WHAT IS DATA STRUCTURE

Array, Tree, Linked List
**WHAT IS DATA STRUCTURE?**

“The way information is organized in the memory of a computer is called a **data structure**”.

(OR)

A data structure is a way of organizing data that considers not only the items stored, but also their relationship to each other. Advance knowledge about the relationship between data items allows designing of efficient algorithms for the manipulation of data.

**Definition of data structures**

- Many algorithms require that we use a proper representation of data to achieve efficiency.
- This representation and the operations that are allowed for it are called data structures.
- Each data structure allows insertion, access, deletion etc.

**Why do we need data structures?**

- Data structures allow us to achieve an important goal: component reuse
- Once each data structure has been implemented once, it can be used over and over Again in various applications.

**Common data structures are**

- Stacks
- Queues
- Lists
- Trees
- Graphs
- Tables

**Classification of data Structure:**

Based on how the data items are operated it will classified into

1. **Primitive Data Structure:** is one the data items are operated closest to the machine level instruction.
   
   **Eg:** int, char and double.

2. **Non-Primitive Data Structure:** is one that data items are not operated closest to machine level instruction.

2.1. **Linear Data Structure:** In which the data items are stored in sequence order.
   
   **Eg:** Arrays, Lists, Stacks and Queues.

2.2. **Non Linear Data Structure:** In which the order of data items is not presence.

   **Eg:** Trees, Graphs.
Linear Data Structure

1. List
   a. Array
      i. One Dimensional
      ii. Multi-Dimensional
      iii. Dynamic Array
      iv. Matrix
      1. Sparse Matrix
   b. Linked List
      i. Single Linked List
      ii. Double Linked List
      iii. Circular Linked List
   c. Ordered List
      i. Stack
      ii. Queue
      1. Circular Queue
      2. Priority Queue
   iii. Deque
2. Dictionary (Associative Array)
   a. Hash Table

Non-Linear Data Structures

1. Graph
   a. Adjacency List
   b. Adjacency Matrix
   c. Spanning Tree

2. Tree
   a. M-Way Tree
      i. B-Tree
      1. 2-3-4 Tree
      2. B+ Tree
   b. Binary Tree
      i. Binary Search Tree
      ii. Self-Balancing Binary Search Tree
      1. AVL Tree
      2. Red-Black Tree
      3. Splay Tree
      iii. Heap
      1. Min Heap
      2. Max Heap
      3. Binary Heap
   iv. Parse Tree

Operations performed on any linear structure:

1. Traversal – Processing each element in the list
2. Search – Finding the location of the element with a given value.
3. Insertion – Adding a new element to the list.
4. Deletion – Removing an element from the list.
5. Sorting – Arranging the elements in some type of order.
6. Merging – Combining two lists into a single list.
An example of several common data structures Characteristics of Data Structures

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

Abstract data type (ADT) is a specification of a set of data and the set of operations that can be performed on the data.

Examples

- Associative array
- Set
- Stack
- Queue
- Tree

Uses of ADT: -

1. It helps to efficiently develop well designed program
2. Facilitates the decomposition of the complex task of developing a software system into a number of simpler subtasks
3. Helps to reduce the number of things the programmer has to keep in mind at any time
4. Breaking down a complex task into a number of earlier subtasks also simplifies testing and debugging

Algorithm:

Definition: An algorithm is a finite set of instructions which, if followed, accomplish a particular task. In addition every algorithm must satisfy the following criteria:

1. input: there are zero or more quantities which are externally supplied;
2. output: at least one quantity is produced;
3. definiteness: each instruction must be clear and unambiguous;
4. finiteness: if we trace out the instructions of an algorithm, then for all cases the algorithm will terminate after a finite number of steps;

Linear Data Structures

A data structure is said to be linear if its elements form a sequence or a linear list.

Examples:

Arrays
Linked Lists
Stacks, Queues
Arrays

Arrays
- The very common linear structure is array. Since arrays are usually easy to traverse, search and sort, they are frequently used to store relatively permanent collections of data.
- An array is a list of a finite number n of homogeneous data elements (i.e., data elements of the same type) such that:
  a) The elements of the array are referenced respectively by an index consisting of n consecutive numbers.
  b) The elements of the array are stored respectively in successive memory locations.

Operations of Array

- Two basic operations in an array are storing and retrieving (extraction)

Storing: A value is stored in an element of the array with the statement of the form,

\[ \text{Data}[i] = X; \] Where I is the valid index in the array
And X is the element

Extraction: Refers to getting the value of an element stored in an array.

\[ X = \text{Data}[i], \] Where I is the valid index of the array and X is the element.

Array Representation
- The number n of elements is called the length or size of the array. If not explicitly stated we will assume that the index starts from 0 and end with n-1.
- In general, the length (range) or the number of data elements of the array can be obtained from the index by the formula,

\[ \text{Length} = \text{UB} - \text{LB} + 1 \]

Where UB is the largest index, called the Upper Bound, and LB is the smallest index, called Lower Bound, of the array.
- If LB = 0 and UB = 4 then the length is,

\[ \text{Length} = 4 - 0 + 1 = 5 \]
- The elements of an array A may be denoted by the subscript notation (or bracket notation),

\[ A[0], A[1], A[2], \ldots, A[N] \]
The number K in A[K] is called a *subscript* or an *index* and A[K] is called a *subscripted variable*.
Subscripts allow any element of A to be referenced by its relative position in A.
If each element in the array is referenced by a single subscript, it is called a *single dimensional array*.
In other words, the number of subscripts gives the dimension of that array.

**Two-dimensional Arrays**

A two-dimensional \( m \times n \) array A is a collection of \( m \times n \) data elements such that each element is specified by a pair of integers (such as \( i, j \)), called subscripts, with the property that,

\[
0 \leq i < m \quad \text{and} \quad 0 \leq j < n
\]

The element of A with first subscript \( i \) and second subscript \( j \) will be denoted by,

\[ A[i,j] \quad \text{or} \quad A[i][j] \quad (c \text{ language}) \]

Two-dimensional arrays are called *matrices* in mathematics and *tables* in business applications; hence two-dimensional arrays are sometimes called *matrix arrays*.

There is a standard way of drawing a two-dimensional \( m \times n \) array A where the elements of A form a rectangular array with \( m \) rows and \( n \) columns and where the element \( A[i][j] \) appears in row \( i \) and column \( j \).

A *row* is a horizontal list of elements, and a *column* is a vertical list of elements.

**Example:**

<table>
<thead>
<tr>
<th>Rows</th>
<th>0</th>
<th>1</th>
<th>2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>A[0][0]</td>
<td>A[0][1]</td>
<td>A[0][2]</td>
</tr>
</tbody>
</table>

The two-dimensional array will be represented in memory by a block of \( m \times n \) sequential memory locations.
Specifically, the programming languages will store the array either

1. Column by column, i.e. *column-major order*, or
2. Row by row, i.e. *row-major order*. 

Unit I
Abstract Data Types (ADT)

- The ADT consists of a set of definitions that allow programmers to use the functions while hiding the implementation. This generalization of operations with unspecified implementations is known as abstraction.
- An ADT is a data declaration packaged together with the operations that are meaningful on the data type.
  1. Declaration of Data
  2. Declaration of Operations

An array is a collection of memory locations which allows storing homogeneous elements. It is an example for linear data structure.

An array lets you declare and work with a collection of values of the same type (homogeneous). For example, you might want to create a collection of five integers. One way to do it would be to declare five integers directly:

```c
int a, b, c, d, e;
```

Suppose you need to find the average of 100 numbers. What will you do? You have to declare 100 variables. For example:

```c
int a, b, c, d, e, f, g, h, i, j, k, l, m, n... etc.;
```

An easier way is to declare an array of 100 integers:

```c
int a[100];
```

The General Syntax is:

```
datatype array_name [size];
```

Example:

```c
int a[5];
```

The five separate integers inside this array are accessed by an index. All arrays start at index zero and go to n-1 in C. Thus, `int a[5];` contains five elements. For example:

```
a[0] = 12;
a[1] = 9;
a[2] = 14;
a[3] = 5;
a[4] = 1;
```

Note: The array name will hold the address of the first element. It is called as BASE ADDRESS of that array. The base address can’t be modified during execution, because it is static. It means that the increment / decrement operation would not work on the base address.
Consider the first element is stored in the address of 1020. It will look like this,

\[
\begin{array}{cccccc}
& 1020 & 1022 & 1024 & 1026 & 1028 \\
0 & 12 & 9 & 14 & 5 & 1 \\
1 & & & & & \\
\end{array}
\]

a[0] means a + 0  \(\rightarrow\) 1020 + 0  \(\rightarrow\) 1020 (locates the 1020)
a[1] means a + 1  \(\rightarrow\) 1020 + 1 \* size of datatype  \(\rightarrow\) 1020 + 2  \(\rightarrow\) 1022  [ for ‘int’ size is 2 byte]
a[2] means a + 2  \(\rightarrow\) 1020 + 2 \* size of datatype  \(\rightarrow\) 1020 + 4  \(\rightarrow\) 1024
a[3] means a + 3  \(\rightarrow\) 1020 + 3 \* size of datatype  \(\rightarrow\) 1020 + 6  \(\rightarrow\) 1026
a[4] means a + 4  \(\rightarrow\) 1020 + 4 \* size of datatype  \(\rightarrow\) 1020 + 8  \(\rightarrow\) 1028

Array indexing helps to manipulate the index using a for loop. Because of that retrieval of element from an array is very easy. For example, the following code initializes all of the values in the array to 0:

```c
int a[5]; /* Array declaration */
int i;

/* Initializing Array Elements to 0 */
for (i=0; i<5; i++)
a[i] = 0;

/* print array */
printf("Elements in the array are\\n");
for (i=0; i < 5; i++)
    printf("%d\\n",a[i]);
```

Note :  (mathematics) A matrix most of whose entries are zeros.

**Advantages:**
- Reduces memory access time, because all the elements are stored sequentially. By incrementing the index, it is possible to access all the elements in an array.
- Reduces no. of variables in a program.
- Easy to use for the programmers.

**Disadvantages:**
- Wastage of memory space is possible. For example: Storing only 10 elements in a 100 size array. Here, remaining 90 elements space is waste because these spaces can’t be used by other programs till this program completes its execution.
- Storing heterogeneous elements are not possible.
- Array bound checking is not available in ‘C’. So, manually we have to do that.

Note :  Memory representation of 1, 2 and multidimensional array refer class notes:
STRUCTURES

1. Definition:

`struct`: Declares a structure, an object consisting of multiple data items that may be of different types.

2. Defining a Structure:

Syntax:

```c
struct tag
{
    data-type member 1;
    data-type member 2;
    ..........  
    data-type member m;
}
```

Here, `struct` is the required keyword; `tag` (optional) is a name that identifies structures of this type; and `member1`, `member2`, ..., `member m` are individual member declarations.

- The individual members can be ordinary variables, pointers, arrays, or other structures.
- A storage class cannot be assigned to an individual member, and individual members can not be initialized within a structure type declaration.

3. Declaring Structure Variables:

Once the composition of the structure has been defined, individual structure-type variables can be declared as follows:

```c
storage-class struct tag variable1, variable2, ..., variable n;
```

where `storage-class` is an optional storage class specifier, `struct` is a required keyword, `tag` is the name that appeared in the structure declaration and `variable1`, `variable2`, ..., `variable n` are structure variables of type `tag`.

Example:

```c
struct student
{
    int regno;
    char name[20];
    char dept[10];
    int year;
};
```
Here, regno, name, dept and year are the members of the student structure. And this is the definition of the datatype. So, no memory will be allocated at this stage. The memory will be allocated after the declaration only. Structure variables can be declared as following methods:

a) Normal way of declaration

```c
struct student s1, s2;
```

b) It is possible to combine the declaration of the structure composition with that of the structure variables, as shown below:

```c
struct student
{
    int regno;
    char name[20];
    char dept[10];
    int year;
} s1, s2;
```

c) If we are going to declare all the necessary structure variables at definition time then we can create them without the tag, as shown below:

```c
struct
{
    int regno;
    char name[20];
    char dept[10];
    int year;
} s1, s2;
```

Since there is no tag name, additional variables cannot be generated other than this location. i.e. cannot create new variables with this structure in the local functions. If we want we have to redefine the structure variable once again.

d) If we use the `typedef` in front of the `struct` keyword then the tag name alone can be used in other places whenever you want to use the student data type.

```c
typedef struct student
{
    int regno;
    char name[20];
    char dept[10];
    int year;
} ;
```

```
struct student
{
    int regno;
    char name[20];
    char dept[10];
    int year;
} ;
```

/* here the struct keyword is not needed because of typedef */
The size of each of these variables is 34 bytes because the size of the student datatype is 34 bytes. And the memory will be allocated for each variable as follows:

![Address Diagram]

4. **Initializing Structure Variables:**

   The members of a structure variable can be assigned initial values in much the same manner as the elements of an array. The initial values must appear in the order in which they will be assigned to their corresponding structure members, enclosed in braces and separated by commas.

   The general form is,

   
   ```
   storage-class struct tag variable = {value1, value2, ...,value n};
   ```

   A structure variable, like an array, can be initialized only if its storage class is either external or static.

   **Example:**

   ```
   static struct student s1 = {340, “Kumara Vel”, “CSE”, 3};
   static struct student s2 = {533, “Sankari”, “CSE”, 4};
   ```
5. Storing Values into the Members of the Structure Variables:

a) Values may be stored by assignment operation.
   
   ```
   s1.regno = 500;
   strcpy(s1.name, “Surya”);
   strcpy(s1.dept, “CSE”);
   s1.year = 3;
   ```

b) also the scanf statement may be used to give values through the keyboard.
   
   ```
   scanf(“%d”, &s1.regno);
   scanf(“%s”, s1.name);
   scanf(“%s”, s1.dept);
   scanf(“%d”, &s1.year);
   ```

   OR

   ```
   scanf(“%d%s%s%d”, &s1.regno, s1.name, s1.dept, &s1.year);
   ```

6. Arrays in the Structure:

   The derived data types like array can be included in the structure as a member.

   Example:

   ```
   struct student
   {
       int roll;
       char name[20];
       int marks[5];
       int total;
       float avg;
       char result[5];
   }
   ```

   In memory it would be stored as given below:

   ```
   stu (size is 43 bytes)
   ```

   ```
   name array (size – 20 bytes)
   mark array (size – 10 bytes)
   result (size – 5 bytes)
   ```

   ```
   2190 2192 2212 2214 2216 2218 2220 2222 2224 2228
   ```

   ```
   0 1 2 3 4
   ```

   ```
   total avg result
   ```

   ```
   To access this location we need to use, stu.mark[3]
   ```
7. **Nested Structures:**

A structure variable may be defined as a member of another structure. In such situations, the declaration of the embedded structure must appear before the declaration of the outer structure.

*Example:*

```c
struct date
{
    int day;
    int month;
    int year;
};

struct bill
{
    int cno;
    char name[20];
    float amt;
    struct date billdate;
    struct date paydate;
};
```

The second structure bill now contains another structure, date, as one of its members. The structure may look like as follows:

<table>
<thead>
<tr>
<th>b1 (size of the variable is 38 bytes)</th>
<th>billdate (size – 6 bytes)</th>
<th>paydate (size – 6 bytes)</th>
</tr>
</thead>
<tbody>
<tr>
<td>2190 2192 2212 2216 2218 2220 2222 2224 2226</td>
<td></td>
<td></td>
</tr>
<tr>
<td>b1.cno</td>
<td>billdate.day</td>
<td>paydate.year</td>
</tr>
<tr>
<td>This can be accessed by b1.cno</td>
<td>This can be accessed by b1.billdate.day</td>
<td>This can be accessed by b1.paydate.year</td>
</tr>
</tbody>
</table>

8. **Processing Structures:**

Consider the following structure:

```c
struct student
{
    int regno;
    char name[20];
    char dept[10];
    struct date
    {
        int day;
        int month;
    }
};
```
The members of a structure are usually processed individually, as separate entities. Therefore, we must be able to access the individual structure members. A structure member can be accessed by writing

```
structure_variable.member
```

where variable refers to the name of a structure-type variable, and member refers to the name of a member within the structure. The period (.) separates the variable name from the member name. It is a member of the highest precedence group, and its associativity is left to right.

**Example:**
`s1.regno, s1.name, s1.dept, s1.year`

A nested structure member can be accessed by writing

```
structure_variable.member.submember;
```

**Example:**
`s1.bday.day, s1.bday.month, s1.bday.year`

where member refers to the name of the member within the outer structure, and submember refers to the name of the member within the embedded structure. Similarly, if a structure is an array, then an individual array element can be accessed by writing

```
structure-variable.member[expression];
```

**Example:**
`s1.mark[0], s1.mark[1], s1.mark[2], s1.mark[3], s1.mark[4]`

10. **Pointers to Structures:**

The address of a given structure variable can be obtained by using the & operator. Pointers to structures, like all other pointer variables may be assigned addresses. The following statements illustrate this concept.

**Example:**
```c
struct student
{
    int regno;
    char name[20];
    char dept[10];
    int year;
};
```
struct student stu, *sptr;
sptr = &stu;

Access to members of the structure is shown below:

printf("Student Registration Number : %d\n", sptr->regno);
printf("Student Name : %s\n", sptr->name);
printf("Department Name : %s\n", sptr->dept);
printf("Year of Study : %d\n", sptr->year);
STACK:

“A stack is an ordered list in which all insertions and deletions are made at one end, called the top”. Stacks are sometimes referred to as Last In First Out (LIFO) lists

Stacks have some useful terminology associated with them:

- **Push**: To add an element to the stack
- **Pop**: To remove an element from the stack
- **Peek**: To look at elements in the stack without removing them
- **LIFO**: Refers to the last in, first out behavior of the stack
- **FILO**: Equivalent to LIFO

Simple representation of a stack

Given a stack $S=(a[1], a[2], \ldots, a[n])$ then we say that $a_1$ is the bottom most element and element $a[i]$ is on top of element $a[i-1]$, $1 < i \leq n$.

Implementation of stack:

1. Array (static memory).
2. Linked list (dynamic memory)

The operations of stack is

1. PUSH operations
2. POP operations
3. PEEK operations
The Stack ADT

A stack S is an abstract data type (ADT) supporting the following three methods:

- push(n) : Inserts the item \( n \) at the top of stack
- pop() : Removes the top element from the stack and returns that top element. An error occurs if the stack is empty.
- peek() : Returns the top element and an error occurs if the stack is empty.

1. Adding an element into a stack. (called PUSH operations)

Adding element into the TOP of the stack is called PUSH operation.

Check conditions:

\( \text{TOP} = N \), then STACK FULL

where \( N \) is maximum size of the stack.

Adding into stack (PUSH algorithm)

```
procedure add(item : items);
{add item to the global stack stack; top is the current top of stack
and n is its maximum size}
begin
  if top = n then stackfull;
  top := top+1;
  stack(top) := item;
end: {of add}
```
Implementation in C using array:
/* here, the variables stack, top and size are global variables */
void push (int item)
{
    if (top == size-1)
        printf(“Stack is Overflow”);
    else
    {
        top = top + 1;
        stack[top] = item;
    }
}

3. Deleting an element from a stack. (called POP operations)
Deleting or Removing element from the TOP of the stack is called POP operations.

Check Condition:

TOP = 0, then STACK EMPTY

Deletion in stack (POP Operation)

procedure delete(var item : items);
{remove top element from the stack stack and put it in the item}
begin
    if top = 0 then stackempty;
    item := stack(top);
    top := top-1;
end; {of delete}
/* here, the variables stack, and top are global variables */
int pop ()
{
    if (top == -1)
    {
        printf("Stack is Underflow");
        return (0);
    }
    else
    {
        return (stack[top--]);
    }
}

3. Peek Operation:

✓ Returns the item at the top of the stack but does not delete it.
✓ This can also result in underflow if the stack is empty.

**Algorithm:**
PEEK(STACK, TOP)
BEGIN
    /* Check, Stack is empty? */
    if (TOP == -1) then
        print "Underflow" and return 0.
    else
        item = STACK[TOP]/ * stores the top element into a local variable */
        return item        / * returns the top element to the user */
END

*Implementation in C using array:*
/* here, the variables stack, and top are global variables */

int pop()
{
    if (top == -1)
    {
        printf("Stack is Underflow");
        return (0);
    }
    else
    {
        return (stack[top]);
    }
}

Applications of Stack
1. It is very useful to evaluate arithmetic expressions. (Postfix Expressions)
2. Infix to Postfix Transformation
3. It is useful during the execution of recursive programs
4. A Stack is useful for designing the compiler in operating system to store local
   variables inside a function block.
5. A stack (memory stack) can be used in function calls including recursion.
6. Reversing Data
7. Reverse a List
8. Convert Decimal to Binary
9. Parsing – It is a logic that breaks into independent pieces for further processing
10. Backtracking

Note:
1. Infix notation A+(B*C) equivalent Postfix notation ABC++
2. Infix notation (A+B)*C Equivalent Postfix notation AB+C*
Expression evaluation and syntax parsing

Calculators employing reverse Polish notation (also known as postfix notation) use a stack structure to hold values.

Expressions can be represented in prefix, postfix or infix notations. Conversion from one form of the expression to another form needs a stack. Many compilers use a stack for parsing the syntax of expressions, program blocks etc. before translating into low level code. Most of the programming languages are context-free languages allowing them to be parsed with stack based machines. Note that natural languages are context sensitive languages and stacks alone are not enough to interpret their meaning.

Infix, Prefix and Postfix Notation

We are accustomed to write arithmetic expressions with the operation between the two operands: \( a+b \) or \( c/d \). If we write \( a+b*c \), however, we have to apply precedence rules to avoid the ambiguous evaluation (add first or multiply first?).

There's no real reason to put the operation between the variables or values. They can just as well precede or follow the operands. You should note the advantage of prefix and postfix: the need for precedence rules and parentheses are eliminated.

<table>
<thead>
<tr>
<th>Infix</th>
<th>Prefix</th>
<th>Postfix</th>
</tr>
</thead>
<tbody>
<tr>
<td>a + b</td>
<td>+ a b</td>
<td>a b +</td>
</tr>
<tr>
<td>a + b * c</td>
<td>+ a * b c</td>
<td>a b c * +</td>
</tr>
<tr>
<td>(a + b) * (c - d)</td>
<td>* + a b - c d</td>
<td>a b + c d - *</td>
</tr>
<tr>
<td>b * b - 4 * a * c</td>
<td></td>
<td></td>
</tr>
<tr>
<td>40 - 3 * 5 + 1</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Examples of use: (application of stack)

Arithmetic Expressions: Polish Notation

- An arithmetic expression will have operands and operators.
- Operator precedence listed below:
  
<table>
<thead>
<tr>
<th>Level</th>
<th>Operators</th>
</tr>
</thead>
<tbody>
<tr>
<td>Highest</td>
<td>($)</td>
</tr>
<tr>
<td>Next Highest</td>
<td>(*) and (/)</td>
</tr>
<tr>
<td>Lowest</td>
<td>(+) and (-)</td>
</tr>
</tbody>
</table>

- For most common arithmetic operations, the operator symbol is placed in between its two operands. This is called **infix notation**.
  
  *Example: A + B, E * F*

- Parentheses can be used to group the operations.
  
  *Example: (A + B) * C*

- Accordingly, the order of the operators and operands in an arithmetic expression does not uniquely determine the order in which the operations are to be performed.

- Polish notation refers to the notation in which the operator symbol is placed before its two operands. This is called **prefix notation**.
  
  *Example: +AB, *EF*

- The fundamental property of polish notation is that the order in which the operations are to be performed is completely determined by the positions of the operators and operands in the expression.

- Accordingly, one never needs parentheses when writing expressions in Polish notation.

- **Reverse Polish Notation** refers to the analogous notation in which the operator symbol is placed after its two operands. This is called **postfix notation**.
  
  *Example: AB+, EF* 

- Here also the parentheses are not needed to determine the order of the operations.

- *The computer usually evaluates an arithmetic expression written in infix notation in two steps,*
  
  1. It converts the expression to **postfix notation**.
  2. It evaluates the postfix expression.

- In each step, the stack is the main tool that is used to accomplish the given task.
(1) Question: (Postfix evaluation)

How to evaluate a mathematical expression using a stack? The algorithm for evaluating a postfix expression?

- Initialise an empty stack
- While token remain in the input stream
  - Read next token
  - If token is a number, push it into the stack
  - Else, if token is an operator, pop top two tokens off the stack, apply the operator, and push the answer back into the stack
- Pop the answer off the stack.

Algorithm postfixexpression

Initialize a stack, opndstk to be empty.

{scan the input string reading one element at a time into symb}
While (not end of input string)
{Symb := next input character;
  If symb is an operand Then
    push (opndstk, symb)
  Else  
    [symbol is an operator]
    {Opnd1 := pop (opndstk);
     Opnd2 := pop (opndstk);
     Value := result of applying symb to opnd1 & opnd2
     Push(opndstk, value);
    }
  Result := pop (opndstk);
Example:

\[
6 \quad 2 \quad 3 + \quad 3 \quad 8 \quad 2 / \quad + \quad 2 \quad 3 +
\]

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Operand 1 (A)</th>
<th>Operand 2 (B)</th>
<th>Value (A ⊙ B)</th>
<th>STACK</th>
</tr>
</thead>
<tbody>
<tr>
<td>6</td>
<td></td>
<td></td>
<td></td>
<td>6</td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td>6, 2</td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
<td></td>
<td>6, 2, 3</td>
</tr>
<tr>
<td>+</td>
<td>2</td>
<td>3</td>
<td>5</td>
<td>6, 5</td>
</tr>
<tr>
<td>-</td>
<td>6</td>
<td>5</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
<td></td>
<td>1, 3</td>
</tr>
<tr>
<td>8</td>
<td></td>
<td></td>
<td></td>
<td>1, 3, 8</td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td>1, 3, 8, 2</td>
</tr>
<tr>
<td>/</td>
<td>8</td>
<td>2</td>
<td>/</td>
<td>1, 3, 4</td>
</tr>
<tr>
<td>+</td>
<td>3</td>
<td>4</td>
<td>7</td>
<td>1, 7</td>
</tr>
<tr>
<td>*</td>
<td>1</td>
<td>7</td>
<td>7</td>
<td>7</td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td>7, 2</td>
</tr>
<tr>
<td>$</td>
<td>7</td>
<td>2</td>
<td>49</td>
<td>49</td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
<td></td>
<td>49, 3</td>
</tr>
<tr>
<td>+</td>
<td>49</td>
<td>3</td>
<td>52</td>
<td>52</td>
</tr>
</tbody>
</table>

The Final value in the STACK is 52. This is the answer for the given expression.

(2) run time stack for function calls (write factorial number calculation procedure)
push local data and return address onto stack
return by popping off local data and then popping off address and returning to it
return value can be pushed onto stack before returning, popped off by caller

(3) expression parsing
e.g. matching brackets: [ ... ( ... ( ... ) [ ...( ... ) ...] ... ) ... ]
push left ones, pop off and compare with right ones

4) INFIX TO POSTFIX CONVERSION

Infix expressions are often translated into postfix form in which the operators appear after their operands. **Steps:**

1. Initialize an empty stack.
2. Scan the Infix Expression from left to right.
3. If the scanned character is an operand, add it to the Postfix Expression.
4. If the scanned character is an operator and if the stack is empty, then push the character to stack.
5. If the scanned character is an operator and the stack is not empty, Then

(a) Compare the precedence of the character with the operator on the top of the stack.
(b) While operator at top of stack has higher precedence over the scanned character & stack is not empty.
   (i) POP the stack.
   (ii) Add the Popped character to Postfix String.
(c) Push the scanned character to stack.

6. Repeat the steps 3-5 till all the characters
7. While stack is not empty,
   (a) Add operator in top of stack
   (b) Pop the stack.
8. Return the Postfix string.

Algorithm Infix to Postfix conversion (without parenthesis)

1. Opstk = the empty stack;
2. while (not end of input)
   {
      symb = next input character;
      
   }
3. if (symb is an operand)
   add symb to the Postfix String
4. else
   {
      
      While(! empty (opstk) && prec (stacktop (opstk), symb))
      {
         topsymb = pop (opstk)
         add topsymb to the Postfix String;
      } /* end of while */
      Push(opstk, symb);
   } /* end else */
6. /* end while */
7. While(! empty (opstk))
   {
      topsymb = pop (opstk)
      add topsymb to the Postfix String
   } /* end of while */
8. Return the Postfix String.
QUEUE:

“A queue is an ordered list in which all insertions at one end called REAR and deletions are made at another end called FRONT.” queues are sometimes referred to as First In First Out (FIFO) lists.

Example
1. The people waiting in line at a bank cash counter form a queue.
2. In computer, the jobs waiting in line to use the processor for execution. This queue is called Job Queue.

Operations Of Queue

There are two basic queue operations. They are,
Enqueue – Inserts an item / element at the rear end of the queue. An error occurs if the queue is full.
Dequeue – Removes an item / element from the front end of the queue, and returns it to the user. An error occurs if the queue is empty.

1. Addition into a queue

procedure addq (item : items);
{add item to the queue q}
begin
    if rear=n then queuefull
    else begin
        rear := rear+1;
        q[rear]:=item;
    end;
end;
{of addq}
2. Deletion in a queue

```pascal
procedure deleteq (var item : items);
{delete from the front of q and put into item}
begin
  if front = rear then queueempty
  else begin
    front := front+1
    item := q[front];
  end;
end
```

Uses of Queues (Application of queue)

Queues remember things in first-in-first-out (FIFO) order. Good for fair (first come first served) ordering of actions.

Examples of use: (Application of stack)

1. scheduling
   - processing of GUI events
   - printing request

2. simulation
   - orders the events
   - models real life queues (e.g. supermarkets checkout, phone calls on hold)

Circular Queue:

Location of queue are viewed in a circular form. The first location is viewed after the last one.
Overflow occurs when all the locations are filled.
Algorithm Circular Queue Insert

Void CQInsert ( int queue[ ], front, rear, item)
{
    if ( front == 0 )
        front = front +1;
    if ( ( ( rear == maxsize ) && ( front == 1 ) ) || ( ( rear ! = 0 ) && ( front == rear +1)))
    {
        printf( “ queue overflow “);
    }
    if( rear == maxsize )
        rear = 1;
    else
        rear = rear + 1;
    q [ rear ] = item;
}

Algorithm Circular Queue Delete

int CQDelete ( queue[ ], front, rear )
{
    if ( front == 0 )
        printf( “ queue underflow “);
    else
    {
        item = queue [ front ];
        if(front == rear )
        {
            front = 0; rear = 0;
        }
        else if ( front == maxsize )
        {
            front = 1;
        }
        else
        {
            front = front + 1;
        }
        return item;
    }
**Priority Queue**

A priority queue is a collection of elements such that each element has been assigned a priority and such that the order in which elements are deleted and processed comes from the following rules:

1. An element of higher priority is processed before any element of lower priority.
2. Two elements with the same priority are processed according to the order in which they were added to the queue.

Two types of queue are

1. **Ascending Priority Queue**
   Collection of items into which item can be inserted arbitrarily & from which only the Smallest item can be removed.

2. **Descending Priority Queue**
   Collection of items into which item can be inserted arbitrarily & from which only the Largest item can be removed.

**Double Ended Queue**

A deque (short for double-ended queue) is an abstract data structure for which elements can be added to or removed from the front or back(both end). This differs from a normal queue, where elements can only be added to one end and removed from the other. Both queues and stacks can be considered specializations of deques, and can be implemented using deques.
Two types of Dqueue are

1. Input Restricted Dqueue
2. Output Restricted Dqueue.

1. Input Restricted Dqueue

Where the input (insertion) is restricted to the rear end and the deletions has the options either end

2. Output Restricted Dqueue.

Where the output (deletion) is restricted to the front end and the insertions has the option either end.

Example: Timesharing system using the prototype of priority queue – programs of high priority are processed first and programs with the same priority form a standard queue.
Linked List

- Some demerits of array, leads us to use linked list to store the list of items. They are,
  1. It is relatively expensive to insert and delete elements in an array.
  2. Array usually occupies a block of memory space, one cannot simply double or triple the size of an array when additional space is required. *(For this reason, arrays are called “dense lists” and are said to be “static” data structures.)*

- A *linked list*, or *one-way list*, is a linear collection of data elements, called *nodes*, where the linear order is given by means of *pointers*. That is, each node is divided into two parts:
  - The first part contains the information of the element i.e. INFO or DATA.
  - The second part contains the *link field*, which contains the address of the next node in the list.

- The linked list consists of series of nodes, which are not necessarily adjacent in memory.
- A list is a *dynamic data structure* i.e. the number of nodes on a list may vary dramatically as elements are inserted and removed.
- The pointer of the last node contains a special value, called the *null pointer*, which is any invalid address. This *null pointer* signals the end of list.
- The list with no nodes on it is called the *empty list* or *null list*.

*Example:* The linked list with 4 nodes.

*Types of Linked Lists:*

- a) Linear Singly Linked List
- b) Circular Linked List
- c) Two-way or doubly linked lists
- d) Circular doubly linked lists
Advantages of Linked List
1. Linked List is dynamic data structure; the size of a list can grow or shrink during the program execution. So, maximum size need not be known in advance.
2. The Linked List does not waste memory
3. It is not necessary to specify the size of the list, as in the case of arrays.
4. Linked List provides the flexibility in allowing the items to be rearranged.

What are the pitfalls encountered in single linked list?
1. A singly linked list allows traversal of the list in only one direction.
2. Deleting a node from a list requires keeping track of the previous node, that is, the node whose link points to the node to be deleted.
3. If the link in any node gets corrupted, the remaining nodes of the list become unusable.

Linearly-linked List

Is a collection of elements called Nodes. Each node consist of two fields, namely data field to hold the values and link(next) field points to the next node in the list.

It consists of a sequence of nodes, each containing arbitrary data fields and one or two references ("links") pointing to the next and/or previous nodes.

A linked list is a self-referential datatype (or) data structure because it contains a pointer or link to another data of the same type.

Linked lists permit insertion and removal of nodes at any point in the list in constant time, but do not allow random access.

Several different types of linked list exist: singly-linked lists, doubly-linked lists, and circularly-linked lists. One of the biggest advantages of linked lists is that nodes may have multiple pointers to other nodes, allowing the same nodes to simultaneously appear in different orders in several linked lists.

Singly-linked list

The simplest kind of linked list is a singly-linked list (or slist for short), which has one link per node. This link points to the next node in the list, or to a null value or empty list if it is the final node.

A singly linked list containing three integer values
Doubly-linked list

A more sophisticated kind of linked list is a **doubly-linked list** or **two-way linked list**. Each node has two links: one points to the previous node, or points to a **null** value or empty list if it is the first node; and one points to the next, or points to a **null** value or empty list if it is the final node.

![An example of a doubly linked list.](image)

Circularly-linked list

In a **circularly-linked list**, the first and final nodes are linked together. This can be done for both singly and doubly linked lists. To traverse a circular linked list, you begin at any node and follow the list in either direction until you return to the original node. Viewed another way, circularly-linked lists can be seen as having no beginning or end. This type of list is most useful for managing buffers for data ingest, and in cases where you have one object in a list and wish to see all other objects in the list.

The pointer pointing to the whole list is usually called the end pointer.

Singly-circularly-linked list

In a **singly-circularly-linked list**, each node has one link, similar to an ordinary **singly-linked list**, except that the next link of the last node points back to the first node.

As in a singly-linked list, new nodes can only be efficiently inserted after a node we already have a reference to. For this reason, it's usual to retain a reference to only the last element in a singly-circularly-linked list, as this allows quick insertion at the beginning, and also allows access to the first node through the last node's next pointer.
• Note that there is no NULL terminating pointer
• Choice of head node is arbitrary
• A tail pointer serves no purpose
• What purpose(s) does the head pointer serve?

Doubly-circularly-linked list

In a doubly-circularly-linked list, each node has two links, similar to a doubly-linked list, except that the previous link of the first node points to the last node and the next link of the last node points to the first node. As in doubly-linked lists, insertions and removals can be done at any point with access to any nearby node.

Sentinel nodes

Linked lists sometimes have a special dummy or sentinel node at the beginning and/or at the end of the list, which is not used to store data.

Basic Operations on Linked Lists

1. Insertion
   a. At first
   b. At last
   c. At a given location (At middle)
2. Deletion
   a. First Node
   b. Last Node
   c. Node in given location or having given data item

Initial Condition

HEAD = NULL;
/* Address of the first node in the list is stored in HEAD. Initially there is no node in the list. So, HEAD is initialized to NULL (No address) */

What are the Applications of linked list?
   • To implement of Stack, Queue, Tree, Graph etc.,
   • Used by the Memory Manager
   • To maintain Free-Storage List
Doubly Linked Lists (or) Two – Way Lists

There are some problems in using the Single linked list. They are

1. A singly linked list allows traversal of the list in only one direction. (Forward only)
2. Deleting a node from a list requires keeping track of the previous node, that is, the
   node whose link points to the node to be deleted.

These major drawbacks can be avoided by using the double linked list. The doubly
linked list is a linear collection of data elements, called nodes, where each node is
divided into three parts. They are:

1. A pointer field LEFT which contains the address of the preceding node in the list
2. An information field INFO which contains the data of the Node
3. A pointer field RIGHT which contains the address of the next node in the list

Example:

```
LEFT INFO RIGHT

head 7060 NULL 7 2140 7060 7 4020 7 2140 4020 end
```

Linked lists vs. arrays

<table>
<thead>
<tr>
<th>Array</th>
<th>Linked list</th>
</tr>
</thead>
<tbody>
<tr>
<td>Indexing</td>
<td>O(1) O(n)</td>
</tr>
<tr>
<td>Inserting / Deleting at end</td>
<td>O(1) O(1)</td>
</tr>
<tr>
<td>Inserting / Deleting in middle (with iterator)</td>
<td>O(n) O(1)</td>
</tr>
<tr>
<td>Persistent</td>
<td>No Singly yes</td>
</tr>
<tr>
<td>Locality</td>
<td>Great Bad</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Array</th>
<th>Linked list</th>
</tr>
</thead>
<tbody>
<tr>
<td>Static memory</td>
<td>Dynamic memory</td>
</tr>
<tr>
<td>Insertion and deletion required to modify the existing element location</td>
<td>Insertion and deletion are made easy.</td>
</tr>
<tr>
<td>Elements stored as contiguous memory as on block.</td>
<td>Element stored as Non-contiguous memory as pointers</td>
</tr>
<tr>
<td>Accessing element is fast</td>
<td>Accessing element is slow</td>
</tr>
</tbody>
</table>
SINGLY LINKED LISTS

1. Insertion of a Node in the Beginning of a List

Step 1: Allocate memory for a node and assign its address to the variable ‘New’
Step 2: Assign the element in the data field of the new node.
Step 3: Make the next field of the new node as the beginning of the existing list.
Step 4: Make the new node as the Head of the list after insertion.

Algorithm InsertBegin (Head, Elt)
[Adding the element elt in the beginning of the list pointed by Head]

1. new ← getnode (NODE)
2. data (new) ← elt
3. next (new) ← Head
4. Head ← new
5. return Head
Insertion of a Node at the End of a Singly Linked List

Step 1: Allocate memory for a node and assign its address to the variable ‘New’
Step 2: Assign the element in the data field of the new node.
Step 3: Make the next field of the new node as NULL. This is because the new node will be the end of the resultant list.
Step 4: If the existing list is empty, call this new node as the list. Else, get the address of the last node in the list by traversing from the beginning pointer.
Step 5: Make the next field of the last node point to the new node.

Algorithm InsertEnd (Head, Elt)
[Adding the element Elt at the end of the list]

1. new ← getnode (NODE)
2. data (new) ← elt
3. next (new) ← NULL
4. if (Head == NULL) Then
   Head ← new
   Return Head
Else
   Temp ← Head
5. While (next (temp) # NULL)
   temp ← next (temp)
6. next (temp) ← new
7. return Head.
Applications of linked lists

Linked lists are used as a building block for many other data structures, such as stacks, queues and their variations.

1. Polynomial ADT:

A polynomial can be represented with primitive data structures. For example, a polynomial represented as \( a_k x^k + a_{k-1} x^{k-1} + \ldots + a_0 \) can be represented as a linked list. Each node is a structure with two values: \( a_i \) and \( i \). Thus, the length of the list will be \( k \). The first node will have \((a_k, k)\), the second node will have \((a_{k-1}, k-1)\) etc. The last node will be \((a_0, 0)\).

The polynomial \( 3x^9 + 7x^3 + 5 \) can be represented in a list as follows: \((3,9) \rightarrow (7,3) \rightarrow (5,0)\) where each pair of integers represent a node, and the arrow represents a link to its neighbouring node.

Derivatives of polynomials can be easily computed by proceeding node by node. In our previous example the list after computing the derivative would represented as follows: \((27,8) \rightarrow (21,2)\). The specific polynomial ADT will define various operations, such as multiplication, addition, subtraction, derivative, integration etc. A polynomial ADT can be useful for symbolic computation as well.

2. Large Integer ADT:

Large integers can also be implemented with primitive data structures. To conform to our previous example, consider a large integer represented as a linked list. If we represent the integer as successive powers of 10, where the power of 10 increments by 3 and the coefficient is a three digit number, we can make computations on such numbers easier. For example, we can represent a very large number as follows:

\[ 513(10^6) + 899(10^3) + 722(10^0) \]

Using this notation, the number can be represented as follows:

\((513) \rightarrow (899) \rightarrow (722)\).

The first number represents the coefficient of the \(10^6\) term, the next number represents the coefficient of the \(10^3\) term and so on. The arrows represent links to adjacent nodes.

The specific ADT will define operations on this representation, such as addition, subtraction, multiplication, division, comparison, copy etc.

An array allocates memory for all its elements lumped together as one block of memory. In contrast, a linked list allocates space for each element separately in its own block of memory called a "linked list element" or "node". The list gets its 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 type the list holds for its client, and a "next" field which is a pointer used to link one node to the next node.

Each node is allocated in the heap with a call to malloc(), so the node memory continues to exist until it is explicitly deallocated with a call to free(). The front of the list is a pointer to the first node. Here is what a list containing the numbers 1, 2, and 3 might look like...

`malloc()`  
`malloc()` is a system function which allocates a block of memory in the "heap" and returns a pointer to the new block. The prototype for malloc() and other heap functions are in `stdlib.h`. The argument to malloc() is the integer size of the block in bytes. Unlike local ("stack") variables, heap memory is not automatically deallocated when the creating function exits. malloc() returns NULL if it cannot fulfill the request. (extra for experts) You may check for the NULL case with assert() if you wish just to be safe. Most modern programming systems will throw an exception or do some other automatic error handling in their memory allocator, so it is becoming less common that source code needs to explicitly check for allocation failures.

`free()`  
`free()` is the opposite of malloc(). Call free() on a block of heap memory to indicate to the system that you are done with it. The argument to free() is a pointer to a block of memory in the heap — a pointer which some time earlier was obtained via a call to malloc().

**Sequential list:** contiguous cells, indexed by location

**Linked list:** noncontiguous cells linked by pointers, implicitly indexed by number of links away from head

Both also contain
location of first element (head)  
length or end-of-list marker  
Examples of end-of-list marker:  
```
\0
```

for C strings (sequential list)  
`NULL` for C linked list
**Representation**

**Sequential list:** contiguous cells, indexed by location **Linked list:** noncontiguous cells linked by pointers, implicitly indexed by number of links away from head

Both also contain

- location of first element (head)
- length or end-of-list marker

Examples of end-of-list marker:

- `\0` for C strings (sequential list)
- `NULL` for C linked list

**Sequential List implementation**

In C: typically an array with an int for the last used index.

- A problem: must reserve memory for the list to grow into.
- limits length of list to reserved length
- reserved memory unusable for other purposes

Main **advantage** of sequential list: fast access to element by index.

**Linked List implementation**

In C: typically individual cells dynamically allocated containing a pointer to the next cell.

**Advantages:**

- space used adapts to size • usually results in better space usage than sequential despite storing a pointer in each cell

- speed improvements for some operations

**Disadvantages:**

- speed reductions for some operations

**Time efficiency 1**

In the following, let n be the length of the list.

**Initialize to empty list**

**Sequential and Linked O(1)**

For sequential lists this really depends on the complexity of memory allocation, a complex subject in itself. For linked lists, memory allocation time can affect the performance of the other operations.

**Select ith element**
Sequential O(1) Linked O(i)

Time efficiency 2
Determine length
Sequential and Linked
O(1) if recorded
O(n) for marker – then linked takes longer following pointers and is more likely to leave the cache

Traverse
Sequential and Linked O(n)
Linked takes longer for reasons above, but may be insignificant compared to processing done on the elements.

Search
Sequential ordered O(log n)
Sequential unordered, Linked O(n)
Linked takes longer for reasons above. Ordering can improve linked on average, since we can more quickly detect that an element isn’t in the list.

Time efficiency 3
Changes
These depend of course on how we locate where to make the change. The following describe the additional cost.

Delete or insert element
Sequential O(n * size of an element) We must move elements over! Linked O(1)
Replace element (unordered)
Sequential and Linked O(1)
Replace data in an element
Sequential and Linked O(1)

Variants
There are other list implementations. For example:

Doubly Linked list
allows efficient backwards traversal
takes longer to insert and delete (but the same complexity)
takes more space for the extra pointer (unless we use the for trick to save space at the cost of time)
Circular list (head and tail linked)
Two Marks

1. Limitations of arrays
   e) Arrays have a fixed dimension. Once the size of an array is decided it cannot be increased or decreased during execution.
   f) Array elements are always stored in contiguous memory locations. So it need contiguous locations otherwise memory will not be allocated to the arrays.
   g) Operations like insertion of a new element in an array or deletion of an existing element from the array are pretty tedious. This is because during insertion or deletion each element after the specified position has to be shifted one position to the right or one position to the left.

2. Define Data Structure.

   A data structure is a way of organizing data that considers not only the items stored, but also their relationship to each other. Advance knowledge about the relationship between data items allows designing of efficient algorithms for the manipulation of data.

3. Why do we need data structures?
   - Data structures allow us to achieve an important goal: component reuse.
   - Once each data structure has been implemented once, it can be used over and over again in various applications.

4. Simple Classification of Data Structure.
   The data structure can be classified into two types as follows:
   a) Linear Data Structures – All the elements are formed in a sequence or maintain a linear ordering
      i. Arrays
      ii. Linked Lists
      iii. Stacks
      iv. Queues
   b) Non-Linear Data Structures – All the elements are distributed on a plane i.e. these have no such sequence of elements as in case of linear data structure.
      i. Trees
      ii. Graphs
      iii. Sets

5. List the operations performed in the Linear Data Structure
   a) Traversal – Processing each element in the list
b) Search – Finding the location of the element with a given value.
c) Insertion / Storing – Adding a new element to the list.
d) Deletion – Removing an element from the list.
e) Sorting – Arranging the elements in some type of order.
f) Merging – Combining two lists into a single list.

6. Explain Linked List
   • A linked list is a list of elements in which the elements of the list can be placed anywhere in memory, and these elements are linked with each other using an explicit link field, that is by storing the address of the next element in the link field of the previous element.
   • A linked list is a self-referential data type because it contains a pointer or link to another data of the same type. This permit insertion and removal of nodes at any point in the list in constant time, but do not allow random access.

7. What is a node?
   Each element structure in a slinked list called node, containing two fields one is data and another is address of next node.

8. Advantages of Linked List
   5. Linked List is dynamic data structure; the size of a list can grow or shrink during the program execution. So, maximum size need not be known in advance.
   6. The Linked List does not waste memory
   7. It is not necessary to specify the size of the list, as in the case of arrays.
   8. Linked List provides the flexibility in allowing the items to be rearranged.

9. What are the pitfalls encountered in single linked list?
   4. A singly linked list allows traversal of the list in only one direction.
   5. Deleting a node from a list requires keeping track of the previous node, that is, the node whose link points to the node to be deleted.
   6. If the link in any node gets corrupted, the remaining nodes of the list become unusable.

10. Define Stack
Stack is a linear data structure and is an ordered collection of homogeneous data elements, where the insertion and deletion operations take place at one end called top of the stack.

A stack data structure exhibits the LIFO (Last In First Out) property.

11. What are operations allowed in a Stack?
   1. PUSH : This operation is used to add an item into the top of the stack.
   2. POP : This operation is used to remove an item from the top of the stack.
   3. PEEK : This operation is used to display the top item in the stack.

12. List the notations used to represent the arithmetic expressions.
   1. Infix: <operand> operator <operand>  
      Ex: $A + B$
   2. Prefix: operator <operand> <operand>  
      (also called as polish notation) Ex: $+AB$
   3. Postfix: <operand> <operand> operator  
      Ex: $AB+$

13. Rules for converting an Infix notation to postfix form
   1. Assume, the fully parenthesized version of the Infix expression
   2. Move all operator, so that they replace their corresponding right part of parenthesis
   3. Remove all parenthesis
      Example: $((A+((B^C)-D))*(E-(A/C)))$ $\rightarrow$ $ABC^D+-EAC/-*$

14. Define Queue
    Queue is an ordered collection of homogeneous data elements, in which the element insertion and deletion takes place at two ends called front and rear. The elements are ordered in linear fashion and inserted at REAR end and deleted FRONT end. This exhibits FIFO (First In First Out) property.

15. Applications of Queue
    Applications of queue as a data structure are more common.
    a) Within a computer system there may be queues of tasks waiting for the line printer, or for access to disk storage, or in a time-sharing system for use of the CPU.
    b) Within a single program, there may be multiple requests to be kept in a queue, or one task may create other tasks, which must be done in turn by keeping them in a queue.

16. What is the need of Circular Queue?
Queue implemented using an array suffers from one limitation i.e. there is a possibility that the queue is reported as full (since rear has reached the end of the array), even though in actuality there might be empty slots at the beginning of the queue. To overcome this limitation circular queue is needed.

Now the queue would be reported as full only when all the slots in the array stand occupied.

17. What is deque?
   - The word deque is a short form of double-ended queue and defines a data structure in which items can be added or deleted at either the front or rear end, but no changes can be made elsewhere in the list.
   - Thus a deque is a generalization of both a stack and a queue.

18. Types of Linked Lists:
   a) Linear Singly Linked List
   b) Circular Linked List
   c) Two-way or doubly linked lists
   d) Circular doubly linked lists

19. What are the Applications of linked list?
   a) Implementation of Stack
   b) Implementation of Queue
   c) Implementation of Tree
   d) Implementation of Graph

20. Applications of Stack
   a) It is very useful to evaluate arithmetic expressions. (Postfix Expressions)
   b) Infix to Postfix Transformation
   c) It is useful during the execution of recursive programs
   d) A Stack is useful for designing the compiler in operating system to store local variables inside a function block.
   e) A stack can be used in function calls including recursion.
   f) Reversing Data
   g) Reverse a List
   h) Convert Decimal to Binary
   i) Parsing – It is a logic that breaks into independent pieces for further processing
   j) Backtracking
Reference:–

www.bcaraipur.blogspot.in