Data Structure Design

Theory and Practice

Gerth Stølting Brodal
Aarhus University
Denmark
Gerth Stølting Brodal

Research
Data structures 1993 –

Teaching
Algorithms and Data Structures 2002 –
Introduction to Programming (Python) 2018 –
Bachelor project advising
Efficient Algorithms
= Algorithms + Data structures

influence on algorithm design, computational models, cost models
TIOBE Programming Community Index

www.tiobe.com/tiobe-index/
Extendable Arrays – Reallocation Strategies

C++ vector
+ 100%

Java ArrayList
+ 50%

Python list
+ 12.5%

append(13)

static int
list_resize(PyListObject *self, Py_ssize_t newsize)
{
    PyObject **items;
    size_t new_allocated,
    num_allocated_bytes;
    Py_ssize_t allocated = self->allocated;
    if (allocated >= newsize && newsize >= (allocated >> 1)) {
        assert(self->ob_item != NULL || newsize == 0);
        Py_SIZE(self) = newsize;
        return 0;
    }
    new_allocated = (size_t)newsize + (newsize >> 3) + (newsize < 9 ? 3 : 6);
    ...
Branches

for (int i=0; i < size; i++)
    if (A[i] <= threshold)
        small ++;

<table>
<thead>
<tr>
<th>threshold</th>
<th>time (seconds)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.0</td>
<td>0.045</td>
</tr>
<tr>
<td>0.5</td>
<td>0.458</td>
</tr>
<tr>
<td>1.0</td>
<td>0.046</td>
</tr>
</tbody>
</table>

A random floats in range [0, 1]

Threshold Counting

11th Gen Intel Core i7-1165G7 @ 2.80GHz, Windows 10 + cygwin, gcc -O2, performed $2^{27}$ comparisons (repeatedly ran over array)
for (int i=0; i < size; i++)
    if (A[i] <= threshold)
        small ++;

.L7: comiss (%rax), %xmm6
    jb .L5
    addq $1, %r14

.L5: addq $4, %rax
    cmpq %rbx, %rax
    jne .L7

gcc -O2 -fno-if-conversion -fno-if-conversion2

No branch mispredictions

.L7: comiss (%rax), %xmm6
    sbbq $-1, %r14
    addq $4, %rax
    cmpq %rbx, %rax
    jne .L7

gcc -O2
Binary Search

int low = 0, high = size;
while (low < high) {
    int mid = low+(int)((high-low)*bias);
    if (A[mid] <= x)
        low = mid + 1;
    else
        high = mid;
}

Binary search – do not choose the middle
Summary Branch Mispredictions

- Mispredictions can slow done programs by a factor 10
- **Binary search** faster with biased pivot
- **Binary search trees** faster with biased pivots [B. and Moruz, ESA 2004]
- **QuickSort** faster with biased pivot [Kaligosi and Sanders, ESA 2006] – also analyzed different prediction models
- **InsertionSort** $O(n^2)$ comparisons but $O(n)$ mispredictions
- **MergeSort** with InsertionSort for small problems (used in standard libraries)
- **Sorting** [B. and Moruz, WADS 2005] $O(d \cdot n \cdot \log n)$ comparisons $\Rightarrow \Omega(n \cdot \log d n)$ mispredictions
position = 0;
for (int i=0; i < iterations; i++)
    position = A[position];

<table>
<thead>
<tr>
<th>step</th>
<th>time (seconds)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0.297</td>
</tr>
<tr>
<td>1024</td>
<td>19.5 x 66</td>
</tr>
</tbody>
</table>

(size = 16777216)
Pointer Chasing

8 way cache associativity

page size 4 KB (step 1024)

cache line size 64 bytes (step 16)

Belianska cave?

TLB misses?

L3 cache 12 MB (size 4194304)

L2 cache 512 KB (size 131072)

L1 cache 48 KB (size 16384)
randomly permute pointer cells
Memory Hierarchy

- **CPU**
- **Bus**
  - **L1**
  - **L2**
  - **L3**
- **Memory (RAM)**
  - 64 KB: 5 ns, 64 bytes
  - 512 KB: 20 ns, 64 bytes
  - 20 MB: 30 ns, 64 bytes
  - 32 GB: 60 ns, 64 bytes
  - 1 TB: 5.000.000 ns, 64 bytes
- **Harddisk**
  - 4.096 bytes

**Cache size**, **Access time**, **Block size**
Cost of Address Translation

Time / RAM complexity

[Jurkiewicz, Mehlhorn, *The cost of address translation*, ALENEX 2013]
External Memory and Cache-Oblivious Models

- **External memory model** parameters $B$ and $M$
- Scanning $O(N/B)$ IOs
- Sorting $O(N/B \cdot \log_{M/B} (N/B))$ IOs
- Searching $O(\log_B N)$ IOs
- **Cache oblivious model** is like external memory model ... but algorithms do not know $B$ and $M$
  (assume optimal cache replacement strategy)
- Optimal on all memory levels
  (under some assumptions)

Recursive Tree Layout
(van Emde Boas layout)

Binary tree

Search $O(\log_B N)$ IOs

Range Searches $O(\log_B N + k/B)$ IOs

[Prokop, MIT MSc thesis Cache-Oblivious Algorithms, 1999]
Random Searches in Perfectly Balanced Search Trees

van Emde Boas Layout

random layouts

[B., Fagerberg, Jacob, *Cache-Oblivious Search Trees via Binary Trees of Small Height*, SODA 2002]
No Balanced Search Trees in Python?

- Python standard library does not contain balanced search trees
- `insert_left` inserts into a sorted list [binary search O(log n) + memcopy O(n)]

```
Python shell
> L = [0, 10, 20, 30, 40, 50, 60, 70, 80, 90]
> bisect.insort_left(L, 42)
> print(L)
[0, 10, 20, 30, 40, 42, 50, 60, 70, 80, 90]
```

- `SortedList` in `sortedcollections` essentially combines list-of-lists with bisect
  
  [[1, 5, 15, 28], [35, 38, 38, 41, 44], [46, 61, 63], [70, 87, 89]]

  updates O(\(\sqrt{n}\)) and queries O(log n)
Random Access Machine (RAM) model great for designing and analyze algorithms

... but final program performance depends on hardware

Have an idea of what the bottleneck is in your computation and choose an appropriate abstract model
An Unexpected Journey

- Bachelor project = shortest paths on Open Street Map graphs
- Students have trouble implementing Dijkstra's algorithm in Java™
Dijkstra’s Algorithm (1956)

- Non-negative edge weights
- Visits nodes in increasing distance from source

```plaintext
Fibonacci heaps
(Fredman, Tarjan 1984) \( \Rightarrow \) \( O(m + n \cdot \log n) \)
```

```
proc Dijkstra_1 (V, E, \( \delta \), s)
    dist[v] = +\( \infty \) for all \( v \in V \setminus \{s\} \)
    dist[s] = 0
    Q = (\( \emptyset, A \) )
    Insert(Q, \( \langle \text{dist}[s], s \rangle \))
    while Q \( \neq \emptyset \) do
        \( \langle d, u \rangle = \text{ExtractMin}(Q) \)
        for \( (u, v) \in E \cap (\{u\} \times V) \) do
            if dist[u] + \( \delta(u, v) \) < dist[v] then
                dist[v] = dist[u] + \( \delta(u, v) \)
            if v \in Q then
                DecreaseKey(Q, v, dist[v])
            else
                Insert(Q, \( \langle v, \text{dist}[v] \rangle \))
        return dist
```

```
proc Dijkstra_2 (V, E, \( \delta \), s)
    dist[v] = +\( \infty \) for all \( v \in V \setminus \{s\} \)
    dist[s] = 0
    Q = (\( \emptyset, A \) )
    Insert(Q, \( \langle \text{dist}[s], s \rangle \))
    while Q \( \neq \emptyset \) do
        \( \langle d, u \rangle = \text{ExtractMin}(Q) \)
        for \( (u, v) \in E \cap (\{u\} \times V) \) do
            if dist[u] + \( \delta(u, v) \) < dist[v] then
                dist[v] = dist[u] + \( \delta(u, v) \)
            if v \in Q then
                Remove(Q, v)
            else
                Insert(Q, \( \langle v, \text{dist}[v] \rangle \))
        return dist
```

\( \Rightarrow \) \( O(\log n) \) Remove

\( \Rightarrow \) \( O(m \cdot \log n) \)
The Challenge - Java’s Builtin Binary Heap

- No `decreasekey`
- `remove` $O(n)$ time $\Rightarrow$ Dijkstra $O(m \cdot n)$

Implementation note: this implementation provides $O(\log(n))$ time for the enqueuing and dequeuing methods (`offer, poll, remove()` and `add`); linear time for the `remove(Object)` and `contains(Object)` methods; and constant time for the retrieval methods (`peek, element, and size`).

This class is a member of the Java Collections Framework.

Since: 1.5
Repeated Insertions

- **Relax** inserts new copies of item
- **Skip outdated** items

```plaintext
proc Dijkstra_3(V, E, δ, s)
    dist[v] = +∞ for all v ∈ V \ {s}
    dist[s] = 0
    Insert(Q, ⟨dist[s], s⟩)
    while Q ≠ ∅ do
        ⟨d, u⟩ = ExtractMin(Q)
        if d = dist[u] then
            for (u, v) ∈ E ∩ ({u} × V) do
                if dist[u] + δ(u, v) < dist[v] then
                    dist[v] = dist[u] + δ(u, v)
                    Insert(Q, ⟨dist[v], v⟩)
        return dist
```
Using a Visited Set

**proc Dijkstra**\(_4\)\((V, E, δ, s)\)

\[
dist[v] = +\infty \text{ for all } v \in V \setminus \{s\}
\]

\[
dist[s] = 0
\]

\[
\text{visited} = \emptyset
\]

Insert\((Q, \langle dist[s], s \rangle)\)

while \(Q \neq \emptyset\) do

\[
\langle d, u \rangle = \text{ExtractMin}(Q)
\]

if \(u \notin \text{visited}\) then

\[
\text{visited} = \text{visited} \cup \{u\}
\]

for \((u, v) \in E \cap (\{u\} \times V)\) do

if \(\text{dist}[u] + \delta(u, v) < \text{dist}[v]\) then

\[
\text{dist}[v] = \text{dist}[u] + \delta(u, v)
\]

Insert\((Q, \langle \text{dist}[v], v \rangle)\)

return \(\text{dist}\)
A Shaky Idea...

```plaintext
proc Dijkstra(V, E, δ, s)
    dist[v] = +∞ for all v ∈ V \ {s}
    dist[s] = 0
    visited = ∅
    Insert(Q, (dist[s], s))
    while Q ≠ ∅ do
        u = ExtractMin(Q)
        if u ∉ visited then
            visited = visited ∪ {u}
            for (u, v) ∈ E ∩ ({u} × V) do
                if dist[u] + δ(u, v) < dist[v] then
                    dist[v] = dist[u] + δ(u, v)
                    Insert(Q, (dist[v], v))
            return dist
```

- ▪ \( Q \) only store nodes (save space)
- ▪ Comparator
- ▪ Key = current distance \( dist \)

Heap invariants break
The Challenge - Java’s Builtin Binary Heap

- Comparator function

```
PriorityQueue(int initialCapacity)  Creates a PriorityQueue with the
                                    specified initial capacity that orders its
                                    elements according to their natural
                                    ordering.

PriorityQueue(int initialCapacity, 
               Comparator<? super E> comparator) Creates a PriorityQueue with the
                                                specified initial capacity that orders its
                                                elements according to the specified
                                                comparator.
```

Experimental Study

- Implemented Dijkstra in Python
- Stress test on random cliques
- Binary heaps failed (default priority queue in Java and Python)
Binary Heaps Fail using \textit{dist} in a Comparator

- Outdated
- Wrong placement
- Not smallest key
- Ignored since visited
Experimental Study

- Implemented Dijkstra\textsubscript{4} in Python
- Stress test on random cliques
- Binary heaps: failed (default priority queue in Java and Python)
  - Skew heaps: worked
  - Leftist heaps: worked
  - Pairing heaps: worked
  - Binomial queues: worked
  - Post-order heaps: worked
  - Binary heaps with top-down insertions: worked

\begin{verbatim}
visited = set()
Q = Queue()
Q.insert(Item(0, source))
while not Q.empty():
    u = Q.extract_min().value
    if u not in visited:
        visited.add(u)
        for v in G.out[u]:
            dist_v = dist[u] + G.weights[(u, v)]
            if dist_v < dist[v]:
                dist[v] = dist_v
                parent[v] = u
                Q.insert(Item(dist[v], v))
\end{verbatim}
Binary Heap Insertions: Bottom-up vs Top-down

- **Theorem** Skew, left, pairing, binomial, post-order, binary top-down heaps support a generalized notion of heap order with decreasing keys

- **Theorem** Dijkstra$_4$ works correctly
Experimental Evaluation of Various Heaps

- Cliques with uniform random weights
- With decreasing keys less comparisons (outdated items removed earlier)

\[(\text{key, value}) \text{ pairs decreasing keys}\]

Graphs showing comparisons for different heap types with varying numbers of nodes.
Reduction in Comparisons

comparisons decreasing keys / comparisons ⟨key, value⟩ pairs
Summary of the Unexpected Journey

- Introduced notion of priority queues with decreasing keys
  ... as an approach to deal with outdated items in Dijkstra’s algorithm
- Experiments identified priority queues supporting decreasing keys
  ... just had to prove it
- Built-in priority queues in Java and Python are binary heaps
  ... do not support decreasing keys
- Binary heaps with top-down insertions do support decreasing keys
  ... and also
  skew heaps, leftist heaps, pairing heaps, binomial queues, post-order heaps
The reviewer is always right
“If there was a implementation where the authors verified that everything did what it was supposed to, I would be more confident that things were correct (I am not talking about a practical implementation, I am talking about one to make sure all invariants hold).”

Anonymous reviewer
# Strict Fibonacci Heaps

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Insert</td>
<td>O(log n)</td>
<td>O(1)</td>
<td>O(1)</td>
</tr>
<tr>
<td>ExtractMin</td>
<td>O(log n)</td>
<td>O(log n)</td>
<td>O(log n)</td>
</tr>
<tr>
<td>DecreaseKey</td>
<td>O(log n)</td>
<td>O(1)</td>
<td>O(1)</td>
</tr>
<tr>
<td>Meld</td>
<td>-</td>
<td>O(1)</td>
<td>O(1)</td>
</tr>
</tbody>
</table>
Strict Fibonacci Heaps

+ many structural invariants
Python Implementation

- 1589 lines
- 215 assert statements
- All claimed invariants turned into assert statements
- Validation methods to traverse full structure to verify all claimed invariants
- Stress test using random inputs
- Supported the theory
Code coverage

- Used the Python module `coverage`
- Some code rarely executed
- Repeat random test 1,000,000 times
- Most code executed at least once

- Realized there was code for cases which provably never can occur
- Implementation → new invariants discovered
Branch coverage

- Thought code coverage would find all "logical errors"
- Found several if statements with no else part, where condition provably would always be true
- Implementation → new invariants discovered (and assertions added)
”The first main suggestion is to have at least one figure with a logical diagram of a non-trivial example structure, [...]. This would go a long way in giving some idea of what the structure is.”

Anonymous reviewer
- Hard to manually create a figure that was guaranteed to be a real example
- Could use implementation to automatically generate (LaTeX tikz) figures
- Generated random inputs
- Formalized requirements to figure as a loop condition
- Repeat until happy
Data Structure Design

confirm & evaluate theory

Theory

Practice

theoretical insights, computational model considerations