

Data Structures and Algorithms

Nirmal Chenichery

1. Introduction

What is an Algorithm?

In mathematics and computer science, an algorithm is a finite sequence of well-defined instructions, typically used to solve a class of specific problems or to perform a computation. Algorithms are used as specifications for performing calculations and data processing.

In computer programming terms, an algorithm is a set of well-defined instructions to solve a particular problem. It takes a set of input and produces a desired output.

Qualities of Good Algorithms

- o Input and output should be defined precisely.
- o Each step in the algorithm should be clear and unambiguous.
- o Algorithms should be most effective among many different ways to solve a problem.
- o An algorithm shouldn't include computer code. Instead, the algorithm should be written in such a way that it can be used in different programming languages.

Algorithm Examples

Algorithm 1: Add two numbers entered by the user

```

Step 1: Start
Step 2: Declare variables num1, num2 and sum.
Step 3: Read values num1 and num2.
Step 4: Add num1 and num2 and assign the result to sum.
        sum ← num1 + num2
Step 5: Display sum
Step 6: Stop
    
```

Algorithm 2: Find the largest number among three numbers

```

Step 1: Start
Step 2: Declare variables a, b and c.
Step 3: Read variables a, b and c.
Step 4: If a > b
        If a > c
            Display a is the largest number.
        Else
            Display c is the largest number.
    Else
        If b > c
            Display b is the largest number.
        Else
            Display c is the greatest number.
Step 5: Stop
    
```

Algorithm 3: Find Roots of a Quadratic Equation $ax^2 + bx + c = 0$

```

Step 1: Start
Step 2: Declare variables a, b, c, D, x1, x2, rp and ip;
Step 3: Calculate discriminant
        D ← b2-4ac
Step 4: If D ≥ 0
        r1 ← (-b+√D)/2a
        r2 ← (-b-√D)/2a
    
```

```

        Display r1 and r2 as roots.
Else
    Calculate real part and imaginary part
    rp ← -b/2a
    ip ← √(-D)/2a
    Display rp+j(ip) and rp-j(ip) as roots
Step 5: Stop

```

Algorithm 4: Find the factorial of a number

```

Step 1: Start
Step 2: Declare variables n, factorial and i.
Step 3: Initialize variables
        factorial ← 1
        i ← 1
Step 4: Read value of n
Step 5: Repeat the steps until i = n
    5.1: factorial ← factorial*i
    5.2: i ← i+1
Step 6: Display factorial
Step 7: Stop

```

Algorithm 5: Check whether a number is prime or not

```

Step 1: Start
Step 2: Declare variables n, i, flag.
Step 3: Initialize variables
        flag ← 1
        i ← 2
Step 4: Read n from the user.
Step 5: Repeat the steps until i=(n/2)
    5.1 If remainder of n÷i equals 0
        flag ← 0
        Go to step 6
    5.2 i ← i+1
Step 6: If flag = 0
        Display n is not prime
    else
        Display n is prime
Step 7: Stop

```

Algorithm 6: Find the Fibonacci series till the term less than 1000

```

Step 1: Start
Step 2: Declare variables first_term,second_term and temp.
Step 3: Initialize variables first_term ← 0 second_term ← 1
Step 4: Display first_term and second_term
Step 5: Repeat the steps until second_term ≤ 1000
    5.1: temp ← second_term
    5.2: second_term ← second_term + first_term
    5.3: first_term ← temp
    5.4: Display second_term
Step 6: Stop

```

Data Structure and Types

What are Data Structures?

Data Structures are a specialized means of organizing and storing data in computers in such a way that we can perform operations on the stored data more efficiently.

In computer science, a data structure is a data organization, management, and storage format that enable efficient access and modification. More precisely, a data structure is a collection of data values, the relationships among them, and the functions or operations that can be applied to the data, i.e., it is an algebraic structure about data.

In computer terms, a data structure is a Specific way to store and organize data in a computer's memory so that these data can be used efficiently later.

Depending on your requirement and project, it is important to choose the right data structure for your project. For example, if you want to store data sequentially in the memory, then you can go for the Array data structure.

Array data Structure



Note: Data structure and data types are slightly different. **Data structure is the collection of data types arranged in a specific order.**

Types of Data Structure

Basically, data structures are divided into two categories:

- o Linear data structure
- o Non-linear data structure

Linear Data Structures

In linear data structures, **the elements are arranged in sequence one after the other**. Since elements are arranged in particular order, they are easy to implement.

However, when the complexity of the program increases, the linear data structures might not be the best choice because of operational complexities.

Array

In an array, **elements in memory are arranged in continuous memory. All the elements of an array are of the same type**. And, the type of elements that can be stored in the form of arrays is determined by the programming language.

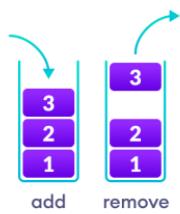
2	1	5	3	4
0	1	2	3	4

index

Stack

In stack data structure, elements are stored in the LIFO principle. That is, the last element stored in a stack will be removed first.

It works just like a pile of plates where the last plate kept on the pile will be removed first.



Queue

Unlike stack, **the queue data structure works in the FIFO principle where first element stored in the queue will be removed first**.

It works just like a queue of people in the ticket counter where first person on the queue will get the ticket first.



Linked List

In linked list data structure, **data elements are connected through a series of nodes**. And, each node contains the data items and address to the next node.



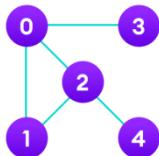
Non Linear data structures

Unlike linear data structures, **elements in non-linear data structures are not in any sequence**. Instead they are arranged in a hierarchical manner where one element will be connected to one or more elements.

Non-linear data structures are further divided into **graph** and **tree based** data structures.

Graph

In graph data structure, each node is called **vertex** and each vertex is connected to other vertices through **edges**.

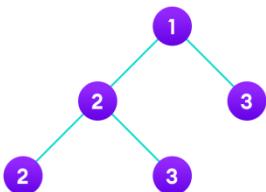


Graph Based Data Structures

- Spanning Tree and Minimum Spanning Tree
- Strongly Connected Components
- Adjacency Matrix
- Adjacency List

Trees

Similar to a graph, **a tree is also a collection of vertices and edges**. However, in tree data structure, **there can only be one edge between two vertices**.



Tree based Data Structure

- Binary Tree
- Binary Search Tree
- AVL Tree
- B-Tree
- B+ Tree
- Red-Black Tree

Linear Vs. Non-linear Data Structures

Now that we know about linear and non-linear data structures, let's see the major differences between them.

Linear Data Structures	Non Linear Data Structures
The data items are arranged in sequential order, one after the other.	The data items are arranged in non-sequential order (hierarchical manner).
All the items are present on the single layer.	The data items are present at different layers.
It can be traversed on a single run. That is, if we start from the first element, we can traverse all the elements sequentially in a single pass.	It requires multiple runs. That is, if we start from the first element it might not be possible to traverse all the elements in a single pass.
The memory utilization is not efficient.	Different structures utilize memory in different efficient ways depending on the need.
The time complexity increase with the data size.	Time complexity remains the same.
Example: Arrays, Stack, Queue	Example: Tree, Graph, Map

Why Data Structure?

Knowledge about data structures help you understand the working of each data structure. And, based on that you can select the right data structures for your project.

This helps you write memory and time efficient code.

Why Learn Data Structures and Algorithms?

we will learn why every programmer should learn data structures and algorithms with the help of examples.

What are Algorithms?

Informally, an algorithm is nothing but a mention **of steps to solve a problem**. They are essentially a solution.

For example, an algorithm to solve the problem of factorials might look something like this:

Problem: Find the factorial of n

```
Initialize fact = 1
For every value v in range 1 to n:
    Multiply the fact by v
fact contains the factorial of n
```

Here, the algorithm is written in English. If it was written in a programming language, we would call it to code instead. Here is a code for finding the factorial of a number in C++.

```
int factorial(int n) {
    int fact = 1;
    for (int v = 1; v <= n; v++) {
        fact = fact * v;
    }
    return fact;
}
```

Programming is all about data structures and algorithms. Data structures are used to hold data while algorithms are used to solve the problem using that data.

Data structures and algorithms (DSA) goes through solutions to standard problems in detail and gives you an insight into how efficient it is to use each one of them. It also teaches you the science of evaluating the efficiency of an algorithm. This enables you to choose the best of various choices.

Watch Videos:-

[Introduction to Algorithms](#)

Priori Analysis and Posteriori Testing

Priori Analysis

Priori analysis is the analysis of an algorithm by studying it deeply to know how it works and to get the time and space complexity.

Posteriori Testing

Posteriori testing on the other hand is done on the program by executing the to check how the speed and performance in terms of how long it takes to run and the number of bytes consumed.

What is the Difference Between Priori Analysis and Posteriori Testing?

The main difference between Priori Analysis and Posteriori Testing is that Priori Analysis is speed and performance evaluation done on algorithms by testing for time and space complexity, while posteriori testing is speed and performance evaluation done on programs by testing the time and bytes consumed.

The difference simply follows the difference between an algorithm and a program.

Priori Analysis	Posteriori Testing
Performed on algorithms	Performed on programs
Does not depend on any language	Depends on a language
Does not rely on the use of a computer hardware, can be done with pen and paper	Requires the use of a computer hardware to implement it efficiently
Checks for time and space complexities	Checks for time and bytes consumed

Watch Videos:-

[Priori Analysis and Posteriori Testing](#)

Characteristics of an Algorithm

- Input: An algorithm **requires some input values**. An algorithm can be given a value other than 0 or more input.
- Output: At the end of an algorithm, you will have **one or more outcomes**.
- Definiteness / Unambiguity: **A perfect algorithm is defined as unambiguous / definite**, which means that its instructions should be clear and straightforward.
- Finiteness: **An algorithm must be finite**. Finiteness in this context means that the algorithm should have a limited number of instructions, i.e., the instructions should be countable.
- Effectiveness: Because each instruction in an algorithm affects the overall process, it should be adequate.
- Language independence: An algorithm must be language-independent, which means that its instructions can be implemented in any language and produce the same results.

Watch Videos:-

[Characteristics of Algorithm](#)

How to write an Algorithm

Algorithm is generally developed before the actual coding is done. It is written using English like language so that it is easily understandable even by non-programmers.

These are the characteristics of a good and correct algorithm –

- o Has a set of inputs
- o Steps are uniquely defined
- o Has finite number of steps
- o Produces desired output

Example

Let us first take an example of a real-life situation for creating algorithm. Here is the algorithm for going to the market to purchase a pen.

1. Get dressed to go the market
2. Check your wallet for money
3. If there is no money in the wallet, replenish it.
4. Go to the shop
5. Ask for your favorite brand of pen
6. If pen is not available, go to step 7 else go to step 10
7. Give money to the shopkeeper
8. Keep the purchased pen safely
9. Go back home
10. As for any other brand of pen
11. Go to Step 7

Add two numbers entered by the user

Step 1: Start

Step 2: Declare variables num1, num2 and sum.

Step 3: Read values num1 and num2.

Step 4: Add num1 and num2 and assign the result to sum.

sum ← num1 + num2

Step 5: Display sum

Step 6: Stop

Watch Videos:-

- [How to write an Algorithm](#)
- [How Write and Analyze Algorithm](#)

Frequency Count Method

Frequency count method can be known by assigning one unit of time for each statements and if any statement is repeating for some number of times the frequency of statements ,frequency of execution of that statement will be calculate and we find the time taken by the algorithm

```
Algorithm sum (A, n)
{
    s = 0; → 1 (times)
    for (i=0;i<n;i++) → n+1
    {
        s = s+A[i]; → n
    }
    return s; → 1
}

time   f(n) = 2n+3
degree  O(n)
```

Space complexity

Space used

A → n
 n → 1
 s → 1
 i → 1

s(n) = n + 3 space complexity

Watch Videos:-

[Frequency Count Method](#)
[Time Complexity #1](#)
[Time Complexity Example #2](#)
[Time Complexity of While and if #3](#)

Time complexities of different data structures

Time Complexity is a concept in computer science that deals with the quantification of the amount of time taken by a set of code or algorithm to process or run as a function of the amount of input. In other words, the time complexity is how long a program takes to process a given input. The efficiency of an algorithm depends on two parameters:

- Time Complexity
- Space Complexity

Time Complexity: It is defined as the number of times a particular instruction set is executed rather than the total time taken. It is because the total time taken also depends on some external factors like the compiler used, the processor's speed, etc.

Space Complexity: It is the total memory space required by the program for its execution.

Best case time complexity of different data structures for different operations

Data structure	Access	Search	Insertion	Deletion
Array	O(1)	O(1)	O(1)	O(1)
Stack	O(1)	O(1)	O(1)	O(1)
Queue	O(1)	O(1)	O(1)	O(1)
Singly Linked list	O(1)	O(1)	O(1)	O(1)
Doubly Linked List	O(1)	O(1)	O(1)	O(1)
Hash Table	O(1)	O(1)	O(1)	O(1)
Binary Search				
Tree	O(log n)	O(log n)	O(log n)	O(log n)
AVL Tree	O(log n)	O(log n)	O(log n)	O(log n)
B Tree	O(log n)	O(log n)	O(log n)	O(log n)
Red Black Tree	O(log n)	O(log n)	O(log n)	O(log n)

Worst Case time complexity of different data structures for different operations

Data structure	Access	Search	Insertion	Deletion
Array	O(1)	O(N)	O(N)	O(N)
Stack	O(N)	O(N)	O(1)	O(1)
Queue	O(N)	O(N)	O(1)	O(1)
Singly Linked list	O(N)	O(N)	O(N)	O(N)
Doubly Linked List	O(N)	O(N)	O(1)	O(1)
Hash Table	O(N)	O(N)	O(N)	O(N)
Binary Search				
Tree	O(N)	O(N)	O(N)	O(N)
AVL Tree	O(log N)	O(log N)	O(log N)	O(log N)
Binary Tree	O(N)	O(N)	O(N)	O(N)
Red Black Tree	O(log N)	O(log N)	O(log N)	O(log N)

The average time complexity of different data structures for different operations

Data structure	Access	Search	Insertion	Deletion
Array	O(1)	O(N)	O(N)	O(N)
Stack	O(N)	O(N)	O(1)	O(1)
Queue	O(N)	O(N)	O(1)	O(1)
Singly Linked list	O(N)	O(N)	O(1)	O(1)
Doubly Linked List	O(N)	O(N)	O(1)	O(1)
Hash Table	O(1)	O(1)	O(1)	O(1)
Binary Search Tree	O(log N)	O(log N)	O(log N)	O(log N)
AVL Tree	O(log N)	O(log N)	O(log N)	O(log N)
B Tree	O(log N)	O(log N)	O(log N)	O(log N)
Red Black Tree	O(log N)	O(log N)	O(log N)	O(log N)

Classes of functions – Types of Time Functions

O(1)	Constant
O(log n)	Logarithmic
O(n)	Linear
O(n^2)	Quadratic
O(n^3)	Cubic
O(2^n)	Exponential

Watch Videos:-

[Classes of functions](#)

[Compare Class of Functions](#)

Asymptotic Analysis: Big-O Notation and More

The efficiency of an algorithm depends on the amount of time, storage and other resources required to execute the algorithm. The efficiency is measured with the help of asymptotic notations.

An algorithm may not have the same performance for different types of inputs. With the increase in the input size, the performance will change.

The study of change in performance of the algorithm with the change in the order of the input size is defined as asymptotic analysis.

Asymptotic Notations

Asymptotic notations are the mathematical notations used to describe the running time of an algorithm when the input tends towards a particular value or a limiting value.

For example: In bubble sort, when the input array is already sorted, the time taken by the algorithm is linear i.e. the best case.

But, when the input array is in reverse condition, the algorithm takes the maximum time (quadratic) to sort the elements i.e. the worst case.

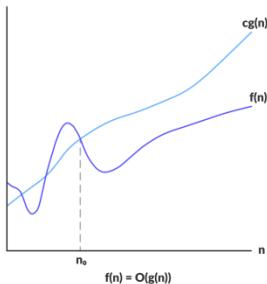
When the input array is neither sorted nor in reverse order, then it takes average time. These durations are denoted using asymptotic notations.

There are mainly three asymptotic notations:

- o Big-O notation
- o Omega notation
- o Theta notation

Big-O Notation (O-notation)

Big-O notation represents the upper bound of the running time of an algorithm. Thus, it gives the worst-case complexity of an algorithm.



Big-O gives the upper bound of a function

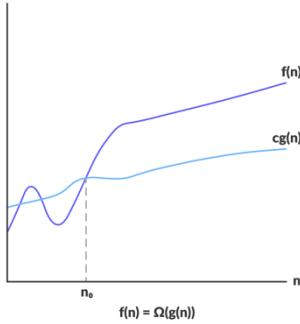
$$O(g(n)) = \{ f(n) : \text{there exist positive constants } c \text{ and } n_0 \text{ such that } 0 \leq f(n) \leq cg(n) \text{ for all } n \geq n_0 \}$$

The above expression can be described as a function $f(n)$ belongs to the set $O(g(n))$ if there exists a positive constant c such that it lies between 0 and $cg(n)$, for sufficiently large n .

For any value of n , the running time of an algorithm does not cross the time provided by $O(g(n))$. Since it gives the worst-case running time of an algorithm, it is widely used to analyze an algorithm as we are always interested in the worst-case scenario.

Omega Notation (Ω -notation)

Omega notation represents the lower bound of the running time of an algorithm. Thus, it provides the best case complexity of an algorithm.



Omega gives the lower bound of a function

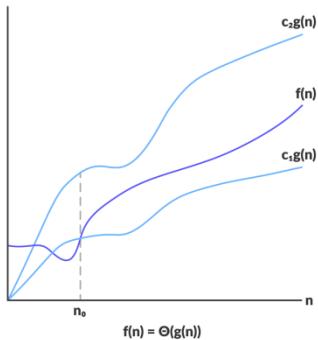
$$\Omega(g(n)) = \{ f(n) : \text{there exist positive constants } c \text{ and } n_0 \text{ such that } 0 \leq cg(n) \leq f(n) \text{ for all } n \geq n_0 \}$$

The above expression can be described as a function $f(n)$ belongs to the set $\Omega(g(n))$ if there exists a positive constant c such that it lies above $cg(n)$, for sufficiently large n .

For any value of n , the minimum time required by the algorithm is given by Omega $\Omega(g(n))$.

Theta Notation (Θ -notation)

Theta notation encloses the function from above and below. Since it represents the upper and the lower bound of the running time of an algorithm, it is used for analyzing the average-case complexity of an algorithm.



Theta bounds the function within constants factors

For a function $g(n)$, $\Theta(g(n))$ is given by the relation:

$$\Theta(g(n)) = \{ f(n) : \text{there exist positive constants } c_1, c_2 \text{ and } n_0 \text{ such that } 0 \leq c_1g(n) \leq f(n) \leq c_2g(n) \text{ for all } n \geq n_0 \}$$

The above expression can be described as a function $f(n)$ belongs to the set $\Theta(g(n))$ if there exist positive constants c_1 and c_2 such that it can be sandwiched between $c_1g(n)$ and $c_2g(n)$, for sufficiently large n .

If a function $f(n)$ lies anywhere in between $c_1g(n)$ and $c_2g(n)$ for all $n \geq n_0$, then $f(n)$ is said to be asymptotically tight bound.

Watch Video:-

[Asymptotic Notations Big Oh - Omega - Theta #1](#)
[Asymptotic Notations - Big Oh - Omega - Theta #2](#)
[Properties of Asymptotic Notations](#)

Comparisons of Functions

Compare two any functions

- o n^2 n^3
- o Apply log on both side

Watch Video:-

[Comparison of Functions #1](#)

[Comparison of Functions #2](#)

Best Worst and Average Case Analysis

Worst Case Analysis (Mostly used)

In the worst-case analysis, we calculate the upper bound on the running time of an algorithm. We must know the case that causes a maximum number of operations to be executed.

For Linear Search, **the worst case happens when the element to be searched (x) is not present in the array.** When x is not present, the search() function compares it with all the elements of arr[] one by one. Therefore, the worst-case time complexity of the linear search would be O(n).

Best Case Analysis (Very Rarely used)

In the best case analysis, we calculate the lower bound on the running time of an algorithm. We must know the case that causes a minimum number of operations to be executed.

In the linear search problem, **the best case occurs when x is present at the first location.** The number of operations in the best case is constant (not dependent on n). So time complexity in the best case would be $\Omega(1)$

Average Case Analysis (Rarely used)

In average case analysis, we take all possible inputs and calculate the computing time for all of the inputs. Sum all the calculated values and divide the sum by the total number of inputs. We must know (or predict) the distribution of cases.

For the linear search problem, let us assume that all cases are uniformly distributed (including the case of x not being present in the array). So we sum all the cases and divide the sum by (n+1).

Watch Video:-

[Best Worst and Average Case Analysis](#)

Master Theorem

The master method is a formula for solving recurrence relations of the form:

$$T(n) = aT(n/b) + f(n),$$

where,

n = size of input

a = number of subproblems in the recursion

n/b = size of each subproblem. All subproblems are assumed to have the same size.

$f(n)$ = cost of the work done outside the recursive call, which includes the cost of dividing the problem and cost of merging the solutions

Here, $a \geq 1$ and $b > 1$ are constants, and $f(n)$ is an asymptotically positive function.

An asymptotically positive function means that for a sufficiently large value of n , we have $f(n) > 0$.

The master theorem is used in calculating the time complexity of recurrence relations (divide and conquer algorithms) in a simple and quick way.

Master Theorem

If $a \geq 1$ and $b > 1$ are constants and $f(n)$ is an asymptotically positive function, then the time complexity of a recursive relation is given by

$$T(n) = aT(n/b) + f(n)$$

where, $T(n)$ has the following asymptotic bounds:

1. If $f(n) = O(n \log_b a - \epsilon)$, then $T(n) = \Theta(n \log_b a)$.
2. If $f(n) = \Theta(n \log_b a)$, then $T(n) = \Theta(n \log_b a * \log n)$.
3. If $f(n) = \Omega(n \log_b a + \epsilon)$, then $T(n) = \Theta(f(n))$.

$\epsilon > 0$ is a constant.

Each of the above conditions can be interpreted as:

1. If the cost of solving the sub-problems at each level increases by a certain factor, the value of $f(n)$ will become polynomially smaller than $n^{\log_b a}$. Thus, the time complexity is oppressed by the cost of the last level ie. $n^{\log_b a}$
2. If the cost of solving the sub-problem at each level is nearly equal, then the value of $f(n)$ will be $n^{\log_b a}$. Thus, the time complexity will be $f(n)$ times the total number of levels ie. $n^{\log_b a} * \log n$
3. If the cost of solving the subproblems at each level decreases by a certain factor, the value of $f(n)$ will become polynomially larger than $n^{\log_b a}$. Thus, the time complexity is oppressed by the cost of $f(n)$.

Solved Example of Master Theorem

$$T(n) = 3T(n/2) + n^2$$

Here,

$a = 3$

$n/b = n/2$

$f(n) = n^2$

$$\log_b a = \log_2 3 \approx 1.58 < 2$$

ie. $f(n) < n \log_b a + \epsilon$, where, ϵ is a constant.

Case 3 implies here.

Thus, $T(n) = f(n) = \Theta(n^2)$

Master Theorem Limitations

The master theorem cannot be used if:

- o $T(n)$ is not monotone. eg. $T(n) = \sin n$
- o $f(n)$ is not a polynomial. eg. $f(n) = 2n$
- o a is not a constant. eg. $a = 2n$
- o $a < 1$

Watch Video:-

[Masters Theorem in Algorithms for Dividing Function Examples for Master Theorem](#)

Divide and Conquer Algorithm

We will also compare the divide and conquer approach versus other approaches to solve a recursive problem.

A divide and conquer algorithm is a strategy of solving a large problem by

- breaking the problem into smaller sub-problems
- solving the sub-problems, and
- combining them to get the desired output.

How Divide and Conquer Algorithms Work?

Here are the steps involved:

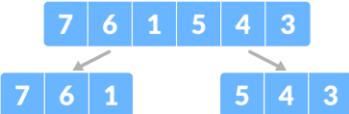
- **Divide:** Divide the given problem into sub-problems using recursion.
- **Conquer:** Solve the smaller sub-problems recursively. If the subproblem is small enough, then solve it directly.
- **Combine:** Combine the solutions of the sub-problems that are part of the recursive process to solve the actual problem.

Here, we will sort an array using the divide and conquer approach (ie. [merge sort](#)).

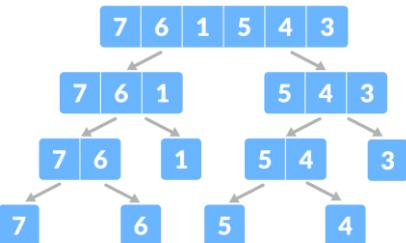
1. Let the given array be:

7	6	1	5	4	3
---	---	---	---	---	---

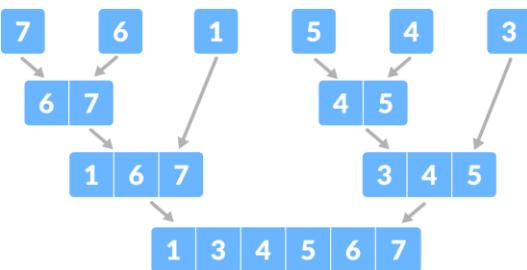
2. **Divide** the array into two halves.



Again, divide each subpart recursively into two halves until you get individual elements.



3. Now, **combine** the individual elements in a sorted manner. Here, conquer and combine steps go side by side.



Divide and Conquer Vs Dynamic Approach

The divide and conquer approach divides a problem into smaller subproblems; these subproblems are further solved recursively. The result of each subproblem is not stored for future reference, whereas, in a dynamic approach, the result of each subproblem is stored for future reference.

Use the divide and conquer approach when the same subproblem is not solved multiple times. Use the dynamic approach when the result of a subproblem is to be used multiple times in the future.

Let us understand this with an example. Suppose we are trying to find the Fibonacci series. Then,

Divide and Conquer approach:

```
fib(n)
  If n < 2, return 1
  Else , return f(n - 1) + f(n -2)
```

Dynamic approach:

```
mem = []
fib(n)
  If n in mem: return mem[n]
  else,
    If n < 2, f = 1
    else , f = f(n - 1) + f(n -2)
    mem[n] = f
  return f
```

In a dynamic approach, mem stores the result of each subproblem.

Advantages of Divide and Conquer Algorithm

- The complexity for the multiplication of two matrices using the naive method is $O(n^3)$, whereas using the divide and conquer approach (i.e. Strassen's matrix multiplication) is $O(n^{2.8074})$. This approach also simplifies other problems, such as the Tower of Hanoi.
- This approach is suitable for multiprocessing systems.
- It makes efficient use of memory caches.

Divide and Conquer Applications

- Binary Search
- Merge Sort
- Quick Sort
- Strassen's Matrix multiplication
- Karatsuba Algorithm

Watch Videos:-

[Divide And Conquer](#)

2. Data Structures(I)

Stack

In stack data structure, elements are stored in the LIFO principle. That is, the last element stored in a stack will be removed first.

A stack is a **LIFO** (Last In First Out) — the element placed at last can be accessed at first) structure which can be commonly found in many programming languages. This structure is named as “stack” because it resembles a real-world stack — a stack of plates.

You can think of the stack data structure as the pile of plates on top of another.

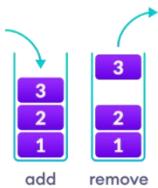


This means the last element inserted inside the stack is removed first. Insertion and deletion are possible on one end (top).

Stack representation similar to a pile of plate. Here, you can:

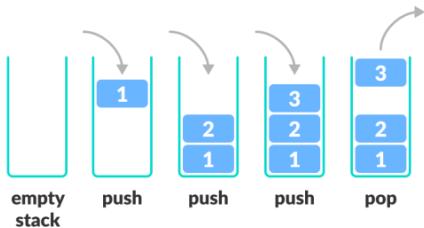
- Put a new plate on top
- Remove the top plate

And, if you want the plate at the bottom, you must first remove all the plates on top. This is exactly how the stack data structure works.



LIFO Principle of Stack

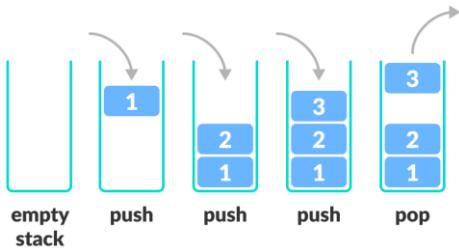
In programming terms, putting an item on top of the stack is called **push** and removing an item is called **pop**.



In the above image, although item 3 was kept last, it was removed first. This is exactly how the LIFO (Last In First Out) Principle works.

Basic Operations of Stack

- **Push:** Insert an element on to the top of the stack.
- **Pop:** Delete the topmost element and return it.



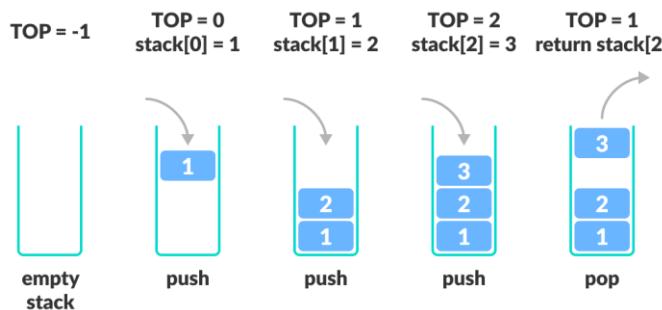
Furthermore, the following additional functions are provided for a stack in order to check its status.

- **Peek:** Return the top element of the stack **without deleting it**.
- **isEmpty:** Check if the **stack is empty**.
- **isFull:** Check if the **stack is full**.

Working of Stack Data Structure

The operations work as follows:

1. A pointer called **TOP** is used to keep track of the top element in the stack.
2. When **initializing the stack**, we set its value to **-1** so that we can check if the stack is empty by comparing **TOP == -1**.
3. **On pushing an element**, we increase the value of **TOP** and place the new element in the position pointed to by **TOP**.
4. **On popping an element**, we return the element pointed to by **TOP** and reduce its value.
5. Before pushing, we check if the stack is already full
6. Before popping, we check if the stack is already empty



Applications of Stacks

- Used for expression evaluation (e.g.: shunting-yard algorithm for parsing and evaluating mathematical expressions, Postfix and Infix expression).
- Used to implement function calls in recursion programming.
- Reverse a string
- Undo mechanism in text editor
- Balance of parenthesis
- Infix to Postfix conversion

Infix Prefix and Postfix Expressions

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

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

These notations are named as how they use operator in expression.

Infix Notation

We write expression in infix notation, e.g. $a - b + c$, where **operators are used in-between operands**.

Prefix Notation

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

Postfix Notation

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

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

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

Parsing Expressions

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

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

Precedence

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

For example –

$$a + b * c \rightarrow a + (b * c)$$

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

Associativity

Associativity describes the rule where operators with the same precedence appear in an expression.

For example, in expression $a + b - c$, both $+$ and $-$ have the same precedence, then which part of the expression will be evaluated first, is determined by associativity of those operators. Here, both $+$ and $-$ are left associative, so the expression will be evaluated as $(a + b) - c$.

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

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

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

For example –

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

Infix to Postfix Conversion

1. Print operand as they arrive
 2. If stack is empty or contains a left parenthesis on top, push the incoming operator on to the stack
 3. If Incoming symbol is '(' push it into stack.
 4. If incoming symbol is ')' pop the stack and print the operator until left parenthesis is found.
 5. If incoming symbol has higher precedence than the top of the stack, push it on the stack.
 6. If incoming symbol has lower precedence than the top of the stack, pop and print the top. Then test the incoming operator against the new top of the stack.
 7. If incoming operator has equal precedence with the top of the stack, use **associative** rule.
 8. At the end of the expression, pop and print all the operator of the stack.
- ⇒ Associativity **L to R** then pop and print the top of the stack and then push the incoming operator
 ⇒ **R to L** then push the incoming operator

$A + B / C \rightarrow A B C / +$

$A - B / C * D + E \rightarrow A B C / D * - E +$

Infix to Postfix conversion using Stack

$K+L-M*N+(O^P)*W/U/V*T+Q$

Operator	Precedence	Associativity
()	Highest	
$^$	Second	R - L
* and /	Third	L - R
+ and -	Lowest	L - R

Incoming	Stack	Prefix
K		K
+	+	K
L	+	KL
-	-	KL+
M	-	KL+M
*	-*	KL+M
N	-*	KL+MN
+	+	KL+MN*-
(+(KL+MN*-
O	+(KL+MN*-O
$^$	+($^$	KL+MN*-O
P	+($^$	KL+MN*-OP
)	+	KL+MN*-OP $^$
*	+*	KL+MN*-OP $^$
W	+*	KL+MN*-OP $^$ W
/	+/	KL+MN*-OP $^$ W*
U	+/	KL+MN*-OP $^$ W*U
/	+/	KL+MN*-OP $^$ W*U/
V	+/	KL+MN*-OP $^$ W*U/V
*	+*	KL+MN*-OP $^$ W*U/V/
T	+*	KL+MN*-OP $^$ W*U/V/T
+	+	KL+MN*-OP $^$ W*U/V/T*+
Q	+	KL+MN*-OP $^$ W*U/V/T*+Q
		KL+MN*-OP $^$ W*U/V/T*+Q+

Question : $A+B+(M^N)*(O+P)-Q/R^S*T+Z$

Answer : $AB-MN^OP+++QRS^/T*-Z+$

Infix to Prefix conversion

1. First, reverse the infix expression given in the problem.
2. Scan the expression from left to right.
3. Whenever the operands arrive, print them.
4. If the operator arrives and the stack is found to be empty, then simply push the operator into the stack.
5. If the incoming operator has higher precedence than the TOP of the stack, push the incoming operator into the stack.
6. If the incoming operator has the same precedence with a TOP of the stack, push the incoming operator into the stack.
7. If the incoming operator has lower precedence than the TOP of the stack, pop, and print the top of the stack. Test the incoming operator against the top of the stack again and pop the operator from the stack till it finds the operator of a lower precedence or same precedence.
8. If the incoming operator has the same precedence with the top of the stack and the incoming operator is $^$, then pop the top of the stack till the condition is true. If the condition is not true, push the $^$ operator.

9. When we reach the end of the expression, pop, and print all the operators from the top of the stack.
10. If the operator is ')', then push it into the stack.
11. If the operator is '(', then pop all the operators from the stack till it finds ')' opening bracket in the stack.
12. If the top of the stack is ')', push the operator on the stack.
13. At the end, reverse the output.

Infix to Prefix conversion using Stack

K+L-M*N+(O^P)*W/U/V*T+Q

Operator	Precedence	Associativity
()	Highest	
^	Second	R - L
* and /	Third	L - R
+ and -	Lowest	L - R

Reversed expression

Q+T*V/U/W*) P^O (+N*M-L+K

Incoming	Stack	Prefix
Q		Q
+	+	Q
T	+	QT
*	+	QT
V	+	QTV
/	+/	QTV
U	+/	QTVU
/	+/ /	QTVU
W	+/ /	QTVUW
*	+/ /*	QTVUW
)	+/ /*)	QTVUW
P	+/ /*)	QTVUWP
^	+/ /*)^	QTVUWP
O	+/ /*)^	QTVUWPO
(+/ /*	QTVUWPO^
+	++	QTVUWPO^*/ //*
N	++	QTVUWPO^*///*N
*	++*	QTVUWPO^*///*N
M	++*	QTVUWPO^*///*NM
-	++-	QTVUWPO^*///*NM*
L	++-	QTVUWPO^*///*NM*L
+	++-+	QTVUWPO^*///*NM*L
K	++-+	QTVUWPO^*///*NM*LK
		QTVUWPO^*///*NM*LK+--+

Reversed: +-+KL*MN*/*^OPWUVTQ

Evaluation of Prefix Expression

Scan Prefix expression from right to left

For each char in prefix expression

Do

```

        If operand is there, push into stack
        Else if operator is there, pop 2 elements
            Op1= top element
            Op2 = next to top element
            Result = Op1operator Op2
        Push result into stack
    Return stack [top]

```

Example

Infix Expression \rightarrow $a + b * c - d / e ^ f$

$a = 2, b = 3, c = 4, d = 16, e = 2, f = 3$

		Conversion
Prefix expression \rightarrow	$- + a * b c / d ^ e f$	
	$- + 2 * 3 4 / 16 ^ 2 3$	$- + 2 * 3 4 / 16 2 ^ 3$
	$- + 2 * 3 4 / 16 8$	$- + 2 * 3 4 16 / 8$
	$- + 2 12 2$	$- 2 + 12 2$
	$- 14 2$	$14 - 2$
	12	

Evaluation of Postfix Expression

Begin

For each character in postfix expression, do

If operand is encounter, push it onto stack

Else if operator is encounter pop 2 elements

```

        A  $\rightarrow$  top element
        B  $\rightarrow$  Next to top element
        Result = B operator A
        Push result into stack
    Return element of stack top
End

```

Example

		Conversion
Infix Expression \rightarrow	$a + b * c - d / e ^ f$	
Postfix expression \rightarrow	$a b c * + d e f ^ / -$	
	$2 3 4 * + 16 2 3 ^ / -$	$2 3 * 4 + 16 2 3 ^ / -$
	$2 12 + 16 2 3 ^ / -$	$2 + 12 16 2 3 ^ / -$
	$14 16 2 3 ^ / -$	$14 16 2 ^ 3 / -$
	$14 16 8 / -$	$14 16 / 8 -$
	$14 2 -$	$14 - 2$
	12	

$2 3 1 * + 9 - \rightarrow - 4$

$53 + 62 / * 3 5 * + \rightarrow 39$

Prefix to Infix conversion using Stack

a b + e f / * → ((a + b) * (e/f))

Prefix to Infix

* + a b / e f → ((a+b) * (e/f))

Program to implement Stack using Array

```
// Implementation of Stack using Array -- Static Allocation
#include <stdio.h>
#include <conio.h>
#define N 5

int stack[N];
int top = -1;

void push()
{
    int data;
    printf("Enter the value : ");
    scanf("%d",&data);

    if(top == N-1)
    {
        printf("Stack overflow");
    }
    else
    {
        top++;
        stack[top] = data;
    }
}

void pop()
{
    int item;
    if(top == -1)
    {
        printf("Stack is empty");
    }
    else
    {
        item = stack[top];
        top--;
    }
    // printf("The popped item is :%d",item);
}

void peek()
{
    if(top == -1)
    {
        printf("Stack is empty");
    }
    else
    {
        printf("\n\nThe top most element in the stack is %d",stack[top]);
    }
}

void display()
{
    int i;

    printf("\nStack element : ");
    for(i=top;i>=0;i--)

```

```

    {
        printf("%d ",stack[i]);
    }
}

int main()
{
    int choice;
    // clrscr();

    do
    {
        printf("\n1 -> Push\n2 -> Pop\n3 -> Peek\n4 -> Display \n");
        scanf("%d",&choice);

        switch(choice)
        {
            case 1: push();
            break;
            case 2: pop();
            break;
            case 3: peek();
            break;
            case 4: display();
            break;
            default: printf("Invalid option");
            break;
        }
    }while(choice!= 0);

    getch();
    return 0;
}

```

Output

```

1 -> Push
2 -> Pop
3 -> Peek
4 -> Display
1
Enter the value : 2

```

```

1 -> Push
2 -> Pop
3 -> Peek
4 -> Display
1
Enter the value : 3

```

```

1 -> Push
2 -> Pop
3 -> Peek
4 -> Display
4

```

```

Stack element : 3 2
1 -> Push

```

```
2 -> Pop  
3 -> Peek  
4 -> Display  
3
```

The top most element in the stack is 3

```
1 -> Push  
2 -> Pop  
3 -> Peek  
4 -> Display  
2
```

```
1 -> Push  
2 -> Pop  
3 -> Peek  
4 -> Display  
4
```

Stack element : 2

```
1 -> Push  
2 -> Pop  
3 -> Peek  
4 -> Display
```

Program to implement Stack using Linked List

```
// Implementation of Stack using Linked List -- dynamic Allocation
#include <stdio.h>
#include <stdlib.h>
#include <conio.h>

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

struct node *top = NULL;

void push()
{
    struct node *newnode;
    int choice;

    while(choice)
    {
        newnode = (struct node *)malloc(sizeof(struct node));
        printf("Enter New Value : ");
        scanf("%d", &newnode->data);

        newnode->link = top;
        top = newnode;

        printf("Do you want to continue (0 -> No,1 -> Yes) : ");
        scanf("%d", &choice);
    }
}

void pop()
{
    struct node *temp;
    temp = top;

    if(top == NULL)
    {
        printf("Stack underflow");
    }
    else
    {
        printf("\nThe popped element is %d", top->data);
        top = top->link;
        free(temp);
    }
}

void display()
{
    struct node *temp;
    temp = top;

    if(top == NULL)
    {
        printf("Stack is empty");
    }
}
```

```

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

int main()
{
    int choice;

    do
    {
        printf("\n1 -> Push\n2 -> Pop\n3 -> Display \n");
        scanf("%d",&choice);

        switch(choice)
        {
            case 1: push();
                       break;
            case 2: pop();
                       break;
            case 3: display();
                       break;
            default: printf("Invalid option");
                       break;
        }
    }

    }while(choice!= 0);

    getch();
    return 0;
}

```

Output

```

1 -> Push
2 -> Pop
3 -> Display
1
Enter New Value : 100
Do you want to continue (0 -> No,1 -> Yes) : 1
Enter New Value : 200
Do you want to continue (0 -> No,1 -> Yes) : 1
Enter New Value : 400
Do you want to continue (0 -> No,1 -> Yes) : 0

1 -> Push
2 -> Pop
3 -> Display

3
400 200 100

```

```
1 -> Push  
2 -> Pop  
3 -> Display  
  
2  
The popped element is 400
```

```
1 -> Push  
2 -> Pop  
3 -> Display
```

```
3  
200 100
```

```
1 -> Push  
2 -> Pop  
3 -> Display
```

Watch Videos: -

[Stack Introduction](#)

[Stack Implementation using Array](#)

[Stack Implementation using Linked List](#)

[Infix Prefix and Postfix Expressions](#)

[Infix to Postfix conversion rules using Stack](#)

[Infix to Postfix Conversion using stack](#)

[Infix to Postfix conversion using Stack - Example](#)

[Infix to Prefix using stack](#)

[Evaluation of Prefix and Postfix expressions using stack](#)

[Postfix Expression evaluation using Stack](#)

[Prefix to Infix Conversion | Postfix to Infix Conversion](#)

[Expression trees | Binary Expression Tree](#)

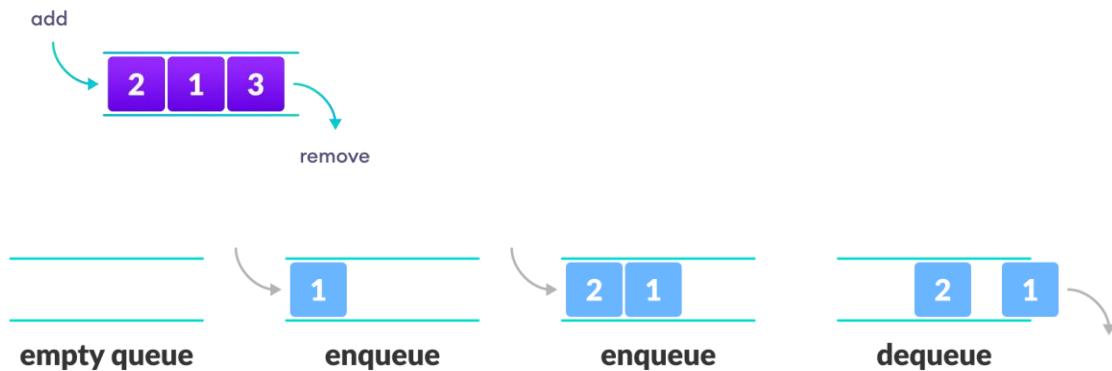
[Expression Tree from Postfix](#)

Queue

Unlike stack, the queue data structure works in the **FIFO** principle where **first element stored in the queue will be removed first**.

A queue is a useful data structure in programming. It is similar to the ticket queue outside a cinema hall, where the first person entering the queue is the first person who gets the ticket.

A queue is a **FIFO** (**First In First Out** — the element placed at first can be accessed at first) structure. This structure is named as “queue” because it resembles a real-world queue — people waiting in a queue.



In the above image, since 1 was kept in the queue before 2, it is the first to be removed from the queue as well. It follows the **FIFO** rule.

In programming terms, putting items in the queue is called **enqueue**, and removing items from the queue is called **dequeue**.

Basic Queue operations

A queue is an object (an abstract data structure - ADT) that allows the following operations:

- **Enqueue:** Add an element to the end of the queue
- **Dequeue:** Remove an element from the front of the queue
- **IsEmpty:** Check if the queue is empty
- **IsFull:** Check if the queue is full
- **Peek:** Get the value of the front of the queue without removing it

Applications of queues

- Used to manage threads in multithreading.
- Used to implement queuing systems (e.g.: priority queues).

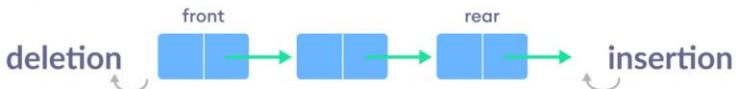
Types of Queues

There are four different types of queues:

- Simple Queue
- Circular Queue
- Priority Queue
- Double Ended Queue

Simple Queue

In a simple queue, insertion takes place at the rear and removal occurs at the front. It strictly follows the FIFO (First in First out) rule.



Working of Queue

Queue operations work as follows:

- two pointers **FRONT** and **REAR**
- **FRONT** track the first element of the queue
- **REAR** track the last element of the queue
- initially, set value of **FRONT** and **REAR** to -1

Enqueue Operation

- check if the queue is full
- for the first element, set the value of **FRONT** to 0
- increase the **REAR** index by 1
- add the new element in the position pointed to by **REAR**

Dequeue Operation

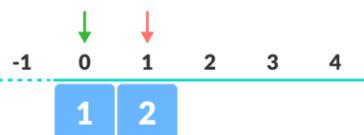
- check if the queue is empty
- return the value pointed by **FRONT**
- increase the **FRONT** index by 1
- for the last element, reset the values of **FRONT** and **REAR** to -1



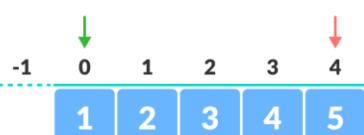
empty queue



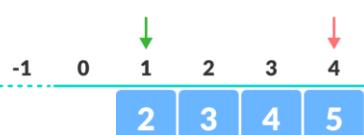
enqueue the first element



enqueue



enqueue



dequeue



dequeue the last element



empty queue

Program to implement Queue using Array

```
// Implementation of Queue using Array -- Static Allocation
#include <stdio.h>
#include <conio.h>

#define N 5
int queue[N];
int front = -1;
int rear = -1;

void enqueue()
{
    int item;
    printf("Enter the value : ");
    scanf("%d",&item);

    if (rear == N-1)
    {
        printf("Queue is overflow / Full");
    }
    else if(front == -1 && rear == -1)
    {
        front = rear = 0;
        queue[rear] = item;
    }
    else
    {
        rear++;
        queue[rear] = item;
    }
}

void dequeue()
{
    if(front == -1 && rear == -1)
    {
        printf("Queue is empty");
    }
    else if(front == rear)
    {
        front = rear = -1;
    }
    else
    {
        printf("\nDeleted item is : %d ",queue[front]);
        front++;
    }
}

void display()
{
    int i;
    if (front == -1 && rear == -1)
    {
        printf("\nQueue is empty");
    }
    else
    {
        printf("\nQueue values are : ");
    }
}
```

```

    for(i=front;i<rear+1;i++)
    {
        printf("%d ",queue[i]);
    }
}

void peek()
{
    if(front == rear)
    {
        printf("\nQueue is empty");
    }
    else
    {
        printf("\nThe Peek Value is %d ",queue[front]);
    }
}

int main()
{
    int choice;

    do
    {
        printf("\n1 -> Enqueue \n2 -> Dequeue\n3 -> Peek\n4 -> Display \n");
        scanf("%d",&choice);

        switch(choice)
        {
            case 1: enqueue();
                       break;
            case 2: dequeue();
                       break;
            case 3: peek();
                       break;
            case 4: display();
                       break;
            default: printf("Invalid option");
                       break;
        }
    }while(choice!= 0);

    getch();
    return 0;
}

```

Output

```

1 -> Enqueue
2 -> Dequeue
3 -> Peek
4 -> Display
1
Enter the value : 100

```

```
1 -> Enqueue
2 -> Dequeue
3 -> Peek
4 -> Display
1
Enter the value : 200

1 -> Enqueue
2 -> Dequeue
3 -> Peek
4 -> Display
1
Enter the value : 300

1 -> Enqueue
2 -> Dequeue
3 -> Peek
4 -> Display
4

Queue values are : 100 200 300
1 -> Enqueue
2 -> Dequeue
3 -> Peek
4 -> Display
2

Deleted item is : 100
1 -> Enqueue
2 -> Dequeue
3 -> Peek
4 -> Display
4

Queue values are : 200 300
1 -> Enqueue
2 -> Dequeue
3 -> Peek
4 -> Display
3

The Peek Value is 200
1 -> Enqueue
2 -> Dequeue
3 -> Peek
4 -> Display
```

Program to implement Queue using Linked List

```
// Implementation of Queue using Linked List -- Dynamic Allocation
#include <stdio.h>
#include <stdlib.h>
#include <conio.h>

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

struct node *front = NULL;
struct node *rear = NULL;

void enqueue()
{
    struct node *newnode;
    int choice = 1;

    while(choice)
    {
        newnode = (struct node *)malloc(sizeof(struct node));

        printf("Enter the data : ");
        scanf("%d", &newnode->data);
        newnode->next = NULL;

        if(front == NULL && rear ==NULL)
        {
            front = rear = newnode;
        }
        else
        {
            rear->next = newnode;
            rear = newnode;
        }

        printf("Do you want to continue (0 --> No,1 --> Yes) : ");
        scanf("%d", &choice);
    }
}

void dequeue()
{
    struct node *temp;
    temp = front;

    if(front == NULL && rear==NULL)
    {
        printf("Queue is empty");
    }
    else
    {
        printf("\nThe Deleted item is : %d ",front->data);
        front = front->next;
        free(temp);
    }
}
```

```

}

void display()
{
    struct node *temp;
    if(front ==NULL && rear == NULL)
    {
        printf("\nQueue is empty");
    }
    else
    {
        temp = front;
        while(temp !=NULL)
        {
            printf("%d ",temp->data);
            temp = temp->next;
        }
    }
}

void peek()
{
    if(front ==NULL && rear == NULL)
    {
        printf("\nQueue is empty");
    }
    else
    {
        printf("\nThe Peek Value is %d ",front->data);
    }
}

int main()
{
    int choice;

    do
    {
        printf("\n1 -> Enqueue \n2 -> Dequeue\n3 -> Peek\n4 -> Display \n");
        scanf("%d",&choice);

        switch(choice)
        {
            case 1: enqueue();
                      break;
            case 2: dequeue();
                      break;
            case 3: peek();
                      break;
            case 4: display();
                      break;
            default: printf("Invalid option");
                      break;
        }
    }while(choice!= 0);

    getch();
    return 0;
}

```

Output

```

1 -> Enqueue
2 -> Dequeue
3 -> Peek
4 -> Display
1
Enter the data : 100
Do you want to continue (0 --> No,1 --> Yes) : 1
Enter the data : 200
Do you want to continue (0 --> No,1 --> Yes) : 1
Enter the data : 300
Do you want to continue (0 --> No,1 --> Yes) : 0

1 -> Enqueue
2 -> Dequeue
3 -> Peek
4 -> Display
4
100 200 300
1 -> Enqueue
2 -> Dequeue
3 -> Peek
4 -> Display
2
The Deleted item is : 100
1 -> Enqueue
2 -> Dequeue
3 -> Peek
4 -> Display
4
200 300
1 -> Enqueue
2 -> Dequeue
3 -> Peek
4 -> Display
3
The Peek Value is 200
1 -> Enqueue
2 -> Dequeue
3 -> Peek
4 -> Display

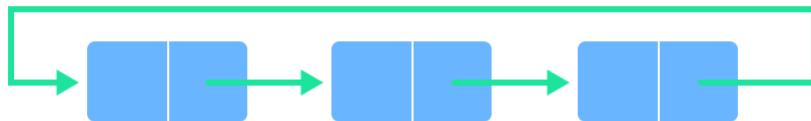
```

Watch Videos: -

[Queue in data structure - Introduction to queues](#)
[Queue implementation using Arrays](#)
[Queue implementation using Linked List](#)

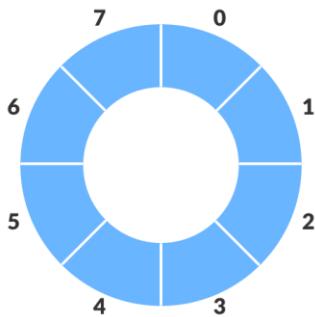
Circular Queue

In a circular queue, **the last element points to the first element making a circular link.**

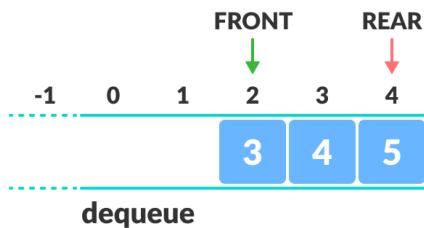


The main advantage of a circular queue over a simple queue is **better memory utilization**. If the last position is full and the first position is empty, we can insert an element in the first position. This action is not possible in a simple queue.

A circular queue is the extended version of a regular queue where **the last element is connected to the first element**. Thus forming a circle-like structure.



The circular queue solves the major limitation of the normal queue. In a normal queue, after a bit of insertion and deletion, there will be non-used empty space.



Here, indexes **0** and **1** can only be used after resetting the queue (deletion of all elements). This reduces the actual size of the queue.

How Circular Queue Works

Circular Queue works by the process of circular increment i.e. **when we try to increment the pointer and we reach the end of the queue, we start from the beginning of the queue.**

Here, the circular increment is performed by modulo division with the queue size. That is,

If $\text{REAR} + 1 == 5$ (overflow!), $\text{REAR} = (\text{REAR} + 1) \% 5 = 0$ (start of queue)

Circular Queue Operations

The circular queue work as follows:

- o two pointers **FRONT** and **REAR**
- o **FRONT** track the first element of the queue
- o **REAR** track the last elements of the queue
- o initially, set value of **FRONT** and **REAR** to -1

Enqueue Operation

- o check if the queue is full
- o for the first element, set value of **FRONT** to 0
- o circularly increase the **REAR** index by 1 (i.e. if the rear reaches the end, next it would be at the start of the queue)
- o add the new element in the position pointed to by **REAR**

Dequeue Operation

- o check if the queue is empty
- o return the value pointed by **FRONT**
- o circularly increase the **FRONT** index by 1
- o for the last element, reset the values of **FRONT** and **REAR** to -1

However, the check for full queue has a new additional case:

- o Case 1: **FRONT** = 0 && **REAR** == **SIZE** - 1
- o Case 2: **FRONT** = **REAR** + 1

The second case happens when **REAR** starts from 0 due to circular increment and when its value is just 1 less than **FRONT**, the queue is full.



empty queue



enqueue the first element



enqueue



enqueue



dequeue



enqueue



queue full

Program to implement Circular Queue using Array

```
// Circular Queue using Array -- Static Allocation
#include <stdio.h>
#include <conio.h>

#define N 5
int queue[N];
int front = -1;
int rear = -1;

void enqueue()
{
    int item;
    printf("Enter the value : ");
    scanf("%d",&item);

    if (front == -1 && rear == -1)
    {
        front = rear = 0;
        queue[rear] = item;
    }
    else if ((rear + 1) == front)
    {
        printf("Queue is full");
    }
    else
    {
        rear = (rear + 1)%N;
        queue[rear] = item;
    }
}

void dequeue()
{
    if(front == -1 && rear == -1)
    {
        printf("Queue is empty");
    }
    else if(front == rear)
    {
        printf("\nDeleted item is : %d ",queue[front]);
        front = rear = -1;
    }
    else
    {
        printf("\nDeleted item is : %d ",queue[front]);
        front = (front + 1)%N;
    }
}

void display()
{
    int i = front;

    if (front == -1 && rear == -1)
    {
        printf("\nQueue is empty");
    }
    else
```

```

{
    printf("\nQueue values are : ");
    while(i!=rear)
    {
        printf("%d ",queue[i]);
        i = (i + 1)%N;
    }
    printf("%d ",queue[rear]);
}

void peek()
{
    if (front == -1 && rear ==-1)
    {
        printf("\nQueue is empty");
    }
    else
    {
        printf("\nThe Peek Value is %d ",queue[front]);
    }
}

int main()
{
    int choice;

    do
    {
        printf("\n1 -> Enqueue \n2 -> Dequeue\n3 -> Peek\n4 -> Display \n");
        scanf("%d",&choice);

        switch(choice)
        {
            case 1: enqueue();
                      break;
            case 2: dequeue();
                      break;
            case 3: peek();
                      break;
            case 4: display();
                      break;
            default: printf("Invalid option");
                      break;
        }
    }while(choice!= 0);

    getch();
    return 0;
}

```

Output

```
1 -> Enqueue
2 -> Dequeue
3 -> Peek
4 -> Display
1
Enter the value : 100

1 -> Enqueue
2 -> Dequeue
3 -> Peek
4 -> Display
1
Enter the value : 200

1 -> Enqueue
2 -> Dequeue
3 -> Peek
4 -> Display
1
Enter the value : 300

1 -> Enqueue
2 -> Dequeue
3 -> Peek
4 -> Display
4

Queue values are : 100 200 300
1 -> Enqueue
2 -> Dequeue
3 -> Peek
4 -> Display
2

Deleted item is : 100
1 -> Enqueue
2 -> Dequeue
3 -> Peek
4 -> Display
4

Queue values are : 200 300
1 -> Enqueue
2 -> Dequeue
3 -> Peek
4 -> Display
3

The Peek Value is 200
1 -> Enqueue
2 -> Dequeue
3 -> Peek
4 -> Display
```

Program to implement Circular Queue using Linked List

```
// Circular Queue using Linked List -- Dynamic Allocation
#include <stdio.h>
#include <stdlib.h>
#include <conio.h>

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

struct node *front = NULL;
struct node *rear = NULL;

void enqueue()
{
    struct node *newnode;
    int choice = 1;

    while(choice)
    {
        newnode = (struct node *)malloc(sizeof(struct node));
        printf("Enter the data : ");
        scanf("%d", &newnode->data);
        newnode->next = NULL;

        if(rear == NULL)
        {
            front = rear = newnode;
        }
        else
        {
            rear->next = newnode;
            rear = newnode;
            rear->next = front;
        }

        printf("Do you want to continue (0 --> No,1 --> Yes) : ");
        scanf("%d", &choice);
    }
}

void dequeue()
{
    struct node *temp;
    temp = front;

    if(front == NULL && rear==NULL)
    {
        printf("Queue is empty");
    }
    else if (front==rear)
    {
        front = rear = NULL;
        free(temp);
    }
    else
}
```

```

    {
        front = front->next;
        rear->next = front;
        free(temp);
    }
}

void display()
{
    struct node *temp;
    temp = front;

    if(front ==NULL && rear == NULL)
    {
        printf("\nQueue is empty");
    }
    else
    {
        while(temp->next !=front)
        {
            printf("%d ",temp->data);
            temp = temp->next;
        }
        printf("%d",temp->data);
    }
}

void peek()
{
    if(front ==NULL && rear == NULL)
    {
        printf("\nQueue is empty");
    }
    else
    {
        printf("\nThe Peek Value is %d ",front->data);
    }
}

int main()
{
    int choice;

    do
    {
        printf("\n1 -> Enqueue \n2 -> Dequeue\n3 -> Peek\n4 -> Display \n");
        scanf("%d",&choice);

        switch(choice)
        {
            case 1: enqueue();
            break;
            case 2: dequeue();
            break;
            case 3: peek();
            break;
            case 4: display();
            break;
            default: printf("Invalid option");
            break;
        }
    }
}

```

```

    }

}while(choice!= 0);

getch();
return 0;
}
}

```

Output

```

1 -> Enqueue
2 -> Dequeue
3 -> Peek
4 -> Display
1
Enter the data : 100
Do you want to continue (0 --> No,1 --> Yes) : 1
Enter the data : 200
Do you want to continue (0 --> No,1 --> Yes) : 1
Enter the data : 300
Do you want to continue (0 --> No,1 --> Yes) : 0

1 -> Enqueue
2 -> Dequeue
3 -> Peek
4 -> Display
4
100 200 300
1 -> Enqueue
2 -> Dequeue
3 -> Peek
4 -> Display
2

1 -> Enqueue
2 -> Dequeue
3 -> Peek
4 -> Display
4
200 300
1 -> Enqueue
2 -> Dequeue
3 -> Peek
4 -> Display
3

The Peek Value is 200
1 -> Enqueue
2 -> Dequeue
3 -> Peek
4 -> Display

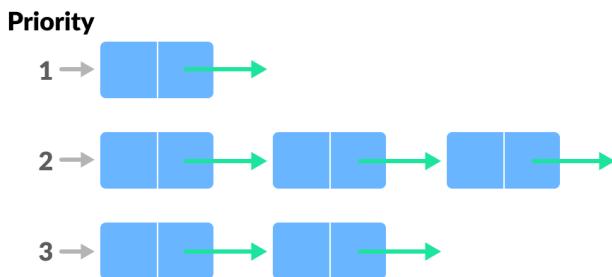
```

Watch Videos: -

[Circular Queue implementation using Array](#)
[Circular Queue implementation using Linked List](#)

Priority Queue

A priority queue is a special type of queue in which each element is associated with a priority and is served according to its priority. If elements with the same priority occur, they are served according to their order in the queue.



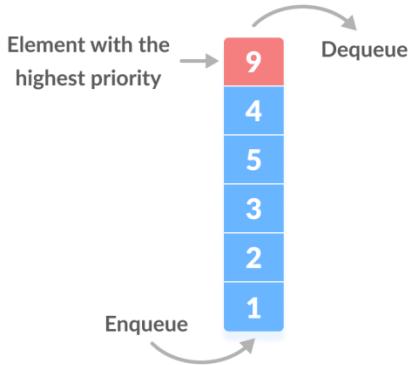
Insertion occurs based on the arrival of the values and removal occurs based on priority.

That is, higher priority elements are served first. However, if elements with the same priority occur, they are served according to their order in the queue.

Assigning Priority Value

Generally, the value of the element itself is considered for assigning the priority. For example, The element with the highest value is considered the highest priority element. However, in other cases, we can assume the element with the lowest value as the highest priority element.

We can also set priorities according to our needs.



Difference between Priority Queue and Normal Queue

In a queue, the first-in-first-out rule is implemented whereas, in a priority queue, the values are removed on the basis of priority. The element with the highest priority is removed first.

A comparative analysis of different implementations of priority queue is given below.

Operations	peek	insert	delete
Linked List	O(1)	O(n)	O(1)
Binary Heap	O(1)	O(log n)	O(log n)
Binary Search Tree	O(1)	O(log n)	O(log n)

Implementation of Priority Queue

Priority queue can be implemented using an array, a linked list, a heap data structure, or a binary search tree. Among these data structures, heap data structure provides an efficient implementation of priority queues.

Hence, we will be using the **heap data structure** to implement the priority queue. A max-heap is implemented in the following operations.

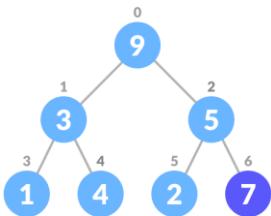
Priority Queue Operations

Basic operations of a priority queue are inserting, removing, and peeking elements.

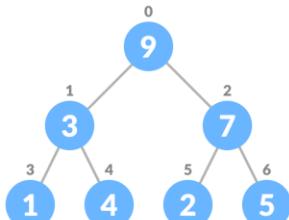
Inserting an Element into the Priority Queue

Inserting an element into a priority queue (max-heap) is done by the following steps.

- o Insert the new element at the end of the tree.



- o Heapify the tree



Algorithm for insertion of an element into priority queue (max-heap)

```

If there is no node,
  create a newNode.
else (a node is already present)
  insert the newNode at the end (last node from left to right.)
  
```

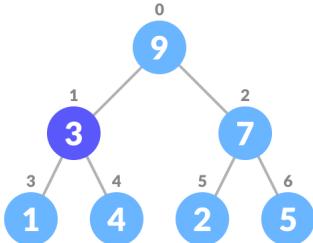
heapify the array

For Min Heap, the above algorithm is modified so that **parentNode** is always smaller than **newNode**.

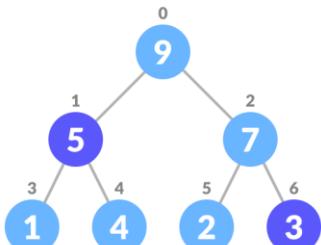
Deleting an Element from the Priority Queue

Deleting an element from a priority queue (max-heap) is done as follows:

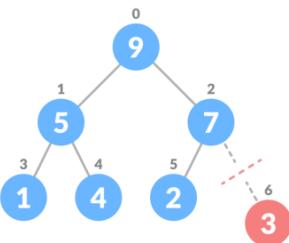
- o Select the element to be deleted.



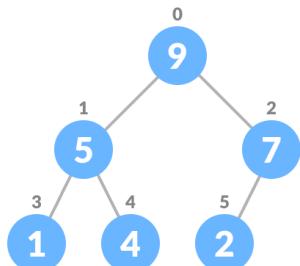
- o Swap it with the last element.



- o Remove the last element.



- o Heapify the tree.



Algorithm for deletion of an element in the priority queue (max-heap)

```
If nodeToBeDeleted is the leafNode
    remove the node
Else swap nodeToBeDeleted with the lastLeafNode
    remove noteToBeDeleted

heapify the array
```

For Min Heap, the above algorithm is modified so that the both **childNodes** are smaller than **currentNode**.

Peeking from the Priority Queue (Find max/min)

Peek operation returns the maximum element from Max Heap or minimum element from Min Heap without deleting the node.

For both Max heap and Min Heap

```
return rootNode
```

Extract-Max/Min from the Priority Queue

Extract-Max returns the node with maximum value after removing it from a Max Heap whereas Extract-Min returns the node with minimum value after removing it from Min Heap.

Program to implement Priority Queue using Heap data structure(max heap)

```
// Implementation of Priority Queue using Heap data structure (Max Heap)
#include <stdio.h>
#include <stdlib.h>
#include <conio.h>

#define N 5

struct heap{
    int size;
    int count;
    int *heaparr;
};

int *heap, size, count;
int initial_size = 4;

void Head_int(struct heap *h)
{
    h->count = 0;
    h->size = initial_size;
    h->heaparr = (int *)malloc(sizeof(int)*4);

    if(!h->heaparr){
        printf("Error while allocating the memory");
        exit(-1);
    }
}

void max_heapify(int *data, int loc,int count)
{
    int left,right, largest,temp;
    left = 2*(loc)+1;
    right = left + 1;
    largest = loc;

    if(left <=count && data[left] > data[largest]){
        largest = left;
    }

    if(right <= count && data[right] > data[largest]){
        largest = right;
    }

    if(largest != loc){
        temp = data[loc];
        data[loc] = data[largest];
        data[largest] = temp;
        max_heapify(data,largest,count);
    }
}

void heap_push(struct heap *h,int value)
{
    int index,parent;
    if(h->count == h->size){
        h->size +=1;
        h->heaparr = realloc(h->heaparr,sizeof(int)*h->size);
        if(!h->heaparr) exit(-1);
    }
}
```

```

}

index = h->count++;

for(;index;index = parent){
    parent = (index - 1)/2;
    if(h->heaparr[parent] >= value)break;
    h->heaparr[index] = h->heaparr[parent];
}
h->heaparr[index] = value;
}

int heap_delete(struct heap *h)
{
    int removed;
    int temp = h->heaparr[--h->count];

    if(h->count <= (h->size+2) && (h->size > initial_size)){
        h->size -= 1;
        h->heaparr = realloc(h->heaparr,sizeof(int)*h->size);
        if (!h->heaparr)exit(-1);
    }

    removed = h->heaparr[0];
    h->heaparr[0] = temp;
    max_heapify(h->heaparr,0,h->count);

    return removed;
}

int emptyPQ(struct heap *h)
{
    while(h->count != 0){
        printf("%d",heap_delete(h));
    }
}

void heap_display(struct heap *h)
{
    int i;
    for(i=0;i<h->count;++i){
        printf(" |%d| ",h->heaparr[i]);
    }
    printf("\n");
}

int main()
{
    struct heap h;
    Head_int(&h);

    heap_push(&h,1);
    heap_push(&h,5);
    heap_push(&h,3);
    heap_push(&h,7);
    heap_push(&h,9);
    heap_push(&h,8);

    heap_display(&h);
    emptyPQ(&h);
}

```

```
getch();  
return 0;  
}
```

Output

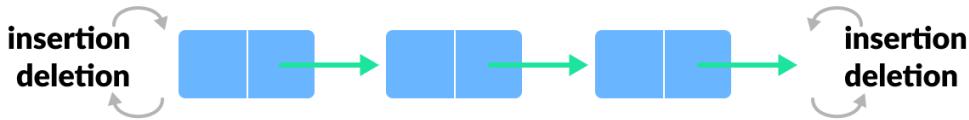
```
| 9 || 7 || 8 || 1 || 5 || 3 |  
987531
```

Watch Videos: -

[Priority Queue In Data Structure](#)

Deque (Double Ended Queue)

In a double ended queue, **insertion and removal of elements can be performed from either from the front or rear**. Thus, it does not follow the FIFO (First In First Out) rule.



Deque or Double Ended Queue is a type of queue in **which insertion and removal of elements can either be performed from the front or the rear**. Thus, it does not follow FIFO rule (First In First Out).



Types of Deque

- **Input Restricted Deque**
In this deque, **input is restricted at a single end but allows deletion at both the ends**.
- **Output Restricted Deque**
In this deque, **output is restricted at a single end but allows insertion at both the ends**.

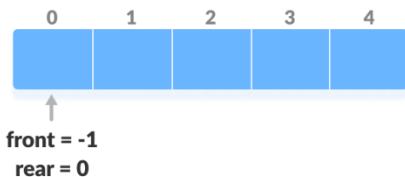
Operations on a Deque

Below is the circular array implementation of deque. **In a circular array, if the array is full, we start from the beginning**.

But in a linear array implementation, if the array is full, no more elements can be inserted. In each of the operations below, if the array is full, "overflow message" is thrown.

Before performing the following operations, these steps are followed.

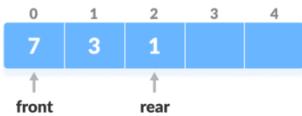
- Take an array (deque) of size **n**.
- Set two pointers at the first position and set **front = -1** and **rear = 0**.



Insert at the Front

This operation adds an element at the front.

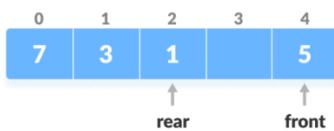
1. Check the position of front.



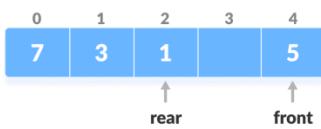
2. If $\text{front} < 1$, reinitialize $\text{front} = n-1$ (last index).



3. Else, decrease front by 1.



4. Add the new key 5 into $\text{array}[\text{front}]$. Insert the element at Front



Insert at the Rear

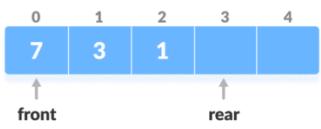
This operation adds an element to the rear.

1. Check if the array is full.



2. If the deque is full, reinitialize $\text{rear} = 0$.

3. Else, increase rear by 1.



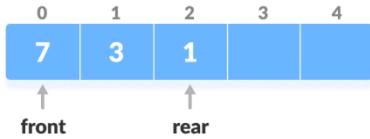
4. Add the new key 5 into $\text{array}[\text{rear}]$.



Delete from the Front

The operation deletes an element from the front.

1. Check if the deque is empty.



2. If the deque is empty (i.e. `front = -1`), deletion cannot be performed (**underflow condition**).
3. If the deque has only one element (i.e. `front = rear`), set `front = -1` and `rear = -1`.
4. Else if `front` is at the end (i.e. `front = n - 1`), set go to the front `front = 0`.
5. Else, `front = front + 1`.



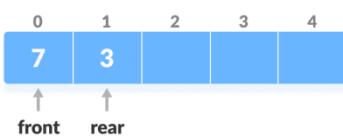
Delete from the Rear

This operation deletes an element from the rear.

1. Check if the deque is empty.



2. If the deque is empty (i.e. `front = -1`), deletion cannot be performed (**underflow condition**).
3. If the deque has only one element (i.e. `front = rear`), set `front = -1` and `rear = -1`, else follow the steps below.
4. If `rear` is at the front (i.e. `rear = 0`), set go to the front `rear = n - 1`.
5. Else, `rear = rear - 1`.



Check Empty

This operation checks if the deque is empty. If `front = -1`, the deque is empty.

Check Full

This operation checks if the deque is full. If `front = 0` and `rear = n - 1` OR `front = rear + 1`, the deque is full.

Program to implement Deque (Double-Ended Queue) Using Circular Array

```
// Implementation of Deque (Double-Ended Queue) Using Circular Array
#include <stdio.h>
#include <stdlib.h>
#include <conio.h>

#define N 5
int deque[N];
int front = -1;
int rear = -1;

void enqueueFront()
{
    int item;
    printf("Enter the value : ");
    scanf("%d",&item);

    if(front==0 && rear == N-1 || (front==rear+1)){
        printf("Queue is full");
    }
    else if(front == -1 && rear == -1){
        front = rear = 0;
        deque[front] = item;
    }
    else if(front == 0) {
        front = N-1;
        deque[front] = item;
    }
    else{
        front--;
        deque[front] = item;
    }
}

void enqueueRear()
{
    int item;
    printf("Enter the value : ");
    scanf("%d",&item);

    if(front==0 && rear == N-1 || (front==rear+1)){
        printf("Queue is full");
    }
    else if (front == -1 && rear == -1){
        front = rear = 0;
        deque[rear] = item;
    }
    else if (rear == N-1){
        rear = 0;
        deque[rear] = item;
    }
    else{
        rear++;
        deque[rear] = item;
    }
}
```

```

void display()
{
    int i = front;

    if(front == -1 && rear == -1) {
        printf("Queue is empty");
    }
    else{
        while (i != rear){
            printf("%d ",deque[i]);
            i = (i+1)%N;
        }
        printf("%d ",deque[rear]);
    }
}

void getFront()
{
    if(front == -1 && rear == -1) {
        printf("Queue is empty");
    }
    else{
        printf("%d ",deque[front]);
    }
}

void getRear()
{
    if(front == -1 && rear == -1) {
        printf("Queue is empty");
    }
    else{
        printf("%d ",deque[rear]);
    }
}

void dequeueFront(){
    if(front == -1 && rear == -1) {
        printf("Queue is empty");
    }
    else if(front == rear){
        front = rear = -1;
    }
    else if (front==N-1){
        printf("The dequeue element is %d",deque[front]);
        front = 0;
    }
    else{
        printf("The dequeue element is %d ",deque[front]);
        front++;
    }
}

void dequeueRear()
{
    if(front == -1 && rear == -1) {
        printf("Queue is empty");
    }
    else if(rear == front){
        front = rear = -1;
    }
}

```

```

    }
    else if(rear == 0){
        printf("The dequeue element is %d ",deque[rear]);
        rear = N-1;
    }
    else{
        printf("The dequeue element is %d ",deque[rear]);
        rear--;
    }
}

int main()
{
    int choice;

    do
    {
        printf("\n1 -> Enqueue Front\n2 -> Enqueue Rear\n3 -> Dequeue Front\n4 ->
Dequeue Rear\n5 -> Display \n");
        scanf("%d",&choice);

        switch(choice)
        {
            case 1: enqueueFront();
            break;
            case 2: enqueueRear();
            break;
            case 3: dequeueFront();
            break;
            case 4: dequeueRear();
            break;
            case 5: display();
            break;
            default: printf("Invalid option");
            break;
        }
    }

    }while(choice!= 0);

    getch();
    return 0;
}

```

Output

```

1 -> Enqueue Front
2 -> Enqueue Rear
3 -> Dequeue Front
4 -> Dequeue Rear
5 -> Display
1
Enter the value : 100
1 -> Enqueue Front
2 -> Enqueue Rear
3 -> Dequeue Front
4 -> Dequeue Rear
5 -> Display
2

```

```
Enter the value : 200
1 -> Enqueue Front
2 -> Enqueue Rear
3 -> Dequeue Front
4 -> Dequeue Rear
5 -> Display
5
100 200
1 -> Enqueue Front
2 -> Enqueue Rear
3 -> Dequeue Front
4 -> Dequeue Rear
5 -> Display
3
The dequeue element is 100
1 -> Enqueue Front
2 -> Enqueue Rear
3 -> Dequeue Front
4 -> Dequeue Rear
5 -> Display
```

Watch Videos: -

[Deque Introduction - Double Ended Queue](#)
[Implementation of Deque using Circular Array](#)

3. Data Structures (II)

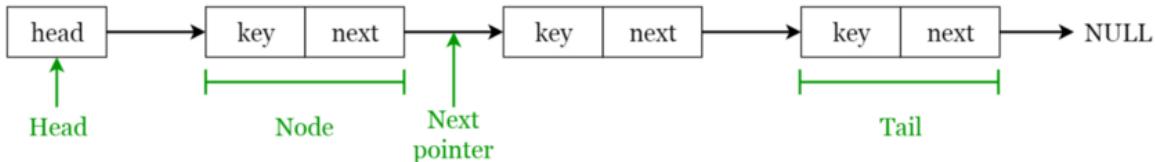
Linked List

In linked list data structure, **data elements are connected through a series of nodes**. And, each node contains the **data items and address to the next node** (Here, each node stores the **data** and the **address** of the next node).

A linked list is a linear / sequential structure that consists of a sequence of items in linear order which are linked to each other. Hence, you have to access data sequentially and random access is not possible. Linked lists provide a simple and flexible representation of dynamic sets.



- Elements in a linked list are known as **nodes**.
- Each node contains a **key** and a **pointer to its successor node**, known as **next**.
- The attribute named **head** points to the **first element of the linked list**.
- The **last element** of the linked list is known as the **tail**.



So we give the address of the first node a special name called **HEAD**. Also, the last node in the linked list can be identified because its next portion points to **NULL**.

Let's see how each node of the linked list is represented. Each node consists:

- A data item
- An address of another node

We wrap both the data item and the next node reference in a struct as:

```

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

Types of Linked List

Linked lists can be of multiple types: singly, doubly, and circular linked list. In this article, we will focus on the singly linked list.

- **Singly linked list** — Traversal of items can be done in the **forward direction only**.
- **Doubly linked list** — Traversal of items can be done in **both forward and backward directions**. Nodes consist of an additional pointer known as **prev**, pointing to the previous node.
- **Circular linked lists** — Linked lists where the **prev pointer of the head points to the tail and the next pointer of the tail points to the head**.

Singly Linked List

It is the most common. Each node has data and a pointer to the next node.



Node is represented as:

```

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

Singly Linked List Operations: Traverse, Insert and Delete

There are various linked list operations that allow us to perform different actions on linked lists.

- **Traversal** - access each element of the linked list
- **Insertion** - adds a new element to the linked list
- **Deletion** - removes the existing elements
- **Search** - find a node in the linked list
- **Sort** - sort the nodes of the linked list

Things to Remember about Linked List

- **head** points to the first node of the linked list
- **next** pointer of the last node is **NULL**, so if the next current node is **NULL**, we have reached the end of the linked list.

We will assume that the linked list has **three nodes 1 --->2 --->3** with node structure as below:

```

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

Traverse a Linked List

Displaying the contents of a linked list is very simple. We keep moving the temp node to the next one and display its contents.

When **temp** is **NULL**, we know that we have reached the end of the linked list so we get out of the while loop.

```

struct node *temp = head;
printf("\n\nList elements are - \n");
while(temp != NULL) {
    printf("%d --->", temp->data);
    temp = temp->next;
}
  
```

Insert Elements to a Linked List

You can add elements to the beginning, specified or end of the linked list.

1. Insert at the Beginning

- o Allocate memory for new node
- o Store data
- o Change next of new node to point to head
- o Change head to point to recently created node

```
struct node *newNode;
newNode = malloc(sizeof(struct node));
newNode->data = 4;
newNode->next = head;
head = newNode;
```

2. Insert at the End

- o Allocate memory for new node
- o Store data
- o Traverse to last node
- o Change next of last node to recently created node

```
struct node *newNode;
newNode = malloc(sizeof(struct node));
newNode->data = 4;
newNode->next = NULL;

struct node *temp = head;
while(temp->next != NULL) {
    temp = temp->next;
}

temp->next = newNode;
```

3. Insert at the Specified Position

- o Allocate memory and store data for new node
- o Traverse to node just before the required position of new node
- o Change next pointers to include new node in between

```
struct node *newNode;
newNode = malloc(sizeof(struct node));
newNode->data = 4;

struct node *temp = head;

for(int i=2; i < position; i++) {
    if(temp->next != NULL) {
        temp = temp->next;
    }
}
newNode->next = temp->next;
temp->next = newNode;
```

Delete from a Linked List

You can delete either from the beginning, end or from a particular position.

1. Delete from Beginning

Point head to the second node

```
head = head->next;
```

2. Delete from End

- Traverse to second last element
- Change its next pointer to null

```
struct node* temp = head;
while(temp->next->next!=NULL) {
    temp = temp->next;
}
temp->next = NULL;
```

3. Delete from specified position

- Traverse to element before the element to be deleted
- Change next pointers to exclude the node from the chain

```
for(int i=2; i< position; i++) {
    if(temp->next!=NULL) {
        temp = temp->next;
    }
}

temp->next = temp->next->next;
```

Search an Element on a Linked List

You can search an element on a linked list using a loop using the following steps. We are finding item on a linked list.

- Make head as the current node.
- Run a loop until the current node is NULL because the last element points to NULL.
- In each iteration, check if the key of the node is equal to `item`. If it the key matches the item, return true otherwise return false.

```
// Search a node
bool searchNode(struct Node** head_ref, int key) {
    struct Node* current = *head_ref;

    while (current != NULL) {
        if (current->data == key) return true;
        current = current->next;
    }
    return false;
}
```

Sort Elements of a Linked List

We will use a simple sorting algorithm, **Bubble Sort**, to sort the elements of a linked list in ascending order below.

- Make the **head** as the **current** node and create another node **index** for later use.
- If **head** is null, return.
- Else, run a loop till the last node (i.e. **NULL**).
- In each iteration, follow the following step 5-6.
- Store the next node of **current** in **index**.
- Check if the data of the current node is greater than the next node. If it is greater, swap **current** and **index**.

Program to implement Singly Linked List

```
// Linked List implementation in singly Linked List
#include <stdio.h>
#include <stdlib.h>
#include <conio.h>

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

    struct node *head, *newnode, *temp;
    head = NULL;
    int choice;

    while(choice)
    {
        // Creating New Node
        newnode = (struct node *)malloc(sizeof(struct node));
        printf("Enter Data : ");
        scanf("%d", &newnode->data);
        newnode->next = NULL;

        if(head == NULL)
        {
            head = temp = newnode;
        }
        else
        {
            temp->next = newnode;
            temp=newnode;
        }
        printf("Do you want to continue (0,1) : ");
        scanf("%d", &choice);
    }

    // Display Linked List -- Traversal
    temp = head;
    while(temp != NULL)
    {
        printf("%d ", temp->data);
        temp = temp->next;
    }
    getch();
    return 0;
}
```

Program to implement Singly Linked List - Insert at the Beginning

```
// Singly Linked List - Inserting node at the beginning
#include <stdio.h>
#include <stdlib.h>
#include <conio.h>
int main()
{
    struct node
    {
        int data;
        struct node *next;
    };
    struct node *head, *newnode, *temp;
    head = NULL;
    int choice;

    while(choice)
    {
        // Creating New Node
        newnode = (struct node *)malloc(sizeof(struct node));
        printf("Enter Data : ");
        scanf("%d", &newnode->data);
        newnode->next = NULL;

        if(head == NULL)
        {
            head = temp = newnode;
        }
        else
        {
            temp->next = newnode;
            temp=newnode;
        }
        printf("Do you want to continue (0,1) : ");
        scanf("%d", &choice);
    }

    // Dispaly Linked List
    temp = head;
    while(temp != NULL)
    {
        printf("%d ", temp->data);
        temp = temp->next;
    }

    // Inserting at the beginning
    newnode = (struct node *)malloc(sizeof(struct node));
    printf("\nEnter New Value : ");
    scanf("%d", &newnode->data);
    newnode->next = head;
    head = newnode;

    printf("New List is : \n");
    temp = head;
    while(temp != NULL)
    {
        printf("%d ", temp->data);
        temp = temp->next;
    }
}
```

```
    }  
  
    getch();  
    return 0;  
}
```

Output

```
Enter Data : 100  
Do you want to continue (0,1) : 1  
Enter Data : 200  
Do you want to continue (0,1) : 1  
Enter Data : 300  
Do you want to continue (0,1) : 0  
100 200 300  
Enter New Value : 50  
New List is :  
50 100 200 300
```

Program to implement Singly Linked List - Insert at the End

```
// Singly Linked List - Inserting node at the end
#include <stdio.h>
#include <stdlib.h>
#include <conio.h>
int main()
{
    struct node
    {
        int data;
        struct node *next;
    };
    struct node *head, *newnode, *temp;
    head = NULL;
    int choice;

    while(choice)
    {
        // Creating New Node
        newnode = (struct node *)malloc(sizeof(struct node));
        printf("Enter Data : ");
        scanf("%d", &newnode->data);
        newnode->next = NULL;

        if(head == NULL)
        {
            head = temp = newnode;
        }
        else
        {
            temp->next = newnode;
            temp=newnode;
        }
        printf("Do you want to continue (0,1) : ");
        scanf("%d", &choice);
    }

    // Display Linked List
    temp = head;
    while(temp != NULL)
    {
        printf("%d ", temp->data);
        temp = temp->next;
    }

    // Inserting at the end
    temp = head;
    while(temp->next != NULL)
    {
        temp = temp->next;
    }

    newnode = (struct node *)malloc(sizeof(struct node));
    printf("\nEnter New Value : ");
    scanf("%d", &newnode->data);
    newnode->next = NULL;
    temp->next = newnode;
```

```
printf("New List is : \n");
temp = head;
while(temp != NULL)
{
    printf("%d ",temp->data);
    temp = temp->next;
}

getch();
return 0;
}
```

Output

```
Do you want to continue (0,1) : 1
Enter Data : 200
Do you want to continue (0,1) : 0
100 200
Enter New Value : 300
New List is :
100 200 300
```

Program to implement Singly Linked List - Insert at the Specified Position

```
// Singly Linked List - Inserting node at the specified location
#include <stdio.h>
#include <stdlib.h>
#include <conio.h>
int main()
{
    struct node
    {
        int data;
        struct node *next;
    };
    struct node *head, *newnode, *temp;
    head = NULL;
    int choice, position, count=0;
    int i=1;

    while(choice)
    {
        // Creating New Node
        newnode = (struct node *)malloc(sizeof(struct node));
        printf("Enter Data : ");
        scanf("%d", &newnode->data);
        newnode->next = NULL;

        if(head == NULL)
        {
            head = temp = newnode;
        }
        else
        {
            temp->next = newnode;
            temp=newnode;
        }
        printf("Do you want to continue (0,1) : ");
        scanf("%d", &choice);
    }

    // Display Linked List
    temp = head;
    while(temp != NULL)
    {
        printf("%d ", temp->data);
        temp = temp->next;
        count++;
    }

    // Inserting at the specified location
    printf("\nEnter the Position : ");
    scanf("%d", &position);

    if(position > count)
    {
        printf("Invalid position\n");
    }
    else
    {
        temp = head;
```

```

while(i < position)
{
    temp = temp->next;
    i++;
}

newnode = (struct node *)malloc(sizeof(struct node));
printf("\nEnter New Value : ");
scanf("%d", &newnode->data);
newnode->next = temp->next;
temp->next = newnode;

}

printf("New List is : \n");
temp = head;
while(temp != NULL)
{
    printf("%d ", temp->data);
    temp = temp->next;
}

getch();
return 0;
}

```

Output

```

Enter Data : 100
Do you want to continue (0,1) : 1
Enter Data : 200
Do you want to continue (0,1) : 1
Enter Data : 300
Do you want to continue (0,1) : 1
Enter Data : 400
Do you want to continue (0,1) : 0
100 200 300 400
Enter the Position : 2

Enter New Value : 350
New List is :
100 200 350 300 400

```

Program to implement Singly Linked List - Delete at the Beginning

```
// Singly Linked List - Deleting node at the beginning
#include <stdio.h>
#include <stdlib.h>
#include <conio.h>
int main()
{
    struct node
    {
        int data;
        struct node *next;
    };
    struct node *head, *newnode, *temp;
    head = NULL;
    int choice;

    while(choice)
    {
        // Creating New Node
        newnode = (struct node *)malloc(sizeof(struct node));
        printf("Enter Data : ");
        scanf("%d", &newnode->data);
        newnode->next = NULL;

        if(head == NULL)
        {
            head = temp = newnode;
        }
        else
        {
            temp->next = newnode;
            temp=newnode;
        }
        printf("Do you want to continue (0 -> No,1 -> Yes) : ");
        scanf("%d", &choice);
    }

    // Display Linked List
    temp = head;
    while(temp != NULL)
    {
        printf("%d ", temp->data);
        temp = temp->next;
    }

    // Deleting at the beginning
    temp = head;
    head = head->next;
    free(temp);

    printf("\nNew List is : \n");
    temp = head;
    while(temp != NULL)
    {
        printf("%d ", temp->data);
        temp = temp->next;
    }
}
```

```
getch();  
    return 0;  
}
```

Output

```
Enter Data : 200  
Do you want to continue (0 -> No,1 -> Yes) : 1  
Enter Data : 300  
Do you want to continue (0 -> No,1 -> Yes) : 0  
100 200 300  
New List is :  
200 300
```

Program to implement Singly Linked List - Delete at the End

```
// Singly Linked List - Deleting node at the End
#include <stdio.h>
#include <stdlib.h>
#include <conio.h>
int main()
{
    struct node
    {
        int data;
        struct node *next;
    };
    struct node *head, *newnode, *temp, *previousnode;
    head = NULL;
    int choice;

    while(choice)
    {
        // Creating New Node
        newnode = (struct node *)malloc(sizeof(struct node));
        printf("Enter Data : ");
        scanf("%d", &newnode->data);
        newnode->next = NULL;

        if(head == NULL)
        {
            head = temp = newnode;
        }
        else
        {
            temp->next = newnode;
            temp=newnode;
        }
        printf("Do you want to continue (0 -> No,1 -> Yes) : ");
        scanf("%d", &choice);
    }

    // Display Linked List
    temp = head;
    while(temp != NULL)
    {
        printf("%d ", temp->data);
        temp = temp->next;
    }

    // Deleting at the end
    temp = head;
    while(temp->next != NULL)
    {
        previousnode = temp;
        temp = temp->next;
    }

    if(temp == head)
    {
        head = NULL;
    }
}
```

```
previousnode->next = NULL;
free(temp);

printf("\nNew List is : \n");
temp = head;
while(temp != NULL)
{
    printf("%d ", temp->data);
    temp = temp->next;
}

getch();
return 0;
}
```

Output

```
Enter Data : 100
Do you want to continue (0 -> No,1 -> Yes) : 1
Enter Data : 200
Do you want to continue (0 -> No,1 -> Yes) : 1
Enter Data : 300
Do you want to continue (0 -> No,1 -> Yes) : 1
Enter Data : 400
Do you want to continue (0 -> No,1 -> Yes) : 0
100 200 300 400
New List is :
100 200 300
```

Program to implement Singly Linked List - Delete at the Specified position

```
// Singly Linked List - Deleting node at the specified location
#include <stdio.h>
#include <stdlib.h>
#include <conio.h>
int main()
{
    struct node
    {
        int data;
        struct node *next;
    };
    struct node *head,*newnode,*temp,*nextnode;
    head = NULL;
    int choice,position,count=0;
    int i=1;

    while(choice)
    {
        // Creating New Node
        newnode = (struct node *)malloc(sizeof(struct node));
        printf("Enter Data : ");
        scanf("%d",&newnode->data);
        newnode->next = NULL;

        if(head == NULL)
        {
            head = temp = newnode;
        }
        else
        {
            temp->next = newnode;
            temp=newnode;
        }
        printf("Do you want to continue (0,1) : ");
        scanf("%d",&choice);
    }

    // Display Linked List
    temp = head;
    while(temp != NULL)
    {
        printf("%d ",temp->data);
        temp = temp->next;
        count++;
    }

    // Deleting at the specified location
    printf("\nEnter the Position : ");
    scanf("%d",&position);

    if(position > count)
    {
        printf("Invalid position\n");
    }
    else
    {
        temp = head;
```

```

while(i < position-1)
{
    temp = temp->next;
    i++;
}

nextnode = temp->next;
temp->next = nextnode->next;
free(nextnode);
}

printf("New List is : \n");
temp = head;
while(temp != NULL)
{
    printf("%d ",temp->data);
    temp = temp->next;
}

getch();
return 0;
}

```

Output

```

Enter Data : 100
Do you want to continue (0,1) : 1
Enter Data : 200
Do you want to continue (0,1) : 1
Enter Data : 300
Do you want to continue (0,1) : 1
Enter Data : 400
Do you want to continue (0,1) : 0
100 200 300 400
Enter the Position : 2
New List is :
100 300 400

```

Program to implement Singly Linked List - Reverse List

```
// Singly Linked List - Reverse node
#include <stdio.h>
#include <stdlib.h>
#include <conio.h>
int main()
{
    struct node
    {
        int data;
        struct node *next;
    };

    struct node *head, *newnode, *previousnode, *temp, *currentnode, *nextnode;
    head = NULL;
    int choice;

    while(choice)
    {
        // Creating New Node
        newnode = (struct node *)malloc(sizeof(struct node));
        printf("Enter Data : ");
        scanf("%d", &newnode->data);
        newnode->next = NULL;

        if(head == NULL)
        {
            head = temp = newnode;
        }
        else
        {
            temp->next = newnode;
            temp=newnode;
        }
        printf("Do you want to continue (0 --> No,1 --> Yes) : ");
        scanf("%d", &choice);
    }

    // Display Linked List
    temp = head;
    while(temp != NULL)
    {
        printf("%d ", temp->data);
        temp = temp->next;
    }

    // Reverse te list

    previousnode = NULL;
    currentnode = nextnode = head;

    while(nextnode!=NULL)
    {
        nextnode      = nextnode->next;
        currentnode->next = previousnode;
        previousnode = currentnode;
        currentnode = nextnode;
    }
}
```

```
head = previousnode;

printf("\nNew List is : \n");
temp = head;
while(temp != NULL)
{
    printf("%d ", temp->data);
    temp = temp->next;
}

getch();
return 0;
}
```

Output

```
Enter Data : 100
Do you want to continue (0 --> No,1 --> Yes) : 1
Enter Data : 200
Do you want to continue (0 --> No,1 --> Yes) : 1
Enter Data : 300
Do you want to continue (0 --> No,1 --> Yes) : 1
Enter Data : 400
Do you want to continue (0 --> No,1 --> Yes) : 0
100 200 300 400
New List is :
400 300 200 100
```

Program to implement Singly Linked List - Search

```
// Singly Linked List - Searching Singly Link List
#include <stdio.h>
#include <stdlib.h>
#include <conio.h>
int main()
{
    struct node
    {
        int data;
        struct node *next;
    };
    struct node *head, *newnode, *temp;
    head = NULL;
    int choice, search;

    while(choice)
    {
        // Creating New Node
        newnode = (struct node *)malloc(sizeof(struct node));
        printf("Enter Data : ");
        scanf("%d", &newnode->data);
        newnode->next = NULL;

        if(head == NULL)
        {
            head = temp = newnode;
        }
        else
        {
            temp->next = newnode;
            temp=newnode;
        }
        printf("Do you want to continue (0 --> No,1 --> Yes) : ");
        scanf("%d", &choice);
    }

    // Display Linked List
    temp = head;
    while(temp != NULL)
    {
        printf("%d ", temp->data);
        temp = temp->next;
    }

    // Search te list
    printf("\nEnter the data for search : ");
    scanf("%d", &search);

    temp = head;
    while(temp != NULL)
    {
        if(temp->data == search)
        {
            printf("Data found.");
            break;
        }
        temp = temp->next;
    }
}
```

```
    }  
  
    getch();  
    return 0;  
}
```

Output

```
Enter Data : 100  
Do you want to continue (0 --> No,1 --> Yes) : 1  
Enter Data : 200  
Do you want to continue (0 --> No,1 --> Yes) : 1  
Enter Data : 300  
Do you want to continue (0 --> No,1 --> Yes) : 1  
Enter Data : 400  
Do you want to continue (0 --> No,1 --> Yes) : 0  
100 200 300 400  
Enter the data for search : 300  
Data found.
```

Watch Videos: -

[Introduction to Linked List](#)
[Types of Linked List](#)
[Arrays vs. Linked List](#)

[Singly Linked List implementation](#)
[Insert a node in Singly Linked List \(at beginning, end, and specified position\)](#)
[Delete a node from Linked List](#)
[Find length of Linked List- Iterative approach](#)
[Reverse a Linked List - Iterative method](#)

Doubly Linked List

We add a pointer to the previous node in a doubly-linked list. Thus, we can go in either direction: forward or backward.



A node is represented as

```

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

A doubly linked list is a type of linked list in which each node consists of 3 components:

- o ***prev** - address of the previous node
- o **data** - data item
- o ***next** - address of next node



Representation of Doubly Linked List

Let's see how we can represent a doubly linked list on an algorithm/code. Suppose we have a doubly linked list:



```

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

Each struct node has a data item, a pointer to the previous struct node, and a pointer to the next struct node.

In the case of the head node, prev points to null, and in the case of the tail pointer, next points to null. Here, one is a head node and three is a tail node.

Insertion on a Doubly Linked List

Pushing a node to a doubly-linked list is similar to pushing a node to a linked list, but extra work is required to handle the pointer to the previous node.

Suppose we have a double-linked list with elements 1, 2, and 3.

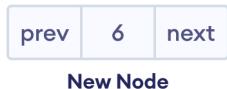


Insertion at the Beginning

Let's add a node with value 6 at the beginning of the doubly linked list we made above.

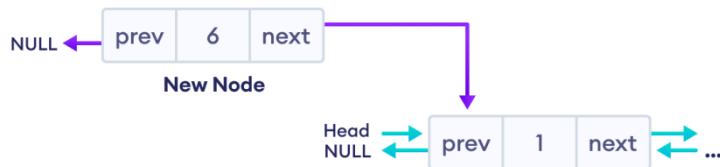
1. Create a new node

- allocate memory for `newNode`
- assign the data to `newNode`.



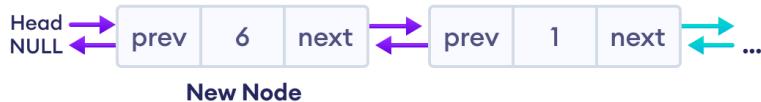
2. Set prev and next pointers of new node

- point `next` of `newNode` to the first node of the doubly linked list
- point `prev` to `null`



3. Make new node as head node

- Point `prev` of the first node to `newNode` (now the previous `head` is the second node)
- Point `head` to `newNode`



Insertion in between two nodes

Let's add a node with value 6 after node with value 1 in the doubly linked list.

1. Create a new node

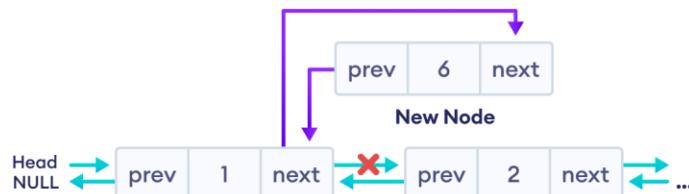
- o allocate memory for `newNode`
- o assign the data to `newNode`.



New Node

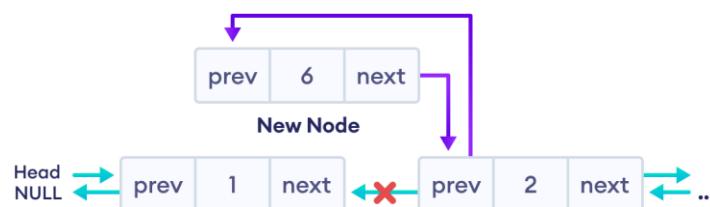
2. Set the next pointer of new node and previous node

- o assign the value of `next` from previous node to the `next` of `newNode`
- o assign the address of `newNode` to the `next` of previous node



3. Set the prev pointer of new node and the next node

- o assign the value of `prev` of next node to the `prev` of `newNode`
- o assign the address of `newNode` to the `prev` of next node



The final doubly linked list is after this insertion is:



Insertion at the End

Let's add a node with value 6 at the end of the doubly linked list.

1. Create a new node

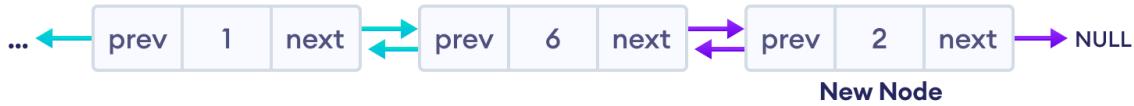


2. Set prev and next pointers of new node and the previous node

If the linked list is empty, make the **newNode** as the head node. Otherwise, traverse to the end of the doubly linked list and



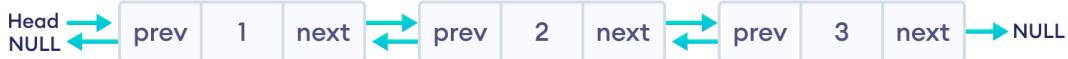
The final doubly linked list looks like this.



Deletion from a Doubly Linked List

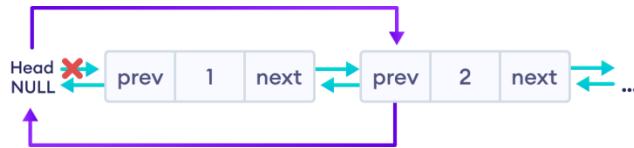
Similar to insertion, we can also delete a node from 3 different positions of a doubly linked list.

Suppose we have a double-linked list with elements 1, 2, and 3.

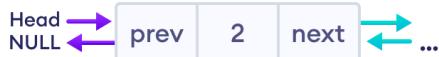


Delete the First Node of Doubly Linked List

If the node to be deleted (i.e. `del_node`) is at the beginning. Reset value node after the `del_node` (i.e. node two)



Finally, free the memory of `del_node`. And, the linked will look like this



Free the space of the first node

Deletion of the Inner Node

If `del_node` is an inner node (second node), we must have to reset the value of `next` and `prev` of the nodes before and after the `del_node`.

For the node before the `del_node` (i.e. first node)

Assign the value of `next` of `del_node` to the `next` of the `first` node.

For the node after the `del_node` (i.e. third node)

Assign the value of `prev` of `del_node` to the `prev` of the `third` node.



Finally, we will free the memory of `del_node`. And, the final doubly linked list looks like this.

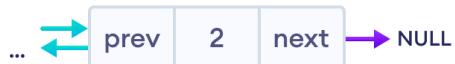


Delete the Last Node of Doubly Linked List

In this case, we are deleting the last node with value **3** of the doubly linked list. Here, we can simply delete the `del_node` and make the `next` of node before `del_node` point to `NULL`.



The final doubly linked list looks like this.



Program to implement Doubly Linked List

```
// Doubly Linked List Implementation
#include <stdio.h>
#include <stdlib.h>
#include <conio.h>
struct node
{
    int data;
    struct node *next;
    struct node *prev;
};
struct node *head,*tail;

// Creating Linked List Node
void createNode()
{
    struct node *newnode;
    int choice = 1;
    head = NULL;

    while(choice)
    {
        newnode = (struct node *)malloc(sizeof(struct node));
        printf("Enter the data : ");
        scanf("%d",&newnode->data);
        newnode->next = NULL;
        newnode->prev = NULL;

        if(head == NULL)
        {
            head = tail = newnode;
        }
        else
        {
            tail->next = newnode;
            newnode->prev = tail;
            tail = newnode;
        }
        printf("Do you want to continue (0 --> No,1 --. Yes) : ");
        scanf("%d",&choice);
    }
}

void displayNode()
{
    struct node *temp;
    temp = head;
    while(temp!=NULL)
    {
        printf("%d ",temp->data);
        temp = temp->next;
    }
};

int main()
{
    createNode();
    displayNode();
    getch();
}
```

```
    return 0;  
}
```

Output

```
Enter the data : 100  
Do you want to continue (0 --> No,1 --. Yes) : 1  
Enter the data : 200  
Do you want to continue (0 --> No,1 --. Yes) : 1  
Enter the data : 300  
Do you want to continue (0 --> No,1 --. Yes) : 1  
Enter the data : 400  
Do you want to continue (0 --> No,1 --. Yes) : 0  
100 200 300 400
```

Program to implement Doubly Linked List - Insert at the Beginning

```
// Doubly Linked List - Insert at the Beginning
#include <stdio.h>
#include <stdlib.h>
#include <conio.h>
struct node
{
    int data;
    struct node *next;
    struct node *prev;
}*head,*tail;

// Creating Linked List Node
void createNode()
{
    struct node *newnode;
    int choice = 1;
    head = NULL;

    while(choice)
    {
        newnode = (struct node *)malloc(sizeof(struct node));
        printf("Enter the data : ");
        scanf("%d",&newnode->data);
        newnode->next = NULL;
        newnode->prev = NULL;

        if(head == NULL)
        {
            head = tail = newnode;
        }
        else
        {
            tail->next = newnode;
            newnode->prev = tail;
            tail = newnode;
        }
        printf("Do you want to continue (0 --> No,1 --. Yes) : ");
        scanf("%d",&choice);
    }
}

void insertNodeAtBeginning()
{
    struct node *newnode;
    newnode = (struct node *)malloc(sizeof(struct node));
    printf("\nEnter the data for insert at beginning : ");
    scanf("%d",&newnode->data);
    newnode->next = NULL;
    newnode->prev = NULL;

    // node at the beginning
    head->prev = newnode;
    newnode->next = head;
    head = newnode;
}

void displayNode()
```

```

{
    struct node *temp;
    temp = head;
    printf("\nList is : \n");
    while(temp!=NULL)
    {
        printf("%d ",temp->data);
        temp = temp->next;
    }
};

int main()
{
    createNode();
    displayNode();

    insertNodeAtBeginning();
    displayNode();

    getch();
    return 0;
}

```

Output

```

Enter the data : 200
Do you want to continue (0 --> No,1 --. Yes) : 1
Enter the data : 300
Do you want to continue (0 --> No,1 --. Yes) : 1
Enter the data : 400
Do you want to continue (0 --> No,1 --. Yes) : 0

```

```

Initial List is :
200 300 400
Enter the data for insert at beginning : 100

```

```

Initial List is :
100 200 300 400

```

Program to implement Doubly Linked List - Insert at the End

```
//// Doubly Linked List - Insert at the End
#include <stdio.h>
#include <stdlib.h>
#include <conio.h>
struct node
{
    int data;
    struct node *next;
    struct node *prev;
}*head,*tail;

// Creating Linked List Node
void createNode()
{
    struct node *newnode;
    int choice = 1;
    head = NULL;

    while(choice)
    {
        newnode = (struct node *)malloc(sizeof(struct node));
        printf("Enter the data : ");
        scanf("%d",&newnode->data);
        newnode->next = NULL;
        newnode->prev = NULL;

        if(head == NULL)
        {
            head = tail = newnode;
        }
        else
        {
            tail->next = newnode;
            newnode->prev = tail;
            tail = newnode;
        }
        printf("Do you want to continue (0 --> No,1 --. Yes) : ");
        scanf("%d",&choice);
    }
}

void insertNodeAtEnd()
{
    struct node *newnode;
    newnode = (struct node *)malloc(sizeof(struct node));
    printf("\nEnter the data for insert at end : ");
    scanf("%d",&newnode->data);
    newnode->next = NULL;
    newnode->prev = NULL;

    // node at the End
    tail->next = newnode;
    newnode->prev = tail;
    tail = newnode;
}
```

```

void displayNode()
{
    struct node *temp;
    temp = head;
    printf("\nList is : \n");
    while(temp!=NULL)
    {
        printf("%d ",temp->data);
        temp = temp->next;
    }
};

int main()
{
    createNode();
    displayNode();

    insertNodeAtEnd();
    displayNode();

    getch();
    return 0;
}

```

Output

```

Enter the data : 100
Do you want to continue (0 --> No,1 --. Yes) : 1
Enter the data : 200
Do you want to continue (0 --> No,1 --. Yes) : 1
Enter the data : 300
Do you want to continue (0 --> No,1 --. Yes) : 0

```

```

List is :
100 200 300
Enter the data for insert at end : 400

```

```

List is :
100 200 300 400

```

Program to implement Doubly Linked List - Insert at the Specified Position

```
// Doubly Linked List - Insert at the Specified Position
#include <stdio.h>
#include <stdlib.h>
#include <conio.h>
struct node
{
    int data;
    struct node *next;
    struct node *prev;
}*head,*tail;

int count = 0;
// Creating Linked List Node
void createNode()
{
    struct node *newnode;
    int choice = 1;
    head = NULL;

    while(choice)
    {
        newnode = (struct node *)malloc(sizeof(struct node));
        printf("Enter the data : ");
        scanf("%d",&newnode->data);
        newnode->next = NULL;
        newnode->prev = NULL;

        if(head == NULL)
        {
            head = tail = newnode;
        }
        else
        {
            tail->next = newnode;
            newnode->prev = tail;
            tail = newnode;
        }
        printf("Do you want to continue (0 --> No,1 --. Yes) : ");
        scanf("%d",&choice);
    }
}

void insertNodeAtPosition()
{
    int position,i=1;
    printf("\nEnter the Position : ");
    scanf("%d",&position);

    if(count < position || position <= -1)
    {
        printf("Invalid position");
    }
    else
    {
        struct node *newnode,*temp;
        temp = head;
        newnode = (struct node *)malloc(sizeof(struct node));
        printf("Enter the data for insert at position : ");

```

```

scanf("%d", &newnode->data);
newnode->next = NULL;
newnode->prev = NULL;

while(i < position-1)
{
    temp = temp->next;
    i++;
}

newnode->prev = temp;
newnode->next = temp->next;
temp->next = newnode;
newnode->next->prev = newnode;
}

void displayNode()
{
    struct node *temp;
    temp = head;
    printf("\nList is : \n");
    while(temp!=NULL)
    {
        printf("%d ",temp->data);
        temp = temp->next;
        count++;
    }
};

int main()
{
    createNode();
    displayNode();

    insertNodeAtPosition();
    displayNode();

    getch();
    return 0;
}

```

Output

```

Enter the data : 200
Do you want to continue (0 --> No,1 --. Yes) : 1
Enter the data : 300
Do you want to continue (0 --> No,1 --. Yes) : 1
Enter the data : 400
Do you want to continue (0 --> No,1 --. Yes) : 0

```

```

List is :
200 300 400
Enter the Position : 3
Enter the data for insert at position : 350

```

```

List is :
200 300 350 400

```

Program to implement Doubly Linked List - Delete at the Beginning

```
// Doubly Linked List -- Delete at the Beginning
#include <stdio.h>
#include <stdlib.h>
#include <conio.h>
struct node
{
    int data;
    struct node *next;
    struct node *prev;
}*head,*tail;

int count = 0;
// Creating Linked List Node
void createNode()
{
    struct node *newnode;
    int choice = 1;
    head = NULL;

    while(choice)
    {
        newnode = (struct node *)malloc(sizeof(struct node));
        printf("Enter the data : ");
        scanf("%d",&newnode->data);
        newnode->next = NULL;
        newnode->prev = NULL;

        if(head == NULL)
        {
            head = tail = newnode;
        }
        else
        {
            tail->next = newnode;
            newnode->prev = tail;
            tail = newnode;
        }
        printf("Do you want to continue (0 --> No,1 --> Yes) : ");
        scanf("%d",&choice);
    }
}

void deleteNodeAtBeginning()
{
    struct node *temp;
    if(head == NULL)
    {
        printf("List is empty");
    }
    else
    {
        temp = head;
        head = head->next;
        head->prev = NULL;
        free(temp);
    }
}
```

```

void displayNode()
{
    struct node *temp;
    temp = head;
    printf("\nList is : \n");
    while(temp!=NULL)
    {
        printf("%d ",temp->data);
        temp = temp->next;
        count++;
    }
};

int main()
{
    createNode();
    displayNode();

    deleteNodeAtBeginning();
    displayNode();

    getch();
    return 0;
}

```

Output

```

Enter the data : 100
Do you want to continue (0 --> No,1 --> Yes) : 1
Enter the data : 200
Do you want to continue (0 --> No,1 --> Yes) : 1
Enter the data : 300
Do you want to continue (0 --> No,1 --> Yes) : 1
Enter the data : 400
Do you want to continue (0 --> No,1 --> Yes) : 0

List is :
100 200 300 400
List is :
200 300 400

```

Program to implement Doubly Linked List - Delete at the End

```
// Doubly Linked List -- Delete at the End
#include <stdio.h>
#include <stdlib.h>
#include <conio.h>
struct node
{
    int data;
    struct node *next;
    struct node *prev;
}*head,*tail;

int count = 0;
// Creating Linked List Node
void createNode()
{
    struct node *newnode;
    int choice = 1;
    head = NULL;

    while(choice)
    {
        newnode = (struct node *)malloc(sizeof(struct node));
        printf("Enter the data : ");
        scanf("%d",&newnode->data);
        newnode->next = NULL;
        newnode->prev = NULL;

        if(head == NULL)
        {
            head = tail = newnode;
        }
        else
        {
            tail->next = newnode;
            newnode->prev = tail;
            tail = newnode;
        }
        printf("Do you want to continue (0 --> No,1 --> Yes) : ");
        scanf("%d",&choice);
    }
}

void deleteNodeAtEnd()
{
    struct node *temp;
    if(tail == NULL)
    {
        printf("List is empty");
    }
    else
    {
        temp = tail;
        temp->prev->next = NULL;
        tail = tail->prev;
        free(temp);
    }
}
```

```

void displayNode()
{
    struct node *temp;
    temp = head;
    printf("\nList is : \n");
    while(temp!=NULL)
    {
        printf("%d ",temp->data);
        temp = temp->next;
        count++;
    }
};

int main()
{
    createNode();
    displayNode();

    deleteNodeAtEnd();
    displayNode();

    getch();
    return 0;
}

```

Output

```

Enter the data : 100
Do you want to continue (0 --> No,1 --> Yes) : 1
Enter the data : 200
Do you want to continue (0 --> No,1 --> Yes) : 1
Enter the data : 300
Do you want to continue (0 --> No,1 --> Yes) : 1
Enter the data : 400
Do you want to continue (0 --> No,1 --> Yes) : 0

```

```

List is :
100 200 300 400
List is :
100 200 300

```

Program to implement Doubly Linked List - Delete at the Specified Position

```
// Doubly Linked List -- Delete at the Specified Position
#include <stdio.h>
#include <stdlib.h>
#include <conio.h>
struct node
{
    int data;
    struct node *next;
    struct node *prev;
}*head,*tail;

int count = 0;
// Creating Linked List Node
void createNode()
{
    struct node *newnode;
    int choice = 1;
    head = NULL;

    while(choice)
    {
        newnode = (struct node *)malloc(sizeof(struct node));
        printf("Enter the data : ");
        scanf("%d",&newnode->data);
        newnode->next = NULL;
        newnode->prev = NULL;

        if(head == NULL)
        {
            head = tail = newnode;
        }
        else
        {
            tail->next = newnode;
            newnode->prev = tail;
            tail = newnode;
        }
        printf("Do you want to continue (0 --> No,1 --> Yes) : ");
        scanf("%d",&choice);
    }
}

void deleteNodeAtSpecifiedPosition()
{
    int position,i=1;
    struct node *temp;

    printf("\nEnter the position : ");
    scanf("%d",&position);

    temp = head;
    while(i < position)
    {
        temp=temp->next;
        i++;
    }

    temp->prev->next = temp->next;
}
```

```

temp->next->prev = temp->prev;
free(temp);
}

void displayNode()
{
    struct node *temp;
    temp = head;
    printf("\nList is : \n");
    while(temp!=NULL)
    {
        printf("%d ",temp->data);
        temp = temp->next;
        count++;
    }
};

int main()
{
    createNode();
    displayNode();

    deleteNodeAtSpecifiedPosition();
    displayNode();

    getch();
    return 0;
}

```

Output

```

Enter the data : 100
Do you want to continue (0 --> No,1 --> Yes) : 1
Enter the data : 200
Do you want to continue (0 --> No,1 --> Yes) : 1
Enter the data : 300
Do you want to continue (0 --> No,1 --> Yes) : 1
Enter the data : 400
Do you want to continue (0 --> No,1 --> Yes) : 0

```

```

List is :
100 200 300 400
Enter the position : 3

```

```

List is :
100 200 400

```

Watch Videos: -

[Introduction to Doubly Linked List](#)

[Implementation of Doubly Linked List](#)

[Insertion in Doubly Linked List \(beginning, end, specific position\)](#)

[Deletion from doubly linked list \(from beginning, end, specific position\)](#)

[Reverse a doubly linked list](#)

Circular Linked List

A circular linked list is a variation of a linked list in which the last element is linked to the first element. This forms a circular loop.

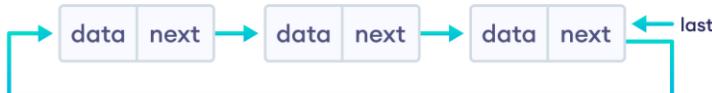


A circular linked list can be either singly linked or doubly linked.

- For singly linked list, **next pointer of last item points to the first item**
- In the doubly linked list, **prev pointer of the first item points to the last item as well.**

Circular Singly Linked List

Here, the address of the last node consists of the address of the first node.



Circular Doubly Linked List

Here, in addition to the last node storing the address of the first node, the first node will also store the address of the last node.



Representation of Circular Linked List

Let's see how we can represent a circular linked list on an algorithm/code. Suppose we have a linked list:



Here, the single node is represented as

```

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

Each struct node has a data item and a pointer to the next struct node.

Insertion on a Circular Linked List

Suppose we have a circular linked list with elements 1, 2, and 3.



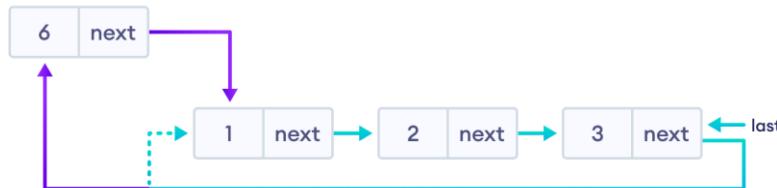
Let's add a node with value 6 at different positions of the circular linked list we made above. The first step is to create a new node.

- allocate memory for `newNode`
- assign the data to `newNode`



1. Insertion at the Beginning

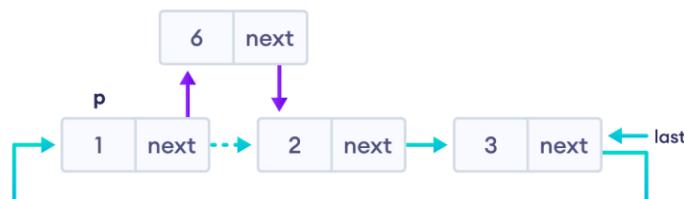
- store the address of the current first node in the `newNode` (i.e. pointing the `newNode` to the current first node)
- point the last node to `newNode` (i.e. making `newNode` as head)



2. Insertion in between two nodes

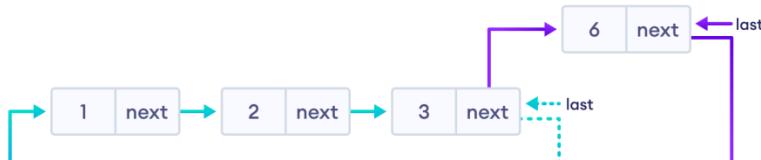
Let's insert `newNode` after the first node.

- travel to the node given (let this node be `p`)
- point the `next` of `newNode` to the node next to `p`
- store the address of `newNode` at `next` of `p`



3. Insertion at the End

- store the address of the head node to `next` of `newNode` (making `newNode` the last node)
- point the current last node to `newNode`
- make `newNode` as the last node



Deletion on a Circular Linked List

Suppose we have a double-linked list with elements 1, 2, and 3.

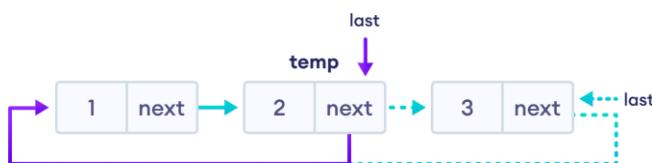


2. If the node to be deleted is the only node

- free the memory occupied by the node
- store `NULL` in `last`

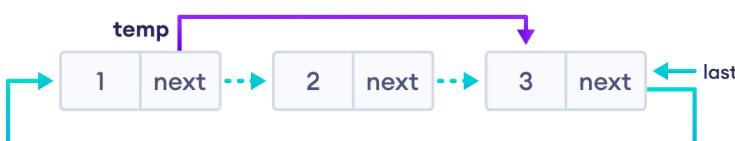
3. If last node is to be deleted

- find the node before the last node (let it be `temp`)
- store the address of the node next to the last node in `temp`
- free the memory of last
- make `temp` as the last node



4. If any other nodes are to be deleted

- travel to the node to be deleted (here we are deleting node 2)
- let the node before node 2 be `temp`
- store the address of the node next to 2 in `temp`
- free the memory of 2



Program to implement Circular Linked List

```

// Circular Linked List -- Implementation Head and Tail
#include <stdio.h>
#include <stdlib.h>
#include <conio.h>
struct node
{
    int data;
    struct node *next;
}*head,*tail;

// Creating Circular Linked List Node
void createCircularListNode()
{
    struct node *newnode;
    int choice = 1;
    head = NULL;

    while(choice)
    {
        newnode = (struct node *)malloc(sizeof(struct node));
        printf("Enter the data : ");
        scanf("%d",&newnode->data);
        newnode->next = NULL;

        if(head == NULL)
        {
            head = tail = newnode;
        }
        else
        {
            tail->next = newnode;
            tail = newnode;
        }

        tail->next = head;

        printf("Do you want to continue (0 --> No,1 --> Yes) : ");
        scanf("%d",&choice);
    }
    /* to check the list is Circular,
    enable below code and see the value is first node*/
    // printf("\n%d",tail->next->data);
}

void displayNode()
{
    struct node *temp;

    if (head == NULL)
    {
        printf("List is empty");
    }
    else
    {
        temp = head;
        printf("\nList is : \n");
        while(temp->next != head)
        {

```

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

int main()
{
    createCircularListNode();
    displayNode();

    getch();
    return 0;
}
```

Output

```
Enter the data : 100
Do you want to continue (0 --> No,1 --> Yes) : 1
Enter the data : 200
Do you want to continue (0 --> No,1 --> Yes) : 1
Enter the data : 300
Do you want to continue (0 --> No,1 --> Yes) : 1
Enter the data : 400
Do you want to continue (0 --> No,1 --> Yes) : 0

List is :
100 200 300 400
```

Program to implement Circular Linked List - Insert at the Beginning

```
// Circular Linked List -- Tail -- Insert at the Beginning
#include <stdio.h>
#include <stdlib.h>
#include <conio.h>

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

// Creating Circular Linked List Node
void createCircularListNode()
{
    struct node *newnode;
    int choice = 1;
    tail = NULL;

    while(choice)
    {
        newnode = (struct node *)malloc(sizeof(struct node));
        printf("Enter the data : ");
        scanf("%d", &newnode->data);
        newnode->next = NULL;

        if(tail == NULL)
        {
            tail = newnode;
            tail->next = newnode;
        }
        else
        {
            newnode->next = tail->next;
            tail->next = newnode;
            tail = newnode;
        }

        printf("Do you want to continue (0 --> No,1 --> Yes) : ");
        scanf("%d", &choice);
    }
    /* to check the list is Circular,
    enable below code and see the value is first node*/
    // printf("\n%d",tail->next->data);
}

void InsertNodeAtBeginning()
{
    struct node *newnode;

    newnode = (struct node *)malloc(sizeof(struct node));
    printf("\nEnter the new node : ");
    scanf("%d", &newnode->data);
    newnode->next = NULL;

    if (tail == NULL)
    {
        tail = newnode;
        tail->next = newnode;
    }
}
```

```

    }
else
{
    newnode->next = tail->next;
    tail->next = newnode;
}
}

void displayNode()
{
    struct node *temp;

    if (tail == NULL)
    {
        printf("List is empty");
    }
else
{
    temp = tail->next;
    printf("\nList is : \n");
    while(temp->next != tail->next)
    {
        printf("%d ",temp->data);
        temp = temp->next;
    }
    printf("%d",temp->data);
}
};

int main()
{
    createCircularListNode();
    displayNode();

    InsertNodeAtBeginning();
    displayNode();

    getch();
    return 0;
}

```

Output

```

Enter the data : 100
Do you want to continue (0 --> No,1 --> Yes) : 1
Enter the data : 200
Do you want to continue (0 --> No,1 --> Yes) : 1
Enter the data : 300
Do you want to continue (0 --> No,1 --> Yes) : 0

```

```

List is :
100 200 300
Enter the new node : 50

```

```

List is :
50 100 200 300

```

Program to implement Circular Linked List - Insert at the End

```
// Circular Linked List -- Tail -- Insert at the End
#include <stdio.h>
#include <stdlib.h>
#include <conio.h>

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

// Creating Circular Linked List Node
void createCircularListNode()
{
    struct node *newnode;
    int choice = 1;
    tail = NULL;

    while(choice)
    {
        newnode = (struct node *)malloc(sizeof(struct node));
        printf("Enter the data : ");
        scanf("%d", &newnode->data);
        newnode->next = NULL;

        if(tail == NULL)
        {
            tail = newnode;
            tail->next = newnode;
        }
        else
        {
            newnode->next = tail->next;
            tail->next = newnode;
            tail = newnode;
        }

        printf("Do you want to continue (0 --> No,1 --> Yes) : ");
        scanf("%d", &choice);
    }
    /* to check the list is Circular,
    enable below code and see the value is first node*/
    // printf("\n%d",tail->next->data);
}

void InsertNodeAtEnd()
{
    struct node *newnode;

    newnode = (struct node *)malloc(sizeof(struct node));
    printf("\nEnter the new node : ");
    scanf("%d", &newnode->data);
    newnode->next = NULL;

    if (tail == NULL)
    {
        tail = newnode;
        tail->next = newnode;
    }
}
```

```

    }
else
{
    newnode->next = tail->next;
    tail->next = newnode;
    tail = newnode;
}
}

void displayNode()
{
    struct node *temp;

    if (tail == NULL)
    {
        printf("List is empty");
    }
else
{
    temp = tail->next;
    printf("\nList is : \n");
    while(temp->next != tail->next)
    {
        printf("%d ",temp->data);
        temp = temp->next;
    }
    printf("%d",temp->data);
}
};

int main()
{
    createCircularListNode();
    displayNode();

    InsertNodeAtEnd();
    displayNode();

    getch();
    return 0;
}

```

Output

```

Enter the data : 100
Do you want to continue (0 --> No,1 --> Yes) : 1
Enter the data : 200
Do you want to continue (0 --> No,1 --> Yes) : 1
Enter the data : 300
Do you want to continue (0 --> No,1 --> Yes) : 0

List is :
100 200 300
Enter the new node : 400

List is :
100 200 300 400

```

Circular Linked List - Insert at the Specified Position

```

// Circular Linked List -- Tail -- Insert at the Specified Position
#include <stdio.h>
#include <stdlib.h>
#include <conio.h>

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

// Creating Circular Linked List Node
void createCircularListNode()
{
    struct node *newnode;
    int choice = 1;
    tail = NULL;

    while(choice)
    {
        newnode = (struct node *)malloc(sizeof(struct node));
        printf("Enter the data : ");
        scanf("%d", &newnode->data);
        newnode->next = NULL;

        if(tail == NULL)
        {
            tail      = newnode;
            tail->next = newnode;
        }
        else
        {
            newnode->next = tail->next;
            tail->next      = newnode;
            tail      = newnode;
        }
        printf("Do you want to continue (0 --> No,1 --> Yes) : ");
        scanf("%d", &choice);
    }
    /* to check the list is Circular,
    enable below code and see the value is first node*/
    // printf("\n%d",tail->next->data);
}

void InsertNodeAtSpecifiedPosition()
{
    struct node *newnode,*temp;
    int position, i = 1;

    printf("\nEnter the position : ");
    scanf("%d",&position);
    /*
        Add position validation
        position < 0 || position > length of the list
    */
    // if position == 1 , then use insertAtBeginning()

    newnode = (struct node *)malloc(sizeof(struct node));
}

```

```

printf("\nEnter the new node : ");
scanf("%d", &newnode->data);
newnode->next = NULL;
temp = tail->next;

while ( i < position - 1)
{
    temp = temp->next;
    i++;
}
newnode->next = temp->next;
temp->next      = newnode;
}

void displayNode()
{
    struct node *temp;

    if (tail == NULL)
    {
        printf("List is empty");
    }
    else
    {
        temp = tail->next;
        printf("\nList is : \n");
        while(temp->next != tail->next)
        {
            printf("%d ", temp->data);
            temp = temp->next;
        }
        printf("%d", temp->data);
    }
};

int main()
{
    createCircularListNode();
    displayNode();
    InsertNodeAtSpecifiedPosition();
    displayNode();
    getch();
    return 0;
}

```

Output

```

Enter the data : 100
Do you want to continue (0 --> No,1 --> Yes) : 1
Enter the data : 200
Do you want to continue (0 --> No,1 --> Yes) : 1
Enter the data : 300
Do you want to continue (0 --> No,1 --> Yes) : 0

List is :
100 200 300
Enter the position : 3
Enter the new node : 250
List is :
100 200 250 300

```

Program to implement Circular Linked List - Delete at the Beginning

```

// Circular Linked List -- Tail -- Delete at the Beginning
#include <stdio.h>
#include <stdlib.h>
#include <conio.h>

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

// Creating Circular Linked List Node
void createCircularListNode()
{
    struct node *newnode;
    int choice = 1;
    tail = NULL;

    while(choice)
    {
        newnode = (struct node *)malloc(sizeof(struct node));
        printf("Enter the data : ");
        scanf("%d", &newnode->data);
        newnode->next = NULL;

        if(tail == NULL)
        {
            tail      = newnode;
            tail->next = newnode;
        }
        else
        {
            newnode->next = tail->next;
            tail->next      = newnode;
            tail      = newnode;
        }

        printf("Do you want to continue (0 --> No,1 --> Yes) : ");
        scanf("%d", &choice);
    }
    /* to check the list is Circular,
    enable below code and see the value is first node*/
    // printf("\n%d",tail->next->data);
}

void deleteNodeAtBeginning()
{
    struct node *temp;
    temp = tail->next;

    if(tail == NULL)
    {
        printf("List is empty");
    }
    //List contain only one node
    else if(temp->next == temp)
    {
        tail = NULL;
    }
}

```

```

        free(tail);
    }
else
{
    tail->next = temp->next;
    free(temp);
}
}

void displayNode()
{
    struct node *temp;

    if (tail == NULL)
    {
        printf("List is empty");
    }
else
{
    temp = tail->next;
    printf("\nList is : \n");
    while(temp->next != tail->next)
    {
        printf("%d ",temp->data);
        temp = temp->next;
    }
    printf("%d",temp->data);
}
};

int main()
{
    createCircularListNode();
    displayNode();

    deleteNodeAtBeginning();
    displayNode();

    getch();
    return 0;
}
}

```

Output

```

Enter the data : 100
Do you want to continue (0 --> No,1 --> Yes) : 1
Enter the data : 200
Do you want to continue (0 --> No,1 --> Yes) : 1
Enter the data : 300
Do you want to continue (0 --> No,1 --> Yes) : 0

List is :
100 200 300
List is :
200 300

```

Program to implement Circular Linked List - Delete at the End

```

// Circular Linked List -- Tail -- Delete at the End
#include <stdio.h>
#include <stdlib.h>
#include <conio.h>

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

// Creating Circular Linked List Node
void createCircularListNode()
{
    struct node *newnode;
    int choice = 1;
    tail = NULL;

    while(choice)
    {
        newnode = (struct node *)malloc(sizeof(struct node));
        printf("Enter the data : ");
        scanf("%d", &newnode->data);
        newnode->next = NULL;

        if(tail == NULL)
        {
            tail = newnode;
            tail->next = newnode;
        }
        else
        {
            newnode->next = tail->next;
            tail->next = newnode;
            tail = newnode;
        }

        printf("Do you want to continue (0 --> No,1 --> Yes) : ");
        scanf("%d", &choice);
    }
    /* to check the list is Circular,
    enable below code and see the value is first node*/
    // printf("\n%d",tail->next->data);
}

void deleteNodeAtEnd()
{
    struct node *currentnode,*previousnode;
    currentnode = tail->next;
    if(tail == NULL)
    {
        printf("List is empty");
    }
    //List contain only one node
    else if(currentnode->next == currentnode)
    {
        tail = NULL;
        free(currentnode);
    }
}

```

```

    }
else
{
    while(currentnode->next != tail->next)
    {
        previousnode = currentnode;
        currentnode = currentnode->next;
    }

    previousnode->next = tail->next;
    tail = previousnode;
    free(currentnode);
}

void displayNode()
{
    struct node *temp;
    if (tail == NULL)
    {
        printf("List is empty");
    }
else
{
    temp = tail->next;
    printf("\nList is : \n");
    while(temp->next != tail->next)
    {
        printf("%d ",temp->data);
        temp = temp->next;
    }
    printf("%d",temp->data);
}
};

int main()
{
    createCircularListNode();
    displayNode();
    deleteNodeAtEnd();
    displayNode();
    getch();
    return 0;
}

```

Output

```

Enter the data : 100
Do you want to continue (0 --> No,1 --> Yes) : 1
Enter the data : 200
Do you want to continue (0 --> No,1 --> Yes) : 1
Enter the data : 300
Do you want to continue (0 --> No,1 --> Yes) : 0

```

```

List is :
100 200 300
List is :
100 200

```

Program to implement Circular Linked List - Delete at the Specified Position

```

// Circular Linked List -- Tail -- Delete at the Specified Position
#include <stdio.h>
#include <stdlib.h>
#include <conio.h>

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

// Creating Circular Linked List Node
void createCircularListNode()
{
    struct node *newnode;
    int choice = 1;
    tail = NULL;

    while(choice)
    {
        newnode = (struct node *)malloc(sizeof(struct node));
        printf("Enter the data : ");
        scanf("%d", &newnode->data);
        newnode->next = NULL;

        if(tail == NULL)
        {
            tail      = newnode;
            tail->next = newnode;
        }
        else
        {
            newnode->next = tail->next;
            tail->next      = newnode;
            tail      = newnode;
        }

        printf("Do you want to continue (0 --> No,1 --> Yes) : ");
        scanf("%d", &choice);
    }
    /* to check the list is Circular,
    enable below code and see the value is first node*/
    // printf("\n%d",tail->next->data);
}

void deleteNodeAtSpecifiedPosition()
{
    struct node *currentnode,*nextnode;
    int position, i = 1;

    currentnode = tail->next;

    printf("\nEnter the position : ");
    scanf("%d", &position);
    //validation [if position < 1 || position > lenth of the list]
    // If position == 1 , then call delete at the beginning.
    while(i < position -1)
    {
}

```

```

        currentnode = currentnode->next;
        i++;
    }

    nextnode = currentnode->next;
    currentnode->next = nextnode->next;
    free(nextnode);
}

void displayNode()
{
    struct node *temp;

    if (tail == NULL)
    {
        printf("List is empty");
    }
    else
    {
        temp = tail->next;
        printf("\nList is : \n");
        while(temp->next != tail->next)
        {
            printf("%d ",temp->data);
            temp = temp->next;
        }
        printf("%d",temp->data);
    }
};

int main()
{
    createCircularListNode();
    displayNode();

    deleteNodeAtSpecifiedPosition();
    displayNode();

    getch();
    return 0;
}

```

Output

```

Enter the data : 100
Do you want to continue (0 --> No,1 --> Yes) : 1
Enter the data : 200
Do you want to continue (0 --> No,1 --> Yes) : 1
Enter the data : 300
Do you want to continue (0 --> No,1 --> Yes) : 0

List is :
100 200 300
Enter the position : 2
List is :
100 300

```

Program to implement Circular Linked List - Reverse

```

// Circular Linked List -- Tail -- Reverse
#include <stdio.h>
#include <stdlib.h>
#include <conio.h>

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

// Creating Circular Linked List Node
void createCircularListNode()
{
    struct node *newnode;
    int choice = 1;
    tail = NULL;

    while(choice)
    {
        newnode = (struct node *)malloc(sizeof(struct node));
        printf("Enter the data : ");
        scanf("%d", &newnode->data);
        newnode->next = NULL;

        if(tail == NULL)
        {
            tail      = newnode;
            tail->next = newnode;
        }
        else
        {
            newnode->next = tail->next;
            tail->next      = newnode;
            tail      = newnode;
        }

        printf("Do you want to continue (0 --> No,1 --> Yes) : ");
        scanf("%d", &choice);
    }
    /* to check the list is Circular,
    enable below code and see the value is first node*/
    // printf("\n%d",tail->next->data);
}

void reverseList()
{
    struct node *currentnode,*previousnode,*nextnode;
    currentnode = tail->next;
    nextnode    = currentnode->next;

    if(tail == NULL)
    {
        printf("List is empty ");
    }
    else
    {
        while(currentnode != tail)
    }
}

```

```

    {
        previousnode = currentnode;
        currentnode = nextnode;
        nextnode     = currentnode->next;
        currentnode->next = previousnode;
    }
    nextnode->next = tail;
    tail = nextnode;
}
}

void displayNode()
{
    struct node *temp;

    if (tail == NULL)
    {
        printf("List is empty");
    }
    else
    {
        temp = tail->next;
        printf("\nList is : \n");
        while(temp->next != tail->next)
        {
            printf("%d ",temp->data);
            temp = temp->next;
        }
        printf("%d",temp->data);
    }
};

int main()
{
    createCircularListNode();
    displayNode();

    reverseList();
    displayNode();

    getch();
    return 0;
}
}

```

Output

```

Enter the data : 100
Do you want to continue (0 --> No,1 --> Yes) : 1
Enter the data : 200
Do you want to continue (0 --> No,1 --> Yes) : 1
Enter the data : 300
Do you want to continue (0 --> No,1 --> Yes) : 1
Enter the data : 400
Do you want to continue (0 --> No,1 --> Yes) : 0

```

```

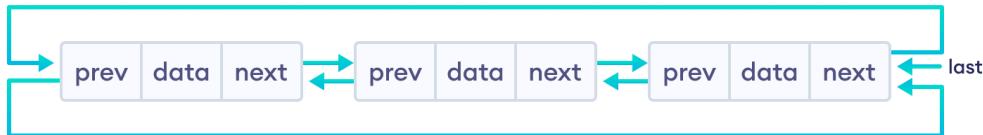
List is :
100 200 300 400
List is :
400 300 200 100

```

Watch Videos: -[Circular Linked List](#)[Implementation of Circular Linked List](#)[Circular Linked List – Insertion](#)[Circular linked list - Deletion \(from beginning, end, and specified position\)](#)[Reverse a Circular Linked List](#)

Doubly Circular Linked List

Here, in addition to the last node storing the address of the first node, the first node will also store the address of the last node.



Program to implement Doubly Circular Linked List

```
// Doubly Circular Linked List --Implementation
#include <stdio.h>
#include <stdlib.h>
#include <conio.h>

struct node
{
    int data;
    struct node *next;
    struct node *prev;
}*head,*tail;

void createDoublyCircularList()
{
    struct node *newnode;
    int choice;
    head = NULL;

    while(choice)
    {
        newnode = (struct node *)malloc(sizeof(struct node));
        printf("Enter the data : ");
        scanf("%d",&newnode->data);

        if(head == NULL)
        {
            head = tail = newnode;
            head->next = head;
            head->prev = head;
        }
        else
        {
            tail->next = newnode;
            newnode->prev = tail;
            newnode->next = head;
            head->prev = newnode;
            tail = newnode;
        }

        printf("Do you want to continue (0 --> No,1 --> Yes) : ");
        scanf("%d",&choice);
    }
    /* to check the list is Circular,
    enable below code and see the value is first node*/
    // printf("\n%d",tail->next->data); // it will print first node
    // printf("\n%d",head->prev->data); // it will print last node
}

void displayList()
{
    struct node *temp;
    temp = head;

    if (head == NULL)
    {
        printf("List is empty");
    }
}
```

```

else
{
    printf("\nList is : \n");
    while(temp != tail)
    {
        printf("%d ",temp->data);
        temp = temp->next;
    }
    printf("%d",temp->data);
}
}

int main()
{
    createDoublyCircularList();
    displayList();

    getch();
    return 0;
}

```

Output

```

Enter the data : 100
Do you want to continue (0 --> No,1 --> Yes) : 1
Enter the data : 200
Do you want to continue (0 --> No,1 --> Yes) : 1
Enter the data : 300
Do you want to continue (0 --> No,1 --> Yes) : 0

List is :
100 200 300

```

Program to implement Doubly Circular Linked List - Insert at the Beginning

```
// Doubly Circular Linked List -- Insert at the Beginning
#include <stdio.h>
#include <stdlib.h>
#include <conio.h>

struct node
{
    int data;
    struct node *next;
    struct node *prev;
}*head,*tail;

void createDoublyCircularList()
{
    struct node *newnode;
    int choice;
    head = NULL;

    while(choice)
    {
        newnode = (struct node *)malloc(sizeof(struct node));
        printf("Enter the data : ");
        scanf("%d",&newnode->data);

        if(head == NULL)
        {
            head = tail = newnode;
            head->next = head;
            head->prev = head;
        }
        else
        {
            tail->next = newnode;
            newnode->prev = tail;
            newnode->next = head;
            head->prev = newnode;
            tail = newnode;
        }

        printf("Do you want to continue (0 --> No,1 --> Yes) : ");
        scanf("%d",&choice);
    }
    /* to check the list is Circular,
       enable below code and see the value is first node*/
    // printf("\n%d",tail->next->data); // it will print first node
    // printf("\n%d",head->prev->data); // it will print last node
}

void insertNodeAtBeginning()
{
    struct node *newnode;
    newnode = (struct node *)malloc(sizeof(struct node));
    printf("\nEnter the data : ");
    scanf("%d",&newnode->data);

    if (head == NULL)
    {
        head = tail = newnode;
```

```

        newnode->prev = tail;
        newnode->next = head;
    }
    else
    {
        newnode->next = head;
        head->prev = newnode;
        newnode->prev = tail;
        tail->next = newnode;
        head = newnode;
    }
}

void displayList()
{
    struct node *temp;
    temp = head;

    if (head == NULL)
    {
        printf("List is empty");
    }
    else
    {
        printf("\nList is : \n");
        while(temp != tail)
        {
            printf("%d ",temp->data);
            temp = temp->next;
        }
        printf("%d",temp->data);
    }
}

int main()
{
    createDoublyCircularList();
    displayList();
    insertNodeAtBeginning();
    displayList();
    getch();
    return 0;
}

```

Output

```

Enter the data : 100
Do you want to continue (0 --> No,1 --> Yes) : 1
Enter the data : 200
Do you want to continue (0 --> No,1 --> Yes) : 1
Enter the data : 300
Do you want to continue (0 --> No,1 --> Yes) : 0

```

```

List is :
100 200 300
Enter the data : 50

```

```

List is :
50 100 200 300

```

Program to implement Doubly Circular Linked List - Insert at the End

```
// Doubly Circular Linked List -- Insert at the End
#include <stdio.h>
#include <stdlib.h>
#include <conio.h>

struct node
{
    int data;
    struct node *next;
    struct node *prev;
}*head,*tail;

void createDoublyCircularList()
{
    struct node *newnode;
    int choice;
    head = NULL;

    while(choice)
    {
        newnode = (struct node *)malloc(sizeof(struct node));
        printf("Enter the data : ");
        scanf("%d",&newnode->data);

        if(head == NULL)
        {
            head = tail = newnode;
            head->next = head;
            head->prev = head;
        }
        else
        {
            tail->next = newnode;
            newnode->prev = tail;
            newnode->next = head;
            head->prev = newnode;
            tail = newnode;
        }

        printf("Do you want to continue (0 --> No,1 --> Yes) : ");
        scanf("%d",&choice);
    }
}

void insertNodeAtEnd()
{
    struct node *newnode;

    newnode = (struct node *)malloc(sizeof(struct node));
    printf("\nEnter the data : ");
    scanf("%d",&newnode->data);

    if (head == NULL)
    {
        head = tail = newnode;
        newnode->prev = tail;
        newnode->next = head;
    }
}
```

```

    }
else
{
    newnode->prev = tail;
    tail->next = newnode;
    newnode->next = head;
    head->prev = newnode;
    tail = newnode;
}
}

void displayList()
{
    struct node *temp;
    temp = head;

    if (head == NULL)
    {
        printf("List is empty");
    }
else
{
    printf("\nList is : \n");
    while(temp != tail)
    {
        printf("%d ",temp->data);
        temp = temp->next;
    }
    printf("%d",temp->data);
}
}
}

int main()
{
    createDoublyCircularList();
    displayList();

    insertNodeAtEnd();
    displayList();

    getch();
    return 0;
}
}

```

Output

```

Enter the data : 100
Do you want to continue (0 --> No,1 --> Yes) : 1
Enter the data : 200
Do you want to continue (0 --> No,1 --> Yes) : 1
Enter the data : 300
Do you want to continue (0 --> No,1 --> Yes) : 0

```

```

List is :
100 200 300
Enter the data : 400

```

```

List is :
100 200 300 400

```

Program to implement Doubly Circular Linked List - Insert at the Specified Position

```
// Doubly Circular Linked List -- Insert at the Specified Position
#include <stdio.h>
#include <stdlib.h>
#include <conio.h>

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

*head,*tail;

void createDoublyCircularList()
{
    struct node *newnode;
    int choice;
    head = NULL;

    while(choice)
    {
        newnode = (struct node *)malloc(sizeof(struct node));
        printf("Enter the data : ");
        scanf("%d",&newnode->data);
        if(head == NULL)
        {
            head = tail = newnode;
            head->next = head;
            head->prev = head;
        }
        else
        {
            tail->next = newnode;
            newnode->prev = tail;
            newnode->next = head;
            head->prev = newnode;
            tail = newnode;
        }
        printf("Do you want to continue (0 --> No,1 --> Yes) : ");
        scanf("%d",&choice);
    }
}

void insertNodeAtSpecifiedPosition()
{
    struct node *newnode,*temp;
    int position,i=1;
    temp = head;
    printf("\nEnter the position : ");
    scanf("%d",&position);
    /*
        Add position validation
        position < 0 || position > length of the list
        if position == 1 , then use insertAtBeginning()
    */
    newnode = (struct node *)malloc(sizeof(struct node));
    printf("\nEnter the data : ");
    scanf("%d",&newnode->data);
```

```

while(i < position -1)
{
    temp = temp->next;
    i++;
}
newnode->prev      = temp;
newnode->next      = temp->next;
temp->next->prev = newnode;
temp->next        = newnode;
}

void displayList()
{
    struct node *temp;
    temp = head;

    if (head == NULL)
    {
        printf("List is empty");
    }
    else
    {
        printf("\nList is : \n");
        while(temp != tail)
        {
            printf("%d ",temp->data);
            temp = temp->next;
        }
        printf("%d",temp->data);
    }
}

int main()
{
    createDoublyCircularList();
    displayList();
    insertNodeAtSpecifiedPosition();
    displayList();
    getch();
    return 0;
}

```

Output

```

Enter the data : 100
Do you want to continue (0 --> No,1 --> Yes) : 1
Enter the data : 200
Do you want to continue (0 --> No,1 --> Yes) : 1
Enter the data : 300
Do you want to continue (0 --> No,1 --> Yes) : 0

List is :
100 200 300
Enter the position : 3
Enter the data : 250
List is :
100 200 250 300

```

Program to implement Doubly Circular Linked List - Delete at the Beginning

```
// Doubly Circular Linked List -- Delete at the Beginning
#include <stdio.h>
#include <stdlib.h>
#include <conio.h>

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

}*head,*tail;

void createDoublyCircularList()
{
    struct node *newnode;
    int choice;
    head = NULL;

    while(choice)
    {
        newnode = (struct node *)malloc(sizeof(struct node));
        printf("Enter the data : ");
        scanf("%d",&newnode->data);

        if(head == NULL)
        {
            head = tail = newnode;
            head->next = head;
            head->prev = head;
        }
        else
        {
            tail->next = newnode;
            newnode->prev = tail;
            newnode->next = head;
            head->prev = newnode;
            tail = newnode;
        }

        printf("Do you want to continue (0 --> No,1 --> Yes) : ");
        scanf("%d",&choice);
    }
}

void deleteNodeAtBeginning()
{
    struct node *temp;
    temp = head;

    if(head == NULL)
    {
        printf("List is empty");
    }
    // Only single node
    else if (head->next == head)
    {
```

```

        head = tail = NULL;
        free(temp);
    }
else
{
    head = head->next;
    head->prev = tail;
    tail->next = head;
    free(temp);
}
}

void displayList()
{
    struct node *temp;
    temp = head;

    if (head == NULL)
    {
        printf("List is empty");
    }
else
{
    printf("\nList is : \n");
    while(temp != tail)
    {
        printf("%d ",temp->data);
        temp = temp->next;
    }
    printf("%d",temp->data);
}
}

int main()
{
    createDoublyCircularList();
    displayList();

    deleteNodeAtBeginning();
    displayList();

    getch();
    return 0;
}

```

Output

```

Enter the data : 100
Do you want to continue (0 --> No,1 --> Yes) : 1
Enter the data : 200
Do you want to continue (0 --> No,1 --> Yes) : 1
Enter the data : 300
Do you want to continue (0 --> No,1 --> Yes) : 0
List is :
100 200 300
List is :
200 300

```

Program to implement Doubly Circular Linked List - Delete at the End

```
// Doubly Circular Linked List -- Delete at the End
#include <stdio.h>
#include <stdlib.h>
#include <conio.h>

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

}*head,*tail;

void createDoublyCircularList()
{
    struct node *newnode;
    int choice;
    head = NULL;

    while(choice)
    {
        newnode = (struct node *)malloc(sizeof(struct node));
        printf("Enter the data : ");
        scanf("%d",&newnode->data);

        if(head == NULL)
        {
            head = tail = newnode;
            head->next = head;
            head->prev = head;
        }
        else
        {
            tail->next = newnode;
            newnode->prev = tail;
            newnode->next = head;
            head->prev = newnode;
            tail = newnode;
        }

        printf("Do you want to continue (0 --> No,1 --> Yes) : ");
        scanf("%d",&choice);
    }
}

void deleteNodeAtEnd()
{
    struct node *temp;
    temp = tail;

    if(head == NULL)
    {
        printf("List is empty");
    }
    // Only single node
    else if (head->next == head)
    {
        head = tail = NULL;
    }
}
```

```

        free(temp);
    }
else
{
    tail = tail->prev;
    tail->next = head;
    head->prev = tail;
    free(temp);
}

void displayList()
{
    struct node *temp;
    temp = head;

    if (head == NULL)
    {
        printf("List is empty");
    }
else
{
    printf("\nList is : \n");
    while(temp != tail)
    {
        printf("%d ",temp->data);
        temp = temp->next;
    }
    printf("%d",temp->data);
}
}

int main()
{
    createDoublyCircularList();
    displayList();

    deleteNodeAtEnd();
    displayList();

    getch();
    return 0;
}

```

Output

```

Enter the data : 100
Do you want to continue (0 --> No,1 --> Yes) : 1
Enter the data : 200
Do you want to continue (0 --> No,1 --> Yes) : 1
Enter the data : 300
Do you want to continue (0 --> No,1 --> Yes) : 0

List is :
100 200 300
List is :
100 200

```

Program to implement Doubly Circular Linked List - Delete at the Specified Position

```
// Doubly Circular Linked List -- Delete at the Specified Position
#include <stdio.h>
#include <stdlib.h>
#include <conio.h>

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

*head,*tail;

void createDoublyCircularList()
{
    struct node *newnode;
    int choice;
    head = NULL;
    while(choice)
    {
        newnode = (struct node *)malloc(sizeof(struct node));
        printf("Enter the data : ");
        scanf("%d",&newnode->data);
        if(head == NULL)
        {
            head = tail = newnode;
            head->next = head;
            head->prev = head;
        }
        else
        {
            tail->next = newnode;
            newnode->prev = tail;
            newnode->next = head;
            head->prev = newnode;
            tail = newnode;
        }
        printf("Do you want to continue (0 --> No,1 --> Yes) : ");
        scanf("%d",&choice);
    }
}

void deleteNodeAtSpecifiedPosition()
{
    struct node *temp;
    int position, i=1;
    temp = head;

    printf("\nEnter the position : ");
    scanf("%d",&position);
    /*
    position validation
    if position < 1 || position > length of list
    if position == 1 then call function delete at the beginning
    */
    while(i < position)
    {
        temp = temp->next;
    }
}
```

```

        i++;
    }
    temp->prev->next = temp->next;
    temp->next->prev = temp->prev;

    if(temp->next == head)
    {
        tail = temp->prev;
    }
    free(temp);
}

void displayList()
{
    struct node *temp;
    temp = head;

    if (head == NULL)
    {
        printf("List is empty");
    }
    else
    {
        printf("\nList is : \n");
        while(temp != tail)
        {
            printf("%d ",temp->data);
            temp = temp->next;
        }
        printf("%d",temp->data);
    }
}
}

int main()
{
    createDoublyCircularList();
    displayList();
    deleteNodeAtSpecifiedPosition();
    displayList();
    getch();
    return 0;
}
}

```

Output

```

Enter the data : 100
Do you want to continue (0 --> No,1 --> Yes) : 1
Enter the data : 200
Do you want to continue (0 --> No,1 --> Yes) : 1
Enter the data : 300
Do you want to continue (0 --> No,1 --> Yes) : 1
Enter the data : 400
Do you want to continue (0 --> No,1 --> Yes) : 0

```

```

List is :
100 200 300 400
Enter the position : 3
List is :
100 200 400

```

Applications of Linked Lists

- Used for symbol table management in compiler design.
- Used in switching between programs using Alt + Tab (implemented using Circular Linked List).

Watch Videos: -

[Doubly Circular Linked List](#)

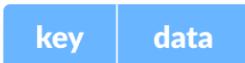
[Doubly Circular linked list - Insertion](#)

[Doubly Circular Linked List - Deletion](#)

Hash Table

The Hash table data structure stores elements in key-value pairs where

- o Key- unique integer that is used for indexing the values
- o Value - data that are associated with keys.

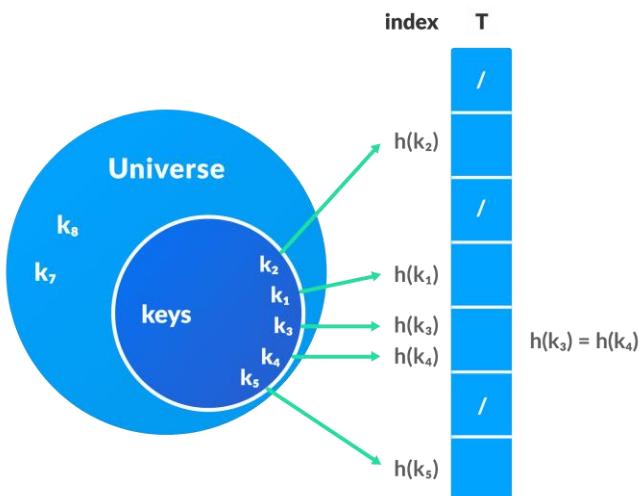


Hashing (Hash Function)

In a hash table, a new index is processed using the keys. And, the element corresponding to that key is stored in the index. This process is called hashing.

Let k be a key and $h(k)$ be a hash function.

Here, $h(k)$ will give us a new index to store the element linked with k .



Hash Collision

When the hash function generates the same index for multiple keys, there will be a conflict (what value to be stored in that index). This is called a hash collision.

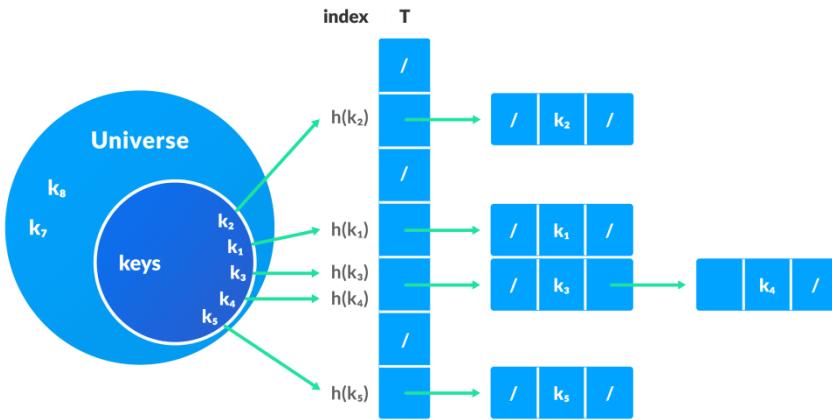
We can resolve the hash collision using one of the following techniques.

- o Collision resolution by chaining
- o Open Addressing: Linear/Quadratic Probing and Double Hashing

1. Collision resolution by chaining

In chaining, if a hash function produces the same index for multiple elements, these elements are stored in the same index by using a doubly-linked list.

If j is the slot for multiple elements, it contains a pointer to the head of the list of elements. If no element is present, j contains **NIL**.



Pseudocode for operations

```

chainedHashSearch(T, k)
    return T[h(k)]
chainedHashInsert(T, x)
    T[h(x.key)] = x //insert at the head
chainedHashDelete(T, x)
    T[h(x.key)] = NIL

```

2. Open Addressing

Unlike chaining, open addressing doesn't store multiple elements into the same slot. Here, each slot is either filled with a single key or left NIL.

Different techniques used in open addressing are:

Linear Probing

In linear probing, collision is resolved by checking the next slot.

$$h(k, i) = (h'(k) + i) \bmod m$$

where

$$i = \{0, 1, \dots\}$$

$h'(k)$ is a new hash function

If a collision occurs at $h(k, 0)$, then $h(k, 1)$ is checked. In this way, the value of i is incremented linearly.

The problem with linear probing is that a cluster of adjacent slots is filled. When inserting a new element, the entire cluster must be traversed. This adds to the time required to perform operations on the hash table.

Quadratic Probing

It works similar to linear probing but the spacing between the slots is increased (greater than one) by using the following relation.

$$h(k, i) = (h'(k) + c_1 i + c_2 i^2) \bmod m$$

where,

c_1 and c_2 are positive auxiliary constants,

$i = \{0, 1, \dots\}$

Double hashing

If a collision occurs after applying a hash function $h(k)$, then another hash function is calculated for finding the next slot.

$$h(k, i) = (h_1(k) + ih_2(k)) \bmod m$$

Good Hash Functions

A good hash function may not prevent the collisions completely however it can reduce the number of collisions.

Here, we will look into different methods to find a good hash function

Division Method

If k is a key and m is the size of the hash table, the hash function $h()$ is calculated as:

$$h(k) = k \bmod m$$

For example, If the size of a hash table is 10 and $k = 112$ then $h(k) = 112 \bmod 10 = 2$. The value of m must not be the powers of 2. This is because the powers of 2 in binary format are 10, 100, 1000, When we find $k \bmod m$, we will always get the lower order p -bits.

```
if m = 22, k = 17, then h(k) = 17 mod 22 = 10001 mod 100 = 01
if m = 23, k = 17, then h(k) = 17 mod 22 = 10001 mod 100 = 001
if m = 24, k = 17, then h(k) = 17 mod 22 = 10001 mod 100 = 0001
if m = 2p, then h(k) = p lower bits of m
```

Multiplication Method

$$h(k) = \lfloor m(kA \bmod 1) \rfloor$$

where,

- o $kA \bmod 1$ gives the fractional part kA ,
- o $\lfloor \cdot \rfloor$ gives the floor value
- o A is any constant. The value of A lies between 0 and 1. But, an optimal choice will be $\approx (\sqrt{5}-1)/2$ suggested by Knuth.

Universal Hashing

In Universal hashing, the hash function is chosen at random independent of keys.

Watch Videos :-

[Hashing Technique](#)

[Hashing techniques to resolve collision | Separate chaining and Linear Probing](#)

[Hashing - Quadratic Probing | Collision Resolution Technique](#)

[Hashing: Double Hashing | Collision Resolution Technique](#)

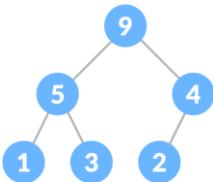
Heap Data Structure

Heap data structure is a complete binary tree that satisfies the heap property, where any given node is

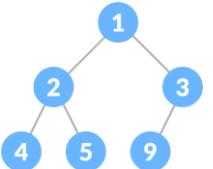
always greater than its child node/s and the key of the root node is the largest among all other nodes. This property is also called max heap property.

always smaller than the child node/s and the key of the root node is the smallest among all other nodes. This property is also called min heap property.

Max-heap



Min-heap



This type of data structure is also called a binary heap.

Heap Operations

Some of the important operations performed on a heap are described below along with their algorithms.

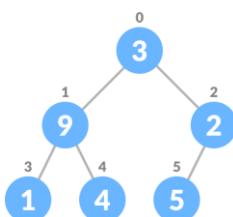
Heapify

Heapify is the process of creating a heap data structure from a binary tree. It is used to create a Min-Heap or a Max-Heap.

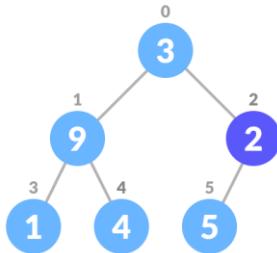
1. Let the input array be

3	9	2	1	4	5
0	1	2	3	4	5

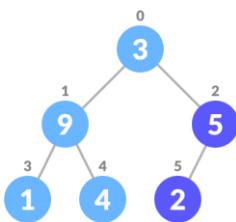
2. Create a complete binary tree from the array



3. Start from the first index of non-leaf node whose index is given by $n/2 - 1$.



4. Set current element i as target
 5. The index of left child is given by $2i + 1$ and the right child is given by $2i + 2$.
 If leftChild is greater than currentElement (i.e. element at i th index), set leftChildIndex as largest.
 If rightChild is greater than element in largest , set rightChildIndex as largest.
 6. Swap largest with currentElement



7. Repeat steps 3-7 until the subtrees are also heapified.

Algorithm

```

Heapify(array, size, i)
  set i as largest
  leftChild =  $2i + 1$ 
  rightChild =  $2i + 2$ 

  if leftChild > array[largest]
    set leftChildIndex as largest
  if rightChild > array[largest]
    set rightChildIndex as largest

  swap array[i] and array[largest]
  
```

To create a Max-Heap:

```

MaxHeap(array, size)
  loop from the first index of non-leaf node down to zero
    call Heapify
  
```

For Min-Heap, both leftChild and rightChild must be larger than the parent for all nodes.

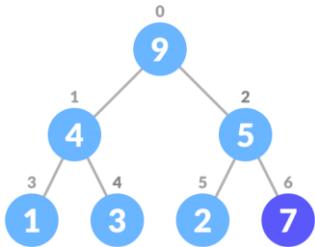
Insert Element into Heap

Algorithm for insertion in Max Heap

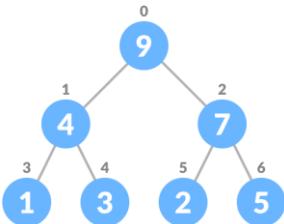
```
If there is no node,
  create a newNode.
else (a node is already present)
  insert the newNode at the end (last node from left to right.)
```

heapify the array

1. Insert the new element at the end of the tree.



2. Heapify the tree.



For Min Heap, the above algorithm is modified so that parentNode is always smaller than newNode.

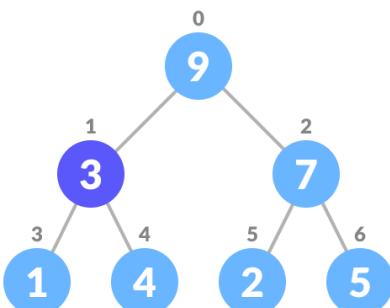
Delete Element from Heap

Algorithm for deletion in Max Heap

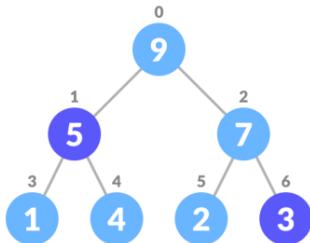
```
If nodeToBeDeleted is the leafNode
  remove the node
Else swap nodeToBeDeleted with the lastLeafNode
  remove noteToBeDeleted
```

heapify the array

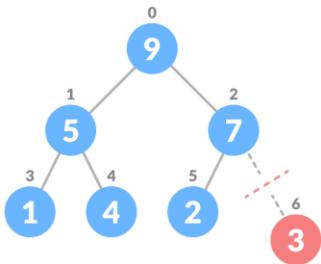
1. Select the element to be deleted.



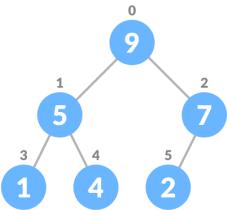
2. Swap it with the last element.



3. Remove the last element.



4. Heapify the tree.



For Min Heap, above algorithm is modified so that both childNodes are greater than currentNode.

Peek (Find max/min)

Peek operation returns the maximum element from Max Heap or minimum element from Min Heap without deleting the node.

For both Max heap and Min Heap

```
return rootNode
```

Extract-Max/Min

Extract-Max returns the node with maximum value after removing it from a Max Heap whereas Extract-Min returns the node with minimum after removing it from Min Heap.

Watch Videos: -

[Max Heap Insertion and Deletion](#)

[Heap](#)

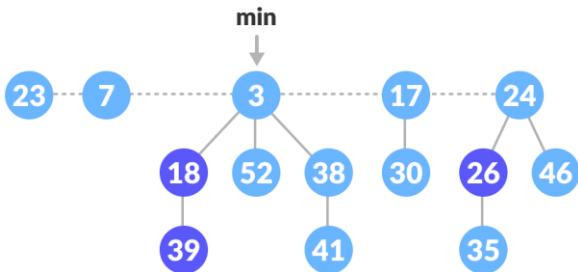
[Heap - Heap Sort - Heapify - Priority Queues](#)

Fibonacci Heap

A fibonacci heap is a data structure that consists of a collection of trees which follow min heap or max heap property. These two properties are the characteristics of the trees present on a fibonacci heap.

In a fibonacci heap, a node can have more than two children or no children at all. Also, it has more efficient heap operations than that supported by the binomial and binary heaps.

The fibonacci heap is called a fibonacci heap because the trees are constructed in a way such that a tree of order n has at least F_{n+2} nodes in it, where F_{n+2} is the $(n + 2)$ th Fibonacci number.



Properties of a Fibonacci Heap

Important properties of a Fibonacci heap are:

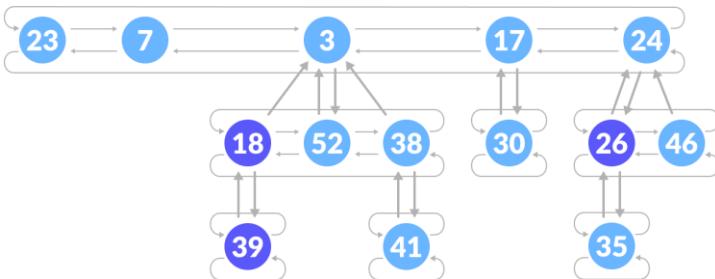
- o It is a set of min heap-ordered trees. (i.e. The parent is always smaller than the children.)
- o A pointer is maintained at the minimum element node.
- o It consists of a set of marked nodes. (Decrease key operation)
- o The trees within a Fibonacci heap are unordered but rooted.

Memory Representation of the Nodes in a Fibonacci Heap

The roots of all the trees are linked together for faster access. The child nodes of a parent node are connected to each other through a circular doubly linked list as shown below.

There are two main advantages of using a circular doubly linked list.

- o Deleting a node from the tree takes O(1) time.
- o The concatenation of two such lists takes O(1) time.



Operations on a Fibonacci Heap

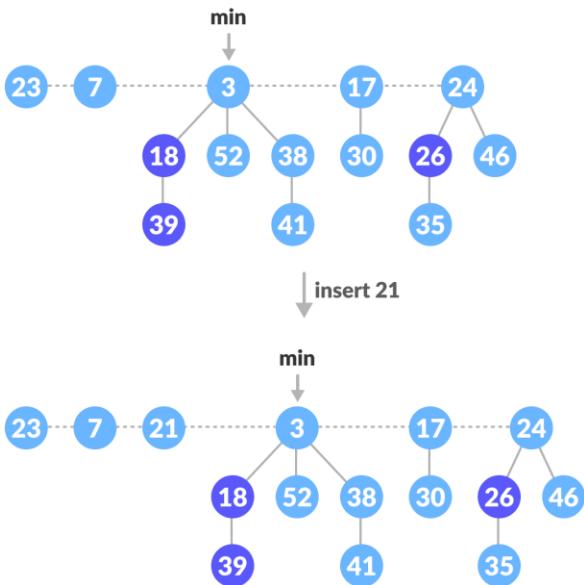
Insertion

Algorithm

```
insert(H, x)
    degree[x] = 0
    p[x] = NIL
    child[x] = NIL
    left[x] = x
    right[x] = x
    mark[x] = FALSE
    concatenate the root list containing x with root list H
    if min[H] == NIL or key[x] < key[min[H]]
        then min[H] = x
    n[H] = n[H] + 1
```

Inserting a node into an already existing heap follows the steps below.

1. Create a new node for the element.
2. Check if the heap is empty.
3. If the heap is empty, set the new node as a root node and mark it min.
4. Else, insert the node into the root list and update min.



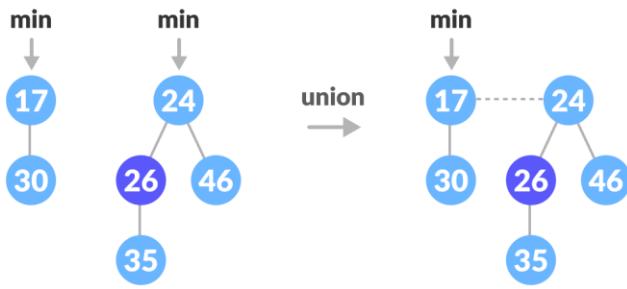
Find Min

The minimum element is always given by the min pointer.

Union

Union of two fibonacci heaps consists of following steps.

1. Concatenate the roots of both the heaps.
2. Update min by selecting a minimum key from the new root lists.



Extract Min

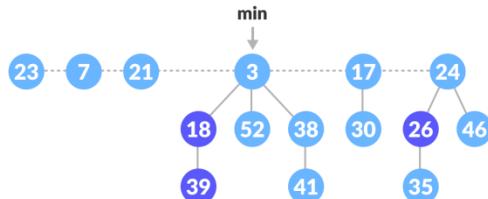
It is the most important operation on a fibonacci heap. In this operation, the node with minimum value is removed from the heap and the tree is re-adjusted.

The following steps are followed:

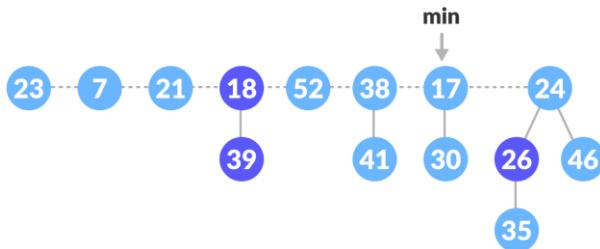
1. Delete the min node.
2. Set the min-pointer to the next root in the root list.
3. Create an array of size equal to the maximum degree of the trees in the heap before deletion.
4. Do the following (steps 5-7) until there are no multiple roots with the same degree.
5. Map the degree of current root (min-pointer) to the degree in the array.
6. Map the degree of next root to the degree in array.
7. If there are more than two mappings for the same degree, then apply union operation to those roots such that the min-heap property is maintained (i.e. the minimum is at the root).

An implementation of the above steps can be understood in the example below.

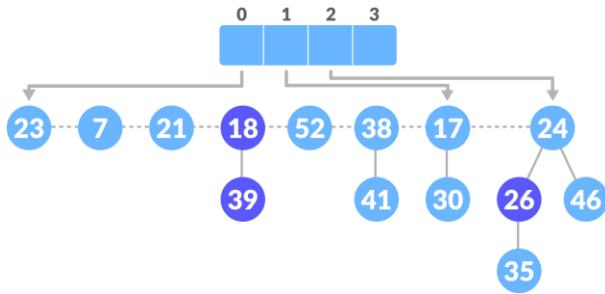
1. We will perform an extract-min operation on the heap below.



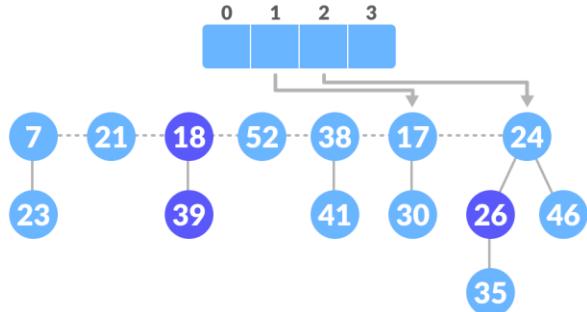
2. Delete the min node, add all its child nodes to the root list and set the min-pointer to the next root in the root list.



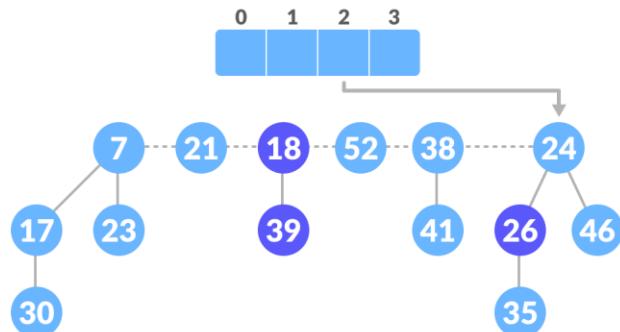
3. The maximum degree in the tree is 3. Create an array of size 4 and map degree of the next roots with the array.



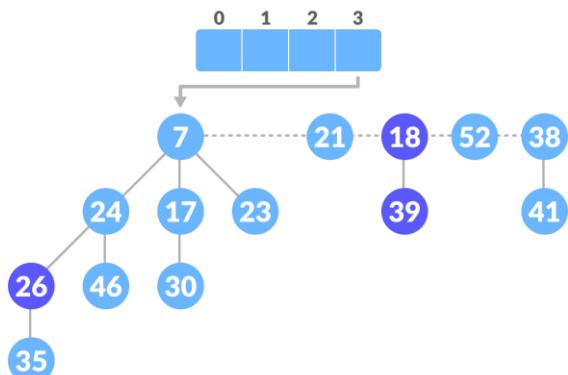
4. Here, 23 and 7 have the same degrees, so unite them.



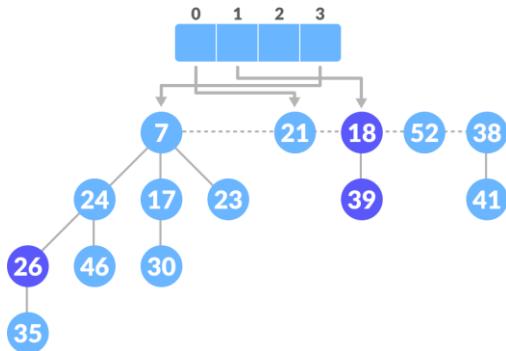
5. Again, 7 and 17 have the same degrees, so unite them as well.



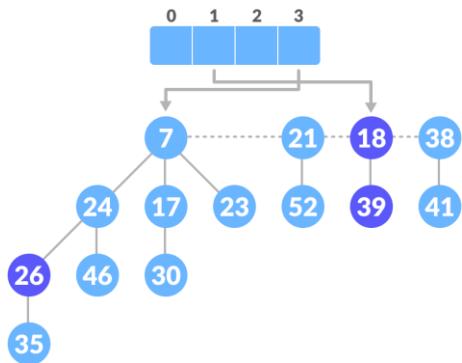
6. Again 7 and 24 have the same degree, so unite them.



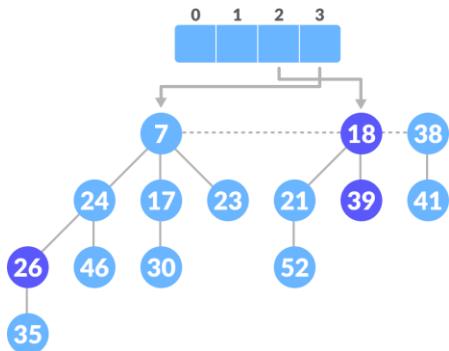
7. Map the next nodes.



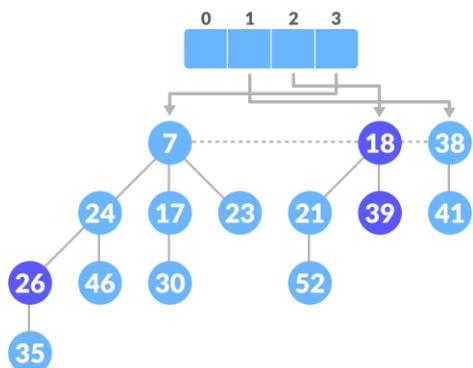
8. Again, 52 and 21 have the same degree, so unite them



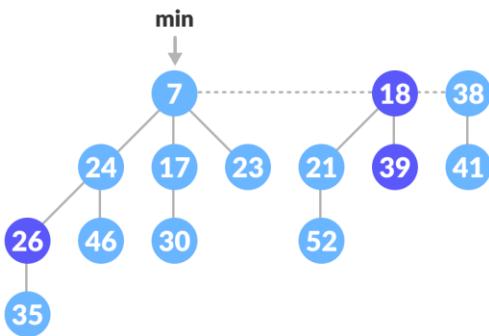
9. Similarly, unite 21 and 18.



10. Map the remaining root.



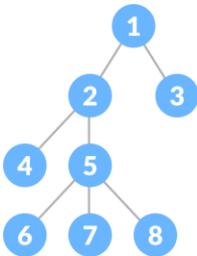
11. The final heap is.



4. Tree Based DSA(I)

Tree Data Structure

A tree is a nonlinear hierarchical data structure that consists of nodes connected by edges.



Why Tree Data Structure?

Other data structures such as arrays, linked list, stack, and queue are linear data structures that store data sequentially. In order to perform any operation in a linear data structure, the time complexity increases with the increase in the data size. But, it is not acceptable in today's computational world.

Different tree data structures allow quicker and easier access to the data as it is a non-linear data structure.

Node

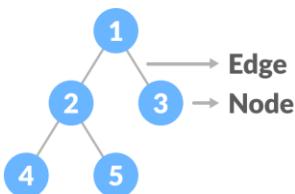
A node is an entity that contains a key or value and pointers to its child nodes.

The last nodes of each path are called leaf nodes or external nodes that do not contain a link/pointer to child nodes.

The node having at least a child node is called an internal node.

Edge

It is the link between any two nodes.



Root

It is the topmost node of a tree.

Height of a Node

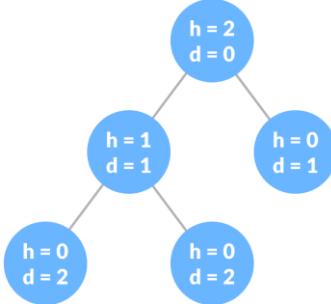
The height of a node is the number of edges from the node to the deepest leaf (ie. the longest path from the node to a leaf node).

Depth of a Node

The depth of a node is the number of edges from the root to the node.

Height of a Tree

The height of a Tree is the height of the root node or the depth of the deepest node.

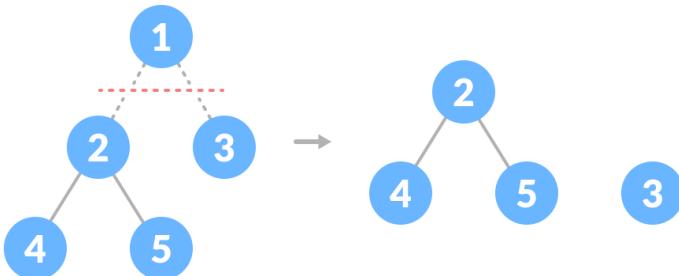


Degree of a Node

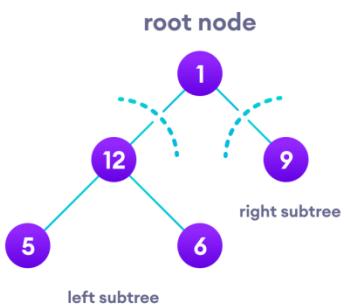
The degree of a node is the total number of branches of that node.

Forest

A collection of disjoint trees is called a forest.



You can create a forest by cutting the root of a tree.



Types of Tree

- Binary Tree
- Binary Search Tree
- AVL Tree
- B-Tree

Tree Traversal

In order to perform any operation on a tree, you need to reach to the specific node. The tree traversal algorithm helps in visiting a required node in the tree.

Tree Applications

- **Binary Search Trees**(BSTs) are used to quickly check whether an element is present in a set or not.
- **Heap** is a kind of tree that is used for heap sort.
- A modified version of a tree called **Tries** is used in modern routers to store routing information.
- Most popular databases use **B-Trees and T-Trees**, which are variants of the tree structure we learned above to store their data
- Compilers use a syntax tree to validate the syntax of every program you write.

Tree Traversal

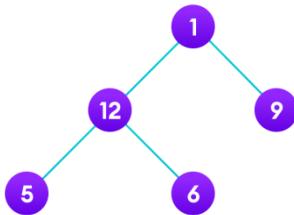
Traversal is a process to visit all the nodes of a tree and may print their values too. Because, **all nodes are connected via edges (links) we always start from the root (head) node.** That is, we cannot randomly access a node in a tree. There are three ways which we use to traverse a tree –

- In-order Traversal
- Pre-order Traversal
- Post-order Traversal

Generally, we traverse a tree to search or locate a given item or key in the tree or to print all the values it contains.

You might, for instance, want to add all the values in the tree or find the largest one. For all these operations, you will need to visit each node of the tree.

Linear data structures like arrays, stacks, queues, and linked list have only one way to read the data. But a hierarchical data structure like a tree can be traversed in different ways.



Let's think about how we can read the elements of the tree in the image shown above.

Starting from top, Left to right

1 → 12 → 5 → 6 → 9

Starting from bottom, Left to right

5 → 6 → 12 → 9 → 1

Although this process is somewhat easy, it doesn't respect the hierarchy of the tree, only the depth of the nodes.

Instead, we use traversal methods that take into account the basic structure of a tree i.e.

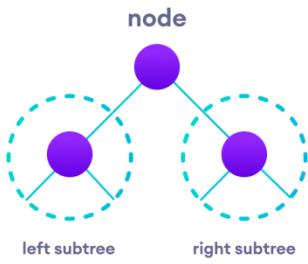
```

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

The struct node pointed to by left and right might have other left and right children so we should think of them as sub-trees instead of sub-nodes.

According to this structure, every tree is a combination of

- A node carrying data
- Two subtrees



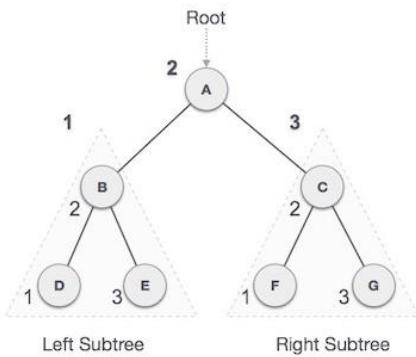
Remember that our goal is to visit each node, so we need to visit all the nodes in the subtree, visit the root node and visit all the nodes in the right subtree as well.

Depending on the order in which we do this, there can be three types of traversal.

In-order Traversal

In this traversal method, **the left subtree is visited first, then the root and later the right sub-tree**. We should always remember that every node may represent a subtree itself.

If a binary tree is traversed in-order, the output will produce sorted key values in an ascending order.



We start from **A**, and following in-order traversal, we move to its left subtree **B**. **B** is also traversed in-order. The process goes on until all the nodes are visited. The output of in-order traversal of this tree will be –

D → B → E → A → F → C → G

Algorithm

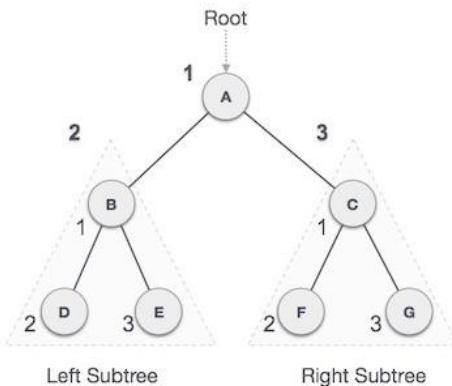
Until all nodes are traversed –

- Step 1 – Recursively traverse **left subtree**.
- Step 2 – Visit root node.
- Step 3 – Recursively traverse **right subtree**.

```
inorder(root->left)
display(root->data)
inorder(root->right)
```

Pre-order Traversal

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



We start from **A**, and following pre-order traversal, we first visit **A** itself and then move to its left subtree **B**. **B** is also traversed pre-order. The process goes on until all the nodes are visited. The output of pre-order traversal of this tree will be –

A → B → D → E → C → F → G

Algorithm

Until all nodes are traversed –

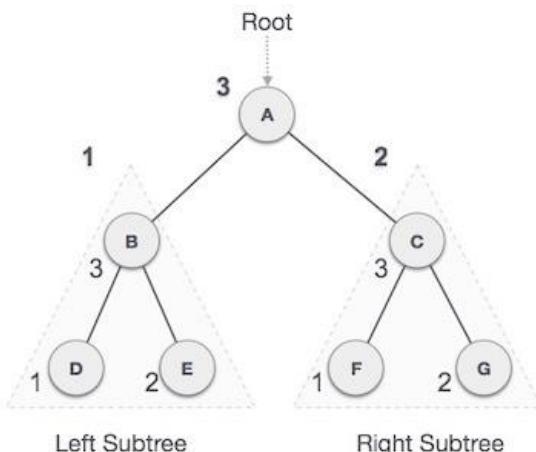
- Step 1 – Visit **root node**.
- Step 2 – Recursively traverse **left subtree**.
- Step 3 – Recursively traverse **right subtree**.

```

display(root->data)
preorder(root->left)
preorder(root->right)
  
```

Post-order Traversal

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



We start from **A**, and following Post-order traversal, we first visit the left subtree **B**. **B** is also traversed post-order. The process goes on until all the nodes are visited. The output of post-order traversal of this tree will be –

D → E → B → F → G → C → A

Algorithm

Until all nodes are traversed –

- Step 1 – Recursively traverse **left subtree**.
- Step 2 – Recursively traverse **right subtree**.
- Step 3 – Visit **root node**.

```
postorder(root->left)
postorder(root->right)
display(root->data)
```

Program to implement Tree and Traversal

```

// Tree traversal
#include <stdio.h>
#include <stdlib.h>

struct node
{
    int item;
    struct node* left;
    struct node* right;
};

// Inorder traversal
void inorderTraversal(struct node* root)
{
    if (root == NULL) return;
    inorderTraversal(root->left);
    printf("%d ->", root->item);
    inorderTraversal(root->right);
}

// preorderTraversal traversal
void preorderTraversal(struct node* root)
{
    if (root == NULL) return;
    printf("%d ->", root->item);
    preorderTraversal(root->left);
    preorderTraversal(root->right);
}

// postorderTraversal traversal
void postorderTraversal(struct node* root)
{
    if (root == NULL) return;
    postorderTraversal(root->left);
    postorderTraversal(root->right);
    printf("%d ->", root->item);
}

// Create a new Node
struct node* createNode(int value)
{
    struct node* newNode = malloc(sizeof(struct node));
    newNode->item = value;
    newNode->left = NULL;
    newNode->right = NULL;

    return newNode;
}

// Insert on the left of the node
struct node* insertLeft(struct node* root, int value)
{
    root->left = createNode(value);
    return root->left;
}

```

```

// Insert on the right of the node
struct node* insertRight(struct node* root, int value)
{
    root->right = createNode(value);
    return root->right;
}

int main()
{
    struct node* root = createNode(1);
    insertLeft(root, 12);
    insertRight(root, 9);

    insertLeft(root->left, 5);
    insertRight(root->left, 6);

    printf("Inorder traversal \n");
    inorderTraversal(root);

    printf("\nPreorder traversal \n");
    preorderTraversal(root);

    printf("\nPostorder traversal \n");
    postorderTraversal(root);
}

```

Output

```

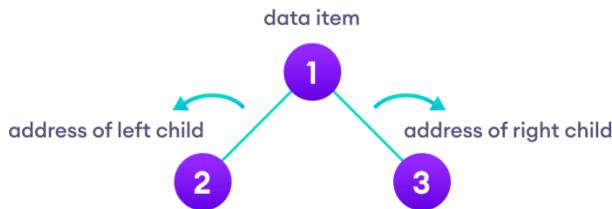
Inorder traversal
5 ->12 ->6 ->1 ->9 ->
Preorder traversal
1 ->12 ->5 ->6 ->9 ->
Postorder traversal
5 ->6 ->12 ->9 ->1 ->

```

Binary Tree

A **binary tree** is a tree data structure in which each parent node can have at most two children. Each node of a binary tree consists of three items:

- data item
- address of left child
- address of right child



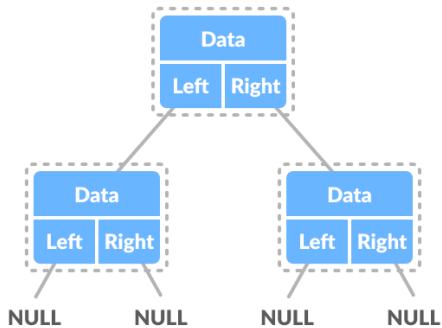
A tree whose elements have at most 2 children is called a **binary tree**. Since each element in a binary tree can have only 2 children, we typically name them the left and right child.

Binary Tree Representation

A node of a binary tree is represented by a structure containing a data part and two pointers to other structures of the same type.

```

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



Types of Binary Tree

- Full Binary Tree
- Perfect Binary Tree
- Complete Binary Tree
- Balanced Binary Tree

Program to implementation Binary Tree and Tree Traversal

```
// Implementation of Binary Tree - Traversal - Recursive Method
#include <stdio.h>
#include <stdlib.h>
#include <conio.h>

struct node
{
    int data;
    struct node *left,*right;
};

struct node * create()
{
    int item;
    struct node *newnode;
    newnode = (struct node *)malloc(sizeof(struct node));

    printf("\nEnter the value (-1 for No Node) : ");
    scanf("%d",&item);

    if(item == -1){
        return NULL;
    }
    newnode->data = item;
    printf("\nEnter left child of %d : ",item);
    newnode->left = create();
    printf("\nEnter right child of %d : ",item);
    newnode->right = create();
    return newnode;
}

void preOrder(struct node *root)
{
    if(root == NULL){
        return;
    }
    printf("%d -> ",root->data);
    preOrder(root->left);
    preOrder(root->right);
}

void inOrder(struct node *root)
{
    if(root == NULL){
        return;
    }
    inOrder(root->left);
    printf("%d -> ",root->data);
    inOrder(root->right);
}

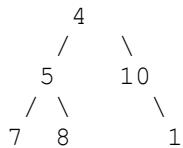
void postOrder(struct node *root)
{
    if(root == NULL){
        return;
    }
    postOrder(root->left);
    postOrder(root->right);
```

```

    printf("%d -> ",root->data);
}

int main()
{
    struct node *root;
    root = NULL;
    root =create();
    printf("\nPreorder is : ");
    preOrder(root);
    printf("\nInorder is : ");
    inOrder(root);
    printf("\nPostorder is : ");
    postOrder(root);
    getch();
    return 0;
}

```

Output

```

Enter the value (-1 for No Node) : 4
Enter left child of 4 :
Enter the value (-1 for No Node) : 5
Enter left child of 5 :
Enter the value (-1 for No Node) : 7
Enter left child of 7 :
Enter the value (-1 for No Node) : -1
Enter right child of 7 :
Enter the value (-1 for No Node) : -1

Enter right child of 5 :
Enter the value (-1 for No Node) : 8
Enter left child of 8 :
Enter the value (-1 for No Node) : -1
Enter right child of 8 :
Enter the value (-1 for No Node) : -1

Enter right child of 4 :
Enter the value (-1 for No Node) : 10
Enter left child of 10 :
Enter the value (-1 for No Node) : -1
Enter right child of 10 :
Enter the value (-1 for No Node) : 1

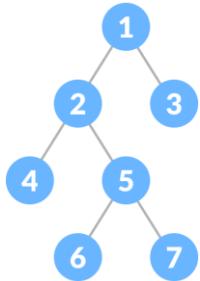
Enter left child of 1 :
Enter the value (-1 for No Node) : -1
Enter right child of 1 :
Enter the value (-1 for No Node) : -1

Preorder is : 4 -> 5 -> 7 -> 8 -> 10 -> 1
Inorder is : 7 -> 5 -> 8 -> 4 -> 10 -> 1
Postorder is : 7 -> 8 -> 5 -> 1 -> 10 -> 4
  
```

Watch Videos: -[Introduction to Trees](#)[Binary Tree and its Types](#)[Binary Tree Implementation](#)[Array representation of Binary Tree](#)[Binary Tree Traversals \(Inorder, Preorder and Postorder\)](#)[Binary Tree traversal : Preorder, Inorder, Postorder](#)[Construct Binary Tree from Preorder and Inorder traversal](#)[Construct Binary Tree from Postorder and Inorder](#)[Construct Binary Tree from Preorder and Postorder traversal](#)

Full Binary Tree

A full Binary tree is a special type of binary tree in which every parent node/internal node has either two or no children.



It is also known as a [proper binary tree](#).

Program to implement Full Binary Tree

```

// Checking if a binary tree is a full binary tree

#include <stdbool.h>
#include <stdio.h>
#include <stdlib.h>

struct node
{
    int data;
    struct node *left,*right;
};

// Creation of new Node
struct node * create()
{
    int item;
    struct node *newnode;
    newnode = (struct node *)malloc(sizeof(struct node));

    printf("\nEnter the value (-1 for No Node) : ");
    scanf("%d",&item);

    if(item == -1){
        return NULL;
    }

    newnode->data = item;
    printf("\nEnter left child of %d : ",item);
    newnode->left = create();

    printf("\nEnter right child of %d : ",item);
    newnode->right = create();

    return newnode;
}

bool isFullBinaryTree(struct node *root) {
    // Checking tree emptiness
    if (root == NULL)
        return true;

    // Checking the presence of children
    if (root->left == NULL && root->right == NULL)
        return true;

    if ((root->left) && (root->right))
        return (isFullBinaryTree(root->left) && isFullBinaryTree(root->right));

    return false;
}

int main() {
    struct node *root;
    root = NULL;
    root =create();

    if (isFullBinaryTree(root))
        printf("The tree is a full binary tree\n");
}

```

```

else
    printf("The tree is not a full binary tree\n");
}

```

Output

```

Enter the value (-1 for No Node) : 1

Enter left child of 1 :
Enter the value (-1 for No Node) : 2

Enter left child of 2 :
Enter the value (-1 for No Node) : 4

Enter left child of 4 :
Enter the value (-1 for No Node) : -1

Enter right child of 4 :
Enter the value (-1 for No Node) : -1

Enter right child of 2 :
Enter the value (-1 for No Node) : 5

Enter left child of 5 :
Enter the value (-1 for No Node) : -1

Enter right child of 5 :
Enter the value (-1 for No Node) : -1

Enter right child of 1 :
Enter the value (-1 for No Node) : 3

Enter left child of 3 :
Enter the value (-1 for No Node) : 6

Enter left child of 6 :
Enter the value (-1 for No Node) : -1

Enter right child of 6 :
Enter the value (-1 for No Node) : -1

Enter right child of 3 :
Enter the value (-1 for No Node) : 7

Enter left child of 7 :
Enter the value (-1 for No Node) : -1

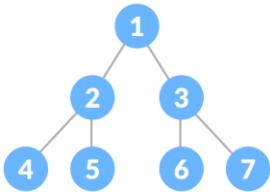
Enter right child of 7 :
Enter the value (-1 for No Node) : -1

```

The tree is a full binary tree

Perfect Binary Tree

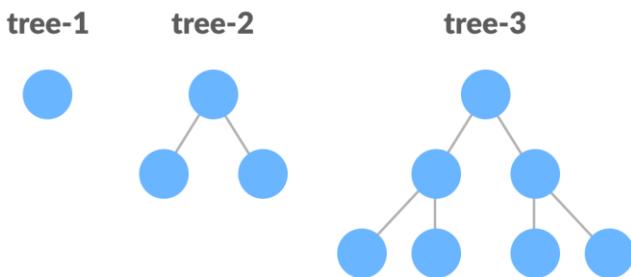
A perfect binary tree is a type of binary tree in which every internal node has exactly two child nodes and all the leaf nodes are at the same level.



All the internal nodes have a degree of 2.

Recursively, a perfect binary tree can be defined as:

- o If a single node has no children, it is a perfect binary tree of height $h = 0$,
- o If a node has $h > 0$, it is a perfect binary tree if both of its subtrees are of height $h - 1$ and are non-overlapping.



Program to implement Perfect Binary Tree

```
// Checking if a binary tree is a perfect binary tree

#include <stdbool.h>
#include <stdio.h>
#include <stdlib.h>

struct node
{
    int data;
    struct node *left,*right;
};

struct node * create()
{
    int item;
    struct node *newnode;
    newnode = (struct node *)malloc(sizeof(struct node));

    printf("\nEnter the value (-1 for No Node) : ");
    scanf("%d",&item);

    if(item == -1){
        return NULL;
    }

    newnode->data = item;
    printf("\nEnter left child of %d : ",item);
    newnode->left = create();

    printf("\nEnter right child of %d : ",item);
    newnode->right = create();

    return newnode;
}

// Calculate the depth
int depth(struct node *node)
{
    int d = 0;
    while (node != NULL) {
        d++;
        node = node->left;
    }
    return d;
}

// Check if the tree is perfect
bool is_perfect(struct node *root, int d, int level)
{
    // Check if the tree is empty
    if (root == NULL)
        return true;

    // Check the presence of children
    if (root->left == NULL && root->right == NULL)
        return (d == level + 1);

    if (root->left == NULL || root->right == NULL)
        return false;

    return is_perfect(root->left, d+1, level+1) && is_perfect(root->right, d+1, level+1);
}
```

```

    return false;

    return is_perfect(root->left, d, level + 1) &&
           is_perfect(root->right, d, level + 1);
}

// Wrapper function
bool is_Perfect(struct node *root) {
    int d = depth(root);
    return is_perfect(root, d, 0);
}

int main() {

    struct node *root;
    root = NULL;
    root =create();

    if (is_Perfect(root))
        printf("The tree is a perfect binary tree\n");
    else
        printf("The tree is not a perfect binary tree\n");
}

```

Output

```

Enter the value (-1 for No Node) : 1
Enter left child of 1 :
Enter the value (-1 for No Node) : 2
Enter left child of 2 :
Enter the value (-1 for No Node) : 4
Enter left child of 4 :
Enter the value (-1 for No Node) : -1
Enter right child of 4 :
Enter the value (-1 for No Node) : -1
Enter right child of 2 :
Enter the value (-1 for No Node) : 5
Enter left child of 5 :
Enter the value (-1 for No Node) : -1
Enter right child of 5 :
Enter the value (-1 for No Node) : -1
Enter right child of 1 :
Enter the value (-1 for No Node) : 3
Enter left child of 3 :
Enter the value (-1 for No Node) : 6
Enter left child of 6 :
Enter the value (-1 for No Node) : -1
Enter right child of 6 :
Enter the value (-1 for No Node) : -1
Enter right child of 3 :
Enter the value (-1 for No Node) : -1

```

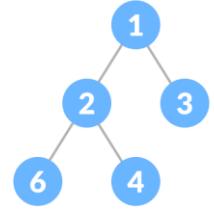
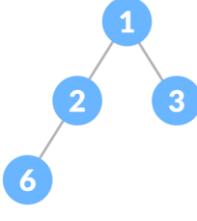
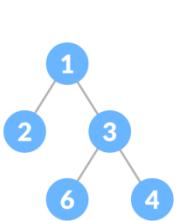
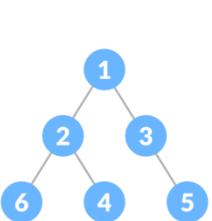
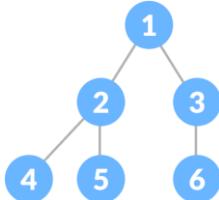
The tree is not a perfect binary tree

Complete Binary Tree

A complete binary tree is a binary tree in which all the levels are completely filled except possibly the lowest one, which is filled from the left.

A complete binary tree is just like a full binary tree, but with two major differences

- o Every level must be completely filled
- o All the leaf elements must lean towards the left.
- o The last leaf element might not have a right sibling i.e. a complete binary tree doesn't have to be a full binary tree.



✗ Full Binary Tree
✗ Complete Binary Tree

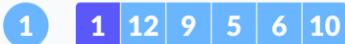
✓ Full Binary Tree
✗ Complete Binary Tree

✗ Full Binary Tree
✓ Complete Binary Tree

✓ Full Binary Tree
✓ Complete Binary Tree

How a Complete Binary Tree is Created?

1. Select the first element of the list to be the root node. (no. of elements on level-I: 1)



2. Put the second element as a left child of the root node and the third element as the right child. (no. of elements on level-II: 2)



3. Put the next two elements as children of the left node of the second level. Again, put the next two elements as children of the right node of the second level (no. of elements on level-III: 4 elements).

4. Keep repeating until you reach the last element.



Program to implement Complete Binary Tree

```

// Checking if a binary tree is a complete binary tree
#include <stdbool.h>
#include <stdio.h>
#include <stdlib.h>

struct node
{
    int data;
    struct node *left,*right;
};

// Node creation
struct node * create()
{
    int item;
    struct node *newnode;
    newnode = (struct node *)malloc(sizeof(struct node));

    printf("\nEnter the value (-1 for No Node) : ");
    scanf("%d",&item);

    if(item == -1){
        return NULL;
    }
    newnode->data = item;
    printf("\nEnter left child of %d : ",item);
    newnode->left = create();

    printf("\nEnter right child of %d : ",item);
    newnode->right = create();

    return newnode;
}

// Count the number of nodes
int countNumNodes(struct node *root)
{
    if (root == NULL)
        return (0);
    return (1 + countNumNodes(root->left) + countNumNodes(root->right));
}

// Check if the tree is a complete binary tree
bool checkComplete(struct node *root, int index, int numberNodes)
{
    // Check if the tree is complete
    if (root == NULL)
        return true;

    if (index >= numberNodes)
        return false;

    return (checkComplete(root->left, 2 * index + 1, numberNodes) &&
checkComplete(root->right, 2 * index + 2, numberNodes));
}

```

```

int main()
{
    struct node *root;
    root = NULL;
    root =create();

    int node_count = countNumNodes(root);
    int index = 0;

    if (checkComplete(root, index, node_count))
        printf("The tree is a complete binary tree\n");
    else
        printf("The tree is not a complete binary tree\n");
}
/*
Output

```

Enter the value (-1 for No Node) : 1

Enter left child of 1 :

Enter the value (-1 for No Node) : 2

Enter left child of 2 :

Enter the value (-1 for No Node) : 4

Enter left child of 4 :

Enter the value (-1 for No Node) : -1

Enter right child of 4 :

Enter the value (-1 for No Node) : -1

Enter right child of 2 :

Enter the value (-1 for No Node) : 5

Enter left child of 5 :

Enter the value (-1 for No Node) : -1

Enter right child of 5 :

Enter the value (-1 for No Node) : -1

Enter right child of 1 :

Enter the value (-1 for No Node) : 3

Enter left child of 3 :

Enter the value (-1 for No Node) : 6

Enter left child of 6 :

Enter the value (-1 for No Node) : -1

Enter right child of 6 :

Enter the value (-1 for No Node) : -1

Enter right child of 3 :

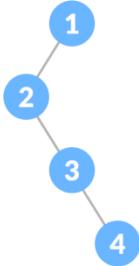
Enter the value (-1 for No Node) : -1

The tree is a complete binary tree

**/*

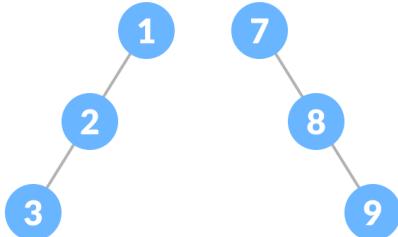
Degenerate or Pathological Tree

A degenerate or pathological tree is the tree having a **single child either left or right**.



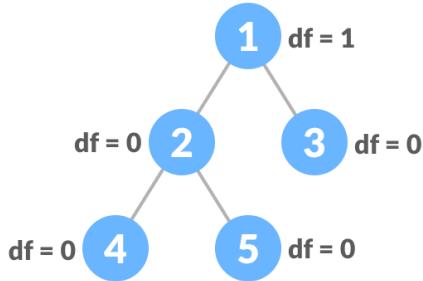
Skewed Binary Tree

A skewed binary tree is a pathological/degenerate tree in which **the tree is either dominated by the left nodes or the right nodes**. Thus, there are two types of skewed binary tree: **left-skewed binary tree** and **right-skewed binary tree**.



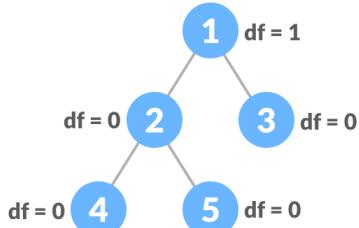
Balanced Binary Tree

It is a type of binary tree in which the difference between the height of the left and the right subtree for each node is either 0 or 1.

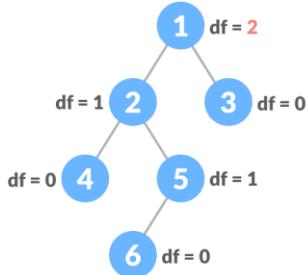


A balanced binary tree, also referred to as a height-balanced binary tree, is defined as a binary tree in which the **height of the left** and **right subtree of any node differs by not more than 1**.

- o difference between the left and the right subtree for any node is not more than one
- o the **left subtree is balanced**
- o the **right subtree is balanced**



Balanced Binary Tree with depth at each level



$$df = |\text{height of left child} - \text{height of right child}|$$

Unbalanced Binary Tree with depth at each level

Program to implement Balanced Binary Tree

```

// Checking if a binary tree is height balanced
#include <stdio.h>
#include <stdlib.h>
#define bool int

// Node creation
struct node
{
    int data;
    struct node *left,*right;
};

// Create a new node
struct node * create()
{
    int item;
    struct node *newnode;
    newnode = (struct node *)malloc(sizeof(struct node));

    printf("\nEnter the value (-1 for No Node) : ");
    scanf("%d",&item);

    if(item == -1){
        return NULL;
    }

    newnode->data = item;
    printf("\nEnter left child of %d : ",item);
    newnode->left = create();

    printf("\nEnter right child of %d : ",item);
    newnode->right = create();

    return newnode;
}

// Check for height balance
bool checkHeightBalance(struct node *root, int *height)
{
    // Check for emptiness
    int leftHeight = 0, rightHeight = 0;
    int l = 0, r = 0;

    if (root == NULL) {
        *height = 0;
        return 1;
    }

    l = checkHeightBalance(root->left, &leftHeight);
    r = checkHeightBalance(root->right, &rightHeight);

    *height = (leftHeight > rightHeight ? leftHeight : rightHeight) + 1;

    if ((leftHeight - rightHeight >= 2) || (rightHeight - leftHeight >= 2))
        return 0;
    else
        return l && r;
}

```

```

int main()
{
    int height = 0;
    struct node *root;
    root = NULL;
    root =create();

    if (checkHeightBalance(root, &height))
        printf("The tree is balanced");
    else
        printf("The tree is not balanced");
}

```

Output

```

Enter the value (-1 for No Node) : 1

Enter left child of 1 :
Enter the value (-1 for No Node) : 2

Enter left child of 2 :
Enter the value (-1 for No Node) : 4

Enter left child of 4 :
Enter the value (-1 for No Node) : -1

Enter right child of 4 :
Enter the value (-1 for No Node) : -1

Enter right child of 2 :
Enter the value (-1 for No Node) : 5

Enter left child of 5 :
Enter the value (-1 for No Node) : -1

Enter right child of 5 :
Enter the value (-1 for No Node) : -1

Enter right child of 1 :
Enter the value (-1 for No Node) : 3

Enter left child of 3 :
Enter the value (-1 for No Node) : -1

Enter right child of 3 :
Enter the value (-1 for No Node) : -1

The tree is balanced

```

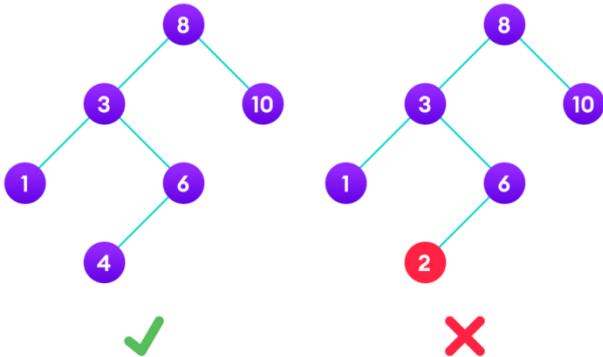
Binary Search Tree

Binary search tree is a data structure that quickly allows us to maintain a sorted list of numbers.

- o It is called a binary tree because each tree node has a maximum of two children.
- o It is called a search tree because it can be used to search for the presence of a number in $O(\log(n))$ time.

The properties that separate a binary search tree from a regular binary tree is

- o All nodes of left subtree are less than the root node
- o All nodes of right subtree are more than(greater) the root node
- o Both subtrees of each node are also BSTs i.e. they have the above two properties



A tree having a right subtree with one value smaller than the root is shown to demonstrate that it is not a valid binary search tree

The binary tree on the right isn't a binary search tree because the right subtree of the node "3" contains a value smaller than it.

There are **two basic operations** that you can perform on a binary search tree:

Search Operation

The algorithm depends on the property of BST that if each left subtree has values below root and each right subtree has values above the root.

If the value is below the root, we can say for sure that the value is not in the right subtree; we need to only search in the left subtree and if the value is above the root, we can say for sure that the value is not in the left subtree; we need to only search in the right subtree.

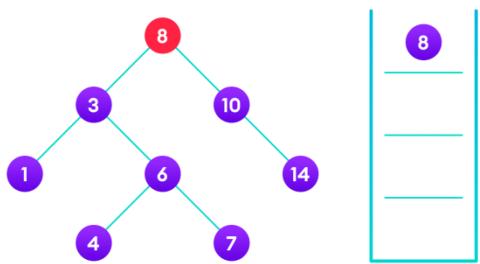
Algorithm:

```

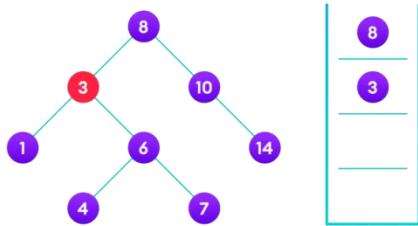
If root == NULL
    return NULL;
If number == root->data
    return root->data;
If number < root->data
    return search(root->left)
If number > root->data
    return search(root->right)

```

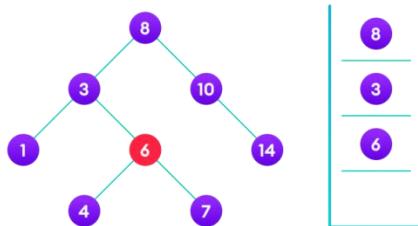
Let us try to visualize this with a diagram.



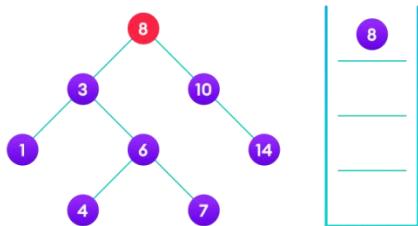
4 is not found so, traverse through the left subtree of 8



4 is not found so, traverse through the right subtree of 3



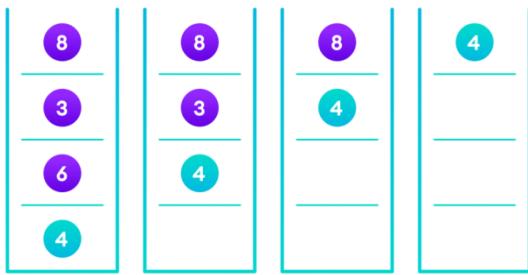
4 is not found so, traverse through the left subtree of 6



4 is found

If the value is found, we return the value so that it gets propagated in each recursion step as shown in the image below.

If you might have noticed, we have called return search (struct node*) four times. When we return either the new node or NULL, the value gets returned again and again until search(root) returns the final result.



If the value is found in any of the subtrees, it is propagated up so that in the end it is returned, otherwise null is returned

If the value is not found, we eventually reach the left or right child of a leaf node which is NULL and it gets propagated and returned.

Insert Operation

Inserting a value in the correct position is similar to searching because we try to maintain the rule that the left subtree is lesser than root and the right subtree is larger than root.

We keep going to either right subtree or left subtree depending on the value and when we reach a point left or right subtree is null, we put the new node there.

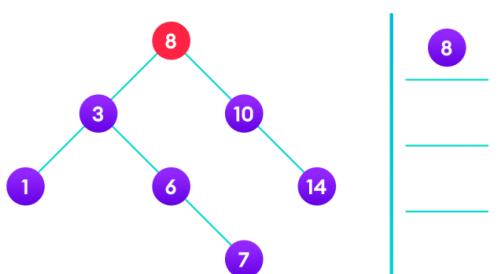
Algorithm:

```

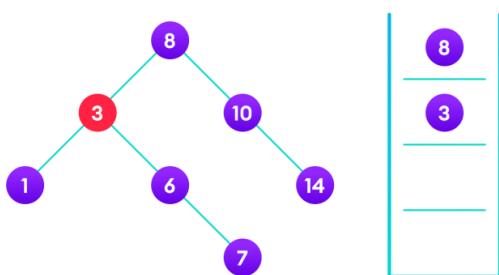
If node == NULL
    return createNode(data)
if (data < node->data)
    node->left = insert(node->left, data);
else if (data > node->data)
    node->right = insert(node->right, data);
return node;

```

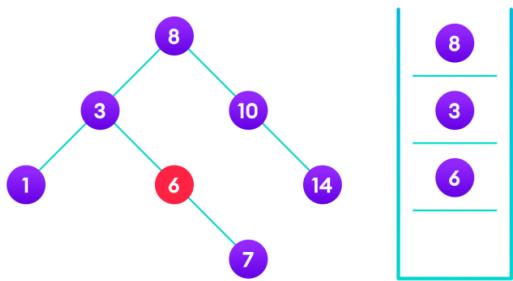
Let's try to visualize how we add a number to an existing BST.



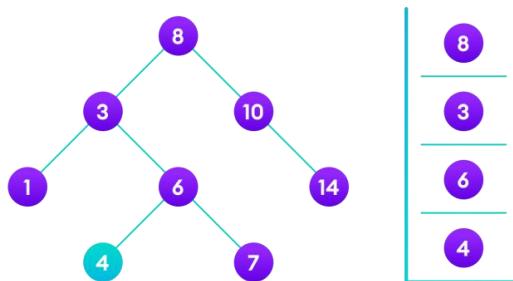
$4 < 8$ so, transverse through the left child of 8



$4 > 3$ so, transverse through the right child of 8



$4 < 6$ so, transverse through the left child of 6



Insert 4 as a left child of 6

We have attached the node but we still have to exit from the function without doing any damage to the rest of the tree. This is where the **return node**; at the end comes in handy. In the case of **NULL**, the newly created node is returned and attached to the parent node, otherwise the same node is returned without any change as we go up until we return to the root.

This makes sure that as we move back up the tree, the other node connections aren't changed.

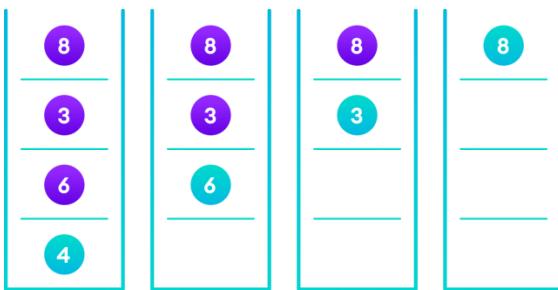


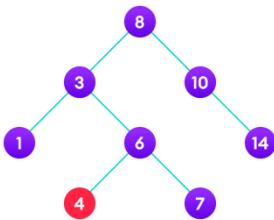
Image showing the importance of returning the root element at the end so that the elements don't lose their position during the upward recursion step.

Deletion Operation

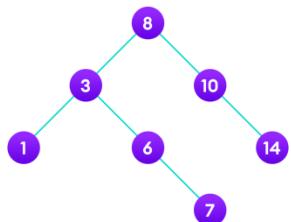
There are **three cases** for deleting a node from a binary search tree.

Case I

In the first case, **the node to be deleted is the leaf node**. In such a case, simply delete the node from the tree.



4 is to be deleted

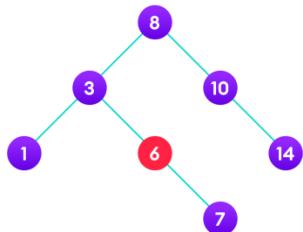


Delete the node

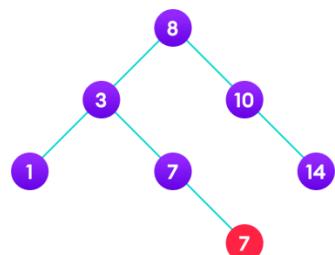
Case II

In the second case, **the node to be deleted lies has a single child node**. In such a case follow the steps below:

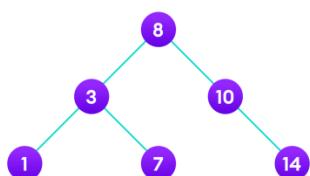
- o Replace that node with its child node.
- o Remove the child node from its original position.



6 is to be deleted



Copy the value of its child to the node and delete the child

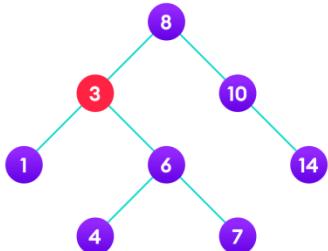


Final tree

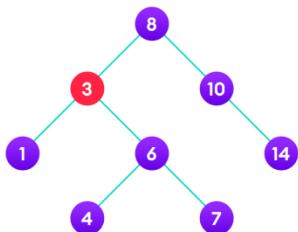
Case III

In the third case, **the node to be deleted has two children**. In such a case follow the steps below:

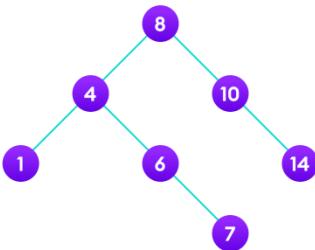
- Get the in-order successor of that node.
- Replace the node with the in-order successor.
- Remove the in-order successor from its original position.



3 is to be deleted



Copy the value of the in-order successor (4) to the node



Delete the in-order successor

Binary Search Tree Applications

- In multilevel indexing in the database
- For dynamic sorting
- For managing virtual memory areas in Unix kernel

Program to implementation of Binary Search Tree operations

```

// Binary Search Tree operations
#include <stdio.h>
#include <stdlib.h>

struct node
{
    int key;
    struct node *left, *right;
};

// Create a node
struct node *newNode(int item)
{
    struct node *temp = (struct node *)malloc(sizeof(struct node));
    temp->key = item;
    temp->left = temp->right = NULL;
    return temp;
}

// Inorder Traversal
void inorder(struct node *root)
{
    if (root != NULL) {
        // Traverse left
        inorder(root->left);
        // Traverse root
        printf("%d -> ", root->key);
        // Traverse right
        inorder(root->right);
    }
}

// Insert a node
struct node *insert(struct node *node, int key)
{
    // Return a new node if the tree is empty
    if (node == NULL) return newNode(key);

    // Traverse to the right place and insert the node
    if (key < node->key)
        node->left = insert(node->left, key);
    else
        node->right = insert(node->right, key);

    return node;
}

// Find the inorder successor
struct node *minValueNode(struct node *node)
{
    struct node *current = node;

    // Find the leftmost leaf
    while (current && current->left != NULL)
        current = current->left;

    return current;
}

```

```

// Deleting a node
struct node *deleteNode(struct node *root, int key)
{
    // Return if the tree is empty
    if (root == NULL) return root;

    // Find the node to be deleted
    if (key < root->key)
        root->left = deleteNode(root->left, key);
    else if (key > root->key)
        root->right = deleteNode(root->right, key);

    else {
        // If the node is with only one child or no child
        if (root->left == NULL) {
            struct node *temp = root->right;
            free(root);
            return temp;
        } else if (root->right == NULL) {
            struct node *temp = root->left;
            free(root);
            return temp;
        }
        // If the node has two children
        struct node *temp = minValueNode(root->right);
        // Place the inorder successor in position of the node to be deleted
        root->key = temp->key;
        // Delete the inorder successor
        root->right = deleteNode(root->right, temp->key);
    }
    return root;
}

int main()
{
    struct node *root = NULL;
    root = insert(root, 8);
    root = insert(root, 3);
    root = insert(root, 1);
    root = insert(root, 6);
    root = insert(root, 7);
    root = insert(root, 10);
    root = insert(root, 14);
    root = insert(root, 4);

    printf("Inorder traversal: ");
    inorder(root);

    printf("\nAfter deleting 10\n");
    root = deleteNode(root, 10);
    printf("Inorder traversal: ");
    inorder(root);
}

```

Output

```

Inorder traversal: 1 -> 3 -> 4 -> 6 -> 7 -> 8 -> 10 -> 14 ->
After deleting 10
Inorder traversal: 1 -> 3 -> 4 -> 6 -> 7 -> 8 -> 14 ->

```

Watch Videos: -[Binary Search Tree](#)[Binary Search Trees \(BST\) - Insertion and Deletion](#)[Construct Binary Search Tree\(BST\) from Preorder](#)[Construct a Binary Search Tree\(BST\) from given Postorder traversal](#)[Binary Search Iterative Method](#)

AVL Tree

AVL tree is a self-balancing binary search tree in which each node maintains extra information called a balance factor whose value is either -1, 0 or +1.

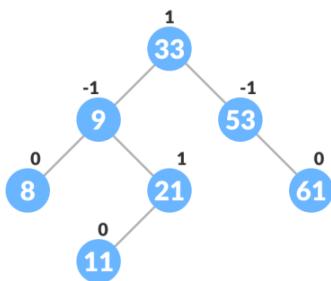
AVL tree got its name after its inventor Georgy Adelson-Velsky and Landis.

Balance Factor

Balance factor of a node in an AVL tree is the difference between the height of the left subtree and that of the right subtree of that node.

Balance Factor = (Height of Left Subtree - Height of Right Subtree) or (Height of Right Subtree - Height of Left Subtree)

The self-balancing property of an avl tree is maintained by the balance factor. The value of balance factor should always be -1, 0 or +1.



Operations on AVL tree

Due to the fact that, AVL tree is also a **binary search tree** therefore, all the operations are performed in the same way as they are performed in a binary search tree.

Searching and traversing do not lead to the violation in property of AVL tree. However, insertion and deletion are the operations which can violate this property and therefore, they need to be revisited.

Insertion

Insertion in AVL tree is performed in the same way as it is performed in a binary search tree. However, it may lead to violation in the AVL tree property and therefore the tree may need balancing. The tree can be balanced by applying rotations.

Deletion

Deletion can also be performed in the same way as it is performed in a binary search tree. Deletion may also disturb the balance of the tree therefore; various types of rotations are used to rebalance the tree.

AVL Rotations

We perform rotation in AVL tree only in case if Balance Factor is other than -1, 0, and 1. There are basically four types of rotations which are as follows:

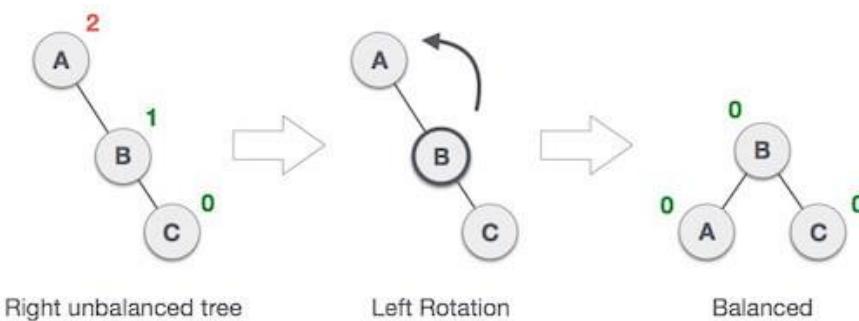
- o **LL rotation** : Inserted node is in the left subtree of left subtree of A
- o **RR rotation** : Inserted node is in the right subtree of right subtree of A
- o **LR rotation** : Inserted node is in the right subtree of left subtree of A
- o **RL rotation** : Inserted node is in the left subtree of right subtree of A

Where node A is the node whose balance Factor is other than -1, 0, 1.

The first two rotations LL and RR are single rotations and the next two rotations LR and RL are double rotations. For a tree to be unbalanced, minimum height must be at least 2, Let us understand each rotation

RR Rotation

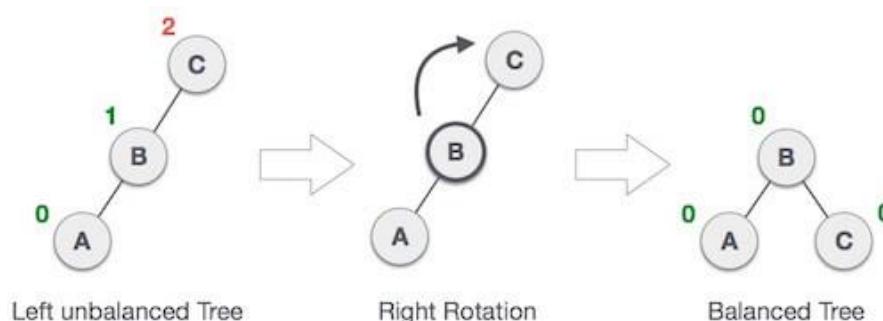
When BST becomes unbalanced, due to a node is inserted into the right subtree of the right subtree of A, then we perform RR rotation, RR rotation is an anticlockwise rotation, which is applied on the edge below a node having balance factor -2



In above example, node A has balance factor -2 because a node C is inserted in the right subtree of A right subtree. We perform the RR rotation on the edge below A.

LL Rotation

When BST becomes unbalanced, due to a node is inserted into the left subtree of the left subtree of C, then we perform LL rotation, LL rotation is clockwise rotation, which is applied on the edge below a node having balance factor 2.



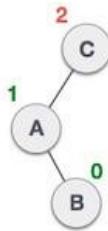
In above example, node C has balance factor 2 because a node A is inserted in the left subtree of C left subtree. We perform the LL rotation on the edge below A.

LR Rotation

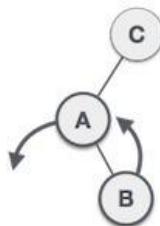
Double rotations are bit tougher than single rotation which has already explained above. LR rotation = RR rotation + LL rotation, i.e., first RR rotation is performed on subtree and then LL rotation is performed on full tree, by full tree we mean the first node from the path of inserted node whose balance factor is other than -1, 0, or 1.

Let us understand each and every step very clearly:

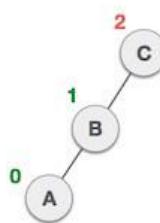
1. A node B has been inserted into the right subtree of A the left subtree of C, because of which C has become an unbalanced node having balance factor 2. This case is L R rotation where: Inserted node is in the right subtree of left subtree of C



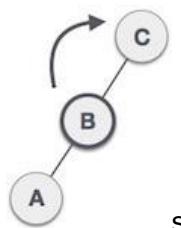
2. As LR rotation = RR + LL rotation, hence RR (anticlockwise) on subtree rooted at A is performed first. By doing RR rotation, node **A**, has become the left subtree of **B**.



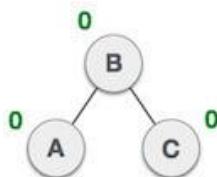
3. After performing RR rotation, node C is still unbalanced, i.e., having balance factor 2, as inserted node A is in the left of left of **C**



4. Now we perform LL clockwise rotation on full tree, i.e. on node C. node **C** has now become the right subtree of node B, A is left subtree of B



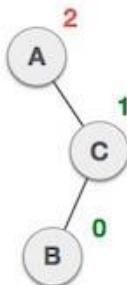
5. Balance factor of each node is now either -1, 0, or 1, i.e. BST is balanced now.



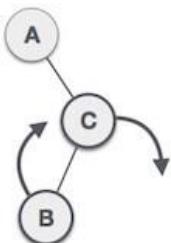
RL Rotation

As already discussed, that double rotations are bit tougher than single rotation which has already explained above. R L rotation = LL rotation + RR rotation, i.e., first LL rotation is performed on subtree and then RR rotation is performed on full tree, by full tree we mean the first node from the path of inserted node whose balance factor is other than -1, 0, or 1.

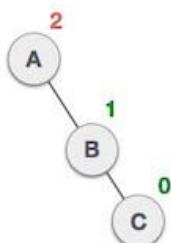
1. A node **B** has been inserted into the left subtree of **C** the right subtree of **A**, because of which **A** has become an unbalanced node having balance factor -2. This case is RL rotation where: Inserted node is in the left subtree of right subtree of A



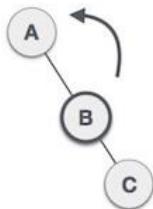
2. As RL rotation = LL rotation + RR rotation, hence, LL (clockwise) on subtree rooted at **C** is performed first. By doing RR rotation, node **C** has become the right subtree of **B**.



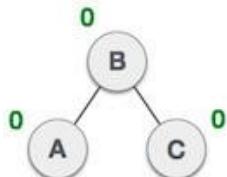
3. After performing LL rotation, node **A** is still unbalanced, i.e. having balance factor -2, which is because of the right-subtree of the right-subtree node A.



4. Now we perform RR rotation (anticlockwise rotation) on full tree, i.e. on node A. node C has now become the right subtree of node B, and node A has become the left subtree of B.



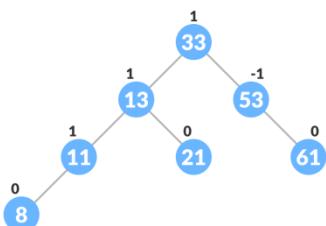
5. Balance factor of each node is now either -1, 0, or 1, i.e., BST is balanced now.



Algorithm to insert a newNode

A newNode is always inserted as a leaf node with balance factor equal to 0.

1. Let the initial tree be:

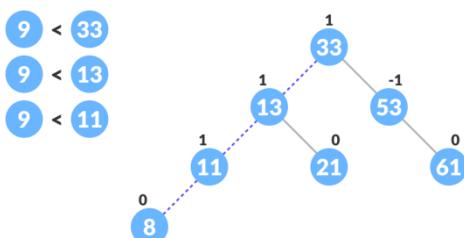


Let the node to be inserted be: **New node**



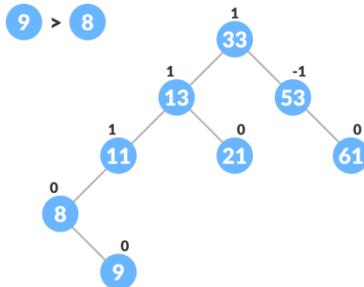
2. Go to the appropriate leaf node to insert a **newNode** using the following recursive steps. Compare **newKey** with **rootKey** of the current tree.

1. If **newKey < rootKey**, call insertion algorithm on **the left subtree of the current node until the leaf node is reached.**
2. Else if **newKey > rootKey**, call insertion algorithm on **the right subtree of current node until the leaf node is reached.**
3. Else, return **leafNode**.

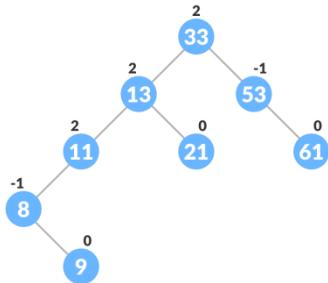


3. Compare `leafKey` obtained from the above steps with `newKey`

1. If `newKey` < `leafKey`, make `newNode` as the `leftChild` of `leafNode`.
2. Else, make `newNode` as `rightChild` of `leafNode`.

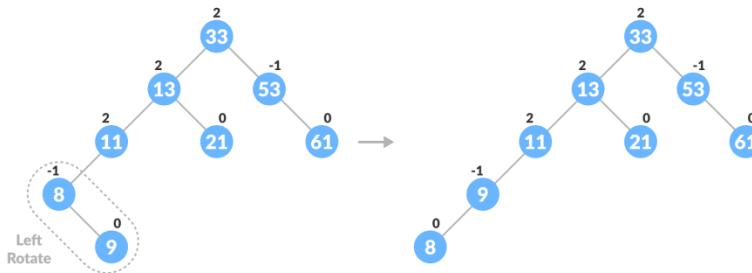


4. Update balanceFactor of the nodes

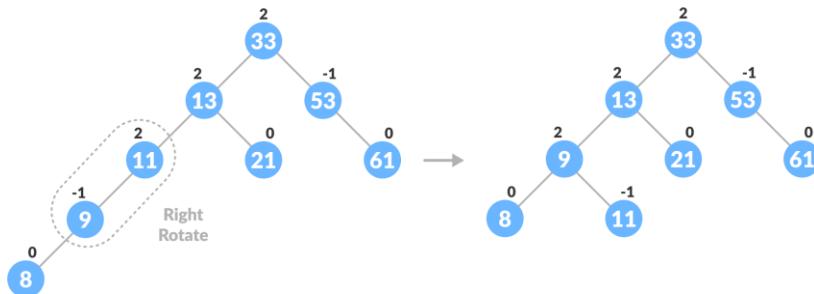


5. If the nodes are unbalanced, then rebalance the node.

1. If `balanceFactor` > 1, it means the height of the left subtree is greater than that of the right subtree. So, do a right rotation or left-right rotation
 - a. If `newNodeKey` < `leftChildKey` do right rotation.
 - b. Else, do left-right rotation.



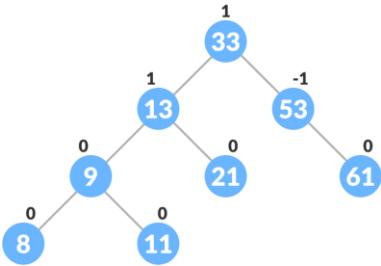
Balancing the tree with rotation



2. If `balanceFactor` < -1, it means the height of the right subtree is greater than that of the left subtree. So, do right rotation or right-left rotation

- If `newNodeKey` > `rightChildKey` do left rotation.
- Else, do right-left rotation

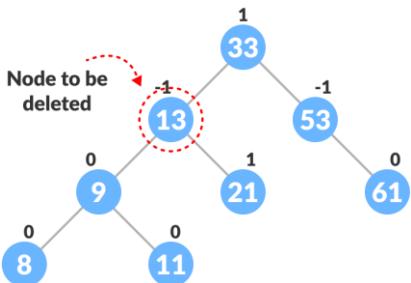
6. The final tree is



Algorithm to Delete a node

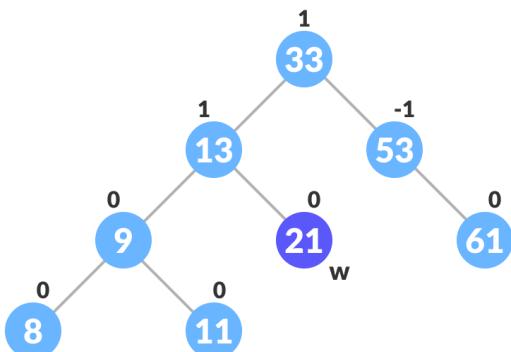
A node is always deleted as a leaf node. After deleting a node, the balance factors of the nodes get changed. In order to rebalance the balance factor, suitable rotations are performed.

1. Locate `nodeToDelete` (recursion is used to find `nodeToDelete` in the code used below).

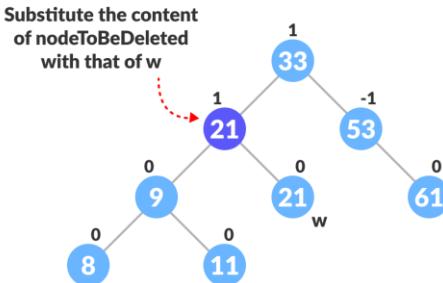


2. There are three cases for deleting a node:

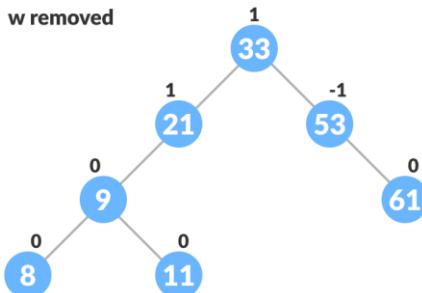
- If `nodeToDelete` is the leaf node (ie. does not have any child), then remove `nodeToDelete`.
- If `nodeToDelete` has one child, then substitute the contents of `nodeToDelete` with that of the child. Remove the child.
- If `nodeToDelete` has two children, find the inorder successor w of `nodeToDelete` (ie. node with a minimum value of key in the right subtree).



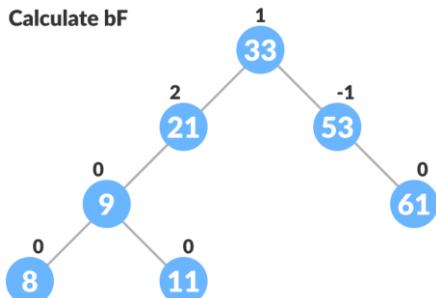
Substitute the contents of `nodeToDelete` with that of `w`.



Remove the leaf node `w`.

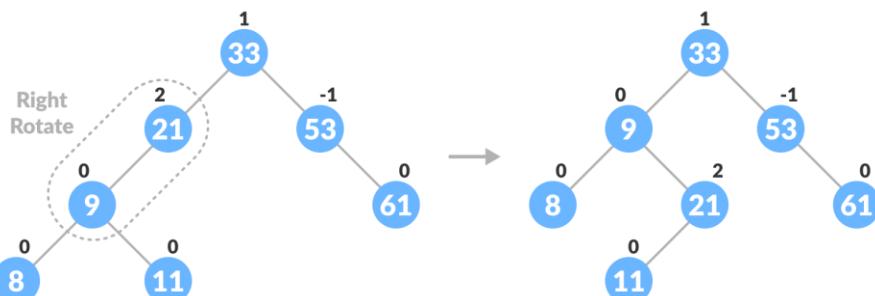


3. Update balanceFactor of the nodes.



4. Rebalance the tree if the balance factor of any of the nodes is not equal to -1, 0 or 1.

- If `balanceFactor` of `currentNode` > 1,
 - If `balanceFactor` of `leftChild` >= 0, do right rotation.

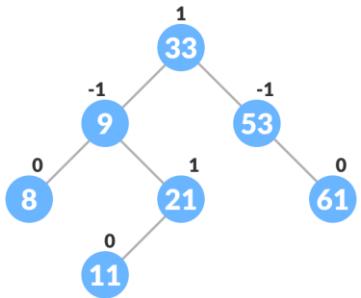


- Else do left-right rotation.

- If `balanceFactor` of `currentNode` < -1,
 - If `balanceFactor` of `rightChild` <= 0, do left rotation.

b. Else do right-left rotation.

5. The final tree is:



Program to implement AVL Tree

```

// AVL tree in C
#include <stdio.h>
#include <stdlib.h>

// Create Node
struct Node {
    int key;
    struct Node *left;
    struct Node *right;
    int height;
};

int max(int a, int b);

// Calculate height
int height(struct Node *N) {
    if (N == NULL)
        return 0;
    return N->height;
}

int max(int a, int b) {
    return (a > b) ? a : b;
}

// Create a node
struct Node *newNode(int key) {
    struct Node *node = (struct Node *)
        malloc(sizeof(struct Node));
    node->key = key;
    node->left = NULL;
    node->right = NULL;
    node->height = 1;
    return (node);
}

// Right rotate
struct Node *rightRotate(struct Node *y) {
    struct Node *x = y->left;
    struct Node *T2 = x->right;

    x->right = y;
    y->left = T2;

    y->height = max(height(y->left), height(y->right)) + 1;
    x->height = max(height(x->left), height(x->right)) + 1;

    return x;
}

// Left rotate
struct Node *leftRotate(struct Node *x) {
    struct Node *y = x->right;
    struct Node *T2 = y->left;

    y->left = x;
    x->right = T2;
}

```

```

x->height = max(height(x->left), height(x->right)) + 1;
y->height = max(height(y->left), height(y->right)) + 1;

    return y;
}

// Get the balance factor
int getBalance(struct Node *N) {
    if (N == NULL)
        return 0;
    return height(N->left) - height(N->right);
}

// Insert node
struct Node *insertNode(struct Node *node, int key) {
    // Find the correct position to insertNode the node and insertNode it
    if (node == NULL)
        return (newNode(key));

    if (key < node->key)
        node->left = insertNode(node->left, key);
    else if (key > node->key)
        node->right = insertNode(node->right, key);
    else
        return node;

    // Update the balance factor of each node and
    // Balance the tree
    node->height = 1 + max(height(node->left),
                           height(node->right));

    int balance = getBalance(node);
    if (balance > 1 && key < node->left->key)
        return rightRotate(node);

    if (balance < -1 && key > node->right->key)
        return leftRotate(node);

    if (balance > 1 && key > node->left->key) {
        node->left = leftRotate(node->left);
        return rightRotate(node);
    }

    if (balance < -1 && key < node->right->key) {
        node->right = rightRotate(node->right);
        return leftRotate(node);
    }

    return node;
}

struct Node *minValueNode(struct Node *node) {
    struct Node *current = node;

    while (current->left != NULL)
        current = current->left;

    return current;
}

```

```

// Delete a nodes
struct Node *deleteNode(struct Node *root, int key) {
    // Find the node and delete it
    if (root == NULL)
        return root;

    if (key < root->key)
        root->left = deleteNode(root->left, key);

    else if (key > root->key)
        root->right = deleteNode(root->right, key);

    else {
        if ((root->left == NULL) || (root->right == NULL)) {
            struct Node *temp = root->left ? root->left : root->right;

            if (temp == NULL) {
                temp = root;
                root = NULL;
            } else
                *root = *temp;
                free(temp);
        } else {
            struct Node *temp = minValueNode(root->right);

            root->key = temp->key;

            root->right = deleteNode(root->right, temp->key);
        }
    }

    if (root == NULL)
        return root;

    // Update the balance factor of each node and
    // balance the tree
    root->height = 1 + max(height(root->left),
                           height(root->right));

    int balance = getBalance(root);
    if (balance > 1 && getBalance(root->left) >= 0)
        return rightRotate(root);

    if (balance > 1 && getBalance(root->left) < 0) {
        root->left = leftRotate(root->left);
        return rightRotate(root);
    }

    if (balance < -1 && getBalance(root->right) <= 0)
        return leftRotate(root);

    if (balance < -1 && getBalance(root->right) > 0) {
        root->right = rightRotate(root->right);
        return leftRotate(root);
    }

    return root;
}

```

```

// Print the tree
void printPreOrder(struct Node *root) {
    if (root != NULL) {
        printf("%d ", root->key);
        printPreOrder(root->left);
        printPreOrder(root->right);
    }
}

int main() {
    struct Node *root = NULL;

    root = insertNode(root, 2);
    root = insertNode(root, 1);
    root = insertNode(root, 7);
    root = insertNode(root, 4);
    root = insertNode(root, 5);
    root = insertNode(root, 3);
    root = insertNode(root, 8);

    printPreOrder(root);

    root = deleteNode(root, 3);

    printf("\nAfter deletion: ");
    printPreOrder(root);

    return 0;
}

```

Output

4 2 1 3 7 5 8
After deletion: 4 2 1 7 5 8

Watch Video: -

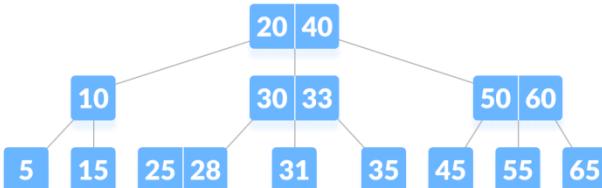
[AVL Tree - Insertion and Rotations](#)
[Construct AVL tree Insertion](#)
[AVL tree - Insertion, Rotations\(LL, RR, LR, RL\) with example](#)
[AVL tree Insertion | with solved example](#)
[AVL Tree Deletion in Data structures](#)

5. Tree Based DSA(II)

B Tree

B-tree is a special type of self-balancing search tree in which each node can contain more than one key and can have more than two children. It is a generalized form of the binary search tree.

It is also known as a height-balanced m-way tree.



Why do you need a B-tree data structure?

The need for B-tree arose with the rise in the need for lesser time in accessing the physical storage media like a hard disk. The secondary storage devices are slower with a larger capacity. There was a need for such types of data structures that minimize the disk accesses.

Other data structures such as a binary search tree, avl tree, red-black tree, etc can store only one key in one node. If you have to store a large number of keys, then the height of such trees becomes very large and the access time increases.

However, B-tree can store many keys in a single node and can have multiple child nodes. This decreases the height significantly allowing faster disk accesses.

B-tree Properties

- For each node x , the keys are stored in increasing order.
- In each node, there is a boolean value $x.\text{leaf}$ which is true if x is a leaf.
- If n is the order of the tree, each internal node can contain at most $n - 1$ keys along with a pointer to each child.
- Each node except root can have at most n children and at least $n/2$ children.
- All leaves have the same depth (i.e. height- h of the tree).
- The root has at least 2 children and contains a minimum of 1 key.
- If $n \geq 1$, then for any n -key B-tree of height h and minimum degree $t \geq 2$, $h \geq \log_t(n+1)/2$.

Operations on a B-tree

Searching an element in a B-tree

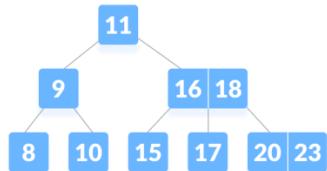
Searching for an element in a B-tree is the generalized form of searching an element in a Binary Search Tree. The following steps are followed.

- Starting from the root node, compare k with the first key of the node.
If $k =$ the first key of the node, return the node and the index.
- If $k.\text{leaf} =$ true, return NULL (i.e. not found).
- If $k <$ the first key of the root node, search the left child of this key recursively.
- If there is more than one key in the current node and $k >$ the first key, compare k with the next key in the node.
If $k <$ next key, search the left child of this key (ie. k lies in between the first and the second keys).
Else, search the right child of the key.

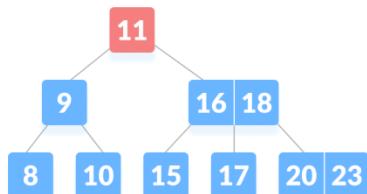
- o Repeat steps 1 to 4 until the leaf is reached.

Searching Example

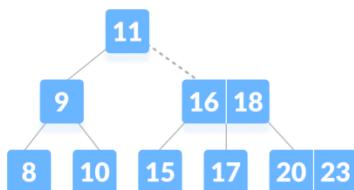
- Let us search key $k = 17$ in the tree below of degree 3.



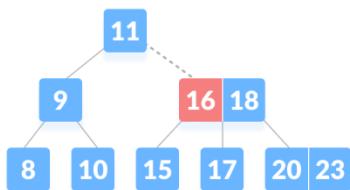
- k is not found in the root so, compare it with the root key.



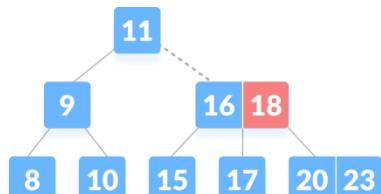
- Since $k > 11$, go to the right child of the root node.



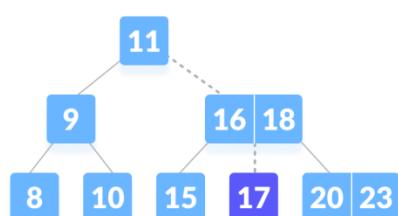
- Compare k with 16. Since $k > 16$, compare k with the next key 18.



- Since $k < 18$, k lies between 16 and 18. Search in the right child of 16 or the left child of 18.



- k is found.



Algorithm for Searching an Element

```
BtreeSearch(x, k)
i = 1
while i ≤ n[x] and k ≥ keyi[x] // n[x] means number of keys in x node
    do i = i + 1
if i = n[x] and k = keyi[x]
    then return (x, i)
if leaf [x]
    then return NIL
else
    return BtreeSearch(ci[x], k)
```

Program to implement B Tree

```

// Searching a key on a B-tree
#include <stdio.h>
#include <stdlib.h>

#define MAX 3
#define MIN 2

struct BTreenode
{
    int val[MAX + 1], count;
    struct BTreenode *link[MAX + 1];
};

struct BTreenode *root;

// Create a node
struct BTreenode *createNode(int val, struct BTreenode *child)
{
    struct BTreenode *newNode;
    newNode = (struct BTreenode *)malloc(sizeof(struct BTreenode));
    newNode->val[1] = val;
    newNode->count = 1;
    newNode->link[0] = root;
    newNode->link[1] = child;
    return newNode;
}

// Insert node
void insertNode(int val, int pos, struct BTreenode *node,
                struct BTreenode *child)
{
    int j = node->count;
    while (j > pos) {
        node->val[j + 1] = node->val[j];
        node->link[j + 1] = node->link[j];
        j--;
    }
    node->val[j + 1] = val;
    node->link[j + 1] = child;
    node->count++;
}

// Split node
void splitNode(int val, int *pval, int pos, struct BTreenode *node,
               struct BTreenode *child, struct BTreenode **newNode)
{
    int median, j;

    if (pos > MIN)
        median = MIN + 1;
    else
        median = MIN;

    *newNode = (struct BTreenode *)malloc(sizeof(struct BTreenode));
    j = median + 1;
    while (j <= MAX) {
        (*newNode)->val[j - median] = node->val[j];
        (*newNode)->link[j - median] = node->link[j];
    }
}

```

```

        j++;
    }
node->count = median;
(*newNode)->count = MAX - median;

if (pos <= MIN) {
    insertNode(val, pos, node, child);
} else {
    insertNode(val, pos - median, *newNode, child);
}
*pval = node->val[node->count];
(*newNode)->link[0] = node->link[node->count];
node->count--;
}
// Set the value
int setValue(int val, int *pval, struct BTreenode *node, struct BTreenode **child)
{
    int pos;
    if (!node) {
        *pval = val;
        *child = NULL;
        return 1;
    }

    if (val < node->val[1]) {
        pos = 0;
    } else {
        for (pos = node->count;
              (val < node->val[pos] && pos > 1); pos--)
        ;
        if (val == node->val[pos]) {
            printf("Duplicates are not permitted\n");
            return 0;
        }
    }
    if (setValue(val, pval, node->link[pos], child)) {
        if (node->count < MAX) {
            insertNode(*pval, pos, node, *child);
        } else {
            splitNode(*pval, pval, pos, node, *child, child);
            return 1;
        }
    }
    return 0;
}

// Insert the value
void insert(int val)
{
    int flag, i;
    struct BTreenode *child;

    flag = setValue(val, &i, root, &child);
    if (flag)
        root = createNode(i, child);
}

// Search node
void search(int val, int *pos, struct BTreenode *myNode)
{

```

```

if (!myNode) {
    return;
}

if (val < myNode->val[1]) {
    *pos = 0;
} else {
    for (*pos = myNode->count;
        (val < myNode->val[*pos] && *pos > 1); (*pos)--)
    ;
    if (val == myNode->val[*pos]) {
        printf("The element %d is found", val);
        return;
    }
}
search(val, pos, myNode->link[*pos]);

return;
}
// Traverse then nodes
void traversal(struct BTreenode *myNode)
{
    int i;
    if (myNode) {
        for (i = 0; i < myNode->count; i++) {
            traversal(myNode->link[i]);
            printf("%d ", myNode->val[i + 1]);
        }
        traversal(myNode->link[i]);
    }
}

int main() {
    int val, ch, item, size;
    printf("Enter the Number of element : ");
    scanf("%d", &size);
    // 9,10,11,15,16,17,18,20,23
    for (int i=0; i<= size-1; i++) {
        printf("Enter the element : ");
        scanf("%d", &item);
        insert(item);
    }
    printf("\nThe tree is : ");
    traversal(root);
    printf("\nEnter the item for search : ");
    scanf("%d", &item);
    search(item, &ch, root);
}

```

Output

The tree is : 8 9 10 11 15 16 17 18 20 23
 Enter the item for search : 10
 The element 10 is found

Watch Video:-

[Introduction to B-Trees](#)

Insertion in a B-tree

Inserting an element on a B-tree consists of two events: searching the appropriate node to insert the element and splitting the node if required. Insertion operation always takes place in the bottom-up approach.

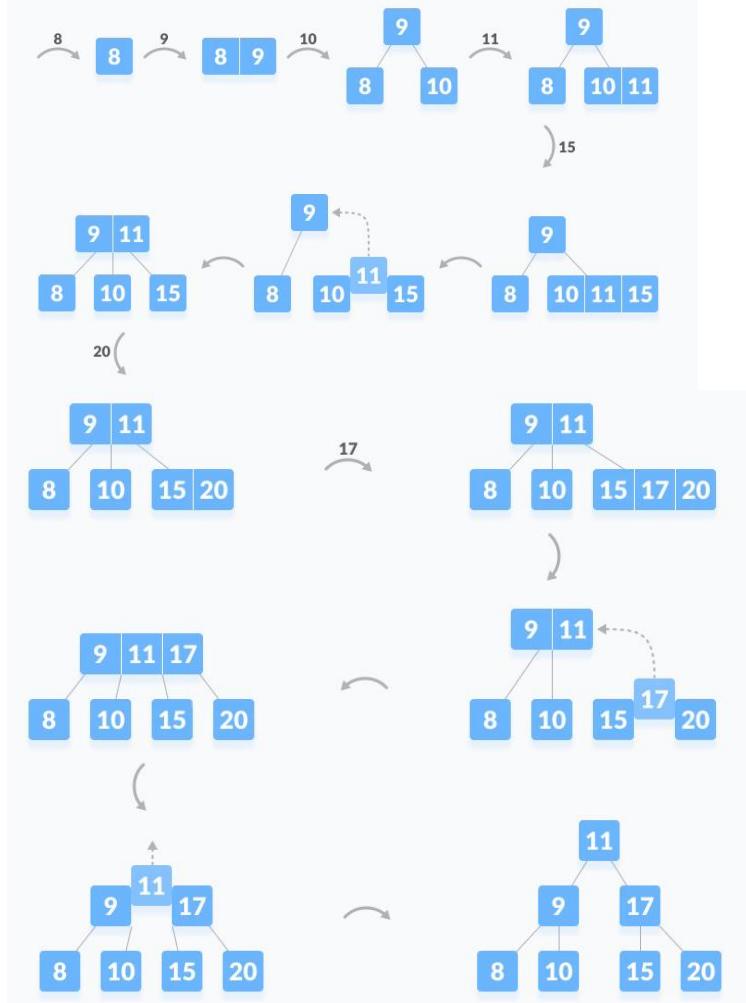
Let us understand these events below.

Insertion Operation

1. If the tree is empty, allocate a root node and insert the key.
2. Update the allowed number of keys in the node.
3. Search the appropriate node for insertion.
4. If the node is full, follow the steps below.
5. Insert the elements in increasing order.
6. Now, there are elements greater than its limit. So, split at the median.
7. Push the median key upwards and make the left keys as a left child and the right keys as a right child.
8. If the node is not full, follow the steps below.
9. Insert the node in increasing order.

Insertion Example

The elements to be inserted are 8, 9, 10, 11, 15, 20, 17.



Program to implement B-tree Insertion

```

// B-Tree - Inserting
#include <stdio.h>
#include <stdlib.h>

#define MAX 3
#define MIN 2

struct btreeNode {
    int item[MAX + 1], count;
    struct btreeNode *link[MAX + 1];
};

struct btreeNode *root;

// Node creation
struct btreeNode *createNode(int item, struct btreeNode *child) {
    struct btreeNode *newNode;
    newNode = (struct btreeNode *)malloc(sizeof(struct btreeNode));
    newNode->item[1] = item;
    newNode->count = 1;
    newNode->link[0] = root;
    newNode->link[1] = child;
    return newNode;
}

// Insert
void insertValue(int item, int pos, struct btreeNode *node,
                 struct btreeNode *child) {
    int j = node->count;
    while (j > pos) {
        node->item[j + 1] = node->item[j];
        node->link[j + 1] = node->link[j];
        j--;
    }
    node->item[j + 1] = item;
    node->link[j + 1] = child;
    node->count++;
}

// Split node
void splitNode(int item, int *pval, int pos, struct btreeNode *node,
               struct btreeNode *child, struct btreeNode **newNode) {
    int median, j;

    if (pos > MIN)
        median = MIN + 1;
    else
        median = MIN;

    *newNode = (struct btreeNode *)malloc(sizeof(struct btreeNode));
    j = median + 1;
    while (j <= MAX) {
        (*newNode)->item[j - median] = node->item[j];
        (*newNode)->link[j - median] = node->link[j];
        j++;
    }
    node->count = median;
    (*newNode)->count = MAX - median;
}

```

```

if (pos <= MIN) {
    insertValue(item, pos, node, child);
} else {
    insertValue(item, pos - median, *newNode, child);
}
*pval = node->item[node->count];
(*newNode)->link[0] = node->link[node->count];
node->count--;
}

// Set the value of node
int setNodeValue(int item, int *pval,
                 struct btreeNode *node, struct btreeNode **child) {
int pos;
if (!node) {
    *pval = item;
    *child = NULL;
    return 1;
}

if (item < node->item[1]) {
    pos = 0;
} else {
    for (pos = node->count;
          (item < node->item[pos] && pos > 1); pos--)
    ;
    if (item == node->item[pos]) {
        printf("Duplicates not allowed\n");
        return 0;
    }
}
if (setNodeValue(item, pval, node->link[pos], child)) {
    if (node->count < MAX) {
        insertValue(*pval, pos, node, *child);
    } else {
        splitNode(*pval, pval, pos, node, *child, child);
        return 1;
    }
}
return 0;
}

// Insert the value
void insertion(int item) {
    int flag, i;
    struct btreeNode *child;

    flag = setNodeValue(item, &i, root, &child);
    if (flag)
        root = createNode(i, child);
}

// Copy the successor
void copySuccessor(struct btreeNode *myNode, int pos) {
    struct btreeNode *dummy;
    dummy = myNode->link[pos];

    for (; dummy->link[0] != NULL;)
        dummy = dummy->link[0];
}

```

```

myNode->item[pos] = dummy->item[1];
}

// Do rightshift
void rightShift(struct btreeNode *myNode, int pos) {
    struct btreeNode *x = myNode->link[pos];
    int j = x->count;

    while (j > 0) {
        x->item[j + 1] = x->item[j];
        x->link[j + 1] = x->link[j];
    }
    x->item[1] = myNode->item[pos];
    x->link[1] = x->link[0];
    x->count++;

    x = myNode->link[pos - 1];
    myNode->item[pos] = x->item[x->count];
    myNode->link[pos] = x->link[x->count];
    x->count--;
    return;
}

// Do leftshift
void leftShift(struct btreeNode *myNode, int pos) {
    int j = 1;
    struct btreeNode *x = myNode->link[pos - 1];

    x->count++;
    x->item[x->count] = myNode->item[pos];
    x->link[x->count] = myNode->link[pos]->link[0];

    x = myNode->link[pos];
    myNode->item[pos] = x->item[1];
    x->link[0] = x->link[1];
    x->count--;

    while (j <= x->count) {
        x->item[j] = x->item[j + 1];
        x->link[j] = x->link[j + 1];
        j++;
    }
    return;
}

// Merge the nodes
void mergeNodes(struct btreeNode *myNode, int pos) {
    int j = 1;
    struct btreeNode *x1 = myNode->link[pos], *x2 = myNode->link[pos - 1];

    x2->count++;
    x2->item[x2->count] = myNode->item[pos];
    x2->link[x2->count] = myNode->link[0];

    while (j <= x1->count) {
        x2->count++;
        x2->item[x2->count] = x1->item[j];
        x2->link[x2->count] = x1->link[j];
        j++;
    }
}

```

```

j = pos;
while (j < myNode->count) {
    myNode->item[j] = myNode->item[j + 1];
    myNode->link[j] = myNode->link[j + 1];
    j++;
}
myNode->count--;
free(x1);
}

// Adjust the node
void adjustNode(struct btreeNode *myNode, int pos) {
    if (!pos) {
        if (myNode->link[1]->count > MIN) {
            leftShift(myNode, 1);
        } else {
            mergeNodes(myNode, 1);
        }
    } else {
        if (myNode->count != pos) {
            if (myNode->link[pos - 1]->count > MIN) {
                rightShift(myNode, pos);
            } else {
                if (myNode->link[pos + 1]->count > MIN) {
                    leftShift(myNode, pos + 1);
                } else {
                    mergeNodes(myNode, pos);
                }
            }
        } else {
            if (myNode->link[pos - 1]->count > MIN)
                rightShift(myNode, pos);
            else
                mergeNodes(myNode, pos);
        }
    }
}

// Traverse the tree
void traversal(struct btreeNode *myNode) {
    int i;
    if (myNode) {
        for (i = 0; i < myNode->count; i++) {
            traversal(myNode->link[i]);
            printf("%d ", myNode->item[i + 1]);
        }
        traversal(myNode->link[i]);
    }
}

int main()
{
    int item, ch, size;

    printf("Enter the Number of element : ");
    scanf("%d", &size);

    // 9,10,11,15,16,17,18,20,23
    for (int i=0; i<= size-1; i++) {

```

```
printf("Enter the element : ");
scanf("%d",&item);
insertion(item);
}

printf("The Tree element is : ");
traversal(root);
}
```

Output

```
Enter the Number of element : 10
Enter the element : 9
Enter the element : 10
Enter the element : 11
Enter the element : 15
Enter the element : 16
Enter the element : 17
Enter the element : 18
Enter the element : 20
Enter the element : 23
Enter the element : 24
The Tree element is : 9 10 11 15 16 17 18 20 23 24
```

Watch Video:-

[Insertion of elements in B-tree of order 3](#)

[Insertion in B-Tree of Order 4](#)

[Insertion of elements in B-Tree of Order 5](#)

[Insertion in B-Tree of Order 5 with Given Alphabets](#)

Deletion from a B-tree

Deleting an element on a B-tree consists of **three main events**: searching the node where the key to be deleted exists, deleting the key and balancing the tree if required.

While deleting a tree, a condition called **underflow** may occur. Underflow occurs when a node contains less than the minimum number of keys it should hold.

The terms to be understood before studying deletion operation are:

- **Inorder Predecessor** :- The largest key on the left child of a node is called its inorder predecessor.
- **Inorder Successor** :- The smallest key on the right child of a node is called its inorder successor.

Deletion Operation

Before going through the steps below, one must know these facts about a B tree of degree m.

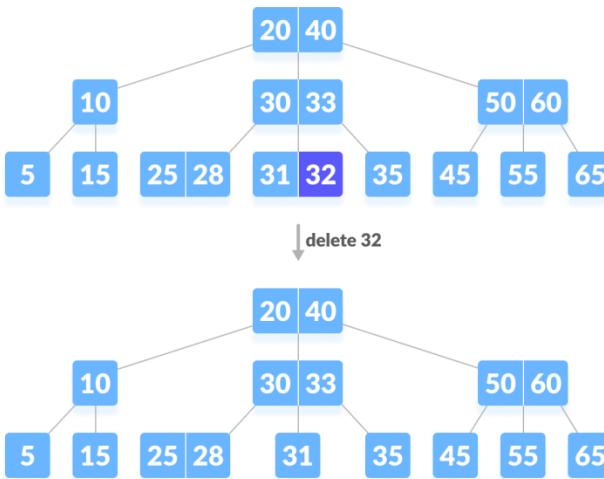
- A node can have a maximum of m children. (i.e. 3)
- A node can contain a maximum of m - 1 keys. (i.e. 2)
- A node should have a minimum of $[m/2]$ children. (i.e. 2)
- A node (except root node) should contain a minimum of $[m/2] - 1$ keys. (i.e. 1)

There are three main cases for deletion operation in a B tree.

Case I

The key to be deleted lies in the leaf. There are two cases for it.

1. **The deletion of the key does not violate the property of the minimum number of keys a node should hold.** In the tree below, deleting 32 does not violate the above properties.



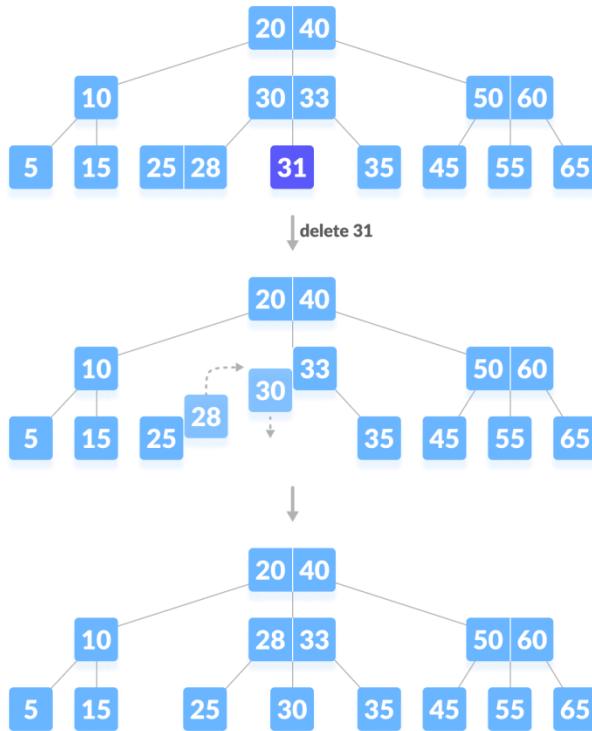
2. **The deletion of the key violates the property of the minimum number of keys a node should hold.**

In this case, we borrow a key from its immediate neighboring sibling node in the order of left to right.

First, visit the immediate left sibling. If the left sibling node has more than a minimum number of keys, then borrow a key from this node.

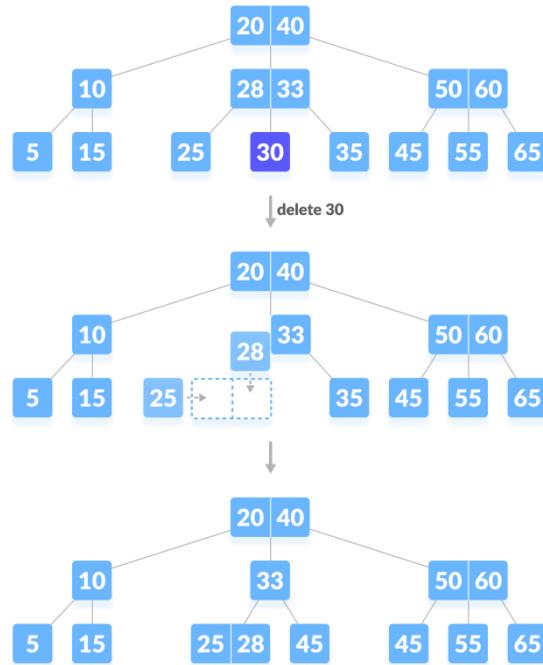
Else, check to borrow from the immediate right sibling node.

In the tree below, deleting 31 results in the above condition. Let us borrow a key from the left sibling node.



Deleting a leaf key (31) If both the immediate sibling nodes already have a minimum number of keys, then merge the node with either the left sibling node or the right sibling node. This merging is done through the parent node.

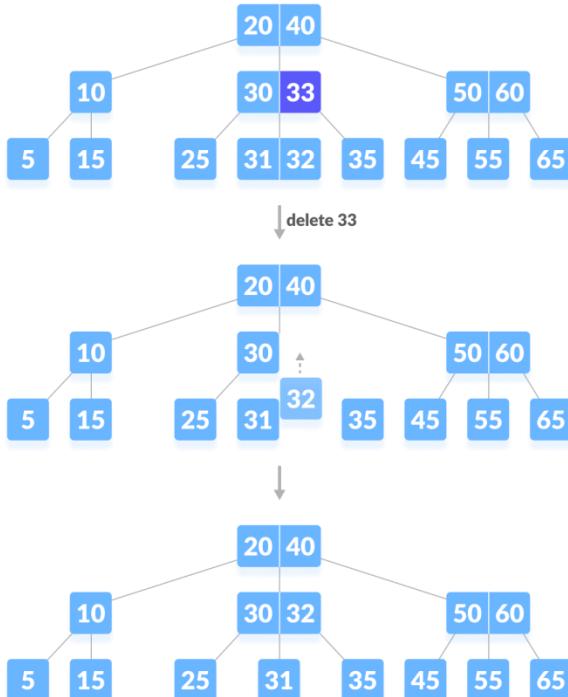
Deleting 30 results in the above case.



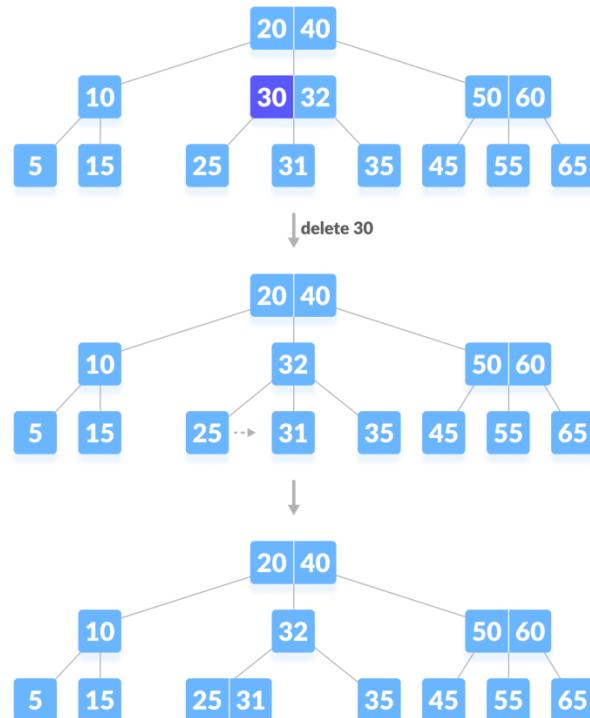
Case II

If the key to be deleted lies in the internal node, the following cases occur.

1. The internal node, which is deleted, is replaced by an inorder predecessor if the left child has more than the minimum number of keys.



2. The internal node, which is deleted, is replaced by an inorder successor if the right child has more than the minimum number of keys.
3. If either child has exactly a minimum number of keys then, merge the left and the right children.

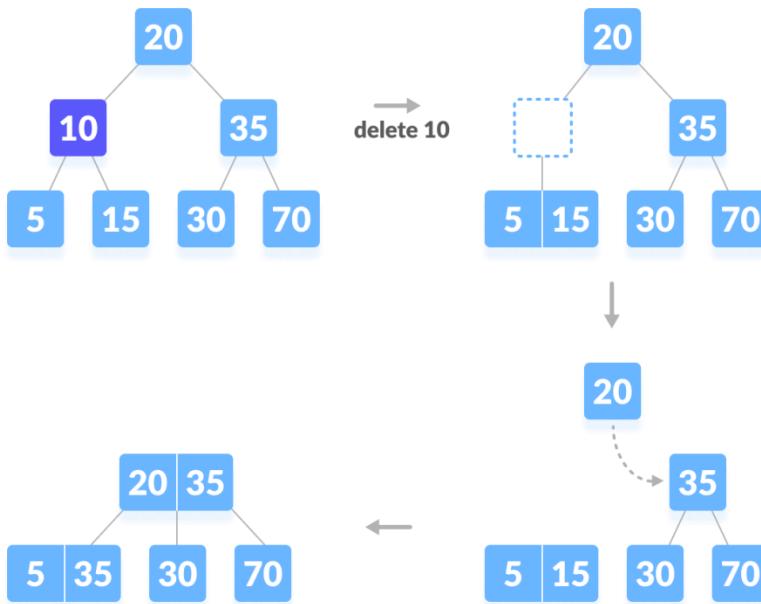


Deleting an internal node (30) After merging if the parent node has less than the minimum number of keys then, look for the siblings as in Case I.

Case III

In this case, the height of the tree shrinks. If the target key lies in an internal node, and the deletion of the key leads to a fewer number of keys in the node (i.e. less than the minimum required), then look for the inorder predecessor and the inorder successor. If both the children contain a minimum number of keys then, borrowing cannot take place. This leads to Case II(3) i.e. merging the children.

Again, look for the sibling to borrow a key. But, if the sibling also has only a minimum number of keys then, merge the node with the sibling along with the parent. Arrange the children accordingly (increasing order).



Program to implement B-Tree Deletion

```

// Deleting a key from a B-tree
#include <stdio.h>
#include <stdlib.h>

#define MAX 3
#define MIN 2

struct BTreenode
{
    int item[MAX + 1], count;
    struct BTreenode *linker[MAX + 1];
};

struct BTreenode *root;

// Node creation
struct BTreenode *createNode(int item, struct BTreenode *child)
{
    struct BTreenode *newNode;
    newNode = (struct BTreenode *)malloc(sizeof(struct BTreenode));
    newNode->item[1] = item;
    newNode->count = 1;
    newNode->linker[0] = root;
    newNode->linker[1] = child;
    return newNode;
}

// Add value to the node
void addValToNode(int item, int pos, struct BTreenode *node, struct BTreenode *child)
{
    int j = node->count;
    while (j > pos) {
        node->item[j + 1] = node->item[j];
        node->linker[j + 1] = node->linker[j];
        j--;
    }
    node->item[j + 1] = item;
    node->linker[j + 1] = child;
    node->count++;
}

// Split the node
void splitNode(int item, int *pval, int pos, struct BTreenode *node,
               struct BTreenode *child, struct BTreenode **newNode)
{
    int median, j;

    if (pos > MIN)
        median = MIN + 1;
    else
        median = MIN;

    *newNode = (struct BTreenode *)malloc(sizeof(struct BTreenode));
    j = median + 1;
    while (j <= MAX) {
        (*newNode)->item[j - median] = node->item[j];
        (*newNode)->linker[j - median] = node->linker[j];
        j++;
    }
}

```

```

}

node->count = median;
(*newNode)->count = MAX - median;

if (pos <= MIN) {
    addValToNode(item, pos, node, child);
} else {
    addValToNode(item, pos - median, *newNode, child);
}
*pval = node->item[node->count];
(*newNode)->linker[0] = node->linker[node->count];
node->count--;
}

// Set the value in the node
int setValueInNode(int item, int *pval,
                    struct BTreenode *node, struct BTreenode **child)
{
    int pos;
    if (!node) {
        *pval = item;
        *child = NULL;
        return 1;
    }

    if (item < node->item[1]) {
        pos = 0;
    } else {
        for (pos = node->count;
              (item < node->item[pos] && pos > 1); pos--)
        ;
        if (item == node->item[pos]) {
            printf("Duplicates not allowed\n");
            return 0;
        }
    }
    if (setValueInNode(item, pval, node->linker[pos], child)) {
        if (node->count < MAX) {
            addValToNode(*pval, pos, node, *child);
        } else {
            splitNode(*pval, pval, pos, node, *child, child);
            return 1;
        }
    }
    return 0;
}

// Insertion operation
void insertion(int item) {
    int flag, i;
    struct BTreenode *child;

    flag = setValueInNode(item, &i, root, &child);
    if (flag)
        root = createNode(i, child);
}

// Copy the successor
void copySuccessor(struct BTreenode *myNode, int pos)
{

```

```

struct BTreenode *dummy;
dummy = myNode->linker[pos];

for (; dummy->linker[0] != NULL;)
    dummy = dummy->linker[0];
myNode->item[pos] = dummy->item[1];
}

// Remove the value
void removeVal(struct BTreenode *myNode, int pos) {
    int i = pos + 1;
    while (i <= myNode->count) {
        myNode->item[i - 1] = myNode->item[i];
        myNode->linker[i - 1] = myNode->linker[i];
        i++;
    }
    myNode->count--;
}

// Do right shift
void rightShift(struct BTreenode *myNode, int pos)
{
    struct BTreenode *x = myNode->linker[pos];
    int j = x->count;

    while (j > 0) {
        x->item[j + 1] = x->item[j];
        x->linker[j + 1] = x->linker[j];
    }
    x->item[1] = myNode->item[pos];
    x->linker[1] = x->linker[0];
    x->count++;

    x = myNode->linker[pos - 1];
    myNode->item[pos] = x->item[x->count];
    myNode->linker[pos] = x->linker[x->count];
    x->count--;
    return;
}

// Do left shift
void leftShift(struct BTreenode *myNode, int pos) {
    int j = 1;
    struct BTreenode *x = myNode->linker[pos - 1];

    x->count++;
    x->item[x->count] = myNode->item[pos];
    x->linker[x->count] = myNode->linker[pos]->linker[0];

    x = myNode->linker[pos];
    myNode->item[pos] = x->item[1];
    x->linker[0] = x->linker[1];
    x->count--;

    while (j <= x->count) {
        x->item[j] = x->item[j + 1];
        x->linker[j] = x->linker[j + 1];
        j++;
    }
    return;
}

```

```

}

// Merge the nodes
void mergeNodes(struct BTreenode *myNode, int pos)
{
    int j = 1;
    struct BTreenode *x1 = myNode->linker[pos], *x2 = myNode->linker[pos - 1];

    x2->count++;
    x2->item[x2->count] = myNode->item[pos];
    x2->linker[x2->count] = myNode->linker[0];

    while (j <= x1->count) {
        x2->count++;
        x2->item[x2->count] = x1->item[j];
        x2->linker[x2->count] = x1->linker[j];
        j++;
    }

    j = pos;
    while (j < myNode->count) {
        myNode->item[j] = myNode->item[j + 1];
        myNode->linker[j] = myNode->linker[j + 1];
        j++;
    }
    myNode->count--;
    free(x1);
}

// Adjust the node
void adjustNode(struct BTreenode *myNode, int pos) {
    if (!pos) {
        if (myNode->linker[1]->count > MIN) {
            leftShift(myNode, 1);
        } else {
            mergeNodes(myNode, 1);
        }
    } else {
        if (myNode->count != pos) {
            if (myNode->linker[pos - 1]->count > MIN) {
                rightShift(myNode, pos);
            } else {
                if (myNode->linker[pos + 1]->count > MIN) {
                    leftShift(myNode, pos + 1);
                } else {
                    mergeNodes(myNode, pos);
                }
            }
        } else {
            if (myNode->linker[pos - 1]->count > MIN)
                rightShift(myNode, pos);
            else
                mergeNodes(myNode, pos);
        }
    }
}

// Delete a value from the node
int delValFromNode(int item, struct BTreenode *myNode) {
    int pos, flag = 0;
}

```

```

if (myNode) {
    if (item < myNode->item[1]) {
        pos = 0;
        flag = 0;
    } else {
        for (pos = myNode->count; (item < myNode->item[pos] && pos > 1); pos--)
            ;
        if (item == myNode->item[pos]) {
            flag = 1;
        } else {
            flag = 0;
        }
    }
    if (flag) {
        if (myNode->linker[pos - 1]) {
            copySuccessor(myNode, pos);
            flag = delValFromNode(myNode->item[pos], myNode->linker[pos]);
            if (flag == 0) {
                printf("Given data is not present in B-Tree\n");
            }
        } else {
            removeVal(myNode, pos);
        }
    } else {
        flag = delValFromNode(item, myNode->linker[pos]);
    }
    if (myNode->linker[pos]) {
        if (myNode->linker[pos]->count < MIN)
            adjustNode(myNode, pos);
    }
}
return flag;
}

// Delete operation
void delete (int item, struct BTreenode *myNode) {
    struct BTreenode *tmp;
    if (!delValFromNode(item, myNode)) {
        printf("Element not found\n");
        return;
    } else {
        if (myNode->count == 0) {
            tmp = myNode;
            myNode = myNode->linker[0];
            free(tmp);
        }
    }
    root = myNode;
    return;
}

void searching(int item, int *pos, struct BTreenode *myNode)
{
    if (!myNode) {
        return;
    }

    if (item < myNode->item[1]) {
        *pos = 0;
    } else {

```

```

for (*pos = myNode->count;
      (item < myNode->item[*pos] && *pos > 1); (*pos)--)
{
    if (item == myNode->item[*pos]) {
        printf("%d present in B-tree", item);
        return;
    }
}
searching(item, pos, myNode->linker[*pos]);
return;
}

void traversal(struct BTreenode *myNode)
{
    int i;
    if (myNode) {
        for (i = 0; i < myNode->count; i++) {
            traversal(myNode->linker[i]);
            printf("%d ", myNode->item[i + 1]);
        }
        traversal(myNode->linker[i]);
    }
}

int main()
{
    int item, ch, size, del_item;
    printf("Enter the Number of element : ");
    scanf("%d", &size);

    // 9,10,11,15,16,17,18,20,23
    for (int i=0; i<= size-1; i++) {
        printf("Enter the element : ");
        scanf("%d", &item);
        insertion(item);
    }
    printf("The Tree element is : ");
    traversal(root);
    printf("\nEnter the element for deletion : ");
    scanf("%d", &del_item);

    delete (del_item, root);

    printf("The Tree after deletion is : ");
    traversal(root);
}

```

Output

```

Enter the Number of element : 9
Enter the element : 9 10 11 15 16 17 18 20 23
The Tree element is : 9 10 11 15 16 17 18 20 23
Enter the element for deletion : 16
The Tree after deletion is : 9 10 11 15 17 18 20 23

```

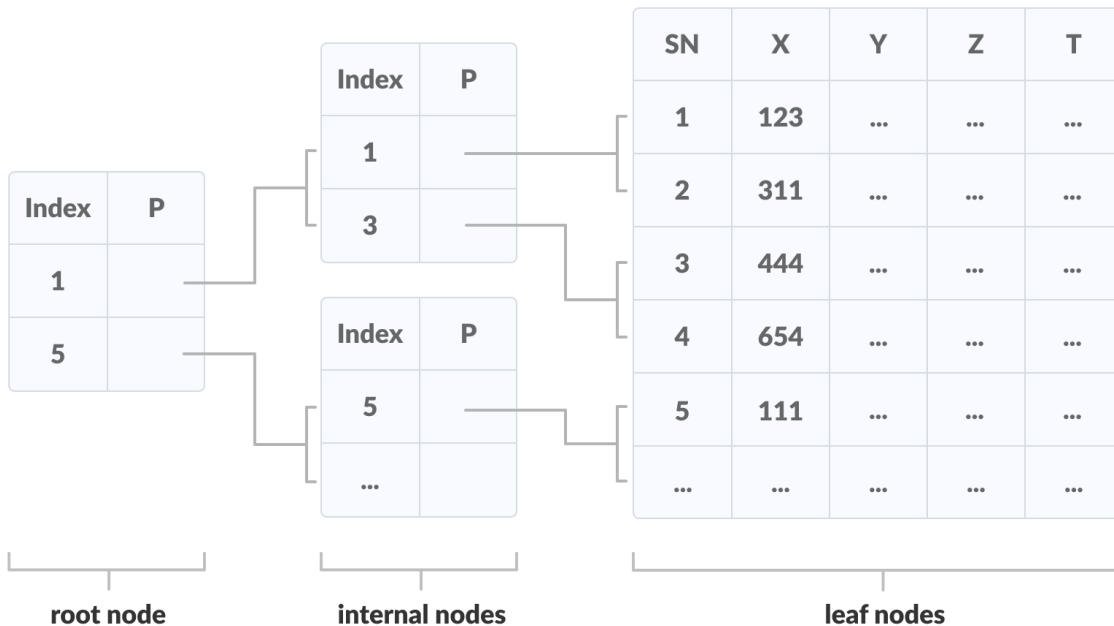
Watch Video:-

[B Tree deletion](#)

B+ Tree

A B+ tree is an advanced form of a self-balancing tree in which all the values are present in the leaf level.

An important concept to be understood before learning B+ tree is multilevel indexing. In multilevel indexing, the index of indices is created as in figure below. It makes accessing the data easier and faster.

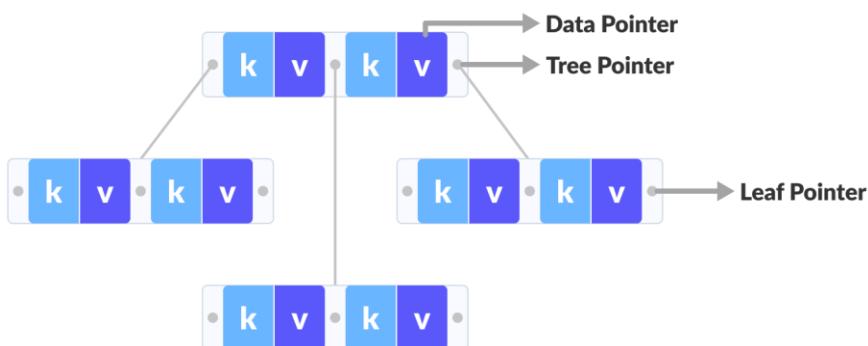


Properties of a B+ Tree

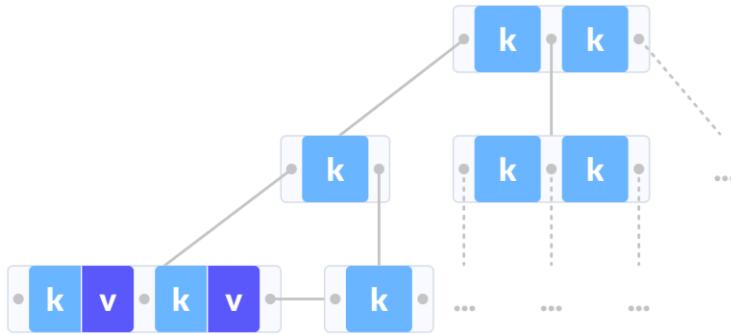
- All leaves are at the same level.
- The root has at least two children.
- Each node except root can have a maximum of m children and at least $m/2$ children.
- Each node can contain a maximum of $m - 1$ keys and a minimum of $[m/2] - 1$ keys.

Comparison between a B-tree and a B+ Tree

B-tree



B+ tree



The data pointers are present only at the leaf nodes on a B+ tree whereas the data pointers are present in the internal, leaf or root nodes on a B-tree.

The leaves are not connected with each other on a B-tree whereas they are connected on a B+ tree.

Operations on a B+ tree are faster than on a B-tree.

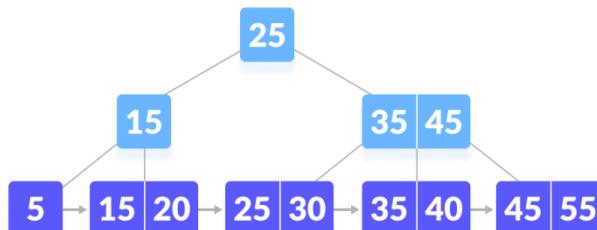
Searching on a B+ Tree

The following steps are followed to search for data in a B+ Tree of order m. Let the data to be searched be k.

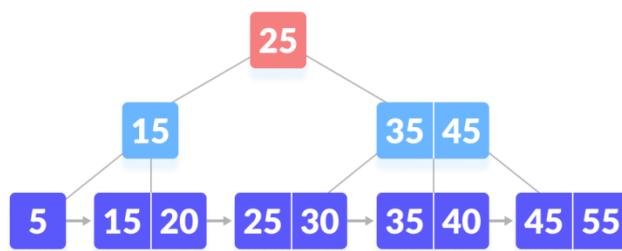
- Start from the root node. Compare k with the keys at the root node [$k_1, k_2, k_3, \dots, k_{m-1}$].
- If $k < k_1$, go to the left child of the root node.
- Else if $k == k_1$, compare k_2 . If $k < k_2$, k lies between k_1 and k_2 . So, search in the left child of k_2 .
- If $k > k_2$, go for k_3, k_4, \dots, k_{m-1} as in steps 2 and 3.
- Repeat the above steps until a leaf node is reached.
- If k exists in the leaf node, return true else return false.

Searching Example on a B+ Tree

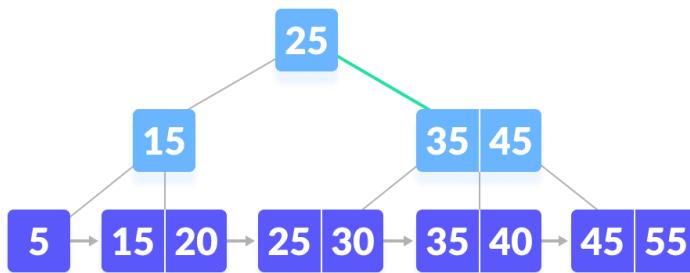
Let us search $k = 45$ on the following B+ tree.



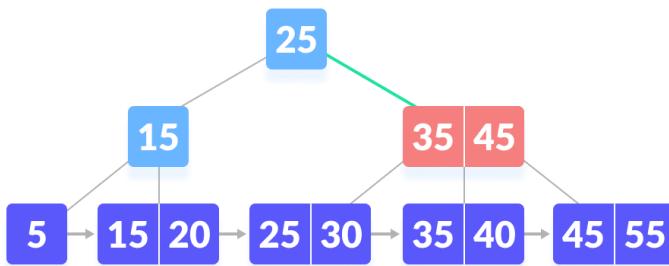
1. Compare k with the root node.



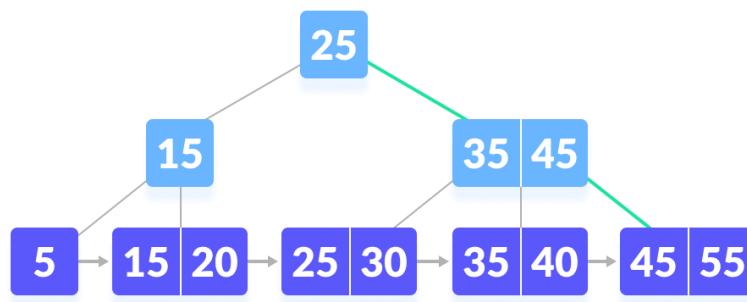
2. Since $k > 25$, go to the right child.



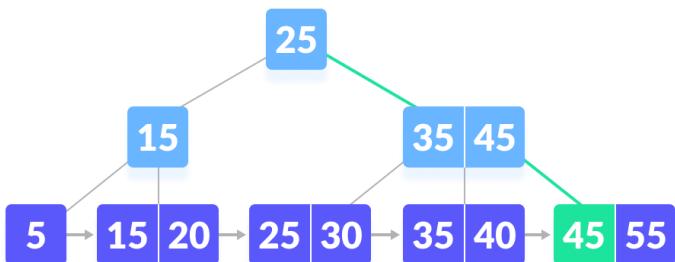
3. Compare k with 35. Since $k > 30$, compare k with 45.



4. Since $k \geq 45$, so go to the right child.



5. k is found.



Program to Implement B+Tree

```

// Searching on a B+ Tree
#include <stdbool.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

// Default order
#define ORDER 3

typedef struct record {
    int value;
} record;

// Node
typedef struct node {
    void **pointers;
    int *keys;
    struct node *parent;
    bool is_leaf;
    int num_keys;
    struct node *next;
} node;

int order = ORDER;
node *queue = NULL;
bool verbose_output = false;

// Enqueue
void enqueue(node *new_node);

// Dequeue
node *dequeue(void);
int height(node *const root);
int pathToLeaves(node *const root, node *child);
void printLeaves(node *const root);
void printTree(node *const root);
void findAndPrint(node *const root, int key, bool verbose);
void findAndPrintRange(node *const root, int range1, int range2, bool verbose);
int findRange(node *const root, int key_start, int key_end, bool verbose,
             int returned_keys[], void *returned_pointers[]);
node *findLeaf(node *const root, int key, bool verbose);
record *find(node *root, int key, bool verbose, node **leaf_out);
int cut(int length);

record *makeRecord(int value);
node *makeNode(void);
node *makeLeaf(void);
int getLeftIndex(node *parent, node *left);
node *insertIntoLeaf(node *leaf, int key, record *pointer);
node *insertIntoLeafAfterSplitting(node *root, node *leaf, int key,
                                   record *pointer);
node *insertIntoNode(node *root, node *parent,
                     int left_index, int key, node *right);
node *insertIntoNodeAfterSplitting(node *root, node *parent,
                                   int left_index,
                                   int key, node *right);
node *insertIntoParent(node *root, node *left, int key, node *right);
node *insertIntoNewRoot(node *left, int key, node *right);

```

```

node *startNewTree(int key, record *pointer);
node *insert(node *root, int key, int value);

// Enqueue
void enqueue(node *new_node) {
    node *c;
    if (queue == NULL) {
        queue = new_node;
        queue->next = NULL;
    } else {
        c = queue;
        while (c->next != NULL) {
            c = c->next;
        }
        c->next = new_node;
        new_node->next = NULL;
    }
}

// Dequeue
node *dequeue(void) {
    node *n = queue;
    queue = queue->next;
    n->next = NULL;
    return n;
}

// Print the leaves
void printLeaves(node *const root) {
    if (root == NULL) {
        printf("Empty tree.\n");
        return;
    }
    int i;
    node *c = root;
    while (!c->is_leaf)
        c = c->pointers[0];
    while (true) {
        for (i = 0; i < c->num_keys; i++) {
            if (verbose_output)
                printf("%p ", c->pointers[i]);
            printf("%d ", c->keys[i]);
        }
        if (verbose_output)
            printf("%p ", c->pointers[order - 1]);
        if (c->pointers[order - 1] != NULL) {
            printf(" | ");
            c = c->pointers[order - 1];
        } else
            break;
    }
    printf("\n");
}

// Calculate height
int height(node *const root) {
    int h = 0;
    node *c = root;
    while (!c->is_leaf) {
        c = c->pointers[0];
        h++;
        while (c->next != NULL)
            c = c->next;
    }
    return h;
}

```

```

        h++;
    }
    return h;
}

// Get path to root
int pathToLeaves(node *const root, node *child) {
    int length = 0;
    node *c = child;
    while (c != root) {
        c = c->parent;
        length++;
    }
    return length;
}

// Print the tree
void printTree(node *const root) {
    node *n = NULL;
    int i = 0;
    int rank = 0;
    int new_rank = 0;

    if (root == NULL) {
        printf("Empty tree.\n");
        return;
    }
    queue = NULL;
    enqueue(root);
    while (queue != NULL) {
        n = dequeue();
        if (n->parent != NULL && n == n->parent->pointers[0]) {
            new_rank = pathToLeaves(root, n);
            if (new_rank != rank) {
                rank = new_rank;
                printf("\n");
            }
        }
        if (verbose_output)
            printf("(%p)", n);
        for (i = 0; i < n->num_keys; i++) {
            if (verbose_output)
                printf("%p ", n->pointers[i]);
            printf("%d ", n->keys[i]);
        }
        if (!n->is_leaf)
            for (i = 0; i <= n->num_keys; i++)
                enqueue(n->pointers[i]);
        if (verbose_output) {
            if (n->is_leaf)
                printf("%p ", n->pointers[order - 1]);
            else
                printf("%p ", n->pointers[n->num_keys]);
        }
        printf(" | ");
    }
    printf("\n");
}

```

```

// Find the node and print it
void findAndPrint(node *const root, int key, bool verbose) {
    node *leaf = NULL;
    record *r = find(root, key, verbose, NULL);
    if (r == NULL)
        printf("Record not found under key %d.\n", key);
    else
        printf("Record at %p -- key %d, value %d.\n",
               r, key, r->value);
}

// Find and print the range
void findAndPrintRange(node *const root, int key_start, int key_end,
                      bool verbose) {
    int i;
    int array_size = key_end - key_start + 1;
    int returned_keys[array_size];
    void *returned_pointers[array_size];
    int num_found = findRange(root, key_start, key_end, verbose,
                               returned_keys, returned_pointers);
    if (!num_found)
        printf("None found.\n");
    else {
        for (i = 0; i < num_found; i++)
            printf("Key: %d  Location: %p  Value: %d\n",
                   returned_keys[i],
                   returned_pointers[i],
                   ((record *)
                    returned_pointers[i])
                   ->value);
    }
}

// Find the range
int findRange(node *const root, int key_start, int key_end, bool verbose,
              int returned_keys[], void *returned_pointers[]) {
    int i, num_found;
    num_found = 0;
    node *n = findLeaf(root, key_start, verbose);
    if (n == NULL)
        return 0;
    for (i = 0; i < n->num_keys && n->keys[i] < key_start; i++)
    ;
    if (i == n->num_keys)
        return 0;
    while (n != NULL) {
        for (i = 0; i < n->num_keys && n->keys[i] <= key_end; i++) {
            returned_keys[num_found] = n->keys[i];
            returned_pointers[num_found] = n->pointers[i];
            num_found++;
        }
        n = n->pointers[order - 1];
        i = 0;
    }
    return num_found;
}

// Find the leaf
node *findLeaf(node *const root, int key, bool verbose) {
    if (root == NULL) {

```

```

if (verbose)
    printf("Empty tree.\n");
return root;
}
int i = 0;
node *c = root;
while (!c->is_leaf) {
    if (verbose) {
        printf("[");
        for (i = 0; i < c->num_keys - 1; i++)
            printf("%d ", c->keys[i]);
        printf("%d] ", c->keys[i]);
    }
    i = 0;
    while (i < c->num_keys) {
        if (key >= c->keys[i])
            i++;
        else
            break;
    }
    if (verbose)
        printf("%d ->\n", i);
    c = (node *)c->pointers[i];
}
if (verbose) {
    printf("Leaf [");
    for (i = 0; i < c->num_keys - 1; i++)
        printf("%d ", c->keys[i]);
    printf("%d] ->\n", c->keys[i]);
}
return c;
}

record *find(node *root, int key, bool verbose, node **leaf_out) {
    if (root == NULL) {
        if (leaf_out != NULL) {
            *leaf_out = NULL;
        }
        return NULL;
    }

    int i = 0;
    node *leaf = NULL;

    leaf = findLeaf(root, key, verbose);

    for (i = 0; i < leaf->num_keys; i++)
        if (leaf->keys[i] == key)
            break;
    if (leaf_out != NULL) {
        *leaf_out = leaf;
    }
    if (i == leaf->num_keys)
        return NULL;
    else
        return (record *)leaf->pointers[i];
}

int cut(int length) {
    if (length % 2 == 0)

```

```

    return length / 2;
else
    return length / 2 + 1;
}

record *makeRecord(int value) {
    record *new_record = (record *)malloc(sizeof(record));
    if (new_record == NULL) {
        perror("Record creation.");
        exit(EXIT_FAILURE);
    } else {
        new_record->value = value;
    }
    return new_record;
}

node *makeNode(void) {
    node *new_node;
    new_node = malloc(sizeof(node));
    if (new_node == NULL) {
        perror("Node creation.");
        exit(EXIT_FAILURE);
    }
    new_node->keys = malloc((order - 1) * sizeof(int));
    if (new_node->keys == NULL) {
        perror("New node keys array.");
        exit(EXIT_FAILURE);
    }
    new_node->pointers = malloc(order * sizeof(void *));
    if (new_node->pointers == NULL) {
        perror("New node pointers array.");
        exit(EXIT_FAILURE);
    }
    new_node->is_leaf = false;
    new_node->num_keys = 0;
    new_node->parent = NULL;
    new_node->next = NULL;
    return new_node;
}

node *makeLeaf(void) {
    node *leaf = makeNode();
    leaf->is_leaf = true;
    return leaf;
}

int getLeftIndex(node *parent, node *left) {
    int left_index = 0;
    while (left_index <= parent->num_keys &&
           parent->pointers[left_index] != left)
        left_index++;
    return left_index;
}

node *insertIntoLeaf(node *leaf, int key, record *pointer) {
    int i, insertion_point;

    insertion_point = 0;
    while (insertion_point < leaf->num_keys && leaf->keys[insertion_point] < key)
        insertion_point++;
}

```

```

for (i = leaf->num_keys; i > insertion_point; i--) {
    leaf->keys[i] = leaf->keys[i - 1];
    leaf->pointers[i] = leaf->pointers[i - 1];
}
leaf->keys[insertion_point] = key;
leaf->pointers[insertion_point] = pointer;
leaf->num_keys++;
return leaf;
}

node *insertIntoLeafAfterSplitting(node *root, node *leaf, int key, record *pointer)
{
    node *new_leaf;
    int *temp_keys;
    void **temp_pointers;
    int insertion_index, split, new_key, i, j;

    new_leaf = makeLeaf();

    temp_keys = malloc(order * sizeof(int));
    if (temp_keys == NULL) {
        perror("Temporary keys array.");
        exit(EXIT_FAILURE);
    }

    temp_pointers = malloc(order * sizeof(void *));
    if (temp_pointers == NULL) {
        perror("Temporary pointers array.");
        exit(EXIT_FAILURE);
    }

    insertion_index = 0;
    while (insertion_index < order - 1 && leaf->keys[insertion_index] < key)
        insertion_index++;

    for (i = 0, j = 0; i < leaf->num_keys; i++, j++) {
        if (j == insertion_index)
            j++;
        temp_keys[j] = leaf->keys[i];
        temp_pointers[j] = leaf->pointers[i];
    }

    temp_keys[insertion_index] = key;
    temp_pointers[insertion_index] = pointer;

    leaf->num_keys = 0;

    split = cut(order - 1);

    for (i = 0; i < split; i++) {
        leaf->pointers[i] = temp_pointers[i];
        leaf->keys[i] = temp_keys[i];
        leaf->num_keys++;
    }

    for (i = split, j = 0; i < order; i++, j++) {
        new_leaf->pointers[j] = temp_pointers[i];
        new_leaf->keys[j] = temp_keys[i];
        new_leaf->num_keys++;
    }
}

```

```

}

free(temp_pointers);
free(temp_keys);

new_leaf->pointers[order - 1] = leaf->pointers[order - 1];
leaf->pointers[order - 1] = new_leaf;

for (i = leaf->num_keys; i < order - 1; i++)
    leaf->pointers[i] = NULL;
for (i = new_leaf->num_keys; i < order - 1; i++)
    new_leaf->pointers[i] = NULL;

new_leaf->parent = leaf->parent;
new_key = new_leaf->keys[0];

return insertIntoParent(root, leaf, new_key, new_leaf);
}

node *insertIntoNode(node *root, node *n,
                     int left_index, int key, node *right) {
int i;

for (i = n->num_keys; i > left_index; i--) {
    n->pointers[i + 1] = n->pointers[i];
    n->keys[i] = n->keys[i - 1];
}
n->pointers[left_index + 1] = right;
n->keys[left_index] = key;
n->num_keys++;
return root;
}

node *insertIntoNodeAfterSplitting(node *root, node *old_node, int left_index,
                                   int key, node *right) {
int i, j, split, k_prime;
node *new_node, *child;
int *temp_keys;
node **temp_pointers;

temp_pointers = malloc((order + 1) * sizeof(node *));
if (temp_pointers == NULL) {
    exit(EXIT_FAILURE);
}
temp_keys = malloc(order * sizeof(int));
if (temp_keys == NULL) {
    exit(EXIT_FAILURE);
}

for (i = 0, j = 0; i < old_node->num_keys + 1; i++, j++) {
    if (j == left_index + 1)
        j++;
    temp_pointers[j] = old_node->pointers[i];
}

for (i = 0, j = 0; i < old_node->num_keys; i++, j++) {
    if (j == left_index)
        j++;
    temp_keys[j] = old_node->keys[i];
}
}

```

```

temp_pointers[left_index + 1] = right;
temp_keys[left_index] = key;

split = cut(order);
new_node = makeNode();
old_node->num_keys = 0;
for (i = 0; i < split - 1; i++) {
    old_node->pointers[i] = temp_pointers[i];
    old_node->keys[i] = temp_keys[i];
    old_node->num_keys++;
}
old_node->pointers[i] = temp_pointers[i];
k_prime = temp_keys[split - 1];
for (++i, j = 0; i < order; i++, j++) {
    new_node->pointers[j] = temp_pointers[i];
    new_node->keys[j] = temp_keys[i];
    new_node->num_keys++;
}
new_node->pointers[j] = temp_pointers[i];
free(temp_pointers);
free(temp_keys);
new_node->parent = old_node->parent;
for (i = 0; i <= new_node->num_keys; i++) {
    child = new_node->pointers[i];
    child->parent = new_node;
}
return insertIntoParent(root, old_node, k_prime, new_node);
}

node *insertIntoParent(node *root, node *left, int key, node *right) {
    int left_index;
    node *parent;

    parent = left->parent;

    if (parent == NULL)
        return insertIntoNewRoot(left, key, right);

    left_index = getLeftIndex(parent, left);

    if (parent->num_keys < order - 1)
        return insertIntoNode(root, parent, left_index, key, right);

    return insertIntoNodeAfterSplitting(root, parent, left_index, key, right);
}

node *insertIntoNewRoot(node *left, int key, node *right) {
    node *root = makeNode();
    root->keys[0] = key;
    root->pointers[0] = left;
    root->pointers[1] = right;
    root->num_keys++;
    root->parent = NULL;
    left->parent = root;
    right->parent = root;
    return root;
}

node *startNewTree(int key, record *pointer) {

```

```

node *root = makeLeaf();
root->keys[0] = key;
root->pointers[0] = pointer;
root->pointers[order - 1] = NULL;
root->parent = NULL;
root->num_keys++;
return root;
}

node *insert(node *root, int key, int value) {
    record *record_pointer = NULL;
    node *leaf = NULL;

    record_pointer = find(root, key, false, NULL);
    if (record_pointer != NULL) {
        record_pointer->value = value;
        return root;
    }
    record_pointer = makeRecord(value);

    if (root == NULL)
        return startNewTree(key, record_pointer);

    leaf = findLeaf(root, key, false);

    if (leaf->num_keys < order - 1) {
        leaf = insertIntoLeaf(leaf, key, record_pointer);
        return root;
    }
    return insertIntoLeafAfterSplitting(root, leaf, key, record_pointer);
}

int main() {
    node *root;
    char instruction;
    root = NULL;
    root = insert(root, 5, 33);
    root = insert(root, 15, 21);
    root = insert(root, 25, 31);
    root = insert(root, 35, 41);
    root = insert(root, 45, 10);
    printTree(root);
    findAndPrint(root, 15, instruction = 'a');
}

```

Output

```

25 |
15 | 35 |
5 | 15 | 25 | 35 45 |
[25] 0 ->
[15] 1 ->
Leaf [15] ->
Record at 0x559a6709d330 -- key 15, value 21.

```

Watch Videos:-

[B+ Trees](#)

Insertion on a B+ Tree

Inserting an element into a B+ tree consists of three main events: searching the appropriate leaf, inserting the element and balancing/splitting the tree.

Let us understand these events below.

Insertion Operation

Before inserting an element into a B+ tree, these properties must be kept in mind.

- The root has at least two children.
- Each node except root can have a maximum of m children and at least $m/2$ children.
- Each node can contain a maximum of $m - 1$ keys and a minimum of $[m/2] - 1$ keys.

The following steps are followed for inserting an element.

- Since every element is inserted into the leaf node, go to the appropriate leaf node.
- Insert the key into the leaf node.

Case I

If the leaf is not full, insert the key into the leaf node in increasing order.

Case II

If the leaf is full, insert the key into the leaf node in increasing order and balance the tree in the following way.

- Break the node at $m/2$ th position.
- Add $m/2$ th key to the parent node as well.
- If the parent node is already full, follow steps 2 to 3.

Insertion Example

Let us understand the insertion operation with the illustrations below.

The elements to be inserted are 5, 15, 25, 35, 45.

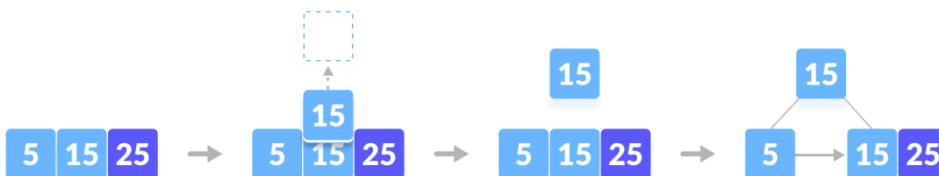
1. Insert 5.



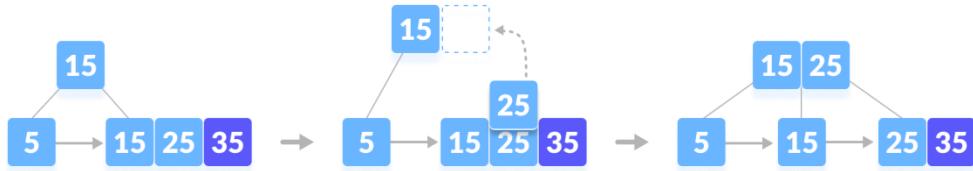
2. Insert 15.



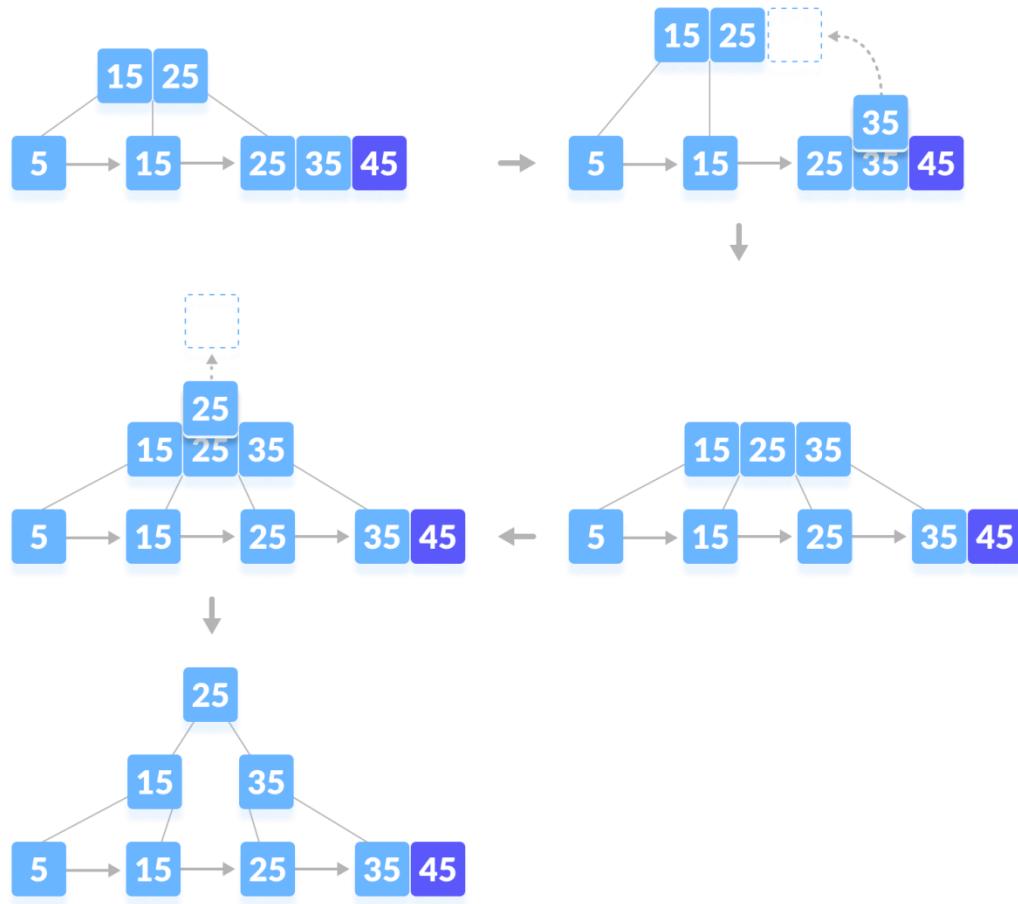
3. Insert 25.



4. Insert 35.



5. Insert 45.



Program to implement B+ Tree - Insertion

```

// B+ Tree insertion
#include <stdbool.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

// Default order
#define ORDER 3

typedef struct record {
    int value;
} record;

// Node
typedef struct node {
    void **pointers;
    int *keys;
    struct node *parent;
    bool is_leaf;
    int num_keys;
    struct node *next;
} node;

int order = ORDER;
node *queue = NULL;
bool verbose_output = false;

// Enqueue
void enqueue(node *new_node);

// Dequeue
node *dequeue(void);
int height(node *const root);
int pathToLeaves(node *const root, node *child);
void printLeaves(node *const root);
void printTree(node *const root);
void findAndPrint(node *const root, int key, bool verbose);
void findAndPrintRange(node *const root, int range1, int range2, bool verbose);
int findRange(node *const root, int key_start, int key_end, bool verbose,
             int returned_keys[], void *returned_pointers[]);
node *findLeaf(node *const root, int key, bool verbose);
record *find(node *root, int key, bool verbose, node **leaf_out);
int cut(int length);

record *makeRecord(int value);
node *makeNode(void);
node *makeLeaf(void);
int getLeftIndex(node *parent, node *left);
node *insertIntoLeaf(node *leaf, int key, record *pointer);
node *insertIntoLeafAfterSplitting(node *root, node *leaf, int key,
                                   record *pointer);
node *insertIntoNode(node *root, node *parent,
                     int left_index, int key, node *right);
node *insertIntoNodeAfterSplitting(node *root, node *parent,
                                   int left_index,
                                   int key, node *right);
node *insertIntoParent(node *root, node *left, int key, node *right);
node *insertIntoNewRoot(node *left, int key, node *right);

```

```

node *startNewTree(int key, record *pointer);
node *insert(node *root, int key, int value);

// Enqueue
void enqueue(node *new_node) {
    node *c;
    if (queue == NULL) {
        queue = new_node;
        queue->next = NULL;
    } else {
        c = queue;
        while (c->next != NULL) {
            c = c->next;
        }
        c->next = new_node;
        new_node->next = NULL;
    }
}

// Dequeue
node *dequeue(void) {
    node *n = queue;
    queue = queue->next;
    n->next = NULL;
    return n;
}

// Print the leaves
void printLeaves(node *const root) {
    if (root == NULL) {
        printf("Empty tree.\n");
        return;
    }
    int i;
    node *c = root;
    while (!c->is_leaf)
        c = c->pointers[0];
    while (true) {
        for (i = 0; i < c->num_keys; i++) {
            if (verbose_output)
                printf("%p ", c->pointers[i]);
            printf("%d ", c->keys[i]);
        }
        if (verbose_output)
            printf("%p ", c->pointers[order - 1]);
        if (c->pointers[order - 1] != NULL) {
            printf(" | ");
            c = c->pointers[order - 1];
        } else
            break;
    }
    printf("\n");
}

// Calculate height
int height(node *const root) {
    int h = 0;
    node *c = root;
    while (!c->is_leaf) {
        c = c->pointers[0];
        h++;
        while (c->next != NULL)
            c = c->next;
    }
    return h;
}

```

```

        h++;
    }
    return h;
}

// Get path to root
int pathToLeaves(node *const root, node *child) {
    int length = 0;
    node *c = child;
    while (c != root) {
        c = c->parent;
        length++;
    }
    return length;
}

// Print the tree
void printTree(node *const root) {
    node *n = NULL;
    int i = 0;
    int rank = 0;
    int new_rank = 0;

    if (root == NULL) {
        printf("Empty tree.\n");
        return;
    }
    queue = NULL;
    enqueue(root);
    while (queue != NULL) {
        n = dequeue();
        if (n->parent != NULL && n == n->parent->pointers[0]) {
            new_rank = pathToLeaves(root, n);
            if (new_rank != rank) {
                rank = new_rank;
                printf("\n");
            }
        }
        if (verbose_output)
            printf("(%p)", n);
        for (i = 0; i < n->num_keys; i++) {
            if (verbose_output)
                printf("%p ", n->pointers[i]);
            printf("%d ", n->keys[i]);
        }
        if (!n->is_leaf)
            for (i = 0; i <= n->num_keys; i++)
                enqueue(n->pointers[i]);
        if (verbose_output) {
            if (n->is_leaf)
                printf("%p ", n->pointers[order - 1]);
            else
                printf("%p ", n->pointers[n->num_keys]);
        }
        printf(" | ");
    }
    printf("\n");
}

// Find the node and print it

```

```

void findAndPrint(node *const root, int key, bool verbose) {
    node *leaf = NULL;
    record *r = find(root, key, verbose, NULL);
    if (r == NULL)
        printf("Record not found under key %d.\n", key);
    else
        printf("Record at %p -- key %d, value %d.\n",
               r, key, r->value);
}

// Find and print the range
void findAndPrintRange(node *const root, int key_start, int key_end,
                      bool verbose) {
    int i;
    int array_size = key_end - key_start + 1;
    int returned_keys[array_size];
    void *returned_pointers[array_size];
    int num_found = findRange(root, key_start, key_end, verbose,
                               returned_keys, returned_pointers);
    if (!num_found)
        printf("None found.\n");
    else {
        for (i = 0; i < num_found; i++)
            printf("Key: %d  Location: %p  Value: %d\n",
                   returned_keys[i],
                   returned_pointers[i],
                   ((record *)
                    returned_pointers[i])
                   ->value);
    }
}

// Find the range
int findRange(node *const root, int key_start, int key_end, bool verbose,
              int returned_keys[], void *returned_pointers[]) {
    int i, num_found;
    num_found = 0;
    node *n = findLeaf(root, key_start, verbose);
    if (n == NULL)
        return 0;
    for (i = 0; i < n->num_keys && n->keys[i] < key_start; i++)
    ;
    if (i == n->num_keys)
        return 0;
    while (n != NULL) {
        for (; i < n->num_keys && n->keys[i] <= key_end; i++) {
            returned_keys[num_found] = n->keys[i];
            returned_pointers[num_found] = n->pointers[i];
            num_found++;
        }
        n = n->pointers[order - 1];
        i = 0;
    }
    return num_found;
}

// Find the leaf
node *findLeaf(node *const root, int key, bool verbose) {
    if (root == NULL) {
        if (verbose)

```

```

    printf("Empty tree.\n");
    return root;
}
int i = 0;
node *c = root;
while (!c->is_leaf) {
    if (verbose) {
        printf("[");
        for (i = 0; i < c->num_keys - 1; i++)
            printf("%d ", c->keys[i]);
        printf("%d] ", c->keys[i]);
    }
    i = 0;
    while (i < c->num_keys) {
        if (key >= c->keys[i])
            i++;
        else
            break;
    }
    if (verbose)
        printf("%d ->\n", i);
    c = (node *)c->pointers[i];
}
if (verbose) {
    printf("Leaf [");
    for (i = 0; i < c->num_keys - 1; i++)
        printf("%d ", c->keys[i]);
    printf("%d] ->\n", c->keys[i]);
}
return c;
}

record *find(node *root, int key, bool verbose, node **leaf_out) {
    if (root == NULL) {
        if (leaf_out != NULL) {
            *leaf_out = NULL;
        }
        return NULL;
    }

    int i = 0;
    node *leaf = NULL;

    leaf = findLeaf(root, key, verbose);

    for (i = 0; i < leaf->num_keys; i++)
        if (leaf->keys[i] == key)
            break;
    if (leaf_out != NULL) {
        *leaf_out = leaf;
    }
    if (i == leaf->num_keys)
        return NULL;
    else
        return (record *)leaf->pointers[i];
}

int cut(int length) {
    if (length % 2 == 0)
        return length / 2;
}

```

```

else
    return length / 2 + 1;
}

record *makeRecord(int value) {
    record *new_record = (record *)malloc(sizeof(record));
    if (new_record == NULL) {
        perror("Record creation.");
        exit(EXIT_FAILURE);
    } else {
        new_record->value = value;
    }
    return new_record;
}

node *makeNode(void) {
    node *new_node;
    new_node = malloc(sizeof(node));
    if (new_node == NULL) {
        perror("Node creation.");
        exit(EXIT_FAILURE);
    }
    new_node->keys = malloc((order - 1) * sizeof(int));
    if (new_node->keys == NULL) {
        perror("New node keys array.");
        exit(EXIT_FAILURE);
    }
    new_node->pointers = malloc(order * sizeof(void *));
    if (new_node->pointers == NULL) {
        perror("New node pointers array.");
        exit(EXIT_FAILURE);
    }
    new_node->is_leaf = false;
    new_node->num_keys = 0;
    new_node->parent = NULL;
    new_node->next = NULL;
    return new_node;
}

node *makeLeaf(void) {
    node *leaf = makeNode();
    leaf->is_leaf = true;
    return leaf;
}

int getLeftIndex(node *parent, node *left) {
    int left_index = 0;
    while (left_index <= parent->num_keys &&
           parent->pointers[left_index] != left)
        left_index++;
    return left_index;
}

node *insertIntoLeaf(node *leaf, int key, record *pointer) {
    int i, insertion_point;

    insertion_point = 0;
    while (insertion_point < leaf->num_keys && leaf->keys[insertion_point] < key)
        insertion_point++;
}

```

```

for (i = leaf->num_keys; i > insertion_point; i--) {
    leaf->keys[i] = leaf->keys[i - 1];
    leaf->pointers[i] = leaf->pointers[i - 1];
}
leaf->keys[insertion_point] = key;
leaf->pointers[insertion_point] = pointer;
leaf->num_keys++;
return leaf;
}

node *insertIntoLeafAfterSplitting(node *root, node *leaf, int key, record *pointer)
{
    node *new_leaf;
    int *temp_keys;
    void **temp_pointers;
    int insertion_index, split, new_key, i, j;

    new_leaf = makeLeaf();

    temp_keys = malloc(order * sizeof(int));
    if (temp_keys == NULL) {
        perror("Temporary keys array.");
        exit(EXIT_FAILURE);
    }

    temp_pointers = malloc(order * sizeof(void *));
    if (temp_pointers == NULL) {
        perror("Temporary pointers array.");
        exit(EXIT_FAILURE);
    }

    insertion_index = 0;
    while (insertion_index < order - 1 && leaf->keys[insertion_index] < key)
        insertion_index++;

    for (i = 0, j = 0; i < leaf->num_keys; i++, j++) {
        if (j == insertion_index)
            j++;
        temp_keys[j] = leaf->keys[i];
        temp_pointers[j] = leaf->pointers[i];
    }

    temp_keys[insertion_index] = key;
    temp_pointers[insertion_index] = pointer;

    leaf->num_keys = 0;

    split = cut(order - 1);

    for (i = 0; i < split; i++) {
        leaf->pointers[i] = temp_pointers[i];
        leaf->keys[i] = temp_keys[i];
        leaf->num_keys++;
    }

    for (i = split, j = 0; i < order; i++, j++) {
        new_leaf->pointers[j] = temp_pointers[i];
        new_leaf->keys[j] = temp_keys[i];
        new_leaf->num_keys++;
    }
}

```

```

free(temp_pointers);
free(temp_keys);

new_leaf->pointers[order - 1] = leaf->pointers[order - 1];
leaf->pointers[order - 1] = new_leaf;

for (i = leaf->num_keys; i < order - 1; i++)
    leaf->pointers[i] = NULL;
for (i = new_leaf->num_keys; i < order - 1; i++)
    new_leaf->pointers[i] = NULL;

new_leaf->parent = leaf->parent;
new_key = new_leaf->keys[0];

return insertIntoParent(root, leaf, new_key, new_leaf);
}

node *insertIntoNode(node *root, node *n,
                     int left_index, int key, node *right) {
int i;

for (i = n->num_keys; i > left_index; i--) {
    n->pointers[i + 1] = n->pointers[i];
    n->keys[i] = n->keys[i - 1];
}
n->pointers[left_index + 1] = right;
n->keys[left_index] = key;
n->num_keys++;
return root;
}

node *insertIntoNodeAfterSplitting(node *root, node *old_node, int left_index,
                                  int key, node *right) {
int i, j, split, k_prime;
node *new_node, *child;
int *temp_keys;
node **temp_pointers;

temp_pointers = malloc((order + 1) * sizeof(node *));
if (temp_pointers == NULL) {
    exit(EXIT_FAILURE);
}
temp_keys = malloc(order * sizeof(int));
if (temp_keys == NULL) {
    exit(EXIT_FAILURE);
}

for (i = 0, j = 0; i < old_node->num_keys + 1; i++, j++) {
    if (j == left_index + 1)
        j++;
    temp_pointers[j] = old_node->pointers[i];
}

for (i = 0, j = 0; i < old_node->num_keys; i++, j++) {
    if (j == left_index)
        j++;
    temp_keys[j] = old_node->keys[i];
}
}

```

```

temp_pointers[left_index + 1] = right;
temp_keys[left_index] = key;

split = cut(order);
new_node = makeNode();
old_node->num_keys = 0;
for (i = 0; i < split - 1; i++) {
    old_node->pointers[i] = temp_pointers[i];
    old_node->keys[i] = temp_keys[i];
    old_node->num_keys++;
}
old_node->pointers[i] = temp_pointers[i];
k_prime = temp_keys[split - 1];
for (++i, j = 0; i < order; i++, j++) {
    new_node->pointers[j] = temp_pointers[i];
    new_node->keys[j] = temp_keys[i];
    new_node->num_keys++;
}
new_node->pointers[j] = temp_pointers[i];
free(temp_pointers);
free(temp_keys);
new_node->parent = old_node->parent;
for (i = 0; i <= new_node->num_keys; i++) {
    child = new_node->pointers[i];
    child->parent = new_node;
}

return insertIntoParent(root, old_node, k_prime, new_node);
}

node *insertIntoParent(node *root, node *left, int key, node *right) {
    int left_index;
    node *parent;

    parent = left->parent;

    if (parent == NULL)
        return insertIntoNewRoot(left, key, right);

    left_index = getLeftIndex(parent, left);

    if (parent->num_keys < order - 1)
        return insertIntoNode(root, parent, left_index, key, right);

    return insertIntoNodeAfterSplitting(root, parent, left_index, key, right);
}

node *insertIntoNewRoot(node *left, int key, node *right) {
    node *root = makeNode();
    root->keys[0] = key;
    root->pointers[0] = left;
    root->pointers[1] = right;
    root->num_keys++;
    root->parent = NULL;
    left->parent = root;
    right->parent = root;
    return root;
}

node *startNewTree(int key, record *pointer) {

```

```

node *root = makeLeaf();
root->keys[0] = key;
root->pointers[0] = pointer;
root->pointers[order - 1] = NULL;
root->parent = NULL;
root->num_keys++;
return root;
}

node *insert(node *root, int key, int value) {
    record *record_pointer = NULL;
    node *leaf = NULL;
    record_pointer = find(root, key, false, NULL);
    if (record_pointer != NULL) {
        record_pointer->value = value;
        return root;
    }
    record_pointer = makeRecord(value);
    if (root == NULL)
        return startNewTree(key, record_pointer);

    leaf = findLeaf(root, key, false);

    if (leaf->num_keys < order - 1) {
        leaf = insertIntoLeaf(leaf, key, record_pointer);
        return root;
    }
    return insertIntoLeafAfterSplitting(root, leaf, key, record_pointer);
}
int main() {
    node *root;
    char instruction;
    root = NULL;
    root = insert(root, 5, 33);
    root = insert(root, 15, 21);
    root = insert(root, 25, 31);
    root = insert(root, 35, 41);
    root = insert(root, 45, 10);
    printTree(root);
    findAndPrint(root, 15, instruction = 'a');
}
/*
25 |
15 | 35 |
5 | 15 | 25 | 35 45 |
[25] 0 ->
[15] 1 ->
Leaf [15] ->
Record at 0x55c91d337330 -- key 15, value 21.
*/

```

Watch Videos:-

[B+ tree insertion](#)
[B+ tree insertion order 5](#)

Deletion from a B+ Tree

Deleting an element on a B+ tree consists of three main events: searching the node where the key to be deleted exists, deleting the key and balancing the tree if required.

Underflow is a situation when there is less number of keys in a node than the minimum number of keys it should hold.

Deletion Operation

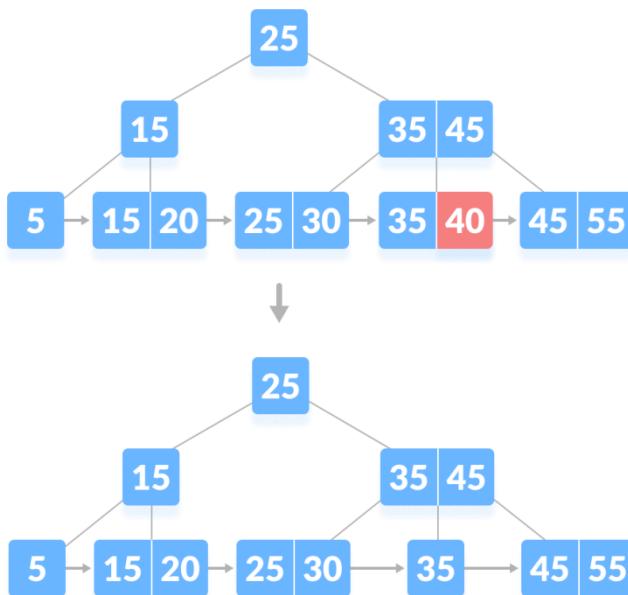
Before going through the steps below, one must know these facts about a B+ tree of degree m.

- A node can have a maximum of m children. (i.e. 3)
- A node can contain a maximum of m - 1 keys. (i.e. 2)
- A node should have a minimum of $[m/2]$ children. (i.e. 2)
- A node (except root node) should contain a minimum of $[m/2] - 1$ keys. (i.e. 1)
- While deleting a key, we have to take care of the keys present in the internal nodes (i.e. indexes) as well because the values are redundant in a B+ tree. Search the key to be deleted then follow the following steps.

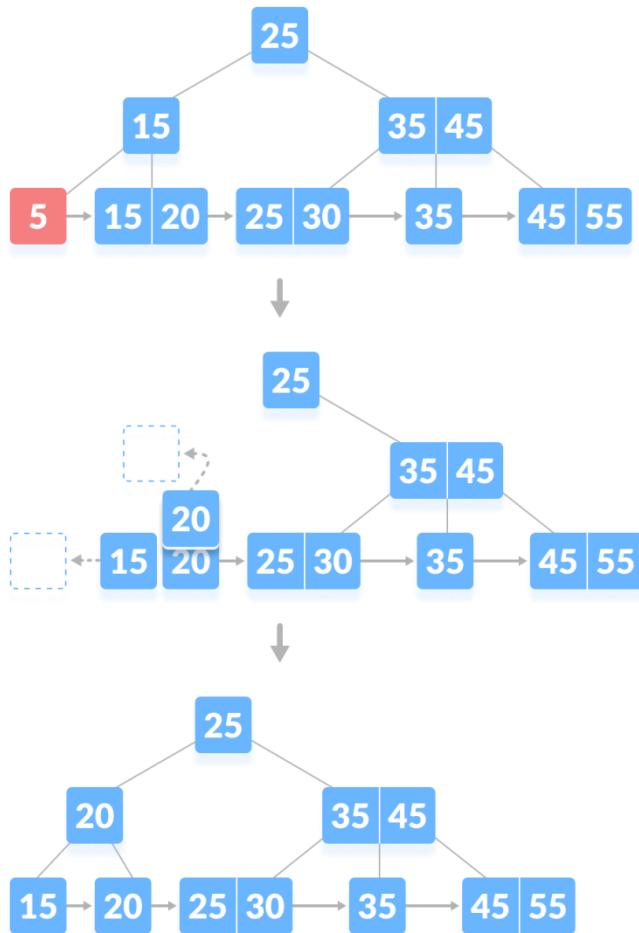
Case I

The key to be deleted is present only at the leaf node not in the indexes (or internal nodes). There are two cases for it:

1. There is more than the minimum number of keys in the node. Simply delete the key.



2. There is an exact minimum number of keys in the node. Delete the key and borrow a key from the immediate sibling. Add the median key of the sibling node to the parent.

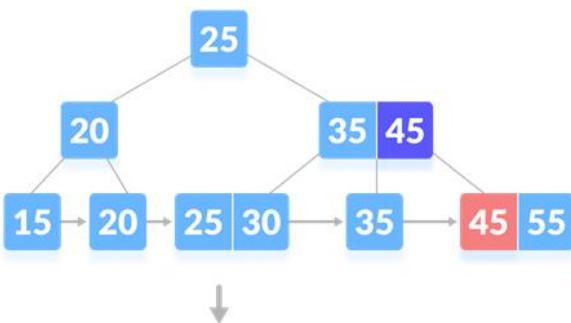


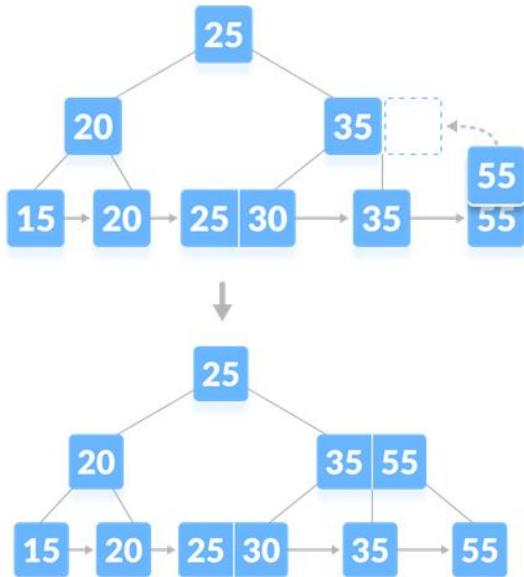
Case II

The key to be deleted is present in the internal nodes as well. Then we have to remove them from the internal nodes as well. There are the following cases for this situation.

1. If there is more than the minimum number of keys in the node, simply delete the key from the leaf node and delete the key from the internal node as well.

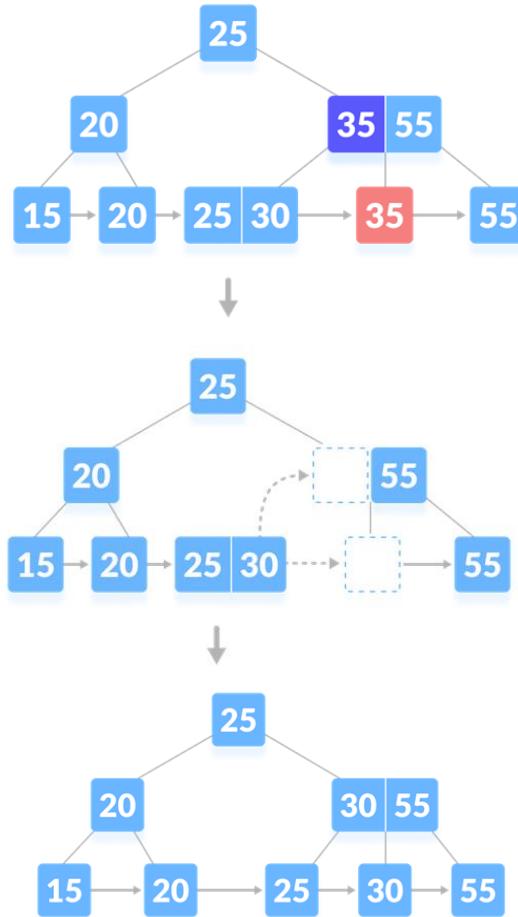
Fill the empty space in the internal node with the inorder successor.





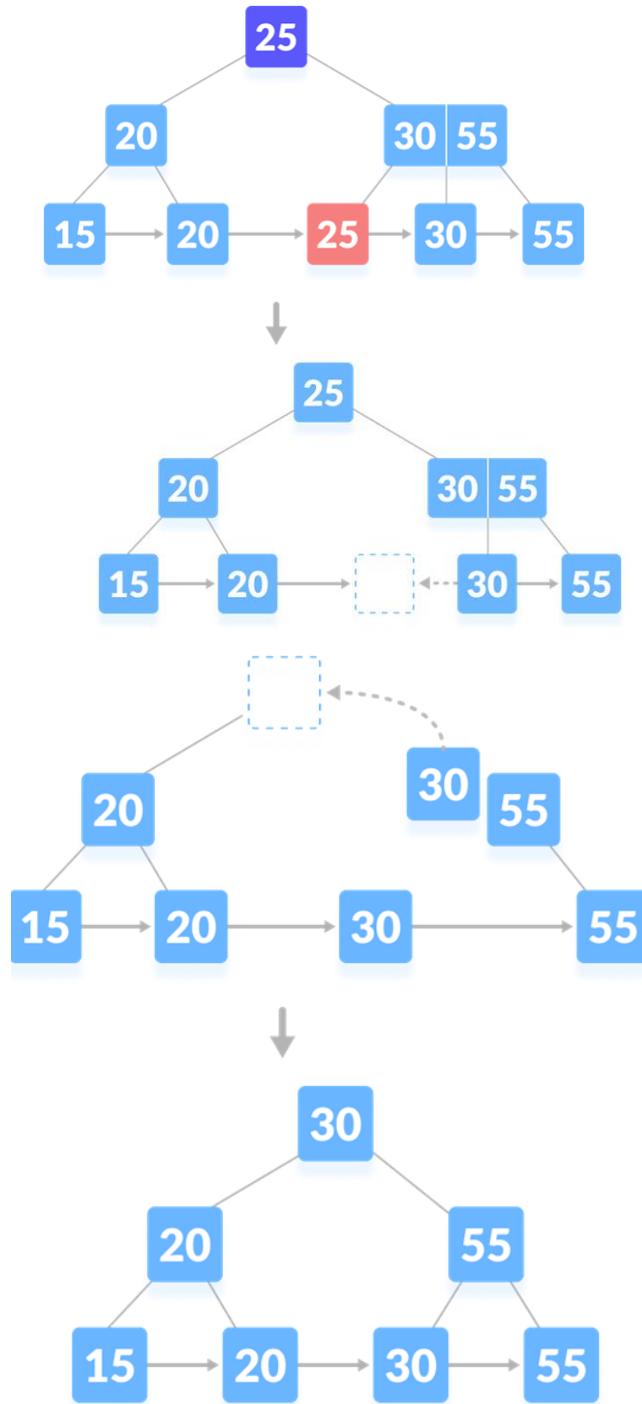
2. If there is an exact minimum number of keys in the node, then delete the key and borrow a key from its immediate sibling (through the parent).

Fill the empty space created in the index (internal node) with the borrowed key.



3. This case is similar to Case II(1) but here, empty space is generated above the immediate parent node. After deleting the key, merge the empty space with its sibling.

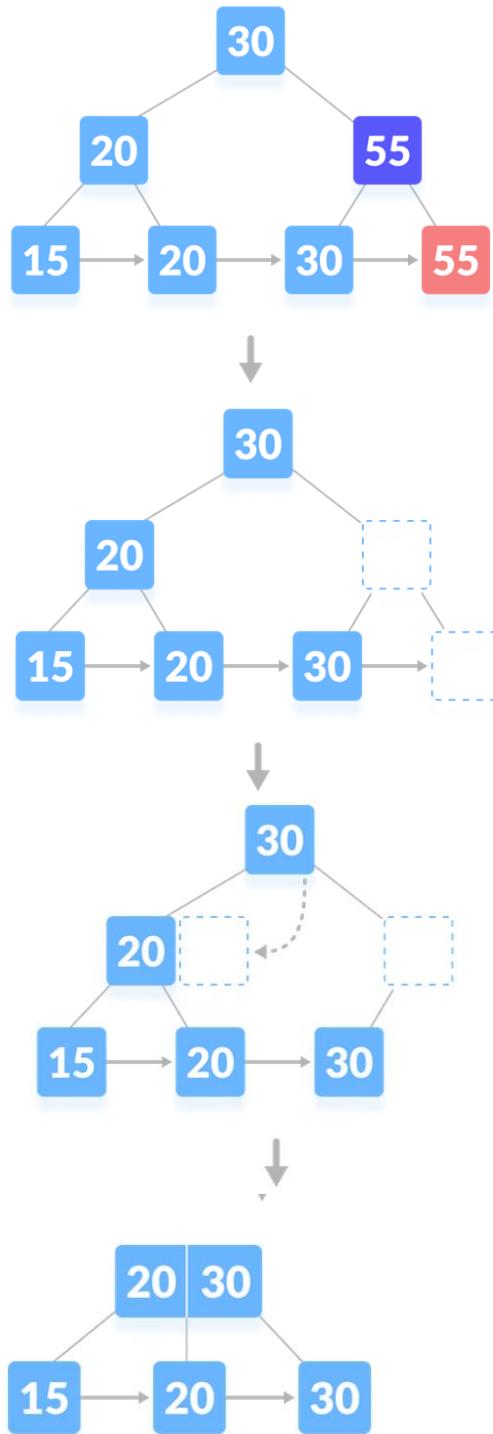
Fill the empty space in the grandparent node with the inorder successor.



Case III

In this case, the height of the tree gets shrunk. It is a little complicated.

Deleting 55 from the tree below leads to this condition. It can be understood in the illustrations below.



Program to implement B + Tree Deletion

```

// Deletion on a B+ Tree
#include <stdbool.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

// Default order
#define ORDER 3

typedef struct record {
    int value;
} record;

// Node
typedef struct node {
    void **pointers;
    int *keys;
    struct node *parent;
    bool is_leaf;
    int num_keys;
    struct node *next;
} node;

int order = ORDER;
node *queue = NULL;
bool verbose_output = false;

// Enqueue
void enqueue(node *new_node);

// Dequeue
node *dequeue(void);
int height(node *const root);
int pathToLeaves(node *const root, node *child);
void printLeaves(node *const root);
void printTree(node *const root);
void findAndPrint(node *const root, int key, bool verbose);
void findAndPrintRange(node *const root, int range1, int range2, bool verbose);
int findRange(node *const root, int key_start, int key_end, bool verbose,
             int returned_keys[], void *returned_pointers[]);
node *findLeaf(node *const root, int key, bool verbose);
record *find(node *root, int key, bool verbose, node **leaf_out);
int cut(int length);

record *makeRecord(int value);
node *makeNode(void);
node *makeLeaf(void);
int getLeftIndex(node *parent, node *left);
node *insertIntoLeaf(node *leaf, int key, record *pointer);
node *insertIntoLeafAfterSplitting(node *root, node *leaf, int key,
                                   record *pointer);
node *insertIntoNode(node *root, node *parent,
                     int left_index, int key, node *right);
node *insertIntoNodeAfterSplitting(node *root, node *parent,
                                   int left_index,
                                   int key, node *right);
node *insertIntoParent(node *root, node *left, int key, node *right);
node *insertIntoNewRoot(node *left, int key, node *right);

```

```

node *startNewTree(int key, record *pointer);
node *insert(node *root, int key, int value);

// Enqueue
void enqueue(node *new_node) {
    node *c;
    if (queue == NULL) {
        queue = new_node;
        queue->next = NULL;
    } else {
        c = queue;
        while (c->next != NULL) {
            c = c->next;
        }
        c->next = new_node;
        new_node->next = NULL;
    }
}

// Dequeue
node *dequeue(void) {
    node *n = queue;
    queue = queue->next;
    n->next = NULL;
    return n;
}

// Print the leaves
void printLeaves(node *const root) {
    if (root == NULL) {
        printf("Empty tree.\n");
        return;
    }
    int i;
    node *c = root;
    while (!c->is_leaf)
        c = c->pointers[0];
    while (true) {
        for (i = 0; i < c->num_keys; i++) {
            if (verbose_output)
                printf("%p ", c->pointers[i]);
            printf("%d ", c->keys[i]);
        }
        if (verbose_output)
            printf("%p ", c->pointers[order - 1]);
        if (c->pointers[order - 1] != NULL) {
            printf(" | ");
            c = c->pointers[order - 1];
        } else
            break;
    }
    printf("\n");
}

// Calculate height
int height(node *const root) {
    int h = 0;
    node *c = root;
    while (!c->is_leaf) {
        c = c->pointers[0];
        h++;
        while (c->next != NULL)
            c = c->next;
    }
    return h;
}

```

```

        h++;
    }
    return h;
}

// Get path to root
int pathToLeaves(node *const root, node *child) {
    int length = 0;
    node *c = child;
    while (c != root) {
        c = c->parent;
        length++;
    }
    return length;
}

// Print the tree
void printTree(node *const root) {
    node *n = NULL;
    int i = 0;
    int rank = 0;
    int new_rank = 0;

    if (root == NULL) {
        printf("Empty tree.\n");
        return;
    }
    queue = NULL;
    enqueue(root);
    while (queue != NULL) {
        n = dequeue();
        if (n->parent != NULL && n == n->parent->pointers[0]) {
            new_rank = pathToLeaves(root, n);
            if (new_rank != rank) {
                rank = new_rank;
                printf("\n");
            }
        }
        if (verbose_output)
            printf("(%p)", n);
        for (i = 0; i < n->num_keys; i++) {
            if (verbose_output)
                printf("%p ", n->pointers[i]);
            printf("%d ", n->keys[i]);
        }
        if (!n->is_leaf)
            for (i = 0; i <= n->num_keys; i++)
                enqueue(n->pointers[i]);
        if (verbose_output) {
            if (n->is_leaf)
                printf("%p ", n->pointers[order - 1]);
            else
                printf("%p ", n->pointers[n->num_keys]);
        }
        printf(" | ");
    }
    printf("\n");
}

```

```

// Find the node and print it
void findAndPrint(node *const root, int key, bool verbose) {
    node *leaf = NULL;
    record *r = find(root, key, verbose, NULL);
    if (r == NULL)
        printf("Record not found under key %d.\n", key);
    else
        printf("Record at %p -- key %d, value %d.\n",
               r, key, r->value);
}

// Find and print the range
void findAndPrintRange(node *const root, int key_start, int key_end,
                      bool verbose) {
    int i;
    int array_size = key_end - key_start + 1;
    int returned_keys[array_size];
    void *returned_pointers[array_size];
    int num_found = findRange(root, key_start, key_end, verbose,
                               returned_keys, returned_pointers);
    if (!num_found)
        printf("None found.\n");
    else {
        for (i = 0; i < num_found; i++)
            printf("Key: %d  Location: %p  Value: %d\n",
                   returned_keys[i],
                   returned_pointers[i],
                   ((record *)
                    returned_pointers[i])
                   ->value);
    }
}

// Find the range
int findRange(node *const root, int key_start, int key_end, bool verbose,
              int returned_keys[], void *returned_pointers[]) {
    int i, num_found;
    num_found = 0;
    node *n = findLeaf(root, key_start, verbose);
    if (n == NULL)
        return 0;
    for (i = 0; i < n->num_keys && n->keys[i] < key_start; i++)
        ;
    if (i == n->num_keys)
        return 0;
    while (n != NULL) {
        for (; i < n->num_keys && n->keys[i] <= key_end; i++) {
            returned_keys[num_found] = n->keys[i];
            returned_pointers[num_found] = n->pointers[i];
            num_found++;
        }
        n = n->pointers[order - 1];
        i = 0;
    }
    return num_found;
}

// Find the leaf
node *findLeaf(node *const root, int key, bool verbose) {

```

```

if (root == NULL) {
    if (verbose)
        printf("Empty tree.\n");
    return root;
}
int i = 0;
node *c = root;
while (!c->is_leaf) {
    if (verbose) {
        printf("[");
        for (i = 0; i < c->num_keys - 1; i++)
            printf("%d ", c->keys[i]);
        printf("%d] ", c->keys[i]);
    }
    i = 0;
    while (i < c->num_keys) {
        if (key >= c->keys[i])
            i++;
        else
            break;
    }
    if (verbose)
        printf("%d ->\n", i);
    c = (node *)c->pointers[i];
}
if (verbose) {
    printf("Leaf [");
    for (i = 0; i < c->num_keys - 1; i++)
        printf("%d ", c->keys[i]);
    printf("%d] ->\n", c->keys[i]);
}
return c;
}

record *find(node *root, int key, bool verbose, node **leaf_out) {
    if (root == NULL) {
        if (leaf_out != NULL) {
            *leaf_out = NULL;
        }
        return NULL;
    }

    int i = 0;
    node *leaf = NULL;

    leaf = findLeaf(root, key, verbose);

    for (i = 0; i < leaf->num_keys; i++)
        if (leaf->keys[i] == key)
            break;
    if (leaf_out != NULL) {
        *leaf_out = leaf;
    }
    if (i == leaf->num_keys)
        return NULL;
    else
        return (record *)leaf->pointers[i];
}

int cut(int length) {

```

```

if (length % 2 == 0)
    return length / 2;
else
    return length / 2 + 1;
}

record *makeRecord(int value) {
    record *new_record = (record *)malloc(sizeof(record));
    if (new_record == NULL) {
        perror("Record creation.");
        exit(EXIT_FAILURE);
    } else {
        new_record->value = value;
    }
    return new_record;
}

node *makeNode(void) {
    node *new_node;
    new_node = malloc(sizeof(node));
    if (new_node == NULL) {
        perror("Node creation.");
        exit(EXIT_FAILURE);
    }
    new_node->keys = malloc((order - 1) * sizeof(int));
    if (new_node->keys == NULL) {
        perror("New node keys array.");
        exit(EXIT_FAILURE);
    }
    new_node->pointers = malloc(order * sizeof(void *));
    if (new_node->pointers == NULL) {
        perror("New node pointers array.");
        exit(EXIT_FAILURE);
    }
    new_node->is_leaf = false;
    new_node->num_keys = 0;
    new_node->parent = NULL;
    new_node->next = NULL;
    return new_node;
}

node *makeLeaf(void) {
    node *leaf = makeNode();
    leaf->is_leaf = true;
    return leaf;
}

int getLeftIndex(node *parent, node *left) {
    int left_index = 0;
    while (left_index <= parent->num_keys &&
           parent->pointers[left_index] != left)
        left_index++;
    return left_index;
}

node *insertIntoLeaf(node *leaf, int key, record *pointer) {
    int i, insertion_point;

    insertion_point = 0;
    while (insertion_point < leaf->num_keys && leaf->keys[insertion_point] < key)

```

```

insertion_point++;

for (i = leaf->num_keys; i > insertion_point; i--) {
    leaf->keys[i] = leaf->keys[i - 1];
    leaf->pointers[i] = leaf->pointers[i - 1];
}
leaf->keys[insertion_point] = key;
leaf->pointers[insertion_point] = pointer;
leaf->num_keys++;
return leaf;
}

node *insertIntoLeafAfterSplitting(node *root, node *leaf, int key, record *pointer)
{
    node *new_leaf;
    int *temp_keys;
    void **temp_pointers;
    int insertion_index, split, new_key, i, j;

    new_leaf = makeLeaf();

    temp_keys = malloc(order * sizeof(int));
    if (temp_keys == NULL) {
        perror("Temporary keys array.");
        exit(EXIT_FAILURE);
    }

    temp_pointers = malloc(order * sizeof(void *));
    if (temp_pointers == NULL) {
        perror("Temporary pointers array.");
        exit(EXIT_FAILURE);
    }

    insertion_index = 0;
    while (insertion_index < order - 1 && leaf->keys[insertion_index] < key)
        insertion_index++;

    for (i = 0, j = 0; i < leaf->num_keys; i++, j++) {
        if (j == insertion_index)
            j++;
        temp_keys[j] = leaf->keys[i];
        temp_pointers[j] = leaf->pointers[i];
    }

    temp_keys[insertion_index] = key;
    temp_pointers[insertion_index] = pointer;

    leaf->num_keys = 0;

    split = cut(order - 1);

    for (i = 0; i < split; i++) {
        leaf->pointers[i] = temp_pointers[i];
        leaf->keys[i] = temp_keys[i];
        leaf->num_keys++;
    }

    for (i = split, j = 0; i < order; i++, j++) {
        new_leaf->pointers[j] = temp_pointers[i];
        new_leaf->keys[j] = temp_keys[i];
    }
}

```

```

    new_leaf->num_keys++;
}

free(temp_pointers);
free(temp_keys);

new_leaf->pointers[order - 1] = leaf->pointers[order - 1];
leaf->pointers[order - 1] = new_leaf;

for (i = leaf->num_keys; i < order - 1; i++)
    leaf->pointers[i] = NULL;
for (i = new_leaf->num_keys; i < order - 1; i++)
    new_leaf->pointers[i] = NULL;

new_leaf->parent = leaf->parent;
new_key = new_leaf->keys[0];

return insertIntoParent(root, leaf, new_key, new_leaf);
}

node *insertIntoNode(node *root, node *n,
                     int left_index, int key, node *right) {
int i;

for (i = n->num_keys; i > left_index; i--) {
    n->pointers[i + 1] = n->pointers[i];
    n->keys[i] = n->keys[i - 1];
}
n->pointers[left_index + 1] = right;
n->keys[left_index] = key;
n->num_keys++;
return root;
}

node *insertIntoNodeAfterSplitting(node *root, node *old_node, int left_index,
                                   int key, node *right) {
int i, j, split, k_prime;
node *new_node, *child;
int *temp_keys;
node **temp_pointers;

temp_pointers = malloc((order + 1) * sizeof(node *));
if (temp_pointers == NULL) {
    exit(EXIT_FAILURE);
}
temp_keys = malloc(order * sizeof(int));
if (temp_keys == NULL) {
    exit(EXIT_FAILURE);
}

for (i = 0, j = 0; i < old_node->num_keys + 1; i++, j++) {
    if (j == left_index + 1)
        j++;
    temp_pointers[j] = old_node->pointers[i];
}

for (i = 0, j = 0; i < old_node->num_keys; i++, j++) {
    if (j == left_index)
        j++;
    temp_keys[j] = old_node->keys[i];
}
}

```

```

}

temp_pointers[left_index + 1] = right;
temp_keys[left_index] = key;

split = cut(order);
new_node = makeNode();
old_node->num_keys = 0;
for (i = 0; i < split - 1; i++) {
    old_node->pointers[i] = temp_pointers[i];
    old_node->keys[i] = temp_keys[i];
    old_node->num_keys++;
}
old_node->pointers[i] = temp_pointers[i];
k_prime = temp_keys[split - 1];
for (++i, j = 0; i < order; i++, j++) {
    new_node->pointers[j] = temp_pointers[i];
    new_node->keys[j] = temp_keys[i];
    new_node->num_keys++;
}
new_node->pointers[j] = temp_pointers[i];
free(temp_pointers);
free(temp_keys);
new_node->parent = old_node->parent;
for (i = 0; i <= new_node->num_keys; i++) {
    child = new_node->pointers[i];
    child->parent = new_node;
}
return insertIntoParent(root, old_node, k_prime, new_node);
}

node *insertIntoParent(node *root, node *left, int key, node *right) {
    int left_index;
    node *parent;

    parent = left->parent;

    if (parent == NULL)
        return insertIntoNewRoot(left, key, right);

    left_index = getLeftIndex(parent, left);

    if (parent->num_keys < order - 1)
        return insertIntoNode(root, parent, left_index, key, right);

    return insertIntoNodeAfterSplitting(root, parent, left_index, key, right);
}

node *insertIntoNewRoot(node *left, int key, node *right) {
    node *root = makeNode();
    root->keys[0] = key;
    root->pointers[0] = left;
    root->pointers[1] = right;
    root->num_keys++;
    root->parent = NULL;
    left->parent = root;
    right->parent = root;
    return root;
}

```

```

node *startNewTree(int key, record *pointer) {
    node *root = makeLeaf();
    root->keys[0] = key;
    root->pointers[0] = pointer;
    root->pointers[order - 1] = NULL;
    root->parent = NULL;
    root->num_keys++;
    return root;
}
node *insert(node *root, int key, int value) {
    record *record_pointer = NULL;
    node *leaf = NULL;
    record_pointer = find(root, key, false, NULL);
    if (record_pointer != NULL) {
        record_pointer->value = value;
        return root;
    }
    record_pointer = makeRecord(value);
    if (root == NULL)
        return startNewTree(key, record_pointer);

    leaf = findLeaf(root, key, false);

    if (leaf->num_keys < order - 1) {
        leaf = insertIntoLeaf(leaf, key, record_pointer);
        return root;
    }

    return insertIntoLeafAfterSplitting(root, leaf, key, record_pointer);
}

int main() {
    node *root;
    char instruction;
    root = NULL;
    root = insert(root, 5, 33);
    root = insert(root, 15, 21);
    root = insert(root, 25, 31);
    root = insert(root, 35, 41);
    root = insert(root, 45, 10);
    printTree(root);
    findAndPrint(root, 15, instruction = 'a');
}

```

Output

```

25 |
15 | 35 |
5 | 15 | 25 | 35 45 |
[25] 0 ->
[15] 1 ->
Leaf [15] ->
Record at 0x559e9ac19330 -- key 15, value 21.

```

Watch Videos:-

[B+ Tree Deletion](#)

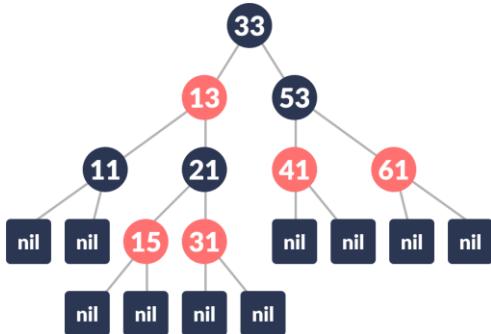
Red-Black Tree

Red-Black tree is a self-balancing binary search tree in which each node contains an extra bit for denoting the color of the node, either red or black.

A red-black tree satisfies the following properties:

- Red/Black Property: Every node is colored, either red or black.
- Root Property: The root is black.
- Leaf Property: Every leaf (NIL) is black.
- Red Property: If a red node has children then, the children are always black.
- Depth Property: For each node, any simple path from this node to any of its descendant leaf has the same black-depth (the number of black nodes).

An example of a red-black tree is:



Each node has the following attributes:

- color
- key
- leftChild
- rightChild
- parent (except root node)

How the red-black tree maintains the property of self-balancing?

The red-black color is meant for balancing the tree.

The limitations put on the node colors ensure that any simple path from the root to a leaf is not more than twice as long as any other such path. It helps in maintaining the self-balancing property of the red-black tree.

Inserting an element into a Red-Black Tree

While inserting a new node, the new node is always inserted as a RED node. After insertion of a new node, if the tree is violating the properties of the red-black tree then, we do the following operations.

- Recolor
- Rotation

Algorithm to insert a node

Following steps are followed for inserting a new element into a red-black tree:

1. Let y be the leaf (ie. `NIL`) and x be the root of the tree.
2. Check if the tree is empty (ie. whether x is `NIL`). If yes, insert `newNode` as a root node and color it black.
3. Else, repeat steps following steps until leaf (`NIL`) is reached.
4. Compare `newKey` with `rootKey`.
5. If `newKey` is greater than `rootKey`, traverse through the right subtree.
6. Else traverse through the left subtree.
7. Assign the parent of the leaf as a parent of `newNode`.
8. If `leafKey` is greater than `newKey`, make `newNode` as `rightChild`.
9. Else, make `newNode` as `leftChild`.
10. Assign `NULL` to the left and `rightChild` of `newNode`.
11. Assign RED color to `newNode`.
12. Call InsertFix-algorithm to maintain the property of red-black tree if violated.

Why newly inserted nodes are always red in a red-black tree?

This is because inserting a red node does not violate the depth property of a red-black tree. If you attach a red node to a red node, then the rule is violated but it is easier to fix this problem than the problem introduced by violating the depth property.

Algorithm to maintain red-black property after insertion

This algorithm is used for maintaining the property of a red-black tree **if the insertion of a newNode violates this property.**

- o Do the following while the parent of `newNode` p is RED.
- o If p is the left child of `grandParent` gP of z , do the following.

Case-I:

- o If the color of the right child of gP of z is RED, set the color of both the children of gP as BLACK and the color of gP as RED.
- o Assign gP to `newNode`.

Case-II:

- o Else if `newNode` is the right child of p then, assign p to `newNode`.
- o Left-Rotate `newNode`.

Case-III:

- o Set color of p as BLACK and color of gP as RED.
- o Right-Rotate gP .

Else, do the following.

- o If the color of the left child of gP of z is RED, set the color of both the children of gP as BLACK and the color of gP as RED.
- o Assign gP to `newNode`.
- o Else if `newNode` is the left child of p then, assign p to `newNode` and Right-Rotate `newNode`.
- o Set color of p as BLACK and color of gP as RED.
- o Left-Rotate gP .
- o Set the root of the tree as BLACK.

Deleting an element from a Red-Black Tree

This operation removes a node from the tree. After deleting a node, the red-black property is maintained again.

1. Algorithm to delete a node
2. Save the color of `nodeToBeDeleted` in `originalColor`.
3. If the left child of `nodeToBeDeleted` is `NULL`
4. Assign the right child of `nodeToBeDeleted` to `x`.
5. Transplant `nodeToBeDeleted` with `x`.
6. Else if the right child of `nodeToBeDeleted` is `NULL`
7. Assign the left child of `nodeToBeDeleted` into `x`.
8. Transplant `nodeToBeDeleted` with `x`.
9. Else
10. Assign the minimum of right subtree of `noteToBeDeleted` into `y`.
11. Save the color of `y` in `originalColor`.
12. Assign the `rightChild` of `y` into `x`.
13. If `y` is a child of `nodeToBeDeleted`, then set the parent of `x` as `y`.
14. Else, transplant `y` with `rightChild` of `y`.
15. Transplant `nodeToBeDeleted` with `y`.
16. Set the color of `y` with `originalColor`.
17. If the `originalColor` is `BLACK`, call `DeleteFix(x)`.

Algorithm to maintain Red-Black property after deletion

This algorithm is implemented when a black node is deleted because it violates the black depth property of the red-black tree.

This violation is corrected by assuming that node `x` (which is occupying `y`'s original position) has an extra black. This makes node `x` neither red nor black. It is either doubly black or black-and-red. This violates the red-black properties.

However, the color attribute of `x` is not changed rather the extra black is represented in `x`'s pointing to the node.

The extra black can be removed if

1. It reaches the root node.
2. If `x` points to a red-black node. In this case, `x` is colored black.
3. Suitable rotations and recoloring are performed.

The following algorithm retains the properties of a red-black tree.

1. Do the following until the `x` is not the root of the tree and the color of `x` is `BLACK`
2. If `x` is the left child of its parent then,
 - a. Assign `w` to the sibling of `x`.
 - b. If the right child of parent of `x` is `RED`,

Case-I:

- a. Set the color of the right child of the parent of `x` as `BLACK`.
- b. Set the color of the parent of `x` as `RED`.
- c. Left-Rotate the parent of `x`.
- d. Assign the `rightChild` of the parent of `x` to `w`.

- e. If the color of both the right and the `leftChild` of `w` is BLACK,

Case-II:

- a. Set the color of `w` as RED
- b. Assign the parent of `x` to `x`.
- c. Else if the color of the `rightChild` of `w` is BLACK

Case-III:

- a. Set the color of the `leftChild` of `w` as BLACK
- b. Set the color of `w` as RED
- c. Right-Rotate `w`.
- d. Assign the `rightChild` of the parent of `x` to `w`.
- e. If any of the above cases do not occur, then do the following.

Case-IV:

- a. Set the color of `w` as the color of the parent of `x`.
- b. Set the color of the parent of `x` as BLACK.
- c. Set the color of the right child of `w` as BLACK.
- d. Left-Rotate the parent of `x`.
- e. Set `x` as the root of the tree.
- f. Else the same as above with right changed to left and vice versa.
- g. Set the color of `x` as BLACK.

Program to implement Red-Balck Tree

```
// Implementing Red-Black Tree
#include <stdio.h>
#include <stdlib.h>

enum nodeColor {
    RED,
    BLACK
};

struct rbNode {
    int data, color;
    struct rbNode *link[2];
};

struct rbNode *root = NULL;

// Create a red-black tree
struct rbNode *createNode(int data) {
    struct rbNode *newnode;
    newnode = (struct rbNode *)malloc(sizeof(struct rbNode));
    newnode->data = data;
    newnode->color = RED;
    newnode->link[0] = newnode->link[1] = NULL;
    return newnode;
}

// Insert an node
void insertion(int data) {
    struct rbNode *stack[98], *ptr, *newnode, *xPtr, *yPtr;
    int dir[98], ht = 0, index;
    ptr = root;
    if (!root) {
        root = createNode(data);
        return;
    }

    stack[ht] = root;
    dir[ht++] = 0;
    while (ptr != NULL) {
        if (ptr->data == data) {
            printf("Duplicates Not Allowed!!\n");
            return;
        }
        index = (data - ptr->data) > 0 ? 1 : 0;
        stack[ht] = ptr;
        ptr = ptr->link[index];
        dir[ht++] = index;
    }
    stack[ht - 1]->link[index] = newnode = createNode(data);
    while ((ht >= 3) && (stack[ht - 1]->color == RED)) {
        if (dir[ht - 2] == 0) {
            yPtr = stack[ht - 2]->link[1];
            if (yPtr != NULL && yPtr->color == RED) {
                stack[ht - 2]->color = RED;
                stack[ht - 1]->color = yPtr->color = BLACK;
                ht = ht - 2;
            } else {
                if (dir[ht - 1] == 0) {

```

```

        yPtr = stack[ht - 1];
    } else {
        xPtr = stack[ht - 1];
        yPtr = xPtr->link[1];
        xPtr->link[1] = yPtr->link[0];
        yPtr->link[0] = xPtr;
        stack[ht - 2]->link[0] = yPtr;
    }
    xPtr = stack[ht - 2];
    xPtr->color = RED;
    yPtr->color = BLACK;
    xPtr->link[0] = yPtr->link[1];
    yPtr->link[1] = xPtr;
    if (xPtr == root) {
        root = yPtr;
    } else {
        stack[ht - 3]->link[dir[ht - 3]] = yPtr;
    }
    break;
}
} else {
yPtr = stack[ht - 2]->link[0];
if ((yPtr != NULL) && (yPtr->color == RED)) {
    stack[ht - 2]->color = RED;
    stack[ht - 1]->color = yPtr->color = BLACK;
    ht = ht - 2;
} else {
    if (dir[ht - 1] == 1) {
        yPtr = stack[ht - 1];
    } else {
        xPtr = stack[ht - 1];
        yPtr = xPtr->link[0];
        xPtr->link[0] = yPtr->link[1];
        yPtr->link[1] = xPtr;
        stack[ht - 2]->link[1] = yPtr;
    }
    xPtr = stack[ht - 2];
    yPtr->color = BLACK;
    xPtr->color = RED;
    xPtr->link[1] = yPtr->link[0];
    yPtr->link[0] = xPtr;
    if (xPtr == root) {
        root = yPtr;
    } else {
        stack[ht - 3]->link[dir[ht - 3]] = yPtr;
    }
    break;
}
}
root->color = BLACK;
}

// Delete a node
void deletion(int data) {
    struct rbNode *stack[98], *ptr, *xPtr, *yPtr;
    struct rbNode *pPtr, *qPtr, *rPtr;
    int dir[98], ht = 0, diff, i;
    enum nodeColor color;
}

```

```

if (!root) {
    printf("Tree not available\n");
    return;
}

ptr = root;
while (ptr != NULL) {
    if ((data - ptr->data) == 0)
        break;
    diff = (data - ptr->data) > 0 ? 1 : 0;
    stack[ht] = ptr;
    dir[ht++] = diff;
    ptr = ptr->link[diff];
}

if (ptr->link[1] == NULL) {
    if ((ptr == root) && (ptr->link[0] == NULL)) {
        free(ptr);
        root = NULL;
    } else if (ptr == root) {
        root = ptr->link[0];
        free(ptr);
    } else {
        stack[ht - 1]->link[dir[ht - 1]] = ptr->link[0];
    }
} else {
    xPtr = ptr->link[1];
    if (xPtr->link[0] == NULL) {
        xPtr->link[0] = ptr->link[0];
        color = xPtr->color;
        xPtr->color = ptr->color;
        ptr->color = color;

        if (ptr == root) {
            root = xPtr;
        } else {
            stack[ht - 1]->link[dir[ht - 1]] = xPtr;
        }
    }

    dir[ht] = 1;
    stack[ht++] = xPtr;
} else {
    i = ht++;
    while (1) {
        dir[ht] = 0;
        stack[ht++] = xPtr;
        yPtr = xPtr->link[0];
        if (!yPtr->link[0])
            break;
        xPtr = yPtr;
    }

    dir[i] = 1;
    stack[i] = yPtr;
    if (i > 0)
        stack[i - 1]->link[dir[i - 1]] = yPtr;

    yPtr->link[0] = ptr->link[0];
    xPtr->link[0] = yPtr->link[1];
    yPtr->link[1] = ptr->link[1];
}

```

```

if (ptr == root) {
    root = yPtr;
}

color = yPtr->color;
yPtr->color = ptr->color;
ptr->color = color;
}

}

if (ht < 1)
    return;

if (ptr->color == BLACK) {
    while (1) {
        pPtr = stack[ht - 1]->link[dir[ht - 1]];
        if (pPtr && pPtr->color == RED) {
            pPtr->color = BLACK;
            break;
        }

        if (ht < 2)
            break;

        if (dir[ht - 2] == 0) {
            rPtr = stack[ht - 1]->link[1];

            if (!rPtr)
                break;

            if (rPtr->color == RED) {
                stack[ht - 1]->color = RED;
                rPtr->color = BLACK;
                stack[ht - 1]->link[1] = rPtr->link[0];
                rPtr->link[0] = stack[ht - 1];

                if (stack[ht - 1] == root) {
                    root = rPtr;
                } else {
                    stack[ht - 2]->link[dir[ht - 2]] = rPtr;
                }
                dir[ht] = 0;
                stack[ht] = stack[ht - 1];
                stack[ht - 1] = rPtr;
                ht++;
            }

            rPtr = stack[ht - 1]->link[1];
        }

        if ((!rPtr->link[0] || rPtr->link[0]->color == BLACK) &&
            (!rPtr->link[1] || rPtr->link[1]->color == BLACK)) {
            rPtr->color = RED;
        } else {
            if (!rPtr->link[1] || rPtr->link[1]->color == BLACK) {
                qPtr = rPtr->link[0];
                rPtr->color = RED;
                qPtr->color = BLACK;
                rPtr->link[0] = qPtr->link[1];
                qPtr->link[1] = rPtr;
            }
        }
    }
}

```

```

    rPtr = stack[ht - 1]->link[1] = qPtr;
}
rPtr->color = stack[ht - 1]->color;
stack[ht - 1]->color = BLACK;
rPtr->link[1]->color = BLACK;
stack[ht - 1]->link[1] = rPtr->link[0];
rPtr->link[0] = stack[ht - 1];
if (stack[ht - 1] == root) {
    root = rPtr;
} else {
    stack[ht - 2]->link[dir[ht - 2]] = rPtr;
}
break;
}
} else {
rPtr = stack[ht - 1]->link[0];
if (!rPtr)
    break;

if (rPtr->color == RED) {
    stack[ht - 1]->color = RED;
    rPtr->color = BLACK;
    stack[ht - 1]->link[0] = rPtr->link[1];
    rPtr->link[1] = stack[ht - 1];

    if (stack[ht - 1] == root) {
        root = rPtr;
    } else {
        stack[ht - 2]->link[dir[ht - 2]] = rPtr;
    }
    dir[ht] = 1;
    stack[ht] = stack[ht - 1];
    stack[ht - 1] = rPtr;
    ht++;
}

rPtr = stack[ht - 1]->link[0];
}
if ((!rPtr->link[0] || rPtr->link[0]->color == BLACK) &&
    (!rPtr->link[1] || rPtr->link[1]->color == BLACK)) {
    rPtr->color = RED;
} else {
    if (!rPtr->link[0] || rPtr->link[0]->color == BLACK) {
        qPtr = rPtr->link[1];
        rPtr->color = RED;
        qPtr->color = BLACK;
        rPtr->link[1] = qPtr->link[0];
        qPtr->link[0] = rPtr;
        rPtr = stack[ht - 1]->link[0] = qPtr;
    }
    rPtr->color = stack[ht - 1]->color;
    stack[ht - 1]->color = BLACK;
    rPtr->link[0]->color = BLACK;
    stack[ht - 1]->link[0] = rPtr->link[1];
    rPtr->link[1] = stack[ht - 1];
    if (stack[ht - 1] == root) {
        root = rPtr;
    } else {
        stack[ht - 2]->link[dir[ht - 2]] = rPtr;
    }
}
break;
}

```

```

        }
    }
    ht--;
}
}

// Print the inorder traversal of the tree
void inorderTraversal(struct rbNode *node) {
    if (node) {
        inorderTraversal(node->link[0]);
        printf("%d ", node->data);
        inorderTraversal(node->link[1]);
    }
    return;
}

int main()
{
    int ch, data;
    while (1) {
        printf("1. Insertion\t2. Deletion\n");
        printf("3. Traverse\t4. Exit");
        printf("\nEnter your choice:");
        scanf("%d", &ch);
        switch (ch) {
            case 1:
                printf("Enter the element to insert:");
                scanf("%d", &data);
                insertion(data);
                break;
            case 2:
                printf("Enter the element to delete:");
                scanf("%d", &data);
                deletion(data);
                break;
            case 3:
                inorderTraversal(root);
                printf("\n");
                break;
            case 4:
                exit(0);
            default:
                printf("Not available\n");
                break;
        }
        printf("\n");
    }
    return 0;
}

```

Watch Videos:-

[Introduction to Red Black trees](#)

Red-Black Tree Insertion

Red-Black tree is a self-balancing binary search tree in which each node contains an extra bit for denoting the color of the node, either red or black.

While inserting a new node, the new node is always inserted as a RED node. After insertion of a new node, if the tree is violating the properties of the red-black tree then, we do the following operations.

- Recolor
- Rotation

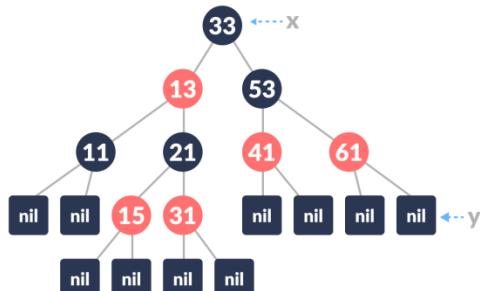
Algorithm to Insert a New Node

Following steps are followed for inserting a new element into a red-black tree:

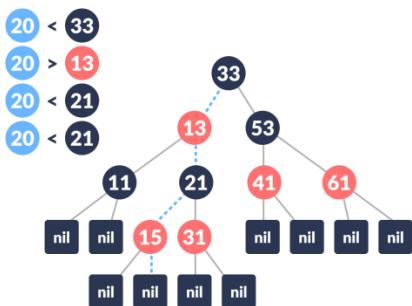
1. The `newNode` be:



2. Let y be the leaf (ie. `NIL`) and x be the root of the tree. The new node is inserted in the following tree.



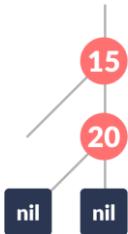
3. Check if the tree is empty (ie. whether x is `NIL`). If yes, insert `newNode` as a root node and color it black.
4. Else, repeat steps following steps until leaf (`NIL`) is reached.
 - a. Compare `newKey` with `rootKey`.
 - b. If `newKey` is greater than `rootKey`, traverse through the right subtree.
 - c. Else traverse through the left subtree.



5. Assign the parent of the leaf as parent of `newNode`.
6. If `leafKey` is greater than `newKey`, make `newNode` as `rightChild`.
7. Else, make `newNode` as `leftChild`.



8. Assign `NULL` to the left and `rightChild` of `newNode`.
9. Assign RED color to `newNode`.
10. Set the color of the `newNode` red and assign null to the children



11. Call `InsertFix-algorithm` to maintain the property of red-black tree if violated.

Why newly inserted nodes are always red in a red-black tree?

This is because inserting a red node does not violate the depth property of a red-black tree.

If you attach a red node to a red node, then the rule is violated but it is easier to fix this problem than the problem introduced by violating the depth property.

Algorithm to Maintain Red-Black Property After Insertion

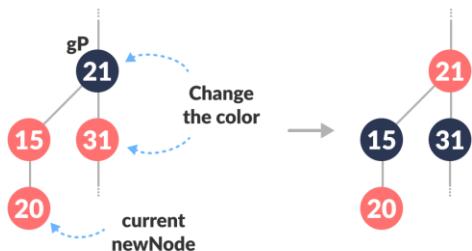
This algorithm is used for maintaining the property of a red-black tree if insertion of a `newNode` violates this property.

1. Do the following until the parent of `newNode` `p` is RED.
2. If `p` is the left child of `grandParent` `gP` of `newNode`, do the following.

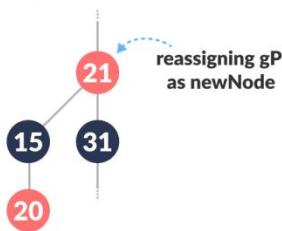
Case-I:

- a. If the color of the right child of `gP` of `newNode` is RED, set the color of both the children of `gP` as BLACK and the color of `gP` as RED.

Case-I(a)

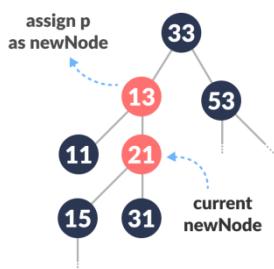


- b. Assign `gP` to `newNode`. Reassigning `newNode`

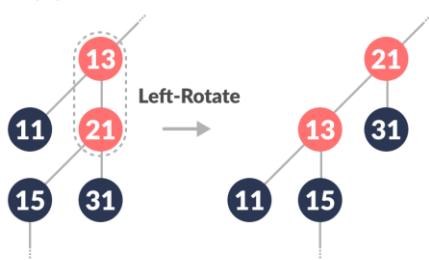
Case-I(b)**Case-II:**

- c. (Before moving on to this step, while loop is checked. If conditions are not satisfied, it the loop is broken.)

Else if `newNode` is the right child of `p` then, assign `p` to `newNode`.Assigning parent of `newNode` as `newNode`

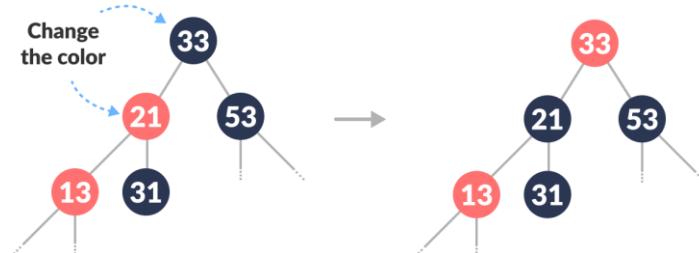
Case-II(a)

- d. Left-Rotate `newNode`.

Case-II(b)**Case-III:**

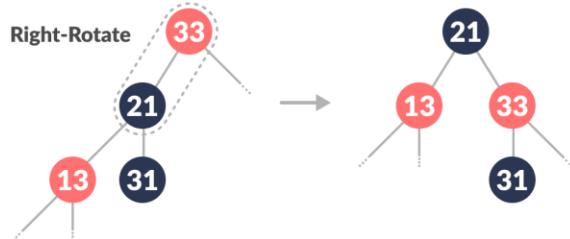
- e. (Before moving on to this step, while loop is checked. If conditions are not satisfied, it the loop is broken.)

Set color of `p` as BLACK and color of `gP` as RED.

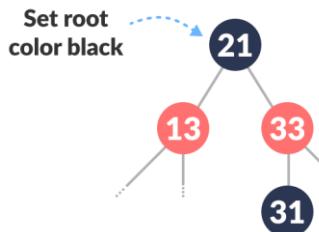
Case-III(a)

f. Right-Rotate gP .

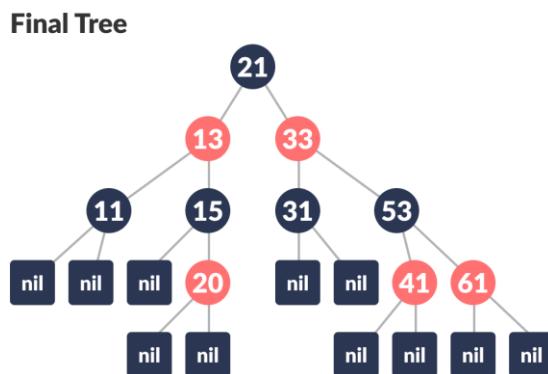
Case-III(b)



3. Else, do the following.
 - a. If the color of the left child of gP of z is RED, set the color of both the children of gP as BLACK and the color of gP as RED.
 - b. Assign gP to newNode .
 - c. Else if newNode is the left child of p then, assign p to newNode and Right-Rotate newNode .
 - d. Set color of p as BLACK and color of gP as RED.
 - e. Left-Rotate gP .
4. (This step is performed after coming out of the while loop.)
Set the root of the tree as BLACK.



The final tree look like this:



Watch Videos:-

[Red Black Tree Insertion](#)

Red-Black Tree Deletion

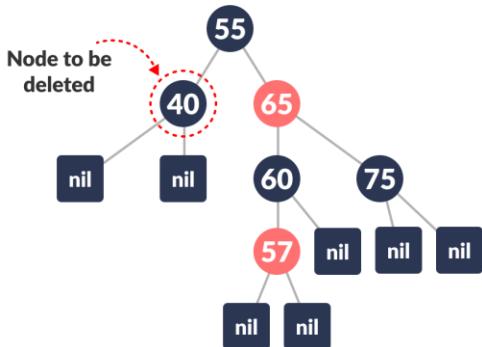
Red-Black tree is a self-balancing binary search tree in which each node contains an extra bit for denoting the color of the node, either red or black.

Deleting a node may or may not disrupt the red-black properties of a red-black tree. If this action violates the red-black properties, then a fixing algorithm is used to regain the red-black properties.

Deleting an element from a Red-Black Tree

This operation removes a node from the tree. After deleting a node, the red-black property is maintained again.

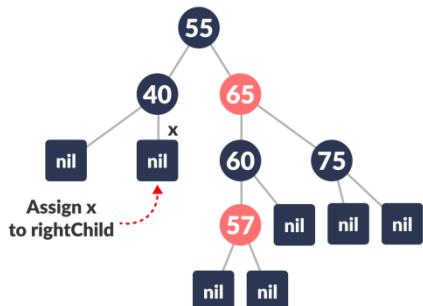
1. Let the nodeToBeDeleted be:



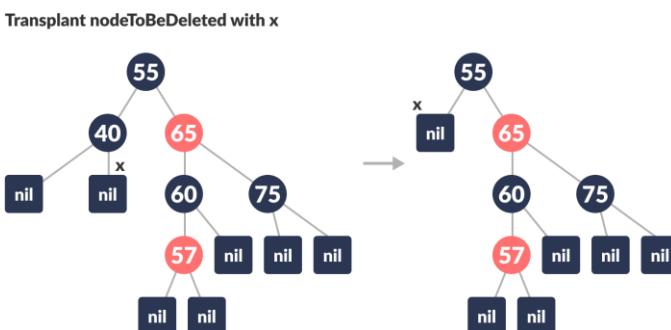
2. Save the color of nodeToBeDeleted in originalColor.

originalColor = Black

3. If the left child of nodeToBeDeleted is NULL
a. Assign the right child of nodeToBeDeleted to x.



- b. Transplant nodeToBeDeleted with x.



4. Else if the right child of nodeToBeDeleted is NULL
 - a. Assign the left child of nodeToBeDeleted into x.
 - b. Transplant nodeToBeDeleted with x.
5. Else
 - a. Assign the minimum of right subtree of nodeToBeDeleted into y.
 - b. Save the color of y in originalColor.
 - c. Assign the rightChild of y into x.
 - d. If y is a child of nodeToBeDeleted, then set the parent of x as y.
 - e. Else, transplant y with rightChild of y.
 - f. Transplant nodeToBeDeleted with y.
 - g. Set the color of y with originalColor.
6. If the originalColor is BLACK, call DeleteFix(x).

Algorithm to maintain Red-Black property after deletion

This algorithm is implemented when a black node is deleted because it violates the black depth property of the red-black tree.

This violation is corrected by assuming that node x (which is occupying y's original position) has an extra black. This makes node x neither red nor black. It is either doubly black or black-and-red. This violates the red-black properties.

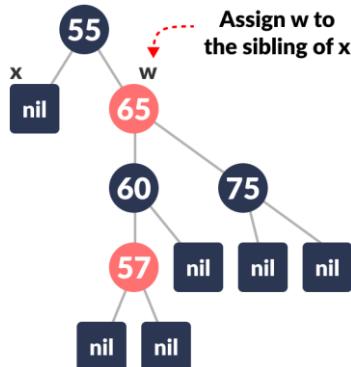
However, the color attribute of x is not changed rather the extra black is represented in x's pointing to the node.

The extra black can be removed if

- o It reaches the root node.
- o If x points to a red-black node. In this case, x is colored black.
- o Suitable rotations and recolorings are performed.

Following algorithm retains the properties of a red-black tree.

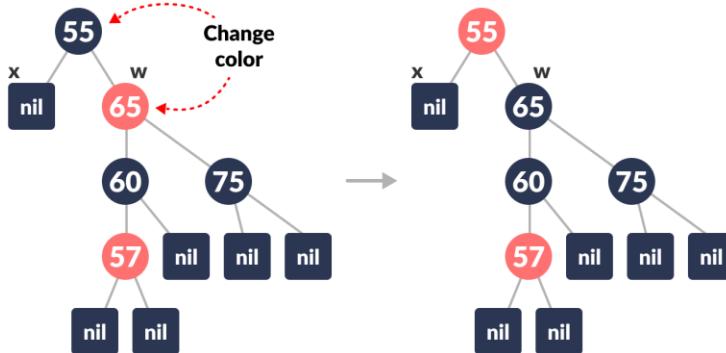
1. Do the following until the x is not the root of the tree and the color of x is BLACK
2. If x is the left child of its parent then,
 - a. Assign w to the sibling of x.



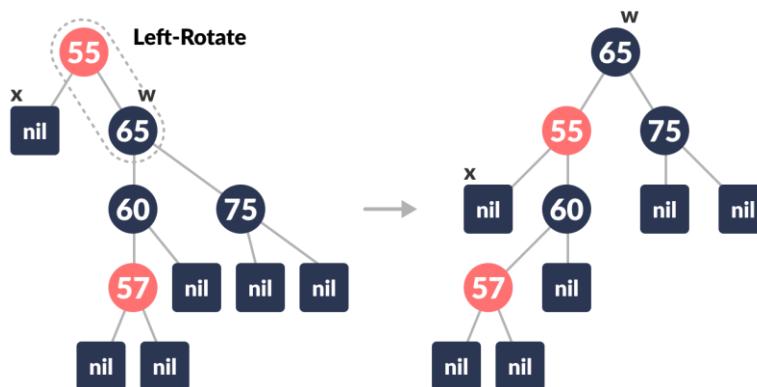
- b. If the sibling of x is RED,

Case-I:

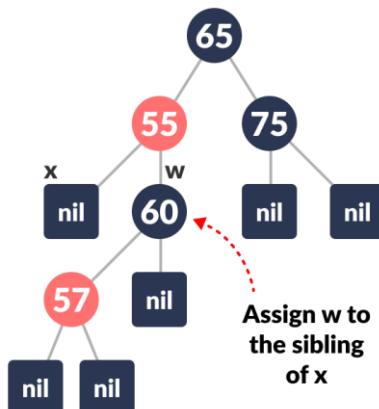
- Set the color of the right child of the parent of x as BLACK.
- Set the color of the parent of x as RED.



- Left-Rotate the parent of x



- Assign the `rightChild` of the parent of x to w .



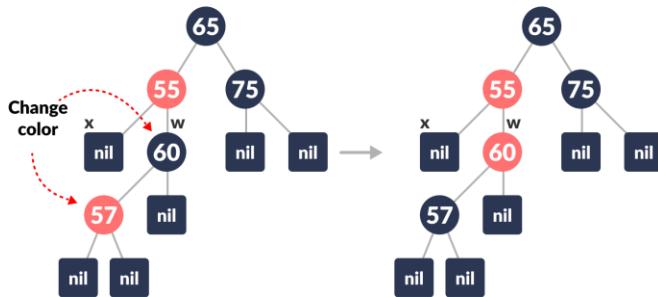
- If the color of both the right and the `leftChild` of w is BLACK,

Case-II:

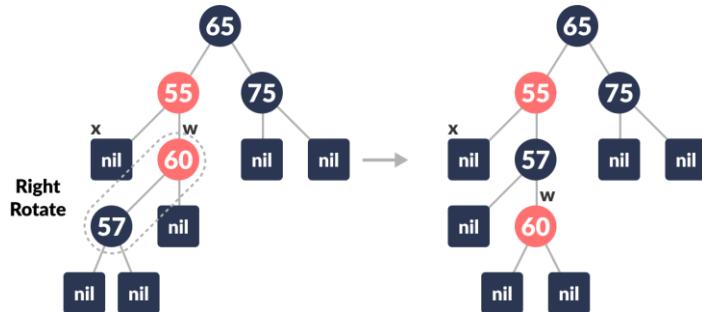
- Set the color of w as RED
 - Assign the parent of x to x .
- Else if the color of the `rightChild` of w is BLACK

Case-III:

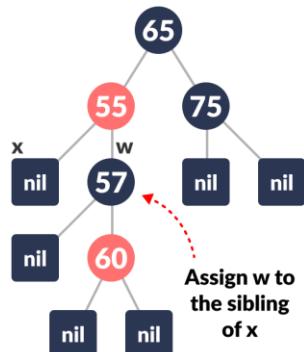
- Set the color of the `leftChild` of `w` as BLACK
- Set the color of `w` as RED



- Right-Rotate `w`.



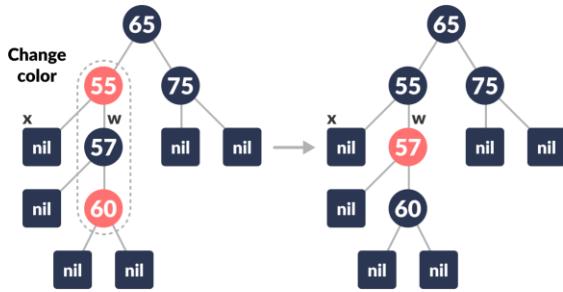
- Assign the `rightChild` of the parent of `x` to `w`.



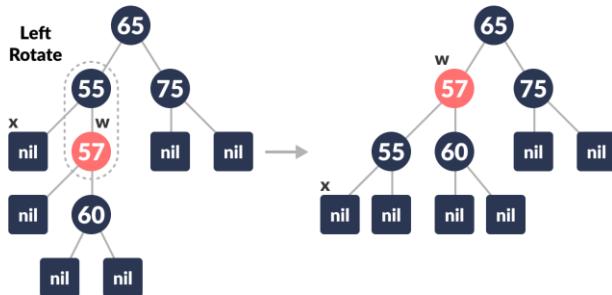
- If any of the above cases do not occur, then do the following.

Case-IV:

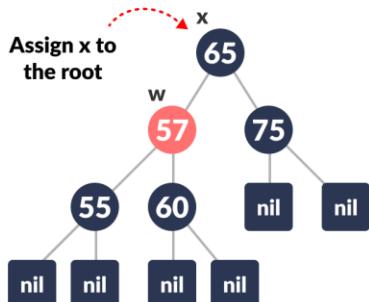
- Set the color of `w` as the color of the parent of `x`.
- Set the color of the parent of parent of `x` as BLACK.
- Set the color of the right child of `w` as BLACK.



d. Left-Rotate the parent of x .

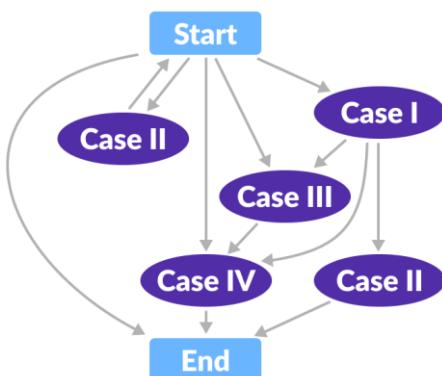


e. Set x as the root of the tree.



3. Else same as above with right changed to left and vice versa.
4. Set the color of x as BLACK.

The workflow of the above cases can be understood with the help of the flowchart below.



Watch Videos:-

[Red Black Tree deletion](#)

Splay Tree

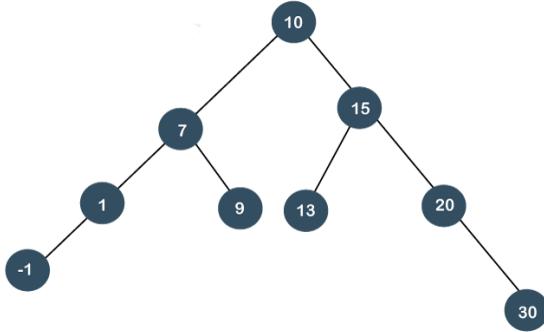
Splay trees are the self-balancing or self-adjusted binary search trees. We can say that the splay trees are the variants of the binary search trees.

A splay tree is a self-balancing tree, but AVL and Red-Black trees are also self-balancing trees then. What makes the splay tree unique two trees. It has one extra property that makes it unique is splaying.

A splay tree contains the same operations as a Binary search tree, i.e., Insertion, deletion and searching, but it also contains one more operation, i.e., splaying. So. all the operations in the splay tree are followed by splaying.

Splay trees are not strictly balanced trees, but they are roughly balanced trees. Let's understand the search operation in the splay-tree.

Suppose we want to search 7 element in the tree, which is shown below:



To search any element in the splay tree, first, we will perform the standard binary search tree operation. As 7 is less than 10 so we will come to the left of the root node. After performing the search operation, we need to perform splaying.

Here **splaying means that the operation that we are performing on any element should become the root node after performing some rearrangements.** The rearrangement of the tree will be done through the rotations.

Rotations

There are six types of rotations used for splaying:

- Zig rotation (Right rotation)
- Zag rotation (Left rotation)
- Zig zag (Zig followed by zag)
- Zag zig (Zag followed by zig)
- Zig zig (two right rotations)
- Zag zag (two left rotations)

Factors required for selecting a type of rotation

The following are the factors used for selecting a type of rotation:

- Does the node which we are trying to rotate have a grandparent?
- Is the node left or right child of the parent?
- Is the node left or right child of the grandparent?

Cases for the Rotations

Case 1: If the node does not have a grand-parent, and if it is the right child of the parent, then we carry out the left rotation; otherwise, the right rotation is performed.

Case 2: If the node has a grandparent, then based on the following scenarios; the rotation would be performed:

- **Scenario 1:** If the node is the right of the parent and the parent is also right of its parent, then zig zig right right rotation is performed.
- **Scenario 2:** If the node is left of a parent, but the parent is right of its parent, then zig zag right left rotation is performed.
- **Scenario 3:** If the node is right of the parent and the parent is right of its parent, then zig zig left left rotation is performed.
- **Scenario 4:** If the node is right of a parent, but the parent is left of its parent, then zig zag right-left rotation is performed.

Now, let's understand the above rotations with examples.

To rearrange the tree, we need to perform some rotations. The following are the types of rotations in the splay tree:

Zig rotations

The zig rotations are used when the item to be searched is either a root node or the child of a root node (i.e., left or the right child).

The following are the cases that can exist in the splay tree while searching:

Case 1: If the search item is a root node of the tree.

Case 2: If the search item is a child of the root node, then the two scenarios will be there

1. If the child is a left child, the right rotation would be performed, known as a zig right rotation.
2. If the child is a right child, the left rotation would be performed, known as a zig left rotation.

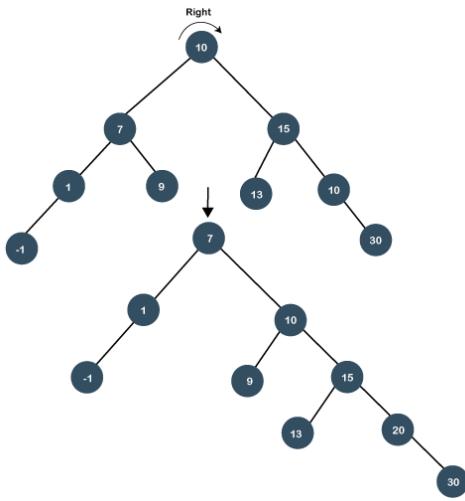
Let's look at the above two scenarios through an example.

Consider the below example:

In the above example, we have to search 7 element in the tree. We will follow the below steps:

Step 1: First, we compare 7 with a root node. As 7 is less than 10, so it is a left child of the root node.

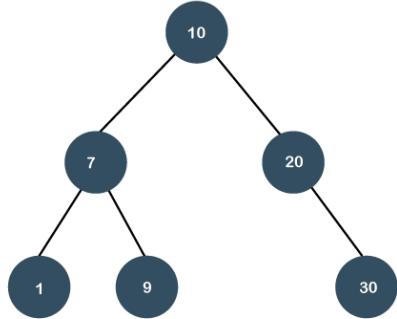
Step 2: Once the element is found, we will perform splaying. The right rotation is performed so that 7 becomes the root node of the tree, as shown below:



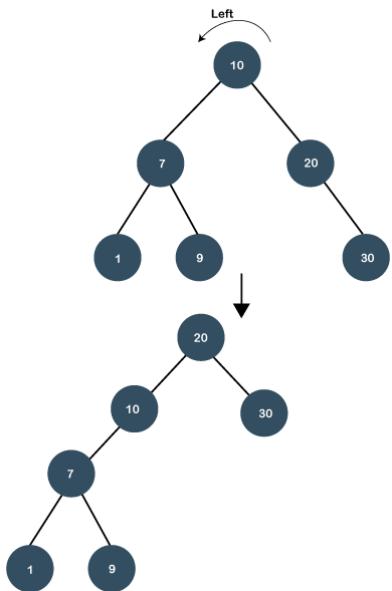
Let's consider another example.

In the above example, we have to **search 20 element** in the tree. We will follow the below steps:

Step 1: First, we compare 20 with a root node. As 20 is greater than the root node, so it is a right child of the root node.



Step 2: Once the element is found, we will perform splaying. The left rotation is performed so that 20 element becomes the root node of the tree.



Zig zig rotations

When the item to be searched is having a parent as well as a grandparent. In this case, we have to perform four rotations for splaying.

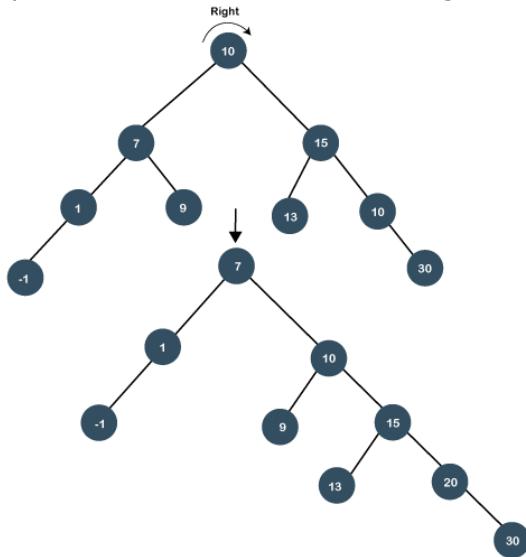
Let's understand this case through an example.

Suppose we have to search 1 element in the tree, which is shown below:

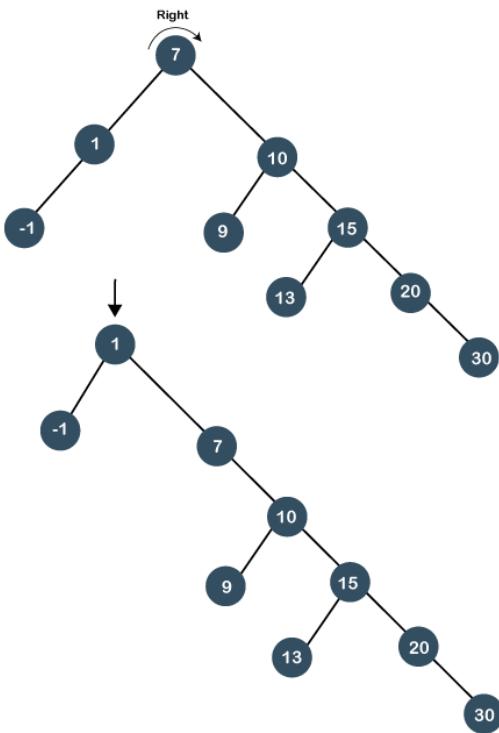
Step 1: First, we have to perform a standard BST searching operation in order to search the 1 element. As 1 is less than 10 and 7, so it will be at the left of the node 7. Therefore, element 1 is having a parent, i.e., 7 as well as a grandparent, i.e., 10.

Step 2: In this step, we have to perform splaying. We need to make node 1 as a root node with the help of some rotations. In this case, we cannot simply perform a zig or zag rotation; we have to implement zig zig rotation.

In order to make node 1 as a root node, we need to perform two right rotations known as zig zig rotations. When we perform the right rotation then 10 will move downwards, and node 7 will come upwards as shown in the below figure:



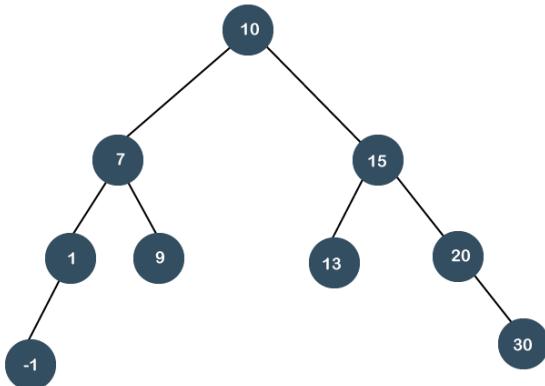
Again, we will perform zig right rotation, node 7 will move downwards, and node 1 will come upwards as shown below:



As we observe in the above figure that node 1 has become the root node of the tree; therefore, the searching is completed.

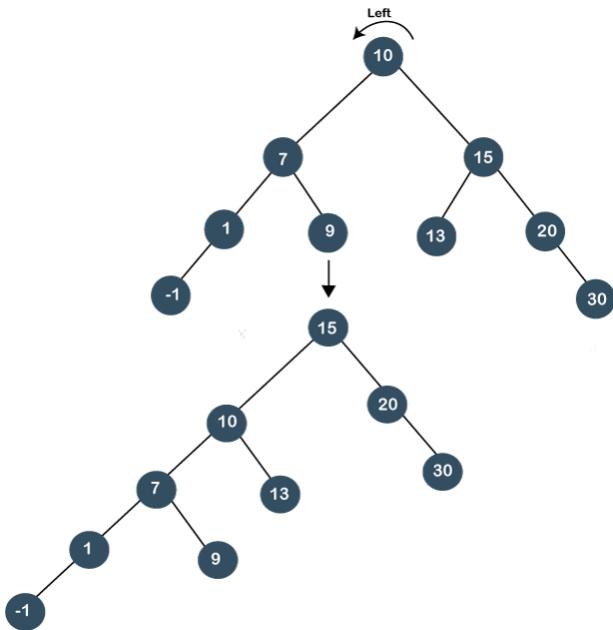
Suppose we want to search 20 in the below tree.

In order to search 20, we need to perform two left rotations. Following are the steps required to search 20 node:

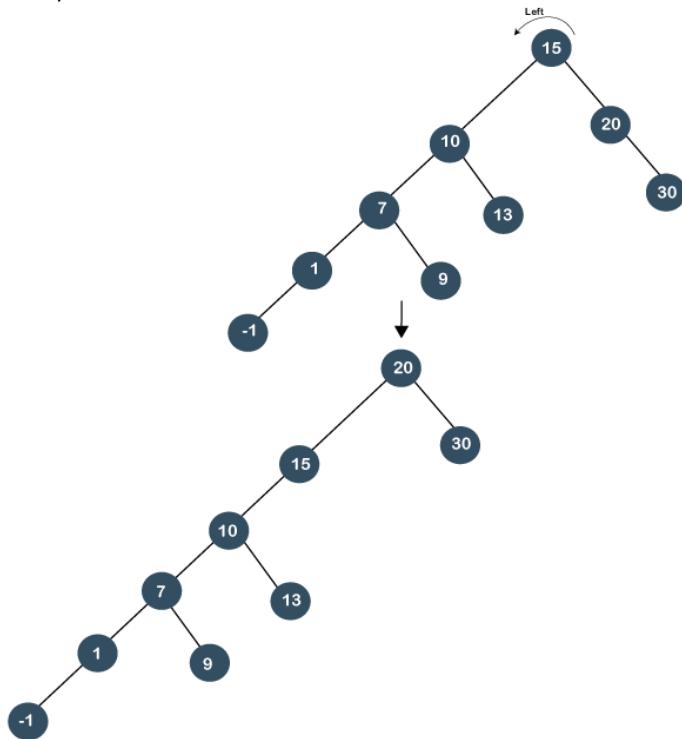


Step 1: First, we perform the standard BST searching operation. As 20 is greater than 10 and 15, so it will be at the right of node 15.

Step 2: The second step is to perform splaying. In this case, two left rotations would be performed. In the first rotation, node 10 will move downwards, and node 15 would move upwards as shown below:



In the second left rotation, node 15 will move downwards, and node 20 becomes the root node of the tree, as shown below:



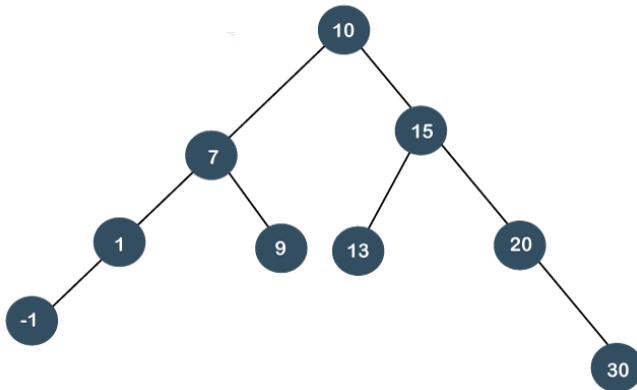
As we have observed that two left rotations are performed; so it is known as a zig zig left rotation.

Zig zag rotations

Till now, we have read that both parent and grandparent are either in RR or LL relationship. Now, we will see the RL or LR relationship between the parent and the grandparent.

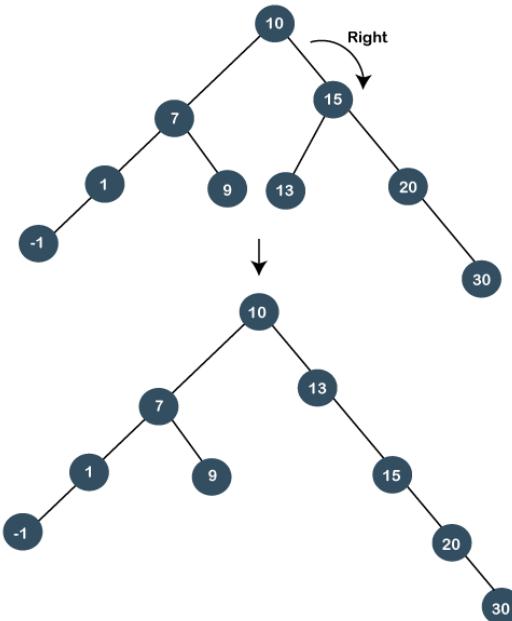
Let's understand this case through an example.

Suppose we want to search 13 element in the tree which is shown below:

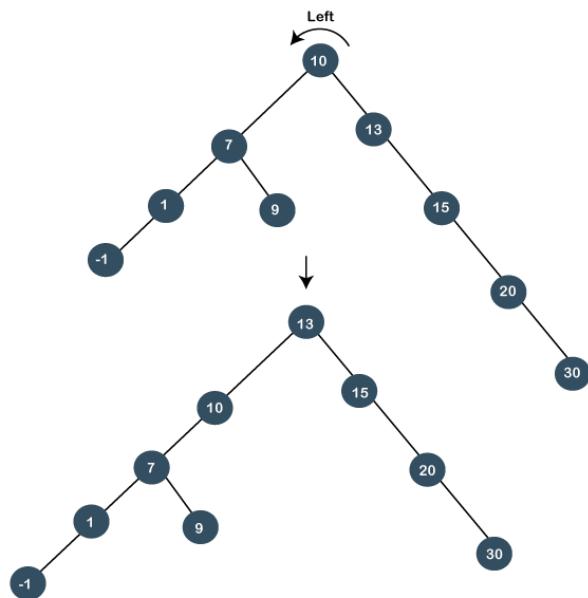


Step 1: First, we perform standard BST searching operation. As 13 is greater than 10 but less than 15, so node 13 will be the left child of node 15.

Step 2: Since node 13 is at the left of 15 and node 15 is at the right of node 10, so RL relationship exists. First, we perform the right rotation on node 15, and 15 will move downwards, and node 13 will come upwards, as shown below:



Still, node 13 is not the root node, and 13 is at the right of the root node, so we will perform left rotation known as a zig rotation. The node 10 will move downwards, and 13 becomes the root node as shown below:

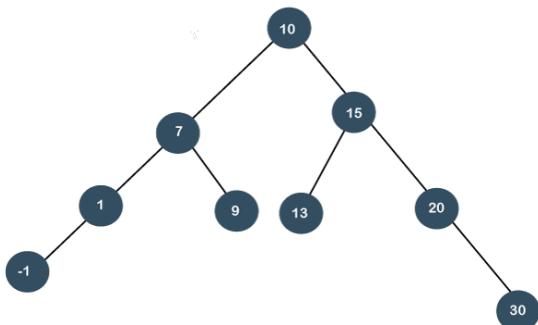


As we can observe in the above tree that node 13 has become the root node; therefore, the searching is completed. In this case, we have first performed the zig rotation and then zag rotation; so, it is known as a zig-zag rotation.

Zag zig rotation

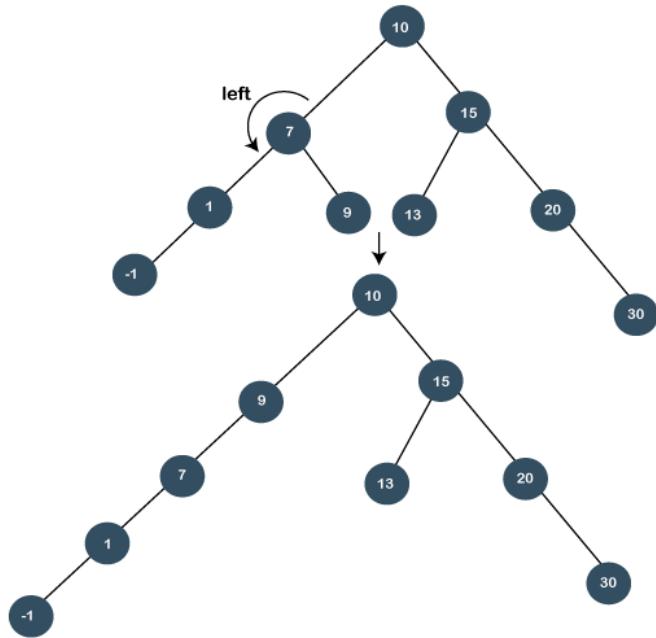
Let's understand this case through an example.

Suppose we want to search 9 element in the tree, which is shown below:

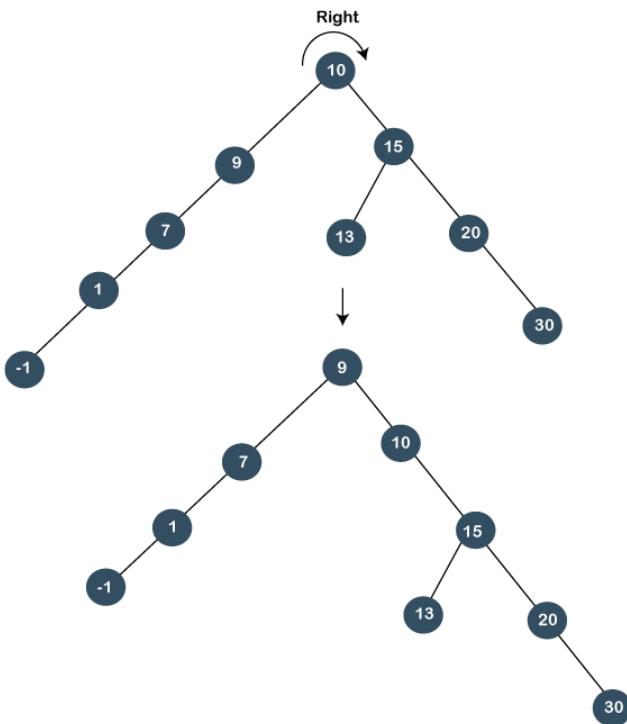


Step 1: First, we perform the standard BST searching operation. As 9 is less than 10 but greater than 7, so it will be the right child of node 7.

Step 2: Since node 9 is at the right of node 7, and node 7 is at the left of node 10, so LR relationship exists. First, we perform the left rotation on node 7. The node 7 will move downwards, and node 9 moves upwards as shown below:



Still the node 9 is not a root node, and 9 is at the left of the root node, so we will perform the right rotation known as zig rotation. After performing the right rotation, node 9 becomes the root node, as shown below:



As we can observe in the above tree that node 13 is a root node; therefore, the searching is completed. In this case, we have first performed the zag rotation (left rotation), and then zig rotation (right rotation) is performed, so it is known as a zig-zig rotation.

Advantages of Splay tree

- In the splay tree, **we do not need to store the extra information**. In contrast, in AVL trees, we need to store the balance factor of each node that requires extra space, and Red-Black

trees also require to store one extra bit of information that denotes the color of the node, either Red or Black.

- It is the fastest type of Binary Search tree for various practical applications. It is used in Windows NT and GCC compilers.
- It provides better performance as the frequently accessed nodes will move nearer to the root node, due to which the elements can be accessed quickly in splay trees. It is used in the cache implementation as the recently accessed data is stored in the cache so that we do not need to go to the memory for accessing the data, and it takes less time.

Drawback of Splay tree

The major drawback of the splay tree would be that trees are not strictly balanced, i.e., they are roughly balanced. Sometimes the splay trees are linear, so it will take O(n) time complexity.

Watch Videos:-

[Splay Tree Introduction](#)

[Splay Tree Insertion](#)

[Splay Trees deletion](#)

[Splay Tree Deletion | Top Down Splaying](#)

6. Graph Based DSA

Graph Data Structure

A graph data structure is a collection of nodes that have data and are connected to other nodes.

Let's try to understand this through an example. On facebook, everything is a node. That includes User, Photo, Album, Event, Group, Page, Comment, Story, Video, Link, Note...anything that has data is a node.

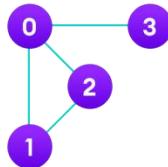
Every relationship is an edge from one node to another. Whether you post a photo, join a group, like a page, etc., a new edge is created for that relationship.



All of facebook is then a collection of these nodes and edges. This is because facebook uses a graph data structure to store its data.

More precisely, a graph is a data structure (V, E) that consists of

- A collection of vertices V
- A collection of edges E , represented as ordered pairs of vertices (u, v)



In the graph,

$$\begin{aligned} V &= \{0, 1, 2, 3\} \\ E &= \{(0,1), (0,2), (0,3), (1,2)\} \\ G &= \{V, E\} \end{aligned}$$

Graph Terminology

- **Adjacency:** A vertex is said to be adjacent to another vertex if there is an edge connecting them. Vertices 2 and 3 are not adjacent because there is no edge between them.
- **Path:** A sequence of edges that allows you to go from vertex A to vertex B is called a path. 0-1, 1-2 and 0-2 are paths from vertex 0 to vertex 2.
- **Directed Graph:** A graph in which an edge (u,v) doesn't necessarily mean that there is an edge (v,u) as well. The edges in such a graph are represented by arrows to show the direction of the edge.

Graph Representation

Graphs are commonly represented in two ways:

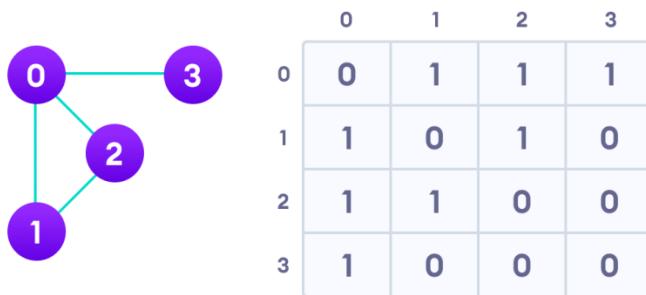
- Adjacency Matrix
- Adjacency List

Adjacency Matrix

An adjacency matrix is a 2D array of $V \times V$ vertices. Each row and column represent a vertex.

If the value of any element $a[i][j]$ is 1, it represents that there is an edge connecting vertex i and vertex j .

The adjacency matrix for the graph we created above is



Since it is an undirected graph, for edge (0,2), we also need to mark edge (2,0); making the adjacency matrix symmetric about the diagonal.

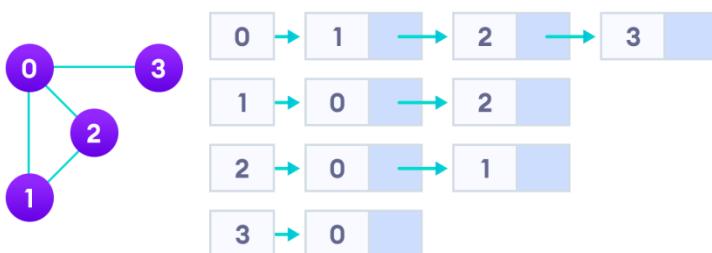
Edge lookup(checking if an edge exists between vertex A and vertex B) is extremely fast in adjacency matrix representation but we have to reserve space for every possible link between all vertices($V \times V$), so it requires more space.

Adjacency List

An adjacency list represents a graph as an array of linked lists.

The index of the array represents a vertex and each element in its linked list represents the other vertices that form an edge with the vertex.

The adjacency list for the graph we made in the first example is as follows:



An adjacency list is efficient in terms of storage because we only need to store the values for the edges. For a graph with millions of vertices, this can mean a lot of saved space.

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

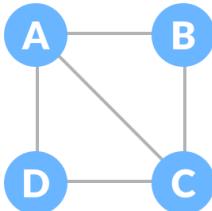
Watch Videos: -

[Adjacency Matrix and Adjacency List](#)

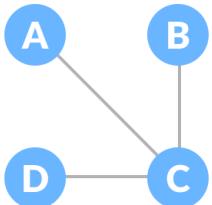
Spanning Tree

Before we learn about spanning trees, we need to understand two graphs: undirected graphs and connected graphs.

An undirected graph is a graph in which the edges do not point in any direction (ie. the edges are bidirectional).



A connected graph is a graph in which there is always a path from a vertex to any other vertex.



Spanning tree

A spanning tree is a sub-graph of an undirected connected graph, which includes all the vertices of the graph with a minimum possible number of edges. If a vertex is missed, then it is not a spanning tree.

The edges may or may not have weights assigned to them.

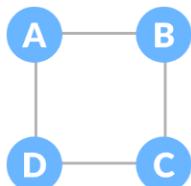
The total number of spanning trees with n vertices that can be created from a complete graph is equal to $n(n-2)$.

If we have $n = 4$, the maximum number of possible spanning trees is equal to $4^{4-2} = 16$. Thus, 16 spanning trees can be formed from a complete graph with 4 vertices.

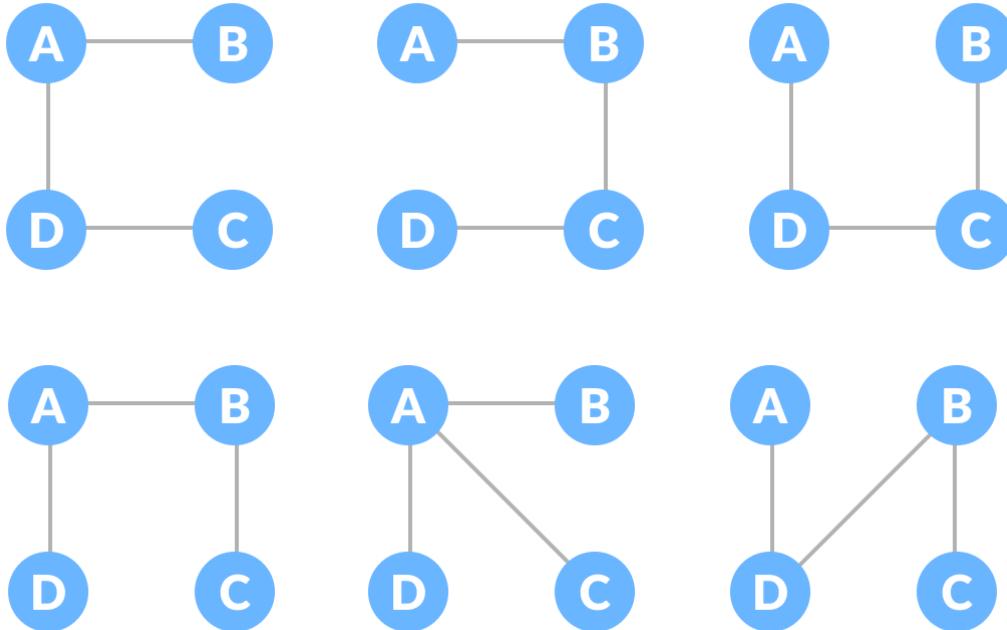
Example of a Spanning Tree

Let's understand the spanning tree with examples below:

Let the original graph be:



Some of the possible spanning trees that can be created from the above graph are:



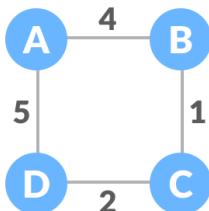
Minimum Spanning Tree

A minimum spanning tree is a spanning tree in which the sum of the weight of the edges is as minimum as possible.

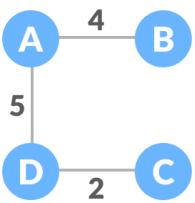
Example of a Spanning Tree

Let's understand the above definition with the help of the example below.

The initial graph is: Weighted graph

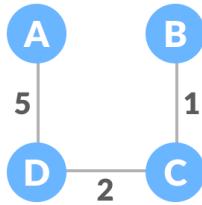


The possible spanning trees from the above graph are:



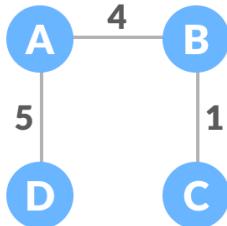
sum = 11

Minimum spanning tree – 1



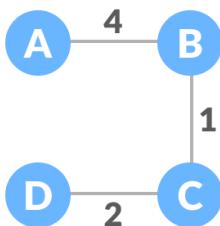
sum = 8

Minimum spanning tree – 2



sum = 10

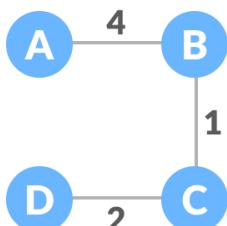
Minimum spanning tree – 3



sum = 7

Minimum spanning tree – 4

The minimum spanning tree from the above spanning trees is:



sum = 7

The minimum spanning tree from a graph is found using the following algorithms:

- o Prim's Algorithm
- o Kruskal's Algorithm

Spanning Tree Applications

- Computer Network Routing Protocol
- Cluster Analysis
- Civil Network Planning

Minimum Spanning tree Applications

- To find paths in the map
- To design networks like telecommunication networks, water supply networks, and electrical grids.

Watch Videos: -

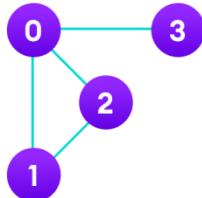
[Minimum spanning tree 1](#)

[Minimum spanning tree 2](#)

Adjacency Matrix

An adjacency matrix is a way of representing a graph as a matrix of booleans (0's and 1's). A finite graph can be represented in the form of a square matrix on a computer, where the boolean value of the matrix indicates if there is a direct path between two vertices.

For example, we have a graph below. An undirected graph



We can represent this graph in matrix form like below.

j \ i	0	1	2	3
0	0	1	1	1
1	1	0	1	0
2	1	1	0	0
3	1	0	0	0

Each cell in the above table/matrix is represented as A_{ij} , where i and j are vertices. The value of A_{ij} is either 1 or 0 depending on whether there is an edge from vertex i to vertex j .

If there is a path from i to j , then the value of A_{ij} is 1 otherwise its 0. For instance, there is a path from vertex 1 to vertex 2, so A_{12} is 1 and there is no path from vertex 1 to 3, so A_{13} is 0.

In case of undirected graphs, the matrix is symmetric about the diagonal because of every edge (i,j) , there is also an edge (j,i) .

Pros of Adjacency Matrix

- o The basic operations like adding an edge, removing an edge, and checking whether there is an edge from vertex i to vertex j are extremely time efficient, constant time operations.
- o If the graph is dense and the number of edges is large, an adjacency matrix should be the first choice. Even if the graph and the adjacency matrix is sparse, we can represent it using data structures for sparse matrices.
- o The biggest advantage, however, comes from the use of matrices. The recent advances in hardware enable us to perform even expensive matrix operations on the GPU.
- o By performing operations on the adjacent matrix, we can get important insights into the nature of the graph and the relationship between its vertices.

Cons of Adjacency Matrix

- The $V \times V$ space requirement of the adjacency matrix makes it a memory hog. Graphs out in the wild usually don't have too many connections and this is the major reason why adjacency lists are the better choice for most tasks.
- While basic operations are easy, operations like `inEdges` and `outEdges` are expensive when using the adjacency matrix representation.

Adjacency Matrix Applications

- Creating routing table in networks
- Navigation tasks

Program to implement Adjacency Matrix

```
// Adjacency Matrix representation

#include <stdio.h>
#define V 4

// Initialize the matrix to zero
void init(int arr[][V]) {
    int i, j;
    for (i = 0; i < V; i++)
        for (j = 0; j < V; j++)
            arr[i][j] = 0;
}

// Add edges
void addEdge(int arr[][V], int i, int j) {
    arr[i][j] = 1;
    arr[j][i] = 1;
}

// Print the matrix
void printAdjMatrix(int arr[][V]) {
    int i, j;

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

int main() {
    int adjMatrix[V][V];

    init(adjMatrix);
    addEdge(adjMatrix, 0, 1);
    addEdge(adjMatrix, 0, 2);
    addEdge(adjMatrix, 1, 2);
    addEdge(adjMatrix, 2, 0);
    addEdge(adjMatrix, 2, 3);

    printAdjMatrix(adjMatrix);

    return 0;
}
```

Output

```
0: 0 1 1 0
1: 1 0 1 0
2: 1 1 0 1
3: 0 0 1 0
```

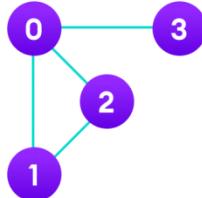
Watch Videos :-

[Adjacency Matrix and Adjacency List](#)

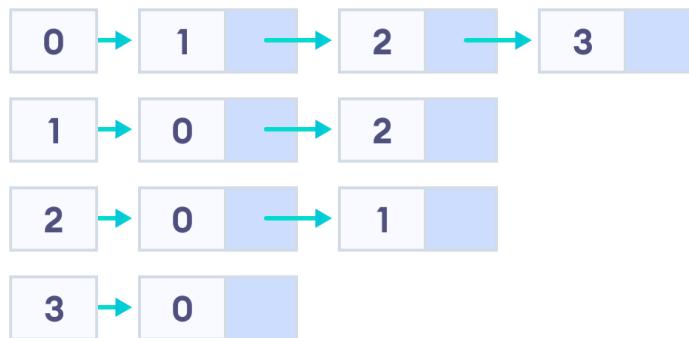
Adjacency List

An adjacency list represents a graph as an array of linked lists. The index of the array represents a vertex and each element in its linked list represents the other vertices that form an edge with the vertex.

For example, we have a graph below. An undirected graph



We can represent this graph in the form of a linked list on a computer as shown below.



Here, **0, 1, 2, 3** are the vertices and each of them forms a linked list with all of its adjacent vertices. For instance, vertex 1 has two adjacent vertices 0 and 2. Therefore, 1 is linked with 0 and 2 in the figure above.

Pros of Adjacency List

- An adjacency list is efficient in terms of storage because we only need to store the values for the edges. For a sparse graph with millions of vertices and edges, this can mean a lot of saved space.
- It also helps to find all the vertices adjacent to a vertex easily.

Cons of Adjacency List

- Finding the adjacent list is not quicker than the adjacency matrix because all the connected nodes must be first explored to find them.

Adjacency List Structure

The simplest adjacency list needs a node data structure to store a vertex and a graph data structure to organize the nodes.

We stay close to the basic definition of a graph - a collection of vertices and edges $\{V, E\}$. For simplicity, we use an unlabeled graph as opposed to a labeled one i.e. the vertices are identified by their indices 0,1,2,3.

Let's dig into the data structures at play here.

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

struct Graph{
    int numVertices;
    struct node** adjLists;
};
```

Don't let the `struct node** adjLists` overwhelm you.

All we are saying is we want to store a pointer to `struct node*`. This is because we don't know how many vertices the graph will have and so we cannot create an array of Linked Lists at compile time.

Program to implement Adjacency List

```

// Adjacency List representation
#include <stdio.h>
#include <stdlib.h>

struct node {
    int vertex;
    struct node* next;
};

struct node* createNode(int);

struct Graph {
    int numVertices;
    struct node** adjLists;
};

// Create a node
struct node* createNode(int v) {
    struct node* newNode = malloc(sizeof(struct node));
    newNode->vertex = v;
    newNode->next = NULL;
    return newNode;
}

// Create a graph
struct Graph* createAGraph(int vertices) {
    struct Graph* graph = malloc(sizeof(struct Graph));
    graph->numVertices = vertices;

    graph->adjLists = malloc(vertices * sizeof(struct node*));

    int i;
    for (i = 0; i < vertices; i++)
        graph->adjLists[i] = NULL;

    return graph;
}

// Add edge
void addEdge(struct Graph* graph, int s, int d) {
    // Add edge from s to d
    struct node* newNode = createNode(d);
    newNode->next = graph->adjLists[s];
    graph->adjLists[s] = newNode;

    // Add edge from d to s
    newNode = createNode(s);
    newNode->next = graph->adjLists[d];
    graph->adjLists[d] = newNode;
}

// Print the graph
void printGraph(struct Graph* graph) {
    int v;
    for (v = 0; v < graph->numVertices; v++) {
        struct node* temp = graph->adjLists[v];
        printf("\n Vertex %d:\n", v);
        while (temp) {
            printf("%d -> ", temp->vertex);
            temp = temp->next;
        }
    }
}

```

```

        temp = temp->next;
    }
    printf("\n");
}
}

int main() {
    struct Graph* graph = createAGraph(4);
    addEdge(graph, 0, 1);
    addEdge(graph, 0, 2);
    addEdge(graph, 0, 3);
    addEdge(graph, 1, 2);

    printGraph(graph);

    return 0;
}

```

Output

```

Vertex 0
: 3 -> 2 -> 1 ->
Vertex 1
: 2 -> 0 ->
Vertex 2
: 1 -> 0 ->
Vertex 3
: 0 ->

```

Watch Videos :-

[Adjacency Matrix and Adjacency List](#)

Depth First Search

Depth first Search or Depth first traversal is a recursive algorithm for searching all the vertices of a graph or tree data structure. Traversal means visiting all the nodes of a graph.

Depth First Search Algorithm

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

- Visited
- Not Visited

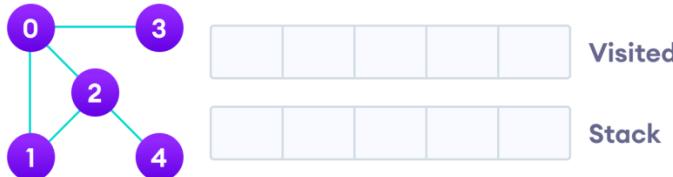
The purpose of the algorithm is to mark each vertex as visited while avoiding cycles.

The DFS algorithm works as follows:

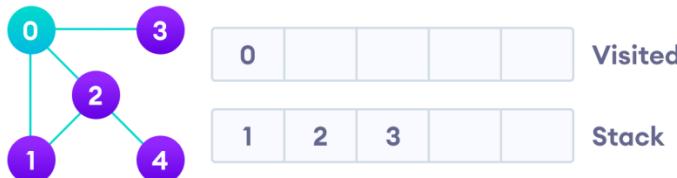
- Start by putting any one of the graph's vertices on top of a stack.
- Take the top item of the stack and add it to the visited list.
- Create a list of that vertex's adjacent nodes. Add the ones which aren't in the visited list to the top of the stack.
- Keep repeating steps 2 and 3 until the stack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

Depth First Search Example

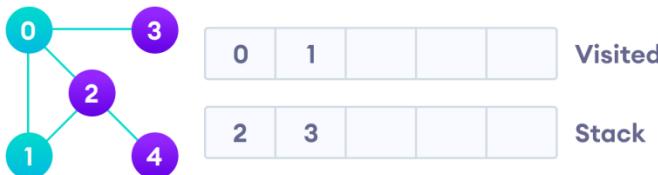
Let's see how the Depth First Search algorithm works with an example. We use an undirected graph with 5 vertices.



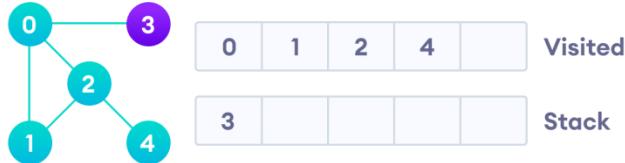
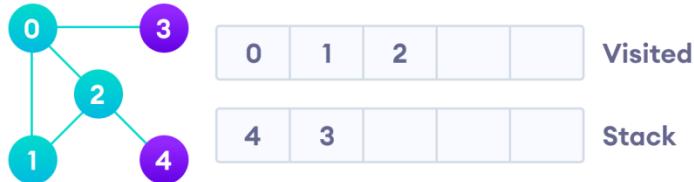
We start from vertex 0, the DFS algorithm starts by putting it in the Visited list and putting all its adjacent vertices in the stack.



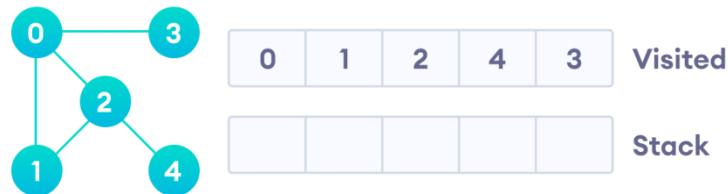
Next, we visit the element at the top of stack i.e. 1 and go to its adjacent nodes. Since 0 has already been visited, we visit 2 instead.



Vertex 2 has an unvisited adjacent vertex in 4, so we add that to the top of the stack and visit it.



After we visit the last element 3, it doesn't have any unvisited adjacent nodes, so we have completed the Depth First Traversal of the graph.



Program to implement Depth First Search

```
// DFS algorithm in C

#include <stdio.h>
#include <stdlib.h>

struct node {
    int vertex;
    struct node* next;
};

struct node* createNode(int v);

struct Graph {
    int numVertices;
    int* visited;

    // We need int** to store a two dimensional array.
    // Similary, we need struct node** to store an array of Linked lists
    struct node** adjLists;
};

// DFS algo
void DFS(struct Graph* graph, int vertex) {
    struct node* adjList = graph->adjLists[vertex];
    struct node* temp = adjList;

    graph->visited[vertex] = 1;
    printf("Visited %d \n", vertex);

    while (temp != NULL) {
        int connectedVertex = temp->vertex;

        if (graph->visited[connectedVertex] == 0) {
            DFS(graph, connectedVertex);
        }
        temp = temp->next;
    }
}

// Create a node
struct node* createNode(int v) {
    struct node* newNode = malloc(sizeof(struct node));
    newNode->vertex = v;
    newNode->next = NULL;
    return newNode;
}

// Create graph
struct Graph* createGraph(int vertices) {
    struct Graph* graph = malloc(sizeof(struct Graph));
    graph->numVertices = vertices;

    graph->adjLists = malloc(vertices * sizeof(struct node*));
    graph->visited = malloc(vertices * sizeof(int));

    int i;
    for (i = 0; i < vertices; i++) {

```

```

graph->adjLists[i] = NULL;
graph->visited[i] = 0;
}
return graph;
}

// Add edge
void addEdge(struct Graph* graph, int src, int dest) {
    // Add edge from src to dest
    struct node* newNode = createNode(dest);
    newNode->next = graph->adjLists[src];
    graph->adjLists[src] = newNode;

    // Add edge from dest to src
    newNode = createNode(src);
    newNode->next = graph->adjLists[dest];
    graph->adjLists[dest] = newNode;
}

// Print the graph
void printGraph(struct Graph* graph) {
    int v;
    for (v = 0; v < graph->numVertices; v++) {
        struct node* temp = graph->adjLists[v];
        printf("\n Adjacency list of vertex %d\n", v);
        while (temp) {
            printf("%d -> ", temp->vertex);
            temp = temp->next;
        }
        printf("\n");
    }
}

int main() {
    struct Graph* graph = createGraph(4);
    addEdge(graph, 0, 1);
    addEdge(graph, 0, 2);
    addEdge(graph, 1, 2);
    addEdge(graph, 2, 3);
    printGraph(graph);
    DFS(graph, 2);
    return 0;
}

```

Output

```

Adjacency list of vertex 0
2 -> 1 ->
Adjacency list of vertex 1
2 -> 0 ->
Adjacency list of vertex 2
3 -> 1 -> 0 ->
Adjacency list of vertex 3
2 ->
Visited 2
Visited 3
Visited 1
Visited 0

```

Watch Video: -

[BFS & DFS -Breadth First Search and Depth First Search](#)

[BFS and DFS Graph Traversals| Breadth First Search and Depth First Search](#)

Breadth-First Search

Traversal means visiting all the nodes of a graph. Breadth First Traversal or Breadth First Search is a recursive algorithm for searching all the vertices of a graph or tree data structure.

BFS algorithm

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

- Visited
- Not Visited

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

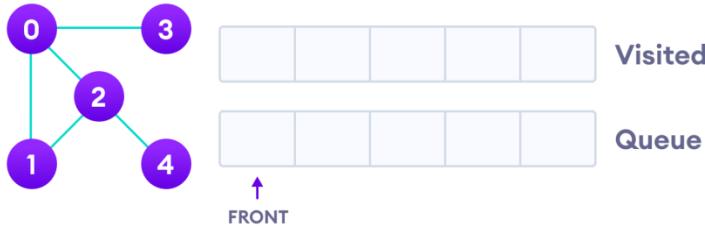
The algorithm works as follows:

- Start by putting any one of the graph's vertices at the back of a queue.
- Take the front item of the queue and add it to the visited list.
- 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.
- Keep repeating steps 2 and 3 until the queue is empty.

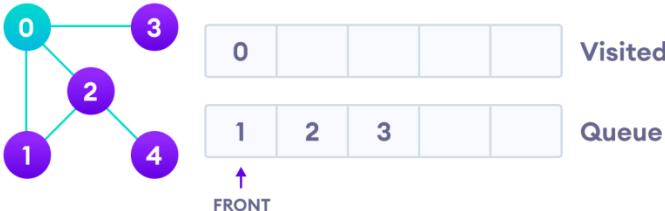
The graph might have two different disconnected parts so to make sure that we cover every vertex, we can also run the BFS algorithm on every node

BFS example

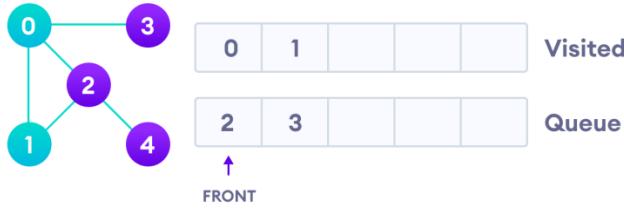
Let's see how the Breadth First Search algorithm works with an example. We use an undirected graph with 5 vertices.



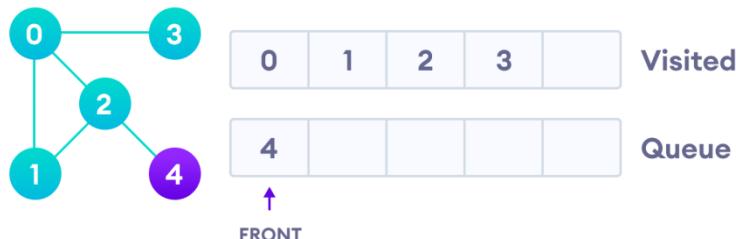
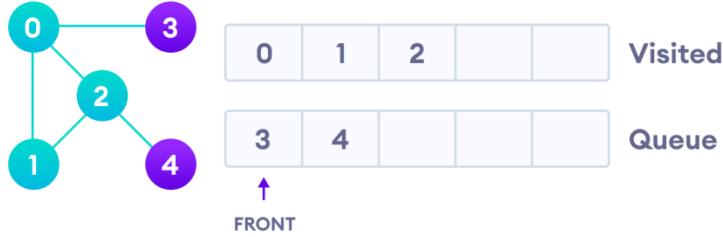
We start from vertex 0, the BFS algorithm starts by putting it in the Visited list and putting all its adjacent vertices in the stack.



Next, we visit the element at the front of queue i.e. 1 and go to its adjacent nodes. Since 0 has already been visited, we visit 2 instead.



Vertex 2 has an unvisited adjacent vertex in 4, so we add that to the back of the queue and visit 3, which is at the front of the queue.



Only 4 remains in the queue since the only adjacent node of 3 i.e. 0 is already visited. We visit it.



Since the queue is empty, we have completed the Breadth First Traversal of the graph.

Program to implement Breadth-First Search

```
// BFS algorithm in C

#include <stdio.h>
#include <stdlib.h>
#define SIZE 40

struct queue {
    int items[SIZE];
    int front;
    int rear;
};

struct queue* createQueue();
void enqueue(struct queue* q, int);
int dequeue(struct queue* q);
void display(struct queue* q);
int isEmpty(struct queue* q);
void printQueue(struct queue* q);

struct node {
    int vertex;
    struct node* next;
};

struct node* createNode(int);

struct Graph {
    int numVertices;
    struct node** adjLists;
    int* visited;
};

// BFS algorithm
void bfs(struct Graph* graph, int startVertex) {
    struct queue* q = createQueue();

    graph->visited[startVertex] = 1;
    enqueue(q, startVertex);

    while (!isEmpty(q)) {
        printQueue(q);
        int currentVertex = dequeue(q);
        printf("Visited %d\n", currentVertex);

        struct node* temp = graph->adjLists[currentVertex];

        while (temp) {
            int adjVertex = temp->vertex;

            if (graph->visited[adjVertex] == 0) {
                graph->visited[adjVertex] = 1;
                enqueue(q, adjVertex);
            }
            temp = temp->next;
        }
    }
}
```

```

// Creating a node
struct node* createNode(int v) {
    struct node* newNode = malloc(sizeof(struct node));
    newNode->vertex = v;
    newNode->next = NULL;
    return newNode;
}

// Creating a graph
struct Graph* createGraph(int vertices) {
    struct Graph* graph = malloc(sizeof(struct Graph));
    graph->numVertices = vertices;

    graph->adjLists = malloc(vertices * sizeof(struct node*));
    graph->visited = malloc(vertices * sizeof(int));

    int i;
    for (i = 0; i < vertices; i++) {
        graph->adjLists[i] = NULL;
        graph->visited[i] = 0;
    }

    return graph;
}

// Add edge
void addEdge(struct Graph* graph, int src, int dest) {
    // Add edge from src to dest
    struct node* newNode = createNode(dest);
    newNode->next = graph->adjLists[src];
    graph->adjLists[src] = newNode;

    // Add edge from dest to src
    newNode = createNode(src);
    newNode->next = graph->adjLists[dest];
    graph->adjLists[dest] = newNode;
}

// Create a queue
struct queue* createQueue() {
    struct queue* q = malloc(sizeof(struct queue));
    q->front = -1;
    q->rear = -1;
    return q;
}

// Check if the queue is empty
int isEmpty(struct queue* q) {
    if (q->rear == -1)
        return 1;
    else
        return 0;
}

// Adding elements into queue
void enqueue(struct queue* q, int value) {
    if (q->rear == SIZE - 1)
        printf("\nQueue is Full!!!");
    else {
        if (q->front == -1)

```

```

        q->front = 0;
        q->rear++;
        q->items[q->rear] = value;
    }
}

// Removing elements from queue
int dequeue(struct queue* q) {
    int item;
    if (isEmpty(q)) {
        printf("Queue is empty");
        item = -1;
    } else {
        item = q->items[q->front];
        q->front++;
        if (q->front > q->rear) {
            printf("Resetting queue ");
            q->front = q->rear = -1;
        }
    }
    return item;
}

// Print the queue
void printQueue(struct queue* q) {
    int i = q->front;

    if (isEmpty(q)) {
        printf("Queue is empty");
    } else {
        printf("\nQueue contains \n");
        for (i = q->front; i < q->rear + 1; i++) {
            printf("%d ", q->items[i]);
        }
    }
}

int main() {
    struct Graph* graph = createGraph(6);
    addEdge(graph, 0, 1);
    addEdge(graph, 0, 2);
    addEdge(graph, 1, 2);
    addEdge(graph, 1, 4);
    addEdge(graph, 1, 3);
    addEdge(graph, 2, 4);
    addEdge(graph, 3, 4);

    bfs(graph, 0);

    return 0;
}

```

Output

Queue contains
0 Resetting queue Visited 0
Queue contains
2 1 Visited 2
Queue contains
1 4 Visited 1

```
Queue contains  
4 3 Visited 4  
Queue contains  
3 Resetting queue Visited 3
```

Watch Video: -

[BFS & DFS -Breadth First Search and Depth First Search](#)

[BFS and DFS Graph Traversals| Breadth First Search and Depth First Search](#)

[Articulation Point and Biconnected Components](#)

Bellman Ford's Algorithm

Bellman Ford algorithm helps us find the shortest path from a vertex to all other vertices of a weighted graph.

It is similar to Dijkstra's algorithm but it can work with graphs in which edges can have negative weights.

Why would one ever have edges with negative weights in real life?

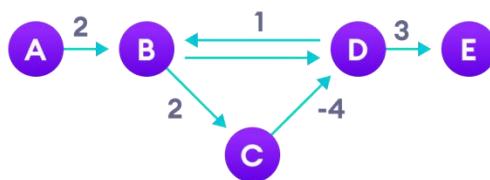
Negative weight edges might seem useless at first but they can explain a lot of phenomena like cashflow, the heat released/absorbed in a chemical reaction, etc.

For instance, if there are different ways to reach from one chemical A to another chemical B, each method will have sub-reactions involving both heat dissipation and absorption.

If we want to find the set of reactions where minimum energy is required, then we will need to be able to factor in the heat absorption as negative weights and heat dissipation as positive weights.

Why do we need to be careful with negative weights?

Negative weight edges can create negative weight cycles i.e. a cycle that will reduce the total path distance by coming back to the same point.



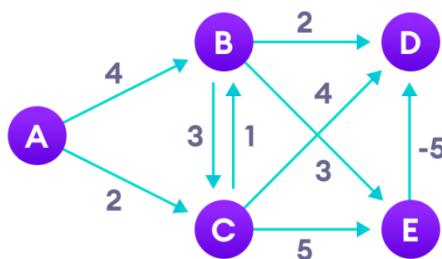
Shortest path algorithms like Dijkstra's Algorithm that aren't able to detect such a cycle can give an incorrect result because they can go through a negative weight cycle and reduce the path length.

How Bellman Ford's algorithm works

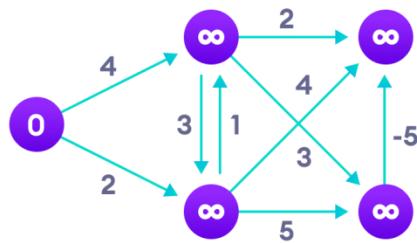
Bellman Ford algorithm works by overestimating the length of the path from the starting vertex to all other vertices. Then it iteratively relaxes those estimates by finding new paths that are shorter than the previously overestimated paths.

By doing this repeatedly for all vertices, we can guarantee that the result is optimized.

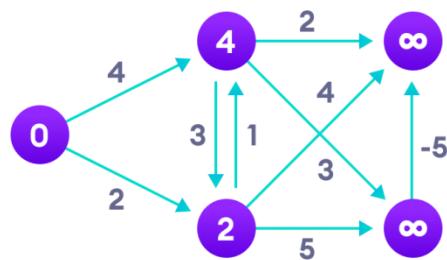
Step 1: Start with the weighted graph



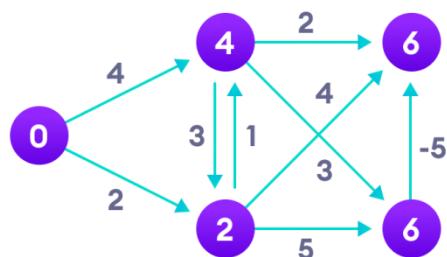
Step 2: Choose a starting vertex and assign infinity path values to all other vertices



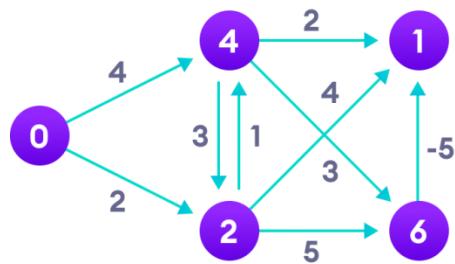
Step 3: Visit each edge and relax the path distances if they are inaccurate



Step 4: We need to do this V times because in the worst case, a vertex's path length might need to be readjusted V times



Step 5: Notice how the vertex at the top right corner had its path length adjusted



Step 6: After all the vertices have their path lengths, we check if a negative cycle is present

	B	C	D	E
0	∞	∞	∞	∞
0	4	2	∞	∞
0	3	2	6	6
0	3	2	1	6
0	3	2	1	6

Program to implement Bellman Ford Algorithm

```

// Bellman Ford Algorithm in C
#include <stdio.h>
#include <stdlib.h>

#define INFINITY 99999

//struct for the edges of the graph
struct Edge {
    int u; //start vertex of the edge
    int v; //end vertex of the edge
    int w; //weight of the edge (u,v)
};

//Graph - it consists of edges
struct Graph {
    int V; //total number of vertices in the graph
    int E; //total number of edges in the graph
    struct Edge *edge; //array of edges
};

void bellmanford(struct Graph *g, int source);
void display(int arr[], int size);

int main(void) {
    //create graph
    struct Graph *g = (struct Graph *)malloc(sizeof(struct Graph));
    g->V = 4; //total vertices
    g->E = 5; //total edges

    //array of edges for graph
    g->edge = (struct Edge *)malloc(g->E * sizeof(struct Edge));

    //----- adding the edges of the graph
    /*
        edge(u, v)
        where u = start vertex of the edge (u,v)
              v = end vertex of the edge (u,v)

        w is the weight of the edge (u,v)
    */

    //edge 0 --> 1
    g->edge[0].u = 0;
    g->edge[0].v = 1;
    g->edge[0].w = 5;

    //edge 0 --> 2
    g->edge[1].u = 0;
    g->edge[1].v = 2;
    g->edge[1].w = 4;

    //edge 1 --> 3
    g->edge[2].u = 1;
    g->edge[2].v = 3;
    g->edge[2].w = 3;

    //edge 2 --> 1
    g->edge[3].u = 2;

```

```

g->edge[3].v = 1;
g->edge[3].w = 6;

//edge 3 --> 2
g->edge[4].u = 3;
g->edge[4].v = 2;
g->edge[4].w = 2;

bellmanford(g, 0); //0 is the source vertex

return 0;
}

void bellmanford(struct Graph *g, int source) {
    //variables
    int i, j, u, v, w;

    //total vertex in the graph g
    int tV = g->V;

    //total edge in the graph g
    int tE = g->E;

    //distance array
    //size equal to the number of vertices of the graph g
    int d[tV];

    //predecessor array
    //size equal to the number of vertices of the graph g
    int p[tV];

    //step 1: fill the distance array and predecessor array
    for (i = 0; i < tV; i++) {
        d[i] = INFINITY;
        p[i] = 0;
    }

    //mark the source vertex
    d[source] = 0;

    //step 2: relax edges |V| - 1 times
    for (i = 1; i <= tV - 1; i++) {
        for (j = 0; j < tE; j++) {
            //get the edge data
            u = g->edge[j].u;
            v = g->edge[j].v;
            w = g->edge[j].w;

            if (d[u] != INFINITY && d[v] > d[u] + w) {
                d[v] = d[u] + w;
                p[v] = u;
            }
        }
    }

    //step 3: detect negative cycle
    //if value changes then we have a negative cycle in the graph
    //and we cannot find the shortest distances
    for (i = 0; i < tE; i++) {
        u = g->edge[i].u;

```

```

v = g->edge[i].v;
w = g->edge[i].w;
if (d[u] != INFINITY && d[v] > d[u] + w) {
    printf("Negative weight cycle detected!\n");
    return;
}
}

//No negative weight cycle found!
//print the distance and predecessor array
printf("Distance array: ");
display(d, tV);
printf("Predecessor array: ");
display(p, tV);
}

void display(int arr[], int size) {
    int i;
    for (i = 0; i < size; i++) {
        printf("%d ", arr[i]);
    }
    printf("\n");
}

```

Output

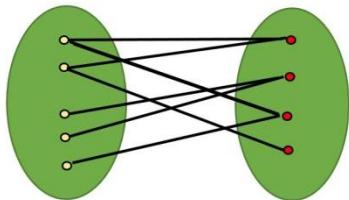
Distance array: 0 5 4 8
 Predecessor array: 0 0 0 1

Watch Video:-

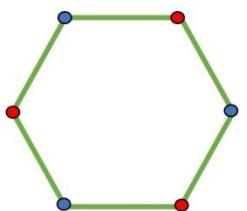
[Bellman Ford Algorithm - Single Source Shortest Path - Dynamic Programming](#)
[Bellman Ford Algorithm-Single Source Shortest Path | Dynamic Programming -2](#)

Bipartite graph

A Bipartite Graph is a graph whose vertices can be divided into two independent sets, U and V such that every edge (u, v) either connects a vertex from U to V or a vertex from V to U. In other words, for every edge (u, v) , either u belongs to U and v to V, or u belongs to V and v to U. We can also say that there is no edge that connects vertices of same set.

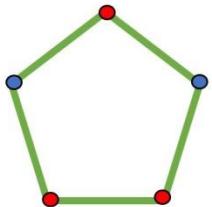


A bipartite graph is possible if the graph coloring is possible using two colors such that vertices in a set are colored with the same color. Note that it is possible to color a cycle graph with even cycle using two colors. For example, see the following graph.



Cycle graph of length 6

It is not possible to color a cycle graph with odd cycle using two colors.



Cycle graph of length 5

Algorithm to check if a graph is Bipartite:

One approach is to check whether the graph is 2-colorable or not using backtracking algorithm m coloring problem.

Following is a simple algorithm to find out whether a given graph is Bipartite or not using Breadth First Search (BFS).

1. Assign RED color to the source vertex (putting into set U).
2. Color all the neighbors with BLUE color (putting into set V).
3. Color all neighbor's neighbor with RED color (putting into set U).
4. This way, assign color to all vertices such that it satisfies all the constraints of m way coloring problem where $m = 2$.
5. While assigning colors, if we find a neighbor which is colored with same color as current vertex, then the graph cannot be colored with 2 vertices (or graph is not Bipartite)

A graph $G=(V, E)$ is called a bipartite graph if its vertices V can be partitioned into two subsets V_1 and V_2 such that each edge of G connects a vertex of V_1 to a vertex V_2 . It is denoted by K_{mn} , where m and n are the numbers of vertices in V_1 and V_2 respectively.

Example: Draw the bipartite graphs $K_{2,4}$ and $K_{3,4}$. Assuming any number of edges.

Solution: First draw the appropriate number of vertices on two parallel columns or rows and connect the vertices in one column or row with the vertices in other column or row. The bipartite graphs $K_{2,4}$ and $K_{3,4}$ are shown in fig respectively.

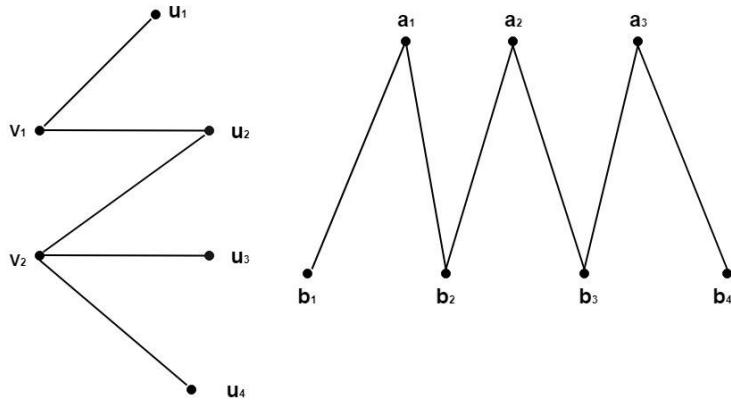


Fig:Bipartite Graph $K_{2,4}$

Fig:Bipartite Graph $K_{3,4}$

Complete Bipartite Graph

A graph $G = (V, E)$ is called a complete bipartite graph if its vertices V can be partitioned into two subsets V_1 and V_2 such that each vertex of V_1 is connected to each vertex of V_2 . The number of edges in a complete bipartite graph is $m \cdot n$ as each of the m vertices is connected to each of the n vertices.

Example: Draw the complete bipartite graphs $K_{3,4}$ and $K_{1,5}$.

Solution: First draw the appropriate number of vertices in two parallel columns or rows and connect the vertices in the first column or row with all the vertices in the second column or row. The graphs $K_{3,4}$ and $K_{1,5}$ are shown in fig:

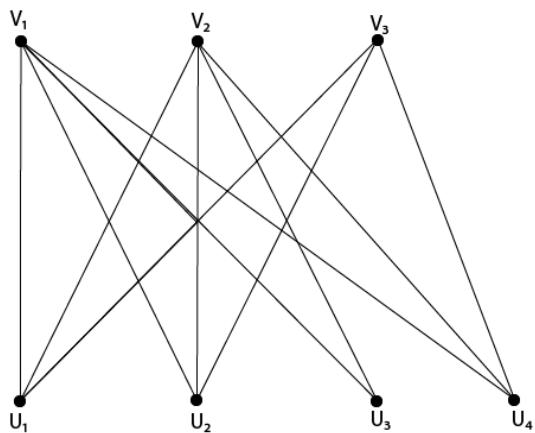


Fig: $K_{3,4}$

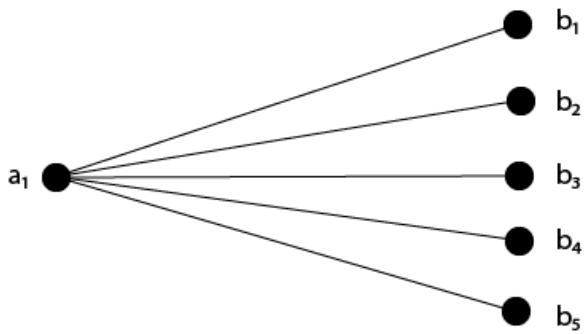


Fig: $K_{1,5}$

Watch Videos: -

[Graph Types - Bipartite Graph](#)

7. Sorting And Searching Algorithms

Bubble Sort

Bubble sort is a sorting algorithm **that compares two adjacent elements and swaps them until they are not in the intended order.**

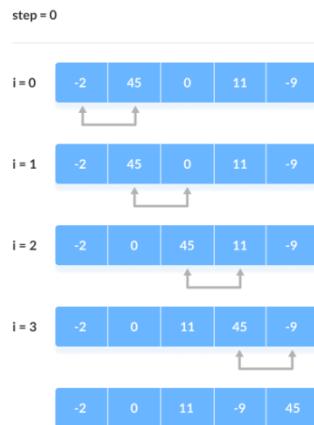
Just like the movement of air bubbles in the water that rise up to the surface, each element of the array move to the end in each iteration. Therefore, it is called a bubble sort.

Working of Bubble Sort

Suppose we are trying to sort the elements in ascending order.

1. First Iteration (Compare and Swap)

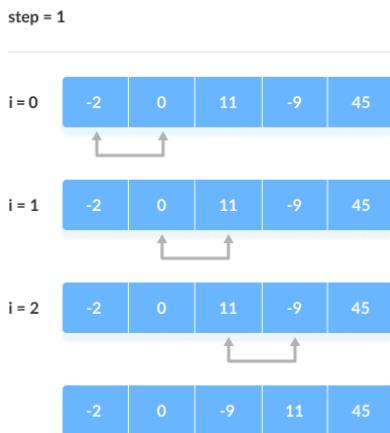
- Starting from the first index, **compare the first and the second elements.**
- **If the first element is greater than the second element, they are swapped.**
- Now, **compare the second and the third elements. Swap them if they are not in order.**
- The above process goes on until the last element.



2. Remaining Iteration

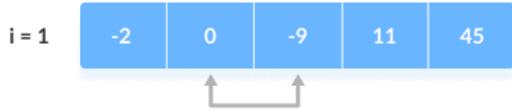
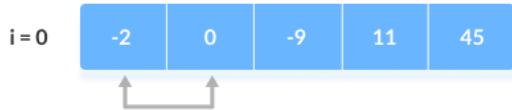
The same process goes on for the remaining iterations.

After each iteration, the largest element among the unsorted elements is placed at the end.



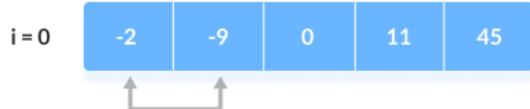
In each iteration, the comparison takes place up to the last unsorted element.

step = 2



The array is sorted when all the unsorted elements are placed at their correct positions.

step = 3



Bubble Sort Algorithm

```
bubbleSort(array)
  for i <- 1 to indexOfLastUnsortedElement-1
    if leftElement > rightElement
      swap leftElement and rightElement
end bubbleSort
```

Program to implement Bubble Sort

```
// Bubble Sort
#include <stdio.h>

void bubbleSort(int Arr[], int size) {
    int i,j,temp;

    for(i=0; i< size -1; i++) {
        for(j = 0;j < size - 1; j++) {
            if (Arr[j] > Arr[j+1]) {
                temp = Arr[j];
                Arr[j] = Arr[j+1];
                Arr[j+1] = temp;
            }
        }
    }

    printf("\nSorted Array is : ");
    for( i = 0; i < size; ++i) {
        printf("%d ", Arr[i]);
    }
}

int main()
{
    int size = 6,i = 0,j = 0;
    int Arr[size];

    printf("Enter 6 Array Values(Integer) :\n");
    for( i=0; i< size; i++){
        scanf("%d",&Arr[i]);
    }

    printf("\nThe Array is :");
    for( i = 0; i < size; ++i) {
        printf("%d ", Arr[i]);
    }

    bubbleSort(Arr,size);
    return 0;
}
```

Output

```
Enter 6 Array Values(Integer) : 1 9 3 7 2 4
The Array is : 1 9 3 7 2 4
Sorted Array is : 1 2 3 4 7 9
```

Optimized Bubble Sort Algorithm

In the above algorithm, all the comparisons are made even if the array is already sorted. This increases the execution time.

To solve this, we can introduce an extra variable swapped. The value of swapped is set true if there occurs swapping of elements. Otherwise, it is set false.

After an iteration, if there is no swapping, the value of swapped will be false. This means elements are already sorted and there is no need to perform further iterations.

This will reduce the execution time and helps to optimize the bubble sort.

Algorithm for optimized bubble sort is

```
bubbleSort(array)
    swapped <- false
    for i <- 1 to indexOfLastUnsortedElement-1
        if leftElement > rightElement
            swap leftElement and rightElement
            swapped <- true
    end bubbleSort
```

Program to implement Bubble Sort (Optimized)

```
// Bubble Sort -- Optimized
#include <stdio.h>

void bubbleSort(int Arr[], int size) {
    int i,j,temp,swaped;

    for(i=0; i< size -1; i++) {

        swaped = 0;
        for(j = 0;j < size - 1; j++) {
            if (Arr[j] > Arr[j+1]){
                temp          = Arr[j];
                Arr[j]        = Arr[j+1];
                Arr[j+1]      = temp;

                swaped        = 1;
            }
        }
        if(swaped ==0) {
            break;
        }
    }

    printf("\nSorted Array is : ");
    for( i = 0; i < size; ++i) {
        printf("%d ", Arr[i]);
    }
}

int main()
{
    int size = 6,i = 0,j = 0;
    int Arr[size];
    printf("Enter 6 Array Values(Integer) :\n");
    for( i=0; i< size; i++){
        scanf("%d",&Arr[i]);
    }

    printf("\nThe Array is :");
    for( i = 0; i < size; ++i) {
        printf("%d ", Arr[i]);
    }
    bubbleSort(Arr,size);
    return 0;
}
```

Output

```
Enter 6 Array Values(Integer) : 1 9 8 7 6 5
The Array is : 1 9 8 7 6 5
Sorted Array is : 1 5 6 7 8 9
```

Watch Videos: -

[Bubble Sort Algorithm](#)

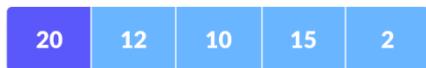
[Bubble Sort](#)

Selection Sort

Selection sort is a sorting algorithm that selects the smallest element from an unsorted list in each iteration and places that element at the beginning of the unsorted list.

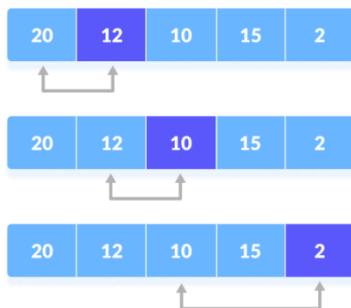
Working of Selection Sort

Set the first element as **minimum**.



Compare **minimum** with the second element. If the second element is smaller than minimum, assign the second element as **minimum**.

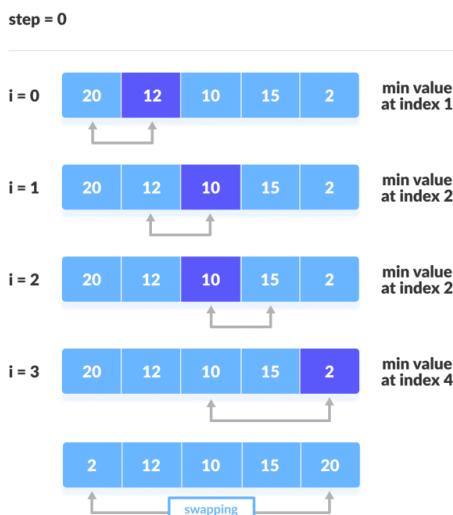
Compare **minimum** with the third element. Again, if the third element is smaller, then assign minimum to the third element otherwise do nothing. The process goes on until the last element.



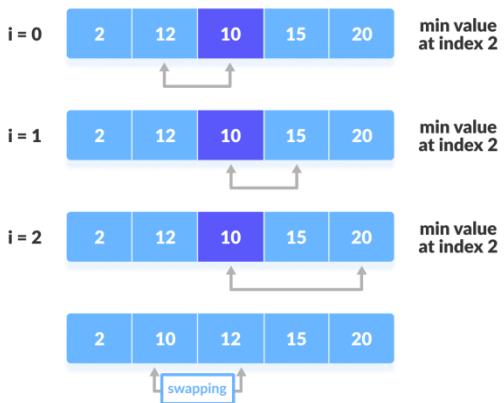
After each iteration, **minimum** is placed in the front of the unsorted list.



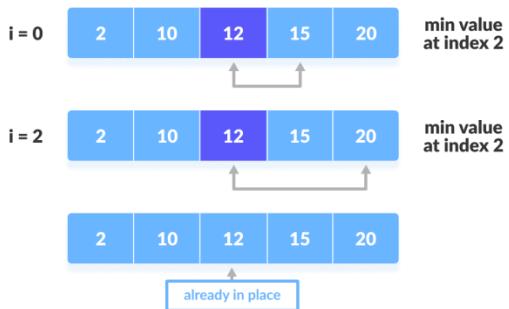
For each iteration, indexing starts from the first unsorted element. Step 1 to 3 are repeated until all the elements are placed at their correct positions.



step = 1



step = 2



The second iteration

step = 3



The third iteration

Selection Sort Algorithm

```
selectionSort(array, size)
repeat (size - 1) times
set the first unsorted element as the minimum
for each of the unsorted elements
    if element < currentMinimum
        set element as new minimum
    swap minimum with first unsorted position
end selectionSort
```

Program to implement Selection Sort

```
// Selection Sort
#include <stdio.h>

void swap(int *a, int *b) {
    int temp = *a;
    *a = *b;
    *b = temp;
}

void selectionSort(int Arr[], int size) {
    int i,j;

    for(i = 0; i < size - 1; i++) {
        int min = i;
        for(j = i + 1;j < size;j++) {
            if(Arr[j] < Arr[min]) {
                min = j;
            }
        }
        if(min != i) {
            swap(&Arr[i],&Arr[min]);
        }
    }
}

int main()
{
    int size = 6,i = 0,j = 0;
    int Arr[size];

    printf("Enter 6 Array Values(Integer) : \n");
    for( i=0; i< size; i++){
        scanf("%d",&Arr[i]);
    }

    printf("\nThe Array is :");
    for( i = 0; i < size; ++i) {
        printf("%d ", Arr[i]);
    }

    selectionSort(Arr,size);
    printf("\nSorted Array is : ");
    for( i = 0; i < size; ++i) {
        printf("%d ", Arr[i]);
    }
    return 0;
}
```

Output

```
Enter 6 Array Values(Integer) : 7 4 10 8 3 1
The Array is :7 4 10 8 3 1
Sorted Array is : 1 3 4 7 8 10
```

Watch Videos: -

[Selection Sort Algorithm](#)

Insertion Sort

Insertion sort is a sorting algorithm that places an unsorted element at its suitable place in each iteration.

Insertion sort works similarly as we sort cards in our hand in a card game.

We assume that the first card is already sorted then, we select an unsorted card. If the unsorted card is greater than the card in hand, it is placed on the right otherwise, to the left. In the same way, other unsorted cards are taken and put in their right place.

A similar approach is used by insertion sort.

Working of Insertion Sort

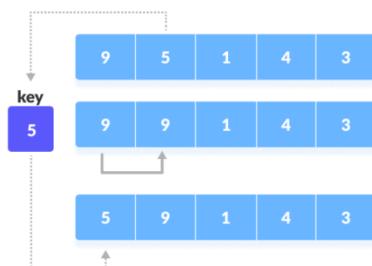
Suppose we need to sort the following array.

9	5	1	4	3
---	---	---	---	---

The first element in the array is assumed to be sorted. Take the second element and store it separately in key.

Compare key with the first element. If the first element is greater than key, then key is placed in front of the first element.

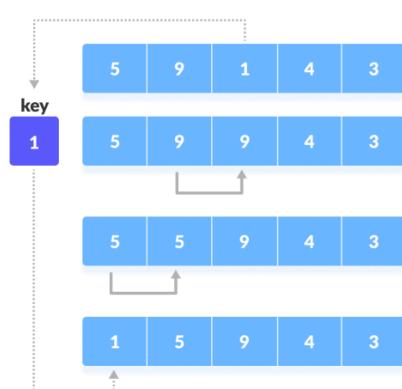
step = 1



Now, the first two elements are sorted.

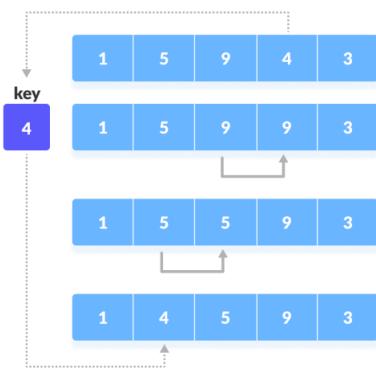
Take the third element and compare it with the elements on the left of it. Placed it just behind the element smaller than it. If there is no element smaller than it, then place it at the beginning of the array.

step = 2

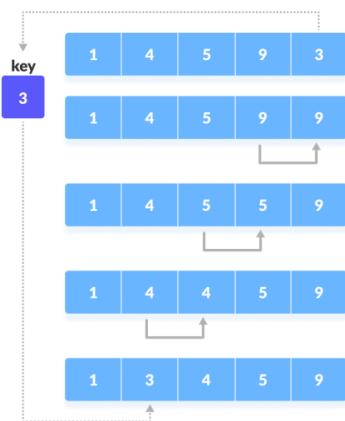


Similarly, place every unsorted element at its correct position.

step = 3



step = 4



Insertion Sort Algorithm

```
insertionSort(array)
    mark first element as sorted
    for each unsorted element X
        'extract' the element X
        for j <- lastSortedIndex down to 0
            if current element j > X
                move sorted element to the right by 1
            break loop and insert X here
end insertionSort
```

Program to implement Insertion Sort

```
// Insertion Sort
#include <stdio.h>

void insertionSort(int Arr[], int size) {
    int i,j,temp;
    for(i = 1; i < size; i++) {
        temp = Arr[i];
        j = i - 1;
        while(j >= 0 && Arr[j] > temp) {
            Arr[j+1] = Arr[j];
            j = j - 1;
        }
        Arr[j+1] = temp;
    }
}

void display(int Arr[], int size)
{
    int i;
    printf("\nSorted Array is : ");
    for( i = 0; i < size; ++i) {
        printf("%d ", Arr[i]);
    }
}

int main()
{
    int size = 6,i = 0;
    int Arr[size];
    printf("Enter 6 Array Values(Integer) :\n");
    for( i=0; i< size; i++){
        scanf("%d",&Arr[i]);
    }

    printf("\nThe Array is : ");
    display(Arr,size);
    insertionSort(Arr,size);
    display(Arr,size);

    return 0;
}
```

Output

```
Enter 6 Array Values(Integer) : 7 4 10 8 3 1
The Array is : 7 4 10 8 3 1
Sorted Array is : 1 3 4 7 8 10
```

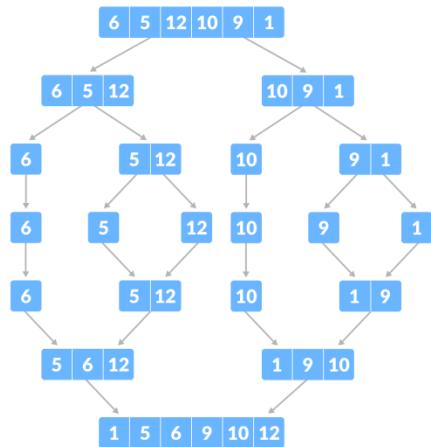
Watch Videos: -

[Insertion Sort Algorithm](#)
[Insertion Sort](#)

Merge Sort

Merge Sort is one of the most popular sorting algorithms that is based on the principle of Divide and Conquer Algorithm.

Here, a problem is divided into multiple sub-problems. Each sub-problem is solved individually. Finally, sub-problems are combined to form the final solution.



Divide and Conquer Strategy

Using the Divide and Conquer technique, we divide a problem into subproblems. When the solution to each subproblem is ready, we 'combine' the results from the subproblems to solve the main problem.

Suppose we had to sort an array A. A subproblem would be to sort a sub-section of this array starting at index p and ending at index r, denoted as A[p..r].

Divide

If q is the half-way point between p and r, then we can split the subarray A[p..r] into two arrays A[p..q] and A[q+1, r].

Conquer

In the conquer step, we try to sort both the subarrays A[p..q] and A[q+1, r]. If we haven't yet reached the base case, we again divide both these subarrays and try to sort them.

Combine

When the conquer step reaches the base step and we get two sorted subarrays A[p..q] and A[q+1, r] for array A[p..r], we combine the results by creating a sorted array A[p..r] from two sorted subarrays A[p..q] and A[q+1, r].

MergeSort Algorithm

The MergeSort function repeatedly divides the array into two halves until we reach a stage where we try to perform MergeSort on a subarray of size 1 i.e. p == r.

After that, the merge function comes into play and combines the sorted arrays into larger arrays until the whole array is merged.

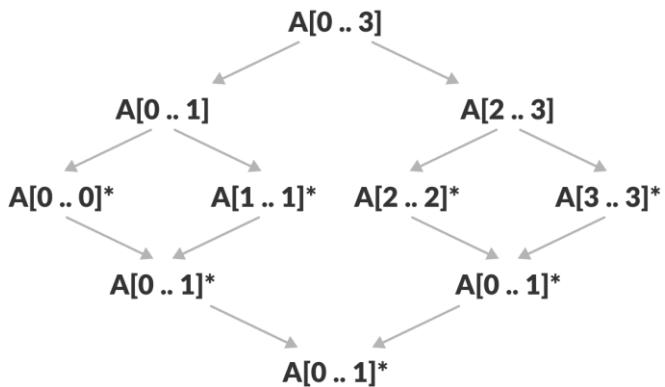
```

MergeSort(A, p, r):
    if p > r
        return
    q = (p+r)/2
    mergeSort(A, p, q)
    mergeSort(A, q+1, r)
    merge(A, p, q, r)

```

To sort an entire array, we need to call `MergeSort(A, 0, length(A)-1)`.

As shown in the image below, the merge sort algorithm recursively divides the array into halves until we reach the base case of array with 1 element. After that, the merge function picks up the sorted sub-arrays and merges them to gradually sort the entire array.



Program to implement Merge Sort

```
// Merge Sort
#include <stdio.h>

#define max 6
int arr[max];

void display(int Arr[], int size)
{
    int i;
    for( i = 0; i < size; ++i) {
        printf("%d ", Arr[i]);
    }
}

void merge(int lb,int mid,int ub) {
    int temp[max];
    int i = lb;
    int j = mid+1;
    int k = lb;

    while(i <= mid && j<=ub) {
        if(arr[i] <= arr[j]){
            temp[k++] = arr[i++];
        }
        else{
            temp[k++] = arr[j++];
        }
    }

    // copying remaining Elements
    while(i <= mid){
        temp[k++] = arr[i++];
    }
    while(j <= ub){
        temp[k++] = arr[j++];
    }

    // updating Original Array
    for(int l=lb;l<=ub;l++){
        arr[l]=temp[l];
    }
}

void mergeSort(int lb,int ub) {
    if (lb != ub){
        int mid;
        mid = (lb+ub)/2;

        mergeSort(lb,mid);
        mergeSort(mid+1,ub);

        merge(lb,mid,ub);
    }
}

int main()
{
    int i;
```

```

printf("Enter 6 Array Values(Integer) :\n");
for(i=0;i<= max-1;i++){
    scanf("%d",&arr[i]);
}

printf("\nThe Array is : ");
display(arr,max);

mergeSort(0,max-1);

printf("\nThe Sorted Array is : ");
display(arr,max);

return 0;
}

```

Output

Enter 6 Array Values(Integer) : 9 8 7 6 5 4
The Array is : 9 8 7 6 5 4
The Sorted Array is : 4 5 6 7 8 9

Watch Videos: -

[Two Way MergeSort - Iterative method](#)
[Merge Sort Algorithm](#)
[Merge Sort Algorithm](#)
[MergeSort in-depth Analysis](#)

Quick Sort

Quicksort is a sorting algorithm based on the **divide and conquer approach** where

- An array is divided into subarrays by selecting a **pivot element** (element selected from the array).

While dividing the array, the **pivot element** should be positioned in such a way that elements **less than pivot** are kept on the left side and elements greater than pivot are on the right side of the pivot.

- The left and right subarrays are also divided using the same approach. This process continues until each subarray contains a single element.
- At this point, elements are already sorted. Finally, elements are combined to form a sorted array.

Working of Quicksort Algorithm

1. Select the Pivot Element

There are different variations of quicksort where the pivot element is selected from different positions. Here, we will be selecting the rightmost element of the array as the pivot element.



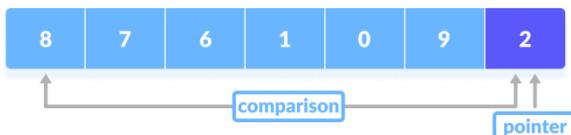
2. Rearrange the Array

Now the elements of the array are rearranged so that elements that are smaller than the pivot are put on the left and the elements greater than the pivot are put on the right.

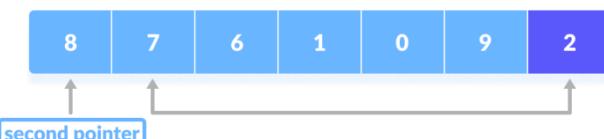


Here's how we rearrange the array:

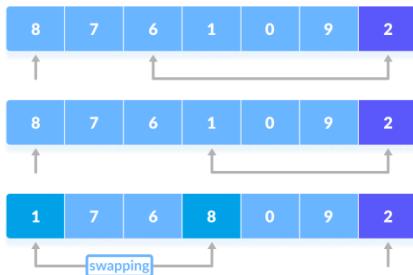
- A pointer is fixed at the pivot element. The pivot element is compared with the elements beginning from the first index.



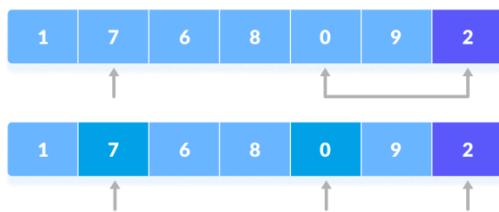
- If the element is greater than the pivot element, a second pointer is set for that element.



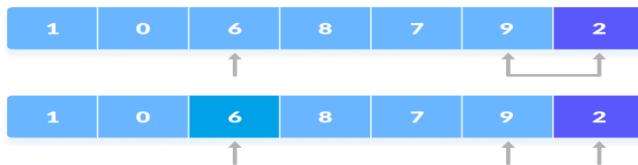
- Now, pivot is compared with other elements. If an element smaller than the pivot element is reached, the smaller element is swapped with the greater element found earlier.



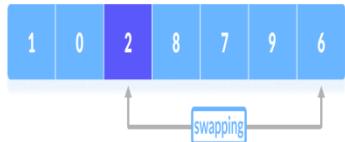
- Again, the process is repeated to set the next greater element as the second pointer. And, swap it with another smaller element.



- The process goes on until the second last element is reached.



- Finally, the pivot element is swapped with the second pointer.



3. Divide Subarrays

Pivot elements are again chosen for the left and the right sub-parts separately. And, step 2 is repeated.



The subarrays are divided until each subarray is formed of a single element. At this point, the array is already sorted.

Quick Sort Algorithm

```
quickSort(array, leftmostIndex, rightmostIndex)
  if (leftmostIndex < rightmostIndex)
    pivotIndex <- partition(array, leftmostIndex, rightmostIndex)
    quickSort(array, leftmostIndex, pivotIndex - 1)
    quickSort(array, pivotIndex, rightmostIndex)

partition(array, leftmostIndex, rightmostIndex)
  set rightmostIndex as pivotIndex
  storeIndex <- leftmostIndex - 1
  for i <- leftmostIndex + 1 to rightmostIndex
    if element[i] < pivotElement
      swap element[i] and element[storeIndex]
      storeIndex++
  swap pivotElement and element[storeIndex+1]
  return storeIndex + 1
```

Program to implement Quick Sort

```

// Quick sort
#include <stdio.h>
#define max 6
int Arr[max];

// function to swap elements
void swap(int *a, int *b) {
    int t = *a;
    *a = *b;
    *b = t;
}

// function to find the partition position
int partition(int array[], int low, int high) {

    // select the rightmost element as pivot
    int pivot = array[high];

    // pointer for greater element
    int i = (low - 1);

    // traverse each element of the array
    // compare them with the pivot
    for (int j = low; j < high; j++) {

        if (array[j] <= pivot) {

            // if element smaller than pivot is found
            // swap it with the greater element pointed by i
            i++;
            // swap element at i with element at j
            swap(&array[i], &array[j]);
        }
    }

    // swap the pivot element with the greater element at i
    swap(&array[i + 1], &array[high]);

    // return the partition point
    return (i + 1);
}

void quickSort(int array[], int low, int high) {
    if (low < high) {

        // find the pivot element such that
        // elements smaller than pivot are on left of pivot
        // elements greater than pivot are on right of pivot
        int pivot = partition(array, low, high);

        // recursive call on the left of pivot
        quickSort(array, low, pivot - 1);

        // recursive call on the right of pivot
        quickSort(array, pivot + 1, high);
    }
}

```

```

// function to print array elements
void printArray(int array[], int size) {
    for (int i = 0; i < size; ++i) {
        printf("%d ", array[i]);
    }
    printf("\n");
}

int main() {

    int i;
    printf("Enter 6 Array Values(Integer) :\n");
    for(i=0;i<= max-1;i++) {
        scanf("%d",&Arr[i]);
    }

    printf("\nThe Unsorted Array : ");
    printArray(Arr, max);

    quickSort(Arr, 0, max - 1);

    printf("The Sorted Array : ");
    printArray(Arr, max);
}

```

Output

```

Enter 6 Array Values(Integer) : 9 8 7 6 5 4
The Unsorted Array : 9 8 7 6 5 4
The Sorted Array : 4 5 6 7 8 9

```

Watch Videos: -

[Quick Sort Algorithm](#)
[QuickSort Algorithm](#)
[QuickSort Analysis](#)

Counting Sort

Counting sort is a sorting algorithm **that sorts the elements of an array by counting the number of occurrences of each unique element in the array**. The count is stored in an auxiliary array and the sorting is done by mapping the count as an index of the auxiliary array.

Working of Counting Sort

Find out the maximum element (let it be **max**) from the given array.

max								
8	4	2	2	8	3	3	1	

Initialize an **array of length $\text{max}+1$** with all elements 0. This array is used for storing the count of the elements in the array.

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

Store the count of each element at their respective index in **count** array

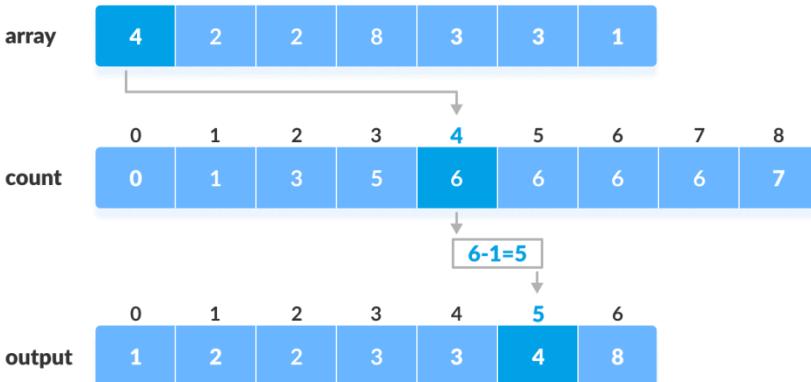
For example: if the count of element 3 is 2 then, 2 is stored in the 3rd position of count array. If element "5" is not present in the array, then 0 is stored in 5th position.

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

Store **cumulative sum of the elements of the count array**. It helps in placing the elements into the correct index of the sorted array.

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

Find the index of each element of the original array in the count array. This gives the cumulative count. Place the element at the index calculated as shown in figure below.



After placing each element at its correct position, decrease its count by one.

Counting Sort Algorithm

```
countingSort(array, size)
    max <- find largest element in array
    initialize count array with all zeros
    for j <- 0 to size
        find the total count of each unique element and
        store the count at jth index in count array
    for i <- 1 to max
        find the cumulative sum and store it in count array itself
    for j <- size down to 1
        restore the elements to array
        decrease count of each element restored by 1
```

Program to implement Counting Sort

```

// Counting Sort
#include <stdio.h>

void countingSort(int array[], int size) {
    int output[10];

    // Find the largest element of the array
    int max = array[0];
    for (int i = 1; i < size; i++) {
        if (array[i] > max)
            max = array[i];
    }

    // The size of count must be at least (max+1) but
    // we cannot declare it as int count(max+1) in C as
    // it does not support dynamic memory allocation.
    // So, its size is provided statically.
    int count[10];

    // Initialize count array with all zeros.
    for (int i = 0; i <= max; ++i) {
        count[i] = 0;
    }

    // Store the count of each element
    for (int i = 0; i < size; i++) {
        count[array[i]]++;
    }

    // Store the cumulative count of each array
    for (int i = 1; i <= max; i++) {
        count[i] += count[i - 1];
    }

    // Find the index of each element of the original array in count array, and
    // place the elements in output array
    for (int i = size - 1; i >= 0; i--) {
        output[count[array[i]] - 1] = array[i];
        count[array[i]]--;
    }

    // Copy the sorted elements into original array
    for (int i = 0; i < size; i++) {
        array[i] = output[i];
    }
}

// Function to print an array
void printArray(int array[], int size) {
    for (int i = 0; i < size; ++i) {
        printf("%d ", array[i]);
    }
    printf("\n");
}

int main() {
    int size;

```

```
printf("Enter the Size of the array : ");
scanf("%d", &size);

int array[size];

printf("Enter 6 Array Values(Integer) :\n");
for(int i=0;i<= size-1;i++){
    scanf("%d",&array[i]);
}

printf("\nThe Unsorted Array : ");
printArray(array, size);

countingSort(array, size);

printf("The Sorted Array : ");
printArray(array, size);

return 0;
}
```

Output

```
Enter the Size of the array : 6
Enter 6 Array Values(Integer) : 9 8 7 6 5 4
The Unsorted Array : 9 8 7 6 5 4
The Sorted Array : 4 5 6 7 8 9
```

Watch Videos: -

[Counting Sort](#)

Radix Sort

Radix sort is a sorting algorithm that sorts the elements by first grouping the individual digits of the same place value. Then, sort the elements according to their increasing/decreasing order.

Suppose, we have an array of 8 elements. First, we will sort elements based on the value of the unit place. Then, we will sort elements based on the value of the tenth place. This process goes on until the last significant place.

Let the initial array be [121, 432, 564, 23, 1, 45, 788]. It is sorted according to radix sort as shown in the figure below.

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

sorting the integers according to units, tens and hundreds place digits

Working of Radix Sort

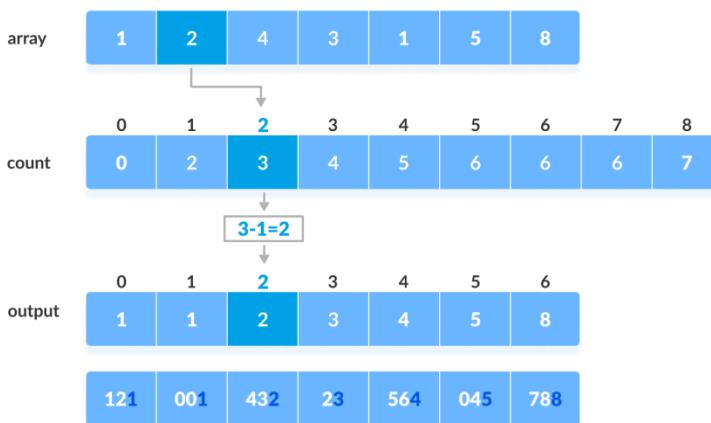
- Find the largest element in the array, i.e. **max**. Let **X** be the number of digits in **max**. **X** is calculated because we have to go through all the significant places of all elements.

In this array [121, 432, 564, 23, 1, 45, 788], we have the largest number 788. It has 3 digits. Therefore, the loop should go up to hundreds place (3 times).

- Now, go through each significant place one by one.

Use any stable sorting technique to sort the digits at each significant place. We have used counting sort for this.

Sort the elements based on the unit place digits (**X=0**).



3. Now, sort the elements based on digits at tens place.

001	121	023	432	045	564	788
-----	-----	-----	-----	-----	-----	-----

4. Finally, sort the elements based on the digits at hundreds place.

001	023	045	121	432	564	788
-----	-----	-----	-----	-----	-----	-----

Radix Sort Algorithm

```

radixSort(array)
    d <- maximum number of digits in the largest element
    create d buckets of size 0-9
    for i <- 0 to d
        sort the elements according to ith place digits using countingSort

countingSort(array, d)
    max <- find largest element among dth place elements
    initialize count array with all zeros
    for j <- 0 to size
        find the total count of each unique digit in dth place of elements and
        store the count at jth index in count array
    for i <- 1 to max
        find the cumulative sum and store it in count array itself
    for j <- size down to 1
        restore the elements to array
        decrease count of each element restored by 1

```

Program to implement Radix Sort

```

// Radix Sort
#include <stdio.h>

// Function to get the largest element from an array
int getMax(int array[], int n) {
    int max = array[0];
    for (int i = 1; i < n; i++)
        if (array[i] > max)
            max = array[i];
    return max;
}

// Using counting sort to sort the elements in the basis of significant places
void countSort(int arr[], int size, int pos)
{
    int output[size]; // output array
    int i, count[10] = { 0 };

    // Store count of occurrences in count[]
    for (i = 0; i < size; i++)
        ++count[(arr[i] / pos) % 10];

    // Change count[i] so that count[i] now contains actual
    // position of this digit in output[]
    for (i = 1; i < 10; i++)
        count[i] = count[i] + count[i - 1];

    // Build the output array
    for (i = size - 1; i >= 0; i--) {
        output[ --count[(arr[i] / pos) % 10]] = arr[i];
    }

    // Copy the output array to arr[], so that arr[] now
    // contains sorted numbers according to current digit
    for (i = 0; i < size; i++)
        arr[i] = output[i];
}

void radixsort(int array[], int size) {
    // Get maximum element
    int max = getMax(array, size);

    // Apply counting sort to sort elements based on place value.
    for (int place = 1; max / place > 0; place *= 10)
        countSort(array, size, place);
}

// Print an array
void printArray(int array[], int size) {
    for (int i = 0; i < size; ++i) {
        printf("%d ", array[i]);
    }
    printf("\n");
}

int main() {
    int size;

```

```
printf("Enter the Size of the array : ");
scanf("%d", &size);

int array[size];

printf("Enter 6 Array Values(Integer) :\n");
for(int i=0;i<= size-1;i++) {
    scanf("%d",&array[i]);
}

printf("\nThe Unsorted Array : ");
printArray(array, size);

radixsort(array, size);

printf("The Sorted Array : ");
printArray(array, size);
}
```

Output

```
Enter the Size of the array : 6
Enter 6 Array Values(Integer) : 9 8 7 6 5 4
The Unsorted Array : 9 8 7 6 5 4
The Sorted Array : 4 5 6 7 8 9
```

Watch Videos: -

[Radix Sort](#)
[Radix Sort](#)

Bucket Sort

Bucket Sort is a sorting algorithm that divides the unsorted array elements into several groups called buckets. Each bucket is then sorted by using any of the suitable sorting algorithms or recursively applying the same bucket algorithm. Finally, the sorted buckets are combined to form a final sorted array.

Scatter Gather Approach

The process of bucket sort can be understood as a scatter-gather approach. Here, **elements are first scattered into buckets then the elements in each bucket are sorted**. Finally, the elements are gathered in order.



Working of Bucket Sort

- Suppose, the input array is:

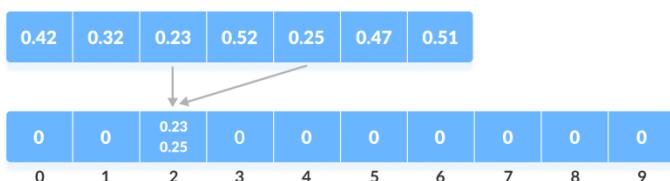
0.42	0.32	0.23	0.52	0.25	0.47	0.51
------	------	------	------	------	------	------

Create an array of size 10. Each slot of this array is used as a bucket for storing elements.

0	0	0	0	0	0	0	0	0	0
---	---	---	---	---	---	---	---	---	---

- Insert elements into the buckets from the array. The elements are inserted according to the range of the bucket.

In our example code, we have buckets each of ranges from 0 to 1, 1 to 2, 2 to 3,..... (n-1) to n. Suppose, an input element is .23 is taken. It is multiplied by size = 10 (ie. $.23 \times 10 = 2.3$). Then, it is converted into an integer (ie. $2.3 \approx 2$). Finally, .23 is inserted into bucket-2.



Similarly, .25 is also inserted into the same bucket. Everytime, the floor value of the floating point number is taken.

If we take integer numbers as input, we have to divide it by the interval (10 here) to get the floor value.

Similarly, other elements are inserted into their respective buckets.

0	0	0.23 0.25	0.32	0.42 0.47	0.52 0.51	0	0	0	0
0	1	2	3	4	5	6	7	8	9

3. The elements of each bucket are sorted using any of the stable sorting algorithms. Here, we have used quicksort (inbuilt function).

0	0	0.23 0.25	0.32	0.42 0.47	0.51 0.52	0	0	0	0
0	1	2	3	4	5	6	7	8	9

4. The elements from each bucket are gathered.

It is done by iterating through the bucket and inserting an individual element into the original array in each cycle. The element from the bucket is erased once it is copied into the original array.

Bucket Sort Algorithm

```

bucketSort()
  create N buckets each of which can hold a range of values
  for all the buckets
    initialize each bucket with 0 values
  for all the buckets
    put elements into buckets matching the range
  for all the buckets
    sort elements in each bucket
  gather elements from each bucket
end bucketSort

```

Program to implement Bucket Sort

```
// Bucket Sort
#include <stdio.h>

// function to get maximum element from the given array
int getMax(int a[], int n)
{
    int max = a[0];
    for (int i = 1; i < n; i++)
        if (a[i] > max)
            max = a[i];
    return max;
}

void bucket(int a[], int n){
    //max is the maximum element of array
    int max = getMax(a, n);
    int bucket[max], i;
    for (int i = 0; i <= max; i++) {
        bucket[i] = 0;
    }

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

    for (int i = 0, j = 0; i <= max; i++) {
        while (bucket[i] > 0) {
            a[j++] = i;
            bucket[i]--;
        }
    }
}

// function to print array elements
void printArray(int a[], int n)
{
    for (int i = 0; i < n; ++i)
        printf("%d ", a[i]);
}

int main(){
    int size;

    printf("Enter the Size of the array : ");
    scanf("%d", &size);

    int array[size];

    printf("Enter 6 Array Values(Integer) :\n");
    for(int i=0;i<= size-1;i++){
        scanf("%d",&array[i]);
    }

    printf("\nThe Unsorted Array : ");
    printArray(array, size);

    bucket(array, size);
}
```

```
printf("\nThe Sorted Array : ");
printArray(array, size);

return 0;
}
```

Output

```
Enter the Size of the array : 6
Enter 6 Array Values(Integer) : 9 8 7 6 5 4
The Unsorted Array : 9 8 7 6 5 4
The Sorted Array : 4 5 6 7 8 9
```

Watch Videos: -

[Bucket Sort](#)

Heap Sort

Heap Sort is a popular and efficient sorting algorithm in computer programming. Learning how to write the **heap sort algorithm** requires knowledge of two types of data structures - arrays and trees.

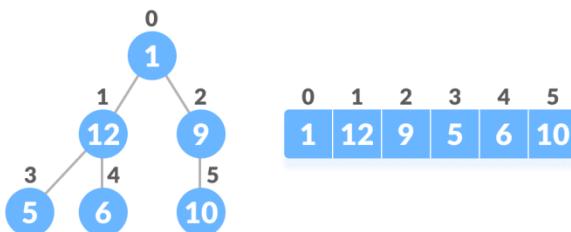
The initial set of numbers that we want to sort is stored in an array e.g. [10, 3, 76, 34, 23, 32] and after sorting, we get a sorted array [3,10,23,32,34,76].

Heap sort works by visualizing the elements of the array as a special kind of complete binary tree called a heap.

Relationship between Array Indexes and Tree Elements

A complete binary tree has an interesting property that we can use to find the children and parents of any node.

If the index of any element in the array is i, the element in the index $2i+1$ will become the left child and element in $2i+2$ index will become the right child. Also, the parent of any element at index i is given by the lower bound of $(i-1)/2$.



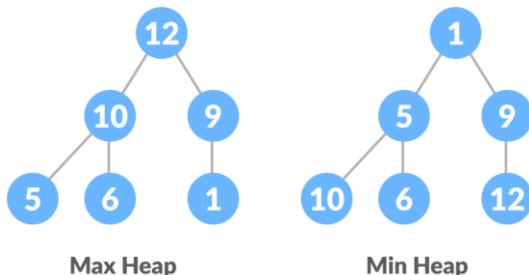
Understanding this mapping of array indexes to tree positions is critical to understanding how the Heap Data Structure works and how it is used to implement Heap Sort.

What is Heap Data Structure?

Heap is a special tree-based data structure. A binary tree is said to follow a heap data structure if

- o it is a complete binary tree
- o All nodes in the tree follow the property that they are greater than their children i.e. the largest element is at the root and both its children and smaller than the root and so on. Such a heap is called a max-heap. If instead, all nodes are smaller than their children, it is called a min-heap

The following example diagram shows Max-Heap and Min-Heap.



How to "heapify" a tree

Starting from a complete binary tree, we can modify it to become a Max-Heap by running a function called **heapify** on all the non-leaf elements of the heap.

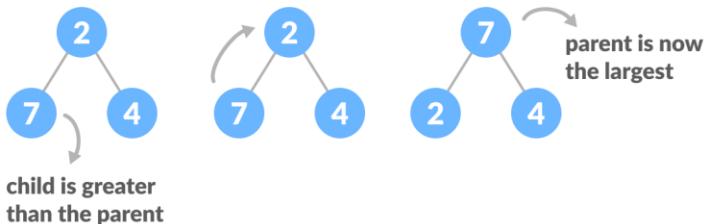
Since heapify uses recursion, it can be difficult to grasp. So let's first think about how you would heapify a tree with just three elements.

```
heapify(array)
    Root = array[0]
    Largest = largest( array[0] , array [2*0 + 1]. array[2*0+2])
    if(Root != Largest)
        Swap(Root, Largest)
```

Scenario-1



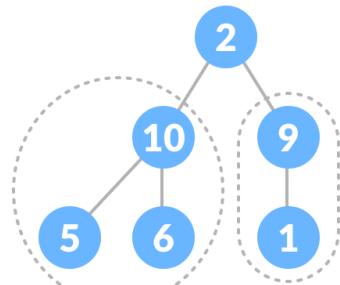
Scenario-2



The example above shows two scenarios - one in which the root is the largest element and we don't need to do anything. And another in which the root had a larger element as a child and we needed to swap to maintain max-heap property.

If you've worked with recursive algorithms before, you've probably identified that this must be the base case.

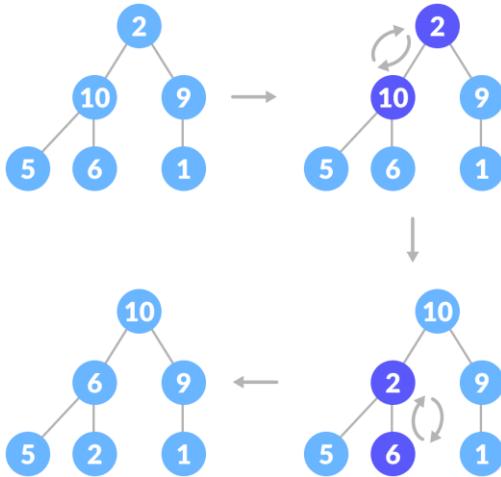
Now let's think of another scenario in which there is more than one level.



**both subtrees of the root
are already max-heaps**

The top element isn't a max-heap but all the sub-trees are max-heaps.

To maintain the max-heap property for the entire tree, we will have to keep pushing 2 downwards until it reaches its correct position.



Thus, to maintain the max-heap property in a tree where both sub-trees are max-heaps, we need to run heapify on the root element repeatedly until it is larger than its children or it becomes a leaf node.

We can combine both these conditions in one heapify function as

```
void heapify(int arr[], int n, int i) {
    // Find largest among root, left child and right child
    int largest = i;
    int left = 2 * i + 1;
    int right = 2 * i + 2;

    if (left < n && arr[left] > arr[largest])
        largest = left;

    if (right < n && arr[right] > arr[largest])
        largest = right;

    // Swap and continue heapifying if root is not largest
    if (largest != i) {
        swap(&arr[i], &arr[largest]);
        heapify(arr, n, largest);
    }
}
```

This function works for both the base case and for a tree of any size. We can thus move the root element to the correct position to maintain the max-heap status for any tree size as long as the sub-trees are max-heaps.

Build max-heap

To build a max-heap from any tree, we can thus start heapifying each sub-tree from the bottom up and end up with a max-heap after the function is applied to all the elements including the root element.

In the case of a complete tree, the first index of a non-leaf node is given by $n/2 - 1$. All other nodes after that are leaf-nodes and thus don't need to be heapified.

So, we can build a maximum heap as

```
// Build heap (rearrange array)
for (int i = n / 2 - 1; i >= 0; i--)
    heapify(arr, n, i);
```

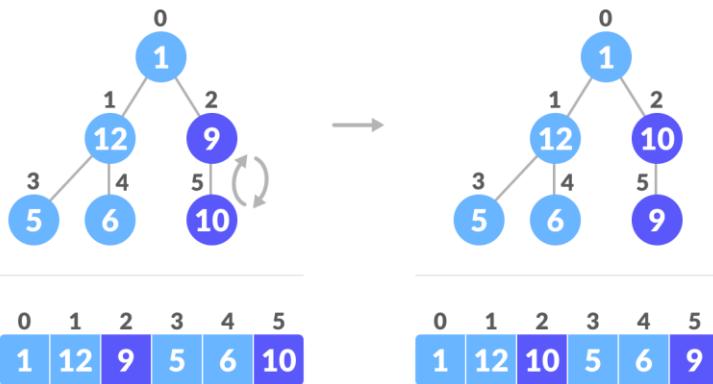
	0	1	2	3	4	5
arr	1	12	9	5	6	10

$$n = 6$$

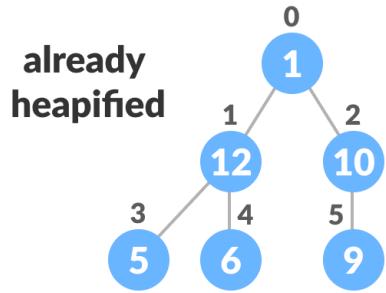
$$i = 6/2 - 1 = 2 \text{ # loop runs from 2 to 0}$$

Create array and calculate i

i = 2 → heapify(arr, 6, 2)

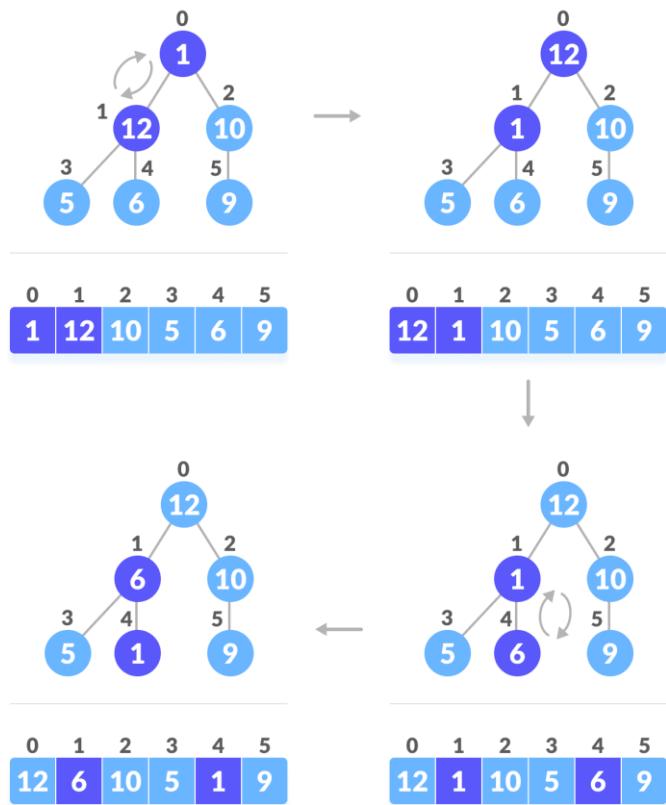


$i = 1 \rightarrow \text{heapify}(arr, 6, 1)$



0	1	2	3	4	5
1	12	10	5	6	9

$i = 0 \rightarrow \text{heapify}(arr, 6, 0)$

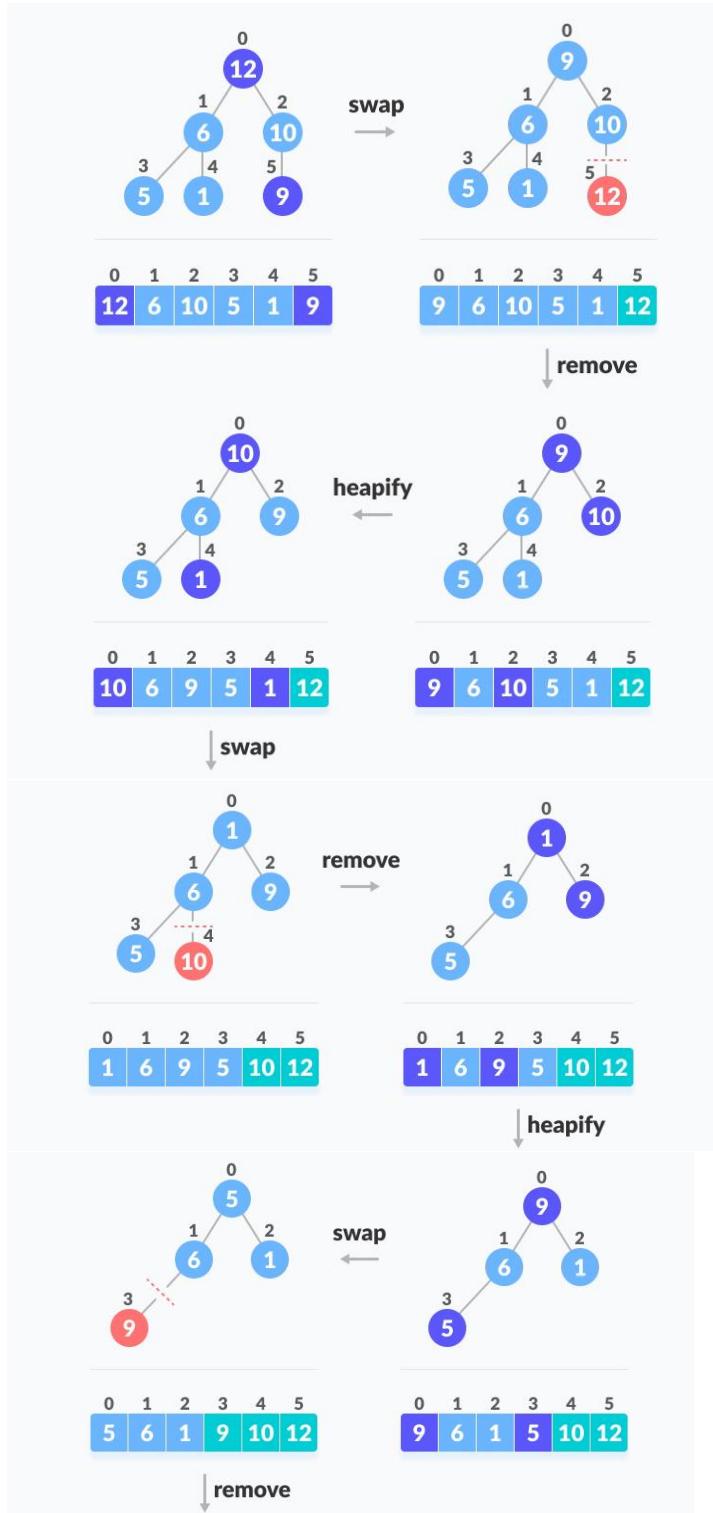


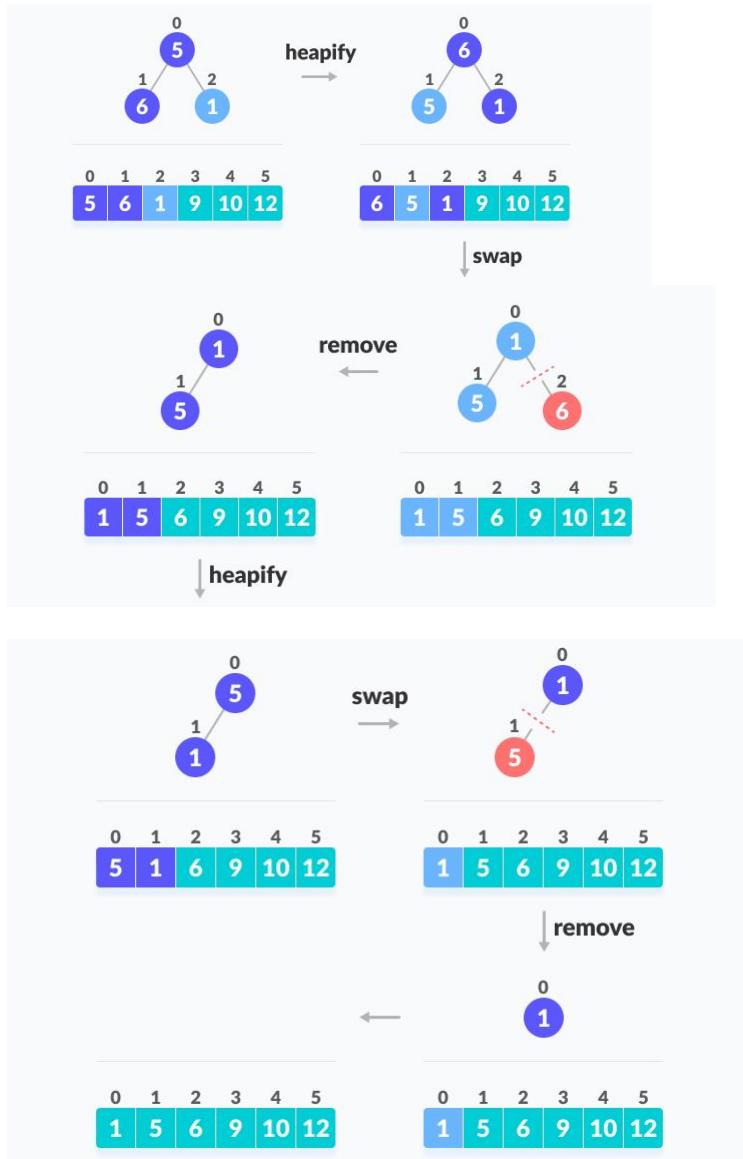
As shown in the above diagram, we start by heapifying the lowest smallest trees and gradually move up until we reach the root element.

If you've understood everything till here, congratulations, you are on your way to mastering the Heap sort.

Working of Heap Sort

1. Since the tree satisfies Max-Heap property, then the largest item is stored at the root node.
2. **Swap:** Remove the root element and put at the end of the array (nth position) Put the last item of the tree (heap) at the vacant place.
3. **Remove:** Reduce the size of the heap by 1.
4. **Heapify:** Heapify the root element again so that we have the highest element at root.
5. The process is repeated until all the items of the list are sorted.





The code below shows the operation.

```
// Heap sort
for (int i = n - 1; i >= 0; i--) {
    swap(&arr[0], &arr[i]);

    // Heapify root element to get highest element at root again
    heapify(arr, i, 0);
}
```

Program to implement Heap Sort

```

// Heap Sort
#include <stdio.h>

// Function to swap the the position of two elements
void swap(int *a, int *b) {
    int temp = *a;
    *a = *b;
    *b = temp;
}

void heapify(int arr[], int n, int i) {
    // Find largest among root, left child and right child
    int largest = i;
    int left = 2 * i + 1;
    int right = 2 * i + 2;

    if (left < n && arr[left] > arr[largest])
        largest = left;

    if (right < n && arr[right] > arr[largest])
        largest = right;

    // Swap and continue heapifying if root is not largest
    if (largest != i) {
        swap(&arr[i], &arr[largest]);
        heapify(arr, n, largest);
    }
}

// Main function to do heap sort
void heapSort(int arr[], int n) {
    // Build max heap
    for (int i = n / 2 - 1; i >= 0; i--)
        heapify(arr, n, i);

    // Heap sort
    for (int i = n - 1; i >= 0; i--) {
        swap(&arr[0], &arr[i]);

        // Heapify root element to get highest element at root again
        heapify(arr, i, 0);
    }
}

// Print an array
void printArray(int arr[], int n) {
    for (int i = 0; i < n; ++i)
        printf("%d ", arr[i]);
    printf("\n");
}

int main() {
    int size;

    printf("Enter the Size of the array : ");
    scanf("%d", &size);
}

```

```

int array[size];

printf("Enter 6 Array Values(Integer) :\n");
for(int i=0;i<= size-1;i++){
    scanf("%d",&array[i]);
}

printf("\nThe Unsorted Array : ");
printArray(array, size);

heapSort(array, size);

printf("The Sorted Array : ");
printArray(array, size);

}

```

Output

Enter the Size of the array : 6
 Enter 6 Array Values(Integer) : 9 8 7 6 5 4
 The Unsorted Array : 9 8 7 6 5 4
 The Sorted Array : 4 5 6 7 8 9

Watch Videos: -

[Heap Sort](#)
[Heap - Heap Sort - Heapify - Priority Queues](#)

Shell Sort

Shell sort is a generalized version of the insertion sort algorithm. It first sorts elements that are far apart from each other and successively reduces the interval between the elements to be sorted.

The interval between the elements is reduced based on the sequence used. Some of the optimal sequences that can be used in the shell sort algorithm are:

Shell's original sequence: $N/2, N/4, \dots, 1$

Knuth's increments: $1, 4, 13, \dots, (3^k - 1)/2$

Sedgewick's increments: $1, 8, 23, 77, 281, 1073, 4193, 16577, \dots, 4j+1 + 3 \cdot 2^{j-1}$

Hibbard's increments: $1, 3, 7, 15, 31, 63, 127, 255, 511, \dots$

Papernov & Stasevich increment: $1, 3, 5, 9, 17, 33, 65, \dots$

Pratt: $1, 2, 3, 4, 6, 9, 8, 12, 18, 27, 16, 24, 36, 54, 81, \dots$

Working of Shell Sort

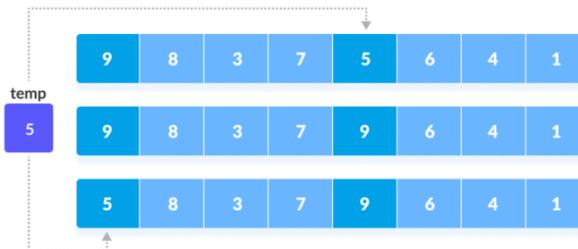
- Suppose, we need to sort the following array.

9	8	3	7	5	6	4	1
---	---	---	---	---	---	---	---

- We are using the shell's original sequence ($N/2, N/4, \dots, 1$) as intervals in our algorithm.

In the first loop, if the array size is $N = 8$ then, the elements lying at the interval of $N/2 = 4$ are compared and swapped if they are not in order.

- The 0th element is compared with the 4th element.
- If the 0th element is greater than the 4th one then, the 4th element is first stored in `temp` variable and the 0th element (ie. greater element) is stored in the 4th position and the element stored in `temp` is stored in the 0th position.



This process goes on for all the remaining elements.

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

3. In the second loop, an interval of $N/4 = 8/4 = 2$ is taken and again the elements lying at these intervals are sorted.

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

You might get confused at this point.

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

The elements at 4th and 2nd position are compared. The elements at 2nd and 0th position are also compared. All the elements in the array lying at the current interval are compared.

4. The same process goes on for remaining elements.

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

5. Finally, when the interval is $N/8 = 8/8 = 1$ then the array elements lying at the interval of 1 are sorted. The array is now completely sorted.

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

Shell Sort Algorithm

```
shellSort(array, size)
  for interval i <- size/2n down to 1
    for each interval "i" in array
      sort all the elements at interval "i"
end shellSort
```

Program to implement Shell Sort

```
// Shell Sort

#include <stdio.h>

void shellSort(int array[], int n) {
    // Rearrange elements at each n/2, n/4, n/8, ... intervals
    for (int interval = n / 2; interval > 0; interval /= 2) {
        for (int i = interval; i < n; i += 1) {
            int temp = array[i];
            int j;
            for (j = i; j >= interval && array[j - interval] > temp; j -= interval) {
                array[j] = array[j - interval];
            }
            array[j] = temp;
        }
    }
}

// Print an array
void printArray(int arr[], int n) {
    for (int i = 0; i < n; ++i)
        printf("%d ", arr[i]);
    printf("\n");
}

int main() {

    int size;
    printf("Enter the size of the array : ");
    scanf("%d", &size);

    int array[size];
    printf("Enter Array Values(Integer) :\n");
    for(int i=0;i<= size-1;i++){
        scanf("%d",&array[i]);
    }

    printf("\nThe Unsorted Array : ");
    printArray(array, size);

    shellSort(array, size);
    printf("Sorted array: ");
    printArray(array, size);
}
}
```

Output

```
Enter the size of the array : 6
Enter Array Values(Integer) : 9 8 7 6 5 4
The Unsorted Array : 9 8 7 6 5 4
Sorted array: 4 5 6 7 8 9
```

Watch Videos: -

[Shell Sort algorithm](#)

Linear Search

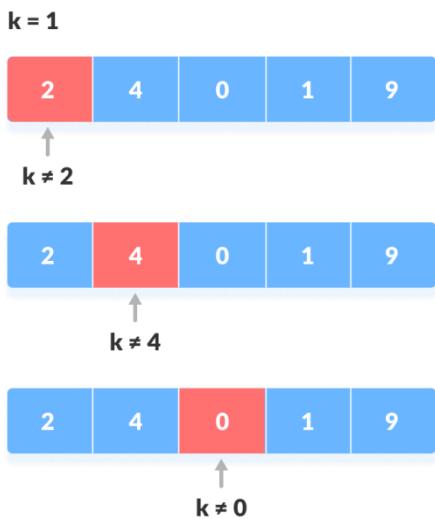
Linear search is a sequential searching algorithm where we start from one end and check every element of the list until the desired element is found. It is the simplest searching algorithm.

How Linear Search Works?

The following steps are followed to search for an element $k = 1$ in the list below.

2	4	0	1	9
---	---	---	---	---

1. Start from the first element, compare k with each element x .



2. If $x == k$, return the index.

2	4	0	1	9
---	---	---	---	---

\uparrow

$k = 1$

3. Else, return not found.

Linear Search Algorithm

```
LinearSearch(array, key)
  for each item in the array
    if item == value
      return its index
```

Program to Implement Linear Search

```
// Linear Search
#include <stdio.h>

int search(int array[], int size, int item)
{
    int i;
    for(i=0;i<size;i++) {
        if(array[i]==item) {
            printf("The element found at the index %d",i);
            break;
        }
    }
    if(i==size)
        printf("The element not found");
}

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

int main()
{
    int size,item;
    printf("Enter the size of the array : ");
    scanf("%d", &size);

    int array[size];
    printf("Enter Array Values(Integer) :\n");
    for(int i=0;i<= size-1;i++) {
        scanf("%d",&array[i]);
    }

    printf("\nThe Array is : ");
    printArray(array, size);

    printf("\nEnter the element for search : ");
    scanf("%d",&item);

    search(array, size, item);
}
```

Output

```
Enter the size of the array : 6
Enter Array Values(Integer) : 10 20 30 40 50 60
The Array is : 10 20 30 40 50 60
Enter the element for search : 30
The element found at the index 2
```

Watch Videos: -

[Linear Search](#)

Binary Search

Binary Search is a searching algorithm for finding an element's position in a sorted array. In this approach, the element is always searched in the middle of a portion of an array.

Binary search can be implemented only on a sorted list of items. If the elements are not sorted already, we need to sort them first.

Binary Search Working

Binary Search Algorithm can be implemented in two ways which are discussed below.

- Iterative Method
- Recursive Method

The recursive method follows the divide and conquer approach.

The general steps for both methods are discussed below.

1. The array in which searching is to be performed is:

3	4	5	6	7	8	9
---	---	---	---	---	---	---

Let $x = 4$ be the element to be searched.

2. Set two pointers low and high at the lowest and the highest positions respectively.



3. Find the middle element mid of the array ie. $\text{arr}[(\text{low} + \text{high})/2] = 6$.



4. If $x == \text{mid}$, then return mid . Else, compare the element to be searched with m .
5. If $x > \text{mid}$, compare x with the middle element of the elements on the right side of mid . This is done by setting low to $\text{low} = \text{mid} + 1$.
6. Else, compare x with the middle element of the elements on the left side of mid . This is done by setting high to $\text{high} = \text{mid} - 1$.



7. Repeat steps 3 to 6 until low meets high.



8. $x = 4$ is found.



Binary Search Algorithm

Iteration Method

```
do until the pointers low and high meet each other.
    mid = (low + high)/2
    if (x == arr[mid])
        return mid
    else if (x > arr[mid]) // x is on the right side
        low = mid + 1
    else                      // x is on the left side
        high = mid - 1
```

Recursive Method

```
binarySearch(arr, x, low, high)
    if low > high
        return False
    else
        mid = (low + high) / 2
        if x == arr[mid]
            return mid
        else if x > arr[mid]      // x is on the right side
            return binarySearch(arr, x, mid + 1, high)
        else                      // x is on the left side
            return binarySearch(arr, x, low, mid - 1)
```

Program to implement Binary Search

```
#include <stdio.h>
int binarySearch(int array[], int size,int item)
{
    int low    = 0;
    int high   = size - 1;
    // Repeat until the pointers low and high meet each other
    while (low <= high) {
        int mid = (low + high) / 2;
        if (array[mid] == item)
            return mid;
        if (array[mid] < item)
            low = mid + 1;
        else
            high = mid - 1;
    }
    return -1;
}
void printArray(int arr[], int n) {
    for (int i = 0; i < n; ++i)
        printf("%d ", arr[i]);
    printf("\n");
}
int main()
{
    int size,item;
    printf("Enter the size of the array : ");
    scanf("%d", &size);
    int array[size];
    printf("Enter Array Values(Integer) :\n");
    for(int i=0;i<= size-1;i++){
        scanf("%d",&array[i]);
    }
    printf("\nThe Array is : ");
    printArray(array, size);
    printf("\nEnter the element for search : ");
    scanf("%d",&item);
    int result = binarySearch(array, size, item);
    if (result == -1)
        printf("The element Not found");
    else
        printf("The element is found at index %d ", result);
}
}
```

Output

```
Enter the size of the array : 6
Enter Array Values(Integer) : 9 8 7 6 5 4
The Array is : 9 8 7 6 5 4
Enter the element for search : 7
The element is found at index 2
```

Watch Videos: -

[Binary Search](#)
[Binary Search Iterative Method](#)
[Binary Search Recursive Method](#)

8. Greedy Algorithms

A greedy algorithm is an approach for solving a problem by selecting the best option available at the moment. It doesn't worry whether the current best result will bring the overall optimal result.

The algorithm never reverses the earlier decision even if the choice is wrong. It works in a top-down approach.

This algorithm may not produce the best result for all the problems. It's because it always goes for the local best choice to produce the global best result.

However, we can determine if the algorithm can be used with any problem if the problem has the following properties:

- **Greedy Choice Property**

If an optimal solution to the problem can be found by choosing the best choice at each step without reconsidering the previous steps once chosen, the problem can be solved using a greedy approach. This property is called greedy choice property.

- **Optimal Substructure**

If the optimal overall solution to the problem corresponds to the optimal solution to its subproblems, then the problem can be solved using a greedy approach. This property is called optimal substructure.

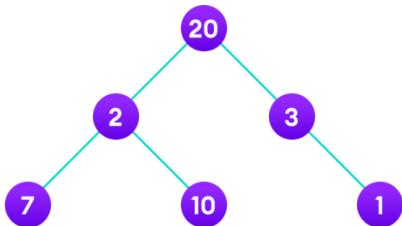
Advantages of Greedy Approach

- The algorithm is easier to describe.
- This algorithm can perform better than other algorithms (but, not in all cases).

Drawback of Greedy Approach

As mentioned earlier, the greedy algorithm doesn't always produce the optimal solution. This is the major disadvantage of the algorithm

For example, suppose we want to find the longest path in the graph below from root to leaf. Let's use the greedy algorithm here.

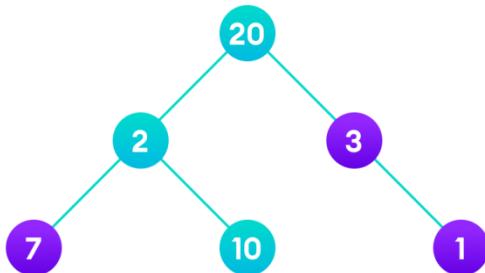


Greedy Approach

1. Let's start with the root node **20**. The weight of the right child is **3** and the weight of the left child is **2**.
2. Our problem is to find the largest path. And, the optimal solution at the moment is **3**. So, the greedy algorithm will choose **3**.

3. Finally the weight of an only child of **3** is **1**. This gives us our final result $20 + 3 + 1 = 24$.

However, it is not the optimal solution. There is another path that carries more weight ($20 + 2 + 10 = 32$) as shown in the image below.



Therefore, greedy algorithms do not always give an optimal/feasible solution.

Greedy Algorithm

- o To begin with, the solution set (containing answers) is empty.
- o At each step, an item is added to the solution set until a solution is reached.
- o If the solution set is feasible, the current item is kept.
- o Else, the item is rejected and never considered again.

Let's now use this algorithm to solve a problem.

Example - Greedy Approach

Problem: You have to make a change of an amount using the smallest possible number of coins.

Amount: \$18

Available coins are

\$5 coin
\$2 coin
\$1 coin

There is no limit to the number of each coin you can use.

Solution:

1. Create an empty `solution-set = {}`. Available coins are `{5, 2, 1}`.
2. We are supposed to find the `sum = 18`. Let's start with `sum = 0`.
3. Always select the coin with the largest value (i.e. 5) until the `sum > 18`. (When we select the largest value at each step, we hope to reach the destination faster. This concept is called **greedy choice property**.)
4. In the first iteration, `solution-set = {5}` and `sum = 5`.
5. In the second iteration, `solution-set = {5, 5}` and `sum = 10`.
6. In the third iteration, `solution-set = {5, 5, 5}` and `sum = 15`.
7. In the fourth iteration, `solution-set = {5, 5, 5, 2}` and `sum = 17`. (We cannot select 5 here because if we do so, `sum = 20` which is greater than 18. So, we select the 2nd largest item which is 2.)
8. Similarly, in the fifth iteration, select 1. Now `sum = 18` and `solution-set = {5, 5, 5, 2, 1}`.

Different Types of Greedy Algorithm

- o Selection Sort
- o Knapsack Problem
- o Minimum Spanning Tree
- o Single-Source Shortest Path Problem
- o Job Scheduling Problem
- o Prim's Minimal Spanning Tree Algorithm
- o Kruskal's Minimal Spanning Tree Algorithm
- o Dijkstra's Minimal Spanning Tree Algorithm
- o Huffman Coding
- o Ford-Fulkerson Algorithm

Watch Videos:-

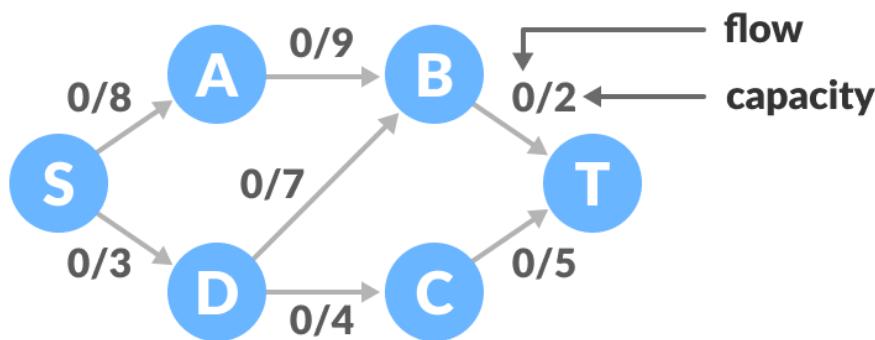
[Greedy Method - Introduction](#)

Ford-Fulkerson Algorithm

Ford-Fulkerson algorithm is a greedy approach for calculating the maximum possible flow in a network or a graph.

A term, **flow network**, is used to describe a network of vertices and edges with a source (**S**) and a sink (**T**). Each vertex, except **S** and **T**, can receive and send an equal amount of stuff through it. **S** can only send and **T** can only receive stuff.

We can visualize the understanding of the algorithm using a flow of liquid inside a network of pipes of different capacities. Each pipe has a certain capacity of liquid it can transfer at an instance. For this algorithm, we are going to find how much liquid can be flowed from the source to the sink at an instance using the network.



Flow network graph

Terminologies Used

- **Augmenting Path** :- It is the path available in a flow network.
- **Residual Graph** :- It represents the flow network that has additional possible flow.
- **Residual Capacity** :- It is the capacity of the edge after subtracting the flow from the maximum capacity.
- **Minimal Cut** :- also known as bottle neck capacity. Which decides maximum possible flow from source to sink through an augmented path.

Ford-Fulkerson Algorithm

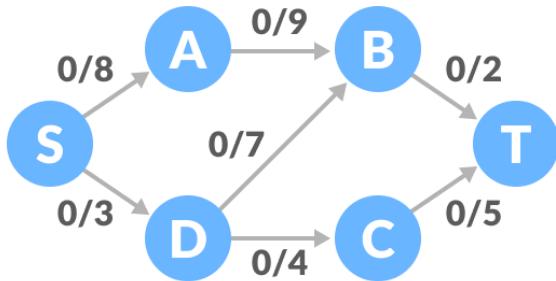
The algorithm follows:

- Initialize the flow in all the edges to 0.
- While there is an augmenting path between the source and the sink, add this path to the flow. / while there is an augmenting path from source to sink add two path flow to flow
- Update the residual graph./ Return flow

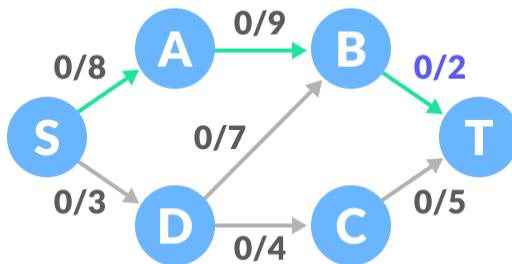
We can also consider reverse-path if required because if we do not consider them, we may never find a maximum flow.

Ford-Fulkerson Example

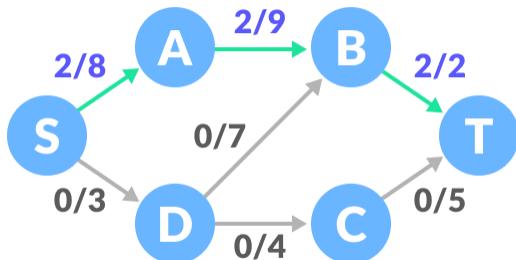
The flow of all the edges is 0 at the beginning.



1. Select any arbitrary path from S to T. In this step, we have selected path S-A-B-T.

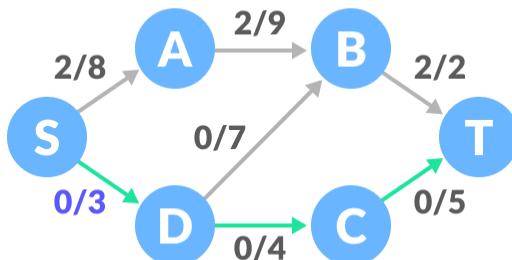


The minimum capacity among the three edges is 2 (B-T). Based on this, update the flow/capacity for each path.

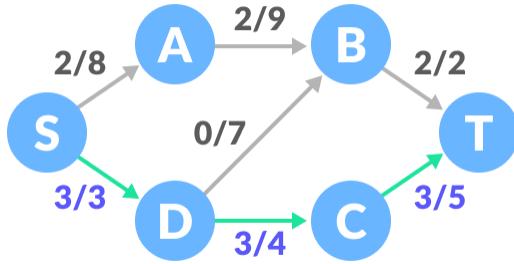


Update the capacities

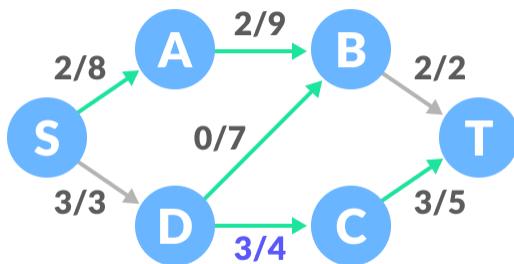
2. Select another path S-D-C-T. The minimum capacity among these edges is 3 (S-D).



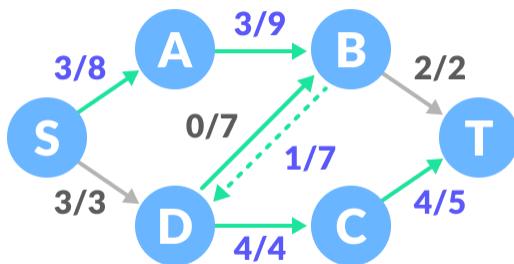
Update the capacities according to this.



3. Now, let us consider the reverse-path $B-D$ as well. Selecting path $S-A-B-D-C-T$. The minimum residual capacity among the edges is 1 ($D-C$).



Updating the capacities.



The capacity for forward and reverse paths are considered separately.

4. Adding all the flows = $2 + 3 + 1 = 6$, which is the maximum possible flow on the flow network.

Ford-Fulkerson Applications

- Water distribution pipeline
- Bipartite matching problem
- Circulation with demands

Program to implement Ford-Fulkerson algorithm

```
// Ford - Fulkerson algorithm
#include <stdio.h>

#define A 0
#define B 1
#define C 2
#define MAX_NODES 1000
#define O 10000000000

int n;
int e;
int capacity[MAX_NODES][MAX_NODES];
int flow[MAX_NODES][MAX_NODES];
int color[MAX_NODES];
int pred[MAX_NODES];

int min(int x, int y) {
    return x < y ? x : y;
}

int head, tail;
int q[MAX_NODES + 2];

void enqueue(int x) {
    q[tail] = x;
    tail++;
    color[x] = B;
}

int dequeue() {
    int x = q[head];
    head++;
    color[x] = C;
    return x;
}

// Using BFS as a searching algorithm
int bfs(int start, int target) {
    int u, v;
    for (u = 0; u < n; u++) {
        color[u] = A;
    }
    head = tail = 0;
    enqueue(start);
    pred[start] = -1;
    while (head != tail) {
        u = dequeue();
        for (v = 0; v < n; v++) {
            if (color[v] == A && capacity[u][v] - flow[u][v] > 0) {
                enqueue(v);
                pred[v] = u;
            }
        }
    }
    return color[target] == C;
}
```

```

// Applying fordfulkerson algorithm
int fordFulkerson(int source, int sink) {
    int i, j, u;
    int max_flow = 0;
    for (i = 0; i < n; i++) {
        for (j = 0; j < n; j++) {
            flow[i][j] = 0;
        }
    }

    // Updating the residual values of edges
    while (bfs(source, sink)) {
        int increment = 0;
        for (u = n - 1; pred[u] >= 0; u = pred[u]) {
            increment = min(increment, capacity[pred[u]][u] - flow[pred[u]][u]);
        }
        for (u = n - 1; pred[u] >= 0; u = pred[u]) {
            flow[pred[u]][u] += increment;
            flow[u][pred[u]] -= increment;
        }
        // Adding the path flows
        max_flow += increment;
    }
    return max_flow;
}

int main() {
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < n; j++) {
            capacity[i][j] = 0;
        }
    }
    n = 6;
    e = 7;

    capacity[0][1] = 8;
    capacity[0][4] = 3;
    capacity[1][2] = 9;
    capacity[2][4] = 7;
    capacity[2][5] = 2;
    capacity[3][5] = 5;
    capacity[4][2] = 7;
    capacity[4][3] = 4;

    int s = 0, t = 5;
    printf("Max Flow: %d\n", fordFulkerson(s, t));
}

```

Output

Max Flow: 6

Watch Videos:-

[Ford Fulkerson Algorithm for Maximum Flow Problem](#)

[Ford Fulkerson algorithm for Maximum Flow Problem Example](#)

[Ford Fulkerson algorithm for Maximum Flow Problem Complexity](#)

Dijkstra's Algorithm

Dijkstra's algorithm allows us to find the shortest path between any two vertices of a graph.

It differs from the minimum spanning tree because the shortest distance between two vertices might not include all the vertices of the graph.

How Dijkstra's Algorithm works

Dijkstra's Algorithm works on the basis that any subpath $B \rightarrow D$ of the shortest path $A \rightarrow D$ between vertices A and D is also the shortest path between vertices B and D.



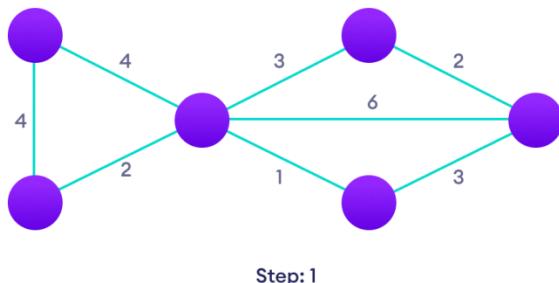
Each subpath is the shortest path

Dijkstra used this property in the opposite direction i.e we overestimate the distance of each vertex from the starting vertex. Then we visit each node and its neighbors to find the shortest subpath to those neighbors.

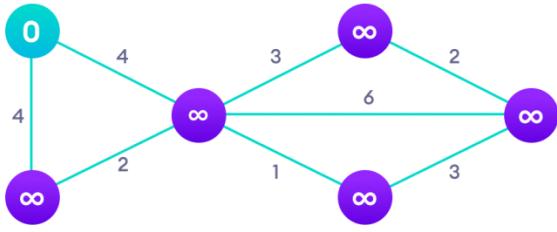
The algorithm uses a greedy approach in the sense that we find the next best solution hoping that the end result is the best solution for the whole problem.

Example of Dijkstra's algorithm

It is easier to start with an example and then think about the algorithm.

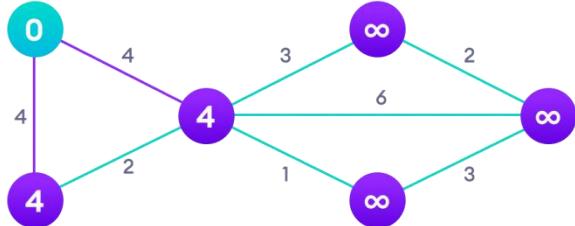


Start with a weighted graph



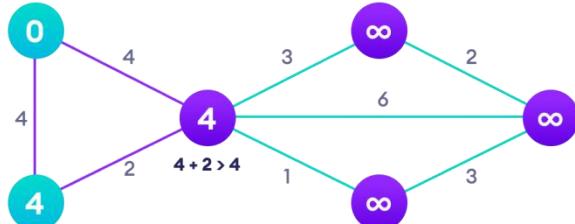
Step: 2

Choose a starting vertex and assign infinity path values to all other devices



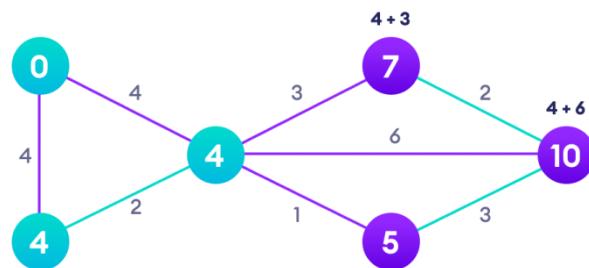
Step: 3

Go to each vertex and update its path length



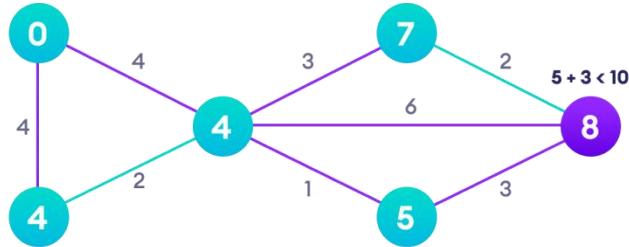
Step: 4

If the path length of the adjacent vertex is lesser than new path length, don't update it



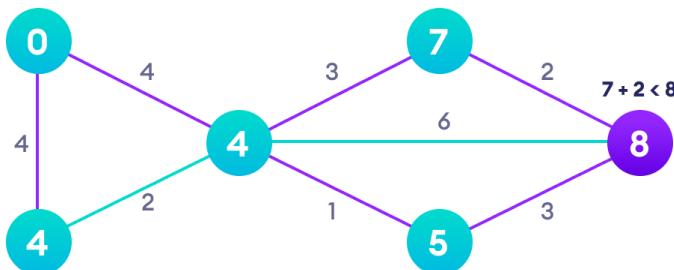
Step: 5

Avoid updating path lengths of already visited vertices



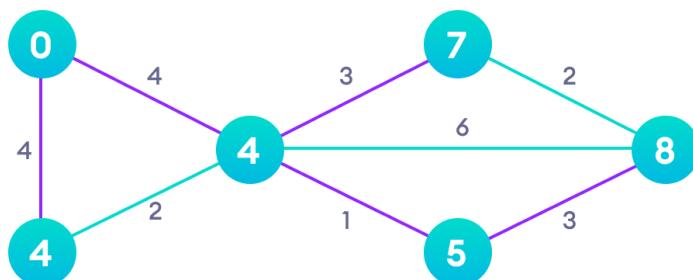
Step: 6

After each iteration, we pick the unvisited vertex with the least path length. So we choose 5 before 7



Step: 7

Notice how the rightmost vertex has its path length updated twice



Step: 8

Repeat until all the vertices have been visited

Djikstra's algorithm pseudocode

We need to maintain the path distance of every vertex. We can store that in an array of size v , where v is the number of vertices.

We also want to be able to get the shortest path, not only know the length of the shortest path. For this, we map each vertex to the vertex that last updated its path length.

Once the algorithm is over, we can backtrack from the destination vertex to the source vertex to find the path.

A minimum priority queue can be used to efficiently receive the vertex with least path distance.

```

function dijkstra(G, S)
    for each vertex V in G
        distance[V] <- infinite
        previous[V] <- NULL
        If V != S, add V to Priority Queue Q
    distance[S] <- 0

    while Q IS NOT EMPTY
        U <- Extract MIN from Q
        for each unvisited neighbour V of U
            tempDistance <- distance[U] + edge_weight(U, V)
            if tempDistance < distance[V]
                distance[V] <- tempDistance
                previous[V] <- U
    return distance[], previous[]

```

Program to implement Djikstra's algorithm

```
// Dijkstra's Algorithm
#include <stdio.h>
#define INFINITY 9999
#define MAX 10

void Dijkstra(int Graph[MAX][MAX], int n, int start);

void Dijkstra(int Graph[MAX][MAX], int n, int start) {
    int cost[MAX][MAX], distance[MAX], pred[MAX];
    int visited[MAX], count, mindistance, nextnode, i, j;

    // Creating cost matrix
    for (i = 0; i < n; i++)
        for (j = 0; j < n; j++)
            if (Graph[i][j] == 0)
                cost[i][j] = INFINITY;
            else
                cost[i][j] = Graph[i][j];

    for (i = 0; i < n; i++) {
        distance[i] = cost[start][i];
        pred[i] = start;
        visited[i] = 0;
    }

    distance[start] = 0;
    visited[start] = 1;
    count = 1;

    while (count < n - 1) {
        mindistance = INFINITY;

        for (i = 0; i < n; i++)
            if (distance[i] < mindistance && !visited[i]) {
                mindistance = distance[i];
                nextnode = i;
            }

        visited[nextnode] = 1;
        for (i = 0; i < n; i++)
            if (!visited[i])
                if (mindistance + cost[nextnode][i] < distance[i]) {
                    distance[i] = mindistance + cost[nextnode][i];
                    pred[i] = nextnode;
                }
        count++;
    }

    // Printing the distance
    for (i = 0; i < n; i++)
        if (i != start)
            printf("\nDistance from source to %d: %d", i, distance[i]);
    }

int main() {
    int Graph[MAX][MAX], i, j, n, u;
    n = 7;
```

```

Graph[0][0] = 0;
Graph[0][1] = 0;
Graph[0][2] = 1;
Graph[0][3] = 2;
Graph[0][4] = 0;
Graph[0][5] = 0;
Graph[0][6] = 0;

Graph[1][0] = 0;
Graph[1][1] = 0;
Graph[1][2] = 2;
Graph[1][3] = 0;
Graph[1][4] = 0;
Graph[1][5] = 3;
Graph[1][6] = 0;

Graph[2][0] = 1;
Graph[2][1] = 2;
Graph[2][2] = 0;
Graph[2][3] = 1;
Graph[2][4] = 3;
Graph[2][5] = 0;
Graph[2][6] = 0;

Graph[3][0] = 2;
Graph[3][1] = 0;
Graph[3][2] = 1;
Graph[3][3] = 0;
Graph[3][4] = 0;
Graph[3][5] = 0;
Graph[3][6] = 1;

Graph[4][0] = 0;
Graph[4][1] = 0;
Graph[4][2] = 3;
Graph[4][3] = 0;
Graph[4][4] = 0;
Graph[4][5] = 2;
Graph[4][6] = 0;

Graph[5][0] = 0;
Graph[5][1] = 3;
Graph[5][2] = 0;
Graph[5][3] = 0;
Graph[5][4] = 2;
Graph[5][5] = 0;
Graph[5][6] = 1;

Graph[6][0] = 0;
Graph[6][1] = 0;
Graph[6][2] = 0;
Graph[6][3] = 1;
Graph[6][4] = 0;
Graph[6][5] = 1;
Graph[6][6] = 0;

u = 0;
Dijkstra(Graph, n, u);

return 0;

```

}

Output

```
Distance from source to 1: 3
Distance from source to 2: 1
Distance from source to 3: 2
Distance from source to 4: 4
Distance from source to 5: 4
Distance from source to 6: 3
```

Watch Videos:-

[Dijkstra Algorithm - Single Source Shortest Path - Greedy Method](#)

[Dijkstra Algorithm- single source shortest path| With example | Greedy Method](#)

Kruskal's Algorithm

Kruskal's algorithm is a minimum spanning tree algorithm that takes a graph as input and finds the subset of the edges of that graph which

- o form a tree that includes every vertex
- o has the minimum sum of weights among all the trees that can be formed from the graph

How Kruskal's algorithm works

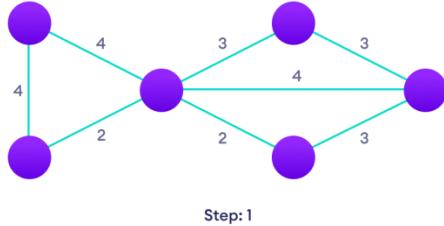
It falls under a class of algorithms called greedy algorithms that find the local optimum in the hopes of finding a global optimum.

We start from the edges with the lowest weight and keep adding edges until we reach our goal.

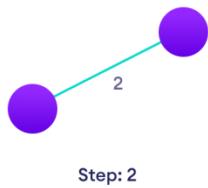
The steps for implementing Kruskal's algorithm are as follows:

- o Sort all the edges from low weight to high
- o Take the edge with the lowest weight and add it to the spanning tree. If adding the edge created a cycle, then reject this edge.
- o Keep adding edges until we reach all vertices.

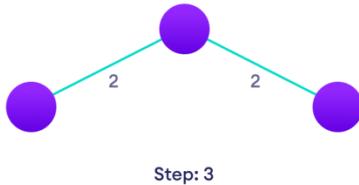
Example of Kruskal's algorithm



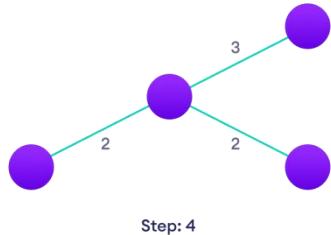
Start with a weighted graph



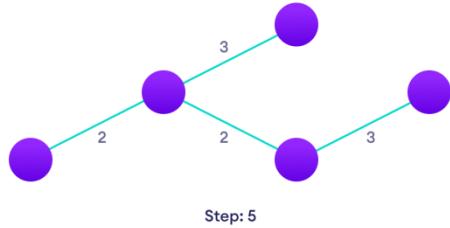
Choose the edge with the least weight, if there are more than 1, choose anyone



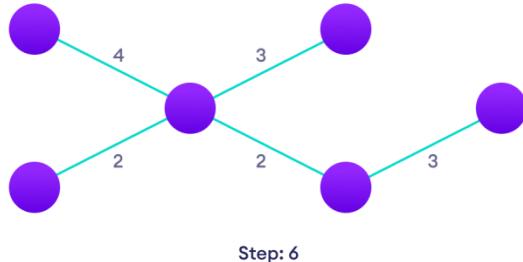
Choose the next shortest edge and add it



Choose the next shortest edge that doesn't create a cycle and add it



Choose the next shortest edge that doesn't create a cycle and add it



Repeat until you have a spanning tree

Kruskal Algorithm Pseudocode

Any minimum spanning tree algorithm revolves around checking if adding an edge creates a loop or not.

The most common way to find this out is an algorithm called Union Find. The Union-Find algorithm divides the vertices into clusters and allows us to check if two vertices belong to the same cluster or not and hence decide whether adding an edge creates a cycle.

```

KRUSKAL(G) :
A = ∅
For each vertex v ∈ G.V:
    MAKE-SET(v)
For each edge (u, v) ∈ G.E ordered by increasing order by weight(u, v):
    if FIND-SET(u) ≠ FIND-SET(v):
        A = A ∪ {(u, v)}
        UNION(u, v)
return A
    
```

Program to implement Kruskals Algorithm

```
// Kruskal's algorithm
#include <stdio.h>

#define MAX 30

typedef struct edge {
    int u, v, w;
} edge;

typedef struct edge_list {
    edge data[MAX];
    int n;
} edge_list;

edge_list elist;

int Graph[MAX][MAX], n;
edge_list spanlist;

void kruskalAlgo();
int find(int belongs[], int vertexno);
void applyUnion(int belongs[], int c1, int c2);
void sort();
void print();

// Applying Krushkal Algo
void kruskalAlgo() {
    int belongs[MAX], i, j, cnol, cno2;
    elist.n = 0;

    for (i = 1; i < n; i++) {
        for (j = 0; j < i; j++) {
            if (Graph[i][j] != 0) {
                elist.data[elist.n].u = i;
                elist.data[elist.n].v = j;
                elist.data[elist.n].w = Graph[i][j];
                elist.n++;
            }
        }
    }

    sort();

    for (i = 0; i < n; i++)
        belongs[i] = i;

    spanlist.n = 0;

    for (i = 0; i < elist.n; i++) {
        cnol = find(belongs, elist.data[i].u);
        cno2 = find(belongs, elist.data[i].v);

        if (cnol != cno2) {
            spanlist.data[spanlist.n] = elist.data[i];
            spanlist.n = spanlist.n + 1;
            applyUnion(belongs, cnol, cno2);
        }
    }
}
```

```

int find(int belongs[], int vertexno) {
    return (belongs[vertexno]);
}

void applyUnion(int belongs[], int c1, int c2) {
    int i;

    for (i = 0; i < n; i++)
        if (belongs[i] == c2)
            belongs[i] = c1;
}

// Sorting algo
void sort() {
    int i, j;
    edge temp;

    for (i = 1; i < elist.n; i++)
        for (j = 0; j < elist.n - 1; j++)
            if (elist.data[j].w > elist.data[j + 1].w) {
                temp = elist.data[j];
                elist.data[j] = elist.data[j + 1];
                elist.data[j + 1] = temp;
            }
}

// Printing the result
void print() {
    int i, cost = 0;

    for (i = 0; i < spanlist.n; i++) {
        printf("\n%d - %d : %d", spanlist.data[i].u, spanlist.data[i].v,
spanlist.data[i].w);
        cost = cost + spanlist.data[i].w;
    }

    printf("\nSpanning tree cost: %d", cost);
}

int main() {
    int i, j, total_cost;

    n = 6;

    Graph[0][0] = 0;
    Graph[0][1] = 4;
    Graph[0][2] = 4;
    Graph[0][3] = 0;
    Graph[0][4] = 0;
    Graph[0][5] = 0;
    Graph[0][6] = 0;

    Graph[1][0] = 4;
    Graph[1][1] = 0;
    Graph[1][2] = 2;
    Graph[1][3] = 0;
    Graph[1][4] = 0;
    Graph[1][5] = 0;
    Graph[1][6] = 0;
}

```

```

Graph[2][0] = 4;
Graph[2][1] = 2;
Graph[2][2] = 0;
Graph[2][3] = 3;
Graph[2][4] = 4;
Graph[2][5] = 0;
Graph[2][6] = 0;

Graph[3][0] = 0;
Graph[3][1] = 0;
Graph[3][2] = 3;
Graph[3][3] = 0;
Graph[3][4] = 3;
Graph[3][5] = 0;
Graph[3][6] = 0;

Graph[4][0] = 0;
Graph[4][1] = 0;
Graph[4][2] = 4;
Graph[4][3] = 3;
Graph[4][4] = 0;
Graph[4][5] = 0;
Graph[4][6] = 0;

Graph[5][0] = 0;
Graph[5][1] = 0;
Graph[5][2] = 2;
Graph[5][3] = 0;
Graph[5][4] = 3;
Graph[5][5] = 0;
Graph[5][6] = 0;

kruskalAlgo();
print();
}

```

Output

2 - 1 : 2
 5 - 2 : 2
 3 - 2 : 3
 4 - 3 : 3
 1 - 0 : 4

Spanning tree cost: 14

Watch Videos:-

[Kruskals Algorithms - Greedy Method](#)

[Kruskals Algorithm for Minimum Spanning Tree- Greedy method](#)

Prim's Algorithm

Prim's algorithm is a minimum spanning tree algorithm that takes a graph as input and finds the subset of the edges of that graph which

- o form a tree that includes every vertex
- o has the minimum sum of weights among all the trees that can be formed from the graph

How Prim's algorithm works

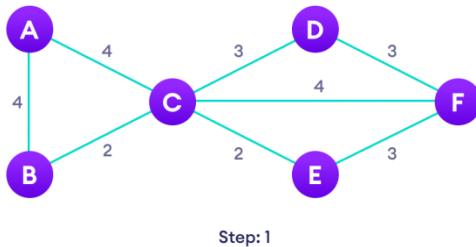
It falls under a class of algorithms called greedy algorithms that find the local optimum in the hopes of finding a global optimum.

We start from one vertex and keep adding edges with the lowest weight until we reach our goal.

The steps for implementing Prim's algorithm are as follows:

- o Initialize the minimum spanning tree with a vertex chosen at random.
- o Find all the edges that connect the tree to new vertices, find the minimum and add it to the tree
- o Keep repeating step 2 until we get a minimum spanning tree

Example of Prim's algorithm

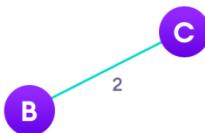


Start with a weighted graph



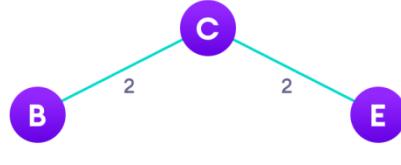
Step: 2

Choose a vertex



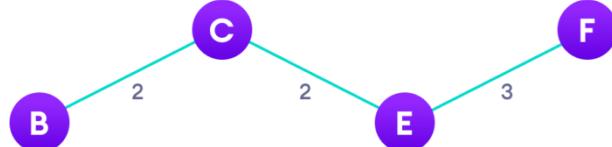
Step: 3

Choose the shortest edge from this vertex and add it



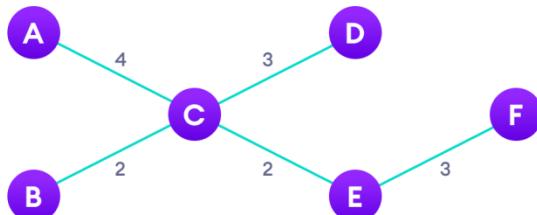
Step: 4

Choose the nearest vertex not yet in the solution



Step: 5

Choose the nearest edge not yet in the solution, if there are multiple choices, choose one at random



Step: 6

Repeat until you have a spanning tree

Prim's Algorithm pseudocode

The pseudocode for prim's algorithm shows how we create two sets of vertices U and $V - U$. U contains the list of vertices that have been visited and $V - U$ the list of vertices that haven't. One by one, we move vertices from set $V - U$ to set U by connecting the least weight edge.

```

T = ∅;
U = { 1 };
while (U ≠ V)
    let (u, v) be the lowest cost edge such that u ∈ U and v ∈ V - U;
    T = T ∪ {(u, v)}
    U = U ∪ {v}
  
```

Program to implement Prims Algorithm

```

// Prim's Algorithm
#include<stdio.h>
#include<stdbool.h>
#include<string.h>

#define INF 9999999

// number of vertices in graph
#define V 5

// create a 2d array of size 5x5
//for adjacency matrix to represent graph
int G[V][V] = {
    {0, 9, 75, 0, 0},
    {9, 0, 95, 19, 42},
    {75, 95, 0, 51, 66},
    {0, 19, 51, 0, 31},
    {0, 42, 66, 31, 0}};

int main() {
    int no_edge; // number of edge

    // create a array to track selected vertex
    // selected will become true otherwise false
    int selected[V];

    // set selected false initially
    memset(selected, false, sizeof(selected));

    // set number of edge to 0
    no_edge = 0;

    // the number of egde in minimum spanning tree will be
    // always less than (V - 1), where V is number of vertices in
    // graph

    // choose 0th vertex and make it true
    selected[0] = true;

    int x; // row number
    int y; // col number

    // print for edge and weight
    printf("Edge : Weight\n");

    while (no_edge < V - 1) {
        //For every vertex in the set S, find the all adjacent vertices
        // , calculate the distance from the vertex selected at step 1.
        // if the vertex is already in the set S, discard it otherwise
        //choose another vertex nearest to selected vertex at step 1.

        int min = INF;
        x = 0;
        y = 0;

        for (int i = 0; i < V; i++) {
            if (selected[i]) {
                for (int j = 0; j < V; j++) {

```

```

    if (!selected[j] && G[i][j]) { // not in selected and there is an edge
        if (min > G[i][j]) {
            min = G[i][j];
            x = i;
            y = j;
        }
    }
}
printf("%d - %d : %d\n", x, y, G[x][y]);
selected[y] = true;
no_edge++;
}

return 0;
}

```

Output

Edge : Weight
0 - 1 : 9
1 - 3 : 19
3 - 4 : 31
3 - 2 : 51

Watch Videos :-

[Prims Algorithms - Greedy Method](#)
[Prim's Algorithm for Minimum Spanning Tree](#)

Huffman Coding

Huffman Coding is a technique of compressing data to reduce its size without losing any of the details. It was first developed by David Huffman.

Huffman Coding is generally useful to compress the data in which there are frequently occurring characters.

How Huffman Coding works?

Suppose the string below is to be sent over a network.

B	C	A	A	D	D	D	C	C	A	C	A	C	A	C
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

Each character occupies 8 bits. There are a total of 15 characters in the above string. Thus, a total of $8 * 15 = 120$ bits are required to send this string.

Using the Huffman Coding technique, we can compress the string to a smaller size.

Huffman coding first creates a tree using the frequencies of the character and then generates code for each character.

Once the data is encoded, it has to be decoded. Decoding is done using the same tree.

Huffman Coding prevents any ambiguity in the decoding process using the concept of prefix **code** ie. a code associated with a character should not be present in the prefix of any other code. The tree created above helps in maintaining the property.

Huffman coding is done with the help of the following steps.

1. Calculate the frequency of each character in the string.

1	6	5	3
B	C	A	D

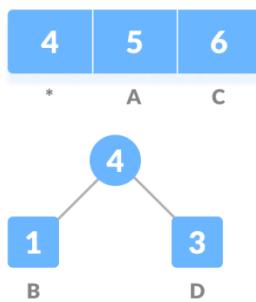
Frequency of string

2. Sort the characters in increasing order of the frequency. These are stored in a priority queue Q.

1	3	5	6
B	D	A	C

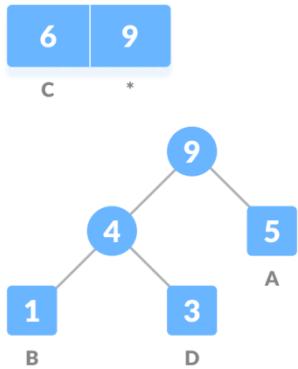
Characters sorted according to the frequency

3. Make each unique character as a leaf node.
4. Create an empty node \underline{z} . Assign the minimum frequency to the left child of \underline{z} and assign the second minimum frequency to the right child of \underline{z} . Set the value of the \underline{z} as the sum of the above two minimum frequencies.

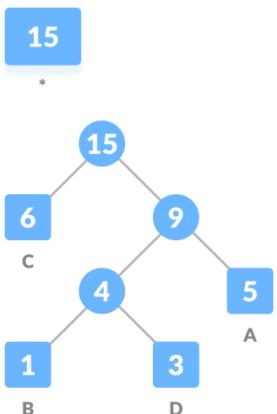


Getting the sum of the least numbers

5. Remove these two minimum frequencies from \underline{Q} and add the sum into the list of frequencies (* denote the internal nodes in the figure above).
6. Insert node \underline{z} into the tree.
7. Repeat steps 3 to 5 for all the characters

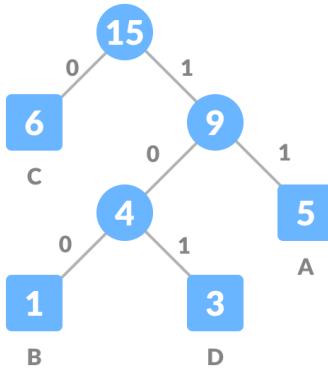


Repeat steps 3 to 5 for all the characters.



Repeat steps 3 to 5 for all the characters.

8. For each non-leaf node, assign 0 to the left edge and 1 to the right edge.



Assign 0 to the left edge and 1 to the right edge

For sending the above string over a network, we have to send the tree as well as the above compressed-code. The total size is given by the table below.

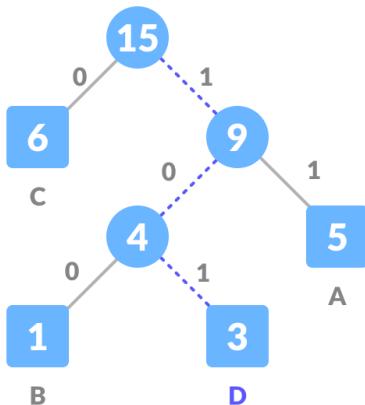
Character	Frequency	Code	Size
A	5	11	$5 \times 2 = 10$
B	1	100	$1 \times 3 = 3$
C	6	0	$6 \times 1 = 6$
D	3	101	$3 \times 3 = 9$
$4 * 8 = 32$ bits	15 bits		28 bits

Without encoding, the total size of the string was 120 bits. After encoding the size is reduced to $32 + 15 + 28 = 75$.

Decoding the code

For decoding the code, we can take the code and traverse through the tree to find the character.

Let 101 is to be decoded, we can traverse from the root as in the figure below.



Huffman Coding Algorithm

```
create a priority queue Q consisting of each unique character.  
sort them in ascending order of their frequencies.  
for all the unique characters:  
    create a newNode  
    extract minimum value from Q and assign it to leftChild of newNode  
    extract minimum value from Q and assign it to rightChild of newNode  
    calculate the sum of these two minimum values and assign it to the value of  
newNode  
    insert this newNode into the tree  
return rootNode
```

Program to implement Huffman Coding Algorithm

```

// Huffman Coding
#include <stdio.h>
#include <stdlib.h>

#define MAX_TREE_HT 50

struct MinHNode {
    char item;
    unsigned freq;
    struct MinHNode *left, *right;
};

struct MinHeap {
    unsigned size;
    unsigned capacity;
    struct MinHNode **array;
};

// Print the array
void printArray(int arr[], int n) {
    int i;
    for (i = 0; i < n; ++i)
        printf("%d", arr[i]);

    printf("\n");
}

// Create nodes
struct MinHNode *newNode(char item, unsigned freq) {
    struct MinHNode *temp = (struct MinHNode *)malloc(sizeof(struct MinHNode));

    temp->left = temp->right = NULL;
    temp->item = item;
    temp->freq = freq;

    return temp;
}

// Create min heap
struct MinHeap *createMinH(unsigned capacity) {
    struct MinHeap *minHeap = (struct MinHeap *)malloc(sizeof(struct MinHeap));

    minHeap->size = 0;

    minHeap->capacity = capacity;

    minHeap->array = (struct MinHNode **)malloc(minHeap->capacity * sizeof(struct MinHNode *));
    return minHeap;
}

// Function to swap
void swapMinHNode(struct MinHNode **a, struct MinHNode **b) {
    struct MinHNode *t = *a;
    *a = *b;
    *b = t;
}

```

```

// Heapify
void minHeapify(struct MinHeap *minHeap, int idx) {
    int smallest = idx;
    int left = 2 * idx + 1;
    int right = 2 * idx + 2;

    if (left < minHeap->size && minHeap->array[left]->freq < minHeap->array[smallest]->freq)
        smallest = left;

    if (right < minHeap->size && minHeap->array[right]->freq < minHeap->array[smallest]->freq)
        smallest = right;

    if (smallest != idx) {
        swapMinHNode(&minHeap->array[smallest], &minHeap->array[idx]);
        minHeapify(minHeap, smallest);
    }
}

// Check if size is 1
int checkSizeOne(struct MinHeap *minHeap) {
    return (minHeap->size == 1);
}

// Extract min
struct MinHNode *extractMin(struct MinHeap *minHeap) {
    struct MinHNode *temp = minHeap->array[0];
    minHeap->array[0] = minHeap->array[minHeap->size - 1];

    --minHeap->size;
    minHeapify(minHeap, 0);

    return temp;
}

// Insertion function
void insertMinHeap(struct MinHeap *minHeap, struct MinHNode *minHeapNode) {
    ++minHeap->size;
    int i = minHeap->size - 1;

    while (i && minHeapNode->freq < minHeap->array[(i - 1) / 2]->freq) {
        minHeap->array[i] = minHeap->array[(i - 1) / 2];
        i = (i - 1) / 2;
    }
    minHeap->array[i] = minHeapNode;
}

void buildMinHeap(struct MinHeap *minHeap) {
    int n = minHeap->size - 1;
    int i;

    for (i = (n - 1) / 2; i >= 0; --i)
        minHeapify(minHeap, i);
}

int isLeaf(struct MinHNode *root) {
    return !(root->left) && !(root->right);
}

```

```

struct MinHeap *createAndBuildMinHeap(char item[], int freq[], int size) {
    struct MinHeap *minHeap = createMinH(size);

    for (int i = 0; i < size; ++i)
        minHeap->array[i] = newNode(item[i], freq[i]);

    minHeap->size = size;
    buildMinHeap(minHeap);

    return minHeap;
}

struct MinHNode *buildHuffmanTree(char item[], int freq[], int size) {
    struct MinHNode *left, *right, *top;
    struct MinHeap *minHeap = createAndBuildMinHeap(item, freq, size);

    while (!checkSizeOne(minHeap)) {
        left = extractMin(minHeap);
        right = extractMin(minHeap);

        top = newNode('$', left->freq + right->freq);

        top->left = left;
        top->right = right;

        insertMinHeap(minHeap, top);
    }
    return extractMin(minHeap);
}

void printHCodes(struct MinHNode *root, int arr[], int top) {
    if (root->left) {
        arr[top] = 0;
        printHCodes(root->left, arr, top + 1);
    }
    if (root->right) {
        arr[top] = 1;
        printHCodes(root->right, arr, top + 1);
    }
    if (isLeaf(root)) {
        printf(" %c | ", root->item);
        printArray(arr, top);
    }
}

// Wrapper function
void HuffmanCodes(char item[], int freq[], int size) {
    struct MinHNode *root = buildHuffmanTree(item, freq, size);

    int arr[MAX_TREE_HT], top = 0;

    printHCodes(root, arr, top);
}

int main() {
    char arr[] = {'A', 'B', 'C', 'D'};
    int freq[] = {5, 1, 6, 3};

    int size = sizeof(arr) / sizeof(arr[0]);
}

```

```
printf(" Char | Huffman code ");
printf("\n-----\n");
HuffmanCodes(arr, freq, size);
}
```

Output

Char		Huffman code
C		0
B		100
D		101
A		11

Watch Videos :-

[Huffman Coding - Greedy Method](#)

[Huffman coding example -Greedy Method](#)

Knapsack Problem

The knapsack problem is a problem in combinatorial optimization: Given a set of items, each with a weight and a value, determine the number of each item to include in a collection so that the total weight is less than or equal to a given limit and the total value is as large as possible. It derives its name from the problem faced by someone who is constrained by a fixed-size knapsack and must fill it with the most valuable items. The problem often arises in resource allocation where the decision-makers have to choose from a set of non-divisible projects or tasks under a fixed budget or time constraint, respectively.

Here knapsack is like a container or a bag. Suppose we have given some items which have some weights or profits. We have to put some items in the knapsack in such a way total value produces a maximum profit.

For example, the weight of the container is 20 kg. We have to select the items in such a way that the sum of the weight of items should be either smaller than or equal to the weight of the container, and the profit should be maximum.

There are two types of knapsack problems:

- 0/1 knapsack problem
- Fractional knapsack problem

0/1 knapsack problem

The 0/1 knapsack problem means that the items are either completely or no items are filled in a knapsack.

For example, we have two items having weights 2kg and 3kg, respectively. If we pick the 2kg item then we cannot pick 1kg item from the 2kg item (item is not divisible); we have to pick the 2kg item completely.

This is a 0/1 knapsack problem in which either we pick the item completely or we will pick that item.

The 0/1 knapsack problem is solved by the dynamic programming.

F knapsack problem?

The fractional knapsack problem means that we can divide the item.

For example, we have an item of 3 kg then we can pick the item of 2 kg and leave the item of 1 kg.

The fractional knapsack problem is solved by the Greedy approach.

Watch Videos:-

[Knapsack Problem - Greedy Method](#)

[Fractional Knapsack Problem using Greedy Method](#)

Program to implement Knapsack Problem

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

void knapsack(int n, float weight[], float profit[], float capacity)
{
    float x[20], tp = 0;
    int i, j, u;
    u = capacity;

    for (i = 0; i < n; i++)
        x[i] = 0.0;

    for (i = 0; i < n; i++) {
        if (weight[i] > u)
            break;
        else {
            x[i] = 1.0;
            tp = tp + profit[i];
            u = u - weight[i];
        }
    }

    if (i < n)
        x[i] = u / weight[i];

    tp = tp + (x[i] * profit[i]);

    printf("\nThe result vector is:- ");
    for (i = 0; i < n; i++)
        printf("%f\t", x[i]);

    printf("\nMaximum profit is:- %f", tp);
}

int main()
{
    float weight[20], profit[20], capacity;
    int num, i, j;
    float ratio[20], temp;

    printf("\nEnter the no. of objects:- ");
    scanf("%d", &num);

    printf("\nEnter the wts and profits of each object:- ");
    for (i = 0; i < num; i++)
    {
        scanf("%f %f", &weight[i], &profit[i]);
    }

    printf("\nEnter the capacity of knapsack:- ");
    scanf("%f", &capacity);

    for (i = 0; i < num; i++)
    {
        ratio[i] = profit[i] / weight[i];
    }
}
```

```

for (i = 0; i < num; i++)
{
    for (j = i + 1; j < num; j++)
    {
        if (ratio[i] < ratio[j])
        {
            temp = ratio[j];
            ratio[j] = ratio[i];
            ratio[i] = temp;

            temp = weight[j];
            weight[j] = weight[i];
            weight[i] = temp;

            temp = profit[j];
            profit[j] = profit[i];
            profit[i] = temp;
        }
    }
}

knapsack(num, weight, profit, capacity);
return (0);
}

```

Output

2 10
 3 5
 5 15
 7 7
 1 6
 4 18
 1 3

Enter the capacityacity of knapsack:- 15
 The result vector is:- 1.000000 1.000000 1.000000 1.000000 1.000000 0.666667
 0.000000

Maximum profit is:- 55.333332

Job Sequencing with Deadlines

Job scheduling is a scheduling problem for numerous jobs with given deadlines to maximize profit. Here, the main objective is to find the sequence of jobs that maximize completion within the deadline. The Job Scheduling Algorithm is a greedy algorithm-based popular problem that has wide implementations in real-world scenarios. Let's get right to the problem then.

Greedy approach for job sequencing problem:

Greedily choose the jobs with maximum profit first, by sorting the jobs in decreasing order of their profit. This would help to maximize the total profit as choosing the job with maximum profit for every time slot will eventually maximize the total profit

Follow the given steps to solve the problem:

- Sort all jobs in decreasing order of profit.
- Iterate on jobs in decreasing order of profit. For each job , do the following :
 - Find a time slot i, such that slot is empty and $i < \text{deadline}$ and i is greatest. Put the job in this slot and mark this slot filled.
 - If no such i exists, then ignore the job.

Example

$n = 7$

maximum time slot = 4

Jobs	J1	J2	J3	J4	J5	J6	J7
Profit	35	30	25	20	15	12	5
Deadline	3	4	4	2	3	1	2

0 → 1 → 2 → 3 → 4
 J4 J4 J1 J2

J5,J6 and J7 are not need to check, because there is no slot available

Program to implement Job Sequencing

```

// Program to implement Job Sequencing Problem
#include <stdbool.h>
#include <stdio.h>
#include <stdlib.h>

// A structure to represent a job
typedef struct Job {
    char id; // Job Id
    int dead; // Deadline of job
    int profit; // Profit if job is over before or on
                 // deadline
} Job;

// This function is used for sorting all jobs according to
// profit
int compare(const void* a, const void* b)
{
    Job* temp1 = (Job*)a;
    Job* temp2 = (Job*)b;
    return (temp2->profit - temp1->profit);
}

// Find minimum between two numbers.
int min(int num1, int num2)
{
    return (num1 > num2) ? num2 : num1;
}

// Returns maximum profit from jobs
void printJobScheduling(Job arr[], int n)
{
    // Sort all jobs according to decreasing order of profit
    qsort(arr, n, sizeof(Job), compare);

    int result[n]; // To store result (Sequence of jobs)
    bool slot[n]; // To keep track of free time slots

    // Initialize all slots to be free
    for (int i = 0; i < n; i++)
        slot[i] = false;

    // Iterate through all given jobs
    for (int i = 0; i < n; i++) {

        // Find a free slot for this job (Note that we start
        // from the last possible slot)
        for (int j = min(n, arr[i].dead) - 1; j >= 0; j--) {

            // Free slot found
            if (slot[j] == false) {
                result[j] = i; // Add this job to result
                slot[j] = true; // Make this slot occupied
                break;
            }
        }
    }
}

```

```

// Print the result
for (int i = 0; i < n; i++)
    if (slot[i])
        printf("%c ", arr[result[i]].id);
}

// Driver's code
int main()
{
    Job arr[] = {
        { 'a', 2, 100 },
        { 'b', 1, 19 },
        { 'c', 2, 27 },
        { 'd', 1, 25 },
        { 'e', 3, 15 }
    };
    int n = sizeof(arr) / sizeof(arr[0]);
    printf(
        "Following is maximum profit sequence of jobs \n");
    // Function call
    printJobScheduling(arr, n);
    return 0;
}

```

Output

Following is maximum profit sequence of jobs
c a e

Watch Videos:-

[Job Sequencing with Deadlines - Greedy Method](#)

Optimal Merge Pattern

Given n number of sorted files, the task is to find the minimum computations done to reach the Optimal Merge Pattern. When two or more sorted files are to be merged altogether to form a single file, the minimum computations are done to reach this file are known as Optimal Merge Pattern.

If more than 2 files need to be merged then it can be done in pairs.

If we have two files of sizes m and n , the total computation time will be $m+n$. Here, we use the greedy strategy by merging the two smallest size files among all the files present.

Merge a set of sorted files of different length into a single sorted file. We need to find an optimal solution, where the resultant file will be generated in minimum time.

If the number of sorted files are given, there are many ways to merge them into a single sorted file. This merge can be performed pair wise. Hence, this type of merging is called as 2-way merge patterns.

As, different pairings require different amounts of time, in this strategy we want to determine an optimal way of merging many files together. At each step, two shortest sequences are merged.

To merge a p -record file and a q -record file requires possibly $p + q$ record moves, the obvious choice being, merge the two smallest files together at each step.

Two-way merge patterns can be represented by binary merge trees. Let us consider a set of n sorted files $\{f_1, f_2, f_3, \dots, f_n\}$. Initially, each element of this is considered as a single node binary tree. To find this optimal solution, the following algorithm is used.

Example

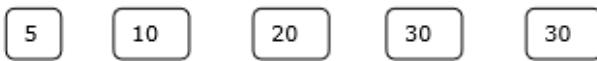
Let us consider the given files, f_1, f_2, f_3, f_4 and f_5 with 20, 30, 10, 5 and 30 number of elements respectively.

Sorting the numbers according to their size in an ascending order, we get the following sequence –

f_4, f_3, f_1, f_2, f_5

Obviously, this is better than the previous one.

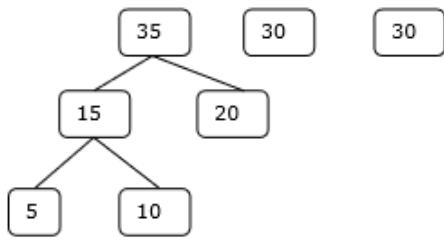
Initial Set



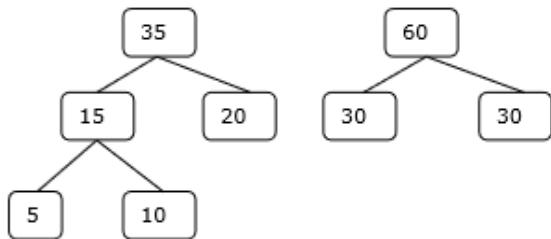
Step-1



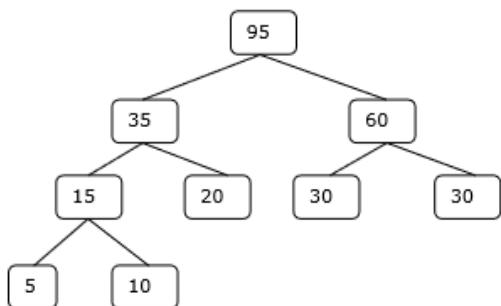
Step-2



Step-3



Step-4



Hence, the solution takes $15 + 35 + 60 + 95 = \mathbf{205}$ number of comparisons

Watch Videos:-

[Optimal Merge Pattern - Greedy Method](#)

9. Dynamic Programming

Dynamic Programming

Dynamic Programming is a technique in computer programming that helps to efficiently solve a class of problems that have overlapping subproblems and optimal substructure property.

If any problem can be divided into subproblems, which in turn are divided into smaller subproblems, and if there are overlapping among these subproblems, then the solutions to these subproblems can be saved for future reference. In this way, efficiency of the CPU can be enhanced. This method of solving a solution is referred to as dynamic programming.

Such problems involve repeatedly calculating the value of the same subproblems to find the optimum solution.

Dynamic Programming Example

Let's find the fibonacci sequence upto 5th term. A fibonacci series is the sequence of numbers in which each number is the sum of the two preceding ones. For example, 0,1,1, 2, 3. Here, each number is the sum of the two preceding numbers.

Algorithm

Let n be the number of terms.

1. If $n \leq 1$, return 1.
2. Else, return the sum of two preceding numbers.

We are calculating the fibonacci sequence up to the 5th term.

1. The first term is 0.
2. The second term is 1.
3. The third term is sum of 0 (from step 1) and 1(from step 2), which is 1.
4. The fourth term is the sum of the third term (from step 3) and second term (from step 2) i.e. $1 + 1 = 2$.
5. The fifth term is the sum of the fourth term (from step 4) and third term (from step 3) i.e. $2 + 1 = 3$.
6. Hence, we have the sequence 0,1,1, 2, 3. Here, we have used the results of the previous steps as shown below. This is called a dynamic programming approach.

$$\begin{aligned} F(0) &= 0 \\ F(1) &= 1 \\ F(2) &= F(1) + F(0) \\ F(3) &= F(2) + F(1) \\ F(4) &= F(3) + F(2) \end{aligned}$$

How Dynamic Programming Works

Dynamic programming works by storing the result of subproblems so that when their solutions are required, they are at hand and we do not need to recalculate them.

This technique of storing the value of subproblems is called memoization. By saving the values in the array, we save time for computations of sub-problems we have already come across.

```

var m = map(0 → 0, 1 → 1)
function fib(n)
    if key n is not in map m
        m[n] = fib(n - 1) + fib(n - 2)
    return m[n]

```

Dynamic programming by memoization is a top-down approach to dynamic programming. By reversing the direction in which the algorithm works i.e. by starting from the base case and working towards the solution, we can also implement dynamic programming in a bottom-up manner.

```

function fib(n)
    if n = 0
        return 0
    else
        var prevFib = 0, currFib = 1
        repeat n - 1 times
            var newFib = prevFib + currFib
            prevFib = currFib
            currFib = newFib
        return currentFib

```

Recursion vs Dynamic Programming

Dynamic programming is mostly applied to recursive algorithms. This is not a coincidence, most optimization problems require recursion and dynamic programming is used for optimization.

But not all problems that use recursion can use Dynamic Programming. Unless there is a presence of overlapping subproblems like in the fibonacci sequence problem, a recursion can only reach the solution using a divide and conquer approach.

That is the reason why a recursive algorithm like Merge Sort cannot use Dynamic Programming, because the subproblems are not overlapping in any way.

Greedy Algorithms vs Dynamic Programming

Greedy Algorithms are similar to dynamic programming in the sense that they are both tools for optimization.

However, greedy algorithms look for locally optimum solutions or in other words, a greedy choice, in the hopes of finding a global optimum. Hence greedy algorithms can make a guess that looks optimum at the time but becomes costly down the line and do not guarantee a globally optimum.

Dynamic programming, on the other hand, finds the optimal solution to subproblems and then makes an informed choice to combine the results of those subproblems to find the most optimum solution.

Different Types of Dynamic Programming Algorithms

- Longest Common Subsequence
- Floyd-Warshall Algorithm

Watch Videos:-

[Principle of Optimality - Dynamic Programming introduction](#)

Floyd-Warshall Algorithm

Floyd-Warshall Algorithm is an algorithm for finding the shortest path between all the pairs of vertices in a weighted graph. This algorithm works for both the directed and undirected weighted graphs. But, it does not work for the graphs with negative cycles (where the sum of the edges in a cycle is negative).

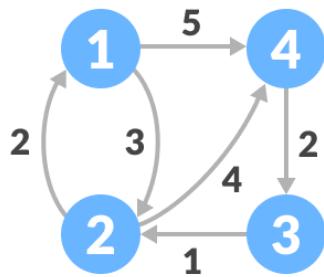
A weighted graph is a graph in which each edge has a numerical value associated with it.

Floyd-Warshall algorithm is also called as Floyd's algorithm, Roy-Floyd algorithm, Roy-Warshall algorithm, or WFI algorithm.

This algorithm follows the dynamic programming approach to find the shortest paths.

How Floyd-Warshall Algorithm Works?

Let the given graph be:



Follow the steps below to find the shortest path between all the pairs of vertices.

1. Create a matrix A^0 of dimension $n \times n$ where n is the number of vertices. The row and the column are indexed as i and j respectively. i and j are the vertices of the graph.

Each cell $A[i][j]$ is filled with the distance from the i^{th} vertex to the j^{th} vertex. If there is no path from i^{th} vertex to j^{th} vertex, the cell is left as infinity.

$$A^0 = \begin{bmatrix} 1 & 2 & 3 & 4 \\ 0 & 3 & \infty & 5 \\ 2 & 0 & \infty & 4 \\ \infty & 1 & 0 & \infty \\ \infty & \infty & 2 & 0 \end{bmatrix}$$

Fill each cell with the distance between i^{th} and j^{th} vertex

2. Now, create a matrix A^1 using matrix A^0 . The elements in the first column and the first row are left as they are. The remaining cells are filled in the following way.

Let k be the intermediate vertex in the shortest path from source to destination. In this step, k is the first vertex. $A[i][j]$ is filled with $(A[i][k] + A[k][j])$ if $(A[i][j] > A[i][k] + A[k][j])$.

That is, if the direct distance from the source to the destination is greater than the path through the vertex k , then the cell is filled with $A[i][k] + A[k][j]$.

In this step, k is vertex 1. We calculate the distance from source vertex to destination vertex through this vertex k.

$$A^1 = \begin{bmatrix} 1 & 2 & 3 & 4 \\ 1 & 0 & 3 & \infty & 5 \\ 2 & 2 & 0 & & \\ 3 & \infty & 0 & & \\ 4 & \infty & & 0 & \end{bmatrix} \rightarrow \begin{bmatrix} 1 & 2 & 3 & 4 \\ 1 & 0 & 3 & \infty & 5 \\ 2 & 2 & 0 & 9 & 4 \\ 3 & \infty & 1 & 0 & 8 \\ 4 & \infty & \infty & 2 & 0 \end{bmatrix}$$

For example: For $A^1[2, 4]$, the direct distance from vertex 2 to 4 is 4 and the sum of the distance from vertex 2 to 4 through vertex (ie. from vertex 2 to 1 and from vertex 1 to 4) is 7. Since $4 < 7$, $A^0[2, 4]$ is filled with 4.

3. Similarly, A^2 is created using A^1 . The elements in the second column and the second row are left as they are.

In this step, k is the second vertex (i.e. vertex 2). The remaining steps are the same as in **step 2**.

$$A^2 = \begin{bmatrix} 1 & 2 & 3 & 4 \\ 1 & 0 & 3 & & \\ 2 & 2 & 0 & 9 & 4 \\ 3 & 1 & 0 & & \\ 4 & \infty & 0 & & \end{bmatrix} \rightarrow \begin{bmatrix} 1 & 2 & 3 & 4 \\ 1 & 0 & 3 & 9 & 5 \\ 2 & 2 & 0 & 9 & 4 \\ 3 & 3 & 1 & 0 & 5 \\ 4 & \infty & \infty & 2 & 0 \end{bmatrix}$$

Calculate the distance from the source vertex to destination vertex through this vertex 2

4. Similarly, A^3 and A^4 is also created.

$$A^3 = \begin{bmatrix} 1 & 2 & 3 & 4 \\ 1 & 0 & & \infty \\ 2 & 0 & 9 & & \\ 3 & \infty & 1 & 0 & 8 \\ 4 & & 2 & 0 & \end{bmatrix} \rightarrow \begin{bmatrix} 1 & 2 & 3 & 4 \\ 1 & 0 & 3 & 9 & 5 \\ 2 & 2 & 0 & 9 & 4 \\ 3 & 3 & 1 & 0 & 5 \\ 4 & 5 & 3 & 2 & 0 \end{bmatrix}$$

Calculate the distance from the source vertex to destination vertex through this vertex 3

$$A^4 = \begin{bmatrix} 1 & 2 & 3 & 4 \\ 1 & 0 & & 5 \\ 2 & 0 & 4 & & \\ 3 & 0 & 5 & & \\ 4 & 5 & 3 & 2 & 0 \end{bmatrix} \rightarrow \begin{bmatrix} 1 & 2 & 3 & 4 \\ 1 & 0 & 3 & 7 & 5 \\ 2 & 2 & 0 & 6 & 4 \\ 3 & 3 & 1 & 0 & 5 \\ 4 & 5 & 3 & 2 & 0 \end{bmatrix}$$

Calculate the distance from the source vertex to destination vertex through this vertex 4

5. A^4 gives the shortest path between each pair of vertices.

Floyd-Warshall Algorithm

```
n = no of vertices
A = matrix of dimension n*n
for k = 1 to n
    for i = 1 to n
        for j = 1 to n
            Ak[i, j] = min (Ak-1[i, j], Ak-1[i, k] + Ak-1[k, j])
return A
```

Program to implement Floyd-Warshall Algorithm

```

#include <stdio.h>
// defining the number of vertices
#define nV 4
#define INF 999
void printMatrix(int matrix[][][nV]);

// Implementing floyd warshall algorithm
void floydWarshall(int graph[][][nV]) {
    int matrix[nV][nV], i, j, k;

    for (i = 0; i < nV; i++)
        for (j = 0; j < nV; j++)
            matrix[i][j] = graph[i][j];

    // Adding vertices individually
    for (k = 0; k < nV; k++) {
        for (i = 0; i < nV; i++) {
            for (j = 0; j < nV; j++) {
                if (matrix[i][k] + matrix[k][j] < matrix[i][j])
                    matrix[i][j] = matrix[i][k] + matrix[k][j];
            }
        }
    }
    printMatrix(matrix);
}

void printMatrix(int matrix[][][nV]) {
    for (int i = 0; i < nV; i++) {
        for (int j = 0; j < nV; j++) {
            if (matrix[i][j] == INF)
                printf("%4s", "INF");
            else
                printf("%4d", matrix[i][j]);
        }
        printf("\n");
    }
}

int main() {
    int graph[nV][nV] = {{0, 3, INF, 5},
                          {2, 0, INF, 4},
                          {INF, 1, 0, INF},
                          {INF, INF, 2, 0}};
    floydWarshall(graph);
}

```

Output

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

Watch Videos:-

[All Pairs Shortest Path \(Floyd-Warshall\) - Dynamic Programming](#)
[Floyd Warshall Algorithm All Pair Shortest Path algorithm](#)

Longest Common Sequence

The longest common subsequence (LCS) is defined as the longest subsequence that is common to all the given sequences, provided that the elements of the subsequence are not required to occupy consecutive positions within the original sequences.

If S_1 and S_2 are the two given sequences then, Z is the common subsequence of S_1 and S_2 if Z is a subsequence of both S_1 and S_2 . Furthermore, Z must be a strictly increasing sequence of the indices of both S_1 and S_2 .

In a strictly increasing sequence, the indices of the elements chosen from the original sequences must be in ascending order in Z .

If

$S_1 = \{B, C, D, A, A, C, D\}$

Then, $\{A, D, B\}$ cannot be a subsequence of S_1 as the order of the elements is not the same (ie. not strictly increasing sequence).

Let us understand LCS with an example.

If

$S_1 = \{B, C, D, A, A, C, D\}$

$S_2 = \{A, C, D, B, A, C\}$

Then, common subsequences are $\{B, C\}$, $\{C, D, A, C\}$, $\{D, A, C\}$, $\{A, A, C\}$, $\{A, C\}$, $\{C, D\}$, ...

Among these subsequences, $\{C, D, A, C\}$ is the longest common subsequence. We are going to find this longest common subsequence using dynamic programming.

Using Dynamic Programming to find the LCS

Let us take two sequences:



The first sequence



Second Sequence

The following steps are followed for finding the longest common subsequence.

1. Create a table of dimension $n+1*m+1$ where n and m are the lengths of X and Y respectively. The first row and the first column are filled with zeros.

	C	B	D	A
C	0	0	0	0
B	0			
A	0			
D	0			
B	0			

2. Fill each cell of the table using the following logic.
3. If the character corresponding to the current row and current column are matching, then fill the current cell by adding one to the diagonal element. Point an arrow to the diagonal cell.
4. Else take the maximum value from the previous column and previous row element for filling the current cell. Point an arrow to the cell with maximum value. If they are equal, point to any of them.

	C	B	D	A
C	0	0	0	0
B	0			
A	0			
D	0			
B	0			

5. Step 2 is repeated until the table is filled.

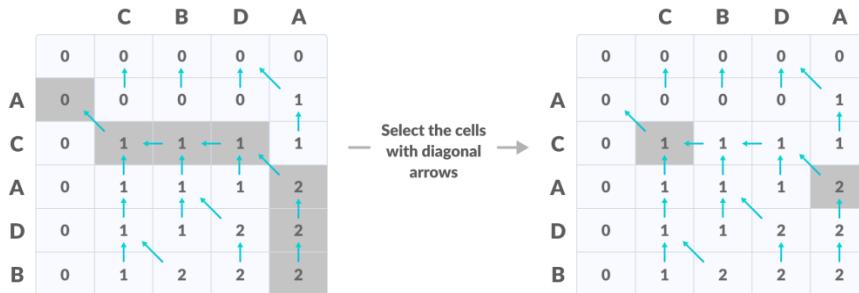
	C	B	D	A
C	0	0	0	0
B	0			
A	0			
D	0			
B	0			

6. The value in the last row and the last column is the length of the longest common subsequence.

	C	B	D	A
C	0	0	0	0
B	0			
A	0			
D	0			
B	0			

The bottom right corner is the length of the LCS

7. In order to find the longest common subsequence, start from the last element and follow the direction of the arrow. The elements corresponding to () symbol form the longest common subsequence.



Create a path according to the arrows

Thus, the longest common subsequence is CA.



How is a dynamic programming algorithm more efficient than the recursive algorithm while solving an LCS problem?

The method of dynamic programming reduces the number of function calls. It stores the result of each function call so that it can be used in future calls without the need for redundant calls.

In the above dynamic algorithm, the results obtained from each comparison between elements of X and the elements of Y are stored in a table so that they can be used in future computations.

So, the time taken by a dynamic approach is the time taken to fill the table (ie. O(mn)). Whereas, the recursion algorithm has the complexity of 2max(m, n).

Longest Common Subsequence Algorithm

```
X and Y be two given sequences
Initialize a table LCS of dimension X.length * Y.length
X.label = X
Y.label = Y
LCS[0][] = 0
LCS[] [0] = 0
Start from LCS[1][1]
Compare X[i] and Y[j]
If X[i] = Y[j]
    LCS[i][j] = 1 + LCS[i-1, j-1]
    Point an arrow to LCS[i][j]
Else
    LCS[i][j] = max(LCS[i-1][j], LCS[i][j-1])
    Point an arrow to max(LCS[i-1][j], LCS[i][j-1])
```

Program to implement Longest Subsequence Algorithm

```

#include <stdio.h>
#include <string.h>

int i, j, m, n, LCS_table[20][20];
char S1[20] = "ACADB", S2[20] = "CBDA", b[20][20];

void lcsAlgo() {
    m = strlen(S1);
    n = strlen(S2);

    // Filling 0's in the matrix
    for (i = 0; i <= m; i++)
        LCS_table[i][0] = 0;
    for (i = 0; i <= n; i++)
        LCS_table[0][i] = 0;

    // Building the matrix in bottom-up way
    for (i = 1; i <= m; i++) {
        for (j = 1; j <= n; j++) {
            if (S1[i - 1] == S2[j - 1]) {
                LCS_table[i][j] = LCS_table[i - 1][j - 1] + 1;
            } else if (LCS_table[i - 1][j] >= LCS_table[i][j - 1]) {
                LCS_table[i][j] = LCS_table[i - 1][j];
            } else {
                LCS_table[i][j] = LCS_table[i][j - 1];
            }
        }
    }

    int index = LCS_table[m][n];
    char lcsAlgo[index + 1];
    lcsAlgo[index] = '\0';

    int i = m, j = n;
    while (i > 0 && j > 0) {
        if (S1[i - 1] == S2[j - 1]) {
            lcsAlgo[index - 1] = S1[i - 1];
            i--;
            j--;
            index--;
        } else if (LCS_table[i - 1][j] > LCS_table[i][j - 1])
            i--;
        else
            j--;
    }

    // Printing the sub sequences
    printf("S1 : %s \nS2 : %s \n", S1, S2);
    printf("LCS: %s", lcsAlgo);
}

int main() {
    lcsAlgo();
    printf("\n");
}

```

Output

S1 : ACADB
S2 : CBDA
LCS: CB

Watch Videos:-

[Longest Common Subsequence \(LCS\) - Recursion and Dynamic Programming](#)
[Longest Common Subsequence- Dynamic Programming](#)

MultiStage Graph

A Multistage graph is a directed, weighted graph in which the nodes can be divided into a set of stages such that all edges are from a stage to next stage only (In other words there is no edge between vertices of same stage and from a vertex of current stage to previous stage).

The vertices of a multistage graph are divided into n number of disjoint subsets $S = \{ S_1, S_2, S_3, \dots, S_n \}$, where S_1 is the source and S_n is the sink (destination). The cardinality of S_1 and S_n are equal to 1. i.e., $|S_1| = |S_n| = 1$.

We are given a multistage graph, a source and a destination, we need to find shortest path from source to destination. By convention, we consider source at stage 1 and destination as last stage.

Algorithm for Multistage Graph

```

Algorithm MULTI_STAGE(G, k, n, p)
// Description: Solve multi-stage problem using dynamic programming

// Input:
k: Number of stages in graph G = (V, E)
c[i, j]:Cost of edge (i, j)

// Output: p[1:k]:Minimum cost path

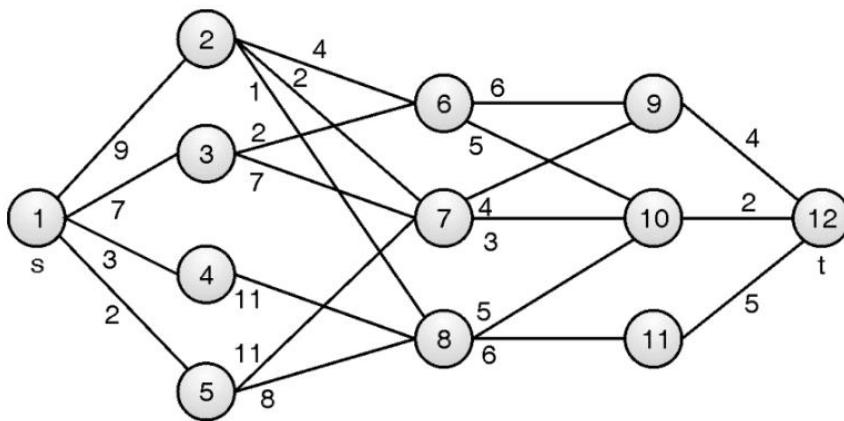
cost[n] ← 0
for j ← n - 1 to 1 do
    //Let r be a vertex such that (j, r) in E and c[j, r] + cost[r] is minimum
    cost[j] ← c[j, r] + cost[r]
    π[j] ← r
end

//Find minimum cost path
p[1] ← 1
p[k] ← n

for j ← 2 to k - 1 do
    p[j] ← π[p[j - 1]]
end

```

Let we understand the working of multistage graph problem by the help of example.



This problem is solved by using tabular method, So we draw a table contain all vertices(v), cost(c) and destination (d).

v	1	2	3	4	5	6	7	8	9	10	11	12
Cost												
d												

Here our main objective is to **select those path which have minimum cost. So we can say that it is an minimization problem.** It can be solved by the principle of optimal which says the sequence of decisions. That means in every stage we have to take decision.

In this problem we will start from 12 so its distance is 12 and cost is 0.

Now calculate for V(12) and stage 5.

Here $\text{cost}(5, 12) = 0$ I.e **Cost(stage number , vertex)**. Now update distance and cost in table for stage 5

v	1	2	3	4	5	6	7	8	9	10	11	12
Cost												0
d												12

Now calculate for V(9 ,10 ,11) and stage 4.

$$\begin{aligned}\text{cost}(4, 9) &= 4 \\ \text{cost}(4, 10) &= 2 \\ \text{cost}(4, 11) &= 5\end{aligned}$$

Here the distance is 12. So update the new cost and distance in table for stage 4

v	1	2	3	4	5	6	7	8	9	10	11	12
Cost									4	2	5	0
d									12	12	12	12

Now calculate for V(6 , 7 , 8) and stage 3.

For calculating v(6) we must find out there connecting link in previous stage i.e 4. which is 9 and 10.

$\text{cost}(v,d) + \text{cost}(\text{stage}+1, \text{vs})$, where vs; vertices of that stage.

For v(6)

$$\begin{aligned}\text{cost}(6, 9) + \text{cost}(4, 9) \\ 6 + 4 = 10\end{aligned}$$

$$\begin{aligned}\text{cost}(6, 10) + \text{cost}(4, 10) \\ 5 + 2 = 7\end{aligned}$$

$$\min[10, 7]$$

Now we find minimum from [10 , 7] which is 7 and the vertex give the minimum cost is 10

For v(7)

$$\begin{aligned}\text{cost}(7, 9) + \text{cost}(4, 9) \\ 4 + 4 = 8\end{aligned}$$

$$\begin{aligned}\text{cost}(7, 10) + \text{cost}(4, 10) \\ 3 + 2 = 5\end{aligned}$$

$\min[8, 5]$

Now we find minimum from $[8, 5]$ which is 5 and the vertex give the minimum cost is 10.

For v(8)

$\text{cost}(8, 10) + \text{cost}(4, 10)$

$$5 + 2 = 7$$

$\text{cost}(8, 11) + \text{cost}(4, 11)$

$$6 + 5 = 11$$

$\min[7, 11]$

Now we find minimum from $[7, 11]$ is 7 and the vertex given the minimum cost is 10.

So Now update the new cost and distance in table for stage 3

v	1	2	3	4	5	6	7	8	9	10	11	12
Cost						7	5	7	4	2	5	0
d						10	10	10	12	12	12	12

Now calculate for V(2, 3, 4, 5) and stage 2.

For calculating v(2) we must find out there connecting link in previous stage i.e 3. which is 6, 7 and 8.

$\text{cost}(v, d) + \text{cost}(\text{stage}+1, \text{vs})$, where vs; vertices of that stage.

For v(2) find cost(stage,vertex) i.e cost(2,2)

$\text{cost}(2, 6) + \text{cost}(3, 6)$

$$4 + 7 = 11$$

$\text{cost}(2, 7) + \text{cost}(3, 7)$

$$2 + 5 = 7$$

$\text{cost}(2, 8) + \text{cost}(3, 8)$

$$1 + 7 = 8$$

$\min[11, 7, 8]$

Now we find minimum from $[11, 7, 8]$ is 7 and the vertex given the minimum cost is 7.

For v(3) find cost(stage,vertex) i.e cost(2,3)

$\text{cost}(3, 6) + \text{cost}(3, 6)$

$$2 + 7 = 9$$

$\text{cost}(3, 7) + \text{cost}(3, 7)$

$$7 + 5 = 12$$

$\min[9, 12]$

Now we find minimum from $[9, 12]$ is 9 and the vertex given the minimum cost is 6

For v(4) find cost(stage,vertex) i.e cost(2,4)

$\text{cost}(4,8) + \text{cost}(3,8)$
 $11 + 7 = 18$

So here the cost is 18 and the vertex given the minimum cost is 8.

For v(5) find cost(stage,vertex) i.e cost(2,5)

$\text{cost}(5,7) + \text{cost}(3,7)$
 $11 + 5 = 16$

$\text{cost}(5,8) + \text{cost}(3,8)$
 $8 + 7 = 15$

$\min[16,15]$

Now we find minimum from [16,15] is 15 and the vertex given the minimum cost is 8

So Now update the new cost and distance in table for stage 2.

V	1	2	3	4	5	6	7	8	9	10	11	12
Cost		7	9	18	15	7	5	7	4	2	5	0
d		7	6	8	8	10	10	10	12	12	12	12

Now calculate for V(1) and stage 1.

For calculating v(1) we must find out there connecting link in previous stage i.e 2. which is 2 , 3 , 4 and 5.

$\text{cost}(v,d) + \text{cost}(\text{stage}+1,vs)$, where vs; vertices of that stage.

For v(1) find cost(stage,vertex) i.e cost(1,1)

$\text{cost}(1,2) + \text{cost}(2,2)$
 $9 + 7 = 16$

$\text{cost}(1,3) + \text{cost}(2,3)$
 $7 + 9 = 16$

$\text{cost}(1,4) + \text{cost}(2,4)$
 $3 + 18 = 21$

$\text{cost}(1,5) + \text{cost}(2,5)$
 $2 + 15 = 17$

$\min[16,16,21,17]$

Now we find minimum from [16,16,21,17] is 16 and the vertex given the minimum cost is 2,3 because here we get 16 twice so we consider both vertices.

So Now update the new cost and distance in table for stage 1.

V	1	2	3	4	5	6	7	8	9	10	11	12
Cost	16	7	9	18	15	7	5	7	4	2	5	0
d	2 / 3	7	6	8	8	10	10	10	12	12	12	12

Note: Formula used for calculating cost for any stage.

$$\text{Cost}(x, y) = \min [\text{cost}(y, v) + \text{cost}(x+1, v)]$$

Where x =stage number , y =current vertex number v =vertex of stage $x+1$.

Now we apply dynamic programming, we know dynamic programming is a sequence of decision on the basis of available data.

Now here data is available in the above table. For solving let we start from vertex 1 to onward. Here decision is taken in forward direction.

First we find $d(\text{stage}, \text{vertex})$ fro taking decision such as , here for 1 we have two value of d 2/3 so we will take 2 first

$$\begin{aligned} d(1, 1) &= 2 \\ d(2, 2) &= 7 \\ d(3, 7) &= 10 \\ d(4, 10) &= 12 \end{aligned}$$

so now we can say that the shortest path will be : $2 \rightarrow 7 \rightarrow 10 \rightarrow 12$

Now we take decision by taking 3 for vertex 1.

$$\begin{aligned} d(1, 1) &= 3 \\ d(2, 3) &= 6 \\ d(3, 6) &= 10 \\ d(4, 10) &= 12 \end{aligned}$$

So now we can say that the shortest path will be : $3 \rightarrow 6 \rightarrow 10 \rightarrow 12$ Here we will consider or select only one option but here we have two path it is optional.

Program to implement multistage Graph

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

#define loop(a, b, c) for(a=b; a<c; a++)
#define aloop(a, b, c) for(a=b; a>=c; a--)

int g[50][50];
int stage[50][50];
int distance[50];
int n;

int get_min(int a, int b)
{
    return a > b? b: a;
}

int main()
{
    int i, j, x, y, c, l, k, s;
    printf("Enter number of stages: ");
    scanf("%d", &l);

    c = 1;
    n = 1;
    stage[0][0] = 0;
    stage[0][1] = -1;

    loop(i, 1, l-1)
    {
        printf("Enter number of nodes in stage %d: ", i+1);
        scanf("%d", &x);
        n += x;
        loop(j, 0, x)
        {
            stage[i][j] = c;
            c++;
        }
        stage[i][j] = -1;
    }

    stage[l-1][0] = n, stage[l-1][1] = -1;
    n+=1;

    loop(i, 0, n)
    {
        distance[i] = INT_MAX;
        loop(j, 0, n) g[i][j] = INT_MAX;
    }

    printf("Enter edges:\n");

    loop(i, 0, l-1)
    {
        printf("Enter edges between stages %d and %d\nEnter -1 to stop\n", i, i+1);
        printf("Nodes are\n");
        j = 0;
        printf("Layer\tNodes\n");
    }
}
```

```

printf("%d\t", i);
while(stage[i][j]!=-1)
{
    printf("%d ", stage[i][j]);
    ++j;
}

j=0;
printf("\n%d\t", i+1);
while(stage[i+1][j]!=-1)
{
    printf("%d ", stage[i+1][j]);
    ++j;
}

printf("\n");
while(1)
{
    scanf("%d %d %d", &x, &y, &c);
    if(x == -1 || y == -1) break;
    g[x][y] = c;
}
}

j = 0;
while(stage[l-1][j]!=-1)
{
    distance[stage[l-1][j]] = 0;
    j++;
}

aloop(i, l-2, 0)
{
    j = 0;
    while(stage[i][j] != -1)
    {
        s = stage[i][j];

        loop(k, 0, n)
        {
            if(g[s][k] != INT_MAX)
            {
                distance[s] = get_min(distance[s], g[s][k] + distance[k]);
            }
        }
        j++;
    }
}

printf("Shortest path is: %d", distance[0]);

return 0;
}

```

Output

```

Enter number of stages: 4
Enter number of nodes in stage 2: 3
Enter number of nodes in stage 3: 3
Enter edges:

```

```
Enter edges between stages 0 and 1
Enter -1 to stop
Nodes are
Layer    Nodes
0        0
1        1 2 3
0 1 3
0 2 4
0 3 6
-1 -1 -1
Enter edges between stages 1 and 2
Enter -1 to stop
Nodes are
Layer    Nodes
1        1 2 3
2        4 5 6
1 4 18
1 5 20
2 5 30
2 6 20
3 4 10
3 6 7
-1 -1 -1
Enter edges between stages 2 and 3
Enter -1 to stop
Nodes are
Layer    Nodes
2        4 5 6
3        7
4 7 70
5 7 20
6 7 8
-1 -1 -1
```

Shortest path is: 21

Watch Videos:-

[MultiStage Graph - Dynamic Programming](#)
[MultiStage Graph \(Program\) - Dynamic Programming](#)

Matrix Chain Multiplication

Matrix chain multiplication problem: Determine the optimal parenthesization of a product of n matrices.

Matrix chain multiplication (or Matrix Chain Ordering Problem, MCOP) is an optimization problem that to find the most efficient way to multiply a given sequence of matrices. The problem is not actually to perform the multiplications but merely to decide the sequence of the matrix multiplications involved.

The matrix multiplication is associative as no matter how the product is parenthesized, the result obtained will remain the same. For example, for four matrices A, B, C, and D, we would have:

$$((AB)C)D = ((A(BC))D) = (AB)(CD) = A((BC)D) = A(B(CD))$$

However, the order in which the product is parenthesized affects the number of simple arithmetic operations needed to compute the product.

For example, if A is a 10×30 matrix, B is a 30×5 matrix, and C is a 5×60 matrix, then computing $(AB)C$ needs $(10 \times 30 \times 5) + (10 \times 5 \times 60) = 1500 + 3000 = 4500$ operations while computing $A(BC)$ needs $(30 \times 5 \times 60) + (10 \times 30 \times 60) = 9000 + 18000 = 27000$ operations. Clearly, the first method is more efficient.

The idea is to break the problem into a set of related subproblems that group the given matrix to yield the lowest total cost.

Following is the recursive algorithm to find the minimum cost:

- Take the sequence of matrices and separate it into two subsequences.
- Find the minimum cost of multiplying out each subsequence.
- Add these costs together, and add in the price of multiplying the two result matrices.
- Do this for each possible position at which the sequence of matrices can be split, and take the minimum over all of them.

For example, if we have four matrices ABCD, we compute the cost required to find each of $(A)(BCD)$, $(AB)(CD)$, and $(ABC)(D)$, making recursive calls to find the minimum cost to compute ABC, AB, CD, and BCD and then choose the best one. Better still, this yields the minimum cost and demonstrates the best way of doing the multiplication.

Program to implement Matrix Chain Multiplication Problem

```

// Matrix chain multiplication using recursion
#include <limits.h>
#include <stdio.h>

// Matrix Ai has dimension p[i-1] x p[i]
// for i = 1 . . . n
int MatrixChainOrder(int p[], int i, int j)
{
    if (i == j)
        return 0;
    int k;
    int min = INT_MAX;
    int count;

    // Place parenthesis at different places
    // between first and last matrix,
    // recursively calculate count of multiplications
    // for each parenthesis placement
    // and return the minimum count
    for (k = i; k < j; k++)
    {
        count = MatrixChainOrder(p, i, k)
            + MatrixChainOrder(p, k + 1, j)
            + p[i - 1] * p[k] * p[j];

        if (count < min)
            min = count;
    }

    // Return minimum count
    return min;
}

// Driver code
int main()
{
    // int arr[] = { 1, 2, 3, 4, 3 };
    int arr[] = { 5, 4, 6, 2, 7 };
    int N = sizeof(arr) / sizeof(arr[0]);

    // Function call
    printf("Minimum number of multiplications is %d ",
        MatrixChainOrder(arr, 1, N - 1));
    getchar();
    return 0;
}

```

Output

Minimum number of multiplications is 158

Watch Videos:-

- [Matrix Chain Multiplication - Dynamic Programming](#)
- [Matrix Chain Multiplication using Dynamic Programming Formula](#)
- [Matrix Chain Multiplication \(Program\) - Dynamic Programming](#)

Bellman-Ford (Single Source Shortest Path) Algorithm

Bellman Ford algorithm is a single-source shortest path algorithm. This algorithm is used to find the shortest distance from the single vertex to all the other vertices of a weighted graph. There are various other algorithms used to find the shortest path like Dijkstra algorithm, etc. If the weighted graph contains the negative weight values, then the Dijkstra algorithm does not confirm whether it produces the correct answer or not. In contrast to Dijkstra algorithm, bellman ford algorithm guarantees the correct answer even if the weighted graph contains the negative weight values.

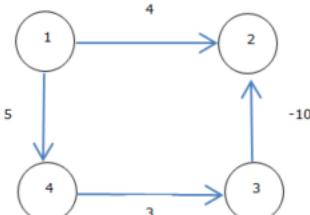
Rule of this algorithm

We will go on relaxing all the edges ($n - 1$) times where,
 n = number of vertices

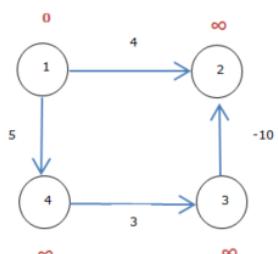
Steps

- o Start with weighted graph
- o Choose a starting vertex(assign 0) and assign infinity path to all other vertex
- o Visit each edge and relax the path distance, if they are inaccurate(min value).

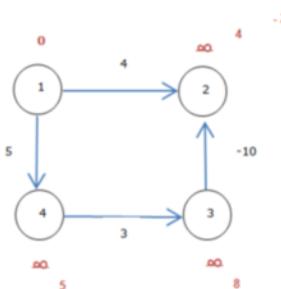
Example



Choose a starting vertex(assign 0) and assign infinity path to all other vertex



Visit each edge and relax the path distance, if they are inaccurate(min value).



Watch Videos:-

[Bellman Ford Algorithm - Single Source Shortest Path - Dynamic Programming](#)

Knapsack Problem (0/1)

The basic idea of Knapsack dynamic programming is to use a table to store the solutions of solved subproblems. If you face a subproblem again, you just need to take the solution in the table without having to solve it again.

The Knapsack Problem is also called as rucksack problem. A basic idea to solve knapsack problem given with set of items each with a mass and value. Describe every individual item included in a collection so that total weight is less than or equal to a given limit and total value is as large as possible. A fixed size knapsack is defined at the beginning. It must fill with most valuable items.

Knapsack algorithm can be further divided into two types:

- The 0/1 Knapsack problem using dynamic programming. In this Knapsack algorithm type, **each package can be taken or not taken**. Besides, the thief cannot take a fractional amount of a taken package or take a package more than once. This type can be solved by Dynamic Programming Approach.
- Fractional Knapsack problem algorithm. This type can be solved by Greedy Strategy.

Given weights and values of n items, put these items in a knapsack of capacity W to get the maximum total value in the knapsack. In other words, given two integer arrays val[0..n-1] and wt[0..n-1] which represent values and weights associated with n items respectively. Also given an integer W which represents knapsack capacity, find out the maximum value subset of val[] such that sum of the weights of this subset is smaller than or equal to W. You cannot break an item, either pick the complete item or don't pick it (0-1 property).

Example 1:

```
m=8 n=4
p={1,2,5,6}
w={2,3,4,5}
```

			0	1	2	3	4	5	6	7	8
pi	wi	0	0	0	0	0	0	0	0	0	0
1	2	1	0	0	1	1	1	1	1	1	1
2	3	2	0	0	1	2	2	3	3	3	3
5	4	3	0	0	1	2	5	5	6	7	7
6	5	4	0	0	1	2	5	6	6	7	8

{0,1,0,1}

Set method (p, vv)

```
S0 = { (0,0) }
S01 = {1,2} // add (1,2)

S1 = { (0,0), (1,2) }
S11 = { (2,3) (3,5) } // add (2,3)

S2 = { (0,0) (1,2) (2,3) (3,5) }
S21 = { (5,4) (6,6) (7,7) (8,9) } // add (4,4), exceed the maximum limit

S3 = { (0,0) (1,2) (3,5) (5,4) (6,6) (7,7) } // weight is decreasing (3,5), (5,4) remove min
S31 = { (6,5) (7,7) (8,8) (11,9) (12,11) (13,12) } // add (6,5), exceeded the limit

S4 = { (0,0) (1,2) (2,3) (6,6) (6,5) (7,7) (8,8) } // weight is decreasing (6,6), (6,5) remove min
```

Example 2;

$w = \{3, 4, 6, 5\}$
 $p = \{2, 3, 1, 4\}$

			0	1	2	3	4	5	6	7	8
pi	wi	0	0	0	0	0	0	0	0	0	0
2	3	1	0	0	0	2	2	2	2	2	2
3	4	2	0	0	0	2	3 *	3	3	5	5
4	5	3	0	0	0	2	3	4	4	5	6
1	6	4	0	0	0	2	3	4	4	5	6

* max(selected row ptofit + select value in column (current weight - selected weight), corresponding value of above cell)

$$\text{Max } (3 + (4-4), 2) = \max(3 + \text{val}(0), 2) = \max(3 + 0, 2) = 3$$

{1, 0, 0, 1}

Program to implement 0/1 Knapsack Problem

```
// 0/1 Knapsack Problem
#include<stdio.h>
int max(int a, int b) { return (a > b)? a : b; }
int knapSack(int W, int wt[], int val[], int n)
{
    int i, w;
    int K[n+1][W+1];
    for (i = 0; i <= n; i++)
    {
        for (w = 0; w <= W; w++)
        {
            if (i==0 || w==0)
                K[i][w] = 0;
            else if (wt[i-1] <= w)
                K[i][w] = max(val[i-1] + K[i-1][w-wt[i-1]], K[i-1][w]);
            else
                K[i][w] = K[i-1][w];
        }
    }
    return K[n][W];
}
int main()
{
    int i, n, val[20], wt[20], W;

    printf("Enter number of items:");
    scanf("%d", &n);

    printf("Enter value and weight of items:\n");
    for(i = 0;i < n; ++i){
        scanf("%d%d", &val[i], &wt[i]);
    }
    printf("Enter size of knapsack:");
    scanf("%d", &W);
    printf("%d", knapSack(W, wt, val, n));
    return 0;
}
```

Output

```
Enter number of items:4
Enter value and weight of items:
1 2
2 3
5 4
6 5
Enter size of knapsack:8
8
```

Watch Videos:-

[0/1 Knapsack - Two Methods - Dynamic Programming](#)
[0/1 knapsack problem-Dynamic Programming](#)
[0/1 Knapsack Problem \(Program\) - Dynamic Programming](#)

Optimal Binary Search Tree

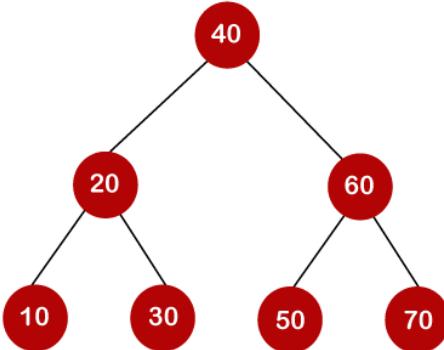
Optimal Binary Search Tree extends the concept of Binary search tree. **Binary Search Tree (BST)** is a nonlinear data structure which is used in many scientific applications for reducing the search time. In BST, left child is smaller than root and right child is greater than root. This arrangement simplifies the search procedure.

As we know that in binary search tree, the nodes in the left subtree have lesser value than the root node and the nodes in the right subtree have greater value than the root node.

We know the key values of each node in the tree, and we also know the frequencies of each node in terms of searching means how much time is required to search a node. The frequency and key-value determine the overall cost of searching a node. The cost of searching is a very important factor in various applications. The overall cost of searching a node should be less. The time required to search a node in BST is more than the balanced binary search tree as a balanced binary search tree contains a lesser number of levels than the BST. There is one way that can reduce the cost of a binary search tree is known as an optimal binary search tree.

Let's understand through an example.

If the keys are 10, 20, 30, 40, 50, 60, 70



In the above tree, all the nodes on the left subtree are smaller than the value of the root node, and all the nodes on the right subtree are larger than the value of the root node. The maximum time required to search a node is equal to the minimum height of the tree, equal to logn.

Now we will see how many binary search trees can be made from the given number of keys.

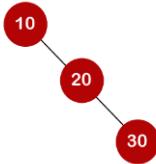
For example: 10, 20, 30 are the keys, and the following are the binary search trees that can be made out from these keys.

The Formula for calculating the number of trees:

$$\frac{2^n}{n+1} C_n$$

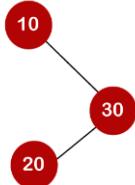
When we use the above formula, then it is found that total 5 number of trees can be created.

The cost required for searching an element depends on the comparisons to be made to search an element. Now, we will calculate the average cost of time of the above binary search trees.



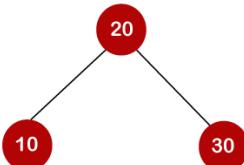
In the above tree, total number of 3 comparisons can be made. The average number of comparisons can be made as:

$$\text{average number of comparisons} = \frac{1+2+3}{3} = 2$$



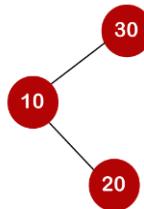
In the above tree, the average number of comparisons that can be made as:

$$\text{average number of comparisons} = \frac{1+2+3}{3} = 2$$



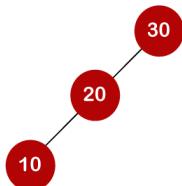
In the above tree, the average number of comparisons that can be made as:

$$\text{average number of comparisons} = \frac{1+2+2}{3} = 5/3$$



In the above tree, the total number of comparisons can be made as 3. Therefore, the average number of comparisons that can be made as:

$$\text{average number of comparisons} = \frac{1+2+3}{3} = 2$$



In the above tree, the total number of comparisons can be made as 3. Therefore, the average number of comparisons that can be made as:

$$\text{average number of comparisons} = \frac{1+2+3}{3} = 2$$

In the third case, the number of comparisons is less because the height of the tree is less, so it's a balanced binary search tree.

Till now, we read about the height-balanced binary search tree. To find the optimal binary search tree, we will determine the frequency of searching a key.

Let's assume that frequencies associated with the keys 10, 20, 30 are 3, 2, 5.

The above trees have different frequencies. The tree with the lowest frequency would be considered the optimal binary search tree. The tree with the frequency 17 is the lowest, so it would be considered as the optimal binary search tree.

Dynamic Approach

Consider the below table, which contains the keys and frequencies.

	1	2	3	4	
Keys →	10	20	30	40	
Frequency →	4	2	6	3	

i	0	1	2	3	4
0					
1					
2					
3					
4					

First, we will calculate the values where $j - i$ is equal to zero.

When $i = 0$, $j = 0$, then $j - i = 0$

When $i = 1$, $j = 1$, then $j - i = 0$

When $i = 2$, $j = 2$, then $j - i = 0$

When $i = 3$, $j = 3$, then $j - i = 0$

When $i = 4$, $j = 4$, then $j - i = 0$

Therefore, $c[0, 0] = 0$, $c[1, 1] = 0$, $c[2, 2] = 0$, $c[3, 3] = 0$, $c[4, 4] = 0$

Now we will calculate the values where $j - i$ equal to 1.

When $j = 1$, $i = 0$ then $j - i = 1$

When $j = 2$, $i = 1$ then $j - i = 1$

When $j = 3$, $i = 2$ then $j - i = 1$

When $j = 4$, $i = 3$ then $j - i = 1$

Now to calculate the cost, we will consider only the jth value.

The cost of $c[0,1]$ is 4 (The key is 10, and the cost corresponding to key 10 is 4).
 The cost of $c[1,2]$ is 2 (The key is 20, and the cost corresponding to key 20 is 2).
 The cost of $c[2,3]$ is 6 (The key is 30, and the cost corresponding to key 30 is 6)
 The cost of $c[3,4]$ is 3 (The key is 40, and the cost corresponding to key 40 is 3)

	0	1	2	3	4
0	0	4			
1		0	2		
2			0	6	
3				0	3
4					0

Now we will calculate the values where $j-i = 2$

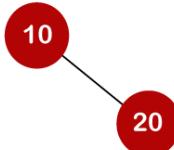
When $j = 2$, $i = 0$ then $j - i = 2$

When $j = 3$, $i = 1$ then $j - i = 2$

When $j = 4$, $i = 2$ then $j - i = 2$

In this case, we will consider two keys.

- When $i=0$ and $j=2$, then keys 10 and 20. There are two possible trees that can be made out from these two keys shown below:



In the first binary tree, cost would be: $4 * 1 + 2 * 2 = 8$

In the second binary tree, cost would be: $4 * 2 + 2 * 1 = 10$

The minimum cost is 8; therefore, $c[0,2] = 8$

	0	1	2	3	4
0	0	4	8		
1		0	2		
2			0	6	
3				0	3
4					0

- When $i=1$ and $j=3$, then keys 20 and 30. There are two possible trees that can be made out from these two keys shown below:

In the first binary tree, cost would be: $1 * 2 + 2 * 6 = 14$
 In the second binary tree, cost would be: $1 * 6 + 2 * 2 = 10$

The minimum cost is 10; therefore, $c[1,3] = 10$

- When $i=2$ and $j=4$, we will consider the keys at 3 and 4, i.e., 30 and 40. There are two possible trees that can be made out from these two keys shown as below:

In the first binary tree, cost would be: $1 * 6 + 2 * 3 = 12$
 In the second binary tree, cost would be: $1 * 3 + 2 * 6 = 15$

The minimum cost is 12, therefore, $c[2,4] = 12$

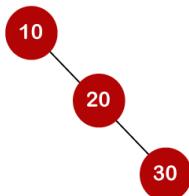
i \ j	0	1	2	3	4
0	0	4	8 ¹		
1		0	2	10 ³	
2			0	6	12 ³
3				0	3
4					0

Now we will calculate the values when $j - i = 3$

When $j = 3$, $i = 0$ then $j - i = 3$
 When $j = 4$, $i = 1$ then $j - i = 3$

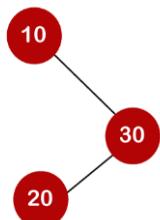
- When $i=0$, $j=3$ then we will consider three keys, i.e., 10, 20, and 30.

The following are the trees that can be made if 10 is considered as a root node.



In the above tree, 10 is the root node, 20 is the right child of node 10, and 30 is the right child of node 20.

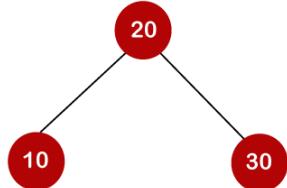
Cost would be: $1 * 4 + 2 * 2 + 3 * 6 = 26$



In the above tree, 10 is the root node, 30 is the right child of node 10, and 20 is the left child of node 20.

Cost would be: $1*4 + 2*6 + 3*2 = 22$

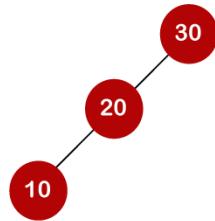
The following tree can be created if 20 is considered as the root node.



In the above tree, 20 is the root node, 30 is the right child of node 20, and 10 is the left child of node 20.

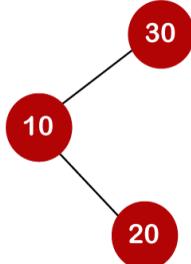
Cost would be: $1*2 + 4*2 + 6*2 = 22$

The following are the trees that can be created if 30 is considered as the root node.



In the above tree, 30 is the root node, 20 is the left child of node 30, and 10 is the left child of node 20.

Cost would be: $1*6 + 2*2 + 3*4 = 22$



In the above tree, 30 is the root node, 10 is the left child of node 30 and 20 is the right child of node 10.

Cost would be: $1*6 + 2*4 + 3*2 = 20$

Therefore, the minimum cost is 20 which is the 3rd root. So, c[0,3] is equal to 20.

- When $i=1$ and $j=4$ then we will consider the keys 20, 30, 40

$$\begin{aligned}
 c[1,4] &= \min\{c[1,1] + c[2,4], c[1,2] + c[3,4], c[1,3] + c[4,4]\} + 11 \\
 &= \min\{0+12, 2+3, 10+0\} + 11 \\
 &= \min\{12, 5, 10\} + 11
 \end{aligned}$$

The minimum value is 5; therefore, $c[1,4] = 5+11 = 16$

i \ j	0	1	2	3	4
0	0	4	8^1	20^3	
1		0	2	10^3	16^3
2			0	6	12^3
3				0	3
4					0

Now we will calculate the values when $j-i = 4$

When $j=4$ and $i=0$ then $j-i = 4$

In this case, we will consider four keys, i.e., 10, 20, 30 and 40. The frequencies of 10, 20, 30 and 40 are 4, 2, 6 and 3 respectively.

$$w[0, 4] = 4 + 2 + 6 + 3 = 15$$

If we consider 10 as the root node then

$$\begin{aligned}
 C[0, 4] &= \min\{c[0,0] + c[1,4]\} + w[0,4] \\
 &= \min\{0 + 16\} + 15 = 31
 \end{aligned}$$

If we consider 20 as the root node then

$$\begin{aligned}
 C[0,4] &= \min\{c[0,1] + c[2,4]\} + w[0,4] \\
 &= \min\{4 + 12\} + 15 \\
 &= 16 + 15 = 31
 \end{aligned}$$

If we consider 30 as the root node then,

$$\begin{aligned}
 C[0,4] &= \min\{c[0,2] + c[3,4]\} + w[0,4] \\
 &= \min\{8 + 3\} + 15 \\
 &= 26
 \end{aligned}$$

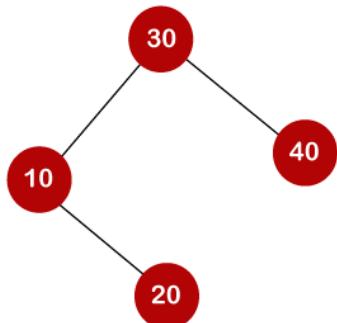
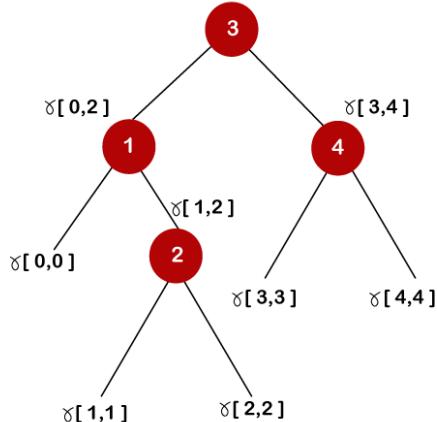
If we consider 40 as the root node then,

$$\begin{aligned}
 C[0,4] &= \min\{c[0,3] + c[4,4]\} + w[0,4] \\
 &= \min\{20 + 0\} + 15 \\
 &= 35
 \end{aligned}$$

In the above cases, we have observed that 26 is the minimum cost; therefore, $c[0,4]$ is equal to 26.

i \ j	0	1	2	3	4
0	0	4	8 ¹	20 ³	26 ³
1		0	2	10 ³	16 ³
2			0	6	12 ³
3				0	3
4					0

The optimal binary tree can be created as:



General formula for calculating the minimum cost is:

$$C[i,j] = \min\{c[i, k-1] + c[k, j]\} + w(i, j)$$

Program to implement Optimal Binary Search

```

// Dynamic Programming code for Optimal Binary Search - Tree Problem
#include <stdio.h>
#include <limits.h>

// A utility function to get sum of array elements
// freq[i] to freq[j]
int sum(int freq[], int i, int j);

/* A Dynamic Programming based function that calculates
minimum cost of a Binary Search Tree. */
int optimalSearchTree(int keys[], int freq[], int n)
{
    /* Create an auxiliary 2D matrix to store results
    of subproblems */
    int cost[n][n];

    /* cost[i][j] = Optimal cost of binary search tree
    that can be formed from keys[i] to keys[j].
    cost[0][n-1] will store the resultant cost */

    // For a single key, cost is equal to frequency of the key
    for (int i = 0; i < n; i++)
        cost[i][i] = freq[i];

    // Now we need to consider chains of length 2, 3, . . .
    // L is chain length.
    for (int L=2; L<=n; L++)
    {
        // i is row number in cost[][][]
        for (int i=0; i<=n-L+1; i++)
        {
            // Get column number j from row number i and
            // chain length L
            int j = i+L-1;
            int off_set_sum = sum(freq, i, j);
            cost[i][j] = INT_MAX;

            // Try making all keys in interval keys[i..j] as root
            for (int r=i; r<=j; r++)
            {
                // c = cost when keys[r] becomes root of this subtree
                int c = ((r > i)? cost[i][r-1]:0) +
                    ((r < j)? cost[r+1][j]:0) +
                    off_set_sum;
                if (c < cost[i][j])
                    cost[i][j] = c;
            }
        }
    }

    return cost[0][n-1];
}

// A utility function to get sum of array elements
// freq[i] to freq[j]
int sum(int freq[], int i, int j)
{
    int s = 0;
    for (int k = i; k <=j; k++)

```

```
s += freq[k];
return s;
}

// Driver program to test above functions
int main()
{
    int keys[] = {10, 20, 30, 40};
    int freq[] = {4, 2, 6, 3};
    int n = sizeof(keys)/sizeof(keys[0]);
    printf("Cost of Optimal BST is %d ",
           optimalSearchTree(keys, freq, n));
    return 0;
}
```

Output

Cost of Optimal BST is 26

Watch Videos:-

[Optimal Binary Search Tree \(Successful Search Only\)](#)

[Optimal Binary Search Tree Successful and Unsuccessful Probability](#)

Travelling Salesman Problem

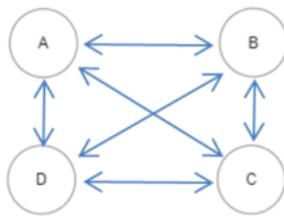
Traveling salesman problem is stated as, "Given a set of n cities and distance between each pair of cities, find the minimum length path such that it covers each city exactly once and terminates the tour at starting city."

There are obviously a lot of different routes to choose from, but finding the best one—the one that will require the least distance or cost

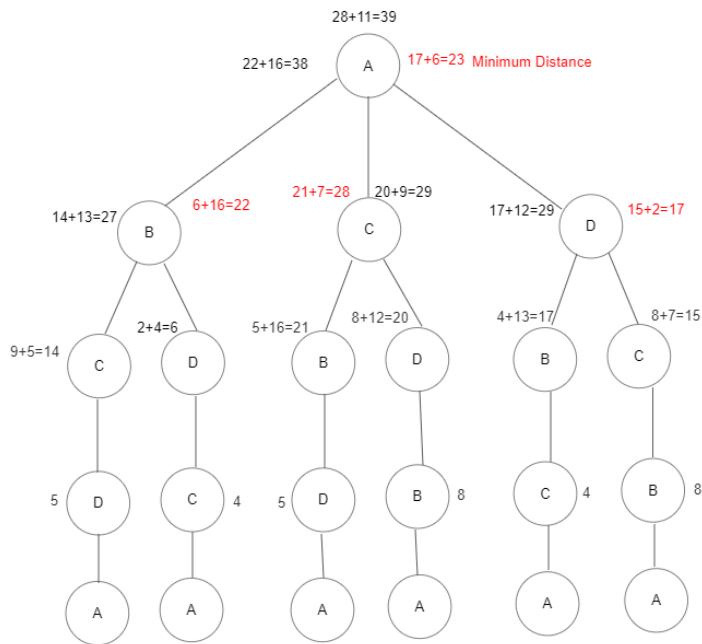
$$g(i, s) = \min [w(i, j) + g(j, \{s - j\})]$$

Example

	A	B	C	D
A	0	16	11	6
B	8	0	13	16
C	4	7	0	9
D	5	12	2	0



Tree Structure



Using Formula

$$\begin{aligned}
 g(A, \{a, c, d\}) &= \min [w(A, B) + g(B, \{C, D\}) = 16 + 22 = 38 \\
 &\quad w(A, C) + g(C, \{B, D\}) = 11 + 28 = 39 \\
 &\quad w(A, D) + g(D, \{B, C\}) = 6 + 17 = 23 \\
 &\quad]
 \end{aligned}$$

w(A,B) + g(B, {C,D})

$$\begin{aligned} g(B, \{C, D\}) &= \min [w(B, C) + g(C, \{D\}) = 13 + 14 = 27 \\ &\quad w(B, D) + g(D, \{C\}) = 16 + 6 = 22 \\ &\quad] \end{aligned}$$

$$\begin{aligned} g(C, \{D\}) &= w(C, D) + g(D, \Phi) \\ &= 9 + 5 = 14 \end{aligned}$$

$$\begin{aligned} g(D, \{C\}) &= w(D, C) + g(C, \Phi) \\ &= 2 + 4 = 6 \end{aligned}$$

w(A,C) + g(C, {B,D})

$$\begin{aligned} g(C, \{B, D\}) &= \min [w(C, B) + g(B, \{D\}) = 7 + 21 = 28 \\ &\quad w(C, D) + g(D, \{B\}) = 9 + 20 = 29 \\ &\quad] \end{aligned}$$

$$\begin{aligned} g(B, \{D\}) &= w(B, D) + g(D, \Phi) \\ &= 16 + 5 = 21 \end{aligned}$$

$$\begin{aligned} g(D, \{B\}) &= w(D, B) + g(B, \Phi) \\ &= 12 + 8 = 20 \end{aligned}$$

w(A,D) + g(D, {B,C})

$$\begin{aligned} g(D, \{B, C\}) &= \min [w(D, B) + g(B, \{C\}) = 12 + 17 = 29 \\ &\quad w(D, C) + g(C, \{B\}) = 2 + 15 = 17 \\ &\quad] \end{aligned}$$

$$\begin{aligned} g(B, \{C\}) &= w(B, C) + g(C, \Phi) \\ &= 13 + 4 = 17 \end{aligned}$$

$$\begin{aligned} g(C, \{B\}) &= w(C, B) + g(B, \Phi) \\ &= 7 + 8 = 15 \end{aligned}$$

A → D → C → B = 23

Program to Implement Travelling Salesperson Problem

```

#include <stdio.h>
int matrix[25][25], visited_cities[10], limit, cost = 0;

int tsp(int c)
{
    int count, nearest_city = 999;
    int minimum = 999, temp;

    for(count = 0; count < limit; count++)
    {
        if((matrix[c][count] != 0) && (visited_cities[count] == 0))
        {
            if(matrix[c][count] < minimum)
            {
                minimum = matrix[count][0] + matrix[c][count];
            }
            temp = matrix[c][count];
            nearest_city = count;
        }
    }

    if(minimum != 999)
    {
        cost = cost + temp;
    }
    return nearest_city;
}

void minimum_cost(int city)
{
    int nearest_city;
    visited_cities[city] = 1;
    printf("%d ", city + 1);
    nearest_city = tsp(city);

    if(nearest_city == 999)
    {
        nearest_city = 0;
        printf("%d", nearest_city + 1);
        cost = cost + matrix[city][nearest_city];
        return;
    }
    minimum_cost(nearest_city);
}

int main()
{
    int i, j;
    printf("Enter Total Number of Cities:\t");
    scanf("%d", &limit);
    printf("\nEnter Cost Matrix\n");

    for(i = 0; i < limit; i++)
    {
        printf("\nEnter %d Elements in Row[%d]\n", limit, i + 1);
        for(j = 0; j < limit; j++)
        {
            scanf("%d", &matrix[i][j]);
        }
    }
}

```

```

        }
        visited_cities[i] = 0;
    }
    printf("\nEntered Cost Matrix\n");

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

    printf("\n\nPath:\t");
    minimum_cost(0);
    printf("\n\nMinimum Cost: \t");
    printf("%d\n", cost);
    return 0;
}

```

Output

Enter Total Number of Cities: 4

Enter Cost Matrix

Enter 4 Elements in Row[1]

0 16 11 6

Enter 4 Elements in Row[2]

8 0 13 16

Enter 4 Elements in Row[3]

4 7 0 9

Enter 4 Elements in Row[4]

5 12 2 0

Entered Cost Matrix

0 16 11 6

8 0 13 16

4 7 0 9

5 12 2 0

Path: 1 4 3 2 1 (A->D->C->B)

Minimum Cost: 23

Watch Videos:-

[Traveling Salesperson Problem](#)

[Traveling Salesman Problem - Dynamic Programming using Formula](#)

[Traveling Salesman Problem](#)

Reliability Design

Reliability can be defined as the probability that a product will keep working beyond its specified interval of time, in the specified working conditions. It implies that if a keyboard has a 99% reliability over 1000 hours, it can still be faulty for 1% of the time within that 1000 hours.

Watch Videos:-

[Reliability Design - Dynamic Programming](#)

Binomial Coefficient

Computing binomial coefficient is very fundamental problem of mathematics and computer science. Binomial coefficient $C(n, k)$ defines coefficient of the term x^n in the expansion of $(1 + x)^n$. $C(n, k)$ also defines the number of ways to select any k items out of n items. Mathematically it is defined as,

$$C(n, k) = \frac{n!}{(n - k)! k!} \quad (0 \leq k \leq n)$$

$C(n, k)$ can be found in different ways. In this article we will discuss divide and conquer as well as dynamic programming approach.

Dynamic Programming was invented by Richard Bellman, 1950. It is a very general technique for solving optimization problems. Using Dynamic Programming requires that the problem can be divided into overlapping similar sub-problems. A recursive relation between the larger and smaller sub problems is used to fill out a table. The algorithm remembers solutions of the sub-problems and so does not have to recalculate the solutions.

Also optimal problem must have the principle of optimality (phase coined by Bellman) meaning that the optimal problem can be posed as the optimal solution among sub problems. This is not true for all optimal problems.

Dynamic Programming requires:

- o Problem divided into overlapping sub-problems
- o Sub-problem can be represented by a table
- o Principle of optimality, recursive relation between smaller and larger problems

Compared to a brute force recursive algorithm that could run exponential, the dynamic programming algorithm runs typically in quadratic time. (Recall the algorithms for the Fibonacci numbers.) The recursive algorithm ran in exponential time while the iterative algorithm ran in linear time. The space cost does increase, which is typically the size of the table. Frequently, the whole table does not have to store.

Binomial Coefficient using Dynamic Programming

- o Many sub problems are called again and again, since they have an overlapping sub problems property. Re-computations of the same sub problems is avoided by storing their results in the temporary array $C[i, j]$ in a bottom up manner.
- o The optimal substructure for using dynamic programming is stated as,

$$C[i, j] = \begin{cases} 1 & , \text{if } i=j \text{ or } j=0 \\ C[i-1, j-1] + C[i-1, j] & \text{otherwise} \end{cases}$$

In Table, index i indicates row and index j indicates column

$n \setminus k$	0	1	2	3	4	.	.	.	(k - 1)	k
0	1									
1	1	1								
2	1	2	1							
3	1	3	3	1						
4	1	4	6	4	1					
.	.									
.	.									
k	1									1
.	.									
.	.									
$n - 1$	1								$C(n - 1, k - 1)$	$C(n - 1, k)$
n	1									$C(n, k)$

- o This tabular representation of binomial coefficients is also known as Pascal's triangle.
- o Algorithm to solve this problem using dynamic programming is shown below

```

Algorithm BINOMIAL_DC (n, k)
// n is total number of items
// k is the number of items to be selected from n

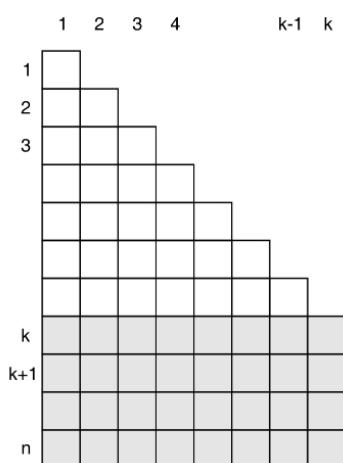
if k == 0 or k == n then
    return 1
else
    return DC_BINOMIAL(n - 1, k - 1) + DC_BINOMIAL(n - 1, k)
end

```

Complexity analysis

The cost of the algorithm is cost of filling out the table. Addition is the basic operation. Because $k \leq n$, the sum needs to be split into two parts, only the half of the table needs to be filled out for $i < k$ and the remaining part of the table is filled out across the entire row.

The growth pattern of these numbers is shown in the following figure.



$T(n, k) = \text{sum for upper triangle} + \text{sum for the lower rectangle}$

$$\begin{aligned}
 T(n, k) &= \sum_{i=1}^k \sum_{j=1}^{i-1} 1 + \sum_{i=k+1}^n \sum_{j=1}^k 1 = \sum_{i=1}^k (i-1) + \sum_{i=k+1}^n k \\
 &= (1 + 2 + 3 + \dots + k-1) + k \sum_{i=k+1}^n 1 = \frac{(k-1)k}{2} + k(n-k) \\
 &= \frac{k^2 - k}{2} + nk - k^2 = \frac{k^2 - k + 2nk - 2k^2}{2} = nk - \frac{k^2}{2} - \frac{k}{2} \\
 &= nk \quad (k \ll n)...
 \end{aligned}$$

$$T(n, k) = O(nk)$$

Program to implement Binomial Coefficient

```

// Binomial Coefficient -Dynamic Programming
#include <stdio.h>
#include <conio.h>

// Prototype of a utility function that returns minimum of two integers
int min (int a, int b);

// Returns value of Binomial Coefficient C(n, k)
int binomialCoeff (int n, int k)
{
    int C[n + 1][k + 1];
    int i, j;

    // Calculate value of Binomial Coefficient in bottom up manner
    for (i = 0; i <= n; i++)
    {
        for (j = 0; j <= min (i, k); j++)
        {
            // Base Cases
            if (j == 0 || j == i)
            {
                C[i][j] = 1;
            }
            // Calculate value using previously stored values
            else
            {
                C[i][j] = C[i - 1][j - 1] + C[i - 1][j];
            }
        }
    }
    return C[n][k];
}

// A utility function to return minimum of two integers
int min (int a, int b)
{
    return (a < b) ? a : b;
}

int main ()
{
    int n = 5, k = 2;
    printf ("Value of C(%d, %d) is %d ", n, k,
    binomialCoeff (n, k));
    return 0;
    getch();
}

```

Output

Value of C(5, 2) is 10

Watch Videos :-

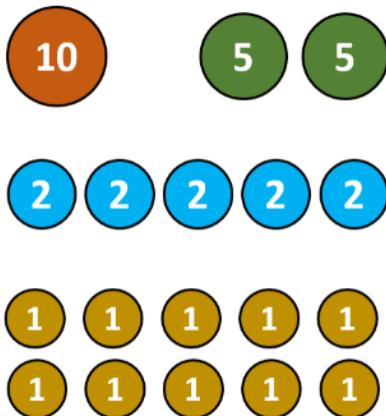
[Binomial Coefficient using Dynamic Programming](#)

[Binomial Coefficient using dynamic programming concepts in design and analysis of algorithm](#)

Making a Change – The Coin Change Problem

Making Change Problem – What is it ?

- Making Change problem is to find change for a given amount using a minimum number of coins from a set of denominations.
- Explanation : If we are given a set of denominations $D = \{d_0, d_1, d_2, \dots, d_n\}$ and if we want to change for some amount N , many combinations are possible. Suppose $\{d_1, d_2, d_5, d_8\}$, $\{d_0, d_2, d_4\}$, $\{d_0, d_5, d_7\}$ all feasible solutions.
- The aim of making a change is to find a solution with a minimum number of coins / denominations. Clearly, this is an optimization problem.
- This problem can also be solved by using a greedy algorithm. However, greedy does not ensure the minimum number of denominations.



Various denominations for amount 10

How to Solve Making Change using Dynamic Programming?

- Conventional / greedy approach selects largest denomination first which is not greater than remaining amount. On selecting denomination of size d_i in every step, problem size keeps reducing by amount d_i .
- If current denomination is not possible to select then select second largest denomination and so on. Continue this process until solution is found.
- However this approach may fail to find best solution. For example, suppose we have denominations $\{1, 4, 5\}$ and if we want change for 8, then conventional or greedy approach returns four coins, $\langle 5, 1, 1, 1 \rangle$, but optimal solution is $\langle 4, 4 \rangle$.
- Let us see how dynamic programming helps to find out the optimal solution.
- General assumption is that infinite coins are available for each denomination. We can select any denomination any number of times.

The Coin Change Problem is considered by many to be essential to understanding the paradigm of programming known as Dynamic Programming. The two often are always paired together because the coin change problem encompass the concepts of dynamic programming.

In other words, dynamic problem is a method of programming that is used to simplify a problem into smaller pieces. For example if you were asked simply what is $3 * 89$? you perhaps would not know the answer off of your head as you probably know what is $2 * 2$. However, if you knew what was $3 * 88$ (264) then certainly you can deduce $3 * 89$. All you would have to do is add 3 to the previous multiple and you would arrive at the answer of 267. Thus, that is a very simple explanation of what is dynamic programming and perhaps you can now see how it can be used to solve large time complexity problems effectively.

By keeping the above definition of dynamic programming in mind, we can now move forward to the Coin Change Problem. The following is an example of one of the many variations of the coin change problem. Given a list of coins i.e 1 cents, 5 cents and 10 cents, can you determine the total number of combinations of the coins in the given list to make up the number N?

Example 1: Suppose you are given the coins 1 cent, 5 cents, and 10 cents with N = 8 cents, what are the total number of combinations of the coins you can arrange to obtain 8 cents.

Input: N=8
Coins : 1, 5, 10
Output: 2

Explanation:

1 way:
1 + 1 + 1 + 1 + 1 + 1 + 1 + 1 = 8 cents.
2 way:
1 + 1 + 1 + 5 = 8 cents.

All you're doing is determining all of the ways you can come up with the denomination of 8 cents. Eight 1 cents added together is equal to 8 cents. Three 1 cent plus One 5 cents added is 8 cents. So there are a total of 2 ways given the list of coins 1, 5 and 10 to obtain 8 cents.

Program to Implement Coin Change Problem

```
// Coin Change Problem - Dynamic Programming
#include <stdio.h>

int count(int coins[], int n, int sum)
{
    int i, j, x, y;

    // We need sum+1 rows as the table is constructed
    // in bottom up manner using the base case 0
    // value case (sum = 0)
    int table[sum + 1][n];

    // Fill the entries for 0 value case (n = 0)
    for (i = 0; i < n; i++)
        table[0][i] = 1;

    // Fill rest of the table entries in bottom up manner
    for (i = 1; i < sum + 1; i++) {
        for (j = 0; j < n; j++) {
            // Count of solutions including S[j]
            x = (i - coins[j] >= 0) ? table[i - coins[j]][j]
                : 0;

            // Count of solutions excluding S[j]
            y = (j >= 1) ? table[i][j - 1] : 0;

            // total count
            table[i][j] = x + y;
        }
    }
    return table[sum][n - 1];
}

int main()
{
    int coins[] = { 1, 2, 3 };
    int n = sizeof(coins) / sizeof(coins[0]);
    int sum = 4;
    printf("%d ", count(coins, n, sum));
    return 0;
}
```

Output

4

Watch Video :-

[Coin Change Problem Number of ways to get total | Dynamic Programming](#)
[Change Problem-Minimum number of coins Dynamic Programming](#)

Assembly Line Scheduling

Assembly line scheduling is a manufacturing problem. In automobile industries, assembly lines are used to transfer parts from one station to another.

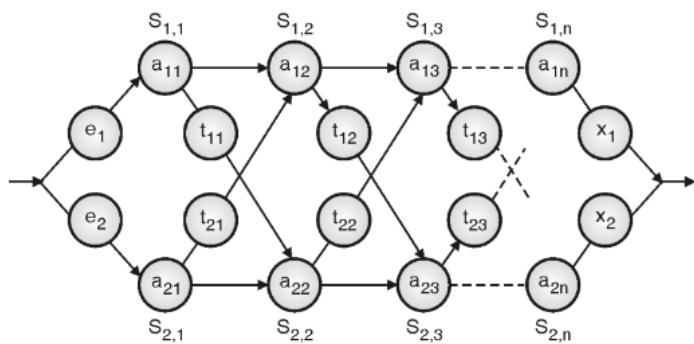
- Manufacturing of large items like car, trucks etc. generally undergoes through multiple stations, where each station is responsible for assembling particular part only. Entire product will be ready after it goes through predefined n stations in sequence.
- For example, manufacturing of car may be done in several stages like engine fitting, coloring, light fitting, fixing of controlling system, gates, seats and many other things.
- Particular task is carried out at the station dedicated for that task only. Based on requirement, there may be more than one assembly line.
- In case of two assembly lines, if load at station j of assembly line 1 is very high, then components are transferred to station j of assembly line 2, converse is also true. This helps to speed up the manufacturing process.
- The time to transfer partial product from one station to next station on same assembly line is negligible. During rush, factory manager may transfer partially completed auto from one assembly line to another, to complete the manufacturing as quick as possible. Some penalty of time t occurs when product is transferred from assembly 1 to 2 or 2 to 1.

Assembly Line Scheduling Problem

Determine which station should be selected from assembly line 1 and which to choose from assembly line 2 in order to minimize the total time to build the entire product.

Selecting stations by brute force attack is quite infeasible. If there are n stations, unfortunately there are 2^n possible ways to choose stations. Brute force attack takes $O(2^n)$ time, it is not acceptable when n is large.

General architecture of two assembly lines with n stations is shown in following figure.



Terminologies are explained here :

- S_{ij} = Station j on assembly line i .
- a_{ij} = Time needed to assemble the partial component at station j on assembly line i .
- t_{ij} = Time required to transfer component from one assembly to other from station j to $(j + 1)$.
- e_i = Entry time on assembly line i .
- x_i = Exit time from assembly line i .

Problem statement

The Assembly line is the mechanism used by industries to manufacture products with less human power and faster speed. In the assembly line, raw material is put on the line, and after a few steps, some operations are done on the raw material.

In this problem, we have two assembly lines, and each line has N stations. N stations have their functionality like painting, shape-making etc. Both the lines are identical in terms of work except for the time used.

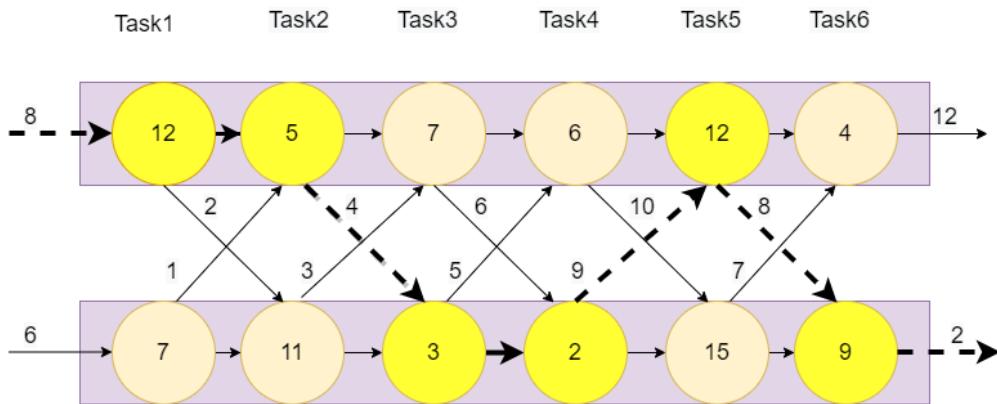
There are e_1 and e_2 as the entry time requires entering into assembly line 1 and assembly line 2, respectively.

There are x_1 and x_2 as exit times for the respective lines.

If we are at any particular station, we will add the time of that specific station, and we will move to the next station. We have two options for the next station, either we can go to the next station of the same line, or we can go to the next station of another line. If we choose to go to the next station of another line, we will have to add the transition time to move from the assembly line.

In the end, we have to calculate the minimum time required to exit a product from the assembly line.

For example:



We have two assembly lines, and let's suppose we decided to choose the highlighted path. The time taken will be:

$$8+12+5+4+3+2+10+12+8+9+2 = 75 \text{ units.}$$

We have to choose the path which will have the minimum time.

Program to implement Assembly Line Scheduling

```

// Assembly Line Scheduling - Dynamic Programming
// program to find minimum possible time by the car chassis to complete
#include <stdio.h>
#define NUM_LINE 2
#define NUM_STATION 4

// Utility function to find minimum of two numbers
int min(int a, int b) { return a < b ? a : b; }

int carAssembly(int a[][NUM_STATION], int t[][NUM_STATION], int *e, int *x)
{
    int T1[NUM_STATION], T2[NUM_STATION], i;

    T1[0] = e[0] + a[0][0]; // time taken to leave first station in line 1
    T2[0] = e[1] + a[1][0]; // time taken to leave first station in line 2

    // Fill tables T1[] and T2[] using the above given recursive relations
    for (i = 1; i < NUM_STATION; ++i)
    {
        T1[i] = min(T1[i-1] + a[0][i], T2[i-1] + t[1][i] + a[0][i]);
        T2[i] = min(T2[i-1] + a[1][i], T1[i-1] + t[0][i] + a[1][i]);
    }

    // Consider exit times and return minimum
    return min(T1[NUM_STATION-1] + x[0], T2[NUM_STATION-1] + x[1]);
}

int main()
{
    int a[][NUM_STATION] = {{4, 5, 3, 2},
                           {2, 10, 1, 4}};
    int t[][NUM_STATION] = {{0, 7, 4, 5},
                           {0, 9, 2, 8}};
    int e[] = {10, 12}, x[] = {18, 7};

    printf("%d", carAssembly(a, t, e, x));

    return 0;
}

```

Output

35

Watch Videos :-

[Assembly Line Scheduling](#)

Flow Shop Scheduling

Flow-shop scheduling is an optimization problem in computer science and operations research. It is a variant of optimal job scheduling.

In a general job-scheduling problem, we are given n jobs J_1, J_2, \dots, J_n of varying processing times, which need to be scheduled on m machines with varying processing power, while trying to minimize the makespan – the total length of the schedule (that is, when all the jobs have finished processing). In the specific variant known as flow-shop scheduling, each job contains exactly m operations. The i -th operation of the job must be executed on the i -th machine. No machine can perform more than one operation simultaneously. For each operation of each job, execution time is specified.

Flow-shop scheduling is a special case of job-shop scheduling where there is strict order of all operations to be performed on all jobs. Flow-shop scheduling may apply as well to production facilities as to computing designs. A special type of flow-shop scheduling problem is the permutation flow-shop scheduling problem in which the processing order of the jobs on the resources is the same for each subsequent step of processing.

In the standard three-field notation for optimal-job-scheduling problems, the flow-shop variant is denoted by F in the first field. For example, the problem denoted by " $F3|p_{ij}|C_{\max}$ " is a 3-machines flow-shop problem with unit processing times, where the goal is to minimize the maximum completion time.

Flow shop scheduling problem:

- o In flow shop, m different machines should process n jobs. Each job contains exactly n operations. The i^{th} operation of the job must be executed on the i^{th} machine. Operations within one job must be performed in the specified order.
- o The first operation gets executed on the first machine, then the second operation on the second machine, and so on. Jobs can be executed in any order. The problem is to determine the optimal such arrangement, i.e. the one with the shortest possible total job execution makespan.
- o With two machines, problem can be solved in $O(n \log n)$ time using Johnson's algorithm. For more than 2 machines, the problem is NP hard. The goal is to minimize the sum of completion time of all jobs.
- o The flow shop contains n jobs simultaneously available at time zero and to be processed by two machines arranged in series with unlimited storage in between them. The processing time of all jobs are known with certainty.
- o It is required to schedule n jobs on machines so as to minimize makespan. The Johnson's rule for scheduling jobs in two machine flow shop is given below: In an optimal schedule, job i precedes job j if $\min\{p_{i1}, p_{j2}\} < \min\{p_{j1}, p_{i2}\}$. Where as, p_{i1} is the processing time of job i on machine 1 and p_{i2} is the processing time of job i on machine 2. Similarly, p_{j1} and p_{j2} are processing times of job j on machine 1 and machine 2 respectively.
- o We can find the optimal scheduling using dynamic programming.

Algorithm for Flow Shop Scheduling

Johnson's algorithm for flow shop scheduling is described below :

```

Algorithm JOHNSON_FLOWSHOP(T, Q)
// T is array of time of jobs, each column indicating time on machine Mi
// Q is queue of jobs
Q = Φ
for j = 1 to n do
    t = minimum machine time scanning in both columns
    if t occurs in column 1 then
        Add Job j to the first empty slot of Q
    else
        Add Job j to last empty slot of Q
    end
    Remove processed job from consideration
end
return Q

```

Example

Each of five jobs needs to go through machines M1 and M2. Find the optimum sequence of jobs using Johnson's rule.

	M ₁	M ₂
A	4	2
B	5	6
C	9	8
D	7	1
E	3	11

Solution: The smallest job is D, which is on machine M1, so schedule this job last. And skip that job from further consideration.

X	X	X	X	D
---	---	---	---	---

Next smallest job is A, which is on machine M2, so schedule the job in last possible empty slot. And skip that job from further consideration.

X	X	X	A	D
---	---	---	---	---

Next smallest job is E, which is on machine M1, so schedule the job in earliest possible empty slot. And skip that job from further consideration.

E	X	X	A	D
---	---	---	---	---

Next smallest job is B, which is on machine M1, so schedule the job in earliest possible empty slot. And skip that job from further consideration.

E	B	X	A	D
---	---	---	---	---

The only job left is C.

E	B	C	A	D
---	---	---	---	---

So final schedule is {E, B, C, A, D}.

Watch Videos :-

[Flow Shop Scheduling - Non-Preemptive Method](#)

[Flow shop scheduling - Pre-emptive Method](#)

10. Other Algorithms

Backtracking Algorithm

A backtracking algorithm is a problem-solving algorithm that uses a brute force approach for finding the desired output.

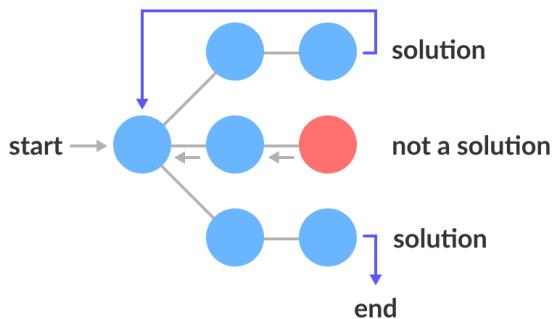
The Brute force approach tries out all the possible solutions and chooses the desired/best solutions.

The term backtracking suggests that if the current solution is not suitable, then backtrack and try other solutions. Thus, recursion is used in this approach.

This approach is used to solve problems that have multiple solutions. If you want an optimal solution, you must go for dynamic programming.

State Space Tree

A space state tree is a tree representing all the possible states (solution or nonsolution) of the problem from the root as an initial state to the leaf as a terminal state.



Backtracking Algorithm

```

Backtrack(x)
    if x is not a solution
        return false
    if x is a new solution
        add to list of solutions
    backtrack(expand x)
    
```

Example Backtracking Approach

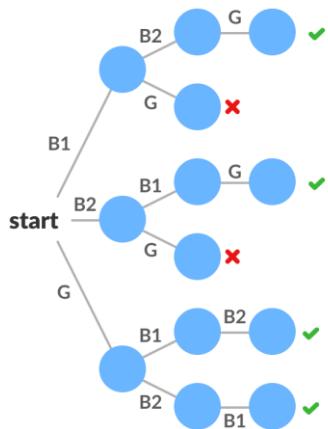
Problem: You want to find all the possible ways of arranging 2 boys and 1 girl on 3 benches.
Constraint: Girl should not be on the middle bench.

Solution: There are a total of $3! = 6$ possibilities. We will try all the possibilities and get the possible solutions. We recursively try all the possibilities.

All the possibilities are:

B1	B2	G	B2	G	B1
B1	G	B2	G	B1	B2
B2	B1	G	G	B2	B1

The following state space tree shows the possible solutions.



Backtracking Algorithm Applications

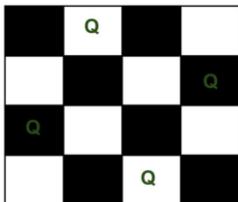
- o To find all Hamiltonian Paths present in a graph.
- o To solve the N Queen problem.
- o Maze solving problem.
- o The Knight's tour problem.

Watch Videos: -

[Introduction to Backtracking - Brute Force Approach](#)

N Queen problem

The N Queen is the problem of placing N chess queens on an NxN chessboard so that no two queens attack each other. For example, the following is a solution for the 4 Queen problem.



The expected output is a binary matrix that has 1s for the blocks where queens are placed. For example, the following is the output matrix for the above 4 queen solution.

```
{ 0, 1, 0, 0}
{ 0, 0, 0, 1}
{ 1, 0, 0, 0}
{ 0, 0, 1, 0}
```

Naive Algorithm

Generate all possible configurations of queens on board and print a configuration that satisfies the given constraints.

```
while there are untried configurations
{
    generate the next configuration
    if queens don't attack in this configuration then
    {
        print this configuration;
    }
}
```

Backtracking Algorithm

The idea is to place queens one by one in different columns, starting from the leftmost column. When we place a queen in a column, we check for clashes with already placed queens. In the current column, if we find a row for which there is no clash, we mark this row and column as part of the solution. If we do not find such a row due to clashes, then we backtrack and return false.

- 1) Start in the leftmost column
- 2) If all queens are placed
 return true
- 3) Try all rows in the current column.
 Do following for every tried row.
 - a) If the queen can be placed safely in this row
 then mark this [row, column] as part of the
 solution and recursively check if placing
 queen here leads to a solution.
 - b) If placing the queen in [row, column] leads to
 a solution then return true.
 - c) If placing queen doesn't lead to a solution then
 unmark this [row, column] (Backtrack) and go to
 step (a) to try other rows.
- 4) If all rows have been tried and nothing worked,
 return false to trigger backtracking.

N-Queens Problem

N - Queens problem is to place n - queens in such a manner on an n x n chessboard that no queens attack each other by being in the same row, column or diagonal.

It can be seen that for n =1, the problem has a trivial solution, and no solution exists for n =2 and n =3. So first we will consider the 4 queens problem and then generate it to n - queens problem. Given a 4 x 4 chessboard and number the rows and column of the chessboard 1 through 4.

	1	2	3	4
1				
2				
3				
4				

4x4 chessboard

Since, we have to place 4 queens such as q1 q2 q3 and q4 on the chessboard, such that no two queens attack each other. In such a conditional each queen must be placed on a different row, i.e., we put queen "i" on row "i."

Now, we place queen q1 in the very first acceptable position (1, 1). Next, we put queen q2 so that both these queens do not attack each other. We find that if we place q2 in column 1 and 2, then the dead end is encountered. Thus the first acceptable position for q2 in column 3, i.e. (2, 3) but then no position is left for placing queen 'q3' safely. So we backtrack one step and place the queen 'q2' in (2, 4), the next best possible solution. Then we obtain the position for placing 'q3' which is (3, 2). But later this position also leads to a dead end, and no place is found where 'q4' can be placed safely. Then we have to backtrack till 'q1' and place it to (1, 2) and then all other queens are placed safely by moving q2 to (2, 4), q3 to (3, 1) and q4 to (4, 3). That is, we get the solution (2, 4, 1, 3). This is one possible solution for the 4-queens problem. For another possible solution, the whole method is repeated for all partial solutions. The other solutions for 4 - queens problems is (3, 1, 4, 2) i.e.

	1	2	3	4
1			q ₁	
2	q ₂			
3				q ₃
4		q ₄		

Program to implement N-Queens Problem

```

/* N Queen Problem using backtracking */
#define N 4
#include <stdbool.h>
#include <stdio.h>

/* A utility function to print solution */
void printSolution(int board[N][N])
{
    for (int i = 0; i < N; i++) {
        for (int j = 0; j < N; j++) {
            printf(" %d ", board[i][j]);
        }
        printf("\n");
    }
}

/* A utility function to check if a queen can be placed on board[row][col]. Note
that this function is called when "col" queens are already placed in columns from
to col -1. So we need to check only left side for attacking queens */
bool isSafe(int board[N][N], int row, int col)
{
    int i, j;

    /* Check this row on left side */
    for (i = 0; i < col; i++)
        if (board[row][i])
            return false;

    /* Check upper diagonal on left side */
    for (i = row, j = col; i >= 0 && j >= 0; i--, j--)
        if (board[i][j])
            return false;

    /* Check lower diagonal on left side */
    for (i = row, j = col; j >= 0 && i < N; i++, j--)
        if (board[i][j])
            return false;

    return true;
}

/* A recursive utility function to solve N Queen problem */
bool solveNQUtil(int board[N][N], int col)
{
    /* base case: If all queens are placed
then return true */
    if (col >= N)
        return true;

    /* Consider this column and try placing this queen in all rows one by one */
    for (int i = 0; i < N; i++) {
        /* Check if the queen can be placed on
        board[i][col] */
        if (isSafe(board, i, col)) {
            /* Place this queen in board[i][col] */
            board[i][col] = 1;

            /* recur to place rest of the queens */
            if (solveNQUtil(board, col + 1))

```

```

return true;

/* If placing queen in board[i][col] doesn't lead to a solution, then
remove queen from board[i][col] */
board[i][col] = 0; // BACKTRACK
}

/*
/* If the queen cannot be placed in any row in this column col then return false
*/
return false;
}

/* This function solves the N Queen problem using Backtracking. It mainly uses
solveNQUtil() to solve the problem. It returns false if queens
cannot be placed, otherwise, return true and prints placement of queens in the form
of 1s. Please note that there may be more than one solutions, this function prints
one of the feasible solutions.*/
bool solveNQ()
{
    int board[N][N] = { { 0, 0, 0, 0 },
                        { 0, 0, 0, 0 },
                        { 0, 0, 0, 0 },
                        { 0, 0, 0, 0 } };

    if (solveNQUtil(board, 0) == false) {
        printf("Solution does not exist");
        return false;
    }

    printSolution(board);
    return true;
}

// driver program to test above function
int main()
{
    solveNQ();
    return 0;
}

```

Output

```

0 0 1 0
1 0 0 0
0 0 0 1
0 1 0 0

```

Watch Videos: -

[N Queens Problem using Backtracking](#)

Sum Of Subsets Problem

Subset sum problem is to find subset of elements that are selected from a given set whose sum adds up to a given number K. We are considering the set contains non-negative values. It is assumed that the input set is unique (no duplicates are presented).

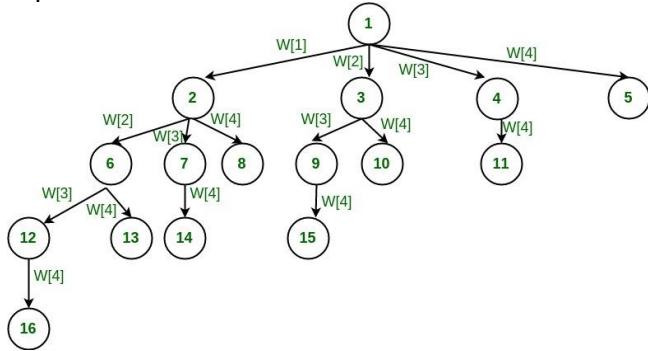
Exhaustive Search Algorithm for Subset Sum

One way to find subsets that sum to K is to consider all possible subsets. A power set contains all those subsets generated from a given set. The size of such a power set is 2^N .

Backtracking Algorithm for Subset Sum

Using exhaustive search we consider all subsets irrespective of whether they satisfy given constraints or not. Backtracking can be used to make a systematic consideration of the elements to be selected.

Assume given set of 4 elements, say $w[1] \dots w[4]$. Tree diagrams can be used to design backtracking algorithms. The following tree diagram depicts approach of generating variable sized tuple.



In the above tree, a node represents function call and a branch represents candidate element. The root node contains 4 children. In other words, root considers every element of the set as different branch. The next level sub-trees correspond to the subsets that includes the parent node. The branches at each level represent tuple element to be considered. For example, if we are at level 1, `tuple_vector[1]` can take any value of four branches generated. If we are at level 2 of left most node, `tuple_vector[2]` can take any value of three branches generated, and so on...

For example the left most child of root generates all those subsets that include $w[1]$. Similarly the second child of root generates all those subsets that includes $w[2]$ and excludes $w[1]$.

As we go down along depth of tree we add elements so far, and if the added sum is satisfying explicit constraints, we will continue to generate child nodes further. Whenever the constraints are not met, we stop further generation of sub-trees of that node, and backtrack to previous node to explore the nodes not yet explored. In many scenarios, it saves considerable amount of processing time.

The tree should trigger a clue to implement the backtracking algorithm (try yourself). It prints all those subsets whose sum add up to given number. We need to explore the nodes along the breadth and depth of the tree. Generating nodes along breadth is controlled by loop and nodes along the depth are generated using recursion (post order traversal). Pseudo code given below,

```

if(subset is satisfying the constraint)
    print the subset
    exclude the current element and consider next element
else
    generate the nodes of present level along breadth of tree and
    recur for next levels

```

Following is the implementation of subset sum using variable size tuple vector. Note that the following program explores all possibilities similar to exhaustive search. It is to demonstrate how backtracking can be used. See next code to verify, how we can optimize the backtracking solution.

The power of backtracking appears when we combine explicit and implicit constraints, and we stop generating nodes when these checks fail. We can improve the above algorithm by strengthening the constraint checks and presorting the data. By sorting the initial array, we need not to consider rest of the array, once the sum so far is greater than target number. We can backtrack and check other possibilities.

Similarly, assume the array is presorted and we found one subset. We can generate next node excluding the present node only when inclusion of next node satisfies the constraints. Given below is optimized implementation (it prunes the subtree if it is not satisfying constraints).

Example

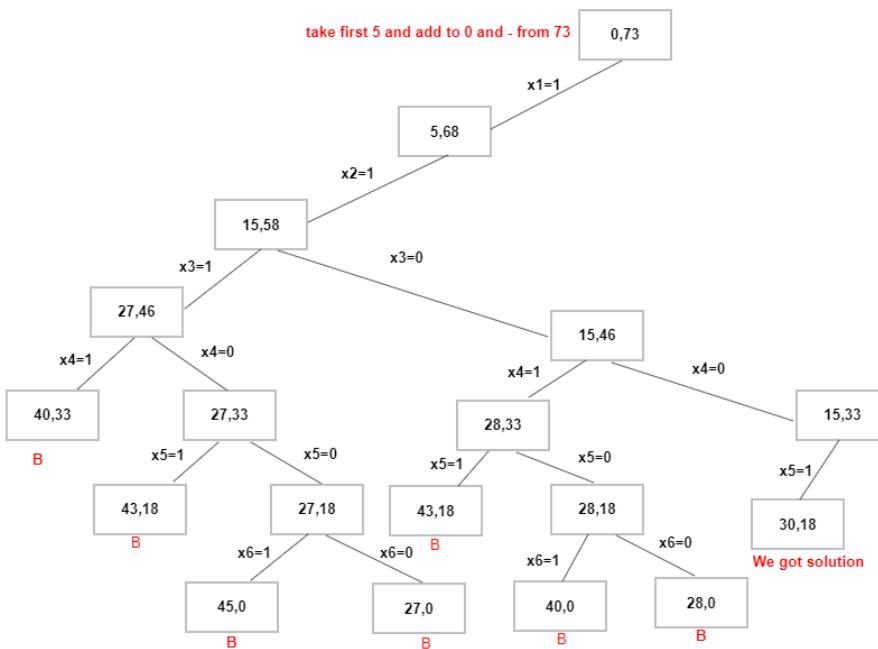
```

n=6 , m=30
w[1:6] ={5,10,12,13,15,18}
Total number of element = 73 (5+10+12+13+15+18)

```

Bounding Condition

$$\sum_{i=0}^k w_i x_i + w_{k+1} \leq M \quad \sum_{i=0}^k w_i x_i + \sum_{i=k+1}^n w_i > M$$



1	1	0	0	1	0
----------	----------	----------	----------	----------	----------

Program to implement Sum Of Subsets Problem

```

#include <stdio.h>
#include <stdlib.h>

#define ARRSIZE(a) (sizeof(a)) / (sizeof(a[0]))

static int total_nodes;

// prints subset found
void printSubset(int A[], int size)
{
    for(int i = 0; i < size; i++)
    {
        printf("%*d", 5, A[i]);
    }
    printf("\n");
}

// qsort compare function
int comparator(const void *pLhs, const void *pRhs)
{
    int *lhs = (int *)pLhs;
    int *rhs = (int *)pRhs;
    return *lhs > *rhs;
}

// inputs
// s      - set vector , t      - tuplet vector
// s_size - set size , t_size  - tuplet size so far
// sum    - sum so far , ite   - nodes count
// target_sum - sum to be found
void subset_sum(int s[], int t[],
                int s_size, int t_size,
                int sum, int ite,
                int const target_sum)
{
    total_nodes++;

    if( target_sum == sum )
    {
        // We found sum
        printSubset(t, t_size);

        // constraint check
        if( ite + 1 < s_size && sum - s[ite] + s[ite+1] <= target_sum )
        {
            // Exclude previous added item and consider next candidate
            subset_sum(s, t, s_size, t_size-1, sum - s[ite], ite + 1, target_sum);
        }
        return;
    }
    else
    {
        // constraint check
        if( ite < s_size && sum + s[ite] <= target_sum )
        {
            // generate nodes along the breadth
            for( int i = ite; i < s_size; i++ )
            {

```

```

t[t_size] = s[i];

if( sum + s[i] <= target_sum )
{
    // consider next level node (along depth)
    subset_sum(s, t, s_size, t_size + 1, sum + s[i], i + 1, target_sum);
}
}

}

// Wrapper that prints subsets that sum to target sum
void generateSubsets(int s[], int size, int target_sum)
{
    int *tuplet_vector = (int *)malloc(size * sizeof(int));

    int total = 0;

    // sort the set
    qsort(s, size, sizeof(int), &comparator);

    for( int i = 0; i < size; i++ )
    {
        total += s[i];
    }

    if( s[0] <= target_sum && total >= target_sum )
    {

        subset_sum(s, tuplet_vector, size, 0, 0, 0, target_sum);
    }
}

free(tuplet_vector);
}

int main()
{
    int weights[] = {15, 22, 14, 26, 32, 9, 16, 8};
    int target = 53;
    int size = ARRSIZE(weights);
    generateSubsets(weights, size, target);
    printf("Nodes generated %d\n", total_nodes);
    return 0;
}

```

Output

```

8      9      14      22
8      14      15      16
15      16      22
Nodes generated 68

```

Watch Videos: -

[Sum Of Subsets Problem - Backtracking](#)
[Subset Sum Problem using Dynamic Programming](#)

Graph Coloring Problem

Graph coloring refers to the problem of coloring vertices of a graph in such a way that no two adjacent vertices have the same color. This is also called the vertex coloring problem.

If coloring is done using at most k colors, it is called k -coloring.

The smallest number of colors required for coloring graph is called its chromatic number. The chromatic number is denoted by $\chi(G)$. Finding the chromatic number for the graph is NP-complete problem.

Graph coloring problem is both, decision problem as well as an optimization problem. A decision problem is stated as, "With given M colors and graph G , whether such color scheme is possible or not?".

The optimization problem is stated as, "Given M colors and graph G , find the minimum number of colors required for graph coloring."

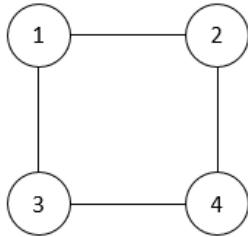
Graph coloring problem is a very interesting problem of graph theory and it has many diverse applications. Few of them are listed below.

Applications of Graph Coloring Problem

- o Design a timetable.
- o Sudoku
- o Register allocation in the compiler
- o Map coloring
- o Mobile radio frequency assignment:

Examples

Apply backtrack on the following instance of graph coloring problem of 4 nodes and 3 colors

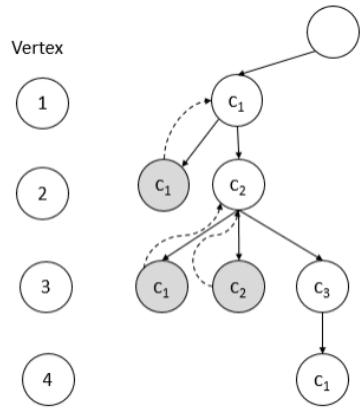


This problem can be solved using backtracking algorithms. The formal idea is to list down all the vertices and colors in two lists. Assign color 1 to vertex 1.

If vertex 2 is not adjacent to vertex 1 then assign the same color, otherwise assign color 2.

The process is repeated until all vertices are colored.

The algorithm backtracks whenever color i is not possible to assign to any vertex k and it selects the next color $i + 1$ and the test is repeated. This graph can be colored with 3 colors.



Vertex	Assigned color
1	C1
2	C2
3	C3
4	C1

Program to implement Graph Coloring - M Coloring

```

// M Coloring problem using backtracking
#include <stdbool.h>
#include <stdio.h>

// Number of vertices in the graph
#define V 4

void printSolution(int color[]);

/* A utility function to check if the current color assignment is safe for vertex v
i.e. checks whether the edge exists or not (i.e., graph[v][i]==1). If exist then
checks whether the color to be filled in the new vertex(c is sent in the parameter)
is already used by its adjacent vertices(i-->adj vertices) or not (i.e., color[i]==c)
*/
bool isSafe(int v, bool graph[V][V], int color[], int c)
{
    for (int i = 0; i < V; i++)
        if (graph[v][i] && c == color[i])
            return false;
    return true;
}

/* A recursive utility function to solve m coloring problem */
bool graphColoringUtil(bool graph[V][V], int m, int color[],
                      int v)
{
    /* base case: If all vertices are assigned a color then return true */
    if (v == V)
        return true;

    /* Consider this vertex v and try different colors */
    for (int c = 1; c <= m; c++) {
        /* Check if assignment of color c to v is fine*/
        if (isSafe(v, graph, color, c)) {
            color[v] = c;

            /* recur to assign colors to rest of the vertices */
            if (graphColoringUtil(graph, m, color, v + 1)
                == true)
                return true;

            /* If assigning color c doesn't lead to a solution then remove it */
            color[v] = 0;
        }
    }

    /* If no color can be assigned to this vertex then return false */
    return false;
}

/* This function solves the m Coloring problem using Backtracking. It mainly
uses graphColoringUtil() to solve the problem. It returns false if the m
colors cannot be assigned, otherwise return true and prints assignments of
colors to all vertices. Please note that there may be more than one solutions,
this function prints one of the feasible solutions.*/
bool graphColoring(bool graph[V][V], int m)
{
    // Initialize all color values as 0.

```

```

// This initialization is needed
// correct functioning of isSafe()
int color[V];
for (int i = 0; i < V; i++)
    color[i] = 0;

// Call graphColoringUtil() for vertex 0
if (graphColoringUtil(graph, m, color, 0) == false) {
    printf("Solution does not exist");
    return false;
}

// Print the solution
printSolution(color);
return true;
}

/* A utility function to print solution */
void printSolution(int color[])
{
    printf("Solution Exists:");
    " Following are the assigned colors \n";
    for (int i = 0; i < V; i++)
        printf(" %d ", color[i]);
    printf("\n");
}

int main()
{
    /* Create following graph and test whether it is 3 colorable
    (3)---(2)
    | / |
    | / |
    | / |
    (0)---(1)
    */
    bool graph[V][V] = {
        { 0, 1, 1, 1 },
        { 1, 0, 1, 0 },
        { 1, 1, 0, 1 },
        { 1, 0, 1, 0 },
    };
    int m = 3; // Number of colors

    // Function call
    graphColoring(graph, m);
    return 0;
}

```

Output

Solution Exists: Following are the assigned colors

1 2 3 2

Watch Videos: -

[Graph Coloring Problem - Backtracking](#)

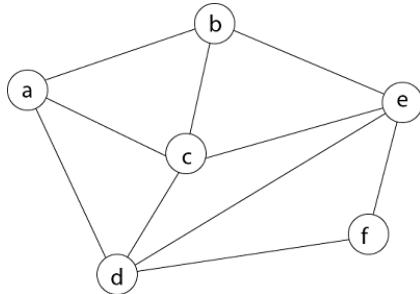
Hamiltonian Paths / Cycle

Hamiltonian Path in an undirected graph is a path that visits each vertex exactly once. A Hamiltonian cycle (or Hamiltonian circuit) is a Hamiltonian Path such that there is an edge (in the graph) from the last vertex to the first vertex of the Hamiltonian Path. Determine whether a given graph contains Hamiltonian Cycle or not. If it contains, then prints the path. Following are the input and output of the required function.

Given a graph $G = (V, E)$ we have to find the Hamiltonian Circuit using Backtracking approach.

We start our search from any arbitrary vertex say 'a.' This vertex 'a' becomes the root of our implicit tree. The first element of our partial solution is the first intermediate vertex of the Hamiltonian Cycle that is to be constructed. The next adjacent vertex is selected by alphabetical order. If at any stage any arbitrary vertex makes a cycle with any vertex other than vertex 'a' then we say that **dead end** is reached. In this case, we backtrack one step, and again the search begins by selecting another vertex and backtrack the element from the partial; solution must be removed. The search using backtracking is successful if a Hamiltonian Cycle is obtained.

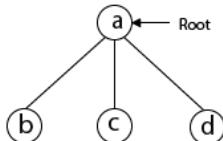
Example: Consider a graph $G = (V, E)$ shown in fig. we have to find a Hamiltonian circuit using Backtracking method.



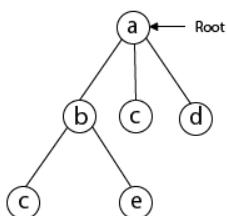
Solution: Firstly, we start our search with vertex 'a.' this vertex 'a' becomes the root of our implicit tree.



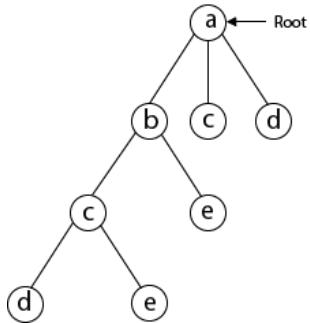
Next, we choose vertex 'b' adjacent to 'a' as it comes first in lexicographical order (b, c, d).



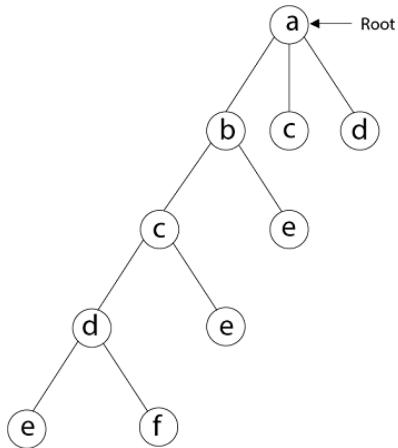
Next, we select 'c' adjacent to 'b.'



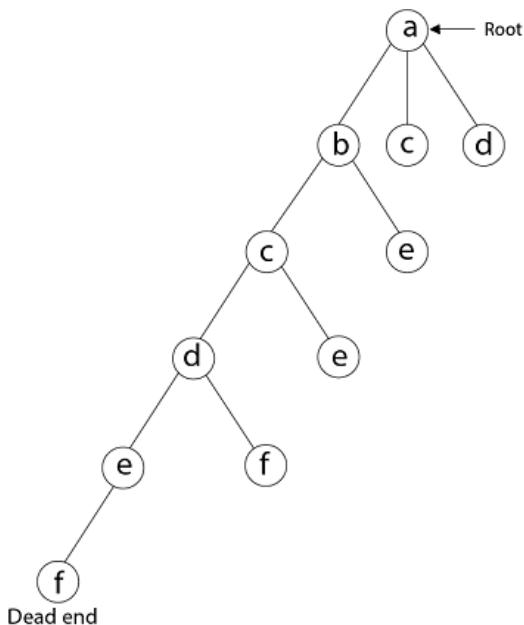
Next, we select 'd' adjacent to 'c.'



Next, we select 'e' adjacent to 'd.'

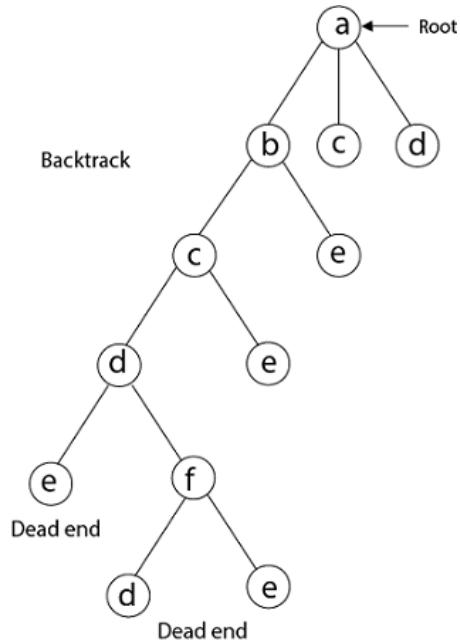
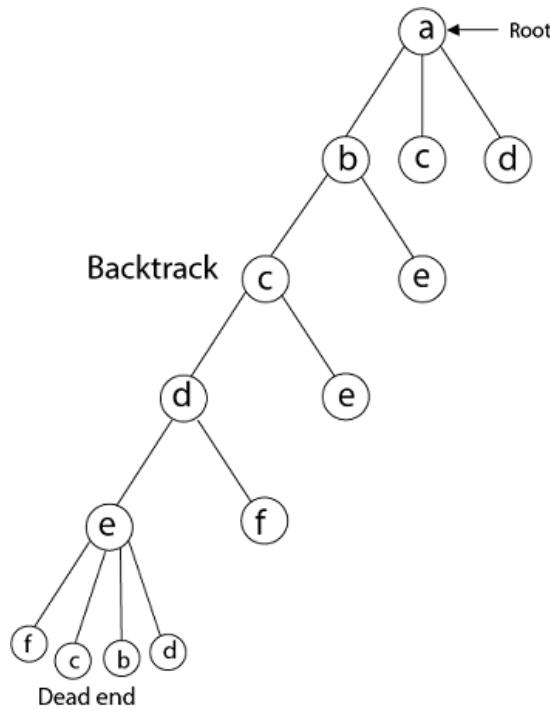


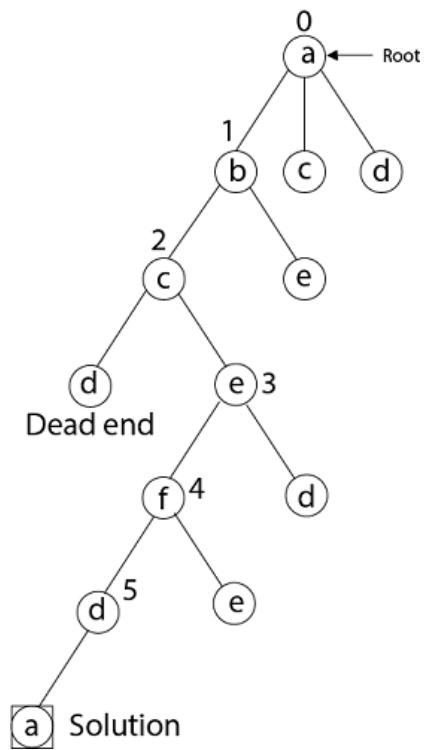
Next, we select vertex 'f' adjacent to 'e.' The vertex adjacent to 'f' is d and e, but they have already visited. Thus, we get the dead end, and we backtrack one step and remove the vertex 'f' from partial solution.



From backtracking, the vertex adjacent to 'e' is b, c, d, and f from which vertex 'f' has already been checked, and b, c, d have already visited. So, again we backtrack one step. Now, the vertex adjacent to d are e, f from which e has already been checked, and adjacent of 'f' are d and e. If 'e' vertex, revisited them we get a dead state. So again we backtrack one step.

Now, adjacent to c is 'e' and adjacent to 'e' is 'f' and adjacent to 'f' is 'd' and adjacent to 'd' is 'a.' Here, we get the Hamiltonian Cycle as all the vertex other than the start vertex 'a' is visited only once. (a - b - c - e - f - d - a).



Again Backtrack

Program to implement Hamiltonian Cycle

```

/* Hamiltonian Cycle problem using backtracking */
#include<stdio.h>
#include<stdbool.h>
#include<malloc.h>

// Number of vertices in the graph
#define V 5

void printSolution(int path[]);

/* A utility function to check if the vertex v can be added at index 'pos' in the
Hamiltonian Cycle constructed so far
(stored in 'path[]') */
bool isSafe(int v, bool graph[V][V], int path[], int pos)
{
    /* Check if this vertex is an adjacent vertex of the previously added vertex. */
    if (graph [ path[pos-1] ][ v ] == 0)
        return false;

    /* Check if the vertex has already been included. This step can be optimized by
creating an array of size V */
    for (int i = 0; i < pos; i++)
        if (path[i] == v)
            return false;

    return true;
}

/* A recursive utility function to solve hamiltonian cycle problem */
bool hamCycleUtil(bool graph[V][V], int path[], int pos)
{
    /* base case: If all vertices are included in Hamiltonian Cycle */
    if (pos == V)
    {
        // And if there is an edge from the last included vertex to the first vertex
        if (graph[ path[pos-1] ][ path[0] ] == 1)
            return true;
        else
            return false;
    }

    // Try different vertices as a next candidate in Hamiltonian Cycle.
    // We don't try for 0 as we included 0 as starting point in hamCycle()
    for (int v = 1; v < V; v++)
    {
        /* Check if this vertex can be added to Hamiltonian Cycle */
        if (isSafe(v, graph, path, pos))
        {
            path[pos] = v;

            /* recur to construct rest of the path */
            if (hamCycleUtil (graph, path, pos+1) == true)
                return true;

            /* If adding vertex v doesn't lead to a solution,
            then remove it */
            path[pos] = -1;
        }
    }
}

```

```

}

/* If no vertex can be added to Hamiltonian Cycle constructed so far,
then return false */
return false;
}

/* This function solves the Hamiltonian Cycle problem using Backtracking. It mainly
uses hamCycleUtil() to solve the
problem. It returns false if there is no Hamiltonian Cycle possible, otherwise
return true and prints the path.
Please note that there may be more than one solutions, this function prints one of
the feasible solutions.*/
bool hamCycle(bool graph[V][V])
{
//    int *path = int [V];
int *path = malloc(V*sizeof(int));

for (int i = 0; i < V; i++)
    path[i] = -1;

/* Let us put vertex 0 as the first vertex in the path. If there is a Hamiltonian
Cycle, then the path can be
started from any point of the cycle as the graph is undirected */
path[0] = 0;
if ( hamCycleUtil(graph, path, 1) == false )
{
    printf("\nSolution does not exist");
    return false;
}

printSolution(path);
return true;
}

/* A utility function to print solution */
void printSolution(int path[])
{
    printf ("Solution Exists:\n"
            " Following is one Hamiltonian Cycle \n");
    for (int i = 0; i < V; i++)
        printf(" %d ", path[i]);

    // Let us print the first vertex again to show the complete cycle
    printf(" %d ", path[0]);
    printf("\n");
}

int main()
{
    /* Let us create the following graph
    (0)--(1)--(2)
    | / \ |
    | / \ |
    | /     \ |
    (3)-----(4) */
    bool graph1[V][V] = {{0, 1, 1, 0, 1},
                          {1, 0, 1, 1, 1},
                          {1, 1, 0, 1, 0},
                          {0, 1, 1, 0, 1},

```

```
    {1, 1, 0, 1, 0},  
};  
  
// Print the solution  
hamCycle(graph1);  
return 0;  
}  
  
Output  
  
Solution Exists: Following is one Hamiltonian Cycle  
0 1 2 3 4 0
```

Watch Videos: -

[Hamiltonian Cycle - Backtracking](#)

Maze Solving Problem.

A Maze is given as $N \times N$ binary matrix of blocks where source block is the upper left most block i.e., $\text{maze}[0][0]$ and destination block is lower rightmost block i.e., $\text{maze}[N-1][N-1]$. A rat starts from source and has to reach the destination. The rat can move only in two directions: forward and down.

In the maze matrix, 0 means the block is a dead end and 1 means the block can be used in the path from source to destination. Note that this is a simple version of the typical Maze problem. For example, a more complex version can be that the rat can move in 4 directions and a more complex version can be with a limited number of moves.

Following is an example maze.

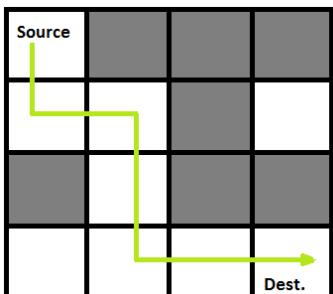
Gray blocks are dead ends (value = 0).



Following is a binary matrix representation of the above maze.

```
{1, 0, 0, 0}
{1, 1, 0, 1}
{0, 1, 0, 0}
{1, 1, 1, 1}
```

Following is a maze with highlighted solution path.



Following is the solution matrix (output of program) for the above input matrix.

```
{1, 0, 0, 0}
{1, 1, 0, 0}
{0, 1, 0, 0}
{0, 1, 1, 1}
```

All entries in solution path are marked as 1.

Backtracking Algorithm

Backtracking is an algorithmic-technique for solving problems recursively by trying to build a solution incrementally. Solving one piece at a time, and removing those solutions that fail to satisfy the constraints of the problem at any point of time (by time, here, is referred to the time elapsed till reaching any level of the search tree) is the process of backtracking.

Approach

Form a recursive function, which will follow a path and check if the path reaches the destination or not. If the path does not reach the destination then backtrack and try other paths.

Algorithm:

1. Create a solution matrix, initially filled with 0's.
2. Create a recursive function, which takes initial matrix, output matrix and position of rat (i, j).
3. if the position is out of the matrix or the position is not valid then return.
4. Mark the position $\text{output}[i][j]$ as 1 and check if the current position is destination or not. If destination is reached print the output matrix and return.
5. Recursively call for position $(i+1, j)$ and $(i, j+1)$.
6. Unmark position (i, j) , i.e $\text{output}[i][j] = 0$.

Program to implement Rat in Maze

```

// Rat in a Maze problem using backtracking
#include <stdio.h>
#include <stdbool.h>
// Maze size
#define N 4

bool solveMazeUtil(int maze[N][N], int x, int y, int sol[N][N]);

// A utility function to print solution matrix sol[N][N]
void printSolution(int sol[N][N])
{
    for (int i = 0; i < N; i++) {
        for (int j = 0; j < N; j++)
            printf(" %d ", sol[i][j]);
        printf("\n");
    }
}

// A utility function to check if x, y is valid index for
// N*N maze
bool isSafe(int maze[N][N], int x, int y)
{
    // if (x, y outside maze) return false
    if (x >= 0 && x < N && y >= 0 && y < N && maze[x][y] == 1)
        return true;
    return false;
}

// This function solves the Maze problem using Backtracking.
// It mainly uses solveMazeUtil() to solve the problem. It
// returns false if no path is possible, otherwise return
// true and prints the path in the form of 1s. Please note
// that there may be more than one solutions, this function
// prints one of the feasible solutions.
bool solveMaze(int maze[N][N])
{
    int sol[N][N] = { { 0, 0, 0, 0 },
                      { 0, 0, 0, 0 },
                      { 0, 0, 0, 0 },
                      { 0, 0, 0, 0 } };
    if (solveMazeUtil(maze, 0, 0, sol) == false) {
        printf("Solution doesn't exist");
        return false;
    }
    printSolution(sol);
    return true;
}

// A recursive utility function to solve Maze problem
bool solveMazeUtil(int maze[N][N], int x, int y, int sol[N][N])
{
    // if (x, y is goal) return true
    if (x == N - 1 && y == N - 1 && maze[x][y] == 1) {
        sol[x][y] = 1;
        return true;
    }
    // Check if maze[x][y] is valid
    if (isSafe(maze, x, y) == true) {

```

```

// Check if the current block is already part of
// solution path.
if (sol[x][y] == 1)
    return false;
// mark x, y as part of solution path
sol[x][y] = 1;
/* Move forward in x direction */
if (solveMazeUtil(maze, x + 1, y, sol) == true)
    return true;
// If moving in x direction doesn't give solution
// then Move down in y direction
if (solveMazeUtil(maze, x, y + 1, sol) == true)
    return true;
// If none of the above movements work then
// BACKTRACK: unmark x, y as part of solution path
sol[x][y] = 0;
return false;
}
return false;
}

// driver program to test above function
int main()
{
    int maze[N][N] = { { 1, 0, 0, 0 },
                        { 1, 1, 0, 1 },
                        { 0, 1, 0, 0 },
                        { 1, 1, 1, 1 } };
    solveMaze(maze);
    return 0;
}

```

Output

```

1 0 0 0
1 1 0 0
0 1 0 0
0 1 1 1

```

Watch Videos: -

[Rat in A Maze Backtracking](#)

The Knight's Tour Problem.

Backtracking is an algorithmic paradigm that tries different solutions until finds a solution that "works". Problems that are typically solved using the backtracking technique have the following property in common. These problems can only be solved by trying every possible configuration and each configuration is tried only once. A Naive solution for these problems is to try all configurations and output a configuration that follows given problem constraints. Backtracking works incrementally and is an optimization over the Naive solution where all possible configurations are generated and tried.

For example, consider the following Knight's Tour problem.

Problem Statement:

Given a $N \times N$ board with the Knight placed on the first block of an empty board. Moving according to the rules of chess knight must visit each square exactly once. Print the order of each cell in which they are visited.

Example:

Input : $N = 8$

Output:

0	59	38	33	30	17	8	63
37	34	31	60	9	62	29	16
58	1	36	39	32	27	18	7
35	48	41	26	61	10	15	28
42	57	2	49	40	23	6	19
47	50	45	54	25	20	11	14
56	43	52	3	22	13	24	5
51	46	55	44	53	4	21	12

The path followed by Knight to cover all the cells

Following is a chessboard with 8×8 cells. Numbers in cells indicate the move number of Knight.

0	59	38	33	30	17	8	63
37	34	31	60	9	62	29	16
58	1	36	39	32	27	18	7
35	48	41	26	61	10	15	28
42	57	2	49	40	23	6	19
47	50	45	54	25	20	11	14
56	43	52	3	22	13	24	5
51	46	55	44	53	4	21	12

Naive Algorithm for Knight's tour

The Naive Algorithm is to generate all tours one by one and check if the generated tour satisfies the constraints.

```

while there are untried tours
{
    generate the next tour
    if this tour covers all squares
    {
        print this path;
    }
}

```

Backtracking works in an incremental way to attack problems. Typically, we start from an empty solution vector and one by one add items (Meaning of item varies from problem to problem. In the context of Knight's tour problem, an item is a Knight's move). When we add an item, we check if adding the current item violates the problem constraint, if it does then we remove the item and try other alternatives. If none of the alternatives works out then we go to the previous stage and remove the item added in the previous stage. If we reach the initial stage back then we say that no solution exists. If adding an item doesn't violate constraints then we recursively add items one by one. If the solution vector becomes complete then we print the solution.

Backtracking Algorithm for Knight's tour

Following is the Backtracking algorithm for Knight's tour problem.

```

If all squares are visited
    print the solution
Else
    a) Add one of the next moves to solution vector and recursively
       check if this move leads to a solution. (A Knight can make maximum
       eight moves. We choose one of the 8 moves in this step).
    b) If the move chosen in the above step doesn't lead to a solution
       then remove this move from the solution vector and try other
       alternative moves.
    c) If none of the alternatives work then return false (Returning false
       will remove the previously added item in recursion and if false is
       returned by the initial call of recursion then "no solution exists" )

```

Following are implementations for Knight's tour problem. It prints one of the possible solutions in 2D matrix form. Basically, the output is a 2D 8*8 matrix with numbers from 0 to 63 and these numbers show steps made by Knight.

Program to implement Knight's Tour

```

// Knight Tour problem
#include <stdio.h>
#define N 8

int solveKTUtil(int x, int y, int movei, int sol[N][N],
                int xMove[], int yMove[]);

/* A utility function to check if i,j are valid indexes for N*N chessboard */
int isSafe(int x, int y, int sol[N][N])
{
    return (x >= 0 && x < N && y >= 0 && y < N
            && sol[x][y] == -1);
}

/* A utility function to print solution matrix sol[N][N] */
void printSolution(int sol[N][N])
{
    for (int x = 0; x < N; x++) {
        for (int y = 0; y < N; y++)
            printf(" %2d ", sol[x][y]);
        printf("\n");
    }
}

/* This function solves the Knight Tour problem using Backtracking. This function
mainly uses solveKTUtil() to solve the problem. It returns false if no complete tour
is possible, otherwise return true and prints the tour. Please note that there may
be more than one solutions, this function prints one of the feasible solutions. */
int solveKT()
{
    int sol[N][N];

    /* Initialization of solution matrix */
    for (int x = 0; x < N; x++)
        for (int y = 0; y < N; y++)
            sol[x][y] = -1;

    /* xMove[] and yMove[] define next move of Knight. xMove[] is for next value of x
    coordinate yMove[] is for next value of y coordinate */
    int xMove[8] = { 2, 1, -1, -2, -2, -1, 1, 2 };
    int yMove[8] = { 1, 2, 2, 1, -1, -2, -2, -1 };

    // Since the Knight is initially at the first block
    sol[0][0] = 0;

    /* Start from 0,0 and explore all tours using
    solveKTUtil() */
    if (solveKTUtil(0, 0, 1, sol, xMove, yMove) == 0) {
        printf("Solution does not exist");
        return 0;
    }
    else
        printSolution(sol);

    return 1;
}

/* A recursive utility function to solve Knight Tour problem */

```

```

int solveKTUtil(int x, int y, int movei, int sol[N][N],
                int xMove[N], int yMove[N])
{
    int k, next_x, next_y;
    if (movei == N * N)
        return 1;

    /* Try all next moves from the current coordinate x, y */
    for (k = 0; k < 8; k++) {
        next_x = x + xMove[k];
        next_y = y + yMove[k];
        if (isSafe(next_x, next_y, sol)) {
            sol[next_x][next_y] = movei;
            if (solveKTUtil(next_x, next_y, movei + 1, sol,
                            xMove, yMove)
                == 1)
                return 1;
            else
                sol[next_x][next_y] = -1; // backtracking
        }
    }

    return 0;
}

/* Driver Code */
int main()
{
    solveKT();
    return 0;
}

```

Output

0	59	38	33	30	17	8	63
37	34	31	60	9	62	29	16
58	1	36	39	32	27	18	7
35	48	41	26	61	10	15	28
42	57	2	49	40	23	6	19
47	50	45	54	25	20	11	14
56	43	52	3	22	13	24	5
51	46	55	44	53	4	21	12

Watch Videos: -

[Knight's Tour](#)
[Knight Tour Problem Backtracking](#)

Rabin-Karp Algorithm

Rabin-Karp algorithm is an algorithm used for searching/matching patterns in the text using a hash function. Unlike Naive string matching algorithm, it does not travel through every character in the initial phase rather it filters the characters that do not match and then performs the comparison.

A hash function is a tool to map a larger input value to a smaller output value. This output value is called the hash value.

How Rabin-Karp Algorithm Works?

A sequence of characters is taken and checked for the possibility of the presence of the required string. If the possibility is found then, character matching is performed.

Let us understand the algorithm with the following steps:

1. Let the text be:

A	B	C	C	D	D	A	E	F	G
---	---	---	---	---	---	---	---	---	---

And the string to be searched in the above text be:

C	D	D
---	---	---

2. Let us assign a numerical value(v)/weight for the characters we will be using in the problem. Here, we have taken first ten alphabets only (i.e. A to J).

Text Weights :

A	B	C	D	E	F	G	H	I	J
1	2	3	4	5	6	7	8	9	10

3. n be the length of the pattern and m be the length of the text. Here, $m = 10$ and $n = 3$. Let d be the number of characters in the input set. Here, we have taken input set {A, B, C, ..., J}. So, $d = 10$. You can assume any suitable value for d .

4. Let us calculate the hash value of the pattern (Hash value of text).

$$\begin{aligned}
 \text{hash value for pattern}(p) &= \sum(v * d^{m-1}) \bmod 13 \\
 &= ((3 * 102) + (4 * 101) + (4 * 100)) \bmod 13 \\
 &= 344 \bmod 13 \\
 &= 6
 \end{aligned}$$

In the calculation above, choose a prime number (here, 13) in such a way that we can perform all the calculations with single-precision arithmetic.

The reason for calculating the modulus is given below.

5. Calculate the hash value for the text-window of size m .

For the first window ABC,

```

hash value for text(t) =  $\Sigma(v * dn-1) \bmod 13$ 
                      =  $((1 * 10^2) + (2 * 10^1) + (3 * 10^0)) \bmod 13$ 
                      =  $123 \bmod 13$ 
                      = 6

```

6. Compare the hash value of the pattern with the hash value of the text. If they match then, character-matching is performed.

In the above examples, the hash value of the first window (i.e. t) matches with p so, go for character matching between ABC and CDD. Since they do not match so, go for the next window.

7. We calculate the hash value of the next window by subtracting the first term and adding the next term as shown below.

```

t =  $((1 * 10^2) + ((2 * 10^1) + (3 * 10^0)) * 10 + (3 * 10^0)) \bmod 13$ 
=  $233 \bmod 13$ 
= 12

```

In order to optimize this process, we make use of the previous hash value in the following way.

```

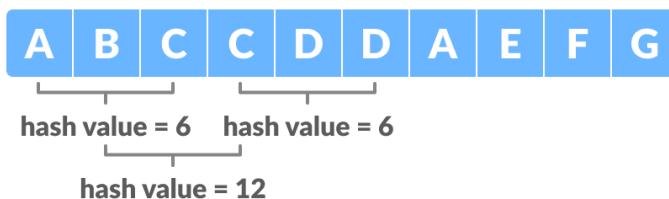
t =  $((d * (t - v[\text{character to be removed}] * h) + v[\text{character to be added}]) \bmod 13$ 
=  $((10 * (6 - 1 * 9) + 3) \bmod 13$ 
= 12
Where,  $h = d^{m-1} = 10^{3-1} = 100$ .

```

8. For BCC, $t = 12 \neq 6$. Therefore, go for the next window.

After a few searches, we will get the match for the window CDA in the text.

Hash value of different windows



Algorithm

```

n = t.length
m = p.length
h = dm-1 mod q
p = 0
t0 = 0
for i = 1 to m
    p = (dp + p[i]) mod q
    t0 = (dt0 + t[i]) mod q
for s = 0 to n - m
    if p = ts
        if p[1.....m] = t[s + 1.....s + m]
            print "pattern found at position" s
    If s < n-m
        ts + 1 = (d (ts - t[s + 1]h) + t[s + m + 1]) mod q

```

Program to implement Rabin-Karp Algorithm

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

#define d 10

void rabinKarp(char pattern[], char text[], int q) {
    int m = strlen(pattern);
    int n = strlen(text);
    int i, j;
    int p = 0;
    int t = 0;
    int h = 1;

    for (i = 0; i < m - 1; i++)
        h = (h * d) % q;

    // Calculate hash value for pattern and text
    for (i = 0; i < m; i++) {
        p = (d * p + pattern[i]) % q;
        t = (d * t + text[i]) % q;
    }

    // Find the match
    for (i = 0; i <= n - m; i++) {
        if (p == t) {
            for (j = 0; j < m; j++) {
                if (text[i + j] != pattern[j])
                    break;
            }
            if (j == m)
                printf("Pattern is found at position: %d \n", i + 1);
        }

        if (i < n - m) {
            t = (d * (t - text[i] * h) + text[i + m]) % q;

            if (t < 0)
                t = (t + q);
        }
    }
}

int main() {
    char text[] = "ABCCDDAEFG";
    char pattern[] = "CDD";
    int q = 13;
    rabinKarp(pattern, text, q);
}
```

Ouput

Pattern is found at position: 4

Watch Videos:-

[Rabin-Karp String Matching Algorithm](#)

Branch and Bound

Branch and bound is one of the techniques used for problem solving. It is similar to the backtracking since it also uses the state space tree. It is used for solving the optimization problems and minimization problems. If we have given a maximization problem then we can convert it using the Branch and bound technique by simply converting the problem into a maximization problem.

Let's understand through an example.

```
Jobs = {j1, j2, j3, j4}
P = {10, 5, 8, 3}
d = {1, 2, 1, 2}
```

The above are jobs, problems and problems given. We can write the solutions in two ways which are given below:

Suppose we want to perform the jobs j1 and j2 then the solution can be represented in two ways:

The first way of representing the solutions is the subsets of jobs.

```
S1 = {j1, j4}
```

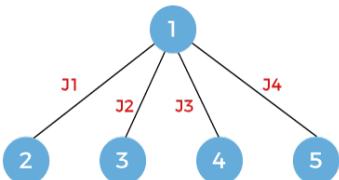
The second way of representing the solution is that first job is done, second and third jobs are not done, and fourth job is done.

```
S2 = {1, 0, 0, 1}
```

The solution s1 is the variable-size solution while the solution s2 is the fixed-size solution.

First, we will see the subset method where we will see the variable size.

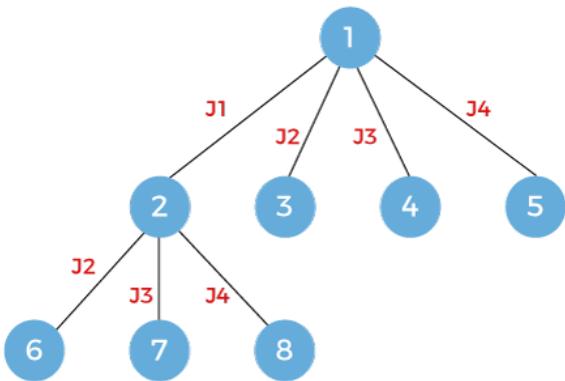
First method:



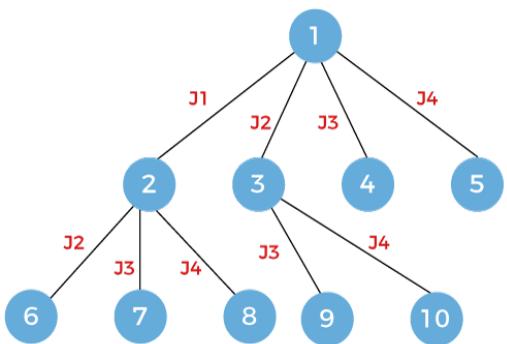
In this case, we first consider the first job, then second job, then third job and finally we consider the last job.

As we can observe in the above figure that the **breadth first search** is performed but not the depth first search. Here we move breadth wise for exploring the solutions. In backtracking, we go depth-wise whereas in branch and bound, we go breadth wise.

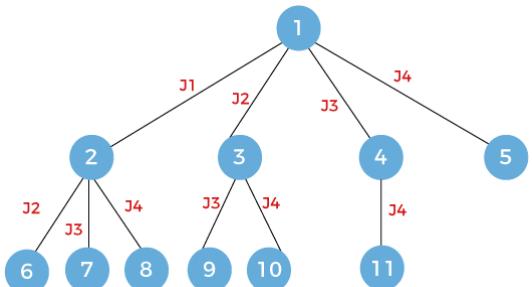
Now one level is completed. Once I take first job, then we can consider either j2, j3 or j4. If we follow the route then it says that we are doing jobs j1 and j4 so we will not consider jobs j2 and j3.



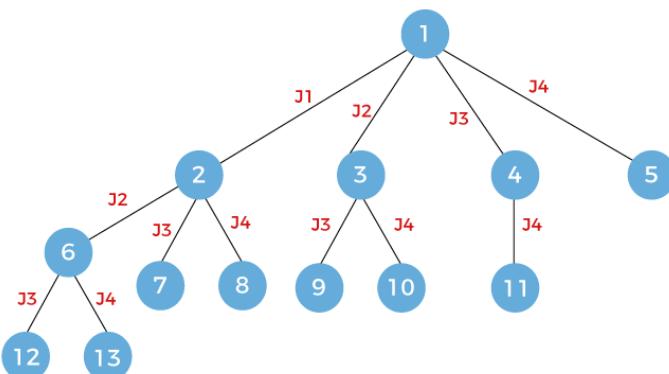
Now we will consider the node 3. In this case, we are doing job j2 so we can consider either job j3 or j4. Here, we have discarded the job j1.



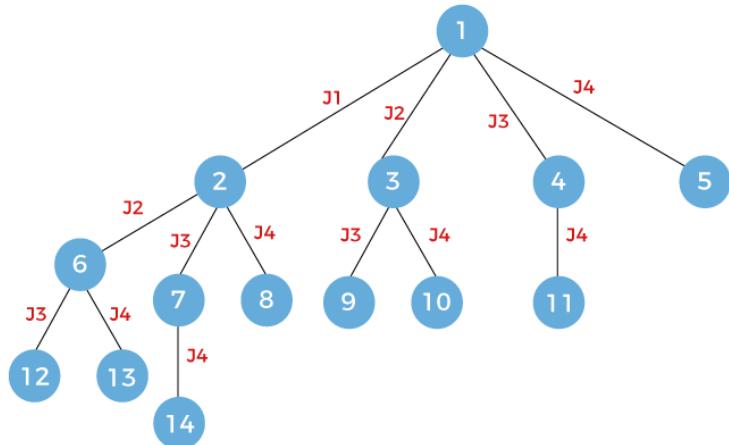
Now we will expand the node 4. Since here we are doing job j3 so we will consider only job j4.



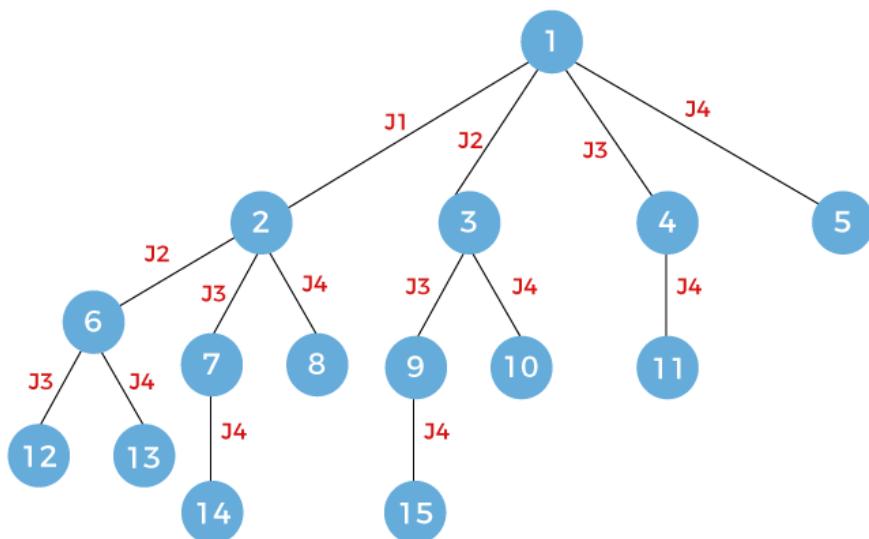
Now we will expand node 6, and here we will consider the jobs j3 and j4.



Now we will expand node 7 and here we will consider job j4.



Now we will expand node 9, and here we will consider job j4.



The last node, i.e., node 12 which is left to be expanded. Here, we consider job j4.

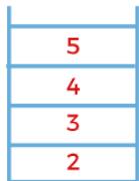
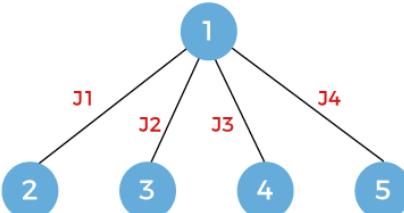
The above is the state space tree for the solution $s1 = \{j1, j4\}$

Second method:

We will see another way to solve the problem to achieve the solution s1.

First, we consider the node 1 shown as below:

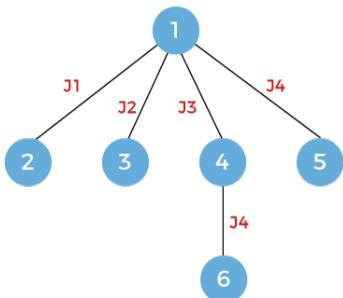
Now, we will expand the node 1. After expansion, the state space tree would be appeared as:
On each expansion, the node will be pushed into the stack shown as below:

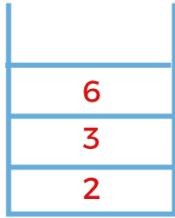


Now the expansion would be based on the node that appears on the top of the stack. Since the node 5 appears on the top of the stack, so we will expand the node 5. We will pop out the node 5 from the stack. Since the node 5 is in the last job, i.e., j4 so there is no further scope of expansion.

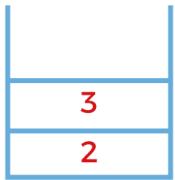


The next node that appears on the top of the stack is node 4. Pop out the node 4 and expand. On expansion, job j4 will be considered and node 6 will be added into the stack shown as below:

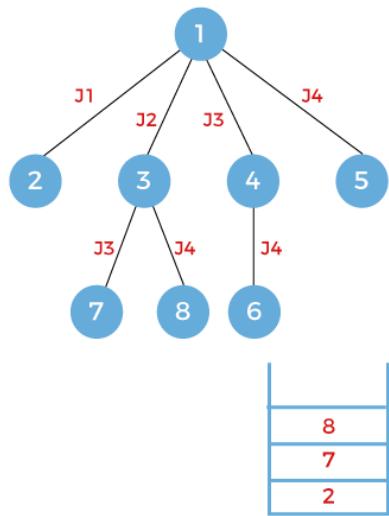




The next node is 6 which is to be expanded. Pop out the node 6 and expand. Since the node 6 is in the last job, i.e., j_4 so there is no further scope of expansion.



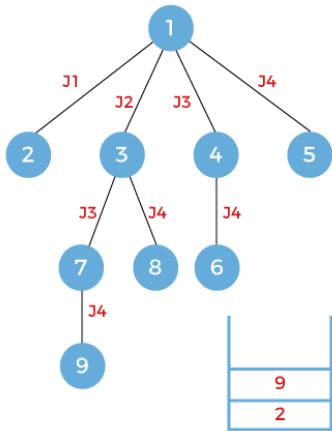
The next node to be expanded is node 3. Since the node 3 works on the job j_2 so node 3 will be expanded to two nodes, i.e., 7 and 8 working on jobs 3 and 4 respectively. The nodes 7 and 8 will be pushed into the stack shown as below:



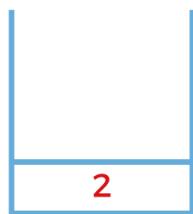
The next node that appears on the top of the stack is node 8. Pop out the node 8 and expand. Since the node 8 works on the job j_4 so there is no further scope for the expansion.



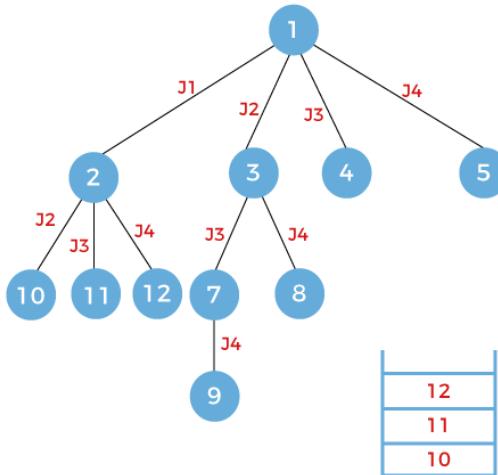
The next node that appears on the top of the stack is node 7. Pop out the node 7 and expand. Since the node 7 works on the job j_3 so node 7 will be further expanded to node 9 that works on the job j_4 as shown as below and the node 9 will be pushed into the stack.



The next node that appears on the top of the stack is node 9. Since the node 9 works on the job 4 so there is no further scope for the expansion.



The next node that appears on the top of the stack is node 2. Since the node 2 works on the job j_1 so it means that the node 2 can be further expanded. It can be expanded upto three nodes named as 10, 11, 12 working on jobs j_2 , j_3 , and j_4 respectively. There newly nodes will be pushed into the stack shown as below:

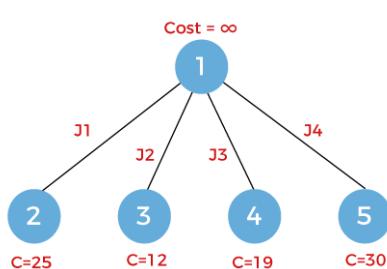


In the above method, we explored all the nodes using the stack that follows the LIFO principle.

Third method

There is one more method that can be used to find the solution and that method is Least cost branch and bound. In this technique, nodes are explored based on the cost of the node. The cost of the node can be defined using the problem and with the help of the given problem, we can define the cost function. Once the cost function is defined, we can define the cost of the node. Let's first consider the node 1 having cost infinity shown as below:

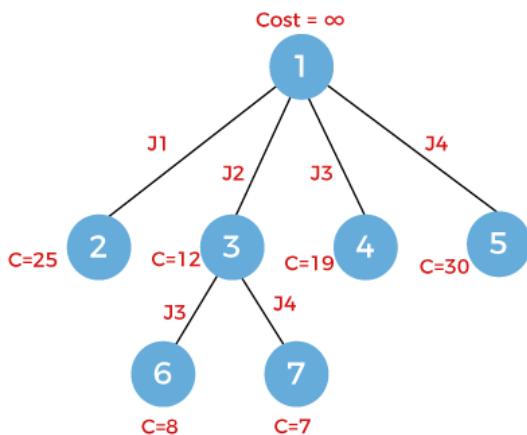
Now we will expand the node 1. The node 1 will be expanded into four nodes named as 2, 3, 4 and 5 shown as below:



Let's assume that cost of the nodes 2, 3, 4, and 5 are 25, 12, 19 and 30 respectively.

Since it is the least cost branch n bound, so we will explore the node which is having the least cost. In the above figure, we can observe that the node with a minimum cost is node 3. So, we will explore the node 3 having cost 12.

Since the node 3 works on the job j2 so it will be expanded into two nodes named as 6 and 7 shown as below:



The node 6 works on job j3 while the node 7 works on job j4. The cost of the node 6 is 8 and the cost of the node 7 is 7. Now we have to select the node which is having the minimum cost. The node 7 has the minimum cost so we will explore the node 7. Since the node 7 already works on the job j4 so there is no further scope for the expansion.

Watch Videos :-

[Branch and Bound Introduction](#)

Job Sequencing with Deadline

Job Sequencing using Branch and Bound: Given n jobs with profit, execution time, and deadline, achieve the schedule which maximizes the profit.

We will solve the problem using the FIFO branch and bound with the variable tuple and fixed tuple representations. Each job i is represented by the tuple (P_i, d_i, t_i) , where P_i , d_i , and t_i represent profit, deadline, and execution time associated with job i . If job i is completed on or before its deadline, profit P_i is earned. But if the job i finish after its deadline, then a penalty P_i will incur. The brute force method finds $2n$ schedules for n jobs and finds the best from them. Branch and bound is a more efficient way of finding the optimal schedule.

Watch Videos :-

[Job Sequencing with Deadline - Branch and Bound](#)

0/1 Knapsack

Branch and bound is an algorithm design paradigm which is generally used for solving combinatorial optimization problems. These problems typically exponential in terms of time complexity and may require exploring all possible permutations in worst case. Branch and Bound solve these problems relatively quickly.

Watch Videos :-

[0/1 Knapsack using Branch and Bound](#)

Traveling Salesman Problem

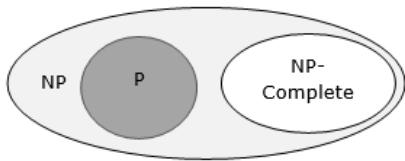
Given a set of cities and distance between every pair of cities, the problem is to find the shortest possible tour that visits every city exactly once and returns to the starting point.

Watch Videos :-

[Traveling Salesman Problem - Branch and Bound](#)

NP-Hard and NP-Complete

A problem is in the class NPC if it is in NP and is as hard as any problem in NP. A problem is NP-hard if all problems in NP are polynomial time reducible to it, even though it may not be in NP itself.



If a polynomial time algorithm exists for any of these problems, all problems in NP would be polynomial time solvable. These problems are called NP-complete. The phenomenon of NP-completeness is important for both theoretical and practical reasons.

Definition of NP-Completeness

A language **B** is **NP-complete** if it satisfies two conditions

- **B** is in NP
- Every **A** in NP is polynomial time reducible to **B**.

If a language satisfies the second property, but not necessarily the first one, the language **B** is known as **NP-Hard**. Informally, a search problem **B** is **NP-Hard** if there exists some **NP-Complete** problem **A** that Turing reduces to **B**.

The problem in NP-Hard cannot be solved in polynomial time, until **P = NP**. If a problem is proved to be NPC, there is no need to waste time on trying to find an efficient algorithm for it. Instead, we can focus on design approximation algorithm.

NP-Complete Problems

Following are some NP-Complete problems, for which no polynomial time algorithm is known.

- Determining whether a graph has a Hamiltonian cycle
- Determining whether a Boolean formula is satisfiable, etc.

NP-Hard Problems

The following problems are NP-Hard

- The circuit-satisfiability problem
- Set Cover
- Vertex Cover
- Travelling Salesman Problem

In this context, now we will discuss TSP is NP-Complete

TSP is NP-Complete

The traveling salesman problem consists of a salesman and a set of cities. The salesman has to visit each one of the cities starting from a certain one and returning to the same city. The challenge of the problem is that the traveling salesman wants to minimize the total length of the trip

Proof

To prove TSP is NP-Complete, first we have to prove that TSP belongs to NP. In TSP, we find a tour and check that the tour contains each vertex once. Then the total cost of the edges of the tour is calculated. Finally, we check if the cost is minimum. This can be completed in polynomial time. Thus TSP belongs to NP.

Secondly, we have to prove that **TSP is NP-hard**. To prove this, one way is to show that **Hamiltonian cycle \leq_p TSP** (as we know that the Hamiltonian cycle problem is NPcomplete).

Assume $\mathbf{G} = (\mathbf{V}, \mathbf{E})$ to be an instance of Hamiltonian cycle.

Hence, an instance of TSP is constructed. We create the complete graph $\mathbf{G}' = (\mathbf{V}', \mathbf{E}')$, where

$$\mathbf{E}' = \{(i, j) : i, j \in \mathbf{V} \text{ and } i \neq j\}$$

Thus, the cost function is defined as follows –

$$t(i, j) = \begin{cases} 0 & \text{if } (i, j) \in \mathbf{E} \\ 1 & \text{otherwise} \end{cases}$$

Now, suppose that a Hamiltonian cycle \mathbf{h} exists in \mathbf{G} . It is clear that the cost of each edge in \mathbf{h} is **0** in \mathbf{G}' as each edge belongs to \mathbf{E} . Therefore, \mathbf{h} has a cost of **0** in \mathbf{G}' . Thus, if graph \mathbf{G} has a Hamiltonian cycle, then graph \mathbf{G}' has a tour of **0** cost.

Conversely, we assume that \mathbf{G}' has a tour \mathbf{h}' of cost at most **0**. The cost of edges in \mathbf{E}' are **0** and **1** by definition. Hence, each edge must have a cost of **0** as the cost of \mathbf{h}' is **0**. We therefore conclude that \mathbf{h}' contains only edges in \mathbf{E} .

We have thus proven that \mathbf{G} has a Hamiltonian cycle, if and only if \mathbf{G}' has a tour of cost at most **0**. TSP is NP-complete.

NP-Completeness

A decision problem L is NP-Hard if

$$L' \leq_p L \text{ for all } L' \in \text{NP}.$$

Definition: L is NP-complete if

$$L \in \text{NP} \text{ and}$$

$L' \leq_p L$ for some known NP-complete problem L' . Given this formal definition, the complexity classes are:

P: is the set of decision problems that are solvable in polynomial time.

NP: is the set of decision problems that can be verified in polynomial time.

NP-Hard: L is NP-hard if for all $L' \in \text{NP}$, $L' \leq_p L$. Thus if we can solve L in polynomial time, we can solve all NP problems in polynomial time.

NP-Complete L is NP-complete if

$$L \in \text{NP} \text{ and}$$

L is NP-hard

If any NP-complete problem is solvable in polynomial time, then every NP-Complete problem is also solvable in polynomial time. Conversely, if we can prove that any NP-Complete problem cannot be solved in polynomial time, every NP-Complete problem cannot be solvable in polynomial time.

Reductions

Concept: - If the solution of NPC problem does not exist then the conversion from one NPC problem to another NPC problem within the polynomial time. For this, you need the concept of reduction. If a solution of the one NPC problem exists within the polynomial time, then the rest of the problem can also give the solution in polynomial time (but it's hard to believe). For this, you need the concept of reduction.

Example: - Suppose there are two problems, **A** and **B**. You know that it is impossible to solve problem **A** in polynomial time. You want to prove that **B** cannot be solved in polynomial time. So you can convert the problem **A** into problem **B** in polynomial time.

Example of NP-Complete problem

NP problem: - Suppose a DECISION-BASED problem is provided in which a set of inputs/high inputs you can get high output.

Criteria to come either in NP-hard or NP-complete.

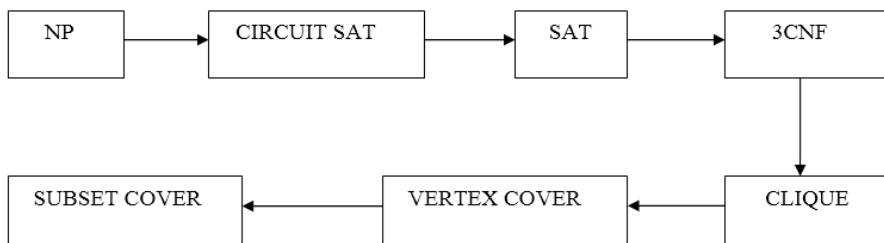
- The point to be noted here, the output is already given, and you can verify the output/solution within the polynomial time but can't produce an output/solution in polynomial time.
- Here we need the concept of reduction because when you can't produce an output of the problem according to the given input then in case you have to use an emphasis on the concept of reduction in which you can convert one problem into another problem.

Note :-

- If you satisfy both points then your problem comes into the category of NP-complete class
- If you satisfy the only 2nd points then your problem comes into the category of NP-hard class

So according to the given decision-based NP problem, you can decide in the form of yes or no. If, yes then you have to do verify and convert into another problem via reduction concept. If you are being performed, both then decision-based NP problems are in NP compete.

Here we will emphasize NPC.



Watch Videos :-

[NP-Hard and NP-Complete Problems](#)

Clique Decision Problem

In the field of computer science, the clique decision problem is a kind of computation problem for finding the cliques or the subsets of the vertices which when all of them are adjacent to each other are also called complete subgraphs.

The clique decision problem has many formulations based on which the cliques and about the cliques the information should be found. There are some common formulations based on which the cliques are based such as finding the maximum clique, finding the maximum weight of the clique in a weighted graph, then listing all the maximum or maximal cliques, and finally solving the problem based on the decision of testing whether the graph has the larger cliques than that of the given size.

Maximum clique: a particular clique that has the largest possible number of vertices.

Maximal cliques: the cliques which further cannot be enlarged.

Applications of Clique decision problems :

Clique decision finding problems algorithms are most abundantly used in chemistry to find the chemicals which have a match with a target structure and then to prepare the docking of the molecule and then the binding sites of common chemical reactions. They might be used to gather the information to find similar structures inside some different molecule. Hence, in these kinds of applications, a graph is formed where every vertex represents a pair that has a match with the pair of atoms one from each of the two molecules. So, two vertices are then connected by an edge if the match they represent is compatible with each other. Therefore, a clique in this particular graph denotes the set of a matched number of pairs of atoms in which all the matched elements are compatible with each other. Several special methods could have also been used such as the modular product of graphs, which can be used to reduce the problem of finding the maximum common induced sub graph in between two graphs to the problem of finding the maximum clique for their product.

It can also be used in automatic test pattern generation, where finding the number of cliques can also help to bound the size of a test case or set.

Used in bioinformatics, the clique algorithm has been used to achieve evolutionary trees, sometimes predicting protein type structures and then finding closely the interacting clusters inside proteins.

Consider an **algorithm to find the maximum** clique as follows :

```

When S = NULL.
for i = 1 till k then start do-while loop.
t : = ch(1 to n)
if t belongs to S then
return fail.
When S:= S union t
Then for all pairs of I and j such that when i belongs to S and j belongs to S and
if i not equal to j then start do-while loop.
And check if I and j is not an edge of that given graph then
Return fail.
Else return a true value.

```

To Prove: - Clique is an NPC or not?

For this you have to satisfy the following below-mentioned points: -

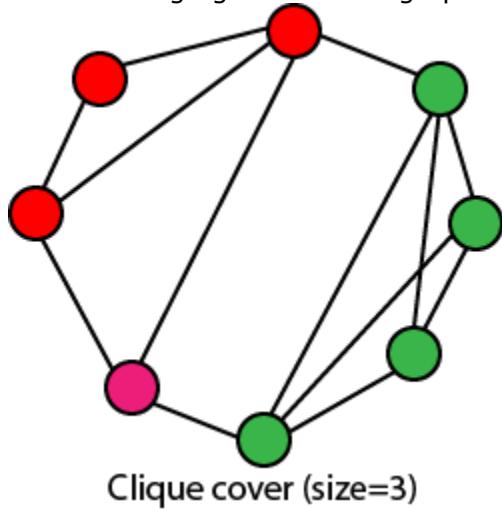
- o Clique
- o $3CNF \leq_p$ Clique
- o $Clique \leq_p 3CNF \leq SAT$
- o $Clique \in NP$

Clique

Definition: - In Clique, every vertex is directly connected to another vertex, and the number of vertices in the Clique represents the Size of Clique.

CLIQUE COVER: - Given a graph G and an integer k, can we find k subsets of vertices $V_1, V_2 \dots V_k$, such that $\cup V_i = V$, and that each V_i is a clique of G.

The following figure shows a graph that has a clique cover of size 3.



$3CNF \leq_p$ Clique

Proof:- For the successful conversion from 3CNF to Clique, you have to follow the two steps:-

Draw the clause in the form of vertices, and each vertex represents the literals of the clauses.

- o They do not complement each other
- o They don't belong to the same clause

In the conversion, the size of the Clique and size of 3CNF must be the same, and you successfully converted 3CNF into Clique within the polynomial time

$Clique \leq_p 3CNF$

Proof: - As you know that a function of K clause, there must exist a Clique of size k. It means that P variables which are from the different clauses can assign the same value (say it is 1). By using these values of all the variables of the CLIQUES, you can make the value of each clause in the function is equal to 1

Example: - You have a Boolean function in 3CNF:-

$$(X+Y+Z) \quad (X+Y+Z') \quad (X+Y'+Z)$$

After Reduction/Conversion from 3CNF to CLIQUE, you will get P variables such as: $-x +y=1$, $x +z=1$ and $x=1$

Put the value of P variables in equation (i)

$$(1+1+0) (1+0+0) (1+0+1)$$

$$(1)(1)(1)=1 \text{ output verified}$$

Clique \in NP

Proof: - As you know very well, you can get the Clique through 3CNF and to convert the decision-based NP problem into 3CNF you have to first convert into SAT and SAT comes from NP.

So, concluded that CLIQUE belongs to NP.

Proof of NPC:-

- o Reduction achieved within the polynomial time from 3CNF to Clique
- o And verified the output after Reduction from Clique To 3CNF above So, concluded that, if both Reduction and verification can be done within the polynomial time that means **Clique also in NPC**.

Watch Videos :-

[NP-Hard Graph Problem - Clique Decision Problem](#)

Vertex cover problem

What is the vertex cover problem?

Find the minimum size vertex cover, when there is a given graph of n nodes and m edges.

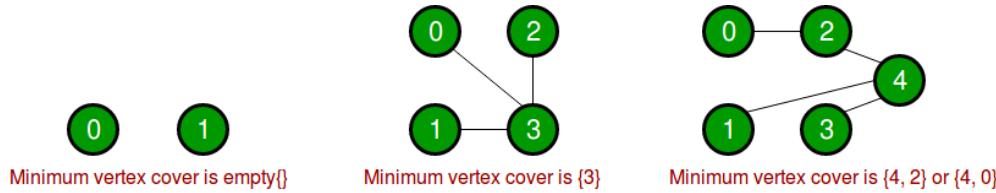
What is vertex cover?

The vertex cover of a graph refers to the subset of its vertices, such for every edge in the graph, that is from every vertex u to v, at least one must be the part of the subset.

The vertex cover problem falls under the category of NP-complete problem. It implies that there is no polynomial-time solution for finding the minimum vertex cover of a graph, until and unless we can prove P=NP. Though, there exist polynomial-time approximate algorithms to find the vertex cover of the graph.

A vertex cover of an undirected graph is a subset of its vertices such that for every edge (u, v) of the graph, either 'u' or 'v' is in the vertex cover. Although the name is Vertex Cover, the set covers all edges of the given graph. Given an undirected graph, the vertex cover problem is to find minimum size vertex cover.

The following are some examples.



Vertex Cover Problem is a known NP Complete problem, i.e., there is no polynomial-time solution for this unless P = NP. There are approximate polynomial-time algorithms to solve the problem though.

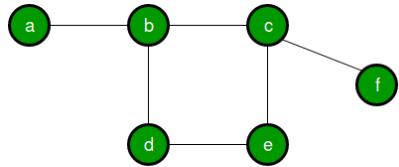
Naive Approach

Consider all the subset of vertices one by one and find out whether it covers all edges of the graph. For eg. in a graph consisting only 3 vertices the set consisting of the combination of vertices are: $\{0,1,2, \{0,1\}, \{0,2\}, \{1,2\}, \{0,1,2\}\}$. Using each element of this set check whether these vertices cover all all the edges of the graph. Hence update the optimal answer. And hence print the subset having minimum number of vertices which also covers all the edges of the graph.

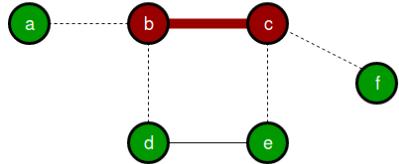
Approximate Algorithm for Vertex Cover:

1. Initialize the result as {}
2. Consider a set of all edges in given graph. Let the set be E.
3. Do following while E is not empty
 - a. Pick an arbitrary edge (u, v) from set E and add 'u' and 'v' to result
 - b. Remove all edges from E which are either incident on u or v.
4. Return result

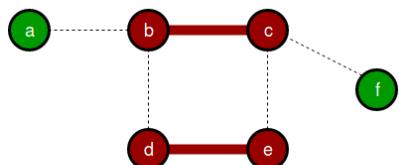
Below diagram to show the execution of the above approximate algorithm:



(a)



(b)



(c)

Minimum Vertex Cover is {b, c, d} or {b, c, e}

Watch Videos:-

[Vertex cover Problem with example](#)

Set Cover Problem

There are a set whose domain is X of n points, and m subsets. We need to find the least number of subsets to cover all the points.

The approach to solving it is as follows:

We pick the set covering the maximum number of points and throw out all the points covered.

The set cover problem is also an NP-complete problem.

Set Cover Problem (Greedy Approximate Algorithm)

Given a universe U of n elements, a collection of subsets of U say $S = \{S_1, S_2, \dots, S_m\}$ where every subset S_i has an associated cost. Find a minimum cost subcollection of S that covers all elements of U .

Example:

```
U = {1,2,3,4,5}
S = {S1,S2,S3}
```

```
S1 = {4,1,3},      Cost(S1) = 5
S2 = {2,5},        Cost(S2) = 10
S3 = {1,4,3,2},   Cost(S3) = 3
```

Output: Minimum cost of set cover is 13 and
set cover is $\{S_2, S_3\}$

There are two possible set covers $\{S_1, S_2\}$ with cost 15
and $\{S_2, S_3\}$ with cost 13.

Set Cover is NP-Hard

There is no polynomial time solution available for this problem as the problem is a known NP-Hard problem. There is a polynomial time Greedy approximate algorithm, the greedy algorithm provides a Logn approximate algorithm.

2-Approximate Greedy Algorithm

Let U be the universe of elements, $\{S_1, S_2, \dots, S_m\}$ be collection of subsets of U and $\text{Cost}(S_1), C(S_2), \dots, \text{Cost}(S_m)$ be costs of subsets.

- 1) Let I represents set of elements included so far. Initialize $I = \{\}$
- 2) Do following while I is not same as U .
 - a) Find the set S_i in $\{S_1, S_2, \dots, S_m\}$ whose cost effectiveness is smallest, i.e., the ratio of cost $C(S_i)$ and number of newly added elements is minimum.
Basically we pick the set for which following value is minimum.
$$\text{Cost}(S_i) / |S_i - I|$$
 - b) Add elements of above picked S_i to I , i.e., $I = I \cup S_i$

Example:

Let us consider the above example to understand Greedy Algorithm.

First Iteration:

```
I = {}
The per new element cost for S1 = Cost(S1)/|S1 - I| = 5/3
The per new element cost for S2 = Cost(S2)/|S2 - I| = 10/2
The per new element cost for S3 = Cost(S3)/|S3 - I| = 3/4
Since S3 has minimum value S3 is added, I becomes {1,4,3,2}.
```

Second Iteration:

```
I = {1,4,3,2}
The per new element cost for S1 = Cost(S1)/|S1 - I| = 5/0
Note that S1 doesn't add any new element to I.
The per new element cost for S2 = Cost(S2)/|S2 - I| = 10/1
Note that S2 adds only 5 to I.
```

The greedy algorithm provides the optimal solution for above example, but it may not provide optimal solution all the time. Consider the following example.

```
S1 = {1, 2}
S2 = {2, 3, 4, 5}
S3 = {6, 7, 8, 9, 10, 11, 12, 13}
S4 = {1, 3, 5, 7, 9, 11, 13}
S5 = {2, 4, 6, 8, 10, 12, 13}
```

Let the cost of every set be same.
 The greedy algorithm produces result as {S₃, S₂, S₁}
 The optimal solution is {S₄, S₅}

Proof that the above greedy algorithm is Logn approximate.

Let OPT be the cost of optimal solution. Say (k-1) elements are covered before an iteration of above greedy algorithm. The cost of the k'th element $\leq \text{OPT} / (n-k+1)$ (Note that cost of an element is evaluated by cost of its set divided by number of elements added by its set). How did we get this result? Since k'th element is not covered yet, there is a S_i that has not been covered before the current step of greedy algorithm and it is there in OPT. Since greedy algorithm picks the most cost effective S_i, per-element-cost in the picked set must be smaller than OPT divided by remaining elements. Therefore cost of k'th element $\leq \text{OPT}/|U-I|$ (Note that U-I is set of not yet covered elements in Greedy Algorithm). The value of |U-I| is n - (k-1) which is n-k+1.

```
Cost of Greedy Algorithm = Sum of costs of n elements
[putting k = 1, 2..n in above formula]
<= (OPT/n + OPT(n-1) + ... + OPT/n)
<= OPT(1 + 1/2 + ..... 1/n)
[Since 1 + 1/2 + .. 1/n ≈ Log n]
<= OPT * Logn
```

Watch Videos: -

[Dynamic Programming Over Subsets for Set Cover](#)
[Algorithm | Algorithm of Set Cover Problem \(Greedy Approximation Algorithm\)](#)

Knuth-Morris-Pratt (KMP) Algorithm

The Knuth-Morris-Pratt string-searching/ Matching algorithm (or KMP algorithm) searches for occurrences of a "word" W within a main "text string" S by employing the observation that when a mismatch occurs, the word itself embodies sufficient information to determine where the next match could begin, thus bypassing re-examination of previously matched characters.

Knuth-Morris and Pratt introduce a linear time algorithm for the string matching problem. A matching time of $O(n)$ is achieved by avoiding comparison with an element of ' S ' that have previously been involved in comparison with some element of the pattern ' p ' to be matched. i.e., backtracking on the string ' S ' never occurs

The Prefix Function (Π): The Prefix Function, Π for a pattern encapsulates knowledge about how the pattern matches against the shift of itself. This information can be used to avoid a useless shift of the pattern ' p '. In other words, this enables avoiding backtracking of the string ' S '.

The KMP Matcher: With string ' S ', pattern ' p ' and prefix function ' Π ' as inputs, find the occurrence of ' p ' in ' S ' and returns the number of shifts of ' p ' after which occurrences are found.

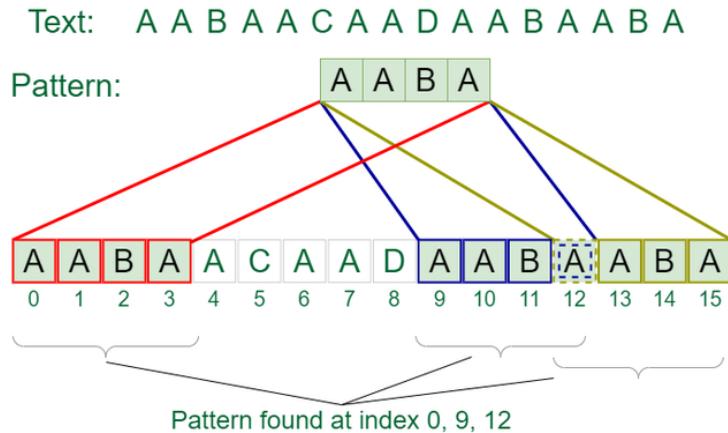
Given a text $txt[0..N-1]$ and a pattern $pat[0..M-1]$, write a function $search(char pat[], char txt[])$ that prints all occurrences of $pat[]$ in $txt[]$. You may assume that $N > M$.

Examples:

Input: $txt[] = \text{"THIS IS A TEST TEXT"}$, $pat[] = \text{"TEST"}$
 Output: Pattern found at index 10

Input: $txt[] = \text{"AABAACAAADAABAABA"}$
 $pat[] = \text{"AABA"}$

Output: Pattern found at index 0, Pattern found at index 9, Pattern found at index 12



Pattern searching is an important problem in computer science. When we do search for a string in a notepad/word file or browser or database, pattern-searching algorithms are used to show the search results.

The worst-case complexity of the Naive algorithm is $O(m(n-m+1))$. The time complexity of the KMP algorithm is $O(n)$ in the worst case. KMP (Knuth Morris Pratt) Pattern Searching. The Naive pattern-searching algorithm doesn't work well in cases where we see many matching characters followed by a mismatching character.

Program to implement Knuth-Morris-Pratt (KMP) Algorithm

```

// Knuth-Morris-Pratt (KMP) Algorithm
#include <stdio.h>
#include <string.h>

void computeLPSArray(char* pat, int M, int* lps);

// Prints occurrences of txt[] in pat[]
void KMPSearch(char* pat, char* txt)
{
    int M = strlen(pat);
    int N = strlen(txt);
    // create lps[] that will hold the longest prefix suffix values for pattern
    int lps[M];
    // Preprocess the pattern (calculate lps[] array)
    computeLPSArray(pat, M, lps);
    int i = 0; // index for txt[]
    int j = 0; // index for pat[]

    while (i < N)
    {
        if (pat[j] == txt[i])
        {
            j++;
            i++;
        }

        if (j == M)
        {
            printf("Found pattern at index %d ", i - j);
            j = lps[j - 1];
        }

        // mismatch after j matches
        else if (i < N && pat[j] != txt[i])
        {
            // Do not match lps[0..lps[j-1]] characters, they will match anyway
            if (j != 0)
                j = lps[j - 1];
            else
                i = i + 1;
        }
    }
}

// Fills lps[] for given pattern pat[0..M-1]
void computeLPSArray(char* pat, int M, int* lps)
{
    // length of the previous longest prefix suffix
    int len = 0;
    lps[0] = 0; // lps[0] is always 0

    // the loop calculates lps[i] for i = 1 to M-1
    int i = 1;
    while (i < M) {
        if (pat[i] == pat[len])
        {
            len++;
            lps[i] = len;
        }
        else
            len = 0;
        i++;
    }
}

```

```

        i++;
    }
    else // (pat[i] != pat[len])
    {
        // This is tricky. Consider the example.
        // AACACAAAA and i = 7. The idea is similar
        // to search step.
        if (len != 0) {
            len = lps[len - 1];

            // Also, note that we do not increment // i here
        }
        else // if (len == 0)
        {
            lps[i] = 0;
            i++;
        }
    }
}

int main()
{
    char txt[] = "ABABDABACDABABCABAB";
    char pat[] = "ABABCABAB";
    KMPSearch(pat, txt);
    return 0;
}

```

Output

Found pattern at index 10

Watch Videos: -

[Knuth-Morris-Pratt KMP String Matching Algorithm](#)

Naive algorithm for Pattern Searching

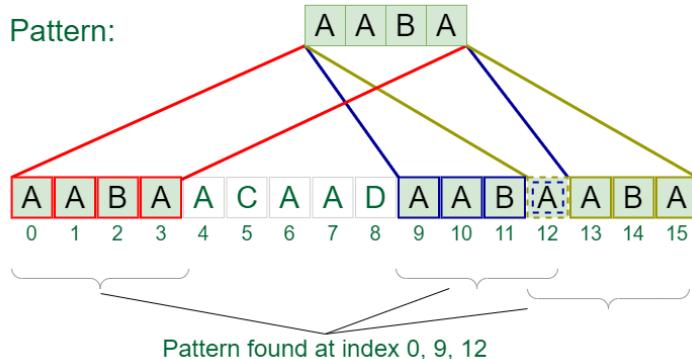
Given a text of length N $txt[0..N-1]$ and a pattern of length M $pat[0..M-1]$, write a function $search(char pat[], char txt[])$ that prints all occurrences of $pat[]$ in $txt[]$. You may assume that $N > M$.

Examples:

Input: $txt[] = "THIS IS A TEST TEXT"$, $pat[] = "TEST"$
 Output: Pattern found at index 10

Input: $txt[] = "AABAACAAADAABAABA"$, $pat[] = "AABA"$
 Output: Pattern found at index 0, Pattern found at index 9, Pattern found at index 12

Text: A A B A A C A A D A A B A A B A



Program to implement Knuth-Morris-Pratt (KMP) Algorithm - Naive

```
//Naive Pattern Searching algorithm
#include <stdio.h>
#include <string.h>

void search(char* pat, char* txt)
{
    int M = strlen(pat);
    int N = strlen(txt);

    /* A loop to slide pat[] one by one */
    for (int i = 0; i <= N - M; i++) {
        int j;

        /* For current index i, check for pattern match */
        for (j = 0; j < M; j++)
            if (txt[i + j] != pat[j])
                break;

        if (j == M) // if pat[0...M-1] = txt[i, i+1, ...i+M-1]
            printf("Pattern found at index %d \n", i);
    }
}

int main()
{
    char txt[] = "AABAACAAADAABAAABAA";
    char pat[] = "AABA";

    // Function call
    search(pat, txt);
    return 0;
}
```

Output

```
Pattern found at index 0
Pattern found at index 9
Pattern found at index 13
```

Rabin-Karp String Matching Algorithm

The Rabin-Karp string matching algorithm calculates a hash value for the pattern, as well as for each M-character subsequences of text to be compared. If the hash values are unequal, the algorithm will determine the hash value for next M-character sequence. If the hash values are equal, the algorithm will analyze the pattern and the M-character sequence. In this way, there is only one comparison per text subsequence, and character matching is only required when the hash values match.

Example: For string matching, working module $q = 11$, how many spurious hits does the Rabin-Karp matcher encounters in Text $T = 31415926535\ldots$

$T = 31415926535\ldots$

$P = 26$

Here $T.Length = 11$ so $Q = 11$

And $P \bmod Q = 26 \bmod 11 = 4$

Now find the exact match of $P \bmod Q\ldots$

Solution:

$T = [3, 1, 4, 1, 5, 9, 2, 6, 5, 3, 5]$

$P = [2, 6]$

$S = 0$



$31 \bmod 11 = 9$ not equal to 4

$S = 1$



$14 \bmod 11 = 3$ not equal to 4

$S = 2$

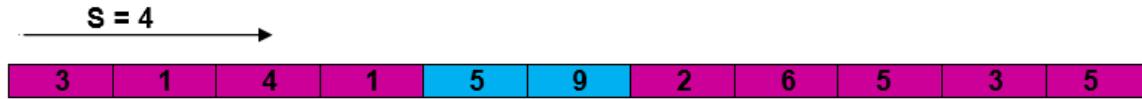


$41 \bmod 11 = 8$ not equal to 4

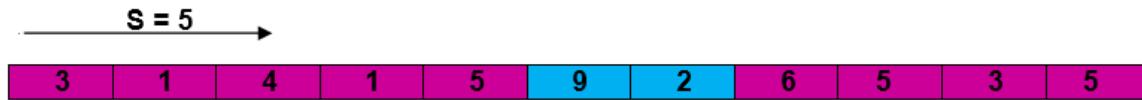
$S = 3$



15 mod 11 = 4 equal to 4 SPURIOUS HIT



59 mod 11 = 4 equal to 4 SPURIOUS HIT



92 mod 11 = 4 equal to 4 SPURIOUS HIT



26 mod 11 = 4 EXACT MATCH



65 mod 11 = 10 not equal to 4



53 mod 11 = 9 not equal to 4



35 mod 11 = 2 not equal to 4

The Pattern occurs with shift 6.

Examples:

Input: txt[] = "THIS IS A TEST TEXT", pat[] = "TEST"
Output: Pattern found at index 10

Input: txt[] = "AABAACAADAABAABA", pat[] = "AABA"
Output: Pattern found at index 0
Pattern found at index 9
Pattern found at index 12

Program to implement Rabin-Karp String Matching Algorithm

```

/* Rabin-Karp Algorithm for Pattern Searching */
#include <stdio.h>
#include <string.h>

// d is the number of characters in the input alphabet
#define d 256

/* pat -> pattern
   txt -> text
   q -> A prime number
*/
void search(char pat[], char txt[], int q)
{
    int M = strlen(pat);
    int N = strlen(txt);
    int i, j;
    int p = 0; // hash value for pattern
    int t = 0; // hash value for txt
    int h = 1;

    // The value of h would be "pow(d, M-1)%q"
    for (i = 0; i < M - 1; i++)
        h = (h * d) % q;

    // Calculate the hash value of pattern and first
    // window of text
    for (i = 0; i < M; i++) {
        p = (d * p + pat[i]) % q;
        t = (d * t + txt[i]) % q;
    }

    // Slide the pattern over text one by one
    for (i = 0; i <= N - M; i++) {

        // Check the hash values of current window of text
        // and pattern. If the hash values match then only
        // check for characters one by one
        if (p == t) {
            /* Check for characters one by one */
            for (j = 0; j < M; j++) {
                if (txt[i + j] != pat[j])
                    break;
            }

            // if p == t and pat[0...M-1] = txt[i, i+1,
            // ...i+M-1]
            if (j == M)
                printf("Pattern found at index %d \n", i);
        }

        // Calculate hash value for next window of text:
        // Remove leading digit, add trailing digit
        if (i < N - M) {
            t = (d * (t - txt[i] * h) + txt[i + M]) % q;

            // We might get negative value of t, converting
            // it to positive
            if (t < 0)

```

```
    t = (t + q);
}
}

int main()
{
    char txt[] = "AABAACAAADAABAABA";
    char pat[] = "AABA";

    // A prime number
    int q = 101;

    // function call
    search(pat, txt, q);
    return 0;
}
```

Output

```
Pattern found at index 0
Pattern found at index 9
Pattern found at index 12
```

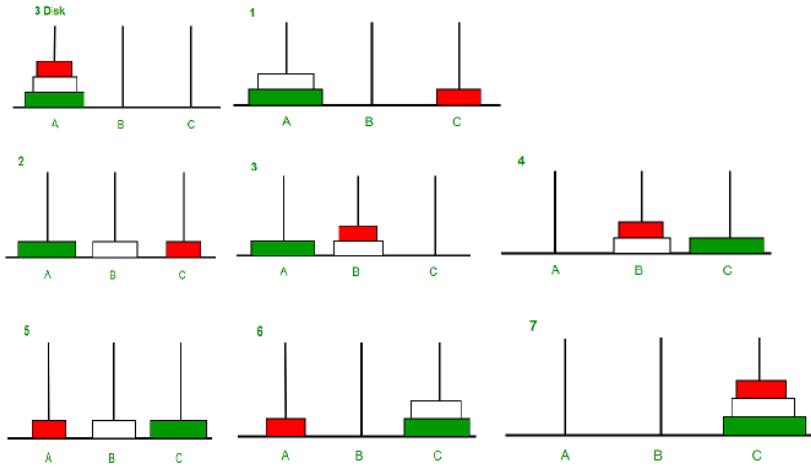
Watch Videos: -

[Rabin-Karp String Matching Algorithm](#)

Tower of Hanoi Problem

Tower of Hanoi is a mathematical puzzle where we have three rods and n disks. **The objective of the puzzle is to move the entire stack to another rod, obeying the following simple rules:**

1. Only one disk can be moved at a time.
2. Each move consists of taking the upper disk from one of the stacks and placing it on top of another stack i.e. a disk can only be moved if it is the uppermost disk on a stack.
3. No disk may be placed on top of a smaller disk.



Program to implement Tower of Hanoi

```
#include <stdio.h>

// C recursive function to solve tower of hanoi puzzle
void towerOfHanoi(int n, char from_rod, char to_rod, char aux_rod)
{
    if (n == 1)
    {
        printf("\n Move disk 1 from rod %c to rod %c", from_rod, to_rod);
        return;
    }
    towerOfHanoi(n-1, from_rod, aux_rod, to_rod);
    printf("\n Move disk %d from rod %c to rod %c", n, from_rod, to_rod);
    towerOfHanoi(n-1, aux_rod, to_rod, from_rod);
}

int main()
{
    int n = 4; // Number of disks
    towerOfHanoi(n, 'A', 'C', 'B'); // A, B and C are names of rods
    return 0;
}
```

Output

```
Move disk 1 from rod A to rod B
Move disk 2 from rod A to rod C
Move disk 1 from rod B to rod C
Move disk 3 from rod A to rod B
Move disk 1 from rod C to rod A
Move disk 2 from rod C to rod B
Move disk 1 from rod A to rod B
Move disk 4 from rod A to rod C
Move disk 1 from rod B to rod C
Move disk 2 from rod B to rod A
Move disk 1 from rod C to rod A
Move disk 3 from rod B to rod C
Move disk 1 from rod A to rod B
Move disk 2 from rod A to rod C
Move disk 1 from rod B to rod C
```

Watch Videos: -

[Tower of Hanoi Problem](#)

More Topic

Fibonacci Sequence

Fibonacci sequence is the sequence of numbers in which every next item is the total of the previous two items. And each number of the Fibonacci sequence is called Fibonacci number.

Example: 0 ,1,1,2,3,5,8,13,21,..... is a Fibonacci sequence.

The Fibonacci numbers F_n are defined as follows:

```

 $F_0 = 0$ 
 $F_1 = 1$ 
 $F_n = F_{(n-1)} + F_{(n-2)}$ 
FIB (n)
1. If (n < 2)
2. then return n
3. else return FIB (n - 1) + FIB (n - 2)

```

Figure: shows four levels of recursion for the call fib (8):

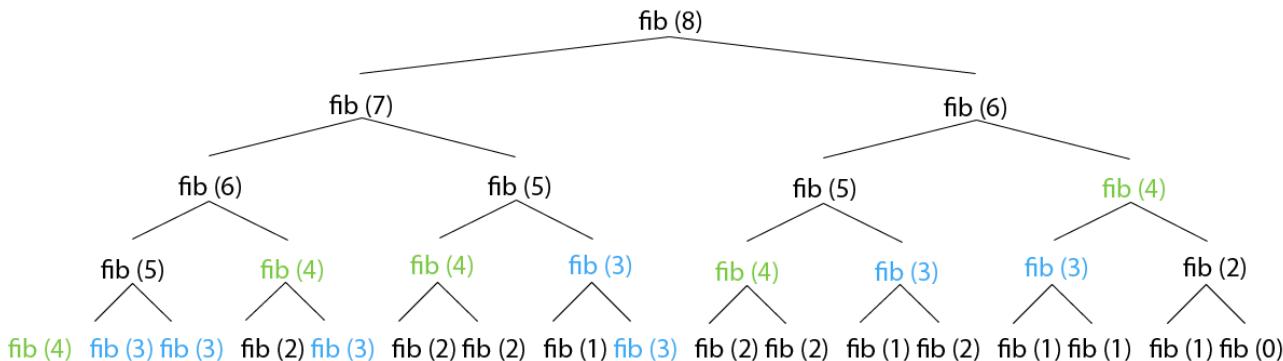


Figure: Recursive calls during computation of Fibonacci number

A single recursive call to fib (n) results in one recursive call to fib (n - 1), two recursive calls to fib (n - 2), three recursive calls to fib (n - 3), five recursive calls to fib (n - 4) and, in general, F_{k-1} recursive calls to fib (n - k). We can avoid this unneeded repetition by writing down the conclusion of recursive calls and looking them up again if we need them later. This process is called memorization.

Here is the algorithm with memorization

```

MEMOFIB (n)
1 if (n < 2)
2 then return n
3 if (F[n] is undefined)
4 then F[n] ← MEMOFIB (n - 1) + MEMOFIB (n - 2)
5 return F[n]

```

If we trace through the recursive calls to MEMOFIB, we find that array F [] gets filled from bottom up. I.e., first F [2], then F [3], and so on, up to F[n]. We can replace recursion with a simple for-loop that just fills up the array F [] in that order

```
ITERFIB (n)
1 F [0] ← 0
2 F [1] ← 1
3 for i ← 2 to n
4 do
5 F[i] ← F [i - 1] + F [i - 2]
6 return F[n]
```

This algorithm clearly takes only $O(n)$ time to compute F_n . By contrast, the original recursive

algorithm takes $O(\phi^n)$, $\phi = \frac{1 + \sqrt{5}}{2} = 1.618$. ITERFIB conclude an exponential speedup over the original recursive algorithm.

Longest Common Subsequence (LCS)

LCS Problem Statement: Given two sequences, find the length of longest subsequence present in both of them. A subsequence is a sequence that appears in the same relative order, but not necessarily contiguous.

For example, "abc", "abg", "bdf", "aeg", "acefg", .. etc are subsequences of "abcdefg".

In order to find out the complexity of brute force approach, we need to first know the number of possible different subsequences of a string with length n , i.e., find the number of subsequences with lengths ranging from 1,2,.. $n-1$. Recall from theory of permutation and combination that number of combinations with 1 element are $nC1$. Number of combinations with 2 elements are $nC2$ and so forth and so on. We know that $nC0 + nC1 + nC2 + \dots + nCn = 2^n$. So a string of length n has 2^{n-1} different possible subsequences since we do not consider the subsequence with length 0. This implies that the time complexity of the brute force approach will be $O(n * 2^n)$. Note that it takes $O(n)$ time to check if a subsequence is common to both the strings. This time complexity can be improved using dynamic programming.

It is a classic computer science problem, the basis of diff (a file comparison program that outputs the differences between two files), and has applications in bioinformatics.

Examples:

LCS for input Sequences "ABCDGH" and "AEDFHR" is "ADH" of length 3.

LCS for input Sequences "AGGTAB" and "GXTXAYB" is "GTAB" of length 4.

The naive solution for this problem is to generate all subsequences of both given sequences and find the longest matching subsequence. This solution is exponential in term of time complexity. Let us see how this problem possesses both important properties of a Dynamic Programming (DP) Problem.

Optimal Substructure:

Let the input sequences be $X[0..m-1]$ and $Y[0..n-1]$ of lengths m and n respectively. And let $L(X[0..m-1], Y[0..n-1])$ be the length of LCS of the two sequences X and Y . Following is the recursive definition of $L(X[0..m-1], Y[0..n-1])$.

If last characters of both sequences match (or $X[m-1] == Y[n-1]$) then
 $L(X[0..m-1], Y[0..n-1]) = 1 + L(X[0..m-2], Y[0..n-2])$

If last characters of both sequences do not match (or $X[m-1] != Y[n-1]$) then
 $L(X[0..m-1], Y[0..n-1]) = \text{MAX} (L(X[0..m-2], Y[0..n-1]), L(X[0..m-1], Y[0..n-2]))$

Examples:

1. Consider the input strings "AGGTAB" and "GXTXAYB". Last characters match for the strings. So length of LCS can be written as:

$$L("AGGTAB", "GXTXAYB") = 1 + L("AGGTA", "GXTXAY")$$

	A	G	G	T	A	B
G	-	-	4	-	-	-
X	-	-	-	-	-	-
T	-	-	-	3	-	-
X	-	-	-	-	-	-
A	-	-	-	-	2	-
Y	-	-	-	-	-	-
B	-	-	-	-	-	1

2. Consider the input strings "ABCDGH" and "AEDFHR". Last characters do not match for the strings. So length of LCS can be written as:

$L("ABCDGH", "AEDFHR") = \text{MAX} (L("ABCDG", "AEDFHR"), L("ABCDGH", "AEDFH"))$
So the LCS problem has optimal substructure property as the main problem can be solved using solutions to subproblems.

Overlapping Subproblems:

The simple recursive implementation of the LCS problem. The implementation simply follows the recursive structure mentioned above.

Program to implement Longest Common Subsequence

```

/* A Naive recursive implementation of LCS problem */
#include<stdio.h>
#include<string.h>

int max(int a, int b);

/* Returns length of LCS for X[0..m-1], Y[0..n-1] */
int lcs( char *X, char *Y, int m, int n )
{
    if (m == 0 || n == 0)
        return 0;
    if (X[m-1] == Y[n-1])
        return 1 + lcs(X, Y, m-1, n-1);
    else
        return max(lcs(X, Y, m, n-1), lcs(X, Y, m-1, n));
}

/* Utility function to get max of 2 integers */
int max(int a, int b)
{
    return (a > b)? a : b;
}

/* Driver program to test above function */
int main()
{
char X[] = "AGGTAB";
char Y[] = "GXTXAYB";

int m = strlen(X);
int n = strlen(Y);

printf("Length of LCS is %d", lcs( X, Y, m, n ) );
}

return 0;
}

```

Output

Length of LCS is 4

Watch Videos: -

[Longest Common Subsequence \(LCS\) - Recursion and Dynamic Programming](#)

Longest Common Substring

The longest common substring problem is a problem that finds the longest substring of two strings.

There is one difference between the Longest common subsequence and the longest common substring. In the case of substring, all the elements in the substring must be in contiguous in a original string and the order of the elements in the substring should be same as in the string. In the case of subsequence, we can miss out some elements which means that it is not mandatory that the elements in the substring should be contiguous.

Let's understand through an example.

Consider two strings given below:

```
S1: a b c d a f
S2: b c d f
```

On comparing the above two strings, we will find that:

The longest common substring is bcd.

The longest common subsequence is bcdf.

For example: The two strings are given below:

```
S1: ABABCD
S2: BABCDA
```

On comparing the above two strings, we will find that BABCD is the longest common substring.

If we have long strings then it won't be possible to find out the longest common substring. So, we use the dynamic programming approach to solve this problem.

Algorithm

```
Consider two strings S1 and S2.
if (S1[i] == S2[j])
T[i][j] = T[i-1][j-1] + 1
else
T[i][j] = 0
```

Consider two strings given below:

```
S1: a b c d a f
S2: z b c d f
```

		a	b	c	d	a	f
z	0	0	0	0	0	0	0
b	0						
c	0						
d	0						
f	0						

As we can observe in the above table that the first row represents the first string, i.e., S1, and the first column represents the second string, i.e., S2.

When $i=0, j = 0$ where $S1[i] = z, S2[j] = a$

Since there is no common string between $S1[i]$ and $S2[j]$ so the length of the longest common substring would be 0.

		a	b	c	d	a	f
	0	0	0	0	0	0	0
z	0	0					
b	0						
c	0						
d	0						
f	0						

When $i=0, j=1$ where $S1[i] = z, S2[j] = ab$

		a	b	c	d	a	f
	0	0	0	0	0	0	0
z	0	0	0				
b	0						
c	0						
d	0						
f	0						

When $i=0, j=2$ where $S1[i] = z, S2[j] = abc$

		a	b	c	d	a	f
	0	0	0	0	0	0	0
z	0	0	0	0			
b	0						
c	0						
d	0						
f	0						

When $i=0, j = 3$ where $S1[i] = z, S2[j] = abcd$

		a	b	c	d	a	f
	0	0	0	0	0	0	0
z	0	0	0	0	0		
b	0						
c	0						
d	0						
f	0						

Similarly, we will fill other two columns and table would be:

		a	b	c	d	a	f
	0	0	0	0	0	0	0
z	0	0	0	0	0	0	0
b	0						
c	0						
d	0						
f	0						

When $i=1, j=0$ where $S1[1] = b, S2[0] = a$

		a	b	c	d	a	f
	0	0	0	0	0	0	0
z	0	0	0	0	0	0	0
b	0	0					
c	0						
d	0						
f	0						

When $i=1, j=1$ where $S1[1] = b, S2[1] = b$

Since there is one common substring between $S1[1]$ and $S2[1]$, i.e., b so the length of the longest common substring would be 1 shown as below:

		a	b	c	d	a	f
	0	0	0	0	0	0	0
z	0	0	0	0	0	0	0
b	0	0	1				
c	0						
d	0						
f	0						

When $i=1, j=2$ where $S1[1] = b, S2[2] = c$

		a	b	c	d	a	f
	0	0	0	0	0	0	0
z	0	0	0	0	0	0	0
b	0	0	1	0			
c	0						
d	0						
f	0						

Since 'b' and 'c' are not same so we put 0 at $S[1][2]$.

When $i=1, j=3$ where $S1[1] = b, S2[3] = d$

		a	b	c	d	a	f
	0	0	0	0	0	0	0
z	0	0	0	0	0	0	0
b	0	0	1	0	0		
c	0						
d	0						
f	0						

Since 'b' and 'd' are not same so we put 0 at $S[1][3]$.

When i=1, j= 4 where S1[1] = b, S2[4] = a

		a	b	c	d	a	f
	0	0	0	0	0	0	0
z	0	0	0	0	0	0	0
b	0	0	1	0	0	0	
c	0						
d	0						
f	0						

When i=1, j=5 where S1[1] = b, S2[5] = f

		a	b	c	d	a	f
	0	0	0	0	0	0	0
z	0	0	0	0	0	0	0
b	0	0	1	0	0	0	0
c	0						
d	0						
f	0						

Since 'b' and 'f' are not same so we put 0 at S[1][5].

When i=2, j= 0 where S1[2] = c and S2[5] = a

		a	b	c	d	a	f
	0	0	0	0	0	0	0
z	0	0	0	0	0	0	0
b	0	0	1	0	0	0	0
c	0	0					
d	0						
f	0						

Since 'c' and 'a' are not same so we put 0 at S[2][0].

When i=2, j = 1 where S1[2] = 'c' and S2[1] = 'b'

		a	b	c	d	a	f
	0	0	0	0	0	0	0
z	0	0	0	0	0	0	0
b	0	0	1	0	0	0	0
c	0	0	0				
d	0						
f	0						

Since 'c' and 'b' are not same so we put 0 at S[2][1].

When i=2, j=2 where S1[2] = 'c' and S2[2] = 'c'

		a	b	c	d	a	f
	0	0	0	0	0	0	0
z	0	0	0	0	0	0	0
b	0	0	1	0	0	0	0
c	0	0	0	2			
d	0						
f	0						

Since both the characters 'c' are same; therefore, "**bc**" is the common substring among the strings "**zbc**" and "**abc**". The length of the longest common substring is 2.

When i=2, j=3 where S1[2] = 'c' and S2[3] = 'd'

		a	b	c	d	a	f
	0	0	0	0	0	0	0
z	0	0	0	0	0	0	0
b	0	0	1	0	0	0	0
c	0	0	0	2	0		
d	0						
f	0						

Since 'c' and 'd' are not same so we put 0 at S[2][3].

When i=2, j=4 where S1[2] = 'c' and S2[4] = 'a'

		a	b	c	d	a	f
	0	0	0	0	0	0	0
z	0	0	0	0	0	0	0
b	0	0	1	0	0	0	0
c	0	0	0	2	0	0	
d	0						
f	0						

Since 'c' and 'a' are not same so we put 0 at S[2][4].

When i=2, j=5 where S1[2] = 'c' and S2[5] = 'f'

		a	b	c	d	a	f
	0	0	0	0	0	0	0
z	0	0	0	0	0	0	0
b	0	0	1	0	0	0	0
c	0	0	0	2	0	0	
d	0						
f	0						

Since 'c' and 'f' are different so we put 0 at S[2][5].

When i=3, j=0 where S1[3] = 'd' and S2[0] = 'a'

		a	b	c	d	a	f
	0	0	0	0	0	0	0
z	0	0	0	0	0	0	0
b	0	0	1	0	0	0	0
c	0	0	0	2	0	0	
d	0	0					
f	0						

Since 'd' and 'a' are different so we put 0 at S[3][0].

When i=3, j=1 where S1[3] = 'd' and S2[1] = 'b'

		a	b	c	d	a	f
	0	0	0	0	0	0	0
z	0	0	0	0	0	0	0
b	0	0	1	0	0	0	0
c	0	0	0	2	0	0	0
d	0	0	0				
f	0						

Since 'd' and 'b' are not same so we put 0 at S[3][1].

When i=3, j=2 where S1[3] = 'd' and S2[2] = 'c'

		a	b	c	d	a	f
	0	0	0	0	0	0	0
z	0	0	0	0	0	0	0
b	0	0	1	0	0	0	0
c	0	0	0	2	0	0	0
d	0	0	0	0			
f	0						

Since 'd' and 'c' are not same so we put 0 at S[3][2].

When i=3, j=3 where S1[3] = 'd' and S2[3] = 'd'

		a	b	c	d	a	f
	0	0	0	0	0	0	0
z	0	0	0	0	0	0	0
b	0	0	1	0	0	0	0
c	0	0	0	2	0	0	0
d	0	0	0	0	3		
f	0						

Since both the characters, i.e., 'd' is same; therefore, 'bcd' is common substring among the strings '**abcd**' and '**zbcd**'. The length of longest common substring is 3.

Similarly, we will calculate the values of other two columns, i.e., S[3][4] and S[3][5] shown in the below table:

		a	b	c	d	a	f
	0	0	0	0	0	0	0
z	0	0	0	0	0	0	0
b	0	0	1	0	0	0	0
c	0	0	0	2	0	0	0
d	0	0	0	0	3	0	0
f	0						

The final table would be:

		a	b	c	d	a	f
	0	0	0	0	0	0	0
z	0	0	0	0	0	0	0
b	0	0	1	0	0	0	0
c	0	0	0	2	0	0	0
d	0	0	0	0	3	0	0
f	0	0	0	0	0	0	1

As we can observe in the above table that the length of the longest common substring is 3. We can also find the longest common substring from the above table. First, we move to the column having highest value, i.e., 3 and the character corresponding to 3 is 'd', move diagonally across 3 and the number is 2. The character corresponding to 2 is 'c' and again we move diagonally across the 2 and the value is 1. The character corresponding to 1 value is 'b'. Therefore, the substring would be "bcd".

<https://www.javatpoint.com/shortest-common-supersequence>

<https://www.javatpoint.com/longest-common-substring>

<https://www.javatpoint.com/daa-longest-palindrome-subsequence>

<https://www.javatpoint.com/longest-increasing-subsequence>

<https://www.javatpoint.com/longest-common-subsequence>

<https://www.javatpoint.com/longest-repeated-subsequence>

<https://www.javatpoint.com/daa-network-flow-problems>

<https://www.javatpoint.com/daa-maximum-bipartite-matching>

<https://www.javatpoint.com/daa-comparison-network>

<https://www.javatpoint.com/daa-bitonic-sorting-network>

<https://www.javatpoint.com/daa-merging-network>

<https://www.javatpoint.com/longest-common-substring>

<https://www.javatpoint.com/maximum-sum-increasing-subsequence>

<https://www.javatpoint.com/wildcard-pattern-matching>

Check the

<https://www.javatpoint.com/shortest-common-supersequence>

<https://www.geeksforgeeks.org/learn-data-structures-and-algorithms-dsa-tutorial/?ref=shm#patternSearching>

END OF FILE

<https://www.programiz.com/dsa/priority-queue>

<https://www.geeksforgeeks.org/data-structures/>

<https://www.programiz.com/dsa/data-structure-types#:~:text=Basically%20data%20structures%20are%20divided,Non%2Dlinear%20data%20structure>

<https://towardsdatascience.com/8-common-data-structures-every-programmer-must-know-171acf6a1a42#:~:text=Data%20Structures%20are%20a%20specialized,Computer%20Science%20and%20Software%20Engineering.>

<https://tutorialsinhand.com/tutorials/data-structure-tutorial/algorithm/algorithm-basics.aspx>

https://www.youtube.com/watch?v=wU6udHRIkcc&list=PLDN4rrl48XKpZkf03iYFl-O29szjTrs_O&index=18&ab_channel=AbdulBari

Strassens Matrix Multiplication

https://www.youtube.com/watch?v=0oJyNmEbS4w&list=PLDN4rrl48XKpZkf03iYFl-O29szjTrs_O&index=38&ab_channel=AbdulBari

<https://www.vrakshacademy.com/2021/06/multistage-graph-dynamic-programming-ada.html>

https://www.youtube.com/watch?v=xZfmHVi7FMg&ab_channel=Jenny%27slecturesCS%2FITNET%26JRF

<https://www.guru99.com/fractional-knapsack-problem-greedy.html>