Module 1: Basic Concepts of Data Structures

System Life Cycle, Algorithms, Performance Analysis, Space Complexity, Time Complexity, Asymptotic Notation, Complexity Calculation of Simple Algorithms.

SYSTEM LIFE CYCLE (SLC)

- Good programmers regard large scale computer programs as systems that contain many complex interacting parts. (Systems: Large Scale Computer Programs.)
- As systems, these programs undergo a development process called System life cycle.(SLC : Development Process of Programs)

  Different Phases of System Life Cycle

  1. Requirements
  2. Analysis
  3. Design
  4. Refinement and coding
  5. Verification

1. Requirement Phase:

- All programming projects begin with a **set of specifications** that defines the purpose of that program.
- Requirements describe the information that the programmers are given (**input**) and the results (**output**) that must be produced.
- Frequently the initial specifications are defined vaguely and we must develop rigorous input and output descriptions that include all cases.

2. Analysis Phase

- In this phase the problem is break down into manageable pieces.
- There are two approaches to analysis:-**bottom up and top down**.
- Bottom up approach is an older, **unstructured** strategy that places an early emphasis on coding fine points. Since the programmer does not have a master plan for the project, the resulting program frequently has many **loosely connected, error ridden segments**.
- Top down approach is a structured approach divide the program into manageable segments.
- This phase generates **diagrams** that are used to design the system.

- Several alternate solutions to the programming problem are developed and compared during this phase

3. Design Phase

- This phase continues the work done in the analysis phase.
- The designer approaches the system from the perspectives of both **data objects** that the program needs and the **operations** performed on them.
- The first perspective leads to the **creation of abstract data types** while the second requires the **specification of algorithms** and a consideration of algorithm design strategies.

  Ex: Designing a scheduling system for university

  Data objects: Students, courses, professors etc

  Operations: insert, remove search etc

  ie. We might add a course to the list of university courses, search for the courses taught by some professor etc.
- Since abstract data types and algorithm specifications are language independent.
- We must specify the information required for each data object and ignore coding details.

  Ex: Student object should include name, phone number, social security number etc.

4. Refinement and Coding Phase

- In this phase we choose representations for data objects and write algorithms for each operation on them.
- Data objects representation can determine the efficiency of the algorithm related to it. So we should write algorithms that are independent of data objects first.
- Frequently we realize that we could have created a much better system. (May be we realize that one of our alternate design is superior than this). If our original design is good, it can absorb changes easily.

5. Verification Phase

- This phase consists of
  - developing correctness proofs for the program
  - Testing the program with a variety of input data.
  - Removing errors.

Correctness of Proofs

- Programs can be proven correct using proofs.(like mathematics theorem)
- Proofs are very time consuming and difficult to develop for large projects.
- Scheduling constraints prevent the development of complete sets of proofs for a larger system.
- However, selecting algorithm that have been proven correct can reduce the number of errors.

Testing

- Testing can be done only after coding.
- Testing requires working code and set of test data.
- Test data should be chosen carefully so that it includes all possible scenarios.
- Good test data should verify that every piece of code runs correctly.
- For example if our program contains a *switch* statement, our test data should be chosen so that we can check each *case* within *switch* statement.

Error Removal

- If done properly, the correctness of proofs and system test will indicate erroneous code.
- Removal of errors depends on the design and code.
- While debugging large undocumented program written in 'spaghetti' code, each corrected error possibly generates several new errors.
- Debugging a well documented program that is divided into autonomous units that interact through parameters is far easier. This especially true if each unit is tested separately and then integrated into system.

ALGORITHMS

Definition: An ***algorithm*** is a finite set of instructions to accomplish a particular task. In addition, all algorithms must satisfy the following criteria:

(1) ***Input***. There are zero or more quantities that are externally supplied.
(2) ***Output***. At least one quantity is produced.
(3) ***Definiteness***. Each instruction is clear and unambiguous.

(4) ***Finiteness***. If we trace out the instructions of an algorithm, then for all cases, the algorithm terminates after a finite number of steps.

(5) ***Effectiveness***. Every instruction must be basic enough to be carried out, in principle, by a person using only pencil and paper. It is not enough that each operation be definite as in (3); it also must be feasible.

We can describe algorithm in many ways

1. We can use a natural language like English
2. Graphical Representation called flow chart, but they work well only if the algorithm is small and simple.

Translating a Problem into an Algorithm

Example [Selection sort]: Suppose we must devise an algorithm that sorts a collection of n > 1 elements of arbitrary type. A simple solution is given by the following

[Selection Sort: In each pass of the selection sort, the smallest element is selected from the unsorted list and exchanged with the elements at the beginning of the unsorted list]

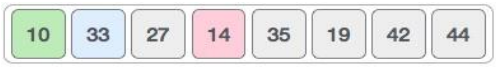
Consider the following depicted array as an example.

For the first position in the sorted list, the whole list is scanned sequentially. The first position where 14 is stored presently, we search the whole list and find that 10 is the lowest value.

So we replace 14 with 10. After one iteration 10, which happens to be the minimum value in the list, appears in the first position of the sorted list.

For the second position, where 33 is residing, we start scanning the rest of the list in a linear manner.

We find that 14 is the second lowest value in the list and it should appear at the second place. We swap these values.



After two iterations, two least values are positioned at the beginning in a sorted manner.

The same process is applied to the rest of the items in the array.

Following is a pictorial depiction of the entire sorting process −

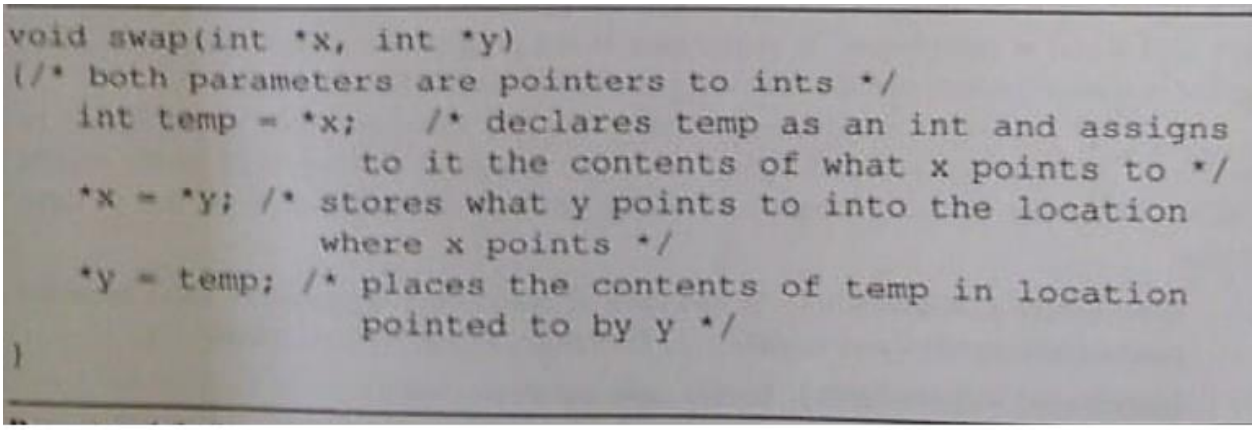
- From those elements that are currently unsorted, find the smallest and place it next in the sorted list
- We assume that the elements are stored in an array 'list', such that the i^{th} integer is stored in the i^{th} Position list[i], 0 <= i <n
- Algorithm 1.1 is our first attempt to deriving a solution

```
for (i = 0; i < n; i++) {
  Examine list[i] to list[n-1] and suppose that the
  smallest integer is  at list[min];

  Interchange list[i] and list[min];
}
```

1.1 Selection sort algorithm

- We are written this partially in C and partially in English
- To turn the program 1.1 into a real C program, two clearly defined sub tasks are remain: **finding the smallest integer** and **interchanging it with list[i].**
- We can solve this by using a function



```
void swap(int *x, int *y)
(/* both parameters are pointers to ints */
   int temp = *x;   /* declares temp as an int and assigns
                       to it the contents of what x points to */
   *x = *y; /* stores what y points to into the location
               where x points */
   *y = temp; /* places the contents of temp in location
                 pointed to by y */
}
```

1.2 Swap Function

- To swap their values one could call swap(&a, &b)
- We can solve the first subtask by assuming that the minimum is the list[i]. Checking list[i] with list[i+1], list[i+2]……,list[n-1]. Whenever we find a smaller number we make it as the minimum. We reach list[n-1] we are finished.

```
#include <stdio.h>
int main()
{
        int a[100], n, i, j, position, swap;
        printf("Enter number of elements");
        scanf("%d", &n);
        printf("Enter %d Numbersn", n);
        for (i = 0; i < n; i++)
        scanf("%d", &a[i]);
        for(i = 0; i < n - 1; i++)
        {
            position=i;
            for(j = i + 1; j < n; j++)
            {
                    if(a[position] > a[j])
                    position=j;
            }
            if(position != i)
            {
                    swap=a[i];
                    a[i]=a[position];
                    a[position]=swap;
            }
        }
        printf("Sorted Array:n");
        for(i = 0; i < n; i++)
        printf("%dn", a[i]);
        return 0;
```

}

- **Correctness Proof**

Theorem 1.1 Algorithm SelectionSort(a, n) correctly sorts a set of $n \geq 1$ elements; the result remains in $a[1:n]$ such that $a[1] \leq a[2] \leq \cdots \leq a[n]$.

Proof: We first note that for any i, say $i = q$, following the execution of lines 6 to 9, it is the case that $a[q] \leq a[r]$, $q < r \leq n$. Also observe that when i becomes greater than q, $a[1:q]$ is unchanged. Hence, following the last execution of these lines (that is, $i = n$), we have $a[1] \leq a[2] \leq \cdots \leq a[n]$.

We observe at this point that the upper limit of the **for** loop in line 4 can be changed to $n-1$ without damaging the correctness of the algorithm. □

Example 1.2 [*Binary search*]: Assume that we have $n \geq 1$ distinct integers that are already sorted and stored in the array *list*. That is, *list*[0] $\leq$ *list*[1] $\leq \cdots \leq$ *list*[n−1]. We must figure out if an integer *searchnum* is in this list. If it is we should return an index, i, such that *list*[i] = *searchnum*. If *searchnum* is not present, we should return −1. Since the list is sorted we may use the following method to search for the value.

Let *left* and *right*, respectively, denote the left and right ends of the list to be searched. Initially, *left* = 0 and *right* = n−1. Let *middle* = (*left*+*right*)/2 be the middle position in the list. If we compare *list*[*middle*] with *searchnum*, we obtain one of three results:

(1) **searchnum < list[middle]**. In this case, if *searchnum* is present, it must be in the positions between 0 and *middle* − 1. Therefore, we set *right* to *middle* − 1.

(2) **searchnum = list[middle]**. In this case, we return *middle*.

(3) **searchnum > list[middle]**. In this case, if *searchnum* is present, it must be in the positions between *middle* + 1 and n − 1. So, we set *left* to *middle* + 1.

If *searchnum* has not been found and there are still integers to check, we recalculate *middle* and continue the search. Program 1.5 implements this searching strategy. The algorithm contains two subtasks: (1) determining if there are any integers left to check, and (2) comparing *searchnum* to *list*[*middle*].

```
while (there are more integers to check ) {
   middle = (left + right) / 2;
   if (searchnum < list[middle])
      right = middle - 1;
   else if (searchnum == list[middle])
           return middle;
        else left = middle + 1;
}
```

Program 1.5: Searching a sorted list

We can handle the comparisons through either a function or a macro. In either case, we must specify values to signify less than, equal, or greater than. We will use the strategy followed in C's library functions:

- We return a negative number (−1) if the first number is less than the second.
- We return a 0 if the two numbers are equal.
- We return a positive number (1) if the first number is greater than the second.

Although we present both a function (Program 1.6) and a macro, we will use the macro throughout the text since it works with any data type.

```
int compare(int x, int y)
{/* compare x and y, return -1 for less than, 0 for equal,
    1 for greater */
   if (x < y) return -1;
   else if (x == y) return 0;
        else return 1;
}
```

Program 1.6: Comparison of two integers

The macro version is:

```
#define COMPARE(x,y) (((x) < (y)) ? -1: ((x) == (y))? 0: 1)
```

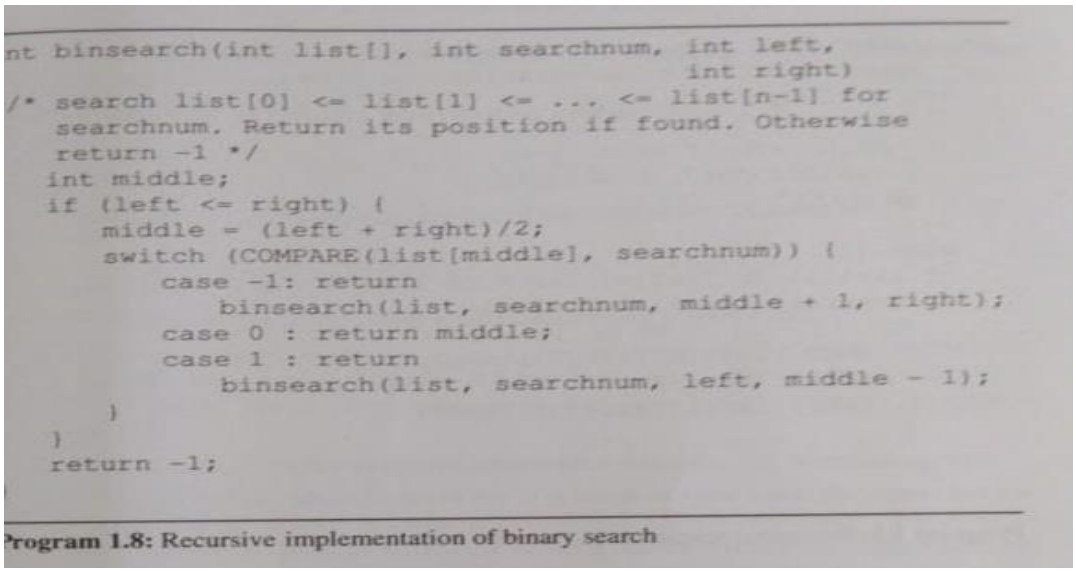
```
int binsearch(int list[], int searchnum, int left,
                                         int right)
{/* search list[0] <= list[1] <=  . . .  <= list[n-1] for
 searchnum. Return its position if found. Otherwise
 return -1 */
   int  middle;
   while (left <= right)  {
      middle = (left + right)/2;
      switch (COMPARE(list[middle], searchnum)) {
         case -1: left = middle + 1;
                  break;
         case 0 : return middle;
         case 1 : right = middle - 1;
      }
   }
   return -1;
}
```

Program 1.7: Searching an ordered list

Recursive Algorithm

- An algorithm is said to be recursive if the same algorithm is invoked in the body.
- An algorithm that calls itself is direct recursive.
- Algorithm A is said to be indirect recursive if it calls another algorithm which in turn calls A.

- These recursive mechanisms are extremely powerful, but even more importantly; many times they can express an otherwise complex process very clearly.



```
nt binsearch(int list[], int searchnum, int left,
                                        int right)
/* search list[0] <= list[1] <= ... <= list[n-1] for
   searchnum. Return its position if found. Otherwise
   return -1 */
   int middle;
   if (left <= right) {
      middle = (left + right)/2;
      switch (COMPARE(list[middle], searchnum)) {
         case -1: return
            binsearch(list, searchnum, middle + 1, right);
         case 0 : return middle;
         case 1 : return
            binsearch(list, searchnum, left, middle - 1);
      }
   }
   return -1;
```

Program 1.8: Recursive implementation of binary search

PERFORMANCE ANALYSIS

An algorithm is said to be efficient and fast, if it takes **less time to execute** & **consume less memory space**

Performance is analyzed based on 2 criteria

1. Space Complexity
2. Time Complexity

1. Space Complexity

- Analysis of **space complexity of an algorithm** or program is the **amount of memory it needs to run to completion**.
- The space needed by a program consists of following components.
    - **Fixed space requirements:** Independent on the number and size of the programs input and output. It include
        - Instruction Space (Space needed to store the code)
        - Space for simple variable
        - Space for constants
    - **Variable space requirements:** This component consists of

- Space needed by structured variable whose size depends on the particular instance *I* of the problem being solved
- Space required when a function uses recursion

- Total Space Complexity **S(P)** of a program is

$$\mathbf{S(P) = C + S_p(I)}$$

Here $\mathbf{S_p(I)}$ is Variable space requirements of program P working on an instance *I*.

C is a constant representing the fixed space requirements

- Example :

1.
```
int sum(int A[], int n)
{
    int sum=0, i;
    for(i=0;i<n;i++)
    {
        Sum=sum+A[i];
        return sum;
    }
}
```

Here Space needed for variable n = 1 byte

Sum = 1 byte

i = 1 byte

Array A[i] = n byte

Total Space complexity = [n+3] byte

2.
```
void main()
{
    int x,y,z,sum;
    printf("Enter 3 numbers");
    scanf("%d%d%d",&x,&y,&z);
    sum = x+y+z;
    printf("The sum = %d",sum);
```

}

Here Space needed for variable x = 1 byte

y = 1 byte

z = 1 byte

sum = 1 byte

Total Space complexity = 4 byte

3. sum (a,n)

{

int s=0;

for(i=0;i<n;i++)

for(j=0;j<m;j++)

s=s+a[i][j];

return s;

}

Here Space needed for variable n = 1 byte

m = 1 byte

s = 1 byte

i = 1 byte

j = 1 byte

Array a[i][j] = nm byte

Total Space complexity = nm+5 byte

2. Time Complexity

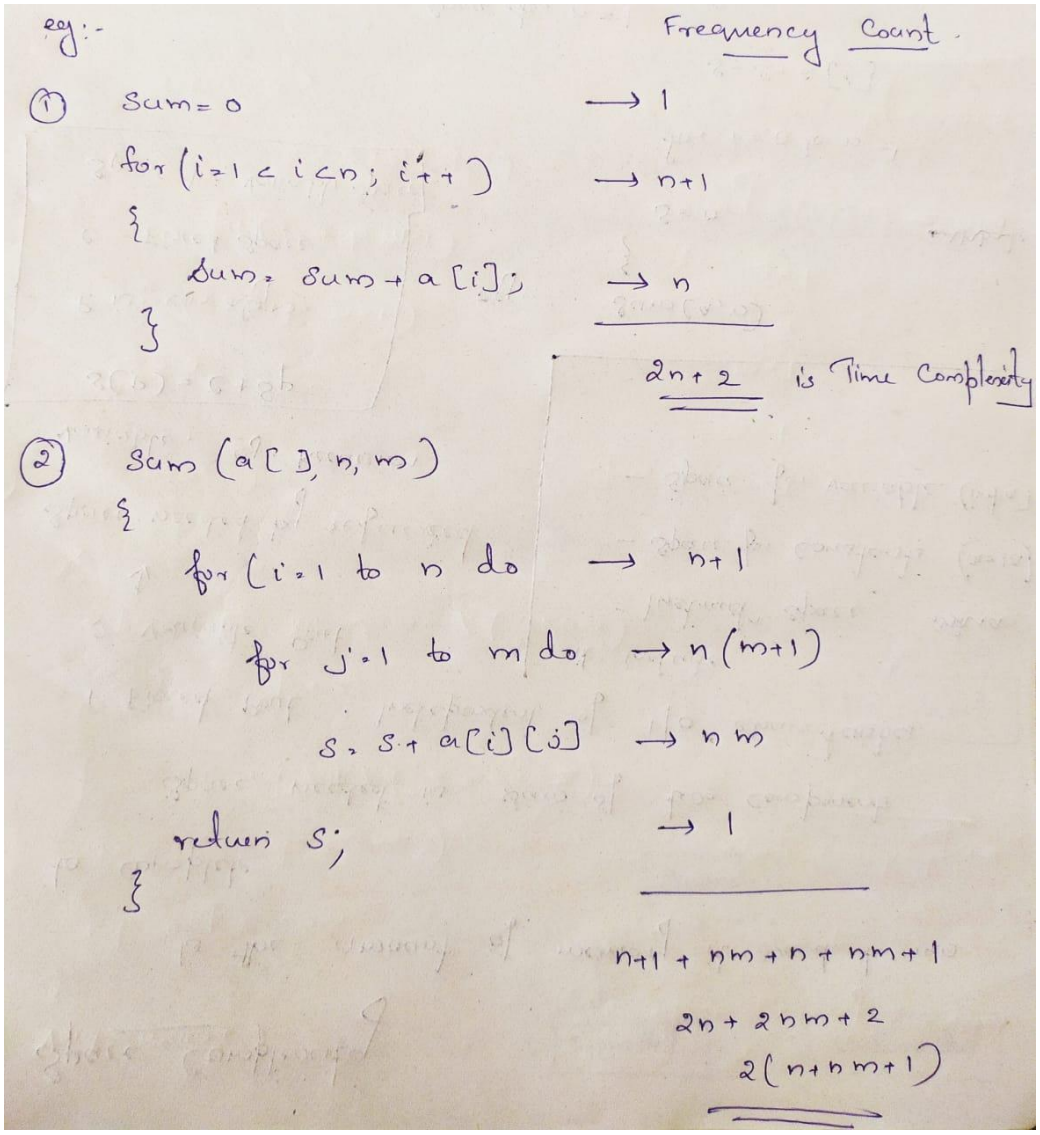
- The **time complexity of an algorithm** or a program is the **amount of time it needs to run to completion**.
- **$T(P)=C+T_P$**

  Here **C** is compile time

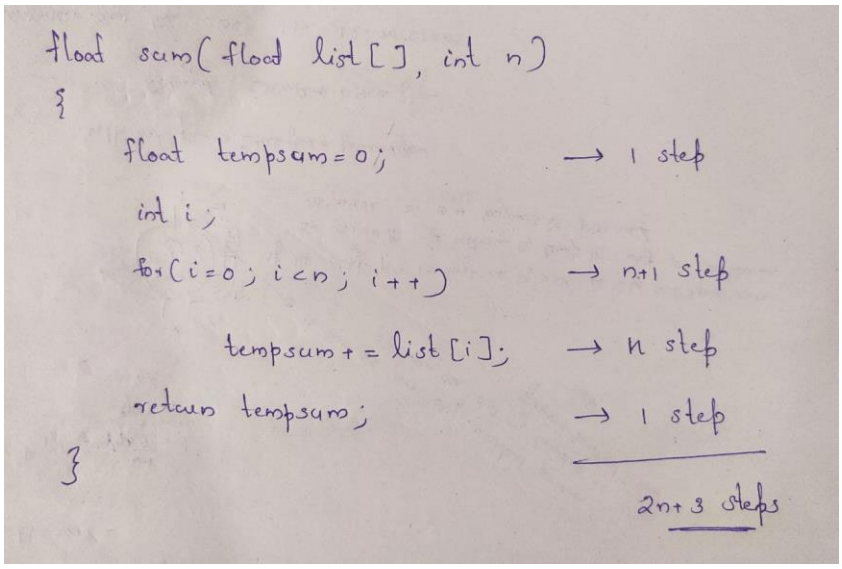
  T_P is Runtime
- For calculating the time complexity, we use a method called **Frequency Count** ie, counting the number of steps
  - Comments – 0 step

- Assignment statement – 1 Step
- Conditional statement – 1 Step
- Loop condition for ‘n’ numbers – n+1 Step
- Body of the loop – n step
- Return statement – 1 Step

- Examples:



3. Iterative function for summing a list of numbers

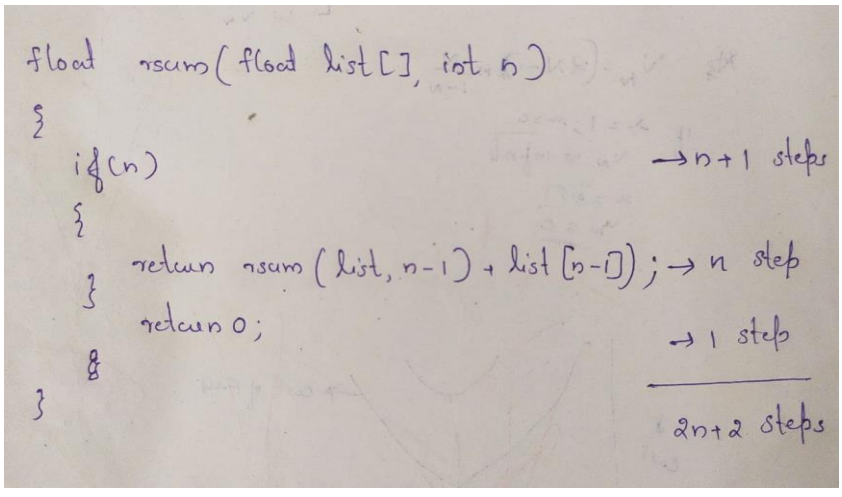


Tabular Method

Statement	s/e	Frequency	Total steps
float sum(float list[ ], int n)	0	0	0
{	0	0	0
float tempsum = 0;	1	1	1
int i;	0	0	0
for(i=0; i <n; i++)	1	n+1	n+1
tempsum += list[i];	1	n	n
return tempsum;	1	1	1
}	0	0	0
Total			2n+3

s/e =steps/execution

4. Recursive summing of a list of numbers



Tabular Method

Statement	s/e	Frequency	Total steps
float rsum(float list[ ], int n)	0	0	0
{	0	0	0
if (n)	1	n+1	n+1
return rsum(list, n-1)+list[n-1];	1	n	n
return list[0];	1	1	1
}	0	0	0
Total			2n+2

- When we analyze an algorithm it depends on the input data, there are three cases :
  a. **Best case:** The best case is the minimum number of steps that can be executed for the given parameters.
  b. **Average case:** The average case is the average number of steps executed on instances with the given parameters.
  c. **Worst case**: In the worst case, is the maximum number of steps that can be executed for the given parameters

ASYMPTOTIC NOTATION

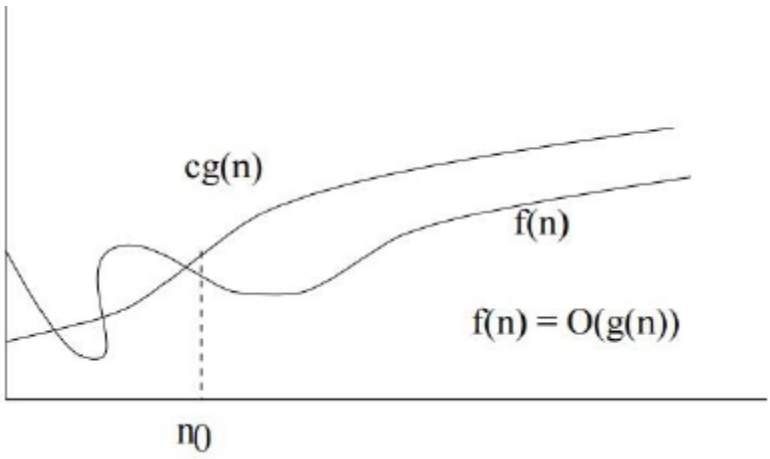
- Complexity of an algorithm is usually a function of n.
- Behavior of this function is usually expressed in terms of one or more standard functions.
- Expressing the complexity function with reference to other known functions is called **asymptotic complexity**.
- Three basic notations are used to express the asymptotic complexity

1. **Big – Oh notation O :** Upper bound of the algorithm
2. **Big – Omega notation Ω :** Lower bound of the algorithm
3. **Big – Theta notation Θ :** Average bound of the algorithm

1. Big – Oh notation O

- Formal method of expressing the upper bound of an algorithm's running time.
- i.e. it is a measure of longest amount of time it could possibly take for an algorithm to complete.
- It is used to represent the **worst case** complexity.

- **f(n) = O(g(n))** if and only if there are two positive constants **c** and **n0** such that **f(n) ≤ c g(n)** for all **n ≥ n0** .
- Then we say that **"f(n) is big-O of g(n)"**.



- Examples:
  1. Derive the Big – Oh notation for f(n) = 2n + 3

     Ans:

     2n + 3 <= 2n+3n

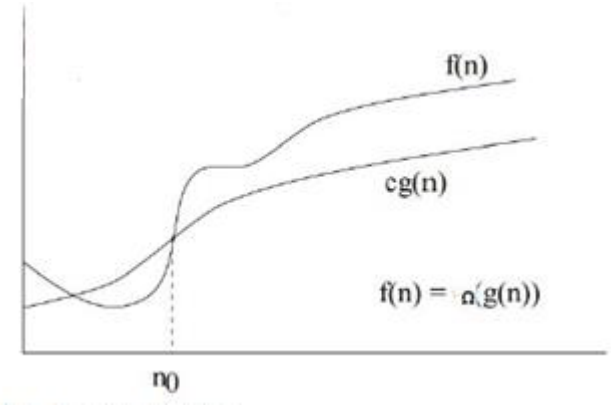
     2n+3 <= 5n for all n>=1

     Here c = 5

     g(n) = n

     so, f(n) = O(n)

2. **Big – Omega notation Ω**
   - **f(n) = Ω(g(n))** if and only if there are two positive constants **c** and **n0** such that **f(n) ≥ c g(n)** for all **n ≥ n0.**
   - Then we say that **"f(n) is omega of g(n)"**.



   - Examples:

Derive the Big – Omega notation for f(n) = 2n + 3

Ans:

2n + 3 >= 1n for all n>=1

Here c = 1

g(n) = n

so, f(n) = Ω (n)

3. Big – Theta notation Θ

- **f(n) = Θ(g(n))** if and only if there are three positive constants c1, c2 and n0 such that **c1 g(n) ≤ f(n) ≤ c2 g(n)** for all **n ≥ n0** .
- Then we say that **"f(n) is theta of g(n)"**.
- Examples:

  Derive the Big – Theta notation for f(n) = 2n + 3

  Ans:

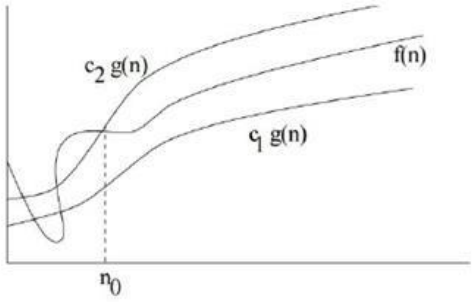
  1n <= 2n + 3 <= 5n for all n>=1

  Here c1 = 1

  C2 = 5

  g1(n) and g2(n) = n

  so, f(n) = **Θ** (n)



Example: $n^2 + 5n + 7 = \Theta(n^2)$

Proof: •

When $n \geq 1$, $n^2 + 5n + 7 \leq n^2 + 5n^2 + 7n^2 \leq 13n^2$

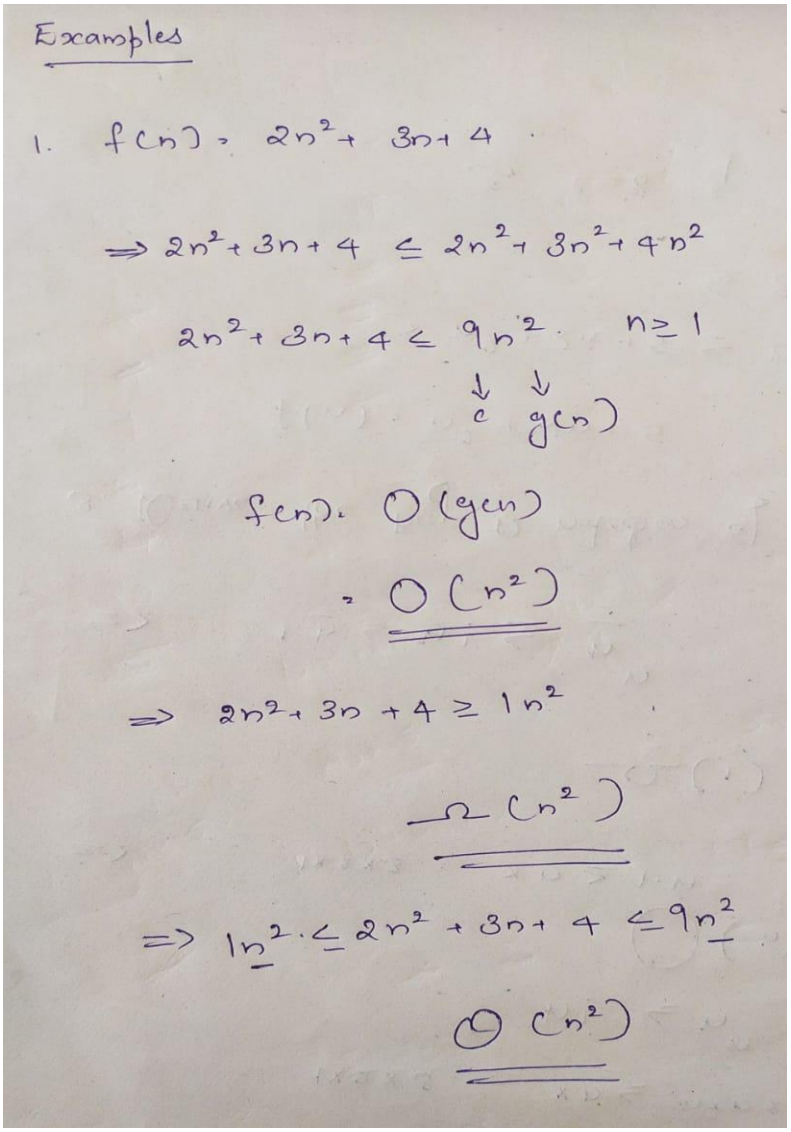
- When $n \geq 0$, $n^2 \leq n^2 + 5n + 7$
- Thus, when $n \geq 1$

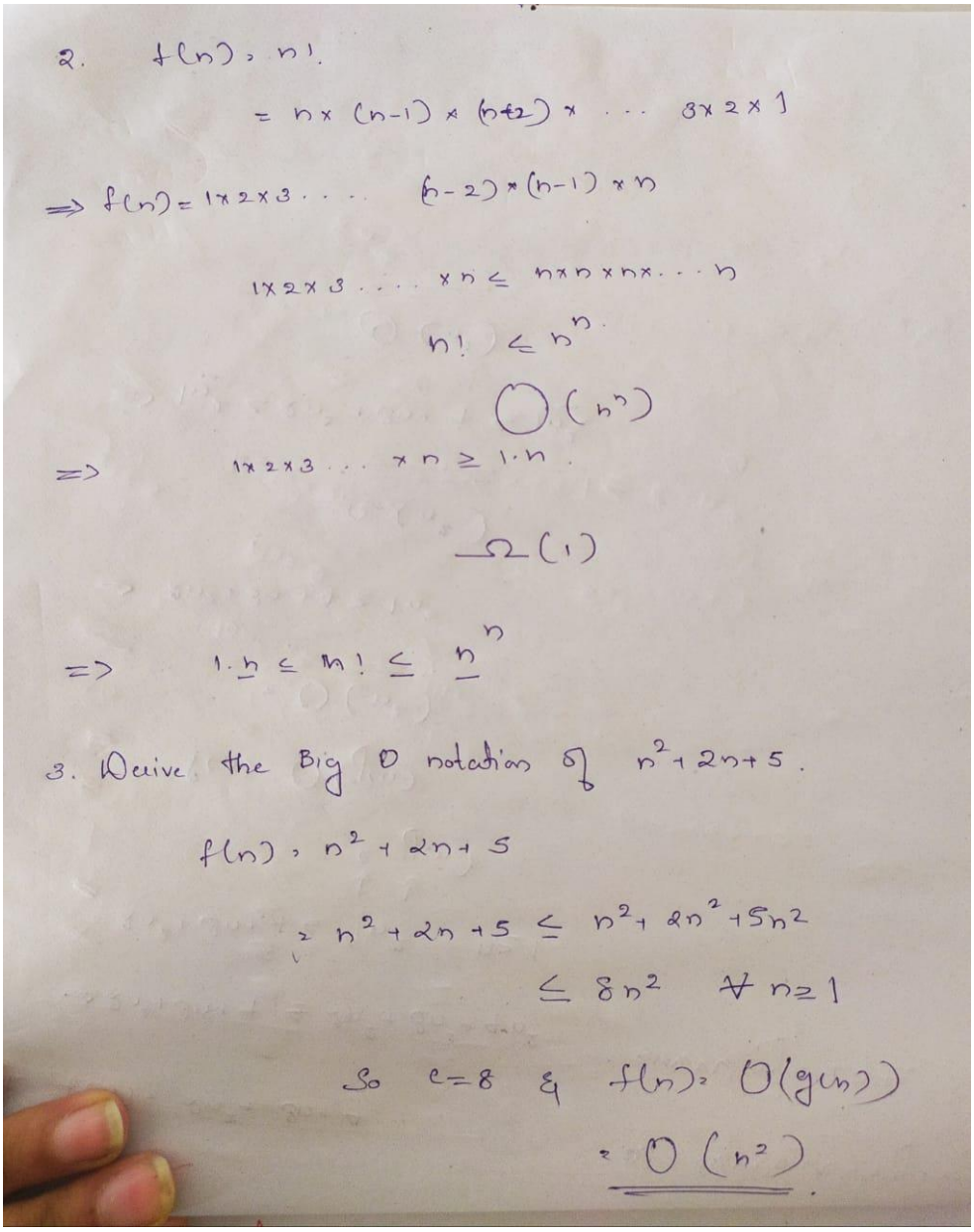
  $1n^2 \leq n^2 + 5n + 7 \leq 13n^2$

  Thus, we have shown that $n^2 + 5n + 7 = \Theta(n^2)$ (by definition of Big-Θ, with $n_0 = 1$, c1 = 1, and c2 = 13.)

Comparison of different Algorithm

Algorithm	Best case	Average case	Worst case
Quick sort	O(n log n)	O(n log n)	$O(n^2)$
Merge sort	O(n log n)	O(n log n)	O(n log n)
Heap sort	O(n log n)	O(n log n)	O(n log n)
Bubble sort	O(n)	$O(n^2)$	$O(n^2)$
Selection Sort	$O(n^2)$	$O(n^2)$	$O(n^2)$
Insertion sort	O(n)	$O(n^2)$	$O(n^2)$
Binary search	O(1)	O(log n)	O(log n)
Linear search	O(1)	O(n)	O(n)





TIME COMPLEXITY OF LINEAR SEARCH

- Any algorithm is analyzed based on the unit of computation it performs. For linear search, we need to count the number of comparisons performed, but each comparison may or may not search the desired item.

Best Case	Worst Case	Average Case
1	n	n / 2

TIME COMPLEXITY OF BINARY SEARCH

- In Binary search algorithm, the target key is examined in a sorted sequence and this algorithm starts searching with the middle item of the sorted sequence.

  a. If the middle item is the target value, then the search item is found and it returns True.

  b. If the target **item < middle** item, then search for the target value in the first half of the list.

  c. If the target **item > middle** item, then search for the target value in the second half of the list.

- In binary search as the list is ordered, so we can eliminate half of the values in the list in each iteration.
- Consider an example, suppose we want to search 10 in a sorted array of elements, then we first determine 15 the middle element of the array. As the middle item contains 18, which is greater than the target value 10, so can discard the second half of the list and repeat the process to first half of the array. This process is repeated until the desired target item is located in the list. If the item is found then it returns True, otherwise False.
- In Binary Search, each comparison eliminates about half of the items from the list. Consider a list with n items, then about ***n/2*** items will be eliminated after first comparison. After second comparison, ***n/4*** items of the list will be eliminated. If this process is repeated for several times, then there will be just one item left in the list. The number of comparisons required to reach to this point is $n/2^i = 1$. If we solve for ***i***, then it gives us $i = \log_2 n$. The maximum number is comparison is logarithmic in nature, hence the time complexity of binary search is ***O(log n).***

Best Case	Worst Case	Average Case
1	O(log n)	O(log n)

MODULE 2 - ARRAYS AND SEARCHING

Polynomial representation using Arrays, Sparse matrix, Stacks, Queues - Circular Queues, Priority Queues, Double Ended Queues, Evaluation of Expressions, Linear Search and Binary Search

DATA STRUCTURE

It is a representation of logical relationship between individual elements of data. It is also defined as a mathematical model of particular organization of data items. It is also called building block of a program.

Classification of data structure

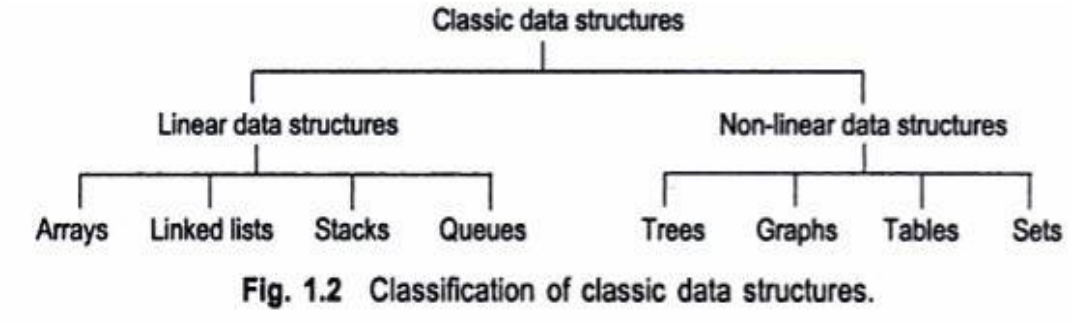
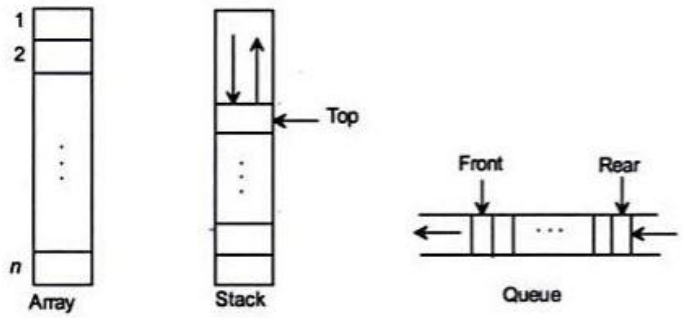


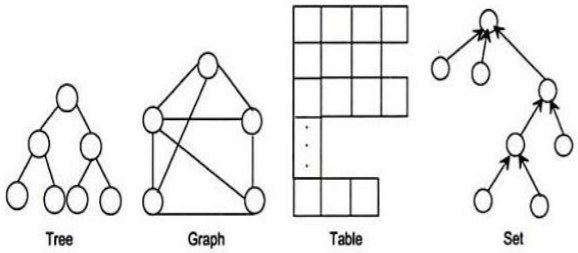
Fig. 1.2 Classification of classic data structures.

1. Linear data structure
    - All the elements form a sequence or maintain a linear ordering.



(a) Linear data structures

2. Non linear data structure
    - Elements are distributed over a plane.



(b) Non-linear data structures

1. POLYNOMIAL REPRESENTATION USING ARRAYS

- A polynomial is a sum of terms where each term has the form $\mathbf{ax^e}$, Where x is the variable, a is the coefficient and e is the exponent.

A general polynomial $A(x)$ can be written as

$$a_n x^n + a_{n-1} x^{n-1} + \ldots + a_1 x + a_0$$

where $a_n \neq 0$ and we say that the degree of A is n.

Polynomial representation using Arrays

If the Polynomial is $-10 + 3x + 5x^2$ then we can write it as : $-10x^0 + 3x^1 + 5x^2$

$-10x^0 + 3x^1 + 5x^2$

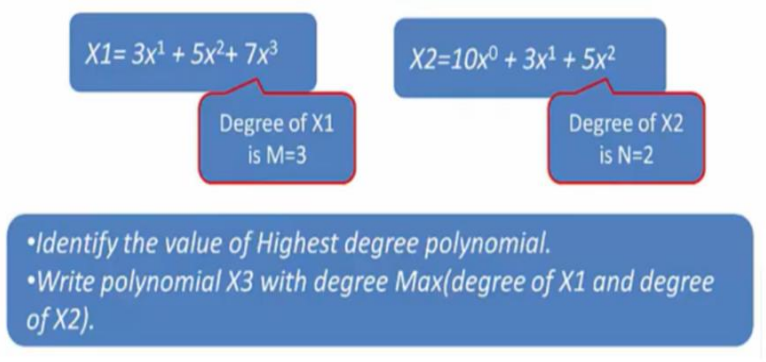
	0	1	2
Poly	-10	3	5

int Poly[3];

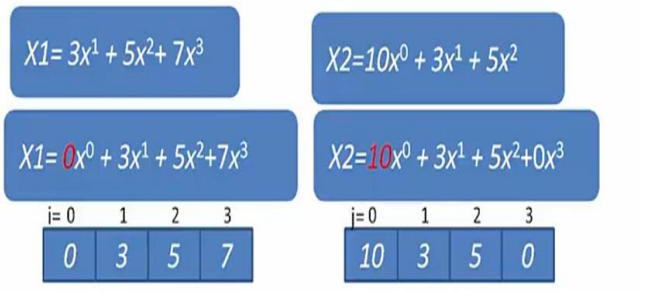
A polynomial of a single variable $A(x)$ can be written as $a_0 + a_1X + a_2X^2 + \ldots + a_nX^n$ where $a_n \neq 0$ and degree of $A(X)$ is n.

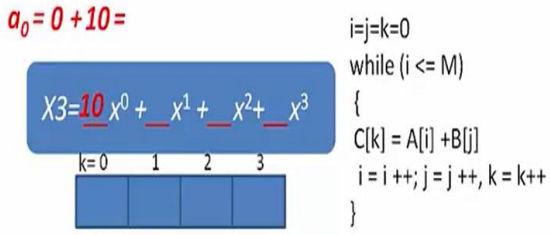
	0	1	2	...	n-1	n
Poly	a_0	a_1	a_2	...	a_{n-1}	a_n

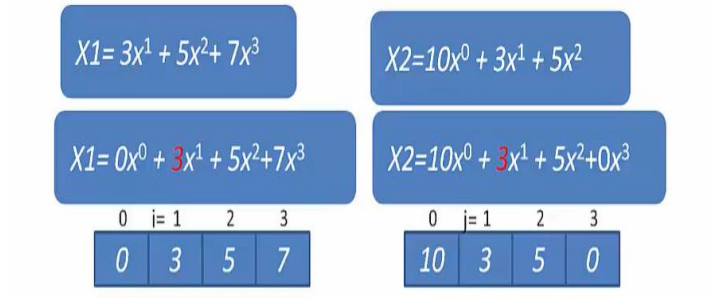
For a polynomial of degree n, n+1 terms are required

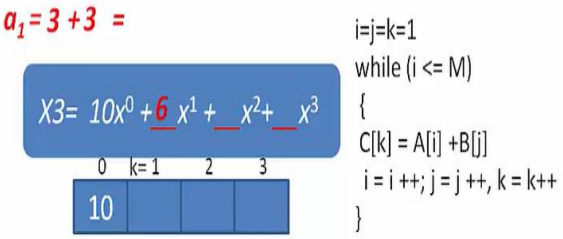


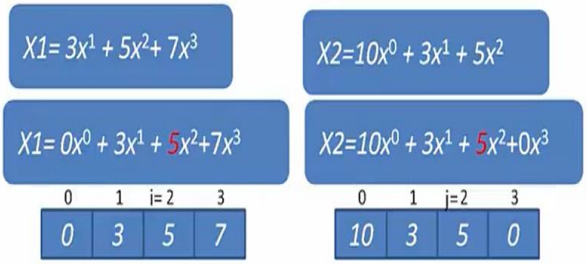
Polynomial Addition Example

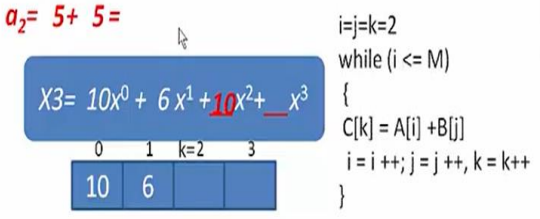


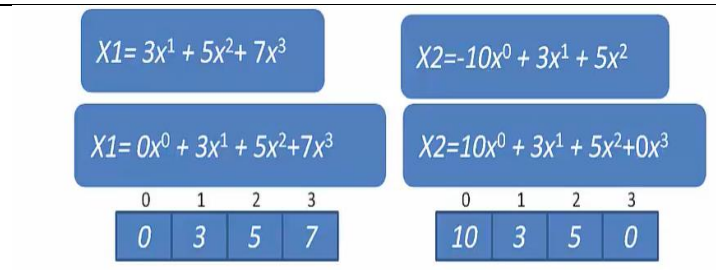


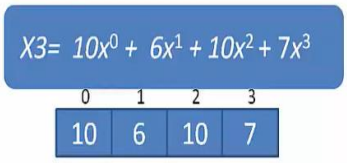




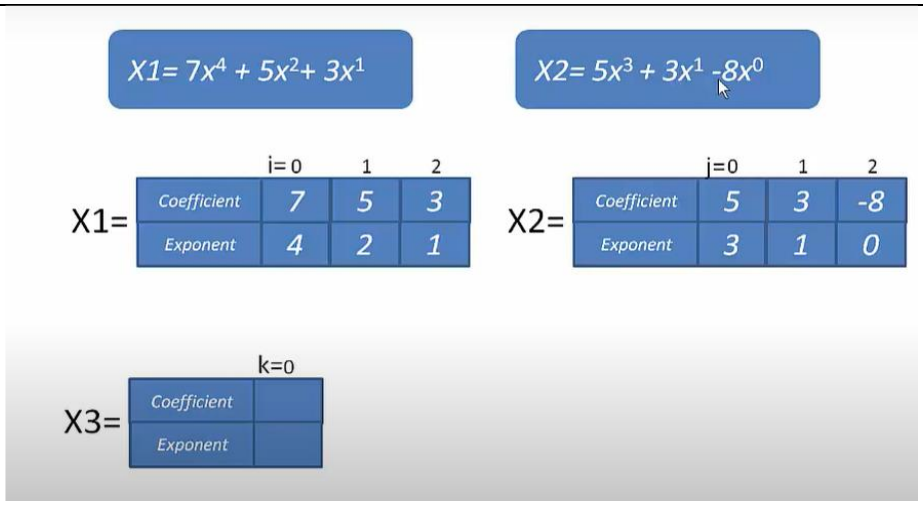


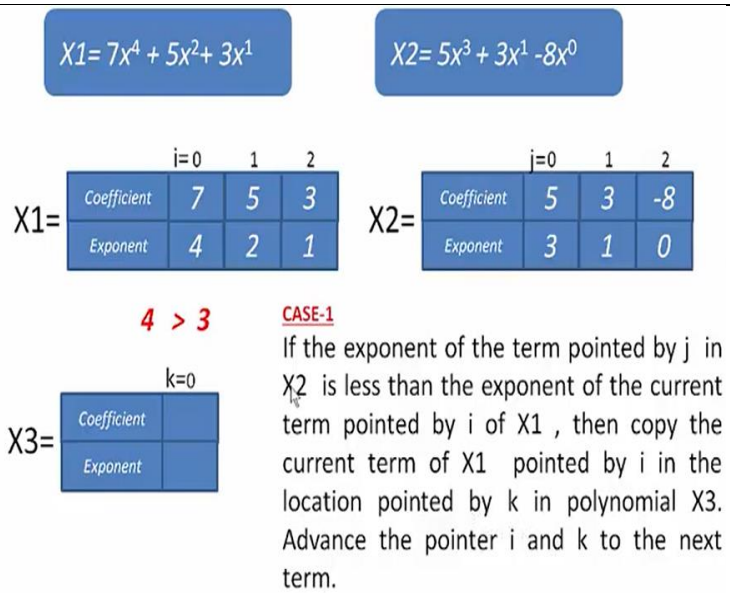


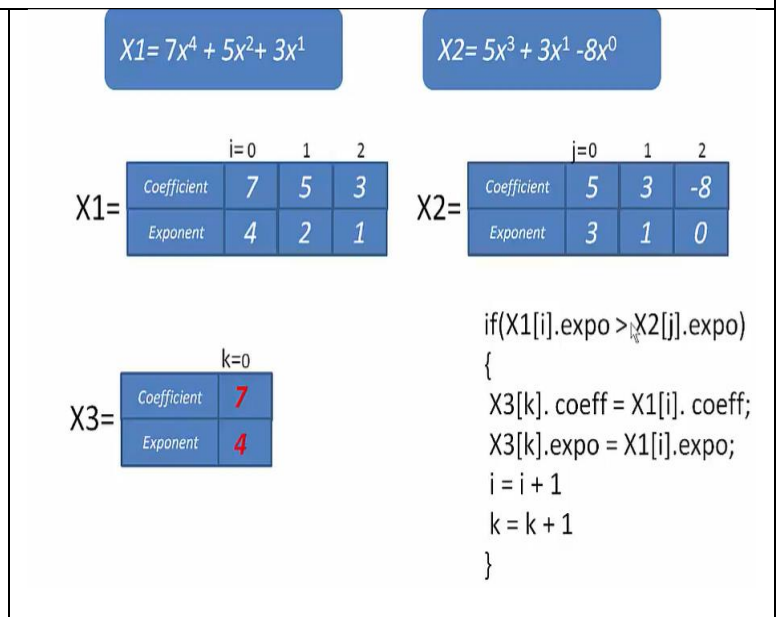


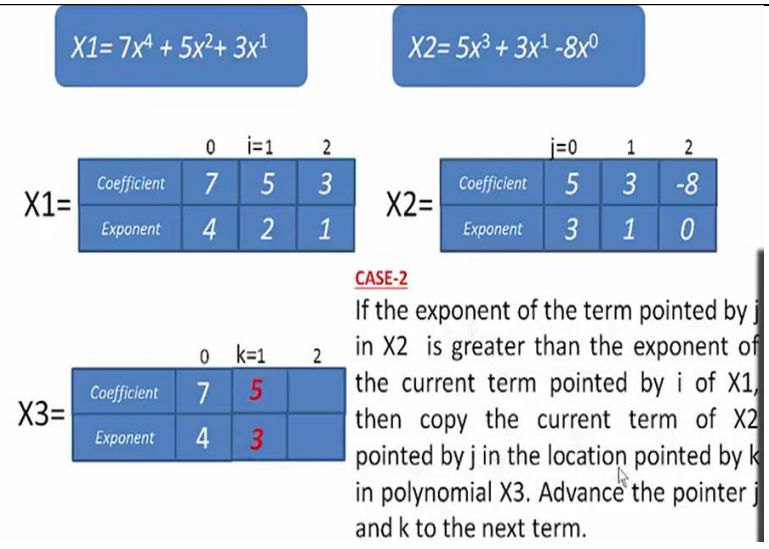


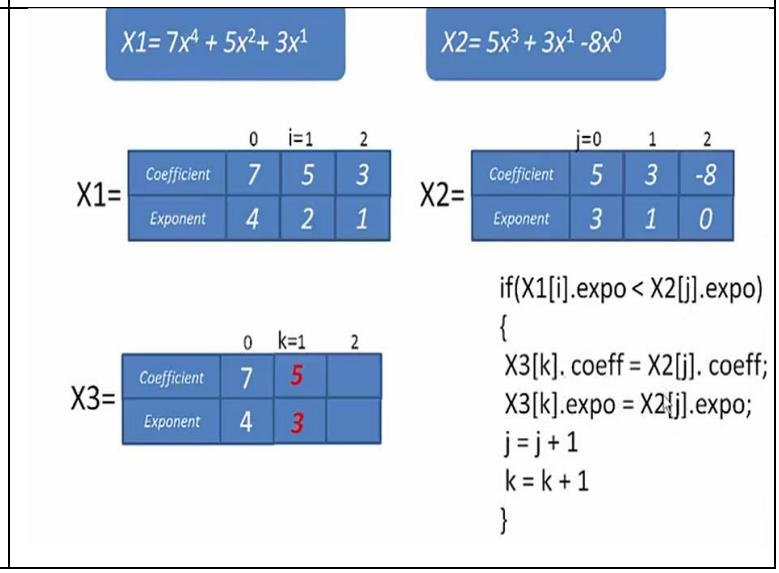
Steps of Polynomial Addition











$X1 = 7x^4 + 5x^2 + 3x^1$

$X2 = 5x^3 + 3x^1 - 8x^0$

X1=

	0	1	i=2
Coefficient	7	5	3
Exponent	4	2	1

X2=

	0	j=1	2
Coefficient	5	3	-8
Exponent	3	1	0

X3=

	0	1	2	k=3	
Coefficient	7	5	5		
Exponent	4	3	2		

```
if(X1[i].expo > X2[j].expo)
{
 X3[k]. coeff = X1[i]. coeff
 X3[k].expo = X1[i].expo;
 i = i + 1
 k = k + 1
```

$X1 = 7x^4 + 5x^2 + 3x^1$

$X2 = 5x^3 + 3x^1 - 8x^0$

X1=

	0	1	i=2
Coefficient	7	5	3
Exponent	4	2	1

X2=

	0	j=1	2
Coefficient	5	3	-8
Exponent	3	1	0

1 = 1

X3=

	0	1	2	k=3	4
Coefficient	7	5	5		
Exponent	4	3	2		

CASE-3

If the exponents of the two terms of polynomials X1 and X2 are equal, then the coefficients are added, and the new term is stored in the resultant polynomial X3 and advance I, j and k to track to the next term.

$X1 = 7x^4 + 5x^2 + 3x^1$

$X2 = 5x^3 + 3x^1 - 8x^0$

X1=

	0	1	i=2
Coefficient	7	5	3
Exponent	4	2	1

X2=

	0	1	j=2
Coefficient	5	3	-8
Exponent	3	1	0

X3=

	0	1	2	3	k=4
Coefficient	7	5	5	6	
Exponent	4	3	2	1	

```
if(X1[i].expo == X2[j].expo)
{
X3[k].coeff = X1[i].coeff +
                        X2[j].coeff
X3[k].expo = X1[i].expo;
i = i + 1
j=j+1
k = k + 1
```

$X1 = 7x^4 + 5x^2 + 3x^1$

$X2 = 5x^3 + 3x^1 - 8x^0$

X1=

	0	1	i=2
Coefficient	7	5	3
Exponent	4	2	1

X2=

	0	1	j=2
Coefficient	5	3	-8
Exponent	3	1	0

No more element in i

X3=

	0	1	2	3	k=4
Coefficient	7	5	5	6	-8
Exponent	4	3	2	1	0

CASE-3

If there is no more elements in X1 and there are few elements remaining in X2 then copy rest of the element in X2 to X3 and advance j and k to track to the next term.

$X1 = 7x^4 + 5x^2 + 3x^1$

$X2 = 5x^3 + 3x^1 - 8x^0$

X1=

	0	1	i=2
Coefficient	7	5	3
Exponent	4	2	1

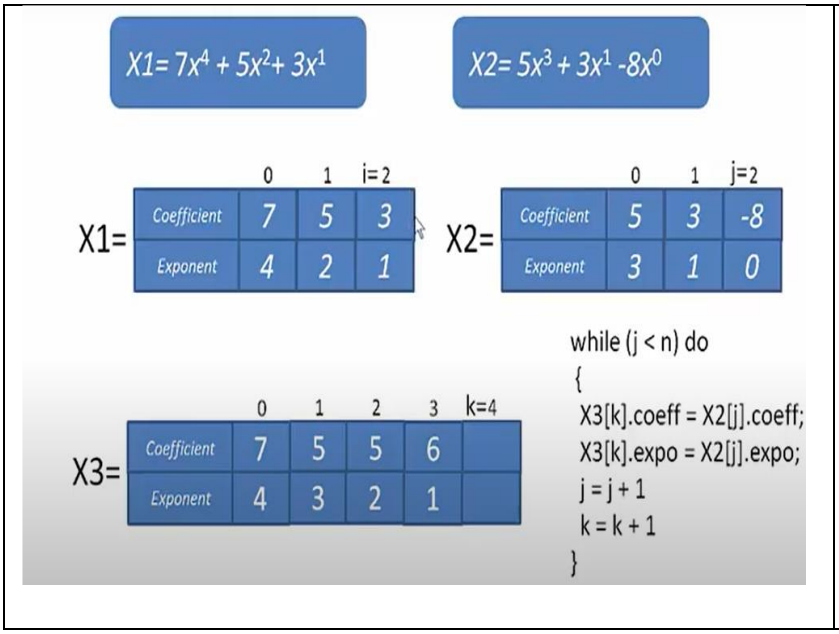
X2=

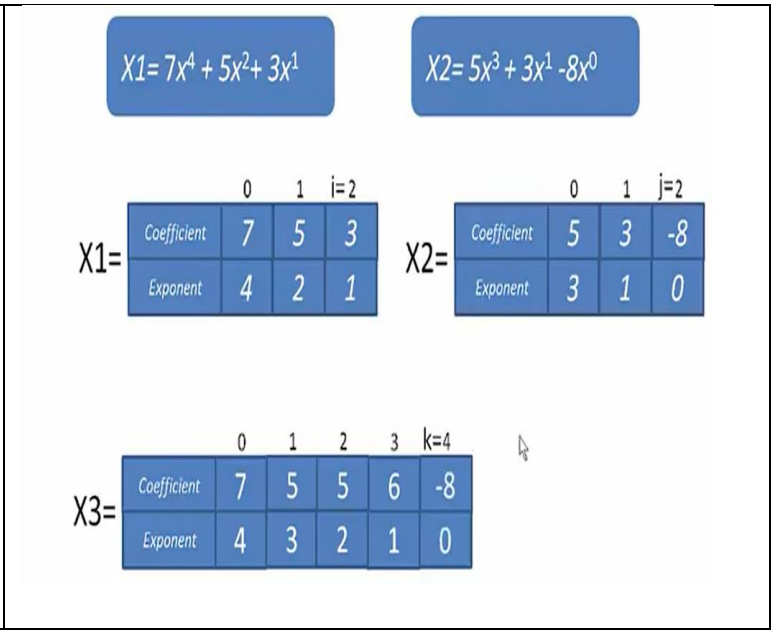
	0	1	j=2
Coefficient	5	3	-8
Exponent	3	1	0

X3=

	0	1	2	3	k=4
Coefficient	7	5	5	6	
Exponent	4	3	2	1	

```
while (j < n) do
{
 X3[k].coeff = X2[j].coeff;
 X3[k].expo = X2[j].expo;
 j = j + 1
 k = k + 1
}
```

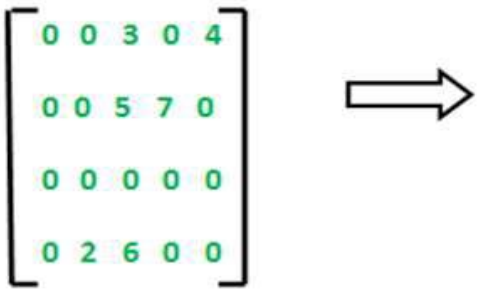




2. SPARSE MATRIX

- A matrix is a two-dimensional data object made of 'm' rows and 'n' columns, therefore having total m x n values. If most of the elements of the matrix have 0 values, then it is called a **sparse matrix**.
- **Sparse matrix is a matrix which contains very few non-zero elements.**
- When a sparse matrix is represented with a 2-dimensional array, we waste a lot of space to represent that matrix.
- Consider a matrix of size **100 X 100** containing only **10 non-zero** elements. In this matrix, **only 10 spaces are filled** with non-zero values and remaining spaces of the matrix are filled with zero. Totally we allocate 100 X 100 X 2 = 20000 bytes of space to store this integer matrix. To access these 10 non-zero elements we have to make scanning for 10000 times.
- Sparse Matrix Representations can be done in many ways following are two common representations:
  1. Array representation
     - Three tuple form
  2. Linked list representation
- 2D array is used to represent a sparse matrix in which there are three columns named as
  - **Row:**Index of row, where non-zero element is located

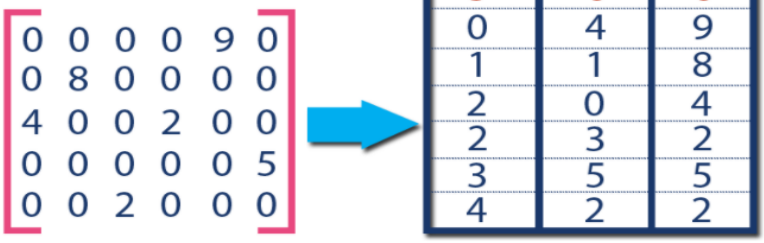
- **Column:**Index of column, where non-zero element is located
- **Value:**Value of the non zero element located at index –(row,column)



Row	Column	Value
0	2	3
0	4	4
1	2	5
1	3	7
3	1	2
3	2	6

Row	0	0	1	1	3	3
Column	2	4	2	3	1	2
Value	3	4	5	7	2	6

Triplets
(0,2,3)
(0,4,4)
(1,2,5)
(1,3,7)
(3,1,2)
(3,2,6)



Rows	Columns	Values
5	6	6
0	4	9
1	1	8
2	0	4
2	3	2
3	5	5
4	2	2

Why to use Sparse Matrix instead of simple matrix ?

- **Storage:** There are lesser non-zero elements than zeros and thus lesser memory can be used to store only those elements.

- **Computing time:** Computing time can be saved by logically designing a data structure traversing only non-zero elements.

3. STACK

- It is a linear data structure in which elements are placed one above another.
- A stack is an ordered collection of homogeneous data elements where the insertion and deletion operations take place only at one end called **Top** of the stack.
- **LIFO -** In stack elements are arranged in **Last-In-First-Out** manner (**LIFO**). So it is also called LIFO lists.
- Anything added to the stack goes on the "**top**" of the stack.
- Anything removed from the stack is taken from the "**top**" of the stack.
- Things are removed in the reverse order from that in which they were inserted

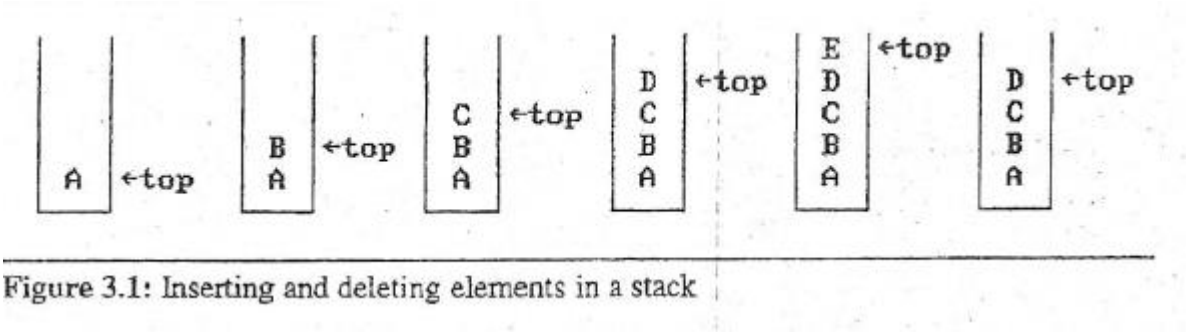
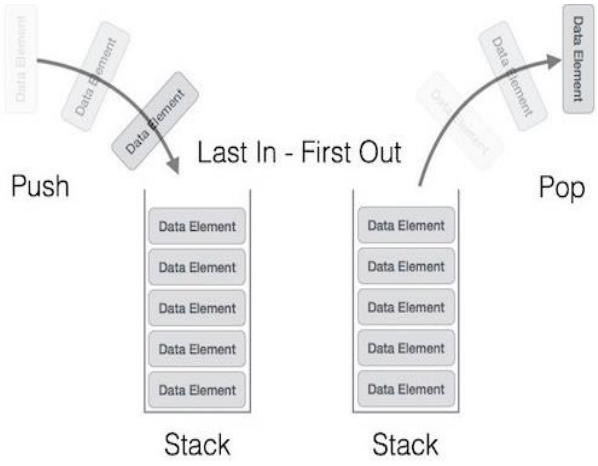


Figure 3.1: Inserting and deleting elements in a stack

Operations of Stack

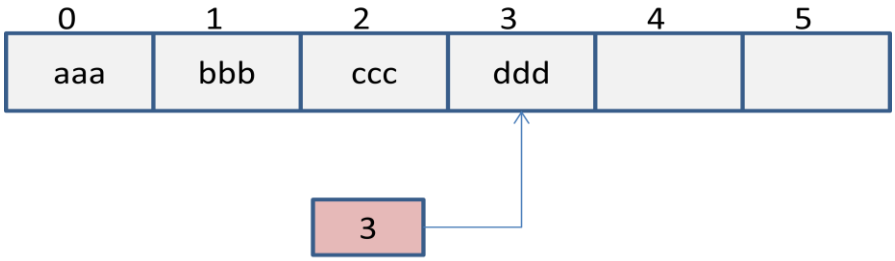
- Two basic operations of stack:
  - **PUSH :** Insert an element at the top of stack
  - **POP:** Delete an element from the top of stack



- An element in the stack is termed as **ITEM.**
- Initially top is set to -1, to indicate an **empty stack**. **(Top = -1)**
- The maximum no. of elements that a stack can accommodate is termed **MAX_SIZE**.
- If stack is full **Top = MAX_SIZE - 1**

Array representation of stack

- Stack can be represented using a linear array.
- There is a pointer called TOP to indicate the top of the stack



- **Overflow:** If we try to insert a new element in the stack top (push) which is already **full**, then the situation is called stack overflow.
- **Underflow:** If we try to delete an element (pop) from an **empty** stack, the situation is called stack underflow.

Basic Operations

- **push()** − Pushing (storing) an element on the stack.
- **pop()** − Removing (accessing) an element from the stack.
- **peek()** − get the top data element of the stack, without removing it.

```
int peek() {
        return stack[top];
 }
```

- **isFull()** − check if stack is full.

```
bool isfull() {
        if (top == MAX_SIZE)
                return true;
        else
```

```
        return false;
    }
```

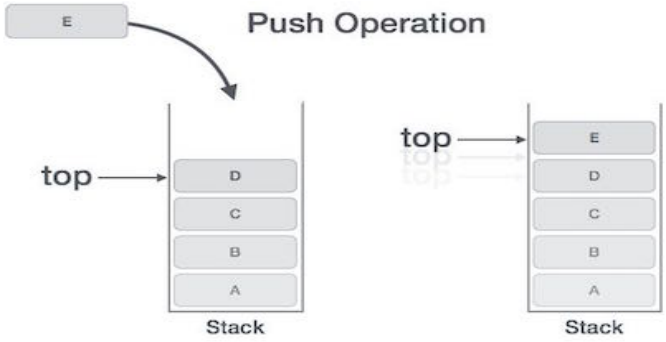
- **isEmpty()** − check if stack is empty.

```
bool isempty() {
        if(top == -1)
            return true;
        else
            return false;
}
```

Push Operation

The process of putting a new data element onto stack is known as a Push Operation. Push operation involves a series of steps −

- **Step 1** − Checks if the stack is full.
- **Step 2** − If the stack is full, produces an error and exit.
- **Step 3** − If the stack is not full, increments **top** to point next empty space.
- **Step 4** − Adds data element to the stack location, where top is pointing.
- **Step 5** − Returns success.



Algorithm: PUSH()

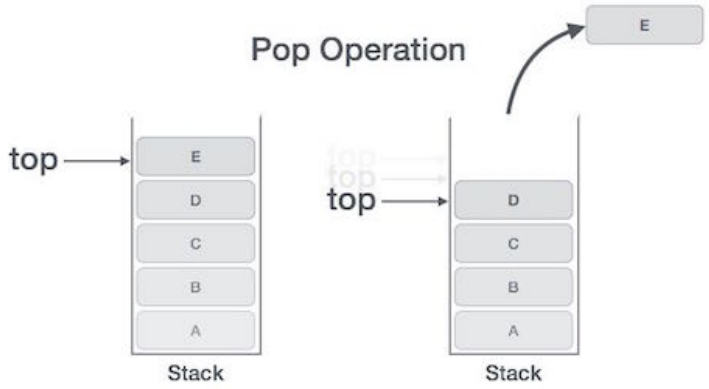
- Let A be an array with Maximum size as MAX_SIZE. Initially, top= -1

1. Start
2. if top < MAX_SIZE – 1
3. set top=top+1
4. Set A[top]=item
5. else
6. print “OVERFLOW”
7. exit

POP Operation

A Pop operation may involve the following steps −

- **Step 1** − Checks if the stack is empty.
- **Step 2** − If the stack is empty, produces an error and exit.
- **Step 3** − If the stack is not empty, accesses the data element at which **top** is pointing.
- **Step 4** − Decreases the value of top by 1.
- **Step 5** − Returns success.



Algorithm: POP()

1. Start
2. if top= -1 then
3. print “UNDERFLOW”
4. else
5. set item=A[top]
6. Set top=top-1
7. exit

Applications of stack

- Reversing an array
  - A B C D
  - Pushing to stack A B C D
  - Popping from stack D C B A
- Undo operations

- Infix to prefix, infix to postfix conversion
- Tree Traversal
- Evaluation of postfix expressions

4. QUEUES

- A queue is an ordered collection of homogeneous data elements. In which insertion is done at one end called **REAR** and deletion is done at another end called **FRONT**.
- **FIFO -** In queue elements are arranged in **First-In-First-Out** manner (**FIFO**).
- First inserted element is removed first

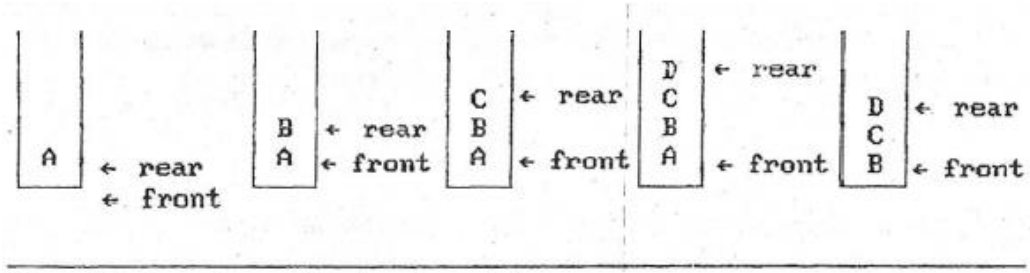
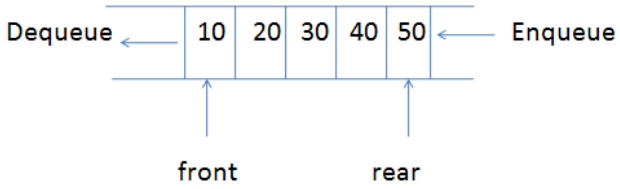


Figure 3.4: Inserting and deleting elements in a queue

- Two basic operations of queue:
  1. **Enqueue** -> Insert an element at the **rear** end of queue.
  2. **Dequeue**-> Delete an element from the **front** end of queue.



- Initial case **rear = -1** and **front = 0**, **MAX SIZE** is the size of the queue.
- If **rear = front** then queue contains only a **single element**
- If **rear < front** then queue is **empty**
- **Queue full :** rear = n-1 and front =0
- Whenever an element is deleted from the queue, the value of FRONT is increased by 1.
- i.e. FRONT=FRONT+1
- Similarly, whenever an element is added to the queue, the REAR is incremented by 1 as,
- REAR=REAR+1

Array Representation of Queue

A one-dimensional array, say $Q[1 \ldots N]$, can be used to represent a queue. Figure 5.3 shows an instance of such a queue. With this representation, two pointers, namely FRONT and REAR, are used to indicate the two ends of the queue. For the insertion of the next element, the pointer REAR will be the consultant and for deletion the pointer FRONT will be the consultant.

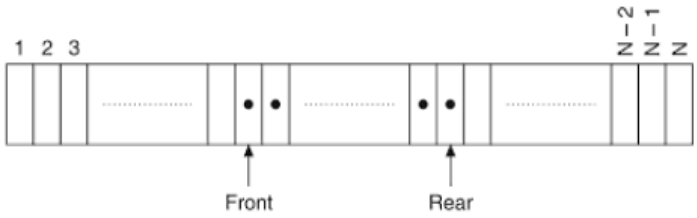


Figure 5.3 Array representation of a queue.

Basic Operations

- **enqueue()** − add (store) an item to the queue.
- **dequeue()** − remove (access) an item from the queue.
- **peek()** − Gets the element at the front of the queue without removing it.

```
int peek()
{
    return queue[front];
}
```

- **isfull()** − Checks if the queue is full

```
bool isfull()
{
        If (rear == MAXSIZE - 1)
                return true;
        else
                return false;
}
```

- **isempty()** − Checks if the queue is empty.

```
bool isempty()

{

        if(front < 0 || front > rear)

                return true;

        else

                return false;
```

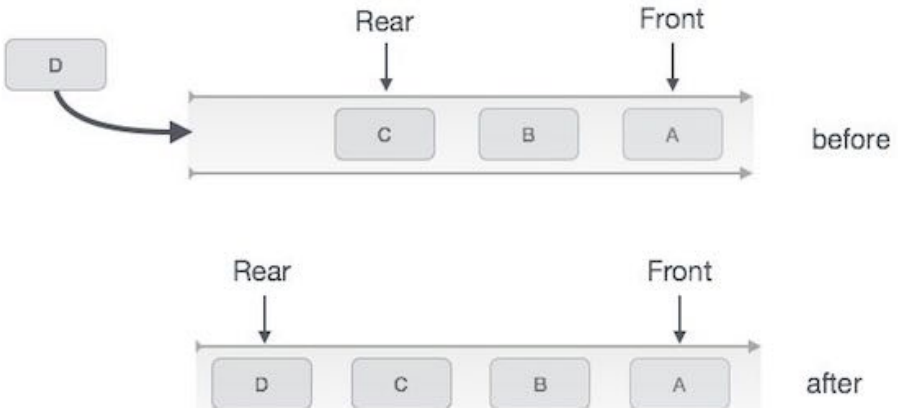
}

Enqueue Operation

Queues maintain two data pointers, **front** and **rear**. Therefore, its operations are comparatively difficult to implement than that of stacks.

The following steps should be taken to enqueue (insert) data into a queue −

- **Step 1** − Check if the queue is full.
- **Step 2** − If the queue is full, produce overflow error and exit.
- **Step 3** − If the queue is not full, increment **rear** pointer to point the next empty space.
- **Step 4** − Add data element to the queue location, where the rear is pointing.
- **Step 5** − return success.



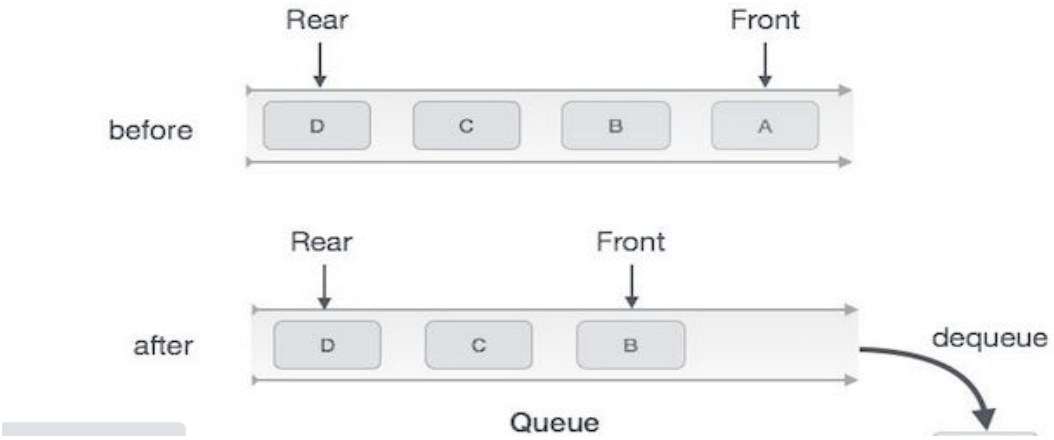
Algorithm : Enqueue

```
1. Start
2. if rear = MAX_SIZE – 1 then
3.       print “OVERFLOW”
4. else
5.       set rear = rear + 1
6.       Set A[rear]=item
7. exit
```

Dequeue Operation

Accessing data from the queue is a process of two tasks – access the data where **front** is pointing and remove the data after access. The following steps are taken to perform **dequeue** operation –

- **Step 1** – Check if the queue is empty.
- **Step 2** – If the queue is empty, produce underflow error and exit.
- **Step 3** – If the queue is not empty, access the data where **front** is pointing.
- **Step 4** – Increment **front** pointer to point to the next available data element.
- **Step 5** – Return success.



Algorithm : Dequeue

1. Start
2. if rear < front then
3. print "UNDER FLOW"
4. else
5. set item = A[front}
6. set front = front + 1
7. exit

Type of Queues

- Circular Queue
- Priority Queue
- Doubly ended Queue

5. CIRCULAR QUEUE

- To utilize space properly, circular queue is derived.
- In this queue the elements are inserted in circular manner.
- So that no space is wasted at all.
- Circular queue empty:

  FRONT= -1

  REAR= -1
- Circular queue full:

  (rear + 1) % max_size = Front
- It is a modification of simple queue in which the rear pointer is set to the initial location, whenever it reaches the location max_size – 1.

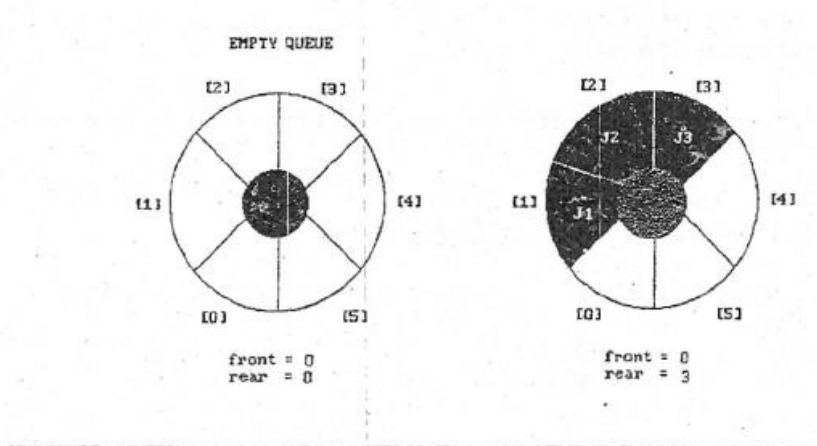


Figure 3.6: Empty and nonempty circular queues

Insertion Algorithm (ENQUEUE)

```
1. if (front == -1 & rear == -1)
2.         set front =0 and rear = 0
3.         Set a[rear]=item
4. else if (front = (rear+1) % max_size) then
5.          Print over flow
6. else
7.         set rear = (rear + 1)% max_size
8.          Set a[rear] = item
9. Exit
```

Deletion Algorithm (DEQUEUE)

```
1. if front = -1 and rear = -1 then
2.         print underflow and exit
3. else if front = rear
4.          set item= a[front]
5.          set front = -1 and rear = -1
6. else
7.         set item= a[front]
8.         set front = (front + 1) % max_size
9. Exit
```

7. PRIORITY QUEUE

- Regular queue follows a First In First Out (FIFO) order to insert and remove an item. Whatever goes in first, comes out first.
- In a priority queue, an item with the highest priority comes out first.
- Therefore, the FIFO pattern is no longer valid.
- Every item in the priority queue is associated with a priority.
- It does not matter in which order we insert the items in the queue
- The item with higher priority must be removed before the item with the lower priority.
- If two elements have the same priority, they are served according to their order in the queue.

Operations on a priority queue

1. **EnQueue**: EnQueue operation inserts an item into the queue. The item can be inserted at the end of the queue or at the front of the queue or at the middle. The item must have a priority.
2. **DeQueue**: DeQueue operation removes the item with the highest priority from the queue.
3. **Peek**: Peek operation reads the item with the highest priority.

1. **Enqueue Operation**

    1. IF((Front == 0)&&(Rear == N-1))
    2. PRINT “Overflow Condition”
    3. Else IF(Front == -1& rear == -1)
    4. Front = Rear =0
    5. Queue[Rear] = Data
    6. Priority[Rear] = Priority
    7. ELSE IF(Rear ==N-1)
    8. FOR (i=Front;i<=Rear;i++)
    9. FOR(i=Front;i<=Rear;i++)
    10. Q[i-Front] =Q[i]
    11. Pr[i-Front] = Pr[i]
    12. Rear = Rear-Front
    13. Front = 0
    14. FOR(i = r;i>f;i–)
    15. IF(p>Pr[i])
    16. Q[i+1] = Q[i] Pr[i+1] = Pr[i]
    17. ELSE
    18. Q[i+1] = data Pr[i+1] = p
    19. Rear++.

2. **Dequeue operation**

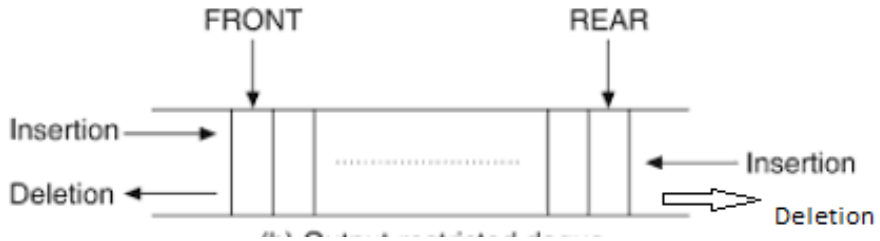
    1. IF(Front == -1)
    2. PRINT “Queue Under flow condition”
    3. ELSE
    4. PRINT”Q[f],Pr[f]”
    5. IF(Front==Rear)
    6. Front = Rear = -1
    7. ELSE
    8. FRONT++

Applications of Priority Queue

1. CPU Scheduling
2. Graph algorithms like Dijkstra's shortest path algorithm, Prim's Minimum Spanning Tree, etc
3. All queue applications where priority is involved.
4. For load balancing and interrupt handling in an operating system

8. DOUBLY ENDED QUEUE

It is a list of elements in which insertion and deletion are perform at both ends



- It has 4 operations
  1. Insertion at rear end
  2. Insertion at front end
  3. Deletion at rear end
  4. Deletion at front end

1. Algorithm : Insertion at rear end

```
1. Start
2. if rear = MAX_SIZE – 1 then
3.       print "OVERFLOW"
4. Else
5.       set rear = rear + 1
6.        Set A[rear]=item
7. exit
```

2. Insertion at front end

```
1. Start
2. if front = 0 then
3.       print “OVERFLOW” and exit
4. Else
5.       set front = front - 1
6.       Set A[front]=item
7. exit
```

3. Deletion at front end

```
1. Start
2. if front = 0 and rear = -1 then
3.       print “UNDER FLOW” and exit
4. set item = A[front]
5. if front = rear then
6.      set front = 0 and rear = -1
7. Else set front = front + 1
8. exit
```

4. Deletion at rear end

```
1. Start
2. if front = 0 and rear = -1 then
3.       print “UNDER FLOW” and exit
4. set item = A[rear]
5. if front = rear then
6.       set front = 0 and rear = -1
7. Else set rear = rear - 1
8. exit
```

9. CONVERSION & EVALUATION OF EXPRESSIONS

- **Infix Expression:** The operator occurs between the operands

  <operand> <operator> <operand>

  Eg: a+b

- **Prefix Expression (Polish notation):** The operators occurs before the operand

  <operator> <operand> <operand>

  Eg : +ab

- **Postfix Expression (Reverse Polish notation):** The operators occurs after the operand

  <operand> <operand> <operator>

  Eg : ab+

Token	Operator	Precedence[1]	Associativity
() [] –> .	function call array element struct or union member	17	left-to-right
–– ++	increment, decrement[2]	16	left-to-right
–– ++ ! ~ – + & * sizeof	decrement, increment[3] logical not one's complement unary minus or plus address or indirection size (in bytes)	15	right-to-left
(type)	type cast	14	right-to-left
* / %	multiplicative	13	left-to-right
+ –	binary add or subtract	12	left-to-right
<< >>	shift	11	left-to-right
> >= < <=	relational	10	left-to-right
== !=	equality	9	left-to-right
&	bitwise and	8	left-to-right
^	bitwise exclusive or	7	left-to-right
|	bitwise or	6	left-to-right
&&	logical and	5	left-to-right
||	logical or	4	left-to-right
?:	conditional	3	right-to-left
= += –= /= *= %= <<= >>= &= ^= |=	assignment	2	right-to-left
,	comma	1	left-to-right

Infix to prefix

? a + b * c

Ans: a + b * c [Two operands + & * consider the highest one]
(* is highest)

a + * b c

+ a * b c

? A + (B * C - D / E) - F

Ans: A + (B * C - D / E) - F

[Consider the expression inside bracket]

A + (* B C - / D E) - F

A + - * B C / D E - F

+ A - * B C / D E - F

- + A - * B C / D E F

Infix to postfix

? a + b * c

Ans:- a + b * c

a + b c *

a b c * +

? A + (B * C - D / E) - F

Ans:- A + (B * C - D / E) - F

A + (B C * - D E /) - F

A + (B C * D E / -) - F

A B C * D E / - + - F

A B C * D E / - + F -

Convert the following expression into postfix.

1. (A + (B * C - (D / E ↑ F) * G) * H)

First consider the inner bracket, that contain / & ↑ operands. ↑ has the highest degree.

(A + (B * C - (D / E F ↑) * G) * H)

(A + (B * C - (D / E F ↑) * G) * H)

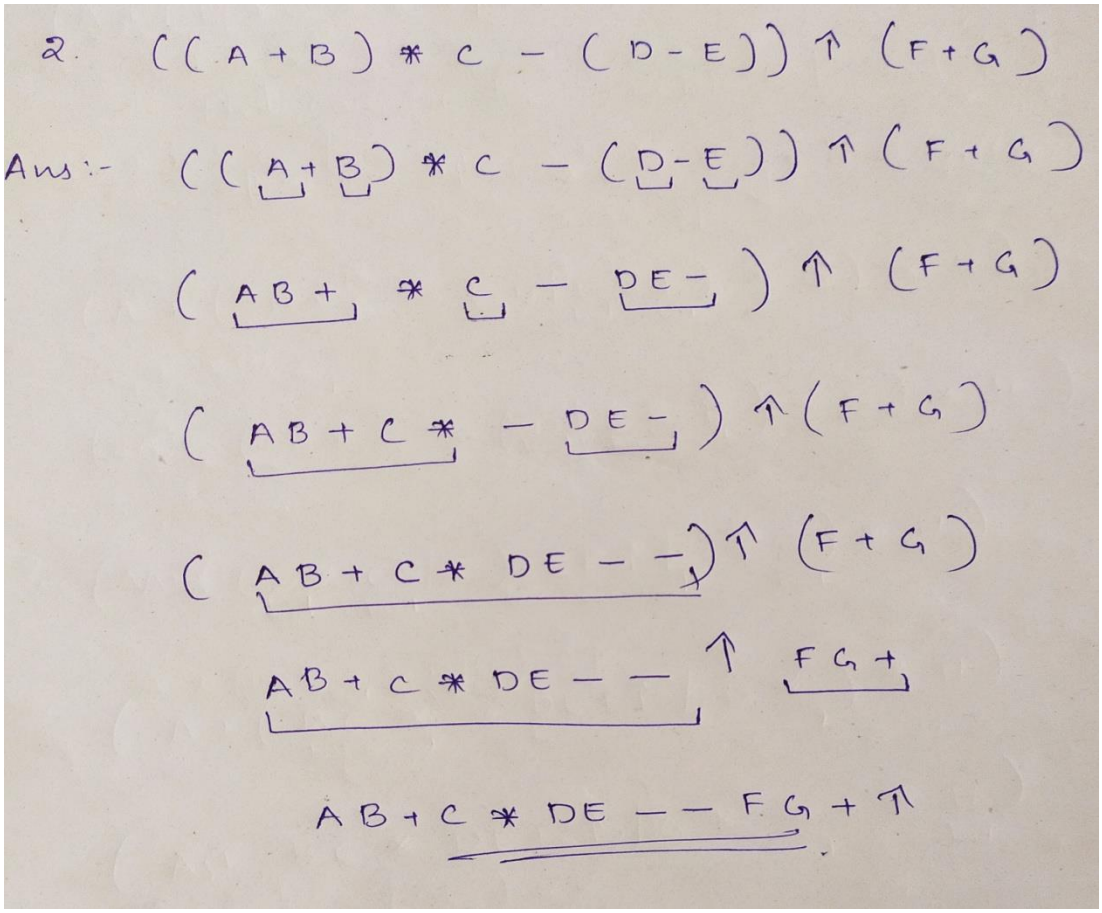
(A + (B * C - D E F ↑ / * G) * H)

(A + (B C * - D E F ↑ / G *) * H)

(A + B C * D E F ↑ / G * - * H)

(A + B C * D E F ↑ / G * - H *)

A B C * D E F ↑ / G * - H * +



A. Postfix Expression Evaluation

Given P is the postfix expression, the following algorithm uses a stack to hold operands. It finds the value of the arithmetic expression P, Written in postfix notation.

Algorithm:

Step 1: Add **") "** at the end of P

Step 2: Scan P from left – right & repeat the steps 3 & 4

Step 3: If an operand occurs, PUSH it to stack.

Step 4: If an operator ⊗ occurs, then

A: Remove the top elements of the stack.

When A is the top element and B is the next top element

B: Evaluate B ⊗ A

C: Place the result of step B back to stack

Step 5: Set the value equals to TOP element of the stack.

1. **Evaluate the expression 5 * (6 + 2) – 12 / 4**

   Ans : Convert to postfix notation

   5 * 6 2 + - 12 / 4

   5 6 2 + * - 12 4 /

   = 5 6 2 + * 12 4 / -

   Add **“) “** at the end of P

   P = 5 6 2 + * 12 4 / -)

Scanned Symbol	**Stack**
5	5
6	5, 6
2	5, 6, 2
+	5, 8
*	40
12	40, 12
4	40, 12, 4
/	40, 3
-	37

2. **Evaluate the expression (6 + 2) / (4 – 2 * 1)**

   Ans: Convert to postfix notation

   6 2 + / (4 – 2 1 *)

   6 2 + / 4 2 1 * -

   6 2 + 4 2 1 * - /

   P = 6 2 + 4 2 1 * - / **)**

Scanned Symbol	**Stack**
6	6
2	6 2
+	8

4	8 4
2	8 4 2
1	8 4 2 1
*	8 4 2
-	8 2
/	4

B. Infix to Postfix conversion

Here the operators used are **^ , * , / , + , -.** The following algorithm converts an Infix expression Q to postfix expression P. This algorithm also uses a stack which holds the left parenthesis and operators. We begin by pushing a Left parenthesis to stack and adding a right parenthesis at the end of Q.

Algorithm

Step 1: PUSH left parenthesis **"("** into stack and add right parenthesis **") "** at the end of Q.

Step 2: Scan the expression Q from Left – Right and repeat the step 3 to 6 for each element of Q until this stack is empty.

Step 3: If an operand occurs add it to P.

Step 4: If a Left parenthesis occurs then PUSH it to stack

Step 5: If an operator $\otimes$ occurs then

A: Repeatedly POP the stack and add to P, each operator which has same or higher precedence than $\otimes$

B: add $\otimes$ to stack

Step 6: If a Right parenthesis occurs then

A: Repeatedly POP from stack and add to P each operator until a left parenthesis occurs.

B: Remove the left parenthesis

Step 7: Exit

1. **Q = A + (B * C - (D / E ^ F) * G) * H**

   Ans : Add right parenthesis at the end of the expression

   Q = A + (B * C - (D / E ^ F) * G) * H **)**

Symbol Scanned	**Stack**	**p**
	(	
A	(	A
+	(+	A
(	(+ (	A
B	(+ (	AB
*	(+ (*	AB
C	(+ (*	ABC
-	(+ (-	ABC*
(	(+ (- (	ABC*
D	(+ (- (	ABC*D
/	(+ (- (/	ABC*D
E	(+ (- (/	ABC*DE
^	(+ (- (/ ^	ABC*DE
F	(+ (- (/ ^	ABC*DEF
)	(+ (-	ABC*DEF ^ /
*	(+ (- *	ABC*DEF ^ /
G	(+ (- *	ABC*DEF ^ /G
)	(+	ABC*DEF ^ /G * -
*	(+ *	ABC*DEF ^ /G * -
H	(+ *	ABC*DEF ^ /G * - H
)		ABC*DEF ^ /G * - H * +

2. **Q = ((A + B) * C – (D – E)) ^ (F + G)**

Ans:

Q = ((A + B) * C – (D – E)) ^ (F + G) **)**

Symbol Scanned	**Stack**	**p**
0	(	
(	((	
(	(((	
A	(((	A
+	(((+	A
B	(((+	AB
)	((	AB+
*	((*	AB+
C	((*	AB+C
-	((-	AB+C*
(	((- (	AB+C*
D	((- (	AB+C*D
-	((- (-	AB+C*D
E	((- (-	AB+C*DE
)	((-	AB+C*DE-
)	(	AB+C*DE--
^	(^	AB+C*DE--
(	(^ (	AB+C*DE--
F	(^ (	AB+C*DE--F
+	(^ (+	AB+C*DE--F
G	(^ (+	AB+C*DE--FG
)	(^	AB+C*DE--FG+
)		AB+C*DE—FG+^

3. **Q = (A + B) * C / D + E ^ F / G**

Ans :

Q = (A + B) * C / D + E ^ F / G)

Symbol Scanned	**Stack**	**p**
	(	
(	((	
A	((	A
+	((+	A
B	((+	AB
)	(	AB+
*	(*	AB+
C	(*	AB+C
/	(/	AB+C*
D	(/	AB+C*D
+	(+	AB+C*D/
E	(+	AB+C*D/E
^	(+ ^	AB+C*D/E
F	(+ ^	AB+C*D/EF
/	(+ /	AB+C*D/EF^
G	(+ /	AB+C*D/EF^G
)		AB+C*D/EF^G/+

10. LINEAR SEARCH AND BINARY SEARCH

1. **Linear search:** Small & unsorted arrays
2. **Binary search :** Large arrays & sorted arrays

1. Linear Search

- It means looking at each element of the array, in turn, until you find the target value.

Algorithm

```
1. Start
2. Read the ITEM to be searched
3. Set flag=0
4. Repeat for i=0 to N
5.          if  A[i]= =ITEM
6.                  print "item found"
7.                  flag=1
8.  If flag= =0
9.          print "item not found"
```

- In the **best case**, the target value is in the first element of the array. So the search takes some tiny, and constant, amount of time. Computer scientists denote this **O(1)** In real life, we don't care about the best case, because it so rarely actually happens.
- In the **worst case**, the target value is in the last element of the array. So the search takes an amount of time proportional to the length of the array. Computer scientists denote this **O(*n*)**
- In the **average case**, the target value is somewhere in the array. So on average, the target value will be in the middle of the array. So the search takes an amount of time proportional to half the length of the array – also proportional to the length of the array – **O(*n*)** again

2. Binary Search

- The general term for a smart search through sorted data is a ***binary search***.

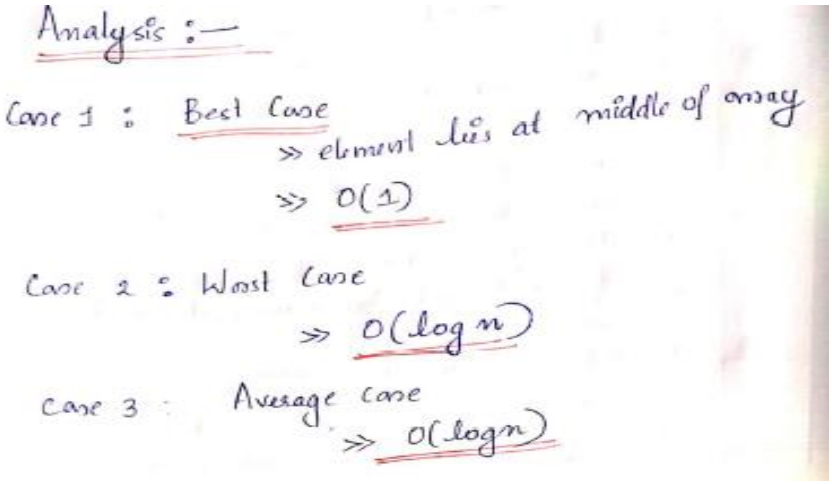
1. The initial search region is the whole array.
2. Look at the data value in the middle of the search region.
3. If you've found your target, stop.
4. If your target is less than the middle data value, the new search region is the lower half of the data.
5. If your target is greater than the middle data value, the new search region is the higher half of the data.
6. Continue from Step 2.

Algorithm

Let A be a sorted array with N elements

```
1. Start
2. Read the ITEM to be searched
3. Set beg=0, end=n-1, mid=(beg+end)/2
4. Repeat steps 5 to 9 while(beg<=end and A[mid]≠ ITEM)
5.         if ITEM< A[mid] then
6.                 set end=mid-1
7.         else
8.                 beg=mid+1
9.         mid=(beg+end)/2
10. If A[mid]=ITEM then
11.             print "item found"
12. Else print "element not found"
```

- Binary search reduces the work by half at each comparison



Compare Binary Search and Linear Search

	LINEAR SEARCH	BINARY SEARCH
1)	Data need not be sorted	• Needs to be sorted.
2)	Sequential access to data	• Random access to data
3)	Performs equality Comparisons	• Perform ordering Comparisons
4)	Time complexity is O(n)	• O(logn)

Q. Give an algorithm to perform binary search. Using the algorithm, search for elements 23 and 47 in the given set of elements [12 23 27 35 39 42 50]. (10).

Ans: (a) Searching for 23

	0	1	2	3	4	5	6
Array	12	23	27	35	39	42	50

Already sorted,

$l = 0, h = 6$

$mid = \left\lfloor \frac{0+6}{2} \right\rfloor = 3$, $A[mid] = A[3] = 35$

$35 > 23$

So, $h = mid - 1$

$= 3 - 1 = 2$

$l = 0, h = 2$

$mid = \left\lfloor \frac{0+2}{2} \right\rfloor = 1$, $A[mid] = A[1] = 23$

$23 = 23$

Search successful!!

(b) Searching for 47

	0	1	2	3	4	5	6
Array	12	23	27	35	39	42	50

Already Sorted,

$l = 0, h = 6, mid = \left\lfloor \frac{0+6}{2} \right\rfloor = 3$

$A[mid] = A[3] = 35$

$35 < 47$

So, $l = mid + 1 = 3 + 1 = 4$.

$l = 4$

$h = 6$, $mid = \left\lfloor \frac{4+6}{2} \right\rfloor = 5$

$A[5] = 42$

$42 < 47$

So, $l = mid + 1 = 5 + 1 = 6$

$l = 6$

$h = 6$ $mid = \left\lfloor \frac{6+6}{2} \right\rfloor = 6$.

$A[6] = 50$.

$50 > 47$

So, $h = mid - 1 = 6 - 1 = 5$

now, $l > h$ - condition ($l < h$ fails)

So, Search Unsuccessful

3 Q. Suppose an array contains elements {10, 13, 21, 33, 35, 44, 55}. Give the steps to find an element 35 using (i) linear search (ii) Binary search

(i) Linear Search :

	0	1	2	3	4	5	6
	10	13	21	32	35	44	55

Search key = 35

$i = 0$

$A[0] = 10 \quad 10 \neq 35$

$i = i+1 = 0+1 = 1$

$A[1] = 13, \quad 13 \neq 35$

$i = i+1 = 1+1 = 2$

$A[2] = 21 \quad 21 \neq 35$

$i = i+1 = 2+1 = 3$

$A[3] = 32 \quad 32 \neq 35$

$i = i+1 = 3+1 = 4$

$A[4] = 35 \quad 35 = 35$ Search Successfull

(ii) Binary Search :-

Array sorted,

$l = 0$

$h = 6$

$mid = \lfloor \frac{0+6}{2} \rfloor = 3, \quad A[3] = 32, \quad 32 < 35$

$l = mid + 1 = 4$

$h = 6$

$mid = \lfloor \frac{4+6}{2} \rfloor = \lfloor \frac{10}{2} \rfloor = 5$

$A[5] = 44$

$44 > 35$

$h = mid - 1 = 5-1 = 4$

$l = 4$

$mid = \lfloor \frac{4+4}{2} \rfloor = 4, \quad A[4] = 35 = 35$

Search Successfull //

MODULE 3 - LINKED LIST AND MEMORY MANAGEMENT

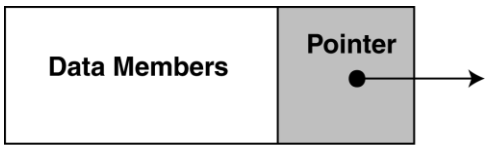
Self Referential Structures, Dynamic Memory Allocation, Singly Linked List-Operations on Linked List, Doubly Linked List, Circular Linked List, Stacks and Queues using Linked List, Polynomial representation using Linked List, Memory allocation and de-allocation-First-fit, Best-fit and Worst-fit allocation schemes

Disadvantage of using array

- Memory resizing is not possible. i.e. array size is fixed- it is a static data structure.
- Array requires continuous memory locations to store data.
- Wastage of memory

1. LINKED LIST

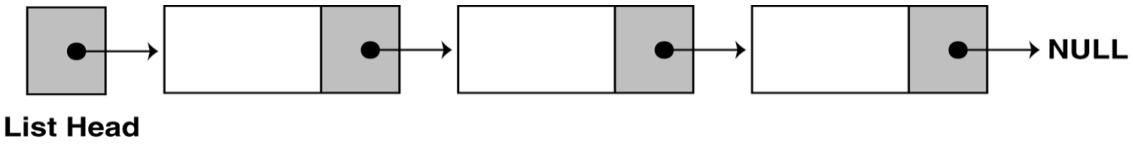
- A linked list is an ordered **collection of finite, homogeneous data elements called nodes** where the linear order is maintained by means of links or pointers.
- A linked list is a **dynamic data structure** where the amount of memory required can be varied during its use.
- In the linked list, the adjacency between the elements is maintained by means of **links or pointers.**
- A link or pointer actually is the **address** (memory location) of the subsequent element.
- An element in a linked list is a specially termed **node**, which can be viewed as shown in the figure.
- A node consists of two fields : **DATA** (to store the actual information) and **LINK** (to point to the next node)



links to the next node

- A linked list is called "linked" because each node in the series has a pointer that points to the next node in the list.

- ***Head*:** pointer to the first node
- The last node points to NULL



- Depending on the requirements the pointers are maintained, and accordingly the linked list can be classified into **three major groups**:
  1. Single linked list
  2. Circular linked list
  3. Double linked list.

SINGLE LINKED LIST

- In any single linked list, every **"Node"** contains two fields, **data** and **link**.
- The **data** field is used to store actual value of that node and **link** field is used to store the address of the next node in the sequence.
- Each node contains only one link which points to the subsequent node in the list.
- The header node points to the 1st node in the list
- The link field of the last node contain NULL(ϕ) value.
- Here one can move from left to right only. So it is also called one-way list

Representation of a linked list in memory

Two ways:

1. Static representation using array
2. Dynamic representation using free pool storage

1. Static representation

Two arrays are maintained:

- One for data and other for links.

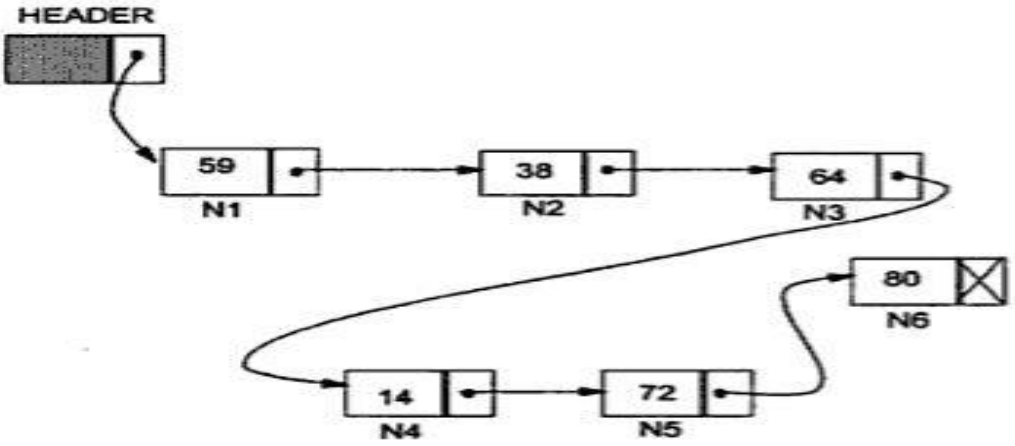
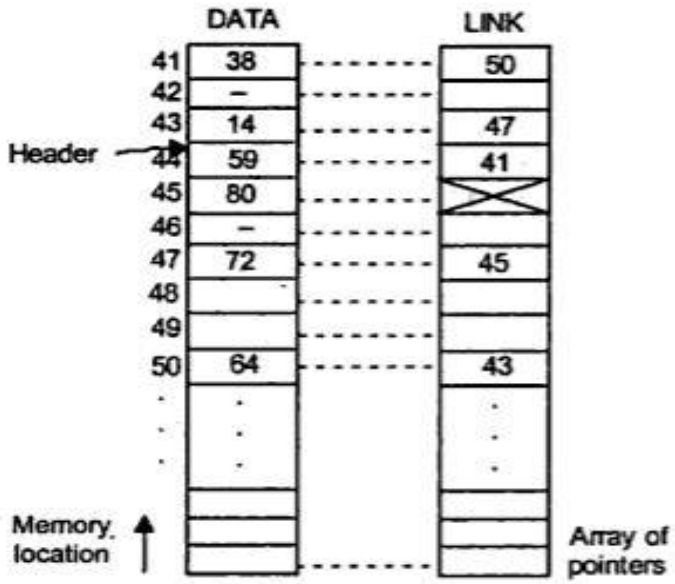


Fig. 3.2 A single linked list with 6 nodes.



2. Dynamic representation

- The efficient way of representing a linked list is using the **free pool of storage**.
- There is a
  - ***memory bank*** : Collection of free memory spaces &
  - ***memory manager*****:** a program
- Whenever a node is required, the request is placed to the memory manager.
- It will search the memory bank for the block. If found, it will be granted.
- ***Garbage collector*****:** Another program that returns the unused node to the memory bank.

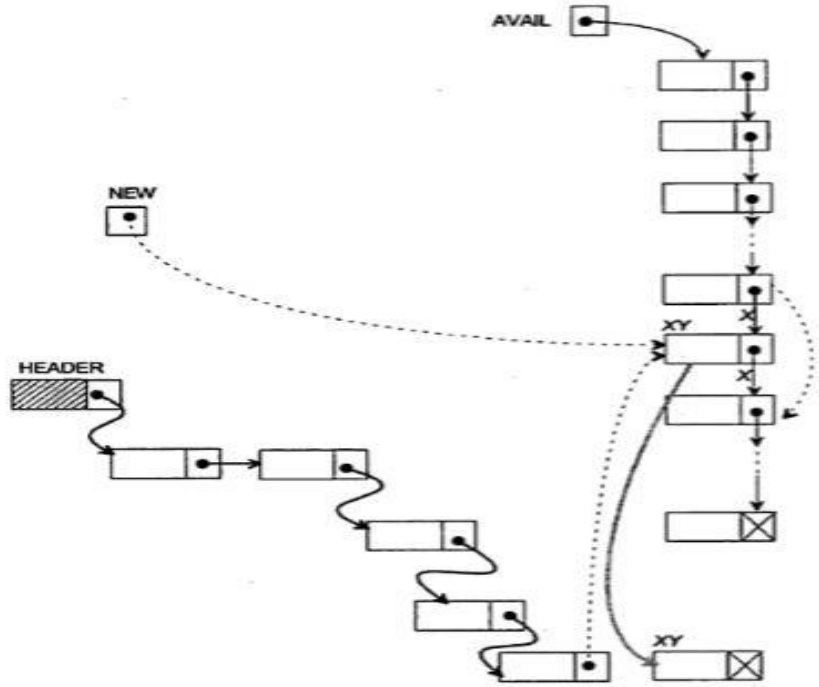
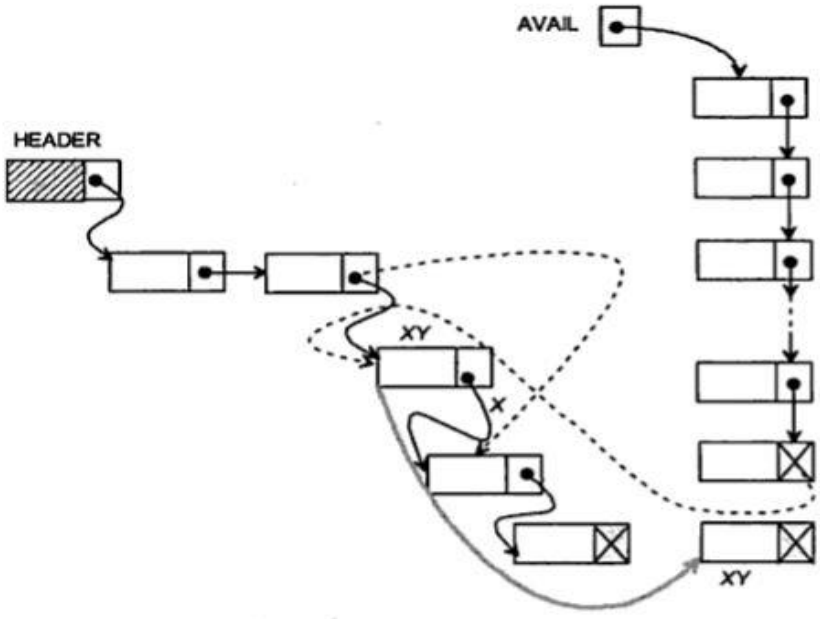


Fig. 3.4(a) Allocation of a node from memory bank to a linked list.

- Returning a node to memory bank



Operations on a Single Linked List

- Traversing the list
- Inserting a node into the list
- Deleting a node from the list
- Merging the list with another to make a larger list

Node creation

```
struct node
{
        int data;
                struct node *link;
};
```

- Function used for memory allocation is **"malloc"**

  New_node = (struct node*)malloc (sizeof (struct node))

1. Traversing a single linked list

- Here we visit every node in the list starting from the first node to the last one.

  Traverse()

  1. Set ptr=head; //initialize the pointer ptr
  2. While (ptr!=null) do
  3. print ptr->data
  4. ptr= ptr->link; //ptr now points to the next node

2. Inserting a node into the list

A. Inserting at the front (as a first element)

B. Inserting at the end(as a last element)

C. Inserting at any other position

A. Inserting at the front

Algorithm: Insert a new node temp with data 'item'

1. Create a pointer temp of type struct node
2. Create a new node temp using malloc function

   temp = (struct node*) malloc(sizeof(struct node));
3. if (temp==NULL)

4. print “memory underflow, no insertion”
5. else
6. temp->data= item
7. Set temp-> link=head
8. head=temp

B. Inserting at the end

- Here first we need to traverse the list to get the last node.

1. Create a pointer temp & ptr of type struct node
2. Create a new node temp using malloc function
   temp = (struct node*) malloc(sizeof(struct node));
3. Set ptr=head; //initialize the pointer ptr
4. While (ptr->link!=null) do
5. ptr= ptr->link; //ptr now points to the next node
6. ptr->link= temp
7. temp->data=item

C. Insertion- At any position in the list

1. Create a pointer temp & ptr of type struct node
2. Create a new node temp using malloc function
   temp = (struct node*) malloc(sizeof(struct node));
3. Read the value **key** of node after which a new node is to be placed
4. Set ptr=head
5. Repeat while (ptr-> data!=key) and (ptr->link!=NULL)
6. ptr=ptr-> link
7. If (ptr->link==NULL)
8. print “search fails”;
9. else
10. temp->link= ptr-> link
11. ptr->link= temp

3. Deleting a node from the list

- In a linked list, an element can be deleted:
  - A. From the 1st location
  - B. From the last location
  - C. From any position in the list

free(ptr) : It will free the location pointed by ptr

A. Deletion- From the beginning

1. Create a pointer ptr of type struct node
2. If (head==NULL) then exit
3. Else set ptr = head
4. set head=ptr-> link
5. free(ptr)

B. Deletion- From the end

1. Create a pointer ptr & temp of type struct node.
2. If (head -> link ==NULL) do step 3,4,5 else goto 6
3. ptr=head
4. head=NULL
5. free(ptr)
6. ptr=head
7. temp = head -> link
8. while(tem -> link !=NULL) do 9,10 else goto 11
9. ptr=temp
10. temp= tem -> link
11. ptr-> link =NULL
12. free (temp)

C. Deletion- From any position

1. Read the value **key** that is to be deleted
2. Create pointer ptr & temp of type struct node
3. Set ptr=head
4. if head=NULL then print underflow and exit
5. temp=ptr
6. while(ptr!=null) do step 7,8,9
7. If(ptr->data=key) then
   a) temp->link=ptr->link
   b) free(ptr) & exit
8. temp=ptr
9. Ptr = ptr->link

4. Merging

- Two linked list L1 and L2.
- Merge L2 after L1
  1. Set ptr= head1
  2. While(ptr->link!= NULL) do step 3 else goto step 4
  3. ptr=ptr->link
  4. ptr->link=head2
  5. Return(head2)
  6. Head=head1
  7. Stop

2. DOUBLY LINKED LIST

- Single linked list= one-way list
  - List can be traversed in one direction only
- Double linked list= Two-way list
  - List can be traversed in two directions

- two- way list is a linear collection of data elements called nodes where each node N is divided in to three parts
  - **Data field** contains the data of N
  - **LLINK field** contains the pointer to the preceding Node in the list
  - **RLINK field** contains the pointer to the next node in the list

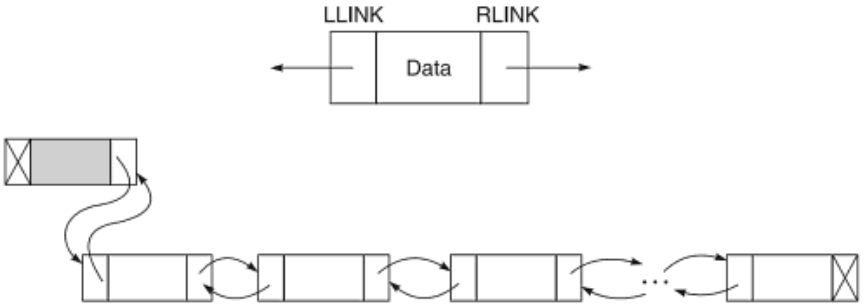


Figure 3.10 Structure of a node and a double linked list.

Operations on a Double Linked List

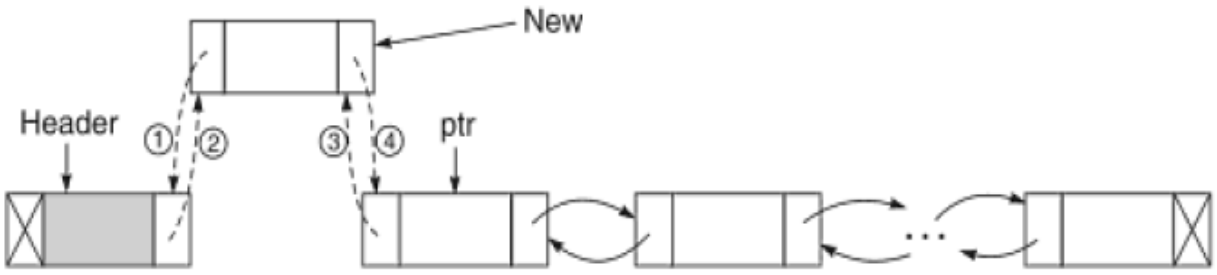
All the operations as mentioned for a single linked list can be implemented more efficiently using a double linked list.

Inserting a node into a Double Linked List (DLL)

Let us consider the algorithms of following cases of insertion in a DLL

i) Inserting a node in the front,
ii) Inserting a node at the end, and
iii) Inserting a node at any position in a double linked list.

i) Inserting a node in the front

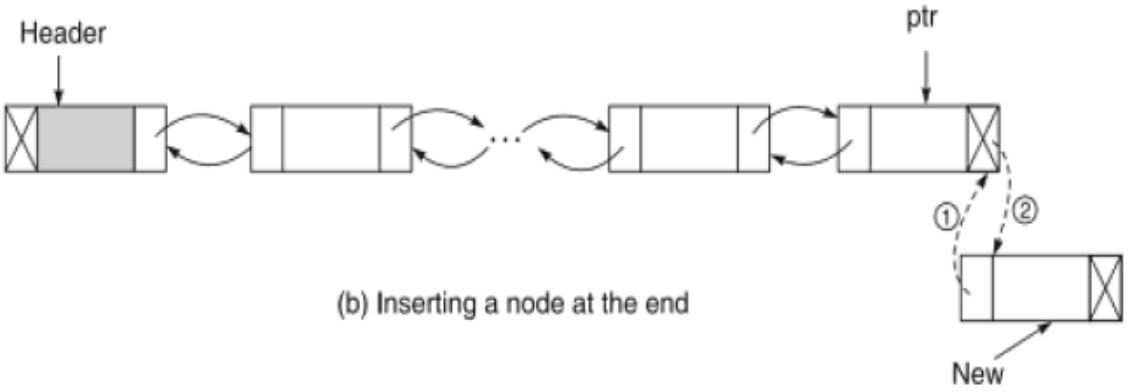


(a) Inserting a node in the front

Steps:

1. ptr = HEADER→RLINK // Points to the first node
2. new = **GetNode**(NODE) // Avail a new node from the memory bank
3. **If** (new ≠ NULL) then // If new node is available
4. new→LLINK = HEADER // Newly inserted node points the header as 1 in Figure 3.11(a)
5. HEADER→RLINK = new // Header now points to then new node as 2 in Figure 3.11(a)
6. new→RLINK = ptr // See the change in pointer shown as 3 in Figure 3.11(a)
7. ptr→LLINK = new // See the change in pointer shown as 4 in Figure 3.11(a)
8. new→DATA = *X* // Copy the data into the newly inserted node
9. **Else**
10. **Print** "Unable to allocate memory: Insertion is not possible"
11. **EndIf**
12. **Stop**

ii) Inserting a node at the end

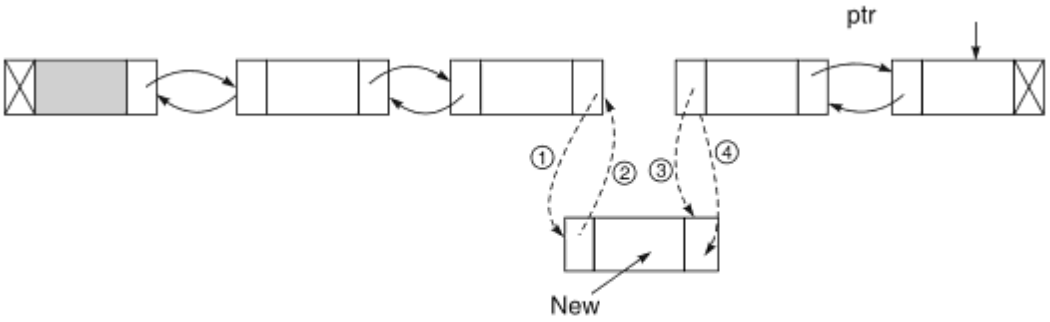


(b) Inserting a node at the end

Steps:

1. ptr = HEADER
2. **While** (ptr→RLINK ≠ NULL) **do** // Move to the last node
3. ptr = ptr→RLINK
4. **EndWhile**
5. new = **GetNode**(NODE) // Avail a new node
6. **If** (new ≠ NULL) **then** // If the node is available
7. new→LLINK = ptr // Change the pointer shown as 1 in Figure 3.11(b)
8. ptr→RLINK = new // Change the pointer shown as 2 in Figure 3.11(b)
9. new→RLINK = NULL // Make the new node as the last node
10. new→.DATA = *X* // Copy the data into the new node
11. **Else**
12. **Print** "Unable to allocate memory: Insertion is not possible"
13. **EndIf**
14. **Stop**

iii) Insertion- after an element key



(c) Inserting a node at any intermediate position

Figure 3.11 Inserting a node at various positions in a double linked list.

Steps:

```
1.  ptr = HEADER
2.  While (ptr→DATA ≠ KEY) and (ptr→RLINK ≠ NULL)      // Move to the key node if the
                                  // current node is not the KEY node or if the list reaches the end
3.      ptr = ptr→RLINK
4.  EndWhile
5.  new = GetNode(NODE)            // Get a new node from the pool of free storage
6.  If (new = NULL) then           // When the memory is not available
7.      Print (Memory is not available)
8.      Exit                                                  // Quit the program
9.  EndIf
10. If (ptr→RLINK = NULL) then     // If the KEY is not found in the list
11.     new→LLINK = ptr
12.     ptr→RLINK = new                                       // Insert at the end
13.     new→RLINK = NULL
14.     new→DATA = X               // Copy the information to the newly inserted node
15. Else                                                      // The KEY is available
16.     ptr1 = ptr→RLINK                                      // Next node after the key node
17.     new→LLINK = ptr            // Change the pointer shown as 2 in Figure 3.11(c)
18.     new→RLINK = ptr1           // Change the pointer shown as 4 in Figure 3.11(c)
19.     ptr→RLINK = new            // Change the pointer shown as 1 in Figure 3.11(c)
20.     ptr1→LLINK = new           // Change the pointer shown as 3 in Figure 3.11(c)
21.     ptr = new                                             // This becomes the current node
22.     new→DATA = X               // Copy the content to the newly inserted node
23. EndIf
24. Stop
```

Deleting a node from a Double Linked List (DLL)

Just like Insertion, Deleting a node from a Double Linked List consists of following cases:

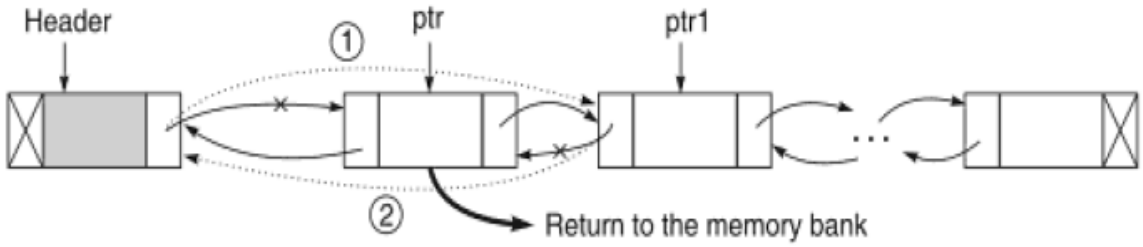
i) Deleting a Node from the front of a DLL,
ii) Deleting a Node at the end of a DLL, and
iii) Deleting a Node from any intermediate position.

i) Deletion- from 1st location

Steps:

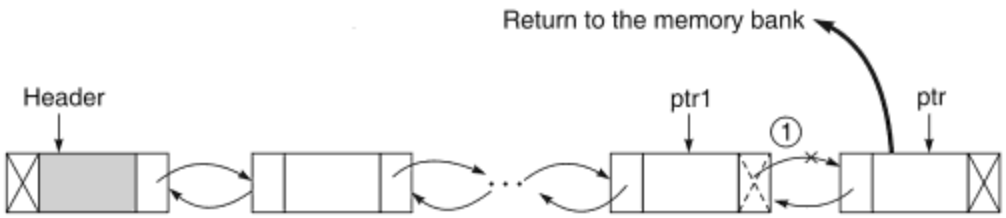
```
1.  ptr = HEADER→RLINK                                  // Pointer to the first node
2.  If (ptr = NULL) then                                // If the list is empty
3.      Print "List is empty: No deletion is made"
4.      Exit
5.  Else
6.      ptr1 = ptr→RLINK                                // Pointer to the second node
7.      HEADER→RLINK = ptr1          // Change the pointer shown as 1 in Figure 3.12(a)
8.      If (ptr1 ≠ NULL)        // If the list contains a node after the first node of deletion
9.          ptr1→LLINK = HEADER      // Change the pointer shown as 2 in Figure 3.12(a)
10.     EndIf
11.     ReturnNode (ptr)             // Return the deleted node to the memory bank
12. EndIf
13. Stop
```

Note that the algorithm *DeleteFront_DL* works even if the list is empty.



(a) Deleting the node placed at the front

ii) Deletion- from last location



(b) Deleting the node placed at the end

Algorithm DeleteEnd_DL

Input: A double linked list with data.

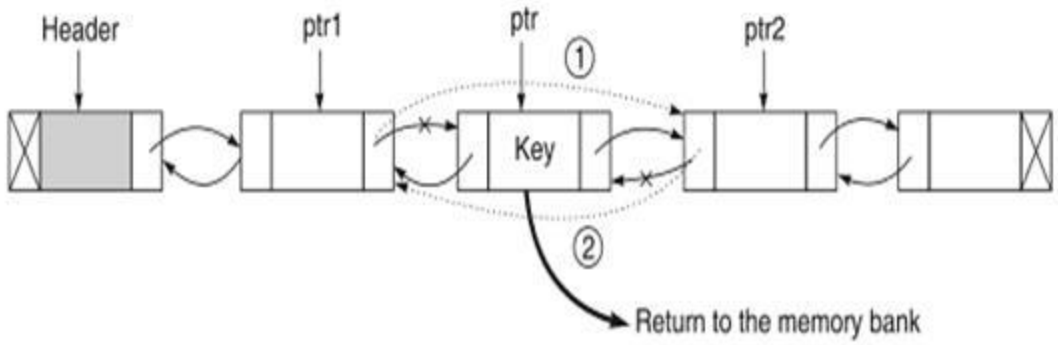
Output: A reduced double linked list.

Data structure: Double linked list structure whose pointer to the header node is the *HEADER*.

Steps:

1. ptr = HEADER
2. **While** (ptr→RLINK ≠ NULL) **do** // Move to the last node
3. ptr = ptr→RLINK
4. **EndWhile**
5. **If** (ptr = HEADER) **then** // If the list is empty
6. **Print** "List is empty: No deletion is made"
7. **Exit** // Quit the program
8. **Else**
9. ptr1 = ptr→LLINK // Pointer to the last but one node
10. ptr1→RLINK = NULL // Change the pointer shown as 1 in Figure 3.12(b)
11. **ReturnNode** (ptr) // Return the deleted node to the memory bank
12. **EndIf**
13. **Stop**

iii) Deletion- from intermediate location



(c) Deleting a node from any intermediate position

Steps:

```
1.  ptr = HEADER→RLINK                                   // Move to the first node
2.  If (ptr = NULL) then
3.      Print "List is empty: No deletion is made"
4.      Exit
5.  EndIf                                                // Quit the program
6.  While (ptr→DATA ≠ KEY) and (ptr→RLINK ≠ NULL) do     // Move to the desired node
7.      ptr = ptr→RLINK
8.  EndWhile
9.  If (ptr→DATA = KEY) then                             // If the node is found
10.     ptr1 = ptr→LLINK                                 // Track the predecessor node
11.     ptr2 = ptr→RLINK                                 // Track the successor node
12.     ptr1→RLINK = ptr2          // Change the pointer shown as 1 in Figure 3.12(c)
13.     If (ptr2 ≠ NULL) then                            // If the deleted node is the last node
14.         ptr2→LLINK = ptr1      // Change the pointer shown as 2 in Figure 3.12(c)
15.     EndIf
16.     ReturnNode(ptr)                                  // Return the free node to the memory bank
17. Else
18.     Print "The node does not exist in the given list"
19. EndIf
20. Stop
```

3. CIRCULAR LINKED LIST

- In a single linked list, the link field of the last node is null.
- If we utilize this link field to store the pointer of the header node, a number of advantages can be gained.
- A linked list, whose last node points back to the first node, instead of containing the null pointer is called a **circular list**

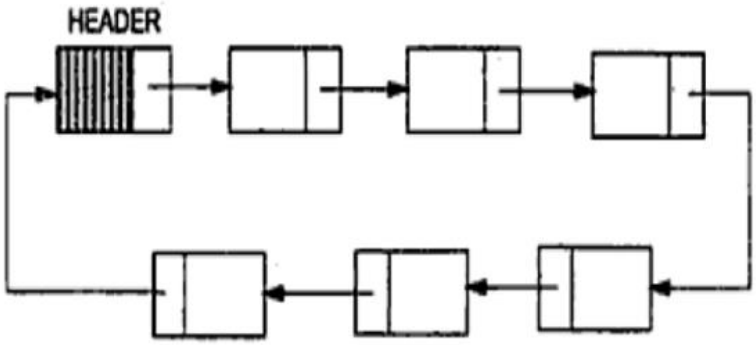


Fig. 3.8 A circular linked list.

- **<u>Advantages:</u>**

1. **Accessibility of a member node** – here every member node is accessible from any node by merely chaining through the list

eg: Finding of earlier occurrence or post occurrence of a data will be easy

2. **Null link problem-** Null value in next field may create problem during the execution of the program if proper care is not taken

3. Some **easy-to-implement operations** - Operations like merging, splitting, deletion, dispose of an entire list etc can be done easily with circular list

- **Disadvantages:**
  - If not cared, system may get trap into in infinite loop
    - It occurs when we are unable to detect the end of the list while moving from one node to the next
    - Solution: Special node can be maintained with data part as NULL and this node does not contain any valid information. So its just a wastage of memory space

Insertion in circular linklist

- We want to insert data 'X' after a given position, 'pos'
- Here we are using a pointer called last, which points to the last node
  1. Create a pointer temp and q of type struct node
  2. Set q=last->link and i=1
  3. While(i<pos) do step 4
  4. q=q->link & increment I
  5. Create a new node temp usin malloc function

     temp = (struct node*) malloc(sizeof(struct node));
  6. temp->link=q->link
  7. temp->data = X
  8. q->link = temp

Deletion in circular Linked List

1. if last = NULL print under flow and exit

// **Linkedlist containing only one node**

2. If last -> link = last & last -> data = key then do the steps 3,4,5

3. temp= last

4. Last = NULL

5. free(temp)

6. q = last ->link

//Deleting first node

7. if q->data =key do 8,9,10

8. temp = q

9. Last->link= q->link

10. free(temp)

// deleting Middle node

11. Repeat steps 12 to 16 while q->link!=last

12. if q->link->data =key do step 13,14,15

13. temp = q->link

14. q->link= tem->link

15. Free(temp)

//Deleting last node

16. If q->link ->data = key

17. temp = q->link

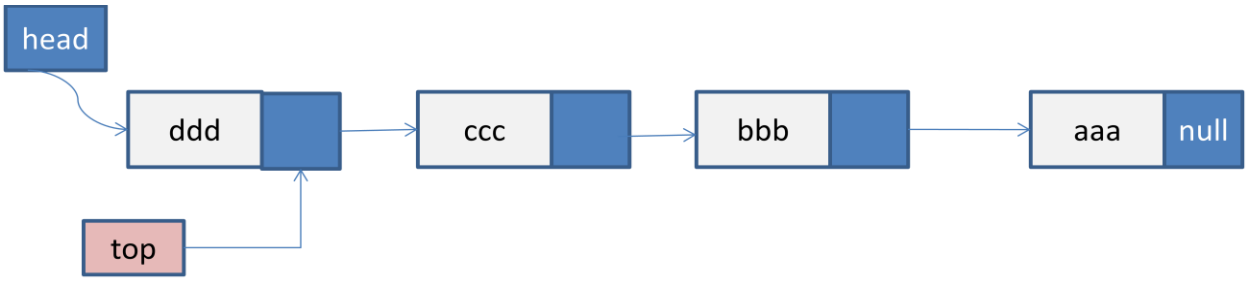
18. q->link=last->link

19. Free(temp)

20. Last=q

4. STACKS USING LINKED LIST

- Stack can also be represented using a singly linked list.
- Linked lists have many advantages compared to arrays.
- In linked list, the **DATA** field contains the elements of stack and **LINK** field points to the next element in the stack.
- Here **Push** operation is accomplished by **inserting a new node in the front** or start of the list.
- **Pop** is done by **removing the element from the front** of the list



Insertion- At the beginning

Algorithm: PUSH()

1. Create a new node temp //struct node *temp = (struct node*) malloc(sizeof(struct node));
2. If (temp==NULL)
3. print "memory underflow, no insertion"
4. else
5. temp->data= item
6. temp-> link=head
7. head=temp

Deletion- From the beginning

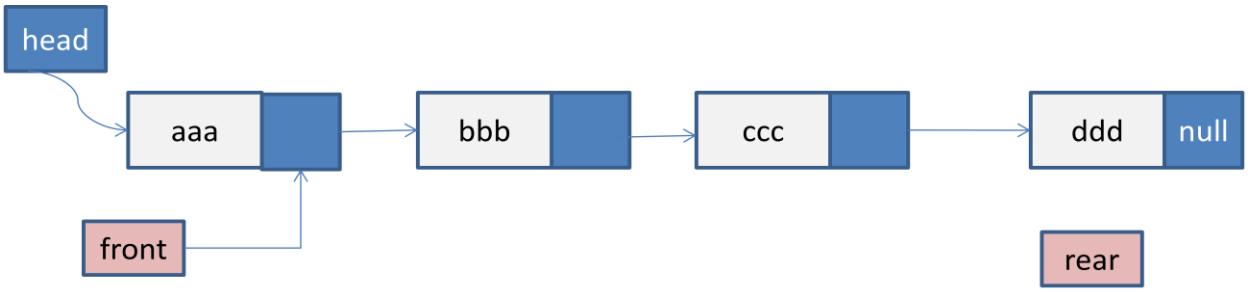
Algorithm: POP()

1. Start
2. If(head==null)
3. print "underflow"
4. Else
5. print the deleted element 'head-> data'
6. head= head->link

5. QUEUES USING LINKED LIST

- Queue can also be represented using a singly linked list.
- Linked lists have many advantages compared to arrays.
- In linked list, the Data field contains the elements of queue and Next pointer points to the next element in the queue.
- Here **enqueue** operation is accomplished by **inserting a new node in the tail** or end of the list.

- **Dequeue** is done by **removing the element from the beginning** of the list



Insertion- At the end

Algorithm: Enqueue()

1. Set ptr=head; //initialize the pointer ptr
2. While (ptr->link!=null) do
3. ptr= ptr->link; //ptr now points to the next node
4. ptr->link= temp
5. temp->data=item

Deletion – At the front

Algorithm: DEQUEUE()

1. Start
2. If(head==null)
3. print “underflow”
4. Else
5. print the deleted element ‘head-> data’
6. head= head-> link

6. POLYNOMIAL REPRESENTATION USING LINKED LIST

Polynomial having a single variable

Let us consider the general form of a polynomial having a single variable:

$$P(x) = a_n x^{e_n} + a_{n-1} x^{e_{n-1}} + \cdots + a_1 x^{e_1}$$

where $a_i x^{e_i}$ is a term in the polynomial so that a_i is a non-zero coefficient and e_i is the exponent. We will assume an ordering of the terms in the polynomial such that $e_n > e_{n-1} > \ldots > e_2 > e_1 \geq 0$. The structure of a node in order to represent a term can be decided as shown below:

Considering the single linked list representation, a node should have three fields: COEFF (to store the coefficient a_i), EXP (to store the exponent e_i) and a LINK (to store the pointer to the next node representing the next term). It is evident that the number of nodes required to represent a polynomial is the same as the number of terms in the polynomial. An additional node may be considered for a header. As an example, let us consider that the single linked list representation of the polynomial $P(x) = 3x^8 - 7x^6 + 14x^3 + 10x - 5$ would be stored as shown in Figure 3.18.

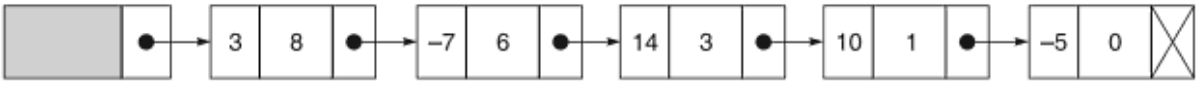


Figure 3.18 Linked list representation of a polynomial (single variable).

Note that the terms whose coefficients are zero are not stored here. Next let us consider two basic operations, namely the addition and multiplication of two polynomials using this representation.

Polynomial addition

In order to add two polynomials, say P and Q, to get a resultant polynomial R, we have to compare their terms starting at their first nodes and moving towards the end one by one. Two pointers Pptr and Qptr are used to move along the terms of P and Q. There may arise three cases during the comparison between the terms of two polynomials.

(i) **Case 1:** The exponents of two terms are equal. In this case the coefficients in the two nodes are added and a new term is created with the values

$$\text{Rptr}\rightarrow\text{COEFF} = \text{Pptr}\rightarrow\text{COEFF} + \text{Qptr}\rightarrow\text{COEFF}$$

and

$$\text{Rptr}\rightarrow\text{EXP} = \text{Pptr}\rightarrow\text{EXP}$$

(ii) **Case 2:** Pptr→EXP > Qptr→EXP, i.e. the exponent of the current term in P is greater than the exponent of the current term in Q. Then, a duplicate of the current term in P is created and inserted in the polynomial R.

(iii) **Case 3:** Pptr→EXP < Qptr→.EXP, i.e. the case when the exponent of the current term in P is less than the exponent of the current term in Q. In this case, a duplicate of the current term of Q is created and inserted in the polynomial R. The algorithm *PolynomialAdd_LL* is described as below:

Algorithm PolynomialAdd_LL

Input: Two polynomials P and Q whose header pointers are *PHEADER* and *QHEADER*.
Output: A polynomial R is the sum of P and Q having the header *RHEADER*.
Data structure: Single linked list structure for representing a term in a single variable polynomial.

Steps:

1. Pptr = PHEADER→LINK, Qptr = QHEADER→LINK
   //Get a header node for the resultant polynomial//
2. RHEADER = **GetNode**(NODE)
3. RHEADER→LINK = NULL, RHEADER→EXP = NULL, RHEADER→COEFF = NULL
4. Rptr = RHEADER // Current pointer to the resultant polynomial R
5. **While** (Pptr ≠ NULL) and (Qptr ≠ NULL) **do**
6. **CASE:** Pptr→EXP = Qptr→EXP // Case 1
7. new = **GetNode** (NODE)
8. Rptr→LINK = new, Rptr = new
9. Rptr→COEFF = Pptr→COEFF + Qptr→COEFF
10. Rptr→EXP = Pptr→EXP
11. Rptr→LINK = NULL
12. Pptr = Pptr→LINK, Qptr = Qptr→LINK
13. **CASE:** Pptr→EXP > Qptr→EXP // Case 2
14. new = **GetNode** (NODE)
15. Rptr→LINK = new, Rptr = new
16. Rptr→COEFF = Pptr→COEFF
17. Rptr→EXP = Pptr→EXP
18. Rptr→LINK = NULL
19. Pptr = Pptr→LINK
20. **CASE:** Pptr→EXP < Qptr→EXP // Case 3
21. new = **GetNode** (NODE)
22. Rptr→LINK = new, Rptr = new
23. Rptr→COEFF = Qptr→COEFF
24. Rptr→EXP = Qptr→EXP
25. Rptr→LINK = NULL
26. Qptr = Qptr→LINK
27. **EndWhile**

```
28.     If (Pptr ≠ NULL) and (Qptr = NULL) then        // To add the trailing part of P, if any
29.     While (Pptr ≠ NULL) do
30.         new = GetNode(NODE
31.         Rptr→LINK = new, Rptr = new
32.         Rptr→COEFF = Pptr→COEFF
33.         Rptr→EXP = Pptr→Exp
34.         Rptr→LINK = NULL
35.         Pptr = Pptr→LINK
36.     EndWhile
37. EndIf
38. If (Pptr = NULL) and (Qptr ≠ NULL) then        //To add the trailing part of Q, if any
39.     While (Qptr ≠ NULL) do
40.         new = GetNode(NODE)
41.         Rptr→LINK = new, Rptr = new
42.         Rptr→COEFF = Qptr→COEFF
43.         Rptr→EXP = Qptr→EXP
44.         Rptr→LINK = NULL
45.         Qptr = Qptr→LINK
46.     EndWhile
47. EndIf
48. Return(RHEADER)
49. Stop
```

Polynomial multiplication

Suppose, we have to multiply two polynomials P and Q so that the result will be stored in R, another polynomial. The method is quite straightforward: let Pptr denote the current term in P and Qptr be that of in Q. For each term of P we have to visit all the terms in Q; the exponent values in two terms are added (R→EXP = P→EXP + Q→EXP), the coefficient values are multiplied (R→COEFF = P→COEFF × Q→COEFF), and these values are included into R in

such a way that if there is no term in R whose exponent value is the same as the exponent value obtained by adding the exponents from P and Q, then create a new node and insert it to *R* with the values so obtained (that is, R→COEFF, and R→EXP); on the other hand, if a node is found in *R* having same exponent value R→EXP, then update the coefficient value of it by adding the resultant coefficient (R→COEFF) into it. The algorithm *PolynomialMultiply_LL* is described as below:

Algorithm PolynomialMultiply_LL
Input: Two polynomials P and Q having their headers as *PHEADER*, *QHEADER*.
Output: A polynomial R storing the result of multiplication of P and Q.
Data structure: Single linked list structure for representing a term in a single variable polynomial.

Steps:

1. Pptr = PHEADER, Qptr = QHEADER
   / Get a node for the header of R */*
2. RHEADER = **GetNode**(NODE)
3. RHEADER→LINK = NULL, RHEADER→COEFF = NULL, RHEADER→EXP = NULL
4. **If** (Pptr→LINK = NULL) or (Qptr→LINK = NULL) **then**
5. Exit // No valid operation possible
6. **EndIf**
7. Pptr = Pptr→.LINK
8. **While** (Pptr ≠ NULL) **do** // For each term of P
9. **While** (Qptr ≠ NULL) **do**
10. C = Pptr→COEFF × Qptr→COEFF
11. X = Ppt→EXP + Qptr→EXP

    / Search for the equal exponent value in R */*
12. Rptr = RHEADER
13. **While** (Rptr ≠ NULL) and (Rptr→EXP > X) **do**
14. Rptr1 = Rptr
15. Rptr = Rptr→LINK
16. **If** (Rptr→EXP = X) **then**
17. Rpt→COEFF = Rptr→COEFF + C
18. **Else** // Add a new node at the correct position in R
19. new = **GetNode**(NODE)
20. new→EXP = X, new→COEFF = C
21. **If** (Rptr→LINK = NULL) **then**
22. Rptr→LINK = new // Append the node at the end
23. new→LINK = NULL
24. **Else**
25. Rptr1→LINK = new // Insert the node in ordered position
26. new→LINK = Rptr
27. **EndIf**
28. **EndIf**
29. **EndWhile**
30. **EndWhile**
31. **EndWhile**
32. **Return** (RHEADER)
33. **Stop**

Polynomials having multiple variables

So far we have considered the case of a polynomial of a single variable. The idea now can be extended to represent any polynomial with two variables, three variables, and so on. Below is a structure of a node that will be suitable to represent a polynomial with three variables x, y and z using a single linked list.

Writing procedures to manipulate such polynomials is as simple as the earlier procedures for polynomials with single variables. These are left as an assignment to the reader.

MODULE 3 – PART 2

Memory Management

The basic task of any program is to manipulate data. These data should be stored in memory during their manipulation. There are two memory management schemes for the storage allocations of data:

1. Static storage management
2. Dynamic storage management

Static Storage Management

In the case of the *static storage management* scheme, the net amount of memory required for various data for a program is allocated before the start of the execution of the program. Once memory is allocated, it can neither be extended nor be returned to the memory bank for the use of other programs at the same time.

Dynamic Storage Management

On the other hand, the *dynamic storage management* scheme allows the user to allocate and deallocate as per the requirement during the execution of programs. This dynamic memory management scheme is suitable in multiprogramming as well as in single-user environment where generally more than one program reside in the memory and their memory requirement can be known only during their execution. An operating system (OS) generally provides the service of dynamic memory management. The data structure for implementing such a scheme is a linked list.

There are various principles on which the dynamic memory management scheme is based. These principles are listed below.

1. *Allocation schemes:* Here, we discuss how a request for a memory block will be serviced. There are two strategies:
   (a) Fixed block allocation
   (b) Variable block allocation. There are four strategies under this:
      (i) First-fit and its variant
      (ii) Next-fit
      (iii) Best-fit
      (iv) Worst-fit
2. *Deallocation schemes:* Here, we discuss how to return a memory block to the memory bank whenever it is no longer required. Two strategies are known for the deallocation schemes:
   (i) Random deallocation
   (ii) Ordered deallocation

We will discuss two more systems for the implementation of allocation and deallocation schemes:

1. Boundary tag system
2. Buddy system

There is, again, one more principle called *garbage collection* to maintain a memory bank so that it can be utilized efficiently.

Now we discuss below the dynamic memory management schemes and possible use of a linked list therein.

Memory Representation

A memory bank or a pool of free storages is often a collection of non-contiguous blocks of memory. Their linearity can be maintained by means of pointers between one block to another, or in other words a memory bank is a linked list where links maintain the adjacency of blocks. Regarding the size of the blocks, there are two practices: fixed block storage and variable block storage.

Fixed Block Storage

This is the simplest storage maintenance method. Here each block is of the same size. The size is determined by the system manager (user). Here, the memory manager (a program of OS) maintains a pointer AVAIL which points a list of non-contiguous memory blocks. The below figure shows a memory bank with fixed size blocks.

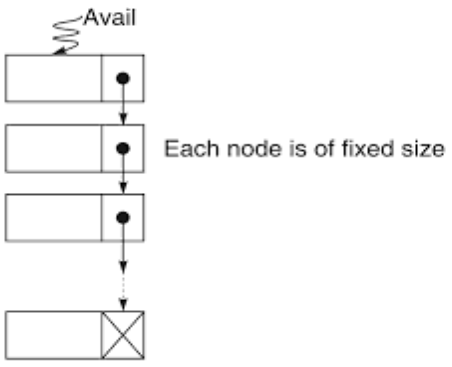


Figure Pool of free storages with fixed size blocks.

A user program communicates with the memory manager by means of two functions *GetNode*( ) and *ReturnNode*( ), which are discussed below.

Procedure GetNode
Input: This procedure avails a block from memory bank for a data whose type is represented by *NODE*.
Output: Returns a pointer to the memory block if available else a message.

```
Steps:
1.  If (AVAIL = NULL) then                                  // Memory is exhausted
        Print "Memory is insufficient"
2.  Else
3.      ptr = AVAIL
4.      AVAIL = AVAIL→LINK
5.      Return(ptr)                  // Return pointer of the available block to the caller
6.  EndIf
7.  Stop
```

The procedure *GetNode* is to get a memory block to store data of type *NODE* (by passing this as an argument we need to mention the size of the memory block required). This procedure when invoked by a program, returns a pointer to the first block in the pool of free storage. The AVAIL then points to the next block. The link modification is shown (by the dotted line) in Figure 3.20. If AVAIL = NULL, it indicates that no more memory is available for allocation.

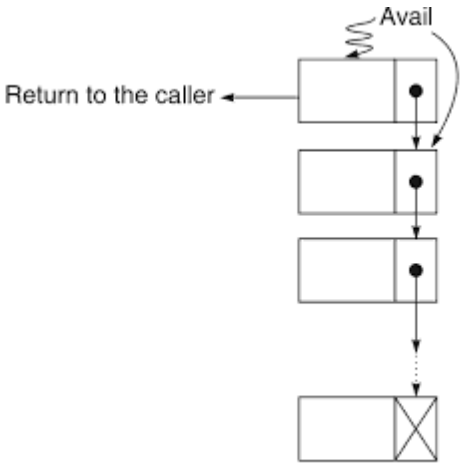


Figure 3.20 Getting a block from the memory bank.

Similarly, whenever a memory block is no more required, it can be returned to the memory bank through a procedure *ReturnNode* which is stated below:

Procedure ReturnNode
Input: This procedure returns a block of memory referenced by the pointer *PTR*.
Output: The memory block is returned to the memory bank.
Remark: Naïve approach, that is, get it as you find it.

Steps:

```
1. ptr1 = AVAIL
2. While (ptr1→LINK ≠ NULL) do          // Move to the end of the list
3.     ptr1 = ptr1→LINK
4. EndWhile
5. ptr1→LINK = PTR
6. PTR→LINK = NULL
7. Stop
```

The procedure *ReturnNode* appends a returned block (bearing pointer *PTR*) at the end of the pool of free storage pointed by AVAIL. Change in pointers can be seen in Figure 3.21 as a dotted line.

So far as the implementation of fixed block allocation is concerned, this is the most simple strategy. But the main drawback of this strategy is the wastage of space. For example, suppose each memory block is of size 1K (1024 bytes); now for a request of a memory block, say, of size 1.1K we have to avail 2 blocks (that is 2K memory space), thus wasting 0.9K memory space. Making the size of the block too small reduces the wastage of space; however, it also reduces the overall performance of the scheme.

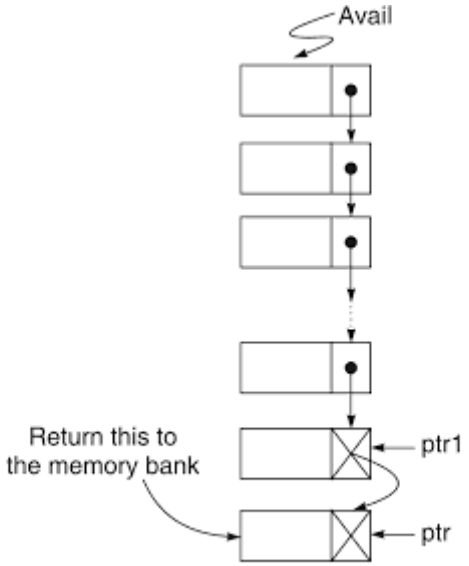


Figure 3.21 Returning a block to the memory bank.

Variable Block Storage

To overcome the disadvantages of fixed block storage, we can maintain blocks of variable sizes instead of those of fixed sizes. Procedures for *GetNode* and *ReturnNode* with this storage management are stated below.

Procedure GetNode
Input: This procedure gets a block of memory from the memory bank.
Output: Returns a pointer to the memory block if available else a message.
Remark: With variable sized memory blocks policy.

Steps:

1. **If** (AVAIL = NULL) **then**
2. **Print** "Memory bank is insufficient"
3. **Exit** // Quit the program
4. **EndIf**

```
5.  ptr = AVAIL
6.  While (ptr→LINK ≠ NULL) and (ptr→SIZE < SizeOf(NODE)) do
                                                    // Move to the right block
7.      ptr1 = ptr
8.      ptr = ptr→LINK
9.  EndWhile
10. If (ptr→.LINK = NULL) and (ptr→SIZE < SizeOf(NODE)) then
11.     Print "Memory request is too large: Unable to serve"
12. Else
13.     ptr1→LINK = ptr→LINK
14.     Return(ptr)
15. EndIf
16. Stop
```

This procedure assumes that blocks of memory are stored in ascending order of their sizes. The node structure maintains a field to store the size of a block, namely SIZE. *SizeOf*( ) is a procedure that will return the size of a node (see Figure 3.22). Note that the above procedure will return a block of exactly the same size or more than the size that a user program requests. Next, let us describe the procedure *ReturnNode* to dispose a block into the pool of free storage in the ascending order of block sizes.

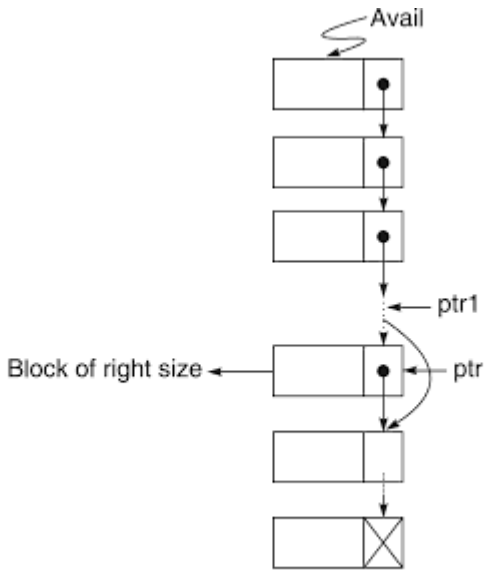


Figure 3.22 Availing a node from a pool of free storages with variable sized blocks.

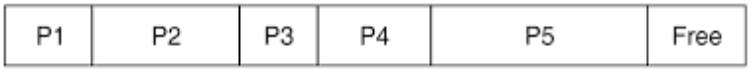
Procedure ReturnNode
Input: This procedure returns a block of memory referenced by the pointer *PTR*.
Output: The memory block is returned to the memory bank.
Remark: With variable sized memory blocks policy.

Steps:

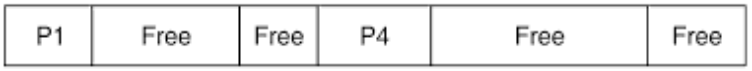
1. ptr1 = AVAIL
2. **While** (ptr1→SIZE < PTR→SIZE) and (ptr1→LINK ≠ NULL)) **do**
   // Move to the right position
3. ptr2 = ptr1
4. ptr1 = ptr1→LINK
5. **EndWhile**
6. **If** (PTR→SIZE < ptr1→SIZE) **then**
7. ptr2→LINK = PTR
8. PTR→LINK = ptr1
9. **Else**
10. ptr1→LINK = PTR
11. PTR→LINK = NULL
12. **EndIf**
13. **Stop**

So far as memory space utilization is concerned, the variable sized block storage policy is preferred to that of the fixed sized block storages. During the discussion on procedure *GetNode* and *ReturnNode*, we have assumed that memory blocks of various sizes are available and they are linked with each other. But the actual case is different. Note that initially when there is no program in the memory, the entire memory is a block. The size of the blocks is then automatically generated, through the use of memory system, with several requests of various sizes and their returns. This is explained in Figure 3.23. We assume that the memory system starts with five programs: P1, P2, P3, P4 and P5.

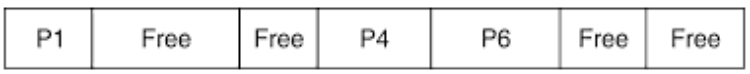
(a) Total memory space, no program is resided



(b) Initially five programs are alloted their memory in order of requests



(c) P2, P3 and P5 returned their spaces to the memory system; blocks of variable sizes have been created



(d) Another request came from another procedure say P6, and the required space is alloted. A block of bigger size is required and fragmented

Figure 3.23 Partition of the memory into smaller blocks during dynamic storage allocation.

From the foregoing discussions, it can be concluded that the dynamic memory management system should provide the following services:

(a) Searching the memory for a block of requested size and servicing the request (allocation)
(b) Handling a free block when it is returned to the memory manager.
(c) Coalescing the smaller free blocks into larger block(s) (garbage collection and compaction).

To serve these facilities, we have two memory management systems: boundary tag system and buddy system.

Storage Allocation Strategies

In order to service a request for a memory block of given size, any one of the following well-known strategies can be used.

(a) **First-Fit** allocation
(b) **Best-Fit** allocation
(c) **Worst-Fit** Allocation
(d) **Next-Fit** Allocation

Let us discuss all these allocation strategies assuming that the memory system has to serve a request for a block of size N.

First-fit storage allocation

This is the simplest of all the storage allocation strategies. Here the list of available storages is searched and as soon as a free storage block of size $\geq N$ is found the pointer of that block is sent to the calling program after retaining the residue space. Thus, for example, for a block of size 2K, if the first-fit strategy finds a block of 7K, then after retaining a storage block of size 5K, 2K memory will be sent to the caller.

Best-fit storage allocation

This strategy will not allocate a block of size $> N$, as it is found in the first-fit method; instead it will continue searching to find a suitable block so that the block size is closer to the block size of request. For example, for a request of 2K, if the list contains the blocks of sizes, 1K, 3K, 7K, 2.5K, 5K, then it will find the block of size 2.5K as the suitable block for allocation. From this block after retaining 0.5K, pointer for 2K block will be returned.

Worst-fit storage allocation

The best-fit strategy finds a block which is small and nearest to the block size as requested, whereas the worst-fit strategy is a reverse of the best-fit strategy. It allocates the largest block available in the available storage list. The idea behind the worst-fit is to reduce the rate of production of small blocks which are quite common when the best-fit strategy is used for memory allocation.

Worst-fit prevents what the best-fit does; it reduces the rate of production of small blocks. However, some simulation studies indicate that the worst-fit allocation is not very effective in reducing wasted memory in processing a series of requests.

Next-fit as discussed, is a modification of the first-fit strategy. In general, the next-fit does not outperform the first-fit in reducing the amount of wasted memory.

So far as processing speed is concerned, a rough comparison can be made as per their order of superiority, which is specified below:

Next-fit > First-fit > Best-fit, Worst-fit

The boundary tag system uses a slight variation of the above-mentioned strategies. It uses the next-fit storage allocation strategy and does not consider a block for allocation which if allocated leaves a small block of size $< \varepsilon$. That is, our pool of free storage will not contain any free block of size $< \varepsilon$, where ε is chosen by a statistic based on the nature of requests. As per the next-fit strategy, after allocating a block, search for the subsequent request will continue from the next block of these allocated blocks, and let AVAIL store the address of such a next block.

Module IV - Trees and Graphs

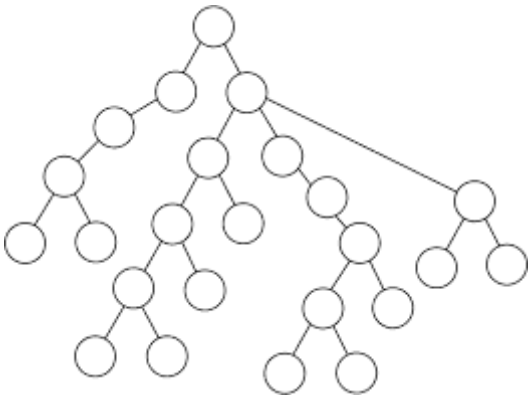
Trees, Binary Trees-Tree Operations, Binary Tree Representation, Tree Traversals, Binary Search Trees- Binary Search Tree Operations

Graphs, Representation of Graphs, Depth First Search and Breadth First Search on Graphs, Applications of Graphs

TREES

- Arrays, linked lists, stacks and queues were examples of **linear data structures** in which elements are arranged in a **linear fashion** (ie, one dimensional representation).
- Tree is another very useful data structure in which elements are appearing in a **non-linear fashion**, which requires a two dimensional representation.

Example Figure:

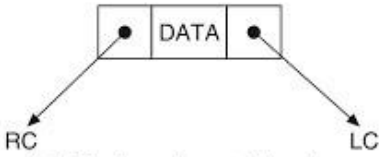


Basic Terminologies

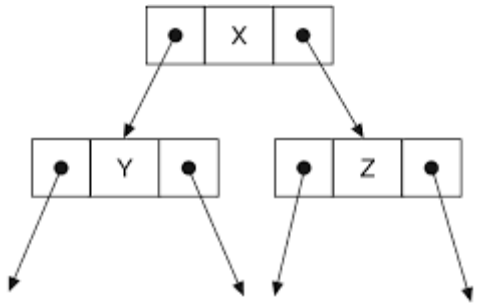
Node. This is the main component of any tree structure. The concept of the node is the same as that used in a linked list. A *node* of a tree stores the actual data and links to the other node. Figure 7.4(a) represents the structure of a node.

Parent. The *parent* of a node is the immediate predecessor of a node. Here, X is the parent of Y and Z. See Figure 7.4(b).

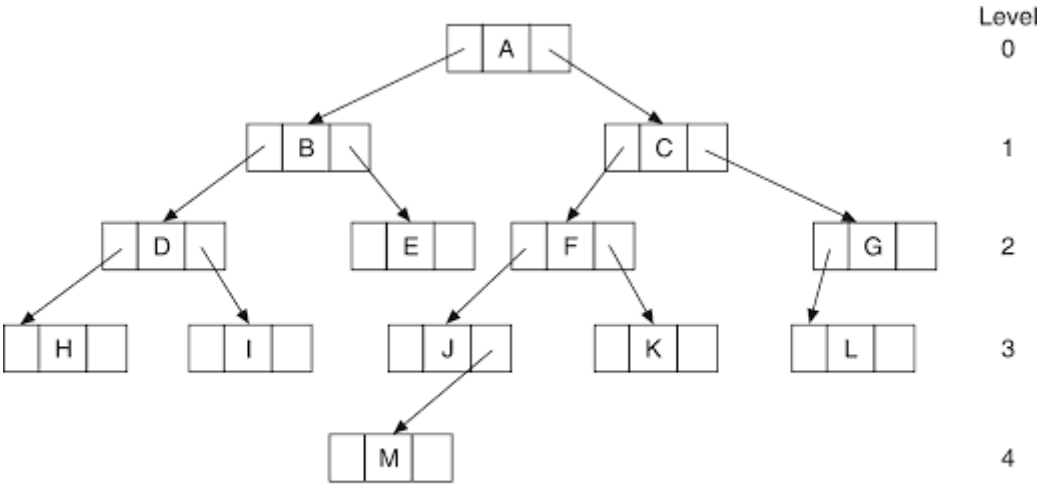
Child. If the immediate predecessor of a node is the parent of the node then all immediate successors of a node are known as *child*. For example, in Figure 7.4(b), Y and Z are the two child of X. The child which is on the left side is called the *left child* and that on the right side is called the *right child*.



(a) Structure of a node in a tree



(b) Parent, left child and right child of a node



(c) A simple tree with 13 nodes

Figure 7.4 A tree and its various components.

Link. This is a pointer to a node in a tree. For example, as shown in Figure 7.4(a), LC and RC are two *links* of a node. Note that there may be more than two links of a node.

Root. This is a specially designated node which has no parent. In Figure 7.4(c), *A* is the *root* node.

Leaf. The node which is at the end and does not have any child is called *leaf* node. In Figure 7.4(c), *H*, *I*, *K*, *L*, and *M* are the leaf nodes. A leaf node is also alternatively termed terminal node.

Level. *Level* is the rank in the hierarchy. The root node has level 0. If a node is at level l, then its child is at level $l + 1$ and the parent is at level $l - 1$. This is true for all nodes except the root node, being at level zero. In Figure 7.4(c), the levels of various nodes are depicted.

Height. The maximum number of nodes that is possible in a path starting from the root node to a leaf node is called the *height* of a tree. For example, in Figure 7.4(c), the longest path is *A–C–F–J–M* and hence the height of this tree is 5. It can be easily seen that $h = l_{max} + 1$, where h is the height and l_{max} is the maximum level of the tree.

Degree. The maximum number of children that is possible for a node is known as the *degree* of a node. For example, the degree of each node of the tree as shown in Figure 7.4(c) is 2.

Sibling. The nodes which have the same parent are called *siblings*. For example, in Figure 7.4(c), J and K are siblings.

Different texts use different terms for the above defined terms, such as *depth* for height, *branch* or *edge* for link, *arity* for degree, *external* node for leaf node and *internal* node for a node other than a leaf node.

DEFINITION AND CONCEPTS

Let us define a tree. A *tree* is a finite set of one or more nodes such that:

(i) there is a specially designated node called the root,

(ii) the remaining nodes are partitioned into n ($n > 0$) disjoint sets $T_1, T_2, ..., T_n$, where each T_i ($i = 1, 2, ..., n$) is a tree; $T_1, T_2, ..., T_n$ are called subtrees of the root.

To illustrate the above definition, let us consider the sample tree shown in Figure 7.6.

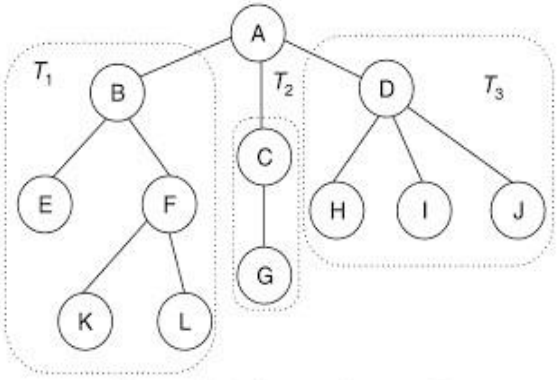


Figure 7.6 A sample tree T.

7.2.1 Binary Trees

A *binary tree* is a special form of a tree. Contrary to a general tree, a binary tree is more important and frequently used in various applications of computer science. Like a general tree, a binary tree can also be defined as a finite set of nodes, such that:

(i) T is empty (called the empty binary tree), or

(ii) T contains a specially designated node called the root of T, and the remaining nodes of T form two disjoint binary trees T_1 and T_2 which are called the left sub-tree and the right sub-tree, respectively.

- Any node N in a binary tree has either 0,1 or 2 successors.
- A tree can never be empty, but binary tree may be empty.
- A tree can have any no. of children, but in a binary tree, a node can have at most two children.

Figure 7.7 depicts a sample binary tree.

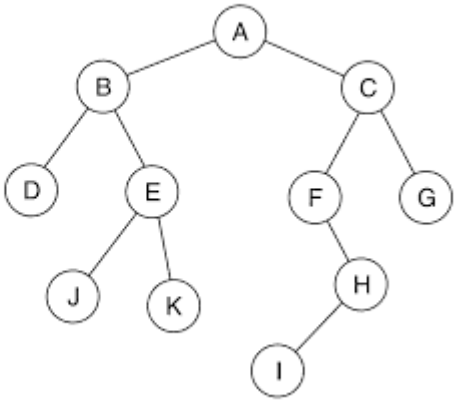


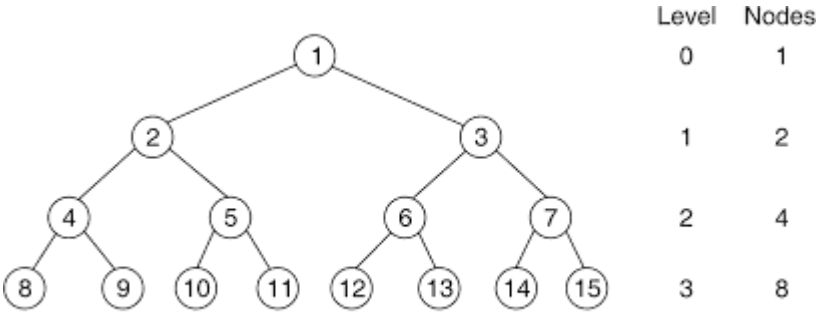
Figure 7.7 A sample binary tree with 11 nodes.

One can easily notice the main difference between the definitions of a tree and a binary tree. A tree can never be empty but a binary tree may be empty. Another difference is that in the case of a binary tree a node may have at most two children (that is, a tree having degree = 2), whereas in the case of a tree, a node may have any number of children.

Two special situations of a binary tree are possible: full binary tree and complete binary tree.

Full binary tree

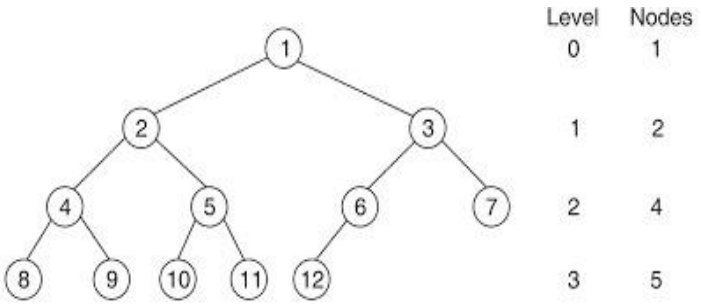
A binary tree is a *full binary* tree if it contains the maximum possible number of nodes at all levels. Figure 7.8(a) shows such a tree with height 4.



(a) A full binary tree of height 4

Complete binary tree

A binary tree is said to be a *complete binary* tree if all its levels, except possibly the last level, have the maximum number of possible nodes, and all the nodes in the last level appear as far left as possible. Figure 7.8(b) depicts a complete binary tree.

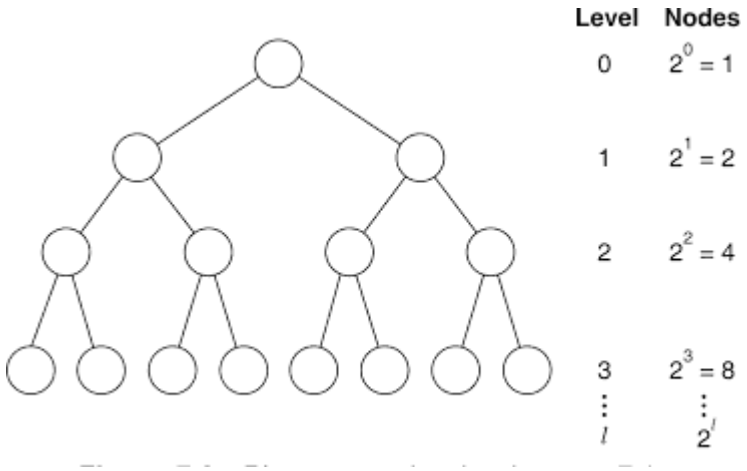


(b) A complete binary tree of height 14

Figure 7.8 Two special cases of binary trees.

Observe that the binary tree represented in Figure 7.7 is neither a full binary tree nor a complete binary tree.

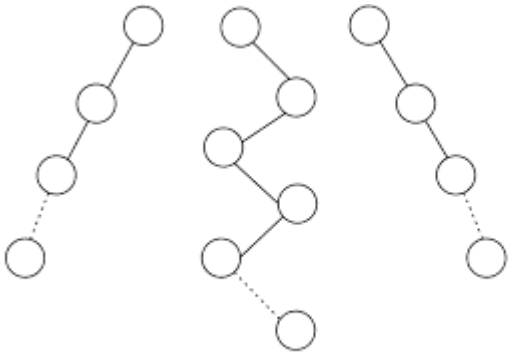
Lemma 7.1
In any binary tree, the maximum number of nodes on level l is 2^l, where $l \geq 0$.



Properties of a binary tree

Lemma 7.2
The maximum number of nodes possible in a binary tree of height h is $2^h - 1$.

Lemma 7.3
The minimum number of nodes possible in a binary tree of height h is h.



Skew binary trees containing the minimum number of nodes.

Lemma 7.4

For any non-empty binary tree, if n is the number of nodes and e is the number of edges, then $n = e + 1$.

REPRESENTATION OF BINARY TREE

Implicit & Explicit representation

- **Implicit representation**
  - **Sequential / Linear** representation, using arrays.
- **Explicit representation**
  - **Linked list representation,** using pointers**.**

Sequential representation

- This representation is static.
- Block of memory for an array is allocated, before storing the actual tree.
- Once the memory is allocated, the size of the tree will be fixed.
- Nodes are stored level by level, starting from the zeroth level.
- Root node is stored in the starting memory location, as the first element of the array.
- Consider a linear array TREE

Rules for storing elements in TREE are:

1. The root R of T is stored in location 1.
2. For any node with index I 1<i<=n:

PARENT(i)= i/2

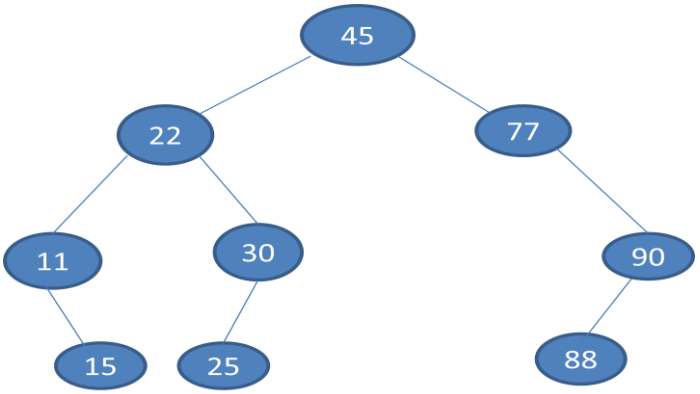
For the node when i=1,there is no parent.

LCHILD(i)=2*I

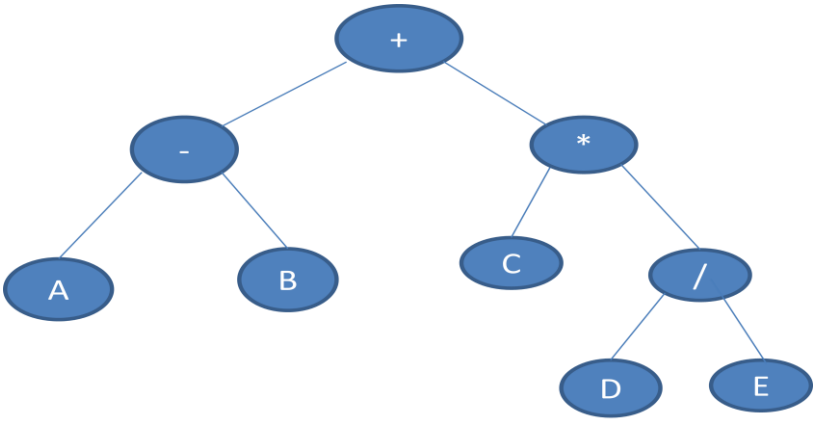
If 2* i >n, then i has no left child

RCHILD(i)=2*i+1

If 2*i+1>n, then I has no right child.



1	2	3	4	5	6	7	8	9	10	11	12	13	14
45	22	77	11	30		90		15	25				88



1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
+	-	*	A	B	C	/							D	E

Sequential representation- Advantages:

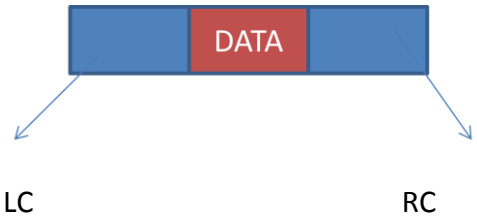
1. Any node can be accessed from any other node by calculating the index.
2. Here, data are stored simply without any pointers to their successor or predecessor.
3. Programming languages, where dynamic memory allocation is not possible(like BASIC, FROTRAN), array representation is only possible.

Sequential representation- Disadvantages:

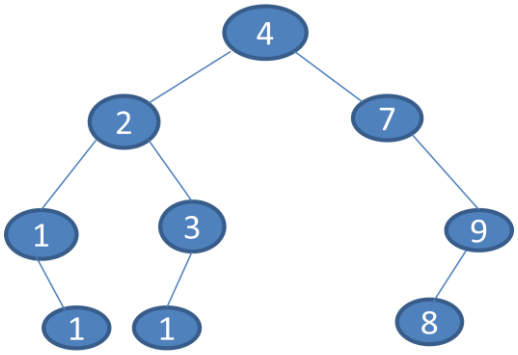
1. Other than full binary trees, majority of the array entries may be empty.
2. It allows only static representation. It is not possible to enhance the tree structure, if the array structure is limited.
3. Inserting a new node and deletion of an existing node is difficult, because it require considerable data movement

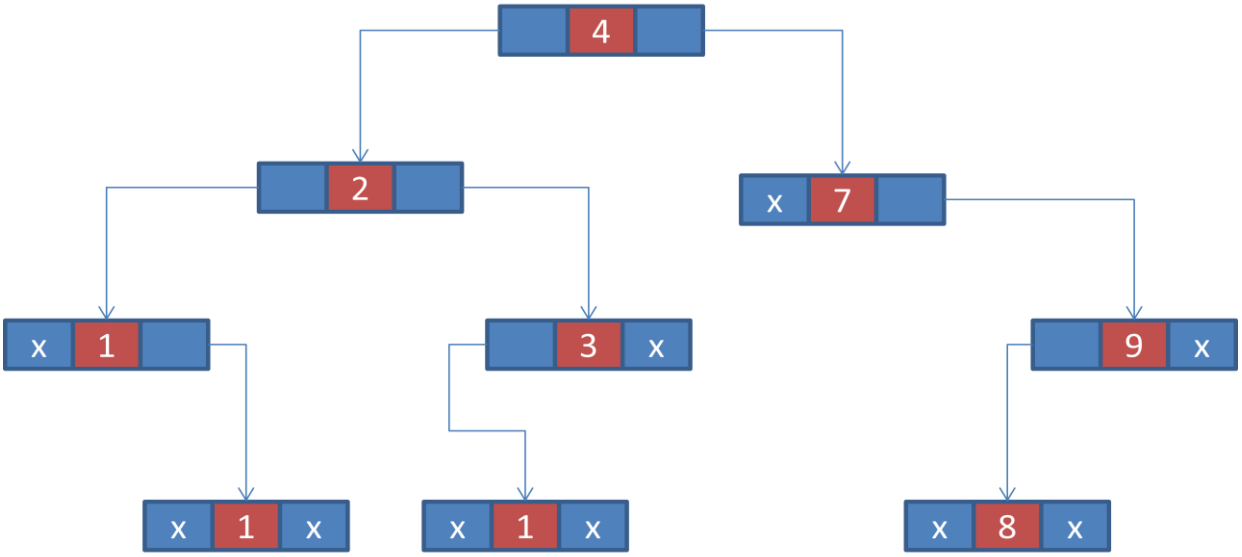
Linked list representation

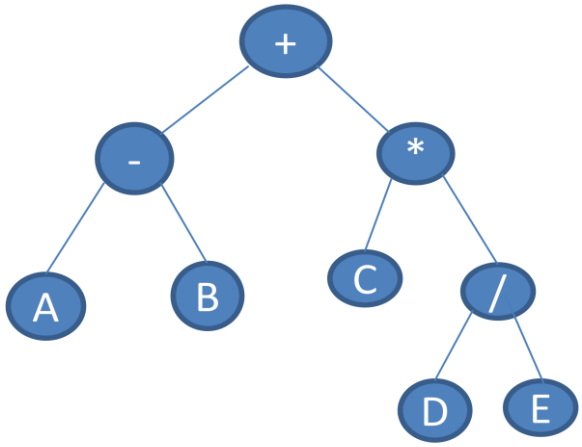
- It consist of three parallel arrays DATA, LC and RC

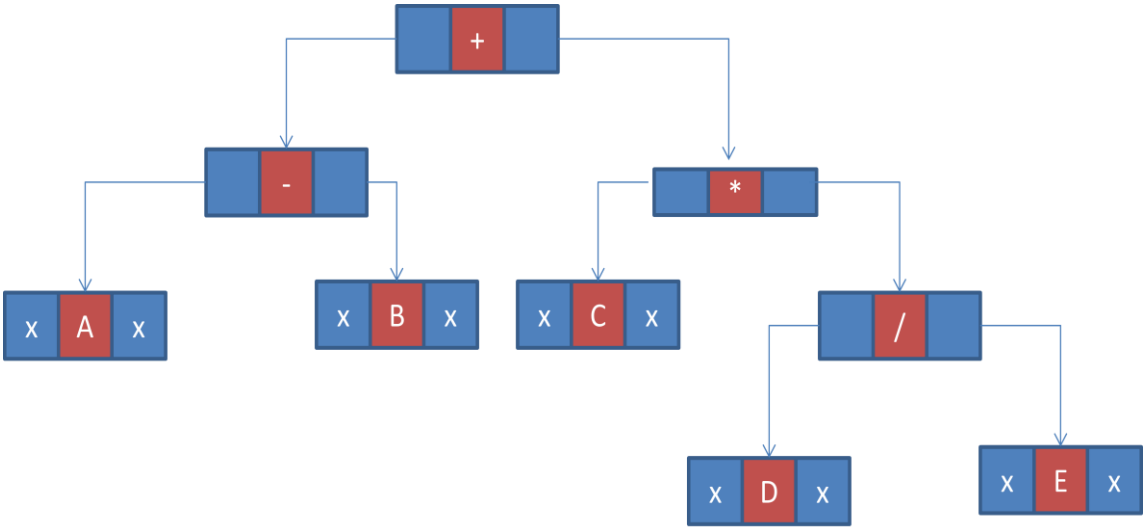


- Each node N of T will correspond to a location K such that:
  - DATA[K] contains the data at the node N
  - LC[K] contains the location of the left child of node N
  - RC[K] contains the location of the right child of node N



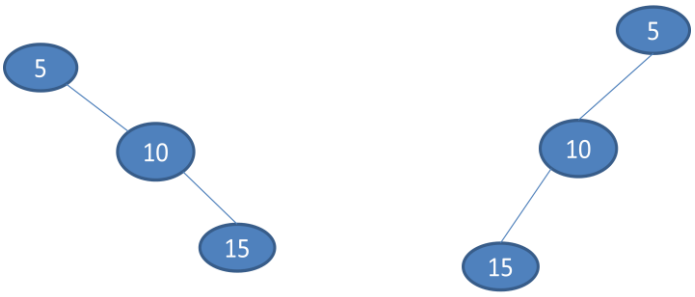

4
2
x 7
x 1
3 x
9 x
x 1 x
x 1 x
x 8 x





Skew binary tree

- Consider a binary tree with n nodes.
- If the maximum height possible h_{max}=n, then it is called skew binary tree.



BINARY TREE TRAVERSALS

- Traversal is a process to visit all the nodes of a tree and may print their values too.
- Because, all nodes are connected via edges (links) we always start from the root node.
- That is, we cannot random access a node in tree.
- There are three ways which we use to traverse a tree –
  1. **Preorder traversal (R, T_l, T_r)**
  2. **Inorder traversal (T_l, R, T_r)**
  3. **Postorder traversal (T_l, T_r, R)**

Preorder traversal

In this traversal, the root is visited first, then the left sub-tree in preorder fashion, and then the right sub-tree in preorder fashion. Such a traversal can be defined as follows:

- Visit the root node R.
- Traverse the left sub-tree of R in preorder.
- Traverse the right sub-tree of R in preorder.

Inorder traversal

With this traversal, before visiting the root node, the left sub-tree of the root node is visited, then the root node and after the visit of the root node the right sub-tree of the root node is visited. Visiting both the sub-trees is in the same fashion as the tree itself. Such a traversal can be stated as follows:

- Traverse the left sub-tree of the root node R in inorder.
- Visit the root node R.
- Traverse the right sub-tree of the root node R in inorder.

Postorder traversal

Here, the root node is visited in the end, that is, first visit the left sub-tree, then the right sub-tree, and lastly the root. A definition for this type of tree traversal is stated below:

- Traverse the left sub-tree of the root R in postorder
- Traverse the right sub-tree of the root R in postorder
- Visit the root node R.

Inorder traversal of a binary tree

Recall that inorder traversal of a binary tree follows three ordered steps as follows:

- Traverse the left sub-tree of the root node R in inorder.
- Visit the root node R.
- Traverse the right sub-tree of the root node R in inorder.

Out of these steps, steps 1 and 3 are defined recursively. The following is the algorithm *Inorder* to implement the above definition.

Algorithm Inorder

Input: ROOT is the pointer to the root node of the binary tree.
Output: Visiting all the nodes in inorder fashion.
Data structure: Linked structure of binary tree.

Steps:

```
1.  ptr = ROOT                                  // Start from ROOT
2.  If (ptr ≠ NULL) then                        // If it is not an empty node
3.      Inorder(ptr→LC)          // Traverse the left sub-tree of the node in inorder
4.      Visit(ptr)                                          // Visit the node
5.      Inorder (ptr→RC)        // Traverse the right sub-tree of the node in inorder
6.  EndIf
7.  Stop
```

Preorder traversal of a binary tree

The definition of preorder traversal of a binary tree, as already discussed, is presented again as follows:

- Visit the root node R.
- Traverse the left sub-tree of the root node R in preorder.
- Traverse the right sub-tree of the root node R in preorder.

The algorithm *Preorder* to implement the above definition is presented below:

Algorithm Preorder

Input: ROOT is the pointer to the root node of the binary tree.
Output: Visiting all the nodes in preorder fashion.
Data structure: Linked structure of binary tree.

Steps:

```
1.  ptr = ROOT                                  // Start from the ROOT
2.  If (ptr ≠ NULL) then                        // If it is not an empty node
3.      Visit(ptr)                                          // Visit the node
4.      Preorder(ptr→LC)         // Traverse the left sub-tree of the node in preorder
5.      Preorder(ptr→RC)        // Traverse the right sub-tree of the node in preorder
6.  EndIf
7.  Stop
```

Postorder traversal of a binary tree

The definition of postorder traversal of a binary tree is repeated below:

- Traverse the left sub-tree of the root node R in postorder.
- Traverse the right sub-tree of the root node R in postorder.
- Visit the root node R.

The algorithm to implement the above is given below:

Algorithm Postorder

Input: ROOT is the pointer to the root node of the binary tree.
Output: Visiting all the nodes in preorder fashion.
Data structure: Linked structure of binary tree.

Steps:

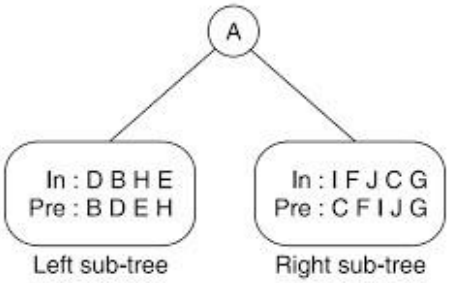
```
1. ptr = ROOT                                   // Start from the root
2. If (ptr ≠ NULL) then                         // If it is not an empty node
3.     Postorder(ptr→LC)                        // Traverse the left sub-tree of the node in inorder
4.     Postorder(ptr→RC)                        // Traverse the right sub-tree of the node in inorder
5.     Visit(ptr)                               // Visit the node
6. EndIf
7. Stop
```

Suppose the inorder and preorder traversals of a binary tree are as follows:

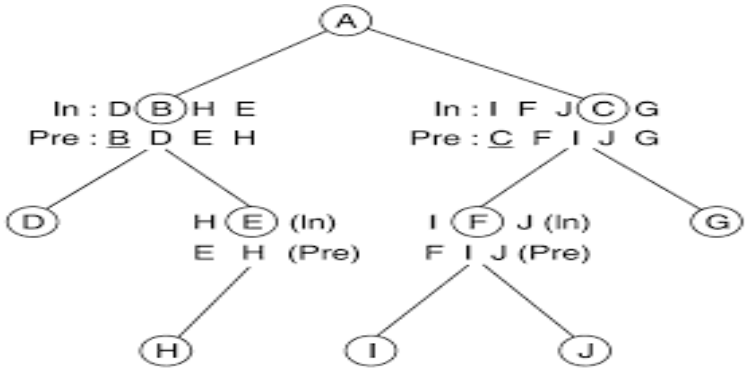
Inorder	*D*	*B*	*H*	*E*	*A*	*I*	*F*	*J*	*C*	*G*
Preorder	*A*	*B*	*D*	*E*	*H*	*C*	*F*	*I*	*J*	*G*

We have to construct the binary tree. The following steps need to be followed.

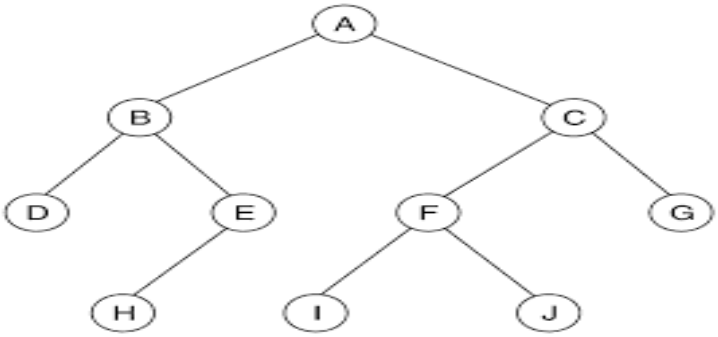
1. From the preorder traversal, it is evident that A is the root node.
2. In inorder traversal, all the nodes which are on the left side of A belong to the left sub-tree and those which are on right side of A belong to the right sub-tree.
3. Now the problem reduces to form sub-trees and the same procedure can be applied repeatedly. Figure 7.23 illustrates this procedure.



(a) Two sub-trees as A being the root from the two traversals



(b) Repeated application of sub-trees formation

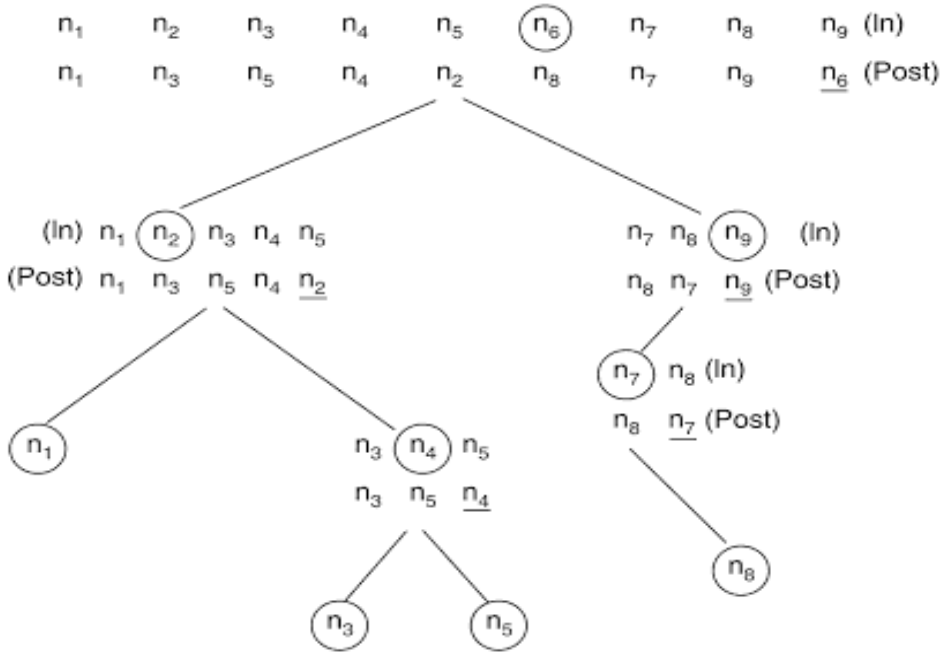


(c) Final binary tree from its Inorder and Preorder traversals

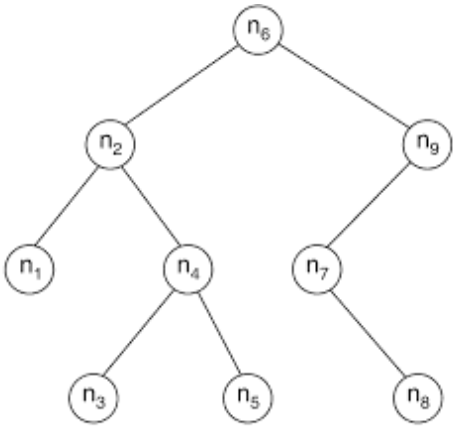
The following are the inorder and postorder traversals of a binary tree:

Inorder	n_1	n_2	n_3	n_4	n_5	n_6	n_7	n_8	n_9
Postorder	n_1	n_3	n_5	n_4	n_2	n_8	n_7	n_9	n_6

The construction of the binary tree is shown in Figure 7.24.



(a) Construction by sub-trees partition



(b) Final binary tree

Figure 7.24 Construction of a binary tree from its inorder and postorder traversals.

7.5.2 Binary Search Tree

A binary tree T is termed *binary search* tree (or *binary sorted* tree) if each node N of T satisfies the following property:

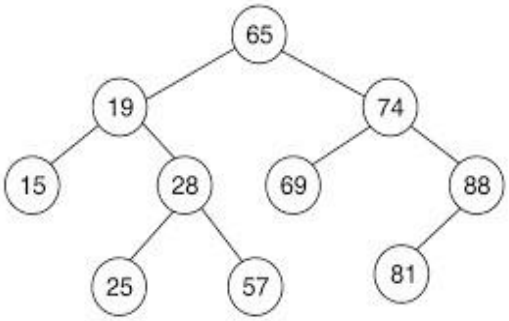
The value at N is greater than every value in the left sub-tree of N and is less than every value in the right sub-tree of N.

Figure 7.31 shows two binary search trees for two different types of data.

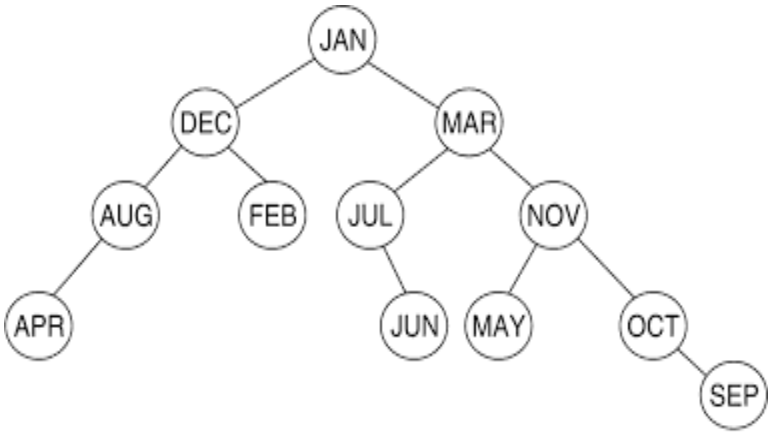
Observe that in Figure 7.31(b), the lexicographical ordering is taken among the data whereas in Figure 7.31(a), numerical ordering is taken.

Now, let us discuss the possible operations on any binary search tree. Operations which are generally encountered to manipulate this data structure are:

- Searching data
- Inserting data
- Deleting data
- Traversing the tree.



(a) A binary search tree with numeric data



(b) A binary search tree with alphabetic data

Searching a binary search tree

Searching data in a binary search tree is much faster than searching data in arrays or linked lists. This is why in the applications where frequent searching operations need to be performed, this data structure is used to store data. In this section, we will discuss how this operation can be defined.

Suppose in a binary search tree *T*, ITEM the item of search. We will assume that the tree is represented using a linked structure.

We start from the root node *R*. Then, if ITEM is less than the value in the root node *R*, we proceed to its left child; if ITEM is greater than the value in the node *R*, we proceed to its right child. The process will be continued till the ITEM is not found or we reach a dead end, that is, the leaf node. Figure 7.32 shows the track (in shaded line) for searching of 54 in a binary search tree.

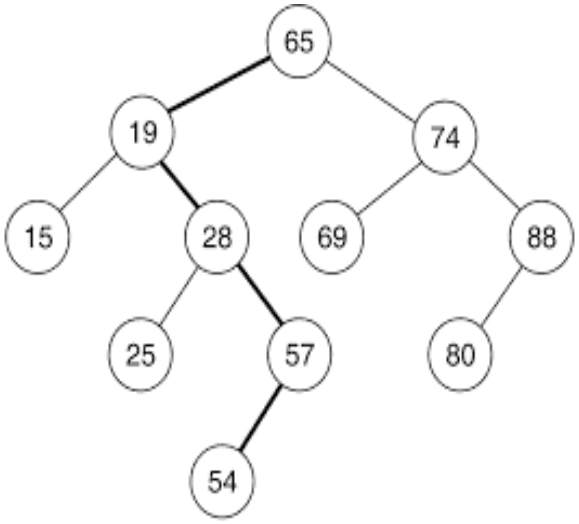


Figure 7.32 Searching 54 in a binary search tree.

Algorithm Search_BST

Input: ITEM is the data that has to be searched.
Output: If found then pointer to the node containing data ITEM else a message.
Data structure: Linked structure of the binary tree. Pointer to the root node is ROOT.

Steps:

```
1.  ptr = ROOT, flag = FALSE                                  // Start from the root
2.  While (ptr ≠ NULL) and (flag = FALSE) do
3.      Case: ITEM < ptr→DATA                                 // Go to the left sub-tree
4.          ptr = ptr→LCHILD
5.      Case: ptr→DATA = ITEM                                 // Search is successful
6.          flag = TRUE
7.      Case: ITEM > ptr→DATA                                 // Go to the right sub-tree
8.          ptr = ptr→RCHILD
9.      EndCase
10. EndWhile
11. If (flag = TRUE) then                                     // Search is successful
12.     Print "ITEM has found at the node", ptr
13. Else
14.     Print "ITEM does not exist: Search is unsuccessful"
15. EndIf
16. Stop
```

Inserting a node into a binary search tree

The insertion operation on a binary search tree is conceptually very simple. It is, in fact, one step more than the searching operation. To insert a node with data, say ITEM, into a tree, the

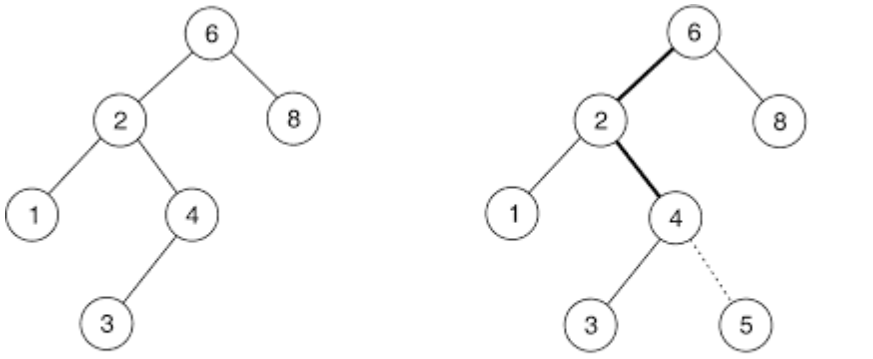
tree is required to be searched starting from the root node. If ITEM is found, do nothing, otherwise ITEM is to be inserted at the dead end where the search halts. Figure 7.33 shows the insertion of 5 into a binary tree. Here, search proceeds starting from the root node as 6–2–4 then halts when it finds that the right child is null (dead end). This simply means that if 5 occurs, then it should have occurred on the right part of the node 4. So, 5 should be inserted as the right child of 4.

Algorithm Insert_BST

Input: ITEM is the data component of a node that has to be inserted.
Output: If there is no node having data ITEM, it is inserted into the tree else a message.
Data structure: Linked structure of binary tree. Pointer to the root node is ROOT.

1. Start
2. Create a node temp and insert ITEM in it.
3. If(ROOT==null)
4. Set ROOT=temp
5. Else

6. Set ptr= ROOT
7. while(ptr≠ null)
8. Set parent= ptr
9. if(ITEM< ptr->data)
10. ptr=ptr->LCHILD
11. if(ptr==null)
12. parent->LCHILD=temp
13. else
14. ptr= ptr->RCHILD
15. if (ptr==null)
16. parent->RCHILD= temp



(a) Before insertion

(b) Search finds the location where 5 should be inserted

Figure 7.33 Inserting 5 into a binary search tree.

Deletion in a BST

- There are the following possible cases when we delete a node:

1. The node to be deleted has **no children**. In this case, all we need to do is delete the node.

2. The node to be deleted has **only one child** (left or right subtree). We delete the node and attach the subtree to the deleted node's parent.

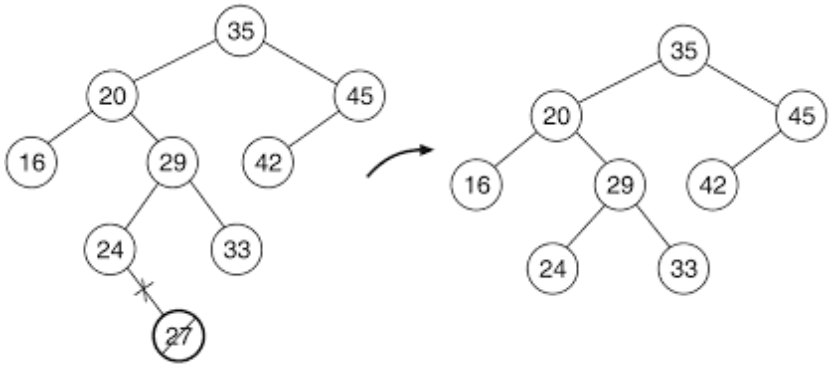
3. The node to be deleted has **two children**. It is possible to delete a node from the middle of a tree, but the result tends to create very unbalanced trees.

Deletion from the middle of a tree

- We can find the largest node in the deleted node's left subtree and move its data to replace the deleted node's data.
- We can find the smallest node on the deleted node's right subtree and move its data to replace the deleted node's data.
- Either of these moves preserves the integrity of the binary search tree.

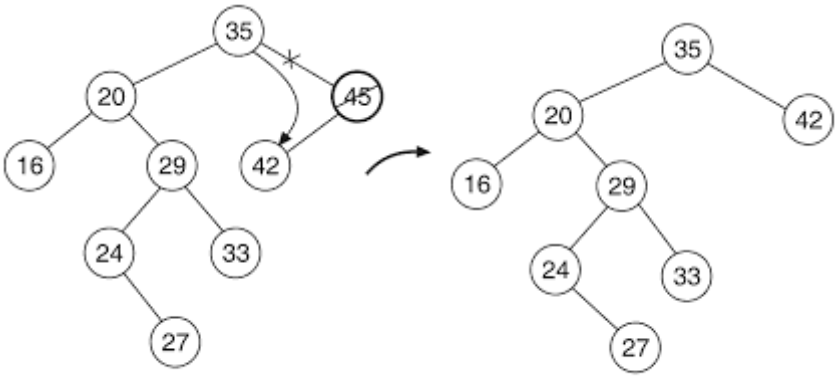
Deletion in a BST: Example

Case 1: The node to be deleted has no children.



1. Deletion of the node 27

Case 2: The node to be deleted has exactly one child.

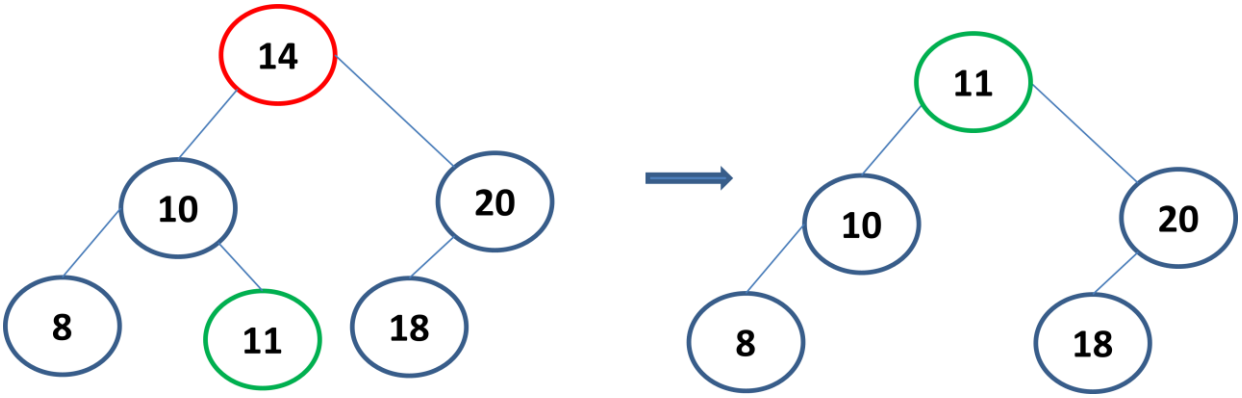


2. Deletion of the node 45

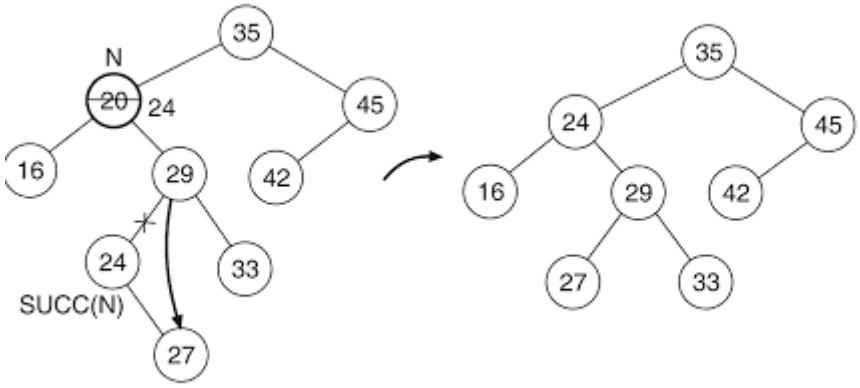
Case 3: The node to be deleted has two children.

Two methods:

1) We can find the largest node in the deleted node's left subtree and move its data to replace the deleted node's data.



2) We can find the smallest node on the deleted node's right subtree and move its data to replace the deleted node's data.



(c) Deletion of the node 20

Deletion Algorithm

Delete(item, ptr)

1. if ptr != null then do step 2 – 7

2. if item < ptr -> data then

Delete(item,ptr->lchild)

3. else if item > ptr -> data

Delete(item, ptr -> rchild)

4. else if (ptr -> lchild = null) and (ptr -> rchild = null)

ptr = null // Deleting leaf node

5. else if (ptr -> lchild = null) then ptr = ptr -> rchild // Single child

6. else if (ptr -> rchild = null) then ptr = ptr -> lchild // Single child

7. else set ptr -> data = deletemin(ptr -> rchild) // Deleting if more children are present

Function deletemin(ptr)

1. if ptr -> lchild = null then return ptr ->item

2. else return deletemin(ptr -> lchild)

GRAPHS

- Graph is an important non-linear data structure.

- Tree is in fact, a special kind of graph structure.
- In tree structure, there is a hierarchical relationship between parent and children, that is, one parent and many children.
- On the other hand, in graph, relationship is less restricted. Here, relationship is from many parents to many children.
- The below figure represents the two non-linear data structures.

Figure:

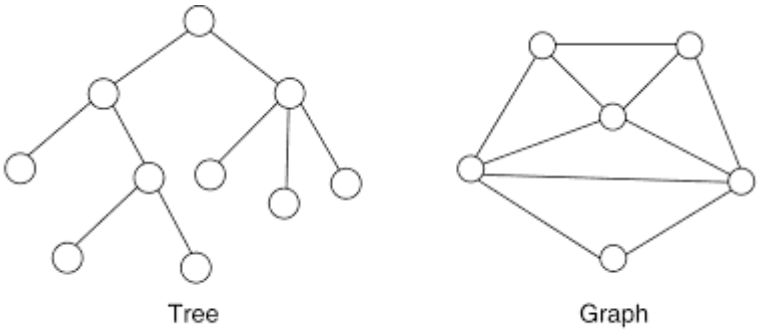


Figure Two non-linear data structures: tree and graph.

Formal definition of graph

- A graph can be represented as **G=(V,E)**

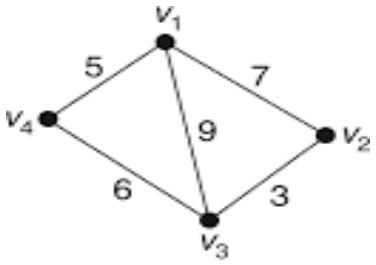
Graph. A *graph* G consists of two sets:

(i) A set V, called the set of all vertices (or nodes)
(ii) A set E, called the set of all edges (or arcs). This set E is the set of all pairs of elements from V.

For example, let us consider the graph $G1$ in Figure . Here

$$V = \{v_1, v_2, v_3, v_4\}$$
$$E = \{(v_1, v_2), (v_1, v_3), (v_1, v_4), (v_2, v_3), (v_3, v_4)\}$$

-

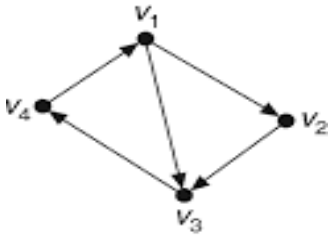


G3

Graph Terminologies

Digraph. A *digraph* is also called a *directed* graph. It is a graph G, such that, $G = <V, E>$, where V is the set of all vertices and E is the set of ordered pairs of elements from V. For example, graph $G2$ is a digraph, where

$$V = \{v_1, v_2, v_3, v_4\}$$
$$E = \{(v_1, v_2), (v_1, v_3), (v_2, v_3), (v_3, v_4), (v_4, v_1)\}$$

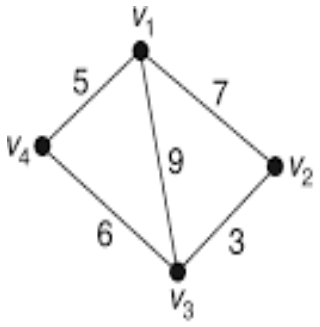


G2

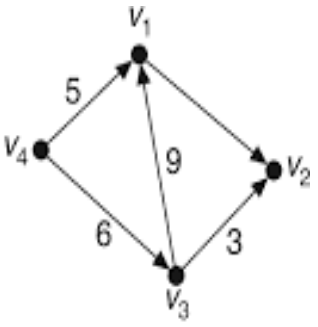
Here, if an order pair (v_i, v_j) is in E then there is an edge directed from v_i to v_j (indicated by the arrowhead).

Note that, in the case of an undirected graph (simple graph), the pair (v_i, v_j) is unordered, that is, (v_i, v_j) and (v_j, v_i) are the same edges, but in case of digraph they correspond to two different edges.

Weighted graph. A graph (or digraph) is termed *weighted* graph if all the edges in it are labelled with some weights. For example, $G3$ and $G4$ are two weighted graphs in Figure 8.3.

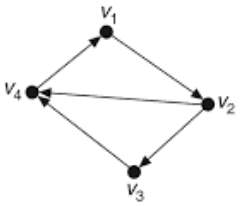


G3

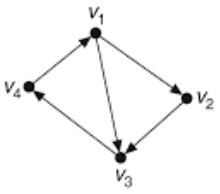


G4

Adjacent vertices. A vertex v_i is *adjacent* to (or neighbour of) another vertex say, v_j, if there is an edge from v_i to v_j. For example, in $G11$ (Figure 8.3), v_2 is adjacent to v_3 and v_4, v_1 is not adjacent to v_4 but to v_2. Similarly the neighbours of v_3 in graph $G2$ are v_1 and v_2 (but not v_4).

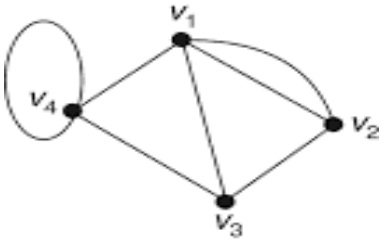


G11



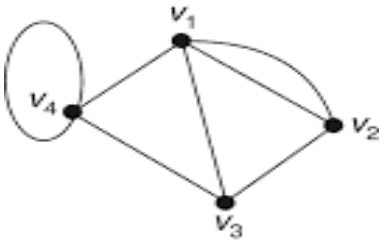
G2

Self loop. If there is an edge whose starting and end vertices are same, that is, (v_i, v_i) is an edge, then it is called a *self loop* (or simply, a loop). For example, the graph $G5$ (in Figure 8.3) has a self loop at vertex v_4.



G5

Parallel edges. If there is more than one edge between the same pair of vertices, then they are known as the *parallel* edges. For example, there are two parallel edges between v_1 and v_2 in graph $G5$ of Figure 8.3. A graph which has either self loop or parallel edges or both, is called *multigraph*. In Figure 8.3, $G5$ is thus a multigraph.



G5

Simple graph (digraph). A graph (digraph) if it does not have any self loop or parallel edges is called a *simple graph* (digraph).

The following graphs G2,G6 and G9 are examples of simple graph since it does not contain any self-loop or parallel edges.

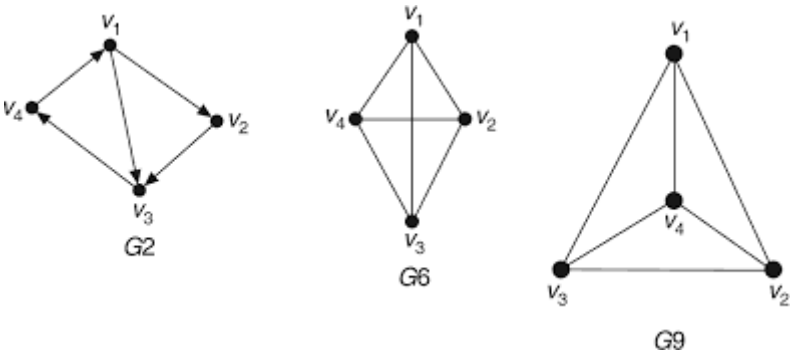


Figure: Examples of simple graphs

The below graphs G5 and G10 are not simple graphs. Here, the graph G5 contains both self loop and parallel edges, where as graph G10 contains parallel edges.

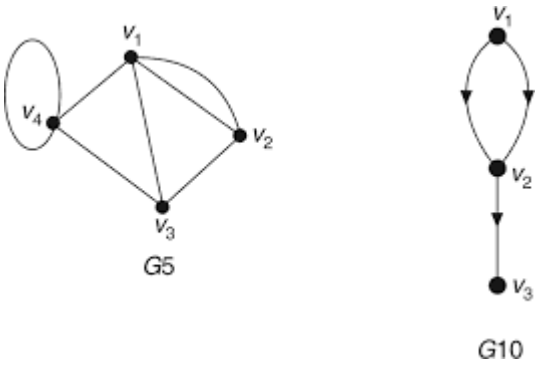
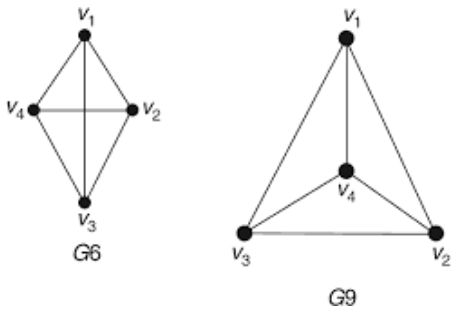


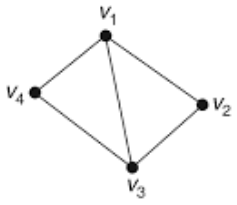
Figure: Examples of graphs which are not simple

Complete graph. A graph (digraph) G is said to be *complete* if each vertex v_i is adjacent to every other vertex v_j in G. In other words, there are edges from any vertex to all other vertices. For examples, $G6$ and $G9$ are two complete graphs.

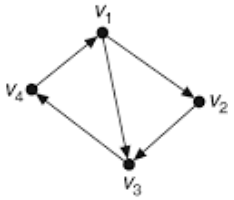


Both G1 and G2 contain cycle.

Acyclic graph. If there is a path containing one or more edges which starts from a vertex v_i and terminates into the same vertex then the path is known as a *cycle*. For example, there is a cycle in both $G1$ and $G2$ (Figure 8.3). If a graph (digraph) does not have any cycle then it is called *acyclic* graph. For example, in Figure 8.3, $G4$, $G7$ are two acyclic graphs.

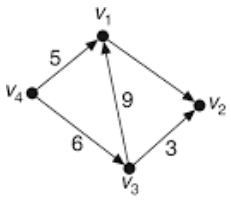


G1

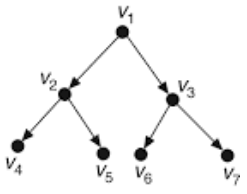


G2

G4 and G7 are two acyclic graphs.

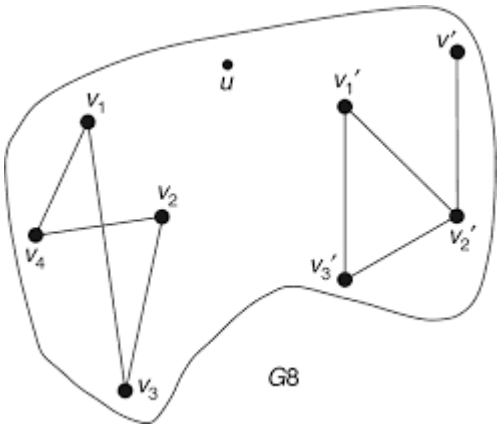


G4

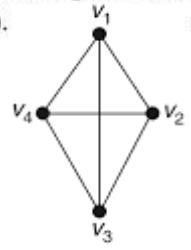


G7

Isolated vertex. A vertex is *isolated* if there is no edge connected from any other vertex to the vertex. For example, in $G8$ (Figure 8.3) the vertex u is an isolated vertex.

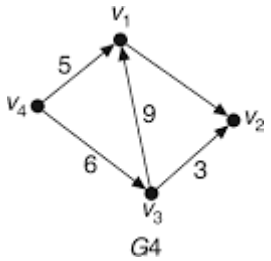


Degree of vertex. The number of edges connected with vertex v_i is called the *degree* of vertex v_i and is denoted by *degree*(v_i). degree(v_i) = 3, $\forall$ $v_i \in G6$ (Figure 8.3).

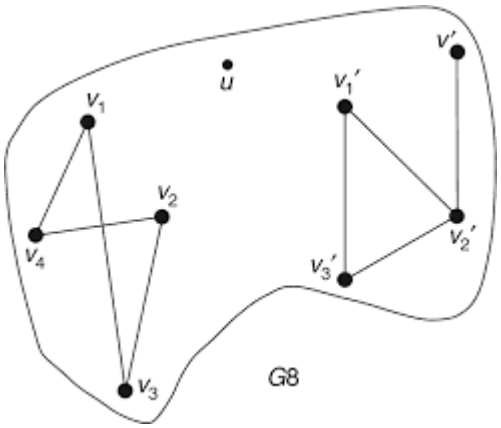


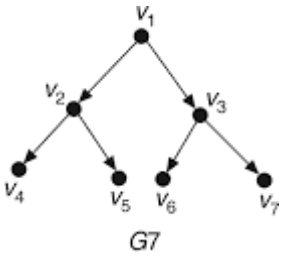
But for a digraph, there are two degrees: *indegree* and *outdegree*. Indegree of v_i denoted as indegree(v_i) = number of edges incident into v_i. Similarly, outdegree(v_i) = number of edges emanating from v_i. For example, let us consider the digraph $G4$. Here:

indegree $(v_1) = 2$,	outdegree $(v_1) = 1$
indegree $(v_2) = 2$	outdegree $(v_2) = 0$
indegree $(v_3) = 1$	outdegree $(v_3) = 2$
indegree $(v_4) = 0$	outdegree $(v_4) = 2$



Pendant vertex. A vertex v_i is *pendant* if its indegree(v_i) = 1 and outdegree(v_i) = 0. For example, in $G8$ v' is a pendant vertex. In $G7$, there are four pendant vertices v_4, v_5, v_6 and v_7.

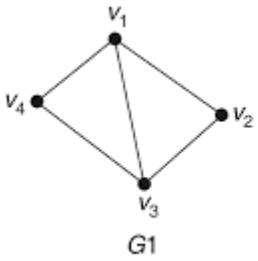




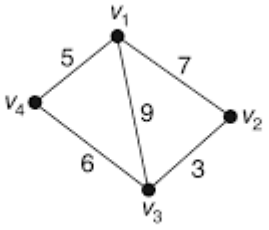
Connected graph. In a graph (not digraph) G, two vertices v_i and v_j are said to be *connected* if there is a path in G from v_i to v_j (or v_j to v_i). A graph is said to be connected if for every pair of distinct vertices v_i, v_j in G, there is a path. For example, $G1$, $G3$ and $G6$ are connected graphs but $G8$ is not (Figure 8.3).

Examples for connected graphs

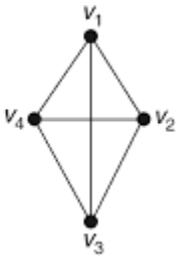
G8 – Not connected



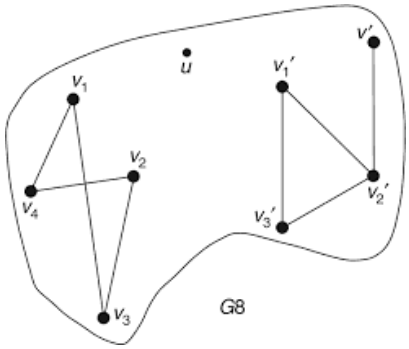
G1



G3



G6



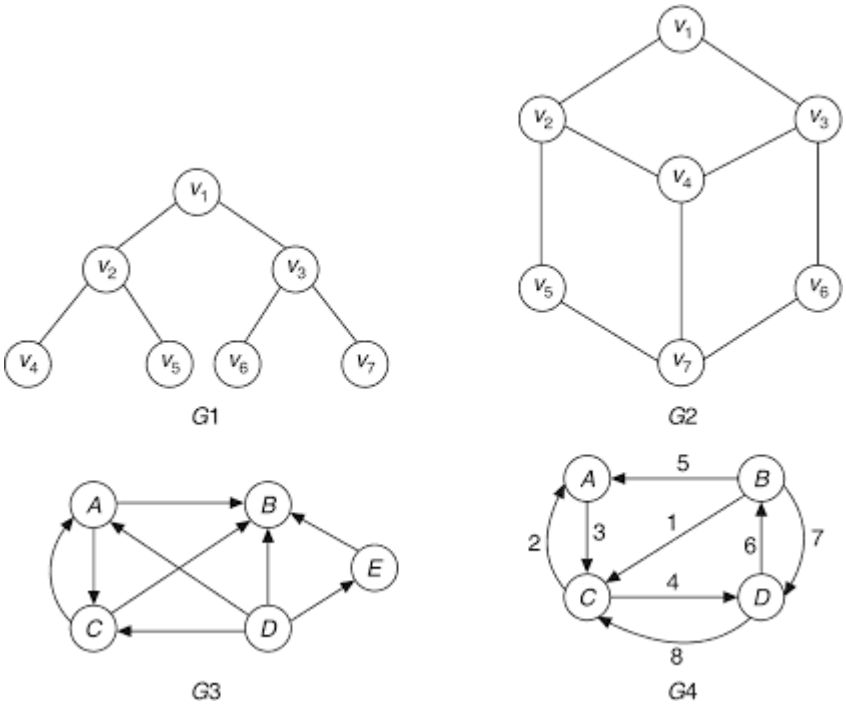
G8

Representation of Graphs

A graph can be represented in many ways. Some of the representations are:

1. Set representation
2. Linked representation
3. Sequential (matrix) representation

Consider the following graphs to be illustrated using the above representations.



1. Set Representation

This is one of the straightforward methods of representing a graph. With this method, two sets are maintained: (i) V, the set of vertices, (ii) E, the set of edges, which is the subset of $V \times V$. But if the graph is weighted, the set E is the ordered collection of three tuples, that is, $E = W \times V \times V$, where W is the set of weights.

Let us see, how all the graphs given in Figure 8.5 can be represented with this technique.

Graph G1

$V(G1) = \{v_1, v_2, v_3, v_4, v_5, v_6, v_7\}$

$E(G1) = \{(v_1, v_2), (v_1, v_3), (v_2, v_4), (v_2, v_5), (v_3, v_6), (v_3, v_7)\}$

Graph G2

$V(G2) = \{v_1, v_2, v_3, v_4, v_5, v_6, v_7\}$

$E(G2) = \{(v_1, v_2), (v_1, v_3), (v_2, v_4), (v_2, v_5), (v_3, v_4), (v_3, v_6), (v_4, v_7), (v_5, v_7), (v_6, v_7)\}$

Graph G3

$V(G3) = \{A, B, C, D, E\}$

$E(G3) = \{(A, B), (A, C), (C, B), (C, A), (D, A), (D, B), (D, C), (D, E), (E, B)\}$

Graph G4

$V(G4) = \{A, B, C, D\}$

$E(G4) = \{(3, A, C), (5, B, A), (1, B, C), (7, B, D), (2, C, A), (4, C, D), (6, D, B), (8, D, C)\}$

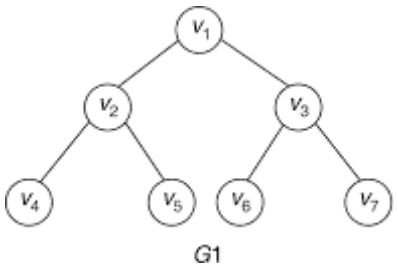
Note that, if the graph is a multigraph and undirected, this method does not allow to store parallel edges, as in a set, two identical elements cannot exist. Although it is a straightforward representation and the most efficient one from the memory point of view, this method of representation is not useful so far as the manipulation of graph is concerned.

Linked Representation

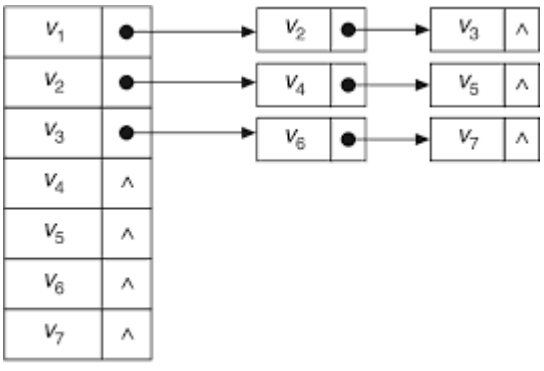
Linked representation is another space-saving way of graph representation. In this representation, two types of node structures are assumed as shown in Figure 8.6.

Node structure for non-weighted graph Node structure for weighted graph

Figure 8.6 Node structures in linked representation.



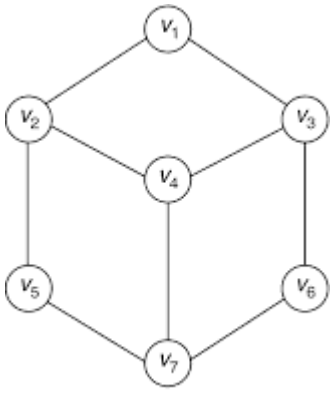
G1



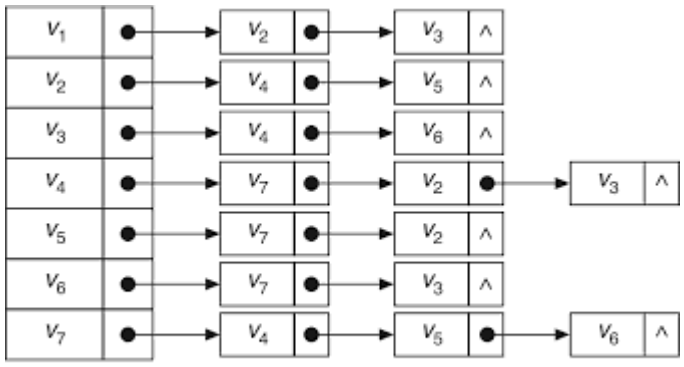
(a) Representation of graph *G1*

The linked list representation of graph G1 is as shown below.

The linked list representation of graph G2 is as shown below.

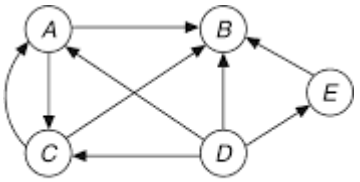


G2

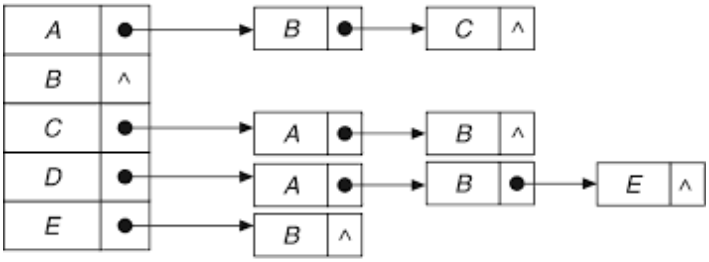


(b) Representation of graph *G2*

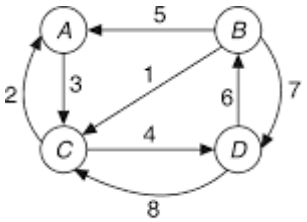
The linked representation of graph G3 is as shown below.



G3



(c) Representation of graph G3



G4

8.3.3 Matrix Representation

Matrix representation is the most useful way of representing any graph. This representation uses a square matrix of order $n \times n$, n being the number of vertices in the graph. A general representation is shown in Figure 8.8.

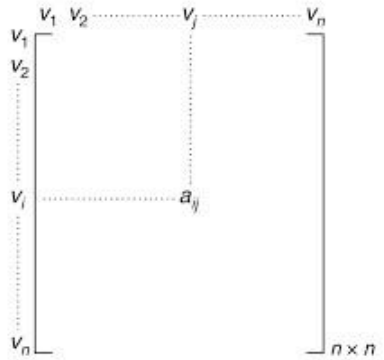
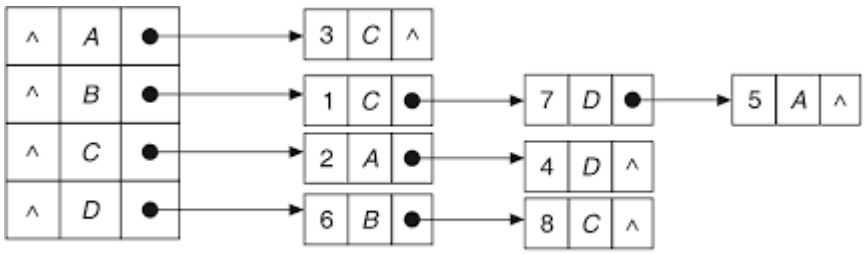


Figure 8.8 Matrix representation of graph.



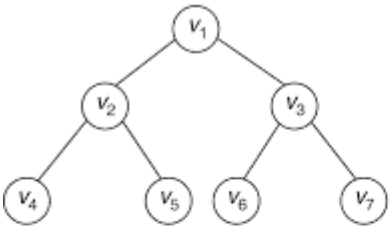
(d) Representation weighted digraph *G*4

Entries in the matrix can be decided as follows:

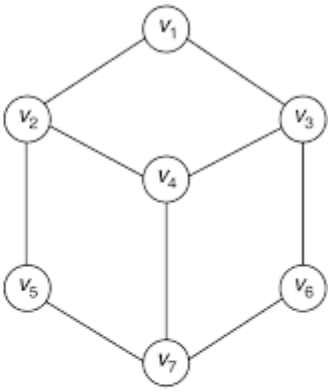
$$a_{ij} = 1, \text{ if there is an edge from } v_i \text{ to } v_j$$
$$= 0, \text{ otherwise}$$

This matrix is known as *adjacency* matrix because an entry stores the information whether two vertices are adjacent or not. Also, the matrix is alternatively termed *bit* matrix or *Boolean* matrix as the entries are either a 0 or a 1.

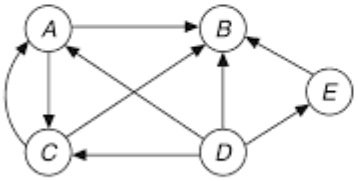
The adjacency matrix is useful to store multigraph as well as weighted graph. In case of multigraph representation, instead of entry 1, the entry will be the number of edges between two vertices. And in case of weighted graph, the entries are the weights of the edges between the vertices instead of 0 or 1. Figure 8.9 shows the adjacency matrix representation of graphs *G*1, *G*2, *G*3 and *G*4 given in Figure 8.5.



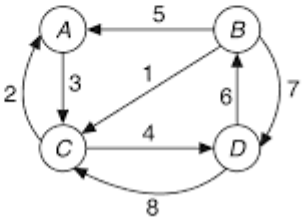
G1



G2



G3



G4

	v_1	v_2	v_3	v_4	v_5	v_6	v_7
v_1	0	1	1	0	0	0	0
v_2	1	0	0	1	1	0	0
v_3	1	0	0	0	0	1	1
v_4	0	1	0	0	0	0	0
v_5	0	1	0	0	0	0	0
v_6	0	0	1	0	0	0	0
v_7	0	0	1	0	0	0	0

G1

	v_1	v_2	v_3	v_4	v_5	v_6	v_7
v_1	0	1	1	0	0	0	0
v_2	1	0	0	1	1	0	0
v_3	1	0	0	1	0	1	0
v_4	0	1	1	0	0	0	1
v_5	0	1	0	0	0	0	1
v_6	0	0	1	0	0	0	1
v_7	0	0	0	1	1	1	0

G2

	A	B	C	D	E
A	0	1	1	0	0
B	0	0	0	0	0
C	1	1	0	0	0
D	1	1	1	0	1
E	0	1	0	0	0

G3

	A	B	C	D
A	0	0	3	0
B	5	0	1	7
C	2	0	0	4
D	0	6	8	0

G4

Figure 8.9 Adjacency matrix representation of graphs given in Figure 8.5.

GRAPH TRAVERSAL

There are 2 types of traversals

1. Breadth First search

2. Depth First Search

BREADTH-FIRST SEARCH

Here the data structure used is QUEUE

Algorithm

1. Initialize all the nodes as unvisited

2. Insert the first node/ starting node Vi to queue

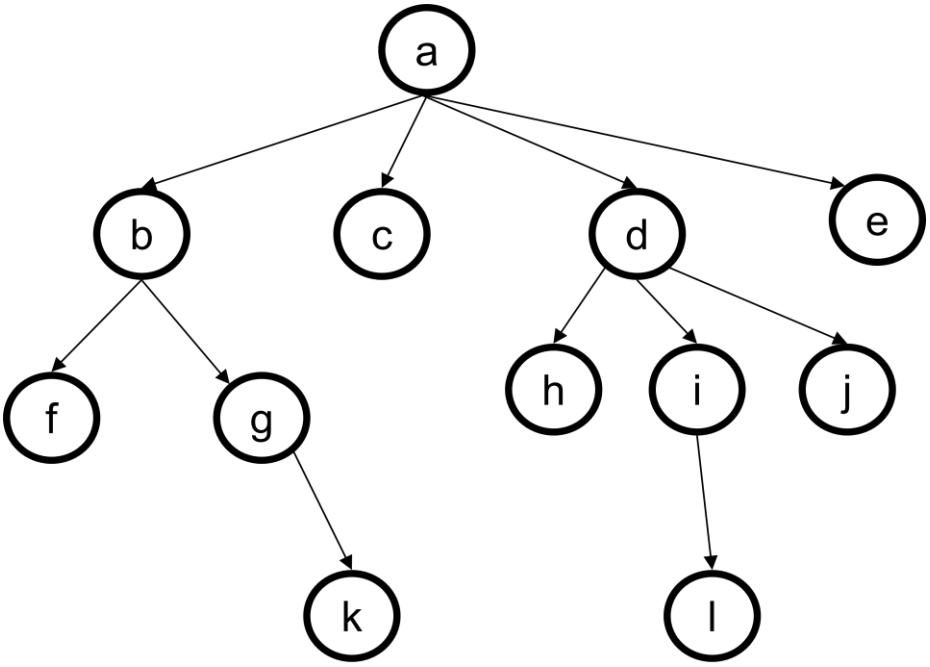
3. Repeat 4 and 5 until queue is empty

4. Delete the node from Queue and mark it as visited

5. Insert the unvisited adjacent nodes of Vi to queue

6. Repeat until all the nodes are visited

7. Stop



Starting from node ‘a’

Insert node ‘a’ into queue

Queue

a									

Delete node ‘a’ from queue and mark it as visited

Result: a

- Then visits nodes adjacent to ‘a’ in some specified order (e.g., alphabetical) and insert into queue

b	c	d	e						

Delete next first element from queue ie, ‘b’ and mark it as visited

Result: ab

- Then visits nodes adjacent to ‘b’ in some specified order (e.g., alphabetical) and insert into queue

c	d	e	f	g					

Delete next first element from queue ie, ‘c’ and mark it as visited

Result: abc

- Then visits nodes adjacent to ‘c’ in some specified order (e.g., alphabetical) and insert into queue. There is nodes are adjacent to ‘c’

d	e	f	g						

Delete next first element from queue ie, ‘d’ and mark it as visited

Result: abcd

- Then visits nodes adjacent to ‘d’ in some specified order (e.g., alphabetical) and insert into queue.

e	f	g	h	i	j				

Delete next first element from queue ie, ‘e’ and mark it as visited

Result: abcde

- Then visits nodes adjacent to ‘e’ in some specified order (e.g., alphabetical) and insert into queue. There is nodes are adjacent to ‘e’

f	g	h	i	j					

Delete next first element from queue ie, ‘f’ and mark it as visited

Result: abcdef

- Then visits nodes adjacent to ‘f’ in some specified order (e.g., alphabetical) and insert into queue. There is nodes are adjacent to ‘f’

g	h	i	j						

Delete next first element from queue ie, ‘g’ and mark it as visited

Result: abcdefg

- Then visits nodes adjacent to ‘g’ in some specified order (e.g., alphabetical) and insert into queue.

h	i	j	k						

Delete next first element from queue ie, ‘h’ and mark it as visited

Result: abcdefgh

- Then visits nodes adjacent to ‘h’ in some specified order (e.g., alphabetical) and insert into queue. There is nodes are adjacent to ‘h’

i	j	k							

Delete next first element from queue ie, ‘i’ and mark it as visited

Result: abcdefghi

- Then visits nodes adjacent to ‘i’ in some specified order (e.g., alphabetical) and insert into queue.

j	k	l							

Delete next first element from queue ie, ‘j’ and mark it as visited

Result: abcdefghij

- Then visits nodes adjacent to ‘j’ in some specified order (e.g., alphabetical) and insert into queue. There is nodes are adjacent to ‘j’

k	l								

Delete next first element from queue ie, ‘k’ and mark it as visited

Result: abcdefghijk

- Then visits nodes adjacent to ‘k’ in some specified order (e.g., alphabetical) and insert into queue. There is nodes are adjacent to ‘k’

l									

Delete next first element from queue ie, ‘l’ and mark it as visited

Result: abcdefghijkl

- Then visits nodes adjacent to ‘l’ in some specified order (e.g., alphabetical) and insert into queue. There is nodes are adjacent to ‘l’

DEPTH-FIRST SEARCH

Here the data structure used is STACK

Algorithm

1. Initialize all the nodes as visited

2. PUSH the first node/ starting node Vi to stack

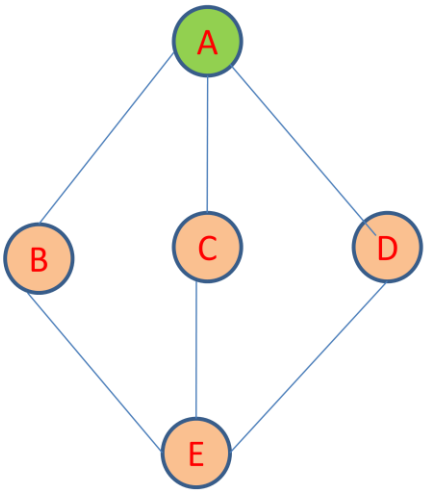
3. Repeat 4 and 5 until stack is empty

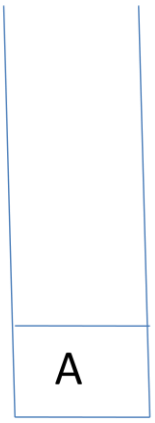
4. POP the top node of stack Vi and mark it as visited

5. PUSH the unvisited adjacent nodes of Vi to stack

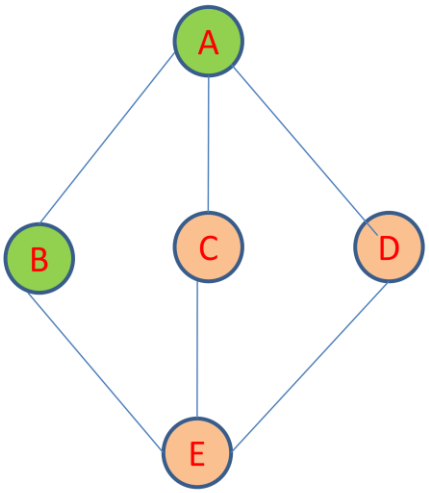
6. Repeat until all the nodes are visited

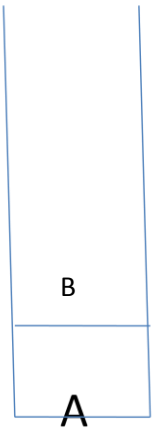
7. Stop





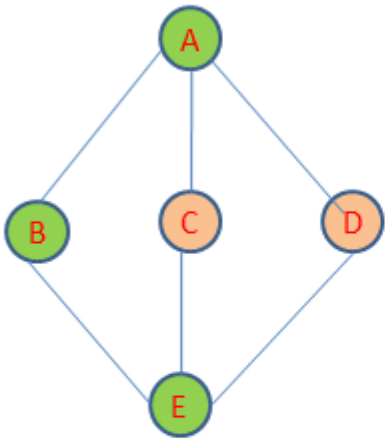
- Initially all nodes are in ready state
- Let the starting node be A. Push it on to stack & display it
- **Output: A**

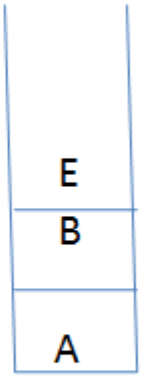




1. Push any of the adjacent unvisited vertex B onto stack and print it

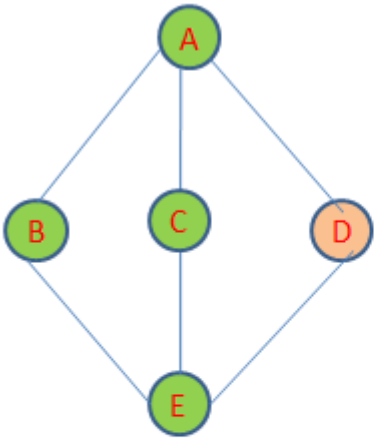
Output: A B

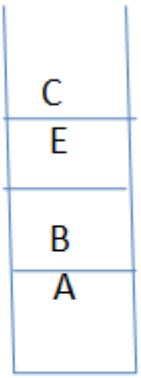




1. Push any of the adjacent unvisited vertex E onto stack and print it

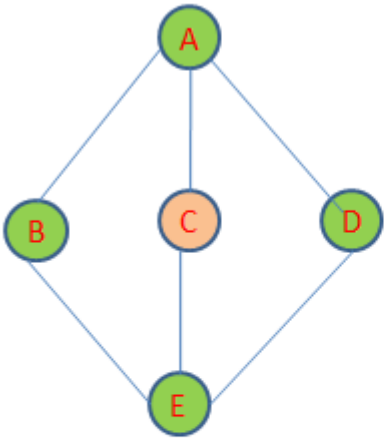
Output: A B E

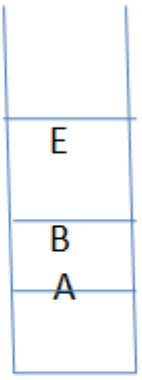




Push any of the adjacent unvisited vertex of E onto stack and print it

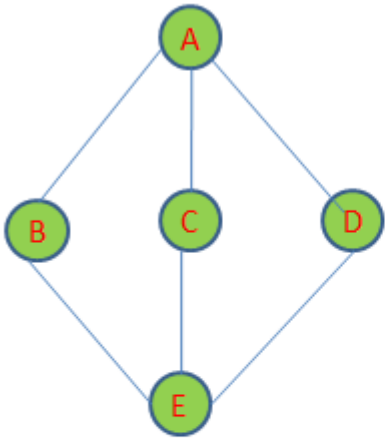
Output: A B E C

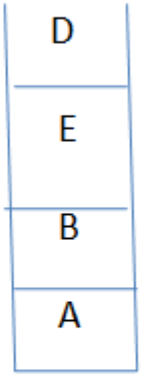




1. Now no adjacent unvisited neighbor node for C
2. Pop it and **backtrack** to next top node in the stack

Output: A B E C





1. Push adiecent unvisited node of E ie, D on stack and print it
2. Now no unvisited vertex!!

Output: A B E C D

There is no unvisited adjacent vertex for D. So backtrack, POP the stack top and backtrack to top element. Then do the same steps until stack is empty

MODULE 5

SEQUENTIAL SEARCH

Sequential_search(key)

Input: An unsorted array a[], n is the no.of elements, key indicates the element to be searched

Output: Target element found status

DS: Array

1. Start
2. i=0
3. flag=0
4. While i<n and flag=0
    1. If a[i]=key
        1. Flag=1
        2. Index=i

    2.end if

    3. i=i+1
5. end while
6. if flag=1
    1. print "the key is found at location index"
7. else

    1. print "key is not found"

7. end if
8. stop

Analysis

In this algorithm the key is searched from first to last position in linear manner. In the case of a successful search, it search elements up to the position in the array where it finds the key. Suppose it finds the key at first position, the algorithm behaves like best case, If the key is at the last position, then algorithm behaves like worst case. Thus the worst case time complexity is equal to the no. of comparison at worst case ie., equal to O(n). The time complexity in best case is O(1).

The average case time complexity =(no. of comparisons required when the key is in the first position + no. of comparisons required when the key is in second position+…+ no. of comparison when key is in nth position)/n

$$\frac{1 + 2 + \ldots n}{n} = \frac{n(n+1)}{2n}$$

=O(n)

Binary Search

Binary Search(key)

Input: An unsorted array a[], n is the no.of elements, key indicates the element to be searched

Output: Target element found status

DS: Array

1. Start
2. Start=0,end=n-1
3. Middle=(start+end)/2
4. While key!=a[middle] and start<end
    1. If key>a[middle]
        1. Start=middle+1

    2.else

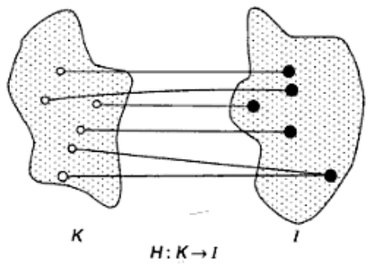
        1. end=middle-1

    3. end if
    4. middle=(start+end)/2
5. end while
6. if key=a[middle]
    1. print “the key is found at the position”
7. else
    1. print “the key is not found”
8. end if
9. stop

HASHING

We have seen about different search techniques (linear search, binary search) where search time is basically dependent on the no of elements and no. of comparisons performed.

Hashing is a technique which gives constant search time. In hashing the key value is stored in the hash table depending on the hash function. The hash function maps the key into corresponding index of the array(hash table). The main idea behind any hashing technique is to findone-to-one correspondence between a key value and an index in the hash table where the key value can be placed. Mathematically, this can be expressed as shown in figure below where K denotes a set of key values, I denotes a range of indices, and H denotes the mapping function from K to I.



All key values are mapped into some indices and more than one key value may be mapped into an index value. The function that governs this mapping is called the hash function. There are two principal criteria in deciding hash function H:K->I as follows.

1) The function H should be very easy and quick to compute
2) It should be easy to implement

As an example let us consider a hash table of size 10 whose indices are 0,1,2,…9.Suppose a set of key values are 10,19,35,43,62,59,31,49,77,33.Let us assume the hash function as stated below

1) Add the two digits in the key
2) Take the digit at the unit place of the result as index , ignore the digits at tenth place if any

Using this hash function, the mapping from key values to indices and to hash tables are shown below.

K	I
10	1
19	0
35	8
43	7
62	8
59	4
31	4
49	3
77	4
33	6

H: K→I

0	19
1	10
2	
3	49
4	59, 31, 77
5	
6	33
7	43
8	35, 62
9	

Hash table

HASH FUNCTIONS

There are various methods to define hash function

Division method

One of the fast hashing functions, and perhaps the most widely accepted, is the division method, which is defined as follows:

Choose a number h larger than the number n of keys in K. The hash function H is then defined by

H(k)=k(MOD h) if indices start from 0

Or

H(k)=k(MOD h)+1 if indices start from 1

Where k€K, a key value. The operator MOD defines the modulo arithmetic operator operation, which is equal to dividing k by h. For example if k=31 and **h=13 then,**

H(31)=31 MOD 13=5 (OR)

H(31)=31(MOD 13)+1=6

h is generally chosen to be a prime number and equal to the sizeof hash table

MID SQUARE METHOD

Another hash function which has been widely used in many applications is the mid square method. The hash function H is defined by H(k)=x, where x is obtained by selecting an appropriate number of bits or digits from the middle of the square of the key value k. example-

k : 1234 2345 3456

k^2 : 1522756 5499025 11943936

H(k) :525 492 933

For a three digit index requirement, after finding the square of key values,the digits at 2^{nd}, 4^{th} and 6^{th} position are chosen as their hash values.

FOLDING METHOD

Another fair method for a hash function is folding method. In this method, the

key k is partitioned into a number of parts k_1 , k2..kn where each part has equal no.of digits as the required address(index) width. Then these parts are added together in the hash function.

H(k)=k1+k2+…+kn. Where the last carry, if any is ignored. There are mainly two variations of this method.

1)fold shifting method

2) fold boundary method

Fold Shifting Method

In this method, after the partition even parts like k2, k4 are reversed before addition.

Fold boundary method

In this method, after the partition the boundary parts are reversed before addition

Example

-Assume size of each part is 2 then, the hash function is computed as follows

Key values k :	**1522756**	**5499025**	**11943936**
Chopping :	**01 52 27 56**	**05 49 90 25**	**11 94 39 36**
Pure folding :	**01+52+27+56=136**	**05+49+90+25=169**	**11+94+39+36=180**
Fold Shifting:	**10+52+72+56=190**	**50+49+09+25=133**	**11+94+93+36=234**
Fold Boundary	**:10+52+27+65=154**	**50+49+90+52=241**	**11+94+39+63=207**

DIGIT ANALYSIS METHOD

This method is particularly useful in the case of static files where the key values of all the records are known in advance. The basic idea of this hash function is to form hash address by extracting and/or shifting the extracted digits of the key. For any given set of keys, the position in the keys and the same rearrangement pattern must be used consistently. The decision for extraction and rearrangement is finalized after analysis of hash functions under different criteria.

Example: given a key value 6732541, it can be transformed to the hash address 427 by extracting the digits from even position. And then reversing this combination.ie 724 is the hash address.

Collision resolution and overflow handling techniques

There are several methods to resolve collision. Two important methods are listed below:

1) **Closed hashing(linear probing)**
2) **Open hashing (chaining)**

CLOSED HASHING

Suppose there is a hash table of size h and the key value is mapped to location i, with a hash function. The closed hashing can then be stated as follows.

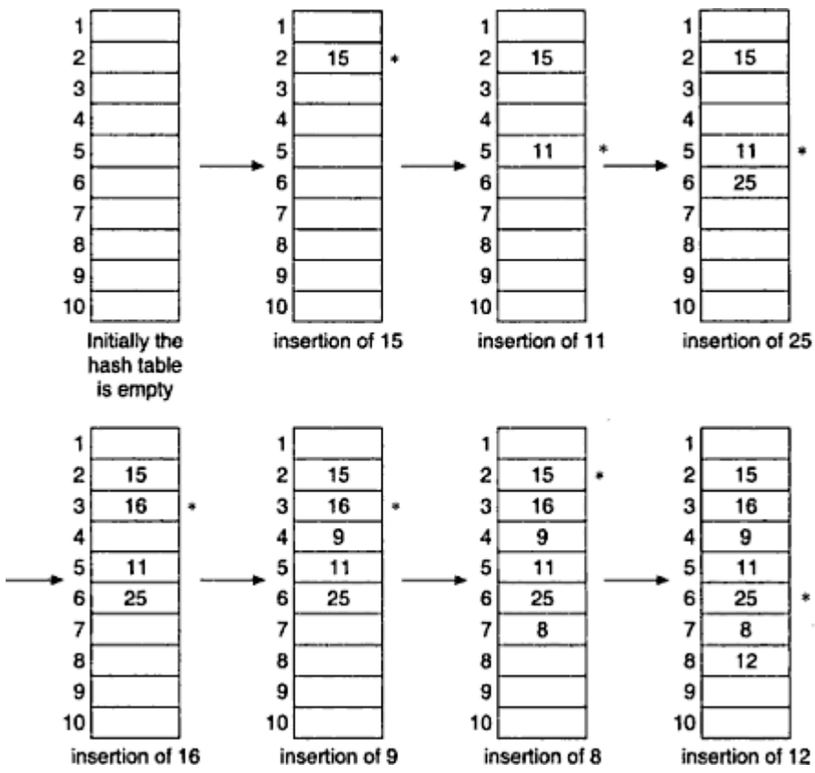
Start with the hash address where the collision has occurred,let it be i.

Then carry out a sequential search in the order:- i, i+1,i+2..h,1,2…,i-1 The search will continue until any one of the following occurs

- The key value is found
- An unoccupied location is found
- The searches reaches the location where search had started

The first case corresponds to successful search , and the other two case corresponds to unsuccessful search.Here the hash table is considered circular, so that when the last location is reached, the search proceeds to the first location of the table. This is why the technique is termed closed hashing. Since the technique searches in a straight line, it is alternatively termed as linear probing.

Example- Assume there is a hash table of size 10 and hash function uses the division method of remainder modulo 7, namely H(k)=k(MOD 7)+1.The construction of hash table for the key values 15,11,25,16,9,8,12,8 is illustrated below.



Drawback of closed hashing and its remedies

The major drawback of closed hashing is that as half of the hash table is filled, there is a tendency towards clustering.That is key values are clustered in large groups.and as a result sequential search becomes slower and slower. This kind of clustering is known as primary clustering.

The following are some solutions to avoid this situation

1)Random probing

2)Double hashing

3)Quadratic probing

Random Probing

Instead of using linear probing that generates sequential locations in order, a random location is generated using random probing.

An example of pseudo random number generator that generates such a random sequence is given below:

I=(i+m)MOD h+1

Where m and h are prime numbers. For example if m=5, and h=11 and initially=2 then random probing generates the sequence

8,3,9,4,10,5,11,6,1,7,2

Here all numbers are generated between 1 and 11 in a random order. Primary clustering problem is solved. Where as there is an issue of clustering when two keys are hashed into the same location and then they make use of the same sequence locations generated by the random probing, which is called as secondary clustering

Double Hashing

An alternative approach to solve the problem of secondary clustering is to make use of second hash function in addition to the first one. Suppose H1(k) is initially used hash function and H2(k) is the second one. These two functions are defined as

H1(k)=(k MOD h)+1 H2(k)=(k MOD (h-4))+1

Let h=11, and k=50 for an instance, then H1(50)=7 and H2(50)=2.

Now let k=28, then H1(28)=7 and H2(28)=5

Thus for the two key values hashing to the same location, rehashing generates two different locations alleviating the problem of secondary clustering.

Quadratic Probing

It is a collision resolution method that eliminates the primary clustering problem of linear probing. For linear probing, if there is a collision at location i, then the next locations i+1, i+2..etc are probed.But in quadratic probing next locations to be probed are $i+1^2$,$i+2^2$,$i+3^2$ ..etc . This method substantially reduces primary clustering, but it doesn't probe all the locations in the table.

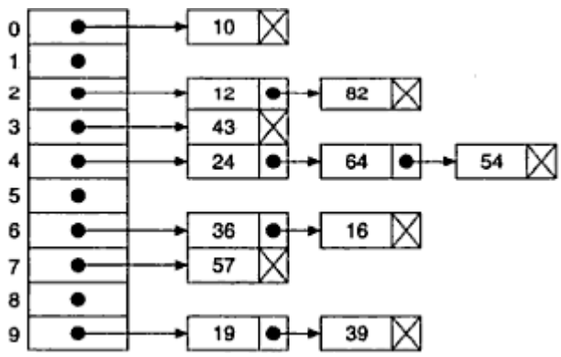
Open Hashing

Closed hashing method for collision resolution deals with arrays as hash tables and thus random positions can be quickly refer red. Two main disadvantages of closed hashing are

1) It is very difficult to handle the problem of overflow in a satisfactory manner
2) The key values are haphazardly intermixed and, on the average majority of the key values are from their hash locations increasing the number of probes which degrades the overall performance

To resolve these problems another hashing method called open hashing or separate chaining is used.

The chaining method uses hash table as an array of pointers. Each pointer points a linked list. That is here the hash table is an array of list of headers. Illustrated below is an example with hash table of size10.



For searching a key in hash table requires the following steps 1)Key is applied to hash function

2) Hash function returns the starting address of a particular linked list(where key may be present)

3)Then key is searched in that linked list

Performance Comparison Expected

Algorithm Name	**Best Case**	**Average Case**	**Worst Case**
Quick Sort	**O(n log n)**	**O(n log n)**	**O(n^2)**
Merge Sort	**O(n log n)**	**O(n log n)**	**O(n log n)**
Heap Sort	**O(n log n)**	**O(n log n)**	**O(n log n)**
Bubble Sort	**O(n)**	**O(n^2)**	**O(n^2)**
Selection Sort	**O(n^2)**	**O(n^2)**	**O(n^2)**
Insertion Sort	**O(n)**	**O(n^2)**	**O(n^2)**
Binary Search	**O(1)**	**O(log n)**	**O(log n)**
Linear Search	**O(1)**	**O(n)**	**O(n)**