Addis Ababa University

College of Natural and Computational Sciences

Department of Computer Science

Programming and Algorithms Module

Part I: Computer Programming

Part II: Object Oriented Programming

Part III: Design and Analysis of Algorithms

Part IV: Data Structure and Algorithms

May 2024

Addis Ababa, Ethiopia

Data Structures and Algorithms Analysis

PART One: Introduction

A Computer program is a solution to a problem which contains two basic entities:

- The way data are organized in a computer's memory which referred as *Data Structure* and
- The sequence of computational steps to solve a problem is said to be an *algorithm*.

Therefore, *"program = algorithm + data structure".* In programming the first step is known as ***abstraction*** which is to have one's own personal view (developing a model) of the problem. **Abstraction** is also the process of classifying characteristics as relevant and irrelevant for the particular purpose at hand and ignoring the irrelevant ones.

Classification of data structure:

1. **Primitive data structures:** data item operated closest to the machine level instruction.
2. **Non-Primitive data structures:** data item are not operated closest to the machine level instruction.
    a. **Linear data structure:** data items stores in sequence
       Example: Array, stack and Queue,
    b. **Non-Linear data structures** : in which there is order of data items
       Example: Trees and Graphs

Operations performed on any linear data structure are:

- *Traversal – Processing each element in the list*
- *Search* – Finding the location of the element with a given value.
- *Insertion* – Adding a new element to the list.
- *Deletion* – Removing an element from the list.
- *Sorting* – Arranging the elements in some type of order.
- *Merging* – Combining two lists into a single list.

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

- Input:there are zero or more quantities which are externally supplied;
- Output: at least one quantity is produced;
- Definiteness: each instruction must be clear and unambiguous;
- Finiteness: if we trace out the instructions of an algorithm, then for all cases the algorithm will terminate after a finite number of steps;

Algorithm analysis refers to the process of determining the amount of computing time and storage space required by different algorithms. It used to produce a function T (n) that describes the algorithm in terms of the operations performed in order to measure the complexity of the algorithm.

Complexity Analysis is the systematic study of the cost of computation, measured either in time units or in operations performed, or in the amount of storage space required.

Asymptotic analysis is concerned with how the running time of an algorithm increases with the size of the input in the limit, as the size of the input increases without bound. There are five notations used to describe a running time function. ***These are:***

I. **Big-Oh Notation (O):** It's only concerned with what happens for very a large value of n. Big-O expresses an *upper bound* on the growth rate of a function, for sufficiently large values of n. An *upper bound* is the best algorithmic solution that has been found for a problem. **NB**: Each of the following functions is big-O of its successors:

```
K (Constant) ➔ log_b n ➔ n ➔nlog_b n  ➔ n² ➔ n to higher powers
➔ 2ⁿ ➔ 3ⁿ ➔ larger constants to the nᵗʰ power ➔ n! ➔ nⁿ
```

II. **Big-Omega Notation (Ω):** provides an asymptotic lower bound; f(n)= Ω(g (n)) means that the growth rate of f(n) is greater than or equal to g(n).

III. **Theta Notation (Θ):** If f(n)= Θ (g(n)), then g(n) is an asymptotically tight bound for f(n) i.e f(n) and g(n) have the same rate of growth.

IV. **Little-o Notation (o):** f(n)=o(g(n)) means f(n) has less growth rate compared to g(n).

V. **Little-Omega Notation (ω):** ω notation used to denote a lower bound that is not asymptotically tight.

PART TWO: Simple Search and Sort algorithms

Searching: looking for a specific element in a list of items or determining that the item is not in the list. There are two simple search algorithms:

Linear/Sequential searching	**Binary Searching**
• Works with **unsorted** List • Loop through the list starting at the first element • Time is proportional to the size of input (*n)* and we call this time **complexity *O(n)*.**	• Works on **ordered**/sorted list • Uses divide and conquer method: it locates mid-point and identify which have the target belongs to since the list is sorted. It performs this recursively till the item found or the list is empty(target is not in the list • The computational time for this algorithm is proportional to **$\log_2 n$.**

Sorting: the process of reordering a list of items in either increasing or decreasing order. It is also one of the most important operations performed by computers.

A. Insertion Sort

The most instinctive type of sorting algorithm; it finds the location for an element and move all others up, and insert the element to its right position. It starts with the left most two items and put them in order. Then add 3rd item to the relatively sorted portion and find its correct location (relative to each other) swapping adjacent elements if they are out of order. In such it sorts all item in the list.

- How many comparisons? => $O(n^2)$ and How many swaps? => $O(n^2)$

B. Selection Sort

Loop through the list/array from i=0 to n-1 and select the smallest element in the list and swap this value with value at position i. Repeat this step for the rest of the items.

- How many comparisons? => $O(n^2)$ and How many swaps? => O(n)

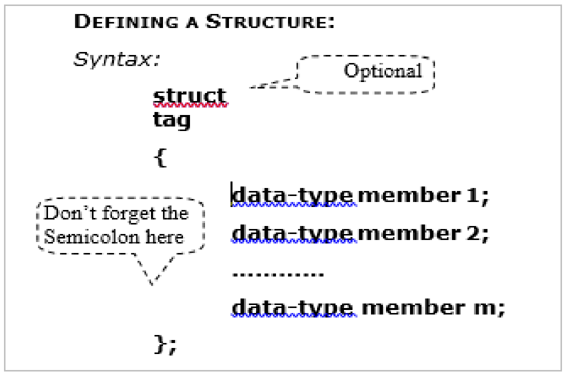
C. Bubble Sort

The simplest to implement but slowest algorithm on very large inputs. The algorithm loops through the array from i=0 to n and swap adjacent elements if they are out of order *bubbling or pushing* out the **smallest** or **largest** element to its proper position.

- How many comparisons? => O(n^2) and How many swaps? => O(n^2)

PART THREE: Structures

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



DEFINING A STRUCTURE:

Syntax:

```
struct
tag                         Optional
{
                    data-type member 1;
Don't forget the    data-type member 2;
Semicolon here      ............
                    data-type member m;
};
```

Here, *struct* is the required keyword; *tag* (optional) is a name that identifies structures of this type; and member1, meber2, …, member m are individual member declarations.

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

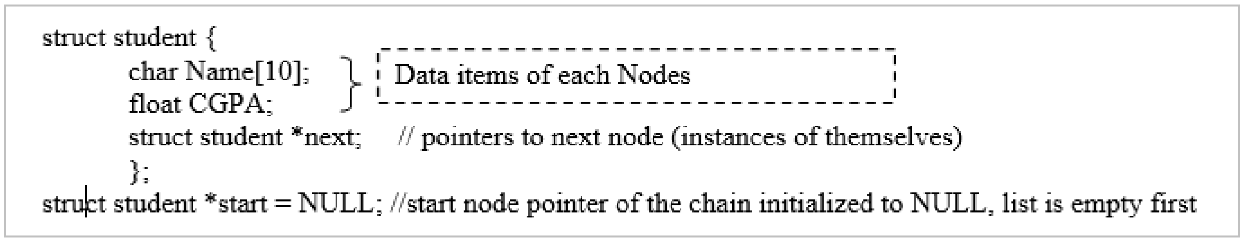
1. Single Linked lists

It is a self-referential structure (Structures can hold pointers to instances of themselves), that allow to create a chain of structs/Nodes with related data. Each node in the list contains data items and pointer to next node.

Basic Operations on Single Linked List

I. Creating Linked Lists

Structure definition of a single linked list need to contain both data items declaration and next node pointer. In addition the first node pointer is essential to keep track of the list to perform various operations on the list. Thus, it is initialized to NULL first as the list is empty before any node added to the chain.



```
struct student {
        char Name[10];   }  Data items of each Nodes
        float CGPA;
        struct student *next;    // pointers to next node (instances of themselves)
        };
struct student *start = NULL; //start node pointer of the chain initialized to NULL, list is empty first
```

II. Adding node to the list

While adding a node to a list, the chain needs to be rearranged properly so that the each nodes are connected to each other. It is possible to add a node anywhere in the chain (at the start, middle or end). Let's consider adding a node at the end of the list:

Step 1: First we need to reserve a space for the new node using the keyword **new.**

*Node *temp = new *Node;* // Reserving a space for temporary pointer which points to new node;

Step 2: Then using the above declared node pointer acquire the data for the new node

```
cin >> temp->Name;
cin >> temp->CGPA;
temp->next = NULL;  // Indicates that this node will be the last node in the list.
```

Step 3: Then add the node at the end of the list. For this we need to check whether the list is empty or not. If list is empty the new node will be the first (Start) node, otherwise navigate forward starting form the first node till last node found and add it next to this node which make the new node the last node.

```
if (start == NULL)    // If list is empty, then new node become start node
  start = temp;
else {
  temp2 = start; // list not empty! Thus temp2 starts from first node
  while (temp2->next!= NULL) { // till the current last node found navigates forward
          temp2 = temp2->next;      // Move to next link in chain
   }
  temp2->next = temp;  // new node (temp) appended at the end of the list
 }
```

III. Displaying the list of Nodes

After adding nodes to the list, we need to display the list of nodes on the screen. Here are the steps required to display all nodes item in the list:

1. Set a temporary pointer to point to the same thing as the start pointer.
2. If the pointer points to **NULL**, display the message "End of list" and stop.
3. Otherwise, display the details of the node pointed to by the start pointer.
4. Make the temporary pointer point to the same thing as the **next** pointer of the node it is currently indicating.
5. Jump back to step 2.

IV. Deleting a node from the list

When a node is deleted, the space that it took up should be reclaimed. Otherwise the computer will eventually run out of memory space. This is done with the **delete** instruction:

```
delete temp;    // Release the memory pointed to by temp
```

However, we can't just delete the nodes willy-nilly as it would break the chain. We need to reassign the pointers and then delete the node at the last moment. We can delete nodes at the start, middle or end of the list. Here let's see the steps to delete at the end.

1. Look at the start pointer. If it is NULL, then the list is empty, so print out a "No nodes to delete" message.
2. Make temp1 point to whatever the start pointer is pointing to.
3. If the next pointer of what temp1 indicates is NULL, then we've found the last node of the list, so jump to step 7.
4. Make another pointer, temp2, point to the current node in the list.
5. Make temp1 point to the next item in the list.
6. Go to step 3.
7. If you get this far, then the temporary pointer, temp1, should point to the last item in the list and the other temporary pointer, temp2, should point to the last-but-one item.
8. Delete the node pointed to by temp1.
9. Mark the next pointer of the node pointed to by temp2 as NULL - it is the new last node.

2. STACK :

"A *stack* is an ordered list in which all insertions and deletions are made at one end, called the *top"*. Stacks are sometimes referred to as Last in First out (LIFO) lists. Stacks have some useful terminology associated with them:

- **Push** To add an element to the stack
- **Pop** To remove an element from the stock
- **Peek** To look at elements in the stack without removing them
- **LIFO** Refers to the last in, first out behavior of thestack
- **FILO (First in Last Out)** : equivalent to LIFO

Implementation of stack:

I. Array implementation (static memory).

Push(n): Adding item at top	**Pop(): removing item from stack top**

top is the current top of stack and n is its maximum size
Begin
  if (top == n) then stack is full;
  else
  top := top+1;
stack[top] := item;
end:
	remove top element from the stack and put it in the item
	Begin:
	  if (top == 0) then stack is empty;
	  else
	    item := stack[top];
	    top := top-1;
	end;

II. Linked list (dynamic memory)

The linked list implementation of stack is adding items at the end of the chain (Push item) and deleting the end node of the list (Pop item).

Applications of Stack

- It is very useful to evaluate arithmetic expressions. (Postfix Expressions)
- Infix to Postfix Transformation
- It is useful during the execution of recursive programs
- A Stack is useful for designing the compiler in operating system to store local variables inside a function block.
- A stack (memory stack) can be used in function calls including recursion.

- Reversing Data and Reverse a List
- Convert Decimal to Binary
- Parsing – It is a logic that breaks into independent pieces for further processing
- Backtracking

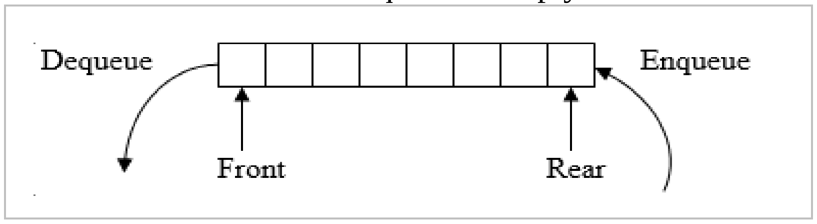
Examples:

1. Infix notation: A+(B*C)	Equivalent Postfix notation	ABC*+
2. Infix notation: (A+B)*C	Equivalent Postfix notation	AB+C*

3. Queue

"A queue is an ordered list in which all insertions at one end called **REAR** and deletions are made at another end called FRONT". Queues are sometimes referred to as First in First out (FIFO) lists. The two basic operations are:

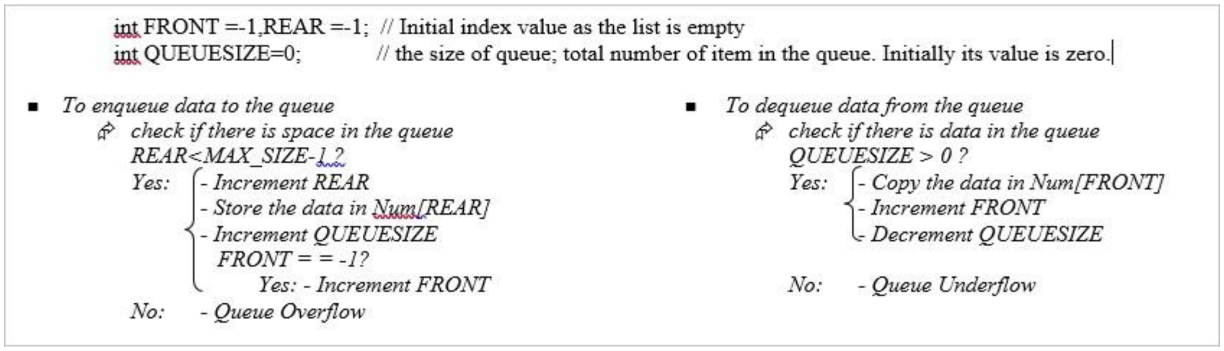
Enqueue – Inserts an item / element at the rear end of the queue. An overflow error occurs if the queue is full.
Dequeue – Removes an item / element from the front end of the queue, and returns it to the user. An underflow error occurs if the queue is empty.



Implementation of Queue Operations:

I. Simple Array implementation (static memory).

A simple array structure can be used to implement enqueue & dequeue operations. Thus item will be added at the end of the array whereas dequeue operation removes items from the front. In simple array implementation, to add item it checks whether there is room at the rear or not. Therefore, even though there are free released spaces at the front ot the array the queue implementation might generate overflow message. This is the key drawback to use simple array to implement Queue.



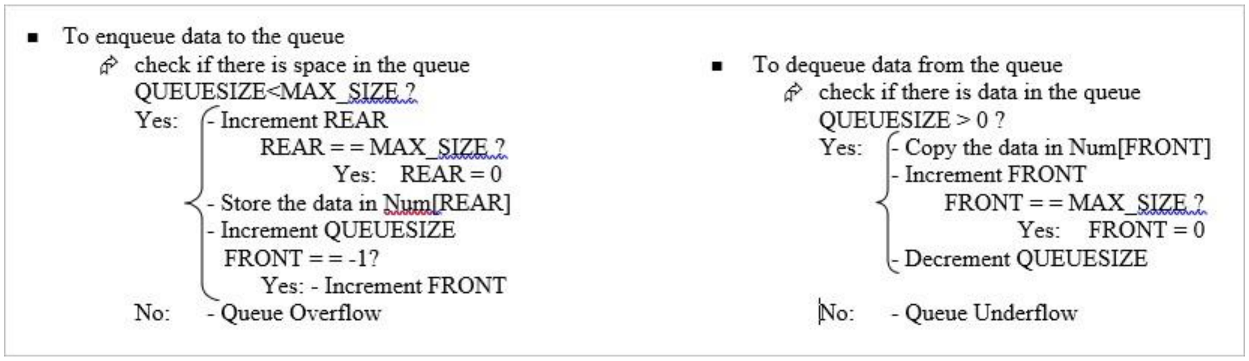
```
int FRONT =-1,REAR =-1;  // Initial index value as the list is empty
int QUEUESIZE=0;         // the size of queue; total number of item in the queue. Initially its value is zero.

▪ To enqueue data to the queue
    check if there is space in the queue
    REAR<MAX_SIZE-1 ?
    Yes: - Increment REAR
         - Store the data in Num[REAR]
         - Increment QUEUESIZE
           FRONT = = -1?
               Yes: - Increment FRONT
    No:  - Queue Overflow

▪ To dequeue data from the queue
    check if there is data in the queue
    QUEUESIZE > 0 ?
    Yes: - Copy the data in Num[FRONT]
         - Increment FRONT
         - Decrement QUEUESIZE
    No:  - Queue Underflow
```

II. Circular array implementation of Queue Operations

To overcome the above drawback of simple array implementation a circular array can be used, in which freed spaces are re-used to store data.



```
■ To enqueue data to the queue
    check if there is space in the queue
    QUEUESIZE<MAX_SIZE ?
    Yes:  - Increment REAR
                REAR = = MAX_SIZE ?
                    Yes:   REAR = 0
          - Store the data in Num[REAR]
          - Increment QUEUESIZE
            FRONT = = -1?
                Yes: - Increment FRONT
    No:   - Queue Overflow

■ To dequeue data from the queue
    check if there is data in the queue
    QUEUESIZE > 0 ?
    Yes:  - Copy the data in Num[FRONT]
          - Increment FRONT
                FRONT = = MAX_SIZE ?
                    Yes:   FRONT = 0
          - Decrement QUEUESIZE
    No:   - Queue Underflow
```

Example: Consider a queue with MAX_SIZE = 4; the following table demonstrate the difference between simple array and circular array implementation of Queue!

Operation	**Simple array**							**Circular array**						
	Content of the array				Content of the Queue	QUEUE SIZE	Message	Content of the array				Content of the queue	QUEUE SIZE	Message
Enqueue(B)	B				B	1		B				B	1	
Enqueue(C)	B	C			BC	2		B	C			BC	2	
Dequeue()		C			C	1			C			C	1	
Enqueue(G)		C	G		CG	2			C	G		CG	2	
Enqueue (F)		C	G	F	CGF	3			C	G	F	CGF	3	
Dequeue()			G	F	GF	2				G	F	GF	2	
Enqueue(A)			G	F	GF	2	Overflow	A		G	F	GFA	3	
Enqueue(D)			G	F	GF	2	Overflow	A	D	G	F	GFAD	4	
Enqueue(C)			G	F	GF	2	Overflow	A	D	G	F	GFAD	4	Overflow
Dequeue()				F	F	1		A	D		F	FAD	3	
Enqueue(H)				F	F	1	Overflow	A	D	H	F	FADH	4	
Dequeue ()					Empty	0		A	D	H		ADH	3	
Dequeue()					Empty	0	Underflow		D	H		DH	2	
Dequeue()					Empty	0	Underflow			H		H	1	
Dequeue()					Empty	0	Underflow					Empty	0	
Dequeue()					Empty	0	Underflow					Empty	0	Underflow

III. Linked list implementation of enqueue and dequeue operations

Enqueue- is inserting a node at the end of a linked list
Dequeue- is deleting the first node in the list

Deque (pronounced as Deck): is a Double Ended Queue in which insertion and deletion can occur at either end.
Priority Queue:- is a queue where each data has an associated key provided at insertion time.

Application of Queue

Queue is good for fair (first come first served) ordering of actions. Some of queue applications:

- Print server- maintains a queue of print jobs
- Disk Driver- maintains a queue of disk input/output requests
- Task scheduler in multiprocessing system- maintains priority queues of processes
- Telephone calls in a busy environment –maintains a queue of telephone calls
- Simulation of waiting line- models real life queues (e.g. supermarkets checkout, customer service line (Bank, custom, airport etc))

4. TREE STRUCTURES

A tree is a set of nodes and edges that connect pairs of nodes that connect pairs of nodes. It is an abstract model of a hierarchical structure. Rooted tree has the following structure:

- One node distinguished as root.
- Every node C except the root is connected from exactly other node P. P is C's parent, and C is one of C's children.
- There is a unique path from the root to the each node.
- The number of edges in a path is the length of the path.

Tree Terminologies:

- **Root**: a node without a parent.
- **Internal node:** a node with at least one child.
- **External (leaf) node:** a node without a child.
- **Ancestors of a node**: parent, grandparent, grand-grandparent, etc of a node.
- **Descendants of a node:** children, grandchildren, grand-grandchildren etc of a node.
- **Depth of a node:** number of ancestors or length of the path from the root to the node.
- ***Height of a tree:*** depth of the deepest node.
- ***Subtree:*** a tree consisting of a node and its descendants.
- ***Binary tree:*** a tree in which each node has at most two children called left child and right child.
- ***Binary search tree (ordered binary tree):*** a binary tree that may be empty, but if it is not empty it satisfies the following.
    - Every node has a key and no two elements have the same key.
    - The keys in the right subtree are larger than the keys in the root.
    - The keys in the left subtree are smaller than the keys in the root.
    - The left and the right subtrees are also binary search trees.

Operations on Binary Search Tree

I. Tree structure definition:

Example:

```
struct Node{
        int Num;   // Node value/Number
        Node * Left, *Right;     // Pointer to Left and right Children
        };
Node *RootNodePtr=NULL;     // pointer to root Node
```

II. Insertion

When a node is inserted the definition of binary search tree should be preserved.

Case 1: There is no data in the tree (i.e. RootNodePtr is NULL)

- The node pointed by InsNodePtr should be made the root node.

Case 2: There is data

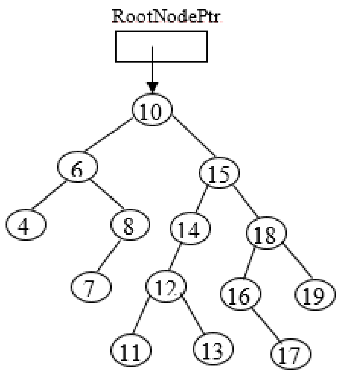
- Search the appropriate position.
- Insert the node in that position.

III. Tree Traversing:

Binary search tree can be traversed in three ways.

a. Pre order traversal - traversing binary tree in the order of **parent**, left and right.
b. Inorder traversal - traversing binary tree in the order of left, **parent** and right.
c. Postorder traversal - traversing binary tree in the order of *left*, *right* and ***parent***.

Example:



Preorder traversal - 10, 6, 4, 8, 7, 15, 14, 12, 11, 13, 18, 16, 17, 19
Inorder traversal - 4, 6, 7, 8, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19
==> Used to display nodes in ascending order.
Postorder traversal- 4, 7, 8, 6, 11, 13, 12, 14, 17, 16, 19, 18, 15, 10

IV. Searching

To search a node (whose Num value is Number) in a binary search tree (whose root node is pointed by RootNodePtr), one of the three traversal methods can be used.

V. Deletion

To delete a node (whose Num value is N) from binary search tree (whose root node is pointed by RootNodePtr), four cases should be considered. When a node is deleted the definition of binary search tree should be preserved.

Case 1: Deleting a leaf node (a node having no child): it is simple and straightforward operation. Delete the leaf node and make the pointer of the deleted node parent NULL (right or left child pointer based on whether the deleted child is left child or right child)

Case 2: Deleting a node having only one child
Case 3: Deleting a node having two children
Case 4: Deleting the root node
Note: For case 2, 3 and 4 there are two approaches
Approach 1: Deletion by copying

- Copy the node containing the largest element in the left (or the smallest element in the right) to the node containing the element to be deleted
- Delete the Copied Node

Approach 2: Deletion by merging

If deleted node is Root:

- the root node pointer is made to point to the right child or left child
- the left child/right child of the root node is made the left child/right child of the node containing the smallest element/largest in the right/left of the root node respectively.

If deleted node has only one child:

- Promote a child (left or right) of deleted node as a child (left or right) for the parent of the deleted node keeping the definition of BST.

If deleted node has two children:

- Promote a child (left or right) of deleted node as a child for the parent of the deleted node.
- The other child of the deleted node is made the left child of the node containing smallest element in the right of the deleted node or the right child of the deleted node is made the right child of the node containing largest element in the left of the deleted node

PART FOUR: Advanced Sorting and Searching Algorithms

i. Shell Sort: it is an improvement of insertion sort. It creates a reasonable order first. Its time complexity is $O(n^{3/2})$

Algorithm:

1. Choose gap g_k between elements to be partly ordered.
2. Generate a sequence (called increment sequence) g_k, g_{k-1},…., g_2, g_1 where for each sequence g_i, A[j]<=A[j+g_i] for 0<=j<=n-1-g_i and k>=i>=1

ii. Quick sort: is the fastest known algorithm. It uses divide and conquer strategy and in the worst case its complexity is O (n2). But its expected complexity is O(nlogn).

Algorithm:

1. Choose a pivot value (mostly the first element is taken as the pivot value)
2. Position the pivot element and partition the list so that:
   - the left part has items less than or equal to the pivot value
   - the right part has items greater than or equal to the pivot value
3. Recursively sort the left part
4. Recursively sort the right part

iii. Heap Sort

Heap sort operates by first converting the list in to a heap tree. Heap tree is a binary tree in which each node has a value greater than both its children (if any). It uses a process called "adjust to accomplish its task (building a heap tree) whenever a value is larger than its parent. The time complexity of heap sort is O(nlogn).

Algorithm:

1. Construct a binary tree
   - The root node corresponds to Data[0].
   - If we consider the index associated with a particular node to be *i*, then the left child of this node corresponds to the element with index 2*i+1 and the right child corresponds to the element with index 2*i+2. If any or both of these elements do not exist in the array, then the corresponding child node does not exist either.
2. Construct the heap tree from initial binary tree using "adjust" process.

Sort by swapping the root value with the lowest, right most value and deleting the lowest, right most value and inserting the deleted value in the array in it proper position.

iv. Merge sort

Merge sort uses divide and conquer strategy and its time complexity is O(nlogn).

Algorithm:

1. Divide the array in to two halves.
2. Recursively sort the first n/2 items.
3. Recursively sort the last n/2 items.
4. Merge sorted items (using an auxiliary array).