CCS 4201 DATA STRUCTURES & ALGORITHMS
Unit Lecturer: Mwangi, Edwin

1.1 Introduction Definitions

Data types define or classify the type of values a variable can store in it. Describes the possible operations allowed on those values. For example, the integer data type can store an integer value. Possible operations on an integer include addition, subtraction, multiplication etc.

1.1.1 **Data structure** is a systematic way of organizing and accessing data in memory

- A data structure is an arrangement of data in a computer's memory or in disk storage. An example of several common data structures are arrays, records, lists, queues, stacks, binary trees, and hash tables.

Representation of data structure

Data structure = Organized data + Allowed operations

- The study of data structures deals with the following

i. Logical description of the data structure
ii. Implementation of the data structure
iii. Quantitive analysis of the data structure

Need of Data Structure –

- Applications are becoming more complex and the amount of data increasing day by day, causing problems with processing speed, searching data, handling multiple requests etc.
- Data structure provides a way of organizing, managing, and storing data efficiently.
- data structure helps to traverse data items easily.
- Data structure provides efficiency, reusability and abstraction.
- Data structure enhance the performance of programs.

Types of Data Structure –

There are 2 types of Data Structure:

- Primitive Data Structure
- Non – Primitive Data Structure

Primitive Data Structure / Atomic data structure

Primitive Data Structures directly operate according to the machine instructions i.e. primitive data types/data structures like int, char, float, double, that can hold a single value.

Non Primitive Data Structure / Structured Data Structure

Complex data structures that are derived from primitive data structures. Non – Primitive data types are further divided into two categories.

- Linear Data Structure
- Non – Linear Data Structure

    - **Linear Data structure**

- The elements forms a sequence, i.e. Linear list, Examples of linear a data structures are arrays, linked lists, stacks and queues.

    - Non linear Data structures.

- The elements do not form a sequence. Examples of non linear data structures are trees and graphs.

Further Classification of Data Structure –

Data Structure can be further classified as

- Static Data Structure
- Dynamic Data Structure

Static Data Structure –

Data structures where the size is allocated at the compile time, Hence, the maximum size is fixed and cannot be changed. i.e Array

Dynami**c Data Structure –**

Dynamic Data Structures are data structures where the size is allocated at the run time. Hence, the maximum size is flexible and can be changed as per requirement i.e Linked Lists.

Data Structure Operations –

The common operations that can be performed on the data structures are as follows :

- Searching – We can easily search for any data element in a data structure.
- Sorting – We can sort the elements either in ascending or descending order.
- Insertion – We can insert new data elements in the data structure.
- Deletion – We can delete the data elements from the data structure.
- Updation – We can update or replace the existing elements from the data structure

Advantages of Data Structure –

- Data structures allow storing the information on hard disks.
- Appropriate choice of ADT (Abstract Data Type) makes the program more efficient.
- Data Structures are necessary for designing efficient algorithms.
- It provides reusability and abstraction.
- Using appropriate data structure, can help programmers save a good amount of time while performing operations such as storage, retrieval or processing of data.

- Manipulation of large amounts of data is easier

Goals for designing Data structures.

- To design an algorithm that is easy to understand, code, and debug.
- To design an algorithm that makes efficient use of the computer's resources.

1.2.2 Memory allocation in a Data Structure

Memory allocation is the process of setting aside sections of memory in a program to be used to store variables, and instances of structures and classes. There are two basic types of memory allocation:

- Static memory allocation.
  - Declaration of variables, structures and classes at the beginning of a class or function. Examples Int x; char C; Int Tom(3);
- Dynamic memory allocation
  - Memory allocated while the program is running i.e using pointers

Examples: int *ptr; //pointer declaration

.

1.2.3 Abstract data types (ADT)

A mathematical model for data types.

Abstract Data Type (abbreviated *ADT*) is a collection of data components, together with a collection of operations that are permitted on those data components..

Example, a List is an abstract data type that is implemented using a dynamic array

Properties of ADT

i. Generalization
ii. Encapsulation
iii. Abstraction: It is a technique of hiding the internal details from the user and only showing the necessary details to the user.
iv. Encapsulation: It is a technique of combining the data and the member function in a single unit is known as encapsulation.

1.2.4 Algorithm

- A step-by-step procedure for performing some task in a finite amount of time
- *Algorithms* are used to manipulate the data contained in data structures as e.g. searching and sorting.

Properties of an algorithm

i. It must be correct i.e. correctly solve a problem producing the correct output from its given input.
ii. Its instructions must be clear and unambiguous
iii. It must be composed of a series of concrete steps. [Executable by that machine]
iv. It must be composed of a finite number of steps.
v. It must terminate.

vi. Ensure correctness and efficiency.

Categories of Algorithmic Operations

- An algorithm must have the ability to alter the order of its instructions by use of control structures as follows.

1. **sequential operations -** instructions are executed in order
2. **conditional / selection operations** - a control structure that asks a true/false question and selects the next instruction based on the answer
3. **iterative operations (loops)** - a control structure that repeats the execution of a block of instructions

Ways of algorithms implementation

- Flowcharts
- Pseudo codes. Natural language constructs modeled to look like statements available in many programming languages

Examples

- Flow chart that prints all odd numbers between 0 and 100,
- Flow chart that prints Sum of all odd numbers between 0 and 100
- Other examples discussed in class

3.1 Running Time Analysis of algorithms

Determine the amount of resources (such as time and storage) necessary to execute it, *to measure the efficiency of a given algorithm*

Analysis of **algorithms** is the determination of the computational complexity of **algorithms**, that is the amount of time, storage and/or other resources necessary to execute them.

Significant in Terms of Time Complexity

The input size has a strong relationship with time complexity. As the size of the input increases, so does the runtime, or the amount of time it takes the algorithm to run.

Here is an example.

Assume you have a set of numbers S= (10, 50, 20, 15, 30)

There are numerous algorithms for sorting the given numbers. However, not all of them are effective. To determine which is the most effective, you must perform computational analysis on each algorithm.

Why Analyze Algorithms?

- We analyze algorithms with the intention of *improving* them,
- To improve system performance
- discover its characteristics in order to evaluate its suitability for various applications or compare it with other algorithms for the same application
- help us understand it better, and can suggest informed improvements.

- To analyze Amount of work done, and space used

T(n), or the running time of a particular algorithm on input of size n, is the number of times the instructions in the algorithm are executed.

Factors affecting the running time

a. The size of the input (searching through a list of length 1,000 takes longer than searching through a list of length 10),
b. Speed of the machine running the program
c. Efficiency of the compiler that created the program
d. The algorithm used to solve the problem (Unordered-Linear-Search is slower than Binary-Search),
e. Organization of the input: i.e. if the item being searched is at the top of the list, it will take less time to find it than if it is at the bottom.
f. The programming language used to implement the algorithm (interpreted languages such as Basic are typically slower than compiled languages such as C++),
g. The quality of the actual implementation (good, tight code can be much faster than poor, sloppy code),
h. The machine on which the code is Factors that affects the running time of an algorithm

Types of algorithm analysis

i. Worst
ii. Average
iii. Best-case

Worst-case running time of an algorithm

- executing the algorithm produces path lengths that are always a maximum, i.e. The longest running time for **any** input of size n

$T(n)$ = maximum time of algorithm on any input of size n.

- Example: Sort a set of numbers in increasing order; and the data is in decreasing order
  E.g. in searching a database for a particular piece of information

Best-case running time

- if the algorithm is executed, the fewest number of instructions are executed

e.g. sorting a set of numbers in increasing order; and the data is already in increasing order

Average-case running time

- executing the algorithm produces path lengths that will on average be the same

$T(n)$ = expected time of algorithm over all inputs of size n.

Examples

1: Pseudo code algorithm that illustrates the calculation of the mean (average) of a set of n numbers:

1. n = read input from user
2. sum = 0
3. i = 0
4. while i < n
5. number = read input from user
6. sum = sum + number
7. i = i + 1
8. mean = sum / n

Calculate the computing time for this algorithm in terms of input size n.
N.B totaling the counts produces the F.C. (frequency count)
Solution; While Condition tested first before the statements within the loop

Statement	Number of times executed
1	1
2	1
3	1
4	n+1
5	n
6	n
7	n
8	1

T(n) = 4n + 5.

- $T(n)$ = maximum time of algorithm on any input of size n.

Statement	s/e	Frequency	Total steps
Algorithm sum(list, n)	0	0	0
{	0	0	0
s := 0;	1	1	1
i:=0;	1	1	1
for i := 1 to n do	1	n+1	n+1
s:= s+ list[i];	1	n	n
return s;	1	1	1
}	0	0	0
Total			2n+4

Statement	s/e	Frequency	Total steps
Algorithm add (a, b, c, m,n)	0	0	0
{	0	0	0
for i:= 1 to m do	1	m+1	m+1
for j:= 1 to n do	1	m • (n+1)	mn+m
c[i][j] := a[i][j] + b[i][j];	1	m • n	mn
}	0	0	0
Total			2mn+2m+1

Statement	s/e	Frequency	Total steps

Algorithm add (a, b, c, m,n)
{
 	
for i:= 1 to m do
for j:= 1 to n do
c[i][j] := a[i][j] + b[i][j];
}
	0
	0
	
	1
	1
	1
	0
		0
		0
		
		m+1
		m • (n+1)
		m • n
		0
			0
			0
			
			m+1
			mn+m
			mn
			0

Total			2mn+2m+1

Statement	s/e	Frequency	Total steps

Algorithm mult(a, b, c, M)
{
for i:= 1 to M do
for j:= 1 to M do
{ c[i][j] = 0;
for i:= k to M do
c[i][j] := c[i][j] + a[i][k] * b[k][j];
}
}
	0
	0
	1
	1
	1
	1
	0
	0
	0
		0
		0
		M+1
		M•(M+1)
		M. M
		M.M.(M+1)
		M.M.M
		0
		0
			0
			0
			M + 1
			M^2 +M
			M^2
			M^3 +M^2
			M^3
			0
			0

Total			$2M^3+3M^2+2M+1$

Compute the running time of some example programs.

```
int sum(int a, int b) {
    int c = a + b;
    return c
}
```

Analysis

The `sum` function has two statements. The first statement (line 2) runs in constant time i.e. *Theta*(1) and second statement (line 3) also runs in constant time $\Theta(1)$. These two statements are consecutive statements, so the total running time is $\Theta(1)+\Theta(1)=\Theta(1)$

Example 2

```
1 int array_sum(int a, int n) {
2     int i;
3     int sum = 0;
4     for (i = 0; i < n; i++) {
5         sum = sum + a[i]
6     }
7     return sum;
8 }
```

Analysis

Line 2 is a variable declaration. The cost is $\Theta(1)$
Line 3 is a variable declaration and assignment. The cost is $\Theta(2)$
Line 4 - 6 is a `for` loop that repeats n times. The body of the for loop requires $\Theta(1)$ to run. The total cost is $\Theta(n)$
Line 7 is a `return` statement. The cost is $\Theta(1)$

1, 2, 3, 4 are consecutive statements so the overall cost is $\Theta(n)$

Example 3

```
1  int sum = 0;
2  for (i = 0; i < n; i++) {
3      for (j = 0; j < n; j++) {
4          for (k = 0; k < n; k++) {
5              if (i == j == k) {
6                  for (l = 0; l < n*n*n; l++) {
7                      sum = i + j + k + l;
8                  }
9              }
10         }
11     }
12 }
```

Analysis

Line 1 is a variable declaration and initialization. The cost is $\Theta(1)$

Line 2 - 11 is a nested for loops. There are four `for` loops that repeat n times. After the third `for` loop in Line 4, there is a condition of `i == j == k`. This condition is true only n times. So the total cost of these loops is $\Theta(n^3)+\Theta(n^4)=\Theta(n^4)$

The overall cost is $\Theta(n^4)$

Types of Notations:

1. Omega(Ω) Notation

The **notation** Ω(n) is the formal way to express the lower bound of an algorithm's running time. It measures the best case time complexity or the best amount of time an algorithm can possibly take to complete.

For eg : let's say we are sorting 1000 number, so the minimum time required to sort 1000 numbers is "*10 secs"*. In *omega notation,* those 1000 numbers cannot be sorted in less than 10 secs. It can be 11 secs, 12 secs but not 9 or 8 secs.

2. Big-o(O) Notation :

The **big-o** is the formal way to express the upper bound of an algorithm's running time. It measures the worst case time complexity or the maximum amount of time an algorithm can possibly take to complete.

For eg : let's say we are sorting 1000 number again, so the maximum time required to sort 1000 numbers is "50 *secs"*. In *big-o notation,* those 1000 numbers can be sorted in less than 50 secs. It can be 48 secs, 46 secs but not 51 or 52 secs.

3. Theta(Θ) Notation :

The **theta notation** bounds a functions from above and below, so it defines exact asymptotic behavior. For any given input, the running time of a given algorithm will "on an average" be equal to given time.

For eg : let's take again the example of sorting 1000 numbers, for the first time the algo takes 10 secs, if we sort it again it take 11 secs, and again if we sort for the 3rd time it takes 7 secs.

So the *theta notation* will say that the algorithm "on an average" take *9 secs* to execute the method.

2.2 Algorithm Complexity

The choice of a particular algorithm depends on the following consideration.

i) Time complexity

ii) Space complexity

Time complexity

- This is the amount of time an algorithm needs to run to completion.

Space complexity

- This is the amount of memory an algorithm needs to run to completion.

2.4 Selecting a Data Structure

1. Analyze the problem to determine the resource constraints.
2. Determine the basic operations that must be supported.
3. Select the data structure that best meets these requirements.

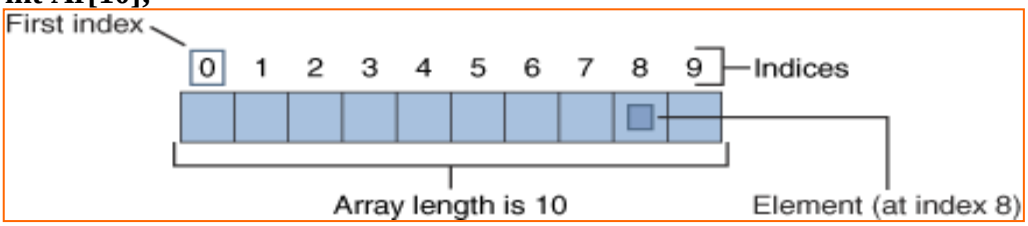
3.0 Arrays

An Arrays is a **physically sequential**, fixed size collection of homogeneous objects.
In addition, objects with the structure have the random access property.

Properties / characteristics of an array

I. The array is physically sequential i.e. data objects are stored in consecutive memory locations.
II. The array has a fixed size. i.e. its size can neither be increased nor reduced though the number of items it contains can vary
III. The array is homogeneous i.e. made up of objects that are all the same type.
IV. The random access property i.e. time it takes to access one object in the structure does not depend on what object in the structure had been accessed previously.

int Ar[10];



N.B Array index are always numbered starting at 0

Note:
An array type is appropriate for representing an abstract data type when the following three conditions are satisfied:

- The data objects in the abstract data type are composed of homogeneous objects
- The solution requires the representation of a fixed, predetermined number of objects
- There is an ordering that can be placed on the objects in the abstract data type

Initialization of an Array

Array declaration by specifying size

```
// Array declaration by specifying size
int arr1[10];
```

Array declaration by initializing elements

```
// Array declaration by initializing elements
int arr[] = { 10, 20, 30, 40 }

// Compiler creates an array of size 4.
// above is same as  "int arr[4] = {10, 20, 30, 40}
```

Array declaration by specifying size and initializing elements

```
// Array declaration by specifying size and initializing
// elements
int arr[6] = { 10, 20, 30, 40 }

// Compiler creates an array of size 6, initializes first
// 4 elements as specified by user and rest two elements as
// 0. above is same as  "int arr[] = {10, 20, 30, 40, 0, 0}"
```

Int expenses [12]

Declares an array of type integer. The array name expenses, and contains 12 elements, the 12 elements are numbered 0-11

NB when you will give more initializer (array elements) than the declared array size than

Examples
Tom [1] = 50, stores the value 50 in the second array element
Example 2
Declare an array to contain 5 integer values of type int called Tom.
int Tom [5]

```
int Tom [5] = { 16, 2, 77, 40, 12 };
```

	0	1	2	3	4
billy	16	2	77	40	12071

3.1.3 Subscripting

- Subscripts are used to access an individual element in an array.
- A subscript is a bracketed expression and the expression in the brackets is known as the index. i.e. name[index]

- Example 1
Write a statement that store the value 75 in the third element of in array tom, :

```
Tom[2] = 75;
```

- Example2

A statement that passes the value of the third element of tom to a variable called a,

```
a = tom[2];
```

N.B. Array uses [ ] bracket to perform two different tasks:

- One is to specify the size of the arrays when they are declared;
- The second use is to specify indices for concrete array elements.

Example

```
1 int billy[5];         // declaration of a new array
2 billy[2] = 75;      // access to an element of the array.
                          Store a value to an array element
```

Basic Operations

Following are the basic operations supported by an array.

- **Traverse** − print all the array elements one by one.

- **Insertion** − Adds an element at the given index.
- **Deletion** − Deletes an element at the given index.
- **Search** − Searches an element using the given index or by the value.
- **Update** − Updates an element at the given index.

Program to traverse an Array

Given an integer array of size **N**, the task is to traverse and print the elements in the array.
Examples:

Input: arr[] = {2, 1, 5, 6, 0, 3}
Output: 2 1 5 6 0 3

Approach: -

1. Start a loop from 0 **to** N-1**, where** N **is the size of array.**

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

2. Access every element of array with help of

arr[index]

3. Print the elements.

printf("%d ", arr[i])

Example.

```
#include <stdio.h>
int main()
{
   int i;
   int arr[3] = {2,3,4};

   for (i=0; i<3; i++)

         printf("[%d]", arr[i]);

}
```

Insert an element in an Array

Given an array **arr** of size **n** insert an element **x** in array **arr** at a specific position **pos**.

Approach:

1. First get the element to be inserted, say x

2. Then get the position at which this element is to be inserted, say pos
3. Then shift the array elements from this position to one position forward, and do this for all the other elements next to pos.
4. Insert the element x now at the position pos, as this is now empty.

```
#include <stdio.h>

int main()
{
   int arr[10];
   int i, x, pos, n = 10;

   // initial array of size 10
   for (i = 0; i < 10; i++)
      arr[i] = i + 1;

   // print the original array
   for (i = 0; i < n; i++)
      printf("%d ", arr[i]);
   printf("\n");

   // element to be inserted
   x = 50;

   // position at which element is to be inserted
   pos = 5;

   // increase the size by 1
   n++;

   // shift elements forward
   for (i = n-1; i >= pos; i--)
      arr[i] = arr[i - 1];

   // insert x at pos
   arr[pos - 1] = x;

   // print the updated array
   for (i = 0; i < n; i++)
      printf("%d ", arr[i]);
   printf("\n");

   return 0;
}
```

Output:

1 2 3 4 5 6 7 8 9 10
1 2 3 4 50 5 6 7 8 9 10

Deletion Operation

Removing an existing element from the array.

Algorithm to delete an element from an array

1. Find the given element in the given array and note the index.

2. If the element found,

   Shift all the elements from index + 1 by 1 position to the left.

   Reduce the array size by 1.

3. Otherwise, print "Element Not Found"

Implementation of the above algorithm –

```
#include <stdio.h>

int main() {
  int LA[] = {1,3,5,7,8};
  int k = 3, n = 5;
  int i, j;

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

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

  j = k;

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

  n = n -1;

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

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

Output

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

Search Operation

Perform a search for an array element based on its value or its index.

Algorithm

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

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

Implementation of the above algorithm

```
#include <stdio.h>

int main() {
  int LA[] = {1,3,5,7,8};
  int item = 5, n = 5;
  int i = 0, j = 0;
```

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

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

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

    j = j + 1;
  }

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

Output

The original array elements are :
LA[0] = 1
LA[1] = 3
LA[2] = 5
LA[3] = 7
LA[4] = 8
Found element 5 at position 3

Update Operation

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

Algorithm

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

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

implementation of the above algorithm

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

```
void main() {
  int LA[] = {1,3,5,7,8};
  int k = 3, n = 5, item = 10;
  int i, j;

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

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

  LA[k-1] = item;

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

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

Output

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

Arrays:Disadvantages

The array implementation has drawback: Drawback:

- capacity of array fixed
- must know max number of values at compile time I nt tom[5];
- either the program runs out of space or wastes space

Lists ADT

The linked list is a linear data structure where each node has two parts Data and Reference to the next node

Data: we can store the required information. It can be any data type.

Example

```
int age;
```

Reference to the next node hold the next nodes address.***a type pointer.***

Head Node - Starting node of a linked list.

Last Node - Node with reference pointer as NULL.

Sample Linked List Node

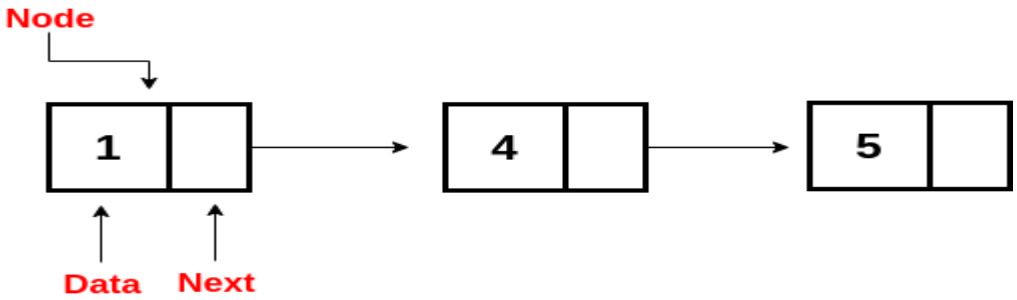
```
struct node
{
    int data;
    struct node *next;
};
```

where,

data - used to store the integer information.

struct node *next - It is used to refer to the next node. It will hold the address of the next node.

N.B: Structure is a data type used to group different data types. Every node in a linked list is a structure data type. Holds different data types (heterogeneous).



A linked list that create and allocate memory for 3 nodes

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

```
struct node *head,*middle,*last;

head   = malloc(sizeof(struct node));
middle = malloc(sizeof(struct node));
last   = malloc(sizeof(struct node));
```

N.B: *malloc*" or "memory allocation" method used to dynamically allocate a single block of memory with the specified size.

Assign values to each node

```
head->data   = 10;
middle->data = 20;
last->data   = 30;
```

N.B: -> operator used to access a member of a struct which is referenced by the pointer in question.

Linking each nodes

headnode ➜ middlenode ➜ lastnode ➜ NULL

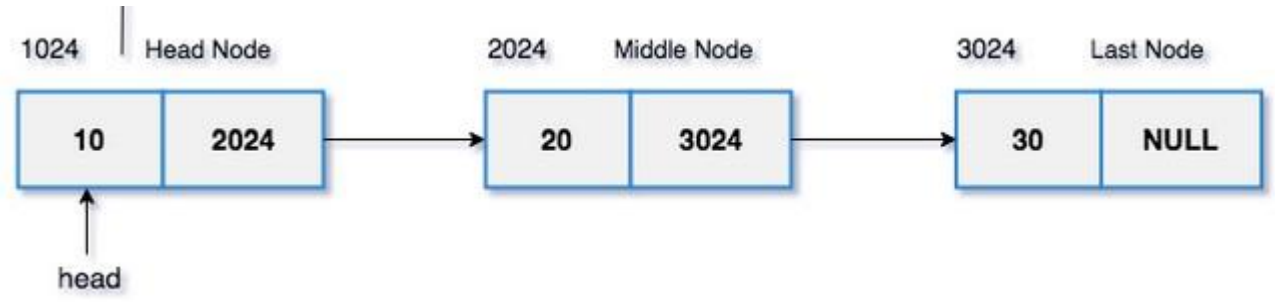
```
head->next   = middle;
middle->next = last;
last->next   = NULL;      //NULL indicates the end of the
linked list
Note:
```

head => next = middle. Hence head => next holds the memory address of the middle node (2024).

middle => next = last. Hence middle => next holds the memory address of the last node (3024)

last => next = NULL which indicates it is the last node in the linked list

Example



printing each node data in a linked list

To print each node's data, traverse the linked list till the end.

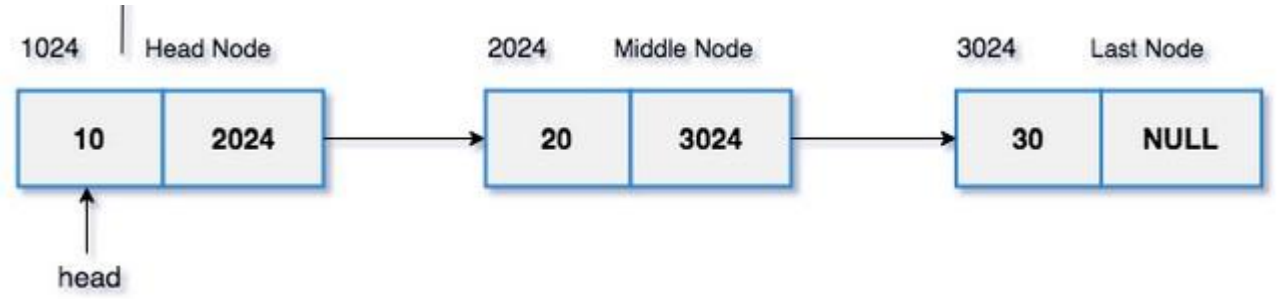
Algorithm

1. Create a temporary node(temp) and assign the head node's address.

2. Print the data which present in the temp node.

3. After printing the data, move the temp pointer to the next node.

4. Do the above process until we reach the null



1. temp points to the head node. temp => data = 10 will be printed. temp will point to the next node (Middle Node).

2. temp != NULL. temp => data = 20 will be printed. Again temp will point to the next node (Last Node).

3. temp != NULL. temp => data = 30 will be printed. Again temp will point to the next node which is NULL.

4. temp == NULL. Stop the process we have printed the whole linked list.

Code

```
struct node *temp = head;

while(temp != NULL)
{
    printf("%d ",temp->data);
    temp = temp->next;
}
```

Example

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

```c
#include<stdlib.h>

int main()
{
  //node structure
  struct node
  {
      int data;
      struct node *next;
  };

  //declaring nodes
  struct node *head,*middle,*last;

  //allocating memory for each node
  head   = malloc(sizeof(struct node));
  middle = malloc(sizeof(struct node));
  last   = malloc(sizeof(struct node));

  //assigning values to each node
  head->data   = 10;
  middle->data = 20;
  last->data   = 30;

  //connecting each nodes. head->middle->last
  head->next   = middle;
  middle->next = last;
  last->next   = NULL;

  //temp is a reference for head pointer.
  struct node *temp = head;

  //till the node becomes null, printing each nodes
data
  while(temp != NULL)
  {
      printf("%d->",temp->data);
      temp = temp->next;
  }
  printf("NULL");

  return 0;
}
```

Properties of a list Data Structure.

- The first node of the linked list is called the **head of the linked list**.
- The last node of the linked list is pointing to **NULL**(None) which indicates that it is the last node.
- Unlike arrays, linked list elements are not stored at contiguous memory locations.
- Linked Lists addresses some of the limitations of arrays of having a fixed size because Linked Lists are dynamic in nature.

scenarios where linked lists are used.

Linked List is used:

- When it is critical to maintain the time taken for insertion or deletion of elements.
- When the number of elements is not certain, a linked list is preferred.
- Linked List is used to store elements when random access to any elements inside the list is not needed.
- When it is required to insert elements at the middle of the list, a Linked list is preferred.

Advantages of Linked Lists

- They are a dynamic in nature which allocates the memory when required.
- Insertion and deletion operations can be easily implemented.
- Stacks and queues can be easily executed.

Disadvantages of Linked Lists

- The memory is wasted as pointers require extra memory for storage.
- No element can be accessed randomly; it has to access each node sequentially.
- Reverse Traversing is difficult in linked list though achieved with the help of Doubly Linked List).

Some of the drawbacks are:

1. Random access is not allowed. Elements are accessed sequentially starting from the first node. A Binary search cannot be performed with linked lists.
2. Pointer requires more space with each element in the list.
3. Poor locality since the memory used for the linked list is scattered.

Applications of Linked Lists

- Linked lists are used to implement stacks, queues, graphs, etc.
- Any application which has to deal with an unknown number of objects will need to use a linked list.

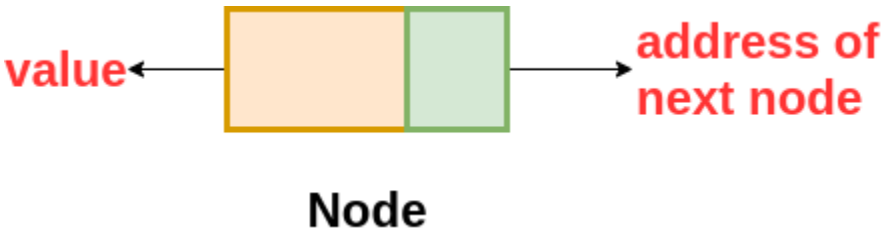
Types of Linked List

- **Singly 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.

Singly Linked List

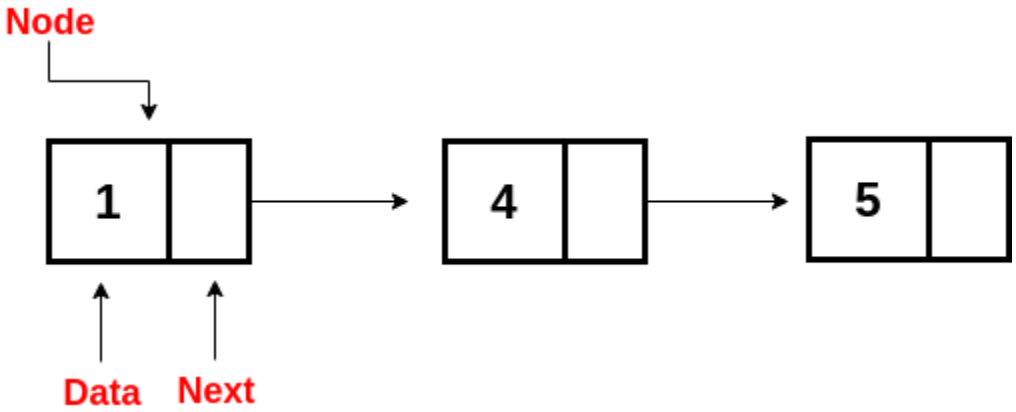
A Singly-linked list is a collection of nodes linked together in a sequential way where each node of the singly linked list contains a data field and an address field that contains the reference of the next node.

The structure of a Singly Linked List




```
class Node {
    int data // variable to store the data of the node
    Node next // variable to store the address of the next node
}
```

The value of the next variable of the last node is NULL i.e. ***next = NULL***, which indicates the end of the linked list.

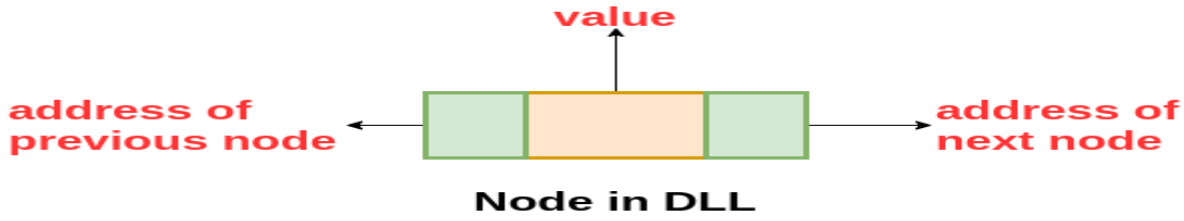


Doubly Linked List

A Doubly Linked List contains an extra memory to store the address of the previous node, together with the address of the next node and data which are there in the singly linked list.

Store the address of the next as well as the previous nodes.

The structure of a Doubly Linked List(DLL):

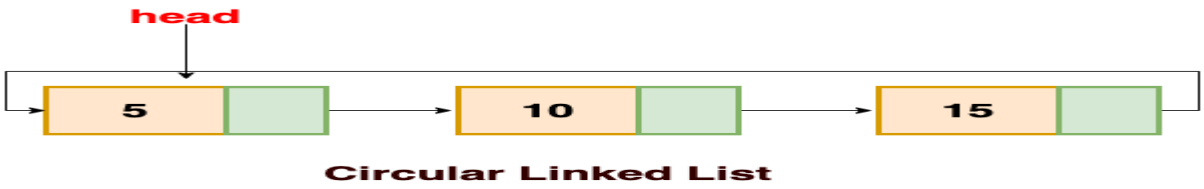



```
class DLLNode {
    int val // variable to store the data of the node
    DLLNode prev // variable to store the address of the previous node
    DLLNode next // variable to store the address of the next node
}
```

First node has **prev = NULL** and the last node has **next = NULL**.

Circular Linked List

A circular linked list is either a singly or doubly linked list in which there are no ***NULL*** values.



Basic Operations on Linked List

- **Traversal**: To traverse all the nodes one after another.
- **Insertion**: To add a node at the given position.
- **Deletion**: To delete a node.
- **Searching**: To search an element(s) by value.
- **Updating**: To update a node.
- **Sorting:** To arrange nodes in a linked list in a specific order.
- **Merging:** To merge two linked lists into one.

Linked List Traversal

Step through the list from beginning to end. ***For example***, we may want to print the list or search for a specific node in the list.

The algorithm for traversing a list

- Start with the head of the list. Access the content of the head node if it is not null.
- Then go to the next node(if exists) and access the node information
- Continue until no more nodes (that is, you have reached the null node)

```
void traverseLL(Node head) {
    while(head != NULL)
    {
        print(head.data)
        head = head.next
    }
}
```

Linked List Node Insertion

There can be three cases that will occur when we are inserting a node in a linked list.

- Insertion at the beginning
- Insertion at the end. (Append)
- Insertion after a given node

Insertion at the beginning

Since there is no need to find the end of the list. If the list is empty, we make the new node as the head of the list. Otherwise, connect the new node to the current head of the

list and make the new node, the head of the list.

```
// function is returning the head of the singly linked-list
Node insertAtBegin(Node head, int val)
{
    newNode = new Node(val) // creating new node of linked list
    if(head == NULL) // check if linked list is empty
        return newNode
    else // inserting the node at the beginning
    {
        newNode.next = head
        return newNode
    }
}
```

To add an item to the beginning of the list, you have to do the following:

- Create a new item and set its value
- Link the new item to point to the head of the list
- Set the head of the list to be our new item

Insertion at end

Traverse the list until we find the last node, insert the new node to the end of the list, consider special cases such as list being empty.

In case of a list being empty, we will return the updated head of the linked list because in this case, the inserted node is the first as well as the last node of the linked list.

```
// the function is returning the head of the singly linked list
Node insertAtEnd(Node head, int val)
{
    if( head == NULL ) // handing the special case
    {
        newNode = new Node(val)
        head = newNode
        return head
    }
    Node temp = head
    // traversing the list to get the last node
    while( temp.next != NULL )
    {
        temp = temp.next
    }
    newNode = new Node(val)
    temp.next = newNode
    return head
}
```

Linked List node Deletion

To delete a node from a linked list, we need to do these steps

- Find the previous node of the node to be deleted.
- Change the next pointer of the previous node
- Free the memory of the deleted node.

In the deletion, there is a special case in which the first node is deleted. In this, we need to update the head of the linked list.

```
// this function will return the head of the linked list
Node deleteLL(Node head, Node del)
{
    if(head == del) // if the node to be deleted is the head node
    {
        return head.next // special case for the first Node
    }
    Node temp = head

    while( temp.next != NULL )
    {
        if(temp.next == del) // finding the node to be deleted
        {
            temp.next = temp.next.next
            delete del // free the memory of that Node
            return head
        }
        temp = temp.next
    }
```

```
    return head // if no node matches in the Linked List
}
```

Difference between Array and Linked List

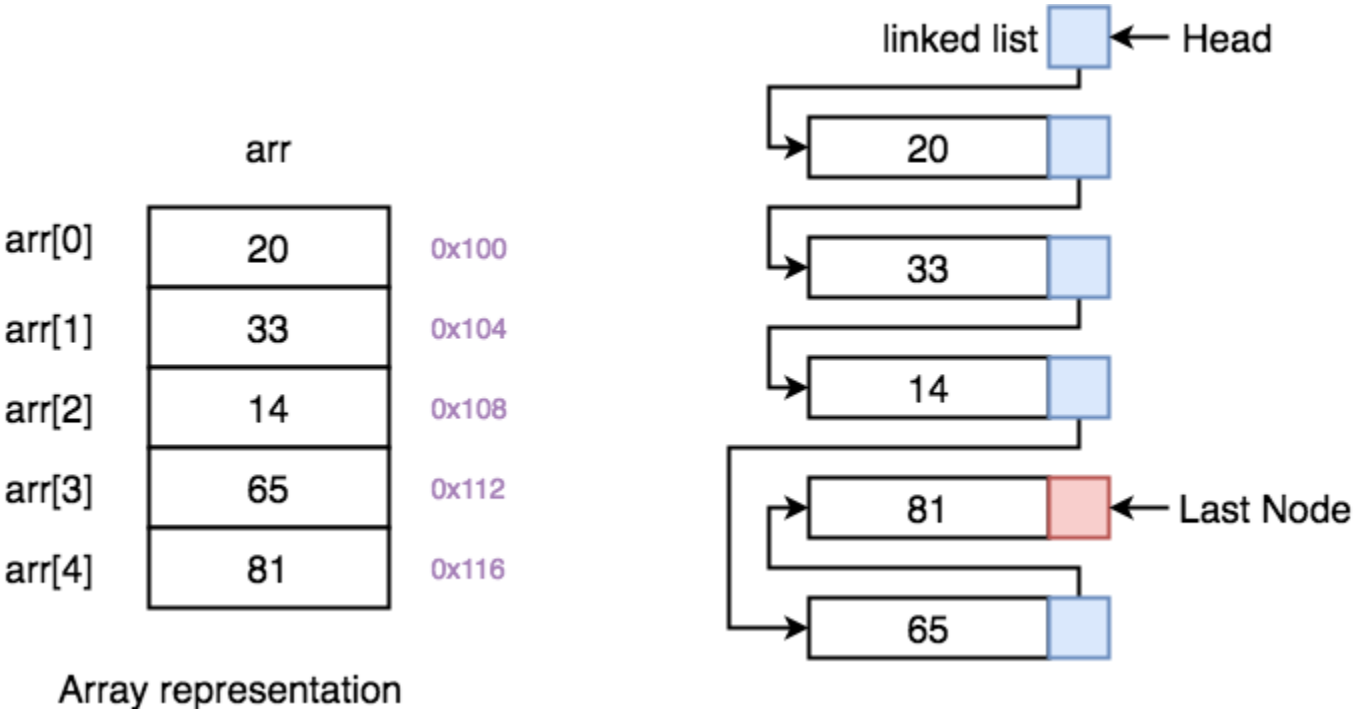
Both Linked List and Array are used to store linear data of similar type, but an array consumes contiguous memory locations allocated at compile time, i.e. at the time of declaration of array,

A linked list, memory is assigned as and when data is added to it, which means at runtime.

ARRAY	LINKED LIST
Array is a collection of elements of similar data type.	Linked List is an ordered collection of elements of same type, which are connected to each other using pointers.
Array supports **Random Access**, which means elements can be accessed directly using their index, like arr[0] for 1st element, arr[6] for 7th element etc.	Linked List supports **Sequential Access**, which means to access any element/node in a linked list, we have to sequentially traverse the complete linked list, upto that element.
Hence, accessing elements in an array is **fast** with a constant time complexity of O(1).	To access **nth** element of a linked list, time complexity is O(n).
	In a linked list, new elements can be stored anywhere in the memory.
In an array, elements are stored in **contiguous memory location** or consecutive manner in the memory.	Address of the memory location allocated to the new element is stored in the previous node of linked list, hence formaing a link between the two nodes/elements.
In array, **Insertion and Deletion** operation takes more time, as the memory locations are consecutive and fixed.	In case of linked list, a new element is stored at the first free and available memory location, with only a single overhead step of storing the address of memory location in the previous node of linked list.
	Insertion and Deletion operations are **fast** in linked list.
Memory is allocated as soon as the array is declared, at **compile time**. It's also known as **Static Memory Allocation**.	Memory is allocated at **runtime**, as and when a new node is added. It's also known as **Dynamic Memory Allocation**.
In array, each element is independent and can be accessed using it's index value.	In case of a linked list, each node/element points to the next, previous, or maybe both nodes.
Array can be **single dimensional**, **two dimensional** or **multidimensional**	Linked list can be **Linear(Singly) linked list**, **Doubly linked list** or **Circular linked list**

	linked list.
Size of the array must be specified at time of array decalaration.	Size of a Linked list is variable. It grows at runtime, as more nodes are added to it.
Array gets memory allocated in the **Stack** section.	Whereas, linked list gets memory allocated in **Heap** section.

Consecutive memory allocated for array / linked list random memory locations



Need for pointers in Linked List

A pointer a variable that stores/points the address of another variable, used to allocate memory dynamically i.e. at run time.

Array, memory allocation is in contiguous manner, consecutive memory locations and access to any array element, is by use the array index, example arr[4] directly access the 5th memory location, returning the data stored there.

In list, data elements are allocated memory at runtime, hence the memory location can be anywhere.

Accessing a node in a list requires an address of the node stored in the previous node, hence forming a link between every node. (Pointers)

We need this additional **pointer** because without it, the data stored at random memory locations will be lost.

Pointer variable declaration

type *var-name;
asterisk designate a variable as a pointer.

```
int    *ip;    /* pointer to an integer */
double *dp;    /* pointer to a double */
float  *fp;    /* pointer to a float */
char   *ch     /* pointer to a character */
```

How to Use Pointers

```
#include <stdio.h>

int main()

{

Int  i,j;

int *p;    /* a pointer to an integer */

   p = &i;

   *p=5;

   J = i;

printf("%d %d %d\n", i, j, *p);

return 0;

}
```

The first declaration in this program declares two normal integer variables named **i** and **j**.

The line **int *p** declares a pointer named **p**. This line asks the compiler to declare a variable **p** that is a **pointer** to an integer.

The ***** indicates that a pointer is being declared rather than a normal variable.

N.B. You can create a pointer to anything: a float, a structure, a char, and so on. Just use a ***** to indicate that you want a pointer rather than a normal variable.

- In C, **&** is called the **address operator**.

- The expression **&i** means, "The memory address of the variable **i**."

- Thus, the expression **p = &i;** means, "Assign to **p** the address of **i**." Once you execute this statement, **p** "points to" **i**.

- Because the location ***p** is also **i**, **i** also takes on the value 5. Consequently, **j=i;** sets **j** to 5, and the **printf** statement produces **5 5 5**.

Trees Data Structure

- A tree is a hierarchical data structure whose point of entry is the root node
- A tree is a collection of elements *(called nodes)*, one of which is distinguished as *root*, along with a relation (*parenthood)* that places a hierarchical structure on the nodes.

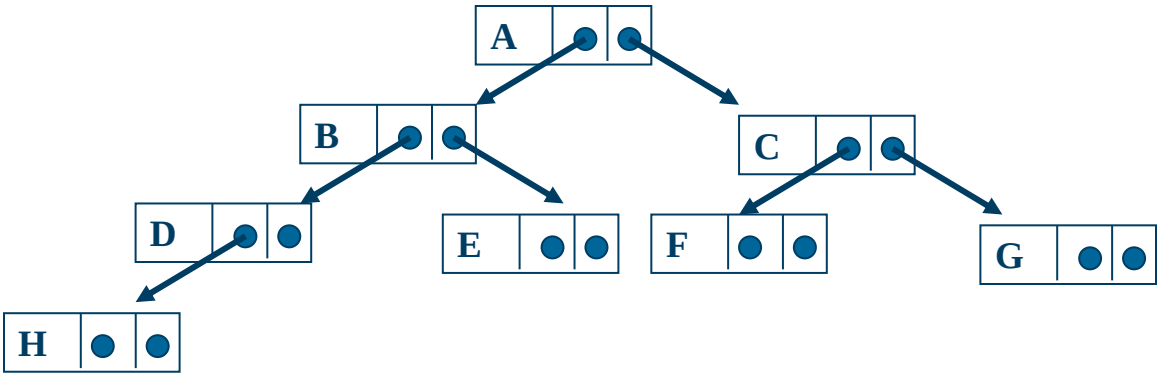
Key terms

- Root. A unique first node which is at the top of the tree structure.
- The unique predecessor of a node is called its **parent,**
- A node's successors are its **children**
- Leaf nodes. The nodes which do not have children
- Interior nodes. Nodes that have children
- Siblings. Nodes of the same parent
- Depth of tree. The length of the longest path from root to any other node.
- Depth of node: number of links leading to a node
- Degree of a node. The number of children it has.
- Size: the number of nodes

Exercise

From the following tree ADT identify the following

- Depth of node G = 2
- Depth of the tree = 3
- Degree of node H = 0
- Degree of node C = 2



- ❑ Degree: number of subtrees of a node
  - ✓ Nodes of degree 0 are leaf nodes
- ❑ Size: the number of nodes in the whole tree
- ❑ Depth of node: number of links leading to a node from the root node
- ❑ Depth of tree: largest depth of all the nodes

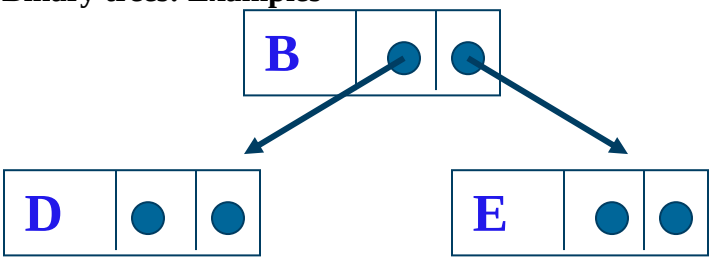
5.1 Why use of tree in data structures

- Trees are used for organisation of data in a manner that makes it efficient to retrieve it.
- To enable self sorting of data items.
- Reduce search time

Binary tree

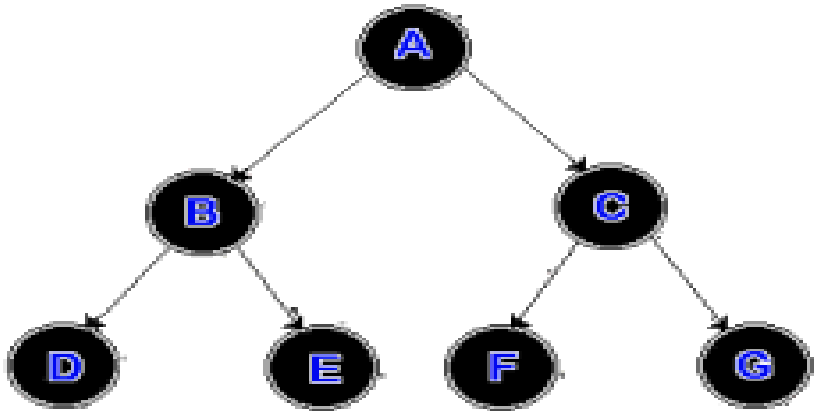
- Is a tree structure which
- May be empty (empty tree or null tree) i.e. has zero nodes
- Any Node in the tree can have degree of 0, 1 or 2
- A Child node is explicitly defined as a left child or a right child
- Consists of subtrees called left subtree and right subtree

Binary trees: Examples



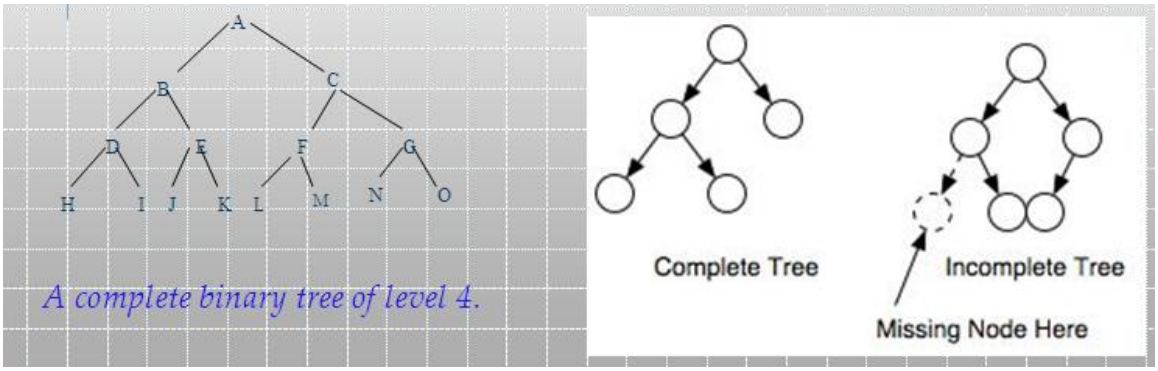
Balanced tree

- An optimisation of a tree which aims to keep equal numbers of items on each subtree of each node so as to minimise the maximum path from the root to any leaf node.
- As items are inserted and deleted, the tree is restructured to keep the nodes balanced and the search paths uniform.
- Such an algorithm is appropriate where the overheads of the reorganisation on update are outweighed by the benefits of faster search.



Complete Binary tree

- A complete tree is one in which there are no gaps between leaves.
- For instance, a tree with a root node that has only one child must have its child as the left node.
- A complete tree is one that has every level filled in before adding a node to the next level, and one that has the nodes in a given level filled in from left to right, with no breaks.



5.3 Binary trees traversal

- ❑ Traversal is the process of visiting every node once
- ❑ Visiting a node entails doing some processing at that node, but when describing a traversal strategy, we need not concern ourselves with what that processing is
- ❑ The process of visiting every node of a tree in a systematic way is referred to as tree traversal
- ❑ There are three tree traversal strategies
    i. Inorder
    ii. Preorder
    iii. Postorder

Binary trees traversal

1) Preorder Traversal
    i. Visit the root
    ii. Visit the left subtree
    iii. Visit the right subtree
2) Inorder Traversal
    i. Traverse the left subtree
    ii. Visit the root
    iii. Traverse the right subtree
3) Postorder Traversal
    i. Traverse the left subtree
    ii. Traverse the right subtree
    iii. Visit the root

- An in order prints the contents of a sorted tree in order, i.e the lowest value first and then increasing in value as it traverse the tree
- The order of a traversal would be A to Z if the tree uses strings & would be increasing in value numerically from 0 if the tree contains numerical values..

1. Traverse left sub tree
2. Visit the root
3. Traverse right sub tree

Post order Traversal:

- The contents of the left sub tree are printed first, followed by the contents of the right sub tree and finally the root node.

1. Traverse left sub tree
2. Traverse right sub tree
3. Visit the root

Algorithm for Preorder Traversal

1. if the tree is empty
2. Return.

else

3. Visit the root.
4. Preorder traverse the left subtree.
5. Preorder traverse the right subtree.

Algorithm for Inorder Traversal

1. if the tree is empty
2. Return.

else

3. Inorder traverse the left subtree.
4. Visit the root.
5. Inorder traverse the right subtree.

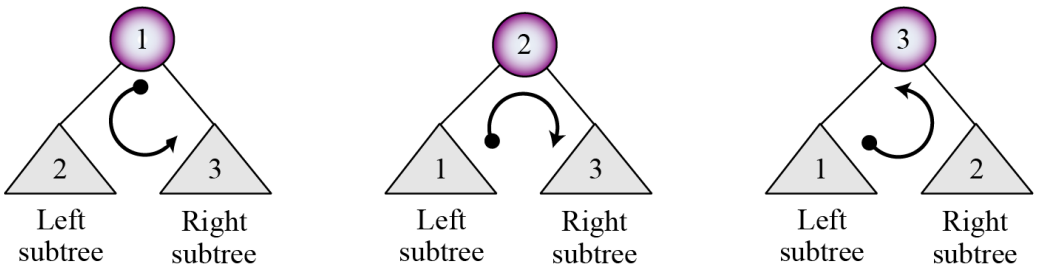
Algorithm for Postorder Traversal

1. if the tree is empty
2. Return.

else

3. Postorder traverse the left subtree.
4. Postorder traverse the right subtree.
5. Visit the root.

Example1



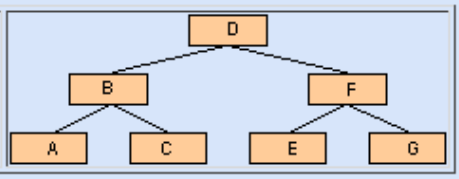
a. Preorder traversal b. Inorder traversal c. Postorder Ttraversal

Pre-order traversal

1. Start at the root node
2. Traverse the left subtree
3. Traverse the right subtree

The nodes of this tree would be visited in the order :

D B A C F E G

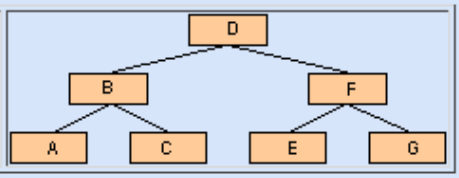


In-order traversal

1. Traverse the left subtree
2. Visit the root node
3. Traverse the right subtree

The nodes of this tree would be visited in the order :

A B C D E F G

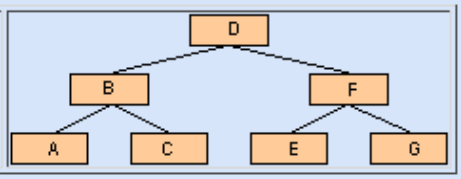


Post-order traversal

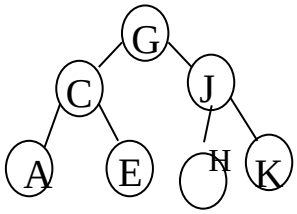
1. **Traverse the left subtree**
2. **Traverse the right subtree**
3. **Visit the root node**

The nodes of this tree would be visited in the order :

A C B E G F D



Example 3



A pre order traversal would result in the following string G,C,A,E,J,H,K
A post order Traversal would result to the following A,E,C,H,K,J,G
An in order Traversal would result in the following string A,C,EG,H,J,K.

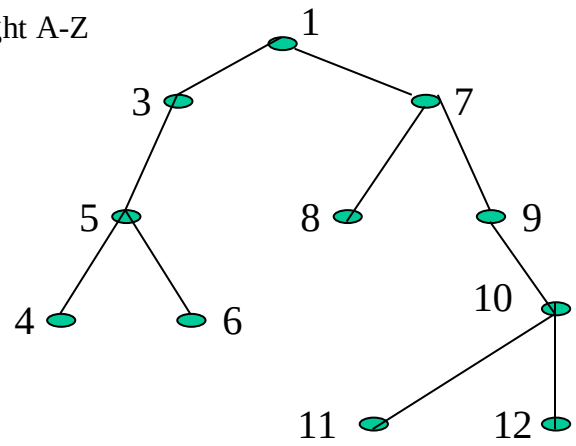
Example 2 Assume: visiting a node is printing its <u>label</u>

- Preorder: (root,left,right)

  1 3 5 4 6 7 8 9 10 11 12
- Inorder (sorted)left,root,right A-Z

  4 5 6 3 1 8 7 9 11 10 12
- Postorder: (left,right,root)

  4 6 5 3 8 11 12 10 9 7 1



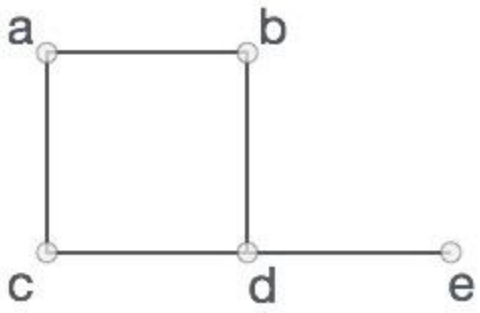
5.4 Binary Trees Traversal: Equations A + (B*C)

Construct binary trees form the equation using the three traversal strategies

Graphs ADT

A Graph is a non-linear data structure consisting of nodes and edges

Formally, a graph is a pair of sets **(V, E)**, where **V** is the set of vertices and **E** is the set of edges, connecting the pairs of vertices. Take a look at the following graph −



In the above graph,

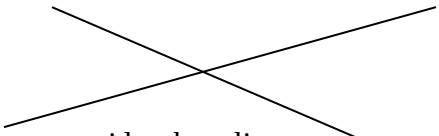
V = {a, b, c, d, e}

E = {ab, ac, bd, cd, de}

Defined as an ordered set (V,E) where G(V) represents a set of all elements called vertices and G(E) represents the edges between these lines.

i.e. **G = (V, E) consists of V a set of vertices, and E, a set of edges.**

Example

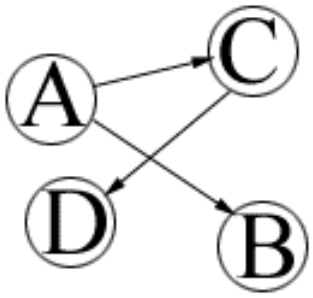


V(G)={v1,v2,v3,v4,v5,}

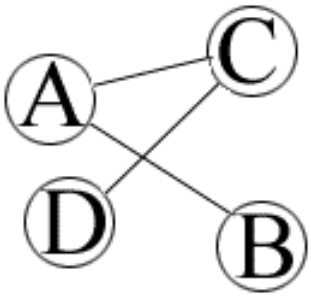
E(G)= {b1,b2,b3,b4}

Two vertices are said to be adjacent to one another if they are connected by a single edge In figure 1, vertices H and G are adjacent Adjacent vertices are also said to be neighbors

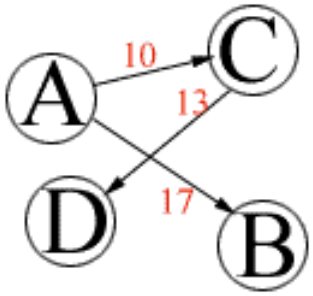
6.1 Types of graphs



Directed Graph



Undirected Graph



Weighted Graph

Directed Graph:- Vertices(cities): A, B, C and D Edges(routes): AB, AC, CD
Undirected Graph:- Vertices: A, B, C, D Edges: AB, AC, BA, CD, DC
Weighted Graph:- Vertices: A, B, C, D Weighted Edges: AC-10, AB-17, CD-13

Application of Graphs

Graphs are used to solve real-life problems that involve representation of the problem space as a network. Examples of networks include telephone networks, circuit networks, social networks (like LinkedIn, Facebook etc.).

For example, a single user in Facebook can be represented as a node (vertex) while their connection with others can be represented as an edge between nodes.

Each node can be a structure that contains information like user's id, name, gender, etc.

- Graphs are used in computer science to depict the flow of computation.
- Users on Facebook are referred to as vertices, and if they are friends, there is an edge connecting them. The Friend Suggestion system on Facebook is based on graph theory.
- You come across the Resource Allocation Graph in the Operating System, where each process and resource are regarded vertically. Edges are drawn from resources to assigned functions or from the requesting process to the desired resource. A stalemate will develop if this results in the establishment of a cycle.
- Web pages are referred to as vertices on the World Wide Web. Suppose there is a link from page A to page B that can represent an edge. This application is an illustration of a directed graph.
- Graph transformation systems manipulate graphs in memory using rules. Graph databases store and query graph-structured data in a transaction-safe, permanent manner.

Real life application I.T

- **Google Maps**: link your journey from the start to the end.
- **Social Networks**: friends are connected with each other using an edge where each user represents a vertex.
- **Recommendation System**: relationship data between user's recommendations uses graphs for connection.

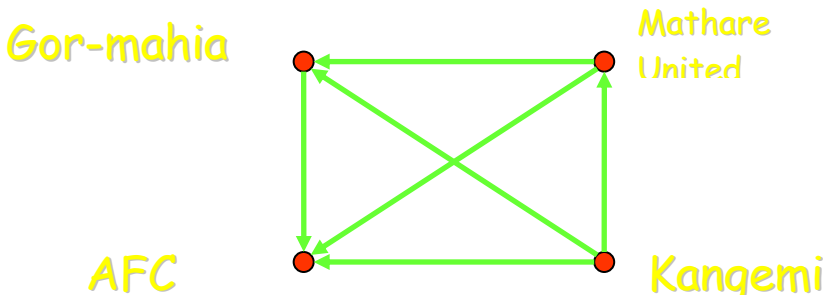
6.3 Graph Models

- Gives geometric arrangement and real life application of graphs

Example II:

In a round robin tournament, where each team plays against each other team exactly once.

The results of the tournament (which team beats which other team)? Can be represented by use of a **directed graph** with an edge (a, b) indicating that team a beats team b.



6.4 Graph Terminology

Definition:

- Two vertices u and v in an undirected graph G are called adjacent (or neighbors) in G if {u, v} is an edge in G.
- If e = {u, v}, the edge e is called incident with the vertices u and v. The edge e is also said to connect u and v.
- The vertices u and v are called endpoints of the edge {u, v}.
- The degree of a vertex in an un-directed graph is the number of edges incident with it, except that a loop at a vertex contributes twice to the degree of that vertex.

N.B one can determine the degree of a vertex in a graph by counting the lines that touch it.

- The degree of the vertex v is denoted by deg(v).
- A vertex of degree 0 is called isolated, since it is not adjacent to any vertex.

Note: A vertex with a loop at it has at least degree 2 and, by definition, is not isolated, even if it is not adjacent to any other vertex.

- A vertex of degree 1 is called pendant. It is adjacent to exactly one other vertex.

Representing A Graph In a Program

- ❑ We need to represent both the vertices and the edges
- ❑ Nodes/Vertices can be represent using an array.
- ❑ In this sense, nodes have to be numbered from node 0, to node n-1 (where n is the number of nodes)

- ❑ Node 0 is stored at position 0 in the array, node 1 at position 1, until node n-1 at position n

Representation

- ❑ Each node will be represented with a structure that is appropriate to store the information associated with a node E.g.
  - ✓ A record /Structure (in C)
  - ✓ An object (object oriented languages)
  - ✓ A hash (in perl)

Graph Operations

The most common graph operations are:

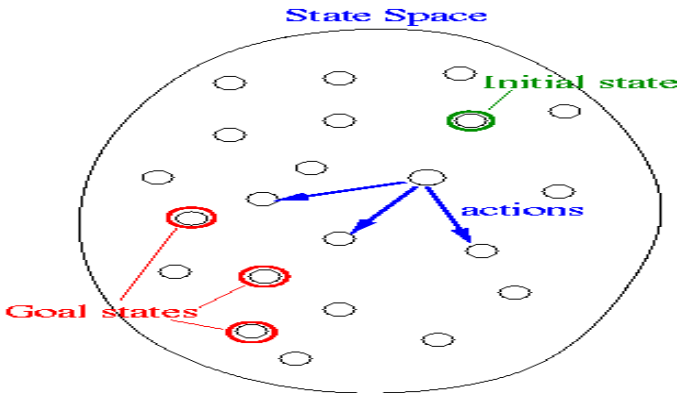
- Check if the element is present in the graph
- Graph Traversal
- Add elements (vertex, edges) to graph
- Finding the path from one vertex to another

6.5 Graph traversal

- The graph traversal involves a search strategy to find a path from the initial state to a goal state.

Examples of Search Problems

- **Chess game**
  - Each turn explore a move for a win
- **Route finding**
  - Explore routes for one to gets the destination
- **Theorem proving** **:** Explore reasoning for proof



Search Terminology

- **States:** "places" the search *can* visit
- **Search space/state space:** the set of possible states
- **Search path**
  - Sequence of states visited
- **Solution**
  - A state which solves the given problem (goal state)
- **Strategy**
  - How to choose the next state in the path at any given state

6.5.1 Evaluating Search strategies

There are four characteristics used to evaluate search graph algorithms:

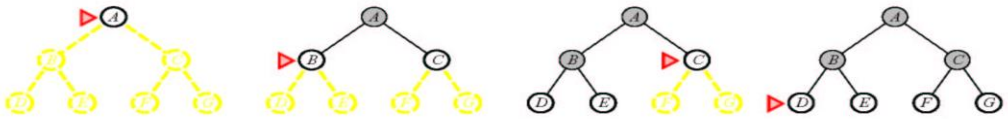
1. Completeness: the strategy should guarantee to find a solution if one exists?
2. Optimality: Does the solution have low cost or the minimal cost? Optimal solution achieved
3. Time complexity: Time taken or the number of nodes visited to find a solution.
4. Space complexity: Space used by the algorithm i.e. measured in terms of the maximum size of the search space

Search strategies.

- Breath First Search. (BFS)
- Depth First Search (DFS)

Breath First Search

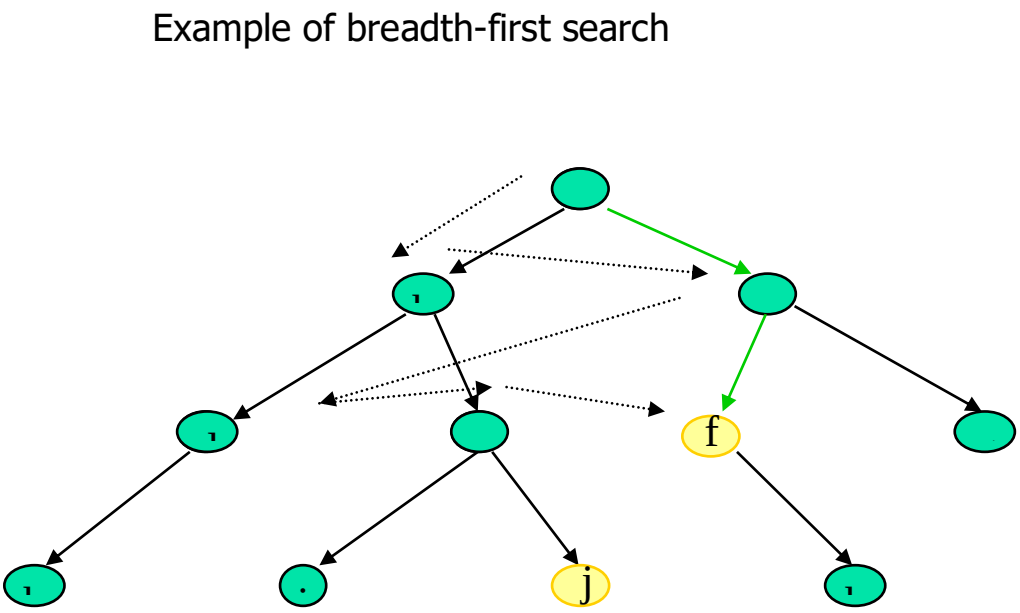
- ✓ The strategy is that nodes at depth i are visited before nodes at depth (i+1).
- ✓ The breadth-first strategy always visits all the nodes at one level of the tree, before visiting any of their children.
- ▪ Implementation: use of a First-In-First-Out queue (FIFO).



Example of BFS goal state is f

BFS Algorithm

The **BFS** algorithm is **implemented** by Using



- Nodes are visited in the order : a, b, c, d, e, f
- Solution path is : a, c, f

a ***queue*** to **store** the **nodes** in the **to Visit Nodes** data structure

BFS algorithm

A standard DFS implementation puts each vertex of the graph into one of two categories:

1. Visited
2. Not Visited

The purpose of the algorithm is to mark each vertex as visited while avoiding cycles.

The algorithm works as follows:

1. Start by putting any one of the graph's vertices at the back of a queue.
2. Take the front item of the queue and add it to the visited list.
3. Create a list of that vertex's adjacent nodes. Add the ones which aren't in the visited list to the back of the queue.
4. Keep repeating steps 2 and 3 until the queue is empty.

BFS pseudocode

create a queue Q

mark v as visited and put v into Q
while Q is non-empty
  remove the head u of Q
  mark and enqueue all (unvisited) neighbours of u

Pseudocode

```
Set all nodes to "not visited";

  q = new Queue();

  q.enqueue(initial node);

  while ( q ≠ empty ) do
  {
    x = q.dequeue();

    if ( x has not been visited )
    {
      visited[x] = true;        // Visit node x !

      for ( every edge (x, y)  /* we are using all edges ! */ )
        if ( y has not been visited )
              q.enqueue(y);       // Use the edge (x,y) !!!
    }
  }
```

Applications of graphs

In **computer science** graph theory is used for the **study of algorithms** like:

- Dijkstra's Algorithm
- Prims's Algorithm
- Kruskal's Algorithm

Applications

- Graphs are used to define the **flow of computation**.
- Graphs are used to represent **networks of communication**.
- Graphs are used to represent **data organization**.
- Graph transformation systems work on rule-based in-memory manipulation of graphs. Graph databases ensure **transaction-safe, persistent storing and querying of graph structured data**.
- Graph theory is used to find **shortest path in road** or a network.
- In **Google Maps**, various locations are represented as vertices or nodes and the roads are represented as edges and graph theory is used to find the shortest path between two nodes.

Google maps uses graphs for building transportation systems, where intersection of two(or more) roads are considered to be a vertex and the road connecting two vertices is

considered to be an edge, thus their navigation system is based on the algorithm to calculate the shortest path between two vertices.

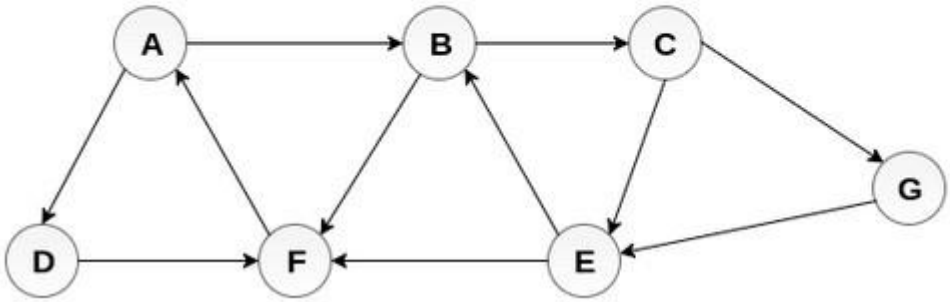
In **Facebook**, users are considered to be the vertices and if they are friends then there is an edge running between them. Facebook's Friend suggestion algorithm uses graph theory. Facebook is an example of **undirected graph**.

In **World Wide Web**, web pages are considered to be the vertices. There is an edge from a page u to other page v if there is a link of page v on page u. This is an example of **Directed graph**. It was the basic idea behind Google Page Ranking Algorithm.

In **Operating System**, we come across the Resource Allocation Graph where each process and resources are considered to be vertices. Edges are drawn from resources to the allocated process, or from requesting process to the requested resource. If this leads to any formation of a cycle then a deadlock will occur.

Example

Consider the graph G shown in the following image, calculate the minimum path p from



Adjacency Lists

A : B, D
B : C, F
C : E, G
G : E
E : B, F
F : A
D : F

node A to node E. Given that each edge has a length of 1.

Solution:

Minimum Path P can be found by applying breadth first search algorithm that will begin at node A and will end at E. the algorithm uses two queues, namely **QUEUE1** and **QUEUE2**. **QUEUE1** holds all the nodes that are to be processed while **QUEUE2** holds all the nodes that are processed and deleted from **QUEUE1**

1. Add A to QUEUE1 and NULL to QUEUE2.

1. QUEUE1 = {A}
2. QUEUE2 = {NULL}

2. Delete the Node A from QUEUE1 and insert all its neighbours. Insert Node A into QUEUE2

1. QUEUE1 = {B, D}
2. QUEUE2 = {A}

3. Delete the node B from QUEUE1 and insert all its neighbours. Insert node B into QUEUE2.

1. QUEUE1 = {D, C, F}
2. QUEUE2 = {A, B}

4. Delete the node D from QUEUE1 and insert all its neighbours. Since F is the only neighbour of it which has been inserted, we will not insert it again. Insert node D into QUEUE2.

1. QUEUE1 = {C, F}
2. QUEUE2 = { A, B, D}

5. Delete the node C from QUEUE1 and insert all its neighbours. Add node C to QUEUE2.

1. QUEUE1 = {F, E, G}
2. QUEUE2 = {A, B, D, C}

6. Remove F from QUEUE1 and add all its neighbours. Since all of its neighbours has already been added, we will not add them again. Add node F to QUEUE2.

1. QUEUE1 = {E, G}
2. QUEUE2 = {A, B, D, C, F}

7. Remove E from QUEUE1, all of E's neighbours has already been added to QUEUE1 therefore we will not add them again. All the nodes are visited and the target node i.e. E is encountered into QUEUE2.

1. QUEUE1 = {G}
2. QUEUE2 = {A, B, D, C, F, E}

Now, backtrack from E to A, using the nodes available in QUEUE2.

The minimum path will be **A → B → C → E**.

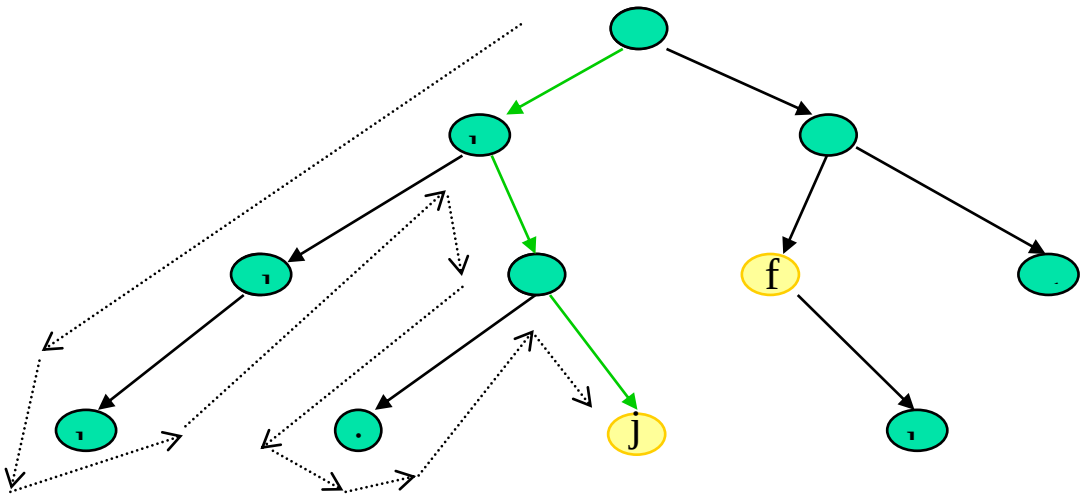
Depth First search

- Depth-First Search (DFS) always selects the left-most node and moves to the sibling node only after it is done with the entire subtree of the current node.
- DFS visits the deepest node in the current search tree.
- DFS strategy is implemented using a Last-In_First-Out (LIFO) stack ADT.

Uninformed search: DFS evaluation

Example :

Example of depth-first search goal state is j



- Nodes are visited in the order : a, b, d, h, e, i, j.
- Solution path is : a, b, e, j

DFS algorithm

The **DFS** algorithm is **implmented** by Using a ***stack***

A standard DFS implementation puts each vertex of the graph into one of two categories:

1. Visited
2. Not Visited

The purpose of the algorithm is to mark each vertex as visited while avoiding cycles.

The DFS algorithm works as follows:

1. Start by putting any one of the graph's vertices on top of a stack.
2. Take the top item of the stack and add it to the visited list.
3. Create a list of that vertex's adjacent nodes. Add the ones which aren't in the visited list to the top of stack.
4. Keep repeating steps 2 and 3 until the stack is empty.

Depth First Search (DFS) Algorithm

Depth first search (DFS) algorithm starts with the initial node of the graph G, and then goes to deeper and deeper until we find the goal node or the node which has no children.

The algorithm, then backtracks from the dead end towards the most recent node that is yet to be completely unexplored.

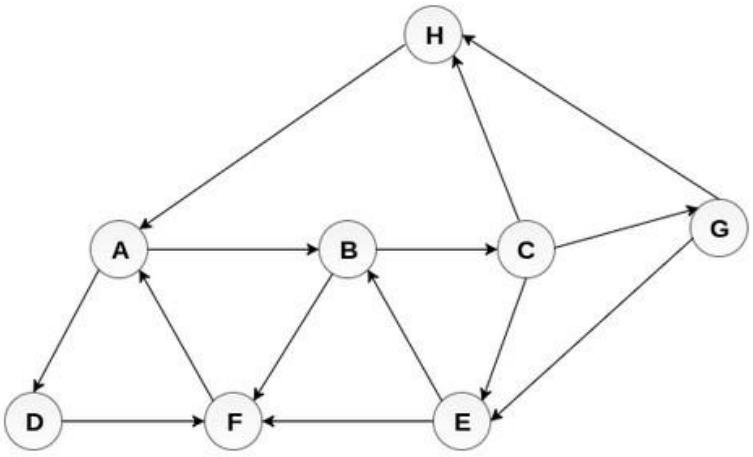
The data structure which is being used in DFS is stack. The process is similar to BFS algorithm. In DFS, the edges that leads to an unvisited node are called discovery edges while the edges that leads to an already visited node are called block edges.

Algorithm

- **Step 1:** SET STATUS = 1 (ready state) for each node in G
- **Step 2:** Push the starting node A on the stack and set its STATUS = 2 (waiting state)
- **Step 3:** Repeat Steps 4 and 5 until STACK is empty
- **Step 4:** Pop the top node N. Process it and set its STATUS = 3 (processed state)
- **Step 5:** Push on the stack all the neighbours of N that are in the ready state (whose STATUS = 1) and set their
  STATUS = 2 (waiting state)
  [END OF LOOP]
- **Step 6:** EXIT

Example :

Consider the graph G along with its adjacency list, given in the figure below. Calculate the order to print all the nodes of the graph starting from node H, by using depth first search (DFS) algorithm.



Adjacency Lists

A : B, D
B : C, F
C : E, G, H
G : E, H
E : B, F
F : A
D : F
H : A

Solution :

Push H onto the stack

1. STACK : H

POP the top element of the stack i.e. H, print it and push all the neighbours of H onto the stack that are is ready state.

1. Print H
2. STACK : A

Pop the top element of the stack i.e. A, print it and push all the neighbours of A onto the stack that are in ready state.

1. Print A
2. Stack : B, D

Pop the top element of the stack i.e. D, print it and push all the neighbours of D onto the stack that are in ready state.

1. Print D
2. Stack : B, F

Pop the top element of the stack i.e. F, print it and push all the neighbours of F onto the stack that are in ready state.

1. Print F
2. Stack : B

Pop the top of the stack i.e. B and push all the neighbours

1. Print B
2. Stack : C

Pop the top of the stack i.e. C and push all the neighbours.

1. Print C
2. Stack : E, G

Pop the top of the stack i.e. G and push all its neighbours.

1. Print G
2. Stack : E

Pop the top of the stack i.e. E and push all its neighbours

1. Print E
2. Stack :

Hence, the stack now becomes empty and all the nodes of the graph have been traversed.

The printing sequence of the graph will be :

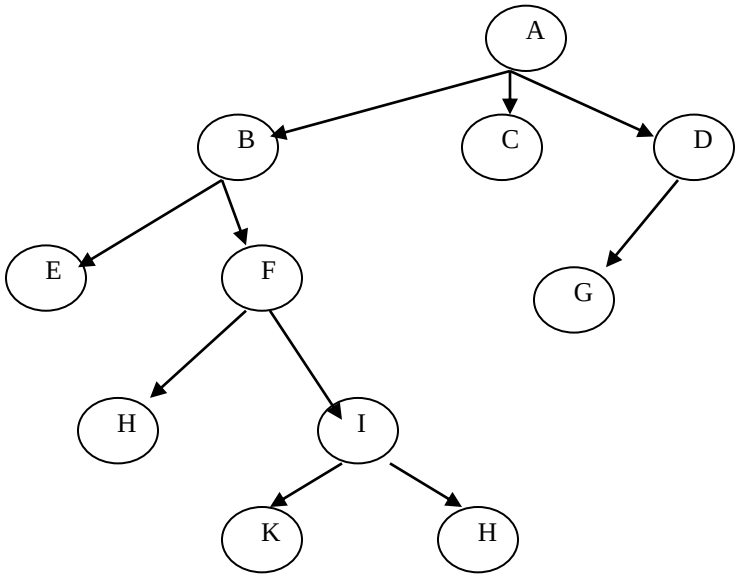
1. $H \rightarrow A \rightarrow D \rightarrow F \rightarrow B \rightarrow C \rightarrow G \rightarrow E$

DFS Pseudocode

```
Set all nodes to "not visited";

  s = new Stack();    ******* Change to use a stack

  s.push(initial node);    ***** Push() stores a value in a stack

  while ( s ≠ empty ) do
  {
    x = s.pop();          ****** Pop() remove a value from the stack

    if ( x has not been visited )
    {
      visited[x] = true;          // Visit node x !

      for ( every edge (x, y)  /* we are using all edges ! */ )
        if ( y has not been visited )
              s.push(y);        ***** Use push() !
    }
  }
```

Exercise

Using the diagram below, show the output after the Depth-first search and Breadth-first-search algorithm given that G is the goal state. Which is the most efficient algorithm



7.0 STACKS AND QUEUES

7.1 Stack

Definition:

- A **stack** is an abstract data type in which accesses are made at only one end
- The insert is called ***Push*** and the delete is called ***Pop***

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.

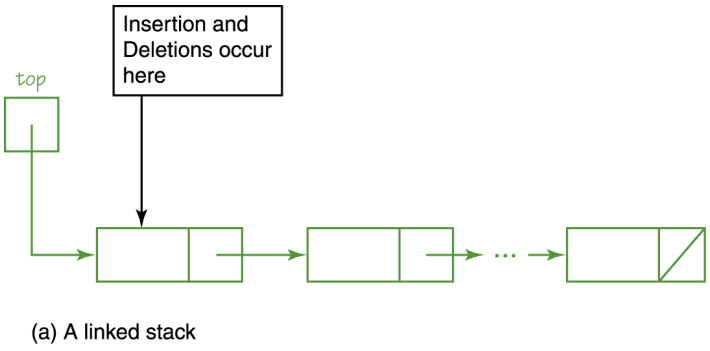
When data is PUSHed onto stack.

To use a stack efficiently we need to check 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. As this pointer always represents the top of the stack, hence named **top**. The **top** pointer provides top value of the stack without actually removing it.

- It is a LIFO (Last In First Out) ADT



(a) A linked stack

All operations are on Top and the pointer is restricted to the Top element.

- If there is no element in stack (n=0.) The top will be 0 and the pointer will be at 0 position.

peek()

Algorithm of peek() function −

begin procedure peek

  return stack[top]

end procedure

Implementation of peek() function in C programming language −

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

isfull()

Algorithm of isfull() function −

```
begin procedure isfull

   if top equals to MAXSIZE
      return true
   else
      return false
   endif

end procedure
```

Implementation of isfull() function in C programming language −

```
bool isfull() {
   if(top == MAXSIZE)
      return true;
   else
      return false;
}
```

isempty()

Algorithm of isempty() function −

```
begin procedure isempty

   if top less than 1
      return true
   else
      return false
   endif

end procedure
```

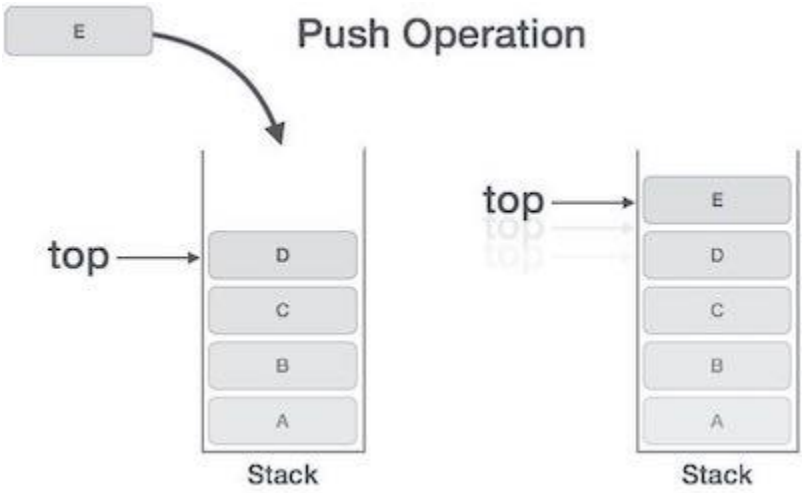
Implementation of isempty() function in C programming language is slightly different. We initialize top at -1, as index in array starts from 0. So we check if top is below zero or -1 to determine if stack is empty. Here's the code −

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

PUSH Operation

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

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



if linked-list is used to implement 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
```

Implementation of this algorithm in C, is very easy. See the below code −

```
void push(int data) {
   if(!isFull()) {
      top = top + 1;
      stack[top] = data;
   }else {
```

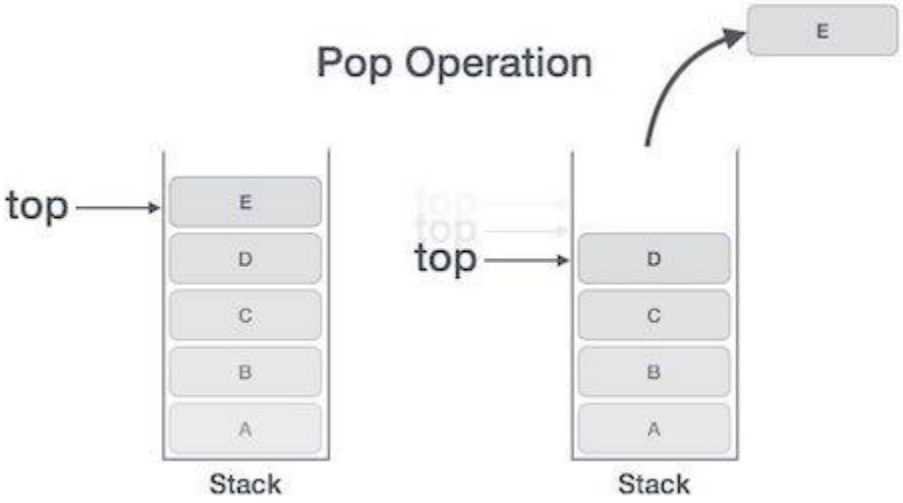
```
    printf("Could not insert data, Stack is full.\n");
  }
}
```

Pop Operation

Accessing the content while removing it from stack, is known as pop operation. In array implementation of pop() operation, data element is not actually removed, instead **top** is decremented to a lower position in stack to point to 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** − Check if stack is empty.
- **Step 2** − If stack is empty, produce error and exit.
- **Step 3** − If stack is not empty, access the data element at which **top** is pointing.
- **Step 4** − Decrease the value of top by 1.
- **Step 5** − return 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
```

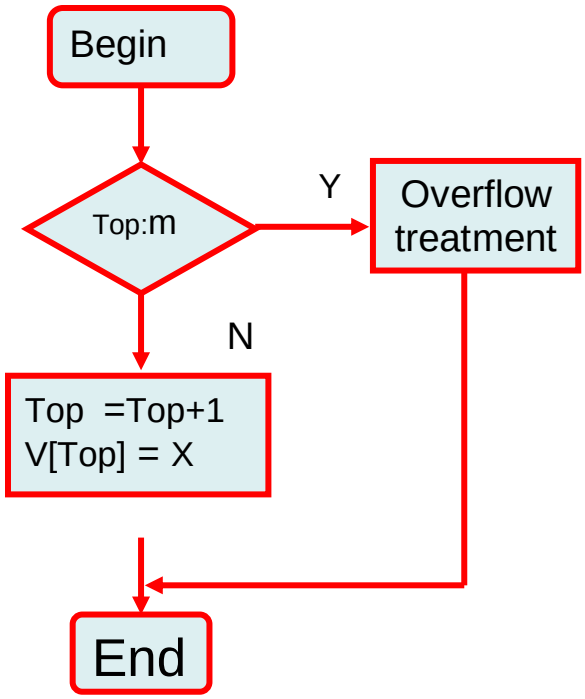
end procedure

Implementation of this algorithm in C, is shown below −

```
int pop(int data) {

  if(!isempty()) {
    data = stack[top];
    top = top - 1;
    return data;
  }else {
    printf("Could not retrieve data, Stack is empty.\n");
  }
```

Practical's Implementation using a procedural language of choice

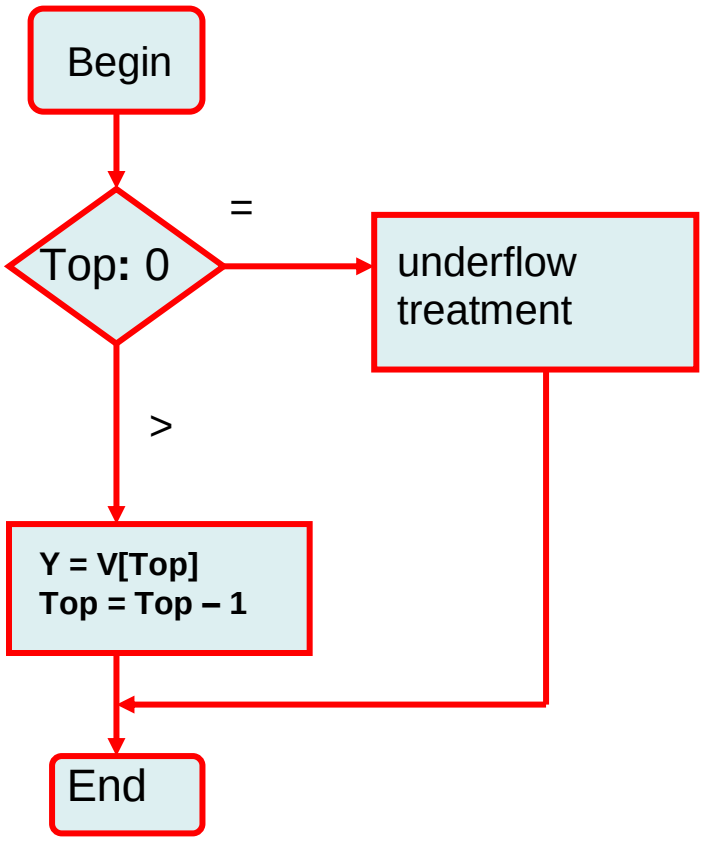
7.1.1 Insertion in a stack



- Inserting new element say X
- A test is made of whether Stack is full:
- If full then abort the procedure or else move pointer Top to position Top+1 and insert element X, i.e. V[Top] = X
- X becomes the new top element:

Stack: Deletion

- Declare temporary variable Y to store the deleted element:
- If Stack is empty i.e. Top = 0 there occurs an underflow error.
- Otherwise delete the top element that is, V[top] , and adjust pointer Top to (Top-[1).
- The deleted element Y may be printed out



7.1.3 Stack Applications

- Reversing Data: Stacks can be used to reverse data i.e records in a files.
- Transferring control form one part of a program to another, Processing Function Calls
- To store the return address when a subroutine is called.
- Evaluation of Arithmetic Expressions.
- Backtracking.

7.2 Queues:

Queue is ordered collection of homogeneous data elements in which insertion and deletion operation take place at two end . insertion allowed from starting of queue called **FRONT** point and deletion allowed from **REAR** end only

- insertion operation is called **ENQUEUE**
- deletion operation is called **DEQUEUE**

Examples

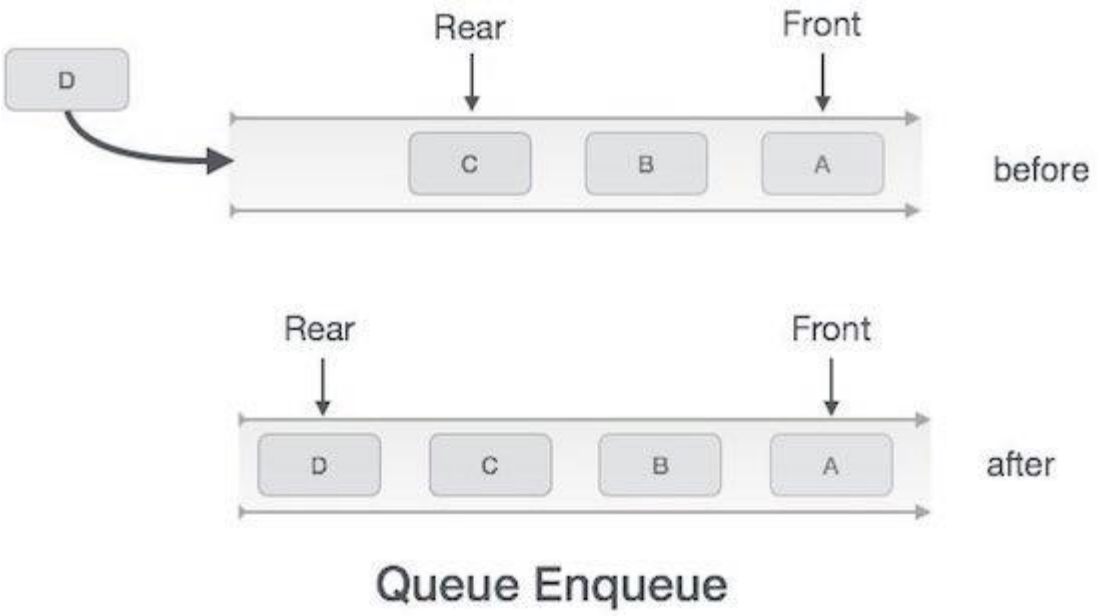
- Like a waiting line in a bank or supermarket
- Is a linear list in which all insertions and deletions are restricted:
- All insertions into the queue take place at one end called the rear (back) while all deletions take place at the other end called the front
- FIFO ADT

Enqueue Operation

Queues maintain two data pointers, **front** and **rear**.

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.



Queue Enqueue

Algorithm for enqueue operation

```
procedure enqueue(data)

  if queue is full
    return overflow
  endif

  rear ← rear + 1
  queue[rear] ← data
  return true

end procedure
```

Implementation of enqueue() in C programming language

Example

```
int enqueue(int data)
```

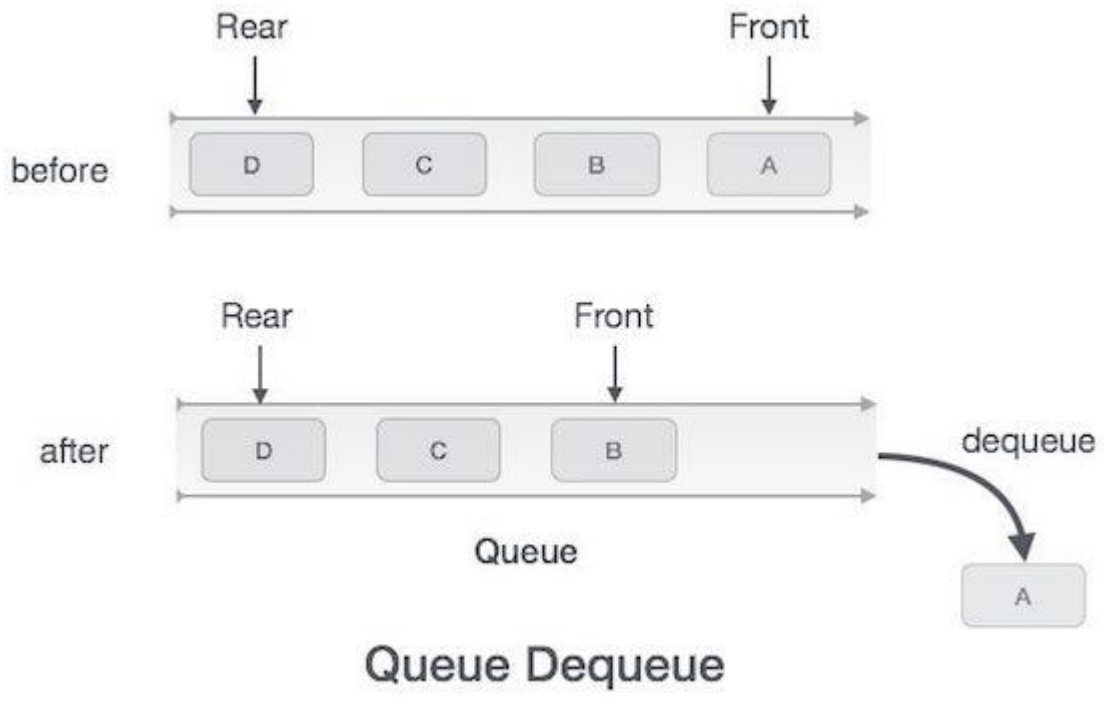
```
   if(isfull())
      return 0;

   rear = rear + 1;
   queue[rear] = data;

   return 1;
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
```

Implementation of dequeue() in C programming language −

Example

```
int dequeue() {
  if(isempty())
    return 0;

  int data = queue[front];
  front = front + 1;

  return data;
}
```

Application
- Queuing items from the processor for output to a peripheral device
- Queuing programs that are ready to be run by a computer.

Practical's Implementation using a procedural language of choice

8.0 SORTING & SERCHING

- Sorting is used to arrange names and numbers in meaningful way and improves the efficiency of searching.

Classification of Sorting Algorithm

Sorting algorithms can be categorized based on the following parameters:

Based on Number of Swaps or Inversion This is the number of times the algorithm swaps elements to sort the input. `Selection Sort` requires the minimum number of swaps.

Based on Number of Comparisons This is the number of times the algorithm compares elements to sort the input. Comparisons in the best case and `O(n^2)` comparisons in the worst case for most of the outputs.

Based on Recursion or Non-Recursion Some sorting algorithms, such as `Quick Sort`, use recursive techniques to sort the input. Other sorting algorithms, such as `Selection Sort` or `Insertion Sort`, use non-recursive techniques. Finally, some sorting algorithm, such as `Merge Sort`, make use of both recursive as well as non-recursive techniques to sort the input.

Based on Stability Sorting algorithms are said to be `stable` if the algorithm maintains the relative order of elements with equal keys. In other words, two equivalent elements remain in the same order in the sorted output as they were in the input.

`Insertion sort`, `Merge Sort`, and `Bubble Sort` are stable

`Heap Sort` and `Quick Sort` are not stable

Based on Extra Space Requirement Sorting algorithms are said to be `in place` if they require a constant `O(1)` extra space for sorting.

`Insertion sort` and `Quick-sort` are `in place` sort as we move the elements about the pivot and do not actually use a separate array which is NOT the case in merge sort where the size of the input must be allocated beforehand to store the output during the sort.

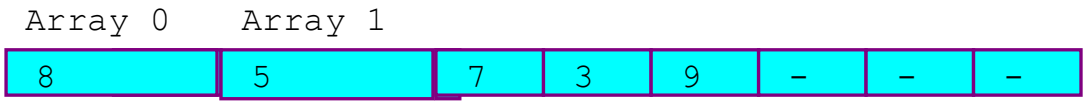
1. `Merge Sort` is an example of `out place` sort as it require extra memory space for it's operations.

8.1 Sorting algorithms

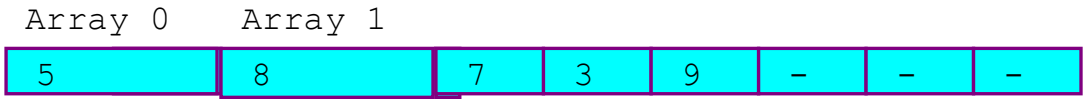
- Bubble sort
- Selection sort
- Insertion sort
- Quick sort

8.1.1 Bubble sort

- Works by repeatedly moving the largest element to the highest index position of the array.
- Rather than searching the entire array to find the largest element, bubble sort focuses on successive adjacent pairs of elements in the array.



Array [0] is greater than array [1] so switch the two elements



8.1.2 General algorithm for Bubble sort

Main loop

- Compare two adjacent elements at index "k" and "k+1"
- If the elements at index "k" is greater than the element "k+1" then swap the positions of the two values.
- One iteration moves largest value to the last position of array
- Repeat loop

Step-by-step example

Let us take the array of numbers "5 1 4 2 8", and sort the array from lowest number to greatest number using bubble sort algorithm. In each step, elements written in **bold** are being compared. Three passes will be required.

First Pass:
(**5 1** 4 2 8) ⟶ (**1 5** 4 2 8), Here, algorithm compares the first two elements, and swaps them.
(1 **5 4** 2 8) ⟶ (1 **4 5** 2 8), Swap since 5 > 4
(1 4 **5 2** 8) ⟶ (1 4 **2 5** 8), Swap since 5 > 2
(1 4 2 **5 8**) ⟶ (1 4 2 **5 8**), Now, since these elements are already in order (8 > 5), algorithm does not swap them.
Second Pass:
(**1 4** 2 5 8) ⟶ (**1 4** 2 5 8)
(1 **4 2** 5 8) ⟶ (1 **2 4** 5 8), Swap since 4 > 2
(1 2 **4 5** 8) ⟶ (1 2 **4 5** 8)
(1 2 4 **5 8**) ⟶ (1 2 4 **5 8**)
Now, the array is already sorted, but our algorithm does not know if it is completed. The algorithm needs one **whole** pass without **any** swap to know it is sorted.
Third Pass:
(**1 2** 4 5 8) ⟶ (**1 2** 4 5 8)
(1 **2 4** 5 8) ⟶ (1 **2 4** 5 8)
(1 2 **4 5** 8) ⟶ (1 2 **4 5** 8)
(1 2 4 **5 8**) ⟶ (1 2 4 **5 8**)

Pseudocode implementation

The algorithm can be expressed as:

```
procedure bubbleSort( A : list of sortable items )
  repeat
    swapped = false
    for i = 1 to length(A) - 1 inclusive do:
      if A[i-1] > A[i] then
        swap( A[i-1], A[i] )
        swapped = true
      end if
    end for
  until not swapped
end procedure
```

8.1.2 Selection sort

- The algorithm finds the smallest element of the array and interchanges it with the element in the first position of the array.
- After that it re- examines the remaining elements in the array to find the second smallest element.
- The element which is found is interchanged with the element in the second position of the array.

- The process continues until the elements are placed in their proper order as defined by the user either as ascending or descending order.

The algorithm works as follows:

1. Find the minimum value in the list
2. Swap it with the value in the first position
3. Repeat the steps above for the remainder of the list (starting at the second position and advancing each time)

Here is an example of this sort algorithm sorting five elements:

64 25 12 22 11

11 25 12 22 64

11 12 25 22 64

11 12 22 25 64

11 12 22 25 64

(nothing appears changed on this last line because the last 2 numbers were already in order)

8.1.3.4 Selection sort Algorithm

Selection-sort:

```
Procedure SELECTION-SORT(A):
 for i = 1 to A.length – 1
            /*set current element as minimum*/
    min = I
            /*check the element to be minimum*/
    for j = i + 1 to A.length do
       if A[j] < A[min]
          min = j
                    /*swap min element with the current element*/
                    Swap A[i] and A[min]
              End if
           End for
End for
```

End procedure

Practical's Implementation using a procedural language of choice

iv)Insertion sort

- A sorting technique that scans the sorted list, starting at the beginning, for the correct insertion point for each of the items from the unsorted list.

Basic idea of the algorithm.

Example: Sort A = 6, 3, 2, 4 with Insertion Sort.

Step 1: 6, 3, 2, 4

Step 2: 3, 6, 2, 4

Step 3: 2, 3, 6, 4

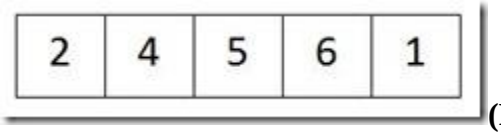
Step 4: 2, 3, 4, 6

Explanation. (Tracing the sorting algorithm)

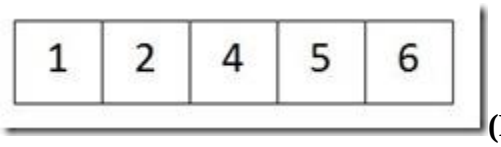
- The item being sorted as the *key*.
- 

(FIG 1) *Start from the element "2". Check it with the previous element "5". Since "5" > "2", move "5" forward (Result shown in Fig 2).*
- 

(FIG 2) *Select the next element "4". Check it with the previous element "5". Since "5" > "4", move "5" forward. Check "4" with "2". Since "2" < "4" then "2" should not move and thus we stop checking with previous values. Move "4" to the proper index (Result shown in Fig 3)*
- 

(FIG 3) Select the next element "6". Check it with the previous element "5". Since "5" < "6", "5" should not move and *thus we stop checking with previous values (Result shown in Fig 4)*
- 

(FIG 4)*Select the next element "1". Check it with the previous element "6". Since 6 > 1 move 6 forward. Check the key("1") with the other item "5" since "5" > "1", move "5" forward. continue the same process with all other elements (Result shown in Fig 5)*



(FIG 5)

Insertion sort pseudo code

INSERTION-SORT (*A*)

```
1  for j = 2 to n (length[A])
2      do key = A[j]   (store the Key element being sorted as index J)
3       Insert A[j] into the sorted sequence A[1 . . j - 1].
4              i = j – 1   // get the previous index
5       while i > 0 and A[i] > key // loop until you meet a smaller number or 0
6         do A[i + 1] = A[i]   // move the greater number forward
7            i = i – 1  // decrement  the counter so that we get the previous element
8       A[i + 1] = key  // set the key in the proper index
```

Practical's Implementation using a procedural language of choice

Merge Sort algorithm. (*Divide & Conquer sorts Algorithms)*

- The algorithms break the problem into several related subproblems, that are similar to the original problem but smaller in size
- This approach involves three steps:

1) Divide
   - ✓ The problem is split into smaller subproblems
2) Conquer
   - ✓ The subproblems solved recursively
   - ✓ If they are small enough, they are solved in a straight forward manner (not recursively)
3) Combine
   - ✓ Put together the subsolutions to get a solution to the original problem.

N.B: There are two sorting algorithms that use divide and conquer approach

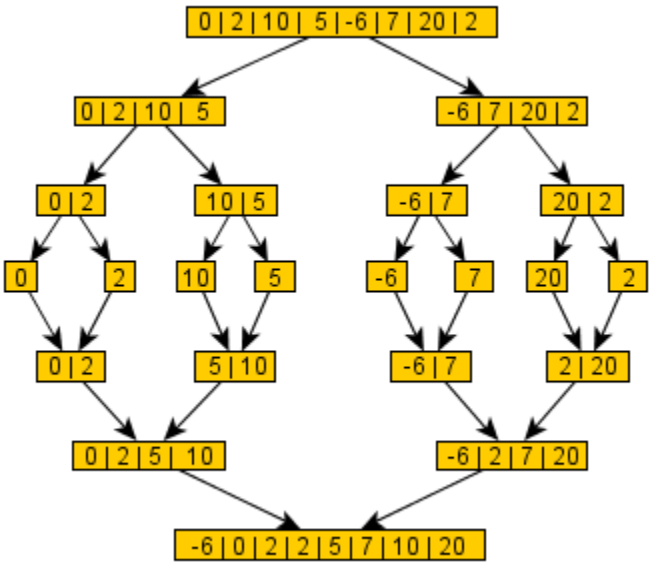
i)Merge sort

ii)Quick sort

Merge sort is a sorting algorithm based on the divide-and-conquer paradigm

- **Divide**: Divide the n-element sequence to be sorted into two subsequences of n/2 elements each.
- **Conquer**: Sort the two subsequences recursively using Merge Sort.
- **Combine**: Merge the two sorted subsequences to produce the sorted answer

Basic idea for the merge sort.



Merge sort Algorithm.

```
function mergesort(A[n])
   if n ≤ 1
      return A
   else
      middle =⌊ n / 2⌋
         create array left[1…middle]
         create array right[1… (n-middle)]
         for each x in A[1…n] up to middle
         add x to left
      for each x in A[middle+1…n]
            add x to right
      left = mergesort(left[middle])
      right = mergesort(right[n-middle])
      A= merge(left[middle], right[n-middle])
      return A
```

Insights

- Read class lecture notes as your reference point
- CAT's act as a mirror to the Exam, kindly take advantage.

End of course

Cheers and Best of luck in your Examination.