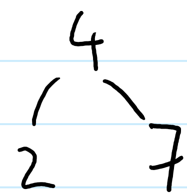
(01) Traversals

With a linear data structure like an array or a Linked List, it is assumed we display from beinning to end.

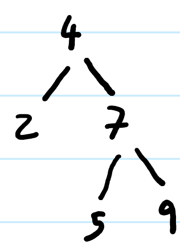
With a tree, we have choices. 3 conventions:



pre-order: *root, left, right*

in-order: *left, root, right*

post-order: *left, right, root*



pre-order: *root, left, right*

in-order: *left, root, right*

post-order: *left, right, root*

(02) BST: Node and Class

```
struct Node{
    int key;
    // Node * parent = nullptr; // to be added later
    Node* left = nullptr;
    Node* right = nullptr;
};
class BST{
    public:
        // Core public methods:
        BST();                          // default constructor
        BST(int data);                  // parameterized constructor
        ~BST();                         // destructor
        void insert(int data);          // insert a node a new element into tree
        bool searchKey(int key);        // search for a key in tree in the tree
        void deleteKey(int key);        // find and delete single element containing key value
        void printTree();               // function to print the tree

        // Extra methods:
        void removeRange(int low, int high);
        void print2DUtil(int space);
        bool isValidBST();
    private:
        Node* root;

        // Helper functions:
        //     Since root is a private member we need helper functions
        //     to access the root and traverse the trees recursively.
        Node* insertHelper(Node* currNode, int data);
        Node* searchKeyHelper(Node* currNode, int data);
        Node* deleteNodeHelper(Node *currNode, int value);
        void destroySubTree(Node *currNode);
        void printTreeHelper(Node* currNode);
        bool isBST(Node *root);
        Node* getMinValueNode(Node* currNode);
        Node* getMaxValueNode(Node* currNode);
        Node* createNode(int data);
};
```

(03) Insert node

Given a key value, insert a node to the next available location, ensuring that BST properties are maintained.

```
void BST::insert(int data){
    root = insertHelper(root, data);

}
```

```
Node* BST::insertHelper(Node* currentNode, int data){
    // Case 1: Either tree is empty
    // OR when we've reached the appropriate place to add new node
    if(currentNode== nullptr){
        return createNode(data);
    }
    // Case 2: new node has key value greater than or equal to current node
    else if(currentNode->key <= data) //
        currentNode->right = insertHelper(currentNode->right,data);
    // Case 3: new node has key value less than current node
    else if(currentNode->key > data)
        currentNode->left = insertHelper(currentNode->left, data);
    return currentNode;
}
```

Example:

Starting with an empty tree, insert nodes with values **8, 0, 9**

insert(8)
insertHelper(root,8)

Case 1:
1.
2.

(04) Adding a parent pointer

Let's say that we would like to update our data structure implementation such that every node has a parent pointer:

```
struct Node{
    int key;
    Node * parent = nullptr;
    Node* left = nullptr;
    Node* right = nullptr;
};
```

Updating the struct definition is trivial, but how do we update our **`insert`** method?

We need to figure out where to set the parent pointer during each insert operation.

(05) Insert node - parent pointer

Given a key value, insert a node to the next available location, ensuring that BST properties are maintained.

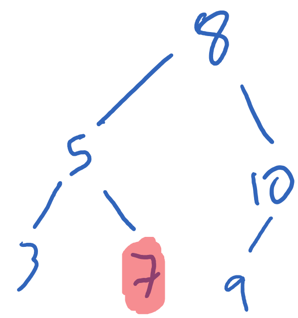
```
void BST::insert(int data){
    root = insertHelper(root, data);

}
```

```
Node *BST::insertHelper(Node * currentNode, int data){
// Case 1. Either tree is empty (if fisrt call to insert Helper)
// OR when we've reached the appropriate place to add new node

    if(currentNode== nullptr){
       return createNode(data);
    }
// Case 2. new node has key value greater or equal to the current node
//
    else if(currentNode->key < data){
        currentNode->right = insertHelper(currentNode->right,data);

    }
// Case 3. new node has key value less than current node
    else if(currentNode->key > data){
        currentNode->left = insertHelper(currentNode->left,data);

    }
    return currentNode;
}
```

(06) BST-Delete 0

Given a tree, delete a node.

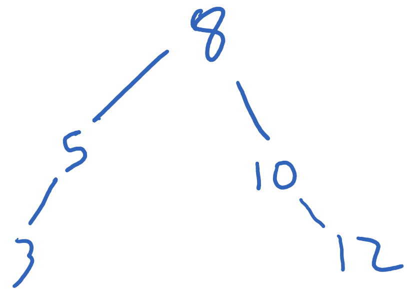
```
delete(value)
        root = delHlp(root,value)

Node* delHlp(currNode, value)
if(currNode == nullptr)
        // end of branch (or tree was empty)
        return null
else if( value < currNode->key )
        currNode->left = delHlp(currNode->left, value)
else if(value > currNode->key)
        currNode->right =  delHlp(currNode->right, value)
else // node found:
// 4 possible cases
   A.   Node has no children
           a.   delete node
           b.   return nullptr
   B.   Node has only right child
              ...
   C.   Node has only left child
              ...
   D.   Node has 2 children
              ...
return currNode
```



example: delete(7)

(07) BST-Delete 1

Given a tree, delete a node.



delete(10)

delete(value)
    root = delHlp(root,value)

Node* delHlp(currNode, value)
    if (currNode == nullptr)
    - end of a branch (or tree was empty)
    - return null

    else if (value < currNode)
        currNode->left= delHlp(currNode->left, value)
    else if (value > currNode)
        currNode->right = delHlp(currNode->right , value)
    else // node found:
    - 4 possible cases:
        A. Node has no children
            - delete node
            - return nullptr
        B. Node has only right child
            Node *p_right = currNode->right
            delete currNode
            return p_right

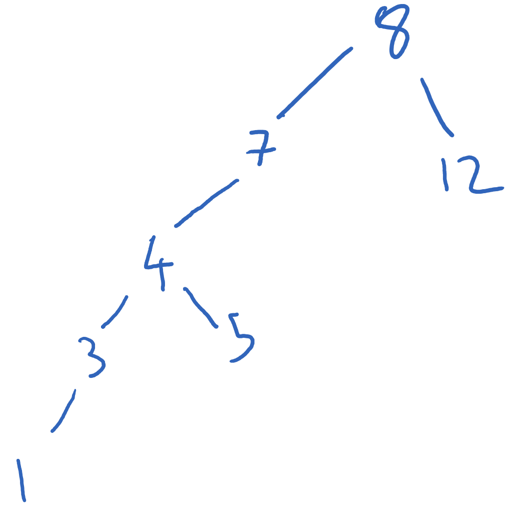
        C. Node only has left child
            - ??
        D. Node has 2 children
            - ???

    return currNode

(08) BST-Delete 2

Given a tree, delete a node.

```
delete(value)
    root = delHlp(root,value)

Node* delHlp(currNode, value)
    if (currNode == nullptr)
        ○ end of a branch (or tree was empty)
        ○ return null
    else if (value < currNode)
        currNode->left= delHlp(currNode->left, value)
    else if (value > currNode)
        currNode->right = delHlp(currNode->right , value)
    else // node found:
        ○ 4 possible cases:
            A. Node has no children
                □ delete node
                □ return nullptr
            B. Node has only right child
                *p_Right = currNode->right
                delete currNode
                return p_Right
            C. Node only has left child
                *p_Left= currNode->left
                delete currNode
                return p_left

            D. Node has 2 children
                □ ???
    return currNode
```

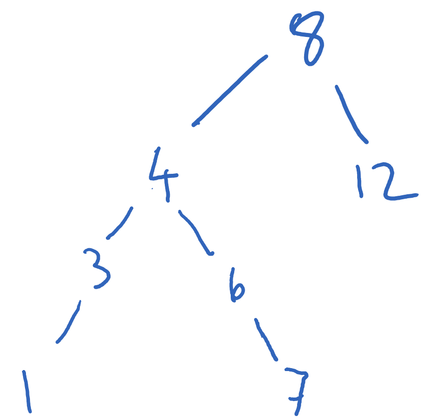


example
delete(7)

(09) BST-Delete 3

Given a tree, delete a node.

```
delete(value)
    root = delHlp(root,value)

Node* delHlp(currNode, value)
    if (currNode == nullptr)
        ○ end of a branch (or tree was empty)
        ○ return null
    else if (value < currNode)
        currNode->left= delHlp(currNode->left, value)
    else if (value > currNode)
        currNode->right = delHlp(currNode->right , value)
    else // node found:
        ○ 4 possible cases:
            A. Node has no children
                □ delete node
                □ return nullptr
            B. Node has only right child
                *p_Right = currNode->right
                delete currNode
                return p_Right
            C. Node only has left child
                *p_Left= currNode->left
                delete currNode
                return p_left

            D. Node has 2 children
               // find the minimum of the right subtree:
               *r_s_m = getMin(currNode->right)
               currNode->key = r_s_m->key
                currNode->right =  delHlp(currNode->right, r_s_m->key)


    return currNode
```

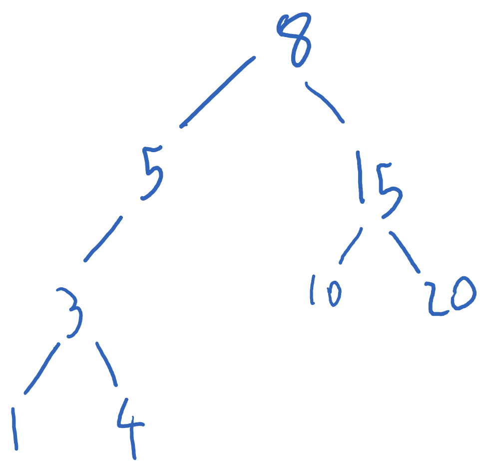


example
delete(4)
root = delHlp(root,4)

(10) Destructor 0

Recall, the different traversal orders.

What if we try to delete in-order? (Left, root, right)

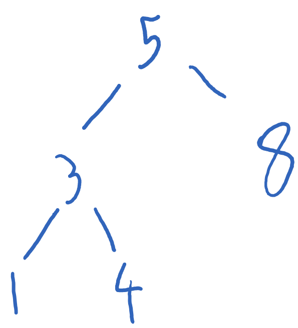


Pre-order? (root, left, right)

Post-order (*left, right, root*)

(11) Destructor 1

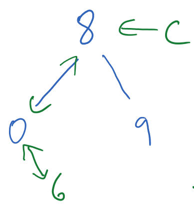
```
// Destructor
BST::~BST(){
    destroySubTree(root);
    root = nullptr;
}

void BST::destroySubTree(Node *currNode){
    if(currNode!=nullptr)
    {
        destroySubTree(currNode->left);
        destroySubTree(currNode->right);
        delete currNode;
    }
}
```




```
dst(root)
--dst(*3)
----dst(*1)
------dst(null)
----dst(*1)
------dst(null)
----dst(*1)
------delete(*1)
--dst(*3)
----dst(*4)
------dst(null)
----dst(*4)
------dst(null)
----dst(*4)
------delete(*4)
--dst(*3)
----delete(*3)
dst(root)
--dst(*8)
----dst(null)
--dst(*8)
----dst(null)
--dst(*8)
----delete(*8)
dst(root)
--delete(root)
```

(Extra) Insert Recap

insert(6)



Insert (6)

root = insertHelper (*8 , 6)

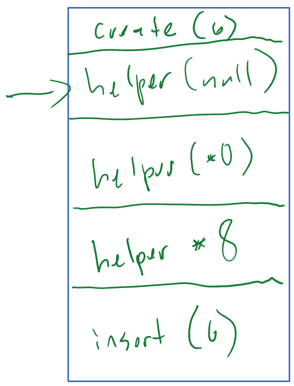
"suspended"

C->left = helper (*0 , 6)

C->right = helper (null , 6)

C = create (6)

C->right




```
Node *BST::insertHelper(Node * currentNode, int data){
// Case 1. Either tree is empty (if fisrt call to insert Helper)
// OR when we've reached the appropriate place to add new node

    if(currentNode== nullptr){
      return createNode(data);
    }
// Case 2. new node has key value greater or equal to the current node
//
    else if(currentNode->key < data){
        currentNode->right = insertHelper(currentNode->right,data);
        currentNode->right->parent = currentNode;
    }
// Case 3. new node has key value less than current node
    else if(currentNode->key > data){
        currentNode->left = insertHelper(currentNode->left,data);
        currentNode->left->parent = currentNode;
    }
    return currentNode;
}
```