Chapter-4

DATA STRUCTURES

➢ Introduction:
- *Data Structure is the way of collecting and organizing the data in such a way that we can perform operation on these data in an effective way.*
- For Example: We have the data student’s Name “Akash” and age “16”. Here “Akash” is a string type data type and 16 are of integer. We can now organize this data as a record like a Student Record. We can collect and store other Student’s Record in a file or a database as a data structure.

➢ Data Representation:
- Computer memory is used to store the data which is required for processing. This process is known as data representation.
- Data may be of same type or different type.
- Data Structure provides efficient way of combining these data types and process data.

➢ Primitive Data structures:
- *Data structures that are directly operated upon the machine-level instructions are known as primitive data structures.*
- The integers, float, character data, pointers are primitive data structures.

➢ Operations on Primitive Data structures:
- Create: Create operation is used to create a new data structure. This operation reserves memory space for the program elements. It may be carried out at compile time and run time.
Example: int n=15; // memory spaced to be created for n.

- **Select**: This operation is used programmers to access the data within the data structure. This operation updates or alters data.
  
  Example: cin>>x;

- **Update**: This operation is used to change data of data structures. An assignment operation is a good example.
  
  Example: int n = 5; //modifies the value of n to store the new value 5 in it.

- **Destroy**: This operation is used to destroy or remove the data structure from the memory space. In C++ one can destroy data structure by using the function called delete.

➢ **Non-Primitive Data structures**:

- *The Data structures that are derived from the primitive data structures are called Non-primitive data structure.*

- These data structures are used to store group of values.

- Non-Primitive data structures are classified as arrays, lists and files.

- Data Structures under lists are classified as linear and non-linear data structure.

  i. **Linear Data structures**:

  - *Linear Data structures are kind of data structure that has homogeneous elements.*
  
  - The data structure in which elements are in a sequence and form a liner series.
  
  - Linear data structures are very easy to implement, since the memory of the computer is also organized in a linear fashion.

  - Some commonly used linear data structures are **Stack, Queue and Linked Lists**.

  ii. **Non-Linear Data structures**:

  - *A Non-Linear Data structures is a data structure in which data item is connected to several other data items.*

  - Non-Linear data structure may exhibit either a hierarchical relationship or parent child relationship.

  - The data elements are not arranged in a sequential structure.

  - The different non-linear data structures are **trees and graphs**.

➢ **Operations on Non-Primitive Data structures**:

1. **Traversing**: The processing of accessing each element exactly once to perform some operation is called **traversing**.

2. **Insertion**: The process of adding a new element into the given collection of data elements is called **insertion**.
3. **Deletion**: The process of removing an existing data element from the given collection of data elements is called *deletion*.

4. **Searching**: The process of finding the location of a data element in the given collection of data elements is called as *searching*.

5. **Sorting**: The process of arrangement of data elements in ascending or descending order is called *sorting*.

6. **Merging**: The process of combining the elements of two structures to form a single structure is called *merging*.

**Arrays:**

- *An array is an ordered collection of elements of same data type that share common name.*
- The elements are arranged one after the other in adjacent memory location.
- These elements are accessed by numbers called subscripts or indices.
- The elements are accessed using subscripts; arrays are also called as subscripted variables.
- There are three types of arrays
  i. One-dimensional Array
  ii. Two-dimensional Array
  iii. Multi-dimensional Array

**One dimensional array:**

- *An array with only one row or column is called one-dimensional array.*
- It is finite collection of n number of elements of same type such that:
  o Elements are stored in continuous locations.
  o Elements can be referred by indexing.
- The syntax to define one-dimensional array is:
  
<table>
<thead>
<tr>
<th>Syntax:</th>
<th>Datatype Array_Name [Size];</th>
</tr>
</thead>
<tbody>
<tr>
<td>Where,</td>
<td>Datatype : Type of value it can store (Example: int, char, float)</td>
</tr>
<tr>
<td></td>
<td>Array_Name: To identify the array.</td>
</tr>
<tr>
<td></td>
<td>Size : The maximum number of elements that the array can hold.</td>
</tr>
</tbody>
</table>
- Example: int a[5];

<table>
<thead>
<tr>
<th>Address</th>
<th>Content</th>
<th>Location</th>
</tr>
</thead>
<tbody>
<tr>
<td>1000</td>
<td>A</td>
<td>A[0]</td>
</tr>
<tr>
<td>1001</td>
<td>B</td>
<td>A[1]</td>
</tr>
</tbody>
</table>
• Calculating the length of the array:
  
  o A[n], the number of n is called the size or length of the array.
  o The length of the array can be calculated by:

\[ L = UB - LB + 1 \]

  o Here, UB is the largest Index and LB is the smallest index.
  o Example: If an array A has values 10, 20, 30, 40, 50, stored in location 0, 1, 2, 3, 4 the UB = 4 and LB = 0
  o Size of the array L = 4 – 0 + 1 = 5

➤ Basic operations on One-dimensional array:

1. **Traversing**: Processing each element in the array.
2. **Insertion**: Inserting a new element into the array.
3. **Deletion**: Removing an element from the array.
4. **Searching**: Finding the location of an element in the array.
5. **Sorting**: Arranging the elements of the array in some order.
6. **Merging**: Combining one or more array to form a single array.

➤ Traversing an array:

• *Traversing is the process of visiting each subscript at least once from the beginning element to last element.*

• For example, to find the maximum element of the array we need to access each element of the array.

**ALGORITHM**: Traversal (A, LB, UB) A is an array with Lower Bound LB and Upper Bound UB.

```
Step 1: for LOC = LB to UB do
Step 2: PROCESS A[LOC]
       [End of for loop]
Step 3: Exit
```
Inserting an element into an array:

- Insertion refers to inserting an element into the array.
- Based on the requirement, new element can be added at the beginning, end or any given position of array.
- When an element is to be inserted into a particular position, all the elements from the asked position to the last element should be shifted into higher order position.
- For example: Let A[5] be an array with items 10, 20, 40, 50, 60 stored at consecutive locations. Suppose item 30 has to be inserted at position 2. The following procedure is applied.
  - Move Number 60 to the position 5.
  - Move Number 50 to the position 4.
  - Move Number 40 to the position 3.

ALGORITHM: Insert (A, N, ITEM, Pos) A is an array with N elements. ITEM is the element to be inserted in the position Pos.

Step 1: for I = N-1 down to Pos
  [End of for loop]
Step 2: A [Pos] = ITEM
Step 3: N = N+1
Step 4: Exit

Deleting an element into an array:

- Deletion refers to deleting an element from the array.
- Based on the requirement, element can be deleted at the beginning, end or any given position of array.
- When an element is to be deleted from a particular position, all the subsequent shifted into lower order position.
For example: Let A[4] be an array with items 10, 20, 30, 40, 50 stored at consecutive locations. Suppose item 30 has to be deleted at position 2. The following procedure is applied.

- Copy 30 to ITEM, i.e. Item = 30.
- Move Number 40 to the position 2.
- Move Number 50 to the position 3.

**ALGORITHM:** Delete (A, N, ITEM, Pos) A is an array with N elements. ITEM is the element to be deleted in the position Pos and it is stored into variable Item.

```
Step 1: ITEM = A[Pos]
Step 2: for I = Pos down to N-1
       [End of for loop]
Step 3: N = N-1
Step 4: Exit
```

 ➢ Searching an element in an array:

- **Searching refers to finding the location of the element in a linear array.**
- There are different algorithms, but the most common methods are linear search and binary search.

 ➢ Linear Search:

- This is the simplest method in which the element to be searched is compared with each element of the array by one from the beginning to end of the array.
- Searching is one after the other, it is also called as *sequential search.*

**ALGORITHM:** Linear Search (A, N, Element) A is an array with N elements. Element is the being searched in the array.

```
Step 1: LOC = -1 [Assume the element does not exist]
Step 2: for I = 0 to N-1 do
       if ( Element = A[I] ) then
```

---

**Page 6**
LOC = I
Goto Step 3
[End if]
[End of for loop]
Step 3: if ( LOC >= 0 ) then
Print Element, “Found in Location”, LOC
else
Print Element, “Not Found”
Step 4: Exit

- **Example**: Consider the following elements stored in an array and we are searching for the element 17. The trace of the algorithm is given below.

<table>
<thead>
<tr>
<th>Index (I)</th>
<th>Compare 17</th>
<th>Location</th>
</tr>
</thead>
<tbody>
<tr>
<td>I = 0</td>
<td>17 = 12</td>
<td>LOC = -1</td>
</tr>
<tr>
<td></td>
<td>(Does not match)</td>
<td></td>
</tr>
<tr>
<td>I = 1</td>
<td>17 = 23</td>
<td>LOC = -1</td>
</tr>
<tr>
<td></td>
<td>(Does not match)</td>
<td></td>
</tr>
<tr>
<td>I = 2</td>
<td>17 = 9</td>
<td>LOC = -1</td>
</tr>
<tr>
<td></td>
<td>(Does not match)</td>
<td></td>
</tr>
<tr>
<td>I = 3</td>
<td>17 = 17</td>
<td>LOC = 3</td>
</tr>
<tr>
<td></td>
<td>(Matches)</td>
<td></td>
</tr>
</tbody>
</table>

- **Binary Search**:
  - When the elements of the array are in sorted order, the best method of searching is binary search.
  - This method compares the element to be searched with the middle element of the array.
  - If the comparison is searched either at the right half of the array or at the left half of the array.
  - Let the position of first (BEG) and last (END) elements of the array. The middle element A[MID] can be obtained by find the middle location MID by MID = (BEG + END)/2.
  - If A[MID] = ELE, then search is successful. Otherwise a new segment is found as follows:
    - If ELE < A[MID], searching is continued at left half of the array. Reset END = MID - 1.
    - If ELE > A[MID], searching is continued at right half of the array. Reset BEG = MID + 1.
    - If ELE not found then we get a condition BEG > END. This results in unsuccessful search.

**ALGORITHM**: Binary_Search (BEG, END, MID ELE) A is an array with N elements. Let BEG, END, MID denote Beginning, end and middle location of the array

Step 1: Set BEG = LB, END = UB LOC = -1
Step 2: While (BEG <= END)
Chapter 4- Data Structures

MID = (BEG+END)/2
if ( ELE = A [ MID ] ) then
   LOC = MID
   Goto Step 3
else
   if( ELE < A[MID])
      END=MID-1;
   else
      BEG=MID+1;
   [End if]
[End if]
[End of While loop]

Step 3: if ( LOC >= 0 ) then
Print Element, “Found in Location”, LOC
else
Print Element, “Search is Unsuccessful”

Step 4: Exit

- Example: Consider the following elements stored in an array and we are searching for the element 47. The trace of the algorithm is given below.

<table>
<thead>
<tr>
<th>BEG &amp; END</th>
<th>MID = (BEG+END)/2</th>
<th>Compare</th>
<th>Location</th>
</tr>
</thead>
<tbody>
<tr>
<td>12 23 39 47 57</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

The search element i.e. 47 is greater than the element in the middle position i.e. 39 then continues the search to the right portion of the middle element.

<table>
<thead>
<tr>
<th>BEG &amp; END</th>
<th>MID = (BEG+END)/2</th>
<th>Compare</th>
<th>Location</th>
</tr>
</thead>
<tbody>
<tr>
<td>12 23 39 47 57</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Since the element in the middle position is the same as the search element LOC gets the value 3. Since the value of LOC is greater than 0, we get the output as **47 Found in location 3**
**Example:** Consider the following elements stored in an array and we are searching for the element 67. The trace of the algorithm is given below.

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>12</td>
<td>23</td>
<td>39</td>
<td>47</td>
<td>57</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>BEG</th>
<th>MID</th>
<th>END</th>
</tr>
</thead>
<tbody>
<tr>
<td>BEG = 0</td>
<td>END = 4</td>
<td></td>
</tr>
</tbody>
</table>

\[ \text{MID} = \frac{\text{BEG} + \text{END}}{2} \]

\[ \text{Comparison} = 67 > 39 \] (Does not match)

\[ \text{Location} = -1 \]

The search element i.e. 67 is greater than the element in the middle position i.e. 39 then continues the search to the right portion of the middle element.

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>12</td>
<td>23</td>
<td>39</td>
<td>47</td>
<td>57</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>BEG</th>
<th>MID</th>
<th>END</th>
</tr>
</thead>
<tbody>
<tr>
<td>BEG = 3</td>
<td>END = 4</td>
<td></td>
</tr>
</tbody>
</table>

\[ \text{MID} = \frac{\text{BEG} + \text{END}}{2} \]

\[ \text{Comparison} = 67 > 47 \] (Does not match)

\[ \text{Location} = -1 \]

The search element i.e. 67 is greater than the element in the middle position i.e. 47 then continues the search to the right portion of the middle element.

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>12</td>
<td>23</td>
<td>39</td>
<td>47</td>
<td>57</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>BEG</th>
<th>MID</th>
<th>END</th>
</tr>
</thead>
<tbody>
<tr>
<td>BEG = 4</td>
<td>END = 4</td>
<td></td>
</tr>
</tbody>
</table>

\[ \text{MID} = \frac{\text{BEG} + \text{END}}{2} \]

\[ \text{Comparison} = 67 > 57 \] (Does not match)

\[ \text{Location} = -1 \]

The search element i.e. 67 is greater than the element in the middle position i.e. 57 then continues the search to the right portion of the middle element.

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>12</td>
<td>23</td>
<td>39</td>
<td>47</td>
<td>57</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>BEG</th>
<th>END</th>
</tr>
</thead>
<tbody>
<tr>
<td>BEG = 5</td>
<td>END = 4</td>
</tr>
</tbody>
</table>

Since the condition \( \text{BEG} \leq \text{END} \) is false the comparison ends

We get the output as 67 Not Found

**Difference between Linear Search and Binary Search**

<table>
<thead>
<tr>
<th>Linear Search</th>
<th>Binary Search</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 This can be used in sorted and unsorted array</td>
<td>This can be used only in sorted array</td>
</tr>
<tr>
<td>2 Array elements are accessed sequentially</td>
<td>One must have direct access to the middle element in the sub list.</td>
</tr>
<tr>
<td>3 Access is very slow</td>
<td>Access is faster.</td>
</tr>
<tr>
<td>4 This can be used in single and multidimensional array</td>
<td>Used only in single dimensional array.</td>
</tr>
<tr>
<td>5 This technique is easy and simple in implementing</td>
<td>Complex in operation</td>
</tr>
</tbody>
</table>
Sorting the elements in an array:

- Sorting is the arrangement of elements of the array in some order.
- There are different sorting methods like Bubble Sort, Selection Sort, Shell Sort, Quick sort, Heap Sort, Insertion Sort etc.

Insertion Sort:

- In Insertion sort, the first element of the array is assumed to be in the correct position next element is considered as the key element and compared with the elements before the key element and is inserted in its correct position.

Example: Consider the following array contains 8 elements as follows:

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>45</td>
<td>26</td>
<td>23</td>
<td>56</td>
<td>29</td>
<td>36</td>
<td>12</td>
<td>4</td>
</tr>
</tbody>
</table>

**Algorithm:**

```
ALGORITHM: Insertion_Sort (A, N) A is an array with N unsorted elements.

Step 1: for I=1 to N-1
Step 2: J = I
While(J >= 1)
        Temp = A[J];
        A[J-1] = Temp;
    [End if]
    J = J-1
[End of While loop]
[End of For loop]
Step 3: Exit
```
Two dimensional array:

- A two dimensional array is a collection of elements and each element is identified by a pair of subscripts. (A[5][5])
- The elements are stored in continuous memory locations.
- The elements of two-dimensional array as rows and columns.
- The number of rows and columns in a matrix is called as the order of the matrix and denoted as mxn.
- The number of elements can be obtained by multiplying number of rows and number of columns.

<table>
<thead>
<tr>
<th></th>
<th>[0]</th>
<th>[1]</th>
<th>[2]</th>
</tr>
</thead>
<tbody>
<tr>
<td>A[0]</td>
<td>1</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>A[1]</td>
<td>7</td>
<td>8</td>
<td>9</td>
</tr>
</tbody>
</table>

Representation of Two Dimensional Array:

- A is the array of order m x n. To store m*n number of elements, we need m*n memory locations. The elements should be in contiguous memory locations.
- There are two methods:
  - Row-major method
  - Column-major method

Row-Major Method: All the first-row elements are stored in sequential memory locations and then all the second-row elements are stored and so on.

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1001</td>
<td>1</td>
<td>A[0][0]</td>
</tr>
<tr>
<td>1002</td>
<td>7</td>
<td>A[0][1]</td>
</tr>
<tr>
<td>1003</td>
<td>4</td>
<td>A[0][2]</td>
</tr>
<tr>
<td>1004</td>
<td>2</td>
<td>A[1][0]</td>
</tr>
<tr>
<td>1005</td>
<td>8</td>
<td>A[1][1]</td>
</tr>
<tr>
<td>1006</td>
<td>5</td>
<td>A[1][2]</td>
</tr>
<tr>
<td>1007</td>
<td>4</td>
<td>A[2][0]</td>
</tr>
<tr>
<td>1008</td>
<td>5</td>
<td>A[2][1]</td>
</tr>
<tr>
<td>1009</td>
<td>6</td>
<td>A[2][2]</td>
</tr>
</tbody>
</table>

Column Major Method:

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1001</td>
<td>1</td>
<td>A[0][0]</td>
</tr>
<tr>
<td>1002</td>
<td>2</td>
<td>A[1][0]</td>
</tr>
<tr>
<td>1003</td>
<td>3</td>
<td>A[2][0]</td>
</tr>
<tr>
<td>1004</td>
<td>7</td>
<td>A[0][1]</td>
</tr>
<tr>
<td>1005</td>
<td>8</td>
<td>A[1][1]</td>
</tr>
<tr>
<td>1006</td>
<td>9</td>
<td>A[2][1]</td>
</tr>
<tr>
<td>1007</td>
<td>3</td>
<td>A[0][2]</td>
</tr>
<tr>
<td>1008</td>
<td>9</td>
<td>A[1][2]</td>
</tr>
<tr>
<td>1009</td>
<td>6</td>
<td>A[2][2]</td>
</tr>
</tbody>
</table>

Column-Major Method: All the first column elements are stored in sequential memory locations and then all the second-column elements are stored and so on.
- **Advantages of Array:**
  - It is used to represent multiple data items of same type by using single name.
  - It can be used to implement other data structures like linked lists, stacks, queues, tree, graphs etc.
  - Two-dimensional arrays are used to represent matrices.
  - Many databases include one-dimensional arrays whose elements are records.

- **Disadvantages of Array:**
  - We must know in advance the how many elements are to be stored in array.
  - Array is static structure. It means that array is of fixed size. The memory which is allocated to array cannot be increased or decreased.
  - Array is fixed size; if we allocate more memory than requirement then the memory space will be wasted.
  - The elements of array are stored in consecutive memory locations. So insertion and deletion are very difficult and time consuming.

- **STACKS:**
  - A stack is an ordered collection of items in which an element may be inserted or deleted only at same end.
  - This end is called as the top of the stack.
  - The end opposite to top is known as the base.
  - Stacks are sometimes known as LIFO (Last In First Out).

- **Representation of Stacks using Array:**
  - Stack can be represented using a one-dimensional array.
  - The items into the stack are stored in a sequential order from the first location of the memory block.
  - A pointer TOP contains the location of the top element of the stack.
  - A variable MAXSTK contains the maximum number of element that can be stored in the stack.
  - If we attempt to add new element beyond the maximum size, we will encounter a stack **overflow condition**. Similarly, you cannot remove elements beyond the base of the stack. If such is the case, we will reach a stack **underflow condition**.
  - The condition TOP = MAXSTX indicates that the stack is full and TOP = NULL indicates that stack is empty.
Operation on Stacks:

- **Stack()**: It creates a new stack that is empty. It needs no parameter and returns an empty stack.
- **push(item)**: It adds a new item to the top of the stack.
- **pop()**: It removes the top item from the stack.
- **peek()**: It returns the top item from the stack but does not remove it.
- **isEmpty()**: It tests whether the stack is empty.
- **size()**: It returns the number of items on the stack.

PUSH Operation:

- The process of adding one element or item to the stack is represented by an operation called **the PUSH operation**.
- The new element is added at the topmost position of the stack.

**ALGORITHM**:
PUSH (STACK, TOP, ITEM)

1. **Step 1**: if TOP = N-1 then 
   PRINT "STACK is Full or Overflow"
   Exit
   [End if]

2. **Step 2**: TOP = TOP + 1 
   [Increment the TOP]

3. **Step 3**: STACK[TOP] = ITEM 
   [Insert the ITEM]

4. **Step 4**: Return

POP Operation:

- The process of deleting one element or item from the stack is represented by an operation called **the POP operation**.

**ALGORITHM**:
POP (STACK, TOP, ITEM)

1. **Step 1**: if TOP = NULL then 
   PRINT "STACK is Empty or Underflow"
   Exit
   [End if]

2. **Step 2**: ITEM = STACK[TOP] 
   [copy the TOP Element]

3. **Step 3**: TOP = TOP - 1 
   [Decrement the TOP]

4. **Step 4**: Return
Application of Stacks:
- It is used to reverse a word. You push a given word to stack – letter by letter and then pop letter from the stack.
- “Undo” mechanism in text editor.
- Backtracking: This is a process when you need to access the most recent data element in a series of elements. Once you reach a dead end, you must backtrack.
- Language Processing: Compiler’ syntax check for matching braces in implemented by using stack.
- Conversion of decimal number to binary.
- Conversion of infix expression into prefix and postfix.
- Quick sort
- Runtime memory management.

Arithmetic Expression:
- An expression is a combination of operands and operators that after evaluation results in a single value.
- Operand consists of constants and variables.
- Operators consists of {, +, -, *, /, ), ] etc.
- Expression can be
  - Infix Expression
  - Postfix Expression
  - Prefix Expression

Infix Expression: If an operator is in between two operands, it is called infix expression.
Example: a + b, where a and b are operands and + is an operator.

Postfix Expression: If an operator follows the two operands, it is called postfix expression.
Example: ab +

Prefix Expression: an operator precedes the two operands, it is called prefix expression.
Example: +ab

Algorithm for Infix to Postfix:
1. Examine the next element in the input.
2. If it is operand, output it.
3. If it is opening parenthesis, push it on stack.
4. If it is an operator, then
   - If stack is empty, push operator on stack.
- If the top of the stack is opening parenthesis, push operator on stack.
- If it has higher priority than the top of stack, push operator on stack.
- Else pop the operator from the stack and output it, repeat step 4.

5. If it is a closing parenthesis, pop operators from the stack and output them until an opening parenthesis is encountered. Pop and discard the opening parenthesis.

6. If there is more input go to step 1.

7. If there is no more input, pop the remaining operators to output.

- Example: Suppose we want to convert 2*3/(2-1)+5*3 into postfix form.

<table>
<thead>
<tr>
<th>Expression</th>
<th>Stack</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>Empty</td>
<td>2</td>
</tr>
<tr>
<td>*</td>
<td>*</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>*</td>
<td>23</td>
</tr>
<tr>
<td>/</td>
<td>/</td>
<td>23*</td>
</tr>
<tr>
<td>(</td>
<td>/(</td>
<td>23*</td>
</tr>
<tr>
<td>2</td>
<td>/(</td>
<td>23*2</td>
</tr>
<tr>
<td>-</td>
<td>/-</td>
<td>23*2</td>
</tr>
<tr>
<td>1</td>
<td>/-</td>
<td>23*21</td>
</tr>
<tr>
<td>)</td>
<td>/</td>
<td>23*21-</td>
</tr>
<tr>
<td>+</td>
<td>+</td>
<td>23*21-/</td>
</tr>
<tr>
<td>5</td>
<td>+</td>
<td>23*21-/5</td>
</tr>
<tr>
<td>*</td>
<td>++</td>
<td>23*21-/5</td>
</tr>
<tr>
<td>3</td>
<td>++</td>
<td>23*21-/53</td>
</tr>
<tr>
<td></td>
<td>Empty</td>
<td>23<em>21-/53</em>+</td>
</tr>
</tbody>
</table>

- Consider the following examples of converting from infix to postfix

<table>
<thead>
<tr>
<th>Sl No</th>
<th>Infix</th>
<th>Postfix</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>(A + B) * C</td>
<td>AB+C*</td>
</tr>
<tr>
<td>2</td>
<td>A + B + C</td>
<td>AB+C+</td>
</tr>
<tr>
<td>3</td>
<td>(A + B) / (X - Y)</td>
<td>AB[XY-]</td>
</tr>
<tr>
<td>4</td>
<td>A ^ B * C - D</td>
<td>AB[C*D-]</td>
</tr>
<tr>
<td>5</td>
<td>((A + B) * (C - (D - E)) ^ (X + Y))</td>
<td>AB[C*DE--XY+^]</td>
</tr>
</tbody>
</table>
• Consider the following examples of converting from infix to prefix.

<table>
<thead>
<tr>
<th>Sl No</th>
<th>Infix</th>
<th>Prefix</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>( A + B ) * C</td>
<td>[*+AB] * C</td>
</tr>
<tr>
<td>2</td>
<td>A + B + C</td>
<td>[+]AB] + C</td>
</tr>
<tr>
<td>3</td>
<td>( A + B ) / ( X – Y )</td>
<td>[*+AB] / [-XY]</td>
</tr>
<tr>
<td>4</td>
<td>A ^ B * C – D</td>
<td>[^AB] * C-D</td>
</tr>
<tr>
<td>5</td>
<td>((A + B)*C–(D–E))^X+Y</td>
<td>(^AB]*[-DE]^ [+XY]</td>
</tr>
</tbody>
</table>

**QUEUES:**

- A queue is an ordered collection of items where an item is inserted at one end called the “rear” and an existing item is removed at the other end, called the “front”.
- Queue is also called as FIFO list i.e. First-In First-Out.
- In the queue only two operations are allowed enqueue and dequeue.
- Enqueue means to insert an item into back of the queue.
- Dequeue means removing the front item.

**Types of Queues:**

- Queue can be of four types:
  - Simple Queue
  - Circular Queue
  - Priority Queue
  - Dequeue (Double Ended Queue)

- **Simple Queue:** In simple queue insertion occurs at the rear end of the list and deletion occurs at the front end of the list.
- **Circular Queue**: A circular queue is a queue in which all nodes are treated as circular such that the last node follows the first node.

- **Priority Queue**: A priority queue is a queue that contains items that have some present priority. An element can be inserted or removed from any position depending upon some priority.

- **Dequeue**: It is a queue in which insertion and deletion takes place at the both ends.

---

**Operation on Queues:**
- **Queue()**: It creates a new queue that is empty.
- **enqueue(item)**: It adds a new item to the rear of the queue.
- **dequeue()**: It removes the front item from the queue.
- **isEmpty()**: It tests to see whether the queue is empty.
- **size()**: It returns the number of items in the queue.

**Memory Representation of a queue using array:**
- Queue is represented in memory using linear array.
- Let QUEUE is a array, two pointer variables called FRONT and REAR are maintained.
- The pointer variable FRONT contains the location of the element to be removed or deleted.
- The pointer variable REAR contains location of the last element inserted.
- The condition FRONT = NULL indicates that queue is empty.
- The condition REAR = N-1 indicates that queue is full.

![Queue Diagram](image)

**Operation Examples:**
- Front = 1
- Rear = 4
- REAR = REAR + 1, QUEUE [REAR] = ‘E’

![Queue Diagram](image)

**Operation Examples:**
- Front = 1
- Rear = 5
ITEM = QUEUE [FRONT], FRONT = FRONT + 1

- Queue Insertion Operation (ENQUEUE):
  - Consider a queue Q with 5 elements.

|------|------|------|------|------|

Initially Queue is Empty FRONT = -1, REAR = -1

- If we want to add one element i.e. 8 in the queue, then FRONT and REAR Indicator are set to 0 and the element 8 would be stored at the position pointed by the REAR.
  \[ \text{REAR} = 0, \quad \text{FRONT} = 0, \quad \text{Q}[\text{REAR}] = \text{Q}[0] = 8 \]

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>8</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- If we want to add one more element i.e. 12 in the queue, then REAR Indicator would be incremented by 1 and the element 12 would be stored at the position pointed by the REAR.
  \[ \text{REAR} = 1, \quad \text{FRONT} = 0, \quad \text{Q}[\text{REAR}] = \text{Q}[1] = 12 \]

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>8</td>
<td>12</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- Repeat this procedure three more times to insert 4, 18 and 34 elements.
  \[ \text{REAR} = 4, \quad \text{FRONT} = 0, \quad \text{Queue is full.} \]

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>8</td>
<td>12</td>
<td>4</td>
<td>18</td>
<td>34</td>
</tr>
</tbody>
</table>

- Algorithm for Insertion:

**ALGORITHM:** ENQUEUE (QUEUE, REAR, FRONT, ITEM) QUEUE is the array with N elements. FRONT is the pointer that contains the location of the element to be deleted and REAR contains the location of the inserted element. ITEM is the element to be inserted.

Step 1: if REAR = N-1 then [Check Overflow]
  PRINT “QUEUE is Full or Overflow”
  Exit
[End if]

Step 2: if FRONT = NULL then [Check Whether Queue is empty]
  FRONT = 0
  REAR = 0
else
  REAR = REAR + 1 [Increment REAR Pointer]

Step 3: QUEUE[REAR] = ITEM [Copy ITEM to REAR position]

Step 4: Return
Queue Deletion Operation (DEQUEUE):

- Consider a queue Q with 3 elements.

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>8</td>
<td>12</td>
<td>4</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

FRONT = 0, REAR = 2, Delete 8

- If we want to delete an element i.e. 8 from the queue, then the value of FRONT will be incremented by 1. Deletions are done from this end of the queue.

REAR = 2, FRONT = 1,

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>12</td>
<td>4</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- If we want to delete an element i.e. 12 from the queue, then the value of FRONT will be incremented by 1 i.e. 2.

REAR = 2, FRONT = 2

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>4</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- Observe that queue has only one element. When FRONT = REAR condition is true, then the queue Q has only one element. If we want to this element also, then the queue Q becomes empty and set the FRONT and REAR pointers to -1.

REAR = -1, FRONT = -1, Queue is Empty.

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Algorithm for Deletion:

**ALGORITHM:** DEQUEUE (QUEUE, REAR, FRONT, ITEM) QUEUE is the array with N elements. FRONT is the pointer that contains the location of the element to be deleted and REAR contains the location of the inserted element. ITEM is the element to be deleted.

1. if FRONT = NULL then
   - PRINT “QUEUE is Empty or Underflow”
   - Exit
2. ITEM = QUEUE[FRONT]
3. if FRONT = REAR then
   - FRONT = NULL
   - REAR = NULL
   - else
   - FRONT = FRONT + 1
4. Return
Application of Queue:
- Simulation
- Various features of Operating system
- Multi-programming platform systems.
- Different types of scheduling algorithms
- Round robin technique algorithms
- Printer server routines
- Various application software’s is also based on queue data structure.

LINKED LIST:
- A linked list is a linear collection of data elements called nodes.
  Each nodes is divided into two parts:
  - The first part contains the information of the element.
  - The second part contains the memory address of the next node in the list. Also called Link part.

In the above figure shows an representation if Linked List with three nodes, each node contains two parts.
- The left part of the node represents Information part of the node.
- The right part represents the next pointer field. That contains the address of the next node.
- A pointer START or HEAD gives the location of the first node.
- The link field of the last node contains NULL.

Types of Linked List:
- There are three types of linked lists:
  - Singly Linked List
  - Doubly Linked List
  - Circular Linked List

Singly Linked List:
- A singly linked list contains two fields in each node - an information field and the linked field.
• The information field contains the data of that node.
• The link field contains the memory address of the next node.
• There is only one link field in each node, the linked list is called singly linked list.

**Circular Linked List**
• The link field of the last node contains the memory address of the first node, such a linked list is called circular linked list.
• In a circular linked list every node is accessible from a given node.

**Doubly Linked List**
• It is a linked list in which each node is points both to the next node and also to the previous node.
• In doubly linked list each node contains three parts:
  - FORW: It is a pointer field that contains the address of the next node
  - BACK: It is a pointer field that contains the address of the previous node.
  - INFO: It contains the actual data.
• In the first node, if BACK contains NULL, it indicated that it is the first node in the list.
• The node in which FORW contains, NULL indicates that the node is the last node.

**Operation on Linked List:**
• The operation that are performed on linked lists are:
  - Creating a linked list
  - Traversing a linked list
  - Inserting an item into a linked list.
  - Deleting an item from the linked list.
  - Searching an item in the linked list
  - Merging two or more linked lists.
✓ Creating a linked list:
- The nodes of a linked list can be created by the following structure declaration.
  ```c
  struct Node
  {
    int info;
    struct Node *link;
  }*node1, node2;
  ```
- Here info is the information field and link is the link field.
- The link field contains a pointer variable that refers the same node structure. Such a reference is called as Self addressing pointer.

✓ Operator new and delete:
- Operators new allocate memory space.
- Operators new [] allocates memory space for array.
- Operators delete deallocate memory space.
- Operators delete [] deallocate memory space for array.

✓ Traversing a linked list:
- Traversing is the process of accessing each node of the linked list exactly once to perform some operation.
  **ALGORITHM:** TRAVERS (START, P) START contains the address of the first node. Another pointer p is temporarily used to visit all the nodes from the beginning to the end of the linked list.
  ```
  Step 1: P = START
  Step 2: while P != NULL
  Step 3: PROCESS data (P)          [Fetch the data]
  Step 4: P = link(P)               [Advance P to next node]
  Step 5: End of while
  Step 6: Return
  ```

✓ AVAIL List
- The operating system of the computer maintains a special list called AVAIL list that contains only the unused deleted nodes.
- Whenever node is deleted from the linked list, its memory is added to the AVAIL list.
- AVAIL list is also called as the free-storage list or free-pool.
✓ Inserting a node into the linked list:
  - Inserting a node at the beginning of the linked list
  - Inserting a node at the given position.
  - Inserting a node at the end of the linked list.

• Inserting a node at the beginning of the linked list
  1. Create a node.
  2. Fill data into the data field of the new node.
  3. Mark its pointer field as NULL
  4. Attach this newly created node to START
  5. Make the new node as the STARTing node.

**ALGORITHM**: INS_BEG (START, P) START contains the address of the first node.

```
Step 1: P ← new Node;
Step 2: data(P) ← num;
Step 3: link(P) ← START
Step 4: START ← P
Step 5: Return
```

✓ Garbage Collection:

• The operating system of the computer periodically collects all the deleted space into the free-storage list. This technique of collecting deleted space into free-storage list is called as **garbage collection**.

✓ Deleting an item from the linked list:

  - Deletion of the first node
  - Deletion of the last node
  - Deletion of the node at the given position

• Deletion of the last node

**ALGORITHM**: DEL_END (P1, P2, START) This used two pointers P1 and P2. Pointer P2 is used to traverse the linked list and pointer P1 keeps the location of the previous node of P2.

```
Step 1: START
Step 2: P2 ← START;
Step 3: while (link(P2) != NULL)
        P1 ← P2
```

P2 ← link(P2)
While end
Step 4: PRINT data(p2)
Step 5: link(P1) ← NULL
        Free(P2)
Step 6: STOP

- **Non-Linear Data structures:**
  - A Non-Linear Data structures is a data structure in which data item is connected to several other data items.
  - The data items in non-linear data structure represent hierarchical relationship.
  - Each data item is called node.
  - The different non-linear data structures are trees and graphs.

- **Trees:**
  - A tree is a data structure consisting of nodes organized as a hierarchy.

- **Terminology of a tree:**
  - **Node:** A node is a fundamental part of tree. Each element of a tree is called a node of the tree.
  - **Root:** The topmost node of the tree.
  - **Edge:** It connects two nodes to show that tree is a relationship between them.
  - **Parent:** It is an immediate predecessor a node. (A is parent of B, C, D)
  - **Child:** A successor of parent node is called the child node. (E, F is a child of B)
  - **Siblings:** Nodes that are children of the same parent are called siblings. (B,C,D) (M,N)
  - **Leaf or terminal node:** Nodes that do not have any children are called leaf node. (J, K, L)
  - **Internal Node:** A node has one or more children is called an internal node. (B, C, D, H)
**Path:** A path is an ordered list of nodes that are connected by edges. In figure, path A to O is A, D, I, M, O.

**Height:** The height or depth of a tree is defined to be the maximum number of nodes in a branch of tree. (The height of tree is 5)

**Ancestor:** A node reachable by repeated proceeding from child to parent. (Ancestor of M is I, D, and A)

**Descendant:** A node reachable by repeated proceedings from parent to child. (Descendent of I are (M, O) and N)

**Subtree:** A Subtree is a set of nodes and edges comprised of parent and all the descendants of that parent. In figure T1, T2, and T3 are subtrees.

➢ **Binary tree:**

- A binary tree is an ordered tree in which each internal node can have maximum of two child nodes connected to it.
- A binary tree consists of:
  - A node (called the root node)
  - Left and right sub trees.

![Binary Tree Diagram]

- A Complete binary tree is a binary tree in which each leaf is at the same distance from the root i.e. all the nodes have maximum two subtrees.

➢ **GRAPH**

- A graph is a set of vertices and edges which connect them.
- A graph is a collection of nodes called vertices and the connection between them called edges.
- **Directed graph:** When the edges in a graph have a direction, the graph is called directed graph or digraph, and the edges are called directed edges or arcs.
## CHAPTER – DATA STRUCTURES BLUE PRINT

<table>
<thead>
<tr>
<th>VSA (1 Marks)</th>
<th>LA (3 Marks)</th>
<th>Essay (5 Marks)</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>01 Question</td>
<td>01 Question</td>
<td>02 Question</td>
<td>04 Questions</td>
</tr>
<tr>
<td>Question No 3</td>
<td>Question No 23</td>
<td>Question No 28 &amp; 29</td>
<td>14 Marks</td>
</tr>
</tbody>
</table>

### Important Questions

**1 Mark Questions:**

1. Definition of Data Structure, Linear Data Structures, Array, Stack, Queue, Linked List, Non Linear Data Structures.
2. Definition on Tree, Tree Terminologies, Binary Tree, Complete Binary Tree, Graph.

**3 Marks Questions:**

1. Types of Data Structure, Array, Queue, Linked list.
2. Memory Representation of Array, Stack.
3. Algorithms – Traverse, Insertion, Deletion, Linear Search, PUSH, POP
5. Advantages/Applications of arrays, stacks, queues

**5 Marks Questions:**

- Operations on Non-Primitive Data Structure, Array, Stack, Queue, Linked list.
- Tracing Algorithm: Binary Search, Insertion Sort.