Data Structure Lab

Assignment Number 05

Roll Number: **21118**

Problem Statement:

Implement all the functions of a dictionary (ADT) using hashing. Data: Set of (key, value) pairs, Keys are mapped to values, Keys must be comparable, Keys must be unique Standard Operations: Insert (key, value), Find(key), Delete(key)

Objectives:

1. To understand Dictionary (ADT)
2. To understand concept of hashing
3. To understand concept & features like searching using hash function.

Learning Objectives**:**

- To understand Dictionary (ADT)
- To understand concept of hashing
- To understand concept & features like searching using hash function.

Learning Outcome:

- Define class for Dictionary using Object Oriented features.
- Analyse working of hash function.

Theory:

- *Hash Table* is a data structure which stores data in an associative manner.
- In a hash table, data is stored in an array format, where each data value has its own unique index value. Access of data becomes very fast if we know the index of the desired data.
- Hash Table uses an array as a storage medium and uses hash technique to generate an index where an element is to be inserted or is to be located from.

Hashing

- Hashing is a technique to convert a range of key values into a range of indexes of an array.
- Generally, we use modulo operator to get a range of key values. Item are in the (key, value) format.

- Basic Operations of hash table Following are the basic primary operations of a hash table.
  - Search − Searches an element in a hash table.
  - Insert − inserts an element in a hash table.
  - delete − Deletes an element from a hash table.

1. DataItem

   Define a data item having some data and key, based on which the search is to be conducted in a hash table.

   ```
   struct DataItem {
   int data;
   int key;
   };
   ```
2. Hash Method

   Define a hashing method to compute the hash code of the key of the data item.

   ```
   int hashCode(int key){
    return key % SIZE;
    }
   ```

3. Search Operation

   Whenever an element is to be searched, compute the hash code of the key passed and locate the element using that hash code as index in the array. Use linear probing to get the element ahead if the element is not found at the computed hash code.

4. Insert Operation

   Whenever an element is to be inserted, compute the hash code of the key passed and locate the index using that hash code as an index in the array. Use linear probing for empty location, if an element is found at the computed hash code.

5. Delete Operation

   Whenever an element is to be deleted, compute the hash code of the key passed and locate the index using that hash code as an index in the array. Use linear probing to get the element ahead if an element is not found at the computed hash code. When found, store a dummy item there to keep the performance of the hash table intact.

Closed Hashing with chaining has 2 types:

- Chaining without replacement

- Chaining with replacement

i) Chaining without replacement:

1) In collision handling method chaining is a concept which introduces an additional field with data i.e. chain.
2) A separate chaining table is maintained for colliding data when collision occurs we store the collision data by linear probing method.
3) The address of this colliding data can be stored with the first colliding element in the chain table without replacement.

   Drawback: Finding the next empty location and losing the meaning of hash function.

ii) Chaining with replacement:

1) Drawback of losing the meaning of hash function in previous technique is overcome.
2) Drawback is each time logic is needed to test the element whether it is at its proper location.

<u>ADT:</u>

```
class Entry
{
private:
string word,meaning;                    // data member
int chain;
public:
Entry();                                // default constructor
Entry(string, string);                  // parameterized constructor
~Entry();                               // destructor
Friend class HashTable;
};

class HashTable
{
```

```
private:
Entry ht[SIZE];                 // data member
Int count;
public:
HashTable();                    // default constructor
Int hashfun(string);
void insertwithrep(string,string);
void insertwithoutrep(string,string);
void search(string);
void delete(string);
void display();
}
```

Pseudocode:

```
Algorithm hashFun(key)
{
    res := 0;
    for i in range(0, len(key)) do
       res += ord(key[i])*(i+1);
    return res % SIZE;
}
Algorithm insertWithoutReplacement(key, val):
{
    if self.count == SIZE then
       print("\n\t Dictionary is Full");
       return ;
    idx := self.hashFun(key)
    if self.ht[idx].word == " " then
       self.ht[idx] := Entry(key, val);
```

```
            print("\n\t Inserted Successfully");
            self.count += 1;
        else then
            temp := idx
            while self.hashFun(self.ht[idx].word) != temp and self.ht[idx].word != " " do
                idx := (idx+1) % SIZE;
            if self.ht[idx].word == " " then
                self.ht[idx] := Entry(key, val);
                print("\n\t Inserted Successfully");
                self.count += 1;
            else then
                while self.ht[idx].chain != -1 do
                    if self.ht[idx].word == key then$
                        print("\n\tWord Already Exist in Dictionary");
                        return ;
                    idx := self.ht[idx].chain;
                if self.ht[idx].word == key then
                    print("\n\tWord Already Exist in Dictionary");
                    return ;
                prevIdx := idx;
                while self.ht[idx].word != " " do
                    idx = (idx+1) % SIZE;
                self.ht[prevIdx].chain := idx;
                self.ht[idx] := Entry(key, val);
                print("\n\t Word inserted successfully");
                self.count += 1;
}
Algorithm insertWithRep(key, val)
{
```

```
if self.count == SIZE then
   print("\n\t Dictionary is Full");
   return ;
idx := self.hashFun(key);
if self.ht[idx].word == " " then
   self.ht[idx] := Entry(key, val);
   print("\n\t Inserted Successfully");
   self.count += 1;
else then
   if(self.hashFun(self.ht[idx].word) == idx) then
      while self.ht[idx].chain != -1 do
         if self.ht[idx].word == key then
            print("\n\tWord Already Exist in Dictionary");
            return ;
         idx := self.ht[idx].chain;
      if self.ht[idx].word == key then
         print("\n\tWord Already Exist in Dictionary");
         return then;
      prevIdx := idx;
      while self.ht[idx].word != " " do
         idx := (idx+1) % SIZE;
      self.ht[prevIdx].chain := idx;
      self.ht[idx] := Entry(key, val);

      print("\n\t Word inserted successfully");
      self.count += 1;
   else then
      temp := idx;
      idx := (idx+1)%SIZE;
```

```
            while(self.ht[idx].chain != temp) do
                idx := (idx+1)%SIZE;
            prevIdx := idx;
            idx := temp;
            while(self.ht[idx].word != " ")do
                idx := (idx+1)%SIZE;
            self.ht[prevIdx].chain := idx;
            self.ht[idx] := Entry(self.ht[temp].word, self.ht[temp].meaning,self.ht[temp].chain);
            self.ht[temp] := Entry(key,val);
            print("\n\t Word inserted successfully");
            self.count += 1;
}
Algorithm search(key)
{
    idx := self.hashFun(key);
    comparisons := 0;
    if self.hashFun(self.ht[idx].word) == idx then
        while self.ht[idx].chain != -1 do
            comparisons += 1;
            if self.ht[idx].word == key then
                print("\n\t Word Found After ", comparisons-1, "collisions");
                print("\n\t Word: ", self.ht[idx].word, "\t Meaning: ", self.ht[idx].meaning);
                return ;
            idx := self.ht[idx].chain;
        if self.ht[idx].word == key then
            comparisons += 1;
            print("\n\t Word Found After ", comparisons-1, "collisions");
            print("\n\t Word: ", self.ht[idx].word, "\t Meaning: ", self.ht[idx].meaning);
        else then
```

```
            print("\n\t Word does not exist in dictionary");
        else then
          temp := idx;
          while self.hashFun(self.ht[idx].word) != temp do
            comparisons += 1;
            idx := (idx+1) % SIZE;
            if idx == temp then
              print("\n\t Word does not exist in dictionary");
              return ;
          while self.ht[idx].chain != -1 do
            comparisons += 1;
            if self.ht[idx].word == key then
              print("\n\t Word Found After ", comparisons-1, "collisions");
              print("\n\t Word: ", self.ht[idx].word, "\t Meaning: ", self.ht[idx].meaning);
              return ;
            idx := self.ht[idx].chain;
          if self.ht[idx].word == key then
            comparisons += 1;
            print("\n\t Word Found After ", comparisons-1, "collisions");
            print("\n\t Word: ", self.ht[idx].word, "\t Meaning: ", self.ht[idx].meaning);
          else then
            print("\n\t Word does not exist in dictionary");
}
Algorithm searchIndex(key)
{
        idx := self.hashFun(key);
        if self.hashFun(self.ht[idx].word) == idx then
          while self.ht[idx].chain != -1 do
            if self.ht[idx].word == key then
```

```
            return idx;
         idx := self.ht[idx].chain;
      if self.ht[idx].word == key then
         return idx’
      else then
         return -1;
   else then
      temp := idx;
      while self.hashFun(self.ht[idx].word) != temp  do
         idx := (idx+1) % SIZE;
         if idx == temp then
            return -1;
      while self.ht[idx].chain != -1 do
         if self.ht[idx].word == key then
            return idx;
         idx := self.ht[idx].chain;
      if self.ht[idx].word == key then
         return idx;
      else then
         return -1;
}
Algorithm delete(key)
{
   if(self.searchIndex(key) == -1) then
      print("\n\t Word does not exist in dictionary")’
      return ;
   self.count -= 1;
   i := self.hashFun(key);
   if(key == self.ht[i].word) then
```

```
    if(self.ht[i].chain != -1) then
        temp := self.ht[i].chain;
        self.ht[i] := Entry(self.ht[temp].word, self.ht[temp].meaning, self.ht[temp].chain );
        self.ht[temp] := Entry();
    else:
        self.ht[i] := Entry();
        temp := i
        i = (i+1)%SIZE;
        while(i != temp) do
            if(self.hashFun(self.ht[i].word) == self.hashFun(key)) then
                while(self.ht[i].chain != temp) do
                    i := self.ht[i].chain;
                self.ht[i].chain = -1;
                print(f"\n\t {key} word deleted successfully");
                return ;
            i := (i+1)%SIZE;
else then
    i := self.searchIndex(key);
    if(self.ht[i].chain != -1) then
        temp := self.ht[i].chain;
        self.ht[i] := Entry(self.ht[temp].word, self.ht[temp].meaning, self.ht[temp].chain );
        self.ht[temp] := Entry();
    else then
        self.ht[i] := Entry();
        temp = I;
        i := (i+1)%SIZE;
        while(i != temp) do
            if(self.hashFun(self.ht[i].word) == self.hashFun(key)) then
                while(self.ht[i].chain != temp) do
```

```
                    i := self.ht[i].chain;
                  self.ht[i].chain = -1;
                  print(f"\n\t {key} word deleted successfully");
                  return ;
                i := (i+1)%SIZE;
        print(f"\n\t {key} word deleted successfully");
}
```

Test Cases:

1.] Test case no.1:

Input	Output	Expected Output	Result
Insert without replacement: Mango: Yellow, Orange: Orange, Banana: Yellow, Strawberry: Red Apple: Red	Inserted successfully	Inserted successfully	
Search (Apple)	Collision=1 Apple: Red	Collision=1 Apple: Red	
Search (Mango)	Collision=0	Collision=0	
	Mango: Yellow	Mango: Yellow	
Delete (Apple)	Deleted successfully	Deleted successfully	Pass

2.] Test case no. 2:

Input	Output	Expected Output	Result
Insert with replacement: Mango: Yellow, Orange: Orange, Banana: Yellow, Strawberry: Red Apple: Red	Inserted successfully	Inserted successfully	
Search (Apple)	Collision=1 Apple: Red	Collision=1 Apple: Red	
Search (Mango)	Collision=0	Collision=0	
	Mango: Yellow	Mango: Yellow	

Delete (Apple)	Deleted successfully	Deleted successfully	Pass

Real-time application:

- Password verification
- Pattern matching
- Compilers
- Message digest

Complexity of functions:

- Insert without replacement:
  Average: O (1)
  Worst: O (n)
- Insert with replacement:
  Average: O (1)
  Worst: O(n)
- Search:
  Average: O (1)
  Worst: O(n)
- Delete:
  O (n)
- Display:
  O (n)

Conclusion:

This program gives us the knowledge of dictionary (ADT). After completion of this assignment I get familiar with OOP features and also implemented standard dictionary features using OOP.