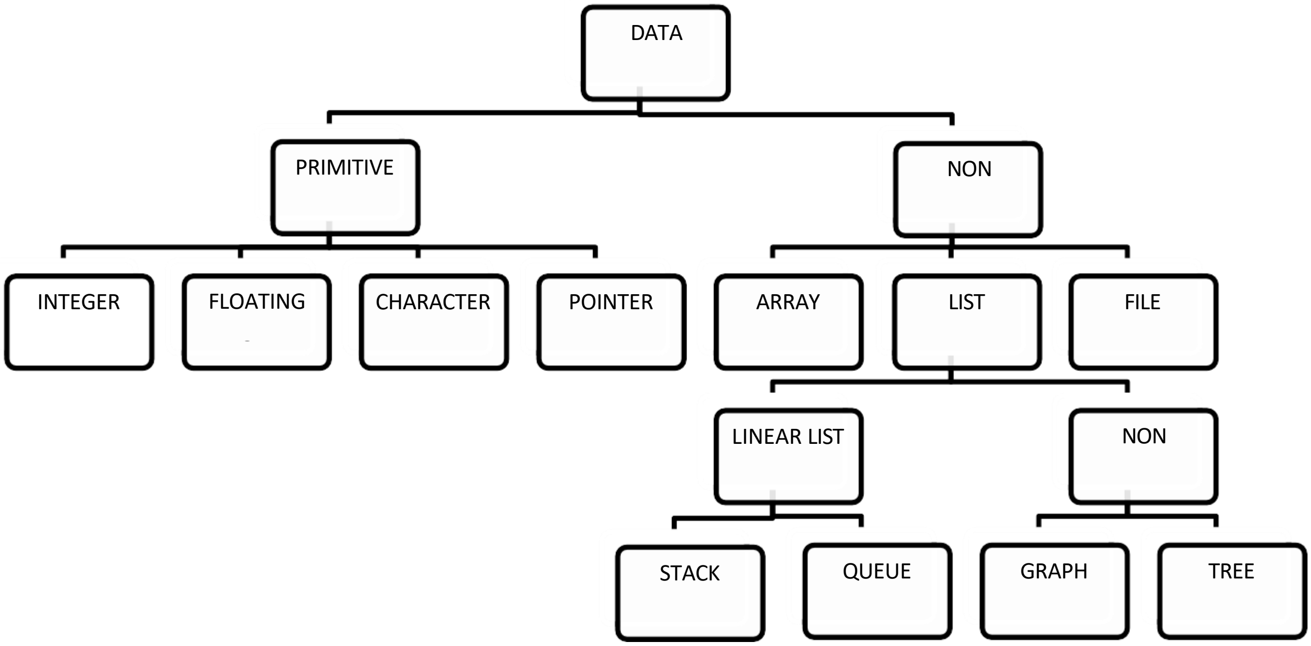
Introduction to Data Structure

- Computer is an electronic machine which is used for data processing and manipulation.
- When programmer collects such type of data for processing, he would require to store all of them in computer's main memory.
- In order to make computer work we need to know
  - Representation of data in computer.
  - Accessing of data.
  - How to solve problem step by step.
- For doing this task we use data structure.

What is Data Structure?

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

Classification of Data Structure



Data Structures are normally classified into two broad categories

1. Primitive Data Structure
2. Non-primitive data Structure

Data types

A particular kind of data item, as defined by the values it can take, the programming language used, or the operations that can be performed on it.

Primitive Data Structure

- Primitive data structures are basic structures and are directly operated upon by machine instructions.
- Primitive data structures have different representations on different computers.
- Integers, floats, character and pointers are examples of primitive data structures.
- These data types are available in most programming languages as built in type.
  - Integer: It is a data type which allows all values without fraction part. We can use it for whole numbers.
  - Float: It is a data type which use for storing fractional numbers.
  - Character: It is a data type which is used for character values.

Pointer: A variable that holds memory address of another variable are called pointer.

Non primitive Data Type

- These are more sophisticated data structures.
- These are derived from primitive data structures.
- The non-primitive data structures emphasize on structuring of a group of homogeneous or heterogeneous data items.
- Examples of Non-primitive data type are Array, List, and File etc.
- A Non-primitive data type is further divided into Linear and Non-Linear data structure
  - **Array:** An array is a fixed-size sequenced collection of elements of the same data type.
  - **List:** An ordered set containing variable number of elements is called as Lists.
  - **File:** A file is a collection of logically related information. It can be viewed as a large list of records consisting of various fields.

Linear data structures

- A data structure is said to be Linear, if its elements are connected in linear fashion by means of logically or in sequence memory locations.
- There are two ways to represent a linear data structure in memory,
  - Static memory allocation
  - Dynamic memory allocation
- The possible operations on the linear data structure are: Traversal, Insertion, Deletion, Searching, Sorting and Merging.
- Examples of Linear Data Structure are Stack and Queue.
- Stack: Stack is a data structure in which insertion and deletion operations are performed at one end only.
  - The insertion operation is referred to as 'PUSH' and deletion operation is referred to as 'POP' operation.
  - Stack is also called as Last in First out (LIFO) data structure.
- Queue: The data structure which permits the insertion at one end and Deletion at another end, known as Queue.
  - End at which deletion is occurs is known as FRONT end and another end at which insertion occurs is known as REAR end.
  - Queue is also called as First in First out (FIFO) data structure.

Nonlinear data structures

- Nonlinear data structures are those data structure in which data items are not arranged in a sequence.
- Examples of Non-linear Data Structure are Tree and Graph.
- **Tree:** A tree can be defined as finite set of data items (nodes) in which data items are arranged in branches and sub branches according to requirement.
  - Trees represent the hierarchical relationship between various elements.
  - Tree consist of nodes connected by edge, the node represented by circle and edge lives connecting to circle.

- **Graph:** Graph is a collection of nodes (Information) and connecting edges (Logical relation) between nodes.
    - A tree can be viewed as restricted graph.
    - Graphs have many types:
        - Un-directed Graph
        - Directed Graph
        - Mixed Graph
        - Multi Graph
        - Simple Graph
        - Null Graph
        - Weighted Graph

Difference between Linear and Non Linear Data Structure

Linear Data Structure	**Non-Linear Data Structure**
Every item is related to its previous and next time.	Every item is attached with many other items.
Data is arranged in linear sequence.	Data is not arranged in sequence.
Data items can be traversed in a single run.	Data cannot be traversed in a single run.
Eg. Array, Stacks, linked list, queue.	Eg. tree, graph.
Implementation is easy.	Implementation is difficult.

Operation on Data Structures

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

1. **Create**
   The create operation results in reserving memory for program elements. This can be done by declaration statement. Creation of data structure may take place either during compile-time or run-time. malloc() function of C language is used for creation.

2. **Destroy**
   Destroy operation destroys memory space allocated for specified data structure. free() function of C language is used to destroy data structure.

3. **Selection**
   Selection operation deals with accessing a particular data within a data structure.

4. **Updation**
   It updates or modifies the data in the data structure.

5. **Searching**
   It finds the presence of desired data item in the list of data items, it may also find the locations of all elements that satisfy certain conditions.

6. **Sorting**
   Sorting is a process of arranging all data items in a data structure in a particular order, say for example, either in ascending order or in descending order.

7. **Merging**
   Merging is a process of combining the data items of two different sorted list into a single sorted list.

8. **Splitting**
   Splitting is a process of partitioning single list to multiple list.

9. **Traversal**
   Traversal is a process of visiting each and every node of a list in systematic manner.

Time and space analysis of algorithms

Algorithm

- An essential aspect to data structures is algorithms.
- Data structures are implemented using algorithms.
- An algorithm is a procedure that you can write as a C function or program, or any other language.
- An algorithm states explicitly how the data will be manipulated.

Algorithm Efficiency

- Some algorithms are more efficient than others. We would prefer to choose an efficient algorithm, so it would be nice to have metrics for comparing algorithm efficiency.
- The complexity of an algorithm is a function describing the efficiency of the algorithm in terms of the amount of data the algorithm must process.
- Usually there are natural units for the domain and range of this function. There are two main complexity measures of the efficiency of an algorithm
- **Time complexity**
  - **Time Complexity** is a function describing the amount of time an algorithm takes in terms of the amount of input to the algorithm.

  - "Time" can mean the number of memory accesses performed, the number of comparisons between integers, the number of times some inner loop is executed, or some other natural unit related to the amount of real time the algorithm will take.

- **Space complexity**

  - **Space complexity** is a function describing the amount of memory (space) an algorithm takes in terms of the amount of input to the algorithm.

  - We often speak of "extra" memory needed, not counting the memory needed to store the input itself. Again, we use natural (but fixed-length) units to measure this.

  - We can use bytes, but it's easier to use, say, number of integers used, number of fixed-sized structures, etc. In the end, the function we come up with will be independent of the actual number of bytes needed to represent the unit.

  - Space complexity is sometimes ignored because the space used is minimal and/or obvious, but sometimes it becomes as important an issue as time.

Worst Case Analysis

In the worst case analysis, we calculate upper bound on running time of an algorithm. We must know the case that causes maximum number of operations to be executed. For Linear Search, the worst case happens when the element to be searched is not present in the array. When x is not present, the search () functions compares it with all the elements of array [] one by one. Therefore, the worst case time complexity of linear search would be.

Average Case Analysis

In average case analysis, we take all possible inputs and calculate computing time for all of the inputs. Sum all the calculated values and divide the sum by total number of inputs. We must know (or predict) distribution of cases. For the linear search problem, let us assume that all cases are uniformly distributed. So we sum all the cases and divide the sum by (n+1).

Best Case Analysis

In the best case analysis, we calculate lower bound on running time of an algorithm. We must know the case that causes minimum number of operations to be executed. In the linear search problem, the best case occurs when x is present at the first location. The number of operations in worst case is constant (not dependent on n). So time complexity in the best case would be.

(1) Asymptotic Notations.

Asymptotic notation is used to describe the running time of an algorithm. It shows order of growth of function.

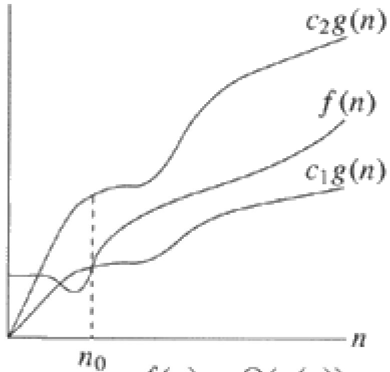
Θ-Notation (Same order)

- For a given function g(n), we denote by **Θ(*g*(*n*))** the set of functions

Θ(g(n)) = { f(n) : there exist positive constants c_1, c_2 and n_0 such that 0

$\leq c_1g(n) \leq f(n) \leq c_2g(n)$ for all $n \geq n_0$ }

- Because **Θ(*g*(*n*))** is a set, we could write f(n) € **Θ(*g*(*n*))** to indicate that f(n) is a member of **Θ(*g*(*n*)).**
- This notation bounds a function to within constant factors. We say *f*(*n*) = Θ(*g*(*n*)) if there exist positive constants n_0, c_1 and c_2 such that to the right of n_0 the value of *f*(*n*) always lies between $c_1g(n)$ and $c_2g(n)$ inclusive.
- Figure a gives an intuitive picture of functions f(n) and g(n). For all values of n to the right of n_0, the value of f(n) lies at or above $c_1g(n)$ and at or below $c_2g(n)$. In other words, for all $n \geq n_0$, the value of f(n) is equal to g(n) to within a constant factor.
- We say that g(n) is an asymptotically tight bound for f(n).



$f(n) = \Theta(g(n))$

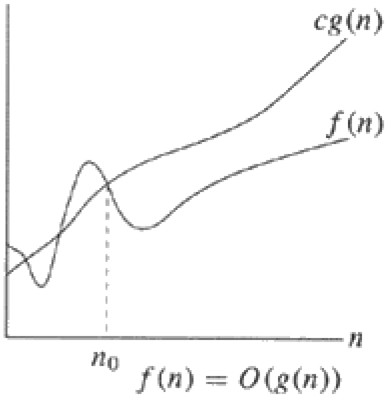
O-Notation (Upper Bound)

- For a given function g(n), we denote by **O(*g*(*n*))** the set of functions

O(g(n)) = { f(n) : there exist positive constants c and n_0 such that
$0 \leq f(n) \leq cg(n)$ for all $n \geq n_0$ }

- We use O notation to give an upper bound on a function, to within a constant factor. For all values of n to the right of n_0, the value of the function f(n) is on or below g(n).

- This notation gives an upper bound for a function to within a constant factor. We write *f*(*n*) = *O*(*g*(*n*)) if there are positive constants *n*$_0$ and *c* such that to the right of *n*$_0$, the value of *f*(*n*) always lies on or below *cg*(*n*).
- We say that g(n) is an asymptotically upper bound for f(n).



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

Example:

Let f(n)=n^2 and g(n)=2^n

n	f(n)=n^2	g(n)=2^n	
1	1	2	f(n) < g(n)
2	4	4	f(n) = g(n)
3	9	8	f(n) > g(n)
4	16	16	f(n) = g(n)
5	25	32	f(n) < g(n)
6	36	64	f(n) < g(n)
7	49	128	f(n) < g(n)

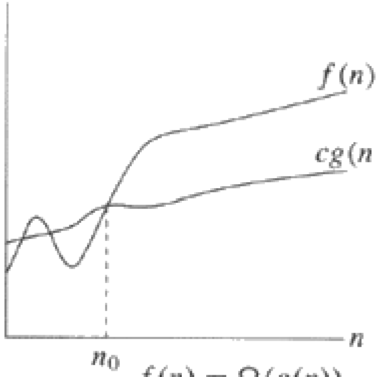
Here for n ≥ 4 we have
behavior f (n) ≤ g(n)
Where n_0=4

Ω-Notation (Lower Bound)

- For a given function g(n), we denote by **Ω(*g*(*n*))** the set of functions

Ω (g(n)) = { f(n) : there exist positive constants c and n_0 such that 0 ≤ cg(n) ≤ f (n)for all n ≥ n_0 }

- **Ω Notation provides an asymptotic lower bound.** For all values of n to the right of n_0, the value of the function f(n) is on or above cg(n).
- This notation gives a lower bound for a function to within a constant factor. We write *f*(*n*) = Ω(*g*(*n*)) if there are positive constants *n*$_0$ and *c* such that to the right of *n*$_0$, the value of *f*(*n*) always lies on or above *cg*(*n*).



$f(n) = \Omega(g(n))$

Example:

Let f(n)= 2^n and g(n)= n^2

n	f(n)= 2^n	g(n)= n^2	
1	2	1	f(n) > g(n)
2	4	4	f(n) = g(n)
3	8	9	f(n) < g(n)
4	16	16	f(n) = g(n)
5	32	25	f(n) > g(n)
6	64	36	f(n) > g(n)
7	128	49	f(n) > g(n)

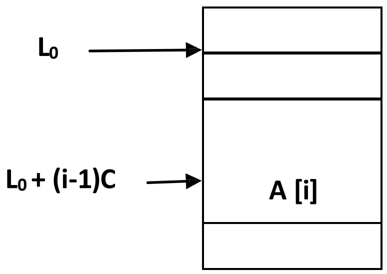
Here for n ≥ 4 we have behavior f (n) ≥ g(n)

Where n_0=4

Explain Array in detail

One Dimensional Array

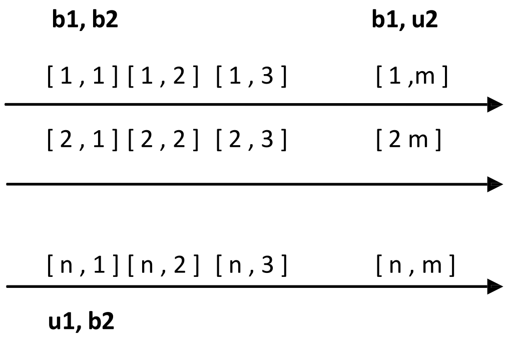
- Simplest data structure that makes use of computed address to locate its elements is the one-dimensional array or vector; number of memory locations is sequentially allocated to the vector.
- A vector size is fixed and therefore requires a fixed number of memory locations.
- Vector A with subscript lower bound of “one” is represented as below….

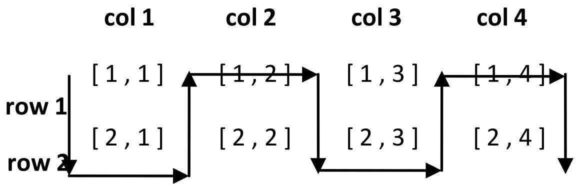


- L_0 is the address of the first word allocated to the first element of vector A.
- C words are allocated for each element or node
- The address of A_i is given equation **Loc $(A_i) = L_0 + C\ (i-1)$**
- Let’s consider the more general case of representing a vector A whose lower bound for it’s subscript is given by some variable b. The location of Ai is then given by **Loc $(A_i) = L_0 + C\ (i-b)$**

Two Dimensional Array

- Two dimensional arrays are also called table or matrix, two dimensional arrays have two subscripts
- Two dimensional array in which elements are stored column by column is called as column major matrix
- Two dimensional array in which elements are stored row by row is called as row major matrix
- First subscript denotes number of rows and second subscript denotes the number of columns
- Two dimensional array consisting of two rows and four columns as above Fig is stored sequentially by columns : A [1, 1], A [2 , 1], A [1 , 2], A [2 , 2], A [1 , 3], A [2 , 3], A [1, 4], A [2, 4]
- The address of element A [i , j] can be obtained by expression **Loc (A [i , j]) = L_0 + (j-1)*2 + i-1**
- In general for two dimensional array consisting of n rows and m columns the address element A [i , j] is given by **Loc (A [i , j]) = L_0 + (j-1)*n + (i – 1)**
- In row major matrix, array can be generalized to arbitrary lower and upper bound in its subscripts, assume that $b1 \leq I \leq u1$ and $b2 \leq j \leq u2$





Row major matrix

No of Columns = m = u2 – b2 + 1

Column major matrix

- For row major matrix : **Loc (A [i , j]) = L_0 + (i – b1) *(u2-b2+1) + (j-b2)**

(2) Insertion sort

- Insertion Sort works by inserting an element into its appropriate position during each iteration.
- Insertion sort works by comparing an element to all its previous elements until an appropriate position is found.
- Whenever an appropriate position is found, the element is inserted there by shifting down remaining elements.

Algorithm

```
Procedure insert (T[1….n])                      cost        times
          for i ← 2 to n do                      C1          n
                x ← T[i]                         C2          n-1
               j  ← i - 1                        C3          n-1
        while j > 0 and x < T[j] do              C4          $\sum_{i=2}^{n}$ Tj
          T[j+1] ← T[j]                          C5          $\sum_{i=2}^{n}$Tj−1
           j ← j  - 1                            C6          $\sum_{i=2}^{n}$ Tj−1
          end
      T[j + 1] ← x                               C7          n-1
      end
```

Analysis

- The running time of an algorithm on a particular input is the number of primitive operations or "steps" executed.
- A constant amount of time is required to execute each line of our pseudo code. One line may take a different amount of time than another line, but we shall assume that each execution of the i^{th} line takes time c_i, where c_i is a constant.
- The running time of the algorithm is the sum of running times for each statement executed; a statement that takes c_i steps to execute and is executed *n* times will contribute c_in to the total running time.
- Let the time complexity of selection sort is given as T(n), then

  T(n) = C_1n+ C_2(n-1)+ C_3(n-1)+ C_4($\sum_{i=2}^{n} T$j)+(C_5 +C_6) $\sum_{i=2}^{n} T$j $-$ 1+ C_7(n-1).

Where,

$$\sum_{i=2}^{n} T\mathrm{j} = \sum_{i=2}^{n} n - i$$

Best case:

Take $j = 1$

$T(n) = C_1n + C_2(n\text{-}1) + C_3(n\text{-}1) + C_4(n\text{-}1) + C_5\ (n\text{-}1) + c_7(n\text{-}1)$

$= (C_1+C_2+C_3+C_4+C_7)\ n\ - (C_2+C_3+C_4+C_7)$

$= an\ - b$

Thus, $T(n) = \Theta(n)$

Worst case: Take j =n

$T\ (n) = C_1n + C_2(n\text{-}1) + C_3(n\text{-}1) + C_4(\sum^4\quad {}_{i=2}\ T\mathrm{j}) + (C_5 + C_6) \sum^4 {}^{i=2}\ T\mathrm{j} - 1 + C_7(n\text{-}1).$

$= C\ n + C\ n + C\ n + C$ $\quad n \quad n \quad n \quad n\hat{}2 \quad n\hat{}2 \quad n\hat{}2$

$1 \quad 2 \quad 3 \quad 4 \frac{}{2} + C_5\ \frac{}{2} + C_6\ \frac{}{2} + C_4\ \frac{}{2} + C_5\ \frac{}{2} + C_6\ \frac{}{2} + C_7n - C_2 - C_3 - C_{7.}$

$2 \quad 1 \quad 1 \quad 1 \quad 1 \quad 1 \quad 1$

$= n\ (C_4\ \frac{}{2} + C_5\ \frac{}{2} + C_6\ \frac{}{2}) + n(C_1 + C_2 + C_3 + C_7 + C_4\ \ \frac{}{2} + C_5 \frac{}{2} + C_6\ \frac{}{2}) - 1(C_2 + C_3 + C_7).$

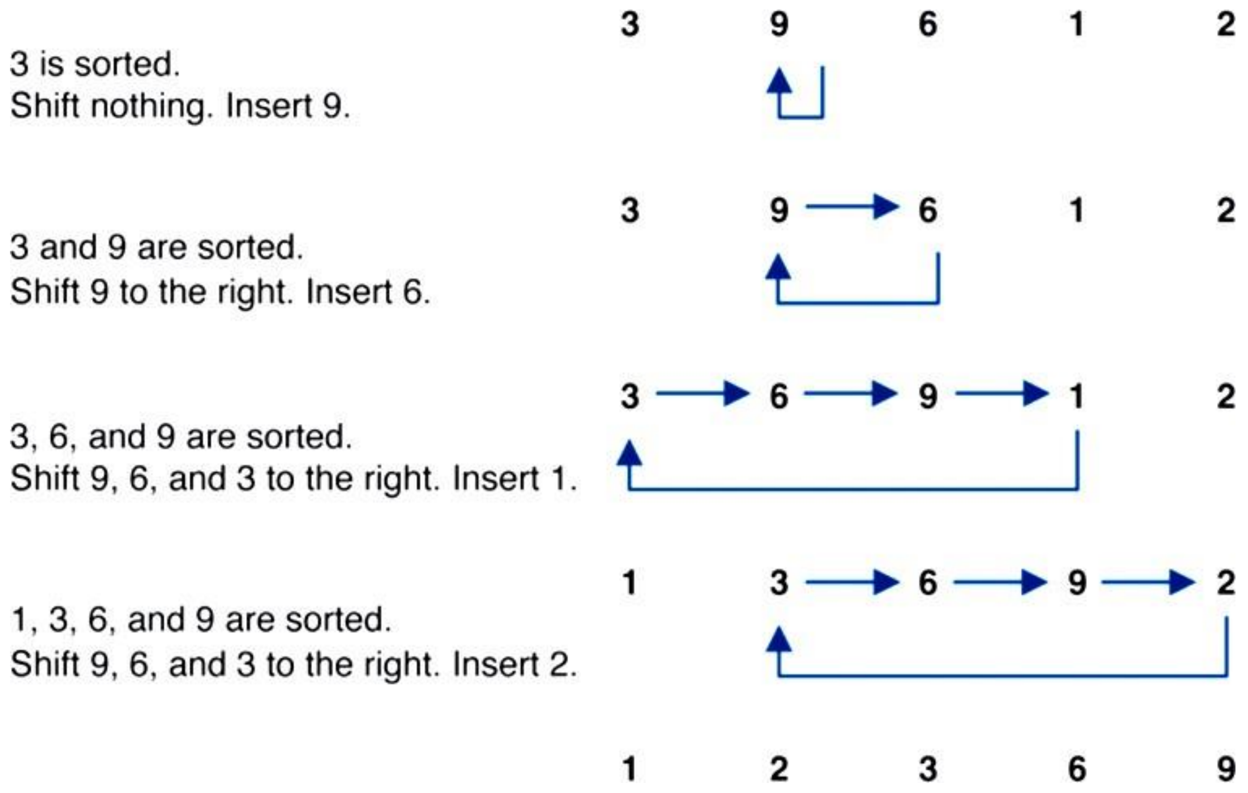
$= an^2 + bn + c.$

Thus, $T(n) = \Theta(n^2)$

Average case: Average case will be same as worst case $T(n) = \Theta(n^2)$ Time complexity of insertion sort

Best case	Average case	Worst case
O(n)	O (n^2)	O (n^2)

Example:


3 9 6 1 2
3 is sorted.
Shift nothing. Insert 9.
3 9 6 1 2
3 and 9 are sorted.
Shift 9 to the right. Insert 6.
3 6 9 1 2
3, 6, and 9 are sorted.
Shift 9, 6, and 3 to the right. Insert 1.
1 3 6 9 2
1, 3, 6, and 9 are sorted.
Shift 9, 6, and 3 to the right. Insert 2.
1 2 3 6 9

(3) Loop Invariant and the correctness of algorithm

We use loop invariants to help us understand why an algorithm is correct. We must show three things about a loop invariant:

1) **Initialization**: It is true prior to the first iteration of the loop.
2) **Maintenance**: If it is true before an iteration of the loop, it remains true before the next iteration.
3) **Termination**: When the loop terminates, the invariant gives us a useful property that helps show that the algorithm is correct.

- When the first two properties hold, the loop invariant is true prior to every iteration of the loop.
- Note the similarity to mathematical induction, where to prove that a property holds, you prove a base case and an inductive step.
- Here, showing that the invariant holds before the first iteration is like the base case, and showing that the invariant holds from iteration to iteration is like the inductive step.
- The third property is perhaps the most important one, since we are using the loop invariant to show correctness.
- It also differs from the usual use of mathematical induction, in which the inductive step is used infinitely; here, we stop the "induction" when the loop terminates.

(4) Bubble sort

- In the bubble sort, the consecutive elements of the table are compared and if the keys of the two elements are not found in proper order, they are interchanged.
- It starts from the beginning of the table and continue till the end of the table. As a result of this the element with the largest key will be pushed to the last element's position.
- After this the second pass is made. The second pass is exactly like the first one except that this time the elements except the last are considered. After the second pass, the next largest element will be pushed down to the next to last position.

Algorithm

```
Procedure bubble (T[1….n])                      cost      times
for i ← 1 to n do                                C1        n+1
      for i ← 1 to n-i do                        C2        $\sum_{i=1}^{n}(n+1-i)$
            if T[i] > T[j]                       C3        $\sum_{i=1}^{n}(n-i)$
                  T[i] ↔ T[j]                    C4        $\sum_{i=1}^{n}(n-i)$
       end
end
```

Analysis

$$T(n)=C1\ (n+1) + C2 \sum_{i=1}^{n}(n+1-i) + C3 \sum_{i=1}^{n}(n-i) + C4 \sum_{i=1}^{n}(n-i)$$

Best case:

Take i = 1

$T(n) = C_1n + C_1 + C_2n + C_3n - C_3 + C_4 n - C_4$

$= (C_1+C_2+C_3+C_4)\ n - (C_2, C_3, C_4, C_7)$

$= an - b$

Thus, $T(n) = \Theta(n)$

Worst Case:

$$= C_1n + C_1 + C_2 n + C_2 - C_2 \left(\frac{n(n+1)}{2}\right) + C_3n - C_3 \left(\frac{n(n+1)}{2}\right) + C_4n - C_4 \left(\frac{n(n+1)}{2}\right)$$

$= [-C_2/n^2-C_3/n^2-C_4/n^2] + [-C_2/n-C_3/n-C_4/n]+C_1+C_2+C_3+C_4$

$= an^2+bn+c$

$= \Theta(n^2)$

Average case: Average case will be same as worst case $T(n) = \Theta(n^2)$

Example:

Consider the following numbers are stored in an array: Original
Array: 32,51,27,85,66,23,13,57
Pass 1 : 32,27,51,66,23,13,57,85
Pass 2 : 27,33,51,23,13,57,66,85
Pass 3 : 27,33,23,13,51,57,66,85
Pass 4 : 27,23,13,33,51,57,66,85Pass 5 : 23,13,27,33,51,57,66,85
Pass 6 : 13,23,27,33,51,57,66,85

(5) Selection sort

- Selection Sort works by repeatedly selecting elements.
- The algorithm finds the smallest element in the array first and exchanges it with the element in the first position.

Then it finds the second smallest element and exchanges it with the element in the second position and continues in this way until the entire array is sorted.

Algorithm

Procedure select (T [1….n])	**Cost**	**times**
for i←1 **to** n-1 **do**	C1	n
minj ← i ; minx ← T[i]	C2	n-1
for j←i+1 **to** n **do**	C3	$\sum_{i=1}^{n-1}(i+1)$
if T[j] < minx **then** minj ← j	C4	$\sum_{i=1}^{n-1} i$
minx ← T[j]	C5	$\sum_{i=1}^{n-1} i$
T[minj] ← T[i]	C6	n-1
T[i] ← minx	C7	n-1

Analysis

$T(n)=C_1n+ C_2(n-1)+ C_3(\sum_{i=1}^{n-1}(i+1)) + C_4(\sum_{i=1}^{n-1}(i))+C_5(\sum_{i=1}^{n-1}(i))+C_6(n-1)+ C_7(n-1)$

$= C_1 n+ C_2 n+ C_6 n+ C_7 n+ C_3\frac{n}{2} +C_4\frac{n}{2}+ C_5\frac{n}{2}+ C_3\frac{n^2}{2} + C_4\frac{n^2}{2}+ C_5\frac{n^2}{2}-C_2-C_6-C_7$

$= n(C_1+ C_2+ C_6 +C_7 + C_3$ — 2

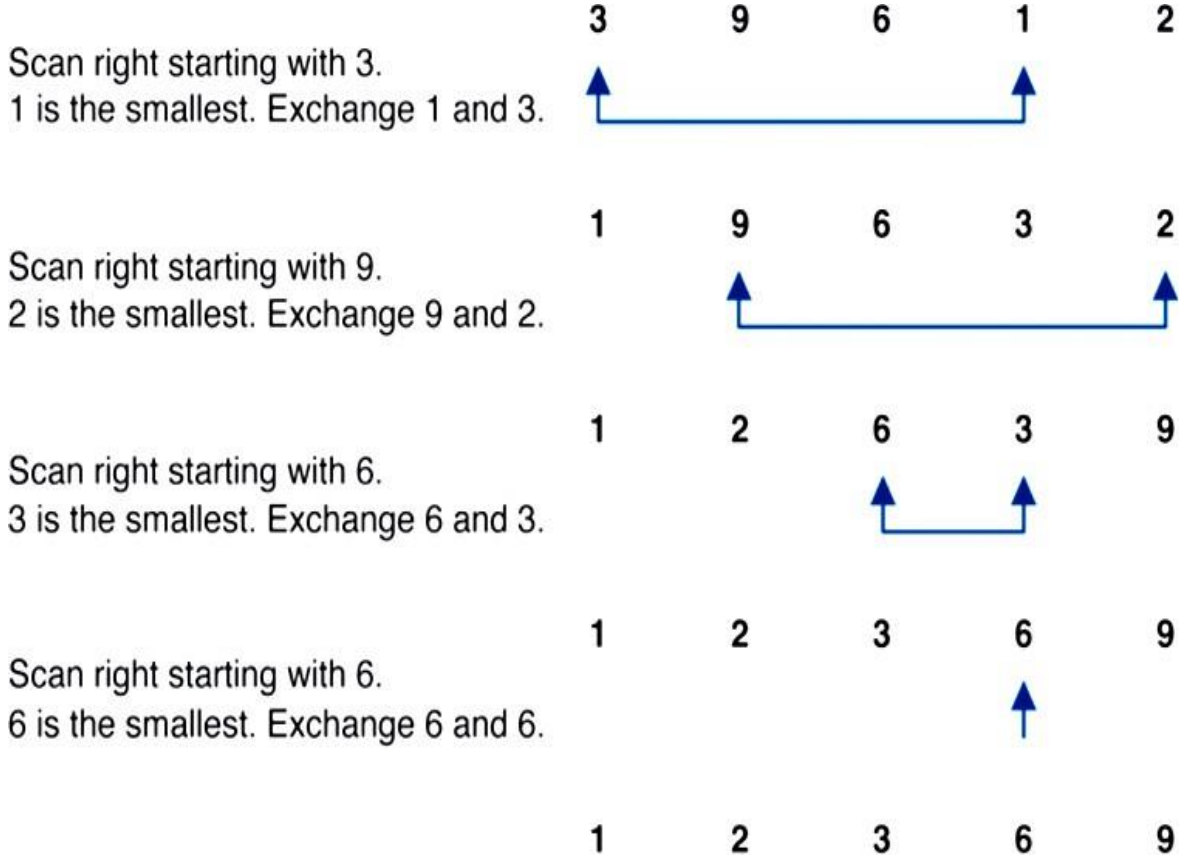
$+ C_4 \frac{}{2} + C_5 \frac{}{2}) + n\ (C_3\ {}_2 + C_4 \frac{}{2} + C_5 \frac{}{2})-1(C_2+ C_6 +C_7)$

$= an^2+bn+c$

Time complexity of insertion sort

Best case	Average case	Worst case
$O(n^2)$	$O\ (n^2)$	$O\ (n^2)$

Example:



Scan right starting with 3.
1 is the smallest. Exchange 1 and 3.

Scan right starting with 9.
2 is the smallest. Exchange 9 and 2.

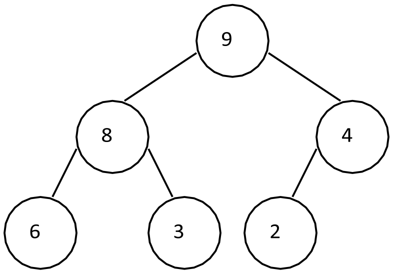
Scan right starting with 6.
3 is the smallest. Exchange 6 and 3.

Scan right starting with 6.
6 is the smallest. Exchange 6 and 6.

(6) Heap

- A heap data structure is a binary tree with the following properties.
  1. It is a complete binary tree; that is each level of the tree is completely filled, except possibly the bottom level. At this level it is filled from left to right.
  2. It satisfies the heap order property; the data item stored in each node is greater than or equal to the data item stored in its children node.

Example:

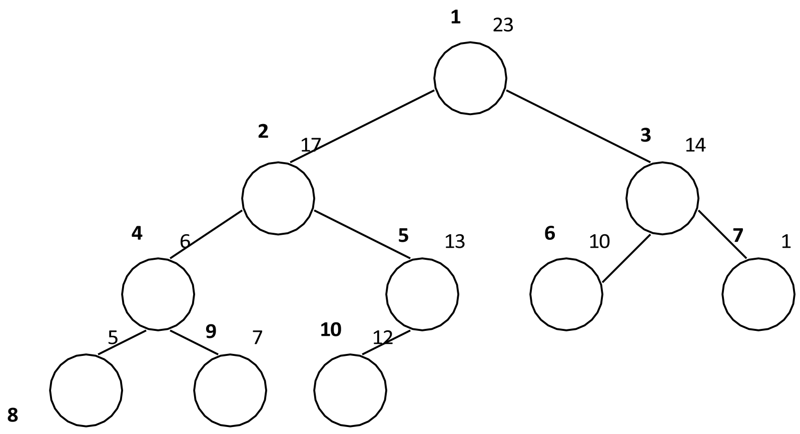


- Heap can be implemented using an array or a linked list structure. It is easier to implement heaps using arrays.
- We simply number the nodes in the heap from top to bottom, numbering the nodes on each level from left to right and store the i^{th} node in the i^{th} location of the array.

- An array *A* that represents a heap is an object with two attributes:
  - i. *length*[*A*], which is the number of elements in the array, and
  - ii. *heap-size*[*A*], the number of elements in the heap stored within array *A*.
- The root of the tree is *A*[1], and given the index *i* of a node, the indices of its parent PARENT(*i*), left child LEFT(*i*), and right child RIGHT(*i*) can be computed simply:

```
PARENT(i)
    return ⌊i/2⌋

LEFT(i)
    return 2i
RIGHT(i)

    return 2i + 1
```

Example:



- The array form for the above heap is,

23	17	14	6	13	10	1	5	7	12

- There are two kinds of binary heaps: max-heaps and min-heaps. In both kinds, the values in the nodes satisfy a ***heap property***.
- In a ***max-heap***, the ***max-heap property*** is that for every node *i* other than the root, ***A*****[PARENT(*i*)] ≥ *A*[*i*] ,**
- That is, the value of a node is at most the value of its parent. Thus, the largest element in a max-heap is stored at the root, and the sub-tree rooted at a node contains values no larger than that contained at the node itself.
- A ***min-heap*** is organized in the opposite way; the ***min-heap property*** is that for every node *i* other than the root, ***A*****[PARENT(*i*)] ≤ *A*[*i*] .**
- The smallest element in a min-heap is at the root.
- For the heap-sort algorithm, we use max-heaps. Min-heaps are commonly used in priority Queues.
- Viewing a heap as a tree, we define the ***height*** of a node in a heap to be the number of edges on the longest simple downward path from the node to a leaf, and we define the height of the Heap to be the height of its root.
- Height of an n element heap based on a binary tree is lg *n*
- The basic operations on heap run in time at most proportional to the height of the tree and thus take *O*(lg *n*) time.
- Since a heap of *n* elements is a complete binary tree which has height k; that is one node on level k, two nodes on level k-1 and so on…
- There will be 2^{k-1} nodes on level 1 and at least 1 and not more than 2^k nodes on level 0.

Building a heap

- For the general case of converting a complete binary tree to a heap, we begin at the last node that is not a leaf; apply the "percolate down" routine to convert the subtree rooted at this current root node to a heap.
- We then move onto the preceding node and percolate down that subtree.
- We continue on in this manner, working up the tree until we reach the root of the given tree.
- We can use the procedure MAX-HEAPIFY in a bottom-up manner to convert an array *A**1,…,*n*], where *n* = *length*[*A*], into a max-heap.

- The elements in the sub-array *A*[(⌊*n*/2⌋+1),…, *n*] are all leaves of the tree, and so each is a 1-element heap to begin with.
- The procedure BUILD-MAX-HEAP goes through the remaining nodes of the tree and runs MAXHEAPIFY on each one.

Algorithm

```
BUILD-MAX-HEAP(A)
heap-size[A] ← length[A]
for i ← ⌊length[A]/2⌋ downto 1
            do MAX-HEAPIFY(A, i)
```

Analysis

- Each call to Heapify costs O(lg *n*) time, and there are O(*n*) such calls. Thus, the running time is at most O(*n* lg n)

Maintaining the heap property

- One of the most basic heap operations is converting a complete binary tree to a heap. Such an operation is called Heapify.
- Its inputs are an array *A* and an index *i* into the array. When MAX-HEAPIFY is called, it is assumed that the binary trees rooted at LEFT(*i*) and RIGHT(*i*) are max-heaps, but that *A*[*i*] may be smaller than its children, thus violating the max-heap property.

- The function of MAX-HEAPIFY is to let the value at *A*[*i*] "float down" in the max-heap so that the sub-tree rooted at index *i* becomes a max-heap.

Algorithm

```
MAX-HEAPIFY(A, i)
l ← LEFT(i)
r ← RIGHT(i)
if l ≤ heap-size[A] and A[l] > A[i]
    then largest ← l
else largest ← i
if r ≤ heap-size[A] and A[r] > A[largest]
    then largest ← r
if largest ≠ i
    then exchange A[i+ ↔ A[largest]
MAX-HEAPIFY(A, largest)
```

- At each step, the largest of the elements *A*[*i*], *A*[LEFT(*i*)], and *A*[RIGHT(*i*)] is determined, and its index is stored in *largest*.
- If *A*[*i*] is largest, then the sub-tree rooted at node *i* is a max-heap and the procedure terminates.
- Otherwise, one of the two children has the largest element, and *A*[*i*] is swapped with *A*[*largest*], which causes node *i* and its children to satisfy the max-heap property.
- The node indexed by *largest*, however, now has the original value *A*[*i*], and thus the sub-tree rooted at *largest* may violate the max-heap property. therefore, MAX-HEAPIFY must be called recursively on that sub-tree.

Analysis

- The running time of MAX-HEAPIFY on a sub-tree of size *n* rooted at given node *i* is the Θ(1) time to fix up the relationships among the elements *A*[*i*], *A*[LEFT(*i*)], and *A*[RIGHT(*i*)], plus the time to run MAX-HEAPIFY on a sub-tree rooted at one of the children of node *i*.
- The children's sub-trees can have size of at most 2*n*/3 and the running time of MAX-HEAPIFY can therefore be described by the recurrence

$$\boldsymbol{T(n) \leq T(2n/3) + \Theta(1).}$$

- The solution to this recurrence is $T(n) = O(\lg n)$.

Heap sort

- The heap sort algorithm starts by using BUILD-MAX-HEAP to build a max-heap on the input array *A**1,…, *n*], where *n* = *length*[*A*].
- Since the maximum element of the array is stored at the root *A*[1], it can be put into its correct final position by exchanging it with *A*[*n*].
- If we now discard node *n* from the heap (by decrementing *heap-size*[*A*]), we observe that *A**1,..,(*n*

- 1)] can easily be made into a max-heap.

- The children of the root remain max-heaps, but the new root element may violate the max-heap property.

- All that is needed to restore the max heap property, however, is one call to MAX-HEAPIFY(*A*, 1), which leaves a max-heap in *A**1,..., (*n* - 1)].
- The heap sort algorithm then repeats this process for the max-heap of size *n* – 1 down to a heap of size 2.

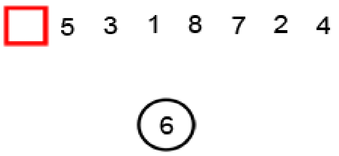
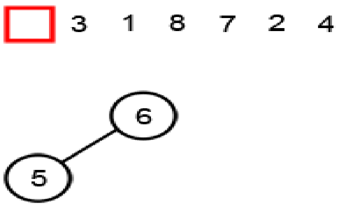
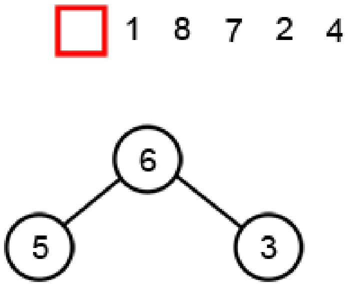
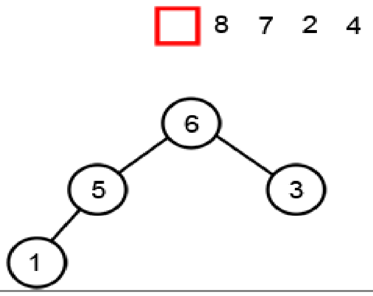
Algorithm

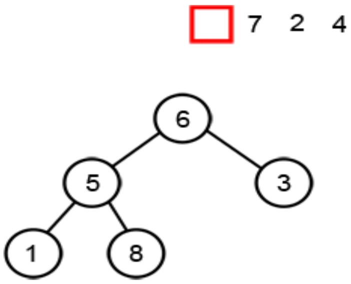
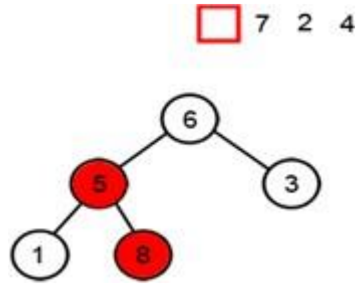
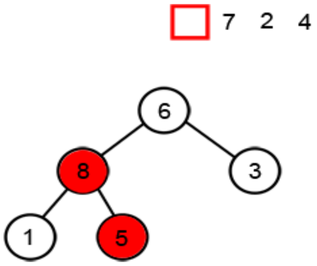
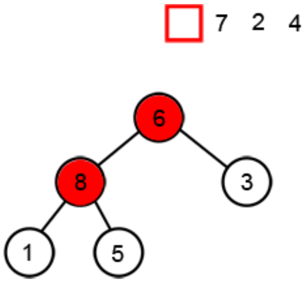
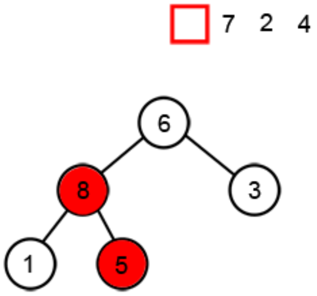
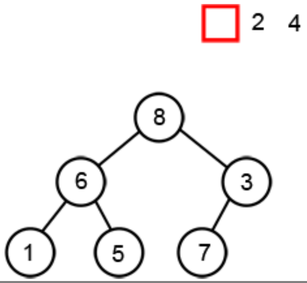
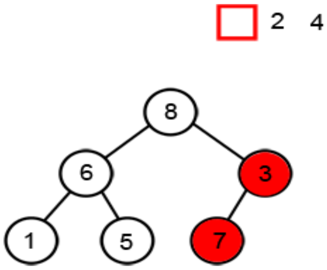
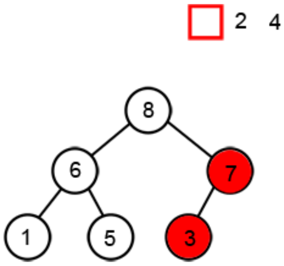
```
HEAPSORT(A)
    BUILD-MAX-HEAP(A)
    for i ← length[A] downto 2
                do exchange A[1] ↔ A[i]
                heap-size[A] ← heap-size[A] - 1
                MAX-HEAPIFY(A, 1)
```

- The HEAPSORT procedure takes time *O*(*n* lg *n*), since the call to BUILD-MAX-HEAP takes time *O*(*n*) and each of the *n* - 1 calls to MAX-HEAPIFY takes time *O*(lg *n*).

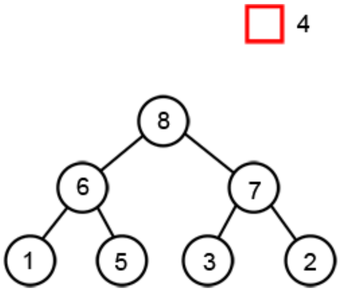
Example

Here is the array: 6, 5, 3, 1, 8, 7, 2, 4

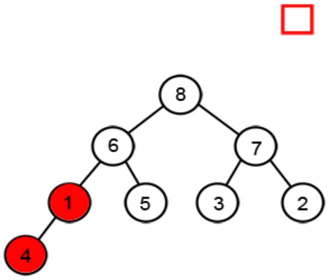
Creating Max Heap. Inserting Node 6 in left to right top to bottom manner 	Inserting Node 5 in left to right top to bottom manner 
Inserting Node 3 in left to right top to bottom manner 	Inserting Node 7 in left to right top to bottom manner 

Inserting Node 8 in left to right top to bottom manner  	Heapify node 8 
Heapify node 8 and swap with node 5  	Heapify node 8 
Heapify node 8 and swap with node 6  	Inserting Node 7 in left to right top to bottom manner 
Heapify node 7  	Heapify node 7 and swap with node 3. 

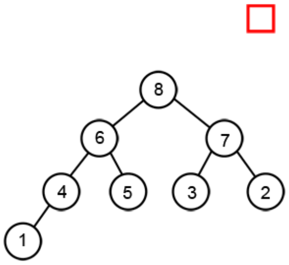
Inserting Node 2 in left to right top to bottom manner



Inserting Node 4 in left to right top to bottom manner

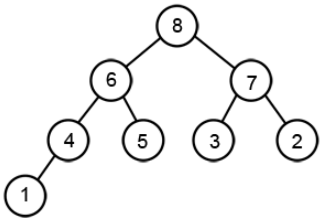


Heapify node 4



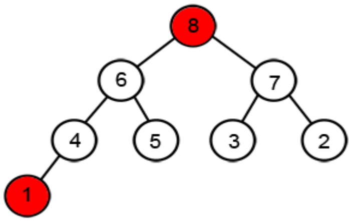
Array representation of MAX heap

8	6	7	4	5	3	2	1



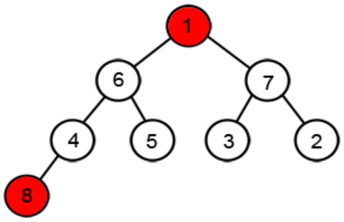
Select first node and swap with last node

8	6	7	4	5	3	2	1



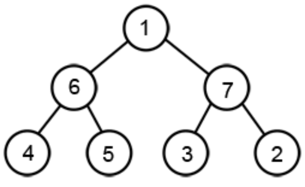
Select first node 8 and swap with last node 1

1	6	7	4	5	3	2	8



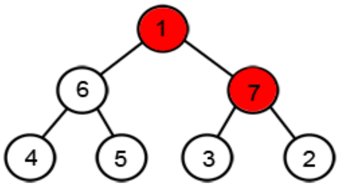
Decrease size of heap by one

1	6	7	4	5	3	2	8

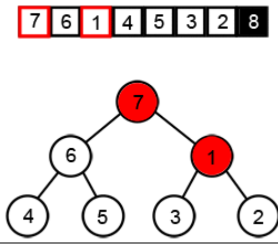


Heapify new array

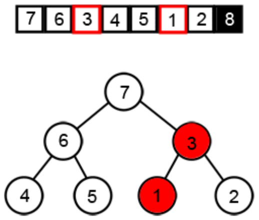
1	6	7	4	5	3	2	8



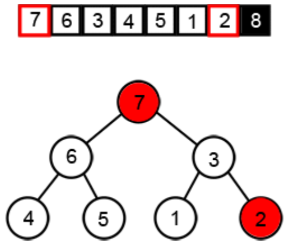
Heapify new array swap node 7 with node 1.



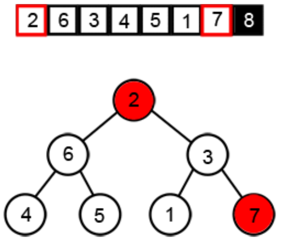
Heapify new array swap node 7 with node 1.



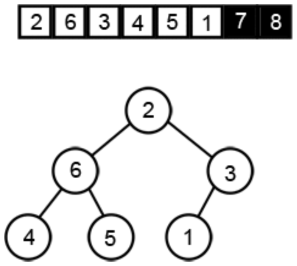
Select first node and swap with last node



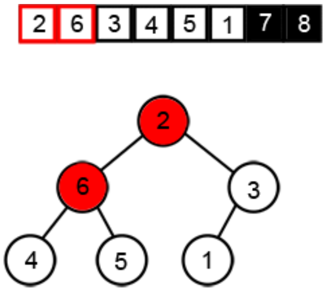
Select first node and swap with last node



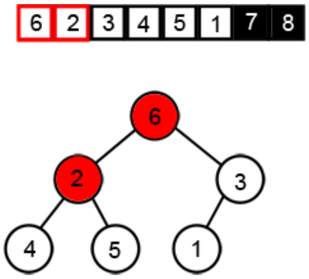
Decrease size of heap by one



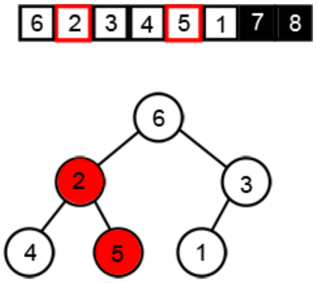
Heapify node 2 and node 6



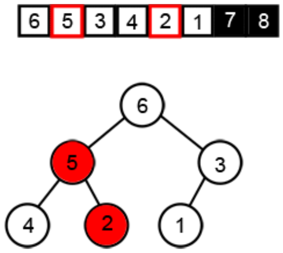
Swap node 6 and node 2



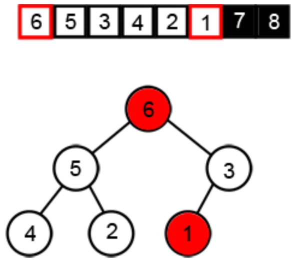
Heapify node 5 and node 2

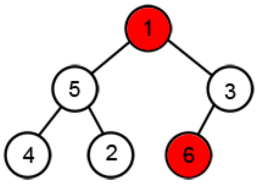
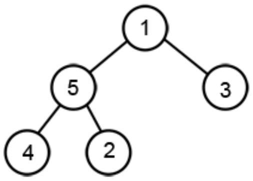
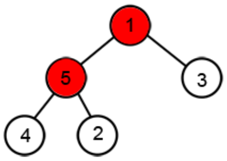
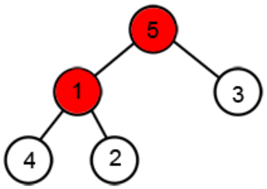
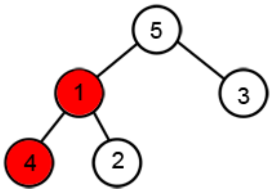
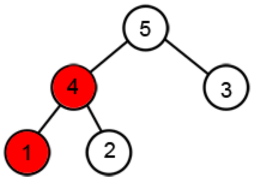
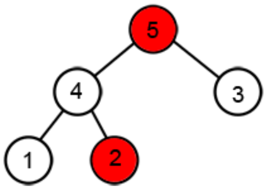
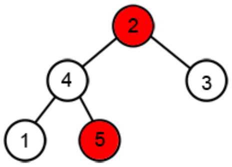
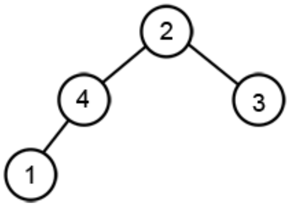
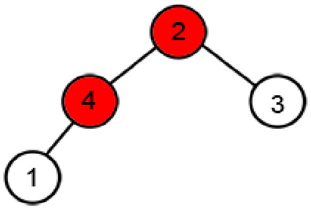


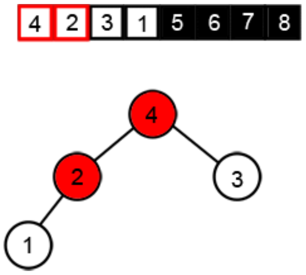
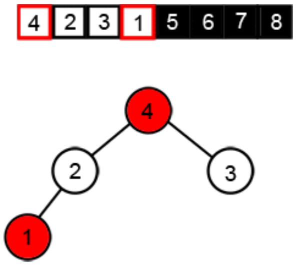
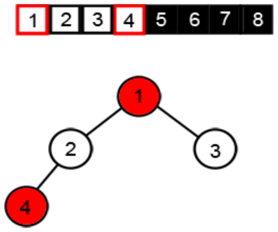
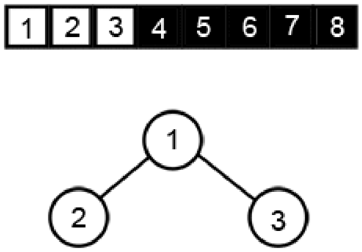
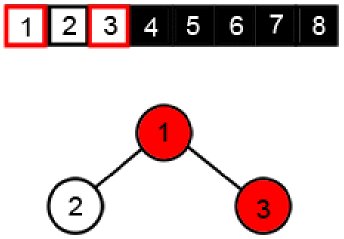
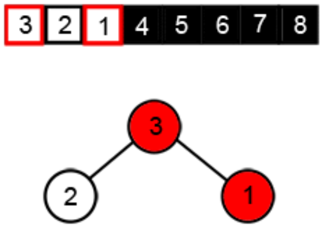
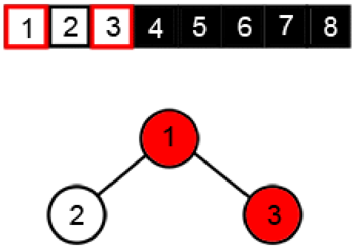
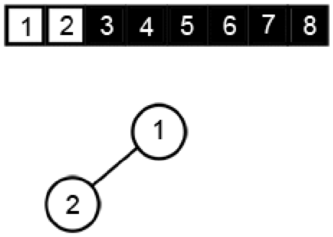
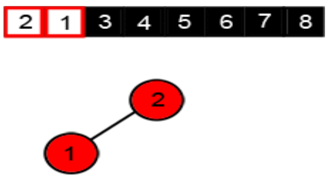
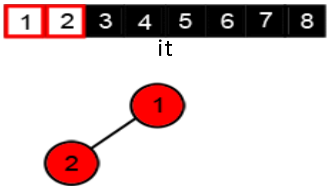
Swap node 5 with node 2

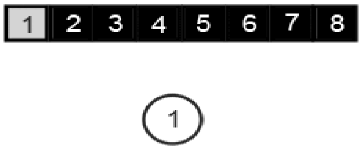


Select first node 6 and last node 1



Swap first node 6 with last node 1 1 5 3 4 2 6 7 8  	Decrease size of heap by one 1 5 3 4 2 6 7 8 
Heapify node 5 and node 1 1 5 3 4 2 6 7 8  	Swap node 5 with node 1 5 1 3 4 2 6 7 8 
Heapify node 1 with node 4 5 1 3 4 2 6 7 8  	Swap node 4 with node 1 5 4 3 1 2 6 7 8 
Select first node 5 and last node 2 5 4 3 1 2 6 7 8  	Swap first node 5 and last node 2 2 4 3 1 5 6 7 8 
Decrease size of heap by one 2 4 3 1 5 6 7 8 heap  	Heapify node 4 and node 2 2 4 3 1 5 6 7 8 

Swap node 4 with node 2  	Select first node 4 and last node 1 
Swap first node 4 with last node 1  	Decrease size of heap by one 
Heapify node 1 and node 3  	Swap node 3 with node 1 
Select first node 3 and last node 1 and swap it  	Decrease size of heap by one 
Heapify node 2 and node 1  	Select first element and last and swap it 

Decrease size of heap by one	Sorted array
	

(7) Shell sort

- Shell sort works by comparing elements that are distant rather than adjacent elements in an array.
- Shell sort uses a sequence h1, h2… ht called the increment sequence. Any increment sequence is fine as long as h1 = 1 and some other choices are better than others.
- Shell sort makes multiple passes through a list and sorts a number of equally sized sets using the insertion sort.

Example:1 54,26,93,17,77,31,44,55,20

54	26	93
17	77	31
44	55	20

→ Row by Row

26	54	93
17	31	77
20	44	55

→ Colum$n$ by Colum$n$

17	31	55
20	44	77
26	54	93

Sorted array: 17,20,26,31,44,54,55,77,93

Example:2 3 7 9 0 5 1 6 8 4 2 0 6 1 5 7 3 4 9 8 2

It is arranged in an array with 7 columns (left), then the columns are sorted (right): Step-1

3 7 9 0 5 1 6		3 3 2 0 5 1 5
8 4 2 0 6 1 5	→	7 4 4 0 6 1 6
7 3 4 9 8 2		8 7 9 9 8 2

Step-2

3 3 2		0 0 1
0 5 1		1 2 2
5 7 4		3 3 4
4 0 6	→	4 5 6
1 6 8		5 6 8
7 9 9		7 7 9
8 2		8 9

(8) Bucket sort

- Bucket sort runs in linear time when the input is drawn from a uniform distribution.
- Like counting sort, bucket sort is fast because it assumes something about the input.
- Whereas counting sort assumes that the input consists of integers in a small range, bucket sort assumes that the input is generated by a random process that distributes elements uniformly over the interval [0, 1)

Assumption: the keys are in the range [0, N) Basic idea:

1. Create N linked lists (buckets) to divide interval [0,N) into subintervals of size 1
2. Add each input element to appropriate bucket
3. Concatenate the buckets

Expected total time is O(n + N), with n = size of original sequence if N is O(n) -> sorting algorithm in O(n) !

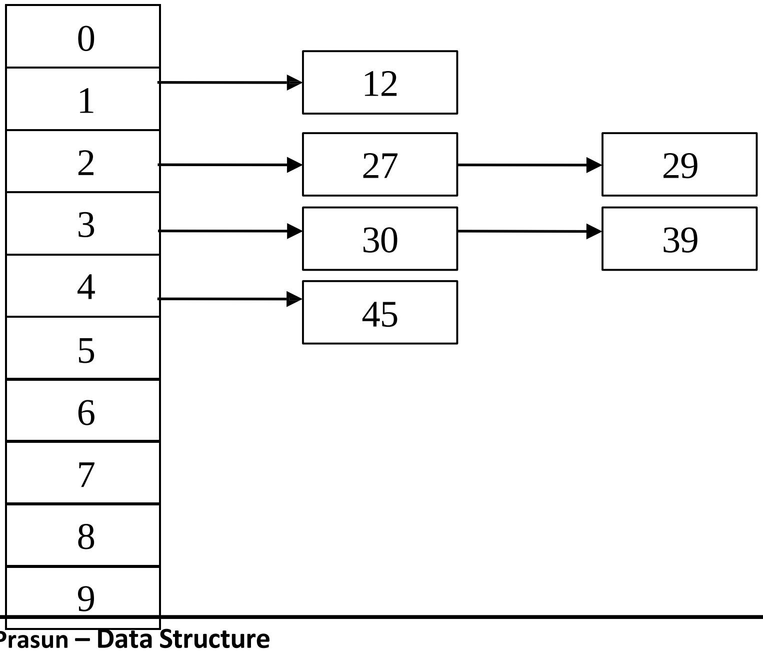
Algorithm

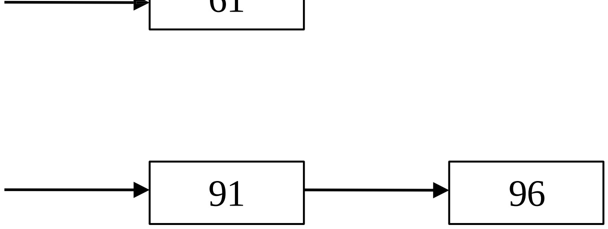
```
BUCKET-SORT(A)

n ← length[A]
for i ← 1 to n
   do insert A[i] into list B[$\lfloor n\ A[i] \rfloor$]
for i ← 0 to n - 1
   do sort list B[i] with insertion sort
concatenate the lists B[0], B[1], . . ., B[n - 1]  together in
 order
```

***Example*:** 45,96,29,30,27,12,39,61,91

Step-1 Add keys one by one in appropriate bucket queue as shown in figure Step-2 sort each bucket queue with insertion sort

Step-3 Merge all bucket queues together in order





After sorting : 12,27,29, 30, 39, 45, 61, 91, 96

(9) Radix sort

- The operation of radix sort on a list of seven 3-digit numbers. The leftmost column is the input. The remaining columns show the list after successive sorts on increasingly significant digit positions.
- Shading indicates the digit position sorted on to produce each list from the previous one.
- The code for radix sort is straightforward. The following procedure assumes that each element in the n-element array A has d digits, where digit 1 is the lowest-order digit and digit d is the highest order digit.
- Given n d-digit numbers in which each digit can take on up to k possible values, RADIXSORT correctly sorts these numbers in Θ(d(n+ k)) time.

Algorithm

```
RADIX-SORT(A, d)
  for i ← 1 to d
      do use a stable sort to sort array A on digit i
```

***Example*:** 363, 729, 329, 873, 691, 521, 435, 297

2	9	7
3	2	9
3	6	3
4	3	5
5	2	1
6	9	1
7	2	9
8	7	3

(10) Counting sort

- Counting sort assumes that each of the n input elements is an integer in the range 0 to k, for some integer k.
- When k = O(n), the sort runs in Θ(n) time.
- The basic idea of counting sort is to determine, for each input element x, the number of elements less than x.
- This information can be used to place element x directly into its position in the output array.

Algorithm

```
COUNTING-SORT(A, B, k)
  for i ← 0 to k
     do C[i] ← 0
  for j ← 1 to length[A]
     do C[A[j]] ← C[A[j]] + 1
 //C[i] now contains the number of elements equal to i.
  for i ← 1 to k
     do C[i] ← C[i] + C[i - 1]
 //C[i] now contains the number of elements less than or equal
   to i.
  for j ← length[A] down to  1
     do B[C[A[j]]] ← A[j]
        C[A[j]] ← C[A[j]] -  1
```

Example

Step-1 Given input array A*1…8+

1	2	3	4	5	6	7	8
3	6	4	1	3	4	1	4

Step-2 Determine size of array C as maximum value in array A, here its ‘6’

Initialize all elements intermediate array C*1…6+ = 0

1	2	3	4	5	6
0	0	0	0	0	0

Step-3 Update array C with occurrences of each value of array A

1	2	3	4	5	6
2	0	2	3	0	1

Step-4 In array C from index 2 to n add value with previous element

1	2	3	4	5	6
2	2	4	7	7	8

Create output array B*1…8+

Start positioning elements of Array A to B shown as follows

Step-5 Array A

1	2	3	4	5	6	7	8
3	6	4	1	3	4	1	4

Array C

1	2	3	4	5	6
2	2	4	7	7	8

Array B

1	2	3	4	5	6	7	8
						4	

Repeat above from element n to 1 we can have Array B

1	2	3	4	5	6	7	8
1	1	3	3	4	4	4	0

(1) Divide & Conquer Technique and the general template for it.

Divide and conquer technique

- Many useful algorithms are ***recursive*** in structure: to solve a given problem, they call themselves recursively one or more times to deal with closely related sub problems.
- These algorithms typically follow a ***divide-and-conquer*** approach:
- The divide-and-conquer approach involves three steps at each level of the recursion:
- **Divide:** Break the problem into several sub problems that are similar to the original problem but smaller in size,
- **Conquer:** Solve the sub problems recursively, and If the sub problem sizes are small enough, just solve the sub problems in a straightforward manner.
- **Combine:** Combine these solutions to create a solution to the original problem.

The general template

- Consider an arbitrary problem, and let adhoc be a simple algorithm capable of solving the problem.
- We call adhoc as basic sub-algorithm such that it can be efficient in solving small instances, but its performance on large instances is of no concern.
- The general template for divide-and-conquer algorithms is as follows.

```
        function DC(x)
if x is sufficiently small or simple then return adhoc(x) decompose x into
smaller instances $x_1$ , $x_2$, …, $x_l$
                for $i = 1$ to $l$ do $y_i \leftarrow DC(x_i)$
recombine the $Y_L$ 's to obtain a solution y for x
                return y
```

- The running-time analysis of such divide-and-conquer algorithms is almost automatic.
- Let *g(n)* be the time required by *DC* on instances of size *n,* not counting the time needed for the recursive calls.
- The total time t*(n)* taken by this divide-and-conquer algorithm is given by Recurrence equation,

  Provided n is large enough.

- The solution of equation is given as,

  then

- The recurrence equation and its solution are applicable to find the time complexity of every problem which can be solved using Divide & Conquer Technique.

(2) Linear Search

Sequential (Linear) search algorithm

```
Function sequential ( T[1,…,n], x )
        for i = 1 to n do
            if T [i] ≥ x then return index i
        return n + 1
```

Analysis

- Here we look sequentially at each element of T until either we reach to end of the array or find a number no smaller than x.
- This algorithm clearly takes time in Θ (r), where r is the index returned. Θ (n) in worst case and O (1) in best case.

(3) Divide and Conquer Technique and the use of it for Binary Searching Method.

Binary Search Method

- Binary Search is an extremely well-known instance of divide-and-conquer approach.
- Let T[*1 . . . n*] be an array of increasing sorted order; that is $T[i] \leq T[j]$ whenever $1 \leq i \leq j \leq n$.
- Let x be some number. The problem consists of finding x in the array T if it is there.
- If x is not in the array, then we want to find the position where it might be inserted.

Binary Search Algorithm (Iterative)

- The basic idea of binary search is that for a given element we check out the middle element of the array.
- We continue in either the lower or upper segment of the array, depending on the outcome of the search until we reach the required (given) element.
- Here the technique Divide & Conquer applies. Total number of elements to be searched is divided in half size every time.

```
Function biniter ( T[1,…,n], x )
        i ← 1; j ← n
        while i < j do
                k ← (i + j ) ÷ 2
                if  x ≤ T [k]  then  j ← k
                        else i ← k + 1
```

return *i*

Analysis

- To analyze the running time of a while loop, we must find a function of the variables involved whose value decreases each time round the loop.
- Here it is ***j – i + 1***
- Which we call as d. d represents the number of elements of T still under consideration.
- Initially $d = n$.
- Loop terminates when $i \geq j$, which is equivalent to $d \leq 1$
- Each time round the loop there are three possibilities,
  - I. Either j is set to $k - 1$
  - II. Or i is set to $k + 1$
  - III. Or both i and j are set to k
- Let d and d' stand for the value of $j - i + 1$ before and after the iteration under consideration. Similarly i, j , i' and j'.
- <u>Case I</u> : if $x < T[k]$

  So, $j \leftarrow k - 1$ is executed.

  Thus $i' = i$ and $j' = k - 1$ where $k = (i + j) \div 2$

  Substituting the value of k, $j' = [(i + j) \div 2] - 1$ $d' =$
  $j' - i' + 1$

  Substituting the value of j', $d' = [(i + j) \div 2] - 1 - i + 1$

  $d' \leq (i + j)/2 - i$

  $d' \leq (j - i)/2 \leq (j - i + 1)/2$

  $d' \leq d/2$
- <u>Case II</u> : if $x > T[k]$

  So, $i \leftarrow k + 1$ is executed

  Thus $i' = K + 1$ and $j' = j$ where $k = (i + j) \div 2$

  Substituting the value of k, $i' = [(i + j) \div 2] + 1$ $d' =$
  $j' - i' + 1$

  Substituting the value of I', $d' = j - [(i + j) \div 2] + 1 + 1$ $d' \leq j$
  $- (i + j - 1)/2 \leq (2j - i - j + 1)/2 \leq (j - i + 1)/2$ **$d' \leq d/2$**
- <u>Case III</u> : if $x = T[k]$

  $i = j \rightarrow d' = 1$
- We conclude that whatever may be case, **$d' \leq d/2$** which means that the value of d is at least getting half each time round the loop.
- Let d_k denote the value of $j - i + 1$ at the end of kth trip around the loop. $d_0 = n$.
- We have already proved that $d_k = d_{k-1} / 2$
- For n integers, how many times does it need to cut in half before it reaches or goes below 1?
- $n / 2^k \leq 1 \rightarrow n \leq 2^k$
- ***k = lgn*** , search takes time.

The complexity of biniter is Θ (lg n).

Binary Search Algorithm (Recursive)

```
Function binsearch ( T[1,…,n], x )
        if n = 0 or x > T[n] then return n + 1
        else return binrec ( T[1,…,n], x )
Function binrec( T[i,…,j], x )
        if i = j then return i
        k ← (i + j) ÷ 2
        if x ≤ T [k] then return binrec( T[i,…,k], x )
                else return binrec( T[k + 1,…,j], x )
```

Analysis

- Let t(n) be the time required for a call on binrec(T[i,…,j], x), where n = j – i + 1 is the number of elements still under consideration in the search.
- The recurrence equation is given as,
  $t(n) = t(n/2) + \Theta(1)$
  Comparing this to the general template for divide and conquer algorithm, l = 1, b = 2 and k = 0.
  So, $t(n) \in \Theta(\lg n)$
- **The complexity of binrec is Θ (lg n).**

(4) Merge Sort and Analysis of Merge Sort

- The divide & conquer approach to sort n numbers using merge sort consists of separating the array T into two parts where sizes are almost same.
- These two parts are sorted by recursive calls and then merged the solution of each part while preserving the order.
- The algorithm considers two temporary arrays U and V into which the original array T is divided.
- When the number of elements to be sorted is small, a relatively simple algorithm is used.
- Merge sort procedure separates the instance into two half sized sub instances, solves them recursively and then combines the two sorted half arrays to obtain the solution to the original instance.

Algorithm for merging two sorted U and V arrays into array T

```
Procedure merge(U[1,…,m+1],V[1,…,n+1],T[1,…,m+n])
        i, j ← 1
        U[m+1], V[n+1] ← ∞
```

```
for k ← 1 to m + n do
    if U[i] < V[j]
        then T[k] ← U[i] ; i ← i + 1
        else T[k] ← V[j] ; j ← j + 1
```

Algorithm merge sort

```
Procedure mergesort(T[1,…,n])
    if n is sufficiently small then insert(T)
    else
        array U[1,…,1+n/2],V[1,…,1+n/2]
        U[1,…,n/2] ← T[1,…,n/2]
        V[1,…,n/2] ← T[n/2+1,…,n]
        mergesort(U[1,…,n/2])
        mergesort(V[1,…,n/2])
        merge(U, V, T)
```

Analysis

- Let T(n) be the time taken by this algorithm to sort an array of n elements.
- Separating T into U & V takes linear time; merge (U, V, T) also takes linear time.
- Now,

  $T(n)=T(n/2)+ T(n/2)+g(n)$ where $g(n) \in \Theta(n)$.

  $T(n) = 2t(n/2)+ \Theta (n)$

  Applying the general case, $l=2$, $b=2$, $k=1$

  Since $l = b^k$ the second case applies which yields $t(n) \in \Theta(nlogn)$.

Time complexity of merge sort is **Θ (nlogn).**

Example

(5) Quick sort method / algorithm and its complexity.

- Quick sort works by partitioning the array to be sorted.
- Each Partition is internally sorted recursively.
- As a first step, this algorithm chooses one element of an array as a pivot or a key element.
- The array is then partitioned on either side of the pivot.
- Elements are moved so that those greater than the pivot are shifted to its right whereas the others are shifted to its left.
- Two pointers low and up are initialized to the lower and upper bounds of the sub array.
- Up pointer will be decremented and low pointer will be incremented as per following condition.
    1. Increase low pointer until T[low] > pivot.
    2. Decrease up pointer until T[up] ≤ pivot.
    3. If low < up then interchange T[low] with T[up].
    4. If up ≤ low then interchange T[up] with T[i].

Algorithm

```
Procedure pivot(T[i,…,j]; var l)

{Permutes the elements in array T[i,…,j] and returns a value l such that, at the end, i<=l<=j,
T[k]<=p for all i ≤ k < l, T[l]=p, And T[k] > p for all l < k ≤ j, where p is the initial value T[i]}

P ← T[i]

K ← i; l ← j+1

Repeat k ← k+1 until T[k] > p
Repeat l ← l-1 until T[l] ≤ p
While k < l do

    Swap T[k] and T[l]

    Repeat k ← k+1 until T[k] > p

    Repeat l ← l-1 until T[l] ≤ p
Swap T[i] and T[l]


Procedure quicksort(T[i,…,j])

{Sorts subarray T[i,…,j] into non decreasing order}
if  j – i is sufficiently small then insert (T[i,…,j])
else

 pivot(T[i,…,j],l)

quicksort(T[i,…, l - 1])
quicksort(T[l+1,…,j]
```

Analysis

1. ***Worst Case***
   - Running time of quick sort depends on whether the partitioning is balanced or unbalanced.
   - And this in turn depends on which element is chosen as key or pivot element.
   - The worst case behavior for quick sort occurs when the partitioning routine produces one sub problem with n-1 elements and one with 0 elements.

   - In this case recurrence will be,

     $T(n)=T(n-1)+T(0)+\Theta(n)$

     $T(n)=T(n-1)+ \Theta\ (n)$

     $\boldsymbol{T(n)= \Theta\ (n^2)}$
2. **Best Case**
   - Occurs when partition produces sub problems each of size n/2.
   - Recurrence equation:

     $T(n)=2T(n/2)+ \Theta(n)$

     $l = 2,\ b = 2,\ k = 1,$ so $l = b^k$

     $\boldsymbol{T(n)= \Theta(nlogn)}$
3. **Average Case**
   - Average case running time is much closer to the best case.
   - If suppose the partitioning algorithm produces a 9-to-1 proportional split the recurrence will be

     $T(n)=T(9n/10)+ T(n/10)+ \Theta(n)$

     Solving it,

     T(n)= Θ(nlogn)

   - **The running time of quick sort is therefore Θ(nlogn) whenever the split has constant proportionality.**

Example

[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]
40	20	10	80	60	50	7	30	100

```
 Pivot,i                                     j

Here, Pivot = T[i], k = i, l= j + 1

       Repeat k = k + 1, Until T[k] > Pivot


       Repeat l = l – 1, Until T[l] <= Pivot


  [1]   [2]  [3]  [4]  [5]  [6]  [7]  [8]  [9]
```

40	20	10	80	60	50	7	30	100
Pivot			k				l	

Is K < l? , Yes then Swap T[k] <-> T[l]

[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]
40	20	10	30	60	50	7	80	100
Pivot			k				l	

```
    Repeat k = k + 1, Until T[k] > Pivot
    Repeat l = l – 1, Until T[l] <= Pivot
```

[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]
40	20	10	30	60	50	7	80	100
Pivot				k		l		

Is K < l? , Yes then Swap T[k] <-> T[l]

[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]
40	20	10	30	7	50	60	80	100
Pivot				k		l		

```
    Repeat k = k + 1, Until T[k] > Pivot
    Repeat l = l – 1, Until T[l] <= Pivot
```

[1] [2] [3] [4] [5] [6] [7] [8] [9]

40	20	10	30	7	50	60	80	100
Pivot				l	k			

Is K < l? , No then Swap Pivot <-> T[l]

[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]
7	20	10	30	40	50	60	80	100
				Pivot				

Now we have two sub list one having elements less or equal Pivot and second having elements greater than Pivot

[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]
Pivot,i			j		Pivot,i			j

Repeat the same procedure for each sub list until list is sorted

Applications of Array

1. Symbol Manipulation (matrix representation of polynomial equation)
2. Sparse Matrix

Symbol Manipulation using Array

- We can use array for different kind of operations in polynomial equation such as addition, subtraction, division, differentiation etc…
- We are interested in finding suitable representation for polynomial so that different operations like addition, subtraction etc… can be performed in efficient manner
- Array can be used to represent Polynomial equation

- **Matrix Representation of Polynomial equation**

	Y	**Y^2**	**Y^3**	**Y^4**
X	X Y	X Y^2	X Y^3	X Y^4
X^2	X^2 Y	X^2 Y^2	X^2 Y^3	X^2 Y^4
X^3	X^3 Y	X^3 Y^2	X^3 Y^3	X^3 Y^4
X^4	X^4 Y	X^4 Y^2	X^4 Y^3	X^4 Y^4

e.g. **$2x^2+5xy+Y^2$**

is represented in matrix form as below

		Y	**Y^2**	**Y^3**	**Y^4**
	0	0	**1**	0	0
X	0	**5**	0	0	0
X^2	**2**	0	0	0	0
X^3	0	0	0	0	0
X^4	0	0	0	0	0

e.g**. $x^2+3xy+Y^2+Y-X$**

is represented in matrix form as below

		Y	**Y^2**	**Y^3**	**Y^4**
	0	0	**1**	0	0
X	**-1**	**3**	0	0	0
X^2	**1**	0	0	0	0
X^3	0	0	0	0	0
X^4	0	0	0	0	0

- Once we have algorithm for converting the polynomial equation to an array representation and another algorithm for converting array to polynomial equation, then different operations in array (matrix) will be corresponding operations of polynomial equation

What is sparse matrix? Explain

- An mXn matrix is said to be sparse if “many” of its elements are zero.
- A matrix that is not sparse is called a dense matrix.
- We can device a simple representation scheme whose space requirement equals the size of the non-zero elements.

- **Example:-**
  - The non-zero entries of a sparse matrix may be mapped into a linear list in row-major order.
  - For example the non-zero entries of 4X8 matrix of below fig.(a) in row major order are 2, 1, 6, 7, 3, 9, 8, 4, 5

$$\begin{vmatrix} 0 & 0 & 0 & 2 & 0 & 0 & 1 & 0 \\ 0 & 6 & 0 & 0 & 7 & 0 & 0 & 3 \\ 0 & 0 & 0 & 9 & 0 & 8 & 0 & 0 \\ 0 & 4 & 5 & 0 & 0 & 0 & 0 & 0 \end{vmatrix}$$

Fig (a) 4 x 8 matrix

Terms	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**
Row	1	1	2	2	2	3	3	4	4
Column	4	7	2	5	8	4	6	2	3
Value	2	1	6	7	3	9	8	4	5

Fig (b) Linear Representation of above matrix

- To construct matrix structure we need to record
  (a) Original row and columns of each non zero entries
  (b) No of rows and columns in the matrix
- So each element of the array into which the sparse matrix is mapped need to have three fields: <u>row, column and value</u>
- A corresponding amount of time is saved creating the linear list representation over initialization of two dimension array.

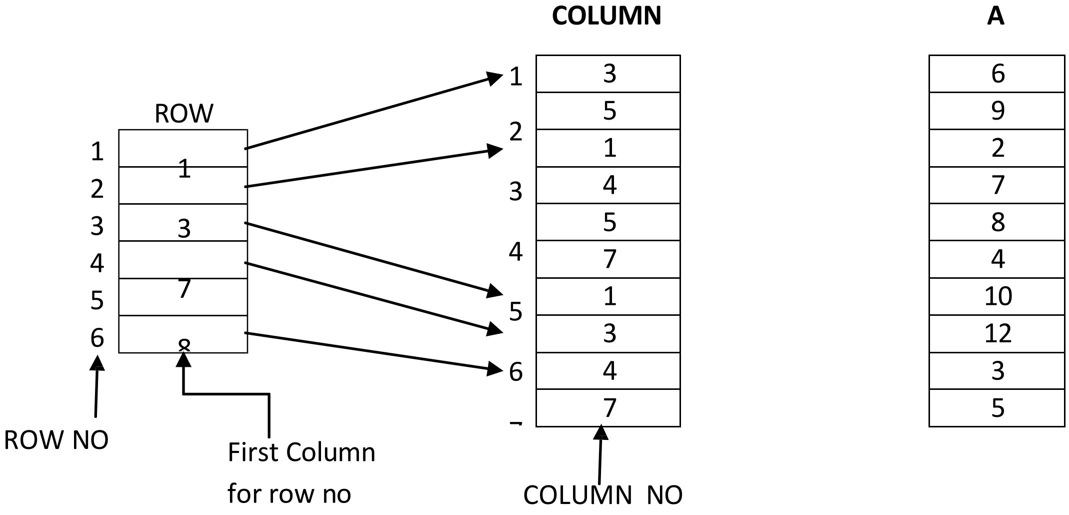
$$A = \begin{vmatrix} 0 & 0 & 6 & 0 & 9 & 0 & 0 \\ 2 & 0 & 0 & 7 & 8 & 0 & 4 \\ 10 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 12 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 3 & 0 & 0 & 5 \end{vmatrix}$$

- Here from 6X7=42 elements, only 10 are non zero. A[1,3]=6, A[1,5]=9, A[2,1]=2, A[2,4]=7, A[2,5]=8, A[2,7]=4, A[3,1]=10, A[4,3]=12, A[6,4]=3, A[6,7]=5.
- One basic method for storing such a sparse matrix is to store non-zero elements in one dimensional array and to identify each array elements with row and column indices fig (c).

	ROW	**COLUMN**	**A**
1	1	3	6
2	1	5	9

3	2	1	2
4	2	4	7
5	2	5	8
6	2	7	4
7	3	1	10
8	4	3	12
9	6	4	3
10	6	7	5

Fig (c)



Fig(d)

- A more efficient representation in terms of storage requirement and access time to the row of the matrix is shown in fid (d). The row vector changed so that its i^{th} element is the index to the first of the column indices for the element in row I of the matrix.

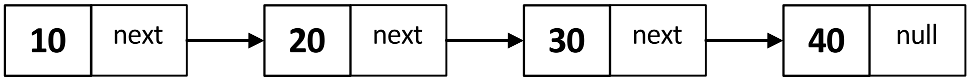
1. Linear Data Structure and their linked storage representation.

There are many applications where sequential allocation method is unacceptable because of following characteristics

- Unpredictable storage requirement
- Extensive manipulation of stored data

The linked allocation method of storage can result in both efficient use of computer storage and computer time.

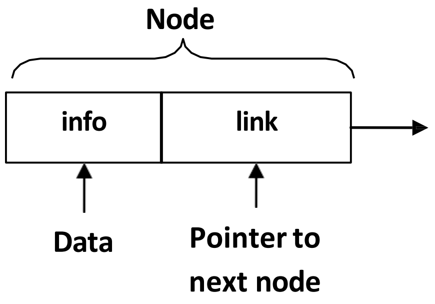
- A linked list is a non-sequential collection of data items.
- The concept of a linked list is very simple, for every data item in the linked list, there is an associated pointer that would give the memory allocation of the next data item in the linked list.
- The data items in the linked list are not in a consecutive memory locations but they may be anywhere in memory.
- Accessing of these data items is easier as each data item contains within itself the address of the next data item.



A Linked List

2. What is linked list? What are different types of linked list? OR Write a short note on singly, circular and doubly linked list. OR Advantages and disadvantages of singly, circular and doubly linked list.

- A linked list is a collection of objects stored in a list form.
- A linked list is a sequence of items (objects) where every item is linked to the next.
- A linked list is a non-primitive type of data structure in which each element is dynamically allocated and in which elements point to each other to define a linear relationship.
- Elements of linked list are called nodes where each node contains two things, data and pointer to next node.
- Linked list require more memory compared to array because along with value it stores pointer to next node.
- Linked lists are among the simplest and most common data structures. They can be used to implement other data structures like stacks, queues, and symbolic expressions, etc…



```
// C Structure to represent a node
struct node
{

        int info
        struct node *link
};
```

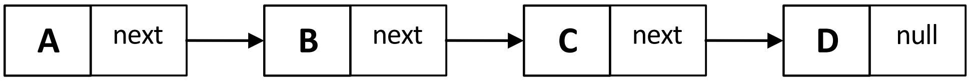
Operations on linked list

- Insert
    - Insert at first position
    - Insert at last position
    - Insert into ordered list
- Delete
- Traverse list (Print list)
- Copy linked list

Types of linked list

Singly Linked List

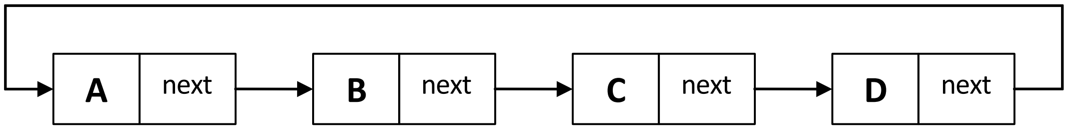
- It is basic type of linked list.
- Each node contains data and pointer to next node.
- Last node's pointer is null.
- Limitation of singly linked list is we can traverse only in one direction, forward direction.



Singly Linked List

Circular Linked List

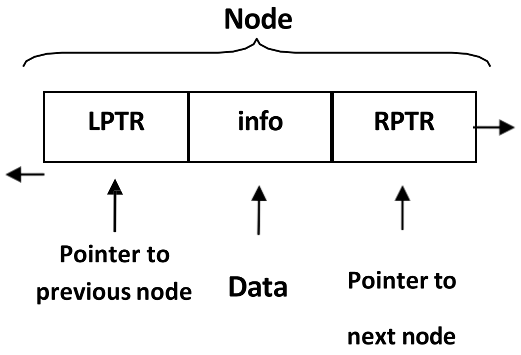
- Circular linked list is a singly linked list where last node points to first node in the list.
- It does not contain null pointers like singly linked list.
- We can traverse only in one direction that is forward direction.
- It has the biggest advantage of time saving when we want to go from last node to first node, it directly points to first node.
- A good example of an application where circular linked list should be used is a timesharing problem solved by the operating system.



Circular Linked List

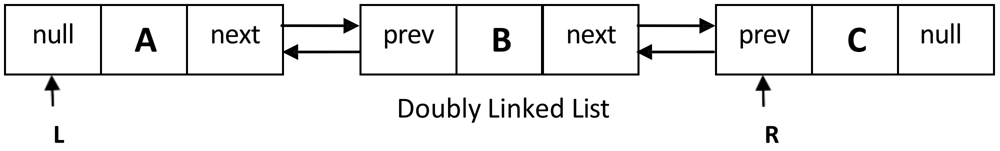
Doubly Linked list

- Each node of doubly linked list contains data and two pointers to point previous (LPTR) and next (RPTR) node.



```
// C Structure to represent a node
struct node
{
        int info
        struct node *lptr;
        struct node *rptr;

};
```

- Main advantage of doubly linked list is we can traverse in any direction, forward or reverse.
- Other advantage of doubly linked list is we can delete a node with little trouble, since we have pointers to the previous and next nodes. A node on a singly linked list cannot be removed unless we have the pointer to its predecessor.
- Drawback of doubly linked list is it requires more memory compared to singly linked list because we need an extra pointer to point previous node.
- L and R in image denote left most and right most nodes in the list.
- Left link of L node and right link of R node is NULL, indicating the end of list for each direction.



Doubly Linked List

3. Discuss advantages and disadvantages of linked list over array.

Advantages of an array

1. We can access any element of an array directly means random access is easy
2. It can be used to create other useful data structures (queues, stacks)

3. It is light on memory usage compared to other structures

Disadvantages of an array

1. Its size is fixed
2. It cannot be dynamically resized in most languages
3. It is hard to add/remove elements
4. Size of all elements must be same.
5. Rigid structure (Rigid = Inflexible or not changeable)

Advantages of Linked List

1. **Linked lists are dynamic data structures:** That is, they can grow or shrink during execution of a program.
2. **Efficient memory utilization:** Here memory is not pre-allocated. Memory is allocated whenever it is required. And it is deallocated (free) when it is no longer needed.
3. **Insertion and deletions are easier and efficient:** Linked list provide flexibility in inserting a data item at a specified position and deletion of a data item from the given position.
4. **Elements of linked list are flexible:** It can be primary data type or user defined data types

Disadvantages of Linked List

1. Random access is not allowed. We have to access elements sequentially starting from the first node. So we cannot do binary search with linked lists.
2. It cannot be easily sorted
3. We must traverse 1/2 the list on average to access any element
4. More complex to create than an array
5. Extra memory space for a pointer is required with each element of the list

---

3. What are the advantages and disadvantages of stack and queue implemented using linked list over array?

Advantages and disadvantages of stack & queue implemented using linked list over array is described below,

Insertion & Deletion Operation

- Insertion and deletion operations are known as push and pop operation in stack and as insert and delete operation in queue.
- In the case of an array, if we have n-elements list and it is required to insert a new element between the first and second element then n-1 elements of the list must be moved so as to make room for the new element.
- In case of linked-list, this can be accomplished by only interchanging pointers.
- Thus, insertion and deletions are more efficient when performed in linked list then array.

Searching a node

- If a particular node in a linked list is required, it is necessary to follow links from the first node onwards until the desired node is found.
- Where as in the case of an array, directly we can access any node

Join & Split

- We can join two linked list by assigning pointer of second linked list in the last node of first linked list.
- Just assign null address in the node from where we want to split one linked list in two parts.
- Joining and splitting of two arrays is much more difficult compared to linked list.

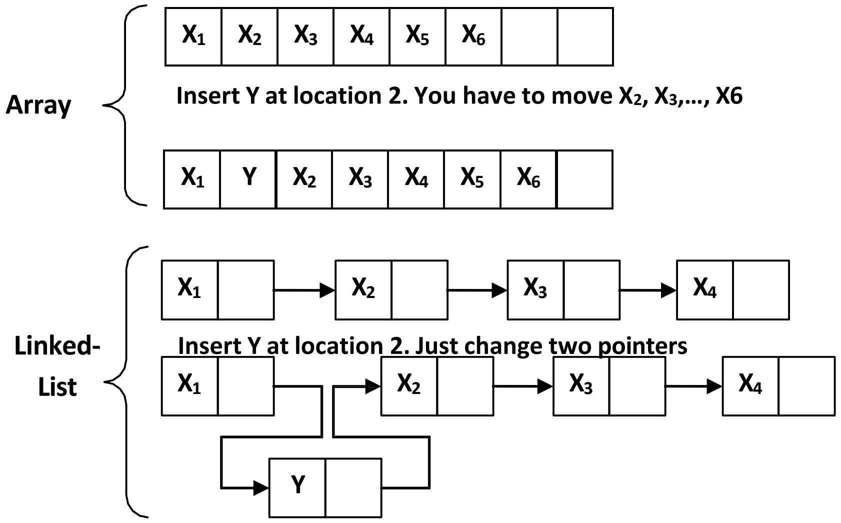
Memory

- The pointers in linked list consume additional memory compared to an array

Size

- Array is fixed sized so number of elements will be limited in stack and queue.
- Size of linked list is dynamic and can be changed easily so it is flexible in number of elements

Insertion and deletion operations in Array and Linked-List

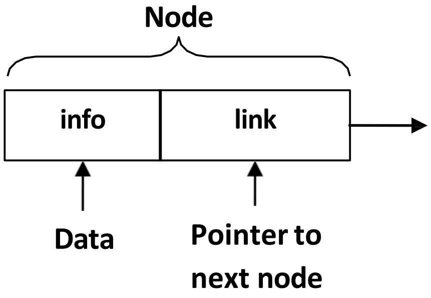

Array
X1 X2 X3 X4 X5 X6
Insert Y at location 2. You have to move X2, X3,…, X6
X1 Y X2 X3 X4 X5 X6
Linked-List
X1 X2 X3 X4
Insert Y at location 2. Just change two pointers
X1 X2 X3 X4
Y

4. Write following algorithms for singly linked list.

1) Insert at first position
2) Insert at last position
3) Insert in Ordered Linked list
4) Delete Element
5) Copy Linked List

Few assumptions,

- We assume that a typical element or node consists of two fields namely; an information field called INFO and pointer field denoted by LINK. The name of a typical element is denoted by NODE.



```
// C Structure to represent a node
struct node
{
      int info
      struct node *link
};
```

Function: INSERT(X, First)

Given X, a new element and FIRST is a pointer to the first element of a linked linear list. Typical node contains INFO and LINK fields. AVAIL is a pointer to the top element of the availability stack; NEW is a temporary pointer variable. This function inserts a new node at the first position of linked list. This function returns address of FIRST node.

```
1   [Underflow?]
    IF      AVAIL = NULL
    Then    Write ("Availability Stack Underflow")
            Return(FIRST)

2   [Obtain address of next free Node]
    NEW←AVAIL

3   [Remove free node from Availability Stack]
    AVAIL←LINK(AVAIL)

4   [Initialize fields of new node and its link to the list]
    INFO (NEW) ← X
    LINK (NEW) ← FIRST

5   [Return address of new node]
    Return (NEW)
```

When INSERT is invoked it returns a pointer value to the variable FIRST

FIRST ← INSERT (X, FIRST)

Function: INSEND(X, First) (Insert at end)

Given X, a new element and FIRST is a pointer to the first element of a linked linear list. Typical node contains INFO and LINK fields. AVAIL is a pointer to the top element of the availability stack; NEW is a temporary pointer variable. This function inserts a new node at the last position of linked list. This function returns address of FIRST node.

```
1   [Underflow?]
    IF        AVAIL = NULL
    Then      Write ("Availability Stack Underflow")
              Return(FIRST)

2   [Obtain address of next free Node]
    NEW←AVAIL

3   [Remove free node from Availability Stack]
    AVAIL←LINK(AVAIL)

4   [Initialize field of NEW node]
    INFO (NEW) ← X
    LINK (NEW) ← NULL

5   [Is the list empty?]
    If        FIRST = NULL
    then      Return (NEW)

6   [Initialize search for a last node]
    SAVE ← FIRST

7   [Search for end of list]
    Repeat while LINK (SAVE) ≠ NULL
              SAVE ← LINK (SAVE)

8   [Set link field of last node to NEW)
    LINK (SAVE) ← NEW

9   [Return first node pointer]
    Return (FIRST)
```

When INSERTEND is invoked it returns a pointer value to the variable FIRST

FIRST ← INSERTEND (X, FIRST)

Insert a node into Ordered Linked List

- There are many applications where it is desirable to maintain an ordered linear list. The ordering is in increasing or decreasing order on INFO field. Such ordering results in more efficient processing.
- The general algorithm for inserting a node into an ordered linear list is as below.
  1. Remove a node from availability stack.
  2. Set the field of new node.
  3. If the linked list is empty then return the address of new node.
  4. If node precedes all other nodes in the list then inserts a node at the front of the list and returns its address.
  5. Repeat step 6 while information contain of the node in the list is less than the information content of the new node.
  6. Obtain the next node in the linked list.
  7. Insert the new node in the list and return address of its first node.

Function: INSORD(X, FIRST)

Given X, a new element and FIRST is a pointer to the first element of a linked linear list. Typical node contains INFO and LINK fields. AVAIL is a pointer to the top element of the availability stack; NEW & SAVE are temporary pointer variables. This function inserts a new node such that linked list preserves the ordering of the terms in increasing order of their INFO field. This function returns address of FIRST node.

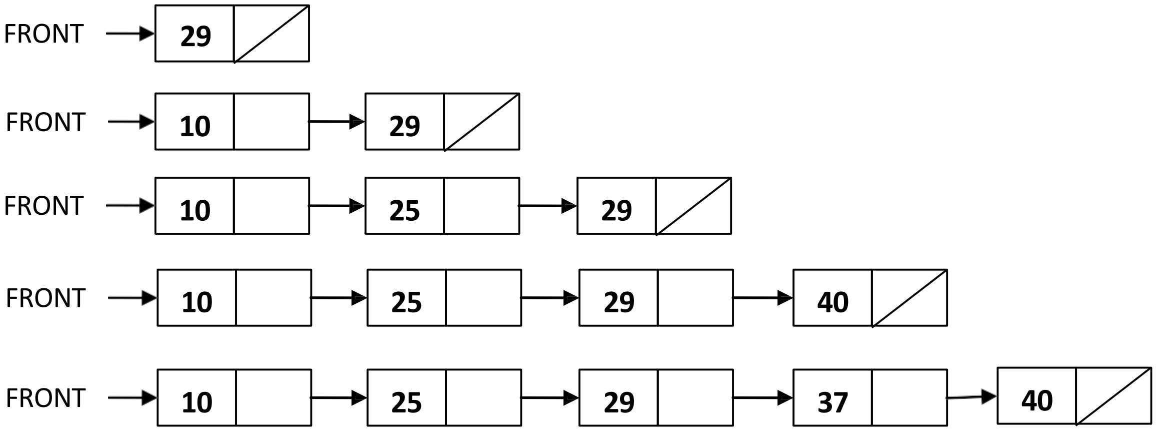
```
1   [Underflow?]
    IF      AVAIL = NULL
    Then    Write (“Availability Stack Underflow”)
            Return(FIRST)

2   [Obtain address of next free Node]
    NEW ← AVAIL

3   [Remove free node from Availability Stack]
    AVAIL ← LINK(AVAIL)

4.  [Is the list is empty]
    If      FIRST = NULL
    then    LINK (NEW) ← NULL
            Return (NEW)

5.  [Does the new node precede all other node in the list?]
    If      INFO(NEW) ≤ INFO (FIRST)
    then    LINK (NEW) ← FIRST
            Return (NEW)

6.  [Initialize temporary pointer]
    SAVE  ← FIRST

7.  [Search for predecessor of new node]
    Repeat while LINK (SAVE) ≠ NULL and  INFO (NEW) ≥ INFO (LINK (SAVE))
            SAVE ← LINK (SAVE)

8.  [Set link field of NEW node and its predecessor]
    LINK (NEW) ← LINK (SAVE)
    LINK (SAVE) ← NEW

9.  [Return first node pointer]
    Return (FIRST)
```

When INSERTORD is invoked it returns a pointer value to the variable FIRST

FIRST ← INSERTORD (X, FIRST)

By repeatedly involving function INSORD; we can easily obtains an ordered liner list for example the sequence of statements.

```
FRONT ← NULL
FRONT ← INSORD (29, FRONT)
FRONT ← INSORD (10, FRONT)
FRONT ← INSORD (25, FRONT)
FRONT ← INSORD (40, FRONT)
FRONT ← INSORD (37, FRONT)
```



Trace of construction of an ordered linked linear list using function INSORD

Algorithm to delete a node from Linked List

- Algorithm that deletes node from a linked linear list:-
  1. If a linked list is empty, then write under flow and return.
  2. Repeat step 3 while end of the list has not been reached and the node has not been found.
  3. Obtain the next node in list and record its predecessor node.
  4. If the end of the list has been reached then write node not found and return.
  5. Delete the node from list.
  6. Return the node into availability area.

Procedure: DELETE(X, FIRST)

Given X, an address of node which we want to delete and FIRST is a pointer to the first element of a linked linear list. Typical node contains INFO and LINK fields. SAVE & PRED are temporary pointer variables.

```
1. [Is Empty list?]
   If      FIRST = NULL
   then    write ('Underflow')
           return

2. [Initialize search for X]
   SAVE ← FIRST

3. [Find X]
   Repeat thru step-5 while SAVE ≠ X and LINK (SAVE) ≠ NULL

4. [Update predecessor marker]
   PRED ← SAVE

5. [Move to next node]
   SAVE ← LINK (SAVE)

6. [End of the list]
   If      SAVE ≠ X
   then    write ('Node not found')
           return

7. [Delete X]
   If      X = FIRST (if X is first node?)
   then    FIRST ← LINK (FIRST)
   else    LINK (PRED) ← LINK (X)

8. [Free Deleted Node]
   Free (X)
```

Function: COPY (FIRST)

- FIRST is a pointer to the first node in the linked list, this function makes a copy of the list.
- The new list is to contain nodes whose information and pointer fields are denoted by FIELD and PTR, respectively. The address of the first node in the newly created list is to be placed in BEGIN. NEW, SAVE and PRED are points variables.
- A general algorithm to copy a linked list
  1. If the list is empty then return null

2. If the availability stack is empty then write availability stack underflow and return else copy the first node.
3. Report thru step 5 while the old list has not been reached.
4. Obtain next node in old list and record its predecessor node.
5. If availability stack is empty then write availability stack underflow and return else copy the node and add it to the rear of new list.
6. Set link of the last node in the new list to null and return.

```
1. [Is Empty List?]
   If      FIRST = NULL
   then    return (NULL)

2. [Copy first node]
   NEW   ⇦   NODE
   New ← AVAIL
   AVAIL ← LINK (AVAIL)
   FIELD (NEW)
   ← INFO
   (FIRST) BEGIN
   ← NEW

3. [Initialize traversal]
   SAVE ← FIRST

4. [Move the next node if not at the end if list]
   Repeat thru step 6 while (SAVE) ≠ NULL

5. [Update predecessor and save pointer]
   PRED ← NEW
   SAVE ← LINK (SAVE)

6. [Copy node]
   If      AVAIL = NULL
   then    write ('Availability stack underflow')
           R
   eturn
   (0)
   else

   NEW
   ←
   AVAIL
           AVAIL ← LINK (AVAIL)
           FIELD
```

```
            (NEW) ←
            INFO (SAVE)
            PTR (PRED)
            ← NEW
7.  [Set link of last node and return]
    PTR (NEW) ← NULL
    Return (BEGIN)
```