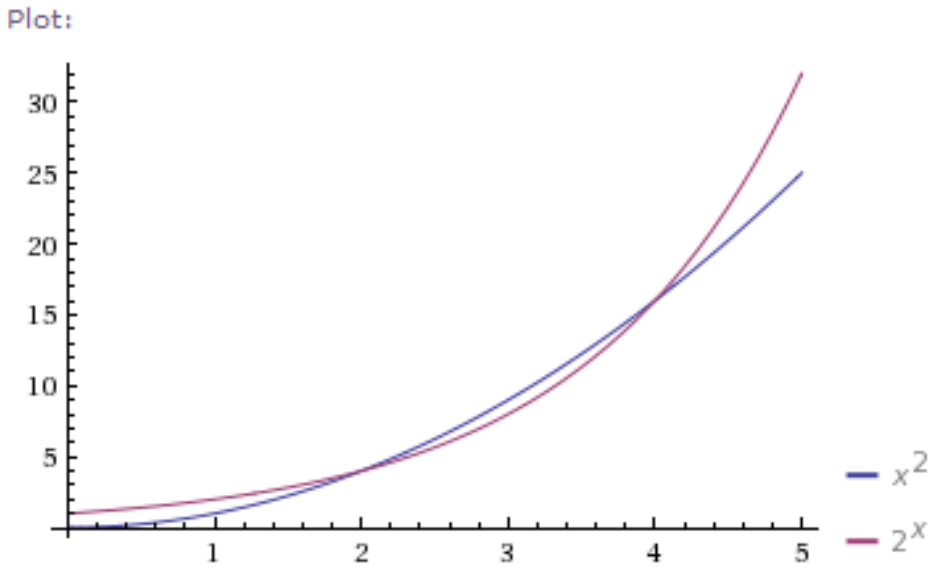
HKN CS61B Midterm 2 Review

Vaibhav Agarwal
Riyaz Faizullabhoy
Christopher Hall
Joey Moghadam
Preetum Nakkiran

Asymptotic Analysis

Intuition

- Compare complicated functions to simple ones in order to understand how they grow
- O is an upper bound
- Ω is a lower bound
- Θ is both



Definition

T(n) is in O(f(n)) iff T(n) ≤ c*f(n) for all n ≥ N where c and N are some positive constants

Asymptotic Analysis

Warm up: True/False

- sqrt($n^3 + n + 1$) is in Θ ($n^{3/2}$)
- n*log(n) is in O(n)
- $\log(n)^2$ is in O(n)
- 2^{2n} is in O(2^n)
- n! is in O(n^2)
- sin(n) + 1 is in O(1)

Formally prove the answer for sqrt($n^3 + n + 1$).

Asymptotic Analysis

Warm up: True/False

- sqrt($n^3 + n + 1$) is in Θ ($n^{3/2}$) True
- n*log(n) is in O(n) False
- $\log(n)^2$ is in O(n) True
- 2^{2n} is in O(2^n) False
- n! is in O(n^2) False
- sin(n) + 1 is in O(1) True

Formally prove the answer for sqrt($n^3 + n + 1$).

Algorithmic Analysis

- Suppose we want to understand how some property of our algorithm (such as run time, or space requirements) varies based on some variable (such as the input size).

- If we can write this relationship as a function, then we can understand it using asymptotic analysis.

Algorithmic Analysis

Below is the same ***pseudocode*** algorithm, one implemented with an **array** and the other with a **doubly linked list**, each of **length n**. What is the best case runtime of each? The worst case runtime?

```
Foo(array) {
     for (i = 0; i < array.length; i++) {
          for (j = i+1; j < array.length; j++) {
               if (list[i] == list[j]) {
                    array.remove(j); //after removing, items after j shift down an index
                    j--; //so we don't skip the item right after the removed one
}}}}

Foo(list) {
     for (i = list; i != Null; i = i.next) {
          for (j = i.next; j != Null; j = j.next) {
               if (i.contents == j.contents) {
                    j.removeNode(); //now j points to the removed node's previous
}}}}
```

Algorithmic Analysis

```
Foo(array) {
     for (i = 0; i < array.length; i++) {
          for (j = i+1; j < array.length; j++) {
               if (array[i] == array[j]) {
                    array.remove(j); //after removing, items after j shift down an index
                    j--; //so we don't skip the item right after the removed one
}}}}
```

Best case: $O(n^2)$
Worst case: $O(n^2)$

```
Foo(list) {
     for (i = list; i != Null; i = i.next) {
          for (j = i.next; j != Null; j = j.next) {
               if (i.contents == j.contents) {
                    j.removeNode(); //now j points to the removed node's previous
}}}}
```

Best case: $O(n)$
Worst case: $O(n^2)$

Stacks and Queues

- A stack is a collection with supports a LIFO policy
- A queue is a collection that supports a FIFO policy
- Stacks and queues are often implemented as linked lists
- Advantages of stacks and queues: constant time insertion/removal using linked lists
- Disadvantages: limited use

Stacks and Queues

Implement methods for a class which acts as a queue but can only have stacks as instance variables. A stack supports `pop()`, `push(Object o)`, and `isEmpty()`.

```
public void enqueue(Object o){
   //Your code here
}
public Object dequeue(){
   //Your code here
}
```

(Hint: try using more than one stack if you're stuck)

Stacks and Queues (solution 1)

```
private Stack<Object> inStack = new Stack<Object>();
private Stack<Object> outStack = new Stack<Object>();
public void enqueue(Object o){
    inStack.push(o);
}
public Object dequeue(){
    if (outStack.isEmpty()){
        while(!inStack.isEmpty()){
            outStack.push(inStack.pop());
        }
    }
    return outStack.pop();
}
```

Why is this an impractical way to implement a queue?

Stacks and Queues (solution 2)

```
private Stack<Object> s = new Stack<Object>();
public void enqueue(Object o){
    s.push(o);
}
public Object dequeue(){
    Object temp = s.pop();
    if (s.isEmpty()){
        return temp;
    } else {
        Object toReturn = s.dequeue();
        s.push(temp);
        return toReturn;
    }
}
```

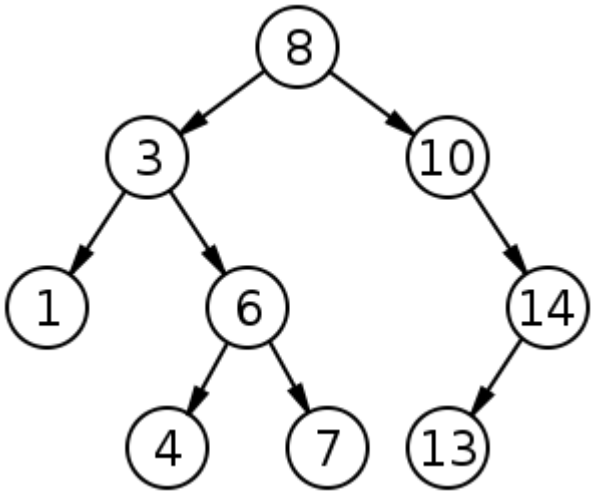
Why is this an impractical way to implement a queue?

Binary Search Trees

Binary tree with property:

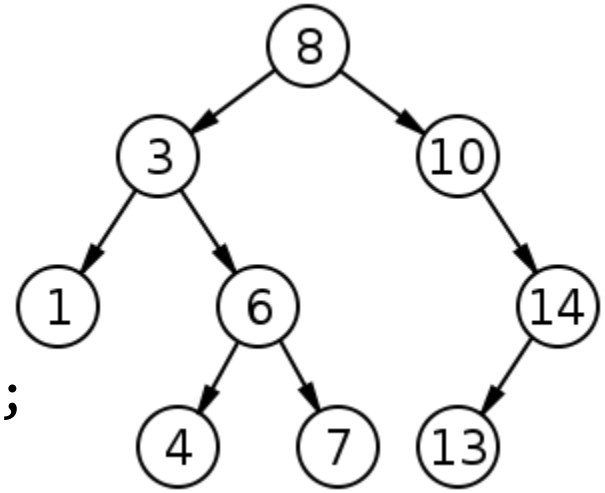
- All nodes in LEFT subtree < root
- All nodes in RIGHT subtree > root

in-order traversal gives sort



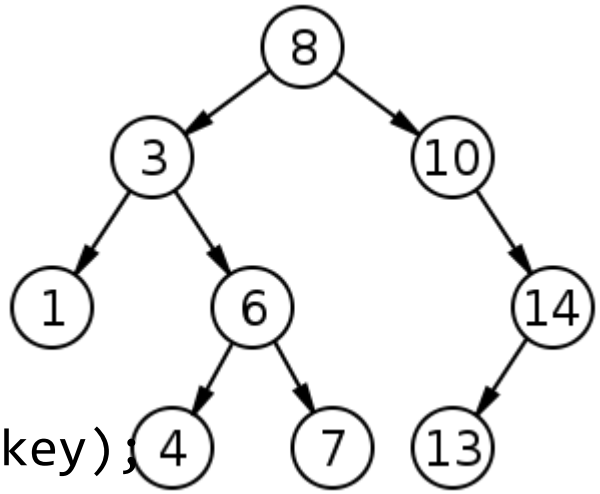
BST Find

```
bst_find(tree, key):
    if (tree == null)
        return false;

    if (key == tree.value)
        return true;

    if (key < tree.value)
        return bst_find(tree.left, key);
    else
        return bst_find(tree.right, key);
```

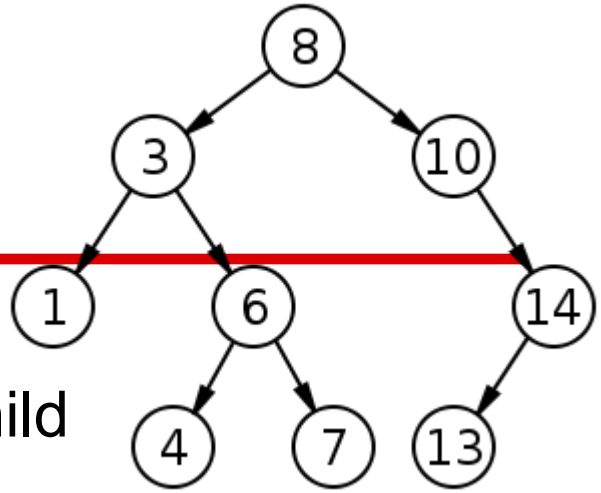


BST Insert (Unique Keys)

```
bst_insert(tree, key):
    if (tree == null)
        return new Tree(key);

    if (key == tree.value)
        return tree;

    if (key < tree.value)
        tree.left = bst_insert(tree.left, key);
    else
        tree.right = bst_insert(tree.right, key);

    return tree;
```



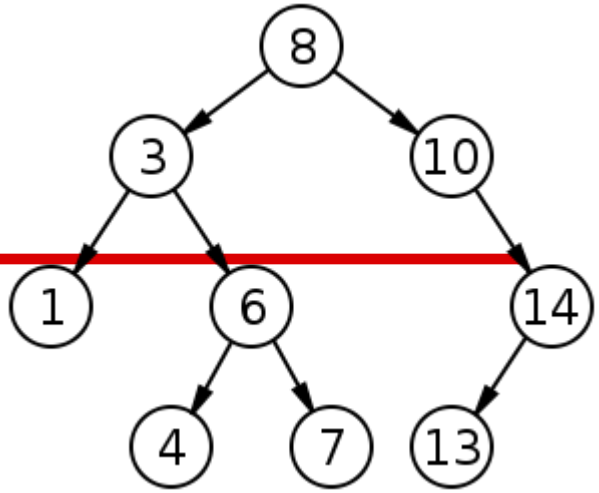
BST Remove?



- Removing leaf nodes is easy.
- Nodes with one child : 'promote' the child
- Internal nodes (2 children)
  - Find the 'successor' (min node greater than deleted node)
  - Replace deleted node with 'successor'
  - Remove successor
    - What about children?
    - Max 1 child -- promote it.

- Think about why above operations preserve BST property.

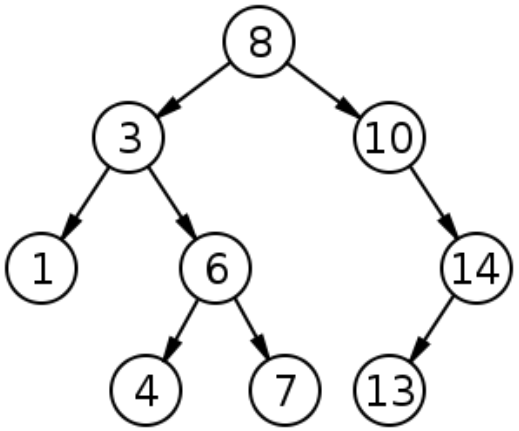
BST Successor




```
bst_next(node):
    if (node.right)
        return min(node.right);
    return first_right_parent(node);

first_right_parent(node):
    while (node.parent != null && node.parent.right == node) {
        node = node.parent;
    }
    return node.parent;

min(tree):
    //try this yourself
    //hint: are smaller nodes on the left or right?
```

Some BST Runtimes

Operation	Average Case	Worst Case
Find	O(log n)	O(n)
Insert	O(log n)	O(n)
Remove	O(log n)	O(n)
Construct	O(nlog n)	

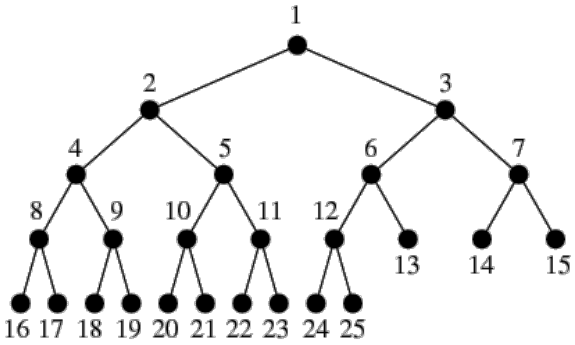


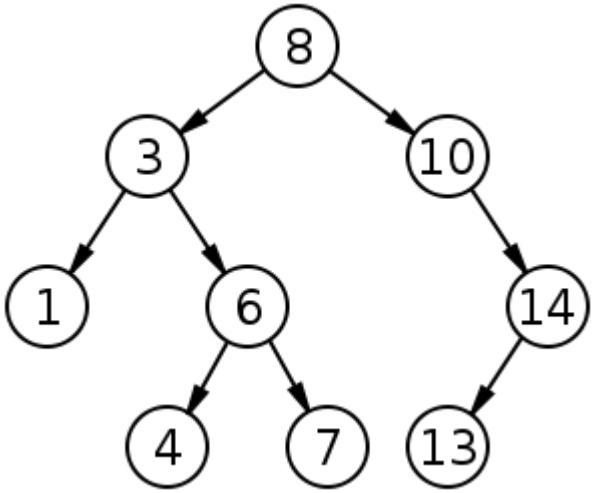
Everything is O (n) worst case!

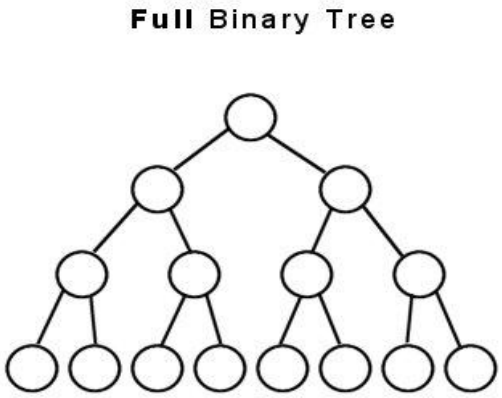
Why?

BST Quiz Questions

- Write functions for:
  - Finding the 2nd-least element
  - Checking if a tree is full.
  - Checking if a tree if complete.
  - Checking if a tree is a BST







Checking BST Property: First Try

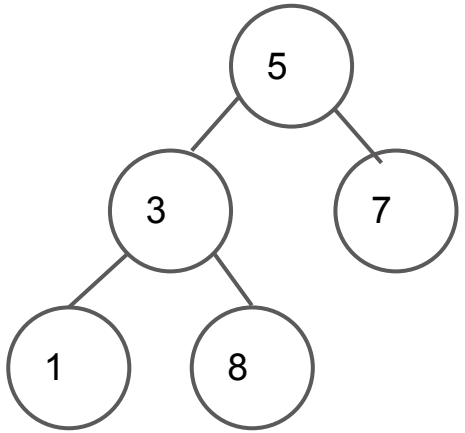
```
is_bst(tree) {
   if(!tree) return true;
   return tree.left.value < tree.value
      && tree.right.value > tree.value
      && is_bst(tree.left)
      && is_bst(tree.right)
}
```

What's wrong with this?

Checking BST Property: First Try

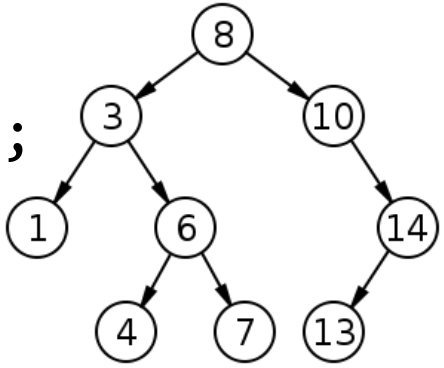
```
is_bst(tree) {
   if(!tree) return true;
   return tree.left.value < tree.value
      && tree.right.value > tree.value
      && is_bst(tree.left)
      && is_bst(tree.right)
}
```

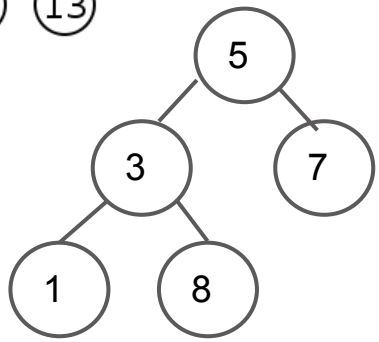
Fails for trees like:



Checking BST Property: Correct

```
is_bst(tree) {
    return is_bst(tree, -inf, +inf);
}
```

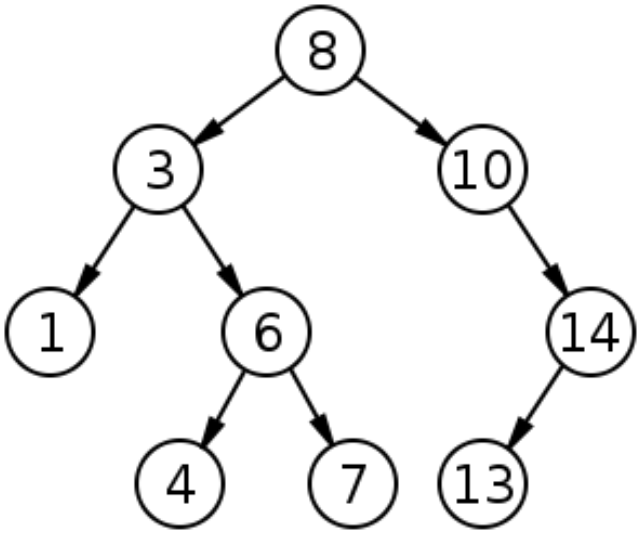





```
is_bst(tree, min, max) {
    if (!tree) return true;
    return      tree.value > min
           && tree.value < max
           && is_bst(tree.left, min, tree.value)
           && is_bst(tree.right, tree.value, max);
}
```

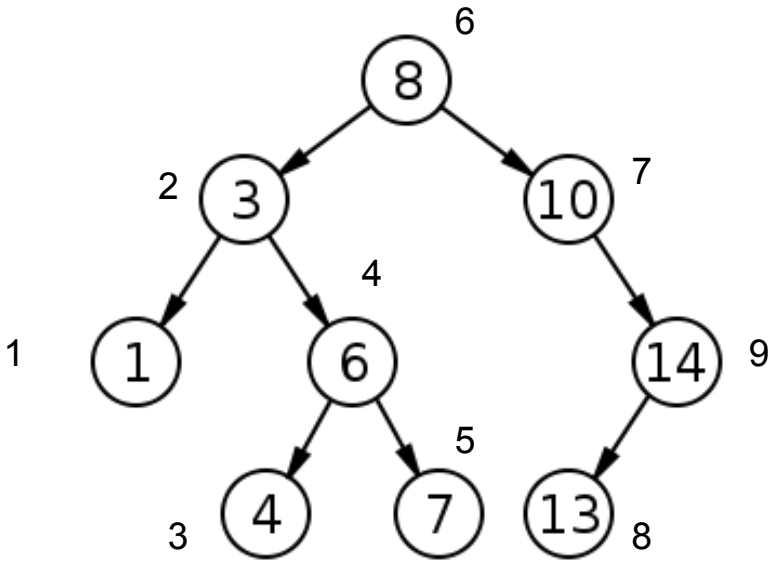
BST Intermediate Questions

- Given a BST annotated by its in-order traversal sequence, find the median element. (Best/Avg/Worst case?)



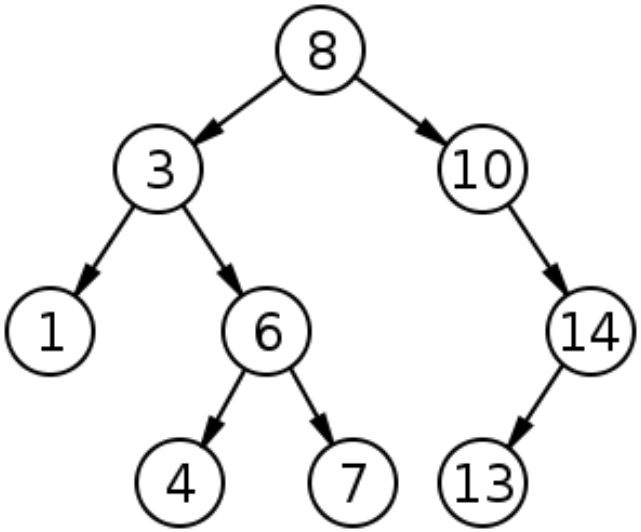
BST Intermediate Questions

- Given a BST annotated by its in-order traversal sequence, find the median element. (Best/Avg/Worst case?)
- Example:



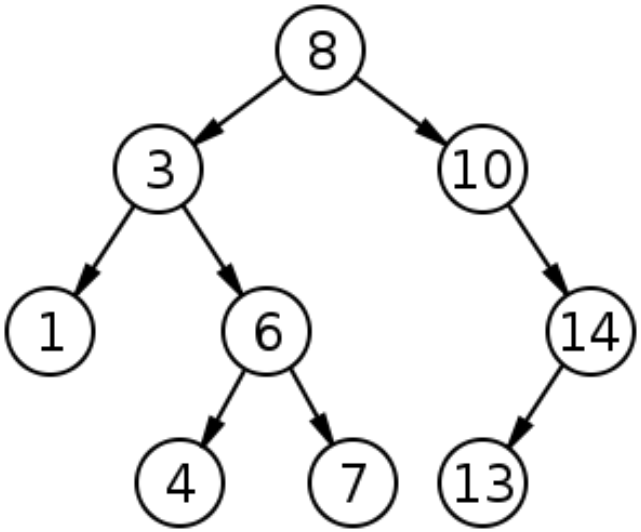
BST Intermediate Questions

- Reconstruct a BST from two of the three: {pre-order, post-order, in-order} traversals.
- Example:
  - In-order: 1,3,4,6,7,8,10,13,14
  - Post-order: 1,4,7,6,3,13,14,10,8



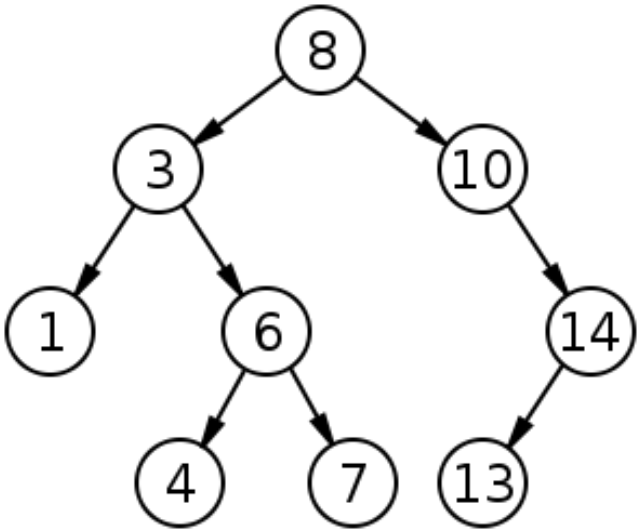
BST Intermediate Questions

- Reconstruct a BST from two of the three: {pre-order, post-order, in-order} traversals.
- Example:
  - In-order: 1,3,4,6,7,8,10,13,14
  - Post-order: 1,4,7,6,3,13,14,10,8



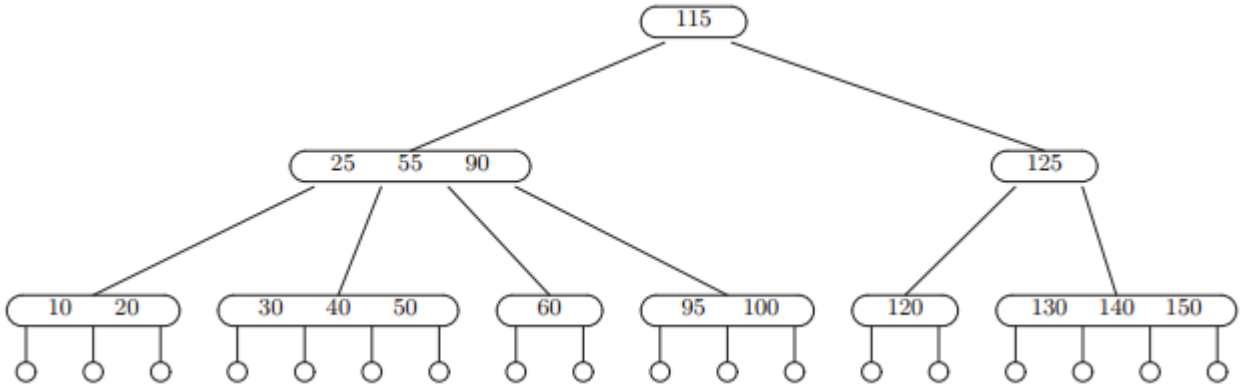
BST Intermediate Questions

- Reconstruct a BST from two of the three: {pre-order, post-order, in-order} traversals.
- Example:
  - In-order: 1,3,4,6,7,8,10,13,14
  - Post-order: 1,4,7,6,3,13,14,10,8



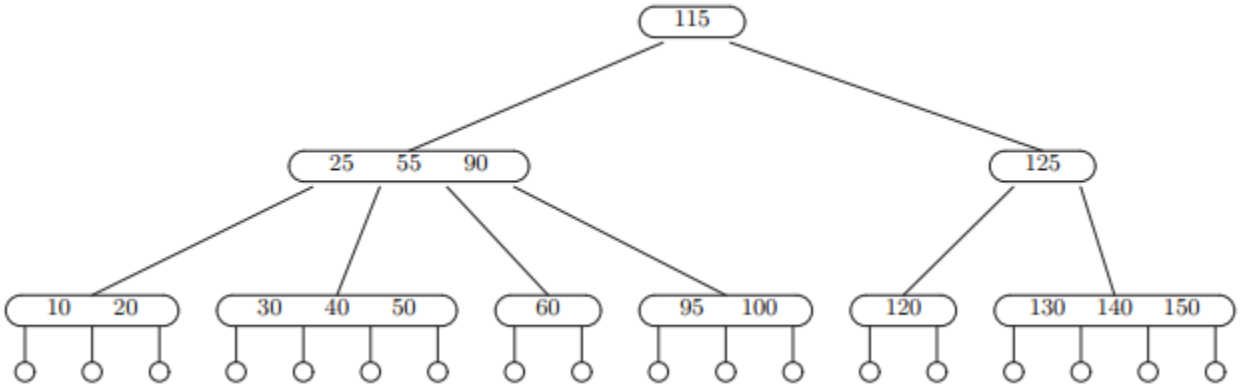
2-4 Trees (B-trees of order 4)

- Each node contains:
  - 2 to 4 children
  - 1 to 3 keys
- All empty children occur at same level ("balanced")
- Essentially a BST, with 1-3 keys per node



B-tree Find

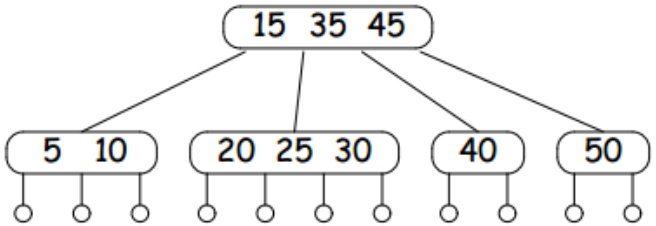
- Essentially same as BST-find.
- Only difference: > 1 comparison per node
  - How many, worst case?
- height = O(log n) always!



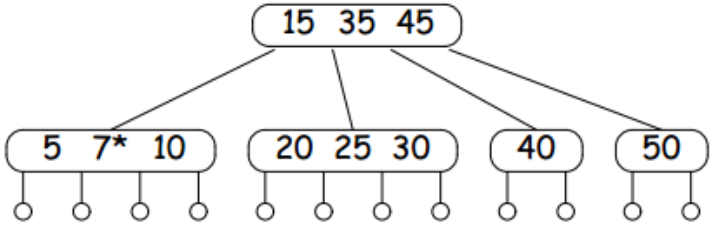
B-tree Insert (Simple case)

- Insert new key into existing node at bottom of tree

- Start:



- Insert 7:



B-tree Insert (Splitting)

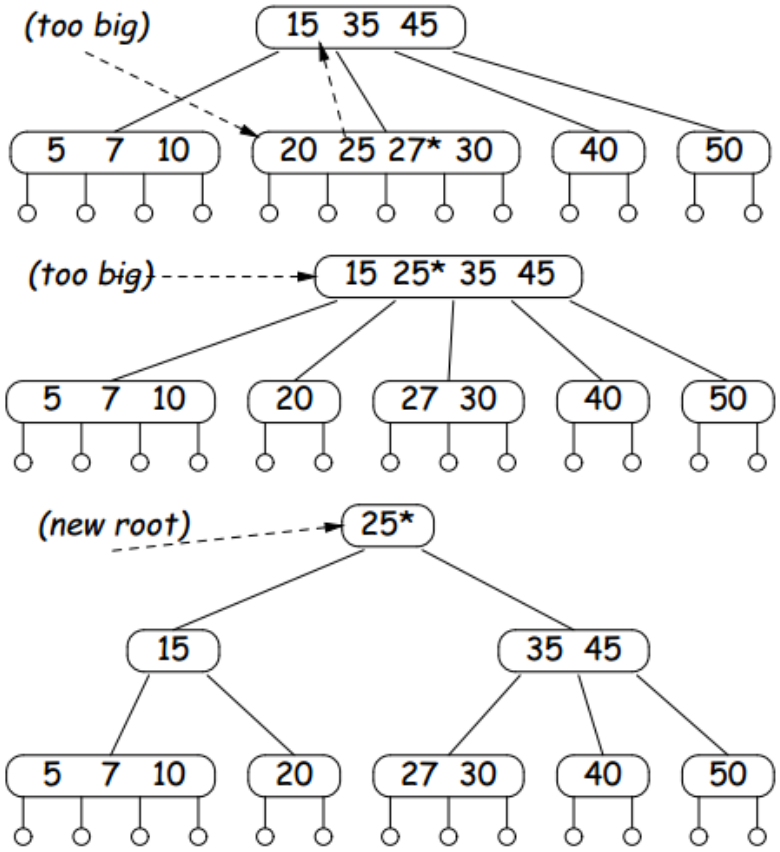
- Insert 27:

Split, and promote.

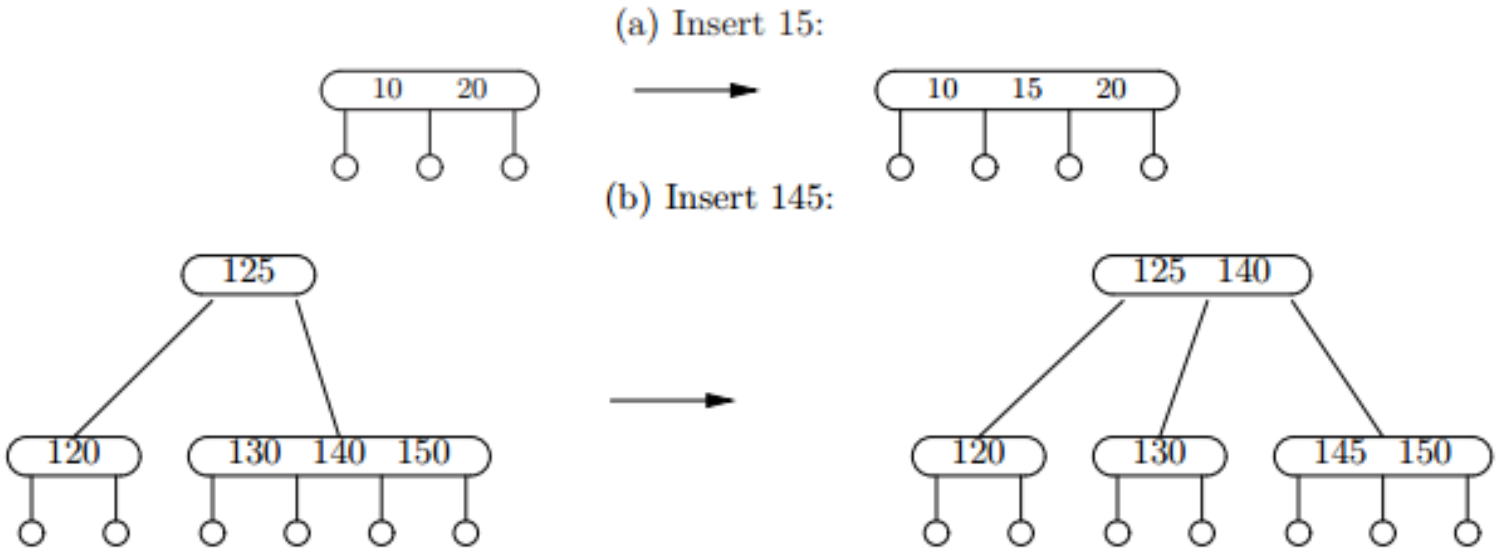
Q: How to decide which one to promote?

A: One of the middle ones. Why?

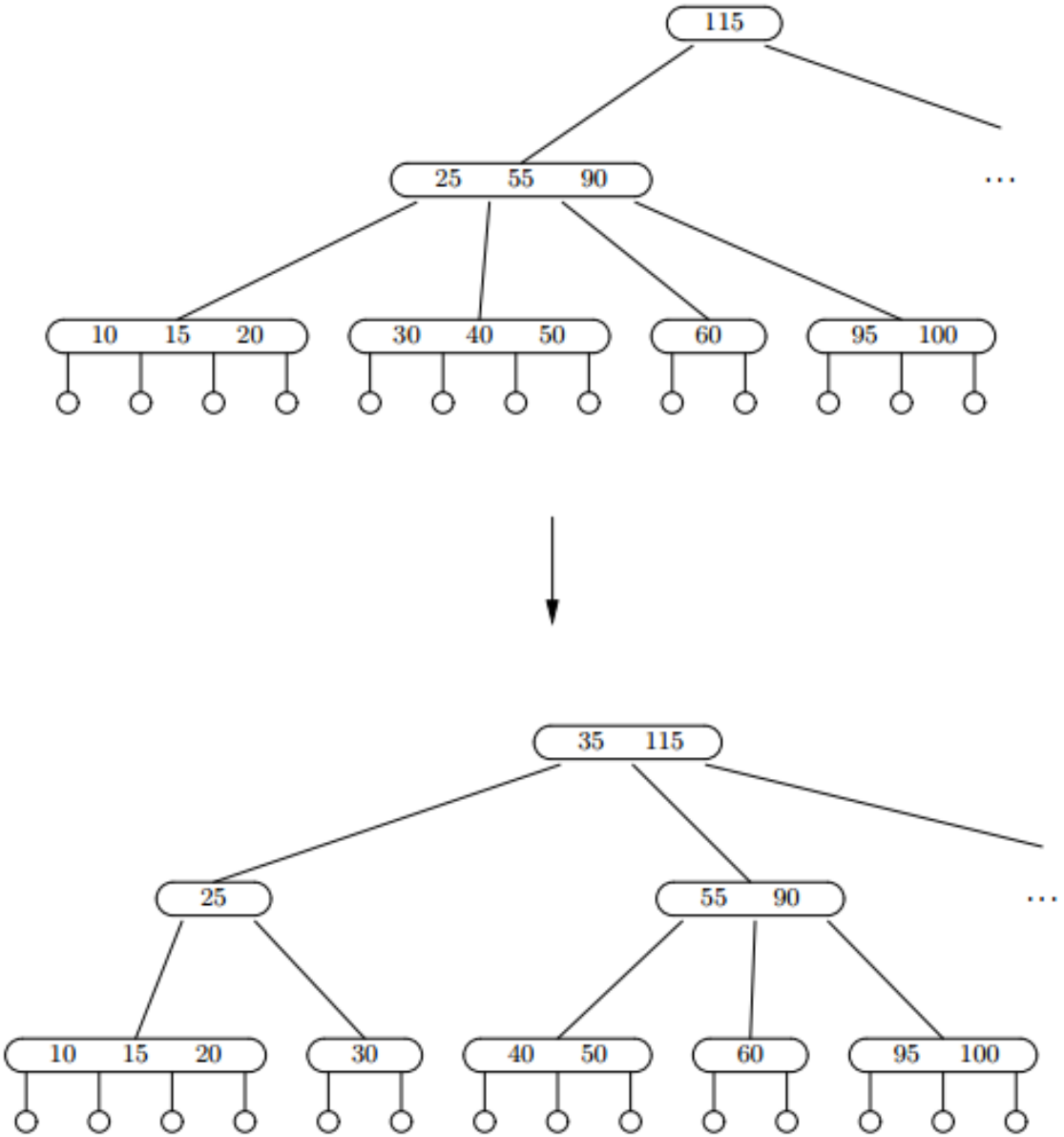
Q: Check that this preserves B-tree property



B-tree Insert (Additional Example)



(c) Insert 35:

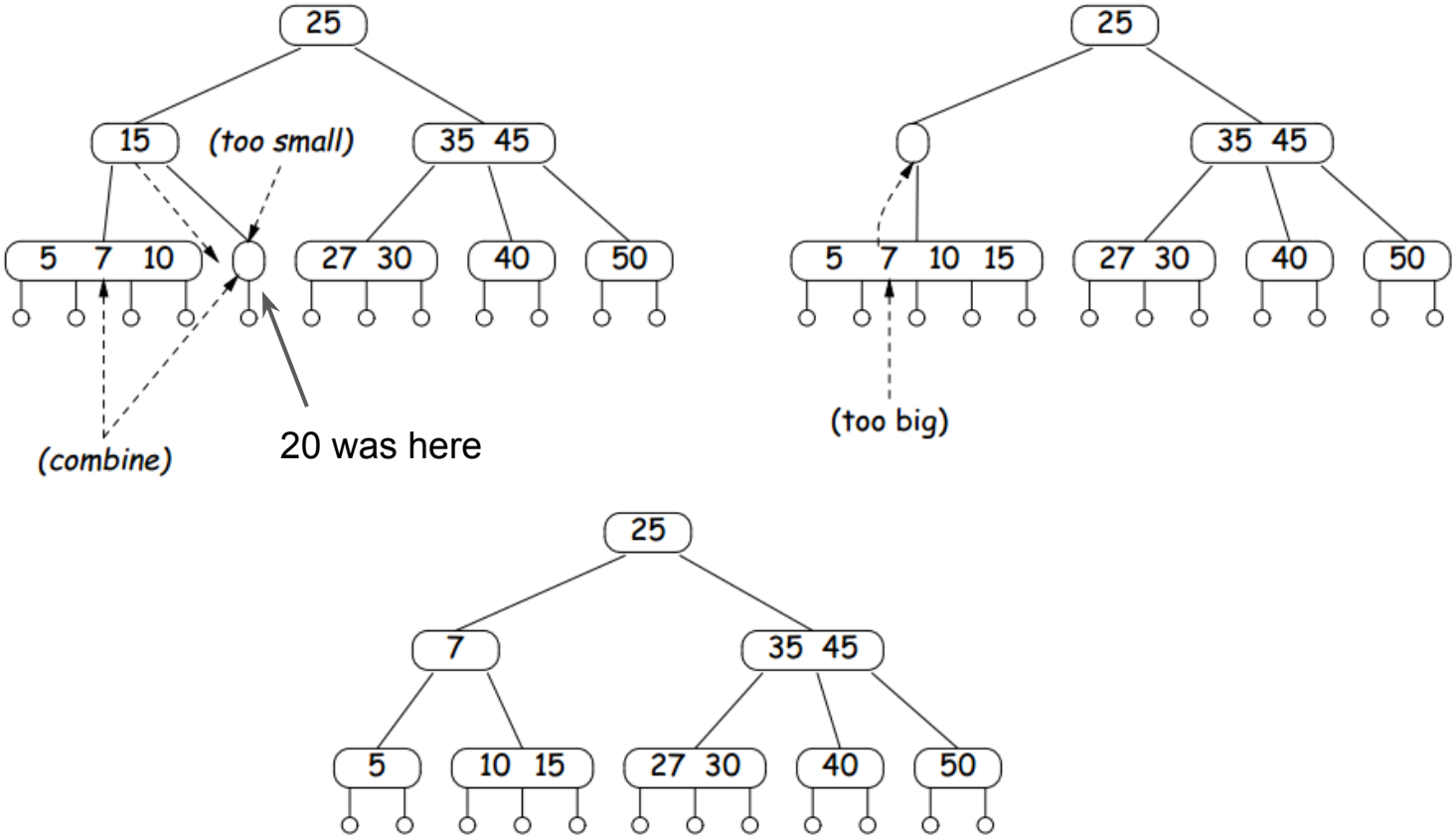

115
25 55 90
...
10 15 20
30 40 50
60
95 100
35 115
25
55 90
...
10 15 20
30
40 50
60
95 100

B-tree Delete

- Deleting from inner nodes complicated...
- Instead, **push down** target node to the bottom, where deletion is easy.
- Then, if nodes are too small, **merge** them with siblings, pulling down parent in between
- Or, if nodes are too big, promote one

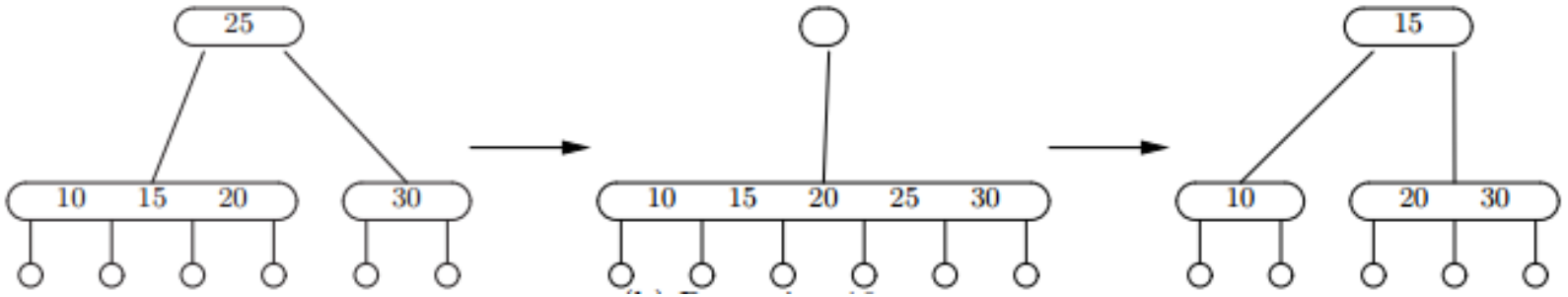
B-tree Delete (Ex 1)

- Remove 20 from last tree.



B-Tree Delete (Ex 2)

(a) Steps in removing 25:



Some B-tree Runtimes

Operation	Average Case	Worst Case
Find	O(log n)	O(log n)
Insert	O(log n)	O(log n)
Remove	O(log n)	O(log n)
Construct	O(nlog n)	O(nlog n)

Everything is O(log n)!

B-trees are *balanced search trees*

B-tree Algorithms

- Deal with abstractions of find/insert/delete rather than implementation

- Everything you learnt for BST applies here!

Applications

BST

- Searching
- Space partitioning

2-4 Tree

- [Distributed] Databases (B-tree)
- MT question: Can we make a hash-table with O(n logn) worst-case time?

Alpha Beta Pruning

Idea: make minimax faster!

α - best (highest) guaranteed score seen so far for a maximizer node.

-Starts at negative infinity

β - the best (lowest) guaranteed score seen so far for a minimizer node

-Starts at positive infinity

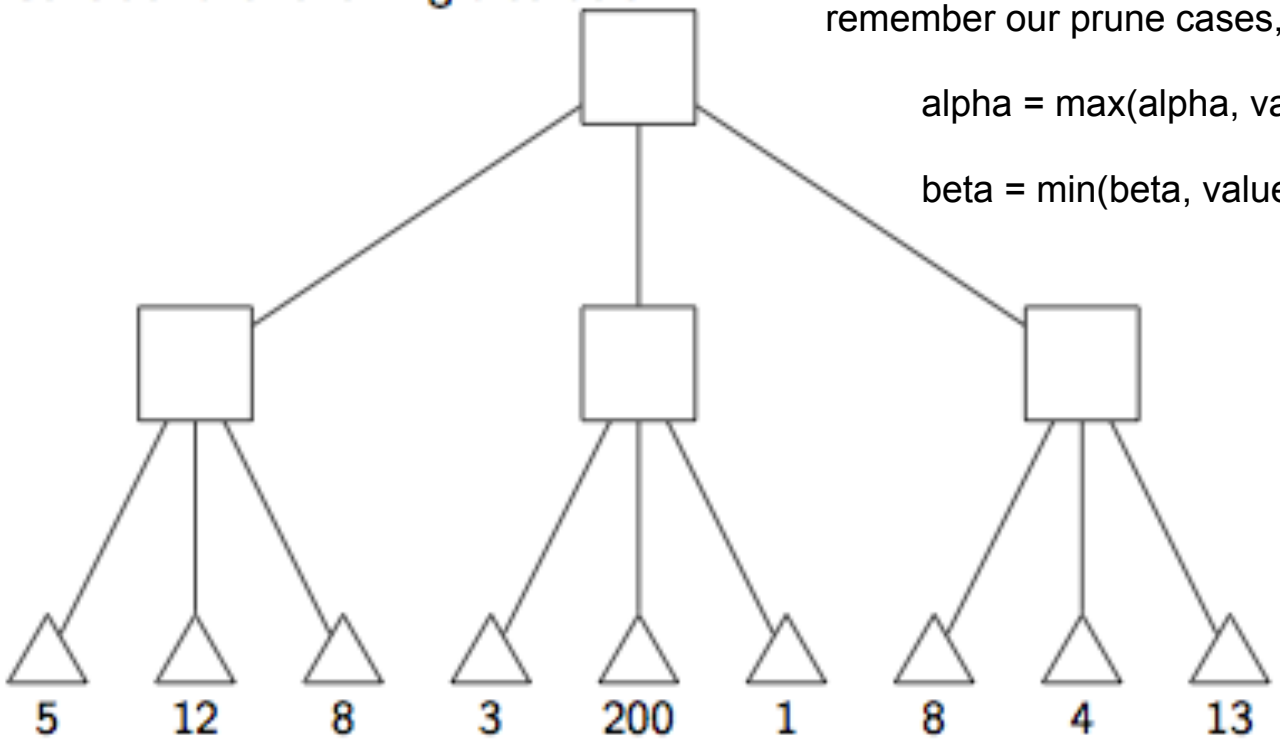
Prune Cases

- prune at a minimizer node whose β value is less than or equal to the α value of any of its maximizer node ancestors

- prune at a maximizer node whose α value is greater than or equal to the β value of any of its minimizer node ancestors

Alpha Beta Pruning

Consider the following tree below:



remember our prune cases, and:

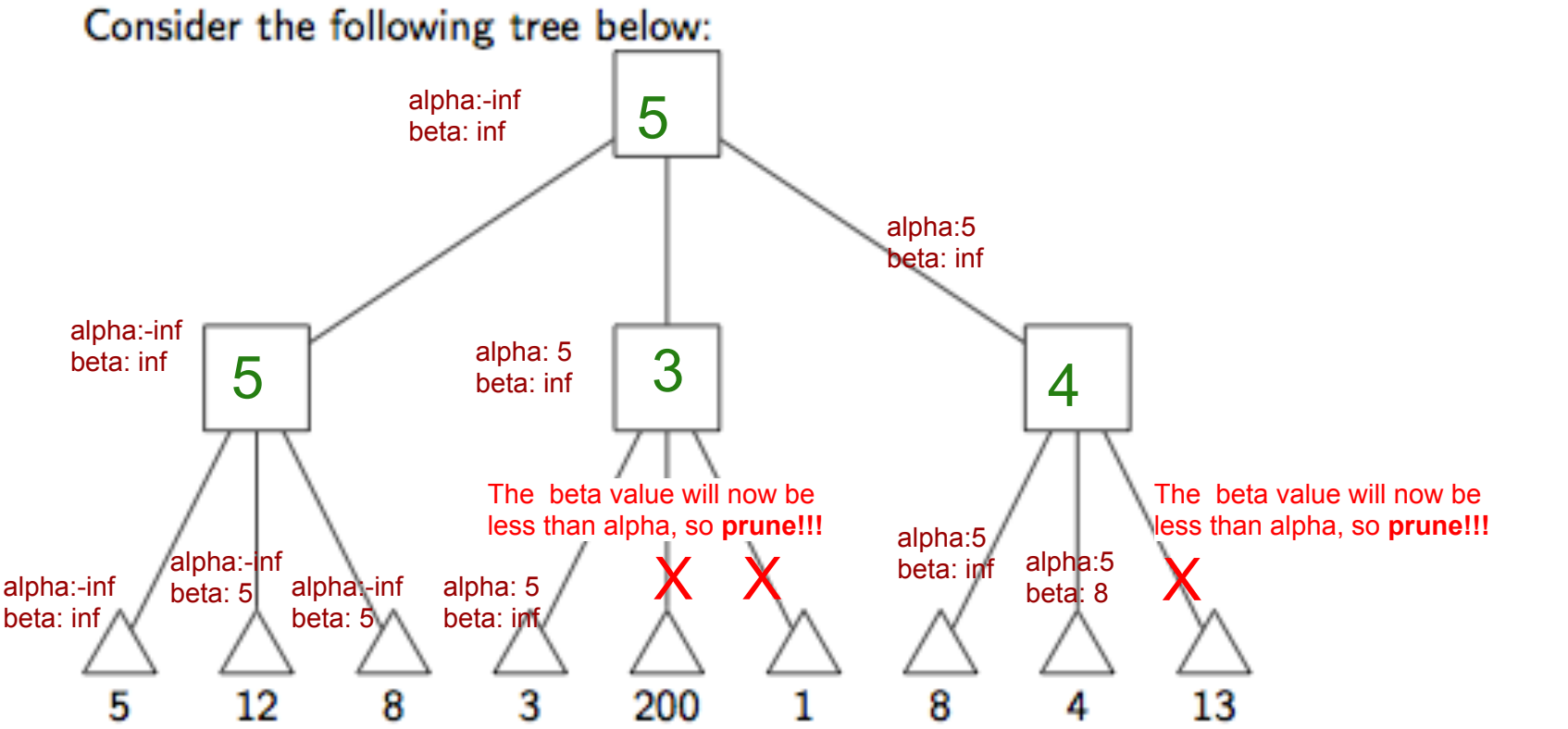
alpha = max(alpha, value_below_max_node)

beta = min(beta, value_below_min_node)

Determine which nodes will be pruned using α-β pruning and the final minimax value of the root. Assume the first player is MAX.

Also assume we traverse children from left to right.

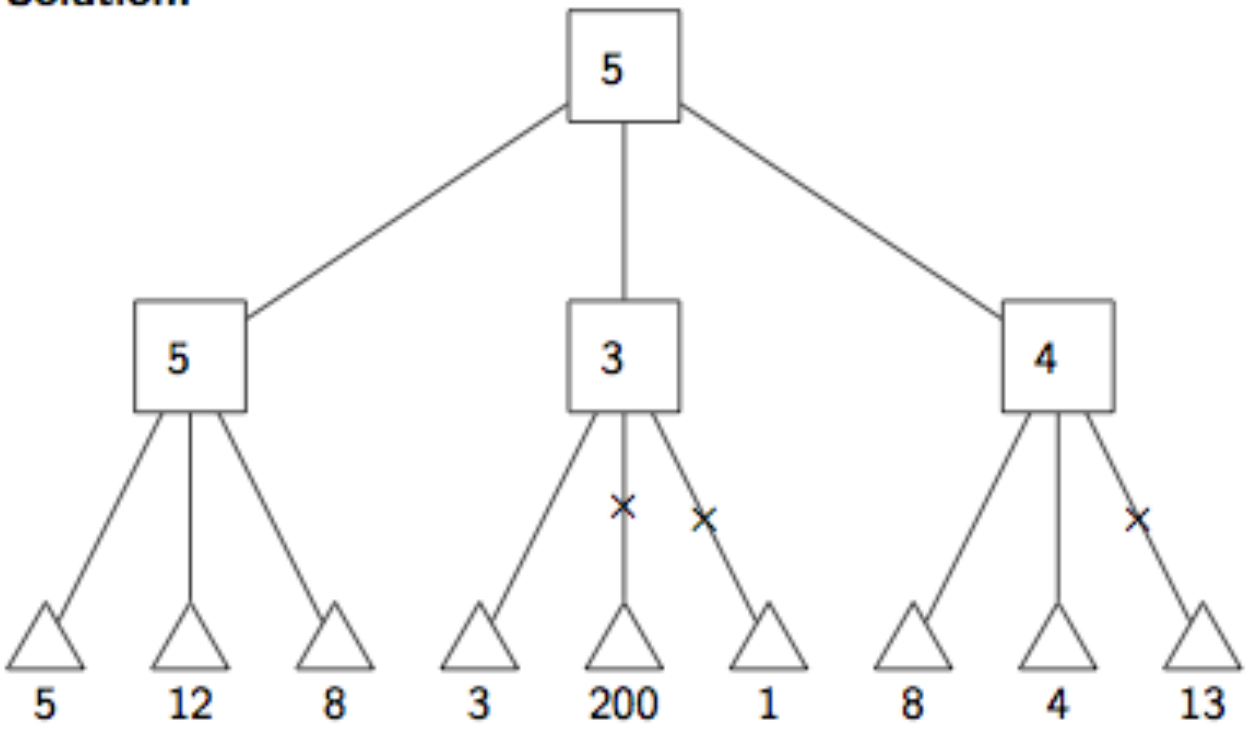
Alpha Beta Pruning



Also assume we traverse children from left to right.

Alpha Beta Pruning

Solution:



Alpha Beta Pruning

True or False:

1. After alpha-beta pruning, the root maximizer node will never have the wrong value.

True -- try doing minimax on our example

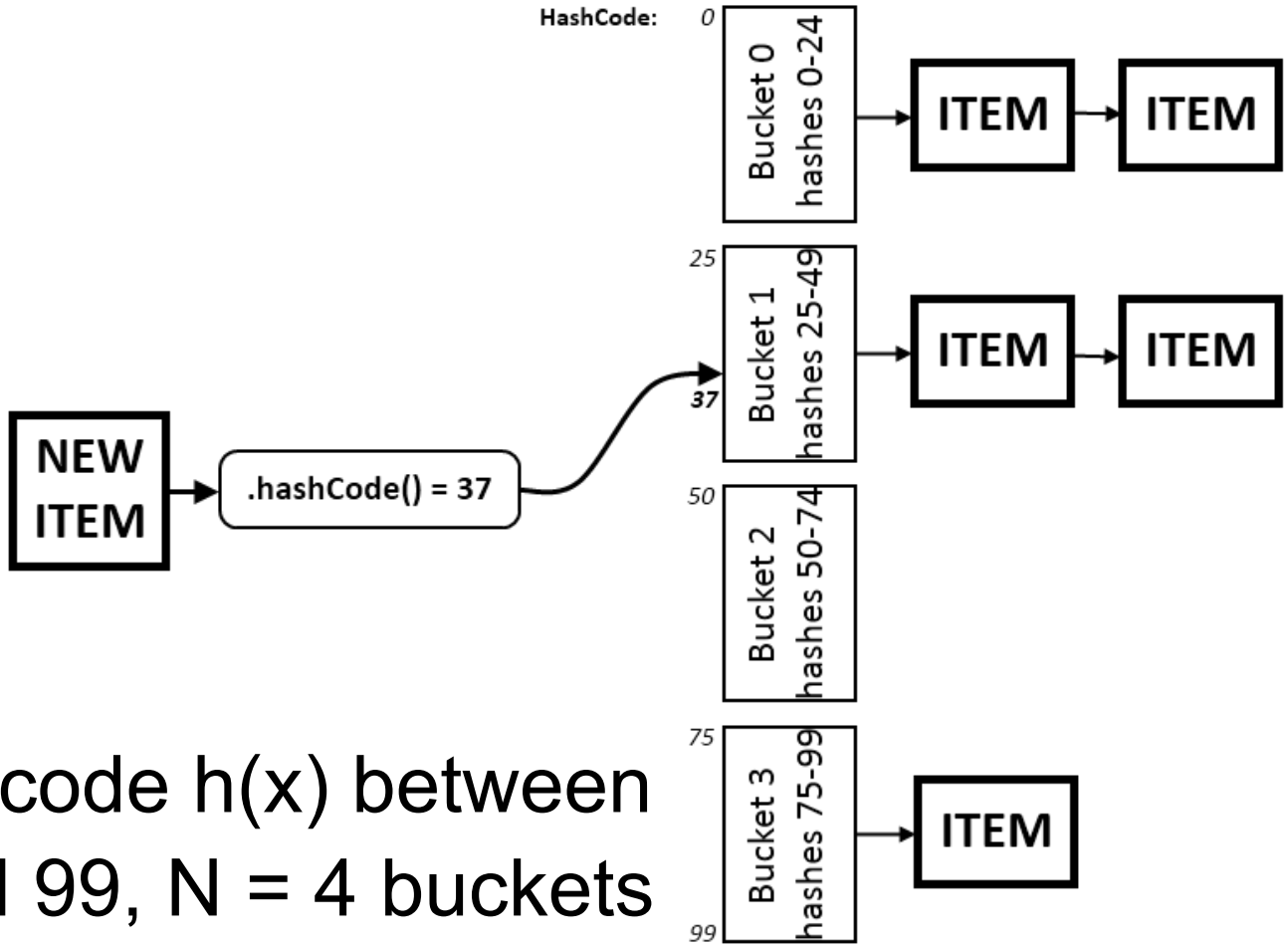
2. During alpha-beta pruning, none of the minimizer or maximizer intermediate nodes will differ from values found when using the normal minimax algorithm.

False -- try doing minimax on our example

3. Alpha-beta pruning can have different prunings based on the order in which the algorithm traverses the tree.

True -- what would have happened if we started with the middle minimizer's children first?

Basic Hash Table



Hashcode h(x) between 0 and 99, N = 4 buckets

Hashing overview

- hashCode() maps an **object** to a **code**
- A hash function maps a **code** to a **bin/bucket**
- When two objects map to the same bin, it is called a **collision**
  - collisions cause linked lists within the bins to grow, slowing down the structure
- A good hash function is *unlikely* to map several items in a set to the same bin
- When the **load factor** (# items / # bins) is less than 1, search time is O(1)
  - If load factor gets too high, double the number of

Awful hash function example

Q: Say you want to add a set of words to a hash table. Why is it a bad idea to simply assign each word to a bin based on its first letter?

Awful hash function example

Q: Say you want to add a set of words to a hash table. Why is it a bad idea to simply assign each word to a bin based on its first letter?

A: It is very easy for certain data to slow down the hash table. For example, if every word in the set starts with either 'a' or 'b', then all of the words end up in two bins, and access time becomes O(n) instead of O(1).

Important points

- Items in a hash table are **not** in order
- Bins in a hash table are usually linked lists, but they don't have to be
- If the number of items in a hash table is *n* and the number of bins is *N*, then the **load factor** is *(n/N)*
  - Load factor can be reduced by adding more bins, but adding bins requires the whole table to be re-hashed
- Choose hash codes and functions that distribute items evenly across the bins

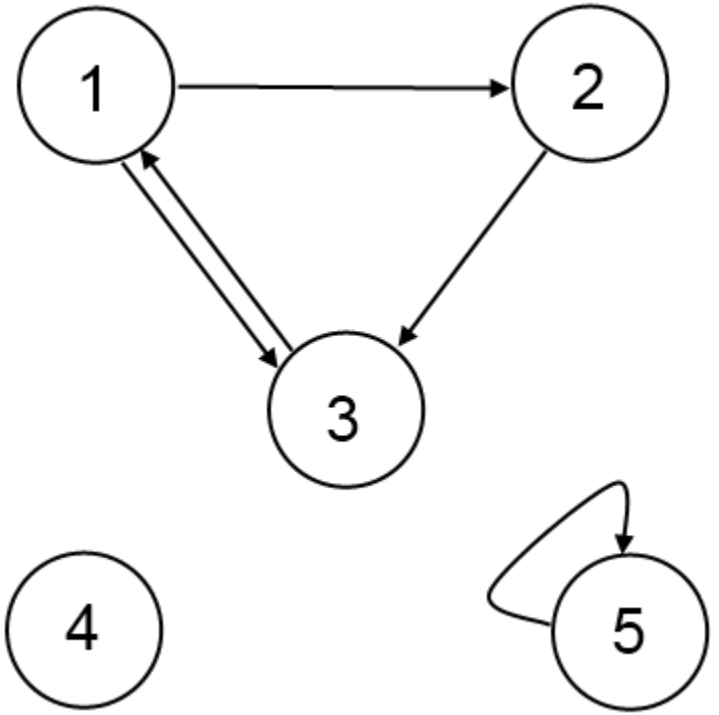
Graphs overview

- Graphs consist of **vertices** and **edges**
- **Vertex:** a point (v) in the graph
- **Edge:** connects two vertices (v, u)
  - If a graph is **directed**, edges only go one way
  - Edges can have **weight** associated with them, e.g. the distance between two points
- The **degree** of a vertex is the number of edges touching it
  - Vertices in a directed graph have **indegree** and **outdegree**
- Graph is **connected** if there is a path between every two vertices

Adjacency Matrix

Create the adjacency matrix for this graph:

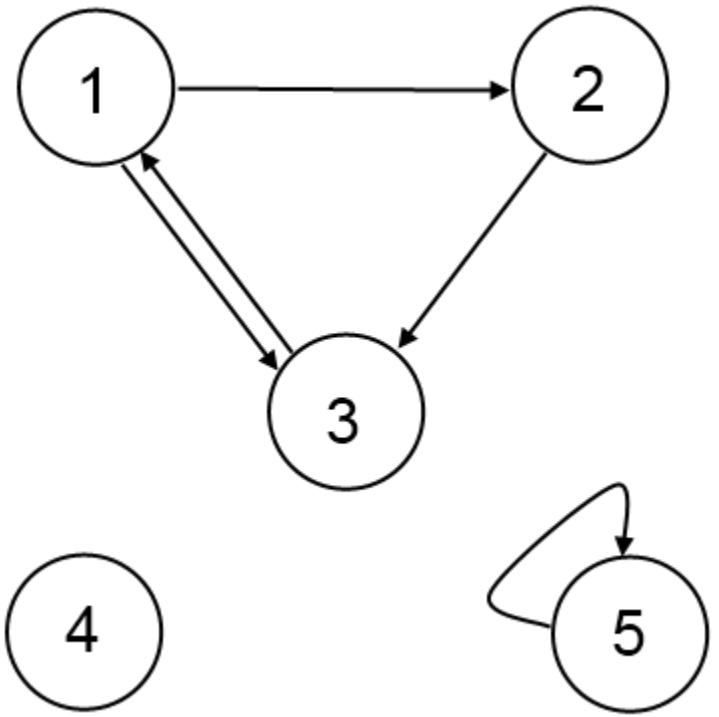
	1	2	3	4	5
1					
2					
3					
4					
5					



Adjacency Matrix

Create the adjacency matrix for this graph:

	1	2	3	4	5
1	-	T	T	-	-
2	-	-	T	-	-
3	T	-	-	-	-
4	-	-	-	-	-
5	-	-	-	-	T

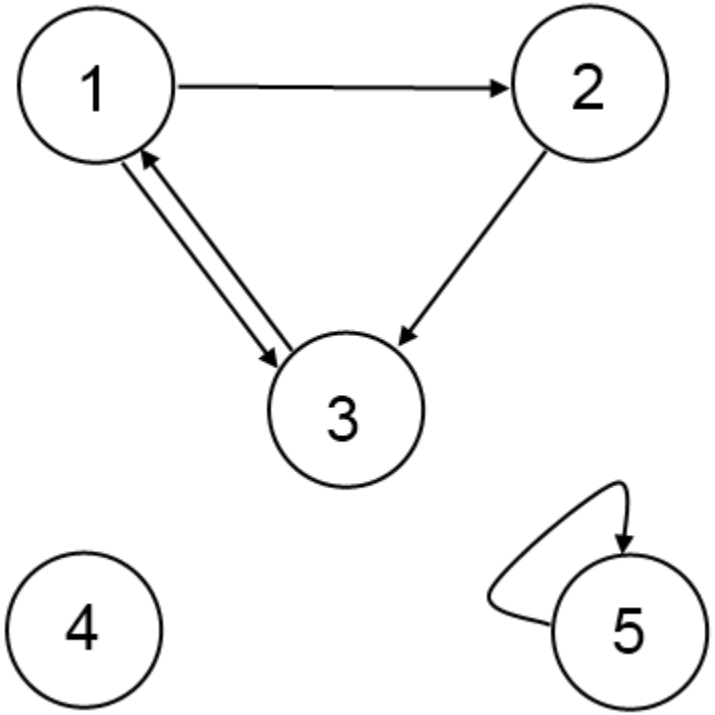


Memory required: $O(V^2)$

Adjacency List

Now create the adjacency list for this graph:

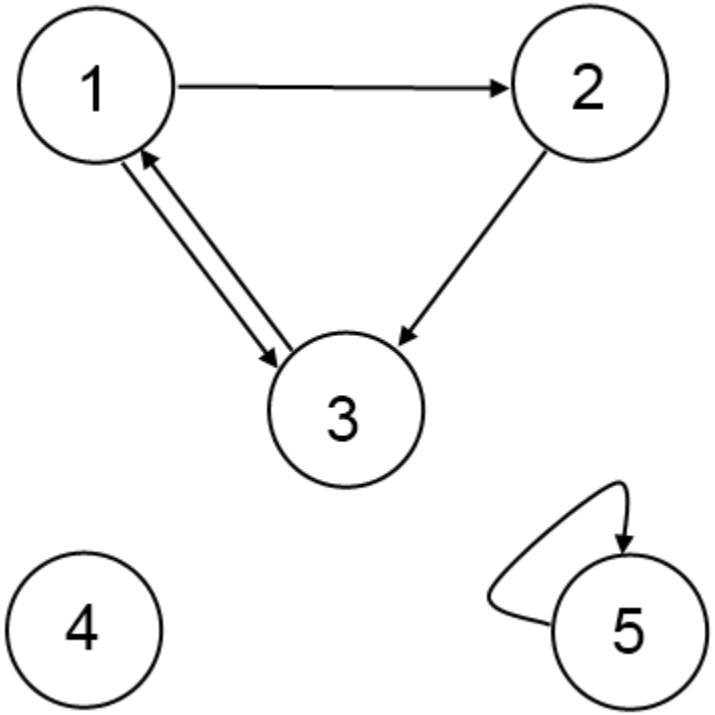
1:
2:
3:
4:
5:



Adjacency List

Now create the adjacency list for this graph:

```
1: 2, 3
2: 3
3: 1
4:
5: 5
```

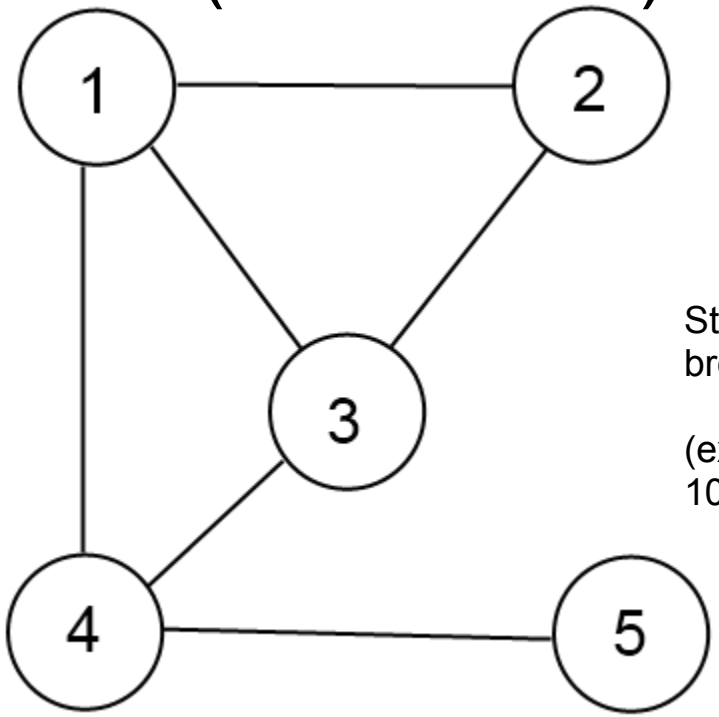


Memory required:

Theta(V+E)

BFS, DFS and Trees

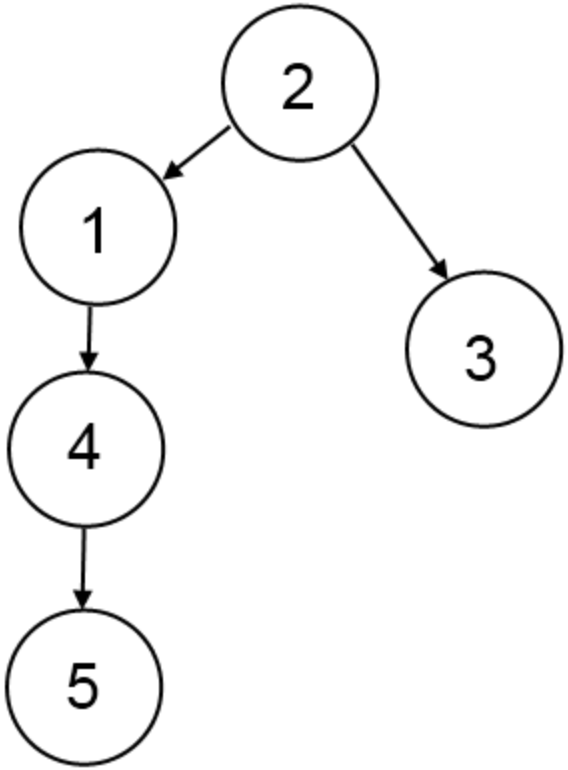
Turn the following graph into a tree rooted at vertex 2, and then specify the Breadth First and Depth First Search (BFS & DFS) orders of this graph.



Start your BFS/DFS at 2, then break ties from least to greatest:

(ex: if you are tied between 9 and 100, pick 9)

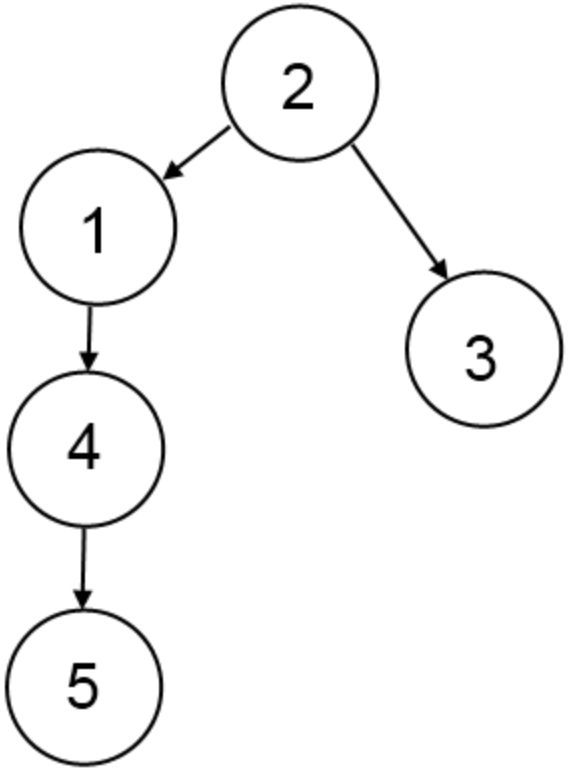
BFS, DFS and Trees



BFS search order:

DFS search order:

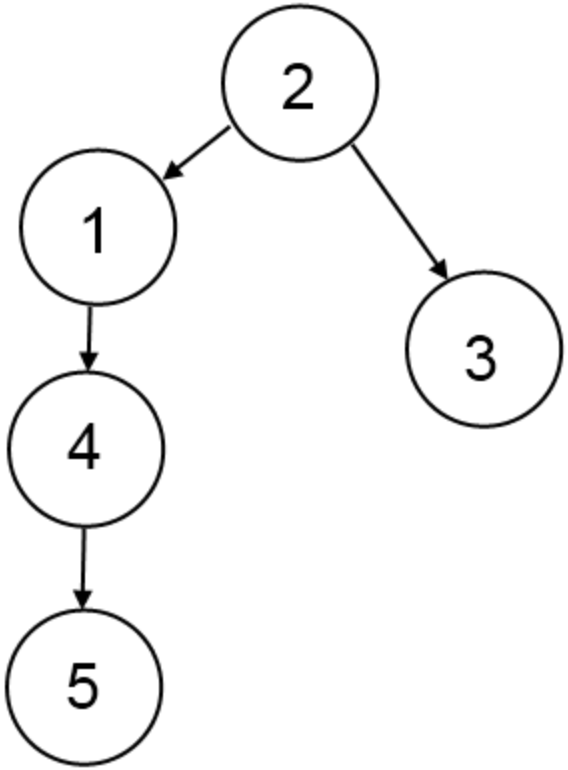
BFS, DFS and Trees



BFS search order: 2, 1, 3, 4, 5

DFS search order:

BFS, DFS and Trees



BFS search order: 2, 1, 3, 4, 5

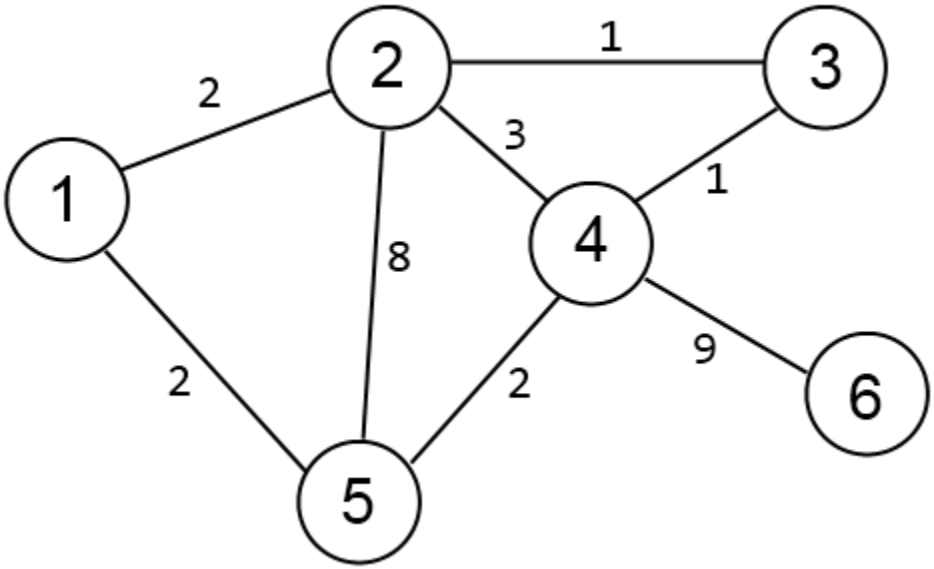
DFS search order: 2, 1, 4, 5, 3

BFS, DFS and Trees info

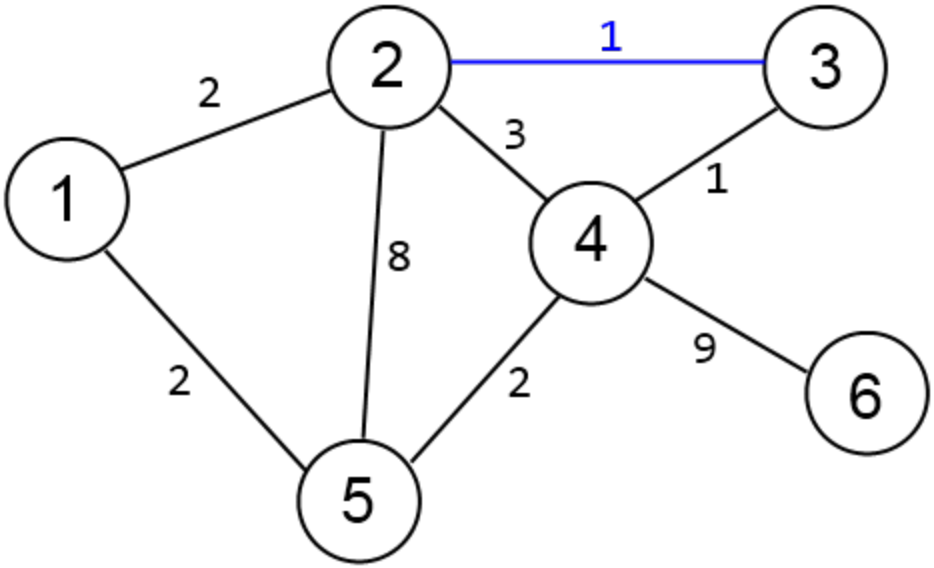
- **DFS** is an algorithm based on recursively checking a node's children
  - See the pseudocode in Lecture 28
  - O(V + E) for adjacency list, $O(V^2)$ for matrix
- **BFS** uses a queue to iteratively deepen in a graph
  - See the pseudocode in Lecture 29
  - Also O(V + E) for adjacency list and $O(V^2)$ for matrix
- Adjacency lists are good for **sparse** trees, but as E -> V^2, adjacency matrices become faster

Kruskal's Minimum Spanning Tree

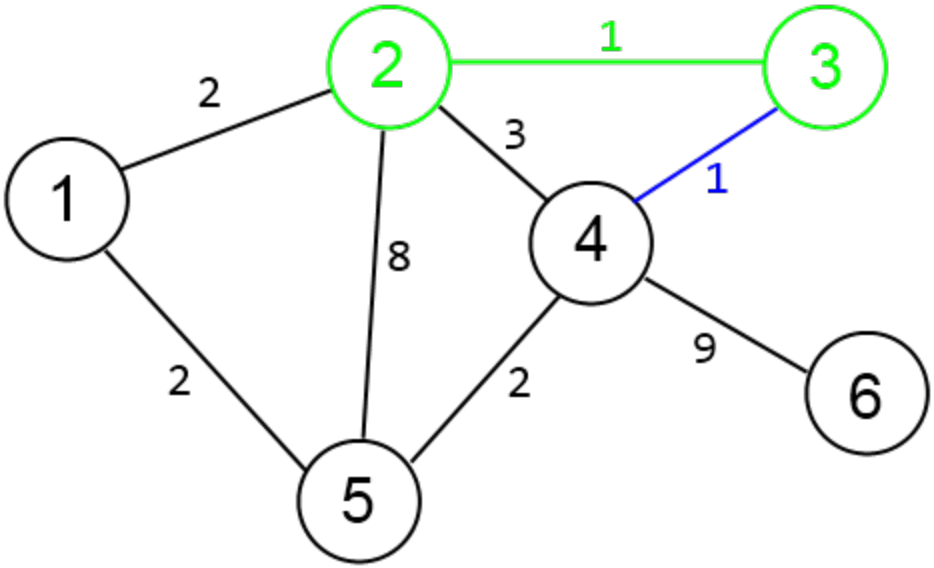
Find a subset of the edges in E that connects every vertex in V with the shortest possible total length:



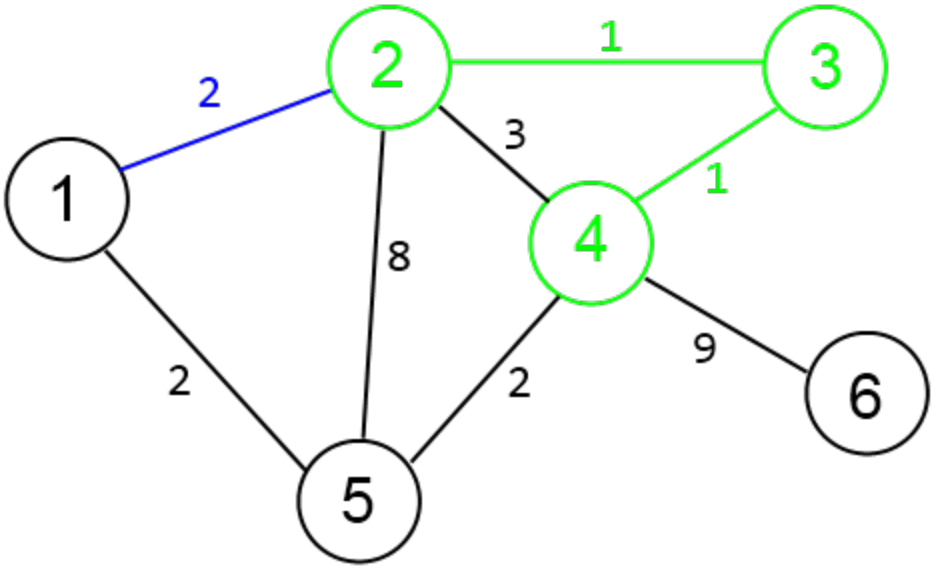
Kruskal's Minimum Spanning Tree



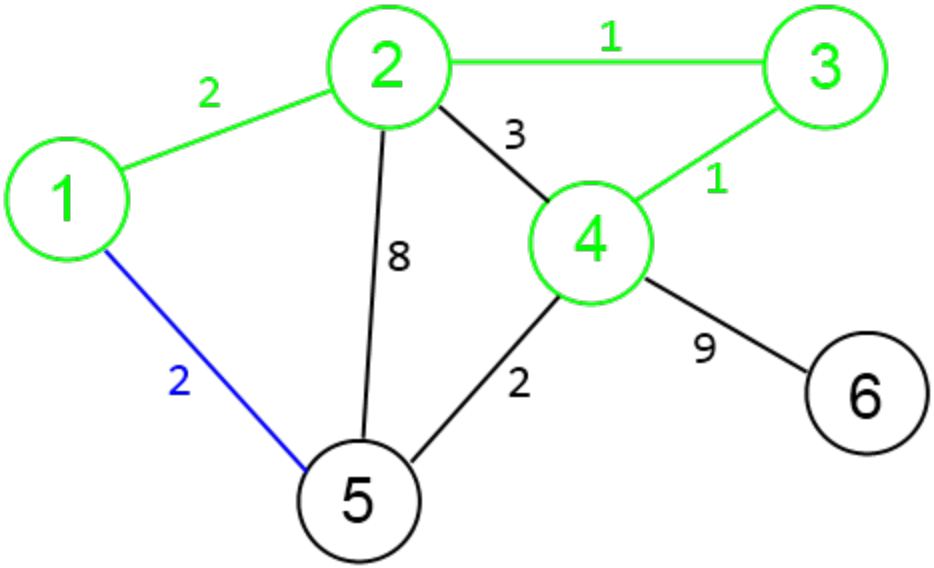
Kruskal's Minimum Spanning Tree



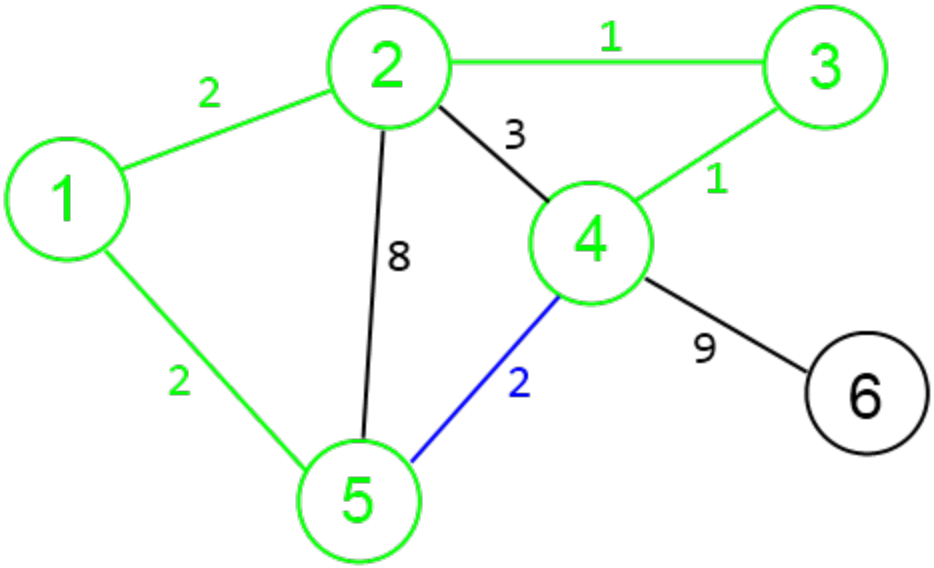
Kruskal's Minimum Spanning Tree



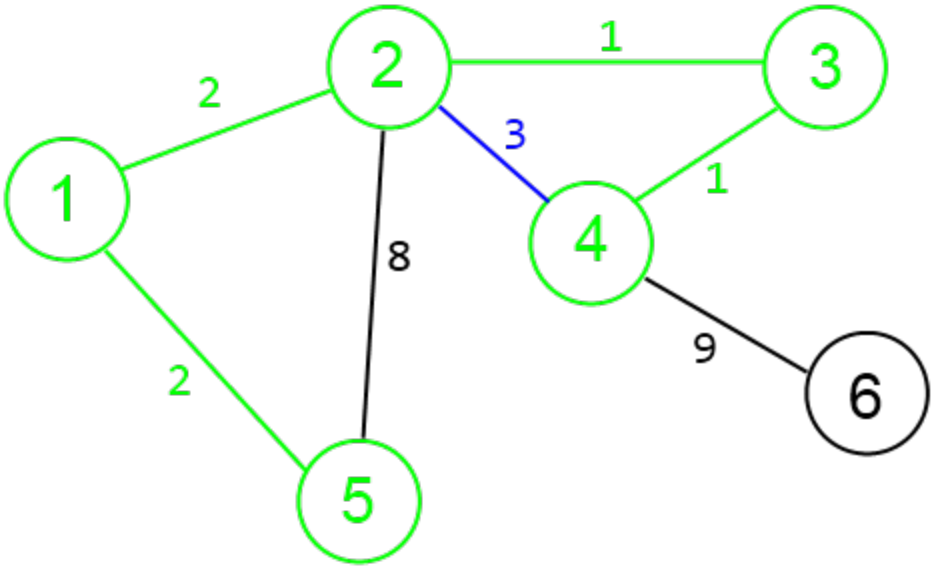
Kruskal's Minimum Spanning Tree



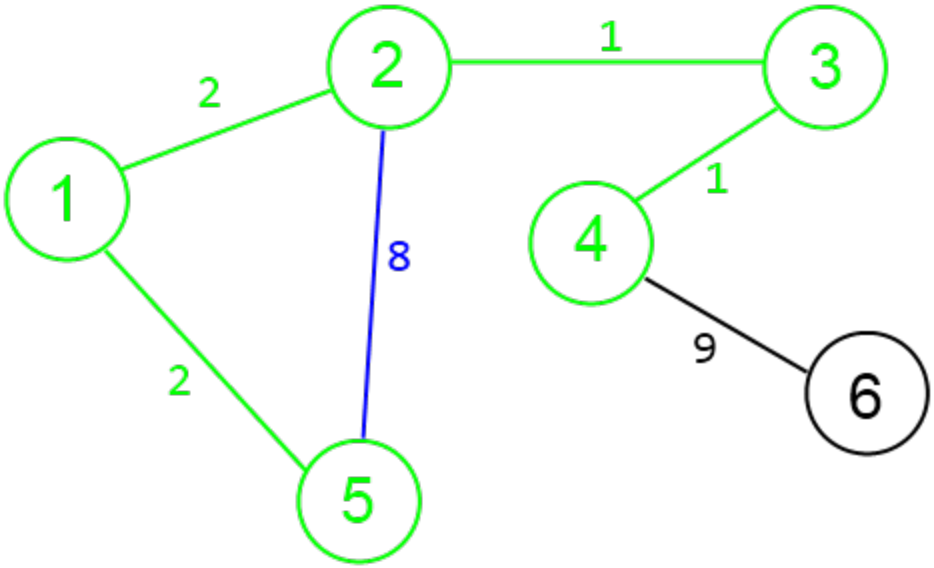
Kruskal's Minimum Spanning Tree



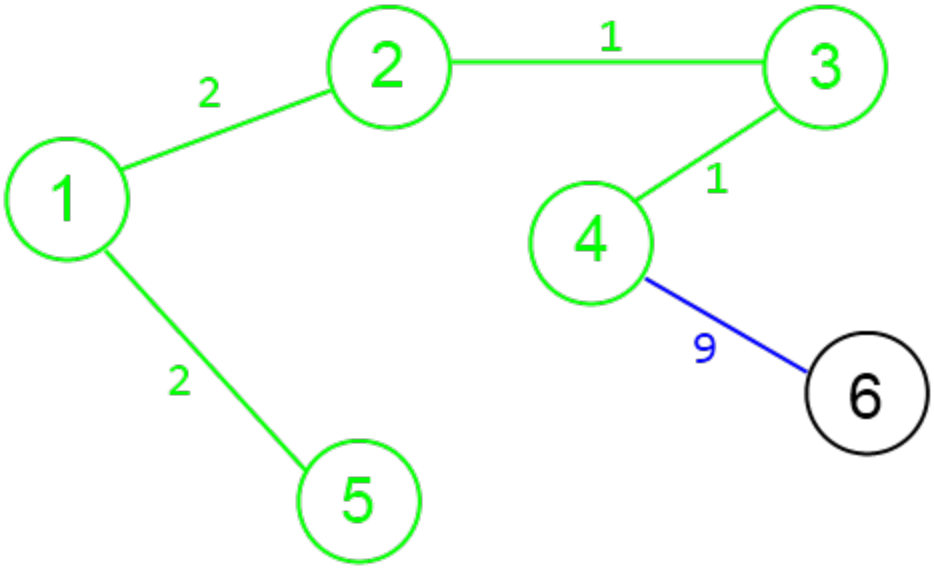
Kruskal's Minimum Spanning Tree



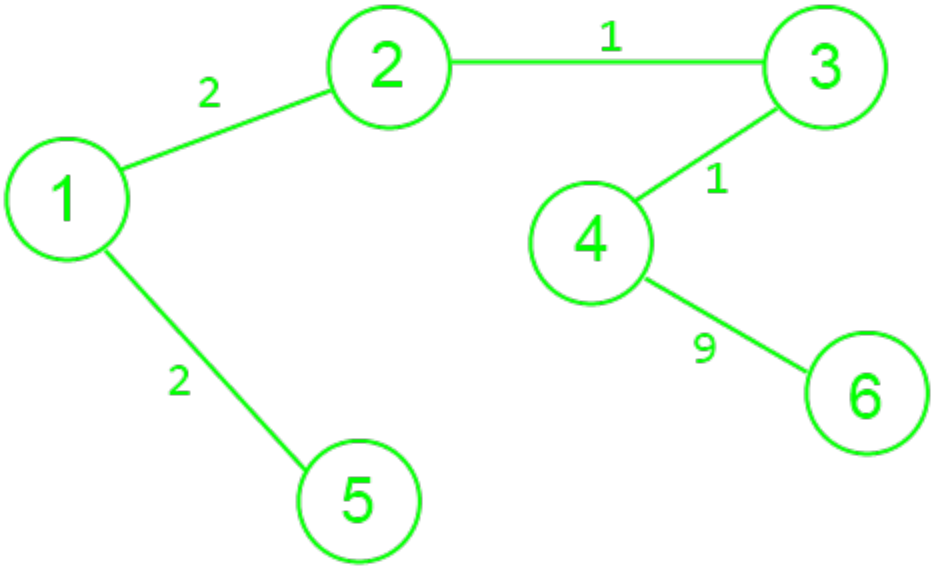
Kruskal's Minimum Spanning Tree



Kruskal's Minimum Spanning Tree



Kruskal's Minimum Spanning Tree



Note: there are two other minimum spanning trees for this graph, depending on how equal-length edges are ordered.

Kruskal's Minimum Spanning Tree

- The algorithm goes through each edge (u, v) in increasing weight order, and if the two points are not already connected, it adds the edge to the set
- Sorting the edges takes O(E log E) time
- Checking connectivity with a DFS takes Theta(E*(V + E)), which is rather slow
  - A trick exists to make the check time O(E log E), you haven't gone over it yet
- The total time you *need to know* is O(V + E log E) = **O(V + E log V)**

Heaps

A heap is a binary tree with the following properties:

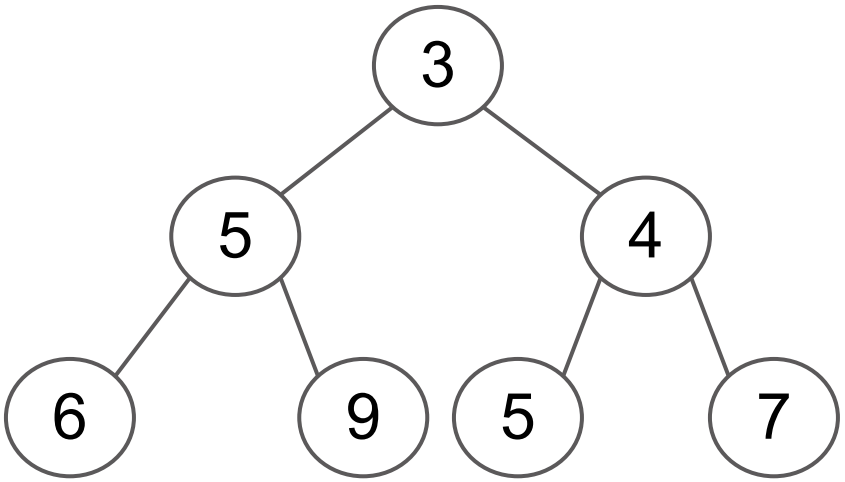
- The tree is complete i.e. every level is filled, except possibly the last, which is filled left to right.

- Heap-order property: If node B is a descendant of node A:
  - Min heap: key of B >= key of A
  - Max heap: key of B <= key of A

Heaps

Heaps are often represented as arrays:

X	3	5	4	6	9	5	7



Heaps

Why do we use arrays instead of linked lists to store heaps?

Heaps

Why do we use arrays instead of linked lists to store heaps?

Ans: So we can access nodes in O(1) time vs. O(n) time for linked lists. This is especially helpful when accessing parents or children of a node.

Heaps

Accessing parents and children:

If we store keys in an array starting from index 1:

Say a key is at index **i**, the parent of its node is at index **floor(i/2)**, while the children are at indices **2i** and **2i+1**.

Ex: Key at index 3.

Parent: index 1.

Children: indices 6 and 7.

Heaps

True/False:

In a min heap:
Key k_1 is at level l_1 and key k_2 is at level l_2.
If $k_1 < k_2$, then $l_1 <= l_2$?

(Assume root is at level 1)

Heaps

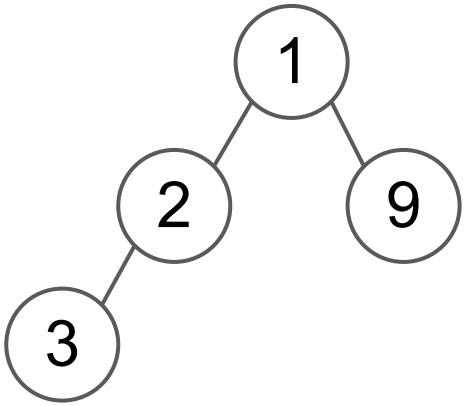
True/False: In a min heap:

Key k_1 is at level l_1 and key k_2 at level l_2.

If $k_1 < k_2$, then $l_1 <= l_2$?

Ans: False

Ex:



k_1: 3

k_2: 9

l_1: 3

l_2: 2

Heaps

Consider the following situation in a min heap:

Suppose key k_1 is the root at level 1 and key $k_2 = k_1 + x$ (assume x is positive). What are the highest and lowest levels k_2 can be?

Assume all keys are unique.

Heaps

Suppose key k_1 is the root at level 1 and key $k_2 = k_1 + x$. What are the highest and lowest levels k_2 can be? Assume all keys are unique.

Ans: Highest level: 2

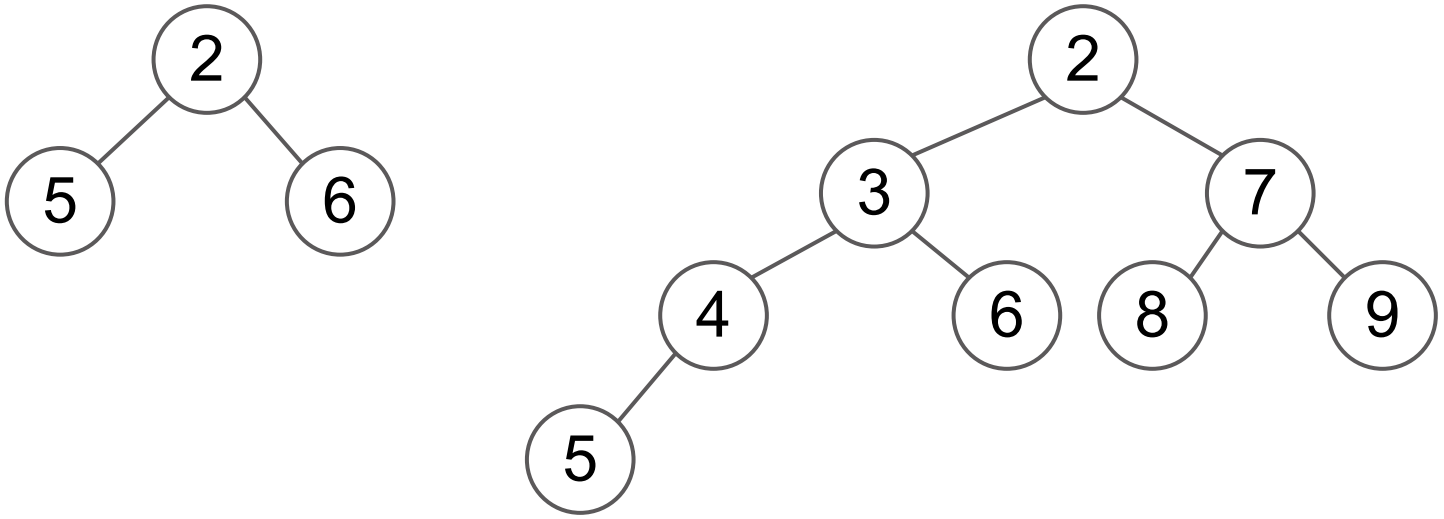
Lowest level: x+1

Heaps

Ans: Highest level: 2

Lowest level: x+1

Ex: $k_1 = 2$, $k_2 = 5$. Therefore, x = 3.



Heaps

To `insert()` in a heap:

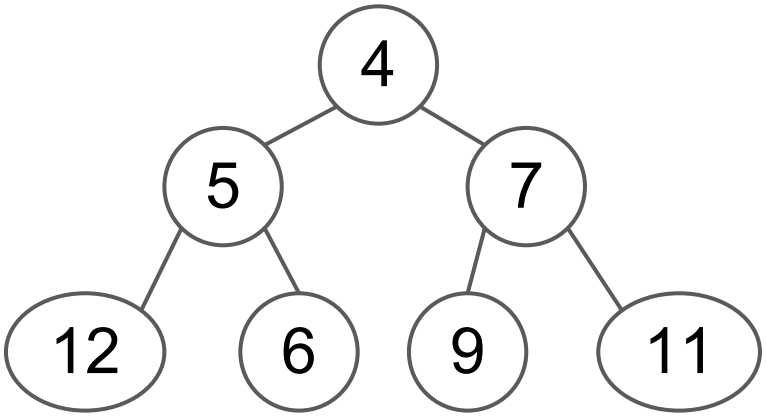
- Insert item at the end of array.
- Bubble up item by repeatedly swapping with parents until heap-order property is satisfied.

To `removeMin()`:

- Replace first element (root) with last element, and remove last node.
- Bubble down new root by repeatedly swapping with smaller of two children until heap-order property is satisfied.

Heaps

Starting with the following min heap, perform the given operations and draw the resulting trees:

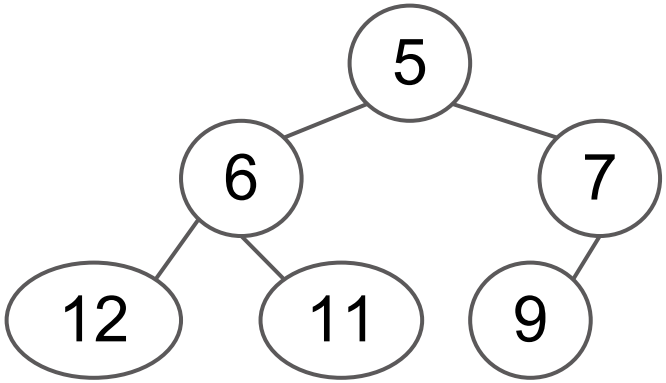


(a) `removeMin()`

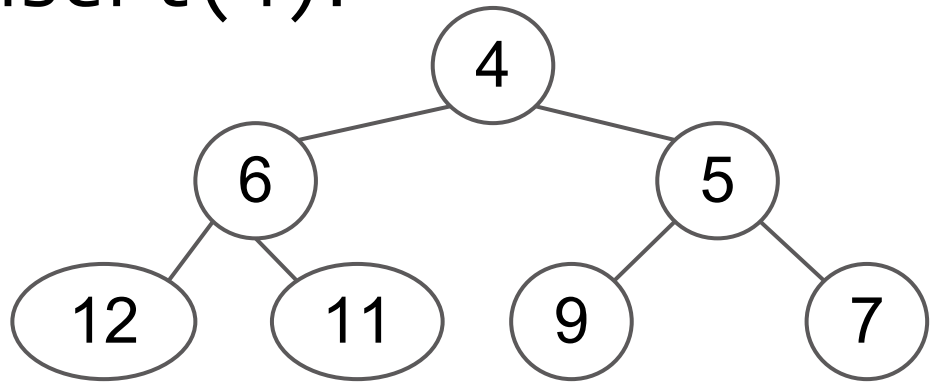
(b) `insert(4)`

Heaps

Ans: (a) `removeMin()`:



(b) `insert(4)`:



Heaps

Bottom-up heap construction to convert an array into a heap:

- Start with last non-leaf node.
- Bubble down element until heap-order property is satisfied.
- Going backwards, do the same for every non-leaf node.

Ex: If the last non-leaf node is at index 4, bubble down the element at index 4, then at index 3, then 2, then 1.

Heaps

If a heap has ***n*** elements, what is the index of the last non-leaf node?

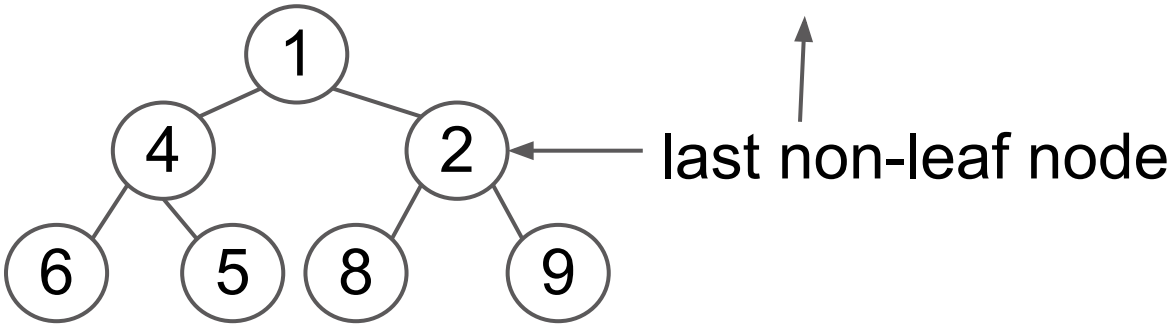
Heaps

If a heap has ***n*** elements, what is the index of the last non-leaf node?

Ans: `floor(n/2)` (parent of last element)

Ex:

1	4	2	6	5	8	9
1	2	3	4	5	6	7



Heaps

Perform bottomUpHeap on the following array, redrawing the array after every swap.

X	8	3	9	6	2	1	4

Heaps

X	8	3	9	6	2	1	4

Ans:

X	8	3	**1**	6	2	**9**	4

X	8	**2**	1	6	**3**	9	4

X	**1**	2	**8**	6	3	9	4

X	1	2	**4**	6	3	9	**8**

Heaps

Write a function **`int max()`** to return the maximum entry in a min heap structured as a tree. The following instance variables are available:

`root` of class `Heap`

`leftChild`, **`rightChild`**, **`parent`** and **`entry`** of class `HeapNode`.

Assume the heap is non-empty and entries are integers.

Heaps

Ans:

```
int max() {
     return rec(root.leftChild, root.rightChild, root.key);
}

int rec(HeapNode n1, HeapNode n2, int m) {
     if (n1 == null && n2 == null)  {
          return m;
     } else if (n1 == null)  {
          return rec(n2.leftChild, n2.rightChild, n2.key);
     } else if (n2 == null) {
          return rec(n1.leftChild, n1.rightChild, n1.key);
     } else {
          int m1 = rec(n1.leftChild, n1.rightChild, n1.key);
          int m2 = rec(n2.leftChild, n2.rightChild, n2.key);
          return m1 > m2 ? m1 : m2;
     }
```

Good luck on your Midterm!