Subject: Data structures using C
Topic: Circular Queue

LISNA THOMAS

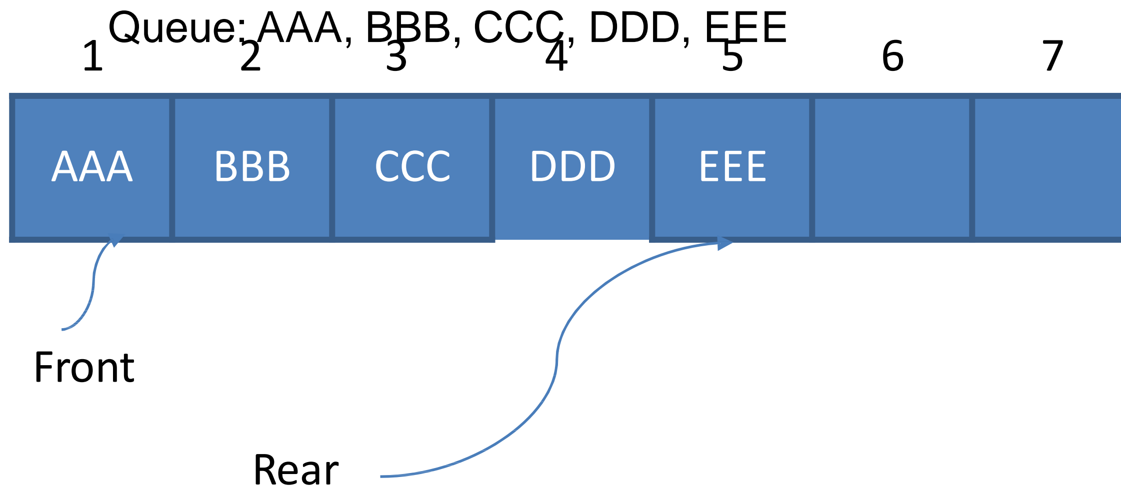
ACADEMIC YEAR:2020-21

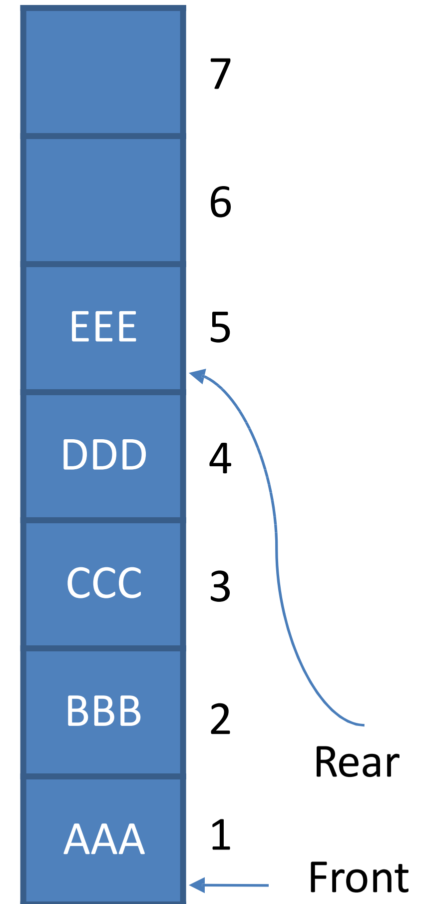
Circular Queue & Priority Queue

Queue (Linear Queue)

- It is a linear data structure consisting of list of items.
- In queue, data elements are added at one end, called ***the rear*** and removed from another end, called ***the front*** of the list.
- Two basic operations are associated with queue:
  1. "Insert" operation is used to insert an element into a queue.
  2. "Delete" operation is used to delete an element from a queue.
- FIFO list
- Example:

Queue: AAA, BBB, CCC, DDD, EEE





Drawback of Linear Queue

- Once the queue is full, even though few elements from the front are deleted and some occupied space is relieved, it is not possible to add anymore new elements, as the rear has already reached the Queue's rear most position.

Circular Queue

- This queue is not linear but circular.
- Its structure can be like the following figure:
- In circular queue, once the Queue is full the "First" element of the Queue becomes the "Rear" most element, if and only if the "Front" has moved forward. otherwise it will again be a "Queue overflow" state.

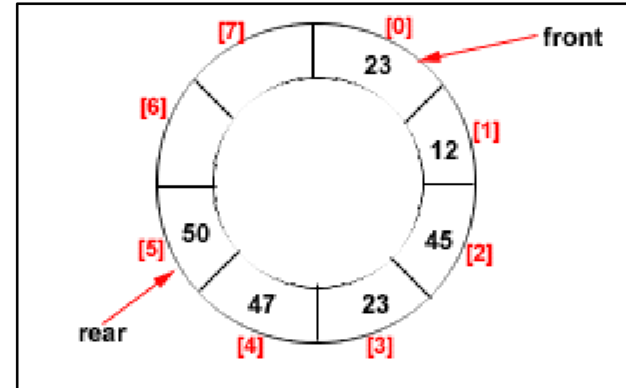


Figure: Circular Queue having Rear = 5 and Front = 0

Algorithms for Insert and Delete Operations in Circular Queue

For Insert Operation

Insert-Circular-Q(CQueue, Rear, Front, N, Item)

- CQueue is a circular queue where to store data.
- Rear represents the location in which the data element is to be inserted and Front represents the location from which the data element is to be removed.
- Here N is the maximum size of CQueue and finally, Item is the new item to be added.
- Initailly Rear = 0 and Front = 0.

1. If Front = 0 and Rear = 0 then Set Front := 1 and go to step 4.
2. If Front =1 and Rear = N or Front = Rear + 1

   then Print: “Circular Queue Overflow” and Return.
3. If Rear = N then Set Rear := 1 and go to step 5.
4. Set Rear := Rear + 1
5. Set CQueue [Rear] := Item.
6. Return

For Delete Operation

Delete-Circular-Q(CQueue, Front, Rear, Item)

Here, CQueue is the place where data are stored. Rear represents the location in which the data element is to be inserted and Front represents the location from which the data element is to be removed. Front element is assigned to Item. Initially, Front = 1.

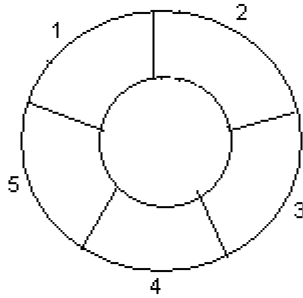
1. If Front = 0 then

    Print: “Circular Queue Underflow” and Return. /*..Delete without Insertion

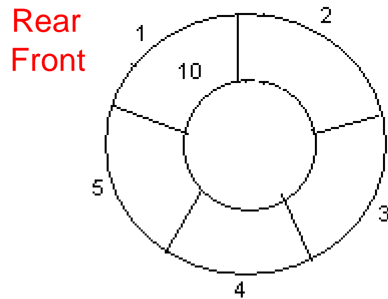
2. Set Item := CQueue [Front]
3. If Front = N then Set Front = 1 and Return.
4. If Front = Rear then Set Front = 0 and Rear = 0 and Return.
5. Set Front := Front + 1
6. Return.

Example: Consider the following circular queue with N = 5.

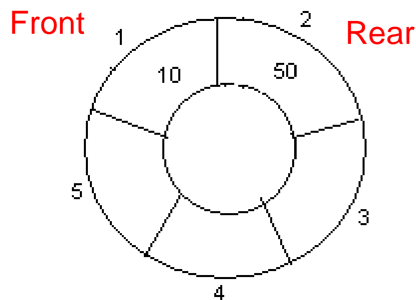
1. Initially, Rear = 0, Front = 0.



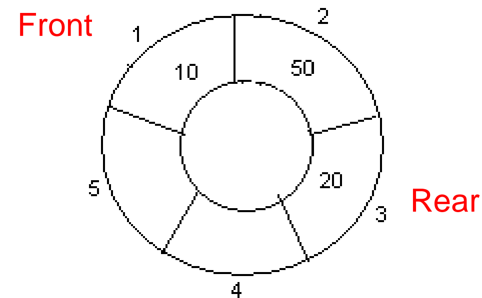
2. Insert 10, Rear = 1, Front = 1.



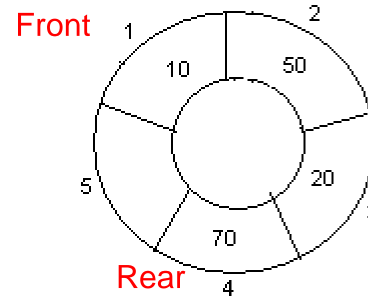
3. Insert 50, Rear = 2, Front = 1.



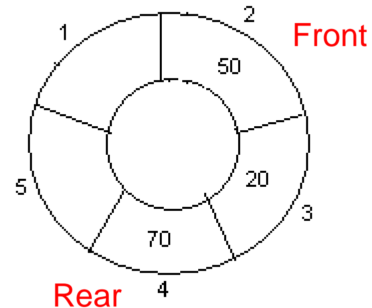
4. Insert 20, Rear = 3, Front = 0.



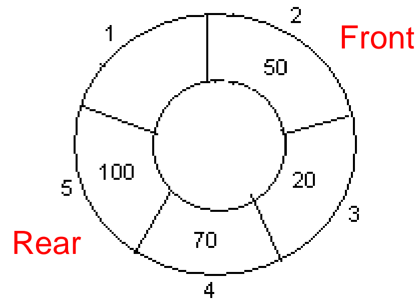
5. Insert 70, Rear = 4, Front = 1.



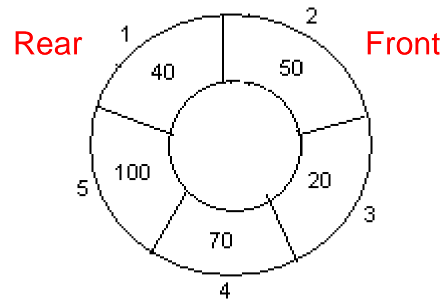
6. Delete front, Rear = 4, Front = 2.



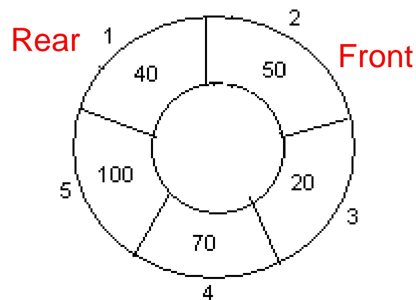
7. Insert 100, Rear = 5, Front = 2.



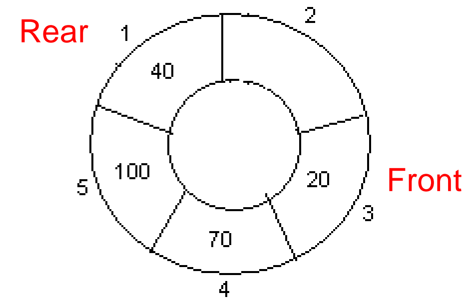
8. Insert 40, Rear = 1, Front = 2.



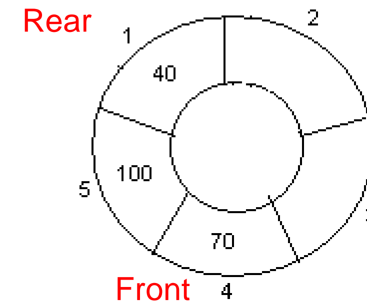
9. Insert 140, Rear = 1, Front = 2.
As Front = Rear + 1, so Queue overflow.



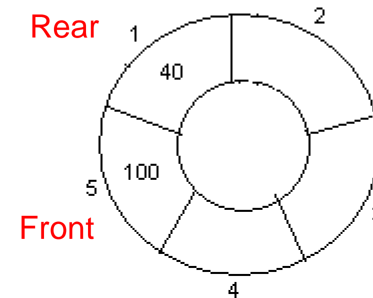
10. Delete front, Rear = 1, Front = 3.



11. Delete front, Rear = 1, Front = 4.



12. Delete front, Rear = 1, Front = 5.



Insertion

```
qfull() {
      if(front==(rear+1)%size)
            return 1;
      else
            return 0;
      }

int insert(int item)
{
   if(qfull())
       printf("circular q is full");
   else
   {
       if(front==-1)
             front=rear=0;
       else
              rear=(rear+1)%size;
              que[rear]=item;
      }
}
```

Deletion

```
int qempty()
{
    if(front==-1)
           return 1;
    else
           return 0;
}
int delete()
{
           int item;
         if(qempty())
                printf("queue is empty");
       else
       {
            item=que[front];
            if(front==rear)  {
                  front=rear=-1;
            }
            else
            front=(front+1)%size;
            printf("the deleted item is%d ",item);
}
```

Display

```
void display()
{
    int i;
    if(qempty())
    {
            printf("queue is empty");
            return;
    }
    i=front;
    while(i!=rear)
    {
        printf(“%d”,que[i]);
        i=(i+1)%size;
    }
        Printf(“%d”,que[i]);
}
```

Priority Queue

Priority Queue

A Priority Queue – a different kind of queue.
Similar to a regular queue:

insert in rear,
remove from front.

Items in priority queue are **ordered** by **some key**
Item with the lowest key / highest key is always at the front from where they are removed.

Items **then** „inserted" in „proper" position

Idea behind the Priority Queue is simple:
Is a queue
But the items are **ordered** by a **key**.
Implies your 'position' in the queue may be changed by the arrival of a new item.

Applications of Priority Queues

Many, many applications.

Scheduling queue**s** for a processor, print queues, transmit queues, backlogs, etc.….

This means this item will be in the **front** of queue

“Obtained” via a **remove**().

Note: a priority queue is no longer FIFO!

You will still **remove** from front of queue, but **insertions** are governed by a priority.

Priority Queues: Access

remove()

So, the first item has priority and can be retrieved (removed) **quickly** and returned to calling environment.

Hence, „remove()" is easy (and will take O(1) time)

insert()

But, we want to **insert** quickly. Must go into proper position.

For our purposes here: Implementing data structure: **array**;

- **slow to insert**(), **but** for
  - ❿ small number of items in the pqueue, **and**
  - ❿ where insertion speed is not critical,
- this is the simplest and best approach.

Priority Queues

Operations performed on priority queues

1) Find an element,
2) Insert a new element
3) Delete an element, etc.

Two kinds of (Min, Max) priority queues exist:

- In a Min priority queue, find/delete operation finds/deletes the element with minimum priority
- In a Max priority queue, find/delete operation finds/deletes the element with maximum priority
- Two or more elements can have the same priority

Implementation of Priority Queues

Implemented using **heaps** and **leftist trees**

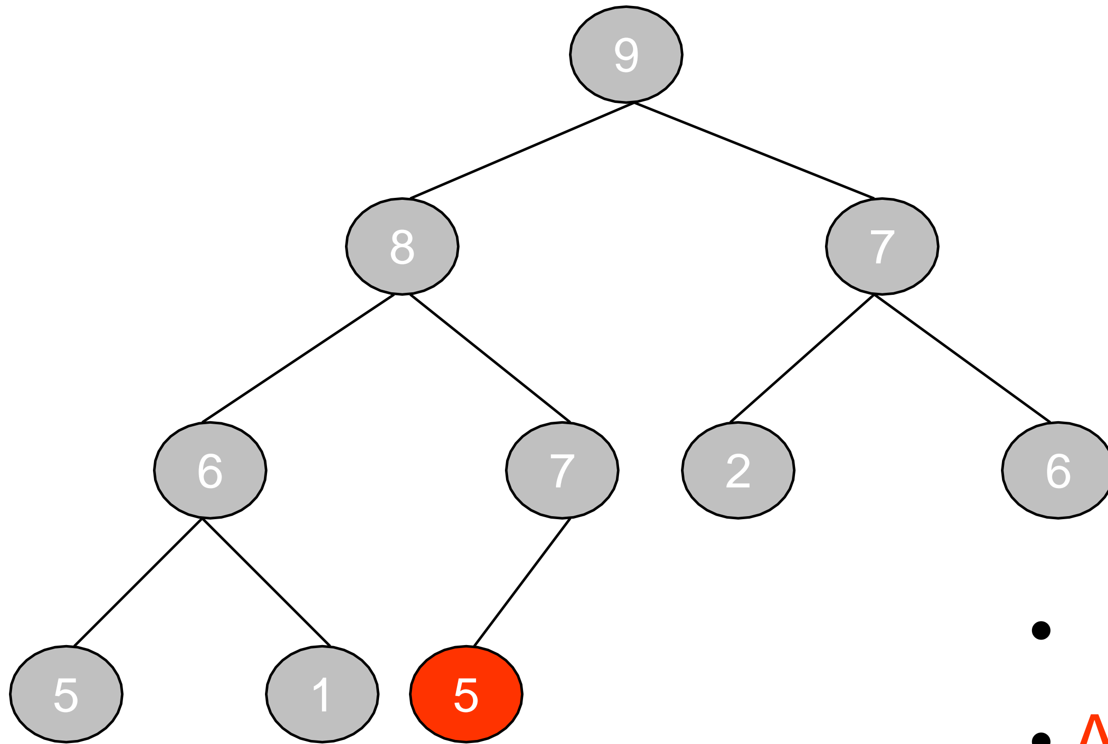
- **Heap** is a complete binary tree that is efficiently stored using the array-based representation
- **Leftist tree** is a linked data structure suitable for the implementation of a priority queue

Heap Operations

When *n* is the number of elements (heap size),

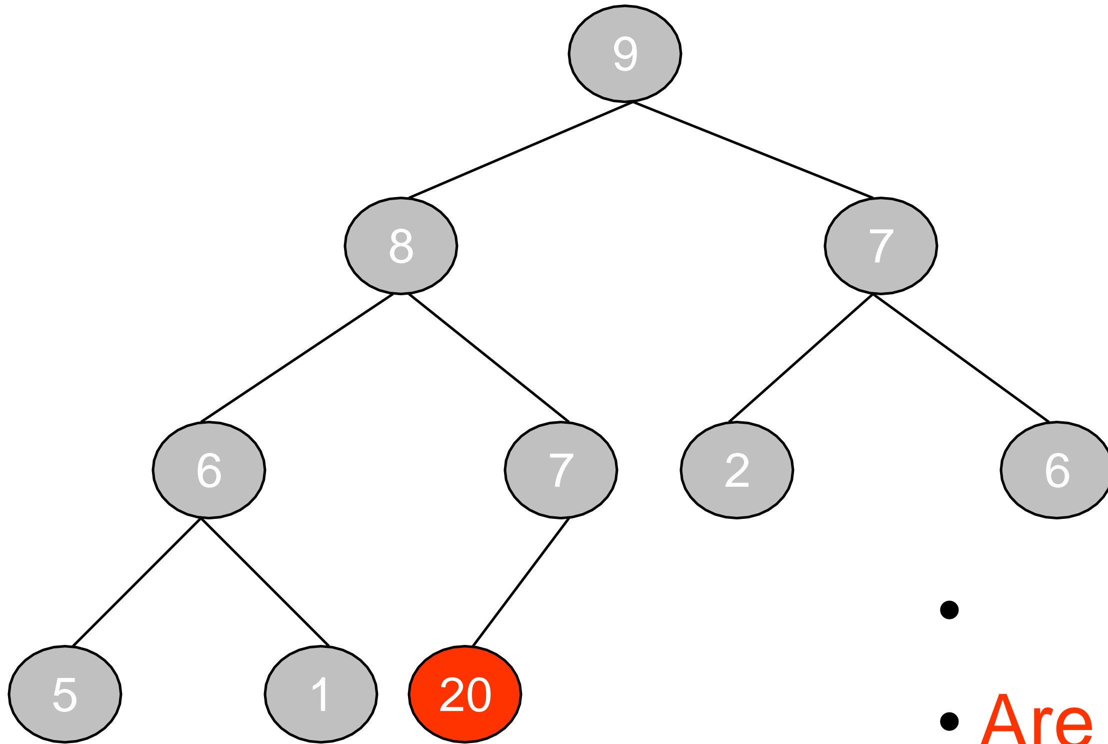
- Insertion → $O(\log_2 n)$
- Deletion → $O(\log_2 n)$
- Initialization → $O(n)$

Insertion into a Max Heap



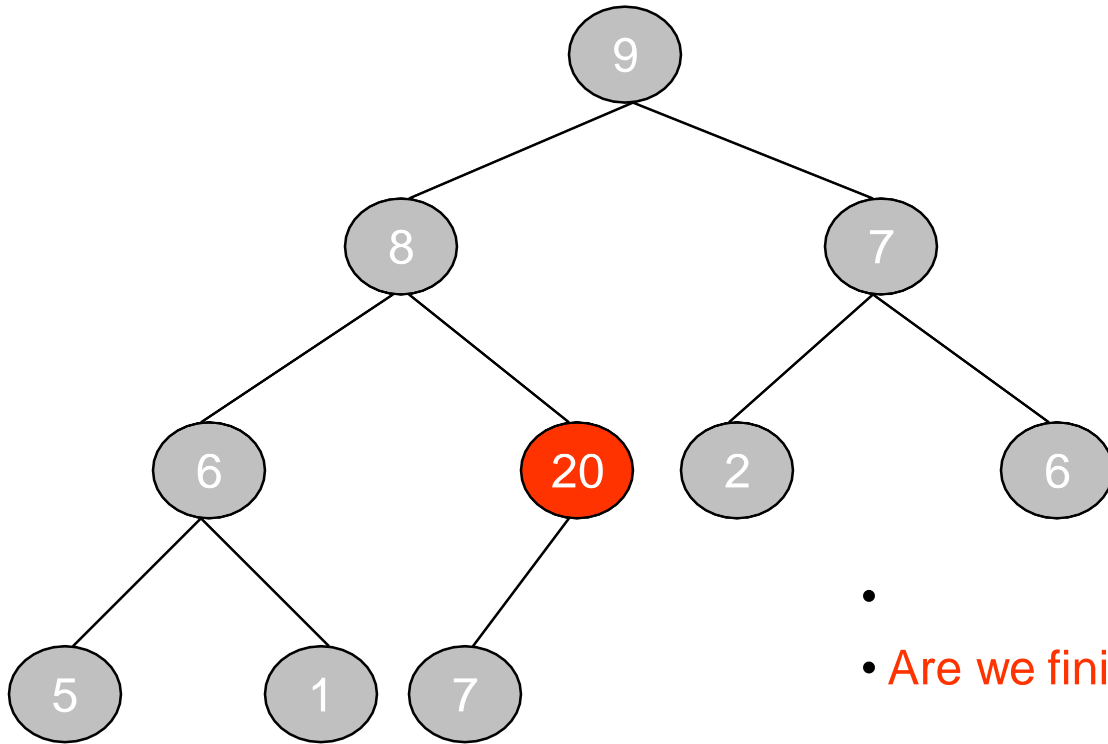
-
- Are we finished?

Insertion into a Max Heap



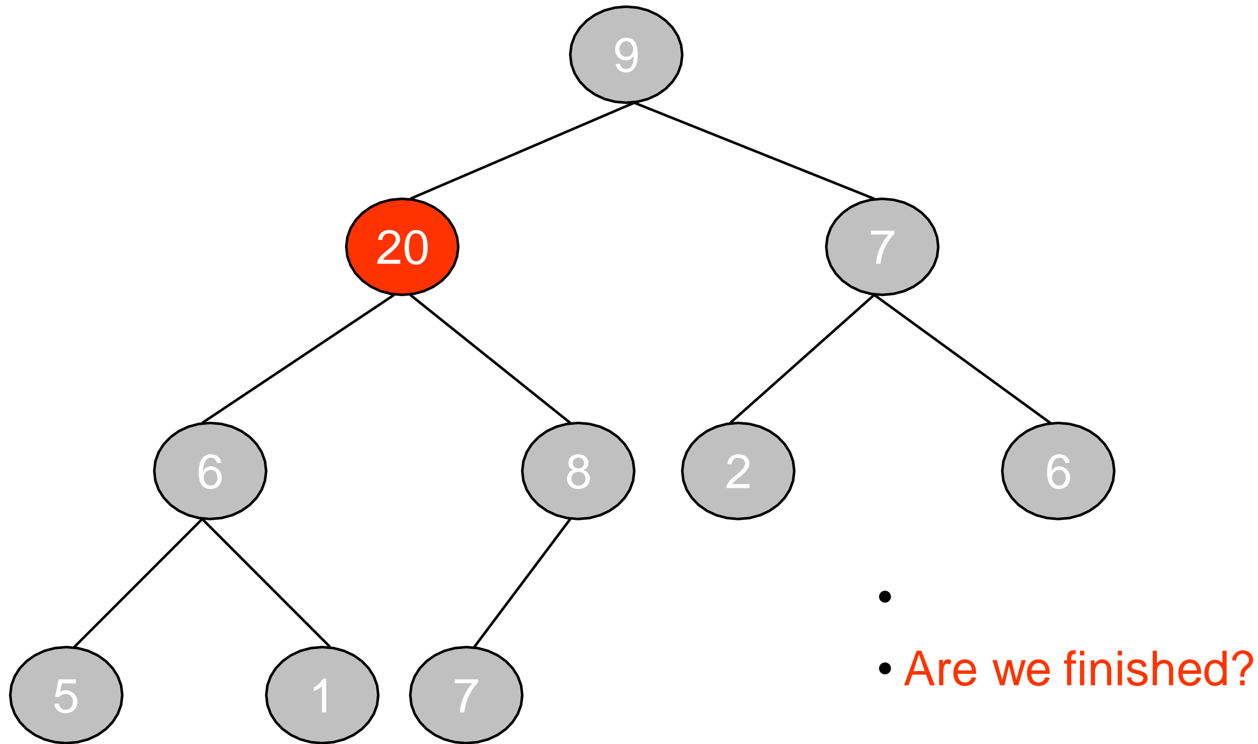
-
- Are we finished?

Insertion into a Max Heap



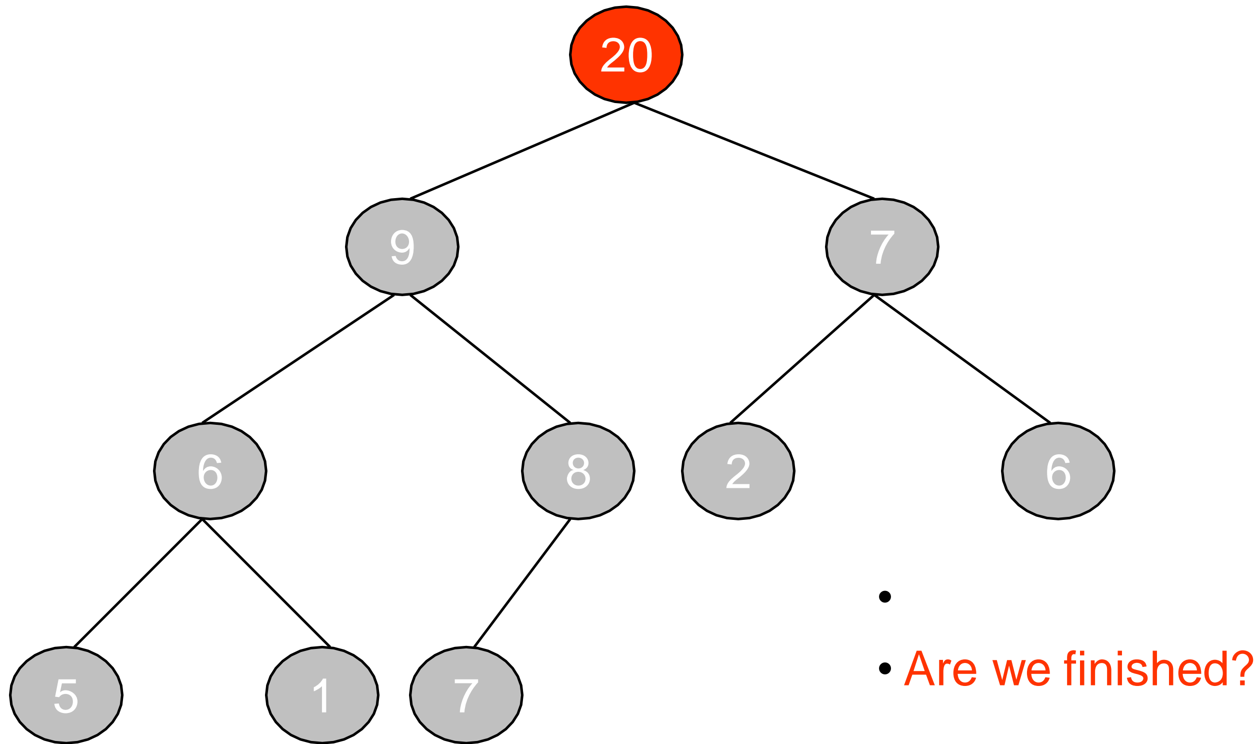
-
- Are we finished?

Insertion into a Max Heap



-
- Are we finished?

Insertion into a Max Heap



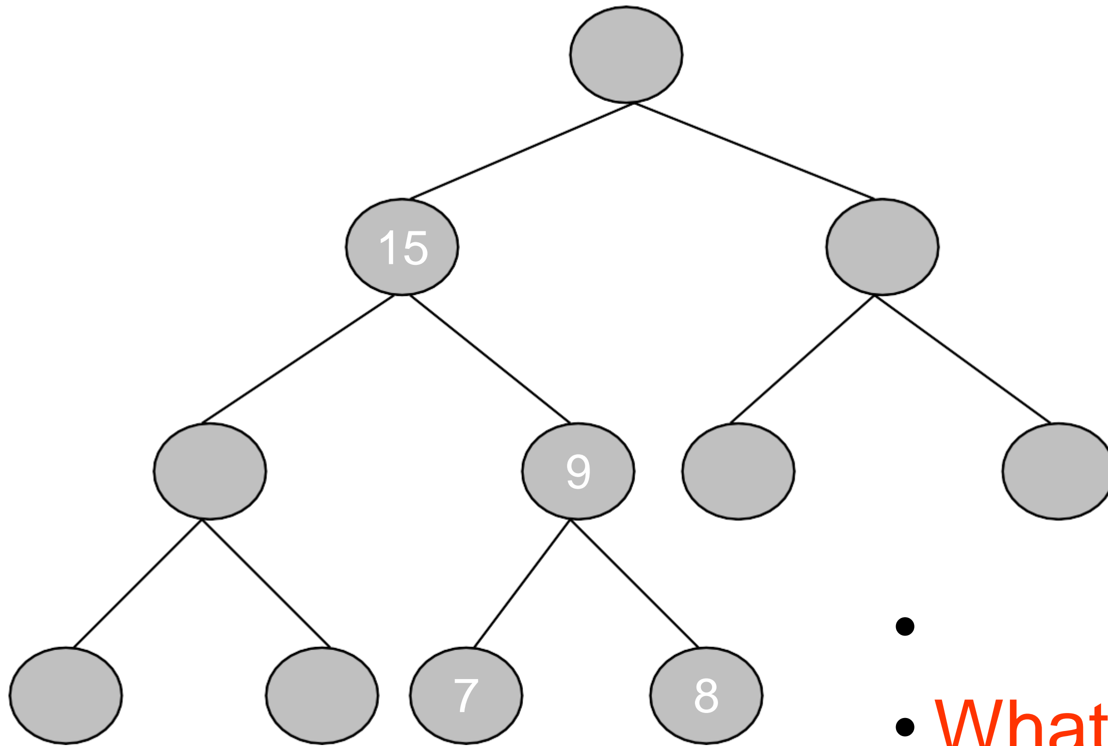
-
- Are we finished?

Complexity of Insertion

At each level, we do $\Theta(1)$ work

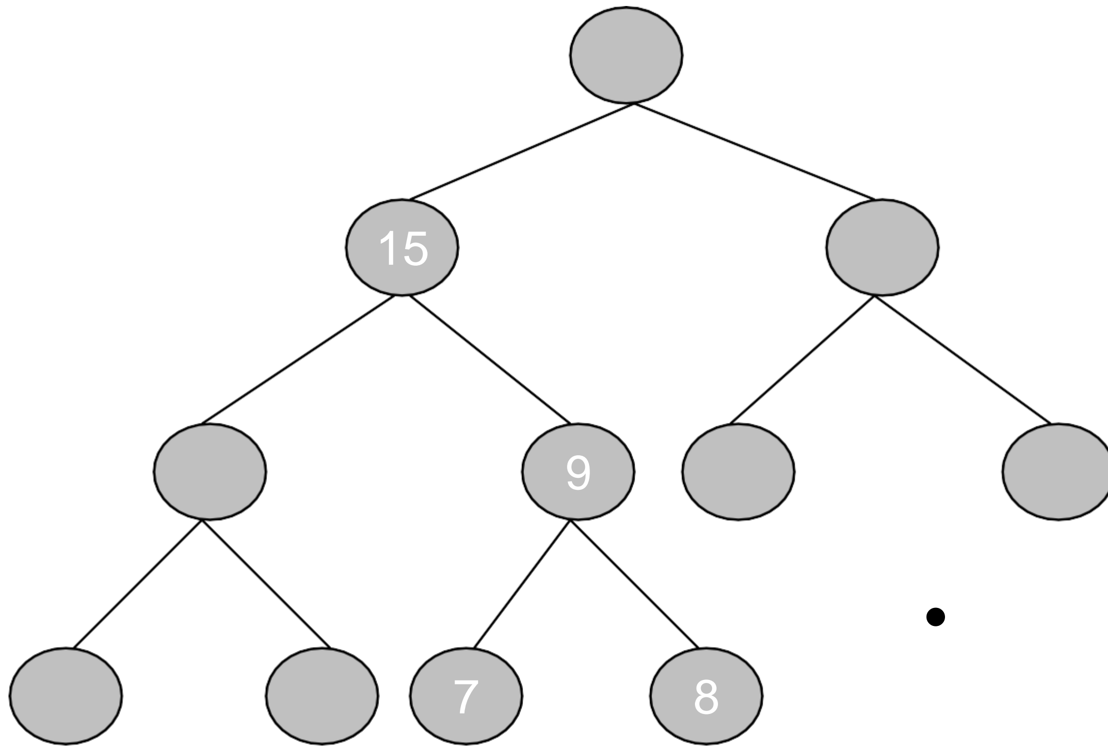
Thus the time complexity is **O(height) = O($log_2 n$)**, where n is the heap size

Deletion from a Max Heap



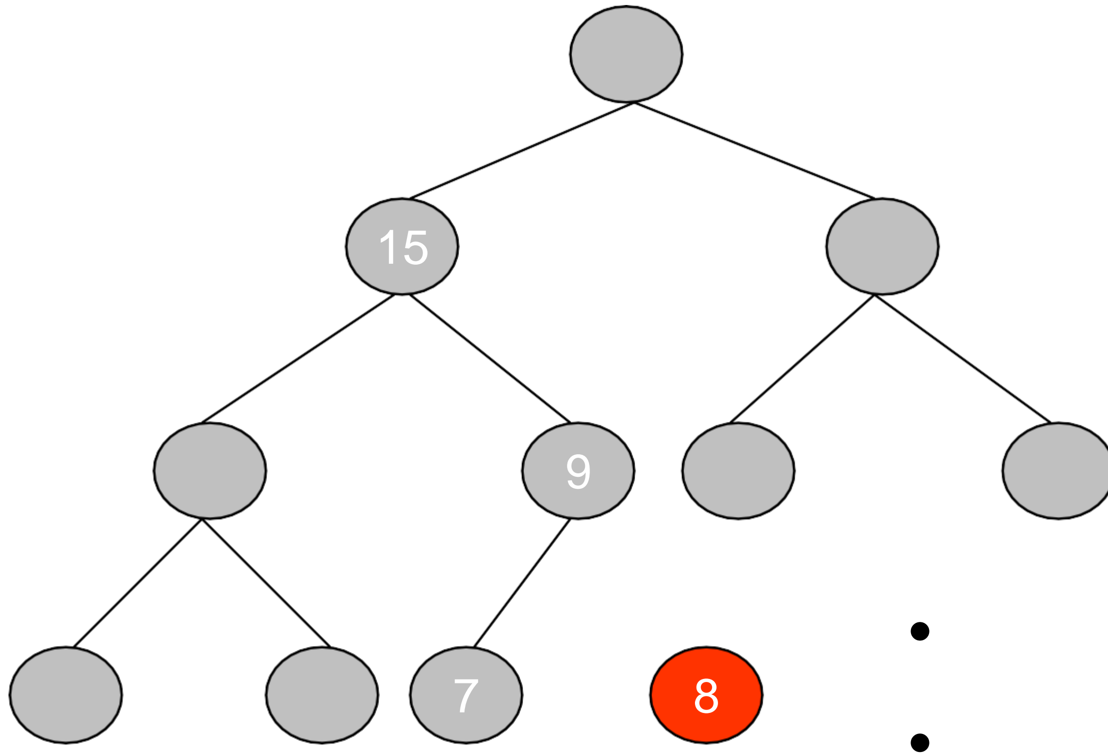
-
- What happens when we delete an element?

Deletion from a Max Heap

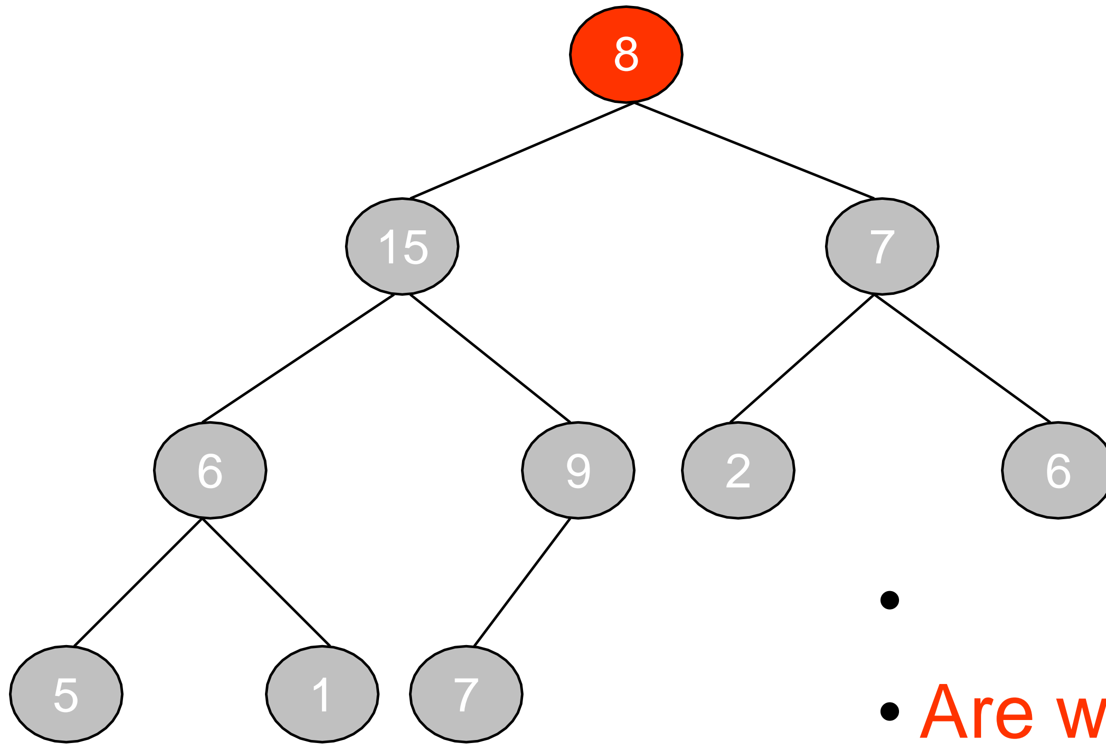


-
- Are we finished?

Deletion from a Max Heap

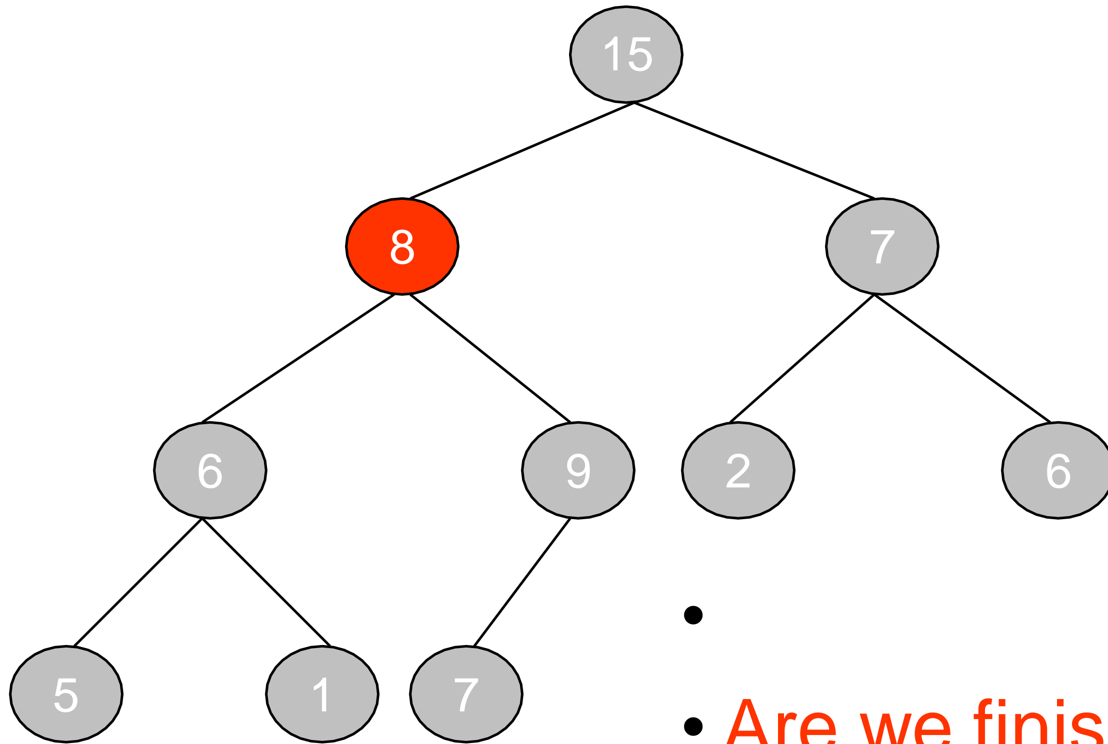


Deletion from a Max Heap



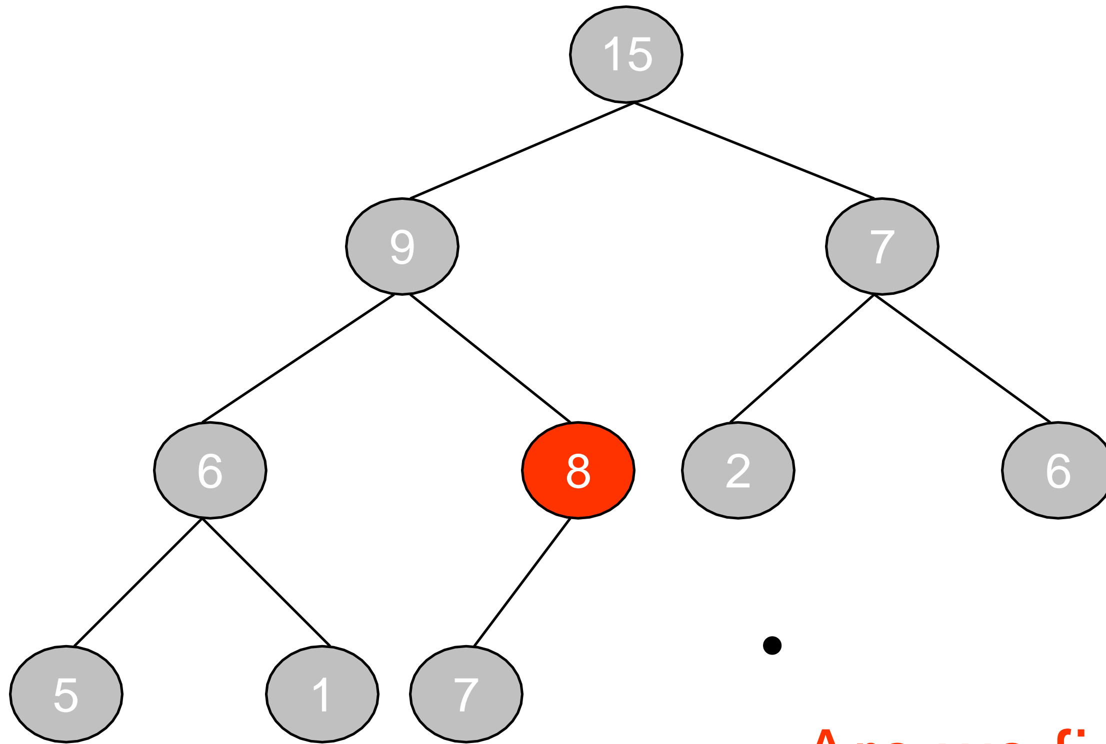
-
- Are we finished?

Deletion from a Max Heap



-
- Are we finished?

Deletion from a Max Heap



-
- Are we finished?

Complexity of Deletion

- The time complexity of deletion is the same as insertion
- At each level, we do $\Theta(1)$ work
- Thus the time complexity is **O(height) = O($log_2 n$)**, where n is the heap size

END!!!