INTRODUCTION

This manual is organized in such a way that the students can directly use it in the laboratory. Each laboratory exercise comprises of

1. Statement of the problem
2. Algorithm
3. C code
4. Results
5. Conclusion

You must follow this sequence to conduct any data structures experiment in the computer laboratory and write your report accordingly. Using the algorithms given in this section one can implement in any language at a later time. For example, to implement the stack program in C++ or Java (of course now you must use C language) simply follow the same algorithmic steps and use additional language features, if any. This is the reason why the students must write the algorithm first and then the C code.

INTRODUCTION TO C

C was brought to us by Dennis Ritchie and Brian at Bell Labs. AT&T people had themselves this Operating System called UNIX that needed a programming language. They made it, and since the previous version was called B, they decided to call it C for obvious reasons. There was never an A programming language. The B in B stood for Bell.

C is a computer programming language. That means that you can use C to create lists of instructions for a computer to follow. C is one of thousands of programming languages currently in use. C has been around for several decades and has won widespread acceptance because it gives programmers maximum control and efficiency.

C is what is called a compiled language. This means that once you write your C program, you must run it through a C compiler to turn your program into an executable that the computer can run (execute). The C program is the human-readable form, while the executable
that comes out of the compiler is the machine-readable and executable form. What this means is that to write and run a C program, you must have access to a C compiler. If you are using a UNIX machine (for example, if you are writing CGI scripts in C on your host's UNIX computer, or if you are a student working on a lab's UNIX machine), the C compiler is available for free. It is called either "cc" or "gcc" and is available on the command line.

CHARACTERISTICS OF ‘C’ LANGUAGE

Modularity:
Ability to breakdown a large module into manageable sub modules called as modularity that is an important feature of structured programming languages.

Advantages:
1. Projects can be completed in time.
2. Debugging will be easier and faster.

Portability:
The ability to port i.e. to install the software in different platform is called portability.
Highest degree of portability: ‘C’ language offers highest degree of portability i.e., percentage of changes to be made to the sources code is at minimum when the software is to be loaded in another platform. Percentage of changes to the source code is minimum.

Extendability:
Ability to extend the existing software by adding new features is called as extendability.

SPEED:
‘C’ is also called as middle level language because programs written in ‘c’ language run at the speeds matching to that of the same programs written in assembly language so ‘c’ language has both the merits of high level and middle level language and because if this feature it is mainly used in developing system software.
Areas of Application

The C programming language is used in many different areas of application, but the most prolific area is UNIX operating system applications. The C language is also used in computer games:

- UNIX operating system
- computer games

DATA STRUCTURES

A data structure is an arrangement of data in a computer's memory or even disk storage. An example of several common data structures are arrays, linked lists, queues, stacks, binary trees, and hash tables. Algorithms, on the other hand, are used to manipulate the data contained in these data structures as in searching and sorting.

C provides us with just one built-in data structure, the array. Although efficient and easily used and understood, arrays often don't provide us with the level of functionality we need from a data structure. If the quantity of data we need to store is not well known at compile time, then using arrays could waste memory if too large, or waste time in resizing at runtime if too small. Several, more abstract, data structures can be constructed to better serve our needs.

Many algorithms apply directly to a specific data structures. When working with certain data structures you need to know how to insert new data, search for a specified item, and deleting a specific item.

Commonly used algorithms include are useful for:

- Searching for a particular data item (or record).
- Sorting the data. There are many ways to sort data. Simple sorting, Advanced sorting
- Iterating through all the items in a data structure. (Visiting each item in turn so as to display it or perform some other action on these items)
Some data structures with the comparisons made on them are shown below.

<table>
<thead>
<tr>
<th>Data Structure</th>
<th>Advantages</th>
<th>Disadvantages</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Array</strong></td>
<td>Quick inserts</td>
<td>Slow search</td>
</tr>
<tr>
<td></td>
<td>Fast access if index known</td>
<td>Slow deletes</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Fixed size</td>
</tr>
<tr>
<td><strong>Ordered Array</strong></td>
<td>Faster search than unsorted array</td>
<td>Slow inserts</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Slow deletes</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Fixed size</td>
</tr>
<tr>
<td><strong>Stack</strong></td>
<td>Last-in, first-out access</td>
<td>Slow access to other items</td>
</tr>
<tr>
<td><strong>Queue</strong></td>
<td>First-in, first-out access</td>
<td>Slow access to other items</td>
</tr>
<tr>
<td><strong>Linked List</strong></td>
<td>Quick inserts, Quick deletes</td>
<td>Slow search</td>
</tr>
<tr>
<td><strong>Binary Tree</strong></td>
<td>Quick search, Quick inserts, Quick deletes</td>
<td>Deletion algorithm is complex</td>
</tr>
<tr>
<td></td>
<td><em>(If the tree remains balanced)</em></td>
<td></td>
</tr>
<tr>
<td><strong>Red-Black Tree</strong></td>
<td>Quick search, Quick inserts, Quick deletes</td>
<td>Complex to implement</td>
</tr>
<tr>
<td></td>
<td><em>(Tree always remains balanced)</em></td>
<td></td>
</tr>
<tr>
<td><strong>2-3-4 Tree</strong></td>
<td>Quick search, Quick inserts, Quick deletes</td>
<td>Complex to implement</td>
</tr>
<tr>
<td></td>
<td><em>(Tree always remains balanced)</em></td>
<td></td>
</tr>
<tr>
<td></td>
<td><em>(Similar trees good for disk storage)</em></td>
<td></td>
</tr>
<tr>
<td><strong>Hash Table</strong></td>
<td>Very fast access if key is known, Quick inserts</td>
<td>Slow deletes</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Access slow if key is not known</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Inefficient memory usage</td>
</tr>
<tr>
<td><strong>Heap</strong></td>
<td>Quick inserts, Quick deletes, Access to largest item</td>
<td>Slow access to other items</td>
</tr>
<tr>
<td><strong>Graph</strong></td>
<td>Best models real-world situations</td>
<td>Some algorithms are slow and very complex</td>
</tr>
</tbody>
</table>

**NOTE:** The data structures shown above (with the exception of the array) can be thought of as Abstract Data Types (ADTs).
- Operating System - Windows XP/Vists/7
- Software - TurboC/ C++

**List of Experiments**

1. Simple C Programs
   a. Fibonacci Series
   b. Factorial of a number by recursion.
   c. Sorting of a set of numbers using arrays.

2. Advanced C Programs
   a. Implementation of basic arithmetic operations using switch – case statements and functions.
   b. Creating an employee database using structure.
   c. Swapping of two numbers using call by reference.

3. Array Implementation of List ADT
4. Linked List Implementation of List ADT
5. Cursor Implementation of List ADT.
6. Array implementation of stack ADT.
7. Linked list implementation of stack ADT
8. Checking ‘Balanced Parentheses’ using
   a. Array implementation of stack
   b. Linked list implementation of stack.

9. Evaluation of Postfix expressions using
   a. Array implementation of stack
   b. Linked list implementation of stack.

10. Array implementation of Queue ADT.
11. Linked List implementation of Queue ADT.
12. Binary Search Trees – implementation
13. Heap Sort
14. Quick Sort
Ex. No 1  Simple C programs

Aim:

To write Simple C programs using arrays and control constructs.

List of Exercises:

a) To generate Fibonacci series.
b) To find the factorial of a given number using recursion.
c) To sort a set of numbers using arrays.

Recursive Functions:

Functions are used to encapsulate a set of operations and return information to the main program or calling routine. The use of functions provides several benefits. First, it makes programs significantly easier to understand and maintain. The main program can consist of a series of function calls rather than countless lines of code. A second benefit is that well written functions may be reused in multiple programs. The C standard library is an example of the reuse of functions. A third benefit of using functions is that different programmers working on one large project can divide the workload by writing different functions. A recursive function is a function that calls itself, thus understanding recursion to understand recursion seems like a call to itself.

Arrays:

Arrays are a type of data structures that is used to store a group of objects of the same type sequentially in memory. All the elements of an array must be the same data type, for example float, char, int, pointer to float, pointer to int, a structure or function. The elements of an array are stored sequentially in memory. This allows convenient and powerful manipulation of array elements using pointers. An array is defined with this syntax.
**Algorithm:**

a) To find Fibonacci Series:
   i. Get the input number n, i.e. the number of terms to be generated in the series.
   ii. Print the first two numbers of the series as 0 and 1, assign these two values to variables a and b.
   iii. Compute the next element in the series as the sum of the previous two elements.
   iv. Keep on sliding the variables a and b to b and c to compute the series.

b) To find the factorial of a given number using recursion:
   i. Get the number N
   ii. call the function for N
   iii. Inside the function compute fact = N * fact (N-1)
   iv. Step ((3) is repeated till N becomes 1.
   v. Return the fact value to the main function and display the result.

c) To sort a set of numbers using arrays:
   i. Initialize an array of required size.
   ii. Get the input set of numbers and store them in the array.
   iii. Take each number of an array and compare that number with the subsequent numbers in the array.
   iv. If the former is greater than the latter, swap the two numbers.
   v. Repeat the procedure for all the numbers in the array.
   vi. At the end of the procedure, the array would be sorted and is printed.
Ex. No 2  
Advanced C programs

Aim:
To write Advanced C programs using pointers & structures

List of Exercises:
  a) Implementation of basic arithmetic operations using switch – case statements and functions.
  b) Creating an employee database using structure.
  c) Swapping of two numbers using call by reference.

Functions:
Function is a portion of code within a larger program, which performs a specific task and can be relatively independent of the remaining code. The syntax of many programming languages includes support for creating self contained subroutines, and for calling and returning from them.

Structures:
A structure provides a means of grouping variables under a single name for easier handling and identification. A structure is declared by using the keyword struct followed by an optional structure tag followed by the body of the structure. The variables or members of the structure are declared within the body.

Pointers:
Pointers are variables that hold addresses in C. A pointer is a special type of variable that contains a memory address rather than a data value. Just as data is modified when a normal variable is used, the value of the address stored in a pointer is modified as a pointer variable is manipulated. A pointer to the variable is passed to the function. The pointer can then be manipulated to change the value of the variable in the calling routine. It is interesting to note that the pointer itself is passed by value. The function cannot change the pointer itself since it gets a local copy of the pointer. However, the function can change the contents of memory, the variable, to which the pointer refers. The advantages of passing by pointer are that any changes to variables will be passed back to the calling routine and that multiple variables can be changed.
Algorithm:

a) Implementation of basic arithmetic operations using switch – case statements and functions.
   i. Get two numbers and the arithmetic operation to be performed on them as input.
   ii. Use a switch case statement
   iii. According to the selected option the desired function is called from inside a switch case function.
   iv. The result is printed.
   v. The steps are repeated until the user wishes to quit.

b) Creating an employee database using structure.
   i. Create a structure using the struct variable for each employee containing the necessary attributes of that employee.
   ii. Create an array of such structures for n number of employees.
   iii. Get the input of each attribute for each employee from the user.
   iv. Calculate the gross salary by the following formulas.
       hra (10 %), da(50%), ta(5%), pf(11%), tax(6%) from basic.
       Gross = basic + hra + da + ta
       Deduction = pf + tax
       Netsalary = gross – deduction
   v. Display the results.

c) Swapping by call by reference
   i. Get the numbers two be swapped
   ii. Pass the elements using their address to the swap function
   iii. Receive the parameters for the swap function using pointers
   iv. Perform swapping inside function
   v. Display the result in the main function
Ex. No 3  

Array Implementation of List ADT

Aim:
To implement a List ADT using arrays.

List ADT:
A list is a set of ordered or unordered elements. An example of a list is $A_1, A_2, A_3… A_n$. For each element $A_i$ in the list there is a previous element $A_{i-1}$, called the predecessor, and an element that follows it, $A_{i+1}$, called the successor. The size of the list is $n$. The first element has no predecessor and the last element has no successor. This list can be implemented using arrays.

We have to declare an array ahead of time. Then a set of elements can be assigned to the array positions. To insert an element $x$, at a position $p$ in an array, first, the array size has to be increased by one and then all the elements after the position $p$ have to be shifted one place downward, and then the element can be inserted at the required position.

| 10 | 15 | 24 | 8 |

Fig 1. Before insertion. Size of list: $n=4$

| 10 | 15 | 45 | 24 | 8 |

Fig.2 After inserting 45 at the third position. Now size $n=5$

To delete an element from a position $p$, we have to shift all the elements after $p$, one position backward and then decrement the size of the array by 1.

| 10 | 15 | 45 | 24 | 8 |

Fig 3. Before deletion. Size of list: $n=5$
Algorithm:  Fig 4. After deletion at position 3. Size of list: n= 5

1. Declare an array ahead of time.
2. Get the input values of n elements and store them in the array.
3. To insert an element at a position p, increase the array size by one, shift all the elements after p one step forward and then assign the new element at the required position.
4. To delete an element at a position p, shift all the elements after that position by one and then decrease the size of the array by 1.
5. To find the position of an element inside an array, search the array elements one by one from the start to the size of the array until the required element is found and return the position of the element. If the element is not found return 0.
Ex. No 4  \hspace{1cm} \textbf{Linked List Implementation of List ADT}

\textbf{Aim:}

To create a linked list of elements using pointer concepts.

\textbf{Linked List:}

A singly-linked list is simply a sequence of dynamically allocated items, each of which refers to its successor in the list. Each item in the list is called a node and contains two fields, an information field and a next address field. The information field holds the actual element on the list and next address field contains the address of the next node in the list. Hence the next address field has to be a pointer of type similar to the elements in the list. The entire list is pointed by an external pointer called ‘header’ (L) which points to the first node in the list. When ever header is pointing to NULL, the list is empty. The next address field of the last node in the list will be pointing to NULL.

\begin{center}
\begin{tabular}{|c|c|}
\hline
100 & 15 \hspace{1cm} 800 \hspace{1cm} 25 \hspace{1cm} 2000 \hspace{1cm} 30 \hspace{1cm} \text{Null} \\
\hline
\end{tabular}
\end{center}

\begin{center}
L = 500 \hspace{1cm} 100 \hspace{1cm} 800 \hspace{1cm} 2000
\end{center}

During insertion a new node is created and the position at which the element is to be inserted is assigned as P. Then the node p is made to point to the new node and the new node points to the node that P was pointing to initially before insertion. The following figure shows the changes that are made during the insertion of the element 10 in the existing list after the position 100.

\begin{center}
\begin{tabular}{|c|c|}
\hline
100 & 15 \hspace{1cm} 400 \hspace{1cm} 25 \hspace{1cm} 2000 \hspace{1cm} 30 \hspace{1cm} \text{Null} \\
\hline
\end{tabular}
\end{center}

\begin{center}
L = 500 \hspace{1cm} P = 100 \hspace{1cm} 800 \hspace{1cm} 2000
\end{center}

New node at 400
Algorithm:

1. Declare and initialize the necessary variables.
2. Create a node structure with two fields, one - an information field, and the other, a pointer to a structure of a similar type of node.
3. Using malloc function create a structure and store the address of that in the variable L as the dummy or header node value.
4. To create a list of elements in the linked list, get the number of elements as input. For each element create a new node using an malloc function. Let the node be P.
   a. The first created node at the address P, should be assigned to L->next. Then the value of the address at P should be assigned to a temporary cell called TempCell.
   b. After that the subsequent nodes are created. Each node called P is assigned to TempCell->next and P->next is assigned as NULL. After that the value of P is assigned to the variable TempCell.
5. To insert an element X, after the position P in the above list, we first create a new node using malloc function and assign the returned memory address to the pointer TempCell. Then,
   a. Assign TempCell->element as X.
   b. Make TempCell->next point to P.
   c. Make P->next point to TempCell.
6. To delete an element X in the linked list,
   a. Find the node previous to the element X in the list in the following manner. Initialize the value of a node pointer, say P to the header value L. Keep on incrementing the value of P to be P->next until P->next->element is not equal to the element X. Once P->next->element is equal to X, then P is the previous node to X.
   b. Store P->next in a temporary variable TempCell. Now the TempCell contains the pointer to the element X to be deleted.
   c. Make P->next as TempCell->next.
   d. Free the memory of TempCell.
7. To find the position of an element X in the list,
   a. Initialize the value of any node pointer variable, P as L->next, i.e. the first
element in the list.
   b. Keep on incrementing the value of P as P=P->next until P->element is not equal
to the element X that is being searched. If P->element is equal to the element
being searched or if we reach the end of the list, i.e. P=Null, we stop and return
the value of P.
Ex .No 5  Cursor Implementation of List ADT

Aim:

To implement the List ADT using cursors.

Cursor Implementation:

Some languages like BASIC and FORTRAN do not support pointers. In such cases we go for cursor implementation of lists. Here, we define a global array of structures. Each array has an index number. The malloc and free functions are implemented by using a Cursor Free Space list, which gives the indexes of the structures that are free to use at the moment. The header index value for the Cursor Free Space is 0. For example, if there are 10 global structures that are defined then the Cursor Free Space would be as follows.

<table>
<thead>
<tr>
<th>Index</th>
<th>Element</th>
<th>Next Free Space</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>-</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>-</td>
<td>2</td>
</tr>
<tr>
<td>2</td>
<td>-</td>
<td>3</td>
</tr>
<tr>
<td>3</td>
<td>-</td>
<td>4</td>
</tr>
<tr>
<td>4</td>
<td>-</td>
<td>5</td>
</tr>
<tr>
<td>5</td>
<td>-</td>
<td>6</td>
</tr>
<tr>
<td>6</td>
<td>-</td>
<td>7</td>
</tr>
<tr>
<td>7</td>
<td>-</td>
<td>8</td>
</tr>
<tr>
<td>8</td>
<td>-</td>
<td>9</td>
</tr>
<tr>
<td>9</td>
<td>-</td>
<td>10</td>
</tr>
<tr>
<td>10</td>
<td>-</td>
<td>Null</td>
</tr>
</tbody>
</table>

To insert an element, first take the first free structure that the header is pointing to, and change the header pointer to the next free space of the structure that has been taken. For e.g. In
the above case to insert an element, first take the structure 1 and change the next free space of the header pointer to the value of 2.

To delete an element simply return the structure index to the Cursor Free Space and make the header pointer point to that index. Make the next free space of the returned structure to point to the next free space that the header was pointing to before the return of the structure to the Cursor Free Space.

**Algorithm:**

1. Declare a structure with the two fields, `Element` – to store the value of the elements and `Next` – that denotes the index where the next element resides. Also initialize the necessary variables.
2. Create an array of such structures (say n in number).
3. Initialize the `CursorSpace[]` array by making the header pointer point to the first array index, (i.e. `CursorSpace[0].Next=1`) and that one to point to the next array index (i.e. `CursorSpace[1].Next=2`) and subsequently until the last index is reached. The next free space of the last array index is made to point to 0 (i.e. `CursorSpace[n].Next=Null`)
4. To insert an element in the list, use a function called `CursorAlloc()` that is equivalent to `malloc` function. In `CursorAlloc()`, get the array index that is pointed to by the header index 0 and store it in a value P. Now make the header’s next free space to be the next free space of the index P by the command `CursorSpace[0].Next = CursorSpace[P].Next`. Now assign the new element to `CursorSpace[P].Element`.
5. To delete an element from the List ADT, take the position P of that element. Now return P to the `CursorSpace` array and make `CursorSpace[P].Next` to point to `CursorSpace[0].Next`. Also make the header pointer point to the position P by the command `CursorSpace[0].Next=P`.
6. To find the position of an element in a List ADT, get the element; initialize an index P as the index of the array pointed to by the header. Now keep on incrementing the value of P by the command P=`CursorSpace[P].Next`, till the element is found in `Cursor[P].Element`. 
Ex .No 6   Array Implementation of Stack ADT

Aim:
To implement a stack using arrays.

Stack:
A stack is a list in which all insertions and deletions are made at one end, called the top. It is a collection of contiguous cells, stacked on top of each other. The last element to be inserted into the stack will be the first to be removed. Thus stacks are sometimes referred to as Last In First Out (LIFO) lists.

The operations that can be performed on a stack are push, pop and top. Push is to insert an element at the top of the stack. Pop is deleting an element that is at the top most position in the stack. Top simple examines and returns the top most value in the stack without deleting it. Push on an already filled stack and pop on an empty stack results in serious errors. Before any insertion, the value of the variable top is initialized to -1.
Algorithm:

1. Declare an array ahead of time called `Array`.
2. Declare a structure called `Stack` that contains the `TopOfStack` and the `Capacity` fields.
3. The variable called `TopOfStack` for the stack is initialized to -1.
4. To push an element into the stack, increment `TopOfStack` and then set
   \[ Array[TopOfStack] = X. \]
5. To pop an element from the array, set the return value to `Array[TopOfStack]` and then decrement `TopOfStack`.  

Ex. No 7  Linked List Implementation of Stack ADT

Aim:

To implement a stack using a linked list.

Linked List Implementation:

We go for a linked list implementation of a stack rather than an array implementation because of the run time memory allocation feature of linked lists. In this implementation we make a head pointer and make it point to the first created node in the stack. Whenever a new node is pushed into the stack we make the head pointer point to the new node and the new node to point to the already existing node that was pointed by the head.
pointer

Creation of head pointer initially:

```
+----+        +-----+
| FN |        | Null |
+----+        +-----+
     Head Pointer
```

Pushing an element:

```
+----+        +-----+
| FN |        | Null |
+----+        +-----+
     Head Pointer
     First Node (FN)
```

Pushing a second element:

```
+----+        +----+       +-----+
| SN |        | FN |       | Null |
+----+        +----+       +-----+
     Head Pointer
     Second Node (SN)
     First Node
```

Popping an element:

```
+----+        +-----+
| FN |        | Null |
+----+        +-----+
     Head Pointer
     First Node (FN)
```

The above figure shows the popping of an element from the stack. The element that was inserted last would be popped out first. So, the head pointer is now made to point to the First Node, which was pointed by the Second Node and the Second Node is freed.

**Algorithm:**
1. Create a structure with an element (Element) and a pointer (Next) that points to the next node in the list.
2. Create a head pointer using malloc function and assign the resultant pointer variable to the head pointer S.
3. In the Push operation, create a new node, called TmpCell using the malloc function, assign the new element X to TmpCell->Element, make TmpCell point to the node that was previously pointed to by the head pointer and make the head pointer point to TmpCell by the statements: TmpCell->Next=S->Next and S->Next=TmpCell.
4. In the Pop operation, store the node pointed to by the head pointer in a pointer called FirstCell by the statement FirstCell=S->Next. Then make the head pointer point to the node next to FirstCell by the statement S->Next=S->Next->Next. Then free FirstCell.
5. In the Top operation, return the element pointed to by the head pointers next node by the statement S->Next->Element.

Ex. No. 8.a

Checking ‘Balanced Parentheses’ using

Array implementation of stack

Aim:
To check whether an expression has balanced parentheses using array implementation of a stack.

Theory:
Compilers check your programs for syntax errors, but frequently a lack of one symbol will cause the compiler to spill out a hundred lines of diagnostics without identifying the real error. Thus, every right brace, bracket and parentheses must correspond to its left counterpart. This can be verified using a stack.

Algorithm:
1. Make an empty stack implemented as an array.
2. Scan the expression from left to right, character by character.
3. During your scanning:
   a. If you find a left parentheses push it into the stack.
   b. If you find a right parentheses examine the status of the stack:
      i. If the stack is empty then the right parentheses does not have a matching
         left parentheses. So stop scanning and print expression is invalid.
      ii. If the stack is not empty, pop the stack and continue scanning.
4. When the end of the expression is reached, the stack must be empty. Otherwise one or
   more left parentheses has been opened and not closed.

Ex. No. 8.b  Checking ‘Balanced Parentheses’ using
Linked List implementation of stack

Aim:
To check whether an expression has balanced parentheses using linked list
implementation of a stack.

Algorithm:
1. Make an empty stack implemented as a linked list.
2. Scan the expression from left to right, character by character.
3. During your scanning:
   a. If you find a left parentheses push it into the stack.
   b. If you find a right parentheses examine the status of the stack:
i. If the stack is empty then the right parentheses does not have a matching left parentheses. So stop scanning and print expression is invalid.

ii. If the stack is not empty, pop the stack and continue scanning.

4. When the end of the expression is reached, the stack must be empty. Otherwise one or more left parentheses has been opened and not closed.

Ex. No. 9.a Evaluation of postfix expression using Array implementation of stack

Aim:

To evaluate a postfix expression using array implementation of a stack.

Theory:

Postfix notation is a way of writing algebraic expressions without the use of parentheses or rules of operator precedence. The expression \((A+B)/(C-D)\) would be written as \(AB+CD-/\) in postfix notation. Evaluating an expression in postfix notation is trivially easy if you use a stack. The postfix expression to be evaluated is scanned from left to right. Variables or constants are pushed onto the stack. When an operator is encountered, the indicated action is performed using the top elements of the stack, and the result replaces the operands on the stack.
Algorithm:
1. Make an empty stack implemented as an array.
2. Get an input postfix expression from the user.
3. Scan the expression from left to right.
4. During your scanning:
   a. If you encounter an operand, push it on to the stack and continue scanning.
   b. If you encounter an operator, pop only the topmost two elements from the stack, apply the operator on the elements and push the result back to the stack.
5. When you reach the end of the string, there should be only one element on to the stack. Pop this value to get the result of your postfix expression.

Ex. No. 9.b Evaluation of postfix expression using Linked List implementation of stack

Aim:
To evaluate a postfix expression using linked list implementation of a stack.

Algorithm:
1. Make an empty stack implemented as a linked list.
2. Get an input postfix expression from the user.
3. Scan the expression from left to right.
4. During your scanning:
   a. If you encounter an operand, push it on to the stack and continue scanning.
b. If you encounter an operator, pop only the topmost two elements from the stack, apply the operator on the elements and push the result back to the stack.

5. When you reach the end of the string, there should be only one element on the stack. Pop this value to get the result of your postfix expression.

Ex. No. 10  Array Implementation of Queue ADT

Aim:
To implement a Queue ADT using an array.

Queue:
A Queue is an ordered collection of items from which items may be deleted at one end (called the front of the queue) and into which items may be inserted at the other end (the rear of the queue). Queues remember things in first-in-first-out (FIFO) order. The basic operations in a queue are: Enqueue - Adds an item to the end of queue. Dequeue - Removes an item from the front.
Array implementation of Queue ADT:

A queue is implemented using a one dimensional array. FRONT is an integer value, which contains the array index of the front element of the array. REAR is an integer value, which contains the array index of the rear element of the array. When an element is deleted from the queue, the value of HEAD is increased by one. i.e. \( \text{HEAD} = \text{HEAD} + 1 \) When an element is inserted into the queue, the value of TAIL is increased by one. i.e. \( \text{TAIL} = \text{TAIL} + 1 \)

Algorithm:

1. Initialize an array of size \( n \), and the two variables associated with queue, front and rear to -1.
2. To insert an element into the queue:
   a. First check whether the queue is full by the condition \( \text{rear} = \text{max} - 1 \); if it is, then print an error message and exit.
   b. Check whether the queue is initially empty by the condition \( \text{front} = -1 \) or \( \text{front} > \text{rear} \); if it is, then increment front by 1.
   c. Increment rear by 1 and insert the element in the rear position.
3. To delete an element from the queue:
a. First check whether the queue is empty by the condition front = -1 or front > rear; if it is, then print an error message and exit.
b. Increment front by 1 to delete the first inserted element.

4. Display the queue contents by printing the elements from front to rear.

Ex. No. 11 Linked List Implementation of Queue ADT

Aim:
To implement a Queue ADT using a linked list.

Linked List Implementation of a Queue:
We go for a linked list implementation of a queue rather than an array implementation because of the run time memory allocation feature of linked lists. In this implementation we make a head pointer and make it point to the first created node in the stack. Whenever an element is enqueued into the queue, a new node is created at the end of the linked list and the
element is kept in that node. Whenever a dequeue operation is performed on the queue, the head pointer is made to point to the node, next to the one that it is currently pointing to.

**Insertion:**

![Insertion Diagram](image)

**Deletion:**

**Before Deletion:**

![Before Deletion Diagram](image)

**After Deletion:**

![After Deletion Diagram](image)

**Algorithm:**

1. Create a structure with an element (Element) and a pointer (Next) that points to the next node in the list.
2. Create a head pointer using malloc function and assign the resultant pointer variable to the head pointer Q.
3. During the Enqueue operation of a new element X:
   a. Create a new node called TmpCell using the malloc function. Assign the element to that node by the statement: TmpCell - > Element = X and assign TmpCell->Next to NULL.
b. If, at the time of insertion, Q->Next is NULL (i.e. the queue is initially empty) assign Q->Next to TmpCell.

c. Otherwise, slide a pointer to the node, P, to the end of the queue by the statement P=P->Next until P->Next!=NULL and assign P->Next= TmpCell.

4. During the Dequeue operation:
   a. Store the value of Q->Next in a temporary pointer FirstCell.
   b. Make Q->Next = Q->Next->Next.
   c. Free the memory contents of FirstCell.

Ex. No. 12 Implementation of Binary Search Trees

Aim:
To implement binary search trees using linked lists.

Binary Search Trees:
A binary tree is a tree in which no node can have more than two children. In a binary search tree, for every node, X, in the tree, the values of all the keys in its left sub tree are smaller than the key value of X, and the values of all the keys in its right sub tree are larger than the key
value of X. The basic operations on a binary search tree take time proportional to the height of the tree.

In the linked list implementation of binary search trees: Each element is represented by a node with two link fields and a data field. Each connecting line (or edge) in a binary tree drawing will be represented by a link field. A leaf node has a leftChild and rightChild link of NULL. Root node will be pointed to by a pointer variable.

**Algorithm:**

1. Create a structure with an element (Element) and two pointers – Left and Right that points to the left and right child node respectively in the tree.
2. Create a new node using malloc function and assign the resultant pointer variable to the root node pointer T and assign it to NULL.
3. To insert a new element X into the tree:
   a. If the value of T is NULL, assign T->Element to X, and the left and right child pointers to NULL and exit the insertion operation.
b. Otherwise if the element to be inserted is less than the root element T, repeat the step 3 recursively, with the new value of T as T->Left.

c. Otherwise if the element to be inserted is more than the root element T, repeat the step 3 recursively, with the new value of T as T->Right.

d. If the element is already present in the tree, do nothing.

4. To delete an element X from the tree:
   a. Find the node where the element resides.
   b. If the node has no left and right children, then the pointer to that node from the parent is changed to NULL and the node is freed of its memory.
   c. If the node has only one child, then the parent of the node is made to point to the child of the node and the node is freed.
   d. If the node has both left and right children:
      i. Look at the right subtree of the node (subtree rooted at the right child of the node).
      ii. Find the Minimum there.
      iii. Replace the key of the node to be deleted by the minimum element.
      iv. Delete the minimum element.

5. To find an element X in the tree with root node T:
   a. If the root node T is initially NULL, then the tree is empty. So return NULL and exit.
   b. Take the element X and compare it with the root node. If X is less than the element found at the root node, then repeat step 5 recursively with the new value of T as T->Left.
   c. Take the element X and compare it with the root node. If X is more than the element found at the root node, then repeat step 5 recursively with the new value of T as T->Right.

6. To find the minimum element in a tree with root node T:
   a. If T is NULL return NULL.
   b. Otherwise slide the value of T to T->Left until T->Left becomes NULL.
   c. Return the value of T.

7. To find the maximum element in a tree with root node T:
a. If T is NULL return NULL.
b. Otherwise slide the value of T to T->Right until T->Right becomes NULL.
c. Return the value of T.

Ex. No. 13

Heap Sort

Aim:
To sort a series of numbers using the heap sort method.

Heap Sort:
A sorting algorithm that works by first organizing the data to be sorted into a special type of binary tree called a heap. The heap itself has, by definition, the largest value at the top of the tree, so the heap sort algorithm must also reverse the order. It does this with the following steps:
1. Remove the topmost item (the largest) and replace it with the rightmost leaf. The topmost item is stored in an array.
2. Re-establish the heap.
3. Repeat steps 1 and 2 until there are no more items left in the heap.

The heap sort works as it name suggests - it begins by building a heap out of the data set, and then removing the largest item and placing it at the end of the sorted array. After removing the largest item, it reconstructs the heap and removes the largest remaining item and places it in the next open position from the end of the sorted array. This is repeated until there are no items left in the heap and the sorted array is full. Elementary implementations require two arrays - one to hold the heap and the other to hold the sorted elements.

**Algorithm:**

1. Build a heap out of the data set.
2. Remove the largest item and place it at the end of the sorted array.
3. After removing the largest item, reconstruct the heap and remove the largest remaining item and places it in the next open position from the end of the sorted array.
4. Step (3) is repeated until there are no items left in the heap and the sorted array is full.
5. Implementations require two arrays - one to hold the heap and the other to hold the sorted elements.

---

**Ex. No. 14 Quick Sort**

**Aim:**

To sort a series of numbers using the quick sort method.

**Quick Sort:**

The quick sort is an in-place, divide-and-conquer, massively recursive sort. As a normal person would say, it's essentially a faster in-place version of the merge sort. The quick sort algorithm is simple in theory, but very difficult to put into code.

The recursive algorithm consists of four steps:
1. If there are one or less elements in the array to be sorted, return immediately.
2. Pick an element in the array to serve as a "pivot" point. (Usually the left-most element in
   the array is used.)
3. Split the array into two parts - one with elements larger than the pivot and the other with
   elements smaller than the pivot.
4. Recursively repeat the algorithm for both halves of the original array.

The quick sort is by far the fastest of the common sorting algorithms.

Algorithm:
1. Get N elements which are to be sorted, and store it in the array A.
2. Select the element from A[0] to A[N-1] for middle. This element is the pivot.
3. Partition the remaining elements into the segments left and right so that no elements in
   left has a key larger than that of the pivot and no elements in right has a key smaller than
   that of the pivot.
4. Sort left using quick sort recursively.
5. Sort right using quick sort recursively.
6. Display the sorted array A.

Contemporary Experiments (Beyond Syllabus)

1. Towers of Hanoi
2. Implementation of Doubly Linked List
3. Implementation of Merge sort, Insertion Sort, Shell Sort
4. Implementation of DFS Algorithm.