UNIT – I

1.1 INTRODUCTION TO DATA STRUCTURES :

Data Structure is a way of collecting and organising data in such a way that we can perform operations on these data in an effective way. Data Structures is about rendering data elements in terms of some relationship, for better organization and storage. For example, we have data player's name "Virat" and age 26. Here "Virat" is of String data type and 26 is of integer data type.

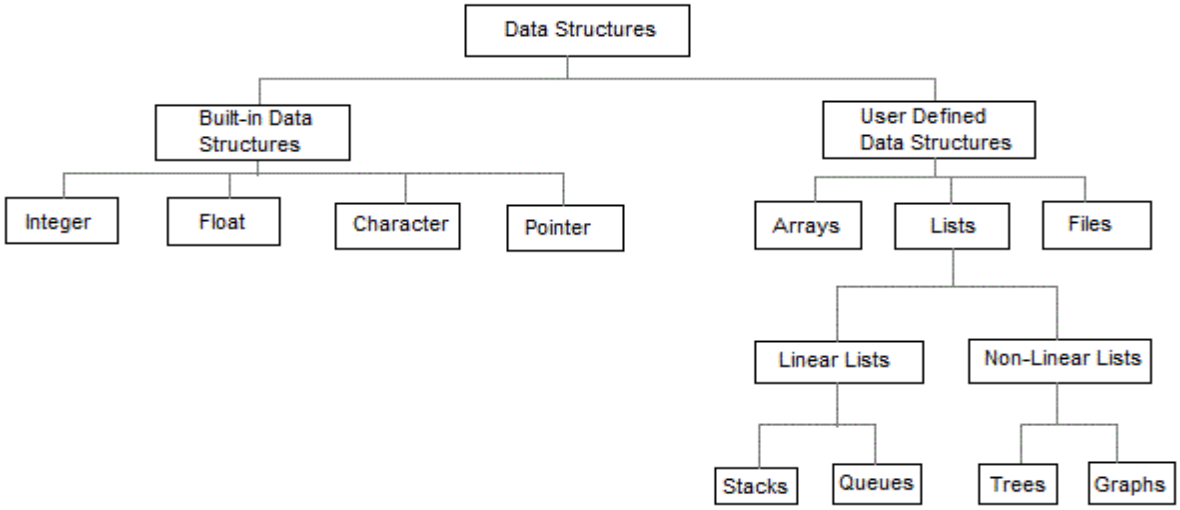
Basic types of Data Structures

As we discussed above, anything that can store data can be called as a data structure, hence Integer, Float, Boolean, Char etc, all are data structures. They are known as Primitive Data Structures.

Then we also have some complex Data Structures, which are used to store large and connected data. Some example of Abstract Data Structure are :

- Linked List
- Tree
- Graph
- Stack, Queue etc.

All these data structures allow us to perform different operations on data. We select these data structures based on which type of operation is required. We will look into these data structures in more details in our later lessons.



INTRODUCTION TO DATA STRUCTURES

A digital computer can manipulate only primitive data. that is. data in terms of O's and 1'5. Manipulation of primitive data is inherent within the computer and docs nOI require any extra effort on the part of the user. BUI in our real-life applications. various kinds of data other than the primitive data are involved. Manipulation of real-life data (also termed user data) requires the following essential tasks:

I. Storage representation of user data: User data should be stored in such a way that the computer can understand it.

2. Retrieval 0/ stored data: Data stored in a computer should be retrieved in such a way that the user can understand it.

3. Transformation 'if IIser data: Various operations which require to be performed on user data so that it can be transformed from one form to another.

The basic theory of computer science deals with the manipulation of various kinds of data. wherefrom the concept of data structures comes in. In fact. data structures constitute the fundamentals of computer science. For a given kind of user data. its structure implies the following:

I. Domain *(z,):* This is the range of values that the data may have. This domain is also termed data object.

2. Function (7): This is the set of operations which may legally be applied to elements of the data object, This implies that for a data structure. we must specify the set of operations.

3. Axioms Os): This is the set of rules with which the different operations belonging to 7 can actually be implemented.

Now we can define the term data structure.

A *data structure* D is a triplet, that is, 0 = *(z,.* 7. *,t)* where *Z,* is a set of data Object, 7 is a set of functions and,t is a set of rules to implement the functions, Let us consider an example.

We know that for the integer data type (int) in the C programming language the structure includes the following types:

Z, = (0. ±!. ±2. ±3.. ,,)

7 = (+, -, ., *I,* %)

,t = (A set of binary arithmctics to perform addition. subtraction, division. multiplication. and modulo operations.)

It can be easily recognized that the triplet *(Z" 7,,t)* is nothing but an abstract data type. Also. the elements in set *Z,* are not necessarily from primitive data; it may contain elements from some other abstract data types. Alternatively. an implementation of a data structure D is a mapping from *Z,* to a set of other data structures 0/. i = 1.2. '". n, for some *n.* More precisely. this mapping specifies how every object of *Z,* is [0 be represented by

the objects of 0i' i = l , 2.
"" *n.* Every function of 0 must be written using the function of the implementing data structures *Di•* i = I. 2. " .. *II.* The fact is that each of the implementing data structures is either a primitive data type or an abstract data type. We can conclude the discussion with another example.

Suppose, we want to implement a data type. namely Complex as an abstract data type. Any variable of the complex data type has two parts: a real part and an imaginary part. In our usual notation. if z is a complex number. then $z = x + iy$, where x and)' are the real and imaginary pans. respectively. Both x and y are of the Real data Iype which is another abstract data type (available as a built-in data type). So, the abstract data type Complex can be defined using the data structure
Real 35

Complex z(*x :*Real y:Real)

Now the set *Z,* of Complex can be realized from the domain of x and)' which is Real in this case. Let us specify the set of operations for the Complex data type. which are stated as 7:

7 = ($, -. ®. +. V.
II)

Assume that 'I = *XI* + *iYI* and 42 = *x2* + iY2 are two data of the Complex data type. Then we call define the rules for implementing the operations in 7. thus giving axioms rio III the current example. for the Complex data type.

1.2 ASYMPTOTIC NOTATIONS:

Introduction:

Performance Evaluation: The goal of it is evaluating efficiency of algorithm. There are many factors in consideration for finding efficiency of an algorithm. Those are:
(i) Does it do what we want to do.
(ii) Does it work correctly according to original specifications.
(iii) Is the documentation describes hot to use it and how it works.
(iv) Are procedures created meet logical sub functions.
(v) Is code readable.
All these criteria are very important when it comes to write software for large systems. There are other criteria which have direct relationship to performance. Those are computing time and space requirements.
The performance evaluation is divided into 2 phases. Those are apriori estimates and posteriori testing. These are also referred as Performance Analysis and Performance Measurement respectively.

The empirical or posteriori testing executes the program for various instances of the problem and then chooses the best among them. The disadvantage is it is dependent on machine on which program is executed, the language in which program is implemented and even on skills of the programmer.
The theoretical or apriori estimates determines resources such as time and space needed by algorithm as a function of a parameter related to problem considered. The parameter often used is the size of input instances. The **advantage of apriori is it entirely machine, language and program independent.**

Space Complexity: It is the amount of memory needed to run program completely.

The space complexities for the following 3 types of problems are shown below.

(i)
```
1   Alorithm abc(a,b,c)
2   {
3   return (a+b+b*c+(a+b-c)/(a+b)+4.0;
4   }
```

The space needed by abc algorithm is 4 words(c value) because the word is adequate for storing a,b,c and result.It means abc is independent of instance characteristics i.e Sp=0.

(ii)
```
1   Algorithm sum(a,n)
2   {
3   s=0.0;
4   for i=0 to n do
5   s:=s+a[i];
6   return s;
7   }
```

The sum() algorithm is characterized by instance n.The space needed by sum(a,n) is >= (n+3) because n,s and i takes one word each and the space for the array is the space needed to store n variables i.e n words.

(iii)
```
1   Algorithm Rsum(a,n)
2   {
3   if(n<=0) return 0.0;
4   else return Rsum(a,n-1)+a[n];
5   }
```

It is characterized by recursion stack space which includes space for return address, local variables and formal parameters. The space needed by Rsum(a,n) is >= 3(n+1) because values of n,return address and a pointer to a[] is 3 words and the depth of the recursion is (n+1).

The space needed by these algorithms is computed based on two components.

(i) **Fixed part:** It is independent of instance characteristics such as number and size of inputs and outputs. It includes instruction space, space of simple and fixed sized component variables and constants.

(ii) **Variable part:** It contains the space needed by component variables whose size dependent on particular problem instance, space of reference variables and recursion stack space.

The space requirement of any algorithm P is written as S(P)=c+Sp where c is a constant and Sp is instance characteristics.

3.2.2 Time Complexity: It is the amount of time needed to run a program completely.

The apriori testing computes time complexity based on two factors:

(i) the number of times a statement is executed in the program called as frequency count.

(ii) the time taken for a single execution of a statement.

The second factor depends on machine instruction set, machine configuration. since the apriori is independent of machine, **it considers only the first factor and computes the efficiency of the program as a function of the total frequency count of the statements of a program.**

```
. . . . . .
x=x+2;
. . . . .
```

segment A

```
. . . . . .
for i=1 to n do
x=x+2;
end
```

segment B

```
. . . .. . .
for i=1 to n do
  for i=1 to n do
        x=x+2;
```

segment C

The frequency count is estimated for the above segments for x=x+2; is

The frequency count for x=x+2; in segment A is 1.It is in segment B is n since the for executes embedded statement for n times.In segment C,it is n2 since the statement embedded in nested for and is executed n2 times.

In apriori analysis,the total frequency count is computed by summing the frequency count fi of each statement in the program.i.e T=∑fi.

The total frequency counts are computed for segments A,B and C are as follows:

The frequency count of for i=low_index to up_index is (up_index - low_index+1)+1.

It is computed for the statement inside for is low_index – up_index+1.

Statements	Frequency count
. .. .	
x=x+2;	1
. . . .	
Total frequency count	1

Statements	**Total frequency count of segment A**
. . . .	
for i=1 to n do	n+1
For k=1 to n do	n(n+1)
x=x+2;	n2
End	n2
End	N
. . . .. . …	
Total frequency count	3n2+3n+1

The total frequency counts of the segments A,B and C are 1,3n+1 and $3n^2+3n+1$ respectively have time complexities O(1),O(n) and $O(n^2)$ because total frequency counts are proportional to 1,n, and n2.

Computing time complexities for recursive functions:

A procedure P has a call statement to itself or to another procedure results in a call to itself, then P is said to be recursive procedure. The former case is called direct recursion and latter is called indirect recursion.

Any recursive procedure have certain properties. They are:

(i) There must be criteria ,one or more called base criteria where the procedure doesn't call itself either directly or indirectly.

(ii) Each time the procedure calls itself directly or indirectly, it must be closer to the base criteria.

(i) **computing Time complexity for recursive factorial function:**

The recurrence relation for a recursive factorial function is

T(n)=c+T(n-1) if n>1

= d if n<=1

The time complexity for T(n) is computed as follows.

T(n)=c+T(n-1)

=c+(c+T(n-2)

=2c+(c+T(n-3))

=3c+T(n-3)

For Kth step, the recurrence relation transformed to as T(n)=k.c+T(n-k) if n>k

Finally, when (k=n-1),it become T(n)=(n-1).c+T(n-(n-1))

= (n-1).c+T(1) // T(n)=1 if n<=1

= (n-1).c+d

=O(n)

(ii) computing Time complexity for Towers of Hanoi:

The recurrence relation for Towers of Hanoi is T(n)=0 if n=0

=2.T(n-1)+1 if n>0

The time complexity is computed as follows.

T(n)=2.T(n-1)+1

= 2.(2.T(n-2)+1)+1

$=2^2$T(n-2)+2+1

$=2^2$.(2.T(n-3)+1)+2+1

$=2^3$.(T(n-3)+2^2+2+1

For kth step,the recurrence relation is transformed as

T(n)=2^kT(n-k)+$2^{(k-1)}$+$2^{(k-2)}$+ . . .+2^2+2+1

Finally,when k=n,It become

T(n)=2^nT(0)+$2^{(n-1)}$ +$2^{(n-2)}$ +…….+2^2+2+1

$=2^n$.0+2^{n-1}

$=2^n$-1

$=O(2^n)$

Asymptotically speaking, in the limit as N tends towards infinity, 2 + 3 + .,,,.. + N gets closer and closer to the pure quadratic function (1/2)N^2. And what difference does the constant factor of 1/2 make, at this level of abstraction? So the behavior is said to be $O(n^2)$.

The O Notation

Definition

The O (pronounced *big-oh*) is the formal method of expressing the upper bound of an algorithm's running time. It's a measure of the longest amount of time it could possibly take for the algorithm to complete.

More formally, for non-negative functions, *f(n)* and *g(n)*, if there exists an integer $n_0$and a constant *c* > 0 such that for all integers $n > n_0$, *f(n)* ≤ *cg(n)*, then *f(n)* is Big O of *g(n)*. This is denoted as "*f(n)* = *O(g(n))*". If graphed, *g(n)* serves as an upper bound to the curve you are analyzing, *f(n)*. Note that if *f* can take on finite values only (as it should happen normally) then this definition implies that there exists some constant *C* (potentially larger than *c*) such that for all values of *n*, *f(n)* ≤ *Cg(n)*. An appropriate value for *C* is the maximum of *c* and $\max_{1 \le n \le n_0} f(n)/g(n)$.

Theory Examples

So, let's take an example of Big-O. Say that f(n) = 2n + 8, and g(n) = n^2. Can we find a constant n_0, so that 2n + 8 <= n^2? The number 4 works here, giving us 16 <= 16. For any number n greater than 4, this will still work. Since we're trying to generalize this for large values of n, and small values (1, 2, 3) aren't that important, we can say that f(n) is generally faster than g(n); that is, f(n) is bound by g(n), and will always be less than it. It could then be said that f(n) runs in O(n^2) time: "f-of-n runs in Big-O of n-squared time".

To find the upper bound - the Big-O time - assuming we know that f(n) is equal to (exactly) 2n + 8, we can take a few shortcuts. For example, we can remove all constants from the runtime; eventually, at some value of c, they become irrelevant. This makes f(n) = 2n. Also, for convenience of comparison, we remove constant multipliers; in this case, the 2. This makes

f(n) = n. It could also be said that f(n) runs in O(n) time; that lets us put a tighter (closer) upper bound onto the estimate.

Practical Examples

O(n): printing a list of n items to the screen, looking at each item once.

O(ln n): also "log n", taking a list of items, cutting it in half repeatedly until there's only one item left.

O(n^2): taking a list of n items, and comparing every item to every other item.

Big-Omega Notation

For non-negative functions, *f(n)* and *g(n)*, if there exists an integer $n_0$and a constant *c* > 0 such that for all integers $n > n_0$, *f(n)* ≥ *cg(n)*, then *f(n)* is omega of *g(n)*. This is denoted as "*f(n) = Ω(g(n))*".

This is almost the same definition as Big Oh, except that "*f(n)* ≥ *cg(n)*", this makes *g(n)* a lower bound function, instead of an upper bound function. It describes the **best that can happen** for a given data size.

Theta Notation

For non-negative functions, *f(n)* and *g(n)*, *f(n)* is theta of *g(n)* if and only if *f(n) = O(g(n))* and *f(n) = Ω(g(n))*. This is denoted as "*f(n) = Θ(g(n))*".

This is basically saying that the function, *f(n)* is bounded both from the top and bottom by the same function, *g(n)*.

The theta notation is denoted by Q.

Little-O Notation

For non-negative functions, *f(n)* and *g(n)*, *f(n)* is little o of *g(n)* if and only if *f(n) = O(g(n))*, but *f(n) ≠ Θ(g(n))*. This is denoted as "*f(n) = o(g(n))*".

This represents a loose bounding version of Big O. *g(n)* bounds from the top, but it does not bound the bottom.

Little Omega Notation

For non-negative functions, *f(n)* and *g(n)*, *f(n)* is little omega of *g(n)* if and only if *f(n) = Ω(g(n))*, but *f(n) ≠ Θ(g(n))*. This is denoted as "*f(n) = ω(g(n))*".

Much like Little Oh, this is the equivalent for Big Omega. *g(n)* is a loose lower boundary of the function *f(n)*; it bounds from the bottom, but not from the top.

1.3 ONE DIMENSIONAL ARRAY

A one-dimensional array (or single dimension array) is a type of linear array. Accessing its elements involves a single subscript which can either represent a row or column index.

As an example consider the C declaration int anArrayName[10];

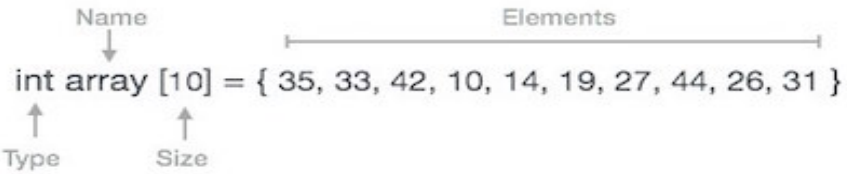
Syntax : datatype anArrayname[sizeofArray];

Array is a container which can hold fix number of items and these items should be of same type. Most of the datastructure make use of array to implement their algorithms. Following are important terms to understand the concepts of Array.

- **Element** − Each item stored in an array is called an element.
- **Index** − Each location of an element in an array has a numerical index which is used to identify the element.

Array Representation

Arrays can be declared in various ways in different languages. For illustration, let's take C array declaration.



Arrays can be declared in various ways in different languages. For illustration, let's take C array declaration.

As per above shown illustration, following are the important points to be considered.

- Index starts with 0.
- Array length is 8 which means it can store 8 elements.
- Each element can be accessed via its index. For example, we can fetch element at index 6 as 9.

Basic Operations

Following are the basic operations supported by an array.

- **Traverse** – print all the array elements one by one.
- **Insertion** – add an element at given index.
- **Deletion** – delete an element at given index.
- **Search** – search an element using given index or by value.
- **Update** – update an element at given index.

In C, when an array is initialized with size, then it assigns defaults values to its elements in following order.

Insertion Operation

Insert operation is to insert one or more data elements into an array. Based on the requirement, new element can be added at the beginning, end or any given index of array.

Here, we see a practical implementation of insertion operation, where we add data at the end of the array –

Algorithm

Let **Array** is a linear unordered array of **MAX** elements.

Example

Result

Let LA is a Linear Array (unordered) with N elements and K is a positive integer such that K<=N. Below is the algorithm where ITEM is inserted into the K^{th} position of LA

1. Start
2. Set J=N
3. Set N = N+1
4. Repeat steps 5 and 6 while J >= K

5. Set LA[J+1] = LA[J]
6. Set J = J-1
7. Set LA[K] = ITEM
8. Stop

Example

Below is the implementation of the above algorithm −

```
#include <stdio.h>
main() {
   int LA[] = {1,3,5,7,8};
   int item = 10, k = 3, n = 5;
   int i = 0, j = n;

   printf("The original array elements are :\n");

   for(i = 0; i<n; i++) {
      printf("LA[%d] = %d \n", i, LA[i]);
   }

   n = n + 1;

   while( j >= k){
      LA[j+1] = LA[j];
      j = j - 1;
   }

   LA[k] = item;

   printf("The array elements after insertion :\n");

   for(i = 0; i<n; i++) {
      printf("LA[%d] = %d \n", i, LA[i]);
   }
}
```

When compile and execute, above program produces the following result −

```
The original array elements are :
LA[0]=1
LA[1]=3
LA[2]=5
LA[3]=7
LA[4]=8
The array elements after insertion :
LA[0]=1
LA[1]=3
LA[2]=5
LA[3]=10
LA[4]=7
LA[5]=8
```

Deletion Operation

Deletion refers to removing an existing element from the array and re-organizing all elements of an array.

Algorithm

Consider LA is a linear array with N elements and K is a positive integer such that K<=N. Below is the algorithm to delete an element available at the K^{th} position of LA.

1. Start
2. Set J=K
3. Repeat steps 4 and 5 while J < N
4. Set LA[J-1] = LA[J]
5. Set J = J+1
6. Set N = N-1
7. Stop

Example

Below is the implementation of the above algorithm −

```
#include <stdio.h>
main() {
   int LA[] = {1,3,5,7,8};
   int k = 3, n = 5;
   int i, j;

   printf("The original array elements are :\n");

   for(i = 0; i<n; i++) {
      printf("LA[%d] = %d \n", i, LA[i]);
   }

   j = k;

   while( j < n){
      LA[j-1] = LA[j];
      j = j + 1;
   }

   n = n -1;

   printf("The array elements after deletion :\n");

   for(i = 0; i<n; i++) {
      printf("LA[%d] = %d \n", i, LA[i]);
   }
}
```

When compile and execute, above program produces the following result −

```
The original array elements are :
LA[0]=1
LA[1]=3
LA[2]=5
```

LA[3]=7
LA[4]=8
The array elements after deletion :
LA[0]=1
LA[1]=3
LA[2]=7
LA[3]=8

Search Operation

You can perform a search for array element based on its value or its index.

Algorithm

Consider LA is a linear array with N elements and K is a positive integer such that K<=N. Below is the algorithm to find an element with a value of ITEM using sequential search.

1. Start
2. Set J=0
3. Repeat steps 4 and 5 while J < N
4. IF LA[J] is equal ITEM THEN GOTO STEP 6
5. Set J = J +1
6. PRINT J, ITEM
7. Stop

Example

Below is the implementation of the above algorithm −

```
#include <stdio.h>
main() {
   int LA[] = {1,3,5,7,8};
   int item = 5, n = 5;
   int i = 0, j = 0;

   printf("The original array elements are :\n");

   for(i = 0; i<n; i++) {
      printf("LA[%d] = %d \n", i, LA[i]);
   }

   while( j < n){

      if( LA[j] == item ){
         break;
      }

      j = j + 1;
   }

   printf("Found element %d at position %d\n", item, j+1);
}
```

When compile and execute, above program produces the following result −

```
The original array elements are :
LA[0]=1
```

LA[1]=3
LA[2]=5
LA[3]=7
LA[4]=8
Found element 5 at position 3

Update Operation

Update operation refers to updating an existing element from the array at a given index.

Algorithm

Consider LA is a linear array with N elements and K is a positive integer such that K<=N. Below is the algorithm to update an element available at the K^{th} position of LA.

1. Start
2. Set LA[K-1] = ITEM
3. Stop

Example

Below is the implementation of the above algorithm −

```
#include <stdio.h>
main() {
   int LA[] = {1,3,5,7,8};
   int k = 3, n = 5, item = 10;
   int i, j;

   printf("The original array elements are :\n");

   for(i = 0; i<n; i++) {
      printf("LA[%d] = %d \n", i, LA[i]);
   }

   LA[k-1] = item;

   printf("The array elements after updation :\n");

   for(i = 0; i<n; i++) {
      printf("LA[%d] = %d \n", i, LA[i]);
   }
}
```

When compile and execute, above program produces the following result −

```
The original array elements are :
LA[0]=1
LA[1]=3
LA[2]=5
LA[3]=7
LA[4]=8
The array elements after updation :
LA[0]=1
LA[1]=3
LA[2]=10
```

LA[3]=7
LA[4]=8

APPLICATION OF ARRAYS

Whenever we require a collection of data objects of the same type and want to process them as a single unit, an array can be used, provided the number of data items is constant or fixed. Arrays have a wide range of applications ranging from business data processing to scientific calculations to industrial projects.

Implementation of a Static Contiguous List

A *list* is a structure in which insertions, deletions, and retrieval may occur at any position in the list. Therefore, when the list is static, it can be implemented by using an array. When a list is implemented or realized by using an array, it is a *contiguous list*. By contiguous, we mean that the elements are placed consecutively one after another starting from some address, called the *base address*. The advantage of a list implemented using an array is that it is randomly accessible. The disadvantage of such a list is that insertions and deletions require moving of the entries, and so it is costlier. A static list can be implemented using an array by mapping the i^{th} element of the list into the i^{th} entry of the array, as shown in Figure below.

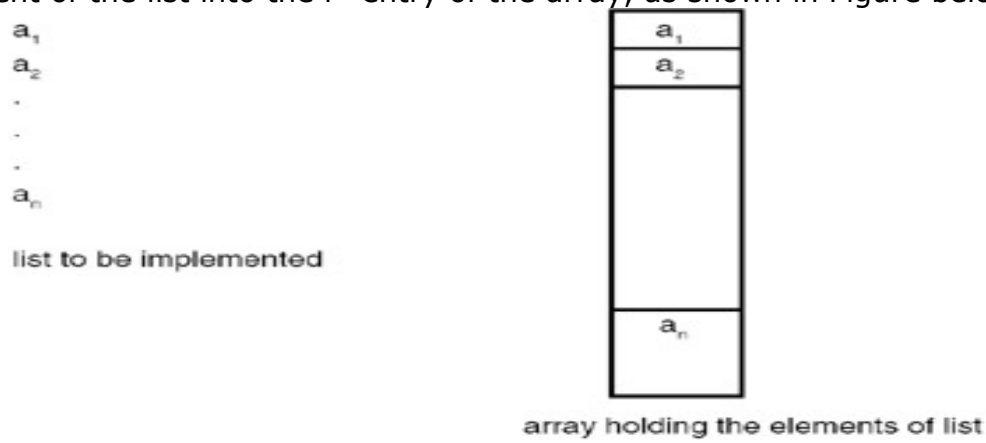


Figure : Implementation of a static contiguous list.

Program

A complete C program for implementing a list with operations for reading values of the elements of the list and displaying them is given here:

```
#include<stdio.h>
#include<conio.h>
void main()
{
  void read(int *,int);
  void dis(int *,int);
  int a[5],i,sum=0;
  clrscr();
  printf("Enter the elements of array \n");
  read(a,5);      /*read the array*/
  printf("The array elements are \n");
  dis(a,5);
```

```
}

void read(int c[],int i)
{
  int j;
  for(j=0;j<i;j++)
  scanf("%d",&c[j]);
  fflush(stdin);
}

void dis(int d[],int i)
{
  int j;
  for(j=0;j<i;j++)
     printf("%d ",d[j]);
  printf("\n");
}
```

Example

Input

Enter the elements of the first array
15
30
45
60
75

Output

The elements of the first array are
15 30 45 60 7

1.4 MULTI DIMENSIONAL ARRAY

1.4.1 Two Dimensional Arrays

The simplest form of multidimensional array is the two-dimensional array. A two-dimensional array is, in essence, a list of one-dimensional arrays. To declare a two-dimensional integer array of size [x][y], you would write something as follows −

type arrayName [x][y];

Where **type** can be any valid C data type and **arrayName** will be a valid C identifier. A two-dimensional array can be considered as a table which will have x number of rows and y number of columns. A two-dimensional array **a**, which contains three rows and four columns can be shown as follows −

	Column 0	Column 1	Column 2	Column 3
Row 0	a[0][0]	a[0][1]	a[0][2]	a[0][3]
Row 1	a[1][0]	a[1][1]	a[1][2]	a[1][3]
Row 2	a[2][0]	a[2][1]	a[2][2]	a[2][3]

Thus, every element in the array **a** is identified by an element name of the form **a[i][j]**, where 'a' is the name of the array, and 'i' and 'j' are the subscripts that uniquely identify each element in 'a'.

Initializing Two-Dimensional Arrays

Multidimensional arrays may be initialized by specifying bracketed values for each row. Following is an array with 3 rows and each row has 4 columns.

```
int a[3][4] = {
   {0, 1, 2, 3} ,   /*  initializers for row indexed by 0 */
   {4, 5, 6, 7} ,   /*  initializers for row indexed by 1 */
   {8, 9, 10, 11}   /*  initializers for row indexed by 2 */
};
```

The nested braces, which indicate the intended row, are optional. The following initialization is equivalent to the previous example −

```
int a[3][4] = {0,1,2,3,4,5,6,7,8,9,10,11};
```

Accessing Two-Dimensional Array Elements

An element in a two-dimensional array is accessed by using the subscripts, i.e., row index and column index of the array. For example −

```
int val = a[2][3];
```

The above statement will take the 4th element from the 3rd row of the array. You can verify it in the above figure. Let us check the following program where we have used a nested loop to handle a two-dimensional array −

```
#include <stdio.h>

int main () {

   /* an array with 5 rows and 2 columns*/
   int a[5][2] = { {0,0}, {1,2}, {2,4}, {3,6},{4,8}};
   int i, j;

   /* output each array element's value */
   for ( i = 0; i < 5; i++ ) {

      for ( j = 0; j < 2; j++ ) {
         printf("a[%d][%d] = %d\n", i,j, a[i][j] );
      }
   }

   return 0;
}
```

When the above code is compiled and executed, it produces the following result −

```
a[0][0]: 0
a[0][1]: 0
a[1][0]: 1
a[1][1]: 2
a[2][0]: 2
a[2][1]: 4
a[3][0]: 3
```

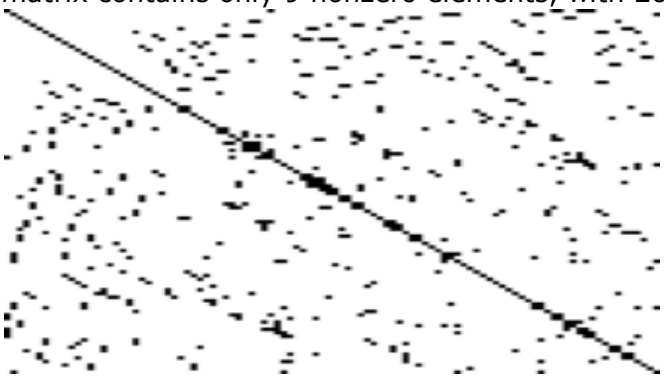
a[3][1]: 6
a[4][0]: 4
a[4][1]: 8
As explained above, you can have arrays with any number of dimensions, although it is likely that most of the arrays you create will be of one or two dimensions.

1.4.2 Sparse matrix

Example of sparse matrix

$$\begin{pmatrix} 11 & 22 & 0 & 0 & 0 & 0 & 0 \\ 0 & 33 & 44 & 0 & 0 & 0 & 0 \\ 0 & 0 & 55 & 66 & 77 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 88 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 99 \end{pmatrix}$$

The above sparse matrix contains only 9 nonzero elements, with 26 zero elements.



A sparse matrix obtained when solving a finite element problem in two dimensions. The non-zero elements are shown in black.

In numerical analysis, a sparse matrix is a matrix in which most of the elements are zero. By contrast, if most of the elements are nonzero, then the matrix is considered **dense**. The fraction of non-zero elements over the total number of elements (i.e., that can fit into the matrix, say a matrix of dimension of m x n can accommodate m x n total number of elements) in a matrix is called the **sparsity** (**density**).

1.4.3 Three Dimensional and n Dimensional Arrays

A 3-D array can be considered as an array of a 2-Dimensional array and thats very simple to understand. Consider an example ,say we have an array declaration int A[2][3][2],thats a 3-D array.We need to develop a decoding scheme to interpret the meaning of such declarations.

Read it like this: int (A[2])[3][2] (Note the braces) .'A' is an array of 2 elements and these 2 elements are not simple integers as it would have been in the case if 'A' was a 1-D array.Here ,the two elements that 'A' is made up of a 2-D array of dimensions 3*2 .i.e A 3-D array in this case is nothing but a collection of two 2-D arrays in our example.

in other words ,we say that 'A' is made of 2 pages ,3 rows and 2 columns each,in all we have total=2*3*2=12 elements in our array

but this is just a logical explanation ,physically an array be it any dimensional ,is always stored sequentially as memory is Linear.

Initialization Of three-dimensional Array

Double program[3][2][4]={
{{-0.1, 0.22, 0.3, 4.3}, {2.3, 4.7, -0.9, 2}},

{{0.9, 3.6, 4.5, 4}, {1.2, 2.4, 0.22, -1}},
{{8.2, 3.12, 34.2, 0.1}, {2.1, 3.2, 4.3, -2.0}}
};

Suppose there is a multidimensional array arr[i][j][k][m]. Then this array can hold i*j*k*m numbers of data.

Similarly, the array of any dimension can be initialized in C programming

1.5 POINTER ARRAYS

The address of a memory variable is or array is known as pointer and an array containing as its elements is known as pointer array.

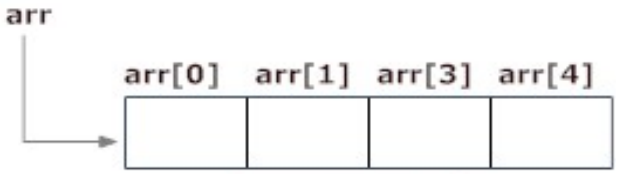
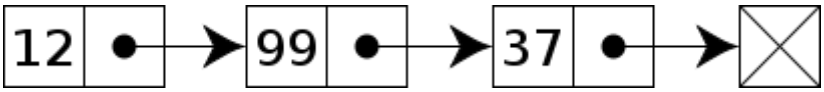


Figure: Array as Pointer

1.6 LINKED LIST

A *linked list* is an ordered collection of finite. homogeneous data elements called *nodes* where the linear order is maintained by means of links or pointers.

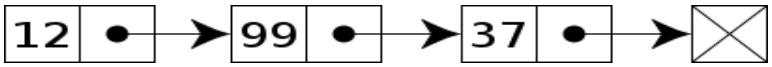
a **linked list** is a data structure consisting of a group of nodes which together represent a sequence. Under the simplest form, each node is composed of data and a reference (in other words, a *link*) to the next node in the sequence; more complex variants add additional links. This structure allows for efficient insertion or removal of elements from any position in the sequence.



1.7 SINGLE LINKED LIST

DEFINITION

Singly linked lists contain nodes which have a data field as well as a 'next' field, which points to the next node in line of nodes. Operations that can be performed on singly linked lists include insertion, deletion and traversal.



1.5.1 Representation of a Linked List

There are two ways to represent a linked list in memory:

1. Static representation using array

2. Dynamic representation using free pool of storage

Linked lists are a way to store data with structures so that the programmer can automatically create a new place to store data whenever necessary. Specifically, the programmer writes a struct definition that contains variables holding information about something and that has a pointer to a struct of its same type (it has to be a pointer--otherwise, every time an element was created, it would create a new element, infinitely). Each of these individual structs or classes in the list is commonly known as a node or element of the list. I.inkffi Lists

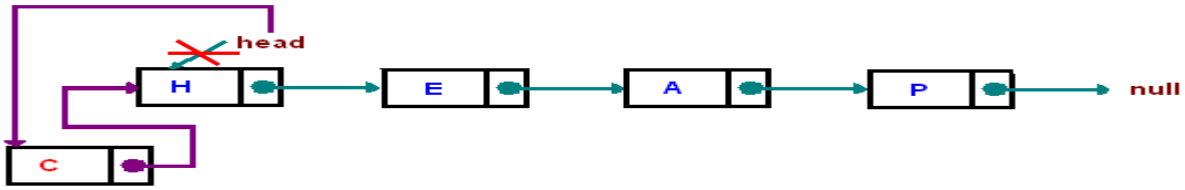
In memory a linked list is often described as looking like this:

```
----------      ----------
- Data   -      - Data   -
----------      ----------
- Pointer- - - -> - Pointer-
----------      ----------
```

Linked List Operations

Add First

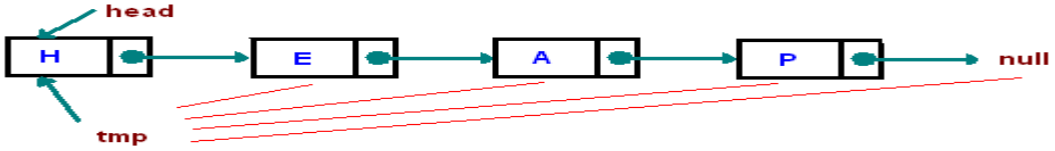
The method creates a node and prepends it at the beginning of the list.




```
public void addFirst(AnyType item)
{
  head = new Node<AnyType>(item, head);
}
```

Traversing

Start with the head and access each node until you reach null. Do not change the head reference.

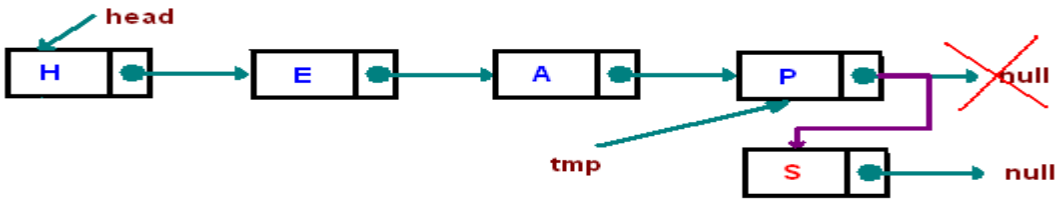



```
Node tmp = head;

while(tmp != null) tmp = tmp.next;
```

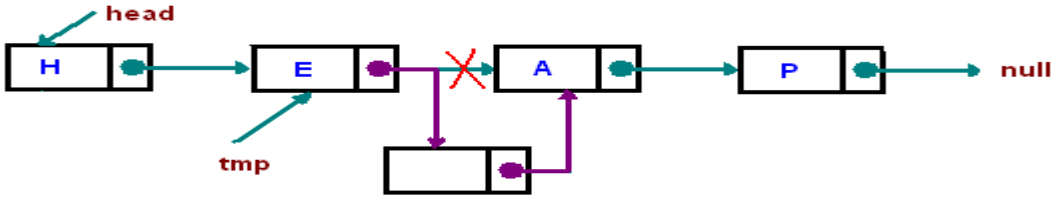
add Last

The method appends the node to the end of the list. This requires traversing, but make sure you stop at the last node



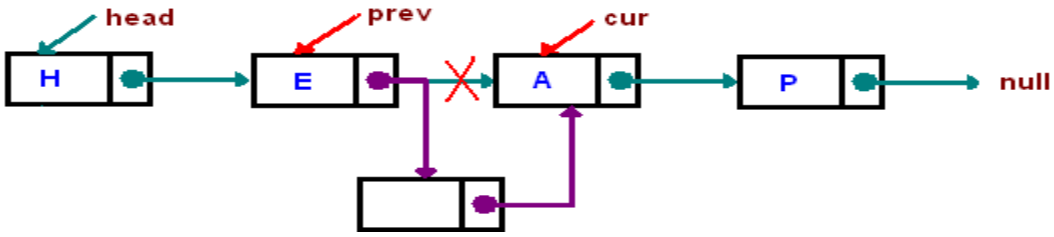
Inserting "after"

Find a node containing "key" and insert a new node after it. In the picture below, we insert a new node after "e":



Inserting "before"

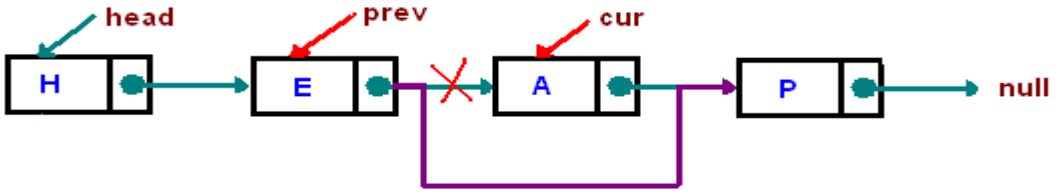
Find a node containing "key" and insert a new node before that node. In the picture below, we insert a new node before "a":



For the sake of convenience, we maintain two references prev and cur. When we move along the list we shift these two references, keeping prev one step before cur. We continue until cur reaches the node before which we need to make an insertion. If cur reaches null, we don't insert, otherwise we insert a new node between prev and cur.

Deletion

Find a node containing "key" and delete it. In the picture below we delete a node containing "A"



The algorithm is similar to insert "before" algorithm. It is convinient to use two references prev and cur. When we move along the list we shift these two references, keeping prev one step before cur. We continue until cur reaches the node which we need to delete. There are three exceptional cases, we need to take care of:

1. list is empty
2. delete the head node
3. node is not in the list

Iterator

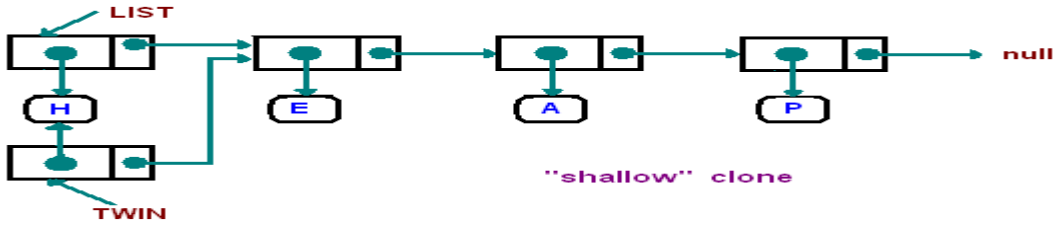
The whole idea of the iterator is to provide an access to a private aggregated data and at the same moment hiding the underlying representation. An iterator is Java is an object, and therefore it's implementation requires creating a class that implements the *Iterator* interface. Usually such class is implemented as a private inner class. The *Iterator* interface contains the following methods:

- AnyType next() - returns the next element in the container
- boolean hasNext() - checks if there is a next element
- void remove() - (optional operation).removes the element returned by next()

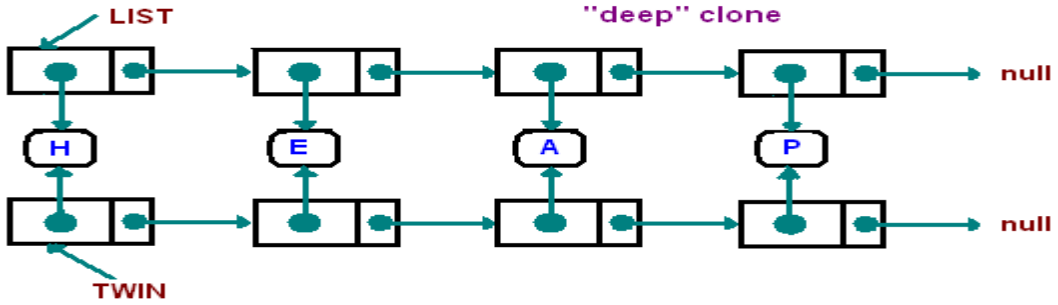
In this section we implement the Iterator in the LinkedList class. First of all we add a new method to the LinkedList class:

Cloning

Like for any other objects, we need to learn how to clone linked lists. If we simply use the clone() method from the Object class, we will get the following structure called a "shallow" copy:



The Object's clone() will create a copy of the first node, and share the rest. This is not exactly what we mean by "a copy of the object". What we actually want is a copy represented by the picture below

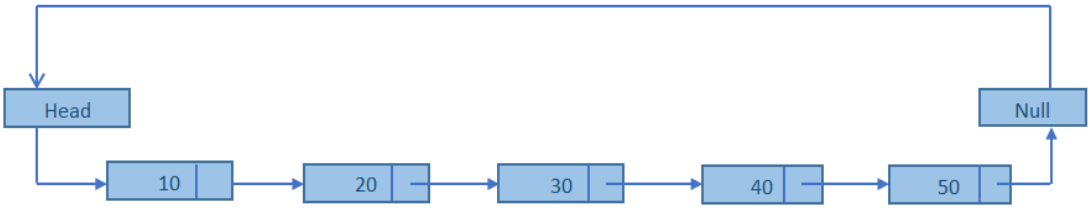


Since out data is immutable it's ok to have data shared between two linked lists. There are a few ideas to implement linked list copying. The simplest one is to traverse the original list and copy each node by using the addFirst() method. When this is finished, you will have a new list in the reverse order. Finally, we will have to reverse the list:

A better way involves using a tail reference for the new list, adding each new node after the last node.

1.8 CIRCULAR LINKED LIST

Circular list is a list in which the link field of the last node is made to point to the start/first node of the list.



2 **1.**A circular linked list is a linked list in which the head element's previous pointer points to the tail element and the tail element's next pointer points to the head element.
2.A circularly linked list node looks exactly the same as a linear singly linked list.

A circularly linked list, or simply circular list, is a linked list in which the last node is always points to the first node. This type of list can be build just by replacing the NULL pointer at the end of the list with a pointer which points to the first node. There is no first or last node in the circular list.

Advantages:

• Any node can be traversed starting from any other node in the list.
• There is no need of NULL pointer to signal the end of the list and hence, all pointers contain valid addresses.
• In contrast to singly linked list, deletion operation in circular list is simplified as the search for the previous node of an element to be deleted can be started from that item itself.

Basic Operations on a Circular Linked List

Insert – Inserts a new element at the end of the list.
Delete – Deletes any node from the list.
Find – Finds any node in the list.
Print – Prints the list.
Algorithm:
The node of a linked list is a structure with fields data (which stored the value of the node) and

*next (which is a pointer of type node that stores the address of the next node).

Two nodes *start (which always points to the first node of the linked list) and *temp (which is

used to point to the last node of the linked list) are initialized. Initially temp = start and temp->next = start. Here, we take the first node as a dummy node. The first node does not contain data, but it used because to avoid handling special cases in insert and delete functions.

Functions –

1. Insert – This function takes the start node and data to be inserted as arguments. New node is inserted at the end so, iterate through the list till we encounter the last node. Then, allocate memory for the new node and put data in it. Lastly, store the address of the first node (start) in the next field of the new node.

2. Delete - This function takes the start node (as pointer) and data to be deleted as arguments. Firstly, go to the node for which the node next to it has to be deleted, If that node points to NULL (i.e. pointer->next=NULL) then the element to be deleted is not present in the list. Else, now pointer points to a node and the node next to it has to be removed, declare a temporary node (temp) which points to the node which has to be removed. Store the address of the node next to the temporary node in the next field of the node pointer (pointer->next = temp->next). Thus, by breaking the link we removed the node which is next to the pointer (which is also temp). Because we deleted the node, we no longer require the memory used for it, free() will deallocate the memory.

3. Find - This function takes the start node (as pointer) and data value of the node (key) to be found as arguments. A pointer start of type node is declared, which points to the head node of the list (node *start = pointer). First node is dummy node so, start with the second node. Iterate through the entire linked list and search for the key. Until next field of the pointer is equal to start, check if pointer->data = key. If it is
then the key is found else, move to the next node and search (pointer = pointer -> next). If key is not found return 0, else return 1.

4.Print - function takes the start node (as start) and the next node (as pointer) as arguments. If pointer = start, then there is no element in the list. Else, print the data value of the node (pointer->data) and move to the next node by recursively calling the print function with pointer->next sent as an argument.

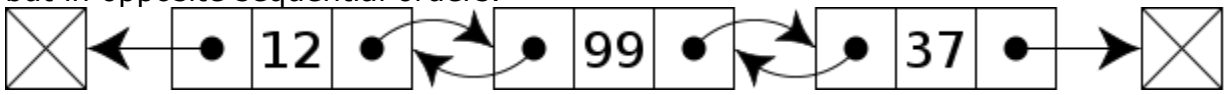
The functions to insert and delete elements to/from the circular list can be written as follows:

```
int insert(CLinkedList *clist, int data)
{
struct Node *node, *tempnode;
node = (struct Node*) malloc(sizeof(struct Node));
node->dataw=data;
node->next=NULL;
if(!clist->Head)
{
clist->Head=node;
node->next=Head;
return 1;
}
```

```
tempnode=slist->Head;
while(tempnode->next!=Head)
tempnode=tempnode->next;
tempnode->next=Head;
return 1;
}
int delete(CLinkedList *clist, int nodeindx)
{
int i=1;
struct Node *tempnode, *prevnode, *nextnode;
if(!clist->Head)
return 0;
tempnode=slist->Head;
while(tempnode->next!=Head && i<nodeindx)
{
tempnode=tempnode->next;
i++;
}
if(i==nodeindx)
{
free((void *)tempnode);
return 1;
}
return 0;
        }
```

1.9 DOUBLE LINKED LIST

A **doubly linked list** is a linked data structure that consists of a set of sequentially linked records called nodes. Each node contains two fields, called *links*, that are references to the previous and to the next node in the sequence of nodes. The beginning and ending nodes' **previous** and **next** links, respectively, point to some kind of terminator, typically a sentinel node or null, to facilitate traversal of the list. If there is only one sentinel node, then the list is circularly linked via the sentinel node. It can be conceptualized as two singly linked lists formed from the same data items, but in opposite sequential orders.



A doubly linked list whose nodes contain three fields: an integer value, the link to the next node, and the link to the previous node.

Doubly-linked list is a more sophisticated form of linked list data structure. Each node of the list contain two references (or links) – one to the previous node and other to the next node. The previous link of the first node and the next link of the last node points to NULL. In comparison to singly-linked list, doubly-linked list requires handling of more pointers but less information is required as one can use the previous links to observe the

preceding element. It has a dynamic size, which can be determined only at run time.
Let's now directly jump over to various operations which can be performed over doubly linked list:

- Add a node in a list at beginning or at end or in between.
- Delete a node from list at specific location.
- Reverse a list.
- Count nodes present in the list.
- Print the list to see all the nodes present in the list.

First let's have a look at the node definition :

```
struct node{
   int data;
   struct node *prev;
   struct node *next;
};
```

It's evident from the above node definition that a node in a doubly linked list have 2 links (***next*** and ***prev***) and one data value (***data***).

- **Add a node in a list.**
  Insertion of a node in a linked list can be done at three places, viz. at start, in between at a specified location or at end.
  - Inserting a node at the start of list :
    Algorithm :
    1. Update the ***next*** pointer of the new node to the head node and make ***prev*** pointer of the new node as NULL
    2. Now update head node's ***prev*** pointer to point to new node and make new node as head node.

Inserting a node in between of list:

Algorithm :

1. Traverse the list to the position where the new node is to be inserted. Let's call this node as Position Node (we have to insert new node just next to it).
2. Make the ***next*** pointer of new pointer to point to next node of position node. Also make the ***prev*** point of new node to point to position node.
3. Now point position node's ***next*** pointer to new node and ***prev*** node of next node of position node to point to new node.

  - Inserting a node at the end of the list :
    Algorithm :
    1. Traverse the list to end. Let's call the current last node of list as Last node.
    2. Make ***next*** pointer of New node to point to NULL and ***prev*** pointer of new node to point to Last node.
    3. Update next pointer of Last node to point to new Node.

  Thus we see how easily we can add a node in a list. Let us now write code for all the three cases. Please note here that we are passing double pointer in the function as we may require to change the head pointer.

Insertion A node can be added in four ways 1) At the front of the DLL 2) After a given node. 3) At the end of the DLL 4) Before a given node.

1.Insertion at the front of list
2.Insertion in the middle of the list
3.Insertion at the end of the list

Deletion
Write a function to delete a given node in a doubly linked list.
(a) Original Doubly Linked List
(b) After deletion of head node
(c) After deletion of middle node

After deletion of last node

ALGORITHM :

Initialize the first and last nodes with NULL values
struct node *first=NULL,*last=NULL,*next,*prev,*cur;

1. Algorithm creating a new node:

Step 1: if the list is empty then first==NULL
Step 2: Create a new node
cur=(struct node*) malloc (sizeof (struct node));
Step 3: Read the content of node
Step 4: Assign new node left and right links to NULL
cur->left=NULL;
cur->right=NULL;
Step 5: Assign new node to first & last node
first=cur
last=cur
Step 6: If the list is not empty call insert function
insert ()
Step 7 : Stop

2. Algorithm for Inserting a new node:

Step 1 : Initialize count c to 1
Step 2 : Create inserting node
cur=(struct node*)malloc(sizeof (struct node));
Step 3: Read the content of node
Step 4: Read the position of insertion
Step 5: Inserting in first position
Check if the pos=1 and first!=NULL
cur->right=first;
cur->left=NULL;
first=cur;
Step 6: Inserting in a given position
next=first;

repeat the steps a to c until c < pos
• prev=next;
• next=prev->right;
• c++;
prev->right=cur;
cur->right=next;
Step 7 : Stop

3. Algorithm for Deleting a node:

Step 1 : Initialize count c to 1
Step 2 : Read the position for deletion
Step 3 : Check if first=NULL
print list is empty
Step 4 : If the list contains single element
Check if pos=1 and first->right=NULL
print deleted element is first->data
Step 5 : Assign first to NULL
first=NULL;
Step 6 : If the list contains more than one element and to delete first element
if pos=1 and first->right!=NULL
cur=first;
first=first->right;
cur->right=NULL;
print deleted element is cur->data
free(cur);
Step 7 : If the list contains more than one element and to delete an element at given position
next=first;
repeat the steps a to c until c < pos
• cur=next;
• next=next->right;
• c++;
cur->right=next->right;
next->right=NULL;
next->left=NULL;
print deleted element is next->data
free(next);
Step 8 : Stop

4. Algorithm for Displaying a node:

Step1 : Check if first node is NULL
print list is empty
Step2: If first node is not NULL then

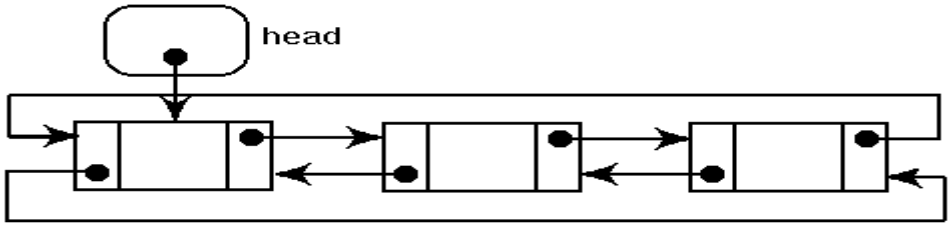
cur=first;
repeat the steps a to b until cur!=NULL
a . print cur->data
b . cur=cur->right;
Step3 : Stop

Creation of a node:

```
Struct DList
{
int data;
struct DList *Lptr;
struct DList *Rptr;
};
```

1.10 CIRCULAR DOUBLE LINKED LIST

Doubly Circular linked list has both the properties of doubly linked list and circular linked list. Two consecutive elements are linked by previous and next pointer and the last node points to first node by next pointer and also the previous pointer of the head node points to the tail node.
Node traversal from any direction is possible and also jumping from head to tail or from tail to head is only one operation: head pointer previous is tail and also tail pointer next is head.



Doubly Linked Circular list

1.11 APPLICATIONS OF LINKED LIST

The main Applications of Linked Lists are

It means in addition/subtraction /multipication.. of two polynimials.
Eg:p1=2x^2+3x+7 and p2=3x^3+5x+2
p1+p2=3x^3+2x^2+8x+9

* In Dynamic Memory Management In allocation and releasing memory at runtime.

*In Symbol Tables in Balancing paranthesis

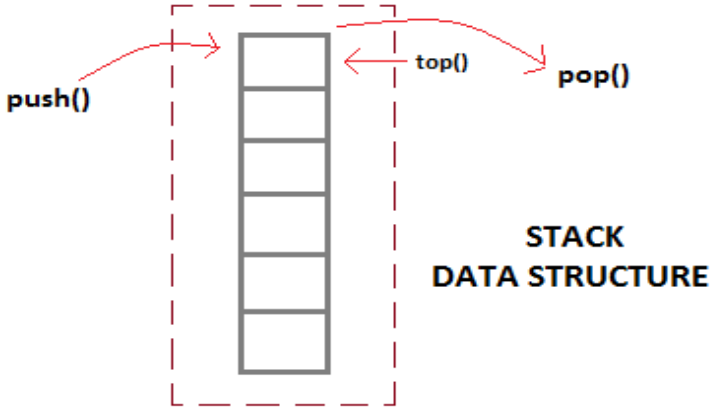
* Representing Sparse Matrix

UNIT – II

STACKS:

2.1 Introduction:

Stack is an abstract data type with a bounded(predefined) capacity. It is a simple data structure that allows adding and removing elements in a particular order. A *Stack* is a linear data structure and very much useful in various applications of computer science Every time an element is added, it goes on the top of the stack, the only element that can be removed is the element that was at the top of the stack, just like a pile of objects.



A stack is a list in which insertions and deletions occur at only one end called top. The fundamental operations performed on stack are push and pop which inserts an item at top of the stack and removes the element at top of the stack respectively.

Basic features of Stack

1. Stack is an ordered list of similar data type.
2. Stack is a **LIFO** structure. (Last in First out).
3. **push()** function is used to insert new elements into the Stack and **pop()** is used to delete an element from the stack. Both insertion and deletion are allowed at only one end of Stack called **Top**.
4. Stack is said to be in **Overflow** state when it is completely full and is said to be in **Underflow** state if it is completely empty.

2.2 Representation of Stack:

A stack may be represented in the memory in various ways. There are two main ways: using a one-dimensional array and a single linked list. Representations of stacks in a memory are discussed in the following two sections.

2.2. Array Representation of Stacks

First we have to allocate a memory block of sufficient size to accommodate the full capacity of the stack. Then, starting from the first location of the memory block, the items of the stack can be stored in a sequential fashion,

ITEM; denotes the *ith* item in the stack; I and *u* denote the index range of the array in use; usually the values of these indices are I and SIZE respectively. TOP is a pointer to point the position of the array up to which it is filled with the items of the stack. With this representation. the following two ways can be stated:

2.2.2 Linked List Representation of Stacks

Although array representation of stacks is very easy and convenient but it allows the representation of only fixed sized stacks. In several applications. the size of the stack may vary during program execution. An obvious solution to this problem is to represent a slack using a linked list A single linked list structure is sufficient to represent any stack. Here, the DATA field is for the ITEM. and the LINK field is. as usual. to point 10 the next item. Figure 4.3(b) depicts such a stack using a single linked list.

In the linked list representation. the first node on the list is the current item that is the item at the top of the stack and the last node is the node containing the bottom- most item. Thus, a PUSH operation will add a new node in the front and a POP operation will remove a node from the front of the list. The SIZE of the stack is not important here because this representation allows dynamic stacks instead of static stacks. as with arrays.

2.3 Operations on Stack

push operation: It takes O(1) time to carry out insertion at top of the stack. Stack have size n. It allows elements into stack unless top=n(stack size).
overflow occurs in stack, when inserting an item if stack is full. otherwise,insert new element into the stack.

```
void push(int x)
{
if(top==n)
 cout<<"stack overflows";
else
{
top++;
stk[top]=x;
cout<<"element inserted is:"<<stk[top];
}
```

remove operation; It also takes O(1) time to remove an element from top of the stack. In this, check if top<0,stack is empty means there are no items in stack. otherwise remove topmost element from the stack.

```
 void remove()
{
if(top<0)
```

```
 cout<<"stack underflows";
else
{
top--;
x=stk[top];
cout<<"deleted element is"<<x;
}
}
```

display operation: It takes O(n) time to display all elements of stack whose size is n. The stacck displays from topmost element to element at top=0.

```
void display()
{
is(top<0)
cout<<"stack is empty";
for(int i=top;i>=0;t--)
cout<<stk[i]<<"\t";
}
```

2.4 Applications of Stack

There are several areas which uses stack. Among them, the most important areas are:

a. Balancing symbols
b. (i) Evaluating the postfix expression
   (ii) Converting infix into postfix
c. Function calls

(a) Balancing symbols: The compiler checks syntax errors for a program and causes many errors when any symbol is missed to place. A useful tool called stack used to check everything is balanced in an expression such as left brace, bracket and parenthesis have corresponding right counter parts. An algorithm designed to check the symbols in an expression and it ignores characters if any in it while reading. Make stack empty. If symbol read is opening symbol, push it in the stack. if symbol read is closing symbol, if the stack is empty, report an error.otherwise,pop the stack. If the popped symbol is not corresponding opening symbol, report error. When end of the expression is reached,if the stack is not empty, report en error.

(b) (i) Evaluating postfix expression**:** Consider the postfix expression 6 5 3 2 + * +8+ 3*. The stack is used to evaluate the postfix expression.

ALGORITHM:

while reading the postfix expression,

(i) if symbol read is an operand, insert it into the stack.

(i) if the symbol read is operator,pop the two topmost operands from the stack and apply the operator on them,then store the result at top of the stack.

Procedure for evaluating the following postfix expression:

step 1: start reading the postfix expression,first symbol read is an operand,push it into the stack

6

step 2: when reading nex **stack with 6** also an operand, insert the operand at top of the stack like

5
6

stack with 6 & 5

3
5
6

step 3: The next symbol read is again operand, now stack has

step 4: The next symbol read is operand, store it at top of stack

2
3
5
6

5
5
6

step 5: The next read is +, it is operator, pop two topmost items from stack such as 2 & 3,apply + on them, the result is 5 that stored on top of stack like below.

step 6: The next symbol used is *,pop two topmost operands from the stack, apply * to then and store result at top of the stack.

25
6

step 7: The next symbol read is +,pop two operands from that stack and apply it.

31

step 8: The next symbol read is an operand(8),push it into the stack, now stack has

8
31

39

step 9: next symbol in the postfix expression is +,pop two top items and apply + on them.

3
39

step 10: next read in postfix is 3,push it into stack, now stack contains

step 11: The last symbol read is *,pop two top items and apply +,then insert result into stack top.

118

(ii) Converting infix to postfix expression:

The stack is also used in converting standard form infix into postfix expression.

Consider infix form a+b*c+(d*e+f)*g,it is converted into postfix abc*+def+*g* +.

The conversion takes place using the following steps.

(i) Make stack empty and empty output.
start reading the infix expression, if the character read is operand, append to output.

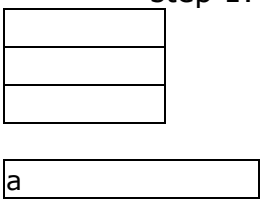
(ii) if character read is operator, push it in to the stack. If recently read operator has lower precedence than the operator on top of the stack, pop the operators until a lower precedence operator is encountered.

(iii) if opening symbol is read, push it into the stack. while reading if closing counterpart is encountered, pop stack until its matched opening is encountered.

(iv) if end of the expression is reached, pop all items from the stack.

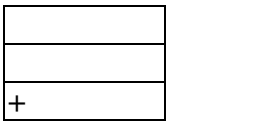
For a+b*c+(d*e+f)*g,

step 1: The very character read is operand,append to the output.

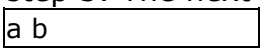


output

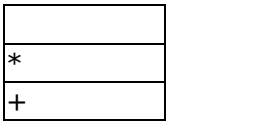
step 2: The next chara(stack s +,insert into stack.

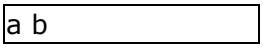


step 3: The next read is b, append to output

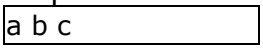


step 4: The next read is *,push it into the stack.

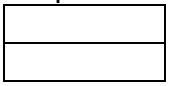


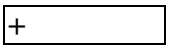


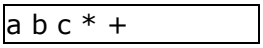
step 5: The next read is c, append to output.



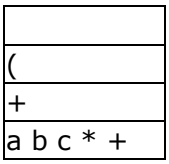
step 6: next character read is+,pop operators from top of the stack until a lower precedence operator is encountered in the stack, then append to the output.



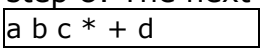




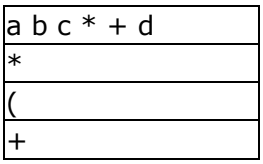
step 7: next read is (,then insert into the stack.

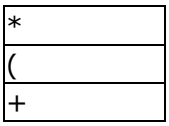


step 8: The next character read is d, append to the output

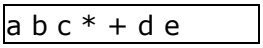


step 9: The next read is *,push it into the stack.

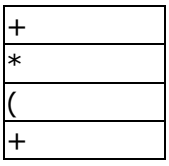




step 10: The next read is e, append to the output



step 11: The next character read is +,push it into the stack.



a b c * + d e

step 12: The next character read is f, append to the output

a b c * + d e f

step 13: The next read is),pop the operators from the stack until its opening symbol is found in the stack.

+

a b c * + d e f + *

step 14: The next read is *,push it into the stack

*
+
a b c * + d e f + *

step 15: The last character read is g, append to the output. since it the last character, pop all items from the stack, then append to the output.

a b c * + d e f + * g * +

(c) Function calls: When a call made to new function in calling routine, all local variables and return address are saved in stack. Generally, the variables are stored in registers and return address in stack. Then, control transferred to called routine. It now freely replaces the registers for its variables. When another function is called again, it follows the same procedure means saving

the function local variables in registers and its return address in stack. After function executes, called routine sees the stack and retrieves return address and restores all register values. This type of saving is called activation record or stack frame. The stack grows when recursion function is used. The system doesn't have any facility for checking overflow. The overflow occurs because of too many simultaneously activation records. When a container has 20,000 elements to print, definitely the stack is running out of space. This can be eliminated by using iteration methods.

QUEUES:

2.5 Introduction:

Queue is a linear data structure in which data can be added to one end and retrieved from the other. Just like the queue of the real world, the data that goes first into the queue is the first one to be retrieved. That is why queues are sometimes called as **First-In-First-Out** data structure.

In case of queues, we saw that data is inserted both from one end but in case of Queues; data is added to one end (known as REAR) and retrieved from the other end (known as FRONT).

The data first added is the first one to be retrieved while in case of queues the data last added is the first one to be retrieved.

A few points regarding Queues:

1. **Queues:** It is a linear data structure; linked lists and arrays can represent it. Although representing queues with arrays have its shortcomings but due to simplicity, we will be representing queues with arrays in this article.
2. **Rear:** A variable stores the index number in the array at which the new data will be added (in the queue).
3. **Front:** It is a variable storing the index number in the array where the data will be retrieved.

2.6 Definition and Representation of Queue:

In computer technology, a queue is a sequence of work objects that are waiting to be processed. A queue is a container of objects (a linear collection) that are inserted and removed according to the first-in first-out (FIFO) principle.

Queue is also an abstract data type or a linear data structure, in which the first element is inserted from one end called **REAR**(also called tail), and the deletion of exisiting element takes place from the other end called as **FRONT**(also called head). This makes queue as FIFO data structure, which means that element inserted first will also be removed first.

The process to add an element into queue is called **Enqueue** and the process of removal of an element from queue is called **Dequeue**.

enqueue( ) is the operation for adding an element into Queue.
dequeue( ) is the operation for removing an element from Queue .

QUEUE DATA STRUCTURE

Basic features of Queue

1. Like Stack, Queue is also an ordered list of elements of similar data types.
2. Queue is a FIFO(First in First Out) structure.
3. Once a new element is inserted into the Queue, all the elements inserted before the new element in the queue must be removed, to remove the new element.
4. **peek( )** function is oftenly used to return the value of first element without dequeuing it.

Queue Operations

Add an Element to Queue:

```
void addq(element item)
{
/* add an item to the queue */
if (rear == MAX_QUEUE_SIZE-1)
queueFull( );
queue [++rear] = item;
}
```

Delete from a Queue:

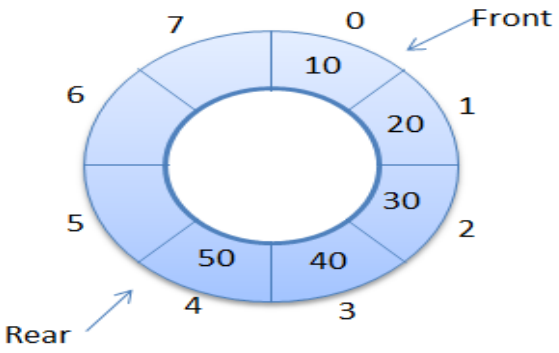
```
Element deleteq()
{
/* remove element at the front of the queue */
if (front == rear)
return queueEmpty( ); /* return an error key */
return queue [++front];
}
```

2.7 Various Queue Structures

- Normal queue (FIFO)
- Circular Queue (Normal Queue)
- Double-ended Queue (Deque)
- Priority Queue

CIRCULAR QUEUE:

A circular queue is an abstract data type that contains a collection of data which allows addition of data at the end of the queue and removal of data at the beginning of the queue. Circular queues have a fixed size. Circular queue follows FIFO principle. Queue items are added at the rear end and the items are deleted at front end of the circular queue.



In a standard queue data structure re-buffering problem occurs for each dequeue operation. To solve this problem by joining the front and rear ends of a queue to make the queue as a circular queue Circular queue is a linear data structure. It follows FIFO principle.

- In circular queue the last node is connected back to the first node to make a circle.
- Circular linked list fallow the First In First Out principle
- Elements are added at the rear end and the elements are deleted at front end of the queue
- Both the front and the rear pointers points to the beginning of the array.
- It is also called as "Ring buffer".
- Items can inserted and deleted from a queue in O(1) time.
- Circular Queue can be created in three ways they are Using single linked list Using doublelinked list Using arrays

Algorithms for Insert and Delete Operations in Circular Queue

For Insert Operation

Insert-Circular-Q(CQueue, Rear, Front, N, Item)

➢CQueue is a circular queue where to store data.

➢Rear represents the location in which the data element is to be inserted and Front represents the location from which the data element is to be removed.

➢Here N is the maximum size of CQueue and finally, Item is the new item to be added.

➢Initailly Rear = 0 and Front = 0.

1. If Front = 0 and Rear = 0 then Set Front := 1 and go to step 4.
2. If Front =1 and Rear = N or Front = Rear + 1
   then Print: "Circular Queue Overflow" and Return.
3. If Rear = N then Set Rear := 1 and go to step 5.
4. Set Rear := Rear + 1
5. Set CQueue [Rear] := Item.
6. Return

For Delete Operation

Delete-Circular-Q(CQueue, Front, Rear, Item)

Here, CQueue is the place where data are stored. Rear represents the location in which the data element is to be inserted and Front represents the location from which the data element is to be removed. Front element is assigned to Item. Initially, Front = 1.

1. If Front = 0 then
   Print: "Circular Queue Underflow" and Return. /*..Delete without Insertion
2. Set Item := CQueue [Front]
3. If Front = N then Set Front = 1 and Return.
4. If Front = Rear then Set Front = 0 and Rear = 0 and Return.
5. Set Front := Front + 1
6. Return.

5

- **Double-ended Queue (Deque)**
  A double-ended queue (dequeue, often abbreviated to deque, pronounced *deck*) is an abstract data type that generalizes a queue, for which elements can be added to or removed from either the front (head) or back (tail).[1] It is also often called a head-tail linked list, though properly this refers to a specific data structure implementation

A double-ended queue is an abstract data type similar to an simple queue, it allows you to insert and delete from both sides means items can be added or deleted from the front or rear end.

Algorithm for Insertion at rear end

```
Step -1: [Check for overflow]
  if(rear==MAX)
        Print("Queue is Overflow");
    return;
Step-2: [Insert element]
  else
        rear=rear+1;
        q[rear]=no;
        [Set rear and front pointer]
        if rear=0
              rear=1;
        if front=0
```

13. front=1;

Algorithm for Insertion at font end

1. Step-1 : [Check for the front position]
2. if(front<=1)
3. Print (“Cannot add item at front end”);
4. return;
5. Step-2 : [Insert at front]
6. else
7. front=front-1;
8. q[front]=no;
9. Step-3 : Return

Algorithm for Deletion from front end

1. Step-1 [Check for front pointer]
2. if front=0
3. print(" Queue is Underflow”);
4. return;
5. Step-2 [Perform deletion]
6. else
7. no=q[front];
8. print(“Deleted element is”,no);
9. [Set front and rear pointer]
10. if front=rear
11. front=0;
12. rear=0;
13. else
14. front=front+1;
15. Step-3 : Return

Algorithm for Deletion from rear end

1. Step-1 : [Check for the rear pointer]
2. if rear=0
3. print(“Cannot delete value at rear end”);
4. return;
5. Step-2: [perform deletion]
6. else
7. no=q[rear];
8. [Check for the front and rear pointer]
9. if front= rear
10. front=0;
11. rear=0;
12. else
13. rear=rear-1;
14. print(“Deleted element is”,no);
15. Step-3 : Return

Priority Queue

A priority queue is an abstract data type which is like a regular queue or stack data structure, but where additionally each element has a "priority" associated with it. In a priority queue, an element with high priority

is served before an element with low priority. If two elements have the same priority, they are served according to their order in the queue.

A priority queue must at least support the following operations:

- *insert_with_priority*: add an element to the queue with an associated priority.
- *pull_highest_priority_element*: remove the element from the queue that has the *highest priority*, and return it.
  This is also known as "*pop_element(Off)*", "*get_maximum_element*" or "*get_front(most)_element*".
  Some conventions reverse the order of priorities, considering lower values to be higher priority, so this may also be known as "*get_minimum_element*", and is often referred to as "*get-min*" in the literature.
  This may instead be specified as separate "*peek_at_highest_priority_element*" and "*delete_element*" functions, which can be combined to produce "*pull_highest_priority_element*".

2.8 Applications of Queue

Application Queues

Queues that are generated, or created, by an application or the Message Queuing administrator are often referred to as application queues. Depending on the service provided by the queue, application queues can be public or private, and they can be transactional or nontransactional (for example, administration queues cannot be transactional).
Application queues may appear in the following roles:

- Destination queues—any queue that the sending application sends messages to or that the receiving application reads messages from.
- Administration queues—queues used for *acknowledgment messages* returned by Message Queuing or *connector applications*.
- Response queues—queues used by receiving applications to return *response messages* to the sending application.
- Report queues—queues used to store *report messages* returned by Message Queuing.

TABLES:

2.9 Hash tables

A **hash table** (**hash map**) is a data structure used to implement an associative array, a structure that can map keys to values. A hash table uses a hash function to compute an *index* into an array of *buckets* or *slots*, from which the desired value can be found.

The idea of hashing is to distribute the entries (key/value pairs) across an array of *buckets*. Given a key, the algorithm computes an *index* that suggests where the entry can be found:
index = f(key, array_size)
Often this is done in two steps:
hash = hashfunc(key)

index = hash % array_size

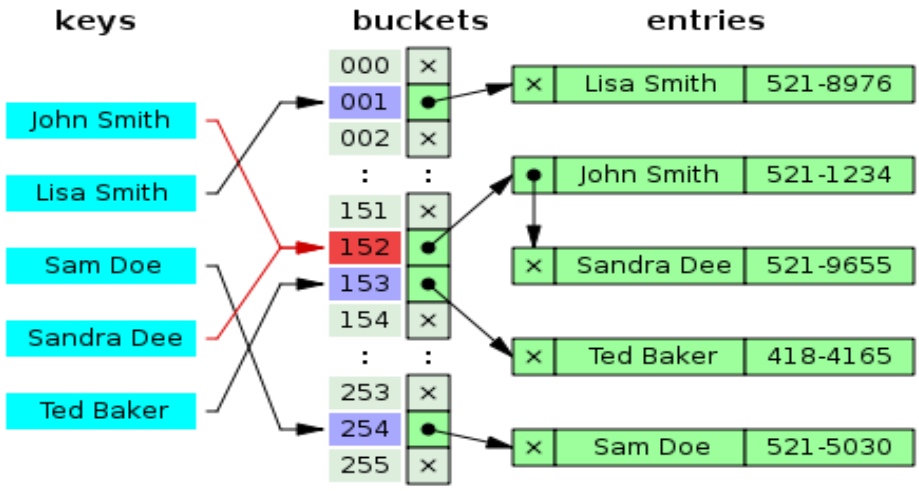
In this method, the *hash* is independent of the array size, and it is then *reduced* to an index (a number between 0 and array_size – 1) using the modulo operator (%).

In the case that the array size is a power of two, the remainder operation is reduced to masking, which improves speed, but can increase problems with a poor hash function.

Collision resolution

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. Therefore, most 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.

Separate chaining

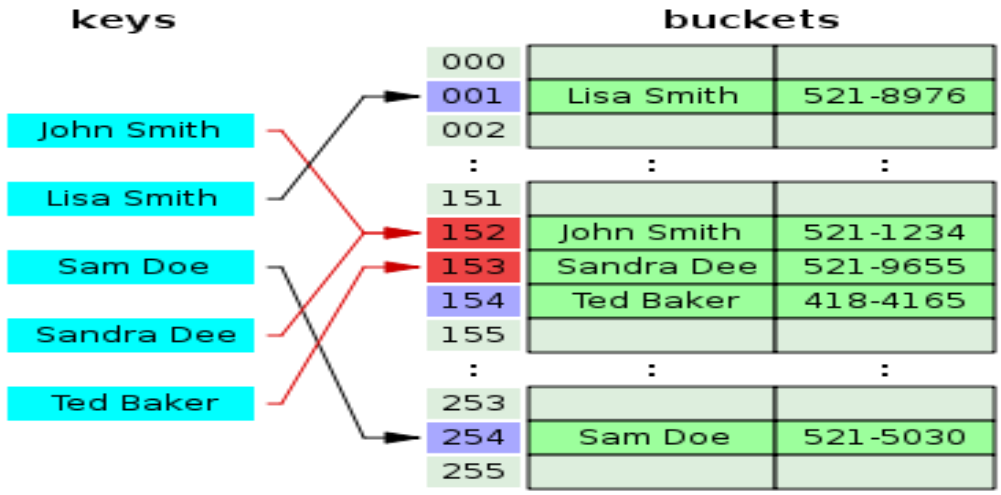


Hash collision resolved by separate chaining.

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 a good hash table, each bucket has zero or one entries, and sometimes two or three, but rarely more than that. 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 is not working well, and this needs to be fixed.

Open addressing



Hash collision resolved by open addressing with linear probing (interval=1). Note that "Ted Baker" has a unique hash, but nevertheless collided with "Sandra Dee", that had previously collided with "John Smith".

In another strategy, called open addressing, all entry records are stored in the bucket array itself. When a new entry has to be inserted, the buckets are examined, starting with the hashed-to slot and proceeding in some *probe sequence*, until an unoccupied slot is found. When searching for an entry, the buckets are scanned in the same sequence, until either the target record is found, or an unused array slot is found, which indicates that there is no such key in the table.[12] The name "open addressing" refers to the fact that the location ("address") of the item is not determined by its hash value. (This method is also called **closed hashing**; it should not be confused with "open hashing" or "closed addressing" that usually mean separate chaining.)

Well-known probe sequences include:

- Linear probing, in which the interval between probes is fixed (usually 1)
- Quadratic probing, in which the interval between probes is increased by adding the successive outputs of a quadratic polynomial to the starting value given by the original hash computation
- Double hashing, in which the interval between probes is computed by another hash function

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 is completely irrelevant.
Open addressing only saves memory if the entries are small (less than four times the size of a pointer) and the load factor is not too small. If the load factor is close to zero (that is, there are far more buckets than stored entries), open addressing is wasteful even if each entry is just two words.

UNIT – III
TREES:

3.1 Basic Terminologies

Terminologies used in Trees

- **Root** – The top node in a tree.
- **Child** – A node directly connected to another node when moving away from the Root.
- **Parent** – The converse notion of a *child*.
- **Siblings** – Nodes with the same parent.
- **Descendant** – A node reachable by repeated proceeding from parent to child.
- **Ancestor** – A node reachable by repeated proceeding from child to parent.
- **Leaf** – A node with no children.
- **Internal node** – A node with at least one child.
- **External node** – A node with no children.
- **Degree** – Number of sub trees of a node.
- **Edge** – Connection between one node to another.
- **Path** – A sequence of nodes and edges connecting a node with a descendant.
- **Level** – The level of a node is defined by 1 + (the number of connections between the node and the root).
- **Height of node** – The height of a node is the number of edges on the longest downward path between that node and a leaf.
- **Height of tree** – The height of a tree is the height of its root node.
- **Depth** – The depth of a node is the number of edges from the node to the tree's root node.
- **Forest** – A forest is a set of n ≥ 0 disjoint trees.

3.2 Definition and Representation

A tree is a (possibly non-linear) data structure made up of nodes or vertices and edges without having any cycle. The tree with no nodes is called the **null** or **empty** tree. A tree that is not empty consists of a root node and potentially many levels of additional nodes that form a hierarchy.

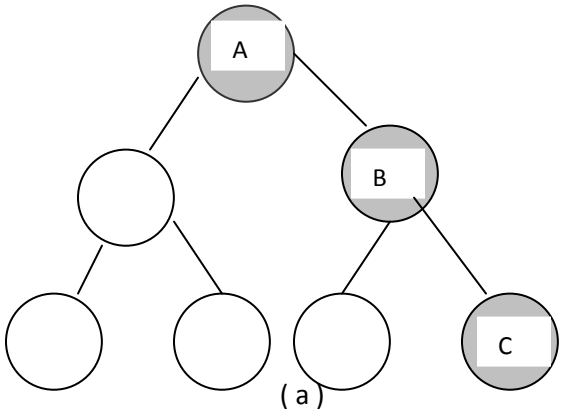
3.3 Representation of Binary Tree

There are two representations used to implement binary trees.

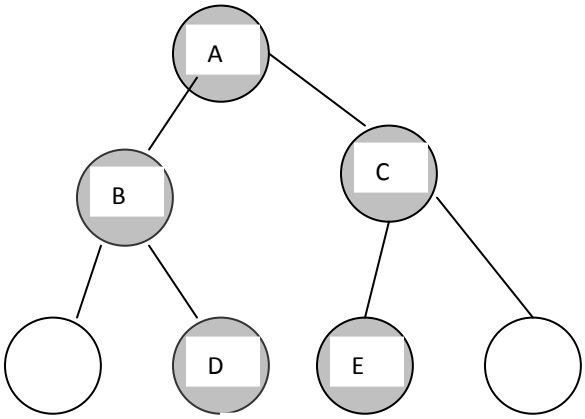
(i) Array Representation
(ii) Linked list Representation

(i) **Array Representation:** In this, the given binary trees even though are not complete binary trees, they are shown as complete binary trees in which missing elements are unshaded circles.

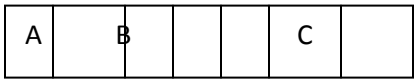
The array representations for the following trees are shown in below.

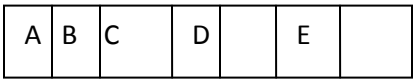


(a)



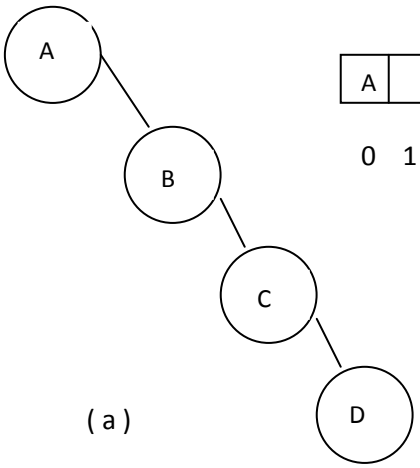
(b)



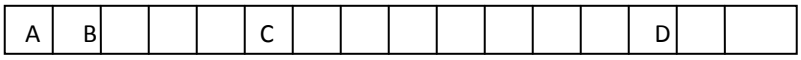


In array, the elements of the binary are placed in the array according to their number assigned. The array starts indexing from 1.The main drawback of array representation is wasteful of memory when there are many missing elements.

The binary tree with n elements requires array size up to 2^n. Suppose array positions indexing from 0, then array size reduces to 2^n-1. The right skewed binary trees have maximum waste of space. The following right skewed binary tree's array representation is shown as follows.



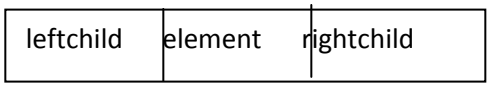
(a)



0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15

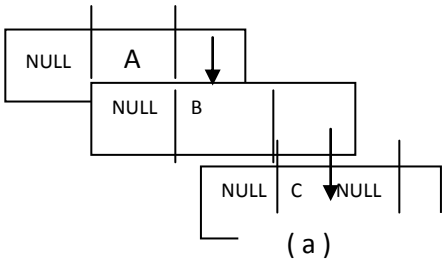
(ii) **Linked list Representation:** The most popular way to represent a binary tree is by using links or pointers. The node structure used in this representation consists of two pointers and an element for each node.

node structure:

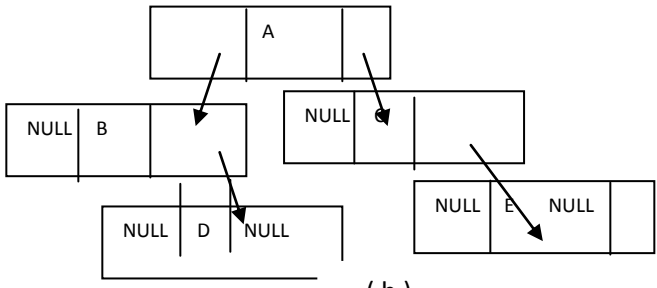


The first field leftchild pointer maintains left child of it. The middle field is the element of the node and last field is the right child pointer maintains right child of it.

The following are the linked list representations for the examples given in array representation.



(a)



(b)

Binary tree traversals: There are four ways to traverse a binary tree. They are

(a) Pre order
(b) In order
(c) Post order
(d) Level order

The first three traversals are performed using recursive approach and are done using linked list scheme.

In these, the left sub tree is visited before visiting right sub tree. The difference among these is position of visiting the node.

(a) **Pre order:** In this, each node is visited before visiting the left and right sub trees.

The pseudo code is:

```
template<class T>
void preorder(binarytreenode<T> *t)   // where t is the root
{
if(t==NULL)
{
cout<<"tree empty";
return;
}
cout<<t->element;
preorder(t->leftchild);
preorder(t->rightchild);
}
```

(b) **In order:** In this, each node is visited after the left sub tree but before the right sub tree.

The pseudo code is:

```
template<class T>
void inorder(binarytreenode<T> *t)   // where t is the root
{
if(t==NULL)
{
```

```
cout<<"tree empty";
return;
}
inorder(t->leftchild);
cout<<t->element;
inorder(t->rightchild);
}
```

(c) **Post order:** In this, each node is visited after left and right sub trees are visited.

The pseudo code is:

```
template<class T>
void postorder(binarytreenode<T> *t)   // where t is the root
{
if(t==NULL)
{
cout<<"tree empty";
return;
}
postorder(t->leftchild);
postorder(t->rightchild);
cout<<t->element;
}
```

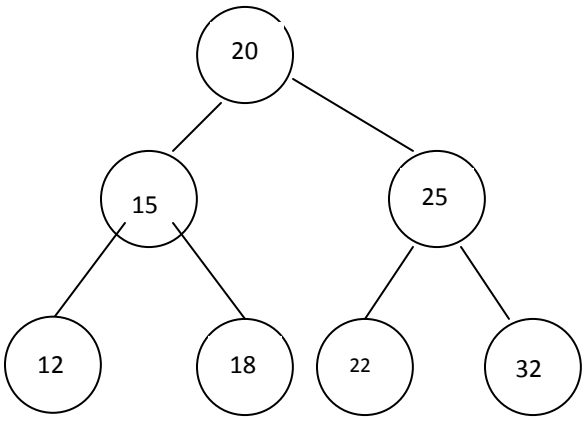
(d) **Level order:** The elements of a binary tree are visited level by level and are from left to right within levels. It is difficult to write recursive function for this and the correct data structure used is queue.

```
template<class T>
void levelorder(binarytreenode<T> *t)
{
arrayqueue<binarytreenode<T> *> q;
while(t!=NULL)
{
cout<<t->element;
if(t->leftchild!=NULL)
    q.push(t->leftchild);
if(t->rightchild!=NULL)
    q.push(t->rightchild);
try
{ t=q.front();
}
catch(queueEmpty) { return; }
q.pop();
}
```

This level order enters into while if t is not empty. If t has children, they are added to queue. It allows accessing front element of queue q.

Let n be the number of elements in the binary tree. Then, the space and time complexities of each of these traversals is O(n).

The following is an example binary tree with pre order, in order, post order and level order traversals:



pre order is : 20 15 12 18 25 22 32

in order is : 12 15 18 20 22 25 32

post order is : 12 18 15 22 32 25 20

level order is : 20 15 25 12 18 22 32

(a)

3.4 Operations on Binary Tree

3.5 Types of Binary trees

3.5.1 Binary Search Trees:

Any empty binary tree is an Binary Search Tree. A nonempty binary search tree has the following properties.

(i) Every element has the key or value and no two elements have the same key. Therefore, all keys in the tree must be distinct.

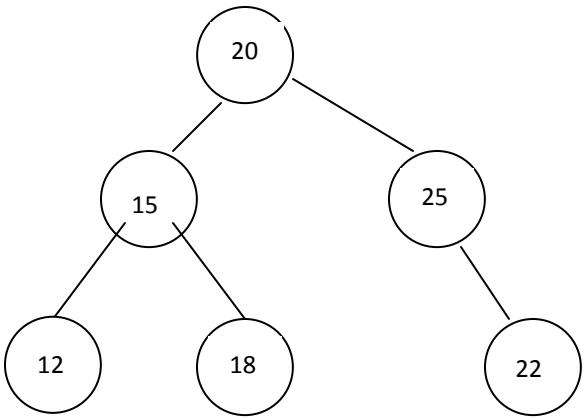
(ii) Any element key in left sub tree is less than the key of the root.

(iii) Any element key in right sub tree is greater than the key of the root.

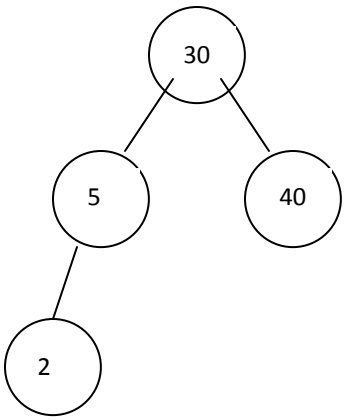
(iv) Both left and right sub trees are also binary search trees.

There is some redundancy in this definition. The properties 2, 3, 4 together imply the keys must be distinct.

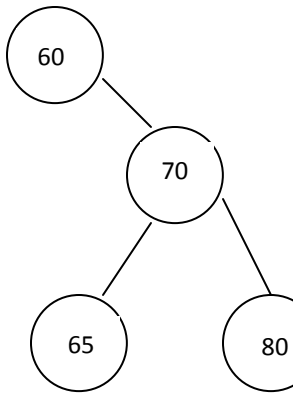
Some binary trees in which the elements with distinct keys are shown in the following figures.



(a)



(b)



(c)

The number inside a node is the element key. The tree (a) is not a binary search tree because the right sub tree of element 25 violating property 4.It means 22 is smaller than its parent 25. The trees of (b) and (c) are not binary search trees.

When the property all keys are distinct is removed, then property 2 is replaced by smaller or equal and property 3 is replaced by larger or equal. The resulting tree is called a binary search tree with duplicates.

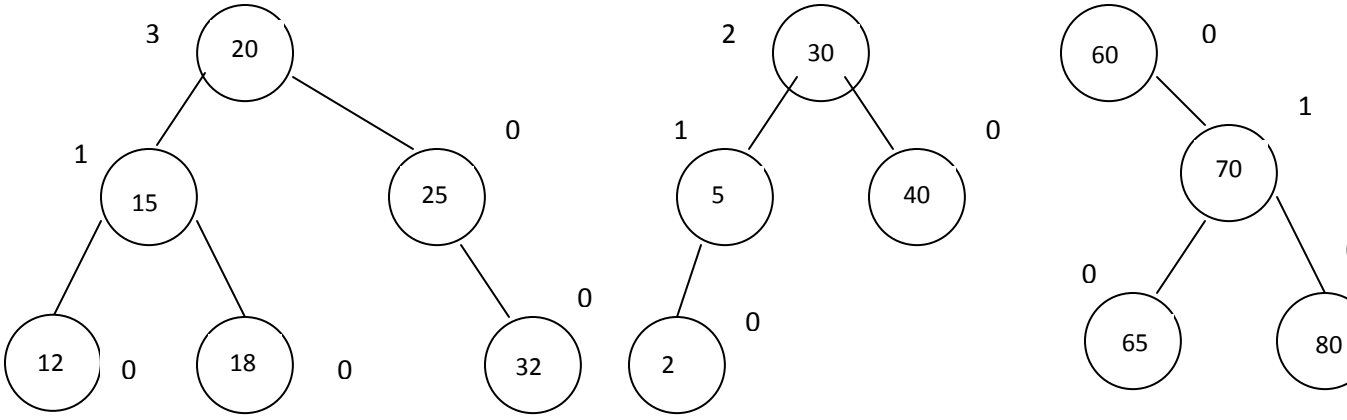
3.5.1.1 Indexed Binary Search Trees: It is derived from ordinary binary search tree by adding the field leftsize to each tree node. The leftsize field for a node gives the number of elements in the node's left sub tree. The leftsize field also gives the index of the element with respect to elements of that node's sub tree.

In the following trees,

(i) the leftsize field for node 20 in (a) tree is 3. When (a) tree is represented (using in order traversal) in the array, the list is 12,15,18,20,25 and 32.

(ii) the leftsize field for a node 30 in tree (b) is 30. When tree (b) is represented in the array using in order gives the list 2,5,30 and 40.

(iii) the leftsize field for a node 70 in tree (c) is 0.When the sub tree rooted 70 represented in the array, the list gives 65,70 and 80.



(a) (b) (c)

3.5.1.2 Abstract Data Type for Binary search tree:

The abstract data type for binary search tree defines instances and operations.

AbstractDataType bst

{

Instances: The key of any node in left sub tree is smaller than key in the root and the key in the right sub tree is larger than key in the root. Hence, all keys are distinct.

Operations:

(i) find(k): It returns the pair with key k.

(ii) insert(p): It inserts the pair into the binary tree.

(iii) erase(k): It removes the pair whose key is k.

(iv) ascend(): It displays all the keys in the ascending order of keys.

}

There is Abstract Data type for Indexed binary search tree is same as binary search tree but adds leftsize field for each tree node.

AbstractDataType Indexedbst

{

Instances: The key of any node in left sub tree is smaller than key in the root and the key in the right sub tree is larger than key in the root. Hence, all keys are distinct. This also maintains leftsize field for each tree node.

Operations:

find(k): It returns the pair with key k.

(i) get(index): It returns the pair with index

(ii) insert(p): It inserts the pair into the binary tree.

(iii) delete(index): It removes the pair whose index is index.

(iv) erase(k): It removes the pair whose key is k.

(v) ascend(): It displays all the keys in the ascending order of keys.

}

There is an abstract data type for duplicate binary search tree and that is shown below.

AbstractDataType dbst

{

Instances: The key of any node in left sub tree is smaller or equal to key in the root and the key in the right sub tree is larger or equal to key in the root. Hence, two elements have the same key.

Operations:

(i) find(k): It returns the pair with key k.

(ii) insert(p): It inserts the pair into the binary tree.

(iii) erase(k): It removes the pair whose key is k.

(iv) ascend(): It displays all the keys in the ascending order of keys.

}

There is an abstract data type for duplicate indexed binary search tree and that is shown below.

AbstractDataType duplicateIndexedbst

{

Instances: The key of any node in left sub tree is smaller or equal to key in the root and the key in the right sub tree is larger or equal to key in the root. Hence, all keys are distinct. This also maintains leftsize field for each tree node.

Operations:

(i) find(k): It returns the pair with key k.

(ii) get(index): It returns the pair with index

(iii) insert(p): It inserts the pair into the binary tree.

(iv) delete(index): It removes the pair whose index is index.

(v) erase(k): It removes the pair whose key is k.

(vi) ascend(): It displays all the keys in the ascending order of keys.

}

3.5.1.3 Operations in binary search tree: There are three operations performed in a binary search tree. They are

(i) Searching: When searching for a pair with key thekey. It begins at root.If root=Null, the search is unsuccessful. Otherwise, thekey is compared with key in the root. If thekey is less than root key, no pair in right sub tree can have thekey and searching continues in left sub tree. If thekey is larger than key in the root, searching continues in right sub tree. The same procedure also applied to sub trees.

pseudo code:

```
template<class K,class E>
pair<K,E> *binarysearchtree<K,E>::find(K &thekey) // returns pointer to matching pair. otherwise,returns NULL
{
binarytreenode<pair<K,E>> *p=root;  // p initially assigned with root
while(p!=NULL)
{
if(thekey<p->element.first)
        p=p->leftchild;
```

```
else if(thekey>p->element.first)
        p=p->rightchild;
else
        return &p->element;
}
return NULL;
}
```

The time complexity of searching operation when binary search tree of height O(log n) is O(log n).

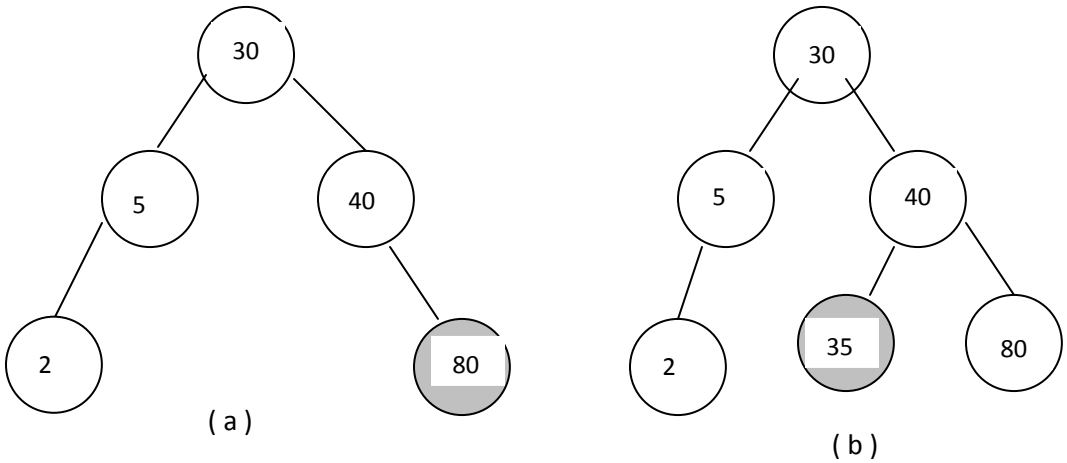
(ii) Insertion: To insert new pair thepair into binary search tree, determine whether thepair.first means key is different from the existing elements in the tree. If search is successful, then replace the existing pair old value with new pair value. If searching is unsuccessful, then new pair is inserted as the child of last node examined during the search.

pseudo code:

```
template<class K,class E>
void binarysearchtree<K,E>::insert(pair<K,E> &thepair) // inserts or overwrites existing pair
{
binarytreenode<pair<K,E>> *p=root,*pp=NULL;
while(p!=NULL)
{
pp=p;
if(thepair.first<p->element.first)
    p=p->leftchild;
else if(thepair.first>p->element.first)
    p=p->rightchild;
else
{
p->element.second=thepair.second;
return;
}
]
binarytreenode<pair<K,E>> *n=binarytreenode<pair<K,E>> (thepair);
if(root!=NULL)
    if(thepair.first<pp->element.first)
            pp->leftchild=n;
    else
            pp->rightchild=n;
else
    root=n;
treesize++;  // treesize maintains (count) number of nodes after insertion thepair.
}
```

The time complexity of insertion for a binary tree of height O(log n) is O(log n).

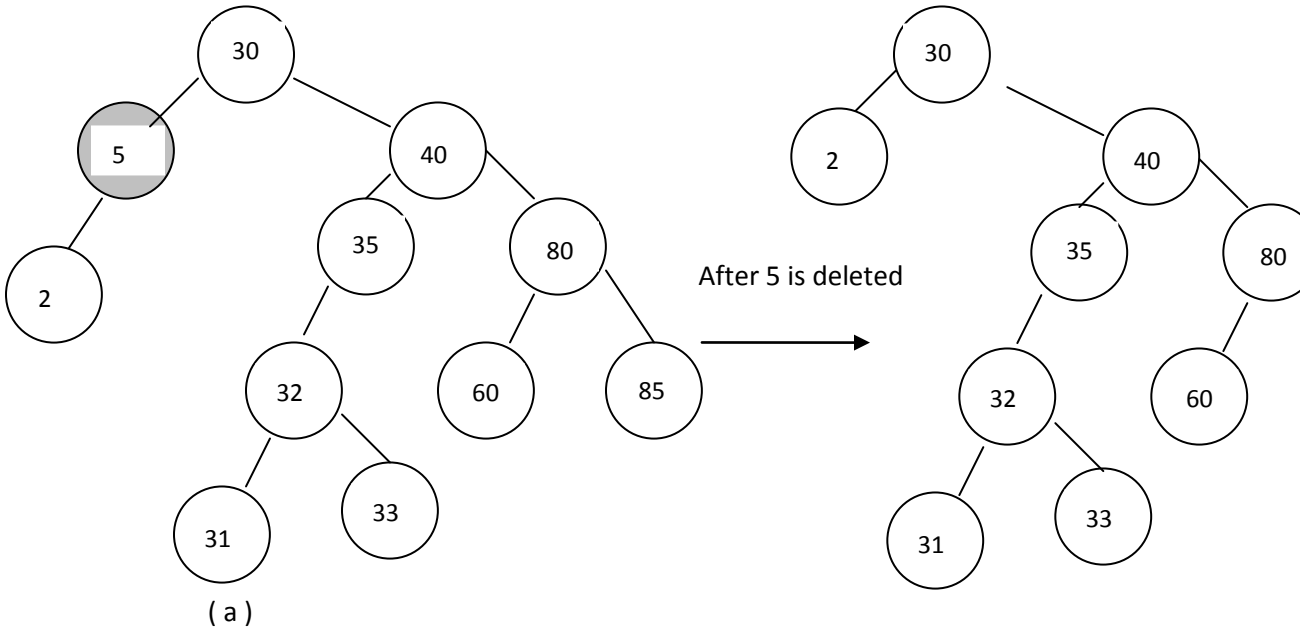
The following are example trees in which 80 inserted in to (a) and 35 into (b).



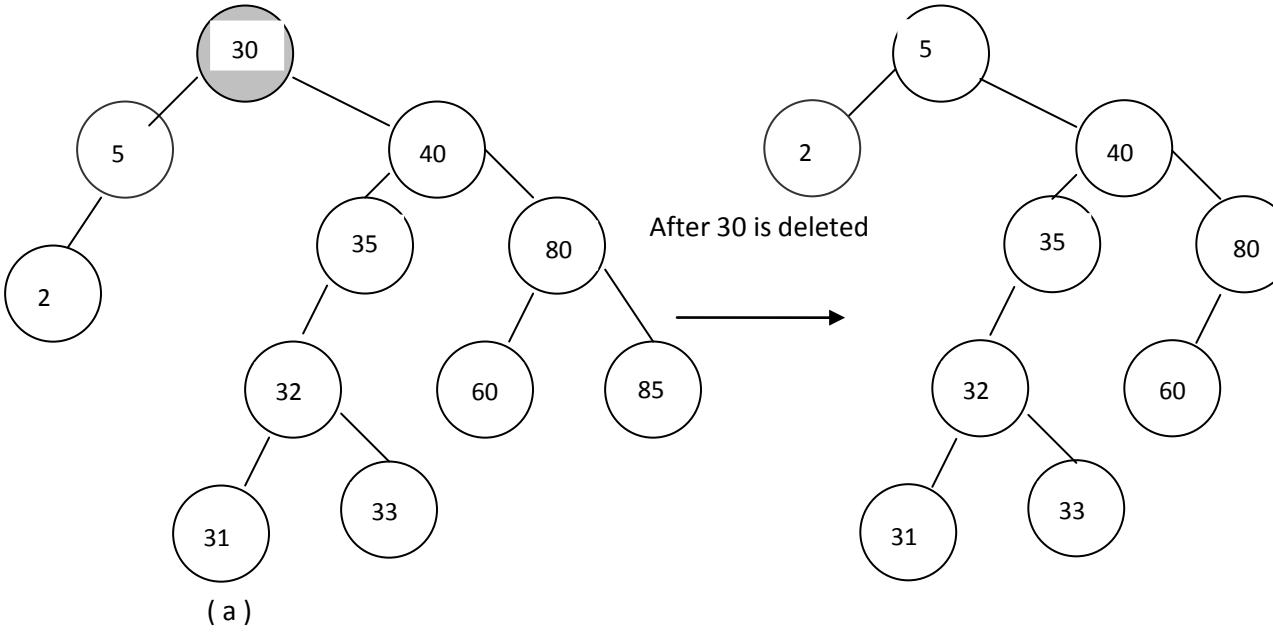
(iii) Deletion: There are possibilities of removing a node

(i) **deleting a leaf:** When deleting leaf is a root, set it to NULL.Example is delete 35 in (b). It falls into case 1 and leftchild pointer of 40 set to NULL.

(ii) **Deleting node has exactly one nonempty sub tree:** Suppose the deleting node has no parent, the root of its single sub tree becomes root for new tree. If deleting node has a parent. then , change the pointer of deleting node parent to point to deleting node only child.



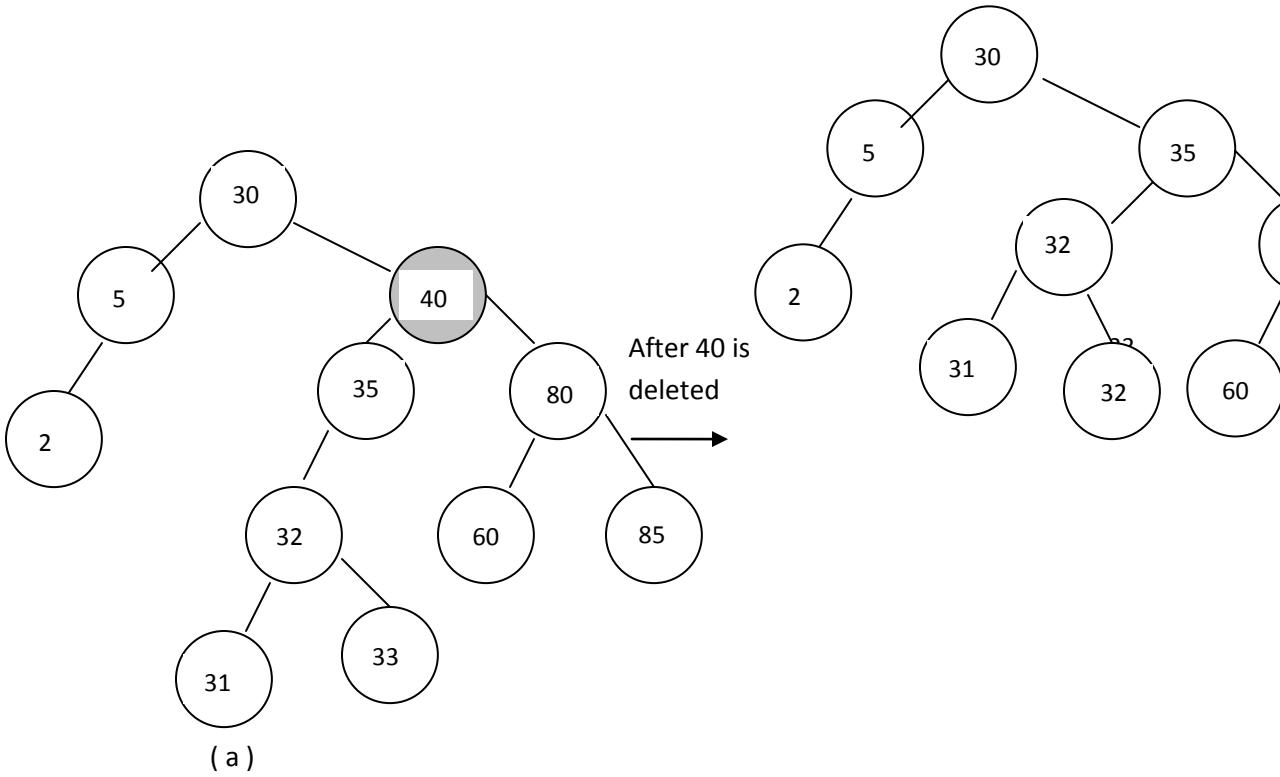
In above tree, the deleting node 5 has only left child. After this node is deleted, its parent pointer point to its only child.



(a)

In the above tree, the deleting node is the root. then, it is the replaced by its left sub tree root.

(iii) **Deleting node has exactly two nonempty sub trees:** The deleting node should be replaced by either with largest key in the left sub tree or smallest key in the right sub tree. To find largest key in the left sub tree, visit first root of the sub tree and traverse only right child pointers until a node is reached whose pointer is NULL. To find smallest key in the right sub tree, visit first root of the sub tree and go to only left child pointers until a node is reached whose left child pointer is NULL.
In the following tree, when deleting the node 40 which has both left and right sub trees, it is replaced by either 35 in left sub tree or 60 in its right sub tree.



(a)

pseudo code:

```
template<class K,class E>
void binarysearchtree<K,E>::erase(K &thekey)
{
binarytreenode<pair<K,E>> *p=root,*pp=NULL;
while(p!=NULL && p->element.first!=thekey)    // it points deleting node in the tree
{
pp=p;
if(thekey<p->element.first)
    p=p->leftchild;
else
    p=p->rightchild;
}
if(p==NULL)      // if root == NULL
    return;
if(p->leftchild!=NULL && p->rightchild!=NULL)     When deleting node has both childs
{
binarytreenode<pair<K,E>> *s=p->lestchild,*ps=p;
while(p->rightchild!=NULL)
{
ps=s;
s=s->rightchild;
}
delete p;
```

```
p->element=p->element;
p=s;
}
binarytreenode<pair<K,E>> *c;
if(p->leftchild!=NULL)     // When deleting node has only one child
    c=p->leftchild;
else
    c=p->rightchild;
if(root==p)
    root=c;
else
{
if(p==pp->leftchild)
pp->leftchild=c;
else
pp->rightchild=c;
}
treesize--;  // it maintains number of nodes in a tree
delete p;
}
```

3.5.1.4 Binary Search tree Applications: It is mainly used in

(i) Histogramming

(ii) Best fit bin packaging

(iv) crossing distribution

3.5.2 Heap Trees

3.6 Height Balanced Trees

3.7 B Trees

AVL and red black trees are used when the dictionary is small enough to reside in internal memory. The search trees of higher degree are needed to get better performance for external dictionaries. ISAM is used to get good sequential and random access for external dictionaries.

3.9.1 m-way search trees**:** An m-way search tree may be empty. If it is not empty, it must satisfy the following properties.

(i) In the tree, each internal node has up to m children and has elements between 1 & m-1.

(ii) Every internal node with p elements has p+1 children.

(iii) Consider any node with p elements. Let k1, k2,……kp be the keys of these elements. The elements are ordered such that k1<k2<……<kp. Let c0,c1,……cp be p+1 children of the node. The elements in the sub tree with root c0 have keys smaller than k1, those in the sub tree with root cp have keys larger than kp and those in sub tree with root ci have the keys larger than ki but smaller than ki+1 where i≤ i < p.

Although external nodes are included when defining m-way search tree, external nodes are not represented physically in actual representation. Consider the following seven way search tree, which have external nodes as solid squares and all other nodes are internal nodes. The root has 2 elements and 3 children. The middle child of the root has 6 elements and 7 children in which 6 are external nodes.

3.9.1.1 searching: To search for an element with key 31, begin checking with root first. The searching drops to middle child of the root because 31 lies between 10 & 80. Since search finds k2 (=30) < 31 < k3 (=40), search drops to third sub tree of this node. There 31 < k1 (=32), search moves to first sun tree of this node but external node is reached. When search terminates at external node, the key is not found. Otherwise means search exited at any internal node, the key is found.

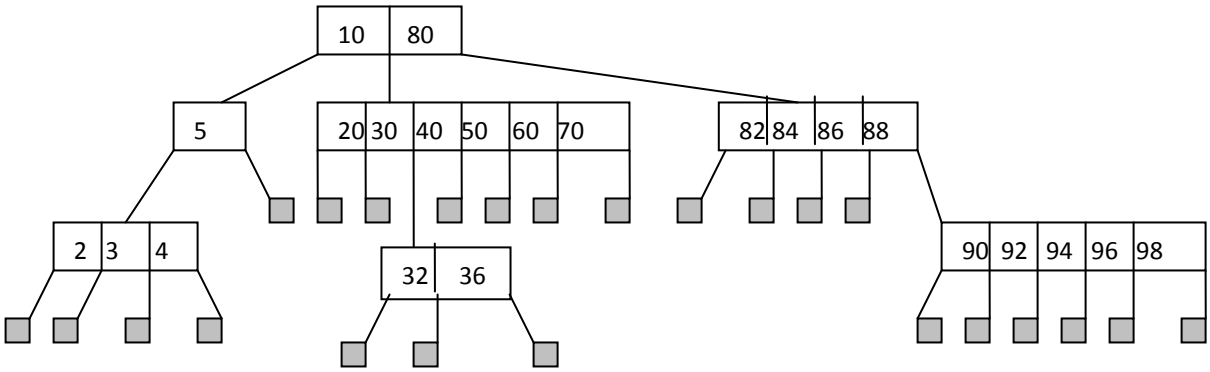


Fig:1 **m-way search tree example**

3.9.1.2 insertion: To insert an element with key, first search the tree. If it doesn't contain the key, then insert it. To insert the key 31, the search begins at root, then goes to middle child of root, then third child of this node, then first child of this node which is the external node. Since the node [32,36] can hold up to 6 elements, the new element is inserted as first element of this node.

Another example is insert an element with key 65, the search terminates at sixth child of middle child of the root. The new element is created and inserted into there.

3.9.1.3 deletion: To delete an element with key, first search for it. If it is there, then delete it.(i) If the deleted element is 20 in fig above, search for it. The searching ends at first element of the middle child of the root. Since it's childs c0 & c1 are 0, it can be deleted easily and results a node [30,40,50,60,70].

(ii) To delete 84, search ends at second element in third child of the root. Since it's childs c1 & c2 are 0, it can be deleted easily by resulting [82,86,88].

(iii) To delete an element with key 5, more work to be done. It has a nonnull c0 and c1 is external node. The largest key in the c0 is brought to the deleted node place.

(iv) To delete an element with key 10, the root take either largest in its c0 or smallest in c1. Suppose 5 of c0 was brought to the root, 4 in the c0 of deleted node 5 is brought to 5's old place.

3.9.1.4 Height: An m-way search tree of height h may have as few as h elements and as many as m^h-1. This upper bound is obtained from the levels 1 through h -1 has exactly m children and nodes at level h have no children. Since each of these nodes has m-1 elements, the number of elements is m^h-1.

An 200 way search tree of height 5 can hold 32 * 10^{10} -1 elements but might fold as few as 5 only.

3.9.2 B – trees: A B – tree of order m is an m – way search tree. If the B – tree is not empty, the corresponding extended tree must satisfy the following properties:

(i) The root has at least two children.

(ii) All internal nodes other than the root have at least [m/2] children.

(iii) All external nodes are at the same level.

The seven way search tree in fig 1 is not a B-tree of order 7 because

(i) all the external nodes are not on the same level.

(ii) Some of the internal nodes have two (= node [5]) and three (= node 32,36]) children which is not satisfying 2nd property i.e [7/2]=4 children.

The following is an B-tree of order 7:

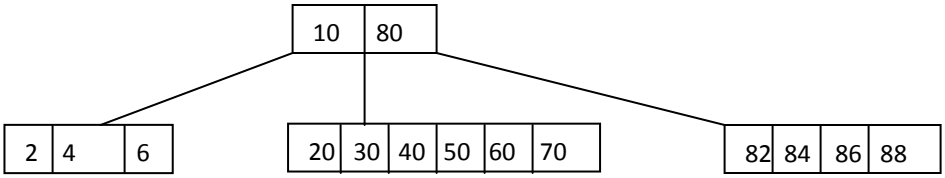


Fig 2 : **B – tree of order 7**

In a B-tree of order 2, all the internal nodes have exactly two children. This requirement coupled with all external nodes on the same level results full binary trees.

In a B-tree of order 3, internal nodes have either two or three children. It is also called 2 -3 tree.

In a B- tree of order 4, internal nodes have two, three or four children. These are also referred as 2-3-4 trees and are called 2-4 trees. The following is an 2-3 tree. It becomes 2-3-4 tree when adding 14 and 16 to left child of 20.

3.9.2.1 Height of a B – tree: Let T be a B – tree of order m and height be h. Let m=[d/2] and n be number of elements in T.

(a) $2d^{h-1} -1 \leq n \leq m^h -1$

(b) $\log_m (n+1) \leq h \leq \log_d (n+1/2)+1$

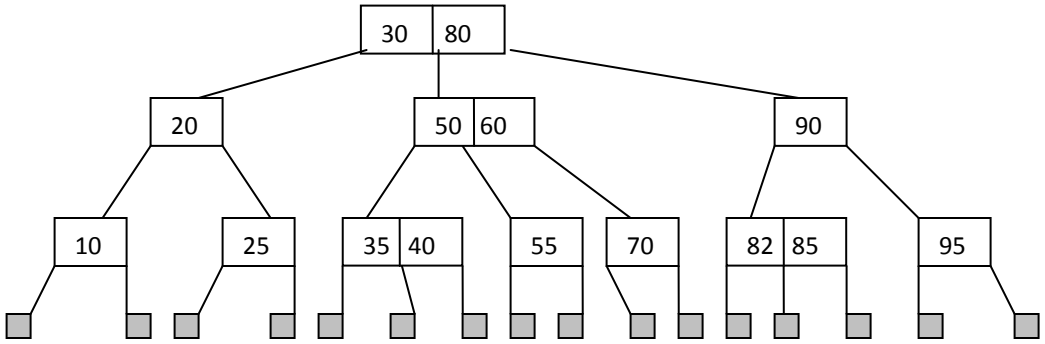


Fig 3: **B – tree of order 3**

3**.9.2.1 Searching:** The searching an element in a B-tree is same as algorithm used for m-way search tree. Searching an element in an internal node of a B-tree of height takes at most h because all internal nodes need to be checked during the search.

3**.9.2.2 Insertion:** To insert an element, first search for the presence of the element with same key. If such an element is found, insertion fails because duplicates are allowed. When searching is unsuccessful, then insert new element into the last internal node encountered on the search path. To insert an element with key 3 into **Fig 2**, search terminates at second child of left child of the root. It can be inserted into [2,4,6] node results [2,3,4,6] node since this node hold up to 6 elements. The number of disk accesses to do this is 3 in which two accesses for reading root and then its left child and another for writing out modified node after insertion. It can be shown as follows.

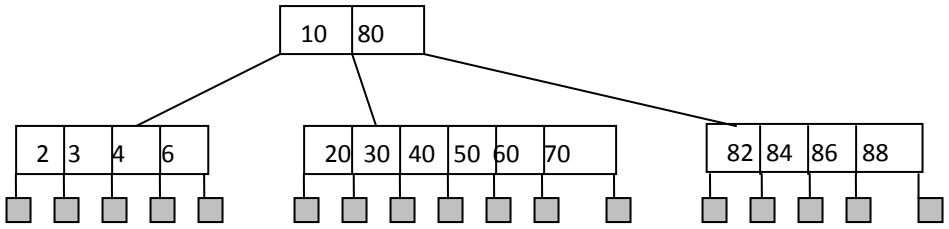


Fig 7.3 .3 (a) : After inserting 3 into Fig 7.3.2

To insert an element with key 25 in B-tree of order 7 (Fig 2), the element goes into middle child of the root i.e [20,30,40,50,60,70] but this node is full. When element goes to full node, the overfull node need to be split as follows.

Let the overfull node be P=[20,25,30,40,50,60,70]. Let it has m elements and m+1 children. It can be denoted as m, c_0, (e_1, c_1), (e_2, c_2),……, (e_m, c_m).

where ei 's indicate elements and ci's represent children pointers. The node is spli around ed where d=[m/2].

The elements to the left remain in P and to the right move into a new node Q but P & Q must contain at least [m/2] children.

The element ed moved to the parent of P. The format of P and Q are
P: d-1, c0, (e1,c1), ……., (ed-1,cd-1)
Q: m-d, cd, (ed+1,cd+1),……., (em,cm).
In this case, the overfull node is 7,0,(20,0),(25,0),(30,0),(40,0),(50,0),(60,0),(70,0). It can be split around d=4 which yields P=3,0,(20,0),25,0),30,0) and Q=3,0,(50,0),(60,0),(70,0). The e4=40 moved to P's parent. Here, it is the root. It can be shown as follows. The number of disk accesses required is 5 in which two for searching the proper position in the tree, two for writing out the split nodes and one for writing modified root.

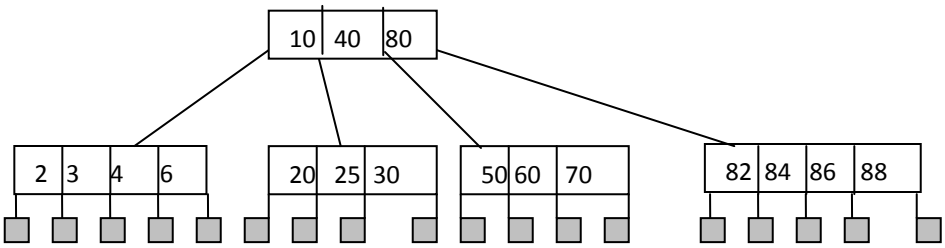


Fig 3(b) : After inserting 25 into Fig 7.3.3 (a)

To insert element with key 44 into B – tree of order 3 like Fig 3 (c), the element goes to [35,40] node. Since it is full, the overfull node is [35,40,44] can be represented as 3,0,(35,0),(40,0),(44,0). It can be split around d=[3/2]=2 yields
P=1,0,(35,0) and Q=1,0,(44,0). The element with key 40 move to P's parent A =[50,60]. The resulted overfull node be 3,P,(40,Q),(50,C),(60,D) where C & D are pointers to the nodes [55] & [60]. The overfull node **A** be split to create a new node B. The new A & B are
A: 1,P,(40,Q) and B: 1, C, (60,D) .
Before insertion, root format is R: 2, S, (30,A),(80,T) where S & T are first and third sub trees of the root. After insertion, the overfull node is R: 3, S, (30,A), (50,B),(80,T). This node is split around d=[3/2]=2 yields R: 1,S,(30,A) and U= 1,B, (80,T). The element 50 moved to R's parent. Since R has no parent, it can be created as new root and that has format 1,R, (50,U). The resulting tree is shown as below.

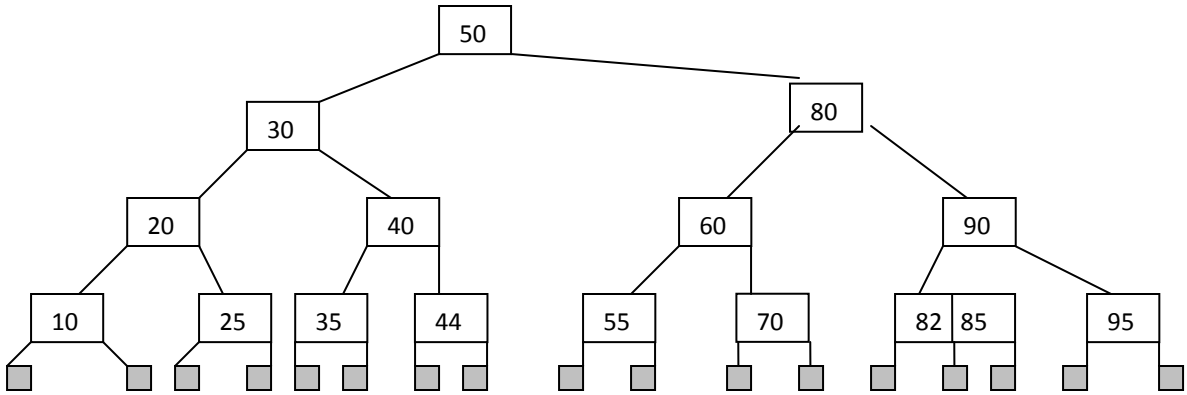


Fig 3 (c): After insertion 44 into Fig 7.3.3

The total number of disk accesses is 10 in which 3 accesses for reading [30,80],[50,60] and [30,40], six disk accesses for writing out 3 split nodes and one for writing out new root.

When insertion cause s nodes to split, the number of disk accesses is h (reading the nodes on the search path) + 2s (to write out two split parts of each node that is split) + 1 (to write out new root). The total number of disk accesses is h+2s+1 which is at most 3h+1.

3.9.2.3 Deletion: Deletion first divided into 2 cases. (1) The element to be deleted in a node whose children are external nodes. (2) the element to be deleted from a non leaf. case (2) is transformed into case (1) by either largest element in its left neighboring sub tree or smallest element in its right neighboring sub tree.

(i) To delete an element with key 80 in Fig 3 (a), the suitable replacement used is either the largest element in its left sub tree 70 or smallest element 82 in its right sub tree.

(ii) To delete an element with key 80 in Fig 3 (c), the replacing element used is either 70 or 82. If 82 is selected, the problem of deleting 82 from the leaf remains.

The (ii) falls into 2 cases. **One is delete an element from a leaf contains more than the minimum number of elements (1 if the leaf is also root and [m/2]-1 if it is not) requires to simply write out modified node.**

Examples: To delete 50 from Fig 3 (a), write out the modified node [20,30,40,60,70]. To delete 85 from 2-3 tree such as Fig 3 (c), write out the modified node [82]. In this case, h disk accesses to go to deleted node place and additional access to write out modified node.

Other is deleting from a nonroot has exact number of elements, try to replace the deleted element with an element from its nearest left or right sibling.

Example: To delete an element 25 from Fig 3 (b), deleting 25 from [20,25,30] leaves a node with 3 children. It violates B-tree of order 7 since every internal node must has at least 4 children. Check its nearest left sibling, it has one extra element [2,3,4,6]. The largest element from here is moved to the parent node and the intervening element means 10 in the root

is moved down to deleted node. It is shown as follows. The number of disk accesses is 6 in which 2 for to go to 25, 1 for reading its nearest left sibling and 3 for to write out modified leaf, its sibling and the root.

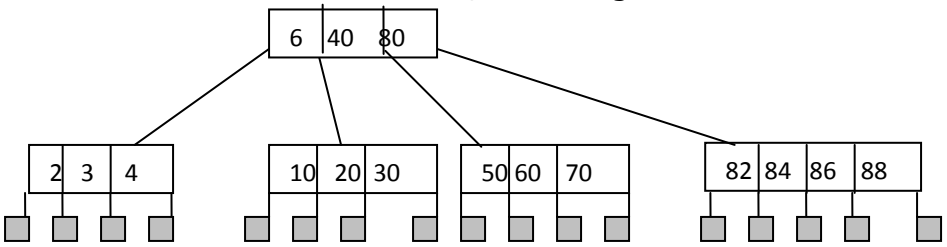


Fig 7.3.3 (d) : After deleting 25 from Fig 7.3.3 (b)

Suppose instead of checking left sibling, right sibling is checked but it don't have extra elements. then, merge the two siblings with the element between them in the parent into a single node. In this case, the node after deleting 25 has [20,30] ,its right sibling [50,60,70] and the element with key 40 are merged into a sinle node [20,30,40,50,60,70]. The resulted tree is as follows.

This deletion takes 5 disk accesses in which 2 for reaching 25,1 for reading its right sibling and 2 for writing out modified nodes(root and merged node).

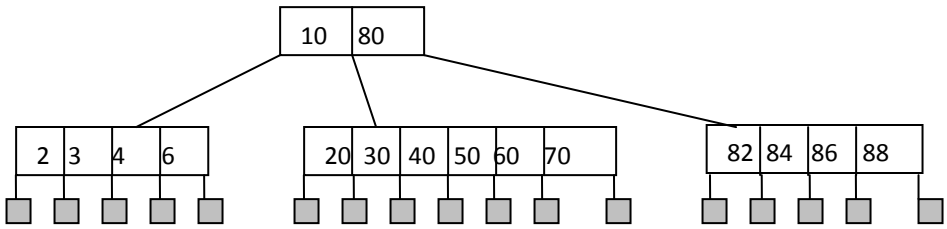


Fig 3 (e) : After deleting 25 from Fig 3(b)

Note that merging reduces number of elements in the parent, the parent may end up with one element short. If the parent is one element short, check the parent's nearest sibling and get an element from there or merge with it. This process suppose yields grand parent become one element short and it is the empty root, discard the empty root. Hence, tree height decreases by 1.

suppose the element 10 need to be deleted from Fig 3. This deletion leaves a leaf with zero elements. Its nearest right sibling [25] doesn't have extra element. Hence, the two sibling nodes and the element between them in the parent are merged into a single node. The new tree structure appears in Fig 3(i) that is as follows.

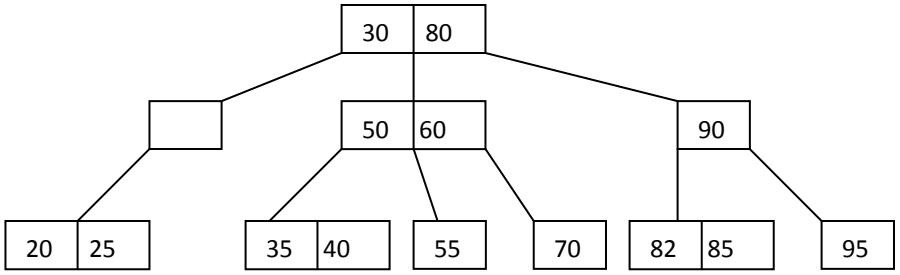


Fig 3 (f) : After merging at leaf node (deleted 10)

Now, the node at level 2 has one element short. Its nearest right sibling has an extra element. The left most element from the right sibling moved to the parent and 30 in the root moved down. The resulting tree is as shown as below.

The number of disk accesses is 9 in which 3 to go to 10, 2 for reading its nearest right sibling and 4 for writing out modified nodes at level 3,2 and 1.

The final example is delete 44 from 2-3 B-tree such as Fig 3(f) is shown as follows:

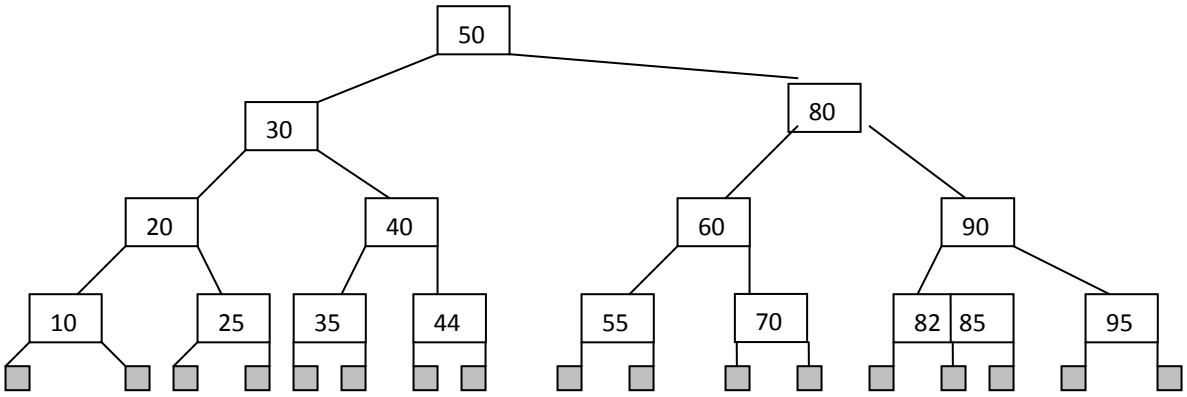


Fig 3 (f): After insertion 44 into Fig 3 (c)

After deleting 44 from leaf, it becomes one element short. Its nearest left sibling doesn't have an extra element. So, two siblings and element between them in the parent are merged into a single node. It can be shown as follows.

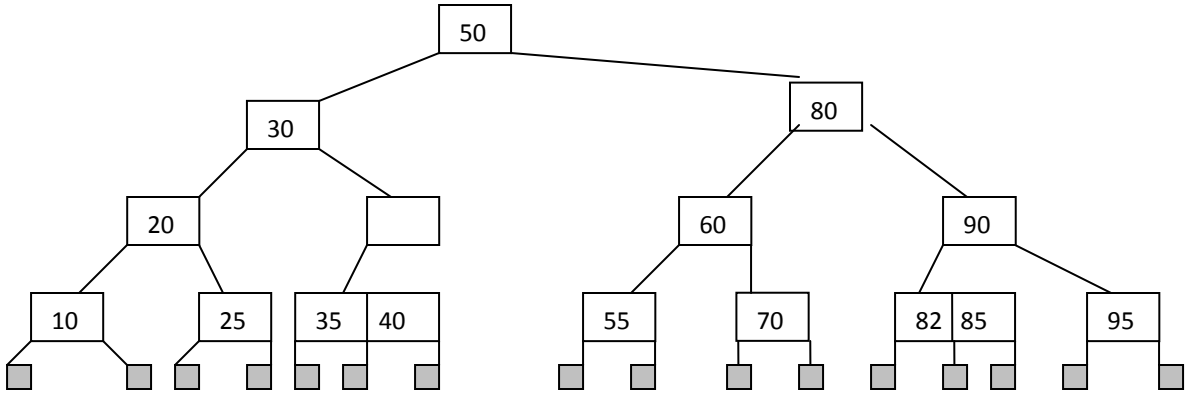


Fig3 (g): After merging at leaf level

The node at level 3 is one element short. Its nearest left sibling is examined and it doesn't have extra elements. So, two siblings and the element between them in the parent are merged into a single node. It can be shown as follows.

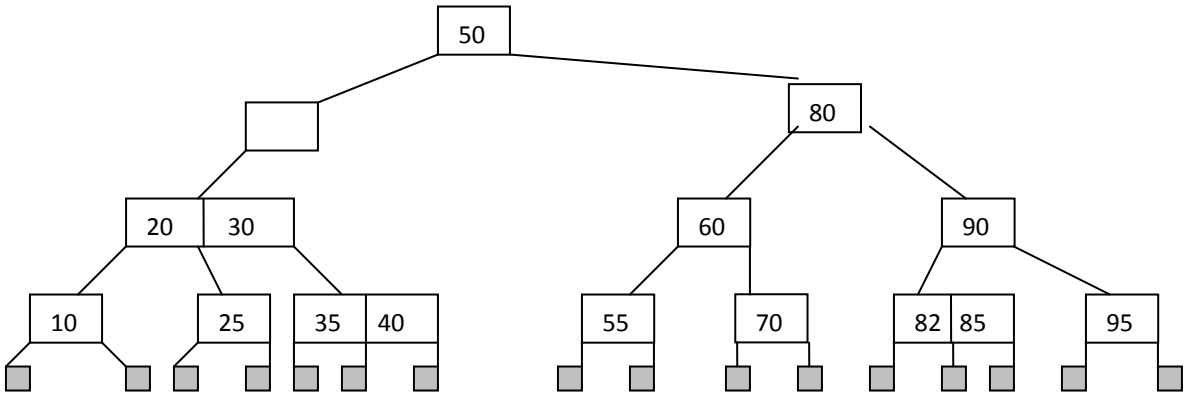


Fig 3 (g): After merging at level 3

Now, the node at level 2 has one element short. Its nearest right sibling has no extra elements. Again, one more
merging is performed that results a tree as follows.

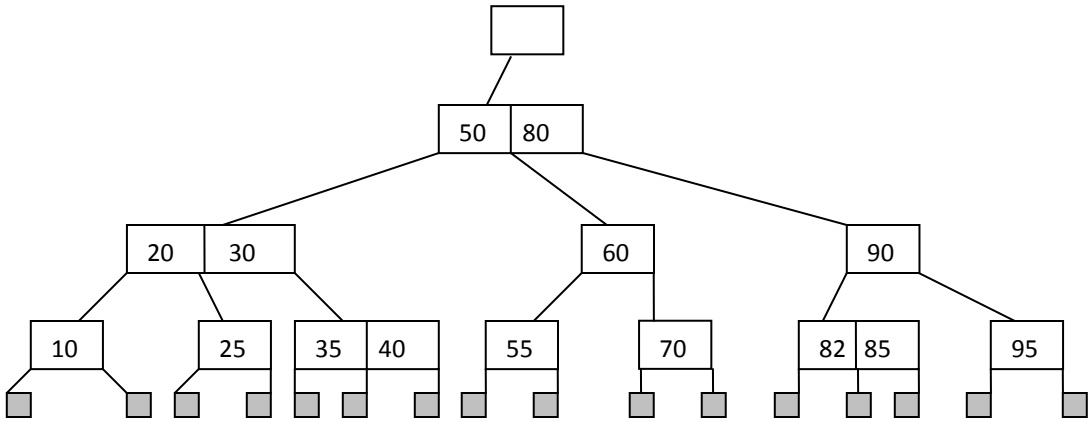


Fig3 (h): After merging at level 2

Now, root is one element short. Since root becomes one element only when it is empty, discard the root. The final 2-3 tree is shown as follows. The number of disk accesses is 10 in which 4 accesses for to reach 44, 3 for reading its nearest siblings and 3 to write out modified nodes.

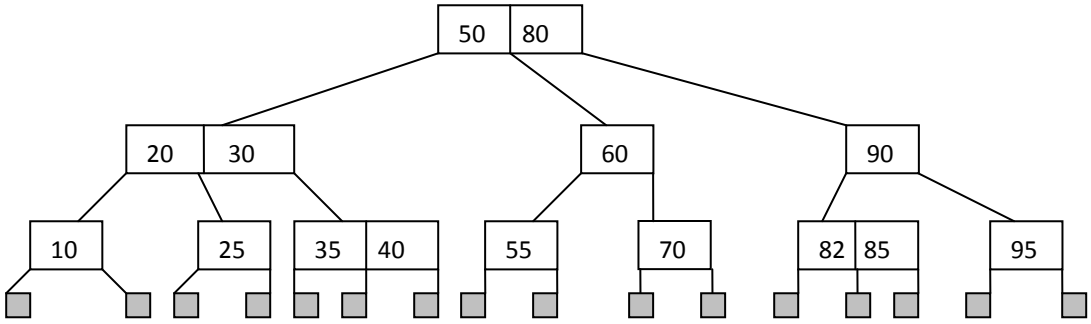


Fig 3 (i): After merging at level 2

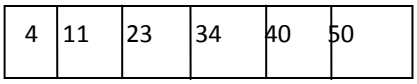
The deletion from a B-tree of height h is when merging tales at levels h,h-1,…..and 3 and getting an element from a nearest sibling at level 2 is 3h.

Note: when the element size is large relative to the size of a key, the following node structure is used.

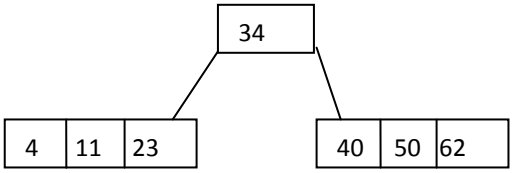
$s,c_0,(k_1,c_1,p_1),(k_2,c_2,p_2),\ldots\ldots,(k_s, c_s, p_s)$ where s is the number of elements in the node, k i 's are element keys, pi's are the disk locations of the corresponding elements and ci's are children pointers.

Exercise 1: Draw the B-tree of order 7 resulting from inserting the following keys into an empty tree T: 4,40,23,50,11,34,62,78,66,22,90,59,25,72,64,77,39 & 12.

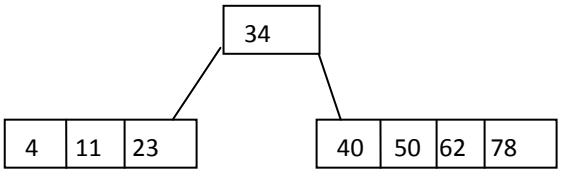
Step 1: Since it is a B-tree of order 7, the maximum number of elements a node contain is 6.



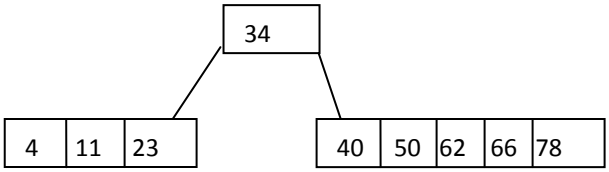
Step 2: Next element to be inserted is 62, but this is full because the maximum number of children that internal node have is 7 and minimum number of children is 4[=7/2].
The overfull node is P= [4,11, 23,34, 40,50,62]. It can be split around e4=34. The elements to left are remain in P and to the right in Q. The element e4 goes to the node parent.



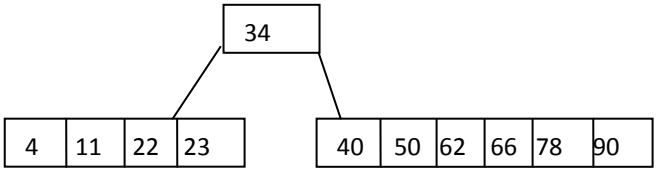
Step 3: The next element 78 goes to root right child.



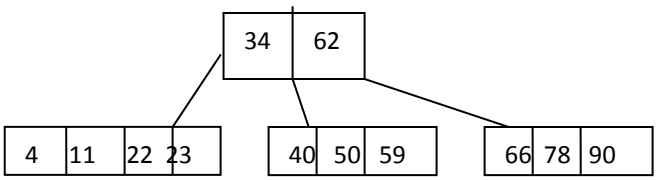
Step 4: The next element inserted is 66, it goes into root right child.



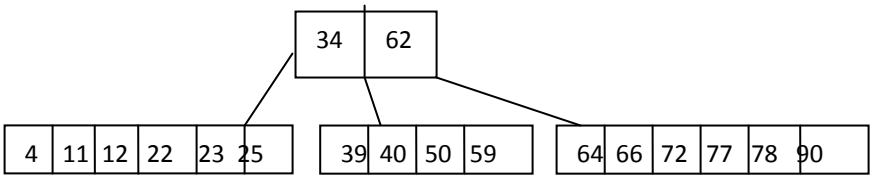
Step 5: The next elements 22 & 90 goes into root left child and root right child respectively. Now, root right child is full. If any element insert into it needs split.



Step 6: The next element 59 goes into root right child and it becomes overfull node. This needs to be split.
Let C= [40,50,59,62,66,78,90]. It can be split around e4=62 leaves C=[40,50,59] and D=[66,78,90]. The element 62 moves to the parent. Now, the root is 34,62]. Its Childs are P,C & D.



Step 7: The elements 25, 72,64,77,39 & 12 are inserted in which 25 & 12 are in root first sub tree, 39 to root middle sub tree and 64,72 & 77 to root third sub tree.



3.8 Red Black trees

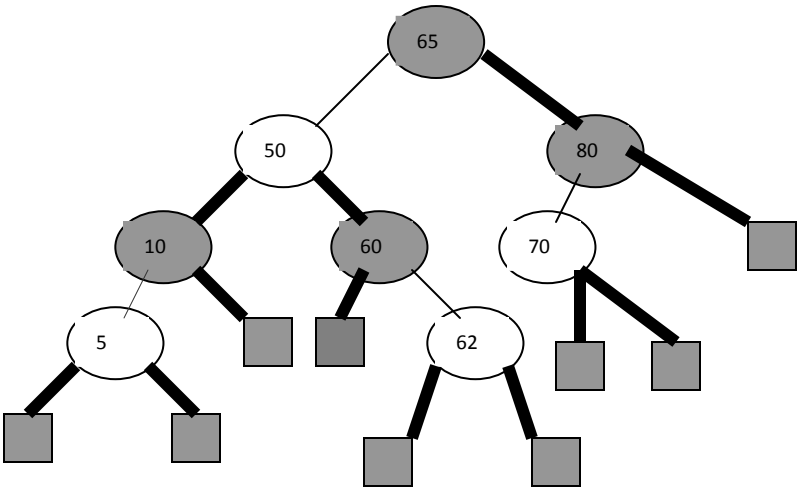
It is an extended binary search tree in which null pointers are replaced by external nodes and also in which every node is either colored red or black. It has same properties of binary search tree and has the following additional properties.

i) The root and all external nodes are in black color.
ii) No two red nodes occur consecutively on any path from root to external nodes.
iii) The number of black nodes are same on any path from root to external node.

Another definition arises from assigning colors to the pointers between a node and its children. The pointer from a parent to a black node is black and to a red node is red. The additional properties for this are:

i) The pointers from an internal node to external nodes are black.
ii) No two red pointers occur consecutively on any path from root to external node.
iii) The number black pointers are same on any path.

Note: If node colors are known, then deduce pointer colors and vice versa.



: **Red black tree example**

The above tree is an example for red black tree because it obeys red black tree properties and also it is an binary search tree.

(i) The root with key 65 and external nodes are in black color.
(ii) No two red nodes occur consecutively on any path from root to external node.
(iii) The number of black nodes is same on any path from root to external nodes i.e 2.

In the above figure, the external nodes are shaded squares, root and black nodes are shaded circles, red nodes are unshaded circles, black pointers are thick lines and red pointers are thin lines.

The rank of a node in red black tree is the number of black pointers or number of black nodes minus 1. The rank of a root (key 65) is 2, the rank of its left tree is 2, the rank of its right tree is 1.

Lemma 1: Let h be height of a red black tree excluding external nodes, n be the number of internal nodes in the tree, and r be the rank of the root.

(i) $h \leq 2r$
(ii) $n \geq 2^r - 1$
(iii) $h \leq 2 \log_2 (n+1)$

Lemma2: Let length of a path from root to external node be number of pointers on the path. If P and Q are two paths from root to external node in a red black tree, then length(P) ≤ length(Q).

- Representation of a Red Black tree: The node structure of a red black tree is same as the node structure employed for binary search tree but an extra field is included to represent the node color. There are two schemes used to represent a node. **One stores the color of the node in addition to its left, right pointers and its data. Another uses the color of its two children in addition to data, left and right pointers.**

3.10.1 Searching for a node in red black tree: The search for a key in red black tree is same as search operation of binary search tree. The time complexity of search is **O(log n).**

Insertion: Inserting an element into a red black tree is same as the method employed for binary search trees. The only concern is determines which color the node must set to. Suppose the new node set to black, the path from root to that node has one extra black node that violates black condition. The alternative is set the node to red also violates red condition. This imbalance set right to balance using rotations.

Suppose u be the newly red node inserted and p-u be its consecutive red node which is parent of u and also has grand parent named gp-u which is a black node. Depending on position of u relative to p-u,gp-u and also gp-u other child color, the imbalances are classified **as LLb, LLr, LRb, LRr, RRb, RRr, RLb and RLr**.

3.10.2 deletion: The deletion in red black tree is same as the method used for binary search tree such as deleting a leaf, a node that has single sub tree or a node that have two sub trees. If the deletion results imbalance that calls the rotations.

If the deleted node is red, there is no problem with this. Suppose deleted node is black, then the specific path in which black node is deleted have shortage of one black node. It violates black condition which results imbalance red black tree. This imbalance is classified as L or R based on whether deleted node v occurs to the left or right of its parent node p-v.

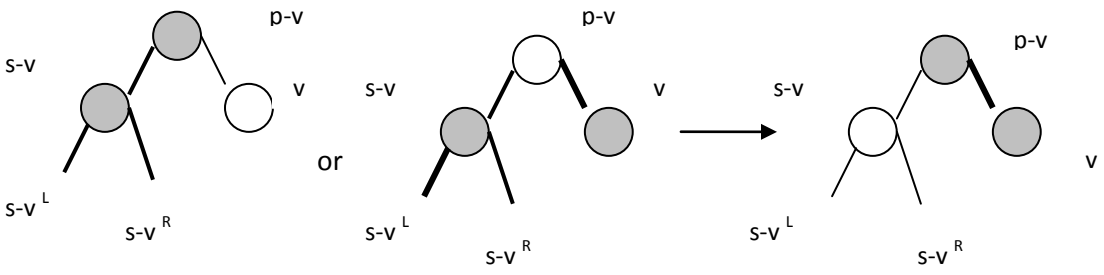
(i) If the sibling node s-v is black, then imbalance is classified as Lb or Rb. Based on whether s-v has 0 or 1 0r 2 red children, Lb and Rb imbalances are further classified as Lb0,Lb1,Lb2 and Rb0,Rb1 and Rb2 respectively.

(ii) If the s-v is a red node, then the imbalance is classified as Lr or Rr. Based on whether s-v has 0 or 1 or 2 red children, Lr and Rr imbalances are further classified as Lr0,Lr1 & Lr2 and Rr0,Rr1 & Rr2.

During rebalancing, the deleted node v is replaced by its descend.

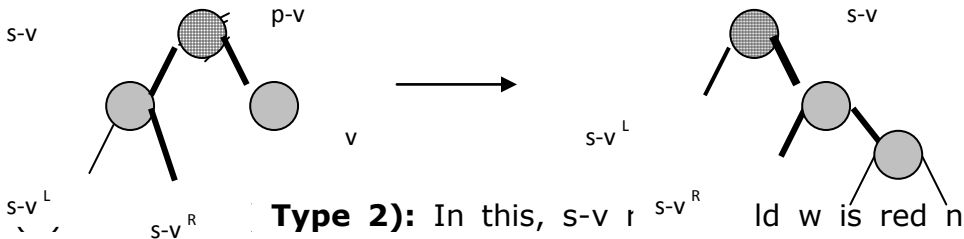
- **Rb0,Rb1 & Rb2 imbalances and their rebalancing:**

(i) **Rb0:** In this, s-v has no red Childs. After applying it, s-v color is changed to red also if p-v color is red that can be changed to black.

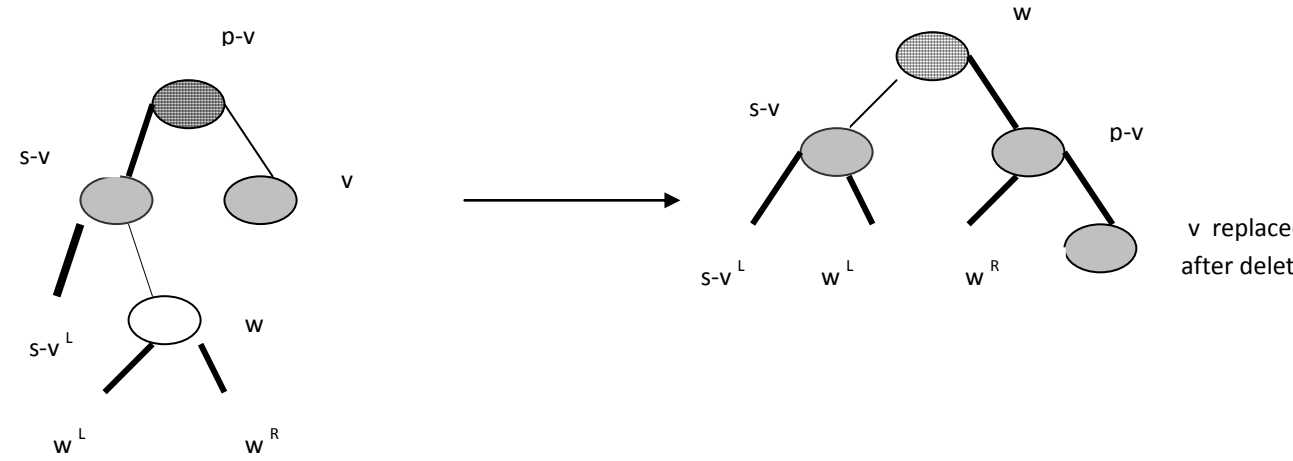


(ii) **Rb1 (Type 1**): In this, s-v ^{L} is red but s-v ^{R} is black. After applying it,

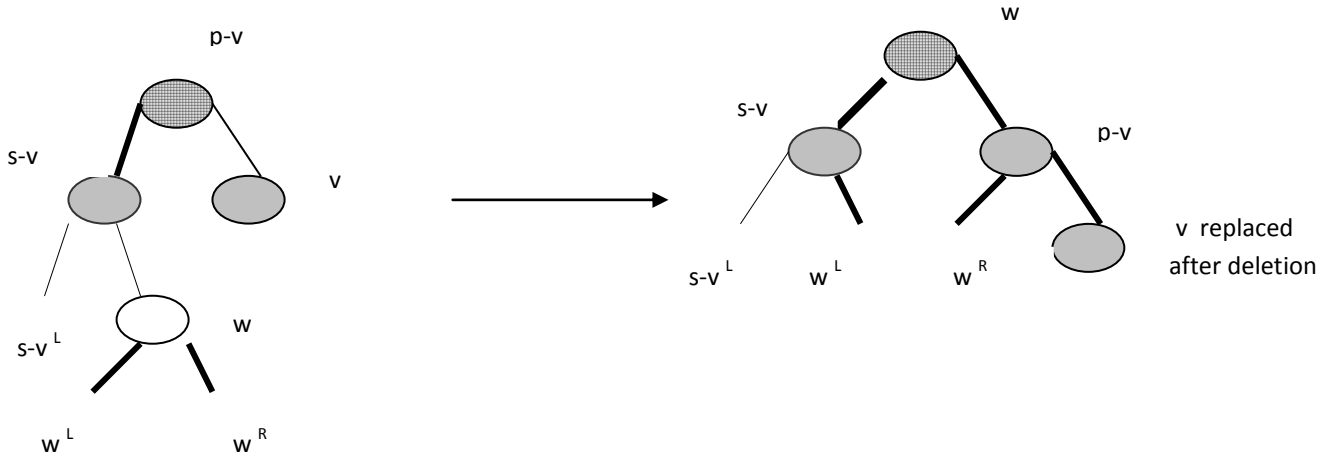
s-v moved to the root and gets root color which moves p-v downwards in its right child direction.



Type 2): In this, s-v r ld w is red node. After applying it, w moved to the root and gets root color which moves p-v downwards in its right child direction.



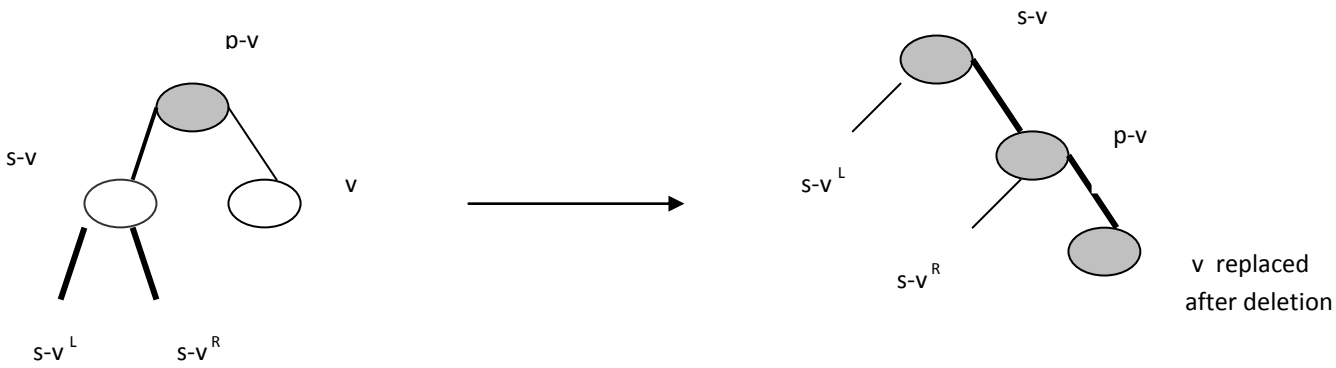
(iv) Rb2: In this, s-v left child s-v L and right child w are both red nodes. After applying it, w moves to the root and gets root color which moves p-v downwards in its right child direction.



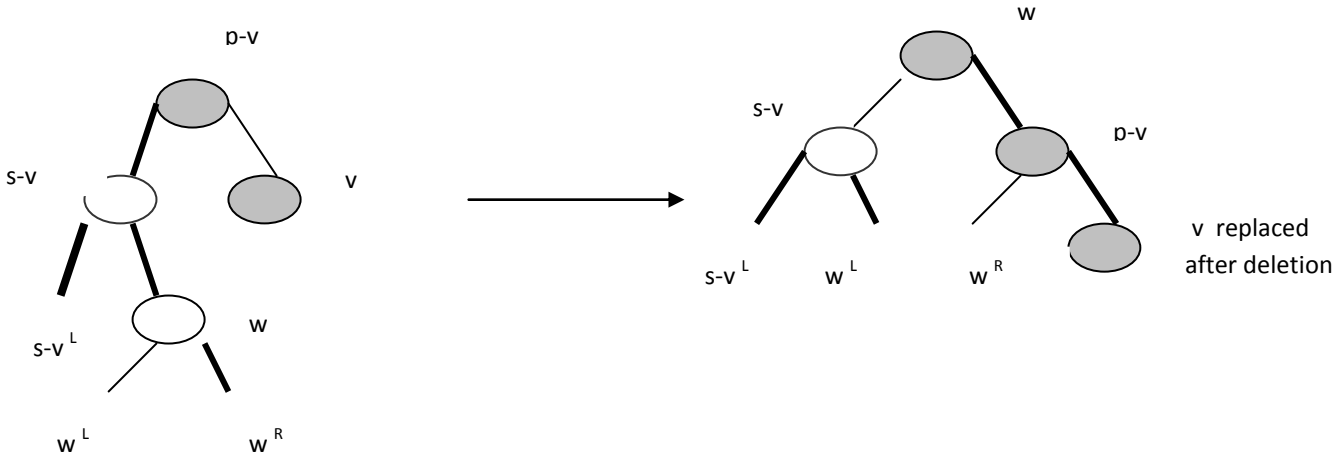
Note: p-v node is of color green can be either black or red that depends in the problem. In the above, the no indicates either black or red.

- Rr0, Rr1 & Rr2 imbalances:

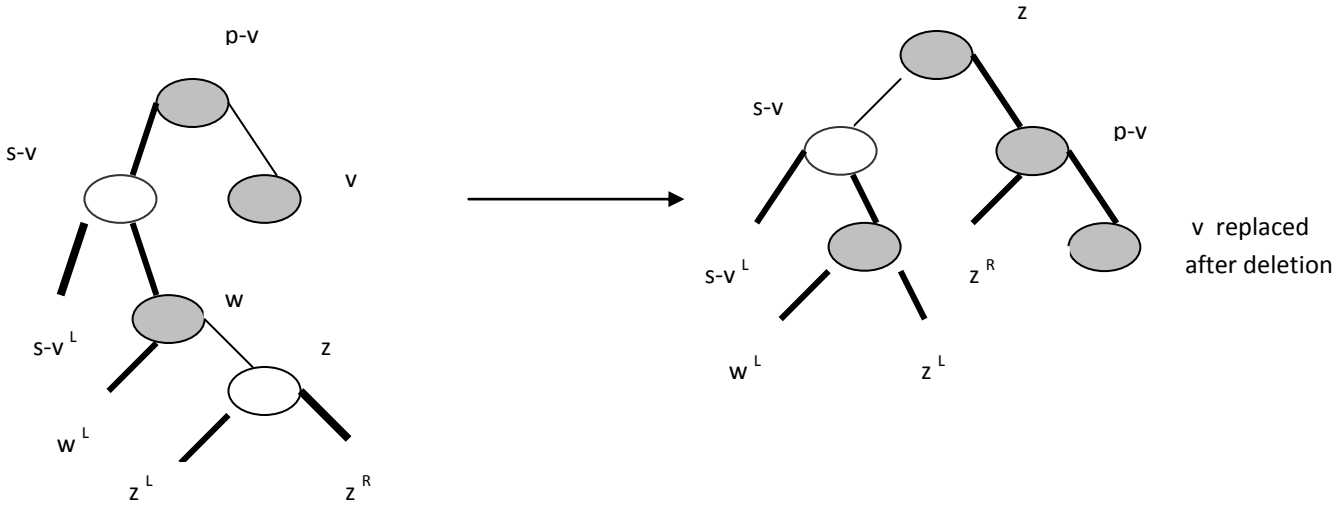
(i) Rr0: In this, s-v moved to root and gets root color which moves p-v downwards in its right child direction.



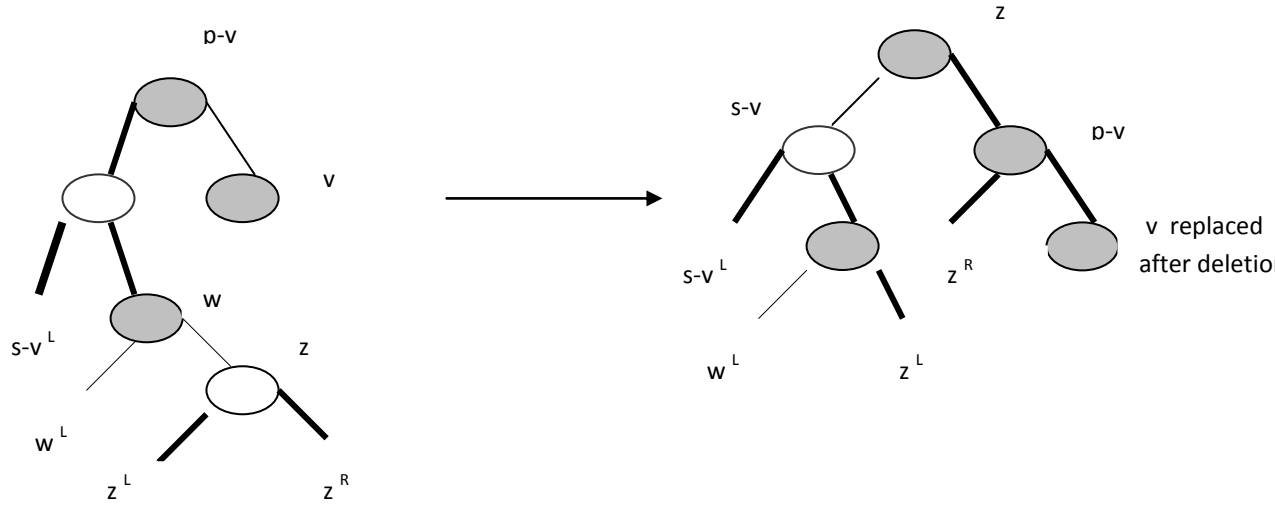
(ii) Rr1 (Type 1): In this, s-v right child w's left child w ^{L} is red node. After applying it, w moved to the root and gets root color which moves p-v downwards in its right child direction.



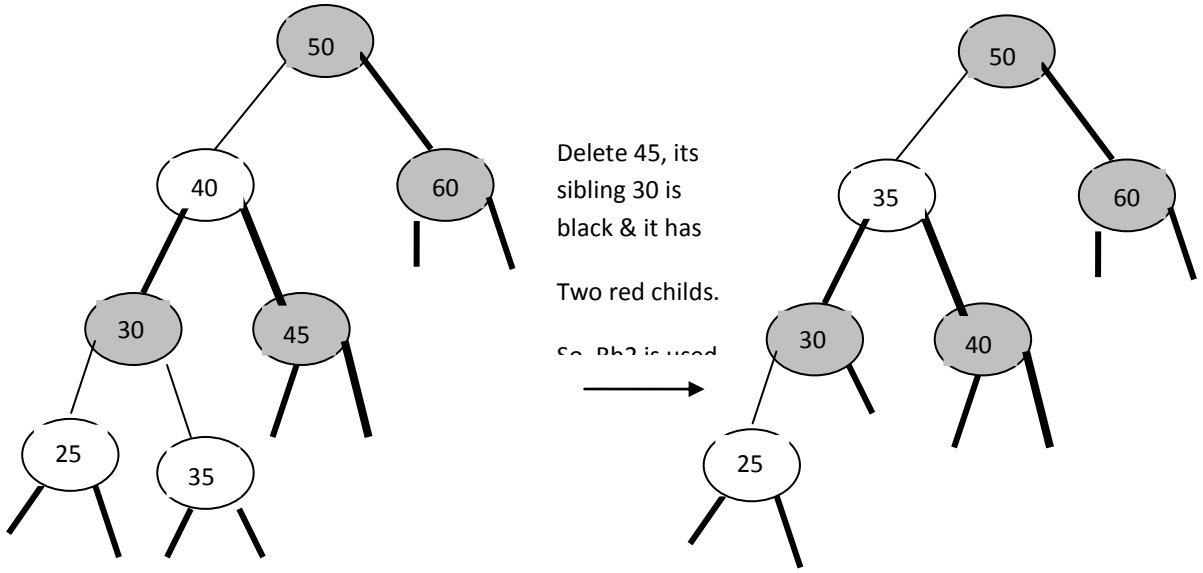
(iii) Rr1 (Type 2): In this, w's right child is red node. After applying it, z moved to root and gets root color which also moves p-v downwards in its right child direction.



(iv) **Rr2:** In this, w have w^L and z as red nodes. After applying this, z moved to the root and gets root color which also moves p-v downwards in tis right child direction.



Example for Rb2:



We can take others as examples which suite to the respective rotations and apply it.

GRAPHS:

3.11 Introduction

Graphs are a fundamental data structure in the world of programming, and this is no less so on topcoder. Usually appearing as the hard problem in Division 2, or the medium or hard problem in Division 1, there are many different forms solving a graph problem can take. They can range in difficulty from finding a path on a 2D grid from a start location to an end location, to something as hard as finding the maximum amount of water that you can route through a set of pipes, each of which has a maximum capacity (also known as the maximum-flow minimum-cut problem – which we will discuss later). Knowing the correct data structures to use with graph problems is critical. A problem that appears intractable may prove to be a few lines with the proper data structure, and luckily for us the standard libraries of the languages used by topcoder help us a great deal here!

- Define a graph *G = (V, E)* by defining a pair of sets:
    1. V = a set of **vertices**
    2. E = a set of **edges**

3.12 Graph Terminology

- Edges:
    - Each edge is defined by a pair of vertices
    - An edge **connects** the vertices that define it
    - In some cases, the vertices can be the same

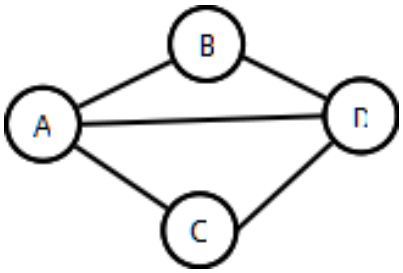
  Vertices:
    - Vertices also called **nodes**
    - Denote vertices with labels

  Representation:
    - Represent vertices with circles, perhaps containing a label

- Represent edges with lines between circles

Example:

- V = {A,B,C,D}
- E = {(A,B),(A,C),(A,D),(B,D),(C,D)}

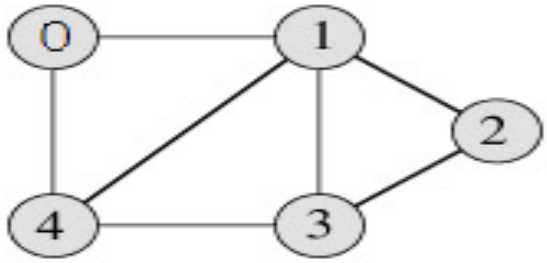


3.13 Representation of Graphs

Graph is a data structure that consists of following two components: **1.** A finite set of vertices also called as nodes. **2.** A finite set of ordered pair of the form (u, v) called as edge. The pair is ordered because (u, v) is not same as (v, u) in case of directed graph(di-graph). The pair of form (u, v) indicates that there is an edge from vertex u to vertex v. The edges may contain weight/value/cost.

Graphs are used to represent many real life applications: Graphs are used to represent networks. The networks may include paths in a city or telephone network or circuit network. Graphs are also used in social networks like linkedIn, facebook. For example, in facebook, each person is represented with a vertex(or node). Each node is a structure and contains information like person id, name, gender and locale. This can be easily viewed by http://graph.facebook.com/barnwal.aashish where barnwal.aashish is the profile name.

Following is an example undirected graph with 5 vertices.



Following two are the most commonly used representations of graph.

1. Adjacency Matrix

2. Adjacency List

There are other representations also like, Incidence Matrix and Incidence List. The choice of the graph representation is situation specific. It totally depends on the type of operations to be performed and ease of use.

Adjacency Matrix:

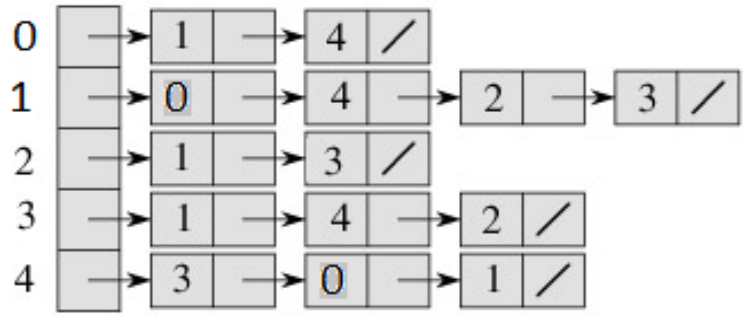
Adjacency Matrix is a 2D array of size V x V where V is the number of vertices in a graph. Let the 2D array be adj[][], a slot adj[i][j] = 1 indicates that there is an edge from vertex i to vertex j. Adjacency matrix for

undirected graph is always symmetric. Adjacency Matrix is also used to represent weighted graphs. If adj[i][j] = w, then there is an edge from vertex i to vertex j with weight w.
The adjacency matrix for the above example graph is:

	0	1	2	3	4
0	0	1	0	0	1
1	1	0	1	1	1
2	0	1	0	1	0
3	0	1	1	0	1
4	1	1	0	1	0

Adjacency Matrix Representation of the above graph
Pros: Representation is easier to implement and follow. Removing an edge takes O(1) time. Queries like whether there is an edge from vertex 'u' to vertex 'v' are efficient and can be done O(1).
Cons: Consumes more space O(V^2). Even if the graph is sparse(contains less number of edges), it consumes the same space. Adding a vertex is O(V^2) time.

Adjacency List:
An array of linked lists is used. Size of the array is equal to number of vertices. Let the array be array[]. An entry array[i] represents the linked list of vertices adjacent to the ***i***th vertex. This representation can also be used to represent a weighted graph. The weights of edges can be stored in nodes of linked lists. Following is adjacency list representation of the above graph.



Adjacency List Representation of the above Graph

3.14 Operations on Graphs

Graph Traversals:

In computer science, **graph traversal** is the problem of visiting all the nodes in a graph in a particular manner, updating and/or checking their values along the way. Tree traversal is a special case of graph traversal.

Depth-first search

A depth-first search (DFS) is an algorithm for traversing a finite graph. DFS visits the child nodes before visiting the sibling nodes; that is, it traverses the depth of any particular path before exploring its breadth. A stack (often the program's call stack via recursion) is generally used when implementing the algorithm.

The algorithm begins with a chosen "root" node; it then iteratively transitions from the current node to an adjacent, unvisited node, until it can no longer find an unexplored node to transition to from its current location. The algorithm then backtracks along previously visited nodes, until it finds a node connected to yet more uncharted territory. It will then proceed down the new path as it had before, backtracking as it encounters dead-ends, and ending only when the algorithm has backtracked past the original "root" node from the very first step.

```
procedure DFS(G,v):

        label v as explored
        for all edges e in G.incidentEdges(v) do
                if edge e is unexplored then
                    w ← G.adjacentVertex(v,e)
                    if vertex w is unexplored then
                            label e as a discovered edge
                            recursively call DFS(G,w)
                    else
                            label e as a back edge
```

Breadth-first search

A breadth-first search (BFS) is another technique for traversing a finite graph. BFS visits the neighbor nodes before visiting the child nodes, and a queue is used in the search process. This algorithm is often used to find the shortest path from one node to another.

```
 procedure BFS(G,v):
    create a queue Q
    enqueue v onto Q
    mark v
    while Q is not empty:
       t ← Q.dequeue()
       if t is what we are looking for:
          return t
       for all edges e in G.adjacentEdges(t) do
         o ← G.adjacentVertex(t,e)
         if o is not marked:
             mark o
             enqueue o onto Q
   return null
```

3.15 Applications of Graph

1) Shortest Path:

In graph theory, the **shortest path problem** is the **problem** of finding a **path** between two vertices (or nodes) in a graph such that the sum of the weights of its constituent edges is minimized.

This is probably the most often used algorithm. It may be applied in situations where the shortest path between 2 points is needed. Examples of such applications would be:

- Computer games - finding the best/shortest route from one point to another.
- Maps - finding the shortest/cheapest path for a car from one city to another, by using given roads.
- May be used to find the fastest way for a car to get from one point to another inside a certain city. E.g. satellite navigation system that shows to drivers which way they should better go.

Minimal Spanning Tree:

- Consider some communications stations (for telephony, cable television, Internet etc.) and a list of possible connections between them, having different costs. Find the cheapest way to connect these stations in a network, so that a station is connected to any other (directly, or through intermediate stations). This may be used for example to connect villages to cable television, or to Internet.
- The same problem, but instead of connecting communications stations - villages are to be connected with roads.

Eulerian Path/Circuit:

- A postman has to visit a set of streets in order to deliver mails and packages. It is needed to find a path that starts and ends at the post-office, and that passes through each street (edge) exactly once. This way the postman will deliver mails and packages to all streets he has to, and in the same time will spend minimum efforts/time for the road.

Floyd's Algorithm

- This is an $O(n^3)$ algorithm, where n is the number of vertices in the digraph.
- Uses the principle of Dynamic Programming
- Let $V = \{1, 2,..., n\}$. The algorithm uses a matrix $A[1..n][1..n]$ to compute the lengths of the shortest paths.
- Initially, Dijkstra's algorithm

```
void Floyd (floatC[n - 1][n - 1], A[n - 1][n - 1])
    { int i, j, k;
      { for(i = 0, i ≤ n - 1;i + +)
          for(j = 0;j ≤ n - 1, j + +)
            A[i, j] = C[i, j];
```

```
                    ≤
        for(i = 0;i     n - 1;i + +)
            A[i, i] = 0;
                          ≤
              for(k = 0;k     n - 1;k + +);
                  {
                                ≤
                       for(i = 0;i     n - 1;i + +)
                            {
                                      ≤
                                for(j = 0;j     n - 1, j + +)
                                   if(A[i, k] + A[k, j] < A[i, j])
                                    A[i, j] = A[i, k] + A[k, j]
                            }
                   }
              }
}
```

Dijkstra's algorithm is an algorithm for finding the shortest paths between nodes in a graph, which may represent, for example, road networks. It was conceived by computer scientist Edsger W. Dijkstra in 1956 and published three years later.[1][2]

The algorithm exists in many variants; Dijkstra's original variant found the shortest path between two nodes,[2] but a more common variant fixes a single node as the "source" node and finds shortest paths from the source to all other nodes in the graph, producing a shortest-path tree.

```
1  function Dijkstra(Graph, source):
 2
 3      create vertex set Q
 4
 5      for each vertex v in Graph:            // Initialization
 6          dist[v] ← INFINITY                 // Unknown distance from source to v
 7          prev[v] ← UNDEFINED                // Previous node in optimal path
from source
 8          add v to Q                         // All nodes initially in Q (unvisited
nodes)
 9
10      dist[source] ← 0                       // Distance from source to source
11
12      while Q is not empty:
13          u ← vertex in Q with min dist[u]   // Source node will be selected
first
14          remove u from Q
15
16          for each neighbor v of u:          // where v is still in Q.
17              alt ← dist[u] + length(u, v)
```

```
18            if alt < dist[v]:               // A shorter path to v has been found
19                dist[v] ← alt
20                prev[v] ← u
21
22        return dist[], prev[]
```

Unit-4

Sorting:

Sorting is nothing but storage of data in sorted order, it can be in ascending or descending order. The term Sorting comes into picture with the term Searching. There are so many things in our real life that we need to search, like a particular record in database, roll numbers in merit list, a particular telephone number, any particular page in a book etc.

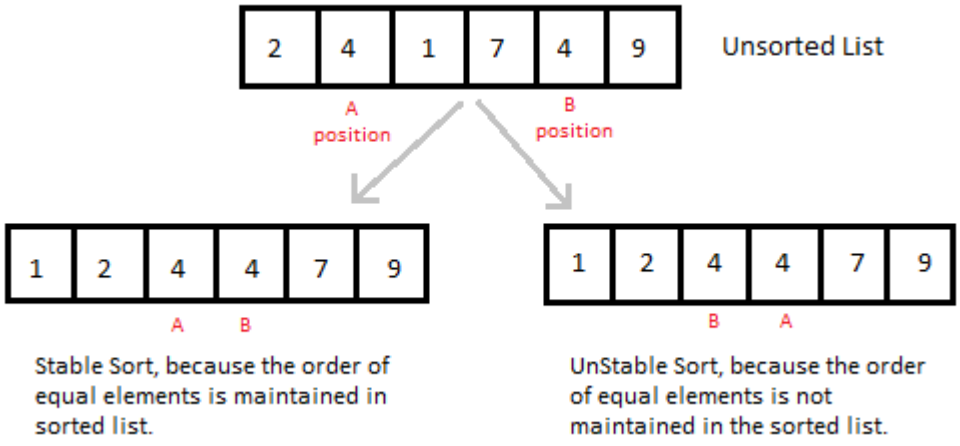
Sorting arranges data in a sequence which makes searching easier. Every record which is going to be sorted will contain one key. Based on the key the record will be sorted. For example, suppose we have a record of students, every such record will have the following data:

- Roll No.
- Name
- Age
- Class

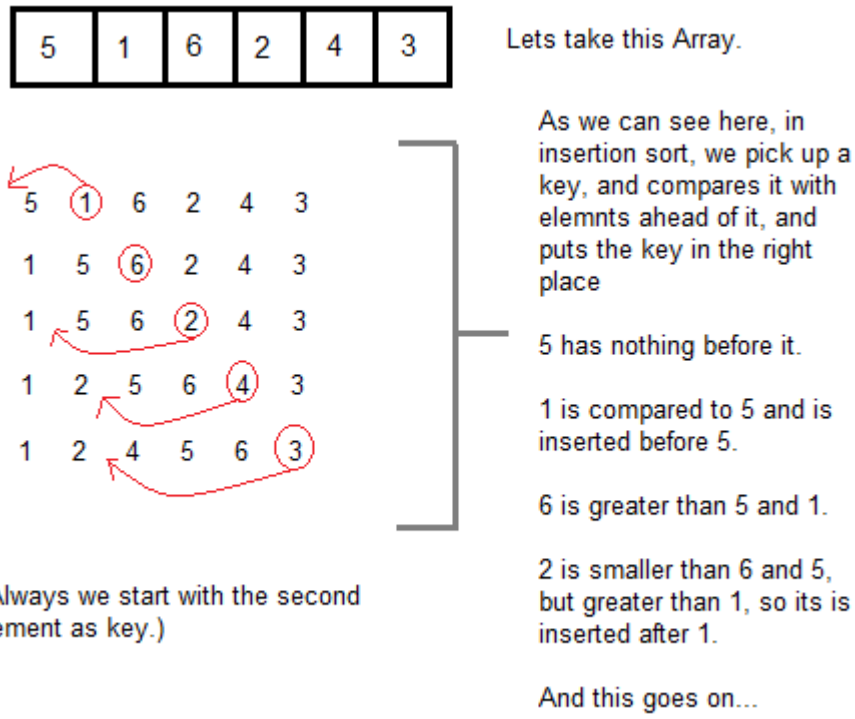
Insertion Sorting

It is a simple Sorting algorithm which sorts the array by shifting elements one by one. Following are some of the important characteristics of Insertion Sort.

1. It has one of the simplest implementation
2. It is efficient for smaller data sets, but very inefficient for larger lists.
3. Insertion Sort is adaptive, that means it reduces its total number of steps if given a partially sorted list, hence it increases its efficiency.
4. It is better than Selection Sort and Bubble Sort algorithms.
5. Its space complexity is less, like Bubble Sorting, inerstion sort also requires a single additional memory space.
6. It is **Stable**, as it does not change the relative order of elements with equal keys



How Insertion Sorting Works



Sorting using Insertion Sort Algorithm

```
int a[6] = {5, 1, 6, 2, 4, 3};
int i, j, key;
for(i=1; i<6; i++)
{
  key = a[i];
  j = i-1;
  while(j>=0 && key < a[j])
  {
    a[j+1] = a[j];
    j--;
  }
  a[j+1] = key;
}
```

Now lets, understand the above simple insertion sort algorithm. We took an array with 6 integers. We took a variable **key**, in which we put each element of the array, in each pass, starting from the second element, that is **a[1]**.
Then using the while loop, we iterate, until **j** becomes equal to zero or we find an element which is greater than **key**, and then we insert the key at that position.
In the above array, first we pick 1 as key, we compare it with 5(element before 1), 1 is smaller than 5, we shift 1 before 5. Then we pick 6, and compare it with 5 and 1, no shifting this time. Then 2 becomes the key and is compared with, 6 and 5, and then 2 is placed after 1. And this goes on, until complete array gets sorted.

Complexity Analysis of Insertion Sorting

Worst Case Time Complexity : $O(n^2)$
Best Case Time Complexity : O(n)

Average Time Complexity : $O(n^2)$
Space Complexity : O(1)

Straight insertion sort:

Straight Insertion Sort Shaded section of array contains original elements in this section, now rearranged in ascending order. Boxed element is element to be inserted, so it becomes part of shaded area at the next step. Unshaded unboxed section of array contains the original elements, completely untouched.

Input: An array A with element type T, and integers p and r with lowerbound(A) ≤ p ≤ r ≤ upperbound(A).

Output: The array A, with A[p..r] sorted, and any remaining elements of A unchanged.

Algorithm (implemented using exchanges)

```
void straight_insertion_sort( T[] A, Integer p, Integer r)
for ( i = p+1, p+2, ..., r )          // Insert A[i] into already
 j = i;                               // sorted subarray A[p..i–1].
while ( j > p and A[j–1] > A[j] )
swap(A[j–1], A[j]); j = j–1;
```

Algorithm (implemented using moves)

```
void straight_insertion_sort( T[] A, Integer p, Integer r)
for ( i = p+1, p+2, ..., r )              // Insert A[i] into already
 temp = A[i];                            // sorted subarray A[p..i–1].
j = i;
 while ( j > p and A[j–1] > temp )
 A[j] = A[j–1]; j = j–1;
 A[j] = temp;
```

Straight Selection Sorting

Selection sorting refers to a class of algorithms for sorting a list of items using comparisons. These algorithms select successively smaller or larger items from the list and add them to the output sequence. Straight selection is an $O(n^2)$ member of the class in which we repeatedly scan the remaining unsorted items in the list in linear time, find the largest (or smallest), and add it to the result. Since it looks at all unprocessed data on each pass, it performs no better for nearly sorted inputs than for random or reverse-sorted inputs, unlike insertion sort.

Binary Insertion Sort:

We can use binary search to reduce the number of comparisons in normal insertion sort. Binary Insertion Sort find use binary search to find the proper location to insert the selected item at each iteration. In normal insertion, sort it takes O(i) (at ith iteration) in worst case. we can reduce it to O(logi) by using binary search.

Time Complexity: The algorithm as a whole still has a running worst case running time of $O(n^2)$ because of the series of swaps required for each insertion.

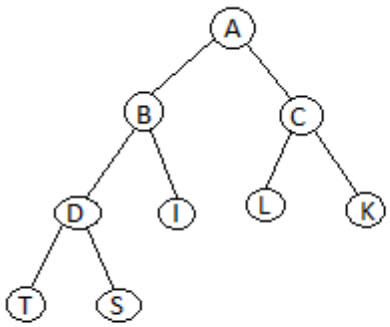
Heap sort:

Heap Sort is one of the best sorting methods being in-place and with no quadratic worst-case scenarios. Heap sort algorithm is divided into two basic parts :

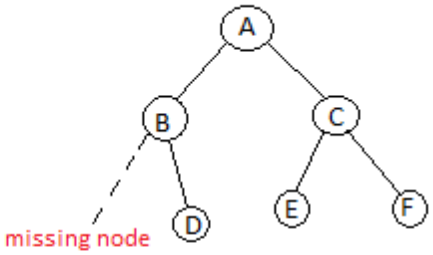
- Creating a Heap of the unsorted list.
- Then a sorted array is created by repeatedly removing the largest/smallest element from the heap, and inserting it into the array. The heap is reconstructed after each removal.

Heap is a special tree-based data structure, that satisfies the following special heap properties :

- **Shape Property :** Heap data structure is always a Complete Binary Tree, which means all levels of the tree are fully filled.

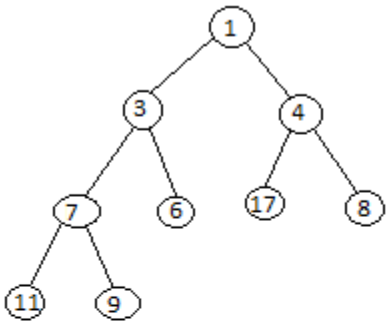


Complete Binary Tree



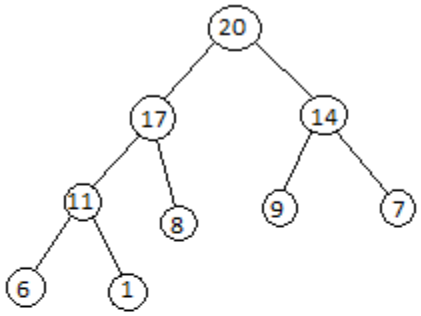
In-Complete Binary Tree

1. **Heap Property :** All nodes are either *[greater than or equal to]* or *[less than or equal to]* each of its children. If the parent nodes are greater than their children, heap is called a **Max-Heap**, and if the parent nodes are smalled than their child nodes, heap is called **Min-Heap**.



Min-Heap

In min-heap, first element is the smallest. So when we want to sort a list in ascending order, we create a Min-heap from that list, and picks the first element, as it is the smallest, then we repeat the process with remaining elements.



Max-Heap

In max-heap, the first element is the largest, hence it is used when we need to sort a list in descending order.

How Heap Sort Works

Initially on receiving an unsorted list, the first step in heap sort is to create a Heap data structure(Max-Heap or Min-Heap). Once heap is built, the first element of the Heap is either largest or smallest(depending upon Max-Heap or Min-Heap), so we put the first element of the heap in our array. Then we again make heap using the remaining elements, to again pick the first element of the heap and put it into the array. We keep on doing the same repeatedly untill we have the complete sorted list in our array.

In the below algorithm, initially **heapsort()** function is called, which calls **buildheap()** to build heap, which inturn uses **satisfyheap()** to build the heap.

Complexity Analysis of Heap Sort

Worst Case Time Complexity : O(n log n)
Best Case Time Complexity : O(n log n)
Average Time Complexity : O(n log n)
Space Complexity **:** O(n)

- Heap sort is not a Stable sort, and requires a constant space for sorting a list.
- Heap Sort is very fast and is widely used for sorting

Sorting by Exchange:

The exchange sort is almost similar as the bubble sort. In fact some people refer to the exchange sort as just a different bubble sort. (When they see the source they even call it a bubble sort instead of its real name exchange sort.)

The exchange sort compares each element of an array and swap those elements that are not in their proper position, just like a bubble sort does. The only difference between the two sorting algorithms is the manner in which they compare the elements.
The exchange sort compares the first element with each element of the array, making a swap where is necessary.
In some situations the exchange sort is slightly more efficient than its counter part the bubble sort. The bubble sort needs a final pass to determine that it is finished, thus is slightly less efficient than the exchange sort, because the exchange sort doesn't need a final pass.

BUBBLE SORT

Bubble sort is a simple sorting algorithm. This sorting algorithm is comparison based algorithm in which each pair of adjacent elements is compared and elements are swapped if they are not in order. This algorithm is not suitable for large data sets as its average and worst case complexity are of $O(n^2)$ where n are no. of items.

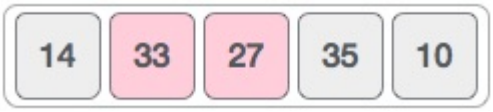
How bubble sort works?

We take an unsorted array for our example. Bubble sort take $O(n^2)$ time so we're keeping short and precise.

Bubble sort starts with very first two elements, comparing them to check which one is greater.

In this case, value 33 is greater than 14, so it is already in sorted locations. Next, we compare 33 with 27.

We find that 27 is smaller than 33 and these two values must be swapped.



The new array should look like this –

Next we compare 33 and 35. We find that both are in already sorted positions.

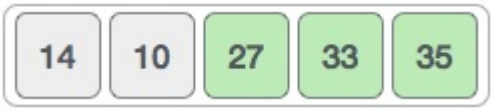
Then we move to next two values, 35 and 10.

We know than 10 is smaller 35. Hence they are not sorted.

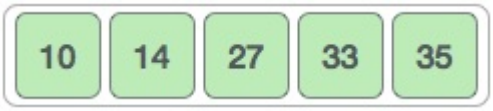
We swap these values. We find that we reach at the end of the array. After one iteration the array should look like this –

To be precise, we are now showing that how array should look like after each iteration. After second iteration, it should look like this –

Notice that after each iteration, at least one value moves at the end.



And when there's no swap required, bubble sorts learns that array is completely sorted.



Now we should look into some practical aspects of bubble sort.

Algorithm

We assume **list** is an array of **n** elements. We further assume that **swap**function, swaps the values of given array elements.

```
begin BubbleSort(list)
   for all elements of list
      if list[i] > list[i+1]
         swap(list[i], list[i+1])
      end if
   end for
      return list
   end BubbleSort
```

Shell Sort:

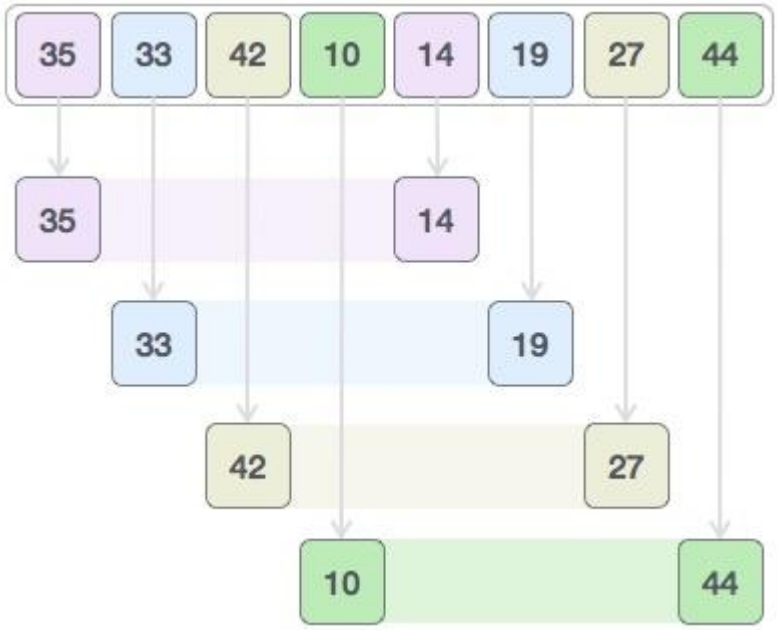
Shell sort is a highly efficient sorting algorithm and is based on insertion sort algorithm. This algorithm avoids large shifts as in case of insertion sort if smaller value is very far right and have to move to far left.

This algorithm uses insertion sort on widely spread elements first to sort them and then sorts the less widely spaced elements. This spacing is termed as interval.
This interval is calculated based on Knuth's formula as −

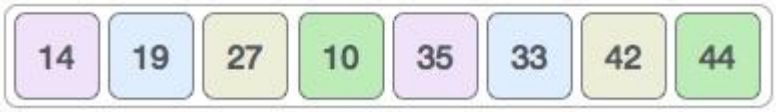
h = h * 3 + 1
where −
h is interval with initial value 1

This algorithm is quite efficient for medium sized data sets as its average and worst case complexity are of O(n) where n are no. of items.

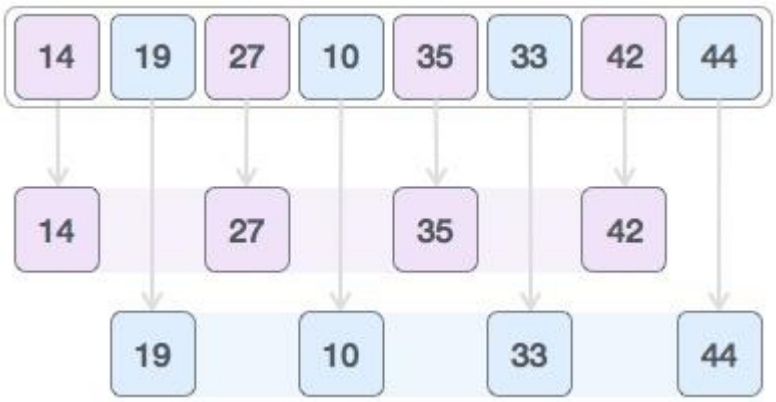
How shell sort works
We take the below example to have an idea, how shell sort works? We take the same array we have used in our previous examples. For our example and ease of understanding we take the interval of 4. And make a virtual sublist of all values located at the interval of 4 positions. Here these values are {35, 14}, {33, 19}, {42, 27} and {10, 14}



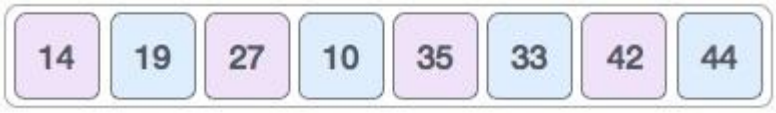
We compare values in each sub-list and swap them (if necessary) in the original array. After this step, new array should look like this −



Then we take interval of 2 and this gap generates two sublists - {14, 27, 35, 42}, {19, 10, 33, 44}



We compare and swap the values, if required, in the original array. After this step, this array should look like this –



And finally, we sort the rest of the array using interval of value 1. Shell sort uses insertion sort to sort the array. The step by step depiction is shown below –

We see that it required only four swaps to sort the rest of the array.

QUICK SORT

Quick Sort, as the name suggests, sorts any list very quickly. Quick sort is not stable search, but it is very fast and requires very less aditional space. It is based on the rule of **Divide and Conquer**(also called *partition-exchange sort*). This algorithm divides the list into three main parts :

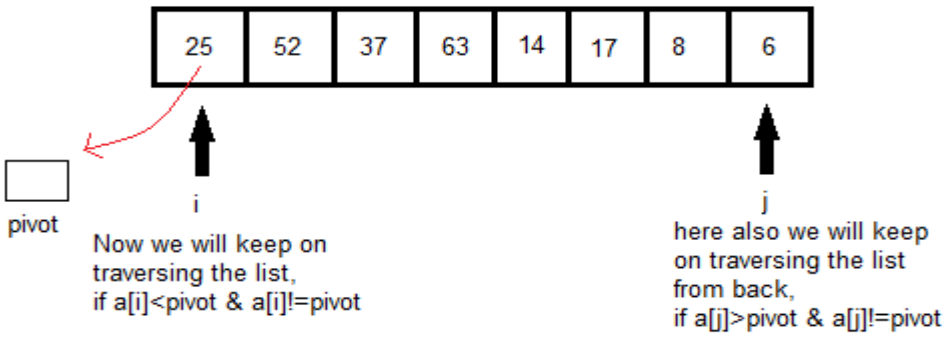
1. Elements less than the Pivot element
2. Pivot element
3. Elements greater than the pivot element

In the list of elements, mentioned in below example, we have taken **25** as pivot. So after the first pass, the list will be changed like this.

6 8 17 14 **25** 63 37 52

Hnece after the first pass, pivot will be set at its position, with all the elements smaller to it on its left and all the elements larger than it on the right. Now 6 8 17 14 and 63 37 52 are considered as two separate lists, and same logic is applied on them, and we keep doing this until the complete list is sorted.

How Quick Sorting Works



if both sides we find the element not satisfying their respective conditions, we swap them. And keep repeating this.

DIVIDE AND CONQUER - QUICK SORT

Complexity Analysis of Quick Sort

Worst Case Time Complexity : **$O(n^2)$**

Best Case Time Complexity : **O(n log n)**

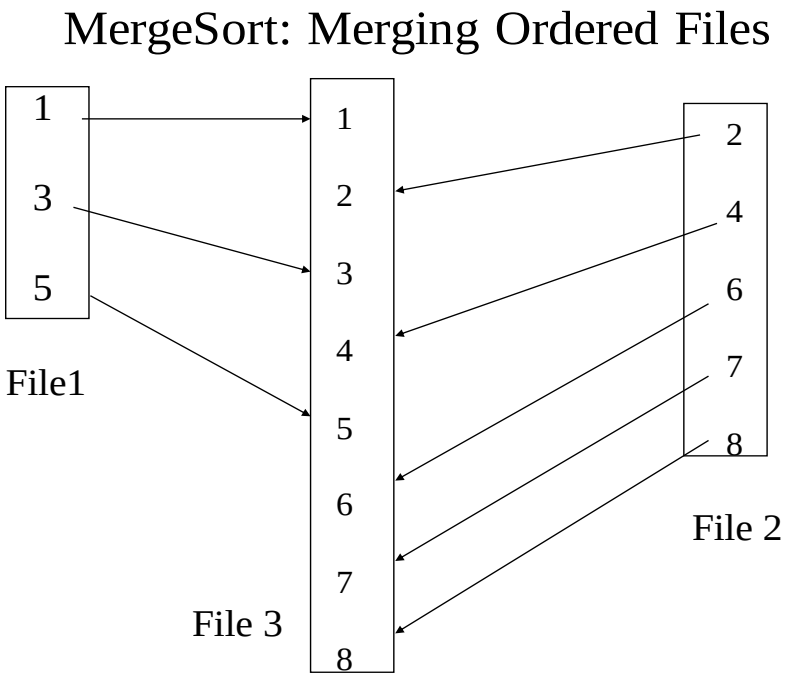
Average Time Complexity : **O(n log n)**
Space Complexity : **O(n log n)**

Merge sort:

MergeSort is a recursive sorting procedure that uses O(nlogn) comparisons in the worst case.

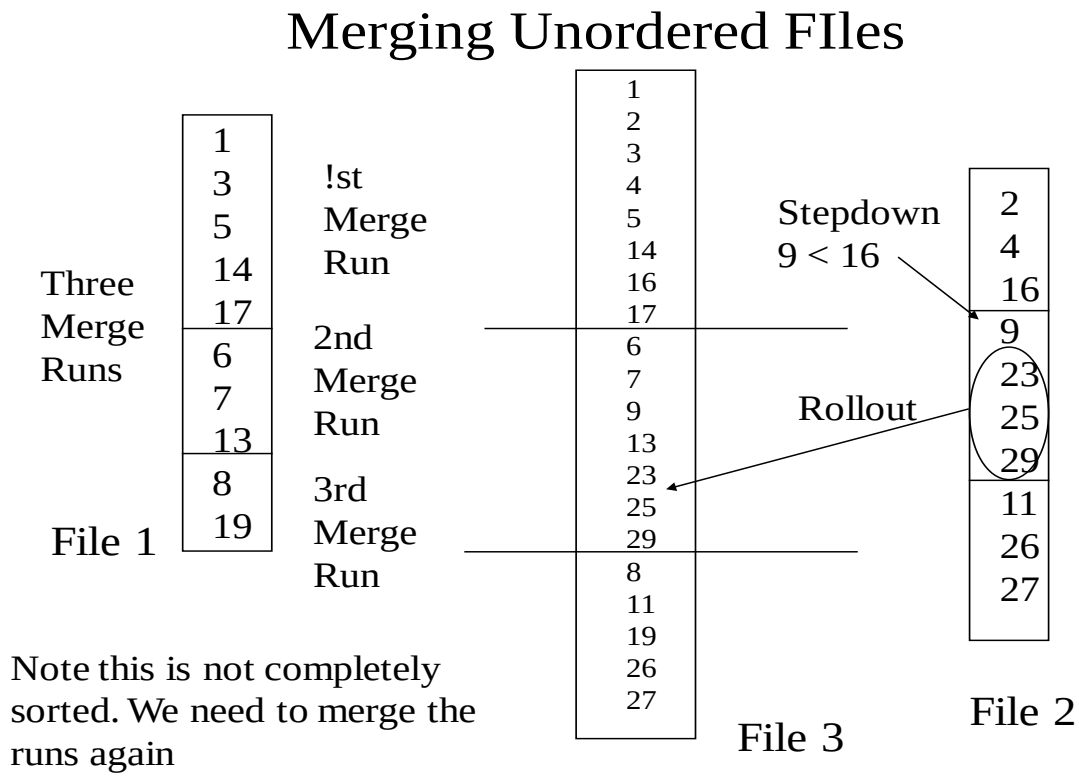
- If n < 2 then the array is already sorted. Stop.
- Otherwise, n > 1, do the following:
  1. Sort the left half of the array
  2. Sort the right half of the array
  3. Merge the sorted left and right halves.
- To sort the left half of the array, the program calls itself recursively, passing the left half of array down to a new activation of MergeSort.
- This happens repeatedly until we have an array of only one element. One-element arrays are already sorted.
- When the left half of an array is sorted, we sort the right half.
- Once the two halves of an array have been sorted by recursive calls,
- MergeSort merges the two sorted halves into a sorted whole.
- This is done by examining and comparing the smallest remaining number in each of the sorted halves.
- When one of the two subarrays becomes empty, the remaining elements in the other subarray are copied in order into parent array. No comparison need to be made.

Merging Ordered Files:



Merging Unordered Files:

- Merge Run - A series of consecutively ordered data in a file
- Stepdown - Occurs when the sequential ordering of a file is broken.
- Rollout - The process of copying the consecutive series of records to the merge output file after a stepdown**.**



The Merge Process:

- Assume we have 2300 records to sort
- We can only put 500 records at a time in main memory
- We read and sort the first 500 records, then write them to an output file.
- Repeat this proscess with the remaining chunks of data until all are processed.
- We now need a process to merge all the chunks to gether for resorting

Unit-5

Linear Search

A linear search is the basic and simple search algorithm. A linear search searches an element or value from an array till the desired element or value is not found and it searches in a sequence order. It compares the element with all the other elements given in the list and if the element is matched it returns the value index else it return -1. Linear Search is applied on the unsorted or unordered list when there are fewer elements in a list.

Example with Implementation

To search the element 5 it will go step by step in a sequence order.


```
function findIndex(values, target)
 {
  for(var i = 0; i < values.length; ++i)
    {
      if (values[i] == target)
        {
          return i;
        }
    }
  return -1;
 }
//call the function findIndex with array and number to be searched
findIndex([ 8 , 2 , 6 , 3 , 5 ] , 5) ;
```

Sequential search

Sequential Search is the most natural searching method. In this method, the searching begins with searching every element of the list till the required record is found. It makes no demands on the ordering of records. It takes considerably amount of time and is slower.

Sequential Search Algorithm

This represents the algorithm to search a list of values of to find the required one.

INPUT: List of size N. Target value T

OUTPUT: Position of T in the list I

BEGIN

1. Set FOUND to false

Set I to 0
2. While (I<=N) and (FOUND is false)
If List [I] = T
FOUND = true
Else
I=I+1
END
3. If FOUND is false
T is not present in List.
End

Analysis of Sequential Search

Whether the sequential search is carried out on lists implemented as arrays or linked lists or on files, the criterial part in performance is the comparison loop step 2. Obviously the fewer the number of comparisons, the sooner the algorithm will terminate.

The fewest possible comparisons = 1. When the required item is the first item in the list. The maximum comparisons = N when the required item is the last item in the list. Thus if the required item is in position I in the list, I comparisons are required. Hence the average number of comparisons done by sequential search is (N+1)/2

$$\frac{1+2+3.....+N}{N}$$

$$= \frac{-N\,(N+1)}{2*N}$$

$$= (N+1)/2$$

Sequential search is easy to write and efficient for short lists. It does not require sorted data. However it is disastrous for long lists. There is no way of quickly establishing that the required item is not in the list or of finding all occurrences of a required item at one place. We can overcome these deficiencies with Binary search.

Binary search: Analyzing Search Algorithm

Binary search is a fast search algorithm with run-time complexity of O(log n). This search algorithm works on the principle of divide and conquer. For this algorithm to work properly the data collection should be in sorted form.
Binary search search a particular item by comparing the middle most item of the collection. If match occurs then index of item is returned. If middle item is greater than item then item is searched in sub-array to the right of the middle item other wise item is search in sub-array to the left of the middle item. This process continues on sub-array as well until the size of subarray reduces to zero.

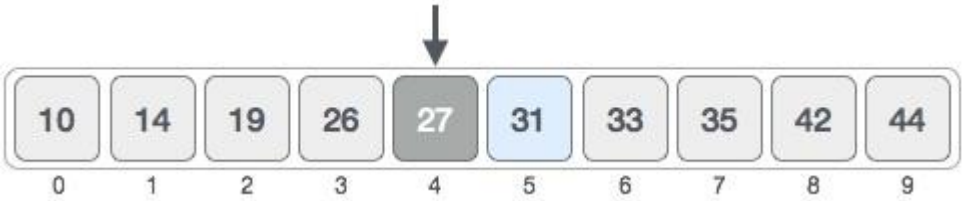
How binary search works?

For a binary search to work, it is mandatory for the target array to be sorted. We shall learn the process of binary search with an pictorial example. The below given is our sorted array and assume that we need to search location of value 31 using binary search.

First, we shall determine the half of the array by using this formula −

```
mid = low + (high - low) / 2
```

Here it is, 0 + (9 - 0) / 2 = 4 (integer value of 4.5). So 4 is the mid of array.



Now we compare the value stored at location 4, with the value being searched i.e. 31. We find that value at location 4 is 27, which is not a match. Because value is greater than 27 and we have a sorted array so we also know that target value must be in upper portion of the array.

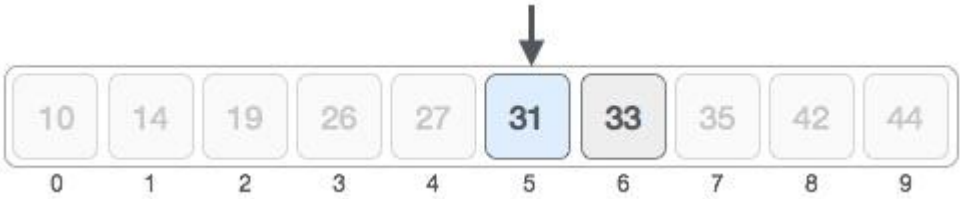
We change our low to mid + 1 and find the new mid value again.

```
low = mid + 1
mid = low + (high - low) / 2
```

Our new mid is 7 now. We compare the value stored at location 7 with our target value 31.

The value stored at location 7 is not a match, rather it is less that what we are looking for. So the value must be in lower part from this location.

So we calculate the mid again. This time it is 5.



We compare the value stored ad location 5 with our target value. We find that it is a match.

We conclude that the target value 31 is stored at location 5.

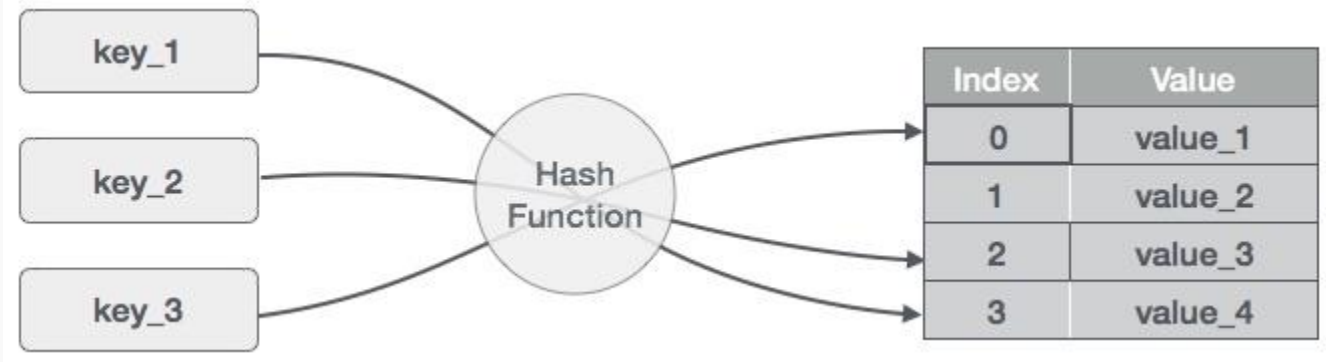
Binary search halves the searchable items and thus reduces the count of comparisons to be made to very less numbers.

Hashing

Hash Table is a data structure which store data in associative manner. In hash table, data is stored in array format where each data values has its own unique index value. Access of data becomes very fast if we know the index of desired data.

Thus, it becomes a data structure in which insertion and search operations are very fast irrespective of size of data. Hash Table uses array as a storage medium and uses hash technique to generate index where an element is to be inserted or to be located from.

Hashing is a technique to convert a range of key values into a range of indexes of an array. We're going to use modulo operator to get a range of key values. Consider an example of hashtable of size 20, and following items are to be stored. Item are in (key,value) format.



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

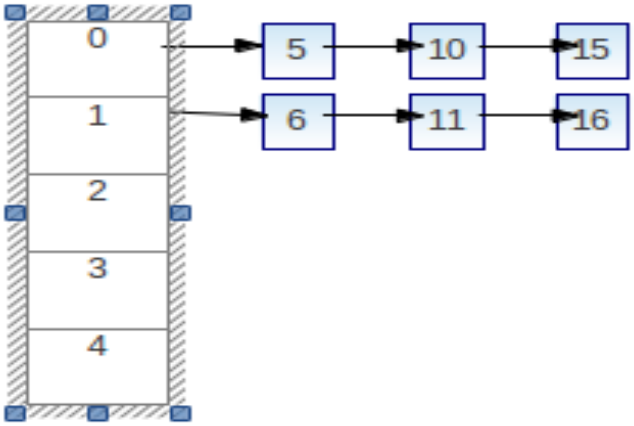
S.no	Key	Hash	Array Index
1	1	1 % 20 = 1	1
2	2	2 % 20 = 2	2

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

Linked List Collision Resolution:

We have seen hash implementation in Array , where we can fill only one value in one slot. If new value comes it overwrites previous value. But using collision resolution by linked list we can resolve this problem and preserve the values. Whenever new value comes to the slot which is already filled then we can put new value in linked list associated with that slot.

In this method performance degrades when more values are matching with same slot.



Collision

Resolutions-

✦ A collision occurs when two different keys hash to the same value

➪ E.g. For TableSize = 17, keys 18 and 35 hash to the same value

➪ 18 mod 17 = 1 and 35 mod 17 = 1
✦ Cannot store both data records in the same slot in array!
✦ Two different methods for collision resolution:
➪ Separate Chaining: Use data structure (such as a linked list) to store multiple items that hash to the same slot
➪ Open addressing (or probing): search for other slots using a second function and store item in first empty slot that is found

Collision Resolutions- Open Addressing:
✦ Linked lists can take up a lot of space
✦ Open addressing (or probing): When collision occurs, try alternative cells in the array until an empty cell is found
✦ Given an item X, try cells h0(X), h1(X), h2(X), …, hi (X)
✦ hi (X) = (Hash(X) + F(i)) mod TableSize ➪ Define F(0) = 0
✦ F is the collision resolution function. Three possibilities: ➪ Linear: F(i) = i ➪ Quadratic: F(i) = i2 ➪ Double Hashing: F(i) = iïHash2(X)

Open addressing:

Open addressing, or closed hashing, is a method of collision resolution in hash tables. With this method a hash collision is resolved by probing, or searching through alternate locations in the array (the *probe sequence*) until either the target record is found, or an unused array slot is found, which indicates that there is no such key in the table.

Bucket Hashing:

• Slots are grouped into buckets
• The hash function transforms the key into a bucket number
• Each bucket contains B slots and no collision occurs until the bucket is full. At that point you need to apply a collision processing strategy to find another bucket

Bucket Hashing Let X = X1 ...Xn be a string, partitioned into n words. To hash X using bucket hashing we will scatter the words of X into N "buckets," then XOR the contents of each bucket, and then concatenate the bucket contents.

Some ways of scattering the words of X work out better than others. In this paper we analyze a particular bucket hashing scheme, which we denote by B. The scheme will depend on parameters n, N, w. Scheme B will scatter each word into three buckets.