PG-DAC SEPT-2021

ALGORITHMS & DATA STRUCTURES

SACHIN G. PAWAR

SUNBEAM INSTITUTE OF INFORMATION & TECHNOLOGIES

PUNE & KARAD

Data Structures: Introduction

Name of the Module : Algorithms & Data Structures Using Java.

Prerequisites: Knowledge of programming in C/C++/Java with Object Oriented Concepts.

Weightage : 100 Marks (Theory Exam : 40% + Lab Exam : 40% + Mini Project : 20%).

 # Importance of the Module:

1. CDAC - Syllabus
2. To improve programming skills
3. Campus Placements
4. Applications in Industry work

Data Structures: Introduction

Q. Why there is a need of data structure?

- There is a need of data structure to achieve 3 things in programming:

1. **efficiency**
2. **abstraction**
3. **reusability**

Q. What is a Data Structure?

Data Structure is **a way to store data elements into the memory** (i.e. into the main memory) in **an organized manner** so that operations like **addition, deletion, traversal, searching, sorting** etc... can be performed on it efficiently.

Data Structures: Introduction

Two types of **Data Structures** are there:

1. Linear / Basic data structures : data elements gets stored / arranged into the memory in a **linear manner** (e.g. sequentially) and hence can be accessed linearly / sequentially.

- **Array**
- **Structure & Union**
- **Linked List**
- **Stack**
- **Queue**

2. Non-Linear / Advanced data structures : data elements gets stored / arranged into the memory in a **non-linear manner** (e.g. hierarchical manner) and hence can be accessed non-linearly.

- **Tree (Hierarchical manner)**
- **Binary Heap**
- **Graph**
- **Hash Table(Associative manner)**

Data Structures: Introduction

+ **Array:** It is a **basic / linear data structure** which is **a collection / list of logically related similar type of data elements** gets stored/arranged into the memory at **contiguos locations**.

+ **Structure:** It is a **basic / linear data structure** which is **a collection / list of logically related similar and disimmilar type of data elements** gets stored/arranged into the memory **collectively i.e. as a single entity/record.**

sizeof of the structure = sum of size of all its members.

+ **Union:** Union is same like structure, except, memory allocation i.e. size of union is the size of max size member defined in it and that memory gets shared among all its members for effective memory utilization (can be used in a special case only).

Data Structures: Introduction

Q. What is a Program?

- A Program is **a finite set of instructions written in any programming language** (either in a high level programming language like C, C++, Java, Python or in a low level programming language like assembly, machine etc...) given to the machine to do specific task.

Q. What is an Algorithm?

- An algorithm is **a finite set of instructions written in any human understandable language (like english)**, if followed, acomplishesh a given task.

- **Pseudocode :** It is a **special form of an algorithm**, which is a finite set of instructions written in any human understandable language (like english) **with some programming constraints**, if followed, acomplishesh a given task.

- An algorithm is a template whereas a program is an implementation of an algorithm.

Data Structures: Introduction

 # Algorithm : to do sum of all array elements

Step-1: initially take value of sum is 0.

Step-2: scan an array seqentially from first element max till last element, and add each array element into the sum.

Step-3: return final sum.

 # Pseudocode : to do sum of all array elements

```
Algorithm ArraySum(A, n){//whereas A is an array of size n
    sum=0;//initially sum is 0
    for( index = 1 ; index <= size ; index++ ) {
    sum += A[ index ];//add each array element into the sum
    }
    return sum;
}
```

Data Structures: Introduction

- There are two types of Algorithms OR there are two approaches to write an algorithm:

1. Iterative (non-recursive) Approach :

```
Algorithm ArraySum( A, n){//whereas A is an array of size n
    sum = 0;
    for( index = 1 ; index <= n ; index++ ){
        sum += A[ index ];
    }
    return sum;
}
```

e.g. iteration

```
for( exp1 ; exp2 ; exp3 ){
    statement/s
}
```

exp1 => initialization
exp2 => termination condition
exp3 => modification

Data Structures: Introduction

2. Recursive Approach:
While writing recursive algorithm -> We need to take care about 3 things
1. Initialization: at the time first time calling to recursive function
2. Base condition/Termination condition : at the begining of recursive function
3. Modification: while recursive function call

Example:

```
Algorithm RecArraySum( A, n, index )
{
   if( index == n )//base condition
         return 0;

   return ( A[ index ] + RecArraySum(A, n, index+1) );
}
```

Data Structures: Introduction

Recursion : it is a process in which we can give call to the function within itself.
function for which recursion is used => recursive function
- there are two types of recursive functions:
1. tail recursive function : recursive function in which recursive function call is the last executable statement.

```
void fun( int n ){
    if( n == 0 )
        return;

    printf("%4d", n);
    fun(n--);//rec function call
}
```

2. non-tail recursive function : recursive function in which recursive function call is not the last executable statement

```
void fun( int n ){
    if( n == 0 )
        return;

    fun(n--);//rec function call
    printf("%4d", n);
}
```

Data Structures: Introduction

- An Algorithm is a solution of a given problem.
- **Algorithm = Solution**
- One problem may has many solutions. For example

Sorting : to arrange data elements in a collection/list of elements either in an ascending order or in descending order.

A1 : Selection Sort
A2 : Bubble Sort
A3 : Insertion Sort
A4 : Quick Sort
A5 : Merge Sort
etc...

- When one problem has many solutions/algorithms, in that case we need to select an efficient solution/algorithm, and to decide efficiency of an algorithm we need to do their analysis.

Data Structures: Introduction

- **Analysis of an algorithm** is a work of determining / calculating how much **time** i.e. computer time and **space** i.e. computer memory it needs to run to completion.

- There are two **measures** of an **analysis of an algorithms:**

1. Time Complexity of an algorithm is the amount of **time i.e. computer time** it needs to run to completion.

2. Space Complexity of an algorithm is the amount of **space i.e. computer memory** it needs to run to completion.

Data Structures: Introduction

 # Space Complexity of an algorithm is the amount of space i.e. computer memory it needs to run to completion.
Space Complexity = Code Space + Data Space + Stack Space (applicable only for recursive algo)
Code Space = space required for an **instructions**
Data Space = space required for **simple variables, constants & instance variables.**
Stack Space = space required for **function activation records** (local vars, formal parameters, return address, old frame pointer etc...).

- Space Complexity has **two components:**
1. Fixed component : code space and **data space** (space required for simple vars & constants).
2. Variable component : data space for instance characteristics (i.e. space required for instance vars) and **stack space** (which is applicable only in recursive algorithms).

Data Structures: Introduction

 # Calculation of Space complexity of non-recursive algorithm:

```
Algorithm ArraySum( A, n){//whereas A is an array of size n
    sum = 0;
    for( index = 1 ; index <= n ; index++ ){
    sum += A[ index ];
    }
    return sum;
}
```

Sp = Data Space + Instance charactristics
simple vars => formal params: A
local vars => sum, index
constants => 0 & 1
instance variable = n, input size of an array = **n units**
Data Space = 5 units (1 unit for simple var : A + 2 units for local vars : sum & index + 2 units for constants : 0 & 1) => **Data Space = 5 units**
Sp = (n + 5) units.

Data Structures: Introduction

S = C (Code Space) + Sp (Data Space)
S = C + (n+5)
S >= (n + 5) ... (as C is constant, it can be neglected)
S >= O(n) => O(n)
Space required for an algo = O(n) => whereas n = input size array.

 # Calculation of Space complexity of recursive algorithm:

```
Algorithm RecArraySum( A, n, index ){
    if( index == n )//base condition
            return 0;
    return ( A[ index ] + RecArraySum(A, n, index+1) );
}
```

Space Complexity = Code Space + Data Space + Stack Space (applicable only in recursive algorithms)
Code Space = space required for instructions
Data Space = space required for variables, constants & instance characteristics.
Stack Space = space required for FAR's.

Data Structures: Introduction

- When any function gets called, one entry gets created onto the stack for that function call, referred as **function activation record / stack frame,** it contains **formal params, local vars, return addr, old frame pointer etc...**
In our example of recursive algorithm:
3 units (for A, index & n) + 2 units (for constants 0 & 1) = total 5 **units** of memory is required per function call.
- for size of an array = **n,** algo gets called **(n+1) no. of times.**
Hence, total space required = **5 * (n+1)**
S = 5n + 5
=> S >= 5n
=> S >= 5n
=> S ~= 5n => O(n), wheras n = size of an array

Data Structures: Introduction

 # Time Complexity:

Time Complexity = Compilation Time + Execution Time

Time complexity has two components :

1. Fixed component : compilation time

2. Variable component : execution time => it depends on instance characteristics of an algorithm.

Example :

```
Algorithm ArraySum( A, n){//whereas A is an array of size n
    sum = 0;
    for( index = 1 ; index <= n ; index++ ){
    sum += A[ index ];
    }

    return sum;
}
```

Data Structures: Introduction

- for size of an array = 5 => instruction/s inside for loop will execute 5 no. of times
- for size of an array = 10 => instruction/s inside for loop will execute 10 no. of times
- for size of an array = 20 => instruction/s inside for loop will execute 20 no. of times
- for size of an array = **n** => instruction/s inside for loop will execute **“n”** no. of times

 # Scenario-1 :

Machine-1 : Pentium-4 : Algorithm : input size = 10
Machine-2 : Core i5 : Algorithm : input size = 10

 # Scenario-2 :

Machine-1 : Core i5 : Algorithm : input size = 10 : system fully loaded with other processes
Machine-2 : Core i5 : Algorithm : input size = 10 : system not fully loaded with other processes.

- It is observed that, **execution time is not only depends on instance characteristics**, it also depends on **some external factors** like hardware on which algorithm is running as well as other conditions, and hence it is not a good practice to decide efficiency of an algo i.e. calculation of time complexity on the basis of an execution time and compilation time, hence to do analysis of an algorithms **asymptotic analysis** is preferred.

Data Structures: Introduction

 # Asymptotic Analysis : It is a **mathematical way** to calculate time complexity and space complexity of an algorithm **without implementing it in any programming language.**

- In this type of analysis, analysis can be done on the basis of **basic operation** in that algorithm.

e.g. in searching & sorting algorithms **comparison** is the basic operation and hence analysis can be done on the basis of no. of comparisons, in addition of matrices algorithm **addition** is the basic operation and hence on the basis of addition operation analysis can be done.

"Best case time complexity" : if an algo takes **minimum** amount of time to run to completion then it is referred as best case time complexity.

"Worst case time complexity" : if an algo takes **maximum** amount of time to run to completion then it is referred as worst case time complexity.

"Average case time complexity" : if an algo takes **neither minimum nor maximum** amount of time to run to completion then it is referred as an average case time complexity.

Asympotic Notations:

1. Big Omega (Ω) : this notation is used to denote **best case time complexity** – also called as **asymptotic lower bound,** running time of an algorithm cannot be less than its asymptotic lower bound.

2. Big Oh (O) : this notation is used to denote **worst case time complexity** - also called as **asymptotic upper bound,** running time of an algorithm cannot be more than its asymptotic upper bound.

3. Big Theta (θ) : this notation is used to denote an **average case time complexity** - also called as **asymptotic tight bound,** running time of an algorithm cannot be less than its asymptotic lower bound and cannot be more than its asymptotic upper bound i.e. it is **tightly bounded.**

Data Structures: Searching Algorithms

1. Linear Search / Sequential Search:

 # Algorithm :

Step-1 : Scan / Accept value of key element from the user which is to be search.

Step-2 : Start traversal of an array and compare value of the key element with each array element sequentially from first element either till match is found or max till last element, **if key is matches with any of array element then return true otherwise return false if key do not matches with any of array element.**

 # Pseudocode:

```
Algorithm LinearSearch(A, size, key){
   for( int index = 1 ; index <= size ; index++ ){
   if( arr[ index ] ==  key )
   return true;
   }
   return false;
}
```

Data Structures: Searching Algorithms

Best Case: If key element is found at very first position in only 1 comparison then it is considered as a best case and running time of an algorithm in this case is **O(1)** => hence time complexity of linear search algorithm in base case = **Ω(1).**

Worst Case: If either key element is found at last position or key element does not exists, in this case maximum **n** no. of comparisons takes place, it is considered as a worst case and running time of an algorithm in this case is **O(n)** => hence time complexity of linear search algorithm in worst case = **O(n).**

Average Case: If key element is found at any in between position it is considered as an average case and running time of an algorithm in this case is **O(n/2) => O(n)** => hence time complexity = **θ(n).**

2. Binary Search / Logarithmic Search / Half Interval Search :

- This algorithm follows **divide-and-conquer** approach.

- To apply binary search on an array **prerequisite is that array elements must be in a sorted manner.**

Step-1: Accept value of key element from the user which is to be search.

Step-2: In first iteration, find/calculate **mid position** by the formula **mid=(left+right)/2**, (by means of finding mid position big size array gets divided logically into 2 subarrays, left subarray and right subarray, **left subarray => [left to mid-1]** & **right subarray => [mid+1 to right]**.

Step-3 : Compare value of key element with an element which is at mid position, **if key matches in very first iteration in only one comparison then it is considered as a best case**, if key matches with mid pos element then return true otherwise if key do not matches then we have to go to next iteration, and in next iteration we go to search key either into the left subarray or into the right subarray.

Step-4 : Repeat Step-2 & Step-3 till either key is found or max till subarray is valid, **if subarray is not valid then key is not found in this case return false.**

Data Structures: Searching Algorithms

- As in each iteration 1 comparison takes place and search space is getting reduced by half.

n => n/2 => n/4 => n/8 ......

after iteration-1 => n/2 + 1 => **$T(n) = (n/2^1) + 1$**

after iteration-2 => n/4 + 2 => **$T(n) = (n/2^2) + 2$**

after iteration-3 => n/8 + 3 => **$T(n) = (n/2^3) + 3$**

Lets assume, after **k** iterations => **$\underline{T(n) = (n/2^k) + k}$ ...... (equation-I)**

let us assume,

=> **$n = 2^k$**

=> $\log n = \log 2^k$ (by taking log on both sides)

=> $\log n = k \log 2$

=> $\log n = k$ (as $\log 2 \sim= 1$)

=> $\underline{k = \log n}$

By substituting value of **$n = 2^k$ & $k = \log n$** in **equation-I,** we get

=> $T(n) = (n / 2^k) + k$

=> $T(n) = (2^k / 2^k) + \log n$

=> $T(n) = 1 + \log n$ => $T(n) = O(1 + \log n$ **) => $T(n) = \underline{O(\log n)}$.**

```
Algorithm BinarySearch(A, n, key)//A is an array of size "n", and key to be search
{
    left = 1;
    right = n;

    while( left <= right )
    {
        //calculate mid position
        mid = (left+right)/2;
        //compare key with an ele which is at mid position
        if( key == A[ mid ] )//if found return true
            return true;

        //if key is less than mid position element
        if( key < A[ mid ] )
        {
            right = mid-1;//search key only in a left subarray
        }
        else//if key is greater than mid position element
        {
            left = mid+1;//search key only in a right subarray
        }
    }//repeat the above steps either key is not found or max any subarray is valid
    return false;
}
```

Best Case: if the key is found in very first iteration at mid position in only 1 comparison OR if key is found at root position it is considered as a best case and running time of an algorithm in this case is O(1) = **Ω(1).**

Worst Case: if either key is not found or key is found at leaf position it is considered as a worst case and running time of an algorithm in this case is O(log n) = **O(log n).**

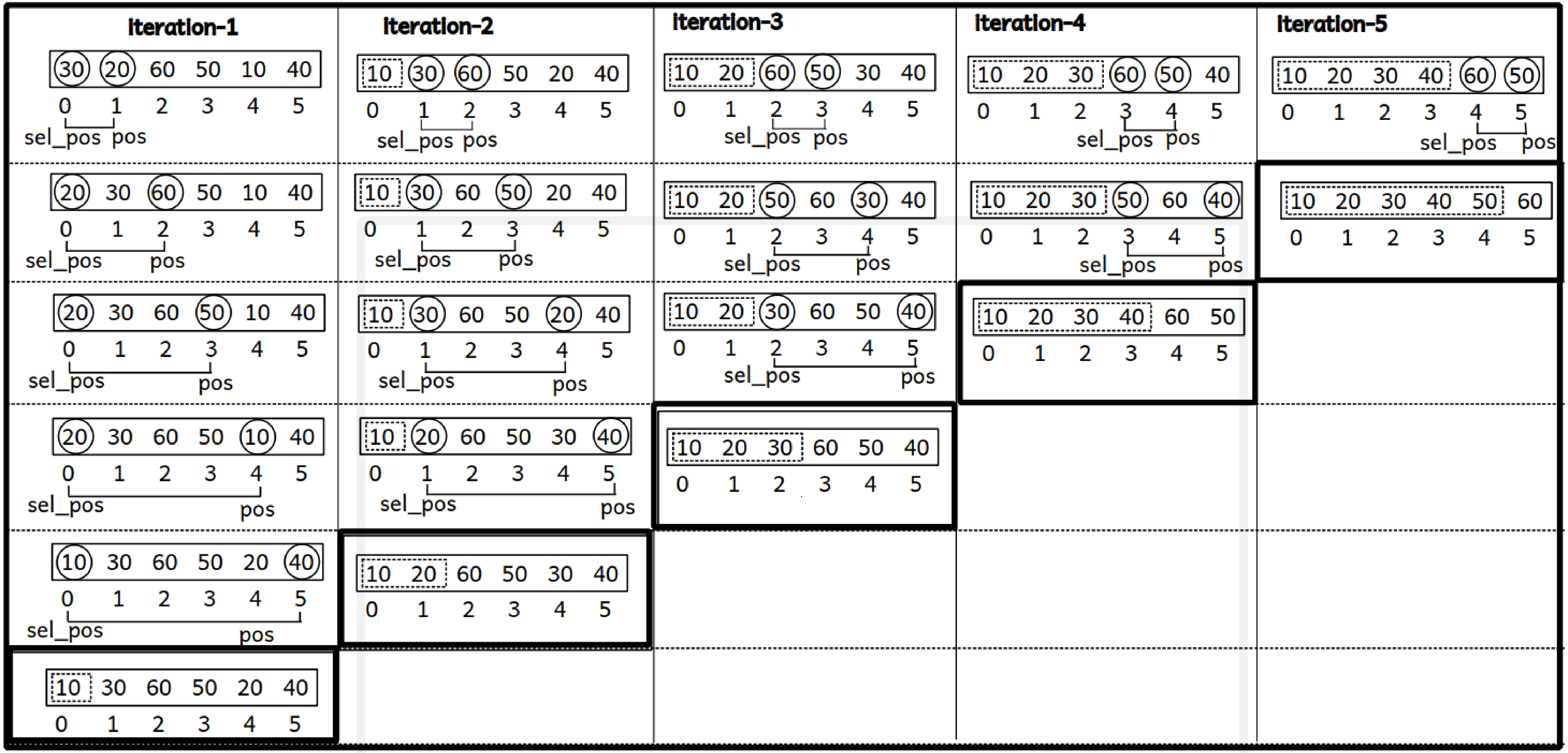
Average Case: if key is found at non-leaf position it is considered as an average case and running time of an algorithm in this case is O(log n) = **θ(log n).**

1. Selection Sort:

- In this algorithm, in first iteration, **first position gets selected** and **element which is at selected position gets compared with all its next position elements sequentially**, **<u>if an element at selected position found greater than any other position element then swapping takes place</u>** and in first iteration **smallest element** gets setteled at first position.

- In the second iteration, **second position gets selected and element which is at selected position gets compared with all its next position elements, if an element selected position found greater than any other position element then swapping takes place** and in second iteration **second smallest element** gets setteled at second position, and so on **<u>in maximum (n-1) no. of iterations all array elements gets arranged in a sorted manner.</u>**

Data Structures: Sorting Algorithms



Best Case : $\Omega(n^2)$

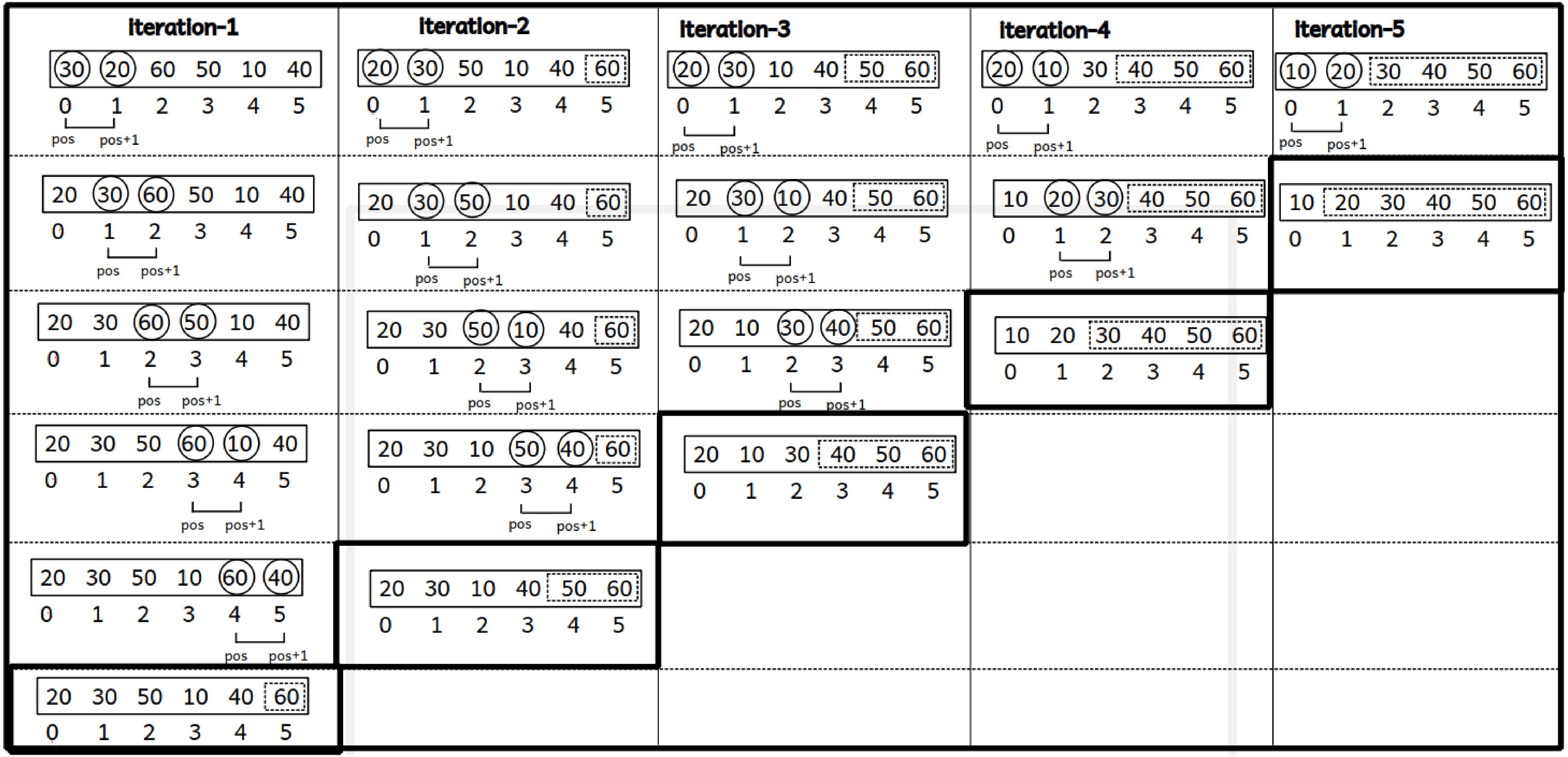
Worst Case : $O(n^2)$

Average Case : $\theta(n^2)$

2. Bubble Sort :

- In this algorithm, **in every iteration elements which are at two consecutive positions gets compared, if they are already in order then no need of swapping between them, but if they are not in order i.e. if prev position element is greater than its next position element then swapping takes place**, and by this logic in first iteration largest element gets setteled at last position, in second iteration second largest element gets setteled at second last position and so on, **in max (n-1) no. of iterations all elements gets arranged in a sorted manner.**

Data Structures: Sorting Algorithms



Best Case **: Ω(n) –** if array elements are already arranged in a sorted manner.

Worst Case **: $O(n^2)$**
Average Case **: $\theta(n^2)$**

3. Insertion Sort:
- In this algorithm, in every iteration one element gets selected as a **key element** and key element gets inserted into an array at its appropriate position towards its left hand side elements in a such a way that elements which are at left side are arranged in a sorted manner, and so on, in max **(n-1)** no. of iterations all array elements gets arranged in a sorted manner.
- **This algorithm works efficiently for already sorted input sequence by design** and hence running time of an algorithm is O(n) and it is considered as a best case.

Best Case : $\Omega(n)$ – if array elements are already arranged in a sorted manner.

Worst Case : $O(n^2)$

Average Case: $\theta(n^2)$

- Insertion sort algortihm is an efficient algorithm for smaller input size array.

Data Structures: Linked List

- Limitations of an array data structure:

1. Array is static, i.e. size of an array is fixed, its size cannot be either grow or shrink during runtime.

2. Addition and deletion operations on an array are not efficient as it takes O(n) time, and hence to overcome these two limitations of an Array data structure **Linked List** data structure has been designed.

Linked List: It is a basic/linear data structure, which is a collection/list of logically related similar type of elements in which, an address of first element in a collection/list is stored into a pointer variable referred as a head pointer and each element contains actual data and link to its next element i.e. an address of its next element (as well as an addr of its previous element).

- An element in a Linked List is also called as a **Node.**

- Four types of linked lists are there: **Singly Linear Linked List, Singly Circular Linked List, Doubly Linear Linked List and Doubly Circular Linked List.**

Data Structures: Linked List

- Basically we can perform **addition, deletion, traversal** etc... operations onto the linked list data structure.

- We can add and delete node into and from linked list by three ways:

add node into the linked list **at last position**, **at first position** and **at any specific position**, simillarly we can delete node from linked list which is **at first position**, **at last position** and **at any specific position.**

1. Singly Linear Linked List: It is a type of linked list in which

- head always contains an address of first element, if list is not empty.

- each node has two parts:

i. data part : it contains actual data of any primitive/non-primitive type.

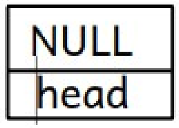
ii. pointer part (next) : it contains an address of its next element/node.

- last node points to NULL, i.e. next part of last node contains NULL.

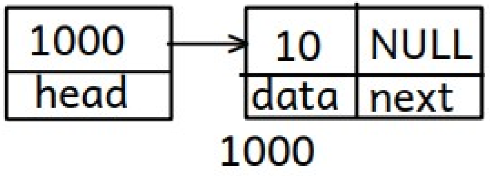
Data Structures: Linked List

SINGLY LINEAR LINKED LIST

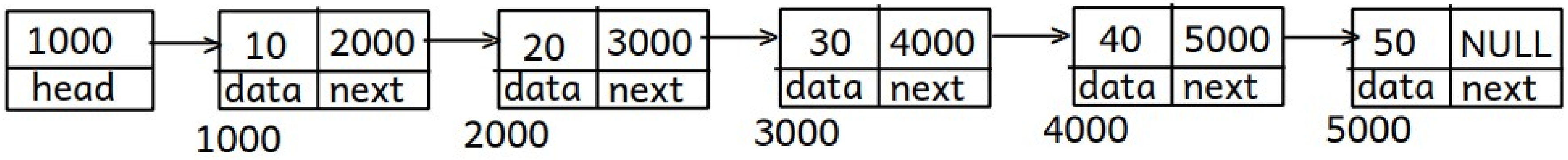
1) singly linear linked list --> list is empty



2) singly linear linked list --> list contains only one node



3) singly linear linked list --> list contains more than one nodes



Data Structures: Linked List

Limitations of Singly Linear Linked List:

- Add node at last position & delete node at last position operations are not efficient as it takes O(n) time.

- We can start traversal only from first node and can traverse the list only in a forward direction.

- Previous node of any node cannot be accesed from it.

- Any node cannot be revisited – to overcome this limitation Singly Circular Linked List has been designed.

2. Singly Circular Linked List: It is a type of linked list in which

- head always contains an address of first node, if list is not empty.

- each node has two parts:

i. data part : contains data of any primitive/non-primitive type.

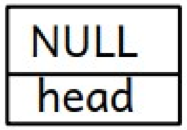
ii. pointer part(next) : contains an address of its next node.

- last node points to first node, i.e. next part of last node contains an address of first node.

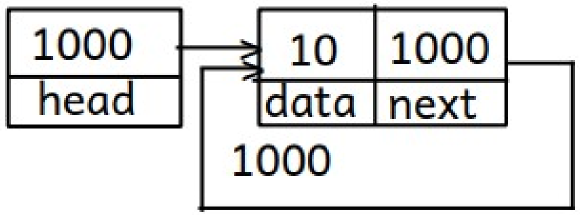
Data Structures: Linked List

SINGLY CIRCULAR LINKED LIST

1) singly circular linked list --> list is empty



2) singly circular linked list --> list contains only one node



3) singly circular linked list --> list contains more than one nodes