MODULE 5: HASHING

Hashing in Data Structure-

In data structures,

- Hashing is a well-known technique to search any particular element among several elements.
- It minimizes the number of comparisons while performing the search.

Advantage-

Unlike other searching techniques,

- Hashing is extremely efficient.
- The time taken by it to perform the search does not depend upon the total number of elements.
- It completes the search with constant time complexity O(1).

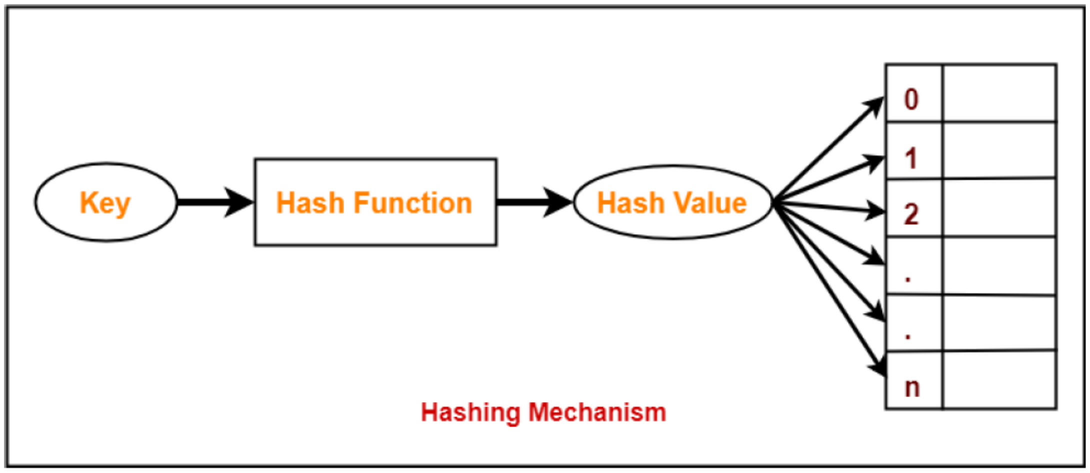
Hashing Mechanism-

In hashing,

- An array data structure called as **Hash table** is used to store the data items.
- Based on the hash key value, data items are inserted into the hash table.

Hash Key Value-

- Hash key value is a special value that serves as an index for a data item.
- It indicates where the data item should be be stored in the hash table.
- Hash key value is generated using a hash function.



Hash Function-

Hash function is a function that maps any big number or string to a small integer value.

- Hash function takes the data item as an input and returns a small integer value as an output.
- The small integer value is called as a hash value.
- Hash value of the data item is then used as an index for storing it into the hash table.

Types of Hash Functions-

There are various types of hash functions available such as-

1. Mid Square Hash Function
2. Division Hash Function
3. Folding Hash Function etc

It depends on the user which hash function he wants to use.

Properties of Hash Function-

The properties of a good hash function are-

- It is efficiently computable.
- It minimizes the number of collisions.
- It distributes the keys uniformly over the table.

To gain better understanding about Hashing in Data Structures,

- There are several searching techniques like linear search, binary search, search trees etc.
- In these techniques, time taken to search any particular element depends on the total number of elements.

Example-

- **Linear Search** takes O(n) time to perform the search in unsorted arrays consisting of n elements.
- **Binary Search** takes O(logn) time to perform the search in sorted arrays consisting of n elements.
- It takes O(logn) time to perform the search in **Binary Search Tree** consisting of n elements.

Drawback-

The main drawback of these techniques is-

- As the number of elements increases, time taken to perform the search also increases.
- This becomes problematic when total number of elements become too large.

Hashing using Arrays

When implementing a hash table using arrays, the nodes are not stored consecutively, instead the location of storage is computed using the key and a ***hash*** function. The computation of the array index can be visualized as shown below:

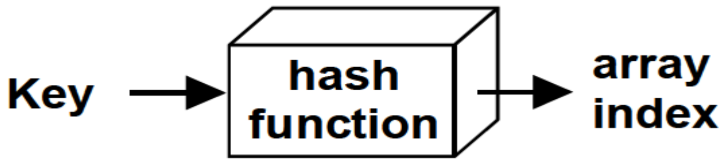


Figure 5. Array Index Computation

The value computed by applying the hash function to the key is often referred to as the hashed key. The entries into the array, are scattered (not necessarily sequential) as can be seen in figure below.

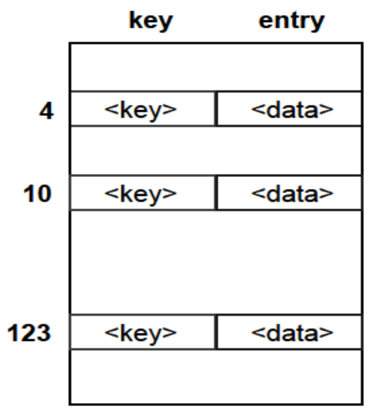


Figure 6. Hashed Array

The cost of the insert, find and delete operations is now only O(1). Can you think of why?

Hash tables are very good if you need to perform a lot of search operations on a relatively stable table (i.e. there are a lot fewer insertion and deletion operations than search operations).

One the other hand, if traversals (covering the entire table), insertions, deletions are a lot more frequent than simple search operations, then ordered binary trees (also called AVL trees) are the preferred implementation choice.

Hashing Performance

There are three factors the influence the performance of hashing:

- Hash function
  - should distribute the keys and entries evenly throughout the entire table
  - should minimize collisions

- Collision resolution strategy
  - Open Addressing: store the key/entry in a different position
  - Separate Chaining: chain several keys/entries in the same position

- Table size
  - Too large a table, will cause a wastage of memory
  - Too small a table will cause increased collisions and eventually force *rehashing* (creating a new hash table of larger size and copying the contents of the current hash table into it)
  - The size should be appropriate to the hash function used and should typically be a prime number. Why? (We discussed this in class).

Selecting Hash Functions

The hash function converts the key into the table position. It can be carried out using:

- Modular Arithmetic: Compute the index by dividing the key with some value and use the remainder as the index. This forms the basis of the next two techniques.

  For Example: index := key MOD table_size

- Truncation: Ignoring part of the key and using the rest as the array index. The problem with this approach is that there may not always be an even distribution throughout the table.

  For Example: If student id's are the key 928324312 then select just the last three digits as the index i.e. 312 as the index. => the table size has to be atleast 999. Why?

- Folding: Partition the key into several pieces and then combine it in some convenient way.

  For Example:
  - For an 8 bit integer, compute the index as follows:
    Index := (Key/10000 + Key MOD 10000) MOD Table_Size.

  - For character strings, compute the index as follows:

    ```
    Index :=0
    For I in 1.. length(string)
    Index := Index + ascii_value(String(I))
    ```

Collision

Let us consider the case when we have a single array with four records, each with two fields, one for the key and one to hold data (we call this a *single slot bucket*). Let the hashing function be a simple modulus operator i.e. array index is computed by finding the remainder of dividing the key by 4.

Array Index := key MOD 4

Then key values 9, 13, 17 will all hash to the same index. When two(or more) keys hash to the same value, a **collision** is said to occur.

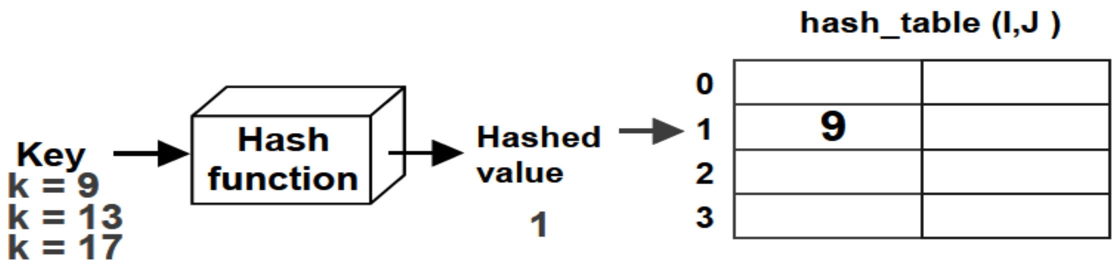


Figure 7. Collision Using a Modulus Hash Function

Collision Resolution

The hash table can be implemented either using

- Buckets: An array is used for implementing the hash table. The array has size m*p where m is the number of hash values and p (≥ 1) is the number of slots (a slot can hold one entry) as shown in figure below. The *bucket* is said to have p slots.

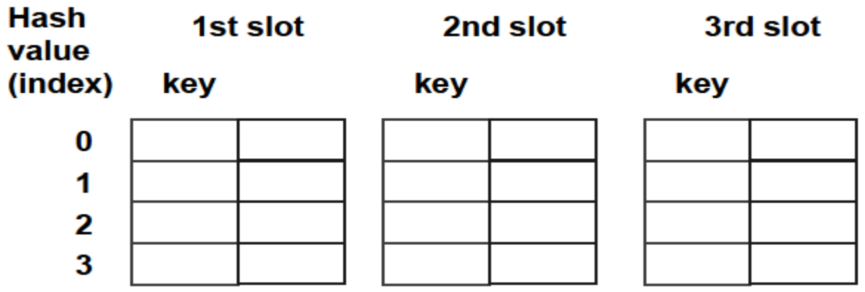


Figure 8. Hash Table with Buckets

- Chaining: An array is used to hold the key and a pointer to a liked list (either singly or doubly linked) or a tree. Here the number of nodes is not restricted (unlike with buckets). Each node in the chain is large enough to hold one entry as shown in figure below.

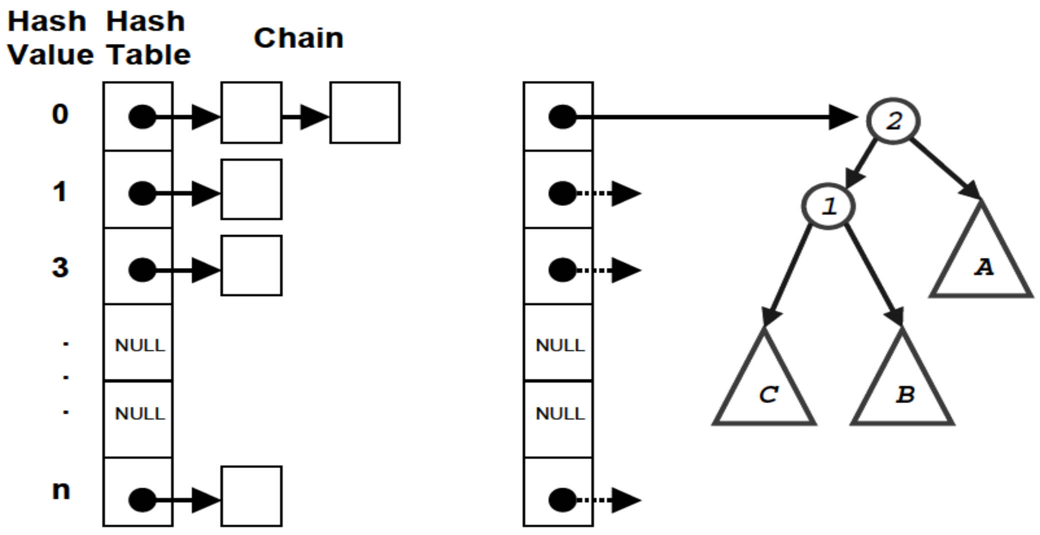


Figure 9. Chaining using Linked Lists / Trees

Open Addressing (Probing)

Open addressing / probing is carried out for insertion into fixed size hash tables (hash tables with 1 or more buckets). If the index given by the hash function is occupied, then increment the table position by some number.

There are three schemes commonly used for probing:

- Linear Probing: The linear probing algorithm is detailed below:

  ```
  Index := hash(key)
  While Table(Index) Is Full do
          index := (index + 1) MOD Table_Size
  if (index = hash(key))
          return table_full
  else
          Table(Index) := Entry
  ```

- Quadratic Probing: increment the position computed by the hash function in quadratic fashion i.e. increment by 1, 4, 9, 16, ….

- Double Hash: compute the index as a function of two different hash functions.

Chaining

In chaining, the entries are inserted as nodes in a linked list. The hash table itself is an array of head pointers.

The advantages of using chaining are

- Insertion can be carried out at the head of the list at the index
- The array size is not a limiting factor on the size of the table

The prime disadvantage is the memory overhead incurred if the table size is small.

Hashing is implemented in two steps:

1. An element is converted into an integer by using a hash function. This element can be used as an index to store the original element, which falls into the hash table.

2. The element is stored in the hash table where it can be quickly retrieved using hashed key.

There are many possibilities for representing the dictionary and one of the best methods for representing is hashing. Hashing is a type of a solution which can be used in almost all situations. Hashing is a technique which uses less key comparisons. This method generally used the hash functions to map the keys into a table, which is called a hash table.

Hashing is of two types.

a) **Static Hashing**
b) **Dynamic Hashing**

Static Hashing

1) Hash Table

Hash table is a type of data structure which is used for storing and accessing data very quickly. Insertion of data in a table is based on a key value. Hence every entry in the hash table is defined with some key. By using this key data can be searched in the hash table by few key comparisons and then searching time is dependent upon the size of the hash table.

2) Hash Function

Hash function is a function which is applied on a key by which it produces an integer, which can be used as an address of hash table. Hence one can use the same hash function for accessing the data from the hash table. In this the integer returned by the hash function is called hash key.

Types of Hash Functions

There are various types of hash function which are used to place the data ina hash table,

1) **Division Method**
2) **Mid square Method**
3) **Digit folding Method**
4) **Digit Analysis Method/Binary/Radix Method**

1)Division Method

In this method the hash function is dependent upon the remainder of a division.

For example if the record **52, 68, 99, 84** is to be placed in a hash table and let us take the table size is 10.

Then:

h(key)= key % table_size
h(52) = 52 % 10 = 2
h(68) = 68 % 10 = 8
h(99) = 99 % 10 = 9
h(84) = 84 % 10 = 4

Division Method

0	
1	
2	52
3	
4	84
5	
6	
7	
8	68
9	99

Hash Table

```
//---------------------------------------------------------------------------------------------------
//Program to insert elements into a hash table using Division method
//---------------------------------------------------------------------------------------------------
            #include <stdio.h>
            #include <conio.h>
            int a[10],size;

            int hashfunction(int e)
            {
                int key;
                key = e % size;
                return key;
            }

            void main()
            {
                int i, j, element;

                clrscr();
                printf("enter size of hash table ");
                scanf("%d",&size);

                for(i=0; i<size; i++)
                {
                    printf("enter element to insert ");
                    scanf("%d",&element);

                    j=hashfunction(element);
                    a[j]=element;
                 }

               printf("values in hash Table\n");
               for(i=0; i<size; i++)
                    printf("%d\n",a[i]);

                getch();
              }
```

2) Mid Square Method

In this method firstly key is squared and then mid part of the result is taken as the index. For example: consider that if we want to place a record of **3101** and the size of table is 1000.

So, **3101*3101=96$\underline{162}$01**

i.e. h (3101) = 162 (middle 3 digit).

k	k*k	Square	index h(k)
15	15 * 15	225	2
22	22 * 22	484	8
16	16 * 16	256	5
29	29 * 29	841	4
31	31 * 31	961	6
5	5 * 5	25	0
3	3 * 3	9	9

Mid Square Method

0	5
1	
2	15
3	
4	29
5	16
6	31
7	
8	22
9	3

Hash Table

If the square value contains even number of digits the index is 0.

If it contains odd value index is middle value.

If the square value is a single digit, the value will be placed in that index only. Value 3 is stored at location 9.

3) Digit Folding Method

In this method, we partition the identifier k into several parts. All parts, except for the last one have the same length. We then add the parts together to obtain the hash address for k. There are two ways of carrying out this addition. In the first method, we shift all parts except for the last one, so that the least significant bit of each part lines up with the corresponding bit of the last part. We then add the parts together to obtain h(k). This method is known as **shift folding.**

The second method, known as **folding at the boundaries**, the key is folded at the partition boundaries, and digits falling into the same position are added together to obtain h(k). This is equivalent to reversing every other partition before adding.

Example 1: Suppose that k=12320324111220, and we partition it into parts that are 3 decimal digits long. The partitions are P1=123, P2=203, P3=241, P4=112 and P5=20

Shift Folding

$$\mathrm{h}(k) = \sum_{i=1}^{5} P_i$$

= P1+P2+P3+P4+P5
= 123 + 203 + 241 + 112 + 20
= 699

Folding at the boundaries

When folding at boundaries is used, we first reverse P2 and P4 to obtain 302 and 211 respectively. Next the five partitions are added to obtain

$$\mathrm{h}(k) = \sum_{i=1}^{5} P_i$$

= P1+P2+P3+P4+P5
= 123 + 302 + 241 + 211 + 20
= 897

Example 2) For example: consider a record of 12465512 then it will be divided into parts.
i.e. 124, 655, 12. After dividing the parts combine these parts by adding it.

Shift folding
H(key) =124+655+12 = 791
791 is the index to store the value 12465512

Folding at the boundaries
H(key) =124+556+12 = 692
692 is the index to store the value 12465512

4) Digit Analysis Method:

In this method we will examine, digit analysis, is used with static files. A static file is one in which all the identifiers are known in advance.

Using this method, we first transform the identifiers into numbers using some radix, r.

We then examine the digits of each identifier, deleting those digits that have the most skewed distributions. We continue deleting digits until the number of remaining digits is small enough to give an address in the range of the hash table. The digits used to calculate the hash address must be the same for all identifiers and must not have abnormally high peaks or valleys (the standard deviation must be small).

For Example store values 4,8,3,7 in the hash table considering the radix 2.

Consider the hash table having a size of 8 elements. The index is in binary. The no of digits in index is 3. So we are going to generate 3 digited index from our has function.

Digit Analysis Method

k	value in binary	h(k)
4	0100	100
8	1000	000
3	0011	011
7	0111	111

000	8
001	
010	
011	3
100	4
101	
110	
111	7

Hash Table

Characteristics of Good Hashing Function

1) The hash function should generate different hash values for the similar string.

2) The hash function is easy to understand and simple to compute.

3) The hash function should produce the keys which will get distributed,uniformly over an array.

4) A number of collisions should be less while placing the data in the hash table.

5) The hash function is a perfect hash function when it uses all the inputdata.

Collision

It is a situation in which the hash function returns the same hash key for more than one record, it is called as collision. Sometimes when we are going to resolve the collision it may lead to an overflow condition and this overflow and collision condition makes the poor hash function.

Collision resolution technique

If there is a problem of collision occurs, then it can be handled by apply some technique. These techniques are called as collision resolution techniques. There are generally four techniques which are described below.

1) **Chaining**
2) **Linear Probing (Open addressing)**
3) **Quadratic Probing (Open addressing) and**
4) **Double Hashing (Open addressing).**

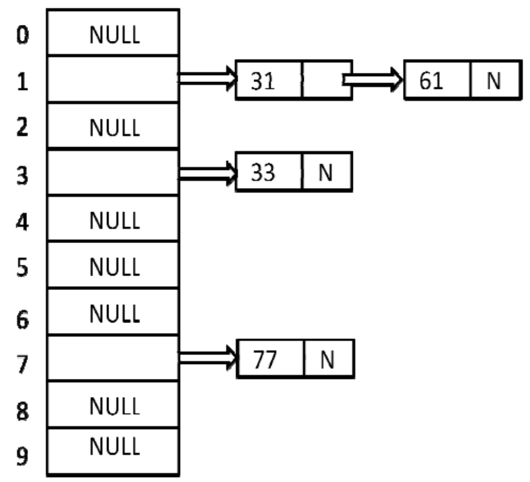
1) Chaining

It is a method in which additional field with data i.e. chain is introduced. A chain is maintained at the home bucket. In this when a collision occurs then a linked list is maintained for colliding data.

For Example: Let us consider a hash table of size 10 and we apply a hash function of H(key)=key % size of table. Let us take the keys to be inserted are **31, 33, 77, 61**.

In the diagram we can see at same bucket 1 there are two records which are maintained by linked list or we can say by chaining method.

H(31) = 31%10 = 1
H(33) = 33%10 = 3
H(77) = 77%10 = 7
H(61) = 61%10 = 1



2) Linear probing (Open addressing)

56 % 10 = 6
64 % 10 = 4

0	NULL
1	NULL
2	NULL
3	NULL
4	64
5	NULL
6	56
7	NULL
8	NULL
9	NULL

36 % 10 = 6

The index 6 is already filled with 56
It is not empty
Collision occurred
To resolve this check the next location
i.e. 6+1 = 7
index 7 is NULL so insert 36 at index 7.

0	NULL
1	NULL
2	NULL
3	NULL
4	64
5	NULL
6	56
7	36
8	NULL
9	NULL

71 % 10 = 1

As the index 1 is null
We can insert 71 at index 1

0	NULL
1	71
2	NULL
3	NULL
4	64
5	NULL
6	56
7	36
8	NULL
9	NULL

It is very easy and simple method to resolve or to handle the collision. In this, collision can be solved by placing the second record linearly down, whenever the empty place is found. In this method there is a problem of clustering which means at some place block of a data is formed in a hash table.

Example: Let us consider a hash table of size 10 and hash function is defined as H(key)=key % table_size. Consider that following keys are to be inserted that are **56, 64, 36, 71**.

In this diagram we can see that 56 and 36 need to be placed at same bucket but by linear probing technique the records linearly placed downward if place is empty i.e. it can be seen 36 is placed at index 7.

```
//-----------------------------------------------------------------------------------------------------
//Program to insert elements into a hash table using linear probing - Division method
//-----------------------------------------------------------------------------------------------------
            #include <stdio.h>
            #include <conio.h>
            int a[10], size;

            int hashfunction(int e)
            {
              int key;
              key = e % size;
              if (a[key]==0)
                return key;
              else
                if (size==key)
                  {
                    printf("hash table is FULL");
                    return -1;
                  }
                else
                  hashfunction(e+1);
            }

            void main()
            {
                int i,j,element;
                clrscr();
                printf("enter size of hash table ");
                scanf("%d",&size);

                    // ---------------------------------------------initialise hash table
```

```
 for(i=0; i<size; i++)
   a[i]=0;

  for(i=0; i<size; i++)
  {
       printf("enter element to insert ");
       scanf("%d",&element);
       j=hashfunction(element);
       a[j]=element;
   }

  printf("values in hash Table\n");
  for(i=0; i<size; i++)
     printf("%d\n",a[i]);

  getch();
}
```

3) Quadratic Probing (Open addressing)

This is a method in which solving of clustering problem is done. In this method the hash function is defined by the

H(key)=(H(key)+x*x)%table_size

Let us consider we have to insert following elements that are:-
67, 90, 55, 17, 49.

67%10 = 7

90%10 = 0

55%10 = 5

17%10 = 7

49%10 = 9

0	**90**
1	
2	
3	
4	
5	**55**
6	
7	**67**
8	
9	

In this we can see if we insert 67, 90, and 55 it can be inserted easily but in the case of 17 hash function is used in such a manner that :-

To insert 17
The initial index generated is 17%10 = 7

But the index 7 is already filled with 67. Collision occurred.
To resolve this we try

(7+0*0)%10 = 7

(when x=0 it provide the index value 7 only) by making the increment in value of x. let x =1 so ,

(7+1*1)%10 = 8.

in this case bucket 8 is empty hence we will place 17 at index 8.

67%10 = 7

90%10 = 0

55%10 = 5

17%10 = 7

49%10 = 9

0	**90**
1	
2	
3	
4	
5	**55**
6	
7	**67**
8	**17**
9	**49**

4) Double hashing (Open addressing)

It is a technique in which two hash functions are used when there is an occurrence of collision. In this method 1 hash function is simple as same as division method. But for the second hash function there are two important rules which are

1. It must never evaluate to zero.
2. Must sure about the buckets, that they are probed.

The hash functions for this technique are:

H1(key) = key % table_size
H2(key) = P-(key mod P)

Where, **p** is a prime number which should be taken smaller than the size of a hash table.

Example: Let us consider we have to insert **67, 90,55,17,49**.

67%10 = 7
90%10 = 0
55%10 = 5
17%10 = 7
= 7 - (17%7)
= 7 - 3
= 4

49%10 = 9

0	**90**
1	**17**
2	
3	
4	
5	**55**
6	
7	**67**
8	
9	**49**

In this we can see 67, 90 and 55 can be inserted in a hash table by using first hash function but in case of 17 the bucket is full and in this case we have to use the second hash function which is

H2(key) = P - (key mod P)

where **P is a prime number which should be taken smaller than the hash table so value of P will be 7.**

i.e. **H2(17) = 7 - (17%7)**
= 7 - 3
= 4

that means we have to take **4 jumps for placing 17**. Therefore 17 will be placed at index 1.

```
/*------------------------------------------------------
      Program to implement Double Hashing
------------------------------------------------------*/
#include <stdio.h>
#include <conio.h>
#include <stdlib.h>
#define TABLE_SIZE 10

int h[TABLE_SIZE]={NULL};

void insert()
{

 int key,index,i,hkey,hash2;

 printf("\nenter a value to insert into hash table\n");
 scanf("%d",&key);

 hkey=key%TABLE_SIZE;
 hash2 = 7-(key %7);

 for(i=0;i<TABLE_SIZE;i++)
 {
   index=(hkey+i*hash2)%TABLE_SIZE;
   if(h[index] == NULL)
   {
       h[index]=key;
       break;
   }
 }

 if (i == TABLE_SIZE)
    printf("\nelement cannot be inserted\n");
}
```

```
void search()
{

 int key,index,i,hkey,hash2;
 printf("\nenter search element\n");
 scanf("%d",&key);
 hkey=key%TABLE_SIZE;
 hash2 = 7-(key %7);

 for (i=0;i<TABLE_SIZE; i++)
 {
   index=(hkey+i*hash2)%TABLE_SIZE;
   if(h[index]==key)
   {
     printf("value is found at index %d",index);
     break;
   }
  }

      if (i == TABLE_SIZE)
          printf("\n value is not found\n");
}


 void display()
 {

     int i;

    printf("\nelements in the hash table are \n");
    for(i=0;i< TABLE_SIZE; i++)
        printf("\n Hash table [ %d ] =  %d",i,h[i]);

}

  void main()
  {
      int opt,i;
      clrscr();
      printf("DOUBLE HASHING\n");

     while(1)
      {
         printf("\nPress 1.Insert 2.Display 3.Search 4.Exit \n");
         scanf("%d",&opt);
         switch(opt)
         {
            case 1: insert();
                    break;
            case 2: display();
                    break;
            case 3: search();
                    break;
            case 4: exit(0);
         }
      }
  }
```

Dynamic Hashing

To ensure good performance, it is necessary to increase the size of a hash table whenever its loading density exceeds a prescribed threshold. So, for example if we currently have b buckets in our hash table and using the division hash function with divisor D=b. When an insert causes the loading density to exceed the pre-specified threshold, we use array doubling to increase the number of buckets to 2b.

2-010	10
4-100	00
5-101	01
3-011	11

00	4
01	5
10	2
11	3

Now if we want to insert any mode values in the hash table they will overflow. To avoid this we can use dynamic hashing. We can add some more memory and readjust the values already stored in the hash table and add the new values in the hash table.

Using 2 digited index there is a possibility of having 4 buckets. If we use 3 digited index there is a possibility of using 8 buckets. The storage of hash table will be doubled to allow you to store more values.

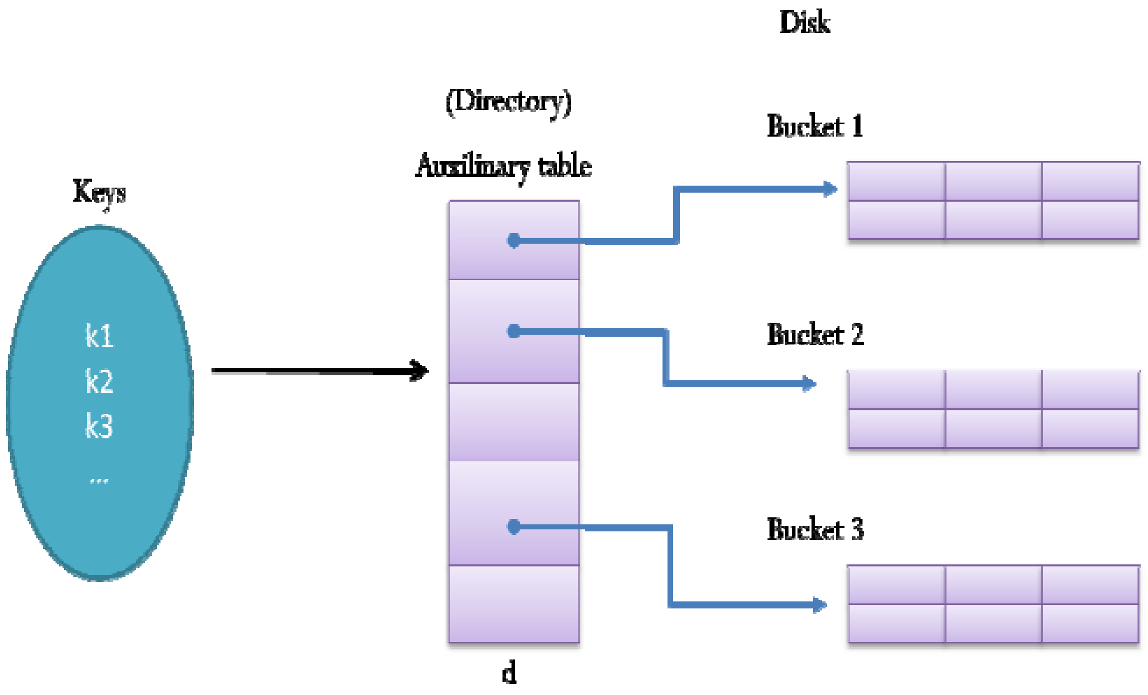
2-010
4-100
5-101
3-011
7-111
6-110

000	
001	
010	2
011	3
100	4
101	5
110	6
111	7

We consider two forms of Dynamic hashing- one uses a directory and the other does not.

1) **Dynamic Hashing Using Directories**
2) **Directory Less Dynamic Hashing**

Dynamic Hashing Using Directories



Dynamic Hashing

- The dynamic hashing method is used to overcome the problems of static hashing like bucket overflow.
- In this method, data buckets grow or shrink as the records increases or decreases. This method is also known as Extendable hashing method.
- This method makes hashing dynamic, i.e., it allows insertion or deletion without resulting in poor performance.

How to search a key

- First, calculate the hash address of the key.
- Check how many bits are used in the directory, and these bits are called as i.
- Take the least significant i bits of the hash address. This gives an index of the directory.
- Now using the index, go to the directory and find bucket address where the record might be.

How to insert a new record

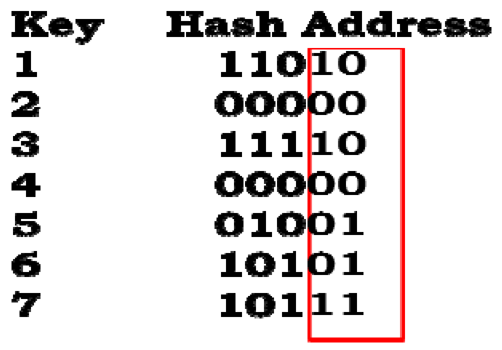
- Firstly, you have to follow the same procedure for retrieval, ending up in some bucket.
- If there is still space in that bucket, then place the record in it.
- If the bucket is full, then we will split the bucket and redistribute the records.

Example 1: Consider the following grouping of keys and insert them into buckets, depending on the prefix of their hash address:

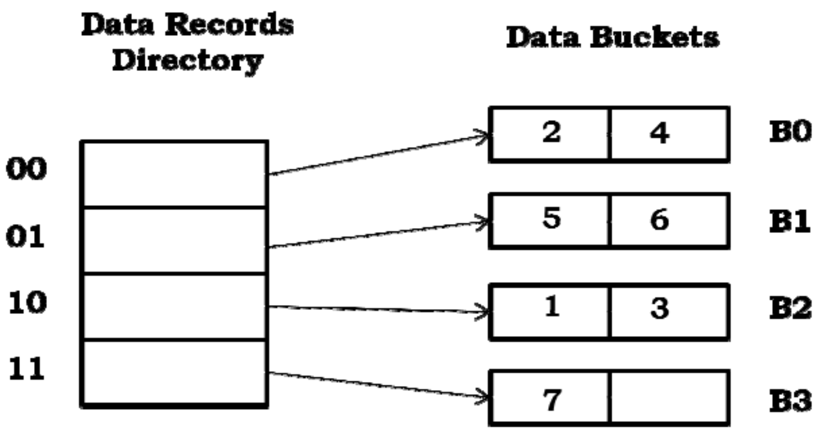
Key	Hash address
1	11010
2	00000
3	11110
4	00000
5	01001
6	10101
7	10111

Using two bits there is a possibility of producing 4 different codes. Assume that the directory has 4 codes and 4 buckets. Each bucket has 2 slots. Consider 00 pointing to Bucket B0, 01 pointing to Bucket B1, 10 pointing to bucket B2 and 11 pointing to bucket B3. Each bucket can store 2 values.

The last two bits of 2 and 4 are 00. So it will go into bucket B0. The last two bits of 5 and 6 are 01, so it will go into bucket B1. The last two bits of 1 and 3 are 10, so it will go into bucket B2. The last two bits of 7 are 11, so it will go into B3.

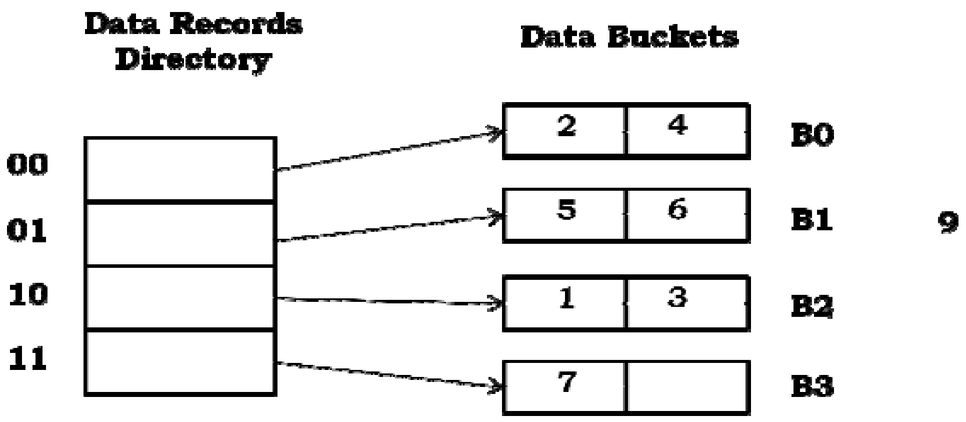


Key	Hash Address
1	11010
2	00000
3	11110
4	00000
5	01001
6	10101
7	10111



Insert key 9 with hash address 10001 into the above structure:

- Since **key 9** has hash address **10001**, it must go into the bucket B1. But bucket B1 is full, so it will get split.



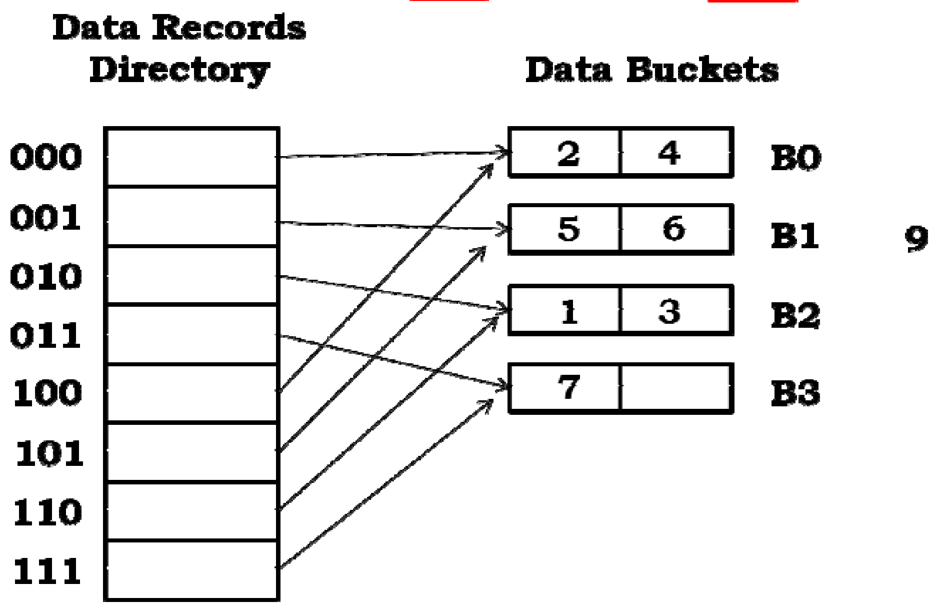
- The splitting will separate 5, 9 from 6 since last three bits of key 5 and key 9 are 001, so it will go into bucket B1, and the last three bits of key 6 are 101, so it will go into bucket B5.

- Keys 2 and 4 are still in B0. The record in B0 pointed by the 000 and 100 entry because last two bits of both the entry are 00.

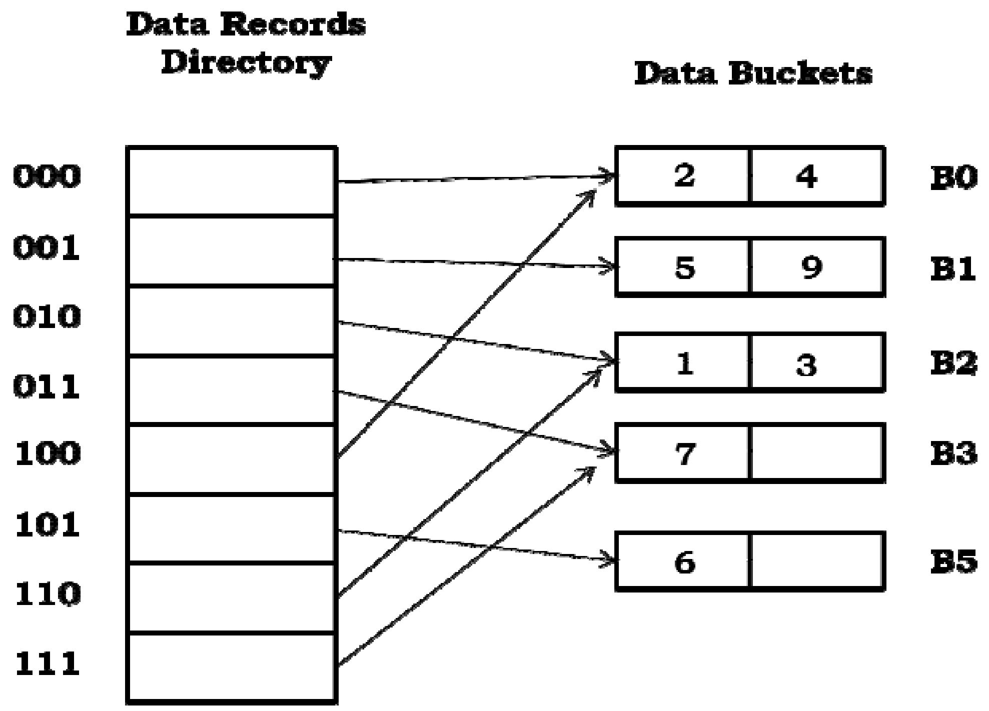
- Keys 1 and 3 are still in B2. The record in B2 pointed by the 010 and 110 entry because last two bits of both the entry are 10.

- Key 7 are still in B3. The record in B3 pointed by the 111 and 011 entry because last two bits of both the entry are 11.

5 - 01001	5 - 01001
6 - 10101	6 - 10101
9 - 10001	9 - 10001



5 - 01001	5 - 01001	B1
6 - 10101	6 - 10101	B5
9 - 10001	9 - 10001	B1



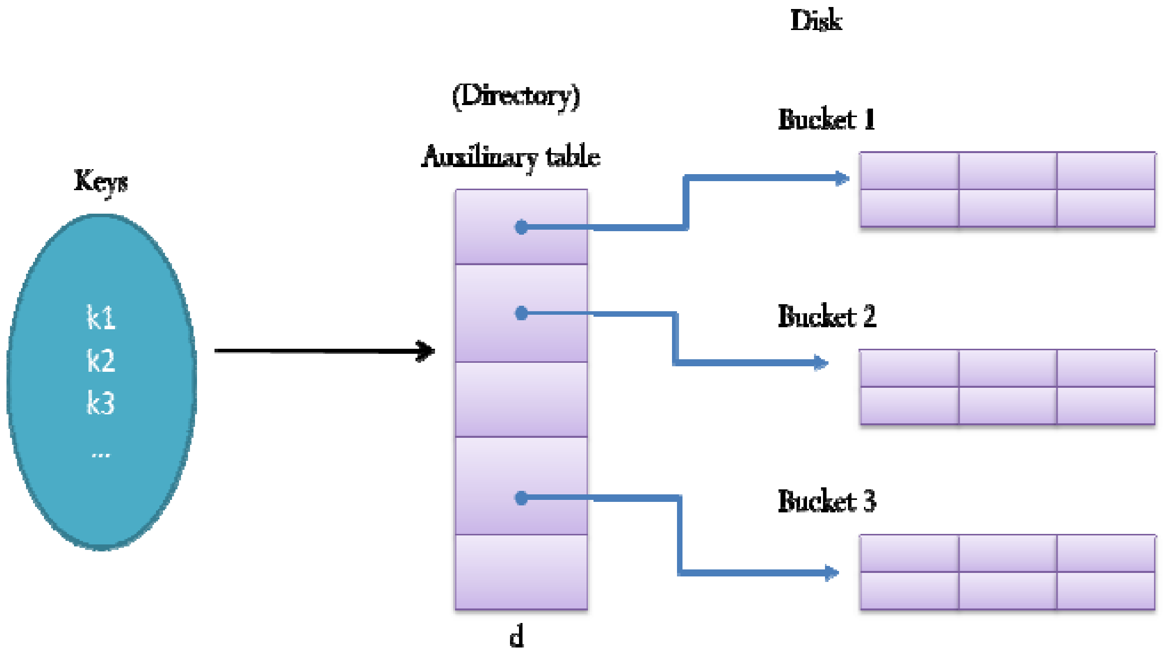
Advantages of dynamic hashing

- In this method, the performance does not decrease as the data grows in the system. It simply increases the size of memory to accommodate the data.
- In this method, memory is well utilized as it grows and shrinks with the data. There will not be any unused memory lying.
- This method is good for the dynamic database where data grows and shrinks frequently.

Disadvantages of dynamic hashing

- In this method, if the data size increases then the bucket size is also increased. These addresses of data will be maintained in the bucket address table. This is because the data address will keep changing as buckets grow and shrink. If there is a huge increase in data, maintaining the bucket address table becomes tedious.
- In this case, the bucket overflow situation will also occur. But it might take little time to reach this situation than static hashing.

Dynamic Hashing Using Directories Uses an auxiliary table to record the pointer of each bucket



PRIORITY QUEUES

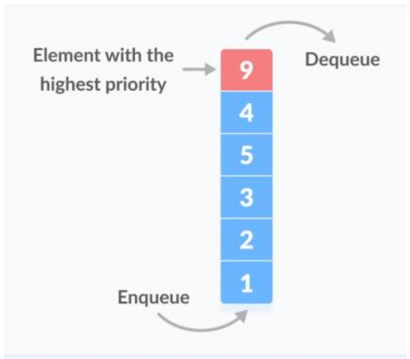
A priority queue is a **special type of queue** in which each element is associated with a **priority value**. And, elements are served on the basis of their priority. That is, higher priority elements are served first.

However, if elements with the same priority occur, they are served according to their order in the queue.

Assigning Priority Value

Generally, the value of the element itself is considered for assigning the priority. For example,

The element with the highest value is considered the highest priority element. However, in other cases, we can assume the element with the lowest value as the highest priority element.



Removing Highest Priority Element

Difference between Priority Queue and Normal Queue

In a queue, the **first-in-first-out rule** is implemented whereas, in a priority queue, the values are removed **on the basis of priority**. The element with the highest priority is removed first.

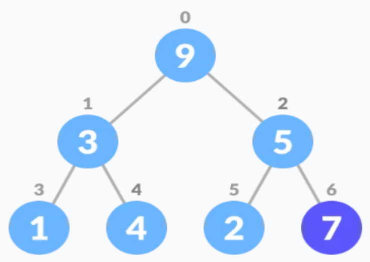
A double ended priority queue supports operations of both max heap (a max priority queue) and min heap (a min priority queue). The following operations are expected from double ended priority queue.

1. getMax() : Returns maximum element.
2. getMin() : Returns minimum element.
3. deleteMax() : Deletes maximum element.
4. deleteMin() : Deletes minimum element.
5. size() : Returns count of elements.
6. isEmpty() : Returns true if the queue is empty.

1. Inserting an Element into the Priority Queue

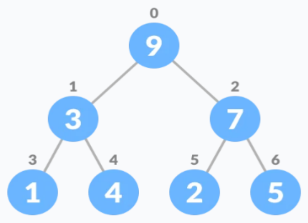
Inserting an element into a priority queue (max-heap) is done by the following steps.

- Insert the new element at the end of the tree.



Insert an element at the end of the queue

- Heapify the tree.

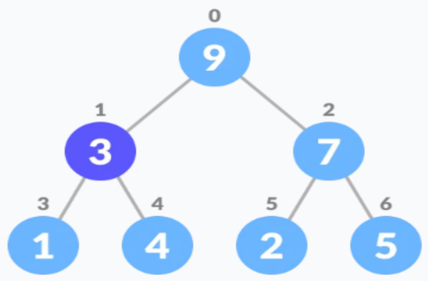


Heapify after insertion

2. Deleting an Element from the Priority Queue

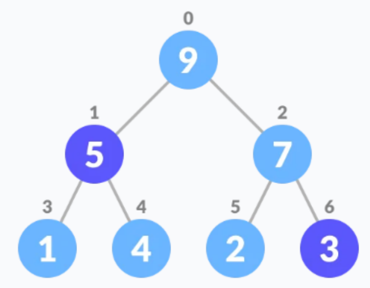
Deleting an element from a priority queue (max-heap) is done as follows:

- Select the element to be deleted.



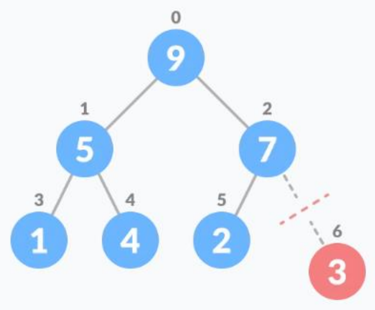
Select the element to be deleted

- Swap it with the last element.



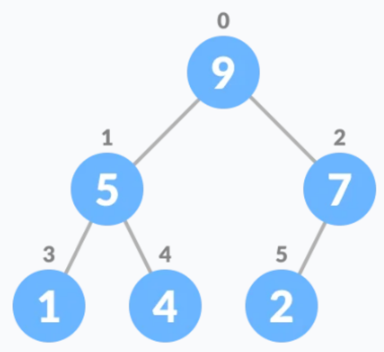
Swap with the last leaf node element

- Remove the last element.



Remove the last element leaf

- Heapify the tree.



Heapify the priority queue

Let n be the total number of elements in the two priority queues that are to be combined. If heaps are used to represent priority queues, then the combine operation takes $O(n)$ time. Using a leftist tree, the combine operation as well as the normal priority queue operations take logarithmic time.

In order to define a leftist tree, we need to introduce the concept of an extended binary tree. An *extended binary* tree is a binary tree in which all empty binary subtrees have been replaced by a square node. Figure 9.11 shows two example binary trees. Their corresponding extended binary trees are shown in Figure 9.12. The square nodes in an extended binary tree are called *external nodes*. The original (circular) nodes of the binary tree are called *internal nodes*.

Let x be a node in an extended binary tree. Let *left – child* (x) and *right – child* (x), respectively, denote the left and right children of the internal node x. Define *shortest* (x) to be the length of a shortest path from x to an external node. It is easy to see that *shortest* (x) satisfies the following recurrence:

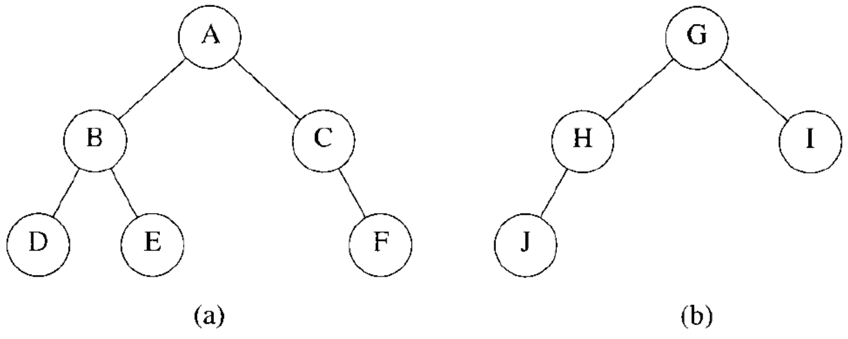


Figure 9.11: Two binary trees

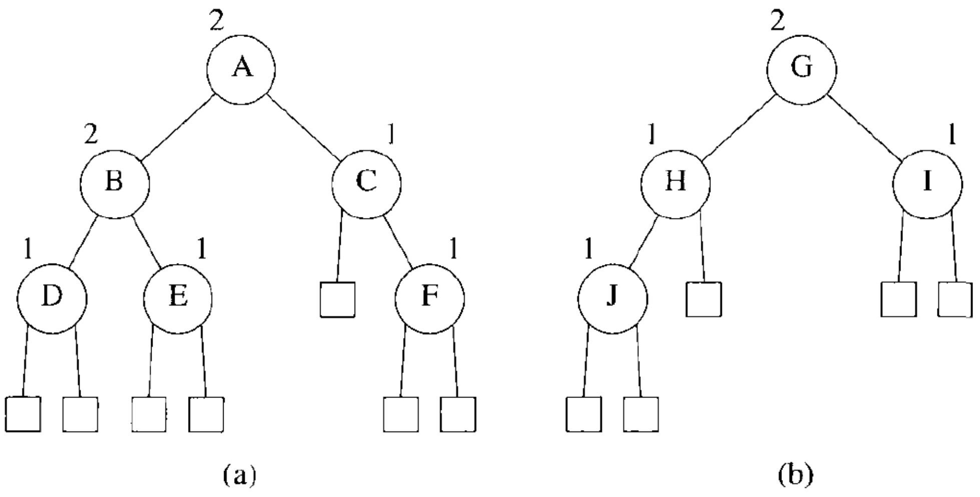


Figure 9.12: Extended binary trees corresponding to Figure 9.11

$$shortest(x) = \begin{cases} 0 & \text{if } x \text{ is an external node} \\ 1 + \min\{shortest(left-child(x)), shortest(right-child(x))\} & \text{otherwise} \end{cases}$$

The number outside each internal node x of Figure 9.12 is the value of $shortest(x)$.

Definition: A *leftist tree* is a binary tree such that if it is not empty, then

$$shortest(left-child(x)) \geq shortest(right-child(x))$$

for every internal node x. □

The binary tree of Figure 9.11(a) which corresponds to the extended binary tree of Figure 9.12(a) is not a leftist tree as $shortest(left-child(C) = 0$ while $shortest(right-child(C) = 1$. The binary tree of Figure 9.11(b) is a leftist tree.

OPTIMAL BINARY SEARCH TREES

To find an optimal binary search tree for a given static list, we must first decide on a cost measure for search trees. Assume that we wish to search for an identifier at level k of a binary search tree using the *search* 2 function. We know that *search* 2 makes k iterations of the **while** loop. Generally, the number of iterations of this loop equals the level number of the identifier we seek. Since the **while** loop determines the computing time of the search, it is reasonable to use the level number of a node as its cost.

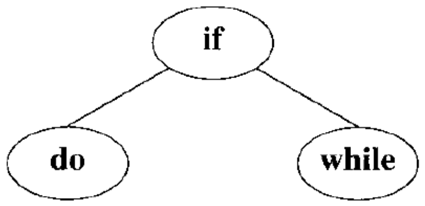


Figure 10.1: Binary search tree corresponding to a binary search on the list (**do, if, while**)

Consider the two search trees of Figure 10.2. The second tree requires at most three comparisons to decide whether the identifier we seek is in the tree. The first binary tree may require four comparisons, since any identifier that alphabetically comes after **for** but precedes **void** tests four nodes. Thus, the second binary tree has a better worst case search time than the first tree. Searching for an identifier in the first tree requires one comparison for **for**, two comparisons each for **do** and **while**, three comparisons for **void**, and four comparisons for **if**. If we search for each with equal probability, the average number of comparisons for a successful search is 2.4. The average number of comparisons for the second tree is only 2.2. Thus, the second tree also has better average behavior.

In evaluating binary search trees, it is useful to add a special *square* node at every place there is a null link. Doing this to the trees of Figure 10.2 yields the trees of Figure 10.3. Remember that every binary tree with n nodes has $n + 1$ null links and hence has $n + 1$ square nodes. We call these nodes *external* nodes because they are not part of the original tree. The remaining nodes are *internal* nodes. Each time we search for an identifier that is not in a binary search tree, the search terminates at an external node. Since all such searches represent unsuccessful searches, we also refer to external nodes as *failure* nodes. A binary tree with external nodes added is an *extended binary tree.* Figure 10.3 shows the extended binary trees corresponding to the search trees of Figure 10.2.

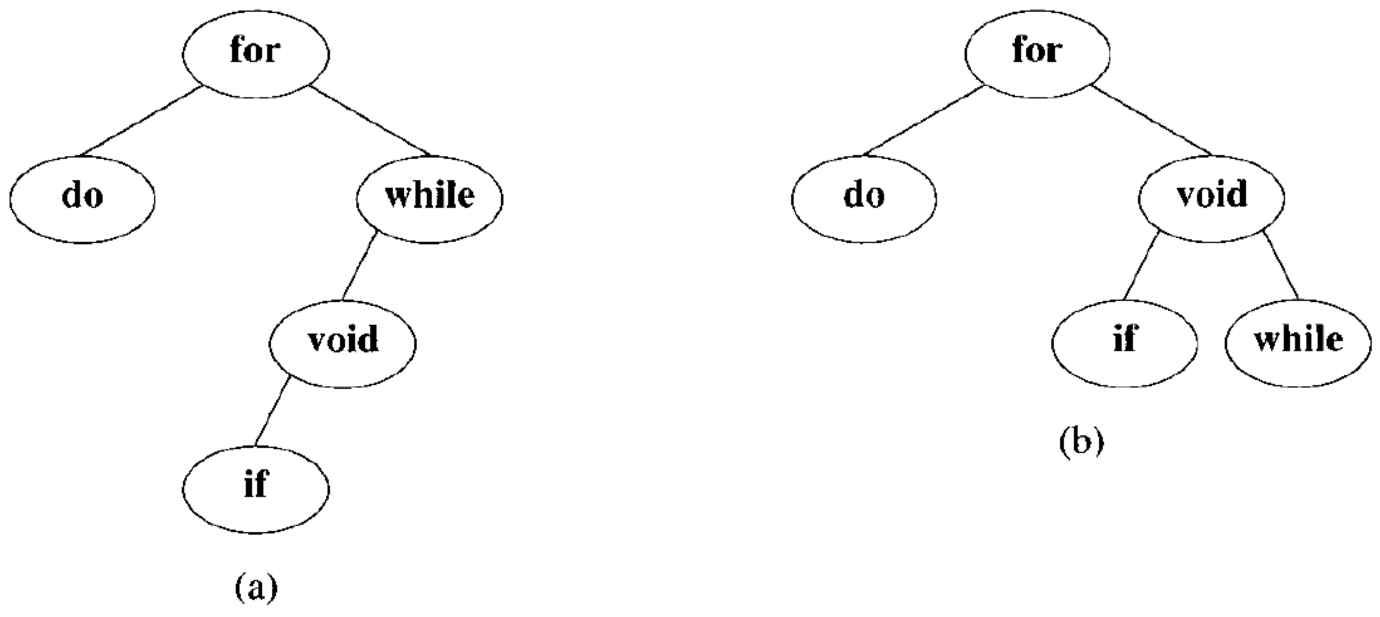


Figure 10.2: Two possible binary search trees

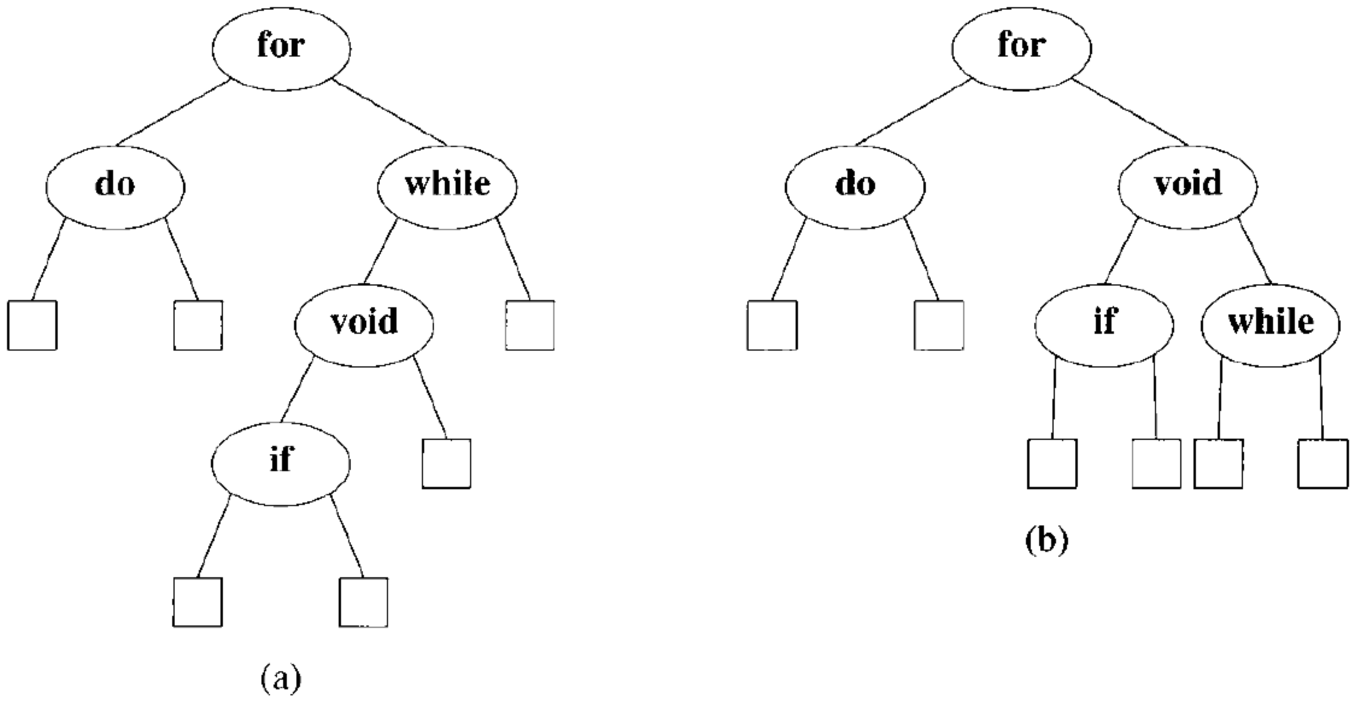


Figure 10.3: Extended binary trees corresponding to search trees of Figure 10.2

If T_{ij} is an optimal binary search tree for $a_{i+1}, \cdots, a_j$ and $r_{ij} = k$, then k satisfies the inequality $i < k \le j$. T_{ij} has two subtrees L and R. L is the left subtree and contains the identifiers $a_{i+1}, \cdots, a_{k-1}$ and R is the right subtree and contains the identifiers $a_{k+1}, \cdots, a_j$ (Figure 10.5). The cost c_{ij} of T_{ij} is

$$c_{ij} = p_k + \text{cost}\ (L) + \text{cost}\ (R) + \text{weight}\ (L) + \text{weight}\ (R) \tag{10.2}$$

where weight (L) = weight $(T_{i,k-1}) = w_{i,k-1}$, and weight (R) = weight $(T_{kj}) = w_{kj}$.

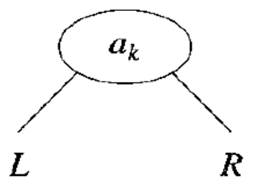


Figure 10.5: An optimal binary search tree T_{ij}

From Eq. (10.2) it is clear that c_{ij} is minimal only if cost(L)=$c_{i,k-1}$ and cost(R) = c_{kj}. Otherwise we could replace either L or R with a subtree of lower cost and thus obtain a binary search tree for $a_{i+1}, \ldots, a_j$ with a lower cost than c_{ij}. This violates the assumption that T_{ij} is optimal. Hence, Eq. (10.2) becomes:

$$\begin{aligned} c_{ij} &= p_k + c_{i,k-1} + c_{kj} + w_{i,k-1} + w_{kj} \\ &= w_{ij} + c_{i,k-1} + c_{kj} \qquad (10.3) \end{aligned}$$

Since T_{ij} is optimal, it follows from Eq. (10.3) that $r_{ij} = k$ is such that

$$w_{ij} + c_{i,k-1} + c_{kj} = \min_{i<l\leq j}\{w_{ij} + c_{i,l-1} + c_{lj}\}$$

or

$$c_{i,k-1} + c_{kj} = \min_{i<l\leq j}\{c_{i,l-1} + c_{lj}\} \qquad (10.4)$$

Equation (10.4) shows us how to obtain T_{on} and c_{on}, starting from the knowledge that $T_{ii} = \phi$ and $c_{ii} = 0$.

```
void obst(int p[], int q[], int cost[][MAX_SIZE],
     int root[][MAX_SIZE], int weight[][MAX_SIZE], int n)
{
/* given n distinct identifiers a[1] ... a[n] and
probabilities p[1] ... p[n], and q[0] ... q[n] compute
the cost c[i][j]  of the optimal binary search tree for
a[i]  ... a[j], 1 ≤ i ≤ j ≤ n. Also compute the weight
and root of the tree */
   int i,j,k,m,min,minpos;
   /* initialize 0 and 1 node trees */
   for (i = 0; i < n; i++) {
      weight[i][i] = q[i];
      root[i][i] = 0;
      cost[i][i] = 0;
      cost[i][i+1] = weight[i][i+1] =
                     q[i] + q[i+1] + p[i+1];
      root[i][i+1] = i+1;
   }
   weight[n][n] = q[n];
   root[n][n] = 0;
   cost[n][n] = 0;
   /* compute remaining diagonals */
   for (m = 2; m <= n; m++)
      for (i = 0; i <= n-m; i++) {
         j = i + m;
         weight[i][j] = weight[i][j-1] + p[j] + q[j];
         k = knuth_min(cost,root,i,j);
         /* knuth_min returns a value, k, in the range
         root[i][j-1] to root[i+1][j], that minimizes
         cost[i][k-1] + cost[k][j] */
         cost[i][j] = weight[i][j] + cost[i][k-1]
                                   + cost[k][j];
         root[i][j] = k;
      }
}
```

Program 10.1: Function to find an optimal binary search tree