Trees

CS 240 Spring 19

Binary Search Trees

Problems with Sequences

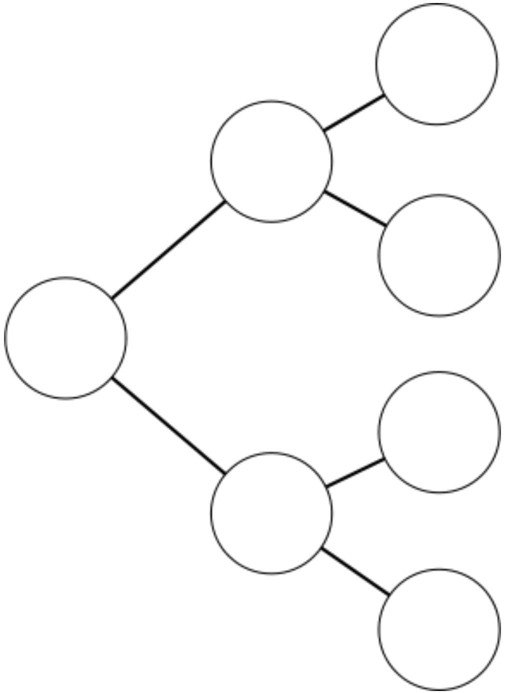
- What are some limitations of sequences?
    - searching
        - *O(n):*
            - length of the list
    - sorting
        - *best case: n logn*

A Better ADT

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

A Tree is a Linked List

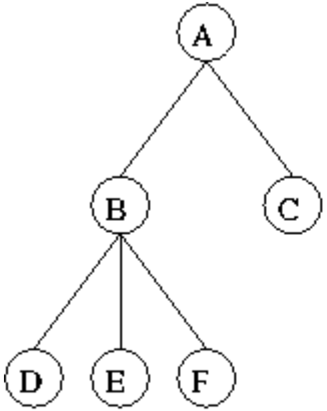
- If we add 1 additional ‘next’ pointer, we have a logarithmic traversal time
    - CRUD is much faster, but the data structure is more complex
- Problem: *we now need to know what* ***‘path’*** *to take*
    - Different kinds of trees are all solving this problem of what ‘path’ to take to find your data

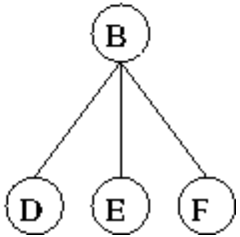


The Tree ADT

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.





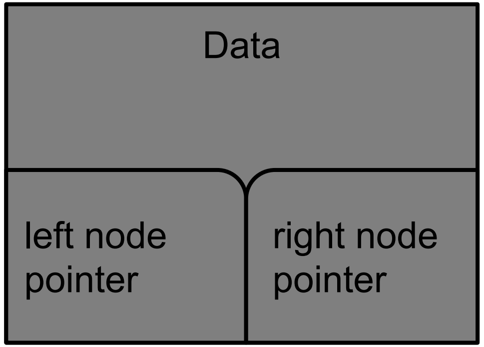
Terminology

- 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

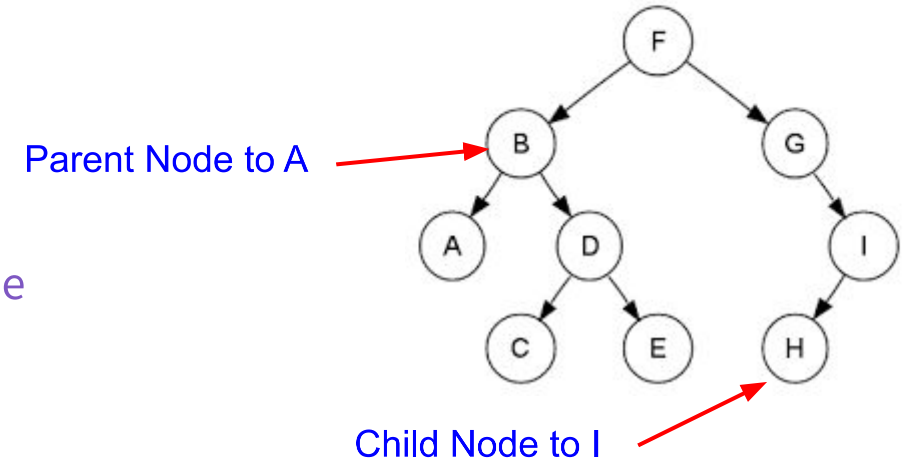
- max # of leaves
  - all nodes that do not have branches
- max # of nodes
  - all nodes in the tree
- Height for a tree
  - path depth
    - *The number of branches between the current node and the farthest leaf*
  - max depth
    - *The number of branches between the root node and and the farthest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

A Binary Node



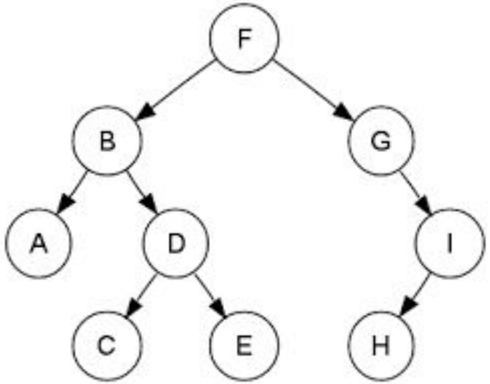
Parents and Children

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

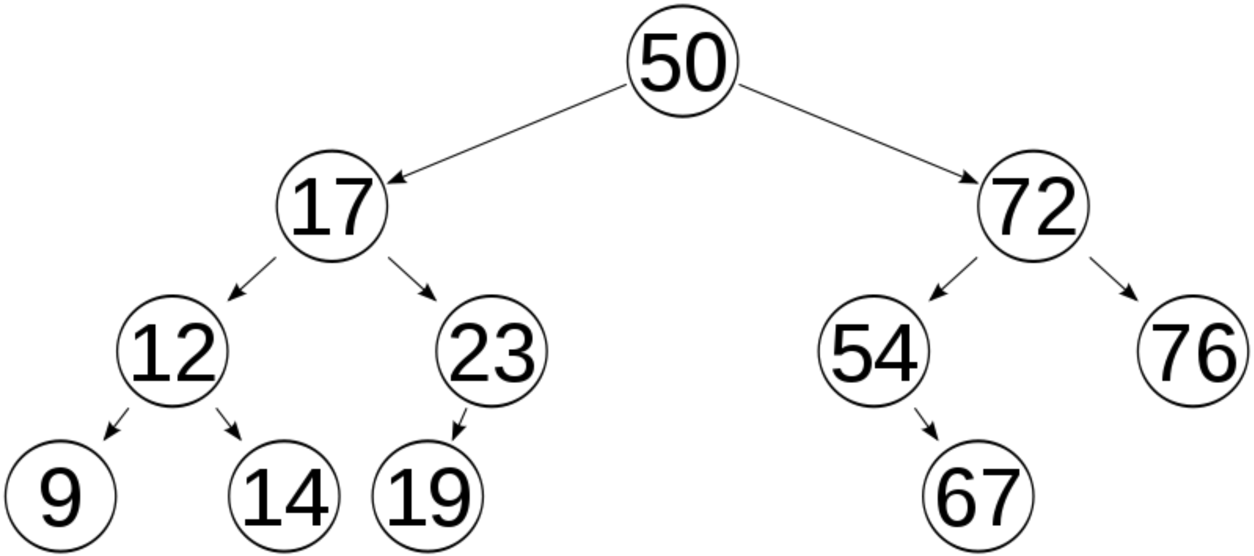


Height

- 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

- Binary Search tree is one of many different kinds of Tree Data Structures
    - 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)
- Pros:
  - Constant time Access
- Cons:
  - Complexity
  - Array Max Size must be known

- 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
- a parent field with a pointer to the parent node
- 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)*

Nested Classes

- A nested class is declared inside another enclosing class.
    - A nested class is a member object and has the same access rights as any other member.
- Member objects of an enclosing class don’t have any special access to member object of a nested class
    - the usual access rules (public, private) are enforced

Nested Class Example

```
class Enclosing {
  private:
        int x;

        class Nested {
            int y;
            void nestedFun(Enclosing *e) {
                std::cout<<e->x;  // works fine: nested class can access
            }
        }; // declaration Nested class ends here

}; // declaration Enclosing class ends here
```

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

Recursion has two parts

- Recursion requires two parts within the recursive function:
    - A base case that defines an atomic object
        - *an end to the recursion.*
    - A recursive step that defines how objects can be modified, reduced, or combined to produce another object closer to the atomic object
- 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

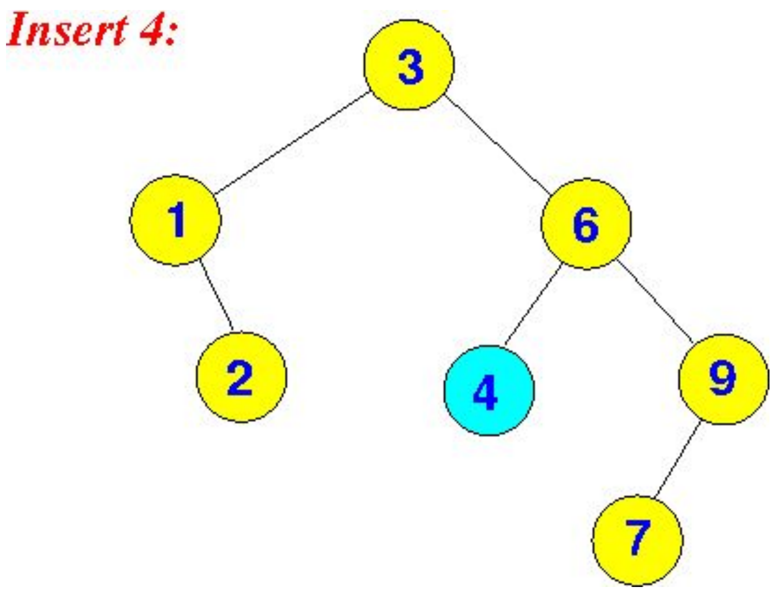
Tree Traversal

- Traversal, whether for insertion, deletion, or read, is much simpler with recursion
- 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

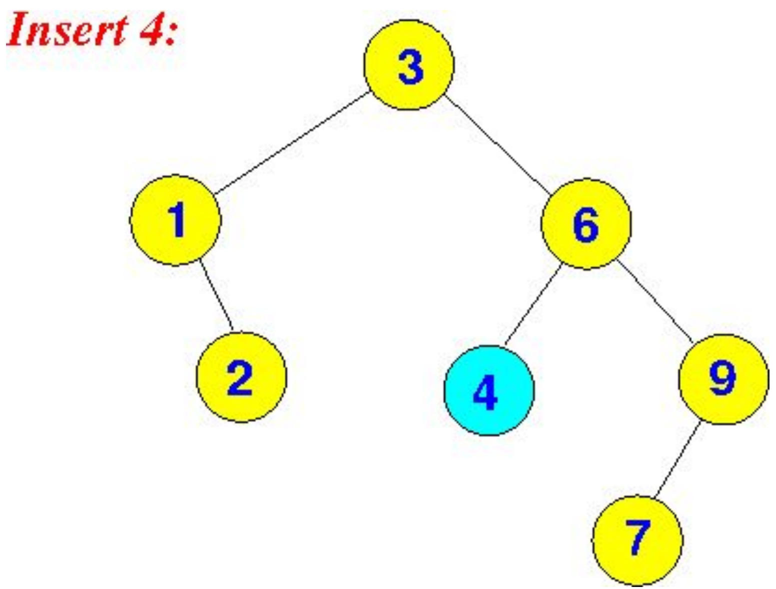


Inserting a node into an existing tree

- 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

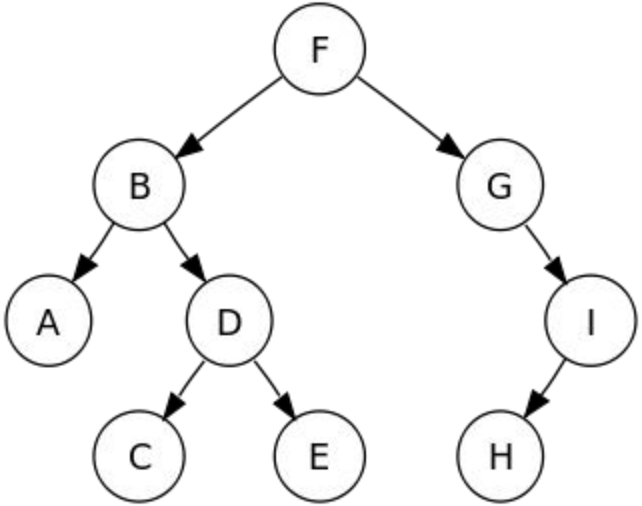
- How to choose the root node?
  - What happens if you choose a bad root node?
    - *Becomes a linked list*
- Insertion (with a well chosen root)
  - Best Case?
    - *O(1)*
  - Worst Case?
    - *O(logn)*

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)

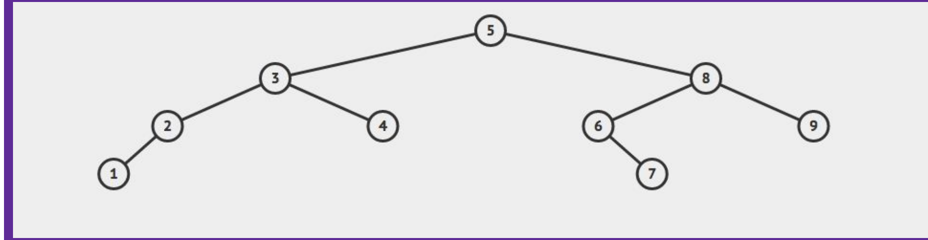
```
read(node, data){
        if(data == node.data)
                return node.data
        else if (data < node.data)
                if(node.left == NULL)
                        print("value not found");
                else
                        return  read(node.left, data)
        else if (data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



**must handle special case where tree is empty*

Classwork

Trees



find(Node[5],7)

find(Node[8],7)
find(Node[5],7)

find(Node[6],7)
find(Node[8],7)
find(Node[5],7)

find(Node[7],7)
find(Node[6],7)
find(Node[8],7)
find(Node[5],7)

Delete

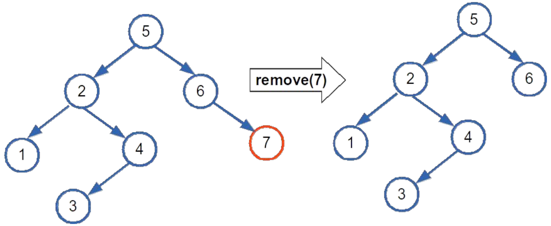
- Basic Deletion process
  - Remove the pointer(s) to the node
    - *Non-leaf nodes will need to update their children's parent pointers as well*
  - delete node
- Operations required
  - Tree Traversal to find the node
  - Must always keep track of the parent node
    - *This is where maintaining a parent pointer in the node is helpful*

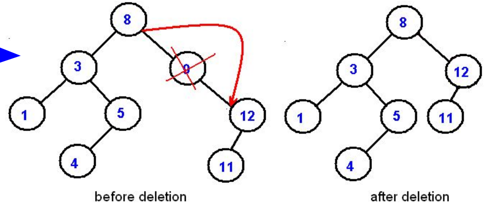
Delete - 3 Scenarios

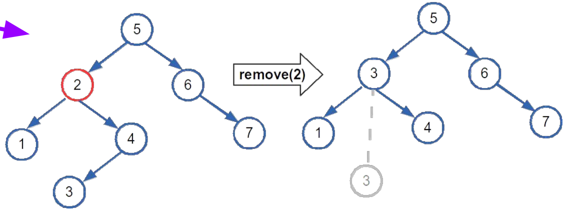
- What are the primary cases for deleting a node?
  - Delete a leaf node
    - *set parent node pointer to null*
    - *delete node*
  - Delete a 1 branch parent node
    - *Can't just delete the node, because then our tree would "fall apart."*
  - Delete a 2 branch parent node
    - *We must promote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

Delete Algorithm

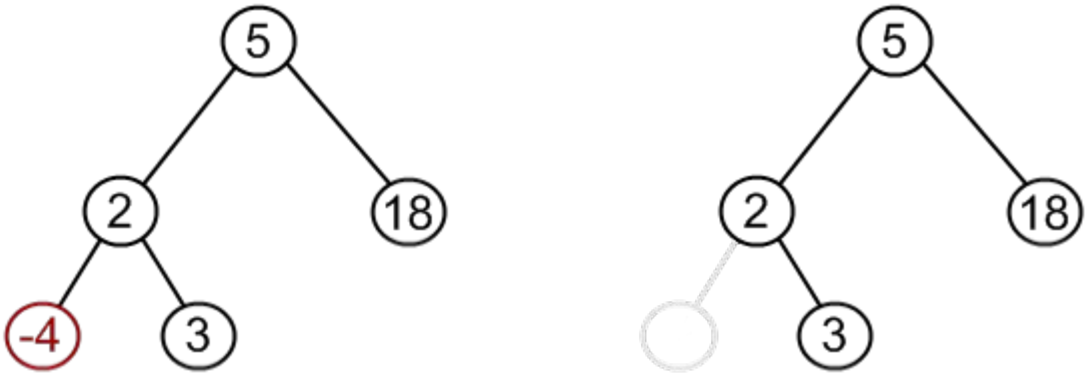
```
node = findNode(data);
if ( node not in BST )
      return;
else if ( node has no subtrees ){
        deleteLeaf(node);
}else if ( node has 1 tree ) {
        shortCircuit(node);
}else if(node has 2 subtrees){
        promotion(node);
}
```







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

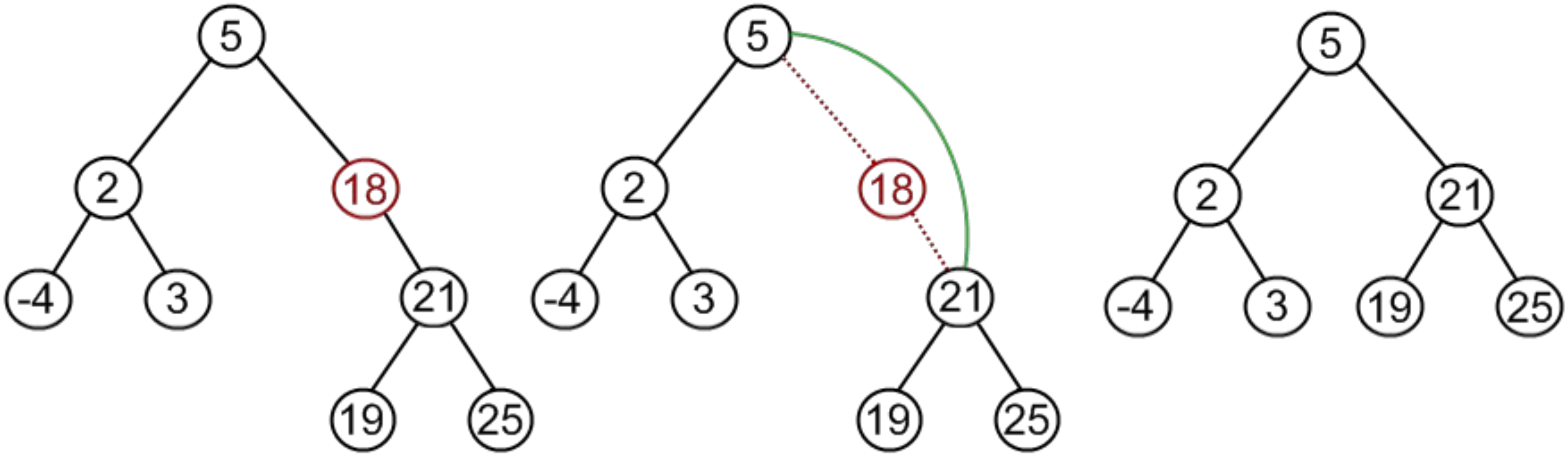
Short Circuit Algorithm

- 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



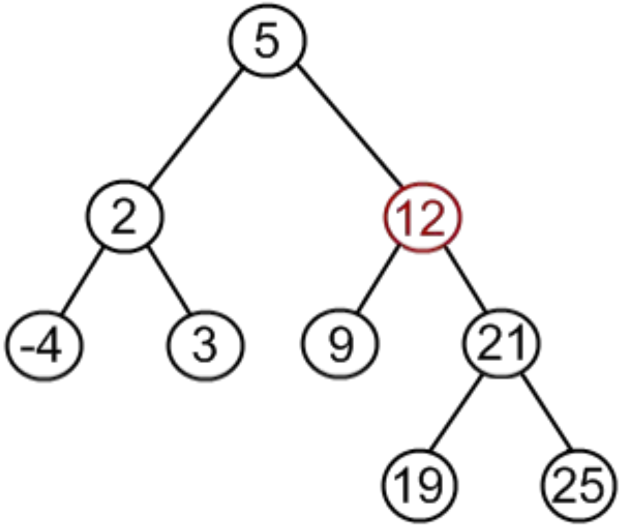
Short Circuit Pseudocode

```
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

Deleting 2 Child Nodes

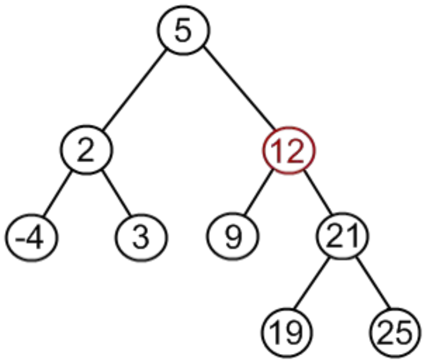


Promotion

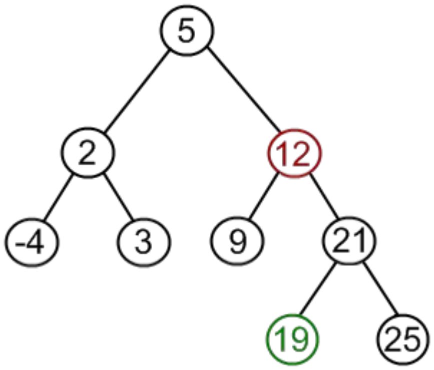
- We must promote a node to a higher space in the tree
- There are at most two possible candidates:
  - the rightmost child of the left subtree
    - *Traverse left once, then right as far as possible*
  - the leftmost child of the right subtree
    - *Traverse right once, then left as far as possible*
- 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

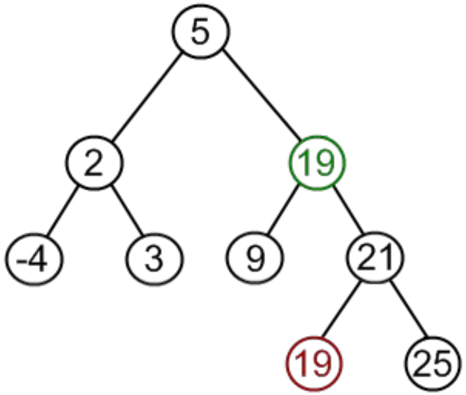
1) Find node to delete



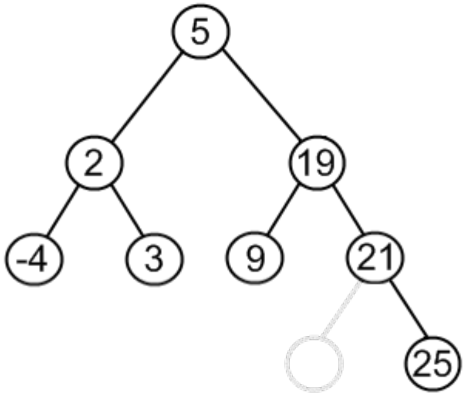
2) Find successor



3) Replace value



4) Delete Successor

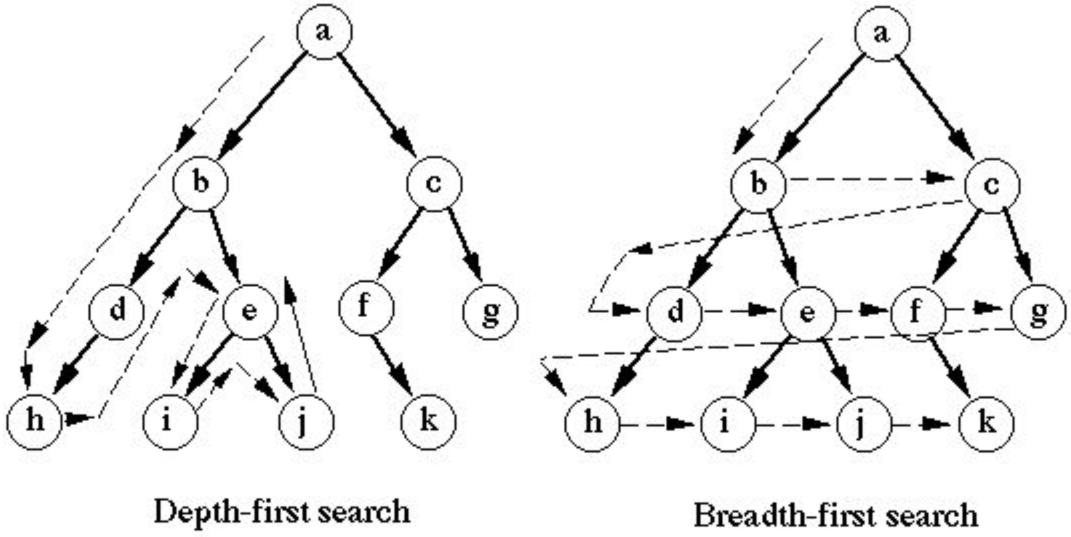


Promotion Selection

- 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){
	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);
	}
}
```



Searching a Tree

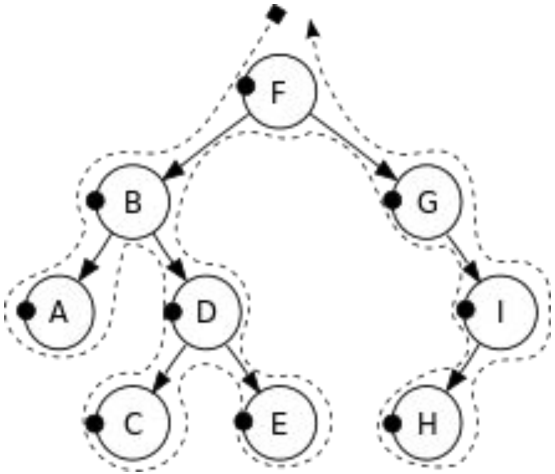
Depth First Traverse

- 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):

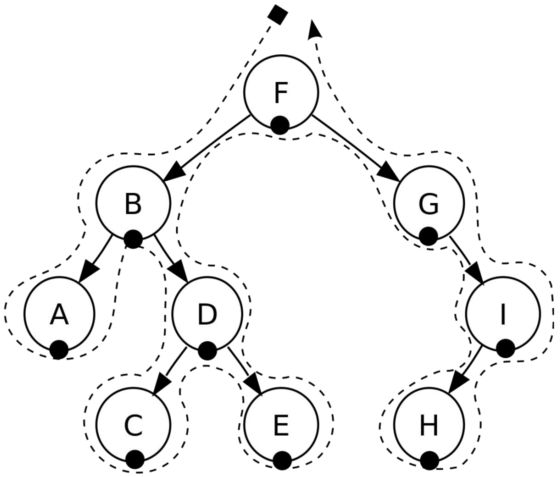
```
preorderTraversal(node):
    process(node)
    preorderTraversal(node->left)
    preorderTraversal(node->right)
```



InOrder

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

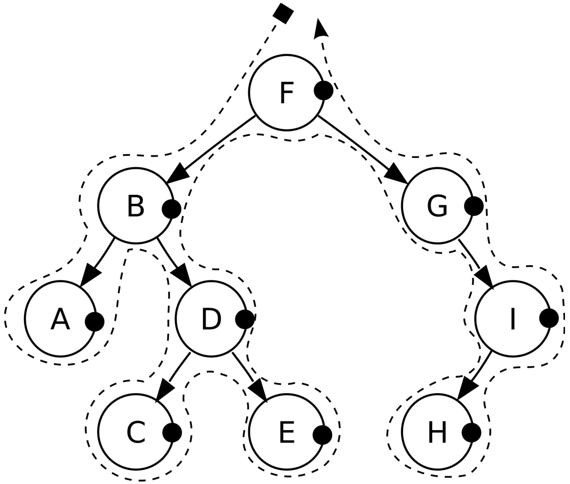
    ```
    inorderTraversal(node->left)
    process(node)
    inorderTraversal(node->right)
    ```



PostOrder

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

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



Classwork:
Tree Traversal

Use Cases

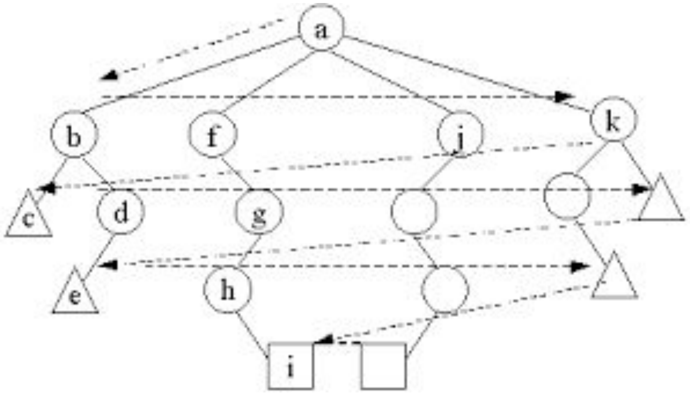
- Basic required Tree operations:
  - Deep Copy Tree
  - Delete Tree
  - Sorted Print
- Depth First Search
  - Traverse as far as possible down a single path
- Determine use case for each DFS Operation
  - *PreOrder:* copy of the tree
  - *InOrder:* gives nodes in non-decreasing order
  - *PostOrder*: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

Breadth First Search

- So far we have looked at depth first search
  - We traverse deep into the tree along a single path until we cannot go farther
- What is we want to traverse by level
  - Breadth First Traversal
    - *Also called Level order*
- Visit every node on a particular level before going to the next level

Breadth First Helper ADT

- How to Implement?
  - Not really a recursive algorithm
    - *Can still implement recursively, but not traditionally done with recursion*
  - Use a helper Data Structure to store the next level
- Which data structure would work best?
  - A Queue
    - *Enforces first in first out*



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(auto child : node.children) //where children is a vector of nodes
                q.enqueue(child);
        }
    }
    ```
    ```

Balancing a BST

- Binary Search Trees use **strict ordering**
    - There is only one place an inserted value can go
- In order to balance the tree, we will have to restructure it
    - There are many different variations of BST that ensures a balanced tree (AVL, Red/Black, Splay, etc)
- We are going to look at the simplest method of keeping a BST balanced

Creating a Balanced BST

- We first need to pull everything out of our tree so we can rebuild it
  - Use inorder traversal to sort the tree
  - Store in an array
- Divide our sorted array to insert in the optimal order
  - Remove the middle element of the array and insert into a new tree
  - Divide remaining array into two parts, left and right
    - *Repeat on both left and right array until the array is empty*
  - Return the new tree

Creating a Balanced BST