Unit – II

Dictionary Data Structure:

A *dictionary* is a general-purpose data structure for storing a group of objects. A dictionary has a set of *keys* and each key has a single associated *value*. When presented with a key, the dictionary will return the associated value.

The concept of a *key-value store* is widely used in various computing systems, such as caches and high-performance databases.

Typically, the keys in a dictionary must be simple types (such as integers or strings) while the values can be of any type. Different languages enforce different type restrictions on keys and values in a dictionary. Dictionaries are often implemented as Hashtable

Keys in a dictionary must be unique; an attempt to create a duplicate key will typically overwrite the existing value for that key.

Note that there is a difference (which may be important) between a key not existing in a dictionary, and the key existing but with its corresponding value being null.

Dictionary data structure supports following operations:

- *insert(key , value)* : add the key/value pair in the dictionary
- *remove(key)* : it removes key/value pair from the dictionary
- *search(key)* : it search for the key in dictionary and it return associated *value* of the if it exits otherwise it return *null*

Example:

contact book

key: name of person; value: telephone number

table of program variable identifiers

key: identifier; value: address in memory

property-value collection

key: property name; value: associated value

natural language dictionary

key: word in language X; value: word in language Y

/* Dictionary Implementation */

```
#include<stdio.h>
#include<stdlib.h>
#include<string.h>

struct Dic{
    int key;
```

```
        char value[20];
        struct Dic *link;
};
typedef struct Dic DicNode;
DicNode *start;

void insert(int,char*);
void removekey(int);
void display();
char* search(int);

int main(){
        char ch;
        int key;
        char value[20];
        start=NULL;
        while(1){
                printf("\nMENU\n1.insert\n2.search\n3.delete\n4.display\n5.exit\n");
                printf("Enter your choice:");
                scanf("%d",&ch);
                switch(ch){
                        case 1:
                                printf("Enter Key and value:");
                                scanf("%d%s",&key,value);
                                insert(key,value);
                                break;
                        case 2:
                                printf("Enter Key for Search:");
                                scanf("%d",&key);
                                printf("Value : %s",search(key));
                                break;
                        case 3:
                                printf("Enter key to delete:");
                                scanf("%d",&key);
                                removekey(key);
                                break;
                        case 4:
                                display();
                                break;
                        case 5:
                                exit(0);
                }
        }
        return 0;
}
```

```
void insert(int key, char *value){
	DicNode *c;
	int flag=0;
	c=start;
	if(start==NULL){
		start=(DicNode*)malloc(sizeof(DicNode));
		start->key=key;
		strcpy(start->value,value);
		start->link=NULL;
	}
	else{
		while(c->link != NULL){
			if(c->key == key){
				strcpy(c->value,value);
				 flag=1;
				 break;
			}
			c=c->link;
		}
		if(flag==0){
			c->link=(DicNode*)malloc(sizeof(DicNode));
			c->link->key=key;
			strcpy(c->link->value,value);
			c->link->link=NULL;
		}
	}
}

void removekey(int key){
	DicNode *c,*pre;
	c=start;
	while(c != NULL){
		if(c->key == key){
			if(c == start){
				start=start->link;
				free(c);
				break;
			}
			else{
				pre->link = c->link;
				free(c);
				break;
			}
		}
		pre=c;
		c=c->link;
```

```
        }
}
char* search(int key){
        DicNode *c;
        int flag=0;
        c=start;

        while(c != NULL){
                if(c->key == key){
                        return c->value;
                        break;
                }
                c=c->link;
        }
        return NULL;
}

void display(){
        DicNode *c;
        c=start;
        while(c!=NULL){
                printf("{%d : %s},",c->key,c->value);
                c=c->link;
        }
}
```

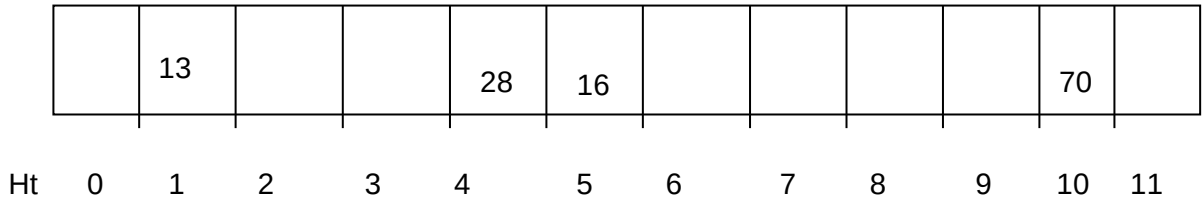
Hash Table Representation:

In hashing, the data is organized with the help of a table, called the hash table, denoted by HT, and the hash table is stored in an array. To determine whether a particular item with a key, say X, is in the table, we apply a function h, called the hash function, to the key X; that is, we compute h(X), read as h of X. The function h is typically an arithmetic function and h(X) gives the address of the item in the hash table. Suppose that the size of the hash table, HT, is N. Then $0 < h(X) < N$. Thus, to determine whether the item with key X is in the table, we look at the entry HT[h(X)] in the hash table. Because the address of an item is computed with the help of a function, it follows that the items are stored in no particular order. Before continuing with this discussion, let us consider the following questions:

- How do we choose a hash function?
- How do we organize the data with the help of the hash table?

First, we discuss how to organize the data in the hash table.
There are two ways that data is organized with the help of the hash table. In the first approach, the data is stored within the hash table, that is, in an array. In the second approach, the data is stored in linked lists and the hash table is an array of pointers to those linked lists. Each approach has its own advantages and disadvantages, and we discuss both approaches in detail. However, first we introduce some more terminology that is used in

this section.
The hash table HT is, usually, divided into, say N buckets HT[0], HT[1], . . ., HT[b – 1]. Each bucket is capable of holding, say r items. Thus, it follows that br = N, where m is the size of HT. Generally, r =1 and so each bucket can hold one item.
The hash function h maps the key X onto an integer t, that is, h(X) = t, such that 0 < h(X) < b – 1



Inserting key 70

h(k)=k%N

h(k)=70%12

h(k)=10

Inserting key 28

h(k) = 28%12

h(k)=4

ht[4]=28

Inserting key 13

h(k)=13%12

h(k)=1

ht[1]=13

Inserting key 16

h(k)=16%12

h(k)=4

ht[4+1]=16

Two keys, X1 and X2, such that X1≠ X2, are called **synonyms** if h(X1) = h(X2). Let X be a key and h(X) = t. If bucket t is full, we say that an **overflow** occurs. Let X1 and X2 be two nonidentical keys. If h(X1) = h(X2), we say that a **collision** occurs. If r = 1, that is, the bucket size is 1, an overflow and a collision occur at the same time.

When choosing a hash function, the main objectives are to:

• Choose a hash function that is easy to compute.
• Minimize the number of collisions.

Next, we consider some examples of hash functions.

Suppose that HTSize denotes the size of the hash table, that is, the size of the array holding the hash table. We assume that the bucket size is 1. Thus, each bucket can hold one item and, therefore, overflow and collision occur simultaneously.

Hash Functions: Some Examples

Several hash functions are described in the literature. Here we describe some of the commonly used hash functions.

Mid-Square: In this method, the hash function, h, is computed by squaring the identifier, and then using the appropriate number of bits from the middle of the square to obtain the bucket address. Because the middle bits of a square usually depend on all the characters, it is expected that different keys will yield different hash addresses with high

probability, even if some of the characters are the same.

Folding: In folding, the key X is partitioned into parts such that all the parts, except possibly the last parts, are of equal length. The parts are then added, in some convenient way, to obtain the hash address.
EX: Suppose the key X = 12320324111220, and wer partion it into parts that are three decimal digits long. The partitions are p1= 123, p2=203, p3=241,p4=112 and p5=20. Using shift folding, we obtain

$H(X) = \sum_{i=1}^{5} pi = 123+203+241+112+20 = 699$

When folding at boundaries is used. We firest reverse =2 and p4 to obtain 302adn 211, respectively. Next the five partitions are added to obtain h(X) =123 +302+241+211+20 = 897

Division (Modular arithmetic): In this method, the key X is converted into an integer X. This integer is then divided by the size of the hash table to get the remainder, giving the address of X in HT.
h(X) = X % D where D is the size of the HT

Converting Keys to Integer:

To use some of the described hash functions. Keys need to first be converted to nonnegative integer. Since all hash functions hash several keys into the same home bucket, it is not necessary for us to convert keys into unique nonnegative integers.

```
unsigned int stringToint(char *key){
        int number = 0;
        while(*key)
                number += *key++;
        return number;
}
```

Overflow Handling

Open addressing:

There are two popular ways to handle overflows: *open addressing* and *chaining*.
Open Addressing are 4 type. Liner probing , which also is known as liner open addressing, quadratic probing , rehashing and random probing

LINEAR PROBING

Suppose that an item with key X is to be inserted in HT. We use the hash function to compute the index h(X) of this item in HT. Suppose that h(X) ¼ t. Then 0 _ h(X) _ HTSize – 1. If HT[t] is empty, we store this item into this array slot. Suppose that HT[t] is already occupied by another item; we have a collision. In linear probing, starting at location t, we search the array sequentially to find the next available array slot.
In linear probing, we assume that the array is circular so that if the lower portion of the array is full, we can continue the search in the top portion of the array. This can be easily

accomplished by using the mod operator. That is, starting at t, we check the array locations t, (t + 1) % HTSize, (t + 2) % HTSize, . . ., (t + j) % HTSize. This is called the probe sequence.

The next array slot is given by (h(X) + j) % HTSize, where j is the j^{th} probe.

From the definition of linear probing, we see that linear probing is easy to implement. However, linear probing causes clustering; that is, more and more new keys would likely be hashed to the array slots that are already occupied. For example, consider the hash table of size 20, as shown in Figure a.

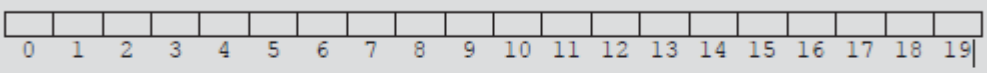


FIGURE a Hash table of size 20

Initially, all the array positions are available. Because all the array positions are available, the probability of any position being probed is (1/20). Suppose that after storing some of the items, the hash table is as shown in Figure b.

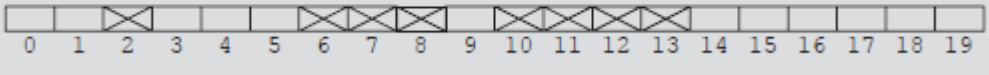


FIGURE b Hash table of size 20 with certain positions occupied

In Figure b, a cross indicates that this array slot is occupied. Slot 9 will be occupied next if, for the next key, the hash address is 6, 7, 8, or 9. Thus, the probability that slot 9 will be occupied next is 4/20. Similarly, in this hash table, the probability that array position 14 will be occupied next is 5/20.

Now consider the hash table of Figure c.

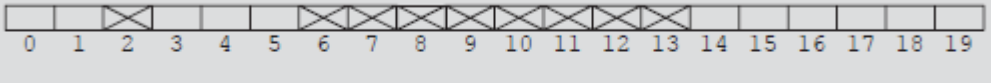


FIGURE c Hash table of size 20 with certain positions occupied

In this hash table, the probability that the array position 14 will be occupied next is 9/20, whereas the probability that the array positions 15, 16, or 17 will be occupied next is 1/20. We see that items tend to cluster, which would increase the search length. Linear probing, therefore, causes clustering. This clustering is called **primary clustering.**

QUADRATIC PROBING

Suppose that an item with key X is hashed at t, that is, h(X) = t and $0 \leq t \leq$ HTSize – 1.

Further suppose that position t is already occupied. In quadratic probing, starting at position t, we linearly search the array at locations (t + 1)% HTSize, (t + 2^2) % HTSize = (t + 4) % HTSize, (t + 3^2) % HTSize = (t + 9) % HTSize, . . ., (t + i^2) % HTSize. That is, the probe sequence is: t, (t + 1) % HTSize (t + 2^2) % HTSize, (t + 3^2) % HTSize, . . ., (t + i^2) % HTSize.

Although quadratic probing reduces primary clustering, we do not know if it probes all the positions in the table. In fact, it does not probe all the positions in the table. However, when HTSize is a prime, quadratic probing probes about half the table before repeating the probe sequence. Let us prove this observation.

Suppose that HTSize is a prime and for $0 \leq i < j \leq$ HTSize,

(t + i^2) % HTSize = (t + j^2) % HTSize

This implies that HTSize divides ($j^2 - i^2$), that is, HTSize divides (j – i) (j + i). Because HTSize is a prime, we get HTSize divides (j – i) or HTSize divides (j + i).

Now because $0 < j - i <$ HTSize, it follows that HTSize does not divide (j - i). Hence, HTSize divides (j + i). This implies that $j + i \geq$ HTSize, so $j \geq$ (HTSize / 2).

Hence, quadratic probing probes half the table before repeating the probe sequence. Thus, it follows that if the size of HTSize is a prime at least twice the number of items, we can resolve all the collisions.

Because probing half the table is already a considerable number of probes, after making these many probes we assume that the table is full and stop the insertion (and search).

REHASHING

In this method, if a collision occurs with the hash function h, we use a series of hash functions, h1, h2, . . ., hs. That is, if the collision occurs at h(X), the array slots hi(X), $1 \leq$ hi(X) $\leq$ s are examined.

RANDOM PROBING

This method uses a random number generator to find the next available slot. The i^{th} slot in the probe sequence is
(h(X) + r_i) % HTSize

where r_i is the i^{th} value in a random permutation of the numbers 1 to HTSize – 1. All insertions and searches use the same sequence of random numbers.

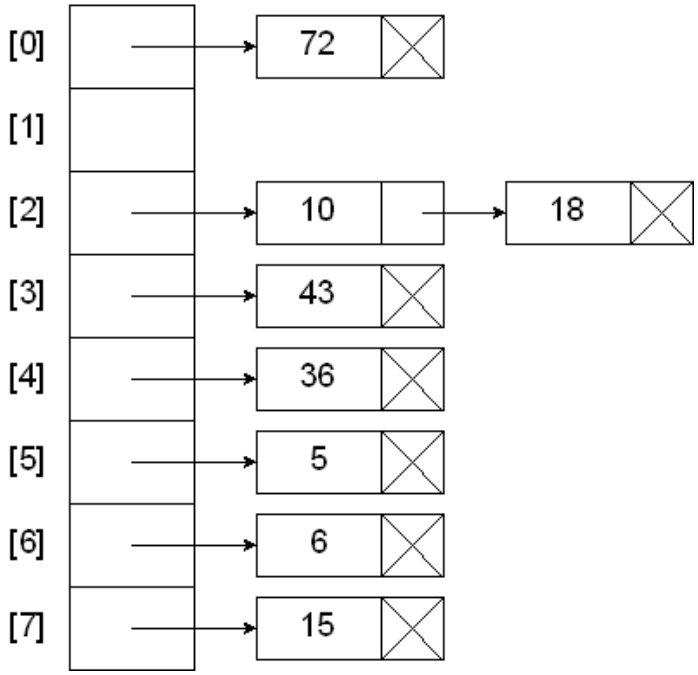
Chaining

Chaining is a possible way to resolve collisions. Each slot of the array contains a link to a singly-linked list containing key-value pairs with the same hash. New key-value pairs are added to the end of the list. Lookup algorithm searches through the list to find matching

key. Initially table slots contain nulls. List is being created, when value with the certain hash is added for the first time.

Hash key = key % table size

```
4     = 36 % 8
2     = 18 % 8
0     = 72 % 8
3     = 43 % 8
6     =  6 % 8
2     = 10 % 8
5     =  5 % 8
7     = 15 % 8
```



/* Hash Table */

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <stdbool.h>

#define SIZE 20

struct DataItem {
   int data;
   int key;
};

struct DataItem* hashArray[SIZE];
struct DataItem* dummyItem;
struct DataItem* item;

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

struct DataItem *search(int key) {
   //get the hash
   int hashIndex = hashCode(key);

   //move in array until an empty
```

```
   while(hashArray[hashIndex] != NULL) {

      if(hashArray[hashIndex]->key == key)
         return hashArray[hashIndex];

      //go to next cell
      ++hashIndex;

      //wrap around the table
      hashIndex %= SIZE;
   }

   return NULL;
}

void insert(int key,int data) {

   struct DataItem *item = (struct DataItem*) malloc(sizeof(struct DataItem));
   item->data = data;
   item->key = key;

   //get the hash
   int hashIndex = hashCode(key);

   //move in array until an empty or deleted cell
   while(hashArray[hashIndex] != NULL && hashArray[hashIndex]->key != -1) {
      //go to next cell
      ++hashIndex;

      //wrap around the table
      hashIndex %= SIZE;
   }

   hashArray[hashIndex] = item;
}

struct DataItem* delete(struct DataItem* item) {
   int key = item->key;

   //get the hash
   int hashIndex = hashCode(key);

   //move in array until an empty
   while(hashArray[hashIndex] != NULL) {

      if(hashArray[hashIndex]->key == key) {
```

```
      struct DataItem* temp = hashArray[hashIndex];

      //assign a dummy item at deleted position
      hashArray[hashIndex] = dummyItem;
      return temp;
    }

    //go to next cell
    ++hashIndex;

    //wrap around the table
    hashIndex %= SIZE;
  }

  return NULL;
}

void display() {
  int i = 0;

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

    if(hashArray[i] != NULL)
      printf(" (%d,%d)",hashArray[i]->key,hashArray[i]->data);
    else
      printf(" (#,#)");
  }

  printf("\n");
}

int main() {
  dummyItem = (struct DataItem*) malloc(sizeof(struct DataItem));
  dummyItem->data = -1;
  dummyItem->key = -1;

  insert(5, 100);
  insert(2, 80);
  insert(23, 90);
  insert(4, 65);
  insert(13, 85);
  insert(30, 96);
  insert(15,44);
  insert(25, 55);
  insert(42, 66);
```

```
   display();
   item = search(42);

   if(item != NULL) {
      printf("Element found: %d\n", item->data);
   } else {
      printf("Element not found\n");
   }

   delete(item);
   item = search(37);

   if(item != NULL) {
      printf("Element found: %d\n", item->data);
   } else {
      printf("Element not found\n");
   }
}
```