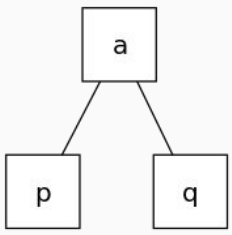
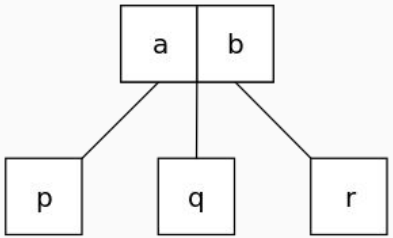
2-3-4 Trees and Heaps

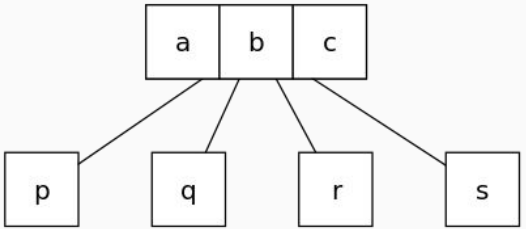
CMSC 132 - Week 7

2-3-4 Trees

- A self-balancing data structure in tree shape.
  - In general, they can search, insert, or delete in O(log n) time.
- The numbers mean a tree where every node with children (internal node) has either two, three, or four child nodes:
  - a 2-node has one data element, and if internal has two child nodes;
  - a 3-node has two data elements, and if internal has three child nodes;
  - a 4-node has three data elements, and if internal has four child nodes.

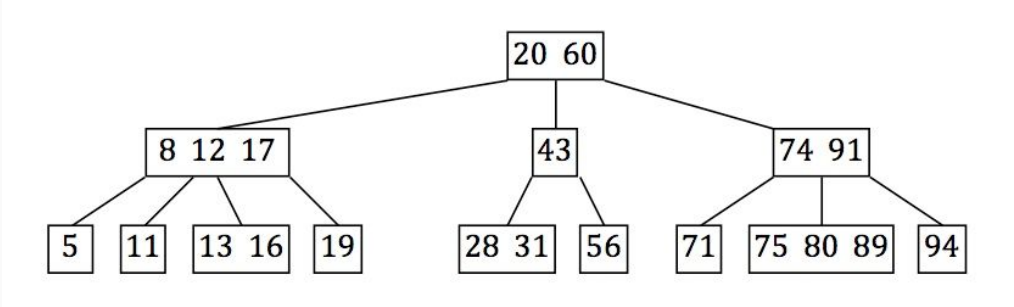






2-3-4 Trees

- Properties:
  - Every node (leaf or internal) is a 2-node, 3-node or a 4-node, and holds one, two, or three data elements, respectively.
  - All leaves are at the the bottom level.
  - Each child node corresponds to its parent interval.
- Example:



2-3-4 Tree Insertion

1. If the current node is a 4-node:
    a. Remove and save the middle value to get a 3-node.
    b. Split the remaining 3-node up into a pair of 2-nodes (the now missing middle value is handled in the next step).
    c. If this is the root node (which thus has no parent): the middle value becomes the new root 2-node and the tree height increases by 1. Ascend into the root.
    d. Otherwise, push the middle value up into the parent node. Ascend into the parent node.
2. Find the child whose interval contains the value to be inserted.
3. If that child is a leaf, insert the value into the child node and finish.
    - Otherwise, descend into the child and repeat from step 1.

2-3-4 Tree Insertion

- Normally, there are three cases you should consider:
  - First, the leaf node to which we want to add is not full and we can insert new element in that node easily.
  - Second, the leaf node to which we want to add is full. Then we split the leaf node and push the leaf key up a level.
  - Third, the parent does not have room for the split key. Then we must split the parent before the child.

2-3-4 Tree Insertion Trick

- When adding a key to a 2-3-4 tree, we traverse from the root to the leaf where we insert the key. At each node that we visit, we split the node if it has three keys. This guarantees we can always push a key into a node's parent because we visited its parent rst. In particular, we will always be able to split a leaf to make room for a new key.

2-3-4 Tree Deletion

Deleting an element in a leaf node

1. Case 1: the node has more than one element. Then, it can be removed easily.
2. Case 2: the node has only one element but one of its immediate sibling nodes has more than one element. Then rotate one of those elements in the sibling node and removed the one that needs to be deleted.
3. Case 3: the node has only one element but there is no immediate sibling that has more than one element. Then steal an element from its parent (if the parent has more than one element), merge itself with the sibling sharing the same parent element and remove the one that needs to be deleted.
4. Case 4: the node has only one element but there is no immediate sibling that has more than one element and the parent has only one element as well. Then attempt to merge the parent with its siblings and grandparent. If the total elements is three, then it should be the new parent. If not, we must do rotation on its parent and delete as the other cases above.

2-3-4 Tree Deletion Trick

- Now that we are deleting, we ensure that every node that we visit has at least two keys. But we have a strong preference: first we attempt to borrow from a sibling (left and then right), and if that fails, we steal from a parent.

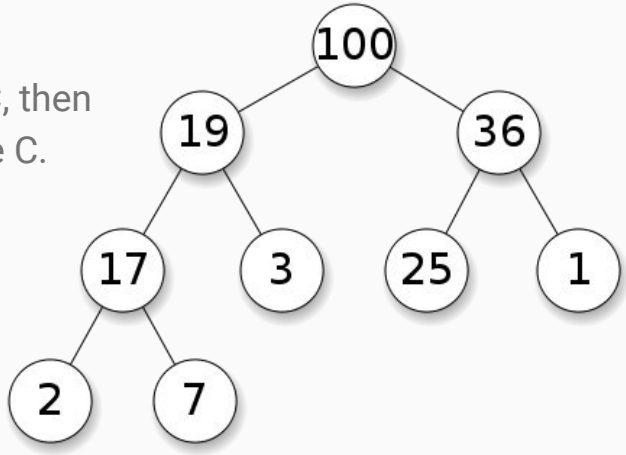
2-3-4 Tree Deletion

Deleting an element from an internal node

1. We remove its predecessor, and then swap keys.
2. Notice that the key we wish to delete might move when we delete it predecessor

Binary Heaps

- Tree-based data structure: balanced binary tree.
  - This means that the height of a tree with N elements is log(N).
- The tree needs to:
  - be balanced.
  - satisfy the heap property: if node P is the parent of node C, then the key of node P needs to be greater than the key of node C.
- A heap is NOT a BST!
- Side note: “min heap” vs “max heap”



Priority Queue (PQ) Interface

```
public interface PriorityQueue <T extends Comparable<T> > {
        void insert(T t);
        void remove() throws EmptyQueueException;
        T top() throws EmptyQueueException;
        boolean empty();
}
```

Maintaining the Balance of the Tree

- Always add / remove at the last position of the tree.
- This guarantees that we won’t start a new level in the tree, unless the last level is already full.
- Similarly, we won’t remove from a node from a level, unless this level is the last one.

Maintaining the Heap Property

- The tree structure changes with either the insert() or remove() operations.
- Inserting/removing a node to/from the tree can cause the heap property to be violated.
- We need to restore the heap property after each insertion/removal.

insert(T t)

- Insert (T ***t***):
  - Insert ***t*** at the last position in the binary tree (to keep it balanced).
  - Heap property can be violated only at ***t***.
  - To restore the heap property:
    - start at ***t***
    - compare it with its parent
    - if ***t*** is > its parent: swap them.
    - repeat until the heap property is restored.
  - At worst case, ***t*** will be moved up all the way to the root. Cost = ?

remove() / extractMax()

- extractMax():
  - By the heap property, the maximum value is at the root.
  - Let r be the root, ***x*** be the last node in the tree: wap(***r***, ***x***)
  - Remove ***r*** (after the swap, ***r*** is a leaf node, so removing it is straightforward)
  - Now, the heap property can only be violated at ***x***.
  - To restore the heap property:
    - start at ***x*** (the new root now)
    - let ***c*** be the greater of its 2 children.
    - if (***x*** < ***c***): swap(***x***, ***c***)
    - repeat until the heap property is restored.
  - In the worst case, ***x*** will be moved all the way down to be a leaf node. Cost = ?

Heap simulation

https://visualgo.net/en/heap

PQ: Cost of different implementations

Implementation	Insert	Remove Max	Max
Unordered Array	1	N	N
Ordered Array	N	1	1
Linked List (unsorted)	1	N	N
Binary Heap	**Log N**	**Log N**	**1**

Code Demo

Heap implementation vs Unordered array.

Clicker Quiz

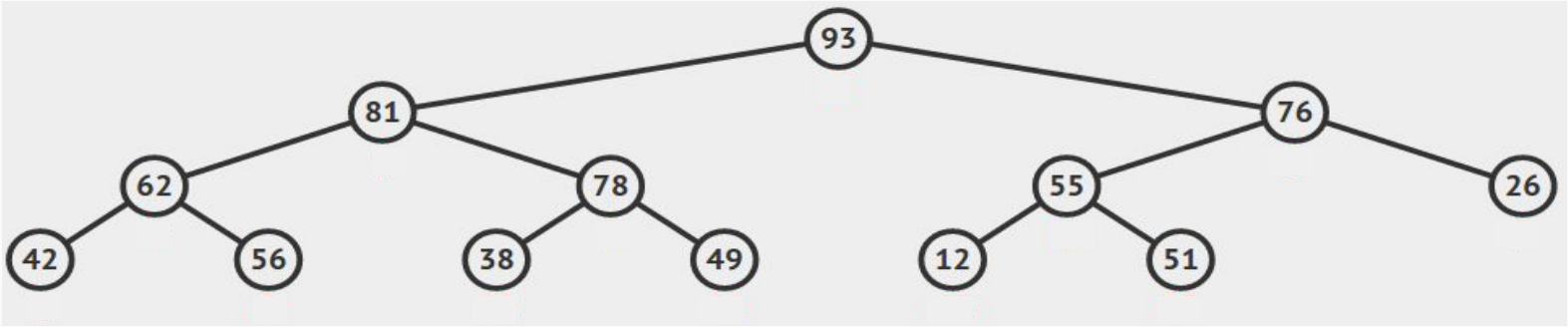
1) How many swaps will occur after inserting 80?

A. 0

B. 1

C. 2

D. 3



Clicker Quiz

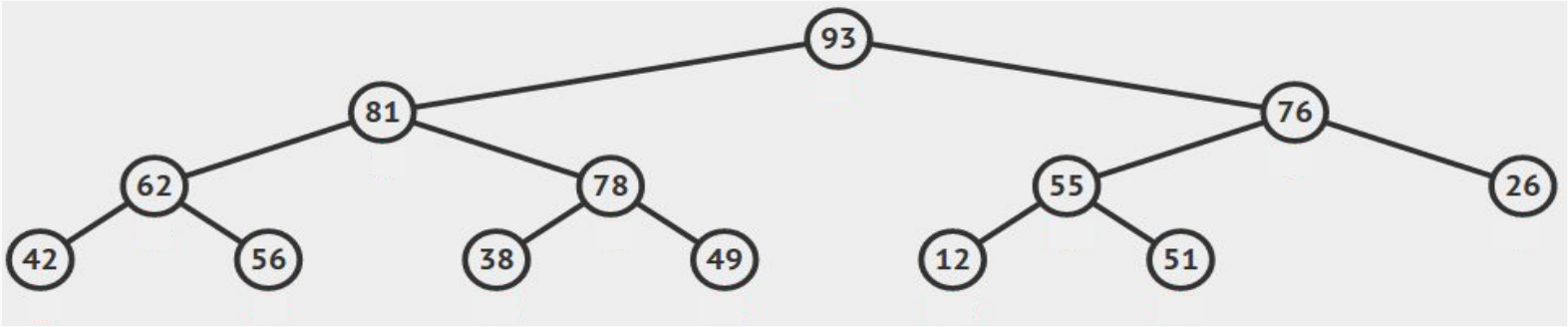
1) How many swaps will occur after inserting 80?

A. 0

B. 1

C. 2

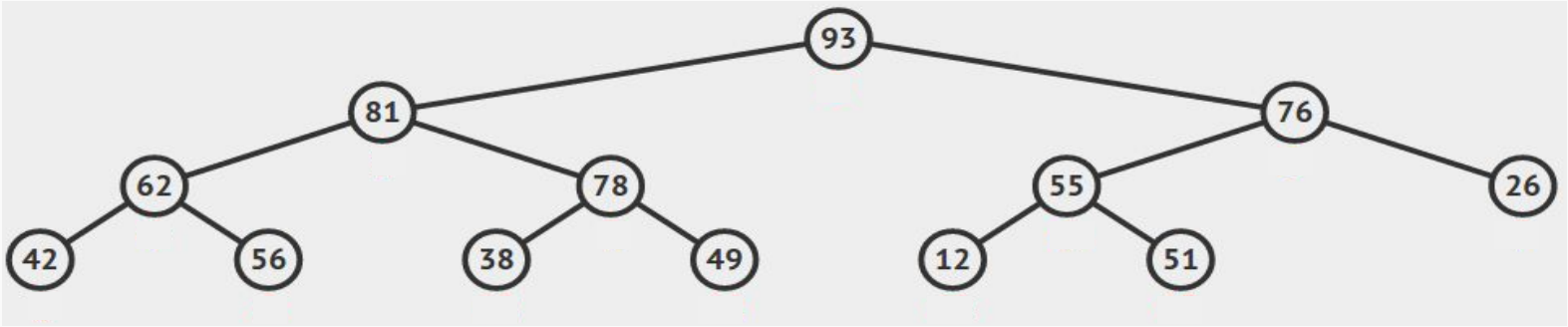
D. 3



Clicker Quiz

2) How many total swaps will occur after removing max (count ALL swaps)?

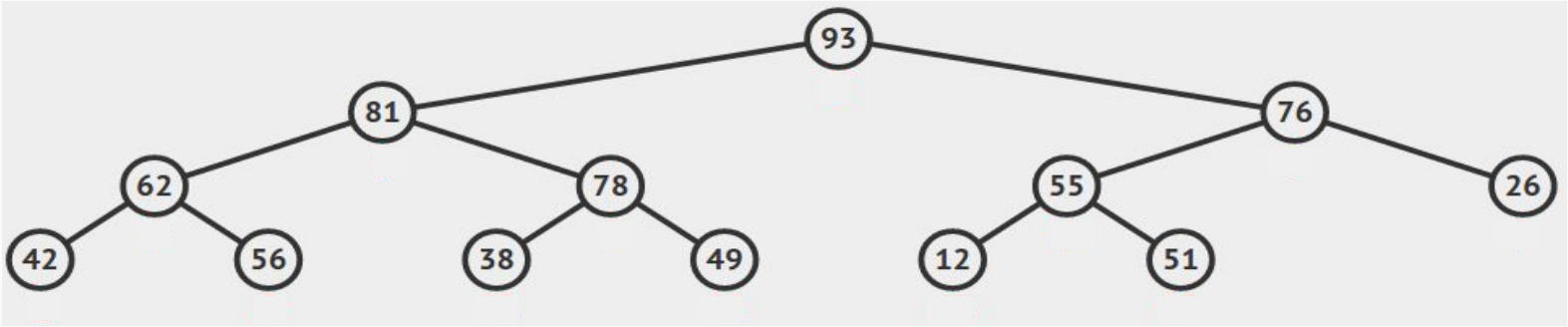
A. 0
B. 1
C. 2
D. 3



Clicker Quiz

2) How many **total** swaps will occur after removing max (count **ALL** swaps)?

A. 0
B. 1
C. 2
D. 3



Clicker Quiz

3) Suppose we have a heap implementation of priority queues and unordered linked list implementation. Which is more efficient for insertions?

A. Heap are more efficient
B. Unordered linked list are more efficient
C. Both have the same complexity (cost)

Clicker Quiz

3) Suppose we have a heap implementation of priority queues and unordered linked list implementation. Which is more efficient for insertions?

A. Heap are more efficient
B. Unordered linked list are more efficient
C. Both have the same complexity (cost)