Operating System Scheduler

Group 08

Outlook…

- According to the presented question the algorithms generally used in operation system scheduling,
  - First-Come, First –Server(FCFS)Scheduling
  - Round Robin Scheduling

Only using these two strategies are not appropriate. Thus,

- Shortest-Job-Next Scheduling –efficiency
- Priority scheduling

Approach to the problem…

- Our approach to the problem is to use a priority queue based heap which will take priority and burst time as its inputs.
- Here the programs with max priorities will have preference over programs with min priorities
- Also with same priority, shortest jobs have preference over the other jobs with same priority

Priority Queue

- A priority queue is an abstract data type which is like a regular queue or stack data structure, but where additionally each element has a "priority" associated with it. In a priority queue, an element with high priority is served before an element with low priority.
- Queue operations
  - insert
  - Delete_min

Heap

- A Heap is a special Tree-based data structure in which the tree is a complete binary tree.

- **Min-Heap**: In a Min-Heap the key present at the root node must be minimum among the keys present at all of it's children. The same property must be recursively true for all sub-trees in that Binary Tree.
- Heap operations
  - build_heap
  - Min_heapify

Main function

```
int main()
{
    Queue Q;
    CreateQueue(&Q);

    char name[20];
    int  priority , no = 0 , i = 1;
    float time , waitingT = 0 , turnaroundT = 0 , totturnT = 0, totwaitT = 0;

   printf("\t\t\tinput number of processes : ");
   scanf(" %d",&no);

   QueueElement P[no],sort;
```

```
PROCESS SHEDULER
----------------------------------------------------------------------

input number of processes : 9

        Process-1
        Process name : P1
        Process priority : 9
        Burst Time (s) : 15

        Process-2
        Process name : P2
        Process priority : 5
        Burst Time (s) : 18

        Process-3
        Process name : P3
        Process priority : 9
        Burst Time (s) : 18

        Process-4
        Process name :
```

```c
while(i!=no+1)
{
    printf("\n\t\t\t\tProcess-%d\n\t\t\t\tProcess name : ",i);
    scanf(" %s",P[i].name);

    printf("\t\t\t\tProcess priority : ");
    scanf("%d",&P[i].priority);
    if(P[i].priority<0 || P[i].priority==0)
    {
        printf("\t\t\tpriority number can not be negative vale or zero!!\n");
        continue;
    }
    printf("\t\t\t\tBurst Time (s) : ");
    scanf("%f",&P[i].time);
    Append(&Q,P[i]);
    i++;
}
```

```
printf("\n\n\tNumber \tProcess Name \tPriority \tBurst time (s)\t\tWaiting Time (s) \tTurnaround time (s)\n");

for(int i = 1 ; i<=no ;i++)
{
    sort = delete_min(&Q);
    turnaroundT = waitingT+sort.time;
    totturnT += turnaroundT;
    printf("\t  %d)\t%s\t\t%d\t\t%.2f\t\t\t%.2f\t\t\t%.2f\n",i,sort.name,sort.priority,sort.time,waitingT,turnaroundT);
    totwaitT +=waitingT;
    waitingT += sort.time;
}
printf("\t-----------------------------------------------------------------------------------------------------\n");
printf("\t\tTotal\t\t\t\t\t\t\t%.2f\t\t\t%.2f\n",totwaitT,totturnT);
printf("\t-----------------------------------------------------------------------------------------------------\n");
printf("\n\n\t\tAverage waiting time (s) = %.2f \t\tAverage turnaround time (s) = %.2f\n\n",totwaitT/no,totturnT/no);
```

Storing priority and burst time

```c
typedef struct process
{
    float time;
    char name[10];
    int priority;
}QueueElement;

typedef struct queue{
    int count;
    int front;
    int rear;
    QueueElement items[MAXQUEUE];
}Queue;
```

P1	**9**	**15**	**P2**	**5**	**18**

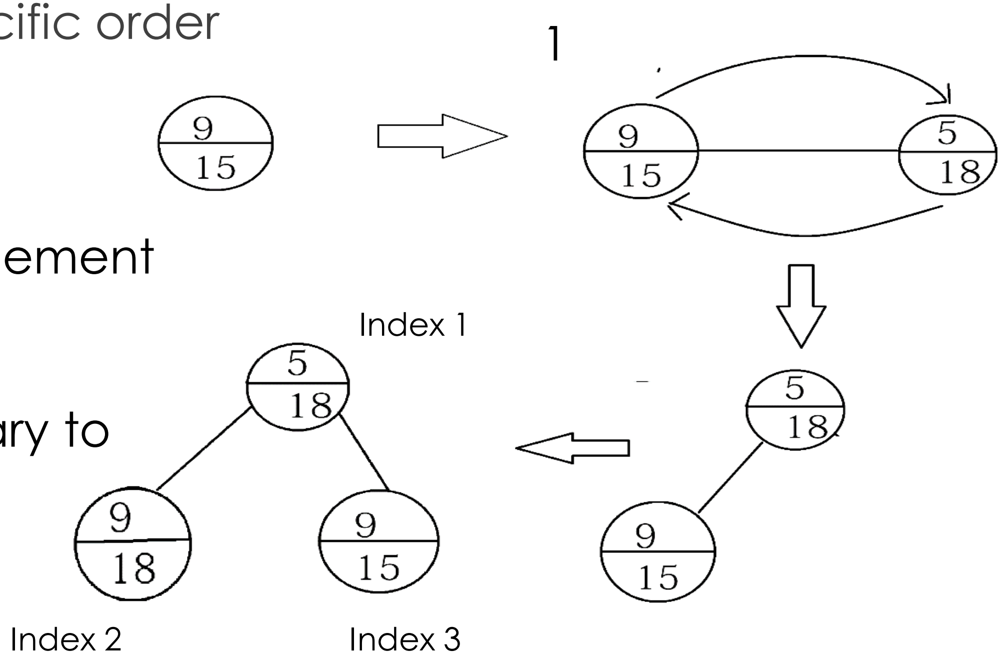
1st index

2nd index

Use two structures to store details

Building the heap

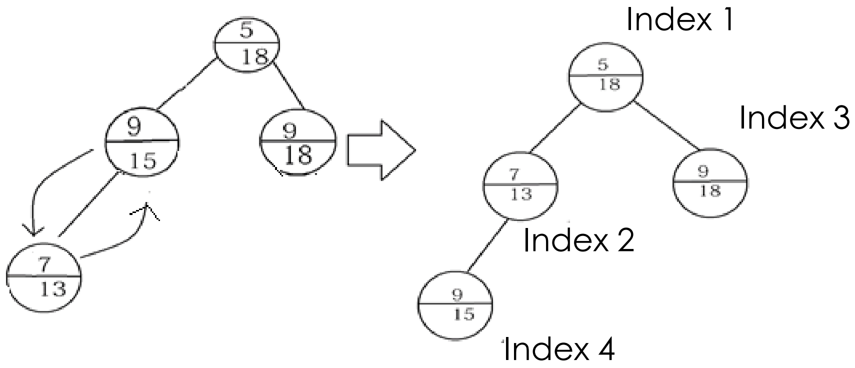
- Creating the heap using given details of the process
- Here we use a min heap to store priorities in a specific order



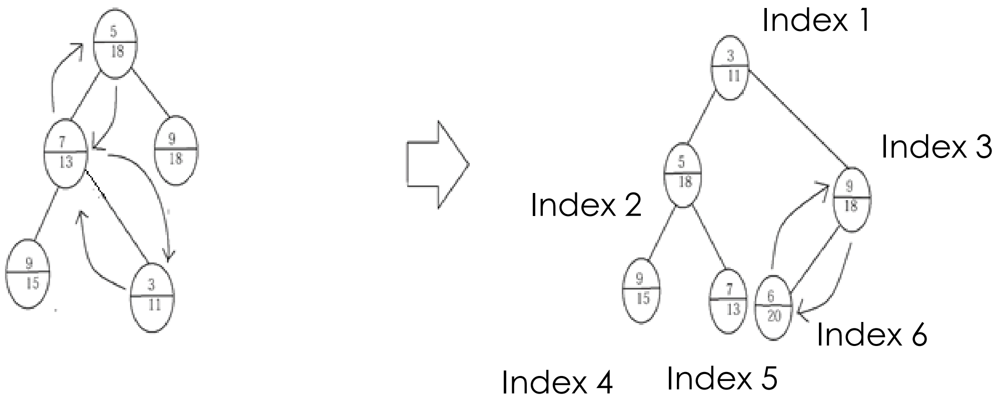
Inserting 1st element and 2nd element

Swapping the root, left and right child when necessary to create the min heap

2

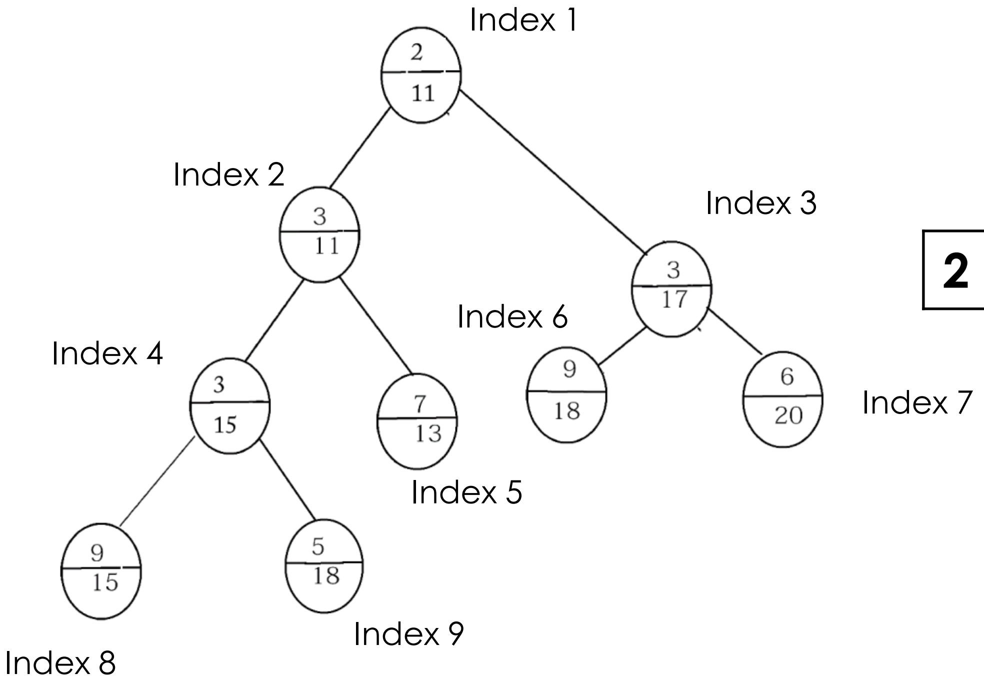


3



Newly inserted detail is checked against its root for similar priority and similarly goes up checking its root and right

Final min heap



2	3	3	3	7	9	6	9	5

Code for building heap

```
void build_min_heap(Queue *q)
{
    int i;
    for(i=q->rear/2; i>=1; i--)
    {
        min_heapify(q, i);
    }
}
```

```
//function to get right child of a node of a tree
int get_right_child(Queue *q, int index)
{
    if((((2*index)+1) < MAXQUEUE) && (index >= 1))
    return (2*index)+1;
    return -1;
}

//function to get left child of a node of a tree
int get_left_child(Queue *q, int index)
{
    if(((2*index) < MAXQUEUE) && (index >= 1))
    return 2*index;
```

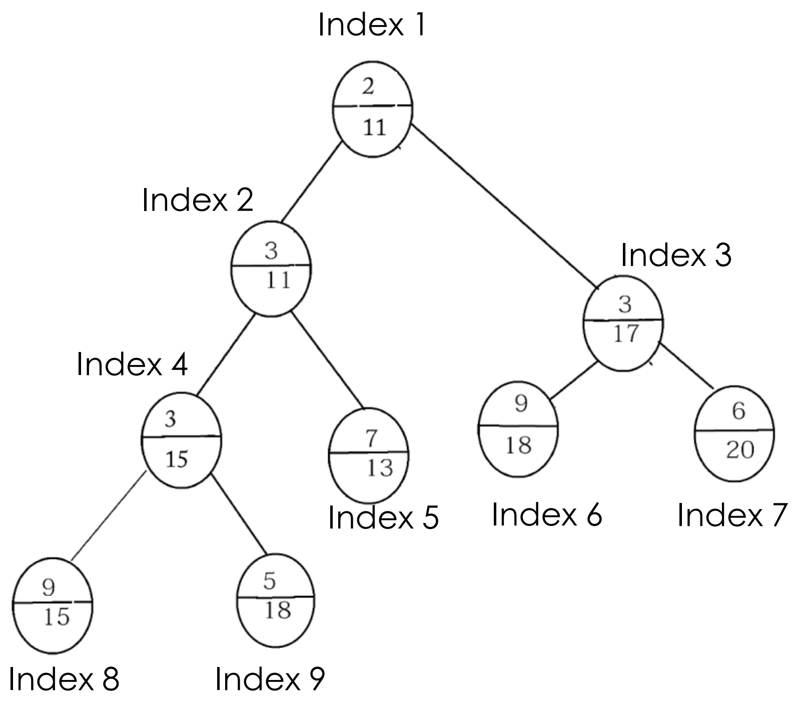
```
void heapSort(Queue *q){
    build_min_heap(q);
}
```

Extracting elements according to priority and burst time

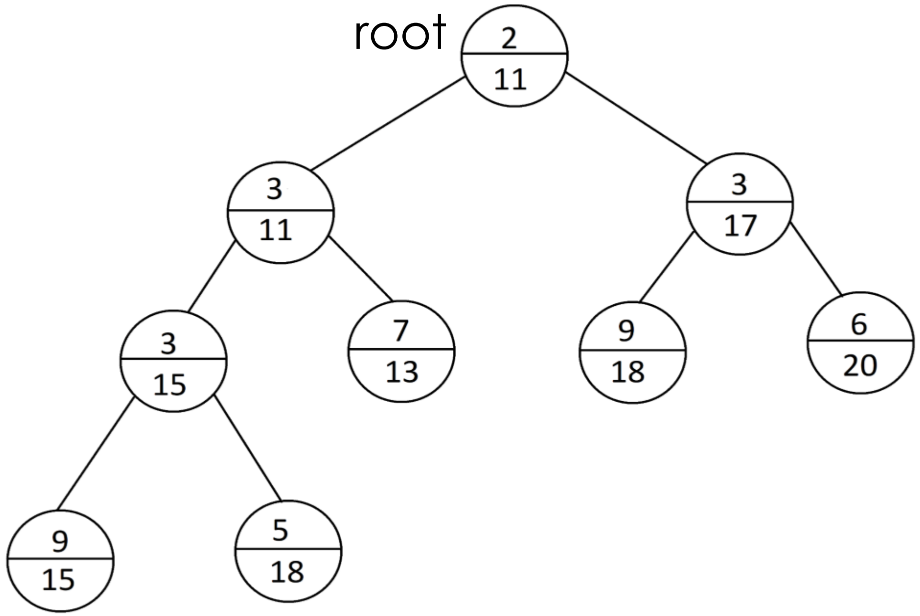
- After insertion of all the elements most least number(maximum priority) stored in the index one of the heap.(root of the heap)

After insertion of all the elements-

2	**3**	**3**	**3**	**7**	**9**	**6**	**9**	**5**



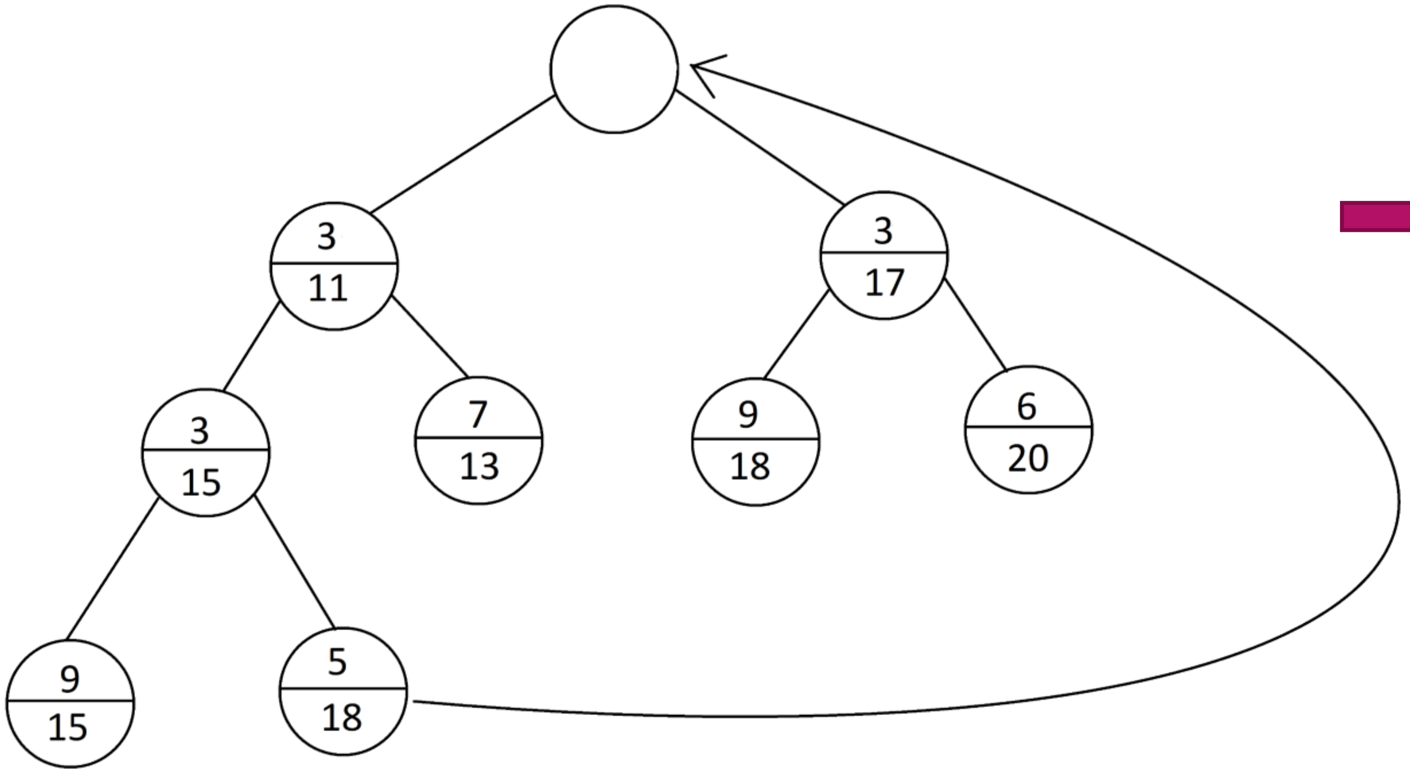
- To extract the highest priority number we call delete_min function.




```
QueueElement delete_min(Queue *q)
{
    QueueElement minimum = q->items[q->front];
    q->items[q->front] = q->items[q->rear];
    q->rear--;
    q->count--;
    min_heapify(q, q->front);

    return minimum;
}
```

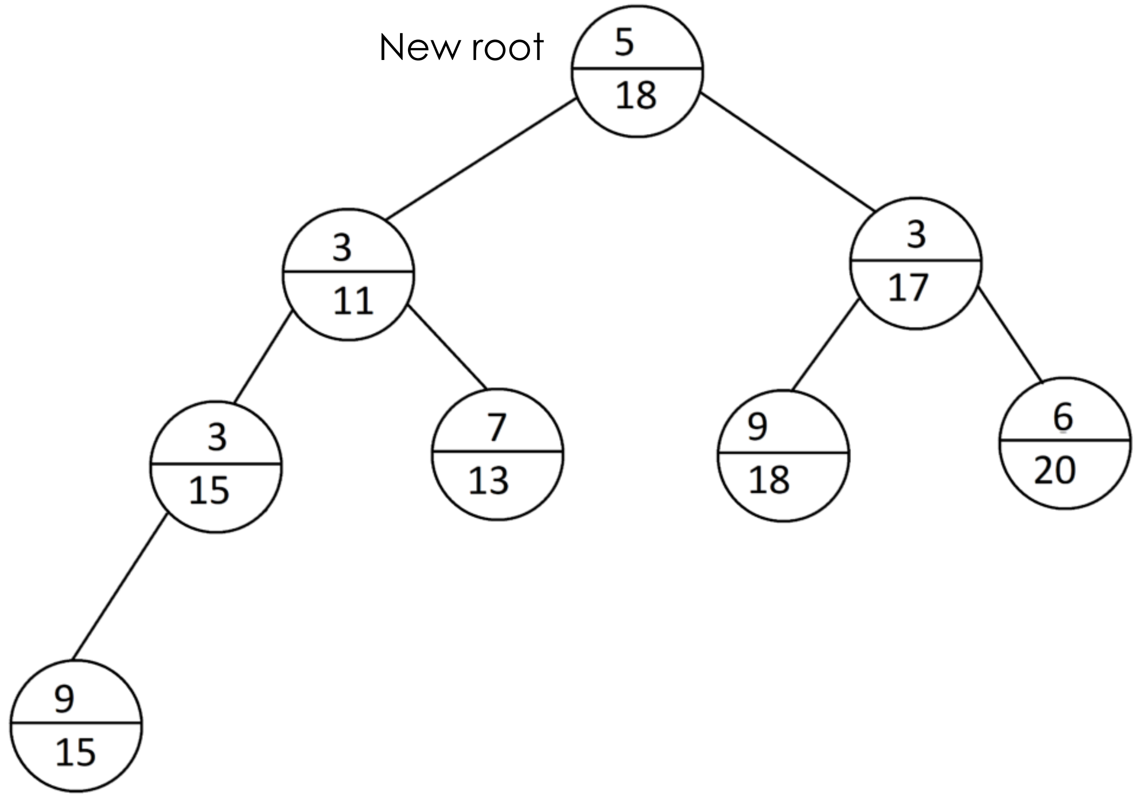
- Assign root element to the variable minimum and return it.

- After returning of the root element we should assign last index value to the root.




```c
QueueElement delete_min(Queue *q)
{
    QueueElement minimum = q->items[q->front];
    q->items[q->front] = q->items[q->rear];
    q->rear--;
    q->count--;
    min_heapify(q, q->front);

    return minimum;
}
```

- Now the order has violated. Should sort again.



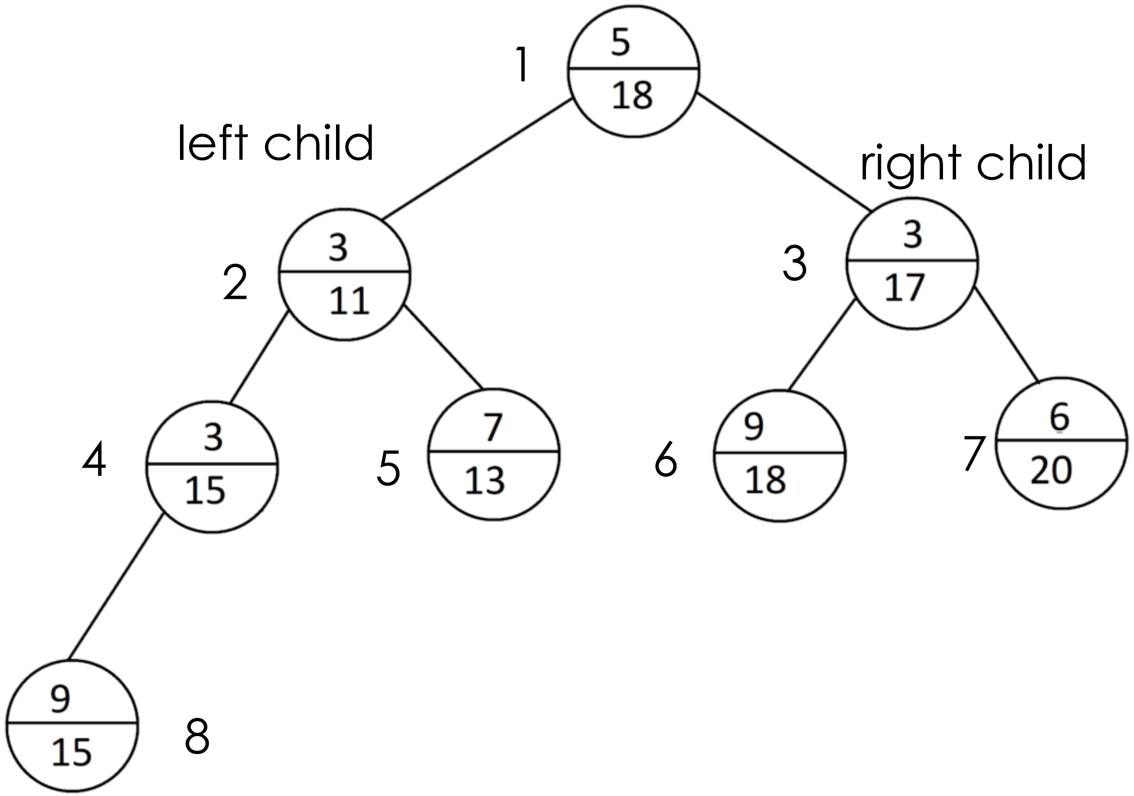

```
QueueElement delete_min(Queue *q)
{
    QueueElement minimum = q->items[q->front];
    q->items[q->front] = q->items[q->rear];
    q->rear--;
    q->count--;
    min_heapify(q, q->front);

    return minimum;
}
```

- Call min_heapify function to sort the elements in correct order.

Code for sorting

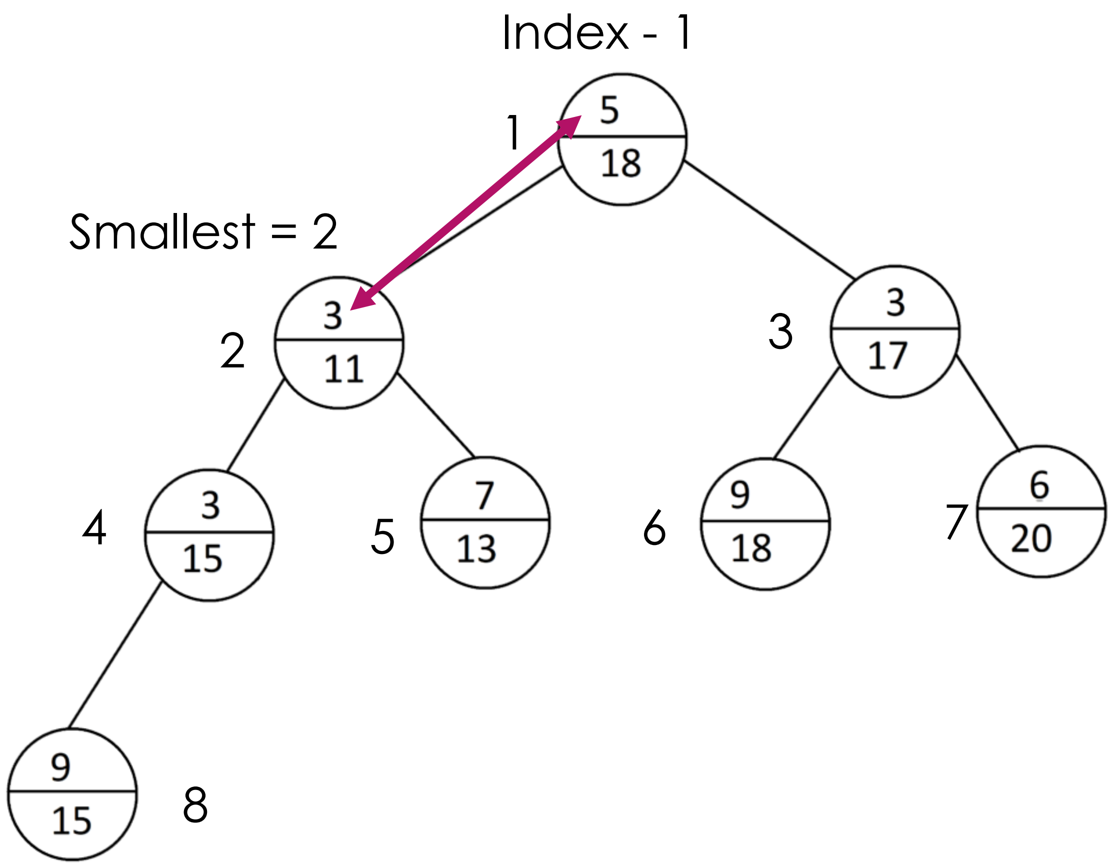
```
void min_heapify(Queue *q, int index)
{
    int indexLeftChild = leftChild(q, index);
    int indexRightChild = rightChild(q, index);
  // finding smallest among index, left child and right child
    int smallest = index;
    if ((indexLeftChild <= q->rear) && (indexLeftChild>0))
    {
        if (q->items[indexLeftChild].priority <= q->items[smallest].priority)
        {
            if (q->items[indexLeftChild].priority == q->items[smallest].priority)
                if (q->items[indexLeftChild].time < q->items[smallest].time)
                    smallest = indexLeftChild;
            else
                smallest = indexLeftChild;
        }
    }
    if ((indexRightChild <= q->rear && (indexRightChild>0)))
    {
        if (q->items[indexRightChild].priority <= q->items[smallest].priority)
        {
            if (q->items[indexRightChild].priority == q->items[smallest].priority)
                if (q->items[indexRightChild].time < q->items[smallest].time)
                    smallest = indexRightChild;
        }
            else
                smallest = indexRightChild;
    }
  // smallest is not the node, node is not a heap
    if (smallest != index)
    {
        swap(&q->items[index], &q->items[smallest]);
        min_heapify(q, smallest);
    }
}
```

Need to access the left child and right child



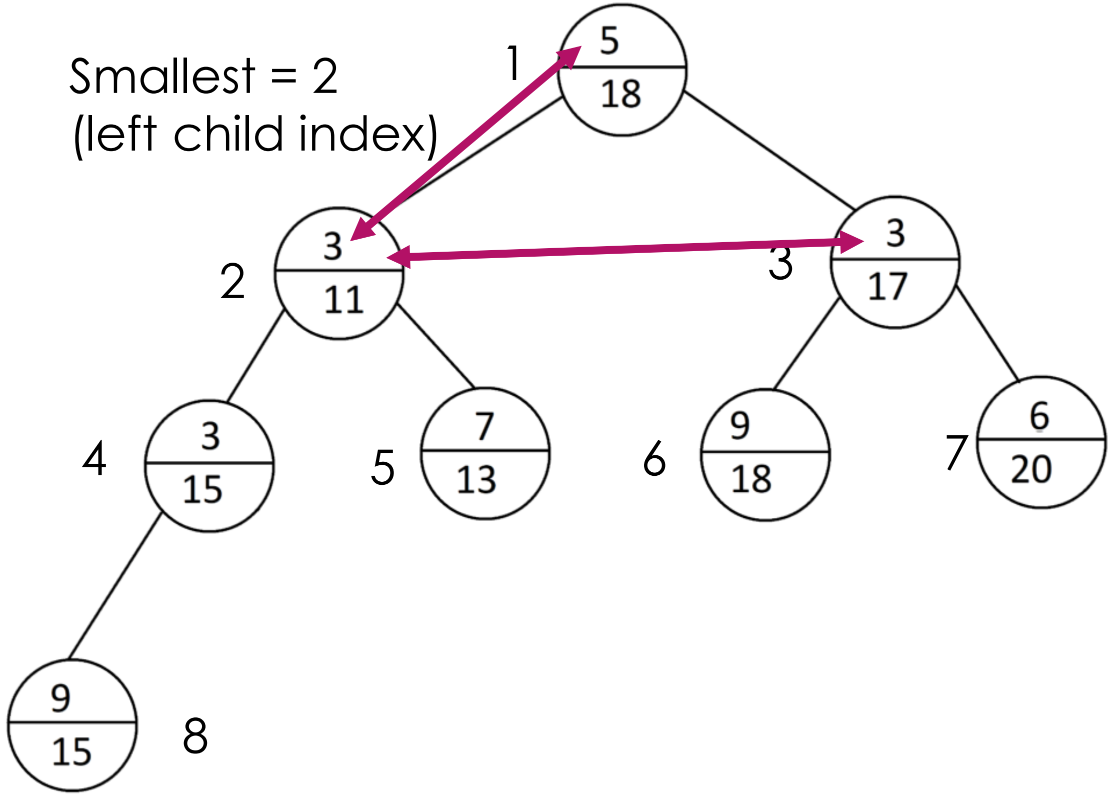

```c
void min_heapify(Queue *q, int index)
{
    int indexLeftChild = leftChild(q, index);
    int indexRightChild = rightChild(q, index);
```

```c
//function to get right child of a node of a tree
int rightChild(Queue *q, int index)
{
    if((((2*index)+1) < MAXQUEUE) && (index >= 1))
    return (2*index)+1;
    return -1;
}

//function to get left child of a node of a tree
int leftChild(Queue *q, int index)
 {
    if(((2*index) < MAXQUEUE) && (index >= 1))
    return 2*index;
    return -1;
}
```

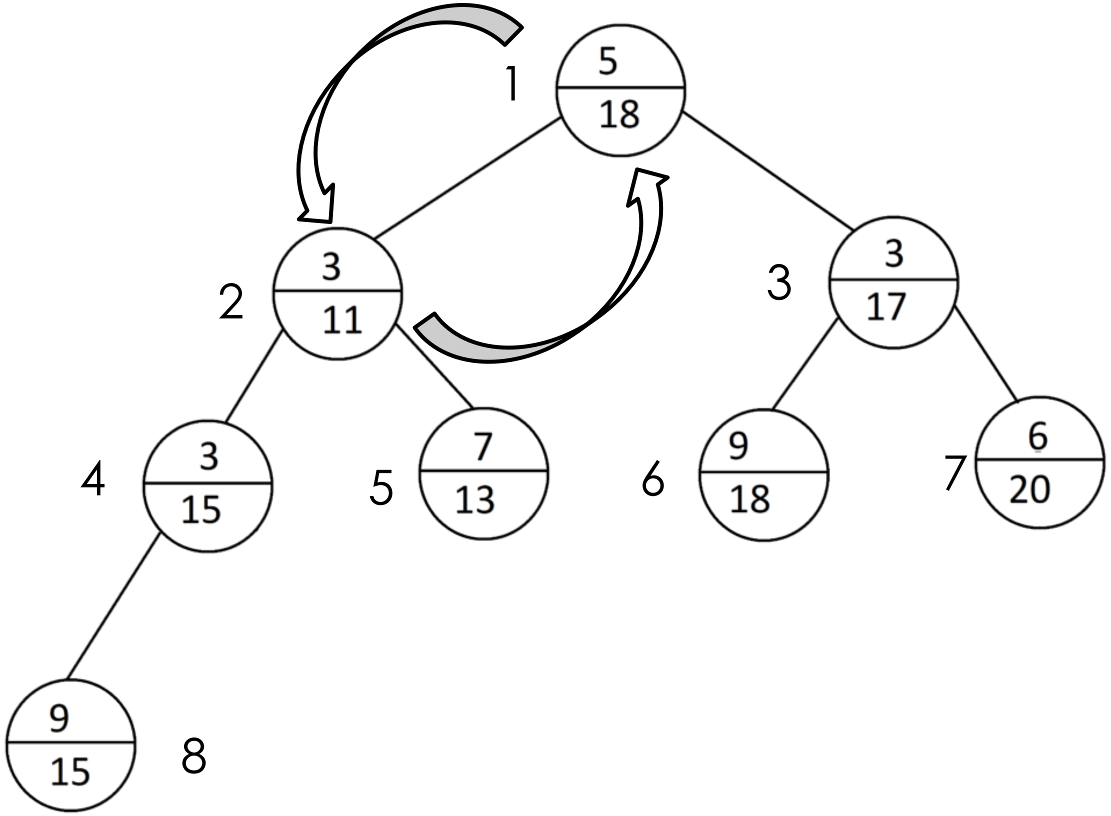



```
int smallest = index;

if ((indexLeftChild <= q->rear) && (indexLeftChild>0))
{
    if (q->items[indexLeftChild].priority <= q->items[smallest].priority)
    {
        if (q->items[indexLeftChild].priority == q->items[smallest].priority)
        {
            if (q->items[indexLeftChild].time < q->items[smallest].time)
            {
                smallest = indexLeftChild;
            }
        }
        else
        {
            smallest = indexLeftChild;
        }
    }
}
```

Smallest = 2
(left child index)




```
if ((indexRightChild <= q->rear && (indexRightChild>0)))
{
    if (q->items[indexRightChild].priority <= q->items[smallest].priority)
    {
        if (q->items[indexRightChild].priority == q->items[smallest].priority)
        {
            if (q->items[indexRightChild].time < q->items[smallest].time)
            {
                smallest = indexRightChild;
            }
        }
        else
        {
            smallest = indexRightChild;
        }
    }
}
```

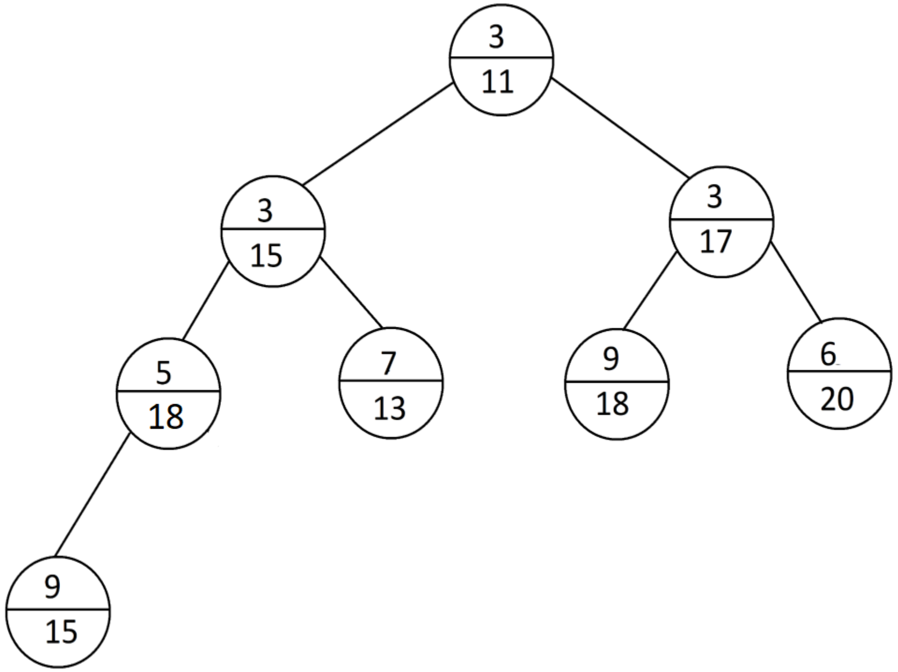



```c
// smallest is not the node, node is not a heap
if (smallest != index)
{
    swap(&q->items[index], &q->items[smallest]);
    min_heapify(q, smallest);
}
}
```

Smallest = 2 (left child index) $\neq$ Index = 1 (root)

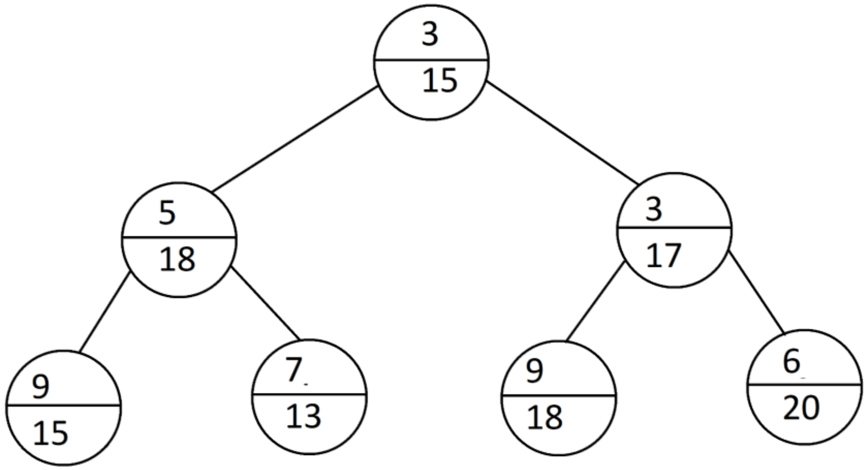
Should swap the index one and two

Heap after re arrange



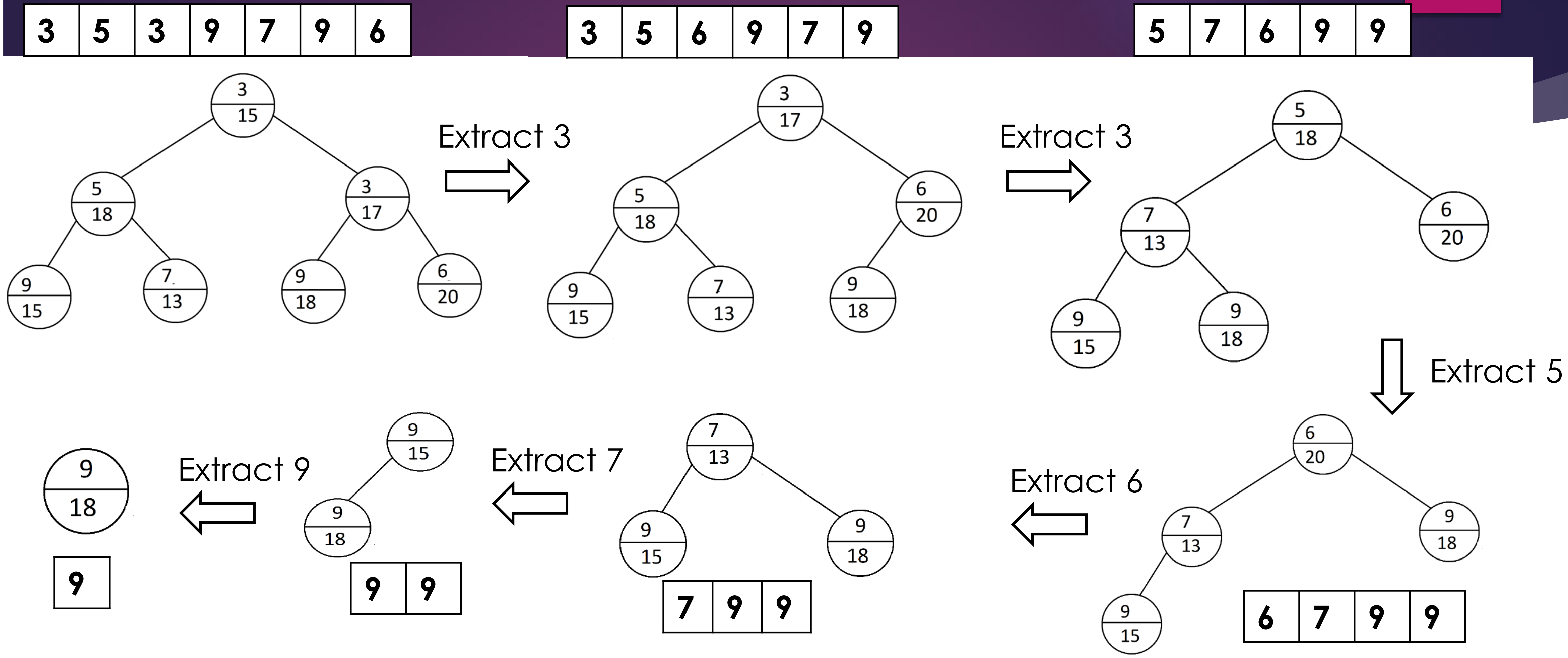
3	**3**	**3**	**5**	**7**	**9**	**6**	**9**

Extract the root



Heap after sort again

3	**3**	**3**	**9**	**7**	**9**	**6**



2	3	3	3	5	6	7	9	9
11	11	15	17	18	20	13	15	18

Execution

- Different priorities, different burst time
- Different priorities, same burst time
- Same priority, different burst time
- Same priority, same burst time

  - Turnaround time = waiting time + burst time

Excecution

Number	Process Name	Priority	Burst time (s)	Waiting Time (s)	Turnaround time (s)
1)	P8	2	11.00	0.00	11.00
2)	P5	3	11.00	11.00	22.00
3)	P9	3	15.00	22.00	37.00
4)	P7	3	17.00	37.00	54.00
5)	P2	5	18.00	54.00	72.00
6)	P6	6	20.00	72.00	92.00
7)	P4	7	13.00	92.00	105.00
8)	P1	9	15.00	105.00	120.00
9)	P3	9	18.00	120.00	138.00
	Total			513.00	651.00

Average waiting time (s) = 57.00

Average turnaround time (s) = 72.33

Future enhancement

- Use round robin method
  - By using a round robin scheduling we take a specific time quantum and let the given process run for that particular time quantum. By using this each process gets an equal time period to execute.
  - A certain process can come with the high priority and also having a longer burst time at the same time, another process can arrive with a shorter burst time but a lower priority. In such case the shorter burst should wait for a longer period of time. If we can use round robin scheduling and also consider the priority parallel, the shortest process doesn't need to wait. This would be more efficient.

References

- Books

Abraham Silberschatz, Greg Gagne, and Peter Baer Galvin, "Operating System Concepts, Eighth Edition ", Chapter 5.

- Web sites

www.codesdope.com

geeksforgeeks.org

YouTube

Wikipedia

Q & A

Thank You!!!