

**15.082J/6.855J/ESD.78J**  
**September 14, 2010**

---

**Data Structures**

# Overview of this Lecture

---

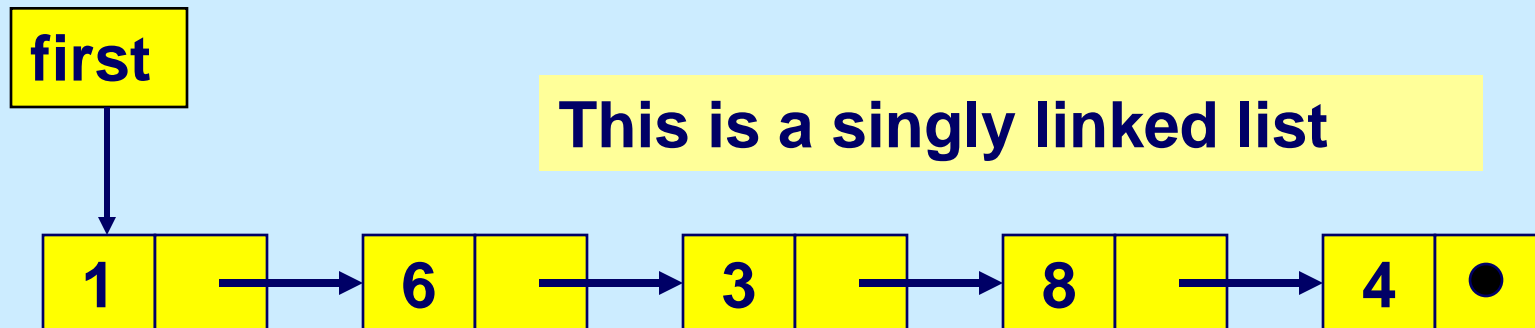
- ◆ **A very fast overview of some data structures that we will be using this semester**
  - **lists, sets, stacks, queues, networks, trees**
  - **a variation on the well known heap data structure**
  - **binary search**
- ◆ **Illustrated using animation**
- ◆ **We are concerned with  $O(\ )$  computation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

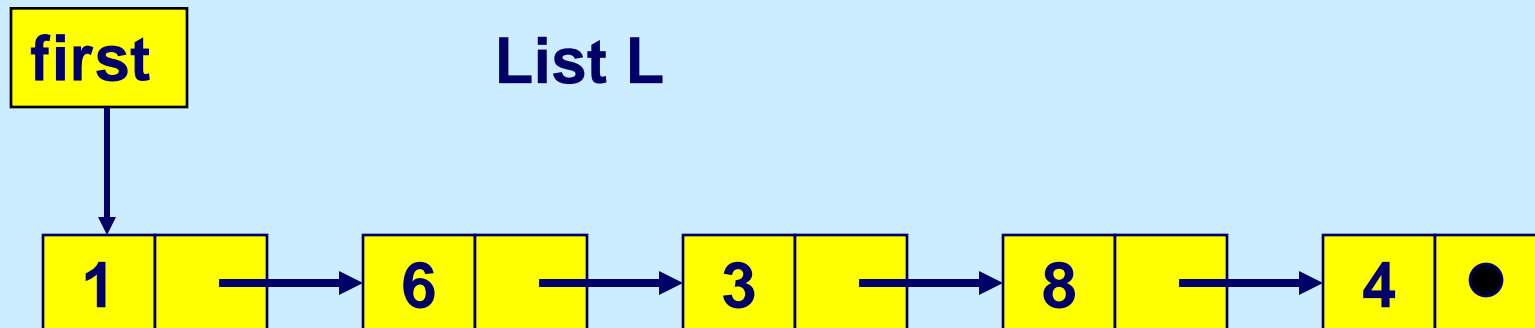
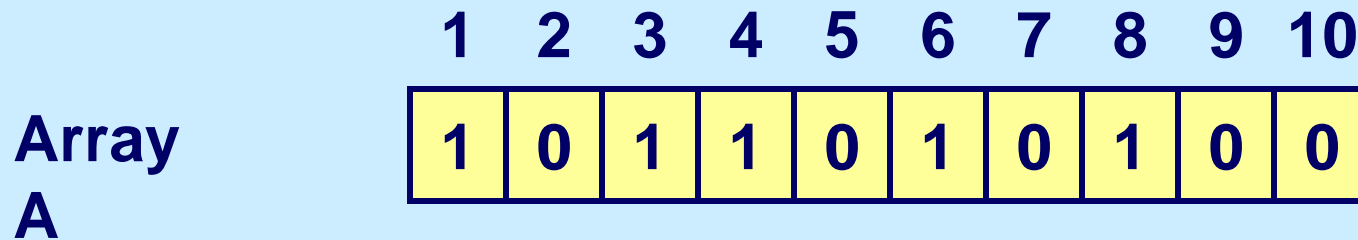


**cells:** hold fields of numbers and pointers to implement lists.

# Representations of subsets of a set

---

subset  $S = \{1, 3, 4, 6, 8\}$

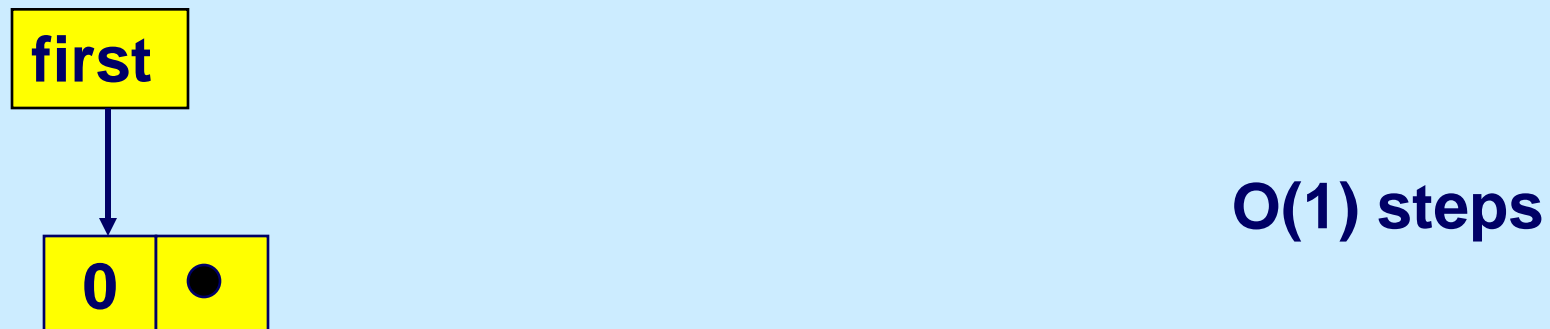


The choice of data structure depends on what operations need to be carried out.

# Example 1: Creating an empty set

---

Initialize: subset  $S = \emptyset$



# Example 2: Is $x \in S$ ?

---

Is  $9 \in S$ ?

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

$O(1)$  steps

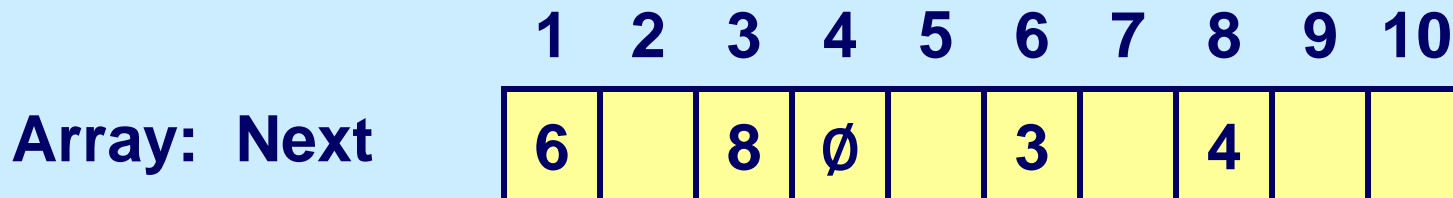
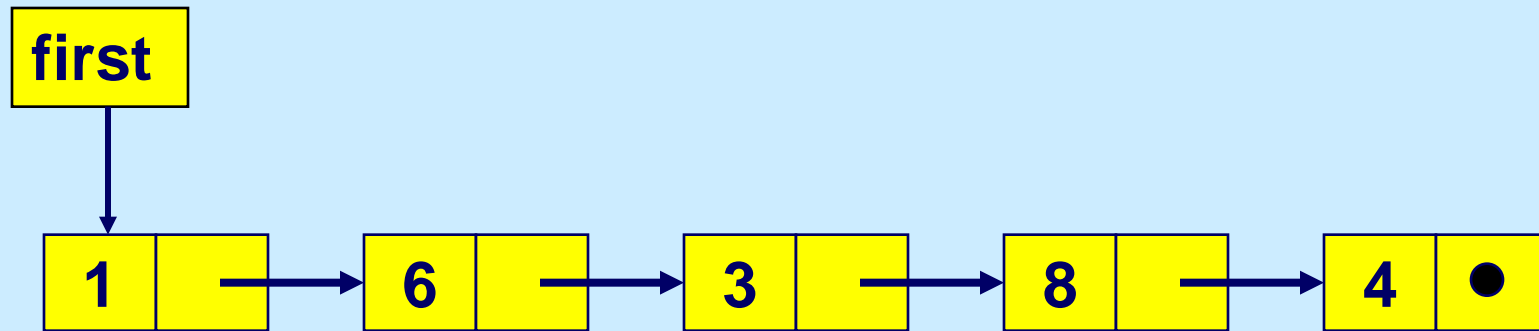
**first**

To determine if  $9 \in S$ ?, one needs to scan the entire list.



$O(n)$  steps

# Representing a linked list as an array



If  $\text{Next}(j)$  is empty, then  $j$  is not on the list

If  $\text{Next}(j) = \emptyset$ , then  $j$  is the last element on the list

# 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  $S \setminus \{s\}$
- ***IsElement( $S, s$ )***: returns true if  $s \in S$
- etc.

**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**

# Abstract data type: SetOperations

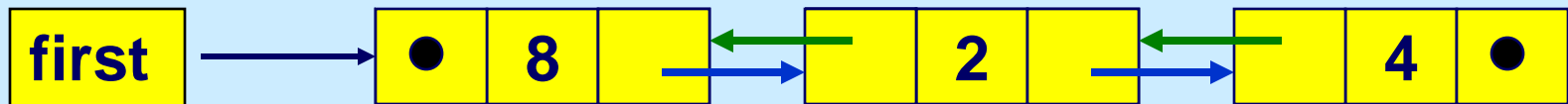
---

Operation (assume  $S \subseteq \{1, 2, 3, \dots, n\}$ ).

- ***initialize(S)***:  $S := \emptyset$
- ***add(S, j)***:  $S := S \cup \{j\}$
- ***delete(S, j)***:  $S := S \setminus \{j\}$
- ***IsElement(S, j)***: returns TRUE if  $s \in S$
- ***FindElement(S)***: if  $S \neq \emptyset$ , returns  $j$  for some  $j \in S$
- ***Next(S, j)***: if  $j \in S$ , it finds the next element after  $j$  on  $S$  (viewed as a list)

# Implementation using doubly linked list

---

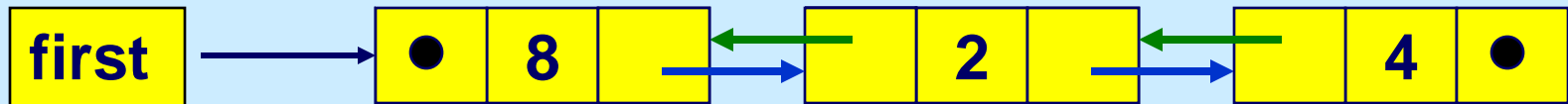


First = 8

	1	2	3	4	5	6	7	8	9	10
Next( )		4		∅				2		

	1	2	3	4	5	6	7	8	9	10
Prev		8		2				∅		

# Add element 5 to the set (in first position)



1 2 3 4 5 6 7 8 9 10

Next( )

	4		∅				2		
--	---	--	---	--	--	--	---	--	--

Prev( )

	8		2				5		
--	---	--	---	--	--	--	---	--	--

Temp := First

First := Elt

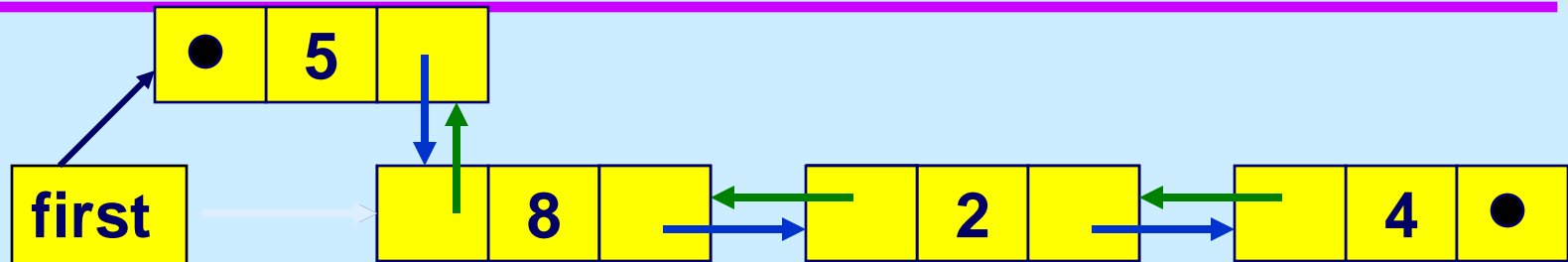
Prev(Temp) := Elt

Prev(Elt) := ∅

Next(Elt) := Temp

Elt	5
First	8
Temp	

# Add element 5 to the set (in first position)



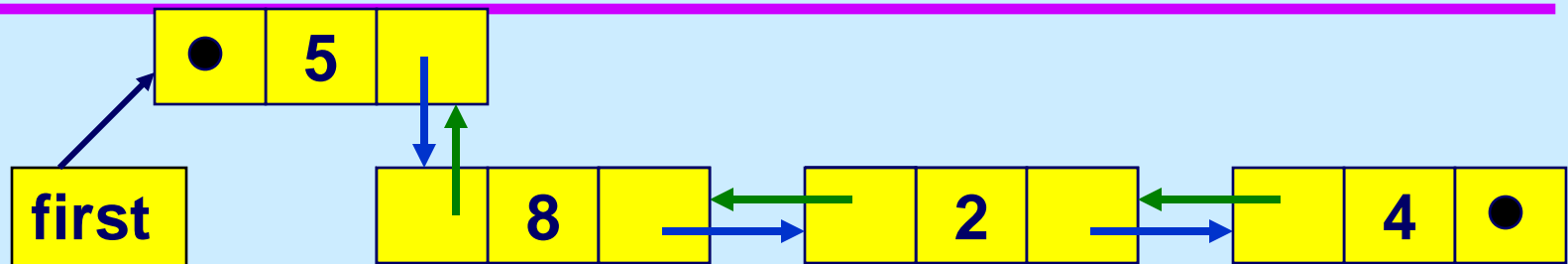
	1	2	3	4	5	6	7	8	9	10
Next( )		4		∅	8			2		
Prev( )		8		2	∅			5		

Elt	5
First	5
Temp	8

**Temp := First**  
**First := Elt**  
**Prev(Temp) := Elt**  
**Prev(Elt) := ∅**  
**Next(Elt) := Temp**

Adding an element to a set takes  $O(1)$  steps using this implementation.

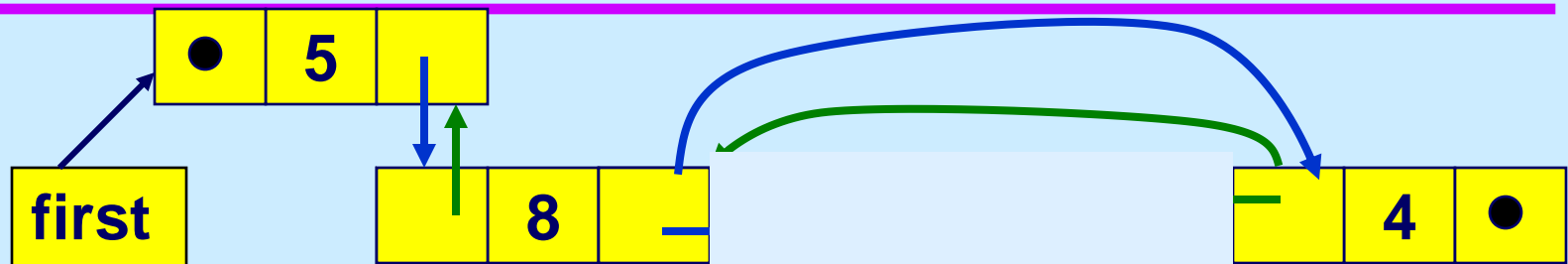
# Delete element 2 from the set (assume it is neither in first or last position)



	1	2	3	4	5	6	7	8	9	10
Next( )		4		∅	8			2		
Prev( )		8		2	∅			5		

Elt	2
First	5
Temp	

# Delete element 2 from the set (assume it is neither in first or last position)



	1	2	3	4	5	6	7	8	9	10
Next( )				∅	8			4		
Prev( )				8	∅			5		

Elt	2
First	5

**Next(Prev(Elt)):=Next(Elt)**

**Prev(Next(Elt)):=Prev(Elt)**

**Prev(Elt) := 0**

**Next(Elt): = 0**

Deleting an element from the set takes  $O(1)$  steps using this implementation.

# Operations using doubly linked lists

---

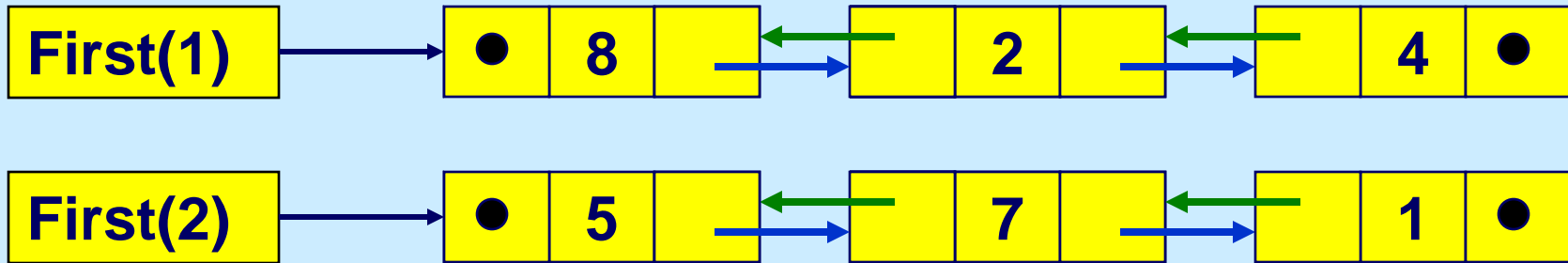
Operation	Number of steps
● <i>initialize(S)</i> :	O(n)
● <i>add(S, j)</i> :	O(1)
● <i>delete(S, j)</i> :	O(1)
● <i>IsElement(S, j)</i> :	O(1)
● <i>FindElement(S)</i> :	O(1)
● <i>Next(S, j)</i> :	O(1)
● <i>Previous(S, j)</i> :	O(1)



# Maintaining disjoint subsets of elements

$$S_1 = \{8, 2, 4\}$$

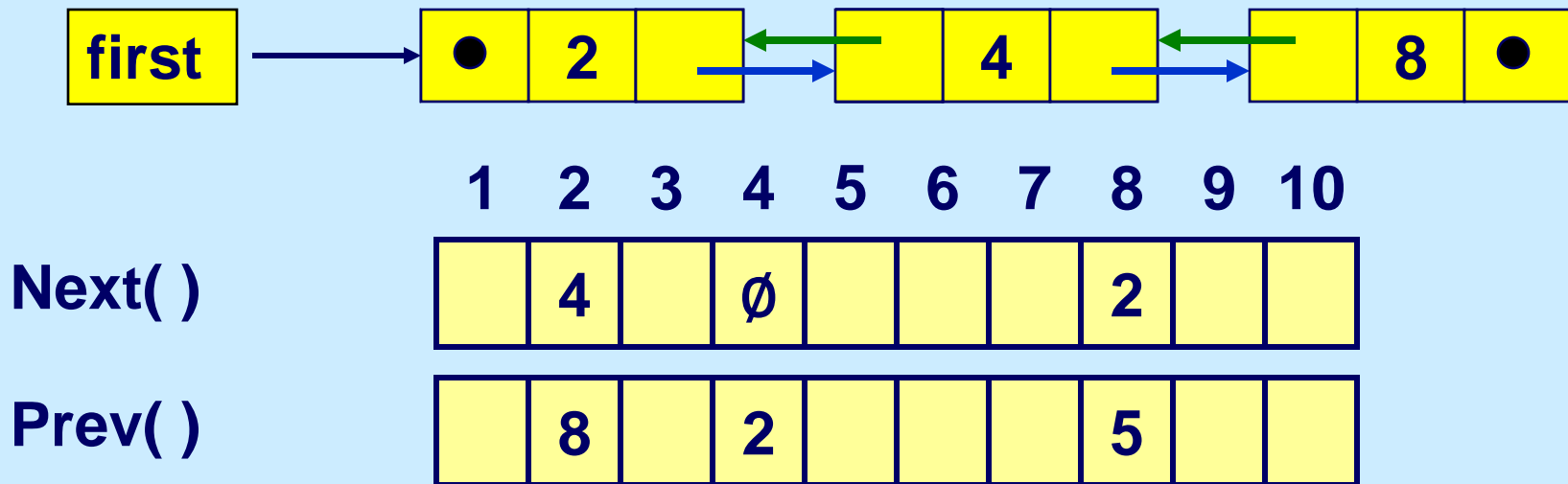
$$S_2 = \{5, 7, 1\}$$



	1	2	3	4	5	6	7	8	9	10
<b>Next( )</b>	$\emptyset$	4		$\emptyset$	7		1	2		
<b>Prev( )</b>	7	8		2	$\emptyset$		5	$\emptyset$		
<b>First( )</b>	8	5								

# Maintaining ordered lists

The doubly linked list is not efficient for maintaining ordered lists of nodes.



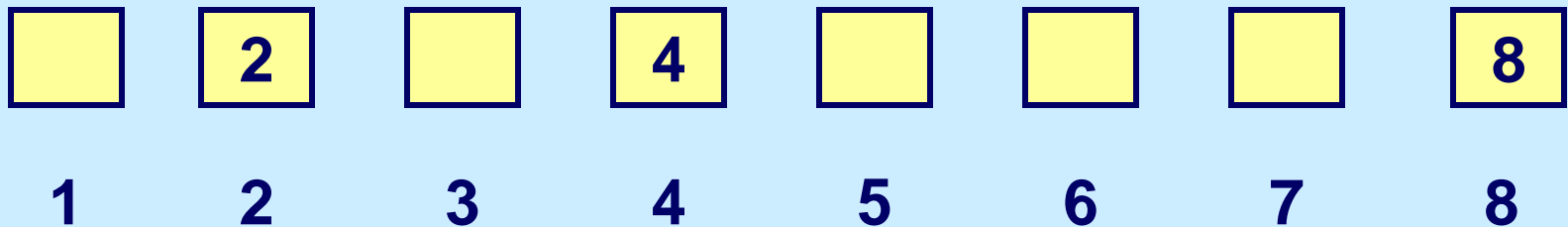
Inserting an element into the set (such as 7) requires finding Prev and Next for 7 (4 and 8), and this requires  $O(n)$  time with this implementation.

# Complete binary tree with n elements

---

Complete binary trees for storing ordered lists of nodes or arcs. We assume that the number of nodes is n (e.g., 8) and the number of arcs is m.

$$S = \{2, 4, 8\} \quad n = 8.$$

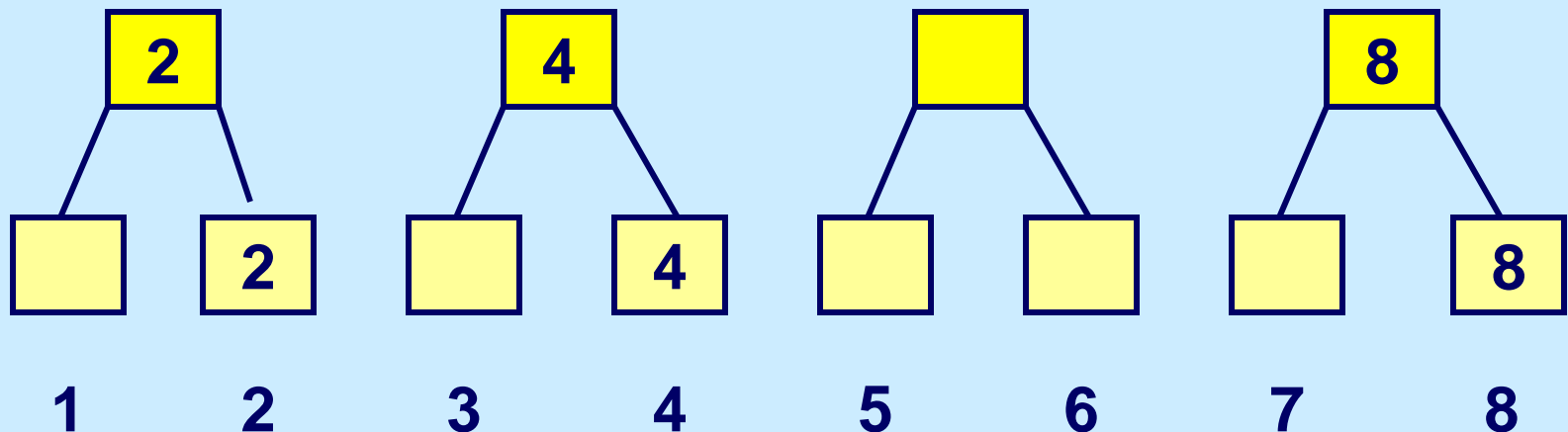


# Complete binary tree with n elements

---

Build up the binary tree. In each parent store the least value of its children.

$S = \{2, 4, 8\}$      $n = 8$ .

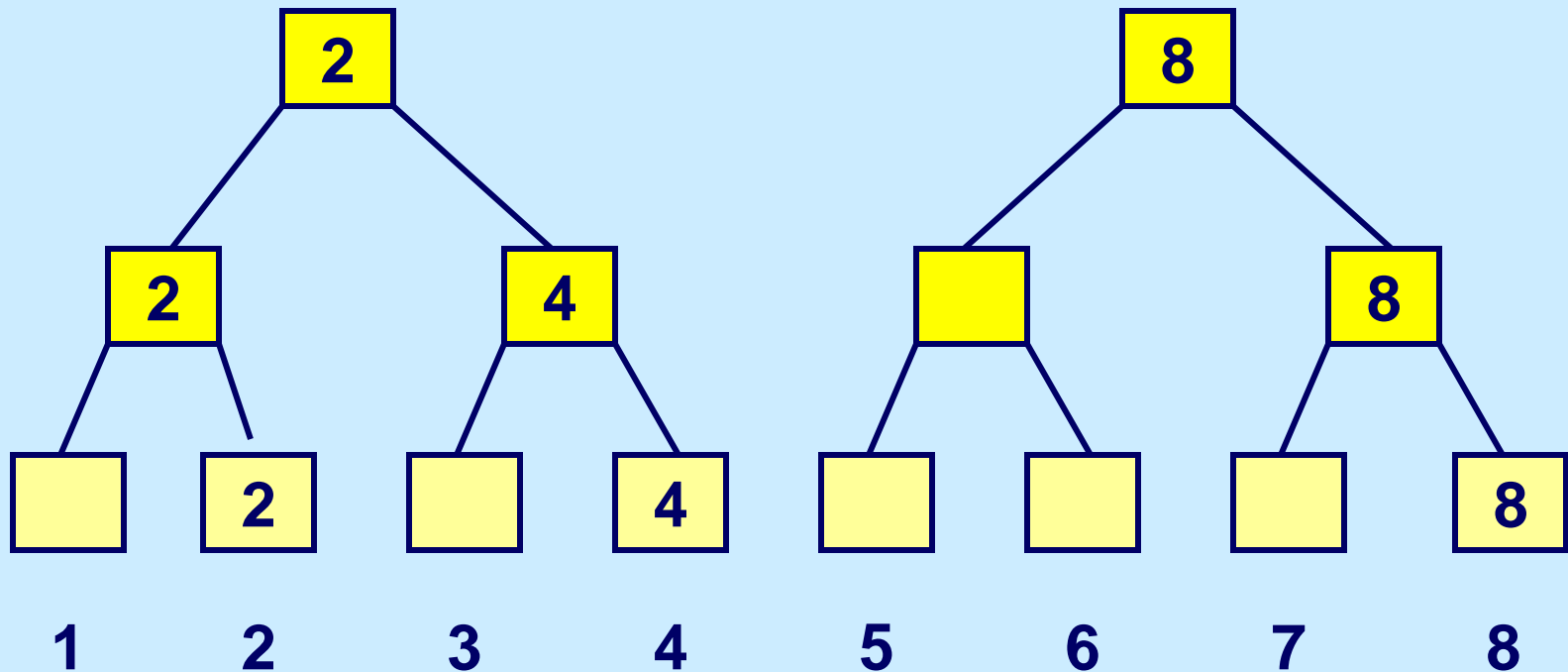


# Complete binary tree with n elements

---

Build up the binary tree. In each parent store the least value of its children.

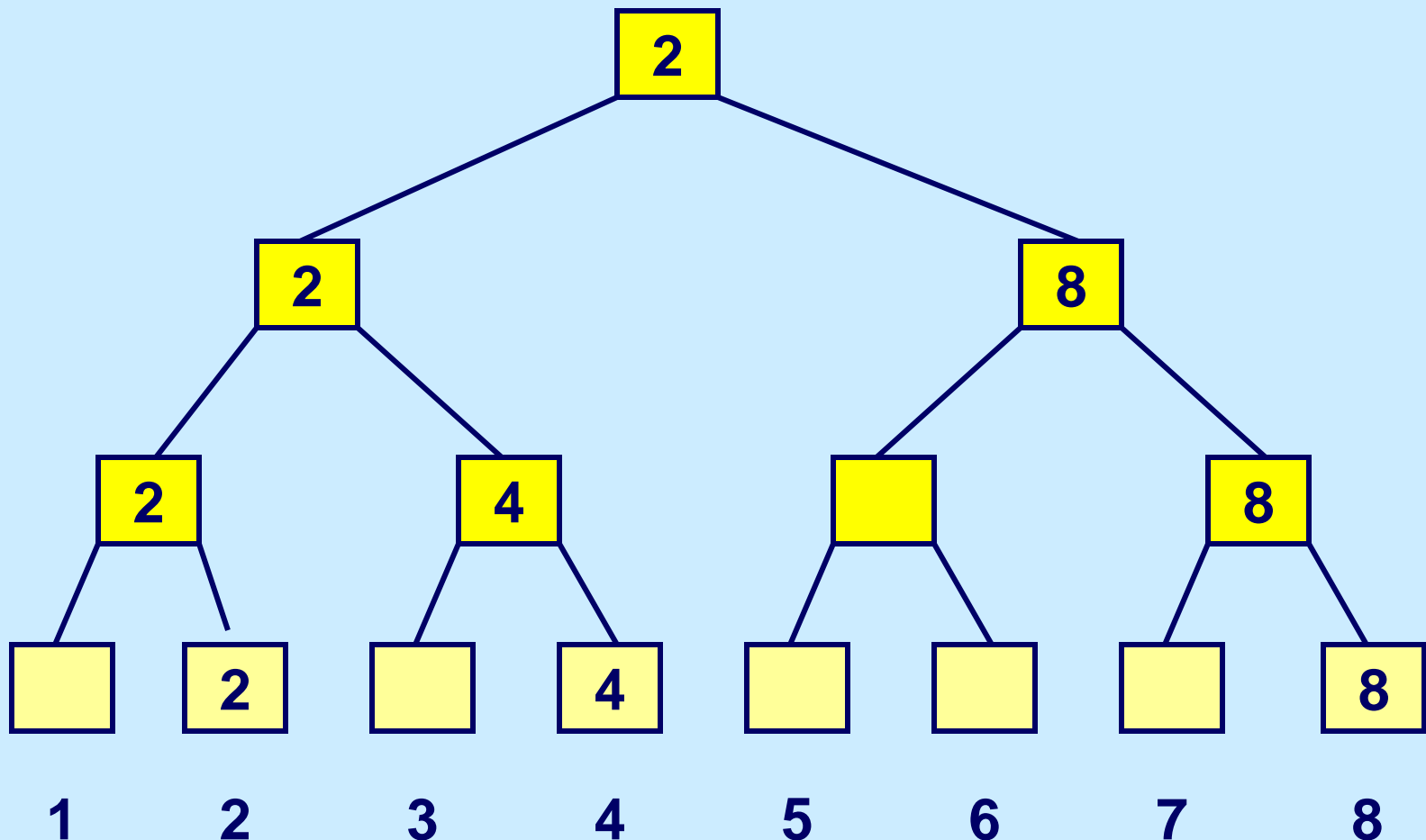
$S = \{2, 4, 8\}$      $n = 8$ .



# Complete binary tree with n elements

Build up the binary tree. In each parent store the least value of its children.

$S = \{2, 4, 8\}$      $n = 8.$

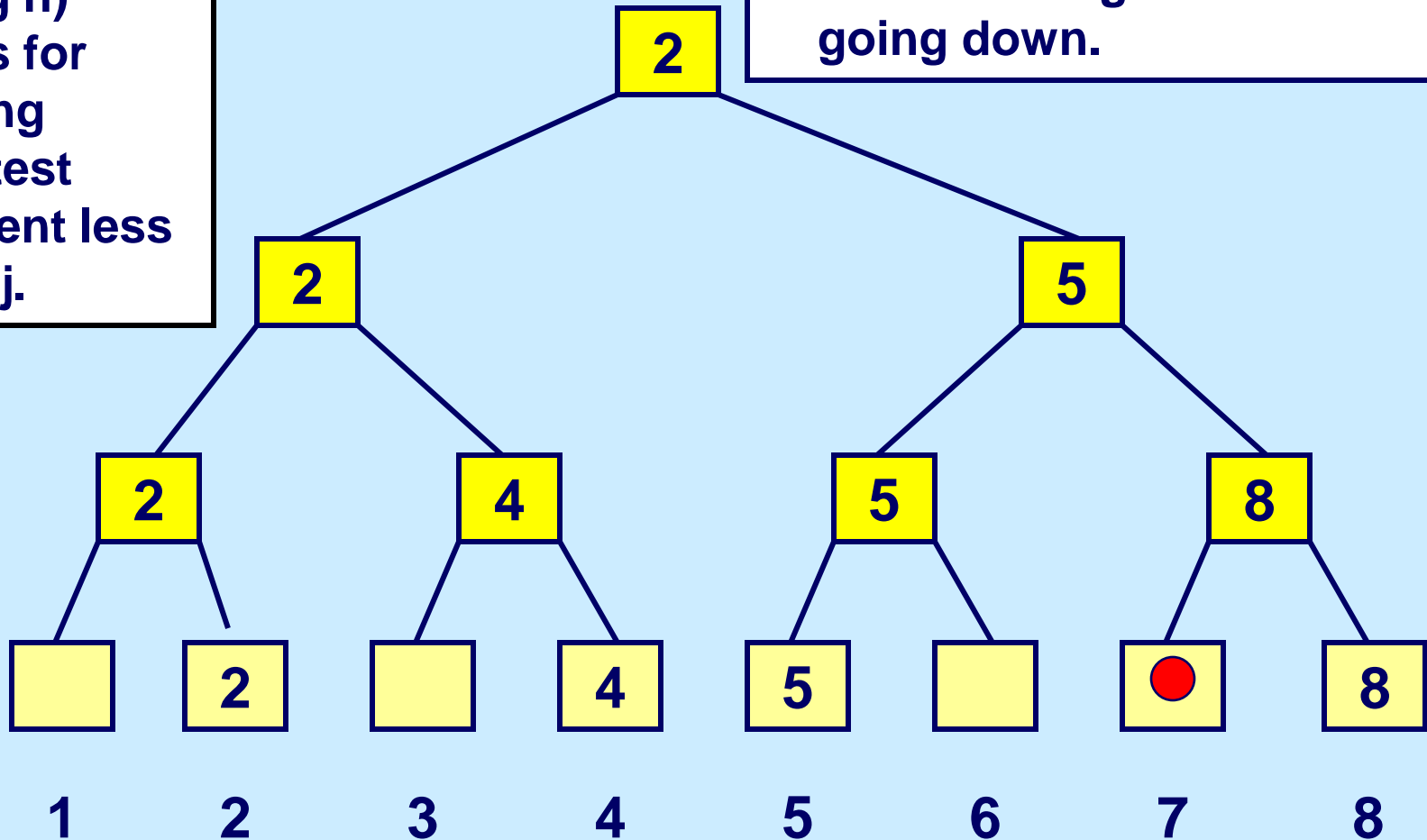


# Find greatest element less than j

e.g., find the greatest element less than 7

start at 7  
go up the tree until a node has label  $< 7$ . Take left branch.  
Choose the largest label child going down.

$O(\log n)$   
steps for  
finding  
greatest  
element less  
than j.

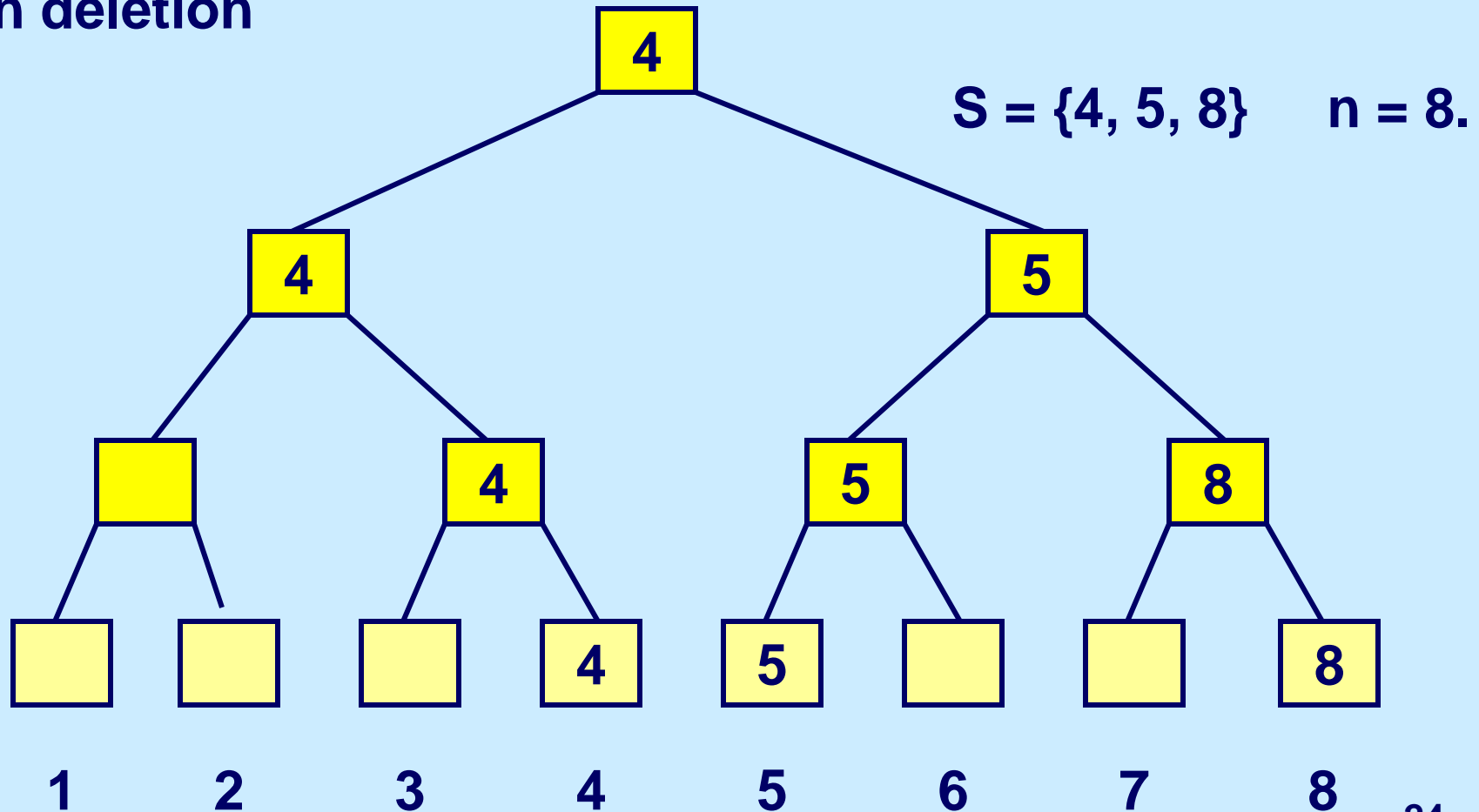


# Delete an element

e.g., delete element 2

$O(\log n)$  steps for  
an deletion

Start at node 2 and update it  
and its ancestors.





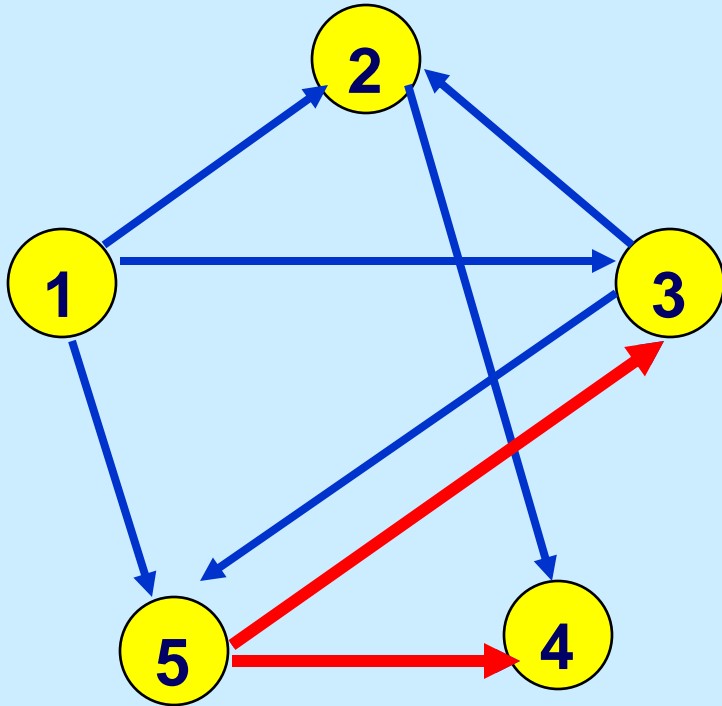
# Operations using complete binary trees

---

Operation	Number of steps
● <i>initialize(S)</i> :	$O(n)$
● <i>add(S, j)</i> :	$O(\log n)$
● <i>delete(S, j)</i> :	$O(\log n)$
● <i>IsElement(S, j)</i> :	$O(1)$
● <i>FindElement(S)</i> :	$O(1)$
● <i>Next(S, j)</i> :	$O(\log n)$
● <i>Previous(S, j)</i> :	$O(\log n)$
● <b>MinElement(S)</b>	$O(1)$
● <b>MaxElement(S)</b>	$O(\log n)$

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

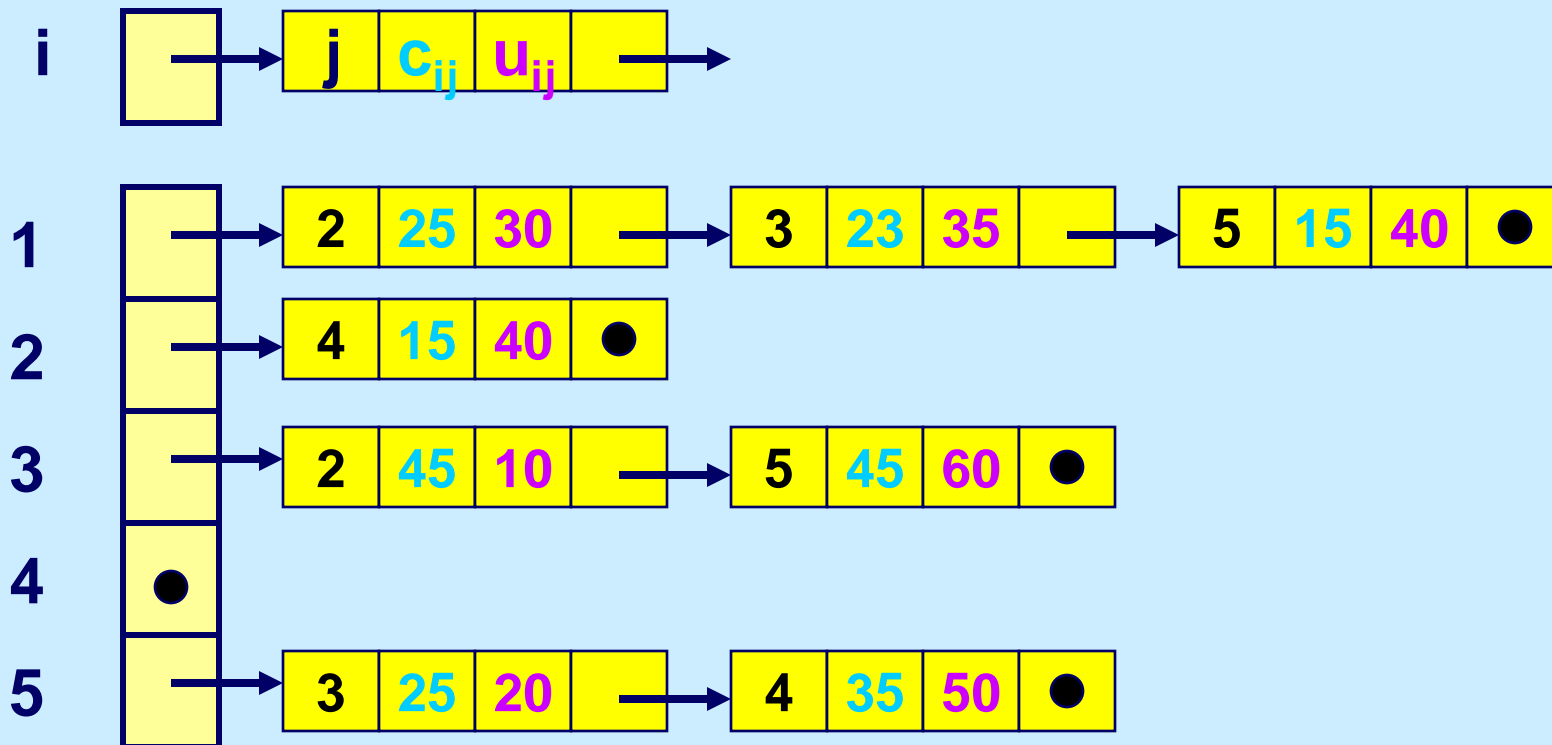
**Note:** these sets are usually static. They stay the same.

**Common operations:** scanning the list  $A(i)$  one arc at a time starting at the first arc.

# Storing Arc Lists: A(i)

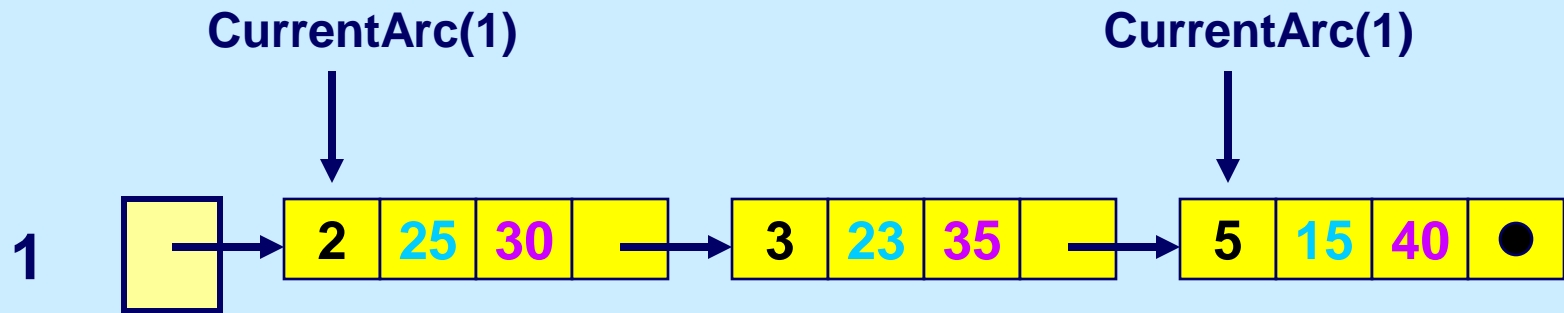
## Operations permitted

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



# Scanning the arc list

**CurrentArc(i)** is a pointer to the arc of A(i) that is being scanned or is the next to be scanned.

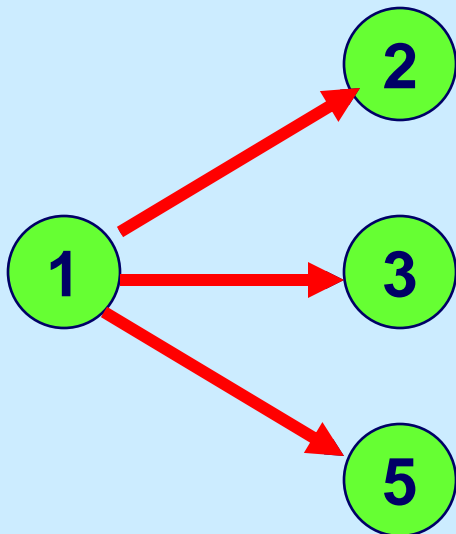
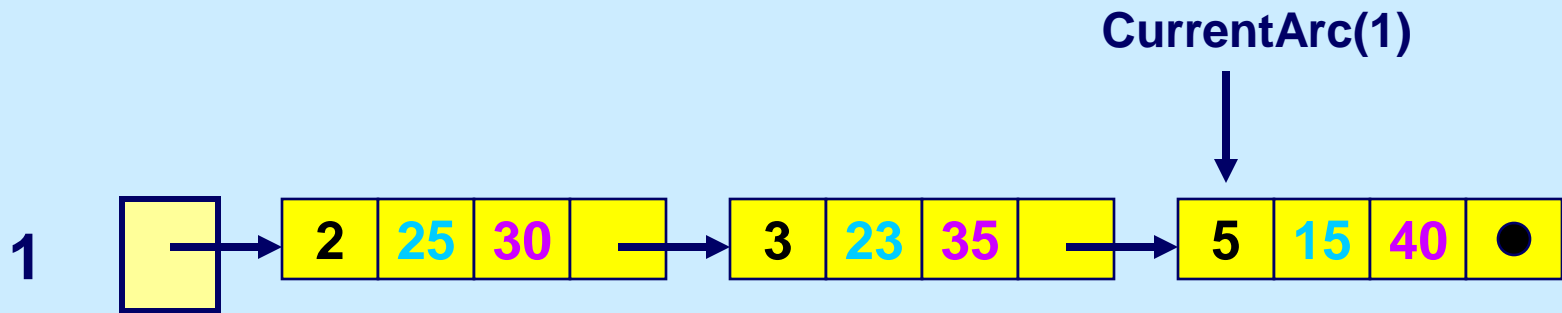


Initially,  $\text{CurrentArc}(i)$  is the first arc of  $A(i)$

After  $\text{CurrentArc}(i)$  is fully scanned,  
 $\text{CurrentArc}(i) := \text{Next}(\text{CurrentArc}(i))$

# Scanning the arc list

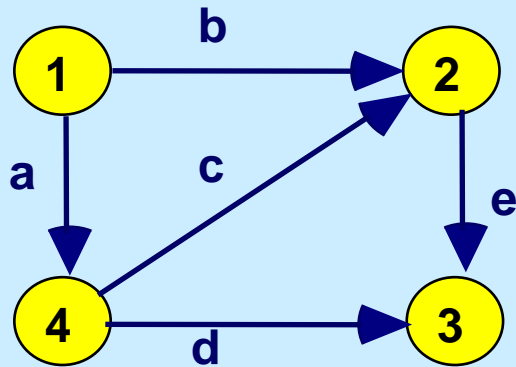
**CurrentArc(i)** is a pointer to the arc of A(i) that is being scanned or is the next to be scanned.



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)



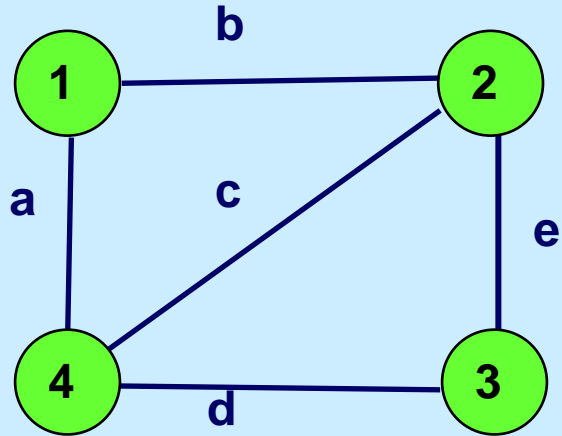
A Directed Graph

	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)



An Undirected Graph

	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

- 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
- 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



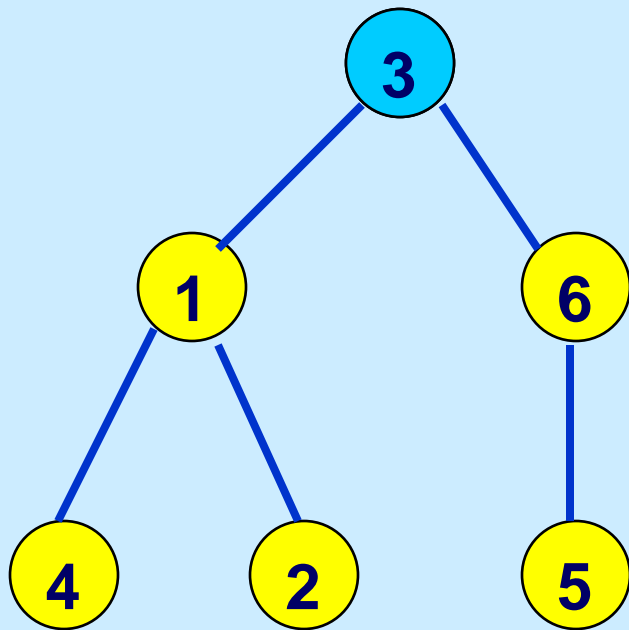
# Trees

---

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**.

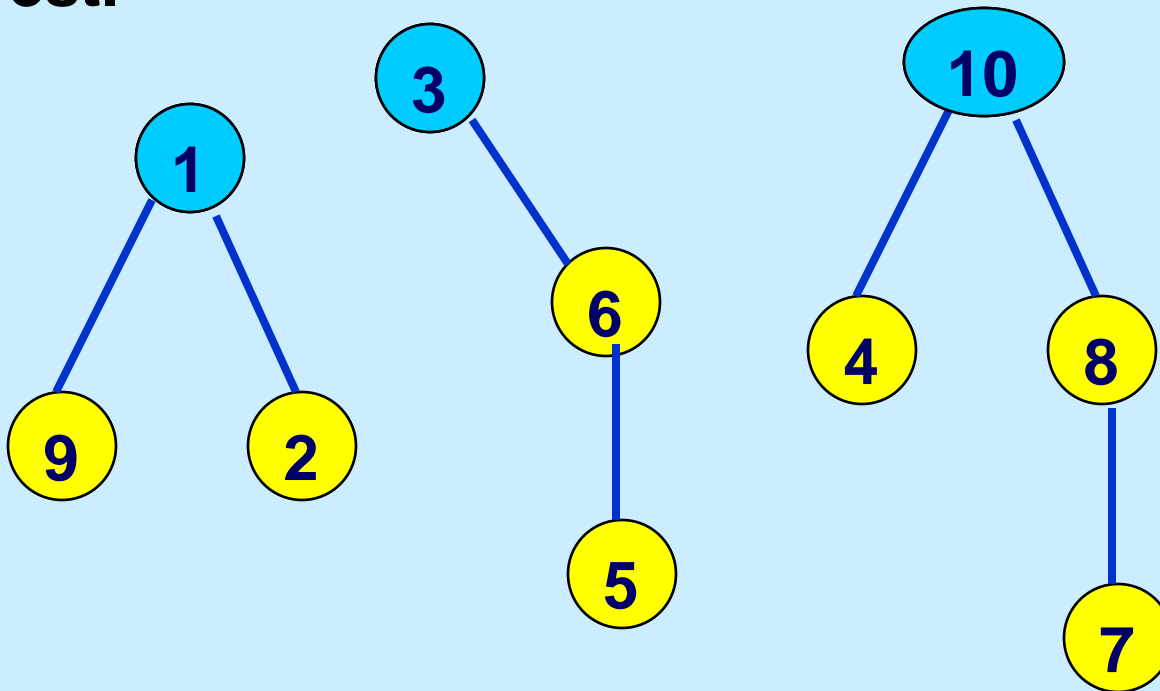
(In principle, any node can be selected for the root.)

# Forest

---

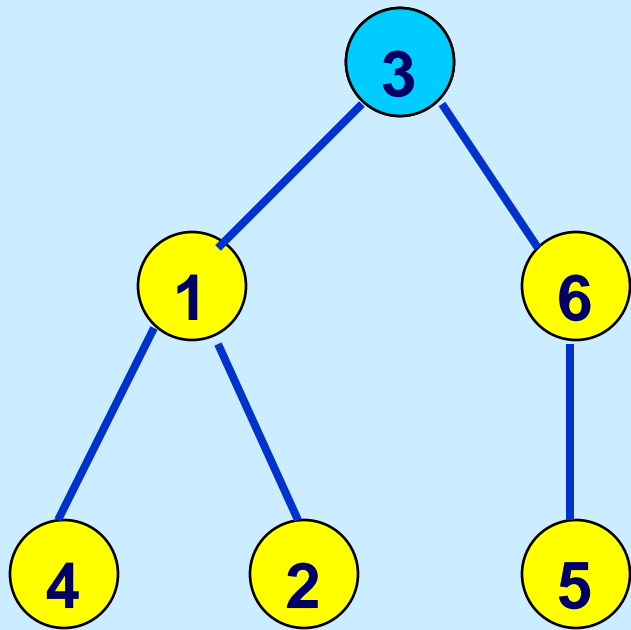
A **forest** is an acyclic graph that includes all of the nodes.

A **subtree** of a forest is a connected component of the forest.

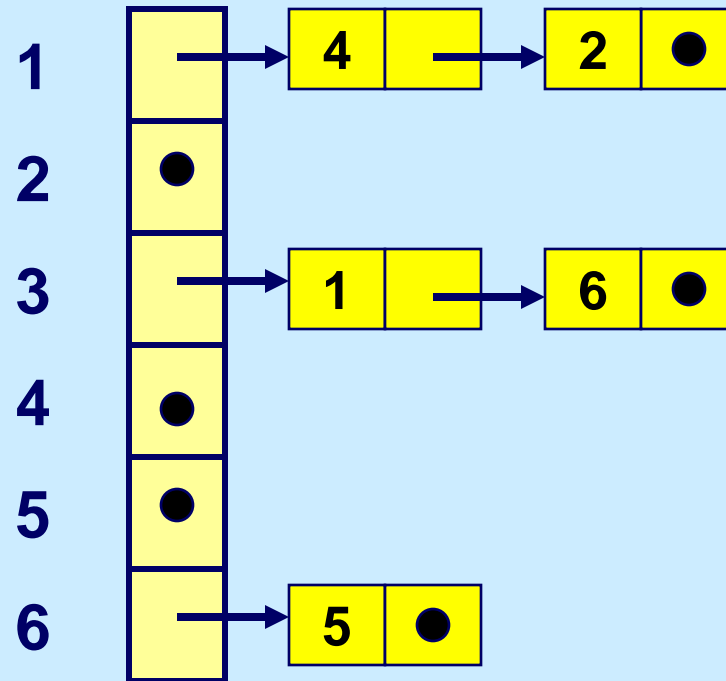


To store trees efficiently, each subtree has a **root node**.

# One way of storing subtrees



## Lists of children

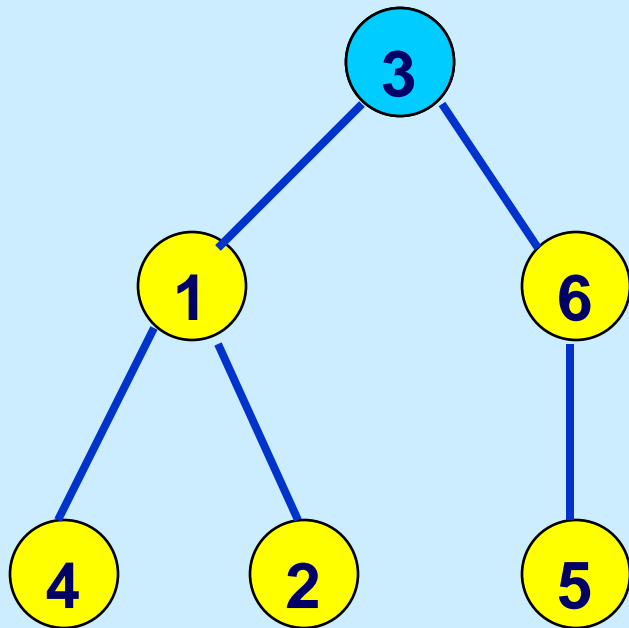


node	1	2	3	4	5	6
parent	3	1		1	6	3

parent (predecessor) array

# On storing trees

---



**Trees are important parts of flow algorithms**

**Some data structures are expressed as trees**

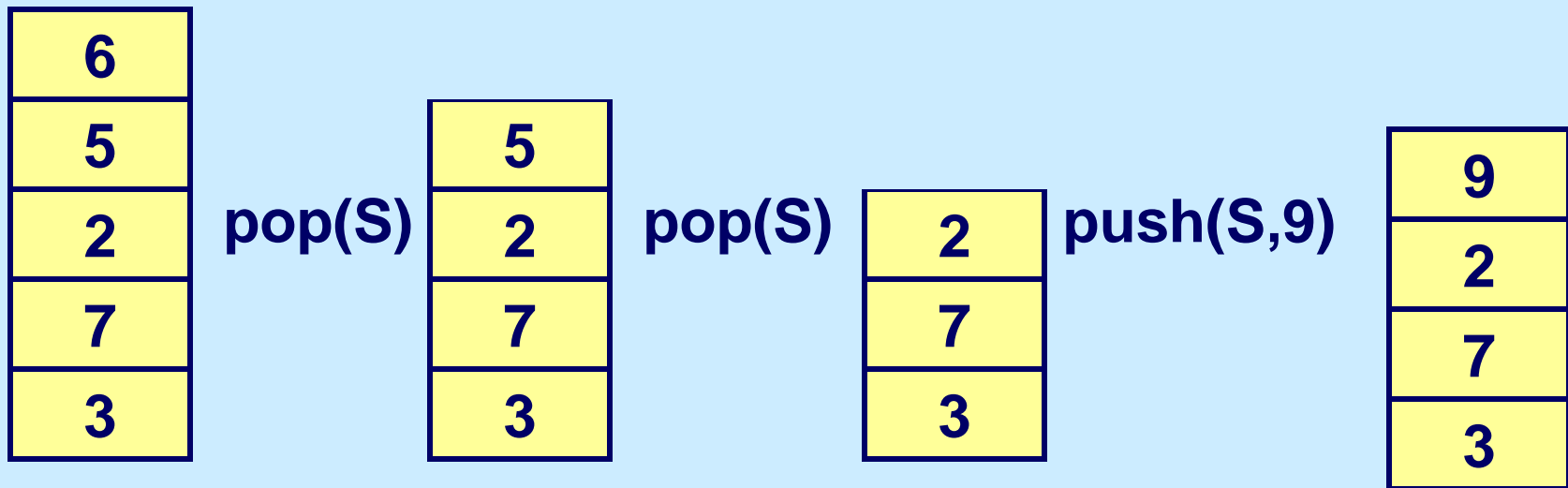
**The best implementation of trees depends on what operations need to be performed in the abstract data type.**

# 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)

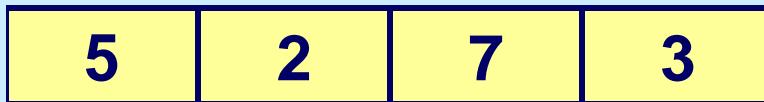
---

## 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



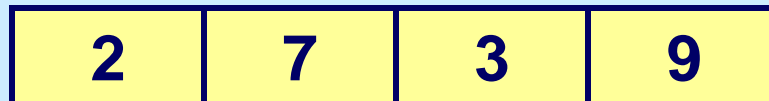
Delete(Q)



Delete(Q)



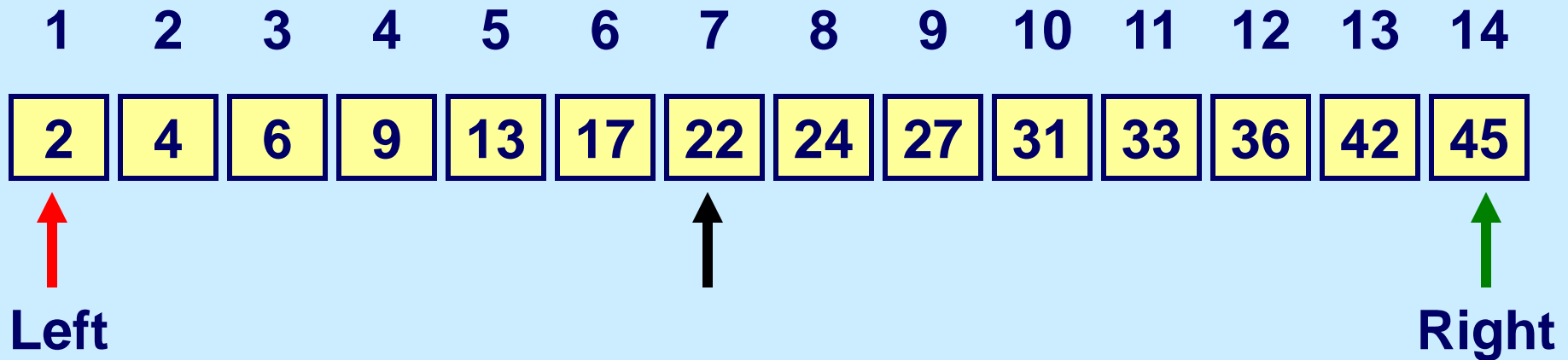
Insert(Q,9)



# Binary Search

---

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



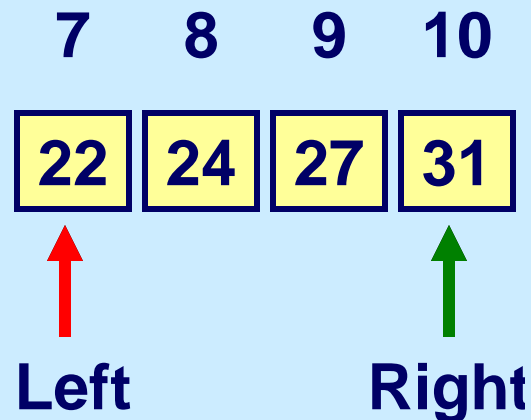




# Binary Search

---

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, sets, complete binary trees, trees
- queues, stacks
- binary search

**Next Lecture: Search algorithms**

MIT OpenCourseWare  
<http://ocw.mit.edu>

15.082J / 6.855J / ESD.78J Network Optimization  
Fall 2010

For information about citing these materials or our Terms of Use, visit: <http://ocw.mit.edu/terms>.