Acknowledgement

The internship opportunity I had with BCAS campus was a great chance for learning and professional development. Therefore I consider myself as a very lucky individual as I was provided with an opportunity to be a part of it. I am also grateful for having chance to meet so many wonderful people and professionals who led me through this internship period.

I express my deepest thanks to the assessor Eng. A. L. Jubailah Begum, who works as an IT professional and take lectures on Data Structure & Algorithm subjects and others supportive subjects. She helped me taking part in useful decision & giving necessary advices and guidance and arrange all facilities for this assignment. I choose this moment to acknowledge his contribution gratefully. I acknowledge that this assignment was done with help of internet resources.

Sincerely,
S. SHALOMSHAN
Date:10.04.2021

Introduction

The primary purpose of this assignment is to give a deep brief explanation about Data Structure & Algorithm concept. This assignment points out some important essential fundamentals about abstract data types.

This assignment have been done in a perfect manner. The task mainly focused on the basic idea of ADTs. The task 1.2 is show the sorting methodologies and compare the performance of bubble sort and quick sort. The task 1.3 is show analyze the operation shortest path algorithm Dijkstra's algorithm and Bellman Ford algorithm. The task 2 is explanation on ADTs of software stack, encapsulation and information hiding, imperative ADTs with regards to object orientation. The task 3 is implementation part I have C# programming language and error handling report. The last task is briefly explain about asymptotic analysis, trade-off is when specifying an ADT, benefits of using implementation independent data structures and two ways in which the efficiency of an algorithm can be measured.

The more detailed about each task has been explained in their own part of this assignment. This main introduction is only giving the basement idea of the assignment.

Contents

PART:- 1

Task:- 1.1

a.

Stack

A stack is a restricted linear list in which all additions and deletions are made at one end, the top. If we insert a series of data items into a stack and then remove them, the order of the data is reversed. This reversing attribute is why stacks are known as last in, first out (LIFO) data structures.

Figure 1 Stack Example

A real-world stack allows operations at one end only. For example, we can place or remove a card or plate from the top of the stack only. Likewise, Stack ADT allows all data operations at one end only. At any given time, we can only access the top element of a stack.

This feature makes it LIFO data structure. LIFO stands for Last-in-first-out. Here, the element which is placed (inserted or added) last, is accessed first. In stack terminology, insertion operation is called PUSH operation and removal operation is called POP operation.

Stack Representation

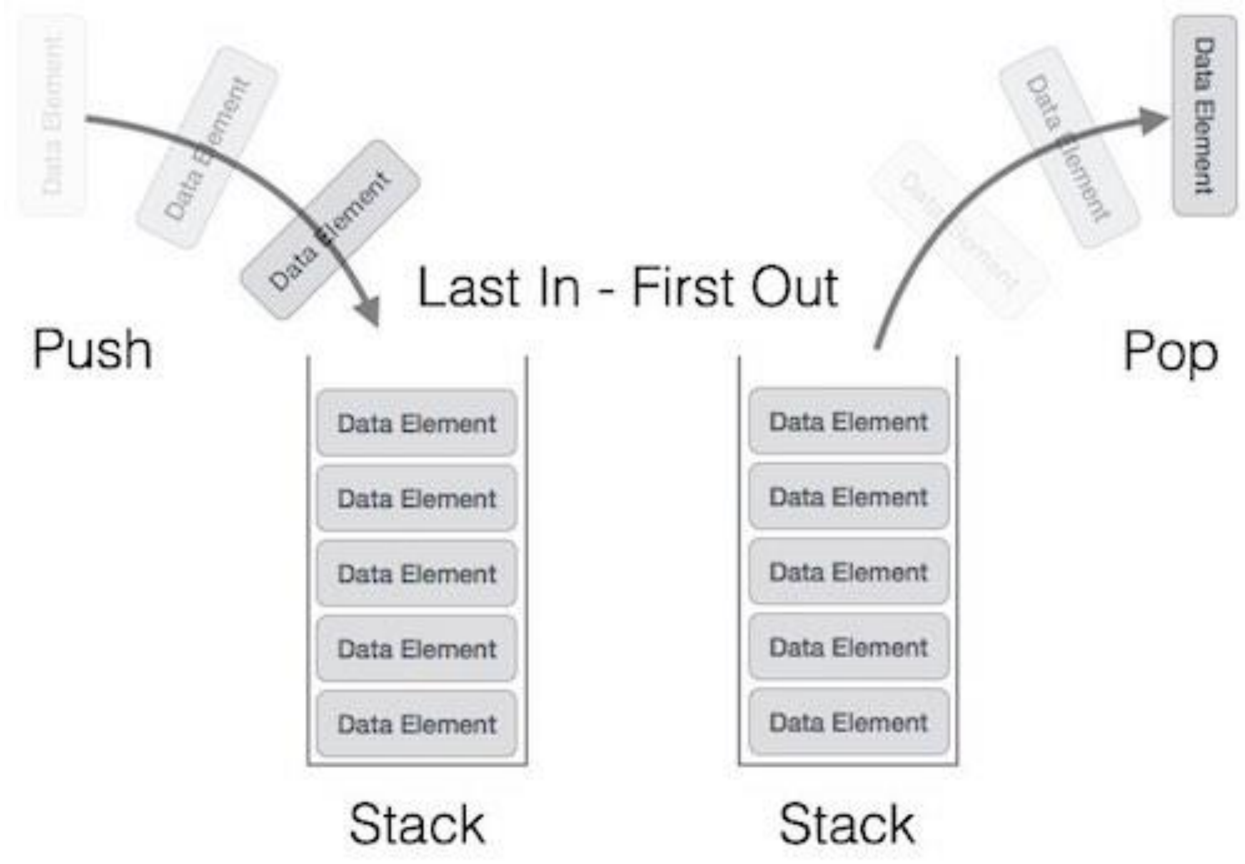


Figure 2 Stack Representation

A stack can be implemented by means of Array, Structure, Pointer, and Linked List. Stack can either be a fixed size one or it may have a sense of dynamic resizing. Here, we are going to implement stack using arrays, which makes it a fixed size stack implementation.

Basic operation

Stack operations may involve initializing the stack, using it and then de-initializing it. Apart from these basic stuffs, a stack is used for the following two primary operations

- push() − Pushing (storing) an element on the stack.
- pop() − Removing (accessing) an element from the stack.

When data is PUSHed onto stack.

To use a stack efficiently, we need to check the status of stack as well. For the same purpose, the following functionality is added to stacks −

- **peek()** − get the top data element of the stack, without removing it.
- **isFull()** − check if stack is full.
- **isEmpty()** − check if stack is empty.

At all times, we maintain a pointer to the last PUSHed data on the stack. As this pointer always represents the top of the stack, hence named top. The top pointer provides top value of the stack without actually removing it.

Push operation

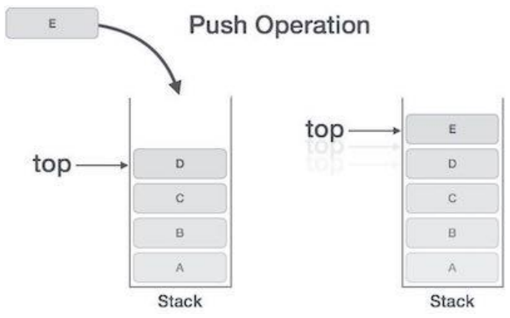


Figure 3 Push operation

The process of putting a new data element onto stack is known as a Push Operation. Push operation involves a series of steps

- Step 1 − Checks if the stack is full.
- Step 2 − If the stack is full, produces an error and exit.
- Step 3 − If the stack is not full, increments top to point next empty space.
- Step 4 − Adds data element to the stack location, where top is pointing.
- Step 5 − Returns success.

If the linked list is used to implement the stack, then in step 3, we need to allocate space dynamically.

Pop operation

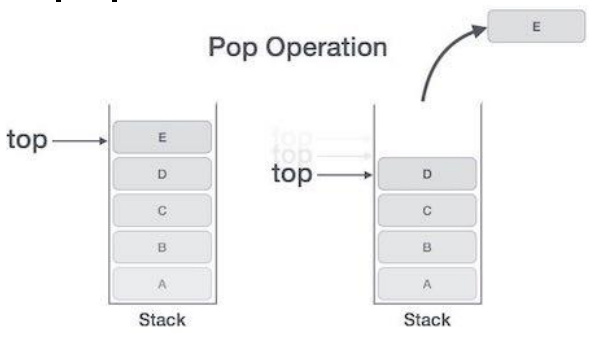


Figure 4 Pop operation

Accessing the content while removing it from the stack, is known as a Pop Operation. In an array implementation of pop() operation, the data element is not actually removed, instead top is decremented to a lower position in the stack to point to the next value. But in linked-list implementation, pop() actually removes data element and deallocates memory space.

A Pop operation may involve the following steps

- Step 1 − Checks if the stack is empty.
- Step 2 − If the stack is empty, produces an error and exit.
- Step 3 − If the stack is not empty, accesses the data element at which top is pointing.
- Step 4 − Decreases the value of top by 1.
- Step 5 − Returns success.

Queue

A queue is a linear list in which data can only be inserted at one end, called the rear, and deleted from the other end, called the front. These restrictions ensure that the data is processed through the queue in the order in which it is received. In other words, a queue is a first in, first out (FIFO) structure.

Figure 5 Queue Example

Queue representation

As we now understand that in queue, we access both ends for different reasons. The following diagram given below tries to explain queue representation as data structure.

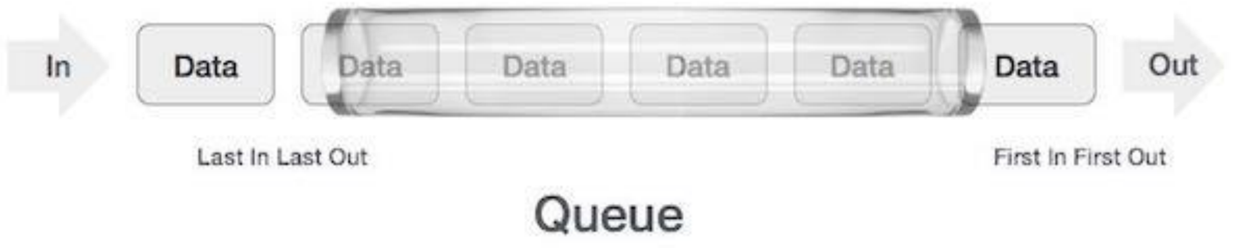


Figure 6 Queue diagram

As in stacks, a queue can also be implemented using Arrays, Linked-lists, Pointers and Structures. For the sake of simplicity, we shall implement queues using one-dimensional array.

Basic operation

Queue operations may involve initializing or defining the queue, utilizing it, and then completely erasing it from the memory. Here we shall try to understand the basic operations associated with queues

- **enqueue()** − add (store) an item to the queue.
- **dequeue()** − remove (access) an item from the queue.

Few more functions are required to make the above-mentioned queue operation efficient. These are

- **peek()** − Gets the element at the front of the queue without removing it.
- **isfull()** − Checks if the queue is full.
- **isempty()** − Checks if the queue is empty.

In queue, we always dequeue (or access) data, pointed by front pointer and while enqueing (or storing) data in the queue we take help of rear pointer.

Enqueue operation

Queues maintain two data pointers, front and rear. Therefore, its operations are comparatively difficult to implement than that of stacks.

The following steps should be taken to enqueue (insert) data into a queue

- **Step 1** − Check if the queue is full.
- **Step 2** − If the queue is full, produce overflow error and exit.
- **Step 3** − If the queue is not full, increment rear pointer to point the next empty space.
- **Step 4** − Add data element to the queue location, where the rear is pointing.
- **Step 5** − return success.

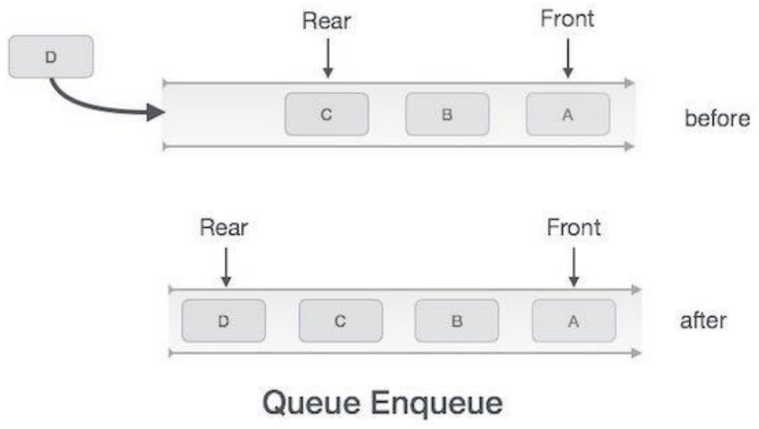


Figure 7 Enqueue diagram

Dequeue operation

Accessing data from the queue is a process of two tasks − access the data where front is pointing and remove the data after access. The following steps are taken to perform dequeue operation −

- **Step 1** − Check if the queue is empty.
- **Step 2** − If the queue is empty, produce underflow error and exit.
- **Step 3** − If the queue is not empty, access the data where front is pointing.
- **Step 4** − Increment front pointer to point to the next available data element.
- **Step 5** − Return success.

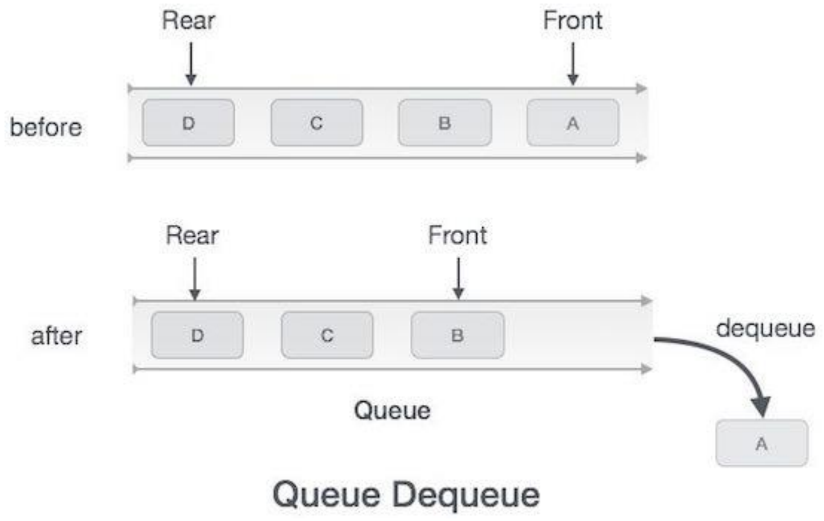
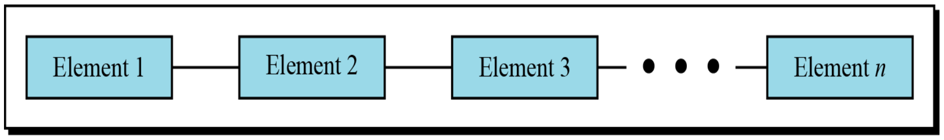


Figure 8 Dequeue diagram

Linear list

Stacks and queues defined in the two previous sections are restricted linear lists. A general linear list is a list in which operations, such as insertion and deletion, can be done anywhere in the list—at the beginning, in the middle or at the end.



General linear list

Operation linear list

Although we can define many operations on a general linear list, we discuss only six common operations in this chapter: list, insert, delete, retrieve, traverse and empty.

The list operation

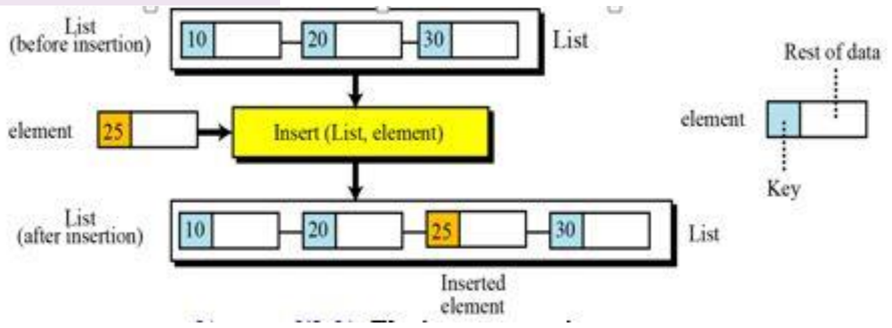
The list operation creates an empty list. The following shows the format:

list (listName)

The insert operation

Since we assume that data in a general linear list is sorted, insertion must be done in such a way that the ordering of the elements is maintained. To determine where the element is to be placed, searching is needed. However, searching is done at the implementation level, not at the ADT level.

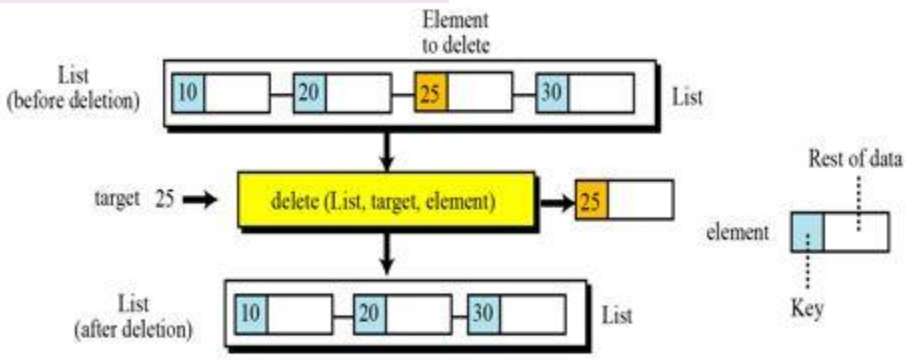
insert (listName, element)



The delete operation

Deletion from a general list (Figure 12.16) also requires that the list be searched to locate the data to be deleted. After the location of the data is found, deletion can be done. The following shows the format:

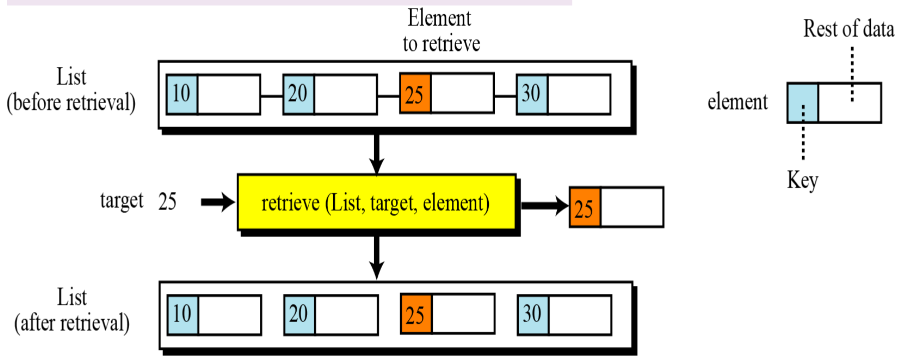
delete (listName, target, element)



The retrieve operation

By retrieval, we mean access of a single element. Like insertion and deletion, the general list should be first searched, and if the data is found, it can be retrieved. The format of the retrieve operation is:

retrieve (listName, target, element)



The traverse operation

Each of the previous operations involves a single element in the list, randomly accessing the list. List traversal, on the other hand, involves sequential access. It is an operation in which all elements in the list are processed one by one. The following shows the format:

traverse (listName, action)

The empty operation

The empty operation checks the status of the list. The following shows the format:

empty (listName)

This operation returns true if the list is empty, or false if the
list is not empty.

Binary tree

A binary tree is a tree in which no node can have more than two subtrees. In other words, a node can have zero, one or two subtrees.

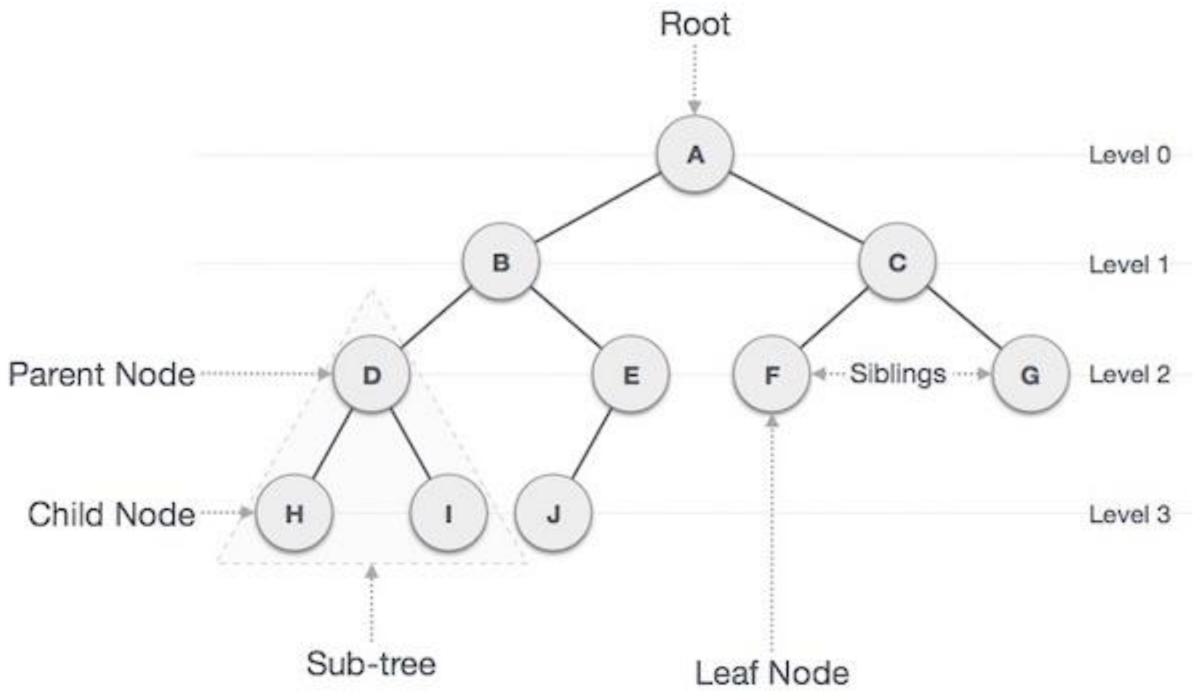


Figure 9 Binary tree

Important terms

Following are the important terms with respect to tree.

- Path − Path refers to the sequence of nodes along the edges of a tree.
- Root − The node at the top of the tree is called root. There is only one root per tree and one path from the root node to any node.
- Parent − Any node except the root node has one edge upward to a node called parent.
- Child − The node below a given node connected by its edge downward is called its child node.
- Leaf − The node which does not have any child node is called the leaf node.
- Subtree − Subtree represents the descendants of a node.
- Visiting − Visiting refers to checking the value of a node when control is on the node.
- Traversing − Traversing means passing through nodes in a specific order.
- Levels − Level of a node represents the generation of a node. If the root node is at level 0, then its next child node is at level 1, its grandchild is at level 2, and so on.
- keys − Key represents a value of a node based on which a search operation is to be carried out for a node.

Binary search tree representation

Binary Search tree exhibits a special behavior. A node's left child must have a value less than its parent's value and the node's right child must have a value greater than its parent value.

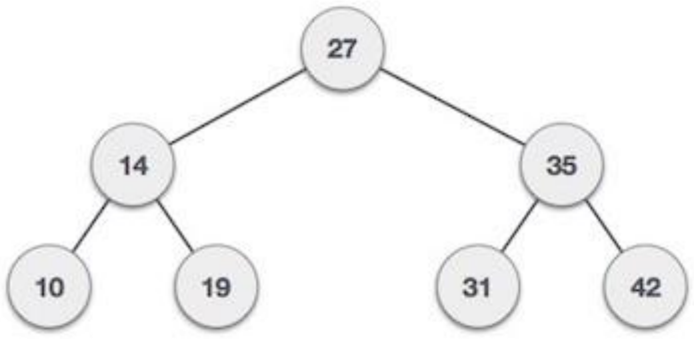


Figure 10 Binary search tree

We're going to implement tree using node object and connecting them through references.

Tree node

The code to write a tree node would be similar to what is given below. It has a data part and references to its left and right child nodes.

Binary search tree basic operation

The basic operations that can be performed on a binary search tree data structure, are the following

- Insert − Inserts an element in a tree/create a tree.
- Search − Searches an element in a tree.
- Preorder Traversal − Traverses a tree in a pre-order manner.
- In order Traversal − Traverses a tree in an in-order manner.
- Post order Traversal − Traverses a tree in a post-order manner.

We shall learn creating (inserting into) a tree structure and searching a data item in a tree in this chapter. We shall learn about tree traversing methods in the coming chapter.

Insert operation

The very first insertion creates the tree. Afterwards, whenever an element is to be inserted, first locate its proper location. Start searching from the root node, then if the data is less than the key value, search for the empty location in the left subtree and insert the data. Otherwise, search for the empty location in the right subtree and insert the data.

Search operation

Whenever an element is to be searched, start searching from the root node, then if the data is less than the key value, search for the element in the left subtree. Otherwise, search for the element in the right subtree. Follow the same algorithm for each node.

b.

Queue A First in First out (FIFO)

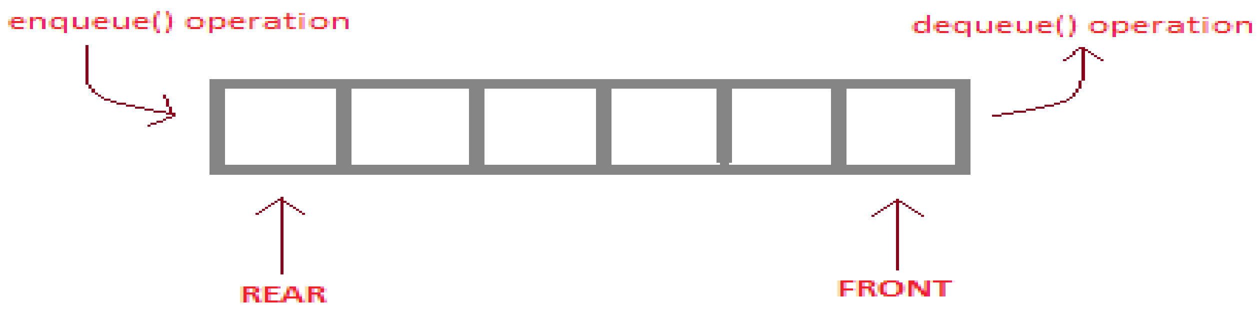
Queue is also an abstract data type or a linear data structure, just like stack data structure, in which the first element is inserted from one end called the REAR (also called tail), and the removal of existing element takes place from the other end called as FRONT (also called head).

This makes queue as FIFO (First in First Out) data structure, which means that element inserted first will be removed first.

Which is exactly how queue system works in real world. If we go to a ticket counter to buy movie tickets, and are first in the queue, then we will be the first one to get the tickets. Right?

Same is the case with Queue data structure. Data inserted first, will leave the queue first.

The process to add an element into queue is called Enqueue and the process of removal of an element from queue is called Dequeue.



enqueue( ) is the operation for adding an element into Queue.

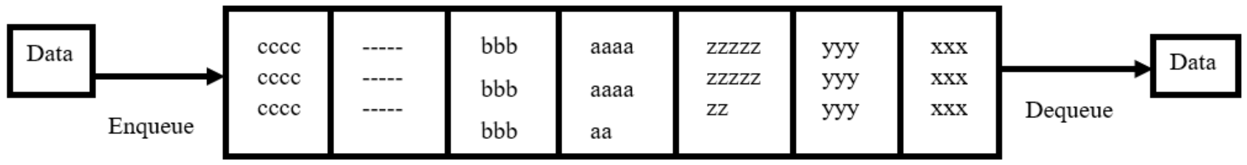
dequeue( ) is the operation for removing an element from Queue .

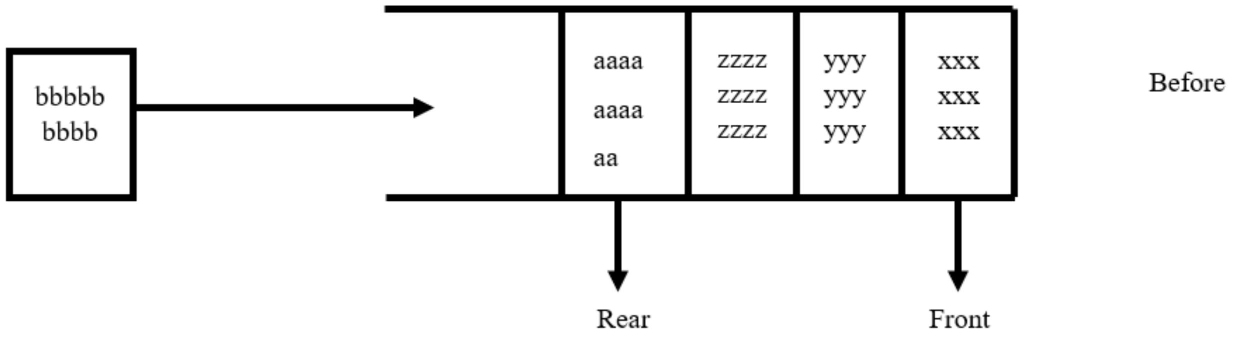
How queue helps to store the data?

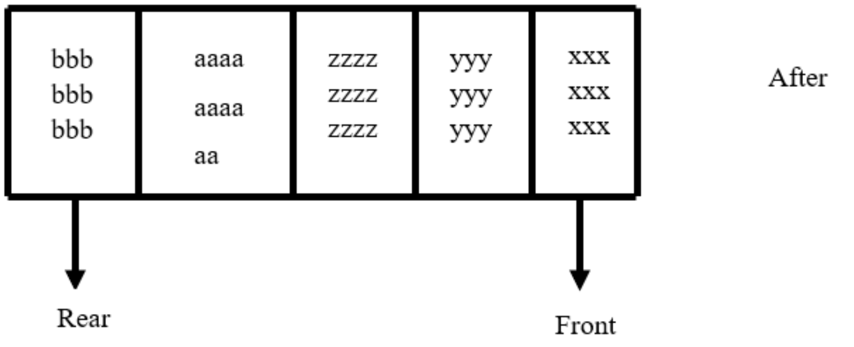
Queue is an abstract data structure, somewhat similar to Stacks. Unlike stacks, a queue is open at both its ends. One end is always used to insert data (enqueue) and the other is used to remove data (dequeue). Queue follows First-In-First-Out methodology, i.e., the data item stored first will be accessed first.

LAST IN LAST OUT
ONE WAY
FIRST IN FIRST OUT

A real-world example of queue can be a single-lane one-way road, where the vehicle enters first, exits first. More real-world examples can be seen as queues at the ticket windows and bus-stops.







Basically, queue is similar to the stack. Based on the scenario I have design a process which show how the data are stored in the queue. Traffic management system contains huge amount of data and those data can be stored with the help of the queue. In the first picture I have shown the function of enqueue and dequeue. Talking about how the data are stored in the queue then the second figure clearly explain that. The place where the first data that is stored in the queue is known as the front and the place where the last data is stored in the queue is known as the rear. There are five steps that should be considered to store the data in the queue which I have shown in the second figure for storing the data of the vehicles management system. The five steps are as:

- Step 1 − Check if the queue is full.
- If the queue is full, produce overflow error and exit.
- Step 3 − If the queue is not full, increment rear pointer to point the next empty space.
- Step 4 − Add data element to the queue location, where the rear is pointing.
- Step 5 − return success.

c.

Memory stack

Basically, memory stack is a special region of our computer’s memory that stores temporary variables created by each function (including the main () function).

The stack is a “LIFO” data structure, that is managed and optimized by the CPU quite closely.

Every time a function declares a new variable it is pushed onto the stack. Then every time a function exits, all the variables pushed onto the stack by that function, are freed.

Once a stack variable is freed that region of memory becomes available for other stack variables..

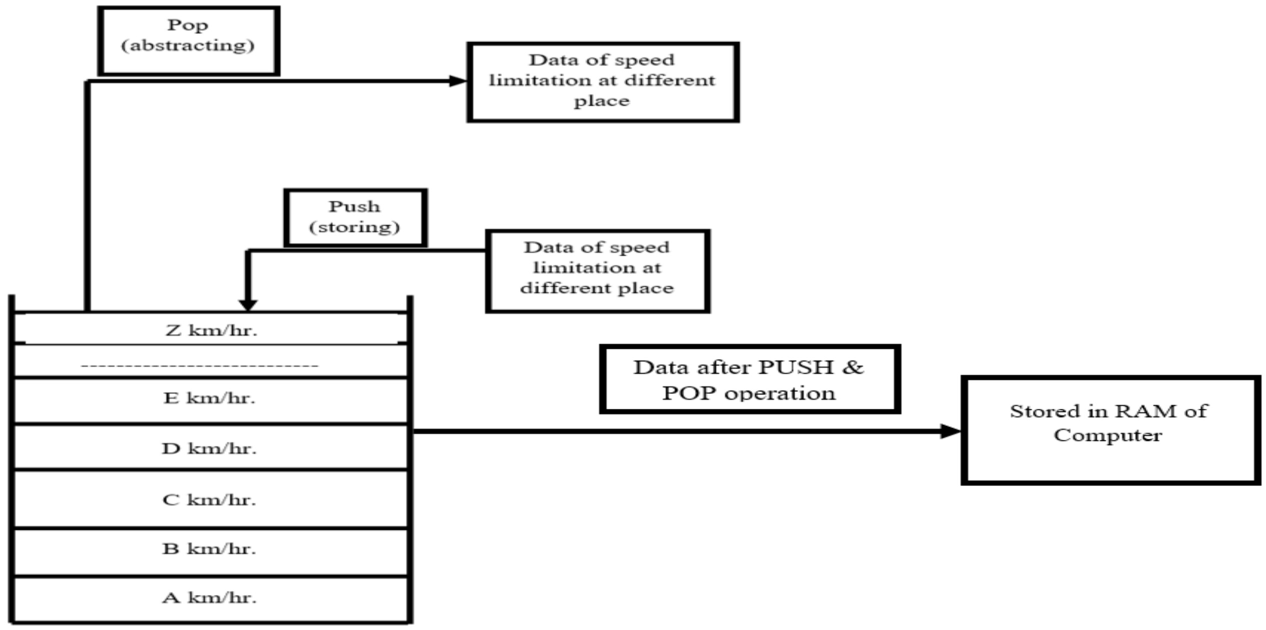
Working of stack memory

In stack memory there is the stack memory segment, which is an area of memory allotted for automatic variables. Automatic variables are allocated and de-allocated automatically when program flow enters and leaves the variable's scope. Stack memory is allocated and de-allocated

at the top of the stack, using Last-In-First-Out (LIFO). Because of this process, stack allocation is simpler and typically faster than heap allocation. In most modern computer systems, each thread has its own stack memory segment, the size of which can be fairly small as little as a few dozen kilobytes. Allocating more memory on the stack than is available can result in a stack overflow error, which is frequently associated with security issues.

Uses of memory stack on memory devices with demonstration

Basically, the main use of the memory stack on memory devices is to store the temporary variables created by each function (including the main () function). Basically, RAM is the best example of memory device and RAM store the data temporarily and the main use of the memory stack is to store the temporary variable. The data of the memory stack is stored in the RAM of the computer. So, in conclusion it can be said that the use of the memory stack on the memory device like RAM is to store the temporary variables.


Pop (abstracting)
Data of speed limitation at different place
Push (storing)
Data of speed limitation at different place
Z km/hr.
E km/hr.
D km/hr.
C km/hr.
B km/hr.
A km/hr.
Data after PUSH & POP operation
Stored in RAM of Computer

In the above figure speed limitation data of the traffic management system is stored in the stack. Two different operation are done Push and POP. The data that are finally obtained after doing the push and pop operation will be the temporary variables and those temporary sate are stored in the RAM of the computer. RAM is consider as the memory device of computer which store the data permanently.

From the above statement is can be concluded that the use of the memory stack in the memory device like RAM is to store the temporary data. Show this is the best demonstration of the use of memory stack on the memory devices.

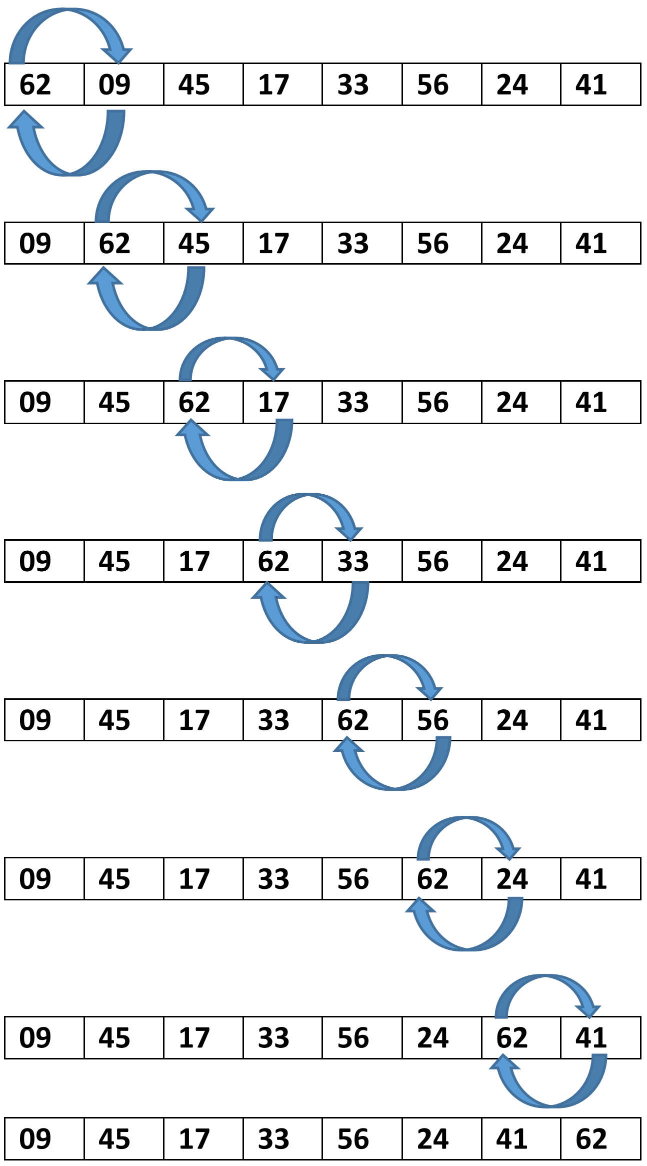
How memory stack help to solve function calling of different operation in computer?

Basically, the main function of stack is to store the data. In the above screenshot I have shown the process of storing the data of speed limitation. There are two types of function in the stack they are Push and Pop and their operation is to store and abstract the data respectively. With the help of the push function I can easily store the needed data of the speed limitation in the systematic way. The data that are store in the stack are the temporary variables. In our computer system the RAM play the vital role to store the temporary data. The data store in stack are save in the computer memory i.e. RAM. In the RAM different operations like: insert, delete, update, etc. takes places and in the stack also the same operation is performed. The push operation is used to insert the data as well as the pop operation is used to delete the data in the stack. The data are arranged in systematic order in stack. So the update operation can easily be done in stack.

Task:- 1.2

62	09	45	17	33	56	24	41

1. Bubble sort



62	09	45	17	33	56	24	41

09	62	45	17	33	56	24	41

09	45	62	17	33	56	24	41

09	45	17	62	33	56	24	41

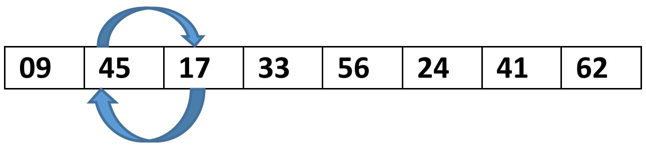
09	45	17	33	62	56	24	41

09	45	17	33	56	62	24	41

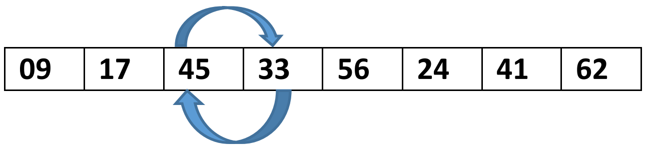
09	45	17	33	56	24	62	41

09	45	17	33	56	24	41	62

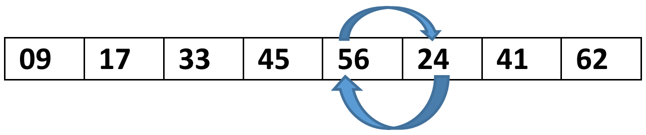
END OF FIRST PASS



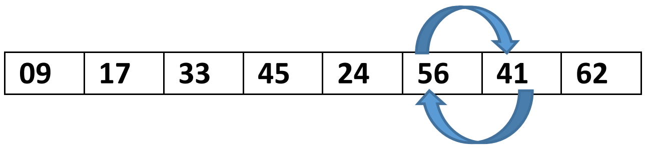
09	45	17	33	56	24	41	62



09	17	45	33	56	24	41	62



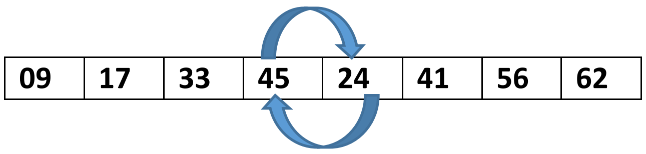
09	17	33	45	56	24	41	62



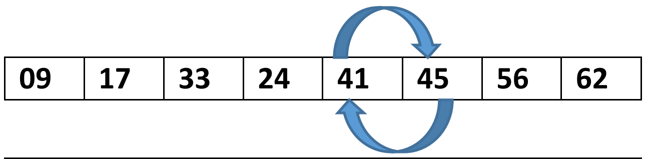
09	17	33	45	24	56	41	62

09	17	33	45	24	41	56	62

END OF SECOND PASS



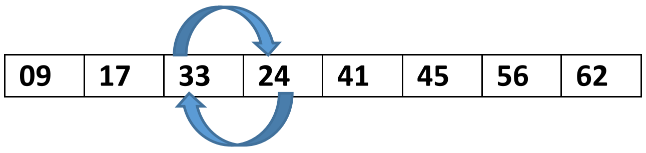
09	17	33	45	24	41	56	62



09	17	33	24	41	45	56	62

09	17	33	24	41	45	56	62

END OF THIRD PASS



09	17	33	24	41	45	56	62

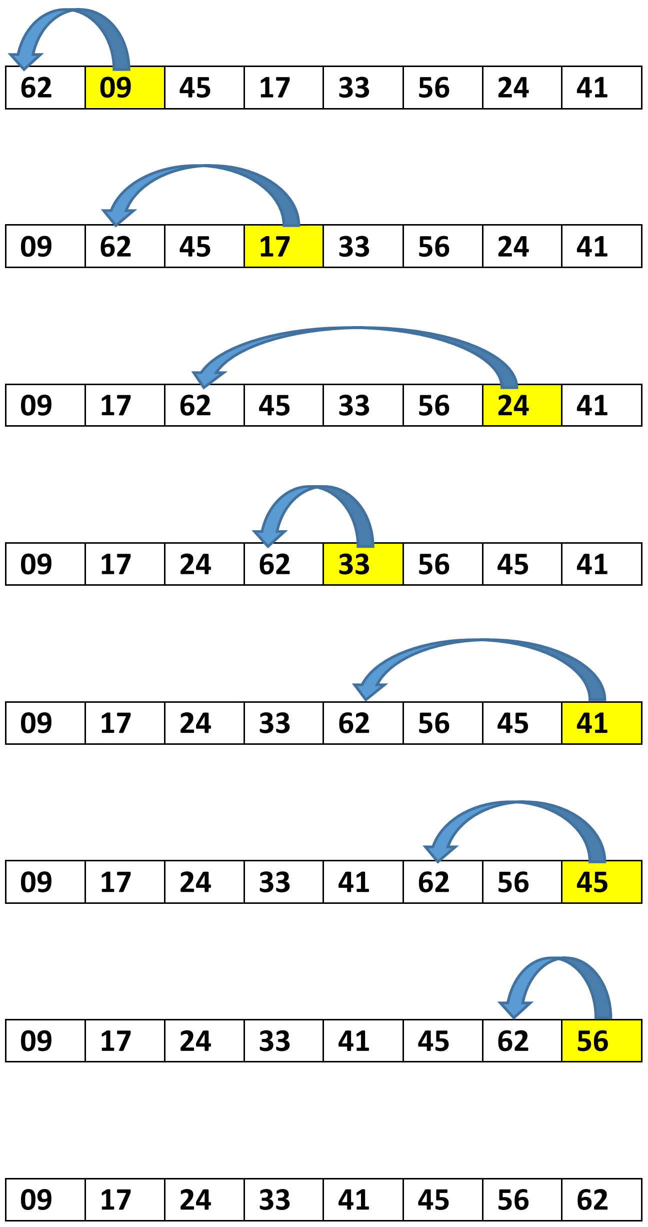
09	17	24	33	41	45	56	62

END OF FOURTH PASS

09	17	24	33	41	45	56	62

SORTED ARRAY

2. Insertion sort



62	09	45	17	33	56	24	41

09	62	45	17	33	56	24	41

09	17	62	45	33	56	24	41

09	17	24	62	33	56	45	41

09	17	24	33	62	56	45	41

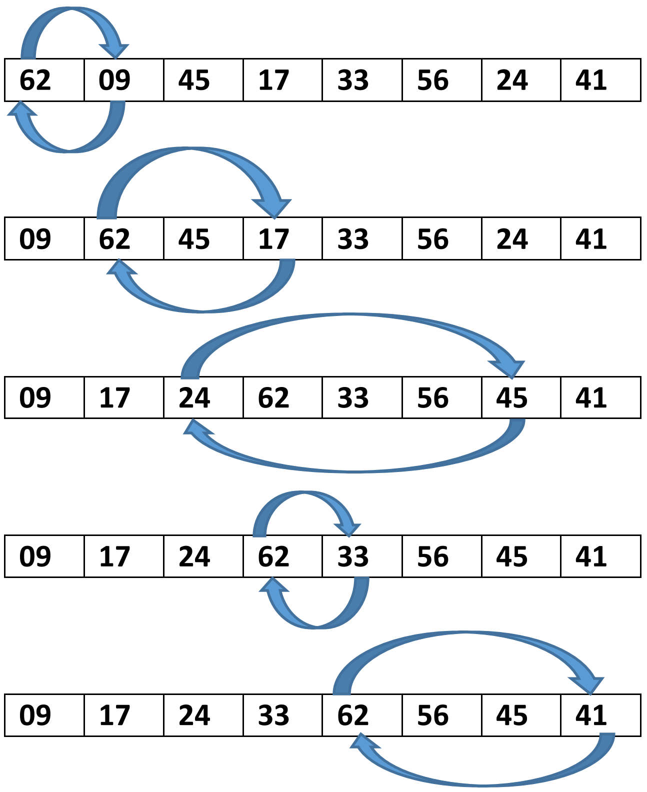
09	17	24	33	41	62	56	45

09	17	24	33	41	45	62	56

09	17	24	33	41	45	56	62

SORTED ARRAY

3. Selection sort



62	09	45	17	33	56	24	41

09	62	45	17	33	56	24	41

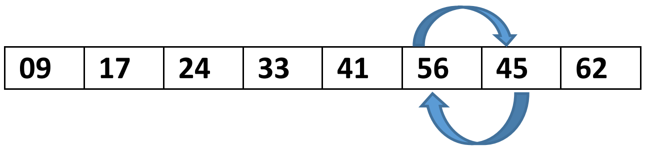
09	17	24	62	33	56	45	41

09	17	24	62	33	56	45	41

09	17	24	33	62	56	45	41

09	17	24	33	41	56	45	62

END OF FIRST PASS



09	17	24	33	41	56	45	62

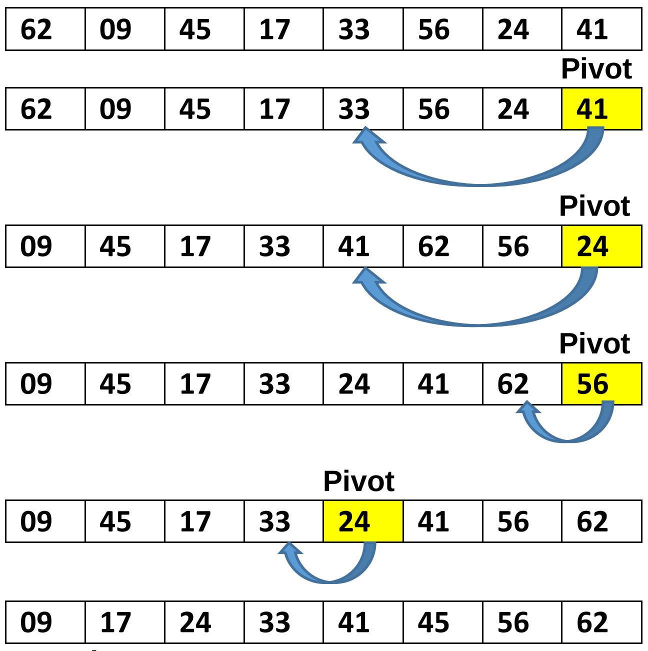
09	17	24	33	41	45	56	62

END OF SECOND PASS

09	17	24	33	41	56	45	62

SORTED ARRAY

4. Quick sort



Sorted array

Advantage and disadvantage of bubble sort

Advantage	Disadvantage
The primary advantage of the bubble sort is that it is popular and easy to implement.	The main disadvantage of the bubble sort is the fact that it does not deal well with a list containing a huge number of items. The main disadvantage of the bubble sort is the fact that it does not deal well with a list containing a huge number of items.
In the bubble sort, elements are swapped in place without using additional temporary storage.	The bubble sort requires n-squared processing steps for every n number of elements to be sorted.
The space requirement is at a minimum.	The bubble sort is mostly suitable for academic teaching but not for real-life applications.

Advantage and disadvantage of quick sort

Advantage	Disadvantage
The quick sort is regarded as the best sorting algorithm.	The slight disadvantage of quick sort is that its worst-case performance is similar to average performances of the bubble, insertion or selections sorts.
It is able to deal well with a huge list of items.	If the list is already sorted than bubble sort is much more efficient than quick sort.
Because it sorts in place, no additional storage is required as well	If the sorting element is integers than radix sort is more efficient than quick sort.

Task:- 1.3

1.

Dijkstra’s algorithm

Dijkstra’s algorithm is very similar to Prim’s algorithm for minimum spanning tree. Like Prim’s MST, we generate a SPT (shortest path tree) with given source as root. We maintain two sets, one set contains vertices included in shortest path tree, other set includes vertices not yet included in shortest path tree. At every step of the algorithm, we find a vertex which is in the other set (set of not yet included) and has a minimum distance from the source.

Below are the detailed steps used in Dijkstra’s algorithm to find the shortest path from a single source vertex to all other vertices in the given graph.

Algorithm

1) Create a set sptSet (shortest path tree set) that keeps track of vertices included in shortest path tree, i.e., whose minimum distance from source is calculated and finalized. Initially, this set is empty.

2) Assign a distance value to all vertices in the input graph. Initialize all distance values as INFINITE. Assign distance value as 0 for the source vertex so that it is picked first.

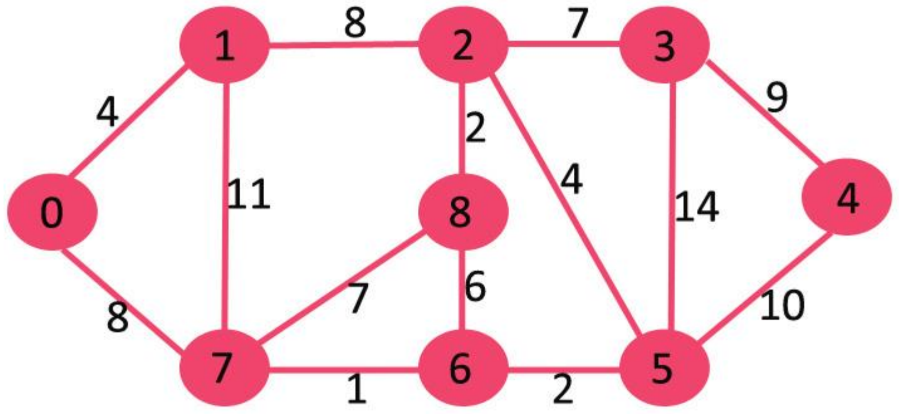
3) While sptSet doesn’t include all vertices

a) Pick a vertex u which is not there in sptSet and has minimum distance value.

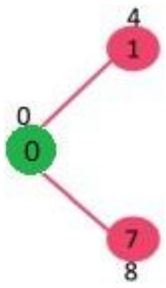
b) Include u to sptSet.

c) Update distance value of all adjacent vertices of u. To update the distance values, iterate through all adjacent vertices. For every adjacent vertex v, if sum of distance value of u (from source) and weight of edge u-v, is less than the distance value of v, then update the distance value of v.

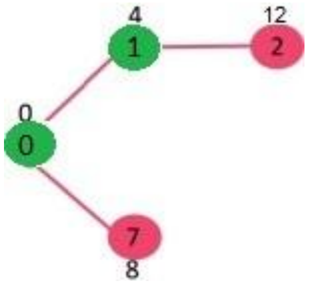
Lets us understand with the following example:



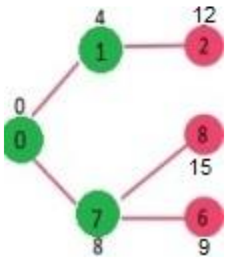
The set sptSet is initially empty and distances assigned to vertices are {0, INF, INF, INF, INF, INF, INF, INF} where INF indicates infinite. Now pick the vertex with minimum distance value. The vertex 0 is picked, include it in sptSet. So sptSet becomes {0}. After including 0 to sptSet, update distance values of its adjacent vertices. Adjacent vertices of 0 are 1 and 7. The distance values of 1 and 7 are updated as 4 and 8. Following subgraph shows vertices and their distance values, only the vertices with finite distance values are shown. The vertices included in SPT are shown in green color.



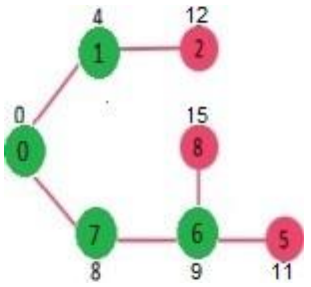
Pick the vertex with minimum distance value and not already included in SPT (not in sptSET). The vertex 1 is picked and added to sptSet. So sptSet now becomes {0, 1}. Update the distance values of adjacent vertices of 1. The distance value of vertex 2 becomes 12.



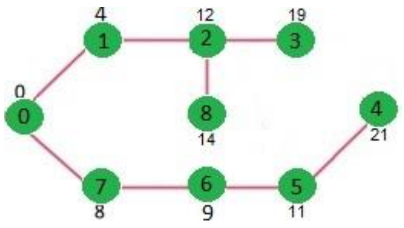
Pick the vertex with minimum distance value and not already included in SPT (not in sptSET). Vertex 7 is picked. So sptSet now becomes {0, 1, 7}. Update the distance values of adjacent vertices of 7. The distance value of vertex 6 and 8 becomes finite (15 and 9 respectively).



Pick the vertex with minimum distance value and not already included in SPT (not in sptSET). Vertex 6 is picked. So sptSet now becomes {0, 1, 7, 6}. Update the distance values of adjacent vertices of 6. The distance value of vertex 5 and 8 are updated.



We repeat the above steps until sptSet does include all vertices of given graph. Finally, we get the following Shortest Path Tree (SPT).



2.

Bellman – Ford Algorithm

Given a graph and a source vertex src in graph, find shortest paths from src to all vertices in the given graph. The graph may contain negative weight edges.

We have discussed Dijkstra's algorithm for this problem. Dijkstra's algorithm is a Greedy algorithm and time complexity is O(VLogV) (with the use of Fibonacci heap). Dijkstra doesn't work for Graphs with negative weight edges, Bellman-Ford works for such graphs. Bellman-Ford is also simpler than Dijkstra and suites well for distributed systems. But time complexity of Bellman-Ford is O(VE), which is more than Dijkstra.

Algorithm

Following are the detailed steps.

Input: Graph and a source vertex src

Output: Shortest distance to all vertices from src. If there is a negative weight cycle, then shortest distances are not calculated, negative weight cycle is reported.

1) This step initializes distances from the source to all vertices as infinite and distance to the source itself as 0. Create an array dist[] of size |V| with all values as infinite except dist[src] where src is source vertex.

2) This step calculates shortest distances. Do following |V|-1 times where |V| is the number of vertices in given graph.

```
a) Do following for each edge u-v
   If dist[v] > dist[u] + weight of edge uv, then update dist[v]
   dist[v] = dist[u] + weight of edge uv
```

3) This step reports if there is a negative weight cycle in graph. Do following for each edge u-v
……If dist[v] > dist[u] + weight of edge uv, then "Graph contains negative weight cycle"
The idea of step 3 is, step 2 guarantees the shortest distances if the graph doesn't contain a negative weight cycle. If we iterate through all edges one more time and get a shorter path for any vertex, then there is a negative weight cycle

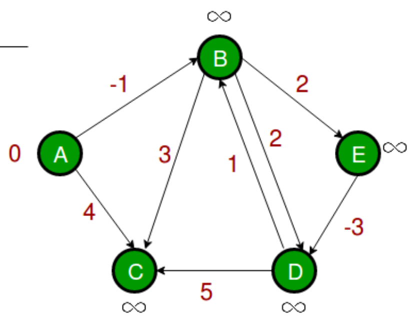
How does this work? Like other Dynamic Programming Problems, the algorithm calculates shortest paths in a bottom-up manner. It first calculates the shortest distances which have at-most one edge in the path. Then, it calculates the shortest paths with at-most 2 edges, and so on. After the i-th iteration of the outer loop, the shortest paths with at most i edges are calculated. There can be maximum |V| – 1 edges in any simple path, that is why the outer loop runs |v| – 1 times. The idea is, assuming that there is no negative weight cycle, if we have calculated shortest paths with at most i edges, then an iteration over all edges guarantees to give shortest path with at-most (i+1) edges

Example

Let us understand the algorithm with following example graph. The images are taken from this source.

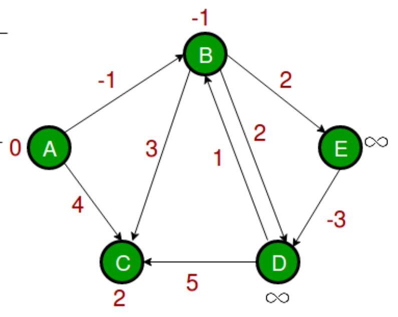
Let the given source vertex be 0. Initialize all distances as infinite, except the distance to the source itself. Total number of vertices in the graph is 5, so all edges must be processed 4 times.

A	B	C	D	E
0	∞	∞	∞	∞



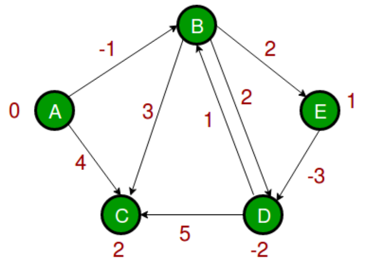
Let all edges are processed in the following order: (B, E), (D, B), (B, D), (A, B), (A, C), (D, C), (B, C), (E, D). We get the following distances when all edges are processed the first time. The first row shows initial distances. The second row shows distances when edges (B, E), (D, B), (B, D) and (A, B) are processed. The third row shows distances when (A, C) is processed. The fourth row shows when (D, C), (B, C) and (E, D) are processed.

A	B	C	D	E
0	∞	∞	∞	∞
0	-1	∞	∞	∞
0	-1	4	∞	∞
0	-1	2	∞	∞



The first iteration guarantees to give all shortest paths which are at most 1 edge long. We get the following distances when all edges are processed second time (The last row shows final values).

A	B	C	D	E
0	∞	∞	∞	∞
0	-1	∞	∞	∞
0	-1	4	∞	∞
0	-1	2	∞	∞
0	-1	2	∞	1
0	-1	2	1	1
0	-1	2	-2	1



The second iteration guarantees to give all shortest paths which are at most 2 edges long. The algorithm processes all edges 2 more times. The distances are minimized after the second iteration, so third and fourth iterations don't update the distances.

PART:- 2

Task:- 2

a.

Abstract data type (ADT)

In computer science, an abstract data type (ADT) is a mathematical model for data types, where a data type is defined by its behavior (semantics) from the point of view of a user of the data, specifically in terms of possible values, possible operations on data of this type, and the behavior of these operations. This contrasts with data structures, which are concrete representations of data, and are the point of view of an implementer, not a user.

Formally, an ADT may be defined as a "class of objects whose logical behavior is defined by a set of values and a set of operations". This is analogous to an algebraic structure in mathematics. What is meant by "behavior" varies by author, with the two main types of formal specifications for behavior being axiomatic (algebraic) specification and an abstract model. These correspond to axiomatic semantics and operational semantics of an abstract machine, respectively. Some authors also include the computational complexity ("cost"), both in terms of time (for computing operations) and space (for representing values). In practice many common data types are not ADTs, as the abstraction is not perfect, and users must be aware of issues like arithmetic overflow that are due to the representation. For example, integers are often stored as fixed width values (32-bit or 64-bit binary numbers), and thus experience integer overflow if the maximum value is exceeded.

Example:

- Integers are an ADT, defined as the values ..., −2, −1, 0, 1, 2, ..., and by the operations of addition, subtraction, multiplication, and division, together with greater than, less than, etc., which behave according to familiar mathematics (with care for integer division), independently of how the integers are represented by the computer. Explicitly, "behavior" includes obeying various axioms (associativity and commutativity of addition etc.), and preconditions on operations (cannot divide by zero). Typically, integers are represented in a data structure as binary numbers, most often as two's complement, but might be binary coded decimal or in ones' complement, but the user is abstracted from the concrete choice of representation, and can simply use the data as data types.

  An ADT consists not only of operations, but also of values of the underlying data and of constraints on the operations. An "interface" typically refers only to the operations, and perhaps some of the constraints on the operations, notably pre-conditions and post conditions, but not other constraints, such as relations between the operations.

- An abstract stack, which is a last-in-first-out structure, could be defined by three operations: push, that inserts a data item onto the stack; pop, that removes a data item from it; and peek or top, that accesses a data item on top of the stack without removal. An abstract queue, which is a first-in-first-out structure, would also have three operations: enqueue, that inserts a data item into the queue; dequeue, that removes the first data item from it; and front, that accesses and serves the first data item in the queue. There would be no way of differentiating these two data types, unless a mathematical constraint is introduced that for a stack specifies that each pop always returns the most recently pushed item that has not been popped yet. When analyzing the efficiency of algorithms that use stacks, one may also specify that all operations take the same time no matter how many data items have been pushed into the stack, and that the stack uses a constant amount of storage for each element.

Why stack is called ADT?

ADT is a data type, just like integers and Booleans primitive data types. An ADT consist not only of operations, but also of values of the underlying data and of constraints on the operations.

A constrains for a stack would be that each pop always returns the most recently pushed item that has not been popped yet.

The actual implementation for an ADT is abstracted away, and we don’t have to care about. So, how a stack is actually implemented is not that important. A stack could be and often is implemented behind the scenes using a dynamic array, but it could be implemented with a linked list instead. It really doesn’t matter.

The bottom line is, … When we say the stack data structure, we refer to how stacks are implemented and arrangement in the memory. But, when we say the stack ADT, we refer to the stack data type which has set of defined operations, the operations constrain, and the possible values.

Also, from another point of view we can say that Stack operates in LIFO order. It has its own attributes. It has its own functions for operations. The representations of those attributes are hidden from, and of no concern to the application code. For e.g.: PUSH (S, x) is enough to push an element x to the top of stack S, without the actual function being defined inside the main function. This is the main concept of ADTs. Hence stack is an abstract data type.

b.

Advantages of encapsulation and information hiding when using an ADT

Data hiding is a software development technique specifically used in object-oriented programming (OOP) to hide internal object details (data members). Data hiding ensures exclusive data access to class members and protects object integrity by preventing unintended or intended changes. Data hiding also reduces system complexity for increased robustness by limiting interdependencies between software components. Data hiding is also known as data encapsulation or information hiding.

Also, data hiding can be introduced as part of the OOP methodology, in which a program is segregated into objects with specific data and functions. This technique enhances a programmer's ability to create classes with unique data sets and functions, avoiding unnecessary penetration from other program classes. Because software architecture techniques rarely differ, there are few data hiding contradictions. Data hiding only hides class data components, whereas data encapsulation hides class data parts and private methods.

Encapsulation is one of the fundamental concepts in object-oriented programming (OOP). It describes the idea of bundling data and methods that work on that data within one unit, e.g., a class in Java. This concept is also often used to hide the internal representation, or state, of an object from the outside. This is called information hiding. The general idea of this mechanism is simple. If we have an attribute that is not visible from the outside of an object, and bundle it with methods that provide read or write access to it, then you can hide specific information and control access to the internal state of the object. In OOP these methods as getter and setter methods. As the names indicate, a getter method retrieves an attribute, and a setter method changes it. Depending on the methods that we implement, we can decide if an attribute can be read and changed, or if it's read-only, or if it is not visible at all.

How ADT helps to maintain encapsulation and data hiding?

When algorithms used to implement each of ADT operation are encapsulated or within ADT. The information hiding feature of abstract data typing that object 'public' interfaces. The interface of an ADT is anything that is directly accessible from outside the ADT's module. The implementation is the internal part of the ADT, which should be hidden from the outside and should only be accessible indirectly via. In general, the interface of an ADT should only consist of public procedures- data should not be accessible. They preventing direct access to an ADT's attributes is known as the data hiding or encapsulations. In java for traffic management system, data hiding involves declaring the content of the ADT's user defined to be private as figure.

The private keyword here applies to the variable declared inside type, preventing them from being accessed outside of the module. The type in self, light() can be used outside the module, because it was declared public but no direct access.

Abstract data types are often called user-defined data types, because they allow programmers to define new types that resemble primitive data types. Just like a primitive type INTEGER with operations +, −, ∗, etc., an abstract data type has a type domain, whose representation is unknown to clients, and a set of operations defined on the domain. Abstract data types were first formulated in their pure form in CLU. The theory of abstract data types is given by existential types. They are also closely related to algebraic specification. In this context the phrase “abstract type” can be taken to mean that there is a type that is “conceived apart from concrete realities”. Object-oriented programming involves the construction of objects which have a collection of methods, or procedures, that share access to private local state. Objects resemble machines or other things in the real world more than any well-known mathematical concept. In this tutorial, Smalltalk is taken as the paradigmatic object-oriented language. A useful theory of objects associates them with some form of closure, although other models are possible. The term “object” is not very descriptive of the use of collections of procedures to implement a data abstraction. Thus we adopt the term procedural data abstraction as a more precise name for a technique that uses procedures as abstract data.

c.

Discussion of imperative ADTs with regard to object orientation

ADT is a container which holds different types of objects with specifications. logical representation (i.e. an interface or protocol) of the data and the operations to manipulate the component elements of the data.

Examples of ADT: List, Map, Set, Stack, Queue, Tree, Graph.

Data structures can implement one or more particular abstract data types (ADT). In java for example Array List, Linked List, Stack and Vector are data structures implementation(classes) of List.

Stack examples in real life

- When a person wear bangles, the last bangle worn is the first one to be removed and the first bangle would be the last to be removed. This follows last in first out (LIFO) principle of stack.
- In a stack of plates, once can take out the plate from top or can keep plate at the top. The plate that was placed first would be the last to take out. This follows the LIFO principle of stack.
- Batteries in the flashlight: - We can’t remove the second battery unless you remove the last in. So, the battery that was put in first would be the last one to take out. This follows the LIFO principle of stack.
- Clothes in the trunk

Queue examples in real life

- A queue of people at ticket-window: The person who comes first gets the ticket first. The person who is coming last is getting the tickets in last. Therefore, it follows first-in-firstout (FIFO) strategy of queue.
- Vehicles on toll-tax bridge: The vehicle that comes first to the toll tax booth leaves the booth first. The vehicle that comes last leaves last. Therefore, it follows first-in-first-out (FIFO) strategy of queue.
- Luggage checking machine: Luggage checking machine checks the luggage first that comes first. Therefore, it follows FIFO principle of queue.
- Patients waiting outside the doctor's clinic: The patient who comes first visits the doctor first, and the patient who comes last visits the doctor last. Therefore, it follows the firstin-first-out (FIFO) strategy of queue.

Is it true that abstract data types (ADT) is one of the basic ideas behind object orientation?

According to the Smalltalk definition of object-oriented programming, then no. According to the definition that descends from Simulate - the one shared by Java, C++ and C# - where classes and inheritance are essential and not optional implementation details, then that version of OOP has some support for the definition of ADTs.

An object is essentially defined by the names of the messages it receives (the names of its methods in the Simulate version) and the values it encapsulates. The behavior of those methods is changeable; we don't know what is actually going to be done in response to a message (a method call), although it's reasonable to expect a returned value of a given type (in Simulate-style OOP, that at least can be guaranteed).

An abstract data type is defined by the values that it can operate on and the operations that can be performed on them. Inheritance is not part of the concept. Parametric polymorphism is required. Encapsulation is required but there is no concept of an object containing its own operations; the semantics of the operations are an essential part of the definition of an ADT and not alterable.

In Smalltalk and its descendants, we can model ADTs dynamically but there's no way to define them.

Java's design was influenced by CLU - a language created by one of the people who gave ADTs their name and formal definition, designed explicitly to enable programming with ADTs. Java's private methods, final methods, generics and interfaces allow you to create things that are effectively ADTs. It still offers no distinct concept of an ADT - it emerges from the choices you make in defining an abstract class or interface.

C++ templates were also inspired by CLU - but then C++ templates are not at all an object-oriented concept, but a concept based on parametric polymorphism.

Short answer: no, although the concept influenced the design of many of the later languages to which the label object-oriented has been attached.

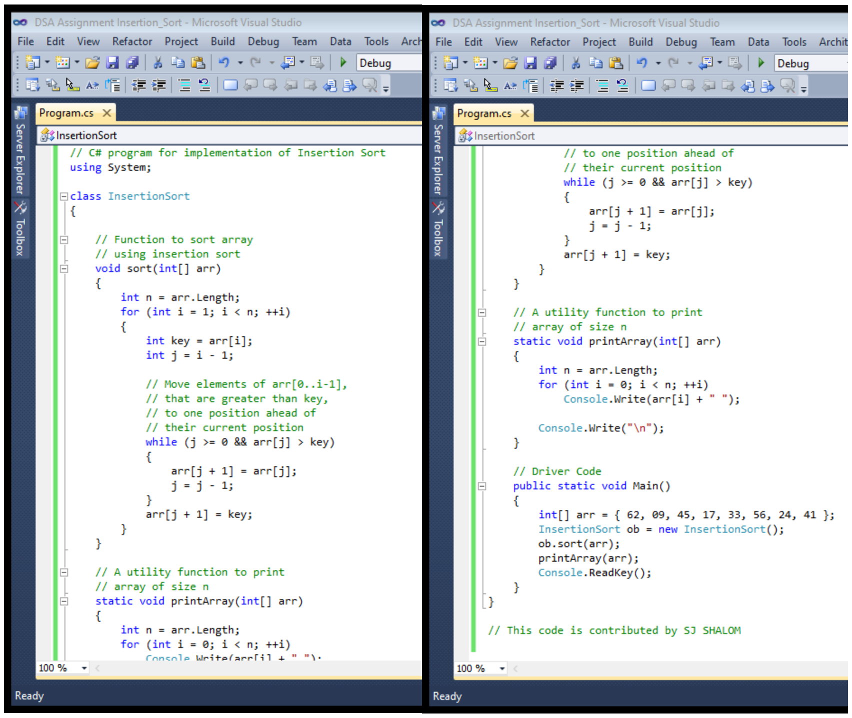
Also, it is the basic idea behind languages like C++, C#, and Java. Since I was asked to answer, I will say that I do not consider these OO languages. They are procedural languages with abstraction capabilities.

ADTs are not one of the basic ideas behind OOP. The basic ideas of OOP are encapsulation(local retention, protection, and hiding of state-process), late-binding in all things, and messaging.

PART:- 3

Task:- 3

Implement a complex ADT and algorithm in an executable programming language to solve a well-defined problem

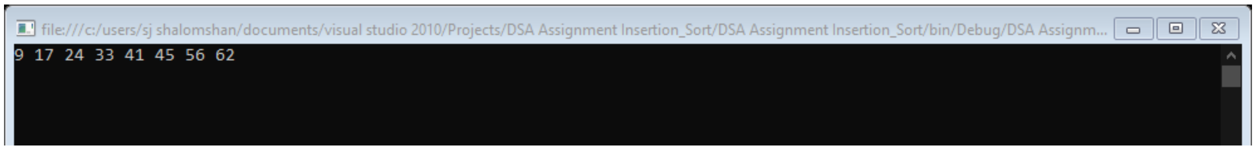


DSA Assignment Insertion_Sort - Microsoft Visual Studio
File Edit View Refactor Project Build Debug Team Data Tools Arch
Debug
Program.cs
InsertionSort
Server Explorer
Toolbox
100 %
Ready

```
// C# program for implementation of Insertion Sort
using System;

class InsertionSort
{

    // Function to sort array
    // using insertion sort
    void sort(int[] arr)
    {
        int n = arr.Length;
        for (int i = 1; i < n; ++i)
        {
            int key = arr[i];
            int j = i - 1;

            // Move elements of arr[0..i-1],
            // that are greater than key,
            // to one position ahead of
            // their current position
            while (j >= 0 && arr[j] > key)
            {
                arr[j + 1] = arr[j];
                j = j - 1;
            }
            arr[j + 1] = key;
        }
    }

    // A utility function to print
    // array of size n
    static void printArray(int[] arr)
    {
        int n = arr.Length;
        for (int i = 0; i < n; ++i)
```

```
            // to one position ahead of
            // their current position
            while (j >= 0 && arr[j] > key)
            {
                arr[j + 1] = arr[j];
                j = j - 1;
            }
            arr[j + 1] = key;
        }
    }

    // A utility function to print
    // array of size n
    static void printArray(int[] arr)
    {
        int n = arr.Length;
        for (int i = 0; i < n; ++i)
            Console.Write(arr[i] + " ");

        Console.Write("\n");
    }

    // Driver Code
    public static void Main()
    {
        int[] arr = { 62, 09, 45, 17, 33, 56, 24, 41 };
        InsertionSort ob = new InsertionSort();
        ob.sort(arr);
        printArray(arr);
        Console.ReadKey();
    }
}

// This code is contributed by SJ SHALOM
```

Figure 11 Insertion sort



file:///c:/users/sj shalomshan/documents/visual studio 2010/Projects/DSA Assignment Insertion_Sort/DSA Assignment Insertion_Sort/bin/Debug/DSA Assignm...

```
9 17 24 33 41 45 56 62
```

Figure 12 Console WriteLine

Insertion sort takes maximum time to sort if elements are sorted in reverse order. And it takes minimum time when elements are sorted.

Uses: Insertion sort is used when numbers of elements is small. It can also be useful when input array is almost sorted only few elements are misplaced in completed big array.

How would you demonstrate how the implementation of the ADT/algorithm solves a well-defined problem?

Insertion sort is simple sorting algorithm that works similar to the way you sort playing cards in your hands. The array is virtually split into a sorted and unsorted part. Values from the unsorted part are picked and placed at the correct position in the sorted part.

Algorithm

To sort an array of size n in ascending order:

1. Iterate from arr[1] to arr[n] over the array.
2. Compare the current element (key) to its predecessor.
3. If the key element is smaller than its predecessor, compare it to the elements before. Move the greater elements one position up to make space for the swapped element.

Show how to critically evaluate the complexity of the implemented ADT/algorithm.

As a software developer working for Code Works and technical project leader my role informs them about the design and implementation abstract data types. In this report our client E-Cube Ltd will be able to understand about utilization of abstract data types, there algorithms and complexity of implementing algorithm in the application.

Abstract data types and algorithm:

- The abstract data types and algorithms are concept which comes under Data Structures. As, we know that more data is generated daily and more complex the database are getting so for storing and searching of data efficiently there is concept in programming called Data Structures which also helps processor speed and handling multiple request at a time.
- **Abstract data types:** An abstract data types which focuses on operation implemented in program or application and how is implemented or what algorithm should use in implementation. There are two categories of data types given below:
    - **Dependent:** Which are Array and Linked List.
    - **Independent:** Which is Stack, Queue, Tree, Graph, Hash and many other.
- Algorithm: Algorithms are step-by-step instruction of execution of program in an orderly manner. Algorithm is an independent to any programming language. There different types of algorithm:
    - Search
    - Sort
    - Insert
    - Delete

Define how to Implement error handling and report test results.

A large amount of the code that application developers write is purely there to handle error conditions. The actual amount of error handling code that is written will depend greatly on the particular application. However, it has been estimated that up to 90% of an application's code is related to handling exceptional or error conditions. This is therefore an important area of API design that will be used frequently by your clients. In fact, it is included in Ken Pugh's Three Laws of Interfaces:

1. An interface's implementation shall do what its methods say it does.
2. An interface's implementation shall do no harm.
3. If an interface's implementation is unable to perform its responsibilities, it shall notify its caller.

Accordingly, the three main ways of dealing with error conditions in your API are

1. Returning error codes.
2. Throwing exceptions.
3. Aborting the program.

The last of these is an extreme course of action that should be avoided at all costs—and indeed it violates the third of Pugh's three laws—although there are far too many examples of libraries out there that call abort() or exit(). As for the first two cases, different engineers have different proclivities toward each of these techniques. I will not take a side on the exceptions versus error code debate here, but rather I'll attempt to present impartially the arguments and drawbacks for each option. Whichever technique you select for your API, the most important issues are that you use a consistent error reporting scheme and that it is well documented.

PART:- 4

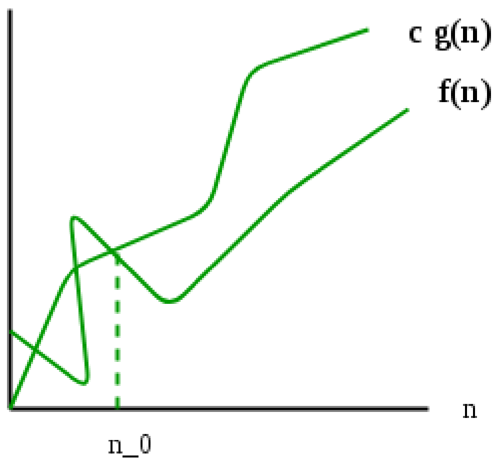
Task:- 4

a.

Asymptotic Analysis

Asymptotic analysis of an algorithm is defining the mathematical framing its run-time performance. Using asymptotic analysis, we can conclude the best case, average case and worst-case scenario of an algorithm. The big idea that can handles above issues in analyzing algorithms. In Asymptotic Analysis, we evaluate the performance of an algorithm in terms of input size but we don't measure the actual running time. We calculate, how does the time or space take by an algorithm increases with the input size.

For example, the running time of one operation is computed as f(n) and may be for another operation it is computed as g(n2). This means the first operation running time will increase linearly with the increase in n and the running time of the second operation will increase exponentially when n increases. Similarly, the running time of both operations will be nearly the same if n is significantly small.



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

How it can be used to find effectiveness of an algorithm

They are finding out more effectiveness of an algorithm can be used so we time required by algorithm falls under the three types: Worst case maximum time required by an algorithm and itis mostly or done while analyzing the algorithm.

The commonly used notation for calculating the running time complexity of the algorithm as download

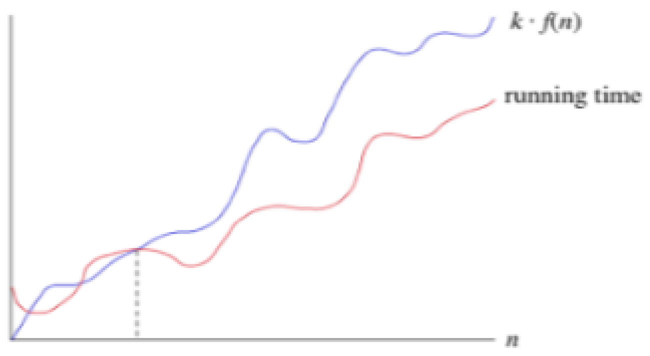
- Big O notation
- Big θ notation
- Big notation

Big Oh Notation, O

Big O is used to measure the performance or complexity of an algorithm. In mathematical term itis upper bound of growth rate of a function, or that if a function g(x) grows no faster than a function f(x), then g is said to be member of O(f). In general, it used to express the upper bound of an algorithm and which gives the measure for the worst time complexity or the longest time an algorithm possibly takes to complete. Big-oh is the formal method of mention the upper bound of an algorithm's running time. It is the measure of the longest amount of time. The function f (n) = O (g (n)) [read as "f of n is big-oh of g of n"] if and only if exist positive constant c and such that

F(n) K.g (n)f(n) ⩽ ⩽K.g(n) for n>n0n>no in all case

Hence, function g (n) is upper bound for function f (n), as g (n) grows faster than f (n)



ASYMPTOTIC UPPER BOUND

For Example:

1. 3n+2=O(n) as 3n+2≤4n for all n≤2

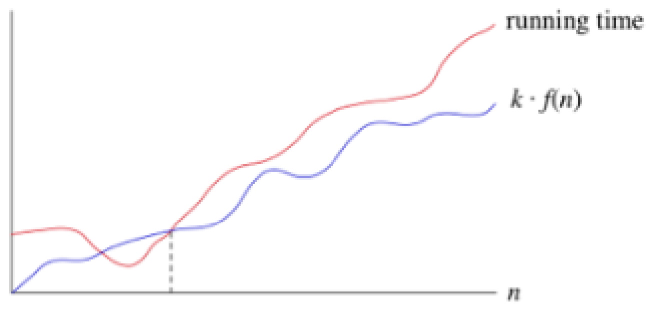
2. 3n+3=O(n) as 3n+3≤4n for all n>3

Hence, the complexity of f(n) can be represented as O (g (n))

Big Omega Notation, Ω

The notation Ω (n) is the formal way to express the lower bound of an algorithm's running time. It measures the best-case time complexity or the best amount time an algorithm can possibly take to complete. The function f (n) = Ω (g (n)) ["f of n is omega of g of n"] if and only if there exists positive constant c and no such that.

F (n) ≥ k * g (n) for all n, n ≥ no



ASYMPTOTIC LOWER BOUND

For Example:

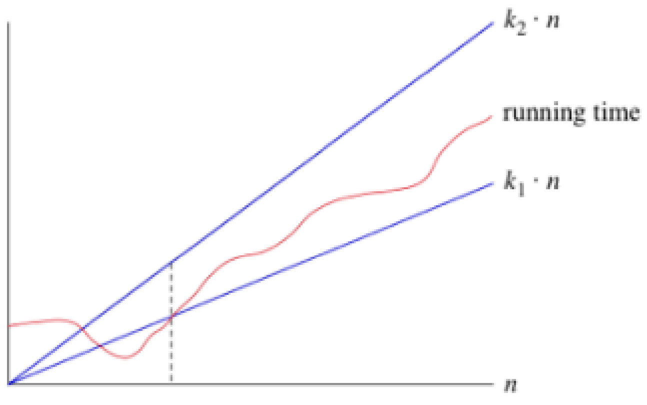
F (n) = 8n2 + 2n – 3 ≥ 8n2 – 3

= 7n2 + (n2 - 3) ≥ 7n2 (g(n))

K1 = 7

Big Theta Notation, θ

The notation θ (n) us the formal way to express the lower bound of an algorithm running time.It measures the best-case time complexity or the best amount of time an algorithm can possibly take to complete. The function f (n) = θ (g (n)) [“f is the theta of g of n”] if and only if there exists positive constant k1, k2 and k0 such that



ASYMPTOTIC TIGHT BOUND

For Example:

3n + 2 = θ (n) as 3n + 2 ≤ 3n and 3n + 2 ≤ 4n, for n

k1 = 3, k2 = 4 and n0 = 2

Hence, the complexity of f (n) can be represented as θ (g (n)).

The Theta Notation is precise than both the big – oh and Omega notation. The function f (n) = function f (n) = θ (g (n)) if g(n) is both an upper and lower bound.

They more efficiency of an algorithm can be measured can performs better than second so inputs second perform better. They can work better on machine in inputs such as

1. Time complexity
2. Space complexity
3. Complexity theory

Time complexity

Time complexity of algorithm is commonly used big O notation. It is an asymptotic notation to represent of time complexity. Time complexity estimates how an algorithm performs any how kind of machine it runs on. We can get the time complexity by counting the number of operations can performed in code. This time complexity is defined as function of the input size n using Big O notation n indicates the magnitude of input while O is growth rate in function. Big-) notation based in running time or space (memory used) as input grow rate. The O function is growth rate in function of input size n.

Space complexity

Space complexity is an algorithm is total space take by algorithm with regard to input size. Space complexity both auxiliary space and space used by input. They execute an algorithm can be loaded in memory. The memory can be used in different forms are

1. Variable
2. Program instruction
3. Execution

Memory usage during program execution

1. Instruction Space can used to store the addresses while a module calls another module or functions during execution
2. 2.Data space can use to store data, variables and constants which ae stored by the program and updated during execution.

Complexity theory

Complexity theory is direct application to computability theory and use of computation model such as Turing machines to help test the complexity. Complexity theory helps for programmer relate and group problems together into complexity classes. If one problem can solve, it opens a way to solve a particular problem. For example, some problems can solve in polynomial amounts of time and others take exponential amounts of time to the input the size.

They more efficiency of an algorithm can be measured can performs better than second so inputs second perform better. They can work better on machine in inputs such as

1. Time complexity
2. Space complexity
3. Complexity theory

Time complexity

Time complexity of algorithm is commonly used big O notation. It is an asymptotic notation to represent of time complexity. Time complexity estimates how an algorithm performs any how kind of machine it runs on. We can get the time complexity by counting the number of operations can performed in code. This time complexity is defined as function of the input size n using Big-O notation n indicates the magnitude of input while O is growth rate in function. Big-) notation based in running time or space (memory used) as input grow rate. The O function is growth rate in function of input size n.

Space complexity

Space complexity is an algorithm is total space take by algorithm with regard to input size. Space complexity both auxiliary space and space used by input. They execute an algorithm can be loaded in memory. The memory can be used in different forms are

1. Variable
2. Program instruction
3. Execution

Memory usage during program execution

1. Instruction Space can used to store the addresses while a module calls another module or functions during execution
2. Data space can use to store data, variables and constants which ae stored by the program and updated during execution.

b.

Clearly interpret what a trade-off is when specifying an ADT using an example to support your answer

What is the cost when I use ADT

Formally, an ADT may be defined as a "class of objects whose logical behavior is defined by a set of values and a set of operations" this is analogous to an algebraic structure in mathematics. What is meant by "behavior" varies by author, with the two main types of formal specifications for behavior being axiomatic (algebraic) specification and an abstract model these correspond to axiomatic semantics and operational semantics of an abstract machine, respectively. Some authors also include the computational complexity ("cost"), in terms of both time (for computing operations) and space (for representing values). In practice, many common data types are not ADTs, as the abstraction is not perfect, and users must be aware of issues like arithmetic overflow that are due to the representation. For example, integers are often stored as fixed width values (32-bit or 64-bit binary numbers), and thus experience integer overflow if the maximum value is exceeded.

Time Complexity

In computer science, the time complexity of an algorithm quantifies the amount of time taken by an algorithm to run as a function of the length of the string representing the input. The time complexity of an algorithm is commonly expressed using big O notation, which excludes coefficients and lower order terms. Time complexity is commonly estimated by counting the number of elementary operations performed by the algorithm, where an elementary operation takes a fixed amount of time to perform. Thus, the amount of time taken and the number of elementary operations performed by the algorithm differ by at most a constant factor.

We will consider the growth rate of some familiar operations, based on this chart; we can visualize the difference of an algorithm with O (1) when compared with O (n2). As the input larger and larger, the growth rate of some operations stays steady, but some grow further as a straight line, some operations in the rest part grow as exponential, quadratic, factorial.

Space Complexity

Space complexity of an algorithm represents the amount of memory space required by the algorithm in its life cycle. The Space complexity will equal to the amount of memory required by an algorithm to run to completion. Some algorithms may be more efficient if data completely loaded into memory

The space required by an algorithm is equal to the sum of the following two components:

- A fixed part: a space required to store certain data and variables that are independent of the size of the problem. For example, simple variables and constants used, program size, etc.
- A variable part: a space required by variables; whose size depends on the size of the problem. For example, dynamic memory allocation, recursion stack space, etc.

Example of using time and space complexity to measure the algorithm:

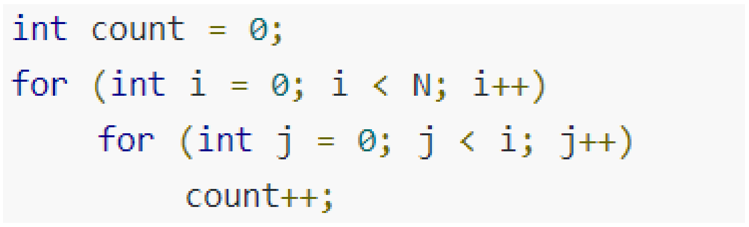
As an illustration, depend on the time & space complexity table above, we can easily find out that the Bubble Sort has Time Complexity (Best-case: O (n); Worst-case: O (n2); Average-case: O (n2)) and Space Complexity: O (1). However, we need to understand the reason why the table have that result. Thus, let us get started with the implementation of bubble sort in Java programming language.

To calculate the complexity of the bubble sort algorithm, it is useful to determine how many comparisons each loop performs. For each element in the array, bubble sort does (n−1) comparisons. In big O notation, bubble sort performs O (n) comparisons. Because the array contains (n) elements, it has an O (n) number of elements. In other words, bubble sort performs O (n) operations on an O (n) number of elements, leading to a total running time of O (n2).

Note: O (n) is the best-case running time for bubble sort. It is possible to modify bubble sort to keep track of the number of swaps it performs. If an array is already in sorted order, and bubble sort makes no swaps, the algorithm can terminate after one pass. With this modification, if bubble sort encounters a list that is already sorted, it will finish in O (n) time.

Though bubble sort is simple and easy to implement, it is highly impractical for solving most problems due to its slow running time. It has an average and worst-case running time of O (n2), and can only run in its best- case running time of O (n) when the input list is already sorted. In addition, Bubble sort is a stable sort with a space complexity of O (1).

Let me consider some example:

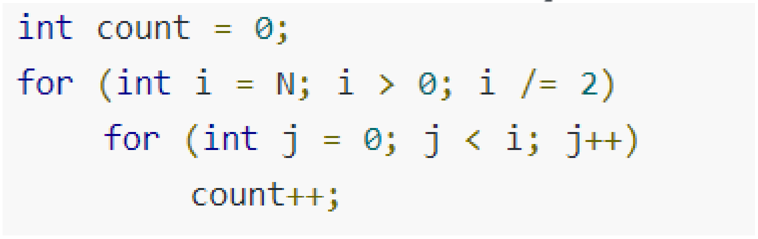


```
int count = 0;
for (int i = 0; i < N; i++)
    for (int j = 0; j < i; j++)
        count++;
```

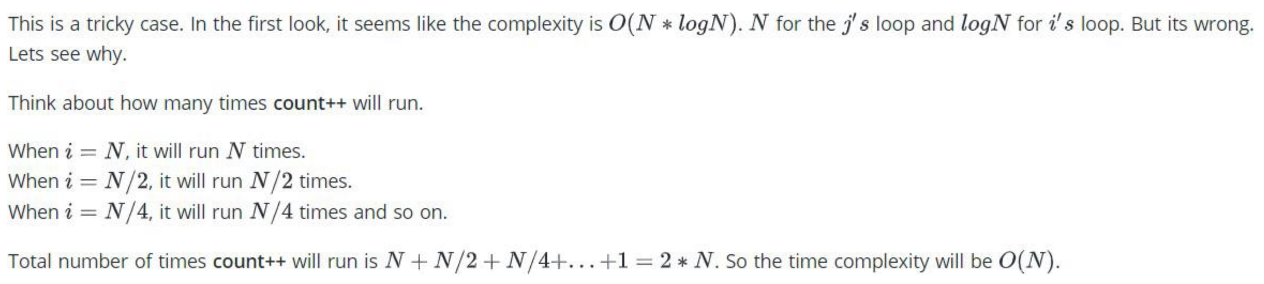
Lets see how many times **count++** will run.

When $i = 0$, it will run 0 times.
When $i = 1$, it will run 1 times.
When $i = 2$, it will run 2 times and so on.

Total number of times **count++** will run is $0 + 1 + 2 + \ldots + (N-1) = \frac{N*(N-1)}{2}$. So the time complexity will be $O(N^2)$.



```
int count = 0;
for (int i = N; i > 0; i /= 2)
    for (int j = 0; j < i; j++)
        count++;
```



This is a tricky case. In the first look, it seems like the complexity is $O(N * logN)$. N for the $j's$ loop and $logN$ for $i's$ loop. But its wrong. Lets see why.

Think about how many times **count++** will run.

When $i = N$, it will run N times.
When $i = N/2$, it will run $N/2$ times.
When $i = N/4$, it will run $N/4$ times and so on.

Total number of times **count++** will run is $N + N/2 + N/4 + \ldots + 1 = 2 * N$. So the time complexity will be $O(N)$.

What are disadvantages of using ADT instead of other data types

Reasoning about ADTs

There is another major downside to using ADTs — they severely affect theorem proving. Consider the following proof that 0 + n = n + 0 using Coq's standard library natural numbers. After the induction, the proof is almost automatic due to Coq's ability to computationally simplify (cbn) the goal.

Compare this to the proof, which uses the ADT, where we have to perform equational reasoning instead of computational reasoning. This is extremely burdensome, especially for more proofs that are complicated.

- ADTs do not support fix/match syntax.
- ADTs have to expose derived operations.
- ADTs prohibit computational reasoning.

c.

Three benefits of using implementation independent data structures

Representation Independence

Most of the program becomes independent of the abstract data type's representation, so that representation can be improved without breaking the entire program.

Modularity

With representation independence, the different parts of a program become less dependent on other parts and on how those other parts are implemented.

In Interchangeability of Parts

Different implementations of an abstract data type may have different performance characteristics. With abstract data types, it becomes easier for each part of a program to use an implementation of its data types that will be more efficient for that particular part of the program.

Example:

- Java's standard libraries supply several different implementations of its Map data type.
- The Tree Map implementation might be more efficient when a total ordering on the keys can be computed quickly but a good hash value is hard to compute efficiently.
- The HashMap implementation might be more efficient when hash values can be computed quickly and there is no obvious ordering on keys.
- The part of a program that creates a Map can decide which implementation to use.
- The parts of a program that deal with a created Map don't have to know how it was implemented once created, it's just a Map.

If it weren't for abstract data types, every part of the program that uses a Map would have to be written twice, with one version to deal with Tree Map implementations and another version to deal with HashMap implementations.

d.

Precisely determine two ways in which the efficiency of an algorithm can be measured, illustrating your answer with an example

Two main measures for the efficiency of an algorithm are:

- Processor and Memory
- Complexity and Capacity
- Time and Space
- Data and Space

Time complexity - a measure of amount of time for an algorithm to execute.

Examples:

Time complexity for Linear search can be represented as O(n) and O(log n) for Binary search (where, n and log(n) are the number of operations).

Time complexity or Big O notations for some popular algorithms are:

1. Binary Search: O(log n)
2. Linear Search: O(n)
3. Quick Sort: O(n * log n)
4. Selection Sort: O(n * n)
5. Travelling salesperson : O(n!)

Conclusion

In conclusion, this assignment has more detailed information about the Data Structure & Algorithm concept. As a student I have gained a lot of knowledge about Data Structure & Algorithm and other essentials things about Data Structure & Algorithm concept.

In this assignment, I have done a lot of things which are very useful to improve my knowledge. Specially I have learned to abstract data types, concrete data structure & algorithm

I have learned so many things about the Data Structure & Algorithm and ADT other essential things. I have improved a lot of skills in testing. This assignment taught me a lot of things.

Finally, I express my deepest thanks to the assessor Eng. A. L. Jubailah Begum. Who gave this awesome opportunity to improve my Data Structure & Algorithm knowledge.

Reference

https://www.tutorialspoint.com/data_structures_algorithms/dsa_quick_guide.htm

https://www.coursehero.com/file/p5daec09/Illustrates-with-an-example-a-concrete-data-structure-for-a-First-In-First-out/

https://study.com/academy/lesson/stacks-in-computer-memory-definition-uses.html

https://www.geeksforgeeks.org/comparison-among-bubble-sort-selection-sort-and-insertion-sort/

https://www.geeksforgeeks.org/dijkstras-shortest-path-algorithm-greedy-algo-7/

https://www.geeksforgeeks.org/bellman-ford-algorithm-dp-23/

https://www.wisdomjobs.com/e-university/data-structures-tutorial-290/abstract-data-type-7124.html

https://www.quora.com/What-are-the-advantages-of-using-encapsulation-and-information-hiding-when-using-abstract-data-types-ADT

https://www.coursehero.com/file/p7nfutu5/HNCHND-Computing-2-Learning-Outcomes-and-Assessment-Criteria-Pass-Merit/

https://www.geeksforgeeks.org/analysis-of-algorithms-set-1-asymptotic-analysis/#:~:text=In%20Asymptotic%20Analysis%2C%20we%20evaluate,item)%20in%20a%20sorted%20array.

https://www.coursehero.com/file/p3f54g1/Interpret-what-a-trade-off-is-when-specifying-an-ADT-using-an-example-to/

https://www.coursehero.com/tutors-problems/Java-Programming/25234637-Evaluate-three-benefits-of-using-implementation-independent-data-struc/

https://www.researchgate.net/post/How_can_I_measure_the_performance_of_an_algorithm