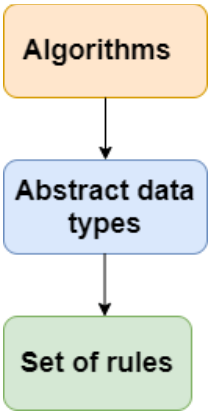
DATA STRUCTURE AND ALGORITHM

Unit I

Arrays and sequential representations – ordered lists – Stacks and Queues – Evaluation ofExpressions – Multiple Stacks and Queues – Singly Linked List – Linked Stacks and queues – Polynomial addition.

Data Structure Introduction:

The data structure name indicates itself that organizing the data in memory. There are many ways of organizing the data in the memory as we have already seen one of the data structures, i.e., array in C language. Array is a collection of memory elements in which data is stored sequentially, i.e., one after another. In other words, we can say that array stores the elements in a continuous manner. This organization of data is done with the help of an array of data structures. There are also other ways to organize the data in memory. To structure the data in memory, 'n' number of algorithms were proposed, and all these algorithms are known as **Abstract data types.** These abstract data types are the set of rules.



Types of Data Structures

There are two types of data structures:

- Primitive data structure
- Non-primitive data structure

Primitive Data structure

The primitive data structures are primitive data types. The int, char, float, double, and pointer are the primitive data structures that can hold a single value.

Non-Primitive Data structure

The non-primitive data structure is divided into two types:

- Linear data structure
- Non-linear data structure

Linear Data Structure

The arrangement of data in a sequential manner is known as a linear data structure. The data structures used for this purpose are Arrays, Linked list, Stacks, and Queues. In these data structures, one element is connected to only one another element in a linear form.

Non- Linear Structure:

When one element is connected to the 'n' number of elements known as a non-linear data structure. The best example is trees and graphs. In this case, the elements are arranged in a random manner.

Data structures can also be classified as:

- **Static data structure:** It is a type of data structure where the size is allocated at the compile time. Therefore, the maximum size is fixed.
- **Dynamic data structure:** It is a type of data structure where the size is allocated at the run time. Therefore, the maximum size is flexible.

Major Operations

The major or the common operations that can be performed on the data structures are:

- **Searching:** We can search for any element in a data structure.
- **Sorting:** We can sort the elements of a data structure either in an ascending or descending order.
- **Insertion:** We can also insert the new element in a data structure.
- **Updation:** We can also update the element, i.e., we can replace the element with another element.
- **Deletion:** We can also perform the delete operation to remove the element from the data structure.

Advantages of Data structures

The following are the advantages of a data structure:

- **Efficiency:** If the choice of a data structure for implementing a particular ADT is proper, it makes the program very efficient in terms of time and space.
- **Reusability:** The data structure provides reusability means that multiple client programs can use the data structure.
- **Abstraction:** The data structure specified by an ADT also provides the level of abstraction. The client cannot see the internal working of the data structure, so it does not have to worry about the implementation part. The client can only see the interface.

Arrays and Sequential Representation

Definition

- Arrays are defined as the collection of similar type of data items stored at contiguous memory locations.
- Arrays are the derived data type in C programming language which can store the primitive type of data such as int, char, double, float, etc.
- Array is the simplest data structure where each data element can be randomly accessed by using its index number.
- For example, if we want to store the marks of a student in 6 subjects, then we don't need to define different variable for the marks in different subject. instead of that, we can define an array which can store the marks in each subject at a the contiguous memory locations.

The array **marks[10]** defines the marks of the student in 10 different subjects where each subject marks are located at a particular subscript in the array i.e. **marks[0]** denotes the marks in first subject, **marks[1]** denotes the marks in 2nd subject and so on.

Properties of the Array

1. Each element is of same data type and carries a same size i.e. int = 4 bytes.
2. Elements of the array are stored at contiguous memory locations where the first element is stored at the smallest memory location.
3. Elements of the array can be randomly accessed since we can calculate the address of each element of the array with the given base address and the size of data element.

Advantages of Array

- Array provides the single name for the group of variables of the same type therefore, it is easy to remember the name of all the elements of an array.
- Traversing an array is a very simple process, we just need to increment the base address of the array in order to visit each element one by one.
- Any element in the array can be directly accessed by using the index.

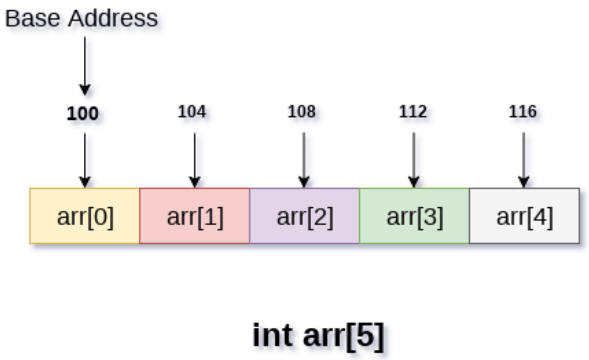
Memory Allocation of the array

As we have mentioned, all the data elements of an array are stored at contiguous locations in the main memory. The name of the array represents the base address or the address of first element in the main memory. Each element of the array is represented by a proper indexing**.**

The indexing of the array can be defined in three ways.

1. 0 (zero - based indexing) : The first element of the array will be arr[0].
2. 1 (one - based indexing) : The first element of the array will be arr[1].
3. n (n - based indexing) : The first element of the array can reside at any random index number.

   In the following image, we have shown the memory allocation of an array arr of size 5. The array follows 0-based indexing approach. The base address of the array is 100th byte. This will be the address of arr[0]. Here, the size of int is 4 bytes therefore each element will take 4 bytes in the memory.



In 0 based indexing, If the size of an array is n then the maximum index number, an element can have is **n-1**. However, it will be n if we use **1** based indexing.

Accessing Elements of an array

To access any random element of an array we need the following information:

1. Base Address of the array.
2. Size of an element in bytes.
3. Which type of indexing, array follows.

Address of any element of a 1D array can be calculated by using the following formula:

Byte address of element A[i] = base address + size * (i - first index)

2D Array

2D array can be defined as an array of arrays. The 2D array is organized as matrices which can be represented as the collection of rows and columns.

However, 2D arrays are created to implement a relational database look alike data structure. It provides ease of holding bulk of data at once which can be passed to any number of functions wherever required.

How to declare 2D Array

The syntax of declaring two dimensional array is very much similar to that of a one dimensional array, given as follows.

```
1. int arr[max_rows][max_columns];
```

however, It produces the data structure which looks like following.

Above image shows the two dimensional array, the elements are organized in the form of rows and columns. First element of the first row is represented by a[0][0] where the number shown in the first index is the number of that row while the number shown in the second index is the number of the column.

How do we access data in a 2D array

Similar to one dimensional arrays, we can access the individual cells in a 2D array by using the indices of the cells. There are two indices attached to a particular cell, one is its row number while the other is its column number.

However, we can store the value stored in any particular cell of a 2D array to some variable x by using the following syntax.

```
int x = a[i][j];
```

where i and j is the row and column number of the cell respectively.

We can assign each cell of a 2D array to 0 by using the following code:

```
for ( int i=0; i<n ;i++)
{
   for (int j=0; j<n; j++)
   {
      a[i][j] = 0;
   }
}
```

Initializing 2D Arrays

We know that, when we declare and initialize one dimensional array in C programming simultaneously, we don't need to specify the size of the array. However this will not work with 2D arrays. We will have to define at least the second dimension of the array.

The syntax to declare and initialize the 2D array is given as follows.

```
int arr[2][2] = {0,1,2,3};
```

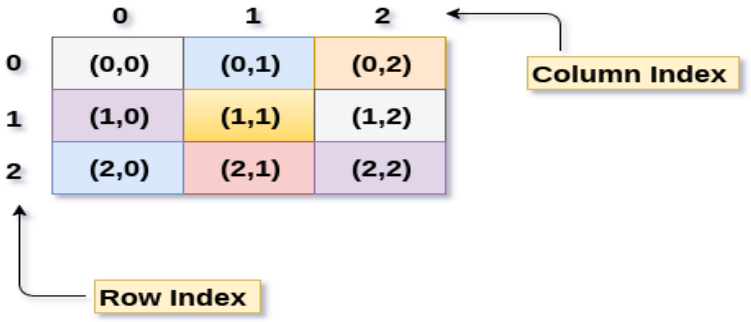
The number of elements that can be present in a 2D array will always be equal to (**number of rows * number of columns**).

Mapping 2D array to 1D array

When it comes to map a 2 dimensional array, most of us might think that why this mapping is required. However, 2 D arrays exists from the user point of view. 2D arrays are created to implement a relational database table lookalike data structure, in computer memory, the storage technique for 2D array is similar to that of an one dimensional array.

The size of a two dimensional array is equal to the multiplication of number of rows and the number of columns present in the array. We do need to map two dimensional array to the one dimensional array in order to store them in the memory.

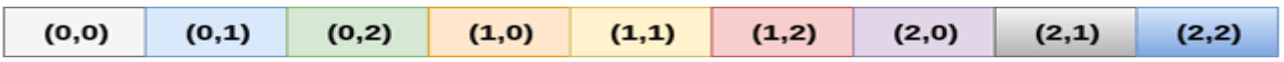
A 3 X 3 two dimensional array is shown in the following image. However, this array needs to be mapped to a one dimensional array in order to store it into the memory.



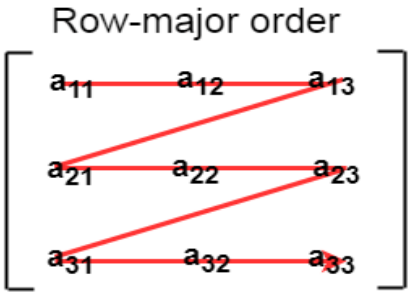
There are two main techniques of storing 2D array elements into memory

1. Row Major ordering

In row major ordering, all the rows of the 2D array are stored into the memory contiguously. Considering the array shown in the above image, its memory allocation according to row major order is shown as follows.

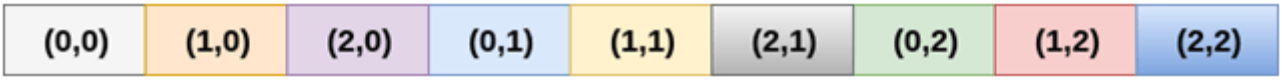


first, the 1st row of the array is stored into the memory completely, then the 2nd row of the array is stored into the memory completely and so on till the last row.

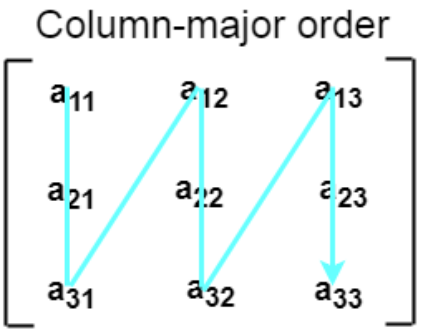


2. Column Major ordering

According to the column major ordering, all the columns of the 2D array are stored into the memory contiguously. The memory allocation of the array which is shown in in the above image is given as follows.



first, the 1st column of the array is stored into the memory completely, then the 2nd row of the array is stored into the memory completely and so on till the last column of the array.



Calculating the Address of the random element of a 2D array

Due to the fact that, there are two different techniques of storing the two dimensional array into the memory, there are two different formulas to calculate the address of a random element of the 2D array.

By Row Major Order

If array is declared by a[m][n] where m is the number of rows while n is the number of columns, then address of an element a[i][j] of the array stored in row major order is calculated as,

1. Address(a[i][j]) = B. A. + (i * n + j) * size
   where, B. A. is the base address or the address of the first element of the array a[0][0] .

Example :

1. a[10...30, 55...75], base address of the array (BA) = 0, size of an element = 4 bytes .
2. Find the location of a[15][68].
3.
4. Address(a[15][68]) = 0 +

5. ((15 - 10) x (68 - 55 + 1) + (68 - 55)) x 4
6.
7. = (5 x 14 + 13) x 4
8. = 83 x 4
9. = 332 answer

By Column major order

If array is declared by a[m][n] where m is the number of rows while n is the number of columns, then address of an element a[i][j] of the array stored in row major order is calculated as,

1. Address(a[i][j]) = ((j*m)+i)*Size + BA

where BA is the base address of the array.

Example:

A [-
5 ... +20][20 ... 70], BA = 1020, Size of element = 8 bytes. Find the location of a[0][30].

Address [A[0][30]) = ((30-20) x 24 + 5) x 8 + 1020 = 245 x 8 + 1020 = 2980 bytes

Ordered list

An ordered list is a list in which the order of the items is significant. However, the items in an ordered list are not necessarily sorted. Consequently, it is possible to change the order o items and still have a valid ordered list.

Consider a list of the titles of the chapters in this book. The order of the items in the list corresponds to the order in which they appear in the book. However, since the chapter titles are not sorted alphabetically, we cannot consider the list to be sorted. Since it is possible to change the order of the chapters in book, we must be able to do the same with the items of the list. As a result, we may insert an item into an ordered list at any position.

A searchable container is a container that supports the following additional operations:

Insert
-used to put objects into a the container;
withdraw
-used to remove objects from the container;
find
-used to locate objects in the container;

isMember

-used to test whether a given object instance is in the container.

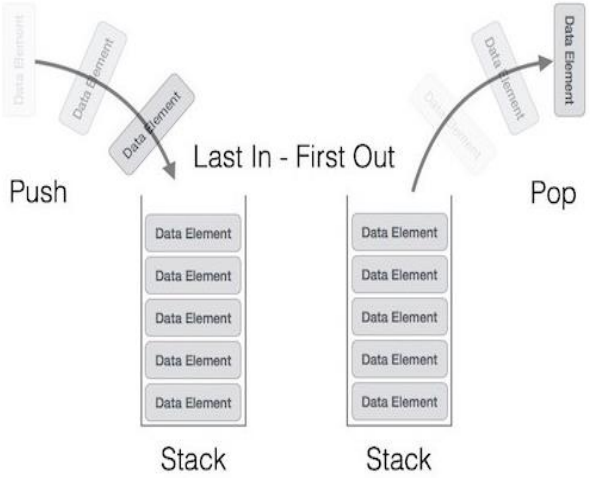
stack

A real-world stack allows operations at one end only. For example, we can place or remove a card or plate from the top of the stack only. At any given time, we can only access the top element of a stack.

This feature makes it LIFO data structure. LIFO stands for Last-in-first-out. Here, the element which is placed (inserted or added) last, is accessed first. In stack terminology, insertion operation is called **PUSH** operation and removal operation is called **POP** operation.

Stack Representation

The following diagram depicts a stack and its operations −



A stack can be implemented by means of Array, Structure, Pointer, and Linked List. Stack can either be a fixed size one or it may have a sense of dynamic resizing. Here, we are going to implement stack using arrays, which makes it a fixed size stack implementation.

Basic Operations

Stack operations may involve initializing the stack, using it and then de-initializing it. Apart from these basic stuffs, a stack is used for the following two primary operations −

- **push()** − Pushing (storing) an element on the stack.
- **pop()** − Removing (accessing) an element from the stack.

When data is PUSHed onto stack.

To use a stack efficiently, we need to check the status of stack as well. For the same purpose, the following functionality is added to stacks −

- **peek()** − get the top data element of the stack, without removing it.
- **isFull()** − check if stack is full.

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

At all times, we maintain a pointer to the last PUSHed data on the stack. The **top** pointer provides top value of the stack without actually removing it.

First we should learn about procedures to support stack functions −

peek()

Algorithm of peek() function −

```
begin procedure peek
   return stack[top]
end procedure
```

isfull()

Algorithm of isfull() function −

```
begin procedure isfull

   if top equals to MAXSIZE
      return true
   else
      return false
   endif

end procedure
```

isempty()

Algorithm of isempty() function −

```
begin procedure isempty

   if top less than 1
      return true
   else
      return false
   endif

end procedure
```

Implementation of isempty() function in C programming language is slightly different. We initialize top at -1, as the index in array starts from 0. So we check if the top is below zero or -1 to determine if the stack is empty. Here's the code −

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.

If the linked list is used to implement the stack, then in step 3, we need to allocate space dynamically.

Algorithm for PUSH Operation

A simple algorithm for Push operation can be derived as follows −

```
begin procedure push: stack, data

   if stack is full
      return null
   endif

   top ← top + 1
   stack[top] ← data

end procedure
```

Pop Operation

Accessing the content while removing it from the stack, is known as a Pop Operation. In an array implementation of pop() operation, the data element is not actually removed, instead **top** is decremented to a lower position in the stack to point to the next value. But in linked-list implementation, pop() actually removes data element and deallocates memory space.

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 for Pop Operation

A simple algorithm for Pop operation can be derived as follows −

```
begin procedure pop: stack

   if stack is empty
      return null
   endif
```

```
   data ← stack[top]
   top ← top - 1
   return data

end procedure
```

Queue

Queue is an abstract data structure, somewhat similar to Stacks. Unlike stacks, a queue is open at both its ends. One end is always used to insert data (enqueue) and the other is used to remove data (dequeue). Queue follows First-In-First-Out methodology, i.e., the data item stored first will be accessed first.

Queue Representation

As we now understand that in queue, we access both ends for different reasons. The following diagram given below tries to explain queue representation as data structure −

Basic Operations

Queue operations may involve initializing or defining the queue, utilizing it, and then completely erasing it from the memory. Here we shall try to understand the basic operations associated with queues −

- **enqueue()** − add (store) an item to the queue.
- **dequeue()** − remove (access) an item from the queue.

Few more functions are required to make the above-mentioned queue operation efficient. These are −

- **peek()** − Gets the element at the front of the queue without removing it.
- **isfull()** − Checks if the queue is full.
- **isempty()** − Checks if the queue is empty.

In queue, we always dequeue (or access) data, pointed by **front** pointer and while enqueing (or storing) data in the queue we take help of **rear** pointer.

peek()

This function helps to see the data at the **front** of the queue. The algorithm of peek() function is as follows −

Algorithm

```
begin procedure peek
   return queue[front]
end procedure
```

isfull()

As we are using single dimension array to implement queue, we just check for the rear pointer to reach at MAXSIZE to determine that the queue is full. In case we maintain the queue in a circular linked-list, the algorithm will differ. Algorithm of isfull() function –

Algorithm

```
begin procedure isfull

   if rear equals to MAXSIZE
      return true
   else
      return false
   endif

end procedure
```

isempty()

Algorithm of isempty() function −

Algorithm

```
begin procedure isempty

   if front is less than MIN  OR front is greater than rear
      return true
   else
      return false
   endif

end procedure
```

If the value of **front** is less than MIN or 0, it tells that the queue is not yet initialized, hence empty.

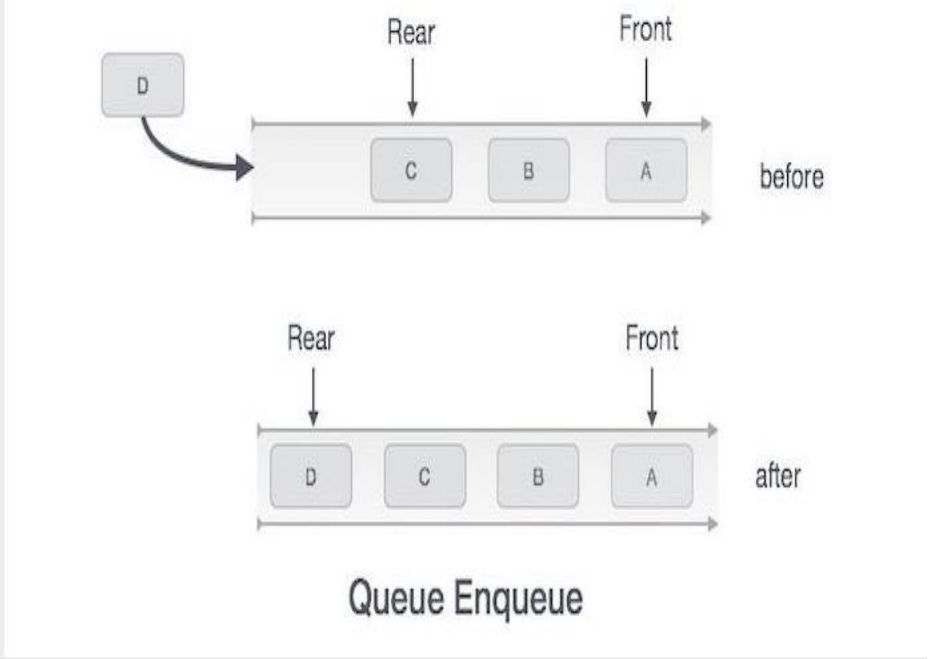
Enqueue Operation

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.

Sometimes, we also check to see if a queue is initialized or not, to handle any unforeseen situations.



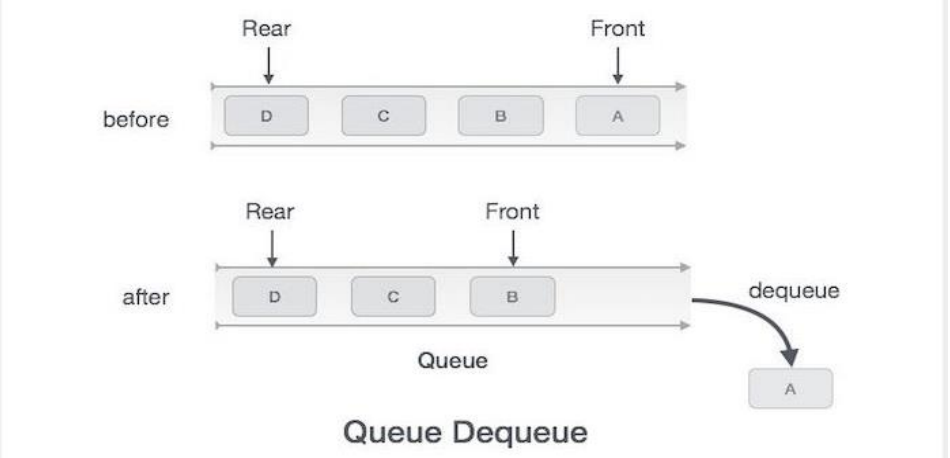
Algorithm for enqueue operation

```
procedure enqueue(data)
      if queue is full
      return overflow
   endif
      rear ← rear + 1
   queue[rear] ← data
   return true
   end procedure
```

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.



Queue Dequeue

Algorithm for dequeue operation

```
procedure dequeue

   if queue is empty
      return underflow
   end if

   data = queue[front]
   front ← front + 1
   return true

end procedure
```

Evaluation of Expression

The way to write arithmetic expression is known as a **notation**. An arithmetic expression can be written in three different but equivalent notations, i.e., without changing the essence or output of an expression. These notations are –

- Infix Notation
- Prefix (Polish) Notation
- Postfix (Reverse-Polish) Notation

These notations are named as how they use operator in expression. We shall learn the same here in this chapter.

Infix Notation

We write expression in **infix** notation, e.g. a - b + c, where operators are used **in**-between operands. It is easy for us humans to read, write, and speak in infix notation but the same does not go well with computing devices. An algorithm to

process infix notation could be difficult and costly in terms of time and space consumption.

Prefix Notation

In this notation, operator is **prefix**ed to operands, i.e. operator is written ahead of operands. For example, **+ab**. This is equivalent to its infix notation **a + b**. Prefix notation is also known as **Polish Notation**.

Postfix Notation

This notation style is known as **Reversed Polish Notation**. In this notation style, the operator is **postfix**ed to the operands i.e., the operator is written after the operands. For example, **ab+**. This is equivalent to its infix notation **a + b**.

The following table briefly tries to show the difference in all three notations −

Sr.No.	Infix Notation	Prefix Notation	Postfix Notation
1	a + b	+ a b	a b +
2	(a + b) ∗ c	∗ + a b c	a b + c ∗
3	a ∗ (b + c)	∗ a + b c	a b c + ∗
4	a / b + c / d	+ / a b / c d	a b / c d / +
5	(a + b) ∗ (c + d)	∗ + a b + c d	a b + c d + ∗
6	((a + b) ∗ c) - d	- ∗ + a b c d	a b + c ∗ d -

Parsing Expressions

As we have discussed, it is not a very efficient way to design an algorithm or program to parse infix notations. Instead, these infix notations are first converted into either postfix or prefix notations and then computed.

To parse any arithmetic expression, we need to take care of operator precedence and associativity also.

Precedence

When an operand is in between two different operators, which operator will take the operand first, is decided by the precedence of an operator over others. For example −

a+b*c -→ a+(b*c)

As multiplication operation has precedence over addition, b * c will be evaluated first. A table of operator precedence is provided later.

Associativity

Associativity describes the rule where operators with the same precedence appear in an expression. For example, in expression a + b − c, both + and – have the same precedence, then which part of the expression will be evaluated first, is determined by associativity of those operators. Here, both + and − are left associative, so the expression will be evaluated as **(a + b) − c**.

Precedence and associativity determines the order of evaluation of an expression. Following is an operator precedence and associativity table (highest to lowest) −

Sr.No.	Operator	Precedence	Associativity
1	Exponentiation ^	Highest	Right Associative
2	Multiplication (∗) & Division (/)	Second Highest	Left Associative
3	Addition (+) & Subtraction (−)	Lowest	Left Associative

The above table shows the default behavior of operators. At any point of time in expression evaluation, the order can be altered by using parenthesis. For example −

In **a + b*c**, the expression part **b*c** will be evaluated first, with multiplication as precedence over addition. We here use parenthesis for **a + b** to be evaluated first, like **(a + b)*c**.

Postfix Evaluation Algorithm

```
Step 1 - scan the expression from left to right
Step 2 - if it is an operand push it to stack
Step 3 - if it is an operator pull operand from stack and perform
operation
Step 4 - store the output of step 3, back to stack
Step 5 - scan the expression until all operands are consumed
Step 6 - pop the stack and perform operation
```

A postfix expression is a collection of operators and operands in which the operator is placed after the operands. That means, in a postfix expression the operator follows the operands.

Postfix Expression has following general structure...

Operand1 Operand2 Operator

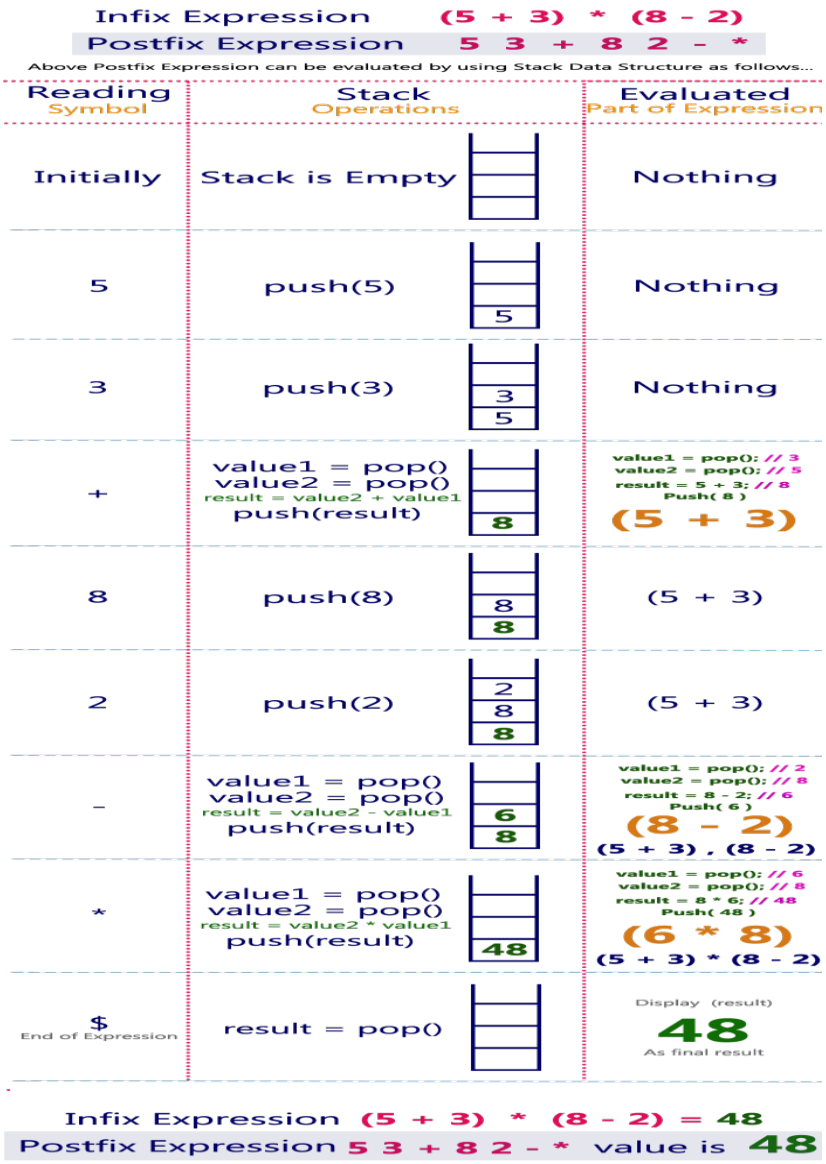
Example

Postfix Expression Evaluation using Stack Data Structure

A postfix expression can be evaluated using the Stack data structure. To evaluate a postfix expression using Stack data structure we can use the following steps...**Read all the symbols one by one from left to right in the given Postfix Expression**

1. **If the reading symbol is operand, then push it on to the Stack.**
2. **If the reading symbol is operator (+ , - , * , / etc.,), then perform TWO pop operations and store the two popped oparands in two different variables (operand1 and operand2). Then perform reading symbol operation using operand1 and operand2 and push result back on to the Stack.**
3. **Finally! perform a pop operation and display the popped value as final result.**

Example Consider the following Expression..



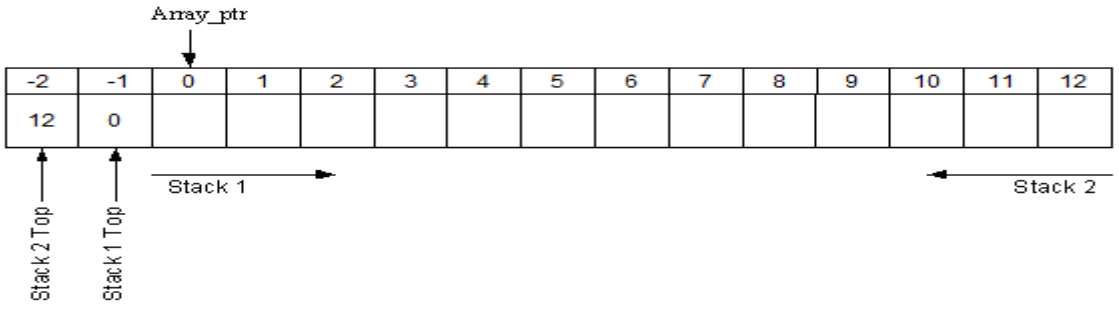
Multiple Stacks and Queues:

Multiple Stacks:

Following pictures are two ways to do two stacks in array:

1. None fixed size of the stacks:

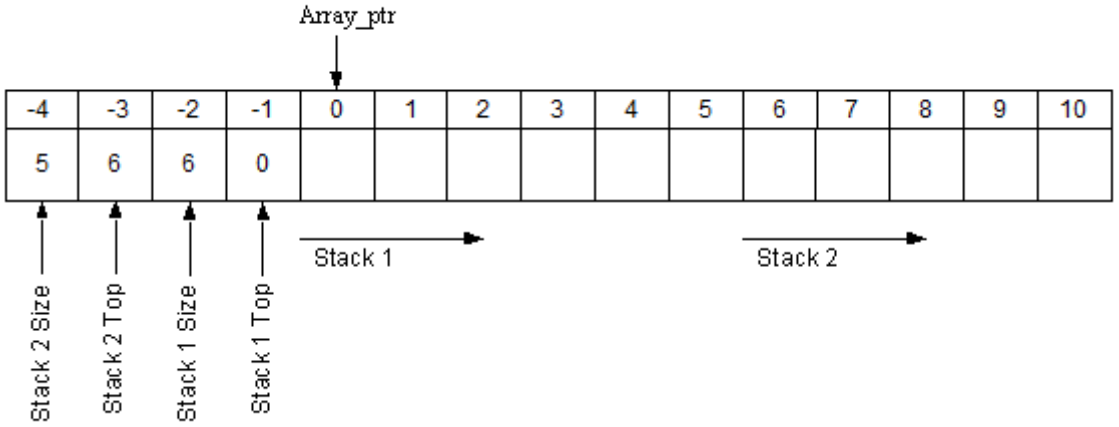
Graphical Picture: without fixed size of stack



- Stack 1 expands from the 0th element to the right
- Stack 2 expands from the 12th element to the left
- As long as the value of Top1 and Top2 are not next to each other, it has free elements for input the data in the array
- When both Stacks are full, Top1 and Top 2 will be next to each other
- There is no fixed boundary between Stack 1 and Stack 2
- Elements –1 and –2 are using to store the information needed to manipulate the stack (subscript for Top 1 and Top 2)

2. Fixed size of the stacks:

Graphical Picture: with fixed size of stack



- Stack 1 expands from the 0th element to the right
- Stack 2 expands from the 6th element to the left
- As long as the value of Top 1 is less than 6 and greater than 0, Stack 1 has free elements to input the data in the array
- As long as the value of Top 2 is less than 11 and greater than 5, Stack 2 has free elements to input the data in the array
- When the value of Top 1 is 5, Stack 1 is full
- When the value of Top 2 is 10, stack 2 is full
- Elements –1 and –2 are using to store the size of Stack 1 and the subscript of the array for Top 1 needed to manipulate Stack 1
- Elements –3 and –4 are using to store the size of Stack 2 and the subscript of the array for Top 2 needed to manipulate Stack 2

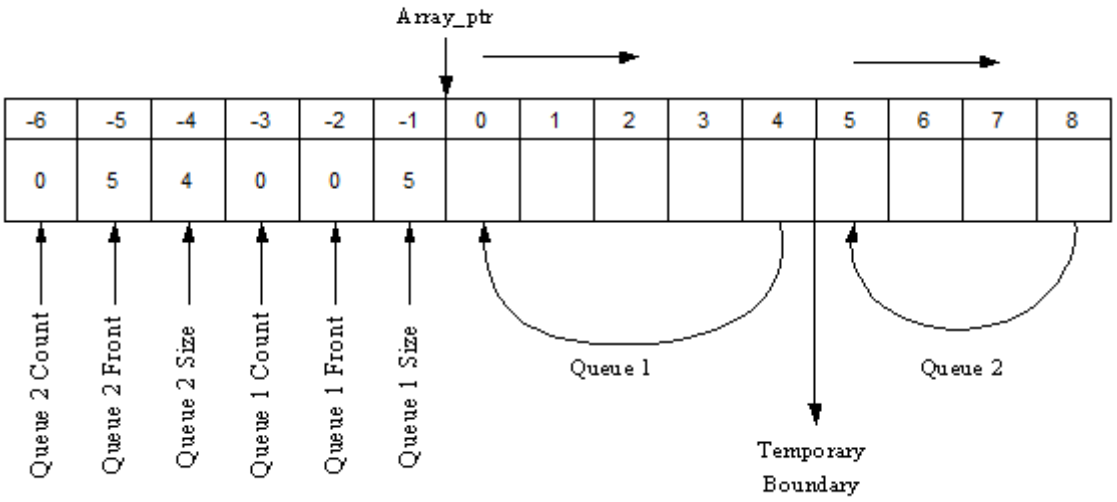
```
 procedure ADD (i,X ) //add element X to the i'th stack, 1 i n//
 if T(i) = B(i + 1) then call STACK-FULL (i)
T(i) ← T(i) + 1
 V(T(i)) ← X //add X to the i'th stack//
 end ADD

procedure DELETE(i,X ) //delete topmost element of stack i//
 if T(i) = B(i) then call STACK-EMPTY(i)
X ← V(T(i))
 T(i) ← T(i) - 1
end DELETE
```

Multiple Queues:

Following pictures are two ways to do two queues in array:

1. None fixed size of the queues:

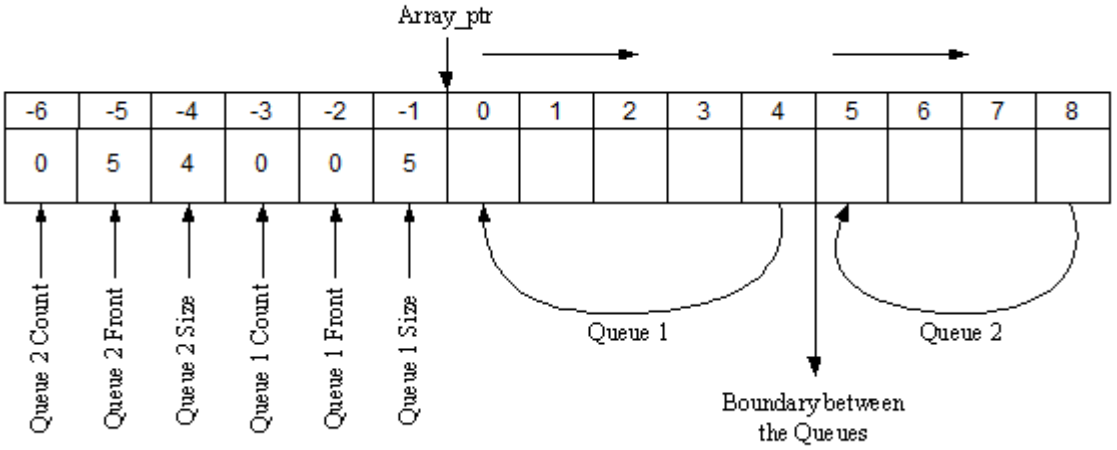
Graphical Picture: without fixed size of the queue



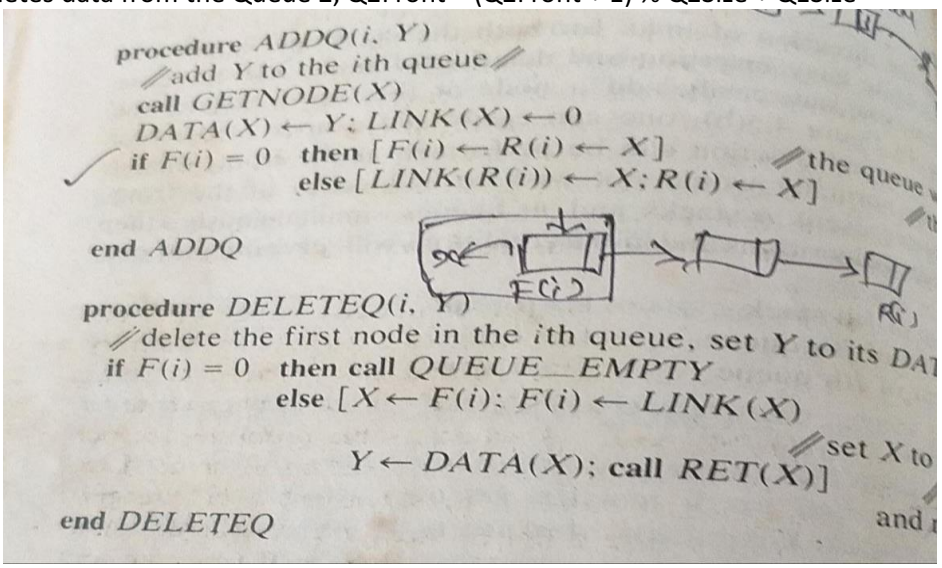
- Queue 1 expands from the 0th element to the right and circular back to the 0th element
- Queue 2 expands from the 8th element to the left and circular back to the 8th element
- Temporary boundary between the Queue 1 and the Queue 2; as long as there has free elements in the array and boundary would be shift
- Free elements could be anywhere in the Queue such as before the front, after the rear, and between front and rear in the Queue
- Queue 1's and Queue 2 's size could be change if it is necessary. When the Queue 1 is full and the Queue 2 has free space; the Queue 1 can increase the size to use that free space from the Queue 2. Same way for the Queue 2
- Elements –1, –2, and –3 are using to store the size of the Queue 1, the front of the Queue 1, and the data count for the Queue 1 needed to manipulate the Queue 1
- Elements –4, –5, and –6 are using to store the size of the Queue 2, the front of the Queue 2, and the data count for the Queue 2 needed to manipulate the Queue 2
- Inserts data to the Queue 1, Q1Rear = (Q1Front + Q1count) % Q1Size
- Inserts data to the Queue 2, Q2Rear = (Q2Front + Q2count) % Q2Size + Q1Size
- Deletes data from the Queue 1, Q1Front = (Q1Front + 1) % Q1Size
- Deletes data from the Queue 2, Q2Front = (Q2Front + 1) % Q2Size + Q1Size

2. Fixed size of the queue:

Graphical Picture: with fixed size of the queue



- Queue 1 expands from the 0th element to the 4th element and circular back to 0th element
- Queue 2 expands from the 8th element to the 5th element and circular back to 8th element
- The boundary is fixed between the Queue 1 and the Queue 2
- Free elements could be anywhere in the Queue such as before the front, after the rear, and between front and rear in the Queue
- Elements –1, –2, and –3 are using to store the size of the Queue 1, the front of the Queue 1, and the data count for the Queue 1 needed to manipulate the Queue 1
- Elements –4, –5, and –6 are using to store the size of the Queue 2, the front of the Queue 2, and the data count for the Queue 2 needed to manipulate the Queue 2
- Inserts data to the Queue 1, Q1Rear = (Q1Front + Q1count) % Q1Size
- Inserts data to the Queue 2, Q2Rear = (Q2Front + Q2count) % Q2Size + Q1Size
- Deletes data from the Queue 1, Q1Front = (Q1Front + 1) % Q1Size
- Deletes data from the Queue 2, Q2Front = (Q2Front + 1) % Q2Size + Q1Size



A linked list is a sequence of data structures, which are connected together via links.

Linked List is a sequence of links which contains items. Each link contains a connection to another link. Linked list is the second most-used data structure after array. Following are the important terms to understand the concept of Linked List.

- **Link** − Each link of a linked list can store a data called an element.

- **Next** − Each link of a linked list contains a link to the next link called Next.
- **LinkedList** − A Linked List contains the connection link to the first link called First.

Linked List Representation

Linked list can be visualized as a chain of nodes, where every node points to the next node.

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

- Linked List contains a link element called first.
- Each link carries a data field(s) and a link field called next.
- Each link is linked with its next link using its next link.
- Last link carries a link as null to mark the end of the list.

Types of Linked List

Following are the various types of linked list.

- **Simple Linked List** − Item navigation is forward only.
- **Doubly Linked List** − Items can be navigated forward and backward.
- **Circular Linked List** − Last item contains link of the first element as next and the first element has a link to the last element as previous.

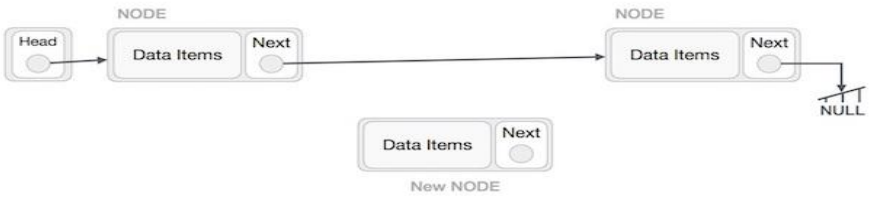
Basic Operations

Following are the basic operations supported by a list.

- **Insertion** − Adds an element at the beginning of the list.
- **Deletion** − Deletes an element at the beginning of the list.
- **Display** − Displays the complete list.
- **Search** − Searches an element using the given key.
- **Delete** − Deletes an element using the given key.

Insertion Operation

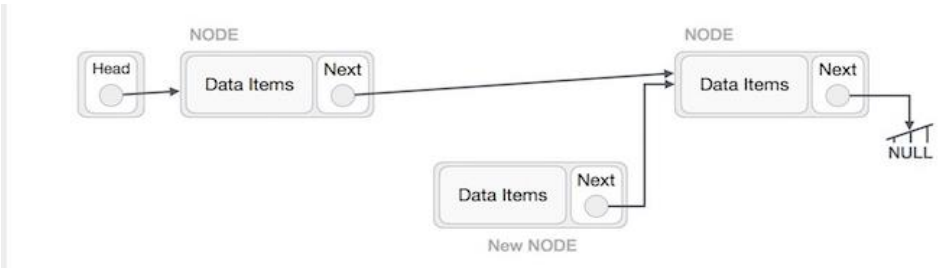
Adding a new node in linked list is a more than one step activity. We shall learn this with diagrams here. First, create a node using the same structure and find the location where it has to be inserted.



Imagine that we are inserting a node **B** (NewNode), between **A** (LeftNode) and **C** (RightNode). Then point B.next to C −

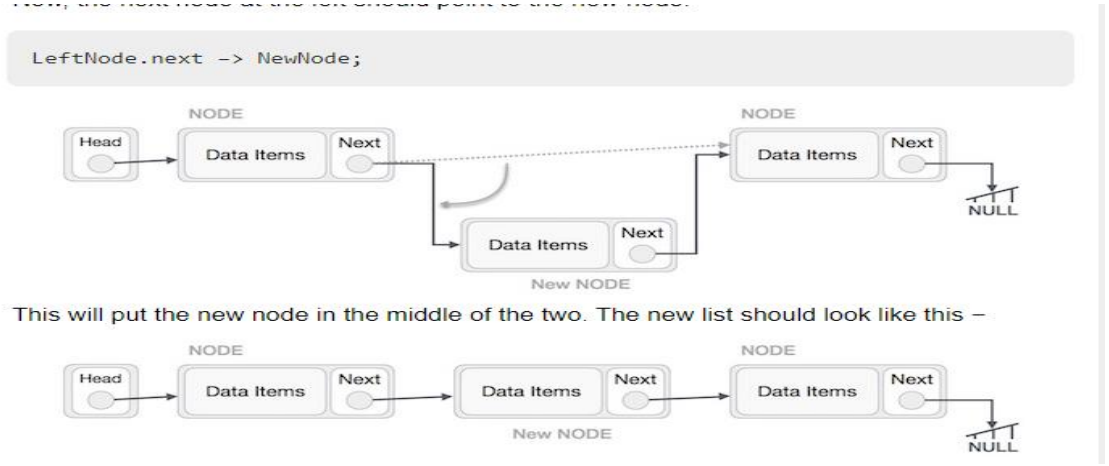
```
NewNode.next -> RightNode;
```

It should look like this −



Now, the next node at the left should point to the new node.

```
LeftNode.next -> NewNode;
```



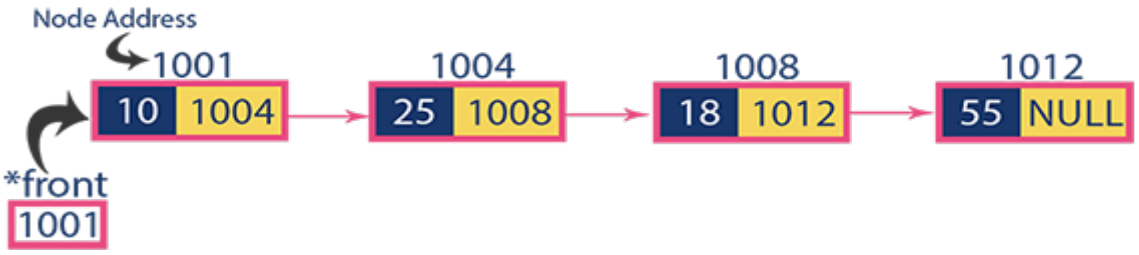
What is Single Linked List?

Simply a list is a sequence of data, and the linked list is a sequence of data linked with each other.
The formal definition of a single linked list is as follows...

Single linked list is a sequence of elements in which every element has link to its next element in the sequence.

In any single linked list, the individual element is called as "Node". Every "Node" contains two fields, data field, and the next field. The data field is used to store actual value of the node and next field is used to store the address of next node in the sequence. The graphical representation of a node in a single linked list is as follows...

Example



Operations on Single Linked List

The following operations are performed on a Single Linked List

- **Insertion**
- **Deletion**
- **Display**

Before we implement actual operations, first we need to set up an empty list. First, perform the following steps before implementing actual operations.

- **Step 1 -** Include all the **header files** which are used in the program.
- **Step 2 -** Declare all the **user defined functions**.
- **Step 3 -** Define a **Node** structure with two members **data** and **next**
- **Step 4 -** Define a Node pointer **'head'** and set it to **NULL**.
- **Step 5 -** Implement the main method by displaying operations menu and make suitable function calls in the main method to perform user selected operation.

Insertion

In a single linked list, the insertion operation can be performed in three ways. They are as follows...

1. Inserting At Beginning of the list
2. Inserting At End of the list
3. Inserting At Specific location in the list

Inserting At Beginning of the list

We can use the following steps to insert a new node at beginning of the single linked list...

- **Step 1 -** Create a **newNode** with given value.
- **Step 2 -** Check whether list is **Empty** (**head** == **NULL**)
- **Step 3 -** If it is **Empty** then, set **newNode→next** = **NULL** and **head** = **newNode**.
- **Step 4 -** If it is **Not Empty** then, set **newNode→next** = **head** and **head** = **newNode**.

Inserting At End of the list

We can use the following steps to insert a new node at end of the single linked list...

- **Step 1 -** Create a **newNode** with given value and **newNode** → **next** as **NULL**.
- **Step 2 -** Check whether list is **Empty** (**head** == **NULL**).
- **Step 3 -** If it is **Empty** then, set **head** = **newNode**.
- **Step 4 -** If it is **Not Empty** then, define a node pointer **temp** and initialize with **head**.
- **Step 5 -** Keep moving the **temp** to its next node until it reaches to the last node in the list (until **temp** → **next** is equal to **NULL**).
- **Step 6 -** Set **temp** → **next** = **newNode**.

Inserting At Specific location in the list (After a Node)

We can use the following steps to insert a new node after a node in the single linked list...

- **Step 1 -** Create a **newNode** with given value.
- **Step 2 -** Check whether list is **Empty** (**head** == **NULL**)
- **Step 3 -** If it is **Empty** then, set **newNode** → **next** = **NULL** and **head** = **newNode**.
- **Step 4 -** If it is **Not Empty** then, define a node pointer **temp** and initialize with **head**.
- **Step 5 -** Keep moving the **temp** to its next node until it reaches to the node after which we want to insert the newNode (until **temp1** → **data** is equal to **location**, here location is the node value after which we want to insert the newNode).
- **Step 6 -** Every time check whether **temp** is reached to last node or not. If it is reached to last node then display **'Given node is not found in the list!!! Insertion not possible!!!'** and terminate the function. Otherwise move the **temp** to next node.
- **Step 7 -** Finally, Set '**newNode** → **next** = **temp** → **next**' and '**temp** → **next** = **newNode**'

Deletion

In a single linked list, the deletion operation can be performed in three ways. They are as follows...

1. Deleting from Beginning of the list
2. Deleting from End of the list
3. Deleting a Specific Node

Deleting from Beginning of the list

We can use the following steps to delete a node from beginning of the single linked list...

- **Step 1 -** Check whether list is **Empty** (**head** == **NULL**)
- **Step 2 -** If it is **Empty** then, display **'List is Empty!!! Deletion is not possible'** and terminate the function.
- **Step 3 -** If it is **Not Empty** then, define a Node pointer **'temp'** and initialize with **head**.
- **Step 4 -** Check whether list is having only one node (**temp** → **next** == **NULL**)
- **Step 5 -** If it is **TRUE** then set **head** = **NULL** and delete **temp** (Setting **Empty** list conditions)
- **Step 6 -** If it is **FALSE** then set **head** = **temp** → **next**, and delete **temp**.

Deleting from End of the list

We can use the following steps to delete a node from end of the single linked list...

- **Step 1 -** Check whether list is **Empty** (**head** == **NULL**)
- **Step 2 -** If it is **Empty** then, display **'List is Empty!!! Deletion is not possible'** and terminate the function.
- **Step 3 -** If it is **Not Empty** then, define two Node pointers **'temp1'** and **'temp2'** and initialize '**temp1**' with **head**.
- **Step 4 -** Check whether list has only one Node (**temp1** → **next** == **NULL**)
- **Step 5 -** If it is **TRUE**. Then, set **head** = **NULL** and delete **temp1**. And terminate the function. (Setting **Empty** list condition)
- **Step 6 -** If it is **FALSE**. Then, set '**temp2 = temp1** ' and move **temp1** to its next node. Repeat the same until it reaches to the last node in the list. (until **temp1** → **next** == **NULL**)
- **Step 7 -** Finally, Set **temp2** → **next** = **NULL** and delete **temp1**.

Deleting a Specific Node from the list

We can use the following steps to delete a specific node from the single linked list...

- **Step 1 -** Check whether list is **Empty** (**head** == **NULL**)
- **Step 2 -** If it is **Empty** then, display **'List is Empty!!! Deletion is not possible'** and terminate the function.
- **Step 3 -** If it is **Not Empty** then, define two Node pointers **'temp1'** and '**temp2**' and initialize '**temp1**' with **head**.
- **Step 4 -** Keep moving the **temp1** until it reaches to the exact node to be deleted or to the last node. And every time set '**temp2 = temp1**' before moving the '**temp1**' to its next node.
- **Step 5 -** If it is reached to the last node then display **'Given node not found in the list! Deletion not possible!!!'**. And terminate the function.
- **Step 6 -** If it is reached to the exact node which we want to delete, then check whether list is having only one node or not
- **Step 7 -** If list has only one node and that is the node to be deleted, then set **head** = **NULL** and delete **temp1** (**free(temp1)**).
- **Step 8 -** If list contains multiple nodes, then check whether **temp1** is the first node in the list (**temp1 == head**).
- **Step 9 -** If **temp1** is the first node then move the **head** to the next node (**head = head** → **next**) and delete **temp1**.
- **Step 10 -** If **temp1** is not first node then check whether it is last node in the list (**temp1** → **next == NULL**).
- **Step 11 -** If **temp1** is last node then set **temp2** → **next** = **NULL** and delete **temp1** (**free(temp1)**).
- **Step 12 -** If **temp1** is not first node and not last node then set **temp2** → **next** = **temp1** → **next** and delete **temp1** (**free(temp1)**).

Displaying a Single Linked List

We can use the following steps to display the elements of a single linked list...

- **Step 1 -** Check whether list is **Empty** (**head** == **NULL**)
- **Step 2 -** If it is **Empty** then, display **'List is Empty!!!'** and terminate the function.
- **Step 3 -** If it is **Not Empty** then, define a Node pointer **'temp'** and initialize with **head**.
- **Step 4 -** Keep displaying **temp** → **data** with an arrow (**--->**) until **temp** reaches to the last node

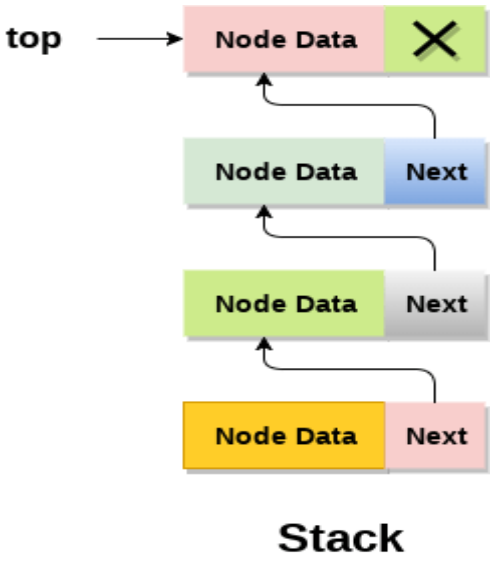
- **Step 5 -** Finally display **temp → data** with arrow pointing to **NULL** (**temp → data ---> NULL**).

Linked stack and Queue:

Linked stack

Instead of using array, we can also use linked list to implement stack. Linked list allocates the memory dynamically. However, time complexity in both the scenario is same for all the operations i.e. push, pop and peek.

In linked list implementation of stack, the nodes are maintained non-contiguously in the memory. Each node contains a pointer to its immediate successor node in the stack. Stack is said to be overflow if the space left in the memory heap is not enough to create a node.



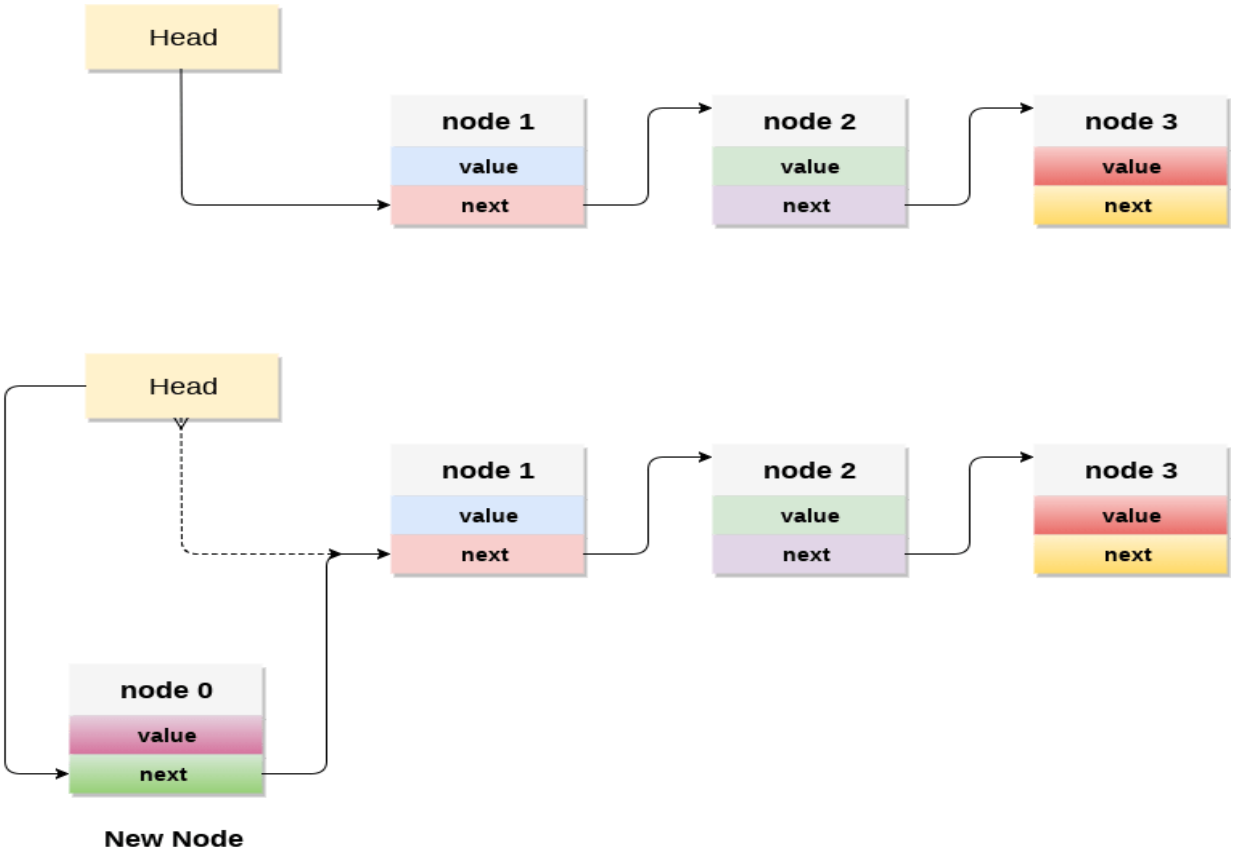
The top most node in the stack always contains null in its address field. Lets discuss the way in which, each operation is performed in linked list implementation of stack.

Adding a node to the stack (Push operation)

Adding a node to the stack is referred to as **push** operation. Pushing an element to a stack in linked list implementation is different from that of an array implementation. In order to push an element onto the stack, the following steps are involved.

1. Create a node first and allocate memory to it.

2. If the list is empty then the item is to be pushed as the start node of the list. This includes assigning value to the data part of the node and assign null to the address part of the node.
3. If there are some nodes in the list already, then we have to add the new element in the beginning of the list (to not violate the property of the stack). For this purpose, assign the address of the starting element to the address field of the new node and make the new node, the starting node of the list.



Deleting a node from the stack (POP operation)

Deleting a node from the top of stack is referred to as **pop** operation. Deleting a node from the linked list implementation of stack is different from that in the array implementation. In order to pop an element from the stack, we need to follow the following steps :

1. **Check for the underflow condition:** The underflow condition occurs when we try to pop from an already empty stack. The stack will be empty if the head pointer of the list points to null.
2. **Adjust the head pointer accordingly:** In stack, the elements are popped only from one end, therefore, the value stored in the head pointer must be deleted and the node must be freed. The next node of the head node now becomes the head node.

Display the nodes (Traversing)

Displaying all the nodes of a stack needs traversing all the nodes of the linked list organized in the form of stack. For this purpose, we need to follow the following steps.

1. Copy the head pointer into a temporary pointer.
2. Move the temporary pointer through all the nodes of the list and print the value field attached to every node.

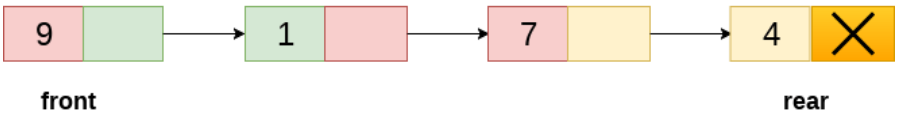
Linked Queue

In a linked queue, each node of the queue consists of two parts i.e. data part and the link part. Each element of the queue points to its immediate next element in the memory.

In the linked queue, there are two pointers maintained in the memory i.e. front pointer and rear pointer. The front pointer contains the address of the starting element of the queue while the rear pointer contains the address of the last element of the queue.

Insertion and deletions are performed at rear and front end respectively. If front and rear both are NULL, it indicates that the queue is empty.

The linked representation of queue is shown in the following figure.



Linked Queue

Operation on Linked Queue

There are two basic operations which can be implemented on the linked queues. The operations are Insertion and Deletion.

The insert operation append the queue by adding an element to the end of the queue. The new element will be the last element of the queue.

Firstly, allocate the memory for the new node ptr by using the following statement.

There can be the two scenario of inserting this new node ptr into the linked queue.

we insert element into an empty queue. In this case, the condition **front = NULL** becomes true. Now, the new element will be added as the only element of the queue and the next pointer of front and rear pointer both, will point to NULL.

Deletion operation removes the element that is first inserted among all the queue elements. Firstly, we need to check either the list is empty or not. The condition front == NULL becomes true if the list is empty, in this case , we simply write underflow on the console and make exit. Otherwise, we will delete the element that is pointed by the pointer front. For this purpose, copy the node pointed by the front pointer into the pointer ptr. Now, shift the front pointer, point to its next node and free the node pointed by the node ptr.

Unit II

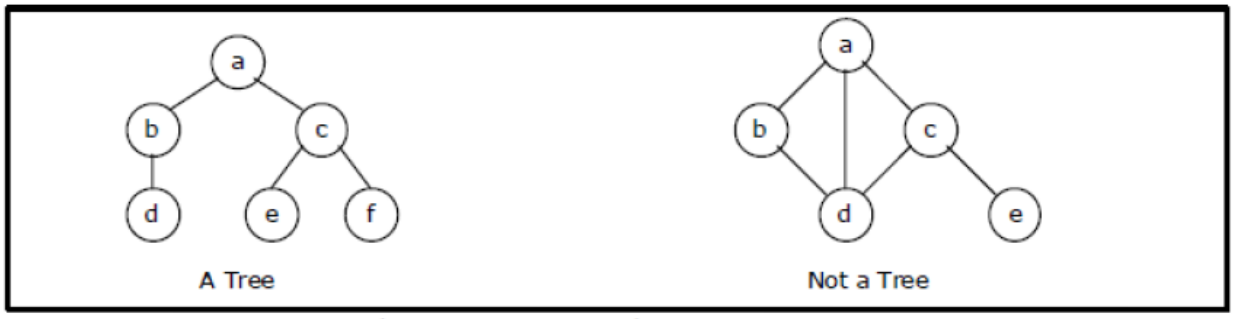
Trees – Binary tree representations – Tree Traversal – Threaded Binary Trees – Binary Tree Representation of Trees – Graphs and Representations – Traversals, Connected Components and Spanning Trees – Shortest Paths and Transitive closure – Activity Networks – Topological Sort and Critical Paths.

Trees: Non-Linear data structure

A data structure is said to be linear if its elements form a sequence or a linear list. Previous
linear data structures that we have studied like an array, stacks, queues and linked lists organize data in linear order. A data structure is said to be non linear if its elements form a hierarchical classification where, data items appear at various levels.

Trees and Graphs are widely used non-linear data structures. Tree and graph structures represent hierarchical relationship between individual data elements. Graphs are nothing but trees with certain restrictions removed.

Trees represent a special case of more general structures known as graphs. In a graph, there is no restrictions on the number of links that can enter or leave a node, and cycles may be present in the graph. The figure shows a tree and a non-tree.



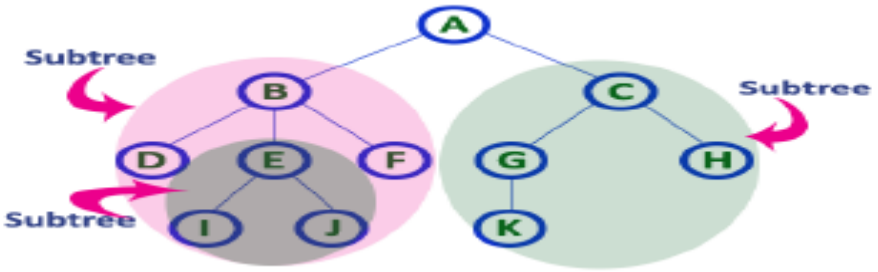
Tree is a popular data structure used in wide range of applications. A tree data structure can be defined as follows...

Tree is a non-linear data structure which organizes data in hierarchical structure and this is a recursive definition.

A tree data structure can also be defined as follows...

A tree is a finite set of one or more nodes such that:

There is a specially designated node called the root. The remaining nodes are partitioned into n>=0 disjoint sets T1, ..., Tn, where each of these sets is a tree. We call T1, ..., Tn are the subtrees of the root.



A tree is hierarchical collection of nodes. One of the nodes, known as the root, is at the top of the hierarchy. Each node can have at most one link coming into it. The node where the link originates is called the parent node. The root node has no parent. The links leaving a node (any number of links are allowed) point to child nodes. Trees are recursive structures. Each child node is itself the root of a subtree. At the bottom of the tree are leaf nodes, which have no children.

Advantages of trees

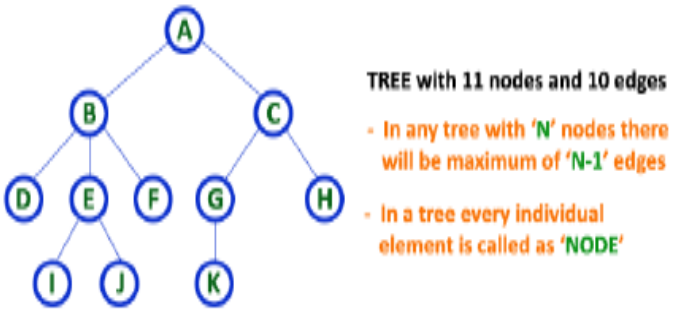
Trees are so useful and frequently used, because they have some very serious advantages:

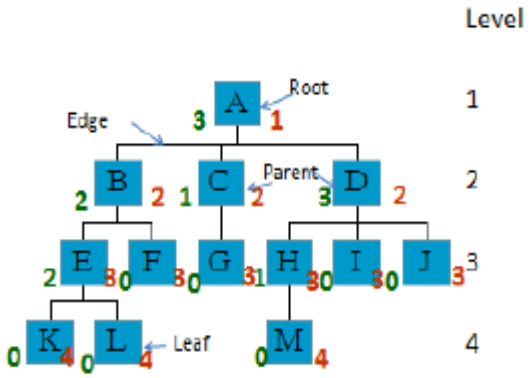
- Trees reflect structural relationships in the data
- Trees are used to represent hierarchies
- Trees provide an efficient insertion and searching
- Trees are very flexible data, allowing to move sub trees around with minimum effort

Introduction Terminology

In a Tree, Every individual element is called as Node. Node in a tree data structure, stores the actual data of that particular element and link to next element in hierarchical structure.

Example





1. Root

In a tree data structure, the first node is called as Root Node. Every tree must have root node. We can say that root node is the origin of tree data structure. In any tree, there must be only one root node. We never have multiple root nodes in a tree. In above tree, **A** is a **Root** node

2. Edge

In a tree data structure, the connecting link between any two nodes is called as EDGE. In a tree with 'N' number of nodes there will be a maximum of 'N-1' number of edges.

3. Parent

In a tree data structure, the node which is predecessor of any node is called as PARENT NODE. In simple words, the node which has branch from it to any other node is called as parent node. Parent node can also be defined as "The node which has child / children". e.g., Parent (A,B,C,D).

4. Child

In a tree data structure, the node which is descendant of any node is called as CHILD Node. In simple words, the node which has a link from its parent node is called as child node. In a tree, any parent node can have any number of child nodes. In a tree, all the nodes except root are child nodes. e.g., Children of D are (H, I,J).

5. Siblings

In a tree data structure, nodes which belong to same Parent are called as SIBLINGS. In simple words, the nodes with same parent are called as Sibling nodes. Ex: Siblings (B,C, D)

6. Leaf

In a tree data structure, the node which does not have a child (or) node with degree zero is called as LEAF Node. In simple words, a leaf is a node with no child. In a tree data structure, the leaf

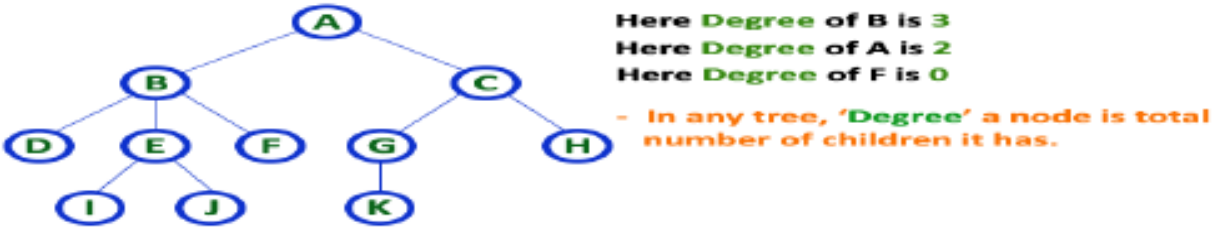
nodes are also called as External Nodes. External node is also a node with no child. In a tree, leaf node is also called as 'Terminal' node. Ex: (K,L,F,G,M,I,J)

7. Internal Nodes

In a tree data structure, the node which has atleast one child is called as INTERNAL Node. In simple words, an internal node is a node with atleast one child. In a tree data structure, nodes other than leaf nodes are called as Internal Nodes. The root node is also said to be Internal Node if the tree has more than one node. Internal nodes are also called as 'Non-Terminal' nodes. Ex:B,C,D,E,H

8. Degree

In a tree data structure, the total number of children of a node (or)number of subtrees of a node is called as DEGREE of that Node. In simple words, the Degree of a node is total number of children it has. The highest degree of a node among all the nodes in a tree is called as 'Degree of Tree'



9. Level

In a tree data structure, the root node is said to be at Level 0 and the children of root node are at Level 1 and the children of the nodes which are at Level 1 will be at Level 2 and so on... In simple words, in a tree each step from top to bottom is called as a Level and the Level count starts with '0' and incremented by one at each level (Step). Some authors start root level with 1.

10. Height

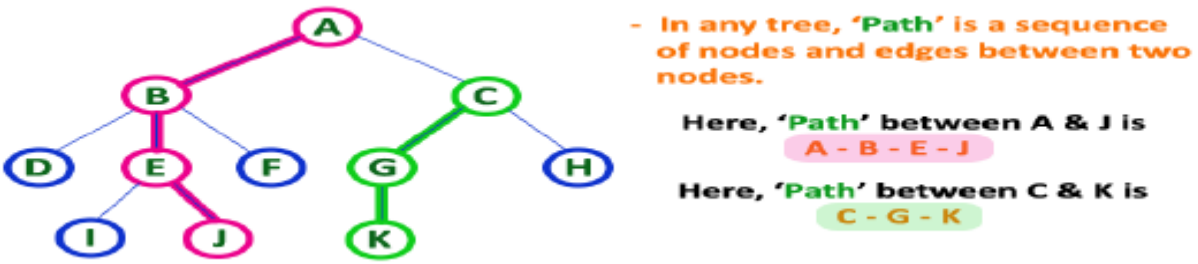
In a tree data structure, the total number of edges from leaf node to a particular node in the longest path is called as HEIGHT of that Node. In a tree, height of the root node is said to be height of the tree. In a tree, height of all leaf nodes is '0'.

11. Depth

In a tree data structure, the total number of edges from root node to a particular node is called as DEPTH of that Node. In a tree, the total number of edges from root node to a leaf node in the longest path is said to be Depth of the tree. In simple words, the highest depth of any leaf node in a tree is said to be depth of that tree. In a tree, depth of the root node is '0'.

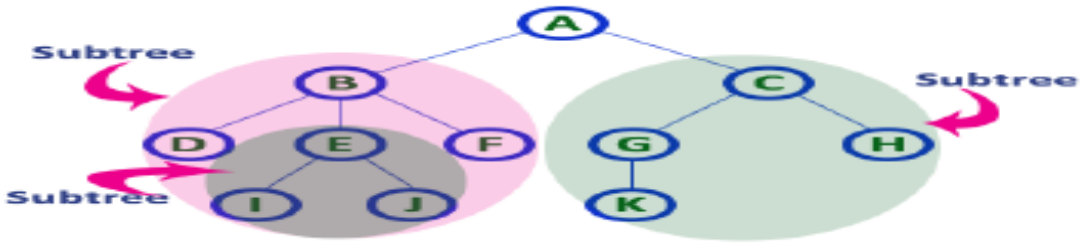
12. Path

In a tree data structure, the sequence of Nodes and Edges from one node to another node is called as PATH between that two Nodes. Length of a Path is total number of nodes in that path. In below example the path A - B - E - J has length 4.



13. Sub Tree

In a tree data structure, each child from a node forms a subtree recursively. Every child node will form a subtree on its parent node



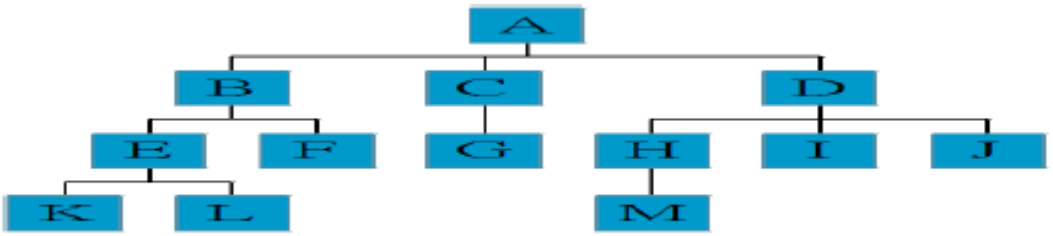
Tree Representations

A tree data structure can be represented in two methods. Those methods are as follows...

1.List Representation

2. Left Child - Right Sibling Representation

Consider the following tree...



1. List Representation

In this representation, we use two types of nodes one for representing the node with data and another for representing only references. We start with a node with data from root node in the tree. Then it is linked to an internal node through a reference node and is linked to any other node directly. This process repeats for all the nodes in the tree.

The above tree example can be represented using List representation as follows...

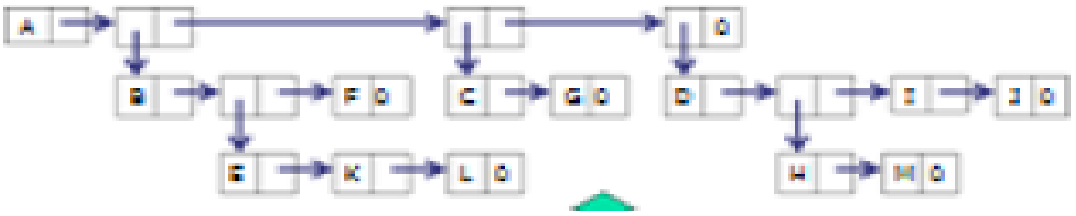


Fig: List representation of above Tree

List Representation

– (A (B (E (K, L), F), C (G), D (H (M), I, J)))

– The root comes first, followed by a list of sub-trees

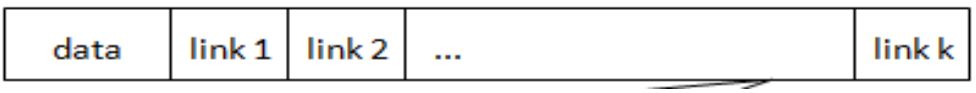


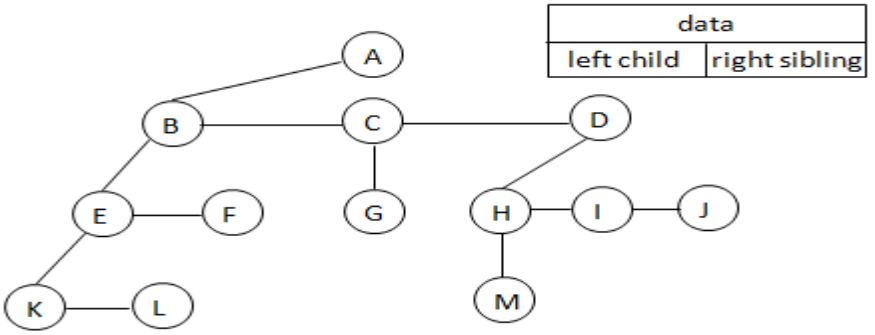
Fig: Possible node structure for a tree of degree k

2. Left Child - Right Sibling Representation

In this representation, we use list with one type of node which consists of three fields namely Data field, Left child reference field and Right sibling reference field. Data field stores the actual

value of a node, left reference field stores the address of the left child and right reference field stores the address of the right sibling node. Graphical representation of that node is as follows...

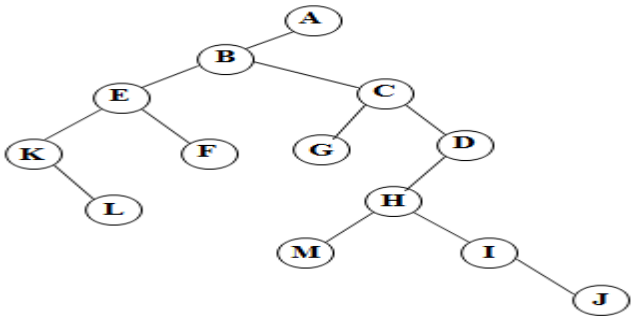
In this representation, every node's data field stores the actual value of that node. If that node has left child, then left reference field stores the address of that left child node otherwise that field stores NULL. If that node has right sibling then right reference field stores the address of right sibling node otherwise that field stores NULL. The above tree example can be represented using Left Child - Right Sibling representation as follows...



Representation as a Degree –Two Tree

To obtain degree-two tree representation of a tree, rotate the right- sibling pointers in the left child-right sibling tree clockwise by 45 degrees. In a degree-two representation, the two children of anode are referred as left and right children.

***Figure 5.6:** Left child-right child tree representation of a tree (p.191)

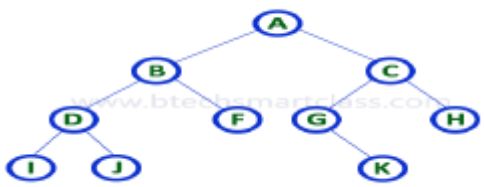


Binary Trees

In a normal tree, every node can have any number of children. Binary tree is a special type of tree data structure in which every node can have a maximum of 2 children. One is known as left child and the other is known as right child.

~A tree in which every node can have a maximum of two children is called as Binary Tree.

~In a binary tree, every node can have either 0 children or 1 child or 2 children but not more than 2 children. Example



There are different types of binary trees and they are...

1. Strictly Binary Tree

In a binary tree, every node can have a maximum of two children. But in strictly binary tree, every node should have exactly two children or none. That means every internal node must have exactly two children. A strictly Binary Tree can be defined as follows...

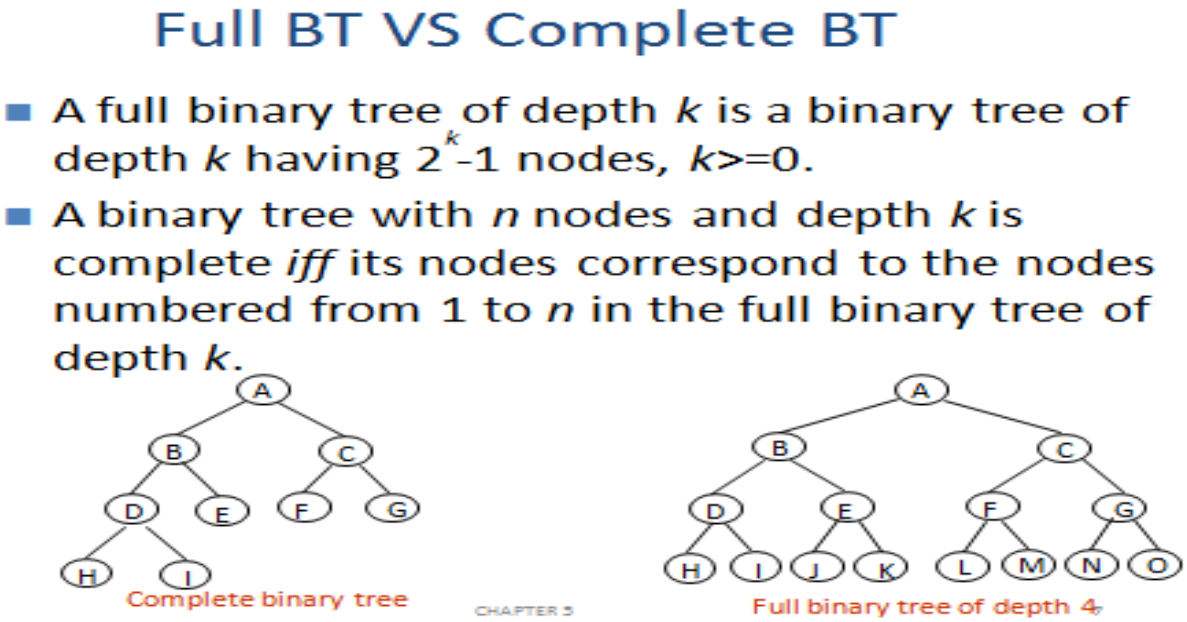
A binary tree in which every node has either two or zero number of children is called Strictly Binary Tree. Strictly binary tree is also called as Full Binary Tree or Proper Binary Tree or 2-Tree

2. Complete Binary Tree

In a binary tree, every node can have a maximum of two children. But in strictly binary tree, every node should have exactly two children or none and in complete binary tree all the nodes must have exactly two children and at every level of complete binary tree there must be 2 level number of nodes. For example at level 2 there must be 2^2 = 4 nodes and at level 3 there must be 2^3 = 8 nodes.

A binary tree in which every internal node has exactly two children and all leaf nodes are at same level is called Complete Binary Tree.

Complete binary tree is also called as Perfect Binary Tree



3. Extended Binary Tree

A binary tree can be converted into Full Binary tree by adding dummy nodes to existing nodes wherever required.

The full binary tree obtained by adding dummy nodes to a binary tree is called as Extended Binary Tree.

Abstract Data Type

Definition: A binary tree is a finite set of nodes that is either empty or consists of a root and two disjoint binary trees called left subtree and right subtree.

ADT contains specification for the binary tree ADT.

Structure Binary_Tree(abbreviated BinTree) is

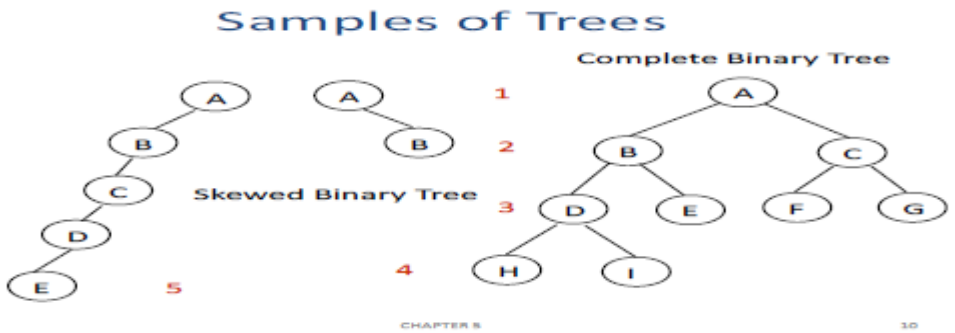
objects: a finite set of nodes either empty or consisting of a root node, left Binary_Tree, and right Binary_Tree.

Functions:

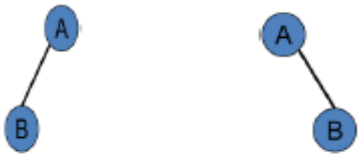
for all bt, bt1, bt2 □ BinTree, item □ element

Bintree Create()::= creates an empty binary tree

Boolean IsEmpty(bt)::= if (bt==empty binary tree) return TRUE else return FALSE
BinTree MakeBT(bt1, item, bt2)::= return a binary tree whose left subtree is bt1, whose right subtree is bt2, and whose root node contains the data item
Bintree Lchild(bt)::= if (IsEmpty(bt)) return error else return the left subtree of bt
element Data(bt)::= if (IsEmpty(bt)) return error else return the data in the root node of bt
Bintree Rchild(bt)::= if (IsEmpty(bt)) return error else return the right subtree of bt



Differences between A Tree and A Binary Tree
• The subtrees of a binary tree are ordered; those of a tree are not ordered.



Above two trees are different when viewed as binary trees. But same when viewed as trees.

Properties of Binary Trees

1.Maximum Number of Nodes in BT

- The maximum number of nodes on level i of a binary tree is 2^{i-1}, i>=1.
- The maximum number of nodes in a binary tree of depth k is $2^{k}-1$, k>=1.

Proof By Induction:
Induction Base: The root is the only node on level i=1.Hence ,the maximum number of nodes on level i=1 is $2^{i-1}=2^0=1$.
Induction Hypothesis: Let I be an arbitrary positive integer greater than 1.Assume that maximum number of nodes on level i-1 is 2^{i-2}.
Induction Step: The maximum number of nodes on level i-1 is 2^{i-2} by the induction hypothesis. Since each node in a binary tree has a maximum degree of 2,the maximum number of nodes on level i is two times the maximum number of nodes on level i-1,or 2^{i-1}.
The maximum number of nodes in a binary tree of depth k is
2.Relation between number of leaf nodes and degree-2 nodes: For any nonempty binary tree, T, if n_0 is the number of leaf nodes and n_2 the number of nodes of degree 2, then $n_0=n_2+1$.
PROOF: Let *n* and *B* denote the total number of nodes and branches in *T*. Let n_0, n_1, n_2 represent the nodes with zero children, single child, and two children respectively.
$B+1=n$ → $B=n_1+2n_2$ ==> $n_1+2n_2+1=n$,
$n_1+2n_2+1=n_0+n_1+n_2$ ==> $n_0=n_2+1$
3. A *full binary tree* of depth *k* is a binary tree of depth *k* having 2 -1 nodes, *k*>=0.
A binary tree with *n* nodes and depth *k* is *complete iff* its nodes correspond to the nodes numbered from 1 to *n* in the full binary tree of depth *k*.

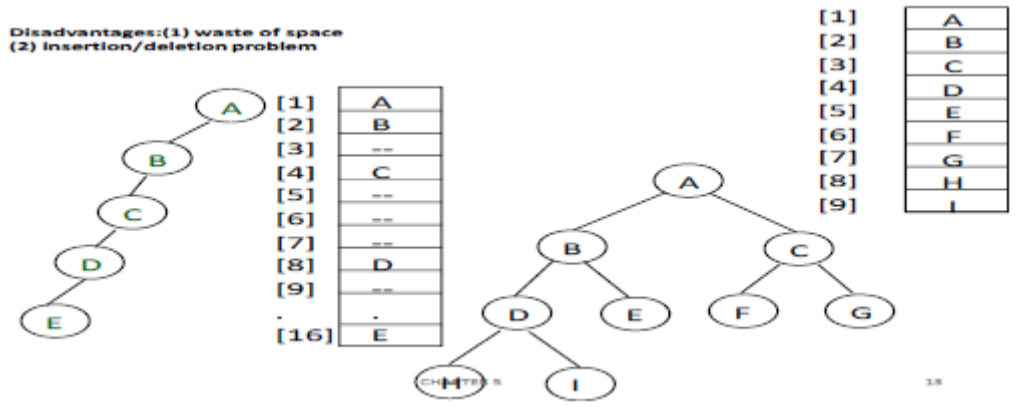
Binary Tree Representation

A binary tree data structure is represented using two methods. Those methods are

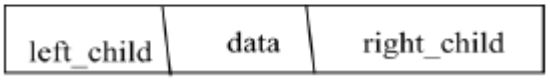
1)Array Representation
2)Linked List Representation

1)Array Representation: In array representation of binary tree, we use a one dimensional array (1-D Array) to represent a binary tree. To represent a binary tree of depth 'n' using array representation, we need one dimensional array with a maximum size of

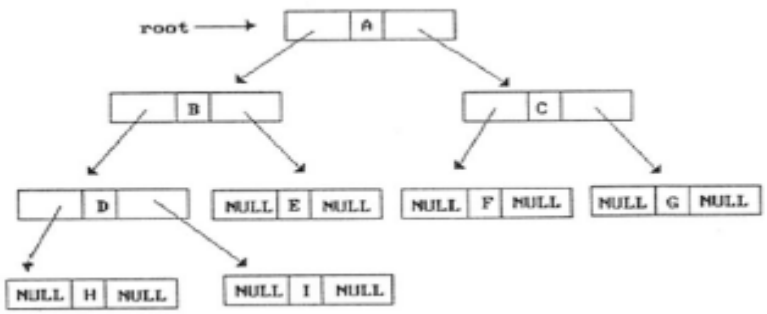
A complete binary tree with *n* nodes (depth = log *n* + 1) is represented sequentially, then for any node with index *i*, 1<=*i*<=*n*, we have: a) *parent*(*i*) is at *i*/2 if *i*!=1. If *i*=1, *i* is at the root and has no parent. b)*left_child*(*i*) ia at 2*i* if 2*i*<=*n*. If 2*i*>n, then *i* has no left child. c) *right_child*(*i*) is at 2*i*+1 if 2*i* +1 <=*n*. If 2*i* +1 >n, then *i* has no right child

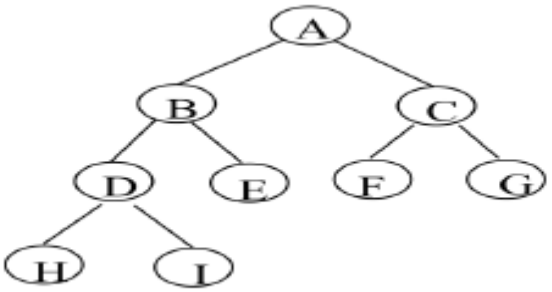


2. Linked Representation : We use linked list to represent a binary tree. In a linked list, every node consists of three fields. First field, for storing left child address, second for storing actual data and third for storing right child address. In this linked list representation, a node has the following structure...




```
typedef struct node *tree_pointer;
typedef struct node
{
int data;
tree_pointer left_child, right_child;
};.
```



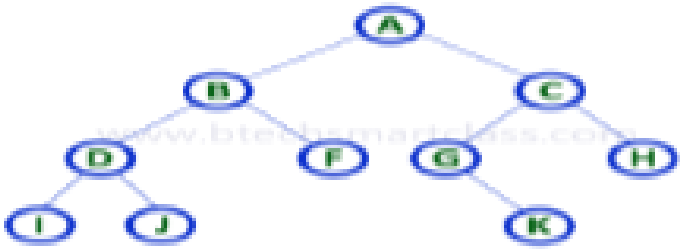


Binary Tree Traversals

When we wanted to display a binary tree, we need to follow some order in which all the nodes of that binary tree must be displayed. In any binary tree displaying order of nodes depends on the

traversal method. Displaying (or) visiting order of nodes in a binary tree is called as Binary Tree Traversal.
There are three types of binary tree traversals.
1)In - Order Traversal 2)Pre - Order Traversal 3)Post - Order Traversal
Binary Tree Traversals

- 1. In - Order Traversal (leftChild - root - rightChild)
  I - D - J - B - F - A - G - K - C – H
- 2. Pre - Order Traversal (root - leftChild - rightChild)
  A - B - D - I - J - F - C - G - K – H
- 3. Post - Order Traversal (leftChild - rightChild - root)
  I - J - D - F - B - K - G - H - C – A



1. In - Order Traversal (leftChild - root - rightChild)
In In-Order traversal, the root node is visited between left child and right child. In this traversal, the left child node is visited first, then the root node is visited and later we go for visiting right child node. This in-order traversal is applicable for every root node of all subtrees in the tree. This is performed recursively for all nodes in the tree. In the above example of binary tree, first we try to visit left child of root node 'A', but A's left child is a root node for left subtree. so we try to visit its (B's) left child 'D' and again D is a root for subtree with nodes D, I and J. So we try to visit its left child 'I' and it is the left most child. So first we visit 'I'then go for its root node 'D' and later we visit D's right child 'J'. With this we have completed the left part of node B. Then visit 'B' and next B's right child 'F' is visited. With this we have completed left part of node A. Then visit root node 'A'. With this we have completed left and root parts of node A. Then we go for right part of the node A. In right of A again there is a subtree with root C. So go for left child of C and again it is a subtree with root G. But G does not have left part so we visit 'G' and then visit G's right child K. With this we have completed the left part of node C. Then visit root node'C' and next visit C's right child 'H' which is the right most child in the tree so we stop the process. That means here we have visited in the order of I - D - J - B - F - A - G - K - C - H using In-Order Traversal.
In-Order Traversal for above example of binary tree is
I - D - J - B - F - A - G - K - C – H

Algorithm
Until all nodes are traversed −
Step 1 − Recursively traverse left subtree.
Step 2 − Visit root node.
Step 3 − Recursively traverse right subtree.
void inorder(tree_pointer ptr) **/* inorder tree traversal */ Recursive**
{
if (ptr) {
inorder(ptr->left_child);
printf(“%d”, ptr->data);
indorder(ptr->right_child);
}
}

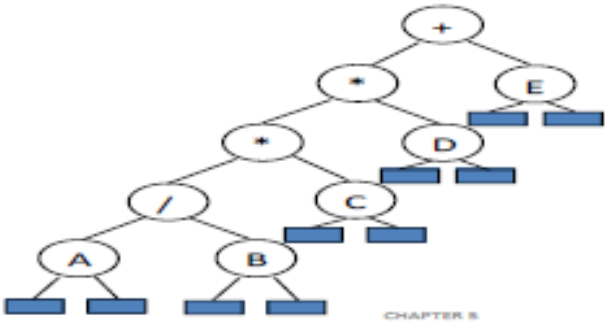
2. Pre - Order Traversal (root - leftChild - rightChild)
In Pre-Order traversal, the root node is visited before left child and right child nodes. In this traversal, the root node is visited first, then its left child and later its right child. This pre-order traversal is applicable for every root node of all subtrees in the tree. In the above example of binary tree, first we visit root node 'A' then visit its left child 'B' which is a root for D and F. So we visit B's left child 'D' and again D is a root for I and J. So we visit D's left child'I' which is the left most child. So next we go for visiting D's right child 'J'. With this we have completed root, left and right parts of node D and root, left parts of node B. Next visit B's right child'F'. With this we have completed root and left parts of node A. So we go for A's right child 'C' which is a root node for G and H. After visiting C, we go for its left child 'G' which is a root for node K. So next we visit left of G, but it does not have left child so we go for G's right child 'K'. With this we have completed node C's root and left parts. Next visit C's right child 'H' which is the right most child in the tree. So we stop the process. That means here we have visited in the order of A-B-D-I-J-F-C-G-K-H using Pre-Order Traversal.

Algorithm
Until all nodes are traversed −
Step 1 − Visit root node.
Step 2 − Recursively traverse left subtree.
Step 3 − Recursively traverse right subtree.
void preorder(tree_pointer ptr) **/* preorder tree traversal */ Recursive**
{
if (ptr) {
printf(“%d”, ptr->data);
preorder(ptr->left_child);
preorder(ptr->right_child);
}
}

3. Post - Order Traversal (leftChild - rightChild - root)
In Post-Order traversal, the root node is visited after left child and right child. In this traversal, left child node is visited first, then its right child and then its root node. This is recursively performed until the right most node is visited. Here we have visited in the order of I - J - D - F - B - K - G - H - C - A using Post-Order Traversal.

Algorithm
Until all nodes are traversed −
Step 1 − Recursively traverse left subtree.
Step 2 − Recursively traverse right subtree.
Step 3 − Visit root node.
void postorder(tree_pointer ptr) **/* postorder tree traversal */ Recursive**
{
if (ptr) {
postorder(ptr->left_child);
postorder(ptr->right_child);
printf(“%d”, ptr->data);
}
}

Arithmetic Expression Using BT



inorder traversal
A / B * C * D + E
infix expression
preorder traversal
+ * * / A B C D E
prefix expression
postorder traversal
A B / C * D * E +
postfix expression
level order traversal
+ * E * D / C A B

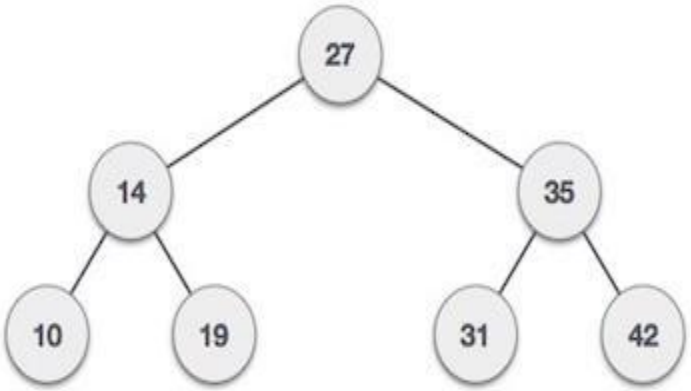
Trace Operations of Inorder Traversal

Call of inorder	Value in root	Action	Call of inorder	Value in root	Action
1	+		11	C	
2	*		12	NULL	
3	*		11	C	printf
4	/		13	NULL	
5	A		2	*	printf
6	NULL		14	D	
5	A	printf	15	NULL	
7	NULL		14	D	printf
4	/	printf	16	NULL	
8	B		1	+	printf
9	NULL		17	E	
8	B	printf	18	NULL	
10	NULL		17	E	printf
3	*	printf	19	NULL	

Binary Search Trees

Binary Search Tree Representation

Binary Search tree exhibits a special behavior. A node's left child must have value less than its parent's value and node's right child must have value greater than it's parent value.



We're going to implement tree using node object and connecting them through references.

Definition: A binary search tree (BST) is a binary tree. It may be empty. If it is not empty,then all nodes follows the below mentioned properties −

- Every element has a unique key.
- The keys in a nonempty left subtree (right subtree) are smaller (larger) than the key in the root of subtree.
- The keys in a nonempty right subtree larger than the key in the root of subtree.
- The left and right subtrees are also binary search trees.

left sub-tree and right sub-tree and can be defined as −

left_subtree (keys) ≤ node (key) ≤ right_subtree (keys)

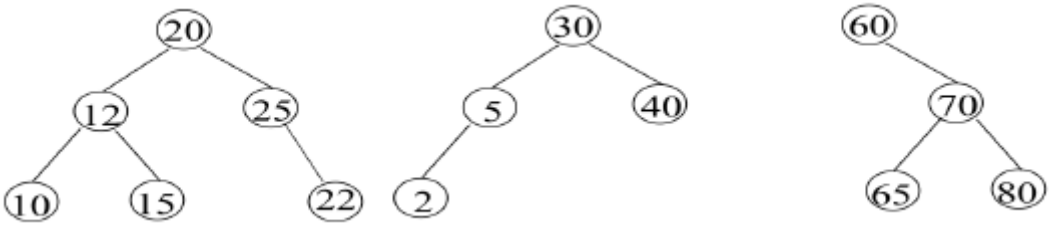
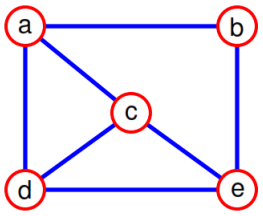


Fig: Example Binary Search Trees

Graph

A **graph G** = (**V**,**E**) is composed of:
V: set of *vertices*
E: set of *edges* connecting the *vertices* in **V**
• An **edge e** = (u,v) is a pair of vertices
Example:



V= {a,b,c,d,e}
E={(a,b),(a,c),(a,d),(b,e),(c,d),(c,e),(d,e)}

Graph Terminology

Undirected Graph:
An undirected graph is one in which the pair of vertices in a edge is unordered, (v0, v1) = (v1,v0)
Directed Graph:
A directed graph is one in which each edge is a directed pair of vertices, <v0, v1> != <v1,v0>

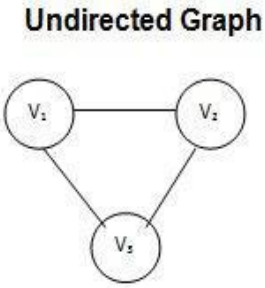


Figure 1: An Undirected Graph

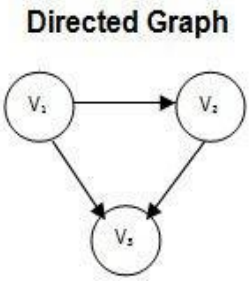
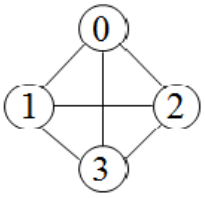


Figure 2: A Directed Graph

Complete Graph:
A complete graph is a graph that has the maximum number of edges for undirected graph with n vertices, the maximum number of edges is n(n-1)/2 for directed graph with n vertices, the maximum number of edges is n(n-1)



G_1

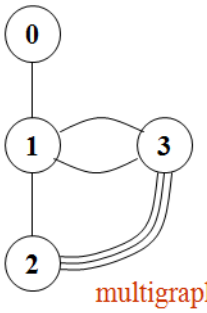
complete graph

$V(G_1)=\{0,1,2,3\}$
$V(G_2)=\{0,1,2,3,4,5,6\}$
$V(G_3)=\{0,1,2\}$

Adjacent and Incident:
If (v0, v1) is an edge in an undirected graph,
– v0 and v1 are adjacent
– The edge (v0, v1) is incident on vertices v0 and v1

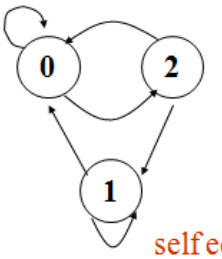
If <v0, v1> is an edge in a directed graph
– v0 is adjacent to v1, and v1 is adjacent from v0
– The edge <v0, v1> is incident on v0 and v1

Multigraph:
In a multigraph, there can be more than one edge from vertex P to vertex Q. In a simple graph there is at most one.



multigraph:
multiple occurrences
of the same edge

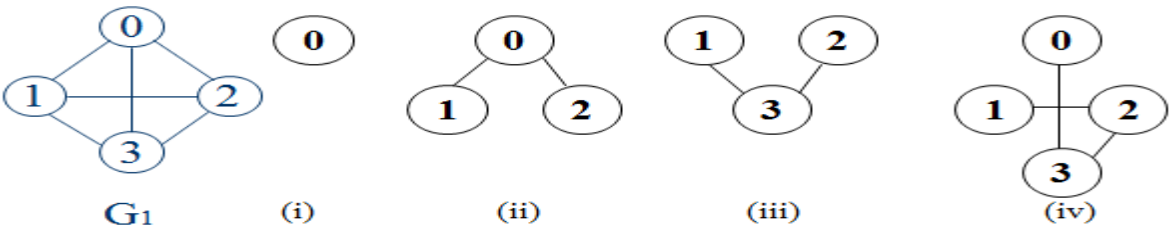
Graph with self edge or graph with feedback loops:
A self loop is an edge that connects a vertex to itself. In some graph it makes sense to allow self-loops; in some it doesn't.



self edge

Subgraph:

A subgraph of G is a graph G' such that V(G') is a subset of V(G) and E(G') is a subset of E(G)

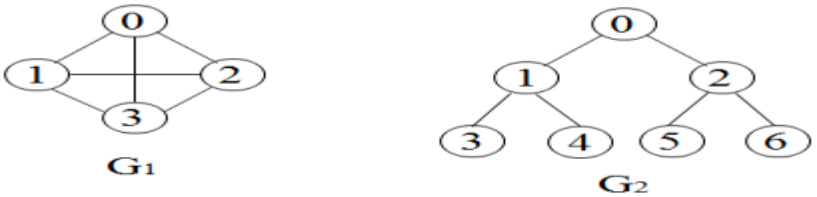


Some of the subgraph of G_1

Path:
A path from vertex vp to vertex vq in a graph G, is a sequence of vertices, vp, vi1, vi2, ..., vin, vq, such that (vp, vi1), (vi1, vi2), ..., (vin, vq) are edges in an undirected graph
The length of a path is the number of edges on it.

Simple Path and Style:
A simple path is a path in which all vertices, except possibly the first and the last, are distinct.
A cycle is a simple path in which the first and the last vertices are the same
In an undirected graph G, two vertices, v0 and v1, are connected if there is a path in G from v0 to v1.
An undirected graph is connected if, for every pair of distinct vertices vi, vj, there is a path from vi to vj

connected

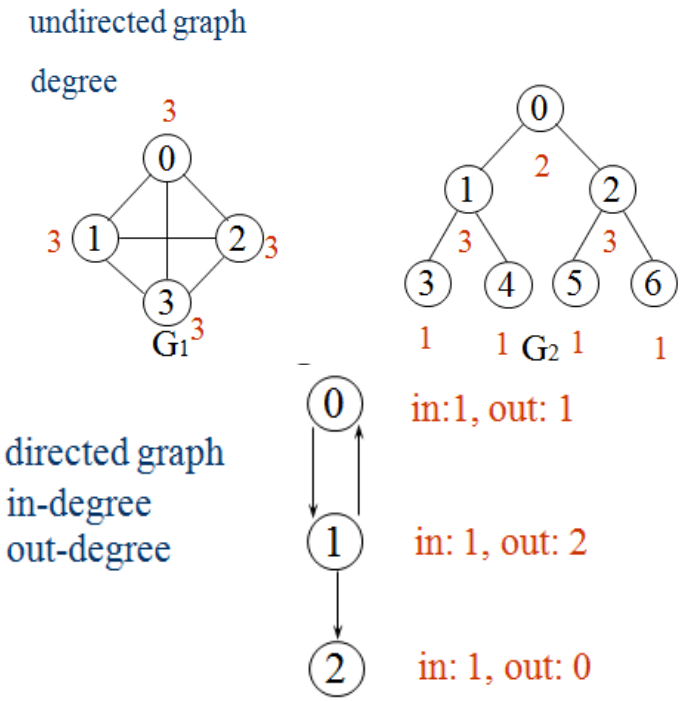


tree (acyclic graph)

Degree
The degree of a vertex is the number of edges incident to that vertex
For directed graph,
– the ***in-degree*** of a vertex *v* is the number of edges that have *v* as the head
– the ***out-degree*** of a vertex *v* is the number of edges that have *v* as the tail
– if *di* is the degree of a vertex *i* in a graph *G* with *n* vertices and *e* edges, the number of edges is

$$e = (\sum_{0}^{n-1} d_i) / 2$$

Example:



ADT for Graph

Graph ADT is

Data structures: a nonempty set of vertices and a set of undirected edges, where each edge is a pair of vertices

Functions: for all *graph* □ *Graph*, *v*, *v*1 and *v*2 □ *Vertices*

- *Graph* Create()::=return an empty graph
- *Graph* InsertVertex(*graph*, *v*)::= return a graph with *v* inserted. *V* has no incident edge.
- *Graph* InsertEdge(*graph*, *v1*,*v2*)::= return a graph with new edge between *v1* and *v2*
- *Graph* DeleteVertex(*graph*, *v*)::= return a graph in which *v* and all edges incident to it are removed
- *Graph* DeleteEdge(*graph*, *v1*, *v2*)::=return a graph in which the edge (*v1*, *v2*) is removed
- *Boolean* IsEmpty(*graph*)::= if (*graph*==*empty graph*) return TRUE else return FALSE
- *List* Adjacent(*graph*,*v*)::= return a list of all vertices that are adjacent to *v*

Graph Representations

Graph can be represented in the following ways:

a) Adjacency Matrix

b) Adjacency Lists

c) Adjacency Multilists

a) Adjacency Matrix

Let G=(V,E) be a graph with n vertices.

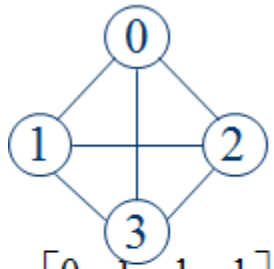
The adjacency matrix of G is a two-dimensional by array, say adj_mat.

If the edge (vi, vj) is in E(G), adj_mat[i][j]=1

If there is no such edge in E(G), adj_mat[i][j]=0

The adjacency matrix for an undirected graph is symmetric; the adjacency matrix for a digraph need not be symmetric

Examples for Adjacency Matrix:

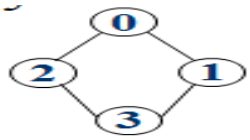


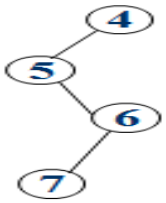
$$\begin{bmatrix} 0 & 1 & 1 & 1 \\ 1 & 0 & 1 & 1 \\ 1 & 1 & 0 & 1 \\ 1 & 1 & 1 & 0 \end{bmatrix}$$

G_1

$$\begin{bmatrix} 0 & 1 & 0 \\ 1 & 0 & 1 \\ 0 & 0 & 0 \end{bmatrix}$$

G_2





$$\begin{bmatrix} 0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \end{bmatrix}$$

G_4

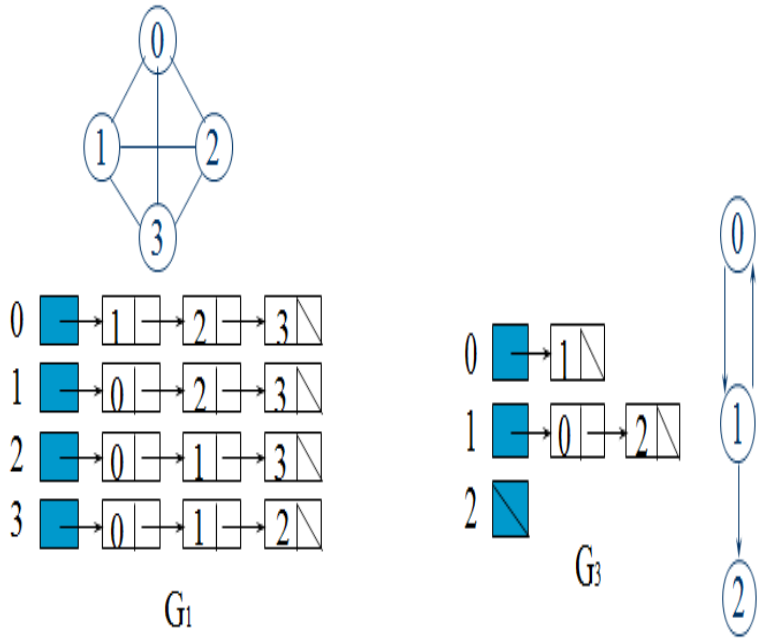
Merits of Adjacency Matrix

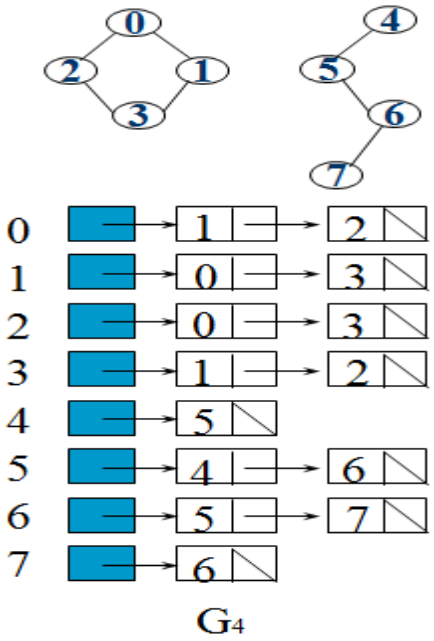
From the adjacency matrix, to determine the connection of vertices is easy
The degree of a vertex is
For a digraph, the row sum is the out_degree, while the column sum is the in_degree

$$ind(vi) = \sum_{j=0}^{n-1} A[j,i] \qquad outd(vi) = \sum_{j=0}^{n-1} A[i,j]$$

b) Adjacency Lists

Each row in adjacency matrix is represented as an adjacency list.

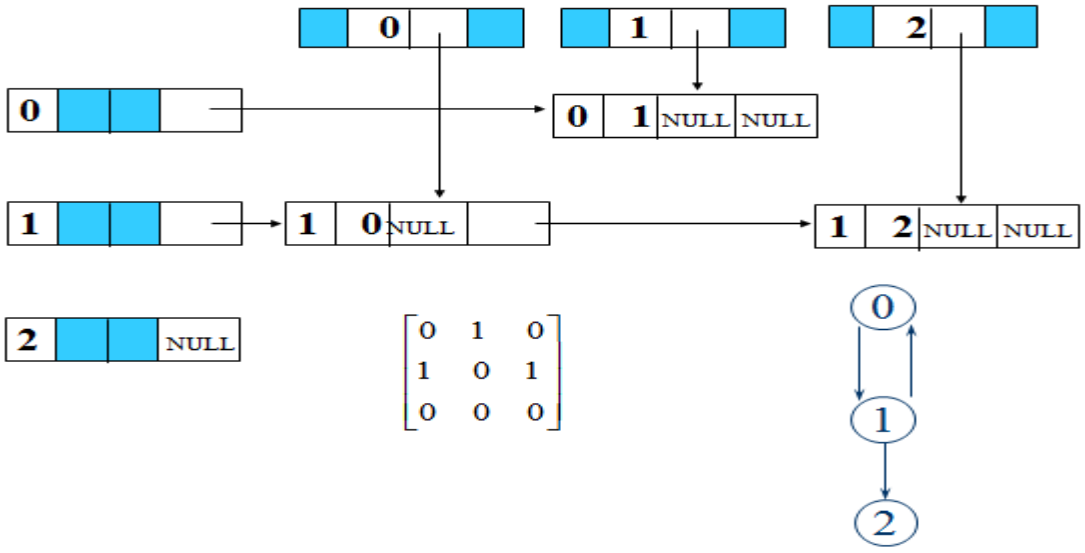




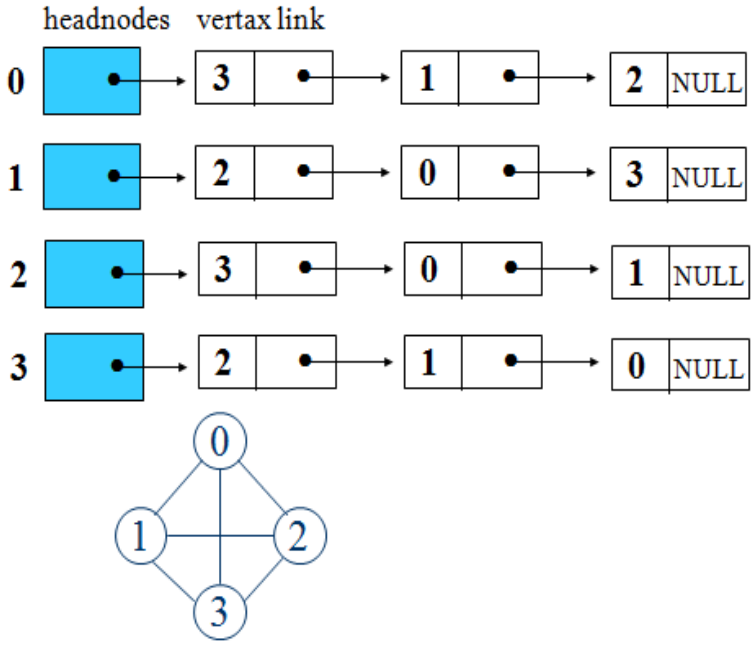
Interesting Operations

- degree of a vertex in an undirected graph
  # of nodes in adjacency list
- # of edges in a graph
  determined in O(n+e)
- out-degree of a vertex in a directed graph
  # of nodes in its adjacency list
- in-degree of a vertex in a directed graph
  traverse the whole data structure

Orthogonal representation for graph G3



Order is of no significance.

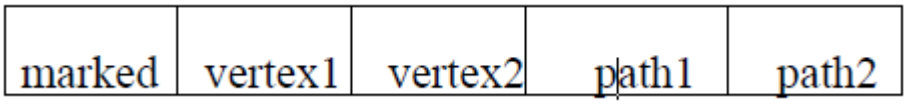


c) Adjacency Multilists

An edge in an undirected graph is represented by two nodes in adjacency list representation.

Adjacency Multilists

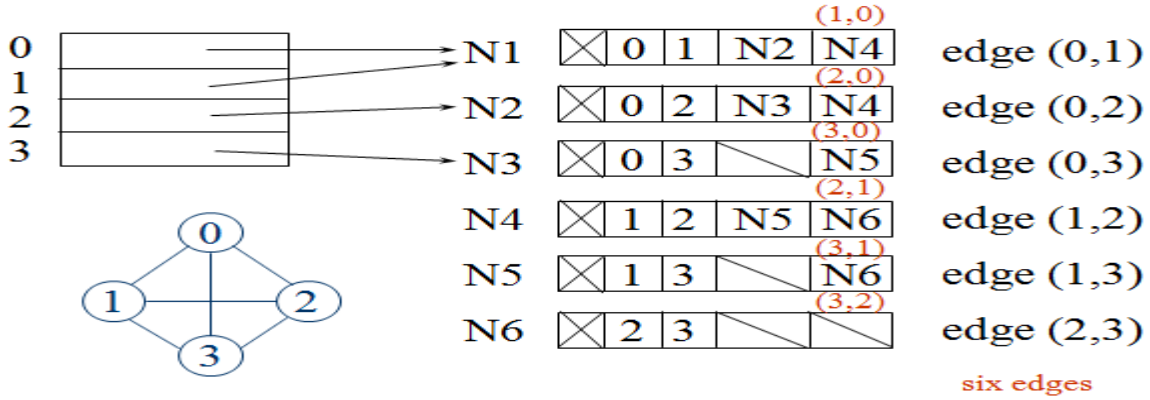
– lists in which nodes may be shared among several lists. (an edge is shared by two different paths)



Example for Adjacency Multlists

Lists: vertex 0: M1->M2->M3, vertex 1: M1->M4->M5

vertex 2: M2->M4->M6, vertex 3: M3->M5->M6



Some Graph Operations

The following are some graph operations:

a) Traversal Given G=(V,E) and vertex v, find all w□V, such that w connects v.

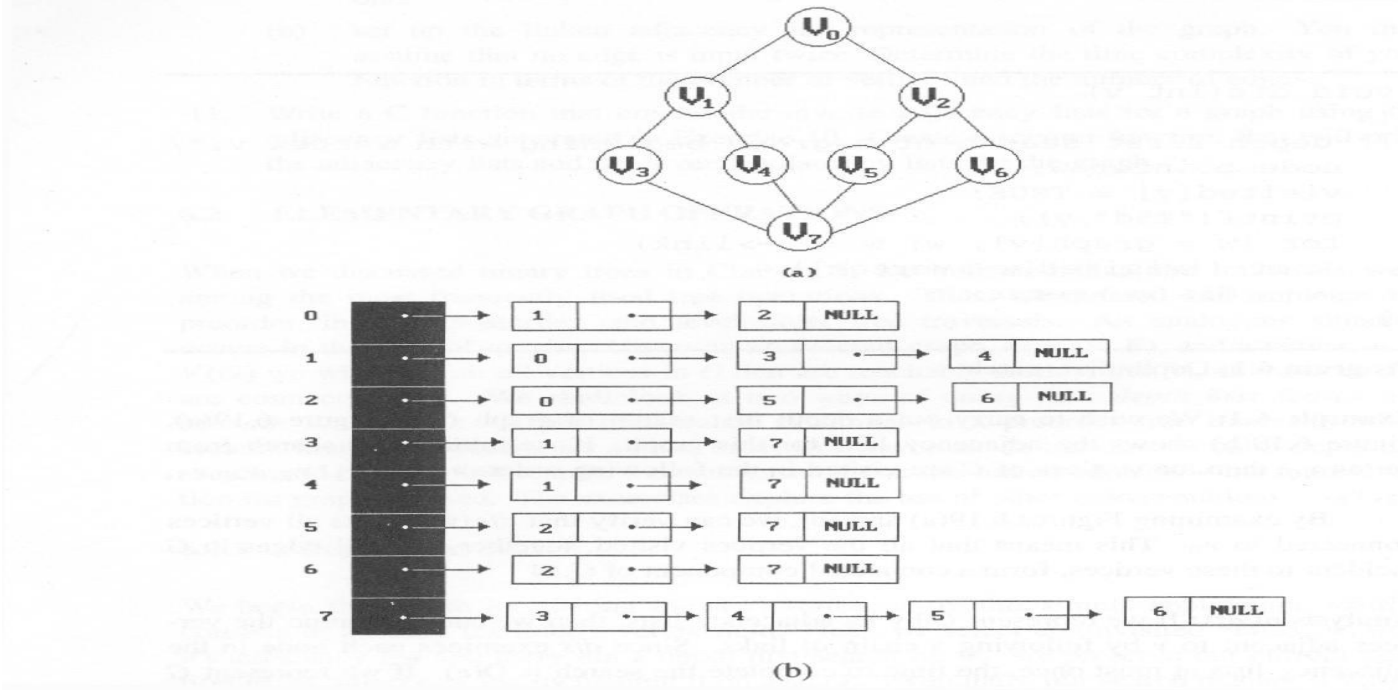
– Depth First Search (DFS) preorder tree traversal

– Breadth First Search (BFS) level order tree traversal

b) Spanning Trees

c) Connected Components

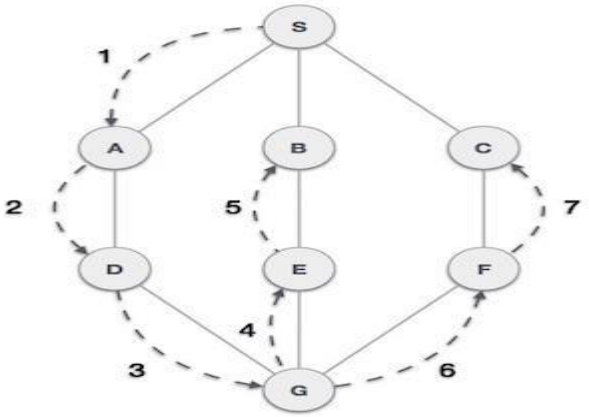
Graph *G* and its adjacency lists



depth first search: v0, v1, v3, v7, v4, v5, v2, v6
breadth first search: v0, v1, v2, v3, v4, v5, v6, v7

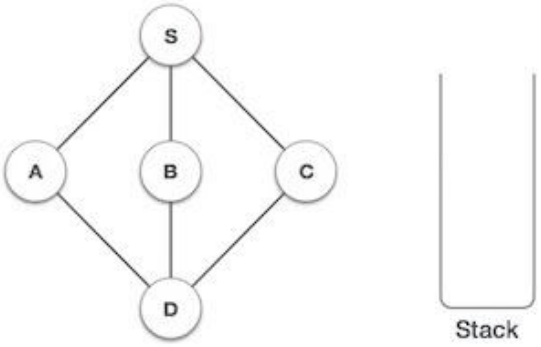
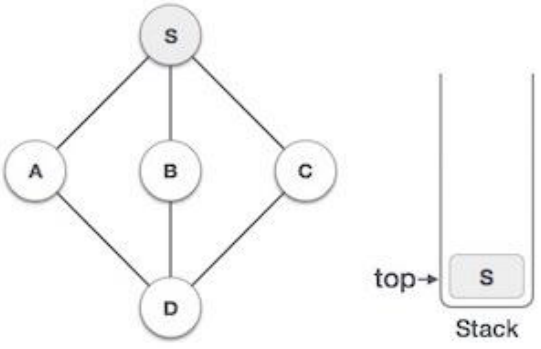
Depth First Search

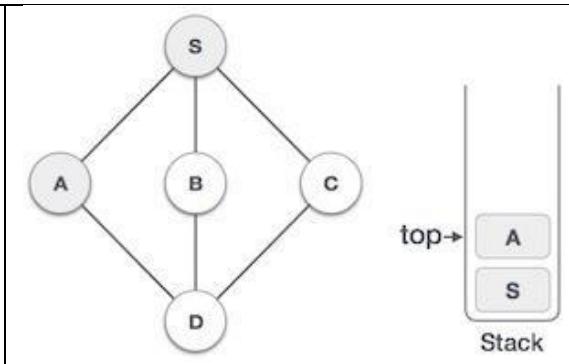
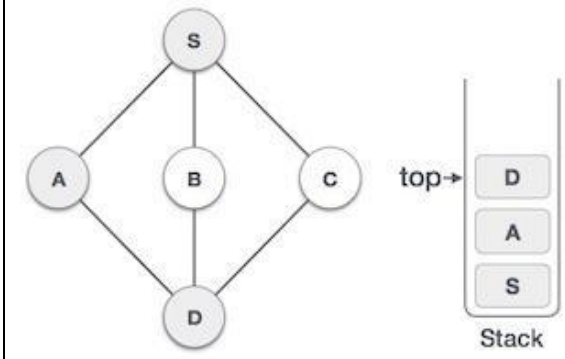
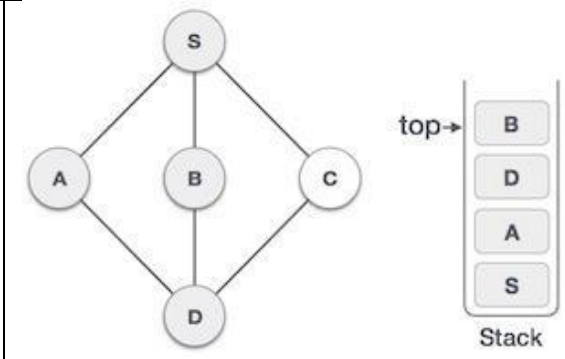
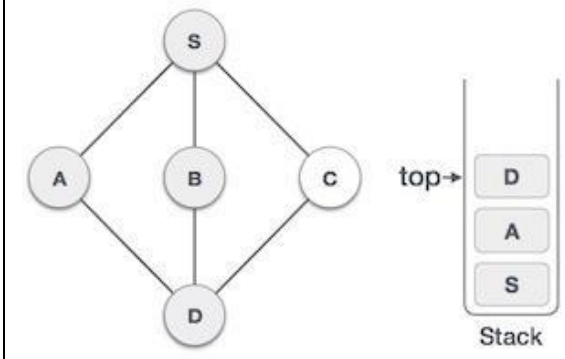
Depth First Search (DFS) algorithm traverses a graph in a depthward motion and uses a stack to remember to get the next vertex to start a search, when a dead end occurs in any iteration.

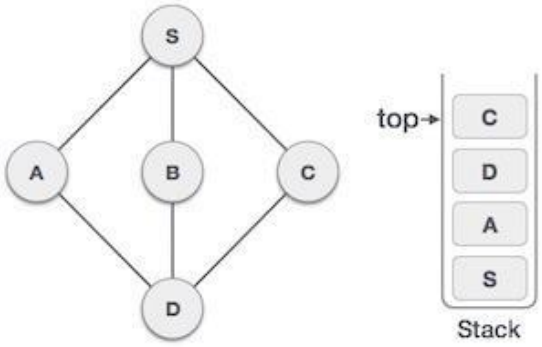


As in the example given above, DFS algorithm traverses from A to B to C to D first then to E, then to F and lastly to G. It employs the following rules.

- **Rule 1** − Visit the adjacent unvisited vertex. Mark it as visited. Display it. Push it in a stack.
- **Rule 2** − If no adjacent vertex is found, pop up a vertex from the stack. (It will pop up all the vertices from the stack, which do not have adjacent vertices.)
- **Rule 3** − Repeat Rule 1 and Rule 2 until the stack is empty.

Step	Traversal	Description
1		Initialize the stack.
2		Mark **S** as visited and put it onto the stack. Explore any unvisited adjacent node from **S**. We have three nodes and we can pick any of them. For this example, we shall take the node in an alphabetical order.

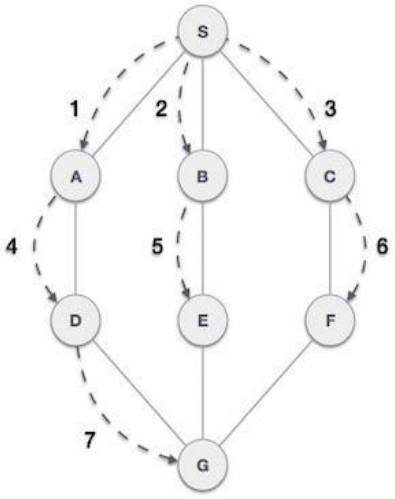
3		Mark **A** as visited and put it onto the stack. Explore any unvisited adjacent node from A. Both **S** and **D** are adjacent to **A** but we are concerned for unvisited nodes only.
4		Visit **D** and mark it as visited and put onto the stack. Here, we have **B** and **C** nodes, which are adjacent to **D** and both are unvisited. However, we shall again choose in an alphabetical order.
5		We choose **B**, mark it as visited and put onto the stack. Here **B** does not have any unvisited adjacent node. So, we pop **B** from the stack.
6		We check the stack top for return to the previous node and check if it has any unvisited nodes. Here, we find **D** to be on the top of the stack.

7		Only unvisited adjacent node is from **D** is **C** now. So we visit **C**, mark it as visited and put it onto the stack.

As **C** does not have any unvisited adjacent node so we keep popping the stack until we find a node that has an unvisited adjacent node. In this case, there's none and we keep popping until the stack is empty.

Breadth First Search

Breadth First Search (BFS) algorithm traverses a graph in a breadthward motion and uses a queue to remember to get the next vertex to start a search, when a dead end occurs in any iteration.



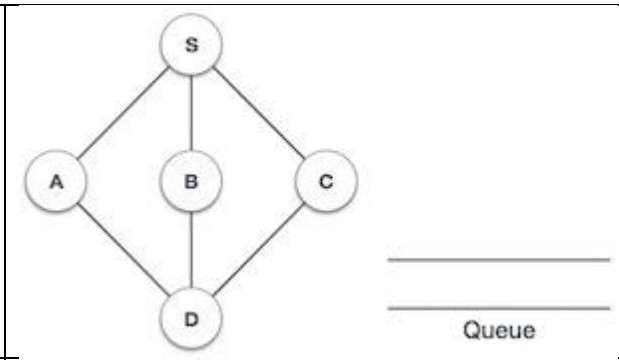
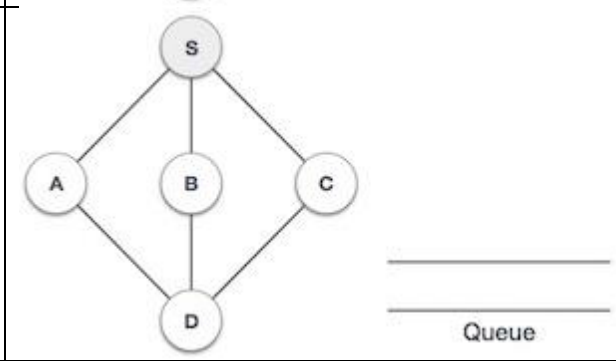
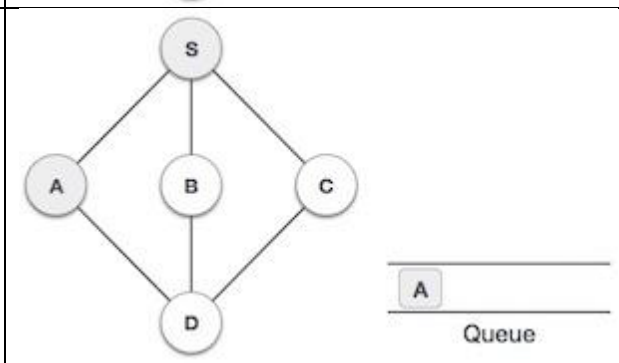
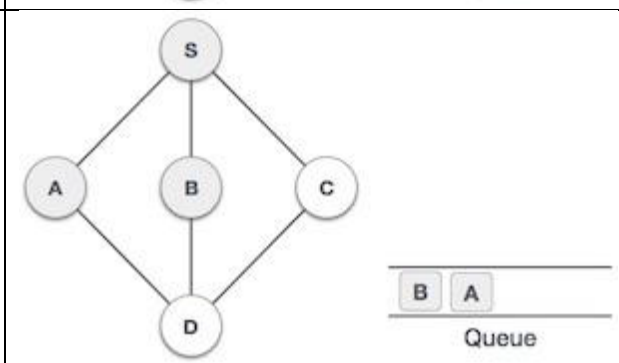
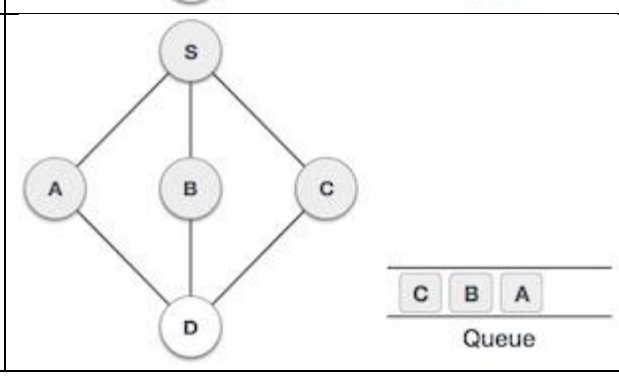
As in the example given above, BFS algorithm traverses from A to B to E to F first then to C and G lastly to D. It employs the following rules.

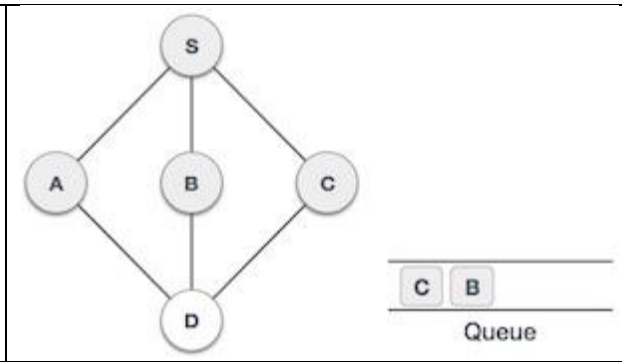
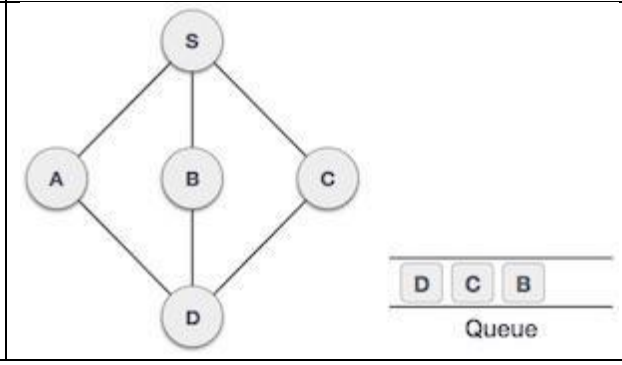
- **Rule 1** − Visit the adjacent unvisited vertex. Mark it as visited. Display it. Insert it in a queue.

- **Rule 2** − If no adjacent vertex is found, remove the first vertex from the queue.

- **Rule 3** − Repeat Rule 1 and Rule 2 until the queue is empty.

Step	Traversal	Description

1		Initialize the queue.
2		We start from visiting **S** (starting node), and mark it as visited.
3		We then see an unvisited adjacent node from **S**. In this example, we have three nodes but alphabetically we choose **A**, mark it as visited and enqueue it.
4		Next, the unvisited adjacent node from **S** is **B**. We mark it as visited and enqueue it.
5		Next, the unvisited adjacent node from **S** is **C**. We mark it as visited and enqueue it.

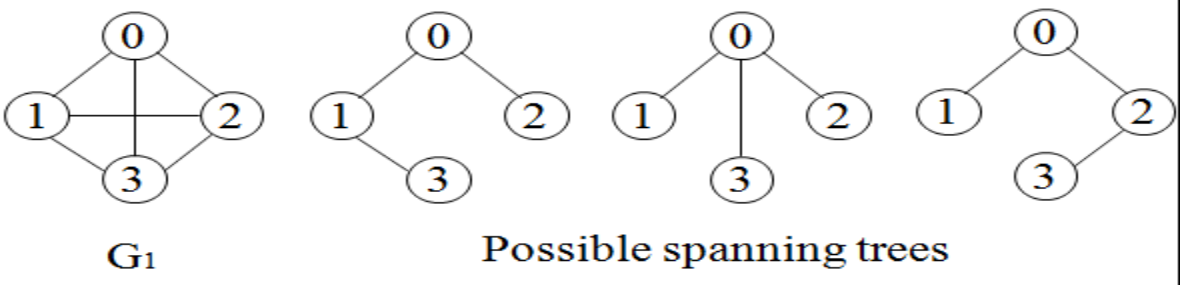
6		Now, **S** is left with no unvisited adjacent nodes. So, we dequeue and find **A**.
7		From **A** we have **D** as unvisited adjacent node. We mark it as visited and enqueue it.

At this stage, we are left with no unmarked (unvisited) nodes. But as per the algorithm we keep on dequeuing in order to get all unvisited nodes. When the queue gets emptied, the program is over.

Spanning Trees

When graph G is connected, a depth first or breadth first search starting at any vertex will visit all vertices in G
A spanning tree is any tree that consists solely of edges in G and that includes all the vertices
E(G): T (tree edges) + N (nontree edges) where T: set of edges used during search
N: set of remaining edges

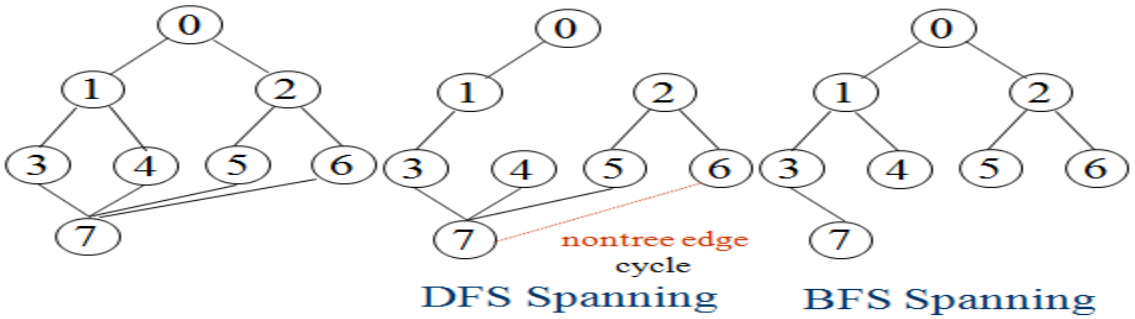
Examples of Spanning Tree



Either dfs or bfs can be used to create a spanning tree
– When dfs is used, the resulting spanning tree is known as a depth first spanning tree
– When bfs is used, the resulting spanning tree is known as a breadth first spanning tree

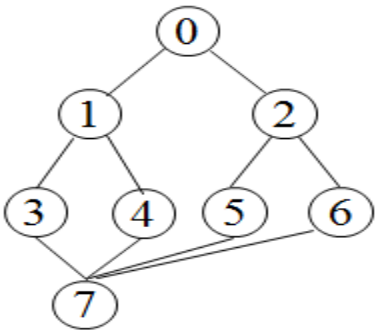
While adding a nontree edge into any spanning tree, this will create a cycle

DFS VS BFS Spanning Tree

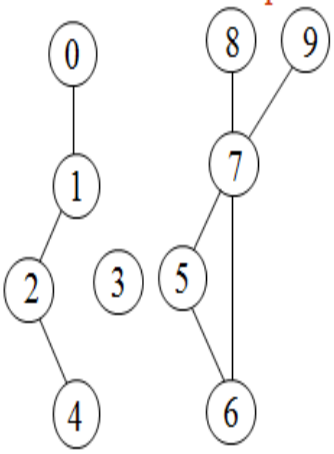


A spanning tree is a minimal subgraph, G', of G such that V(G')=V(G) and G' is connected.
Any connected graph with n vertices must have at least n-1 edges.
A biconnected graph is a connected graph that hasno articulation points.

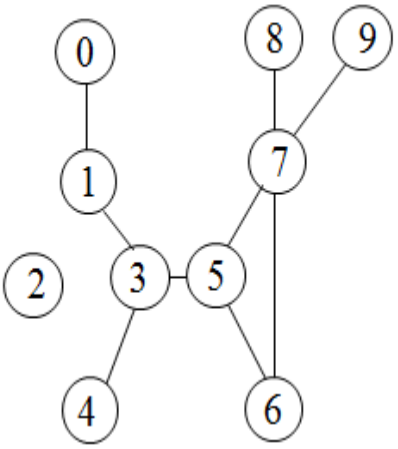
biconnected graph

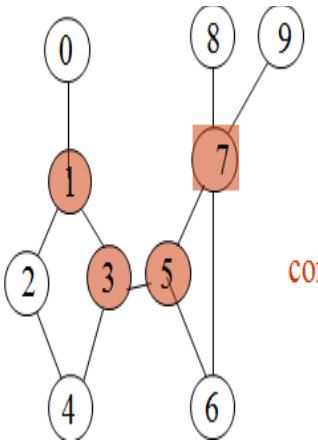


two connected components



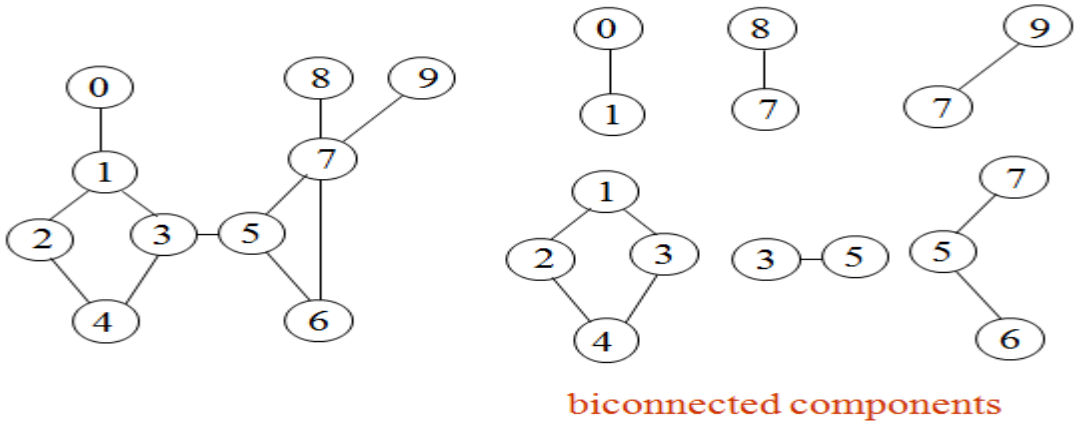
one connected graph





connected graph

biconnected component: a maximal connected subgraph H (no subgraph that is both biconnected and properly contains H).



Minimum Cost Spanning Tree

 The cost of a spanning tree of a weighted undirected graph is the sum of the costs of the edges in the spanning tree

 A minimum cost spanning tree is a spanning tree of least cost

 Three different algorithms can be used

– Kruskal
– Prim
– Sollin

Kruskal's Algorithm

Build a minimum cost spanning tree T by adding edges to T one at a time
Select the edges for inclusion in T in nondecreasing order of the cost
An edge is added to T if it does not form a cycle
Since G is connected and has n > 0 vertices, exactly n-1 edges will be selected

Kruskal's algorithm

1. Sort all the edges in non-decreasing order of their weight.
2. Pick the smallest edge. Check if it forms a cycle with the spanning tree formed so far. If cycle is not formed, include this edge. Else, discard it.
3. Repeat step#2 until there are (V-1) edges in the spanning tree.

Psuedocode for Kruskal's Algorithm

```
Kruskal(G, V, E)
{
T= {};
while(T contains less than n-1 edges && E is not empty)
{
choose a least cost edge (v,w) from E;
delete (v,w) from E;
if ((v,w) does not create a cycle in T)
add (v,w) to T
else
discard (v,w);
}
if (T contains fewer than n-1 edges)
printf("No spanning tree\n");
```

}

Examples for Kruskal's Algorithm

0 10 5

2 12 3

1 14 6

1 16 2

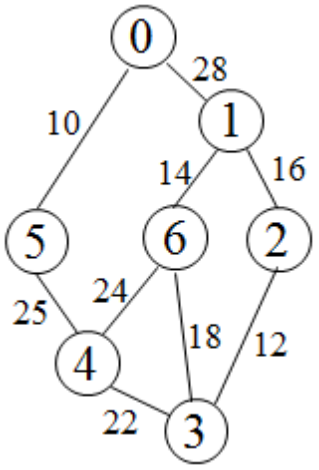
3 18 6

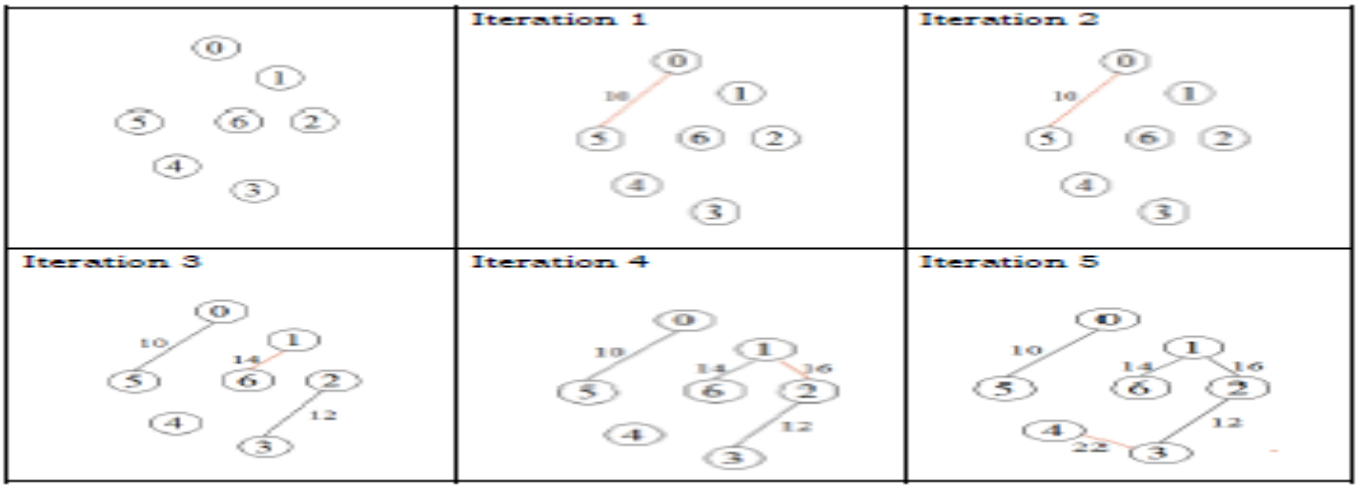
3 22 4

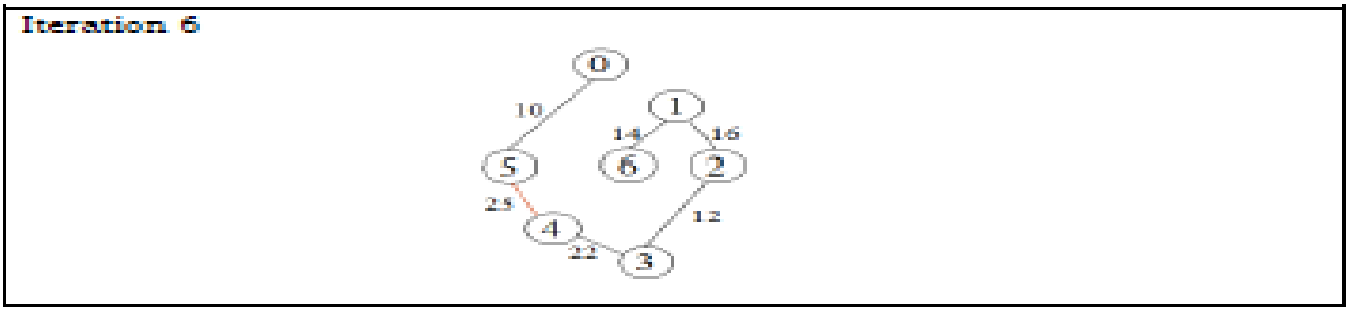
4 24 6

4 25 5

0 28 1







Prim's Algorithm

Prim's algorithm to find minimum cost spanning tree (as Kruskal's algorithm) uses the greedy approach. Prim's algorithm shares a similarity with the **shortest path first** algorithms.

Prim's algorithm, in contrast with Kruskal's algorithm, treats the nodes as a single tree and keeps on adding new nodes to the spanning tree from the given graph.

To contrast with Kruskal's algorithm and to understand Prim's algorithm better, we shall use the same example –

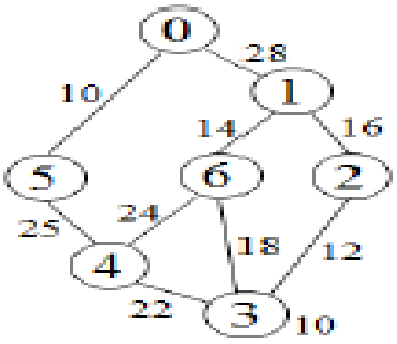
Steps of Prim's Algorithm: The following are the main 3 steps of the Prim's Algorithm:

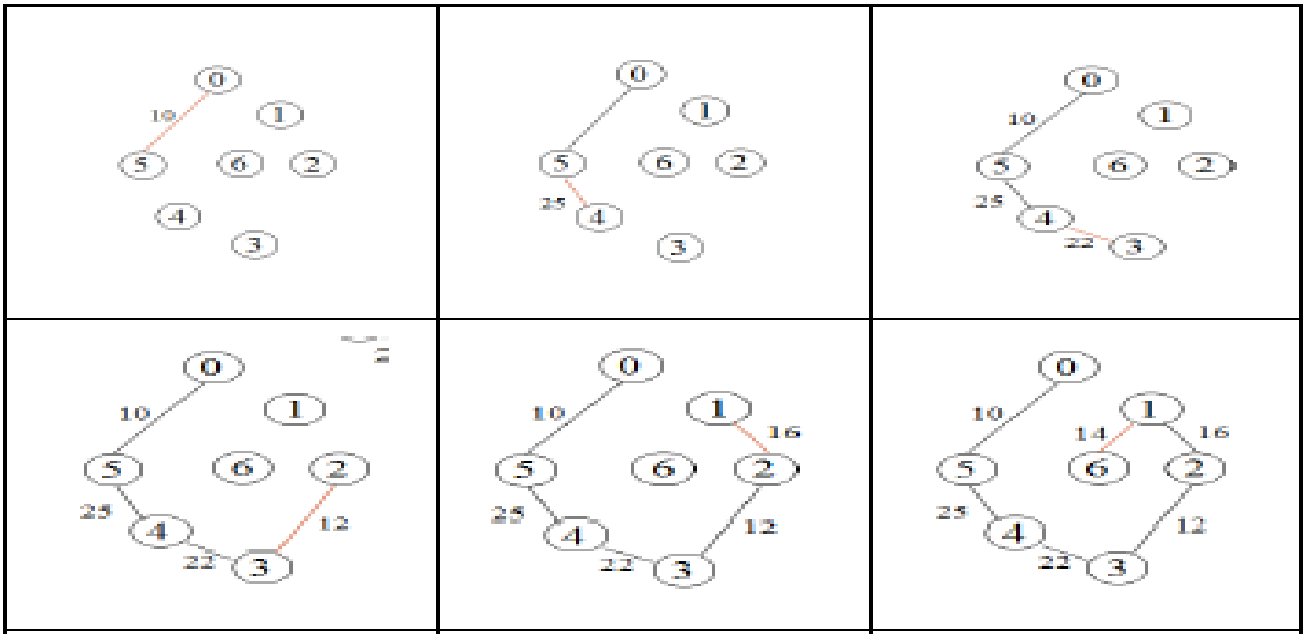
1. Begin with any vertex which you think would be suitable and add it to the tree.
2. Find an edge that connects any vertex in the tree to any vertex that is not in the tree. Note that, we don't have to form cycles.
3. Stop when n - 1 edges have been added to the tree.

Psuedocode of Prim's algorithm

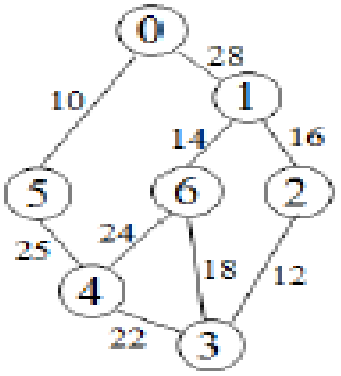
```
Prims(G,V,E)
{
T={};
TV={0};
while (T contains fewer than n-1 edges)
{
let (u,v) be a least cost edge such that and if (there is no such edge ) break;
add v to TV;
add (u,v) to T;
}
if (T contains fewer than n-1 edges)
printf("No spanning tree\n");
}
```

Examples for Prim's Algorithm

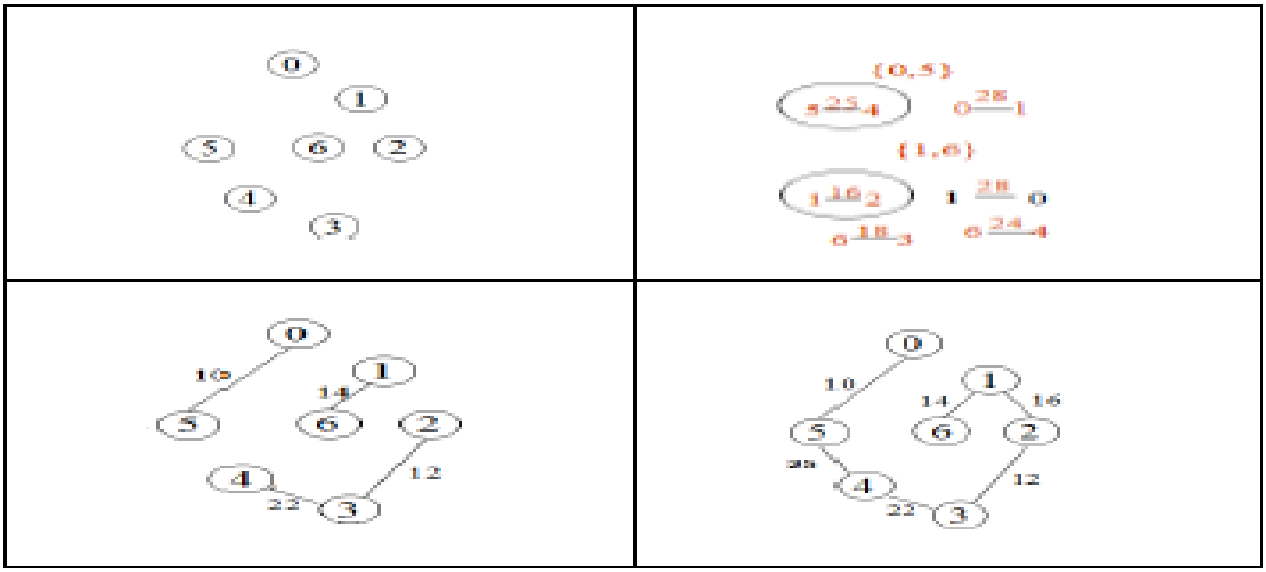




Sollin's Algorithm

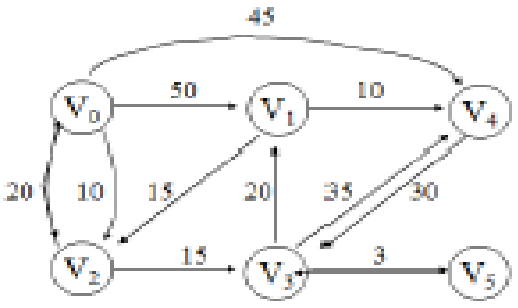


vertex	edge
0	0 -- 10 --> 5, 0 -- 28 --> 1
1	1 -- 14 --> 6, 1-- 16 --> 2, 1 -- 28 --> 0
2	2 -- 12 --> 3, 2 -- 16 --> 1
3	3 -- 12 --> 2, 3 -- 18 --> 6, 3 -- 22 --> 4
4	4 -- 22 --> 3, 4 -- 24 --> 6, 5 -- 25 --> 5
5	5 -- 10 --> 0, 5 -- 25 --> 4
6	6 -- 14 --> 1, 6 -- 18 --> 3, 6 -- 24 --> 4



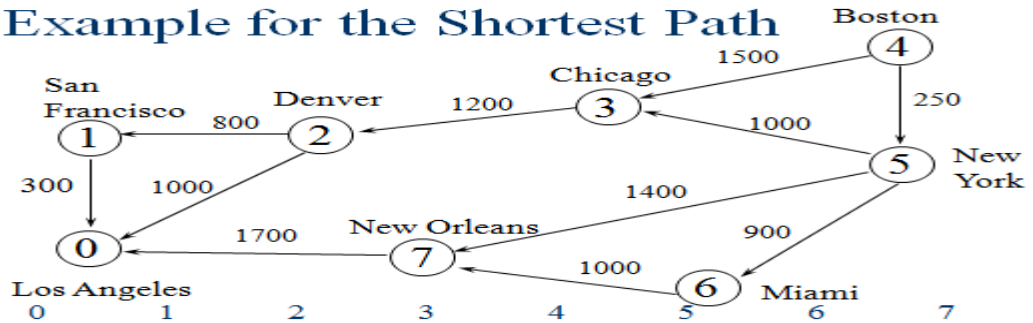
Single Source All Destinations

Graph and shortest paths from v_0



path	length
1) v0 v2	10
2) v0 v2 v3	25
3) v0 v2 v3 v1	45
4) v0 v4	45

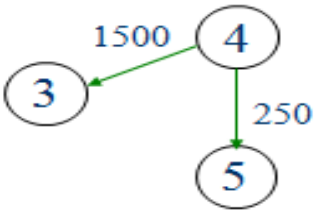
Example for the Shortest Path



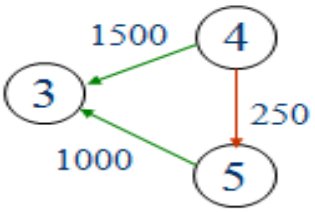
	0	1	2	3	4	5	6	7
0	0							
1	300	0						
2	1000	800	0					
3			1200	0				
4				1500	0	250		
5				1000		0	900	1400
6							0	1000
7	1700							0

Cost adjacency matrix

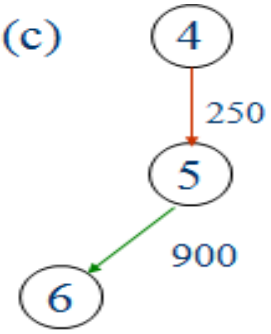
(a)



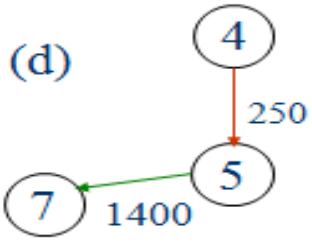
(b)



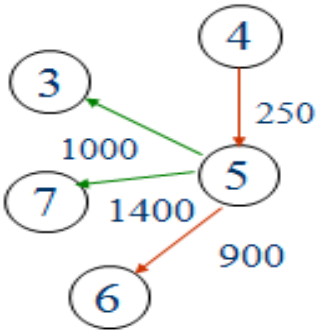
(c)



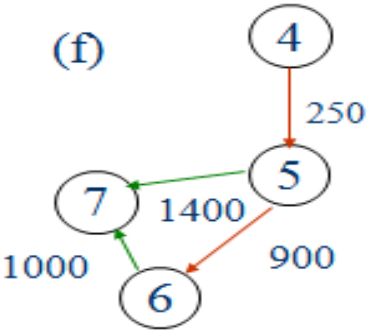
(d)



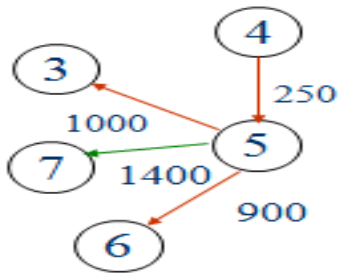
(e)



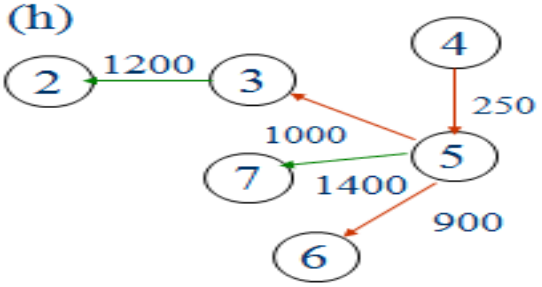
(f)



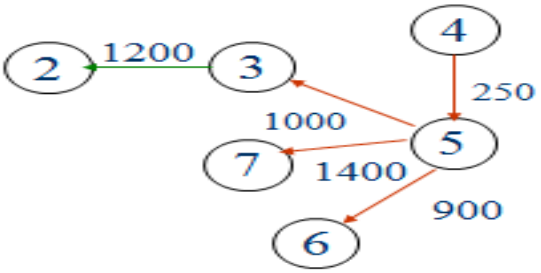
(g)



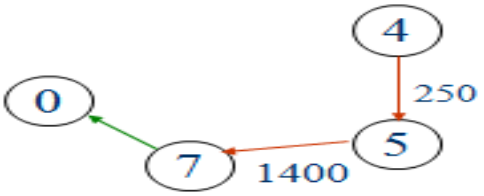
(h)



(i)



(j)



Iteration	S	Vertex Selected	LA [0]	SF [1]	DEN [2]	CHI [3]	BO [4]	NY [5]	MIA [6]	NO
Initial	--	----	+∞	+∞	+∞	1500	0	250	+∞	+∞
1	{4} (a)	5	+∞	+∞	+∞ (b)	1250	0	250 (c)	1150 (d)	1650
2	{4,5} (e)	6	+∞	+∞	+∞	1250	0	250	1150 (f)	1650
3	{4,5,6} (g)	3	+∞	+∞ (h)	2450	1250	0	250	1150	1650
4	{4,5,6,3} (i)	7 (j)	3350	+∞	2450	1250	0	250	1150	1650
5	{4,5,6,3,7}	2	3350	3250	2450	1250	0	250	1150	1650
6	{4,5,6,3,7,2}	1	3350	3250	2450	1250	0	250	1150	1650
7	{4,5,6,3,7,2,1}									

```
#define MAX_VERTICES 6
int cost[][MAX_VERTICES]=
{{ 0, 50, 10, 1000, 45, 1000},
{1000, 0, 15, 1000, 10, 1000},
{ 20, 1000, 0, 15, 1000, 1000},
{1000, 20, 1000, 0, 35, 1000},
{1000, 1000, 30, 1000, 0, 1000},
{1000, 1000, 1000, 3, 1000, 0}};
int distance[MAX_VERTICES];
short int found{MAX_VERTICES];
int n = MAX_VERTICES;
void shortestpath(int v, int cost[][MAX_ERXTICES], int distance[], int n,
short int found[])
{
```

```
int i, u, w;
for (i=0; i<n; i++)
{
found[i] = FALSE;
distance[i] = cost[v][i];
}
found[v] = TRUE;
distance[v] = 0;
for (i=0; i<n-2; i++)
{
determine n-1 paths from v
u = choose(distance, n, found);
found[u] = TRUE;
for (w=0; w<n; w++)
if (!found[w])
if (distance[u]+cost[u][w]<distance[w])
distance[w] = distance[u]+cost[u][w];
}
}
```

All Pairs Shortest Paths

All pairs shortest path algorithm finds the shortest paths between all pairs of vertices.

Solution 1

 Apply shortest path n times with each vertex as source. $O(n^3)$

Solution 2

 Represent the graph G by its cost adjacency matrix with cost[i][j]

 If the edge <i,j> is not in G, the cost[i][j] is set to some sufficiently large number

 A[i][j] is the cost of the shortest path form i to j, using only those intermediate vertices with an index <= k

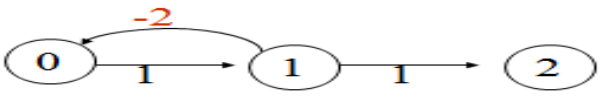
 The cost of the shortest path from i to j is A [i][j], as no vertex in G has an index greater than n-1

 A [i][j]=cost[i][j]

 Calculate the A, A, A, …, A from A iteratively

 A [i][j]=min{A [i][j], A [i][k]+A [k][j]}, k>=0

Graph with negative cycle



(a) Directed graph

$$\begin{bmatrix} 0 & 1 & \infty \\ -2 & 0 & 1 \\ \infty & \infty & 0 \end{bmatrix}$$

(b) A^{-1}

The length of the shortest path from vertex 0 to vertex 2 is $-\infty$.

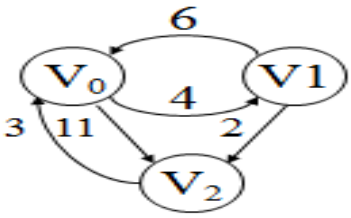
0, 1, 0, 1,0, 1, . . ., 0, 1, 2

Algorithm for All Pairs Shortest Paths

```
void allcosts(int cost[][MAX_VERTICES], int distance[][MAX_VERTICES], int n)
{
int i, j, k;
for (i=0; i<n; i++)
for (j=0; j<n; j++) distance[i][j] = cost[i][j];
for (k=0; k<n; k++)
for (i=0; i<n; i++)
for (j=0; j<n; j++)
if (distance[i][k]+distance[k][j] < distance[i][j])
distance[i][j]= distance[i][k]+distance[k][j];
}
```

Example

Directed graph and its cost matrix



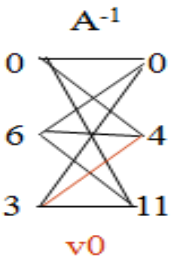
(a)Digraph G

	0	1	2
0	0	4	11
1	6	0	2
2	3	∞	0

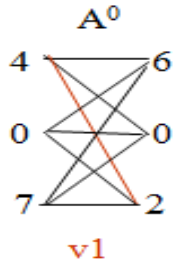
(b)Cost adjacency matrix for G

A^{-1}	0	1	2
0	0	4	11
1	6	0	2
2	3	∞	0

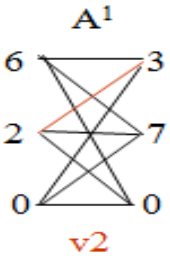
A^{0}	0	1	2
0	0	4	11
1	6	0	2
2	3	7	0



A^{1}	0	1	2
0	0	4	6
1	6	0	2
2	3	7	0



A^{2}	0	1	2
0	0	4	6
1	5	0	2
2	3	7	0

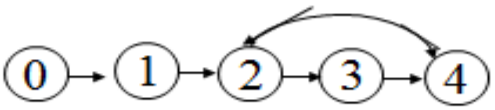


Transitive Closure

Goal: given a graph with unweighted edges, determine if there is a path from i to j for all i and j.

(1) Require positive path (> 0) lengths. transitive closure matrix

(2) Require nonnegative path (≥0) lengths. reflexive transitive closure matrix



(a) Digraph G

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

(b) Adjacency matrix A for G

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

cycle

(c) transitive closure matrix A^{+}

There is a path of length > 0

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

reflexive

(d) reflexive transitive closure matrix A^{*}

There is a path of length ≥0

Unit III

Algorithms – Priority Queues - Heaps – Heap Sort – Merge Sort – Quick Sort – Binary Search – Finding the Maximum and Minimum.

Algorithm

- An algorithm is a step-by-step procedure to solve a problem in a finite number of steps.
- Branching and repetition are included in the steps of an algorithm.
- This branching and repetition depend on the problem for which Algorithm is developed.
- All the steps of Algorithm during the definition should be written in a human-understandable language which does not depend on any programming language.
- we can choose any programming language to implement the Algorithm.
- Pseudocode and flow chart are popular ways to represent an algorithm.

An algorithm must satisfy the following criteria:

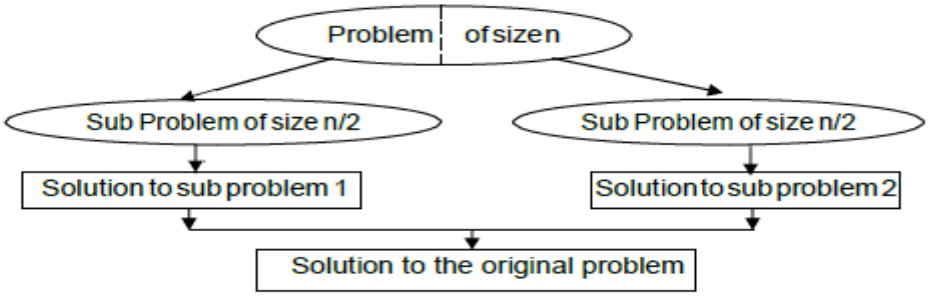
1. Input: An algorithm should have zero or more but should be a finite number of inputs. We can also say that it is essential for any algorithm before starting. Input should be given to it initially before the Algorithm begins.
2. Output: An algorithm must give at least one required result from the given set of input values. These output values are known as the solution to a problem.
3. Definiteness: Each step must be clear, unambiguous, and precisely defined.
4. Finiteness: Finiteness means Algorithm should be terminated after a finite number of steps. Also, each step should be finished in a finite amount of time.
5. Effectiveness: Each step of the Algorithm must be feasible i.e., it should be practically possible to perform the action. Every Algorithm is generally expected to be effective.

Divide and Conquer

Divide and Conquer is one of the best-known general algorithm design technique. It works according to the following general plan:

- Given a function to compute on 'n' inputs the divide-and-conquer strategy suggests splitting the inputs into 'k' distinct subsets, 1<k<=n, yielding 'k' sub problems.
- These sub problems must be solved, and then a method must be found to combine sub solutions into a solution of the whole.
- If the sub problems are still relatively large, then the divide-and-conquer strategy can possibly be reapplied.
- Often the sub problems resulting from a divide-and-conquer design are of the same type as the original problem. For those cases the reapplication of the divide-and- conquer principle is naturally expressed by a recursive algorithm.

A typical case with k=2is diagrammatically shown below.



Control Abstraction for divide and conquer:

```
Algorithm DAndC(P)
{
    if Small(P) then return S(P);
    else
    {
        divide P into smaller instances P1, P2, ..., Pk, k ≥ 1;
        Apply DAndC to each of these subproblems;
        return Combine(DAndC(P1),DAndC(P2),...,DAndC(Pk));
    }
}
```

In the above specification,

- Initially ***DAndC(P)*** is invoked, where 'P' is the problem to be solved.
- ***Small (P)*** is a Boolean-valued function that determines whether the input size is small enough that the answer can be computed without splitting. If this so, the function '***S'*** is invoked. Otherwise, the problem P is divided into smaller sub problems. These sub problems P_1, P_2 ...P_k are solved by recursive application of ***DAndC***.
- ***Combine*** is a function that determines the solution to P using the solutions to the 'k' sub problems.

Binary Search

- **Problem definition:** Let a_i, $1 \le i \le n$ be a list of elements that are sorted in non-decreasing order. The problem is to find whether a given element x is present in the list or not. If x is present we have to determine a value j (element's position) such that a_j=x. If x is not in the list, then j is set to zero.
- **Solution:** Let **P = (n, a_i…a_l, x)** denote an arbitrary instance of search problem where ***n*** is the number of elements in the list, **a_i…a_l** is the list of elements and **x** is the key element to be searched for in the given list. **Binary search** on the list is done as follows:
- Step1: Pick an index *q* in the middle range [i, *l*] i.e. *q*= [(n + 1)/2] and compare x with a_q. Step 2: if x = a_q i.e key element is equal to mid element, the problem is immediately solved.
- Step 3: if x <$a_q$in this case x has to be searched for only in the sub-list a_i, a_{i+1}, ..., a_{q-} Therefore, problem reduces to **(q-i, a_i…a_{q-1}, x).**
- Step 4: if x >a_q,x has to be searched for only in the sub-list a_{q+1}, ..., a_l . Therefore problem reduces to **(*l*-i, a_{q+1}…a_l, x).**
- For the above solution procedure, the Algorithm can be implemented as recursive or non-recursive algorithm.

Recursive binary search algorithm

```
int BinSrch(Type a[], int i, int l, Type x)
// Given an array a[i:l] of elements in nondecreasing
// order, 1<=i<=l, determine whether x is present, and
// if so, return j such that x == a[j]; else return 0.
{
   if (l==i) { // If Small(P)
      if (x==a[i]) return i;
      else return 0;
   }
   else { // Reduce P into a smaller subproblem.
      int mid = (i+l)/2;
      if (x == a[mid]) return mid;
      else if (x < a[mid]) return BinSrch(a,i,mid-1,x);
      else return BinSrch(a,mid+1,l,x);
   }
}
```

Iterative binary search:

```
int BinSearch(Type a[], int n, Type x)
// Given an array a[1:n] of elements in nondecreasing
// order, n>=0, determine whether x is present, and
// if so, return j such that x == a[j]; else return 0.
{
   int low = 1, high = n;
   while (low <= high){
      int mid = (low + high)/2;
      if (x < a[mid]) high = mid - 1;
      else if (x > a[mid]) low = mid + 1;
      else return(mid);
   }
   return(0);
}
```

Finding the maximum and minimum

Problem statement: Given a list of n elements, the problem is to find the maximum and minimum items.

StraightMaxMin: A simple and straight forward algorithm to achieve this is given below.

```
void StraightMaxMin(Type a[], int n, Type& max, Type& min)
// Set max to the maximum and min to the minimum of a[1:n].
{
    max = min = a[1];
    for (int i=2; i<=n; i++) {
       if (a[i] > max) max = a[i];
       if (a[i] < min) min = a[i];
    }
}
```

Explanation:

- ***StraightMaxMin*** requires 2(n-1) comparisons in the best, average & worst cases.
- By realizing the comparison of a[i]>max is false, improvement in a algorithm can be done.

Hence we can replace the contents of the for loop by,

```
If(a[i]>Max) then Max = a[i]; Else if (a[i]<min) min=a[i]
```

- On the average a[i] is > max half the time. So, the avg. no. of comparison is 3n/2-1.

Algorithm based on Divide and Conquer strategy

Let P = (n, a [i],……,a [j]) denote an arbitrary instance of the problem. Here 'n' is the no. of elements in the list (a[i],….,a[j]) and we are interested in finding the maximum and minimum of the list. If the list has more than 2 elements, P has to be divided into smaller instances.

For example, we might divide 'P' into the 2 instances, P1= ([n/2],a[1], a[n/2])

P2= (n-[n/2], a[[n/2]+1],……., a[n])

After having divided 'P' into 2 smaller sub problems, we can solve them by recursively invoking the same divide-and-conquer algorithm.

Algorithm:

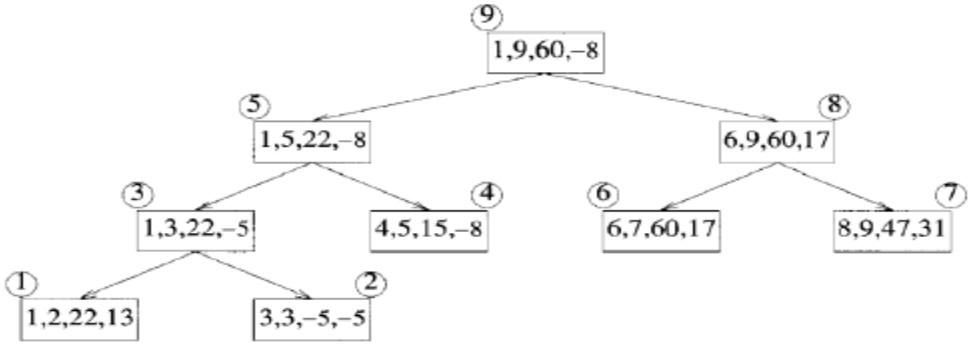
```
void MaxMin(int i, int j, Type& max, Type& min)
// a[1:n] is a global array. Parameters i and j are
// integers, 1 <= i <= j <= n. The effect is to set
// max and min to the largest and smallest values in
// a[i:j], respectively.
{
   if (i == j) max = min = a[i]; // Small(P)
   else if (i == j-1) { // Another case of Small(P)
           if (a[i] < a[j]) { max = a[j]; min = a[i]; }
           else { max = a[i]; min = a[j]; }
        }
   else { // If P is not small
          // divide P into subproblems.
      // Find where to split the set.
          int mid=(i+j)/2; Type max1, min1;
      // Solve the subproblems.
        MaxMin(i, mid, max, min);
        MaxMin(mid+1, j, max1, min1);
      // Combine the solutions.
        if (max < max1) max = max1;
        if (min > min1) min = min1;
   }
}
```

Example:

Suppose we simulate MaxMin on the following nine elements:

a:	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]
	22	13	−5	−8	15	60	17	31	47

A good way of keeping track of recursive calls is to build a tree by adding a node each time a new call is made. For this algorithm each node has four items of information: i, j, max, and min. On the array $a[\,]$ above, the tree of recursive calls of MaxMin is as follows



Analysis - Time Complexity

Now what is the number of element comparisons needed for MaxMin? If $T(n)$ represents this number, then the resulting recurrence relation is

$$T(n) = \begin{cases} T(\lceil n/2 \rceil) + T(\lceil n/2 \rceil) + 2 & n > 2 \\ 1 & n = 2 \\ 0 & n = 1 \end{cases}$$

When n is a power of two, $n = 2^k$ for some positive integer k, then

$$\begin{aligned} T(n) &= 2T(n/2) + 2 \\ &= 2(2T(n/4) + 2) + 2 \\ &= 4T(n/4) + 4 + 2 \\ &\vdots \\ &= 2^{k-1}T(2) + \textstyle\sum_{1 \le i \le k-1} 2^i \\ &= 2^{k-1} + 2^k - 2 = 3n/2 - 2 \end{aligned} \tag{3.3}$$

Note that $3n/2 - 2$ is the best-, average-, and worst-case number of comparisons when n is a power of two.

Compared with the straight forward method (2n-2) this method saves 25% in comparisons.

Space Complexity

Compared to the straight forward method, the MaxMin method requires extra stack space for i, j, max, min, max1 and min1. Given n elements there will be $[\log_2 n] + 1$ levels of recursion and we need to save seven values for each recursive call. (6 + 1 for return address).

Merge Sort

Merge sort is a perfect example of a successful application of the divide-and conquer technique. It sorts a given array A [O ... n - 1] by dividing it into two halves A [0 .. \n/2]-1] and A [\n/2] .. n-1], sorting each of them recursively, and then merging the two smaller sorted arrays into a single sorted one.

ALGORITHM *Mergesort*$(A[0..n-1])$

```
//Sorts array A[0..n − 1] by recursive mergesort
//Input: An array A[0..n − 1] of orderable elements
//Output: Array A[0..n − 1] sorted in nondecreasing order
if n > 1
    copy A[0..⌊n/2⌋ − 1] to B[0..⌊n/2⌋ − 1]
    copy A[⌊n/2⌋..n − 1] to C[0..⌈n/2⌉ − 1]
    Mergesort(B[0..⌊n/2⌋ − 1])
    Mergesort(C[0..⌈n/2⌉ − 1])
    Merge(B, C, A)    //see below
```

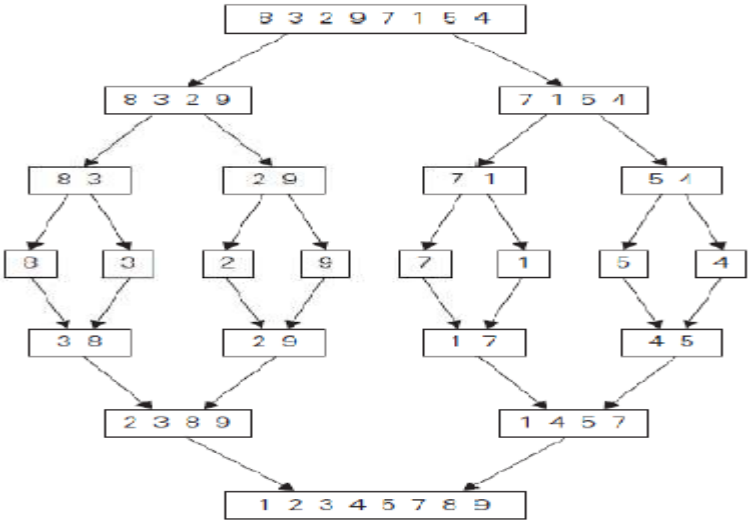
The merging of two sorted arrays can be done as follows.

- Two pointers (array indices) are initialized to point to the first elements of the arrays being merged.
- The elements pointed to are compared, and the smaller of them is added to a new array being constructed
- After that, the index of the smaller element is incremented to point to its immediate successor in the array it was copied from. This operation is repeated until one of the two given arrays is exhausted, and then the remaining elements of the other array are copied to the end of the new array.

ALGORITHM *Merge*$(B[0..p-1], C[0..q-1], A[0..p+q-1])$

```
//Merges two sorted arrays into one sorted array
//Input: Arrays B[0..p − 1] and C[0..q − 1] both sorted
//Output: Sorted array A[0..p + q − 1] of the elements of B and C
i ← 0; j ← 0; k ← 0
while i < p and j < q do
    if B[i] ≤ C[j]
        A[k] ← B[i]; i ← i + 1
    else A[k] ← C[j]; j ← j + 1
    k ← k + 1
if i = p
    copy C[j..q − 1] to A[k..p + q − 1]
else copy B[i..p − 1] to A[k..p + q − 1]
```

Example:
The operation of the algorithm on the list 8, 3, 2, 9, 7, 1, 5, 4 is illustrated in the figure.



Analysis

Here the basic operation is key comparison. As merge sort execution does not depend on the order of the data, best case and average case runtime are the same as worst case runtime.

Worst case: During key comparison, neither of the two arrays becomes empty before the
other one contains just one element leads to the worst case of merge sort. Assuming for
simplicity that total number of elements **n** is a power of 2, the recurrence relation for the
number of key comparisons $C(n)$ is

$$C(n) = 2C(n/2) + C_{merge}(n) \quad \text{for } n > 1, \quad C(1) = 0.$$

where, $C_{merge}(n)$ is the number of key comparisons made during the merging stage.

Let us analyze $C_{merge}(n)$, the number of key comparisons performed during the merging stage. At each step, exactly one comparison is made, after which the total number of elements in the two arrays still needing to be processed is reduced by 1. In the worst case, neither of the two arrays becomes empty before the other one contains just one element (e.g., smaller elements may come from the alternating arrays).Therefore, for the worst case, $C_{merge}(n) = n - 1$.

Now,

$$C_{worst}(n) = 2C_{worst}(n/2) + n - 1 \quad \text{for } n > 1, \quad C_{worst}(1) = 0.$$

Solving the recurrence equation using **master theorem:**

Here a = 2, b = 2, f (n) = n, d = 1. Therefore $2 = 2^1$, case 2 holds in the master theorem

$C_{worst}(n) = \Theta(n^d \log n) = \Theta(n^1 \log n) = \Theta(n \log n)$Therefore **$C_{worst}(n) = \Theta(n \log n)$**

Advantages:

- Number of comparisons performed is nearly optimal.
- For large n, the number of comparisons made by this algorithm in the average case turns out to be about 0.25n less and hence is also in $\Theta(n \log n)$.

- Mergesort will never degrade to O (n^2)
- Another advantage of mergesort over quicksort and heapsort is its **stability**. (A sorting algorithm is said to be stable if two objects with equal keys appear in the same order in sorted output as they appear in the input array to be sorted.)

Limitations:

The principal shortcoming of mergesort is the linear amount [O(n)] of extra storage the algorithm requires. Though merging can be done in-place, the resulting algorithm is quite complicated and of theoretical interest only.

Variations of merge sort

1. The algorithm can be implemented bottom up by merging pairs of the array's elements, then merging the sorted pairs, and so on. (If n is not a power of 2, only slight bookkeeping complications arise.) This avoids the time and space overhead of using a stack to handle recursive calls.
2. We can divide a list to be sorted in more than two parts, sort each recursively, and then merge them together. This scheme, which is particularly useful for sorting files residing on secondary memory devices, is called multiway mergesort.

Quick sort

Quicksort is the other important sorting algorithm that is based on the divide-and-conquer approach. Unlike mergesort, which divides its input elements according to their position in the array, quicksort divides (or partitions) them according to their value.

A partition is an arrangement of the array's elements so that all the elements to the left of some element A[s] are less than or equal to A[s], and all the elements to the right of A[s] are greater than or equal to it:

$$\underbrace{A[0] \ldots A[s-1]}_{\text{all are} \le A[s]} \quad A[s] \quad \underbrace{A[s+1] \ldots A[n-1]}_{\text{all are} \ge A[s]}$$

Obviously, after a partition is achieved, A[s] will be in its final position in the sorted array, and we can continue sorting the two subarrays to the left and the right of A[s] independently (e.g., by the same method).

In quick sort, the entire work happens in the division stage, with no work required to combine the solutions to the sub problems.

ALGORITHM *Quicksort*$(A[l..r])$

```
//Sorts a subarray by quicksort
//Input: Subarray of array A[0..n − 1], defined by its left and right
//       indices l and r
//Output: Subarray A[l..r] sorted in nondecreasing order
if l < r
    s ← Partition(A[l..r]) //s is a split position
    Quicksort(A[l..s − 1])
    Quicksort(A[s + 1..r])
```

Partitioning

We start by selecting a pivot—an element with respect to whose value we are going to divide the subarray. There are several different strategies for selecting a pivot. We use

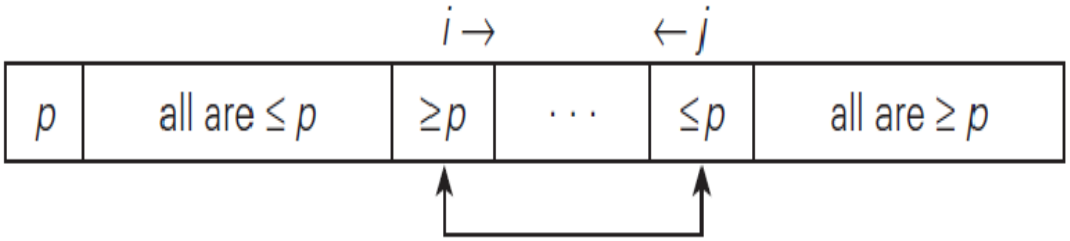
the sophisticated method suggested by C.A.R. Hoare, the prominent British computer scientist who invented quicksort.
Select the subarray's first element: p = A[l].Now scan the subarray from both ends, comparing the subarray's elements to the pivot.

- The left-to-right scan, denoted below by index pointer i, starts with the second element. Since we want elements smaller than the pivot to be in the left part of the subarray, this scan skips over elements that are smaller than the pivot and stops upon encountering the first element greater than or equal to the pivot.

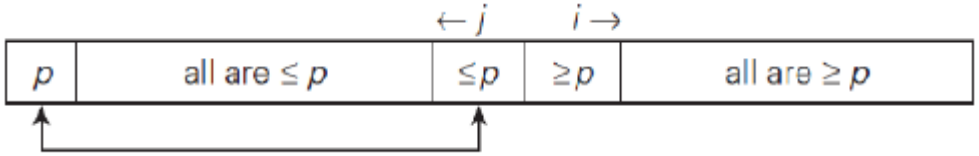
- The right-to-left scan, denoted below by index pointer j, starts with the last element of the subarray. Since we want elements larger than the pivot to be in the right part of the subarray, this scan skips over elements that are larger than the pivot and stops on encountering the first element smaller than or equal to the pivot.

After both scans stop, three situations may arise, depending on whether or not the scanning indices have crossed.

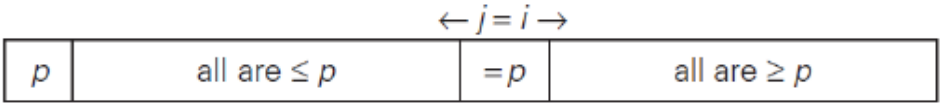
- If scanning indices i and j have not crossed, i.e., i< j, we simply exchange A[i] and A[j] and resume the scans by incrementing I and decrementing j, respectively:



If the scanning indices have crossed over, i.e., i> j, we will have partitioned the subarray after exchanging the pivot with A[j]:



If the scanning indices stop while pointing to the same element, i.e., i = j, the value they are pointing to must be equal to p. Thus, we have the subarray partitioned, with the split position s = i = j :



We can combine this with the case-2 by exchanging the pivot with A[j] whenever i≥j

ALGORITHM *HoarePartition*$(A[l..r])$

//Partitions a subarray by Hoare's algorithm, using the first element as a pivot

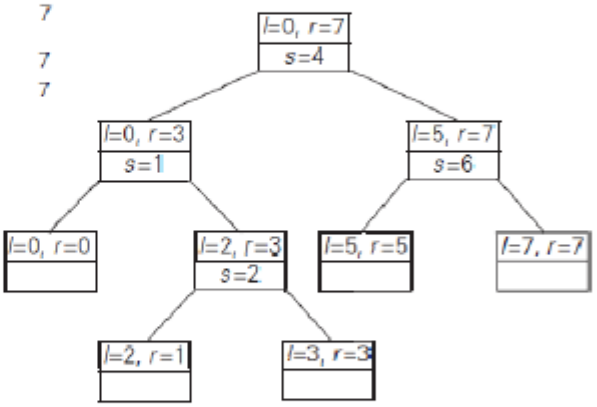
//Input: Subarray of array $A[0..n-1]$, defined by its left and right indices l and r $(l<r)$

//Output: Partition of $A[l..r]$, with the split position returned as this function's value

$p \leftarrow A[l]$
$i \leftarrow l;\ j \leftarrow r+1$
repeat
    repeat $i \leftarrow i+1$ **until** $A[i] \geq p$
    repeat $j \leftarrow j-1$ **until** $A[j] \leq p$
    swap$(A[i], A[j])$
until $i \geq j$
swap$(A[i], A[j])$ //undo last swap when $i \geq j$
swap$(A[l], A[j])$
return j

Example: Example of quicksort operation. (a) Array's transformations with pivots shown in bold. (b) Tree of recursive calls to Quicksort with input values land r of subarray bounds and split position s of a partition obtained.

0	1	2	3	4	5	6	7
5	3	1	9	8	2	4	7
5	3	1	9	8	2	4	7
5	3	1	4	8	2	9	7
5	3	1	4	8	2	9	7
5	3	1	4	2	8	9	7
5	3	1	4	2	8	9	7
2	3	1	4	5	8	9	7
2	3	1	4				
2	3	1	4				
2	1	3	4				
2	1	3	4				
1	2	3	4				
1							
		3	4				
		3	4				
			4				
					8	9	7
					8	7	9
					8	7	9
					7	8	9
					7		
							9



Analysis

Best Case -Here the basic operation is key comparison. Number of key comparisons made before a partition is achieved is n + 1 if the scanning indices cross over and n if they coincide. If all the splits happen in the middle of corresponding subarrays, we will have the best case. The number of key comparisons in the best case satisfies the recurrence,

$$C_{best}(n) = 2C_{best}(n/2) + n \quad \text{for } n > 1, \quad C_{best}(1) = 0.$$

According to the Master Theorem, $C_{best}(n) \in \Theta(n \log_2 n)$; solving it exactly for n = 2^k yields

$C_{best}(n) = n \log_2 n$.

Worst Case – In the worst case, all the splits will be skewed to the extreme: one of the two subarrays will be empty, and the size of the other will be just 1 less than the size of the subarray being partitioned. This unfortunate situation will happen, in particular, for increasing arrays. Indeed, if A[0..n − 1] is a strictly increasing array and we use A[0] as the pivot, the left-to-right scan will stop on A[1] while the right-to-left scan will go all the way to reach A[0], indicating the split at position 0:So, after making n + 1 comparisons to get to this partition and exchanging the pivot A[0] with itself, the algorithm will be left with the strictly increasing array A[1..n − 1] to sort. This sorting of strictly increasing arrays of diminishing sizes will continue until the last one A[n−2.. n−1] has been processed. The total number of key comparisons made will be equal to

$$C_{worst}(n) = (n+1) + n + \cdots + 3 = \frac{(n+1)(n+2)}{2} - 3 \in \Theta(n^2).$$

Average Case - Let $C_{avg}(n)$ be the average number of key comparisons made by quicksort on a randomly ordered array of size n. A partition can happen in any position s ($0 \le s \le n-1$) after n+1comparisons are made to achieve the partition. After the partition, the left and right subarrays will have s and n − 1− s elements, respectively. Assuming that the partition split can happen in each position s with the same probability 1/n, we get the following recurrence relation:

$$C_{avg}(n) = \frac{1}{n}\sum_{s=0}^{n-1}[(n+1) + C_{avg}(s) + C_{avg}(n-1-s)] \quad \text{for } n > 1,$$

$$C_{avg}(0) = 0, \quad C_{avg}(1) = 0.$$

Its solution, which is much trickier than the worst- and best-case analyses, turns out to be

$$C_{avg}(n) \approx 2n \ln n \approx 1.39n \log_2 n.$$

Thus, on the average, quicksort makes only 39% more comparisons than in the best case. Moreover, its innermost loop is so efficient that it usually runs faster than mergesort on randomly ordered arrays of nontrivial sizes. This certainly justifies the name given to the algorithm by its inventor.

Variations: Because of quicksort's importance, there have been persistent efforts over the years to refine the basic algorithm. Among several improvements discovered by researchers are:

- Better pivot selection methods such as randomized quicksort that uses a random element or the median-of-three method that uses the median of the leftmost, rightmost, and the middle element of the array

- Switching to insertion sort on very small subarrays (between 5 and 15 elements for most computer systems) or not sorting small subarrays at all and finishing the algorithm with insertion sort applied to the entire nearly sorted array
- Modifications of the partitioning algorithm such as the three-way partition into segments smaller than, equal to, and larger than the pivot

Limitations: 1. It is not stable. 2. It requires a stack to store parameters of subarrays that are yet to be sorted. 3. While Performance on randomly ordered arrays is known to be sensitive not only to the implementation details of the algorithm but also to both computer architecture and data type.

Unit - 4

Greedy Method : The General Method – Optimal Storage on Tapes – Knapsack Problem – Job Sequencing with Deadlines – Optimal Merge Patterns.

GENERAL METHOD

Greedy Method

Greedy is the most straight forward design technique. Most of the problems have n inputs and require us to obtain a subset that satisfies some constraints. Any subset that satisfies these constraints is called a feasible solution. We need to find a feasible solution that either maximizes or minimizes the objective function. A feasible solution that does this is called an optimal solution.

The greedy method is a simple strategy of progressively building up a solution, one element at a time, by choosing the best possible element at each stage. At each stage, a decision is made regarding whether or not a particular input is in an optimal solution. This is done by considering the inputs in an order determined by some selection procedure. If the inclusion of the next input, into the partially constructed optimal solution will result in an infeasible solution then this input is not added to the partial solution. The selection procedure itself is based on some optimization measure. Several optimization measures are plausible for a given problem. Most of them, however, will result in algorithms that generate sub-optimal solutions. This version of greedy technique is called *subset paradigm*. Some problems like Knapsack, Job sequencing with deadlines and minimum cost spanning trees are based on *subset paradigm.*

For the problems that make decisions by considering the inputs in some order, each decision is made using an optimization criterion that can be computed using decisions already made. This version of greedy method is *ordering paradigm*. Some problems like optimal storage on tapes, optimal merge patterns and single source shortest path are based on *ordering paradigm.*

CONTROL ABSTRACTION

Algorithm Greedy (a, n)

```
// a(1 : n) contains the „n“ inputs
{
solution := □; // initialize the solution to empty for i:=1 to n do
{
x := select (a);
if feasible (solution, x) then
solution := Union (Solution, x);
}
return solution;
}
```

Procedure Greedy describes the essential way that a greedy based algorithm will look, once a particular problem is chosen and the functions select, feasible and union are properly implemented.
The function select selects an input from „a“, removes it and assigns its value to „x“. Feasible is a Boolean valued function, which determines if „x“ can be included into the solution vector. The function Union combines „x“ with solution and updates the objective function.

KNAPSACK PROBLEM

Let us apply the greedy method to solve the knapsack problem. We are given „n“ objects and a knapsack. The object „i“ has a weight w_i and the knapsack has a capacity „m“. If a fraction x_i, $0 < x_i < 1$ of object i is placed into the knapsack then a profit of $p_i x_i$ is earned. The objective is to fill the knapsack that maximizes the total profit earned.
Since the knapsack capacity is „m“, we require the total weight of all chosen objects to be at most „m“. The problem is stated as:

$$\text{maximize} \sum_{i=1}^{n} p_i x_i$$

$$\text{subject to} \sum_{i=1}^{n} a_i \quad x_i \leq M \qquad \text{where, } 0 \leq x_i \leq 1 \text{ and } 1 \leq i \leq n$$

The profits and weights are positive numbers.
Algorithm
If the objects are already been sorted into non-increasing order of p[i] / w[i] then the algorithm given below obtains solutions corresponding to this strategy.

Algorithm GreedyKnapsack (m, n)

```
// P[1 : n] and w[1 : n] contain the profits and weights respectively of
// Objects ordered so that p[i] / w[i] > p[i + 1] / w[i + 1].
// m is the knapsack size and x[1: n] is the solution vector.
{
for i := 1 to n do x[i] := 0.0 // initialize x U := m;
for i := 1 to n do
{
if (w(i) > U) then break;
x [i] := 1.0; U := U – w[i];
}
if (i < n) then x[i] := U / w[i];
}
```

Running time:
The objects are to be sorted into non-decreasing order of p_i / w_i ratio. But if we disregard the time to initially sort the objects, the algorithm requires only O(n) time.
Example:
Consider the following instance of the knapsack problem: n = 3, m = 20, $(p_1, p_2, p_3) = (25, 24, 15)$ and $(w_1, w_2, w_3) = (18, 15, 10)$.

OPTIMAL STORAGE ON TAPES

There are „n“ programs that are to be stored on a computer tape of length „L“. Each program „i“ is of length l_i, $1 \leq i \leq n$. All the programs can be stored on the tape if and only if the sum of the lengths of the programs is at most „L“.
We shall assume that whenever a program is to be retrieved from this tape, the tape is initially positioned at the front. If the programs are stored in the order $i = i_1, i_2, \ldots, i_n$, the time t_J needed to retrieve program i_J is proportional to

$$\sum_{1 \le k \le j} l_{i_k}$$

If all the programs are retrieved equally often then the expected or mean retrieval time (MRT) is:

$$\frac{1}{n} \cdot \sum_{1 \le j \le n} t_j$$

For the optimal storage on tape problem, we are required to find the permutation for the 'n' programs so that when they are stored on the tape in this order the MRT is minimized.

$$d(I) = \sum_{j=1}^{n} \sum_{k=1}^{j} l_{i_k}$$

Example
Let n = 3, (l_1, l_2, l_3) = (5, 10, 3). Then find the optimal ordering?
Solution:
There are n! = 6 possible orderings. They are:

Ordering I	d(I)		
1, 2, 3	5 + (5 +10) +(5 + 10 + 3)	=	38
1, 3, 2	5 + (5 + 3) + (5 + 3 + 10)	=	31
2, 1, 3	10 + (10 + 5) + (10 + 5 + 3)	=	43
2, 3, 1	10 + (10 + 3) + (10 + 3 + 5)	=	41
3, 1, 2	3 + (3 + 5) + (3 + 5 + 10)	=	29
3, 2, 1	3 + (3 + 10) + (3 + 10 + 5)	=	34

From the above, it simply requires to store the programs in non-decreasing order (increasing order) of their lengths. This can be carried out by using a efficient sorting algorithm (Heap sort). This ordering can be carried out in O (n log n) time using heap sort algorithm.

The tape storage problem can be extended to several tapes. If there are m > 1 tapes, $T_o, \ldots\ldots, T_{m-1}$, then the programs are to be distributed over these tapes.

The total retrieval time (RT) is $\sum_{j=0}^{m-1} d(I_j)$

The objective is to store the programs in such a way as to minimize RT.

The programs are to be sorted in non decreasing order of their lengths l_i's, $l_1 \le l_2 \le \ldots\ldots l_n$.
The first 'm' programs will be assigned to tapes $T_o, \ldots, T_{m-1}$ respectively. The next 'm' programs will be assigned to $T_0, \ldots, T_{m-1}$ respectively. The general rule is that program i is stored on tape T_i mod m.

Algorithm:

The algorithm for assigning programs to tapes is as follows:

Algorithm Store (n, m)

```
// n is the number of programs and m the number of tapes
{
j := 0; // next tape to store on for i :=1 to n do
{
Print („append program“, i, „to permutation for tape“, j); j := (j + 1) mod m;
}
}
```

On any given tape, the programs are stored in non-decreasing order of their lengths.

JOB SEQUENCING WITH DEADLINES

When we are given a set of „n“ jobs. Associated with each Job i, deadline $d_i > 0$ and profit $P_i > 0$. For any job „i“ the profit pi is earned iff the job is completed by its deadline. Only one machine is available for processing jobs. An optimal solution is the feasible solution with maximum profit.

Sort the jobs in „j“ ordered by their deadlines. The array d [1 : n] is used to store the deadlines of the order of their p-values. The set of jobs j [1 : k] such that j [r], $1 \le r \le k$ are the jobs in „j“ and d (j [1]) ≤ d (j[2]) ≤ . . . ≤ d (j[k]). To test whether J U {i} is feasible, we have just to insert i into J preserving the deadline ordering and then verify that d [J[r]] ≤ r, $1 \le r \le k+1$.

Example:

Let n = 4, (P_1, P_2, P_3, P_4,) = (100, 10, 15, 27) and (d_1 d_2 d_3 d_4) = (2, 1, 2, 1). The feasible solutions and their values are:

S.No	Feasible Solution	Procuring sequence	Value	Remarks
1	1,2	2,1	110	
2	1,3	1,3 or 3,1	115	
3	1,4	4,1	127	**OPTIMAL**
4	2,3	2,3	25	
5	3,4	4,3	42	
6	1	1	100	
7	2	2	10	
8	3	3	15	
9	4	4	27	

Algorithm:

The algorithm constructs an optimal set J of jobs that can be processed by their deadlines.

Algorithm GreedyJob (d, J, n)

```
// J is a set of jobs that can be completed by their deadlines.
{
        J := {1};
        for i := 2 to n do
        {
                if (all jobs in J U {i} can be completed by their dead lines)
                then J := J U {i};
        }
}
```

OPTIMAL MERGE PATERNS

Given „n“ sorted files, there are many ways to pair wise merge them into a single sorted file. As, different pairings require different amounts of computing time, we want to determine an optimal (i.e., one requiring the fewest comparisons) way to pair wise merge „n“ sorted files together. This type of merging is called as 2-way merge patterns. To merge an n-record file and an m-record file requires possibly n + m record moves, the obvious choice choice is, at each step merge the two smallest files together. The two-way merge patterns can be represented by binary merge trees.

Algorithm to Generate Two-way Merge Tree:

```
struct treenode
{
  treenode * lchild;
  treenode * rchild;
};
```

Algorithm to Generate Two-way Merge Tree:

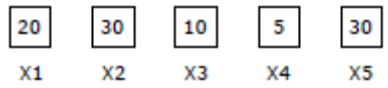
```
struct treenode
{
        treenode * lchild;
        treenode * rchild;
};

Algorithm TREE (n)
// list is a global of n single node binary trees
{
        for i := 1 to n – 1 do
        {
                pt ← new treenode
                (pt → lchild) ← least (list);          //  merge  two  trees with smallest
        lengths
                (pt → rchild) ← least (list);
                (pt → weight) ← ((pt → lchild) → weight) + ((pt → rchild) → weight);
                 insert (list, pt);
        }
        return  least  (list);                         // The tree left  in list is the      merge
tree
}
```

Example 1:
Suppose we are having three sorted files X_1, X_2 and X_3 of length 30, 20, and 10 records each. Merging of the files can be carried out as follows:

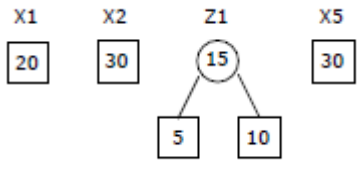
S.No	First Merging	Record moves in first merging	Second merging	Record moves in second merging	Total no. of records moves
1.	X_1 & X_2 = T1	50	T_1 & X_3	60	50 + 60 = 110
2.	X_2 & X_3 = T1	30	T_1 & X_1	60	30 + 60 = 90

The Second case is optimal.

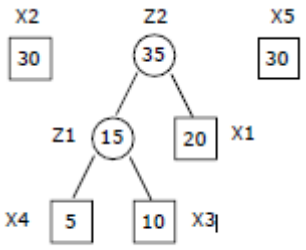
Example 2:
Given five files (X1, X2, X3, X4, X5) with sizes (20, 30, 10, 5, 30). Apply greedy rule to find optimal way of pair wise merging to give an optimal solution using binary merge tree representation.
Solution:



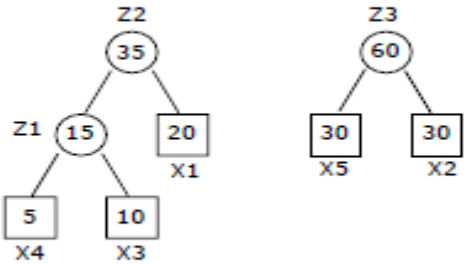
Merge X_4 and X_3 to get 15 record moves. Call this Z_1.



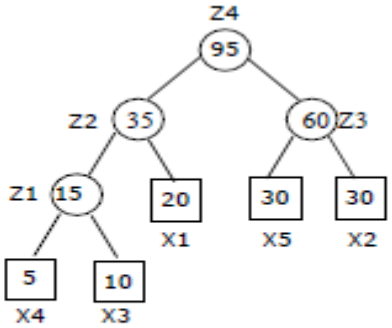
Merge Z_1 and X_1 to get 35 record moves. Call this Z_2.



Merge X_2 and X_5 to get 60 record moves. Call this Z_3.



Merge Z_2 and $Z_3$to get 90 record moves. This is the answer. Call this Z_4.



Therefore the total number of record moves is 15 + 35 + 60 + 95 = 205. This is an optimal merge pattern for the given problem.

Unit V

Back tracking: The General Method – The 8-Queens Problem – Sum of Subsets – Graph Coloring.

Backtracking

Some problems can be solved, by exhaustive search. The exhaustive-search technique suggests generating all candidate solutions and then identifying the one (or the ones) with a desired property.

Backtracking is a more intelligent variation of this approach. The principal idea is to construct solutions one component at a time and evaluate such partially constructed candidates as follows. If a partially constructed solution can be developed further without violating the problem's constraints, it is done by taking the first remaining legitimate option for the next component. If there is no legitimate option for the next component, no alternatives for any remaining component need to be considered. In this case, the algorithm

backtracks to replace the last component of the partially constructed solution with its next option.
It is convenient to implement this kind of processing by constructing a tree of choices being made, called the **state-space tree**. Its root represents an initial state before the search for a solution begins. The nodes of the first level in the tree represent the choices made for the first component of a solution; the nodes of the second level represent the choices for the second component, and soon. A node in a state-space tree is said to be promising if it corresponds to a partially constructed solution that may still lead to a complete solution; otherwise, it is called **non-promising**. Leaves represent either non-promising dead ends or complete solutions found by the algorithm.
In the majority of cases, a state space tree for a backtracking algorithm is constructed in the manner of depth-first search. If the current node is promising, its child is generated by adding the first remaining legitimate option for the next component of a solution, and the processing moves to this child. If the current node turns out to be non-promising, the algorithm backtracks to the node's parent to consider the next possible option for its last component; if there is no such option, it backtracks one more level up the tree, and so on. Finally, if the algorithm reaches a complete solution to the problem, it either stops (if just one solution is required) or continues searching for other possible solutions.

General method

In many applications of the backtrack method, the desired solution is expressible as an n-tuple $(x_1, \ldots, x_n)$, where the x_i are chosen from some finite set S_i. Often the problem to be solved calls for finding one vector

Suppose m_i is the size of set S_i. Then there are $m = m_1 m_2 \cdots m_n$ n-tuples that are possible candidates for satisfying the function P. The *brute force approach* would be to form all these n-tuples, evaluate each one with P, and save those which yield the optimum. The backtrack algorithm has as its virtue the ability to yield the same answer with far fewer than m trials. Its basic idea is to build up the solution vector one component at a time and to use modified criterion functions $P_i(x_1, \ldots, x_i)$ (sometimes called bounding functions) to test whether the vector being formed has any chance of success. The major advantage of this method is this: if it is realized that the partial vector $(x_1, x_2, \ldots, x_i)$ can in no way lead to an optimal solution, then $m_{i+1} \cdots m_n$ possible test vectors can be ignored entirely.

Many of the problems we solve using backtracking require that all the solutions satisfy a complex set of constraints. For any problem these constraints can be divided into two categories: *explicit* and *implicit*.

Definition 7.1 Explicit constraints are rules that restrict each x_i to take on values only from a given set. □

Common examples of explicit constraints are

$$\begin{array}{rcrcl} x_i \geq 0 & \text{or} & S_i & = & \{\text{all nonnegative real numbers}\} \\ x_i = 0 \text{ or } 1 & \text{or} & S_i & = & \{0,1\} \\ l_i \leq x_i \leq u_i & \text{or} & S_i & = & \{a : l_i \leq a \leq u_i\} \end{array}$$

The explicit constraints depend on the particular instance I of the problem being solved. All tuples that satisfy the explicit constraints define a possible *solution space* for I.

Definition 7.2 The implicit constraints are rules that determine which of the tuples in the solution space of I satisfy the criterion function. Thus implicit constraints describe the way in which the x_i must relate to each other. □

General Algorithm (Recursive)

```
Algorithm Backtrack(k)
// This schema describes the backtracking process using
// recursion. On entering, the first k − 1 values
// x[1], x[2], ..., x[k − 1] of the solution vector
// x[1 : n] have been assigned. x[ ] and n are global.
{
    for (each x[k] ∈ T(x[1], ..., x[k − 1])) do
    {
        if (B_k(x[1], x[2], ..., x[k]) ≠ 0) then
        {
            if (x[1], x[2], ..., x[k] is a path to an answer node)
                then write (x[1 : k]);
            if (k < n) then Backtrack(k + 1);
        }
    }
}
```

General Algorithm (Iterative)

```
Algorithm IBacktrack(n)
// This schema describes the backtracking process.
// All solutions are generated in x[1 : n] and printed
// as soon as they are determined.
{
     k := 1;
     while (k ≠ 0) do
     {
          if (there remains an untried x[k] ∈ T(x[1], x[2], . . .,
               x[k − 1]) and B_k(x[1], . . . , x[k]) is true) then
          {
                    if (x[1], . . . , x[k] is a path to an answer node)
                         then write (x[1 : k]);
                    k := k + 1; // Consider the next set.
          }
          else k := k − 1; // Backtrack to the previous set.
     }
}
```

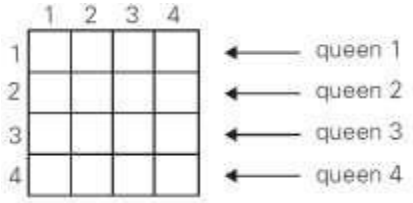
General Algorithm for backtracking

ALGORITHM *Backtrack*$(X[1..i])$

//Gives a template of a generic backtracking algorithm
//Input: $X[1..i]$ specifies first i promising components of a solution
//Output: All the tuples representing the problem's solutions
if $X[1..i]$ is a solution **write** $X[1..i]$
else //see Problem 9 in this section's exercises
  for each element $x \in S_{i+1}$ consistent with $X[1..i]$ and the constraints **do**
    $X[i+1] \leftarrow x$
    Backtrack$(X[1..i+1])$

N-Queens problem

The problem is to place n queens on an n × n chessboard so that no two queens attack each other by being in the same row or in the same column or on the same diagonal.
So let us consider the **four-queens problem** and solve it by the backtracking technique. Since each of the four queens has to be placed in its own row, all we need to do is to assign a column for each queen on the board presented in figure.



We start with the empty board and then place queen 1 in the first possible position of its row, which is in column 1 of row 1. Then we place queen 2, after trying unsuccessfully columns 1 and 2, in the first acceptable position for it, which is square (2, 3), the square in row 2 and column 3. This proves to be a dead end because there is no acceptable position for queen 3. So, the algorithm backtracks and puts queen 2 in the next possible position at (2, 4). Then queen 3 is placed at (3, 2), which proves to be another dead end. The algorithm then backtracks all the way to queen 1 and moves it to (1, 2). Queen 2 then goes to (2, 4), queen 3 to(3, 1), and queen 4 to (4, 3), which is a solution to the problem. The state-space tree of this search is shown in figure.

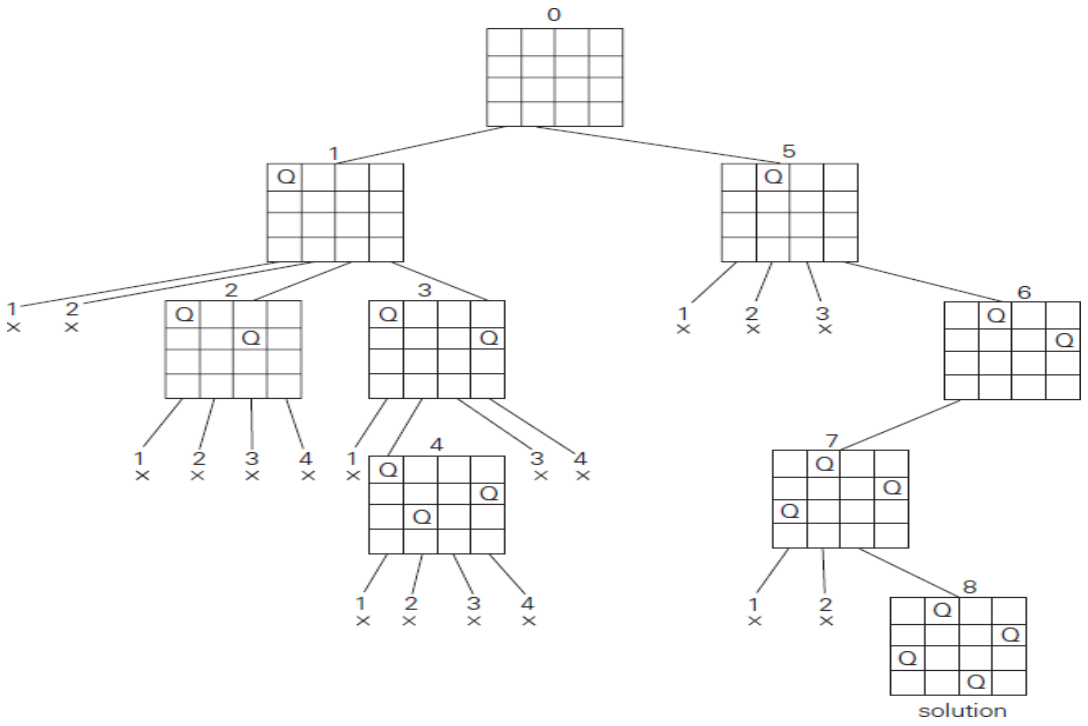


Figure: State-space tree of solving the four-queens problem by backtracking. × denotes an unsuccessful attempt to place a queen in the indicated column. The numbers above the nodes indicate the order in which the nodes are generated.

If other solutions need to be found, the algorithm can simply resume its operations at the leaf at which it stopped. Alternatively, we can use the board's symmetry for this purpose.

Finally, it should be pointed out that a single solution to the n-queens problem for any n ≥ 4 can be found in **linear time**.

Note: The algorithm NQueens() is not in the syllabus. It is given here for interested learners. The algorithm is referred from textbook T2.

```
Algorithm NQueens(k, n)
// Using backtracking, this procedure prints all
// possible placements of n queens on an n × n
// chessboard so that they are nonattacking.
{
    for i := 1 to n do
    {
        if Place(k, i) then
        {
            x[k] := i;
            if (k = n) then write (x[1 : n]);
            else NQueens(k + 1, n);
        }
    }
}
```

```
Algorithm Place(k, i)
// Returns true if a queen can be placed in kth row and
// ith column. Otherwise it returns false. x[ ] is a
// global array whose first (k − 1) values have been set.
// Abs(r) returns the absolute value of r.
{
    for j := 1 to k − 1 do
        if ((x[j] = i) // Two in the same column
            or (Abs(x[j] − i) = Abs(j − k)))
                // or in the same diagonal
            then return false;
    return true;
}
```

Sum of subsets problem

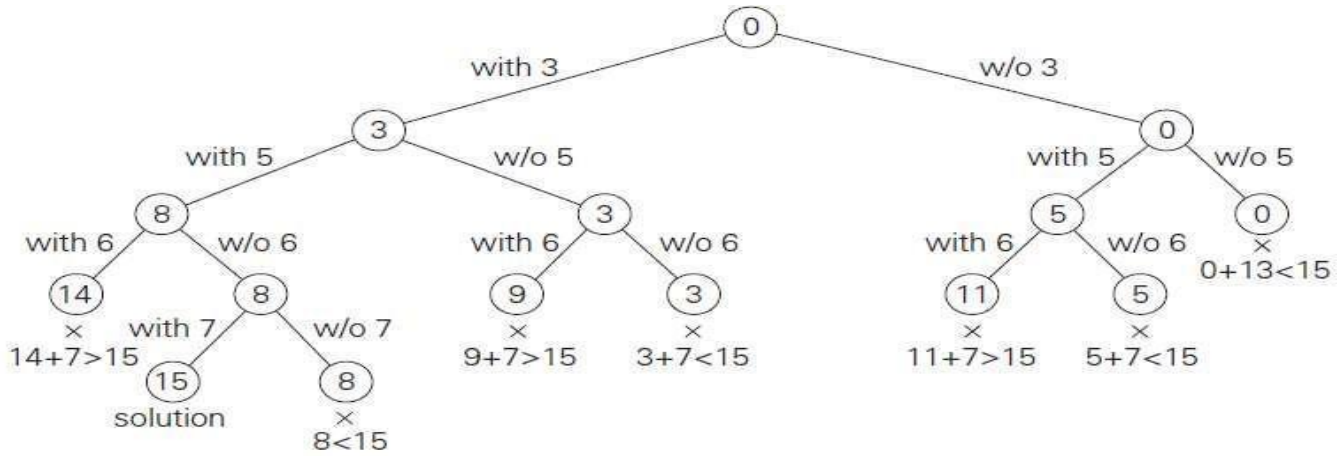
Problem definition: Find a subset of a given set A = {$a_1, \ldots, a_n$} of n positive integers whose sum is equal to a given positive integer d.

For example, for A = {1, 2, 5, 6, 8} and d = 9, there are two solutions: {1, 2, 6} and {1, 8}. Of course, some instances of this problem may have no solutions.

It is convenient to sort the set's elements in increasing order. So, we will assume that $a_1 < a_2 < \ldots < a_n$.

The state-space tree can be constructed as a binary tree like that in Figure shown below for the instance A = {3, 5, 6, 7} and d = 15.

The number inside a node is the sum of the elements already included in the subsets represented by the node. The inequality below a leaf indicates the reason for its termination.



The root of the tree represents the starting point, with no decisions about the given elements made as yet. Its left and right children represent, respectively, inclusion and exclusion of a_1 in a set being sought.

Similarly, going to the left from a node of the first level corresponds to inclusion of a_2 while going to the right corresponds to its exclusion, and so on. Thus, a path from the root to a node on the i^{th} level of the tree indicates which of the first in numbers have been included in the subsets represented by that node.

We record the value of s, the sum of these numbers, in the node. If s is equal to ***d***, we have a solution to the problem. We can either report this result and stop or, if all the solutions need to be found, continue by backtracking to the node's parent. If s is not equal to ***d***, we can terminate the node as non-promising if either of the following two inequalities holds:

$$s + a_{i+1} > d \quad \text{(the sum } s \text{ is too large)},$$

$$s + \sum_{j=i+1}^{n} a_j < d \quad \text{(the sum } s \text{ is too small)}.$$

Example: Apply backtracking to solve the following instance of the subset sum problem: A = {1, 3, 4, 5} and d = 11.

Graph coloring

Let G be a graph and m be a given positive integer. We want to discover whether the nodes of G can be colored in such a way that no two adjacent nodes have the same color yet only m colors are used. This is termed the *m-colorability decision* problem and it is discussed again in Chapter 11. Note that if d is the degree of the given graph, then it can be colored with $d+1$ colors. The *m-colorability optimization* problem asks for the smallest integer m for which the graph G can be colored. This integer is referred to as the *chromatic number* of the graph. For example, the graph of Figure 7.11 can be colored with three colors 1, 2, and 3. The color of each node is indicated next to it. It can also be seen that three colors are needed to color this graph and hence this graph's chromatic number is 3.

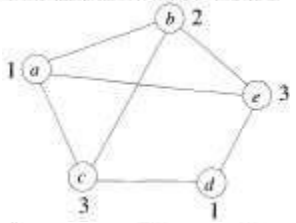


Figure 7.11 An example graph and its coloring

A graph is said to be *planar* iff it can be drawn in a plane in such a way that no two edges cross each other. A famous special case of the *m*-colorability decision problem is the 4-color problem for planar graphs. This problem asks the following question: given any map, can the regions be colored in such a way that no two adjacent regions have the same color yet only four colors are needed? This turns out to be a problem for which graphs are very useful, because a map can easily be transformed into a graph. Each region of the map becomes a node, and if two regions are adjacent, then the corresponding nodes are joined by an edge. Figure 7.12 shows a map with five regions and its corresponding graph. This map requires four

colors. For many years it was known that five colors were sufficient to color any map, but no map that required more than four colors had ever been found. After several hundred years, this problem was solved by a group of mathematicians with the help of a computer. They showed that in fact four colors are sufficient. In this section we consider not only graphs that are produced from maps but all graphs. We are interested in determining all the different ways in which a given graph can be colored using at most m colors.

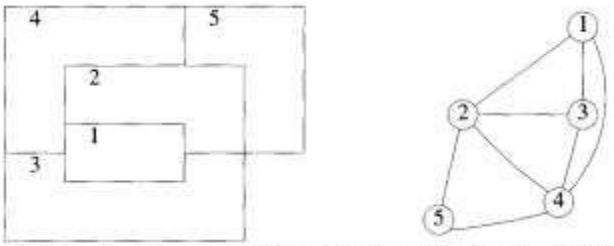


Figure 7.12 A map and its planar graph representation

Suppose we represent a graph by its adjacency matrix $G[1:n, 1:n]$, where $G[i,j] = 1$ if (i,j) is an edge of G, and $G[i,j] = 0$ otherwise. The colors are represented by the integers $1, 2, \ldots, m$ and the solutions are given by the n-tuple $(x_1, \ldots, x_n)$, where x_i is the color of node i. Using the recursive backtracking formulation as given in Algorithm 7.1, the resulting algorithm is mColoring (Algorithm 7.7). The underlying state space tree used is a level $n+1$ are leaf nodes. Figure 7.13 shows the state space tree when $n = 3$ and $m = 3$.

Algorithm 7.7 Finding all m-colorings of a graph

```
Algorithm mColoring(k)
// This algorithm was formed using the recursive backtracking
// schema. The graph is represented by its boolean adjacency
// matrix G[1:n, 1:n]. All assignments of 1, 2, ..., m to the
// vertices of the graph such that adjacent vertices are
// assigned distinct integers are printed. k is the index
// of the next vertex to color.
{
    repeat
    {// Generate all legal assignments for x[k].
        NextValue(k); // Assign to x[k] a legal color.
        if (x[k] = 0) then return; // No new color possible
        if (k = n) then     // At most m colors have been
                            // used to color the n vertices.
            write (x[1:n]);
        else mColoring(k + 1);
    } until (false);
}
```

```
Algorithm NextValue(k)
// x[1], ..., x[k − 1] have been assigned integer values in
// the range [1, m] such that adjacent vertices have distinct
// integers. A value for x[k] is determined in the range
// [0, m]. x[k] is assigned the next highest numbered color
// while maintaining distinctness from the adjacent vertices
// of vertex k. If no such color exists, then x[k] is 0.
{
    repeat
    {
        x[k] := (x[k] + 1) mod (m + 1); // Next highest color.
        if (x[k] = 0) then return; // All colors have been used.
        for j := 1 to n do
        {   // Check if this color is
            // distinct from adjacent colors.
            if ((G[k, j] ≠ 0) and (x[k] = x[j]))
            // If (k, j) is and edge and if adj.
            // vertices have the same color.
                then break;
        }
        if (j = n + 1) then return; // New color found
    } until (false); // Otherwise try to find another color.
}
```

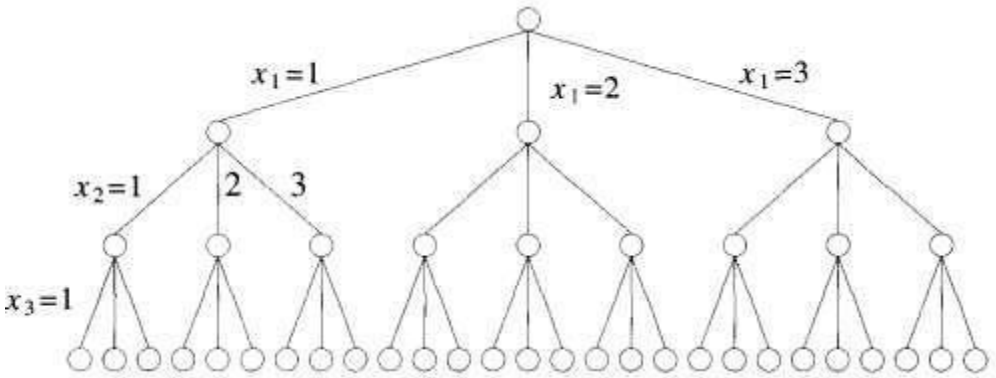
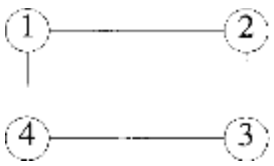


Figure 7.13 State space tree for mColoring when n =3 and m = 3

Function mColoring is begun by first assigning the graph to its adjacency matrix, *setting the array x[ ] to zero*, and then invoking the statement mColoring(1);.
recursive backtracking schema of Algorithm 7.1. Function NextValue (Algorithm 7.8) produces the possible colors for x_k after x_1 through x_{k-1} have been defined. The main loop of mColoring repeatedly picks an element from the set of possibilities, assigns it to x_k, and then calls mColoring recursively. For instance, Figure 7.14 shows a simple graph containing four nodes. Below that is the tree that is generated by mColoring. Each path to a leaf represents a coloring using at most three colors. Note that only 12 solutions exist with *exactly* three colors. In this tree, after choosing $x_1 = 2$ and $x_2 = 1$, the possible choices for x_3 are 2 and 3. After choosing $x_1 = 2$, $x_2 = 1$, and $x_3 = 2$, possible values for x_4 are 1 and 3. And so on.



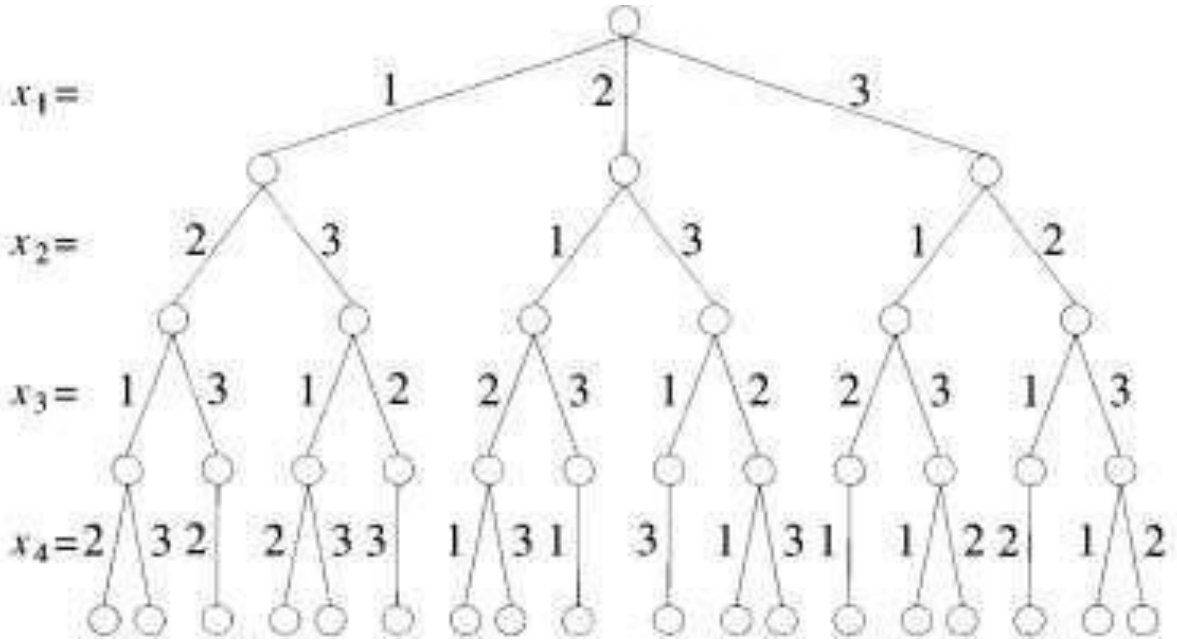


Figure 7.14 A 4-node graph and all possible 3-colorings

Analysis

An upper bound on the computing time of mColoring can be arrived at by noticing that the number of internal nodes in the state space tree is $\sum_{i=0}^{n-1} m^i$. At each internal node, $O(mn)$ time is spent by NextValue to determine the children corresponding to legal colorings. Hence the total time is bounded by $\sum_{i=0}^{n-1} m^{i+1}n = \sum_{i=1}^{n} m^i n = n(m^{n+1} - 2)/(m-1) = O(nm^n)$.