This week

- Finish Quick-Sort
- Heaps
### Partition Routine

Partition($A, p, r$)

1. $x = A(r)$
2. $i = p - 1$
3. for $j = p$ to $r - 1$
   - if $A(j) \leq x$ then
     - $i++$
     - exchange$(A(i), A(j))$
   - exchange$(A(i+1), A(r))$
4. return$(i+1)$

<table>
<thead>
<tr>
<th>$&lt;= x$</th>
<th>$&gt; x$</th>
<th>$??$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$p$</td>
<td>$i$</td>
<td>$j$</td>
</tr>
</tbody>
</table>

### Quick Sort

Quicksort($A, p, r$)

1. while $(p < r)$
   - $q = \text{partition}(A, p, r)$
   - Quicksort$(A, p, q-1)$
   - Quicksort$(A, q+1, r)$
2. end

- To simplify, assume distinct elements:
  - Lucky always an even element: $T(n) = 2T(n/2) + \Theta(n) = \Theta(n \log n)$
  - Unlucky: $T(n) = 2T(0) + T(n-1) + \Theta(n) = \Theta(n^2)$
- How to avoid bad case?
  - Partition around middle element (does not work!)
  - Idea: Partition around a random element!
Quicksort (cont’d.)

- Partition around a Randomly chosen element and let $T(n)$ be the expected time to sort.
- Consider the case where the partition is $(k, n-k-1)$. In this case, the expected time to terminate is:

$$T(k) + T(n-k-1) + \Theta(n)$$

- Condition on $k$ being a specific value, note that any value of $k$ from 0 to $n-1$ is equally likely:

$$T(n) = \sum_k \Pr[(k, n-k-1)\text{ split}]T(n \mid (k, n-k-1)\text{ split})$$

$$= \frac{1}{n} \sum_k [T(k) + T(n-k-1) + \Theta(n)]$$

$$= \frac{2}{n} \sum_{k=1}^{n-1} [T(k) + \Theta(n)]$$

Solving the recurrence

- Next: We try to prove that $T(n) \leq an \log n + b$

First, Choose $b$ large enough to satisfy $T(1) \leq a \log 1 + b = b$

Inductive step:

$$T(n) = \frac{2}{n} \sum_{k=1}^{n-1} T(k) + \Theta(n) \leq \frac{2}{n} \sum_{k=1}^{n-1} ak \log k + b + \Theta(n)$$

$$= \frac{2a}{n} \sum_{k=1}^{n-1} k \log k + \frac{2n}{n} nb + \Theta(n)$$

Need to prove this is $\leq \frac{1}{2} n \log n - \frac{1}{8} n^2$

$$\leq \frac{2a}{n} \left( \frac{1}{2} n^2 \log n - \frac{1}{8} n^2 \right) + 2b + \Theta(n)$$

$$= an \log n + b + \left( \Theta(n) + b - \frac{an}{4} \right)$$
Technical Lemma

- We need to show $n^2 \log n$ bound is true.

$$\sum_{k=1}^{n-1} k \log k = \sum_{k=1}^{n-1} k \log k + \sum_{k=\lceil n/2 \rceil}^{n-1} k \log k$$

$$\leq \log n \left( \sum_{k=1}^{n-1} k - \sum_{k=1}^{n/2-1} k \right)$$

$$\leq \log n \left( \frac{n(n-1)}{2} - \frac{n-1}{2} \right)$$

$$\leq \frac{1}{2} n^2 \log n - \frac{n^2}{8}$$

Heaps, Priority Queues and Heap Sort
Priority Queue

- Handles a collection of items, called keys.
- There exists a way to compare keys to each other. This is called an order relation.
- The result of these comparisons determines the priority of the keys.
- Operations supported:
  - *insert a key*
  - *Remove* the largest key

Applications

- Scheduling
- Operating systems
- Keeping track of largest $n$ elements in a sequence
- *Sorting*
Methods of a Priority Queue

- **Initialize**: initialize the structure
- **Insert (key)**: insert a new key
- **Remove Max**: return and remove largest key

**PQ-Sort in procedural pseudocode**
- (sorting an array with using a priority queue)
  - **Initialize**
  - **for** $i = 1$ to $n$
    - **Insert**($a[i]$)
  - **for** $i := n$ downto $1$
    - $a[i] := RemoveMax$

How to Implement a Priority Queue

<table>
<thead>
<tr>
<th>Implementation</th>
<th>Insert</th>
<th>Remove Max</th>
<th>Delete</th>
<th>Average</th>
</tr>
</thead>
<tbody>
<tr>
<td>Unsorted Array or Linked List</td>
<td>$O(1)$</td>
<td>$O(n)$</td>
<td>$O(1)$</td>
<td>$O(n)$</td>
</tr>
<tr>
<td>Sorted Array or Linked List</td>
<td>$O(n)$</td>
<td>$O(1)$</td>
<td>$O(1)$</td>
<td>$O(n)$</td>
</tr>
<tr>
<td>Heap</td>
<td>$O(\log n)$</td>
<td>$O(\log n)$</td>
<td>$O(\log n)$</td>
<td>$O(\log n)$</td>
</tr>
</tbody>
</table>
A heap is a binary tree storing keys, with the following properties:
- partial order:
  - key (child) < key(parent)
- left-filled levels:
  - the last level is left-filled
  - the other levels are full

Logarithmic Height

- A heap with $n$ keys has height: $H(n) = \log_2 n$
- Proof:
  - Let $n$ be the number of keys, and $H(n)$ be the height.
  - We have:
    \[ 2^{H(n)-1} \leq n \leq 2^{H(n)} \]
  - Taking logarithm of both sides; the result will follow.
Heap Representations

- left_child(i) = 2i
- right_child(i) = 2i + 1
- parent(j) = j div 2

<table>
<thead>
<tr>
<th>X</th>
<th>T</th>
<th>O</th>
<th>G</th>
<th>S</th>
<th>M</th>
<th>N</th>
<th>A</th>
<th>E</th>
<th>R</th>
<th>B</th>
<th>I</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>9</td>
<td>10</td>
<td>11</td>
<td>12</td>
</tr>
</tbody>
</table>

Heap Insertion

- Add the key in the next available spot in the heap.
- Upheap checks if the new node is greater than its parent. If so, it switches the two.
- Upheap continues up the tree
Heap Insertion

- **Upheap** terminates when new key is less than the key of its parent or the top of the heap is reached.
- (total #switches) \( \leq (\text{height of tree} - 1) = \log n \)
Heapify Algorithm

- Assumes L and R sub-trees of i are already Heaps and makes It's sub-tree a Heap:
  
  \[
  \text{Heapify}(A, i, n) \\
  \text{If } (2i \leq n) \land (A[2i] > A[i]) \text{ Then} \\
  \quad \text{largest} = 2i \\
  \text{Else largest} = i \\
  \text{If } (2i+1 \leq n) \land (A[2i+1] > A[\text{largest}]) \text{ Then} \\
  \quad \text{largest} = 2i+1 \\
  \text{If } (\text{largest} \neq i) \text{ Then} \\
  \quad \text{Exchange} (A[i], A[\text{largest}]) \\
  \quad \text{Heapify}(A, \text{largest}, n) \\
  \text{Endif} \\
  \text{End Heapify}
  \]

Extracting the Maximum from a Heap:

- Here is the algorithm:
  
  \[
  \text{Heap-Extract-Max}(A) \\
  \text{Remove } A[1] \\
  n = n - 1 \\
  \text{Heapify}(a, 1, n) \\
  \text{End Heap-Extract-Max}
  \]
Building a Heap

- Builds a heap from an unsorted array:
  \[ \text{Build\_Heap}(A, n) \]
  \[ \text{For } i = \text{floor}(n/2) \text{ down to } 1 \text{ do} \]
  \[ \text{Heapify}(A, i, n) \]
  \[ \text{End Build\_Heap} \]
- Example:

\[
\begin{array}{c}
A = [4, 1, 3, 2, 16, 9, 10, 14, 8, 7] \\
\end{array}
\]

Building a Heap (cont’d.)

\[
\begin{array}{c}
\end{array}
\]

Building a Heap (cont’d.)

Running time of Building a Heap

- $O(n \log n)$ is trivial: $n$ calls of Heapify, each of cost $O(\log n)$
- Tighter Bound: $O(n)$
  - The cost of “Heapify” is proportional to the number of levels visited (height of node $i$)
  - Assume $n=2^k-1$ (complete binary tree):
    - For each leaf node, the number of levels visited is 1,
    - For each node at next level is 2,
    - 3 for next level, etc.

Total # of levels visited:

$$\sum_{i=0}^{\log_2(n+1)} \frac{i}{2^i} + \frac{n+1}{2} \times 1 + \frac{n+1}{4} \times 2 + \frac{n+1}{8} \times 3 + \cdots + \frac{n+1}{2^{\log_2(n+1)}} \times \log(n+1)$$
Running time of Building a Heap (cont’d.)

- Using Induction, it is easy to see that

\[ \sum_{i=1}^{\log(n+1)} \frac{i}{2^i} = O(1) \]

Implying:

\[ T(n) = \text{Total } \# \text{ of levels visited} = O(n) \]