Lecture 11: Heapsort & Its Analysis

Agenda:

- Heap recall:
  - Heap: definition, property
  - Max-Heapify
  - Build-Max-Heap

- Heapsort algorithm

- Running time analysis

Reading:

- Textbook pages 127 – 138
(Binary-)Heap data structure (recall):

- An array $A[1..n]$ of $n$ comparable keys either $\geq$ or $\leq$

- An implicit binary tree, where
  - $A[2j]$ is the left child of $A[j]$
  - $A[2j + 1]$ is the right child of $A[j]$
  - $A[\lfloor j/2 \rfloor]$ is the parent of $A[j]$

- Keys satisfy the max-heap property: $A[\lfloor j/2 \rfloor] \geq A[j]$

- There are max-heap and min-heap. We use max-heap.

- $A[1]$ is the maximum among the $n$ keys.

- Viewing heap as a binary tree, height of the tree is $h = \lfloor \lg n \rfloor$. Call the height of the heap.
  [— the number of edges on the longest root-to-leaf path]

- A heap of height $k$ can hold $2^k$ —— $2^{k+1} - 1$ keys.
  Why ??

Since $\lg n - 1 < k \leq \lg n$

$\iff n < 2^{k+1}$ and $2^k \leq n$

$\iff 2^k \leq n < 2^{k+1}$
Lecture 11: Heapsort

Max-Heapify (recall):

- It makes an almost-heap into a heap.

- Pseudocode:

  procedure Max-Heapify(A, i)  **p 130
  **turn almost-heap into a heap
  **pre-condition: tree rooted at A[i] is almost-heap
  **post-condition: tree rooted at A[i] is a heap

  $lc \leftarrow \text{leftchild}(i)$
  $rc \leftarrow \text{rightchild}(i)$
  if $lc \leq \text{heapsize}(A)$ and $A[lc] > A[i]$ then
    $\text{largest} \leftarrow lc$
  else
    $\text{largest} \leftarrow i$
  if $rc \leq \text{heapsize}(A)$ and $A[rc] > A[\text{largest}]$ then
    $\text{largest} \leftarrow rc$
  if $\text{largest} \neq i$ then
    exchange $A[i] \leftrightarrow A[\text{largest}]$
  Max-Heapify(A, largest)

- WC running time: $\lg n$. 

Lecture 11: Heapsort

Build-Max-Heap (recall):

- **Given:** an array of \( n \) keys \( A[1], A[2], \ldots, A[n] \)
- **Output:** a permutation which is a heap
- **Ideas:**
  Repeatedly apply Max-Heapify to nodes in the binary tree representation
  — bottom up
- **Pseudocode:**

  ```plaintext
  procedure Build-Max-Heapify(A) **p 133
    **turn an array into a heap
    
    heapsize(A) ← length[A]
    for \( i ← \left\lfloor \frac{\text{length}[A]}{2} \right\rfloor \) downto 1
      do Max-Heapify(A, i)
  
  • WC running time:
    \( \lg n + 2(\lg n - 1) + 2^2(\lg n - 2) + \ldots + 2^{(\lg n - 1)} \cdot 1 = 2n - \lg n - 2. \)```
Lecture 11: Heapsort

Heapsort algorithm:

- Heapsort is a data structure algorithm.

- The ideas:
  - Build the array into a heap (WC cost $\Theta(n)$)
  - The first key $A[1]$ is the maximum and thus should be in the last position when sorted
  - Max-Heapify the array $A[1..(n-1)]$, which is an almost-heap

- An example: $A[1..10] = \{4, 1, 7, 9, 3, 10, 14, 8, 2, 16\}$
  Build into a heap:
  
  1
   2 3
  4 5 6 7
  8 9 10
Heapsort algorithm (cont’d):

- **Heapsort** is a data structure algorithm.

- The ideas:
  - Build the array into a heap (WC cost $\Theta(n)$)
  - The first key $A[1]$ is the maximum and thus should be in the last position when sorted
  - Max-Heapify the array $A[1..(n-1)]$, which is an almost-heap

- An example: $A[1..10] = \{4, 1, 7, 9, 3, 10, 14, 8, 2, 16\}$

  Heapsize = 10:

  \[
  \begin{array}{ccccccc}
  & & 1 & 16 & & & \\
  & 2 & 9 & & 3 & 14 & \\
  4 & 8 & 5 & 3 & 6 & 10 & 7 & 7 \\
  8 & 4 & 9 & 2 & 10 & 1 & \\
  \end{array}
  \]
Heapsort algorithm (cont’d):

- **Heapsort** is a data structure algorithm.

- **The ideas:**
  - Build the array into a heap (WC cost $\Theta(n)$)
  - The first key $A[1]$ is the maximum and thus should be in the last position when sorted
  - Max-Heapify the array $A[1..(n−1)]$, which is an almost-heap

- **An example:** $A[1..10] = \{4, 1, 7, 9, 3, 10, 14, 8, 2, 16\}$


  1 1
  
  2 9 3 14
  
  4 8 5 3 6 10 7 7
  
  8 4 9 2 10 16
Heapsort algorithm (cont’d):

- Heapsort is a data structure algorithm.

- The ideas:
  - Build the array into a heap (WC cost $\Theta(n)$)
  - The first key $A[1]$ is the maximum and thus should be in the last position when sorted
  - Max-Heapify the array $A[1..(n-1)]$, which is an almost-heap

- An example: $A[1..10] = \{4, 1, 7, 9, 3, 10, 14, 8, 2, 16\}$
  Resultant tree: Heapsize = 9:

```
          1
          14
          2
          9
          3
          10
         4
         8
         5
         3
         6
         1
         7
         7
     8
     4
     9
     2
     10
     16
```
Heapsort algorithm (cont’d):

- **Pseudocode:**

  ```
  procedure Heapsort(A) **p 136
  **post-condition: sorted array

  Build-Max-Heap(A)
  for i ← length[A] downto 2 do
    heapsize(A) ← heapsize(A) − 1
    Max-Heapify(A, 1)
  ```

- **WC running time analysis:**
  - Build-Max-Heap in \(2n - \lg n - 2\)
  - For each \(i\), Max-Heapify in \(\lg i\)
    sum to \(\sum_{i=2}^{n} \lg i \in \Theta(n \log n)\)
  - So, in total \(\Theta(n \log n)\)

- **Questions:**
  1. What is the Worst Case (array) for Build-Max-Heap?
  2. What is the Worst Case (heap) for the for loop?
  3. What is the Worst Case (array) for Heapsort?
Heapsort algorithm (cont’d):

- BC running time analysis:
  - all keys equal:
    $\Theta(n)$
  - all keys distinct:
    $\Theta(n \log n)$ — next lecture

- AC running time analysis — very complicated, not required
  - But when all keys distinct:
    $\Theta(n \log n)$ — why ???

- Space requirement:
  $\Theta(1)$ — in space sorting algorithm

- Correctness:
  By Loop Invariants:
  - correctness for Max-Heapify (which is a recursion)
  - LI for Build-Max-Heap (p. 133)
  - LI for heapsort (p. 136, Ex 6.4-2)
### Lecture 11: Heapsort

Have you understood the lecture contents?

<table>
<thead>
<tr>
<th>well</th>
<th>ok</th>
<th>not-at-all</th>
<th>topic</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td>heap, almost-heap</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Max-Heapify</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Build-Max-Heap</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>heapsort algorithm &amp; idea</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>heapsort analysis (WC running time)</td>
</tr>
</tbody>
</table>