A Faster External Memory Priority Queue with DecreaseKeys

[Larsen, Jiang] SODA19'

Aviel Zecharia

Authors

Shunhua Jiang

Kasper Green Larsen

Agenda

- Background
  - Definitions
  - The computational model
- Applications & Motivation
- Previous results overview
- A faster priority queue

1.

Background

Priority Queue [RAM model]

- Fibonacci heap (Strict-Fibonacci for de-amortization)
  - DeleteMin() – $O(\log(N))$
  - FindMin() – $\theta(1)$
  - Insert(k, p) – $\theta(1)$
  - DecreaseKey(k, p) – $\theta(1)$
- Many interesting implementations
  - Tournament Tree

External Memory Model

- The cost of an algorithm is the number of memory transfers required, operations on cached data are considered free
- The parameters of the model are as follow:
  - N – The total size of the problem in external-memory (`HDD`)
  - M – The number of **items** can fit into the local memory (`RAM`)
  - B – The number of **items** in one block of data (`PAGE`)
- Cache-Oblivious Model
  - M and B are not explicit known (multi-level memory hierarchy)
  - Won't be discussed in this session

FOLKLORE

- Clearly, T(N) memory accesses algorithm in the RAM model can be trivially converted into T(N) memory transfers – by ignoring locality
- Optimal Scanning
  - O(N/B) by continuous scanning (B-factor improvement 😉)
- Optimal Searching
  - O($log_B(N)$) using B-Trees (log(B)-factor improvement 😒)
- Optimal Sorting
  - O(${}^N/_B\, log_{{}^M/_B}({}^N/_B)$) using M/B merge-sort
  - computationally equivalent to priority queues (in both models)

Known Results

[Kumar, Schwabe 96']

- Priority Queue (with DecreaseKey)
  - FindMin() – $\theta(1)$
  - DeleteMin, Insert, DecreaseKey – O($1/B log_2(N/B)$) amortized
  - Can we do better ?

2.

Applications

Your OS !

Matrix scanning

✗
```
for i in 1 to ROW:
    for j in 1 to COL:
        ACCESS(MAT, i, j)
```

✗
```
for j in 1 to COL:
    for i in 1 to ROW:
        ACCESS(MAT, i, j)
```

1	2	3	4	B_0
5	6	7	8	B_1
9	10	11	12	B_2
13	14	15	16	B_3

Your FS !

Modern file-systems (Ext4/BTRFS/…)

- ✗ Using B-tree variations for disk blocks indexing
- ✗ Only 3-4 levels for full HDD indexing
    - ✗ Optimal number of I/O operations

Algorithms

- Dijkstra SSSP
  - O(V) times DeleteMin
  - O(E) times DecreaseKey
- Prim MST
  - O(V) times DeleteMin
  - O(E) times DecreaseKey
- etc.

3.

Previous Results

An I/O-efficient tournament tree

[Kumar, Schwabe 96']

- “Static” tournament tree with θ(N/M) leaves -> $\theta(log_2(N/M))$ height
- Each leaf represent the i’th key with initialized priority of infinity
- Each internal node conatins O(M) memory:
  - ‘Winners’ buffer of size M
  - **‘Signal’ buffer of size M (order matters !)**
- The root node is always loaded in local memory
- Invariants
  - Winners buffer (real) size must be greater than M/2
  - Signals buffer size must be smaller than M

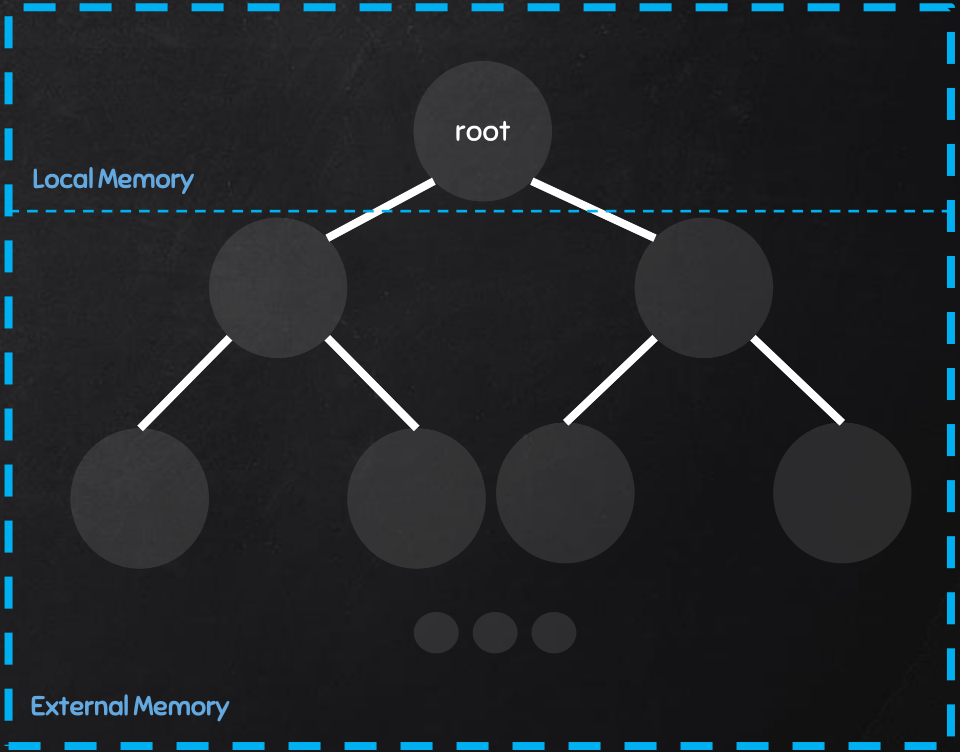
DS Overview

Winners

O(M)

(3, 5)	(2, 7)	
U(3,5)	D(7)	U(8,2)

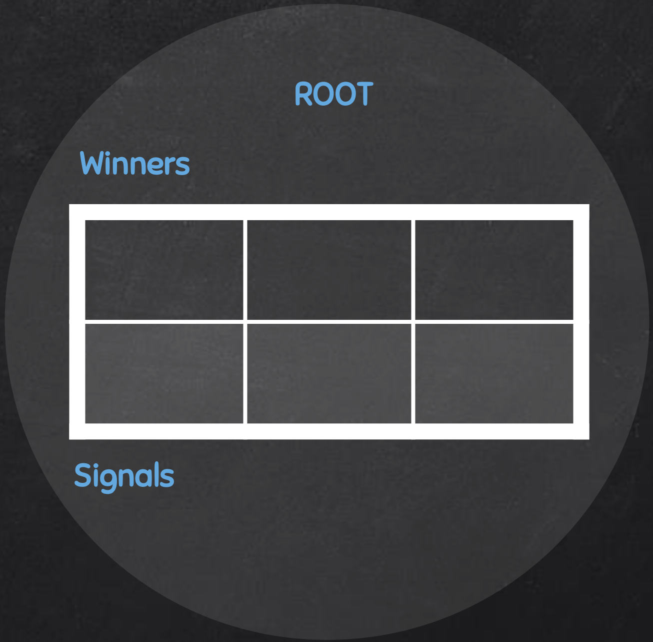
Signals



Update Key with Priority Signal

Changes key's priority only if the new priority is smaller than the old one

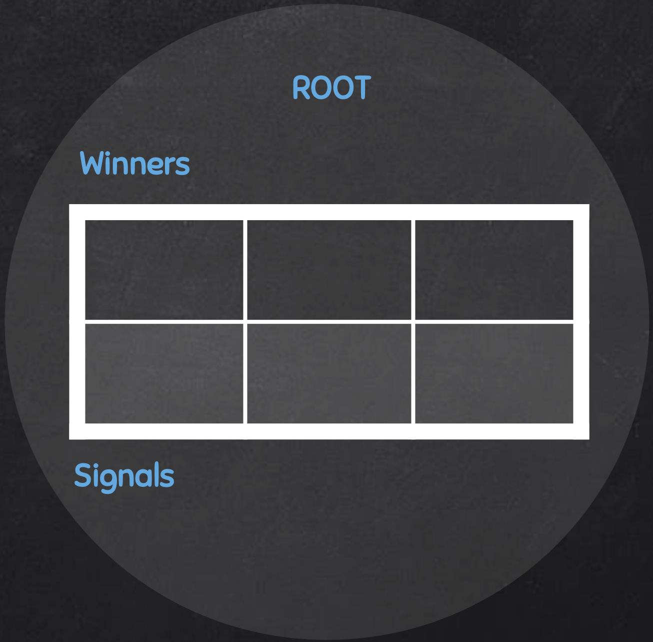
- If the key exists in the winners buffer
  - Try to update key priority, otherwise drop the signal
- Otherwise, if there is a greater priority key'
  - Replace (maximal priority) key' with key in the winning buffer
- Add the operation to the signals buffer
- If the signals buffer is full
  - *Empty* procedure


ROOT
Winners
Signals

Delete Key (and ExtractMin) Signal

Changes the priority of the key to infinity.

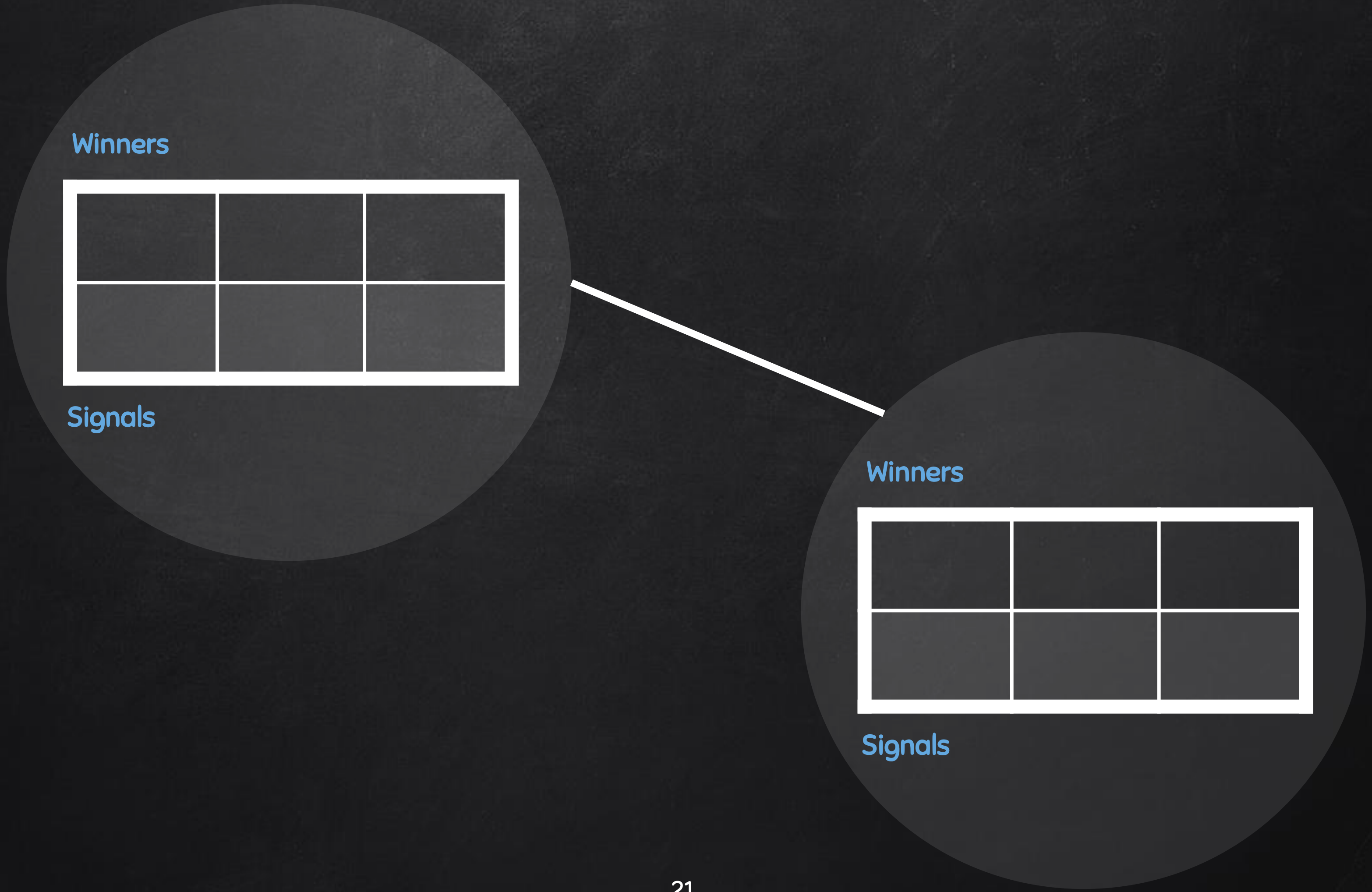
- If the key exists in the winners buffer
  - Delete the key from the winners buffer
- Add the operation to the signals buffer
- If the signals buffer is full
  - *Empty* procedure
- If the winners buffer is too sparse
  - *fill-up* procedure


ROOT
Winners
Signals

Empty Procedure

Signal buffer is full and must be emptied

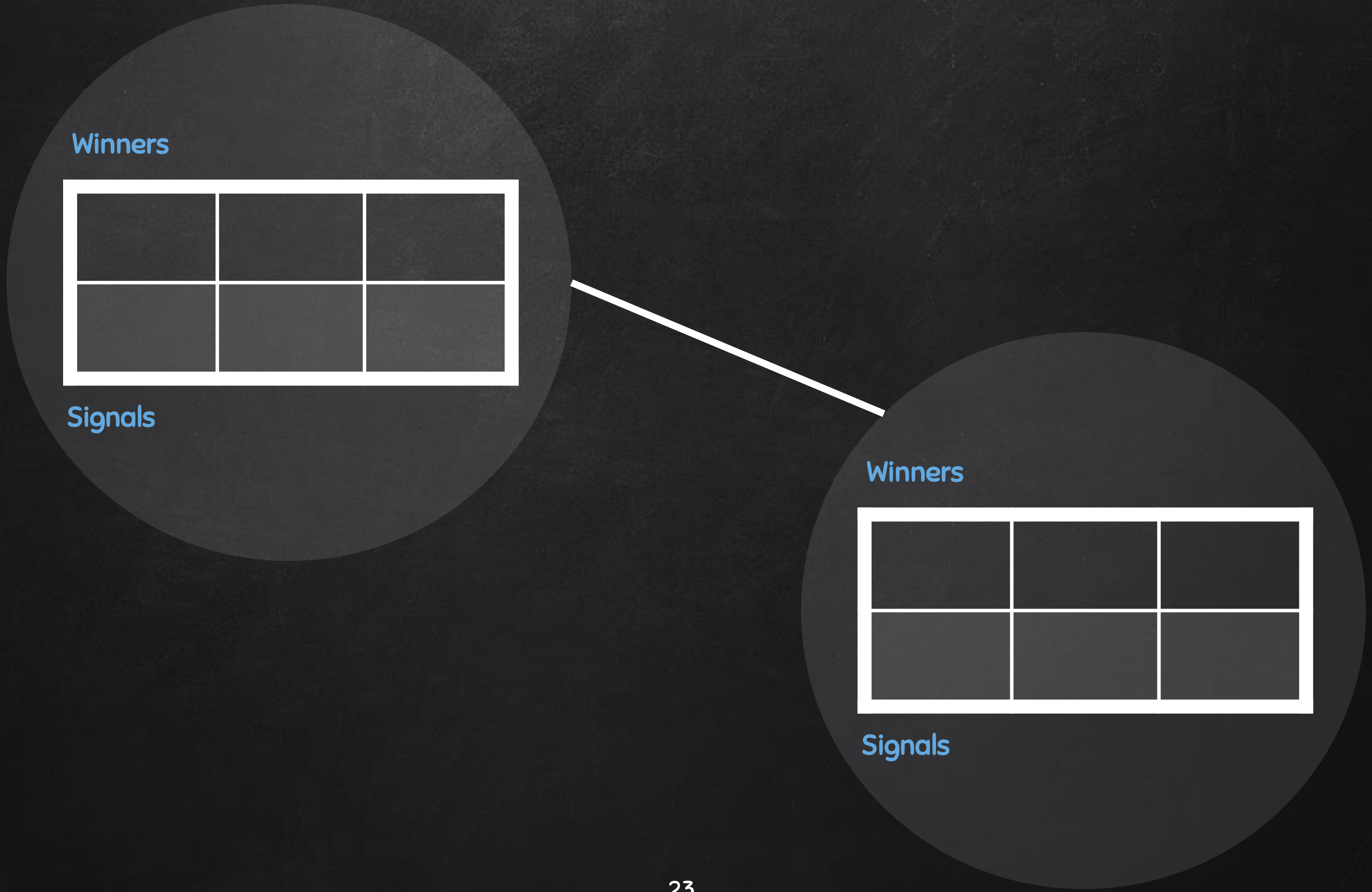
- ✗ Forward signals to the children
  - Load children to memory
  - Apply & push parent signals
  - Signals order must be kept (FIFO)
- ✗ Recursively


Winners
Signals
Winners
Signals

Fill-Up Procedure

Winners buffer is too sparse and must be filled

- ✗ *Empty* procedure
  - In order to force parent signals before children loading.
- ✗ Load & Update M elements from children
  - Minimal priorities.
  - Parent signals have already applied !
- ✗ Recursively


Winners
Signals
Winners
Signals

Amortized Analysis

Standard credits argument

- ✗ Real signal logic does not cost any I/O's
  - Anyway, we will pay O(1/B) for each vertex in the path to Leaf(K)
- ✗ Empty procedure called after O(M) new signals
  - Signal may only prorogate down
- ✗ Fill-Up procedure called after O(M) delete signals occurred
  - No signals are propagate up
- ✗ On an IO-efficient tree with N elements, a sequence of k operations (Delete\DeleteMin\Update) requires at most O($\mathrm{K}/\mathrm{B}log_2(N/B)$) I/O's

4.

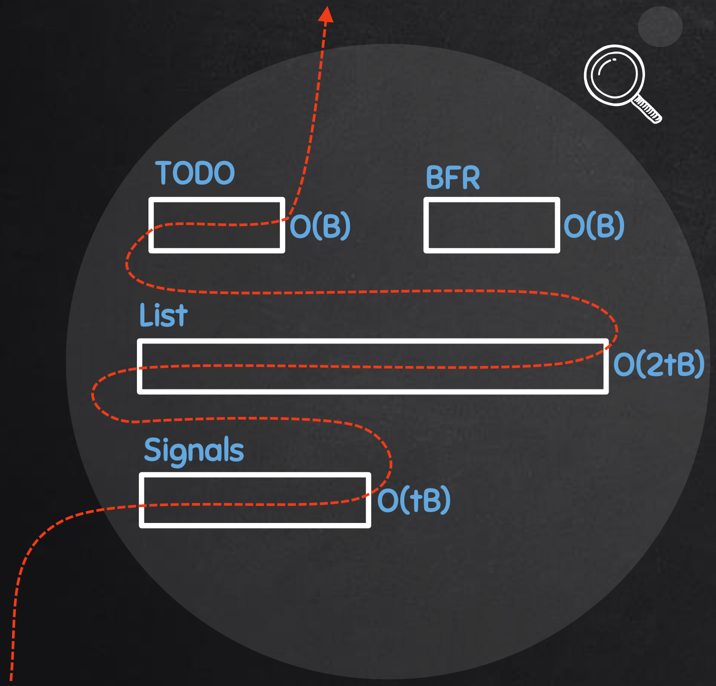
A Faster Priority Queue

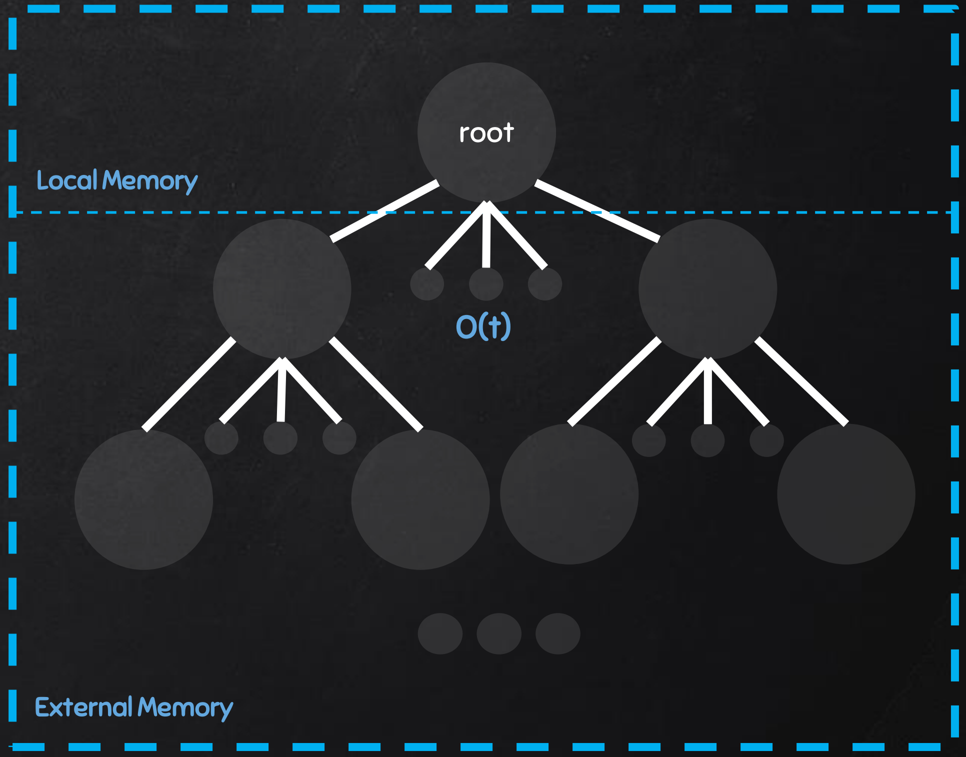
A Faster Priority Queue

[Larsen, Jiang 19']

- “Static” t-ary tree with θ(N/tB) leaves -> θ($log_2(N/tB)$) height
  - t~$log^{0.01}N$
- Each leaf represent the i’th key with initialized priority of infinity
- Each internal node conatins O(tB) memory:
  - List of elements of size 2tB
  - Signals buffer of size tB
  - **Bloom Filter Replacement of elements of size B**
  - **TODO buffer of size B**
- The root node is always loaded in local memory

DS Overview


TODO
O(B)
BFR
O(B)
List
O(2tB)
Signals
O(tB)


root
Local Memory
O(t)
External Memory

Bloom Filter Replacement

[Pagh, Rao 05']

- Insertion & Deletion can be done in amortized expected constant time
- The data structure can only make false positive errors with probability at most ε
- The space usage is at most $\left(1 + \mathrm{o}(1)\right)\mathrm{n}log\, {}^{1}/_{\varepsilon} + O(n + w)$ **bits**
- In our case
  - $\mathrm{n} = 2\mathrm{tB}, t = log^{0.01}N, w = logN, \varepsilon = {}^{1}/_{log^{3}N}$
  - Space complexity O(B) **words**

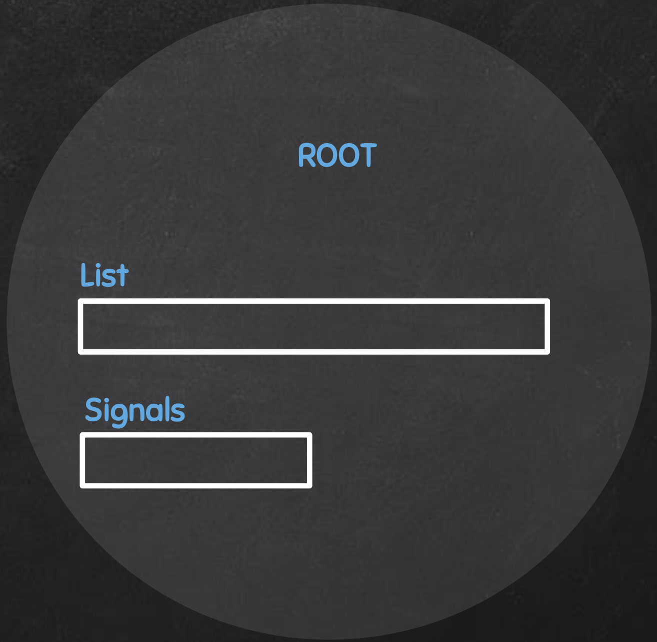
Invariants

- Each key is uniquely stored in the tree
  - Otherwise, there must exist Delete signals that will delete the redundant copies & lower update signals
- TODO buffer may only contain single Delete | Update | Delete->Update
- The deeper you go, the lower the priority

Delete Key (and ExtractMin) Signal

Changes the priority of the key to infinity.

- ✗ If there exists an entry with key k in the root
  - Delete the key from the list
- ✗ Otherwise
  - Push Delete signal


ROOT
List
Signals

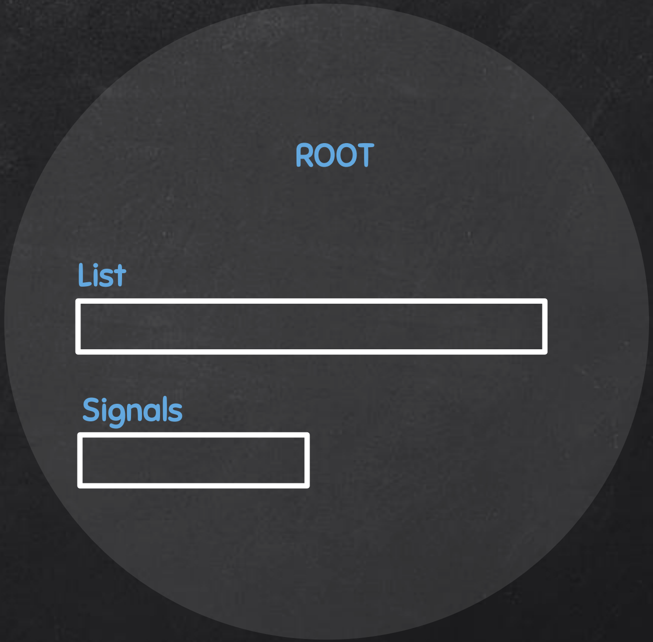
Update Key with Priority Signal

Changes key's priority only if the new priority is smaller than the old one

✗ If there exists an entry (k, ρ') in the list of the root
  - Update to $(k, \min(\rho, \rho'))$

✗ Otherwise, if $\rho >$ Boundary(root)
  - Push Update signal

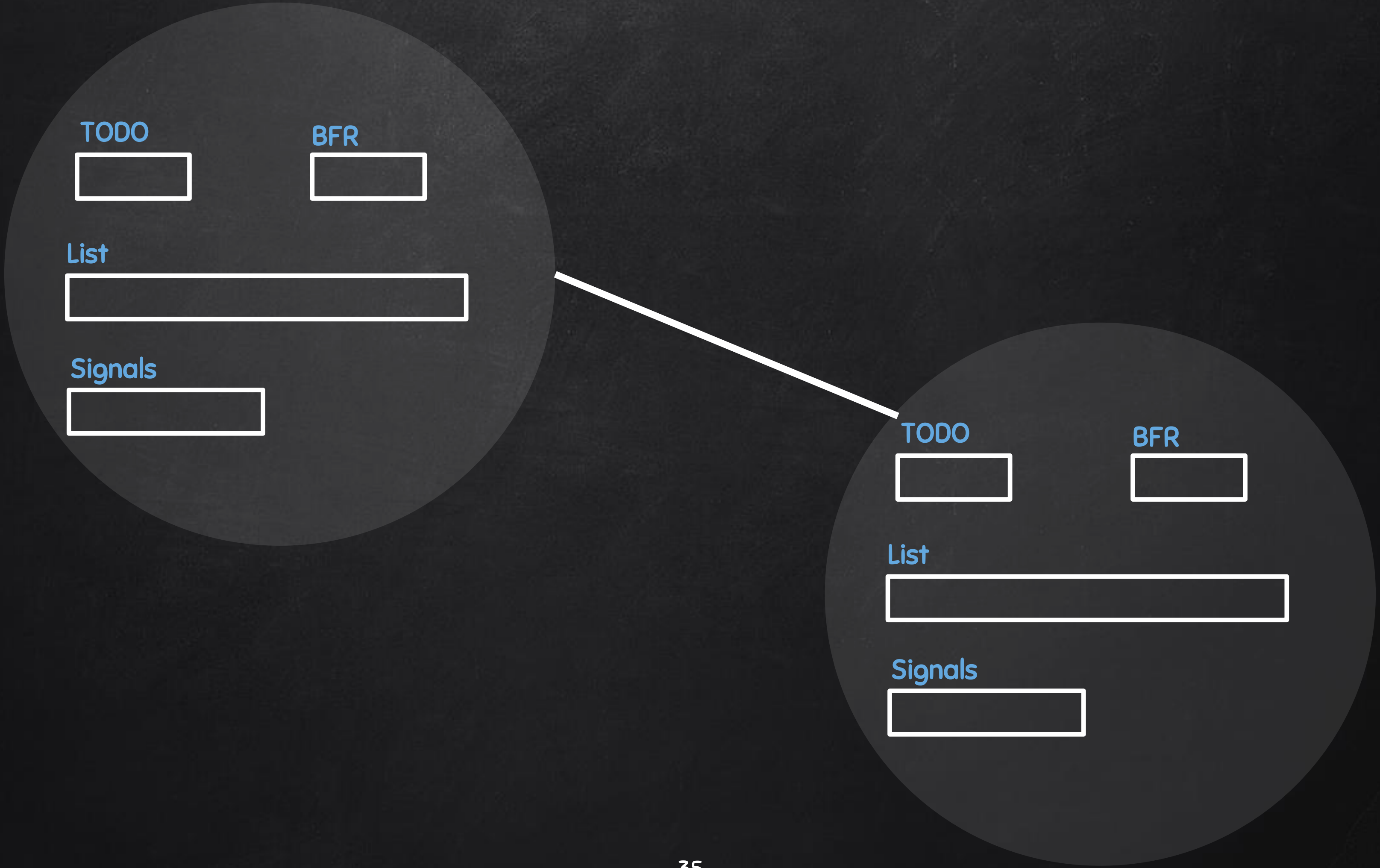
✗ Otherwise
  - Insert (k, ρ) to the list
  - Push Delete signal


ROOT
List
Signals

Empty-List Procedure

Node's list has more than 2tB entries and must be restored to tB entries

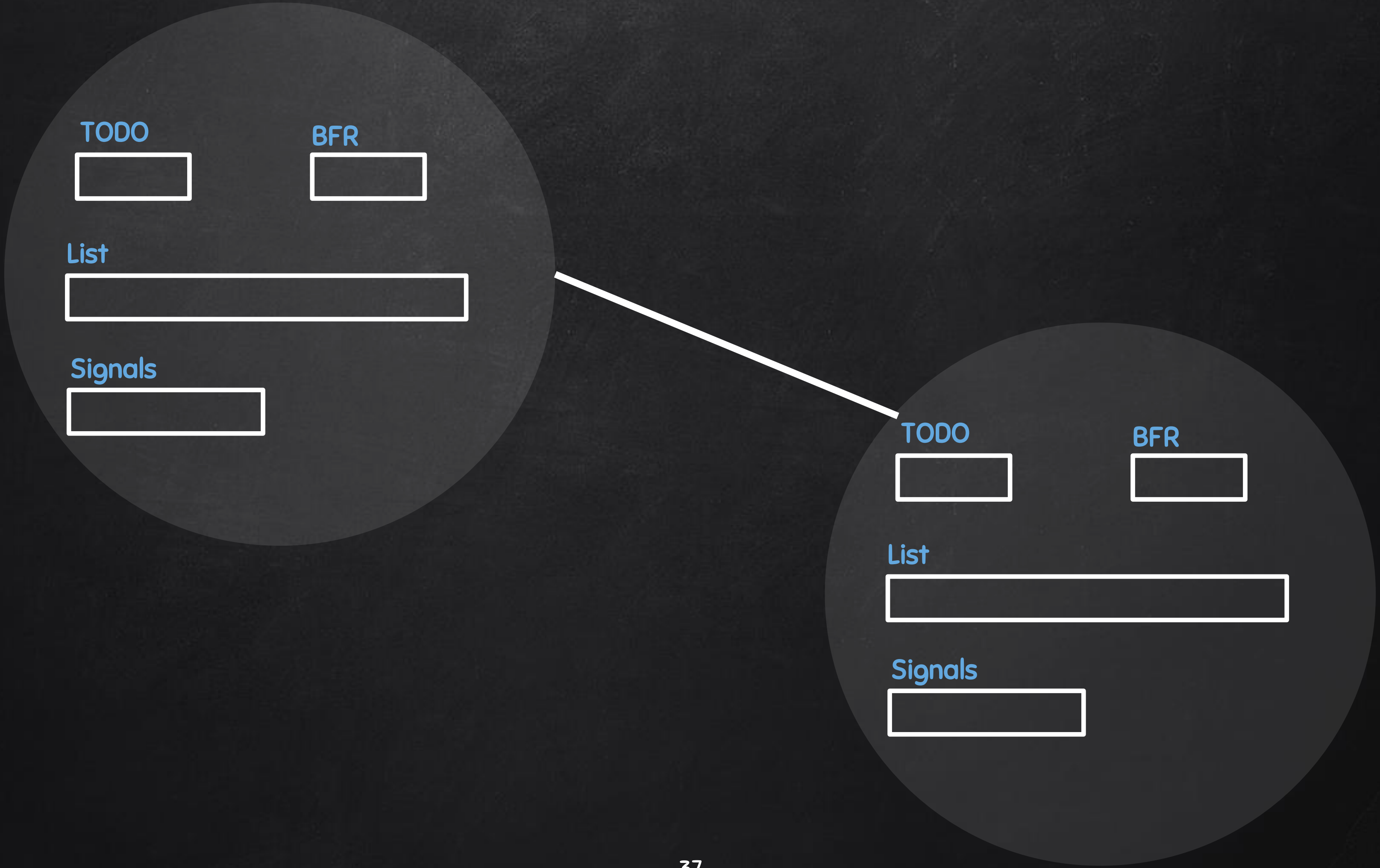
- Call *apply-TODO & push-signal* procedures
- For each element in the list
  - If child's TODO buffer contains Delete, replace it with Update
  - Otherwise, append it to the children's list
  - Update BFRs & Boundaries


TODO
BFR
List
Signals
TODO
BFR
List
Signals

Fill-Up Procedure

Node's list is empty, and must be filled with tB minimal entries

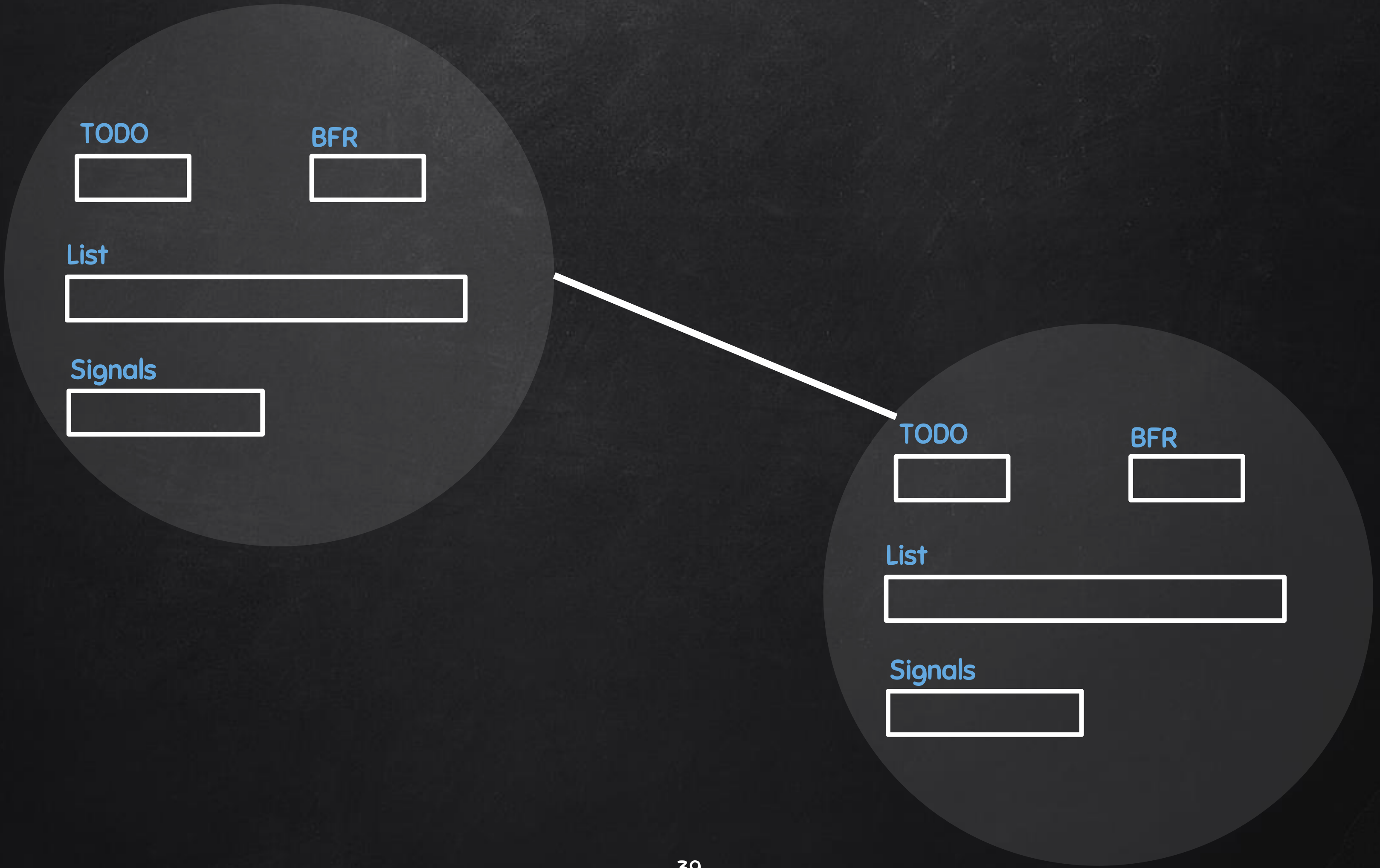
- ✗ Call *apply-TODO & push-signal* procedures
- ✗ Iterate the children's B lowest elements (considering TODO)
  - Add the element to the node's list
  - Delete from children's list & TODO buffer.
  - Update BFRs & Boundaries


TODO
BFR
List
Signals
TODO
BFR
List
Signals

Push-Signal Procedure

Pushes the signals in the signal buffer down to its children

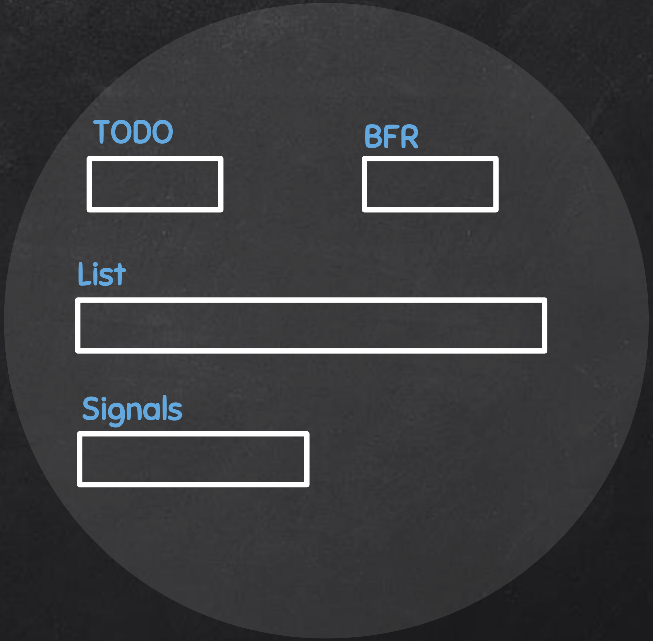
- Delete(k) signal
  - If k is “actual” in c – insert delete to TODO, else to signal buffer
- Update(k, ρ) signal
  - if ρ < Boundary(c), update TODO & push delete signal
  - Otherwise
    - If k is “actual” in c – update TODO & push delete signal
    - Else, push update signal


TODO
BFR
List
Signals
TODO
BFR
List
Signals

Apply-TODO Procedure

Node's TODO buffer is full and must be applied to node's list entries.

- For each TODO signal
  - If false-positive update signal detected, push to signal buffer
  - Else, exactly the same as Delete & Update operations !


TODO
BFR
List
Signals

I/O Complexity

Standard credits argument

- ✗ Real signal logic does not cost any I/O's
  - Anyway, we will pay $O(1/B)$ for each vertex in the path to Leaf(K)
- ✗ Push Signal – each signal goes into at most one signal buffer at each level $O(h/B)$
- ✗ Apply TODO – In expectation, each signal goes into $1 + \varepsilon h < 2$ TODO buffers $O(t/B)$

I/O Complexity

- Empty List – using the update signal credits, each entry could empty at most one entry at each level O(h/b)
- Fill Up – same has Empty List with delete signals credits
- The expected amortized cost for each Delete, Update, ExtractMin operation is O(h/B + t/B) = $O(\frac{1}{B} log_2 \frac{N}{B} / loglogN)$ I/O's

Further Work

- Cache-oblivious model
  - "Cache-Oblivious Priority Queues with Decrease-Key and Applications to Graph Algorithms" [Iacono, Jacob, Tsakalidis 20']
- Without DecreaseKey, there is a tight bound $\theta({}^{1}/_{B}\, log_{M/_B}({}^{N}/_{B}))$
- There is still a gap between the algorithm complexity and the known lower bound of $O({}^{1}/_{B}\, log(B)/\log(\log(N)))$
  - The lower bound does not include the Delete(k) operation.

Conclusion

- This data structure improves the external memory priority queue with DecreaseKey for the first time in over a decade
- Our computers designed to work in the external-memory model
- The priority queue has a direct impact on various core algorithms

THANKS!

Any questions?

Link to the paper: https://arxiv.org/pdf/1806.07598.pdf