Chapter 4
Heaps

- Definition of heaps
- Operations on heaps
- HeapSort

Data Structures

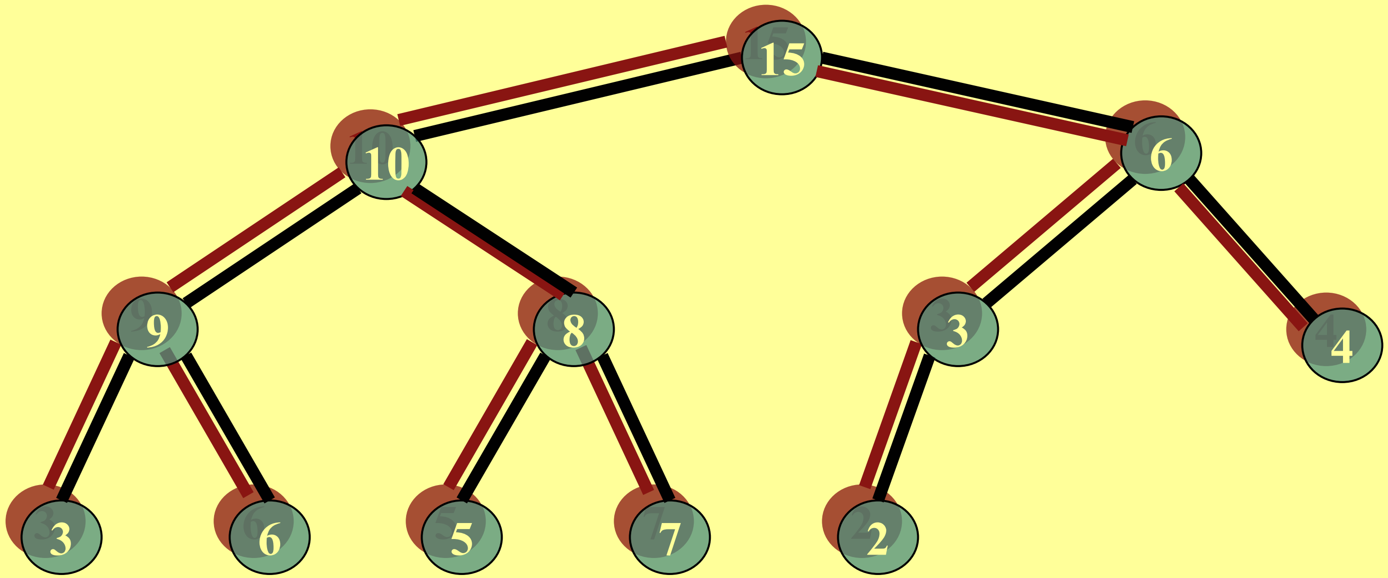
- Linked Lists
- Graphs
- Trees
- Stacks (LIFO)
- Queues (FIFO)
- Heaps (Priority Queues): Support quick insert and delete-max operations.

Definition

A heap is an almost-complete binary tree

- where every level is full except possibly the last one and it is completed from left to right,
- each node has a key,
- and the tree satisfies the following Heap Property: The key of any child node v is $\leq$ the key of its parent $p(v)$. That is, the keys along any path from a leaf to the root are ordered non-decreasingly.

Example of a Heap



Note: Heap is not a BST

Array Representation

Every heap T can be represented by an array H[1..n] where

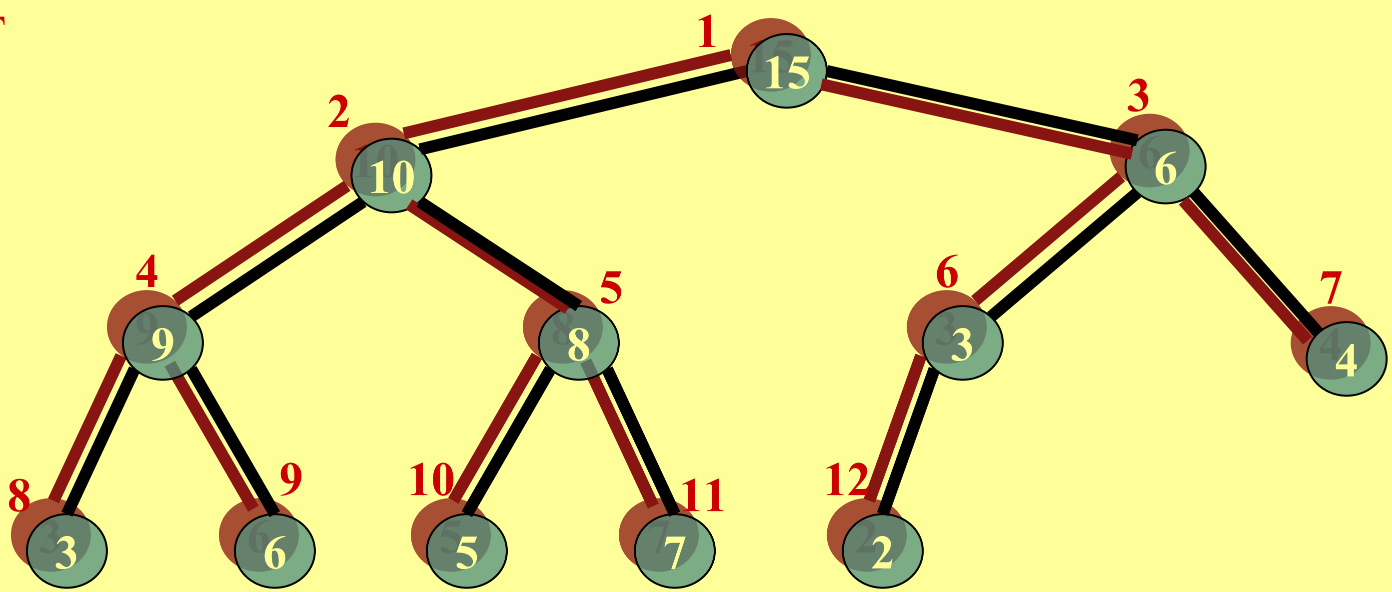
- H[1] is the root
- If *x* is a node in the heap T stored in H[i] then its left and right children (if exist) are stored in H[2i] and H[2i+1], respectively.
- The parent of x (if it is not the root) is stored in H[$\lfloor i/2 \rfloor$].

Example of a Heap

Array H

15	10	6	9	8	3	4	3	6	5	7	2
1	2	3	4	5	6	7	8	9	10	11	12

Heap T



So ....

If H is the array representation of the heap T, then

$$\text{Key}(\ H[\lfloor i/2 \rfloor]\ ) \geq \text{key}(\ H[i]\ ) \geq \text{key}(\ H[2i]\ )$$

- We will informally call H a heap.
- If H is not a heap, then

how do we make it a heap?

Operations on Heaps

- MakeHeap[A]: makes the array A into heap, i.e., makes it satisfies the heap property.
- Delete_Max[H]: deletes the max element in H.
- Insert[H, x]: inserts the element x into H.
- Delete[H, i]: Deletes the i-th element from H.

Procedures

- There are two procedure that we will use frequently within the heap operations.
  - Shift-Up
  - Shift-Down
- These procedures are used mainly for fixing (or restoring) the heap property when it gets damaged.
- They are good for keeping the heap up-to-date when the values of its items are changed.

Procedure Shift-Up

- Suppose that key(H[i]) has been changed to a greater value than key(H[⌊ i/2 ⌋]).
- Then the key at H[i] has to be moved up or indeed shifted up to its proper location.
- The procedure shift-up walks along the unique path from node H[i] to the root until it finds the correct position of the key(H[i]) where it’s no longer larger than the key of its parent.

Procedure Shift-Up

Input: Heap H[1..n] & index i in {1, 2, .., n}

Output: Heap H[1..n] where key(H[i]) is moved up to its correct position.

1. **done ← false;**
2. **if i=1 then exit; %node is the root**
3. **repeat**
4. **if key(H[i]) > key(H[⌊i/2⌋]) then**
5. **interchange H[i] & H[⌊i/2⌋]**
6. **else done ← true;**
7. **i ← ⌊i/2⌋;**
8. **until i=1 or done**

Procedure Shift-down

- Suppose that key(H[i]) (where $i \leq \lfloor n/2 \rfloor$) has been changed to a smaller value than the key of one of its children, i.e. smaller than

$$\max\Big(key(H[2i]), key(H[2i+1])\Big).$$

- Then the key at H[i] has to be moved down or indeed shifted down to its proper location.
- The procedure shift-down percolate the key of node H[i] down the binary tree. It always interchange the key of H[i] with the max key of its children.

Procedure Shift-down

Notice that to shift down the key of H[i], we must have at least a left child of H[i], i.e., we must have **$i \leq \lfloor n/2 \rfloor$**, otherwise **$2i \geq 2(\lfloor n/2 \rfloor + 1) > n$** which implies that H[i] has no left child!

Procedure Shift-down

Input: Heap H[1..n] & index i in {1, 2, .., n}

Output: Heap H[1..n] where key(H[i]) is percolated down to its correct position.

1. **done $\leftarrow$ false;**
2. **if 2i > n then exit; %node is a leaf**
3. **Repeat**
4. **   i $\leftarrow$ 2 i;**
5. **   if i+1 $\leq$ n and key(H[i+1]) > key(H[i]) then i $\leftarrow$ i+1;**
6. **   if key(H[$\lfloor$ i/2 $\rfloor$]) < key(H[i]) then**
7. **   interchange H[i] & H[$\lfloor$ i/2 $\rfloor$]**
8. **   else done $\leftarrow$ true;**
9. **until 2i > n or done**