Assignment:

Data Structures

Report on Semester Assignment.

Submitted by:

Name: Fahad Naeem

Roll No: **MCSM-19-08**

Class: MCS Morning

Session: 2019-2021

Date: February 8, 2021

Submitted to:

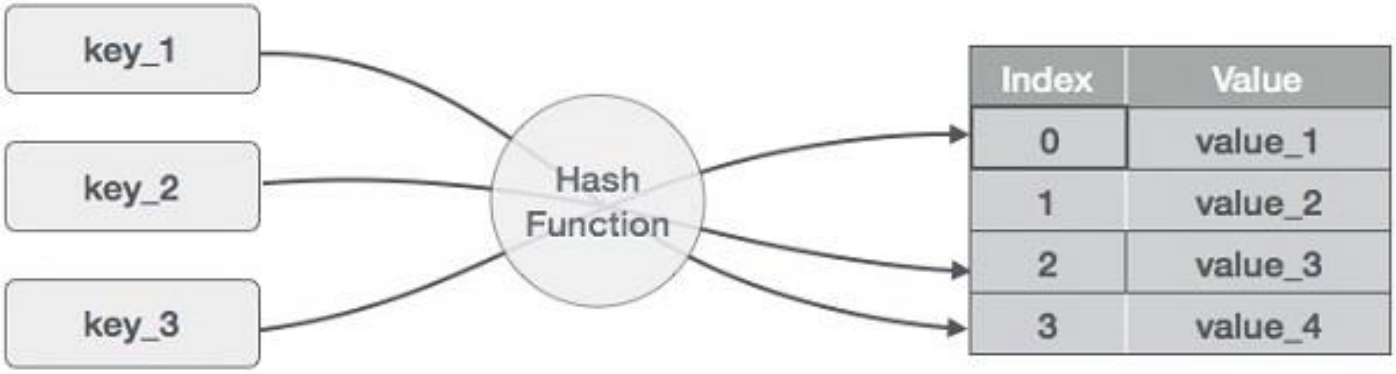
Dr. M. Asif

Department of Computer Science, Bahauddin Zakariya University Multan

Hashing

"Hashing is a technique to convert a range of key values into a range of indexes of an array." It is an important Data Structure which is designed to use a special function called the **Hash function** which is used to map a given value with a particular key for faster access of elements. The efficiency of mapping depends of the efficiency of the hash function used.

We're going to use modulo operator to get a range of key values. Consider an example of hash table of size 20, and the following items are to be stored. Item are in the **(key,value)** format.



- ☐ (1,20)
- ☐ (2,70)
- ☐ (42,80)
- ☐ (4,25)
- ☐ (12,44)
- ☐ (14,32)
- ☐ (17,11)
- ☐ (13,78)
- ☐ (37,98)

Sr.No.	Key	Hash	Array Index
1	1	1 % 20 = 1	1
2	2	2 % 20 = 2	2
3	42	42 % 20 = 2	2
4	4	4 % 20 = 4	4
5	12	12 % 20 = 12	12
6	14	14 % 20 = 14	14
7	17	17 % 20 = 17	17
8	13	13 % 20 = 13	13
9	37	37 % 20 = 17	17

A hash table is a data structure that is used to store keys/value pairs. It uses a hash function to compute an index into an array in which an element will be inserted or searched. By using a good hash function, hashing can work well.

Time complexity in big O notation

Algorithm	Average	Worst case
Search	O(1)	O(n)
Insert	O(1)	O(n)
Delete	O(1)	O(n)

Hash Collisions

Hash collisions are practically unavoidable when hashing a random subset of a large set of possible keys. For example, if 2,450 keys are hashed into a million buckets, even with a perfectly uniform random distribution, according to the birthday problem there is approximately a 95% chance of at least two of the keys being hashed to the same slot.

Resolving Hash Collisions

Almost all hash table implementations have some collision resolution strategy to handle such events. Some common strategies are described below. All these methods require that the keys (or pointers to them) be stored in the table, together with the associated values.

Types of Collisions Resolving

- Open addressing (closed hashing)
  - Linear probing
  - Quadratic probing
  - Double hashing
- Closed addressing (open hashing)
  - Chaining

Linear probing

Linear probing is a scheme in computer programming for resolving collisions in hash tables, data structures for maintaining a collection of key–value pairs and looking up the value associated with a given key. It was invented in 1954 by Gene Amdahl, Elaine M. McGraw, and Arthur Samuel and first analyzed in 1963 by Donald Knuth.

Linear probing is a form of open addressing. In these schemes, each cell of a hash table stores a single key–value pair. When the hash function causes a collision by mapping a new key to a cell of the hash table that is already occupied by another key, linear probing searches the table for the closest following free location and inserts the new key there. Lookups are performed in the same way, by searching the table sequentially starting at the position given by the hash function, until finding a cell with a matching key or an empty cell.

Quadratic probing

Quadratic probing is an open addressing scheme in computer programming for resolving hash collisions in hash tables. Quadratic probing operates by taking the original hash index and adding successive values of an arbitrary quadratic polynomial until an open slot is found.

An example sequence using quadratic probing is:

$$H + 1^2, H + 2^2, H + 3^2, H + 4^2, \dots\dots\dots, H + k^2$$

Quadratic probing can be a more efficient algorithm in an open addressing table, since it better avoids the clustering problem that can occur with linear probing, although it is not immune. It also provides good memory caching because it preserves some locality of reference; however, linear probing has greater locality and, thus, better cache performance.

Double hashing

Double hashing is a computer programming technique used in conjunction with open-addressing in hash tables to resolve hash collisions, by using a secondary hash of the key as an offset when a collision occurs. Double hashing with open addressing is a classical data structure on a table T.

The double hashing technique uses one hash value as an index into the table and then repeatedly steps forward an interval until the desired value is located, an empty location is reached, or the entire table has been searched; but this interval is set by a second, independent hash function. Unlike the alternative collision-resolution methods of linear probing and quadratic probing, the interval depends on the data, so that values mapping to the same location have different bucket sequences; this minimizes repeated collisions and the effects of clustering.

Drawbacks of Open Addressing

A drawback of all these open addressing schemes is that the number of stored entries cannot exceed the number of slots in the bucket array. In fact, even with good hash functions, their performance dramatically degrades when the load factor grows beyond 0.7 or so. For many applications, these restrictions mandate the use of dynamic resizing, with its attendant costs.

Open addressing schemes also put more stringent requirements on the hash function: besides distributing the keys more uniformly over the buckets, the function must also minimize the clustering of hash values that are consecutive in the probe order. Using separate chaining, the only concern is that too many objects map to the same hash value; whether they are adjacent or nearby.

Chaining

In the method known as separate chaining, each bucket is independent, and has some sort of list of entries with the same index. The time for hash table operations is the time to find the bucket (which is constant) plus the time for the list operation.

In most implementations buckets will have few entries, if the hash function is working properly. Therefore, structures that are efficient in time and space for these cases are preferred. Structures that are efficient for a fairly large number of entries per bucket are not needed or desirable. If these cases happen often, the hashing function needs to be fixed.

There are some implementations which give excellent performance for both time and space, with the average number of elements per bucket ranging between 5 and 100.

Implementation of Hash Table in C++

```
using namespace std;
#include<iostream>
#include<conio.h>
#include<string.h>
#include<iomanip>
struct HashData{
	int key;
	int value;
};
struct Node{
	HashData Data;
	Node* Next;
	Node* Previous;
};
void Graphics(const char str[])
{
	for(int i=1;i<=50;i++)
		cout<<"=";
	cout<<endl<<setw(25+strlen(str)/2)<<str<<endl;
	for(int i=1;i<=50;i++)
		cout<<"=";
	cout<<endl;
}
class LinkList{
	Node* Head;
	public:
		LinkList()
		{
			Head = NULL;
		}
		void Insert(HashData Data)
		{
			if(Head==NULL)
```

```
		{
			Head = new Node;
			Head->Data = Data;
			Head->Next = NULL;
			Head->Previous = NULL;
		}
		else
		{
			Node* Current = Head;
			while(Current->Next!=NULL)
				Current=Current->Next;
			Node* temp = new Node;
			temp->Data = Data;
			temp->Next = NULL;
			temp->Previous = Current;
			Current->Next = temp;
		}
		return;
	}
	bool Delete(HashData Data)
	{
		Node* Current = Head;
		while(Current!=NULL)
		{
			if(Current->Data.key == Data.key && Current->Data.value == Data.value)
			{
				if(Current == Head)
				{
					Head = Head->Next;
					delete Current;
					return true;
				}
				else
				{
					if(Current->Next!=NULL)
						Current->Next->Previous = Current->Previous;
					Current->Previous->Next = Current->Next;
					delete Current;
					return true;
				}
			}
			Current= Current->Next;
		}
		return false;
	}
	void Show()
	{
		Node* Current = Head;
		while(Current!=NULL)
		{
			cout<<Current->Data.value<<" ";
			Current=Current->Next;
		}
	}
	HashData getMin()
	{
		HashData Data = Head->Data;
		Node* Current = Head;
```

```
                while(Current!=NULL)
                {
                        if(Current->Data.value < Data.value)
                                Data = Current->Data;
                        Current=Current->Next;
                }
                return Data;
        }
        HashData getMax()
        {
                HashData Data = Head->Data;
                Node* Current = Head;
                while(Current!=NULL)
                {
                        if(Current->Data.value > Data.value)
                                Data = Current->Data;
                        Current=Current->Next;
                }
                return Data;
        }
        void Append(LinkList* List)
        {
                Node* Current = List->Head;
                while(Current!=NULL)
                {
                        this->Insert(Current->Data);
                        Current = Current->Next;
                }
                return;
        }
        void Sort()
        {
                LinkList* List = new LinkList();
                while(!this->isNULL())
                {
                        List->Insert(this->getMin());
                        this->Delete(this->getMin());
                }
                this->Head = List->Head;
                return;
        }
        bool isNULL()
        {
                return Head==NULL;
        }
        ~LinkList()
        {
                delete Head;
        }
};
class HashTable{
        int Size;
        LinkList* Table;
        public:
                HashTable(int Size=10)
                {
                        this->Size = Size;
                        Table = new LinkList[Size]();
```

```
}
void Insert(HashData Data)
{
	int Index = Data.key % Size;
	Table[Index].Insert(Data);
}
void Show()
{
	cout<<"HashTable"<<endl;
	for(int i=0; i<Size; i++)
	{
		cout<<"Index <"; (i<10?cout<<'0':cout<<""); (i<100?cout<<'0':cout<<""); cout<<i<<"> : ";
		Table[i].Show();
		cout<<endl;
	}
}
bool Delete(HashData Data)
{
	int Index = Data.key % Size;
	return Table[Index].Delete(Data);
}
HashData getMin()
{
	int i=0;
	while(Table[i].isNULL() && i<Size)
		i++;
	if(i>=Size)
	{
		HashData Data;
		Data.key = -1;
		return Data;
	}
	HashData HD = Table[i].getMin();
	while(i<Size)
	{
		if(!Table[i].isNULL())
			if(Table[i].getMin().value < HD.value)
				HD = Table[i].getMin();
		i++;
	}
	return HD;
}
HashData getMax()
{
	int i=0;
	while(Table[i].isNULL() && i<Size)
		i++;
	if(i>=Size)
	{
		HashData Data;
		Data.key = -1;
		return Data;
	}
	HashData HD = Table[i].getMax();
	while(i<Size)
	{
		if(!Table[i].isNULL())
			if(Table[i].getMax().value > HD.value)
```

```
                                                HD = Table[i].getMax();
                        i++;
                }
                return HD;
        }
        void getSortedItems()
        {
                if(this->isNULL())
                {
                        cout<<"Table is empty"<<endl;
                }
                else
                {
                        LinkList* List = new LinkList();
                        for(int i=0; i<Size; i++)
                                List->Append(&Table[i]);
                        cout<<"Unsorted List of Items : ";
                        List->Show();
                        cout<<endl;
                        List->Sort();
                        cout<<"Sorted   List of Items : ";
                        List->Show();
                        cout<<endl;
                        delete List;
                }
                return;
        }
        bool RemoveMin()
        {
                return this->Delete(this->getMin());
        }
        bool RemoveMax()
        {
                return this->Delete(this->getMax());
        }
        bool isNULL()
        {
                return (this->getMin().key == -1);
        }
        ~HashTable()
        {
                delete Table;
        }
};
int main()
{
        char ch;
        int Size;
        HashTable* HT;
        Graphics("HashTable Implementation");
        cout<<"Enter Size for Table : ";
        cin>>Size;
        HT = new HashTable(Size);
        cout<<"Table created sucessfully, Press any key to continue...";
        getch();

        HT->Insert(HashData{5, 241});
        HT->Insert(HashData{45, 1451});
```

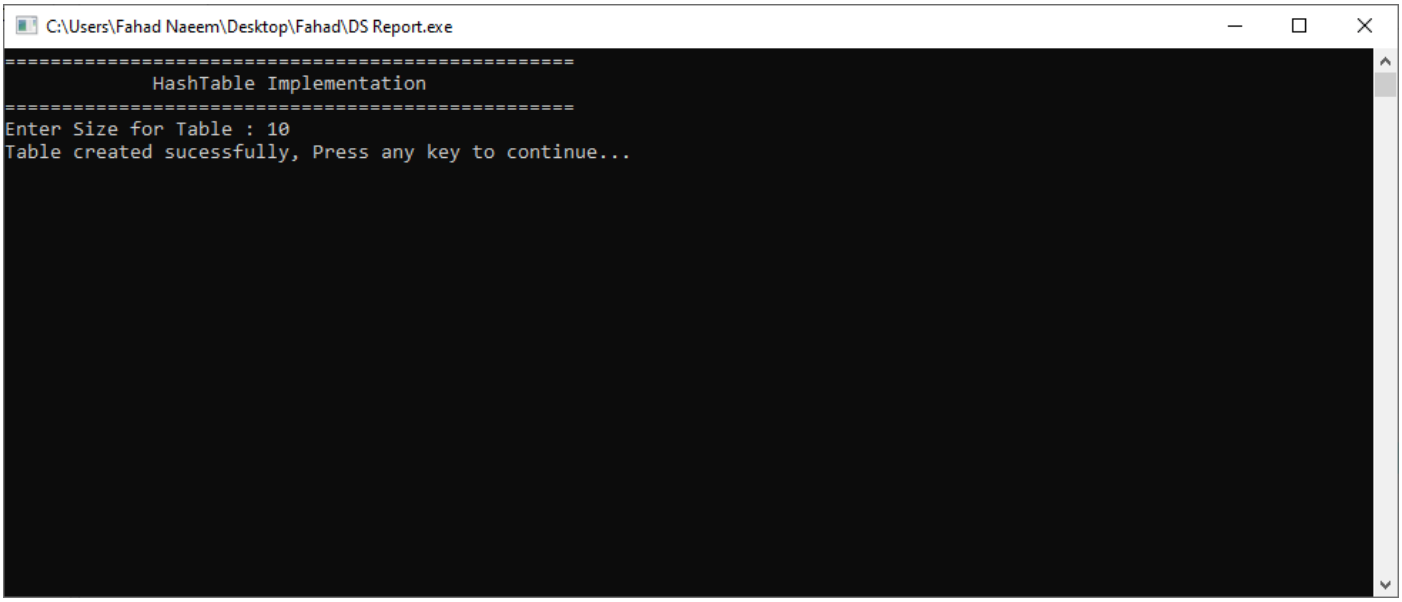
```
HT->Insert(HashData{21, 241});
HT->Insert(HashData{36, 5125});
HT->Insert(HashData{82, 9150});
HT->Insert(HashData{214, 142});
HT->Insert(HashData{27, 628});
HT->Insert(HashData{362, 1524});
HT->Insert(HashData{91, 398});
HT->Insert(HashData{74, 754});

while(true)
{
	system("cls");
	Graphics("HashTable Implementation");
	cout<<"1. Insert Data"<<endl;
	cout<<"2. Get Minimum Value"<<endl;
	cout<<"3. Get Maximum Value"<<endl;
	cout<<"4. Remove Minimum Value"<<endl;
	cout<<"5. Remove Maxium Value"<<endl;
	cout<<"6. Show Complete Table"<<endl;
	cout<<"7. Show Sorted List of Items"<<endl;
	cout<<"8. Exit : ";
	ch = getche();
	cout<<endl<<endl;
	if(ch=='1')
	{
		HashData Data;
		cout<<"Enter Key : ";
		cin>>Data.key;
		cout<<"Enter Value : ";
		cin>>Data.value;
		HT->Insert(Data);
		cout<<"Insered Sucessfully, Press any key to continue...";
		getch();
	}
	else if(ch=='2')
	{
		if(HT->isNULL())
			cout<<"Table is Empty"<<endl;
		else
		{
			HashData Data = HT->getMin();
			cout<<"Minimum Element"<<endl;
			cout<<"Key   = "<<Data.key<<endl;
			cout<<"Value = "<<Data.value<<endl;
		}
		cout<<"Press any key to continue...";
		getch();
	}
	else if(ch=='3')
	{
		if(HT->isNULL())
			cout<<"Table is Empty"<<endl;
		else
		{
			HashData Data = HT->getMax();
			cout<<"Maximum Element"<<endl;
			cout<<"Key   = "<<Data.key<<endl;
			cout<<"Value = "<<Data.value<<endl;
```

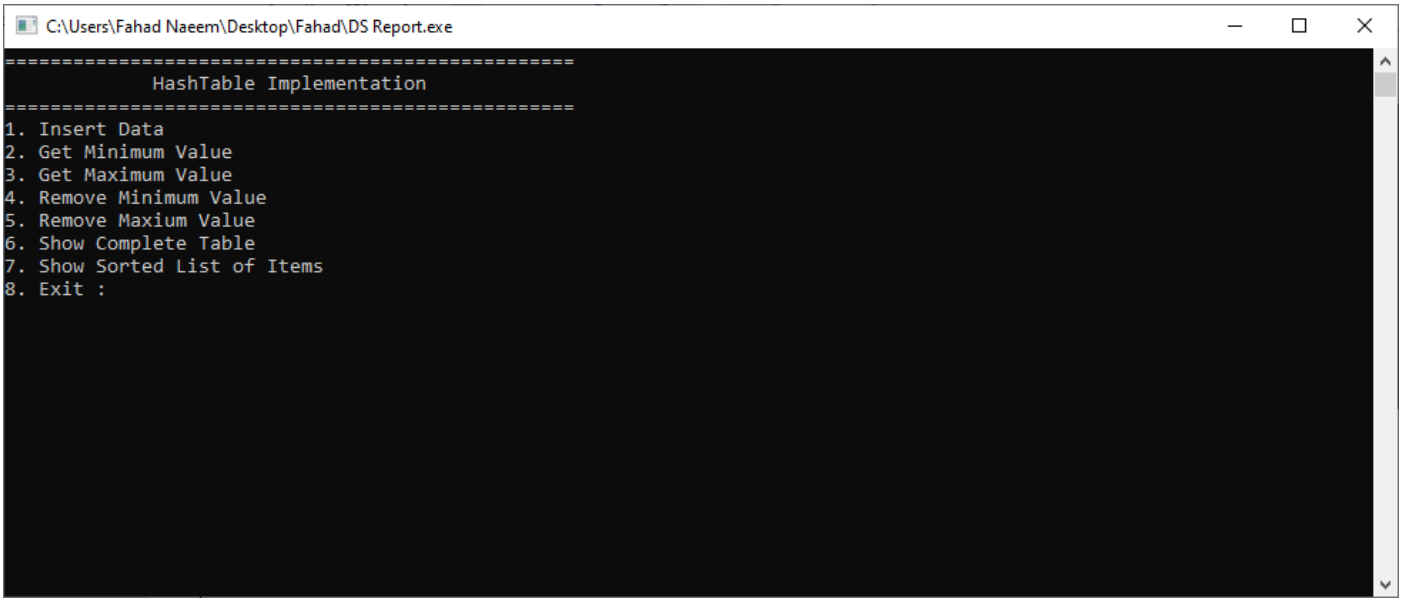
```
                        }
                        cout<<"Press any key to continue...";
                        getch();
                }
                else if(ch=='4')
                {
                        if(HT->isNULL())
                                cout<<"Table is Empty"<<endl;
                        else
                        {
                                HashData Data = HT->getMin();
                                HT->RemoveMin();
                                cout<<"Minimum Item Removed Sucessfully with the following details"<<endl;
                                cout<<"Key   = "<<Data.key<<endl;
                                cout<<"Value = "<<Data.value<<endl;
                        }
                        cout<<"Press any key to continue...";
                        getch();
                }
                else if(ch=='5')
                {
                        if(HT->isNULL())
                                cout<<"Table is Empty"<<endl;
                        else
                        {
                                HashData Data = HT->getMax();
                                HT->RemoveMax();
                                cout<<"Maximum Item Removed Sucessfully with the following details"<<endl;
                                cout<<"Key   = "<<Data.key<<endl;
                                cout<<"Value = "<<Data.value<<endl;
                        }
                        cout<<"Press any key to continue...";
                        getch();
                }
                else if(ch=='6')
                {
                        HT->Show();
                        cout<<"Press any key to continue...";
                        getch();
                }
                else if(ch=='7')
                {
                        HT->getSortedItems();
                        cout<<"Press any key to continue...";
                        getch();
                }
                else if(ch=='8')
                {
                        break;
                }
                else
                {
                        cout<<"In-Valid choice, Try again. Press any key to continue...";
                        getch();
                }
        }
        return 0;
}
```

Execution in Dev C++ IDE

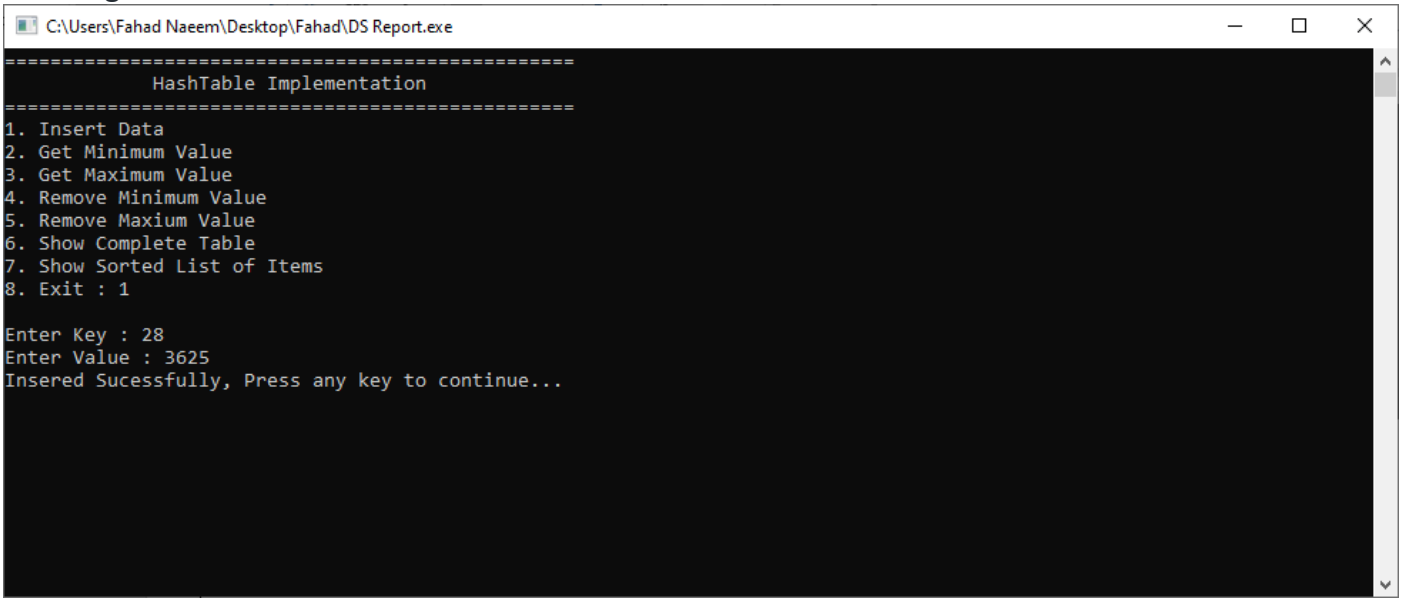
Starting Screen (Setting size of Table)



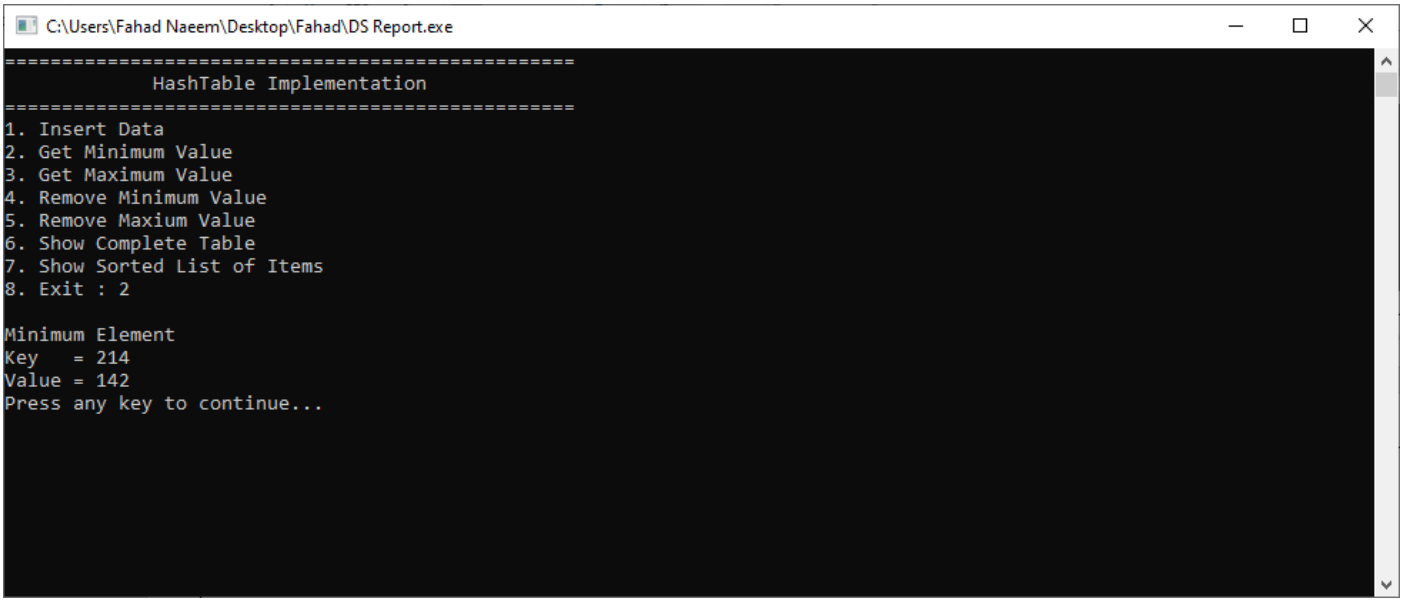
Main Menu



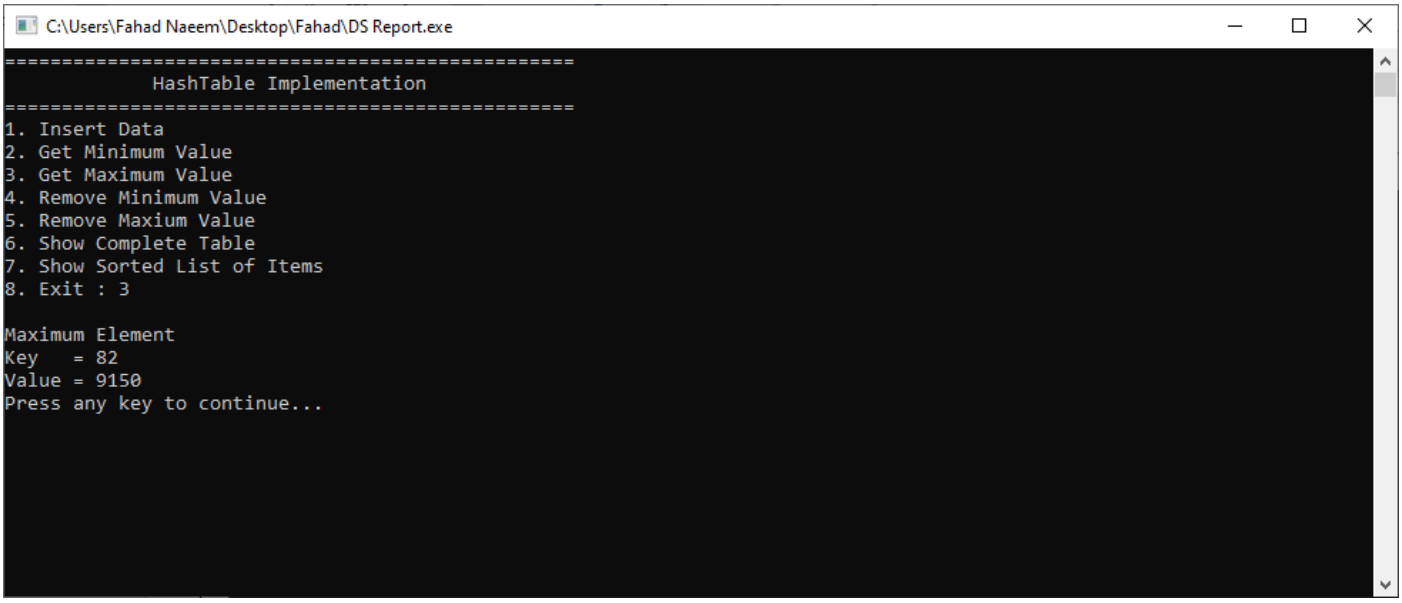
Inserting Data



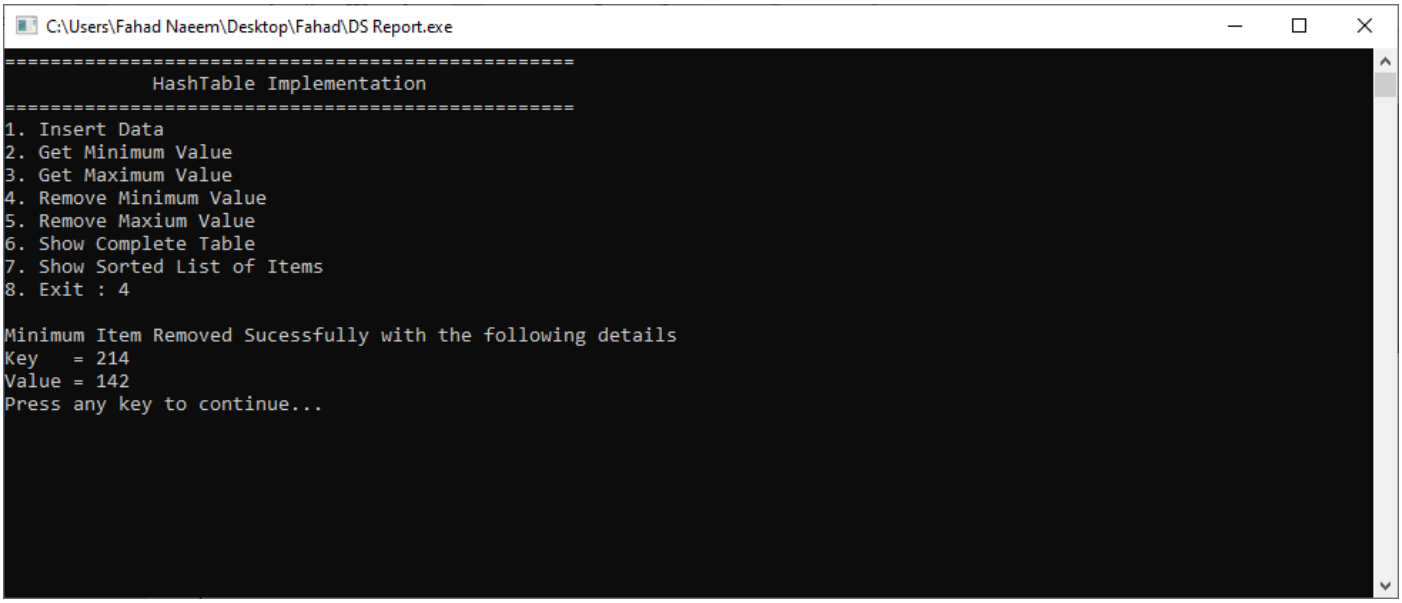
Get Minimum Value



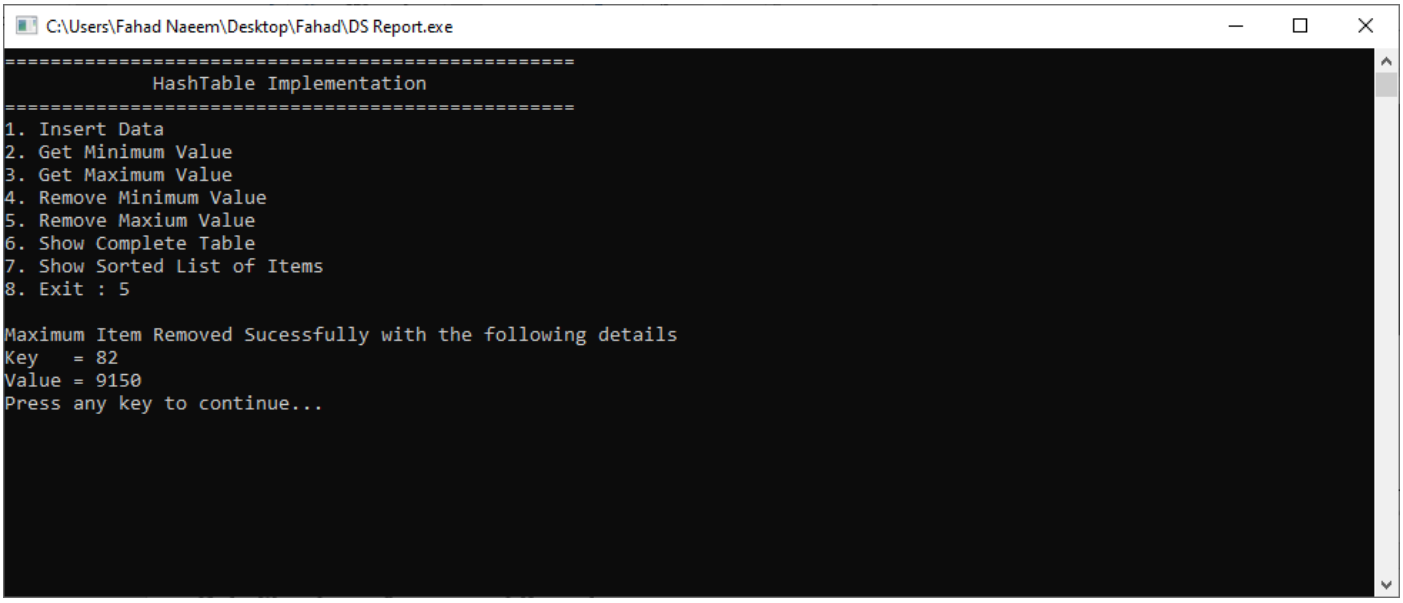
Get Maximum Value



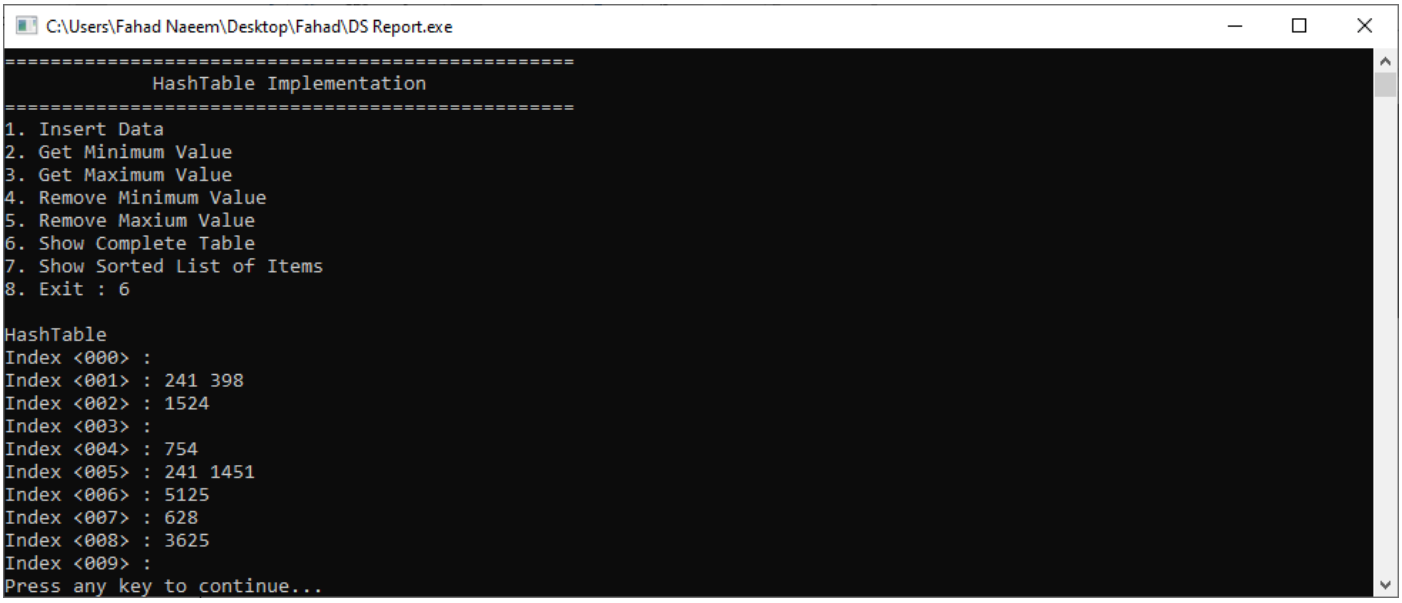
Remove Minimum Value



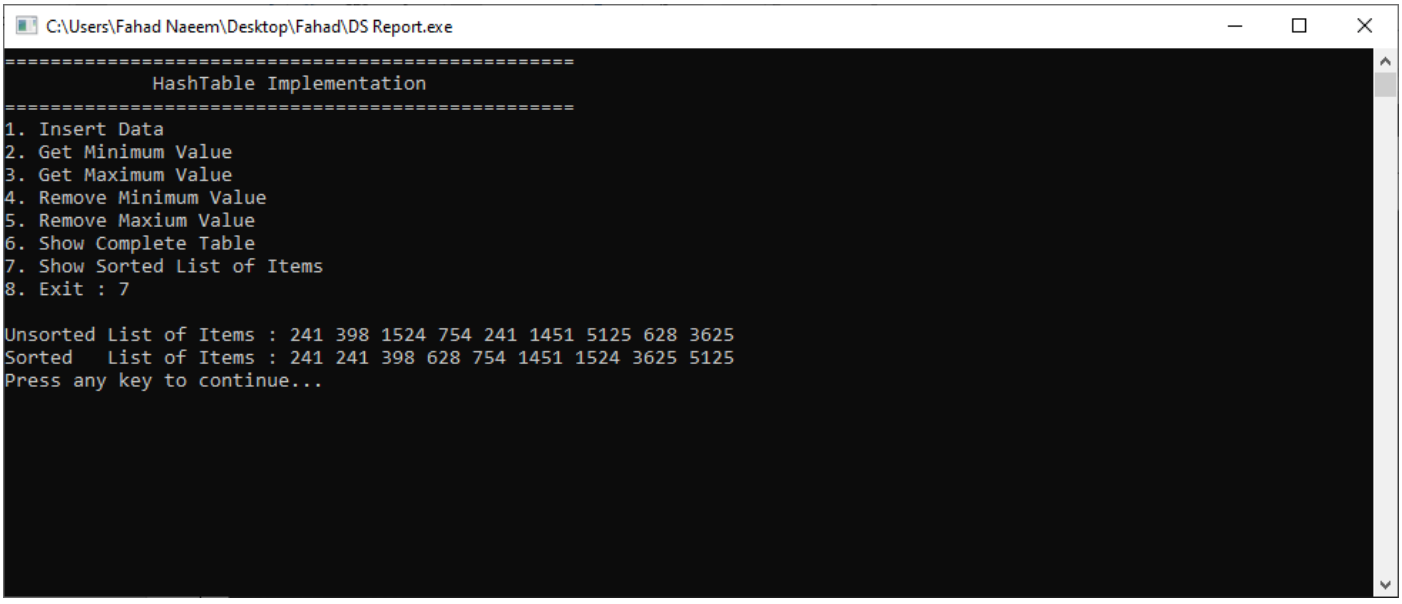
Remove Maximum Value



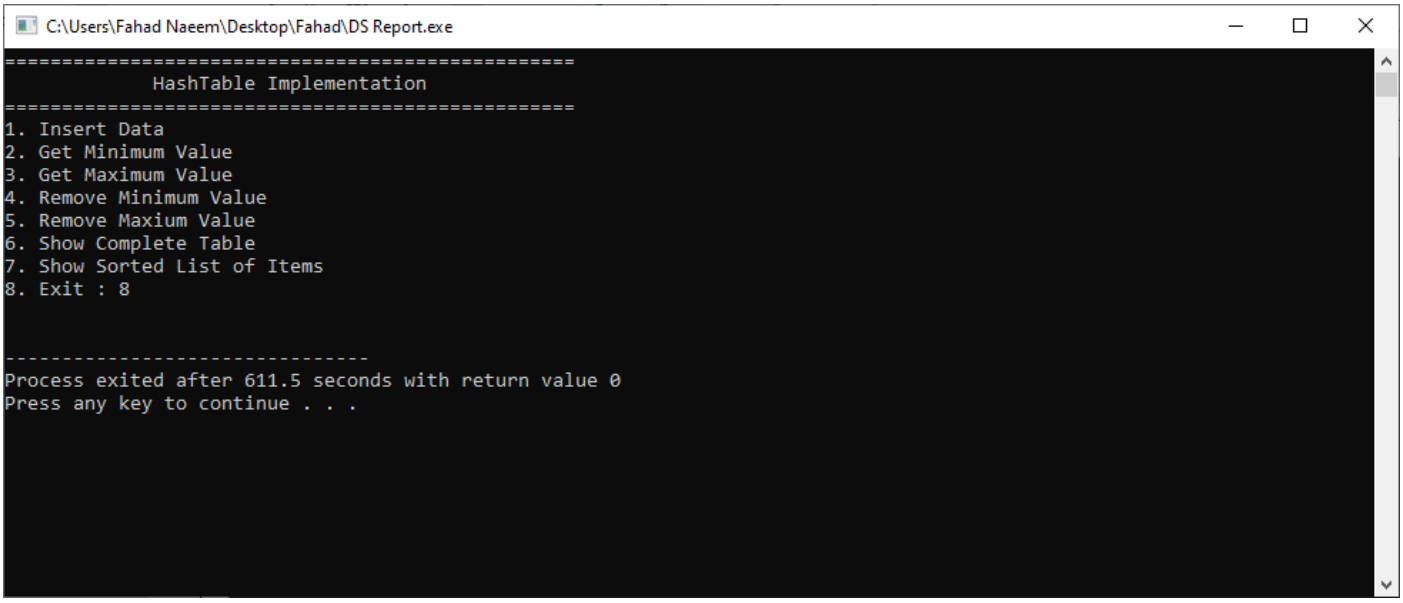
Show Table



Show Sorted List of Items



Exit



End of Report