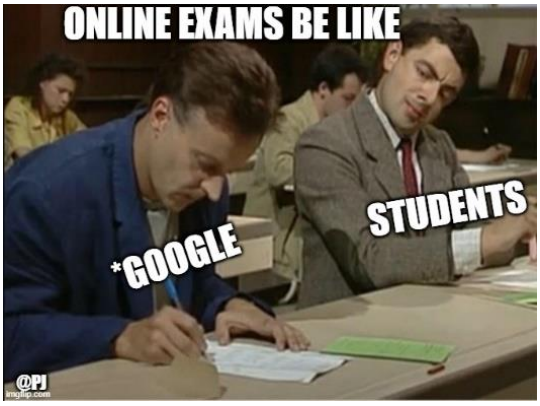
NOTES-CLICK

Data Structures and Algorithms



1 Define data structure. List different operations performed on Data Structures.

Overview of Data Structures

- **Data structure** is a representation of the logical relationship existing between individual elements of data.
- Data Structure is a way of organizing all data items that considers not only the elements stored but also their relationship to each other.
- We can also define data structure as a mathematical or logical model of a particular organization of data items.
- The representation of particular data structure in the main memory of a computer is called as **storage structure.**
- The storage structure representation in auxiliary memory is called as **file structure**.
- It is defined as the way of storing and manipulating data in organized form so that it can be used efficiently.
- Data Structure mainly specifies the following four things
    - Organization of Data
    - Accessing methods
    - Degree of associativity
    - Processing alternatives for information
- Algorithm + Data Structure = Program
- Data structure study covers the following points
    - Amount of memory require to store.
    - Amount of time require to process.
    - Representation of data in memory.
    - Operations performed on that data.

Operation on Data Structures

Design of efficient data structure must take operations to be performed on the data structures into account. The most commonly used operations on data structure are broadly categorized into following types

1. **Create:-** The create operation results in reserving memory for program elements. This can be done by declaration statement. Creation of data structure may take place either during compile-time or run-time. malloc() function of C language is used for creation.
2. **Destroy:-** Destroy operation destroys memory space allocated for specified data structure. free() function of C language is used to destroy data structure.
3. **Selection:-** Selection operation deals with accessing a particular data within a data structure.
4. **Updation:-** It updates or modifies the data in the data structure.
5. **Searching:-** It finds the presence of desired data item in the list of data items, it may also find the locations of all elements that satisfy certain conditions.
6. **Sorting:-** Sorting is a process of arranging all data items in a data structure in a particular order, say for example, either in ascending order or in descending order.
7. **Merging:-** Merging is a process of combining the data items of two different sorted list into a single sorted list.
8. **Splitting:-** Splitting is a process of partitioning single list to multiple list.
9. **Traversal:-** Traversal is a process of visiting each and every node of a list in systematic manner.

2 Define Algorithms, Differentiate between time complexity and spacecomplexity of an algorithm.

An algorithm is a step-by-step procedure that defines a set of instructions that must be carried out in a specific order to produce the desired result. Algorithms are generally developed independently of underlying languages, which means that an algorithm can be implemented in more than one programming language. Unambiguity, fineness, effectiveness, and language independence are some of the characteristics of an algorithm. The scalability and performance of an algorithm are the primary factors that contribute to its importance.

What is Time complexity?

Time complexity is defined as the amount of time taken by an algorithm to run, as a function of the length of the input. It measures the time taken to execute each statement of code in an algorithm. It is not going to examine the total execution time of an algorithm. Rather, it is going to give information about the variation (increase or decrease) in execution time when the number of operations (increase or decrease) in an algorithm. Yes, as the definition says, the amount of time taken is a function of the length of input only.

There are different types of time complexities used, let's see one by one:

1. Constant time – O (1)

2. Linear time – O (n)

3. Logarithmic time – O (log n)

4. Quadratic time – O (n^2)

5. Cubic time – O (n^3)

Space Complexity

You might have heard of this term, 'Space Complexity', that hovers around when talking about time complexity. What is Space Complexity? Well, it is the working space or storage that is required by any algorithm. It is directly dependent or proportional to the amount of input that the algorithm takes. To calculate space complexity, all you have to do is calculate the space taken up by the variables in an algorithm. The lesser space, the faster the algorithm executes. It is also important to know that time and space complexity are not related to each other.

3 Define: Sparse Matrices, how we store it in memory?

Sparse matrices are those matrices that have the majority of their elements equal to zero. In other words, the sparse matrix can be defined as the matrix that has a greater number of zero elements than the non-zero elements.
Storage - We know that a sparse matrix contains lesser non-zero elements than zero, so less memory can be used to store elements. It evaluates only the non-zero elements.

Array representation of the sparse matrix

Representing a sparse matrix by a 2D array leads to the wastage of lots of memory. This is because zeroes in the matrix are of no use, so storing zeroes with non-zero elements is wastage of memory. To avoid such wastage, we can store only non-zero elements. If we store only non-zero elements, it reduces the traversal time and the storage space.

In 2D array representation of sparse matrix, there are three fields used that are named as -

ROW	COL	VALUE

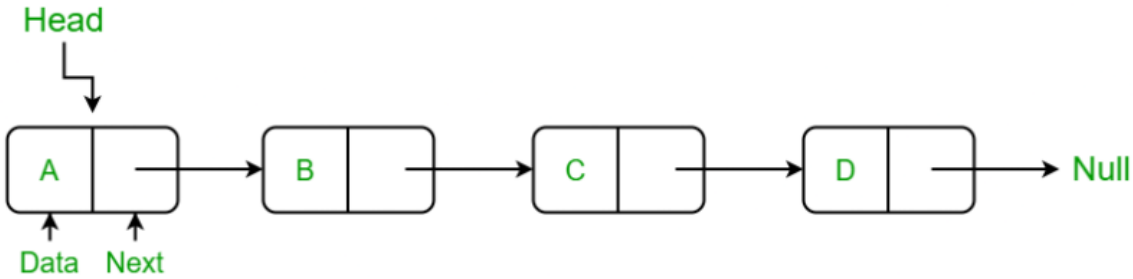
- **Row -** It is the index of a row where a non-zero element is located in the matrix.
- **Column -** It is the index of the column where a non-zero element is located in the matrix.
- **Value -** It is the value of the non-zero element that is located at the index (row, column)

4 Define: (a) Sparse Matrices

5 Explain linked lists. What are the disadvantages of arrays. How linked listovercome these disadvantages.

Linked List:

A Linked list is a dynamic arrangement that contains a "link" to the structure containing the subsequent items. It's a set of structures ordered not by their physical placement in memory (like an array) but by logical links that are stored as a part of the info within the structure itself.



Advantages Of Linked List:

- **Dynamic data structure:** A linked list is a dynamic arrangement so it can grow and shrink at runtime by allocating and deallocating memory. So there is no need to give the initial size of the linked list.
- **No memory wastage:** In the Linked list, efficient memory utilization can be achieved since the size of the linked list increase or decrease at run time so there is no memory wastage and there is no need to pre-allocate the memory.
- **Implementation:** Linear data structures like stacks and queues are often easily implemented using a linked list.
- **Insertion and Deletion Operations:** Insertion and deletion operations are quite easier in the linked list. There is no need to shift elements after the insertion or deletion of an element only the address present in the next pointer needs to be updated.

Disadvantages of array data structure:

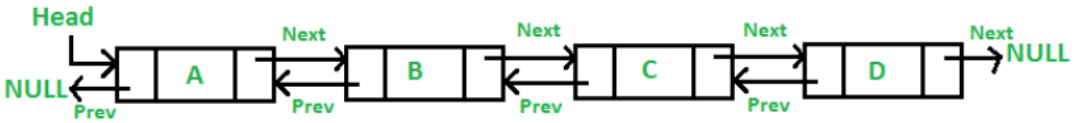
- The size of the array should be known in advance.
- The array is a static data structure with a fixed size so, the size of the array cannot be modified further and hence no modification can be done during runtime.
- Insertion and deletion operations are costly in arrays as elements are stored in contiguous memory.
- If the size of the declared array is more than the required size then, it can lead to memory wastage.

6 Explain (a) Generalized linked list, (b) Two way linked list and (c) Circular linked list.

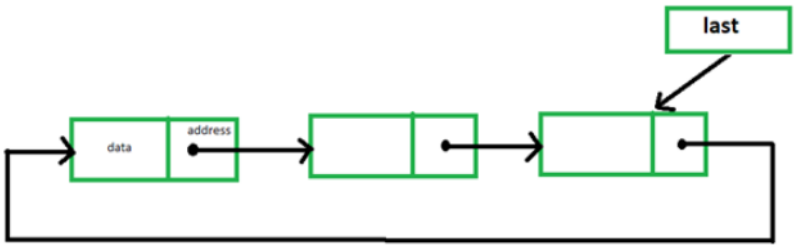
A Generalized Linked List L, is defined as a finite sequence of n>=0 elements, l_1, l_2, l_3, l_4, …, l_n, such that l_i are either atom or the list of atoms. Thus
$L = (l_1, l_2, l_3, l_4, \ldots, l_n)$
where n is total number of nodes in the list.

A Doubly Linked List (DLL) contains an extra pointer, typically called the previous pointer, together with the next pointer and data which are there in the singly linked list.

Prerequisites: Linked List Introduction, Inserting a node in Singly Linked List



In a Circular linked list, every element has a link to its next element in the sequence, and the last element has a link to the first element. A circular linked list is similar to the singly linked list except that the last node points to the first node. Below is the image to illustrate the same:



7 Define Stack in data structures. Explain push (), pop () operations. Write thepush () and pop () functions in any programming.

A Stack is a linear data structure that follows the **LIFO (Last-In-First-Out)** principle. Stack has one end, whereas the Queue has two ends (**front and rear**). It contains only one pointer **top pointer** pointing to the topmost element of the stack. Whenever an element is added in the stack, it is added on the top of the stack, and the element can be deleted only from the stack. In other words, a ***stack can be defined as a container in which insertion and deletion can be done from the one end known as the top of the stack.***

Some key points related to stack

- It is called as stack because it behaves like a real-world stack, piles of books, etc.
- A Stack is an abstract data type with a pre-defined capacity, which means that it can store the elements of a limited size.
- It is a data structure that follows some order to insert and delete the elements, and that order can be LIFO or FILO.
- **push():** When we insert an element in a stack then the operation is known as a push. If the stack is full then the overflow condition occurs.
- **pop():** When we delete an element from the stack, the operation is known as a pop. If the stack is empty means that no element exists in the stack, this state is known as an underflow state.

```
void Push()
{
        int x;

        if(Top==Size-1)
        {
                printf("\nOverflow!!");
        }
        else
        {
                printf("\nEnter element to be inserted to the stack:");
                scanf("%d",&x);
                Top=Top+1;
                inp_array[Top]=x;
        }
}

void Pop()
{
        if(Top==-1)
        {
                printf("\nUnderflow!!");
        }
        else
        {
                printf("\nPopped element:  %d",inp_array[Top]);
                Top=Top-1;
        }
}
```

8 What are the applications of stack?

Applications of Stack

The following are the applications of the stack:

- **Balancing of symbols:** Stack is used for balancing a symbol. For example, we have the following program:
- **String reversal:** Stack is also used for reversing a string. For example, we want to reverse a "**javaTpoint**" string, so we can achieve this with the help of a stack.
  First, we push all the characters of the string in a stack until we reach the null character. After pushing all the characters, we start taking out the character one by one until we reach the bottom of the stack.
- **UNDO/REDO:** It can also be used for performing UNDO/REDO operations. For example, we have an editor in which we write 'a', then 'b', and then 'c'; therefore, the text written in an editor is abc. So, there are three states, a, ab, and abc, which are stored in a stack. There would be two stacks in which one stack shows UNDO state, and the other shows REDO state.
  If we want to perform UNDO operation, and want to achieve 'ab' state, then we implement pop operation.
- **Recursion:** The recursion means that the function is calling itself again. To maintain the previous states, the compiler creates a system stack in which all the previous records of the function are maintained.
- **DFS(Depth First Search):** This search is implemented on a Graph, and Graph uses the stack data structure.
- **Backtracking:** Suppose we have to create a path to solve a maze problem. If we are moving in a particular path, and we realize that we come on the wrong way. In order to come at the beginning of the path to create a new path, we have to use the stack data structure.
- **Expression conversion:** Stack can also be used for expression conversion. This is one of the most important applications of stack. The list of the expression conversion is given below:
  - Infix to prefix
  - Infix to postfix
  - Prefix to infix
  - Prefix to postfix
    - Postfix to infix
- **Memory management:** The stack manages the memory. The memory is assigned in the contiguous memory blocks. The memory is known as stack memory as all the variables are assigned in a function call stack memory. The memory size assigned to the program is known to the compiler. When the function is created, all its variables are assigned in the stack memory. When the function completed its execution, all the variables assigned in the stack are released.

9 What are various asymptotic notations. Explain Big oh notation (O) Big Omega notation (Ω) Theta notation (Θ).

There are mainly three asymptotic notations:

- ***Big-O Notation (O-notation)***
- ***Omega Notation (Ω-notation)***
- ***Theta Notation (Θ-notation)***

1. Theta Notation (Θ-Notation):

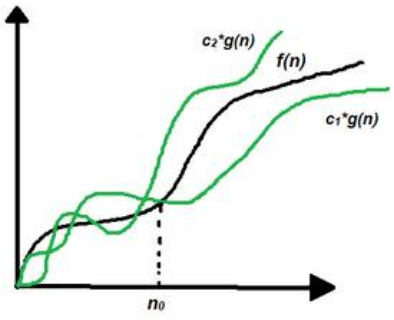
*Theta notation encloses the function from above and below. Since it represents the upper and the lower bound of the running time of an algorithm, it is used for analyzing the **average-case** complexity of an algorithm.*

2. Big-O Notation (O-notation):

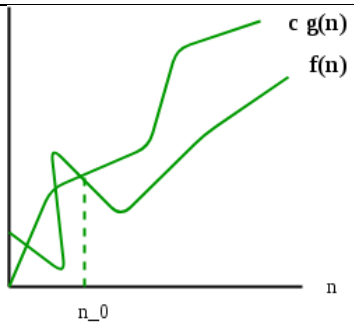
Big-O notation represents the upper bound of the running time of an algorithm. Therefore, it gives the worst-case complexity of an algorithm.

3. Omega Notation (Ω-Notation):

Omega notation represents the lower bound of the running time of an algorithm. Thus, it provides the best case complexity of an algorithm.

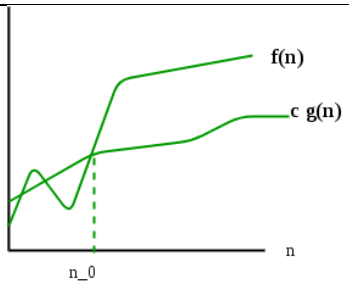


1



f(n) = O(g(n))

2



f(n) = Omega(g(n))

3

10 Write a program (a) to multiply two matrices, (b) to find the maximum andminimum elements in an array

```
void mulMat(int mat1[][C1], int mat2[][C2])
{
    int rslt[R1][C2];

    cout << "Multiplication of given two matrices is:\n";

    for (int i = 0; i < R1; i++) {
        for (int j = 0; j < C2; j++) {
            rslt[i][j] = 0;

            for (int k = 0; k < R2; k++) {
                rslt[i][j] += mat1[i][k] * mat2[k][j];
            }

            cout << rslt[i][j] << "\t";
        }

        cout << endl;
    }
}
```

full link for A and B

```
struct Pair{
    int min;
    int max;
};
Pair getMinMax(int arr[], int n){
    struct Pair minmax;
    int i;
    if (n == 1){
        minmax.max = arr[0];
        minmax.min = arr[0];
        return minmax;
    }
    if (arr[0] > arr[1]){
        minmax.max = arr[0];
        minmax.min = arr[1];
    }
    else{
        minmax.max = arr[1];
        minmax.min = arr[0];
    }
    for(i = 2; i < n; i++){
        if (arr[i] > minmax.max)
            minmax.max = arr[i];

        else if (arr[i] < minmax.min)
            minmax.min = arr[i];
    }
    return minmax;
}
```

11 Explain Infix, Prefix and Postfix Expressions.

1. Infix notation: X + Y

   Operators are written in-between their operands. This is the usual way we write expressions. An expression such as `A * ( B + C ) / D` is usually taken to mean something like: "First add B and C together, then multiply the result by A, then divide by D to give the final answer."

   Infix notation needs extra information to make the order of evaluation of the operators clear: rules built into the language about operator precedence and associativity, and brackets `( )` to allow users to override these rules. For example, the usual rules for associativity say that we perform operations from left to right, so the multiplication by A is assumed to come before the division by D. Similarly, the usual rules for precedence say that we perform multiplication and division before we perform addition and subtraction.

2. Postfix notation (also known as "Reverse Polish notation"): X Y +

   Operators are written after their operands. The infix expression given above is equivalent to `A B C + * D /`

   The order of evaluation of operators is always left-to-right, and brackets cannot be used to change this order. Because the "+" is to the left of the "*" in the example above, the addition must be performed before the multiplication.

   Operators act on values immediately to the left of them. For example, the "+" above uses the "B" and "C". We can add (totally unnecessary) brackets to make this explicit:

   ```
   ( (A (B C +) *) D /)
   ```

   Thus, the "*" uses the two values immediately preceding: "A", and the result of the addition. Similarly, the "/" uses the result of the multiplication and the "D".

3. Prefix notation (also known as "Polish notation"): + X Y

   Operators are written before their operands. The expressions given above are equivalent to `/ * A + B C D`

   As for Postfix, operators are evaluated left-to-right and brackets are superfluous. Operators act on the two nearest values on the right. I have again added (totally unnecessary) brackets to make this clear:

   ```
   (/ (* A (+ B C) ) D)
   ```

Examples:

Infix	Postfix	Prefix	Notes
A * B + C / D	A B * C D / +	+ * A B / C D	multiply A and B, divide C by D, add the results
A * (B + C) / D	A B C + * D /	/ * A + B C D	add B and C, multiply by A, divide by D
A * (B + C / D)	A B C D / + *	* A + B / C D	divide C by D, add B, multiply by A

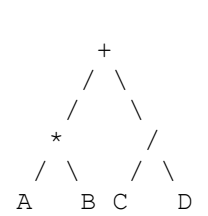
Converting between these notations

The most straightforward method is to start by inserting all the implicit brackets that show the order of evaluation e.g.:

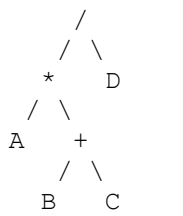
Infix	Postfix	Prefix
((A * B) + (C / D))	((A B *) (C D /) +)	(+ (* A B) (/ C D))
((A * (B + C)) / D)	((A (B C +) *) D /)	(/ (* A (+ B C)) D)
(A * (B + (C / D)))	(A (B (C D /) +) *)	(* A (+ B (/ C D)))

You can convert directly between these bracketed forms simply by moving the operator within the brackets e.g. `(X + Y)` or `(X Y +)` or `(+ X Y)`. Repeat this for all the operators in an expression, and finally remove any superfluous brackets.

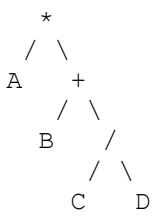
You can use a similar trick to convert to and from parse trees - each bracketed triplet of an operator and its two operands (or sub-expressions) corresponds to a node of the tree. The corresponding parse trees are:



((A*B)+(C/D))



((A*(B+C))/D)



(A*(B+(C/D)))

12 Define two-dimensional array? What is meant by row major and column major arrays? How address calculation in row major and column major arraysis done?

Two Dimensional Array in C

The two-dimensional array can be defined as an array of arrays. The 2D array is organized as matrices which can be represented as the collection of rows and columns. However, 2D arrays are created to implement a relational database lookalike data structure. It provides ease of holding the bulk of data at once which can be passed to any number of functions wherever required.

```
int arr[4][3]={{1,2,3},{2,3,4},{3,4,5},{4,5,6}};
```

Row Major Order

In row major order, the elements of a particular row are stored at adjacent memory locations. The first element of the array (arr[0][0]) is stored at the first location followed by the arr[0][1] and so on. After the first row, elements of the next row are stored next.

arr[3][3] =
[a00, a01, a02]
[b10, b11, b12]
[c20, c21, c22]

Row major order = a00, a01, a02, b10, b11, b12, c20, c12, c22

If the first element is stored at memory location 1048 and the elements are integers, then

- [1048] - a00
- [1052] - a01
- [1056] - a02
- [1060] - b10
- [1064] - b11
- [1068] - b12
- [1072] - c20
- [1076] - c21
- [1080] - c22

Column Major Order

In column major order, the elements of a column are stored adjacent to each other in the memory.The first element of the array (arr[0][0]) is stored at the first location followed by the arr[1][0] and so on. After the first column, elements of the next column are stored stating from the top.

arr[3][3] =
[a00, a01, a02]
[b10, b11, b12]
[c20, c21, c22]

Column major order = a00, b10, c20, a01, b11, c21, a02, b12, c22
If the first element is stored at memory location 1048 and the elements are integers, then:

- [1048] - a00
- [1052] - b10
- [1056] - c20
- [1060] - a01
- [1064] - b11
- [1068] - c21
- [1072] - a02
- [1076] - b12
- [1080] - c22

Finding address of element given the index

1D Array

*address[i] = I + i(sizeof (data type) - lower bound)**

Example:

Consider the base address of an boolean array to be 1048. Find the address of the element at index = 5. (Indexing is 0 based)

address[5] = I + i*(sizeof(boolean) - lower bound)

address[5] = 1048 + 5*(2) = 1048 + 10 = 1058

- 1048, 1049 = arr[0]

- 1050, 1051 = arr[1]
- 1052, 1053 = arr[2]
- 1054, 1055 = arr[3]
- 1056, 1057 = arr[4]
- **1058, 1059 = arr[5]**

2D Array

address[i][j] = I + W * (i - l_row) * N + (j - l_col)
Example:

Consider an integer array of size 3X3. The address of the first element is 1048. Calculate the address of the element at index i = 2, j = 1. (0 based index)

I = 1048, l_row = 0 = l_col, i = 2, j = 1, W = 2, M = 3

address[2][1] = I + W * (j - l_col) * M + (i - l_row)

address[2][1] = 1048 + 2 * 1 * 3 + 2 = 1048 + 6 + 2 = 1056