Experiment No. : 7

Title: Map implementation

Batch: B1 **Roll No.: 16010420133** **Experiment No.: 6**

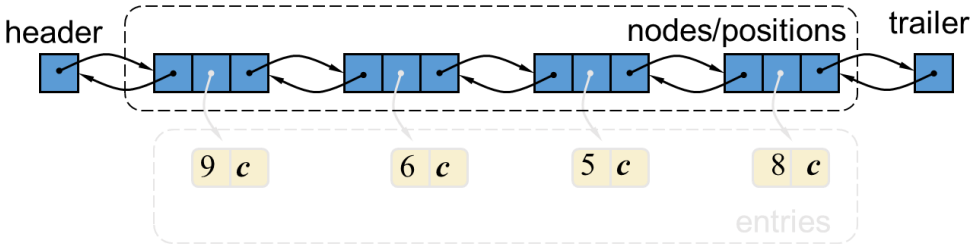
Aim: Write a menu driven program to implement a map using linked list.

Resources Used: C/ C++ editor and compiler.

Theory:

MAP

A map allows us to store elements so they can be located quickly using keys. The motivation for such searches is that each element typically stores additional useful information besides its search key, but the only way to get at that information is to use the search key. Specifically, a map stores key-value pairs (k,v), which we call entries, where k is the key and v is its corresponding value. In addition, the map ADT requires that each key be unique, so the association of keys to values defines a mapping. In order to achieve the highest level of generality, we allow both the keys and the values stored in a map to be of any object type. In a map storing student records (such as the student's name, address, and course grades), the key might be the student's ID number. In some applications, the key and the value may be the same.



For example, if we had a map storing prime numbers, we could use each number itself as both a key and its value. In either case, we use a key as a unique identifier that is assigned by an application or user to an associated value object. Thus, a map is most appropriate in situations where each key is to be viewed as a kind of unique index address for its value, that is, an object that serves as a kind of location for that value. For example, if we wish to store student records, we would probably want to use student ID objects as keys (and disallow two students having the same student ID). In other words, the key associated with an object can be viewed as an "address" for that object. Indeed, maps are sometimes referred to as associative stores or associative containers, because the key associated with an object determines its "location" in the data structure

---

Algorithm :

createMap() : Creates a map and initialize it to empty map.

Int size(): Return the number of elements in the map.

Boolean empty(): Return true if the map is empty and false otherwise.

Typedef find(int k): Find the entry with key k and return an iterator to it; if no such key exists return end.

Void insert(pair(k,v)): If key k is already present on the Map, replaces the value with v; otherwise inserts the pair (k,v) at the beginning.

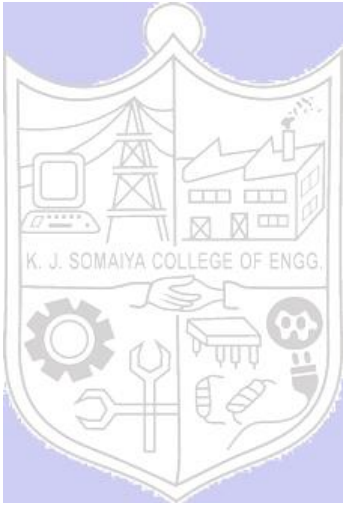
Typedef erase(int k): Remove the element with key k and return associated value.

Typedef begin(): Return an iterator to the beginning of the map.

Tyepdef end(): Return an iterator just past the end of the map

Results:

```
#include<iostream>

#include<cstdlib>

#include<string>

#include<cstdio>

using namespace std;

const int TABLE_SIZE = 128;

class HashNode

{

public:

int key;

int value; HashNode* next;

HashNode(int key, int value)

{
```



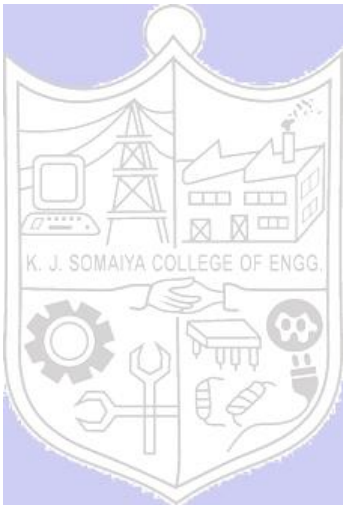
```
this->key = key;

this->value = value; this->next = NULL;

}

};

class HashMap

{

private:

HashNode** htable; public:

HashMap()

{

htable = new HashNode*[TABLE_SIZE]; for (int i = 0; i < TABLE_SIZE; i++)

htable[i] = NULL;

}

~HashMap()

{

for (int i = 0; i < TABLE_SIZE; ++i)

{

HashNode* entry = htable[i]; while (entry != NULL)

{
```

```
HashNode* prev = entry; entry = entry->next; delete prev;

}

}

delete[] htable;

}

int HashFunc(int key)

{

return key % TABLE_SIZE;

}

int size(int key) {

int hash_val = HashFunc(key);

cout<<(hash_val+1)<<" is the size."<<endl;

}

int empty(int key) {

if(key==-1)

{

cout<<"We are currently empty"<<endl;

}

else {
```

```
cout<<"We are having elements in map"<<endl;
}
}
void Insert(int key, int value)
{
int hash_val = HashFunc(key);
HashNode* prev = NULL;
HashNode* entry = htable[hash_val];
while (entry != NULL)
{
prev = entry;
entry = entry->next;
}
if (entry == NULL)
{
entry = new HashNode(key, value);
if (prev == NULL)
{
htable[hash_val] = entry;
```

```
}
else
{
prev->next = entry;
}
}
else
{
entry->value = value;
}
}
void Remove(int key)
{
int hash_val = HashFunc(key); HashNode* entry = htable[hash_val]; HashNode* prev = NULL;
if (entry == NULL || entry->key != key)
{
cout<<"No Element found at key "<<key<<endl; return;
}
while (entry->next != NULL)
```

```
{
prev = entry;
entry = entry->next;
}
if (prev != NULL)
{
prev->next = entry->next;
}
delete entry;
cout<<"Element Deleted"<<endl;
}
int Search(int key)
{
bool flag = false;
int hash_val = HashFunc(key); HashNode* entry = htable[hash_val]; while (entry != NULL)
{
if (entry->key == key)
{
cout<<entry->value<<" "; flag = true;
```



```
}
entry = entry->next;
}
if (!flag) return -1;
}
};
int main()
{
HashMap hash; int key=-1, value; int choice; while (1)
{
cout<<"\n "<<endl;
cout<<"Operations on Hash Table"<<endl; cout<<"\n "<<endl;
cout<<"1.Insertion"<<endl; cout<<"2.Search element from the key"<<endl;
cout<<"3.Deletion"<<endl; cout<<"4.Size \n5.Empty \n6.Exit"<<endl;
cout<<"Enter your choice: ";
cin>>choice; switch(choice)
{
case 1:
cout<<"Enter element to be inserted: "; cin>>value;
```

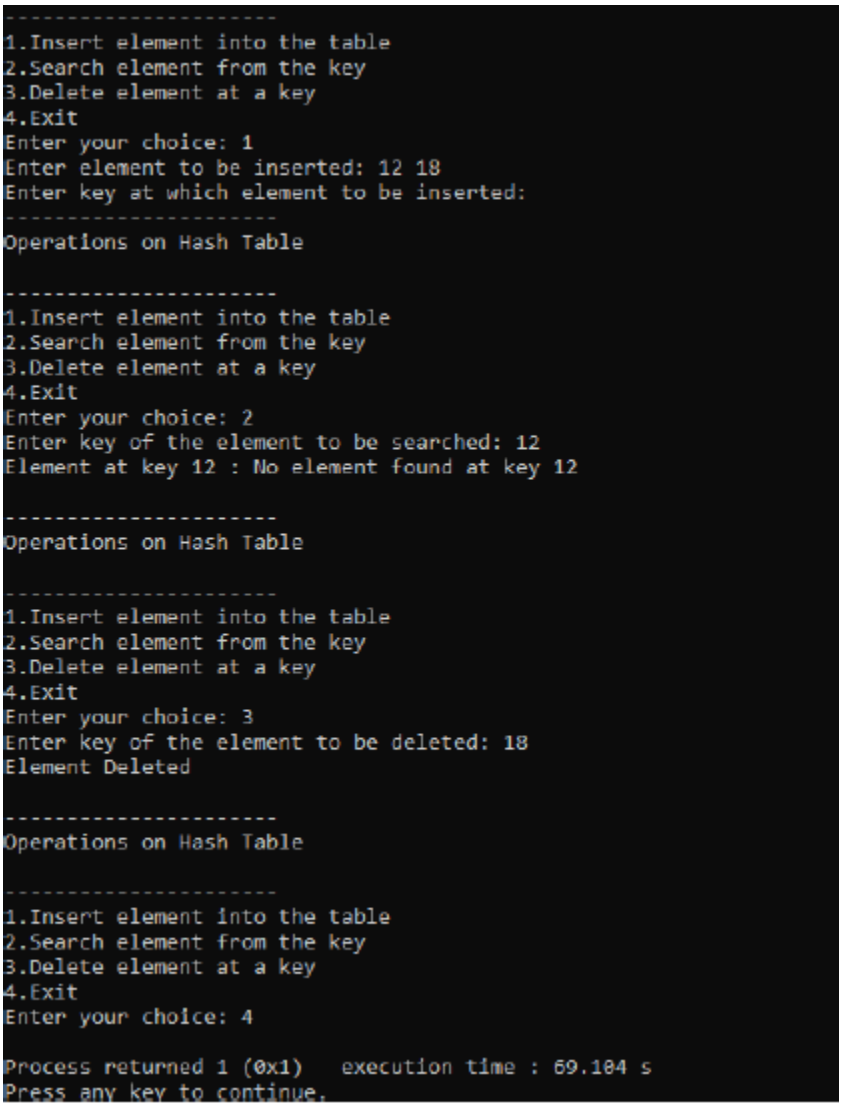
```
cout<<"Enter key at which element to be inserted: "; cin>>key;

hash.Insert(key, value); break;

case 2:

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

cout<<"Element at key "<<key<<" : "; if (hash.Search(key) == -1)

{

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

}

break; case 3:

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

hash.Remove(key); break;

case 4:

hash.size(key); break;

case 5:

hash.empty(key); break;

case 6:

exit(1); default:

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

}
```

}

return 0;

}

****OUTPUT****



A program depicting the Map behaviour using singly/doubly linked list and capable of handling all possible boundary conditions and the same is reflected clearly in the output.

Outcomes:

Describe concepts of advance data structures like set, map & dictionary

Conclusion:

We have successfully implemented a map using the concept of linked list.

Grade: AA / AB / BB / BC / CC / CD /DD

Signature of faculty in-charge with date

References:

Books/ Journals/ Websites:

- Michael T. Goodrich, Roberto Tamassia, and David M. Mount. 2009. Data Structures and Algorithms in C++ (2nd. ed.). Wiley Publishing.