Hardware-Aware Algorithms and Data Structures

Gabriel Moruz
BRICS
University of Aarhus

MADALGO
CENTER FOR MASSIVE DATA ALGORITHMICS

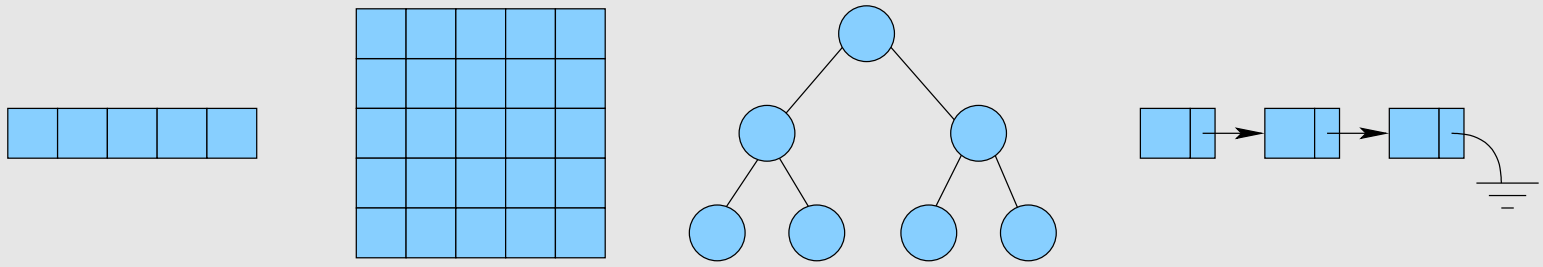
DANMARKS GRUNDFORSKNINGSFOND
DANISH NATIONAL RESEARCH FOUNDATION

Hardware /nm./: "the part of the computer that you can kick."

– Geeky folklore.

Algorithms and Data Structures

- Algorithm:
  - A finite sequence of steps to solve a problem
  - Is given an input
  - Is required to produce an output
  - Should be efficient
- Data structure:
  - The "way" in which data is stored
  - Supports operations

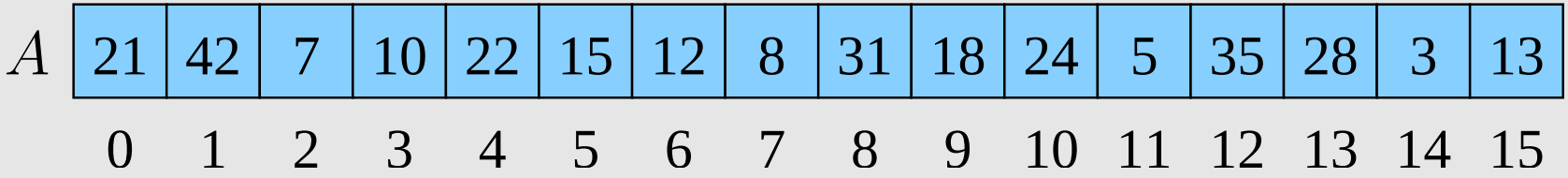


Example – Searching

- The problem
  - Input: A sequence of numbers A, an element e
  - Output: YES, if e is in A, NO otherwise
- Dictionary – underlying data structure
  - Static: Supports only searches
  - Dynamic: Supports searches and updates
- Why bother
  - Numerous applications: Database systems, search engines, implementing sets, sorting, interval trees, orthogonal range searching, line segment intersection, phone book, the search for the Holy Grail, finding Nemo, the vial of life, cherchez la femme, the lost city of Atlantis, the bad Mafia guys, the dark Mordor, pirates' treasure chest etc.

Linear Search

- Consider sequence A to be an array of size n
- Efficiency - the number of comparisons



- The algorithm
  - Compare elements in A against e left-to-right
  - Stop upon encountering an element equal to e
- Analysis
  - What if $e = 13$ or e not in A?
  - Worst case scenario: need to access all elements in A!!!
  - Why avoiding this approach: imagine $n = 100,000,000$

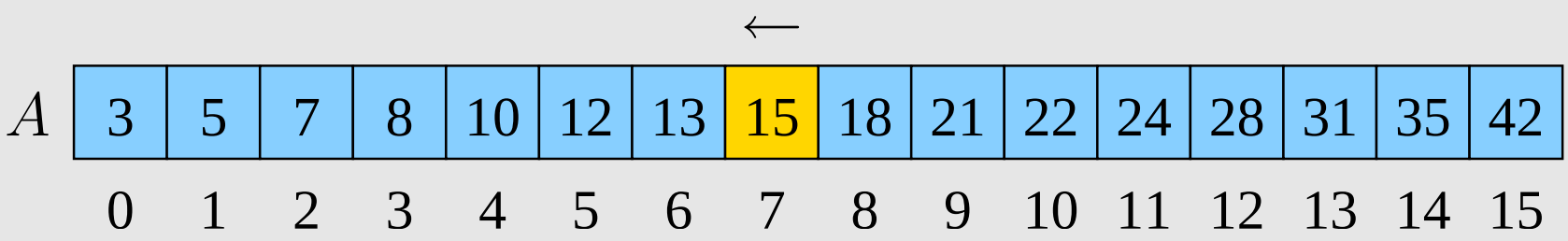
What if A is sorted?

A	3	5	7	8	10	12	13	15	18	21	22	24	28	31	35	42
	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

The algorithm – binary search:

- Compare e against the middle element in A
- If e is smaller then restrict to the left half of A
- If e is larger then restrict to the right half of A
- Stop when:
  - an element in A matching e is found, or
  - the sequence in which we search has one element

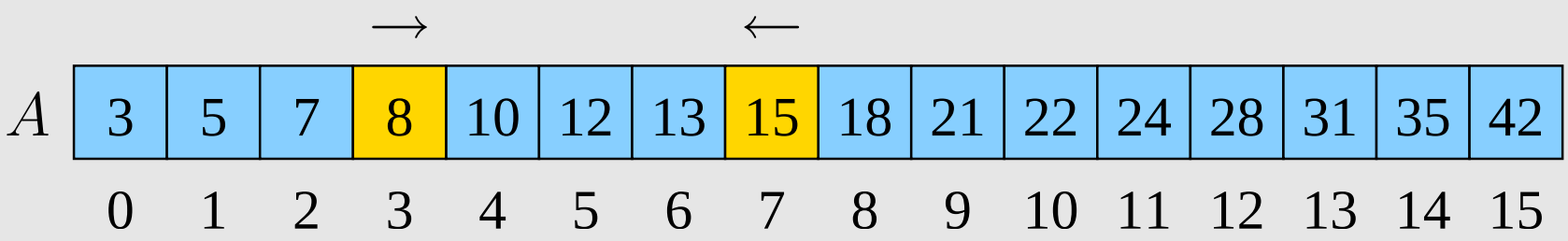
What if A is sorted?



The algorithm – binary search:

- Compare e against the middle element in A
- If e is smaller then restrict to the left half of A
- If e is larger then restrict to the right half of A
- Stop when:
  - an element in A matching e is found, or
  - the sequence in which we search has one element
- The searched element $e = 13$

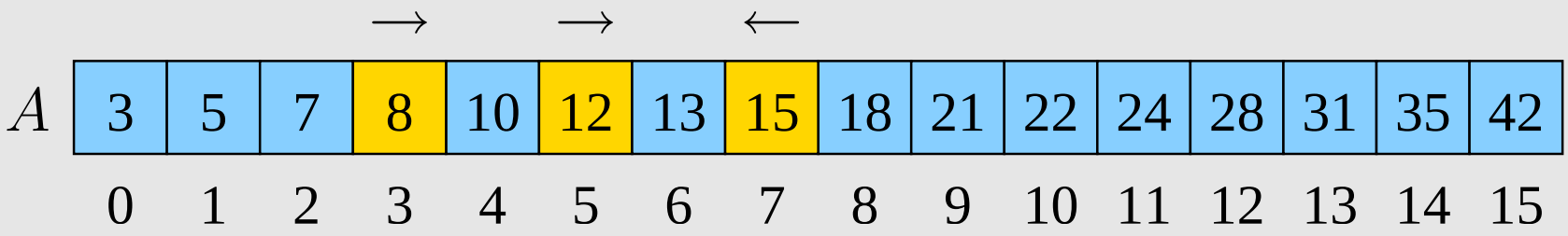
What if A is sorted?



The algorithm – binary search:

- Compare e against the middle element in A
- If e is smaller then restrict to the left half of A
- If e is larger then restrict to the right half of A
- Stop when:
  - an element in A matching e is found, or
  - the sequence in which we search has one element
- The searched element $e = 13$

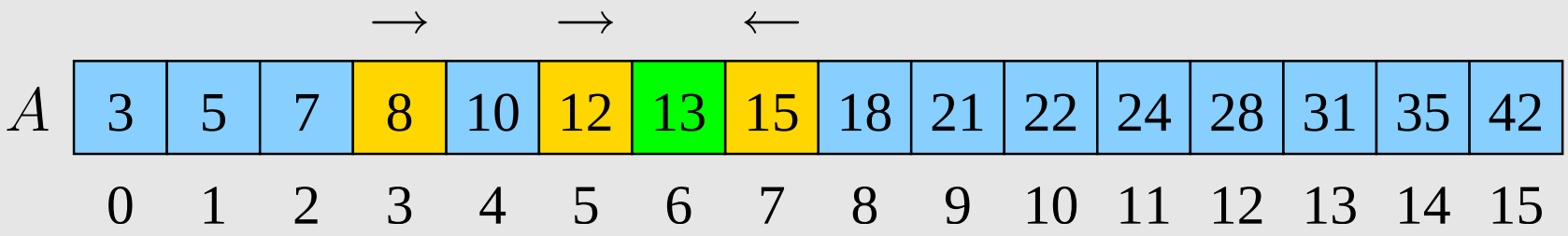
What if A is sorted?



The algorithm – binary search:

- Compare e against the middle element in A
- If e is smaller then restrict to the left half of A
- If e is larger then restrict to the right half of A
- Stop when:
  - an element in A matching e is found, or
  - the sequence in which we search has one element
- The searched element $e = 13$

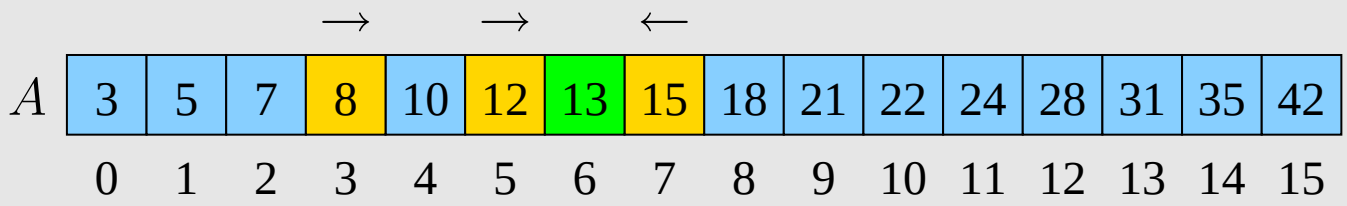
What if A is sorted?



The algorithm – binary search:

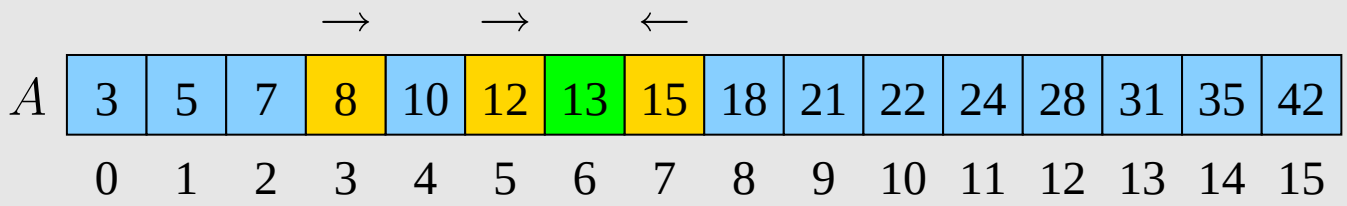
- Compare e against the middle element in A
- If e is smaller then restrict to the left half of A
- If e is larger then restrict to the right half of A
- Stop when:
  - an element in A matching e is found, or
  - the sequence in which we search has one element
- The searched element $e = 13$

Analyzing Binary Search



- Analysis
  - One comparison: search in a sequence of size $n/2$
  - Two comparisons: search in a sequence of size $n/4$
  - k comparisons: search in a sequence of size $n/(2^k)$
  - Worst case scenario: stop in a sequence of size 1
  - Sequence size $n/(2^k) = 1$, meaning $k \approx \log_2 n$
  - Conclusion: we need about $\log_2 n$ comparisons

Analyzing Binary Search



- Analysis
  - One comparison: search in a sequence of size $n/2$
  - Two comparisons: search in a sequence of size $n/4$
  - k comparisons: search in a sequence of size $n/(2^k)$
  - Worst case scenario: stop in a sequence of size 1
  - Sequence size $n/(2^k) = 1$, meaning $k \approx \log_2 n$
  - Conclusion: we need about $\log_2 n$ comparisons

- Imagine $n = 100,000,000$: $\log_2 100,000,000 \approx 26.5$

Outline

- Hardware factors affecting the running time
  - Instructions performed by microprocessor
  - Branch mispredictions
  - Memory transfers
  - Streaming
- Hardware factors affecting the reliability
  - Memory corruptions
- Optimal resilient dictionaries

Theory vs Practice

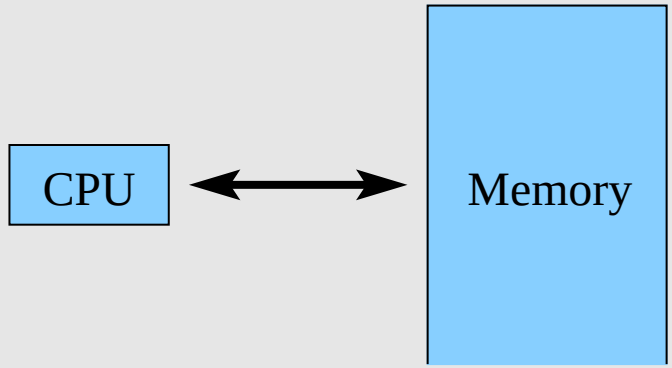
In theory, theory and practice are the same

Theory vs Practice

In theory, theory and practice are the same

In practice, theory and practice may be quite different . . .

Traditional RAM model



- Consists of a processor and an infinite memory
- Instructions:
  - Load/stores of memory cells, assignments, comparisons, simple math operations
  - NO loops!
- Complexity: given by # instructions
- Not always adequate!!!

Branch Mispredictions – Motivation

- Input:
  - a – array of size 2×10^7, $a_i \in [1, \ldots, 100]$
  - $param$ – a threshold, $param \in [0, \ldots, 101]$
- Output:
  - g – # elements in a larger than $param$
  - s – # elements in a smaller or equal to $param$
- Algorithm:
  - Compare each element in a against $param$
  - Use a left-to-right scan

72	21	3	45	98	53	87	17	24	33	52	8	81	79	63	48

$param = 30,\ g = 0,\ s = 0$

Branch Mispredictions – Motivation

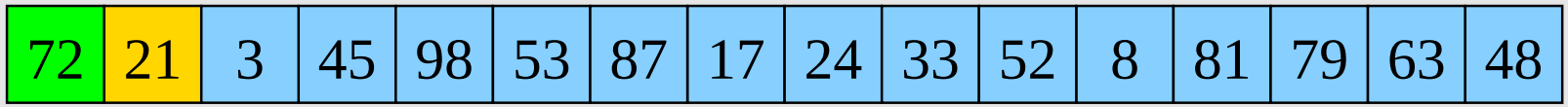
- Input:
  - a – array of size 2×10^7, $a_i \in [1, \ldots, 100]$
  - $param$ – a threshold, $param \in [0, \ldots, 101]$
- Output:
  - g – # elements in a larger than $param$
  - s – # elements in a smaller or equal to $param$
- Algorithm:
  - Compare each element in a against $param$
  - Use a left-to-right scan

72	21	3	45	98	53	87	17	24	33	52	8	81	79	63	48

$param = 30$, $g = 1$, $s = 0$

Branch Mispredictions – Motivation

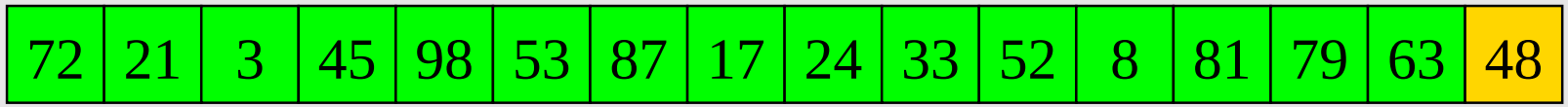
- Input:
  - a – array of size 2×10^7, $a_i \in [1, \ldots, 100]$
  - $param$ – a threshold, $param \in [0, \ldots, 101]$
- Output:
  - g – # elements in a larger than $param$
  - s – # elements in a smaller or equal to $param$
- Algorithm:
  - Compare each element in a against $param$
  - Use a left-to-right scan



$param = 30$, $g = 1$, $s = 1$

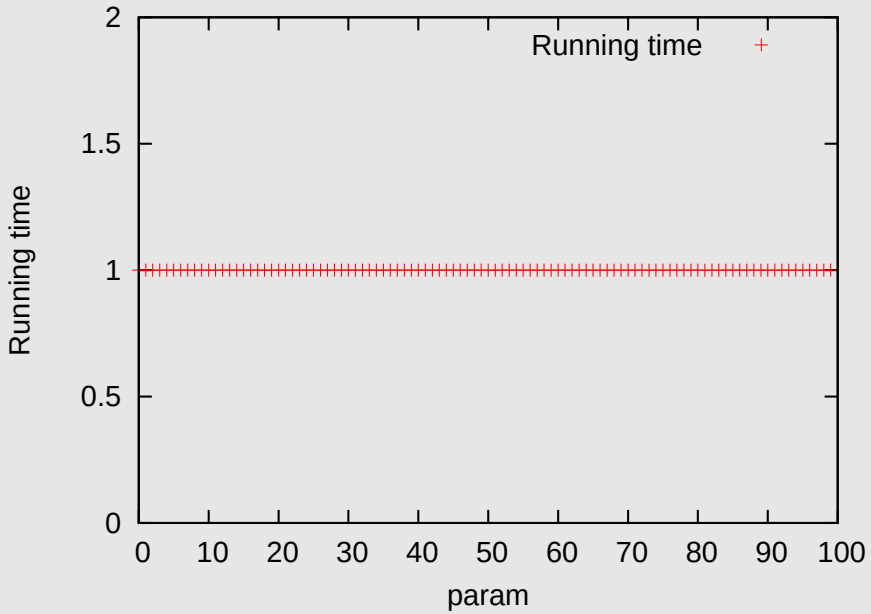
Branch Mispredictions – Motivation

- Input:
  - a – array of size 2×10^7, $a_i \in [1, \ldots, 100]$
  - $param$ – a threshold, $param \in [0, \ldots, 101]$
- Output:
  - g – # elements in a larger than $param$
  - s – # elements in a smaller or equal to $param$
- Algorithm:
  - Compare each element in a against $param$
  - Use a left-to-right scan



$param = 30$, $g = 11$, $s = 5$

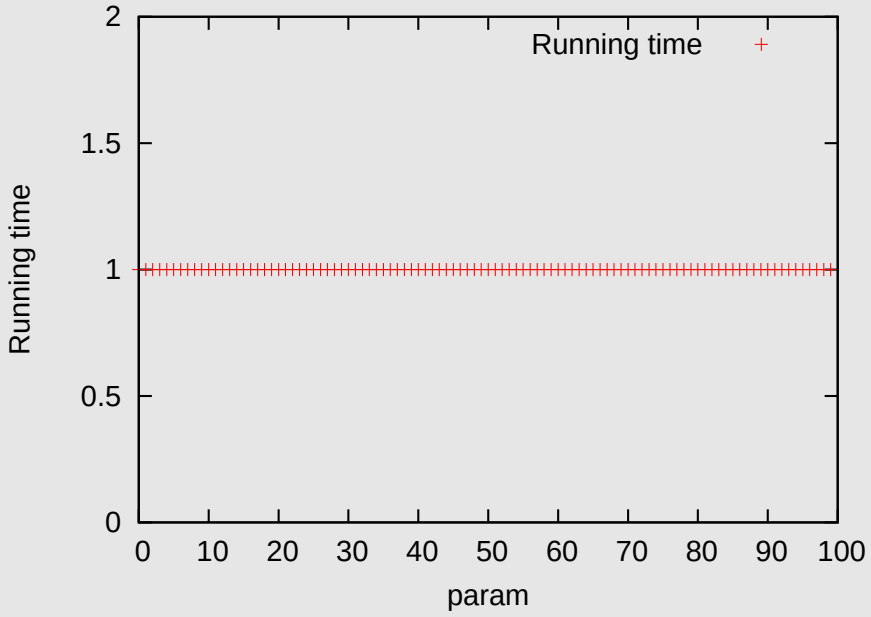
Running Time



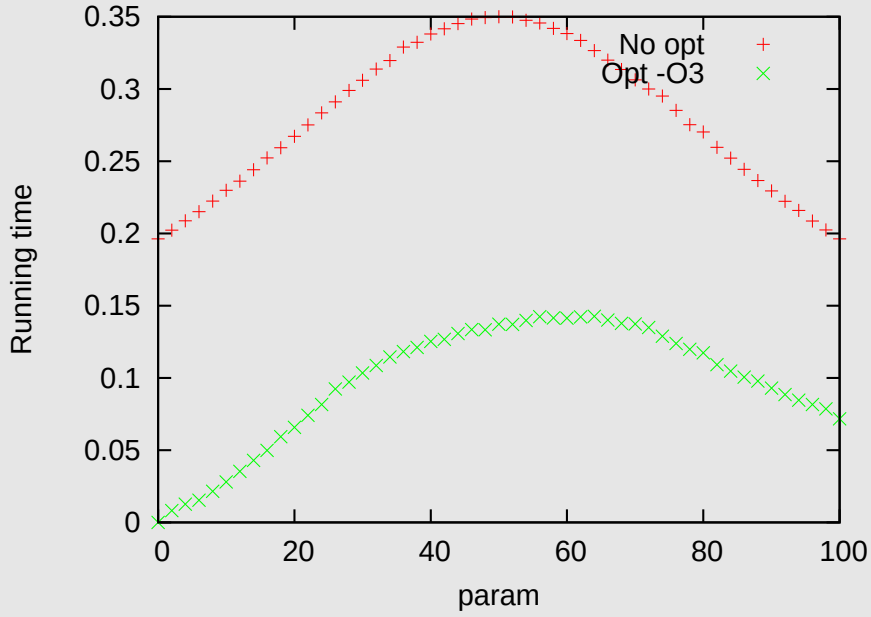
Theory

- The number of instructions is the same regardless of $param$

Running Time



Theory



Practice

Explanation: branch mispredictions!

Pipelining

Pipelining

- Each instruction is broken into several stages
- The smaller pieces can be executed in the same time
- Significant gains in running time


```
x=1;
y=y-1;
z=m+n;
if (t==0)
   printf(‘‘It’s zero’’);
else
   t=0;
```

Pipelining

- Each instruction is broken into several stages
- The smaller pieces can be executed in the same time
- Significant gains in running time

fetch decode get ops execute write

```
→x=1;
 y=y-1;
 z=m+n;
 if (t==0)
    printf(‘‘It’s zero’’);
 else
    t=0;
```

Pipelining

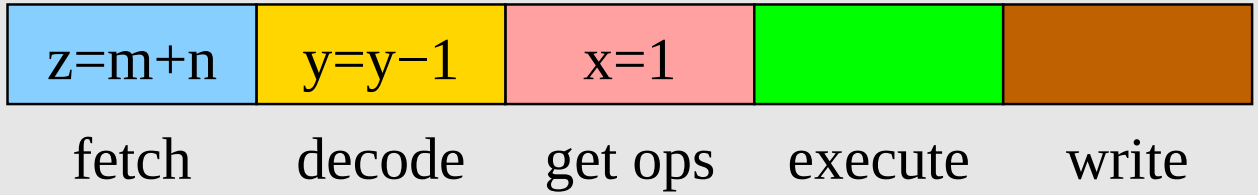
- Each instruction is broken into several stages
- The smaller pieces can be executed in the same time
- Significant gains in running time

fetch decode get ops execute write

```
  x=1;
→ y=y-1;
  z=m+n;
  if (t==0)
     printf(‘‘It’s zero’’);
  else
     t=0;
```

Pipelining

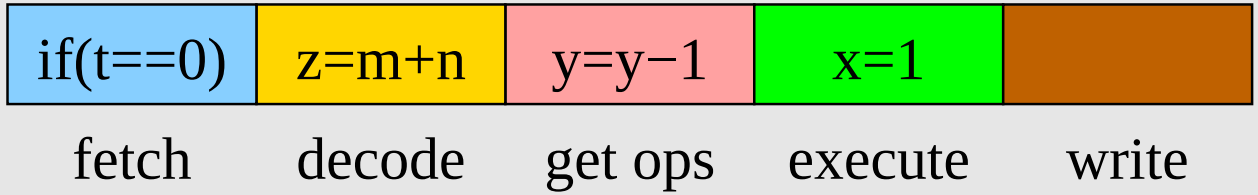
- Each instruction is broken into several stages
- The smaller pieces can be executed in the same time
- Significant gains in running time




```
x=1;
y=y-1;
→z=m+n;
if (t==0)
   printf(‘‘It’s zero’’);
else
   t=0;
```

Pipelining

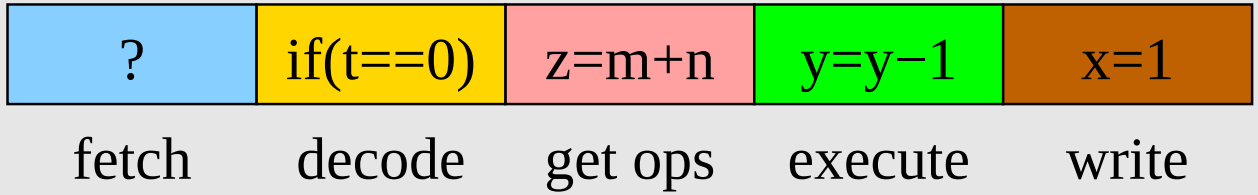
- Each instruction is broken into several stages
- The smaller pieces can be executed in the same time
- Significant gains in running time




```
  x=1;
  y=y-1;
  z=m+n;
→if (t==0)
     printf(‘‘It’s zero’’);
  else
     t=0;
```

Pipelining

- Each instruction is broken into several stages
- The smaller pieces can be executed in the same time
- Significant gains in running time

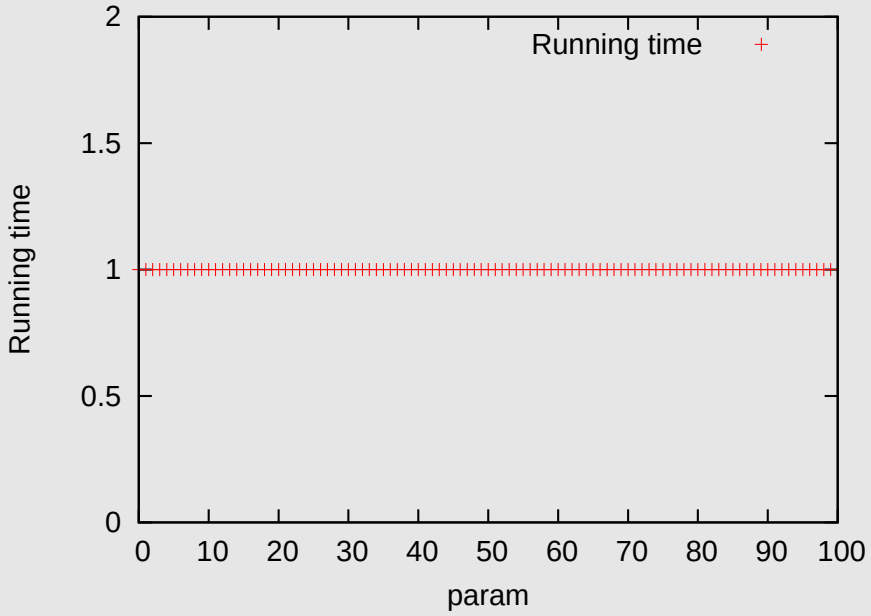



```
  x=1;
  y=y-1;
  z=m+n;
  if (t==0)
→    printf(‘‘It’s zero’’);
  else
→    t=0;
```

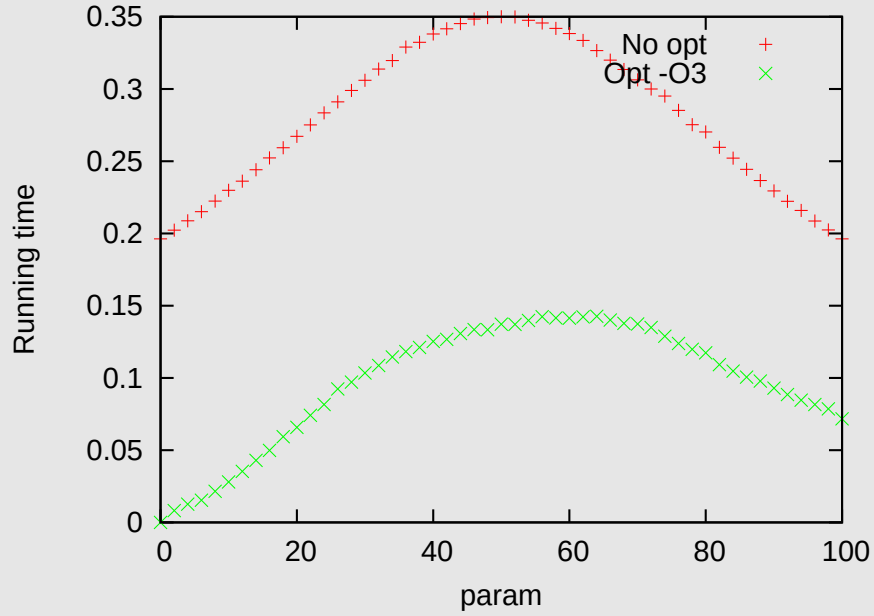
Branch Predictor

- Modern processors include branch predictors
- Attempts to predict the direction of each branch
- Accurate over 90% of the times
- Significant penalties upon mispredictions
- Pipelines are getting longer

Running Time



Theory



Practice

Many branch mispredictions for $param \approx 50$!

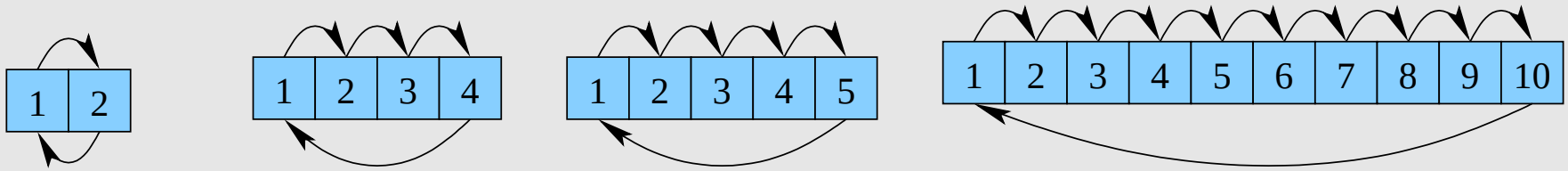
Memory Transfers and Streaming

Memory Hierarchy – Motivation

Simple algorithm:

- Consider an array of size n
- Perform r element accesses circularly
- n is a parameter, r is fixed

Accesses per element (apm) for $r = 20$:



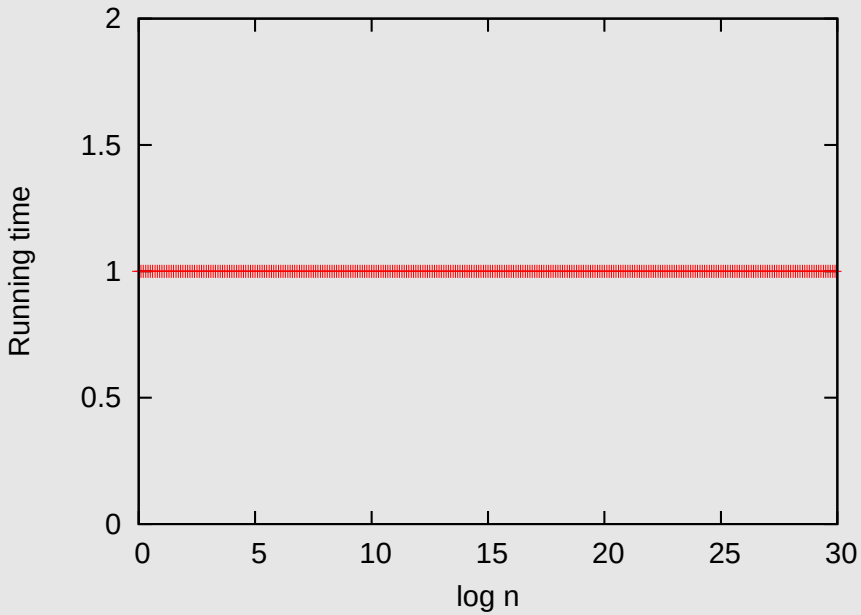
$n = 2$ $apm = 10$

$n = 4$ $apm = 5$

$n = 5$ $apm = 4$

$n = 10$ $apm = 2$

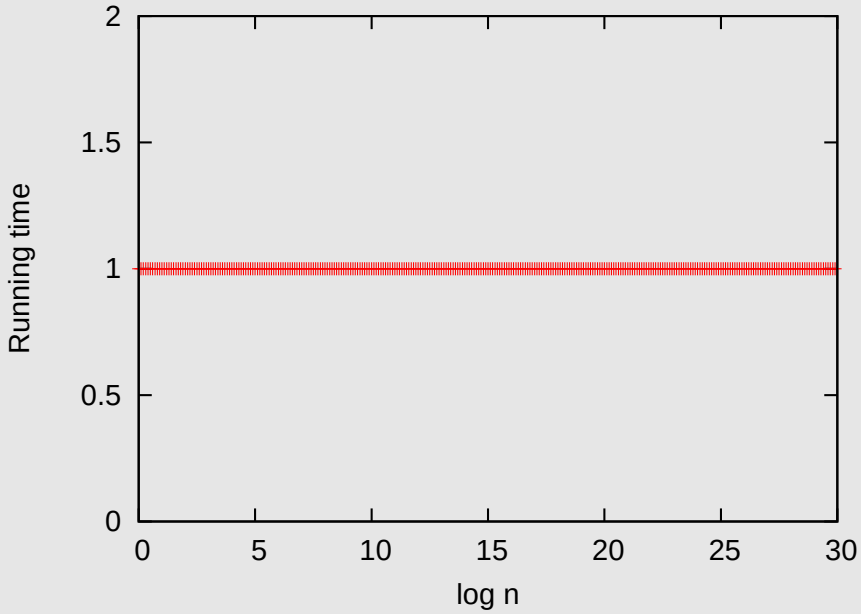
Running Time



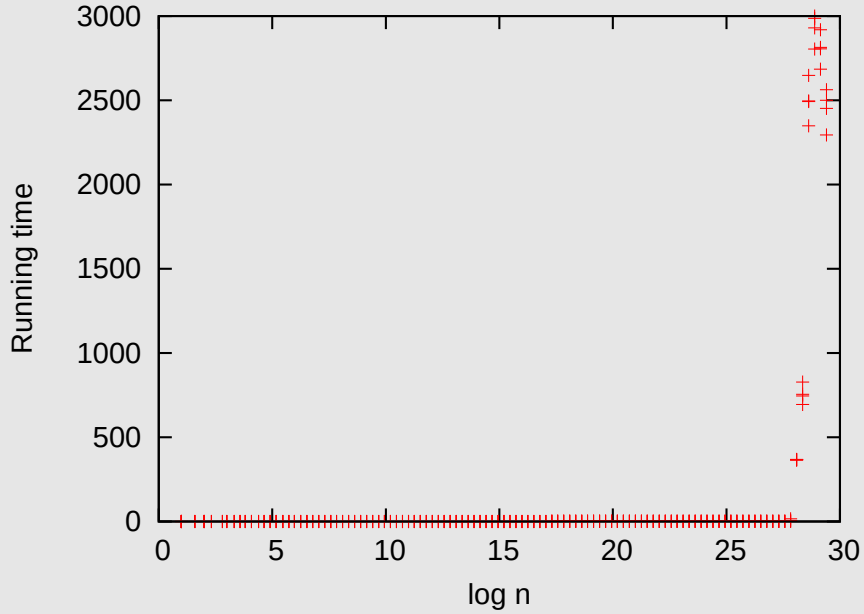
Theory

- The number of instructions is the same regardless of n
- The number of memory accesses is also the same
- Branch mispredictions don't stand in the way

Running Time



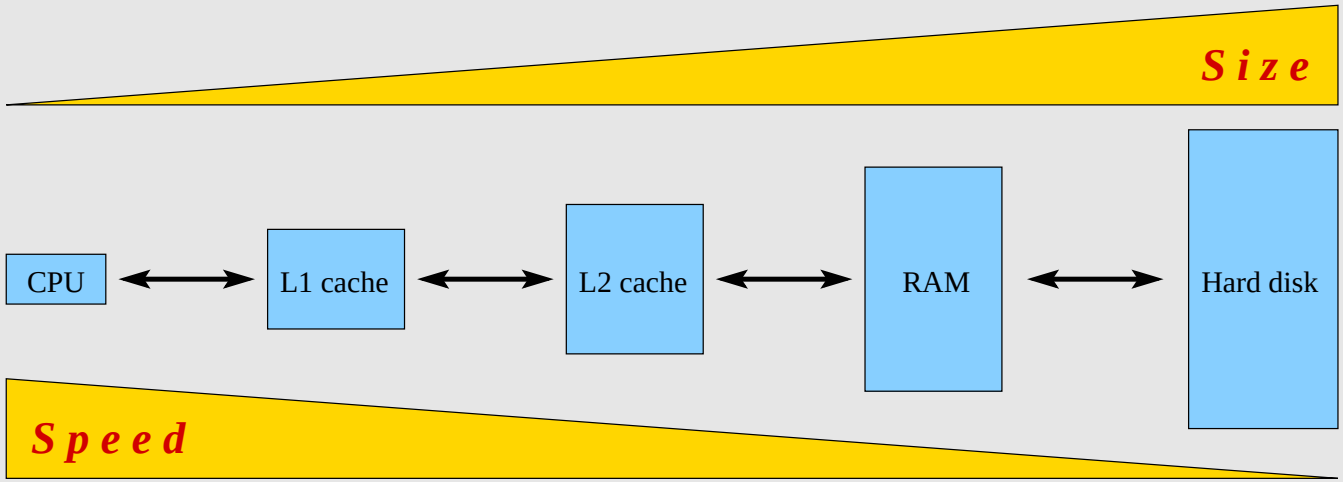
Theory



Practice

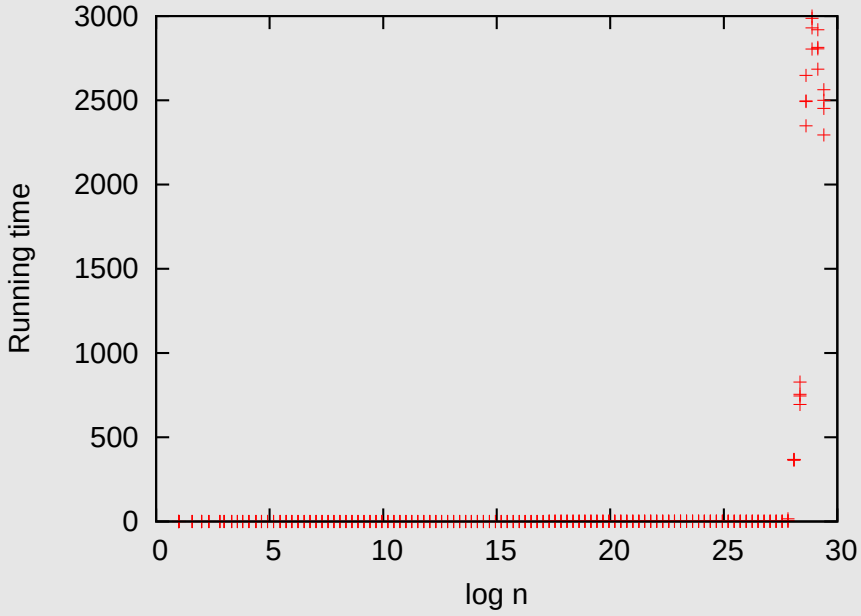
Explanation: memory hierarchy!

Memory Hierarchy

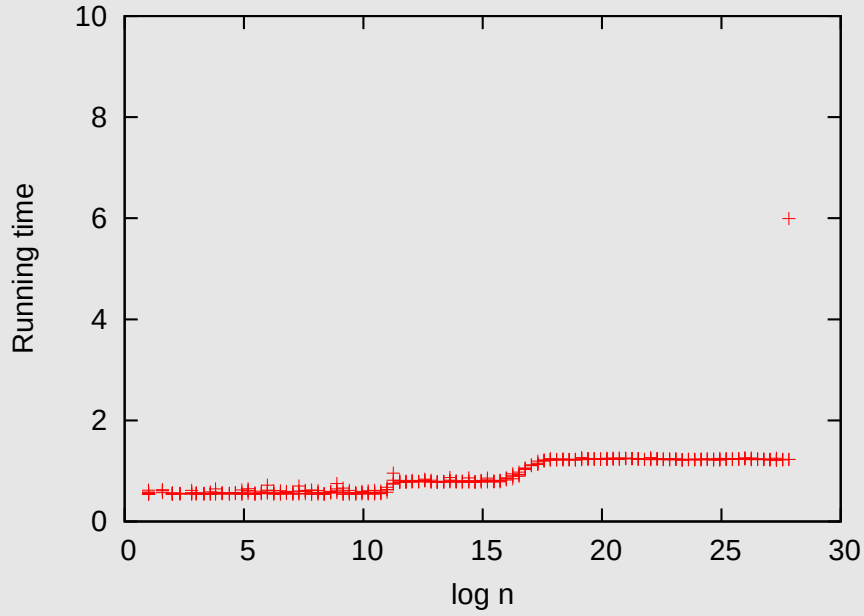


- Each level is larger and slower than the previous
- Transfers are done only between consecutive levels
- Transfer large blocks of data at once
- Real bottleneck: between memory and disk
- Bad news: data sets are getting huge

Running Time



Practice



Same chart zoomed

Many memory transfers when n exceeds memory!

Streaming

- Data access is done only sequentially
- Don’t want to store all data, use only small memory
- One pass streaming
  - Data comes on the fly: sensor data, IP monitoring
  - Use a single pass, get as much use of it as possible
- Multi pass streaming
  - Modern disks have high sequential access
  - A tempting approach for really massive data sets

Outline

- Hardware factors affecting the running time
  - Instructions performed by microprocessor
  - Branch mispredictions
  - Memory transfers
  - Streaming
- Hardware factors affecting the reliability
  - Memory corruptions
- Optimal resilient dictionaries

Soft Memory Errors

- Nowadays memories:
  - Small, complex, high frequency, low voltage
  - Price to pay - reliability
- Soft memory errors:
  - Bit flip, implying cell corruption
  - Caused by radiation, power failures, cosmic rays
- Good news:
  - Doesn't happen often (every few months)
- Bad news:
  - Happen often for large clusters
  - Soft memory error rate is increasing

Soft Memory Errors – Applications

[Govindavajhala and Appel '03]

Soft Memory Errors – Applications

[Govindavajhala and Appel '03]

Applications:

- Break JVM
- Insecure cryptographic protocols, smart-cards

Outline

- Hardware factors affecting the running time
  - Instructions performed by microprocessor
  - Branch mispredictions
  - Memory transfers
  - Streaming
- Hardware factors affecting the reliability
  - Memory corruptions
- Optimal resilient dictionaries

Contributions

1. **On the Adaptiveness of Quicksort.** G. S. Brodal, R. Fagerberg, and G. Moruz. In Proc. 7th Workshop on Algorithm Engineering and Experiments (ALENEX), 2005.
2. **Cache-Aware and Cache-Oblivious Adaptive Sorting.** G. S. Brodal, R. Fagerberg, and G. Moruz. In Proc. Int. Colloquium on Automata, Languages, and Programming, 2005.
3. **Tradeoffs Between Branch Mispredictions and Comparisons for Sorting Algorithms.** G. S. Brodal and G. Moruz. In Proc. 9th Int. Workshop on Algorithms and Data Structures (WADS), 2005.
4. **Skewed Binary Search Trees.** G. S. Brodal and G. Moruz. In Proc. 14th Annual European Symposium on Algorithms (ESA), 2006.
5. **Adapting Parallel Algorithms to the W-Stream Model, with Applications to Graph Problems.** C. Demetrescu, B. Escoffier, G. Moruz, and A. Ribichini. In Proc. 32nd Int. Symposium on Mathematical Foundations of Computer Science (MFCS), 2007.
6. **Resilient Priority Queues.** A. G. Jørgensen, G. Moruz, and T. Mølhave. In Proc. 10th Int. Workshop on Algorithms and Data Structures (WADS), 2007.
7. **Optimal Resilient Dynamic Dictionaries.** G. S. Brodal, R. Fagerberg, I. Finocchi, F. Grandoni, G. F. Italiano, A. G. Jørgensen, G. Moruz, and T. Mølhave. In Proc. 15th Annual European Symposium on Algorithms (ESA), 2007. To appear.

ESA '07

Submissions:

Optimal resilient dictionaries

G. S. Brodal, R. Fagerberg, A. G. Jørgensen, G. Moruz, and T. Mølhave

Resilient Search Trees: Randomization and Prejudice

I. Finocchi, F. Grandoni, and G. F. Italiano

Reviewers deciding:

Optimal resilient dictionaries

G. S. Brodal, R. Fagerberg, A. G. Jørgensen, G. Moruz, and T. Mølhave

Resilient Search Trees: Randomization and Prejudice

I. Finocchi, F. Grandoni, and G. F. Italiano

Acceptance notification:

Optimal resilient dictionaries

G. S. Brodal, R. Fagerberg, I. Finocchi, F. Grandoni,G. F. Italiano, A. G. Jørgensen, G. Moruz, and T. Mølhave

Faulty-Memory RAM

[Finocchi and Italiano '04]

- A regular RAM with possibly corrupted cells
- Bad news:
  - Memory corruptions occur at any time and at any place
  - Corruptions are performed by an adversary
  - Corrupted and uncorrupted cells can't be distinguished
  - No space increase (asymptotically)

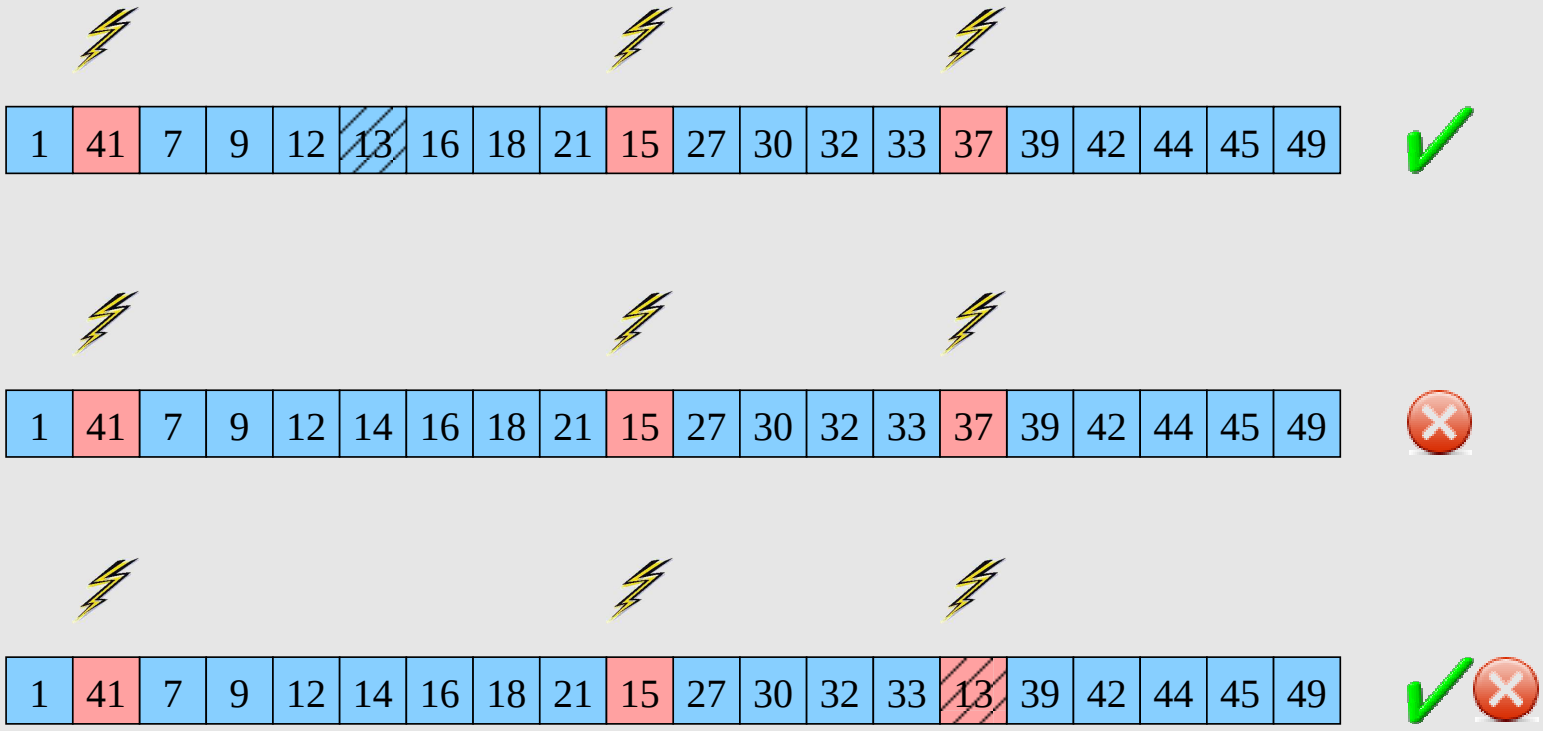
Faulty-Memory RAM

[Finocchi and Italiano '04]

- A regular RAM with possibly corrupted cells
- Bad news:
  - Memory corruptions occur at any time and at any place
  - Corruptions are performed by an adversary
  - Corrupted and uncorrupted cells can't be distinguished
  - No space increase (asymptotically)
- Good news:
  - Assumption: at most δ corruptions
  - $O(1)$ corruption-free cells (reliable CPU registers)

Resilient Algorithms

- Work correctly for uncorrupted values
- Searching:



Search key $e = 13$.

Resilient Results

[Finocchi and Italiano '04, Finocchi et al. '06, Finocchi et al. '07, Jørgensen et al. '07]

- Sorting: $\Theta(n \log n + \delta^2)$
- Static dictionaries:
  - Randomized: $\Theta(\log n + \delta)$ expected time
  - Deterministic: $\Omega(\log n + \delta)$, $O(\log n + \delta^{1+\epsilon})$ worst case
- Search trees: amortized $O(\log n + \delta^2)$ time per operation
- Priority queues: amortized $O(\log n + \delta)$ time per operation

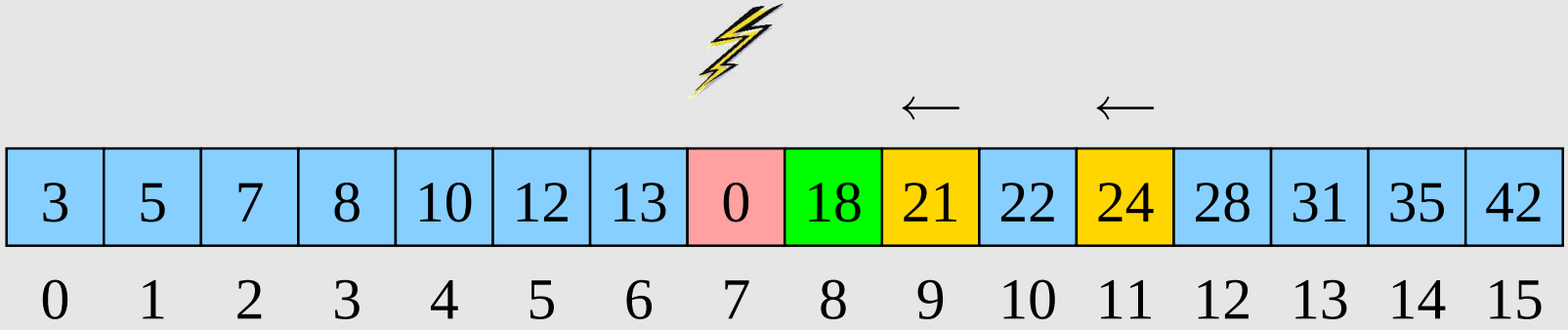
Our paper:

- Randomized static dictionary: $\Theta(\log n + \delta)$ expected time
- Deterministic static dictionary: $O(\log n + \delta)$ worst case time
- Deterministic dynamic dictionary: $O(\log n + \delta)$ worst case time for search, $O(\log n + \delta)$ amortized time for updates

Classical Binary Search

3	5	7	8	10	12	13	0	18	21	22	24	28	31	35	42
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

Classical Binary Search



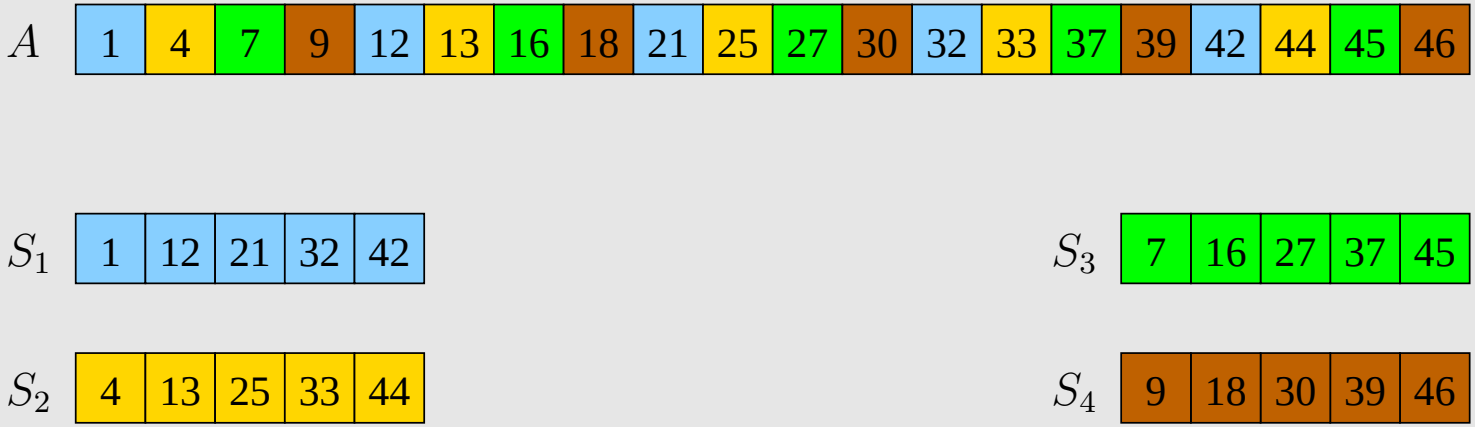
Search key $e = 3$

Problems

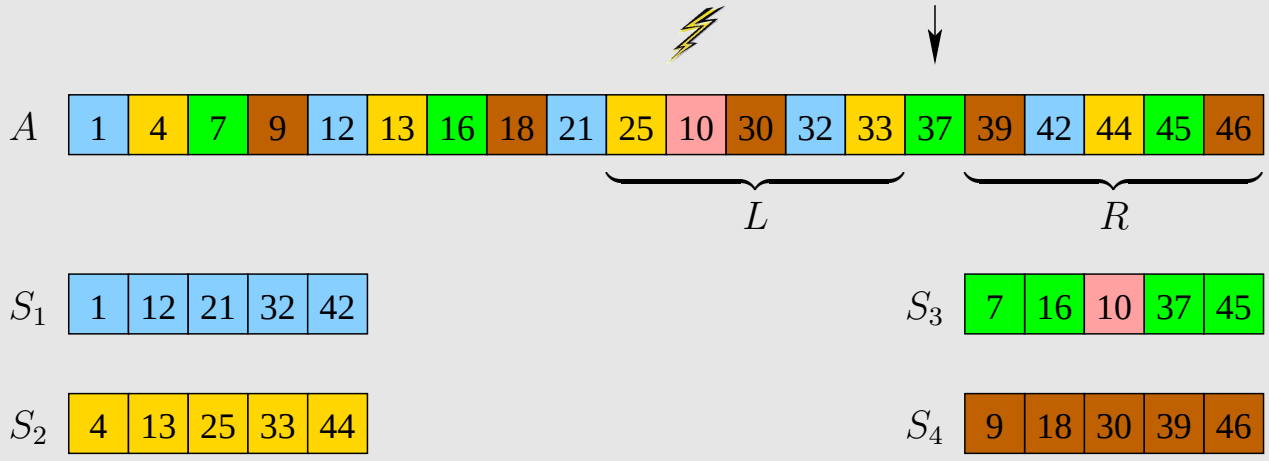
- The adversary can mislead the search
- The answer may be wrong
- The search may end very far from the correct location
- A single corruption suffices!!!

Randomized Static Dictionary

1. Split the input in 2δ disjoint sequences $S_1, \ldots, S_{2\delta}$
2. Perform a classic binary search on a random S_k
3. Check whether the search was not mislead by corruptions
4. If search was mislead restart from step 2. with a new S_k



The Magic Step 3



$$e = 13,\ \delta = 2,\ c_l = 1,\ c_r = 5$$

- $|L| = |R| = 2\delta + 1$
- c_l – # keys in L smaller than e
- c_r – # keys in R larger than e
- Restart if $c_l \leq \delta$ or $c_r \leq \delta$
- Scan all elements between L and R otherwise

Analysis

1. Split the input in 2δ disjoint sequences $S_1, \ldots, S_{2\delta}$
2. Perform a classic binary search on a random S_k
3. Check whether the search was not mislead by corruptions
4. If search was mislead restart from step 2. with a new S_k

Analysis

1. Split the input in 2δ disjoint sequences $S_1, \ldots, S_{2\delta}$
2. Perform a classic binary search on a random S_k
3. Check whether the search was not mislead by corruptions
4. If search was mislead restart from step 2. with a new S_k

- Step 2: $O(\log n)$ time and $O(\log \delta)$ random bits
- Step 3: $O(\delta)$ time
- Probability theory: expected at most two iterations
- Altogether: $O(\log n + \delta)$ time, $O(\log \delta)$ random bits

Analysis

1. Split the input in 2δ disjoint sequences $S_1, \ldots, S_{2\delta}$
2. Perform a classic binary search on a random S_k
3. Check whether the search was not mislead by corruptions
4. If search was mislead restart from step 2. with a new S_k

- Step 2: $O(\log n)$ time and $O(\log \delta)$ random bits
- Step 3: $O(\delta)$ time
- Probability theory: expected at most two iterations
- Altogether: $O(\log n + \delta)$ time, $O(\log \delta)$ random bits

Note

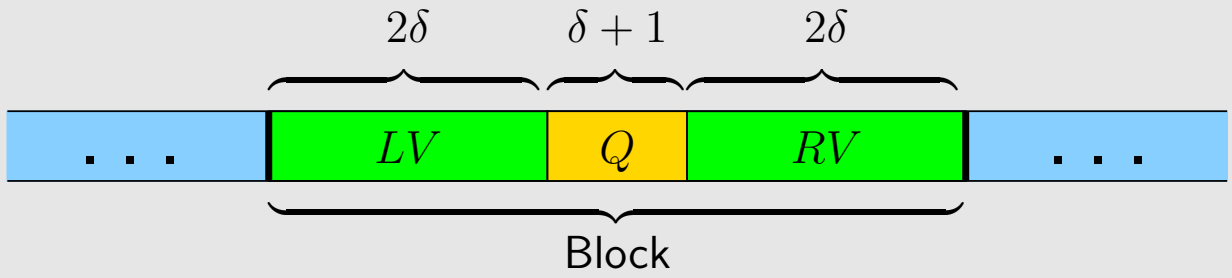
- Adaptive adversaries can compute index k of S_k!!!
- For adaptive adversaries: $O(\delta \log n)$ time

Deterministic Static Dictionary

High Level Picture

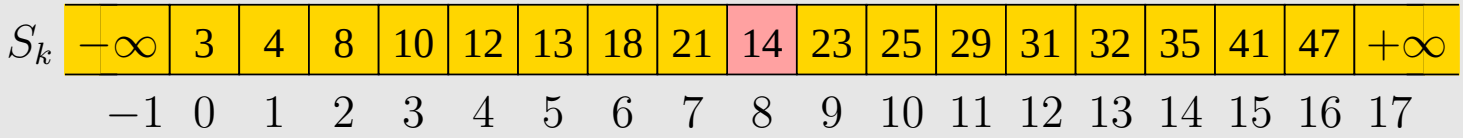
- Adapted binary search
  - Reuse the sub-sequencing idea
  - Perform adapted binary search on subsequences
  - Change the subsequence when identifying corruptions
  - A corruption forces it to advance one level
- Verification procedure
  - Checks whether the search was mislead by corruptions
  - Upon success takes $O(\delta)$ time
  - Upon failure takes $O(f)$ time and identifies $\Omega(f)$ errors
- Final scan
  - Performed once, check $O(\delta)$ elements

Structure



- Use different elements for search and verification
- Query segment Q:
  - Used only by the binary search
  - Defines subsequences $S_0, \ldots, S_{\delta+1}$
  - There is at least an S_k corruption-free
- Verification segments LV and RV
  - Used only by verification
  - Allow the use of a majority argument

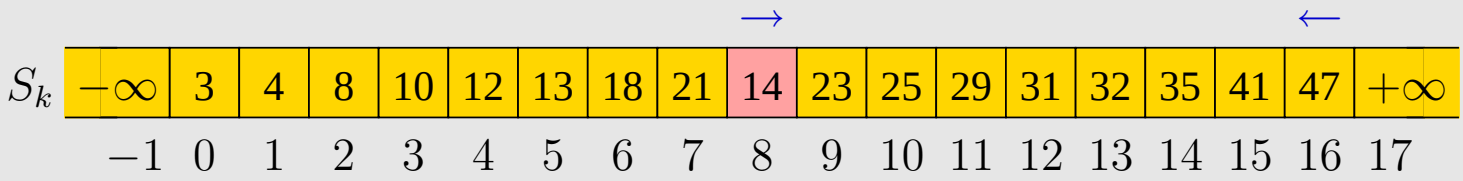
Adapted Binary Search



The search key $e = 21$.

- Check the next to last element in the pointed direction

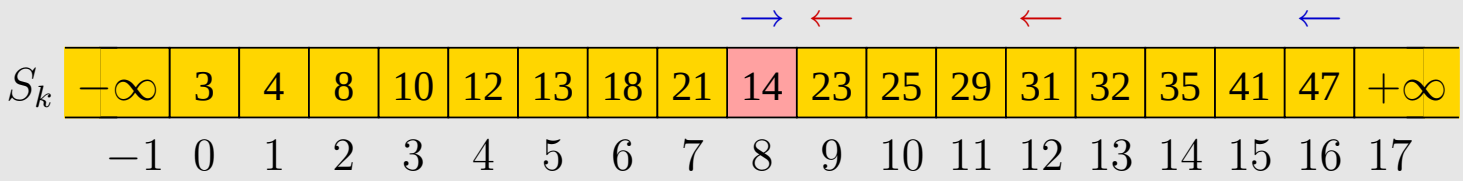
Adapted Binary Search



The search key $e = 21$.

- Check the next to last element in the pointed direction

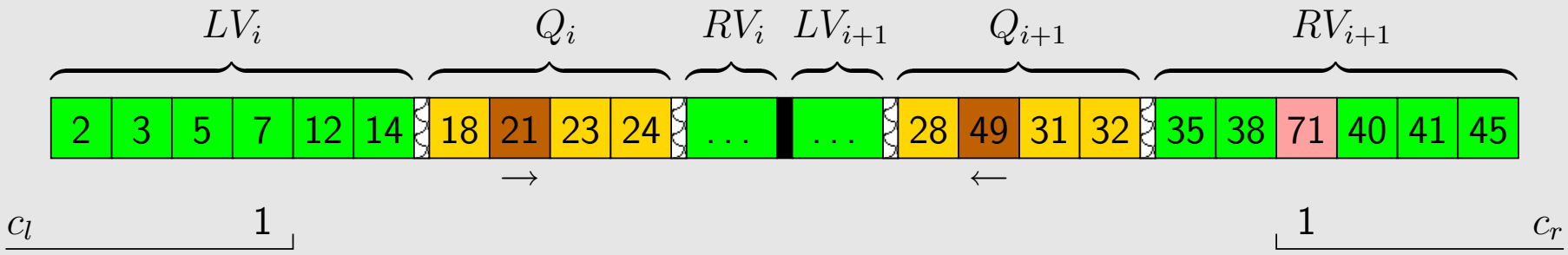
Adapted Binary Search



The search key $e = 21$.

- Check the next to last element in the pointed direction
- A corruption would be identified in the next step (unless another corruption occurs)
- Big idea: each step in the wrong direction corresponds to a corruption
- Conflict area: search key must be there or corruption
- Call verification procedure on conflict area:
  - Succeeds: search key must be there, scan two blocks
  - Fails: Backtrack the search on a different S_k

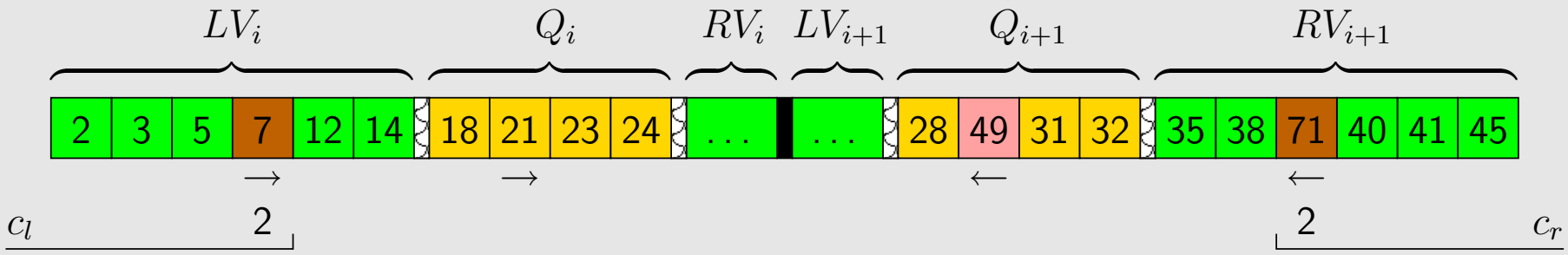
Verification procedure



Search key $e = 45$, $\delta = 3$, # corruptions found $k = 1$

- Performed on LV_i and RV_{i+1}
- c_l – confidence that e is to the right of LV_i
- c_r – confidence that e is to the left of RV_{i+1}

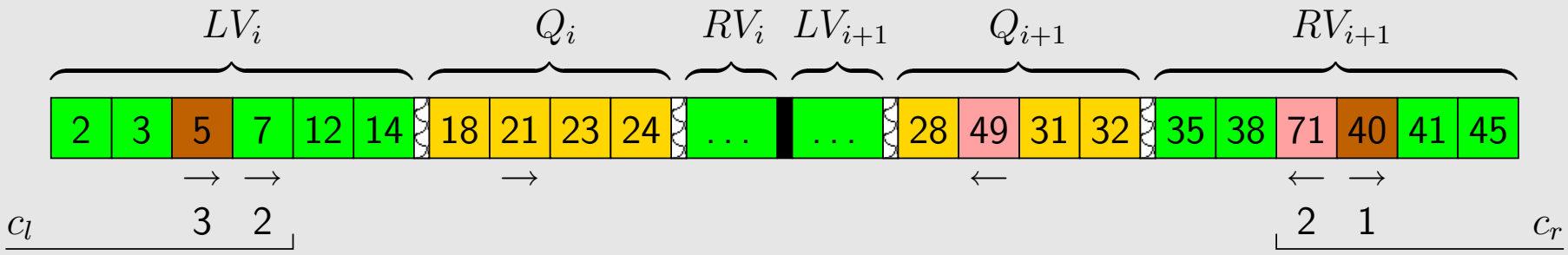
Verification procedure



Search key $e = 45$, $\delta = 3$, # corruptions found $k = 1$

- Performed on LV_i and RV_{i+1}
- c_l – confidence that e is to the right of LV_i
- c_r – confidence that e is to the left of RV_{i+1}

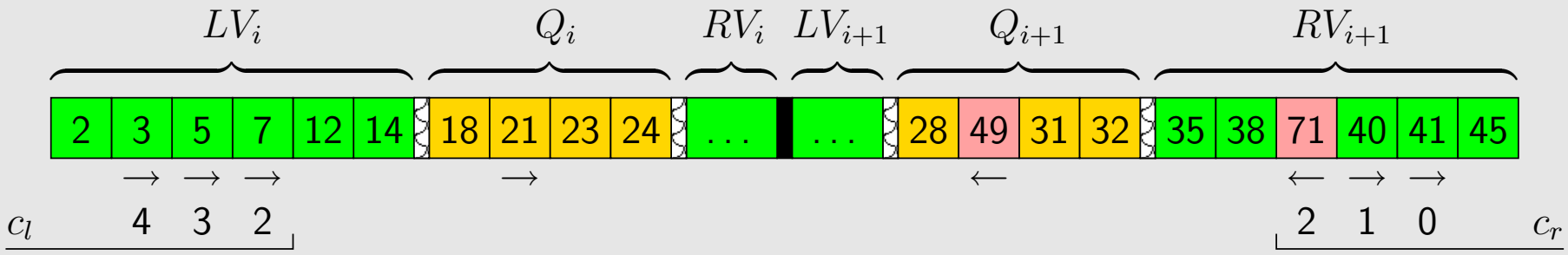
Verification procedure



Search key $e = 45$, $\delta = 3$, # corruptions found $k = 1$

- Performed on LV_i and RV_{i+1}
- c_l – confidence that e is to the right of LV_i
- c_r – confidence that e is to the left of RV_{i+1}

Verification procedure



Search key $e = 45$, $\delta = 3$, # corruptions found $k = 1$

- Performed on LV_i and RV_{i+1}
- c_l – confidence that e is to the right of LV_i
- c_r – confidence that e is to the left of RV_{i+1}
- Fails if $c_l = 0$ or $c_r = 0$, succeeds otherwise
- $2f$ elements visited in each segment to detect f errors
- Start $2k$ positions away from end of the each segment

Analysis

- Adapted binary search:
  - Corruption-free: $O(\log n)$
  - Time spent in wrong direction: $O(f)$ for f corruptions
- Verification:
  - A single verification: $O(f)$ time for f corruptions
  - All verifications: $O(\delta)$
- Final scan: $O(\delta)$ time to scan two blocks

Altogether:

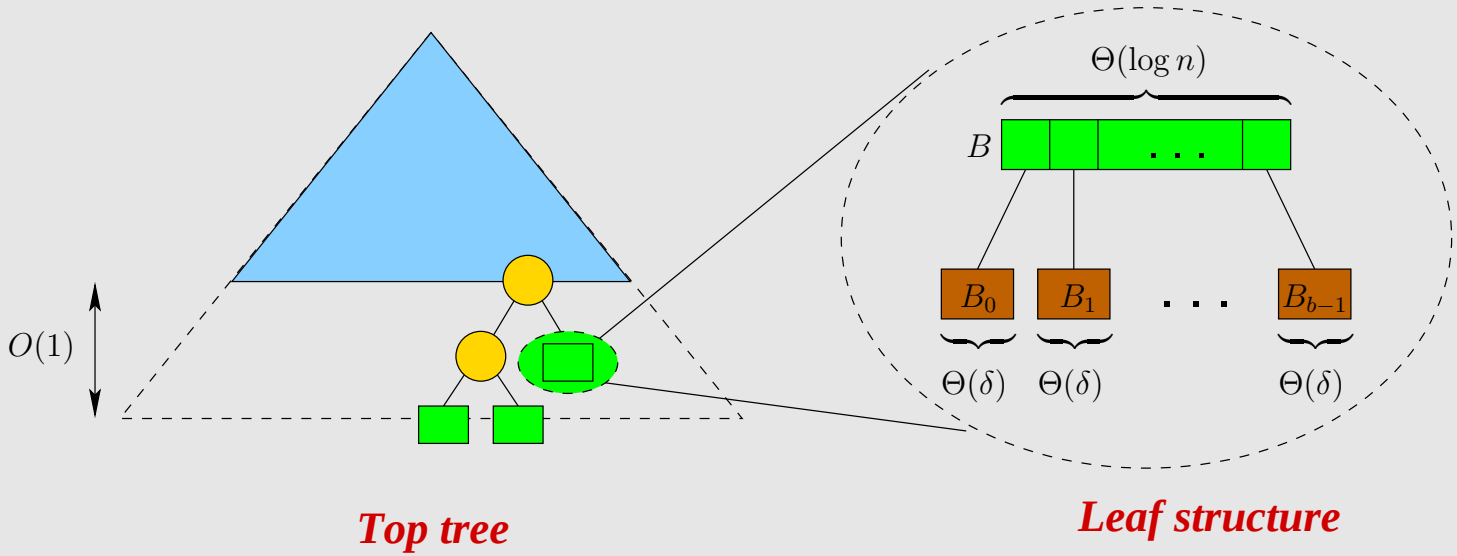
The resilient static deterministic dictionary supports searches in $O(\log n + \delta)$ time.

Dynamic Deterministic Dictionary

Reliable Value

- Stored in unreliable memory, retrieved reliably
- Uses $O(\delta)$ time and $O(\delta)$ space
- Replicate the given value $2\delta + 1$ times
- Retrieve during a scan using a majority argument
  - Keep in safe memory a candidate element and a counter
  - Increase counter when encountering a matching element
  - Decrease counter when encountering a different element
  - Discard candidate when counter becomes zero

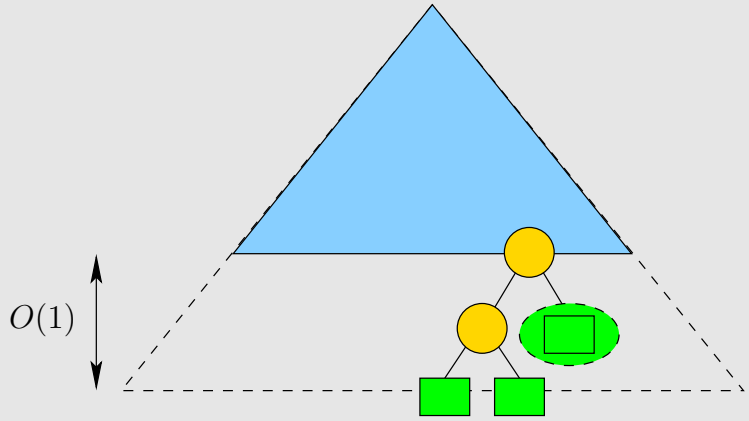
Dynamic Dictionary – Structure



- Top tree
  - Introduced in [Brodal et al. '02]
  - Stores only guiding elements, not input elements
- Leaf structure
  - Consists of $\Theta(\log n)$ buckets and a top bucket
  - Only $B_0, \ldots, B_{b-1}$ contain input elements

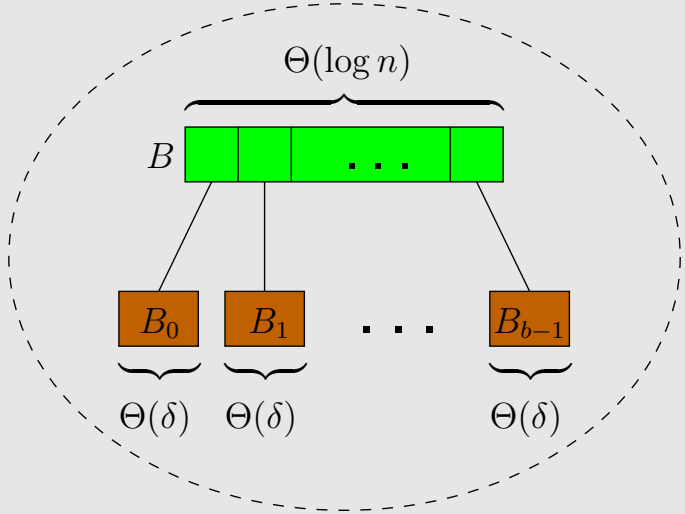
Top Tree

[Brodal et al. '02]



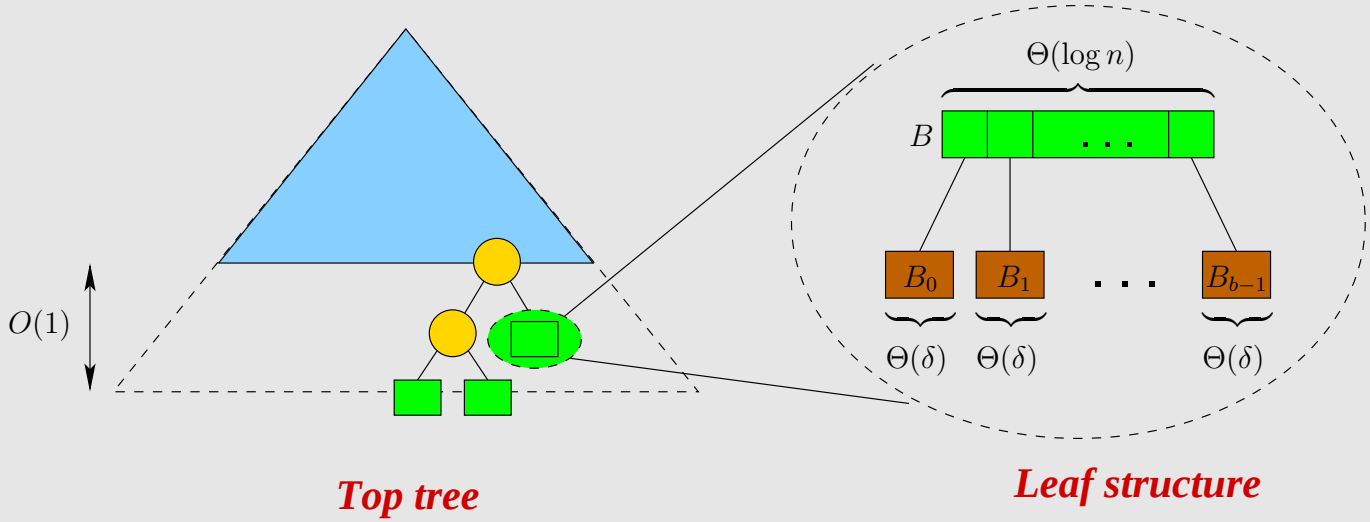
- Common knowledge:
  - Has height $\log |T| + O(1)$, can be laid in BFS order
  - Supports updates in amortized $O(\log^2 |T|)$ time
- We store it reliably:
  - Updates cost becomes amortized $O(\delta \log^2 |T|)$ time

Leaf Structure



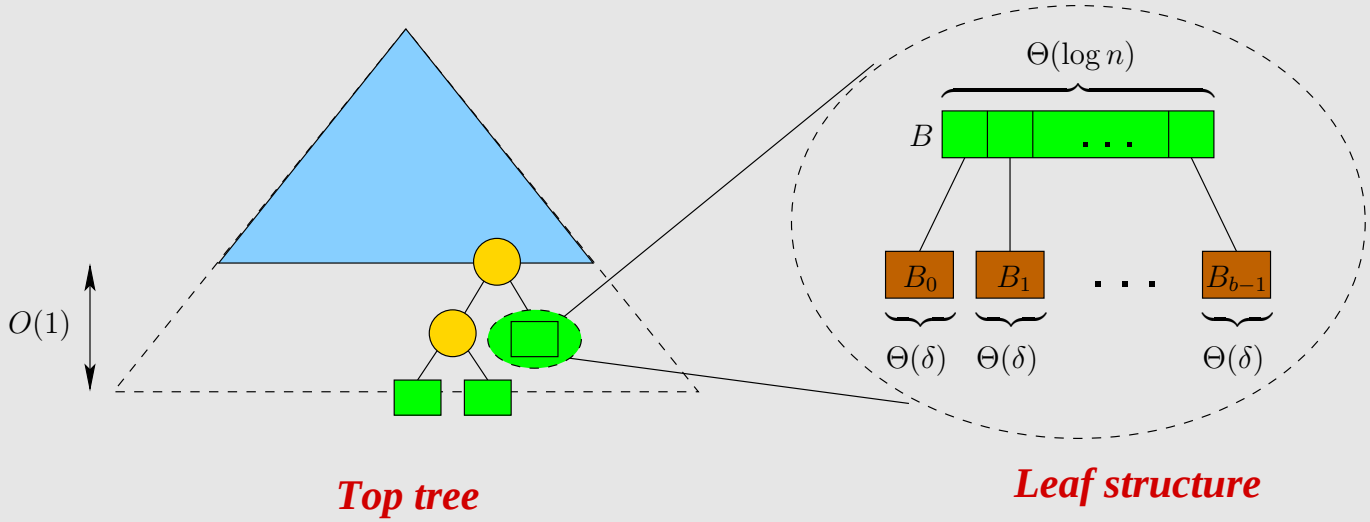
- Stores $\Theta(\delta \log n)$ input elements
- Each bucket B_i store $\Theta(\delta)$ input elements
- Top bucket contains guiding elements stored reliably

Searches



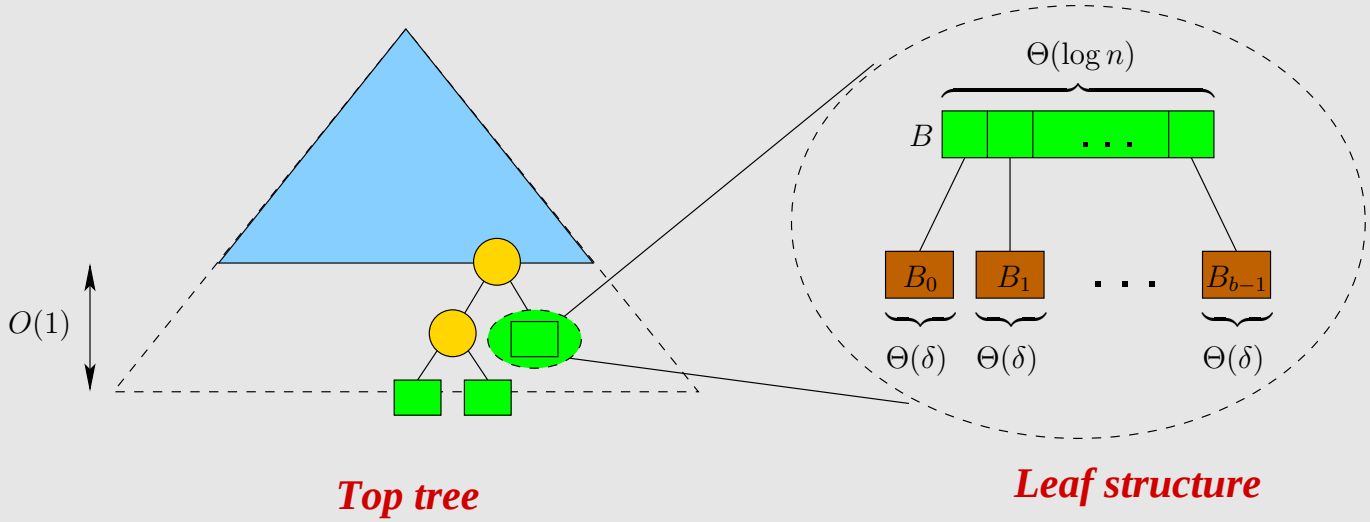
- Search the last level of internal nodes in the top-tree, and identify two consecutive nodes
- Search reliably the $O(1)$ remaining nodes
- Search the top bucket, identify some bucket B_i
- Scan B_i and report result

Searches



- Search the last level of internal nodes in the top-tree, and identify two consecutive nodes
- Search reliably the $O(1)$ remaining nodes
- Search the top bucket, identify some bucket B_i
- Scan B_i and report result
- Time: $O(\log n + \delta)$ worst case.

Updates



- Use standard bucketing techniques
  - Split/merge buckets each $\Omega(\delta)$ operations
  - Insert/delete new elements in B each $\Omega(\delta)$ operations
  - Insert/delete new elements in the top tree each $\Omega(\delta \log n)$ operations
- Time: $O(\log n + \delta)$ amortized for insertions and deletions.

Conclusion

Will theory catch practice?