# Applied Programming



# Name: M. Hassan Sattar

# Roll No: 22F-3773

**Section: MS-1A**

# Submitted to: Sir. Muhammad Yousaf

**Q-1**

**a. Linear Probing**

#include <iostream>

#include <cstdio>

#include <cstdlib>

using namespace std;

const int T\_S = 15;

class HashTable {

public:

int k;

int v;

HashTable(int k, int v) {

this->k = k;

this->v = v;

}

};

class DelNode:public HashTable {

private:

static DelNode \*en;

DelNode():HashTable(-1, -1) {}

public:

static DelNode \*getNode() {

if (en == NULL)

en = new DelNode();

return en;

}

};

DelNode \*DelNode::en = NULL;

class HashMapTable {

private:

HashTable \*\*ht;

public:

HashMapTable() {

ht = new HashTable\* [T\_S];

for (int i = 0; i < T\_S; i++) {

ht[i] = NULL;

}

}

int HashFunc(int k) {

return k % T\_S;

}

void Insert(int k, int v) {

int hash\_val = HashFunc(k);

int init = -1;

int delindex = -1;

while (hash\_val != init && (ht[hash\_val] == DelNode::getNode() || ht[hash\_val] != NULL && ht[hash\_val]->k != k)) {

if (init == -1)

init = hash\_val;

if (ht[hash\_val] == DelNode::getNode())

delindex = hash\_val;

hash\_val = HashFunc(hash\_val + 1);

}

if (ht[hash\_val] == NULL || hash\_val == init) {

if(delindex != -1)

ht[delindex] = new HashTable(k, v);

else

ht[hash\_val] = new HashTable(k, v);

}

if(init != hash\_val) {

if (ht[hash\_val] != DelNode::getNode()) {

if (ht[hash\_val] != NULL) {

if (ht[hash\_val]->k== k)

ht[hash\_val]->v = v;

}

} else

ht[hash\_val] = new HashTable(k, v);

}

}

int SearchKey(int k) {

int hash\_val = HashFunc(k);

int init = -1;

while (hash\_val != init && (ht[hash\_val] == DelNode::getNode() || ht[hash\_val] != NULL && ht[hash\_val]->k!= k)) {

if (init == -1)

init = hash\_val;

hash\_val = HashFunc(hash\_val + 1);

}

if (ht[hash\_val] == NULL || hash\_val == init)

return -1;

else

return ht[hash\_val]->v;

}

void Remove(int k) {

int hash\_val = HashFunc(k);

int init = -1;

while (hash\_val != init && (ht[hash\_val] == DelNode::getNode() || ht[hash\_val] != NULL && ht[hash\_val]->k!= k)) {

if (init == -1)

init = hash\_val;

hash\_val = HashFunc(hash\_val + 1);

}

if (hash\_val != init && ht[hash\_val] != NULL) {

delete ht[hash\_val];

ht[hash\_val] = DelNode::getNode();

}

}

~HashMapTable() {

delete[] ht;

}

};

int main() {

HashMapTable hash;

int k, v;

int c;

while(1) {

cout<<"1.Insert element into the table"<<endl;

cout<<"2.Search element from the key"<<endl;

cout<<"3.Delete element at a key"<<endl;

cout<<"4.Exit"<<endl;

cout<<"Enter your choice: ";

cin>>c;

switch(c) {

case 1:

cout<<"Enter element to be inserted: ";

cin>>v;

cout<<"Enter key at which element to be inserted: ";

cin>>k;

hash.Insert(k, v);

break;

case 2:

cout<<"Enter key of the element to be searched: ";

cin>>k;

if(hash.SearchKey(k) == -1) {

cout<<"No element found at key "<<k<<endl;

continue;

} else {

cout<<"Element at key "<<k<<" : ";

cout<<hash.SearchKey(k)<<endl;

}

break;

case 3:

cout<<"Enter key of the element to be deleted: ";

cin>>k;

hash.Remove(k);

break;

case 4:

exit(1);

default:

cout<<"\nEnter correct option\n";

}

}

return 0;

}

**b. Linear probing with Step Size**

#include <iostream>

#include <cstdio>

#include <cstdlib>

using namespace std;

const int T\_S = 15;

class HashTable {

public:

int k;

int v;

HashTable(int k, int v) {

this->k = k;

this->v = v;

}

};

class DelNode:public HashTable {

private:

static DelNode \*en;

DelNode():HashTable(-1, -1) {}

public:

static DelNode \*getNode() {

if (en == NULL)

en = new DelNode();

return en;

}

};

DelNode \*DelNode::en = NULL;

class HashMapTable {

private:

HashTable \*\*ht;

public:

HashMapTable() {

ht = new HashTable\* [T\_S];

for (int i = 0; i < T\_S; i++) {

ht[i] = NULL;

}

}

int HashFunc(int k) {

return k % T\_S;

}

void Insert(int k, int v) {

int hash\_val = HashFunc(k);

int init = -1;

int delindex = -1;

while (hash\_val != init && (ht[hash\_val] == DelNode::getNode() || ht[hash\_val] != NULL && ht[hash\_val]->k != k)) {

if (init == -1)

init = hash\_val;

if (ht[hash\_val] == DelNode::getNode())

delindex = hash\_val;

hash\_val = HashFunc(hash\_val + 1);

}

if (ht[hash\_val] == NULL || hash\_val == init) {

if(delindex != -1)

ht[delindex] = new HashTable(k, v);

else

ht[hash\_val] = new HashTable(k, v);

}

if(init != hash\_val) {

if (ht[hash\_val] != DelNode::getNode()) {

if (ht[hash\_val] != NULL) {

if (ht[hash\_val]->k== k)

ht[hash\_val]->v = v;

}

} else

ht[hash\_val] = new HashTable(k, v);

}

}

int SearchKey(int k) {

int hash\_val = HashFunc(k);

int init = -1;

while (hash\_val != init && (ht[hash\_val] == DelNode::getNode() || ht[hash\_val] != NULL && ht[hash\_val]->k!= k)) {

if (init == -1)

init = hash\_val;

hash\_val = HashFunc(hash\_val + 1);

}

if (ht[hash\_val] == NULL || hash\_val == init)

return -1;

else

return ht[hash\_val]->v;

}

void Remove(int k) {

int hash\_val = HashFunc(k);

int init = -1;

while (hash\_val != init && (ht[hash\_val] == DelNode::getNode() || ht[hash\_val] != NULL && ht[hash\_val]->k!= k)) {

if (init == -1)

init = hash\_val;

hash\_val = HashFunc(hash\_val + 1);

}

if (hash\_val != init && ht[hash\_val] != NULL) {

delete ht[hash\_val];

ht[hash\_val] = DelNode::getNode();

}

}

~HashMapTable() {

delete[] ht;

}

};

int main() {

HashMapTable hash;

int k, v;

int c;

while(1) {

cout<<"1.Insert element into the table"<<endl;

cout<<"2.Search element from the key"<<endl;

cout<<"3.Delete element at a key"<<endl;

cout<<"4.Exit"<<endl;

cout<<"Enter your choice: ";

cin>>c;

switch(c) {

case 1:

cout<<"Enter element to be inserted: ";

cin>>v;

cout<<"Enter key at which element to be inserted: ";

cin>>k;

hash.Insert(k, v);

break;

case 2:

cout<<"Enter key of the element to be searched: ";

cin>>k;

if(hash.SearchKey(k) == -1) {

cout<<"No element found at key "<<k<<endl;

continue;

} else {

cout<<"Element at key "<<k<<" : ";

cout<<hash.SearchKey(k)<<endl;

}

break;

case 3:

cout<<"Enter key of the element to be deleted: ";

cin>>k;

hash.Remove(k);

break;

case 4:

exit(1);

default:

cout<<"\nEnter correct option\n";

}

}

return 0;

}

**c. Quadratic Probing**

#include <iostream>

#include <cstdlib>

#define T\_S 10

using namespace std;

enum EntryType {

Legi, Emp, Del};

struct HashTableEntry {

int e;

enum EntryType info;

};

struct HashTable {

int s;

HashTableEntry \*t;

};

bool isPrime (int n) {

if (n == 2 || n == 3)

return true;

if (n == 1 || n % 2 == 0)

return false;

for (int i = 3; i \* i <= n; i += 2)

if (n % i == 0)

return false;

return true;

}

int nextPrime(int n) {

if (n <= 0)

n == 3;

if (n % 2 == 0)

n++;

for (; !isPrime( n ); n += 2);

return n;

}

int HashFunc(int k, int s) {

return k % s;

}

HashTable \*initiateTable(int s) {

HashTable \*ht;

if (s < T\_S) {

cout<<"Table Size is Too Small"<<endl;

return NULL;

}

ht= new HashTable;

if (ht == NULL) {

cout<<"Out of Space"<<endl;

return NULL;

}

ht->s = nextPrime(s);

ht->t = new HashTableEntry [ht->s];

if (ht->t == NULL) {

cout<<"Table Size is Too Small"<<endl;

return NULL;

}

for (int i = 0; i < ht->s; i++) {

ht->t[i].info = Emp;

ht->t[i].e = NULL;

}

return ht;

}

int SearchKey(int k, HashTable \*ht) {

int pos = HashFunc(k, ht->s);

int collisions = 0;

while (ht->t[pos].info != Emp && ht->t[pos].e != k) {

pos = pos + 2 \* ++collisions -1;

if (pos >= ht->s)

pos = pos - ht->s;

}

return pos;

}

void Insert(int k, HashTable \*ht) {

int pos = SearchKey(k, ht);

if (ht->t[pos].info != Legi) {

ht->t[pos].info = Legi;

ht->t[pos].e = k;

}

}

HashTable \*Rehash(HashTable \*ht) {

int s = ht->s;

HashTableEntry \*t= ht->t;

ht= initiateTable(2 \* s);

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

if (t[i].info == Legi)

Insert(t[i].e, ht);

}

free(t);

return ht;

}

void display(HashTable \*ht) {

for (int i = 0; i < ht->s; i++) {

int value = ht->t[i].e;

if (!value)

cout<<"Position: "<<i + 1<<" Element: Null"<<endl;

else

cout<<"Position: "<<i + 1<<" Element: "<<value<<endl;

}

}

int main() {

int v, s, pos, i = 1;

int c;

HashTable \*ht;

while(1) {

cout<<"1.Initialize size of the table"<<endl;

cout<<"2.Insert element into the table"<<endl;

cout<<"3.Display Hash Table"<<endl;

cout<<"4.Rehash The Table"<<endl;

cout<<"5.Exit"<<endl;

cout<<"Enter your choice: ";

cin>>c;

switch(c) {

case 1:

cout<<"Enter size of the Hash Table: ";

cin>>s;

ht = initiateTable(s);

cout<<"Size of Hash Table: "<<nextPrime(s);

break;

case 2:

if (i > ht->s) {

cout<<"Table is Full, Rehash the table"<<endl;

continue;

}

cout<<"Enter element to be inserted: ";

cin>>v;

Insert(v, ht);

i++;

break;

case 3:

display(ht);

break;

case 4:

ht = Rehash(ht);

break;

case 5:

exit(1);

default:

cout<<"\nEnter correct option\n";

}

}

return 0;

}

**Q-2**

**a. Chaining**

#include<stdio.h>

#include<stdlib.h>

#define size 15

struct node

{

int data;

struct node \*next;

};

struct node \*chain[size];

void init()

{

int i;

for(i = 0; i < size; i++)

chain[i] = NULL;

}

void insert(int value)

{

//create a newnode with value

struct node \*newNode = malloc(sizeof(struct node));

newNode->data = value;

newNode->next = NULL;

//calculate hash key

int key = value % size;

//check if chain[key] is empty

if(chain[key] == NULL)

chain[key] = newNode;

//collision

else

{

//add the node at the end of chain[key].

struct node \*temp = chain[key];

while(temp->next)

{

temp = temp->next;

}

temp->next = newNode;

}

}

void print()

{

int i;

for(i = 0; i < size; i++)

{

struct node \*temp = chain[i];

printf("chain[%d]-->",i);

while(temp)

{

printf("%d -->",temp->data);

temp = temp->next;

}

printf("NULL\n");

}

}

int main()

{

init();

insert(17);

insert(26);

insert(15);

insert(9);

insert(11);

insert(43);

insert(75);

insert(19);

insert(35);

insert(45);

insert(55);

insert(9);

insert(10);

insert(21);

insert(61);

insert(23);

print();

return 0;

}

**b. Bucketing**

#include <iomanip>

#include <iostream>

using namespace std;

#define NARRAY 15

#define NBUCKET 3

#define INTERVAL 15

struct Node {

int data;

struct Node \*next;

};

void BucketSort(int arr[]);

struct Node \*InsertionSort(struct Node \*list);

void print(int arr[]);

void printBuckets(struct Node \*list);

int getBucketIndex(int value);

void BucketSort(int arr[]) {

int i, j;

struct Node \*\*buckets;

buckets = (struct Node \*\*)malloc(sizeof(struct Node \*) \* NBUCKET);

for (i = 0; i < NBUCKET; ++i) {

buckets[i] = NULL;

}

for (i = 0; i < NARRAY; ++i) {

struct Node \*current;

int pos = getBucketIndex(arr[i]);

current = (struct Node \*)malloc(sizeof(struct Node));

current->data = arr[i];

current->next = buckets[pos];

buckets[pos] = current;

}

// Print the buckets along with their elements

for (i = 0; i < NBUCKET; i++) {

cout << "Bucket[" << i << "] : ";

printBuckets(buckets[i]);

cout << endl;

}

// Sort the elements of each bucket

for (i = 0; i < NBUCKET; ++i) {

buckets[i] = InsertionSort(buckets[i]);

}

cout << "-------------" << endl;

cout << "Bucktets after sorted" << endl;

for (i = 0; i < NBUCKET; i++) {

cout << "Bucket[" << i << "] : ";

printBuckets(buckets[i]);

cout << endl;

}

// Put sorted elements on arr

for (j = 0, i = 0; i < NBUCKET; ++i) {

struct Node \*node;

node = buckets[i];

while (node) {

arr[j++] = node->data;

node = node->next;

}

}

for (i = 0; i < NBUCKET; ++i) {

struct Node \*node;

node = buckets[i];

while (node) {

struct Node \*tmp;

tmp = node;

node = node->next;

free(tmp);

}

}

free(buckets);

return;

}

// Function to sort the elements of each bucket

struct Node \*InsertionSort(struct Node \*list) {

struct Node \*k, \*nodeList;

if (list == 0 || list->next == 0) {

return list;

}

nodeList = list;

k = list->next;

nodeList->next = 0;

while (k != 0) {

struct Node \*ptr;

if (nodeList->data > k->data) {

struct Node \*tmp;

tmp = k;

k = k->next;

tmp->next = nodeList;

nodeList = tmp;

continue;

}

for (ptr = nodeList; ptr->next != 0; ptr = ptr->next) {

if (ptr->next->data > k->data)

break;

}

if (ptr->next != 0) {

struct Node \*tmp;

tmp = k;

k = k->next;

tmp->next = ptr->next;

ptr->next = tmp;

continue;

} else {

ptr->next = k;

k = k->next;

ptr->next->next = 0;

continue;

}

}

return nodeList;

}

int getBucketIndex(int value) {

return value / INTERVAL;

}

// Print buckets

void print(int ar[]) {

int i;

for (i = 0; i < NARRAY; ++i) {

cout << setw(3) << ar[i];

}

cout << endl;

}

void printBuckets(struct Node \*list) {

struct Node \*cur = list;

while (cur) {

cout << setw(3) << cur->data;

cur = cur->next;

}

}

int main(void) {

int array[NARRAY] = {17, 26, 15, 9, 11, 43, 75, 19, 35, 45, 55, 9, 10, 21, 61};

cout << "Initial array: " << endl;

print(array);

cout << "-------------" << endl;

BucketSort(array);

cout << "-------------" << endl;

cout << "Sorted array: " << endl;

print(array);

}