CS 253: Algorithms

Chapter 6

Heapsort
Appendix B.5
Special Types of Trees

- **Def:** Full binary tree
  a binary tree in which each node is either a leaf or has degree exactly 2.

- **Def:** Complete binary tree
  a binary tree in which all leaves are on the same level and all internal nodes have degree 2.
Definitions

- **Height** of a node = the number of edges on the longest simple path from the node down to a leaf
- **Level** of a node = the length of a path from the root to the node
- **Height** of tree = height of root node

Height of root = 3
Height of (2) = 1
Level of (10) = 2
Useful Properties

- There are at most $2^l$ nodes at level (or depth) $l$ of a binary tree.
- A binary tree of height $h$ has at most $(2^{h+1} - 1)$ nodes.
- A binary tree with $n$ nodes has height at least $\lceil \lg n \rceil$.  

\[ n \leq \sum_{i=0}^{h} 2^i = \frac{2^{h+1} - 1}{2 - 1} = 2^{h+1} - 1 \]

Height of root = 3

Height of (2) = 1

Level of (10) = 2
The Heap Data Structure

- **Def:** A heap is a nearly complete binary tree with the following two properties:
  - **Structural property:** all levels are full, except possibly the last one, which is filled from left to right
  - **Order (heap) property:** for any node $x$, $\text{Parent}(x) \geq x$

From the heap property, it follows that:

The root is the maximum element of the heap!
A heap can be stored as an array $A$.

- Root of tree is $A[1]$
- Parent of $A[i] = A[\lfloor i/2 \rfloor]$
- Heapsize[$A$] $\leq$ length[$A$]

The elements in the subarray $A[(\lfloor n/2 \rfloor + 1) .. n]$ are leaves.
Heap Types

- **Max-heaps** (largest element at root), have the *max-heap property*:
  - for all nodes $i$, excluding the root: $A[\text{PARENT}(i)] \geq A[i]$

- **Min-heaps** (smallest element at root), have the *min-heap property*:
  - for all nodes $i$, excluding the root: $A[\text{PARENT}(i)] \leq A[i]$
Adding/Deleting Nodes

- New nodes are always **inserted** at the bottom level (**left to right**)
- Nodes are **deleted** from the bottom level (**right to left**)

![Binary Tree Diagram](image-url)
Operations on Heaps

- Maintain/Restore the max-heap property
  - MAX-HEAPIFY

- Create a max-heap from an unordered array
  - BUILD-MAX-HEAP

- Sort an array in place
  - HEAPSORT

- Priority queues
Maintaining the Heap Property

**MAX-HEAPIFY**

- Suppose a node $i$ is smaller than a child and Left and Right subtrees of $i$ are max-heaps

- How do you fix it?

- To eliminate the violation:
  - Exchange node $i$ with the larger child
  - Move down the tree
  - Continue until node is not smaller than children

- **MAX-HEAPIFY**
Example

MAX-HEAPIFY(A, 2, 10)

A[2] violates the heap property

A[4] violates the heap property


Heap property restored
Maintaining the Heap Property

- **Assumptions:**
  - Left and Right subtrees of $i$ are max-heaps
  - $A[i]$ may be smaller than its children

**Alg:** MAX-HEAPIFY($A$, $i$, $n$)

1. $l \leftarrow \text{LEFT}(i)$
2. $r \leftarrow \text{RIGHT}(i)$
3. if $l \leq n$ and $A[l] > A[i]$ then $\text{largest} \leftarrow l$ 
   else $\text{largest} \leftarrow i$
4. if $r \leq n$ and $A[r] > A[\text{largest}]$ then $\text{largest} \leftarrow r$
5. if $\text{largest} \neq i$ then exchange $A[i] \leftrightarrow A[\text{largest}]$
6. MAX-HEAPIFY($A$, $\text{largest}$, $n$)
MAX-HEAPIFY Running Time

- It traces a path from the root to a leaf (longest path length \( h \))
- At each level, it makes two comparisons
- Total number of comparisons = \( 2h \)
- Running Time of MAX-HEAPIFY = \( O(2h) = O(\log n) \)
Building a Heap

- Convert an array $A[1 \ldots n]$ into a **max-heap** ($n = \text{length}[A]$)
- The elements in the subarray $A[(\lfloor n/2 \rfloor+1) \ldots n]$ are leaves
- Apply MAX-HEAPIFY on elements between 1 and $\lfloor n/2 \rfloor$ (bottom up)

**Alg:** \textsc{Build-Max-Heap}(A)

1. $n = \text{length}[A]$
2. \textbf{for} $i \leftarrow \lfloor n/2 \rfloor$ \textbf{downto} 1
3. \textbf{do} MAX-HEAPIFY(A, i, n)

A: \begin{tabular}{cccccccc}
4 & 1 & 3 & 2 & 16 & 9 & 10 & 14 & 8 & 7
\end{tabular}
Example:

```
i = 5
4
  /\   \
 1   3
 /     \
16    9
 /       \
8      10
```

```
i = 4
4
  /\   \
 1   3
 /     \
16    9
 /       \
8      7
```

```
i = 3
4
  /\   \
 1   3
 /     \
16    9
 /       \
8      7
```

```
i = 2
4
  /\   \
 1   10
 /     \
16    9
 /       \
8      7
```

```
i = 1
4
  /\   \
 1   10
 /     \
16    9
 /       \
8      7
```

```
i = 1
4
  /\   \
 1   10
 /     \
16    9
 /       \
8      7
```

```
i = 1
4
  /\   \
 1   10
 /     \
16    9
 /       \
8      7
```
Running Time of BUILD MAX HEAP

\textbf{Alg: BUILD-MAX-HEAP}(A)

1. \( n = \text{length}[A] \)
2. \( \text{for } i \leftarrow \lfloor n/2 \rfloor \text{ downto 1} \)
3. \( \text{do MAX-HEAPIFY}(A, i, n) \)

\( \Rightarrow \text{Running time: } O(n \log n) \)

- This is not an asymptotically tight upper bound! Why?
Running Time of BUILD MAX HEAP

- MAX-HEAPIFY takes $O(h)$ \( \Rightarrow \) the cost of MAX-HEAPIFY on a node \( i \) proportional to the height of the node \( i \) in the tree

<table>
<thead>
<tr>
<th>Height</th>
<th>Level</th>
<th>No. of nodes</th>
</tr>
</thead>
<tbody>
<tr>
<td>( h_0 = 3 \lfloor \log n \rfloor )</td>
<td>( i = 0 )</td>
<td>( 2^0 )</td>
</tr>
<tr>
<td>( h_1 = 2 )</td>
<td>( i = 1 )</td>
<td>( 2^1 )</td>
</tr>
<tr>
<td>( h_2 = 1 )</td>
<td>( i = 2 )</td>
<td>( 2^2 )</td>
</tr>
<tr>
<td>( h_3 = 0 )</td>
<td>( i = 3 \lfloor \log n \rfloor )</td>
<td>( 2^3 )</td>
</tr>
</tbody>
</table>

\( h_i = h - i \)  \( \text{height of the heap rooted at level } i \)  
\( n_i = 2^i \)  \( \text{number of nodes at level } i \)

\[
T(n) = \sum_{i=0}^{h} n_i h_i = \sum_{i=0}^{h} 2^i (h - i) = O(n)
\]
Running Time of BUILD MAX HEAP

\[
T(n) = \sum_{i=0}^{h} n_i h_i
\]

Cost of HEAPIFY at level \(i\) * (# of nodes at that level)

\[
= \sum_{i=0}^{h} 2^i (h - i)
\]

Replace the values of \(n_i\) and \(h_i\) computed before

\[
= \sum_{i=0}^{h} \frac{h - i}{2^{h-i}} 2^h
\]

Multiply by \(2^h\) both at the nominator and denominator and write \(2^i\) as \((1/2^{-i})\)

\[
= 2^h \sum_{k=0}^{h} \frac{k}{2^k}
\]

Change variables: \(k = h - i\)

\[
\leq n \sum_{k=0}^{\infty} \frac{k}{2^k}
\]

The sum above is smaller than the sum of all elements to \(\infty\) and \(h = \lgn\)

\[
= n \frac{1/2}{(1 - 1/2)^2}
\]

\[
x = 1/2 \text{ in the summation } \sum_{k=0}^{\infty} kx^k = \frac{x}{(1-x)^2}
\]

\[
= O(n)
\]

Running time of BUILD-MAX-HEAP

\[T(n) = O(n)\]
Heapsort

- **Goal:**
  - Sort an array using heap representations

- **Idea:**
  - Build a **max-heap** from the array
  - Swap the root (the maximum element) with the last element in the array
  - “Discard” this last node by decreasing the heap size
  - Call MAX-HEAPIFY on the new root
  - Repeat this process until only one node remains
Example:

\[ A = [7, 4, 3, 1, 2] \]

\[
\text{MAX-HEAPIFY}(A, 1, 4)
\]

\[
\text{MAX-HEAPIFY}(A, 1, 3)
\]

\[
\text{MAX-HEAPIFY}(A, 1, 2)
\]

\[
\text{MAX-HEAPIFY}(A, 1, 1)
\]

\[
A = [1, 2, 3, 4, 7]
\]
Alg: HEAPSORT(A)

1. BUILD-MAX-HEAP(A)
2. for i ← length[A] downto 2
4. MAX-HEAPIFY(A, 1, i - 1)

• Running time: \(O(n \log n)\)
  \(\Rightarrow\) Can be shown to be \(\Theta(n \log n)\)
Priority Queues

- Each element is associated with a value (priority)
- The key with the highest (lowest) priority is extracted first

Support the following operations:

- $\text{INSERT}(S, x)$ : inserts element $x$ into set $S$
- $\text{EXTRACT-MAX}(S)$ : removes and returns element of $S$ with largest key
- $\text{MAXIMUM}(S)$ : returns element of $S$ with largest key
- $\text{INCREASE-KEY}(S, x, k)$ : increases value of element $x$’s key to $k$
  (Assume $k \geq x$’s current key value)
**HEAP-MAXIMUM**

**Goal:**
- Return the largest element of the heap

**Alg:**

Heap-Maximum(A) returns 7

Heap A:
**HEAP-EXTRACT-MAX**

**Goal:**
- Extract the largest element of the heap (i.e., return the **max value** and also remove that element from the heap)

**Idea:**
- Exchange the root element with the last
- Decrease the size of the heap by 1 element
- Call MAX-HEAPIFY on the new root, on a heap of size n-1

Heap A:

```
Root is the largest element
```

```
7

4     3

1     2
```
Example: **HEAP-EXTRACT-MAX**

Call **MAX-HEAPIFY**(*A*, 1, n-1)

max = 16

Heap size decreased with 1
**HEAP-EXTRACT-MAX**

*Alg:* HEAP-EXTRACT-MAX(A, n)

1. if n < 1
2. then error “heap underflow”
3. max ← A[1]
5. MAX-HEAPIFY(A, 1, n-1) % remakes heap
6. return max

**Running time:** $O(lgn)$
**HEAP-INCREASE-KEY**

- **Goal:**
  - Increases the key of an element $i$ in the heap

- **Idea:**
  - Increment the key of $A[i]$ to its new value
  - If the max-heap property does not hold anymore: traverse a path toward the root to find the proper place for the newly increased key

Key $[i] \leftarrow 15$
Example:  HEAP-INCREASE-KEY

Key[i] ← 15
**Alg:** HEAP-INCREASE-KEY(A, i, key)

1. if key < A[i]
2. then error “new key is smaller than current key”
3. A[i] ← key
4. while i > 1 and A[PARENT(i)] < A[i]
5. do exchange A[i] ↔ A[PARENT(i)]
6. i ← PARENT(i)

- Running time: $O(\log n)$

Key \[i\] ← 15
MAX-HEAP-INSERT

- **Goal:**
  - Inserts a new element into a max-heap

- **Idea:**
  - Expand the max-heap with a new element whose key is $-\infty$
  - Calls HEAP-INCREASE-KEY to set the key of the new node to its correct value and maintain the max-heap property
Example: MAX-HEAP-INSERT

Insert value 15: Start by inserting $-\infty$

Increase the key to 15

The restored heap containing the newly added element
Alg: MAX-HEAP-INSERT(A, key, n)

1. heap-size[A] ← n + 1
3. HEAP-INCREASE-KEY(A, n + 1, key)

Running time: \(O(\log n)\)
### Summary

We can perform the following operations on heaps:

- **MAX-HEAPIFY** \( O(\lg n) \)
- **BUILD-MAX-HEAP** \( O(n) \)
- **HEAP-SORT** \( O(n\lg n) \)
- **MAX-HEAP-INSERT** \( O(\lg n) \)
- **HEAP-EXTRACT-MAX** \( O(\lg n) \)
- **HEAP-INCREASE-KEY** \( O(\lg n) \)
- **HEAP-MAXIMUM** \( O(1) \)

Average: \( O(\lg n) \)
Priority Queue Using Linked List

Remove a key: O(1)
Insert a key: O(n)
Increase key: O(n)
Extract max key: O(1)

Average: O(n)
Problems

a. What is the maximum number of nodes in a max heap of height $h$?

b. What is the maximum number of leaves?

c. What is the maximum number of internal nodes?

d. Assuming the data in a max-heap are distinct, what are the possible locations of the second-largest element?
Problems

- Demonstrate, step by step, the operation of Build-Heap on the array $A = [5, 3, 17, 10, 84, 19, 6, 22, 9]$

- Let $A$ be a heap of size $n$. Give the most efficient algorithm for the following tasks:
  
  (a) Find the sum of all elements
  (b) Find the sum of the largest $\log n$ elements