Lab Manual for

Data Structures and Algorithms Lab (CSE-2111)

Credit: 1.5, Contact hour: 3 Hours Per week

Department of
Computer Science and Engineering
Sheikh Hasina University
Netrokona, Bangladesh

INDEX

Lab Instructions:

- Students should come with thorough preparation for the experiment to be conducted.
- Students will not be permitted to attend the laboratory unless they bring the practical record fully completed in all respects pertaining to the experiment conducted in the previous class.
- Be honest in developing and representing your program. If a particular program output appears wrong repeat the program carefully.
- Strictly observe the instructions given by the Faculty / Lab. Instructor.
- Take permission before entering in the lab and keep your belongings in the racks.
- NO FOOD, DRINK, IN ANY FORM is allowed in the lab.
- TURN OFF CELL PHONES! If you need to use it, please keep it in bags.
- Avoid all horseplay in the laboratory. Do not misbehave in the computer laboratory. Work quietly.
- Save often and keep your files organized
- Do not reconfigure the cabling/equipment without prior permission.
- Do not play games on systems.
- If you finish early, spend the remaining time to complete the laboratory report writing. Come equipped with calculator and other materials related to lab works.
- Handle instruments with care. Report any breakage or faulty equipment to the Instructor. Shutdown your computer you have used for the purpose of your experiment before leaving the Laboratory.
- Violation of the above rules and etiquette guidelines will result in disciplinary action.

Sheikh Hasina University

COURSE SYLLABUS

1	Faculty	Faculty of Engineering and Technology
2	Department	Department of CSE
3	Program	B.Sc. in Computer Science and Engineering
4	Name of Course	Data Structures and Algorithm Lab
5	Course Code	CSE-2111
6	Year and Semester	2nd year 1st semester
7	Pre-requisites	CSE-1211
8	Status	Core Course
9	Credit Hours	1.5
10	Class Hours	3 hours
11	Class Location	Lab Room-406
12	Name (s) of Academic staff / Instructor(s)	**Abdullah Al Shiam** Lecturer, Dept. of CSE, Sheikh Hasina University
13	Contact	shiam@vu.edu.bd
14	Counseling Hours	
15	Text Book	Data structure schaum outline series by Seymour Lipschutz (*McGraw-Hill*)
16	Reference	*1.* Data Structures and Algorithms in Java, *Robert Lafore* *2.* The Art of Computer Programming, *Donald E. Knuth* 3. Data Structures and Algorithms Made Easy , NarasimhaKarumanchi 4. An Introduction to Data Structures and Algorithms, J A Storer and John C Cherniavsky
17	Equipment & Aids	• Lab Sheet • Text Book • Code: Blocks
18	Course Description	The purpose of this course is to provide the students with solid foundations in the basic concepts of programming: data structures and algorithms. The course is to teach the students how to select and design data structures and algorithms that are appropriate for problems that they might encounter. This course is also about showing the correctness of algorithms and studying their computational complexities. This course offers the students a mixture of theoretical knowledge and practical experience.
19	Course Objectives	This course will help students to achieve the following objectives: 1. Familiarize the student with good programming design methods, particularly Top-Down design. 2. Develop algorithms for manipulating stacks, queues, linked lists, trees and graphs. 3. Develop the data structures for implementing the above algorithms. 4. Develop recursive algorithms as they apply to trees and graphs. 5. Familiarize the student with the issues of Time complexity and examine various algorithms from this perspective.

21	Learning Outcomes	Upon completion of this course, students should be able to: I. identify fundamental data structures and algorithms and summarize their typical uses, strengths, and weaknesses II. analyze the complexity of algorithms III. solve problems computationally through the application of fundamental data structures and algorithms
22	Teaching Methods	Lecture, Video Demonstration, Problem Solving, Project Development
23	Topic Outline	

	Class	**Topics Or Assignments**		**Reading Reference**	**Activities**
	Lab-1	Introduction Of Array			Problem Solving, Question Answer
	Lab-2	Array Insertion & Deletion			Implementation, Problem Solving
	Lab-3	Linear Search Algorithm			Problem Solving, Question Answer
	Lab-4	Binary Search Algorithm			Problem Solving, Question Answer
	Lab-5	Bubble Sorting and Complexity Analysis			Problem Solving, Question Answer
	Lab-6	Linked List			Implementation, Problem Solving
	Lab-7	Stack			Implementation, Problem Solving
	Lab-8	Queue			Problem Solving, Question Answer
	Lab-9	Tree Traversal			Problem Solving, Question Answer
	Lab-10	Graph Traversal			Implementation, Problem Solving
	Lab-11	Insertion & Selection Sort			Problem Solving, Question Answer
	Lab-12	Lab Final Examination			Multiple Choice Question Answer, Viva

25 Assessment Methods

Assessment Types	**Marks**
Attendance	10%
Lab Report	20%
Continuous Assessment	50%
Lab Viva-voce	20%
Total	100%

25	Grading Policy	

Numerical Grade	Letter Grade	Grade Point
80% and above	A+	4.00
75% to less than 80%	A	3.75
70% to less than 75%	A•	3.50
65% to less than 70%	B+	3.25
60% to less than 65%	B	3.00
55% to less than 60%	B•	2.75
50% to less than 55%	C+	2.50
45% to less than 50%	C	2.25
40% to less than 45%	D	2.00
less than 40%	F	0.00

26	Additional Course Policies	1. 1. Lab Reports Report on previous Experiment must be submitted before the beginning of new experiment. A bonus may be obtained if a student submits a neat, clean and complete lab report. 2. 2. Examination There will be a lab exam at the end of the semester that will be closed book. 3. 3. Unfair means policy In case of copying/plagiarism in any of the assessments, the students involved will receive zero marks. Zero Tolerance will be shown in this regard. In case of severe offences, actions will be taken as per university rule. 4. 4. Counseling Students are expected to follow the counseling hours posted. In case of emergency/unavoidable situations, students can e-mail me to make an appointment. Students are regularly advised to check the piazza course page for updates/materials. 5. 5. Policy for Absence in Class/Exam If a student is absent in the class for anything other than medical reasons, he/she will not receive attendance. If a student misses a class for genuine medical reasons, he/she must apply with the supporting documents (prescription/medical report). He/she will then have to follow the instructions given by the instructor for makeup. In case of absence in the mid/final exam for medical grounds, the student must also get his/her application forwarded by the head of the department before a make-up exam can be taken. It is recommended that the students inform the instructor beforehand through mail if they feel that they will miss a class/evaluation due to medical reasons.

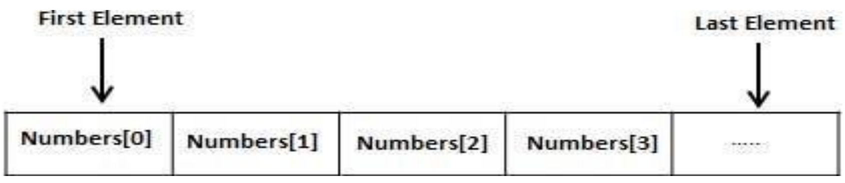
EXPERIMENT NO – 1

Aim: Study of Array declaration and traversing

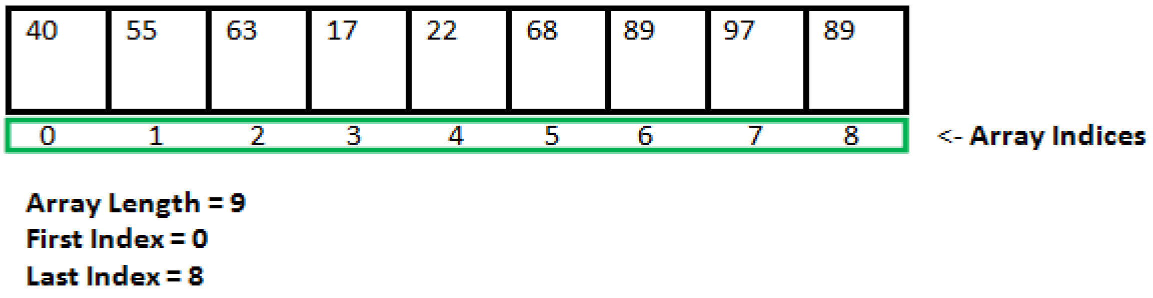
ARRAYS:

Arrays a kind of data structure that can store a fixed-size sequential collection of elements of the same type. An array is used to store a collection of data, but it is often more useful to think of an array as a collection of variables of the same type.

All arrays consist of contiguous memory locations. The lowest address corresponds to the first element and the highest address to the last element.



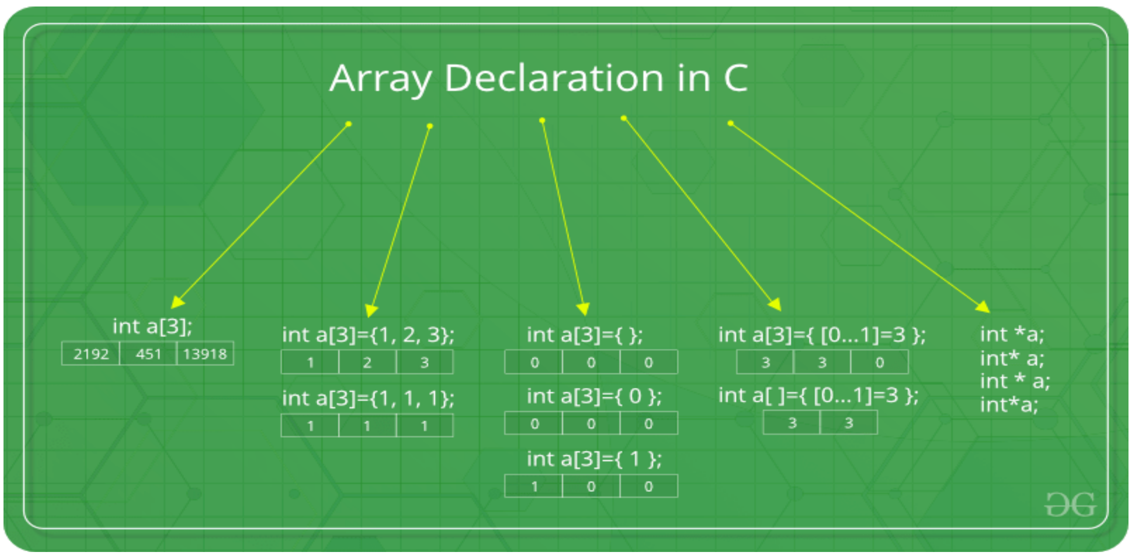
An array is collection of items stored at continuous memory locations.



Why do we need arrays?

We can use normal variables (v1, v2, v3,...) when we have small number of objects, but if we want to store large number of instances, it becomes difficult to manage them with normal variables. The idea of array is to represent many instances in one variable.

Array declaration in C:



We can declare an array by specifying its type and size or by initializing it or by both.

1. **Array declaration by specifying size**

```
// Array declaration by specifying size
intarr1[10];

// With recent C/C++ versions, we can also
// declare an array of user specified size
intn = 10;
intarr2[n];
```

2. **Array declaration by initializing elements**

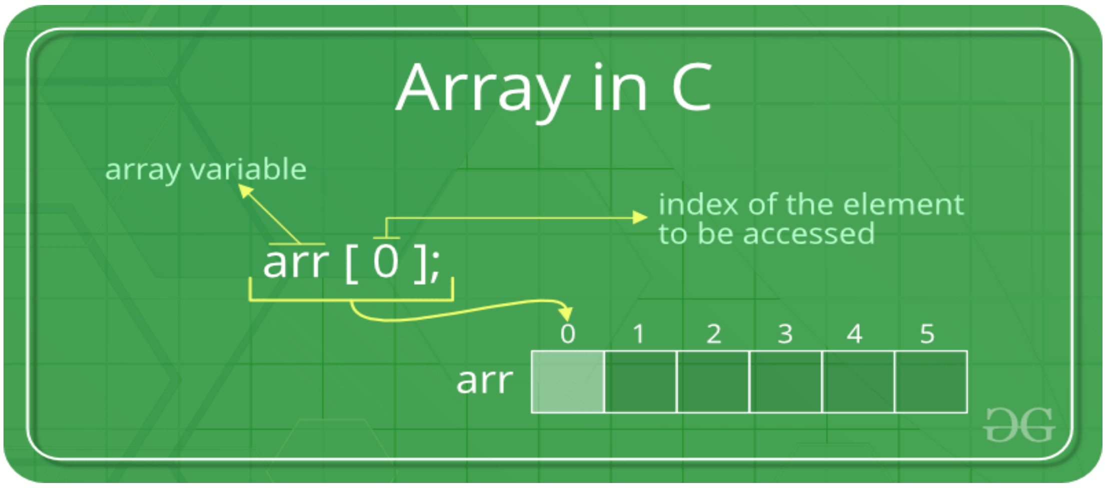
```
// Array declaration by initializing elements
intarr[] = { 10, 20, 30, 40 }

// Compiler creates an array of size 4.
// above is same as "intarr[4] = {10, 20, 30, 40}"
```

3. **Array declaration by specifying size and initializing elements**

```
// Array declaration by specifying size and initializing
// elements
intarr[6] = { 10, 20, 30, 40 }

// Compiler creates an array of size 6, initializes first
// 4 elements as specified by user and rest two elements as 0.
// above is same as "intarr[] = {10, 20, 30, 40, 0, 0}"
```

Facts about Array in C/C++:

- **Accessing Array Elements:**
  Array elements are accessed by using an integer index. Array index starts with 0 and goes till size of array minus 1.



Following are few examples.

```
#include <stdio.h>

intmain()
```

```
{
    intarr[5];
    arr[0] = 5;
    arr[2] = -10;
    arr[3 / 2] = 2; // this is same as arr[1] = 2
    arr[3] = arr[0];

    printf("%d %d %d %d", arr[0], arr[1], arr[2], arr[3]);

    return0;
}
```

Output:

```
5    2     -10      5
```

<u>Remember:</u> array index in C/ C++ always start from 0.
<u>Another Example:</u>

Example 01: Program to find the average of n (n < 10) numbers using arrays

```
#include <stdio.h>
int main()
{
int marks[10], i, n, sum = 0, average;
printf("Enter n: ");
scanf("%d", &n);
for(i=0; i<n; ++i)
    {
printf("Enter number%d: ",i+1);
scanf("%d", &marks[i]);
sum += marks[i];
    }
average = sum/n;

printf("Average = %d", average);

return 0;
}
```

Output

```
Enter n: 5
Enter number1: 45
Enter number2: 35
Enter number3: 38
Enter number4: 31
Enter number5: 49
Average = 39
```

Task 01: Write a C++ program for Character Matching, Maximum number finding and Minimum number finding from an array of data.

Algorithm:

Character Search Operation:

You can perform a search for an array element based on its value or its index.

Algorithm:

Consider **LA** is a linear array with **N** elements and **K** is a positive integer such that **K<=N**. Following is the algorithm to find an element with a value of ITEM using sequential search.

```
1. Start
2. Set J = 0
3. Repeat steps 4 and 5 while J < N
4. IF LA[J] is equal ITEM THEN GOTO STEP 6
5. Set J = J +1
6. PRINT J, ITEM
7. Stop
```

Maximum and Minimum number finding Operation:

You can perform a search for an array element based on its value or its index.

Algorithm:

```
Max-Min-Element (numbers[])
max := numbers[1]
min := numbers[1]

fori = 2 to n do
if numbers[i] > max then
max := numbers[i]
if numbers[i] < min then
min := numbers[i]
return (max, min)
```

LAB ASSIGNMENT:

1. Write a Program to Find the Largest Two Numbers in a given Array
2. Write a program to count the number of even and odd numbers in a given array.
3. Write a Program to Put Even & Odd Elements of an Array in 2 Separate Arrays

EXPERIMENT NO – 02

Aim: Array Insertion, Deletion operations

Insertion Operation:

Insert operation is to insert one or more data elements into an array. Based on the requirement, a new element can be added at the beginning, end, or any given index of array.

Here, we see a practical implementation of insertion operation, where we add data at the end of the array −

ALGORITHM:

Let **Array** be a linear unordered array of **MAX** elements.

EXAMPLE

Result

Let **LA** be a Linear Array (unordered) with **N** elements and **K** is a positive integer such that **K<=N**. Following is the algorithm where ITEM is inserted into the Kth position of LA −

```
1. Start
2. Set J = N
3. Set N = N+1
4. Repeat steps 5 and 6 while J >= K
5. Set LA[J+1] = LA[J]
6. Set J = J-1
7. Set LA[K] = ITEM
8. Stop
```

EXAMPLE:

Following is the implementation of the above algorithm −

```
#include<stdio.h>
main(){
int LA[]={1,3,5,7,8};
int item =10, k =3, n =5;
inti=0, j = n;

printf("The original array elements are :\n");

for(i=0;i<n;i++){
printf("LA[%d] = %d \n",i, LA[i]);
}

  n = n +1;

while( j >= k){
```

```
LA[j+1]= LA[j];
      j = j -1;
}

   LA[k]= item;

printf("The array elements after insertion :\n");

for(i=0;i<n;i++){
printf("LA[%d] = %d \n",i, LA[i]);
}
}
```

When we compile and execute the above program, it produces the following result −

OUTPUT

```
The original array elements are:
            LA[0] = 1
            LA[1] = 3
            LA[2] = 5
            LA[3] = 7
            LA[4] = 8

The array elements after insertion:
            LA[0] = 1
            LA[1] = 3
            LA[2] = 5
            LA[3] = 10
            LA[4] = 7
            LA[5] = 8
```

Deletion Operation:

Deletion refers to removing an existing element from the array and re-organizing all elements of an array.

ALGORITHM:

Consider **LA** is a linear array with **N** elements and **K** is a positive integer such that **K<=N**. Following is the algorithm to delete an element available at the Kth position of LA.

```
1. Start
2. Set J = K
3. Repeat steps 4 and 5 while J < N
4. Set LA[J] = LA[J + 1]
5. Set J = J+1
6. Set N = N-1
7. Stop
```

Example:

Following is the implementation of the above algorithm −

```
#include<stdio.h>
```

```
void main(){
int LA[]={1,3,5,7,8};
int k =3, n =5;
inti, j;

printf("The original array elements are :\n");

for(i=0;i<n;i++){
printf("LA[%d] = %d \n",i, LA[i]);
}

   j = k;

while( j < n){
LA[j-1]= LA[j];
      j = j +1;
}

   n = n -1;

printf("The array elements after deletion :\n");

for(i=0;i<n;i++){
printf("LA[%d] = %d \n",i, LA[i]);
}
}
```

When we compile and execute the above program, it produces the following result −
Output

```
The original array elements are:
            LA[0] = 1
            LA[1] = 3
            LA[2] = 5
            LA[3] = 7
            LA[4] = 8

The array elements after deletion:
            LA[0] = 1
            LA[1] = 3
            LA[2] = 7
            LA[3] = 8
```

LAB ASSIGNMENT

1. Write a program to read an array and insert an element in first and last position.
2. WAP to read a sorted array. Now search an ITEM & if the ITEM is not found then insert the ITEM into the sorted array.

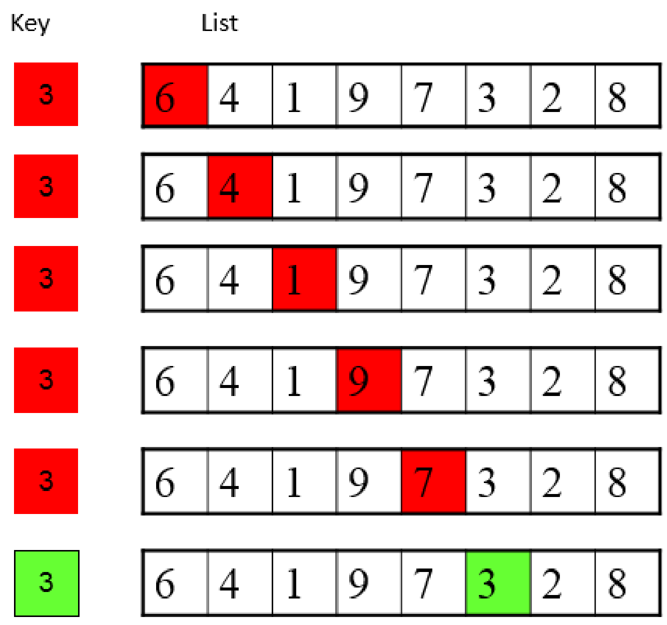
EXPERIMENT NO – 03

Aim: Implementing searching algorithms

PROGRAM LOGIC:

Linear Search:

Linear search is a very simple search algorithm. In this type of search, a sequential search is made over all items one by one. Every item is checked and if a match is found then that particular item is returned, otherwise the search continues till the end of the data collection.



Algorithm:

Linear Search (Array A, Value x)

```
Step 1: Set i to 1
Step 2: if i> n then go to step 7
Step 3: if A[i] = x then go to step 6
Step 4: Set i to i + 1
Step 5: Go to Step 2
Step 6: Print Element x Found at index i and go to step 8
Step 7: Print element not found
Step 8: Exit
```

Pseudocode:

```
procedurelinear_search(list, value)

for each item in the list
if match item == value
```

```
return the item's location
end if
end for

end procedure
```

LAB ASSIGNMENT

1. WAP to search an element in an array using Linear searching Algorithm. If found then print the element with its position and if the element is not found then print search is unsuccessful.
2. Write a program to print all unique elements in the array.
3. Write a C program to print all negative elements in an array
4. Write a program to count the number of duplicate elements
5. Write a program to count the number of unique elements.

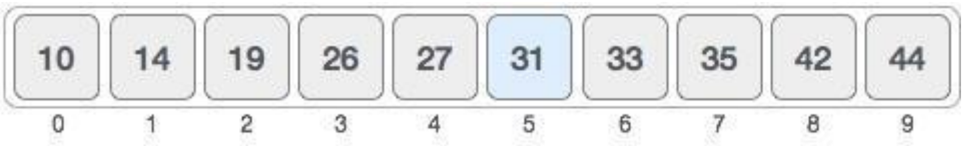
EXPERIMENT NO – 04

Binary search is a fast search algorithm with run-time complexity of O(log n). This search algorithm works on the principle of divide and conquer. For this algorithm to work properly, the data collection should be in the sorted form.

Binary search looks for a particular item by comparing the middle most item of the collection. If a match occurs, then the index of item is returned. If the middle item is greater than the item, then the item is searched in the sub-array to the left of the middle item. Otherwise, the item is searched for in the sub-array to the right of the middle item. This process continues on the sub-array as well until the size of the subarray reduces to zero.

How Binary Search Works?

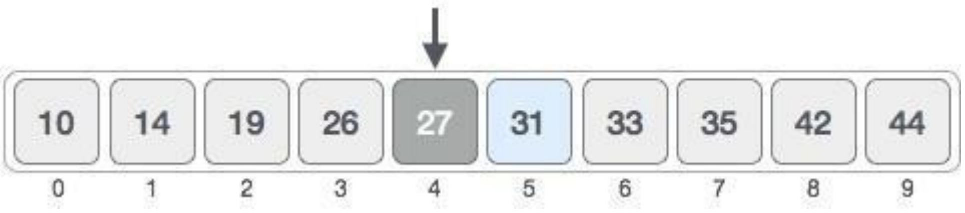
For a binary search to work, it is mandatory for the target array to be sorted. We shall learn the process of binary search with a pictorial example. The following is our sorted array and let us assume that we need to search the location of value 31 using binary search.



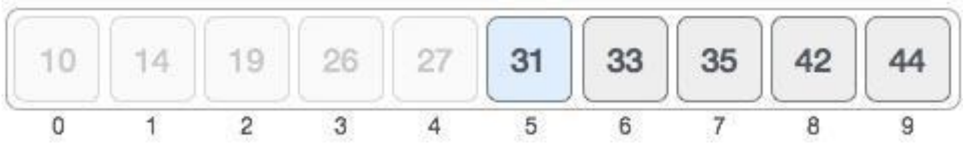
First, we shall determine half of the array by using this formula −

mid = low + (high - low) / 2

Here it is, 0 + (9 - 0) / 2 = 4 (integer value of 4.5). So, 4 is the mid of the array.



Now we compare the value stored at location 4, with the value being searched, i.e. 31. We find that the value at location 4 is 27, which is not a match. As the value is greater than 27 and we have a sorted array, so we also know that the target value must be in the upper portion of the array.

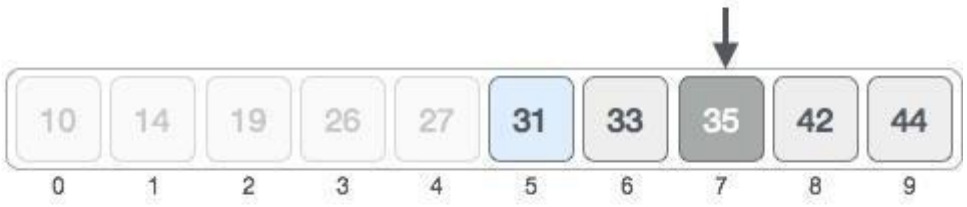


We change our low to mid + 1 and find the new mid value again.

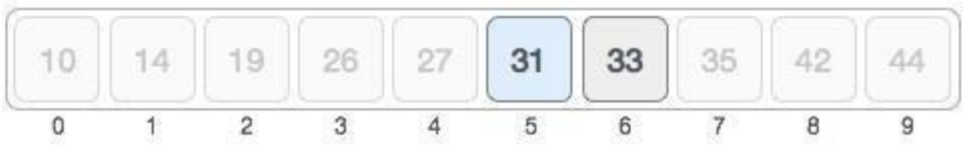
low = mid + 1

mid = low + (high - low) / 2

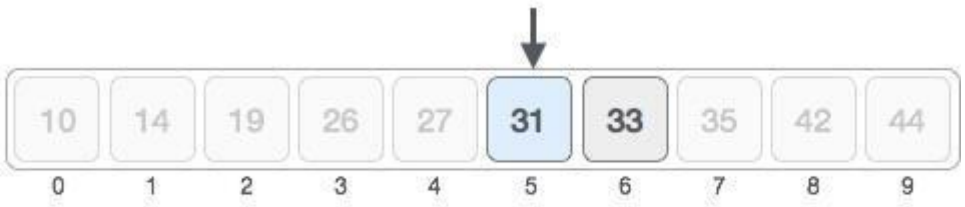
Our new mid is 7 now. We compare the value stored at location 7 with our target value 31.



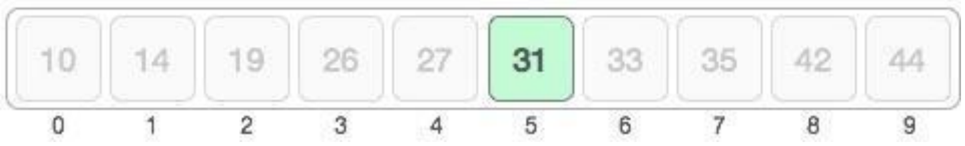
The value stored at location 7 is not a match, rather it is more than what we are looking for. So, the value must be in the lower part from this location.



Hence, we calculate the mid again. This time it is 5.



We compare the value stored at location 5 with our target value. We find that it is a match.



We conclude that the target value 31 is stored at location 5.

Binary search halves the searchable items and thus reduces the count of comparisons to be made to very less numbers.

Pseudocode

The pseudocode of binary search algorithms should look like this −

```
Procedurebinary_search
   A ← sorted array
n← size of array
x← value to be searched

SetlowerBound=1
SetupperBound= n

while x not found
ifupperBound<lowerBound
         EXIT: x does not exists.

setmidPoint=lowerBound+(upperBound-lowerBound)/2
```

```
if A[midPoint]< x
setlowerBound=midPoint+1

if A[midPoint]> x
setupperBound=midPoint-1

if A[midPoint]= x
        EXIT: x found at location midPoint
endwhile

end procedure
```

LAB ASSIGNMENT

- Write a program to read a sorted array. Now Using Binary Search Algorithm Find an ITEM In that array.
- Write a program to sort data elements (ascending order) in an array. Now search an ITEM using binary search algorithm. If item found, then show a message 'Search is successful.' If not then print "Item isn't found".
- Write a program to sort data elements (descending order) in an array. Now search an ITEM using binary search algorithm. If item found, then show a message 'Search is successful.' If not then print "Item isn't found".

EXPERIMENT NO – 05

Aim: Implementing Sorting Algorithm

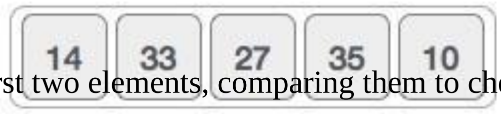
1. Sorting the list of integers in ascending order using Bubble sort.
2. Sorting the list of integers in ascending order using Quick sort

<u>Bubble sort:</u>

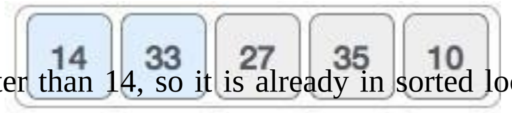
Bubble sort is a simple sorting algorithm. This sorting algorithm is comparison-based algorithm in which each pair of adjacent elements is compared and the elements are swapped if they are not in order. This algorithm is not suitable for large data sets as its average and worst case complexity are of O (n^2) where **n** is the number of items.

How Bubble Sort Works?

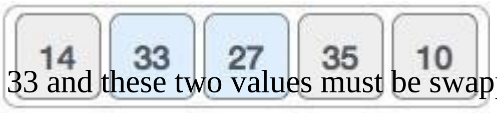
We take an unsorted array for our example. Bubble sort takes O(n^2) time so we're keeping it short and precise.



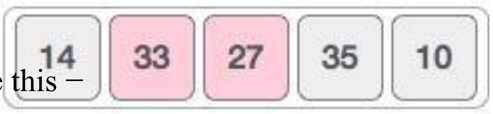
Bubble sort starts with very first two elements, comparing them to check which one is greater.



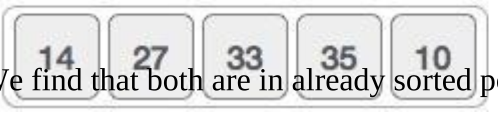
In this case, value 33 is greater than 14, so it is already in sorted locations. Next, we compare 33 with 27.



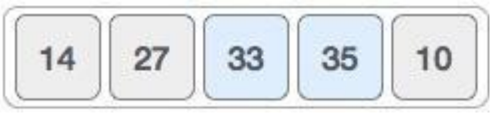
We find that 27 is smaller than 33 and these two values must be swapped.



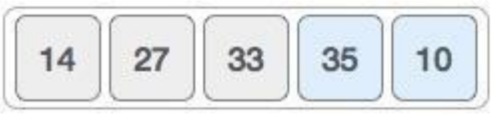
The new array should look like this –



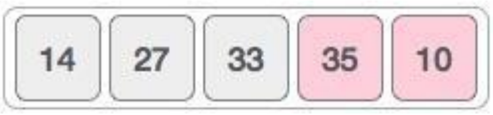
Next we compare 33 and 35. We find that both are in already sorted positions.



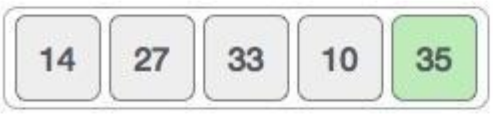
Then we move to the next two values, 35 and 10.



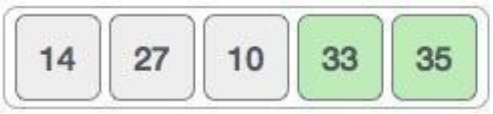
We know then that 10 is smaller 35. Hence they are not sorted.



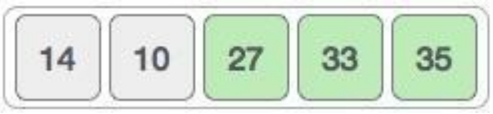
We swap these values. We find that we have reached the end of the array. After one iteration, the array should look like this −



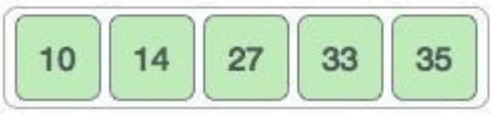
To be precise, we are now showing how an array should look like after each iteration. After the second iteration, it should look like this −



Notice that after each iteration, at least one value moves at the end.



And when there's no swap required, bubble sorts learns that an array is completely sorted.



Now we should look into some practical aspects of bubble sort.

Algorithm

We assume **list** is an array of **n** elements. We further assume that **swap** function swaps the values of the given array elements.

```
beginBubbleSort(list)

   for all elements of list
      if list[i]> list[i+1]
         swap(list[i], list[i+1])
      endif
   endfor

   return list

endBubbleSort
```

Pseudocode

We observe in algorithm that Bubble Sort compares each pair of array element unless the whole array is completely sorted in an ascending order. This may cause a few complexity issues like what if the array needs no more swapping as all the elements are already ascending.

To ease-out the issue, we use one flag variable **swapped** which will help us see if any swap has happened or not. If no swap has occurred, i.e. the array requires no more processing to be sorted, it will come out of the loop.

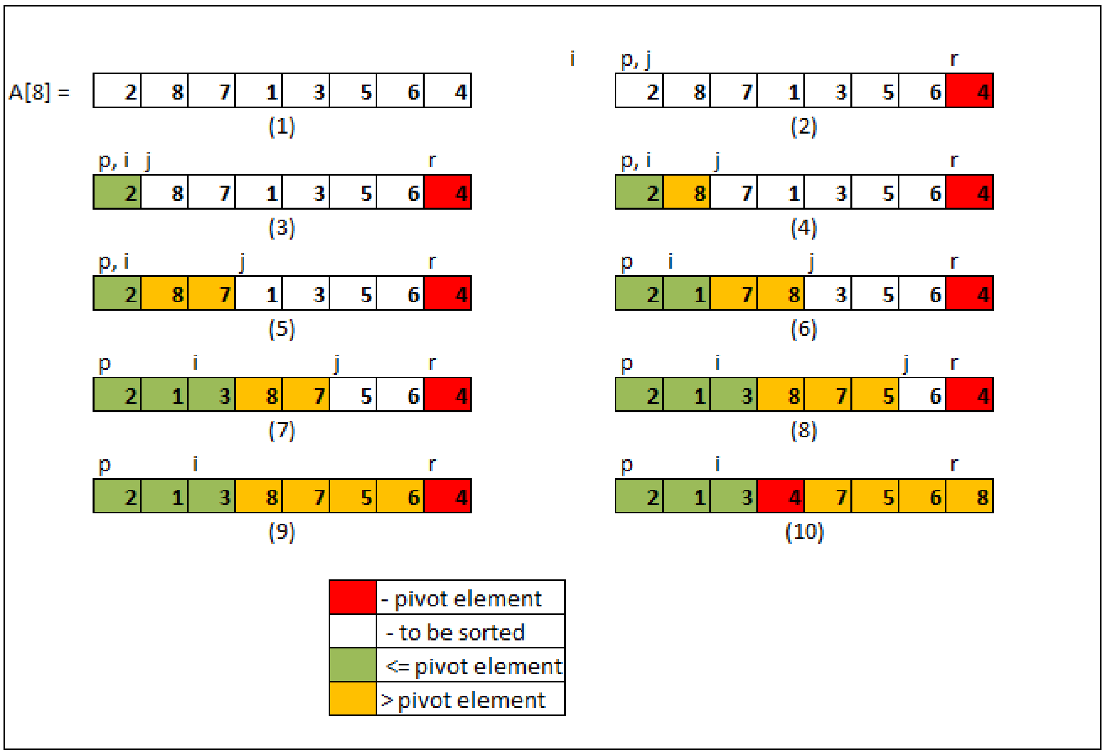
Pseudocode of BubbleSort algorithm can be written as follows −

```
procedurebubbleSort( list : array of items )

loop=list.count;

fori=0 to loop-1do:
swapped=false

for j =0 to loop-1do:

/* compare the adjacent elements */
if list[j]> list[j+1]then
/* swap them */
swap( list[j], list[j+1])
swapped=true
endif

endfor

/*if no number was swapped that means
array is sorted now, break the loop.*/

if(not swapped)then
break
endif

endfor

end procedure return list
```

Quick Sort:
Quick sort is a highly efficient sorting algorithm and is based on partitioning of array of data into smaller arrays. A large array is partitioned into two arrays one of which holds values smaller than the specified value, say pivot, based on which the partition is made and another array holds values greater than the pivot value.
Quick sort partitions an array and then calls itself recursively twice to sort the two resulting subarrays. This algorithm is quite efficient for large-sized data sets as its average and worst case complexity are of $O(n^2)$, where n is the number of items.

Partition in Quick Sort

Following animated representation explains how to find the pivot value in an array.



The pivot value divides the list into two parts. And recursively, we find the pivot for each sub-lists until all lists contains only one element.

Quick Sort Pivot Algorithm

Based on our understanding of partitioning in quick sort, we will now try to write an algorithm for it, which is as follows.

Step 1 − Choose the highest index value has pivot
Step 2 − Take two variables to point left and right of the list excluding pivot
Step 3 − left points to the low index
Step 4 − right points to the high
Step 5 − while value at left is less than pivot move right
Step 6 − while value at right is greater than pivot move left
Step 7 − if both step 5 and step 6 does not match swap left and right
Step 8 − if left ≥ right, the point where they met is new pivot

Quick Sort Pivot Pseudocode

The pseudocode for the above algorithm can be derived as −

```
functionpartitionFunc(left, right, pivot)
leftPointer= left
rightPointer= right -1

whileTruedo
while A[++leftPointer]< pivot do
//do-nothing
endwhile

whilerightPointer>0&& A[--rightPointer]> pivot do
```

```
//do-nothing
endwhile

ifleftPointer>=rightPointer
break
else
swapleftPointer,rightPointer
endif

endwhile

swapleftPointer,right
returnleftPointer

endfunction
```

Quick Sort Algorithm

Using pivot algorithm recursively, we end up with smaller possible partitions. Each partition is then processed for quick sort. We define recursive algorithm for quicksort as follows −

Step 1 − Make the right-most index value pivot
Step 2 − partition the array using pivot value
Step 3 − quicksort left partition recursively
Step 4 − quicksort right partition recursively

Quick Sort Pseudocode

To get more into it, let see the pseudocode for quick sort algorithm −

```
procedurequickSort(left, right)

if right-left <=0
return
else
pivot= A[right]
partition=partitionFunc(left, right, pivot)
quickSort(left,partition-1)
quickSort(partition+1,right)
endif

end procedure
```

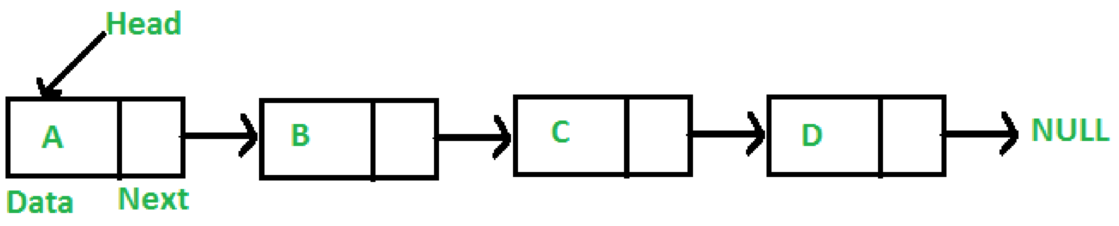
LAB ASSIGNMENT

1. Write a program that read N numbers in array and sort the number in ascending order using Quick Sort Algorithm.
2. Write a program that read N numbers in array and sort the number in ascending order using Bubble Sort Algorithm.
3. Write a program to sort array elements in descending order.

EXPERIMENT NO – 06

Aim: Implementation of Linked List.

Linked list:
A linked list is a linear data structure, in which the elements are not stored at contiguous memory locations. The elements in a linked list are linked using pointers as shown in the below image:



In simple words, a linked list consists of nodes where each node contains a data field and a reference (link) to the next node in the list.

Why Linked List?
Arrays can be used to store linear data of similar types, but arrays have following limitations.

- The size of the arrays is fixed: So we must know the upper limit on the number of elements in advance. Also, generally, the allocated memory is equal to the upper limit irrespective of the usage.
- Inserting a new element in an array of elements is expensive; because room has to be created for the new elements and to create room existing elements have to be shifted. For example, in a system if we maintain a sorted list of IDs in an array id[].

  Id[] = [1000, 1010, 1050, 2000, 2040].

And if we want to insert a new ID 1005, then to maintain the sorted order, we have to move all the elements after 1000 (excluding 1000).
Deletion is also expensive with arrays until unless some special techniques are used. For example, to delete 1010 in id[], everything after 1010 has to be moved.

Representation:
A linked list is represented by a pointer to the first node of the linked list. The first node is called head. If the linked list is empty, then value of head is NULL.Each node in a list consists of at least two parts:

- Data
- Pointer (Or Reference) to the next node

```
structNode {
    intdata;
    structNode* next;
};
```

CREATE LINKED LIST:

STEPS:

1. Declare a pointer to node type variable to store link of first node of linked list. Say struct node *head;.

   Note: You can also declare variable of node type along with node structure definition.

2. Input number of nodes to create from user, store it in some variable say n.
3. Declare two more helper variable of node type, say struct node *newNode, *temp;.
4. If n > 0 then, create our first node i.e. head node. Use dynamic memory allocation to allocate memory for a node. Say head = (struct node*)malloc(sizeof(struct node)); .
5. If there is no memory to allocate for head node i.e. head == NULL. Then print some error message and terminate program, otherwise move to below step.
6. Input data from user and assign to head using head->data = data; .
7. At first head node points to NULL. Hence, assign head->next = NULL;.
8. Now, we are done with head node we should move to creation of other nodes. Copy reference of head to some other temporary variable, say temp = head;. We will use temp to store reference of previous node.
9. Allocate memory and assign memory reference to newNode, say newNode = (struct node*)malloc(sizeof(node));.
10. If memory got allocated successfully then read data from user and assign to data section of new node. Say newNode->data = data;.
11. Make sure new node points to NULL.
12. Now link previous node with newly created node i.e. temp->next = newNode;.
13. Make current node as previous node using temp = temp->next; .
14. Repeat step 10-14 for remaining n - 2 other nodes.

TRAVERSING A LINKED LIST: In this algorithm a linked list, pointed by first, is traversed. The number of nodes in the list is also counted during the traverse. A pointer ptr is being used to visit the various nodes in the list. The traverse stops when a NULL is encountered.

STEPS:

1. Create a temporary variable for traversing. Assign reference of head node to it, say temp = head.
2. Repeat below step till temp != NULL.
3. temp->data contains the current node data. You can print it or can perform some calculation on it.
4. Once done, move to next node using temp = temp->next;.
5. Go back to 2nd step.

LAB ASSIGNMENT

1. Write a Program to create a link list and display the elements of link list.
2. Write a program to create a link list and Count the number of node on that list.
3. Write a program to create a link list and add 10 to every node info.

EXPERIMENT NO – 06

Aim: Searching, Inserting and Deleting in singly linked list

SEARCHING A LINKED LIST:

Search is an operation in which an item is searched in a linked list. This operation is similar to traveling the list. An algorithm for search operation is given below:
STEPS:

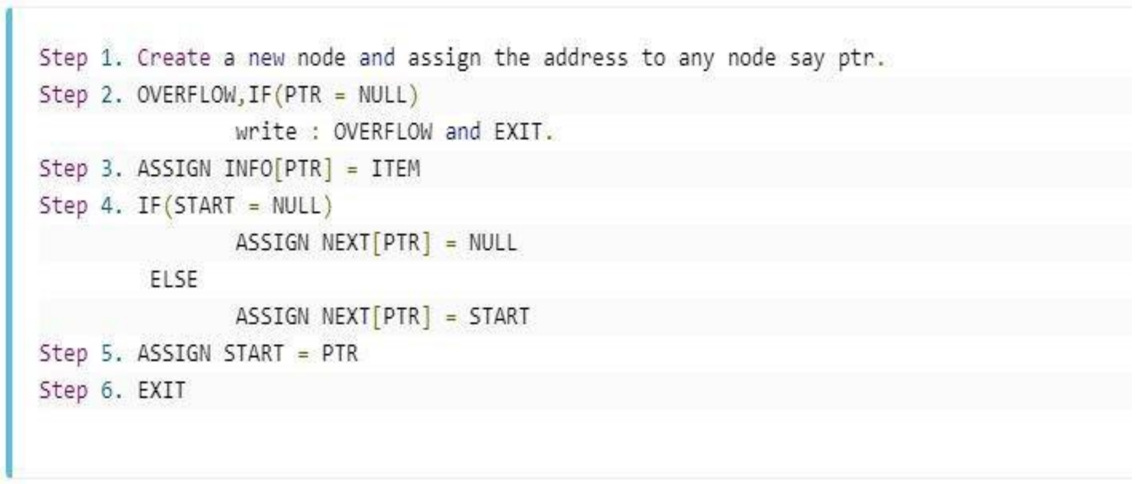
```
1.If first=NULL then{
        Print "List empty"; STOP;}
2.ptr=First;     [point ptr to the 1st node]
3.while (ptr<>NULL) repeat steps 4 to 5
4.If (DATA (ptr)= 'X')
        Then {print "item found";
        STOP
        }
5.ptr=NEXT (ptr); [shift ptr to the next node]
                [end of while]
6.Print "item not found";
7.END
```

Task-01: Write a program to create a Single unsorted link list and search a specific input item.

Task-02: Write a Program to create a Sorted link list and search a specific input item.

INSERTION:

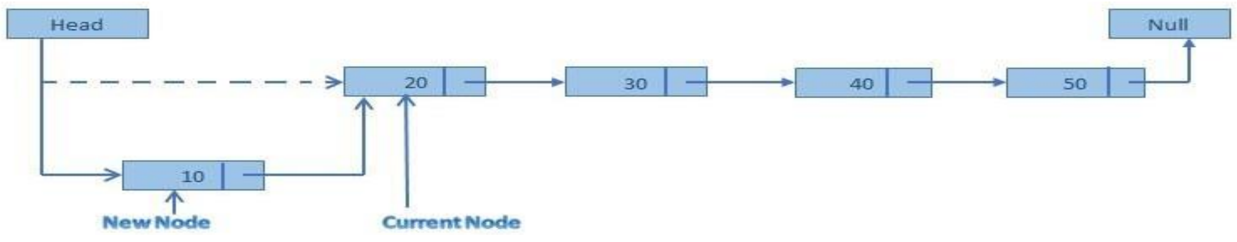
Insertion at the beginning of the singly linked lists



```
Step 1. Create a new node and assign the address to any node say ptr.
Step 2. OVERFLOW,IF(PTR = NULL)
                write : OVERFLOW and EXIT.
Step 3. ASSIGN INFO[PTR] = ITEM
Step 4. IF(START = NULL)
                ASSIGN NEXT[PTR] = NULL
         ELSE
                ASSIGN NEXT[PTR] = START
Step 5. ASSIGN START = PTR
Step 6. EXIT
```

Example:

```
void insertion(struct node *nw){
struct node start,*previous,*new1;
nw=start.next;
previous=&start;
new1 =(struct node*)malloc(sizeof(struct node));
 new1->next=nw;
previous->next= new1;
printf("\n Input the fisrt node value: ");
scanf("%d",&new1->data);
}
```



Task-01: Write a program to create a link list and insert an item at the beginning of the list.

Task-02: Write a program to create a link list and insert an item after a specific Location.

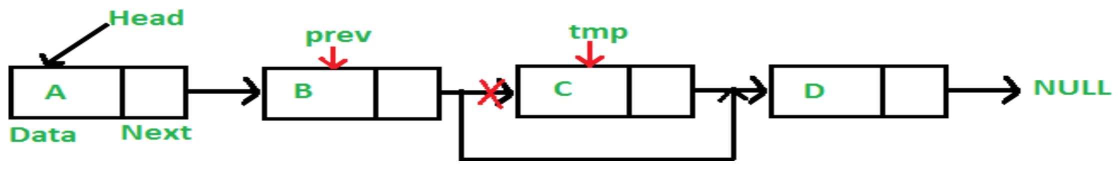
DELETING A NODE FROM A LINKED LIST:

Algorithm for Deletion at the beginning in a linked list:

To delete a node from linked list, we need to do following steps.

1) Find previous node of the node to be deleted.

2) Change the next of previous node.
3) Free memory for the node to be deleted.



Example:

```
//delete first item
struct node*deleteFirst(){
//save reference to first link
struct node *tempLink= head;

//mark next to first link as first
head= head->next;

//return the deleted link
returntempLink;
}
```

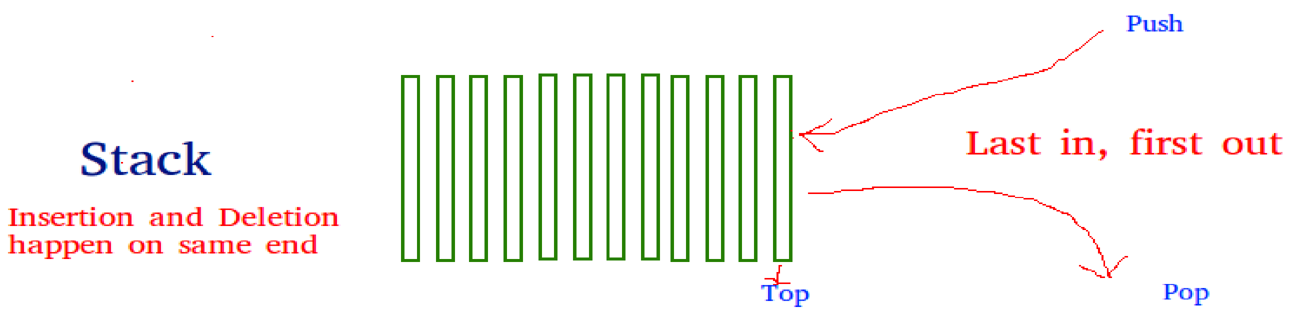
LAB ASSIGNMENT

Task 1: Create and display linked list.
Task 2: Insert a node at a specific position in a linked list.
Task 3: Delete a node from given information of that node.

EXPERIMENT NO – 07

Aim: To push, pop and display the Stack elements**.**

How Stack works:
Stack is a linear data structure which follows a particular order in which the operations are performed. The order may be LIFO (Last In First Out) or FILO(First In Last Out).



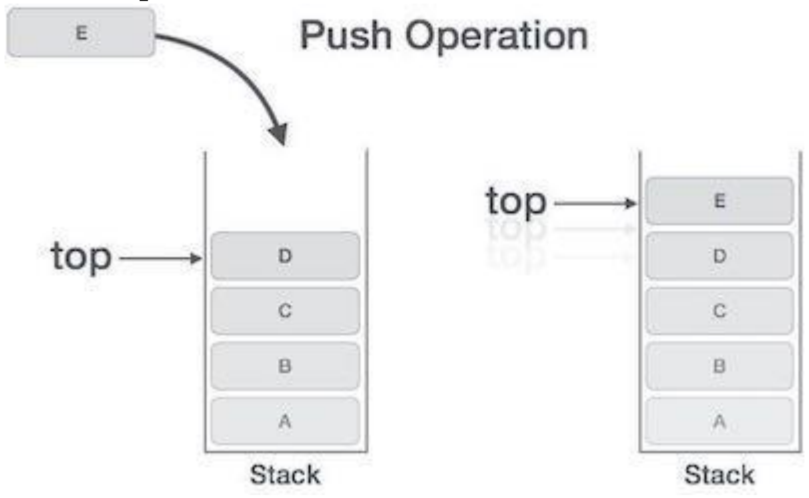
There are many real-life examples of a stack. Consider an example of plates stacked over one another in the canteen. The plate which is at the top is the first one to be removed, i.e. the plate which has been placed at the bottommost position remains in the stack for the longest period of time. So, it can be simply seen to follow LIFO(Last In First Out)/FILO(First In Last Out) order.

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.

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.

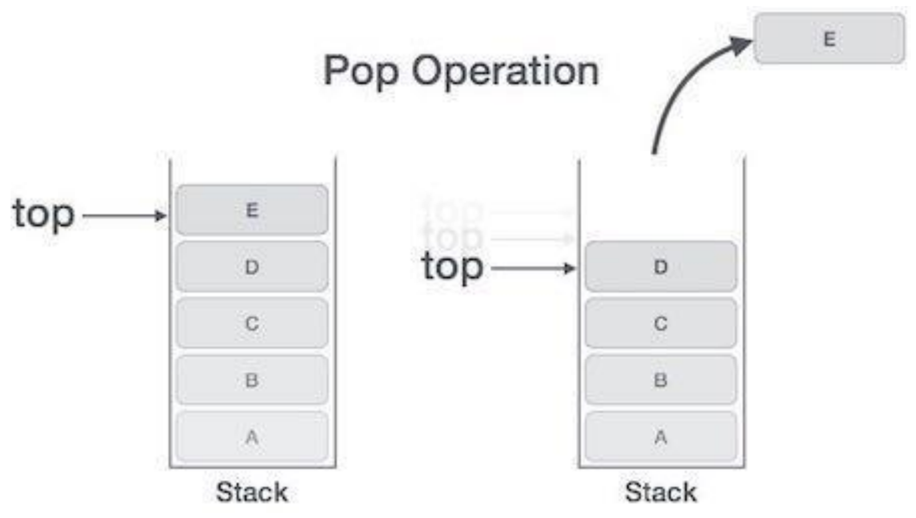


If the linked list is used to implement the stack, then in step 3, we need to allocate space dynamically.

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.



<u>Algorithm for PUSH operation in a Stack:</u>

A simple algorithm for Push operation can be derived as follows −

```
begin procedure push: stack, data

if stack is full
returnnull
endif

top← top +1
stack[top]← data

end procedure
```

Implementation of this algorithm in C, is very easy. See the following code −

<u>Example:</u>

```
void push(int data){
if(!isFull()){
top= top +1;
stack[top]= data;
}else{
printf("Could not insert data, Stack is full.\n");
}
}
```

<u>Algorithm for POP operation in a Stack:</u>

A simple algorithm for Pop operation can be derived as follows –

```
begin procedure pop: stack

if stack is empty
returnnull
endif

data← stack[top]
top← top -1
return data

end procedure
```

Implementation of this algorithm in C, is as follows −

Example

```
int pop(int data){

if(!isempty()){
data= stack[top];
top= top -1;
return data;
}else{
printf("Could not retrieve data, Stack is empty.\n");
}
}
```

Task1: Push some data into stack.

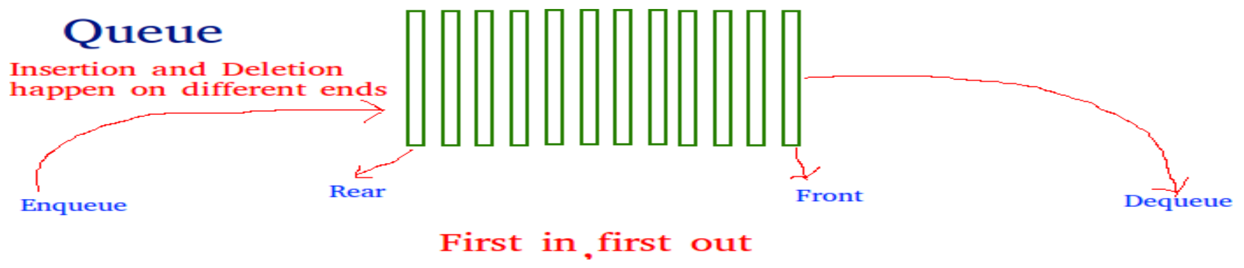
Task2: Display the data in reverse order.

EXPERIMENT NO – 08

Aim: Insertion and deletion operation in QUEUE data Structure.

How QUEUE works:

A Queue is a linear structure which follows a particular order in which the operations are performed. The order is First In First Out (FIFO). A good example of a queue is any queue of consumers for a resource where the consumer that came first is served first. The difference between stacks and queues is in removing. In a stack we remove the item the most recently added; in a queue, we remove the item the least recently added.



Applications of Queue Data Structure:

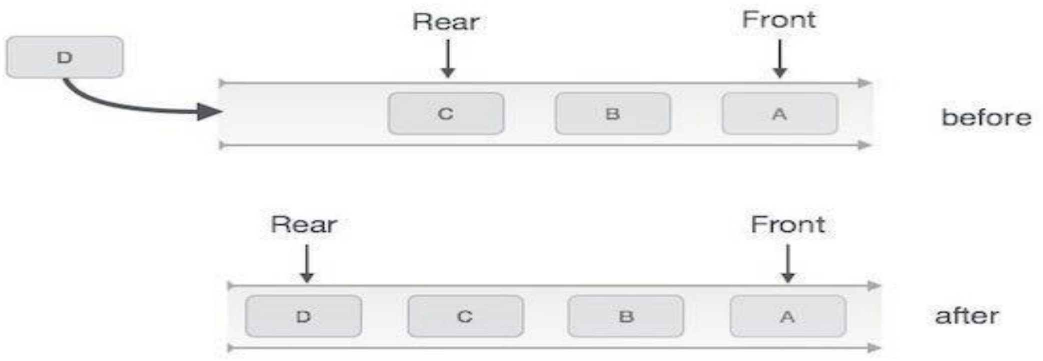
QUEUE is used when things don’t have to be processed immediately, but have to be processed in **F**irst **I**n **F**irst **O**ut order like Breadth First Search. This property of Queue makes it also useful in following kind of scenarios.

1) When a resource is shared among multiple consumers. Examples include CPU scheduling, Disk Scheduling.
2) When data is transferred asynchronously (data not necessarily received at same rate as sent) between two processes. Examples include IO Buffers, pipes, file IO, etc.

Queue Insertion 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 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.



Queue Enqueue

Sometimes, we also check to see if a queue is initialized or not, to handle any unforeseen situations.

Algorithm for insertion operation:

```
procedureenqueue(data)

if queue is full
return overflow
endif

rear← rear +1
queue[rear]← data
returntrue

end procedure
```

Implementation of enqueue() in C programming language –

Example

```
intenqueue(int data)
if(isfull())
return0;
```

```
rear=    rear    +1;
queue[rear]= data;

return1;
end procedure
```

Deletion 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 **deletion** operation −

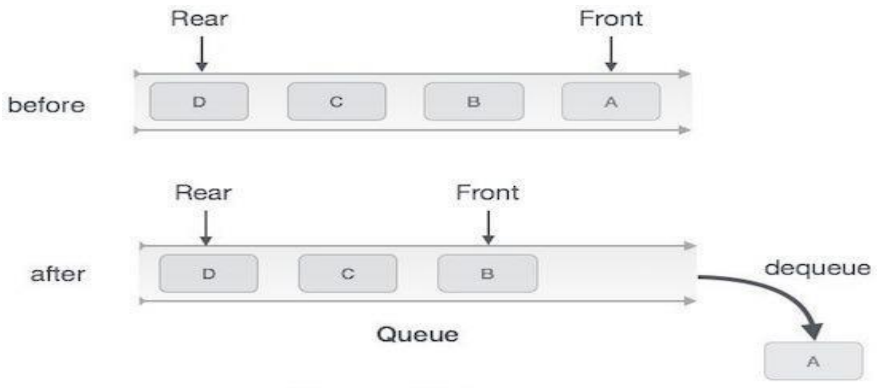
Step 1 − Check if the queue is empty.

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.



Queue Dequeue

Algorithm for deletion operation

```
proceduredequeue

if queue is empty
return  underflow
endif

data= queue[front]
front← front +1
returntrue

end procedure
```

Implementation of dequeue() in C programming language −

Example:

```
intdequeue(){
if(isempty())
return0;

int data = queue[front];
front= front +1;
```

```
        return data;
        }
```

LAB ASSIGNMENT

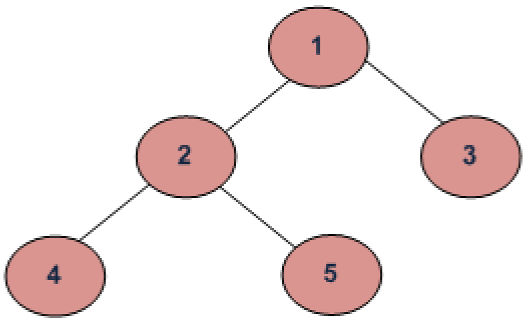
Task 1: Insert data into queue and then display them.
Task 2: Delete data from queue and display the deleted data.

EXPERIMENT NO – 9

Aim: ImplementPreorder, Inorder and Postorder Tree Traversal Algorithm.

Tree Traversals (Inorder, Preorder and Postorder):

Unlike linear data structures (Array, Linked List, Queues, Stacks, etc) which have only one logical way to traverse them, trees can be traversed in different ways. Following are the generally used ways for traversing trees.



Tree Traversals:

a) Inorder (Left, Root, Right) : 4 2 5 1 3
b) Preorder (Root, Left, Right) : 1 2 4 5 3
c) Postorder (Left, Right, Root) : 4 5 2 3 1

InorderTraversal :

```
Algorithm Inorder(tree)
  1. Traverse the left subtree, i.e., call Inorder(left-subtree)
  2. Visit the root.
  3. Traverse the right subtree, i.e., call Inorder(right-subtree)
```

Uses of Inorder:
In case of binary search trees (BST), Inorder traversal gives nodes in non-decreasing order. To get nodes of BST in non-increasing order, a variation of Inorder traversal where Inorder traversals reversed can be used.
Example: Inorder traversal for the above-given figure is 4 2 5 1 3.

Preorder Traversal:
Algorithm Preorder (tree)

1. Visit the root.
2. Traverse the left subtree, i.e., call Preorder(left-subtree)
3. Traverse the right subtree, i.e., call Preorder(right-subtree)

Uses of Preorder:
Preorder traversal is used to create a copy of the tree. Preorder traversal is also used to get prefix expression on of an expression tree.
Example: Preorder traversal for the above given figure is 1 2 4 5 3.

PostorderTraversal:

Algorithm Postorder(tree)
1. Traverse the left subtree, i.e., call Postorder(left-subtree)
2. Traverse the right subtree, i.e., call Postorder(right-subtree)
3. Visit the root.

Uses of Postorder:
Postorder traversal is used to delete the tree. Postorder traversal is also useful to get the postfix expression of an expression tree.
Example: Postorder traversal for the above given figure is 4 5 2 3 1.

LAB ASSIGNMENT

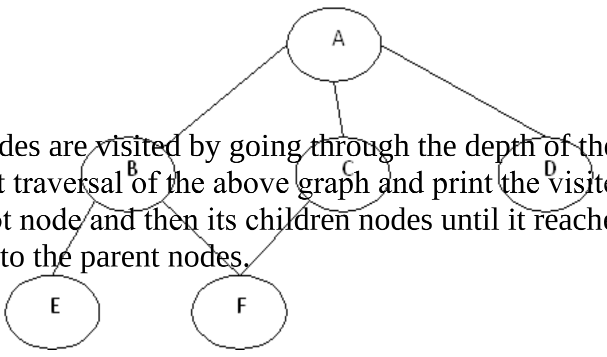
Task 1: Create a tree structure and traverse it using the algorithms mentioned above.

EXPERIMENT NO – 10

Aim: ImplementBFS and DFS Graph Traversal Algorithm.

Graph Traversal:
The breadth first search (BFS) and the depth first search (DFS) are the two algorithms used for traversing and searching a node in a graph. They can also be used to find out whether a node is reachable from a given node or not.

DEPTH FIRST SEARCH (DFS):
The aim of DFS algorithm is to traverse the graph in such a way that it tries to go far from the root node. Stack is used in the implementation of the depth first search. Let's see how depth first search works with respect to the following graph:



As stated before, in DFS, nodes are visited by going through the depth of the tree from the starting node. If we do the depth first traversal of the above graph and print the visited node, it will be "A B E F C D". DFS visits the root node and then its children nodes until it reaches the end node, i.e. E and F nodes, then moves up to the parent nodes.

__**Algorithmic Steps**__

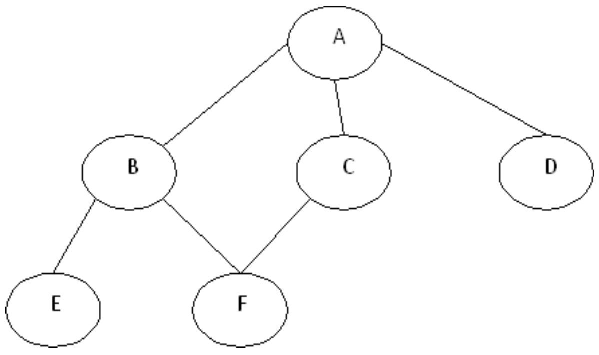
Step 1: Push the root node in the Stack.
Step 2: Loop until stack is empty.
Step 3: Peek the node of the stack.
Step 4: If the node has unvisited child nodes, get the unvisited child node, mark it as traversed and push it on stack.
Step 5: If the node does not have any unvisited child nodes, pop the node from the stack.

Based upon the above steps, the following Java code shows the implementation of the DFS algorithm:

```
publicvoiddfs()
```

```
{
        //DFS uses Stack data structure
        Stack s=new Stack();
        s.push(this.rootNode);
        rootNode.visited=true;
        printNode(rootNode);
        while(!s.isEmpty())
        {
                Node n=(Node)s.peek();
                Node child=getUnvisitedChildNode(n);
                if(child!=null)
                {
                        child.visited=true;
                        printNode(child);
                        s.push(child);
                }
                else
                {
                        s.pop();
                }
        }
        //Clear visited property of nodes
        clearNodes();
}
```

Breadth first search (BFS):
This is a very different approach for traversing the graph nodes. The aim of BFS algorithm is to traverse the graph as close as possible to the root node. Queue is used in the implementation of the breadth first search. Let's see how BFS traversal works with respect to the following graph:



If we do the breadth first traversal of the above graph and print the visited node as the output, it will print the following output. "A B C D E F". The BFS visits the nodes level by level, so it will start with level 0 which is the root node, and then it moves to the next levels which are B, C and D, then the last levels which are E and F.

Algorithmic Steps

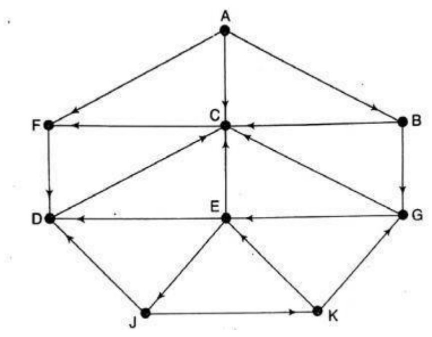
Step 1: Push the root node in the Queue.
Step 2: Loop until the queue is empty.
Step 3: Remove the node from the Queue.
Step 4: If the removed node has unvisited child nodes, mark them as visited and insert the unvisited children in the queue.

Based upon the above steps, the following Java code shows the implementation of the BFS algorithm:

```
//
publicvoidbfs()
{
        //BFS uses Queue data structure
        Queue q=new LinkedList();
        q.add(this.rootNode);
        printNode(this.rootNode);
        rootNode.visited=true;
        while(!q.isEmpty())
        {
                Node n=(Node)q.remove();
                Node child=null;
                while((child=getUnvisitedChildNode(n))!=null)
                {
                        child.visited=true;
                        printNode(child);
                        q.add(child);
                }
        }
        //Clear visited property of nodes
        clearNodes();
}
//
```

LAB ASSIGNMENT

Task1: Consider the below directed graph G. Suppose G represent the daily flights of airlines between different cities. Suppose you want to fly from city A to city K with the minimum number of stops. Which traversing algorithm do you choose? Find the minimum path from A to K.



EXPERIMENT NO – 11

Aim:

1. Sorting the list of integers in ascending order using insertion sort.
2. Sorting the list of integers in ascending order using Selection sort

PROGRAM LOGIC:

Insertion sort

1. Read the elements to be sort
2. Select the minimum element
3. Apply the selection sort to sort the remaining elements

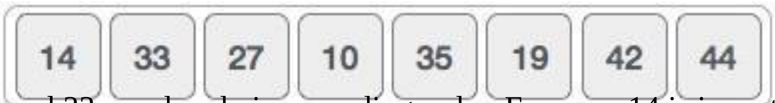
Selection sort

1. Read the elements to be sort
2. Select the minimum element
3. Apply the selection sort to sort the remaining elements

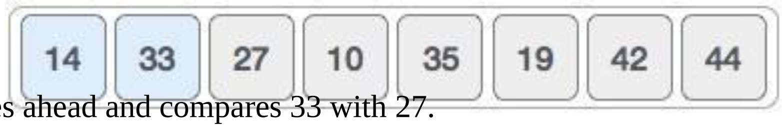
How Insertion Sort Works?

We take an unsorted array for our example.

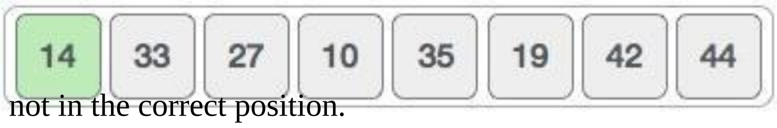
Insertion sort compares the first two elements.



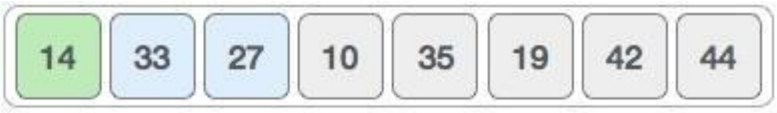
It finds that both 14 and 33 are already in ascending order. For now, 14 is in sorted sub-list.

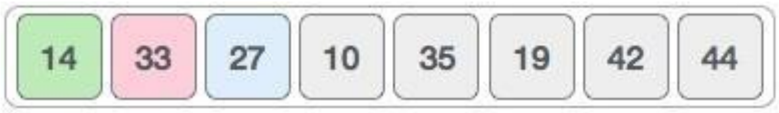


Insertion sort moves ahead and compares 33 with 27.

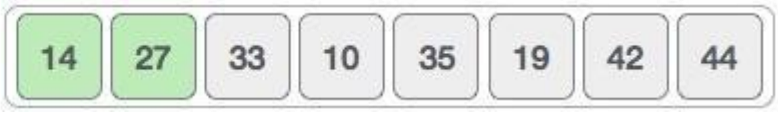


And finds that 33 is not in the correct position.

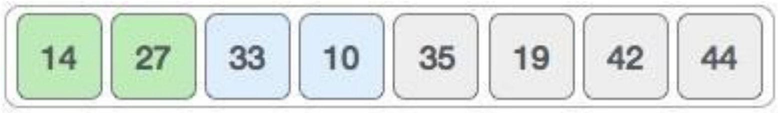




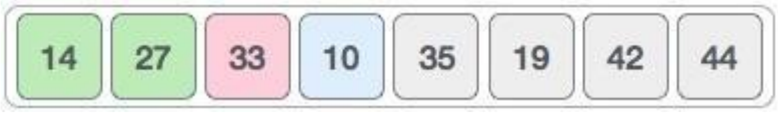
It swaps 33 with 27. It also checks with all the elements of sorted sub-list. Here we see that the sorted sub-list has only one element 14, and 27 is greater than 14. Hence, the sorted sub-list remains sorted after swapping.



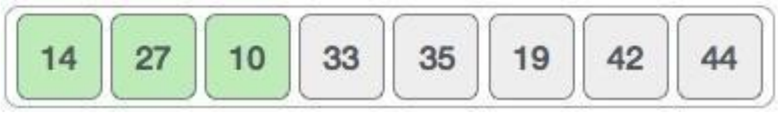
By now we have 14 and 27 in the sorted sub-list. Next, it compares 33 with 10.



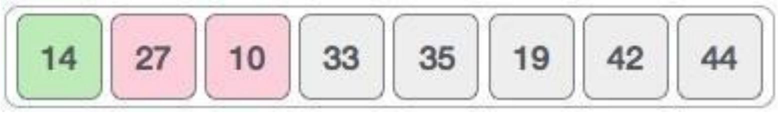
These values are not in a sorted order.



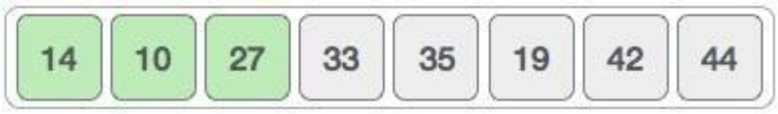
So we swap them.



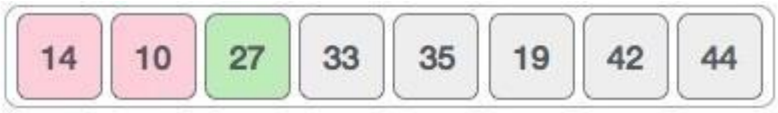
However, swapping makes 27 and 10 unsorted.



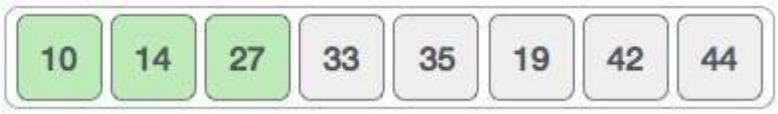
Hence, we swap them too.



Again we find 14 and 10 in an unsorted order.



We swap them again. By the end of third iteration, we have a sorted sub-list of 4 items.



This process goes on until all the unsorted values are covered in a sorted sub-list. Now we shall see some programming aspects of insertion sort.

<u>Insertion sort:</u>

Algorithm:

1) Repeat step 2 to 5 for K=1 to n-1

2) Set temp=arr[k]
3) Set j=k-1
4) Repeat while temp <=arr[j]&&j>=0
Set arr[j+1]=arr[j]
Set j=j-1
5) Set arr [j+1] = temp
6) Exit

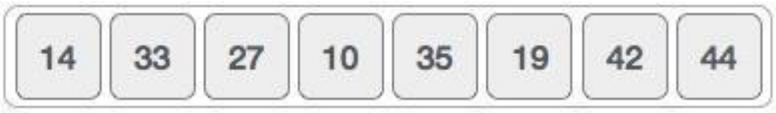
Pseudocode:

```
procedureinsertionSort( A : array of items )
intholePosition
intvalueToInsert

fori=1 to length(A) inclusive do:

/* select value to be inserted */
valueToInsert= A[i]
holePosition=i

/*locate hole position for the element to be inserted */

whileholePosition>0and A[holePosition-1]>valueToInsertdo:
        A[holePosition]=A[holePosition-1]
holePosition=holePosition-1
endwhile

/* insert the number at hole position */
      A[holePosition]=valueToInsert

endfor

end procedure
```

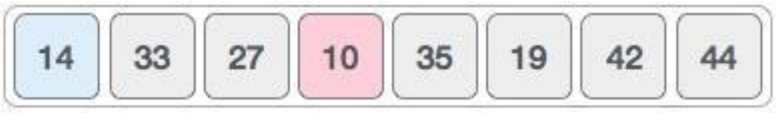
Selection sort

How Selection Sort Works?

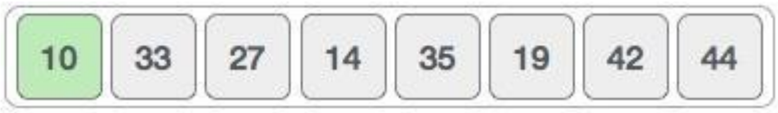
Consider the following depicted array as an example.



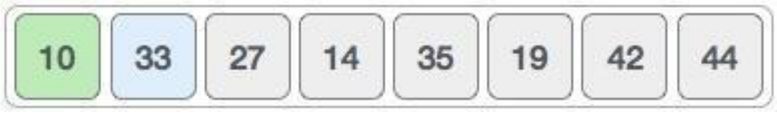
For the first position in the sorted list, the whole list is scanned sequentially. The first position where 14 is stored presently, we search the whole list and find that 10 is the lowest value.



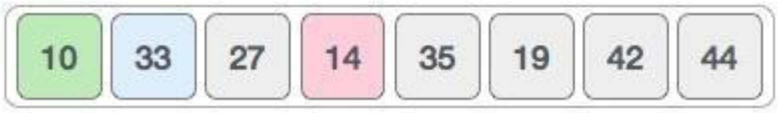
So we replace 14 with 10. After one iteration 10, which happens to be the minimum value in the list, appears in the first position of the sorted list.



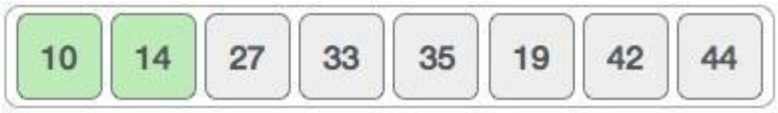
For the second position, where 33 is residing, we start scanning the rest of the list in a linear manner.



We find that 14 is the second lowest value in the list and it should appear at the second place. We swap these values.

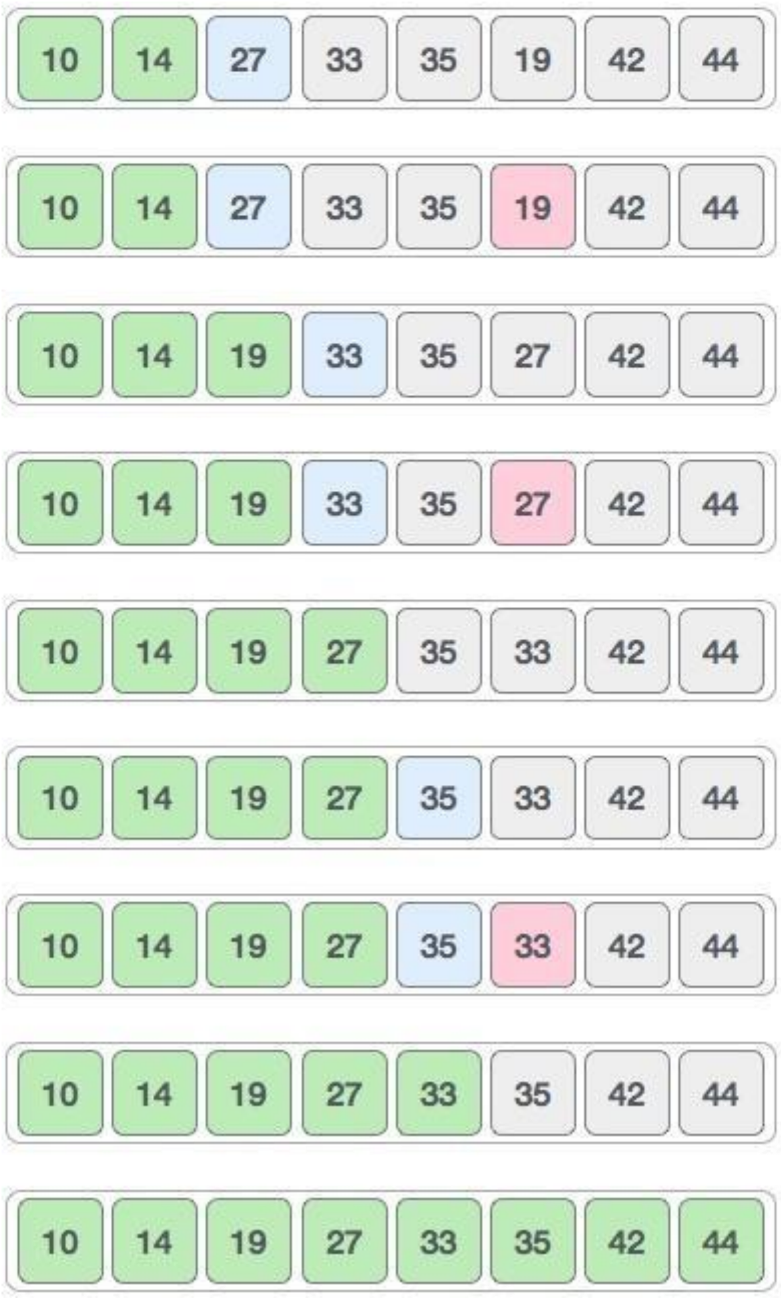


After two iterations, two least values are positioned at the beginning in a sorted manner.



The same process is applied to the rest of the items in the array.

Following is a pictorial depiction of the entire sorting process −



Now, let us learn some programming aspects of selection sort.

Algorithm:

```
1)  Repeat step 2 and 3 for K=1 to n-1
2)  Call smallest (arr ,k,n,pos)
3)  Swap a[k] with arr[pos]
4)  Exit
Smallest(arr,k,n,pos)
1)  Set small = arr[k]
2)  Set pos=k
3)  Repeat for j=k+1 to n-1
If small>arr[j]
Set pos=j
[end of if]
[end of loop]
```

4) Exit

<u>Pseudocode:</u>

```
procedure selection sort
list : array of items
n: size of list

fori=1 to n -1
/* set current element as minimum*/
min=i

/* check the element to be minimum */

for j = i+1 to n
if list[j]< list[min]then
min= j;
endif
endfor

/* swap the minimum element with the current element*/
ifindexMin!=ithen
swap list[min]and list[i]
endif
endfor

end procedure
```

SOURCE CODE:

```
#include<stdio.h>
voidsel_sort(int[]);
void main()
{
        intnum[5],count;
        printf("enter the five elements to sort:\n");
        for(count=0;count<5;count++)
        scanf("%d",&num[count]);
        sel_sort(num); /*function call*/
        printf("\n\n elements after sorting:\n");
        for(count=0;count<5;count++)
        printf("%d\n",num[count]);
}
/*called function*/
        voidsel_sort(intnum[])
        {
                inti,j,min,temp;
                for(j=0;j<5;j++)
                {
                        min=j;
                        for(i=j;i<5;i++)
                        if(num[min]>num[i])
                                min=i;
                        if(min<5)
                        {
                                temp=num[j];
                                num[j]=num[min];
                                num[min]=temp;
                }
                printf("%d\t",num[j]);
                }
        }
```

LAB ASSIGNMENT

1. Rearrange the following numbers using Insertion sort procedure. 42, 12, 18, 98, 67, 83, 8, 10, 71
2. Apply the selection sort on the following elements 21,11,5,78,49, 54,72,88

EXPERIMENT NO – 12

Aim: Final Lab Examination

Task 1: Quiz
Task 2: Three Lab Problem Solving.
Task 3: Viva Voce