On Sorting, Heaps, and Minimum Spanning Trees

Gonzalo Navarro, Rodrigo Paredes

R99922089 林霓苗

Outline

1. Abstract & Introduction (related works)
2. Incremental Quick Sorting algorithm
3. Quickheaps
4. Boosting the MST Construction
5. Experimental result
6. Conclusion & Future work

1. Abstract & Introduction (related works) (1/4 pages)

1. Incremental Quicksort (IQS) : incrementally gives the next smallest element of the set.
2. Quickheap (QH) : (Based on IQS) QH is a simple and efficient priority queue for main and secondary memory.
3. Use IQS to implement **Kruskal's MST**.
4. Use QHs to implement **Prim's MST**.

- Problem: *Need to obtain the smallest elements form a fixed set.*

  (1) Kruskal's MST.

  (2) Ranking by Web search engines. (which display a very small sorted subset results, if user wants more, then display the next group of result.)

1. Abstract & Introduction (related works) (2/4 pages)

- Incremental Sorting problem：
  1. Given a set A of m numbers, output the elements of A from smallest to largest.
  2. Process can be stopped after k elements have been output.

- Solved by: [complexity: **O(m+ k*logk)**]
  1. Finding the k-th smallest element of A using O(m) time (**QuickSelect algorithm**).
  2. Then collecting and sorting the elements smaller than the k-th element (**QuickSort algorithm**).

- Selection and sorting steps can be interleaved, which improves the constant terms.

1. Abstract & Introduction (related works) (3/4 pages)

- Priority Queues：
  - insert, findMin(findMax), extractMin(extractMax)
  - increaseKey, decreaseKey
  - delete...
  - ➡ Well-known priority queues are **sequence heaps** , **binomial queues**, **Fibonacci heaps** , **pairing heaps**, **skew heaps**, and **van Emde Boas queues**...
- External Memory Priority Queues：
  - Only offer basic operations : insert, findMin , extractMin.
  - ➡ Some external memory PQs are **buffer trees** ,**M/B-ary heaps**, **Array heaps** ,**R heaps**...

1. Abstract & Introduction (related works) (4/4 pages)

- IQS is **4 times** faster than the classic alternative to solve the online problem.
- QHs is up to **4 times** faster than binary heaps. (fastest priority queue implementations in practice)
- QHs performs up to **3 times** fewer I/O accesses than R-Heaps.
- QHs performs up to **5 times** fewer I/O accesses than Array-Heaps.
- *External-memory Sequence Heaps* are faster than QHs, but much more sophisticated and not cache-oblivious.

2. Incremental Quick Sorting (1/4 pages)

IQS (Set A, Index idx, Stack S)

// Precondition: $idx \leq S.\mathbf{top}()$

1. **If** $idx = S.\mathbf{top}()$ **Then** $S.\mathbf{pop}()$, **Return** $A[idx]$
2. $pidx \leftarrow \mathbf{random}[idx,\ S.\mathbf{top}() - 1]$
3. $pidx' \leftarrow \mathbf{partition}(A,\ A[pidx],\ idx,\ S.\mathbf{top}() - 1)$

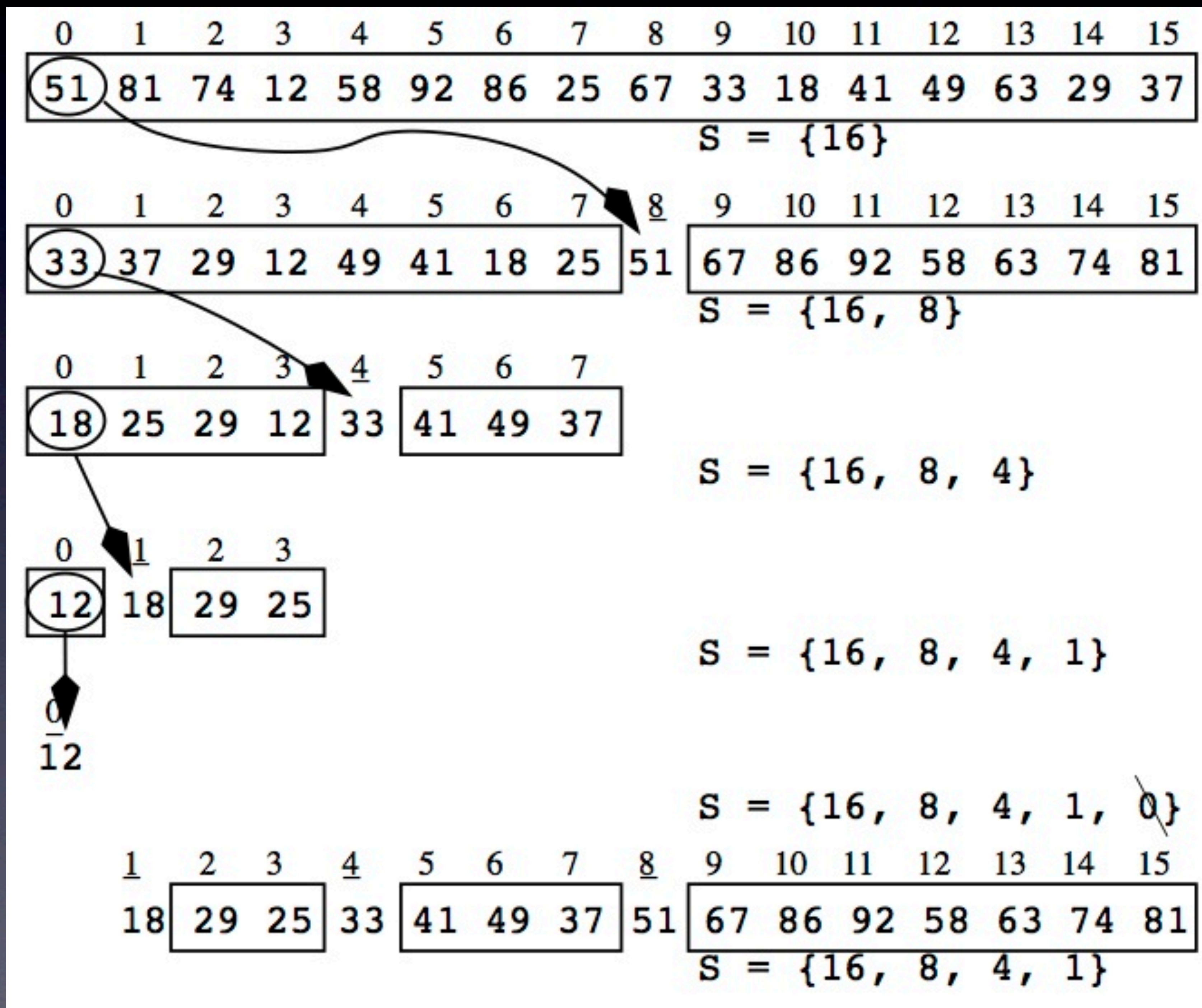
// Invariant: $A[0] \leq \ldots \leq A[idx-1] \leq A[idx, pidx'-1] \leq A[pidx']$

// $\leq A[pidx'+1, S.\mathbf{top}() - 1] \leq A[S.\mathbf{top}(), m-1]$

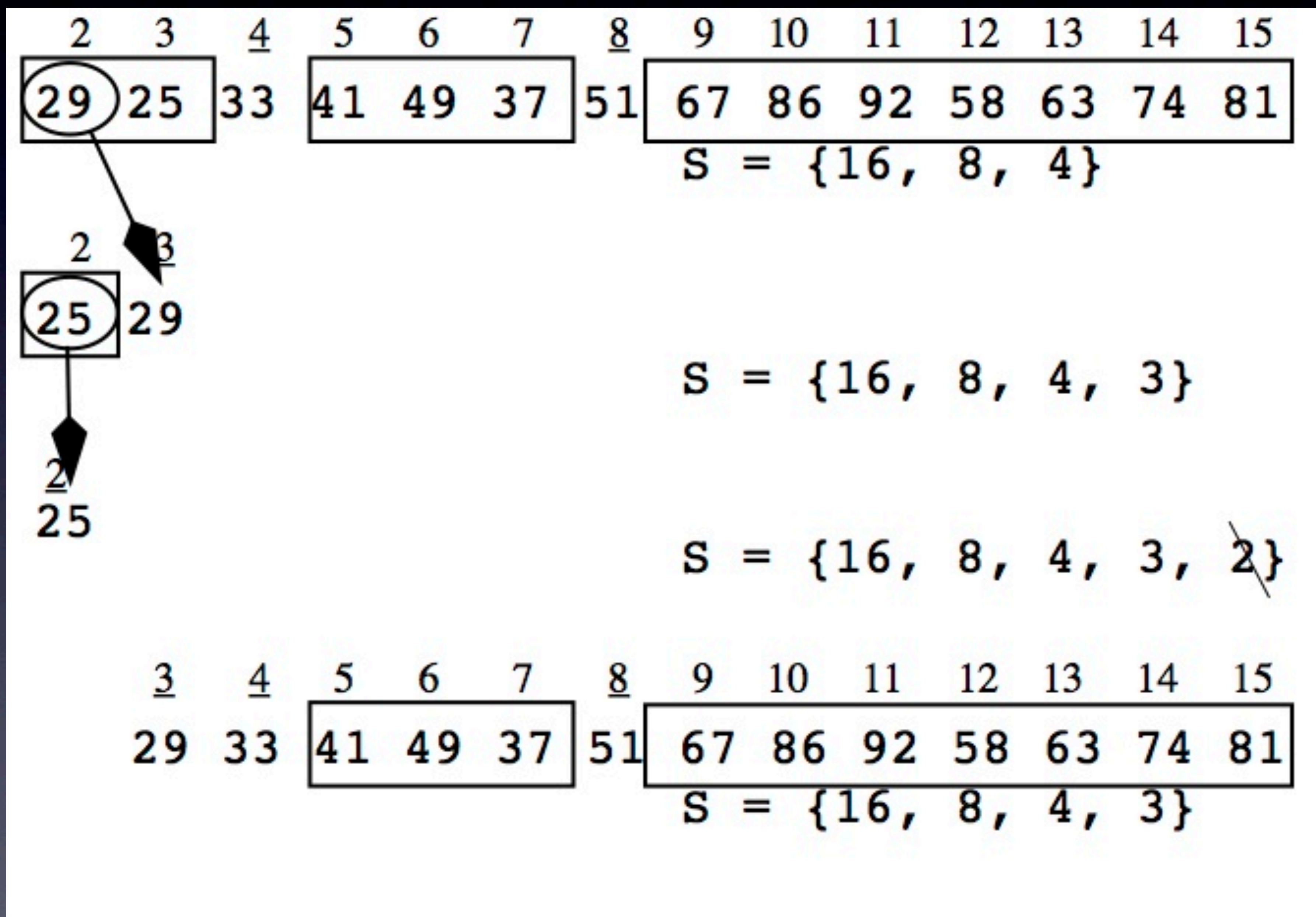
4. $S.\mathbf{push}(pidx')$
5. **Return IQS**$(A,\ idx,\ S)$

QuickSelect

2. Incremental Quick Sorting (2/4 pages)



2. Incremental Quick Sorting (3/4 pages)



2. Incremental Quick Sorting (4/4 pages)

Lemma 2.1 : (After i minima have been obtained in A[0, i–1])

(1) The pivot indices in S are decreasing bottom to top.

(2) Each pivot position p != m in S, A[p] is not smaller than any element in A[i, p – 1] and not larger than any element in A[p + 1, m – 1].

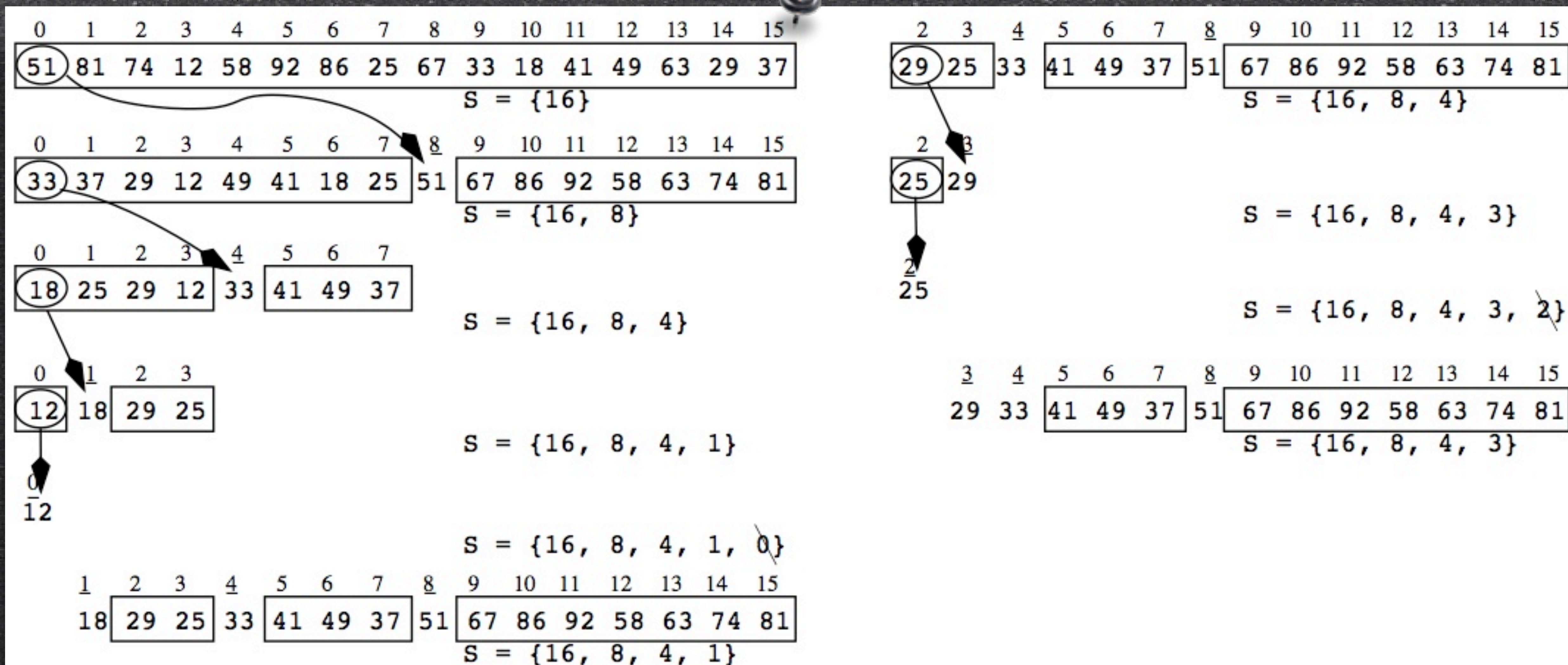
pf : Assume this is valid before pushing p, when p′ was the top of the stack.

(1) Since the pivot was chosen from A[i, p′ – 1] and left at some position i ≤ p ≤ p′ – 1 after partitioning, property (1) is guaranteed.

(2) With respect to p, the partitioning ensures that elements smaller than p are left at A[i, p – 1], while larger elements are left at A[p + 1, p′ – 1].

IQS

Incremental Quick Sort


0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15
51 81 74 12 58 92 86 25 67 33 18 41 49 63 29 37
S = {16}
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15
33 37 29 12 49 41 18 25 51 67 86 92 58 63 74 81
S = {16, 8}
0 1 2 3 4 5 6 7
18 25 29 12 33 41 49 37
S = {16, 8, 4}
0 1 2 3
12 18 29 25
S = {16, 8, 4, 1}
0
12
S = {16, 8, 4, 1, 0}
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15
18 29 25 33 41 49 37 51 67 86 92 58 63 74 81
S = {16, 8, 4, 1}
2 3 4 5 6 7 8 9 10 11 12 13 14 15
29 25 33 41 49 37 51 67 86 92 58 63 74 81
S = {16, 8, 4}
2 3
25 29
S = {16, 8, 4, 3}
2
25
S = {16, 8, 4, 3, 2}
3 4 5 6 7 8 9 10 11 12 13 14 15
29 33 41 49 37 51 67 86 92 58 63 74 81
S = {16, 8, 4, 3}

```
IQS (Set A, Index idx, Stack S)
    // Precondition: idx ≤ S.top()
    If idx = S.top() Then S.pop(), Return A[idx]
    pidx ← random[idx, S.top()−1]
    pidx′ ← partition(A, A[pidx], idx, S.top()−1)
    // Invariant: A[0] ≤ ... ≤ A[idx − 1] ≤ A[idx, pidx′ − 1] ≤ A[pidx′]
    // ≤ A[pidx′ + 1, S.top()−1] ≤ A[S.top(), m − 1]
    S.push(pidx′)
    Return IQS(A, idx, S)
```

Given a set A of m numbers IQS finds the k smallest elements, for any unknown value $k \leq m$, in $O(m + k \log k)$ expected time.

In IQS, the final pivot position p after the partitioning of $A[0, m-1]$ distributes uniformly in $[0, m-1]$.

Let $T(m, k)$ be the expected number of key comparisons needed to obtain the k smallest elements of $A[0, m-1]$. After the $m-1$ comparisons used in the partitioning, there are three cases depending on p.

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
	18	29	25	33	41	49	37	51	67	86	92	58	63	74	81	∞

S = {16, 8, 4, 1}

$k \le p$ $\qquad T(p,k)$

$k = p + 1$ $\qquad \mathrm{T}(p,p)$

$k > p + 1$ $\qquad \mathrm{T}(p,p)+\mathrm{T}(m\text{-}1\text{-}p,k\text{-}p\text{-}1)$

$$m-1+\frac{1}{m}\left(\sum_{p=k}^{m-1} T(p,k) + T(k-1,k-1) + \sum_{p=0}^{k-2}\left(T(p,p)+T(m-1-p,k-p-1)\right)\right)$$

$O(m + k \log k)$

Quickheaps

Heap structure in the sense that objects in the array are semi-ordered.

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
	18	29	25	33	41	49	37	51	67	86	92	58	63	74	81	∞

S = {16, 8, 4, 1}

Data Structure for Quickheaps

Assume we know beforehand the value of capacity

- (Circular)Array A- to store the elements.
- Stack S- to store the position of pivots.
- Integer idx- to indicate the first cell of the quickheap.
- Integer capacity- to indicate the size of heap.

Creation of Quickheaps

- With no elements
  - $S = \{0\}$, $idx = 0$
- From an array B
  - copy B to A
  - $S = \{|A|\}$, $idx = 0$

Finding the Minimum

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
	18	29	25	33	41	49	37	51	67	86	92	58	63	74	81	∞

S = {16, 8, 4, 1}

IQS(A,idx,S)

return $A[idx]$

Extracting the Minimum

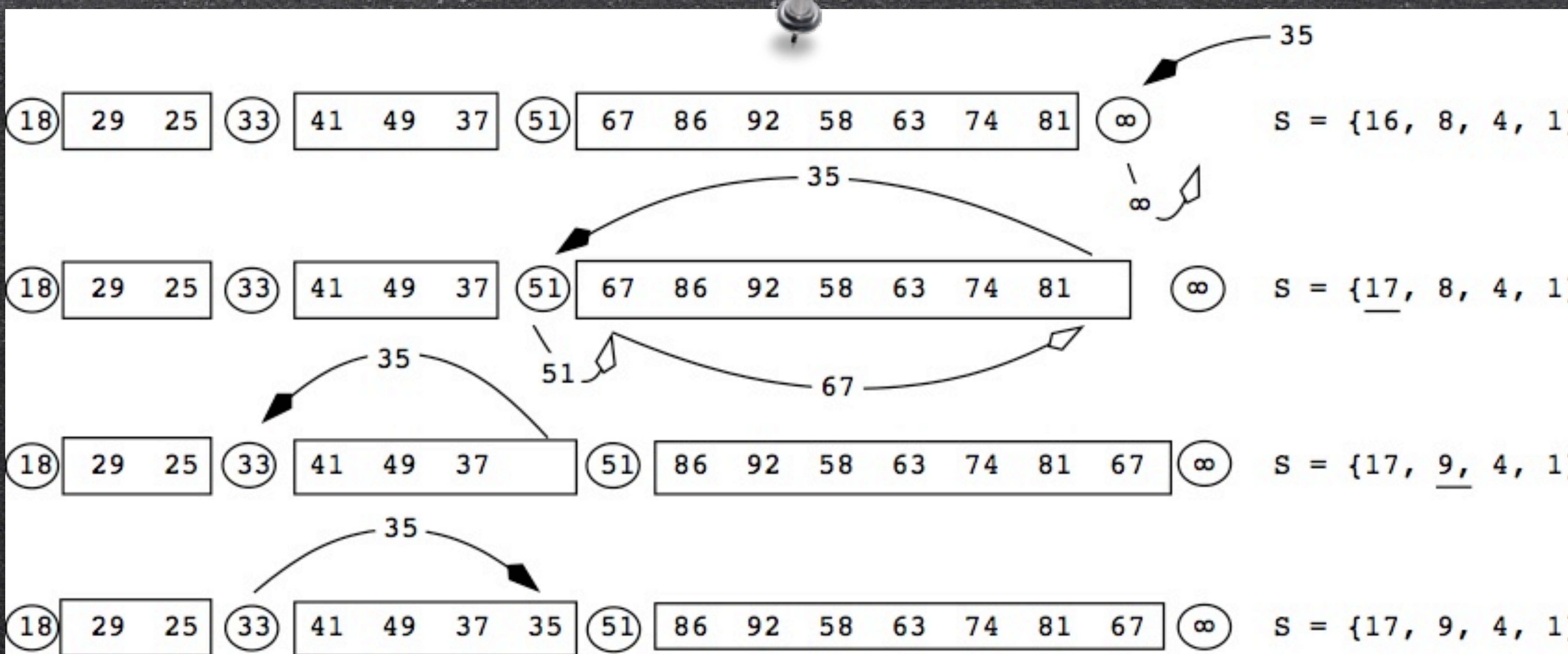
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
	18	29	25	33	41	49	37	51	67	86	92	58	63	74	81	∞

S = {16, 8, 4, 1}

idx++; *S*.pop()

return *A*[*idx*-1]

Inserting Elements



Deleting Arbitrary Elements

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
	18	29	25	33	41	49	37	51	67	86	92	58	63	74	81	∞

S = {16, 8, 4, 1}

- Non-pivot
  - Find $S[pidx] \geq pos$
  - swap($A[S[pidx]-1]$,$A[pos]$)
  - swap($A[S[pidx]-1]$,$A[S[pidx]]$)
  - Until reach the fictitious pivot.
- Pivot
  - drop the pivot
  - Join two chunks
  - delete as non-pivot
  -

Decreasing a Key

Given a position *pos* of some element in the quickheap and a value $\delta \geq 0$, we change the priority of the element $A[pos]$ to $heap[pos] - \delta$.

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	
	18	29	25	33	41	49	37	51	67	86	92	58	63	74	81	∞	S = {16, 8, 4, 1}

$newValue = A[pos] - \delta$
Find $S[pidx] \geq pos$
if $|S| == pidx + 1$ then $A[pos] = newValue$
else if $newValue \geq A[S[pidx+1]]$ then $A[pos] = newValue$
else swap($A[S[pidx+1]+1]$, A[pos]) and do as insertion.

Increasing a Key

Given a position *pos* of some element in the quickheap, and a value $\delta \geq 0$, this operation changes the value of the element $A[pos]$ to $A[pos] + \delta$.

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
	18	29	25	33	41	49	37	51	67	86	92	58	63	74	81	∞

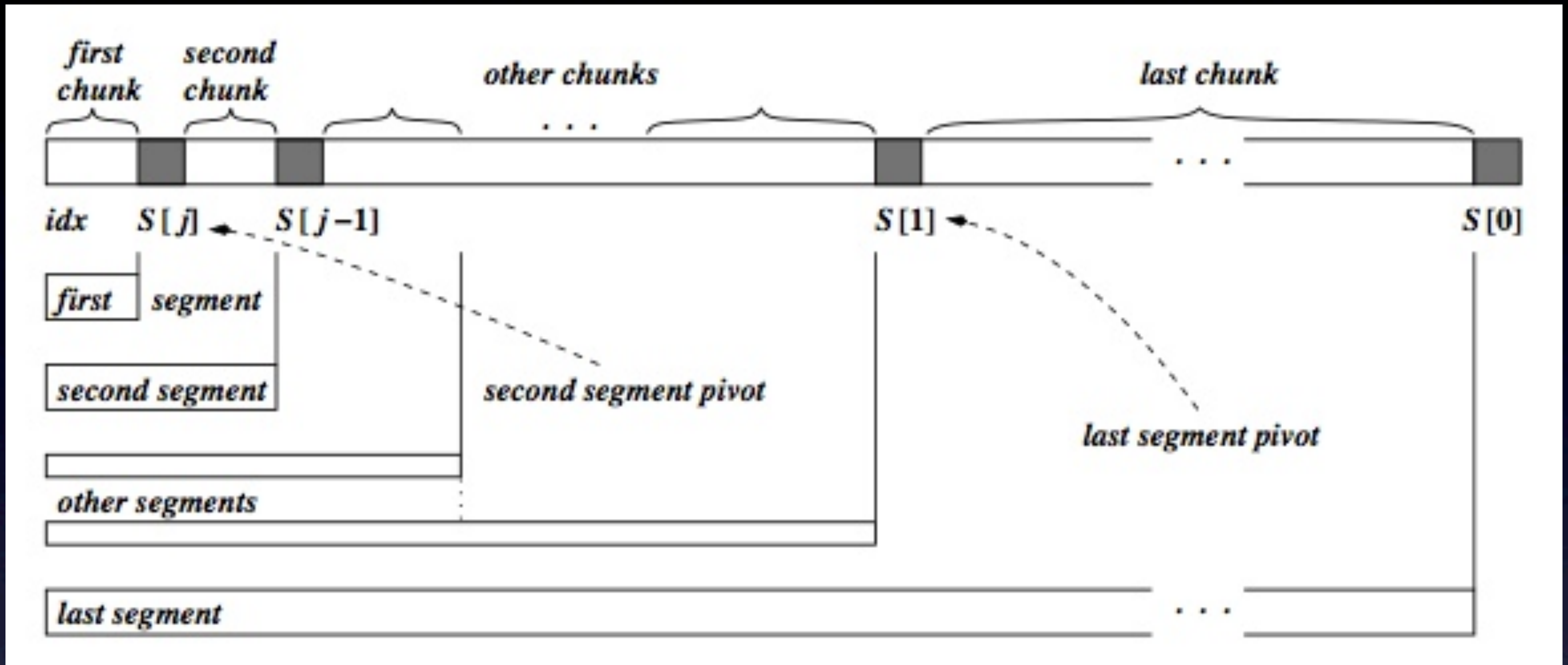
S = {16, 8, 4, 1}

4 Analysis of Quickheaps
5 Quickheaps in External Memory

Prensent by R99922121 Li-de Yang

- 4 Analysis of Quickheaps
  - Prove that quickheap operations cost $O(\log m)$ expected amortized time, where m is the maximum size of the quickheap.

- 4.1 The Quickheap's Exponential-Decrease Property



  - array segments: heap[idx, S[pidx] − 1], thus segments overlap.
  - array chunks: heap[S[pidx]+1, S[pidx−1]−1] or heap[idx,S.top()−1].

- pivot of a segment:
  Rightmost pivot within such segment.
  Thus, the pivot of the last segment is S[1],
  whereas the first segment is the only one not
  having a pivot.
- median of a n-element set:
  $\frac{n+1}{2}$-th largest element, n is odd
  the average of the $\frac{n}{2}$-th and $(\frac{n}{2}+1)$-th largest ones, n is even

- Definition 4.1
  Quickheap's exponential-decrease property: for all the segments P(pivot is large) ≤ 0.5
- $P_{i,j,n}$, $1 \leq i \leq n$, $j \geq 0$, $n > 0$
  the probability that the i-th element of the segment, of size n, is the pivot of the segment after the j-th operation
- Prove that $P_{i,j,n} \leq P_{i-1,j,n}$, for all j, n and $2 \leq i \leq n$

- Lemma 4.1
  For each segment, the property $P_{i,j,n} \leq P_{i-1,j,n}$ for $i \geq 2$ is preserved after inserting a new element x at a position uniformly chosen in [1, n].

$$\mathbb{P}_{i,j,n} = \mathbb{P}_{i-1,j-1,n-1}\frac{i-1}{n} + \mathbb{P}_{i,j-1,n-1}\frac{n-i}{n}$$

$$\mathbb{P}_{i-1,j-1,n-1}\frac{i-1}{n} + \mathbb{P}_{i,j-1,n-1}\frac{n-i}{n} \leq \mathbb{P}_{i-1,j-1,n-1}\frac{i-2}{n} + \mathbb{P}_{i-1,j-1,n-1}\frac{n+1-i}{n}$$

- Lemma 4.2
For each segment, the property $P_{i,j,n} \leq P_{i-1,j,n}$ for $i \geq 2$ is preserved after deleting an element at a position chosen uniformly from $[1, n + 1]$.

$$\mathbb{P}_{i,j,n} = \mathbb{P}_{i+1,j-1,n+1}\frac{i}{n+1} + \mathbb{P}_{i,j-1,n+1}\frac{n+1-i}{n+1}$$

$$\mathbb{P}_{i,j-1,n+1}\frac{i}{n+1} + \mathbb{P}_{i,j-1,n+1}\frac{n+1-i}{n+1} \leq \mathbb{P}_{i,j-1,n+1}\frac{i-1}{n+1} + \mathbb{P}_{i-1,j-1,n+1}\frac{n+2-i}{n+1}$$

- pivoting:
  partition the first segment with a pivot and pushes it into stack S.
- takeMin:
  increment idx, pops stack S and returns element heap[idx − 1].

- extractMin:
  - execute pivoting as many times as we need to push idx in stack S
  - takeMin
- findMin:
  - execute pivoting as many times as we need to push idx in stack S
  - return element heap[idx]

- Lemma 4.3
  For each segment, the property $P_{i,j,n} \leq P_{i-1,j,n}$ for $i \geq 2$ is preserved after taking the minimum element of the quickheap.

$$\mathbb{P}_{i,j,n} = \mathbb{P}_{i+1,j-1,n+1} \frac{n+1}{n}$$

- Theorem 4.1
  Quickheap's exponential-decrease property: Given a segment heap[idx, S[pidx]−1], the probability that its pivot is large is smaller than or equal to 0.5 , that is, P(pivot is large) ≤ 0.5 .

- Lemma 4.4
  The expected value of the height H of stack S is O(log m).

$$\mathcal{H} = T(m) = 1 + \tfrac{1}{2}T(m-1) + \tfrac{1}{2}T\left(\left\lfloor \tfrac{m}{2} \right\rfloor\right), T(1) = 1$$

$$T(m) \leq 2 + T\left(\tfrac{m}{2}\right) \leq \ldots \leq 2j + T\left(\tfrac{m}{2^j}\right)$$

- Lemma 4.5
  The expected value of the sum of the sizes of array segments is Θ(m).

$$T(m) = m + \tfrac{1}{2}T(m-1) + \tfrac{1}{2}T\left(\left\lfloor \tfrac{m}{2} \right\rfloor\right),\ T(1) = 0$$

$$T(m) \leq 2m + T\left(\tfrac{m}{2}\right) \leq \ldots \leq 2m + m + \tfrac{m}{2} + \tfrac{m}{2^2} + \ldots + \tfrac{m}{2^{j-2}} + T\left(\tfrac{m}{2^j}\right)$$

- 4.2 The Potential Debt Method
  - The potential function represents a total cost that has not yet been paid.
  - c_i: actual cost of the i-th operation
  - D_i: data structure that results from applying the i-th operation to D_{i-1}
  - Φ: potential debt function maps each data structure D_i to a real number $\Phi(D_i)$, which is the potential debt associated with data structure D_i up to then

$$\widetilde{c_i} = c_i - \Phi(D_i) + \Phi(D_{i-1})$$

$$\widehat{c_i} = \widetilde{c_i} + \frac{\Phi(D_N) - \Phi(D_0)}{N}$$

- 4.3 Expected-case Amortized Analysis of Quickheaps

$$\Phi(qh) = 2 \cdot \sum_{i=0}^{\mathcal{H}} (S[i] - idx) = \Theta(m) \text{ expected, by Lemma 4.5}$$

$$\frac{\Phi(qh_N) - \Phi(qh_0)}{N} \text{ is } O(1)$$

$$\widehat{c_i} = c_i - \Phi(qh_i) + \Phi(qh_{i-1})$$

- Expected (individual) cost
  - Operation insert
    $= 1 + (1 - P_1)(1 + (1 - P_2)(1 + (1 - P_3)(1 + \ldots)))$
    $= O(1)$
  - Operation delete
    $= 1+(1-P_1)(1+(1-P_2)(1+(1-P_3)(1+\ldots)))$
    $= O(1)$
  - Creation of a quickheap = $\Theta(m)$

- Expected (individual) cost
  - Operation extractMin
    = 2H + 2
    = O(log m)
  - Operation findMin = O(1)
  - Operation increaseKey
    = H + 2H + 2
    = O(log m)
  - Operation decreaseKey:
    In practice, this operation performs reasonably well.

- Theorem 4.2
  Quickheap’s complexity:
  The expected amortized cost of any sequence of m operations insert, delete, findMin, extractMin and increaseKey over an initially empty quickheap is O(log m) per operation.

- 5 Quickheaps in External Memory
  - 5.1 Adapting Quickheap Operations to External Memory
    - Quickheaps exhibit high locality of reference:
      - Stack S is small and accessed sequentially.
      - Each pivot in S points to a position in the array heap.
        Array heap is only modified at those positions, and the positions themselves increase at most by one at each insertion.
      - IQS sequentially accesses the elements of the first chunk.

- 5.2 Analysis of External Memory Quickheaps
  - Theorem 5.1
    External quickheap's complexity:
    $M = \Omega(B \log m)$.
    The expected amortized I/O cost of any sequence of m operations insert, findMin, and extractMin over an initially empty quickheap is $O((1/B)\log(m/M))$ per operation.

6. Boosting the MST Construction

On Sorting, Heaps, and Minimum Spanning Trees

Gonzalo Navarro, Rodrigo Paredes

Miao-En Chien | R00944028

IQS implements **Kruskal's** MST algorithm

QH implements **Prim's** MST algorithm

m' = *1/2 n lon n + O(n)*
edges

$\longrightarrow$ *Kruskal variant*

O(*m+n log^2 n*)

on random graphs

Use QH to find the **node u*** with minimum connecting cost to the

Update the value of each **u*'s neighbor** in QH

Augment the QH structure with a **dictionary managing** the position of

inser	→	$O(1)$		$O(\mathbf{n})$
			$\times\, n$	
extract	→	$O(\log n)$		$O(\mathbf{n} \log n)$
decrease	→	?	$\times\, m$	?

inser	→	$O(1)$		$O(\mathbf{n})$
			$\times\, n$	
extract	→	$O(\log n)$		$O(\mathbf{n} \log n)$
decrease	→	$O(\log n)$	$\times\, m$	$O(\mathbf{m} \log n)$

Assuming that each call to decreaseKey has cost ***O(logn)***, this accounts for a total ***O(n log n log m/n)*** *expected time*.

$$O(m + n \log n \log m/n)$$

Expected amortized time for their **Prim variant** on graphs with **random weights**.

7. Experimental Results

On Sorting, Heaps, and Minimum Spanning Trees

Gonzalo Navarro, Rodrigo Paredes

Miao-En Chien | R00944028

Compare **IQS** with other alternatives

The empirical behavior of **QHs**

Compare **IQS** with other alternatives

The empirical behavior of **QHs**

Evaluating

1. **Classical Quickselect + Quicksort solution**: QSS
   Use random permutations of non-repeated numbers uniformly distributed.

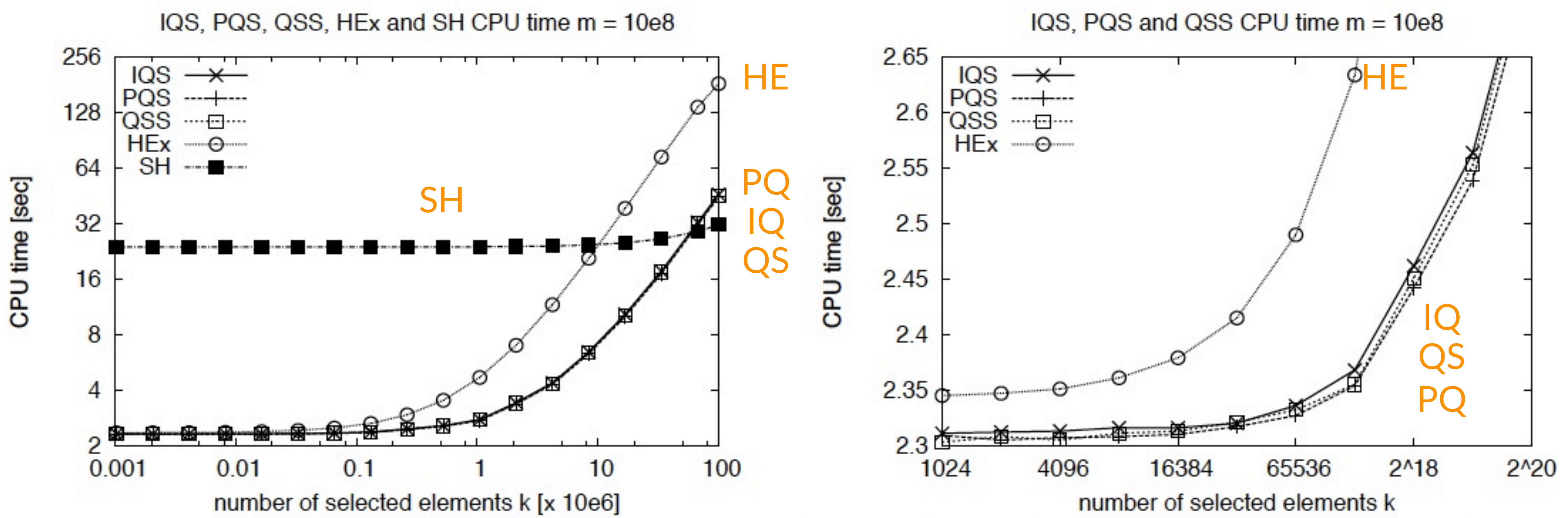
2. **Partial Quicksort algorithm**: PQS
   Select the *k* first elements, and the selection is in **one shot** for PQS and QSS.

3. **Incremental Quicksort**: IQS
   Verify that IQS is in practice a competitive algorithm for the ***Partial Sorting*** problem of finding the smallest elements in ascending order.

4. **Classical heaps**: HEx
   Implemented using the bottom-up deletion algorithm.

5. **Sequence heaps**: SH
   Select the *k* first elements, and the selection is **incremental** for IQS, HEx, and SH.

CPU time + Key comparisons



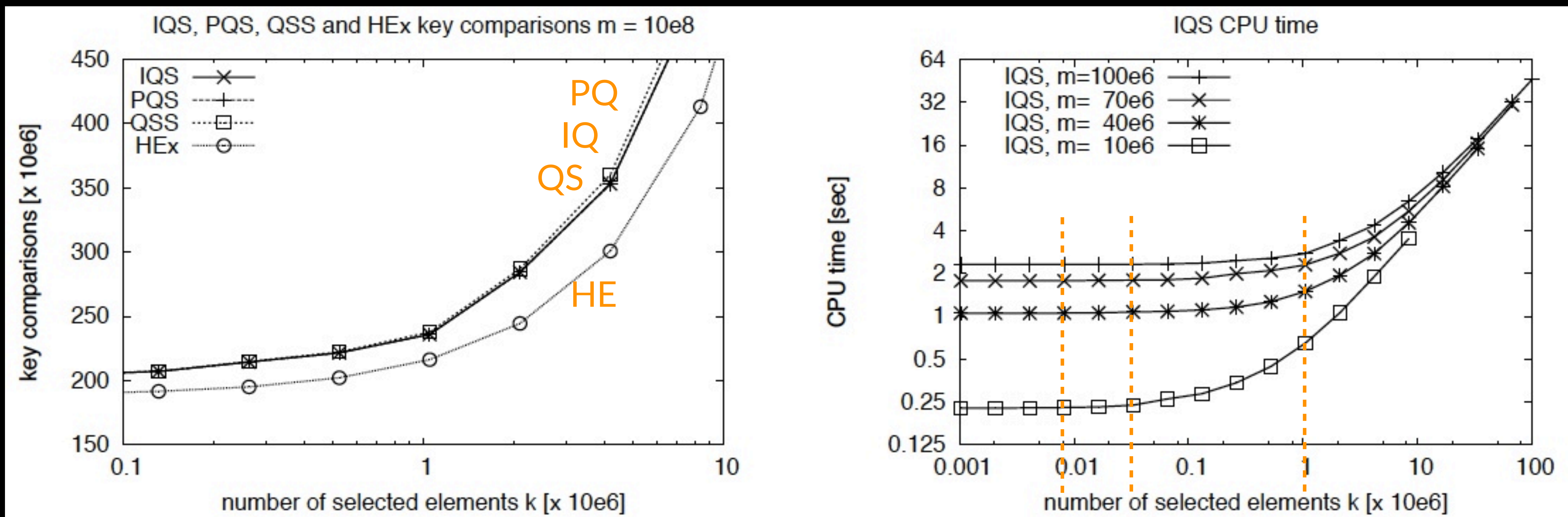
(a) CPU time for the five algorithms.

(b) Detail of CPU time for **IQS**, **PQS**, **QSS** and **HEx**.

extractMin

insert

$$O(k + m\log m)$$



(c) Detail of key comparisons for the four algorithms.

(d) **IQS** CPU time as a function of k and m.

It is preferable to pay a lower insertion and a higher extraction cost (just like **IQS**) than to perform most of the work in the

Weighted least square fittings

	CPU time	Error	Key comparisons	Error
PQS	$25.8m + 16.9k \log_2 k$	6.77%	$2.14m + 1.23k \log_2 k$	5.54%
IQS	$25.8m + 17.4k \log_2 k$	6.82%	$2.14m + 1.23k \log_2 k$	5.54%
QSS	$25.8m + 17.2k \log_2 k$	6.81%	$2.14m + 1.29k \log_2 k$	5.53%
HEx	$23.9m + 67.9k \log_2 m$	6.11%	$1.90m + 0.97k \log_2 m$	1.20%
SH	$9.17m \log_2 m + 66.2k$	2.20%	—	—

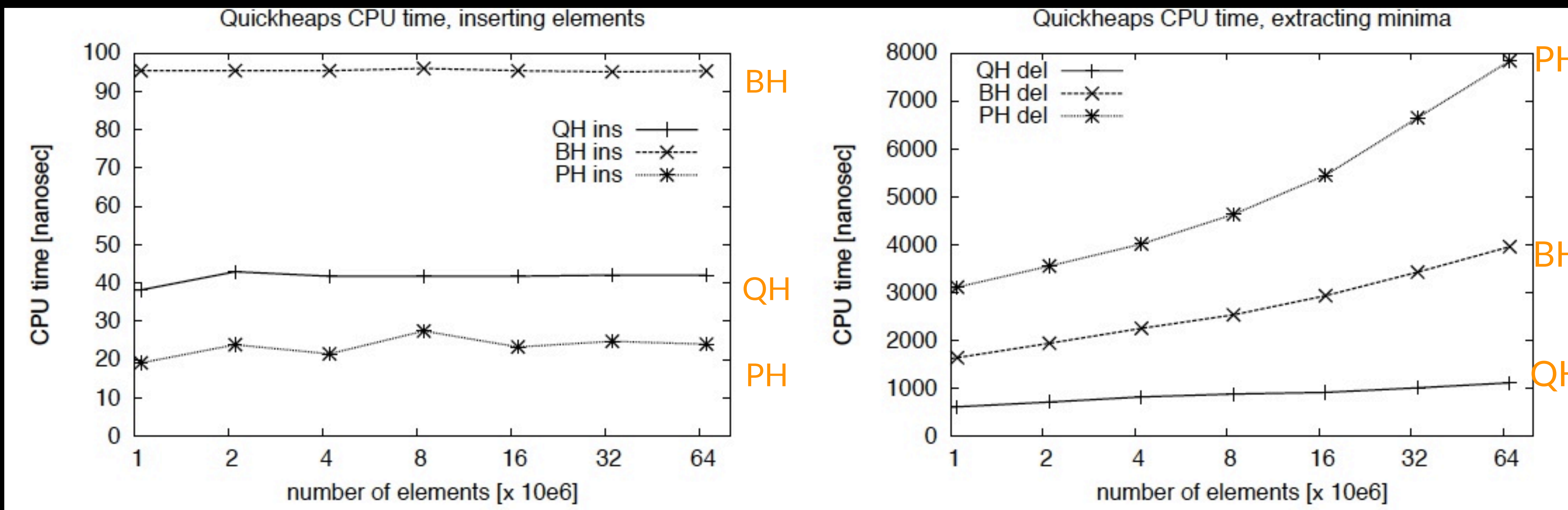
2.9 1.3 4.2

1. **Quickheaps**: QHs
   Compare the empirical performance of quickheaps.

2. **Binary heaps**: BH
   The canonical implementation of PQs, efficient and easy to program.
   The most efficient PQ implementations in practice.

3. **Paring heaps**: PH
   Implement efficiently key update operations, and also the most efficient PQ implementations.
   *Includes operations **insert**, **extractMin**, and **decreaseKey**.*

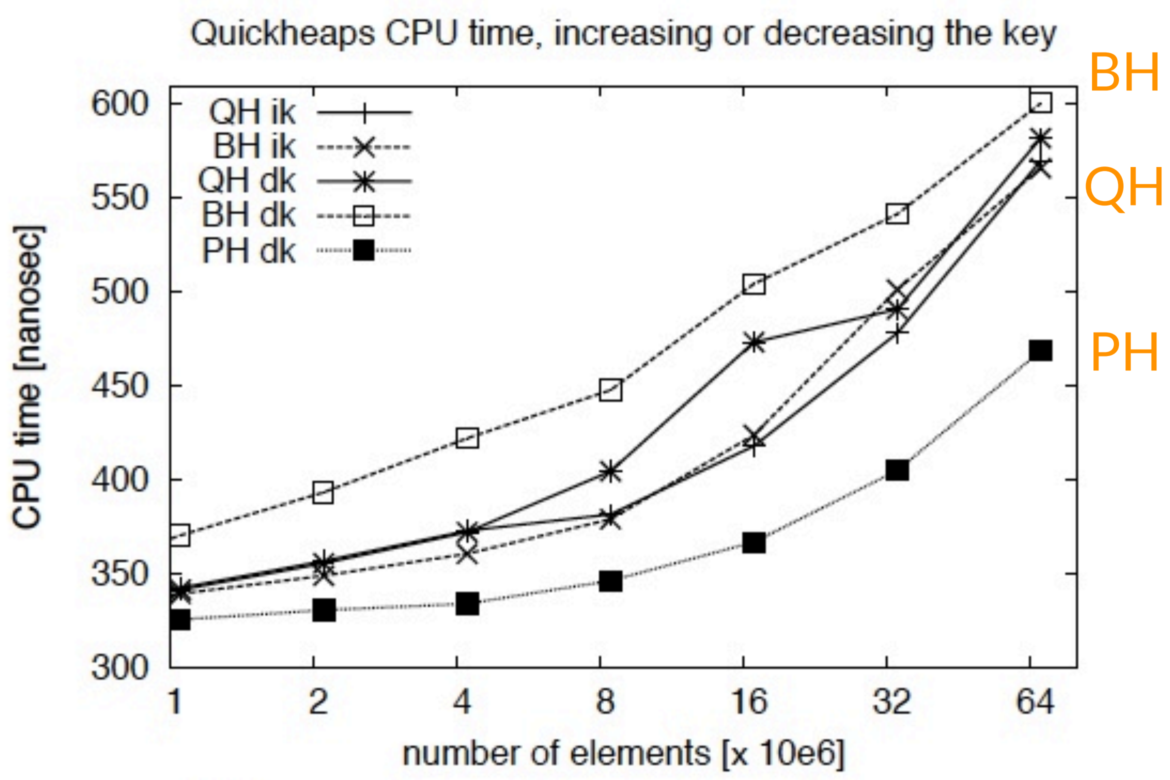
insert $\rightarrow O(n)$ extractMin $\rightarrow O(n \log n)$



(a) Inserting elements. (b) Extracting minima.

increaseKey
decreaseKey $\rightarrow O(m \log n)$

BH < QH < PH

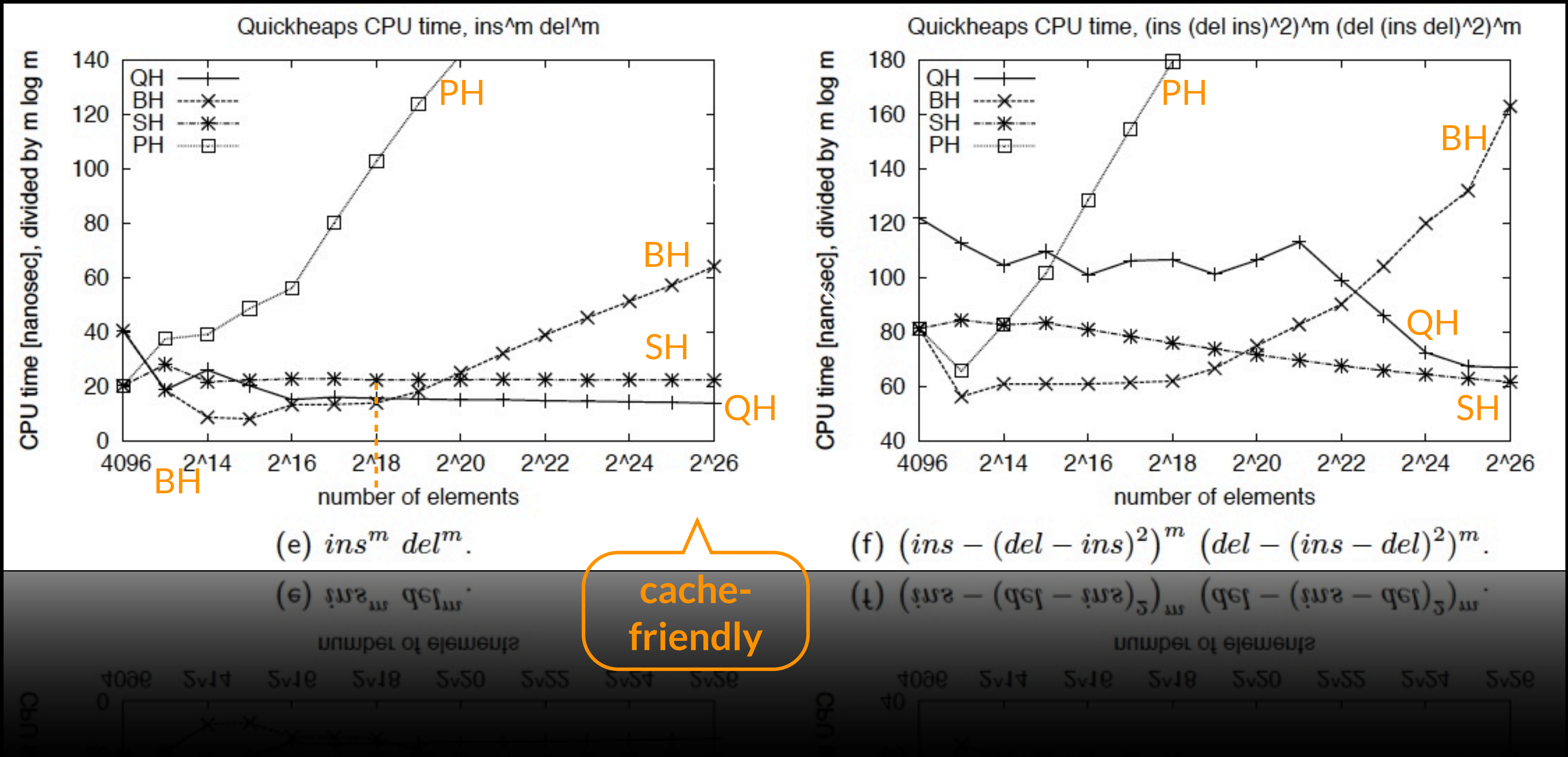


(c) Increasing and decreasing the key.

(d) Least square fittings for priority queue operations.

	Fitting	Error
QH_{ins}	42	1.28%
BH_{ins}	99	3.97%
☆ PH_{ins}	26	10.35%
☆ QH_{del}	$35 \log_2 m$	9.86%
BH_{del}	$105 \log_2 m$	15.13%
PH_{del}	$\boxed{201 \log_2 m}$	15.99%
QH_{ik}	$18 \log_2 m$	8.06%
BH_{ik}	$18 \log_2 m$	9.45%
QH_{dk}	$18 \log_2 m$	8.88%
BH_{dk}	$20 \log_2 m$	6.75%
PH_{dk}	$16 \log_2 m$	5.39%

Quickheaps perform well under arbitrarily long sequences of insertions and minimum extractions



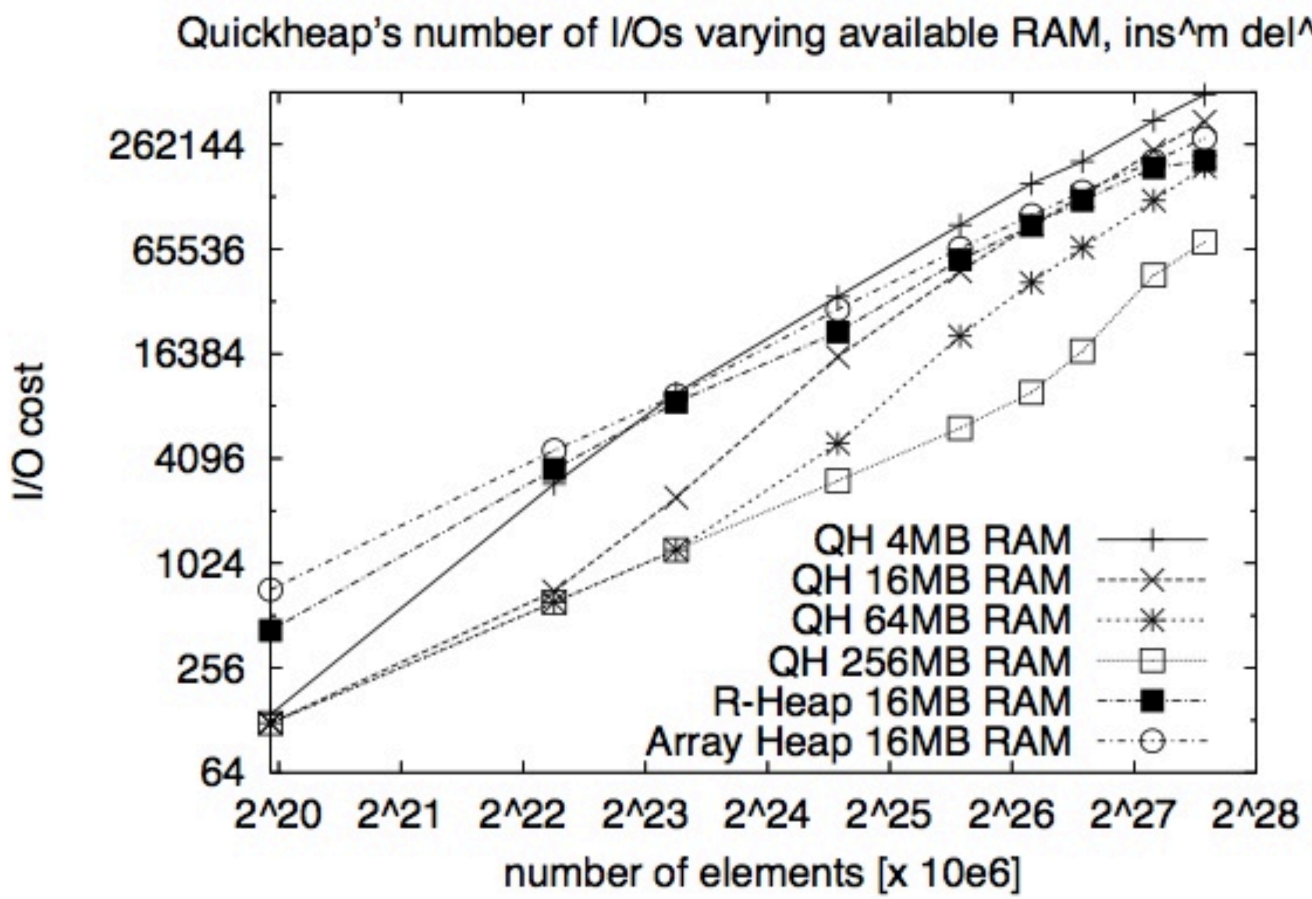
(e) $ins^m\ del^m$.

(f) $(ins - (del - ins)^2)^m\ (del - (ins - del)^2)^m$.

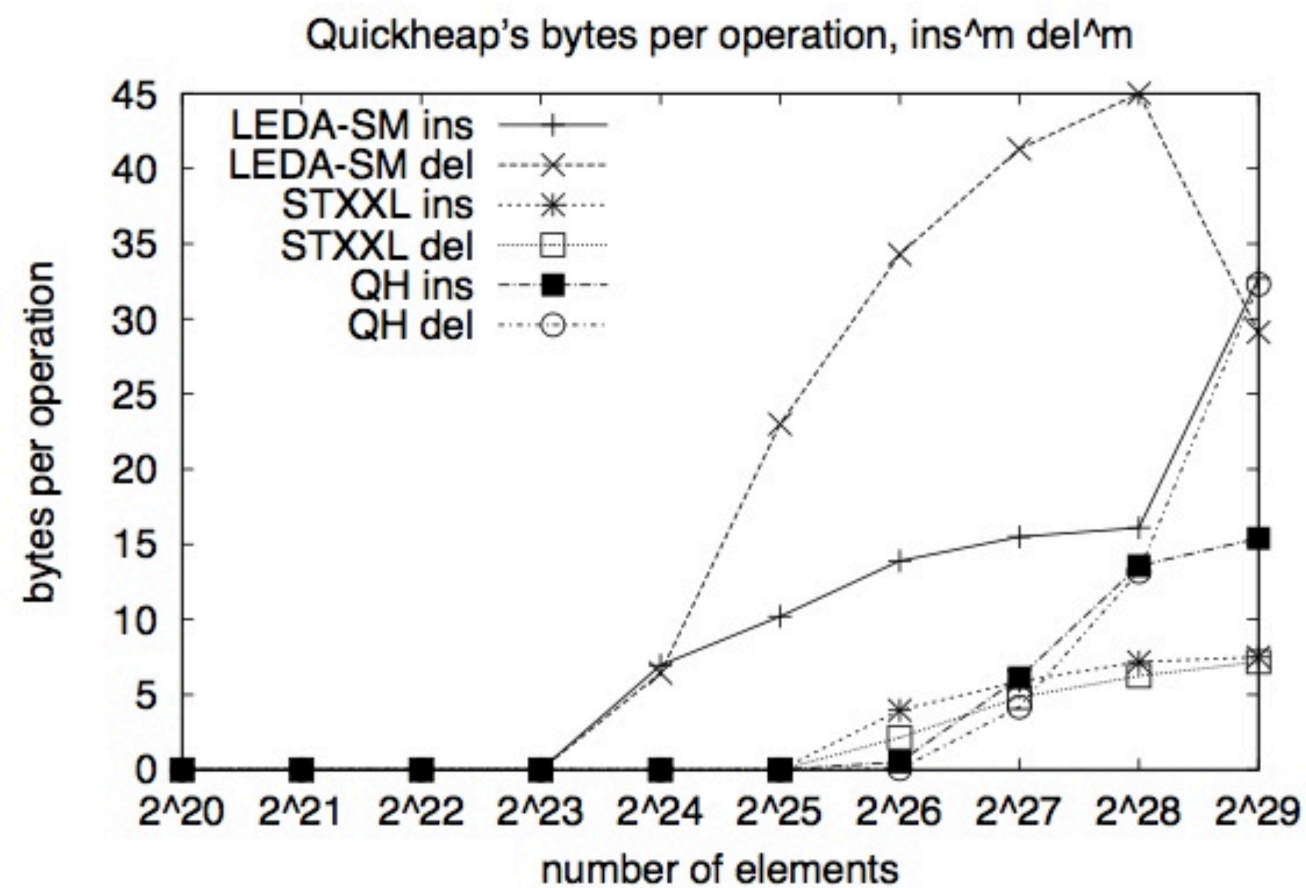
Evaluating External Memory Quickheaps

- 驗證在external memory裡QH的效能
- 複製了Brengel的setup
- 和R-heaps Array-Heaps比較

Results



(a) I/O complexity ins^m del^m.



(b) Bytes per operation ins^m del^m.

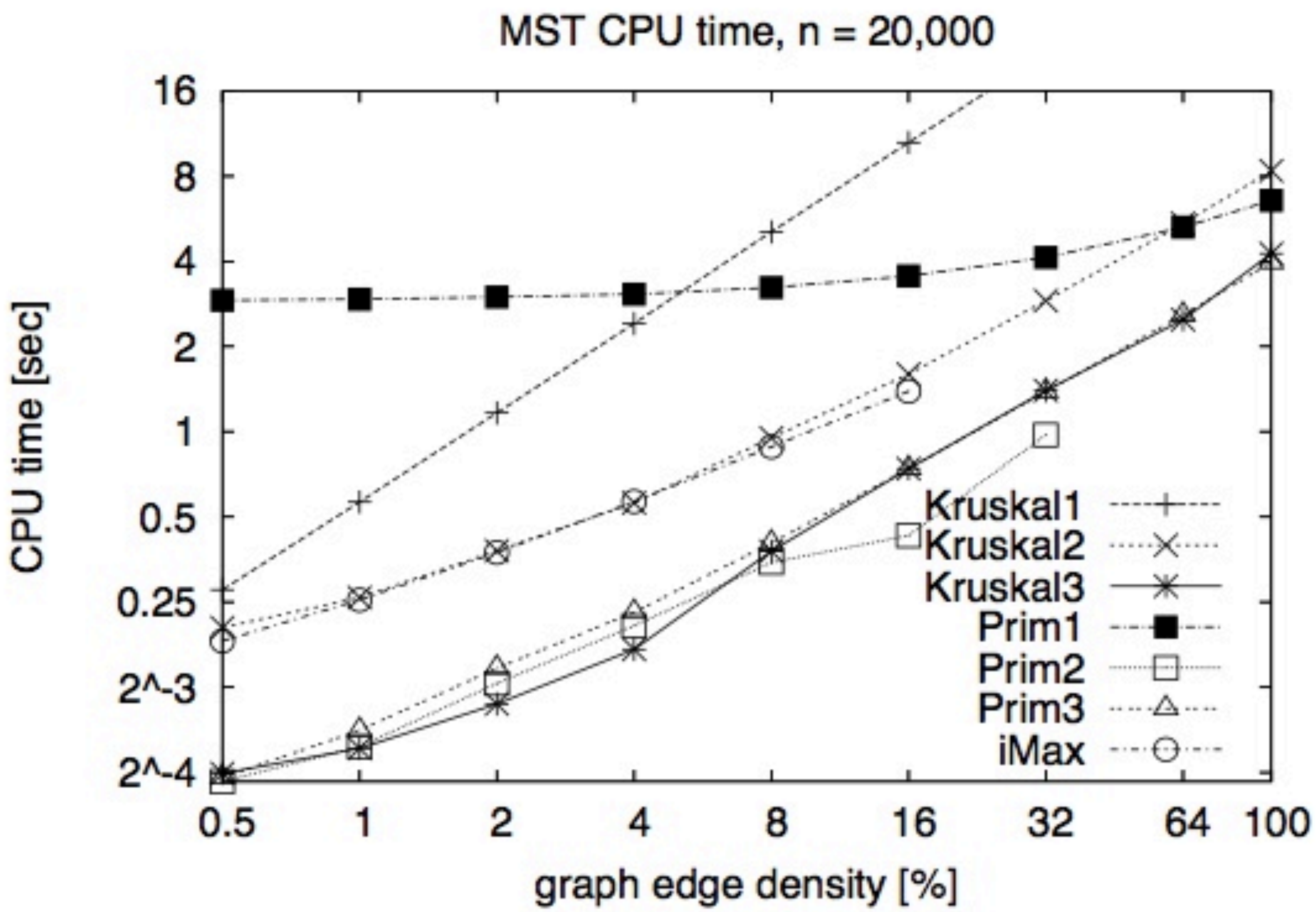
Evaluating the MST Construction

- 他們用MST Construction，去比較各個方法的performance
- 目標不是去做出一個新的MST algorithms而是，他們對現有的algorithms提出新的fundamental contributions。

Evaluating the MST Construction

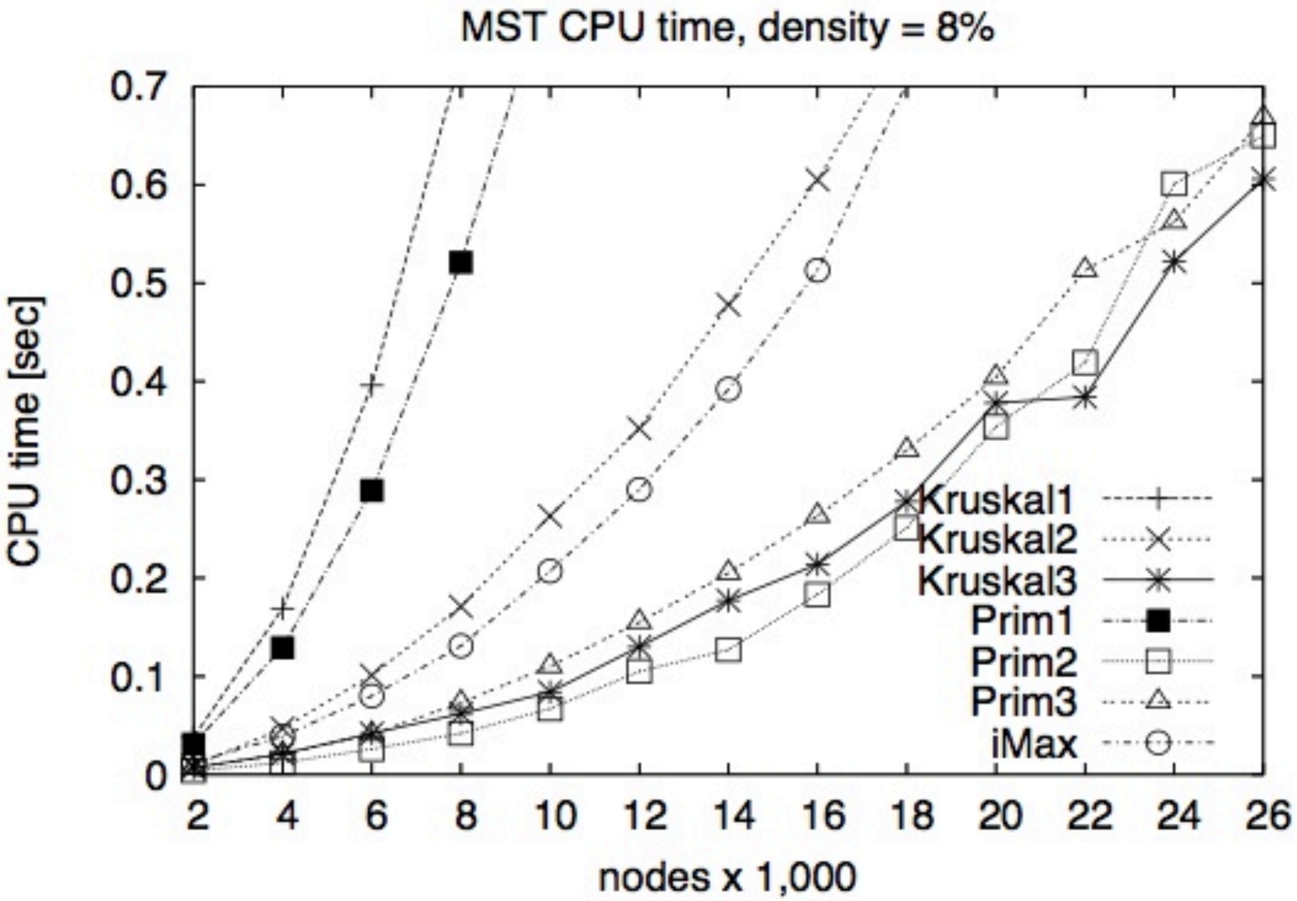
- Kruskal1 （basic Kruskal's MST）
- Kruskal2 （with demand sorting）
- Kruskal3 （IQS-based）
- Prim1 （basic Prim's MST algorithm）
- Prim2 （implemented with PH）
- Prim3 （implementation using QHs）
- iMax （iMax algorithm）

Results

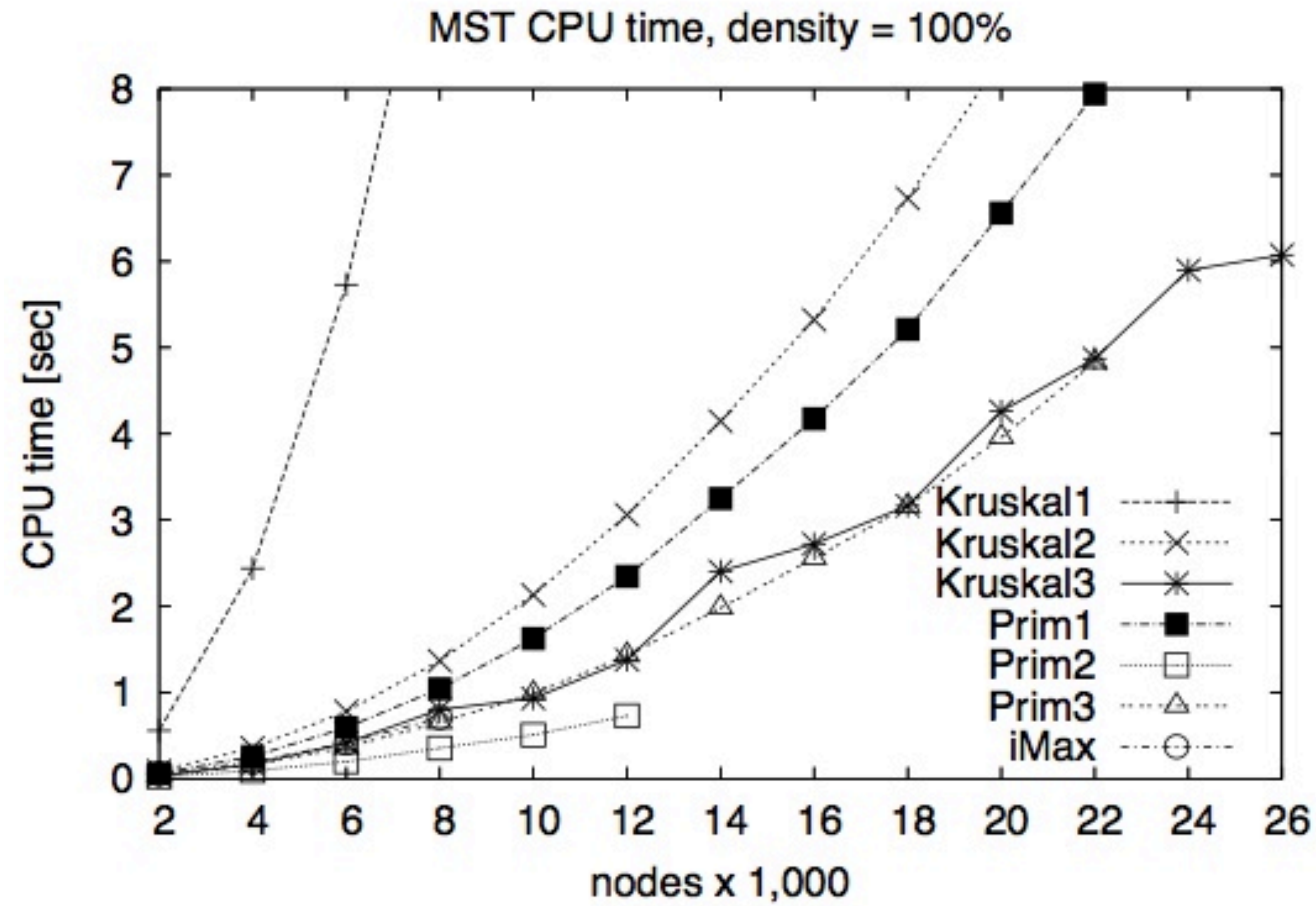


(a) MST construction CPU times.

Results

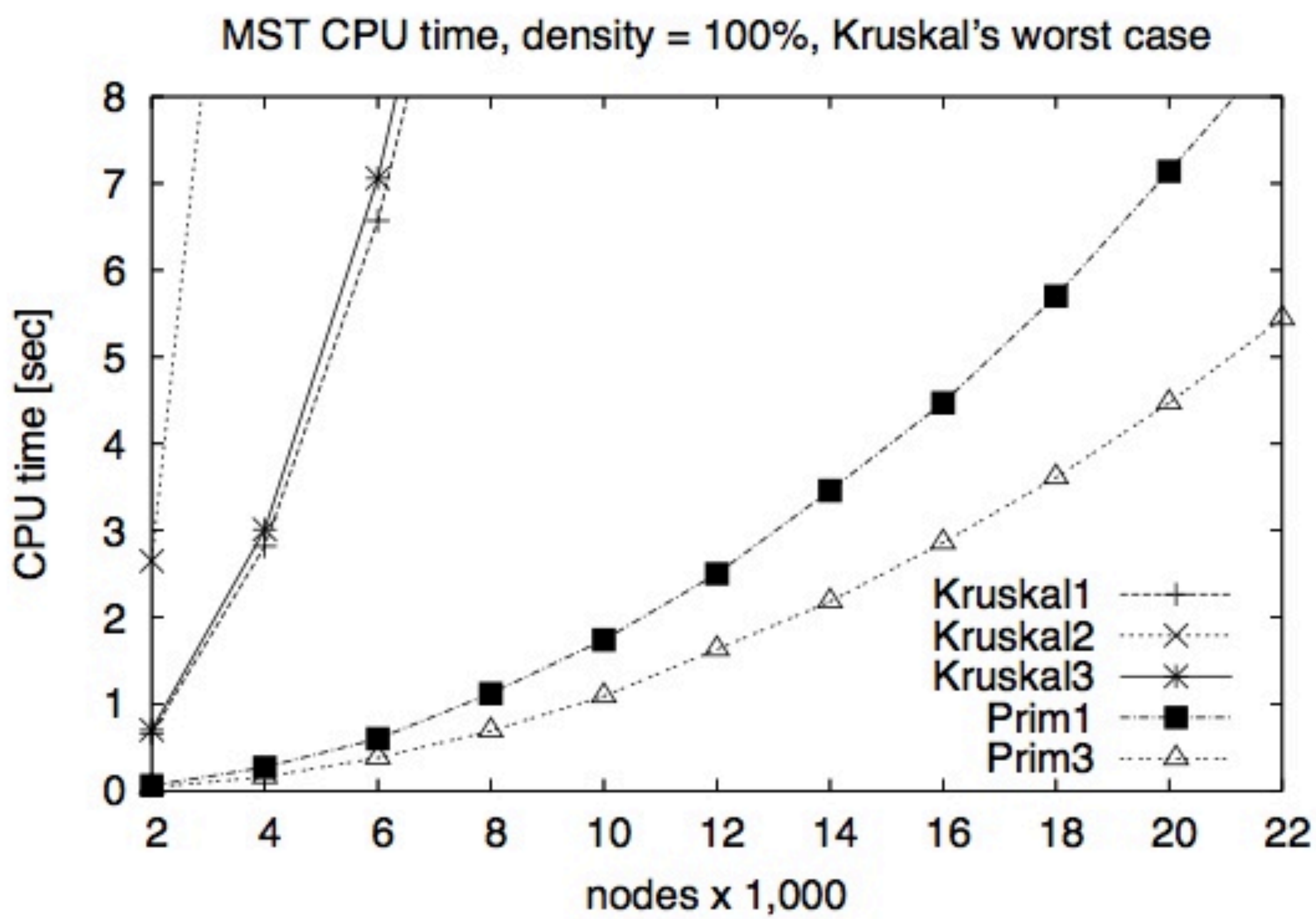


(b) MST CPU time, depending on n, for $\rho = 8\%$.



(c) MST CPU time, depending on n, for $\rho = 100\%$.

Results



(d) MST CPU time, Kruskal's worst case, for $\rho = 100\%$.

Conclusions

- IQS和現有的solution有差不多的時間複雜度，但是它實做起來相當快。
- Quickheaps執行許多動作都是高效率的
- Quickheap有高區段性參考（high locality of reference），所以在secondary memory執行起來是幾乎是最佳化。

Conclusions

- IQS跟quickheap改善了在許多scenarios下現有的演算法的performance
- incremental sort去強化Kruskal’s MST algorithm
- priority queue去強化Prim’s MST algorithm
- 最重要的future是去設計一個更強大的Quickheaps的變化（特別是他們能夠去證明Quickheap-based Prim的upper bound）