Unit 1
Introduction

Algorithm and types

An algorithm is a step-by-step procedure to solve a problem. A good algorithm should be optimized in terms of time and space. Different types of problems require different types of algorithmic techniques to be solved in the most optimized manner. There are different types of algorithms discussed below:

i. Brute Force Algorithm:

This is the most basic and simplest type of algorithm. A Brute Force Algorithm is the straightforward approach to a problem i.e., the first approach that comes to our mind on seeing the problem. More technically it is just like iterating every possibility available to solve that problem.

ii. Recursive Algorithm:

This type of algorithm is based on recursion. In recursion, a problem is solved by breaking it into subproblems of the same type and calling own self again and again until the problem is solved with the help of a base condition.

Some common problem that is solved using recursive algorithms are Factorial of a Number, Fibonacci Series, Tower of Hanoi, DFS for Graph, etc.

a) Divide and Conquer Algorithm:

In Divide and Conquer algorithms, the idea is to solve the problem in two sections, the first section divides the problem into subproblems of the same type. The second section is to solve the smaller problem independently and then add the combined result to produce the final answer to the problem. Some common problem that is solved using Divide and Conquers Algorithms are Binary Search, Merge Sort, Quick Sort, Strassen's Matrix Multiplication, etc.

b) Dynamic Programming Algorithms:

This type of algorithm is also known as the memoization technique because in this the idea is to store the previously calculated result to avoid calculating it again and again. In Dynamic Programming, divide the complex problem into smaller overlapping subproblems and store the result for future use. The following problems can be solved using the Dynamic Programming algorithm Knapsack Problem, Weighted Job Scheduling, Floyd Warshall Algorithm, etc.

c) Greedy Algorithm:

In the Greedy Algorithm, the solution is built part by part. The decision to choose the next part is done on the basis that it gives an immediate benefit. It never considers the choices that had been taken previously. Some common problems that can be solved through the Greedy Algorithm are Dijkstra Shortest Path Algorithm, Prim's Algorithm, Kruskal's Algorithm, Huffman Coding, etc.

d) Backtracking Algorithm:

In Backtracking Algorithm, the problem is solved in an incremental way i.e. it is an algorithmic technique for solving problems recursively by trying to build a solution incrementally, one piece at a time, removing those solutions that fail to satisfy the constraints of the problem at any point of time. Some common problems that can be solved through the Backtracking Algorithm are the Hamiltonian Cycle, M-Coloring Problem, N Queen Problem, Rat in Maze Problem, etc.

iii. Randomized Algorithm:

In the randomized algorithm, we use a random number.it helps to decide the expected outcome. The decision to choose the random number so it gives the immediate benefit

Some common problems that can be solved through the Randomized Algorithm are Quicksort:

In Quicksort we use the random number for selecting the pivot.

iv. Sorting Algorithm:

The sorting algorithm is used to sort data in maybe ascending or descending order. Its also used for arranging data in an efficient and useful manner.

Some common problems that can be solved through the sorting Algorithm are Bubble sort, insertion sort, merge sort, selection sort, and quick sort are examples of the Sorting algorithm.

v. Searching Algorithm:

The searching algorithm is the algorithm that is used for searching the specific key in particular sorted or unsorted data. Some common problems that can be solved through the Searching Algorithm are Binary search or linear search is one example of a Searching algorithm.

vi. Hashing Algorithm:

Hashing algorithms work the same as the Searching algorithm but they contain an index with a key ID i.e a key-value pair. In hashing, we assign a key to specific data.

Some common problems can be solved through the Hashing Algorithm in password verification.

Data Structure: -

It is a wave to organize the data in some way so we can do the operations on these data in effective way. Data structure may be organized in many different ways. The logical or mathematical model of a particular organization of data is called data structure. Choice of data model depends on two considerations.

It must be rich enough in structure to mirror the actual relationship of the data in the real world.

The structure should be simple enough that one can effectively process the data when necessary.

Data structure is mainly divided into two types. They are:

i. **Linear data structure:** Data structure in which data elements are arranged sequentially or linearly, where each element is attached to its previous and next adjacent elements, is called a linear data structure.
Examples of linear data structures are array, stack, queue, linked list, etc.
   a. **Static data structure:** Static data structure has a fixed memory size. It is easier to access the elements in a static data structure.
   An example of this data structure is an array.
   b. **Dynamic data structure:** In dynamic data structure, the size is not fixed. It can be randomly updated during the runtime which may be considered efficient concerning the memory (space) complexity of the code.
   Examples of this data structure are queue, stack, etc.

ii. **Non-linear data structure:** Data structures where data elements are not placed sequentially or linearly are called non-linear data structures. In a non-linear data structure, we can't traverse all the elements in a single run only.
Examples of non-linear data structures are trees and graphs.

iii. **Abstract Data type (ADT)**: It is a type (or class) for objects whose behavior is defined by a set of values and a set of operations. The definition of ADT only mentions what operations are to be performed but not how these operations will be implemented. It does not specify how data will be organized in memory and what algorithms will be used for implementing the operations. It is called "abstract" because it gives an implementation-independent view

Tools for algorithm analysis

Algorithm analysis is an important part of computational complexity theory, which provides theoretical estimation for the required resources of an algorithm to solve a specific computational problem. Analysis of algorithms is the determination of the amount of time and space resources required to execute it.

Why analysis of an algorithm is necessary?

- To predict the behavior of an algorithm without implementing it on a specific computer.
- It is much more convenient to have simple measures for the efficiency of an algorithm than to implement the algorithm and test the efficiency every time a certain parameter in the underlying computer system changes.
- It is impossible to predict the exact behavior of an algorithm. There are too many influencing factors.
- The analysis is thus only an approximation; it is not perfect.
- More importantly, by analyzing different algorithms, we can compare them to determine the best one for our purpose.

Types of algorithm analysis

1. Best case
2. Worst case
3. Average case

- **Best case:** Define the input for which algorithm takes less time or minimum time. In the best case calculate the lower bound of an algorithm. Example: In the linear search when search data is present at the first location of large data then the best case occurs.
- **Worst Case**: Define the input for which algorithm takes a long time or maximum time. In the worst calculate the upper bound of an algorithm. Example: In the linear search when search data is not present at all then the worst case occurs.
- **Average case**: In the average case take all random inputs and calculate the computation time for all inputs.
  And then we divide it by the total number of inputs.
  Average case = all random case time / total no of case

Algorithm analysis complexity

There are basically two aspects of computer programming. One is the data organization i.e. the data and structure to represent the data of the problem in hand, and is the subject of present text. The other one involves choosing the appropriate algorithm to solve the problem in hand. Data structure and algorithm designing, both involved with each other.

The choice of particular algorithm depends on the following considerations.

1. Performance required i.e. time complexity
2. Memory requirement i.e. space complexity

Time complexity

The time complexity of an algorithm is the amount of time it needs to run a completion. In computer programming the time complexity any program or any code quantifies the amount of time taken by a program to run. The time complexity is define using some of notations like Big O notations, which excludes coefficients and lower order terms. The time complexity is said to be described asymptotically, when we describe it in this way i.e., as the input size goes to infinity. For example, if the time required by an algorithm on all inputs of size n is at most $5n^3 + 3n$ for any n (bigger than some n_0), the asymptotic time complexity is $O(n^3)$.

It is commonly calculated by calculating the number of instructions executed by the program or the algorithm, where an elementary operation takes a fixed amount of time to perform.

Space Complexity

The complexity of an algorithm, i.e., a program is the amount of memory; it needs to run to completion. Some of the reasons for studying space complexities are:

- If the program is to run on multi user system, it may be required to specify amount of memory to be allocated to the program.
- We may be interested to know in advance that weather sufficient memory available to run program.
- There may be several possible solutions with different space requirements.

Asymptotic Notation

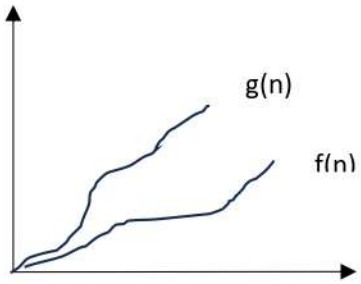
Asymptotic notation is used to describe the running time of an algorithm that how much time an algorithm takes with a given input.

There are three different notations. They are:

i. Big Oh Notation (O)
ii. Omega Notation (Ω)
iii. Theta Notation (□)

Big Oh Notation (O)

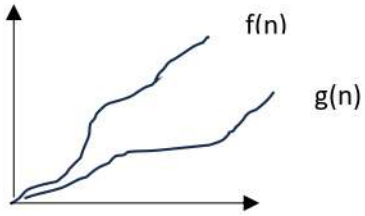
The notation O(n) is the formal way to express the upper bound of an algorithm running time. It measures the worst-case time complexity or the longest amount of time an algorithm can possibly take to complete.



Big(O)={g(n) and c such that g(n) ≥ c. f(n)}

Omega Notation (Ω)

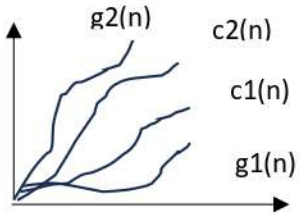
The notation Ω(n) is the formal way to express the lower bound of an algorithm's running time. It measures the best- case complexity or best amount of time an algorithm can possibly take to complete.



Ω(n)={g(n) and c such that g(n) ≤ f(n) }

Theta Notation (□)

The notation is the formal way to express both lower bound and the upper bound of an algorithm's running time. It is represented in such a way that



□(n)={g1(n), g2(n) , c1 & c2 such that g1(n) ≤ c1. f1(n) ≤ c2. f2(n) ≤ g2(n) }

Unit 2
Stack and Queue

Stack and its operations

A stack is an Abstract Data Type (ADT), that is popularly used in most programming languages. It is named stack because it has the similar operations as the real-world stacks, for example – a pack of cards or a pile of plates, etc.

The stack follows the LIFO (Last in - First out) structure where the last element inserted would be the first element deleted.

Stack operations usually are performed for initialization, usage and, de-initialization of the stack ADT.

The most fundamental operations in the stack ADT include: push(), pop(), peek(), isFull(), isEmpty(). These are all built-in operations to carry out data manipulation and to check the status of the stack.

Stack uses pointers that always point to the topmost element within the stack, hence called as the **top** pointer.

Insertion (Push) Operation

push() is an operation that inserts elements into the stack. The following is an algorithm that describes the push() operation in a simpler way.

Algorithm

i. Checks if the stack is full
ii. If the stack is full, produces an error and exit.
iii. If the stack is not full, increments top to point next empty space.
iv. Adds data element to the stack location, where top is pointing
v. Returns success.

Deletion (Pop) Operation

pop() is a data manipulation operation which removes elements from the stack. The following pseudo code describes the pop() operation in a simpler way.

Algorithm

i. Checks if the stack is empty.
ii. If the stack is empty, produces an error and exit.
iii. If the stack is not empty, accesses the data element at which top is pointing.
iv. Decreases the value of top by 1.
v. Returns success.

Peek()

The *peek()* is an operation retrieves the topmost element within the stack, without deleting it. This operation is used to check the status of the stack with the help of the top pointer.

Algorithm

i. START
ii. Return the element at the top of the stack
iii. END

IsFull()

isFull() operation checks whether the stack is full. This operation is used to check the status of the stack with the help of top pointer.

Algorithm

1. START
2. If the size of the stack is equal to the position of the stack, the stack is full. Return 1.
3. Otherwise, return 0.
4. END

IsEmpty()

The *isEmpty()* operation verifies whether the stack is empty. This operation is used to check the status of the stack with the help of top pointer.

Algorithm

1. START
2. If the top value is -1, the stack is empty. Return 1.
3. Otherwise, return 0.
4. END

Types of Stack

- Fixed Size Stack: As the name suggests, a fixed size stack has a fixed size and cannot grow or shrink dynamically. If the stack is full and an attempt is made to add an element to it, an overflow error occurs. If the stack is empty and an attempt is made to remove an element from it, an underflow error occurs.
- Dynamic Size Stack: A dynamic size stack can grow or shrink dynamically. When the stack is full, it automatically increases its size to accommodate the new element, and when the stack is empty, it decreases its size. This type of stack is implemented using a linked list, as it allows for easy resizing of the stack.

Application of Stack

Stacks are fundamental data structures that follow the Last-In-First-Out (LIFO) principle, meaning that the most recently added item is the first one to be removed. Stacks are used in various computer science and software engineering applications due to their simplicity and efficiency. Here are some common applications of stacks:

- **Function Call Management:** Stacks are extensively used in programming languages and their runtimes to manage function calls and track the execution of programs. When a function is called, its local variables and return address are pushed onto the stack. When the function returns, these values are popped from the stack.
- **Expression Evaluation:** Stacks are used in evaluating mathematical expressions, such as infix, postfix, or prefix notation. They help in managing the order of operations and operands.
- **Undo and Redo Functionality:** Many applications, such as text editors and graphic design software, implement undo and redo functionality using stacks. Each user action is pushed onto the undo stack, and redo operations are stored on a separate stack.
- **Backtracking Algorithms:** Algorithms that involve backtracking, like depth-first search (DFS) in graph traversal and solving puzzles (e.g., the Eight-Puzzle problem), often use stacks to keep track of the current path or state.
- **Memory Management:** Stacks are used for managing memory in computer systems. The call stack is responsible for storing function call data, including local variables and return addresses, during program execution.
- **Browser History:** The navigation history in web browsers is often implemented using two stacks: one for forward navigation and one for backward navigation. URLs are pushed onto these stacks as users navigate through web pages.
- **Data Structures:** Stacks are often used as a building block for other data structures like queues, which are implemented using two stacks (one for enqueue and one for dequeue operations).

These are just a few examples of the many applications of stacks in computer science and software engineering. Stacks provide an efficient way to manage and manipulate data in various context.

Queue

Queue, like Stack, is also an abstract data structure. The thing that makes queue different from stack is that a queue is open at both its ends. Hence, it follows FIFO (First-In-First-Out) structure, i.e. the data item inserted first will also be accessed first. The data is inserted into the queue through one end and deleted from it using the other end.

A real-world example of queue can be a single-lane one-way road, where the vehicle enters first, exits first. More real-world examples can be seen as queues at the ticket windows and bus-stops

Basic operation of Queue

- Queue operations also include initialization of a queue, usage and permanently deleting the data from the memory.
- The most fundamental operations in the queue ADT include: enqueue(), dequeue(), peek(), isFull(), isEmpty(). These are all built-in operations to carry out data manipulation and to check the status of the queue.
- Queue uses two pointers – front and rear. The front pointer accesses the data from the front end (helping in enqueueing) while the rear pointer accesses data from the rear end (helping in dequeuing).

Insertion operation : enqueue()

The *enqueue()* is a data manipulation operation that is used to insert elements into the queue. The following algorithm describes the enqueue() operation in a simpler way.

Algorithm:

1. START
2. Check if the queue is Full
3. If the queue is full, produce overflow error and exit.
4. If the queue is not full, increment rear pointer to point the next empty space.
5. Add data to element to the queue location, where the rear pointer is pointing.
6. END

Deletion operation: dequeue()

The *dequeue()* is a data manipulation operation that is used to remove elements from the queue. The following algorithm describes the dequeue() operation in a simpler way.

Algorithm

1. START
2. Check if the queue is empty.
3. If the queue is empty, produce underflow error and exit.
4. If the queue is not empty, access the data where front is pointing.
5. Increment front pointer to point to the next available data elements.
6. Return Success
7. END

Types of Queues

There are four different types of queue that are listed as follows – Simple Queue or Linear Queue

- Circular Queue
- Priority Queue
- Double Ended Queue (or Dequeue)

Let's discuss each of them.

Simple or Linear Queue

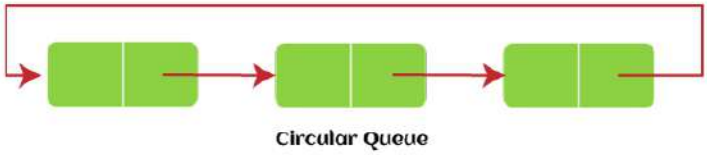
In Linear Queue, an insertion takes place from one end while the deletion occurs from another end. The end at which the insertion takes place is known as the rear end, and the end at which the deletion takes place is known as front end. It strictly follows the FIFO rule.

The major drawback of using a linear Queue is that insertion is done only from the rear end. If the first three elements are deleted from the Queue, we cannot insert more elements even though the space is available in a Linear Queue. In this case, the linear Queue shows the overflow condition as the rear is pointing to the last element of the Queue.

Circular Queue

In Circular Queue, all the nodes are represented as circular. It is similar to the linear Queue except that the last element of the queue is connected to the first element. It is also known as Ring Buffer, as all the ends are connected to another end.

The drawback that occurs in a linear queue is overcome by using the circular queue. If the empty space is available in a circular queue, the new element can be added in an empty space by simply incrementing the value of rear. The main advantage of using the circular queue is better memory utilization.



Circular Queue

Priority Queue

It is a special type of queue in which the elements are arranged based on the priority. It is a special type of queue data structure in which every element has a priority associated with it. Suppose some elements occur with the same priority, they will be arranged according to the FIFO principle. The representation of priority queue is shown in the below image.

Insertion in priority queue takes place based on the arrival, while deletion in the priority queue occurs based on the priority. Priority queue is mainly used to implement the CPU scheduling algorithms.

There are two types of priority queue that are discussed as follows -

a) **Ascending priority queue** - In ascending priority queue, elements can be inserted in arbitrary order, but only smallest can be deleted first. Suppose an array with elements 7, 5, and 3 in the same order, so, insertion can be done with the same sequence, but the order of deleting the elements is 3, 5, 7.

b) **Descending priority queue** - In descending priority queue, elements can be inserted in arbitrary order, but only the largest element can be deleted first. Suppose an array with elements 7, 3, and 5 in the same order, so, insertion can be done with the same sequence, but the order of deleting the elements is 7, 5, 3.

Application of Queue

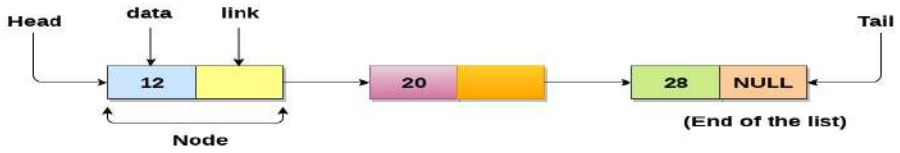
- **Task Scheduling**: Queues can be used to schedule tasks based on priority or the order in which they were received.
- **Resource Allocation:** Queues can be used to manage and allocate resources, such as printers or CPU processing time.
- **Batch Processing**: Queues can be used to handle batch processing jobs, such as data analysis or image rendering.
- **Message Buffering**: Queues can be used to buffer messages in communication systems, such as message queues in messaging systems or buffers in computer networks.
- **Event Handling**: Queues can be used to handle events in event-driven systems, such as GUI applications or simulation systems.
- **Traffic Management**: Queues can be used to manage traffic flow in transportation systems, such as airport control systems or road networks.

- **Operating systems:** Operating systems often use queues to manage processes and resources. For example, a process scheduler might use a queue to manage the order in which processes are executed.
- **Network protocols:** Network protocols like TCP and UDP use queues to manage packets that are transmitted over the network. Queues can help to ensure that packets are delivered in the correct order and at the appropriate rate.
- **Printer queues :**In printing systems, queues are used to manage the order in which print jobs are processed. Jobs are added to the queue as they are submitted, and the printer processes them in the order they were received.
- **Web servers:** Web servers use queues to manage incoming requests from clients. Requests are added to the queue as they are received, and they are processed by the server in the order they were received.
- **Breadth-first search algorithm:** The breadth-first search algorithm uses a queue to explore nodes in a graph level-by-level. The algorithm starts at a given node, adds its neighbors to the queue, and then processes each neighbor in turn.

Unit 3
List

Definition and structure of Linked List

- Linked List can be defined as collection of objects called **nodes** that are randomly stored in the memory.
- A node contains two fields i.e. data stored at that particular address and the pointer which contains the address of the next node in the memory.
- The last node of the list contains pointer to the null.



Advantages of Linked List

1. Dynamic Size:
   - Unlike arrays, linked lists do not have a fixed size. They can dynamically grow or shrink as needed, allowing for efficient memory usage.
2. Easy Insertion and Deletion:
   - Inserting or deleting elements in a linked list is generally more efficient than in an array, especially when dealing with elements in the middle of the list. In a linked list, you can easily change pointers to insert or delete elements without the need for shifting other elements.
3. No Wasted Memory:
   - Linked lists do not require a fixed amount of memory in advance. They only use the memory required by the elements they contain, reducing wasted memory when the size of the list is not known in advance.
4. Dynamic Memory Allocation:
   - Linked lists allow for dynamic memory allocation, which means nodes can be allocated or deallocated at runtime. This flexibility is particularly useful in situations where the size of the data structure is not known in advance.
5. Ease of Implementation:
   - Implementing certain algorithms or operations, such as reversing a list, is often simpler with linked lists compared to arrays.

5. Ease of Implementation:
   - Implementing certain algorithms or operations, such as reversing a list, is often simpler with linked lists compared to arrays.

6. No Pre-allocation of Memory:
   - In contrast to arrays, where you need to pre-allocate a certain amount of memory, linked lists can grow or shrink without requiring the allocation of memory in advance.

7. Efficient for Insertions and Deletions at the Beginning:
   - Inserting or deleting elements at the beginning of a linked list is a constant-time operation, as it involves updating the head pointer.

8. Support for Different Data Sizes:
   - Nodes in a linked list can be of different sizes, which can be beneficial in situations where elements vary in size.

Disadvantages of Linked List

1. **Random Access Time:**
   1. Accessing elements in a linked list takes O(n) time in the worst case because you need to traverse the list from the head to the desired element. This is in contrast to arrays, where random access is O(1).

2. **Memory Overhead:**
    1. Linked lists require additional memory for storing the pointers/references to the next node. This can result in higher memory overhead compared to arrays, especially for small amounts of data.
3. **Sequential Access:**
    1. While insertion and deletion are efficient in linked lists, sequential access can be slower than with arrays due to potential cache locality issues. Arrays offer better cache performance because their elements are stored in contiguous memory locations.
4. **Complexity:**
    1. Implementing and managing linked lists can be more complex than arrays. Pointers need to be properly managed to avoid memory leaks or dangling pointers.
5. **Extra Space for Pointers:**
    1. Each element in a linked list requires extra space to store a reference/pointer to the next node. This can be significant for large datasets, leading to higher space requirements compared to arrays
6. **Lack of Cache Locality:**
    - ❑ The lack of contiguous memory storage in linked lists can result in poor cache locality, leading to more cache misses and potentially slower performance.
7. **Traversal Overhead:**
    - ❑ Traversing a linked list requires following pointers from one node to another. This traversal overhead can lead to slower performance compared to arrays, particularly when dealing with large datasets.
8. **Not Suitable for Small Data Sets:**
    - ❑ For small data sets, the overhead of managing pointers in a linked list may outweigh the benefits, making arrays a more efficient choice.
9. **Limited Support for Direct Access:**
    - ❑ Unlike arrays that support direct access to any element using an index, linked lists do not offer direct access to arbitrary elements without traversing the list.

Uses of Linked list

- ❖ It is used in the navigation systems where front and back navigation is required.
- ❖ It is used by the browser to implement backward and forward navigation of visited web pages that is a back and forward button.
- ❖ It is also used to represent a classic game deck of cards.
- ❖ It is also used by various applications to implement undo and redo functionality.
- ❖ Doubly Linked List is also used in constructing MRU/LRU (Most/least recently used) cache.
- ❖ Other data structures like stacks, Hash Tables, Binary trees can also be constructed or programmed using a doubly-linked list.
- ❖ Also in many operating systems, the thread scheduler (the thing that chooses what process needs to run at which time) maintains a doubly-linked list of all processes running at that time.
- ❖ Implementing LRU Cache.
- ❖ Implementing Graph algorithms.

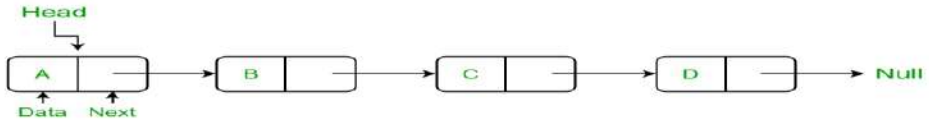
Types of Linked List

Linked list are mainly classified into four types. They are:

- Singly linked lists.
- Doubly linked lists.
- Circular linked lists.
- Circular doubly linked lists.

Singly Linked List

A singly linked list is a linear data structure in which the elements are not stored in contiguous memory locations and each element is connected only to its next element using a pointer.



Commonly Used Operations in Singly Linked List

The following operations are performed on a Single Linked List

- Insertion: The insertion operation can be performed in three ways. They are as follows…
  - Inserting At the Beginning of the list
  - Inserting At End of the list
  - Inserting At Specific location in the list
- Deletion: The deletion operation can be performed in three ways. They are as follows…
  - Deleting from the Beginning of the list
  - Deleting from the End of the list
  - Deleting a Specific Node
- Search: It is a process of determining and retrieving a specific node either from the front, the end or anywhere in the list.
- Display: This process displays the elements of a Single-linked list.

Insertion at the first node

Code

```
struct node* add_beg(struct node* head, int d)
{
struct node *ptr=malloc(sizeof(struct node));
ptr->data=d;
ptr->link=NULL;
ptr->link=head;
head=ptr;
return head;
}
```

Insertion at nth position

```
p=start;
for(i=1;i<position-1&&p!=NULL;i++)
{
p=p->link;
tmp->link=p->link;
p->link=tmp;
}
```

Insertion at the end

```
struct node
{
```

```
int data;
struct node *link;
}
struct node *tmp;
tmp=malloc(sizeof(struct node));
struct node *p;
p=start;
while(p!=NULL)
{
P=p->link;
}
p->link=tmp;
tmp->link=NULL;
}
```

Deletion of first node

```
struct node
{
int data;
struct node *link;
}
struct node *del_first(struct node *head)
if(head==NULL)
printf("List is already empty");
else
{
struct node *tmp=head;
head=head->link;
free(tmp);
tmp=NULL;
}
return head;
}
```

Deletion of nth position

```
void del_post(struct node **head, int position)
{
struct node *current=*head;
struct node *previous=*head;
if(*head==null)
printf("list is already empty\n");
else if(position==1)
{
*head=current->link;
free(current);
current=null;
}
else
{
while(position!=1)
{
```

```
previous=current;
current=current->link;
position--;
}
previous->link=current->link;
free(current);
current=null;
}
}
```

Deletion the last node

```
struct node *del_lst(struct node *head)
{
if(head==null)
printf(list is already empty\n");
else if(head->link==null)
{
free(head);
head=null;
}
return head;
}
```

Display the data of Linked List

```
void print_data(struct node *head)
{
if(head==null)
printf("linked list is empty!");
struct node *ptr=null;
ptr=head;
while(ptr!=null)
{
printf("%d\t",ptr->data);
ptr=ptr->link;
}
}
```


Doubly Linked List

A doubly linked list or a two-way linked list is a more complex type of linked list that contains a pointer to the next as well as the previous node in sequence.

Therefore, it contains three parts of data, a pointer to the next node, and a pointer to the previous node. This would enable us to traverse the list in the backward direction as well. Below is the image for the same:

Representation of Doubly Linked List

```
struct node
{
```

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

Basic Operations in doubly linked list

Creating a Node

A node is the basic building block of a doubly linked list. It consists of data, a pointer to the next node in the list, and a pointer to the previous node in the list.

Syntax

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

Inserting a Node at the Beginning

To insert a node at the beginning of a doubly linked list, we need to create a new node, set its data and pointers, and update the head and tail pointers of the list if necessary.

Syntax

```
void insert_at_beginning(struct node **head, struct node **tail, int data)
{
struct node *new_node = malloc(sizeof(struct node));
new_node->data = data;
new_node->next = *head;
if (*head == NULL)
{
*tail = new_node;
}
Else
{
(*head)->prev = new_node;
}
*head = new_node;
```

```
}
```

Inserting a Node at the Beginning

To insert a node at the beginning of a doubly linked list, we need to create a new node, set its data and pointers, and update the head and tail pointers of the list if necessary.

Syntax

```
void insert_at_beginning(struct node **head, struct node **tail, int data)
{
struct node *new_node = malloc(sizeof(struct node));
new_node->data = data;
new_node->next = *head;
if (*head == NULL)
{
*tail = new_node;
}
Else
{
(*head)->prev = new_node;
}
*head = new_node;
}
```

Deleting a Node

To delete a node from a doubly linked list, we need to find the node to delete, update the pointers of its neighbors, and free the memory allocated to the node.

Syntax

```
void delete_node(struct node **head, struct node *node)
{
if (*head == node)
{ *head = (*head)->next; if (*head != NULL)
{
(*head)->prev = NULL;
}
}
else
```

```
{
node->prev->next = node->next;
 if (node->next != NULL)
 {
 node->next->prev = node->prev;
}
}
free(node);
}
```

Advantages

- Reversing the doubly linked list is very easy.
- It can allocate or reallocate memory easily during its execution.
- As with a singly linked list, it is the easiest data structure to implement.
- The traversal of this doubly linked list is bidirectional which is not possible in a singly linked list.
- Deletion of nodes is easy as compared to a Singly Linked List. A singly linked list deletion requires a pointer to the node and previous node to be deleted but in the doubly linked list, it only required the pointer which is to be deleted.'
- Doubly linked lists have a low overhead compared to other data structures such as arrays.
- Implementing graph algorithms.

Disadvantages

- It uses extra memory when compared to the array and singly linked list.
- Since elements in memory are stored randomly, therefore the elements are accessed sequentially no direct access is allowed.
- Traversing a doubly linked list can be slower than traversing a singly linked list.
- Implementing and maintaining doubly linked lists can be more complex than singly linked lists.

Unit 4

Recursion

Introduction

Recursion is a powerful technique for solving problems in computer science, particularly in data structures and algorithms (DSA). It is a process where a function calls itself directly or indirectly to solve a problem. Recursive functions break down a problem into smaller, self-similar subproblems until a base case is reached, and then the solutions to the subproblems are combined to solve the original problem.

Why use recursion in DSA?

Recursion can be a very elegant and concise way to solve certain types of problems, especially those that have a natural recursive structure. It can also make code more modular and easier to understand, as the recursive function can encapsulate the logic for solving the problem.

Examples of Recursion in DSA

Here are some examples of how recursion is used in DSA:

- Factorial: Calculating the factorial of a number can be done recursively by defining a function that takes a positive integer n as input and returns n * factorial(n - 1) if n > 1, or 1 if n = 1.
- Fibonacci sequence: Generating the Fibonacci sequence can be done recursively by defining a function that takes an integer n as input and returns the nth Fibonacci number by adding the previous two Fibonacci numbers.
- Traversing trees: Traversing a tree data structure can be done recursively by defining a function that takes a node of the tree as input and visits the node, then recursively traverses its left child and its right child.

Here are some key properties of recursion:

- Self-reference: A recursive function calls itself directly or indirectly.
- Base cases: Recursive functions must have at least one base case, which is a simple case of the problem that can be solved directly without further recursion.
- State change: Recursive functions must change their state as they progress towards the base case. This ensures that the recursion eventually terminates.
- Elegance and conciseness: Recursion can often provide a more elegant and concise solution to problems than iterative solutions.
- Modularity: Recursive functions can encapsulate the logic for solving a problem, making the code more modular and easier to understand.
- Potential overhead: Recursion can involve more overhead than iterative solutions due to the function call overhead.
- Tail recursion: Tail recursion is a special type of recursion where the recursive call is the last action performed by the function. Tail recursion can be optimized by compilers to eliminate the overhead of function calls.
- Trade-offs: The use of recursion should be carefully considered, as there are often trade-offs between recursion and iterative solutions in terms of efficiency, code complexity, and readability.

Recursion is a valuable tool in the arsenal of problem-solving techniques, and understanding its

Recursion VS iteration

Recursion and iteration are two fundamental techniques for solving problems in computer science, particularly in data structures and algorithms (DSA). While both techniques allow for repetitive execution of a set of instructions, they differ in their approach and applicability.

- Recursion is a technique where a function calls itself directly or indirectly to solve a problem. It is based on the principle of breaking down a problem into smaller, self-similar subproblems until a base case is reached, at which point the solutions to the subproblems are combined to solve the original problem. Recursion is often used for problems that have a natural recursive structure, such as calculating factorials, generating the Fibonacci sequence, or traversing trees.

- ❖ Iteration, on the other hand, involves repeatedly executing a block of code until a specific condition is met. This is typically achieved using control flow structures like loops (for, while, do-while) that control the number of iterations. Iteration is well-suited for problems that involve processing a sequence of data or iteratively refining a solution.

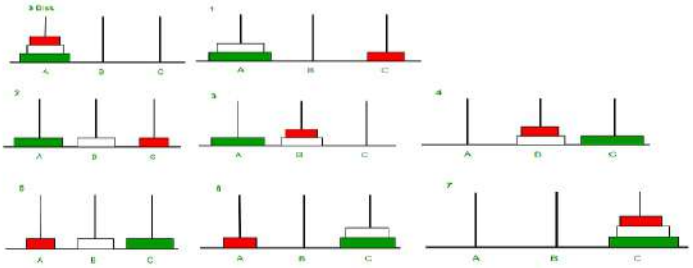
Recursion	**Iteration**
Breaks down problem into self-similar subproblems	Executes a block of code repeatedly
Function calls	Control flow structures (for, while, do-while)
Reached through base cases	Reached when a condition is met
Self-similar problems	Sequential processing, iterative refinement
Can be more complex due to self-reference	Typically simpler due to explicit control flow
Can consume more memory due to function call stack	Typically more memory-efficient
Can be less efficient due to function call overhead	Can be more efficient for certain problems
Optimization technique for tail-recursive functions	Not applicable

TOH and Solution

The Towers of Hanoi (TOH) is a classic problem-solving puzzle with mathematical origins. It consists of a set of rods (usually three) and a different number of disks which can slide onto any rod. The problem is to transfer

the entire stack of disks from one rod (the source) to another rod (the destination) while obeying the following rules:

- Only one disk can be moved at a time.
- Each disk must be placed on top of another disk of larger size or on an empty rod.



Follow the steps below to solve the problem:

- Create a function **towerOfHanoi** where pass the **N** (current number of disk), **from_rod**, **to_rod**, **aux_rod.**
- Make a function call for N – 1 th disk.
- Then print the current the disk along with **from_rod** and **to_rod**
- Again make a function call for N – 1 th disk.

Implementation Using C using Recursive Function

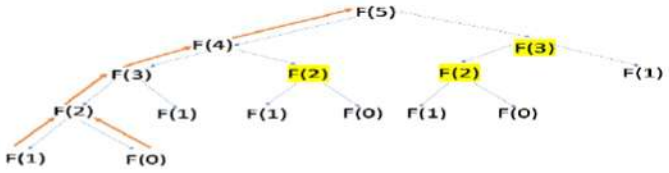
```
void TOH(int n, int A, int B, int C)
{
if(n>0)
{
TOH(n-1,A,C,B);
printf("Move a disc from %d to %d",A,C);
TOH(n-1,B,A,C);
}
```

Solution of Fibonacci Sequence

- The Fibonacci numbers are the numbers in the following integer sequence in the following integer sequence. 0,1,1,2,3,5,8,13,21,34,55,89,144,.....
- In mathematical terms, the sequence F_n of Fibonacci numbers is defined by the recurrence relation $F_n = F_{n-1} + F_{n-2}$ with seed values $F_0 = 0$ and $F_1 = 1$.

  Recursive Pseudocode

```
def Fibonacci(n)
if(n==0)
return 0
else if (n==1)
return 1
else
return Fibonacci(n-1)+Fibonacci(n-2)
```



Code

```
#include<stdio.h>
#include<conio.h>
int fib(int);//function declare
int main()
{
     int n,i;
     printf("How many terms do you need\n");
     scanf("%d",&n);
     for(i=0;i<n;i++)
     {
          printf("%d\t",fib(i));
     }
     return 0;
}
int fib(int a)
{
     if(a==0)
     return 0;
```

```
    else if(a==1)
    return 1;
    else
    return fib(a-1)+fib(a-2);
}
```

Solution of Factorial sequence

- The factorial of a number is the product of all integers from 1 to that number.
- For Example, the factorial of 5 is 5*4*3*2*1=120.
- Factorial is not defined for negative numbers and the factorial is one, 0!=1.
- Recursive definition of the factorial function f:

f(0)=1 f(1)=1 <- **Base Case**(s): It doesn't require recursion, where the recursive calls of f stops.

f(n)=n. f(n-1) <- **Recursive Case**(s): the function call itself, with a simpler instance/ smaller value.

Recursive Pseudocode

def factorial_recursive(n)

if(n==1 or n==0)

return 1

else

return n*factorial_recursive(n-1)

Code

```
#include<stdio.h>
#include<conio.h>
int factorial(int n);
int main()
{
    int n,x;
    printf("Enter any number\n");
    scanf("%d",&n);

    printf("Factorial of %d is %d",n,factorial(n));
    return 0;
```

```
}
int factorial(int n)
{
      if(n<=1)
      return 1;
      else
      return (n*factorial(n-1));
}
```

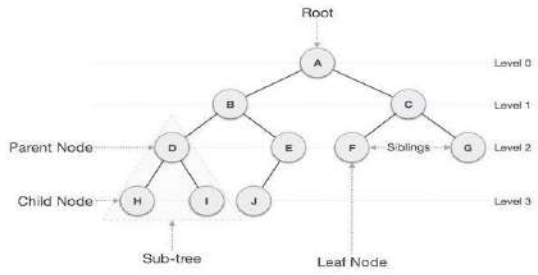
Unit 5

Trees

Tree Concepts

A tree is a non-linear abstract data type with a hierarchy-based structure. It consists of nodes (where the data is stored) that are connected via links. The tree data structure stems from a single node called a root node and has subtrees connected to the root. Trees can be mainly classified into three types. They are:

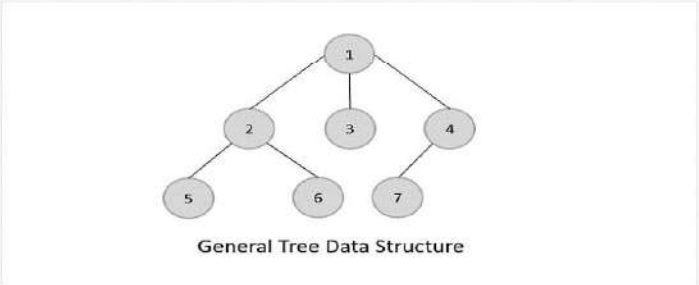
- i. General Trees
- ii. Binary Trees
- iii. Binary Search Trees



i. General Trees

General trees are unordered tree data structures where the root node has minimum 0 or maximum 'n' subtrees.

The General trees have no constraint placed on their hierarchy. The root node thus acts like the superset of all the other subtrees.



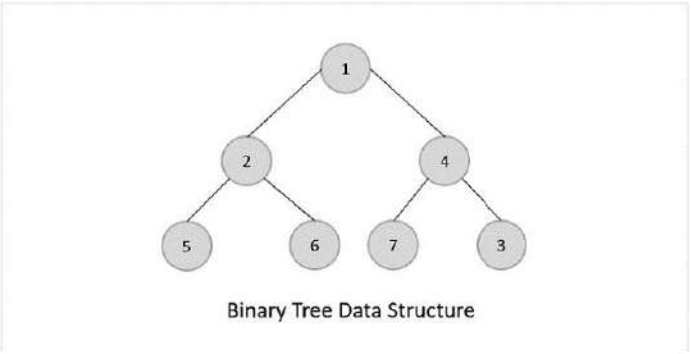
ii. Binary Tree

Binary Trees are general trees in which the root node can only hold up to maximum 2 subtrees: left subtree and right subtree. Based on the number of children, binary trees are divided into three types.

a) **Full Binary Tree:** A full binary tree is a binary tree type where every node has either 0 or 2 child nodes.

b) **Complete Binary Tree**: A complete binary tree is a binary tree type where all the leaf nodes must be on the same level. However, root and internal nodes in a complete binary tree can either have 0, 1 or 2 child nodes.

c) **Perfect Binary Tree:** A perfect binary tree is a binary tree type where all the leaf nodes are on the same level and every node except leaf nodes have 2 children.



Binary Tree Data Structure

Application of Binary Tree

- **Search algorithms:** Binary search algorithms use the structure of binary trees to efficiently search for a specific element. The search can be performed in O(log n) time complexity, where n is the number of nodes in the tree.
- **Sorting algorithms:** Binary trees can be used to implement efficient sorting algorithms, such as binary search tree sort and heap sort.
- **Database systems:** Binary trees can be used to store data in a database system, with each node representing a record. This allows for efficient search operations and enables the database system to handle large amounts of data.
- **File systems:** Binary trees can be used to implement file systems, where each node represents a directory or file. This allows for efficient navigation and searching of the file system.
- **Compression algorithms:** Binary trees can be used to implement Huffman coding, a compression algorithm that assigns variable-length codes to characters based on their frequency of occurrence in the input data.
- **Decision trees:** Binary trees can be used to implement decision trees, a type of machine learning algorithm used for classification and regression analysis.

Operation in Binary Tree

There are several fundamental operations that can be performed on a binary tree. These operations are crucial for manipulating and utilizing the data stored within the tree structure. Here are some of the most common:

- Traversal
- Searching
- Insertion
- Deletion

Node Declaration

```
struct Node
{
int data;
struct Node *left;
struct Node *right;
}
```

Insertion

This operation involves adding a new node with a specific key or value to the appropriate location in the tree. It maintains the binary search property while inserting the new node.

Algorithm

i. Create a queue q.
ii. If root is NULL, add node and return.
iii. Else continue until q is not empty.
iv. If a child does not exists, add the node there.
v. Otherwise add the node to the leftmost node.
vi. **Insertion Routine**

```
treeNode *Insert(treeNode *Node, int data)
{
If(node==NULL)
{
```

Code

```
treeNode *temp;
Temp=(treeNode *)malloc(sizeof(treeNode));
Temp->data=data;
Temp->left=temp->right=NULL;
Return temp;
}
If(data>(node->data))
{
Node->right=Insert(node->right,data);
Else if(data<(node->data))
```

```
{
Node->left=Insert(Node->Left,data);
}
return node;
}
```

Deletion

This operation involves removing a specific node from the tree without violating the tree's structure and properties. This can be a complex operation depending on the configuration of the tree and the presence of child nodes.

Algorithm

i. Starting at the root, find the deepest and rightmost node in the binary tree and the node which we want to delete.

ii. Replace the deepest rightmost node's data with the node to be deleted.

iii. Then delete the deepest rightmost no

Deletion Routine

```
treeNode *Delete(treeNode *node, int data)
{
treeNode *temp;
If(node==NULL)
{
Printf("Element Not found");
}
Else if(data<node->data)
{
Node->left=Delete(node->left,data);
}
Else if(data>node->data)
{
Node->left=Delete(node->right,data);
}
```

Algorithm of Tree Search

i. Start at the root node.

ii. Compare the key value of the current node with the target key.

iii. If the target key is found, return the node.

iv. If the target key is smaller than the current node's key, move to the left child and repeat the process.

Tree Traversal

The process of visiting the nodes is known as tree traversal. There are three types traversals used to visit a node:

- In order traversal
- Preorder traversal
- Post order traversal

1.In order Traversal

The left subtree is visited first, followed by the root, and finally the right subtree in this traversal strategy. Always keep in mind that any node might be a subtree in and of itself. The output of a binary tree traversal in order produces sorted key values in ascending order.

Algorithm

1. First, visit all the nodes in the left subtree
2. Then the root node
3. Visit all the nodes in the right subtree

2.Preorder Traversal

In this traversal method, the root node is visited first, then the left subtree, and finally the right subtree.

Algorithm

1. Visit root node
2. Visit all the nodes in the left subtree
3. Visit all the nodes in the right subtree

3.Postorder Traversal

The root node is visited last in this traversal method, hence the name. First, we traverse the left subtree, then the right subtree, and finally the root node

Algorithm

1. Visit all the nodes in the left subtree
2. Visit all the nodes in the right subtree
3. Visit the root node

Height level and depth of tree and its importance

The **depth** of a node is the number of edges present in path from the root node of a **tree** to that node.

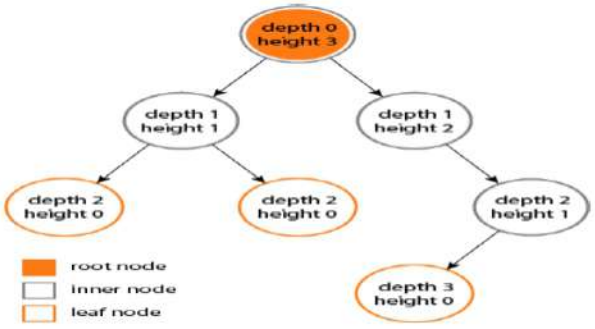
The **height** of a node is the number of edges present in the longest path connecting that node to a leaf node.

Follow the steps below to find the depth of the given node:

- If the tree is empty, print -1.
- Otherwise, perform the following steps:
    1. Initialize a variable, say **dist** as **-1**.
    2. Check if the node **K** is equal to the given node.
    3. Otherwise, check if it is present in either of the **subtrees,** by recursively checking for the left and right subtrees respectively.
    4. If found to be true, print the value of **dist + 1**.
    5. Otherwise, print **dist**.

Follow the steps below to find the height of the given node:

- If the tree is empty, print -1.
- Otherwise, perform the following steps:
    i. Calculate the height of the left subtree recursively.
    ii. Calculate the height of the right subtree recursively.
    iii. Update height of the current node by adding 1 to the maximum of the two heights obtained in the previous step. Store the height in a variable, say **ans**.
    iv. If the current node is equal to the given node **K,** print the value of **ans** as the required answer.



Importance of height of Binary Tree

- Determines the worst-case time complexity of search, insertion, and deletion operations in many tree-based algorithms.
- Provides an estimate of the average path length in the tree, which impacts the efficiency of operations that involve traversing the tree.
- Helps differentiate between balanced and unbalanced trees. Balanced trees, like AVL trees and Red-Black trees, have a logarithmic height, leading to efficient operations.

- Influences the storage space complexity of the tree, as taller trees require more memory to store their nodes.

Importance of Depth of Binary Tree

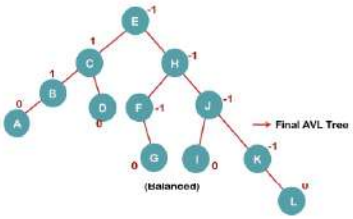
- Provides context to the location of a node within the tree hierarchy.
- Helps identify the level of a node, which can be useful for specific operations like level-order traversal.
- In some algorithms, the depth of a node might be used to determine its priority or influence decision-making during operations.
- Together with the height, the depth helps analyze the overall structure of the tree and understand its properties.

AVL balance Tree

An AVL Tree is a self-balancing Binary Search Tree (BST). This means it automatically maintains a balance between its left and right subtrees, ensuring efficient search, insertion, and deletion operations. Unlike a traditional BST, which can become imbalanced after certain operations, leading to degraded performance, an AVL tree guarantees a maximum height difference of 1 between its subtrees.

Properties

- Balanced: The difference between the heights of the left and right subtrees of any node cannot be more than 1.
- Self-balancing: The tree automatically rebalances itself after insertions and deletions to maintain the balance property.
- Efficient Search: Due to its balanced structure, searching for an element in an AVL tree takes O(log n) time on average.
- Efficient Insertion and Deletion: Insertion and deletion also take O(log n) time on average, making AVL trees efficient for dynamic data structures.



Detection of Unbalance in AVL tree

An AVL (Adelson-Velsky and Landis) tree is a self-balancing binary search tree where the heights of the two child subtrees of any node differ by at most one. The balance factor of a node is the height of its left subtree minus the height of its right subtree (or vice versa).

To detect an imbalance in an AVL tree, you need to check the balance factor of each node and ensure that it satisfies the AVL property. The balance factor of a node should be in the range [-1, 0, 1]. If at any node, the balance factor is not within this range, the tree is unbalanced.

Here are the steps to detect an imbalance in an AVL tree:

1. Calculate the Balance Factor: The balance factor of a node **N** is calculated as follows:

BalanceFactor(N) = Height(LeftSubtree(N)) - Height(RightSubtree(N))

Alternatively, you can use the formula

BalanceFactor(N) = Height(N.left) - Height(N.right)

2. **Check Balance Factor Range:** Ensure that the balance factor of every node in the AVL tree is in the range [-1, 0, 1]. If any node has a balance factor outside this range, the tree is unbalanced.

3. Recursively Check Subtrees: Check the balance factors of each node in a recursive manner, starting from the root and moving down to the leaves. If you encounter a node with an imbalanced factor, you can either perform rotations to restore balance or mark the tree as unbalanced.

This is a basic implementation, and in a real-world scenario, you would likely perform rotations to restore balance when an imbalance is detected. The specific rotation required depends on the type of imbalance (left-left,left-right, right-left, right-right).

Single and double rotation in balancing

In AVL trees, rotations are used to maintain or restore the balance of the tree when an insertion or deletion operation causes an imbalance. There are two primary types of rotations: single rotations and double rotations. The specific rotation needed depends on the type of imbalance introduced.

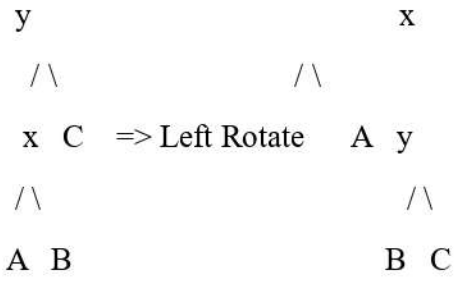
a. Single Rotation

b. Double Rotation

Single Rotation

1. **left Rotation (LL Rotation)**

Imbalance occurs at the left child of the left subtree.



Unit 6

Sorting

Definition

Sorting is the method to arrange and array elements of array in a particular order either in ascending or descending order.

For Example: 4,1,14,7,9,12

➔ 1,4,7,9,12,14.

Sorting can be mainly divided into two types. They are

i. Internal Sorting and

ii. External Sorting

Types of sorting

Sorting algorithms can be broadly categorized into two types based on the location of data during the sorting process: internal sorting and external sorting.

1. **Internal Sorting:**
   - **Definition:** Internal sorting refers to sorting algorithms that operate entirely within the computer's main memory (RAM).
   - **Characteristics:**
     - All data to be sorted is present in the main memory.
     - It is suitable for relatively small datasets that can fit into the available RAM.
     - Examples of internal sorting algorithms include Bubble Sort, Selection Sort, Insertion Sort, Merge Sort, Quick Sort, Heap Sort, Radix Sort, and others.
2. **External Sorting:**
   - **Definition:** External sorting is a technique used when the data to be sorted is too large to fit into the computer's main memory, requiring a combination of internal and external storage (usually disk storage).
   - **Characteristics:**
     - Data is stored on external storage, typically a disk.
     - The algorithm reads chunks of data into memory, sorts them internally, and writes the sorted chunks back to external storage.
     - It is more suitable for large datasets that cannot be accommodated entirely in RAM.
     - Examples of external sorting algorithms include Merge Sort and Polyphase Sort.

Bubble Sort

Bubble Sort is the simplest sorting algorithm that works by repeatedly swapping the adjacent elements if they are in the wrong order. This algorithm is not suitable for large data sets as its average and worst-case time complexity is quite high.

Algorithm

- traverse from left and compare adjacent elements and the higher one is placed at right side.
- In this way, the largest element is moved to the rightmost end at first.
- This process is then continued to find the second largest and place it and so on until the data is sorted.

Insertion Sort

Insertion sort is a simple sorting algorithm that works by building a sorted array one element at a time. It is considered an " in-place " sorting algorithm, meaning it doesn't require any additional memory space beyond the original array.

Algorithm

- We have to start with second element of the array as first element in the array is assumed to be sorted.
- Compare second element with the first element and check if the second element is smaller then swap them.
- Move to the third element and compare it with the second element, then the first element and swap as necessary to put it in the correct position among the first three elements.
- Continue this process, comparing each element with the ones before it and swapping as needed to place it in the correct position among the sorted elements.
- Repeat until the entire array is sorted.

Selection sort

Selection sort is a simple and efficient sorting algorithm that works by repeatedly selecting the smallest (or largest) element from the unsorted portion of the list and moving it to the sorted portion of the list.

Algorithm

1. Set MIN to location 0.
2. Search the minimum element in the list.
3. Swap with value at location MIN.
4. Increment MIN to point to next element.
5. Repeat until the list is sorted.

Quick Sort

Quick Sort is a sorting algorithm based on the Divide and Conquer algorithm that picks an element as a pivot and partitions the given array around the picked pivot by placing the pivot in its correct position in the sorted array.

Algorithm

1. Choose the highest index value has pivot
2. Take two variables to point left and right of the list

excluding pivot

3. Left points to the low index
4. Right points to the high
5. While value at left is less than pivot move right
6. While value at right is greater than pivot move left
7. If both step 5 and step 6 does not match swap left and right
8. If left ≥ right, the point where they met is new pivot.

Merge Sort

Merge sort is a sorting algorithm that follows the **divide-and-conquer** approach. It works by recursively dividing the input array into smaller subarrays and sorting those subarrays then merging them back together to obtain the sorted array.

Algorithm

Step 1: If it is only one element in the list, consider it already

sorted, so return.

Step 2: Divide the list recursively into two halves until it can no

more be divided.

Step 3: Merge the smaller lists into new list in sorted order.

Heap Sort

Heap sort is a comparison-based sorting technique based on Binary Heap data structure. It is similar to the selection sort where we first find the minimum element and place the minimum element at the beginning. Repeat the same process for the remaining elements.

Algorithm

- Builds a heap H from the input data using the **heapify** (explained further into the chapter) method, based on the way of sorting – ascending order or descending order.
- Deletes the root element of the root element and repeats until all the input elements are processed.

Note: (Numerical are not included in this note, practice from your note copy and previous year questions)

Unit 7

Search

Searching Algorithm

Searching algorithms are methods or procedures used to find a specific item or element within a collection of data. These algorithms are widely used in computer science and are crucial for tasks like searching for a particular record in a database, finding an element in a sorted list, or locating a file on a computer. These are some commonly used searching algorithms:

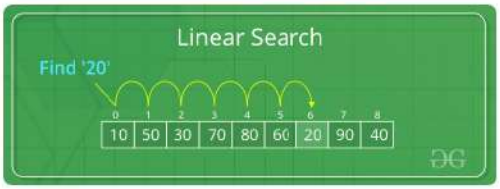
- ❖ Linear Search
- ❖ Binary Search
- ❖ Hashing
- ❖ Tree-based Searching
- ❖ Ternary Search

Searching is mainly classified into two types. They are:

i. Linear or Sequential Search
ii. Non-Linear Search

➢ Sequential Search or Linear Search

Data structure where data elements are arranged sequentially or linearly where each and every element is attached to its previous and next adjacent is called a **linear data structure**. In linear data structure, single level is involved. Therefore, we can traverse all the elements in single run only. Linear data structures are easy to implement because computer memory is arranged in a linear way. Its examples are array, stack, queue, linked list, etc



Advantages

i. Easy to implement and understand.
ii. Works well for unsorted data.
iii. No additional space requirements beyond the data structure itself.

Disadvantages

i. Can be slow for large datasets as it iterates through all elements.
ii. Performance degrades significantly as the data size increases.
iii. Not suitable for sorted data structures where more efficient algorithms like binary search exist.

➢ Non- Linear Search

Data structures where data elements are not arranged sequentially or linearly are called **non-linear data structures**. In a non-linear data structure, single level is not involved. Therefore, we can't traverse all the elements in single run only. Non-linear data structures are not easy to implement in comparison to linear data structure. It utilizes computer memory efficiently in comparison to a linear data structure. Its examples are trees and graphs.

Advantages

i. Efficient Data Representation: Suitable for hierarchical and complex relationships (e.g., trees, graphs).
ii. Optimized Search and Retrieval: Faster search and modification operations (e.g., O(log n) in balanced binary trees).
iii. Improved Performance: Ideal for complex operations such as pathfinding and priority queues (e.g., graphs, heaps).
iv. Models Real-World Relationships: Better at representing real-world networks (e.g., social networks, organizational charts).
v. Better Memory Utilization: Dynamically allocates memory, reducing waste compared to arrays.

Disadvantages

i. Complex Implementation: Harder to code and manage compared to linear structures.
ii. Higher Time Complexity in Some Cases: Certain operations might take more time, especially in unbalanced structures.
iii. More Memory Overhead: Extra memory is needed for pointers/references (e.g., child nodes in trees).
iv. Difficult Debugging and Visualization: Harder to debug and visualize due to complexity.
v. Traversal Complexity: Requires advanced algorithms (e.g., DFS, BFS) for traversal and manipulation.

➢ Binary Search

Binary Search is defined as a searching algorithm used in a sorted array by repeatedly dividing the search interval in half. The idea of binary search is to use the information that the array is sorted and reduce the time complexity to O(log N).

Condition for when to apply Binary Search in a Data Structure.

To apply Binary Search algorithm:

- ❖ The data structure must be sorted.
- ❖ Access to any element of the data structure takes constant time.

➢ **Binary Search Algorithm**

- Divide the search space into two halves by finding the middle index "mid".
- Compare the middle element of the search space with the key.
- If the key is found at middle element, the process is terminated.
- If the key is not found at middle element, choose which half will be used as the next search space.
  - If the key is smaller than the middle element, then the left side is used for next search.
  - If the key is larger than the middle element, then the right side is used for next search.
- This process is continued until the key is found or the total search space is exhausted.

Advantages

- Binary search is faster than linear search, especially for large arrays.
- More efficient than other searching algorithms with a similar time complexity, such as interpolation search or exponential search.
- Reduced Number of Comparisons.

Disadvantages

- Requires Sorted Data:
- Not Suitable for Dynamic Data:
- Limited Applicability:

Tree Search Algorithm

They're algorithms designed to efficiently locate specific keys or values within tree data structures. They leverage the hierarchical organization of trees to navigate and search quickly. They're essential for various applications, including databases, file systems, game AI, and more. Some of the common Tree search algorithm are listed below:

- Depth-First Search (DFS)
- Breadth First Search (BFS)
- Binary Search Tree (BST) Traversal etc.

1. Depth First Search (DFS) Traversal

Depth first Search (DFS) follows first a path from the starting node to an ending none then another path from the start to end and so forth until all the nodes have nodes have been visited.

2. Breadth First Search (BFS) Traversal

In breadth first Search, one node is selected as a start position. It's visited and marked, then all unvisited nodes adjacent of the next node are visited ad marked in the same sequential order.

3. Binary Search Tree (BST)

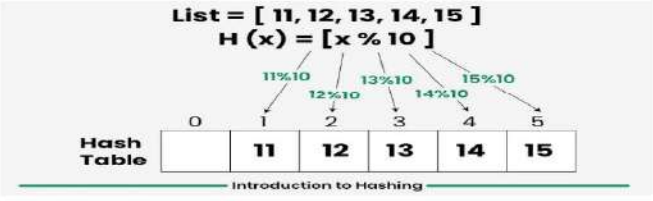
Binary Search Tree (BST) also known as ordered or sorted binary tree, is a rooted binary tree structure with the key of each internal node being greater than all the keys in the respective node's left subtree and less than the ones in its right subtree.

S.N.	BFS	DFS
1.	BFS stands for Breadth First Search.	DFS stands for Depth First Search.
2	BFS(Breadth First Search) uses Queue data structure for finding the shortest path.	DFS(Depth First Search) uses Stack data structure.
3	BFS is a traversal approach in which we first walk through all nodes on the same level before moving on to the next level.	DFS is also a traversal approach in which the traverse begins at the root node and proceeds through the nodes as far as possible until we reach the node with no unvisited nearby nodes.
4	BFS builds the tree level by level	DFS builds the tree sub-tree by sub-tree
5	It works on the concept of FIFO (First In First Out).	It works on the concept of LIFO (Last In First Out).
6	BFS is more suitable for searching vertices closer to the given source.	DFS is more suitable when there are solutions away from source.
7	In BFS there is no concept of backtracking.	DFS algorithm is a recursive algorithm that uses the idea of backtracking

Hashing

Hashing is a technique used in data structures to store and retrieve data efficiently. It involves using a **hash function** to map data items to a fixed-size array which is called a **hash table**. Below are basic terminologies in hashing.

- **Hash Function:** You provide your data items into the hash function.
- **Hash Code:** The hash function crunches the data and give a unique hash code. This hash code is typically integer value that can be used an index.
- **Hash Table:** The hash code then points you to a specific location within the hash table.



Hashing Function and hash table

A hash table is also referred as a hash map (key value pairs) or a hash set (only keys). It uses a hash function to map keys to a fixed-size array, called a hash table. This allows in faster search, insertion, and deletion operations.

The hash function is a function that takes a key and returns an index into the hash table. The goal of a hash function is to distribute keys evenly across the hash table, minimizing collisions (when two keys map to the same index).

Common hash functions include:

1. Division Method: Key % Hash Table Size
2. Multiplication Method: (Key * Constant) % Hash Table Size
3. Universal Hashing: A family of hash functions designed to minimize collisions

Collision in hashing

A hash collision occurs when two different keys map to the same index in a hash table. This can happen even with a good hash function, especially if the hash table is full or the keys are similar.

Causes of Hash Collisions:

1. Poor Hash Function: A hash function that does not distribute keys evenly across the hash table can lead to more collisions.
2. High Load Factor: A high load factor (ratio of keys to hash table size) increases the probability of collisions.
3. Similar Keys: Keys that are similar in value or structure are more likely to collide.

Collision resolution technique

There are two types of collision resolution technique. They are

i. Open Addressing technique
ii. Closed Addressing technique

Open Addressing Technique

Open addressing is a collision resolution technique used in **hash tables** where, instead of storing multiple entries in the same bucket (as in closed addressing), all entries are stored directly in the hash table itself. If a collision occurs (i.e., two keys map to the same hash index), open addressing looks for the next available slot in the table based on a probing sequence.

Advantages

- **Space-efficient**: Since all data is stored within the hash table, there's no need for extra memory for pointers or linked lists (as in closed addressing).
- **Better cache performance**: Data is stored in contiguous memory locations, leading to better cache performance compared to chaining, where linked lists might be scattered in memory.
- **Avoids overhead**: No need for additional structures like linked lists to handle collisions.

Disadvantages

- Clustering: Linear and quadratic probing can lead to clusters of occupied slots, reducing efficiency over time and leading to longer search times as clusters grow.
- Table resizing: Open addressing performs poorly when the hash table becomes too full. To maintain performance, the table often needs to be resized (usually when the load factor exceeds 0.7 or 0.8).
- More complex deletion: When deleting keys, maintaining the integrity of the probing sequence can be challenging.

Closed Addressing Technique

The **closed addressing technique** is a method used in **hash tables** to resolve collisions, which occur when two keys map to the same hash value. The essence of closed addressing is that each hash table entry or bucket contains a list (or another collection) of all elements that hash to the same index.

Advantages of Closed Addressing:

- Efficient use of space: If the number of elements is less than or equal to the number of buckets, you can avoid collisions entirely, but closed addressing allows for efficient handling even when collisions occur.
- Easy to implement: Closed addressing with linked lists is relatively straightforward to code.
- No need for resizing: Unlike other techniques (e.g., open addressing), the table doesn't need resizing since collisions are handled by growing the linked lists.

Disadvantages:

- Potentially slower searches: If many elements collide into the same bucket, searching may degrade to O(n) in the worst case.
- Extra memory: Each bucket requires additional memory for the linked list or other structures, making it less space-efficient compared to some other techniques.

Unit 8

Graphs

Introduction to Graphs

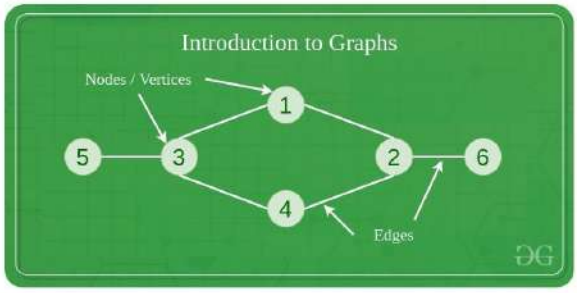
A Graph is a non-linear data structure consisting of vertices and edges. The vertices are sometimes also referred to as nodes and the edges are lines or arcs that connect any two nodes in the graph. More formally a Graph is composed of a set of vertices(V) and a set of edges(E). The graph is denoted by G(E, V). Graphs can be mainly classified into different types. They are:

i. Connected

ii. Unconnected Graph

iii. Directed Graph

iv. Undirected Graph



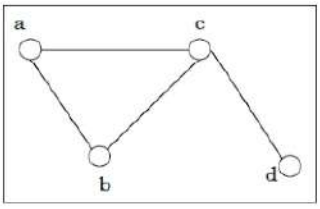
Components of Graphs

- **Vertices:** Vertices are the fundamental units of the graph. Sometimes, vertices are also known as vertex or nodes. Every node/vertex can be labeled or unlabeled.
- **Edges:** Edges are drawn or used to connect two nodes of the graph. It can be ordered pair of nodes in a directed graph. Edges can connect any two nodes in any possible way. There are no rules. Sometimes, edges are also known as arcs. Every edge can be labeled/unlabeled.

Connected graph and Disconnected Graph

A graph G is said to be connected if there exists a path between every pair of vertices.

A graph is said to e disconnected, if it does not contain at least two connected vertices.



Directed and Undirected Graph

A graph in which edges have no direction, i.e., the edges do not have arrows indicating the direction of traversal. Example: A social network graph where friendships are not directional is known as Undirected Graph.

A graph in which edges have a direction, i.e., the edges have arrows indicating the direction of traversal. Example: A web page graph where links between pages are directional.

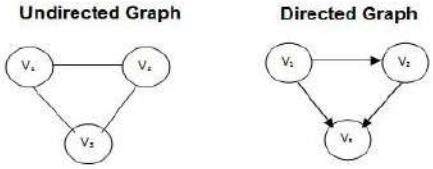


Figure 1: An Undirected Graph Figure 2: A Directed Graph

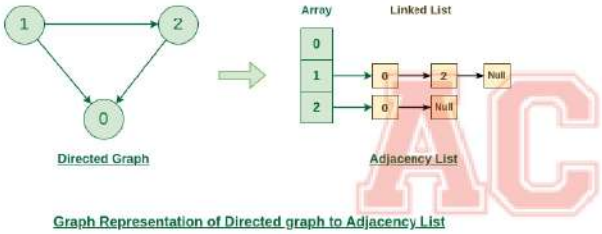
Path and Cycle

Path is a sequence of vertices connected by edges, where no vertex is repeated. It represents a traversal through the graph from a starting vertex to an ending vertex.

Cycle is special kind of path where the starting and ending vertices are the same. It forms a closed loop within the graph.

Adjacency Sets

An adjacency list is a data structure used to represent a graph where each node in the graph stores a list of its neighboring vertices.



Graph Representation of Directed graph to Adjacency List

Characteristics of the Adjacency List:

- The size of the matrix is determined by the number of nodes in the network.
- The number of graph edges is easily computed.
- The adjacency list is a **jagged array**. (member arrays can be of different sizes)

Applications of Adjacency List

- Graph algorithms: Many graph algorithms like Dijkstra's algorithm, Breadth First Search, and Depth First Search use adjacency lists to represent graphs.
- Image Processing: Adjacency lists can be used to represent the adjacency relationships between pixels in an image.
- Game Development: These lists can be used to store information about the connections between different areas or levels the game developers use graphs to represent game maps or levels.

Advantages

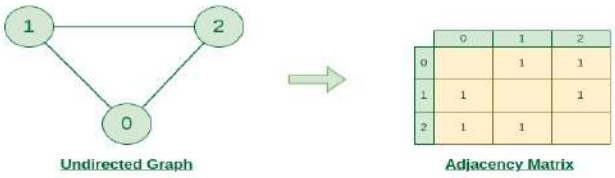
- An adjacency list is simple and easy to understand.
- Adding or removing edges from a graph is quick and easy

Disadvantages

- ❖ In adjacency lists accessing the edges can take longer than the adjacency matrix.
- ❖ It requires more memory than the adjacency matrix for dense graphs.

Adjacency Matrix

An adjacency matrix is a square matrix of N x N size where N is the number of nodes in the graph and it is used to represent the connections between the edges of a graph.



Graph Representation of Undirected graph to Adjacency Matrix

Characteristics of Adjacency Matrix

- ❖ The size of the matrix is determined by the number of vertices in the graph.
- ❖ The number of nodes in the graph determines the size of the matrix.
- ❖ The number of edges in the graph is simply calculated.
- ❖ If the graph has few edges, the matrix will be sparse.

Advantages

- ❖ An adjacency matrix is simple and easy to understand.
- ❖ Adding or removing edges from a graph is quick and easy.
- ❖ It allows constant time access to any edge in the graph

Disadvantages

- It is inefficient in terms of space utilization for sparse graphs because it takes up $O(N^2)$ space.
- Computing all neighbors of a vertex takes O(N) time.

Array based Representation

Arrays-based representation is a fundamental concept in Data structures and Algorithms (DSA). It refers to using arrays to store and manage collection of elements which are discussed listed below:

- ❖ Arrays are linear data structures that hold elements of the same data type.
- ❖ Elements are stored in contiguous memory locations, meaning they are next to each other in memory.
- ❖ Accessing elements is done using an index, which starts from 0 and goes up to the size of the array minus 1.

Advantages

- Efficient random access: Accessing any element takes constant time (O(1)), regardless of its position in the array. This is because we can directly calculate the memory address based on the index.
- Cache-friendly: Due to contiguous memory allocation, data access benefits from CPU cache, improving performance for iterative operations.
- Simple implementation: Implementing basic operations like insertion, deletion, and searching is relatively straightforward with arrays.

Disadvantages

- Fixed size: Once declared, the size of an array cannot be changed in most languages. This can be a limitation if the data size varies dynamically.
- Insertion/deletion: Inserting or deleting elements in the middle can be expensive, as it requires shifting other elements to maintain contiguity.
- Memory waste: Even if unused, the entire allocated memory block remains reserved for the array.

Linked based Representation

It uses linked lists for both vertices and edges. Each vertex node stores data and pointers to its adjacent vertices. Edges can be separate nodes with pointers to the connected vertices.

Advantages

- Dynamic size: Efficiently handles graphs with dynamically changing sizes as adding or removing vertices/edges only involves updating pointers.
- Memory efficiency: Only allocates memory for needed vertices and edges, avoiding waste in sparse graphs.
- Flexibility: Easily represents different graph types (directed, undirected, weighted) by modifying the node structure and edge pointers.

Disadvantages

1. **Memory Overhead**: Linked-based representations require extra memory for storing pointers or references to adjacent nodes. This overhead can be significant, especially for large graphs, as each edge requires additional space.
2. **Slower Access**: Accessing nodes or edges in a linked-based representation typically involves following pointers, which can be slower compared to direct access in array-based representations, especially for large graphs where memory locality becomes important.
3. **Insertion and Deletion Overhead**: Inserting or deleting nodes or edges can be more complex and slower in linked-based representations compared to array-based representations. This is because it involves updating pointers and potentially reorganizing the linked structure.
4. **Cache Inefficiency**: Traversing linked structures may lead to poor cache performance due to non-contiguous memory access patterns. This can result in increased cache misses and slower overall performance, particularly in algorithms that require frequent access to graph elements.
5. **Space Overhead for Pointers**: In addition to the memory overhead for storing actual data, linked-based representations also require additional space for storing pointers or references, which can further increase memory usage.

Mixed Implementation Representation

Mixed implementation representation combines both arrays and linked lists. Usually, an array stores vertices, and each vertex holds a linked list of its adjacent vertices.

Advantages

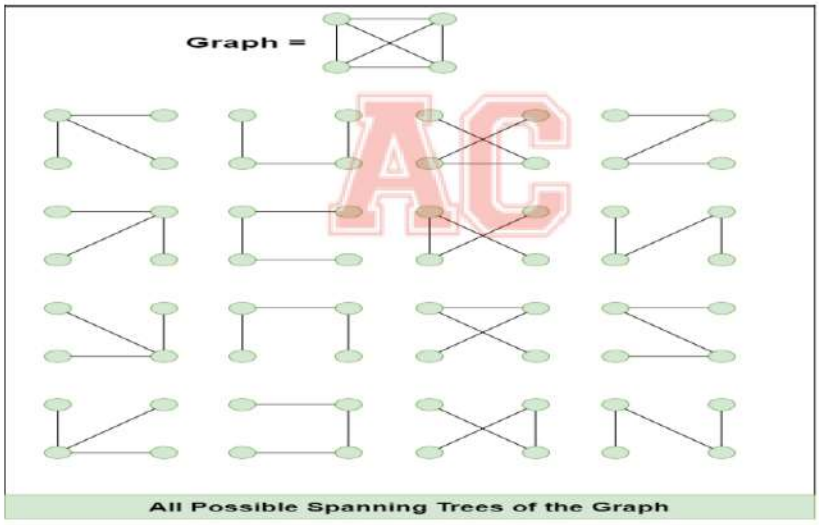
- Balances efficiency and flexibility: Offers random access to vertices (like arrays) while handling dynamic changes efficiently (like linked lists).
- Suitable for specific graph types: Works well for dense graphs (many connections) where array-based access benefits outweigh pointer overhead.

Disadvantages

- Complexity: Still more complex than array-based representation, requiring managing both arrays and linked lists.
- Memory overhead: Might have some unused space in the array even for sparse graphs.

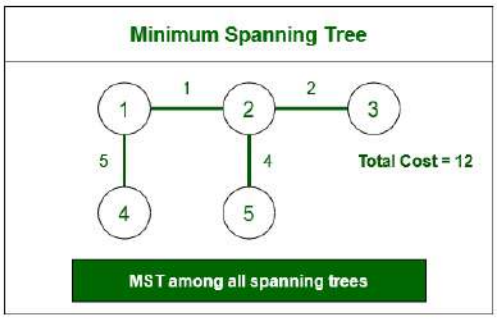
Spanning Tree

A spanning tree is a subset of Graph G, such that all the vertices are connected using minimum possible number of edges. Hence, a spanning tree does not have cycles and a graph may have more than one spanning tree.



Minimum Spanning Tree

The minimum spanning tree has all the properties of a spanning tree with an added constraint of having the minimum possible weights among all possible spanning trees. Like a spanning tree, there can also be many possible MSTs for a graph.



Kruskal's Minimum Spanning Tree Algorithm:

This is one of the popular algorithms for finding the minimum spanning tree from a connected, undirected graph. This is a greedy algorithm. The algorithm workflow is as below:

- First, it sorts all the edges of the graph by their weights,
- Then starts the iterations of finding the spanning tree.
- At each iteration, the algorithm adds the next lowest-weight edge one by one, such that the edges picked until now does not form a cycle.

This algorithm can be implemented efficiently using a DSU (Disjoint-Set) data structure to keep track of the connected components of the graph. This is used in a variety of practical applications such as network design, clustering, and data analysis.

(Numerical Portion taught in class)

Prim's Minimum Spanning Tree Algorithm:

This is also a greedy algorithm. This algorithm has the following workflow:

- It starts by selecting an arbitrary vertex and then adding it to the MST.
- Then, it repeatedly checks for the minimum edge weight that connects one vertex of MST to another vertex that is not yet in the MST.
- This process is continued until all the vertices are included in the MST.

To efficiently select the minimum weight edge for each iteration, this algorithm uses priority_queue to store the vertices sorted by their minimum edge weight currently. It also simultaneously keeps track of the MST using an array or other data structure suitable considering the data type it is storing.

This algorithm can be used in various scenarios such as image segmentation based on color, texture, or other features. For Routing, as in finding the shortest path between two points for a delivery truck to follow.

(Numerical Portion taught in class)

Breadth First Search/Traversal (BFS)

Breadth First Search (BFS) is a fundamental graph traversal algorithm. It begins with a node, then first traverses all its adjacent. Once all adjacent are visited, then their adjacent are traversed. This is different from DFS in a way that closest vertices are visited before others. We mainly traverse vertices level by level. A lot of popular graph algorithms like Dijkstra's shortest path, Kahn's Algorithm, and Prim's algorithm are based on BFS. BFS itself can be used to detect cycle in a directed and undirected graph, find shortest path in an unweighted graph and many more problems.

(Numerical Portion taught in class)

Depth First Search/ Traversal (DFS)

Depth-first search (DFS) is an algorithm for traversing or searching tree or graph data structures. The algorithm starts at the root node (selecting some arbitrary node as the root node in the case of a graph) and explores as far as possible along each branch before backtracking. Extra memory, usually a stack, is needed to keep track of the nodes discovered so far along a specified branch which helps in backtracking of the graph.

Shortest Path Algorithm

The shortest path algorithms are the ones that focuses on calculating the minimum travelling cost from **source node** to **destination node** of a graph in optimal time and space complexities.

Types

As we know there are various types of graphs (weighted, unweighted, negative, cyclic, etc.) therefore having a single algorithm that handles all of them efficiently is not possible. In order to tackle different problems, we have different shortest-path algorithms, which can be categorised into two categories:

1. Depth-First Search (DFS)
2. Breadth-First Search (BFS)
3. Multi-Source BFS
4. Dijkstra's algorithm
5. Bellman-Ford algorithm

Dijkstra's Algorithm

The idea is to generate a **SPT (shortest path tree)** with a given source as a root. Maintain an Adjacency Matrix with two sets,

- one set contains vertices included in the shortest-path tree,
- other set includes vertices not yet included in the shortest-path tree.

At every step of the algorithm, find a vertex that is in the other set (set not yet included) and has a minimum distance from the source.

(Algorithm and Example are taught in class)

Bellman-Ford algorithm

Bellman-Ford is a single source shortest path algorithm that determines the shortest path between a given source vertex and every other vertex in a graph. This algorithm can be used on both weighted and unweighted graphs.

A Bellman-Ford algorithm is also guaranteed to find the shortest path in a graph, similar to Dijkstra's algorithm. Although Bellman-Ford is slower than Dijkstra's algorithm, it is capable of handling graphs with negative edge weights, which makes it more versatile. The shortest path cannot be found if there exists a negative cycle in the graph. If we continue to go around the negative cycle an infinite number of times, then the cost of the path will continue to decrease (even though the length of the path is increasing). As a result, Bellman-Ford is also capable of detecting negative cycles, which is an important feature.