

---

**15.082 and 6.855J**  
**February 6, 2003**

**Data Structures**

# Overview of this Lecture

---

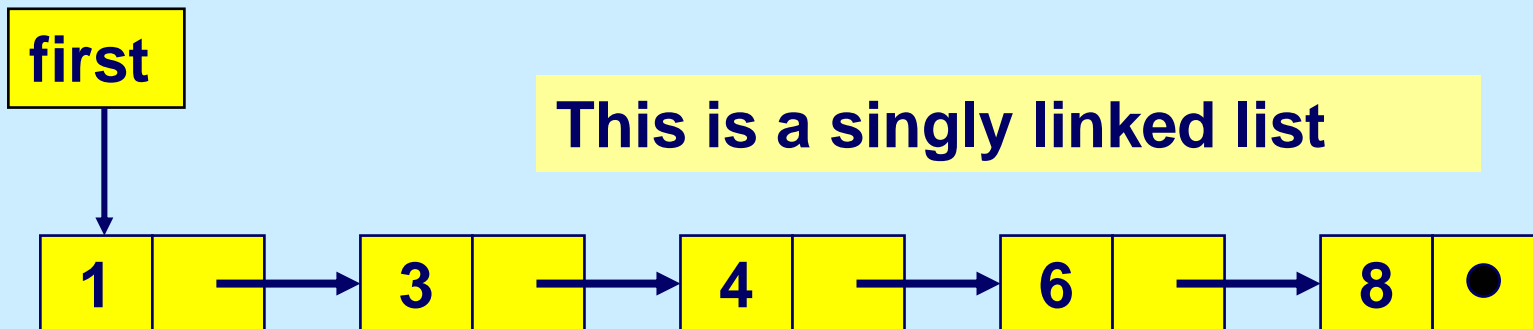
- ◆ **A very fast overview of some data structures that we will be using this semester**
  - **lists, sets, stacks, queues, networks, trees**
  - **a variation on the well known heap data structure**
  - **binary search**
- ◆ **Illustrated using animation**
- ◆ **We are concerned with  $O(\ )$  computation counts, and so do not need to get down to C- level (or Java level).**

# Two standard data structures

---

1	2	3	4	5	6	7	8	9	10
1	0	1	1	0	1	0	1	0	0

**Array:** a vector: stored consecutively in memory, and typically allocated in advance



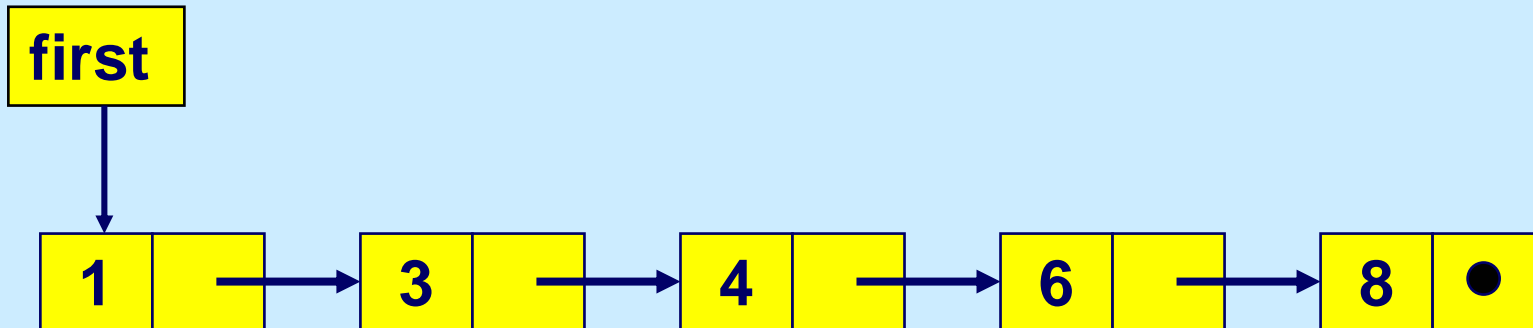
**cells:** hold fields of numbers and pointers to implement lists.

# Two standard data structures

---

1	2	3	4	5	6	7	8	9	10
1	0	1	1	0	1	0	1	0	0

Each has its advantages. Each can be more efficient than the other. It depends on what operations you want to perform.



*Let's consider which data structure to use for working with node subsets, or arc subsets.*

# Two key concepts

---

**Abstract data types**: a descriptor of the operations that are permitted, e.g.,

**abstract data type: set  $S$**

- ***initialize( $S$ )***: creates an empty set  $S$
- ***add( $S, s$ )***: replaces  $S$  by  $S \cup \{s\}$ .
- ***delete( $S, s$ )***: replaces  $S$  by  $S \setminus \{s\}$
- ***FindElement( $S, s$ )***:  $s :=$  some element of  $S$
- ***IsElement( $S, s$ )***: returns true if  $s \in S$

**Data structure**: usually describes the high level implementation of the abstract data types, and can be analyzed for running time.

- doubly linked list, etc

# A note on data structures

---

- ◆ **Preferences**
  - **Simplicity**
  - **Efficiency**
  - **In case there are multiple good representations, we will choose one**

# Storing a subset S of Nodes

---

## Operations

create an empty set S

$S = \emptyset$

add an element to S

$S = \{2, 4, 7\}$

delete an element from S

$S = \{2, 4\}$

determine if a node is in S

Is  $7 \in S$ ?

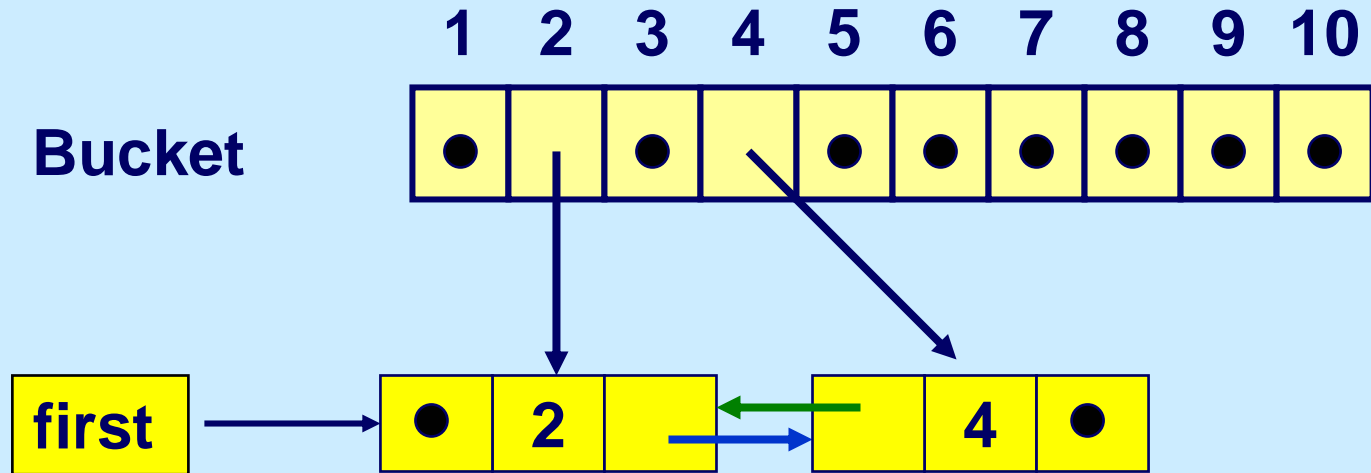
Array *Bucket*

Bucket(i) points to i if  $i \in S$

use a doubly linked list to store the set S

# Animating the List of Nodes

---

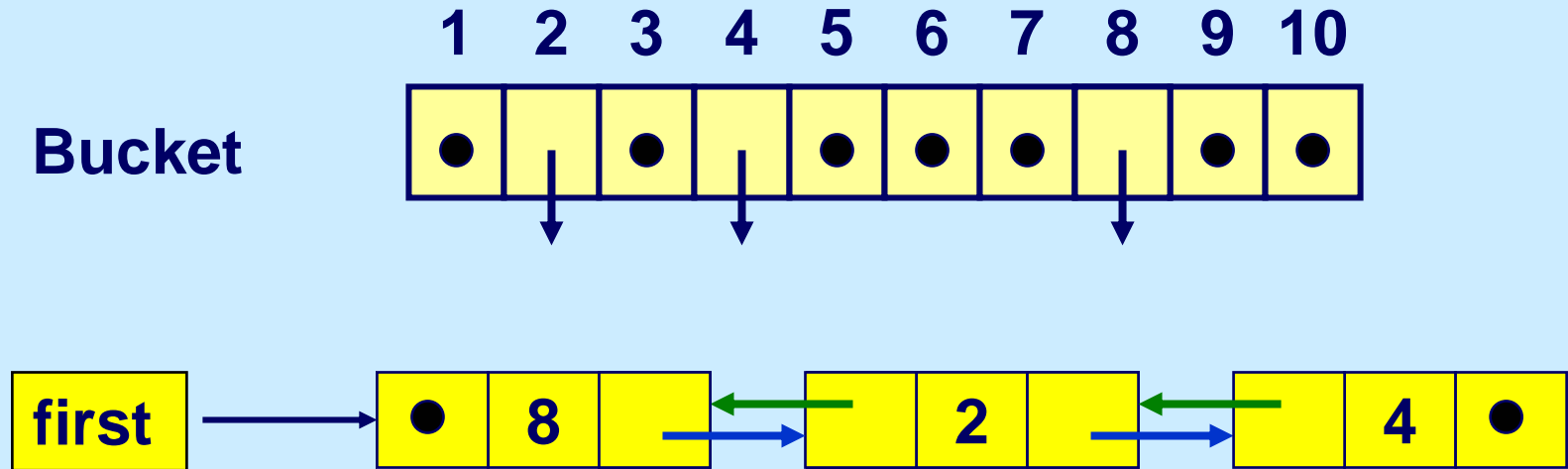


This is a doubly linked list

Operation: insert node 8 into the list

# Operation: Insert node 8 into list

---



Insert node 8 as the first node on the list.

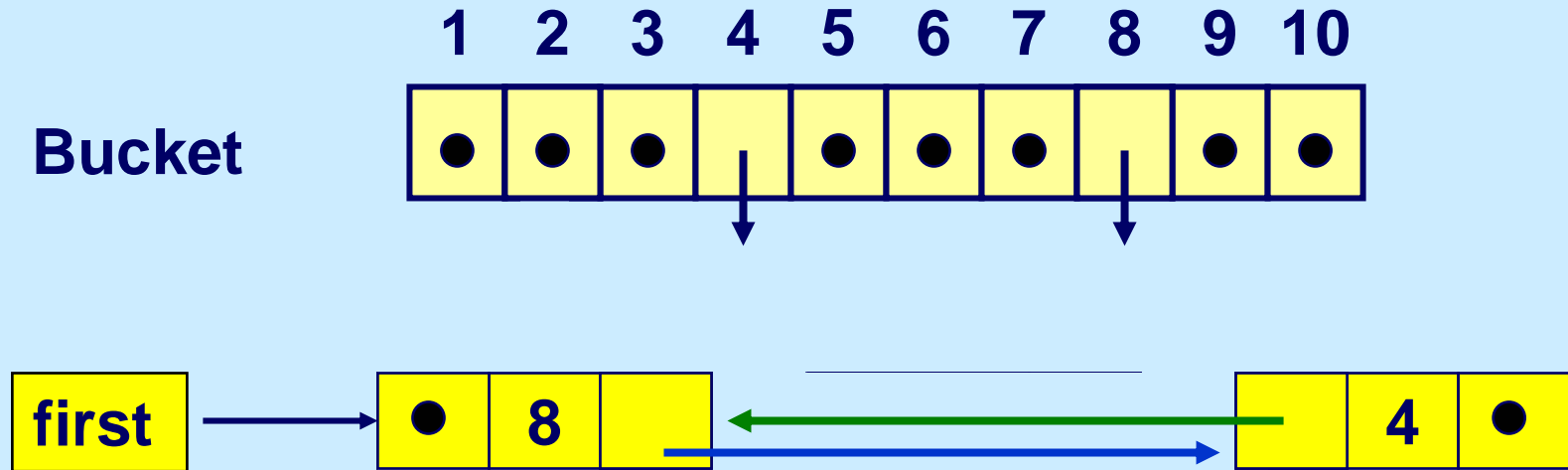
Update pointers on the doubly linked list.

Update the pointer in the array.

The entire insertion takes  $O(1)$  steps.

# Operation: Delete node 2 from the list

---



Delete the cells containing 2 plus the pointers

Update the pointers on the doubly linked list.

Update the pointer on the array

The entire deletion takes  $O(1)$  steps.

# Summary on maintaining a subset

---

## Operations

- Determine if  $j \in S$
- Add  $j$  to  $S$
- Delete  $j$  from  $S$

Each operation takes  $O(1)$  steps using a doubly linked list and an array.

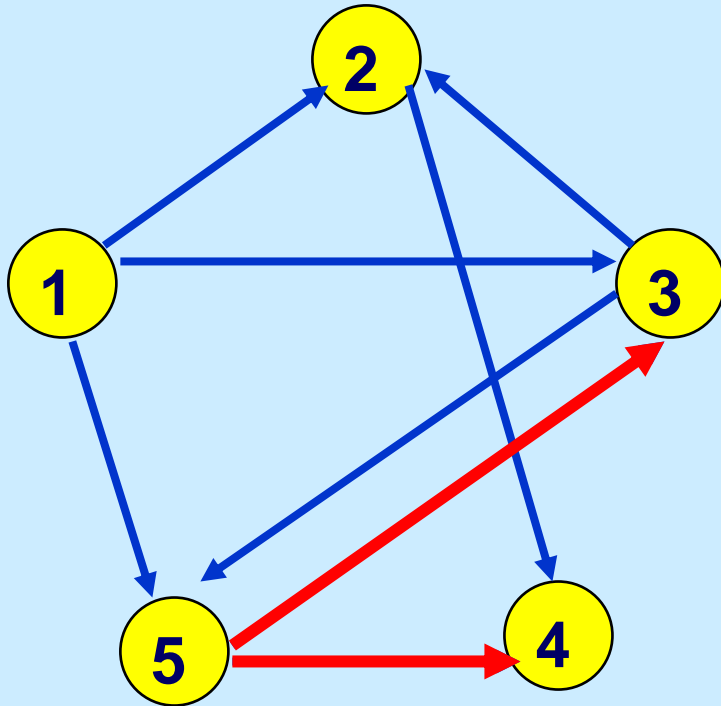
Can be used to maintain a subset of arcs.

Can be used to maintain multiple subsets of nodes

- Set1, Set2, Set3, ..., Setk
- Use an index to keep track of the set for each node

# A network

---



We can view the arcs of a network as a collection of sets.

Let  $A(i)$  be the arcs emanating from node  $i$ .

e.g.,  $A(5) = \{ (5,3), (5,4) \}$

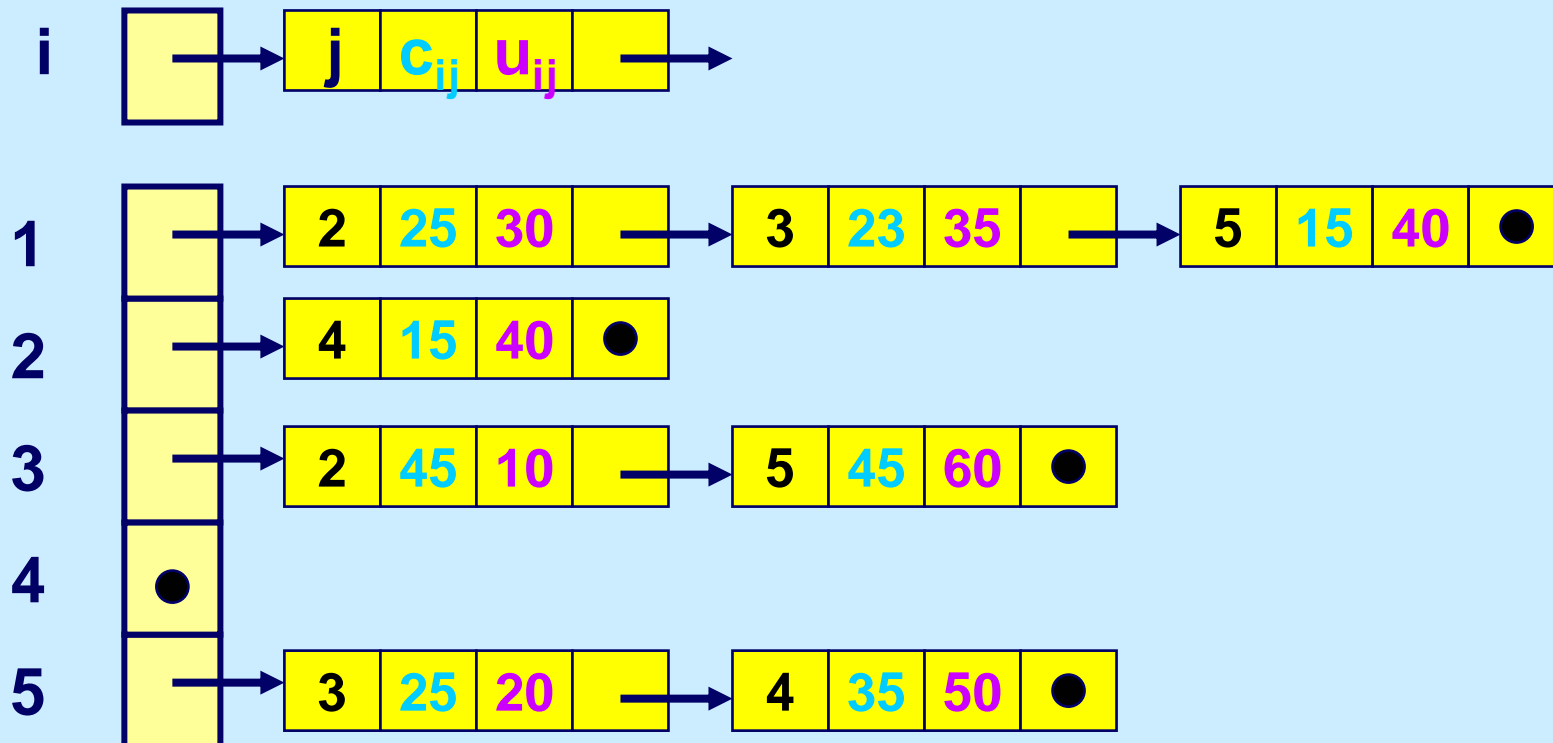
**Note:** these sets are static. They stay the same.

**Common operations:** find the first arc in  $A(i)$   
find the arc  $\text{CurrentArc}(i)$  on  $A(i)$

# Storing Arc Lists: A(i)

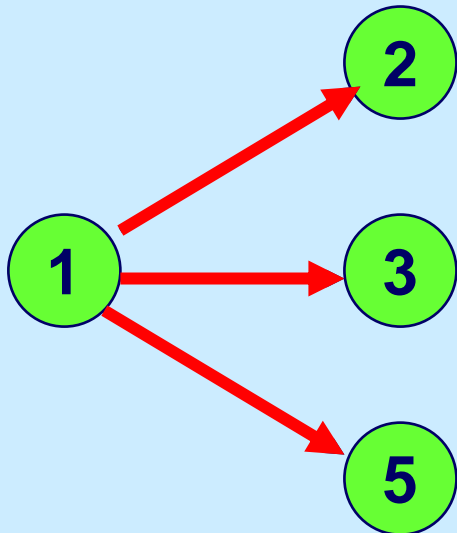
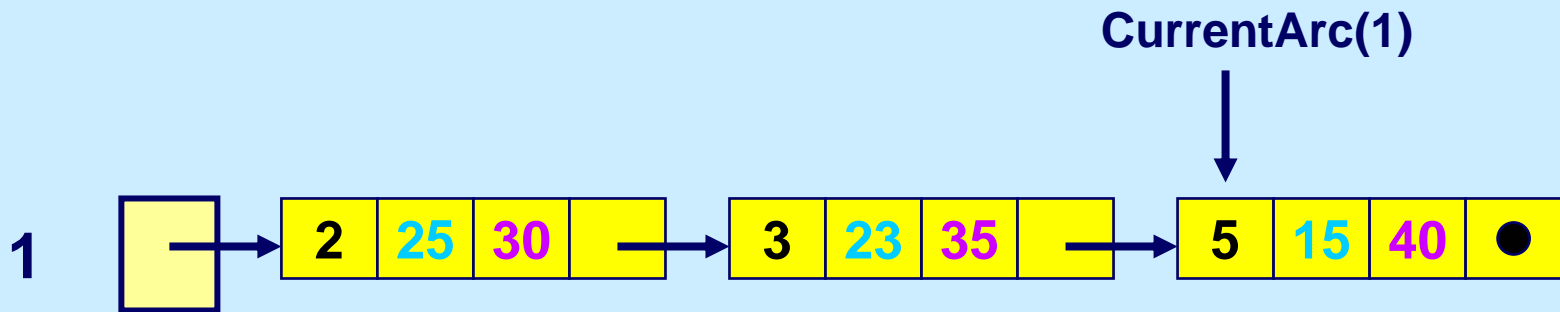
## Operations permitted

- Find first arc in A(i)
- Store a pointer to the current arc in A(i)
- Find the arc after CurrentArc(i)



# Current Arc

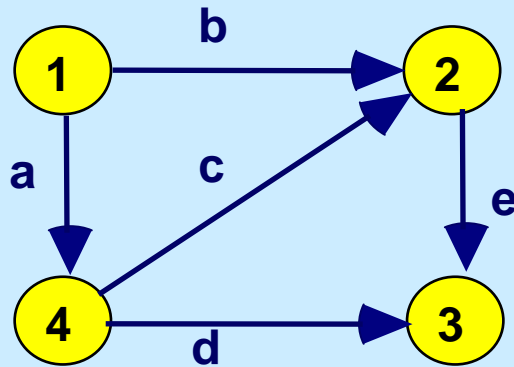
**CurrentArc(i)** is a placeholder for when one scans through the arc list A(i)



Finding **CurrentArc** and the arc after **CurrentArc** takes  $O(1)$  steps.

These are also implemented often using arrays called ***forward star representations***.

# The Adjacency Matrix (for directed graphs)



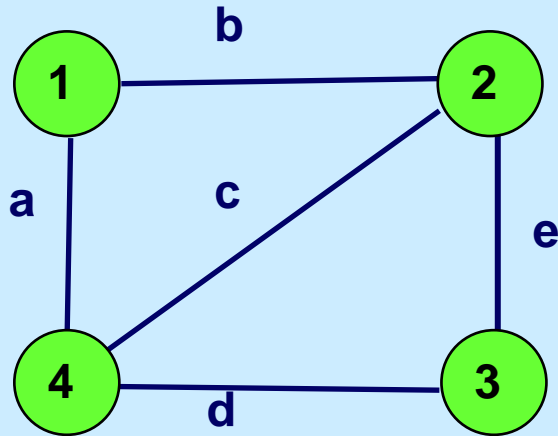
A Directed Graph

	1	2	3	4
1	0	1	0	1
2	0	0	1	0
3	0	0	0	0
4	0	1	1	0

- Have a row for each node
- Have a column for each node
- Put a 1 in row  $i$ - column  $j$  if  $(i,j)$  is an arc

What would happen if  $(4,2)$  became  $(2,4)$ ?

# The Adjacency Matrix (for undirected graphs)



	1	2	3	4	degree
1	0	1	0	1	2
2	1	0	1	1	3
3	0	1	0	1	2
4	1	1	1	0	3

An Undirected Graph

- Have a row for each node
- Have a column for each node
- Put a 1 in row  $i$ - column  $j$  if  $(i,j)$  is an arc

The degree of a node is the number of incident arcs

# Adjacency Matrix vs Arc Lists

---

## Adjacency Matrix?

Efficient storage if matrix is very “*dense*.”

Can determine if  $(i,j) \in A(i)$  in  $O(1)$  steps.

Scans arcs in  $A(i)$  in  $O(n)$  steps.

## Adjacency lists?

Efficient storage if matrix is “*sparse*.”

Determining if  $(i,j) \in A(i)$  can take  $|A(i)|$  steps

Can scan all arcs in  $A(i)$  in  $|A(i)|$  steps

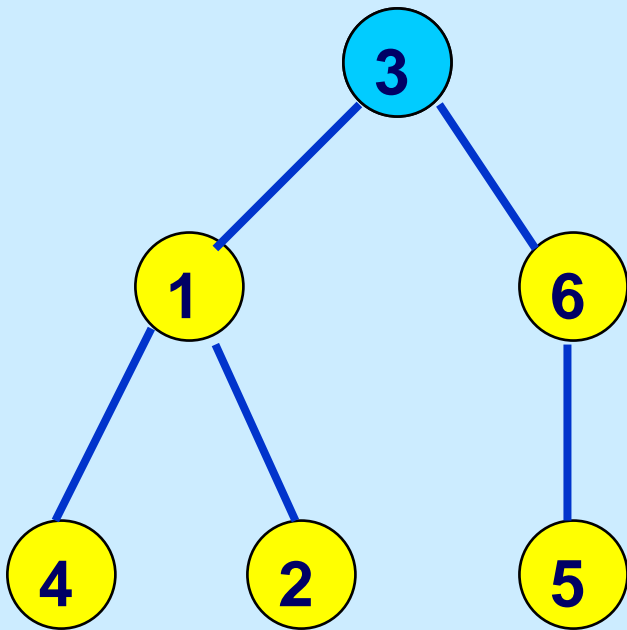
# Trees

---

A **tree** is a connected acyclic graph.

(Acyclic here, means it has no undirected cycles.)

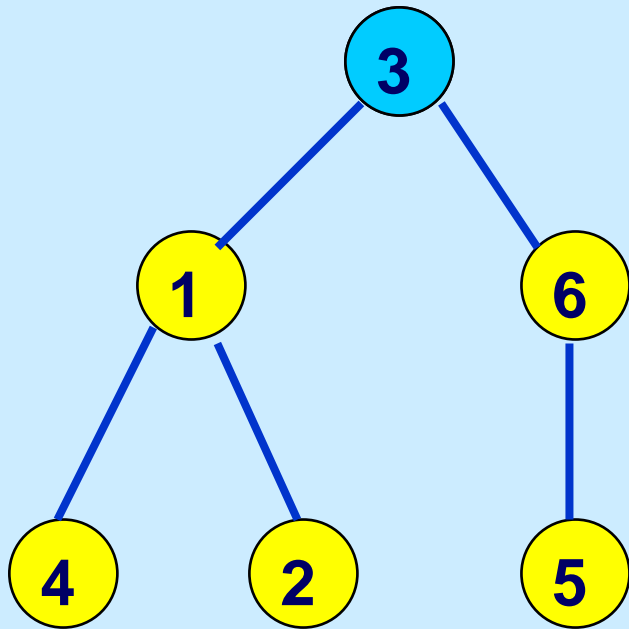
If a tree has  $n$  nodes, it has  $n-1$  arcs.



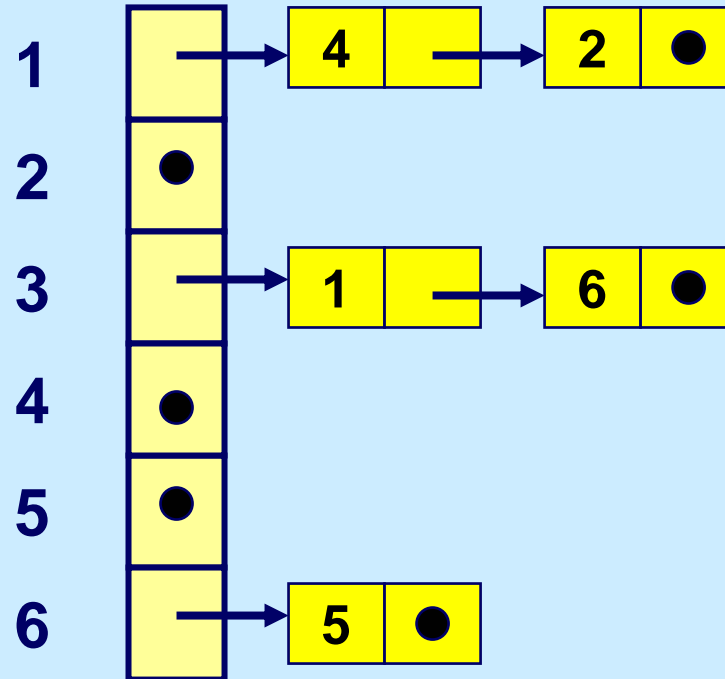
This is an undirected tree.

To store trees efficiently, we hang the tree from a **root node**.

# Trees



## Lists of children



node

1 2 3 4 5 6

pred

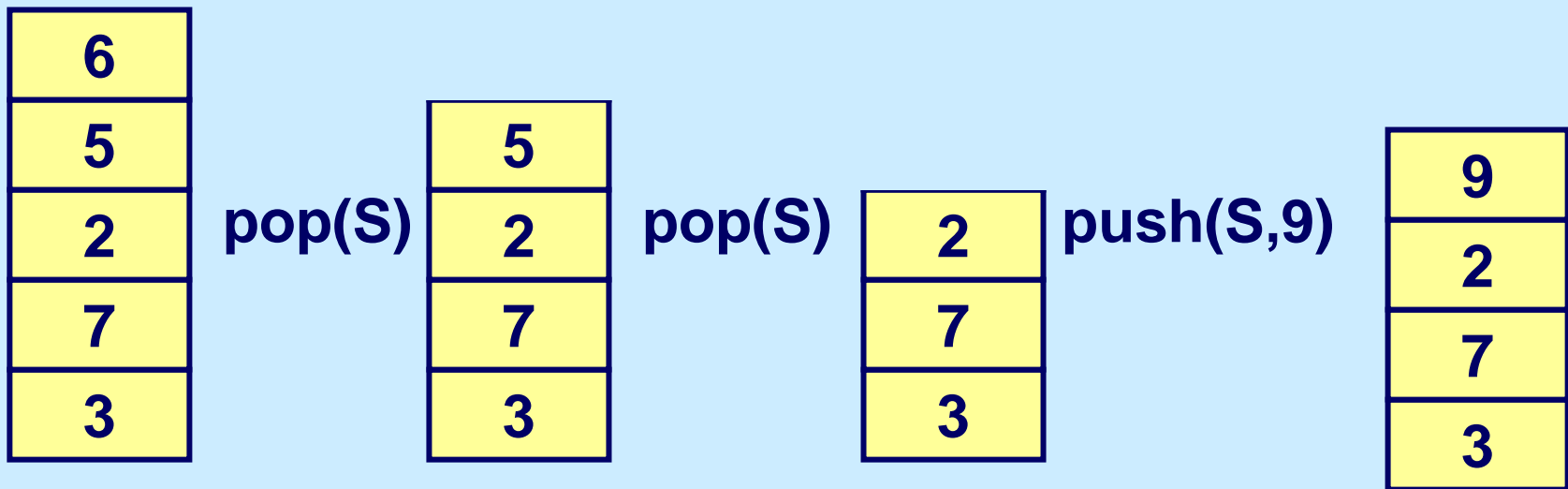
3	1	0	1	6	3
---	---	---	---	---	---

predecessor (parent) array

# Stacks -- Last In, First Out (LIFO)

## Operations:

- ◆ *create(S)* creates an empty stack S
- ◆ *push(S, j)* adds j to the top of the stack
- ◆ *pop(S)* deletes the top element in S
- ◆ *top(S)* returns the top element in S



# Queues – First in, First Out (FIFO)

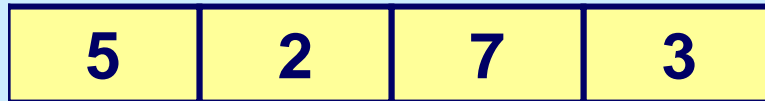
---

## Operations:

- ◆ *create(Q)* creates an empty queue Q
- ◆ *Insert(Q, j)* adds j to the end of the queue
- ◆ *Delete(Q)* deletes the first element in Q
- ◆ *first(Q)* returns the top element in S



Delete(Q)



Delete(Q)



Insert(Q,9)



# Priority Queues

---

Each element  $j$  has a value  $key(j)$

- ◆ ***create(P, S)*** initializes P with elements of S
- ◆ ***Insert(P, j)*** adds j to P
- ◆ ***Delete(P, j)*** deletes j from P
- ◆ ***FindMin(P, k)*** k: index with minimum key value
- ◆ ***ChangeKey(j, Δ)***  $key(j) := key(j) + \Delta$ .

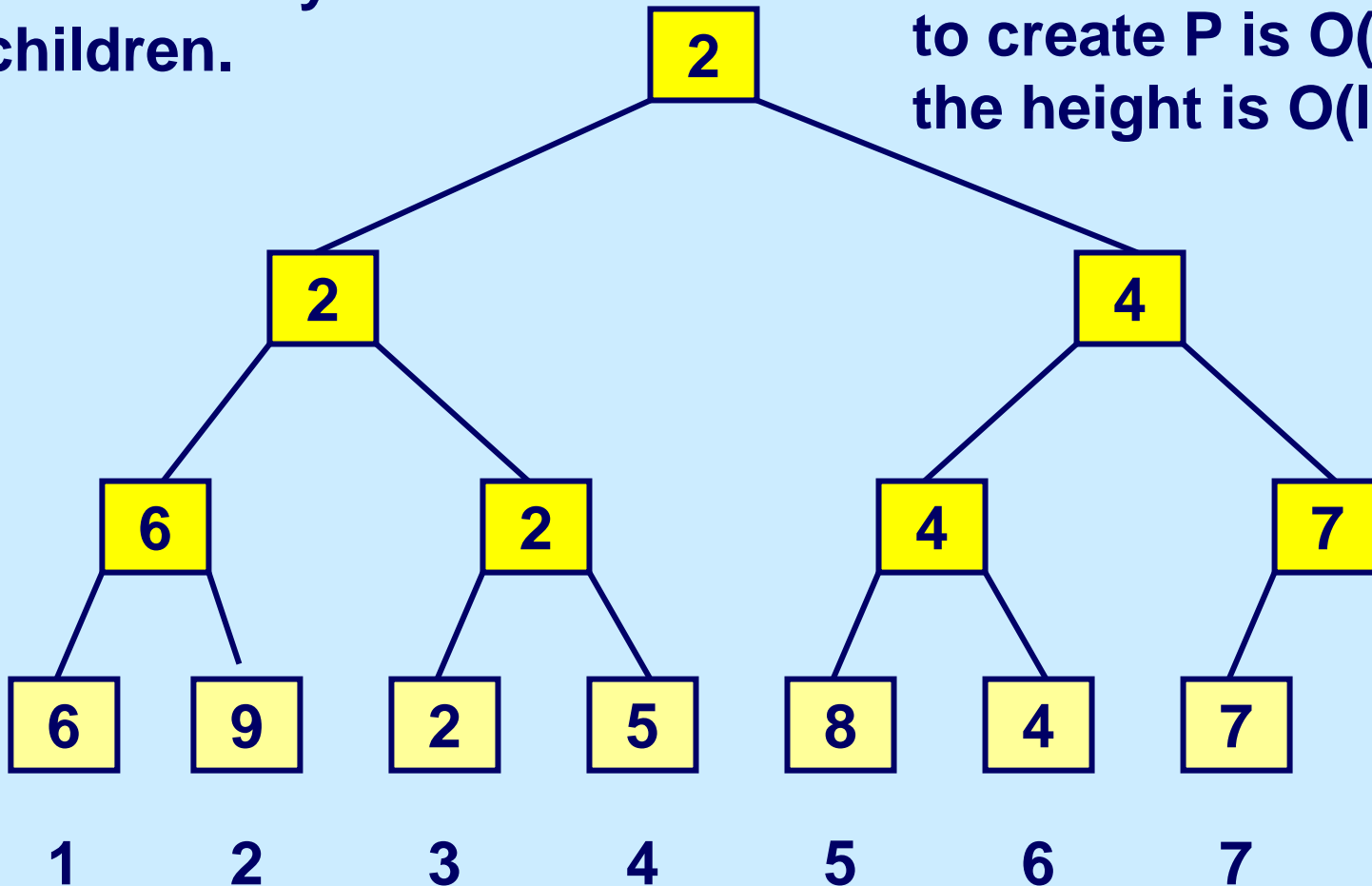
A common data structure, the first one that cannot be implemented as a doubly linked list.

Goal: always locate the minimum key, the index with the highest priority. (Think deli counter.)

# Implementing Priority Queues using Leaf-Heaps

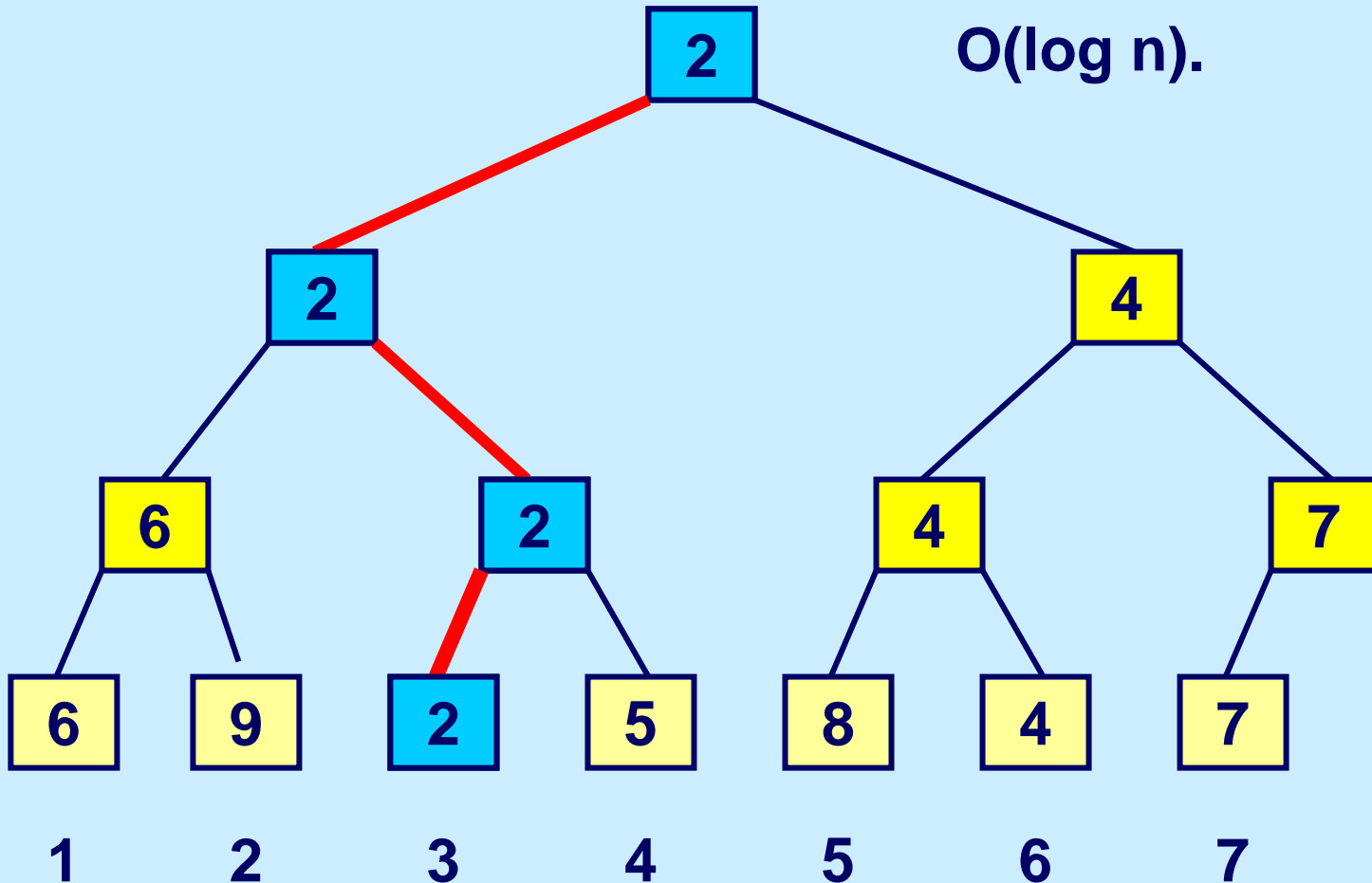
Parent node stores the smaller key of its children.

Note: if there are  $n$  leaves, the running time to create  $P$  is  $O(n)$  and the height is  $O(\log n)$ .



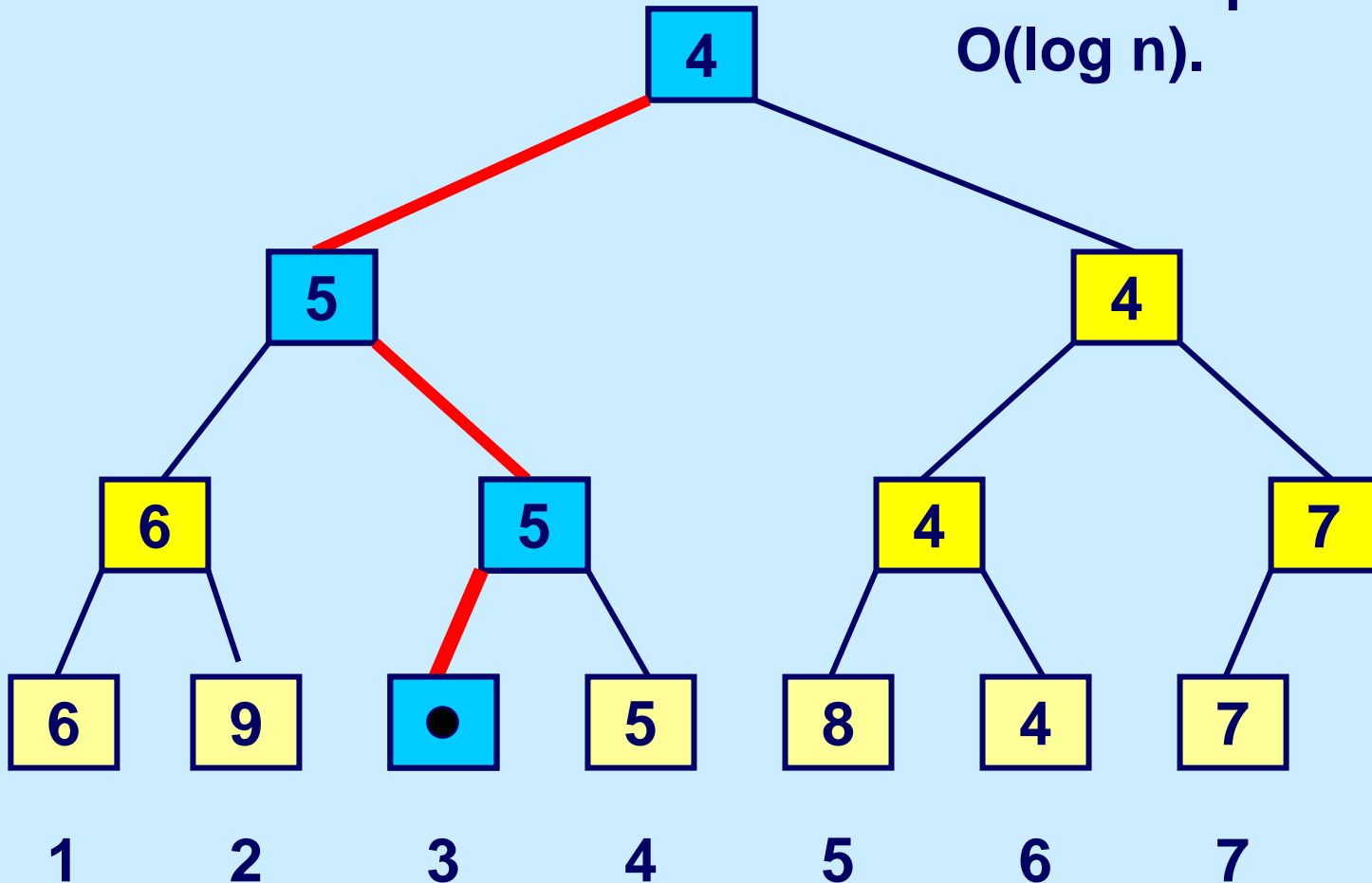
# Find Min using Leaf-Heaps

Start at the top and sift down. Time =  $O(\log n)$ .



# Deleting node 3 using Leaf-Heaps

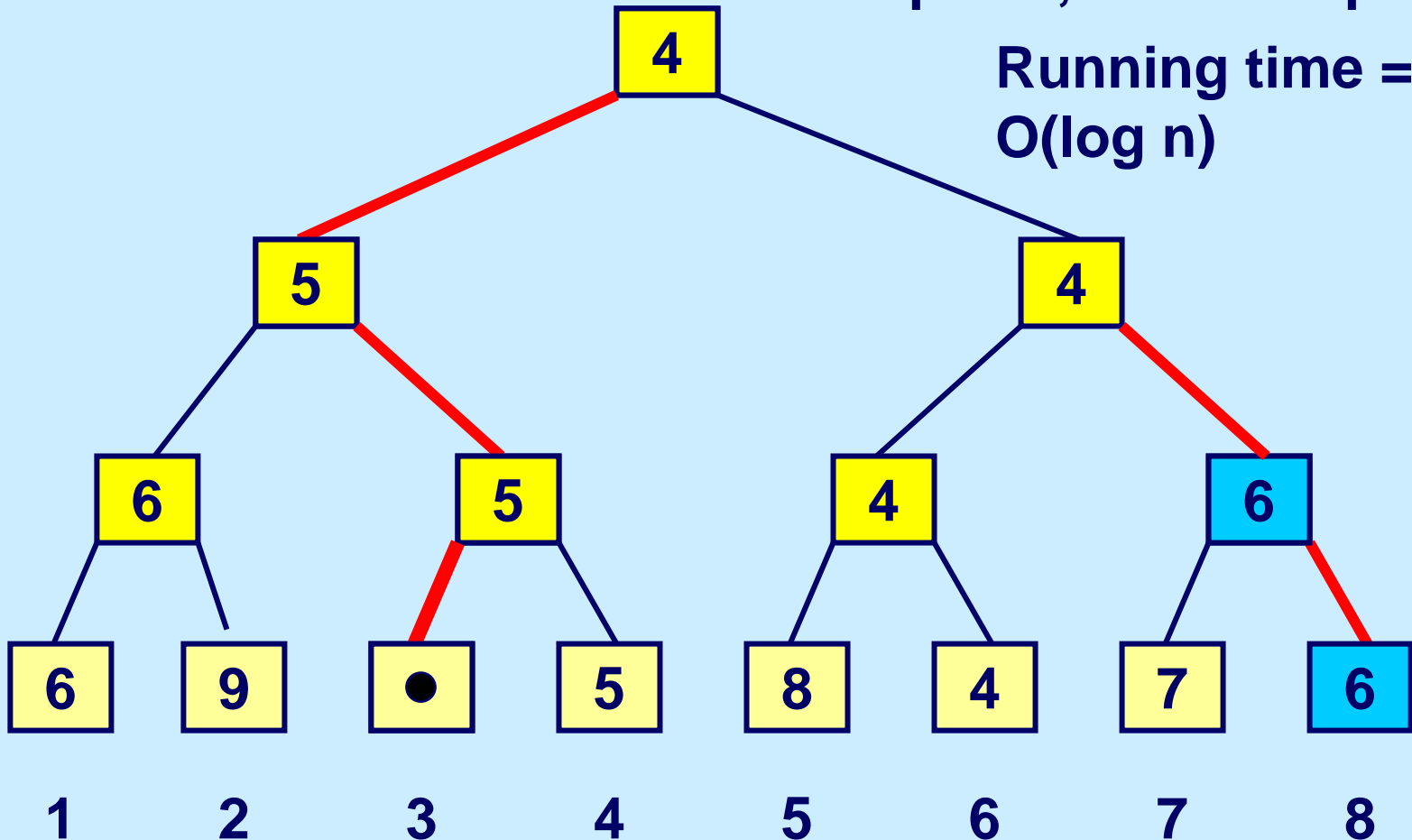
Start at the bottom and sift up. Time is  $O(\log n)$ .



# Insert node 8 using Leaf-Heaps

Insert in the last place, and sift up.

Running time =  $O(\log n)$



# Summary of Heaps

---

- ◆ *create(P, S)*             $O(n)$  time;     $|S| = O(n)$
  - ◆ *Insert(P, j)*             $O(\log n)$  time
  - ◆ *Delete(P, j)*             $O(\log n)$  time
  - ◆ *FindMin(P, k)*           $O(\log n)$  time
  - ◆ *ChangeKey(j, Δ)*       $O(\log n)$  time
- 
- ◆ Can be used to sort in  $O(\log n)$  by repeatedly finding and deleting the minimum.
  - ◆ The standard data structure to implement heaps is binary heaps, which is very similar, and more efficient in practice.

# Sorting using Priority Queues

---



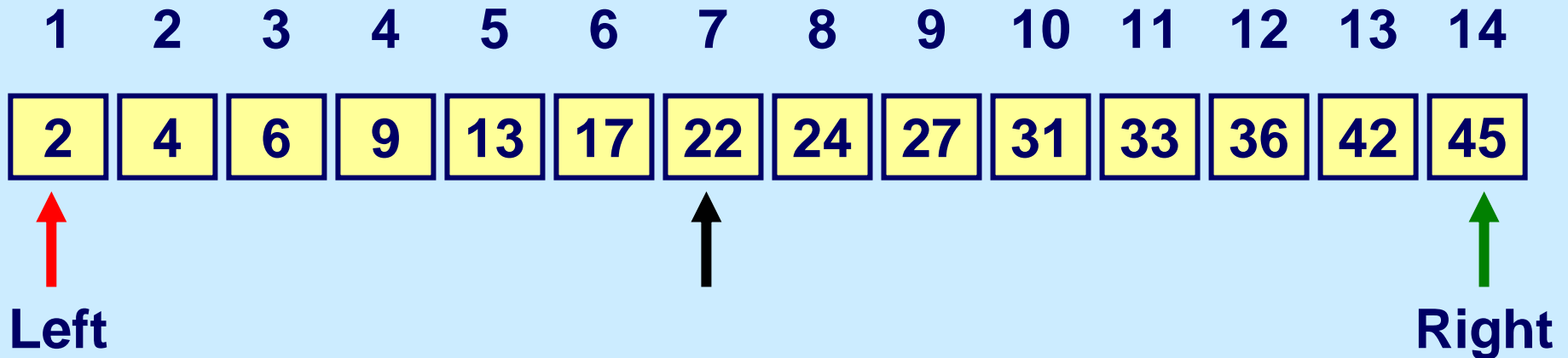
Each find min and delete min take  $O(\log n)$  steps.  
So the running time for sorting is  $O(n \log n)$ .

There are many other comparison based sorting methods that take  $O(n \log n)$ .

# Binary Search

---

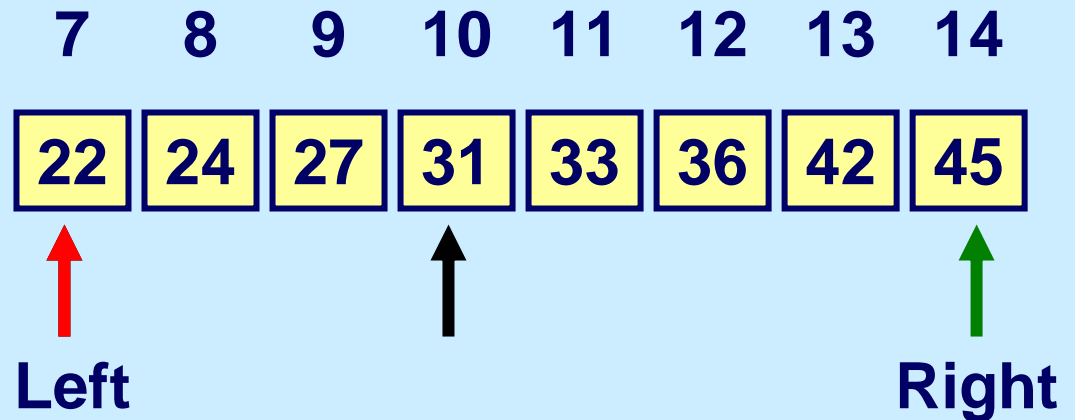
In the ordered list of numbers below, stored in an array, determine whether the number 25 is on the list.



# Binary Search

---

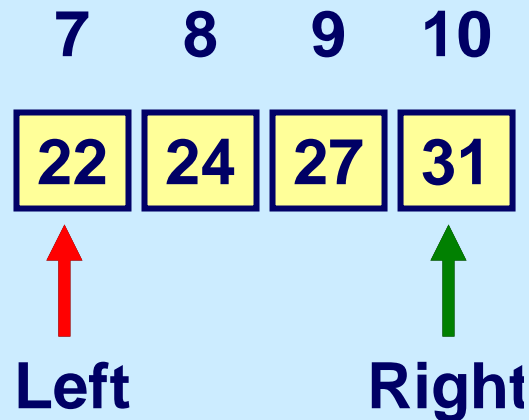
In the ordered list of numbers below, stored in an array, determine whether the number 25 is on the list.



# Binary Search

---

In the ordered list of numbers below, stored in an array, determine whether the number 25 is on the list.



After two more iterations, we will determine that 25 is not on the list, but that 24 and 27 are on the list.

Running time is  $O(\log n)$  for running binary search.

# Summary

---

## Review of data structures

- Lists, stacks, queues, sets
- priority queues
- binary search