ASSIGNMENT - 01

1. "To write an efficient program, we should know about DS: Explain the above statements.

⇒ The reason why we should know about data structures to write an efficient program is because a good knowledge of data structure help in data organization, management and storage format that enables efficient access and modification. Data structures provide a set of techniques to the programmer for handling the data efficiently. The concept of data structure can be implemented in any programming language. It helps the programmer to collect data values in efficient manner and also makes them understand the relationship among those data values and gives us the clear idea about the functions or operations that can be applied to the data.

2. convert the following into both postfix and prefix.

a) $(A+B)*(C+D-E)*F$

POSTFIX

Input	Stack	Output
(	(	#
A	(	A
+	(+	A
B	(+	AB
)		AB+
X	X	AB+
C	XC	AB+
C	XC	AB+C
+	XC+	AB+C
D	XC+	AB+CD
−	XC+−	AB+CD+

E	×C−	AB+CD+E
)	×	AB+CD+E−
×	×	AB+CD+E−×
F	×	AB+CD+E−×F
#	#	AB+CD+E−×F×

So, the required postfix expression is AB+CD+E−*F*

Prefix

First we find the reverse of the given exp^n.
"(A+B)*(C+D−E)*F"

F*(E−D+C)*(B+A)

Then we find the postfix of the given expression

Input	Stack	Output
F		F
*	*	F
(	*(	F
E	*(	FE
−	*(−	FE
D	*(−	FED
+	*(+	FED−
C	*(+	FED−C
)	*	FED−C+
*	*	FED−C+*
C	*C	FED−C+*
B	*C	FED−C+*B
+	*C+	FED−C+*B
A	*C+	FED−C+*BA
)	*	FED−C+*BA+
		FED−C+*BA+*

Now, we reverse the output.
So, the equivalent prefix form is

+AB+C-DEF

b) (A+B^D)/(E-F)+G

Postfix

Input	Stack	Output
(	(	#
A	(	A
+	(+	A
B	(+	AB
^	(+^	AB
D	(+^	ABD
)		ABD^+
/	/	ABD^+
C	/C	ABD^+
E	/C	ABD^+E
-	/C-	ABD^+E
F	/C-	ABD^+EF
)	/	ABD^+EF-
+	+	ABD^+EF-/
G	+	ABD^+EF-/G
#	#	ABD^+EF-/G+

So, the equivalent postfix form is
ABD^+EF-/G+

Prefix
First we reverse the given expression.
"(A+B^D)/(E-F)+G" as
+G/(F-E)/(D^B+A)

Now, we find the postfix of the reversed expression.

Input	Stack	Output
G		G
+	+	G
C	+C	G
F	+C	GF
−	+C−	GF
E	+C−	GFE
)	+	GFE−
/	+/	GFE−
(	+/C	GFE−
D	+/C	GFE−D
^	+/C^	GFE−D
B	+/C^	GFE−DB
+	+/C+	GFE−DB^
A	+/C+	GFE−DB^A
)	+/	GFE−DB^A+
#	#	GFE−DB^A+/+

Now, we reverse the output.
So, the equivalent prefix form is

+/+A^BD−EFG

G⇒ ~~We can~~

C. How do you evaluate a postfix expression? Explain with an example.

⇒ We can evaluate a postfix expression by using the following steps stated below:-

i) Start

ii) Scan the postfix expression from left to right

iii) If the scanned character is an operand, push it into the stack.

iv) If the scanned character is an operator,

Operand 1 = pop();

Operand 2 = pop();

Result = Operand 2 operator Operand 1

Push the result into the stack

v) Stop

Example

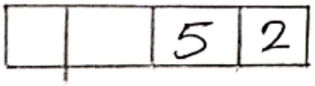
25+

Scanned character = 2

Push 2 into the stack

Scanned character = 5

Push 5 into the stack



Scanned character = +

Pop both operands and perform the + operation and push the result back into the stack

Operand 1 = Pop() = 5

Operand 2 = pop() = 2

result = Operand 2 operator Operand 1 = 2 + 5 = 7

2. Explain the concept of Priority Queue in detail.

=) A priority queue is a collection of elements such that each element has been assigned a priority value such that the order in which elements are deleted and processed comes from the following rules:

i) An element of higher priority is processed before any element of lower priority.

ii) Two elements with the same priority are processed according to the order in which they were added to the queue.

There are two types of priority queues:

a) Descending priority queue (max priority queue)

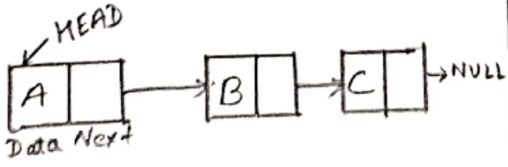
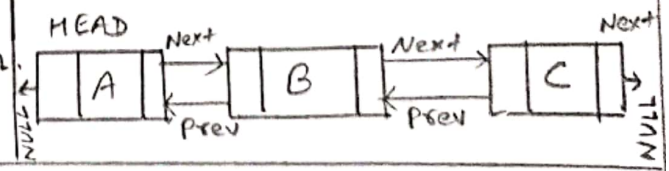
In max priority queue, elements are inserted in the order in which they arrive the queue and always maximum value is removed first from the queue. The following are the operations perpormed in a max priority queue: isEmpty(), insert(), findMax(), remove().

b) Ascending priority queue (min priority queue)

Min priority queue is similar to max priority queue except removing maximum element first, we remove minimum element first. The following operations are performed in Min priority Queue: isEmpty(), insert(), findMin(), remove().

4. Differentiate between SLL and DLL. Explain the applications of Linked list. Explain the implementation of Queue using SLL.

=) The differentiate between SLL and DLL is as follows:

	Singly Linked List (SLL)	Doubly linked List (DLL
i)	SLL nodes contains 2 field - data field and next link field.	DLL nodes contains 3 fields data field, a previous link field and a next link field.
ii)	In SLL, the traverssal can be done using the next node link only. Thus traversal is possible in one direction only.	In DLL, the traverssal can be done using the previous node link or the next node link. Thus traversal is possible in both directions (forward or backward).
iii)	The SLL occupies less memory than DLL as it has only 2 fields.	The DLL occupies more memory than SLL as it has 3 fields.
iv)	Complexity of insertion and deletion at a given position is O(n).	Complexity of insertion and deletion at a given position is O(1).
v)	Ex 	

=) The application of Linked list are as follows:

i) Implementation of stacks and queues.

ii) Implementation of graphs: Adjacency ~~of~~ list representation of graphs is most popular which uses linked list to store adjacent vertices.

iii) Dynamic Memory Allocation: We use linked list of free blocks.

iv) Maintaining directory of names.

v) Performing arithmetic operations on long integers.

vi) Manipulation of polynomials by storing constants in the node of linked list.

vii) Representing sparse matrices.

=> In Linked list implementation of queue, we need to create a node for each data items and arrange nodes in first in first out manner.

Algorithm (Enqueue Operation)

=> Create a new node with the value to be inserted.

=> If the Queue is empty, then set both front and rear to point to newNode.

=> If the Queue is not empty, then set next of rear to the newNode and the rear to point to the newNode.

The time complexity of Enqueue operation is O(1).

Algorithm (Dequeue)

=> If the Queue is empty, terminate the method.

=> If the Queue is not empty, increment front to point to nextNode.

=> Finally check if the front is NULL, then set rear to NULL also. This signifies empty Queue.

The time complexity for Dequeue operation is O(1).

5. Explain the TOH problem in detail. Also draw the recursion tree for move('A', 'C', 'B', 4).

=> TOH problem is a recursion based problem having initial state

a) There are three poles named as origin, intermediate and destination.

ii) n number of different-sized disks having hole at the center is stacked around the origin pole in decreasing order.

iii) The disks are numbered as 1, 2, 3, 4, ... n.

Objective

i) Transfer all disks from origin pole to destination pole using intermediate pole for temporary storage.

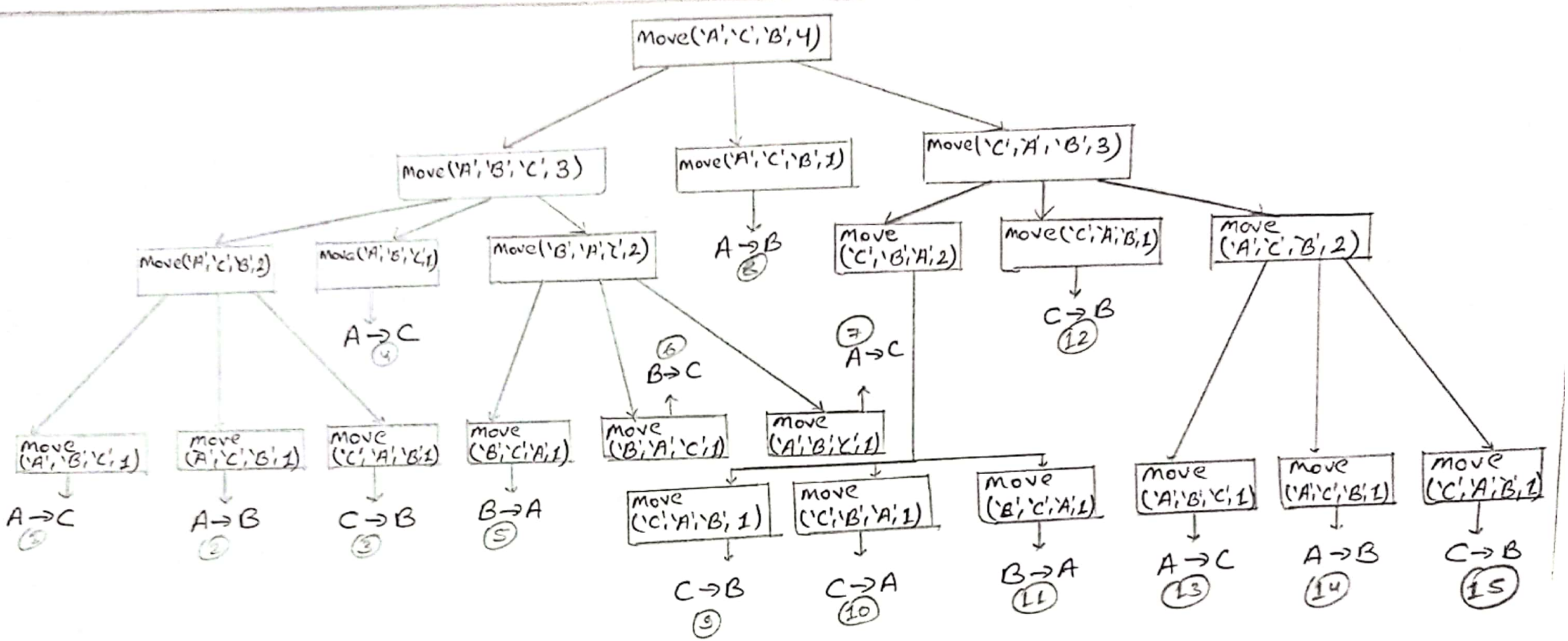
Conditions

i) Move only one disk at a time

ii) Each disk must always be placed around one of the pole.

iii) Never place larger disk on top of smaller disk.

=> The recursion tree for move('A', 'C', 'B', 4) is shown below:-



Move C

6. Hand-test Insertion Sort, Bubble Sort, Selection-Sort, Quick-Sort, Merge Sort, Heap Sort algorithm with the data given below:

56, 23, 14, 20, 65, 7, 8, 14, 15, 25

⇒ Insertion Sort

Pass	Array Position	0	1	2	3	4	5	6	7	8	9
0	Initial state	56	23	14	20	65	7	8	14	15	25
1	After A[0..1] is sorted	23	56	14	20	65	7	8	14	15	25
2	After A[0..2] is sorted	14	23	56	20	65	7	8	14	15	25
3	After A[0..3] is sorted	14	20	23	56	65	7	8	14	15	25
4	After A[0..4] is sorted	14	20	23	56	65	7	8	14	15	25
5.	After A[0..5] is sorted.	7	14	20	23	56	65	8	14	15	25
6.	After A[0..6] is sorted	7	8	14	20	23	56	65	14	15	25
7.	After A[0..7] is sorted	7	8	14	14	20	23	56	65	15	25
8.	After A[0..8] is sorted	7	8	14	14	15	20	23	56	65	25
9.	After A[0..9] is sorted	7	8	11	14	15	20	23	25	56	65

⇒ Bubble Sort

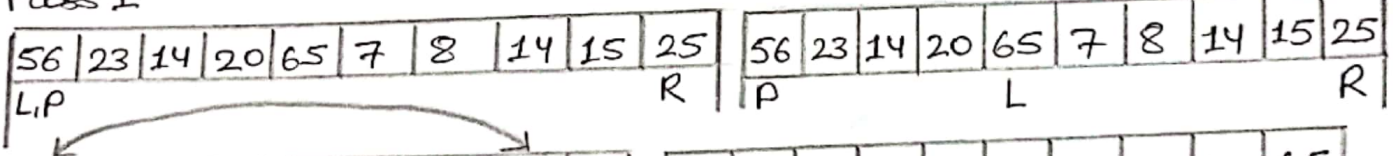
Pass	Array Position	0	1	2	3	4	5	6	7	8	9
	Initial State	56	23	14	20	65	7	8	14	15	25
	Pass 1	23	14	20	56	7	8	14	15	25	65
	Pass 2	14	20	23	7	8	14	15	25	56	65
	pass 3	14	20	7	8	14	15	23	25	56	65
	Pass 4	14	7	8	14	15	20	23	25	56	65
	Pass 5	7	8	14	14	15	20	23	25	56	65
	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮
	pass 19	7	8	14	14	15	20	23	25	56	65

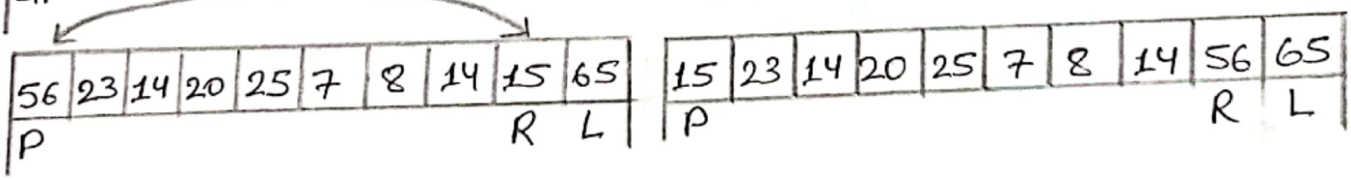
⇒ SELECTION SORT

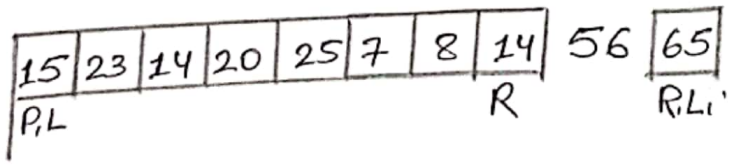
Array Pos.	0	1	2	3	4	5	6	7	8	9
Initial state	56	23	14	20	65	7	8	14	15	25
Pass 1	7	23	14	20	65	56	8	14	15	25
Pass 2	7	8	14	20	65	56	23	14	15	25
Pass 3	7	8	14	20	65	56	23	14	15	25
Pass 4	7	8	14	14	65	56	23	20	15	25
Pass 5	7	8	14	14	15	56	23	20	65	25
Pass 6	7	8	14	14	15	20	23	56	65	25
Pass 7	7	8	14	14	15	20	23	56	65	25
Pass 8	7	8	14	14	15	20	23	25	65	56
Pass 9	7	8	14	14	15	20	23	25	56	65
Pass 10	7	8	14	14	15	20	23	25	56	65

⇒ Quick Sort

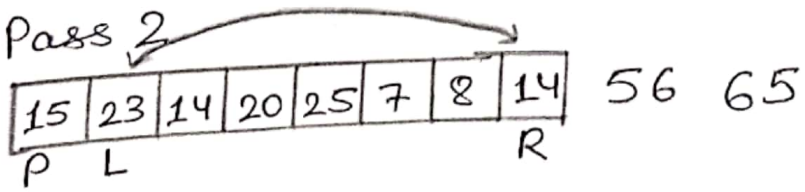
Pass 1



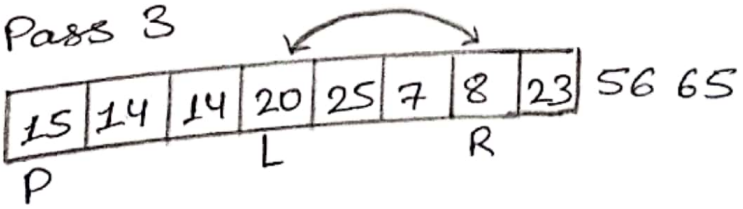


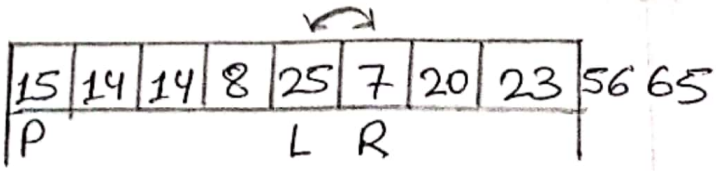


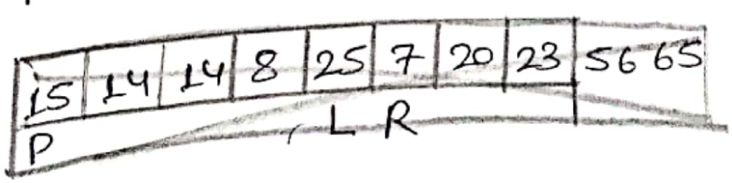
Pass 2

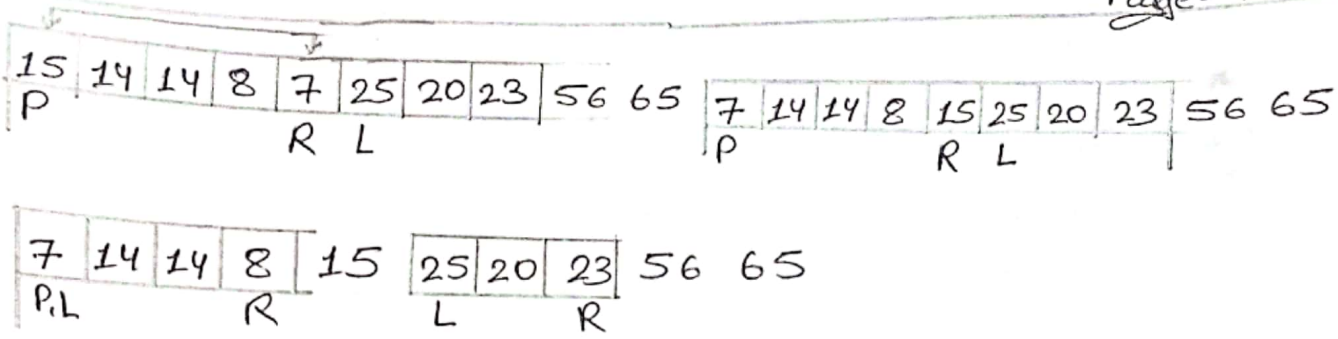


Pass 3

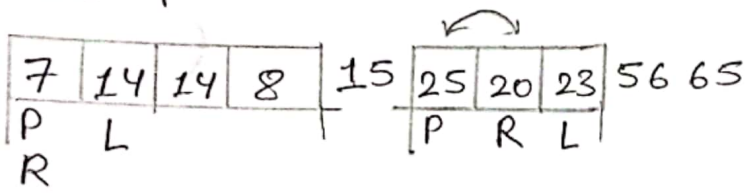




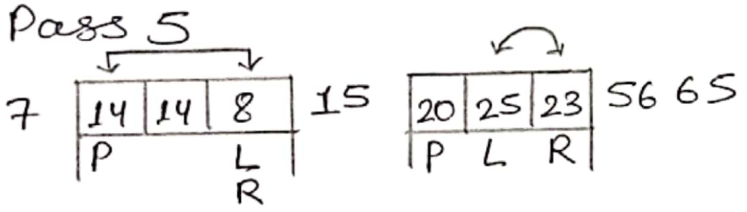


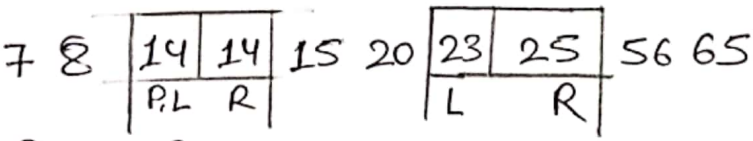


Pass 4

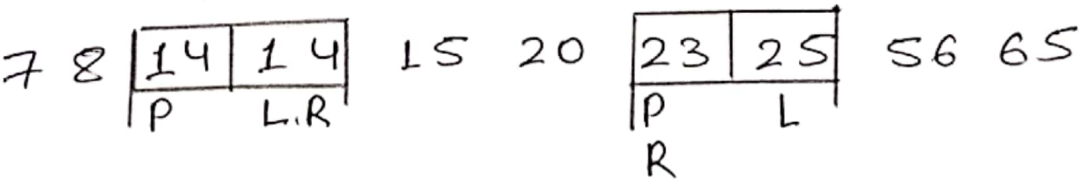


Pass 5





Pass 6

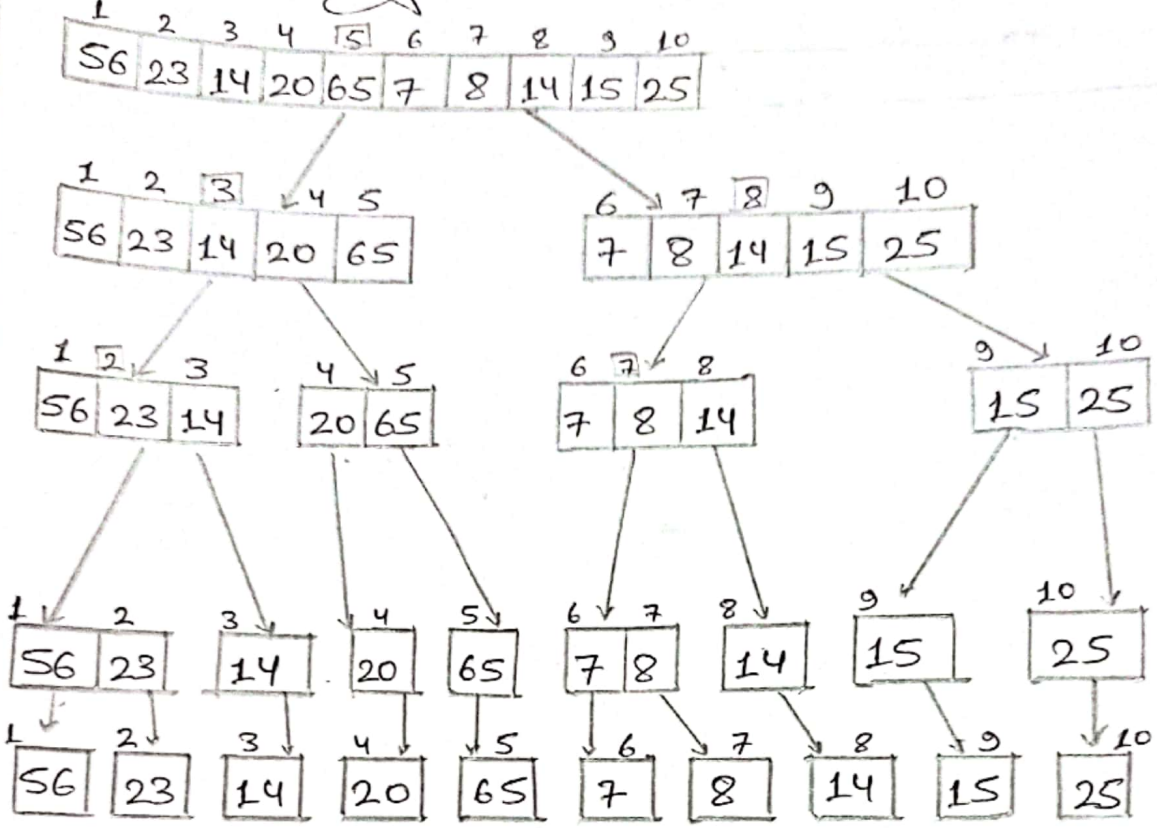


Pass 7

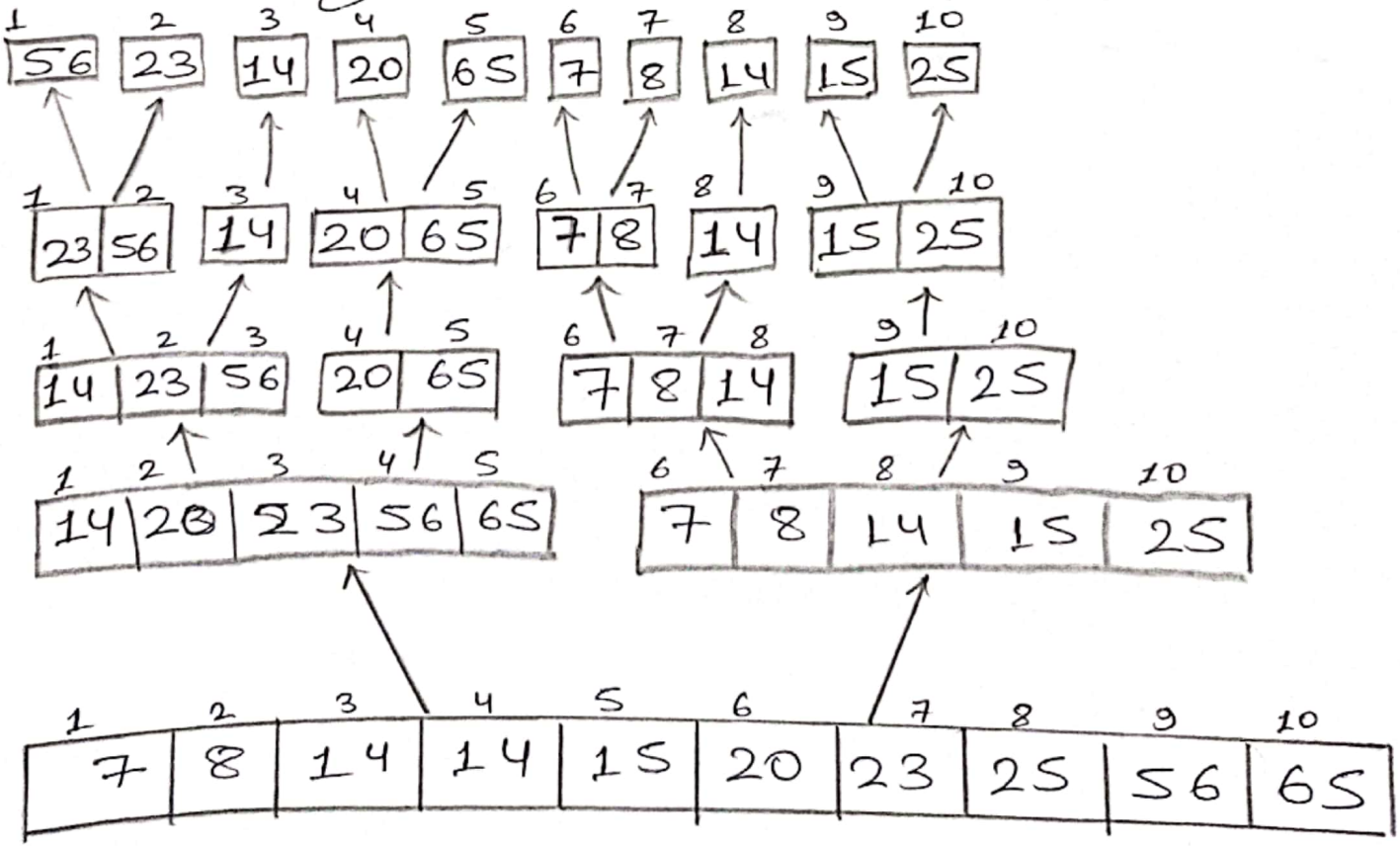
7 8 14 14 15 20 23 25 56 65

Merge Sort

Pass 1: Dividing

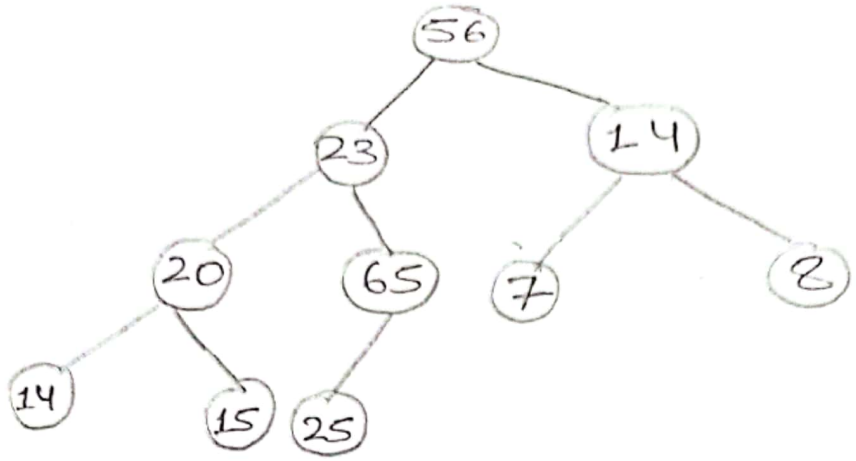


Pass 2: Merging

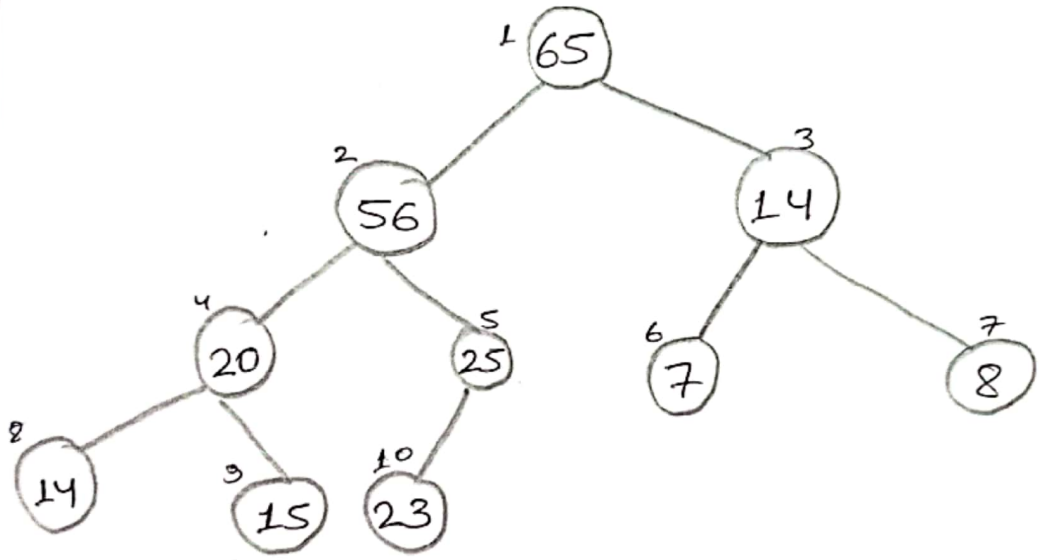


⇒ HEAP Sort

Constructing a binary tree of given array

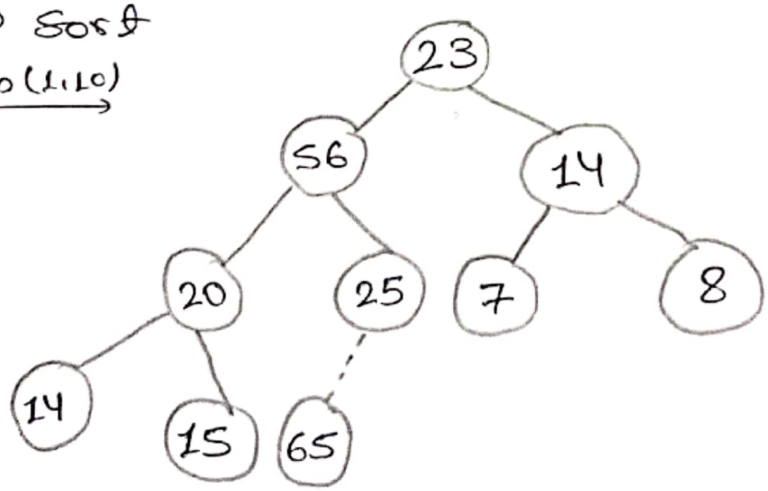


Now, constructing heap of given tree as

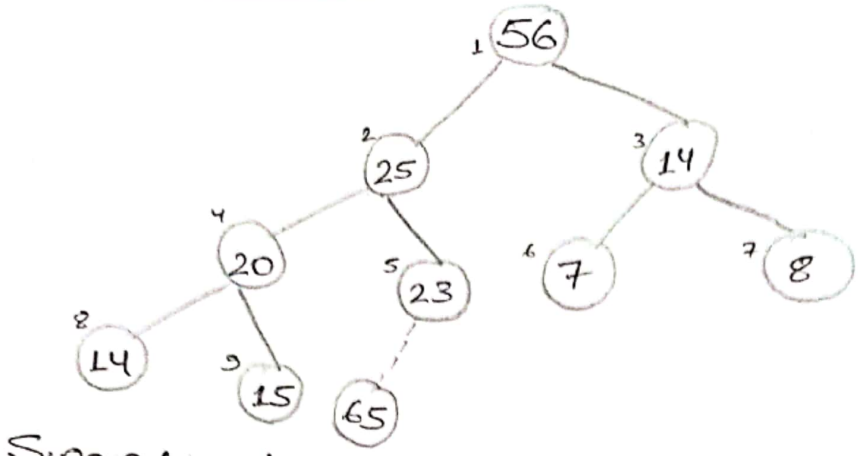


Heap Sort

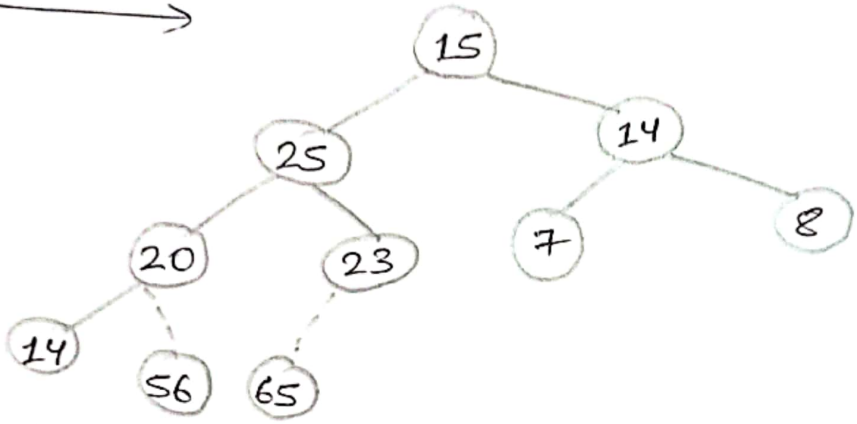
Swap (1, 10)



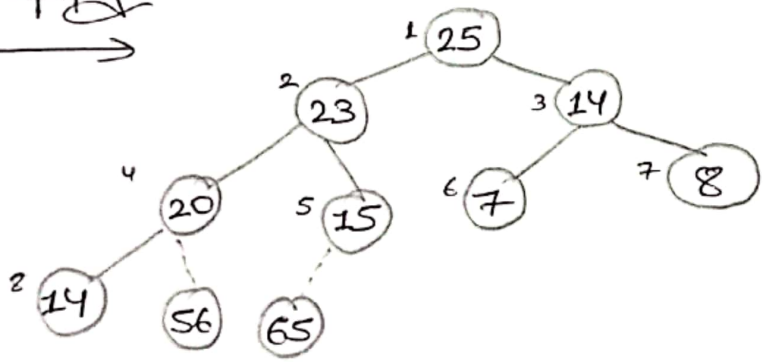
Heapify



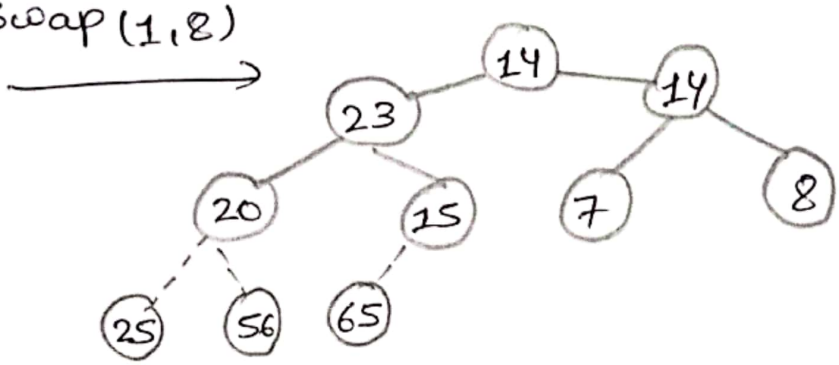
Swap (1,9)



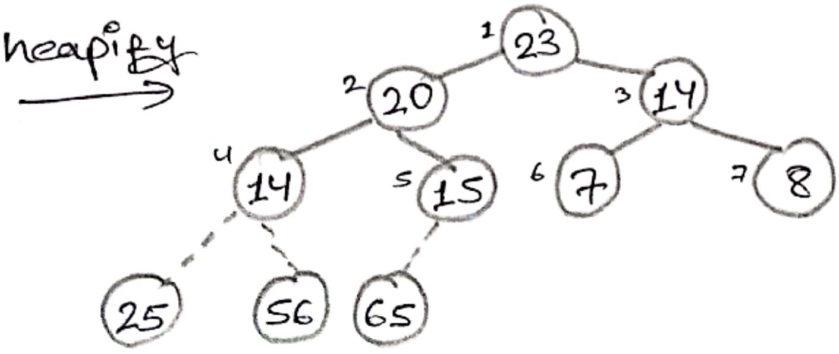
heapify



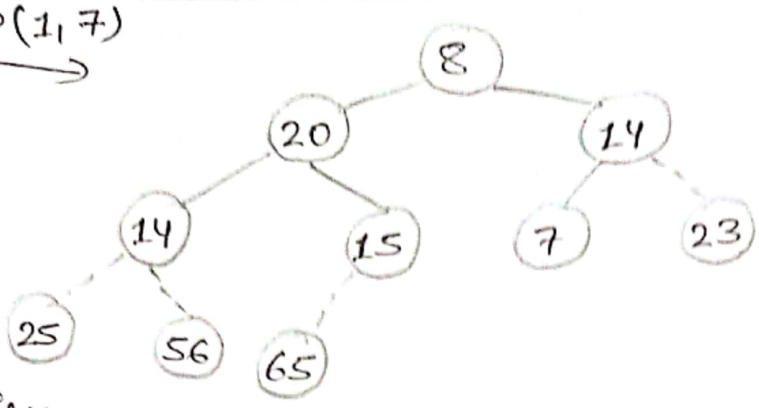
Swap (1,8)



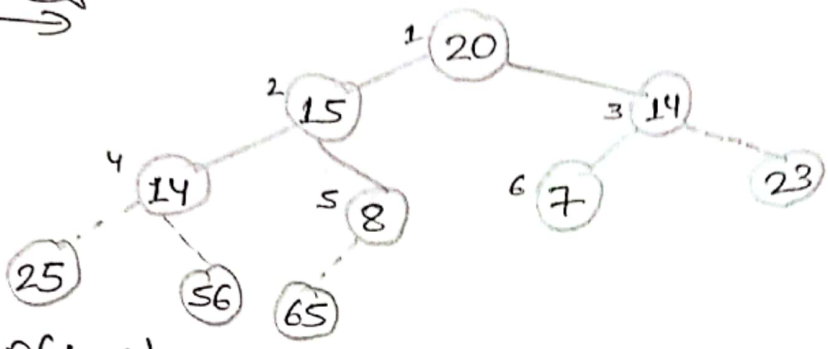
heapify



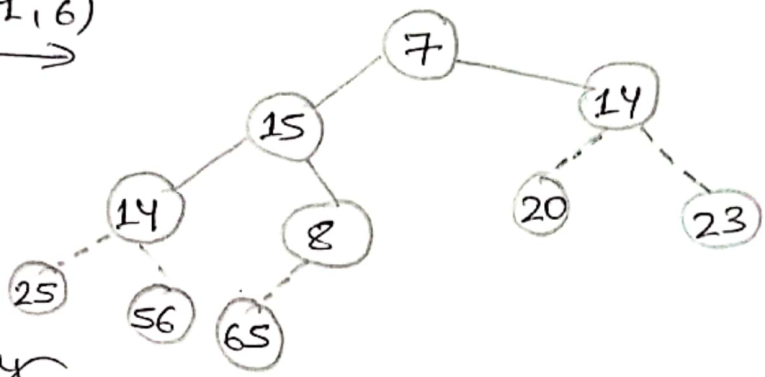
Swap(1,7)



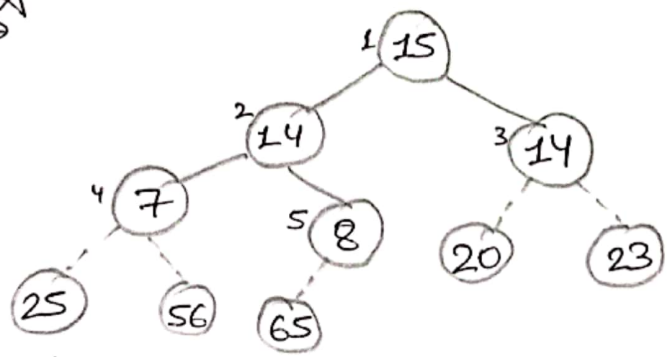
heapify



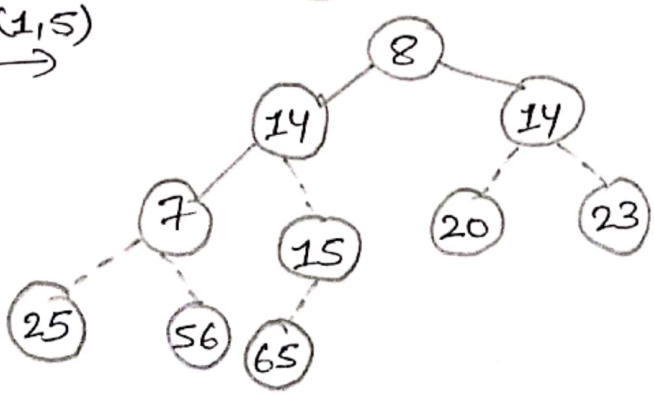
Swap(1,6)



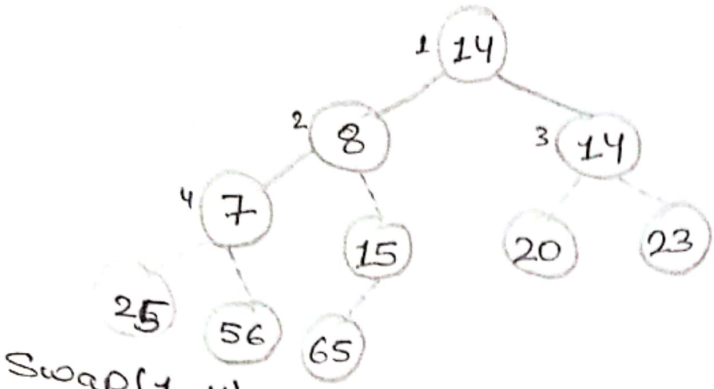
heapify



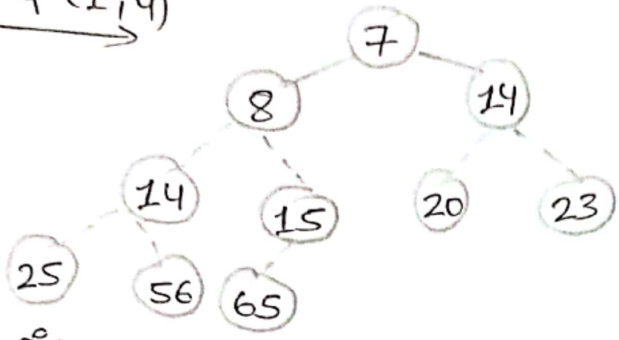
Swap(1,5)



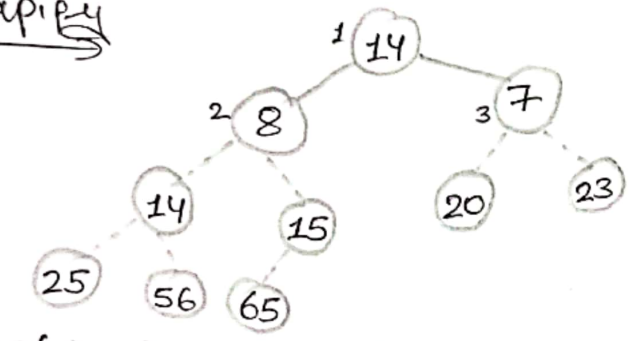
heapify



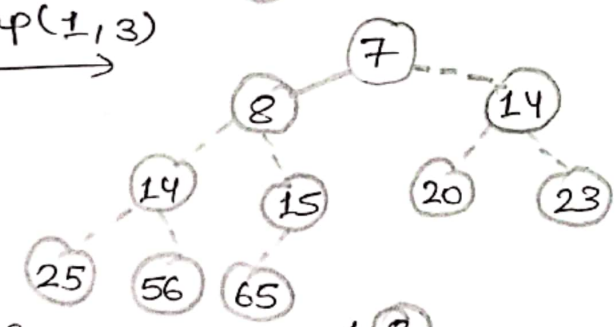
Swap(1,4)



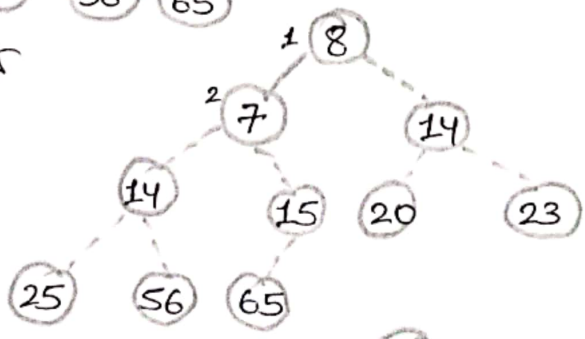
heapify



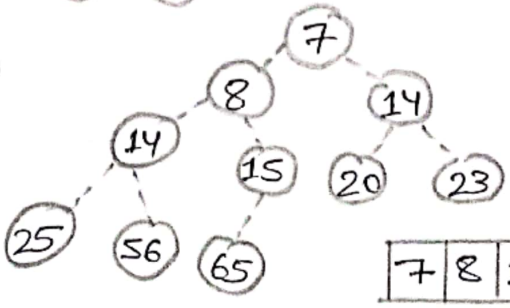
Swap(1,3)



heapify



Swap(1,2)



7	8	14	14	15	20	23	25	56	65

7. Discuss binary search algorithm? Write a recursive algorithm to implement binary search. What are the benefits of using hashing? How do you choose a hash function?

=> Binary search algorithm finds given element in a list of elements with O(log n) time complexity where n is total number of elements in the list. The binary search algorithm can be used with only sorted list of element. That means, binary search can be used only with list of elements which are already arranged in an order.

Algorithm

i) Start
ii) Read the search element from the user.
iii) Find the middle element in the sorted list.
iv) Compare, the search element with the middle element in the sorted list.
v) If both are matching, then display "Given element found!!!" and terminate the function.
vi) If both are not matching, then check whether the search element is smaller or larger than middle element.
vii) If the search element is smaller than middle element, then repeat steps 2,3,4 and 5 for the left sub list of the middle element.
viii) If the search element is larger than middle element, then repeat steps 2,3,4 and 5 for the right sub list of the middle element.
ix) Repeat the same process until we find the search element in the list or until sub list contains only one element.
x) If that element also doesn't match with the search element, then display "Element not found in the list" and terminate the function.
xi) Stop

Pseudo code

```
BinarySearch(a, l, r, key)
{
   int m;
   int flag=0;
   if(l<=r)
   {
      m=(l+r)/2;
      if(key==a[m])
         flag=m;
      else if(key<a[m])
         return BinarySearch(a, l, m-1, key);
      else
         return BinarySearch(a, m+1, r, key);
   }
   else
      return flag;
}
```

⇒ Hashing is an efficient searching technique in which key is placed in direct accessible address for rapid search. Hashing provides the direct access of records from the file no matter where the record is in the file. Due to which it reduces the unnecessary comparisons. The search time is reduced from O(n), as in a sequential search, or from O(logn), as in a binary search, to at least O(1); regardless of the number of elements being searched, the run time is always the same. But this is just an ideal, and in real applications, this ideal can only be approximated. These are the benefits of using hashing.

⇒ A function that transforms a key into a table index is called a hash function. A hash function is any function that can be used to map data of arbitrary size to data of fixed size.

Choosing a good hashing function, h(k), is essential for hash-table based searching. h should distribute the elements of our collection as uniformly as possible to the "slots" of the hash table. The key criterion is that there should be a minimum number of collision.

8.

a) Define graph. Discuss Dijkstra's algorithm for finding shortest path in a graph.

⇒ Graph is a non linear data structure; it contains a set of points known as nodes (or vertices) and set of links known as edges (or Arcs) which connects the vertices. A graph is an ordered pair G=(V,E) comprising a set V of vertices or nodes together with a set E of edges or links. Each edge has either one or two vertices associated with it, called its end points. If the edge pair is ordered, the graph is called a directed graph.

The algorithm of Dijkstra's for finding shortest path in a graph is as follows:

Precondition: G=(V,w) is a weighted graph with initial vertex V_0 then it holds following steps:

(i) Initialize the distance field to 0 for V_0 and to for each of the other vertices.

(ii) Enqueue all the vertices into a priority queue Q with highest priority being the lowest distance field value.

(iii) Repeat steps 4-10 until Q is empty.

(iv) The distance and back reference fields of every vertex that is not in Q are correct.

(v) Dequeue the highest priority vertex into V.

(vi) Do steps 7-10 for each vertex W that are adjacent to V and in the priority queue.

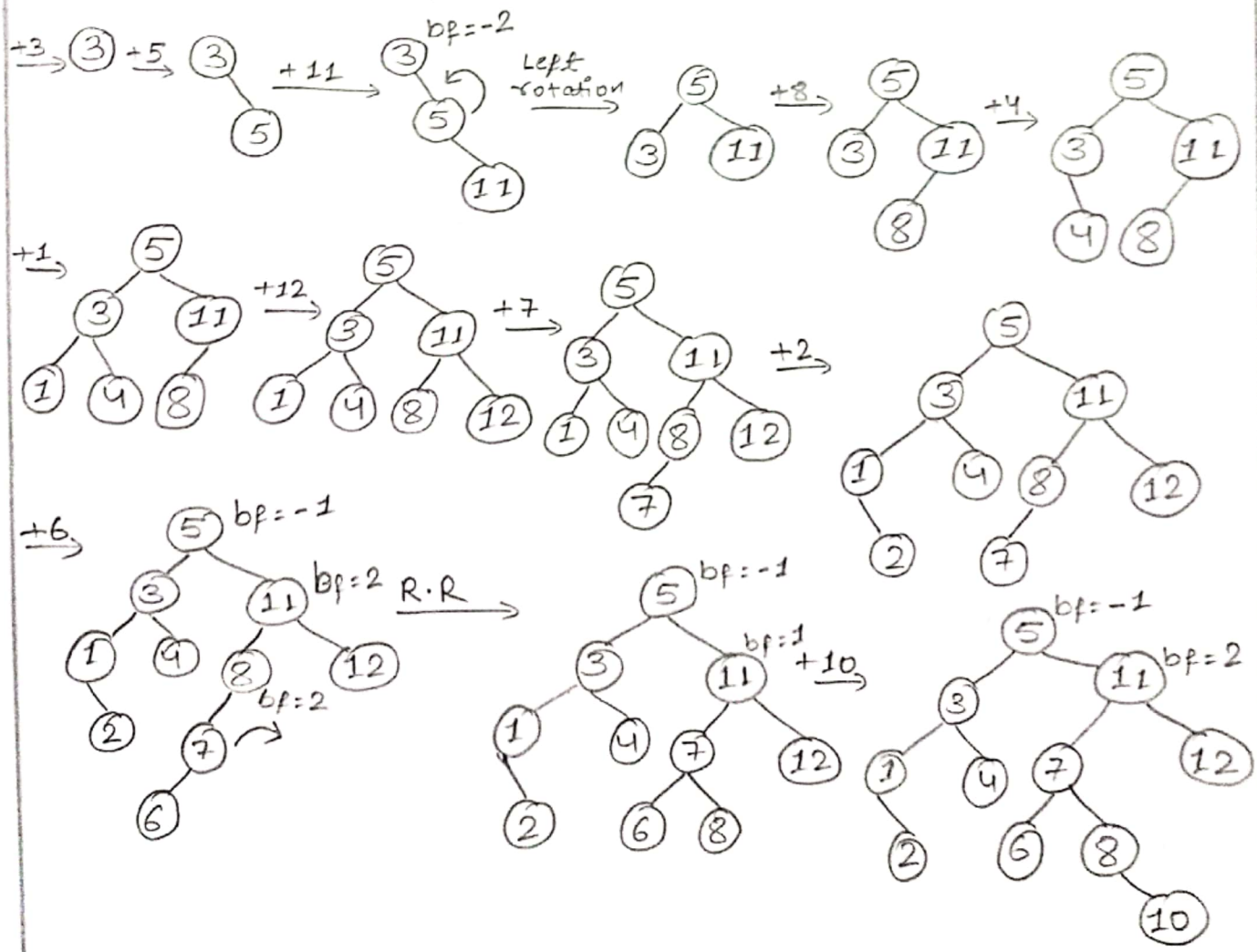
(vii) let S be the sum of the V's distance field plus the weight of the edge from V to W.

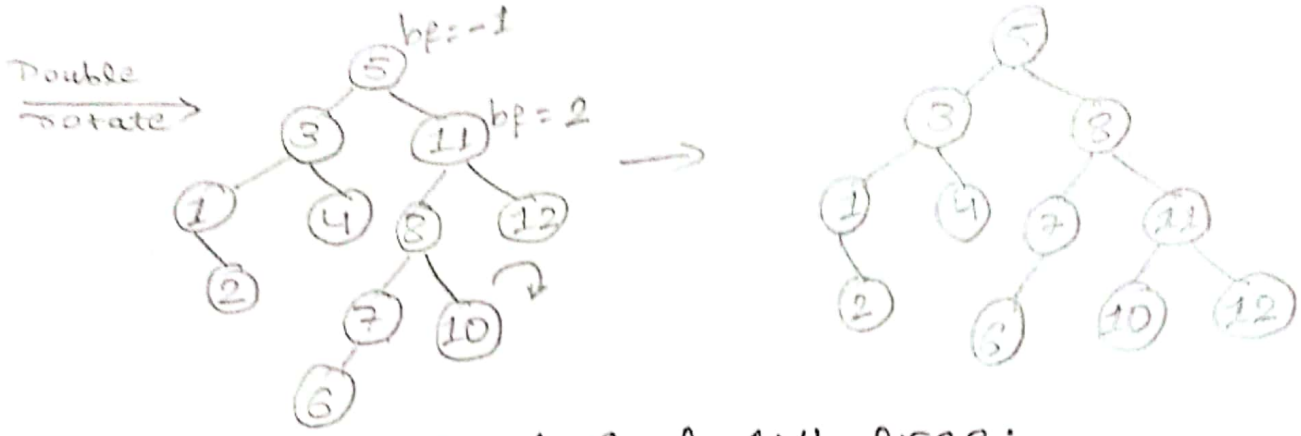
(viii) If S is less than W's distance field, do Step 9-10; Otherwise go back to step 3.

(ix) Assign S to W's distance field.

(x) Assign V to W's back reference field.

b) Draw AVL tree for the 3, 5, 11, 8, 4, 1, 12, 7, 2, 6 and 10



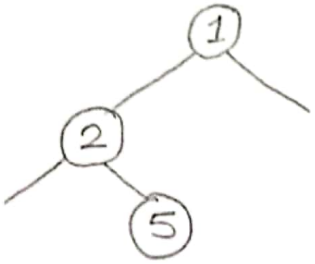


This is the required final AVL tree.

c) Construct the BST from the given data.

Preorder : 1, 2, 4, 8, 9, 10, 11, 5, 3, 6, 7 (DLR)

In-order : 8, 4, 10, 9, 11, 2, 5, 1, 6, 3, 7 (LDR)



⇒ Here, as we can see values that are greater than 1 (root node) on the left side which is against the properties of BST, we can conclude that the data presented for construction of BST is invalid or wrong.