# GAME2001 Data Structures and Algorithms

## Fall 2020

# Week 12

## Trees

## Trees

- Used for maintaining a hierarchy of data
- Some trees offer
  - Fast searches
  - Fast insertion
  - Fast deletion
  - Fast resizing
  - Can store ordered data easily

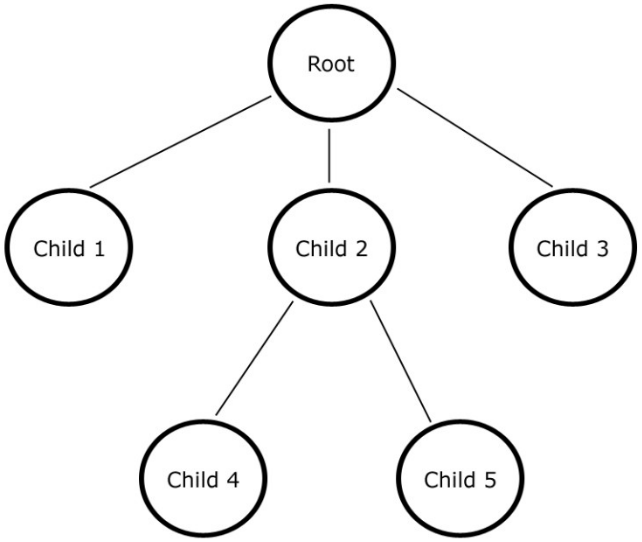
STL does <u>not</u> have trees.

## Trees

- Many different types:
  - General trees
  - Binary trees
  - kd-trees
  - B-trees
  - 2-3 trees
  - 2-3-4 trees
  - AVL trees
  - Red-black trees
  - Heaps

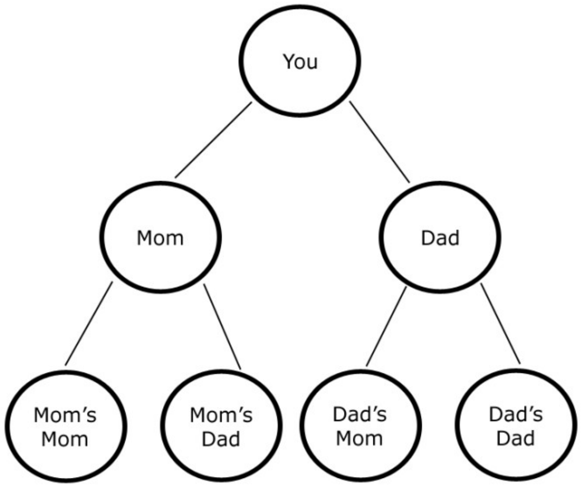
## Trees

- a data structure that forms some kind of meaningful hierarchy
- start off with what is called the root
  - the first node in a link list in the sense that it is the starting point of the data structure's container
- made up of a hierarchy of nodes that are connected by what are called edges



## Trees

- a tree is a type of graph
- is made up of nodes and edges
- the relationship between nodes is a parent-child relationship



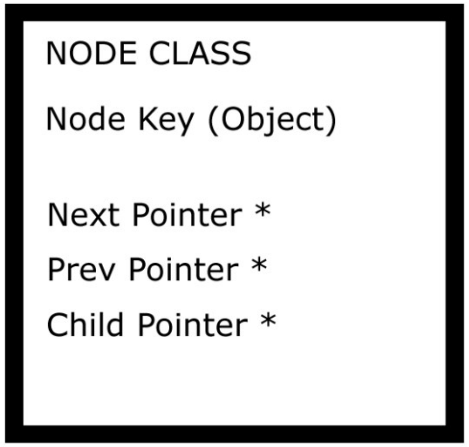
each generation is a different level

## Trees

- Leaf node
  - a type of node that has no children nodes attached to it
- Leafless node
  - a node with one or more children attached
- Key
  - used to determine how nodes are to be inserted into a tree
- Traversing
  - the process of moving through the nodes of the tree, normally done to perform some set of algorithms on the tree (in-order, pre-order, or post-order)
- Parent node
  - like the previous node pointer in a link list in the sense that it is the node that the current node is attached to
- Sibling nodes
  - nodes that share the same parent node

## General Tree

- any number of children per node
  - create a node class that has both a child pointer and a sibling pointer
  - child pointer is used to point to the first child node,
  - any additional children of a node can be accessed through the sibling pointer of the first child and so on, thus creating a link list



NODE CLASS

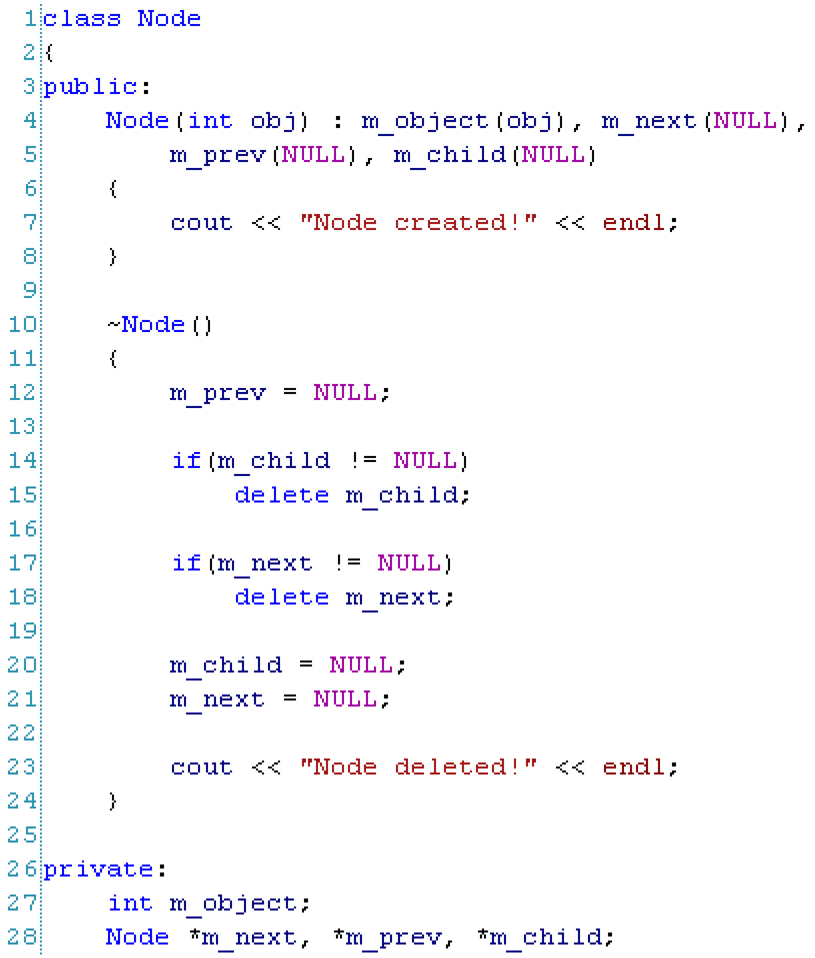
Node Key (Object)

Next Pointer *

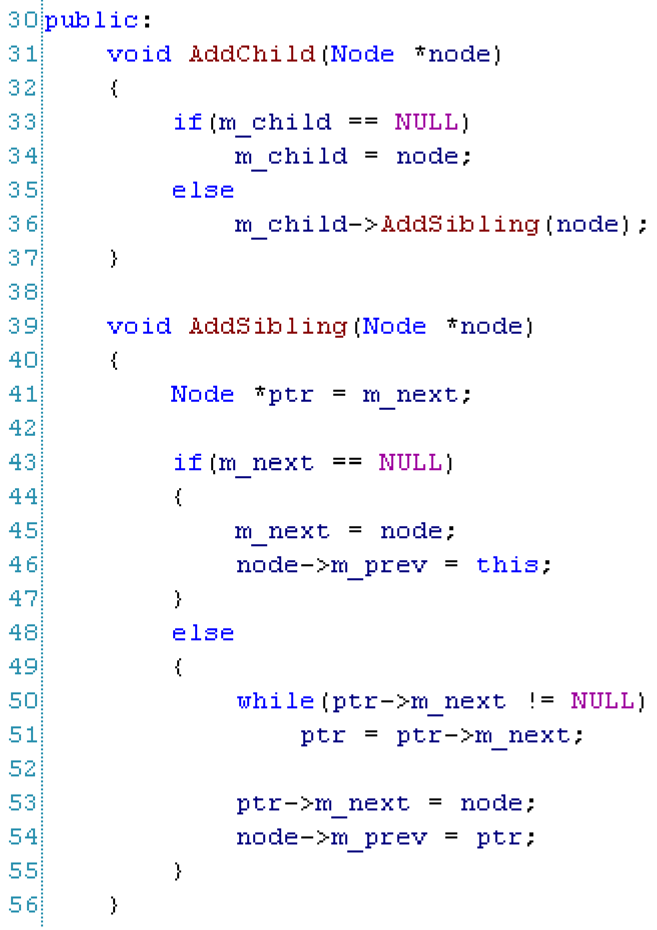
Prev Pointer *

Child Pointer *

## General Tree

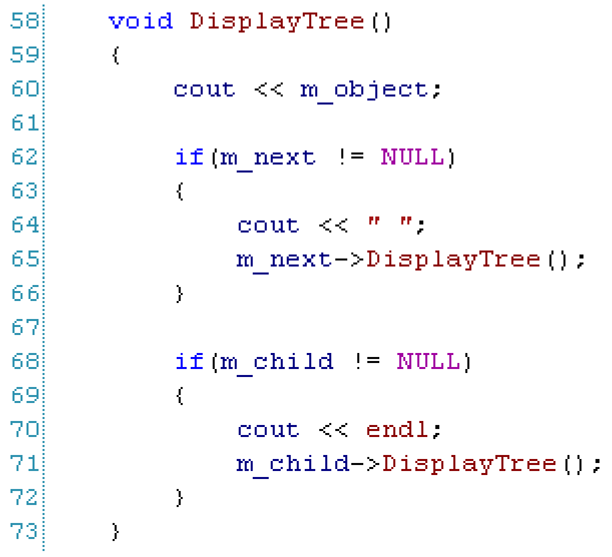


```cpp
class Node
{
public:
    Node(int obj) : m_object(obj), m_next(NULL),
        m_prev(NULL), m_child(NULL)
    {
        cout << "Node created!" << endl;
    }

    ~Node()
    {
        m_prev = NULL;

        if(m_child != NULL)
            delete m_child;

        if(m_next != NULL)
            delete m_next;

        m_child = NULL;
        m_next = NULL;

        cout << "Node deleted!" << endl;
    }

private:
    int m_object;
    Node *m_next, *m_prev, *m_child;
```

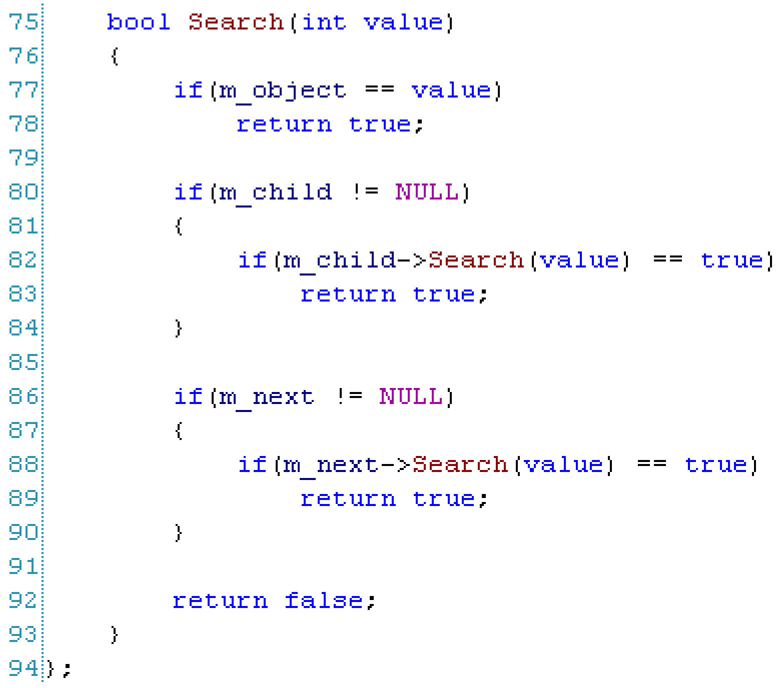


```cpp
public:
    void AddChild(Node *node)
    {
        if(m_child == NULL)
            m_child = node;
        else
            m_child->AddSibling(node);
    }

    void AddSibling(Node *node)
    {
        Node *ptr = m_next;

        if(m_next == NULL)
        {
            m_next = node;
            node->m_prev = this;
        }
        else
        {
            while(ptr->m_next != NULL)
                ptr = ptr->m_next;

            ptr->m_next = node;
            node->m_prev = ptr;
        }
    }
```

## General Tree

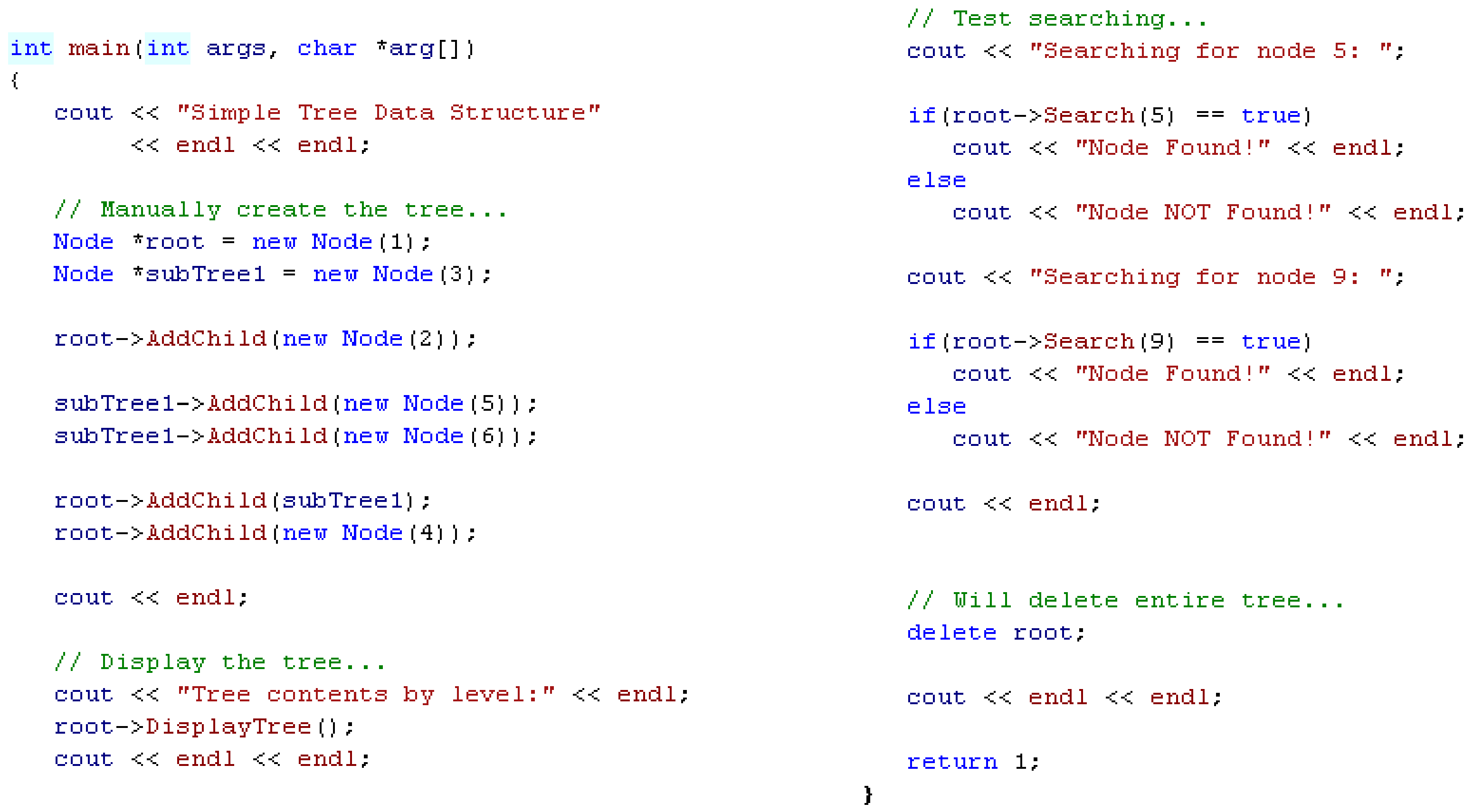


```cpp
    void DisplayTree()
    {
        cout << m_object;

        if(m_next != NULL)
        {
            cout << " ";
            m_next->DisplayTree();
        }

        if(m_child != NULL)
        {
            cout << endl;
            m_child->DisplayTree();
        }
    }
```



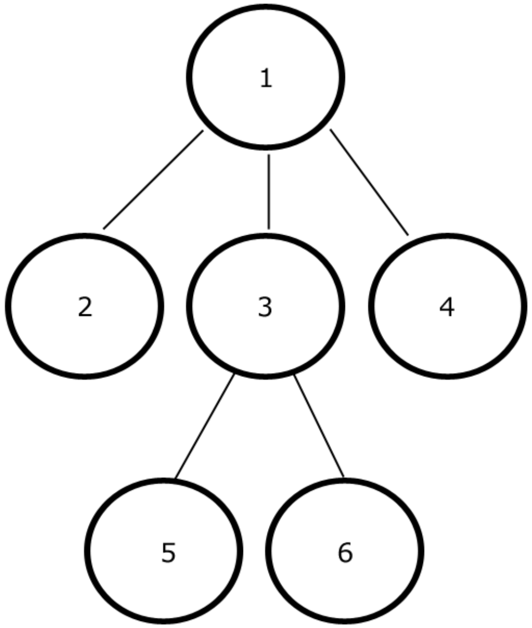
```cpp
    bool Search(int value)
    {
        if(m_object == value)
            return true;

        if(m_child != NULL)
        {
            if(m_child->Search(value) == true)
                return true;
        }

        if(m_next != NULL)
        {
            if(m_next->Search(value) == true)
                return true;
        }

        return false;
    }
};
```

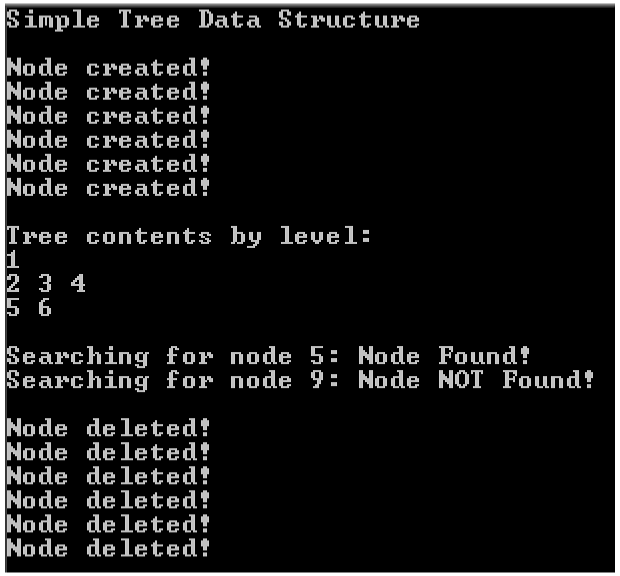
## General Tree Example



```
int main(int args, char *arg[])
{
    cout << "Simple Tree Data Structure"
         << endl << endl;

    // Manually create the tree...
    Node *root = new Node(1);
    Node *subTree1 = new Node(3);

    root->AddChild(new Node(2));

    subTree1->AddChild(new Node(5));
    subTree1->AddChild(new Node(6));

    root->AddChild(subTree1);
    root->AddChild(new Node(4));

    cout << endl;

    // Display the tree...
    cout << "Tree contents by level:" << endl;
    root->DisplayTree();
    cout << endl << endl;

    // Test searching...
    cout << "Searching for node 5: ";

    if(root->Search(5) == true)
        cout << "Node Found!" << endl;
    else
        cout << "Node NOT Found!" << endl;

    cout << "Searching for node 9: ";

    if(root->Search(9) == true)
        cout << "Node Found!" << endl;
    else
        cout << "Node NOT Found!" << endl;

    cout << endl;


    // Will delete entire tree...
    delete root;

    cout << endl << endl;

    return 1;
}
```

## General Tree Example



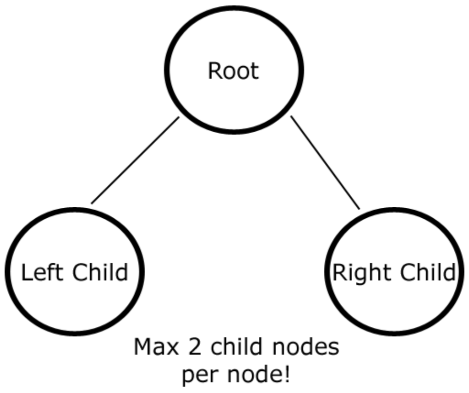


```
Simple Tree Data Structure

Node created!
Node created!
Node created!
Node created!
Node created!
Node created!

Tree contents by level:
1
2 3 4
5 6

Searching for node 5: Node Found!
Searching for node 9: Node NOT Found!

Node deleted!
Node deleted!
Node deleted!
Node deleted!
Node deleted!
Node deleted!
```

# Binary Trees

- maximum of two child nodes per node
  - often referred to as the left and right child nodes
    - left child is less than the parent, right child is greater than parent

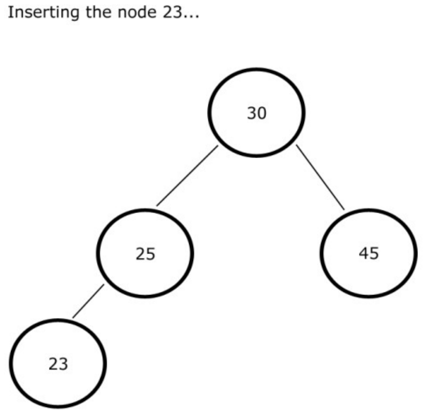
An example of a binary tree node with 2 children...



- fast insertion, deletion and searching

## Binary Trees

- Insertion
  - Node placement depends on the key
  - If less than the root node its placed on the left side, otherwise placed on right side



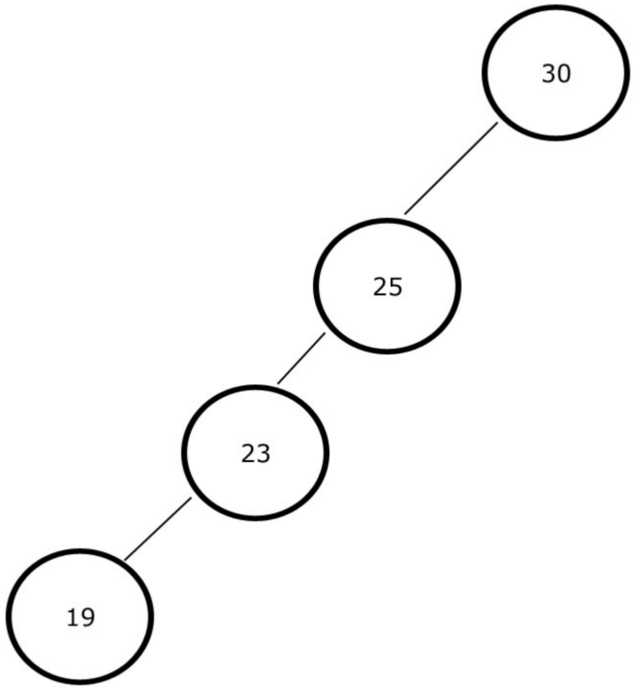
C = log2(N – 1)  max comparisons to insert into correct position

## Binary Trees

- Maximum number of comparisons depends on if the tree is balanced or not
- Balanced tree
  - means that on average there are as many left nodes as there are right nodes
- Unbalanced tree
  - a tree with uneven sides
  - take longer to process

## Binary Trees

An example of a worst case unbalanced tree...



## Binary Trees

- Searching
  - Real fast
  - As trees grow, the difference in searching efficiency becomes extremely large
  - If the key is lower than the key of the current node, then the object, if it exists in the tree, would be on the left branch
  - Otherwise it will be on the right branch

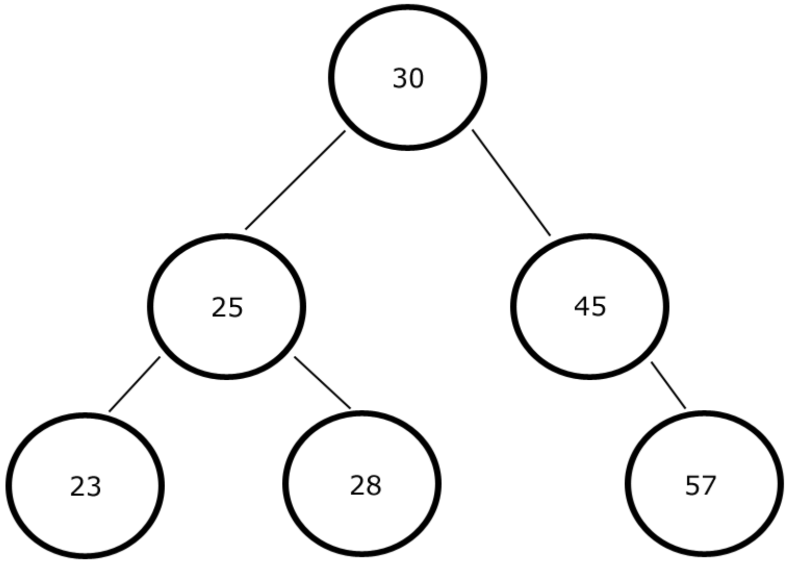
## Binary Trees

- Finding the minimum key
  - start at the root node and continue moving through <u>left</u> pointers until you reach a node with no left node
  - once there, you know that you have arrived at the minimum key value in the tree

- Finding the maximum key
  - start at the root node and continue moving through <u>right</u> pointers until you reach a node with no right node
  - once there, you know that you have arrived at the maximum key value in the tree

## Binary Trees

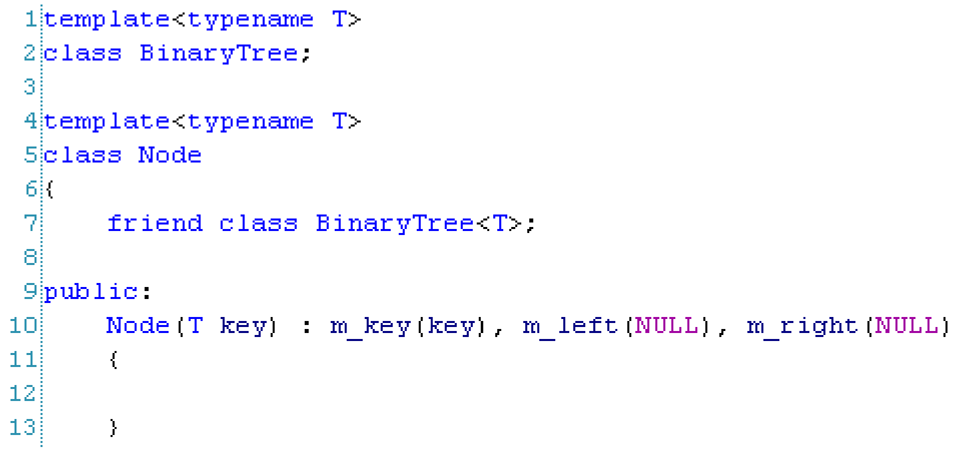
The min value is the left- most value of 23.
The max value is the right-most value of 57.



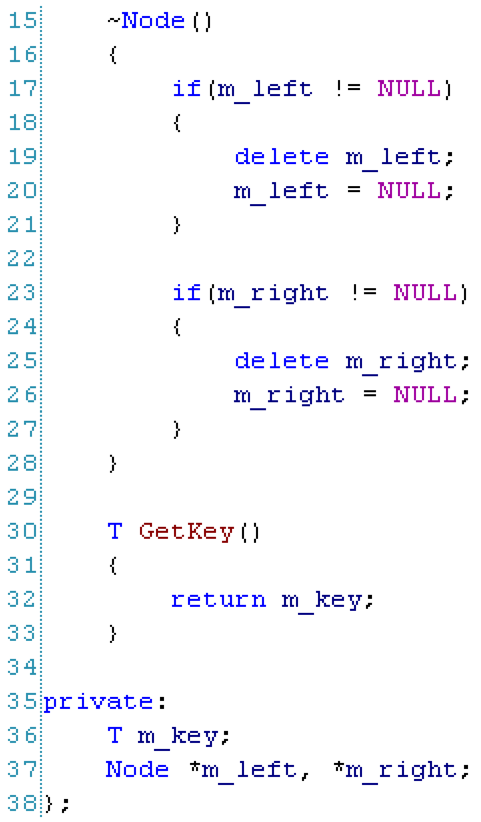
## Binary Trees

- Removal
  1) No children: just delete the node
  2) One child: just replace the node with the child
  3) Two children: replace the node with the in-order successor or the in-order predecessor

## Binary Trees

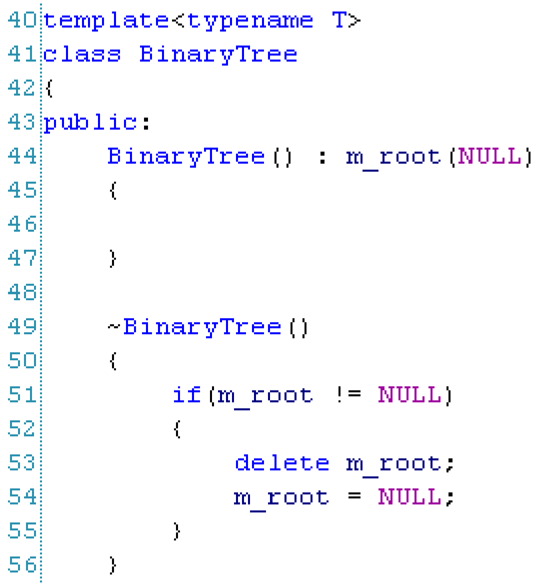


```
template<typename T>
class BinaryTree;

template<typename T>
class Node
{
    friend class BinaryTree<T>;

public:
    Node(T key) : m_key(key), m_left(NULL), m_right(NULL)
    {

    }
```



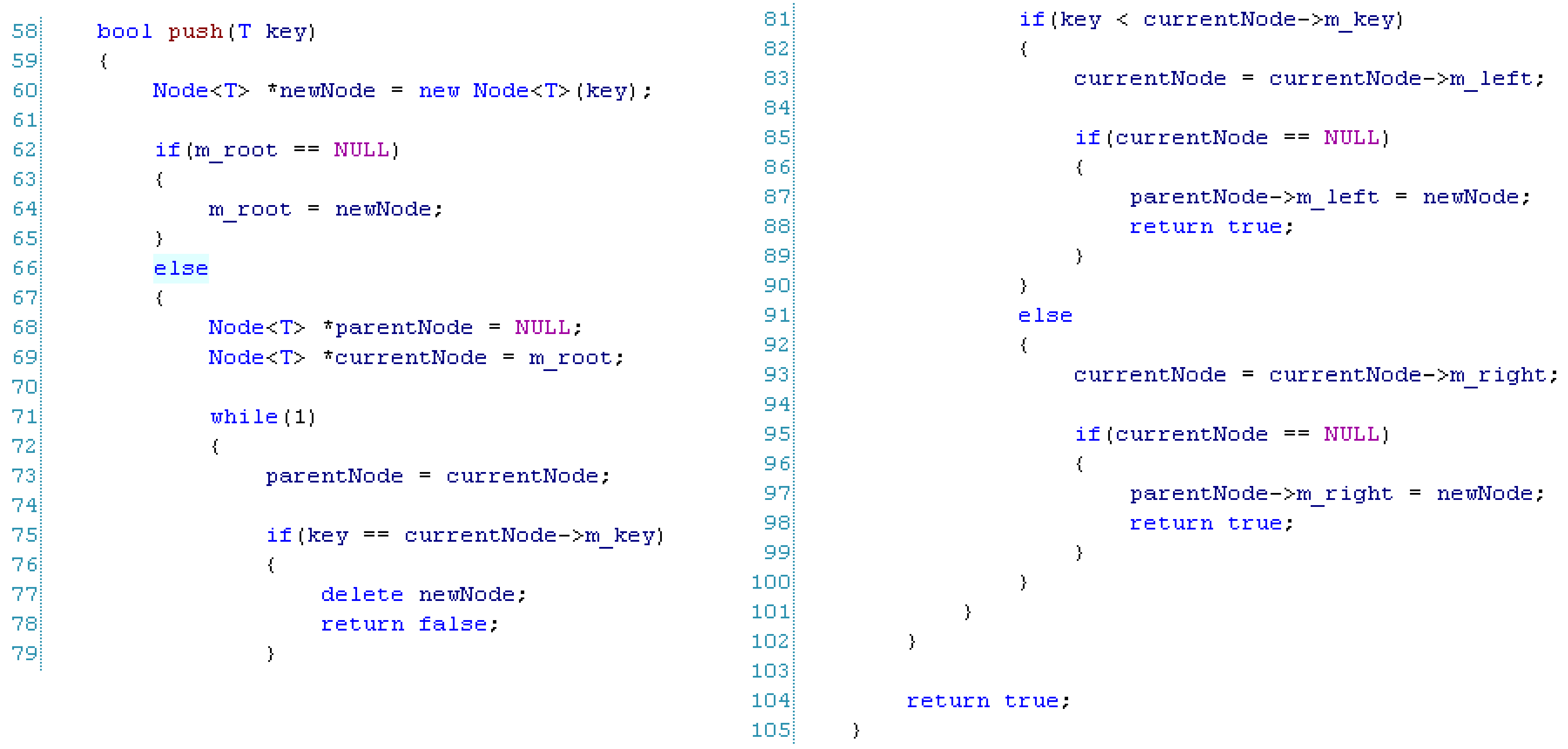
```
    ~Node()
    {
        if(m_left != NULL)
        {
            delete m_left;
            m_left = NULL;
        }

        if(m_right != NULL)
        {
            delete m_right;
            m_right = NULL;
        }
    }

    T GetKey()
    {
        return m_key;
    }

private:
    T m_key;
    Node *m_left, *m_right;
};
```

## Binary Trees



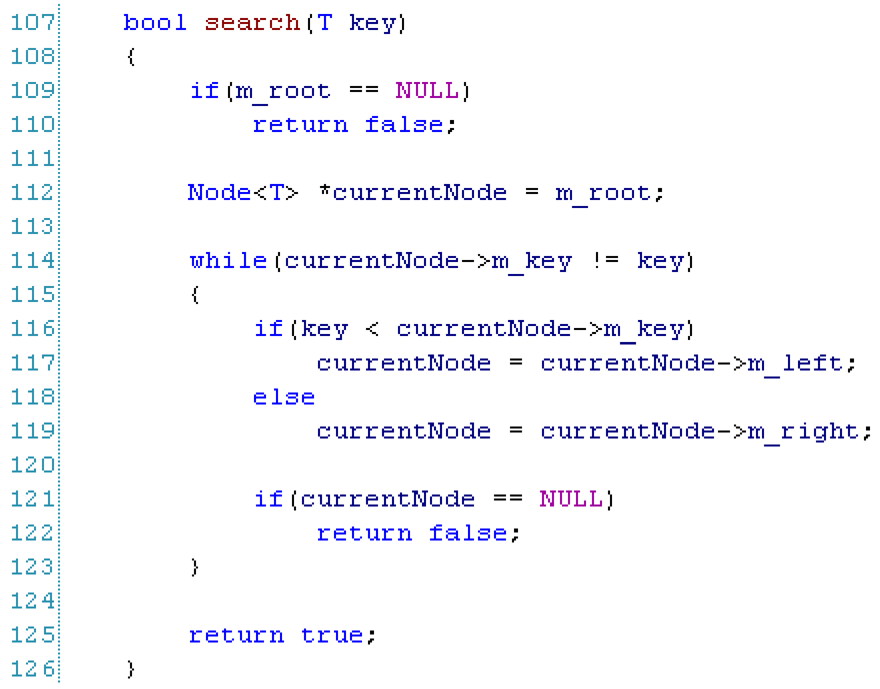
```
template<typename T>
class BinaryTree
{
public:
    BinaryTree() : m_root(NULL)
    {

    }

    ~BinaryTree()
    {
        if(m_root != NULL)
        {
            delete m_root;
            m_root = NULL;
        }
    }
```

## Binary Trees



```
bool push(T key)
{
    Node<T> *newNode = new Node<T>(key);

    if(m_root == NULL)
    {
        m_root = newNode;
    }
    else
    {
        Node<T> *parentNode = NULL;
        Node<T> *currentNode = m_root;

        while(1)
        {
            parentNode = currentNode;

            if(key == currentNode->m_key)
            {
                delete newNode;
                return false;
            }

            if(key < currentNode->m_key)
            {
                currentNode = currentNode->m_left;

                if(currentNode == NULL)
                {
                    parentNode->m_left = newNode;
                    return true;
                }
            }
            else
            {
                currentNode = currentNode->m_right;

                if(currentNode == NULL)
                {
                    parentNode->m_right = newNode;
                    return true;
                }
            }
        }
    }

    return true;
}
```

# Binary Trees



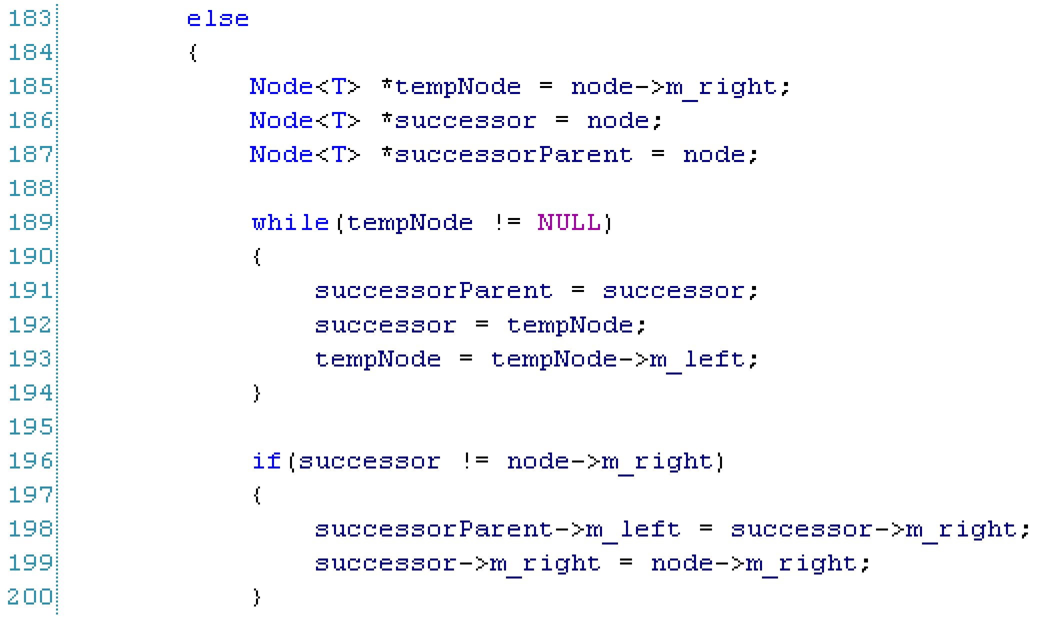
```
bool search(T key)
{
    if(m_root == NULL)
        return false;

    Node<T> *currentNode = m_root;

    while(currentNode->m_key != key)
    {
        if(key < currentNode->m_key)
            currentNode = currentNode->m_left;
        else
            currentNode = currentNode->m_right;

        if(currentNode == NULL)
            return false;
    }

    return true;
}
```

# Binary Trees

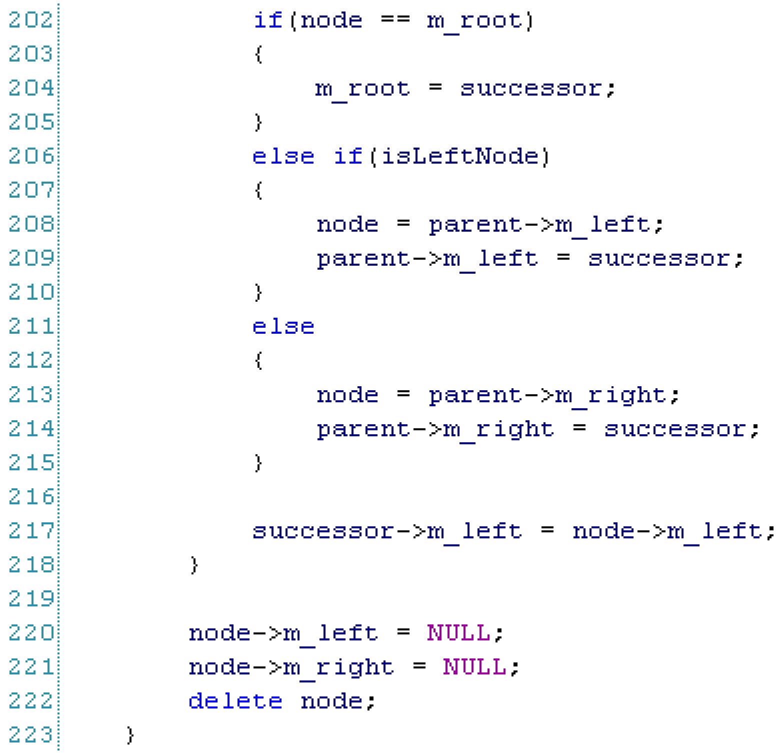
```
    void remove(T key)
    {
        if(m_root == NULL)
            return;

        Node<T> *parent = m_root;
        Node<T> *node = m_root;
        bool isLeftNode = false;

        while(node->m_key != key)
        {
            parent = node;

            if(key < node->m_key)
            {
                node = node->m_left;
                isLeftNode = true;
            }
            else
            {
                node = node->m_right;
                isLeftNode = false;
            }

            if(node == NULL)
                return;
        }
```

```
        if(node->m_left == NULL && node->m_right == NULL)
        {
            if(node == m_root)
                m_root = NULL;
            else if(isLeftNode == true)
                parent->m_left = NULL;
            else
                parent->m_right = NULL;
        }
        else if(node->m_left == NULL)
        {
            if(node == m_root)
                m_root = node->m_right;
            else if(isLeftNode == true)
                parent->m_left = node->m_right;
            else
                parent->m_right = node->m_right;
        }
        else if(node->m_right == NULL)
        {
            if(node == m_root)
                m_root = node->m_left;
            else if(isLeftNode == true)
                parent->m_left = node->m_left;
            else
                parent->m_right = node->m_left;
        }
```

# Binary Trees

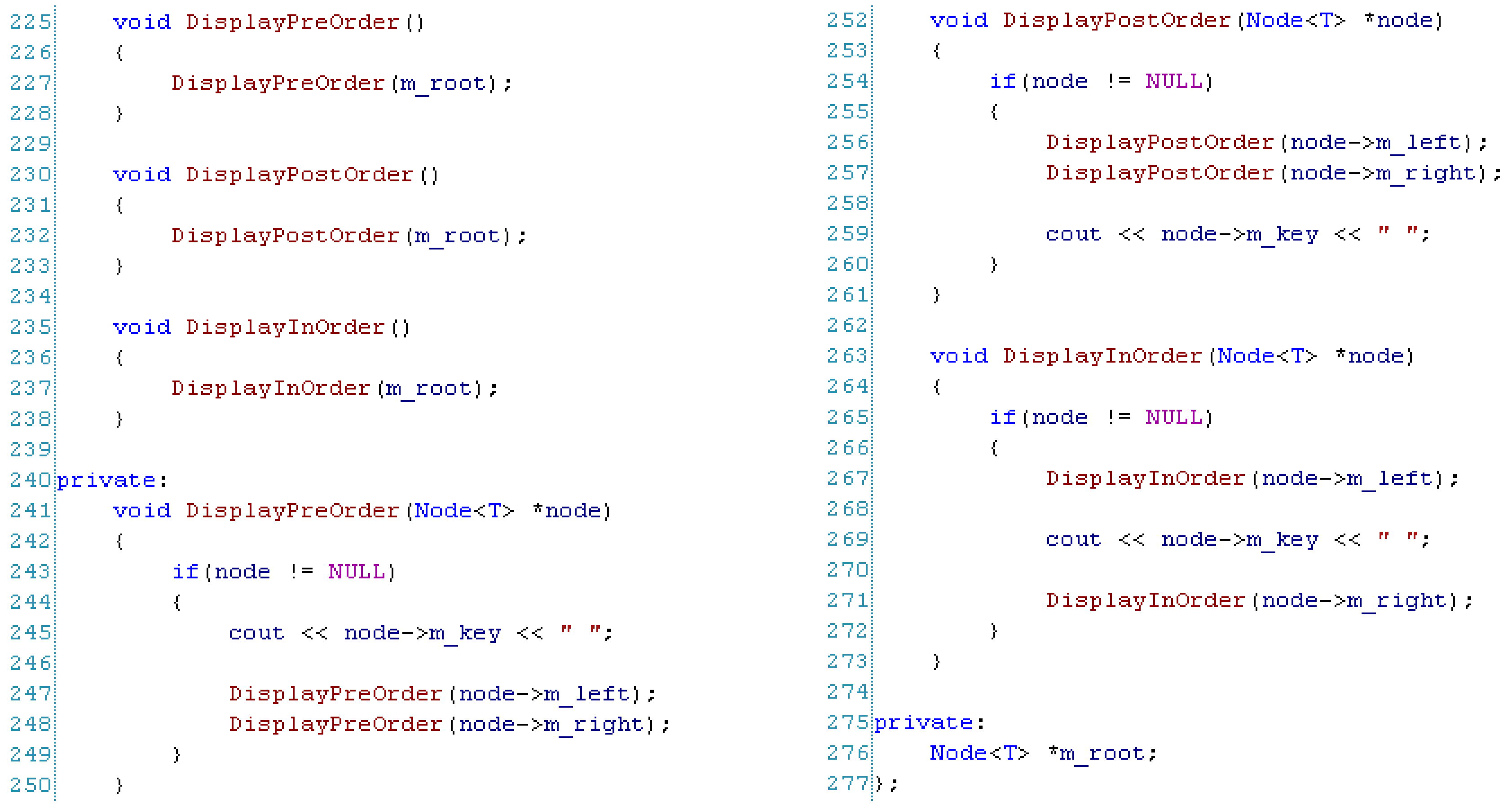


```cpp
        else
        {
            Node<T> *tempNode = node->m_right;
            Node<T> *successor = node;
            Node<T> *successorParent = node;

            while(tempNode != NULL)
            {
                successorParent = successor;
                successor = tempNode;
                tempNode = tempNode->m_left;
            }

            if(successor != node->m_right)
            {
                successorParent->m_left = successor->m_right;
                successor->m_right = node->m_right;
            }
```



```cpp
            if(node == m_root)
            {
                m_root = successor;
            }
            else if(isLeftNode)
            {
                node = parent->m_left;
                parent->m_left = successor;
            }
            else
            {
                node = parent->m_right;
                parent->m_right = successor;
            }

            successor->m_left = node->m_left;
        }

        node->m_left = NULL;
        node->m_right = NULL;
        delete node;
    }
```

## Binary Trees



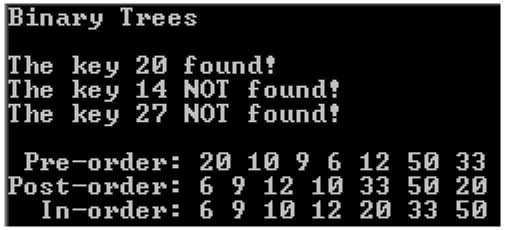
```
    void DisplayPreOrder()
    {
        DisplayPreOrder(m_root);
    }

    void DisplayPostOrder()
    {
        DisplayPostOrder(m_root);
    }

    void DisplayInOrder()
    {
        DisplayInOrder(m_root);
    }

private:
    void DisplayPreOrder(Node<T> *node)
    {
        if(node != NULL)
        {
            cout << node->m_key << " ";

            DisplayPreOrder(node->m_left);
            DisplayPreOrder(node->m_right);
        }
    }
```

```
    void DisplayPostOrder(Node<T> *node)
    {
        if(node != NULL)
        {
            DisplayPostOrder(node->m_left);
            DisplayPostOrder(node->m_right);

            cout << node->m_key << " ";
        }
    }

    void DisplayInOrder(Node<T> *node)
    {
        if(node != NULL)
        {
            DisplayInOrder(node->m_left);

            cout << node->m_key << " ";

            DisplayInOrder(node->m_right);
        }
    }

private:
    Node<T> *m_root;
};
```

## Binary Trees Example

```cpp
int main(int args, char **argc)
{
    cout << "Binary Trees" << endl;
    cout << endl;

    BinaryTree<int> binaryTree;

    binaryTree.push(20);
    binaryTree.push(10);
    binaryTree.push(12);
    binaryTree.push(27);
    binaryTree.push(9);
    binaryTree.push(50);
    binaryTree.push(33);
    binaryTree.push(6);

    binaryTree.remove(27);

    if(binaryTree.search(20) == true)
        cout << "The key 20 found!" << endl;
    else
        cout << "The key 20 NOT found!" << endl;

    if(binaryTree.search(14) == true)
        cout << "The key 14 found!" << endl;
    else
        cout << "The key 14 NOT found!" << endl;

    if(binaryTree.search(27) == true)
        cout << "The key 27 found!" << endl;
    else
        cout << "The key 27 NOT found!" << endl;

    cout << endl;
```

```cpp
    cout << " Pre-order: ";
    binaryTree.DisplayPreOrder();
    cout << endl;

    cout << "Post-order: ";
    binaryTree.DisplayPostOrder();
    cout << endl;

    cout << "  In-order: ";
    binaryTree.DisplayInOrder();
    cout << endl << endl;

    return 1;
}
```



```
Binary Trees

The key 20 found!
The key 14 NOT found!
The key 27 NOT found!

 Pre-order: 20 10 9 6 12 50 33
Post-order: 6 9 12 10 33 50 20
  In-order: 6 9 10 12 20 33 50
```

## kd Trees

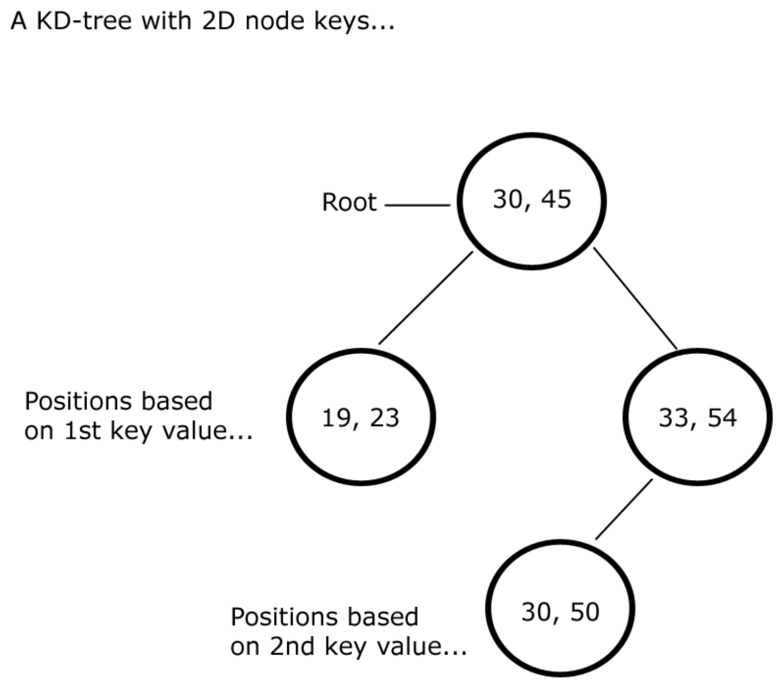
- k-Dimensional Trees
- Type of binary tree that uses keys to have multiple dimensions
- The multidimensional can vary
  - Hence the k in kd-trees

- Used for:
  - Range searches
  - Nearest neighbor search
  - Space partitioning (BSP)

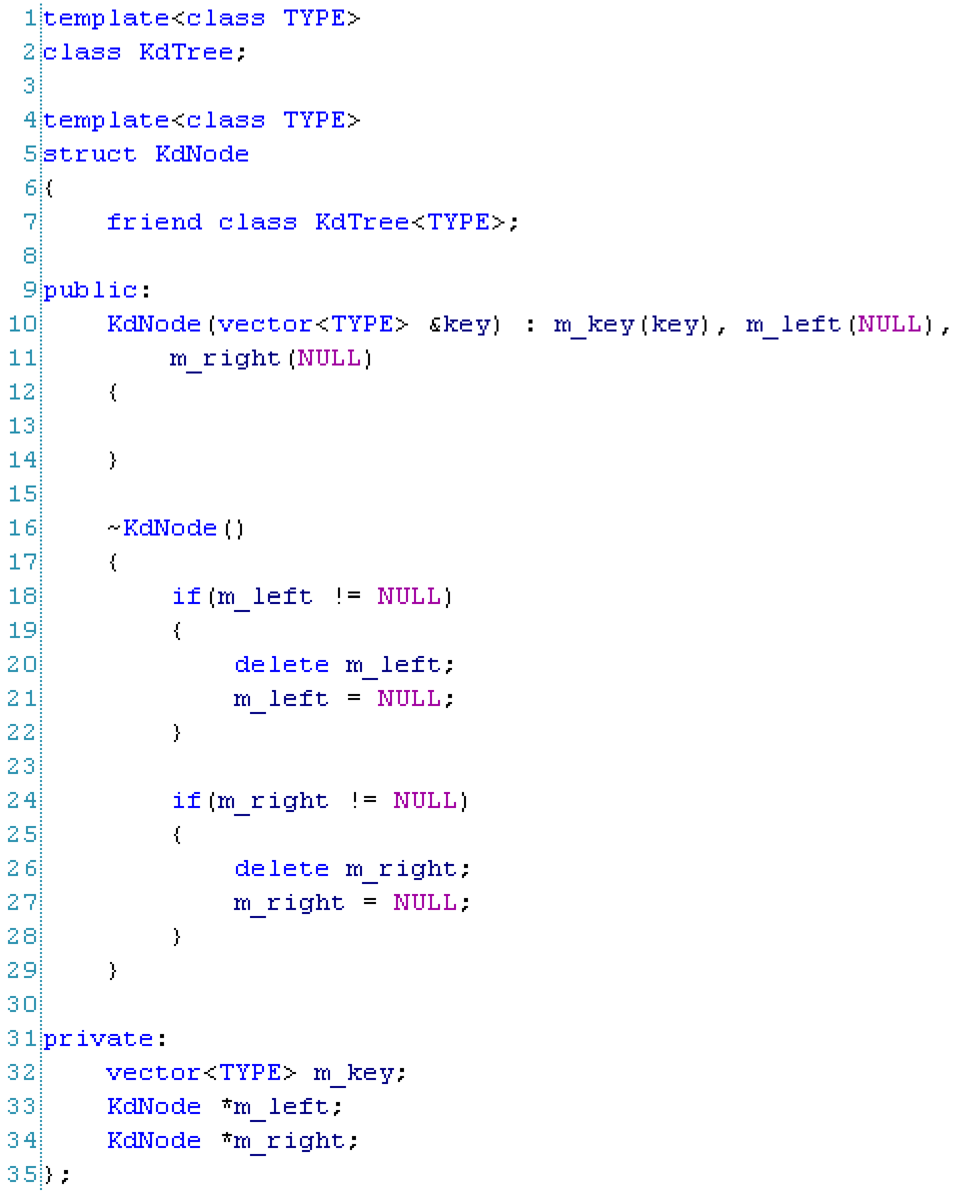
## kd Trees

- By sending in a range of key values, a kd-tree can find all nodes that fall within that range or that match it
- Insertion
  - the current depth of the tree is used to determine which dimension of the key is used when determining the direction of traversal

## kd Trees

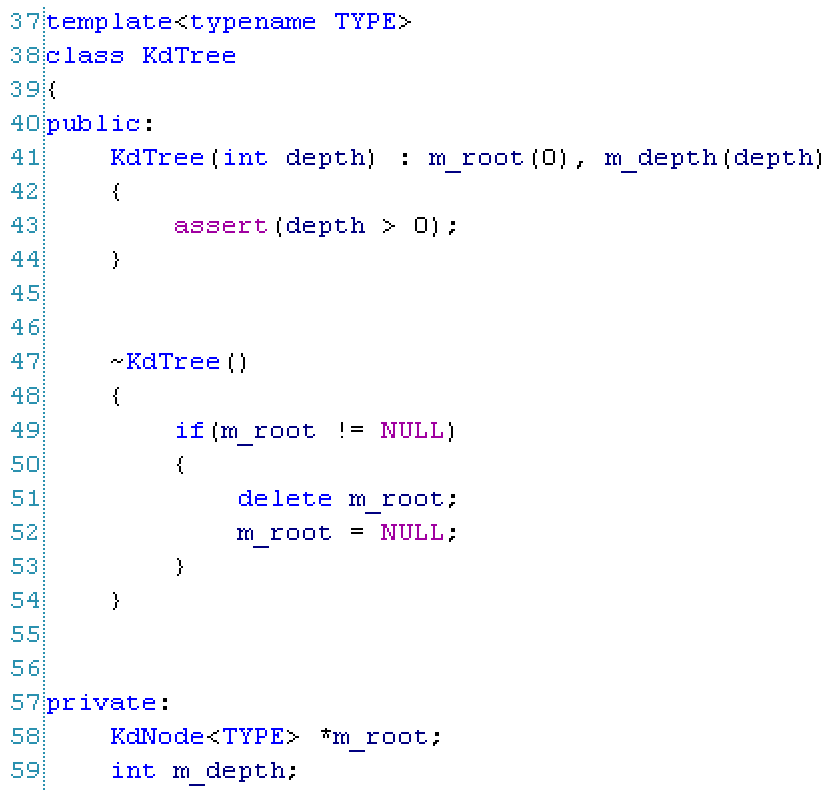

A KD-tree with 2D node keys...
Root
30, 45
19, 23
33, 54
30, 50
Positions based on 1st key value...
Positions based on 2nd key value...

## kd Trees



```cpp
template<class TYPE>
class KdTree;

template<class TYPE>
struct KdNode
{
    friend class KdTree<TYPE>;

public:
    KdNode(vector<TYPE> &key) : m_key(key), m_left(NULL),
        m_right(NULL)
    {

    }

    ~KdNode()
    {
        if(m_left != NULL)
        {
            delete m_left;
            m_left = NULL;
        }

        if(m_right != NULL)
        {
            delete m_right;
            m_right = NULL;
        }
    }

private:
    vector<TYPE> m_key;
    KdNode *m_left;
    KdNode *m_right;
};
```

## kd Trees



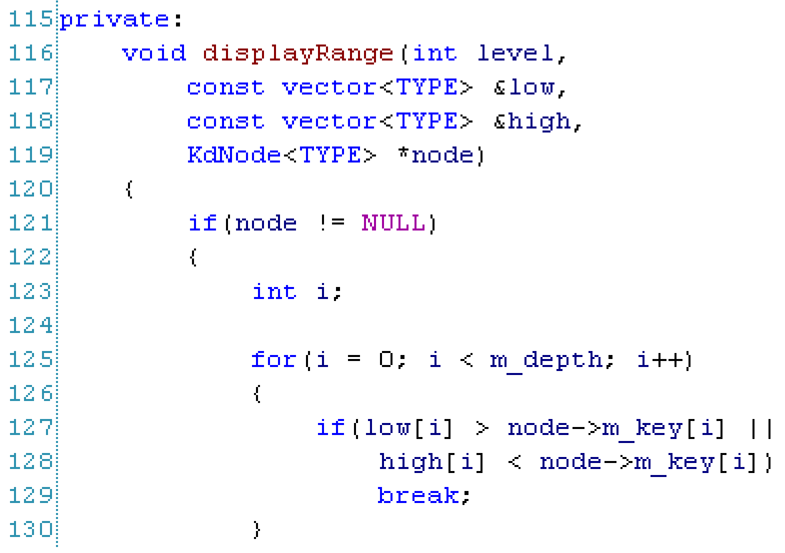
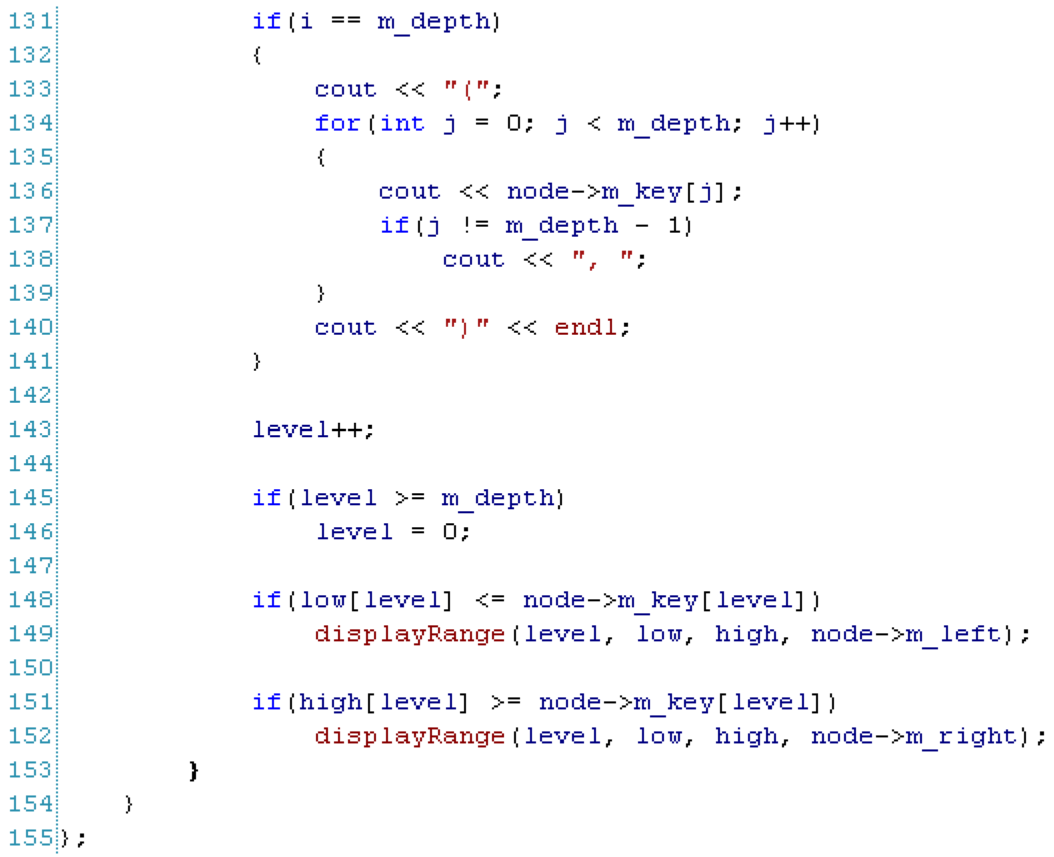
```
template<typename TYPE>
class KdTree
{
public:
    KdTree(int depth) : m_root(0), m_depth(depth)
    {
        assert(depth > 0);
    }


    ~KdTree()
    {
        if(m_root != NULL)
        {
            delete m_root;
            m_root = NULL;
        }
    }


private:
    KdNode<TYPE> *m_root;
    int m_depth;
```

## kd Trees

```
public:
    void push(vector<TYPE> &key)
    {
        KdNode<TYPE> *newNode = new KdNode<TYPE>(key);

        if(m_root == NULL)
        {
            m_root = newNode;
            return;
        }

        KdNode<TYPE> *currentNode = m_root;
        KdNode<TYPE> *parentNode = m_root;
        int level = 0;
```

```
        while(1)
        {
            parentNode = currentNode;

            if(key[level] < currentNode->m_key[level])
            {
                currentNode = currentNode->m_left;

                if(currentNode == NULL)
                {
                    parentNode->m_left = newNode;
                    return;
                }
            }
            else
            {
                currentNode = currentNode->m_right;

                if(currentNode == NULL)
                {
                    parentNode->m_right = newNode;
                    return;
                }
            }

            level++;

            if(level >= m_depth)
                level = 0;
        }
    }
```

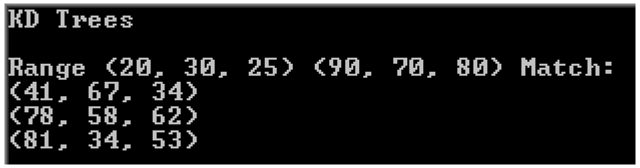
## kd Trees

```
private:
    void displayRange(int level,
        const vector<TYPE> &low,
        const vector<TYPE> &high,
        KdNode<TYPE> *node)
    {
        if(node != NULL)
        {
            int i;

            for(i = 0; i < m_depth; i++)
            {
                if(low[i] > node->m_key[i] ||
                    high[i] < node->m_key[i])
                    break;
            }
            if(i == m_depth)
            {
                cout << "(";
                for(int j = 0; j < m_depth; j++)
                {
                    cout << node->m_key[j];
                    if(j != m_depth - 1)
                        cout << ", ";
                }
                cout << ")" << endl;
            }

            level++;

            if(level >= m_depth)
                level = 0;

            if(low[level] <= node->m_key[level])
                displayRange(level, low, high, node->m_left);

            if(high[level] >= node->m_key[level])
                displayRange(level, low, high, node->m_right);
        }
    }
};
```

## kd Tree Example

```
int main(int args, char **argc)
{
   cout << "KD Trees" << endl;
   cout << endl;

   // Create KD tree and populate it.
   KdTree<int> kdTree(3);

   for(int i = 0; i < 100; i++)
   {
     vector<int> key(3);

     key[0] =  rand() % 100;
     key[1] =  rand() % 100;
     key[2] =  rand() % 100;

     kdTree.push(key);
   }

   // Display range of values that falls within the range.
   vector<int> low(3), high(3);

   low[0] = 20;
   low[1] = 30;
   low[2] = 25;

   high[0] = 90;
   high[1] = 70;
   high[2] = 80;

   cout << "Range (20, 30, 25) (90, 70, 80) Match:" << endl;
   kdTree.displayRange(low, high);
   cout << endl << endl;

   return 1;
}
```



```
KD Trees

Range (20, 30, 25) (90, 70, 80) Match:
(41, 67, 34)
(78, 58, 62)
(81, 34, 53)
```

## Additional Types Of Trees

- B-Trees
  - balanced tree data structure with multiple child nodes that can be attached to one node
  - keep the number of child nodes within a certain range
  - when a node violates this range, the tree is altered so that it follows the rules of a b-tree
    - done by joining nodes together or splitting them

## Additional Types Of Trees

- AVL Trees
  - self-balancing binary search tree
    - here are two child nodes for every node
  - height of the child nodes in an AVL tree differ at most by one (height-balanced trees)
    - balance factor - the height of the right child minus the height of the left child of any given node (between -1 and 1)
  - insertion and deletion
    - alter the structure of the tree as needed to keep it balanced

## Additional Types Of Trees

- Red-Black Trees
  - balanced binary trees
  - different insertions and deletions algorithms
    - alter the structure of the tree as needed to keep it balanced
    - not as fast

## Additional Types Of Trees

- 2-3 Trees
  - b-trees of order 3, they can have up to three child nodes for each node
  - self-balancing tree
  - a node with a data item can have two children, and a node with two data items can have three children

- 2-3-4 Trees
  - can have up to four child nodes

## Additional Types Of Trees

- Heaps
  - weakly ordered binary tree that keeps the node with the largest key (root node) on the top of the tree
  - all nodes within the heap are not necessarily in order
  - the only thing that is certain in a heap is that the child nodes of any given node have a key that is less than their parent
  - often implemented as arrays