Part A

Aim:
1. Design and analysis algorithms for Heapsort

Prerequisite: Any programming language

Outcome: Algorithms and their implementation

Theory:
A Binary Heap is a Complete Binary Tree where items are stored in a special order such that the value in a parent node is greater(or smaller) than the values in its two children nodes. The former is called max heap and the latter is called min-heap. The heap can be represented by a binary tree or array.

1. Build a max heap from the input data.
2. At this point, the largest item is stored at the root of the heap.
3. Replace it with the last item of the heap followed by reducing the size of the heap by 1
4. Finally, heapify the root of the tree.
5. Repeat step 2 while the size of the heap is greater than 1.

Procedure:
1. Design algorithm and find best, average, and worst-case complexity
2. Implement algorithms in any programming language.
3. Paste output

Practice Exercise:

S.no	Statement
1	Implement max heap and min-heap.
2	Sort the given numbers using heap sort.
3	Find the best, worst and average-case complexity for 2.

Instructions:
1. Design, analysis and implement the algorithms.
2. Paste the snapshot of the output in the input & output section.

Part B

Code

1)

Implementing min-heap

code:

```
def parent_node(pos):
    return pos//2

def left_child(pos):
    return pos*2+1

def right_child(pos):
    return pos*2+2

def leaf_node(heap,pos):
    global size
    if(pos<=size or pos//2+1>=size):
        return True
    return False
def heapify():
    pass
def swap(heap,a,b):
    heap[a],heap[b]=heap[b],heap[a]

def insert_node(heap,n):
    global pos
    heap[pos]=n
    pos+=1
    for i in range(pos//2-1,-1,-1):
        minheapify(heap,pos,i)

def delete_min(heap):

    global front,pos,n
    rem=heap[front]
    heap[front]=999999
    minheapify(heap,pos,front)
    pos-=1
    n-=1
    return rem


def minheapify(heap,pos,i):

    min=i
    left=left_child(i)
    right=right_child(i)
    if left<pos and heap[left]<heap[min]:
        min=left

    if right<pos and heap[right]<heap[min]:
        min=right
```

```
    if min!=i:
        heap[i],heap[min]=heap[min],heap[i]
        minheapify(heap,pos,min)


def display(heap):
    global n
    for i in range(0, n//2):
        print()
        if(heap[i]!=0 and heap[i]!=999999 and i<pos):
            print(" P : ", heap[i],end=" ")
            if(heap[i*2+1]!=0 and heap[i*2+1]!=999999):
                print("---LC : ",heap[2 * i+1 ],end="")
            if(heap[i*2+2]!=0 and heap[i*2+2]!=999999):
                print("\n       \\---RC : ",heap[2 * i + 2])

        else:
            return


def rootnode(heap):
    return heap[0]

pos=0
size=1
n=int(input('size of heap'))
front=0
heap=[0]*n
p=int(input('number of insertions '))

for i in range(p):
    insert_node(heap,int(input()))
print('heap after insertion of elements')
display(heap)
root=rootnode(heap)
print("root node of heap is ",root)
min_=delete_min(heap)
print('heap after deletion of minimum ',min_)
display(heap)
```

Max heap implementation

```
def parent_node(pos):
    return pos//2

def left_child(pos):
    return pos*2+1

def right_child(pos):
    return pos*2+2

def leaf_node(heap,pos):
    global size
    if(pos<=size or pos//2+1>=size):
        return True
    return False
def heapify():
    pass
def swap(heap,a,b):
    heap[a],heap[b]=heap[b],heap[a]

def insert_node(heap,n):
    global pos
    heap[pos]=n
    pos+=1
    for i in range(pos//2-1,-1,-1):
        maxheapify(heap,pos,i)

def delete_min(heap):

    global front,pos,n
    rem=heap[front]
    heap[front]=-999999
    maxheapify(heap,pos,front)
    pos-=1
    n-=1
    return rem


def maxheapify(heap,pos,i):

    max=i
    left=left_child(i)
```

```
    right=right_child(i)
    if left<pos and heap[left]>heap[max]:
        max=left

    if right<pos and heap[right]>heap[max]:
        max=right

    if max!=i:
        heap[i],heap[max]=heap[max],heap[i]
        maxheapify(heap,pos,max)
def display(heap):
    global n
    for i in range(0, n//2):
        print()
        if(heap[i]!=0 and heap[i]!=-999999 and i<pos):
            print(" P : ", heap[i],end=" ")
            if(heap[i*2+1]!=0 and heap[i*2+1]!=-999999):
                print("---LC : ",heap[2 * i+1 ],end="")
            if(heap[i*2+2]!=0 and heap[i*2+2]!=-999999):
                print("\n       \\---RC : ",heap[2 * i + 2])

        else:
            return

pos=0
size=1
n=int(input('size of heap'))
front=0
heap=[0]*n
p=int(input('number of insertions '))

for i in range(p):
    insert_node(heap,int(input()))

print('heap after insertion of elements')
display(heap)
min_=delete_min(heap)
print('heap after deletion of maximum ',min_)
display(heap)
```

2)heapsort :
In ascending order using max heap:

code:

```
def heapify(array,n,i):
    max=i
    left=i*2+1
```

```
    right=i*2+2
    if left<n and array[left]>array[max]:
        max=left
    if right<n and array[right]>array[max]:
        max=right
    if max!=i:
        array[i],array[max]=array[max],array[i]
        heapify(array,n,max)

def heap(array):
    n=len(array)
    for i in range(n//2-1,-1,-1):
        heapify(array,n,i)
    print("heap array ",array)
    for i in range(n-1,0,-1):
        array[i],array[0]=array[0],array[i]
        heapify(array,i,0)

array=list(map(int,input('enter tree elements ').split()))
print(array)
heap(array)
print("ascending sort of given array is ",array)
```

In descending order using min heap:

```
def heapify(array1,n,i):
    min=i
    left=2*i+1
    right=2*i+2
    if left<n and array1[left]<array1[min]:
        min=left

    if right<n and array1[right]<array1[min]:
        min=right

    if min!=i:
        array1[i],array1[min]=array1[min],array1[i]
        heapify(array1,n,min)

def heap(array1):
    n=len(array1)
    for i in range(n//2-1,-1,-1):
        heapify(array1,n,i)
    print("heap array ",array1)
    #heap sort
```

```
    for i in range(n-1,0,-1):
        array1[i],array1[0]=array1[0],array1[i]
        heapify(array1,i,0)

array1=list(map(int,input('enter tree elements ').split()))
print(array1)
heap(array1)
print("ascending sort of given array is ",array1)
```

Input & Output:

1)minheap implementation

```
PS E:\books and pdfs\sem4 pdfs\DAA lab\week5> python .\minheap_implementation.py
size of heap 20
number of insertions 8
5
3
7
9
1
2
6
8
heap after insertion of elements

 P :  1 ---LC :  3
       \---RC :  2

 P :  3 ---LC :  8
       \---RC :  5

 P :  2 ---LC :  7
       \---RC :  6

 P :  8 ---LC :  9
 P :  5
 P :  7
 P :  6
 P :  9
root node of heap is  1
heap after deletion of minimum  1

 P :  2 ---LC :  3
       \---RC :  6

 P :  3 ---LC :  8
       \---RC :  5

 P :  6 ---LC :  7
 P :  8 ---LC :  9
 P :  5
 P :  7
PS E:\books and pdfs\sem4 pdfs\DAA lab\week5>
```

maxheap implementation

```
PS E:\books and pdfs\sem4 pdfs\DAA lab\week5> python .\maxheap_implementation.py
size of heap 20
number of insertions 8
4
3
1
7
6
2
9
10
heap after insertion of elements

 P :  10 ---LC :  9
       \---RC :  7

 P :  9 ---LC :  6
       \---RC :  4

 P :  7 ---LC :  1
       \---RC :  2

 P :  6 ---LC :  3
 P :  4
 P :  1
 P :  2
 P :  3
heap after deletion of maximum  10

 P :  9 ---LC :  6
       \---RC :  7

 P :  6 ---LC :  3
       \---RC :  4

 P :  7 ---LC :  1
       \---RC :  2

 P :  3
 P :  4
 P :  1
 P :  2
PS E:\books and pdfs\sem4 pdfs\DAA lab\week5>
```

2)sorting:
maxheap sort

```
PS E:\books and pdfs\sem4 pdfs\DAA lab\week5> python .\max_heap_sort.py
enter tree elements 2 3 1 5 6 13 7
[2, 3, 1, 5, 6, 13, 7]
heap array  [13, 6, 7, 5, 3, 1, 2]
ascending sort of given array is  [1, 2, 3, 5, 6, 7, 13]
PS E:\books and pdfs\sem4 pdfs\DAA lab\week5>
```

minheap sort

```
PS E:\books and pdfs\sem4 pdfs\DAA lab\week5> python .\min_heap_sort.py
enter tree elements 2 3 1 5 6 13 7
[2, 3, 1, 5, 6, 13, 7]
heap array  [1, 3, 2, 5, 6, 13, 7]
ascending sort of given array is  [13, 7, 6, 5, 3, 2, 1]
PS E:\books and pdfs\sem4 pdfs\DAA lab\week5>
```

Best , average, worst case complexities of Heap sort:

Time complexity:

Time complexity of heapify = **O(log n)**
Therefore time complexity for 1 insertion/ deletion would be **O(log n)** as only 1 heapify is required

So time complexity of n insertions or deletions would be **O(n log n)**

Time complexity of heapify is **O(log n)**. Time complexity of create and build heap is O(n) and overall time complexity of Heap Sort is **O(n*log n)**
Best, average and worst case time complexity is independent of distribution of data ,i.e **O(nlog n)**

best case : O(n log n)

average case :O(n log n)

worst case :O(n log n)

Space complexity:

O(1) (heapsort uses O(1) auxiliary space (since it is an in-place sort))

Observation & Learning:

I have observed :
i)The consistent performance of heap sort i.e.(it performs equally well in best, average, and worst-case scenarios)
iii) Uses only less memory as it uses same datastructure.(requires a constant space for sorting a list.)
iv)Heap sort algorithm has limited uses because Quicksort and Mergesort are better in practice.

I have learned
i) to implement max heap and min heap algorithms in python
ii) to implement heap sort using min heap and max heap to obtain descending and ascending orders respectively

Conclusion:
I have successfully implemented min heap, max heap and heap sort in python language.

Questions:

1. Is Heap stable sorting?
2. Is Heap internal sorting?
3. Is Heap in-place sorting?

Answers:

1)Heap sort is not a Stable sort, because operations in the heap can change the relative order of equivalent keys.(requires a constant space for sorting a list)

2)Heap sort is internal sorting as data to be sorted is small enough to all be held in the main memory, i.e. data sorting process that takes place entirely within the main memory of a computer.

3) Heap sort is inplace sorting as it uses same datastructure for sorting.