

Engineering Notebook

VOLUME 1

EE2315 Algorithms and Data Structures

Dr. Pravin Zode



**DEPARTMENT OF ELECTRONICS
ENGINEERING**

YESHWANTRAO CHAVAN COLLEGE OF ENGINEERING,
(An autonomous institution affiliated to Rashtrasant Tukadoji Maharaj Nagpur University, Nagpur)
NAGPUR – 441110

Copyright © 2012 Author/ s Name

All rights reserved.

ISBN:

MESSAGE / MESSAGES

Insert message text here. Insert message text here. Insert message text here. Insert message text here.
Insert message text here. Insert message text here. Insert message text here. Insert message text here.
Insert message text here. Insert message text here.

CONTENTS

UN	Content	Page no
1	Unit-1	1
2	Unit-2	15
3	Unit-3	31
4	Unit-4	46
5	Unit-5	60
6	Unit-6	79

UNIT- 1

Q1. What is algorithm? Explain in detail

Answer:

The word Algorithm means “a process or set of rules to be followed in calculations or other problem-solving operations”. Therefore Algorithm refers to a set of rules/instructions that step-by-step define how a work is to be executed upon in order to get the expected results.

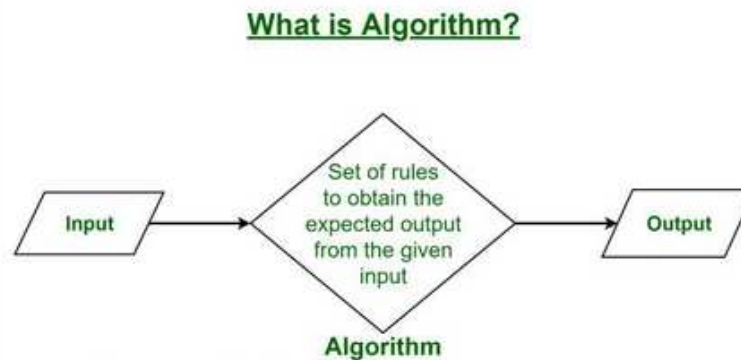


Fig.1 : Algorithm

It can be understood by taking an example of cooking a new recipe. To cook a new recipe, one reads the instructions and steps and execute them one by one, in the given sequence. The result thus obtained is the new dish cooked perfectly. Similarly, algorithms help to do a task in programming to get the expected output.

The Algorithm designed are language-independent, i.e. they are just plain instructions that can be implemented in any language, and yet the output will be the same, as

expected.

Q2. What are the Characteristics of an Algorithm?

Answer:

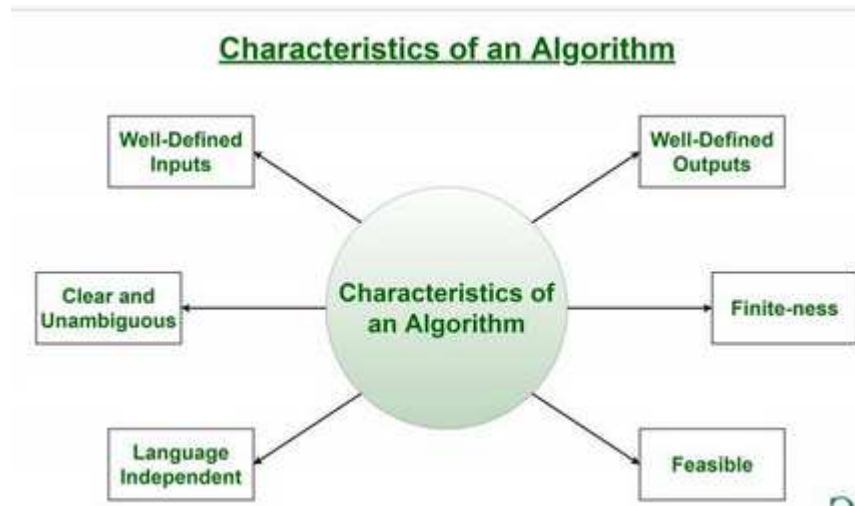


Fig.2 : Characteristics of Algorithm

As one would not follow any written instructions to cook the recipe, but only the standard one. Similarly, not all written instructions for programming is an algorithm. In order for some instructions to be an algorithm, it must have the following characteristics:

Clear and Unambiguous: Algorithm should be clear and unambiguous. Each of its steps should be clear in all aspects and must lead to only one meaning.

Well-Defined Inputs: If an algorithm says to take inputs, it should be well-defined

inputs.

Well-Defined Outputs: The algorithm must clearly define what output will be yielded and it should be well-defined as well.

Finite-ness: The algorithm must be finite, i.e. it should not end up in an infinite loops or similar.

Feasible: The algorithm must be simple, generic and practical, such that it can be executed upon will the available resources. It must not contain some future technology, or anything.

Language Independent: The Algorithm designed must be language-independent, i.e. it must be just plain instructions that can be implemented in any language, and yet the output will be same, as expected.

Q3. How to Design an Algorithm?

Inorder to write an algorithm, following things are needed as a pre-requisite:

The **problem** that is to be solved by this algorithm.

The **constraints** of the problem that must be considered while solving the problem.

The **input** to be taken to solve the problem.

The **output** to be expected when the problem the is solved.

The **solution** to this problem, in the given constraints.

Then the algorithm is written with the help of above parameters such that it solves the problem.

Example: Consider the example to add three numbers and print the sum.

Step 1: Fulfilling the pre-requisites

As discussed above, in order to write an algorithm, its pre-requisites must be fulfilled.

The problem that is to be solved by this algorithm: Add 3 numbers and print their sum.

The constraints of the problem that must be considered while solving the problem:

The numbers must contain only digits and no other characters.

The input to be taken to solve the problem: The three numbers to be added.

The output to be expected when the problem the is solved: The sum of the three numbers taken as the input.

The solution to this problem, in the given constraints: The solution consists of adding the 3 numbers. It can be done with the help of '+' operator, or bit-wise, or any other method.

Step 2: Designing the algorithm

Now let's design the algorithm with the help of above pre-requisites:

Algorithm to add 3 numbers and print their sum:

START

Declare 3 integer variables num1, num2 and num3.

Take the three numbers, to be added, as inputs in variables num1, num2, and num3 respectively.

Declare an integer variable sum to store the resultant sum of the 3 numbers.

Add the 3 numbers and store the result in the variable sum.

Print the value of variable sum

END

Step 3: Testing the algorithm by implementing it.

Q4. How to Analyse an Algorithm?

Answer : For a standard algorithm to be good, it must be efficient. Hence the efficiency of an algorithm must be checked and maintained. It can be in two stages:

Priori Analysis: “Priori” means “before”. Hence Priori analysis means checking the algorithm before its implementation. In this, the algorithm is checked when it is written in the form of theoretical steps. This Efficiency of an algorithm is measured by assuming that all other factors, for example, processor speed, are constant and have no effect on the implementation. This is done usually by the algorithm designer. It is in this method, that the Algorithm Complexity is determined.

Posterior Analysis: “Posterior” means “after”. Hence Posterior analysis means checking the algorithm after its implementation. In this, the algorithm is checked by implementing it in any programming language and executing it. This analysis helps to get the actual and real analysis report about correctness, space required, time consumed etc.

Q5. What is Algorithm Complexity and How to find it?

Answer : An algorithm is defined as complex based on the amount of Space and Time it consumes. Hence the Complexity of an algorithm refers to the measure of the Time that it will need to execute and get the expected output, and the Space it will need to

store all the data (input, temporary data and output). Hence these two factors define the efficiency of an algorithm.

The two factors of Algorithm Complexity are:

Time Factor: Time is measured by counting the number of key operations such as comparisons in the sorting algorithm.

Space Factor: Space is measured by counting the maximum memory space required by the algorithm.

Therefore the complexity of an algorithm can be divided into two types:

Space Complexity: Space complexity of an algorithm refers to the amount of memory that this algorithm requires to execute and get the result. This can be for inputs, temporary operations, or outputs.

Q6. How to calculate Space Complexity?

Answer : The space complexity of an algorithm is calculated by determining following 2 components:

Fixed Part: This refers to the space that is definitely required by the algorithm. For example, input variables, output variables, program size, etc.

Variable Part: This refers to the space that can be different based on the implementation of the algorithm. For example, temporary variables, dynamic memory allocation, recursion stack space, etc.

Q7. What are the differences between Stored procedures and functions?**Answer :**

Functions	Procedures
A function has a return type and returns a value.	A procedure does not have a return type. But it returns values using the OUT parameters.
You cannot use a function with Data Manipulation queries. Only Select queries are allowed in functions.	You can use DML queries such as insert, update, select etc... with procedures.
A function does not allow output parameters	A procedure allows both input and output parameters.
You cannot manage transactions inside a function.	You can manage transactions inside a function.
You cannot call stored procedures from a function	You can call a function from a stored procedure.
You can call a function using a select statement.	You cannot call a procedure using select statements.

Q8. What is top down and bottom up approach

Answer :

Top-Down Design Model:

In top down model, an overview of system is formulated without going into details for any part of it. Each part of it then refined into more details, defining it in yet more details until the entire specification is detailed enough to validate the model. if we glance at a haul as a full, it's going to appear not possible as a result of it's so complicated For example: Writing a University system program, writing a word processor. Complicated issues may be resolved victimization high down style, conjointly referred to as Stepwise refinement where,

We break the problem into parts,

Then break the parts into parts soon and now each of part will be easy to do.

Advantages:

Breaking problems into parts help us to identify what needs to be done.

At each step of refinement new parts will become less complex and therefore easier to solve.

Parts of solution may turn out to be reusable.

Breaking problems into parts allows more than one person to solve the problem.

Bottom-Up Design Model:

In this design, individual parts of the system are specified in details. The parts are the linked to form larger components, which are in turn linked until a complete system is

formed. Object oriented language such as C++ or java uses bottom up approach where each object is identified first.

Advantage:

Make decisions about reusable low level utilities then decide how there will be put together to create high level construct.

Contrast between Top down design and bottom up design.

S.No.	TOP DOWN APPROACH	BOTTOM UP APPROACH
1.	In this approach We focus on breaking up the problem into smaller parts.	In bottom up approach, we solve smaller problems and integrate it as whole and complete the solution.
2.	Mainly used by structured programming language such as COBOL, Fortan, C etc.	Mainly used by object oriented programming language such as C++, C#, Python.
3.	Each part is programmed separately therefore contain redundancy.	Redundancy is minimized by using data encapsulation and data hiding.
4.	In this the communications is less among modules.	In this module must have communication.
5.	It is used in debugging, module documentation, etc.	It is basically used in testing.
6.	In top down approach,	In bottom up approach composition

S.No.	TOP DOWN APPROACH	BOTTOM UP APPROACH
	decomposition takes place.	takes place.
7.	In this top function of system might be hard to identify.	In this sometimes we can not build a program from the piece we have started.
8.	In this implementation details may differ.	This is not natural for people to assemble.

Q8. Structured Programming Approach with Advantages and Disadvantages

Answer :

Structured Programming Approach, as the word suggests, can be defined as a programming approach in which the program is made as a single structure. It means that the code will execute the instruction by instruction one after the other. It doesn't support the possibility of jumping from one instruction to some other with the help of any statement like GOTO, etc. Therefore, the instructions in this approach will be executed in a serial and structured manner. The languages that support Structured programming approach are:

C

C++

Java

C#

On the contrary, in the Assembly languages like Microprocessor 8085, etc, the statements do not get executed in a structured manner. It allows jump statements like GOTO. So the program flow might be random.

The structured program mainly consists of three types of elements:

- Selection Statements
- Sequence Statements
- Iteration Statements

The structured program consists of well structured and separated modules. But the entry and exit in a Structured program is a single-time event. It means that the program uses single-entry and single-exit elements. Therefore a structured program is well maintained, neat and clean program. This is the reason why the Structured Programming Approach is well accepted in the programming world.

Q. 9 What does ‘Space Complexity’ mean?

Answer :

Space Complexity:

The term Space Complexity is misused for Auxiliary Space at many places. Following are the correct definitions of Auxiliary Space and Space Complexity. ***Auxiliary Space* is the extra space or temporary space used by an algorithm.**

Space Complexity of an algorithm is total space taken by the algorithm with respect to the input size. Space complexity includes both Auxiliary space and space used by input.

For example, if we want to compare standard sorting algorithms on the basis of space, then Auxiliary Space would be a better criteria than Space Complexity. Merge Sort uses $O(n)$ auxiliary space, Insertion sort and Heap Sort use $O(1)$ auxiliary space. Space complexity of all these sorting algorithms is $O(n)$ though.

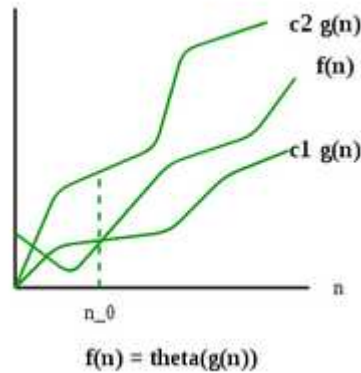
.

Q.10 Short note on Analysis of Algorithms

Answer :

We have discussed Asymptotic Analysis, and Worst, Average and Best Cases of Algorithms. The main idea of asymptotic analysis is to have a measure of efficiency of algorithms that doesn't depend on machine specific constants, and doesn't require

algorithms to be implemented and time taken by programs to be compared. Asymptotic notations are mathematical tools to represent time complexity of algorithms for asymptotic analysis. The following 3 asymptotic notations are mostly used to represent time complexity of algorithms.



1) Θ Notation: The theta notation bounds a functions from above and below, so it defines exact asymptotic behavior.

A simple way to get Theta notation of an expression is to drop low order terms and ignore leading constants. For example, consider the following expression.

$$3n^3 + 6n^2 + 6000 = \Theta(n^3)$$

Dropping lower order terms is always fine because there will always be a n_0 after which $\Theta(n^3)$ has higher values than $\Theta(n^2)$ irrespective of the constants involved.

For a given function $g(n)$, we denote $\Theta(g(n))$ is following set of functions.

2) Big O Notation: The Big O notation defines an upper bound of an algorithm, it bounds a unction only from above. For example, consider the case of Insertion Sort. It takes linear time

in best case and quadratic time in worst case. We can safely say that the time complexity of Insertion sort is $O(n^2)$. Note that $O(n^2)$ also covers linear time. If we use Θ notation to represent time complexity of Insertion sort, we have to use two statements for best and worst cases:

1. The worst case time complexity of Insertion Sort is $\Theta(n^2)$.
2. The best case time complexity of Insertion Sort is $\Theta(n)$.

The Big O notation is useful when we only have upper bound on time complexity of an algorithm. Many times we easily find an upper bound by simply looking at the algorithm.

3) Ω Notation: Just as Big O notation provides an asymptotic upper bound on a function, Ω notation provides an asymptotic lower bound.

Ω Notation can be useful when we have lower bound on time complexity of an algorithm. As discussed in the previous post, the best case performance of an algorithm is generally not useful, the Omega notation is the least used notation among all three.

UNIT No 2

Q1. Explain arrays in detail

Answer :

An array is collection of items stored at contiguous memory locations. The idea is to store multiple items of same type together. This makes it easier to calculate the position of each element by simply adding an offset to a base value, i.e., the memory location of the first element of the array (generally denoted by the name of the array).

For simplicity, we can think of an array a fleet of stairs where on each step is placed a value (let's say one of your friends). Here, you can identify the location of any of your friends by simply knowing the count of the step they are on. Remember: "Location of next index depends on the data type we use".

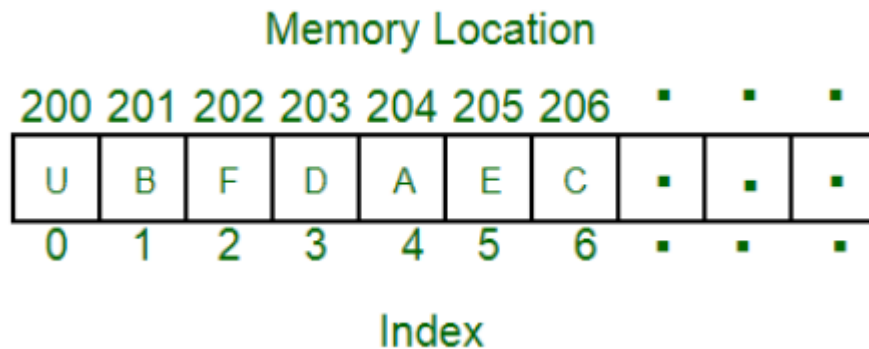


Fig. 2.1 Memory Location

The above image can be looked as a top-level view of a staircase where you are at the base of staircase. Each element can be uniquely identified by their index in the array (in

a similar way as you could identify your friends by the step on which they were on in the above example).

Types of indexing in array:

0 (zero-based indexing): The first element of the array is indexed by subscript of 0

1 (one-based indexing): The second element of the array is indexed by subscript of 1

n (n-based indexing): The base index of an array can be freely chosen. Usually programming languages allowing n-based indexing also allow negative index values and other scalar data types like enumerations, or characters may be used as an array index.

Advantages of using arrays:

Arrays allow random access of elements. This makes accessing elements by position faster. Arrays have better [cache locality](#) that can make a pretty big difference in performance. Usually, an array of characters is called a 'string', whereas an array of ints or floats is called simply an array.

Q. 2 What is Selection Sort**Answer :**

The selection sort algorithm sorts an array by repeatedly finding the minimum element (considering ascending order) from unsorted part and putting it at the beginning. The algorithm maintains two subarrays in a given array.

- 1) The subarray which is already sorted.
- 2) Remaining subarray which is unsorted.

In every iteration of selection sort, the minimum element (considering ascending order) from the unsorted subarray is picked and moved to the sorted subarray.

Following example explains the above steps:

`arr[] = 64 25 12 22 11`

```
// Find the minimum element in arr[0...4]
// and place it at beginning
11 25 12 22 64

// Find the minimum element in arr[1...4]
// and place it at beginning of arr[1...4]
11 12 25 22 64

// Find the minimum element in arr[2...4]
// and place it at beginning of arr[2...4]
11 12 22 25 64

// Find the minimum element in arr[3...4]
// and place it at beginning of arr[3...4]
11 12 22 25 64
```

Time Complexity: $O(n^2)$ as there are two nested loops.

Auxiliary Space: $O(1)$

The good thing about selection sort is it never makes more than $O(n)$ swaps and can be useful when memory write is a costly operation.

Q3. What is Bubble sort**Answer :**

Bubble Sort is the simplest sorting algorithm that works by repeatedly swapping the adjacent elements if they are in wrong order.

Example:**First Pass:**

(**5** 1 4 2 8) \rightarrow (1 **5** 4 2 8), Here, algorithm compares the first two elements, and swaps since $5 > 1$.

(1 **5** 4 2 8) \rightarrow (1 4 **5** 2 8), Swap since $5 > 4$

(1 4 **5** 2 8) \rightarrow (1 4 2 **5** 8), Swap since $5 > 2$

(1 4 2 **5** 8) \rightarrow (1 4 2 5 **8**), Now, since these elements are already in order ($8 > 5$), algorithm does not swap them.

Second Pass:

(1 **4** 2 5 8) \rightarrow (1 4 2 5 8)

(1 **4** 2 5 8) \rightarrow (1 2 **4** 5 8), Swap since $4 > 2$

(1 2 **4** 5 8) \rightarrow (1 2 4 5 8)

(1 2 4 **5** 8) \rightarrow (1 2 4 5 8)

Now, the array is already sorted, but our algorithm does not know if it is completed. The algorithm needs one **whole** pass without **any** swap to know it is sorted.

Third Pass:

(1 **2** 4 5 8) \rightarrow (1 2 4 5 8)

(1 **2** 4 5 8) \rightarrow (1 2 4 5 8)

(1 2 **4** 5 8) \rightarrow (1 2 4 5 8)

(1 2 4 **5 8**) \rightarrow (1 2 4 **5 8**)

Q4. What is insertion sort ?

Answer :

Insertion sort is a simple sorting algorithm that works the way we sort playing cards in our hands.

Algorithm

// Sort an arr[] of size n

insertionSort(arr, n)

Loop from i = 1 to n-1.

.....a) Pick element arr[i] and insert it into sorted sequence arr[0...i-1]

Example:

Another Example:

12, 11, 13, 5, 6

Let us loop for i = 1 (second element of the array) to 4 (last element of the array)

i = 1. Since 11 is smaller than 12, move 12 and insert 11 before 12

11, 12, 13, 5, 6

i = 2. 13 will remain at its position as all elements in A[0..i-1] are smaller than 13

11, 12, 13, 5, 6

i = 3. 5 will move to the beginning and all other elements from 11 to 13 will move one position ahead of their current position.

5, 11, 12, 13, 6

i = 4. 6 will move to position after 5, and elements from 11 to 13 will move one position ahead of their current position.

5, 6, 11, 12, 13

Q. 5 What is merge sort ?

Answer :

Like QuickSort, Merge Sort is a Divide and Conquer algorithm. It divides input array in two halves, calls itself for the two halves and then merges the two sorted halves. **The merge() function** is used for merging two halves. The merge(arr, l, m, r) is key process that assumes that arr[l..m] and arr[m+1..r] are sorted and merges the two sorted sub-arrays into one. See following C implementation for details.

MergeSort(arr[], l, r)

If $r > l$

1. Find the middle point to divide the array into two halves:

middle $m = (l+r)/2$

2. Call mergeSort for first half:

Call mergeSort(arr, l, m)

3. Call mergeSort for second half:

Call mergeSort(arr, m+1, r)

4. Merge the two halves sorted in step 2 and 3:

Call merge(arr, l, m, r)

for an example array {38, 27, 43, 3, 9, 82, 10}. If we take a closer look at the diagram, we can see that the array is recursively divided in two halves till the size becomes 1. Once the size becomes 1, the merge processes comes into action and starts merging arrays back till the complete array is merged.

Q6. What is binary search

Answer :

Given a sorted array `arr[]` of n elements, write a function to search a given element x in `arr[]`.

A simple approach is to do **linear search**. The time complexity of above algorithm is $O(n)$. Another approach to perform the same task is using Binary Search.

Binary Search: Search a sorted array by repeatedly dividing the search interval in half. Begin with an interval covering the whole array. If the value of the search key is less than the item in the middle of the interval, narrow the interval to the lower half. Otherwise narrow it to the upper half. Repeatedly check until the value is found or the interval is empty.



Fig. 2.2 Binary Search

Time Complexity:

The time complexity of Binary Search can be written as

$$T(n) = T(n/2) + c$$

The above recurrence can be solved either using Recurrence T ree method or Master method. It

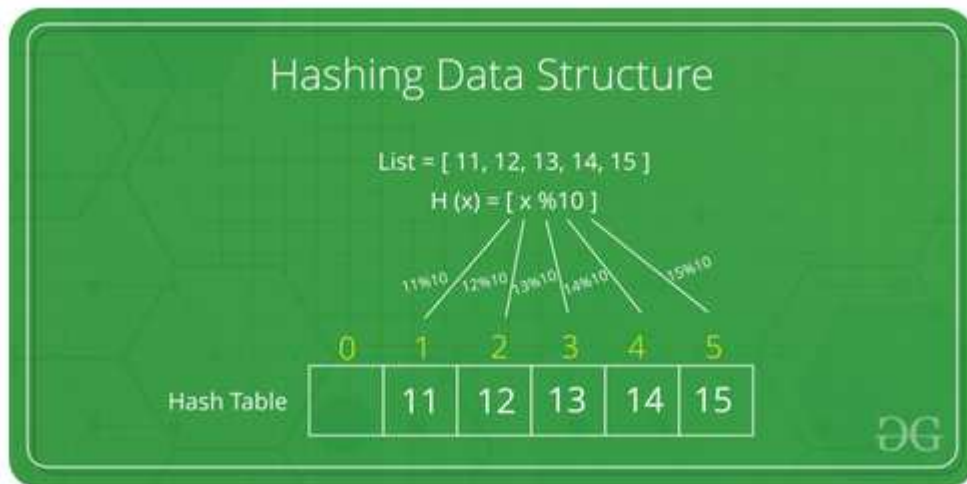
falls in case II of Master Method and solution of the recurrence is

Auxiliary Space: $O(1)$ in case of iterative implementation. In case of recursive implementation, $O(\text{Log}n)$ recursion call stack space.

Q.7 What is Hashing Data Structure**Answer :**

Hashing is an important Data Structure which is designed to use a special function called the Hash function which is used to map a given value with a particular key for faster access of elements. The efficiency of mapping depends on the efficiency of the hash function used.

Let a hash function $H(x)$ maps the value at the index $x \% 10$ in an Array. For example if the list of values is [11,12,13,14,15] it will be stored at positions {1,2,3,4,5} in the array or Hash table respectively.



Q8. What is Collision? List advantages and disadvantages**Answer :**

Since a hash function gets us a small number for a key which is a big integer or string, there is a possibility that two keys result in the same value. The situation where a newly inserted key maps to an already occupied slot in the hash table is called collision and must be handled using some collision handling technique.

Advantages:

- 1) Simple to implement.
- 2) Hash table never fills up, we can always add more elements to the chain.
- 3) Less sensitive to the hash function or load factors.
- 4) It is mostly used when it is unknown how many and how frequently keys may be inserted or deleted.

Disadvantages:

- 1) Cache performance of chaining is not good as keys are stored using a linked list. Open addressing provides better cache performance as everything is stored in the same table.
- 2) Wastage of Space (Some Parts of hash table are never used)
- 3) If the chain becomes long, then search time can become $O(n)$ in the worst case.
- 4) Uses extra space for links.

Q9. What is sparse matrix ?**Answer :**

Given two sparse matrices (Sparse Matrix and its representations | Set 1 (Using Arrays and Linked Lists)), perform operations such as add, multiply or transpose of the matrices in their sparse form itself. The result should consist of three sparse matrices, one obtained by adding the two input matrices, one by multiplying the two matrices and one obtained by transpose of the first matrix.

Example: Note that other entries of matrices will be zero as matrices are sparse.

Input :

Matrix 1: (4x4)

Row Column Value

1 2 10

1 4 12

3 3 5

4 1 15

4 2 12

Matrix 2: (4X4)

Row Column Value

1 3 8

2 4 23

3 3 9

4 1 20

4 2 25

Output :

Result of Addition: (4x4)

Row Column Value

1 2 10

1 3 8

1 4 12

2 4 23

3 3 14

4 1 35

4 2 37

Result of Multiplication: (4x4)

Row Column Value

1 1 240

1 2 300

1 4 230

3 3 45

4 3 120

4 4 276

Result of transpose on the first matrix: (4x4)

Row Column Value

1 4 15

2 1 10

2 4 12

3 3 5

4 1 12

The sparse matrix used anywhere in the program is sorted according to its row values. Two elements with the same row values are further sorted according to their column values.

Now to **Add** the matrices, we simply traverse through both matrices element by element and insert the smaller element (one with smaller row and col value) into the resultant matrix. If we come across an element with the same row and column value, we simply add their values and insert the added data into the resultant matrix.

To **Transpose** a matrix, we can simply change every column value to the row value and vice-versa, however, in this case, the resultant matrix won't be sorted as we require. Hence, we initially determine the number of elements less than the current element's column being inserted in order to get the exact index of the resultant matrix where the current element should be placed. This is done by maintaining an array `index[]` whose `ith` value indicates the number of elements in the matrix less than the column `i`.

To **Multiply** the matrices, we first calculate transpose of the second matrix to simplify our comparisons and maintain the sorted order. So, the resultant matrix is obtained by traversing through the entire length of both matrices and summing the appropriate multiplied values. Any row value equal to `x` in the first matrix and row value equal to `y` in the second matrix (transposed one) will contribute towards `result[x][y]`. This is obtained by multiplying all such

elements having col value in both matrices and adding only those with the row as x in first matrix and row as y in the second transposed matrix to get the result[x][y].

Q.10 : Sparse Matrix Representations

Answer :

If most of the elements in the matrix are zero then the matrix is called a sparse matrix. It is wasteful to store the zero elements in the matrix since they do not affect the results of our computation. This is why we implement these matrices in more efficient representations than the standard 2D Array. Using more efficient representations we can cut down space and time complexities of operations significantly.

```
Input : 0  0  0  0
        5  8  0  0
        0  0  3  0
        0  6  0  0
```

Solution: When the matrix is read row by row, the A vector is [5 8 3 6]
 The JA vector stores column indices of elements in A hence, JA = [0 1 2 1].
 $IA[0] = 0$. $IA[1] = IA[0] + \text{no of non-zero elements in row 0}$
 i.e $0 + 0 = 0$.
 Similarly,
 $IA[2] = IA[1] + 2 = 2$
 $IA[3] = IA[2] + 1 = 3$
 $IA[4] = IA[3] + 1 = 4$
 Therefore $IA = [0\ 0\ 2\ 3\ 4]$
 The trick is remember that $IA[i]$ stores NNZ upto and not-including i row.

Unit -3

Q1. What is Stack ? Write its applications

Answer :

Stack is a linear data structure which follows a particular order in which the operations are performed. The order may be LIFO (Last In First Out) or FILO (First In Last Out).

Mainly the following three basic operations are performed in the stack:

- **Push:** Adds an item in the stack. If the stack is full, then it is said to be an Overflow condition.
- **Pop:** Removes an item from the stack. The items are popped in the reversed order in which they are pushed. If the stack is empty, then it is said to be an Underflow condition.
- **Peek or Top:** Returns top element of stack.
- **isEmpty:** Returns true if stack is empty, else false.

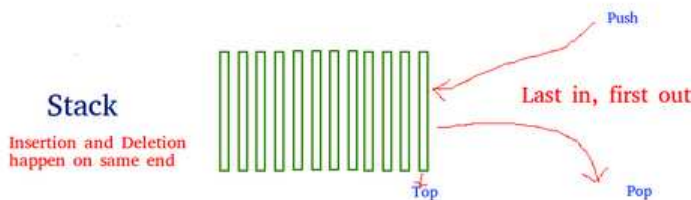


Fig. 3.1 Stack

Applications of stack:

- Balancing of symbols
- Infix to Postfix /Prefix conversion
- Redo-undo features at many places like editors, photoshop.
- Forward and backward feature in web browsers
- Used in many algorithms like Tower of Hanoi, tree traversals, stock span problem, histogram problem.
- Other applications can be Backtracking, Knight tour problem, rat in a maze, N queen problem and sudoku solver
- In Graph Algorithms like Topological Sorting and Strongly Connected Components

Q2 Queue using Stacks

The problem is opposite of this post. We are given a stack data structure with push and pop operations, the task is to implement a queue using instances of stack data structure and operations on them.

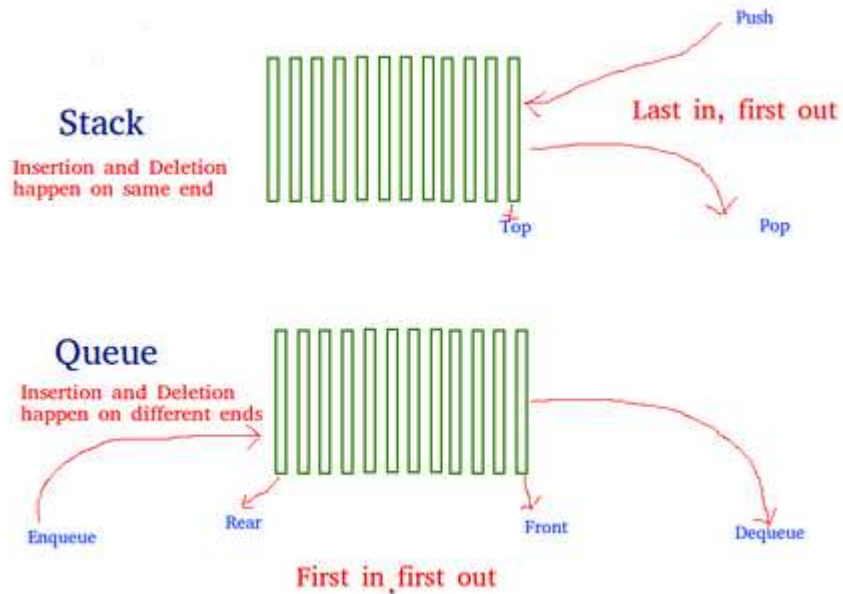


Fig. 3.2 Queue

A queue can be implemented using two stacks. Let queue to be implemented be q and stacks used to implement q be $stack1$ and $stack2$. q can be implemented in two ways:

Method 1 (By making enQueue operation costly) This method makes sure that oldest entered element is always at the top of stack 1, so that deQueue operation just pops from stack1. To put the element at top of stack1, stack2 is used.

Complexity Analysis:

- **Time Complexity:**
 - **Push operation:** $O(N)$.
In the worst case we have empty whole of stack 1 into stack 2.

- **Pop operation:** $O(1)$.
Same as pop operation in stack.
- **Auxiliary Space:** $O(N)$.
Use of stack for storing values.

Q3. Implement two stacks in an array

Answer :

Create a data structure *twoStacks* that represents two stacks. Implementation of *twoStacks* should use only one array, i.e., both stacks should use the same array for storing elements. Following functions must be supported by *twoStacks*.

push1(int x) → pushes x to first stack
push2(int x) → pushes x to second stack

pop1() → pops an element from first stack and return the popped element
pop2() → pops an element from second stack and return the popped element

Implementation of *twoStack* should be space efficient.

The problem with this method is inefficient use of array space. A stack push operation may result in stack overflow even if there is space available in arr[]. For example, say the array size is 6 and we push 3 elements to stack1 and do not push anything to second stack2. When we push 4th element to stack1, there will be overflow even if we have space for 3 more elements in array.

Q4. How to implement stack using priority queue or heap?**Answer :**

In priority queue, we assign priority to the elements that are being pushed. A stack requires elements to be processed in Last in First Out manner. The idea is to associate a count that determines when it was pushed. This count works as a key for the priority queue. So the implementation of stack uses a priority queue of pairs, with the first element serving as the key.

Q5. Implement Stack using Queues**Answer :**

We are given a Queue data structure that supports standard operations like enqueue() and dequeue(). We need to implement a Stack data structure using only instances of Queue and queue operations allowed on the instances.

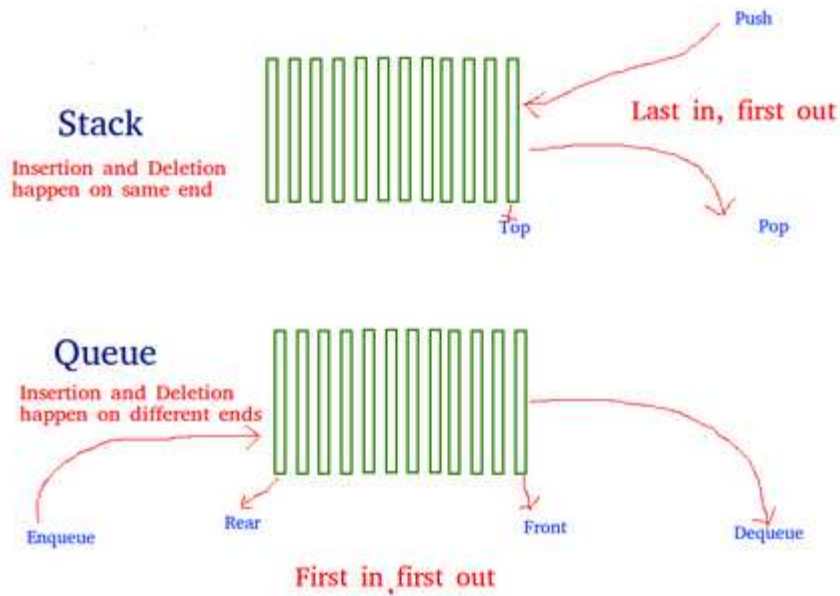


Fig.3.3 Queue Operation

A stack can be implemented using two queues. Let stack to be implemented be 's' and queues used to implement be 'q1' and 'q2'. Stack 's' can be implemented in two ways:

Method 1 (By making push operation costly)

This method makes sure that newly entered element is always at the front of 'q1', so that pop operation just dequeues from 'q1'. 'q2' is used to put every new element at front of 'q1'.

1. **push(s, x)** operation's step are described below:

- Enqueue x to q2
 - One by one dequeue everything from q1 and enqueue to q2.
 - Swap the names of q1 and q2
2. **pop(s)** operation's function are described below:
- Dequeue an item from q1 and return it.

Method 2 (By making pop operation costly)

In push operation, the new element is always enqueued to q1. In pop() operation, if q2 is empty then all the elements except the last, are moved to q2. Finally the last element is dequeued from q1 and returned.

1. **push(s, x)** operation:
- Enqueue x to q1 (assuming size of q1 is unlimited).
2. **pop(s)** operation:
- One by one dequeue everything except the last element from q1 and enqueue to q2.
 - Dequeue the last item of q1, the dequeued item is result, store it.
 - Swap the names of q1 and q2
 - Return the item stored in step 2.

Q6 List Different Types of Queues and its Applications**Answer :**

A Queue is a linear structure which follows a particular order in which the operations are performed. The order is First In First Out (FIFO). A good example of a queue is any queue of consumers for a resource where the consumer that came first is served first. In this article, the different types of queues are discussed.

The queue is used when things don't have to be processed immediately, but have to be processed in First In First Out order like Breadth First Search. This property of Queue makes it also useful in the following kind of scenarios.

1. When a resource is shared among multiple consumers. Examples include CPU scheduling, Disk Scheduling.
2. When data is transferred asynchronously (data not necessarily received at the same rate as sent) between two processes. Examples include IO Buffers, pipes, file IO, etc.

There are five different types of queues which are used in different scenarios. They are:

1. **Circular Queue:** Circular Queue is a linear data structure in which the operations are performed based on FIFO (First In First Out) principle and the last position is connected back to the first position to make a circle. It is also called '**Ring Buffer**'. This queue is primarily used in the following cases:

1. **Memory Management:** The unused memory locations in the case of ordinary queues can be utilized in circular queues.
 2. **Traffic system:** In a computer-controlled traffic system, circular queues are used to switch on the traffic lights one by one repeatedly as per the time set.
 3. **CPU Scheduling:** Operating systems often maintain a queue of processes that are ready to execute or that are waiting for a particular event to occur.
2. **Input restricted Queue:** In this type of Queue, the input can be taken from one side only(rear) and deletion of element can be done from both side(front and rear). This kind of Queue does not follow FIFO(first in first out).

Q7 What is circular queue? Explain in detail

Answer :

Circular Queue is a linear data structure in which the operations are performed based on FIFO (First In First Out) principle and the last position is connected back to the first position to make a circle. It is also called '**Ring Buffer**'.

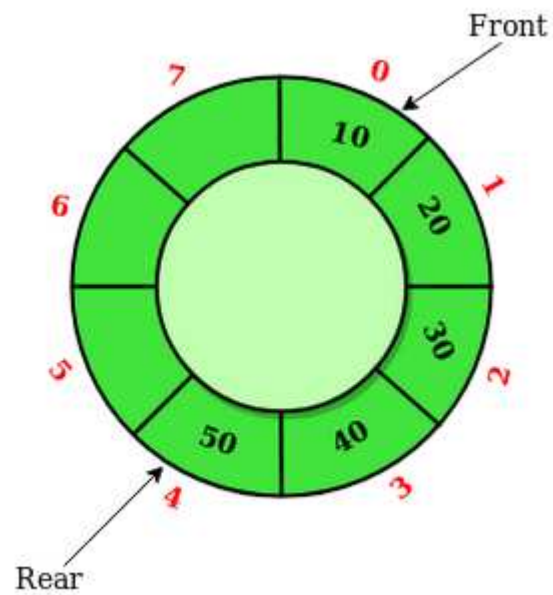


Fig. 3.4 Circular Queue

In a normal Queue, we can insert elements until queue becomes full. But once queue becomes full, we can not insert the next element even if there is a space in front of queue

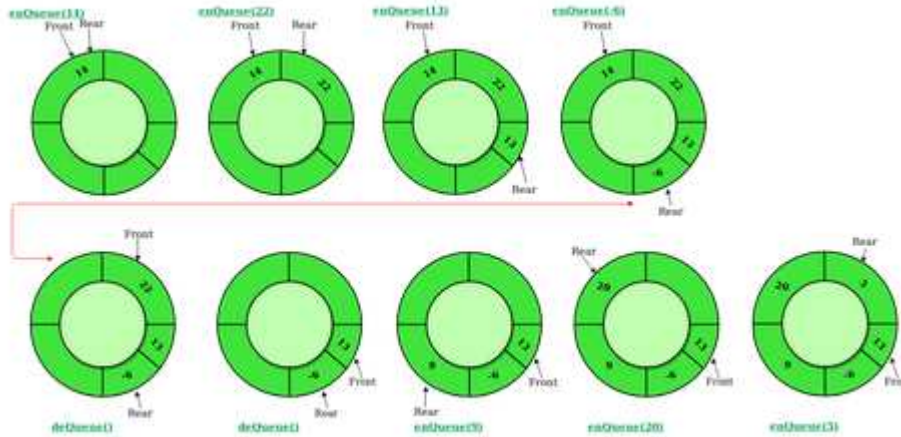


Fig.3.5 Circular Queue operation

Operations on Circular Queue:

- **Front:** Get the front item from queue.
- **Rear:** Get the last item from queue.
- **enQueue(value)** This function is used to insert an element into the circular queue. In a circular queue, the new element is always inserted at Rear position.

Steps:

1. Check whether queue is Full – Check $((\text{rear} == \text{SIZE}-1 \ \&\& \ \text{front} == 0) \ || \ (\text{rear} == \text{front}-1))$.
 2. If it is full then display Queue is full. If queue is not full then, check if $(\text{rear} == \text{SIZE} - 1 \ \&\& \ \text{front} != 0)$ if it is true then set $\text{rear}=0$ and insert element.
- **deQueue()** This function is used to delete an element from the circular queue. In a circular queue, the element is always deleted from front position.

Steps:

1. Check whether queue is Empty means check $(\text{front} == -1)$.
2. If it is empty then display Queue is empty. If queue is not empty then step 3
3. Check if $(\text{front} == \text{rear})$ if it is true then set $\text{front} = \text{rear} = -1$ else check if $(\text{front} == \text{size}-1)$, if it is true then set $\text{front}=0$ and return the element.

Q8. How dequeue is implemented ?

Answer :

Deque or Double Ended Queue is a generalized version of Queue data structure that allows insert and delete at both ends. In previous post we had discussed introduction of deque. Now in this post we see how we implement deque Using circular array.

Operations on Deque:

Mainly the following four basic operations are performed on queue:

insetFront(): Adds an item at the front of Deque.

insertRear(): Adds an item at the rear of Deque.

deleteFront(): Deletes an item from front of Deque.

deleteRear(): Deletes an item from rear of Deque.

In addition to above operations, following operations are also supported

getFront(): Gets the front item from queue.

getRear(): Gets the last item from queue.

isEmpty(): Checks whether Deque is empty or not.

isFull(): Checks whether Deque is full or not.

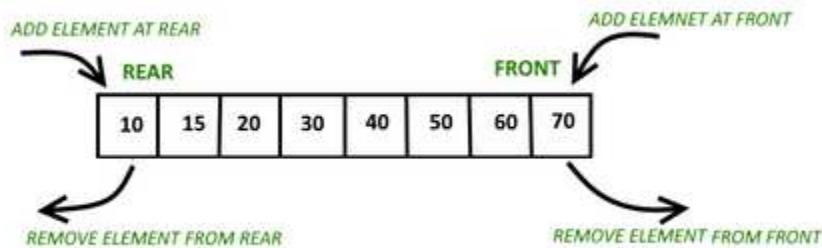


Fig.3.6 Queue Operation

Q9. How Implementation of Deque using doubly linked list**Answer :**

Deque or Double Ended Queue is a generalized version of Queue data structure that allows insert and delete at both ends.

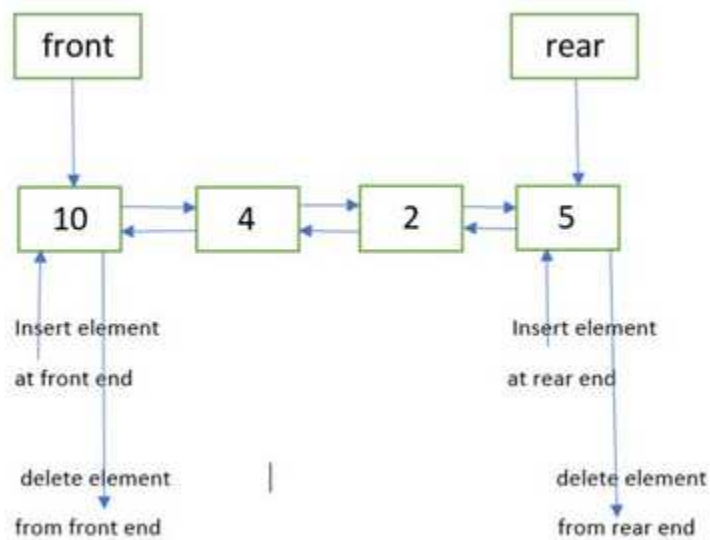


Fig. 3.7 : Deque

Doubly Linked List Representation of Deque :

For implementing deque, we need to keep track of two pointers, **front** and **rear**. We **enqueue (push)** an item at the rear or the front end of deque and **dequeue(pop)** an item from both rear and front end.

Working :

Declare two pointers **front** and **rear** of type **Node**, where **Node** represents the structure of a node of a doubly linked list. Initialize both of them with value NULL.

Q10. Evaluation of expression

Evaluate an expression represented by a String. Expression can contain parentheses, you can assume parentheses are well-matched. For simplicity, you can assume only binary operations allowed are +, -, *, and /. Arithmetic Expressions can be written in one of three forms:

Infix Notation: Operators are written between the operands they operate on, e.g. $3 + 4$.

Prefix Notation: Operators are written before the operands, e.g. $+ 3 4$

Postfix Notation: Operators are written after operands.

Infix Expressions are harder for Computers to evaluate because of the additional work needed to decide precedence. Infix notation is how expressions are written and recognized by humans and, generally, input to programs. Given that they are harder to evaluate, they are generally converted to one of the two remaining forms. A very well known algorithm for converting an infix notation to a postfix notation is Shunting Yard Algorithm by Edgar Dijkstra. This

algorithm takes as input an Infix Expression and produces a queue that has this expression converted to a postfix notation. Same algorithm can be modified so that it outputs result of evaluation of expression instead of a queue.

Unit -4

Q1 : Explain in detail singly linked list

Answer :

A linked list is a linear data structure, in which the elements are not stored at contiguous memory locations. The elements in a linked list are linked using pointers as shown in the below image:

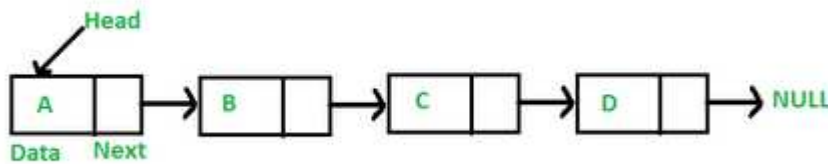


Fig. 4.1 Linked List

In simple words, a linked list consists of nodes where each node contains a data field and a reference(link) to the next node in the list.

Q 2 : List Key Differences Between Array and Linked List

Answer :

1. An array is the data structure that contains a collection of similar type data elements whereas the Linked list is considered as non-primitive data structure contains a collection of unordered linked elements known as nodes.
2. In the array the elements belong to indexes, i.e., if you want to get into the fourth element you have to write the variable name with its index or location within the square bracket.
3. In a linked list though, you have to start from the head and work your way through until you get to the fourth element.
4. Accessing an element in an array is fast, while Linked list takes linear time, so it is quite a bit slower.
5. Operations like insertion and deletion in arrays consume a lot of time. On the other hand, the performance of these operations in Linked lists is fast.
6. Arrays are of fixed size. In contrast, Linked lists are dynamic and flexible and can expand and contract its size.
7. In an array, memory is assigned during compile time while in a Linked list it is allocated during execution or runtime.
9. Elements are stored consecutively in arrays whereas it is stored randomly in Linked lists.
10. The requirement of memory is less due to actual data being stored within the

index in the array. As against, there is a need for more memory in Linked Lists due to storage of additional next and previous referencing elements.

11. In addition memory utilization is inefficient in the array. Conversely, memory utilization is efficient in the linked list.

Advantages of Linked list as compared to arrays

- (1) The size of the arrays is fixed: So we must know the upper limit on the number of elements in advance. Also, generally, the allocated memory is equal to the upper limit irrespective of the usage, and in practical uses, the upper limit is rarely reached.
3. (2) Inserting a new element in an array of elements is expensive because a room has to be created for the new elements and to create room existing elements have to be shifted.
4. For example, suppose we maintain a sorted list of IDs in an array `id[]`.
5. `id[] = [1000, 1010, 1050, 2000, 2040,]`.
6. And if we want to insert a new ID 1005, then to maintain the sorted order, we have to move all the elements after 1000 (excluding 1000).
7. Deletion is also expensive with arrays until unless some special techniques are used. For example, to delete 1010 in `id[]`, everything after 1010 has to be moved.
8. So Linked list provides the following two advantages over arrays
 - 1) Dynamic size
 - 2) Ease of insertion/deletion

Linked lists have following drawbacks:

- 1) Random access is not allowed. We have to access elements sequentially starting from the first node. So we cannot do a binary search with linked lists.
- 2) Extra memory space for a pointer is required with each element of the list.
- 3) Arrays have better cache locality that can make a pretty big difference in performance.

Q3 : Difference between Singly linked list and Doubly linked list

Answer :

Introduction to Singly linked list : A singly linked list is a set of nodes where each node has two fields 'data' and 'link'. The 'data' field stores actual piece of information and 'link' field is used to point to next node. Basically 'link' field is nothing but address only.

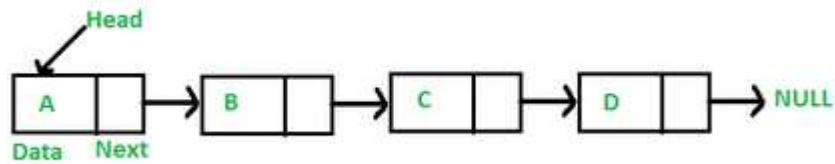


Fig. 4.2 Singly Linked List

Introduction to Doubly linked list : A **Doubly Linked List (DLL)** contains an extra pointer, typically called *previous pointer*, together with next pointer and data which are there in singly linked list.

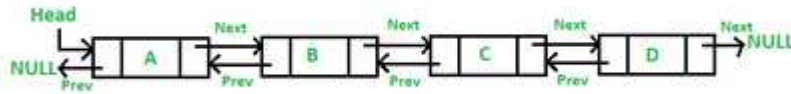


Fig. 4.3 Doubly Linked List

Singly linked list vs Doubly linked list

Singly linked list (SLL)	Doubly linked list (DLL)
SLL has nodes with only a data field and next link field.	DLL has nodes with a data field, a previous link field and a next link field.
In SLL, the traversal can be done using the next node link only.	In DLL, the traversal can be done using the previous node link or the next node link.
The SLL occupies less memory than DLL as it has only 2 fields.	The DLL occupies more memory than SLL as it has 3 fields.
Less efficient access to elements.	More efficient access to elements.

Q4 . major differences between a Static Queue and a Singly Linked List

Answer :

Static Queue: A queue is an ordered list of elements. It always works in first in first out(FIFO) fashion. All the elements get inserted at the *REAR* and removed from the

FRONT of the queue. In implementation of the static Queue, an array will be used so all operation of queue are index based which makes it faster for all operations except deletion because deletion requires shifting of all the remaining elements to the front by one position.

A **Static Queue** is a queue of fixed size implemented using array.

Singly Linked List: A linked list is also an ordered list of elements. You can add an element anywhere in the list, change an element anywhere in the list, or remove an element from any position in the list. Each node in the list stores the content and a pointer or reference to the next node in the list. To store a single linked list, only the reference or pointer to the first node in that list must be stored. The last node in a single linked list points to nothing (or null).

Static Queue	Singly Linked List
Queue is a collection of one or more elements arranged in memory in a contiguous fashion.	A linked list is a collection of one or more elements arranged in memory in a dis-contiguous fashion.
Static Queue is always fixed size.	List size is never fixed.
In Queue, only one and single type of information is stored because static Queue implementation is through Array.	List also stored the address for the next node along with it's content.

Static Queue is index based.	Singly linked list is reference based.
Insertion can always be performed on a single end called <i>REAR</i> and deletion on the other end called <i>FRONT</i> .	Insertion as well as deletion can be performed anywhere within the list.
Queue is always based on FIFO.	List may be based on FIFO or LIFO etc.
Queue has two pointers FRONT and REAR.	While List has only one pointer basically called HEAD.

Q5 : What is Garbage collection ?

Answer :

Garbage collection comes under memory management. It is systematic recovery of storage which was earlier being used but now it is no longer needed. Garbage collection is an automatic memory management feature in many modern programming languages.

When the program has no more references to that Object, the Object's memory becomes unreachable, but it is not immediately freed. The Garbage Collection checks to see if

there are any Objects in the heap that are no longer being used by the application. In C , we use free() function and in C++ delete() function is used.

Advantages:-

- 1) It makes our system memory efficient by de-referring process which are no longer in use.
- 2) No requirement of extra application for cleaning memory.

Garbage collection means, if a variable is not used then it is automatically deleted from the memory, meaning that memory allocated to that variable is deleted. In java the garbage collector does this. In computer science, **garbage collection** (GC) is a form of automatic memory management. The **garbage collector**, or just **collector**, attempts to reclaim **garbage**, or memory occupied by objects that are no longer in use by the program.

Q6 . What is compaction ?

Answer :

- Compaction is a process in which the free space is collected in a large memory chunk to make some space available for processes.

- Compaction attacks the problem of fragmentation by moving all the allocated blocks to one end of memory, thus combining all the holes.
- Compaction is used to reduce the external fragmentation. Shuffle memory content to place all free memory together in one large block. Compaction is only possible if the relocation is dynamic
- Compaction refers to combining all the empty spaces together and processes.
- It solve the problem of fragmentation, but it requires too much of CPU time.
- It moves all the occupied areas of store to one end and leaves one large free space for incoming jobs, instead of numerous small ones
- In compaction, the system also maintains relocation information and it must be performed on each new allocation of job to the memory or completion of job from memory.
- Goal of compaction is to have effective algorithm and make it cost effective.

Q7. Adding two polynomials using Linked List

Answer :

Given two polynomial numbers represented by a linked list. Write a function that add these lists means add the coefficients who have same variable powers.

Input:

$$\text{1st number} = 5x^2 + 4x^1 + 2x^0$$

$$\text{2nd number} = 5x^1 + 5x^0$$

Output:

$$5x^2 + 9x^1 + 7x^0$$

Input:

$$\text{1st number} = 5x^3 + 4x^2 + 2x^0$$

$$\text{2nd number} = 5x^1 + 5x^0$$

Output:

$$5x^3 + 4x^2 + 5x^1 + 7x^0$$

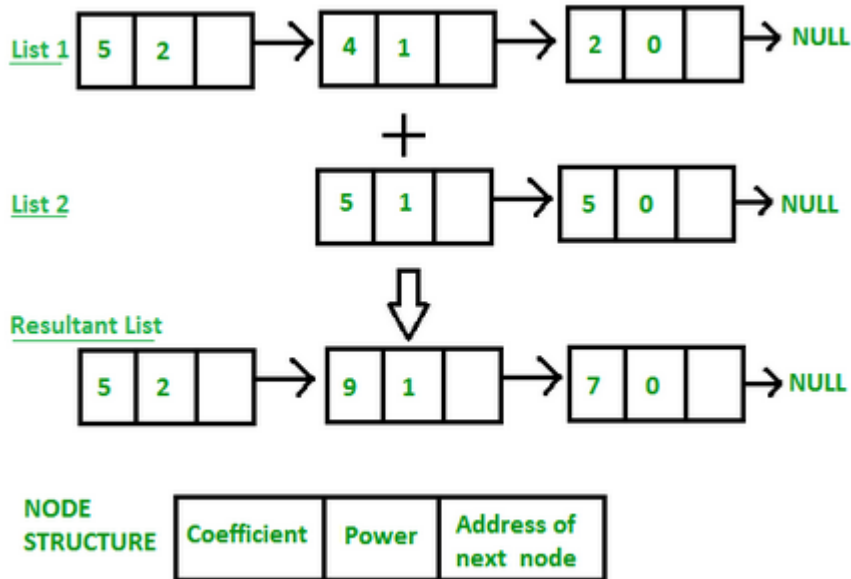


Fig. 4.5 : Representation

Q8. What is Dynamic Memory Allocation?**Answer :**

Resources are always a premium. We have strived to achieve better utilization of resources at all times; that is the premise of our progress. Related to this pursuit, is the concept of memory allocation.

Memory has to be allocated to the variables that we create, so that actual variables can be brought to existence. Now there is a constraint as how we think it happens, and how it actually happens.

When we think of creating something, we think of creating something from the very scratch, while this isn't what actually happens when a computer creates a variable 'X'; to the computer, is more like an allocation, the computer just assigns a memory cell from a lot of pre-existing memory cells to X. It's like someone named 'RAJESH' being allocated to a hotel room from a lot of free or empty pre-existing rooms. This example probably made it very clear as how the computer does the allocation of memory.

Now, what is **Static Memory Allocation**? When we declare variables, we actually are preparing all the variables that will be used, so that the compiler knows that the variable being used is actually an important part of the program that the user wants and

not just a rogue symbol floating around. So, when we declare variables, what the compiler actually does is allocate those variables to their rooms (refer to the hotel analogy earlier). Now, if you see, this is being done before the program executes, you can't allocate variables by this method while the program is executing.

Q9. What is Generalized Linked List ?

Answer :

A Generalized Linked List L , is defined as a finite sequence of $n \geq 0$ elements, $l_1, l_2, l_3, l_4, \dots, l_n$, such that l_i are either atom or the list of atoms. Thus

$$L = (l_1, l_2, l_3, l_4, \dots, l_n)$$

where n is total number of nodes in the list.

Flag	Data	Down pointer	Next pointer
------	------	--------------	--------------

To represent a list of items there are certain assumptions about the node structure.

- Flag = 1 implies that *down pointer* exists
- Flag = 0 implies that *next pointer* exists
- Data means the atom
- Down pointer is the address of node which is down of the current node
- Next pointer is the address of node which is attached as the next node

Generalized linked lists are used because although the efficiency of polynomial operations using linked list is good but still, the disadvantage is that the linked list is unable to use *multiple variable polynomial equation* efficiently. It helps us to represent multi-variable polynomial along with the list of elements.

Q10. Applications of linked list in computer science –

Answer :

Implementation of stacks and queues

1. Implementation of graphs : Adjacency list representation of graphs is most popular which is uses linked list to store adjacent vertices.
2. Dynamic memory allocation : We use linked list of free blocks.
3. Maintaining directory of names
4. Performing arithmetic operations on long integers
5. Manipulation of polynomials by storing constants in the node of linked list
6. representing sparse matrices

Applications of linked list in real world-

1. *Image viewer* – Previous and next images are linked, hence can be accessed by next and previous button.

2. *Previous and next page in web browser* – We can access previous and next url searched in web browser by pressing back and next button since, they are linked as linked list.
3. *Music Player* – Songs in music player are linked to previous and next song. you can play songs either from starting or ending of the list.

Applications of Circular Linked Lists:

1. Useful for implementation of queue. Unlike this implementation, we don't need to maintain two pointers for front and rear if we use circular linked list. We can maintain a pointer to the last inserted node and front can always be obtained as next of last.
2. Circular lists are useful in applications to repeatedly go around the list. For example, when multiple applications are running on a PC, it is common for the operating system to put the running applications on a list and then to cycle through them, giving each of them a slice of time to execute, and then making them wait while the CPU is given to another application. It is convenient for the operating system to use a circular list so that when it reaches the end of the list it can cycle around to the front of the list.
3. Circular Doubly Linked Lists are used for implementation of advanced data structures like Fibonacci Heap.

Unit – 5

Q1. What is tree ? Explain in detail

Answer :

Trees: Unlike Arrays, Linked Lists, Stack and queues, which are linear data structures, trees are hierarchical data structures.

Tree Vocabulary: The topmost node is called root of the tree. The elements that are directly under an element are called its children. The element directly above something is called its parent. For example, 'a' is a child of 'f', and 'f' is the parent of 'a'. Finally, elements with no children are called leaves.

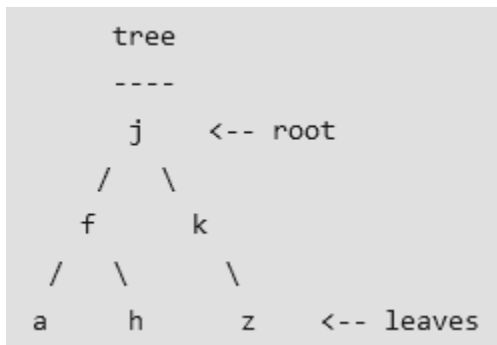


Fig. 5.1 Trees

Why Trees?

1. One reason to use trees might be because you want to store information that naturally forms a hierarchy. For example, the file system on a computer:

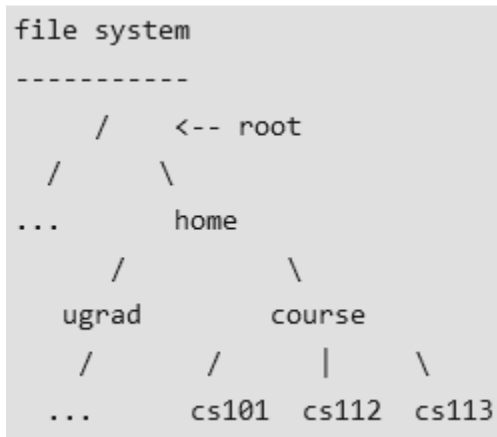


Fig. 5.2 Trees Operations

2. Trees (with some ordering e.g., BST) provide moderate access/search (quicker than Linked List and slower than arrays).
3. Trees provide moderate insertion/deletion (quicker than Arrays and slower than Unordered Linked Lists).
4. Like Linked Lists and unlike Arrays, Trees don't have an upper limit on number of nodes as nodes are linked using pointers.

Main applications of trees include:

1. Manipulate hierarchical data.
2. Make information easy to search (see tree traversal).

3. Manipulate sorted lists of data.
4. As a workflow for compositing digital images for visual effects.
5. Router algorithms
6. Form of a multi-stage decision-making (see business chess).

Q2. Tree Traversals (Inorder, Preorder and Postorder)

Unlike linear data structures (Array, Linked List, Queues, Stacks, etc) which have only one logical way to traverse them, trees can be traversed in different ways. Following are the generally used ways for traversing trees.

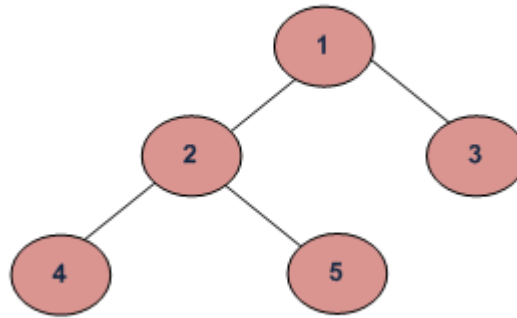


Fig. 5.3 Tree Traversal

Depth First Traversals:

- (a) Inorder (Left, Root, Right) : 4 2 5 1 3
- (b) Preorder (Root, Left, Right) : 1 2 4 5 3
- (c) Postorder (Left, Right, Root) : 4 5 2 3 1

Breadth First or Level Order Traversal : 1 2 3 4 5

Q3. Applications of tree data structures

Answer :

Unlike Array and Linked List, which are linear data structures, tree is hierarchical (or non-linear) data structure.

1. One reason to use trees might be because you want to store information that naturally forms a hierarchy. For example, the file system on a computer:
2. ☐ If we organize keys in form of a tree (with some ordering e.g., BST), we can search for a given key in moderate time (quicker than Linked List and slower than arrays). Self-balancing search trees like AVL and Red-Black trees guarantee an upper bound of $O(\log n)$ for search.
3. We can insert/delete keys in moderate time (quicker than Arrays and slower than Unordered Linked Lists). Self-balancing search trees like AVL and Red-Black trees guarantee an upper bound of $O(\log n)$ for insertion/deletion.
4. Like Linked Lists and unlike Arrays, Pointer implementation of trees don't have an upper limit on number of nodes as nodes are linked using pointers.

Other Applications :

1. Store hierarchical data, like folder structure, organization structure, XML/HTML data.
2. Binary Search Tree is a tree that allows fast search, insert, delete on a sorted data. It also allows finding closest item

3. Heap is a tree data structure which is implemented using arrays and used to implement priority queues.
4. B-Tree and B+ Tree : They are used to implement indexing in databases.
5. Syntax Tree: Used in Compilers.
6. K-D Tree: A space partitioning tree used to organize points in K dimensional space.
7. Trie : Used to implement dictionaries with prefix lookup.
8. Suffix Tree : For quick pattern searching in a fixed text.
9. Spanning Trees and shortest path trees are used in routers and bridges respectively in computer networks
10. As a workflow for compositing digital images for visual effects.

Q4. What is b+ tree

Answer :

In order, to implement dynamic multilevel indexing, B-tree and B+ tree are generally employed. The drawback of B-tree used for indexing, however is that it stores the data pointer (a pointer to the disk file block containing the key value), corresponding to a particular key value, along with that key value in the node of a B-tree. This technique, greatly reduces the number of entries that can be packed into a node of a B-tree, thereby contributing to the increase in the number of levels in the B-tree, hence increasing the search time of a record.

B+ tree eliminates the above drawback by storing data pointers only at the leaf nodes of the tree. Thus, the structure of leaf nodes of a B+ tree is quite different from the structure of

internal nodes of the B tree. It may be noted here that, since data pointers are present only at the leaf nodes, the leaf nodes must necessarily store all the key values along with their corresponding data pointers to the disk file block, in order to access them. Moreover, the leaf nodes are linked to provide ordered access to the records. The leaf nodes, therefore form the first level of index, with the internal nodes forming the other levels of a multilevel index. Some of the key values of the leaf nodes also appear in the internal nodes, to simply act as a medium to control the searching of a record.

From the above discussion it is apparent that a B+ tree, unlike a B-tree has two orders, 'a' and 'b', one for the internal nodes and the other for the external (or leaf) nodes.

The structure of the internal nodes of a B+ tree of order 'a' is as follows:

1. Each internal node is of the form :

$$\langle P_1, K_1, P_2, K_2, \dots, P_{c-1}, K_{c-1}, P_c \rangle$$
 where $c \leq a$ and each P_i is a **tree pointer (i.e points to another node of the tree)** and, each K_i is a **key value** (see diagram-I for reference).
2. Every internal node has : $K_1 < K_2 < \dots < K_{c-1}$
3. For each search field values 'X' in the sub-tree pointed at by P_i , the following condition holds :

$$K_{i-1} < X \leq K_i, \text{ for } 1 < i < c \text{ and,}$$

$$K_{i-1} < X, \text{ for } i = c$$
 (See diagram I for reference)
4. Each internal nodes has at most 'a' tree pointers.

5. The root node has, at least two tree pointers, while the other internal nodes have at least $\lceil a/2 \rceil$ tree pointers each.
6. If any internal node has 'c' pointers, $c \leq a$, then it has 'c - 1' key values.

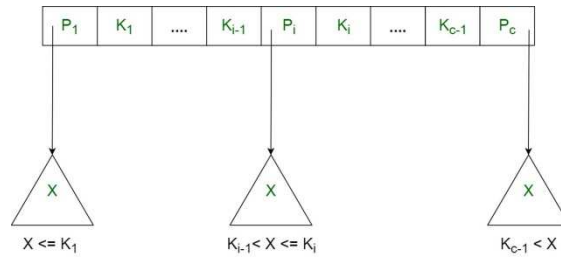


Fig Trees

Q5 : What is AVL tree ? Where it is used

Answer :

AVL tree is a self-balancing Binary Search Tree (BST) where the difference between heights of left and right subtrees cannot be more than one for all nodes.

An Example Tree that is an AVL Tree

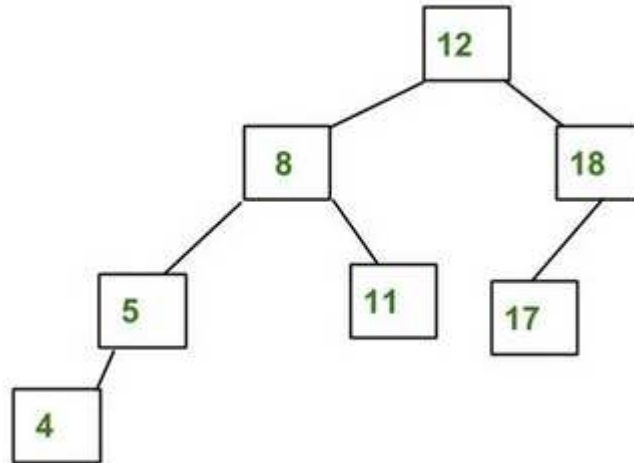


Fig. 5.5 Tree

The above tree is AVL because differences between heights of left and right subtrees for every node is less than or equal to 1.

An Example Tree that is NOT an AVL Tree

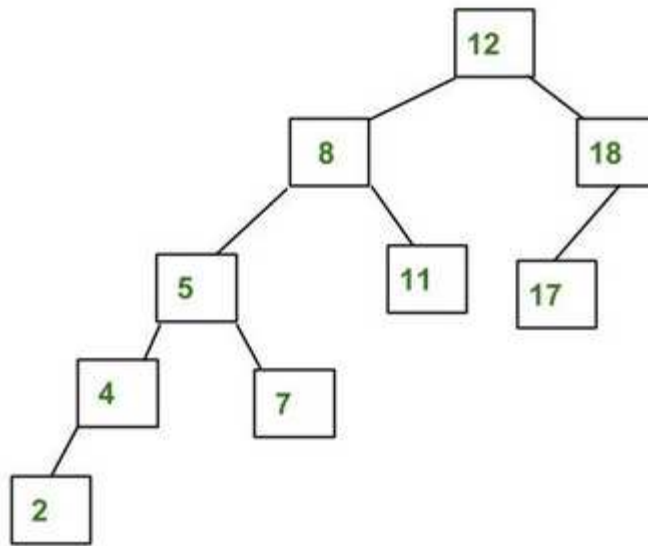


Fig. 5.6 Trees

The above tree is not AVL because differences between heights of left and right subtrees for 8 and 12 is greater than 1.

Why AVL Trees?

Answer :

Most of the BST operations (e.g., search, max, min, insert, delete.. etc) take $O(h)$ time where h is the height of the BST. The cost of these operations may become $O(n)$ for a skewed Binary tree. If we make sure that height of the tree remains $O(\log n)$ after every insertion and deletion, then we can guarantee an upper bound of $O(\log n)$ for all these operations. The height of an AVL tree is always $O(\log n)$ where n is the number of nodes in the tree.

Insertion

To make sure that the given tree remains AVL after every insertion, we must augment the standard BST insert operation to perform some re-balancing. Following are two basic operations that can be performed to re-balance a BST without violating the BST property ($\text{keys}(\text{left}) < \text{key}(\text{root}) < \text{keys}(\text{right})$).

- 1) Left Rotation
- 2) Right Rotation

Q6. How to determine if a binary tree is height-balanced?**Answer :**

A tree where no leaf is much farther away from the root than any other leaf. Different balancing schemes allow different definitions of “much farther” and different amounts of work to keep them balanced.

Consider a height-balancing scheme where following conditions should be checked to determine if a binary tree is balanced.

An empty tree is height-balanced. A non-empty binary tree T is balanced if:

- 1) Left subtree of T is balanced
- 2) Right subtree of T is balanced
- 3) The difference between heights of left subtree and right subtree is not more than 1.

The above height-balancing scheme is used in AVL trees. The diagram below shows two trees, one of them is height-balanced and other is not. The second tree is not height-balanced because height of left subtree is 2 more than height of right subtree.

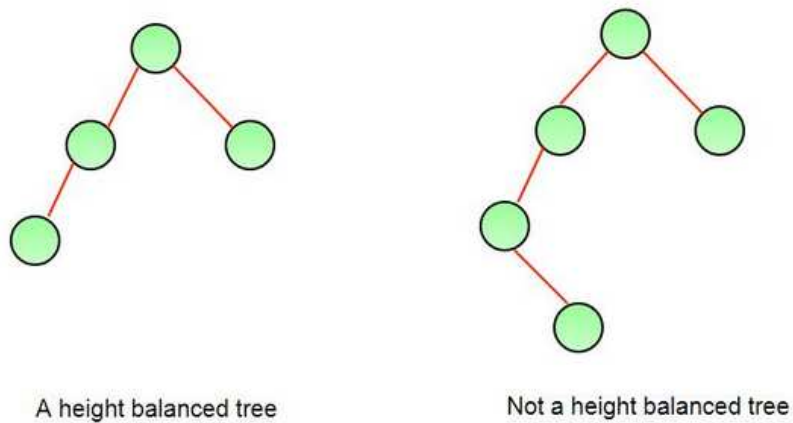


Fig. 5.7 Balanced Trees

Q7. What is heap sort ? explain in detail**Answer :**

Heap sort is a comparison based sorting technique based on Binary Heap data structure. It is similar to selection sort where we first find the maximum element and place the maximum element at the end. We repeat the same process for remaining element.

A Binary Heap is a Complete Binary Tree where items are stored in a special order such that value in a parent node is greater(or smaller) than the values in its two children nodes. The former is called as max heap and the latter is called min heap. The heap can be represented by binary tree or array.

Since a Binary Heap is a Complete Binary Tree, it can be easily represented as array and array based representation is space efficient. If the parent node is stored at index I, the left child

can be calculated by $2 * I + 1$ and right child by $2 * I + 2$ (assuming the indexing starts at 0).

Heap Sort Algorithm for sorting in increasing order:

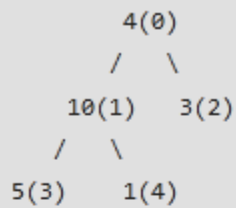
1. Build a max heap from the input data.
2. At this point, the largest item is stored at the root of the heap. Replace it with the last item of the heap followed by reducing the size of heap by 1. Finally, heapify the root of tree.
3. Repeat above steps while size of heap is greater than 1.

Q8. How to build the heap?

Heapify procedure can be applied to a node only if its children nodes are heapified. So the heapification must be performed in the bottom up order.

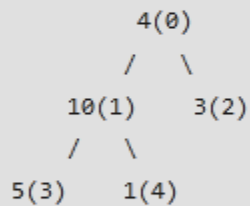
Lets understand with the help of an example:

Input data: 4, 10, 3, 5, 1

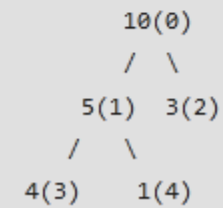


The numbers in bracket represent the indices in the array representation of data.

Applying heapify procedure to index 1:



Applying heapify procedure to index 0:



The heapify procedure calls itself recursively to build heap in top down manner.

Q9. what is Threaded Binary Tree

Inorder traversal of a Binary tree can either be done using recursion or with the use of a auxiliary stack. The idea of threaded binary trees is to make inorder traversal faster and do it without stack and without recursion. A binary tree is made threaded by making all right child pointers that would normally be NULL point to the inorder successor of the node (if it exists).

There are two types of threaded binary trees.

Single Threaded: Where a NULL right pointers is made to point to the inorder successor (if successor exists)

Double Threaded: Where both left and right NULL pointers are made to point to inorder predecessor and inorder successor respectively. The predecessor threads are useful for reverse inorder traversal and postorder traversal.

The threads are also useful for fast accessing ancestors of a node.

Following diagram shows an example Single Threaded Binary Tree. The dotted lines represent threads.

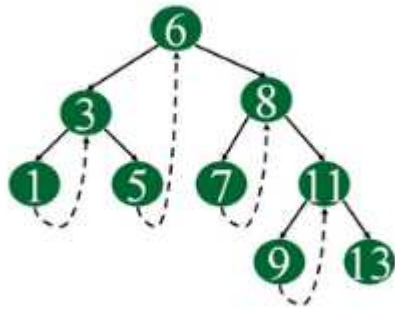


Fig. 5.8 : Threaded Trees

Q.10 : list various tree traversal methods

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.

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.

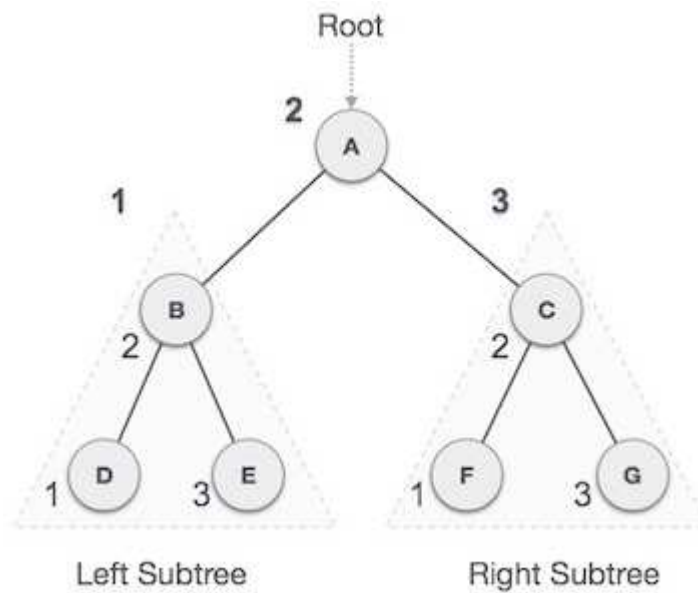


Fig. 5.10 : trees with subtrees

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 inorder traversal of this tree will be –

$$D \rightarrow B \rightarrow E \rightarrow A \rightarrow F \rightarrow C \rightarrow G$$

Algorithm

Until all nodes are traversed –

Step 1 – Recursively traverse left subtree.

Step 2 – Visit root node.

Step 3 – Recursively traverse right subtree.

Pre-order Traversal

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

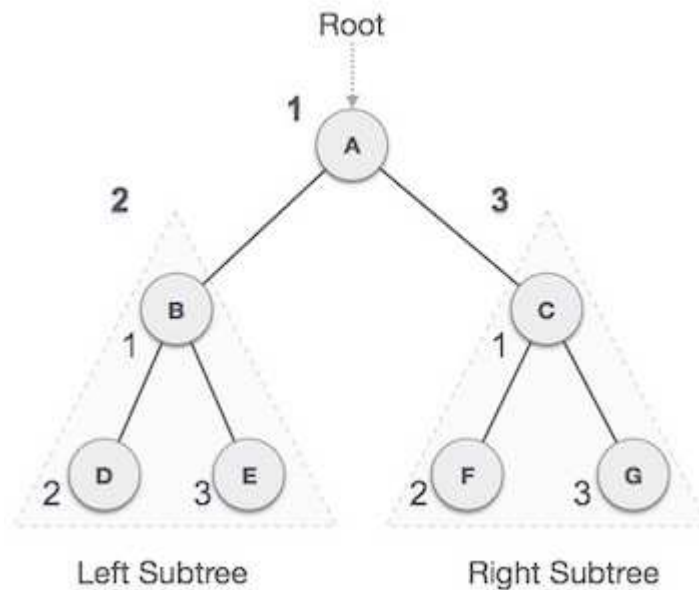


Fig. 5.11 Trees with subtrees

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 –

$$A \rightarrow B \rightarrow D \rightarrow E \rightarrow C \rightarrow F \rightarrow G$$

Algorithm

Until all nodes are traversed -

Step 1 - Visit root node.

Step 2 - Recursively traverse left subtree.

Step 3 - Recursively traverse right subtree.

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.

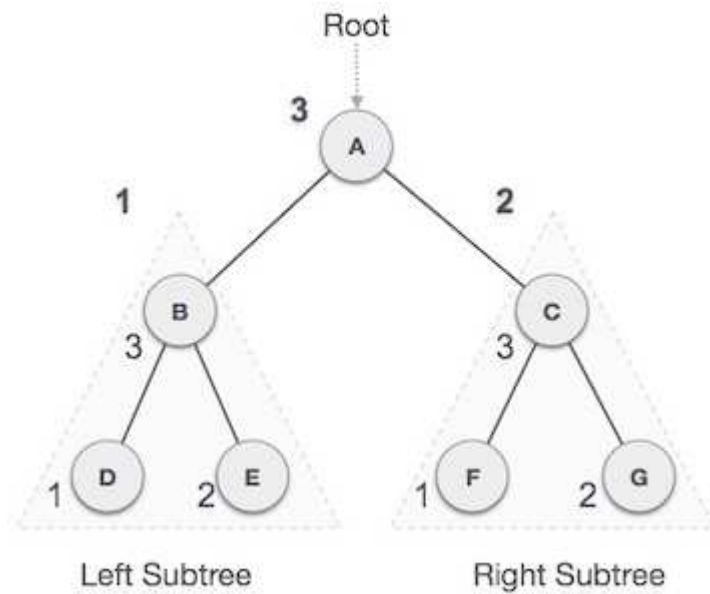


Fig. 5.11 : Trees with Traversal

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 –

$$D \rightarrow E \rightarrow B \rightarrow F \rightarrow G \rightarrow C \rightarrow A$$

Algorithm

Until all nodes are traversed –

Step 1 – Recursively traverse left subtree.

Step 2 – Recursively traverse right subtree.

Step 3 – Visit root node.

Unit- 6

Q1: Explain Graph Data Structure in detail

Answer :

Mathematical graphs can be represented in data structure. We can represent a graph using an array of vertices and a two-dimensional array of edges. Before we proceed further, let's familiarize ourselves with some important terms –

- **Vertex** – Each node of the graph is represented as a vertex. In the following example, the labeled circle represents vertices. Thus, A to G are vertices. We can represent them using an array as shown in the following image. Here A can be identified by index 0. B can be identified using index 1 and so on.
- **Edge** – Edge represents a path between two vertices or a line between two vertices. In the following example, the lines from A to B, B to C, and so on represents edges. We can use a two-dimensional array to represent an array as shown in the following image. Here AB can be represented as 1 at row 0, column 1, BC as 1 at row 1, column 2 and so on, keeping other combinations as 0.
- **Adjacency** – Two node or vertices are adjacent if they are connected to each other through an edge. In the following example, B is adjacent to A, C is adjacent to B, and so on.

- **Path** – Path represents a sequence of edges between the two vertices. In the following example, ABCD represents a path from A to D.

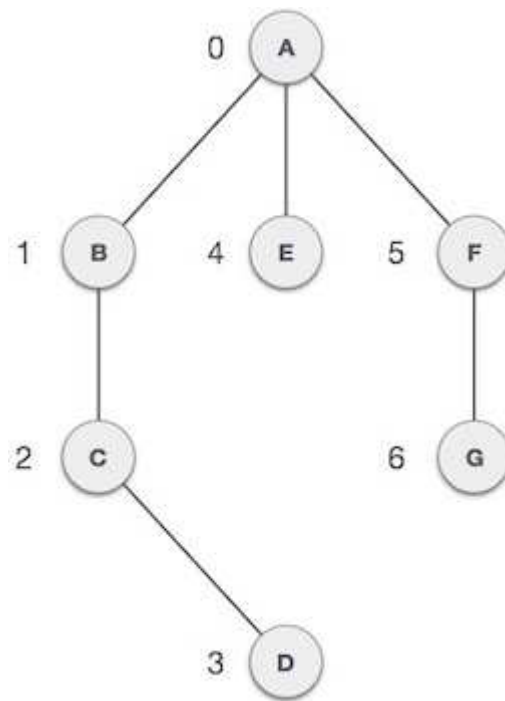


Fig 6.1 Graph

Q3: Explain in detail binary search tree representation**Binary Search Tree Representation****Answer :**

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

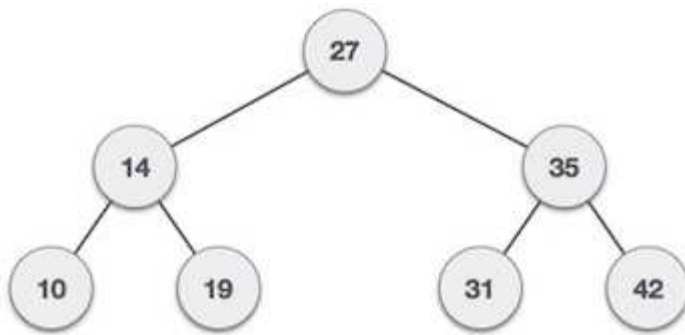


Fig. 6.2 : Graph

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

Tree Node

The code to write a tree node would be similar to what is given below. It has a data part and references to its left and right child nodes.

```

struct node {
    int data;
    struct node *leftChild;
    struct node *rightChild;
};

```

In a tree, all nodes share common construct.

BST Basic Operations

The basic operations that can be performed on a binary search tree data structure, are the following –

- **Insert** – Inserts an element in a tree/create a tree.
- **Search** – Searches an element in a tree.
- **Preorder Traversal** – Traverses a tree in a pre-order manner.
- **Inorder Traversal** – Traverses a tree in an in-order manner.
- **Postorder Traversal** – Traverses a tree in a post-order manner.

Q4: Explain in detail tree traversal methods

Answer :

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.

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.

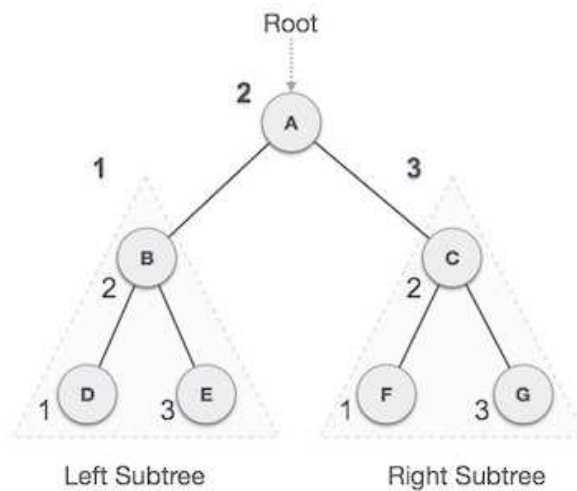


Fig.6.3 Traversal

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 inorder traversal of this tree will be –

$$D \rightarrow B \rightarrow E \rightarrow A \rightarrow F \rightarrow C \rightarrow G$$

Algorithm

Until all nodes are traversed –

Step 1 – Recursively traverse left subtree.

Step 2 – Visit root node.

Step 3 – Recursively traverse right subtree.

Pre-order Traversal

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

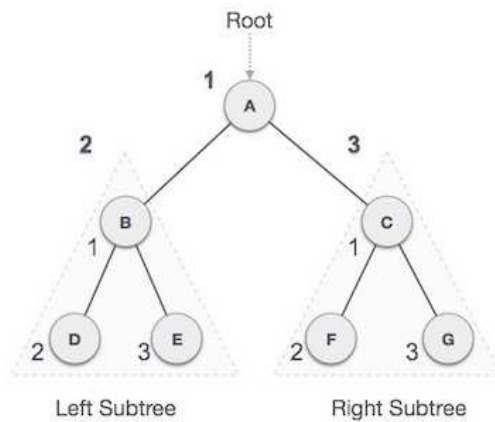


Fig. 6.4 Traversal

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 –

$$A \rightarrow B \rightarrow D \rightarrow E \rightarrow C \rightarrow F \rightarrow G$$

Algorithm

Until all nodes are traversed –

Step 1 – Visit root node.

Step 2 – Recursively traverse left subtree.

Step 3 – Recursively traverse right subtree.

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.

Unit - 6

Q1: Explain in detail tree representaion

Answer :

A graph is a pictorial representation of a set of objects where some pairs of objects are connected by links. The interconnected objects are represented by points termed as **vertices**, and the links that connect the vertices are called **edges**.

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

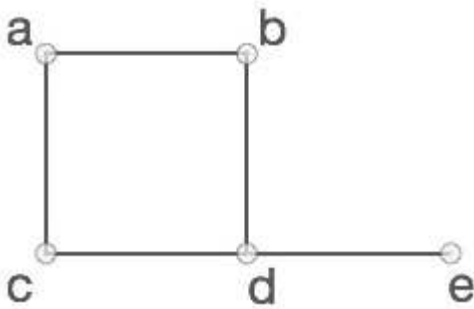


Fig.6.1 Graph

In the above graph,

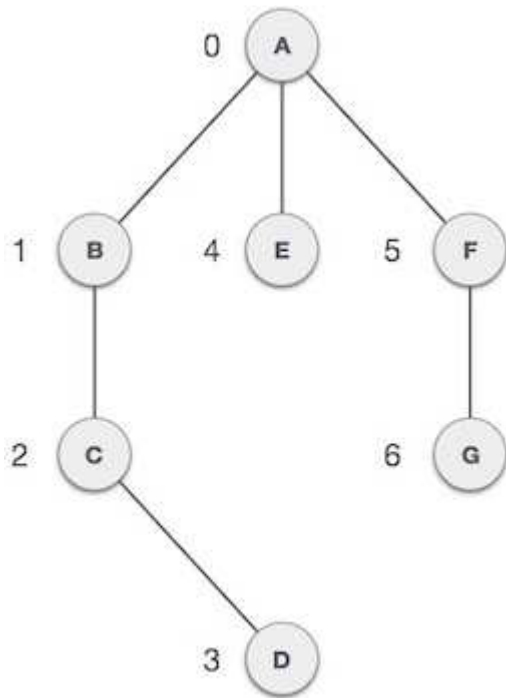
$$V = \{a, b, c, d, e\}$$

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

Graph Data Structure

Mathematical graphs can be represented in data structure. We can represent a graph using an array of vertices and a two-dimensional array of edges. Before we proceed further, let's familiarize ourselves with some important terms –

- **Vertex** – Each node of the graph is represented as a vertex. In the following example, the labeled circle represents vertices. Thus, A to G are vertices. We can represent them using an array as shown in the following image. Here A can be identified by index 0. B can be identified using index 1 and so on.
- **Edge** – Edge represents a path between two vertices or a line between two vertices. In the following example, the lines from A to B, B to C, and so on represents edges. We can use a two-dimensional array to represent an array as shown in the following image. Here AB can be represented as 1 at row 0, column 1, BC as 1 at row 1, column 2 and so on, keeping other combinations as 0.
- **Adjacency** – Two node or vertices are adjacent if they are connected to each other through an edge. In the following example, B is adjacent to A, C is adjacent to B, and so on.
- **Path** – Path represents a sequence of edges between the two vertices. In the following example, ABCD represents a path from A to D.



Fig, 6.2 Graph

Basic Operations

Following are basic primary operations of a Graph –

- **Add Vertex** – Adds a vertex to the graph.
- **Add Edge** – Adds an edge between the two vertices of the graph.
- **Display Vertex** – Displays a vertex of the graph.

Q2: Explain important terms used in graphs

Answer :

Following are the important terms with respect to tree.

- **Path** – Path refers to the sequence of nodes along the edges of a tree.
- **Root** – The node at the top of the tree is called root. There is only one root per tree and one path from the root node to any node.
- **Parent** – Any node except the root node has one edge upward to a node called parent.
- **Child** – The node below a given node connected by its edge downward is called its child node.
- **Leaf** – The node which does not have any child node is called the leaf node.
- **Subtree** – Subtree represents the descendants of a node.
- **Visiting** – Visiting refers to checking the value of a node when control is on the node.
- **Traversing** – Traversing means passing through nodes in a specific order.
- **Levels** – Level of a node represents the generation of a node. If the root node is at level 0, then its next child node is at level 1, its grandchild is at level 2, and so on.
- **keys** – Key represents a value of a node based on which a search operation is to be carried out for a node.

Q3: Graph and its representations**Answer :**

A graph is a data structure that consists of the following two components:

1. A finite set of vertices also called as nodes.
2. A finite set of ordered pair of the form (u, v) called as edge. The pair is ordered because (u, v) is not the same as (v, u) in case of a directed graph(di-graph). The pair of the form (u, v) indicates that there is an edge from vertex u to vertex v . The edges may contain weight/value/cost.

Graphs are used to represent many real-life applications: Graphs are used to represent networks. The networks may include paths in a city or telephone network or circuit network. Graphs are also used in social networks like linkedIn, Facebook. For example, in Facebook, each person is represented with a vertex(or node). Each node is a structure and contains information like person id, name, gender, and locale. See this for more applications of graph.

Following is an example of an undirected graph with 5 vertices.

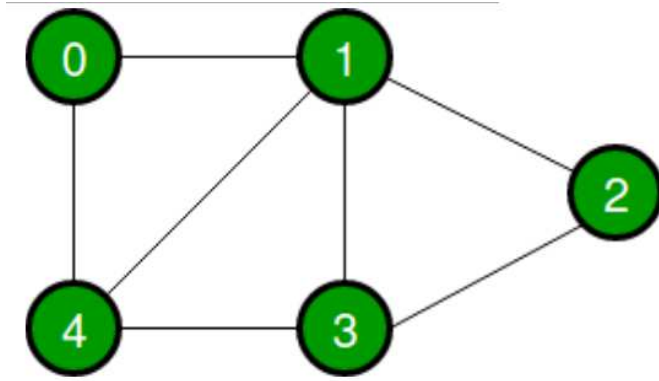


Fig.6.3 graph

The following two are the most commonly used representations of a graph.

1. Adjacency Matrix
2. Adjacency List

There are other representations also like, Incidence Matrix and Incidence List. The choice of graph representation is situation-specific. It totally depends on the type of operations to be performed and ease of use.

Adjacency Matrix:

Adjacency Matrix is a 2D array of size $V \times V$ where V is the number of vertices in a graph. Let the 2D array be $adj[][]$, a slot $adj[i][j] = 1$ indicates that there is an edge from vertex i to vertex j . Adjacency matrix for undirected graph is always symmetric. Adjacency Matrix is also used to represent weighted graphs. If $adj[i][j] = w$, then there is an edge from vertex i to vertex j with weight w .

Q4: Explain in detail adjacency list representation of graph**Adjacency List:****Answer :**

An array of lists is used. The size of the array is equal to the number of vertices. Let the array be an array[]. An entry array[i] represents the list of vertices adjacent to the *i*th vertex. This representation can also be used to represent a weighted graph. The weights of edges can be represented as lists of pairs. Following is the adjacency list representation of the above graph.

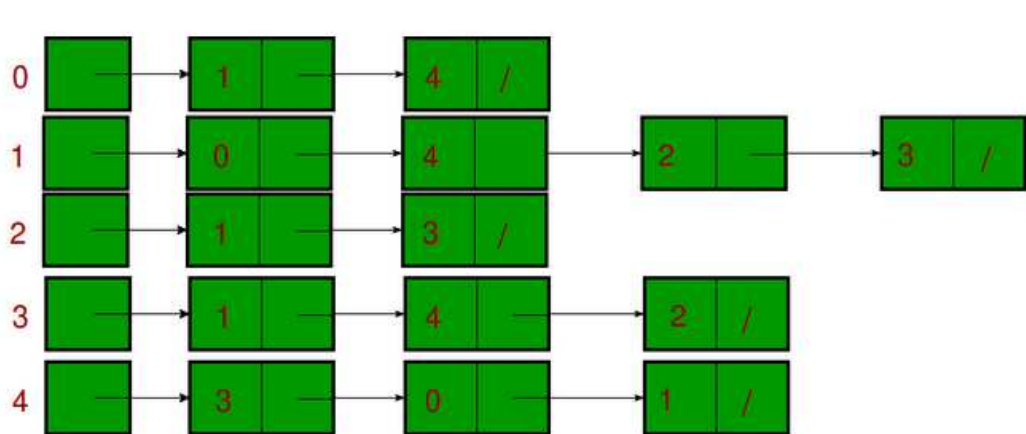


Fig 6.4 linked List representation

Q5: Explain in detail Graph traversal methods

Answer :

Graph traversals

Graph traversal means visiting every vertex and edge exactly once in a well-defined order. While using certain graph algorithms, you must ensure that each vertex of the graph is visited exactly once. The order in which the vertices are visited are important and may depend upon the algorithm or question that you are solving.

During a traversal, it is important that you track which vertices have been visited. The most common way of tracking vertices is to mark them.

Breadth First Search (BFS)

There are many ways to traverse graphs. BFS is the most commonly used approach.

BFS is a traversing algorithm where you should start traversing from a selected node (source or starting node) and traverse the graph layerwise thus exploring the neighbour nodes (nodes which are directly connected to source node). You must then move towards the next-level neighbour nodes.

As the name BFS suggests, you are required to traverse the graph breadthwise as follows:

1. First move horizontally and visit all the nodes of the current layer
2. Move to the next layer

Consider the following diagram.

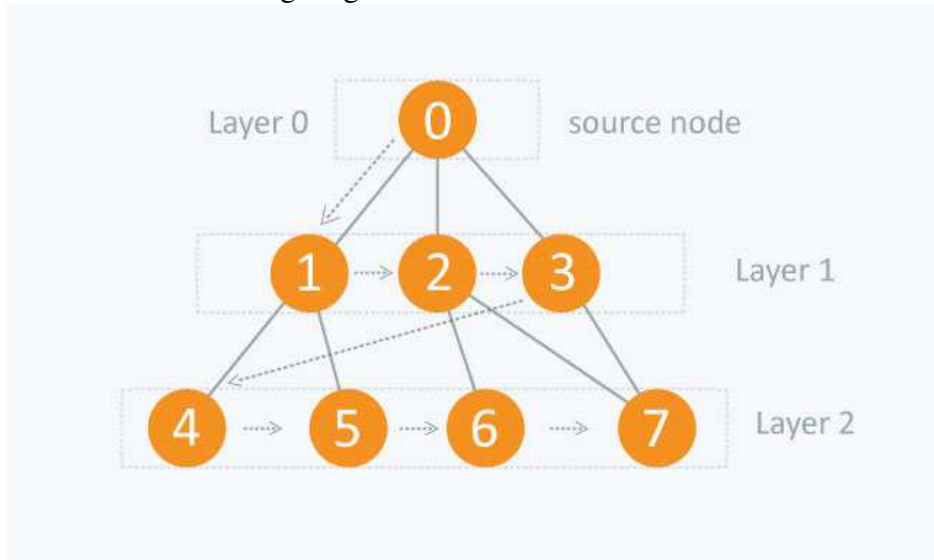


Fig. 6.5 Graph Traversal

The distance between the nodes in layer 1 is comparatively lesser than the distance between the nodes in layer 2. Therefore, in BFS, you must traverse all the nodes in layer 1 before you move to the nodes in layer 2.

Traversing child nodes

A graph can contain cycles, which may bring you to the same node again while traversing the graph. To avoid processing of same node again, use a boolean array which marks the node after it is processed. While visiting the nodes in the layer of a graph, store them in a manner such that you can traverse the corresponding child nodes in a similar order.

In the earlier diagram, start traversing from 0 and visit its child nodes 1, 2, and 3. Store them in the order in which they are visited. This will allow you to visit the child nodes of 1 first (i.e. 4 and 5), then of 2 (i.e. 6 and 7), and then of 3 (i.e. 7) etc.

To make this process easy, use a queue to store the node and mark it as 'visited' until all its neighbours (vertices that are directly connected to it) are marked. The queue follows the First In First Out (FIFO) queuing method, and therefore, the neighbors of the node will be visited in the order in which they were inserted in the node i.e. the node that was inserted first will be visited first, and so on.

Q6: What is a Minimum Spanning Tree?

Answer :

The cost of the spanning tree is the sum of the weights of all the edges in the tree. There can be many spanning trees. Minimum spanning tree is the spanning tree where the cost is minimum among all the spanning trees. There also can be many minimum spanning trees.

Minimum spanning tree has direct application in the design of networks. It is used in algorithms approximating the travelling salesman problem, multi-terminal minimum cut problem and minimum-cost weighted perfect matching. Other practical applications are:

1. Cluster Analysis
2. Handwriting recognition
3. Image segmentation

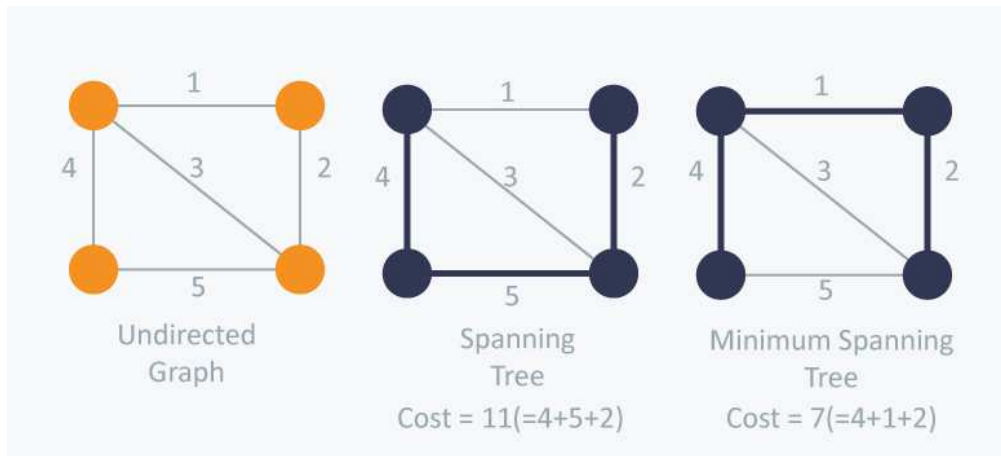


Fig. 6.6 Spanning Tree

Q7: Explain in detail kruskal algorithm

I. KRUSKAL'S ALGORITHM

Answer :

Kruskal's Algorithm builds the spanning tree by adding edges one by one into a growing spanning tree. Kruskal's algorithm follows greedy approach as in each iteration it finds an edge which has least weight and add it to the growing spanning tree.

Algorithm Steps:

- Sort the graph edges with respect to their weights.

- Start adding edges to the MST from the edge with the smallest weight until the edge of the largest weight.
- Only add edges which doesn't form a cycle , edges which connect only disconnected components.

So now the question is how to check if

vertices are connected or not ?

This could be done using DFS which starts from the first vertex, then check if the second vertex is visited or not. But DFS will make time complexity large as it has an order of

where n is the number of vertices,

m is the number of edges. So the best solution is "**Disjoint Sets**":

Disjoint sets are sets whose intersection is the empty set so it means that they don't have any element in common.

Consider following example:

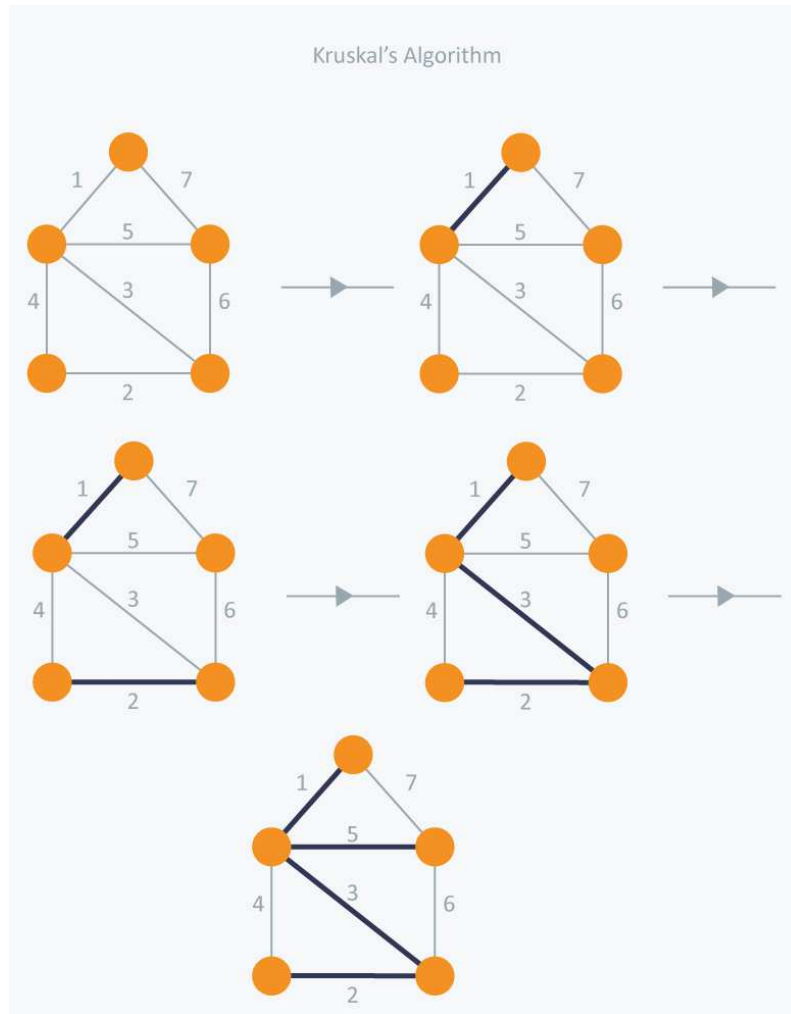


Fig. 6.7 Kruskshel Algorithm

In Kruskal's algorithm, at each iteration we will select the edge with the lowest weight. So, we will start with the lowest weighted edge first i.e., the edges with weight 1. After that we will select the second lowest weighted edge i.e., edge with weight 2. Notice

these two edges are totally disjoint. Now, the next edge will be the third lowest weighted edge i.e., edge with weight 3, which connects the two disjoint pieces of the graph. Now, we are not allowed to pick the edge with weight 4, that will create a cycle and we can't have any cycles. So we will select the fifth lowest weighted edge i.e., edge with weight 5. Now the other two edges will create cycles so we will ignore them. In the end, we end up with a minimum spanning tree with total cost 11 ($= 1 + 2 + 3 + 5$).

Q8: Explain Prim's Algorithm

Answer :

Prim's Algorithm also use Greedy approach to find the minimum spanning tree. In Prim's Algorithm we grow the spanning tree from a starting position. Unlike an **edge** in Kruskal's, we add **vertex** to the growing spanning tree in Prim's.

Algorithm Steps:

- Maintain two disjoint sets of vertices. One containing vertices that are in the growing spanning tree and other that are not in the growing spanning tree.
- Select the cheapest vertex that is connected to the growing spanning tree and is not in the growing spanning tree and add it into the growing spanning tree. This can be done using Priority Queues. Insert the vertices, that are connected to growing spanning tree, into the Priority Queue.

- Check for cycles. To do that, mark the nodes which have been already selected and insert only those nodes in the Priority Queue that are not marked.

Consider the example below:

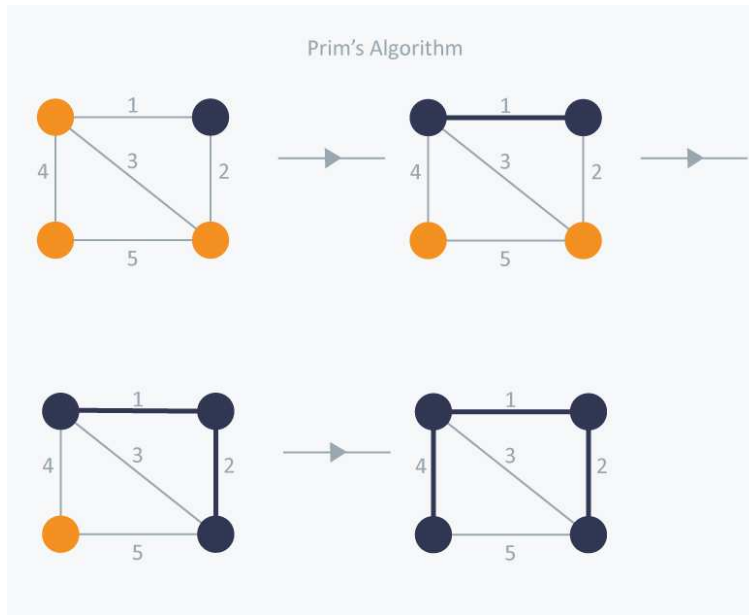


Fig. 6.8 Prim's Algorithm

In Prim's Algorithm, we will start with an arbitrary node (it doesn't matter which one) and mark it. In each iteration we will mark a new vertex that is adjacent to the one that we have already marked. As a greedy algorithm, Prim's algorithm will select the cheapest edge and mark the vertex. So we will simply choose the edge with weight 1. In the next iteration we have three options, edges with weight 2, 3 and 4. So, we will

select the edge with weight 2 and mark the vertex. Now again we have three options, edges with weight 3, 4 and 5. But we can't choose edge with weight 3 as it is creating a cycle. So we will select the edge with weight 4 and we end up with the minimum spanning tree of total cost 7 ($= 1 + 2 + 4$).

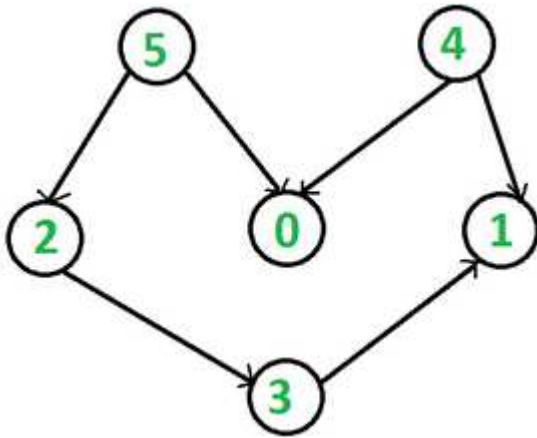
Q9: Topological Sorting

Answer :

Topological sorting for Directed Acyclic Graph (DAG) is a linear ordering of vertices such that for every directed edge uv , vertex u comes before v in the ordering.

Topological Sorting for a graph is not possible if the graph is not a DAG.

For example, a topological sorting of the following graph is “5 4 2 3 1 0”. There can be more than one topological sorting for a graph. For example, another topological sorting of the following graph is “4 5 2 3 1 0”. The first vertex in topological sorting is always a vertex with in-degree as 0 (a vertex with no incoming edges).



Topological Sorting vs Depth First Traversal (DFS):

In DFS, we print a vertex and then recursively call DFS for its adjacent vertices. In topological sorting, we need to print a vertex before its adjacent vertices. For example, in the given graph, the vertex '5' should be printed before vertex '0', but unlike DFS, the vertex '4' should also be printed before vertex '0'. So Topological sorting is different from DFS. For example, a DFS of the shown graph is "5 2 3 1 0 4", but it is not a topological sorting

Algorithm to find Topological Sorting:

We recommend to first see implementation of DFS [here](#). We can modify DFS to find Topological Sorting of a graph. In DFS, we start from a vertex, we first print it and then recursively call DFS for its adjacent vertices. In topological sorting, we use a temporary stack. We don't print the vertex immediately, we first recursively call topological sorting for all its adjacent vertices, then push it to a stack. Finally, print contents of stack. Note that a vertex is

pushed to stack only when all of its adjacent vertices (and their adjacent vertices and so on) are already in stack.

Q10: Explain Key Steps in Critical Path Method

Answer :

Let's have a look at how critical path method is used in practice. The process of using critical path method in project planning phase has six steps.

Step 1: Activity specification

You can use the Work Breakdown Structure (WBS) to identify the activities involved in the project. This is the main input for the critical path method.

In activity specification, only the higher-level activities are selected for critical path method.

When detailed activities are used, the critical path method may become too complex to manage and maintain.

Step 2: Activity sequence establishment

In this step, the correct activity sequence is established. For that, you need to ask three questions for each task of your list.

- Which tasks should take place before this task happens.
- Which tasks should be completed at the same time as this task.
- Which tasks should happen immediately after this task.

Step 3: Network diagram

Once the activity sequence is correctly identified, the network diagram can be drawn (refer to the sample diagram above).

Although the early diagrams were drawn on paper, there are a number of computer softwares, such as Primavera, for this purpose nowadays.

Step 4: Estimates for each activity

This could be a direct input from the WBS based estimation sheet. Most of the companies use 3-point estimation method or COCOMO based (function points based) estimation methods for tasks estimation.

You can use such estimation information for this step of the process.

Step 5: Identification of the critical path

For this, you need to determine four parameters of each activity of the network.

- Earliest start time (ES) - The earliest time an activity can start once the previous dependent activities are over.

- Earliest finish time (EF) - ES + activity duration.
- Latest finish time (LF) - The latest time an activity can finish without delaying the project.
- Latest start time (LS) - LF - activity duration.

The float time for an activity is the time between the earliest (ES) and the latest (LS) start time or between the earliest (EF) and latest (LF) finish times.

During the float time, an activity can be delayed without delaying the project finish date.

The critical path is the longest path of the network diagram. The activities in the critical path have an effect on the deadline of the project. If an activity of this path is delayed, the project will be delayed.

In case if the project management needs to accelerate the project, the times for critical path activities should be reduced.

Step 6: Critical path diagram to show project progresses

Critical path diagram is a live artefact. Therefore, this diagram should be updated with actual values once the task is completed.

This gives more realistic figure for the deadline and the project management can know whether they are on track regarding the deliverables Advantages of Critical Path Method

Following are advantages of critical path methods:

- Offers a visual representation of the project activities.
- Presents the time to complete the tasks and the overall project.
 - Tracking of critical activities.

References

- WWW.geeksforgeeks.org
- www.hackererth.com
- Scahum's Series Algorithms and Data Structures
- Data structures in C by Y.P. Kanetkar