Lecture 12: Heapsort, Priority Queue & Quicksort

Agenda:

- Heapsort BC running time — all keys are distinct
- Priority queue
- Quicksort
  - A divide-and-conquer algorithm
  - The ideas & execution

Reading:

- Textbook pages 138 – 149

Heapsort BC running time — all keys distinct:

- To simplify argument, assume $n = 2^k - 1$ and so there are 2^{k-1} nodes at the bottom level of the heap (draw as a binary tree)

- To count how many KCs to put the first 2^{k-1} largest keys into positions

- In the original heap, the nodes holding these largest 2^{k-1} keys:

  If they are at the bottom level, colored RED,

  If not, colored BLUE

- RED and BLUE nodes form a binary tree (a subtree inside the original tree), without any non-colored nodes inside, according to heap property.

- Therefore, there are $\leq 2^{k-2}$ RED nodes and so $\geq 2^{k-2}$ BLUE nodes.

- To count how many KCs to put the BLUE keys into positions

- Ask: how did those BLUE keys leave the heap?

  They have to be moved up to the root.

  — **not through exchange !!!** why ???

  — **BLUE keys cannot sink down to the bottom level during the first 2^{k-1} iterations ...**

Heapsort BC running time — all keys distinct (cont'd):

- So, ...
  - total #KCs
  - $\geq$ #KCs for the first 2^{k-1} key extractions
  - $\geq$ #KCs to move all BLUE keys to the root
  - $\geq$ sum of the depths of BLUE nodes in the binary tree
  - $\geq \sum_{j=1}^{2^{k-2}} \left\lceil \lg(j+1) \right\rceil \in \Theta(n \log n)$

- So, ...
  - running time $\in \Omega(n \log n)$
  - Since WC in $\Theta(n \log n)$, it is $\in \Theta(n \log n)$ too.

Priority Queue:

- An abstract data structure for maintaining a set S of *elements* each associated with a *key*
- Key — represents the priority of the element
- Defined by operations, not implementation
- Operations:
  - `initialize` — insert all keys at once
  - `insert` — a new element
  - `maximum` — return the element with the maximum key
  - `extract maximum` — return the maximum and remove the element from the queue
  - `increase key` — increase the priority for an element
- Implementation? Heap !!!

  Arrange the priorities (keys) into a max-heap, and

  make a link from each priority to the corresponding element

  — rearrangement of priorities $\Longleftrightarrow$ rearrangement of elements.

  - `Initialize`(A) — Build-Max-Heap
  - `Maximum`(A) — return $A[1]$
  - `Extract-Maximum`(A) — return $A[1]$ and when $heapsize(A) > 1$, decrease heapsize, pull the last key to the top, and Max-Heapify (Question: when $heapsize(A) \leq 1$?)
  - `Increase-Key`(A, i, new_key) — increase the priority value for $A[i]$ and bubble up to the right position
  - `Insert`(A, new_key) — increase heapsize, add a new key with priority value equal to new_key, and bubble up

Another sorting meets divide-and-conquer:

- The ideas:
  - Pick one key
  - Compare to others: partition into 'smaller' and 'greater' sublists
  - Recursively sort two sublists
- Pseudocode:

```
procedure Quicksort(A, p, r)      **p 146

   if p < r then
      q ← Partition(A, p, r)
      Quicksort(A, p, q − 1)
      Quicksort(A, q + 1, r)


procedure Partition(A, p, r)      **p 146
   ** A[r] is the key picked to do the partition

   x ← A[r]
   i ← p − 1
   for j ← p to r − 1 do
      if A[j] ≤ x then
         i ← i + 1
         exchange A[i] ↔ A[j]
   exchange A[i + 1] ↔ A[r]
   return i + 1
```

`Partition`(A, p, r):

- The invariant:
  - $A[p..i]$ contains keys $\leq A[r]$
  - $A[(i+1)..(j-1)]$ contains keys $> A[r]$

- Ideas:
  - $A[j]$ is the current key under examination — $j \geq i$
  - If $A[j] \leq A[r]$, exchange $A[j] \leftrightarrow A[i+1]$ and increment i to maintain the invariant
  - At the end, exchange $A[r] \leftrightarrow A[i+1]$ such that:
    * $A[p..i]$ contains keys $\leq A[i+1]$
    * $A[(i+2)..r]$ contains keys $> A[i+1]$
    * After $A[p..i]$ and $A[(i+2)..r]$ been sorted, $A[p..r]$ is sorted.

- An example: $A[1..8] = \{3, 1, 7, 6, 4, 8, 2, 5\}$, $p = 1$, $r = 8$

3	1	7	6	4	8	2	5	$i = 0, j = 1$
3	1	7	6	4	8	2	5	$i = 1, j = 2$
3	1	7	6	4	8	2	5	$i = 2, j = 3$
3	1	7	6	4	8	2	5	$i = 2, j = 4$
3	1	4	6	7	8	2	5	$i = 3, j = 5$
3	1	4	6	7	8	2	5	$i = 3, j = 6$
3	1	4	6	7	8	2	5	$i = 3, j = 7$
3	1	4	2	7	8	6	5	$i = 4, j = 7$
3	1	4	2	5	8	6	7	$i = 4, j = 7$

Have you understood the lecture contents?

well	ok	not-at-all	topic
☐	☐	☐	BC running time (two cases)
☐	☐	☐	priority queue
☐	☐	☐	priority queue operations (via heap)
☐	☐	☐	quicksort idea
☐	☐	☐	quicksort pseudocode(s), execution