]\mathsf{p}_{P\setminus S}(e), e[$ to be the Group of Immediate Left points of e in S which does not have any other point of $P\setminus S$ in between them, see Figure 2.5. Similarly we define $\mathsf{GIR}_S(e) = S \cap]e, \mathsf{s}_{P\setminus S}(e)[$ to the right of e. We will notice that we will never find all points of $\mathsf{GIL}_S(e)$ unless $|\mathsf{GIL}_S(e)| < c$, the same applies for $\mathsf{GIR}_S(e)$. For $S \subseteq P$, we define $\mathsf{FGL}_S(e) = S \cap]\mathsf{p}_{P\setminus S}(\mathsf{p}_S(e)), \mathsf{p}_S(e)]$ to be the First Group of points from S Left of e, i.e., the group does not have any points of $P\setminus S$ in between its points, see Figure 2.5. Similarly we define $\mathsf{FGR}_S(e) = S \cap [\mathsf{s}_S(e), \mathsf{s}_{P\setminus S}(\mathsf{s}_S(e))[$. We will notice that we will never find all points of $\mathsf{FGL}_S(e)$ unless $|\mathsf{FGL}_S(e)| < c$, the same applies for $\mathsf{FGR}_S(e)$.

We will sometimes use the phrasings *a group of points* or *e's group of points*. This refers to a group of points of the same type, i.e., arriving, resting, etc., and with no other types of points in between them. Later we will need to move elements around between the structures D_i, A_i, R_i, W_i, H_i, C_i and G_i. For this we have the notation $X \xrightarrow{h} Y$, meaning that we move h arbitrary points from X into Y, where X and Y can be one of D_i, A_i, R_i, W_i, H_i, C_i and G_i for any i.

When we describe the intervals we let $]a, b]$ be an interval from a to b that is

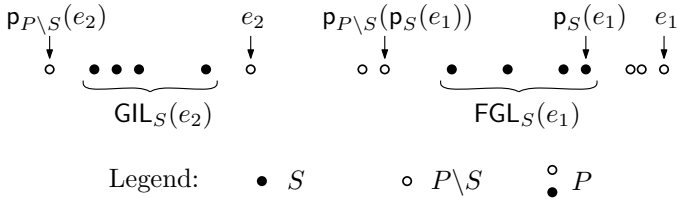

Figure 2.5: Here is an illustration of GIL and FGL. Notice that $\mathsf{GIL}_S(e_1) = \emptyset$ whereas $\mathsf{FGL}_S(e_1) \neq \emptyset$.

open at a and closed at b. We let (a, b) be an interval from a to b that can be open or closed at a and b. We use this notation when we do not care if a and b are open or closed. In the operations updating the intervals we will sometimes branch depending on which type an interval is. For clarity we will explain how to determine this given the level i of the interval and its two endpoints e_1 and e_2. The interval (e_1, e_2) is of type $[e_1, e_2)$ if $e_1 \in G_i$, else $e_1 \in G_{\leq i-1}$ and the interval is of type $]e_1, e_2)$. This is symmetric for the other endpoint e_2.

2.2.3 Invariants

We will now define the invariants which will help us define and prove correctness of our interface operations: Insert(e), Delete(e), Search(e), Predecessor(e) and Successor(e). We maintain the following invariants which uniquely determine the intervals[2]:

I.1 A guarding point is part of the definition of at most two intervals[3], one to its left, at level i and one to its right at level j, where $i \neq j$. The guarding point e lies at level $\min(i, j)$. The interval at level $\min(i, j)$ is closed at e, and the interval at level $\max(i, j)$ is open at e. We also require that $\min(P)$ and $\max(P)$ are guarding points stored in G_0, but they do not define an interval to their left and right, respectively, and the intervals they help define are open in the end they define. A non-guarding point intersecting an interval at level i, lies in level i. Each interval contains at least one non-guarding point. The union of all intervals gives $]\min(P), \max(P)[$.

I.2 Any climbing point, which lies in an interval with other non-climbing points, is part of a group of at least c points. In intervals of type $[e_1, e_2]$ which only contain climbing points, we allow there to be less than c of them.

I.3 Any helping point is part of a group of size at most $c - 1$. A helping point cannot have a climbing point as a predecessor or successor. An interval of type $[e_1, e_2]$ cannot contain only helping points.

We maintain the following invariants for the working-set numbers:

I.4 Each arriving point in D_i and A_i has a working set value of at least $2^{2^{i-1+k}}$, arriving points in D_0 and A_0 have a working-set value of at least 0. Each resting point in R_i will have a working-set value of at least $2^{2^{i-1+k}} + |A_i|$, resting points in R_0 have a working-set value of at least $|A_0|$. Each waiting, helping or climbing point in W_i, H_i and C_i, respectively, will have a working-set value of at least $2^{2^{i+k}}$. Each guarding point in G_i, who's left interval lies at level i and right interval lies at level j, has a working set value of at least $2^{2^{\max(i,j)-1+k}}$.

We maintain the following invariants for the size of each block and their components:

I.5 $|D_0| = \min(|B_0 \backslash \{\min(P), \max(P)\}|, w_0)$ and $|D_i| = \min(|B_i|, w_i)$ for $i = 1, \ldots, m$.

I.6 $|R_i| \leq 2^{2^{i+k}}$ for $i = 0, \ldots, m$ and $|W_i| + |H_i| + |C_i| \neq 0 \Rightarrow |R_i| = 2^{2^{i+k}}$ for $i = 0, \ldots, m$.

I.7 $|A_i| + |W_i| = 2^{2^{i+k}}$ for $i = 0, \ldots, m-1$, and $|A_m| + |W_m| \leq 2^{2^{m+k}}$.

[2]We assume that $|P| = n \geq 2$ at all times if this is not the case we only store G_0 which contains a single element and we ignore all invariants.

[3]Only the smallest and largest guarding points will not participate in the definition of two intervals, all other guarding points will.

I.8 $|A_i| < 2^{2^{i+k}}$ for $i = 0, \ldots, m$.

I.9 $|H_i| + |C_i| = 4c2^{2^{i+k}} + c_i$, where $c_i \in [-c, c]$, for $i = 0, \ldots, m-1$.

From the above invariants we have the following observation:

O.1 From I.1 all points in G_i are endpoints of intervals in level i, and each interval has at most two endpoints. Hence for $i = 0, \ldots, m$ we have that

$$|G_i| \leq 2(|D_i| + |A_i| + |R_i| + |W_i| + |H_i| + |C_i|) \overset{(*)}{\leq} (4 + 2d + 8c) \cdot 2^{2^{i+k}} + 2c \,,$$

where we in $(*)$ we have used I.5, I.6, I.7 and I.9.

From I.1 we have the following lemma.

Lemma 2.1. *Let e be an element, $e_1 = \mathsf{p}_{G_{\leq i}}(e)$, $e_2 = \mathsf{s}_{G_{\leq i}}(e)$, $I(e_1, e_2, i) =]e_1, e_2[\cap \bigcup_{j=0}^{i} B_j$ and let i be the smallest integer for which $I(e_1, e_2, i) \neq \emptyset$, then*

1. *(e_1, e_2) is an interval at level i if e is non-guarding and*
2. *(e_1, e) or (e, e_2) is an interval at level i if e is guarding.*

Proof. Assume that i is the minimal i that fulfills $I(e_1, e_2, i) \neq \emptyset$, where $e_1 = \mathsf{p}_{G_{\leq i}}(e)$ and $e_2 = \mathsf{s}_{G_{\leq i}}(e)$. We will have two cases depending on if e is guarding or not.

Lets first handle case 2) where e is guarding and hence in the dictionary: Since e is in the dictionary and $e_1 < e < e_2$ we have from the minimality of i that e lies in level i, and from I.1 e is then part of an interval lying in level i either to the left or to the right. Say e is part of an interval to the left i.e., the interval (e_1', e). If $e_1 < e_1'$ then this would contradict that $e_1 = \mathsf{p}_{G_{\leq i}}(e)$ hence $e_1' \leq e_1$, but since e_1' defines the interval (e_1', e) left of e then e_1' is also the predecessor of e and we have that $e_1' = e_1$. So we know that (e_1, e) defines an interval at level i. The argument for (e, e_2) is symmetric.

In the case 1) e is non-guarding and e may lie in the dictionary or not: Since $e_1 < e < e_2$ we have from the minimality of i that e lies in level i, hence from I.1 we have that the interval (e_1, e_2) lies at level i. □

2.2.4 Operations

We will briefly give an overview of the helper operations and state their requirements *(denoted as R:)* and guarantees *(denoted as G:)*, then we will describe the helper and interface operations in details. Search(e) uses the helper operations as follows: when a search for element e is performed then the level i where e lies, is found using Find, then e and $\mathcal{O}(1)$ of its surrounding elements are moved into level 0 by use of Move-Down while maintaining I.1–I.4. Calls to Fix for the levels we have altered will ensure that I.5–I.8 will be maintained, finally a call to Rebalance-Below$(i-1)$ will ensure that I.9 is maintained by use of Shift-Up(j) which will take climbing points from level j and make them arriving in level $j+1$ for $j = 0, \ldots, i-1$. Insert(e) uses Find to find the level where e intersects an interval, then uses Fix to ensure the size constraints and finally e is moved to level 0 by use of Search.

- Find(e) - returns the level i of the interval that e intersects along with e's type and whatever e is in the dictionary or not. *[R&G: I.1–I.9]*
- Fix(i) - moves points around inside of B_i to ensure the size invariants for each type of point. Fix(i) might violate I.9 for level i. *[R: I.1–I.4 and that there exist $\tilde{c}_1, \ldots, \tilde{c}_6$ such that $|D_i| + \tilde{c}_1, |A_i| + \tilde{c}_2, |R_i| + \tilde{c}_3, |W_i| + \tilde{c}_4, |H_i| + \tilde{c}_5, |C_i| + \tilde{c}_6$ fulfill I.5–I.8, where $|\tilde{c}_i| = \mathcal{O}(1)$ for $i = 1, \ldots, 6$. G: I.1–I.8].*

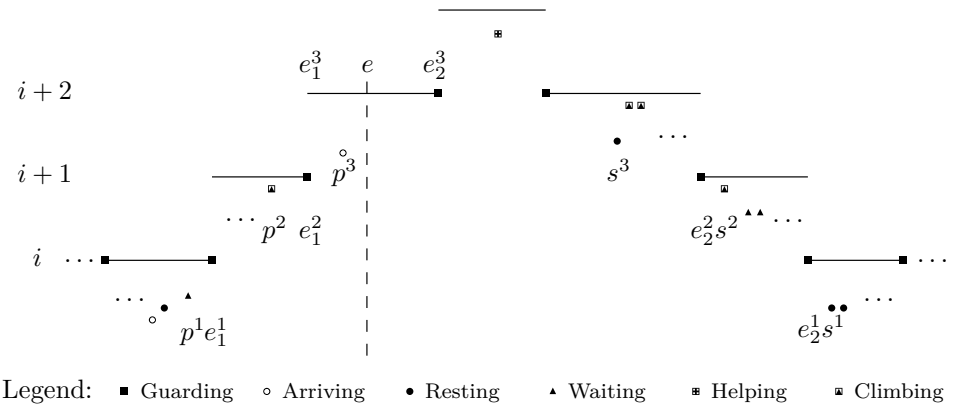

Figure 2.6: The last three iterations of the while-loop of Find(e), Predecessor(e) and Successor(e).

- Shift-Down(i) - will move at least 1 and at most c points from level i into level $i-1$. *[R: I.1–I.8 and* $|H_i| + |C_i| = 4c2^{2^{i+k}} + c'_i$*, where* $0 \leq c'_i = \mathcal{O}(1)$*. G: I.1–I.8].*

- Shift-Up(i) - will move at least 1 and at most c points from level i into level $i+1$. *[R: I.1–I.8 and* $|H_i| + |C_i| = 4c2^{2^{i+k}} + c'_i$*, where* $c \leq c'_i = \mathcal{O}(1)$*. G: I.1–I.8].*

- Move-Down($e, i, j, t_{\text{before}}, t_{\text{after}}$) - If e is in the dictionary at level i it is moved from level i to level j, where $i \geq j$. The type t_{before} is the type of e before the move and t_{after} is the type that e should have after the move, unless $i = j$ in which case e will be made arriving in level j. *[R&G: I.1–I.8].*

- Rebalance-Below(i) - If any $c_l > c$ for $l = 0, \ldots, i$ Rebalance-Below(i) will correct it so I.9 will be fulfilled again for $l = 0, \ldots, i$. *[R: I.1–I.8 and* $\sum_{l=0}^{i} \mathit{slack}(c_l) = \mathcal{O}(1)$*, where*

$$\mathit{slack}(c_l) = \begin{cases} 0 & \text{if} \quad c_l \in [-c, c]\,, \\ |c_l| - c & \text{otherwise}\,. \end{cases}$$

  G: I.1–I.9].

- Rebalance-Above(i) - If any $c_l < -c$ for $l = i, \ldots, m-1$ Rebalance-Above(i) will correct it so I.9 will be fulfilled again for $l = i, \ldots, m-1$. *[R: I.1–I.8 and* $\sum_{l=i}^{m-1} \mathit{slack}(c_l) = \mathcal{O}(1)$*. G: I.1–I.9].*

Find(e)

We start at level $i = 0$. If $e < \min(P)$ or $\max(P) < e$ we return false and 0. For each level we let $e_1 = \mathsf{p}_{G_{\leq i}}(e)$, $e_2 = \mathsf{s}_{G_{\leq i}}(e)$, $p = \mathsf{p}_{B_i \setminus G_i}(e)$ and $s = \mathsf{s}_{B_i \setminus G_i}(e)$. We find p and s by querying each of the structures D_i, A_i, R_i, W_i, H_i and C_i, we find e_1 and e_2 by querying G_i and comparing with the values of e_1 and e_2 from level $i-1$. While $p < e_1$ and $e_2 < s$ we continue to the next level, that is we increment i. Now outside the loop, if $e \in B_i$ we return i, the type of e and the boolean true as we found e, else we return i and false as we did not find e. See Figure 2.6 for an example of the execution.

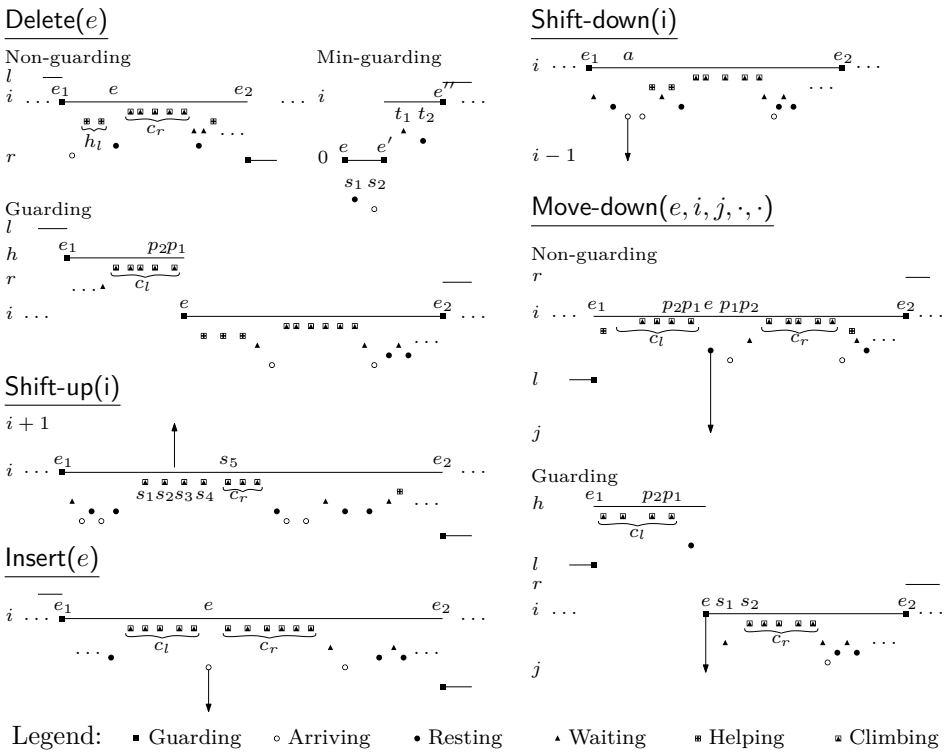

Figure 2.7: Here we see illustrations of how we maintain the intervals when updating the intervals. These only show single instances of each of the update operations many different cases.

Predecessor/Successor(e)

We describe the Predecessor (and Successor) operation. We start at level $i = 0$. If $e < \min(P)$ then return $-\infty$ ($\min(P)$). If $\max(P) < e$ then return $\max(P)$ (∞). For each level we let $e_1 = \mathsf{p}_{G_{\leq i}}(e)$, $e_2 = \mathsf{s}_{G_{\leq i}}(e)$ $p = \mathsf{p}_{B_i \setminus G_i}(e)$, and $s = \mathsf{s}_{B_i \setminus G_i}(e)$. While $p < e_1$ and $e_2 < s$ we continue to the next level, that is we increment i. When the loop breaks we return $\max(e_1, p)$ ($\min(s, e_2)$). See Figure 2.6 for an example of the execution.

Insert(e)

If $e < \min(P)$ we insert e as guarding at level 0 and make $\min(P)$ arriving at level 0, call Fix(0), Rebalance-Below(m) and return. If $\max(P) < e$ we insert e as guarding at level 0 and make $\max(P)$ arriving at level 0, call Fix(0), Rebalance-Below(m) and return.

Let $c_l = \mathsf{GIL}_{C_i}(e)$, $c_r = \mathsf{GIR}_{C_i}(e)$, $h_l = \mathsf{GIL}_{H_i}(e)$ and $h_r = \mathsf{GIR}_{H_i}(e)$. We find the level i of the interval (e_1, e_2) which e intersects using Find(e) and let $e_1 = \mathsf{p}_{G_{\leq i}}(e)$, $e_2 = \mathsf{s}_{G_{\leq i}}(e)$. See Figure 2.7.

If e is already in the dictionary we give an error. If $|c_l| > 0$ or $|c_r| > 0$ or (e_1, e_2) is of type $[e_1, e_2]$ and does not contain non-climbing points then insert e as climbing at level i. Else if $|h_l| + 1 + |h_r| \geq c$ then insert e as climbing at level i and make the points in h_l and h_r climbing at level i. Else insert e as helping at level i. Finally we call Rebalance-Below(m) and then Search(e) to move e from the current level i down to level 0.

Search(e)

We first find e's current level i and its type t, by a call to $\mathsf{Find}(e)$. If e is in the dictionary then we call $\mathsf{Move\text{-}Down}(e, i, 0, t, \text{arriving})$ which will move e from level i down to level 0 and make it arriving, while maintaining I.1–I.8, but I.9 might be broken so we finally call $\mathsf{Rebalance\text{-}Below}(i-1)$ to fix this.

Fix(i)

In the following we will be moving elements around between D_i, A_i, R_i, W_i, H_i and C_i. The moves $A_i \to R_i$ and $R_i \to W_i$, i.e., between structures which are next to each other in the memory layout, are simply performed by deleting an element from the left structure and inserting it into the right structure. The moves $W_i \to H_i \cup C_i$ and the other way around $H_i \cup C_i \to W_i$ will be explained below.

If $|D_i| > w_i$ then perform $D_i \xrightarrow{h} A_i$ where $h = |D_i| - w_i$. If $|D_i| < w_i$ and $|B_i \backslash \{\min(P), \max(P)\}| > |D_i|$ then perform $H_i \cup C_i \xrightarrow{h_1} W_i$, $W_i \xrightarrow{h_2} R_i$, $R_i \xrightarrow{h_3} A_i$ and $A_i \xrightarrow{h_4} D_i$ where $h_1 = \min(w_i - |D_i|, |H_i| + |C_i|)$, $h_2 = \min(w_i - |D_i|, |W_i| + h_1)$, $h_3 = \min(w_i - |D_i|, |R_i| + h_2)$ and $h_4 = \min(w_i - |D_i|, |A_i| + h_3)$.

If $|W_i| + |H_i| + |C_i| \neq 0$ and $|R_i| < 2^{2^{i+k}}$ then perform $H_i \cup C_i \xrightarrow{h_1} W_i$ and $W_i \xrightarrow{h_2} R_i$ where $h_1 = \min(2^{2^{i+k}} - |R_i|, |H_i| + |C_i|)$ and $h_2 = \min(2^{2^{i+k}} - |R_i|, |W_i| + h_1)$. If $|R_i| > 2^{2^{i+k}}$ then perform $R_i \xrightarrow{h_1} A_i$ where $h_1 = |R_i| - 2^{2^{i+k}}$.

If $i < m$ and $|A_i| + |W_i| < 2^{2^{i+k}}$ then perform $H_i \cup C_i \xrightarrow{h_1} W_i$, where $h_1 = \min(2^{2^{i+k}} - (|A_i| + |W_i|), |H_i| + |C_i|)$. If $|A_i| + |W_i| > 2^{2^{i+k}}$ then perform $W_i \xrightarrow{h_1} H_i \cup C_i$ where $h_1 = \min(|A_i| + |W_i| - 2^{2^{i+k}}, |W_i|)$.

If $|A_i| \geq 2^{2^{i+k}}$ then let $h_1 = |A_i| - 2^{2^{i+k}}$, delete W_i as it is empty and rename R_i to W_i. Now move h_1 elements from A_i into a new moveable dictionary X, rename A_i to R_i, rename X to A_i and perform $W_i \xrightarrow{h_1} H_i \cup C_i$.

Performing $W_i \to H_i \cup C_i$*:* Let $w = \mathsf{s}_{W_i}(-\infty)$, $c_l = \mathsf{GIL}_{C_i}(w)$, $c_r = \mathsf{GIR}_{C_i}(w)$, $h_l = \mathsf{GIL}_{H_i}(w)$, $h_r = \mathsf{GIR}_{H_i}(w)$, $e_1 = \mathsf{p}_{G_{\leq i}}(w)$ and $e_2 = \mathsf{s}_{G_{\leq i}}(w)$. If $|c_l| > 0$ or $|c_r| > 0$ or (e_1, e_2) is of type $[e_1, e_2]$ and only contains climbing points then make w climbing at level i. Else if $|h_l| + 1 + |h_r| \geq c$ then make h_l, w and h_r climbing at level i. Else make w helping at level i.

Performing $H_i \cup C_i \to W_i$*:* Let w be the minimum element of $\mathsf{s}_{H_i}(-\infty)$ and $\mathsf{s}_{C_i}(-\infty)$, and let $c_r = \mathsf{GIR}_{C_i}(w)$. Make w waiting at level i. If w was climbing and $|c_r| < c$ then make c_r helping at level i.

Shift-Down(i)

We move at least one element from level i into level $i-1$, see Figure 2.7. If $|D_i| < w_i$ then we let a be some element in D_i. If $|D_i| < |B_i|$ then: if $|A_i| = 0$ we perform[4] $H_i \cup C_i \xrightarrow{h_1} W_i$, $W_i \xrightarrow{h_2} R_i$ and $R_i \xrightarrow{h_3} A_i$, where $h_1 = \min(1, |H_i| + |C_i|)$, $h_2 = \min(1, |W_i| + h_1)$ and $h_3 = \min(1, |R_i| + h_2)$, now we know that $|A_i| > 0$ so let $a = \mathsf{s}_{A_i}(-\infty)$, i.e., a is the leftmost arriving point in A_i at level i. We call $\mathsf{Move\text{-}Down}(a, i, i-1, \text{arriving}, \text{climbing})$.

[4]The move $H_i \cup C_i \xrightarrow{l} W_i$ will be performed the same way as we did it in Fix.

Shift-Up(i)

Assume we are at level i, we want to move at least one and at most c arbitrary points from B_i into B_{i+1}. Let[5] $s_1 = \mathsf{s}_{C_i}(-\infty)$, $e_1 = \mathsf{p}_{G_{\leq i}}(s_1)$ and $e_2 = \mathsf{s}_{G_{\leq i}}(s_1)$, and let $s_2 = \mathsf{s}_{C_i \cap [e_1,e_2]}(s_1)$, $s_3 = \mathsf{s}_{C_i \cap [e_1,e_2]}(s_2)$, $s_4 = \mathsf{s}_{C_i \cap [e_1,e_2]}(s_3)$ and $s_5 = \mathsf{s}_{C_i \cap [e_1,e_2]}(s_4)$, if they exist, also let $c_r = \mathsf{GIR}_{C_i \cap [e_1,e_2]}(s_4)$ be the group of climbing elements to the immediate right of s_4, if they exist, see Figure 2.7. We will now move one or more climbing points from B_i into B_{i+1} where they become arriving points. If $i = m-1$ or $i = m$ then we put arriving points into D_{i+1}, which we might have to create, instead of A_{i+1}.

We now deal with the case where (e_1, e_2) is of type $[e_1, e_2]$ and only contains climbing points. Let l be the level of e_1's left interval, and r the level of e_2's right interval, also let c_I be the number of climbing points in the interval. If $l = i+1$ we make e_1 arriving, else we make it guarding, at level $i+1$. Make the points of s_1, s_2, s_3 and s_4 that exist arriving at level $i+1$. If $c_I \leq c$ then make s_5 arriving at level $i+1$ if it exists, also if $r = i+1$ we make e_2 arriving, else we make it guarding, at level $i+1$. Else make s_5 guarding at level i.

We now deal with the cases where (e_1, e_2) might contain non-climbing points. If $\mathsf{p}(s_1) = e_1$ we make s_1 and s_2 waiting and guarding at level i, respectively, else we make s_1 guarding at level i and s_2 arriving at level $i+1$. Now in both cases we make s_3 arriving at level $i+1$ and s_4 guarding at level i. If $\langle (s_4, e_2)$ is not of type $[s_4, e_2]$ or contains non-climbing points$\rangle$ and $|c_r| < c$, i.e., there are less than c consecutive climbing points to the right of s_4, then we make the points c_r helping at level i.

We have moved climbing points from B_i into B_{i+1}, and made them arriving. Finally we call $\mathsf{Fix}(i)$ and $\mathsf{Fix}(i+1)$.

Move-Down($e, i, j, t_{\text{before}}, t_{\text{after}}$)

Depending on the type t_{before} of point e we have different cases, see Figure 2.7.

Non-guarding Let $e_1 = \mathsf{p}_{G_{\leq i}}(e)$, $e_2 = \mathsf{s}_{G_{\leq i}}(e)$ and let l be the level of the left interval of e_1 and r the level of the right interval of e_2. Also let $p_2 = \mathsf{p}_{(B_i \setminus G_i) \cap [e_1,e_2]}(p_1)$, $p_1 = \mathsf{p}_{(B_i \setminus G_i) \cap [e_1,e_2]}(e)$, $s_1 = \mathsf{s}_{(B_i \setminus G_i) \cap [e_1,e_2]}(e)$ and $s_2 = \mathsf{s}_{(B_i \setminus G_i) \cap [e_1,e_2]}(s_1)$, let $c_l = \mathsf{FGL}_{C_i \cap [e_1,e_2]}(e)$ be the elements in the first climbing group left of e, and let $c_r = \mathsf{FGR}_{C_i \cap [e_1,e_2]}(e)$ be the elements in the first climbing group right of e.

Case $i = j$: make e arriving in level j, if $|c_l| < c$ then make the points in c_l helping at level j, if $|c_r| < c$ then make the points in c_r helping at level j. Finally call $\mathsf{Fix}(j)$.

Case $i > j$: If both p_2 and p_1 exists we make p_1 guarding in level j and let e_1' denote p_1, else make e_1 guarding in level $\min(l, j)$, and let e_1' denote e_1. If p_1 exists we make p_1 of type t_{after} at level j. If both s_1 and s_2 exists we make s_1 guarding at level j, and let e_2' denote s_1, else make e_2 guarding at level $\min(j, r)$ and let e_2' denote e_2. If s_1 exists we make s_1 of type t_{after} at level j. Lastly we make e of type t_{after} in level j. Now let c_l' denote the elements of c_l which we have not moved in the previous steps, likewise let c_r' denote the elements of c_r which we have not moved. If $\langle (e_1, e_1']$ is not of type $[e_1, e_1']$ or contains non-climbing points$\rangle$ and $|c_l'| < c$ then make c_l' helping at level i. If $\langle [e_2', e_2)$ is not of type $[e_2', e_2]$ or contains non-climbing points$\rangle$ and $|c_r'| < c$ then make c_r' helping at level i. Call $\mathsf{Fix}(i)$, $\mathsf{Fix}(j)$, $\mathsf{Fix}(\min(l, i))$ and $\mathsf{Fix}(\min(i, r))$.

Guarding If $e = \min(P)$ or $e = \max(P)$ we simply do nothing and return. Let $e_1 = \mathsf{p}_{G_{\leq h}}(e)$ be the left endpoint of the left interval $(e_1, e[$ lying at level h and

[5] See the analysis in Subsection 2.2.6 for a proof that $|C_i| > 0$.

$e_2 = \mathsf{s}_{G_{\leq i}}(e)$ be the right endpoint of the right interval $[e, e_2)$ lying at level i, we assume without loss of generality that $h > i$, the case $h < i$ is symmetric. Also let l be the level of the left interval of e_1 and r the level of the right interval of e_2. Let $p_2 = \mathsf{p}_{(B_h \setminus G_h) \cap [e_1, e]}(p_1)$ and $p_1 = \mathsf{p}_{(B_h \setminus G_h) \cap [e_1, e]}(e)$ be the two left points of e, if they exists, $s_1 = \mathsf{s}_{(B_i \setminus G_i) \cap [e, e_2]}(e)$ and $s_2 = \mathsf{s}_{(B_i \setminus G_i) \cap [e, e_2]}(s_1)$ the two right points of e, if they exits. Also let $c_l = \mathsf{FGL}_{C_h \cap [e_1, e]}(e)$ and $c_r = \mathsf{FGR}_{C_i \cap [e, e_2]}(e)$.

If p_2 does not exist we make e_1 guarding at level $\min(l, j)$, we make p_1 of type t_{after} at level j and let e_1' denote e_1, else we make p_1 guarding at level j and let e_1' denote p_1. If it is the case that $i > j$ then we check: if s_2 does not exist then we make s_1 of type t_{after} at level j, e_2 guarding at level $\min(j, r)$ and let e_2' denote e_2, else we make s_1 guarding at level j and let e_2' denote s_1. We make e of type t_{after} at level j.

Now let c_l' be the points of c_l which was not moved and c_r' the points of c_r which was not moved. If $|c_l'| < c$ then make c_l' helping at level h. We now have two cases if e_2' exists: if $|c_r'| < c$ then make c_r' helping at level i. The other case is if e_2' does not exist: if $\langle (e_1', e_2)$ is not of type $[e_1', e_2]$ or contains non-climbing points$\rangle$ and $|c_r'| < c$ then make c_r' helping at level i. In all cases call $\mathsf{Fix}(\min(l, h))$, $\mathsf{Fix}(h)$ and $\mathsf{Fix}(i)$. If $i > j$ then call $\mathsf{Fix}(j)$ and $\mathsf{Fix}(\min(j, r))$.

Delete(e)

We first call $\mathsf{Find}(e)$ to get the type of e and its level i, if e is not in the dictionary we just return. If e is in the dictionary we have two cases, depending on if e is guarding or not.

Non-guarding Let $c_l = \mathsf{GIL}_{C_i}(e)$ be the elements in the climbing group immediately left of e, let $c_r = \mathsf{GIR}_{C_i}(e)$ be the elements in the climbing group immediately right of e, let $h_l = \mathsf{GIL}_{H_i}(e)$ be the elements in the helping group immediately left of e, and let $h_r = \mathsf{GIR}_{H_i}(e)$ be the elements in the helping group immediately right of e. Let $e_1 = \mathsf{p}_{G_{\leq i}}(e)$ and let $e_2 = \mathsf{s}_{G_{\leq i}}(e)$. Let l be the level of the interval left of e_1 and r the level of the interval right of e_2. See Figure 2.7.

We have two cases, the first is $|]e_1, e_2[\cap B_i| = 1$: if $l > r$ make e_1 guarding and e_2 arriving at level r, if $l < r$ then make e_1 arriving and e_2 guarding at level l. If $l = r$ and $|P| = n \geq 4$ then make e_1 and e_2 arriving at level $l = r$. Delete e, call $\mathsf{Fix}(r)$, $\mathsf{Fix}(l)$, $\mathsf{Fix}(i)$ and $\mathsf{Rebalance\text{-}Above}(1)$.

The other case is $|]e_1, e_2[\cap B_i| > 1$: If $\langle (e_1, e_2)$ is not of type $[e_1, e_2]$ or contains non-climbing points$\rangle$ and $|c_l| + |c_r| < c$ then make c_l and c_r helping at level i. If $|h_l| + |h_r| \geq c$ then make h_l and h_r climbing at level i. Delete e, call $\mathsf{Fix}(i)$ and $\mathsf{Rebalance\text{-}Above}(1)$.

Min-guarding If $e = \min(P)$ then let $e' = \mathsf{s}_{G_{\leq m}}(e)$ and $e'' = \mathsf{s}_{G_{\leq m}}(e')$ where 0 is the level of (e, e') and i is the level of (e', e''). The case of $e = \max(P)$ is symmetric. Also let $s_1 = \mathsf{s}_{(B_0 \setminus G_0) \cap [e, e']}(e)$, $s_2 = \mathsf{s}_{(B_0 \setminus G_0) \cap [e, e']}(s_1)$, $t_1 = \mathsf{s}_{(B_i \setminus G_i) \cap [e', e'']}(e')$ and $t_2 = \mathsf{s}_{(B_i \setminus G_i) \cap [e', e'']}(t_1)$. See Figure 2.7.

If s_2 exists then delete e make s_1 guarding at level 0 and call $\mathsf{Fix}(0)$. If s_2 does not exist and t_2 exists then delete e make s_1 and t_1 guarding and e' arriving at level 0 and finally call $\mathsf{Fix}(0)$ and $\mathsf{Fix}(i)$. If s_2 does not exist and t_2 does not exist then delete e, make s_1 and e'' guarding and e' and t_1 arriving at level 0 and call $\mathsf{Fix}(0)$ and $\mathsf{Fix}(i)$. Finally call $\mathsf{Rebalance\text{-}Above}(1)$.

Guarding Let $e_1 = \mathsf{p}_{G_{\leq h}}(e)$ and $e_2 = \mathsf{s}_{G_{\leq h}}(e)$, where h is the level of the left interval $(e_1 : e[$, i the level of the right interval $[e : e_2)$ and l the level of the left interval that e_1 participates in. We assume without loss of generality that $h > i$, the case $h < i$

is symmetric. Let $p_2 = \mathsf{p}_{(B_h \setminus G_h) \cap [e_1, e]}(p_1)$, $p_1 = \mathsf{p}_{(B_h \setminus G_h) \cap [e_1, e]}(e)$ and $c_l = \mathsf{FGL}_{C_i}(e)$ be the points in the first group of climbing points left of e. See Figure 2.7.

If p_2 exist we make p_1 guarding at level i, and let e' denote p_1, else we make e_1 guarding at level $\min(l, i)$, let e' denote e_1 and if $[e', e_2)$ is of type $[e', e_2]$ and contains only climbing points then we make p_1 climbing at level i else we make p_1 waiting at level i. Let c'_l be the points in c_l which was not moved in the previous movement of points. If $|c'_l| < c$ make c'_l helping at level h. If e' is e_1 then call $\mathsf{Fix}(l)$. Delete e, call $\mathsf{Fix}(h)$, $\mathsf{Fix}(i)$ and $\mathsf{Rebalance\text{-}Above}(1)$.

Rebalance-Below(i)

For each level $l = 0, \ldots, i$ we perform a $\mathsf{Shift\text{-}Up}(l)$ while $c < c_l$.

Rebalance-Above(i)

For each level $l = i, \ldots, m-1$ we perform $\mathsf{Shift\text{-}Down}(l+1)$ while $c_l < -c$.

2.2.5 Memory Management

We will now deal with the memory layout of the data structure. We will put the blocks in the order $B_0, \ldots, B_m$, where block B_i has its dictionaries in the order $D_i, A_i, R_i, W_i, H_i, C_i$ and G_i, see Figure 2.3. Block B_m grows and shrinks to the right when elements are inserted and deleted from the working set dictionary.

The D_i structure is not a moveable dictionary as the other structures in a block are, it is simply an array of $w_i = d \cdot 2^{i+k}$ elements which we use to encode the size of each of the structures A_i, R_i, W_i, H_i, C_i and G_i along with their own auxiliary data, as they are not implicit and need to remember $\mathcal{O}(2^{i+k})$ bits which we store here. As each of the moveable dictionaries in B_i have size $\mathcal{O}(2^{2^{i+k}})$ we need to encode numbers of $\mathcal{O}(2^{i+k})$ bits in D_i.

We now describe the memory management concerning the movement, insertion and deletion of elements from the working-set dictionary. First notice that the operations Find, Predecessor and Successor do not change the working-set dictionary or memory layout. Also the operations Shift-Down, Search, Rebalance-Below and Rebalance-Above only call other operations, hence their memory management are handled by the operations they call. The only operations where actual memory management comes into play are in Insert, Shift-Up, Fix, Move-Down and Delete. We will now describe two operations Internal-Movement – which handles movement inside a single block/level – and External-Movement – which handles movement across different blocks/levels. Together these two operations handle all memory management.

Internal-Movement($i, m_1, \ldots, m_l$)

Internal-Movement performs the list of *internal moves* $m_1, \ldots, m_l$ on block B_i at level i, where $l = \mathcal{O}(1)$ and we assume that there is sufficient space to the right of block B_i for Internal-Movement to perform all its moves. Move m_j consists of:

- the index $\gamma = D_i, A_i, R_i, W_i, H_i, C_i, G_i$ of the dictionary to change, where we assume[6] that $m_j.\gamma < m_h.\gamma$, for $j < h$,
- the set of elements S_{in} to put into γ, where $|S_{\text{in}}| = \mathcal{O}(1)$,

[6] We will misuse notation and let $\gamma + 1$ denote the next in the total order D, A, R, W, H, C, G. We will also compare $m_j.\gamma$ and $m_h.\gamma$ with $\leq$ in this order.

- the set of elements S_{out} to take out of γ, where $|S_{\text{out}}| = \mathcal{O}(1)$ and
- the total size difference $\delta = |S_{\text{in}}| - |S_{\text{out}}|$ of γ after the move.

Internal-Movement$(i, m_1, \ldots, m_l)$

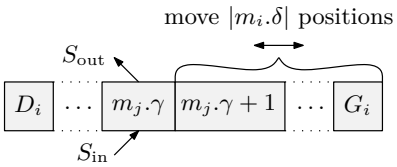

Figure 2.8: Memory movement of Internal-Movement inside of a block B_i.

For $j = 1, \ldots, l$ do: if $m_j.\delta < 0$ then remove S_{out} from γ, insert S_{in} into γ and move $\gamma + 1, \ldots, G$ left $|m_j.\delta|$ positions, where we move them in the order $\gamma + 1, \ldots, G$. If $m_j.\delta > 0$ then move $\gamma + 1, \ldots, G$ right $m_j.\delta$ positions, where we move them in the order $G, \ldots, \gamma + 1$, remove S_{out} from γ and insert S_{in} into γ. See Figure 2.8.

External-Movement$(M_1, \ldots, M_l)$

External-Movement takes a list of *external moves* $M_1, \ldots, M_l$, where $l = \mathcal{O}(1)$. Move M_j consists of:

- the index $0 \leq \gamma \leq m$ of the block/level to perform the internal moves $m_1, \ldots, m_q$ on, where $M_j.\gamma < M_h.\gamma$ for $j < h$,
- the list of internal moves $m_1, \ldots, m_q$ to perform on block γ, where $q = \mathcal{O}(1)$, and
- the total size difference $\Delta = \sum_{h=1}^{q} m_h.\delta$ of block γ after all the internal moves $m_1, \ldots, m_q$ have been performed.

Let $\overline{\Delta} = \sum_{i=1}^{l} M_i.\Delta$ be the total size change of the dictionary after the external-moves have been performed. If $\overline{\Delta} = 0$ then we let $\gamma_{\text{end}} = M_l.\gamma$ else we let $\gamma_{\text{end}} = m$. Let $p_{\text{end}} = \sum_{j=0}^{\gamma_{\text{end}}} |B_j| + \overline{\Delta}$ be the last address of the right most block γ_{end} that we need to alter. Let $s_1, \ldots, s_k$ be the sublist of the indexes $\{1, \ldots, l\}$ where $M_{s_i}.\Delta \leq 0$ for $i = 1, \ldots, k$. Let $a_1, \ldots, a_h$ be the sublist of the indexes $\{1, \ldots, l\}$ where $M_{a_i}.\Delta > 0$ for $i = 1, \ldots, h$.

External-Movement operates as follows: 1) We first perform all the internal moves of each of the external moves $M_{s_1}, \ldots, M_{s_k}$. 2) We compact all the blocks i right, where $i \in [M_1.\gamma, \gamma_{\text{end}}]$, so the rightmost block ends at position p_{end}. 3) We compact blocks $i \in [M_1.\gamma, M_{a_1}.\gamma]$ to the left so they align with block $M_1.\gamma - 1$. 4) Finally for each block $M_{a_i}.\gamma$, for $i = 1, \ldots, h$, we compact all blocks $i \in [M_{a_{i-1}}.\gamma + 1, M_{a_i}.\gamma]$ left and perform each internal move of the external move M_{a_i} on block $M_{a_i}.\gamma$. See Figure 2.9.

Memory Management in Updates of Intervals

With the above two operations we can perform the memory management when updating the intervals in Subsection 2.2.4: Whenever an element moves around, is deleted or inserted, it is simply put in one or two internal moves. All internal moves in a single block/level are grouped into one external move. Since all updates of intervals only move around a constant number of elements, the requirements for internal/external-movement that $l = \mathcal{O}(1)$ and $q = \mathcal{O}(1)$ are fulfilled.

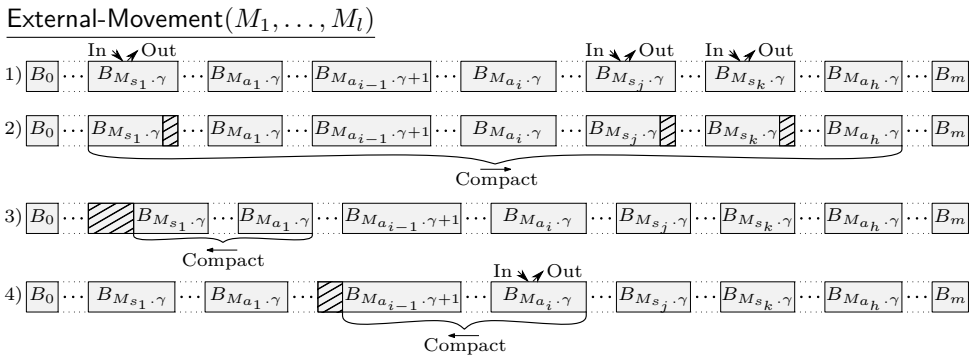

Figure 2.9: Memory movement of External-Movement across multiple blocks $B_{M_1.\gamma}, \ldots, B_{M_l.\gamma}$.

2.2.6 Analysis

We will leave it for the reader to check that the pre-conditions for each operation in Subsection 2.2.4 are fulfilled and that the operations maintains all invariants. We will instead concentrate on using the invariants to prove correctness of the Find, Predecessor, Successor and Shift-Up operations along with proving time and cache-miss bounds for these. We will leave the time and cache-miss bounds of Search, Rebalance-Above, Rebalance-Below, Shift-Down, Insert, Delete and Fix for the reader as they are all similarly in nature. At the end of this subsection we will have proved Theorem 2.1:

Theorem 2.1. *There exists a cache-oblivious implicit dynamic dictionary with the working-set property that supports the operations Insert and Delete in time $\mathcal{O}(\log n)$ and $\mathcal{O}(\log_B n)$ cache-misses, Search, Predecessor and Successor in time $\mathcal{O}(\log \min(\ell_{\mathsf{p}(e)}, \ell_e, \ell_{\mathsf{s}(e)}))$, $\mathcal{O}(\log \ell_{\mathsf{p}(e)})$ and $\mathcal{O}(\log \ell_{\mathsf{s}(e)})$, and cache-misses $\mathcal{O}(\log_B \min(\ell_{\mathsf{p}(e)}, \ell_e, \ell_{\mathsf{s}(e)}))$, $\mathcal{O}(\log_B \ell_{\mathsf{p}(e)})$ and $\mathcal{O}(\log_B \ell_{\mathsf{s}(e)})$, respectively, where $\mathsf{p}(e)$ and $\mathsf{s}(e)$ are the predecessor and successor of e, respectively.*

Find(e)

We only consider the cases where $\min(P) < e < \max(P)$, the other cases trivially gives the correct answer in $\mathcal{O}(1)$ time and cache-misses as $\min(P), \max(P) \in G_0$. Assume that Find(e) stops at level i, then we have that $e_1 \leq p$ or $s \leq e_2$ so $I(e_1, e_2, i) \neq \emptyset$ and i is the minimal i where this happens, see Lemma 2.1. Notice that $e_1 = \mathsf{p}_{G_{\leq i}}(e)$ and $e_2 = \mathsf{s}_{G_{\leq i}}(e)$, so e_1 and e_2 are the same as in Lemma 2.1. When the while loop breaks we have all the preconditions for Lemma 2.1. Now e is either in the dictionary, or not, and if e is in the dictionary it is either guarding or not, so we have three cases.

Case 1) e is in the dictionary and is non-guarding: then we have from Lemma 2.1 that (e_1, e_2) is an interval at level i and $e \in B_i$. From this we also have that $\log(\ell_e) \geq \log(2^{2^{i+k-1}})$.

Case 2) e is not in the dictionary: from Lemma 2.1 the interval (e_1, e_2) lie at level i and we know that e intersects it. Since e is not in the dictionary $\ell_e = n$ and then $\log(\ell_e) \geq \log(2^{2^{i+k-1}})$.

Case 3) e is in the dictionary and is guarding: from Lemma 2.1 we have that either (e_1, e) or (e, e_2) lie in level i, hence $e \in G_i \subseteq B_i$. From this we also have that $\log(\ell_e) \geq \log(2^{2^{\max(i,j)+k-1}}) \geq \log(2^{2^{i+k-1}})$.

From the above we see that $\mathsf{Find}(e)$ runs in $\mathcal{O}(\log(2^{2^{i+k-1}})) = \mathcal{O}(\log \min(\ell_{\mathsf{p}(e)}, \ell_e, \ell_{\mathsf{s}(e)}))$ time. When we look at the cache-misses we will first notice that the first $\lfloor \log \log B \rfloor$ levels will fit in a single cache-line because all levels are next to each other in the memory layout, so the total cache-misses will be

$$\begin{aligned} & \mathcal{O}\left(1 + \sum_{j=\lfloor \log \log B \rfloor + 1}^{i} \left(1 + \log_B \left(2^{2^{j+k}}\right)\right)\right) \\ = \quad & \mathcal{O}\left(\frac{2^{i+k}}{\log B}\right) = \mathcal{O}(\log_B \min(\ell_{\mathsf{p}(e)}, \ell_e, \ell_{\mathsf{s}(e)})). \end{aligned}$$

Predecessor/Successor(e)

We will only handle the $\mathsf{Predecessor}$ operation, the case for the $\mathsf{Successor}$ is symmetric. Since we have the same condition in the while loop as for Find, we know that when it breaks it implies that $I(e_1, e_2, i) \neq \emptyset$. So from Lemma 2.1, e intersects an interval at level i and the predecessor of e is now $\max(e_1, p)$.

From I.4 we know that $\log(\ell_p) \geq \log(2^{2^{i+k-1}})$ and the total time usage is $\sum_{j=0}^{i} \mathcal{O}(\log(2^{2^{i+k}})) = \mathcal{O}(2^{i+k}) = \mathcal{O}(\log(\ell_p))$. Like in Find, the first $\lfloor \log \log B \rfloor$ levels fit into one block/cache-line hence the total cache-misses will be $\mathcal{O}(\log_B(\ell_p))$.

Shift-Up(i)

For $\mathsf{Shift\text{-}Up}$ to work for level i it is mandatory that $|C_i| > 0$ so that $\mathsf{s}_{C_i}(-\infty)$ will return an element which can be moved to level $i+1$. From the precondition that $|H_i| + |C_i| = 4c2^{2^{i+k}} + c_i'$, where $c \leq c_i' = \mathcal{O}(1)$, we have that

$$|C_i| = 4c2^{2^{i+k}} + c_i' - |H_i| \geq 4c2^{2^{i+k}} - c - |H_i|$$

so proving that $|H_i| < 4c2^{2^{i+k}} - c$ is enough. From I.3 we can at most have $c-1$ helping points in a helping group, so for every $c-1$ helping points we need a separating point, the role of the separating point can be played by a point from D_i, A_i, R_i, W_i or $G_{\leq i-1}$. These are the only ways to contribute points to H_i hence for $i \geq 1$ we have this bound

$$\begin{aligned} |H_i| \quad & \leq \quad (c-1)(|D_i| + |A_i| + |R_i| + |W_i| + |G_{\leq i-1}|) \\ & \overset{(1)}{\leq} \quad (c-1)\left(w_i + 2 \cdot 2^{2^{i+k}} + \sum_{j=0}^{i-1} \left((4 + 2d + 8c)2^{2^{j+k}} + 2c\right)\right) \\ & \leq \quad (c-1)\left(w_i + 2 \cdot 2^{2^{i+k}} + 2ci + (4 + 2d + 8c) \sum_{j=0}^{i-1} 2^{2^{j+k}}\right) \\ & \overset{(2)}{\leq} \quad (c-1)\left(d \cdot 2^{i+k} + 2 \cdot 2^{2^{i+k}} + (4 + 2d + 8c) \cdot 2 \cdot 2^{2^{i+k-1}} + 2ci\right) \end{aligned}$$

Where we in (1) have used I.5, I.6 I.7 and O.1, and in (2) we used that $\sum_{j=0}^{i-1} 2^{2^{j+k}} \leq 2 \cdot 2^{2^{i-1+k}}$, which can easily be prove by induction in i with the base case of $i = 1$.

If we now use that $c = 5$ then we get

$$\begin{aligned} |H_i| \quad & \leq \quad 8 \cdot 2^{2^{i+k}} + (88 + 16d) \cdot 2^{2^{i+k-1}} + 4d \cdot 2^{i+k} + 40i \\ & \overset{(3)}{\leq} \quad 11 \cdot 2^{2^{i+k}} \end{aligned}$$

where we in (3) require that $88 + 16d \leq 2^{2^{i+k-1}}$, $4d \cdot 2^{i+k} \leq 2^{2^{i+k}}$ and $40i \leq 2^{2^{i+k}}$. Since $i \geq 1$, we can insert $i = 1$ as that will make all the requirements the strongest. From the first we get $k \geq \log\log(88 + 16d)$, the second $k \geq \log k$ and the third $k \geq \log\log(40) - 1$. Which are satisfied for $k \geq \max(\log\log(88 + 16d), 2)$. This gives us that $|C_i| \geq 4c2^{2^{i+k}} - c - |H_i| > 0$ for $i = 1, \ldots, m-1$.

For $i = 0$ we have a different bound as $G_{\leq i-1}$ is empty, we get the bound

$$\begin{aligned} |H_0| &\leq (c-1)(|D_i| + |A_i| + |R_i| + |W_i|) \\ &\leq (c-1)\left(d \cdot 2^{i+k} + 2 \cdot 2^{2^{i+k}}\right) \end{aligned}$$

If we again insert that $c = 5$ we get that

$$\begin{aligned} |H_0| &\leq 4\left(d \cdot 2^{i+k} + 2 \cdot 2^{2^{i+k}}\right) \\ &= 8 \cdot 2^{2^k} + 4d \cdot 2^k \\ &\overset{(4)}{\leq} 9 \cdot 2^{2^k} \end{aligned}$$

where we in (4) require that $4d \cdot 2^k \leq 2^{2^k}$ which is true when $k \geq \log\log(4d) + 1$, which is already true when $k \geq \max(\log\log(88 + 16d), 2)$. So we have proved that $|C_i| > 0$ for level $i = 0, \ldots, m-1$.

Move-Down$(e, i, j, t_{\textbf{before}}, t_{\textbf{after}})$

Move-Down moves a constant number of points around and into level j from level i. If e is non-guarding we call Fix(i), Fix(j), Fix$(\min(l, i))$ and Fix$(\min(i, r))$. If e is guarding where $h > i$ we call Fix$(\min(l, h))$, Fix(h) and Fix(i), and if $i > j$ we also call Fix(j) and Fix$(\min(j, r))$. In the non-guarding case the time is bounded by $\mathcal{O}(\log 2^{2^{i+k}}) = \mathcal{O}(\log \ell_e)$ and the cache-miss bounds are dominated by $\mathcal{O}(\log_B 2^{2^{i+k}}) = \mathcal{O}(\log_B \ell_e)$. In the guarding case the time is bounded by $\mathcal{O}(\log 2^{2^{h+k}})$ $= \mathcal{O}(\log \ell_e)$ and the cache-miss bounds are dominated by $\mathcal{O}(\log_B 2^{2^{h+k}}) = \mathcal{O}(\log_B \ell_e)$.

Internal-Movement$(i, m_1, \ldots, m_l)$

It takes $\mathcal{O}(\log(2^{2^{i+k}})) = \mathcal{O}(2^{i+k})$ time and $\mathcal{O}(\log_B(2^{2^{i+k}})) = \mathcal{O}(\frac{2^{i+k}}{\log B})$ cache-misses to perform move j. In total all the moves $m_1, \ldots, m_l$ use $\mathcal{O}(2^{i+k})$ time and $\mathcal{O}(\frac{2^{i+k}}{\log B})$ cache-misses, as $l = \mathcal{O}(1)$.

External-Movement$(M_1, \ldots, M_l)$

It takes $\mathcal{O}(l \log(2^{2^{i+k}})) = \mathcal{O}(l2^{i+k})$ time and $\mathcal{O}(l \log_B(2^{2^{i+k}})) = \mathcal{O}(l\frac{2^{i+k}}{\log B})$ cache-misses to perform the internal moves on level i. In total all the external moves $M_1, \ldots, M_l$ use $\mathcal{O}(2^{\gamma_{\text{end}}+k})$ time and $\mathcal{O}(\frac{2^{\gamma_{\text{end}}+k}}{\log B})$ cache-misses, as the external move at level γ_{end} dominates the rest and $l = \mathcal{O}(1)$.

Chapter 3

Catenable Priority Queues with Attrition

In this chapter we will present ephemeral *I/O-efficient catenable priority queues with attrition (I/O-CPQAs)* that store a set of elements from a total order. The I/O-CPQAs support the operations of Find-Min, Delete-Min, Insert-and-Attrite and Catenate-and-Attrite in $\mathcal{O}(1)$ worst-case I/Os and $\mathcal{O}(1/b)$ amortized I/Os. For the amortized bound we need to pre-load a constant number of blocks into main memory for every root I/O-CPQA, for any parameter $1 \leq b \leq B$. We call these pre-loaded records *critical records*. For the sake of simplicity, we identify an element with its value. Denote by Q an I/O-CPQA and by $\min(Q)$ the smallest element stored in Q. We denote by Q also the set of elements in I/O-CPQA Q. An I/O-CPQA supports the operations formally defined as:

- Find-Min(Q) returns $\min(Q)$.
- Delete-Min(Q) removes $\min(Q)$ from Q and returns it. The resulting I/O-CPQA is $Q' = Q \backslash \{\min(Q)\}$, and Q is discarded.
- Catenate-and-Attrite(Q_1, Q_2) catenates I/O-CPQA Q_2 at the end of another I/O-CPQA Q_1, removes all elements in Q_1 that are larger than or equal to $\min(Q_2)$ (attrition), and returns the result as a combined I/O-CPQA $Q_1' = \{e \in Q_1 \mid e < \min(Q_2)\} \cup Q_2$. The old I/O-CPQAs Q_1 and Q_2 are discarded.
- Insert-and-Attrite(Q, e)[1] appends e to the end of Q while attriting elements larger than e. The resulting I/O-CPQA is $Q' = \{e' \in Q \mid e' < e\} \cup \{e\}$.

3.1 Structure

An I/O-CPQA Q consists of two sorted buffers, called the first buffer $F(Q)$ with $[b, 4b]$ elements and the last buffer $L(Q)$ with $[0, 4b]$ elements, and $k_Q + 2$ deques of records, called the *clean* deque $C(Q)$, the *buffer* deque $B(Q)$ and the *dirty* deques $D_1(Q), \ldots, D_{k_Q}(Q)$, where $k_Q \geq 0$. A *record* $r = (l, p)$ consists of a buffer l of $[b, 4b]$ sorted elements and a pointer p to an I/O-CPQA. A record is *simple* when its pointer p is *nil*. The definition of I/O-CPQAs implies an underlying tree structure when pointers are considered as edges and I/O-CPQAs as subtrees. We define the ordering of the elements in a record r to be all elements of its buffer l followed by

[1] Insert-and-Attrite(Q, e) is implemented by a call to Catenate-and-Attrite(Q_1, Q_2), where Q_2 contains only element e.

all elements in the I/O-CPQA referenced by pointer p. We define the queue order of I/O-CPQA Q to be $F(Q)$, $C(Q)$, $B(Q)$ and $D_1(Q), \ldots, D_{k_Q}(Q)$ and $L(Q)$. It corresponds to an Euler tour over the tree structure. See Figure 3.1 for an overview of the structure.

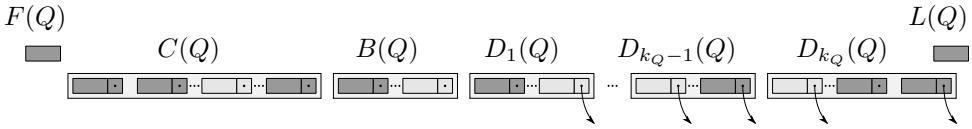

Figure 3.1: The records in $C(Q)$ and $B(Q)$ are simple, the records of $D_1(Q), \ldots, D_{k_Q}(Q)$ may contain pointers to other I/O CPQAs. I/O-CPQAs imply a tree structure. Dark gray records are critical.

Given a record $r = (l, p)$, the minimum and maximum elements in the buffer of r, are denoted by $\min(r) = \min(l)$ and $\max(r) = \max(l)$, respectively. They appear respectively first and last in the queue order of l, since the buffer of r is sorted by value. Given a deque q, the first and the last records are denoted by $\text{first}(q)$ and $\text{last}(q)$, respectively. Also, $\text{rest}(q)$ denotes all records of the deque q excluding the record $\text{first}(q)$. Similarly, $\text{front}(q)$ denotes all records of the deque q excluding the record $\text{last}(q)$. The size $|F(Q)|$ ($|L(Q)|$) of the buffer $F(Q)$ ($L(Q)$) is defined to be the number of elements in $F(Q)$ ($L(Q)$). The size $|r|$ of a record r is defined to be the number of elements in its buffer. The size $|q|$ of a deque q is defined to be the number of records it contains. The size $|Q|$ of the I/O-CPQA Q is defined to be the number of elements (both attrited and non-attrited) that Q contains. For an I/O-CPQA Q we denote by $\text{first}(Q)$ and $\text{last}(Q)$, respectively, the first and last records out of all the records of all the deques $C(Q), B(Q), D_1(Q), \ldots, D_{k_Q}(Q)$ that exist in Q.

3.2 Invariants

Having defined the structure of the I/O-CPQA we will now state the invariants that we will maintain and use later to prove correctness and I/O bounds.

I.1) For every record $r = (l, p)$ where pointer p references I/O-CPQA Q', $\max(l) < \min(Q')$ holds.

I.2) In all deques of Q where record $r_1 = (l_1, p_1)$ precedes record $r_2 = (l_2, p_2)$: $\max(l_1) < \min(l_2)$ holds.

I.3) For the buffer $F(Q)$ and deques $C(Q), B(Q), D_1(Q)$: $\max(F(Q)) < \min(\text{first}(C(Q))) < \max(\text{last}(C(Q))) < \min(\text{first}(B(Q))) < \min(\text{first}(D_1(Q)))$ holds.

I.4) Element $\min(\text{first}(D_1(Q)))$ is the smallest element in the dirty deques $D_1(Q), \ldots, D_k(Q)$.

I.5) $\min(\text{first}(D_1(Q))) < \min(L(Q))$.

I.6) All records in the deques $C(Q)$ and $B(Q)$ are simple.

I.7) $|C(Q)| \geq \sum_{i=1}^{k_Q} |D_i(Q)| + k_Q$.

I.8) $|F(Q)| < b$ holds iff $|Q| < b$ holds.

I.9) If Q is a child of another I/O-CPQA then $F(Q) = \emptyset$ and $L(Q) = \emptyset$ holds.

From invariants I.2), I.3), I.4) and I.5), we have that $\min(Q) = \min(F(Q))$. Define the *state* $\Delta(Q)$ of Q to be:

$$\Delta(Q) = |C(Q)| - \sum_{i=1}^{k_Q} |D_i(Q)| - k_Q$$

We say that an operation *improves* or *aggravates* the inequality of Invariant I.7) by a parameter c for I/O-CPQA Q, when the operation, respectively, increases or decreases the state $\Delta(Q)$ of Q by c.

To argue about the $\mathcal{O}(\frac{1}{b})$ amortized I/O bounds we need more definitions. By records(Q) we denote all records in Q and the records in the I/O-CPQAs pointed to by Q and its descendants. We call an I/O-CPQA Q *large* if $|Q| \geq b$ and *small* otherwise. We define the following potential functions for large and small I/O-CPQAs. In particular, for large I/O-CPQAs Q the potential $\Phi(Q)$ is defined as

$$\Phi(Q) = \Phi_F(|F(Q)|) + |\text{records}(Q)| + \Phi_L(|L(Q)|),$$

where

$$\Phi_F(x) = \begin{cases} 5 - \frac{2x}{b}, & b \leq x < 2b \\ 1, & 2b \leq x < 3b \\ \frac{2x}{b} - 5, & 3b \leq x \leq 4b \end{cases}$$

and

$$\Phi_L(x) = \begin{cases} \frac{x}{b}, & 0 \leq x < b \\ 1, & b \leq x \leq 3b \\ \frac{2x}{b} - 5, & 3b < x \leq 4b \end{cases}$$

For small I/O-CPQAs Q, the potential $\Phi(Q)$ is defined as

$$\Phi(Q) = \frac{3|Q|}{b}$$

The total potential Φ_T is defined as

$$\Phi_T = \sum_{Q} \Phi(Q) + \sum_{Q\colon b \leq |Q|} 1,$$

where the first sum is the total potential of all I/O-CPQAs Q and the second sum counts the number of large I/O-CPQAs Q.

3.3 Operations

In the following, we describe the algorithms that implement the operations supported by the I/O-CPQA Q. Most of the operations call the auxiliary operations Bias(Q) and Fill(Q), which we describe last. Bias improves the inequality of I.7) for Q by at least 1 if Q contains any records. Fill(Q) ensures that I.8) is maintained.

Find-Min(Q)

Find-Min(Q) returns the value $\min(F(Q))$.

Delete-Min(Q)

Delete-Min removes element $e = \min(F(Q))$ from the first buffer $F(Q)$, calls Fill(Q) and returns e.

Catenate-and-Attrite(Q_1, Q_2)

Catenate-and-Attrite(Q_1, Q_2) creates a new I/O-CPQA Q_1' by modifying Q_1 and Q_2, and by calling Bias(Q_1'), Bias(Q_2), Fill(Q_1') and Fill(Q_2).

If $|Q_1| < b$ holds, then Q_1 consists only of the first buffer $F(Q_1)$. Let $F'(Q_1)$ be the non-attrited elements of $F(Q_1)$, under attrition by $\min(F(Q_2))$. Prepend $F'(Q_1)$ onto the first buffer $F(Q_2)$ of Q_2. If this prepend causes $|F(Q_2)| > 4b$, then we take the last $2b$ elements out of $F(Q_2)$, make a new record out of them and we prepend it onto the deque $C(Q_2)$.

If $|Q_2| < b$ holds, then Q_2 only consists of $F(Q_2)$. If $|Q_1| < b$ then we delete the attrited elements in $F(Q_1)$ and append $F(Q_2)$ onto $F(Q_1)$. We now assume that $|Q_1| \geq b$. We have three cases, depending on how much of Q_1 is attrited by Q_2. Let $r = (l, \cdot) = \text{last}(Q_1)$ and let $e = \min(Q_2)$.

1. $e \leq \min(r)$: Delete r. We now have four cases:
   1) If $e \leq \min(F(Q_1))$ holds, we discard I/O-CPQA Q_1 and set $Q_1' = Q_2$.
   2) Else if $e \leq \max(\text{last}(C(Q_1)))$ holds, we prepend $F(Q_1)$ onto $C(Q_1)$, set $F(Q_1') = \emptyset$, $C(Q_1') = \emptyset$, $B(Q_1') = C(Q_1)$, $k_{Q_1'} = 0$ and $L(Q_1') = F(Q_2)$. We call Bias(Q_1') once to restore I.7) and then call Fill(Q_1') once to restore Invariant I.8).
   3) Else if $e \leq \min(\text{first}(B(Q_1)))$ or $e \leq \min(\text{first}(D_1(Q_1)))$ holds, we set $Q_1' = Q_1$ and $k_{Q_1'} = 0$ and set $L(Q_1') = F(Q_2)$. If $e \leq \min(\text{first}(B(Q_1)))$ holds, we set $B(Q_1') = \emptyset$, else we set $B(Q_1') = B(Q_1)$.
   4) Else, let $L'(Q_1)$ be the non-attrited elements under attrition by $\min(F(Q_2))$. If $|L'(Q_1)| + |F(Q_2)| \leq 4b$ then append $F(Q_2)$ to $L'(Q_1)$, else $|L'(Q_1)| + |F(Q_2)| > 4b$ so take the first $4b$ elements of $L'(Q_1)$ and $F(Q_2)$ and make into a new record in a new last dirty queue of Q_1', leave the rest in $L(Q_1')$, set $k_{Q_1'} = k_{Q_1} + 1$ and call Bias(Q_1') twice to restore I.7).
2. Else if $e \leq \min(L(Q_1))$, we set $Q_1' = Q_1$ and $L(Q_1') = F(Q_2)$.
3. Else $\min(L(Q_1)) < e$: Let l' be the non-attrited elements of l, under attrition by $\min(L(Q_1))$, and $L'(Q_1)$ be the non-attrited elements, under attrition by e. If $|L'(Q_1)| + |F(Q_2)| > 4b$ holds, we do the following: if $|l'| < |l|$ holds, we put the first $4b - |l'|$ elements of $L'(Q_1)$ and $F(Q_2)$ into l along with l'. Moreover, if we still have more than $3b$ elements left in $L'(Q_1)$ and $F(Q_2)$, we put the first $3b$ elements into a new last record of $D_{k_{Q_1}}(Q_1)$. Finally, we leave the remaining elements in $L(Q_1)$. If we added a new last record to $D_{k_{Q_1}}(Q_1)$, we also call Bias(Q) once.

We have now entirely dealt with the cases where $|Q_1| < b$ or $|Q_2| < b$ holds, so in the following we assume that $|Q_1| \geq b$ and $|Q_2| \geq b$ hold, i.e., any I/Os incurred in the cases (1–4) below are already paid for, since the total number of large I/O-CPQAs decreases by one. Let $e = \min(Q_2)$.

1. If $e \leq \min(F(Q_1))$ holds, we discard I/O-CPQA Q_1 and set $Q_1' = Q_2$.
2. Else if $e \leq \max(\text{last}(C(Q_1)))$ holds, we prepend $F(Q_1)$ onto $C(Q_1)$ and $F(Q_2)$ onto $C(Q_2)$. We remove the simple record $(l, \cdot) = \text{first}(C(Q_2))$ from $C(Q_2)$, set $Q_1' = Q_1$, $F(Q_1') = \emptyset$, $C(Q_1') = \emptyset$, $B(Q_1') = C(Q_1)$, $D_1(Q_1') = (l, p)$, $k_{Q_1'} = 1$, $L(Q_1') = L(Q_2)$ and $L(Q_2') = \emptyset$, where p points to Q_2' if it exists. This gives $\Delta(Q_1') = -2$, thus we call Bias(Q_1') twice and Fill(Q_1') once.

3. Else if $e \leq \min(\text{first}(B(Q_1)))$ or $e \leq \min(\text{first}(D_1(Q_1)))$ holds, we prepend $F(Q_2)$ onto $C(Q_2)$ and remove the simple record $(l, \cdot) = \text{first}(C(Q_2))$ from $C(Q_2)$, set $Q_1' = Q_1$, $D_1(Q_1') = (l, p)$, $k_{Q_1'} = 1$, $L(Q_1') = L(Q_2)$, $L(Q_2') = \emptyset$ and set p to point to Q_2', if it exists. If $e \leq \min(\text{first}(B(Q_1)))$ holds, we set $B(Q_1') = \emptyset$, else we set $B(Q_1') = B(Q_1)$. This gives $\Delta(Q_1') = -2$ in the worst-case, thus we call $\mathsf{Bias}(Q_1')$ twice.

4. Else let $L'(Q_1)$ be the non-attrited elements of $L(Q_1)$, under attrition by $F(Q_2)$. If $|L'(Q_1)| + |F(Q_2)| \leq 4b$ holds, then we make $L'(Q_1)$ and $F(Q_2)$ into the first record of $C(Q_2)$. Else we make them into the first two records of $C(Q_2)$ of size $\lfloor(|L'(Q_1)| + |F(Q_2)|)/2\rfloor$ and $\lceil(|L(Q_1)| + |F(Q_2)|)/2\rceil$ each. We set $Q_1' = Q_1$, $F(Q_2') = \emptyset$, $L(Q_1') = L(Q_2)$, $L(Q_2') = \emptyset$, remove $(l_2, \cdot) = \text{first}(C(Q_2))$ from $C(Q_2)$. Moreover, we add (l_2, p) as a new single record in $D_{k_{Q_1}+1}(Q_1')$, where p points to the rest of Q_2', if it exists, and set $k_{Q_1'} = k_{Q_1} + 1$. All this aggravates the inequality of I.7) for Q_1' by at most 2, so we call $\mathsf{Bias}(Q_1')$ twice.

Fill(Q)

$\mathsf{Fill}(Q)$ restores invariant I.8), if it is violated. In particular, if $|F(Q)| < b$ and $|Q| \geq b$, let $r = (l, \cdot) = \text{first}(C(Q))$. If $|l| \geq 2b$ holds, then we take the b first elements of l and append them to $F(Q)$. Else $|l| < 2b$ holds, so we append l to $F(Q)$, discard r and call $\mathsf{Bias}(Q)$ once.

Bias(Q)

$\mathsf{Bias}(Q)$ improves the inequality of I.7) for Q by at least 1 if Q contains any records. It also ensures that invariant I.8) is maintained. We distinguish two basic cases with respect to $|B(Q)|$, namely $|B(Q)| = 0$ and $|B(Q)| > 0$.

1. $|B(Q)| > 0$: We have two cases depending on if $k_Q \geq 1$ or $k_Q = 0$.

   1) $k_Q = 0$: Let $e = \min(L(Q))$, if it exists. We remove the first record $r_1 = (l_1, \cdot) = \text{first}(B(Q))$ from $B(Q)$. Let l_1' be the non-attrited elements of l_1, under attrition by element e. If $|l_1'| = |l_1|$ holds, nothing is attrited, so we just add $r_1 = (l_1, \cdot)$ at the end of $C(Q)$.

   Else $|l_1'| < |l_1|$ holds, so we set $B(Q) = \emptyset$. If $|l_1'| \geq b$ holds, then we make record r_1 with buffer l_1' into the new last record of $C(Q)$. Else $|l_1'| < b$ holds, so if $|l_1'| + |L(Q)| \leq 3b$ also holds, we add l_1' to $L(Q)$ and discard r_1. Else $|l_1'| + |L(Q)| > 3b$ also holds, so we take the $2b$ first elements of l_1' and $L(Q)$ and put them into r_1, making it the new last record of $C(Q)$.

   2) $k_Q \geq 1$: Let $e = \min(\text{first}(D_1(Q)))$. We remove the first record $r_1 = (l_1, \cdot) = \text{first}(B(Q))$ from $B(Q)$. Let l_1' be the non-attrited elements of l_1, under attrition by element e.

   If $|l_1'| = |l_1|$ or $b \leq |l_1'| < |l_1|$ holds, we just add $r_1 = (l_1', \cdot)$ at the end of $C(Q)$ and if $|l_1'| < |l_1|$ we set $B(Q) = \emptyset$. Else $|l_1'| < b$ hold, we set $B(Q) = \emptyset$, let $r_2 = (l_2, p_2) = \text{first}(D_1(Q))$. If $|l_1'| + |l_2| \leq 4b$ holds, we discard r_1 and prepend l_1' onto l_2 of r_2. Else $|l_1'| + |l_2| > 4b$ holds, so we take the first $2b$ elements of l_1' and l_2 and put them in r_1, making it the new last record of $C(Q)$. If this causes $\min(L(Q)) \leq \min(\text{first}(D_1(Q)))$, we discard all dirty queues.

   If r_1 was discarded, then we have that $|B(Q)| = 0$ and we call Bias recursively, which will not invoke this case again. In all cases the inequality of I.7) for Q is improved by 1.

2. $|B(Q)| = 0$: we have three cases depending on the number of dirty queues, namely cases $k_Q > 1$, $k_Q = 1$ and $k_Q = 0$.

   1) $k_Q > 1$: If $\min(L(Q)) \leq \min(\text{first}(D_{k_Q}(Q)))$ holds, we set $k_Q = k_Q - 1$ and discard $D_{k_Q}(Q)$. This improves the inequality of I.7) for Q by at least 2. Else let $e = \min(\text{first}(D_{k_Q}(Q)))$.

   If $e \leq \min(\text{last}(D_{k_Q-1}(Q)))$ holds, we remove the record $\text{last}(D_{k_Q-1}(Q))$ from $D_{k_Q-1}(Q)$. This improves the inequality of I.7) for Q by 1.

   If $\min(\text{last}(D_{k_Q-1}(Q))) < e \leq \max(\text{last}(D_{k_Q-1}(Q)))$ holds, we remove record $r_1 = (l_1, p_1) = \text{last}(D_{k_Q-1}(Q))$ from $D_{k_Q-1}(Q)$, and let $r_2 = (l_2, p_2) = \text{first}(D_{k_Q}(Q))$. We delete any elements in l_1 that are attrited by e, and let l_1' denote the set of non-attrited elements. If $|l_1'| + |l_2| \leq 4b$ holds, we prepend l_1' onto l_2 of r_2 and discard r_1. Else we take the first $\lfloor (|l_1'| + |l_2|)/2 \rfloor$ elements of l_1' and l_2 and replace r_1 of $D_{k_Q-1}(Q)$ with them. Finally, we concatenate $D_{k_Q-1}(Q)$ and $D_{k_Q}(Q)$ into a single deque. This improves the inequality of I.7) for Q by at least 1.

   Else $\max(\text{last}(D_{k_Q-1}(Q))) < e$ holds and we just concatenate the deques $D_{k_Q-1}(Q)$ and $D_{k_Q}(Q)$, which improves the inequality of I.7) for Q by 1.

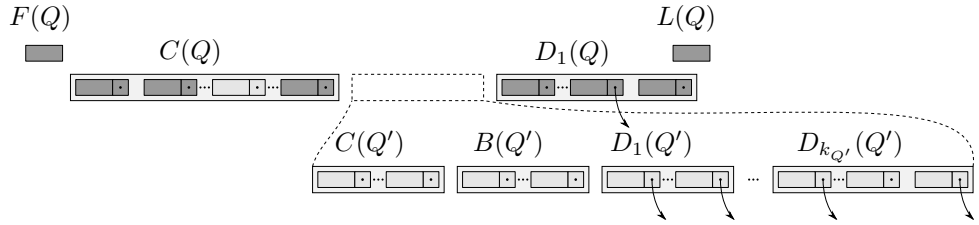

Figure 3.2: Merging I/O-CPQAs Q and Q'. This case can only occur when $B(Q) = \emptyset$ and $k_Q = 1$.

   2) $k_Q = 1$: In this case Q contains only deques $C(Q)$ and $D_1(Q)$. Let $r = (l, p) = \text{first}(D_1(Q))$. If $\min(L(Q)) \leq \min(\text{first}(\text{rest}(D_1(Q))))$ holds, we discard all dirty queues, except for record r of $D_1(Q)$.

   If $\min(L(Q)) \leq \max(l)$ holds, we discard all the dirty deques and let l' be the non-attrited elements of l. If $|l'| + |L(Q)| \leq 3b$ holds, we prepend l' onto $L(Q)$. Else $|l'| + |L(Q)| > 3b$ holds, so we take the first $2b$ elements of l' and $L(Q)$ and make them the new last record of $C(Q)$ and leave the rest in $L(Q)$. This improves the inequality of I.7) for Q by 1.

   Else $\max(\ell) < \min(L(Q))$ holds, so we remove r and insert buffer l into a new record at the end of $C(Q)$. This improves the inequality of I.7) for Q by at least 1. If r is not simple, let the pointer p of r reference I/O-CPQA Q'. We restore I.6) for Q by *merging* I/O-CPQAs Q and Q' into one I/O-CPQA; see Figure 3.2. In particular, let $e = \min(\min(\text{first}(D_1(Q))), \min(L(Q)))$.

   We proceed as follows: If $e \leq \min(Q')$ holds, we discard Q'. The inequality of I.7) for Q remains unaffected. Else if $\min(\text{first}(C(Q'))) < e \leq \max(\text{last}(C(Q'))$ holds, we set $B(Q) = C(Q')$ and discard the rest of Q'. The inequality of I.7) for Q remains unaffected.

   Else if $\max(\text{last}(C(Q')) < e \leq \min(\text{first}(D_1(Q')))$ holds, we concatenate the deque $C(Q')$ at the end of $C(Q)$. If moreover $\min(\text{first}(B(Q'))) < e$

holds, we set $B(Q) = B(Q')$. Finally, we discard the rest of Q'. This improves the inequality of I.7) for Q by $|C(Q')|$.

Else $\min(\text{first}(D_1(Q'))) < e$ holds. We concatenate the deque $C(Q')$ at the end of $C(Q)$, we set $B(Q) = B(Q')$, we set $D_1(Q'), \ldots, D_{k_{Q'}}(Q')$ as the first $k_{Q'}$ dirty queues of Q and we set $D_1(Q)$ as the last dirty queue of Q. This improves the inequality of I.7) for Q by $\Delta(Q') \geq 0$, since Q' satisfied Invariant I.7) before the operation.

3) $k_Q = 0$: If all deques are empty, $L(Q) \neq \emptyset$ and $|F(Q)| \leq 2b$ hold, we take the first b elements of $L(Q)$ and append to $F(Q)$. The inequality of I.7) for Q remains $\Delta(Q) = 0$.

3.4 Analysis

In this section we will prove correctness and I/O-complexity for the I/O-CPQA, we will do this in Subsection 3.4.1. In Subsection 3.4.2 we show how to change the Catenate-and-Attrite operation to concatenate an arbitrary number l of I/O-CPQAs without doing any I/Os, assuming that the I/O-CPQAs fulfill and extra requirement.

3.4.1 Correctness and I/O-complexity

We will now prove correctness and I/O complexity of the operations of the I/O-CPQA. The correctness follows by closely noticing that we maintain invariants I.1)–I.9), which in turn imply that Delete-Min(Q) and Find-Min(Q) always return the minimum element of Q. The $\mathcal{O}(1)$ worst-case I/O bound is trivial as every operation only accesses $\mathcal{O}(1)$ records. Although Bias is recursive, notice that in the case where $|B(Q)| > 0$, Bias only calls itself after making $|B(Q)| = 0$, so it will not end up in this case again.

For the amortized I/O-complexity we will use a potential analysis and in the following we will elaborate on all the operations that modify the I/O-CPQA in order to argue for the amortized bounds. At the end of this subsection we will have proven the following theorem:

Theorem 3.1. *An I/O-CPQA supports Find-Min, Delete-Min, Catenate-and-Attrite and Insert-and-Attrite in $\mathcal{O}(1)$ I/Os per operation. It occupies $\mathcal{O}(\frac{n-m}{B})$ blocks after calling Catenate-and-Attrite and Insert-and-Attrite n times and Delete-Min m times, respectively.*

All operations are supported by a set of ℓ I/O-CPQAs in $\mathcal{O}(\frac{1}{b})$ amortized I/Os, when $M = \Omega(\ell b)$, using $\mathcal{O}(\frac{n-m}{b})$ blocks of space, for any parameter $1 \leq b \leq B$.

Delete-Min

If $|F(Q)| \geq b$ holds after deleting $\min(F(Q))$, then no I/Os are incurred and we only pay an amortized cost of $\leq \frac{3}{b}$ for increasing the potential. Else $|F(Q)| = b - 1$ holds, so $\Phi_F(|F(Q)|) \geq 3$ also holds, which pays for any I/Os in calling Fill and Bias.

Catenate-and-Attrite

If $|Q_1| < b$ holds, then we prepend the non-attrited elements $F'(Q_1)$ onto $F(Q_2)$. So if $|F'(Q_1)| + |F(Q_2)| \leq 4b$ holds, then each element of $F(Q_1)$ has a potential of $\frac{3}{b}$, which is higher than the potential for each element in $\Phi_F(x)$, independent of the value of x. Thus $\Phi(|F(Q_1)|)$ pays for any increase in potential. If instead

$|F'(Q_1)| + |F(Q_2)| > 4b$ holds, then $|F(Q_2)| > 3b$ holds, so

$$\begin{aligned}\Delta\Phi_T &= \left(\frac{3|F(Q_1)|}{b} + \Phi_F(|F(Q_2)|)\right) - (1+1) \\ &\geq \frac{|F'(Q_1)|}{b} + \frac{2(|F'(Q_1)| + |F(Q_2)|)}{b} - 7 > 1\end{aligned}$$

which pays for making the new first record of $C(Q_2)$.

If $|Q_2| < b$ holds, then we have three cases depending on how much of Q_1 is attrited by Q_2. Let $e = \min(Q_2)$ and $r = (l, \cdot) = \text{last}(Q_1)$:

1. $e \leq \min(\text{last}(D_{k_{Q_1}}(Q_1)))$: We discard r which releases 1 potential and have the four cases:

   1) If $e \leq \min(F(Q_1))$: The potential decreases, because we only discard records.

   2) Else if $e \leq \max(\text{last}(C(Q_1)))$: We prepend $F(Q_1)$ onto $C(Q)$ and discard records, which only decreases the potential, since $\Phi_F(x) \geq 1$ when $x \geq b$. Our calls to Bias and Fill are paid for as we discard r.

   3) Else if $e \leq \min(\text{first}(B(Q_1)))$ or $e \leq \min(\text{first}(D_1(Q_1)))$: We set $L(Q_1) = F(Q_2)$ and discard records, which only decreases the potential, since $\Phi_L(x) \leq \Phi_F(x)$ for all x.

   4) Else: If $|L'(Q_1)| + |F(Q_2)| \leq 4b$ we append $F(Q_2)$ to $L'(Q_1)$ and $\Phi_F(|Q_2|)$ pays. Else we make a new dirty queue with one new record, which costs 1 potential and 1 potential to cover the I/Os in Bias. The total potential difference is

   $$\begin{aligned}\Delta\Phi_T &\geq (\Phi_L(|L(Q_1)|) + \Phi(|Q_2|)) - (1+1) \\ &\geq \left(\frac{2(|L'(Q_1)| + |F(Q_2)|)}{b} + \frac{|F(Q_2)|}{b}\right) - 7 \\ &> 1\end{aligned}$$

2. $e \leq \min(L(Q_1))$: We set $L(Q_1') = F(Q_2)$, which again only decreases the potential.

3. $\min(L(Q_1)) < e$: If $|L'(Q_1)| + |F(Q_2)| > 4b$ holds, then if furthermore $|l'| < |l|$ we put the first $4b - |l'|$ elements of $L'(Q_1)$, $F(Q_2)$ and l' into l, with no change in potential. If there are still more than $3b$ elements left in $L'(Q_1)$ and $F(Q_2)$, then we put the first $3b$ elements into a new last record of $D_{k_{Q_1}}(Q_1)$ for a cost of 1 in potential and call Bias for a cost of 1 for I/Os, and leave the remaining $\leq 2b$ elements in $L(Q_1)$ for a cost of ≤ 1. All this is paid for, as the total decrease in potential is

   $$\begin{aligned}\Delta\Phi_T &\geq (\Phi_L(|L(Q_1)|) + \Phi_F(|F(Q_2)|)) - (1+1+1) \\ &= \frac{2|L(Q_1)|}{b} + \frac{3|F(Q_2)|}{b} - 8 \\ &\geq \frac{2(|L'(Q_1)| + |F(Q_2)|)}{b} + \frac{|F(Q_2)|}{b} - 8 > 0\end{aligned}$$

Both Q_1 and Q_2 are large in all the cases (1–4), hence when we concatenate them, we decrease the potential by at least 1, since the number of large I/O-CPQA's decreases by one, which is enough to pay for any other I/Os incurred, also in Bias and Fill. So we only need to argue that the potential does not increase in any of the cases.

1. If $e \leq \min(F(Q_1))$: the potential decreases, since we discard Q_1.
2. Else if $e \leq \max(\text{last}(C(Q_1)))$: we prepend $F(Q_1)$ onto $C(Q_1)$ and $F(Q_2)$ onto $C(Q_2)$, discard and move around records, which only decreases the potential, as $\Phi_F(x) \geq 1$ when $x \geq b$.
3. Else if $e \leq \min(\text{first}(B(Q_1)))$: we prepend $F(Q_2)$ onto $C(Q_2)$, discard and move around records, which only decreases the potential, as $\Phi_F(x) \geq 1$ when $x \geq b$.
4. Else: We make $L'(Q_1)$ and $F(Q_2)$ into the first one or two records of $C(Q_2)$. Since Q_2 is large, $|F(Q_2)| \geq b$ holds, and hence we have that $\Phi_F(|F(Q_2)|) \geq 1$. If we only make one new record, $\Phi_F(|F(Q_2)|)$ pays for it. If we make two records, then $|L'(Q_1)| + |F(Q_2)| > 4b$ holds. So if $|L'(Q_1)| \geq b$ moreover holds, then $\Phi_L(|L(Q_1)|) \geq 1$ pays for the other record. Else $|L'(Q_1)| < b$ holds, but then $|F(Q_2)| > 3b$ also holds, so

$$\begin{aligned}
&\Phi_L(|L(Q_1)|) + \Phi_F(|F(Q_2)|) \\
&= \frac{|L(Q_1)|}{b} + \frac{2|F(Q_2)|}{b} - 5 \\
&\geq \frac{|L'(Q_1)| + |F(Q_2)|}{b} + \frac{|F(Q_2)|}{b} - 5 > 2
\end{aligned}$$

   which pays for both new records.

Insert-and-Attrite

The total cost is $\mathcal{O}(\frac{1}{b})$ I/Os amortized, since creating a new I/O-CPQA with only one element and calling Catenate-and-Attrite only costs as much.

Fill

Any I/Os incurred, are prepaid by a decrease in potential made in the procedure calling Fill, so we only need to argue that the potential does not increase. If $|F(Q)| < b$ and $|Q| \geq b$ then we append at most $2b$ elements to $F(Q)$, hence $\Phi_F(|F(Q)|)$ will only decrease.

Bias

All I/Os have been paid for by a decrease in potential caused by the caller of Bias. So we only need to argue that the potential does not increase because of Bias.

1. $|B(Q)| > 0$: We discard, move around and merge records, but we do not create new ones. Thus the potential will only decrease.
2. $|B(Q)| = 0$: We follow the cases of Bias.
   1) $k_Q > 1$: We again discard and move around records, and rearrange their elements, but we do not create new records, so the potential will only decrease.
   2) $k_Q = 1$: Let $r = (l, p) = \text{first}(D_1(Q))$. If $\min(L(Q)) \leq \max(l)$ holds, we might prepend l' onto $L(Q)$, but only if $|l'| + |L(Q)| \leq 3b$. This will not increase the potential of $L(Q)$ by more than 1, and r pays for that. For the rest of the case, we discard and move around records and rearrange their elements, but we do not create new records, so the potential only decreases.

3) $k_Q = 0$: If we append the first b elements of $L(Q)$ onto $F(Q)$, then $|F(Q)| \leq 2b$ holds, so $\Phi_F(|F(Q)|)$ can only decrease. Likewise, when taking at most b elements from $L(Q)$, then $\Phi_L(|L(Q)|)$ will only decrease.

3.4.2 Catenating a Set of I/O-CPQAs

Define the *critical records* of an I/O-CPQA Q, to be the first three records of $C(Q)$, last($C(Q)$), first($B(Q)$), first($D_1(Q)$), last($D_{k_Q}(Q)$) and last(front($D_{k_Q}(Q)$)), if it exists. Otherwise last($D_{k_Q-1}(Q)$) is critical.

Lemma 3.1. *A set of I/O-CPQAs Q_i for $i \in [1, \ell]$ can be concatenated into a single I/O-CPQA without doing any I/Os, by calling only* Catenate-and-Attrite *operations, provided that for all i:*

1. *$\Delta(Q_i) \geq 2$ holds, unless Q_i contains only one record, in which case $\Delta(Q_i) = 0$ or Q_i contains only two records, in which case $\Delta(Q_i) = +1$ suffices.*
2. *The critical records of Q_i are loaded in main memory.*

Proof. The algorithm considers the I/O-CPQAs Q_i in decreasing index i (from right to left). It first sets $Q^\ell = Q_\ell$ and constructs the temporary I/O-CPQA $Q^{\ell-1}$ by calling Catenate-and-Attrite($Q_{\ell-1}$,Q^ℓ). After the end of the sequence of operations, the resulting I/O-CPQA Q^1 is the concatenation of all I/O-CPQAs Q_i.

To avoid any I/Os during the sequence of Catenate-and-Attrite's, we ensure that Bias and Fill are not called, and that no more than the critical records need to be already loaded into memory. To avoid calling Bias we maintain the following invariant during the sequence of concatenations.

I.10) Each I/O-CPQAs $Q^i, i \in [1, \ell]$ constructed during the sequence of catenations is in state at least $+1$ unless it consists only of the front buffer in which case it is in state 0.

We prove the invariant inductively on the sequence of operations. Let the invariant hold for Q^{i+1} and let Q^i be constructed by Catenate-and-Attrite(Q_i,Q^{i+1}). In the following, we parse the cases of the Catenate-and-Attrite algorithm assuming that $e = \min(Q^{i+1})$.

If $|Q_i| < b$ holds, then Bias is not invoked and the state of Q^{i+1} remains ≥ 1 or is increased by 1.

If $|Q^{i+1}| < b$ and $|Q_i| \geq b$ then we have to go through the three respective cases, where $r = (l, \cdot) = \text{last}(Q_i)$.

1. If $e \leq \min(r)$: if record r exists then the state of Q_i is increased by 1 and it becomes ≥ 3.
   1) If $e \leq \min(F(Q_1))$: Since Bias is not called I.10) holds trivially.
   2) Else if $e \leq \max(\text{last}(C(Q_1)))$: Q^i is constructed as before and we then do the following. Since $k_{Q^i} = 0$, we take out the first two records of $B(Q^i)$ which are critical since they came from $F(Q_i)$ and first(Q_i). Then, we fill $F(Q^i)$ with one of these records provided that no attrition was enforced by $L(Q^i)$. In this case, the state of Q^i is ≥ 1 and the invariant holds. If attrition took place then $B(Q^i)$ is discarded and the at most two records of $C(Q^i)$ and the buffer $L(Q_i)$ are combined (notice that all of them are critical) to make Q^i consisting only of records in $F(Q^i)$ and $C(Q^i)$ and thus I.10) holds.
   3) Else if $e \leq \min(\text{first}(B(Q_1)))$ or $e \leq \min(\text{first}(D_1(Q_1)))$: Since Bias is not called I.10) holds trivially.

4) Else: the state at the end is ≥ 0, since the state of Q_i was ≥ 2 by the induction hypothesis. To restore the invariant that the state of Q^i should be ≥ 1 we check whether $\text{last}(D_{k_{Q_i}}(Q^i))$ is attrited or not by the new dirty queue. Since both are critical this can be done with no I/Os and thus the state of Q^i is increased to ≥ 1.

2. Else if $e \leq \min(L(Q_1))$: since we do not call Bias, I.10) holds trivially.

3. Else $\min(L(Q_1)) < e$: the state of Q_i is only reduced by 1 which makes the state of Q^i being ≥ 1 which is sufficient to maintain I.10).

Now we move to the more general case where $|Q_1| \geq b$ and $|Q_2| \geq b$.

1. If $e \leq \min(F(Q_1))$: we do not call Bias, so I.10) holds trivially.

2. Else if $e \leq \max(\text{last}(C(Q_1)))$: To increase the state of Q^i from -2 to ≥ 1 we do as follows. We extract the 4 records of $B(Q^i)$, which incurs no I/Os since all four of them are critical (the first was from $F(Q_i)$ and the other three from the first 3 critical records of $C(Q_i)$). If no attrition was enforced by $e = \min(Q^{i+1})$, then the state of Q^i is ≥ 1. If attrition is enforced then there are not that many records in $B(Q^i)$, then Q^{i+1} is reconstructed (just prepend $(l, \cdot)$, which was the old $\text{first}(C(Q^{i+1}))$, onto $C(Q^{i+1})$ and then prepend the non-attrited records (at most 4 records) from Q_i onto $C(Q^{i+1})$ remaking $F(Q^{i+1})$. At the end of this process, the new I/O-CPQA Q^i has state at least equal to Q^{i+1}, which is ≥ 1 by the induction hypothesis and hence I.10) holds.

3. Else if $e \leq \min(\text{first}(B(Q_1)))$ or $e \leq \min(\text{first}(D_1(Q_1)))$: we will only consider the case where $k_{Q_i} = 0$ before the concatenation, since otherwise the state of Q^i will be equal or larger to the state of Q_i, which by the inductive hypothesis is ≥ 2. Since Q_i must be in state ≥ 2, there are either at least three records in $C(Q_i)$, in which case I.10) holds and the case is terminated. Otherwise, exactly two records exist in $C(Q_i)$ and $B(Q_i)$ is non-empty or there are less than two records in $C(Q_i)$ (so the state of Q_i is ≥ 1 or 0) and $B(Q_i)$ is empty. In the case where two records exist in $C(Q_i)$ and $B(Q_i)$ is non-empty: if $\text{first}(B(Q_i))$ is not attrited by e we put this record into $C(Q_i)$ and now the final I/O-CPQA Q^i has state ≥ 1. Otherwise, we restructure Q^{i+1} (as done in the previous case) and prepend the non-attrited elements of Q_i onto Q^{i+1} resulting in an I/O-CPQA with state at least ≥ 1 since this was the state of Q^{i+1}. We follow exactly the same approach in the latter case where $C(Q_i)$ contains less than two records and $B(Q_i)$ is empty.

4. Else: the algorithm works exactly as before with the following exception. At the end, Q^i will be in state ≥ 0, since we added the deque $D_{k_{Q^{i+1}}+1}$ with a new record and the inequality of I.7) is aggravated by 2. To restore the invariant we apply Case 2. 1) of Bias. This step requires access to records $\text{last}(D_{k_{Q^i}-1})$ and $\text{first}(D_{k_{Q^i}})$. These records are both critical, since the former corresponds to $\text{last}(D_{k_{Q^{i+1}}})$ and the latter to $\text{first}(C(Q^{i+1}))$. In addition, $\text{Bias}(Q^{i+1})$ need not be called, since by the invariant, Q^{i+1} was in state ≥ 1 before the removal of $\text{first}(C(Q^{i+1}))$. In this way, we improve the inequality for Q^i by 1 and hence I.10) holds.

□

Chapter 4

Dynamic Planar Skyline Queries

In this chapter we present two linear size dynamic data structures for skyline queries. The first is for top-open queries and uses I/O-CPQAs as an essential ingredient. The second is for 4-sided queries and it uses our top-open structure as a black box. Finally we create a point and query set which we use to give two lower bounds: a space lower bound in the framework of Chazelle and Liu [CL04] and a query lower bound in the indexability model of [HKM+02].

The dynamic top-open data structure uses linear space and can be build in a linear number of I/Os, assuming the input is pre-sorted. The data structure supports top-open queries in $\mathcal{O}(\log_{2B^\epsilon} \frac{n}{B} + \frac{k}{B^{1-\epsilon}})$ I/Os and updates in $\mathcal{O}(\log_{2B^\epsilon} \frac{n}{B})$ I/Os, for any parameter $0 \leq \epsilon \leq 1$. Our upper bound is inspired by the approach of Overmars and van Leeuwen [OvL81] for maintaining the planar skyline in the pointer machine. As a brief review of [OvL81], a dynamic binary base tree indexes the x-coordinates of P, and every internal node stores the skyline of the points in its subtree using a secondary search tree. More specifically, the skyline of an internal node is $(L \backslash L') \cup R$, where L (resp. R) is the skyline of its left (resp. right) child node, and L' is the set of points in L dominated by the leftmost (and thus also highest) point of R. We use I/O-CPQAs from Chapter 3 to maintain the sets L and R in internal nodes, and we generalize this to (a, b)-trees.

The dynamic 4-sided data structure uses linear space and answers queries in $\mathcal{O}((\frac{n}{B})^\epsilon + \frac{k}{B})$ I/Os and support updates in amortized $\mathcal{O}(\log \frac{n}{B})$ I/Os. We utilize an (a, b)-tree augmented with our right-open structure (which is symmetric to the top-open structure) on internal nodes and then answers the 4-sided queries by issuing right-open queries.

Our point and query sets are given in the framework proposed by Chazelle and Liu [CL04] in which we give two lower bounds. The first lower bound is a space lower bound saying if we support queries in $\mathcal{O}(\log^\gamma n + k)$ time then we need $\Omega(n \frac{\log n}{\log \log n})$ space. The second is a query lower bound in the indexability model, with the indivisibility assumption, saying that if we want $\mathcal{O}(\frac{n}{B})$ space then the query will spent $\Omega((\frac{n}{B})^\epsilon + \frac{k}{B})$ I/Os to report the skyline within a query range.

4.1 Top-Open Structure

Our approach is based on I/O-CPQAs, as described in Chapter 3. We observe that attrition can be utilized to maintain the internal node skylines in [OvL81], after

mirroring the y-axis. To explain this, let us first map the input set P to its y-mirrored counterpart $\widetilde{P} = \{(x_p, -y_p) \mid (x_p, y_p) \in P\}$. In the context of PQAs, we will interpret each point $(\widetilde{x}_p, \widetilde{y}_p) \in \widetilde{P}$ as an *element* with "key" value $\widetilde{y}_p$ that is inserted at "time" $\widetilde{x}_p$. To formalize the notion of time, we define the $<_x$-ordering of two elements $\widetilde{p}, \widetilde{q} \in \widetilde{P}$ to be $\widetilde{p} <_x \widetilde{q}$, if and only if $\widetilde{x}_p < \widetilde{x}_q$ holds. We see that element $\widetilde{p} \in \widetilde{P}$ is attrited by element $\widetilde{q} \in \widetilde{P}$, i.e., $\widetilde{y}_p \geq \widetilde{y}_q$, if and only if point $p \in P$ is dominated by point $q \in P$. We have that

$$\begin{aligned} & \widetilde{q} \text{ attrites } \widetilde{p} \\ \Leftrightarrow \quad & \widetilde{x}_p \leq \widetilde{x}_q \wedge \widetilde{y}_p \geq \widetilde{y}_q \\ \Leftrightarrow \quad & x_p \leq x_q \wedge -y_p \geq -y_q \\ \Leftrightarrow \quad & x_p \leq x_q \wedge y_p \leq y_q \\ \Leftrightarrow \quad & q \text{ dominates } p \end{aligned}$$

From this we see that if we build a PQA on the point set $\widetilde{P}$ the non-attrited points in the PQA are exactly the skyline points. See Figure 4.1 for a geometric illustration of the mirroring transformation and the effects of attrition.

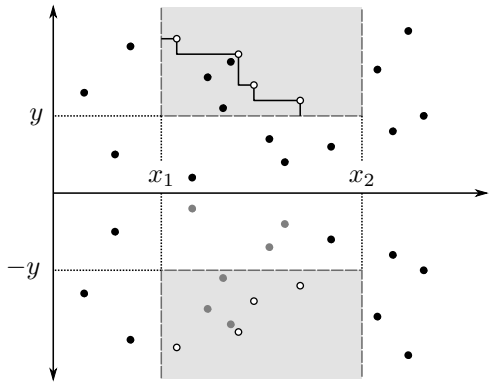

Figure 4.1: The skyline problem (above) mirrored to the attrition problem (below). White points are reported for the gray query area $[x_1, x_2] \times [y, \infty[$ and are non-attrited, while gray points are attrited within $[x_1, x_2]$.

Thus, we index the $<_x$-ordering of $\widetilde{P}$ in a $(2B^\epsilon, 4B^\epsilon)$-tree, for a parameter $0 \leq \epsilon \leq 1$, and employ I/O-CPQAs as secondary structures, such that the I/O-CPQA at an internal node is simply the concatenation of its children's I/O-CPQAs. To obtain logarithmic query and update I/Os, this sequence of consecutive Catenate-and-Attrite operations at an internal node must be performed in $\mathcal{O}(1)$ I/Os, which we can do thanks to Lemma 3.1. The presented I/O-CPQAs are *ephemeral* (not persistent), and thus the supported operations are *destructive*, as they destroy the initial configuration of the structure. This only preserves the I/O-CPQA that is the final result of all concatenations and resides at the root of the base tree. However, in order to support top-open queries efficiently, accessing the I/O-CPQAs at the internal nodes is required. This is made possible by non-destructive operations. Therefore, we render the I/O-CPQAs confluently persistent by merely replacing the catenable deques, which are used as black boxes in our ephemeral construction, with real-time purely functional catenable deques [KT99]. Since the imposed overhead is $\mathcal{O}(1)$ worst-case I/Os, confluently persistent I/O-CPQAs ensure the same I/O bounds as their ephemeral counterparts.

4.1.1 Structure

The data structure consists of a base tree, implemented as a dynamic $(a, 2a)$-tree where the leaves store between k and $2k$ elements. We set $a = \lceil 2B^\epsilon \rceil$ and $k = B$, for a given $0 \leq \epsilon \leq 1$. The base tree indexes the $<_x$-ordering of $\widetilde{P}$, and is augmented with confluently persistent I/O-CPQAs with buffer size $b = B^{1-\epsilon}$ as secondary structures. See Figure 4.2. In particular, after constructing the base tree, we augment it with secondary I/O-CPQAs in a bottom-up manner, as follows. For every leaf we make one I/O-CPQA over its elements, and execute an appropriate amount of Bias operations, such that the state of the I/O-CPQA satisfies Lemma 3.1. We associate the I/O-CPQA with the leaf. In a second pass over the leaves, we gather its critical records into a *representative block* in its parent. The procedure continues one level above. For every internal node u, we access the representative blocks that contain the critical records of the children's I/O-CPQAs of u, and Catenate-and-Attrite them into a new I/O-CPQA as implied by Lemma 3.1. We execute Bias on the I/O-CPQA enough times such that its state also satisfies Lemma 3.1. We associate the I/O-CPQA with node u and put its critical records in the representative block of u's parent. After the level has been processed, the representative blocks for the I/O-CPQAs associated with the nodes of the current level are ready for use in the level above. The augmentation ends at the root node of the base tree. We will ensure that our algorithms access the I/O-CPQA associated with a node through the representative block stored at the parent of the node. Thus, it will suffice to explicitly store only the representative blocks of the children in every internal node and the I/O-CPQAs of the leafs.

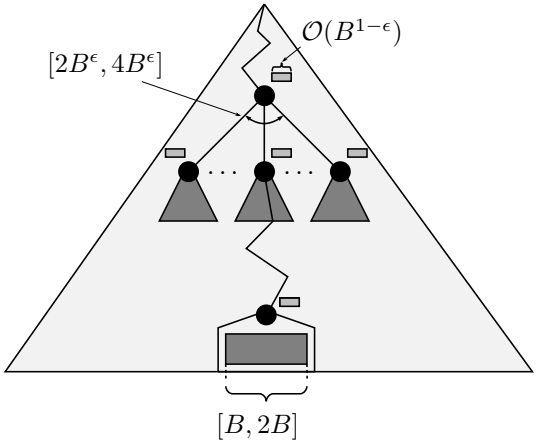

Figure 4.2: The skyline structure is a $(\lceil 2B^\epsilon \rceil, \lceil 4B^\epsilon \rceil)$-tree augmented with I/O-CPQAs on the internal nodes and the leafs.

4.1.2 Invariants

We are using a $(\lceil 2B^\epsilon \rceil, \lceil 4B^\epsilon \rceil)$-tree augmented with I/O-CPQAs. We will maintain the invariants from Subsection 1.5.1 for the (a, b)-tree along with the invariants from Chapter 3 for the I/O-CPQAs.

4.1.3 Operations

Having defined the (a, b)-tree and the invariants about how to augment it, we are ready to define the update and query operations.

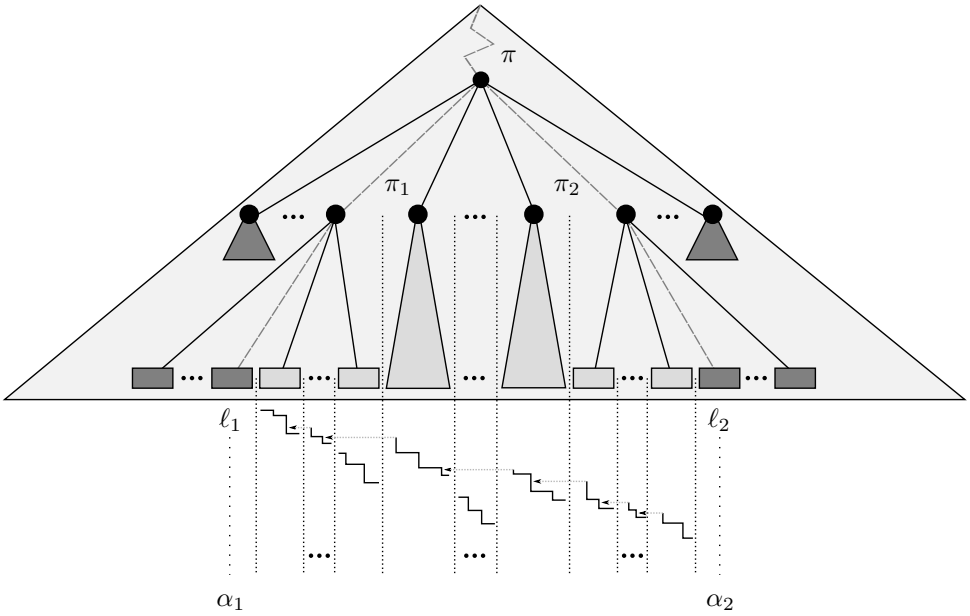

Figure 4.3: A top-open query is answered by calling Catenate-and-Attrite on ℓ_1, the gray subtrees and ℓ_2 returning one I/O-CPQA and then calling Delete-Min on it.

Updates

To insert (delete) a point p into (from) P, we insert (delete) $\widetilde{p} = (\widetilde{x}_p, \widetilde{y}_p)$ in (from) the structure. In particular, we first find the leaf to insert into (delete from) that contains the predecessor of $\widetilde{x}_p$ (contains $\widetilde{x}_p$), by a top-down traversal of the path from the root of the base tree. For every node u on the path, we also discard the part of its representative block corresponding to the child that the search path goes into, and u's associated I/O-CPQA by discarding the changes from the operations that created it. Next we insert (delete) $\widetilde{p}$ into (from) the accessed leaf, and rebalance the base tree by executing the appropriate splits and merges on the nodes along the path in a bottom-up manner. Moreover, we recompute the I/O-CPQA of every accessed node on the path, as described above. See Figure 4.2.

Queries

To report the skyline points of P that reside within a given top-open query range $[\alpha_1, \alpha_2] \times [\beta, \infty[$, we first traverse top-down the two search paths $\widetilde{\pi_1} = \pi\pi_1$ and $\widetilde{\pi_2} = \pi\pi_2$ from the root of the base tree to the leaves ℓ_1 and ℓ_2 containing α_1 and α_2 in the $<_x$-ordering, respectively. See Figure 4.3. Let node u be on the path $\pi_1 \cup \pi_2$, and let $c(u)$ be the children nodes of u whose subtrees are fully contained within $[\alpha_1, \alpha_2]$. For every node u, we load its representative block into memory in order to access the critical records of the I/O-CPQAs associated with $c(u)$ and to Catenate-and-Attrite them into a temporary I/O-CPQA, as implied by Lemma 3.1. We consider the temporary I/O-CPQAs over the nodes u and the I/O-CPQAs of the leaves ℓ_1 and ℓ_2 from right to left, and we Catenate-and-Attrite them into one auxiliary I/O-CPQA. The I/O-CPQAs for ℓ_1 and ℓ_2 are created only on the points within the x-range $[\alpha_1, \alpha_2]$ in $\mathcal{O}(1)$ I/Os. See Figure 4.3.

To report the skyline points within the query range, we call Delete-Min on the

auxiliary I/O-CPQA. The procedure stops as soon as a point with $\widetilde{y}_p > -\beta$ is returned, or when the auxiliary I/O-CPQA becomes empty.

4.1.4 Analysis

We will now analyse the space usage, pre-processing, update and query I/O complexity of the skyline structure which will serve as the proof of Theorem 4.1 below.

Theorem 4.1. *There is a linear-size dynamic data structure on n points in $\mathbb{R}^2$ that supports top-open range skyline queries in $\mathcal{O}(\log_{2B^\epsilon} \frac{n}{B} + \frac{k}{B^{1-\epsilon}})$ I/Os when k points are reported, and updates in $\mathcal{O}(\log_{2B^\epsilon} \frac{n}{B})$ I/Os for any parameter $0 \leq \epsilon \leq 1$. The structure can be constructed in $\mathcal{O}(\frac{n}{B})$ I/Os, assuming an initial sorting on the input points' x-coordinates.*

Space and Preprocessing

Lets first look at the space usage of the skyline structure. Since every leaf contains $\mathcal{O}(B)$ elements, the base tree has $\mathcal{O}(\frac{n}{B})$ leaves and thus also $\mathcal{O}(\frac{n}{B})$ internal nodes. Every internal node has $\Theta(B^\epsilon)$ children, each associated with an I/O-CPQA with $\mathcal{O}(1)$ critical records of size $\mathcal{O}(B^{1-\epsilon})$. Thus the representative blocks stored in the internal node occupy $\mathcal{O}(1)$ blocks of space. Thus the structure occupies $\mathcal{O}(\frac{n}{B})$ blocks in total.

We will now look at the pre-processing required to build the skyline structure on a non-empty point set P. Assume that $\widetilde{P}$ is already sorted by the $<_x$-ordering. The leaves' I/O-CPQAs are created in $\mathcal{O}(1)$ I/Os, since they contain at most $\mathcal{O}(B)$ elements. All representative blocks are created in $\mathcal{O}(\frac{n}{B})$ I/Os. To create the internal nodes' I/O-CPQAs, we need only $\mathcal{O}(1)$ I/Os to access the representative blocks and to execute Bias on the resulting I/O-CPQA. Its representative blocks residing in memory thus are written to disk in $\mathcal{O}(1)$ I/Os. Thus the total pre-processing cost is $\mathcal{O}(\frac{n}{B})$ I/Os, assuming the input is already pre-sorted.

Update Cost

When we perform an update of the skyline structure we spend $\mathcal{O}(1)$ I/Os on each node to first discard all the I/O-CPQAs on the path to the leaf. Then we spend $\mathcal{O}(1)$ I/Os to rebalance every accessed node on the path from the leaf to the root. We do splits and joins and recompute each nodes secondary structures, i.e., concatenating the children's I/O-CPQAs into one I/O-CPQA , calling Bias on it and storing its critical records in the representative block of its parent. The total update cost is $\mathcal{O}(\log_{2B^\epsilon} \frac{n}{B})$ I/Os, in the worst-case, as this is also the height of the tree.

Query Cost

There are $\mathcal{O}(\log_{2B^\epsilon} \frac{n}{B})$ nodes on the search paths $\pi_1 \cup \pi_2$ to the leafs ℓ_1 and ℓ_2. We spend $\mathcal{O}(1)$ I/Os to access the representative block of each node and concatenating its children within $[\alpha_1, \alpha_2]$ together, by use of Lemma 3.1. After this, the construction of the auxiliary I/O-CPQA by concatenating the auxiliary I/O-CPQAs for all subtrees within $[\alpha_1, \alpha_2]$ costs $\mathcal{O}(\log_{2B^\epsilon} \frac{n}{B})$ I/Os. Finally reporting the k output points costs $\mathcal{O}(\frac{k}{B^{1-\epsilon}} + 1)$ I/Os, since the buffer sizes of the I/O-CPQAs are $\mathcal{O}(B^{1-\epsilon})$. Therefore the query takes $\mathcal{O}(\log_{2B^\epsilon} \frac{n}{B} + \frac{k}{B^{1-\epsilon}})$ I/Os in total.

4.2 4-Sided Structure

In this section we will use our dynamic top-open structure to develop a linear size dynamic 4-sided structure that can answer queries in $\mathcal{O}\left(\left(\frac{n}{B}\right)^{\epsilon} + \frac{k}{B}\right)$ I/Os and support updates in amortized $\mathcal{O}\left(\log \frac{n}{B}\right)$ I/Os.

4.2.1 Structure

We maintain an augmented $(f, 2f)$-tree T on the x-coordinates of the points in P where $f = \frac{(\frac{n}{B})^{\epsilon}}{\log \frac{n}{B}}$. Each leaf node of T has capacity $k \in [B, 2B]$, and each internal node has $\Theta(f)$ child nodes. For a node u in T, let $P(u)$ be the set of points whose x-coordinates are in the subtree of u. We augment each node u with with a right-open[1] structure $R(u)$ of Theorem 4.1.

4.2.2 Invariants

For the $(f, 2f)$-tree T we maintain the invariants from Subsection 1.5.1 along with the invariants of Section 4.1, required to maintain the augmentation of the $(f, 2f)$-tree with the right-open structures.

4.2.3 Operations

Having described the structure and the invariants we are now ready to describe the query and update operations.

Queries

Given a 4-sided query with search rectangle $Q = [\alpha_1, \alpha_2] \times [\beta_1, \beta_2]$, we find the leaf nodes ℓ_1 and ℓ_2 of T containing the successor and predecessor of α_1 and α_2 respectively, among the x-coordinates indexed by T. See Figure 4.4. If $\ell_1 = \ell_2$, solve the query by loading the $\mathcal{O}(B)$ points of leaf ℓ_1 into memory and answer the query internally.

We now consider the case where $\ell_1 \neq \ell_2$. Let π_1 (π_2) be the path from the lowest common ancestor of ℓ_1 and ℓ_2 to ℓ_1 (ℓ_2). Let u be a node on the path $\pi_1 \cup \pi_2$ and let $c(u)$ be the children of u that are fully contained within $[\alpha_1, \alpha_2]$, these are the light gray subtrees in Figure 4.4.

Find the skyline of $P(\ell_2) \cap Q$, let β^* be the y-coordinate of the highest point in this skyline. Next we process the children $v \in c(u)$ for each node u on the path $\pi_1 \cup \pi_2$, from right-to-left. We perform a right-open query with $]-\infty, \infty[\times[\beta^*, \beta_2]$ on $R(v)$, and output all the points retrieved. If the query returns at least one point, update β^* to the y-coordinate of the highest point returned. Finally, issue a 4-sided query with $[\alpha_1, \alpha_2] \times [\beta^*, \beta_2]$ on the leaf ℓ_1.

Updates

To insert (delete) a point p into (from) P, first descend a root-to-leaf path π to the leaf node ℓ of T where p_x should be inserted into (deleted from). For each internal node u along π, insert (delete) p into (from) $R(u)$. Next, update the base tree T by inserting (deleting) p_x. If an internal node u is split (joined), we construct $R(u')$ for each new node u' from scratch by simply inserting into $R(u')$ all the relevant points.

We reconstruct the entire structure after $\Omega(n)$ updates to make sure that the height does not change until T is rebuilt next time.

[1] Notice that top-open and right-open queries are symmetric.

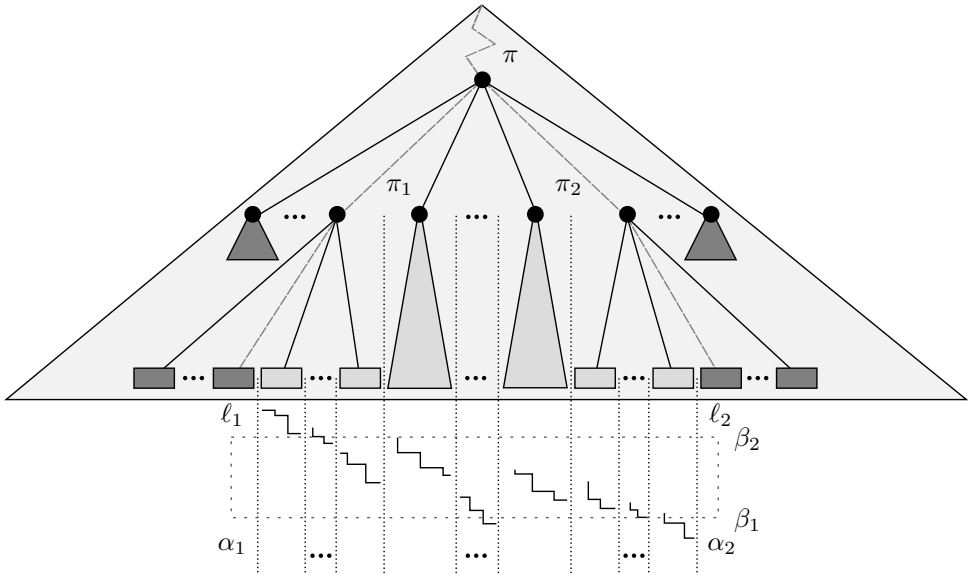

Figure 4.4: We solve a 4-sided query by doing $\mathcal{O}(f)$ queries on the augmenting right-open structures.

4.2.4 Analysis

We will now analyse the I/O bounds of the 4-sided structure, at the end of this subsection we will have proved the following Theorem.

Theorem 4.2. *There is a linear-size structure on n points in $\mathbb{R}^2$ such that, 4-sided range skyline queries can be answered in $\mathcal{O}((\frac{n}{B})^\epsilon + \frac{k}{B})$ I/Os, where k is the number of reported points. The structure can be updated in $\mathcal{O}(\log \frac{n}{B})$ I/Os amortized.*

Space Usage

Since T has fanout $\Theta(f) = \Theta\left(\frac{(\frac{n}{B})^\epsilon}{\log \frac{n}{B}}\right)$ the height h of the tree will be

$$\mathcal{O}\left(\log_f \frac{n}{B}\right) = \mathcal{O}\left(\frac{\log \frac{n}{B}}{\log\left(\frac{(\frac{n}{B})^\epsilon}{\log \frac{n}{B}}\right)}\right) = \mathcal{O}(1).$$

The right-open structures of all nodes at the same level of T consume a total of $\mathcal{O}(\frac{n}{B})$ blocks of space and as T only has a constant number of levels, the total space usage is $\mathcal{O}(\frac{n}{B})$ blocks.

Query Cost

The correctness follows as we first find the skyline of ℓ_2 then we find the skyline for each subtree fully contained in $[\alpha_1, \alpha_2]$ from right-to-left by querying the secondary structures taking into account the leftmost and highest skyline point returned from the previously queried subtree and ℓ_2. Likewise when querying ℓ_1 we take into account the highest skyline point from the subtrees and ℓ_2.

In T we first find the leafs ℓ_1 and ℓ_2 in $\mathcal{O}(hf)$ I/Os. Then we answer the query by making $\Theta(f)$ queries to the right-open structure of the $\Theta(f)$ subtrees fully contained

within the x-interval $[\alpha_1, \alpha_2]$ on each of the $h = \mathcal{O}(1)$ levels of the tree T. Each query reports the skyline points within its subtree in $\mathcal{O}(\log \frac{n}{B} + \frac{k}{B})$ I/Os. The total I/O cost is then

$$\mathcal{O}\left(hf \log \frac{n}{B} + \frac{k}{B}\right) = \mathcal{O}\left(\left(\frac{\left(\frac{n}{B}\right)^{\epsilon}}{\log \frac{n}{B}}\right) \log \frac{n}{B} + \frac{k}{B}\right) = \mathcal{O}\left(\left(\frac{n}{B}\right)^{\epsilon} + \frac{k}{B}\right).$$

Update Cost

It is trivial to argue for the correctness of the update as we clearly maintain the augmenting structures doing the update.

The navigation to the leaf ℓ costs $\mathcal{O}(\log f)$ I/Os on each of the h levels. Therefore the total cost of the navigation is $\mathcal{O}(h \log \frac{n}{B})$ I/Os. Updating each of the right-open structures on the path to ℓ costs $\mathcal{O}(\log \frac{n}{B})$ I/Os. When rebalancing T by splitting and joining nodes, we simply rebuild the right-open structure $R(u)$ for the new node u in question by inserting all the points in $R(v)$ of the children v, one by one, this will take $\mathcal{O}(|S(u)| \log \frac{n}{B})$ I/Os if the subtree of the new node u has size $S(u)$. The factor $|S|$ is amortized away, as mentioned in Subsection 1.5.1, because a node will only split or join after a constant fraction of the points in its subtree have been deleted or inserted. The total amortized cost of an update is then $\mathcal{O}(\log \frac{n}{B})$.

After $\Theta(n)$ updates we build a new structure T' from T in order to preserve that the height of the tree is $\mathcal{O}(1)$. This will take $\mathcal{O}(n \log \frac{n}{B})$ I/Os. So amortized this rebuilding of T gives an additive overhead of $\mathcal{O}(\log \frac{n}{B})$ to the updates.

4.3 Lower Bounds

In this section we will prove two lower bounds for anti-dominance queries as these are not symmetric nor subsumed by top-open queries. It would be nice if they could be answered in $\mathcal{O}(\log_B n + \frac{k}{B})$ I/Os by a linear-size structure. Unfortunately, we will prove its impossibility. We prove that anti-dominance, left-open and 4-sided queries are just as hard as each other, by giving a lower bound for anti-dominance queries that match our upper bound for 4-sided queries in Section 4.2. The first is a space lower bound in the *PM* model, it proves that if we want polylogarithmic queries $\mathcal{O}(\log^{\gamma} n + k)$ then we will need super-linear space $\Omega(n \frac{\log n}{\log \log n})$. The second is a query lower bound in the Indexability Model of [HKM+02], it will show that if we require our data structure to use linear space $\mathcal{O}(\frac{n}{B})$ in the *EM* model then anti-dominance queries will require $\Omega((\frac{n}{B})^{\epsilon} + \frac{k}{B})$ I/Os. Both lower bounds have, as the essential ingredient, a point and query set inspired by the low-discrepancy point sets proposed by Chazelle and Liu [CL04], which we will describe next in Subsection 4.3.1.

4.3.1 (ω, λ)-input

Before formally defining the (ω, λ)-input we will first recall the formal setting and definitions of [CL04] in which we will phrase our (ω, λ)-input and one of our lower bounds. In the *PM* model, as described in Subsection 1.1.3, a data structure that stores a data set S and supports range reporting queries for a query set $\mathcal{Q}$, can be modelled as a directed graph G of bounded out-degree with some nodes being *entry nodes*. In particular, every node in G may be assigned an element of S or may contain some other useful information. For a query range $Q_i \in \mathcal{Q}$, the algorithm navigates over the edges of G in order to locate all nodes that contain the answer to the query.

The algorithm may also traverse other nodes. The time complexity of reporting the output of Q_i is at least equal to the number of nodes accessed in graph G for Q_i.

Given a directed graph G modelling a data structure in the *PM*, Chazelle and Liu [Cha90, CL04] define two properties of the graph G.

Definition 4.1 ((α,ω)-*effective* from [CL04, Definition 2.1])**.** *A search structure G for a data set S is* (α,ω)-effective*, with α being a positive constant and ω an additive overhead, if for any query q, we have $|G(q)| \leq \alpha(k+\omega)$. Here $G(q)$ is the set of nodes visited in G while answering query q and k is the number of objects in S intersected by q.*

Definition 4.2 ((m,ω)-*favorable* from [CL04, Definition 2.2])**.** *A collection of queries $\mathcal{Q} = \{Q_i\}$ is* (m,ω)-favorable *with $m > 1$ for S, if $\mathcal{Q}$ satisfies the relevance and independence conditions:*

- *Relevance: $|S \cap Q_i| \geq \omega$, for any $Q_i \in \mathcal{Q}$.*
- *Independence: $|S \cap Q_{i_1} \cap \ldots \cap Q_{i_m}| = \mathcal{O}(1)$ for all $i_1 < \ldots < i_m$.*

Intuitively, the first part of this property requires that the size of the output is large enough (at least ω) so that it dominates the additive factor of ω in the time complexity. The second part requires that the query outputs have minimum overlap, in order to force G to be large, without many nodes containing the output of many queries. The following lemma exploits these properties to provide a lower bound on the minimum size of G:

Lemma 4.1 (From [CL04, Lemma 2.3])**.** *For an (α,ω)-effective graph G for the data set S, and for an (m,ω)-favorable set of queries $\mathcal{Q}$, the graph G contains $\Omega(\frac{|\mathcal{Q}|\omega}{\alpha})$ nodes, for constant m and α and for any large enough ω.*

Let S be a set of n points in $\mathbb{R}^2$. Let $\mathcal{Q} = \{Q_i \subseteq \mathbb{R}^2\}$ be a set of orthogonal 2-sided query ranges $Q_i = [q_{i_x}, \infty[\times[q_{i_y}, \infty[\subseteq \mathbb{R}^2$. Query range Q_i is the subspace of $\mathbb{R}^2$ that dominates a given point $q_i \in \mathbb{R}^2$ in the positive x- and y- direction (the "upper-right" quadrant defined by q_i). Let $S_i = S \cap Q_i$ be the set of all points in S that lie in the range Q_i. An *inverse anti-dominance reporting query* Q_i contains the points of S_i that do not dominate any other point in S_i. This problem is equivalent to the anti-dominance problem, of Section 1.4, by inverting the coordinates of all points and of the query. By making a crucial observation on a variant of the low-discrepancy point set proposed by Chazelle and Liu [CL04], we manage to prove the next geometric fact:

Lemma 4.2. *For any integer $\omega \geq 1$ and $\lambda \geq 1$, there is a set P of ω^λ points in $\mathbb{R}^2$ and a set G of $\lambda\omega^{\lambda-1}$ anti-dominance queries that is $(2,\omega)$-favorable, i.e., such that (i) each query in G retrieves ω points of P, and (ii) at most one point in P is returned by two different queries in G simultaneously.*

Proof. We will now construct a $(2,\omega)$-favorable query set $\mathcal{Q}$ and its corresponding point set S, where $\omega > 1$. Without loss of generality, we assume that $n = \omega^\lambda$, where $\lambda > 0$, since this restriction generates a countably infinite number of inputs and thus the lower bound is general. Let us write $0 \leq i < n$ as $i = i_0^{(\omega)} i_1^{(\omega)} \ldots i_{\lambda-1}^{(\omega)}$, where $i_j^{(\omega)}$ is the j-th digit of number i in base ω ($i_{\lambda-1}^{(\omega)}$ is the least significant and $i_0^{(\omega)}$ is the most significant digit.). Then define

$$\rho_\omega(i) = (\omega - i_{\lambda-1}^{(\omega)} - 1)(\omega - i_{\lambda-2}^{(\omega)} - 1)\ldots(\omega - i_0^{(\omega)} - 1)$$

So $\rho_\omega(i)$ is the integer obtained by writing $0 \leq i < n$ using λ digits in base ω, by first reversing the digits and then taking their complement with respect to ω. We define

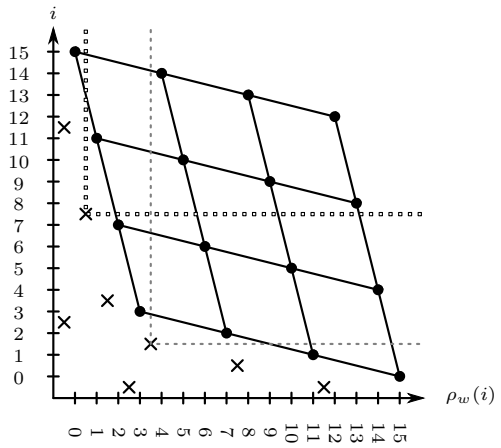

Figure 4.5: An example for $\omega = 4$ and $\lambda = 2$, the point set S is shown with circles and the queries $\mathcal{Q}$ are shown with crosses. Two examples of queries are shown, the first in black/white, the second in gray.

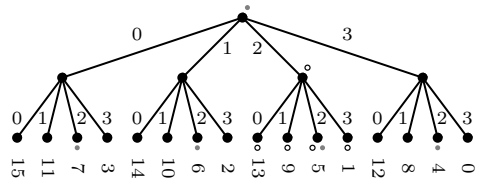

Figure 4.6: The corresponding trie that we used to generate the point set, here the black/white and the gray queries are also shown, along with the internal node which generated the queries.

the points of S to be the set $\{(i, \rho_\omega(i))|0 \leq i < n\}$. Figure 4.5 shows an example with $\omega = 4$ and $\lambda = 2$.

To define the query set $\mathcal{Q}$, we encode the set of points $\{\rho_\omega(i)|0 \leq i < n\}$ in a full trie structure of depth λ. Recall that $n = \omega^\lambda$. Notice that the trie structure is implicit and it is used only for presentation purposes. Input points correspond to the leaves of the trie and their y value is their label at the edges of the trie, where the edges at the root have labels $\omega - i_{\lambda-1}^{(\omega)} - 1$ and the edges at the leafs of the trie have labels $\omega - i_0^{(\omega)} - 1$. Let v be an internal node at depth d (namely v has d ancestors), whose prefix $v_0, v_1, \ldots, v_{d-1}$ corresponds to the path from the root r of the trie to v. We take all points in its subtree and sort them by y. From this sorted list we construct groups of size ω by always picking every $\omega^{\lambda-d-1}$-th element starting from the smallest non-picked element for each group. In this case, we say that the query is *associated* to node v. Each such group corresponds to the output of a query. See Figure 4.5 and 4.6 for an example.

A node at depth d has $\frac{n}{\omega^d}$ points in its subtree and thus it defines at most $\frac{n}{\omega^{d+1}}$ queries. Thus, the total number of queries is:

$$|\mathcal{Q}| = \sum_{d=0}^{\lambda-1} \omega^d \frac{n}{\omega^{d+1}} = \sum_{d=0}^{\lambda-1} \frac{n}{\omega} = \frac{\lambda n}{\omega}$$

In the following we prove that $\mathcal{Q}$ is $(2, \omega)$-favorable. To achieve that we need to prove that $\forall Q_i \in \mathcal{Q} : |S \cap Q_i| \geq \omega$ and $\forall i_1 < i_2 : |S \cap Q_{i_1} \cap Q_{i_2}| = \mathcal{O}(1)$.

First we prove that we can construct the queries so that they have output size ω. Assume that we take one of the groups of ω points associated to node v at depth d. Let the y-coordinates of these points be $\rho_\omega(i_1), \rho_\omega(i_2), \ldots, \rho_\omega(i_\omega)$ in increasing order. These have a common prefix of length d since they all belong to the subtree of v. But we also choose these points so that $\rho_\omega(i_j) - \rho_\omega(i_{j-1}) = \omega^{\lambda-d-1}, 1 < j \leq \omega$. This means that these numbers differ only at the $\lambda - d - 1$-th digit. By inversing the procedure to construct these y-coordinates, the corresponding x-coordinates $i_j, 1 \leq j \leq \omega$ are determined. By complementing we take the increasing sequence $\bar{\rho}_\omega(i_\omega), \ldots, \bar{\rho}_\omega(i_2), \bar{\rho}_\omega(i_1)$, where $\bar{\rho}_\omega(i_j) = \omega^\lambda - \rho_\omega(i_j) - 1$ and $\bar{\rho}_\omega(i_{j-1}) -$

$\bar{\rho}_\omega(i_j) = \omega^{\lambda-d-1}, 1 < j \leq \omega$. By reversing the digits we finally get the increasing sequence of x-coordinates $i_\omega, \ldots, i_2, i_1$, since the numbers differ at only one digit. Thus, the y-coordinate of the group of ω points are decreasing as the x-coordinates increase, and as a result a query q whose horizontal line is just below $\rho_\omega(i_1)$ and the vertical line just to the left of $\rho_\omega(i_\omega)$ will certainly contain this set of points in the query. In addition, there cannot be any other points between this sequence and the horizontal or vertical lines defining query q. This is because all points in the subtree of v have been sorted with respect to y, while the horizontal line is positioned just below $\rho_\omega(i_1)$, so that no other element lies in between. In the same manner, no points to the left of $\rho_\omega(i_\omega)$ exist, when positioning the vertical line of q appropriately. Thus, for each query $q \in \mathcal{Q}$, it holds that $|S \cap q| = \omega$.

We now want to prove that for any two query ranges $p, q \in \mathcal{Q}$, $|S \cap q \cap p| \leq 1$ holds. Assume that p and q are associated to nodes v and u, respectively, and that their subtrees are disjoint. That is, u is not a proper ancestor or descendant of v. In this case, p and q share no common point, since each point is used only once in the trie. For the other case, assume without loss of generality that u is a proper ancestor of v ($u \neq v$). By the discussion in the previous paragraph, each query contains ω numbers that differ at one and only one digit. Since u is a proper ancestor of v, the corresponding digits will be different for the queries defined in u and for the queries defined in v. This implies that there can be at most one common point between these sequences, since the digit that changes for one query range is always set to a particular value for the other query range. □

We use the term (ω, λ)-*input* to refer to the point set P, obtained in Lemma 4.2, after ω and λ have been fixed. We can now use the (ω, λ)-input to prove two lower bounds: the space lower bound in Subsection 4.3.2 and the query lower bound in the Subsection 4.3.3.

4.3.2 Space Lower Bound

We will now combine our (ω, λ)-input with Lemma 4.1 to get our space lower bound for anti-dominance reporting queries in the *PM* model:

Theorem 4.3. *The anti-dominance reporting problem in the Pointer Machine requires $\Omega(n\frac{\log n}{\log\log n})$ space, if the query is supported in $\mathcal{O}(\log^\gamma n + k)$ time, where k is the size of the answer to the query and parameter γ fulfills $0 < \gamma = \mathcal{O}(1)$.*

Proof. From Lemma 4.2 we have a $(2, \omega)$-favorable point set of size $n = \omega^\lambda$ and query set $\mathcal{Q}$ of size $\lambda\omega^{\lambda-1}$. From Lemma 4.1 we have that a (α, ω)-effective graph for an (m, ω)-favorable set of queries $\mathcal{Q}$ must contain $\Omega(\frac{|\mathcal{Q}|\omega}{\alpha})$ nodes. If we put $\alpha = \mathcal{O}(1)$, $m = 2$, $\omega = \log^\gamma n$ and $\lambda = \lfloor \frac{\log n}{1+\gamma \log\log n} \rfloor$ for some constant $0 < \gamma = \mathcal{O}(1)$, then the number of nodes in the graph is at least:

$$\Omega\left(\frac{\omega|\mathcal{Q}|}{\alpha}\right) = \Omega\left(\frac{\omega(\lambda\omega^{\lambda-1})}{\alpha}\right) = \Omega\left(\omega^\lambda\lambda\right) = \Omega\left(n\frac{\log n}{\log\log n}\right).$$

Thus the query time of $\alpha(\omega + k) = \mathcal{O}(\log^\gamma n + k)$, for output size k, can only be achieved at a space cost of $\Omega(n\frac{\log n}{\log\log n})$. □

4.3.3 Query Lower Bound

In this subsection we will give a query lower bound with the indivisibility assumption in the Indexability and *EM* model, when we require the data structure to use at most linear space. We will use the (ω, λ)-input along with the Indexability Theorem

of [HKM+02] to prove the following Lemma, which we will then use to prove our query lower bound in Theorem 4.5:

Lemma 4.3. *Any structure supporting anti-dominance queries in $\mathcal{O}((\frac{n}{B})^{\frac{1}{25c}} + \frac{k}{B})$ I/Os in the worst-case on n points in $\mathbb{R}^2$, in the indexability model, must use $\frac{cn}{B}$ blocks of space, where $c \geq 1$ is a constant and k is the result size.*

Before we continue we will first state the definitions necessary to understand the Indexability Theorem which we will use to prove Lemma 4.3. We will need the following definitions of [HKM+02] from the indexability model.

Definition 4.3 (*Indexing Workload* from [HKM+02])**.** *An* Indexing Workload W *is a tuple $W = (\mathcal{D}, I, \mathcal{Q})$ where $\mathcal{D}$ is a non-empty domain, $I \subseteq \mathcal{D}$ is a finite input set, the instance, and $\mathcal{Q} \subseteq 2^I$ is the set*[2] *of queries.*

Definition 4.4 (*Indexing Scheme* from [HKM+02])**.** *An* Indexing Scheme S *is a tuple $S = (W, \mathcal{B})$ where $W = (\mathcal{D}, I, \mathcal{Q})$ is an indexing workload and $\mathcal{B} \subseteq 2^I$ is a cover of I, i.e. a blocking of the input I.*

Definition 4.5 (*Storage Redundancy* from [HKM+02])**.** *Let $S = (W, \mathcal{B})$ be an indexing scheme over an indexing workload $W = (\mathcal{D}, I, \mathcal{Q})$ with blocking $\mathcal{B}$. Define $r(x) = |\{b \in \mathcal{B} \mid x \in b\}|$ to be the redundancy of element $x \in I$. The* Storage Redundancy r *of S is:*

$$r = \frac{1}{|I|} \sum_{x \in I} r(x).$$

Definition 4.6 (*Access Overhead* from [HKM+02])**.** *Given an indexing scheme $S = (W, \mathcal{B})$ over indexing workload $W = (\mathcal{D}, I, \mathcal{Q})$, let $Q \in \mathcal{Q}$ be a query. Let $C_Q \subseteq \mathcal{B}$ be the minimal set such that $Q \subseteq \bigcup_{b \in C_Q} b$. The access overhead for query Q is:*

$$A(Q) = \frac{|C_Q|}{\left\lceil \frac{|Q|}{B} \right\rceil}.$$

The Access Overhead *for indexing scheme S is $A = \max_{Q \in \mathcal{Q}}\{A(Q)\}$.*

Theorem 4.4 (Indexability Theorem from [HKM+02])**.** *Let $S = (W, \mathcal{B})$ be an indexing scheme with access overhead $A \leq \frac{\sqrt{B}}{4}$ and let $Q_1, \ldots, Q_m$ be queries such that for $1 \leq i \leq m$:*

- $|Q_i| \geq \frac{B}{2}$ *and*
- $|Q_i \cap Q_j| \leq \frac{B}{16A^2}$ *for $1 \leq i < j \leq m$.*

Then the redundancy r is bounded by:

$$r \geq \frac{1}{12|I|} \sum_{i=1}^{m} |Q_i|.$$

Having stated all the definitions and the Indexability Theorem, we are ready to give the proof of Lemma 4.3 and then use it to finally prove our query lower bound in Theorem 4.5.

[2] The notation 2^I denotes the set of all subsets of I.

Proof of Lemma 4.3. Let S be an indexing scheme for the (ω, λ)-input, with access overhead $A \leq \frac{\sqrt{B}}{4}$. Lets put $\omega = B$ and $\lambda = 12c + \frac{11}{10}$ where $c \geq 1$. We first notice that from Lemma 4.2 we have that $n = \omega^\lambda$, $m = \frac{\lambda n}{\omega}$ and any of the queries Q_i of the (ω, λ)-input fulfills that $|Q_i| = \omega$. Also for any two different queries Q_i and Q_j we have that $|Q_i \cap Q_j| \leq 1$. So Theorem 4.4 applies and gives us that:

$$rn = r|I| \geq \frac{1}{12}\sum_{i=1}^{m}|Q_i| = \frac{1}{12}\frac{\lambda}{\omega}n\omega = \frac{(12c + \frac{11}{10})n}{12} \geq cn$$

when $A \leq \frac{\sqrt{B}}{4}$. So each point has redundancy r and the blocking of S must use at least $\frac{cn}{B}$ blocks of space.

Consider any data structure D on the (ω, λ)-input which can answer a query in $\mathcal{O}((\frac{n}{B})^{\frac{1}{25c}} + \frac{k}{B})$ I/Os, this data structure is an indexing scheme. For a query of size $k = \omega$ we have for some constant a that

$$\begin{aligned} A &\leq a\left(\left(\frac{\omega^\lambda}{B}\right)^{\frac{1}{25c}} + \frac{\omega}{B}\right) \\ &= aB^{\frac{120}{250}+\frac{1}{250c}} + a \\ &\leq aB^{\frac{121}{250}} + a. \end{aligned}$$

Now when B is sufficiently large we have that $aB^{\frac{121}{250}} + a \leq \frac{\sqrt{B}}{4}$ and hence $A \leq \frac{\sqrt{B}}{4}$, so from before we have that the data structure must use at least $\frac{cn}{B}$ blocks of space. □

Theorem 4.5. *Any linear space structure supporting anti-dominance queries on n points in the indexability model must incur $\Omega((\frac{n}{B})^\epsilon + \frac{k}{B})$ I/Os, where $\epsilon > 0$ can be an arbitrarily small constant, and k is the result size.*

Proof. Assume for contradiction that there exists a data structure D using at most $\frac{dn}{B}$ blocks of space, for some constant d, which can answer queries in $o((\frac{n}{B})^{\frac{1}{25(d+1)}} + \frac{k}{B})$ I/Os. Then we can choose $c = d + 1$ in Lemma 4.3 and prove that D will use $\frac{(d+1)n}{B} > \frac{dn}{B}$ blocks of space, which is a contradiction to our assumption. □

Bibliography

[AHU74] Alfred Vaino Aho, John Edward Hopcroft, and Jeffrey David Ullman. *The Design and Analysis of Computer Algorithms*. Addison-Wesley Longman Publishing Company Incorporated, 1974. ISBN 0-201-00029-6.

[AM78] Brian Allen and James Ian Munro. Self-organizing binary search trees. *Journal of the ACM (JACM)*, 25(4):526–535, 1978.

[Arg03] Lars Arge. The buffer tree: A technique for designing batched external data structures. *Algorithmica*, 37(1):1–24, 2003.

[AT00] Arne Andersson and Mikkel Thorup. Tight(er) worst-case bounds on dynamic searching and priority queues. In *Proceedings of ACM Symposium on Theory of Computing (STOC)*, pages 335–342. ACM, 2000.

[AV88] Alok Aggarwal and Jeffrey Scott Vitter. The input/output complexity of sorting and related problems. *Communications of the ACM (CACM)*, 31:1116–1127, 1988.

[AVL62] Georgy Adelson-Velsky and Evgenii Mikhailovich Landis. An information organization algorithm. In *Doklady Akademia Nauk SSSR*, volume 146, pages 263–266, 1962.

[Bac78] John Backus. Can programming be liberated from the von neumann style?: A functional style and its algebra of programs. *Communications of the ACM (CACM)*, 21:613–641, 1978.

[Bay74] Rudolf Bayer. Symmetric binary b-trees: Data structure and maintenance algorithms. *Acta Informatica*, 1(4):290–306, 1974.

[BCDI07] Mihai Bădoiu, Richard Cole, Erik D. Demaine, and John Iacono. A unified access bound on comparison-based dynamic dictionaries. *Theoretical Computer Science (TCS)*, 382(2):86–96, 2007.

[BCP08] Ilaria Bartolini, Paolo Ciaccia, and Marco Patella. Efficient sort-based skyline evaluation. *ACM Transactions on Database Systems (TODS)*, 33(4), 2008.

[BCR02] Michael Anthony Bender, Richard Cole, and Rajeev Raman. Exponential structures for efficient cache-oblivious algorithms. In *International Colloquium on Automata, Languages and Programming (ICALP)*, pages 195–207. LNCS, 2002.

[BD04] Mihai Bădoiu and Erik D. Demaine. A simplified and dynamic unified structure. In *Latin American Symposium on Theoretical Informatics (LATIN)*, pages 466–473. LNCS, 2004.

[BDF+10] Gerth Stølting Brodal, Erik D. Demaine, Jeremy T. Fineman, John Iacono, Stefan Langerman, and James Ian Munro. Cache-oblivious dynamic dictionaries with update/query tradeoffs. In *Proceedings of the Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 1448–1456. SIAM, 2010.

[BDIW02] Michael Anthony Bender, Ziyang Duan, John Iacono, and Jing Wu. A locality-preserving cache-oblivious dynamic dictionary. In *Proceedings of the Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 29–38. ACM-SIAM, 2002.

[BDL08] Prosenjit Bose, Karim Douïeb, and Stefan Langerman. Dynamic optimality for skip lists and B-trees. In *Proceedings of the Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 1106–1114. SIAM, 2008.

[Ben80] Jon Louis Bentley. Multidimensional divide-and-conquer. *Communications of the ACM (CACM)*, 23:214–229, 1980.

[BF00] Gerth Stølting Brodal and Rolf Fagerberg. Amortized analysis of (a, b)-trees, 2000. URL `http://www.cs.au.dk/~gerth/emF01/Notes/BF00.ps.gz`.

[BHM09] Prosenjit Bose, John Howat, and Pat Morin. A distribution-sensitive dictionary with low space overhead. In *Algorithms and Data Structures Workshop (WADS)*, volume 5664 of *LNCS*, pages 110–118, 2009.

[BKR12] Gerth Stølting Brodal and Casper Kejlberg-Rasmussen. Cache-oblivious implicit predecessor dictionaries with the working-set property. In *Symposium on Theoretical Aspects of Computer Science (STACS)*, volume 14, pages 112–123. 2012.

[BKRT10] Gerth Stølting Brodal, Casper Kejlberg-Rasmussen, and Jakob Truelsen. A cache-oblivious implicit dictionary with the working set property. In *International Symposium on Algorithms and Computation (ISAAC)*, volume 6507, pages 37–48. 2010.

[BKS01] Stephan Börzsönyi, Donald Kossmann, and Konrad Stocker. The skyline operator. In *Proceedings of International Conference on Data Engineering (ICDE)*, pages 421–430, 2001.

[BLM+02] Gerth Stølting Brodal, George Lagogiannis, Christos Makris, Athanasios Konstantinou Tsakalidis, and Kostas Tsichlas. Optimal finger search trees in the pointer machine. In *Proceedings of ACM Symposium on Theory of Computing (STOC)*, pages 381–418. ACM, 2002.

[BM72] Rudolf Bayer and Edward M. McCreight. Organization and maintenance of large ordered indexes. *Acta Informatica*, 1:173–189, 1972.

[BNT12] Gerth Stølting Brodal, Jesper Sindahl Nielsen, and Jakob Truelsen. Finger search in the implicit model. In *International Symposium on Algorithms and Computation (ISAAC)*, volume 7676, pages 527–536. 2012.

[Bro98] Gerth Stølting Brodal. Finger search trees with constant insertion time. In *Proceedings of the Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 540–549. SIAM, 1998.

[BST85] Samuel Watkins Bent, Daniel Dominic Sleator, and Robert Endre Tarjan. Biased search trees. *SIAM Journal of Computing*, 14:545–568, 1985.

[BT11] Gerth Brodal and Konstantinos Tsakalidis. Dynamic planar range maxima queries. In *International Colloquium on Automata, Languages and Programming (ICALP)*, pages 256–267. LNCS, 2011.

[CGGL05] Jan Chomicki, Parke Godfrey, Jarek Gryz, and Dongming Liang. Skyline with presorting: Theory and optimizations. In *Intelligent Information Systems (IIS)*, pages 595–604, 2005.

[Cha90] Bernard Chazelle. Lower bounds for orthogonal range searching: I. the reporting case. *Journal of the ACM (JACM)*, 37(2):200–212, 1990.

[CL04] Bernard Chazelle and Ding Liu. Lower bounds for intersection searching and fractional cascading in higher dimension. *Journal of Computer and System Sciences (JCSS)*, 68(2):269–284, 2004.

[CLRS01] Thomas H. Cormen, Charles Eric Leiserson, Ronald Linn Rivest, and Clifford Stein. *Introduction to Algorithms*. McGraw-Hill, 2 edition, 2001. ISBN 0-07-013151-1.

[CMSS00] Richard Cole, Bud Mishra, Jeanette Schmidt, and Alan Siegel. On the dynamic finger conjecture for splay trees. part i: Splay sorting log n-block sequences. *SIAM Journal of Computing*, 30:1–43, 2000.

[Col00] Richard Cole. On the dynamic finger conjecture for splay trees. part ii: The proof. *SIAM Journal of Computing*, 30:44–85, 2000.

[CS84] Richard Cole and Alan Siegel. River routing every which way, but loose. In *Proceedings of Annual IEEE Symposium on Foundations of Computer Science (FOCS)*, pages 65–73, 1984.

[DF84] George Diehr and Bruce Faaland. Optimal pagination of b-trees with variable-length items. *Communications of the ACM (CACM)*, 27:241–247, 1984.

[dFGT97] Fabrizio d'Amore, Paolo Giulio Franciosa, Roberto Giaccio, and Maurizio Talamo. Maintaining maxima under boundary updates. In *International Conference on Algorithms and Complexity (CIAC)*, pages 100–109, 1997.

[DGK+12] Ananda Swarup Das, Prosenjit Gupta, Anil Kishore Kalavagattu, Jatin Agarwal, Kannan Srinathan, and Kishore Kothapalli. Range aggregate maximal points in the plane. In *Workshop on Algorithms and Computation (WALCOM)*, pages 52–63, 2012.

[DHI+09] Erik D. Demaine, Dion Harmon, John Iacono, Daniel Kane, and Mihai Pătraşcu. The geometry of binary search trees. In *Proceedings of the Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 496–505. SIAM, 2009.

[DHIP07] Erik D. Demaine, Dion Harmon, John Iacono, and Mihai Pătraşcu. Dynamic optimality – almost. *SIAM Journal of Computing*, 37:240–251, 2007.

[DR94] Paul F. Dietz and Rajeev Raman. A constant update time finger search tree. *Information Processing Letters (IPL)*, 52:147–154, 1994.

[DZ04] H. K. Dai and X. W. Zhang. Improved linear expected-time algorithms for computing maxima. In *Latin American Symposium on Theoretical Informatics (LATIN)*, pages 181–192. LNCS, 2004.

[FG03] Gianni Franceschini and Roberto Grossi. Optimal worst-case operations for implicit cache-oblivious search trees. In *Algorithms and Data Structures Workshop (WADS)*, volume 2748 of *LNCS*, pages 114–126, 2003.

[FG06] Gianni Franceschini and Roberto Grossi. Optimal implicit dictionaries over unbounded universes. *Theoretical Computer Science (TCS)*, 39: 321–345, 2006.

[FGMP02] Gianni Franceschini, Roberto Grossi, James Ian Munro, and Linda Pagli. Implicit B-trees: New results for the dictionary problem. In *Proceedings of Annual IEEE Symposium on Foundations of Computer Science (FOCS)*, pages 145–154, 2002.

[Fle93] Rudolf Fleischer. A simple balanced search tree with $\mathcal{O}(1)$ worst-case update time. In *International Symposium on Algorithms and Computation (ISAAC)*, volume 762, pages 138–146. 1993.

[FLPR99] Matteo Frigo, Charles Eric Leiserson, Harald Prokop, and Sridhar Ramachandran. Cache-oblivious algorithms. In *Proceedings of Annual IEEE Symposium on Foundations of Computer Science (FOCS)*, pages 285–297. IEEE, 1999.

[FR90] Greg Norman Frederickson and Susan Rodger. A new approach to the dynamic maintenance of maximal points in a plane. *Discrete & Computational Geometry*, 5(1):365–374, 1990.

[Fre84] Greg Norman Frederickson. Self-organizing heuristics for implicit data structures. *SIAM Journal of Computing*, 13:277–291, 1984.

[FT83] Joan Feigenbaum and Robert Endre Tarjan. Two new kinds of biased search trees. *The Bell System technical journal*, 62(10):3139–3158, 1983.

[GBT84] Harold N. Gabow, Jon Louis Bentley, and Robert Endre Tarjan. Scaling and related techniques for geometry problems. In *Proceedings of ACM Symposium on Theory of Computing (STOC)*, pages 135–143. ACM, 1984.

[GMPR77] Leonidas John Guibas, Edward M. McCreight, Michael Frederick Plass, and Janet R. Roberts. A new representation for linear lists. In *Proceedings of ACM Symposium on Theory of Computing (STOC)*, pages 49–60. ACM, 1977.

[GT86] Hania Gajewska and Robert Endre Tarjan. Deques with heap order. *Information Processing Letters (IPL)*, 22:197–200, 1986.

[HKM+02] Joseph M. Hellerstein, Elias Koutsoupias, Daniel P. Miranker, Christos H. Papadimitriou, and Vasilis Samoladas. On a model of indexability and its bounds for range queries. *Journal of the ACM (JACM)*, 49(1):35–55, 2002.

[HM82] Scott Huddleston and Kurt Mehlhorn. A new data structure for representing sorted lists. *Acta Informatica*, 17(2):157–184, 1982.

[Hug89] John Hughes. Why functional programming matters. *The Computer Journal*, 32:98–107, 1989.

[Iac01] John Iacono. Alternatives to splay trees with $\mathcal{O}(\log n)$ worst-case access times. In *Proceedings of the Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 516–522. SIAM, 2001.

[Jan91] Ravi Janardan. On the dynamic maintenance of maximal points in the plane. *Information Processing Letters (IPL)*, 40:59–64, 1991.

[Kap00] Sanjiv Kapoor. Dynamic maintenance of maxima of 2-d point sets. *SIAM Journal of Computing*, 29:1858–1877, 2000.

[KDKS11] Anil Kishore Kalavagattu, Ananda Swarup Das, Kishore Kothapalli, and Kannan Srinathan. On finding skyline points for range queries in plane. In *Proceedings of the Canadian Conference on Computational Geometry (CCCG)*, 2011.

[KLP75] H. T. Kung, Fabrizio L Luccio, and Franco P Preparata. On finding the maxima of a set of vectors. *Journal of the ACM (JACM)*, 22(4): 469–476, 1975.

[Knu71] Donald Ervin Knuth. Optimum binary search trees. *Acta Informatica*, 1:14–25, 1971.

[KRR02] Donald Kossmann, Frank Ramsak, and Steffen Rost. Shooting stars in the sky: An online algorithm for skyline queries. In *Proceedings of Very Large Data Bases (VLDB)*, pages 275–286, 2002.

[KRTT+13] Casper Kejlberg-Rasmussen, Yufei Tao, Konstantinos Tsakalidis, Kostas Tsichlas, and Jeonghun Yoon. I/O-efficient planar range skyline and attrition priority queues. In *Proceedings of ACM Symposium on Principles of Database Systems (PODS)*, pages 103–114. 2013.

[KS85] David G. Kirkpatrick and Raimund Seidel. Output-size sensitive algorithms for finding maximal vectors. In *Proceedings of Symposium on Computational Geometry (SoCG)*, pages 89–96. ACM, 1985.

[KT96] Haim Kaplan and Robert Endre Tarjan. Purely functional representations of catenable sorted lists. In *Proceedings of ACM Symposium on Theory of Computing (STOC)*, pages 202–211. ACM, 1996.

[KT99] Haim Kaplan and Robert Endre Tarjan. Purely functional, real-time deques with catenation. *Journal of the ACM (JACM)*, 46(5):577–603, 1999.

[LH85] Lawrence Louis Larmore and Daniel S Hirschberg. Efficient optimal pagination of scrolls. *Communications of the ACM (CACM)*, 28:854–856, 1985.

[McC77] Edward M. McCreight. Pagination of b-trees with variable-length records. *Communications of the ACM (CACM)*, 20:670–674, 1977.

[MPJ07] Michael David Morse, Jignesh Manubhai Patel, and H. V. Jagadish. Efficient skyline computation over low-cardinality domains. In *Proceedings of Very Large Data Bases (VLDB)*, pages 267–278, 2007.

[MS79] James Ian Munro and Hendra Suwanda. Implicit data structures (preliminary draft). In *Proceedings of ACM Symposium on Theory of Computing (STOC)*, pages 108–117. ACM, 1979.

[Mun86] James Ian Munro. An implicit data structure supporting insertion, deletion, and search in $\mathcal{O}(\log^2 n)$ time. *Journal of Computer and System Sciences (JCSS)*, 33:66–74, 1986.

[OvL81] Mark Hendrik Overmars and Jan van Leeuwen. Maintenance of configurations in the plane. *Journal of Computer and System Sciences (JCSS)*, 23:166–204, 1981.

[PT06] Mihai Pătraşcu and Mikkel Thorup. Time-space trade-offs for predecessor search. In *Proceedings of ACM Symposium on Theory of Computing (STOC)*, pages 232–240. ACM, 2006.

[PTFS05] Dimitris Papadias, Yufei Tao, Greg Fu, and Bernhard Seeger. Progressive skyline computation in database systems. *ACM Transactions on Database Systems (TODS)*, 30(1):41–82, 2005.

[SLNX09] Atish Das Sarma, Ashwin Lall, Danupon Nanongkai, and Jun Xu. Randomized multi-pass streaming skyline algorithms. *Proceedings of the VLDB Endowment (PVLDB)*, 2(1):85–96, 2009.

[SSK09] Mehdi Sharifzadeh, Cyrus Shahabi, and Leyla Kazemi. Processing spatial skyline queries in both vector spaces and spatial network databases. *ACM Transactions on Database Systems (TODS)*, 34(3), 2009.

[ST85] Daniel Dominic Sleator and Robert Endre Tarjan. Self-adjusting binary search trees. *Journal of the ACM (JACM)*, 32:652–686, 1985.

[ST11] Cheng Sheng and Yufei Tao. On finding skylines in external memory. In *Proceedings of ACM Symposium on Principles of Database Systems (PODS)*, pages 107–116, 2011.

[Sun89] Rajamani Sundar. Worst-case data structures for the priority queue with attrition. *Information Processing Letters (IPL)*, 31:69–75, 1989.

[Tar79] Robert Endre Tarjan. A class of algorithms which require nonlinear time to maintain disjoint sets. *Journal of Computer and System Sciences (JCSS)*, 18:110–127, 1979.

[Tar85] Robert Endre Tarjan. Sequential access in splay trees takes linear time. *Combinatorica*, 5(4):367–378, 1985.

[Tsa85] Athanasios Konstantinou Tsakalidis. Avl-trees for localized search. *Information and Control*, 67(1–3):173–194, 1985.

[vEB75] Peter van Emde Boas. Preserving order in a forest in less than logarithmic time. In *Proceedings of Annual IEEE Symposium on Foundations of Computer Science (FOCS)*, pages 75–84, 1975.

[vN45] John von Neumann. First draft of a report on the edvac. Technical Report W-670-ORD-4926, 1945.

[Wil83] Dan E. Willard. Log-logarithmic worst-case range queries are possible in space $\Theta(N)$. *Information Processing Letters (IPL)*, 17:81–84, 1983.

[Wil89] Robert Wilber. Lower bounds for accessing binary search trees with rotations. *SIAM Journal of Computing*, 18:56–67, 1989.