**Linear Probing**

**HashTable<TYPE>::HashTable(int maxExpected){**record = new Records[maxExpected]; capacity = maxExpected;size = 0;}

**bool HashTable<TYPE>::update(const string& key, const TYPE& values){**

std::hash<std::string> hash\_fn;size\_t address = hash\_fn(key) % capacity;

**if (size == capacity)**{Records\* newRecord = new Records[capacity \* 2];capacity = capacity \* 2;for (int i = 0; i < size; i++){if (!record[i].empty){address = hash\_fn(record[i].key) % capacity;if (newRecord[address].empty){newRecord[address].key = record[i].key;newRecord[address].value = record[i].value;

newRecord[address].empty = false;}else{int j = address;while (newRecord[j].empty == false){

j++;if (j > capacity){j = 0;}}newRecord[j].key = record[i].key;newRecord[j].value = record[i].value;

newRecord[j].empty = false;}}}delete [] record;record = newRecord;}**address = hash\_fn(key) % capacity;**

if(record[address].empty){record[address].key = key;record[address].value = values;record[address].empty = false;

size++;}else{int i = address;bool found = false;while (record[i].empty == false && !found){

if (record[i].key == key){record[i].value = values;found = true;} else{if (i > capacity){i = 0;}else{i++;}}}if (!found){record[i].key = key;record[i].value = values;record[i].empty = false;size++;}}return true;}

**bool HashTable<TYPE>::remove(const string& key){**

std::hash<std::string> hash\_fn;size\_t address = hash\_fn(key) % capacity;bool found = false;int emptySpot = address;int i;if (!record[address].empty){

if (record[address].key == key){record[address].key = "";record[address].value = TYPE{};record[address].empty = true;size--;found = true;i = address + 1;

if (i > capacity){i = 0;}}else{i = address;while (!record[i].empty && !found){if (record[i].key == key){record[i].key = "";record[i].value = TYPE{};

record[i].empty = true;size--;found = true;break;}i++;if (i > capacity){i = 0;}}address = i;i++;if (i > capacity){i = 0;}}if (found){

while (!record[i].empty){emptySpot = hash\_fn(record[i].key) % capacity;if ( emptySpot <= address){record[address].key = record[i].key;

record[address].value = record[i].value;record[address].empty = false;record[i].key = "";record[i].value = TYPE{};record[i].empty = true;

address = i;i++;if (i > capacity){i = 0;}}else{i++;if (i > capacity){i = 0;}}}}}return found;}

**bool HashTable<TYPE>::find(const string& key, TYPE& value){**std::hash<std::string>hash\_fn;size\_t address = hash\_fn(key) % capacity;

bool found = false;if (!record[address].empty){if (record[address].key == key){value = record[address].value;found = true;}else{

int i = address;while (!record[i].empty){if (record[i].key == key){value = record[i].value;found = true;break;}i++;if (i > capacity){

i = 0;}}}}return found;}

**Queue**

**void enqueue(int data)**{if (filled != size - 1){arry[back] = data;back = (back + 1) % size;filled++;}

else{int newSize = size \* 2;int\* newArry = new int[newSize];int j = 0;int tmpFront = front;

int newBack = 0;for (int i = filled; i >= 0 && tmpFront != size - 1; i--){newArry[j++] = arry[tmpFront];

tmpFront = (tmpFront + 1) % newSize;++newBack;}newArry[j] = data;filled++;delete arry;arry = newArry;

front = 0;back = newBack + 1;size = newSize;}}

**void dequeue()**{;front = (front + 1) % size;filled--;}

**BinarySearchTree**

**BST(BST& cpy)**{Node\* source = cpy.root\_;Node\* destination = root\_;

root\_ = copy(destination, source**);}Node\* copy(Node\*& destination, Node\*& source)**{

if (source){destination = new Node(source->data\_);copy(destination->right\_, source->right\_);

copy(destination->left\_, source->left\_);}return destination;}

**bool operator==(const BST& obj)**{Node\* obj1 = root\_;Node\* obj2 = obj.root\_;

return compare(obj1, obj2**);}bool compare(Node\*& n1, Node\*& n2)**{

if (n1 && n2){if (n1->data\_ == n2->data\_){compare(n1->left\_, n2->left\_);

compare(n1->right\_, n2->right\_);}else{return false;}}

else if (!n1 && n2 || n1 && !n2){return false;}else{return true;}}

**void removeMain(int data**){Node\* delteAndReplace = root\_;

bool rootChange = false;if (root\_->data\_ == data){

rootChange = true;}**remove(data, delteAndReplace);**

if (rootChange){root\_ = delteAndReplace;}}**void remove(int data, Node\*& subrootTree)**{

Node\* srch = subrootTree;if (srch){bool found = false;if (srch->data\_ == data){

if (srch->left\_ && !srch->right\_){subrootTree = subrootTree->left\_;}

else if (!srch->left\_ && srch->right\_){subrootTree = subrootTree->right\_;}

else{Node\* rc = detatchSmallest(subrootTree->right\_);rc->left\_ = subrootTree->left\_;

rc->right\_ = subrootTree->right\_;subrootTree = rc;}delete srch;srch = nullptr;

}else{if (srch->data\_ < data){remove(data, subrootTree->right\_);}else{

remove(data, subrootTree->left\_);}}}}Node\* **detatchSmallest(Node\*& n){**

if (!n->left\_){Node\* tmp = n;n = n->right\_;return tmp;}return detatchSmallest(n->left\_);}

**void printBetween(int min, int max)**{Node\* prntBetwn = root\_;printBetweenRecursive(min, max, prntBetwn);}

**void printBetweenRecursive(const int min, const int max, Node\*& prntVal**){if (prntVal){

if (prntVal->data\_ <= max && prntVal->data\_ >= min){std::cout << prntVal->data\_ << std::endl;}

if (prntVal->left\_){if (prntVal->left\_->data\_ >= min){printBetweenRecursive(min, max, prntVal->left\_);}}

if (prntVal->right\_){if (prntVal->right\_->data\_ <= max){printBetweenRecursive(min, max, prntVal->right\_);}}}}

**int numOfNodesInTree()**{Node\* srch = root\_;int count = 0;

**numOfNodesInTreeRecursive(srch, count);** return count;}

**void numOfNodesInTreeRecursive(Node\*& srch, int& count)**{

if (srch){++count;numOfNodesInTreeRecursive(srch->left\_, count);

numOfNodesInTreeRecursive(srch->right\_, count);}}

**void printInOrder()**{Node\* prnt = root\_;inOrderPrint(prnt);}

**void inOrderPrint(Node\*& subtreeroot)**{if (subtreeroot){

inOrderPrint(subtreeroot->left\_);std::cout << subtreeroot->data\_ << std::endl;

inOrderPrint(subtreeroot->right\_);}}

**void breadthFirst()**{Node\* curr;Queue<Node\*>q;

q.enqueue(root\_);while (!q.isEmpty()){

curr = q.frontVal();q.dequeue();if (curr){

cout << curr->data\_ << endl;q.enqueue(curr->left\_);

q.enqueue(curr->right\_);}}}

**void postOrder()**{Node\* prnt = root\_;postOrderRecursive(prnt);}

**void postOrderRecursive(Node\* pstOrdr)**{if (pstOrdr){

postOrderRecursive(pstOrdr->left\_);postOrderRecursive(pstOrdr->right\_);

std::cout << pstOrdr->data\_ << std::endl;}}

**void preOrder()**{Node\* prnt = root\_;preOrderRecursive(prnt);}

**void preOrderRecursive(Node\* preOrdr)**{if (preOrdr){

std::cout << preOrdr->data\_ << std::endl;

preOrderRecursive(preOrdr->left\_);preOrderRecursive(preOrdr->right\_);}}

**Simple Sort**

**void selectionSort(int arr[],int size)**{

int minIdx;int tmp;for(int i=0;i<size;i++){

minIdx=i;for(int j=i;j<size;j++){

if(arr[j] < arr[minIdx]){minIdx=j;}}tmp=arr[i];

arr[i]=arr[minIdx];arr[minIdx]=tmp;}}

**void insertionSort(int arr[],int size)**{int curr;

int i,j;for(i=0;i<size;i++){curr=arr[i];

for(j=i;j>0 && arr[j-1] > curr;j--){

arr[j]=arr[j-1];}arr[j]=curr;}}

**void bubble(int array[],int sz)**{int i,j;

int tmp;for(i=0;i<sz-1;i++){for(j=0;j<sz-i-1;j++){

if(array[j] > array[j+1]){tmp=array[j];

array[j]=array[j+1];array[j+1]=tmp;}}}}

**int BinarySearch(int arry[], int size, int key){**int rc = -1;

int high = size - 1;int low = 0;int mid = 0;while (low <= high && rc == -1){

mid = (low + high) / 2;if (arry[mid] > key){high = mid - 1;}

else if (arry[mid] < key){low = mid + 1;}else{rc = mid;}}return rc;}

**int BinarySearchRecursion(int arry[], int high, int low, int key)**{

int mid = (low + high) / 2;int rc = -1;if (low <= high){

if (arry[mid] > key){high = mid - 1;} else if (arry[mid] < key){

low = mid + 1;} else{rc = mid;}if(rc == -1){

rc = BinarySearchRecursion(arry, high, low, key);}}return rc;}

**int BinarySearch(int arry[], int size, int key)**{int rc = -1;

int high = size - 1;int low = 0;rc = BinarySearchRecursion(arry, high, low, key);

return rc;}

**int LinearSearchRecursion(int arry[], int size, int key){**int rc;

if (arry[size] == key){rc = size;}else if (size > 0){

rc = LinearSearchRecursion(arry, size - 1, key);}return rc;}

**Rotation**

**void leftRotate(Node\*& A)**{Node\* B=A->right\_;

Node\* y=B->left\_;A->right\_= y;

B->left\_=A; A=B;}

**void rightRotate(Node\*& A){**Node\* B = A->left\_;

Node\* y = B->right\_;A->left\_ = y;B->right\_ = A;A = B;}

**Red Black Tree Cases**

**Case 1:** uncle is red = color fix

**Case 2:** uncle is black and right of left child or left of right child = rotate parent and rotate grand parent

**Case 3:** uncle is black and left of left child or right of right child = rotate grand parent

**Definitions**

**edge weight/edge cost** - a value associated with a connection between two nodes

**path -** a ordered sequence of vertices where a connection must exist between consecutive pairs in the sequence.

**simplepath -** every vertex in path is distinct

**pathlength** number of edges in a path

cycle - a path where the starting and ending node is the same

**strongly connected -** If there exists some path from every vertex to every other vertex, the graph is strongly connected.

**weakly connect -** if we take away the direction of the edges and there exists a path from every node to every other node, the digraph is weakly connected**.**

**Node:** data stored by the tree (All the circles)

**Root Node:** the top most node from which all other nodes come from.

**Child:** Root node of a subtree of a node is the child node Ex. B is the child of A. I is the child of H

**Parent :** Opposite of a child node Ex. A is parent of B. H is the parent of I

**Ancestor:** All nodes that can be reached by moving only in an upward direction in the tree. Ex. C, A and H are all ancestors of I but G and B are not.

**Descendants or Successors** of a node are nodes that can be reached by only going down in the tree.

**Levels:** Nodes in Level 0 of our tree is A.

**Path:** Set of branches taken to connect an ancestor of a node to the node.