Binary Search Trees

CS 580U Fall 17

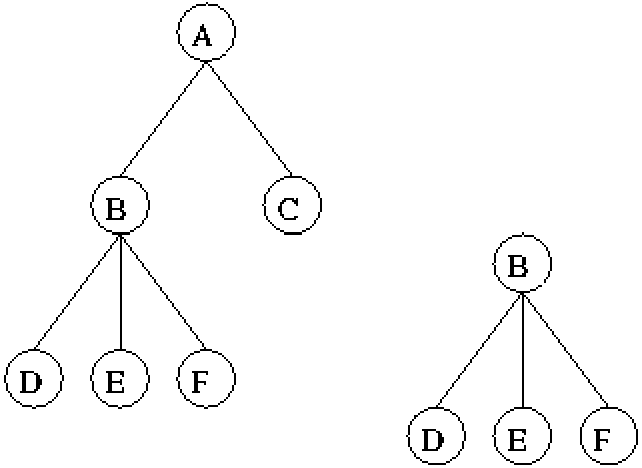
A Better ADT

- Vectors make working with arrays easier, but no real performance improvements
- Linked list improves the array, but slow on random access
    - using nodes gives us memory flexibility
- How can we improve?
    - What if we add additional node pointers to each node
        - *we can divide the problem in half*

The Tree ADT

Definition:

A finite set of nodes such that one node is designated as the root. All other nodes are partitioned into sets, each of which is a tree.



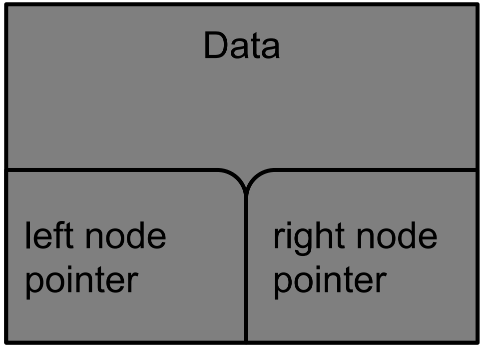
Terminology

- Root node
  - The ‘top’-most node in the tree
- Branch
  - the connection to a child node that may or may not contain a subtree
- Subtree
  - subnode that contains subnodes

Properties of a Tree

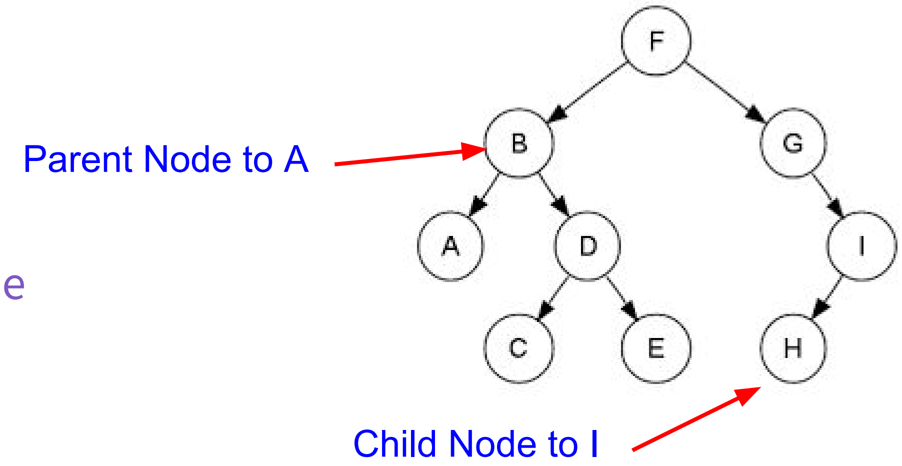
- max # of leaves
  - all nodes that do not have branches
- max # of nodes
  - all nodes in the tree
- Height for a tree
  - path depth
    - *The number of branches between the current node and the farthest leaf*
  - max depth
    - *The number of branches between the root node and and the farthest leaf*

A Binary Node



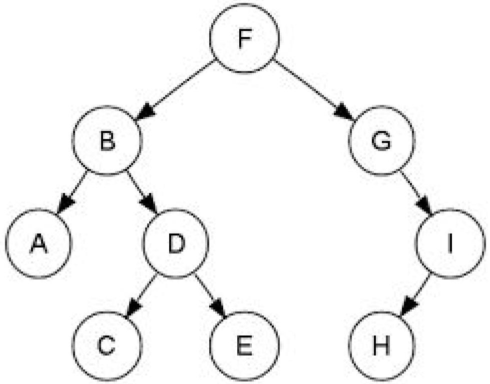
Parents and Children

- Parent Node
  - The immediate predecessor node in the tree structure
- Child Node
  - the immediate following node in the tree structure

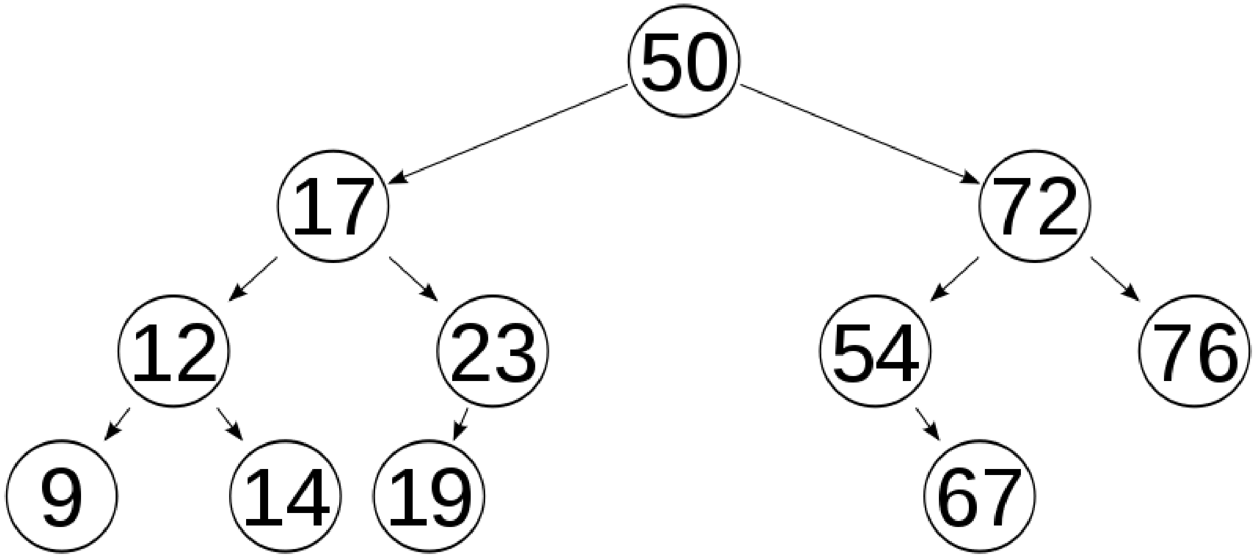


Height

- A tree's depth is the number of 'steps' to get to a leaf
  - Only count branches, not nodes
  - What is the depth of A? H?
- A tree's height is based on its maximum depth
  - What is the tree's height?



Binary Search Tree



Binary Search Tree

- Common Tree Data Structure
  - Each node has two branches
  - Branches may point to another node, or NULL
  - all data in every node of the left subtree are less than the data in the node
  - all data in every node of the right subtree are greater than the data in the node
  - All data in a BST is unique

Implementation

- Array
  - You can use an array to implement your tree
- left child index = 2*(*Parent Index*)+1
- right child index = 2*(*Parent Index*)+2
- parent index = (*Child Index*-1)/2 (truncate)

- Linked
  - a linked list using structs and pointers
- a data field
- a left child field with a pointer to a node
- a right child field with a pointer to a node
  - ...additional pointers if not a binary tree
- a parent field with a pointer to the parent node

Compare

- Array
  - Pros:
    - Constant time Access
  - Cons:
    - Complexity
    - Array Max Size must be known

- Linked Nodes
  - Pros:
    - Dynamic Memory size
  - Cons:
    - Complexity
    - Linked Traversal

The BST ADT

Made up of two structs:

- Tree
    - Node * root
- Node
    - Node * left
    - Node * right
    - Data * data
    - Node * parent *(optional, but recommended)*

BST's use recursion

- Recursion is the process of a function calling itself to perform iteration
    - Basically, using the stack as your loop
- Why use recursion
    - Simplifies code greatly
- Why not use recursion?
    - Uses more memory
    - Can be very slow

Formal Def. of Recursion

Recursion is an iterative procedure that defines the value of a function, argument n, by using the value of the previous argument n − 1

$$pd(n) = \begin{cases} 0 & \text{if } n=0 \text{ or } n=1 \\ pd(n-1) & \text{otherwise} \end{cases}$$

Recursion has two parts

- Recursion requires two parts within the recursive function:
    - A base case that defines the simplest objects in S
        - *an end to the recursion.*
    - A recursive step that defines how objects in S can be modified, reduced, or combined to produce another object closer to the final result
- Recursion Rules
    - Every recursive method must have a base case -- a condition under which no recursive call is made -- to prevent infinite recursion.
    - Every recursive method must make progress toward the base case to prevent infinite recursions

Classwork

Recursion

```
int toZero(int num){
        printf("%d\n", num);
        if(num == 0)
                return;
        else
                (num > 0) ? toZero(num-1) :
                        toZero(num+1);
}
```

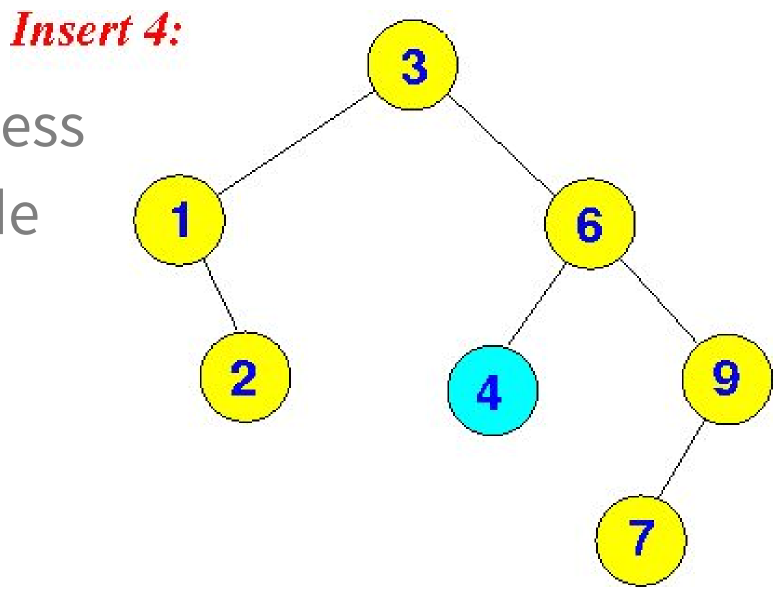
Tree Traversal

- Traversal, whether for insertion, deletion, or read, uses recursion
  - each can be completed iteratively as well, but it is slightly more complex
- A method is recursive if it can call itself directly or indirectly
  - A function, foo(), is indirectly recursive if it calls, bar(), which in turn calls foo()

Recursive Insert

Recursive insert

- Check if value is greater than or less than the value in the current node
    - If greater, go right, check again
    - If less, go left, check again
    - If null, add a leaf to the tree
    - if equal, NOOP
        - *All values in a tree should be unique*

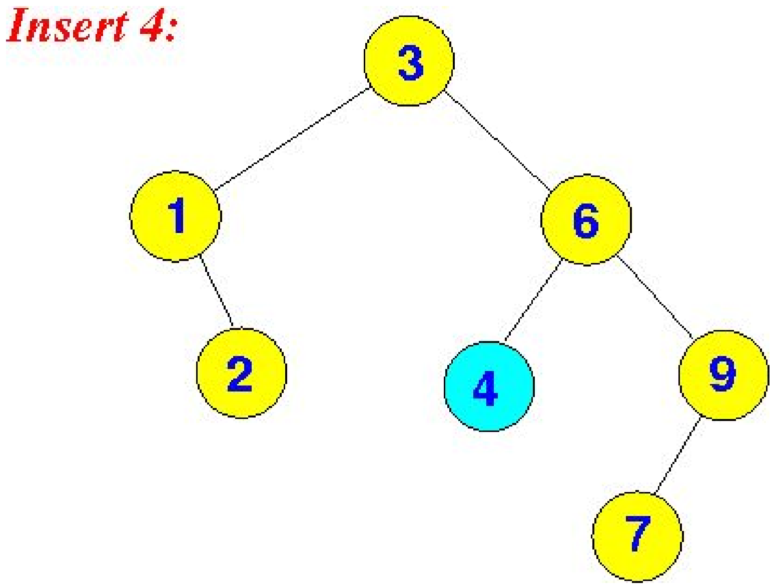


Inserting a node into an existing tree

- Operations:
  - Use recursion to traverse through the tree
  - Insert node as a leaf
- What information do you need?
  - Data inserted
  - current node visited
    - *with access to child nodes*

Insert (pseudocode)

```
insert(node, data){
    if (data < node.data)
        if(node.left == NULL)
            addLeaf(node, data);
        else
            insert(node.left, data)
    else if (data > node.data)
        if(node.right == NULL)
            addLeaf(node, data);
        else
            insert (node.right, data)
```



**must handle special case where tree is empty*

Problems

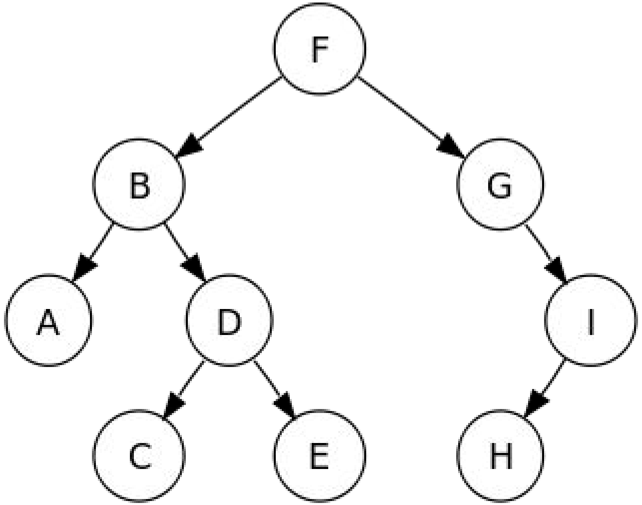
- How to choose the root node?
  - What happens if you choose a bad root node?
    - *Becomes a linked list*
- Insertion (with a well chosen root)
  - Best Case?
    - *constant time*
  - Worst Case?
    - *size of the list*

Recursive Read

- Almost identical to insert
    - Requires a return statement
        - *Reference or pointer*
- Check if value is greater than or less than the value in the current node
    - If greater, go right, check again
    - If less, go left, check again
    - If equal, return data
    - If null, error message

Read (pseudocode)

```
read(node, data){
        if(data == node.data)
                return node.data
        else if (data < node.data)
                if(node.left == NULL)
                        print("value not found");
                else
                        return  read(node.left, data)
        else if (data > node.data)
                if(node.right == NULL)
                        print("value not found");
                else
                        return read (node.right, data)
```



**must handle special case where tree is empty*

Delete

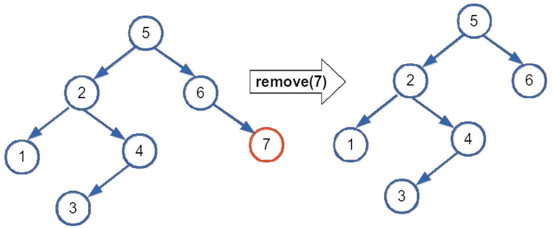
- Basic Deletion process
  - Remove the pointer to the node
  - delete node
- Operations required
  - Tree Traversal to find the node
  - Must always keep track of the parent node
    - *This is where maintaining a parent pointer in the node is helpful*

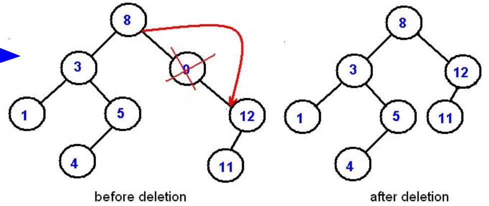
Delete - 3 Scenarios

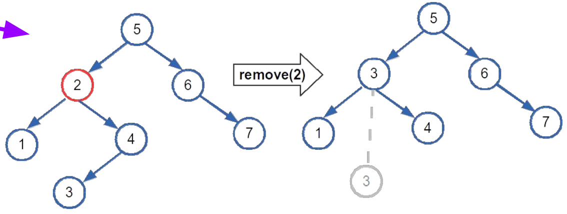
- What are the primary cases for deleting a node?
  - Delete a leaf node - easy
    - *set parent node pointer to null*
    - *delete node*
  - Delete a 1 branch parent node
    - *Can't just delete the node, because then our tree would "fall apart."*
  - Delete a 2 branch parent node
    - *We must promote one of the children to become the new parent.*

Delete Algorithm

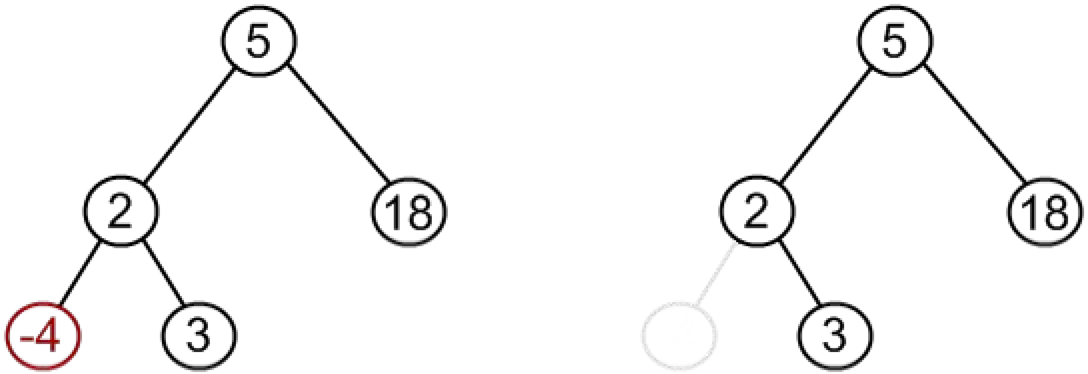
```
node = findNode(data);
if ( node not in BST )
      return;
else if ( node has no subtrees ){
        deleteLeaf(node);
}else if ( node has 1 tree ) {
        shortCircuit(node);
}else if(node has 2 subtrees){
        promotion(node);
}
```







Deleting Leaf Nodes



DeleteLeaf Pseudocode

```
void removeLeaf(Node * leaf)
    if leaf->parent->right == leaf
        leaf->parent->right = NULL
    else
        leaf->parent->left = NULL
    delete leaf
```

What if the leaf is the root?

- Delete root
- Set root to null to signify an empty tree

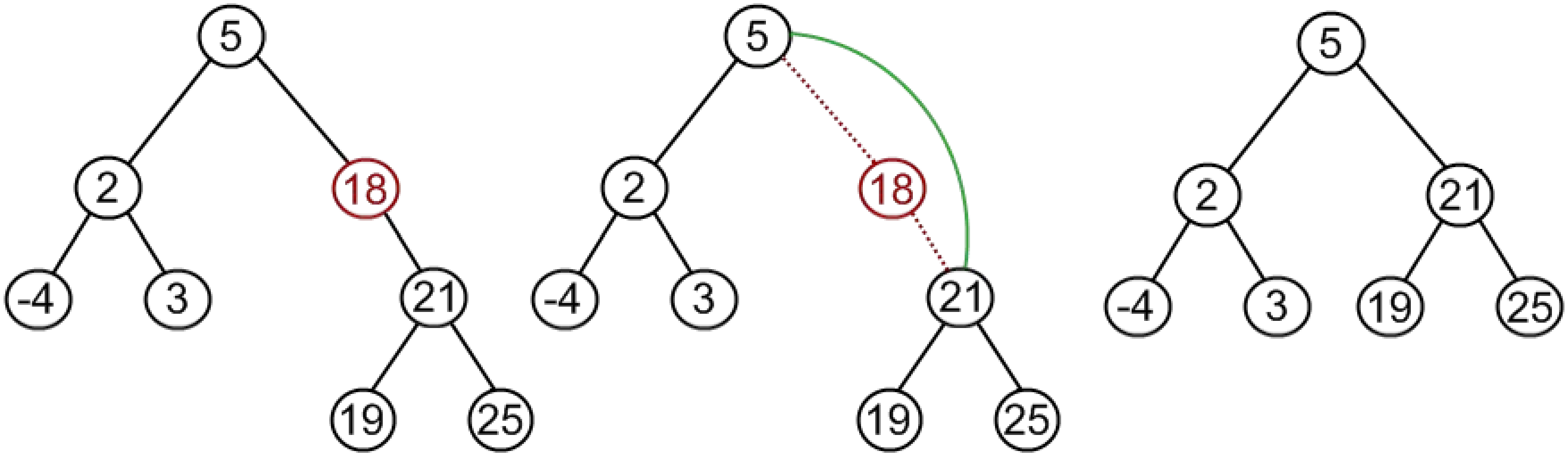
Short Circuit Algorithm

- The Short Circuit Algorithm sets the child node's child to be the child of the parent, then deletes the extra leaf node
- When deleting a parent node in a BST, you must ensure that the new parent is:
    - *bigger than all the other children in the left tree*
    - *smaller than all the other children in the right tree*

You must maintain the BST structure

Deleting Single Branch Nodes

Short Circuit Method



Short Circuit Pseudocode

```
void shortCircuit(Node * node)
	if( node->parent->right == node)
		if node->right == NULL
			parent->right = node->left
			node->left->parent = node->parent
		else
			parent->right = node->right
			node->right>parent = node->parent
	else
		...
	delete node
```

What if the node to be delete is the root?

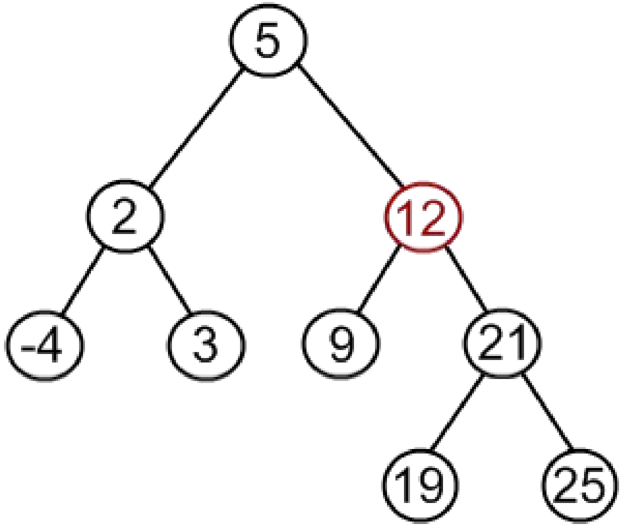
We have to promote the child node to become the root

Classwork: Max Or Min

```
Node* searchMin(Node * node){
	if(node->left != NULL)
		return searchMin(node->left);
	else
		return node;
}

Node* searchMax(Node * node){
	while(node->right != NULL)
		node = node->right;
	return node;
}
```

Deleting 2 Child Nodes

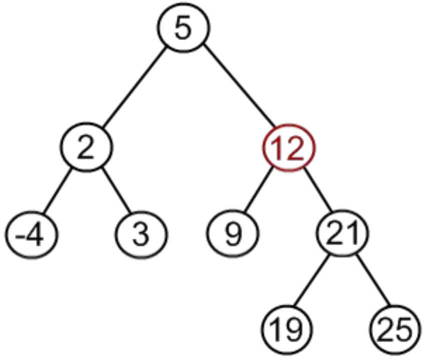


Promotion

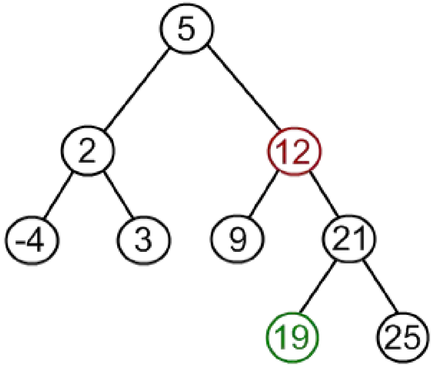
- We must promote a node to a higher space in the tree
- There are at most two possible candidates:
  - the rightmost child of the left subtree
    - *Traverse left once, then right as far as possible*
  - the leftmost child of the right subtree
    - *Traverse right once, then left as far as possible*
- It doesn't matter which one we pick,
  - both choices will maintain the BST structure
  - Successor node
    - *the node in the right subtree that is min value -or-*
    - *the node in the left subtree that is max value*

Promote Successor

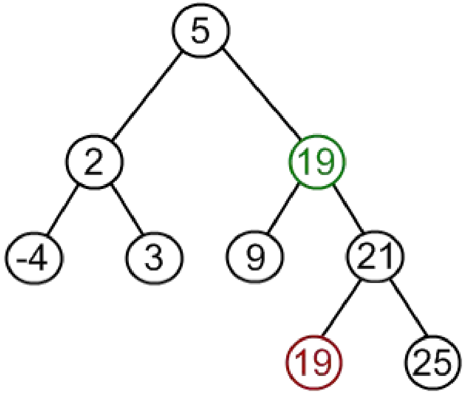
1) Find node to delete



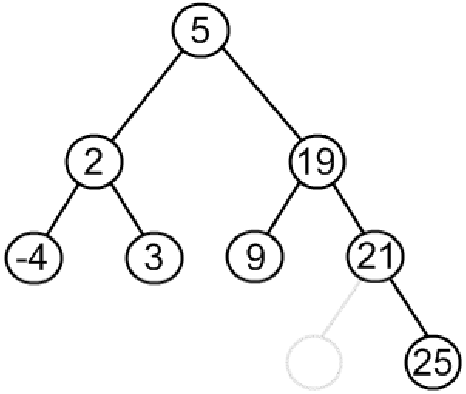
2) Find successor



3) Replace value



4) Delete Successor

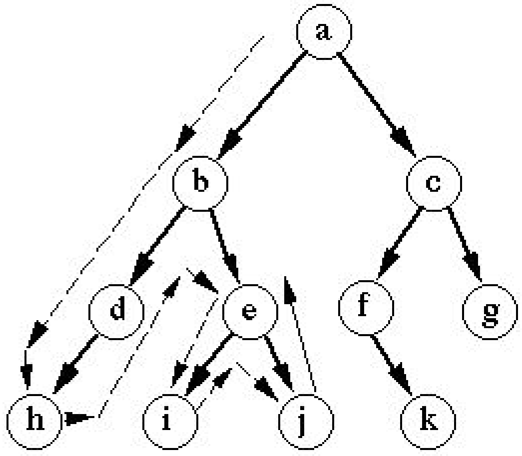


Promotion Selection

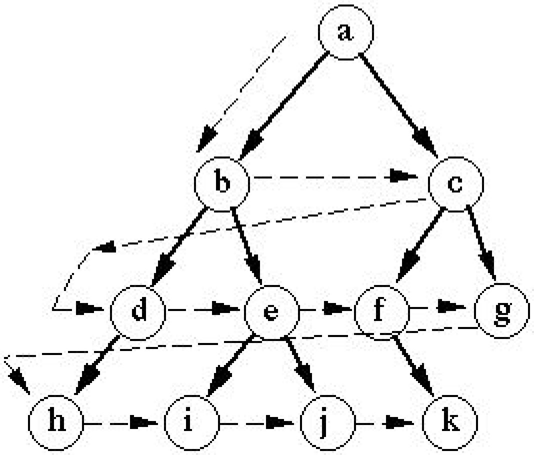
- functions needed
  - Node * getMaxNode(Node * node)
  - Node * getMinNode(Node * node)
- What if the min or max is not a leaf node?
  - We call our short circuit algorithm
  - min value has right subtree?
    - *Run single subtree (short-circuit) algorithm*
- What if the delete node is the root node?
  - Special case: must promote leftmost max or rightmost min to root node

Promotion Algorithm

```
void promotion(Node * n){
    Node * d_node = searchMin(n->right);
    n->data = d_node->data;
    //Leaf
    if(d_node->left==NULL && d_node->right==NULL){
        removeLeaf(d_node);
        //one branch
    }else{
        shortCircuit(d_node);
    }
}
```



Depth-first search



Breadth-first search

Searching a Tree

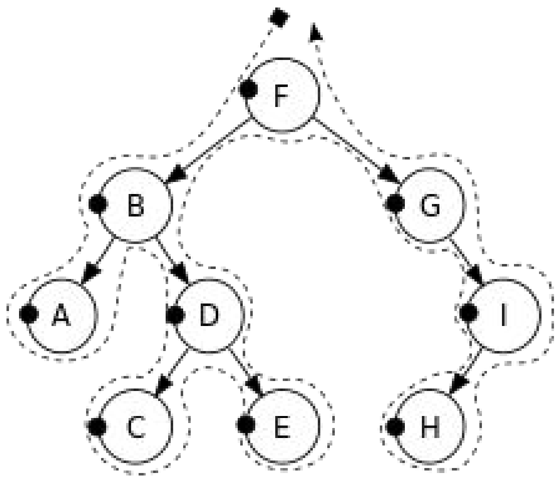
Depth First Traverse

- Depth First travels down the tree structure to find a value
- Basic Algorithm:
  - Start at the root node,
    - *traverse down the left until finding a leaf*
    - *traverse back up until finding a right branch*
    - *repeat until no right branch*

PreOrder

- Process each node as you reach it in traversal order
- Algorithm:
  - preorderTraversal(node):

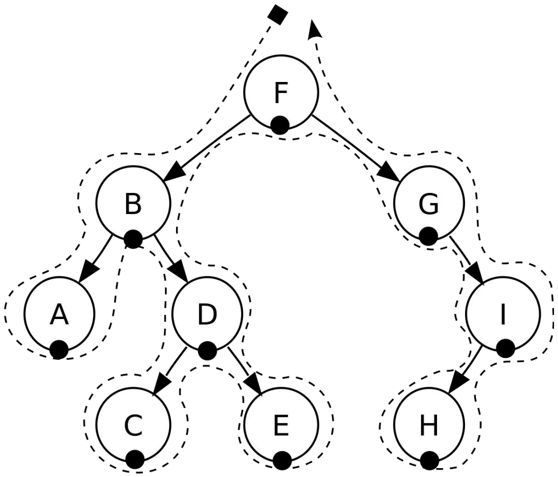
```
preorderTraversal(node):
    process(node)
    preorderTraversal(node->left)
    preorderTraversal(node->right)
```



InOrder

- Visit each node in ascending order
- Algorithm:
  - inorderTraversal(node):

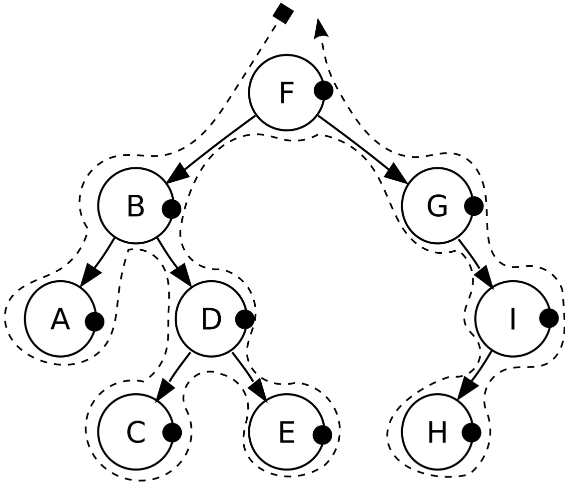
```
inorderTraversal(node->left)
process(node)
inorderTraversal(node->right)
```



PostOrder

- Visit each node as you reach it in traversal order
- Algorithm:
  - postorderTraversal(node):

```
postorderTraversal(node->left)
postorderTraversal(node->right)
process(node)
```



Classwork

Tree Traversal

- 5,3,2,1,4,8,6,7,9
- 1,2,3,4,5,6,7,8,9
- 1,2,4,3,7,6,9,8,5
- Post Order - delete
- In Order - sort
- Pre Order - copy

Use Cases

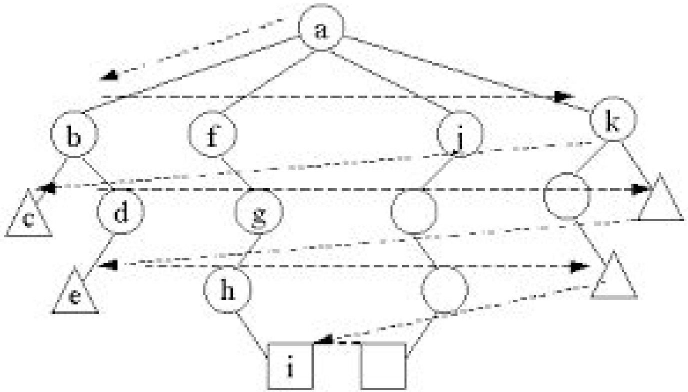
- Basic required Tree operations:
  - Deep Copy Tree
  - Delete Tree
  - Sorted Print
- Depth First Search
  - Traverse as far as possible down a single path
- Determine use case for each DFS Operation
  - *PreOrder:* copy of the tree
  - *InOrder:* gives nodes in non-decreasing order
  - *PostOrder*: used to delete the tree

Breadth First Search

- Visit every node on a particular level before going to the next level
  - Also called Level order
- How to Implement?
  - Not really a recursive algorithm
    - *Can still implement recursively, but not traditionally done with recursion*
  - With a helper Data Structure to store the next level
    - *need to store each node in order, and ensure they are accessed in the same order*
  - Which data structure would work best?

Breadth First Helper ADT

- A Queue
  - Enforces first in first out
- Assume we have a Tree ADT
  - Child nodes are stored in an array within the Node class



Breadth First Search Algorithm

- Assume a tree where each node has an unknown number of children
  - ```
    breadthFirstSearch(){
        Queue q;
        q.enqueue(root);
        while(!q.empty()){
            node = q.dequeue();
            processNode(node);
            for  child in node.children //where children is a vector of nodes
                q.enqueue(child);
        }
    }
    ```
    ```