Introduction to Data Structures and Algorithms

Introduction

- A *data structure* is an arrangement of data either in computer's memory or on the disk storage.
- Some common examples of data structures are arrays, linked lists, queues, stacks, binary trees, graphs, and hash tables.
- Data structures are widely applied in areas like:
  - ✔ Compiler design
  - ✔ Operating system
  - ✔ Statistical analysis package
  - ✔ DBMS
  - ✔ Numerical analysis
  - ✔ Simulation
  - ✔ Artificial Intelligence

Classification of Data Structures

- Primitive data structures are the fundamental data types which are supported by a programming language. Some basic data types are integer, real, and boolean. The terms 'data type', 'basic data type', and 'primitive data type' are often used interchangeably.
- Non-primitive data structures are those data structures which are created using primitive data structures. Examples of such data structures include linked lists, stacks, trees, and graphs.
- Non-primitive data structures can further be classified into two categories: *linear* and *non-linear* data structures.

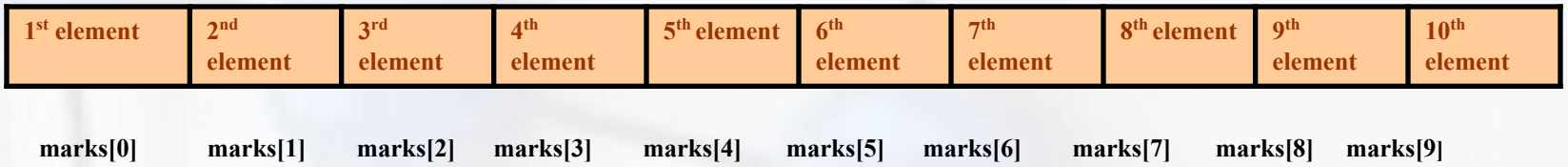
Classification of Data Structures

- If the elements of a data structure are stored in a linear or sequential order, then it is a *linear* data structure. Examples are arrays, linked lists, stacks, and queues.
- If the elements of a data structure are not stored in a sequential order, then it is a *non-linear* data structure. Examples are trees and graphs.

Arrays

- An array is a collection of similar data elements.
- The elements of an array are stored in consecutive memory locations and are referenced by an index (also known as the subscript).
- Arrays are declared using the following syntax:

type name[size];

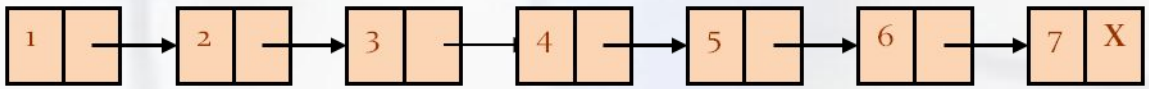


Linked Lists

- A linked list is a very flexible dynamic data structure in which elements can be added to or deleted from anywhere.
- In a linked list, each element (called a *node*) is allocated space as it is added to the list.
- Every node in the list points to the next node in the list. Therefore, in a linked list every node contains two types of information:

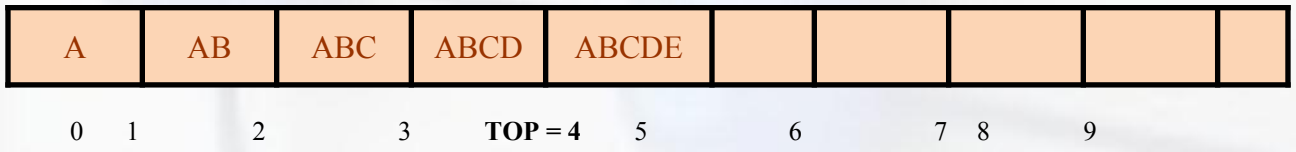
✔ The data stored in the node

✔ A pointer or link to the next node in the list



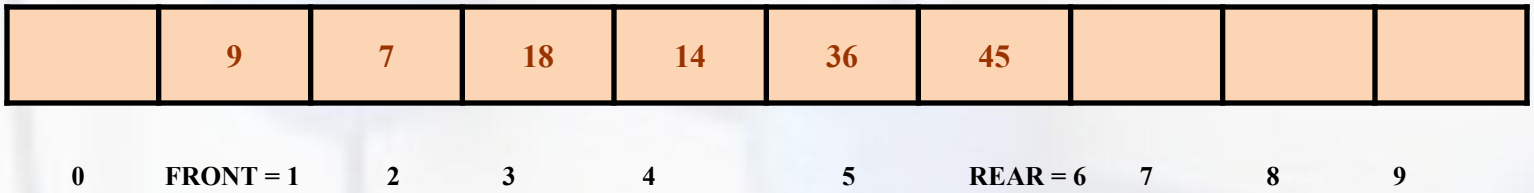
Stack

- A stack is a last-in, first-out (LIFO) data structure in which insertion and deletion of elements are done only at one end, known as TOP of the stack.
- Every stack has a variable TOP associated with it, which is used to store the address of the topmost element of the stack.
- If TOP = NULL, then it indicates that the stack is empty.
- If TOP = MAX-1, then the stack is full.



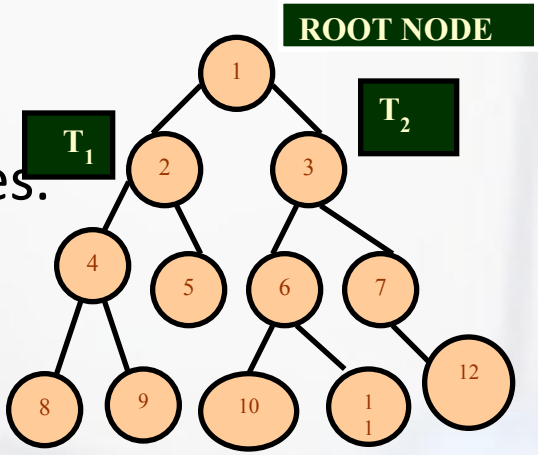
Queue

- A queue is a FIFO (first-in, first-out) data structure in which the element that is inserted first is the first one to be taken out.
- The elements in a queue are added at one end called the REAR and removed from the other one end called FRONT.
- When REAR = MAX – 1, then the queue is full.
- If FRONT = NULL and Rear = NULL, this means there is no element in the queue.



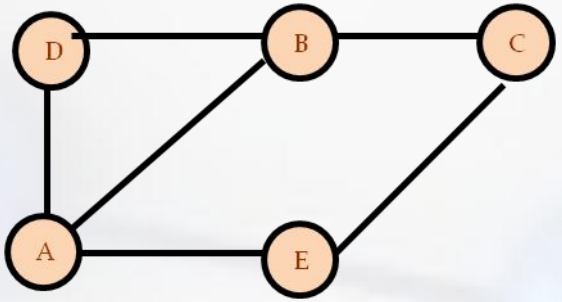
Tree

- A tree is a non-linear data structure which consists of a collection of nodes arranged in a hierarchical order.
- One of the nodes is designated as the root node, and the remaining nodes can be partitioned into disjoint sets such that each set is the sub-tree of the root.
- A binary tree is the simplest form of tree which consists of a root node and left and right sub-trees.
- The root element is pointed by 'root' pointer.
- If root = NULL, then it means the tree is empty.



Graph

- A graph is a non-linear data structure which is a collection of *vertices* (also called *nodes*) and *edges* that connect these vertices.
- A graph is often viewed as a generalization of the tree structure, where instead of a having a purely parent-to-child relationship between nodes, any kind of complex relationship can exist.
- Every node in the graph can be connected with any other node.
- When two nodes are connected via an edge, the two nodes are known as neighbors.



Abstract Data Type

- An ***Abstract Data Type*** **(ADT)** is the way at which we look at a data structure, focusing on what it does and ignoring how it does its job.
- For example, stacks and queues are perfect examples of an abstract data type. We can implement both these ADTs using an array or a linked list. This demonstrates the "abstract" nature of stacks and queues.

Abstract Data Type

- In C, an Abstract Data Type can be a structure considered without regard to its implementation. It can be thought of as a "description" of the data in the structure with a list of operations that can be performed on the data within that structure.
- The end user is not concerned about the details of how the methods carry out their tasks. They are only aware of the methods that are available to them and are concerned only about calling those methods and getting back the results but not HOW they work.

Algorithm

- An “algorithm" is a formally defined procedure for performing some calculation. It provides a blueprint to write a program to solve a particular problem.
- It is considered to be an effective procedure for solving a problem in finite number of steps. That is, a well-defined algorithm always provides an answer and is guaranteed to terminate.
- Algorithms are mainly used to achieve **software re-use**. Once we have an idea or a blueprint of a solution, we can implement it in any high level language like C, C++, Java, so on and so forth.

Algorithm

Write an algorithm to find whether a number is even or odd

Step 1: Input the first number as A

Step 2: IF A%2 =0

Then Print "EVEN"

ELSE

PRINT "ODD"

Step 3: END

Time and Space Complexity of Algorithm

- To analyze an algorithm means determining the amount of resources (such as time and storage) needed to execute it.
- Efficiency or complexity of an algorithm is stated in terms of time complexity and space complexity.
- *Time complexity* of an algorithm is basically the running time of the program as a function of the input size.
- Similarly, *space complexity* of an algorithm is the amount of computer memory required during the program execution, as a function of the input size.

Time and Space Complexity of Algorithm

- Time complexity of an algorithm depends on the number of instructions executed. This number is primarily dependent on the size of the program's input and the algorithm used.
- The space needed by a program depends on:

✔ Fixed part includes space needed for storing instructions, constants, variables, and structured variables.

✔ Variable part includes space needed for recursion stack, and for structured variables that are allocated space dynamically during the run-time of the program.

Calculating Algorithm Efficiency

- The efficiency of an algorithm is expressed in terms of the number of elements that has to be processed. So, if n is the number of elements, then the efficiency can be stated as ***Efficiency = f(n)***.
- If a function is linear (without any loops or recursions), the efficiency of that algorithm or the running time of that algorithm can be given as the number of instructions it contains.
- If an algorithm contains certain loops or recursive functions then its efficiency may vary depending on the number of loops and the running time of each loop in the algorithm.

Calculating Algorithm Efficiency

Linear loops

```
for(i=0;i<100;i++)
    statement block
```

f(n) = (n)

Logarithmic Loops

```
for(i=1;i<1000;i*=2)
    statement block;
```

```
for(i=1000;i>=1;i/=2)
statement block;
```

f(n) = log n

Nested Loops

Linear logarithmic

```
for(i=0;i<10;i++)
    for(j=1; j<10;j*=2)
        statement block;
```

f(n)= n log n

Big O Notation

- Big-O notation, where the "O" stands for "order of", is concerned with what happens for very large values of n.
- For example, if a sorting algorithm performs n^2 operations to sort just n elements, then that algorithm would be described as an $O(n^2)$ algorithm.
- When expressing complexity using Big O notation, constant multipliers are ignored. So a O(4n) algorithm is equivalent to O(n), which is how it should be written.

Big O Notation

- If f(n) and g(n) are functions defined on positive integer number n, then

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

- That is, f of n is big O of g of n if and only if there exists positive constants c and n, such that

  $f(n) \leq cg(n)$

- This means that for large amounts of data, f(n) will grow no more than a constant factor than g(n). Hence, g provides an upper bound.

Categories of Algorithms

- Constant time algorithms have running time complexity given as O(1)
- Linear time algorithms have running time complexity given as O(n)
- Logarithmic time algorithms have running time complexity given as O(log n)
- Polynomial time algorithms have running time complexity given as O(nk) where k>1
- Exponential time algorithms have running time complexity given as $O(2^n)$

n	O(1)	O(log n)	O(n)	O(n log n)	$O(n^2)$	$O(n^3)$
1	1	1	1	1	1	1
2	1	1	2	2	4	8
4	1	2	4	8	16	64
8	1	3	8	24	64	512
16	1	4	16	64	256	4,096

Omega Notation

- Omega notation provides a tight lower bound for f(n). This means that the function can never do better than the specified value but it may do worse.
- Ω notation is simply written as, $f(n) \in \Omega(g(n))$, where n is the problem size and $\Omega(g(n)) = \{h(n): \exists$ positive constants $c > 0$, n_0 such that $0 \leq cg(n) \leq h(n)$, $\forall\ n \geq n_0\}$.
- Examples of functions in $\Omega(n^2)$ include: n^2, $n^{2.9}$, $n^3 + n$, $540n^2 + 10$
- Examples of functions not in $\Omega(n^3)$ include: n, $n^{2.9}$, n^2

Theta Notation

- Theta notation provides an asymptotically tight bound for f(n).
- Θ notation is simply written as, $f(n) \in \Theta(g(n))$, where n is the problem size and $\Theta(g(n)) = \{h(n): \exists$ positive constants c_1, c_2 and n_0 such that $0 \leq c_1 g(n) \leq h(n) \leq c_2 g(n), \forall\ n \geq n_0\}$.
- Hence, we can say that Θ(g(n)) comprises a set of all the functions h(n) that are between c1g(n) and c2g(n) for all values of $n \geq n_0$.
- To summarize,

✔ The best case in Θ notation is not used.

✔ Worst case Θ describes asymptotic bounds for worst case combination of input values.

✔ If we simply write Θ, it means same as worst case Θ.

Little o Notation

- Little o notation provides a non-asymptotically tight upper bound for f(n).
- To express a function using this notation, we write $f(n) \in o(g(n))$ where $o(g(n)) = \{h(n) : \exists$ positive constants c, n_0 such that for any $c > 0$, $n_0 > 0$ and $0 \leq h(n) \leq cg(n)$, $\forall\ n \geq n_0\}$.
- This is unlike the Big O notation where we say for some c > 0 (not any).
- For example, $5n^3 = O(n^3)$ is asymptotically tight upper bound but $5n^2 = O(n^3)$ is non-asymptotically tight bound for f(n).
- Examples of functions in $o(n^3)$ include: $n^{2.9}$, $n^3 / \log n$, $2n^2$
- Examples of functions not in $o(n^3)$ include: $3n^3$, n^3, $n^3 / 1000$

Little Omega Notation

- Little Omega notation provides a non-asymptotically tight lower bound for f(n).
- It can be simply written as, $f(n) \in \omega(g(n))$, where $\omega(g(n)) = \{h(n) : \exists$ positive constants c, n_0 such that for any $c > 0$, $n_0 > 0$ and $0 \leq cg(n) < h(n)$, $\forall\ n \geq n_0\}$.
- This is unlike the Ω notation where we say for some c > 0 (not any).
- For example, $5n^3 = \Omega(n^3)$ is asymptotically tight upper bound but $5n^2 = \omega(n^3)$ is non-asymptotically tight bound for f(n).
- Example of functions in $\omega(g(n))$ include: $n^3 = \omega(n^2)$, $n^{3.001} = \omega(n^3)$, $n^2 \log n = \omega(n^2)$
- Example of a function not in $\omega(g(n))$ is $5n^2 \neq \omega(n^2)$ (just as $5 \neq 5$) |