INDALA COLLEGE OF ENGINEERNG

Bapsai, Kalyan

Department of Computer Engineering

Data Structure Lab Manual

SEMISTER-III

Practical List

Sr. No	Name of the Experiment	Page No.
1	Implement Stack ADT using array.	
2	Convert an Infix expression to Postfix expression using stack ADT.	
3	Applications of Stack ADT.	
4	Implement Priority Queue ADT using array.	
5	Implement Singly Linked List ADT.	
6	Implement Binary Search Tree ADT using Linked List	
7	Implement Graph Traversal techniques: a) Breadth First Search b) Depth First Search	

EXPERIMENT NO: 1

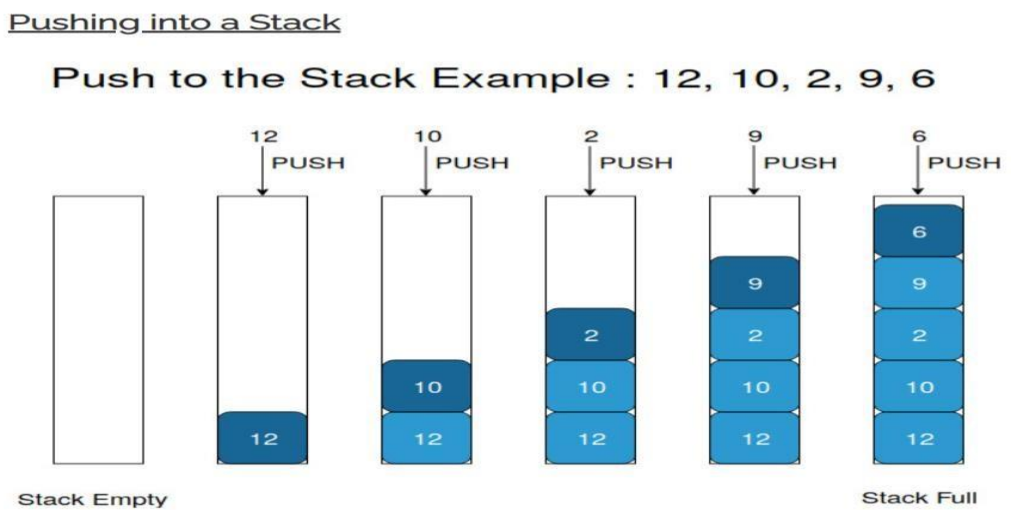
AIM: Implement Stack ADT using array.

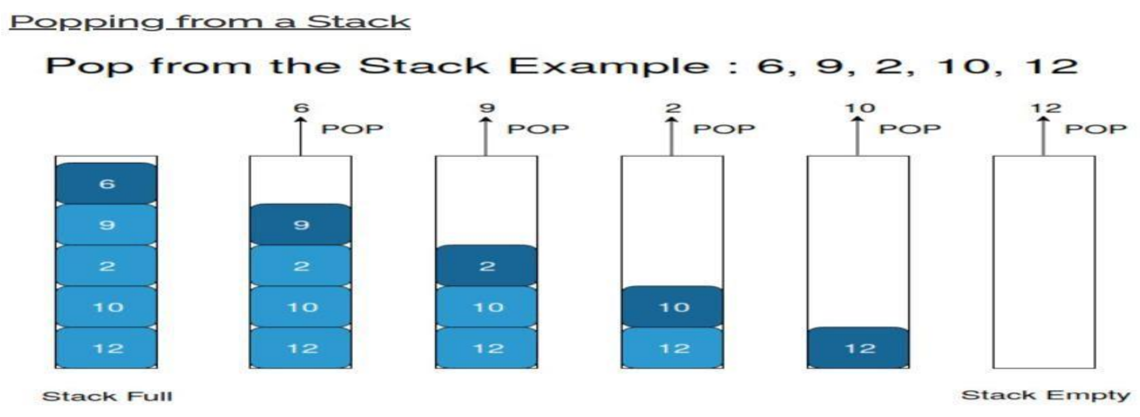
Theory:

- **I**magine a pile of books, with books stacked one over the other. From this pile of books, you can either put another book on top or remove a book from the top.
- The book which is at the bottom of the pile is the last one to be taken out, while the books at the top are removed first. Books can only be added to the top of the pile.
- Let the action of putting a book on the top be called as push and let the action of removing a book be called pop. A type of structure, similar to the example of the pile of books, can be represented as a data structure. Such a data structure is known as a stack.

Stack Operations and Applications

Just like how we saw in the example of a stack of books, a stack data structure has two types of operations
: push and pop. As we can see, a stack is an example of a last in, first out data structure (LIFO). That is, an element that is pushed last into a stack is the first to be popped out. Stacks have many applications. Lets explore a few of them. Reversing a word : Think about how you would reverse a word using a stack. First all the letters are pushed into the stack and then popped out one by one to get the reversed word. This would take linear time O(n). Undoing Changes in a Text Editor : A stack is also commonly used in text editors. Changes that the user makes are pushed into a stack. While undoing, they are popped out





Conclusion:

Program: (Printout)

Output: (Printout)

EXPERIMENT NO: 2

.

AIM: Implement Infix expression to Postfix expression using stack ADT

Theory:
Infix expression: The expression of the form a op b. When an operator is in-between every pair of operands.
Postfix expression: The expression of the form a b op. When an operator is followed for every pair of operands.
Why postfix representation of the expression?
The compiler scans the expression either from left to right or from right to left.
Consider the below expression: a op1 b op2 c op3 d
If op1 = +, op2 = *, op3 = +
The compiler first scans the expression to evaluate the expression b * c, then again scans the expression to add a to it. The result is then added to d after another scan.
The repeated scanning makes it very in-efficient. It is better to convert the expression to postfix(or prefix) form before evaluation.
The corresponding expression in postfix form is abc*+d+. The postfix expressions can be evaluated easily using a stack. We will cover postfix expression evaluation in a separate post.

Algorithm
1. Scan the infix expression from left to right.
2. If the scanned character is an operand, output it.
3. Else,
1 If the precedence of the scanned operator is greater than the precedence of the operator in the stack(or the stack is empty or the stack contains a '('), push it.
2 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.)
4. If the scanned character is an '(', push it to the stack.
5. If the scanned character is an ')', pop the stack and output it until a '(' is encountered, and discard both the parenthesis.
6. Repeat steps 2-6 until infix expression is scanned.
7. Print the output
8. Pop and output from the stack until it is not empty.

Conclusion:

Program: (Printout)

Output: (Printout)

EXPERIMENT NO:3

AIM: Implement Iterative Tower of Hanoi (Applications of Stack ADT.)

Theory :

The Tower of Hanoi is a mathematical puzzle. It consists of three poles and a number of disks of different sizes which can slide onto any poles. The puzzle starts with the disk in a neat stack in ascending order of size in one pole, the smallest at the top thus making a conical shape. The objective of the puzzle is to move all the disks from one pole (say ‘source pole’) to another pole (say ‘destination pole’) with the help of the third pole (say auxiliary pole).

The puzzle has the following two rules:

1. You can’t place a larger disk onto a smaller disk
2. Only one disk can be moved at a time

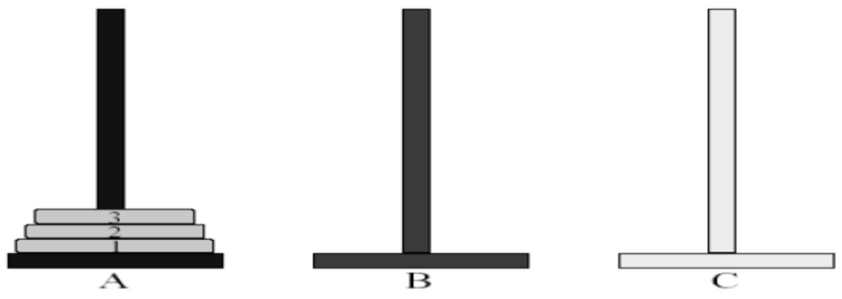
Iterative Algorithm:

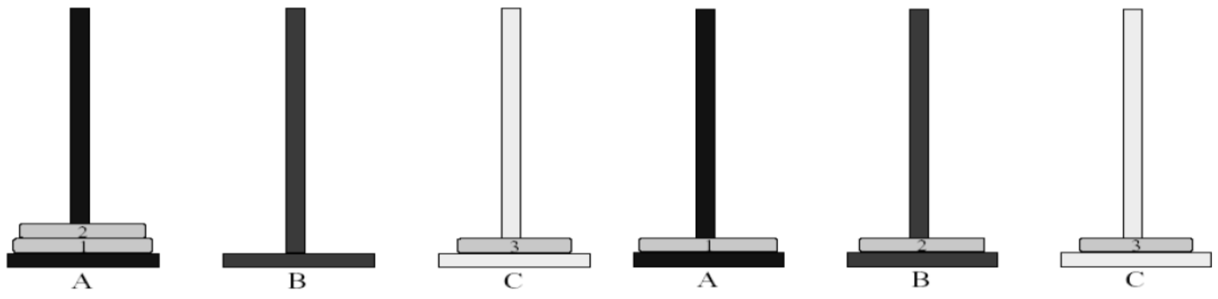
```
1. Calculate the total number of moves required i.e. "pow(2, n)
   - 1" here n is number of disks.
2.If number of disks (i.e. n) is even then interchange destination
   pole and auxiliary pole.
3. for i = 1 to total number of moves:
     if i%3 == 1:
    legal movement of top disk between source pole and
        destination pole
     if i%3 == 2:
    legal movement top disk between source pole and
        auxiliary pole
     if i%3 == 0:
        legal movement top disk between auxiliary pole
        and destination pole
```

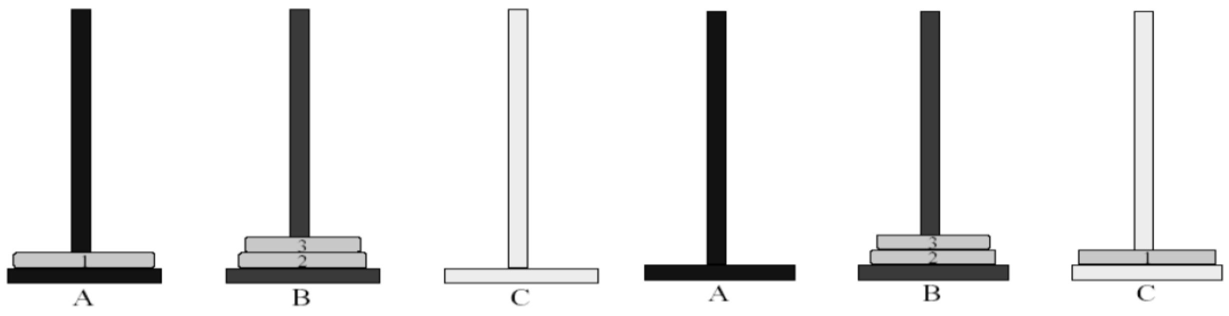
Example:

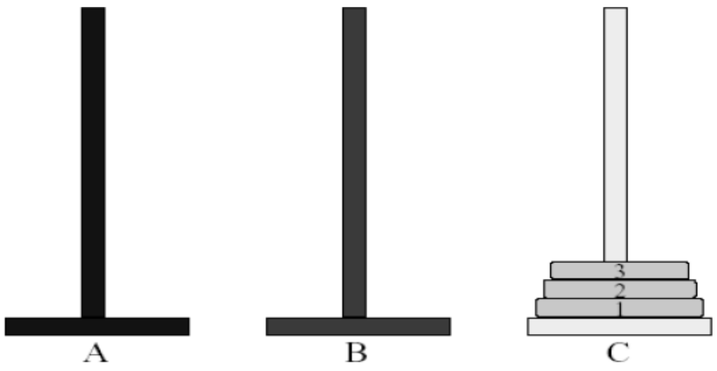
Let us understand with a simple example with 3 disks:

So, total number of moves required = 7









So, after all these destination poles contains all the in order of size.
After observing above iterations, we can think that after a disk other than the smallest disk is moved, the next disk to be moved must be the smallest disk because it is the top disk resting on the spare pole and there are no other choices to move a disk.
Above figure Source Pole =A=S; Auxillary pole= B=A; Destination pole= C= D;

Conclusion:

Program: (Printout)

Output: (Printout)

EXPERIMENT NO: 4

AIM: To Implement Priority Queue ADT using array

Theory :

Priority Queue is an extension of the Queue data structure where each element has a particular priority associated with it. It is based on the priority value, the elements from the queue are deleted.

- **enqueue():** This function is used to insert new data into the queue.
- **dequeue():** This function removes the element with the highest priority from the queue.
- **peek()/top():** This function is used to get the highest priority element in the queue without removing it from the queue.

Approach: The idea is to create a structure to store the value and priority of the element and then create an array of that structure to store elements. Below are the functionalities that are to be implemented:

- **enqueue():** It is used to insert the element at the end of the queue.
- **peek():**
    - Traverse across the priority queue and find the element with the highest priority and return its index.
    - In the case of multiple elements with the same priority, find the element with the highest value having the highest priority.
- **dequeue():**
    - Find the index with the highest priority using the **peek()** function let's call that position as **ind**, and then shift the position of all the elements after the position **ind** one position to the left.
    - Decrease the size by one.

Application of Priority Queue:

- For Scheduling Algorithms the CPU has to process certain tasks having priorities. The process of having higher priority gets executed first.
- In a time-sharing computer system, the process of waiting for the CPU time gets loaded in the priority queue.
- A Sorting-priority queue is used to sort heaps.

Conclusion:

Program: (Printout)

Output: (Printout)

.

EXPERIMENT NO:5

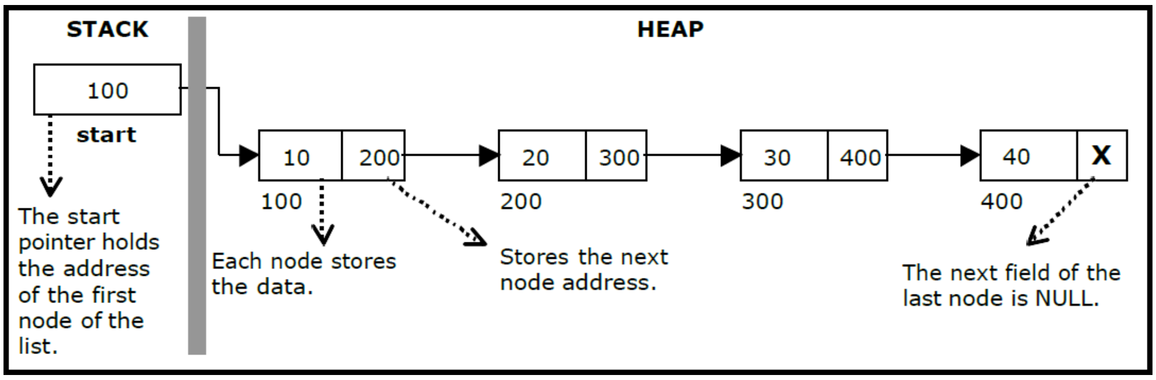
AIM: To Implement Singly Linked List ADT.

THEORY:

A linked list is a non-sequential collection of data items. It is a dynamic data structure. For every data item in a linked list, there is an associated pointer that would give the memory location of the next data item in the linked list.

The data items in the linked list are not in consecutive memory locations. They may be anywhere, but the accessing of these data items is easier as each data item contains the address of the next data item

A linked list allocates space for each element separately in its own block of memory called a "node". The list gets an overall structure by using pointers to connect all its nodes together like the links in a chain. Each node contains two fields; a "data" field to store whatever element, and a "next" field which is a pointer used to link to the next node. Each node is allocated in the heap using malloc(), so the node memory continues to exist until it is explicitly de-allocated using free(). The front of the list is a pointer to the ―start node


STACK
HEAP
100
start
The start pointer holds the address of the first node of the list.
10
200
100
20
300
200
30
400
300
40
X
400
Each node stores the data.
Stores the next node address.
The next field of the last node is NULL.

Before writing the code to build the above list, we need to create a **start** node**,** used to create and access other nodes in the linked list. The following structure definition will do (see figure):

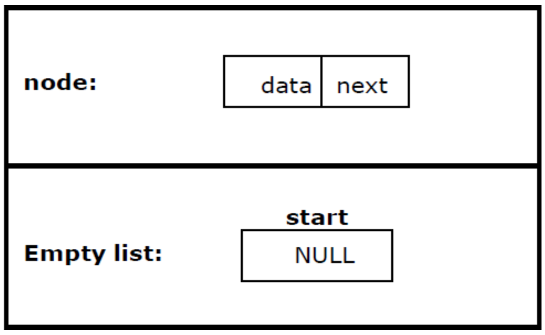
- Creating a structure with one data item and a next pointer, which will be pointing to next node of the list. This is called as self-referential structure.

- Initialise the start pointer to be NULL.

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

typedef struct slinklist node;

node *start = NULL;
```



The basic operations in a single linked list are:

- Creation.
- Insertion.
- Deletion.
- Traversing.

Conclusion:

Program: (Printout)

Output: (Printout)

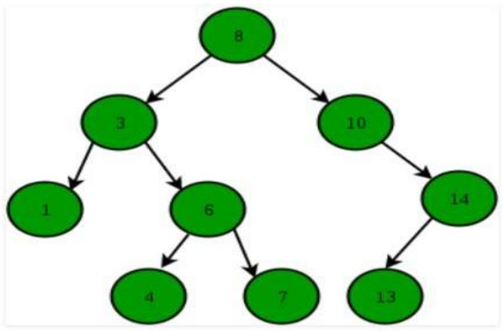
EXPERIMENT NO: 6

AIM: To Implement Binary Search Tree ADT using Linked List.

Theory:

Binary Search Tree is a node-based binary tree data structure which has the following properties:

- The left subtree of a node contains only nodes with keys lesser than the node's key.
- The right subtree of a node contains only nodes with keys greater than the node's key.
- The left and right subtree each must also be a binary search tree.
  There must be no duplicate nodes.



The above properties of Binary Search Tree provides an ordering among keys so that the operations like search, minimum and maximum can be done fast. If there is no ordering, then we may have to compare every key to search for a given key.

Illustration to insert 2 in below tree:
1. Start from the root.
2. Compare the inserting element with root, if less than root, then recurse for left, else recurse for right.
3. After reaching the end, just insert that node at left(if less than current) else right.

Time Complexity: The worst-case time complexity of search and insert operations is O(h) where h is the height of the Binary Search Tree. In the worst case, we may have to travel from root to the deepest leaf node. The height of a skewed tree may become n and the time complexity of search and insert operation may become O(n).

Conclusion:

Program: (Printout)

Output: (Printout)

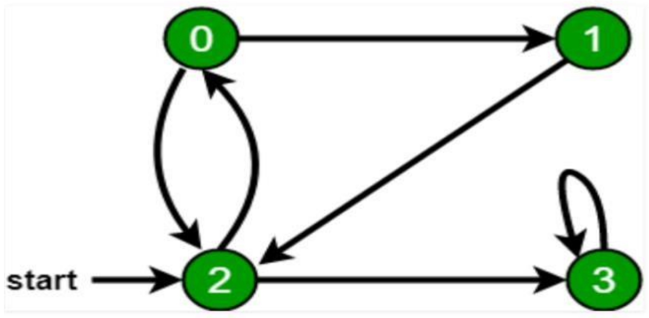
:

EXPERIMENT NO:7

AIM: To implement Graph Traversal techniques: a) Depth First Search b) Breadth First Search.

Theory:

Breadth First Search for a graph is similar to Breadth-First Traversal of a tree. The only catch here is, unlike trees, graphs may contain cycles, so we may come to the same node again. To avoid processing a node more than once, we use a boolean visited array. For simplicity, it is assumed that all vertices are reachable from the starting vertex.
For example, in the following graph, we start traversal from vertex 2. When we come to vertex 0, we look for all adjacent vertices of it. 2 is also an adjacent vertex of 0. If we don't mark visited vertices, then 2 will be processed again and it will become a non-terminating process. A Breadth-First Traversal of the following graph is 2, 0, 3, 1.

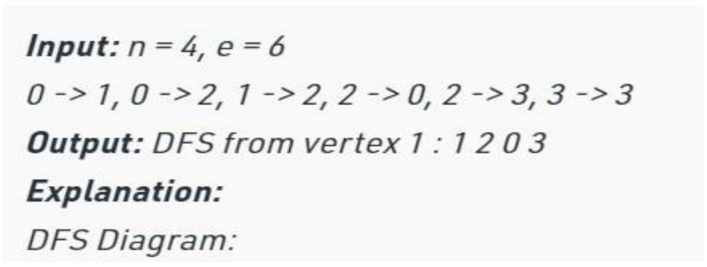


Following are the implementations of simple Breadth-First Traversal from a given source.
The implementation uses an adjacency list representation of graphs. STL's list container is used to store lists of adjacent nodes and the queue of nodes needed for BFS traversal.

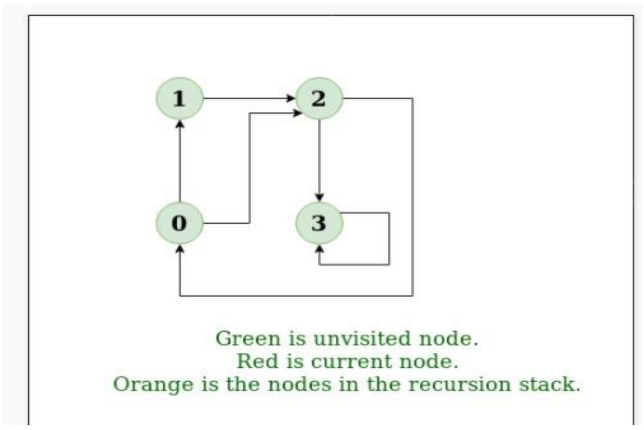
Theory:

Depth First Traversal for a graph is similar to Depth First Traversal of a tree. The only catch here is, unlike trees, graphs may contain cycles, a node may be visited twice. To avoid processing a node more than once, use a boolean visited array.

- **Approach:** Depth-first search is an algorithm for traversing or searching tree or graph data structures. The algorithm starts at the root node (selecting some arbitrary node as the root node in the case of a graph) and explores as far as possible along each branch before backtracking. So the basic idea is to start from the root or any arbitrary node and mark the node and move to the adjacent unmarked node and continue this loop until there is no unmarked adjacent node. Then backtrack and check for other unmarked nodes and traverse them. Finally, print the nodes in the path.
- **Algorithm:**
  1. Create a recursive function that takes the index of the node and a visited array.
  2. Mark the current node as visited and print the node.
  3. Traverse all the adjacent and unmarked nodes and call the recursive function with the index of the adjacent node.



Input: *n = 4, e = 6*
0 -> 1, 0 -> 2, 1 -> 2, 2 -> 0, 2 -> 3, 3 -> 3
Output: *DFS from vertex 1 : 1 2 0 3*
Explanation:
DFS Diagram:



Conclusion:

Program: (Printout)

Output: (Printout)