Program Structure and Algorithms

Sid Nath

Lecture 7

Agenda

- Administrative
  - Midterm next week; All topics from Lec 1 to Lec 7
- Lecture
  - BST
  - Binary heaps and heapsort
- Quiz

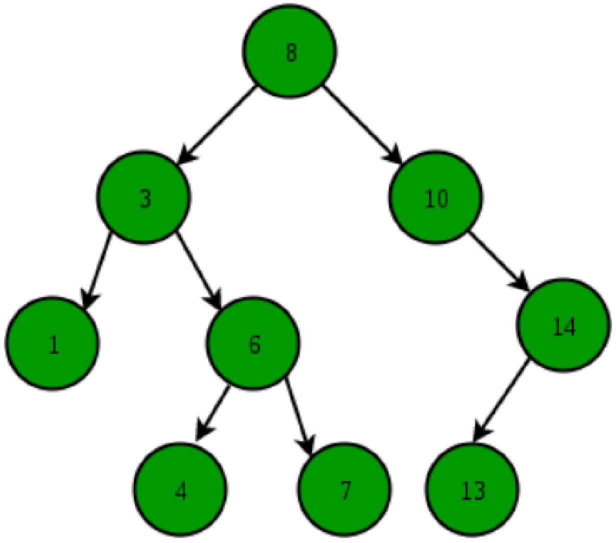
Midterm Details

- One A4 size cheatsheet, calculator OK; nothing else
- Please focus on all HW and Quiz questions till today
- Five questions
  - Short answer
  - Graph and D/Q algo execution (recursion trees, Master Theorem)
  - Graph algo design
  - D/Q algo design

Binary Search Tree

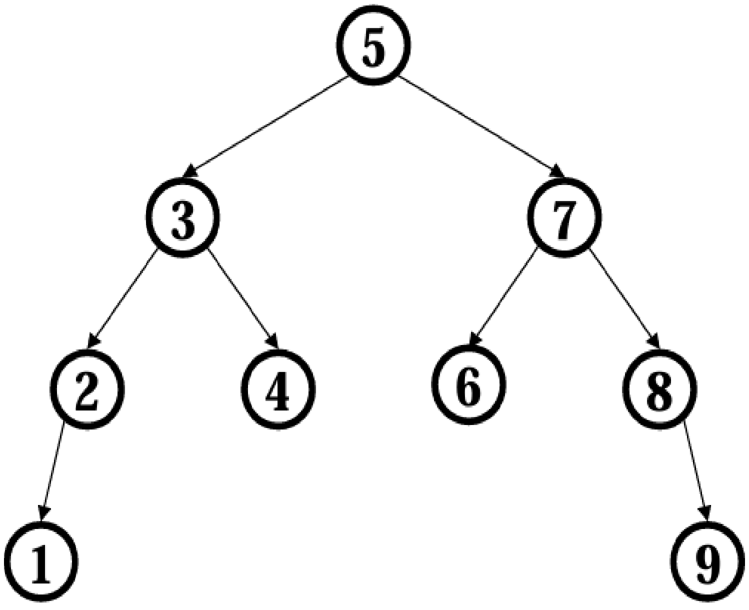
BST is a binary tree data structure with the following properties

- Left subtree contains only nodes with key values lesser than the subtree's root's key value
- Right subtree contains only nodes with key values greater than the subtree's root's key value
- Left and right subtree also must each be a BST



Create a BST

- {1, 2, 3, 4, 5, 6, 7, 8, 9}
- If inserted in order: 1+2+3+….+n = n(n+1)/2 = $O(n^2)$
- Re-arrange and find median
  - Median, left median, right median
  - 5, 3, 7, 2, 1, 4, 8, 6, 9
  - $O(n \log n)$
- Order of key values is important!
  - Better to keep tree balanced



Find in BST

```
def find(root, key):
    if root == NULL:
        return NULL
    if key < root.key:
        return find(root.left, key)
    if key > root.key:
        return find(root.right, key)
    return root.key
```

Worst-case running time: O(n)

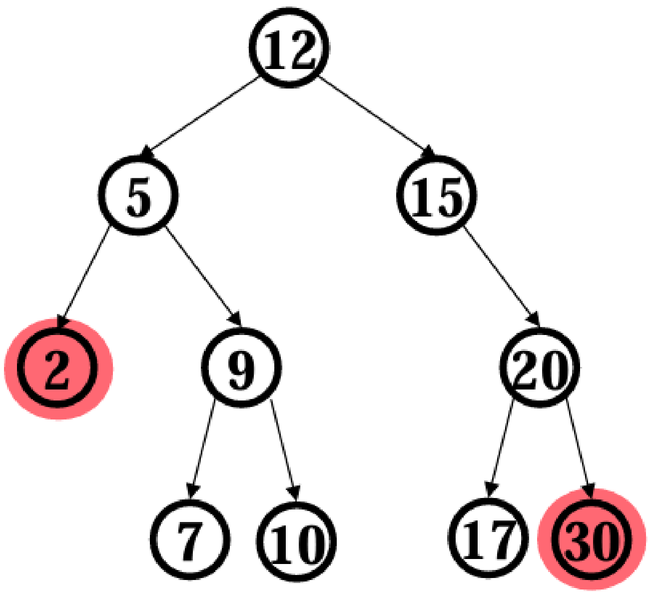
Tree is lopsided: 1 → 2 → 3 → 4

Find in BST

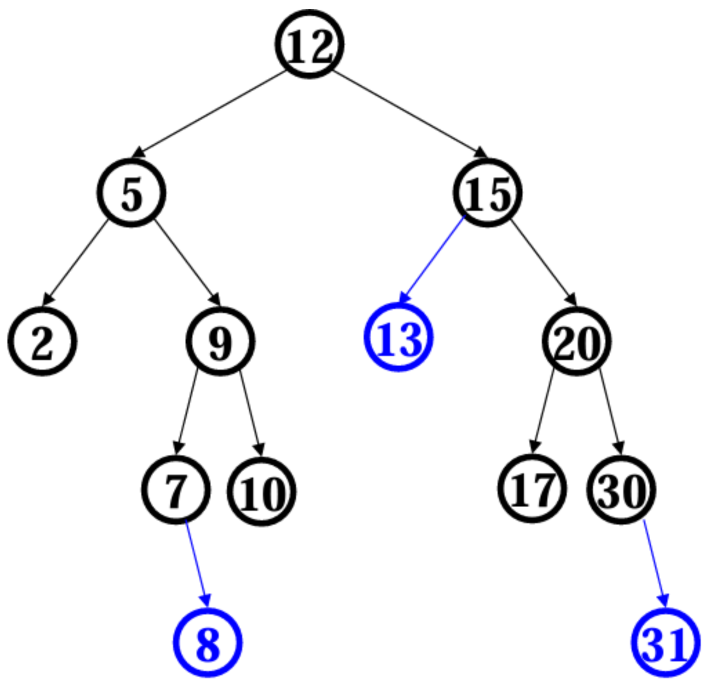
findMin: Find the node with min key value (left-most node)

findMax: Find the node with max key value (right-most node)

Try implementing these!!!



Insert in BST




```
insert(13)
insert(8)
insert(31)
```

Insertions happen only at leaves

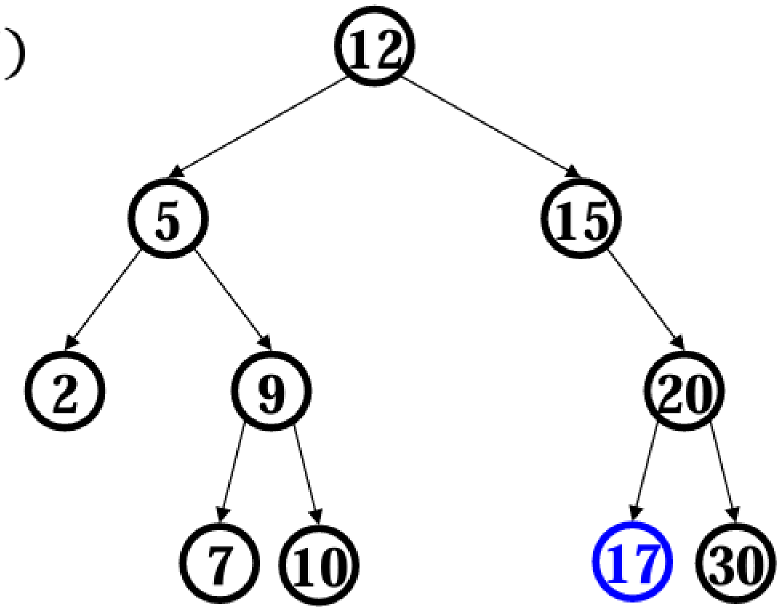
Worst-case running time: O(n)

Deletion in BST

- Removing an item disrupts the tree structure – hard!
- Idea: find the node to be removed, then “fix” the tree so that it is still a BST
- Fixing considerations
  - Node has no children (leaf)
  - Node has one child
  - Node has two children

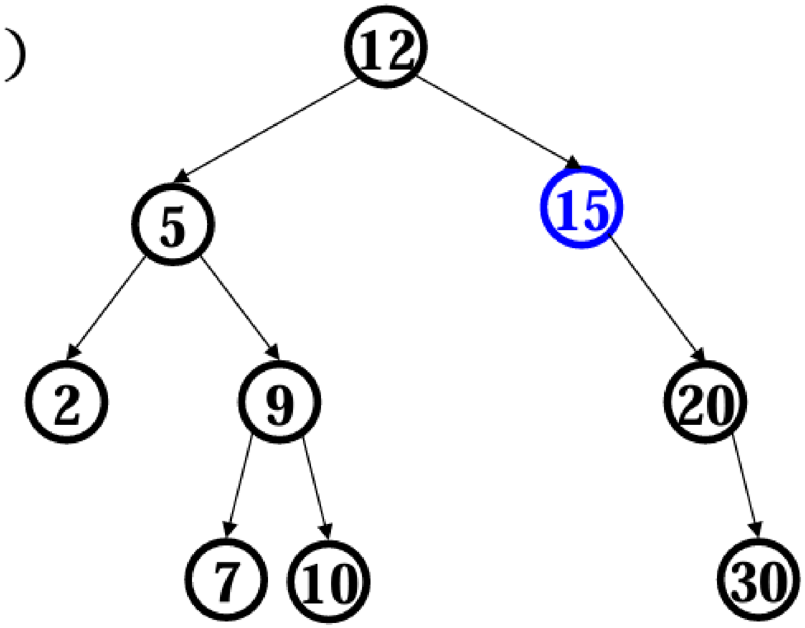
Deletion: Leaf Node

```
delete(17)
```



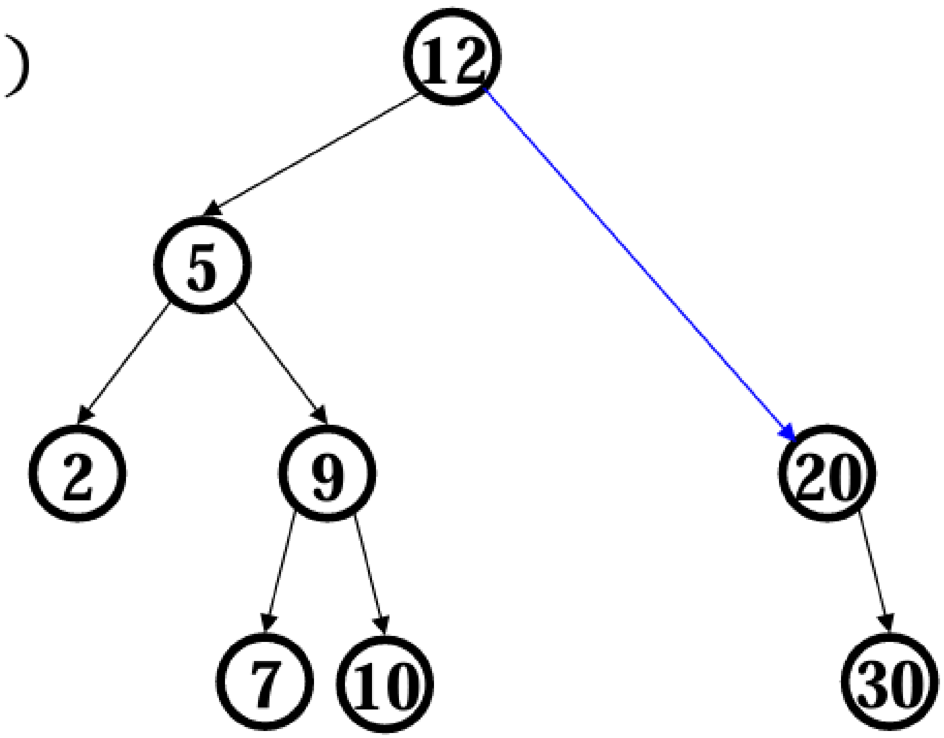
Deletion: Node w/ One Child

delete(15)



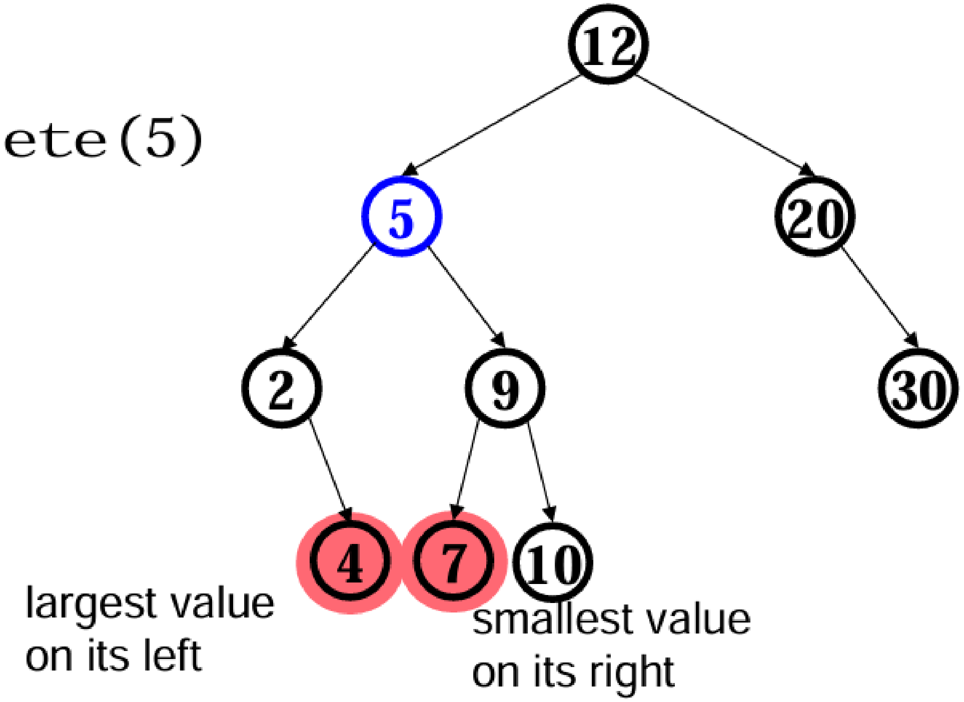
Deletion: Node w/ One Child

`delete(15)`



Deletion: Node w/ Two Children

`delete(5)`



What can we replace 5 with?

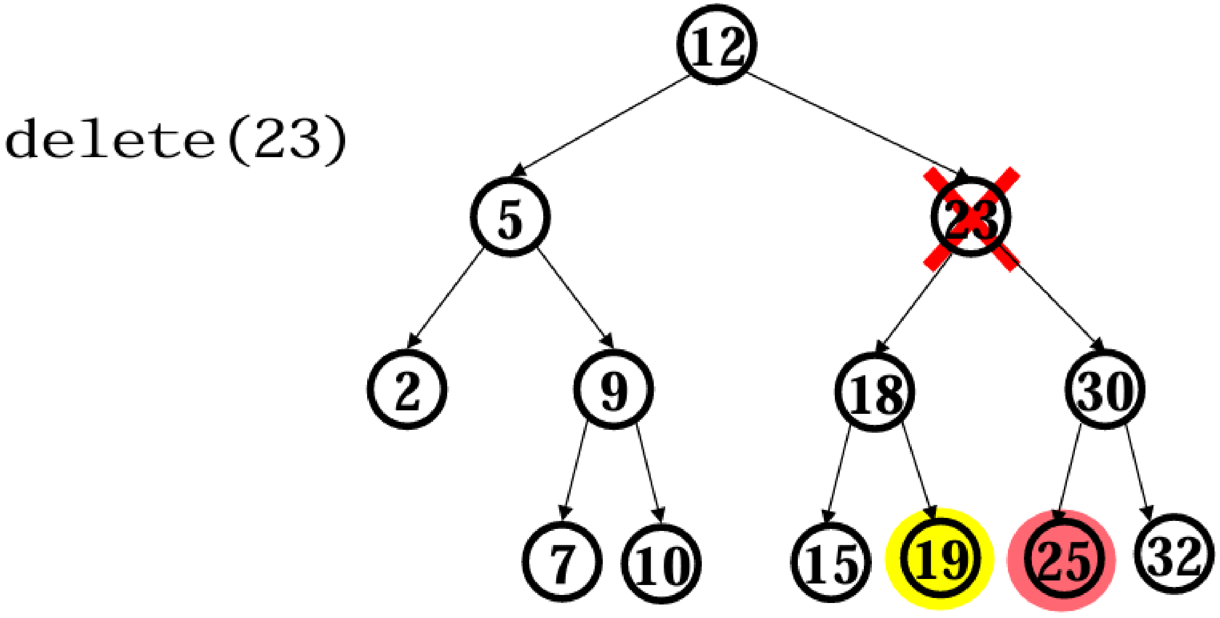
Deletion: Node w/ Two Children

- Idea: Replace the deleted node with a key whose value is guaranteed to be between the two child subtrees
- Option #1: Min node from right subtree (successor) (findMin(root.right))
- Option #2: Max node from left subtree (predecessor) (findMax(root.left))

- Delete the original node containing successor or predecessor

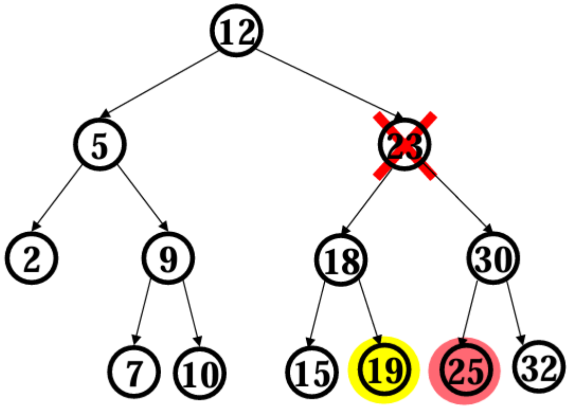
Deletion: Node w/ Two Children

- Delete the original node containing successor or predecessor

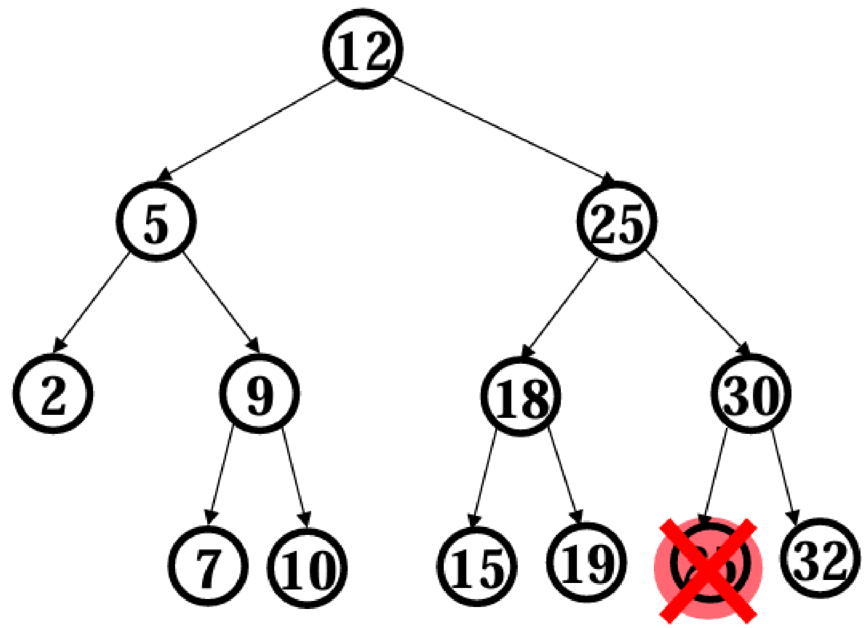


Deletion: Node w/ Two Children

delete(23)



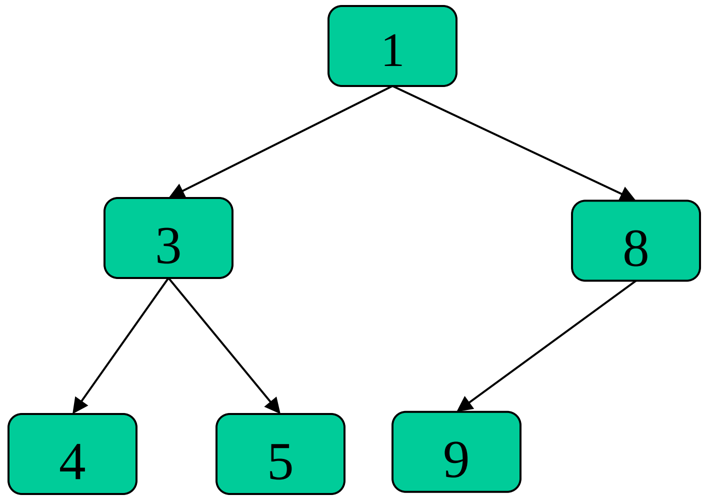
delete(23)



Binary Heaps/Priority Queues

- A data structure for storing elements associated with priorities (often called keys)
- Optimized to find the element that currently has the smallest key
- Operations
  - enqueue(k, v): adds element v to the queue with key k
  - isEmpty(): returns whether the PQ is empty
  - dequeue(): removes the element with the least priority from the queue

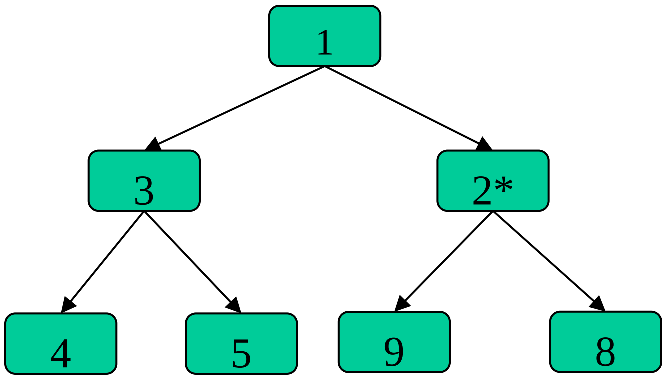
PQ Implementation

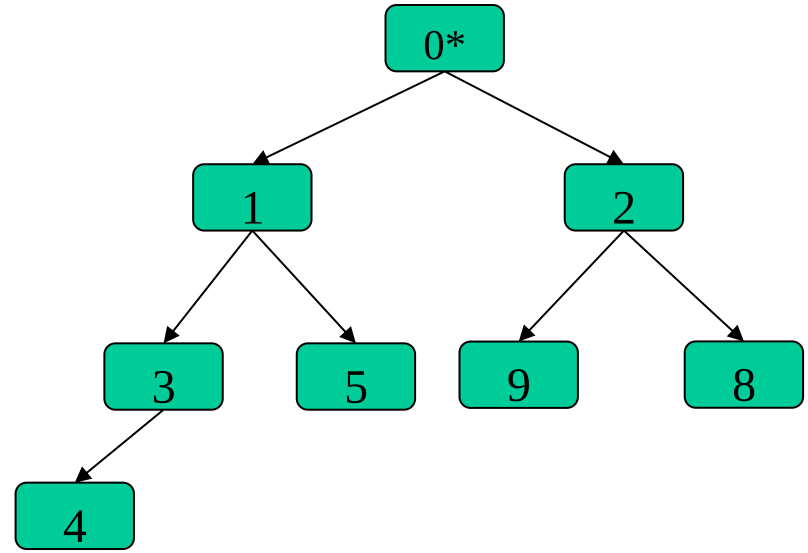


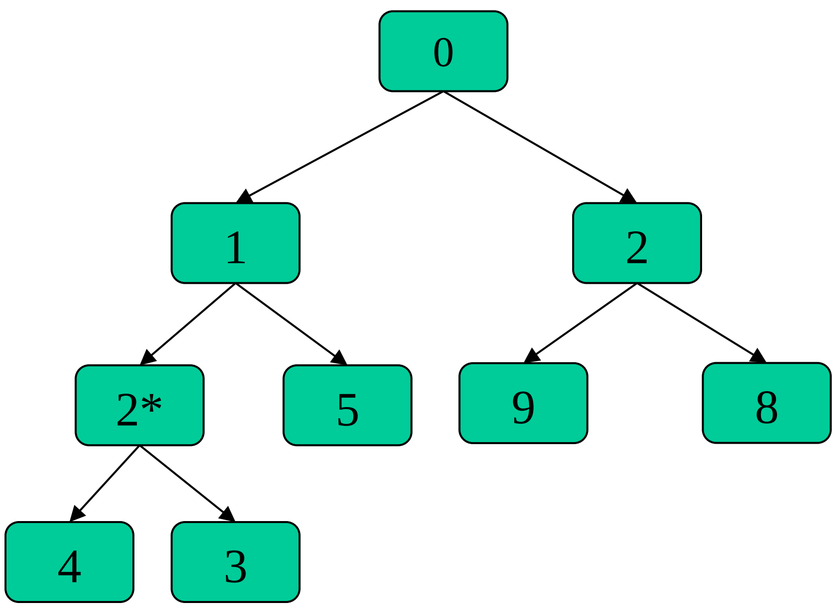
This tree obeys the heap property:

- Each node's key is less than or equal to all of its descendant's keys
- Also, a complete binary tree, i.e., all levels except the last one is filled in completely

PQ: Enqueue()







PQ Efficiency

- Enqueue() and dequeue() operations run in $O(h), h$ is the tree height
- For a perfect binary tree of height h, there are 1 + 2 + 4 + 8 + … + $2^h = 2^{h+1} - 1$ nodes
- If there are n nodes, the max height is $n = 2^{h+1} - 1 \Rightarrow h = \log_2(n+1) - 1$
- Thus, $h = \Theta(\log n)$, so enqueue() and dequeue() take $\mathrm{O}(\log n)$
- Most implementations assume one-indexing:
  - Root is at floor(n/2), children are at 2n, 2n+1

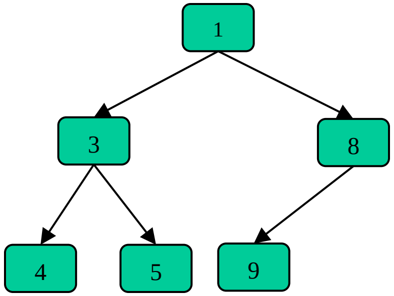
Heapsort

- Input: Unsorted array
- Idea:
  - Build a max-heap from array elements
  - Repeatedly dequeue() from the heap using heapify()
  - Place deleted element in the sorted list
  - Repeat until all elements are placed in sorted order/heap is empty
- Runtime: $O(n \log n)$ (at most n enqueues and dequeues)
- Space: O(1) vs. O(n) in mergesort

(Max)Heapify()

- Converts a binary tree into a heap datastructure by using recursion
- Output: Largest among root and children
- Procedure:
  - Get the left and right children of the (subtree's) root
  - If the left child's value is greater than root's value
    - Swap the left and root's values
    - Heapify(left subtree)
  - If the right child's value is greater than the root's value
    - Swap the right and root's values
    - Heapify(right subtree)

Max Heapify



Graph Problem #1

There are two types of professional wrestlers: “faces” and “heels”. Between any pair of these wrestlers there may or may not be rivalry.

Given names of n wrestlers and a list of r pairs of them who have rivalries, give a linear time algorithm to determine if its possible to designate some wrestlers as faces and some as heels such that each rivalry is between face and heel.

Idea

Map problem to a graph, n nodes and r edges

Run BFS on all vertices and label 0, 1 for faces and heels.

Check if sum of labels on edge = 1 if face-heels rivalry possible

Graph Problem #2

You are given a DAG G = (V, E). A Hamiltonian Path visits every vertex in the graph exactly once, starting from a source vertex.

Design an efficient algorithm to determine if a DAG contains a Hamiltonian Path.

Idea

Idea 1: Every DAG as a topological ordering that can be found in linear time.

Idea 2: A DAG with Hamiltonian Path will have a unique topological ordering

Topological sort G

Starting from the source vertex, follow all the tree edges. If all vertices are visited, then it is a Hamiltonian Path.

Lecture 7 summary

- Binary trees and BSTs
- Binary heaps and heapsort
- Midterm review