Ahsanullah University of Science and Technology (AUST)
Department of Computer Science and Engineering

LABORATORY MANUAL

Course No. : CSE2104
Course Title: Data Structure Lab

For the students of 2nd Year, 1st Semester of
B.Sc. in Computer Science and Engineering program

TABLE OF CONTENTS

COURSE OBJECTIVES

1. Explain various types of data structures in different types of programming problems.
2. Apply various data structures like Array, Linked List, Stack, Queue, Graph, Tree, Heap, Hash tables on some applications like sorting, searching, Hashing techniques, some graph related problems, TOH, Expression etc.
3. Analyze and select appropriate data structures in terms of time and memory complexity to solve different types of programming problems.

PREFFERED PROGRAMMING LANGUAGE/TOOL(S)

- C
- C++
- Java

TEXT/REFERENCE BOOK(S)

- Edward M. Reingold & Wilfred J. Hansen, ***Data Structures***, Addison Wesley School, 1998.
- Seymour Lipschutz, ***Schaum's outline of theory and problems of data structures***, McGraw-Hill, 1986.

ADMINISTRATIVE POLICY OF THE LABORATORY

✔ Students must perform class assessment tasks individually without help of others.

✔ Viva for each program will be taken and considered as a performance.

✔ Plagiarism is strictly forbidden and will be dealt with punishment.

Session 1

Binary Search and Bubble Sort

Binary Search:

Binary search is a searching algorithm which is used for searching specific data from an array (or any index accessible data structure). There are two preconditions to perform binary search on an array.

1. Data must be stored in indexed based data structure. Like array, vector.
2. Data must be sorted in ascending or descending order.

When we perform a binary search we divide the array into two segments. In every operation, we continue searching only one segment and skip other one until we find the value.

Let consider an array **{1, 2, 5, 6, 8, 9, 12, 15, 20}**.

Suppose we need to find the value **12**.

We will check the middle value. Here "middle" value means which element is now at the middle index.

Let us consider **1** is the lowest index and **9** is the highest index.

So, middle index will be **(1+9)/2**(integer division) = **5**.

Below we can see at **5**th index the value is **8**. **8** is less than **12**.

So it is confirmed that the value we are seeking is on the right side of the array.

So next we'll search from index **6** to index **9**.

Now middle becomes **(6+9)/2**(integer division) = **7**.

At **7**th index value is **12**. So value is found.

Index	1	2	3	4	5	6	7	8	9
Value	1	2	5	6	8	9	12	15	20

Pseudo code of binary search on an ascending ordered sorted array:

```
function binary_search(A, n, key):
    low  := 0
    high := n - 1
    while low <= high:
        mid := floor((low + high) / 2)
        if A[mid] < key:
            low := mid + 1
        else if A[mid] > key:
            high := mid - 1
        else:
            return mid
    return unsuccessful
```

Bubble Sort:

Bubble sort is a sorting algorithm. Main idea of bubble sort is we'll put maximum value at topmost index.

Index	10	9	8	7	6	5	4	3	2	1
Value	7	9	11	5	3	6	3	12	8	1

We'll compare i^{th} value with $(i+1)^{th}$ linearly if(i^{th} value > $(i+1)^{th}$ value) then swap them. By doing this we put the maximum value to the top. After comparing all the values from 1 to N, finally we'll get the sorted array.

After the first comparison the array should look like this.

Index	10	9	8	7	6	5	4	3	2	1
Value	12	7	9	11	5	3	6	3	8	1

After second comparison,

Index	10	9	8	7	6	5	4	3	2	1
Value	12	11	7	9	8	5	3	6	3	1

So after 9th comparing, array will be,

Index	10	9	8	7	6	5	4	3	2	1
Value	12	11	9	8	7	6	5	3	3	1

Pseudo code for Bubble Sort:

```
function Bubble_Sort(A):
    for i = 1 to A.length – 1
        for j = A.length downto i+1
              if A[j] < A[j-1]
                  exchange A[j] with A[j-1]
```

Practice Problems

1. Implementation of the "Bubble Sort & Linear Search" Algorithms to create program in C/CPP or JAVA
2. Take a randomly generated array and sort it using bubble sort. Then find an element using linear search.
3. Take a randomly generated array and reverse-sort it using bubble sort. Then find an element using linear search.
4. Take a randomly generated array and sort the 2nd half using bubble sort. Then find the median using linear search.

Session 2

Insertion Sort and Selection Sort

Insertion sort

Insertion sort is a simple sorting algorithm that works the way we sort playing cards in our hands. Insertion sort is a simple sorting algorithm that builds the final sorted array (or list) one item at a time. It is much less efficient on large lists than more advanced algorithms such as quicksort, heap sort, or merge sort. Although still $O(n^2)$, works in a slightly different way. It always maintains a sorted sub list in the lower positions of the list. Each new item is then "inserted" back into the previous sub list such that the sorted sub list is one item larger.

Pseudo Code:

```
INSERTION-SORT(A)
1.  for j = 2 to n
2.      key ← A [j]
3.      // Insert A[j] into the sorted sequence A[1..j-1]
4.      j ← i – 1
5.      while i> 0 and A[i] > key
6.          A[i+1] ← A[i]
7.          i ← i – 1
8.      A[j+1] ← key
```

Example:

Say the array given is {12, 11, 13, 5, 6}

Let us loop for i = 1 (second element of the array) to 5 (Size of input 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

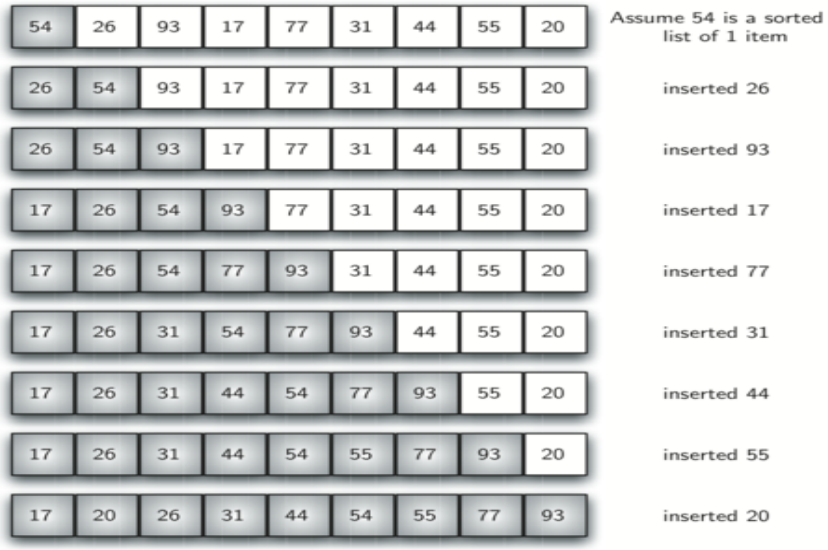
i = 3. 5 will move to the beginning and all other elements from 11 to 13 will move one position ahead of their current position.

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.

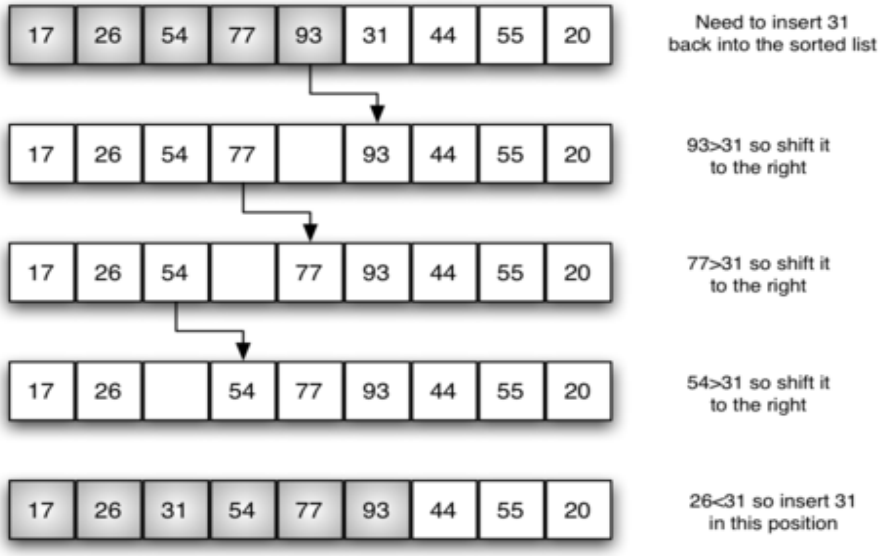
5, 6, 11, 12, 13

The following figure shows the insertion sorting process. The shaded items represent the ordered sub lists as the algorithm makes each pass.



We begin by assuming that a list with one item (position 0) is already sorted. On each pass, one for each item 1 through n−1, the current item is checked against those in the already sorted sub list. As we look back into the already sorted sub list, we shift those items that are greater to the right. When we reach a smaller item or the end of the sub list, the current item can be inserted.

The next figure shows the fifth pass in detail. At this point in the algorithm, a sorted sub list of five items consisting of 17, 26, 54, 77, and 93 exists. We want to insert 31 back into the already sorted items. The first comparison against 93 causes 93 to be shifted to the right. 77 and 54 are also shifted. When the item 26 is encountered, the shifting process stops and 31 is placed in the open position. Now we have a sorted sub list of six items.



Selection Sort

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

Pseudocode of Selection Sort:

```
SELECTION-SORT(A)
1.  for j ← 1 to n-1
2.      smallest ← j
3.       for i ← j + 1 to n
4.            if A[i ] < A[ smallest ]
5.                 smallest ← i
6.        Exchange A[ j ] ↔ A[ smallest ]
```

Example:

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]

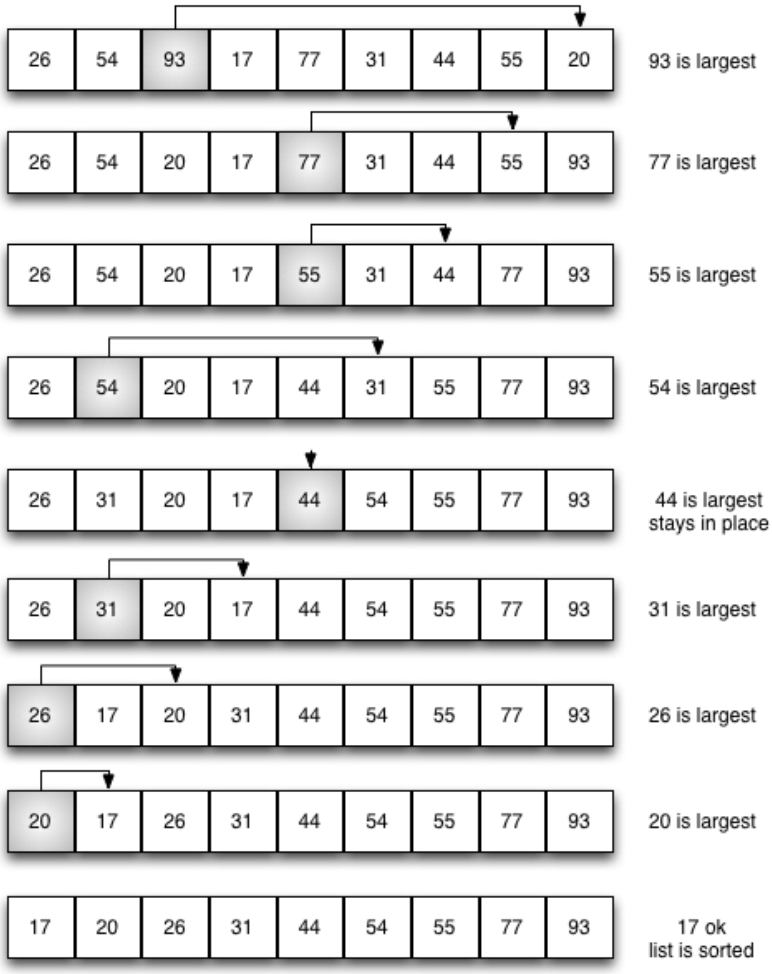
11 12 **22** 25 64

Find the minimum element in arr[3...4]and place it at beginning of arr[3...4]

11 12 22 **25** 64

The **selection sort** improves on the bubble sort by making only one exchange for every pass through the list. In order to do this, a selection sort looks for the largest value as it makes a pass and, after completing the pass, places it in the proper location. As with a bubble sort, after the first pass, the largest item is in the correct place. After the second pass, the next largest is in place. This process continues and requires $\mathbf{n-1}$ passes to sort $\boldsymbol{n}$ items, since the final item must be in place after the $\mathbf{(n-1)}$th pass.

The following figure shows the entire sorting process. On each pass, the largest remaining item is selected and then placed in its proper location. The first pass places 93, the second pass places 77, the third places 55, and so on.



Practice Problems

1. Implementation of the "Insertion, Selection Sort & Binary Search" Algorithm to create a program in C/CPP or JAVA.
2. Take a randomly generated array and sort it using insertion sort. Then find an element using binary search.
3. Take a randomly generated array and sort it using selection sort. Then find an element using binary search.
4. Take a randomly generated array and reverse-sort it using insertion sort. Then find an element using binary search.

Session 3

Merge Sort and Quick Sort

Merge Sort

The **Merge Sort** algorithm closely follows the **Divide and Conquer** paradigm (pattern). The divide and conquer approach involves three main steps:

Divide: Here we divide a problem into a no. of sub-problems having smaller instances of the same problem.

Conquer: Then we conquer the sub-problems by solving them recursively.

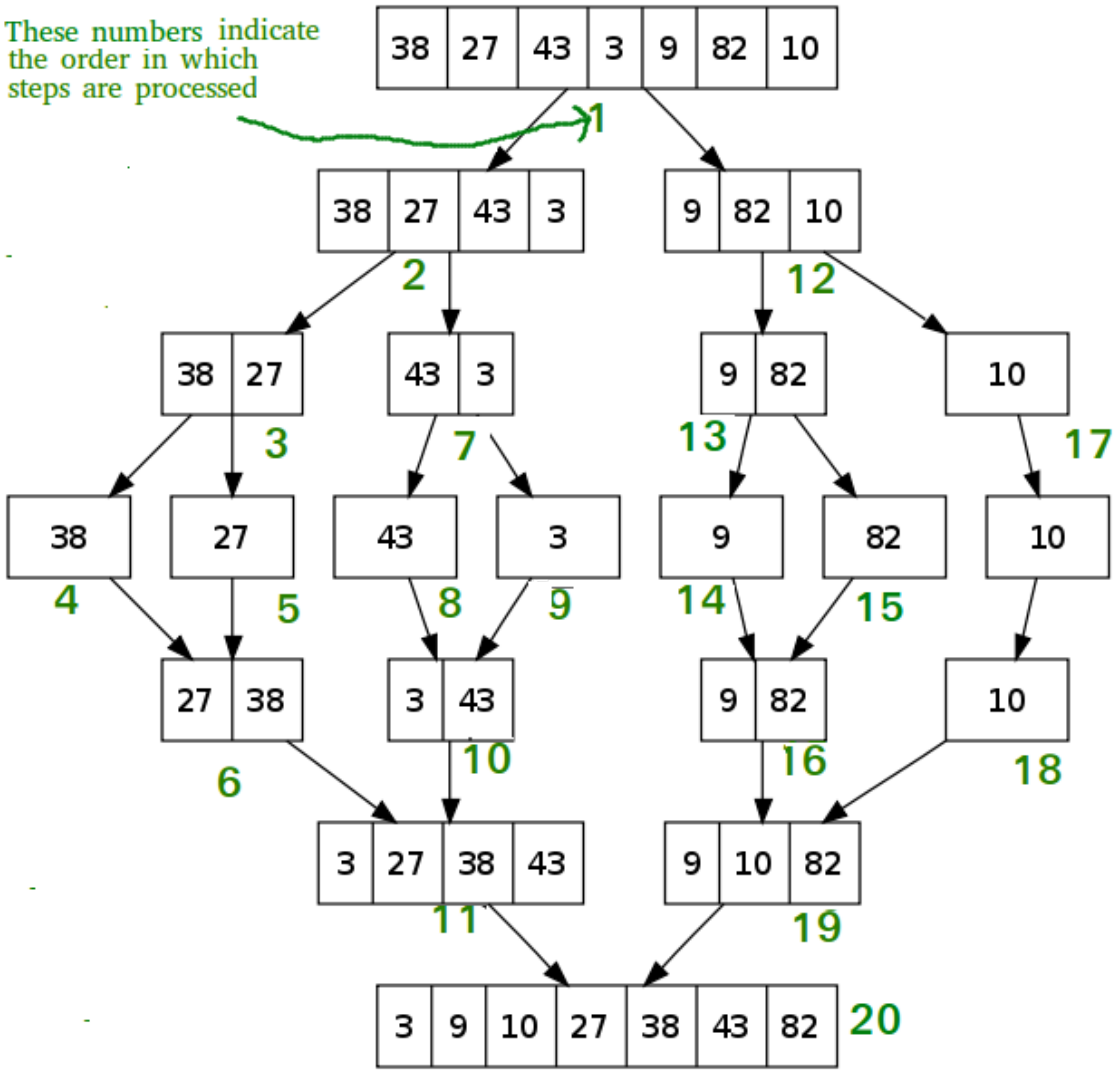
Combine: And in last, we Combines the solutions of the sub-problems into the solution for the original problem.

Algorithm

Ex

```
MRGE-SORT(A, p, r)
    If p < r
    q = ( p + r ) /2
    MERGE-SORT(A, p, q)
    MERGE-SORT(A, q+1, r)
    MERGE(A, p, q, r)
```

```
MERGE (A, p, q, r)
    n1 = q – p +1
    n2 = r – q
    let L [1.. n1 + 1 ] and L [1.. n2 + 1 ]  be new arrays
    for i=1 to  n1
        L[i ]  =  A [ p + i -1]
    for j=1 to n2
        R[ j ]= A[ q + j ]
    L [n1 + 1 ] = ∞
    R [n2 + 1 ] = ∞
    i = 1
    j = 1
    for k = p to r
        if L[i ] ≤R [ j ]
            A[ k ] = L[ i ]
            i = i + 1
        else
            A[ k ] = R [ j ]
            j = j + 1
```



QuickSort

QuickSort is a Divide and Conquer algorithm. It picks an element as pivot and partitions the given array around the picked pivot. There are many different versions of quick sort that pick pivot in different ways.

1. Always pick first element as pivot (implemented below).
2. Always pick last element as pivot.
3. Pick a random element as pivot.
4. Pick median as pivot.

The key process in quick sort is partition. Target of partitions is, given an array and an element x of array as pivot, put x at its correct position in sorted array and put all smaller elements (smaller than x) before x, and put all greater elements (greater than x) after x. All this should be done in linear time.

Algorithm

```
Quicksort(f, l)
    if f < l then
        i = f + 1
        j = l
        while x[i] < x[f] do
            i = i + 1
        while x[j] > x[f] do
            j = j – 1
        while i < j do
            swap x[i] with x[j]
            while x[i] < x[f] do
                i = i + 1
            while x[j] > x[f] do
                j = j – 1
        swap x[j] with x[f]
        Quicksort(f, j-1)
        Quicksort(j+1, l)
```

Example

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**
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 // No change in i and arr[]
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
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
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

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.

Practice Problems

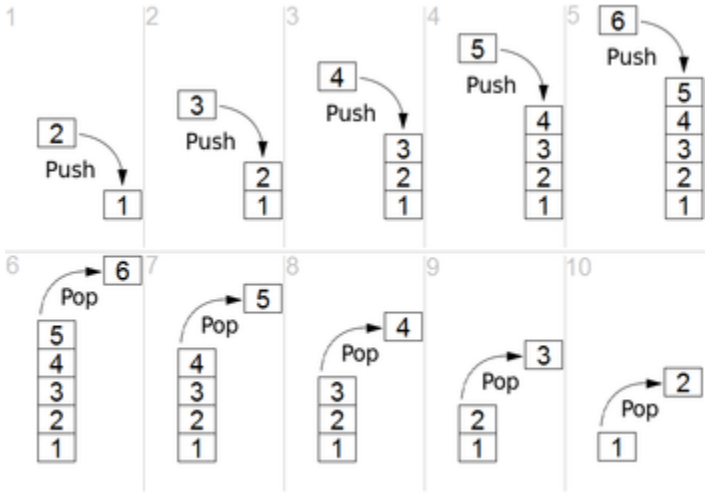
1. Implementation of the "Merge & Quick Sort" Algorithm to create a program in C/CPP or JAVA. Also demonstrate a comparison table (Data movement and Data Comparison) of 5 different Sorting Algorithms.
2. Take an array and sort it using merge sort. Use modulo value of each item as comparison value.
3. Find median of an array using quicksort. You cannot sort the entire array. Additionally, if it is evident that median is in a particular half of the array, you must stop sorting the other half of the array.

Session 4

Stack and Queue

Stacks

A stack is a container of objects that are inserted and removed according to the last-in first-out (LIFO) principle. In the pushdown stacks only two operations are allowed: push the item into the stack, and pop the item out of the stack. In addition, a stack is a limited access data structure - elements can be added and removed from the stack only at the top. Push adds an item to the top of the stack, pop removes the item from the top. A helpful analogy is to think of a stack of books; you can remove only the top book, also you can add a new book on the top.



Stack Implementation using Array

Before implementing actual operations, first follow the below steps to create an empty stack:

Step 1: Include all the header files which are used in the program and define a constant 'SIZE' with specific value.

Step 2: Declare all the functions used in stack implementation.

Step 3: Create a one dimensional array with fixed size (int stack[SIZE])

Step 4: Define an integer variable 'top' and initialize with '-1'. (int top = -1)

Step 5: In main method display menu with list of operations and make suitable function calls to perform operation selected by the user on the stack.

Stack Operations

- **Push(value) - Inserting value into the stack**

  In a stack, push() is a function used to insert an element into the stack. In a stack, the new element is always inserted at top position. Push function takes one integer value as parameter and inserts that value into the stack. We can use the following steps to push an element on to the stack.

  Step 1: Check whether stack is FULL. (top == SIZE-1)

  Step 2: If it is FULL, then display "Stack is FULL!!! Insertion is not possible!!!" and terminate the function.

  Step 3: If it is NOT FULL, then increment top value by one (top++) and set stack[top] to value (stack[top] = value).

- **Pop() - Delete a value from the Stack**

  In a stack, pop() is a function used to delete an element from the stack. In a stack, the element is always deleted from top position. Pop function does not take any value as parameter. We can use the following steps to pop an element from the stack...

  Step 1: Check whether stack is EMPTY. (top == -1)

  Step 2: If it is EMPTY, then display "Stack is EMPTY!!! Deletion is not possible!!!" and terminate the function.

  Step 3: If it is NOT EMPTY, then delete stack[top] and decrement top value by one (top--).

Queue:

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 queue is any queue of consumers for a resource where the consumer that came first is served first.

The difference between stacks and queues is in removing. In a stack we remove the item the most recently added; in a queue, we remove the item the least recently added.

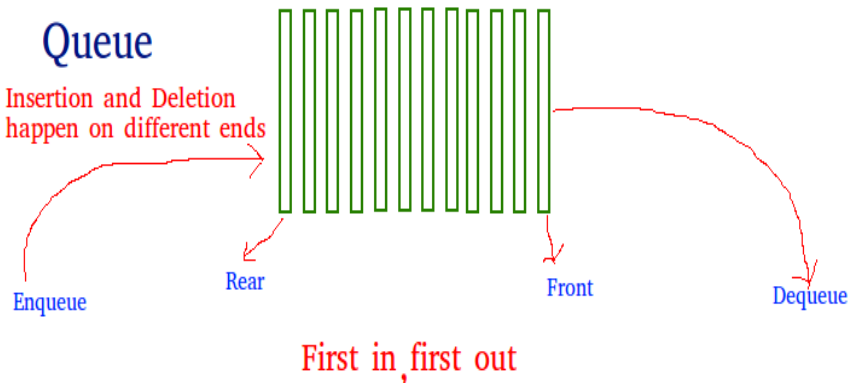
Queue Operations

Mainly the following four basic operations are performed on queue:

Enqueue: Adds an item to the queue. If the queue is full, then it is said to be an Overflow condition.
Dequeue: Removes an item from the queue. The items are popped in the same order in which they are pushed. If the queue is empty, then it is said to be an Underflow condition.
Front: Get the front item from queue.
Rear: Get the last item from queue.

isfull() − Checks if the queue is full.

isempty() − Checks if the queue is empty.



Enqueue Operation

Queues maintain two data pointers, front and rear. Therefore, its operations are comparatively difficult to implement than that of stacks.

The following steps should be taken to enqueue (insert) data into a queue −

Step 1 − Check if the queue is full.

Step 2 − If the queue is full, produce overflow error and exit.

Step 3 − If the queue is not full, increment rear pointer to point the next empty space.

Step 4 − Add data element to the queue location, where the rear is pointing.

Step 5 − return success.

Dequeue Operation

Accessing data from the queue is a process of two tasks − access the data where front is pointing and remove the data after access. The following steps are taken to perform dequeue operation −

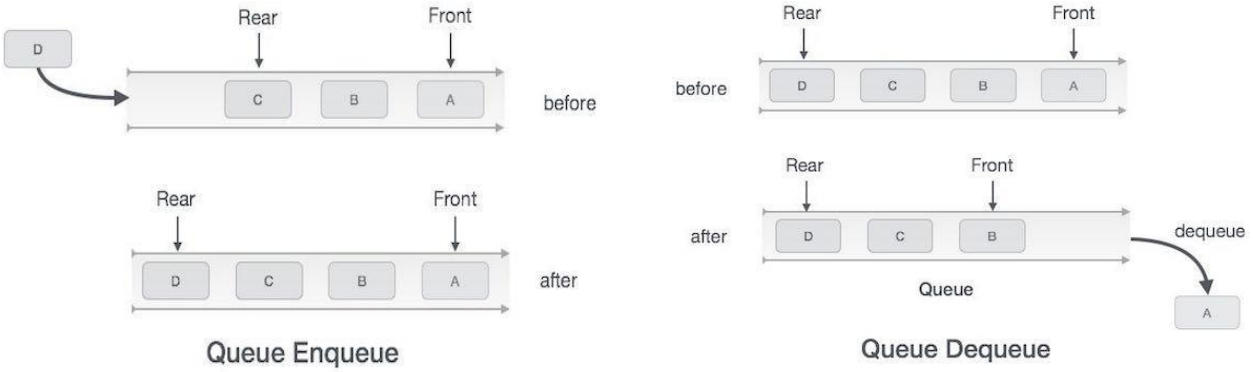
Step 1 − Check if the queue is empty.

Step 2 − If the queue is empty, produce underflow error and exit.

Step 3 − If the queue is not empty, access the data where front is pointing.

Step 4 − Increment front pointer to point to the next available data element.

Step 5 − Return success.



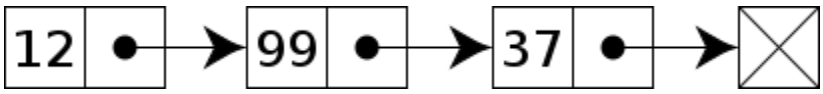
Practice Problems

1. Implementation of the "Stack and Queue using an array". (No STL can be used)
2. Implement queue using stack.
3. Implement stack using queue.
4. Implement a function popmin() in a stack that pops the minimum value in a stack.
5. Implement a function popmin() in a queue that pops the minimum value in a queue.

Session 5

Linked Lists

A **linked list** is a linear collection of data elements, in which linear order is not given by their physical placement in memory. Instead, each element points to the next. It is a data structure consisting of a group of nodes which together represent a sequence. Under the simplest form, each node is composed of data and a reference (in other words, a link) to the next node in the sequence. This structure allows for efficient insertion or removal of elements from any position in the sequence during iteration. In linked lists random access is not allowed.



A linked list whose nodes contain two fields: an integer value and a link to the next node. The last node is linked to a terminator used to signify the end of the list.

The principal benefit of a linked list over a conventional array is that the list elements can easily be inserted or removed without reallocation or reorganization of the entire structure because the data items need not be stored contiguously in memory or on disk, while an array has to be declared in the source code, before compiling and running the program. Linked lists allow insertion and removal of nodes at any point in the list, and can do so with a constant number of operations if the link previous to the link being added or removed is maintained during list traversal.

On the other hand, simple linked lists by themselves do not allow random access to the data, or any form of efficient indexing. Thus, many basic operations — such as obtaining the last node of the list (assuming that the last node is not maintained as separate node reference in the list structure), or finding a node that contains a given datum, or locating the place where a new node should be inserted — may require sequential scanning of most or all of the list elements.

Advantages

- Linked lists are a dynamic data structure, which can grow and be pruned, allocating and deallocating memory while the program is running.
- Insertion and deletion node operations are easily implemented in a linked list.
- Dynamic data structures such as stacks and queues can be implemented using a linked list.
- There is no need to define an initial size for a linked list.
- Items can be added or removed from the middle of list.
- Backtracking is possible in two way linked list.

Types

1. Singly Linked List
2. Doubly Linked List
3. Circular Linked List

Practice Problems

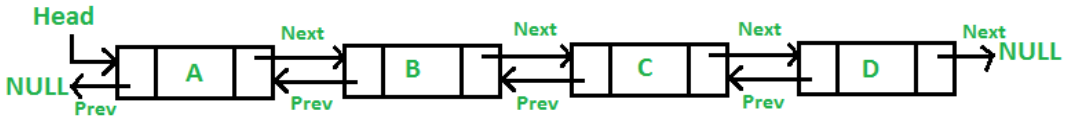
1. Implementation of the "Single Linked List" Algorithm to create a program in C/CPP or JAVA.
2. Insert into and delete from a single sorted Linked List.
3. Print the contents of a simple linked list in reverse order using recursion.
4. Merge two sorted Linked List.
5. Find the median of a sorted Linked List.

Session 6

Doubly and Circular Linked Lists

Doubly Linked Lists

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.



Insert a Node:

A node can be added in four ways:

- At the front of the DLL
- After a given node.
- At the end of the DLL
- Before a given node.

Delete a Node:

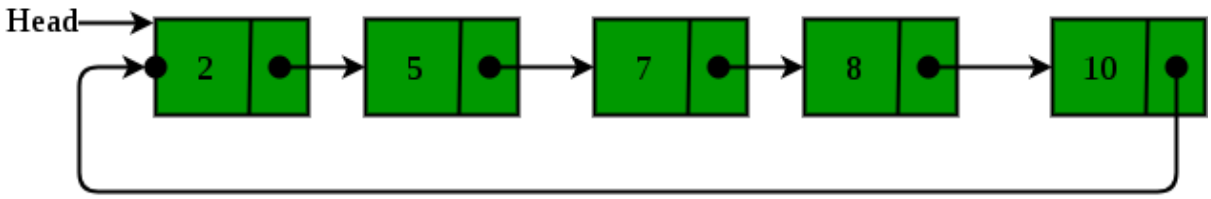
Let the node to be deleted is *del*.
1) If node to be deleted is head node, then change the head pointer to next current head.
2) Set *next* of previous to *del*, if previous to *del* exixts.
3) Set *prev* of next to *del*, if next to *del* exixts.

Advantages:

- A DLL can be traversed in both forward and backward direction
- The delete operation in DLL is more efficient if pointer to the node to be deleted is given.
  In singly linked list, to delete a node, pointer to the previous node is needed. To get this previous node, sometimes the list is traversed. In DLL, we can get the previous node using previous pointer.

Circular Linked List

Circular linked list is a linked list where all nodes are connected to form a circle. There is no NULL at the end. A circular linked list can be a singly circular linked list or doubly circular linked list.



Advantages:

1. Any node can be a starting point. We can traverse the whole list by starting from any point. We just need to stop when the first visited node is visited again.
2. Useful for implementation of queue. 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.
3. 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

Practice Problems

1. Insert into and delete from a Doubly Linked List
2. Reverse a Doubly Linked List
3. Reverse a Doubly Linked List using Recursion
4. Find size of a Doubly Linked List
5. Insert into and delete from a Circular Linked List
6. Convert Singly Linked List into Doubly Linked List
7. Josephus Circle using Circular Linked List

Session 7

Stack and Queue using Linked Lists

Stack Using Linked List

Stack can be implemented using both arrays and linked lists. The limitation, in the case of array, is that we need to define the size at the beginning of the implementation. This makes our stack static. It can also result in "stack overflow" if we try to add elements after the array is full. So, to alleviate this problem, we use a linked list to implement the stack so that it can grow in real time.

Stack Operations:

Step 2 - Define a '**Node**' structure with two members **data** and **next**.

Step 3 - Define a **Node** pointer '**top**' and set it to **NULL**.

Push (value)

We can use the following steps to insert a new node into the stack...

Step 1 - Create a **newNode** with given value.

Step 2 - Check whether stack is **Empty** (**top** == **NULL**)

Step 3 - If it is **Empty**, then set **newNode** → **next** = **NULL**.

Step 4 - If it is **Not Empty**, then set **newNode** → **next** = **top**.

Step 5 - Finally, set **top** = **newNode**.

Pop ()

We can use the following steps to delete a node from the stack...

Step 1 - Check whether **stack** is **Empty** (**top == NULL**).

Step 2 - If it is **Empty**, then display **"Stack is Empty!!! Deletion is not possible!!!"** and terminate the function

Step 3 - If it is **Not Empty**, then define a **Node** pointer '**temp**' and set it to '**top**'.

Step 4 - Then set '**top** = **top** → **next**'.

Step 5 - Finally, delete '**temp**'. (**free(temp)**).

Queue Operations:

Similar to stack, the queue can also be implemented using both arrays and linked lists. But it also has the same drawback of limited size. Hence, we will be using a linked list to implement the queue.

Step 1 - Define a '**Node**' structure with two members **data** and **next**.
Step 2 - Define two **Node** pointers '**front**' and '**rear**' and set both to **NULL**.

Enqueue (value)

We can use the following steps to insert a new node into the queue...

Step 1 - Create a **newNode** with given value and set '**newNode → next**' to **NULL**.

Step 2 - Check whether queue is **Empty** (**rear** == **NULL**)

Step 3 - If it is **Empty** then, set **front** = **newNode** and **rear** = **newNode**.

Step 4 - If it is **Not Empty** then, set **rear → next** = **newNode** and **rear** = **newNode**.

Dequeue:

We can use the following steps to delete a node from the queue...

Step 1 - Check whether **queue** is **Empty** (**front == NULL**).

Step 2 - If it is **Empty**, then display **"Queue is Empty!!! Deletion is not possible!!!"** and terminate from the function

Step 3 - If it is **Not Empty** then, define a Node pointer '**temp**' and set it to '**front**'.

Step 4 - Then set '**front** = **front → next**' and delete '**temp**' (**free(temp)**).

Practice Problems

1. Implementation of the stack and queue using Linked List
2. Implement queue using stack
3. Implement stack using queue
4. Implement a function popmin() in a stack/queue that pops the minimum value in a stack

Session 8

Tower of Hanoi

Tower of Hanoi is a mathematical puzzle where we have three rods and n disks. The objective of the puzzle is to move the entire stack to another rod, obeying the following simple rules:

1. Only one disk can be moved at a time.
2. Each move consists of taking the upper disk from one of the stacks and placing it on top of another stack i.e. a disk can only be moved if it is the uppermost disk on a stack.
3. No disk may be placed on top of a smaller disk.

Pseudo-Code for Non - Recursive approach:

```
S ← empty stack
S ← (n,1,2,3)
While S ≠ empty do
        (n, i, j, k) ← S
        if n == 1 then move the top disk from i to k
        else
                S ← (n-1, j, i, k)
                S ← (1, i, j, k)
                S ← (n-1, j, i, k)
```

Pseudo-Code for Recursive approach:

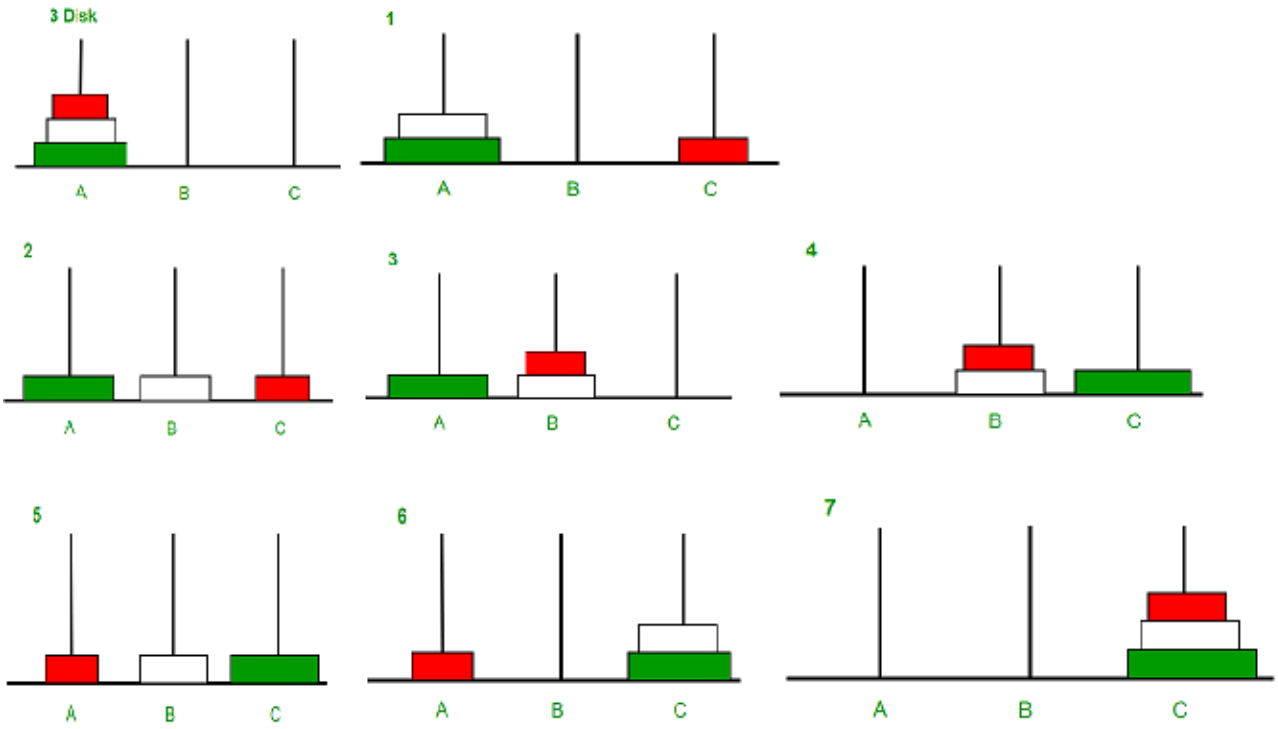
```
Hanoi (n, i, j, k)
    if (n == 1) then
            move the top disk from i to k
    else
        Hanoi (n-1, i, k, j)
        move the top disk from i to k
        Hanoi (n-1, j, i, k)
```

Example:

Let Hanoi method uses these parameters (num_of_disks, src, aux, dest) as (n, A, B, C).

if n = 3 then,

1. Move the disk 1 from 'A' to 'C'
2. Move the disk 2 from 'A' to 'B'
3. Move the disk 1 from 'C' to 'B'
4. Move the disk 3 from 'A' to 'C'
5. Move the disk 1 from 'B' to 'A'
6. Move the disk 2 from 'B' to 'C'
7. Move the disk 1 from 'A' to 'C'



Practice Problems

1. Implementation of the "Tower of Hanoi (Recursive and non-Recursive)" Algorithms to create a program in C/CPP or JAVA.
2. Implement tower of Hanoi in recursive fashion.
3. Implement tower of Hanoi in non-recursive fashion.
4. Print tower of Hanoi steps when each pole has one disk initially.

Session 9

Infix and Postfix Notation

Procedure for Postfix Conversion

1. Scan the Infix string from left to right.
2. Initialize an empty stack.
3. If the scanned character is an operand, add it to the Postfix string.
4. If the scanned character is an operator and if the stack is empty push the character to stack.
5. If the scanned character is an Operator and the stack is not empty, compare the precedence of the character with the element on top of the stack.
6. If top Stack has higher precedence over the scanned character pop the stack else push the scanned character to stack. Repeat this step until the stack is not empty and top Stack has precedence over the character.
7. Repeat 4 and 5 steps till all the characters are scanned.
8. After all characters are scanned, we have to add any character that the stack may have to the Postfix string.
9. If stack is not empty add top Stack to Postfix string and Pop the stack.
10. Repeat this step as long as stack is not empty.

Algorithm for Postfix Conversion

1. Create a stack
2. for each character 't' in the input stream {
   - if (t is an operand)
     output t
   - else if (t is a right parentheses){
     POP and output tokens until a left parentheses is popped(but do not output)
     }
   - else {
     POP and output tokens until one of lower priority than t is encountered or a left parentheses is encountered
     or the stack is empty
     PUSH t
     }
3. POP and output tokens until the stack is empty.

For better understanding, let us trace out an example **A * B – (C + D) + E**

INPUT CHARACTER	OPERATION ON STACK	STACK	POSTFIX EXPRESSION
A		Empty	A
*	Push	*	A
B		*	A B
–	Check and Push	–	A B *
(	Push	– (	A B *
C		– (	A B * C
+	Check and Push	– (+	A B * C
D		– (+	A B * C D
)	Pop and append to postfix till '('	–	A B * C D +
+	Check and push	+	A B * C D + –
E		+	A B * C D + – E
End of Input	Pop till Empty	Empty	A B * C D + – E +

So our final **POSTFIX Expression :- A B * C D + – E +**

Practice Problems

1. Implementation of the "Infix to Postfix format" Algorithm to create a program in C/CPP or JAVA.
2. Implementation of the "Postfix Evaluation" Algorithm to create a program in C/CPP or JAVA.
3. Implementation of the "Prefix to Postfix format" Algorithm to create a program in C/CPP or JAVA.

Session 10

Breadth-First Search and Depth-First Search

BFS

Breadth First Search (BFS) algorithm traverses a graph in a breadthward motion and uses a queue to remember to get the next vertex to start a search, when a dead end occurs in any iteration.

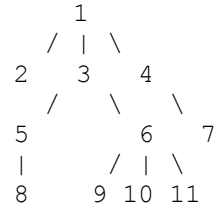
Pseudocode:

Input: A graph *Graph* and a *starting vertex root* of *Graph*

Output: Goal state. The *parent* links trace the shortest path back to *root*

```
1  procedure BFS(G,start_v):
2      let Q be a queue
3      label start_v as discovered
4      Q.enqueue(start_v)
5      while Q is not empty
6          v = Q.dequeue()
7          if v is the goal:
8              return v
9          for all edges from v to w in G.adjacentEdges(v) do
10             if w is not labeled as discovered:
11                 label w as discovered
12                 w.parent = v
13                 Q.enqueue(w)
```

Example:



BFS: 1 2 3 4 5 6 7 8 9 10 11

DFS:

Depth-first search (DFS) is an algorithm for traversing or searching tree or graph data structures. One starts at the root(selecting some arbitrary node as the root in the case of a graph) and explores as far as possible along each branch before backtracking.

Pseudocode:

Input: A graph *G* and a vertex *v* of G

Output: All vertices reachable from *v* labeled as discovered

A recursive implementation of DFS:

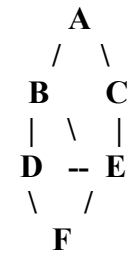
```
1  procedure DFS(G,v):
2      label v as discovered
3      for all directed edges from v to w that are in G.adjacentEdges(v)
do
4          if vertex w is not labeled as discovered then
5              recursively call DFS(G,w)
```

The order in which the vertices are discovered by this algorithm is called the lexicographic order.

A non-recursive implementation of DFS:

```
1  procedure DFS-iterative(G,v):
2      let S be a stack
3      S.push(v)
4      while S is not empty
5          v = S.pop()
6          if v is not labeled as discovered:
7              label v as discovered
8              for all edges from v to w in G.adjacentEdges(v) do
9                  S.push(w)
```

Example:



DFS: ABDEFC

Practice Problems

1. Implementation of the "Breadth First Search & Depth First Search" Algorithms to create a program in C/CPP or JAVA.

2. Finding height of a tree using DFS.
3. Finding height of each branch of root of a tree using DFS.
4. Finding number of nodes in each level of a tree using BFS.
5. Finding total number of nodes in a tree using BFS.

Session 11

Tree Traversal

In computer science, **tree traversal** (also known as **tree search**) is a form of graph traversal and refers to the process of visiting (checking and/or updating) each node in a tree data structure, exactly once. Such traversals are classified by the order in which the nodes are visited.

Preorder:
Algorithm Preorder(tree)

1. Visit the root.
2. Traverse the left subtree, i.e., call Preorder(left-subtree)
3. Traverse the right subtree, i.e., call Preorder(right-subtree)

Pseudo code

Recursive

```
preorder(node)
        if (node = null)
                return
        visit(node)
        preorder(node.left)
        preorder(node.right)
```

Non recursive

```
iterativePreorder(node)
        if (node = null)
                return
        s ← empty stack
        s.push(node)
        while (not s.isEmpty())
                node ← s.pop()
                visit(node)
                 //right child is pushed first so that left is processed first
                if (node.right ≠ null)
                        s.push(node.right)
                if (node.left ≠ null)
```

```
                    s.push(node.left)
```

Postorder:

Algorithm Postorder(tree)

1. Traverse the left subtree, i.e., call Postorder(left-subtree)
2. Traverse the right subtree, i.e., call Postorder(right-subtree)
3. Visit the root.

Pseudo code

Recursive

```
postorder(node)
        if (node = null)
                return
        postorder(node.left)
        postorder(node.right)
        visit(node)
```

Non recursive

```
iterativePostorder(node)
        s ← empty stack
        lastNodeVisited ← null
        while (not s.isEmpty() or node ≠ null)
                if (node ≠ null)
                        s.push(node)
                        node ← node.left
                else
                        peekNode ← s.peek()
                        // if right child exists and traversing node
                        // from left child, then move right
                if (peekNode.right ≠ nullandlastNodeVisited ≠ peekNode.right)
                        node ← peekNode.right
                else
```

```
                    visit(peekNode)
                lastNodeVisited ← s.pop()
```

Inorder:

Algorithm Inorder(tree)

1. Traverse the left subtree, i.e., call Inorder(left-subtree)
2. Visit the root.
3. Traverse the right subtree, i.e., call Inorder(right-subtree)

Pseudo code

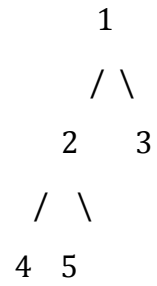
Recursive

```
inorder(node)
        if (node = null)
                return
        inorder(node.left)
        visit(node)
        inorder(node.right)
```

Non recursive

```
iterativeInorder(node)
        s ← empty stack
        while (nots.isEmpty() or node ≠ null)
                if (node ≠ null)
                        s.push(node)
                        node ← node.left
                else
                        node ← s.pop()
                        visit(node)
                        node ← node.right
```

Example:



(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

Practice Problems

1. Implementation of the "Complete Binary Tree" Algorithms to create a program in C/CPP or JAVA. Also a program to implement "Pre Order, In Order, Post Order Traversal of Complete Binary Tree (Recursive & Non Recursive)" Algorithms.

Session 12

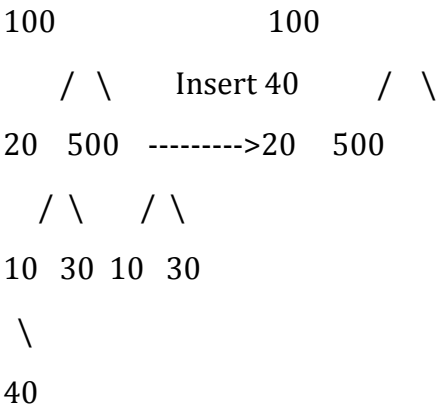
Binary Search Tree

A binary search tree is a rooted binary tree, whose internal nodes each store a key (and optionally, an associated value) and each have two distinguished sub-trees, commonly denoted *left* and *right*. The tree additionally satisfies the binary search property, which states that the key in each node must be greater than or equal to any key stored in the left sub-tree, and less than or equal to any key stored in the right sub-tree (The leaves (final nodes) of the tree contain no key and have no structure to distinguish them from one another. Leaves are commonly represented by a special leaf or nil symbol, a NULL pointer, etc.)

Binary Search Tree Insert:

```
struct node* insert(struct node* node, int key)
{
   /* If the tree is empty, return a new node */
   if (node == NULL) return newNode(key);

   /* Otherwise, recur down the tree */
   if (key < node->key)
      node->left  = insert(node->left, key);
   else if (key > node->key)
      node->right = insert(node->right, key);

   /* return the (unchanged) node pointer */
   return node;
}
```

Example



Binary Search Tree Search:

```
struct node* search(struct node* root, int key)
{
   // Base Cases: root is null or key is present at root
   if (root == NULL || root->key == key)
      return root;

   // Key is greater than root's key
   if (root->key < key)
      return search(root->right, key);

   // Key is smaller than root's key
   return search(root->left, key);
}
```

Example:

```
100
    / \
  20   500
  / \
10  30
```

If key is 500 show found

If key is 600 show not found

Binary search Tree Successor:

If right subtree of *node* is not *NULL*, then *succ* lies in right subtree. Do following.
Go to right subtree and return the node with minimum key value in right subtree.

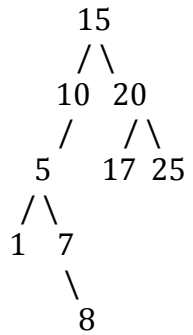
If right sbtree of *node* is NULL, then start from root and us search like technique. Do following.

Travel down the tree, if a node's data is greater than root's data then go right side, otherwise go to left side.

```
struct node * inOrderSuccessor(struct node *root, struct node *n)
{
   // step 1 of the above algorithm
   if( n->right != NULL )
      returnminValue(n->right);

   struct node *succ = NULL;

   // Start from root and search for successor down the tree
```

```
    while (root != NULL)
    {
       if (n->data < root->data)
       {
          succ = root;
          root = root->left;
       }
       else if (n->data > root->data)
          root = root->right;
       else
         break;
    }

    Return succ;
}
```

Example:



InorderSuccesor of 1 is 5

InorderSuccesor of 8 is 10

Practice Problems

1. Implementation of the "Binary Search Tree & insert, search, finding successor of BST" Algorithms to create a program in C/CPP or JAVA.
2. Finding number of duplicate entries in a BST
3. Finding the list of unique values in a BST
4. Finding the difference between the left- and right-subtree of root in terms of summative value

Session 13

Hashing

Hashing is the process of mapping large amount of data item to a smaller table with the help of a hashing function. The essence of hashing is to facilitate the next level searching method when compared with the linear or binary search. The advantage of this searching method is its efficiency to hand vast amount of data items in a given collection (i.e. collection size).

Following are the basic primary operations of a hash table.

Search − Searches an element in a hash table.

Insert − inserts an element in a hash table.

Delete − Deletes an element from a hash table.

Hashing is implemented in two steps:

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.

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 (%).

Hashing methods:

1. Direct method
2. Modulo-division
3. Digit extraction
4. Comparison
5. Multiplication

Direct Method

In direct hashing the key is the address without any algorithmic manipulation.
Direct hashing is limited, but it can be very powerful because it guarantees that there are no synonyms and therefore no collision.

Modulo-division Method

- This is also known as division remainder method.

- This algorithm works with any list size, but a list size that is a prime number produces fewer collisions than other list sizes.
- The formula to calculate the address is:

Address = key MODULO listsize + 1

Where listsize is the number of elements in the arrary.

Example:
Given data keys are: 137456 214562 140145
137456 % 19 +1 = 11
214562 % 19 + 1 = 15
140145 % 19 + 1 = 2

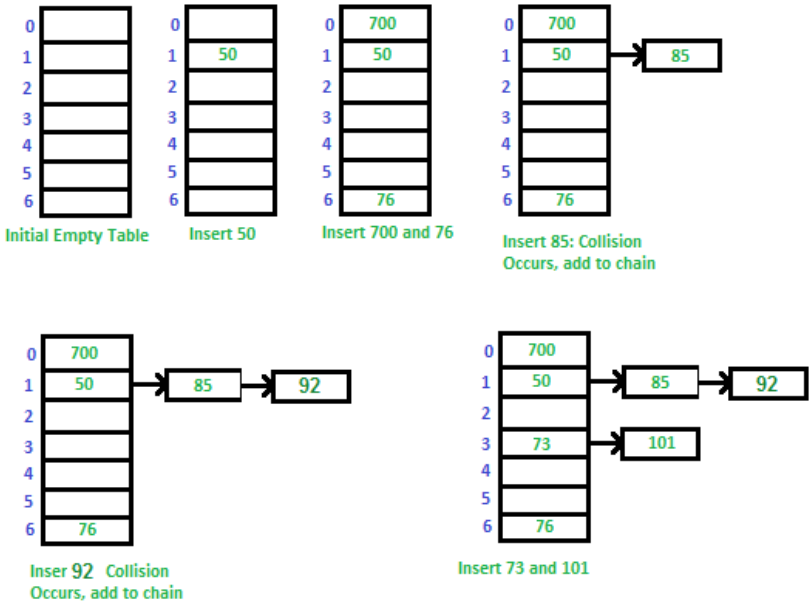
Digit-extraction Method

Using digit extraction selected digits are extracted from the key and used as the address.

Example:
Using six-digit employee number to hash to a three digit address (000-999), we could select the first, third, and fourth digits(from the left) and use them as the address.

The keys are:

379452 -> 394
121267 -> 112
378845 -> 388



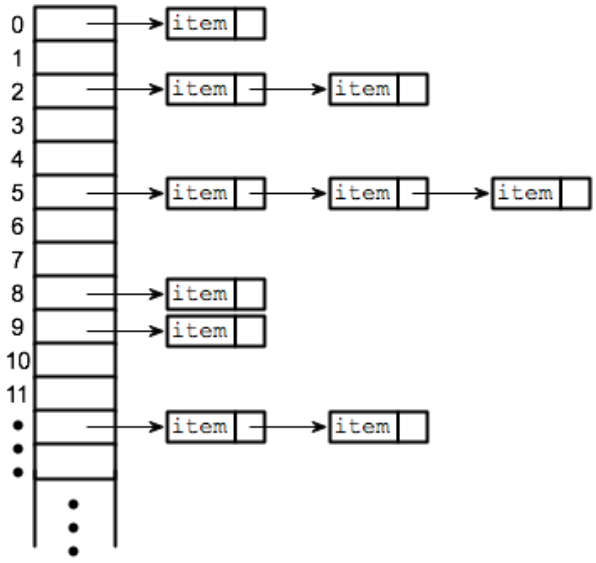
Hashing Code:

```
structDataItem {
int data;
int key;
};
inthashCode(int key){
return key % SIZE;
}
voidinsert(intkey,int data){
structDataItem*item =(structDataItem*)malloc(sizeof(structDataItem));
item->data = data;
item->key = key;
//get the hash
inthashIndex=hashCode(key);
//move in array until an empty or deleted cell
while(hashArray[hashIndex]!= NULL &&hashArray[hashIndex]->key !=-1){
//go to next cell
++hashIndex;
//wrap around the table
hashIndex%= SIZE;
}
hashArray[hashIndex]= item;
}
structDataItem*search(int key){
//get the hash
inthashIndex=hashCode(key);
//move in array until an empty
while(hashArray[hashIndex]!= NULL){
```

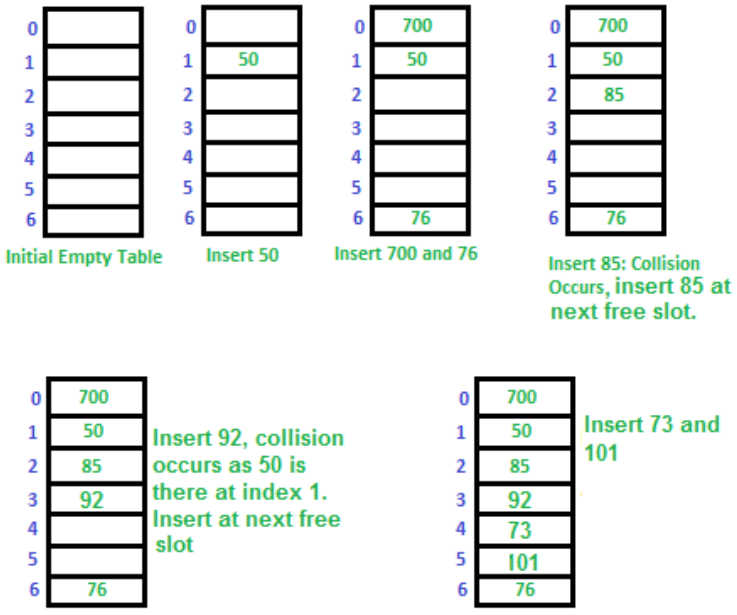
```
if(hashArray[hashIndex]->key == key)
returnhashArray[hashIndex];
//go to next cell
++hashIndex;
//wrap around the table
hashIndex%= SIZE;
}
return NULL;
}
```

Colision Resolve Method:

SEPARATE CHAINING:



Open Addressing:



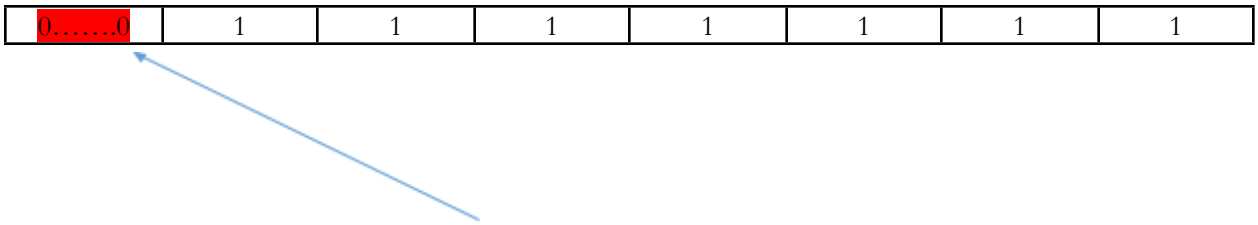
Practice Problems

1. Implementation of the "Hash Function" (using array to insert and search). Also one (any) "Collision resolution Method" Algorithms to create a program in C/CPP or JAVA.
2. Implement a hashing module in an array. Use modulo value as hash function.

Session 14

Packed Words

Packed word is a data structure that is best suited for memory optimization. We know that, when we declare a variable x as in integer, it allocates 16 bit memory. That means, x capable of storing a maximum value of 65535. But in cases when we are dealing with certain numbers, let's say, someone's age, then we can presume that a person is not capable of living more than 115/120 years. So if we declare a variable to age, it will allocate 16 bit data, but in reality to store the above mentioned age, we need a maximum of 7 bits(capable of storing up to 127).



The rest of the bits remain unused, which is waste of memory space. Now if we use only 7 bits to store the age, we can use the rest of the bits to store other important information about a person, like sex, sibling, kids. We know that a person can be male or female, which will take 1 bit, the rest of the 8 bits can be used for kids and siblings (4 bit each, assuming max value is 15).

So in a 16 bit space, we are packing 4 different integers each containing different information but utilizing all the space that is allocated in the memory.

Let's look at the table,

7 bits for age	4 bits for siblings	4 bits for kids	1 bit for sex

Here, age is going to take up the first 7 bits of our integer used for packing. To make this happen, we have to left shift the value by 9 bits, (16 – 7 = 9), siblings will use the first 4 bits of the remaining 9 bits, so it will be left shifted (9 – 4 = 5) times, and so on.

After left shifting, if we do an OR operation on all the left shifted values, we will have an integer packcontaining all 4 of those information.

To extract the information from the packed word, we will have to do an AND (&) operation on them with the maximum integer possible with the allocated bits, after doing right shifts according to their bits (like in packing).

Example:

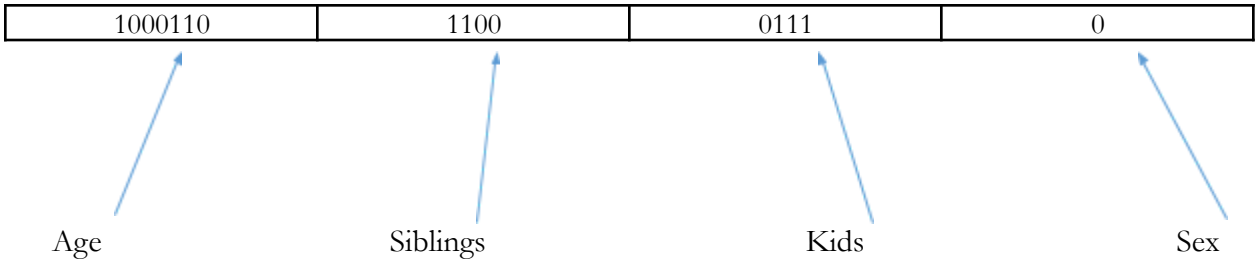
Let, age = 70 $(1000110)_2$

Siblings = 12 $(1100)_2$

Kids = 7 $(0111)_2$

Sex = 0 [0 for male, 1 for female]

After packing this will look something like this,



Practice Problems

1. Implementation of the "Packed Word Technique" Algorithms to create a program in C/CPP or JAVA. [Where one 16 bit array should be packed from four different arrays containing 16 bit data each but respectively 9, 12, 12, 15 bit data unused (wasted)]
2. Unpack the data from a packed word.
3. Modify the data from a packed word.

MID TERM EXAMINATION

There will be a 40-minutes written mid-term examination. Different types of questions will beincluded such as MCQ, writing code fragments etc.

FINAL TERM EXAMINATION

There will be a one-hour written examination. Different types of questions will be included such as MCQ, writing programs etc.