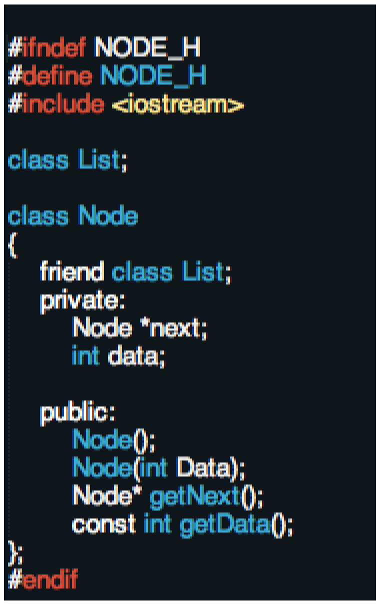
Data Structures

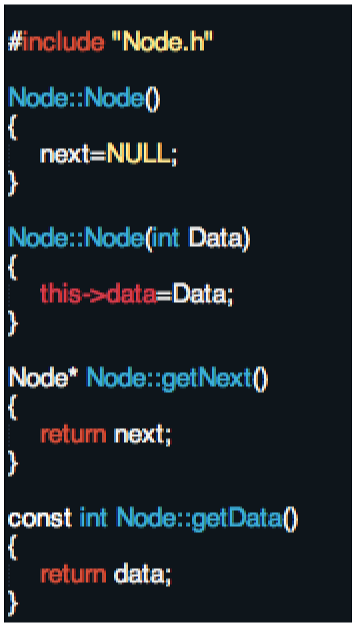
LINKED LIST IMPLEMENTATION

Node.h



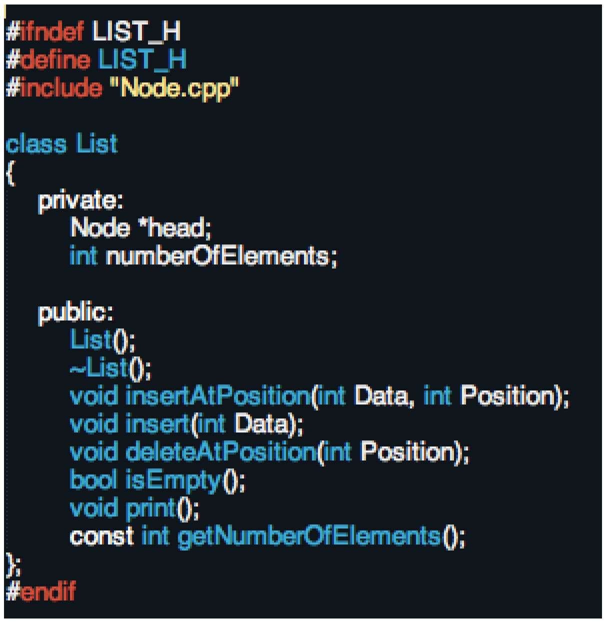
```cpp
#ifndef NODE_H
#define NODE_H
#include <iostream>

class List;

class Node
{
    friend class List;
    private:
        Node *next;
        int data;

    public:
        Node();
        Node(int Data);
        Node* getNext();
        const int getData();
};
#endif
```

Node.cpp



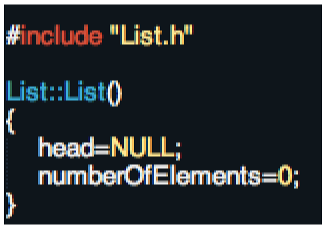
```cpp
#include "Node.h"

Node::Node()
{
    next=NULL;
}

Node::Node(int Data)
{
    this->data=Data;
}

Node* Node::getNext()
{
    return next;
}

const int Node::getData()
{
    return data;
}
```

List.h

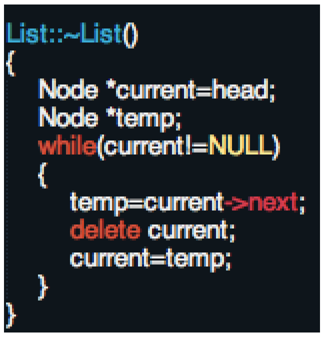


```cpp
#ifndef LIST_H
#define LIST_H
#include "Node.cpp"

class List
{
    private:
        Node *head;
        int numberOfElements;

    public:
        List();
        ~List();
        void insertAtPosition(int Data, int Position);
        void insert(int Data);
        void deleteAtPosition(int Position);
        bool isEmpty();
        void print();
        const int getNumberOfElements();
};
#endif
```

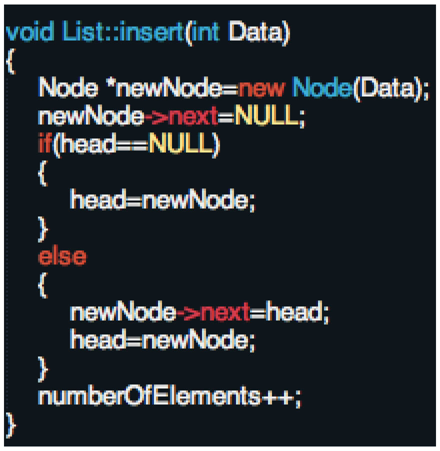
List.cpp



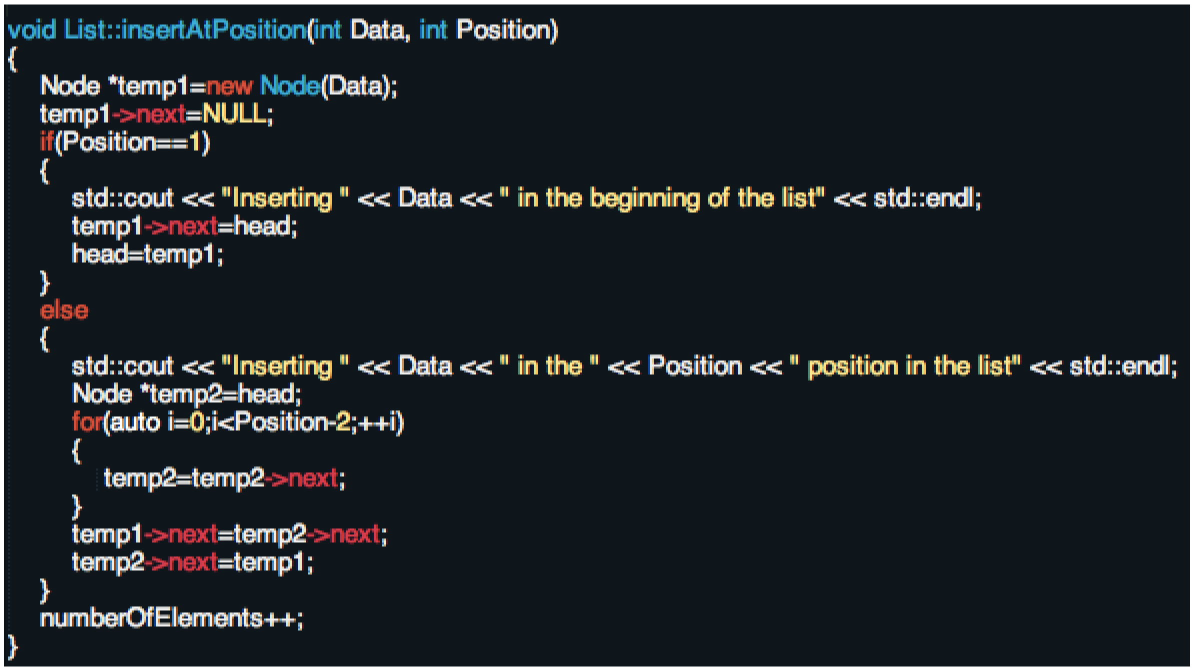
```cpp
#include "List.h"

List::List()
{
    head=NULL;
    numberOfElements=0;
}
```



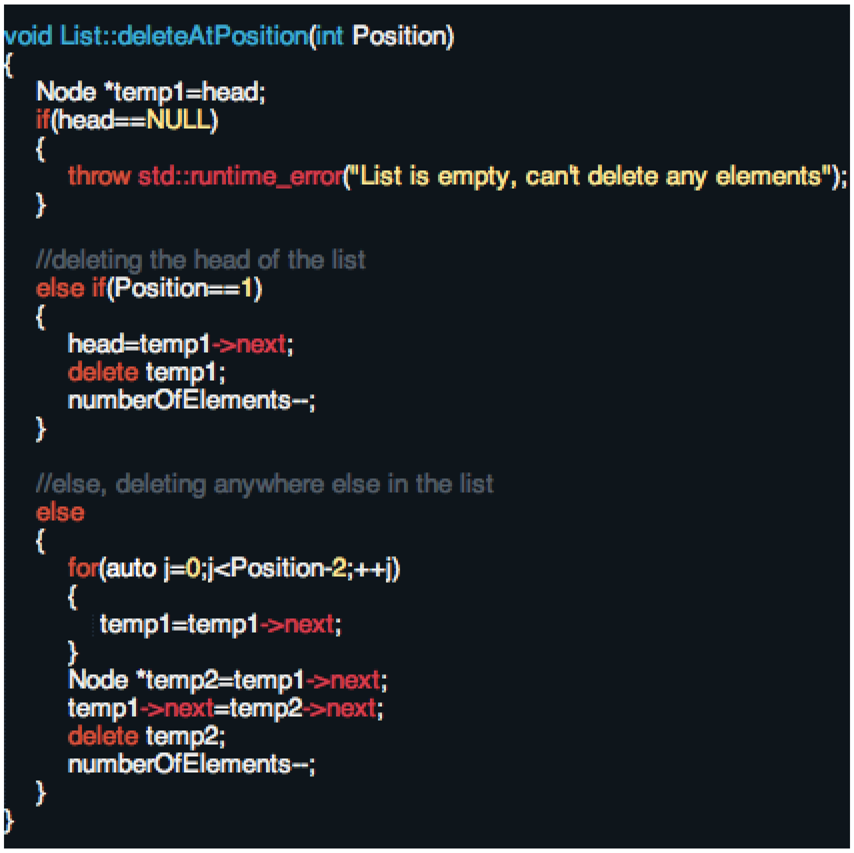
```cpp
List::~List()
{
    Node *current=head;
    Node *temp;
    while(current!=NULL)
    {
        temp=current->next;
        delete current;
        current=temp;
    }
}
```



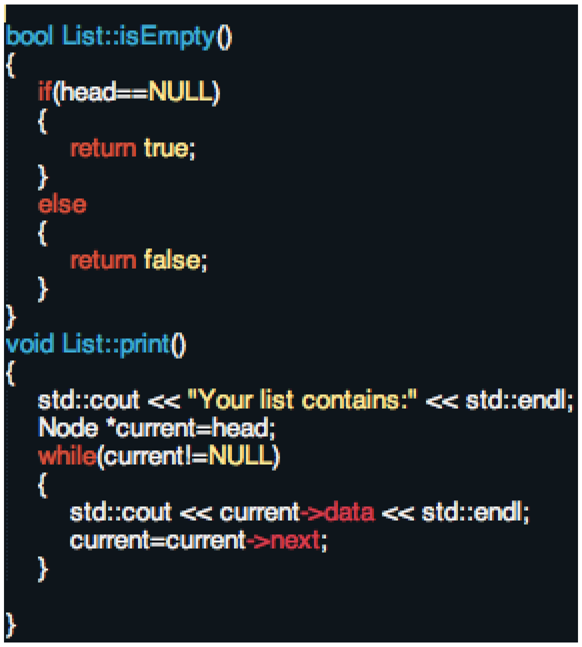
```cpp
void List::insert(int Data)
{
    Node *newNode=new Node(Data);
    newNode->next=NULL;
    if(head==NULL)
    {
        head=newNode;
    }
    else
    {
        newNode->next=head;
        head=newNode;
    }
    numberOfElements++;
}
```



```cpp
void List::insertAtPosition(int Data, int Position)
{
    Node *temp1=new Node(Data);
    temp1->next=NULL;
    if(Position==1)
    {
        std::cout << "Inserting " << Data << " in the beginning of the list" << std::endl;
        temp1->next=head;
        head=temp1;
    }
    else
    {
        std::cout << "Inserting " << Data << " in the " << Position << " position in the list" << std::endl;
        Node *temp2=head;
        for(auto i=0;i<Position-2;++i)
        {
            temp2=temp2->next;
        }
        temp1->next=temp2->next;
        temp2->next=temp1;
    }
    numberOfElements++;
}
```



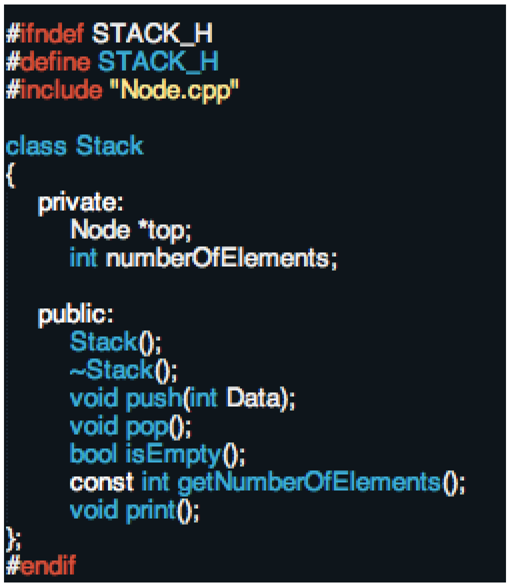
```cpp
void List::deleteAtPosition(int Position)
{
    Node *temp1=head;
    if(head==NULL)
    {
        throw std::runtime_error("List is empty, can't delete any elements");
    }

    //deleting the head of the list
    else if(Position==1)
    {
        head=temp1->next;
        delete temp1;
        numberOfElements--;
    }

    //else, deleting anywhere else in the list
    else
    {
        for(auto j=0;j<Position-2;++j)
        {
            temp1=temp1->next;
        }
        Node *temp2=temp1->next;
        temp1->next=temp2->next;
        delete temp2;
        numberOfElements--;
    }
}
```



```cpp
bool List::isEmpty()
{
    if(head==NULL)
    {
        return true;
    }
    else
    {
        return false;
    }
}
void List::print()
{
    std::cout << "Your list contains:" << std::endl;
    Node *current=head;
    while(current!=NULL)
    {
        std::cout << current->data << std::endl;
        current=current->next;
    }

}
```

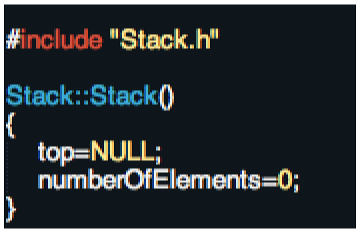
STACK IMPLEMENTATION (USING THE SAME NODE CLASS AS LINKED LIST, QUEUE, AND STACK)

Stack.h

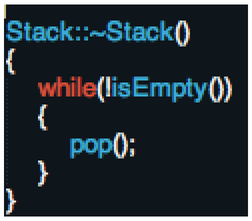


```
#ifndef STACK_H
#define STACK_H
#include "Node.cpp"

class Stack
{
    private:
        Node *top;
        int numberOfElements;

    public:
        Stack();
        ~Stack();
        void push(int Data);
        void pop();
        bool isEmpty();
        const int getNumberOfElements();
        void print();
};
#endif
```

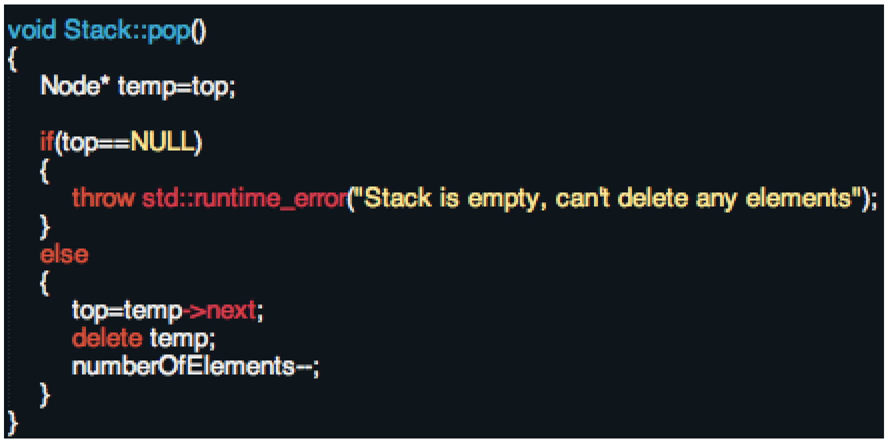
Stack.cpp



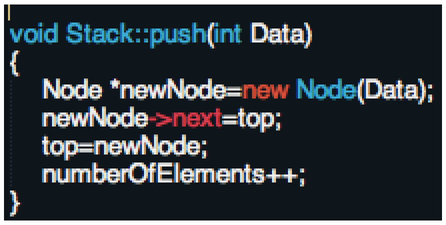
```
#include "Stack.h"

Stack::Stack()
{
    top=NULL;
    numberOfElements=0;
}
```



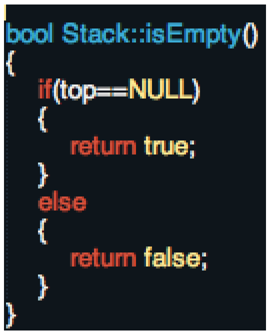
```
Stack::~Stack()
{
    while(!isEmpty())
    {
        pop();
    }
}
```



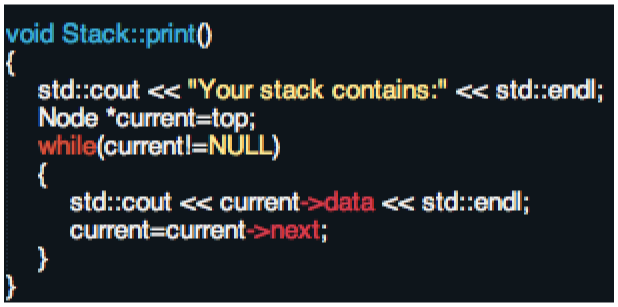
```
void Stack::pop()
{
    Node* temp=top;

    if(top==NULL)
    {
        throw std::runtime_error("Stack is empty, can't delete any elements");
    }
    else
    {
        top=temp->next;
        delete temp;
        numberOfElements--;
    }
}
```



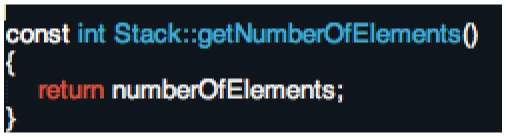
```
void Stack::push(int Data)
{
    Node *newNode=new Node(Data);
    newNode->next=top;
    top=newNode;
    numberOfElements++;
}
```



```
bool Stack::isEmpty()
{
    if(top==NULL)
    {
        return true;
    }
    else
    {
        return false;
    }
}
```



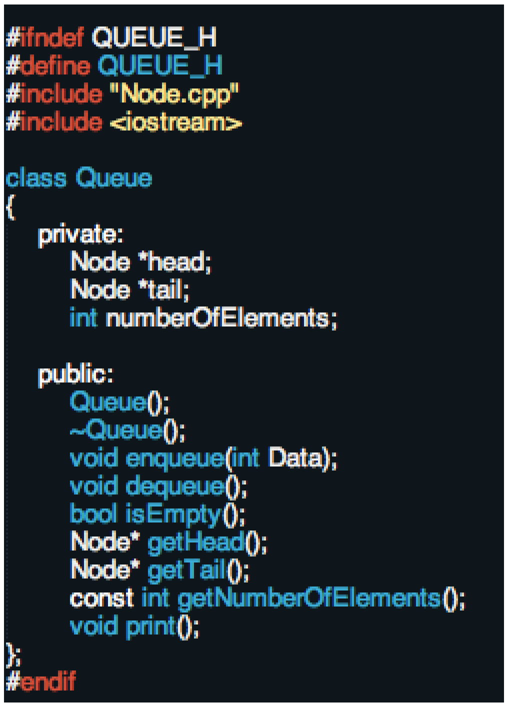
```
void Stack::print()
{
    std::cout << "Your stack contains:" << std::endl;
    Node *current=top;
    while(current!=NULL)
    {
        std::cout << current->data << std::endl;
        current=current->next;
    }
}
```



```
const int Stack::getNumberOfElements()
{
    return numberOfElements;
}
```

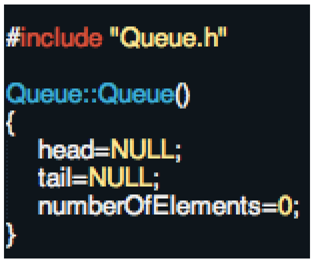
Queue Implementation(using same Node Class as Stack)

Queue.h

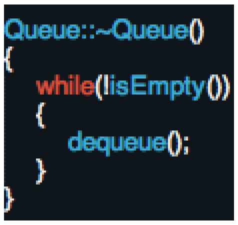


```
#ifndef QUEUE_H
#define QUEUE_H
#include "Node.cpp"
#include <iostream>

class Queue
{
    private:
        Node *head;
        Node *tail;
        int numberOfElements;

    public:
        Queue();
        ~Queue();
        void enqueue(int Data);
        void dequeue();
        bool isEmpty();
        Node* getHead();
        Node* getTail();
        const int getNumberOfElements();
        void print();
};
#endif
```

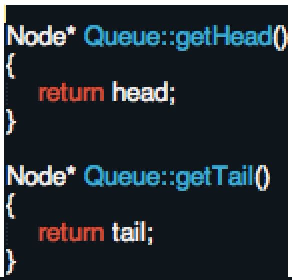
Queue.cpp



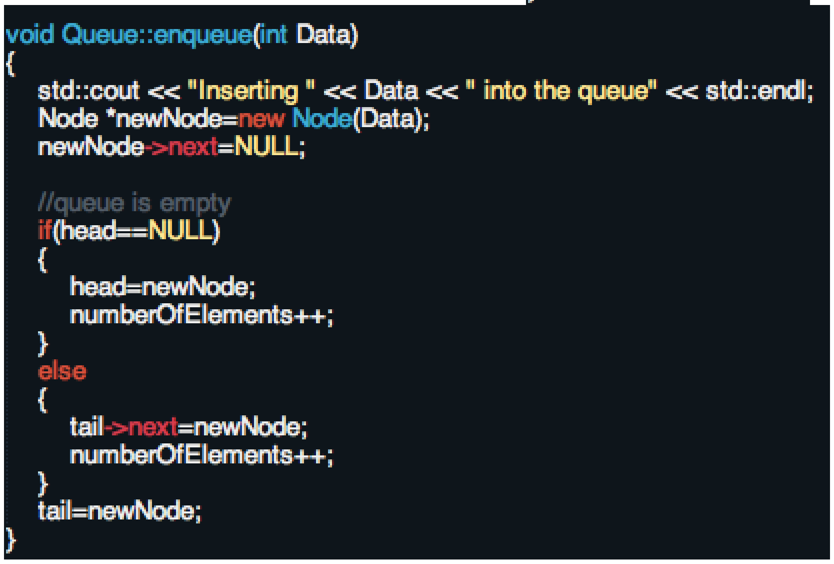
```
#include "Queue.h"

Queue::Queue()
{
    head=NULL;
    tail=NULL;
    numberOfElements=0;
}
```



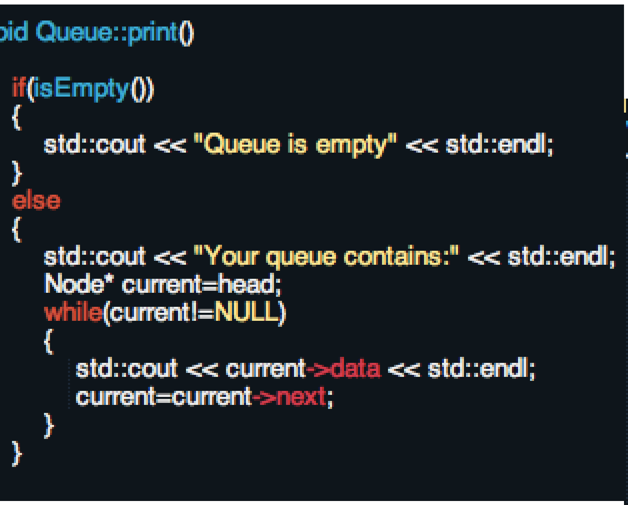
```
Queue::~Queue()
{
    while(!isEmpty())
    {
        dequeue();
    }
}
```



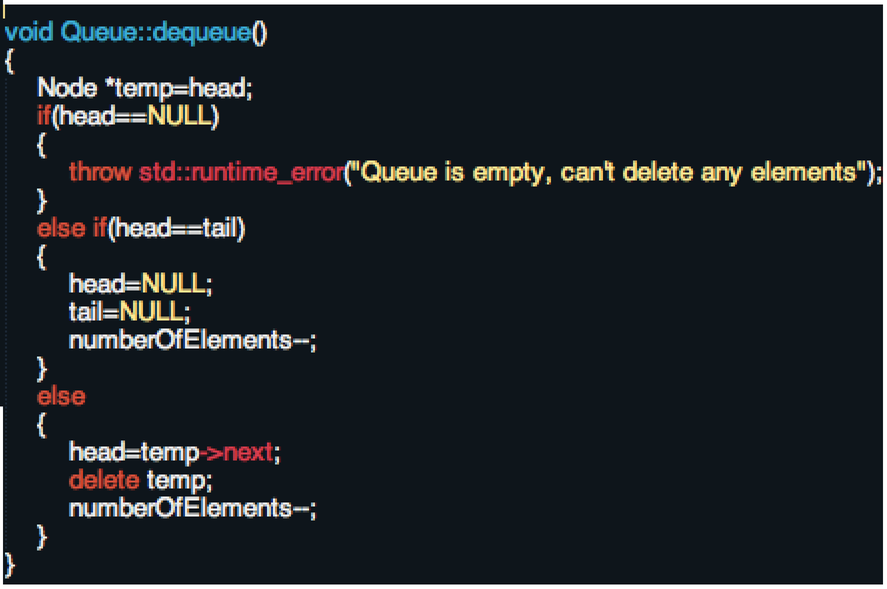
```
Node* Queue::getHead()
{
    return head;
}

Node* Queue::getTail()
{
    return tail;
}
```



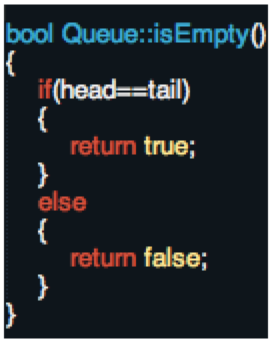
```
void Queue::enqueue(int Data)
{
    std::cout << "Inserting " << Data << " into the queue" << std::endl;
    Node *newNode=new Node(Data);
    newNode->next=NULL;

    //queue is empty
    if(head==NULL)
    {
        head=newNode;
        numberOfElements++;
    }
    else
    {
        tail->next=newNode;
        numberOfElements++;
    }
    tail=newNode;
}
```



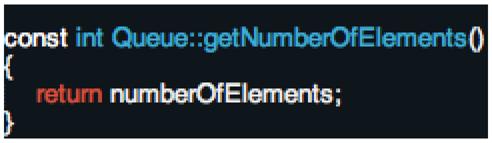
```
oid Queue::print()

 if(isEmpty())
 {
     std::cout << "Queue is empty" << std::endl;
 }
 else
 {
     std::cout << "Your queue contains:" << std::endl;
     Node* current=head;
     while(current!=NULL)
     {
         std::cout << current->data << std::endl;
         current=current->next;
     }
 }
```



```
void Queue::dequeue()
{
    Node *temp=head;
    if(head==NULL)
    {
        throw std::runtime_error("Queue is empty, can't delete any elements");
    }
    else if(head==tail)
    {
        head=NULL;
        tail=NULL;
        numberOfElements--;
    }
    else
    {
        head=temp->next;
        delete temp;
        numberOfElements--;
    }
}
```



```
bool Queue::isEmpty()
{
    if(head==tail)
    {
        return true;
    }
    else
    {
        return false;
    }
}
```

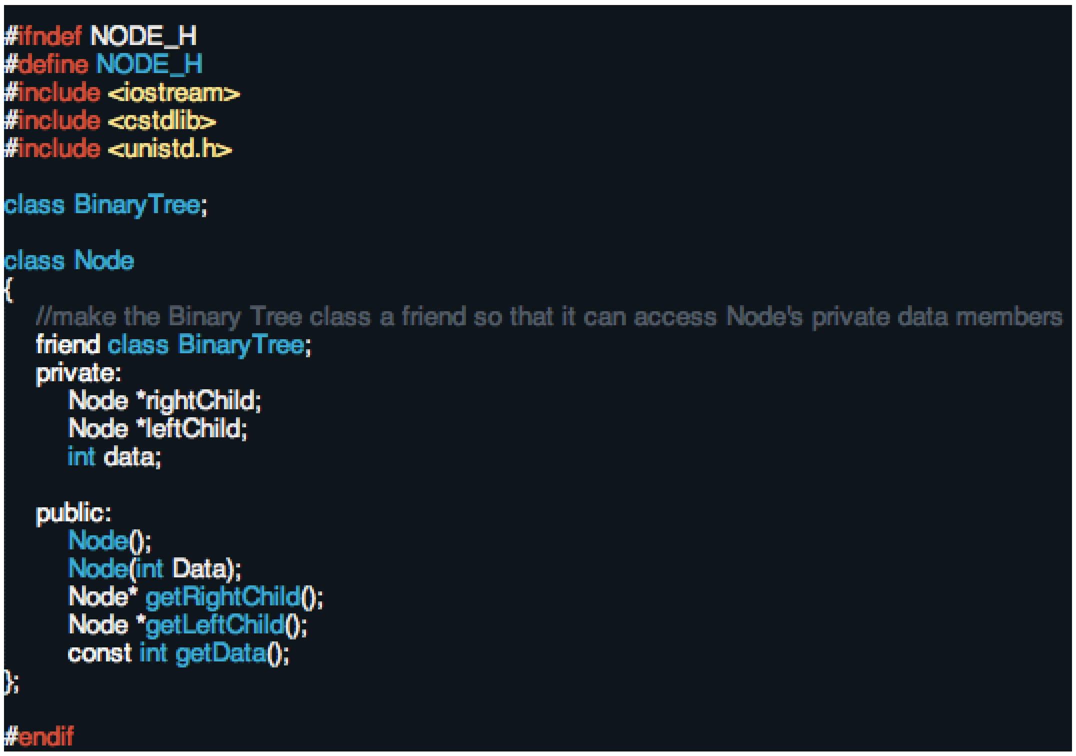


```
const int Queue::getNumberOfElements()
{
    return numberOfElements;
}
```

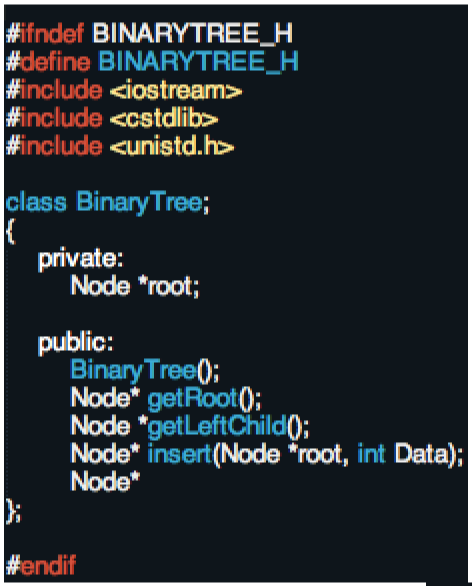
TREES

1) **ALWAYS BE THINKING RECURSIVELY**
2) General facts
    a) With N nodes you have N-1 edges
    b) **Depth of node x:** length of the path from the root node to node x, where each edge from node to node contributes one unit of length to the path
    c) **Height of Node x:** number of edges in longest path from node x to a leaf
    d) **Height of tree is** equal to the **height of** the **root node**
3) General applications
    a) Storing hierarchical data (such as in file system on a computer)
    b) Organization of data for quick search, insertion, and deletion
    c) Trie tree used as dictionary storage container
4) Binary Trees
    a) Strict binary trees: each node can have either 0 or 2 children
    b) Only restriction: each node can have at most 2 children
    c) Tree **is complete** if all levels except the bottom are filled and nodes are as far left as possible
    d) **Is balanced** if for each node, the difference in height between the left and right subtree is no more than 1
    e) Maximum number of nodes we can have at level X is 2^X
    f) Maximum number of nodes we can have in a tree with height h is 2^(h+1)-1, where (h+1) is the same as the number of levels within the tree
    g) Height of a perfect binary tree with n nodes is log base 2 of (n+1)-1
    h) minimum height of binary tree with n nodes is the floor of log base 2 of n
5) Binary Search Trees
    a) a tree in which for each node, the value of the nodes in the left subtree is less than the value of all the nodes in the right subtree
    b) **Insert**, **deletion**, and **search** take **O(logN) time** for the **average case**
    c) **O(n) in** the **worst case** because the tree is arranged as a linked list

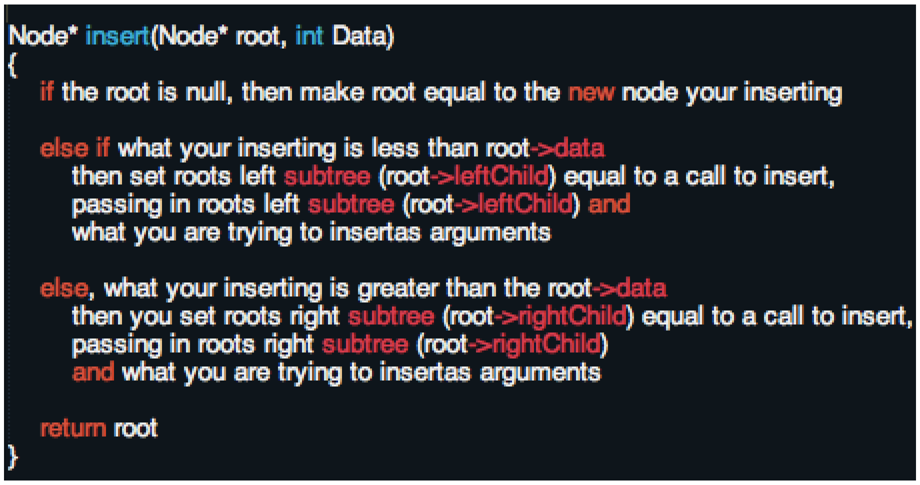
Implementation (in words/code)



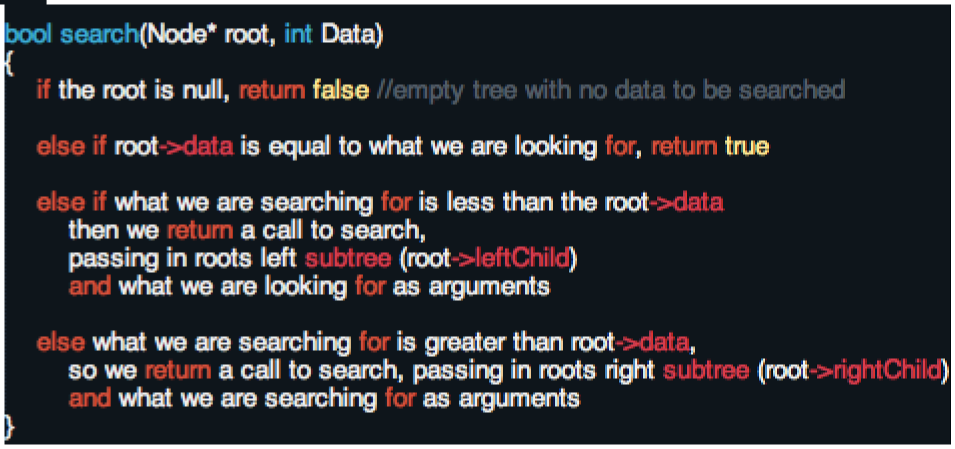
```
#ifndef NODE_H
#define NODE_H
#include <iostream>
#include <cstdlib>
#include <unistd.h>

class BinaryTree;

class Node
{
    //make the Binary Tree class a friend so that it can access Node's private data members
    friend class BinaryTree;
    private:
        Node *rightChild;
        Node *leftChild;
        int data;

    public:
        Node();
        Node(int Data);
        Node* getRightChild();
        Node *getLeftChild();
        const int getData();
};

#endif
```



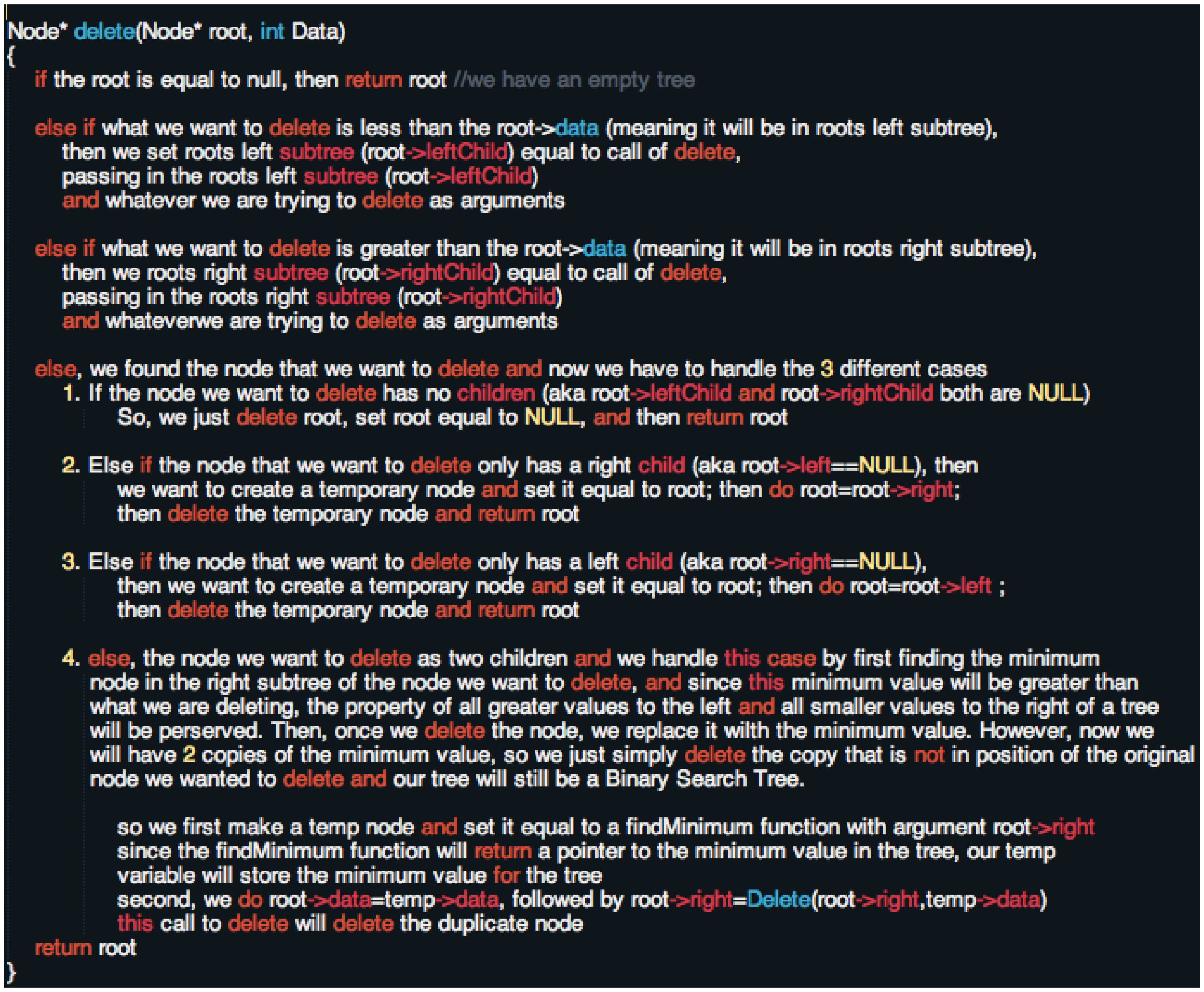
```
#ifndef BINARYTREE_H
#define BINARYTREE_H
#include <iostream>
#include <cstdlib>
#include <unistd.h>

class BinaryTree;
{
    private:
        Node *root;

    public:
        BinaryTree();
        Node* getRoot();
        Node *getLeftChild();
        Node* insert(Node *root, int Data);
        Node*
};

#endif
```



```
Node* insert(Node* root, int Data)
{
    if the root is null, then make root equal to the new node your inserting

    else if what your inserting is less than root->data
        then set roots left subtree (root->leftChild) equal to a call to insert,
        passing in roots left subtree (root->leftChild) and
        what you are trying to insertas arguments

    else, what your inserting is greater than the root->data
        then you set roots right subtree (root->rightChild) equal to a call to insert,
        passing in roots right subtree (root->rightChild)
        and what you are trying to insertas arguments

    return root
}
```



```
bool search(Node* root, int Data)
{
    if the root is null, return false //empty tree with no data to be searched

    else if root->data is equal to what we are looking for, return true

    else if what we are searching for is less than the root->data
        then we return a call to search,
        passing in roots left subtree (root->leftChild)
        and what we are looking for as arguments

    else what we are searching for is greater than root->data,
        so we return a call to search, passing in roots right subtree (root->rightChild)
        and what we are searching for as arguments
}
```



```
Node* delete(Node* root, int Data)
{
    if the root is equal to null, then return root //we have an empty tree

    else if what we want to delete is less than the root->data (meaning it will be in roots left subtree),
        then we set roots left subtree (root->leftChild) equal to call of delete,
        passing in the roots left subtree (root->leftChild)
        and whatever we are trying to delete as arguments

    else if what we want to delete is greater than the root->data (meaning it will be in roots right subtree),
        then we roots right subtree (root->rightChild) equal to call of delete,
        passing in the roots right subtree (root->rightChild)
        and whateverwe are trying to delete as arguments

    else, we found the node that we want to delete and now we have to handle the 3 different cases
        1. If the node we want to delete has no children (aka root->leftChild and root->rightChild both are NULL)
            So, we just delete root, set root equal to NULL, and then return root

        2. Else if the node that we want to delete only has a right child (aka root->left==NULL), then
            we want to create a temporary node and set it equal to root; then do root=root->right;
            then delete the temporary node and return root

        3. Else if the node that we want to delete only has a left child (aka root->right==NULL),
            then we want to create a temporary node and set it equal to root; then do root=root->left ;
            then delete the temporary node and return root

        4. else, the node we want to delete as two children and we handle this case by first finding the minimum
           node in the right subtree of the node we want to delete, and since this minimum value will be greater than
           what we are deleting, the property of all greater values to the left and all smaller values to the right of a tree
           will be perserved. Then, once we delete the node, we replace it with the minimum value. However, now we
           will have 2 copies of the minimum value, so we just simply delete the copy that is not in position of the original
           node we wanted to delete and our tree will still be a Binary Search Tree.

            so we first make a temp node and set it equal to a findMinimum function with argument root->right
            since the findMinimum function will return a pointer to the minimum value in the tree, our temp
            variable will store the minimum value for the tree
            second, we do root->data=temp->data, followed by root->right=Delete(root->right,temp->data)
            this call to delete will delete the duplicate node
    return root
}
```

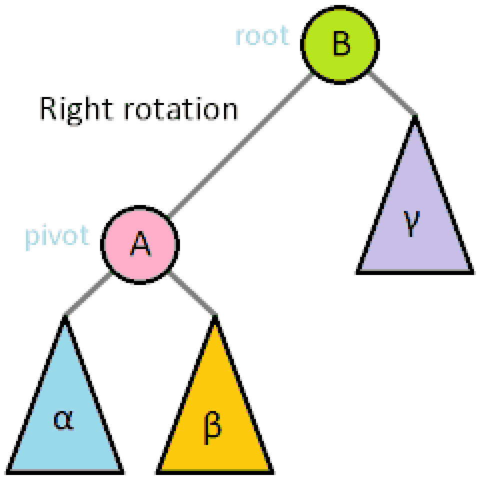
6) AVL Trees
    a) Subset of Binary Search Tree with the following properties:
        i) the sub-trees of every node differ in height by at most 1
        ii) every sub-tree is an AVL tree
    b) Balance factor = height(right-subtree) - height(left-subtree)
    c) Like binary search trees, **insertion**, **deletion**, and **search** are **O(logN)** in **average case**
    d) **Worst case: O(n)**

Rotation

If we add an element to a left subtree and then the difference between that left subtree and the right subtree becomes more than one, we need to perform a right rotation

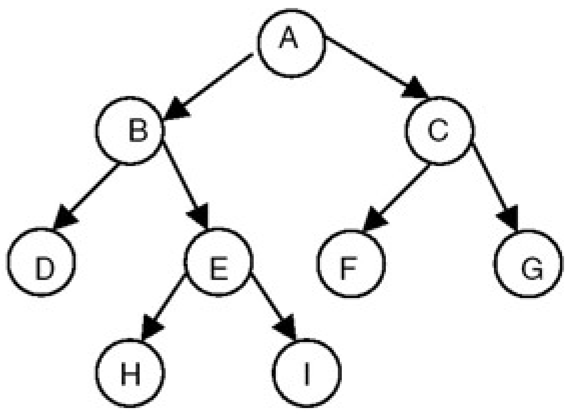
If we add an element to a right subtree and then the difference between the right subtree and left subtree becomes more than one, we need a left rotation

However, there are times when a single rotation doesn't restore balance in the tree and we need to do a double rotation. Basically, just check the height after your first rotation, and if the height of the left and right subtrees are greater than one, do the appropriate left or right rotation again



1) Red and Black Trees
    a) Every node is either red or black (hence the name)
    b) Root of the tree is always black
    c) There are no two adjacent red nodes (no red node has a red parent or red child)
    d) Every path from the root to a NULL node has the same number of black nodes
    e) WHY DO I CARE: Since the height of a Red and Black tree is always O(LogN), where n is the number of nodes in the tree, then we can guarantee an upper bound of **O(LogN)** for **insertion**, **deletion**, and **search**
2) Printing out a tree
   Pre-order: Root, Left subtree, Right subtree
   In order: Left subtree, Root, Right subtree
   Post Order: Left subtree, Right subtree, Root
   Level order (just a depth first search by level): so, starting at the root move down level by level and on each level, print the nodes from left to right

So, for the tree below, level order is: A,B,C,D,E,F,G,H,I



Inorder : DBHEIAFCG

Preorder : ABDEHICFG

Postorder : DHIEBFGCA

GRAPHS

1) Graphs have n(n-1) edges at most if it is a directed graph
2) Graphs have n(n-1)/2 edges at most if it is an undirected graph
3) All depth first searches require a priority queue
    a) to use the STL priority queue, need #include <queue> and declare priority queue as priority_queue<type> myQueue;
4) Directed- edges go one way
5) Undirected- edges go either way

HEAPS

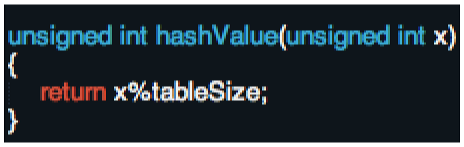
1) A complete binary tree whose nodes contain comparable objects sorted in the following two ways:
    a) Maxheap- parent nodes are always greater than or equal to child nodes and the max value is the root node
    b) Minheap- parent nodes are always less than or equal to child nodes and the min value is the root node
2) Complexity: **search**, **insert**, and **delete** are **O(nLogn)**
3) We can make priority queues out of heaps

HASHING

<u>Methods for integers</u>

Division:

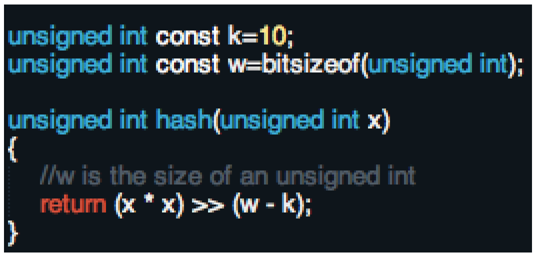
- hash key= key % table size, where the table size is a prime
- Advantage(s):
    - if table size is a prime, then the likelihood the hash keys will be spread out increases
    - also, if the table size is prime, then the division depends on all the bits of the key, not just the bottom k bits, for some integer k
- Disadvantage
    - consecutive keys maps to consecutive hash values



```
unsigned int hashValue(unsigned int x)
{
   return x%tableSize;
}
```

Middle Square Method:

- square the key, and then take out the middle two numbers
- Advantage: works well because most or all bits of the key contribute to the result
- Disadvantage: since the middle-square method only considers a subset of the bits in the middle of x^2, keys which have a large number of leading zeroes will collide
- Variation: use the middle three digits of the key to be hashed so that the method can be used when the key is too large to square



```
unsigned int const k=10;
unsigned int const w=bitsizeof(unsigned int);

unsigned int hash(unsigned int x)
{
    //w is the size of an unsigned int
    return (x * x) >> (w - k);
}
```

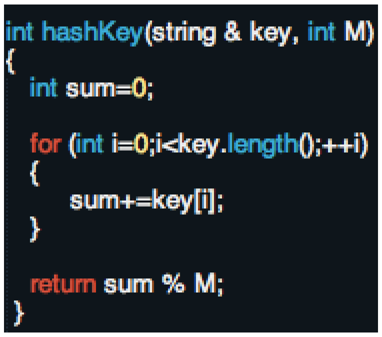
Folding method:

- the key being hashed is divided into several parts and those parts are folded or combined in a certain manner and then result is then modded by the table size

<u>Methods for Strings</u>

Summation function:

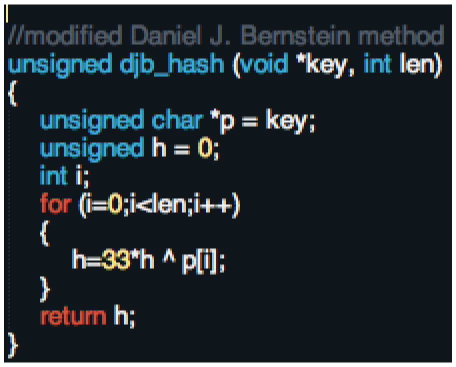
- sums the ASCII values for the string and then mods it by the table size
- advantages:
    - if the hash table size M is small compared to the resulting summations, then this hash function should do a good job of distributing strings evenly among the hash table slots, because it gives equal weight to all characters in the string
    - order of characters has no effect on the result



```
int hashKey(string & key, int M)
{
   int sum=0;

   for (int i=0;i<key.length();++i)
   {
        sum+=key[i];
   }

   return sum % M;
}
```

Modified Daniel J. Bernstein (djb2) method:

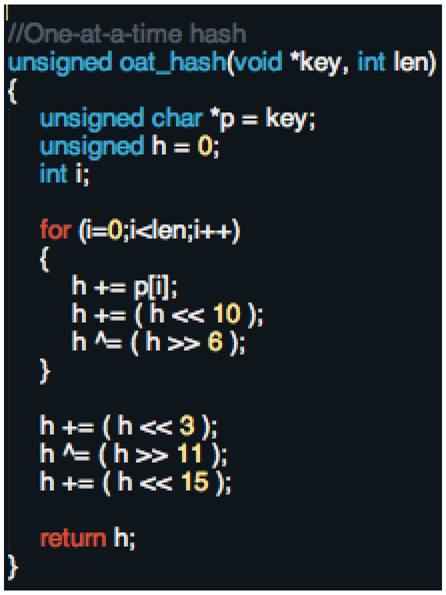
- starts with a seed, multiplies it by 33, and then adds the the current character to create the new hash value; does this for every character in the string
- first reported by Dan Bernstein



```c
//modified Daniel J. Bernstein method
unsigned djb_hash (void *key, int len)
{
    unsigned char *p = key;
    unsigned h = 0;
    int i;
    for (i=0;i<len;i++)
    {
        h=33*h ^ p[i];
    }
    return h;
}
```

One-at-a-time hash function:

- Part of the collection of Jenkin’s hash functions



```c
//One-at-a-time hash
unsigned oat_hash(void *key, int len)
{
    unsigned char *p = key;
    unsigned h = 0;
    int i;

    for (i=0;i<len;i++)
    {
        h += p[i];
        h += ( h << 10 );
        h ^= ( h >> 6 );
    }

    h += ( h << 3 );
    h ^= ( h >> 11 );
    h += ( h << 15 );

    return h;
}
```

Clustering

- Primary Clustering: when some collision resolution schemes create long clusters of filled slots near positions determined by the hash function
    - most **commonly found** in **linear probing**
- Secondary Clustering: The tendency for some collision resolution schemes to create a long run of filled slots away from a position determined by the hash function

Resolving Collisions

Separate Chaining:

- each position in the hash table is a linked list. When we try to insert something into a spot that's already taken, we just make a new node and added it to the end of the desired linked list
- only useful for collisions that take place **within memory**

Linear probing:

- simply tries the next bucket, and then the one after that, and if we have too, wrapping around the table, until we finds an empty bucket
- suffers from **primary clustering**

Quadratic Probing

- Just a variation on linear probing: On the first collision it looks 1 position ahead. On the second, it looks 4 positions ahead (two squared), and on the third collision it looks 9 positions (three squared), on the fourth collision, it looks 16 positions ahead (four squared), wrapping around as necessary (modulo by the size of our hash table)
- susceptible to **secondary clustering** since keys that have the same hash value also have the same probe sequence

Double Hashing:

- When a collision occurs, switch to a second hash function entirely (still with the same table size)

Tips for Avoiding Collisions

1) Make your table double the size of what you think you will need
2) Make your table size a prime number. Otherwise, any element that is a multiple of the table size will be hashed to the index equal to the table size on the first insertion and cause a collision

SORTING ALGORITHMS

Quick Sort

Parameters: an array to be sorted, a right value, and a left value

Steps:

1) Selecting the pivot: the pivot can be any element in the array, there are no restrictions on it
2) Partitioning:

   a) starting with a pointer to the beginning, called i and a pointer to the end, called j
   b) travel forward in the array with i until you find a number that is greater than or equal to the pivot; travel back in the array with j until you find a number that is less than or equal to the pivot
   c) after you have found the two numbers from parts b and c, swap them IF AND ONLY IF i<=j
      i) this means that the two pointers haven't crossed yet, so there are still numbers to be analyzed
   d) stop the partitioning once i becomes greater than j
   e) After this is done, you have divided your list into two sub lists: the left sublist is made up of numbers less than or equal to the pivot and the right sublist is made up of numbers greater than or equal to the pivot
3) Recursive calls:
   a) if our left value we passed into the function is less than j, then do quicksort(array, left, j)
   b) if our right value we passed into the function is greater than i, then do quicksort(array, right, i)

Complexity:

- **Best case** and **Average case**: **O(nLogn)**
- **Worst case: O(n^2)**

<u>Merge Sort</u>

Steps:

1) Continuously split your list into halves until you have lists of just one element each
2) Repeatedly combine the single elements from step 1 into pairs, making sure that the pairs are sorted.
3) Again, repeatedly merge two of the pairs that were created from step 2 until all of the pairs have been combined, making two lists, call them Left and Right, of size 4. Be sure that the merged lists of size 4 are also sorted.
4) Finally, merge the lists of 4 together into one list.
   a) To do this:
      i) Compare the first element in the Left list to the first element in the Right list and if the Left list element is smaller, it becomes the first element in our final sorted array. Then we start the process again, comparing the second element in the Left list with the same Right list element as before. We keep going through the Left list until we find a Right list element smaller than our Left list element
      ii) However, if the first element in the Left list is greater than the first element in the Right list, then the element from the Right list becomes the first element in our final sorted array. Then, we start the process all over again, comparing the second element in the Right list with the same Left list element as before.

Complexity:

- ❖ **Best case**, **Average case**, and **Worst case**: **O(nLogn)**

<u>Radix Sort</u>

Steps:

1) Sort by the one's digit. If two numbers have the same ones digit, then whichever one came first before sorting by one's digit will come first in the sorted list.
2) Sort by the ten's digit. If two numbers have the same ten's digit, then whichever one came first before sorting by the ten's digit will come first in the sorted list.
3) Sort by the 100's digit. If two numbers have the same 100's digit, then whichever one came first before sorting by the 100's digit will come first in the sorted list
4) Repeat this process for the 1000's digit,10,000's digit, etc.

Complexity:

- ❖ **Best Case**, **Average Case**, and **Worst Case**: **O(kn)**

SEARCHING ALGORITHMS

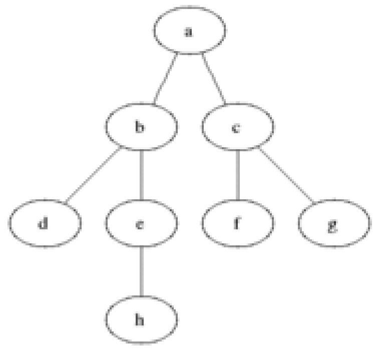
<u>Breadth First Search</u>

Description: The BFS begins at a root node and inspects all the neighboring nodes. Then for each of those neighbor nodes in turn, it inspects their neighbor nodes which were unvisited, and so on.

Algorithm:

1. Enqueue the root node
2. Dequeue a node and examine it
    - If the element sought is found in this node, quit the search and return a result.
    - Otherwise, enqueue any successors (the direct child nodes) that have not yet been discovered.
3. If the queue is empty, every node on the graph has been examined – quit the search and return "not found".
4. If the queue is not empty, repeat from Step 2.

Complexity: the entire thing is **O(V+E)**, since it travels to every single vertex via its edge



Depth First Search
Description: One starts at the root (selecting some arbitrary node as the root in the case of a graph) and explores as far as possible along each branch before backtracking

Complexity: **Θ(|V| + |E|), where |V|=the number of vertices and |E|=the number of Edges**

Breadth First Search vs Depth First Search: If you're trying to get to the bottom of the graph, DFS is more important. Plus, DFS is more efficient space wise, since there is no need to put all the pointers at the level into the queue. BFS is great at finding values closer to the top/the one you selected.

Binary Search
Prerequisite: the array passed to Binary Search must be sorted (either in ascending or descending order)

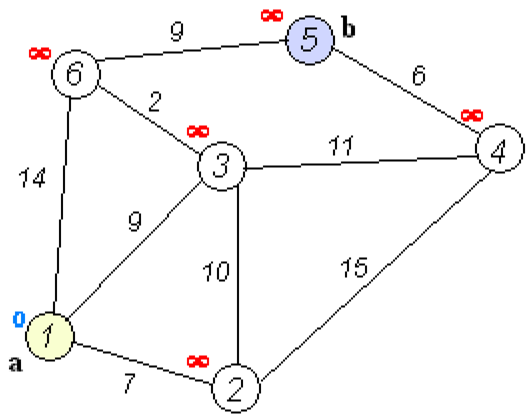
Description: we check the middle element to see if it matches the element we are searching for and if it does, return that position. if what we are searching for is smaller than the middle element, then we recursively call binary search on the left half of the original array. if what we are searching for is greater than the middle element, we recursively call binary search on the right half of the original array

If you freeze up while trying to explain this, think of the number guessing game Mr. Bui played in class back in high school: I'm thinking of a number, guess. Once you guessed he would tell you either higher or lower. If you were told higher, then you would guess from the numbers higher than your previous guess. If you were told lower, then you would guess from the numbers lower than your previous guess. Same concept, just with arrays instead.

Complexity: **Best case O(1)**, **Average** and **Worst case O(logN)**

Shortest Path by Dijkstra, using unsorted array as a priority queue
Description: For a given source vertex (node) in the graph, the algorithm finds the path with lowest cost (i.e. the shortest path) between that vertex and every other vertex

Complexity: **Average and Worst** is **O(|V|^2)**, where |V| is the number of vertices



OPERATING SYSTEMS
Virtual Memory
The basic idea is that if you don't have enough RAM your OS will use part of the hard drive to store some of that memory instead. It means if we only had 2GB of memory but actually need 3GB, we can put 1GB of memory as a manager.

Paging and Page Faults
A Page fault exists when the "page" is not in physical memory. This in itself isn't a terrible thing. In just means that we need to access the hard drive to get that information. In order to solve paging questions you basically make a queue. So when stuff gets added to memory the oldest one gets pushed out. Swapping and Paging are the exact same thing. So when you are paging something you're just seeing if it is in memory if not swapping it with what exists.

Processes vs Threads
Processes basically create threads. While they can do the same functions a process has access to multiple threads. Not only that but a multiple threads in a process can access the same data structures compared to a thread.
Processes can communicate with each other but the problem is that they're resource intensive.

1. Threads are easier to create than processes since they don't require a separate address space.

2. Multithreading requires careful programming since threads share data structures that should only be modified by one thread at a time. Unlike threads, processes don't share the same address space.
3. Threads are considered lightweight because they use far less resources than processes.
4. Processes are independent of each other. Threads, since they share the same address space are interdependent, caution must be taken so that different threads don't step on each other. This is really another way of stating #2 above.
5. A process can consist of multiple threads.

Monitors
A way to protect a thread from its data getting modified when you don't want it to be modified. Basically it allows only one thread to do a set of routines while in the monitor. Monitors handle a lot of the unlocking and locking of routines.

Semaphores
It's like a monitor, but instead of it being used for routines, it is used to lock data structures. Sephamores require the application to handle the locking and unlocking of routines

Livelock
Basic idea of items in the process constantly changing but they never progress. It's the equivalent of two people trying to pass each other in a narrow corridor but keep on stepping to the side in the same way so they never pass.

<u>Other Concepts</u>

Singleton design method

Factor design method

Memory(stack vs heap)

Interview Procedure

1. Ask Questions
2. Figure out algorithm
3. Once an idea is figured out, write a step by step comment somewhere explaining the function
4. Code the function
5. Debug your code
6. Talk about complexity (In time as well)
7. Talk about alternatives

GENERAL ADVICE

1) *ALWAYS BE ASKING QUESTIONS!!!!*
2) *ALWAYS BE TALKING WHEN CODING!!! THEY WANT TO SEE HOW YOU THINK!!!*
3) If the question is design an algorithm to sort a list, ask what sort of list, an array, linked list? What does it hold? Characters, Numbers? Ask what type of Numbers? Are these values of something? Do these values have a certain range? Like are they ages of customers? Ask how many people.
4) Don't use phrases like "um" or "like"