JAYPEE INSTITUTE OF INFORMATION TECHNOLOGY

Department of CSE&IT

Lab Manual

Course Code: 15B11CI578

Course Name: Data Structures and Algorithms Lab

B.Tech

3rt year (5th Sem)

Prepared by:
Dr Manju (62), Dr Raju Pal (128)

LABORATORY INTRODUCTION:

Module No.	Subtitle of the Module	Topics in the module	No. of Labs (Module Wise)
1	Introduction & Algorithm Complexity	**Lab Assignment 1:** Conversion from one number system to another; Manipulation with arrays and strings, structures; **Lab Assignment 2 and 3:** Manipulation with a single Linked lists of integers; **Lab Assignment 4:** Stacks and Queues Finding Complexity: Big O, Big Omega Cost Analysis	3
2	Sorting, Searching & Trees	**Lab Assignments 2 and 3:** Doubly Linked List, Circular Linked List **Lab Assignments 4:** Multi-Linked Lists **Lab Assignments 5 and 6**: Sorting, Searching, Application based. **Lab Assignments 7, 8, 9**: Binary Tree, Binary Search Trees, AVL Tree , Case-study: Priority Queue with Binary Trees,	6
3	Introduction to Heaps Directed and undirected graphs, weighted graphs, etc.	**Lab Assignments 10:** Heaps **Lab Assignment 11**: Directed and undirected graphs, weighted graphs, etc.	2
4	Hashing, Backtracking, Branch and Bound, Greedy Algorithms, Dynamic Programming.	**Lab Assignments 12:** Hashing, Backtracking, Branch and Bound, Greedy Algorithms, Dynamic Programming.	1
Total number of Labs		12	

Assignment - 1

(Practice Assignment)

Number System:

When we type some letters or words, the computer translates them in numbers as computers can understand only numbers. A computer can understand the positional number system where there are only a few symbols called digits and these symbols represent different values depending on the position they occupy in the number.

The value of each digit in a number can be determined using −

- The digit
- The position of the digit in the number
- The base of the number system (where the base is defined as the total number of digits available in the number system)

Question 1: Conversion between number systems

(a) Write a program for converting a given integer in decimal number system to its binary equivalent. You should use a function that access values of an array using pointers for reversal and pointer as an argument.

(b) Add another function to the above question to convert the integer to its octal number system.

(c) Add an additional function to convert the integer to its hexadecimal number system.

(d) Modify all the above functions to convert a number having integer as well as decimal part to its respective number system.

(e) Make a menu driven program to include all above conversions. Further extend them to have their complementary functions too, i.e., decimal to binary & binary to decimal; decimal to octal & octal to decimal; decimal to hexadecimal & hexadecimal to decimal;

(f) Extend the program to have an option for conversion from decimal to any other number system.

Approach: There are many methods or techniques which can be used to convert numbers from one base to another.

- Decimal to Other Base System
- Other Base System to Decimal
- Other Base System to Non-Decimal
- Binary to Octal
- Octal to Binary
- Binary to Hexadecimal
- Hexadecimal to Binary

Decimal to Other Base System

Steps

- **Step 1** − Divide the decimal number to be converted by the value of the new base.
- **Step 2** − Get the remainder from Step 1 as the rightmost digit (least significant digit) of new base number.
- **Step 3** − Divide the quotient of the previous divide by the new base.
- **Step 4** − Record the remainder from Step 3 as the next digit (to the left) of the new base number.

Repeat Steps 3 and 4, getting remainders from right to left, until the quotient becomes zero in Step 3.

The last remainder thus obtained will be the Most Significant Digit (MSD) of the new base number.

Other Base System to Decimal System

Steps

- **Step 1** − Determine the column (positional) value of each digit (this depends on the position of the digit and the base of the number system).
- **Step 2** − Multiply the obtained column values (in Step 1) by the digits in the corresponding columns.
- **Step 3** − Sum the products calculated in Step 2. The total is the equivalent value in decimal.

Other Base System to Non-Decimal System

Steps

- **Step 1** − Convert the original number to a decimal number (base 10).
- **Step 2** − Convert the decimal number so obtained to the new base number.

Shortcut method - Binary to Octal

Steps

- **Step 1** − Divide the binary digits into groups of three (starting from the right).
- **Step 2** − Convert each group of three binary digits to one octal digit.

Shortcut method - Octal to Binary

Steps

- **Step 1** − Convert each octal digit to a 3 digit binary number (the octal digits may be treated as decimal for this conversion).
- **Step 2** − Combine all the resulting binary groups (of 3 digits each) into a single binary number.

Shortcut method - Binary to Hexadecimal

Steps

- **Step 1** − Divide the binary digits into groups of four (starting from the right).
- **Step 2** − Convert each group of four binary digits to one hexadecimal symbol.

Shortcut method - Hexadecimal to Binary

Steps

- **Step 1** − Convert each hexadecimal digit to a 4 digit binary number (the hexadecimal digits may be treated as decimal for this conversion).
- **Step 2** − Combine all the resulting binary groups (of 4 digits each) into a single binary number.

Question 2: Manipulation with arrays and strings, structures.

(i) Student Dataset

(a) Take input from user for "n" number of students. Store their names, enrolment numbers and marks for 5 different subjects. ("n" is a variable whose value is to be input at runtime).

(b) Sort alphabetically on keyvalue name and display accordingly.

(c) Sort enrolment number wise and display accordingly.

(d) Sort marks wise for individual subjects and display accordingly.

(e) Make a menu-driven program for inclusion of all the above tasks from (i)(a) to (i)(d).

Approach: An **array** is a collection of like variables that share a single name. The individual elements of an array are referenced by appending a subscript, in square brackets, behind the name. The subscript itself can be any legitimate C expression that yields an integer value, even a general expression. Therefore, arrays in C may be regarded as collections of like variables. Although arrays represent one of the simplest data structures, it has wide-spread usage in embedded systems.

Strings are similar to arrays with just a few differences. Usually, the array size is fixed, while strings can have a variable number of elements. Arrays can contain any data type (char short int even other arrays) while strings are usually ASCII characters terminated with a NULL (0) character. In general we allow random access to individual array elements. On the other hand, we usually process strings sequentially character by character from start to end. Since these differences are a matter of semantics rather than specific limitations imposed by the syntax of the C programming language, the descriptions in this chapter apply equally to data arrays and character strings. String literals were discussed earlier in Chapter 3; in this chapter we will define data structures to hold our strings. In addition, C has a rich set of predefined functions to manipulate strings.

Hint:

- Make a student structure with following fields: name as string, enrol number as int, and marks as array of length 5 containing marks of 5 subjects.
- To store the data for n students make array of structures
- Apply operations given in the question on above created data structure

(ii) Take string as an input and convert it to its “Piglatin” form. Make a function for depicting the rule and using it on the input given by the user. Eg:

Input	Output
Lame	Amelay
Happy	Appyhay
Child	Ildchay
Pig	Igpay
Hair	Airhay
code	odecay

Hint:

“Piglatin” form.

The first vowel occurring in the input word is placed at the start of the new word along with the remaining alphabet of it. The alphabet is present before the first vowel is shifted, at the end of the new word it is followed by “ay”.

Question 3: An unsorted array A of size N of positive integer numbers is given. One number 'x' is missing from set {1, 2, …N} and one number 'y' occurs twice in the input array. Find these two numbers.

Example 1:

Input: N = 2, A[] = {2, 2}

Output: 2 1

Explanation: Repeating number is 2 and smallest positive missing number is 1.

Example 2:

Input: N = 3, A[] = {1, 3, 3}

Output: 3 2

Explanation: Repeating number is 3 and

smallest positive missing number is 2.

Approach:

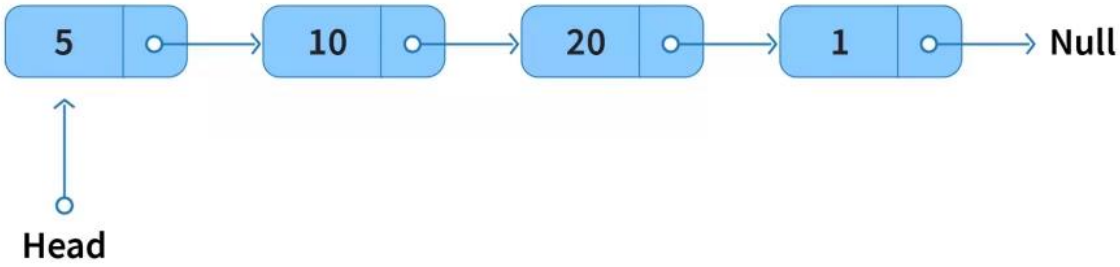
Sort the input array.

Traverse the array and check for missing and repeating.

Assignment-2
Single Linked List

Singly Linked List

A Singly Linked List is a specialized case of a generic linked list. In a singly linked list, each node links to only the next node in the sequence, i.e., if we start traversing from the first node of the list, we can only move in one direction (pun intended).



Algorithm:

- Create a temporary node(temp) and assign the head node's address.
- Print the data which present in the temp node.
- After printing the data, move the temp pointer to the next node.
- Do the above process until we reach the end.

Question 1: Create a linked list of numbers. Take input from user. Search and find the position of a particular key value. If the key-value exists, return its position, else add it to the end of the linked list.

(i) Write functions for each of these. Compare their performance in terms of number of comparisons and number of swaps.
(ii) Delete a particular element, chosen by the user, from the above linked list.
(iii)Add an element, input by the user, at a position (i) mentioned by the user, (ii) at the starting of the linked list, (ii) middle of the linked list, (iii) end of the linked list.
(iv) Update the value of a particular element, chosen by the user, to another value.
(v) If there are multiple instances of the same number, delete it from all the positions, except its

(i) last position, (ii) first position, (iii) all the positions, where the element was found.

(vi)Display the contents of SLL in reverse order.

(vii) Display second largest number in SLL (Do not sort the link list).

Algorithm

1. If the head node has the given key,

   make the head node points to the second node and free its memory.

2. Otherwise,

   From the current node, check whether the next node has the given key

   if yes, make the current->next = current->next->next and free the memory.

   else, update the current node to the next and do the above process (from step 2) till the last node.

Algorithm – insert at start

1. Declare a head pointer and make it as NULL.
2. Create a new node with the given data.
3. Make the new node points to the head node.
4. Finally, make the new node as the head node.

Algorithm – insert at the end

1. Declare head pointer and make it as NULL.
2. Create a new node with the given data. And make the new node => next as NULL. (Because the new node is going to be the last node.)
3. If the head node is NULL (Empty Linked List),

   make the new node as the head.

4. If the head node is not null, (Linked list already has some elements),

   find the last node.

   make the last node => next as the new node.

Algorithm

1. Iterate the linked list using a loop.
2. If any node has the given key value, return 1 and update its value
3. If the program execution comes out of the loop (the given key is not present in the linked list), return -1.

2. Write a program to design a library management application using linked list which will maintain a record of all the books available in the library including (Book name, author, ISBN, publication year, no of copies available etc.). There is another list containing the name of the issued books and the name of student, to whom the book is issued. Provide menu which will contain options to add, delete, display and update the entry in the records.

Hint:

Make a structure containing all the information given in the question

Construct a linked list of maintain record of books

Make another linked list for issued books

Perform operations on these linked list as per user's choice given in the menu at runtime

3. Write a program to create another single link list (SLL1) having elements as 24, 6, 7, 8, 1, 2, 8, 10,4, 27, 16, 26. Later, write a program to sort LL2 (retain all the duplicate elements). Call this list as SLL2. Finally merge SLL2 with SLL1 such that elements of merged linked list will be in sorted order (retaining all the duplicate elements).

Hint:

Concatenate the two lists by traversing the first list until we reach it's a tail node and then point the next of the tail node to the head node of the second list. Store this concatenated list in the first list.

Sort the above-merged linked list. Here, we will use a bubble sort. So, if node->next->data is less then node->data, then swap the data of the two adjacent nodes.

4. Write a program to find sum of all elements of linked list created in Q1 and Q2 above separately. Write a function to perform following operations: If (sum of all elements of first LL - sum of all elements of second LL) > last element of first LL then start deleting last elements of first LL until (remaining elements of first LL - sum of all elements of second LL) ≈ 0. Otherwise delete elements of first LL from beginning until (remaining elements of first LL - sum of all elements of second LL) ≈ 0

Hint:

Develop linked list as per the Q1 and Q2

Perform the delete operation on first LL as per the condition given in the Question.

Assignment- 3

Circular linked list

Circular Linked List

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

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

for singly linked list, next pointer of last item points to the first item

In the doubly linked list, prev pointer of the first item points to the last item as well.

Q1. Create a circular integer linked list, then write a function to search an element from the list.

Hint:

Algorithm

Step 1: SET PTR = HEAD

Step 2: Set I = 0

STEP 3: IF PTR = NULL

WRITE "EMPTY LIST"
GOTO STEP 8
END OF IF

STEP 4: IF HEAD → DATA = ITEM

WRITE i+1 RETURN [END OF IF]

STEP 5: REPEAT STEP 5 TO 7 UNTIL PTR->next != head

STEP 6: if ptr → data = item

write i+1
RETURN
End of IF

STEP 7: I = I + 1

STEP 8: PTR = PTR → NEXT

[END OF LOOP]

STEP 9: EXIT

Q2. Create a circular singly linked list of N nodes, then modify every node of list such that each node stores the sum of all nodes except that node.

Example:

Input: 4 ↔ 5 ↔ 6 ↔7 ↔ 8

Output: 26 ↔ 25 ↔ 24 ↔ 23 ↔ 22

Input: 1 ↔ 2

Output:2 ↔ 1

Hint:

Calculate the sum of all elements in the linked list and store it in a variable

Create another linked list and insert the elements in it.

The value for each node would be determined by (overall sum – current node) in the given linked list.

Q3. Given a circular singly linked list containing N nodes, the task is to remove all the nodes from the list which contains elements whose digit sum is even.

Example:

Input: CLL = 9 -> 11 -> 34 -> 6 -> 13 -> 21

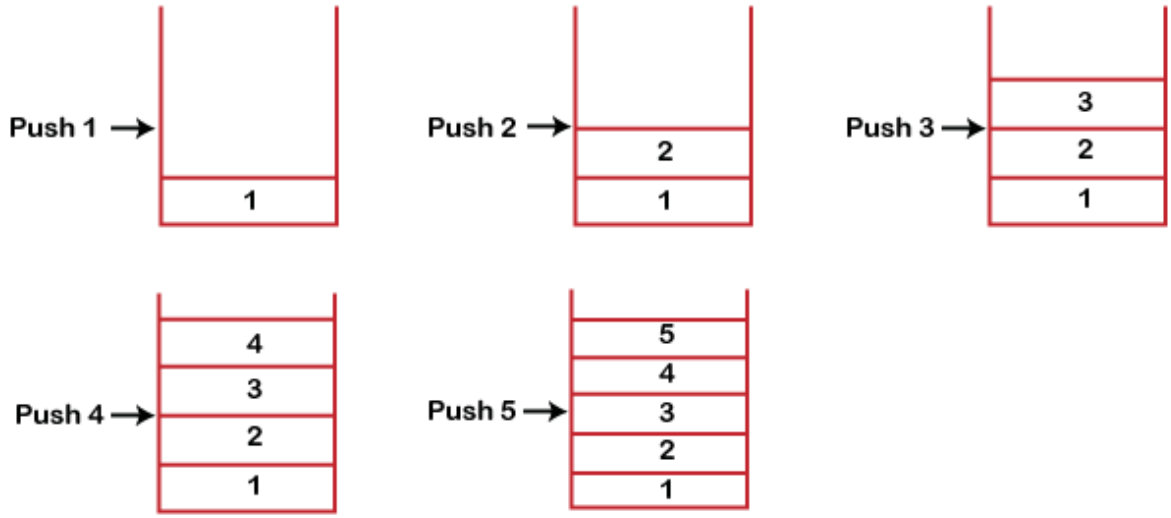
Output: 9 -> 34 -> 21

Approach: The idea is to traverse the nodes of the circular singly linked list one by one and for each node, find the digit sum for the value present in the node by iterating through each digit. If the digit sum is even, then remove the nodes. Else, continue.

Assignment 4

Stack

A Stack is a linear data structure that follows the **LIFO (Last-In-First-Out)** principle. Stack has one end, whereas the Queue has two ends (**front and rear**). It contains only one pointer **top pointer** pointing to the topmost element of the stack. Whenever an element is added in the stack, it is added on the top of the stack, and the element can be deleted only from the stack. In other words, a *stack can be defined as a container in which insertion and deletion can be done from the one end known as the top of the stack.*



Q1. Write a program to reverse a string using stack.

Algorithm:

- Create an empty stack.
- One by one push all characters of string to stack.
- One by one pop all characters from stack and put them back to string.

Q2. Convert an infix notation to prefix and postfix notation.

Algorithm: Infix -> postfix

- Scan the infix expression from left to right.
- If the scanned character is an operand, output it.
- Else,
    - If the precedence and associativity of the scanned operator are greater than the precedence and associativity of the operator in the stack(or the stack is empty or the stack contains a '('), then push it.
    - '^' operator is right associative and other operators like '+','-','*' and '/' are left-associative. Check especially for a condition when both, operator at the top of the stack and the scanned operator are '^'. In this condition, the precedence of the scanned operator is higher due to its right associativity.

So it will be pushed into the operator stack. In all the other cases when the top of the operator stack is the same as the scanned operator, then pop the operator from the stack because of left associativity due to which the scanned operator has less precedence.

- Else, Pop all the operators from the stack which are greater than or equal to in precedence than that of the scanned operator. After doing that Push the scanned operator to the stack. (If you encounter parenthesis while popping then stop there and push the scanned operator in the stack.)

- If the scanned character is an '(', push it to the stack.
- If the scanned character is an ')', pop the stack and output it until a '(' is encountered, and discard both the parenthesis.
- Repeat steps 2-6 until the infix expression is scanned.
- Print the output
- Pop and output from the stack until it is not empty.

Algorithm: Infix -> prefix

- Step 1: Reverse the infix expression i.e A+B*C will become C*B+A. Note while reversing each '(' will become ')' and each ')' becomes '('.
- Step 2: Obtain the "nearly" postfix expression of the modified expression i.e CB*A+.
- Step 3: Reverse the postfix expression. Hence in our example prefix is +A*BC

Queue

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

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

Q3. For a given queue, Q, write a recursive function to shuffle the values stored in first half of queue with the second half of it in an alternate manner. The first and the last values of the queue is not affected by the shuffle. For example, (LHS is front of queue and RHS as rear of queue)

Example 1:

Input Q: 1 2 3 4 5 6

Output Q: 1 5 3 4 2 6

Example 2:

Input Q: 7 4 6 2 1

Output Q: 7 2 6 4 1

Algorithm

Make a temporary queue and push the first half of the original queue into the temp queue.

Till the temp queue is empty

Pop the front of the temp queue and push it to the original queue

Pop the front of the original queue and push it to the original queue

The original queue is converted to the interleaved queue.

Q4. Given a queue, shuffle values of the queue, such that the smallest element is shifted to the end of the queue without changing the basic ordering of the elements in the queue. Further, write a modified stack for the same function such that the smallest element is shifted to the top of the stack without changing the ordering of the other elements.

Algorithm:

On every pass on the queue, we seek for the next minimum index. To do this we dequeue and enqueue elements until we find the next minimum. In this operation the queue is not changed at all. After we have found the minimum index, we dequeue and enqueue elements from the queue except for the minimum index, after we finish the traversal in the queue we insert the minimum to the rear of the queue.

We keep on this until all minimums are pushed all way long to the front and the queue becomes sorted.

On every next seeking for the minimum, we exclude seeking on the minimums that have already sorted.

We repeat this method n times.
At first we seek for the maximum, because on every pass we need find the next minimum, so we need to compare it with the largest element in the queue.

Assignment 5

Sorting

A Sorting Algorithm is used to rearrange a given array or list of elements according to a comparison operator on the elements. The comparison operator is used to decide the new order of elements in the respective data structure.

Searching Algorithms are designed to check for an element or retrieve an element from any data structure where it is stored. Based on the type of search operation, these algorithms are generally classified into two categories:

1. **Sequential Search:** In this, the list or array is traversed sequentially and every element is checked. For example: Linear Search.

2. **Interval Search:** These algorithms are specifically designed for searching in sorted data-structures. These type of searching algorithms are much more efficient than Linear Search as they repeatedly target the center of the search structure and divide the search space in half. For Example: Binary Search.

Q1. Write a program having functions to perform various searching algorithms including: Linear Search, Binary Search and Compare their complexity.

Linear Search

Linear Search is defined as a sequential search algorithm that starts at one end and goes through each element of a list until the desired element is found, otherwise the search continues till the end of the data set. It is the easiest searching algorithm

Algorithm:

- Start from the leftmost element of arr[] and one by one compare x with each element of arr[]
- If x matches with an element, return the index.
- If x doesn't match with any of the elements, return -1.

Binary Search

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

Algorithm:

- Begin with the mid element of the whole array as a search key.

- If the value of the search key is equal to the item then return an index of the search key.
- Or 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 from the second point until the value is found or the interval is empty.

Q2: Write recursive program for binary search also.

Hint: Use the algorithm given above to write a recursive program.

Q3. Write a program having functions to perform various sorting algorithms including:

Selection Sort, Bubble Sort, Insertion Sort.

Selection Sort

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

The algorithm maintains two subarrays in a given array.

- The subarray which already sorted.
- The remaining subarray was unsorted.

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

Bubble Sort

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

Insertion Sort

Insertion sort is a simple sorting algorithm that works similar to the way you sort playing cards in your hands. The array is virtually split into a sorted and an unsorted part. Values from the unsorted part are picked and placed at the correct position in the sorted part.

Q4: Write recursive program above sorting methods (Question 3)

Bubble Sort

1. Base Case: If array size is 1, return.

2. Do One Pass of normal Bubble Sort. This pass fixes last element of current subarray.

3. Recur for all elements except last of current subarray.

Insertion Sort

1. Base Case: If array size is 1 or smaller, return.
2. Recursively sort first n-1 elements.
3. Insert last element at its correct position in sorted array.

Assignment – 6

Sorting

A Sorting Algorithm is used to rearrange a given array or list of elements according to a comparison operator on the elements. The comparison operator is used to decide the new order of elements in the respective data structure.

Q1. Write a program having functions to perform various sorting algorithms including:

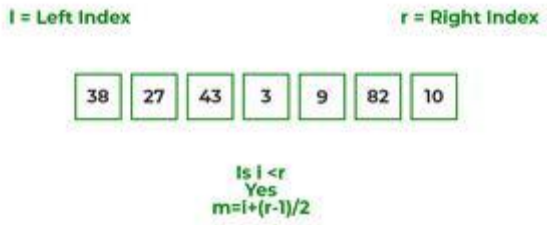
Merge Sort, Quick Sort.

Merge Sort:

A step by step illustration of merge sort

array arr[] = {38, 27, 43, 3, 9, 82, 10}

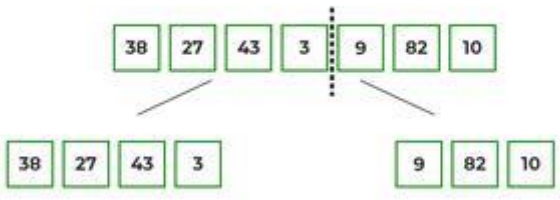
- At first, check if the left index of array is less than the right index, if yes then calculate its mid point



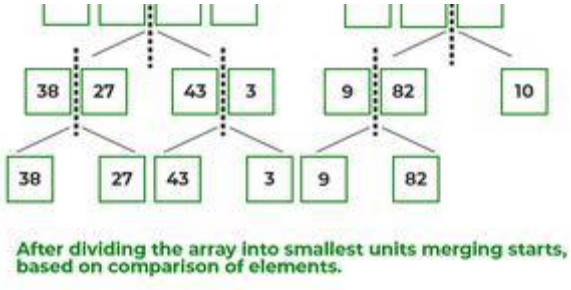
Now, as we already know that merge sort first divides the whole array iteratively into equal halves, unless the atomic values are achieved.

- Here, we see that an array of 7 items is divided into two arrays of size 4 and 3 respectively.

- Now, again find that is left index is less than the right index for both arrays, if found yes, then again calculate mid points for both the arrays.



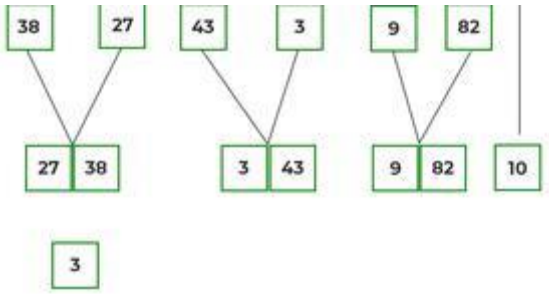
Now, further divide these two arrays into further halves, until the atomic units of the array is reached and further division is not possible.



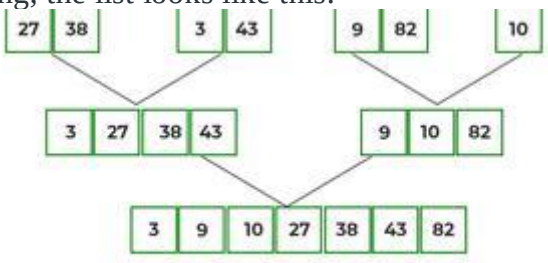
After dividing the array into smallest units merging starts, based on comparison of elements.

After dividing the array into smallest units, start merging the elements again based on comparison of size of elements

- Firstly, compare the element for each list and then combine them into another list in a sorted manner.



- After the final merging, the list looks like this:



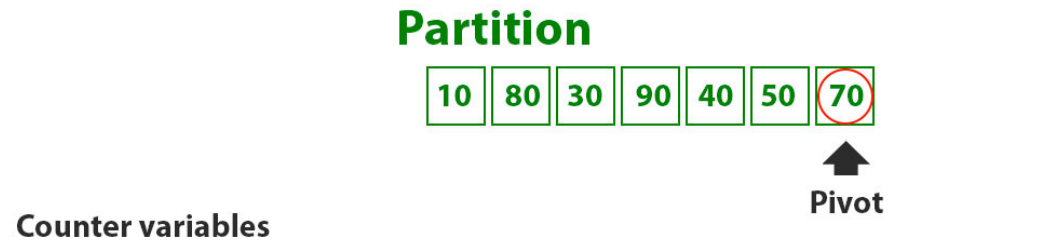
Quick Sort:

There are many different versions of quick Sort that pick pivot in different ways.

- Always pick the first element as a pivot.
- Always pick the last element as a pivot (implemented below)
- Pick a random element as a pivot.
- Pick median as the pivot.

The key process in quickSort is a partition().
Illustration of partition() :
Consider: arr[] = {10, 80, 30, 90, 40, 50, 70}

- Indexes: 0 1 2 3 4 5 6
- low = 0, high = 6, pivot = arr[h] = 70
- Initialize index of smaller element, i = -1
-

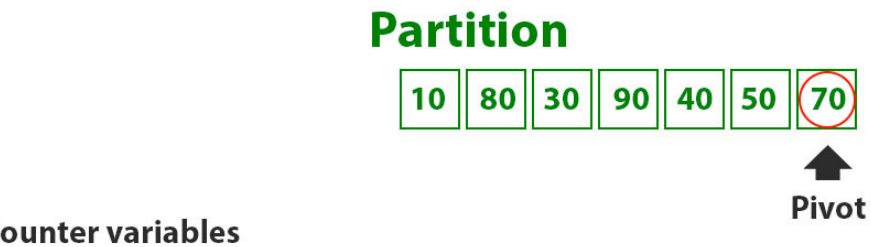


Counter variables
I: Index of smaller element
J: Loop variable

We start the loop with initial values

Test Condition	Actions	Value of variables
arr [J] <= pivot		I = -1 J = 0

- Traverse elements from j = low to high-1
    - j = 0: Since arr[j] <= pivot, do i++ and swap(arr[i], arr[j])
    - i = 0
- arr[] = {10, 80, 30, 90, 40, 50, 70} // No change as i and j are same
    - j = 1: Since arr[j] > pivot, do nothing
-



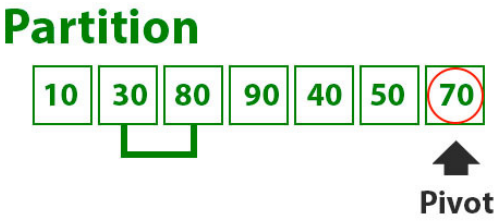
Counter variables
I: Index of smaller element
J: Loop variable

Pass 2

Test Condition	Actions	Value of variables
arr [J] <= pivot 80 < 70 false	No Action	I = 0 J = 1

-

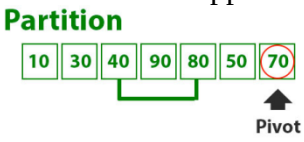
- j = 2 : Since arr[j] <= pivot, do i++ and swap(arr[i], arr[j])
- i = 1
- arr[] = {10, 30, 80, 90, 40, 50, 70} // We swap 80 and 30
-



Counter variables
I: Index of smaller element
J: Loop variable

Test Condition	Actions	Value of variables
arr [J] <= pivot 30 < 70 true	i++ Swap(arr[i],arr[j])	I = 1 J = 2

- j = 3 : Since arr[j] > pivot, do nothing // No change in i and arr[]
- j = 4 : Since arr[j] <= pivot, do i++ and swap(arr[i], arr[j])
- i = 2
- arr[] = {10, 30, 40, 90, 80, 50, 70} // 80 and 40 Swapped

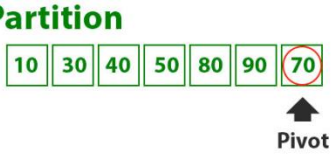


Counter variables
I: Index of smaller element
J: Loop variable

Pass 5

Test Condition	Actions	Value of variables
arr [J] <= pivot 40 < 70 true	i++ Swap(arr[i],arr[j])	I = 2 J = 4

- j = 5 : Since arr[j] <= pivot, do i++ and swap arr[i] with arr[j]
- i = 3
- arr[] = {10, 30, 40, 50, 80, 90, 70} // 90 and 50 Swapped

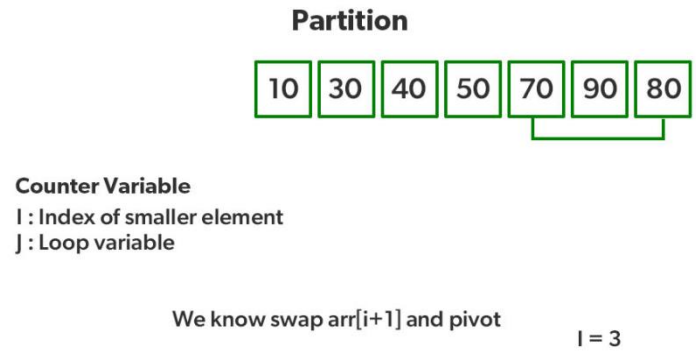


Counter variables
I: Index of smaller element
J: Loop variable

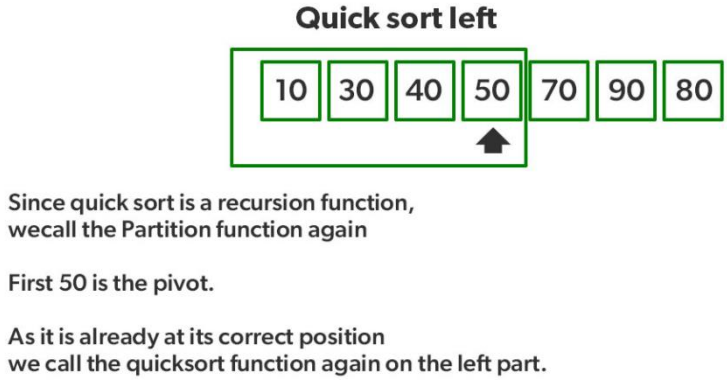
Before Pass 7, J becomes 6
so we come out of the loop

Test Condition	Actions	Value of variables
arr [J] <= pivot		I = 3 J = 6

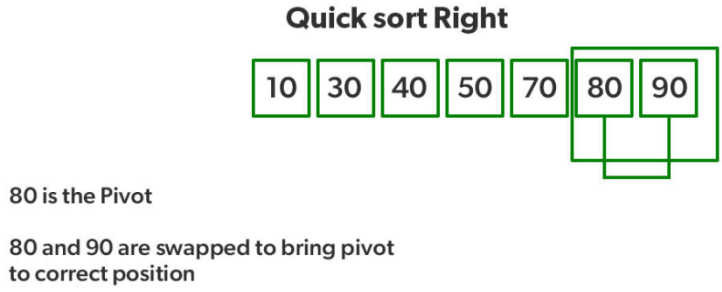
- We come out of loop because j is now equal to high-1.
- Finally we place pivot at correct position by swapping arr[i+1] and arr[high] (or pivot)
- arr[] = {10, 30, 40, 50, 70, 90, 80} // 80 and 70 Swapped



- Now 70 is at its correct place. All elements smaller than 70 are before it and all elements greater than 70 are after it.
- Since quick sort is a recursive function, we call the partition function again at left and right partitions



- Again call function at right part and swap 80 and 90



Q2: Write recursive program for above sorting methods

Pseudo code for quicksort()

```
/* low  –> Starting index,  high  –> Ending index */
quickSort(arr[], low, high) {
   if (low < high) {
      /* pi is partitioning index, arr[pi] is now at right place */
      pi = partition(arr, low, high);
```

```
        quickSort(arr, low, pi – 1);  // Before pi

        quickSort(arr, pi + 1, high); // After pi

    }

}
```

Pseudo code for partition ()

/* This function takes last element as pivot, places the pivot element at its correct position in sorted array, and places all smaller (smaller than pivot) to left of pivot and all greater elements to right of pivot */

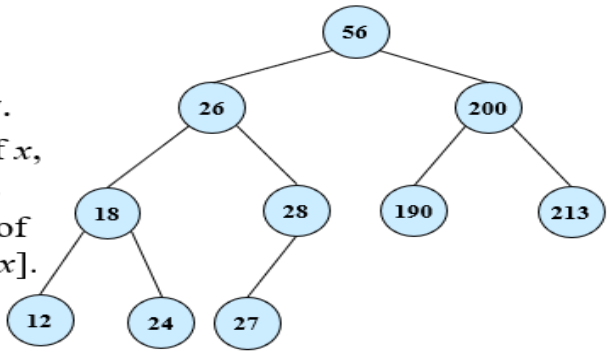
```
partition (arr[], low, high)
{
    // pivot (Element to be placed at right position)
    pivot = arr[high];

    i = (low – 1)  // Index of smaller element and indicates the
    // right position of pivot found so far

    for (j = low; j <= high- 1; j++){

        // If current element is smaller than the pivot
        if (arr[j] < pivot){
            i++;    // increment index of smaller element
            swap arr[i] and arr[j]
        }
    }
    swap arr[i + 1] and arr[high])
    return (i + 1)
}
```

Assignment-7

Binary Search Tree

Binary Search Tree: Binary search is a variant of Binary tree which follows following property.

- Stored keys must satisfy the *binary search tree* property.
  - ∀ y in left subtree of x, then $key[y] \leq key[x]$.
  - ∀ y in right subtree of x, then $key[y] \geq key[x]$.



BST – Representation

- Represented by a linked data structure of nodes.
- *root*(*T*) points to the root of tree *T*.
- Each node contains fields:
  - *key*
  - *left* – pointer to left child: root of left subtree.
  - *right* – pointer to right child : root of right subtree

Basic operations on a BST

1. Create: creates an empty tree.
2. Insert: insert a node in the tree.
3. Search: Searches for a node in the tree.
4. Delete: deletes a node from the tree.
5. Inorder: in-order traversal of the tree.
6. Preorder: pre-order traversal of the tree.
7. Postorder: post-order traversal of the tree.

Q1. Write Programs to create a binary search tree for the following input.

13, 3, 4, 12, 14, 10, 5, 1, 8, 2, 7, 9, 11, 6, 18

Steps: Construct a Binary Search Tree

1. Set the current node to be the root node of the tree.

2. If the target value equals the key value for the current node, then the target value is found. ...
3. If the target value is less than the key value for a node, make the current node the left child.

Q2. Perform following traversals on the above BST

1. Traverse and display a BST by doing InOrder Traversal.
2. Traverse and display a BST by doing PreOrder Traversal.
3. Traverse and display a BST by doing PostOrder Traversal.
4. Traverse and display a BST by doing LevelOrder Traversal.

Traversal is a process to visit all the nodes of a tree and may print their values too. Because, all nodes are connected via edges (links) we always start from the root (head) node. That is, we cannot randomly access a node in a tree. There are four ways which we use to traverse a tree −

In-order Traversal

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

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

Until all nodes are traversed −
Step 1 − Recursively traverse left subtree.
Step 2 − Recursively traverse right subtree.
Step 3 − Visit root node.
Level-order Traversal

Until all nodes are traversed −
Step 1 −Visit root node.
Step 2 − Recursively traverse subtree at level all the levels

Q3. Search for a particular (kth) value in a BST and return its depth and position, if it exists; or return a NULL if the value is not present in the BST.

Search Operation

The algorithm depends on the property of BST that if each left subtree has values below root and each right subtree has values above the root. If the value is below the root, we can say for sure that the value is not in the right subtree; we need to only search in the left subtree and if

the value is above the root, we can say for sure that the value is not in the left subtree; we need to only search in the right subtree.

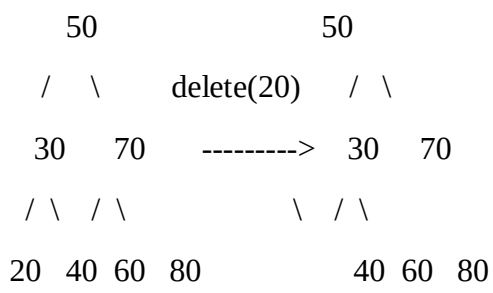
Algorithm:

```
If root == NULL
   return NULL;
If number == root->data
   return root->data;
If number < root->data
   return search(root->left)
If number > root->data
   return search(root->right)
```

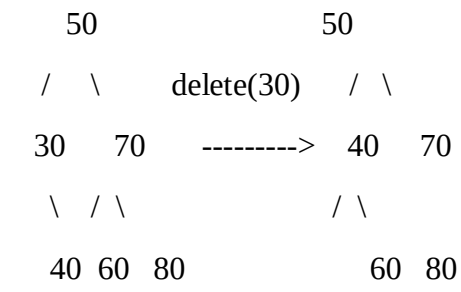
Q4. Delete a particular element from a BST and then display the resultant Binary Search Tree.

When we delete a node, three possibilities arise.

1) Node to be deleted is the leaf: Simply remove from the tree.



2) Node to be deleted has only one child: Copy the child to the node and delete the child



3) Node to be deleted has two children: Find inorder successor of the node. Copy contents of the inorder successor to the node and delete the inorder successor. Note that inorder predecessor can also be used.

Assignment - 8

AVL Tree

AVL tree is a height-balanced binary search tree. That means, an AVL tree is also a binary search tree but it is a balanced tree. A binary tree is said to be balanced if, the difference between the heights of left and right subtrees of every node in the tree is either -1, 0 or +1.

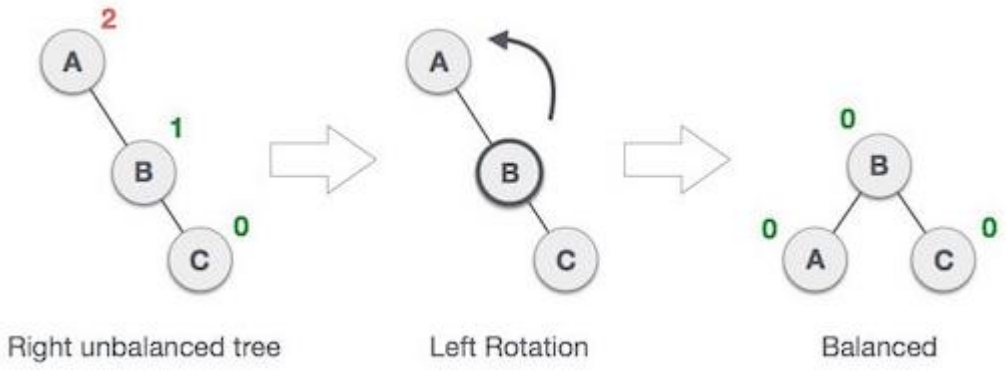
AVL Rotations

We perform rotation in AVL tree only in case if Balance Factor is other than -1, 0, and 1. There are basically four types of rotations which are as follows:

1. L L rotation: Inserted node is in the left subtree of left subtree of A
2. R R rotation : Inserted node is in the right subtree of right subtree of A
3. L R rotation : Inserted node is in the right subtree of left subtree of A
4. R L rotation : Inserted node is in the left subtree of right subtree of A

1. RR Rotation

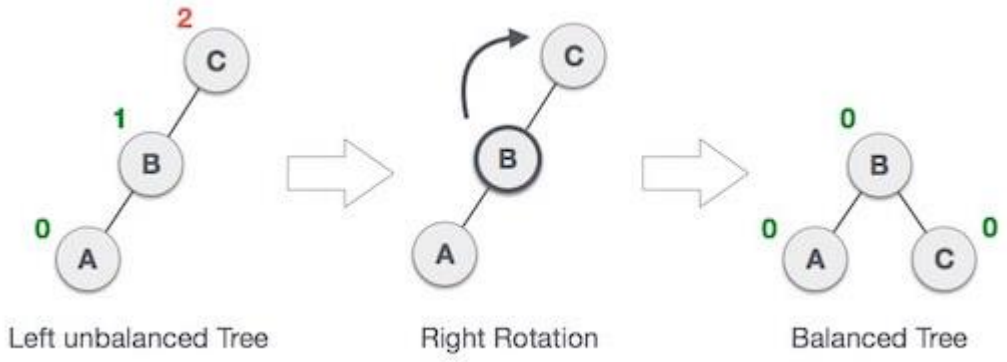
When BST becomes unbalanced, due to a node is inserted into the right subtree of the right subtree of A, then we perform RR rotation, RR rotation is an anticlockwise rotation, which is applied on the edge below a node having balance factor -2



In above example, node A has balance factor -2 because a node C is inserted in the right subtree of A right subtree. We perform the RR rotation on the edge below A.

2. LL Rotation

When BST becomes unbalanced, due to a node is inserted into the left subtree of the left subtree of C, then we perform LL rotation, LL rotation is clockwise rotation, which is applied on the edge below a node having balance factor 2.



In above example, node C has balance factor 2 because a node A is inserted in the left subtree of C left subtree. We perform the LL rotation on the edge below A.

3. LR Rotation

Double rotations are bit tougher than single rotation which has already explained above. LR rotation = RR rotation + LL rotation, i.e., first RR rotation is performed on subtree and then LL rotation is performed on full tree, by full tree we mean the first node from the path of inserted node whose balance factor is other than -1, 0, or 1.

4. RL Rotation

As already discussed, that double rotations are bit tougher than single rotation which has already explained above. R L rotation = LL rotation + RR rotation, i.e., first LL rotation is performed on subtree and then RR rotation is performed on full tree, by full tree we mean the first node from the path of inserted node whose balance factor is other than -1, 0, or 1.

Q1. Write Programs to create an AVL tree for the following input.

13, 3, 4, 12, 14, 10, 5, 1, 8, 2, 7, 9, 11, 6, 18

Steps to follow for insertion:

Let the newly inserted node be **w**

- Perform standard **BST** insert for **w**.
- Starting from **w**, travel up and find the first **unbalanced node**. Let **z** be the first unbalanced node, **y** be the **child** of **z** that comes on the path from **w** to **z** and **x** be the **grandchild** of **z** that comes on the path from **w** to **z**.
- Re-balance the tree by performing appropriate rotations on the subtree rooted with **z.** There can be 4 possible cases that need to be handled as **x, y** and **z** can be arranged in 4 ways.

Q2. Perform following traversals on the above AVL Tree

1. **Display the height of the AVL tree formed**

2. **Traverse and display the AVL by doing InOrder Traversal.**
3. **Traverse and display the AVL by doing LevelOrder Traversal.**

In-order Traversal

Algorithm

Until all nodes are traversed −

Step 1 − Recursively traverse left subtree.

Step 2 − Visit root node.

Step 3 − Recursively traverse right subtree.

Level-order Traversal

Algorithm

Until all nodes are traversed −
Step 1 −Visit root node.
Step 2 − Recursively traverse subtree at level all the levels

Q3. Search for a particular (kth) value in given AVL tree and return its depth and position, if it exists; or return a NULL if the value is not present in the AVL tree.

Search like in Binary Search tree

Q4. Delete a particular element from the given AVL tree and then display the AVL tree and write a function to merge two given AVL Trees into a single AVL Tree.

The deletion operation in the AVL tree is the same as the deletion operation in BST. In the AVL tree, the node is always deleted as a leaf node and after the deletion of the node, the balance factor of each node is modified accordingly. Rotation operations are used to modify the balance factor of each node. The algorithm steps of deletion operation in an AVL tree are:

1. Locate the node to be deleted
2. If the node does not have any child, then remove the node
3. If the node has one child node, replace the content of the deletion node with the child node and remove the node
4. If the node has two children nodes, find the inorder successor node ‘k' which has no child node and replace the contents of the deletion node with the ‘k’ followed by removing the node.
5. Update the balance factor of the AVL tree

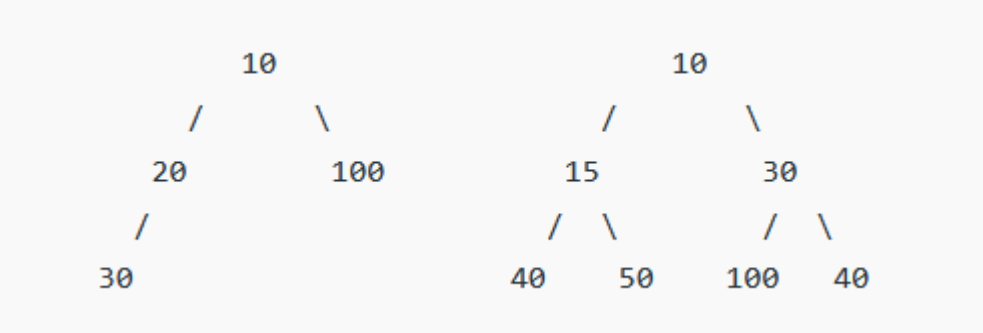
Assignment 9

Heap Tree

HEAP- Heap data structure is a complete binary tree that satisfies the heap property, where any given node is

- always greater than its child node/s and the key of the root node is the largest among all other nodes. This property is also called max heap property.
- always smaller than the child node/s and the key of the root node is the smallest among all other nodes. This property is also called min heap property.

Examples of Min Heap:



Array Representation- A binary heap is represented as an array. The representation follows some property.

- The root of the tree will be at Arr[0].
- For any node at Arr[i], its left and right children will be at Arr[2*i + 1] and Arr[2*i+2] respectively.
- For any node at Arr[i], its parent node will be at Arr[(i-1)/2].

Q1: Write a program to construct priority queue using heap. Print the final contents of the priority queue.

Step 1 − Create a new node at the end of heap.

Step 2 − Assign new value to the node.

Step 3 − Compare the value of this child node with its parent.

Step 4 − If value of parent is less than child, then swap them.

Step 5 − Repeat step 3 & 4 until Heap property holds.

Q2. Write a program to perform Heapsort and compute the number of comparisons that is required to perform this.

- Step 1 - Construct a Binary Tree with given list of Elements.
- Step 2 - Transform the Binary Tree into Min Heap.
- Step 3 - Delete the root element from Min Heap using Heapify method.
- Step 4 - Put the deleted element into the Sorted list.
- Step 5 - Repeat the same until Min Heap becomes empty.
- Step 6 - Display the sorted list.

Q3. Given a set of numbers 23,17,14,6,13,20,10,11,5,7,12 find if it represents max-heap or not.

We can efficiently solve this problem by using recursion. The idea is to start from the root node (at index 0) and check if the left and right child (if any) of the root node is greater than the root node or not. If true, recursively call the procedure for both its children; otherwise, return false from the function. Following is the complete algorithm:

- If the current node is a leaf node, return true as every leaf node is a heap.
- If the current node is an internal node,
    - Recursively check if the left child is min-heap or not.
    - Recursively check if the right child is min-heap or not (if it exists).
    - Return true if both left and right child are min-heap; otherwise, return false.

Assignment – 10

Graph Traversal

Graph- A graph data structure is a collection of nodes that have data and are connected to other nodes. More precisely, a graph is a data structure (V, E) that consists of

- A collection of vertices V
- A collection of edges E, represented as ordered pairs of vertices (u,v)

Adjacency: A vertex is said to be adjacent to another vertex if there is an edge connecting them. Vertices 2 and 3 are not adjacent because there is no edge between them.

Path: A sequence of edges that allows you to go from vertex A to vertex B is called a path. 0-1, 1-2 and 0-2 are paths from vertex 0 to vertex 2.

Directed Graph: A graph in which an edge (u,v) doesn't necessarily mean that there is an edge (v, u) as well. The edges in such a graph are represented by arrows to show the direction of the edge.

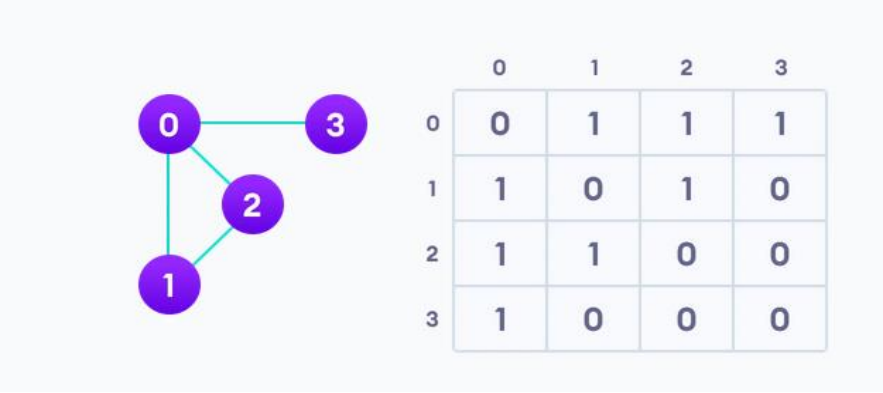
Graph Representation

Graphs are commonly represented in two ways:

1. Adjacency Matrix

An adjacency matrix is a 2D array of V x V vertices. Each row and column represent a vertex.

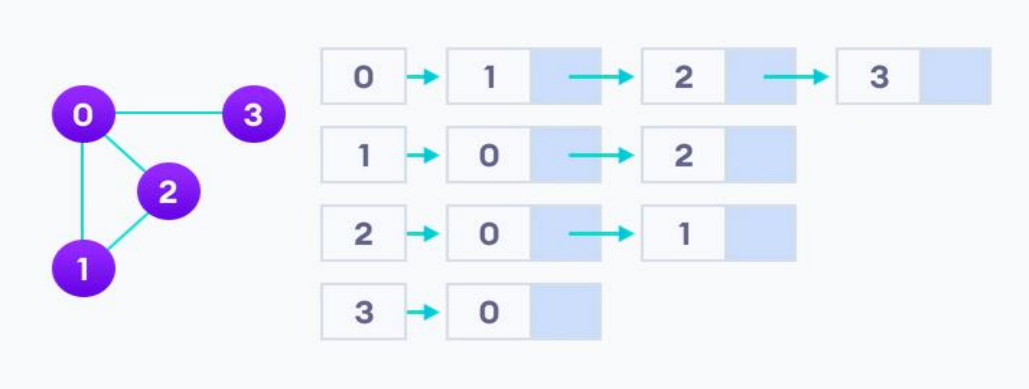
If the value of any element `a[i][j]` is 1, it represents that there is an edge connecting vertex i and vertex j.



2. Adjacency List

An adjacency list represents a graph as an array of linked lists.

The index of the array represents a vertex and each element in its linked list represents the other vertices that form an edge with the vertex.



Q1. Write a function to input a directed graph. You can use any method for input such as multi-list, oradjacency matrix or vertex and edge list, etc. Perform the following by writing individual functions forthe same:

(i) Find the in-degree of a particular node.

(ii) Find the out-degree of a particular node.

(iii) Find the node with the maximum in-degree.

(iv) Find the node with the minimum in-degree.

(v) Find the degree of a given node.

Approach: Traverse adjacency list for every vertex, if size of the adjacency list of vertex **i** is **x** then the out degree for **i = x** and increment the in degree of every vertex that has an incoming edge from **i**. Repeat the steps for every vertex and print the in and out degrees for all the vertices in the end.

Algorithm:

1. Create the graphs adjacency matrix from src to des
2. For the given vertex then check if
   a path from this vertices to other exists then
   increment the degree.
3. Return degree

Q2. Write a program to store the graph data structure on an adjacency list and perform the following operations:

a. To traverse the graph in depth-first search (DFS) manner

b. To traverse the graph in breadth-first search (BFS) manner

Algorithm-DFS

1. Start by putting one of the vertexes of the graph on the stack's top.
2. Put the top item of the stack and add it to the visited vertex list.
3. Create a list of all the adjacent nodes of the vertex and then add those nodes to the unvisited at the top of the stack.
4. Keep repeating steps 2 and 3, and the stack becomes empty.

Algorithm-BFS

1. Start putting anyone vertices from the graph at the back of the queue.
2. First, move the front queue item and add it to the list of the visited node.
3. Next, create nodes of the adjacent vertex of that list and add them which have not been visited yet.
4. Keep repeating steps two and three until the queue is found to be empty.

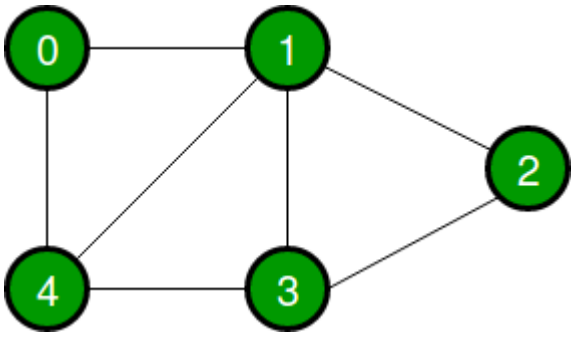
Assignment – 11

Graph: Minimum spanning Tree

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

Vertices: Vertices are the fundamental units of the graph. Sometimes, vertices are also known as vertex or nodes. Every node/vertex can be labeled or unlabelled.

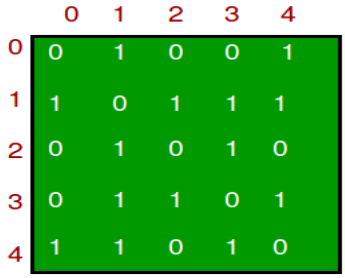
Edges: Edges are drawn or used to connect two nodes of the graph. It can be ordered pair of nodes in a directed graph. Edges can connect any two nodes in any possible way. There are no rules. Sometimes, edges are also known as arcs. Every edge can be labeled/unlabelled.



REPRESENTATION OF GRAPH

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

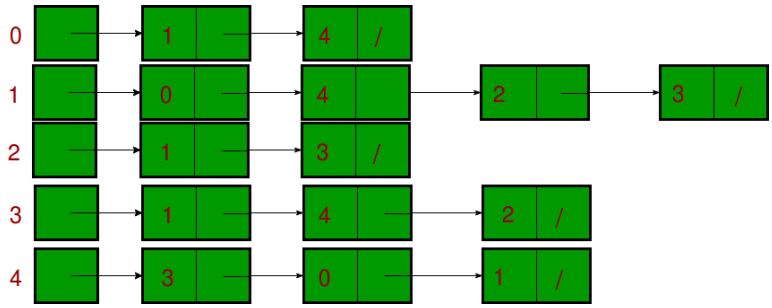
1. **Adjacency Matrix**
   Adjacency Matrix is a 2D array of size V x V where V is the number of vertices in a graph. Let the 2D array be adj[][], a slot adj[i][j] = 1 indicates that there is an edge from vertex i to vertex j. Adjacency matrix for undirected graph is always symmetric. Adjacency Matrix 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.



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

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.



MINIMUM SPANNING TREE

Given an undirected and connected graph G(V,E) , a spanning tree of the graph is a tree that spans (that is, it includes every vertex of G) and is a acyclic subgraph of G. 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.

There are two famous algorithms for finding the Minimum Spanning Tree:

1. **Kruskal's Algorithm**

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.

2. **Prim's Algorithm**

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.

Q1. Write a program to store the graph data structure on an adjacency list and generate Minimum Spanning Tree using Prims algorithms

Q2. Write a program to store the graph data structure on an adjacency list and generate Minimum Spanning Tree using Kruskal algorithms.

Assignment – 12

Graph and Hashing

Dijkstra's Algorithm

Dijkstra's algorithm allows us to find the shortest path between any two vertices of a graph. It differs from the minimum spanning tree because the shortest distance between two vertices might not include all the vertices of the graph

Algorithm

1. The very first step is to mark all nodes as unvisited,
2. Mark the picked starting node with a current distance of 0 and the rest nodes with infinity,
3. Now, fix the starting node as the current node,
4. For the current node, analyse all of its unvisited neighbours and measure their distances by adding the current distance of the current node to the weight of the edge that connects the neighbour node and current node,
5. Compare the recently measured distance with the current distance assigned to the neighbouring node and make it as the new current distance of the neighbouring node,
6. After that, consider all of the unvisited neighbours of the current node, mark the current node as visited,
7. If the destination node has been marked visited then stop, an algorithm has ended, and
8. Else, choose the unvisited node that is marked with the least distance, fix it as the new current node, and repeat the process again from step 4.

Q1. Write a program to store the graph data structure on an adjacency list and find shortest path starting from the source vertex using Dijkastra Algorithms.

Hashing

In hashing, large keys are converted into small keys by using hash functions. The values are then stored in a data structure called hash table. The idea of hashing is to distribute entries (key/value pairs) uniformly across an array. Each element is assigned a key (converted key). By using that key you can access the element in O(1) time. Using the key, the algorithm (hash function) computes an index that suggests where an entry can be found or inserted.

Hashing is implemented in two steps:

1. An element is converted into an integer by using a hash function. This element can be used as an index to store the original element, which falls into the hash table.

2. The element is stored in the hash table where it can be quickly retrieved using hashed

key.

hash = hashfunc(key)

index = hash % array_size

In this method, the hash is independent of the array size and it is then reduced to an index (a number between 0 and array_size − 1) by using the modulo operator (%).

Q2. Construct a hash table of size 7 using the hash function H = key mod 7.

i) Insert elements {11, 12, 15, 17, 19, 26, 34, 37, 56, 58} in a hash table using separate chaining.

ii) Modify your program which returns the number of collisions encountered when hashing data[0],data[1], data[2], ...,data[n-1] into a hash table.

iii) Modify this program so that it uses open addressing hash table using linear probe.