

Unit VI : Queues

Basic concept, Queue as Abstract Data Type, Representation of Queue using Sequential organization, Queue Operations, Circular Queue and its advantages, Multi-queues, Linked Queue and Operations. Deque-Basic concept, types (Input restricted and Output restricted), Priority Queue- Basic concept, types (Ascending and Descending).
Case Study : Priority queue in bandwidth management

Queues

- A queue is a data structure that models/enforces the **first-come first-serve** order, or equivalently the **first-in first-out (FIFO)** order
- Like a stack, special kind of linear list
- One end is called **front**
- Other end is called **rear**
- Additions (insertions or enqueue) are done at the rear only
- Removals (deletions or dequeue) are made from the front only

Bus Stop Queue



- Remove a person from the queue

Bus Stop Queue



front



rear



Bus Stop Queue



front



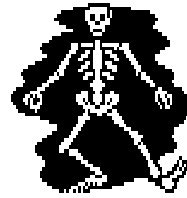
rear



Bus Stop Queue



front



rear



- Add a person to the queue
- A queue is a FIFO (First-In, First-Out) list.

Queue ADT

AbstractDataType queue {

instances

ordered list of elements; one end is the front; the other is the rear;

operations

empty(): Return true if queue is empty, return false otherwise

size(): Return the number of elements in the queue

front(): Return the front element of queue

back(): Return the back (rear) element of queue

dequeue(): Remove an element from the queue

enqueue(x): Add element x to the queue

}

Queue ADT

objects: a finite ordered list with zero or more elements.

methods:

for all $queue \in \text{Queue}$, $item \in \text{element}$,
 $\text{max_queue_size} \in \text{positive integer}$

Queue $\text{createQ}(\text{max_queue_size}) ::=$
create an empty queue whose maximum size is
 max_queue_size

Boolean $\text{isFullQ}(\text{queue}, \text{max_queue_size}) ::=$
if (number of elements in $\text{queue} == \text{max_queue_size}$) **return** TRUE
else **return** FALSE

Queue $\text{Enqueue}(\text{queue}, \text{item}) ::=$
if ($\text{isFullQ}(\text{queue})$) queue_full
else insert item at rear of queue and return queue

Boolean $\text{isEmptyQ}(\text{queue}) ::=$
if ($\text{queue} == \text{CreateQ}(\text{max_queue_size})$) **return** TRUE
else **return** FALSE

Element $\text{dequeue}(\text{queue}) ::=$
if ($\text{isEmptyQ}(\text{queue})$) **return**
else remove and return the item at front of queue.

Operations on Queues

- **Insert(item):** (also called enqueue)
 - It adds a new item to the tail of the queue
- **Remove():** (also called delete or dequeue)
 - It deletes the head item of the queue, and returns to the caller. If the queue is already empty, this operation returns NULL
- **getHead():**
 - Returns the value in the head element of the queue
- **getTail():**
 - Returns the value in the tail element of the queue
- **isEmpty()**
 - Returns **true** if the queue has no items
- **isFull()**
 - Returns **true** if the queue queue is full
- **size()**
 - Returns the number of items in the queue

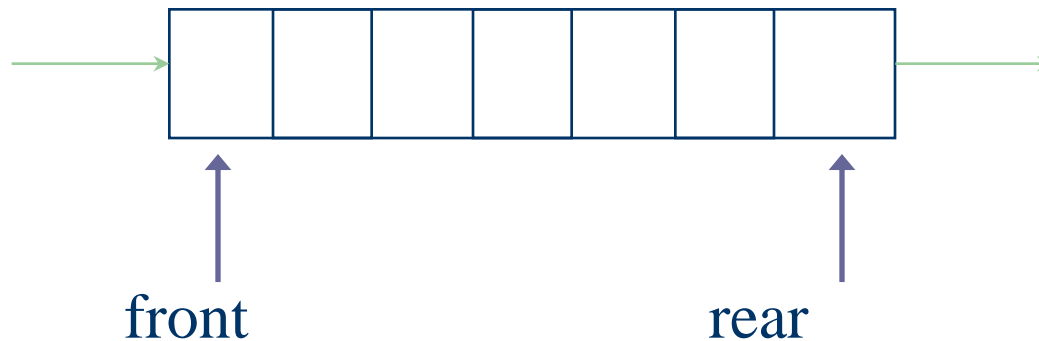
Examples of Queues

- An electronic mailbox is a queue
 - The ordering is chronological (by arrival time)
- A waiting line in a store, at a service counter, on a one-lane road
- Equal-priority processes waiting to run on a processor in a computer system
- Printer queue,
- keystroke queue
- PRACTICAL EXAMPLE : A line at a ticket counter for buying tickets operates on above rules
- IN COMPUTER WORLD : In a batch processing system, jobs are queued up for processing.

Queue implementation

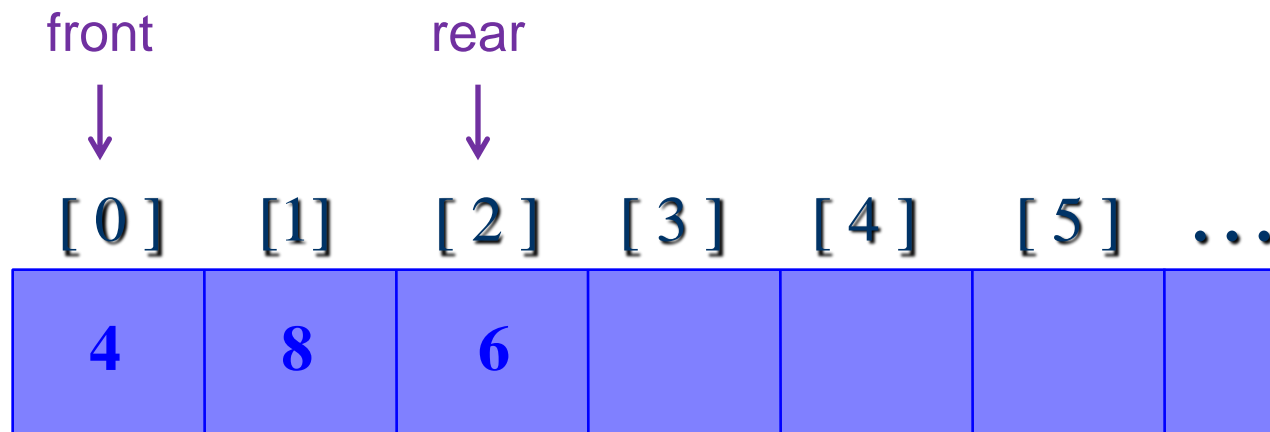
It is also possible to represent Queues using

1. Array-based representation
2. Linked representation



Array Implementation

- A queue can be implemented with an array, as shown here. For example, this queue contains the integers 4 (at the front), 6 (at the rear).



An array of integers
to implement a queue
of integers

We don't care what's in
this part of the array.

Array Implementation

- The easiest implementation also keeps track of the number of items in the queue and the index of the first element (at the front of the queue), the last element (at the rear).

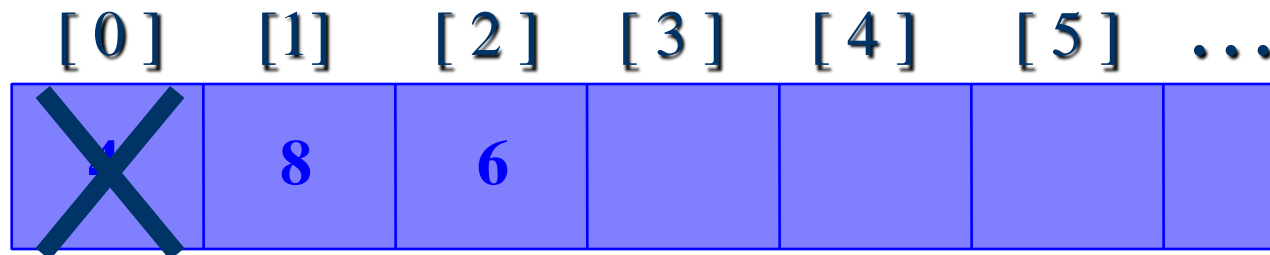
3	size
0	front
2	rear

[0]	[1]	[2]	[3]	[4]	[5]	...
4	8	6				

A Dequeue Operation

- When an element leaves the queue, size is decremented, and first changes, too.

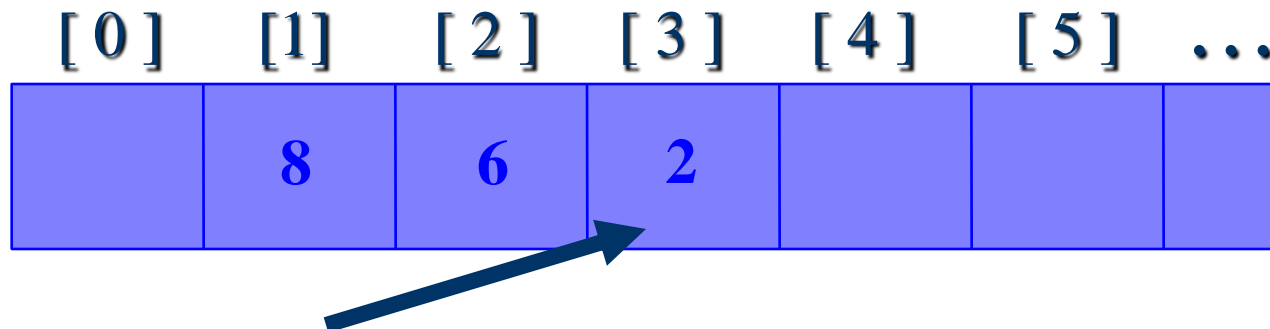
2 size
1 front
2 rear



An Enqueue Operation

- When an element enters the queue, size is incremented, and last changes, too.

3	size
1	front
3	rear



At the End of the Array

- There is special behavior at the end of the array. For example, suppose we want to add a new element to this queue, where the last index is [5]:

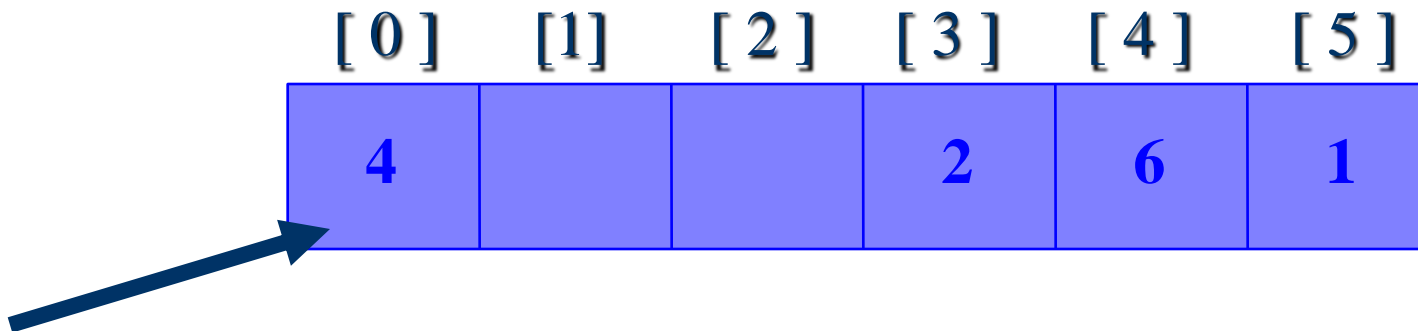
3 size
3 front
5 rear

[0]	[1]	[2]	[3]	[4]	[5]
			2	6	1

At the End of the Array

- The new element goes at the front of the array (if that spot isn't already used):

4	size
3	front
0	rear



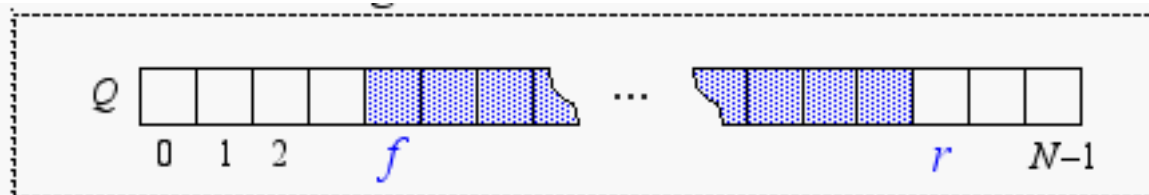
Pros and cons of Linear Queue using Arrays

- Easy to implement
- But it has a limited capacity with a fixed array
- Or you must use a dynamic array for an unbounded capacity
- Inefficient utilization of memory
 - After dequeue operation only front gets incremented by 1, but that position is not used later.
 - so when we perform more add and delete operation, memory wastage increase.
- Special behavior is needed when the rear reaches the end of the array.

Array-based Queue Implementation

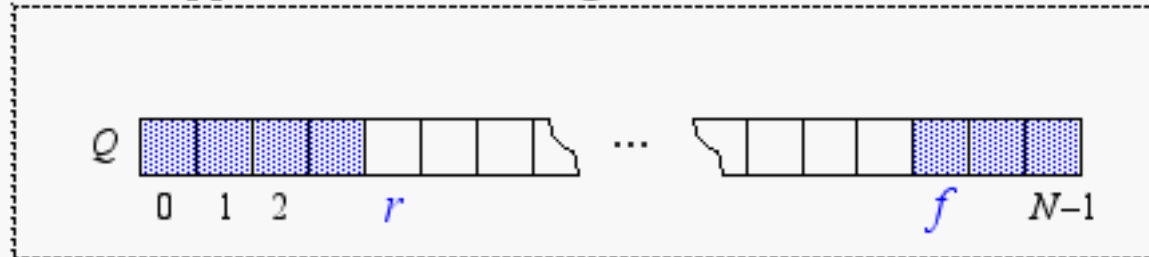
- As with the array-based stack implementation, the array is of fixed size
 - A queue of maximum N elements
- Slightly more complicated
 - Need to maintain track of both **front** and **rear**

Implementation 1



- “wrapped around” configuration

Implementation 2



Implementation 1:

createQ, isEmptyQ, isFullQ

```
# define MAX_QUEUE_SIZE 100  /* Maximum queue size */

int queue[MAX_QUEUE_SIZE];
int rear = -1;
int front = 0;
Boolean isEmpty(queue) :: front > rear or rear+1 == front
Boolean isFull(queue) :: rear == MAX_QUEUE_SIZE-1
```

```
int isEmpty()
{
    if(front > rear)
        return 1;
    else
        return 0;
}
```

```
int isFull()
{
    if(rear == MAX_QUEUE_SIZE-1)
        return 1;
    else
        return 0;
}
```

```
void enqueue(int X)
{
    if ( ! isQfull() )
    {
        rear = rear + 1;
        Queue[rear] = X;
    }
    else
        print queue is full;
}
```

```
int dequeue()
{
    if ( ! isQempty() )
    {
        x = Queue[front]
        front = front + 1;
        return x;
    }
    else
        return an error key ;
}
```

Implementation 2: Parameter passing

createQ, isEmptyQ, isFullQ

```
# define MAX_QUEUE_SIZE 100  /* Maximum queue size */

int queue[MAX_QUEUE_SIZE];
int rear = -1;
int front = 0;
Boolean isEmpty(queue) :: front > rear or rear+1 == front
Boolean isFull(queue) :: rear == MAX_QUEUE_SIZE-1
```

<pre>int isEmpty(int front,int rear) { if(front > rear) return 1; else return 0; }</pre>	<pre>int isFull(int rear) { if(rear ==MAX_QUEUE_SIZE-1) return 1; else return 0; }</pre>
--	---

```
void enqueue(int Queue[], int *rear,int X)
```

```
{  
    if ( ! isQfull(*rear) )  
    {  
        *rear = *rear + 1;  
        Queue[*rear] = X;  
    }  
    else  
        print queue is full;  
}
```

```
int dequeue(int Queue[],int *front, int rear)
```

```
{  
    if ( ! isQempty(*front,rear) )  
    {  
        x = Queue[*front]  
        *front = *front + 1;  
        return x;  
    }  
    else  
        return an error key ;  
}
```

Implementation 3: *createQ, isEmptyQ, isFullQ*

```
# define MAX_QUEUE_SIZE 100  /* Maximum queue size */
```

```
int queue[MAX_QUEUE_SIZE];
```

```
int rear = -1;
```

```
int front = -1;
```

```
Boolean isEmpty(queue) :: front == rear
```

```
Boolean isFull(queue) :: rear == MAX_QUEUE_SIZE-1
```

```
int isEmpty(int front,int rear)
```

```
{
```

```
    if(front == rear)
```

```
        return 1;
```

```
    else
```

```
        return 0;
```

```
}
```

```
int isQfull(int rear)
```

```
{
```

```
    if(rear ==MAX_QUEUE_SIZE-1)
```

```
        return 1;
```

```
    else
```

```
        return 0;
```

```
}
```



```
void enqueue(int Queue[], int *rear,int X)
```

```
{  
    if ( ! isQfull(*rear) )  
    {  
        *rear = *rear + 1;  
        Queue[*rear] = X;  
    }  
    else  
        print queue is full;  
}
```

```
int dequeue(int Queue[],int *front, int rear)
```

```
{  
    if ( ! isQempty(*front,rear) )  
    {  
        *front = *front + 1;  
        x = Queue[*front]  
        return x;  
    }  
    else  
        return an error key ;  
}
```

Implementation 4: Structure

createQ, isEmptyQ, isFullQ

```
#define MAX_STACK_SIZE 100 /* maximum stack size */
typedef struct {
    int Q[MAX_STACK_SIZE];
    int front, rear;
} queue;
```

```
queue q;
```

```
void initialize_queue(queue *qp)
{
    qp->front = 0, qp->rear = -1;
}
```

```
int isQempty(int front,int rear)
{
    if(front > rear)
        return 1;
    else
        return 0;
}
```

```
int isQfull(int rear)
{
    if(rear ==MAX_QUEUE_SIZE-1)
        return 1;
    else
        return 0;
}
```

```
void enqueue(queue *qp,int X)
{
    if ( ! isQfull(qp->rear) )
    {
        qp->rear = qp->rear + 1;
        qp->Q[qp->rear] = X;
    }
    else
        print queue is full;
}
```

```
int dequeue(queue *qp)
{
    if ( ! isQempty(qp->front,qp->rear) )
    {
        x = qp->Q[qp->front]
        qp->front = qp->front + 1;
        return x;
    }
    else
        return an error key ;
}
```

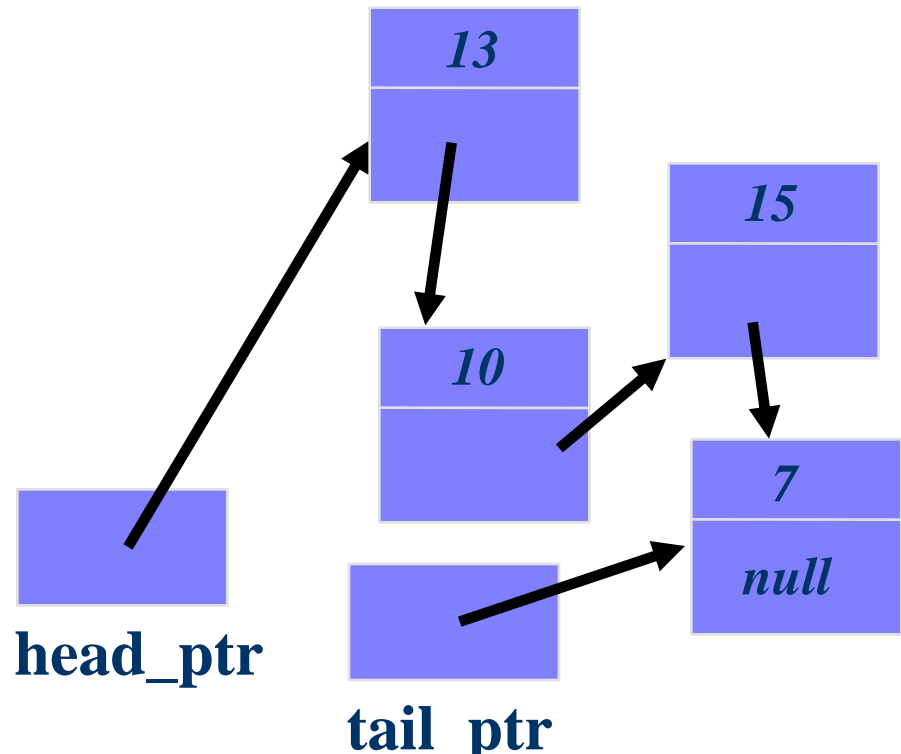
Modified linear Queue

```
void enqueue(int X)
{
    if ( ! isQfull() )
    {
        rear = rear + 1;
        Queue[rear] = X;
    }
    else
        print queue is full;
}
```

```
int dequeue()
{
    if ( ! isQempty() )
    {
        x = Queue[front]
        if(front == rear)
            front = 0, rear = -1;
        else
            front = front + 1;
        return x;
    }
    else
        return an error key ;
}
```

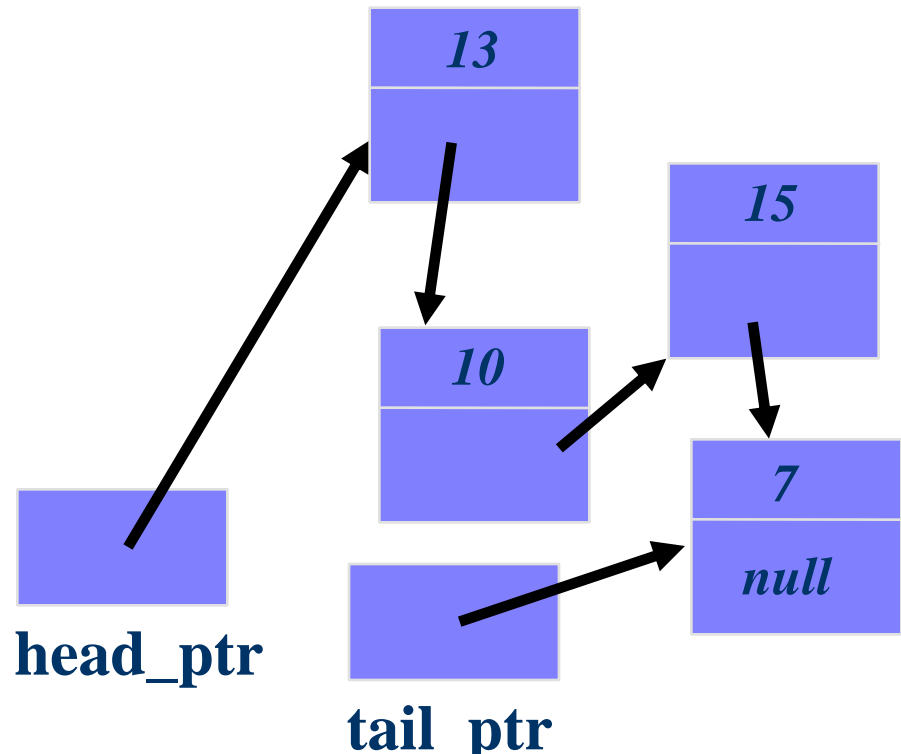
Linked List Implementation

- A queue can also be implemented with a linked list with both a head and a tail pointer.



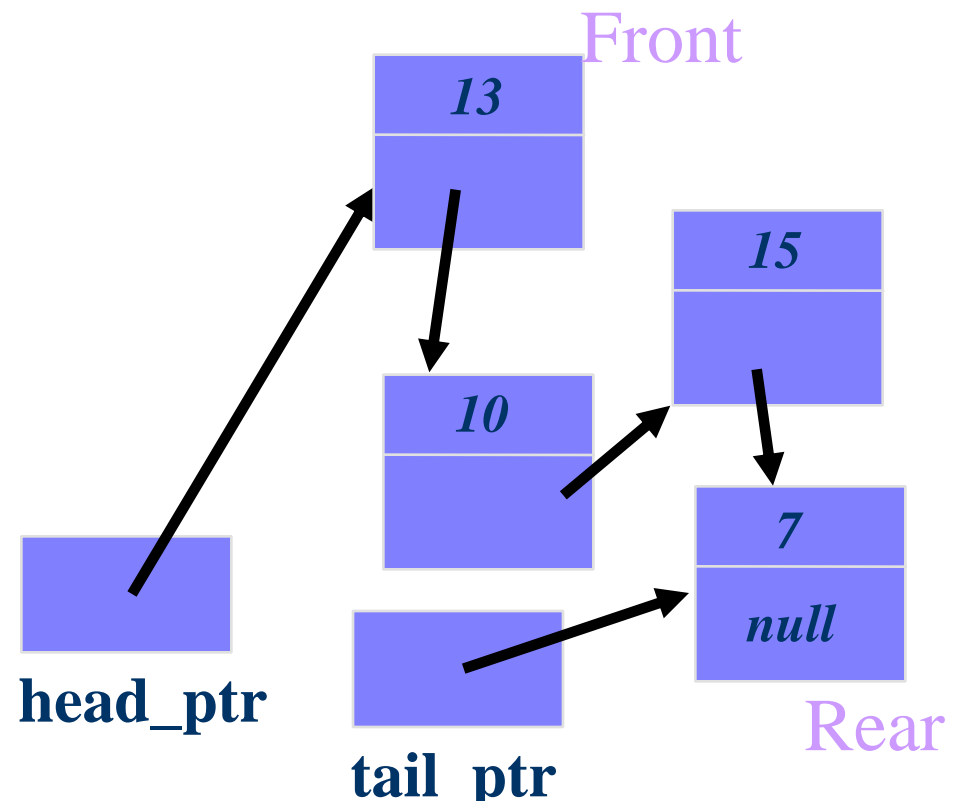
Linked List Implementation

- Which end do you think is the front of the queue? Why?



Linked List Implementation

- The head_ptr points to the front of the list.
- Because it is harder to remove items from the tail of the list.

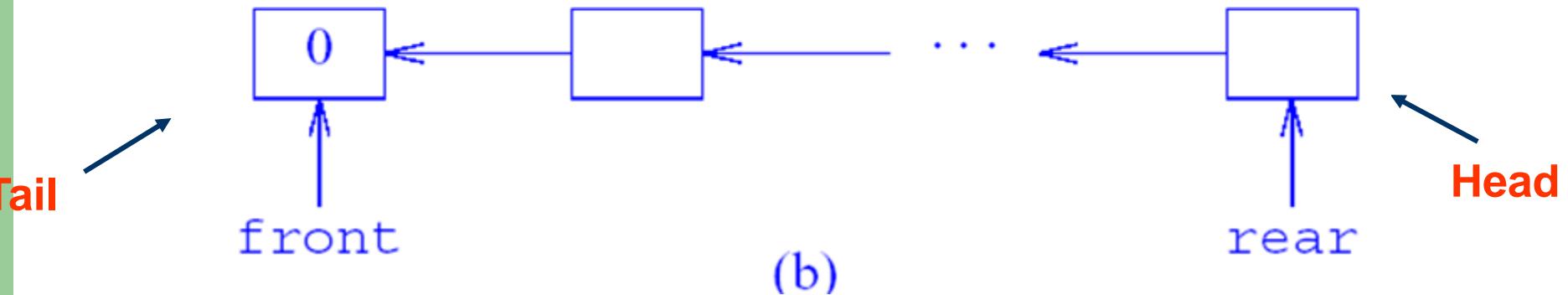
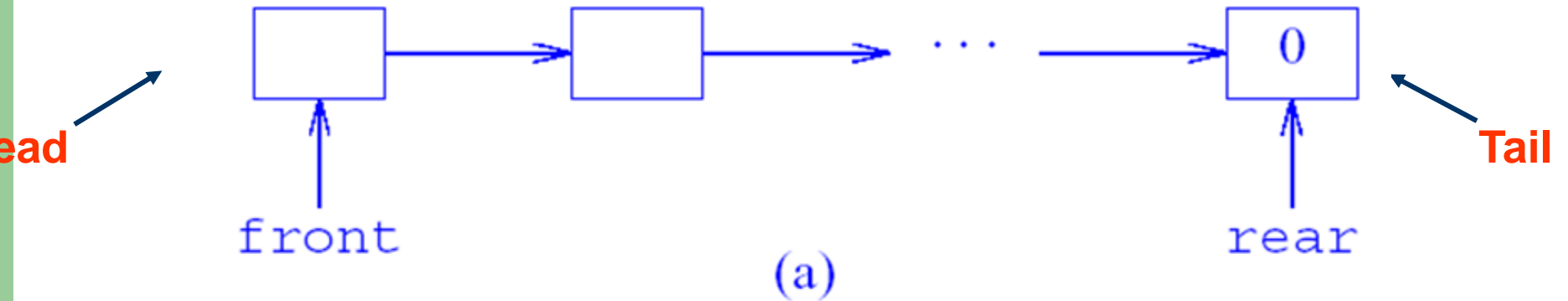


Linked Representation of Queue

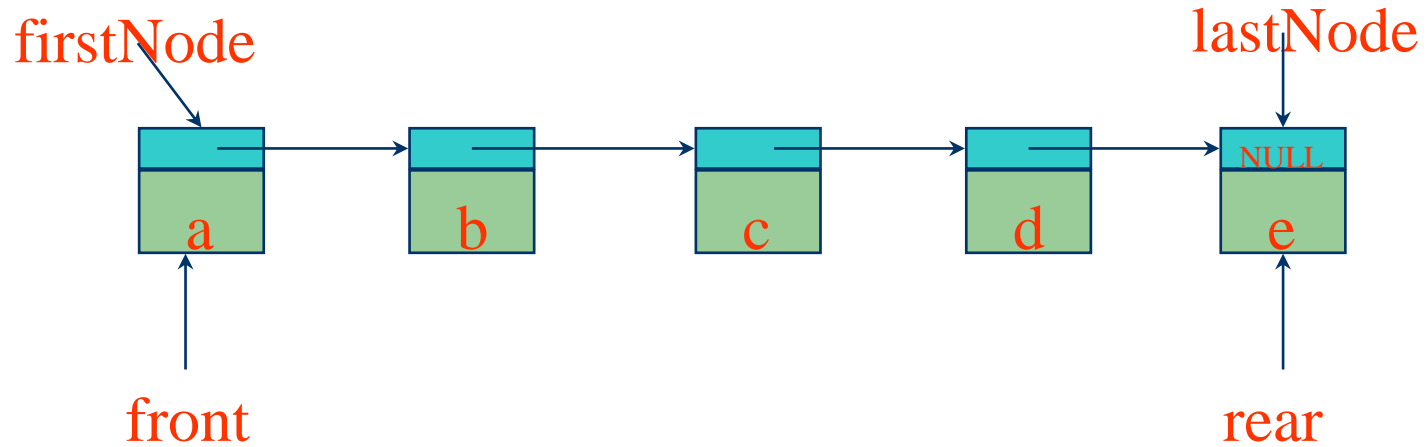
- Can represent a queue using a chain
- Need two variables, front and rear, to keep track of the two ends of a queue
- Two options:
 - 1) assign head as front and tail as rear
 - 2) assign head as rear and tail as front

Which option is better and why?

Linked Queue Representations



Linklist representation of Queue



Implementation : Linklist

```
typedef struct Queue{  
    int data;  
    struct Queue *next;  
} queue;
```

```
queue *front,*rear;
```

```
void initialize_queue()  
{  
    front = NULL, rear = NULL;  
}
```

```
int isQempty()  
{  
    if(front == NULL)  
        return 1;  
    else  
        return 0;  
}
```

```
int isQfull(int rear)  
{  
    if(no memory available in the system)  
        return 1;  
    else  
        return 0;  
}
```

List-based QueueImplementation

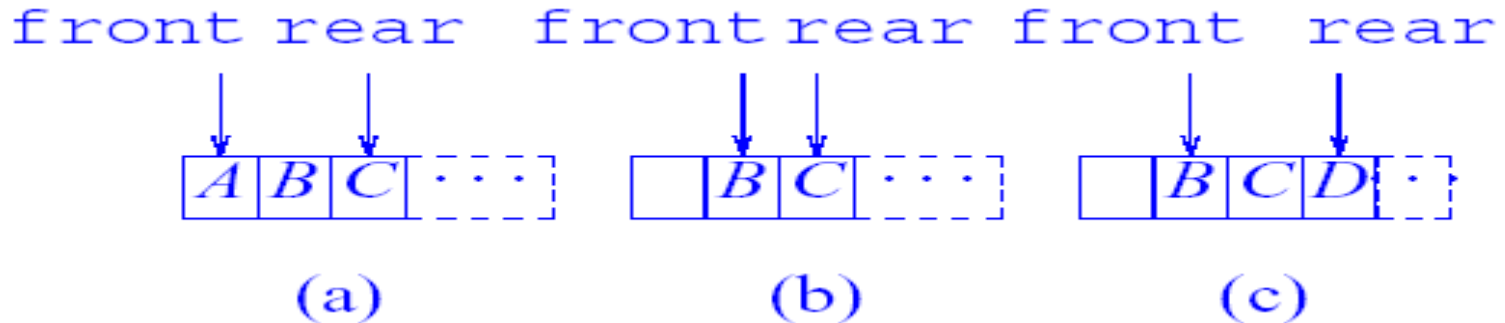
n: enqueue

```
void enqueue(int item)
{
    node = (queue *) malloc (sizeof (queue));
    node->data = item;
    node->next = NULL;
    if (node == NULL)
        { display queue is full
          exit(1);
        }
    else
    {
        if(rear == NULL)
        {
            front = rear = node;
        }
        else
        {
            rear->next = node;
            rear = node;
        }
    }
}
```

Dequeue

```
int dequeue()
{
    if ( ! isQempty())
    {
        x = front->data;
        temp = front;
        if(front == rear)
        {
            rear = NULL
        }
        front = front->next;
        free (temp);
        return x;
    }
    else
        return an error key indicating Queue empty
}
```

Array-based Representation of Queue



- Using modified formula equation

$$location(i) = location(1) + i - 1$$

- No need to shift the queue one position left each time an element is deleted from the queue
- Instead, each deletion causes front to move right by 1
- Front = location(1), rear = location(last element), and empty queue has rear < front
- What do we do when rear = Maxsize - 1 and front > 0?

Time complexity

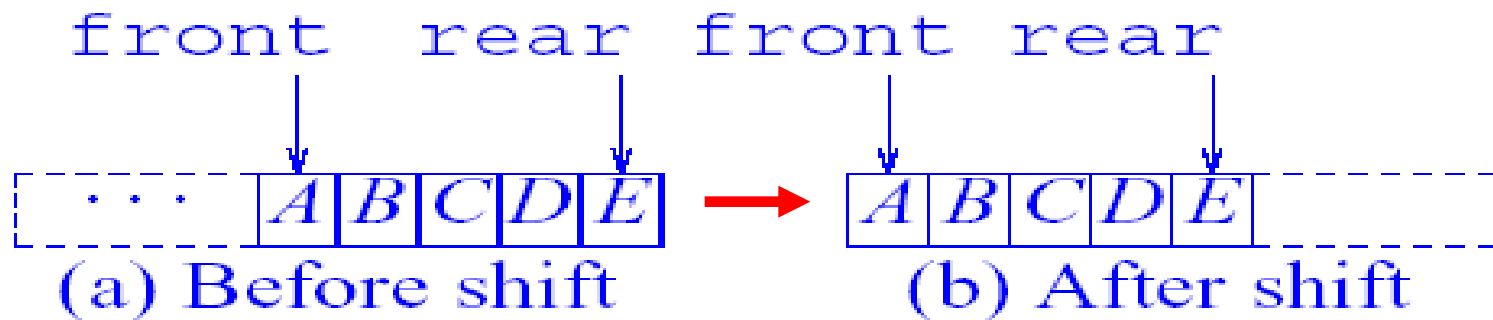
Queue Operation Complexity for Different Implementations

	Array	List Singly-Linked
dequeue()	$O(1)$	$O(1)$
Enqueue()	$O(1)$	$O(1)$
front()	$O(1)$	$O(1)$
Size(), isEmpty().,isfull()	$O(1)$	$O(1)$

Array-based Representation of Queue

- **Shifting a queue**

- To continue adding to the queue, we shift all elements to the left end of the queue
 - But shifting increases the worst-case add time from $\Theta(1)$ to $\Theta(n)$
- ➔ Need a better method!



Shifting a Queue

Array-based Representation of Queue

- Remedy in modified formula equation that can provide the worst-case add and delete times in $\Theta(1)$:

$$location(i) = (location(1) + i - 1) \% Maxsize$$

→ This is called a **Circular Queue**

Circular Queue

- When a new item is inserted at the rear, the pointer to rear moves upwards.
- Similarly, when an item is deleted from the queue the front arrow moves downwards.
- After a few insert and delete operations the rear might reach the end of the queue and no more items can be inserted although the items from the front of the queue have been deleted and there is space in the queue.

Circular Queue


- To solve this problem, queues implement wrapping around. Such queues are called Circular Queues.
- Both the front and the rear pointers wrap around to the beginning of the array.
- It is also called as “Ring buffer”.
- Items can inserted and deleted from a queue in $O(1)$ time.

Advantage of circular queue

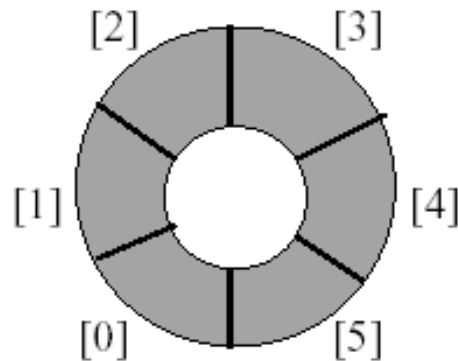
- circular queue uses memory efficiently as compared to linear queue because while doing insertion after deletion operation it allocate an extra space the first remaining vacant but in circular queue the first is used as it comes immediate after the last.

Custom Array Queue

- Use a 1D array queue

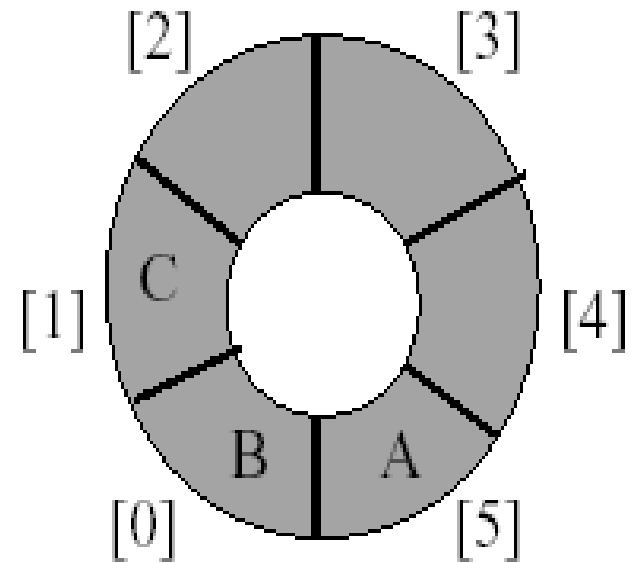
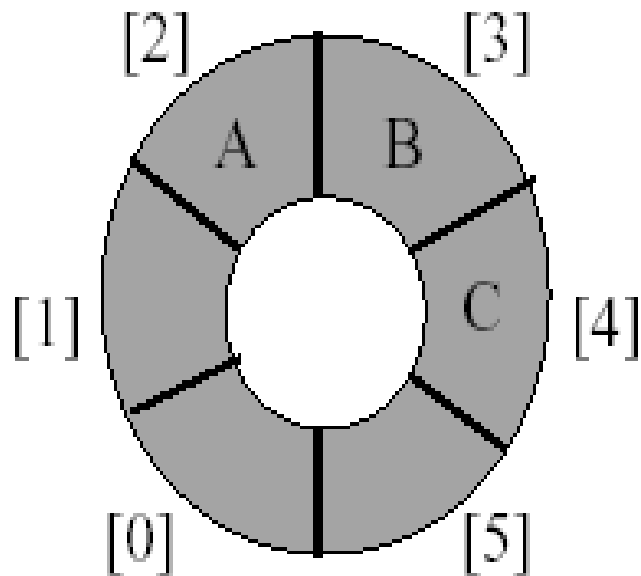
queue[] 

- Circular view of array



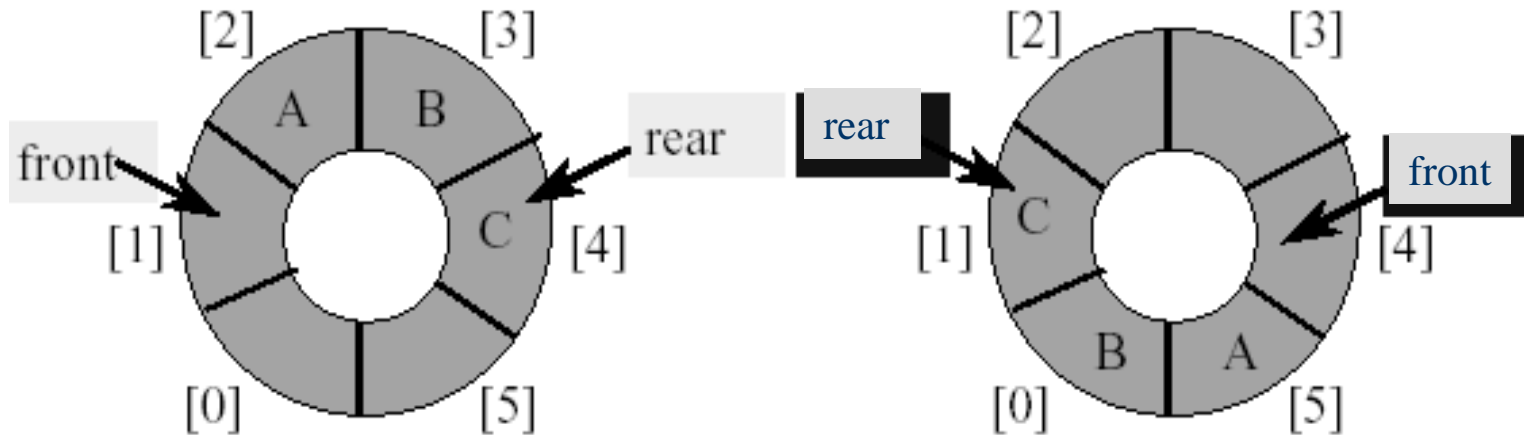
Custom Array Queue

- Possible configurations with three elements.



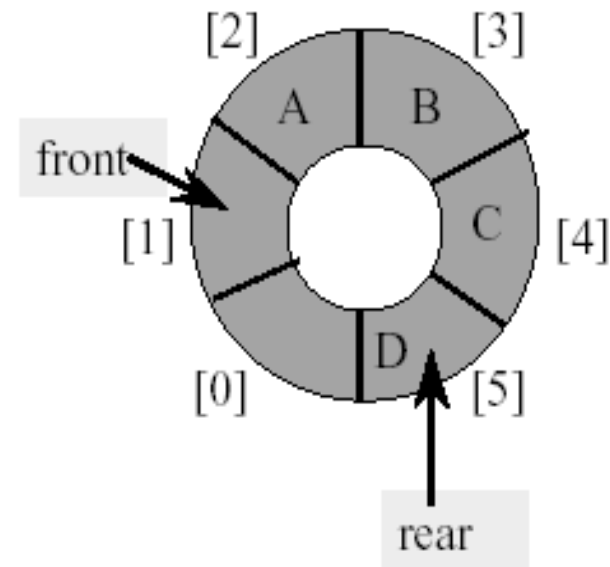
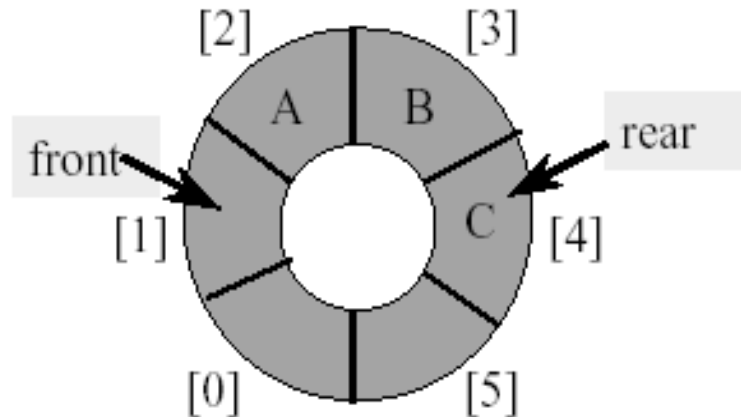
Custom Array Queue

- Use integer variables 'front' and 'rear'.
 - 'front' is one position counter-clockwise from first element
 - 'rear' gives the position of last element



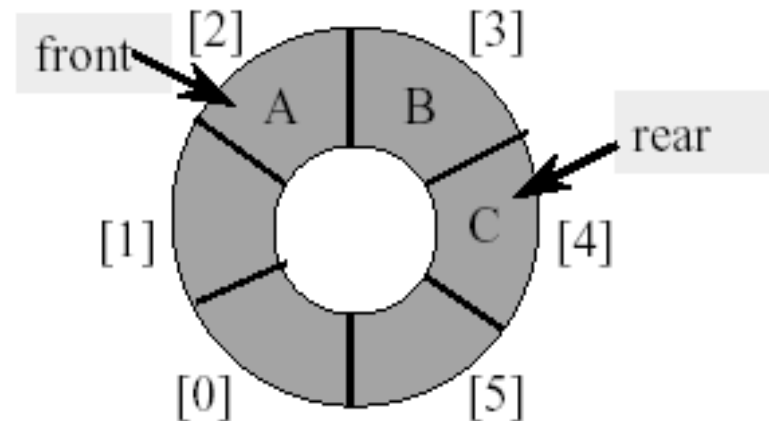
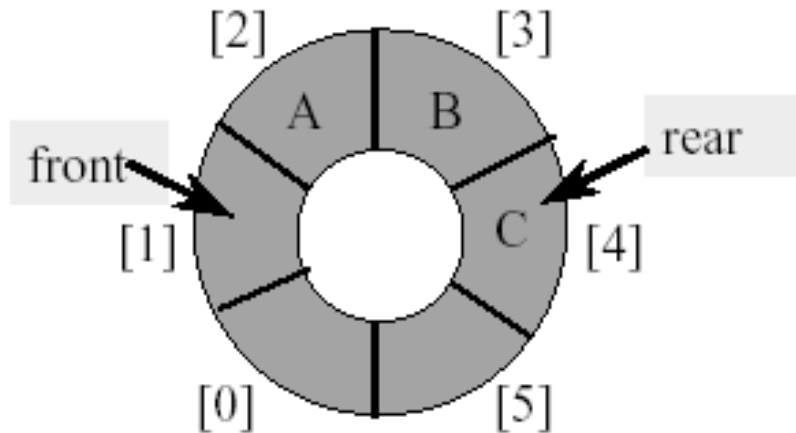
Custom Array Queue

- Add an element
 - Move 'rear' one clockwise.
 - Then put an element into queue[rear].



Custom Array Queue

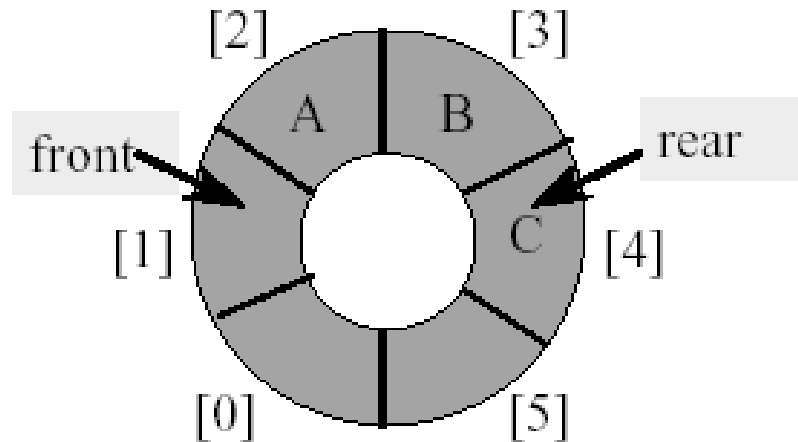
- Remove an element
 - Move front one clockwise.
 - Then extract from queue[front].



Custom Array Queue

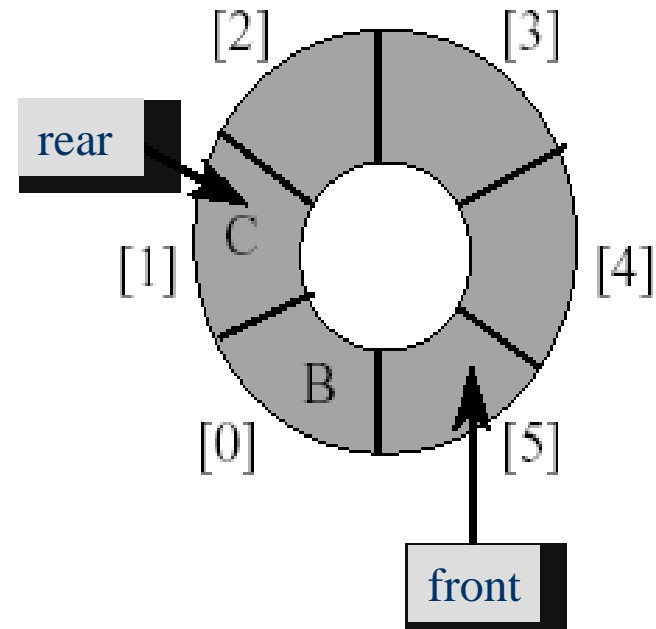
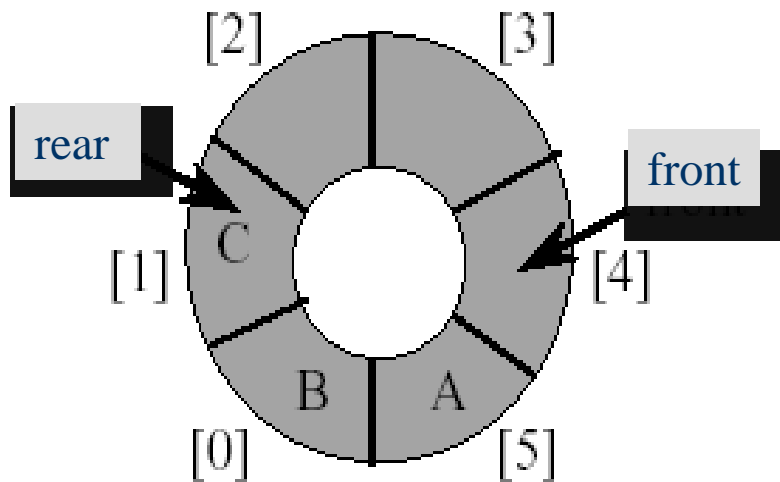
- Moving clockwise

- rear++;
- if (rear == queue.length) rear = 0;
- rear = (rear + 1) % queue.length;



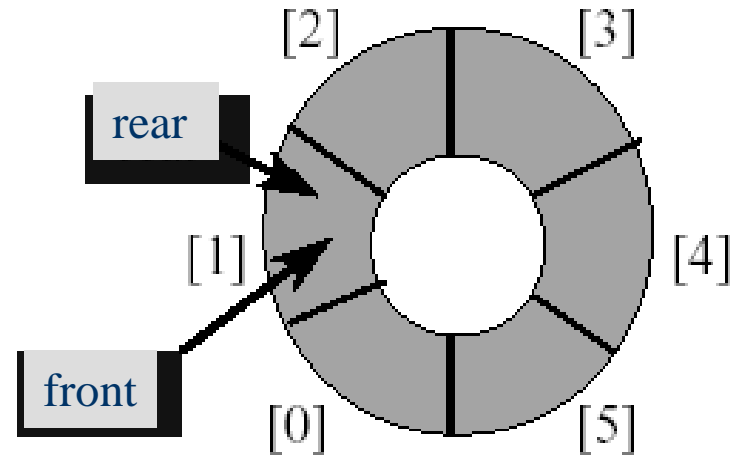
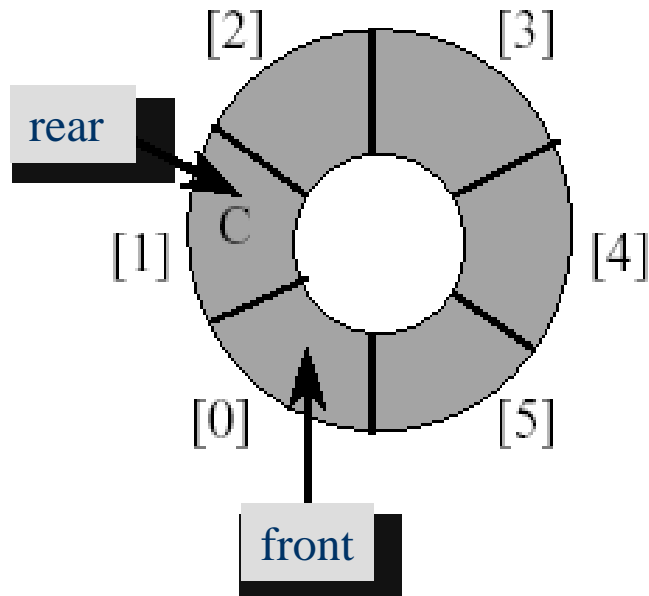
Custom Array Queue

- Empty that queue



Custom Array Queue

- Empty that queue

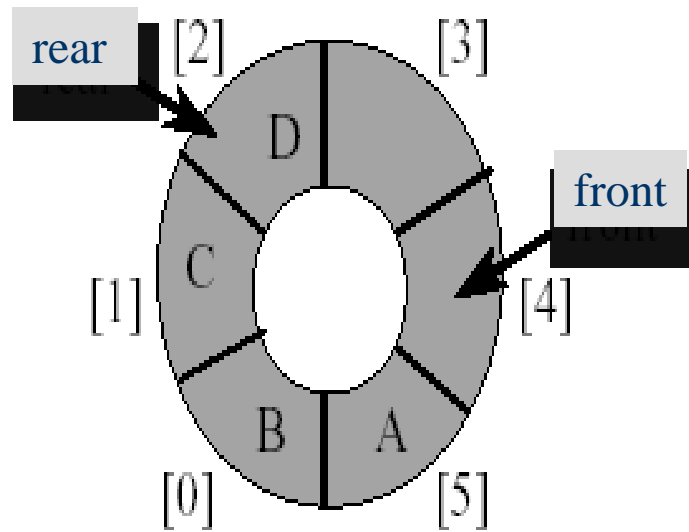
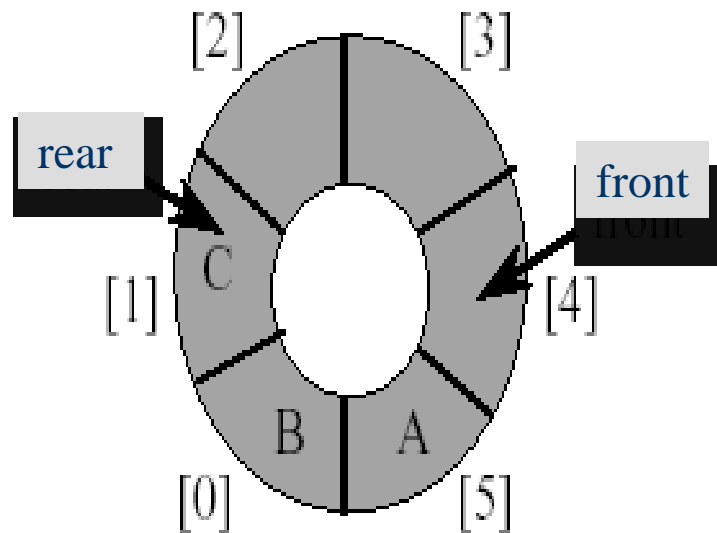


Custom Array Queue

- Empty that queue
 - When a series of removals causes the queue to become empty, **front = rear**.
 - When a queue is constructed, it is empty.
 - So initialize $\text{front} = \text{rear} = 0$.

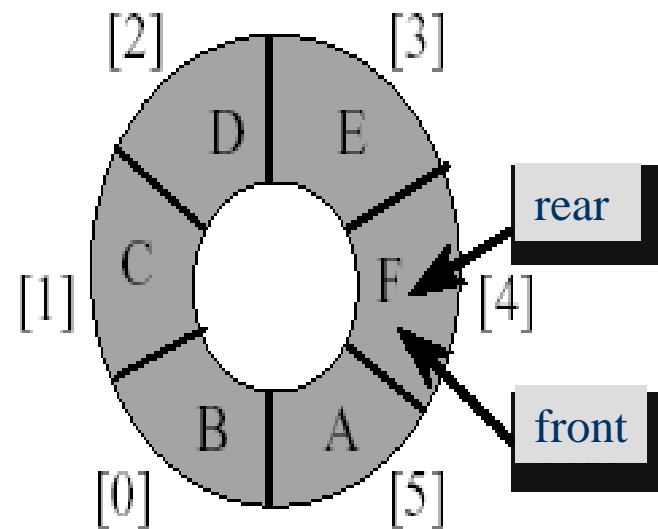
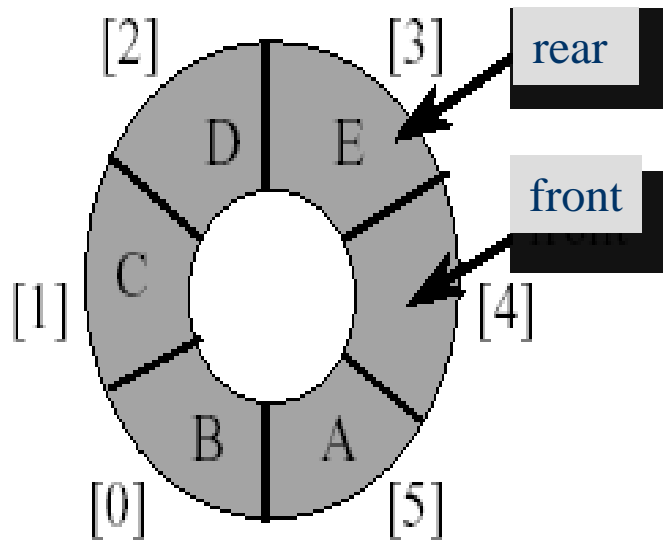
Custom Array Queue

- A Full Tank Please



Custom Array Queue

- A Full Tank Please



Custom Array Queue

- A Full Tank Please
 - When a series of adds causes the queue to become full, $\text{front} = \text{rear}$.
 - So we cannot distinguish between a full queue and an empty queue.
- How to differentiate two cases: queue empty and queue full?

Custom Array Queue

- Remedies

- Don't let the queue get full
 - When the addition of an element will cause the queue to be full, increase array size
- Define a boolean variable `lastOperationIsAdd`
 - Following each add operation set this variable to true.
 - Following each delete operation set this variable to false.
 - Queue is empty iff `(front == rear) && !lastOperationIsAdd`
 - Queue is full iff `(front == rear) && lastOperationIsAdd`

Custom Array Queue

- Remedies (cont'd)
 - Define a variable NumElements / count
 - Following each add operation, increment this variable
 - Following each delete operation, decrement this variable
 - Queue is empty iff `(front == rear) && (!NumElements)`
 - Queue is full iff `(front == rear) && (NumElements)`

Implementation : Circular Queue

```
# define MAX 100  /* Maximum queue size */
```

```
int queue[MAX];
```

```
int rear = -1;
```

```
int front = 0;
```

```
int count = 0;
```

```
Boolean isQEmpty(queue) :: count == 0
```

```
Boolean isQFull(queue) :: count == MAX
```

```
int isQempty(int count)
```

```
{
```

```
    if(count == 0)
```

```
        return 1;
```

```
    else
```

```
        return 0;
```

```
}
```

```
int isQfull(int rear)
```

```
{
```

```
    if( count == MAX)
```

```
        return 1;
```

```
    else
```

```
        return 0;
```

```
}
```

```
void enqueue(int Queue[], int *rear,int *count,int X)
```

```
{
    if ( ! isQfull(*count) )
    {
        *rear = (*rear + 1)%MAX;
        (*count)++;
        Queue[*rear] = X;
    }
    else
        print queue is full;
}
```

```
int dequeue(int Queue[],int *front, int *count)
```

```
{
    if ( ! isQempty(*front,rear) )
    {
        x = Queue[*front];
        *front = (*front + 1)%MAX;
        (*count)--;
        return x;
    }
    else
        return an error key ;
}
```

Implementation : Circular Queue

```
# define MAX 100  /* Maximum queue size */
```

```
int queue[MAX];
```

```
int rear = -1;
```

```
int front = 0;
```

```
Boolean isQEmpty(queue) :: rear == -1
```

```
Boolean isQFull(queue) :: (rear + 1)%MAX == front && rear != -1
```

```
int isQempty(int front,int rear)  int isQfull(int rear)
```

```
{
```

```
    if(rear== -1)
```

```
        return 1;
```

```
    else
```

```
        return 0;
```

```
}
```

```
{
```

```
    if((rear+1)%MAX ==front && rear != -1)
```

```
        return 1;
```

```
    else
```

```
        return 0;
```

```
}
```

```
int enqueue(int Queue[], int *rear,int X)
```

```
{  
    if ( ! isQfull(*rear) )  
    {  
        *rear = (*rear + 1)%MAX;  
        Queue[*rear] = X;  
    }  
    else  
        print queue is full;  
}
```

```
int dequeue(int Queue[],int *front, int rear)
```

```
{  
    if ( ! isQempty(*front,rear) )  
    {  
        x = Queue[*front];  
        if(*front == *rear)  
            *front = 0,*rear = -1;  
        else  
            *front = (*front + 1)%MAX;  
        return x;  
    }  
    else  
        return an error key ;  
}
```

Revisit of Stack Applications

- Applications in which the stack cannot be replaced with a queue
 - Parentheses matching
 - Towers of Hanoi
 - Switchbox routing
 - Method invocation and return
- Application in which the stack may be replaced with a queue
 - Railroad Car Rearrangement
 - Rat in a maze

Application: Rearranging Railroad Cars

Problem description:

A freight train has n railroad cars. Each is to be left at a different station. Assume that the n stations are numbered 1 through n and that the freight train visits these stations in the order n through 1. The railroad cars are labeled by their destination. To facilitate removal of the railroad cars from the train, we must reorder the cars so that they are in the order 1 through n from front to back. When the cars are in this order, the last car is detached at each station. We rearrange the cars at a shunting yard that has an input track, an output track, and k holding tracks between the input and output tracks.

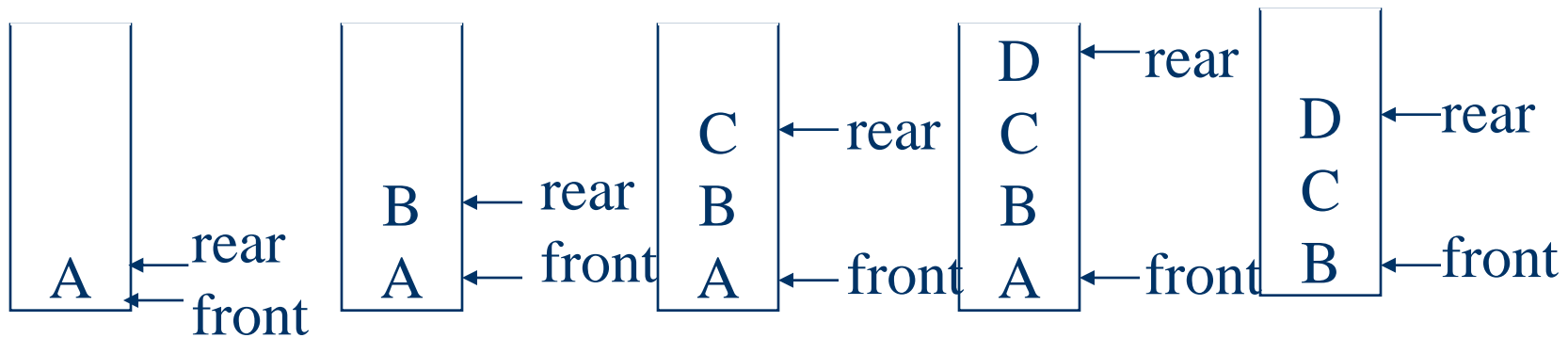
Application: Rearranging Railroad Cars

- This time holding tracks lie between the input and output tracks with the following same conditions:
 - Moving a car from a holding track to the input track or from the output track to a holding track is forbidden
- These tracks operate in a FIFO manner
→ can implement using **queues**
- We reserve track H_k for moving cars from the input track to the output track. So the number of tracks available to hold cars is $k-1$.

Queue Applications

- Real life examples
 - Waiting in line
 - Waiting on hold for tech support
- Applications related to Computer Science
 - Threads
 - Job scheduling (e.g. Round-Robin algorithm for CPU allocation)

First In First Out



Applications: Job Scheduling

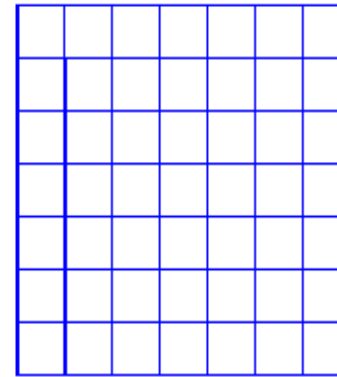
front	rear	Q[0]	Q[1]	Q[2]	Q[3]	Comments
-1	-1					queue is empty
-1	0	J1				Job 1 is added
-1	1	J1	J2			Job 2 is added
-1	2	J1	J2	J3		Job 3 is added
0	2		J2	J3		Job 1 is deleted
1	2			J3		Job 2 is deleted

Rearranging Railroad Cars

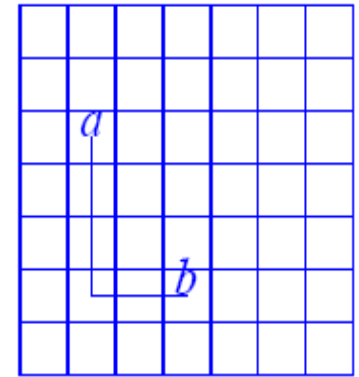
- When a car is to be moved to a holding track, use the following selection method:
 - *Move a car c to a holding track that contains only cars with a smaller label*
 - *If multiple such tracks exist, select one with the largest label at its left end*
 - *Otherwise, select an empty track (if one remains)*
- What happens if no feasible holding track exists?
 - ➔ rearranging railroad cars is NOT possible

Application: Wire Routing

- Similar to Rat in a Maze problem, but this time it has **to find the shortest path between two points to minimize signal delay**
- Used in designing electrical circuit boards

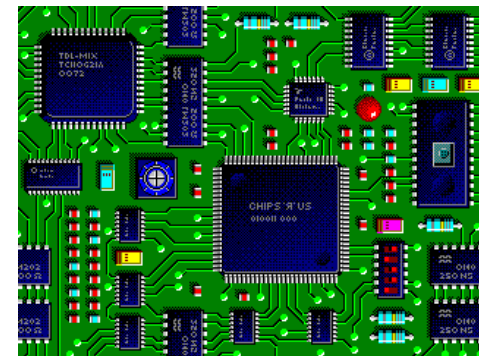


(a) A 7×7 grid



(b) A wire between a and b

Wire Routing Example



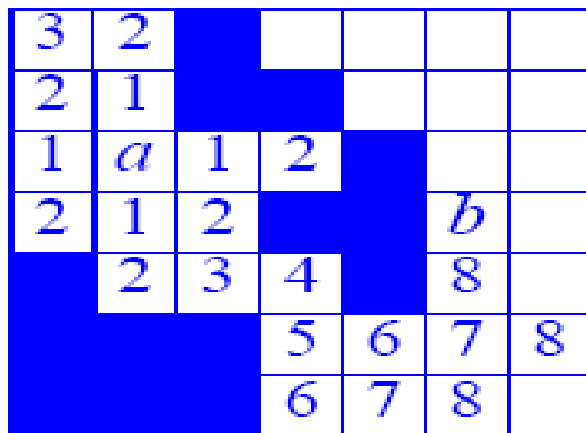
Wire Routing Algorithm

The shortest path between grid positions a and b is found in two passes

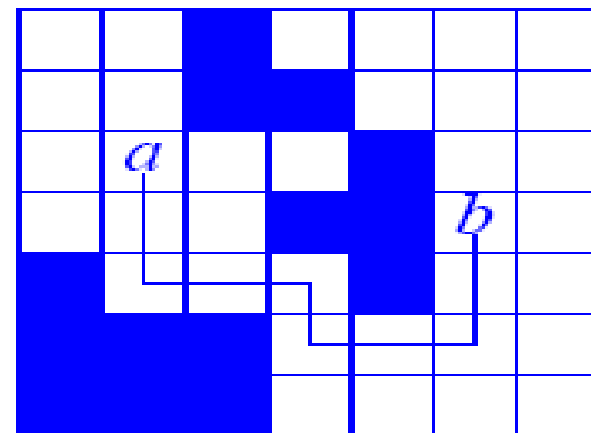
1. Distance-labeling pass (i.e., labeling grids)
2. Path-identification pass (i.e., finding the shortest path)

Wire Routing Algorithm

1. Labeling Grids: Starting from position *a*, label its reachable neighbors 1. Next, the reachable neighbors of squares labeled 1 are labeled 2. This labeling process continues until we either reach *b* or have no more reachable squares. The shaded squares are blocked squares.



(a) Distance labeling

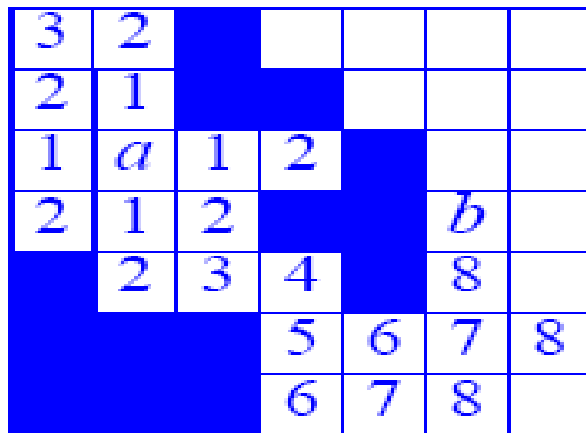


(b) Wire path

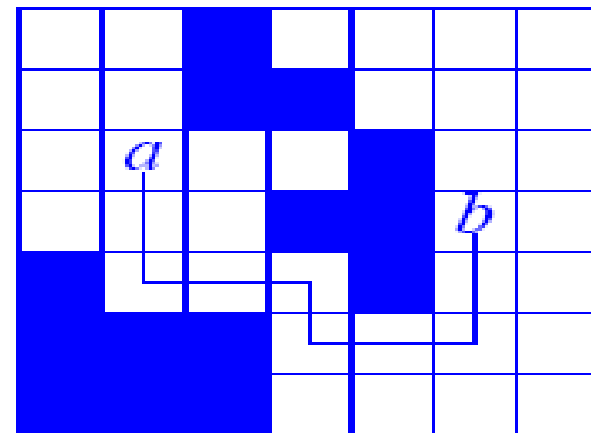
Wire Routing

Wire Routing Algorithm

2. Finding the shortest path: Starting from position b , move to any one of its neighbors labeled one less than b 's label. Such a neighbor must exist as each grid's label is one more than that of at least one of its neighbors. From here, we move to one of its neighbors whose label is one less, and so on until we reach a .



(a) Distance labeling



(b) Wire path

Wire Routing

Wire Routing Algorithm Exercise

Consider the wire-routing grid of Figure 9.13(a). You are to route a wire between $a=(1, 1)$ and $b=(1, 6)$. Label all grid positions that are reached in the distance-labeling pass by their distance value. Then use the methodology of the path-identification pass to mark the shortest wire path.

a	1				b	
1	2				14	
2	3	4	5		13	14
3	4	5			12	13
	5	6	7		11	12
			8	9	10	11
			9	10	11	

a					b	

a					b	

Is there only one shortest path?

Implementing Wire Routing Algorithm

- An $m \times m$ grid is represented as a 2-D array with a 0 representing an open position and a 1 representing a blocked position
- the grid is surrounded by a wall of 1s
- the array `offsets` helps us move from a position to its neighbors
- A queue keeps track of labeled grid positions whose neighbors have yet to be labeled
- What is the time complexity of the algorithm?
 - ➔ $O(m^2)$ for labeling grids & $O(\text{length of the shortest path})$ for path construction

Application: Image-Component Labeling

- Background information:
 - A digitized image is an $m \times m$ matrix of pixels.
 - In a binary image, each pixel is 0 or 1.
 - A 0 pixel represents image background, while a 1 represents a point on an image component.
 - Two pixels are adjacent if one is to the left, above, right, or below the other.
 - Two component pixels that are adjacent are pixels of the same image component.
- Problem: Label the component pixels so two pixels get the same label iff they are pixels of the same image component.

Image-Component Labeling Example

		1				
		1	1			
				1		
			1	1		
	1			1		1
1	1	1				1
1	1	1			1	1

(a) A 7×7 image

		2				
		2	2			
				3		
			3	3		
	4			3		5
4	4	4				5
4	4	4			5	5

(b) Labeled components

Image-Component Labeling

- The blank squares represent background pixels and 1s represent component pixels
- Pixels (1,3), (2,3), and (2,4) are from the same component

Image-Component Labeling Algorithm

- Idea: Similar to the wire-routing problem
- Steps:
 1. Scan pixels one by one (row-major search).
 2. When encounter a 1 pixel, give a unique component identifier.
 3. Find all neighbors by expanding from that pixel and give the unique identifier.
 4. Repeat for all pixels.

Dequeues

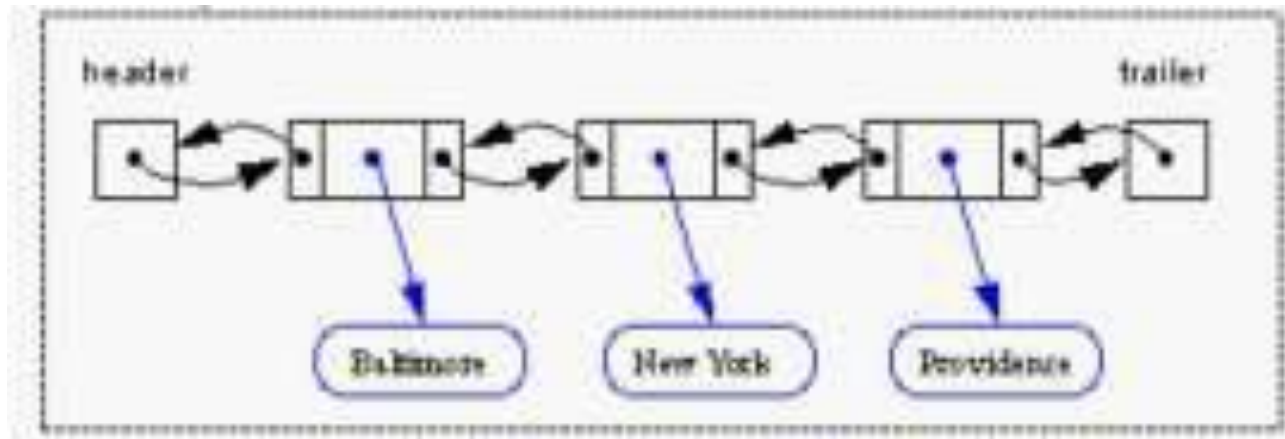
- A **deque** is a double-ended queue
- Insertions *and* deletions can occur at *either* end
- Implementation is similar to that for queues
- Deques are not heavily used
- You should know what a deque is, but we won't explore them much further

Double-Ended Queue ADT

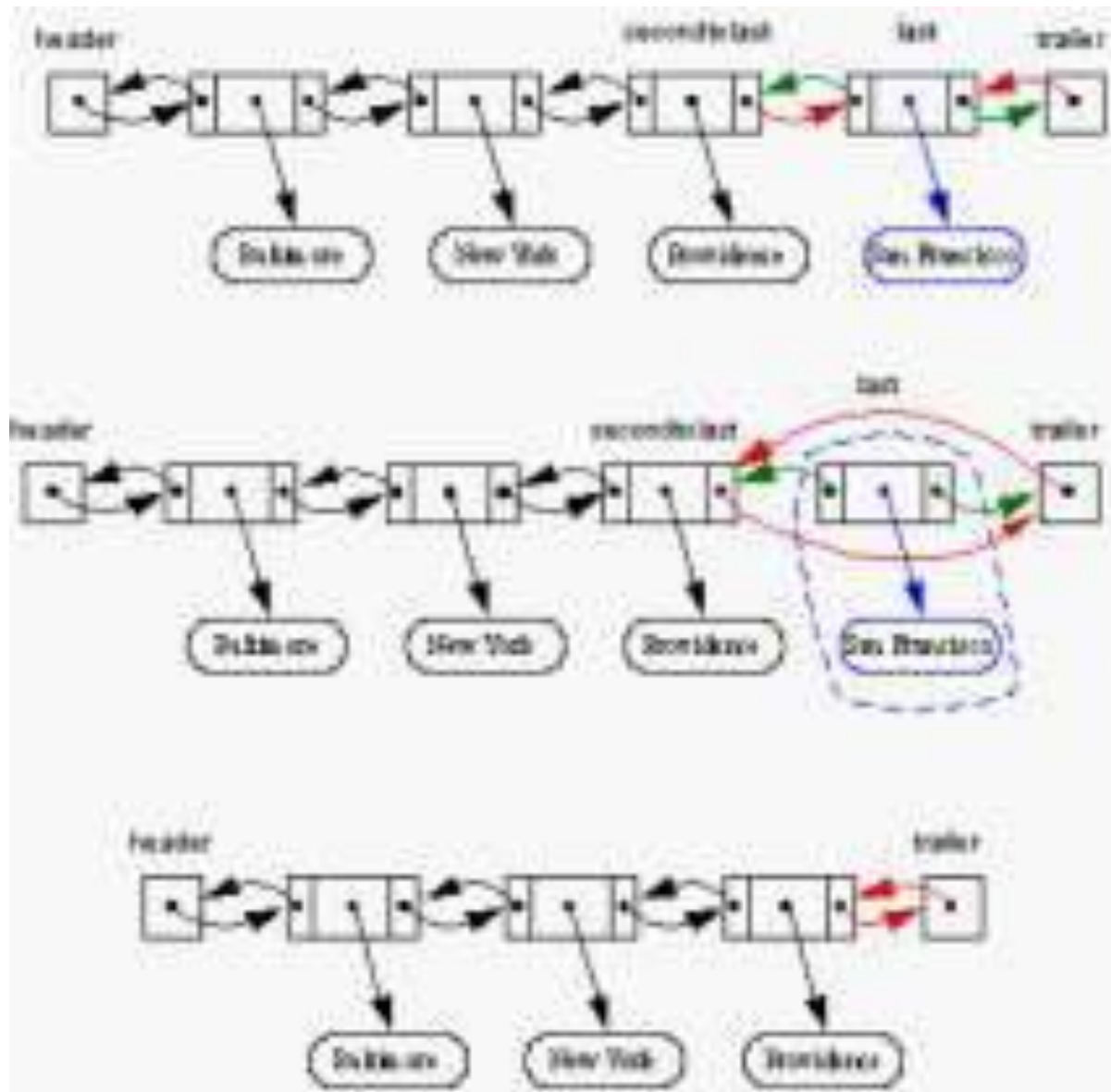
- Deque ADT: operations
- *addFirst(e): insert e at the beginning of the deque*
- *addLast(e): insert e at the end of the deque*
- *removeFirst(): remove and return the first element*
- *removeLast(): remove and return the last element*
- *getFirst(): return the first element*
- *getLast(): return the last element*
- *isEmpty(): return true if deque is empty; false otherwise*
- *size(): return the number of objects in the deque*

Implementation Choices

- Arrays
 - Similar to queue implementation (homework)
- Linked lists: singly or doubly linked?
 - Removing at the tail costs $\theta(n)$



removeLast() and addLast()



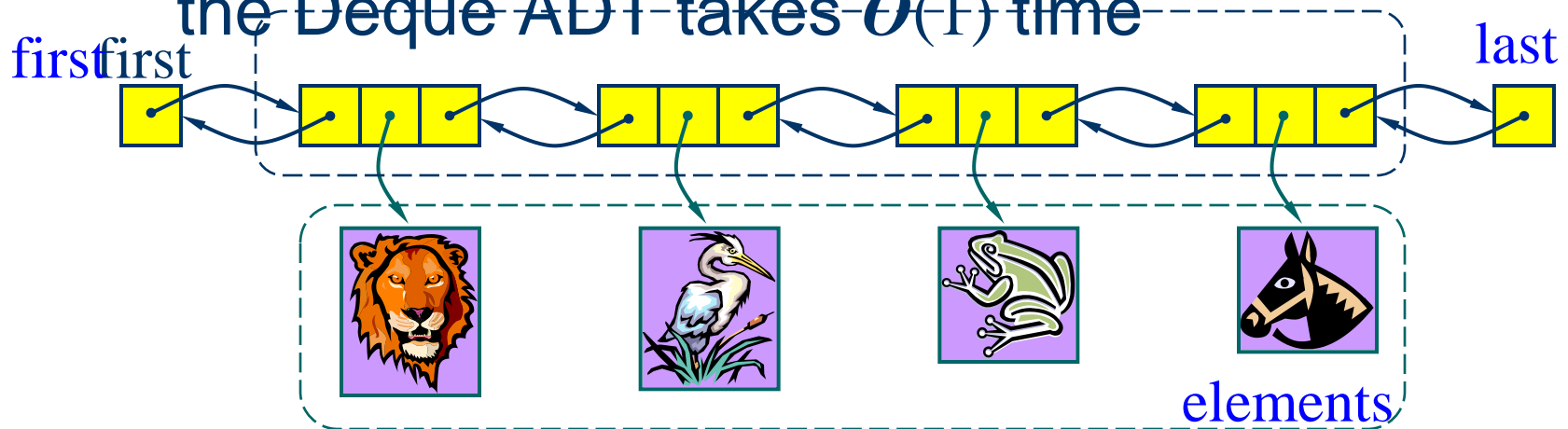
Implementing Stacks and Queues with Deques

Stack Method	Deque Implementation
size()	size()
isEmpty()	isEmpty()
top()	last()
push(e)	insertLast(e)
pop()	removeLast()

Queue Method	Deque Implementation
size()	size()
isEmpty()	isEmpty()
front()	first()
enqueue()	insertLast(e)
dequeue()	removeFirst()

Deque with a Doubly Linked List

- We can implement a deque with a doubly linked list
 - The front element is stored at the first node
 - The rear element is stored at the last node
- The space used is $O(n)$ and each operation of the Deque ADT takes $O(1)$ time



Performance and Limitations

- doubly linked list implementation of deque ADT

- Performance

- Let n be the number of elements in the queue
- The space used is $O(n)$
- Each operation runs in time $O(1)$

- Limitations

- NOTE: we do not have the limitation of the array based implementation on the size of the queue because the size of the linked list is not fixed, i.e., the deque is NEVER full.

Deque Summary

- Deque Operation Complexity for Different Implementations

	Array Fixed-Size	List Singly-Linked	List Doubly-Linked
removeFirst(), removeLast()	$O(1)$	$O(n)$ for one at list tail, $O(1)$ for other	$O(1)$
insertFirst(o), InsertLast(o)	$O(1)$	$O(1)$	$O(1)$
first(), last	$O(1)$	$O(1)$	$O(1)$
Size(), isEmpty()	$O(1)$	$O(1)$	$O(1)$

Application of Dequeue

- *A-Steal* job scheduling algorithm.
- This algorithm implements task scheduling for several processors.
- A separate deque with threads to be executed is maintained for each processor.
- To execute the next thread, the processor gets the first element from the deque (using the "remove first element" deque operation).
- If the current thread forks, it is put back to the front of the deque ("insert element at front") and a new thread is executed.
- When one of the processors finishes execution of its own threads (i.e. its deque is empty), it can "steal" a thread from another processor: it gets the last element from the deque of another processor ("remove last element") and executes it

Priority Queues

- A priority queue is a collection of zero or more elements → each element has a **priority** or value
- Unlike the FIFO queues, the **order** of deletion from a priority queue (**e.g., who gets served next**) is **determined by the element priority**
- Elements are deleted by increasing or decreasing order of priority rather than by the order in which they arrived in the queue

The Priority Queue Abstract Data Type

Suppose that you have a few assignments from different courses. Which assignment will you want to work on first?

Course	Priority	Due day
Database Systems	2	October 3
UNIX	4	October 10
Data Structure & Algorithm	1	September 29
Structured Systems Analysis	3	October 7

You set your priority based on due days. Due days are called keys.

When you want to determine the priority for your assignments, you need a value for each assignment, that you can compare with each other.

key: An object that is assigned to an element as a specific attribute for that element, which can be used to identify, rank, or weight that element.

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

PRIORITY QUEUES

- The priority queue is a data structure in which intrinsic ordering of the elements determines the results of its basic operations.
- An ***ascending priority queue*** is a collection of items into which items can be inserted arbitrarily and from which only the smallest item can be removed.
- On the other hand a ***descending priority queue*** allows only the largest item to be removed.

Priority QUEUE Operations

- **Insertion**

- The insertion in Priority queues is **the same as** in non-priority queues.

- **Deletion**

- Deletion requires a search for the element of highest priority and deletes the element with highest priority. The following methods can be used for deletion/removal from a given Priority Queue:
 - An empty indicator replaces deleted elements.
 - **After each deletion elements can be moved up in the array decrementing the rear.**
 - The array in the queue can be maintained as an ordered circular array

Priority Queue Declaration

- ***Queue data type of Priority Queue is the same as the Non-priority Queue.***

```
#define MAXQUEUE 10 /* size of the queue items*/  
typedef struct {  
    int front, rear;  
    int items[MAXQUEUE];  
}QUEUE;
```

REMOVE OPERATION

- ***PriQremove*** Operation using removing the element with highest priority and shifting the elements up in the array and decrementing rear. Consider Ascending Priority Queue.

```
int PRiQremove(QUEUE *qptr)
{
    int smallest, loc, f, i;
    f=qptr->front;
    if(qptr->front == qptr->rear){
        printf("Queue underflow");
        exit(1);
    }
    smallest = qptr->items[(qptr->front+1)%MAXQUEUE];
    loc = (qptr->front+1)%MAXQUEUE;
    (qptr->front++)%MAXQUEUE; /* Circular increment*/
    while(qptr->front != qptr->rear){
        if(qptr->items[(qptr->front+1)%MAXQUEUE] <smallest){
            smallest = qptr->items[(qptr->front+1)%MAXQUEUE];
            loc = (qptr->front+1)%MAXQUEUE;
        }
        qptr->front =(qptr->front+1)%MAXQUEUE; /* Circular inc.*/
    }
}
```



```
while(loc != qptr->rear){  
    qptr->items[loc] = qptr->items[(loc+1)%MAXQUEUE];  
    (loc++)%MAXQUEUE;  
}  
qptr->front=f;  
if(qptr->rear == 0) /*Decrement rear after removing one item*/  
    qptr->rear = MAXQUEUE - 1;  
else  
    qptr->rear--;  
return smallest;  
}
```

Insert Operation of Priority Queue is the same as the insert of the non-priority queues.

```
void insert(struct queue *qptr, int x)
{
    qptr->rear = (qptr->rear++)%MAXQUEUE; /*Circular increment*/
    if(qptr->rear == qptr->front){
        printf("Queue overflow");
        exit(1);
    }
    qptr->items[qptr->rear]=x;
}
```

Priority queue implementation

```
void enqueue(int X)
{
    if ( ! isQfull() )
    {
        for(i = rear; i >= front; i--)
            if ( Queue[i] < X)
                break;
        else
            Queue[i+1] = Queue[i];
        Queue [ i + 1 ] = X;
        rear = rear + 1;
    }
    else
        print queue is full;
}
```

```
int dequeue()
{
    if ( ! isQempty() )
    {
        front = front + 1;
        x = Queue[front]
        return x;
    }
    else
        return an error key ;
}
```

Priority queue implementation

```
void enqueue(int X)
{
    if ( ! isQfull(rear) )
    {
        rear = rear + 1;
        Queue[rear] = X;
    }
    else
        print queue is full;
}
```

```
int dequeue()
{
    if ( ! isQempty() )
    {
        max = front;
        for(i=front + 1; i <= rear; i++)
            if( Queue[i] > Queue[max]
                max = i;
        X = Queue[max];
        Swap( Queue[max],Queue[front])
        front = front + 1;
        return x;
    }
    else
        return an error key ;
}
```

Efficient 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

Priority Queues : Applications

- Used in multitasking operating system.
- They are generally represented using “heap” data structure.
- Insertion runs in $O(n)$ time, deletion in $O(1)$ time.

Application : Process control systems

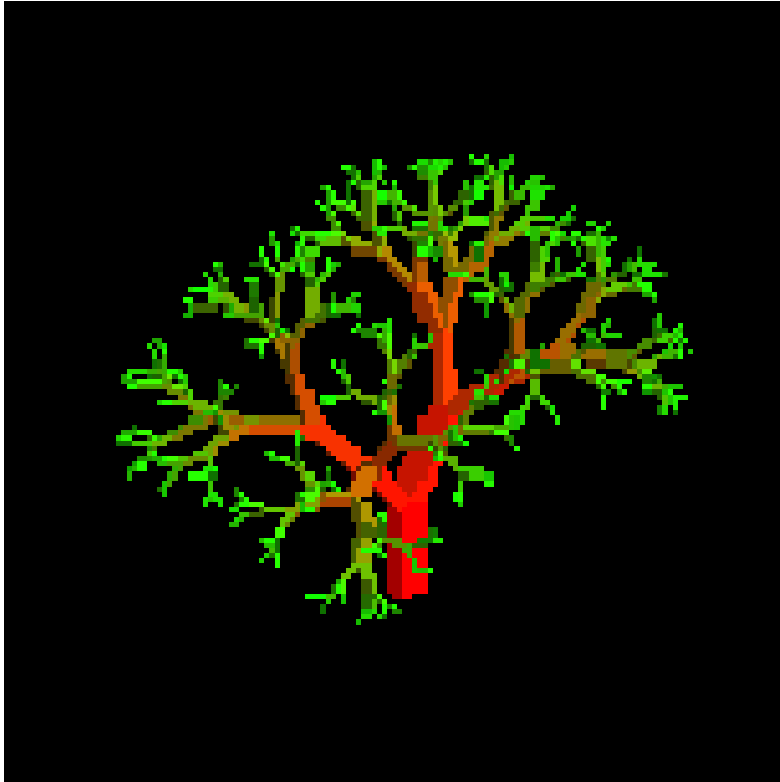
Imagine the operator's console in a large automated factory. It receives many routine messages from all parts of the system: they are assigned a low priority because they just report the normal functioning of the system - they update various parts of the operator's console display simply so that there is some confirmation that there are no problems. It will make little difference if they are delayed or lost.

However, occasionally something breaks or fails and alarm messages are sent. These have high priority because some action is required to fix the problem (even if it is mass evacuation because nothing can stop the imminent explosion!).

Typically such a system will be composed of many small units, one of which will be a buffer for messages received by the operator's console. The communications system places messages in the buffer so that communications links can be freed for further messages while the console software is processing the message. The console software extracts messages from the buffer and updates appropriate parts of the display system. Obviously we want to sort messages on their priority so that we can ensure that the alarms are processed immediately and not delayed behind a few thousand routine messages while the plant is about to explode.

Trees

How We View a Tree



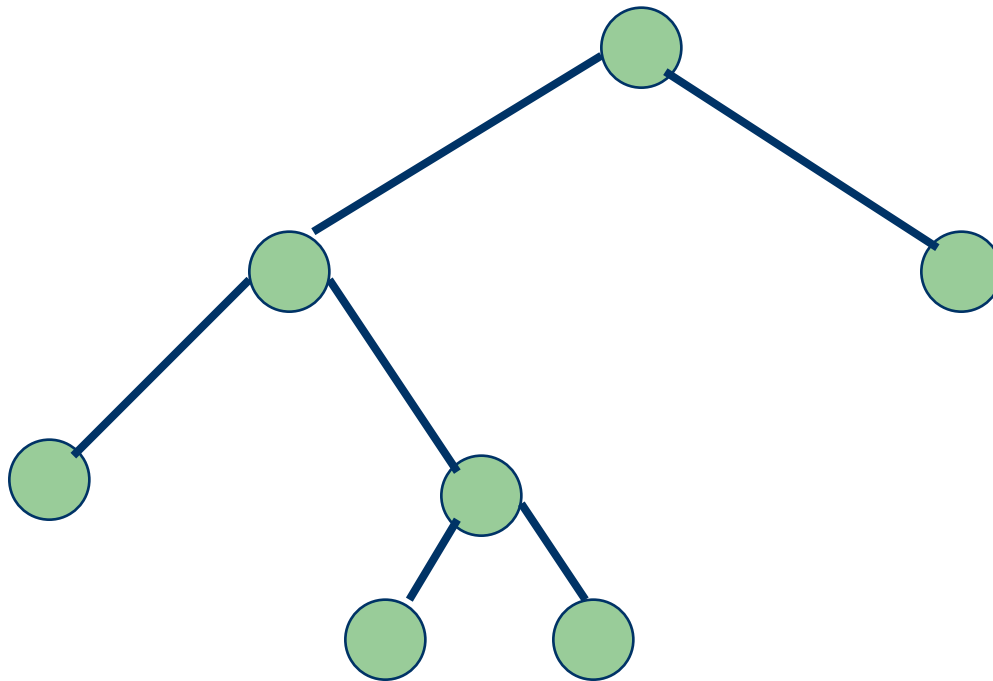
Nature Lovers View



Computer Scientists View

Strictly Binary Tree

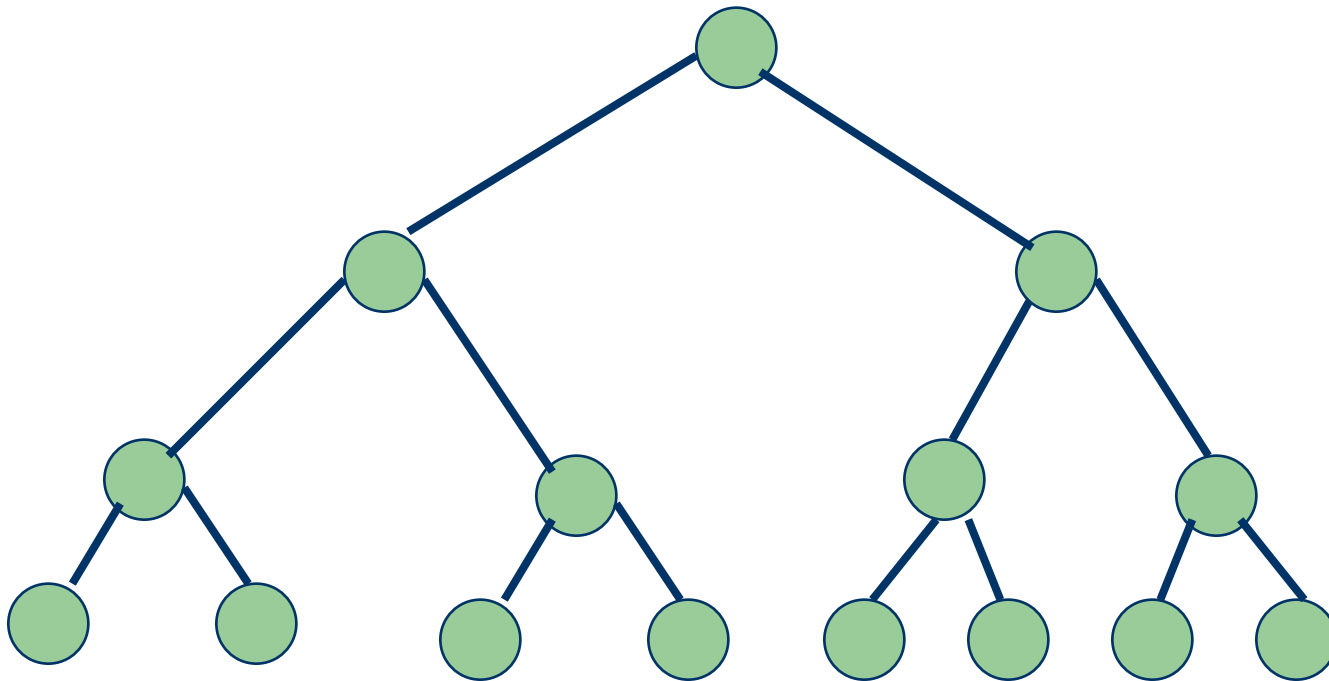
- Is a binary tree where all non leaf nodes have two branches.



Height 4 full binary tree - 15 nodes

Full Binary Tree

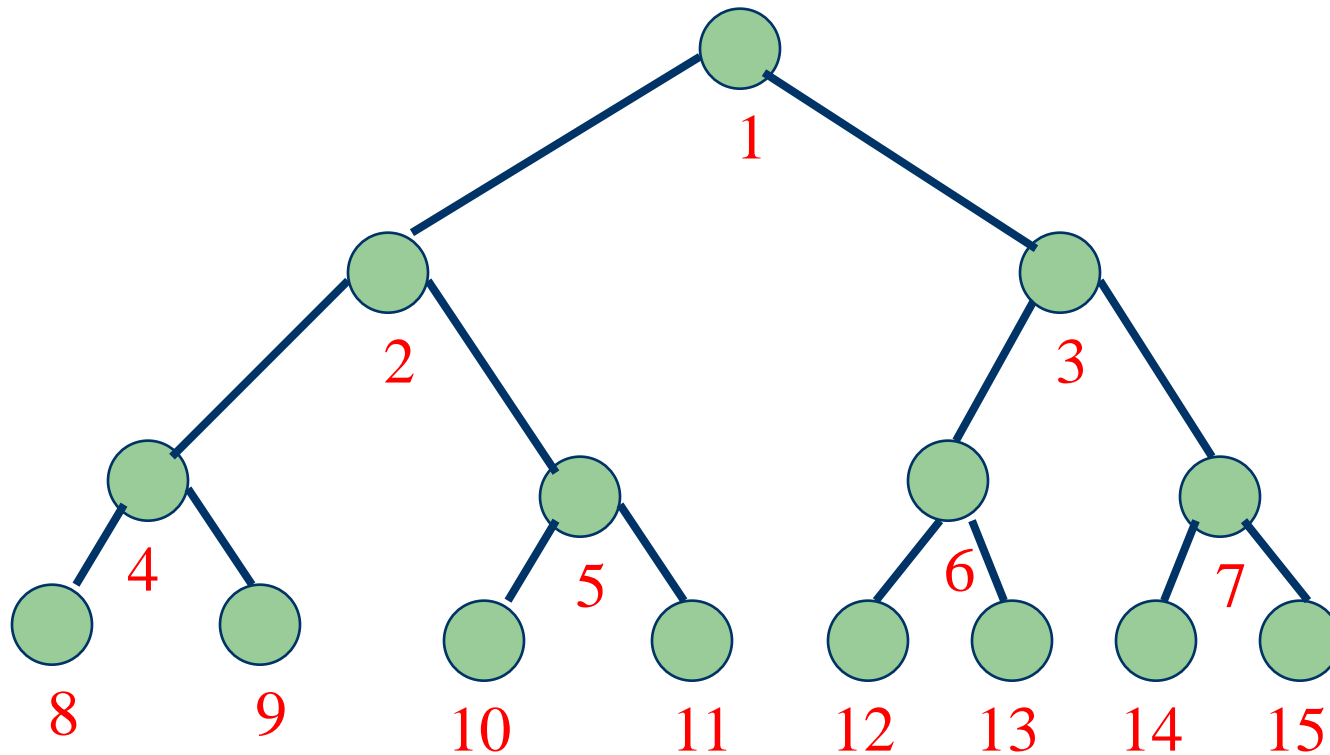
- A full binary tree of a given height h has $2^h - 1$ nodes



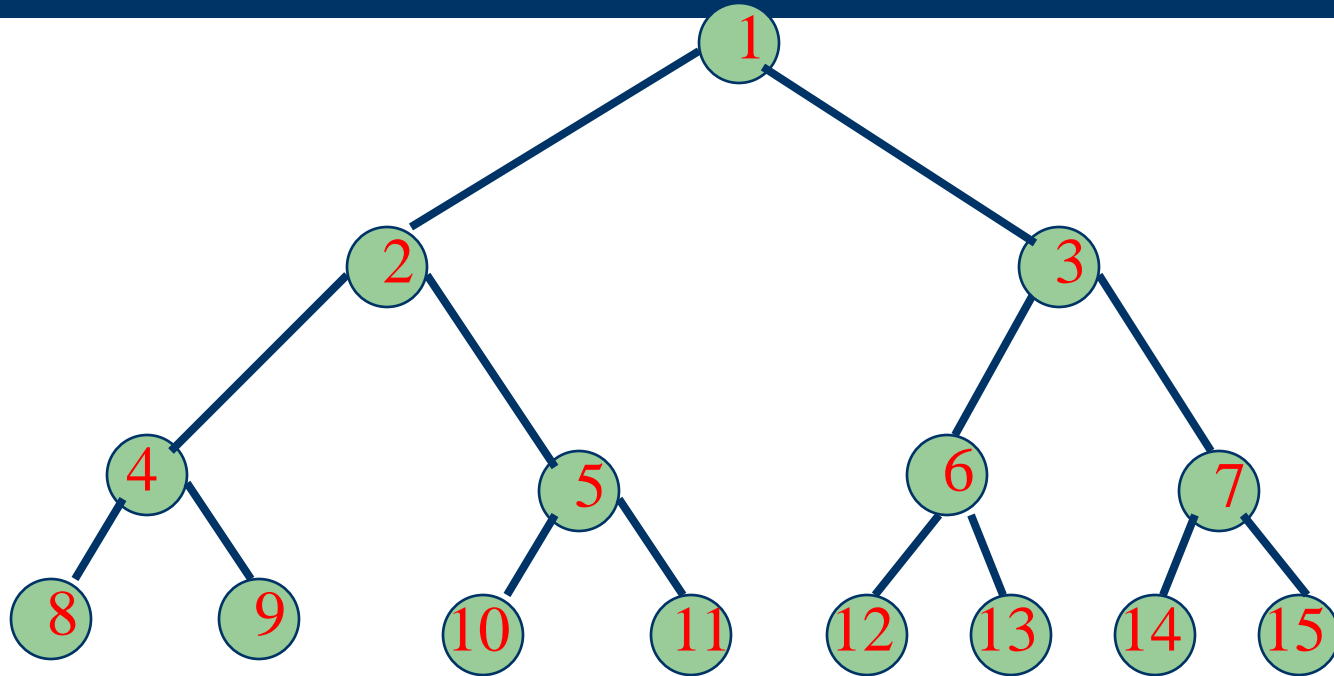
Height 4 full binary tree - 15 nodes

Numbering Nodes In Full Binary Tree

- Number the nodes **1** through $2^h - 1$
- Number by levels from top to bottom
- Within a level number from left to right

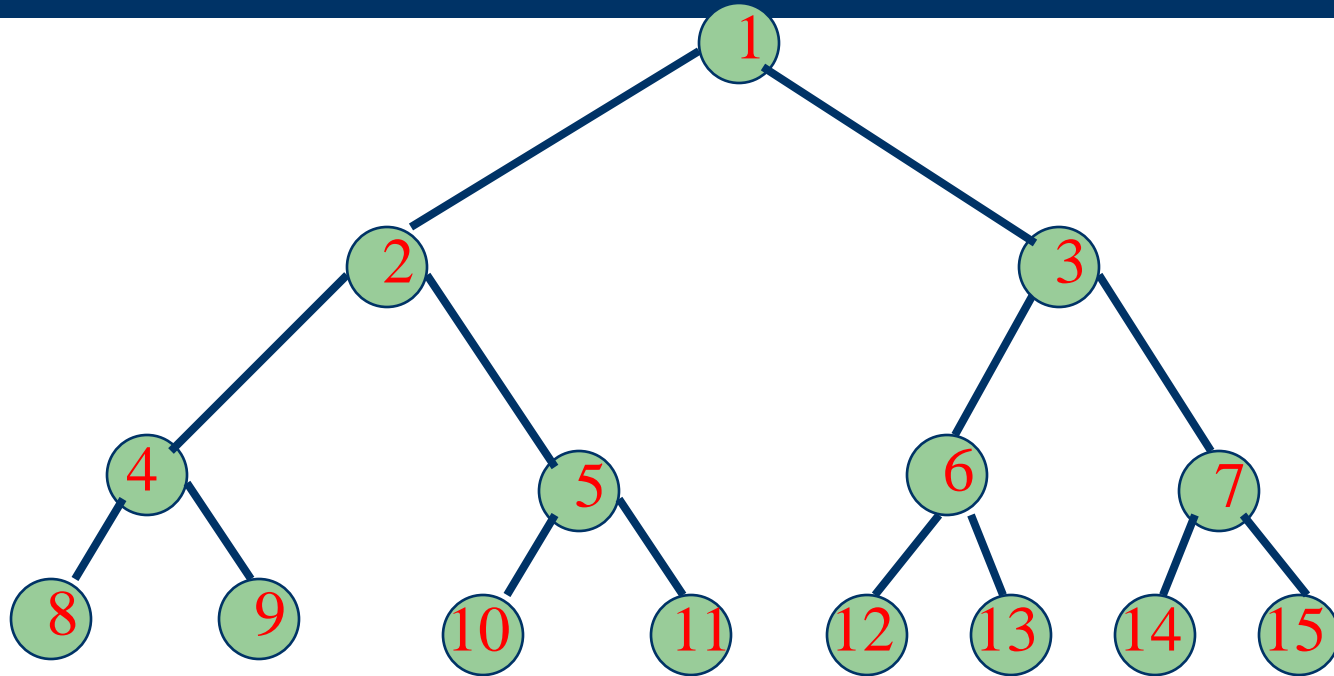


Node Number Properties



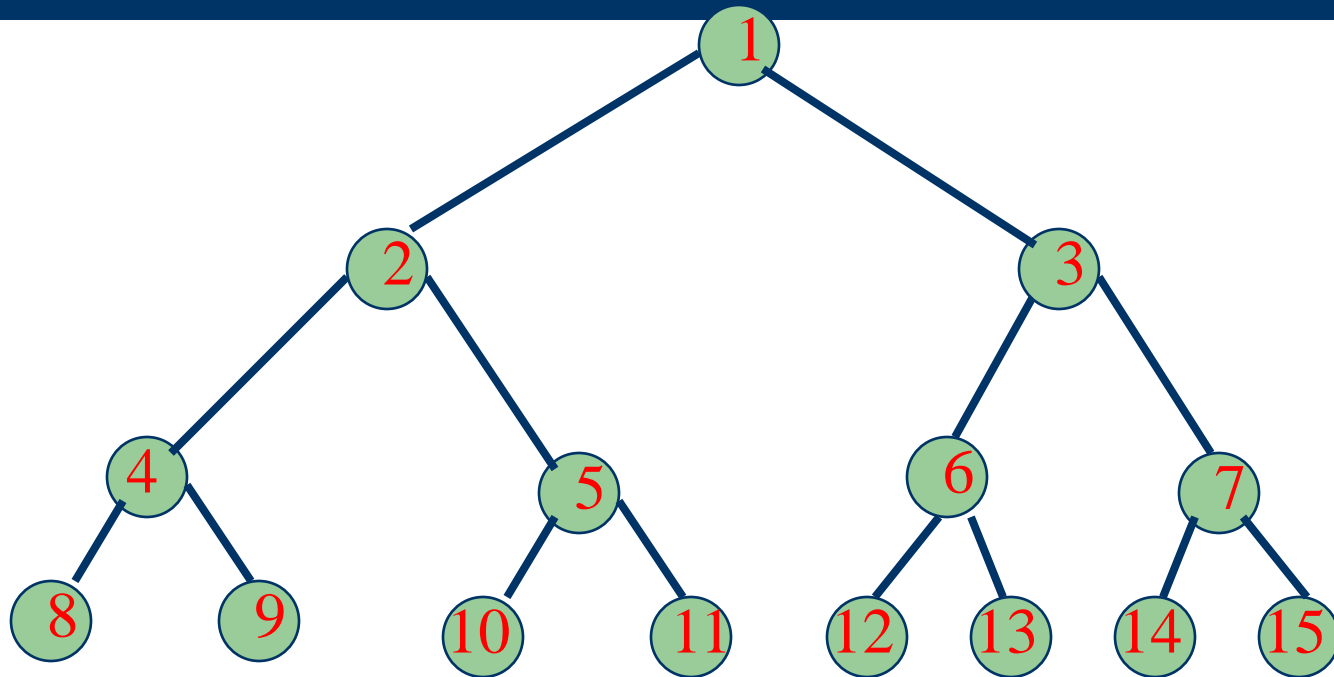
- Parent of node i is node $i / 2$, unless $i = 1$
- Node 1 is the root and has no parent

Node Number Properties



- Left child of node i is node $2i$, unless $2i > n$, where n is the number of nodes
- If $2i > n$, node i has no left child

Node Number Properties

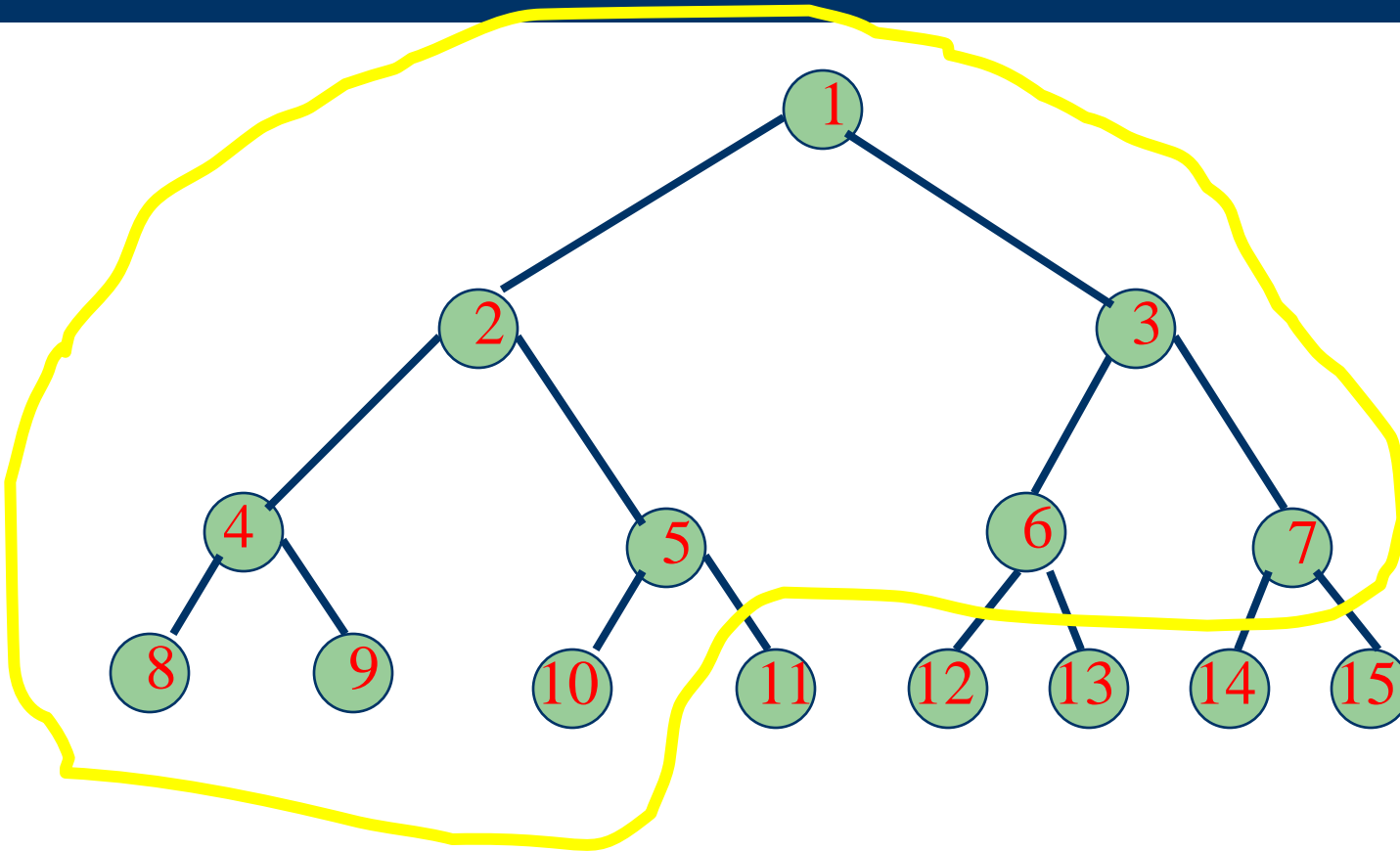


- Right child of node i is node $2i+1$, unless $2i+1 > n$, where n is the number of nodes
- If $2i+1 > n$, node i has no right child

Complete Binary Tree With n Nodes

- Start with a full binary tree that has at least n nodes
- Number the nodes as described earlier
- The binary tree defined by the nodes numbered 1 through n is the unique n node complete binary tree

Example



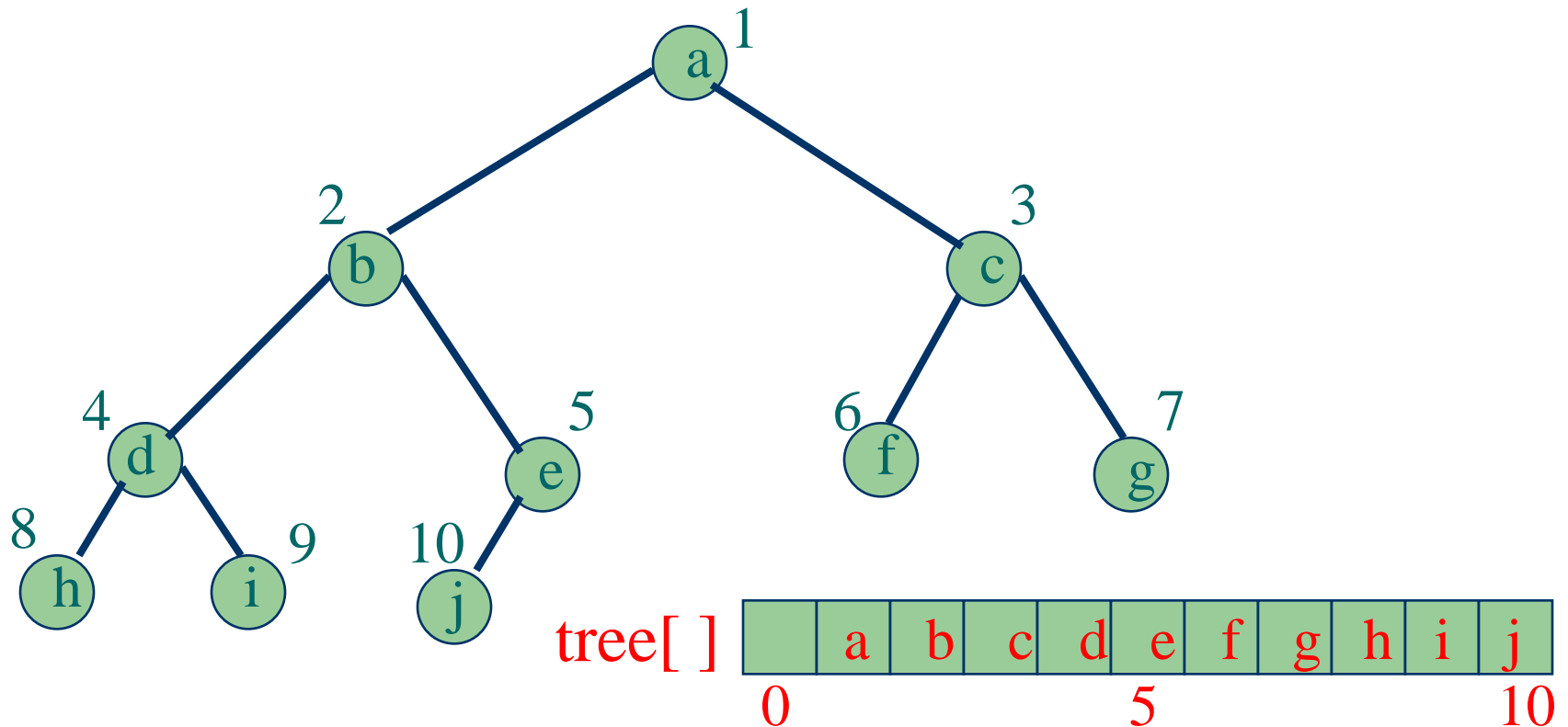
- Complete binary tree with 10 nodes

Binary Tree Representation

- Again, there are two ways to implement a tree data structure:
 - Array representation
 - Linked representation

Array Representation

- Number the nodes using the numbering scheme for a full binary tree
- The node that is numbered i is stored in $tree[i]$

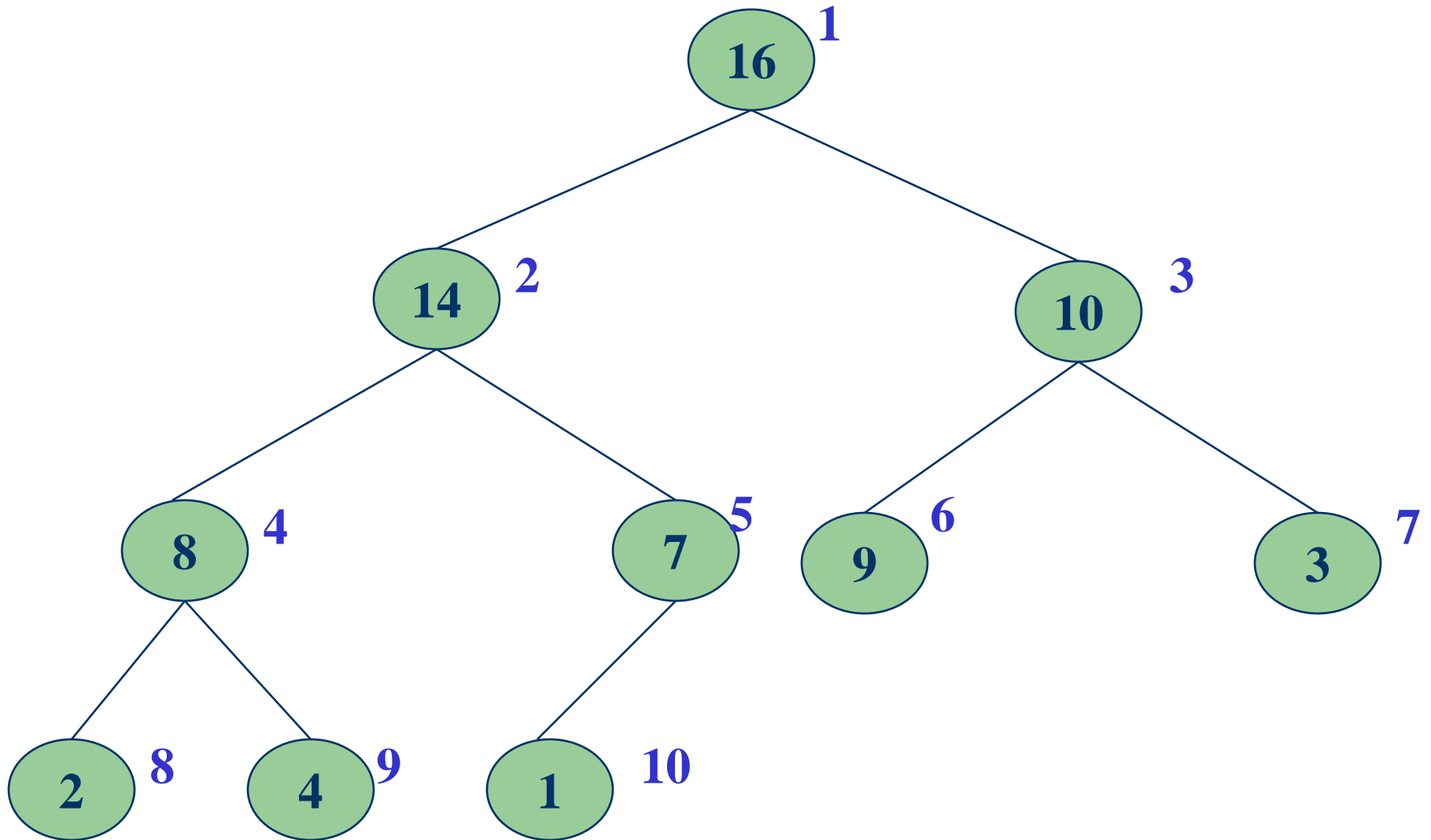


Heap

- An array A that represents a heap is an object with two attributes : length_A , which is the number of elements in the array.
- And heapSize_A , the number of elements in the heap stored within array A .
- That is, although $A[1 \dots \text{length}_A]$ may contain valid numbers, no element past $A[\text{heapSize}_A]$, where $\text{heapSize}_A \leq \text{length}_A$, is an element of the heap.

A max heap viewed as binary tree and an array

1	2	3	4	5	6	7	8	9	10
16	14	10	8	7	9	3	2	4	1



Types

- Max-heaps and Min-heaps
- In both kinds the values in the nodes satisfy a heap property, the specifics of which depend on the kind of heap.
- In a max-heap, the max-heap property is that for every node other than the root,
 - $A[\text{Parent}(i)] \geq A[i]$
- In a min-heap, the min-heap property is that for every node other than the root,
 - $A[\text{Parent}(i)] \leq A[i]$

Basic Procedures

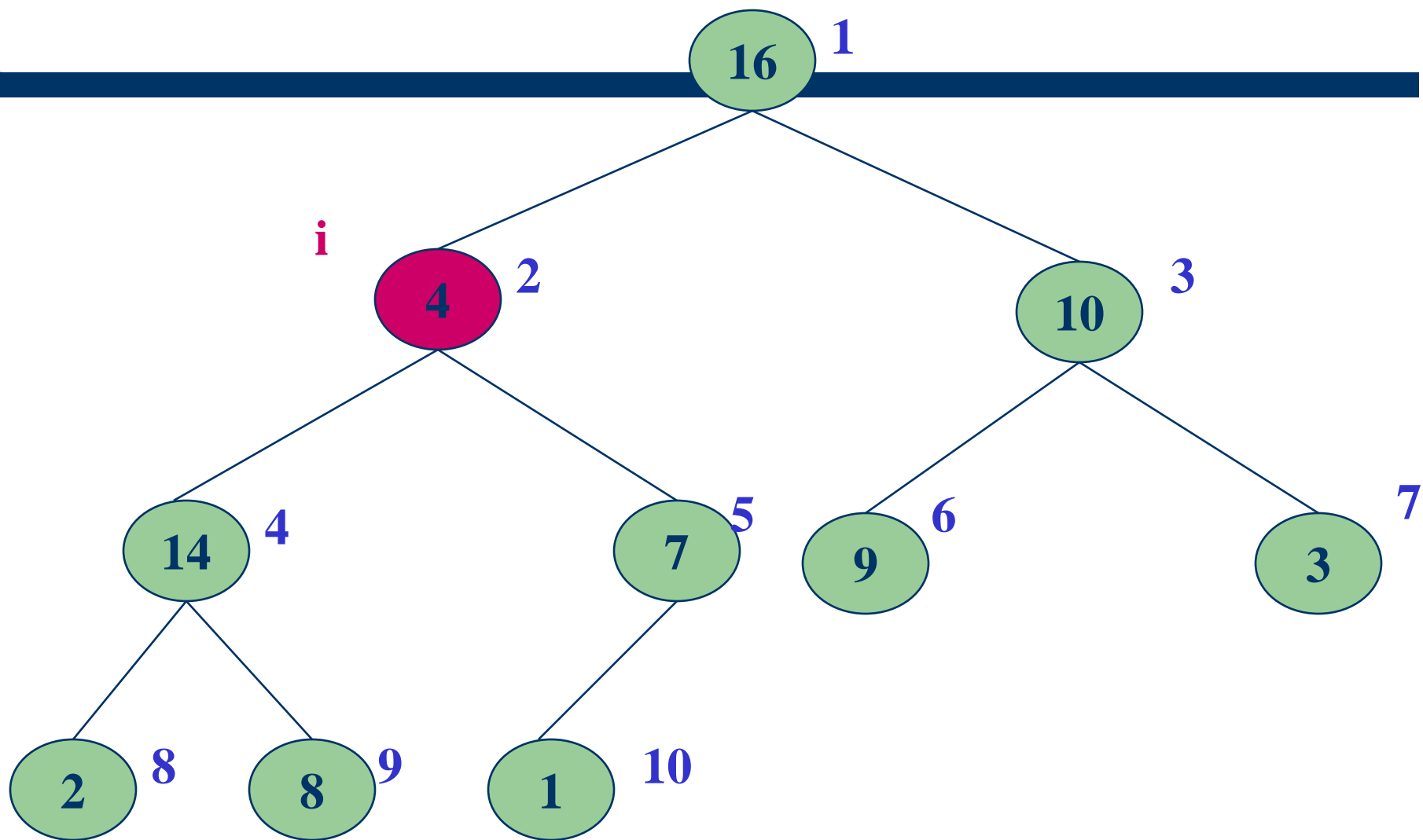
- The MAX-HEAPIFY procedure, which runs in $O(\lg n)$ time, is the key to maintaining the max-heap property.
- The BUILD-MAX-HEAP procedure, which runs in linear time, produces a max-heap from a n unordered input array.
- The MAX-HEAP-INSERT, HEAP-EXTRACT-MAX procedures, which runs in $O(\lg n)$ time, allow the heap data structure to be used as a priority queue.

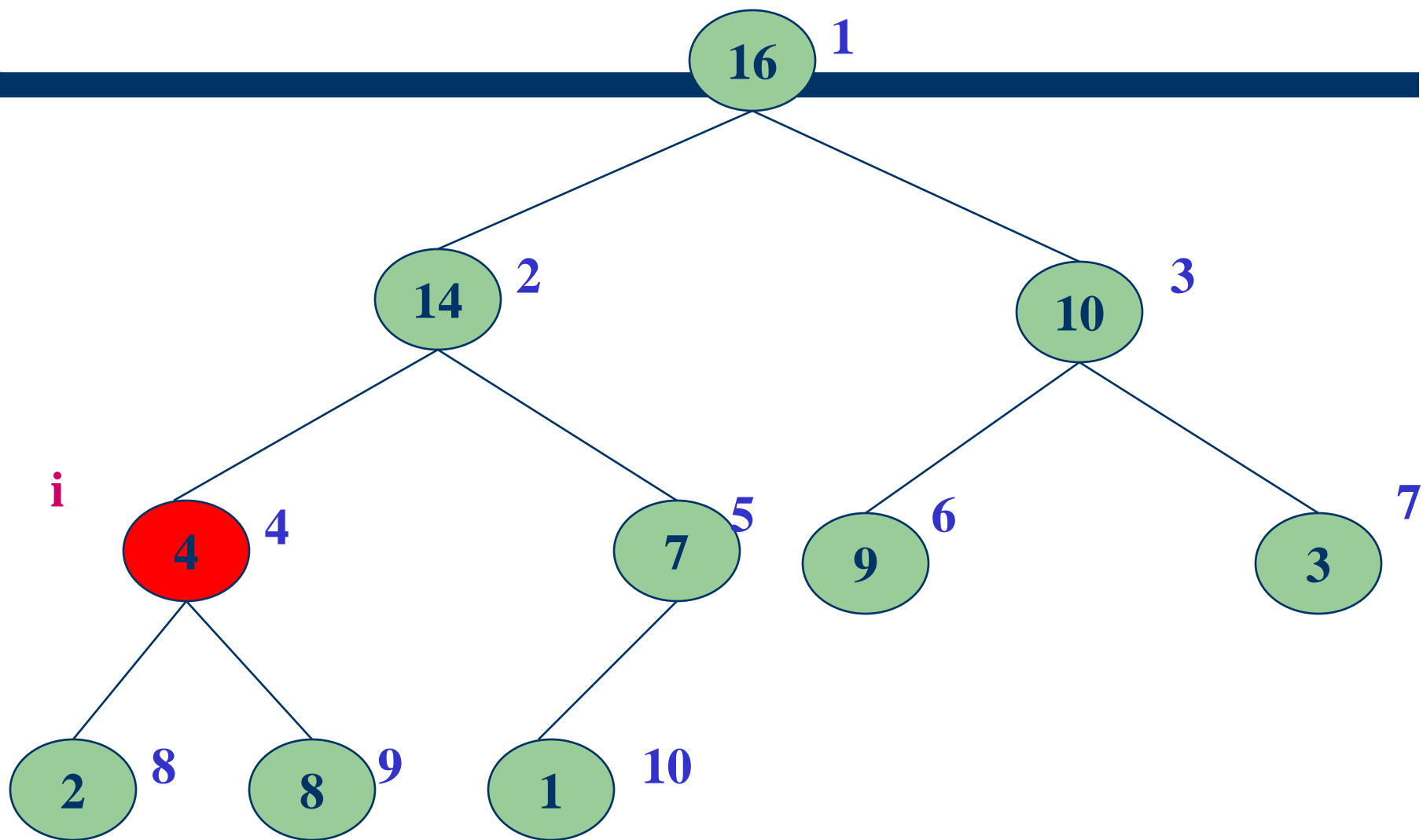
Maintaining the heap property

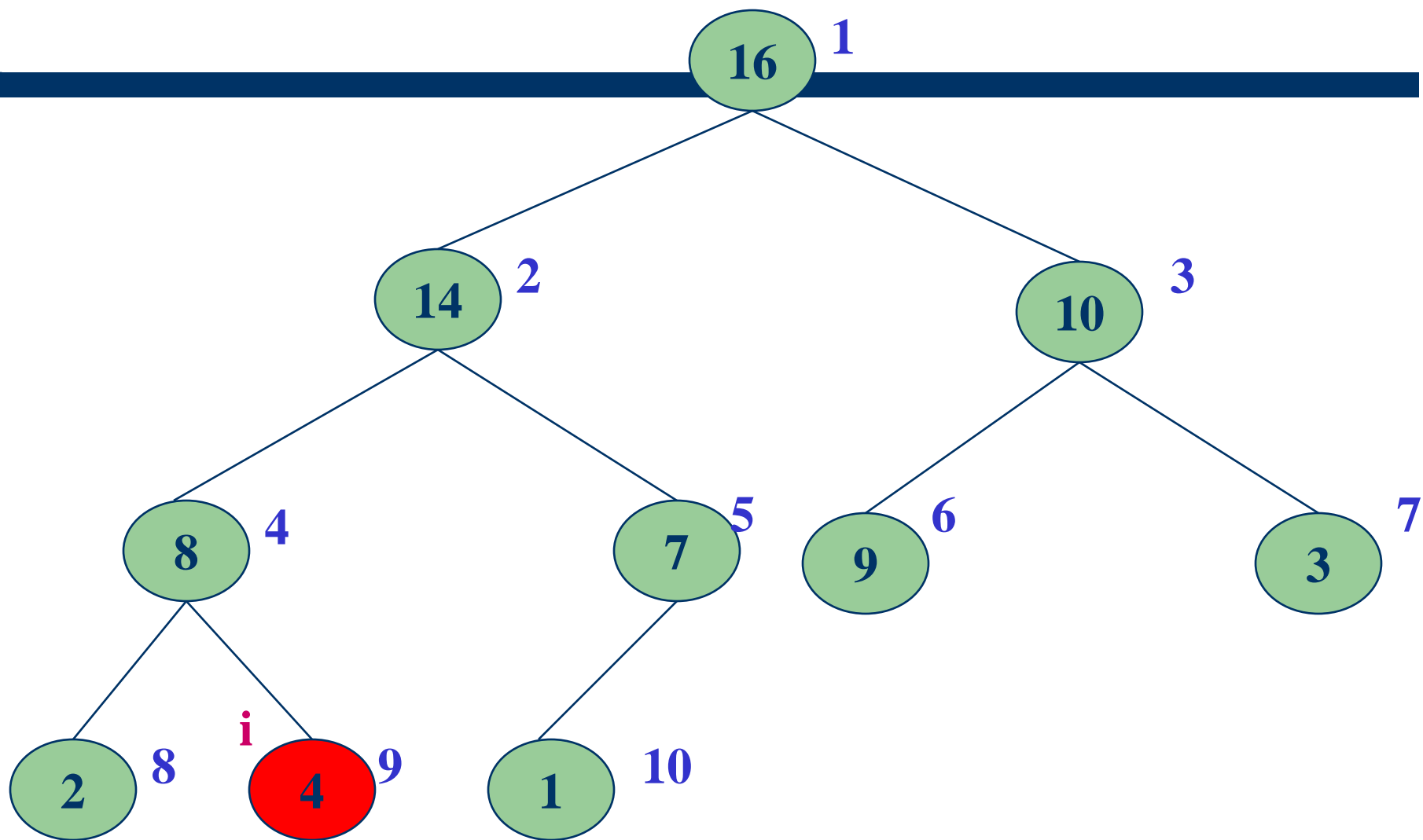
- MAX-HEAPIFY is an important subroutine for manipulating max-heaps.
- Its input are an array A and an index i into the array.
- When MAX-HEAPIFY is called, it is assumed that the binary trees rooted as $\text{Left}(i)$ and $\text{Right}(i)$ are max-heaps, but the $A[i]$ may be smaller than its children, thus violating the max-heap property.
- The function of MAX-HEAPIFY is to let the value at $A[i]$ float down in the max-heap so that the subtree rooted at index i becomes a max-heap.

MAX_HEAPIFY (A , i)

1. $l = \text{Left}(i)$
2. $r = \text{Right}(i)$
3. If $l \leq \text{heapSize}_A$ and $A[l] > A[i]$
4. then $\text{largest} \leftarrow l$
5. else $\text{largest} \leftarrow i$
6. If $r \leq \text{heapSize}_A$ and $A[r] > A[\text{largest}]$
7. then $\text{largest} \leftarrow r$
8. If $\text{largest} \neq i$
9. then Exchange $A[i] \leftrightarrow A[\text{largest}]$
10. MAX-HEAPIFY(A, largest)







Building a heap

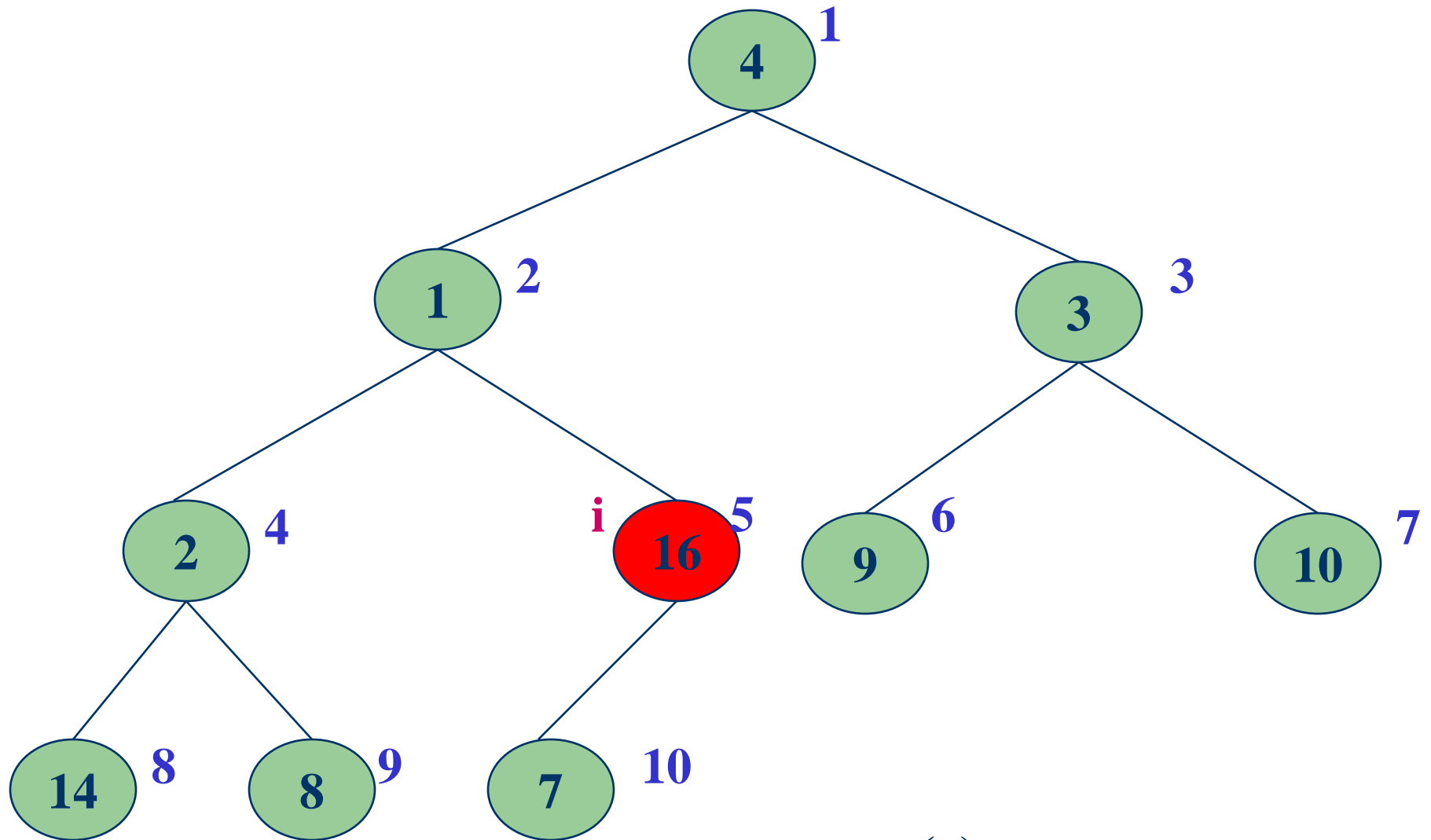
- We can use the procedure MAX-HEAPIFY in a bottom-up manner to convert an array $A[1 \dots n]$, where $n = \text{length}_A$, into a max-heap.
- The Elements in the subarray $A[(\lfloor n/2 \rfloor + 1) \dots n]$ are all leaves of the tree and so each is 1-element heap to begin with.
- The procedure BUILD-MAX_HEAP goes through the remaining nodes of the tree and runs MAX-HEAPIFY on each one.

BUILD_MAX_HEAP(A)

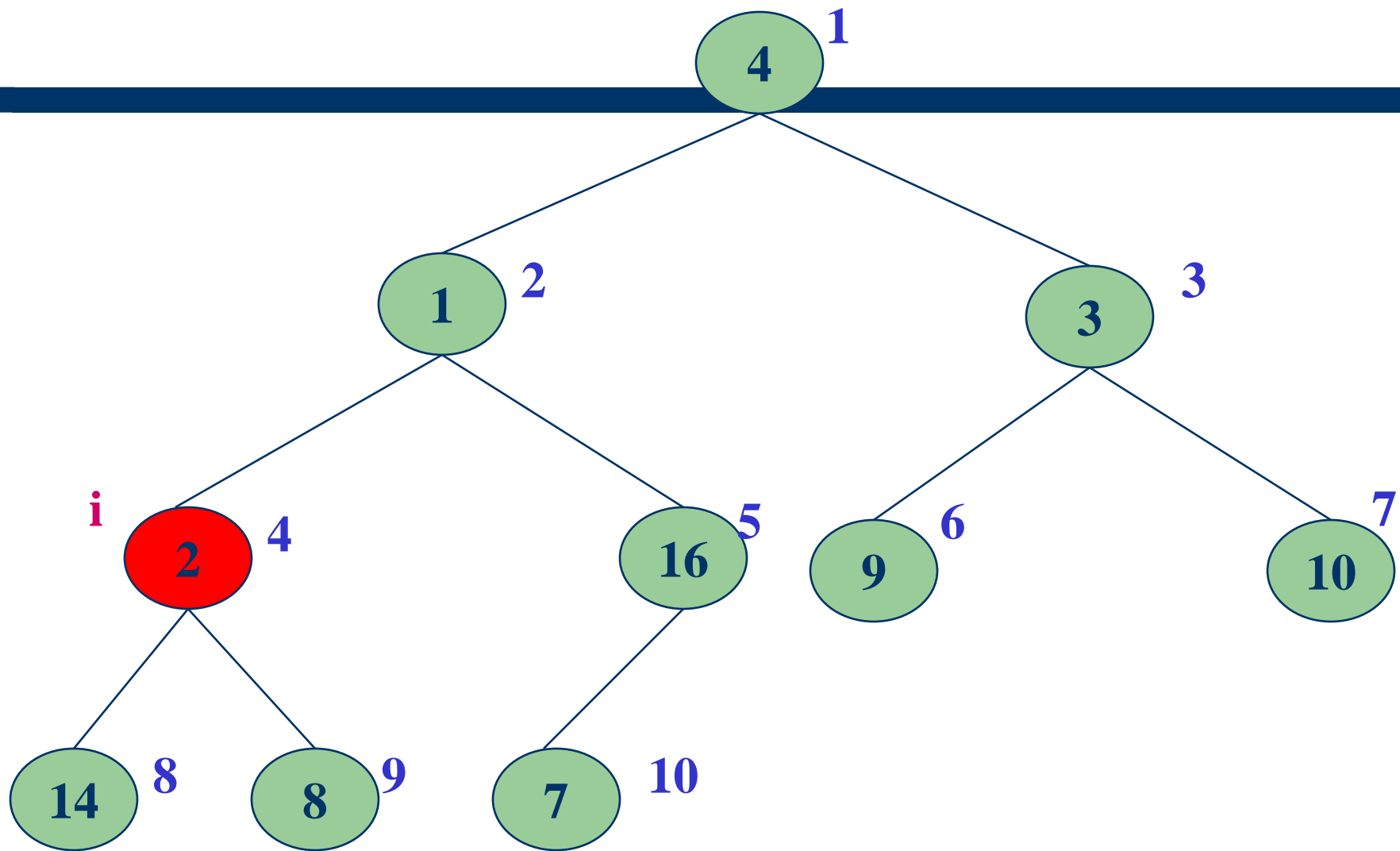
1. $\text{heapsize}_A \leftarrow \text{length}_A$
2. for $i \leftarrow \lfloor \text{length}_A / 2 \rfloor$ down to 1
3. do MAX_HEAPIFY(A,i)

Build heap example

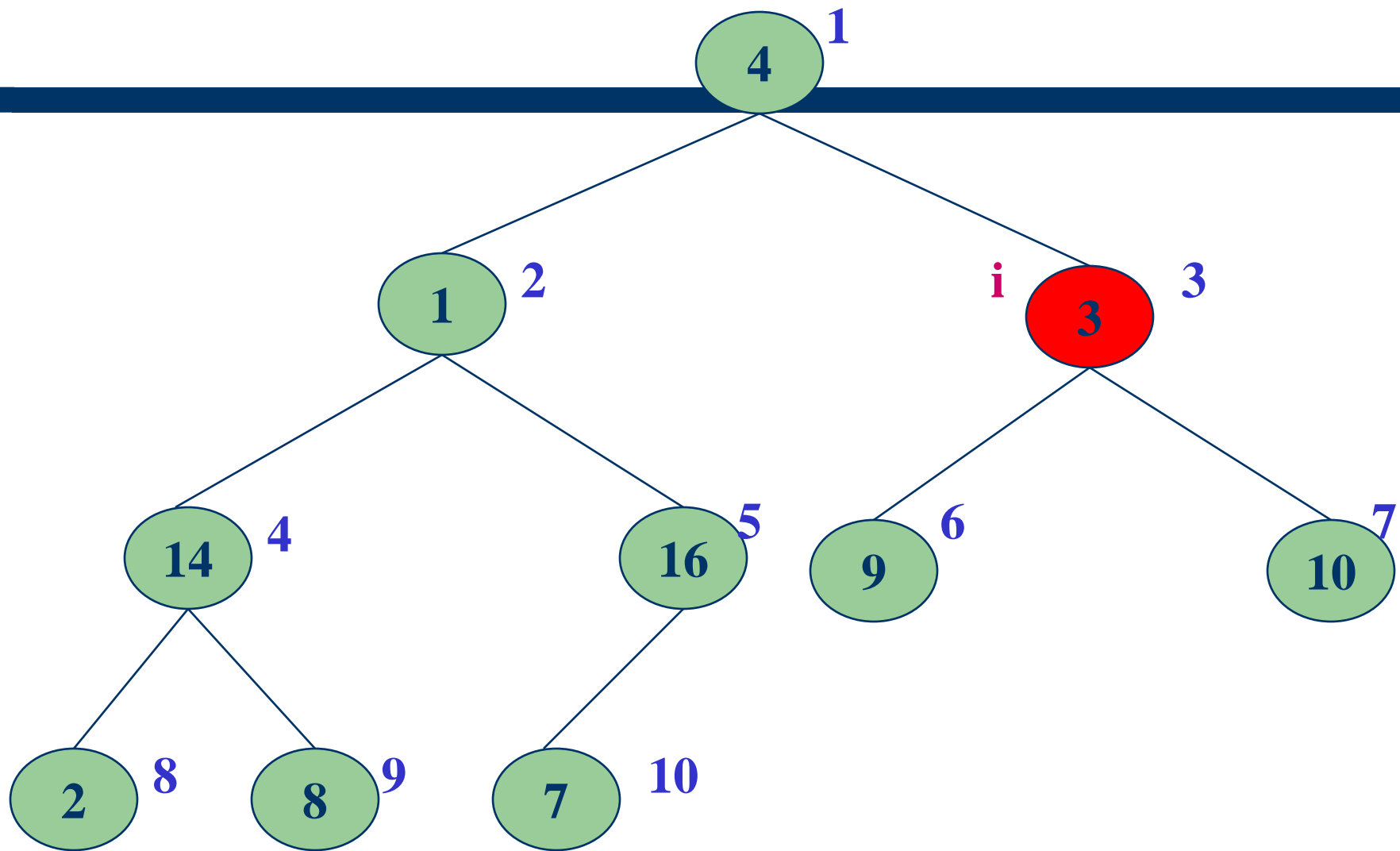
1	2	3	4	5	6	7	8	9	10
4	1	3	2	16	9	10	14	8	7



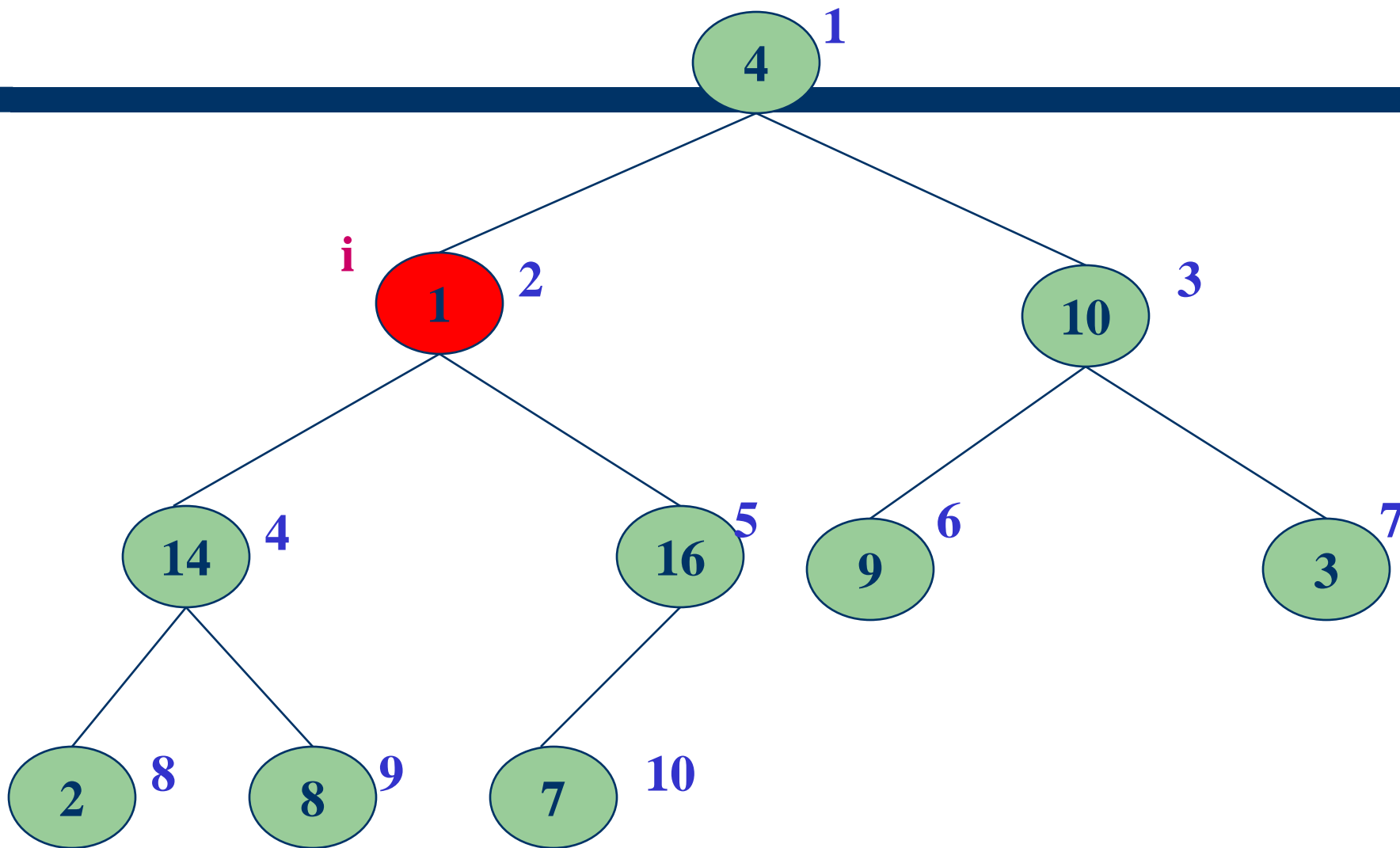
(a)



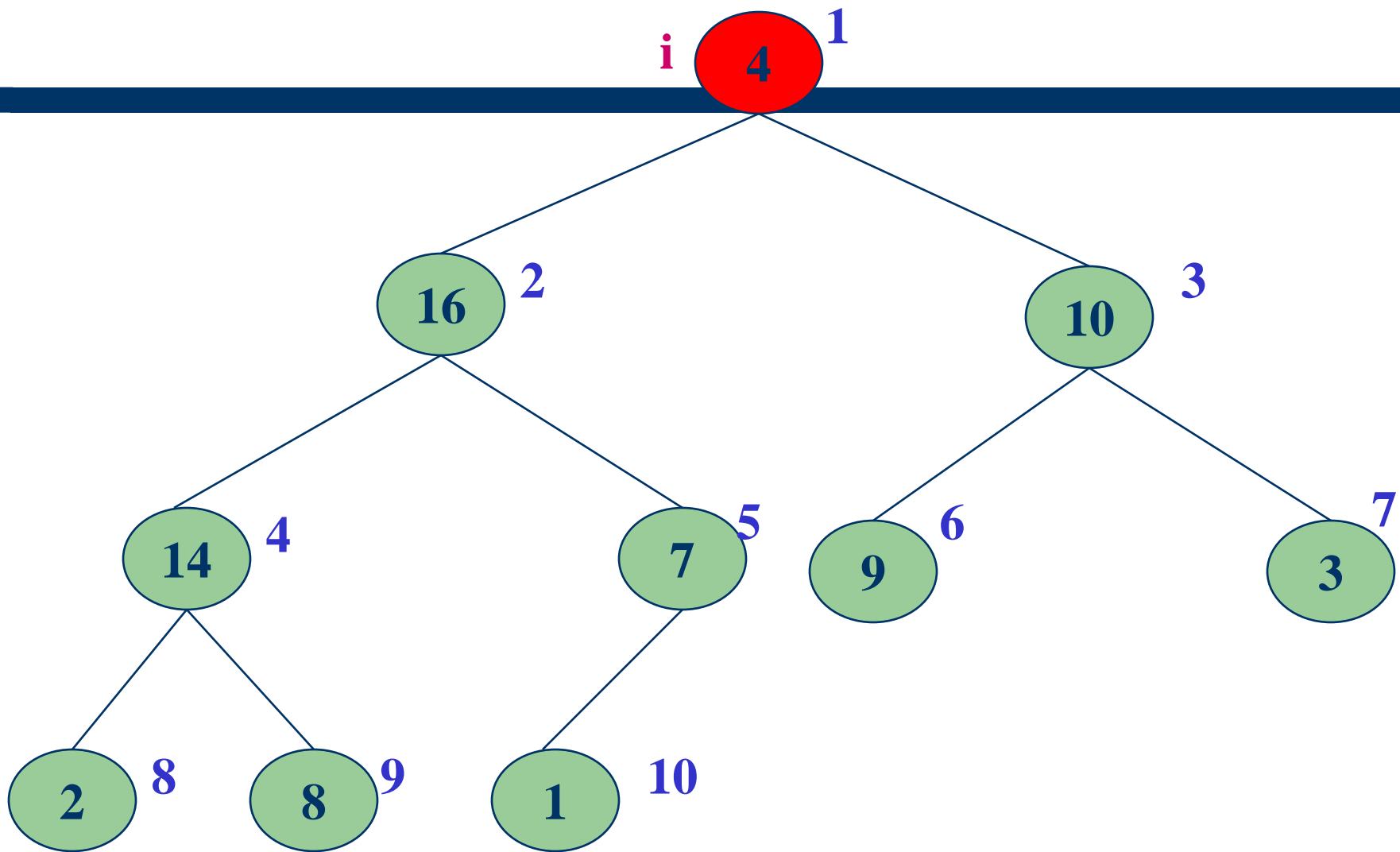
(b)



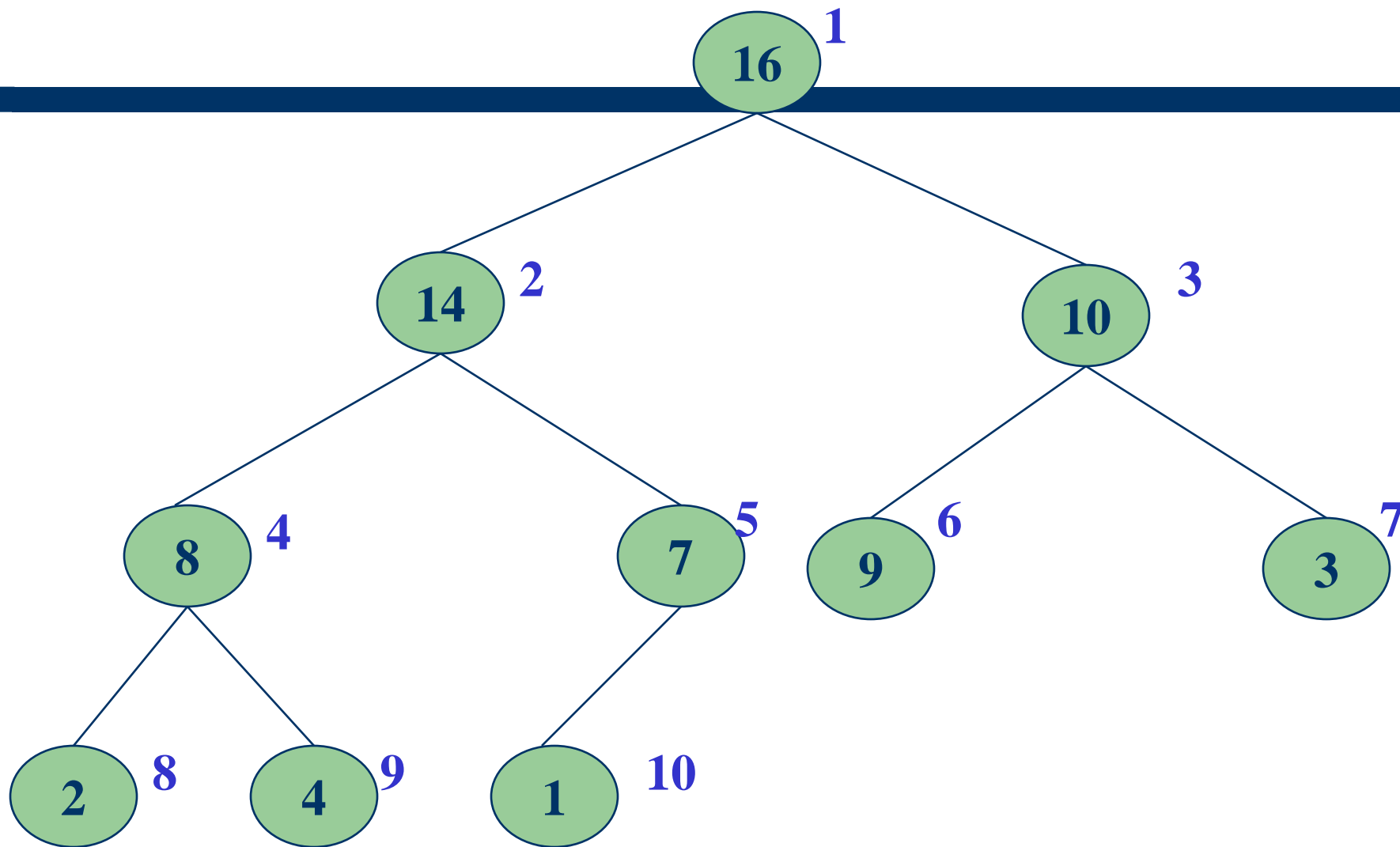
(c)



(d)



(e)



(f)

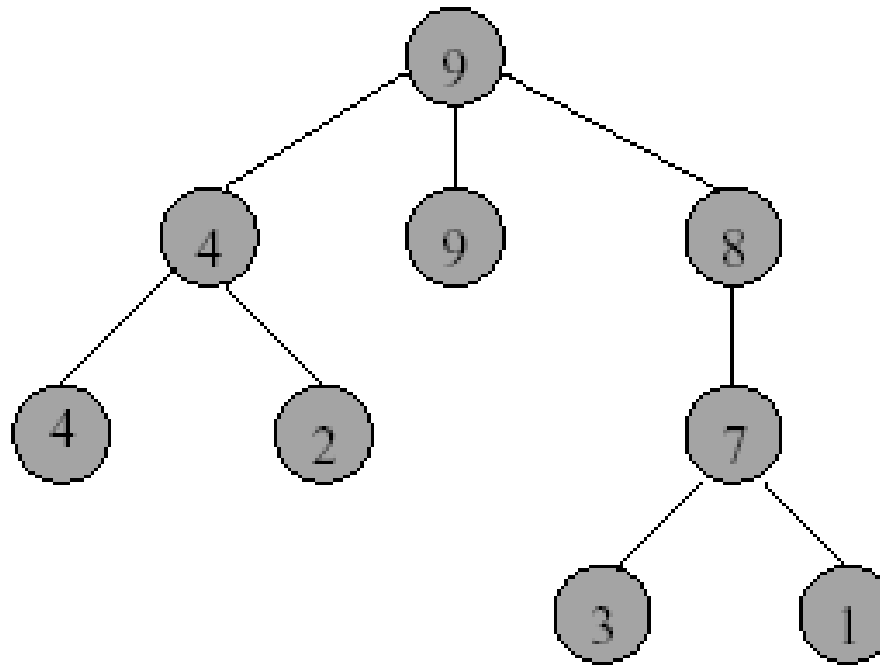
Priority Queues

- Most popular application of heap, its use as an efficient priority queue.
- A priority queue is a data structure for maintaining a set S of elements, each with an associated value called a key.
- Operations
 - $\text{INSERT}(S, x)$ inserts the element x into the Set S .
 - $\text{EXTRACT_MAX}(S)$ removes and returns the element of S with the largest key.
 - $\text{MAXIMUM}(S)$ returns the element of S with the largest key.

Max (Min) Tree

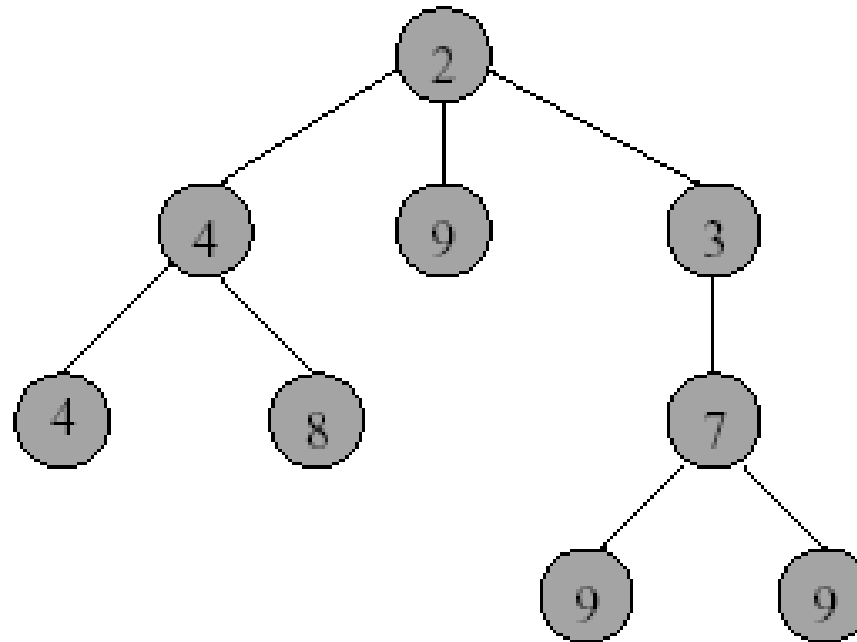
- A max tree (min tree) is a tree in which the value in each node is **greater (less) than or equal** to those in its children (if any)
 - Nodes of a max or min tree may have more than two children (i.e., may not be binary tree)

Max Tree Example



Root has maximum element.

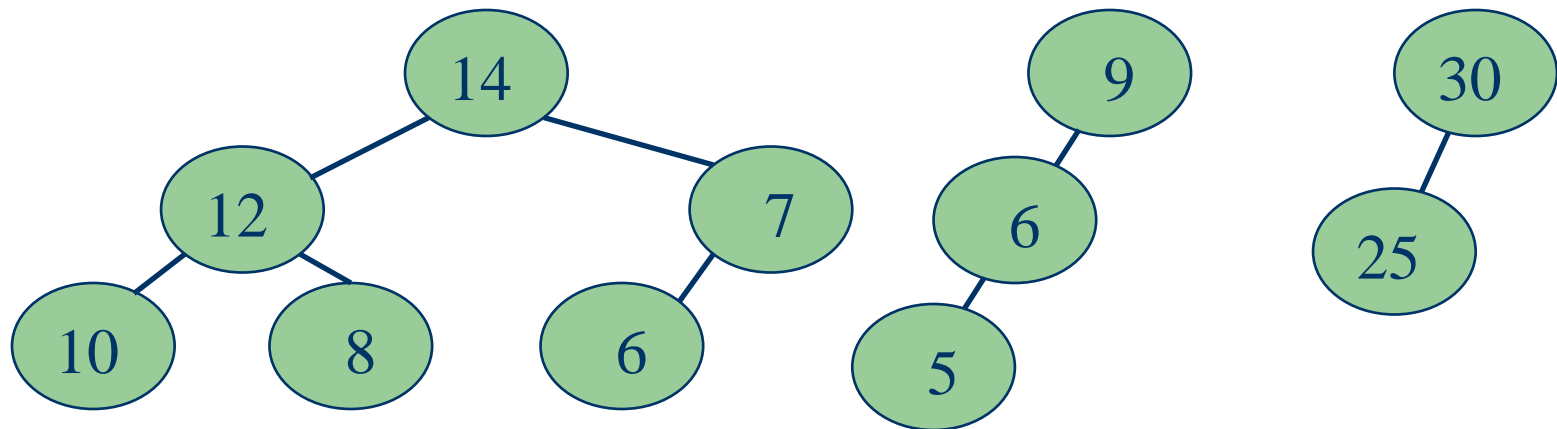
Min Tree Example



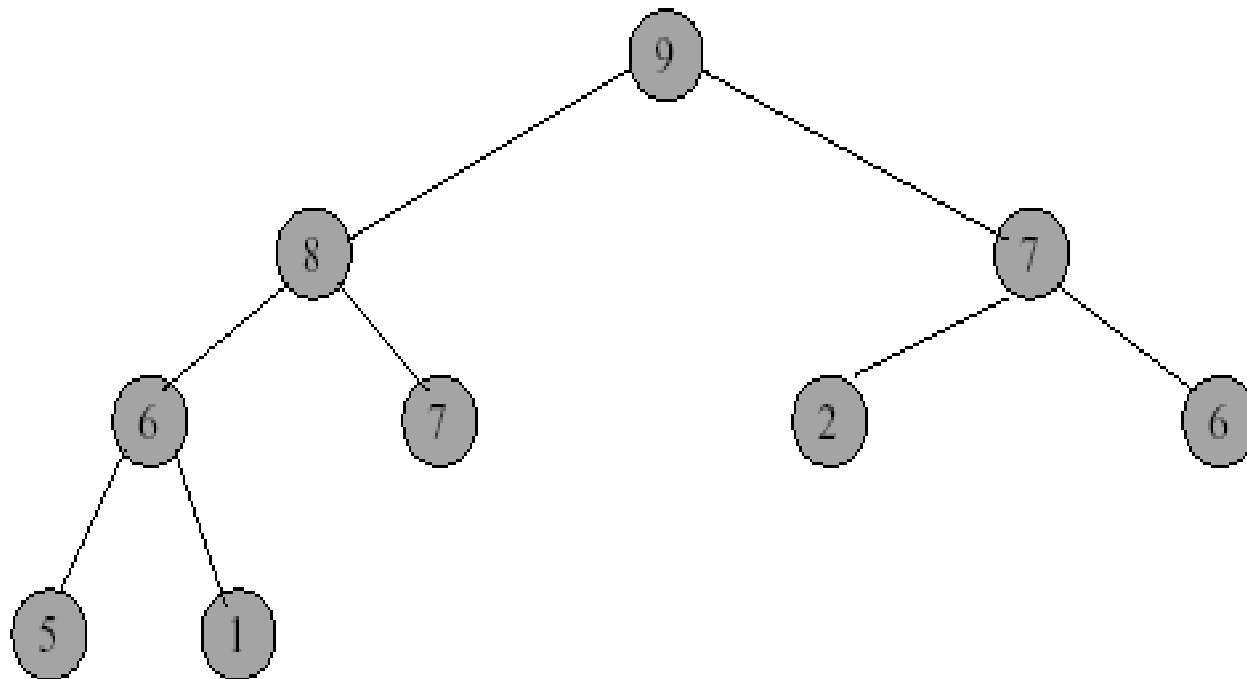
Root has minimum element.

Heaps - Definitions

- A max heap (min heap) is a max (min) tree that is also a complete binary tree
 - Identify are they heap and if yes which type of heap?

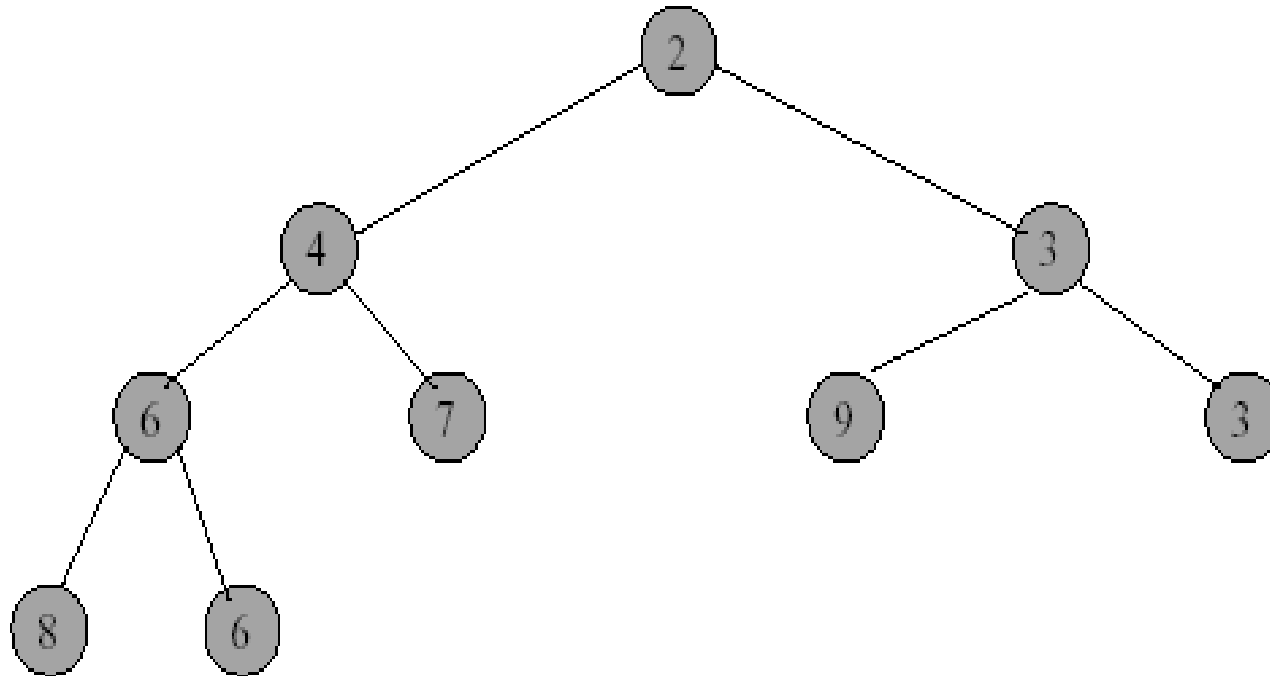


Max Heap with 9 Nodes



Complete binary tree with 9 nodes
that is also a max tree.

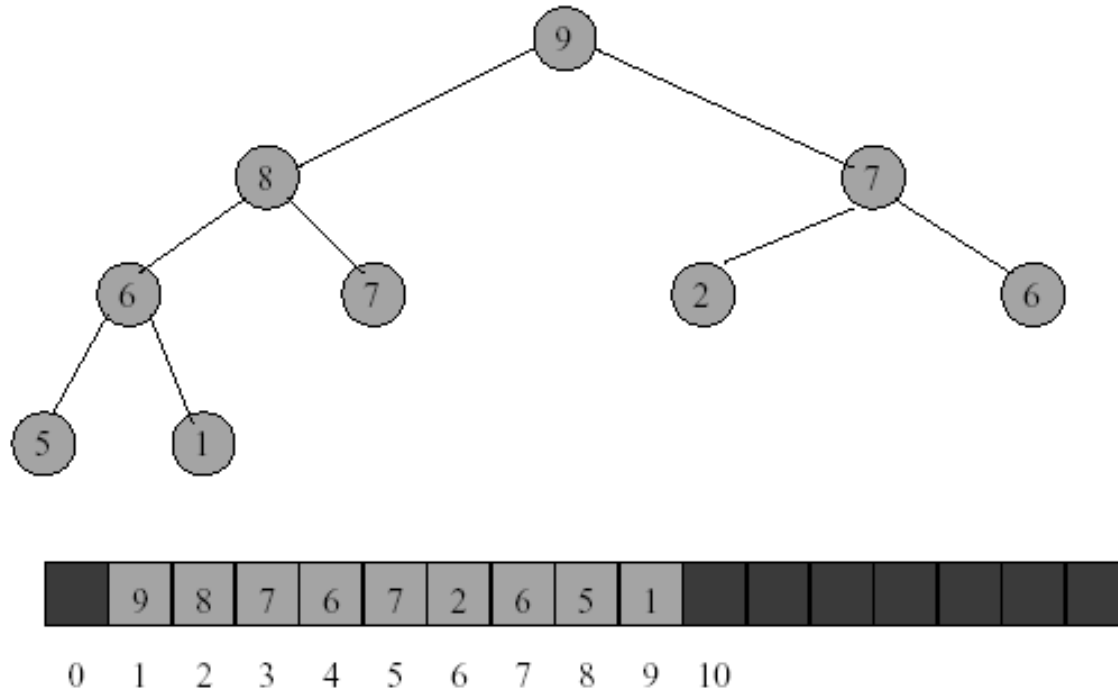
Min Heap with 9 Nodes



Complete binary tree with 9 nodes
that is also a min tree.

Array Representation of Heap

- A heap is efficiently represented as an array.

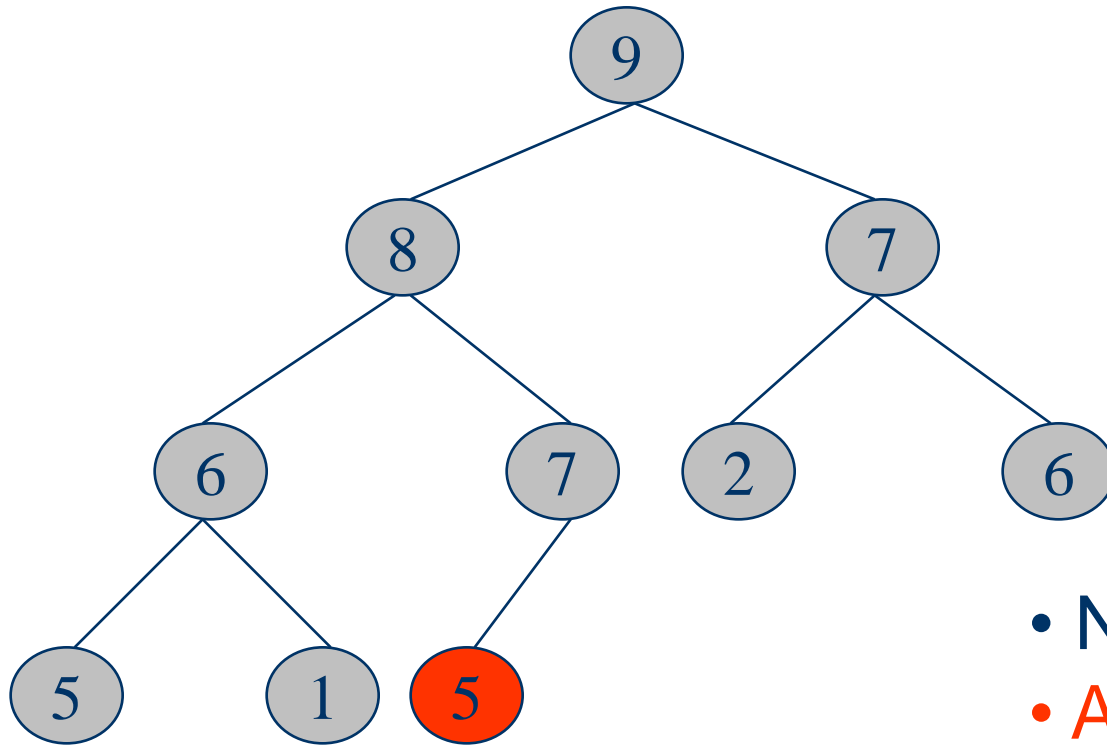


Heap Operations

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

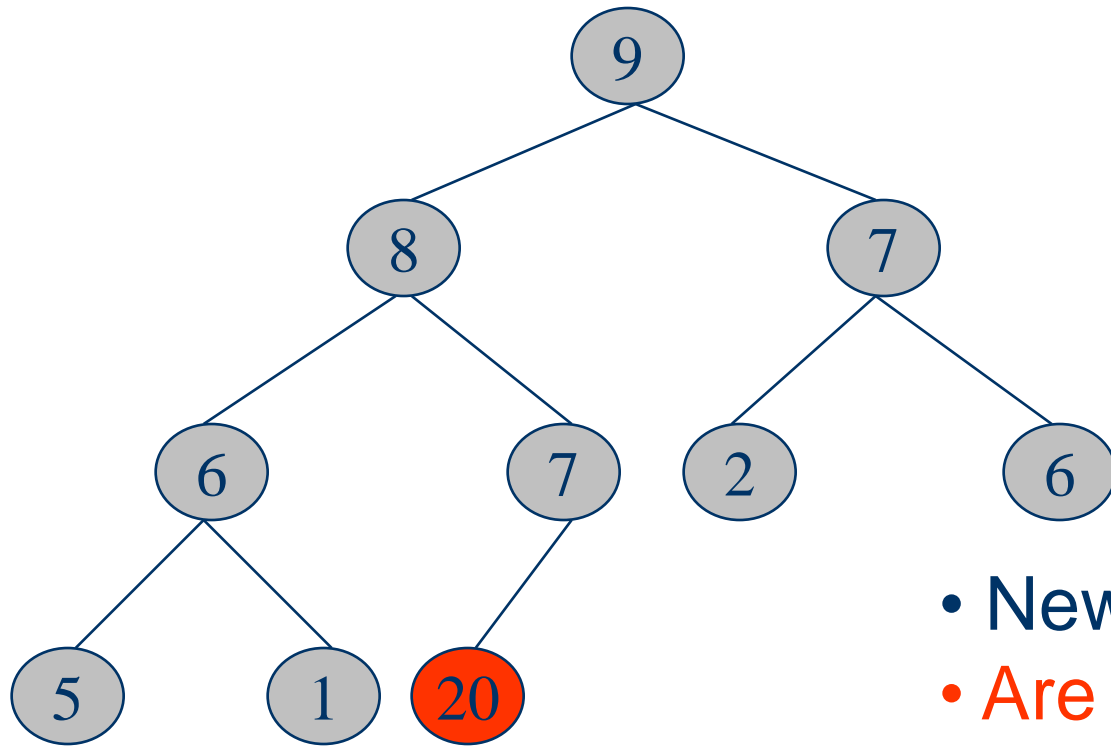
- Insertion $\rightarrow O(\log_2 n)$
- Deletion $\rightarrow O(\log_2 n)$
- Initialization $\rightarrow O(n)$

Insertion into a Max Heap



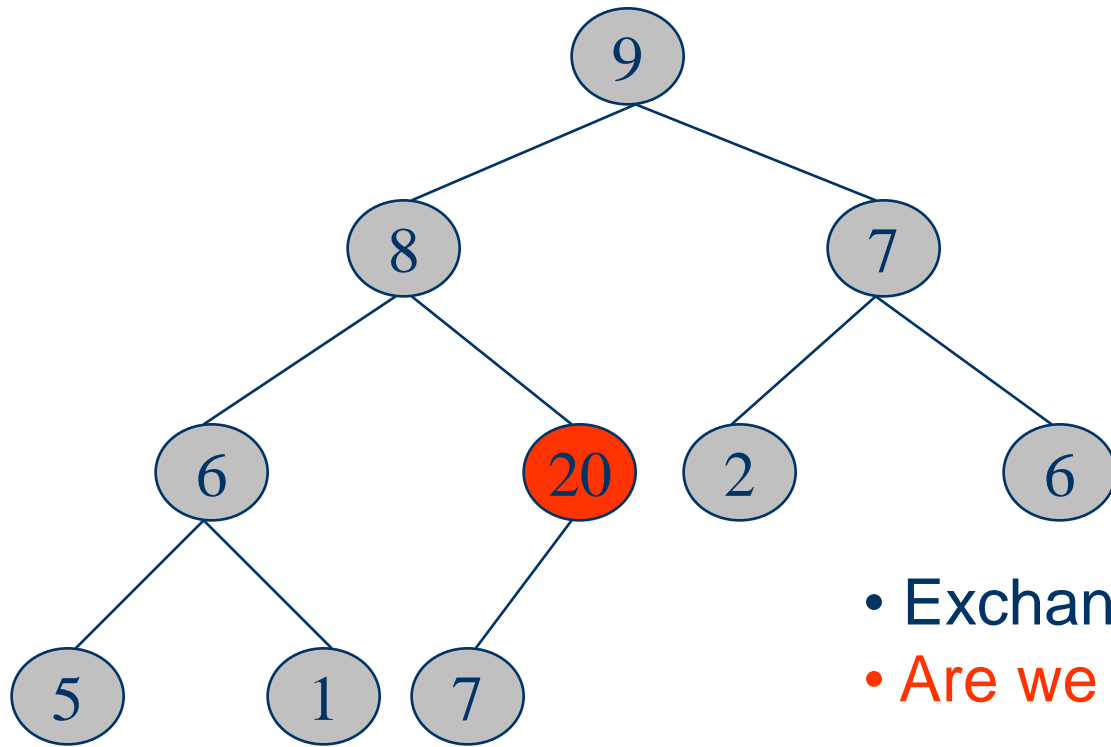
- New element is 5
- Are we finished?

Insertion into a Max Heap



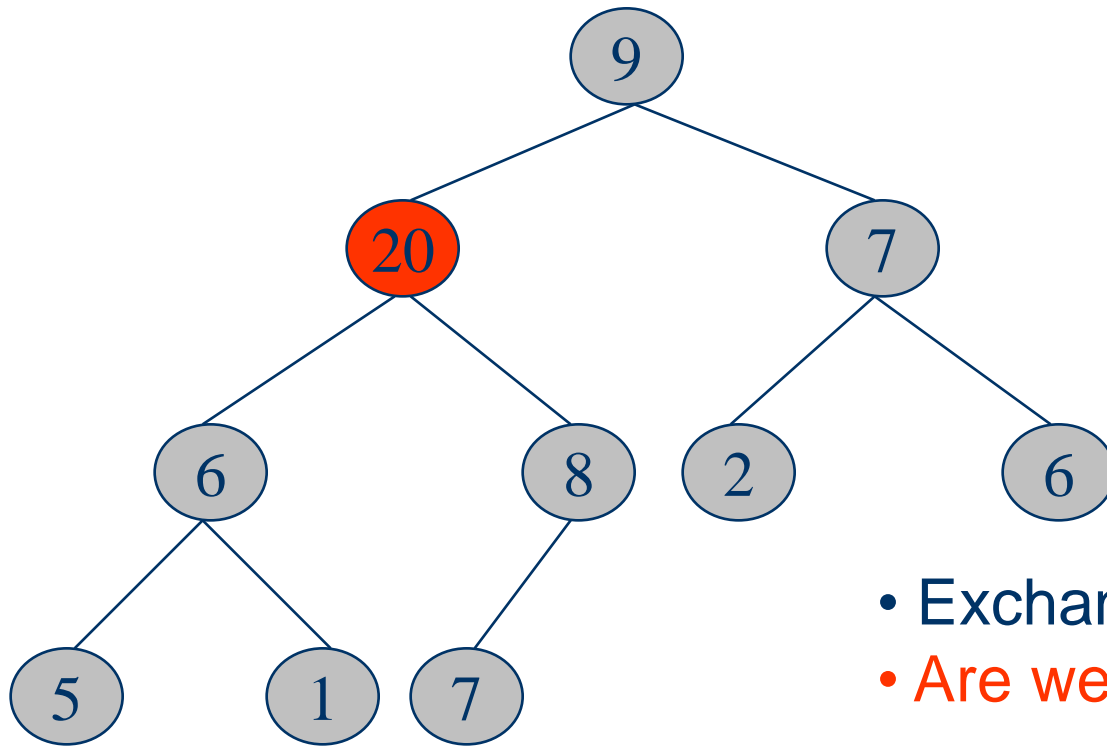
- New element is 20
- Are we finished?

Insertion into a Max Heap



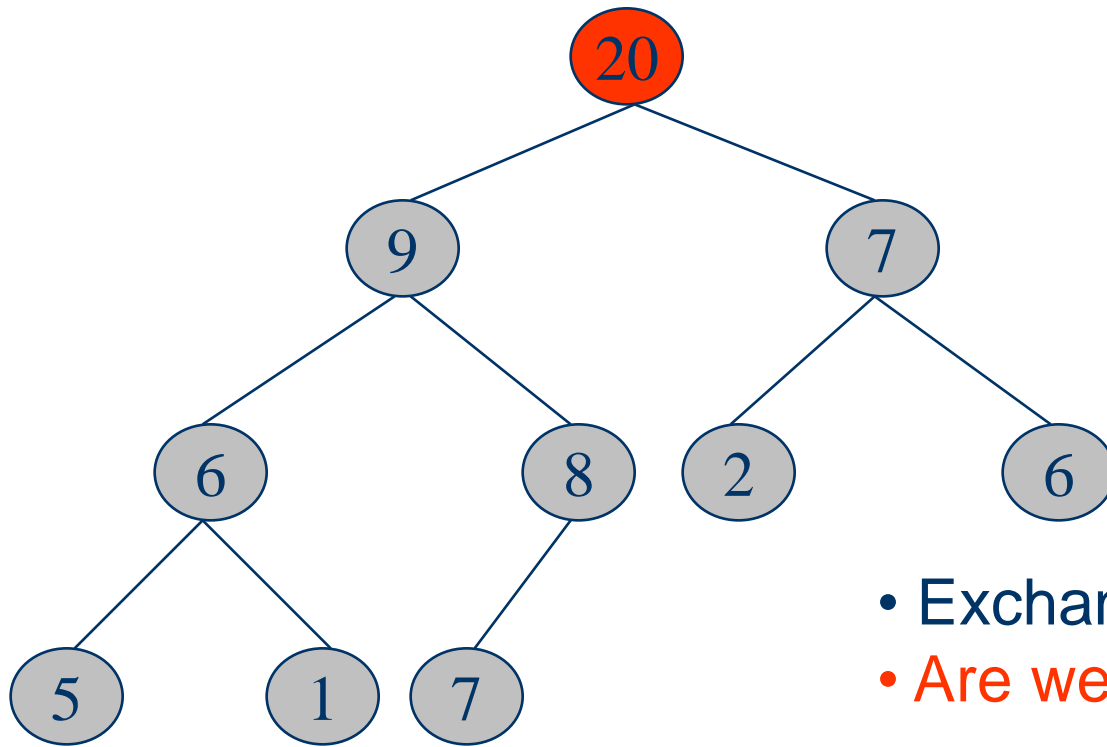
- Exchange the positions with 7
- Are we finished?

Insertion into a Max Heap



- Exchange the positions with 8
- Are we finished?

Insertion into a Max Heap

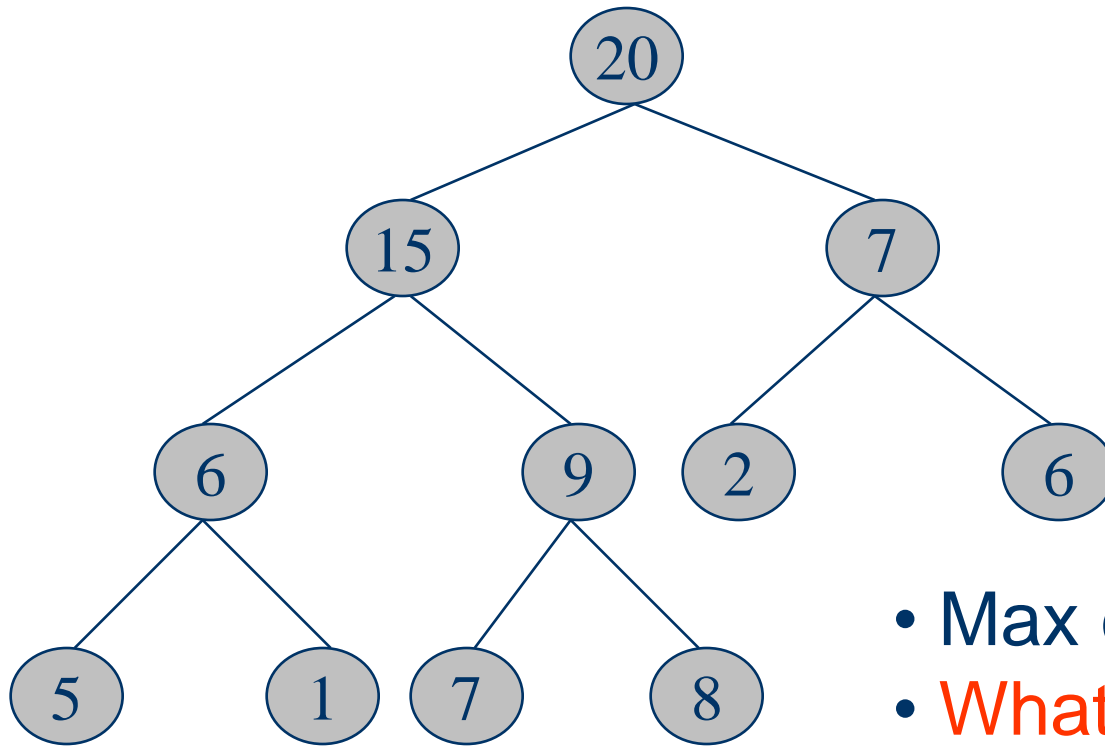


- Exchange the positions with 9
- Are we finished?

Complexity of Insertion

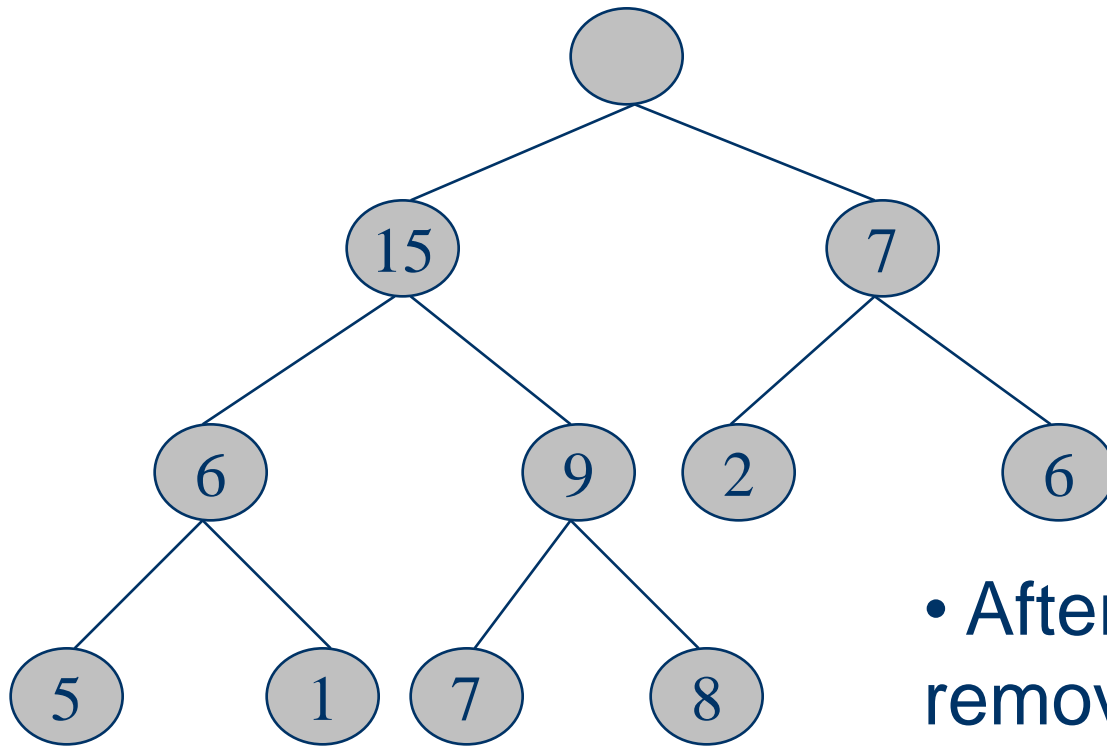
- At each level, we do $\Theta(1)$ work
- Thus the time complexity is **$O(\text{height}) = O(\log_2 n)$** , where n is the heap size

Deletion from a Max Heap



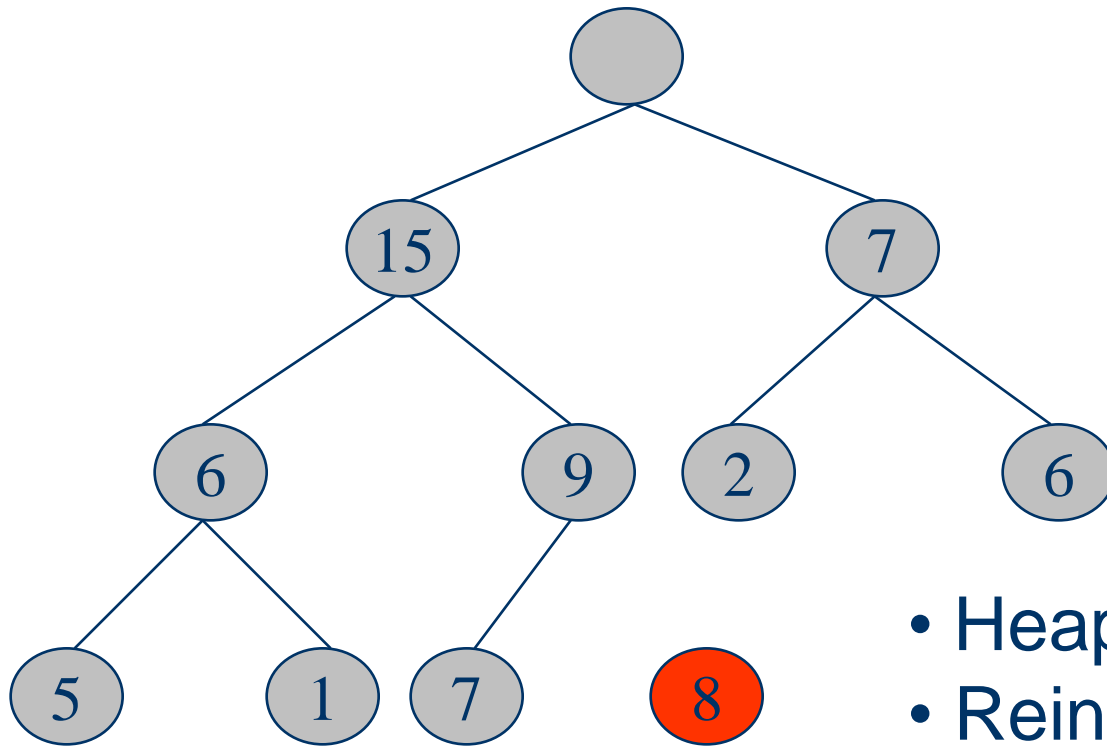
- Max element is in the root
- What happens when we delete an element?

Deletion from a Max Heap



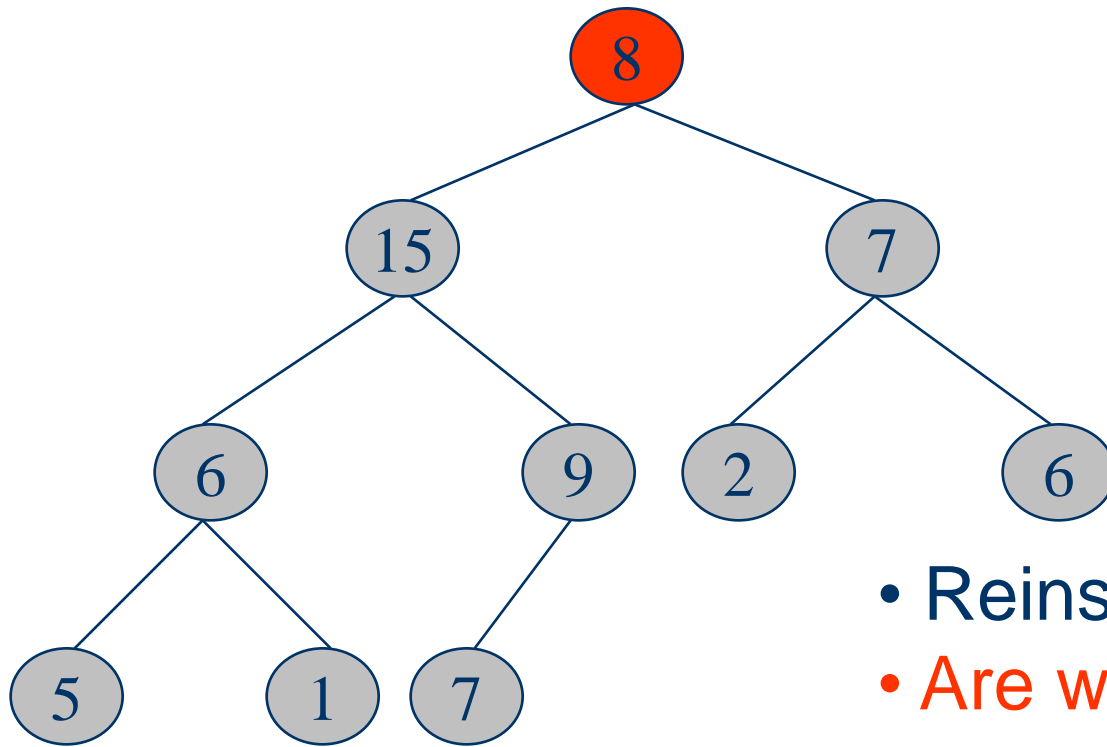
- After the max element is removed.
- Are we finished?

Deletion from a Max Heap



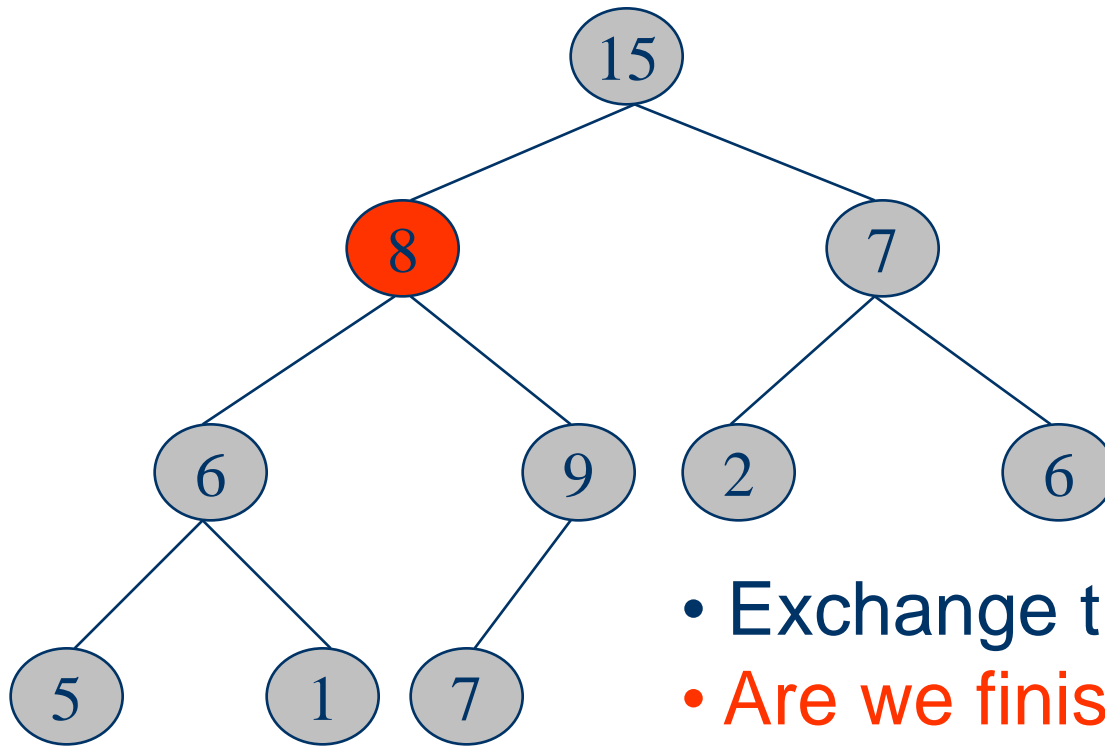
- Heap with 10 nodes.
- Reinsert 8 into the heap.

Deletion from a Max Heap



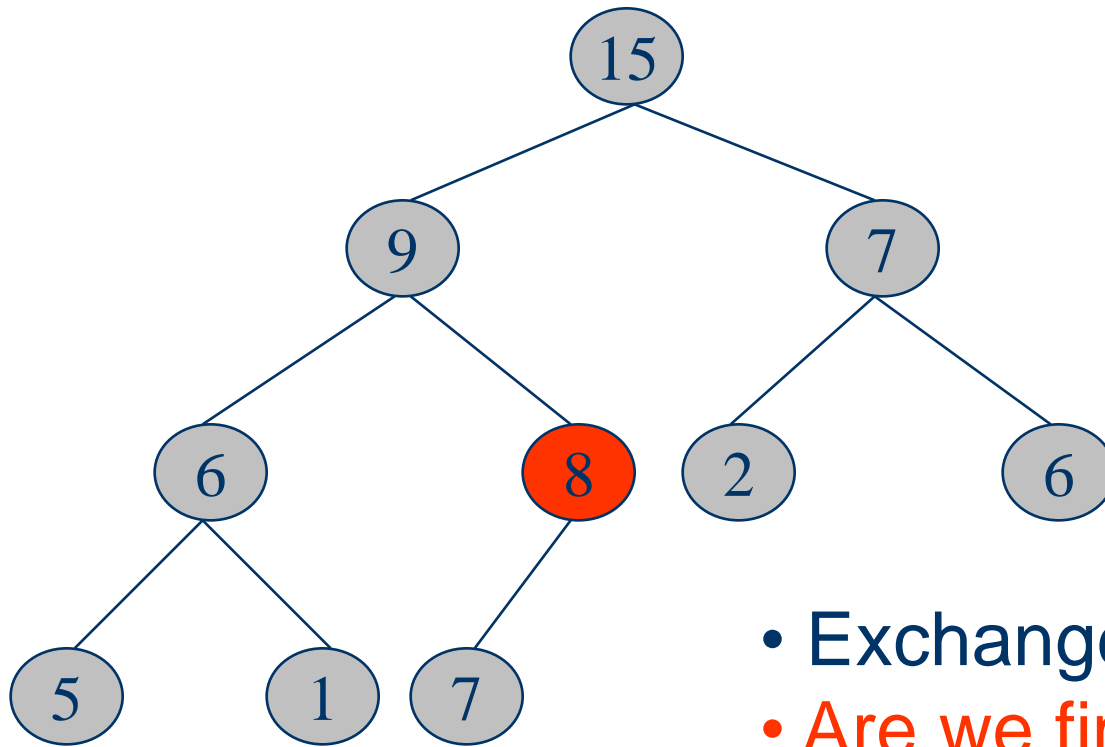
- Reinsert 8 into the heap.
- Are we finished?

Deletion from a Max Heap



- Exchange the position with 15
- Are we finished?

Deletion from a Max Heap



- Exchange the position with 9
- 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 **$O(\text{height}) = O(\log_2 n)$** , where n is the heap size

HEAP_EXTRACT_MAX(A)

1. if $\text{heapsize}_A < 1$
2. then error “heap underflow”
3. $\text{max} \leftarrow A[1]$
4. $A[1] \leftarrow A[\text{heapsize}_A]$
5. $\text{heapsize}_A \leftarrow \text{heapsize}_A - 1$
6. $\text{MAX_HEAPIFY}(A, 1)$
7. return max

HEAP_INSERT(A,key)

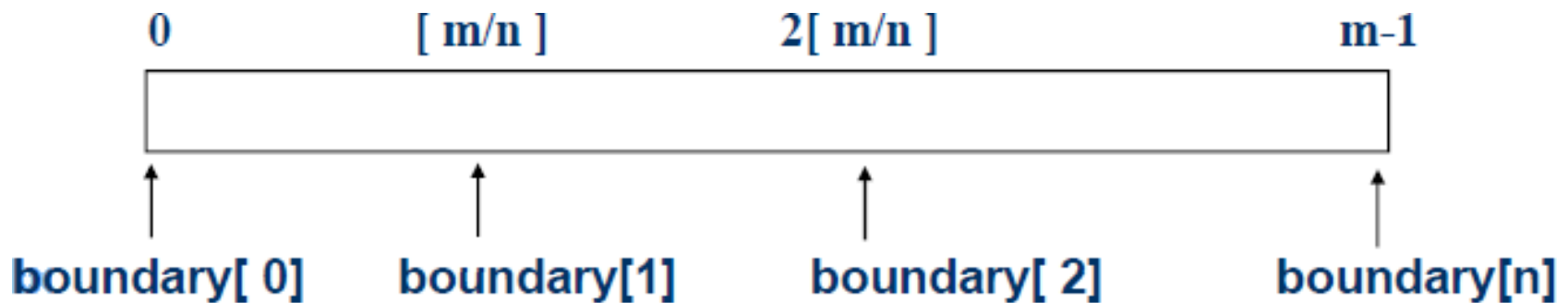
1. $\text{heapsize}_A \leftarrow \text{heapsize}_A + 1$
2. $i \leftarrow \text{heapsize}_A$
3. while $i > 1$ and $A[\text{Parent}(i)] < \text{key}$
4. do $A[i] \leftarrow A[\text{Parent}(i)]$
5. $i \leftarrow \text{Parent}(i)$
6. $A[i] \leftarrow \text{key}$

Multiqueues

- Here only one single one-dimensional array used to store multiple queues.
- m denotes the maximum size of the array being used.
- n denotes the number of queues.
- We also assume that equal segments of array will be used for each queue.
- Again we can have ***enqueue*** or ***dequeue*** operations, which can be performed on each of these queues .

Multiqueues

- More than two queues (n)
- Memory is divided into n segments
 - The initial division of these segments may be done in proportion to expected sizes of these queues if these are known
 - All queues are empty and divided into roughly equal segments



Queues vs. Stacks

- Stacks are a LIFO container
 - Store data in the reverse of order received
- Queues are a FIFO container
 - Store data in the order received
- Stacks then suggest applications where some sort of reversal or unwinding is desired.
- Queues suggest applications where service is to be rendered relative to order received.
- Stacks and Queues can be used in conjunction to compare different orderings of the same data set.
- From an ordering perspective, then, Queues are the “opposite” of stacks
 - Easy solution to the palindrome problem