**BST**

Add node.

Node\* BST:: addHelper(Node\* curr, int data)

* If Current = null
  + Return
* Else if curr->key < data
  + Recursion on curr->right
* Else if curr->key > data
  + Recursion on curr->left

Void BST::addNode(int data)

* Pass root and data into recur fn.

Print Tree

void BST:: printTreeHelper(Node\* currNode){

if(currNode) {

printTreeHelper( currNode->left);

cout << " "<< currNode->key;

printTreeHelper( currNode->right);

}

}

void BST:: printTree(){

printTreeHelper(root);

cout<<endl;

}

Search

Node\* BST::searchKeyHelper(Node\* currNode, int data){

if(currNode == NULL)

return NULL;

if(currNode->key == data)

return currNode;

if(currNode->key > data)

return searchKeyHelper(currNode->left, data);

return searchKeyHelper (currNode->right, data);

}

bool BST::searchKey(int key){

Node\* tree = searchKeyHelper(root, key);

if(tree != NULL) {

return true;

}

cout<<"Key not present in the tree"<<endl;

return false;

}

Get Min/Max Value

Node\* BST::getMinValueNode(Node\* currNode){

if(currNode->left == NULL){

return currNode;

}

return getMinValueNode(currNode->left);

}

Delete subtree

void BST:: destroyNode(Node \*currNode){

if(currNode!=NULL)

{

destroyNode(currNode->left);

destroyNode(currNode->right);

delete currNode;

currNode = NULL;

}

}

Delete Node

Node\* BST::deleteNode(Node \*currNode, int value) {

if(currNode == NULL)

return NULL;

else if(value < currNode->key)

currNode->left = deleteNode(currNode->left, value);

else if(value > currNode->key)

currNode->right = deleteNode(currNode->right, value);

else {

if(currNode->left == NULL && currNode->right == NULL){

delete currNode;

return NULL;

}

else if(currNode->left == NULL){

Node \*tmp = currNode;

currNode = currNode->right;

delete tmp;

}

else if(currNode->right == NULL){

Node \*tmp = currNode;

currNode = currNode->left;

delete tmp;

}

else{

Node \*tmp = currNode;

tmp = getMinValueNode(currNode->right);

currNode->key = tmp->key;

currNode->right = deleteNode(currNode->right, tmp->key);

}

}

return currNode;

}

Sum of Nodes

int Tree::sumNodes(Node \*node){

if(node == NULL) return 0;

else

return (node->data + sumNodes(node->left) + sumNodes(node->right));

}

PreOrder Traverse

void Tree::preOrderTraverse(Node \*node) {

if (node == NULL) return;

cout << node->data << " ";

preOrderTraverse(node->left);

preOrderTraverse(node->right);

}

**GRAPHS**

ADD EDGE

void Graph::addEdge(string v1, string v2, int weight)

Loop through vertices

If v1 found

Loop through vertices

If v2 found (not same index as v1)

adjVertex ad;

ad.v = vertices[j];

ad.weight = weight;

vertices[i]->adj.push\_back(ad);

adjVertex ad2;

ad2.v = vertices[i];

ad2.weight = weight;

vertices[j]->adj.push\_back(ad2);

ADD VERTEX

void Graph::addVertex(string name) {

bool exists = false;

for(int i = 0; i <vertices.size(); i++) {

if(vertices[i]->name==name)

exists = true;

}

if(exists == false) {

vertex \*nVert = new vertex;

nVert->name = name;

vertices.push\_back(nVert);

Depth First

void dftHelper(vertex \*v) {

v->visited = true;

int adjSize = v->adj.size();

for(int i = 0; i < adjSize; i++) {

if(v->adj[i].v->visited==false) {

cout << v->adj[i].v->name << " --> ";

dftHelper(v->adj[i].v);

}

}

}

void Graph::depthFirstTraversal(string sourceVertex) {

int vSize = vertices.size();

for(int i = 0; i < vSize; i++){

if(vertices[i]-> name == sourceVertex) {

cout << vertices[i]->name << " --> ";

dftHelper(vertices[i]);

}

}

cout << "Done";

}

Breadth First

void Graph::breadthFirstTraverse(string sourceVertex) {

vertex \*vRoot;

int vSize = vertices.size();

int adjSize;

for(int i = 0; i < vSize; i++) {

if(vertices[i]->name==sourceVertex)

vRoot = vertices[i];

}

vRoot->visited=true;

queue<vertex\*> q;

q.push(vRoot);

vertex \*n;

cout << "Starting vertex (root): ";

cout << vRoot->name << "-> ";

while(!q.empty()) {

n=q.front();

adjSize = n->adj.size();

q.pop();

for(int i = 0; i < adjSize;i++) {

if(!n->adj[i].v->visited) {

n->adj[i].v->visited = true;

q.push(n->adj[i].v);

n->adj[i].v->distance = n->distance+1;

cout << n->adj[i].v->name << "(";

cout << n->adj[i].v->distance << ")" << " ";

}

}

}

}

Get Connected Components

int Graph::getConnectedComponents() {

int vSize = vertices.size();

int numSubGraphs = 0;

for(int i = 1; i < vSize; i++) {

if(vertices[i]->visited == false) {

numSubGraphs++;

bft(vertices[i]->name ,vertices);

}

}

if(numSubGraphs==0) {

if(vSize > 0)

numSubGraphs=1;

}

return numSubGraphs;

}

Check Bipartite

bool bipartiteHelper(vertex \*v, int vSize) {

int adjSize;

v->visited=true;

v->color="RED";

queue<vertex\*> q;

q.push(v);

vertex \*cur;

while(!q.empty()) {

cur=q.front();

adjSize = cur->adj.size();

q.pop();

for(int i = 0; i < adjSize;i++) {

if(!cur->adj[i].v->visited) {

cur->adj[i].v->visited = true;

q.push(cur->adj[i].v);

if(cur->color=="RED")

cur->adj[i].v->color = "BLUE";

if(cur->color=="BLUE")

cur->adj[i].v->color = "RED";

}

if(cur->color==cur->adj[i].v->color)

return false;

}

}

return true;

}

bool Graph::checkBipartite() {

int vSize = vertices.size();

bool bi;

for(int i = 0; i < vSize; i++) {

if(!vertices[i]->visited) {

bi = bipartiteHelper(vertices[i], vSize);

if(bi==false)

return false;

}

}

return true;

}

Dijkstra

vertex\* Graph::DijkstraAlgorithm(string start, string end) {

int minDist = 1000;

int SLsize;

int adjSize;

int dist;

vertex \*vStart = search(start, vertices);

vertex \*vEnd = search(end, vertices);

vertex \*solvedV = NULL;

vertex \*s = NULL;

if(vStart==NULL||vEnd==NULL)

return NULL;

vector<vertex\*>solvedList;

solvedList.push\_back(vStart);

vStart->distance = 0;

vStart->visited = true;

while(!vEnd->visited) {

minDist = 1000;

SLsize = solvedList.size();

for(int i = 0; i < SLsize; i++) {

s = solvedList[i];

adjSize = s->adj.size();

for(int x = 0; x < adjSize; x++) {

if(!s->adj[x].v->visited) {

dist = s->distance + s->adj[x].weight;

if(dist < minDist) {

solvedV = s->adj[x].v;

minDist = dist;

s->adj[x].v->pred = s;

}

}

}

}

solvedV->distance = minDist;

solvedV->visited = true;

solvedList.push\_back(solvedV);

} // While end

return vEnd;

}

**BST NOTES**

void​ ​ preorder​ ( ​ int​ \*root) {

if​ (root != ​ NULL​ ) {

print(root->data);

preorder(root->left);

preorder(root->right);

}

}

void​ ​ inorder​ ( ​ int​ \*root){

if​ (root != ​ NULL​ ){

inorder(root->left);

print(root->data);

inorder(root->right);

}

}

void​ ​ postorder​ ( ​ int​ \*root){

if(root == NULL){

return;

}

postorder(root->left);

postorder(root->right);

print(root->data);

}

**BigO**

Acc S Ins Del

BST O(logn) ” “

Hash NA O(1) O(1)

Worst case

BST O(n) “ “

Hash NA O(n) “

**Hash**

# Hash Tables

+ Definition: A data structure that stores records in a linear structure, such as array.

+ The index of the array is calculated for each element (record) by a hashing function

+ Example: If I have four records to store: Anna, Jamie, Bryan, Laura

## Two Components

- 1) Hash Function:

- [Generates Unique code]that corresponds to a valid array index.

- [Is Repeatable] hashF(Laura)=>3, [i get 3 every time]

- 2). Hash table

a. It's an array

# Types of Hash Functions // There are many kinds.

## 1) Mod-of-Sum

- sum all the values in a string, then mod by array length.

- ex: take ascii value of each letter of someones name. add them up as the sum in the hash table.

- Sudo:

int hashSumMod(key, tableSize)

sum = 0;

for i = 0 key.size()-1

sum = sum + key(i)

return sum & tableSize;

## 2) Multiplicative Hash Function:

1) Sum all the ascii vals of key;

int hash = 0;

// Sum of key should be calculated [Like mod-of-sum]

2) Multiply by a const. decimal between 0 and 1.

A = 13/32;

hash = A \* hash;

3) Keep the fractional part

hash = fract(hash); [// 3.14 -> 0.14]

4) Multiply by table size

hash = hash \* tableSize;

return hash;

## Cryptographic hash functions

- Maps a "message" of anf size to a hash val of fixed size

- One-way fn not invertible

- lots of security applications

# Operations

- [\*Store Record (insert)\*]

1. Compute the index value (hash(key))

2. Write data to the table (array) at the index

- [\*Retrieve Record (search)\*]

1. Use hash function to calculate the index

2. Read data from the array at the index.

- \*What about operation costs?\*

+ There is no need to traverse with HT

+ Does not depend on key size

+ No need to traverse other elements

+ Access for array is O(1)

\*+ [Both store and retrieve (insert and search) have a cost of O(1)!!]

- Collision

+ When 2 records hash to the same index in hash table.

## Dealing With Colisions

### Open addressing

- if location in table is occupied, find some other available location

+ Need another algo that finds an open spot.(linear probing)

+ linear probing

-finds next available location in array

```python

index = hash(record.key)

while(table[index] is occupied)

index++

table[index] = record

+ Double Hashing

\* Hash something again by multiplying hash by const.

\* [tableSize]- important member

+ Quadratic Hashing

\* Instead of looking at the next adjacent slot, skip over by i^2

indecies

// Could cause over-use of memory.

### Chaining

- Using a Linked List. Each element in hash table is head of a ll

### Clustering (Downside of linear probing)

- Elements are likely to get bunched up in one area of the table.l

## ADT

1. Hash Function

2. Hash Table

- Perfect hash function would have 0 collisions and perfectly fill a hash table

- Perfect hash functions are not realistic so we have:

1.) Open Addressing

\* Linear probing

\* Quadratic Hashing

\* Double hashing

2.) Chaining

\* Each element in table is head of LL.

## Universal Hashing

- Uses a family of hashing fn's and randomly picks a fn for each hash

## Load Factor

n is the number of records, then load factor can be calculated as: loadFactor = n/tableSize

- Keeping [loadFactor] < 1 and num of collisions minimal will give a [performance close to O(1)]