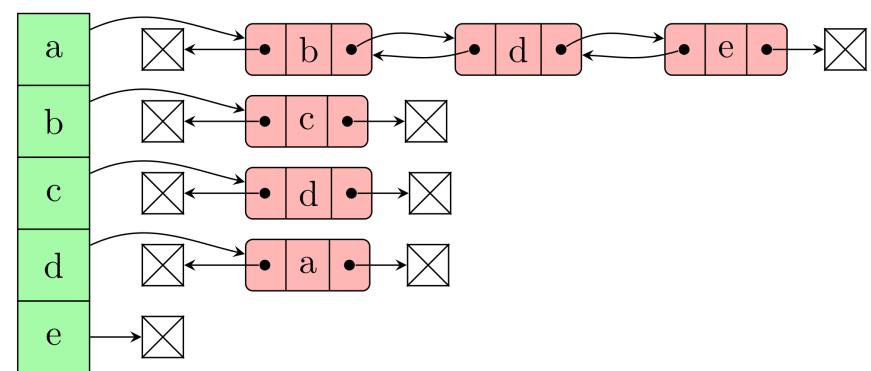
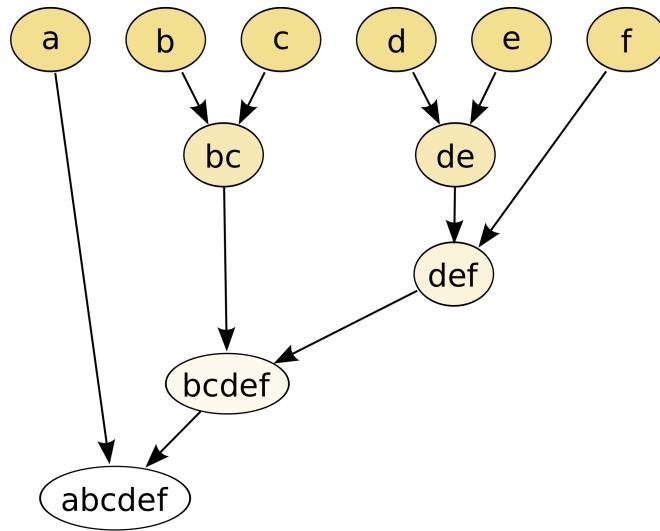


Data Structures Review



Amin Milani Fard

NYIT

Software Design Goals

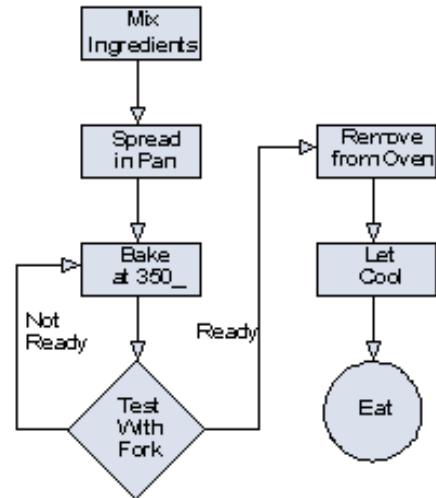
- What goals come to your mind?

- Correctness
- Efficiency (run-time)
- Reliability, robustness, and usability
- Maintainability and reusability
- ...

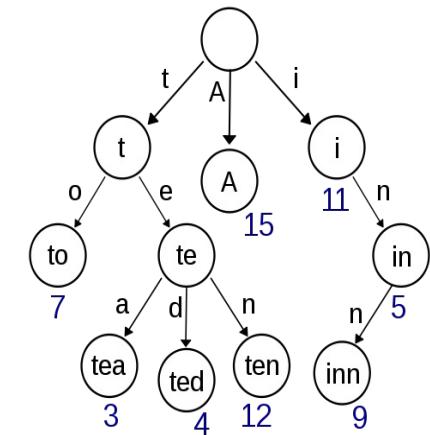
How do data structures and algorithms apply to above goals?

- modular, small-sized program pieces = easier to understand, debug, figure out correctness/efficiency

Algorithms



Data Structures



Data Structures + Algorithms = Programs

Algorithm analysis – determining how long an algorithm will take to solve a problem

Who cares? Aren't computers fast enough and getting faster?

What is an Algorithm

An **algorithm** is a set of **instructions/steps** that solves a particular problem.

Each algorithm operates on **input data**.

Each algorithm produces an **output result**.

Each algorithm can be classified by how long it takes to run on a particular input.

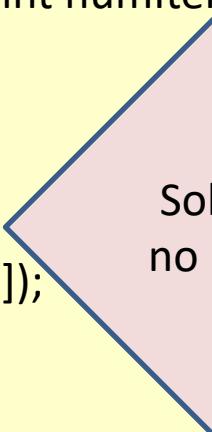
Each algorithm can be classified by the quality of its results.

Data Structures

A **data structure** is the **data** that's operated on by an algorithm to solve a problem.

Sometimes an algorithm can operate just on the **data** passed in to it:

```
void printNumbersBackward(int array[], int numItems)
{
    while (numItems > 0)
    {
        --numItems;
        System.out.println(array[numItems]);
    }
}
```



Solves the problem with no new variables or data!

Other times an algorithm will have to create its own **secondary data structures** to solve a problem.

A Data Structure for Facebook

Imagine that you're building a social network website like **Facebook**...



Friend A	Friend B
Danny Hsu	Jenny Oh
Danny Hsu	Rich Lim
Jenny Oh	David Sun
Danny Hsu	Carey Nash
Carey Nash	Scott Wilhelm
Melani Kan	Danny Hsu
Carey Nash	David Small
Jenny Oh	Len Kleinrock
Jenny Oh	Mario Giso
Mario Giso	Rich Nguyen

For example, this line indicates that Danny and Jenny are friends.

How would you do it?

Well, one *data structure* we could use would be a long list of every pair of friends...

Now how could I find out if Jenny Oh is a friend of Mario Giso?

Hmmm. But if we had 100 million users with 10 billion friendships, that might be a bit too slow! ☺

Well, we could search through every pair until we find the friendship we are looking for...

A Better Friendship Data Structure

What if instead we assigned each of our **N** users a number...

And kept an **alphabetized list** of each user → number pair.

User	Number
Carey Nash	0
Danny Hsu	1
David Small	2
David Sun	3
Jenny Oh	4
Len Kleinrock	5

M	FriendA	FriendB
M	Danny Hsu	Jenny Oh
R	David Sun	Carey Nash
R	Jenny Oh	David Sun
S	Danny Hsu	Carey Nash
	Carey Nash	Scott Wilhelm
	Melani Kan	Danny Hsu
	Carey Nash	David Small

Now given a user's name, how can we quickly find their number?
Right – just use a Binary Search!

1. Look up their numbers A, B
 2. Check (A,B) or (B,A) in our array
 3. If we find a*, they're buddies!

* means parent relationships!

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

A Better Friendship Data Structure

So what we've done is created a new **data structure** that makes it faster to determine if two people are friends!

Coupled with our new, more efficient **algorithm**, we can determine if two people are friends!

User Number

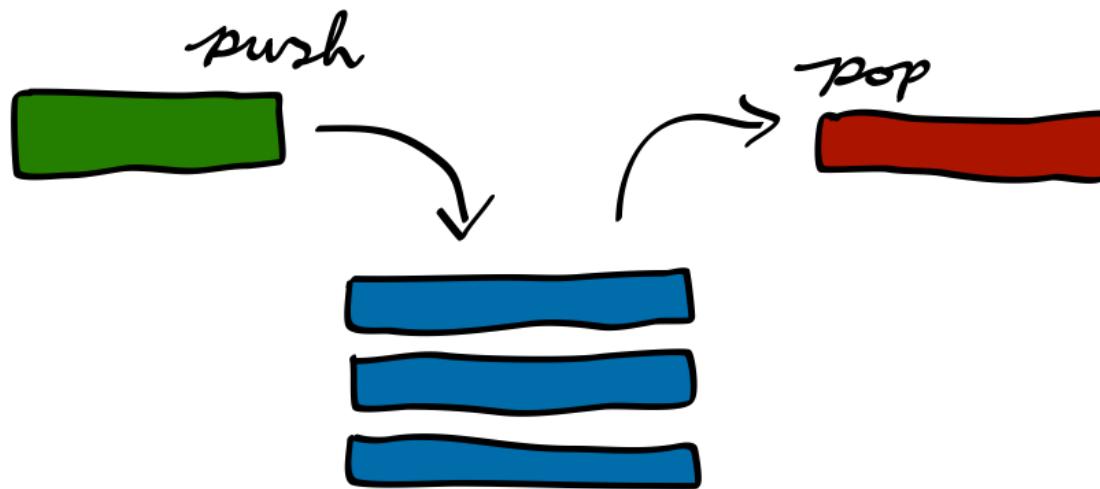
Carey Nash	0
Danny Hsu	1
David Small	2
David Sun	3
Jenny Oh	4
Len Kleinrock	5
Mario Giso	6
Melani Kan	7

Friend-finding Algorithm:

1. Look up their numbers A, B
2. Check (A,B) or (B,A) in our array
3. If we find a *, they're buddies!

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

Stacks



Slides partly from Data Structures and Algorithms in Java, by
M. T. Goodrich, et. Al.

Abstract Data Types (ADTs)

- An abstract data type (ADT) is an abstraction of a data structure
- An ADT specifies:
 - Data stored
 - Operations on the data
 - Error conditions associated with operations
- Example: ADT modeling a simple stock trading system
 - The data stored are buy/sell orders
 - The operations supported are
 - ◆ order **buy**(stock, shares, price)
 - ◆ order **sell**(stock, shares, price)
 - ◆ void **cancel**(order)
 - Error conditions:
 - ◆ Buy/sell a nonexistent stock
 - ◆ Cancel a nonexistent order

The Stack ADT

- The **Stack** ADT is a data structure in which data is added and removed at only one end called the top
- Insertions and deletions follow the **last-in first-out** scheme
- Main stack operations:
 - **push(object)**: inserts an element
 - **object pop()**: removes and returns the last inserted element
- Auxiliary stack operations:
 - object **peek()**: returns the last inserted element without removing it
 - integer **size()**: returns the number of elements stored
 - boolean **isEmpty()**: indicates whether no elements are stored

Example

Method	Return Value	Stack Contents
push(5)	–	(5)
push(3)	–	(5, 3)
size()	2	(5, 3)
pop()	3	(5)
isEmpty()	false	(5)
pop()	5	()
isEmpty()	true	()
pop()	null	()
push(7)	–	(7)
push(9)	–	(7, 9)
top()	9	(7, 9)
push(4)	–	(7, 9, 4)
size()	3	(7, 9, 4)
pop()	4	(7, 9)
push(6)	–	(7, 9, 6)
push(8)	–	(7, 9, 6, 8)
pop()	8	(7, 9, 6)

Applications of Stacks

Some direct applications:

- Page-visited history in a Web browser
- Undo sequence in a text editor
- Chain of method calls
- Evaluating postfix expressions (e.g., $xy+$)

Some indirect applications

- Auxiliary data structure for some algorithms (e.g., Depth First Search algorithm)
- Component of other data structures

Stack Errors

Stack Overflow

- An attempt to add a new element in an already full stack is an error
- A common mistake often made in stack Implementation

Stack Underflow

- An attempt to remove an element from the empty stack is also an error
- Again, a common mistake often made in stack implementation

Array-based Stack

- A simple way of implementing the Stack ADT uses an array
- We add elements from left to right
- A variable keeps track of the index of the top element

Algorithm *size()*

return $t + 1$

Algorithm *pop()*

if *isEmpty()* then

return null

else

$t \leftarrow t - 1$

return $S[t + 1]$



Array-based Stack (cont.)

- ❑ The array storing the stack elements may become full
- ❑ A push operation will then throw a **FullStackException**
 - Limitation of the array-based implementation
 - Not intrinsic to the Stack ADT

Algorithm *push(o)*

```
if  $t = S.length - 1$  then  
    throw  
IllegalStateException  
else  
     $t \leftarrow t + 1$   
     $S[t] \leftarrow o$ 
```



Performance and Limitations

❑ Performance

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

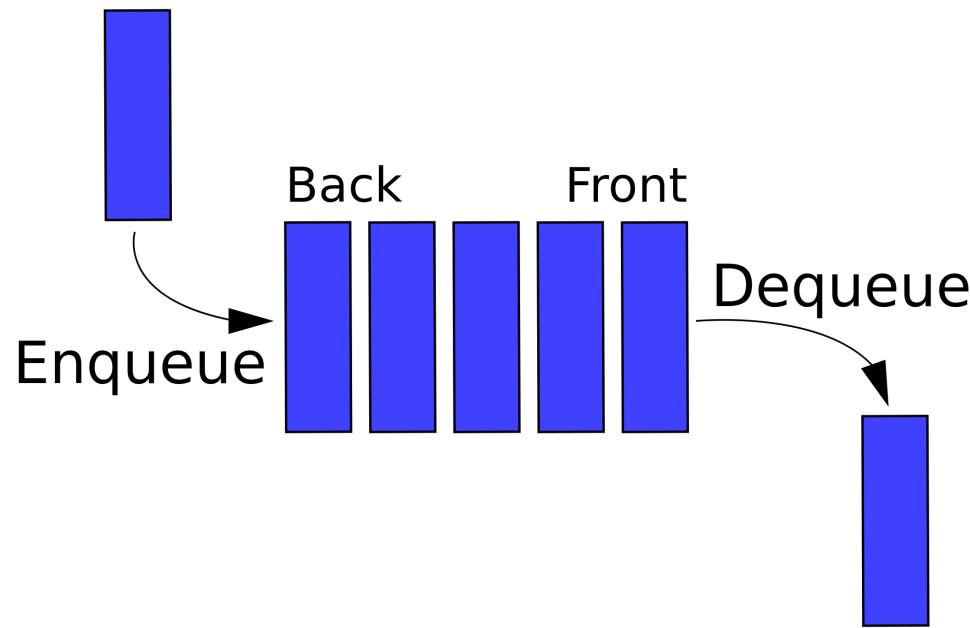
❑ Limitations

- The maximum size of the stack must be defined a priori and cannot be changed
- Trying to push a new element into a full stack causes an implementation-specific exception

Parentheses Matching

- ❑ Each "(", "{", or "[" must be paired with a matching ")" , "}" , or "["
 - correct: ()(()){([()])}
 - correct: ((()(())){([()])})
 - incorrect:)(()){([()])}
 - incorrect: ({[]})
 - incorrect: (

Queues



Slides partly from Data Structures and Algorithms in Java, by
M. T. Goodrich, et. Al.

The Queue ADT

- The Queue is a data structure in which data insertions are at the rear of the queue and removals are at the front of the queue
- Insertions and deletions follow the **first-in first-out (FIFO)** scheme
- Main queue operations:
 - object **enqueue(object)**: inserts an element at the end of the queue
 - object **dequeue()**: removes and returns the element at the front of the queue
- Auxiliary queue operations:
 - object **first()**: returns the element at the front without removing it
 - integer **size()**: returns the number of elements stored
 - boolean **isEmpty()**: indicates whether no elements are stored
- Boundary cases:
 - Attempting the execution of **dequeue** or **first** on an empty queue returns **null**

Queue Operations

- A queue should implement at least the first two of these operations:
 - **insert** – insert item at the back of the queue
 - **remove** – remove an item from the front
 - **first/peek** – return the item at the front of the queue without removing it
- Like stacks, it is assumed that these operations will be implemented efficiently
 - That is, in constant time

Queue example

<i>Operation</i>	<i>Output</i>	<i>Q</i>
enqueue(5)	—	(5)
enqueue(3)	—	(5, 3)
dequeue()	5	(3)
enqueue(7)	—	(3, 7)
dequeue()	3	(7)
first()	7	(7)
dequeue()	7	()
dequeue()	<i>null</i>	()
isEmpty()	<i>true</i>	()
enqueue(9)	—	(9)
enqueue(7)	—	(9, 7)
size()	2	(9, 7)
enqueue(3)	—	(9, 7, 3)
enqueue(5)	—	(9, 7, 3, 5)
dequeue()	9	(7, 3, 5)

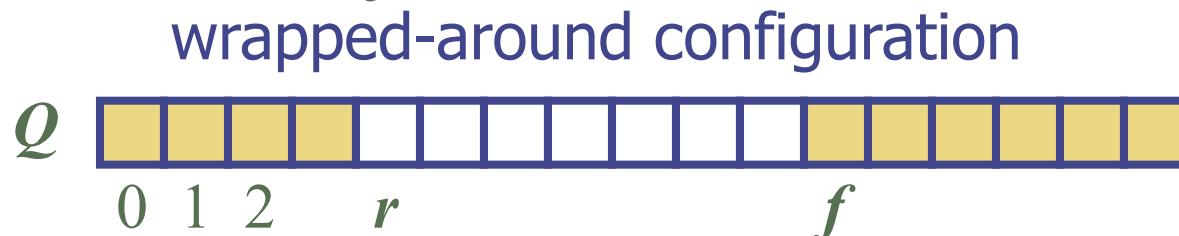
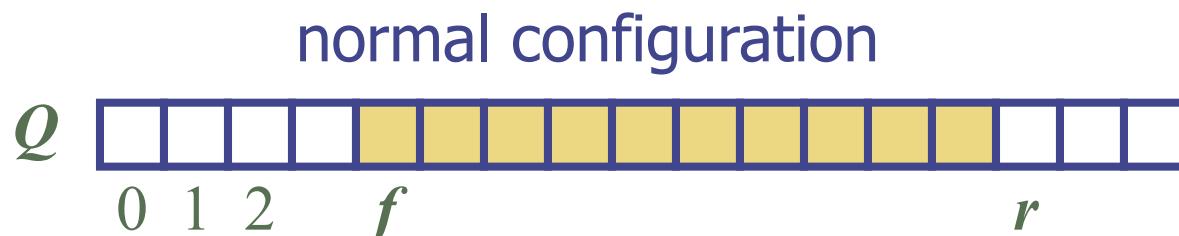
Applications of Queues

- Direct applications
 - Waiting lists, bureaucracy
 - Access to shared resources (e.g., printer)
 - Server requests (Instant messaging servers, Database requests)
 - Scheduling CPU jobs

- Indirect applications
 - Auxiliary data structure for algorithms
 - Component of other data structures

Array-based Queue

- ❑ Use an array of size N in a circular fashion
- ❑ Two variables keep track of the front and size
 - f index of the front element
 - sz number of stored elements
- ❑ When the queue has fewer than N elements, array location $r = (f + sz) \bmod N$ is the first empty slot past the rear of the queue



Queue Operations

- We use the modulo operator (remainder of division)

Algorithm *size()*

return *sz*

Algorithm *isEmpty()*

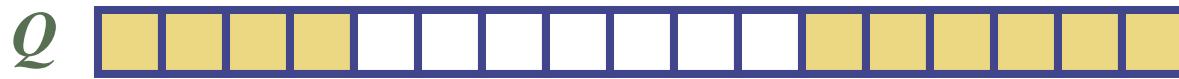
return (*sz* == 0)



0 1 2

f

r



0 1 2

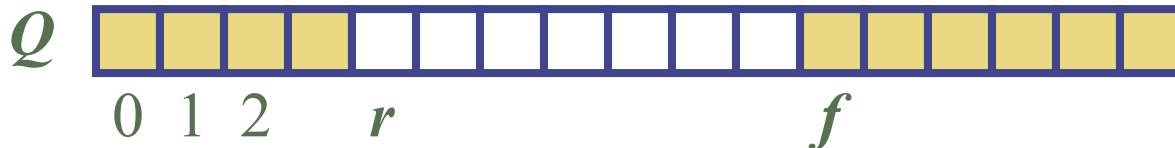
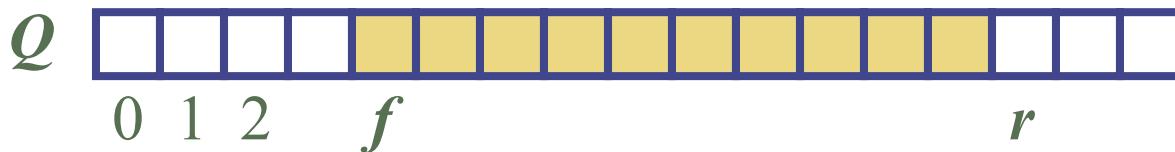
r

f

Queue Operations (cont.)

- ❑ Operation enqueue throws an exception if the array is full
- ❑ This exception is implementation-dependent

```
Algorithm enqueue(o)
    if size() =  $N - 1$  then
        throw
            IllegalStateException
    else
         $r \leftarrow (f + sz) \bmod N$ 
         $Q[r] \leftarrow o$ 
         $sz \leftarrow (sz + 1)$ 
```



Queue Operations (cont.)

- ❑ Note that operation dequeue returns null if the queue is empty

Algorithm *dequeue()*

if *isEmpty()* **then**

return *null*

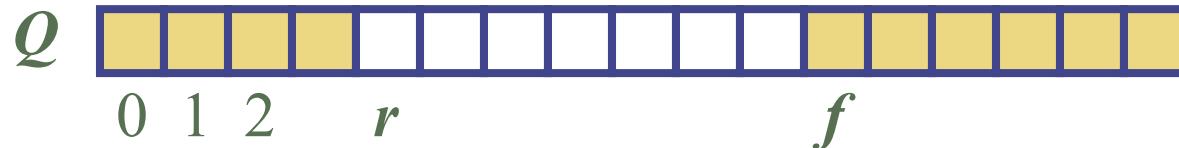
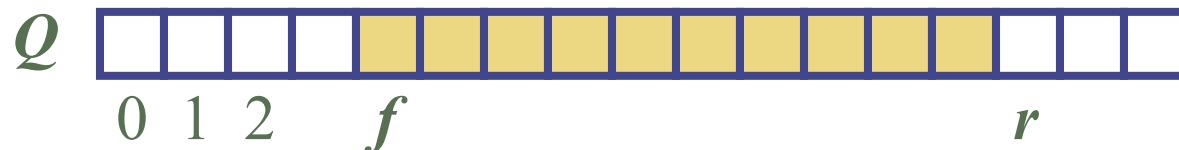
else

$$o \leftarrow Q[f]$$

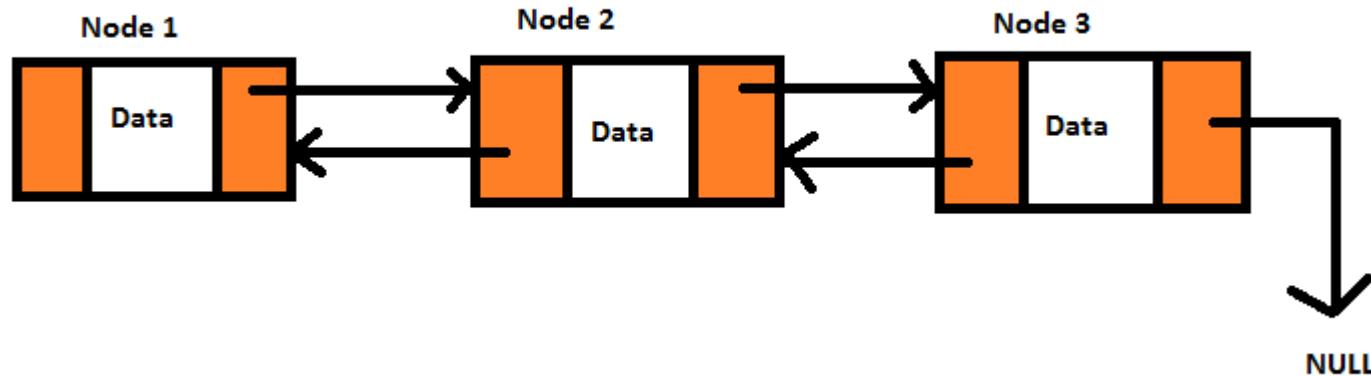
$$f \leftarrow (f + 1) \bmod N$$

$$sz \leftarrow (sz - 1)$$

return *o*



Linked Lists



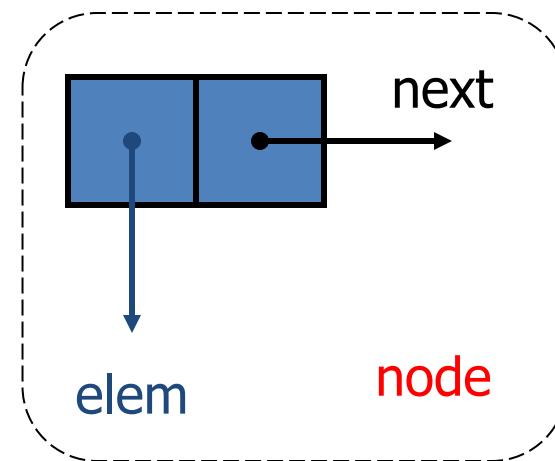
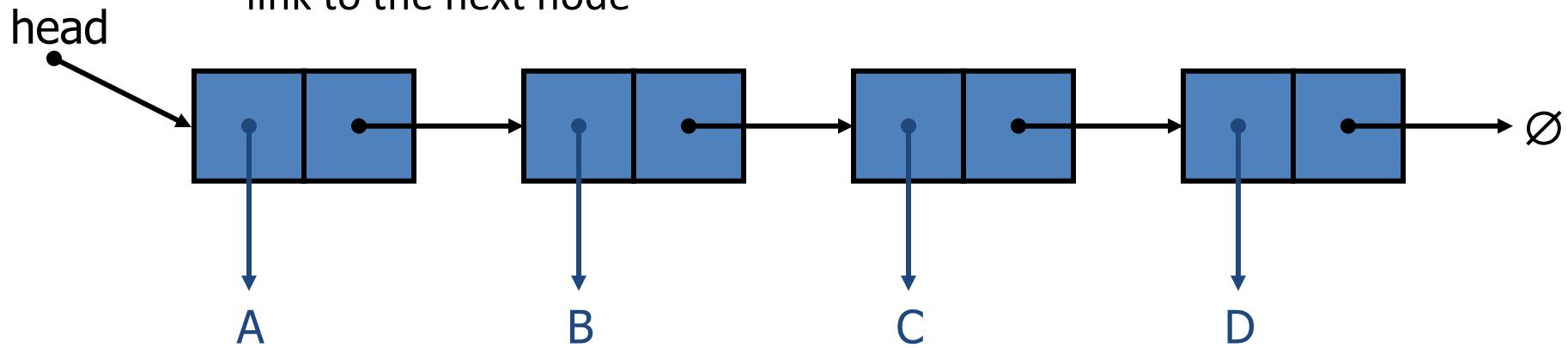
Slides partly from Data Structures and Algorithms in Java, by
M. T. Goodrich, et. Al.

Dynamic vs. Static Data Structure

- ❑ Array-based data structures have problem of fixed size.
- ❑ What if you don't know the size of the data in advance?
- ❑ **Static data** (e.g. variables, arrays, function calls) are stored in the **stack memory** while **dynamic data** is created in the **heap memory**.

Singly Linked List

- A singly linked list is a concrete data structure consisting of a sequence of nodes, starting from a head pointer
- Each node stores
 - element
 - link to the next node



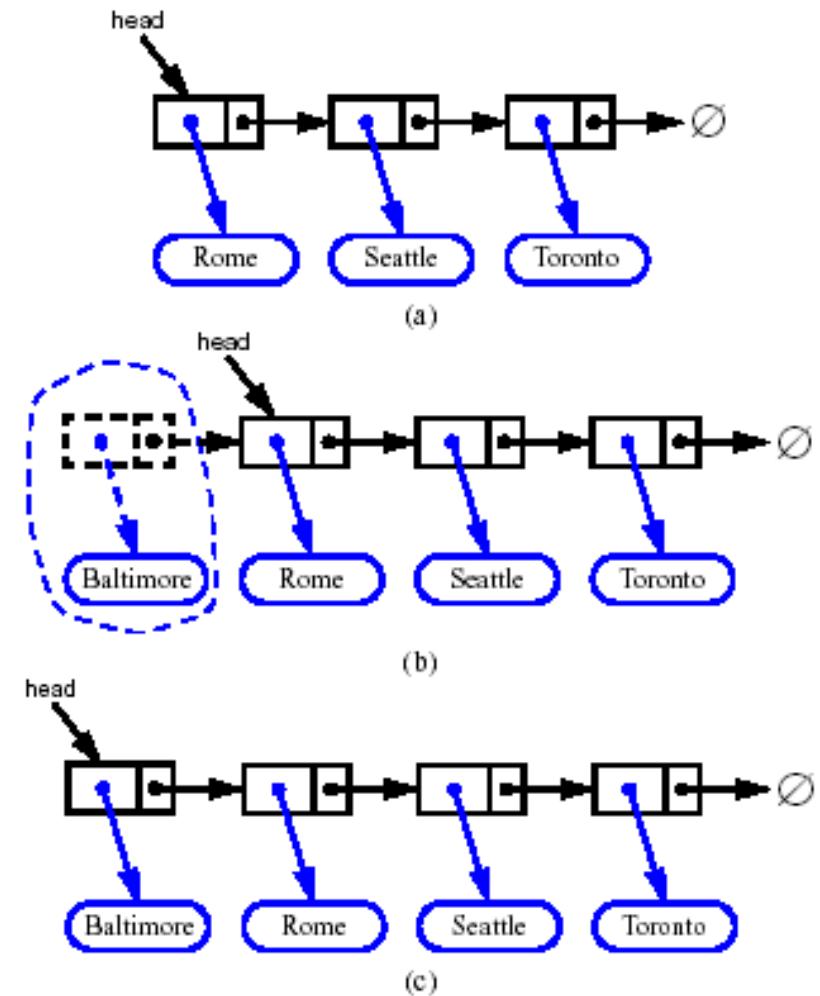
Node and LinkedList Classes

```
class Node{  
    public int data;  
    public Node nextLink;  
  
    // constructor  
    public Node(int d) {  
        data = d;  
    }  
}
```

```
class LinkedList {  
    private Node first;  
    // constructor  
    public LinkedList() {  
        first = null;  
    }  
    public boolean isEmpty() {  
        return first == null;  
    }  
}
```

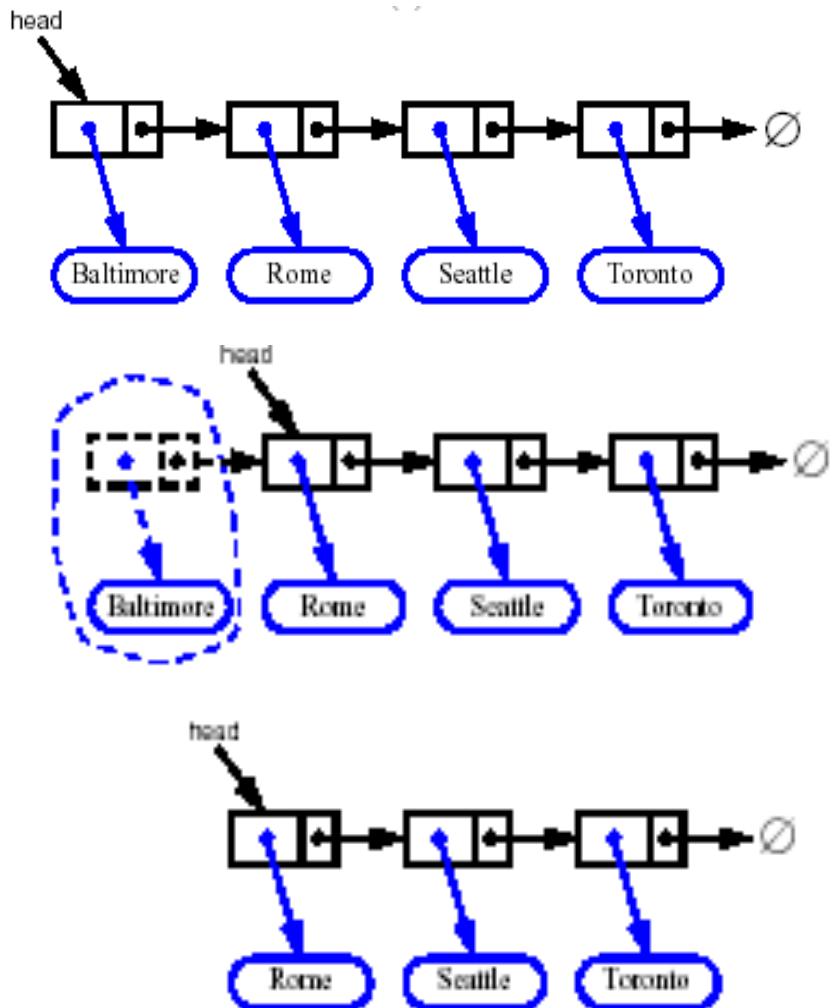
Inserting at the Head

1. Allocate a new node
2. Insert new element
3. Have new node point to old head
4. Update head to point to new node



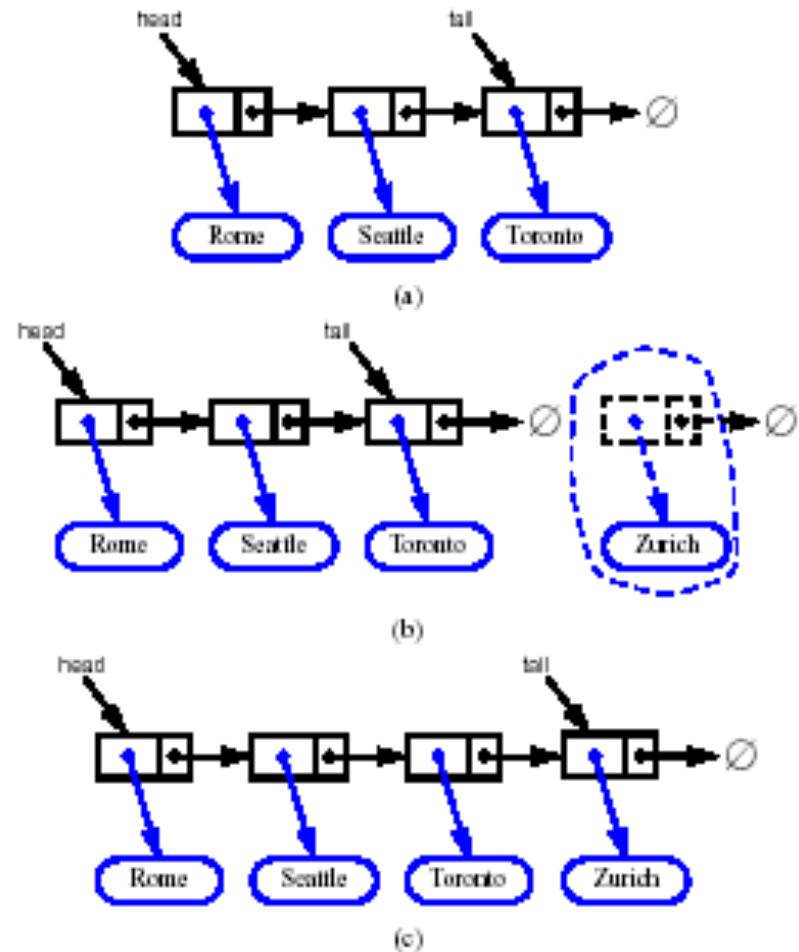
Removing at the Head

1. Update head to point to next node in the list
2. Allow garbage collector to reclaim the former first node



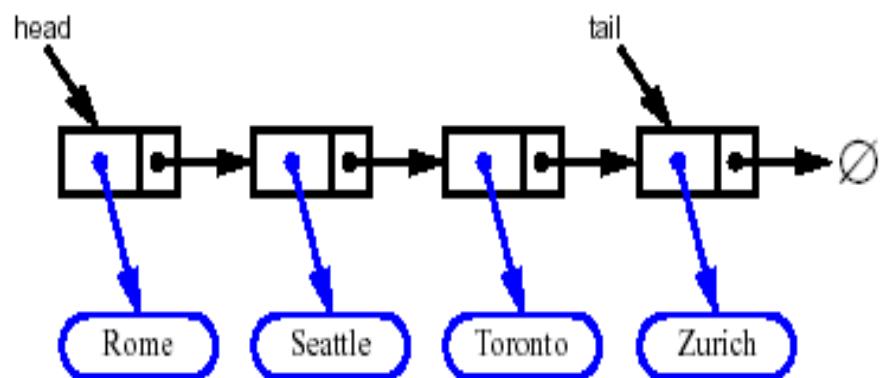
Inserting at the Tail

1. Allocate a new node
2. Insert new element
3. Have new node point to null
4. Have old last node point to new node
5. Update tail to point to new node



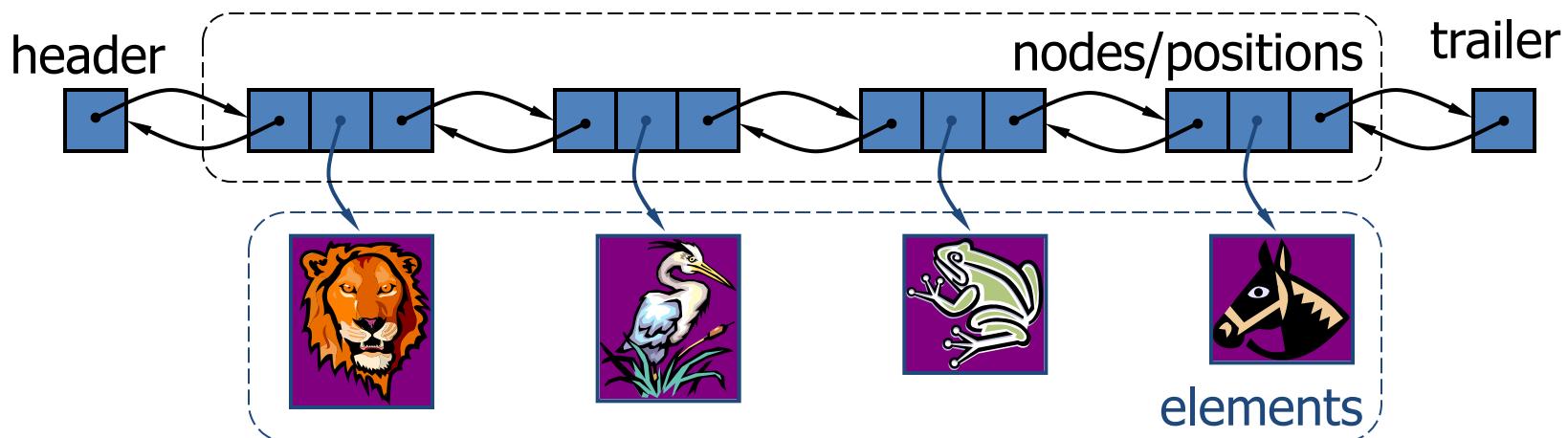
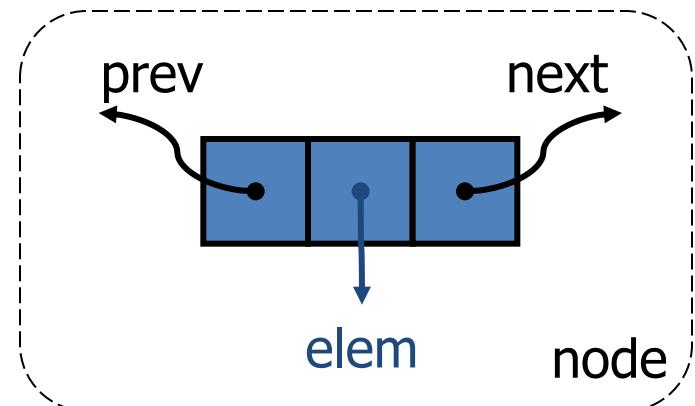
Removing at the Tail

- Removing at the tail of a singly linked list is not efficient!
- There is no constant-time way to update the tail to point to the previous node



Doubly Linked List

- A doubly linked list provides a natural implementation of the Node List ADT
- Nodes implement Position and store:
 - element
 - link to the previous node
 - link to the next node
- Special trailer and header nodes

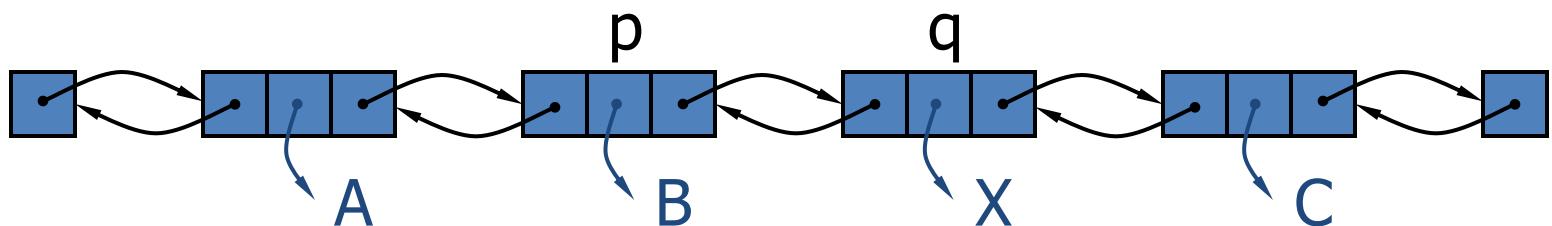
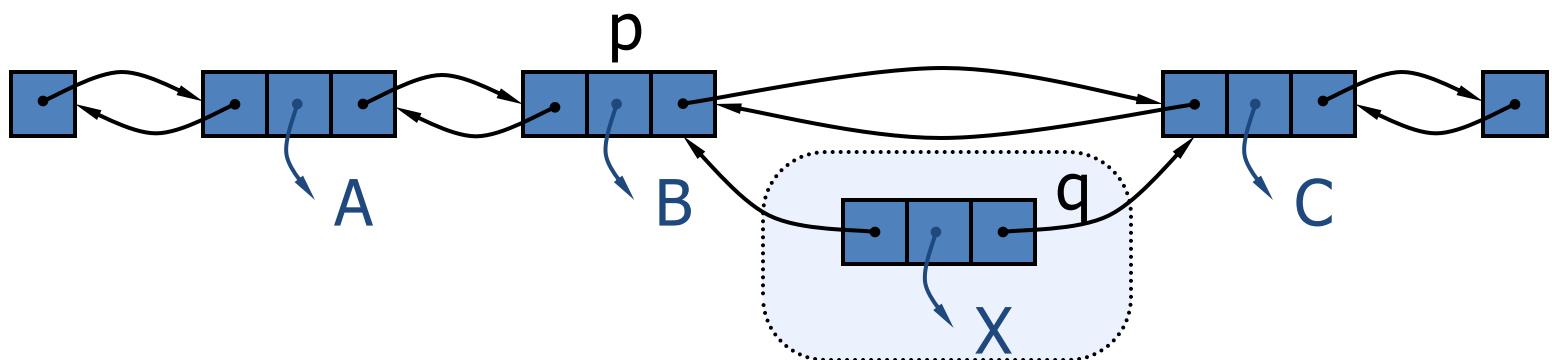
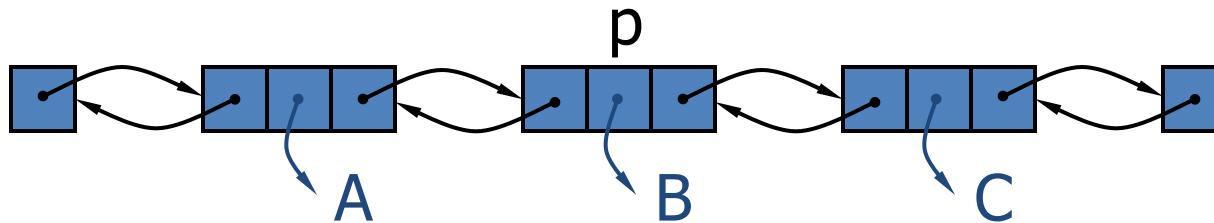


Better than singly linked list

- we can efficiently insert a node at either end of a singly linked list, and can delete a node at the head of a list,
 - but we are unable to efficiently delete a node at the tail of the list.
- More generally, we cannot efficiently delete an arbitrary node from an interior position of the list,
 - because we cannot determine the node that immediately *precedes* the node to be deleted (yet, that node needs to have its next reference updated).

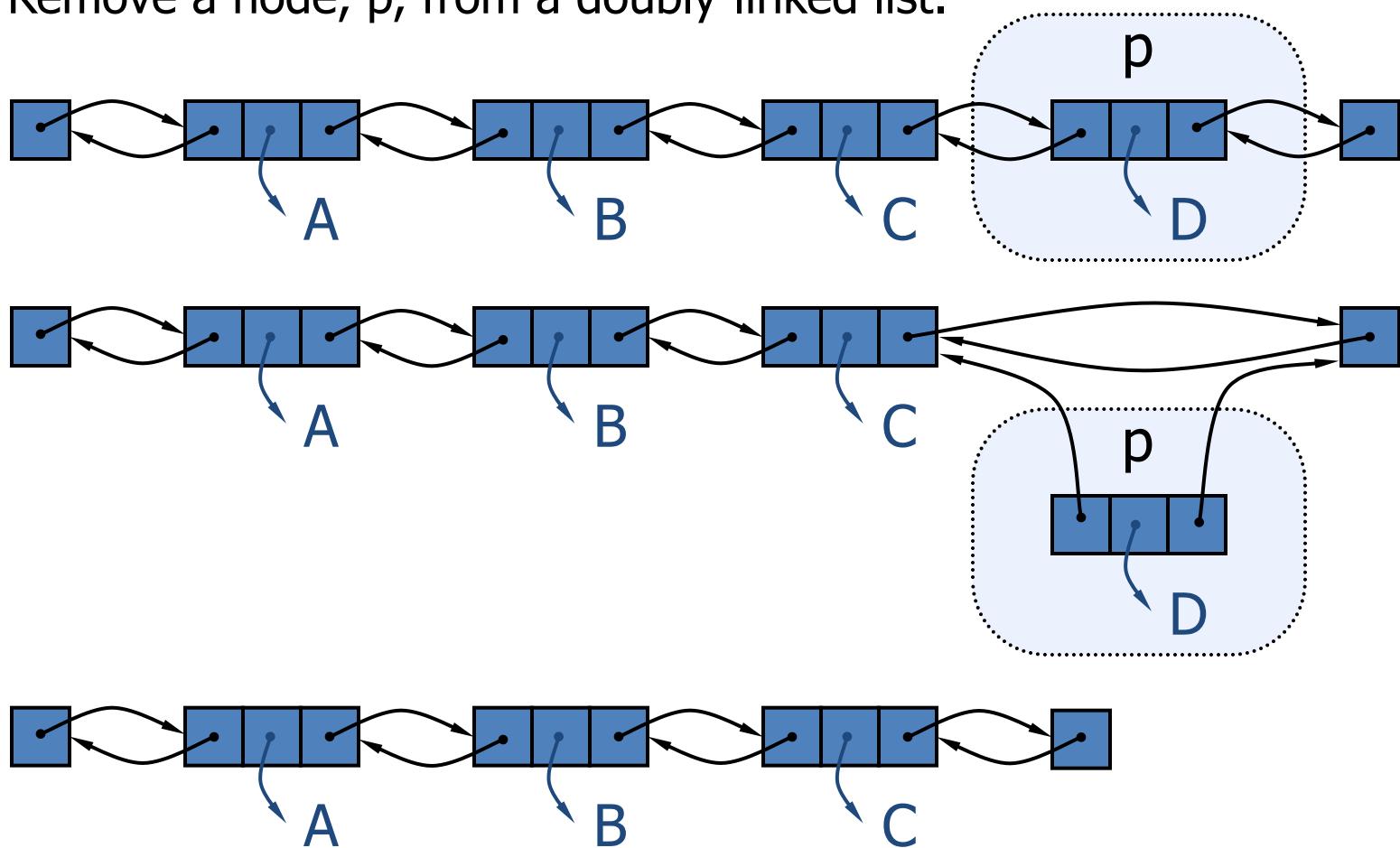
Insertion

- Insert a new node, q , between p and its successor.



Deletion

- Remove a node, p , from a doubly-linked list.



Performance

- In a doubly linked list
 - The space used by a list with n elements is $O(n)$
 - The space used by each position of the list is $O(1)$
 - All the standard operations of a list run in $O(1)$ time

The Positional List ADT

- What if a waiting customer decides to hang up before reaching the front of the customer service queue? Or what if someone who is waiting in line to buy tickets allows a friend to “cut” into line at that position?
- indices are not a good abstraction for describing a local position in some applications, because the index of an entry changes over time due to insertions or deletions that happen earlier in the sequence.

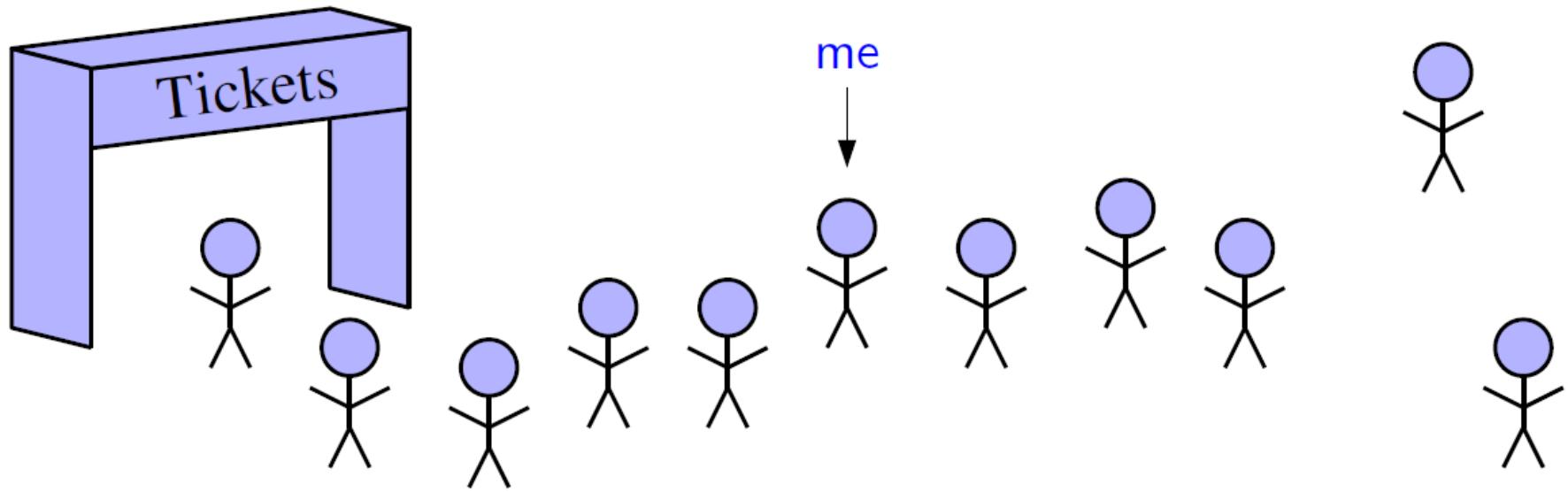


Figure 7.14: We wish to be able to identify the position of an element in a sequence without the use of an integer index.

As another example

- a text document can be viewed as a long sequence of characters.
- A word processor uses the abstraction of a **cursor** to describe a position within the document without explicit use of an integer index,
 - allowing operations such as “delete the character at the cursor” or “insert a new character just after the cursor.”
- Furthermore, we may be able to refer to an inherent position within a document,
 - such as the beginning of a particular section, without relying on a character index (or even a section number) that may change as the document evolves.

Positional List

- To provide for a general abstraction of a sequence of elements with the ability to identify the location of an element, we define a **positional list** ADT.
- A position p is unaffected by changes elsewhere in a list; the only way in which a position becomes invalid is if an explicit command is issued to delete it.
- A position instance is a simple object, supporting only the following method:
 - $p.\text{element}()$: Return the element stored at position p .

Positional List ADT

❑ Accessor methods:

`first()`: Returns the position of the first element of L (or null if empty).

`last()`: Returns the position of the last element of L (or null if empty).

`before(p)`: Returns the position of L immediately before position p (or null if p is the first position).

`after(p)`: Returns the position of L immediately after position p (or null if p is the last position).

`isEmpty()`: Returns true if list L does not contain any elements.

`size()`: Returns the number of elements in list L .

Positional List ADT, 2

□ Update methods:

`addFirst(e)`: Inserts a new element *e* at the front of the list, returning the position of the new element.

`addLast(e)`: Inserts a new element *e* at the back of the list, returning the position of the new element.

`addBefore(p, e)`: Inserts a new element *e* in the list, just before position *p*, returning the position of the new element.

`addAfter(p, e)`: Inserts a new element *e* in the list, just after position *p*, returning the position of the new element.

`set(p, e)`: Replaces the element at position *p* with element *e*, returning the element formerly at position *p*.

`remove(p)`: Removes and returns the element at position *p* in the list, invalidating the position.

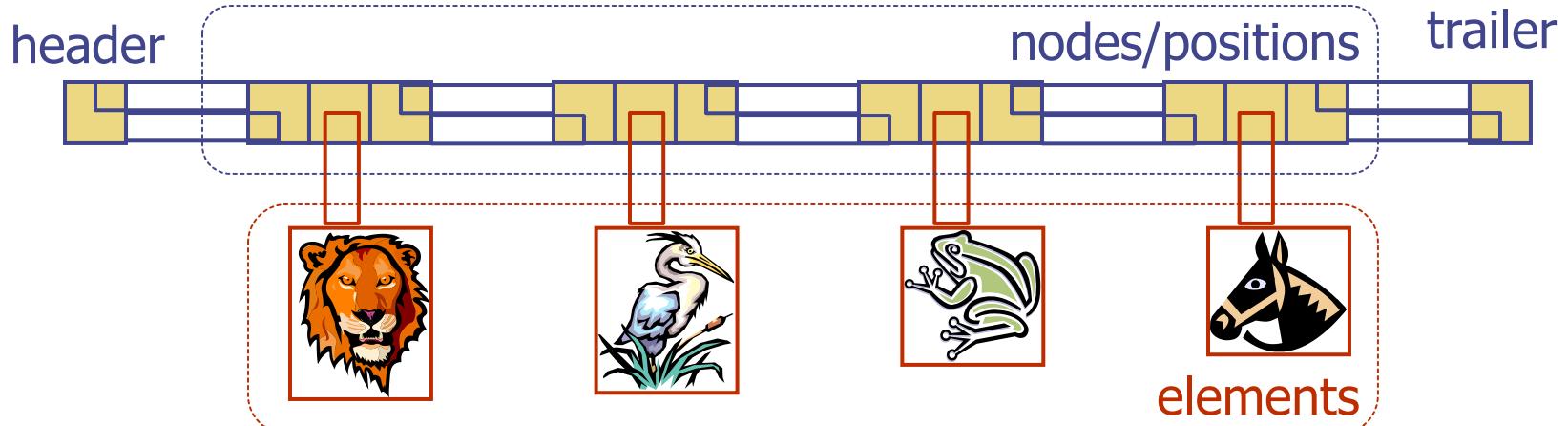
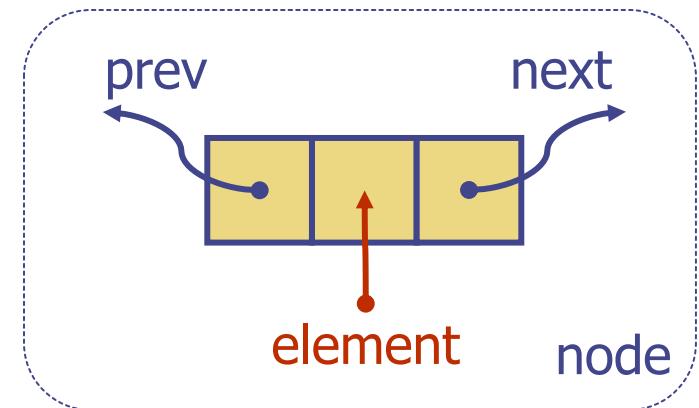
Example

□ A sequence of Positional List operations:

Method	Return Value	List Contents
addLast(8)	p	(8 p)
first()	p	(8 p)
addAfter(p , 5)	q	(8 p , 5 q)
before(q)	p	(8 p , 5 q)
addBefore(q , 3)	r	(8 p , 3 r , 5 q)
r .getElement()	3	(8 p , 3 r , 5 q)
after(p)	r	(8 p , 3 r , 5 q)
before(p)	null	(8 p , 3 r , 5 q)
addFirst(9)	s	(9 s , 8 p , 3 r , 5 q)
remove(last())	5	(9 s , 8 p , 3 r)
set(p , 7)	8	(9 s , 7 p , 3 r)
remove(q)	“error”	(9 s , 7 p , 3 r)

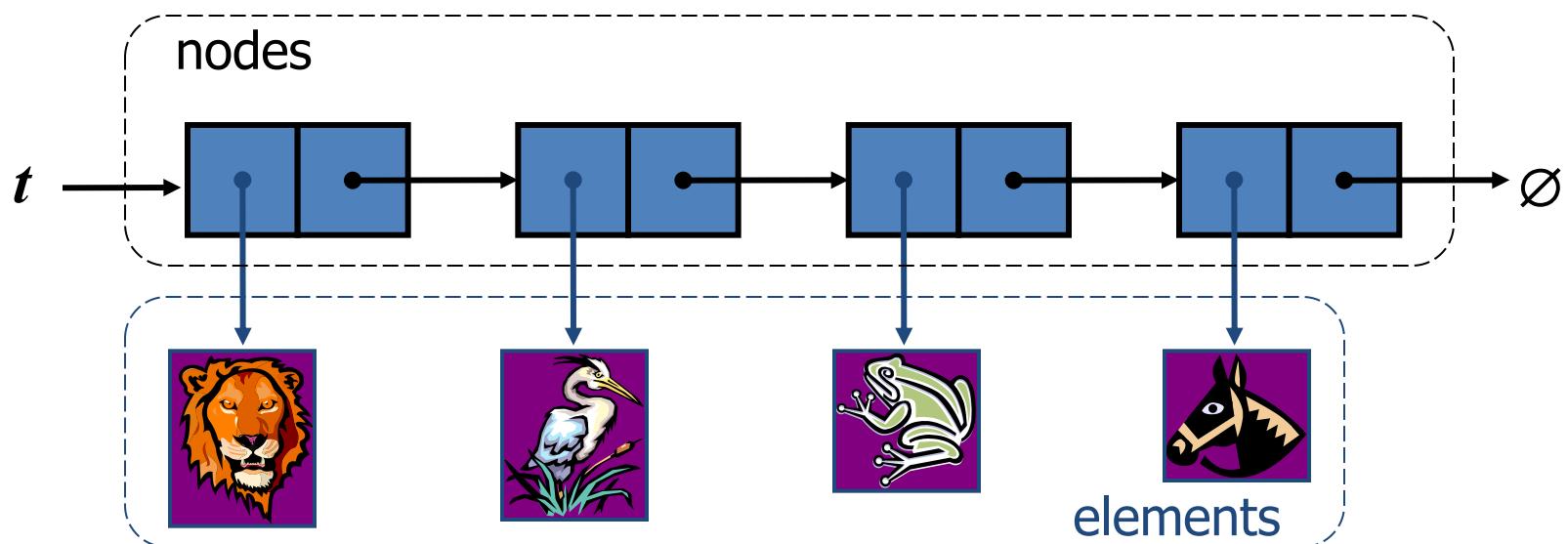
Positional List Implementation

- The most natural way to implement a positional list is with a doubly-linked list.



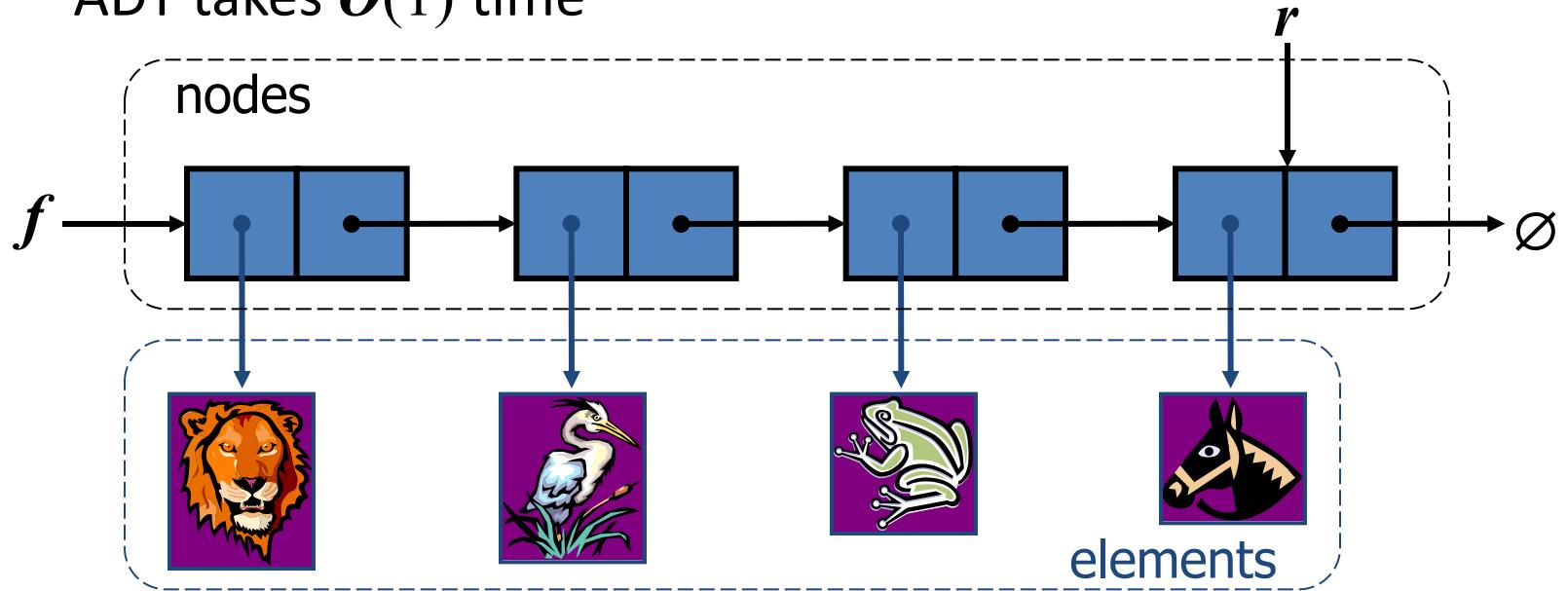
Stack as a Linked List

- We can implement a stack with a singly linked list
- The top element is stored at the first node of the list
- The space used is $O(n)$ and each operation of the Stack ADT takes $O(1)$ time

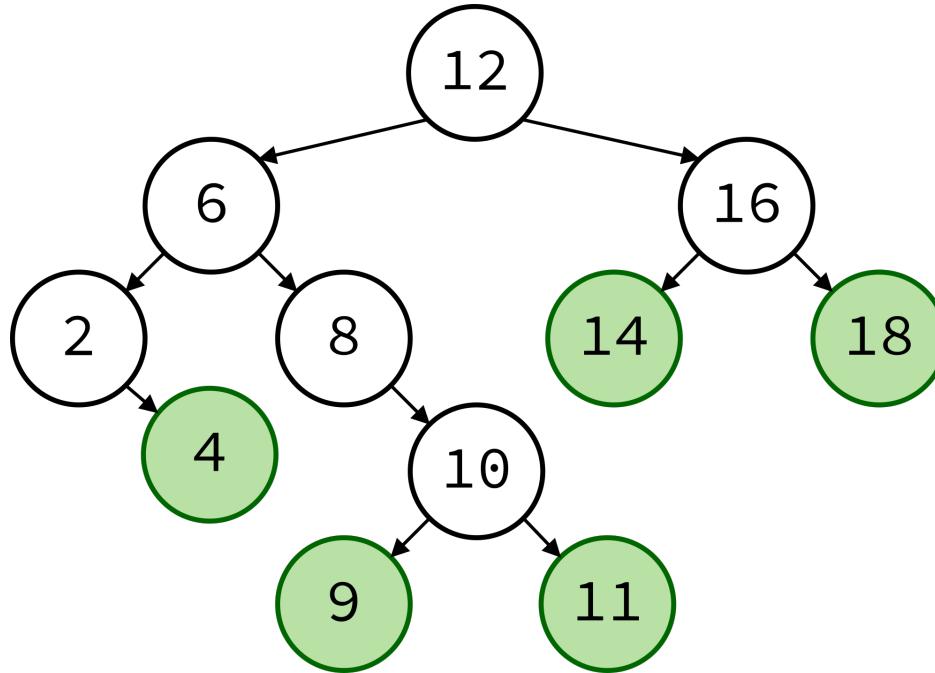


Queue as a Linked List

- We can implement a queue with a singly 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 Queue ADT takes $O(1)$ time



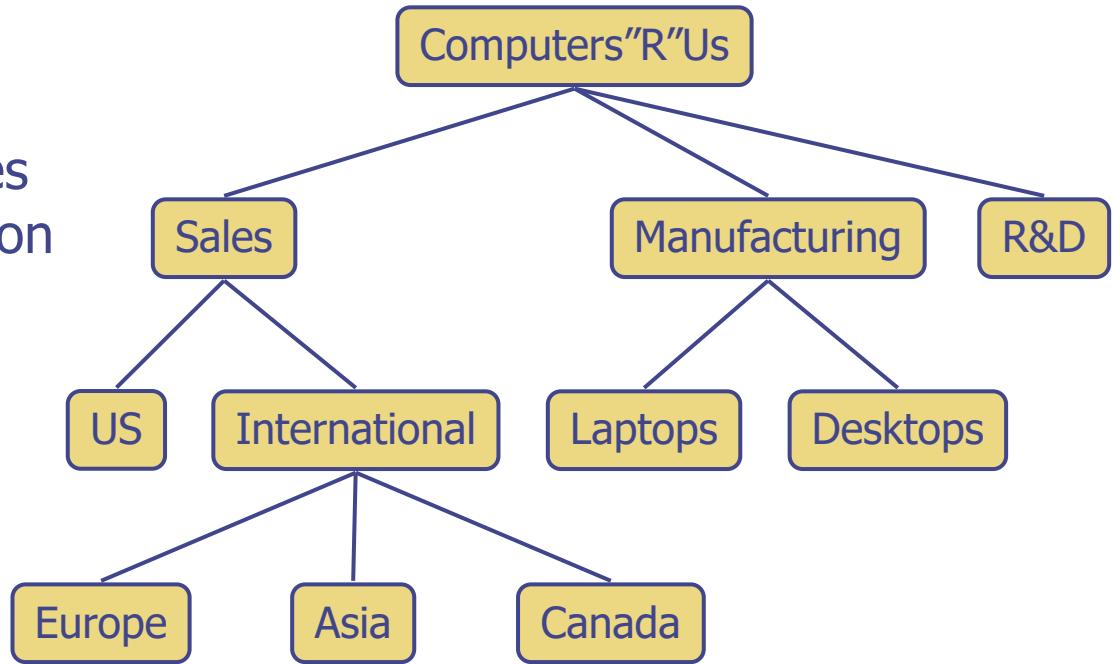
Tree Data Structure



Slides partly from Data Structures and Algorithms in Java
(Goodrich, et. al.), Data Structures and Algorithm Analysis in Java
(M. A. Weiss), and J. Edgar course at SFU

What is a Tree?

- An abstract model of a hierarchical structure
- A tree consists of nodes with a parent-child relation
- Applications:
 - Organization charts
 - File systems
 - Programming environments
 - Many more...



Definitions

- A **tree** is a non-empty collection of **vertices** & **edges**
- A **vertex** (node) can have a name and carry other associated information
- A **path** is list of distinct vertices in which successive vertices are connected by edges.
- Any two vertices must have one and only one path between them else its not a tree.
- A tree with N nodes has N-1 edges

Definitions

- **root**: starting point (top) of the tree
- **parent** (ancestor): the vertex above this vertex
- **child** (descendent): the vertices below this vertex
- **leaves** (terminal nodes): have no children
- **level**: the number of edges between this node and the root
- **Ordered tree**: where children's order is significant

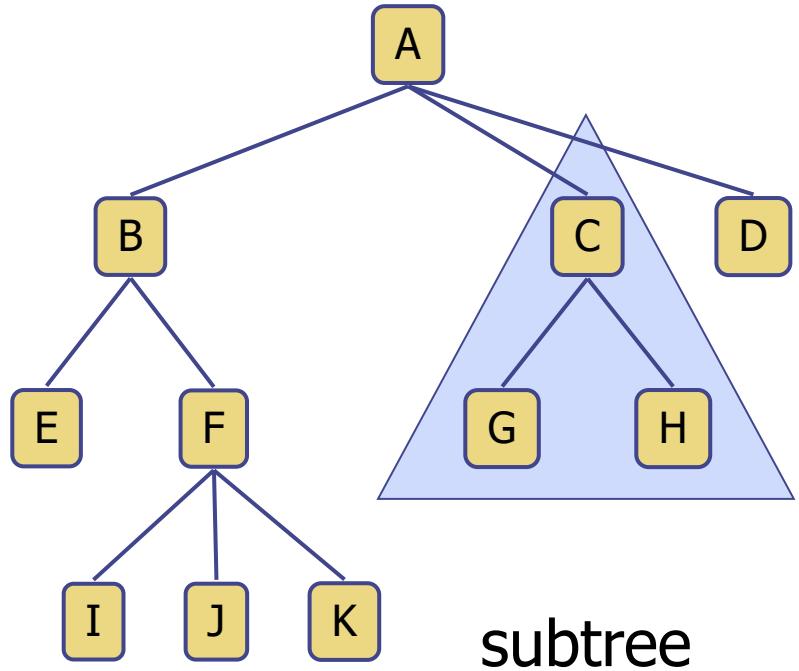
Definitions

- **Depth of a node:** the length of the path from the root to that node (root: depth 0)
- **Height of a node:** the length of the longest path from that node to a leaf (any leaf: height 0)
- **Height of a tree:** The length of the longest path from the root to a leaf
- **Balanced Tree:** the difference between the height of the left sub-tree and the height of the right sub-tree is not more than 1.

Definitions

- **Subtree**: tree consisting of a node and its descendants
- **Internal node**: node with at least one child
- **External node** (a.k.a. leaf): node without children
- **Ancestors** of a node: parent, grandparent, grand-grandparent, etc.
- **Descendant** of a node: child, grandchild, grand-grandchild, etc.

- ❑ Root: (A)
- ❑ Internal node: (A, B, C, F)
- ❑ External node (a.k.a. leaf): (E, I, J, K, G, H, D)
- ❑ Height of a tree: (3)



Trees - Example

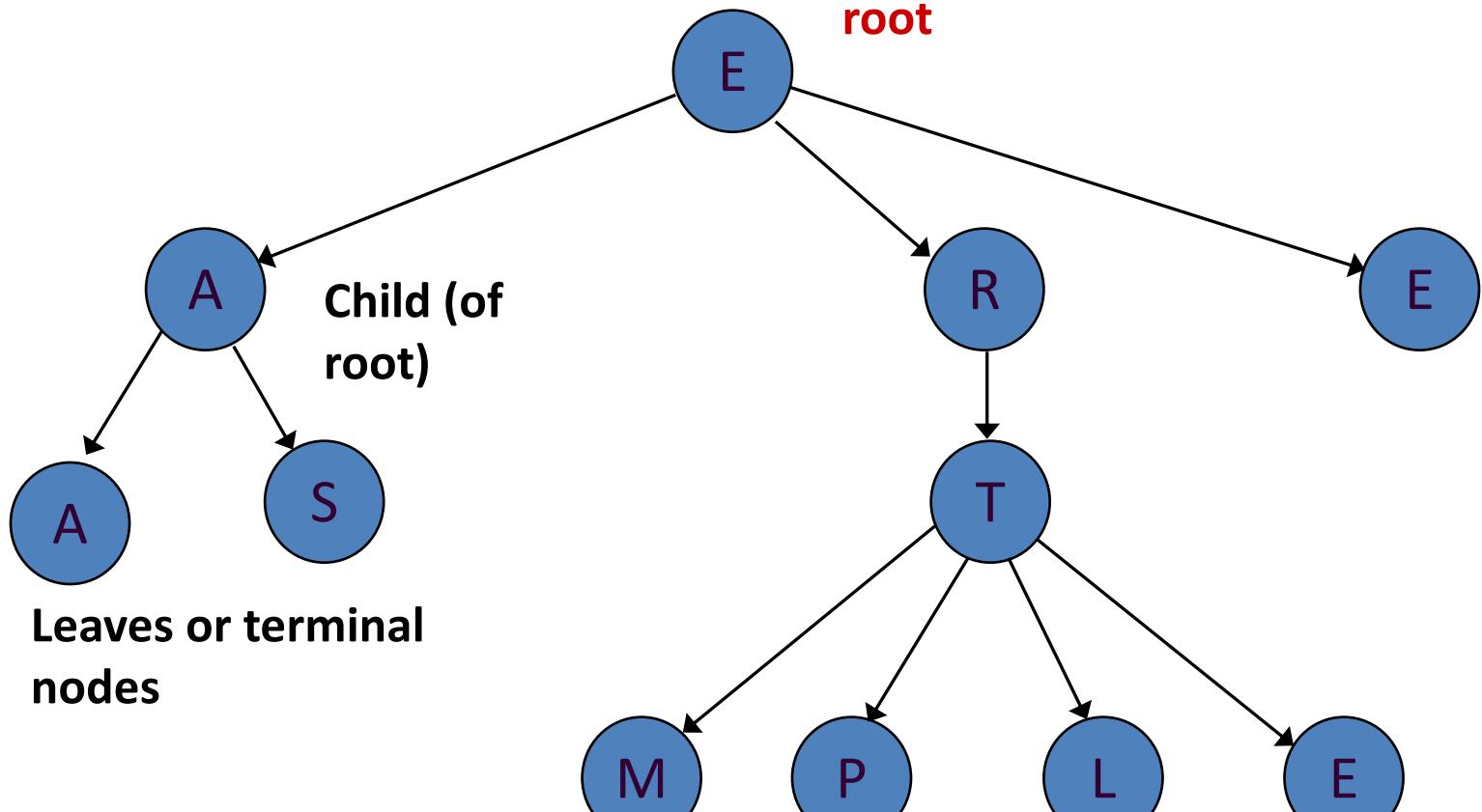
Level

0

1

2

3



Depth of T: 2

Height of T: 1

More Tree Terminology

- A **leaf** is a node with no children
- A **path** is a sequence of nodes $v_1 \dots v_n$
 - where v_i is a parent of v_{i+1} ($1 \leq i \leq n-1$)
- A **subtree** is any node in the tree along with all of its descendants
- A **binary tree** is a tree with at most two children per node
 - The children are referred to as **left** and **right**
 - We can also refer to left and right subtrees

Tree ADT

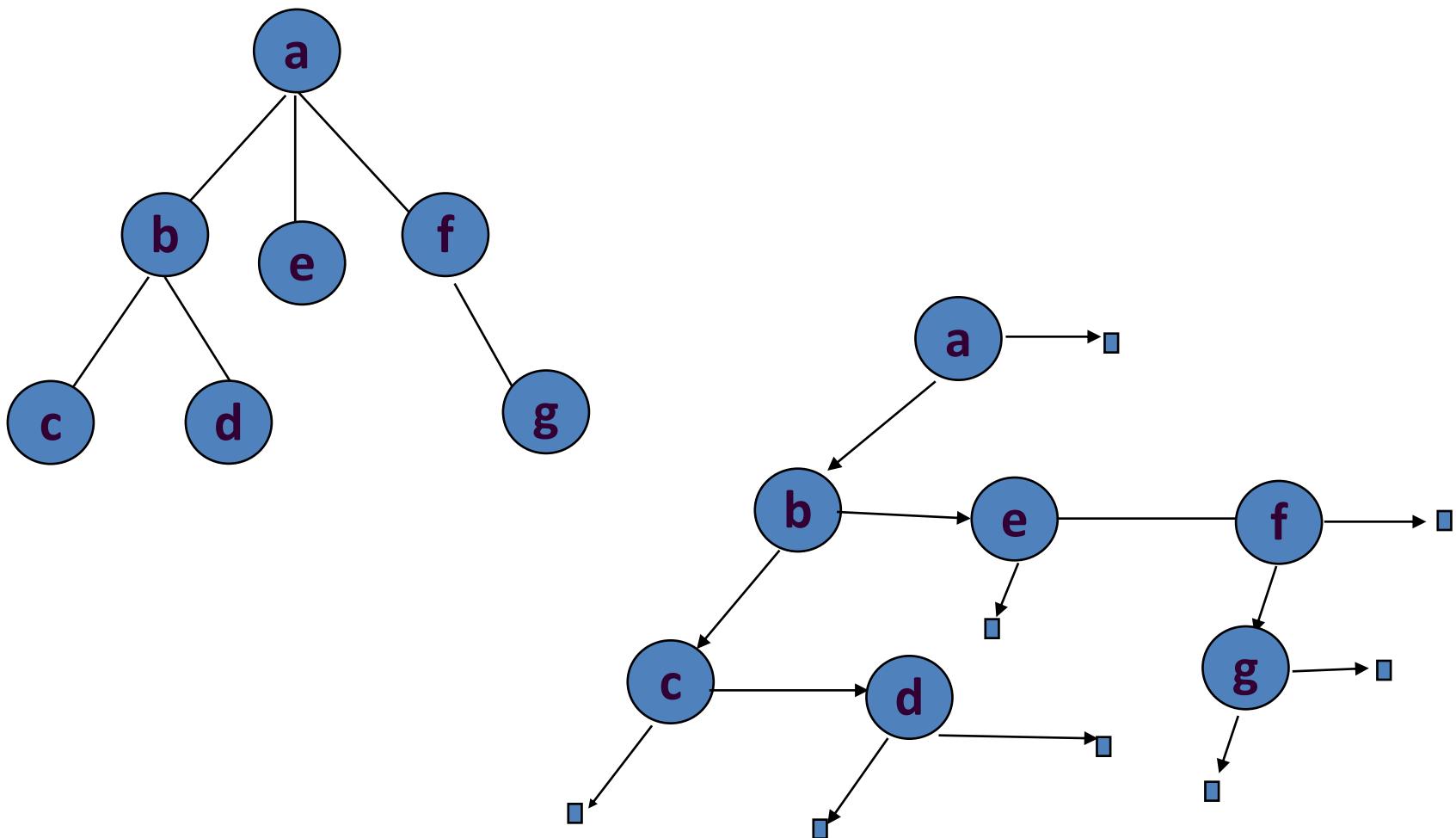
- Generic methods:
 - integer **size()**
 - boolean **isEmpty()**
 - Iterator **iterator()**
- Accessor methods:
 - position **root()**
 - position **parent(p)**
 - Iterable **children(p)**
 - Integer **numChildren(p)**

- ◆ Query methods:
 - boolean **isInternal(p)**
 - boolean **isExternal(p)**
 - boolean **isRoot(p)**
- ◆ Additional update methods may be defined by data structures implementing the Tree ADT

Tree Representation

```
Class TreeNode {  
    Object      element;  
    TreeNode   firstChild;  
    TreeNode   nextSibling;  
}
```

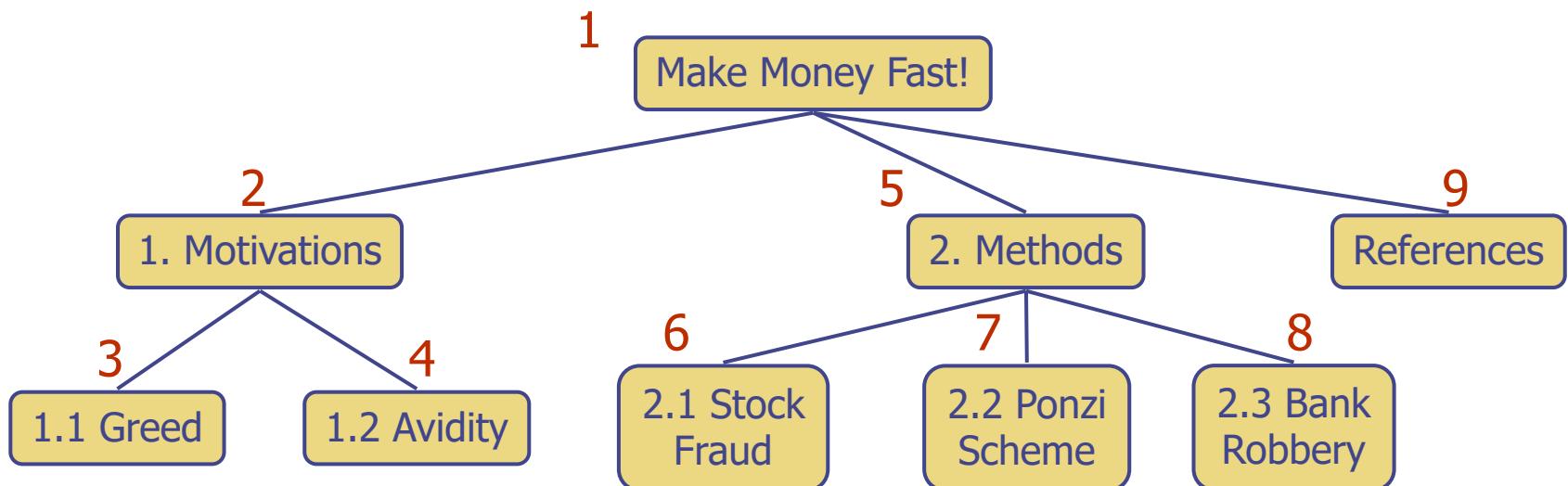
Example



Preorder Traversal

- ❑ A traversal visits the nodes of a tree in a systematic manner
- ❑ In a preorder traversal, a node is visited before its descendants
- ❑ Application: print a structured document

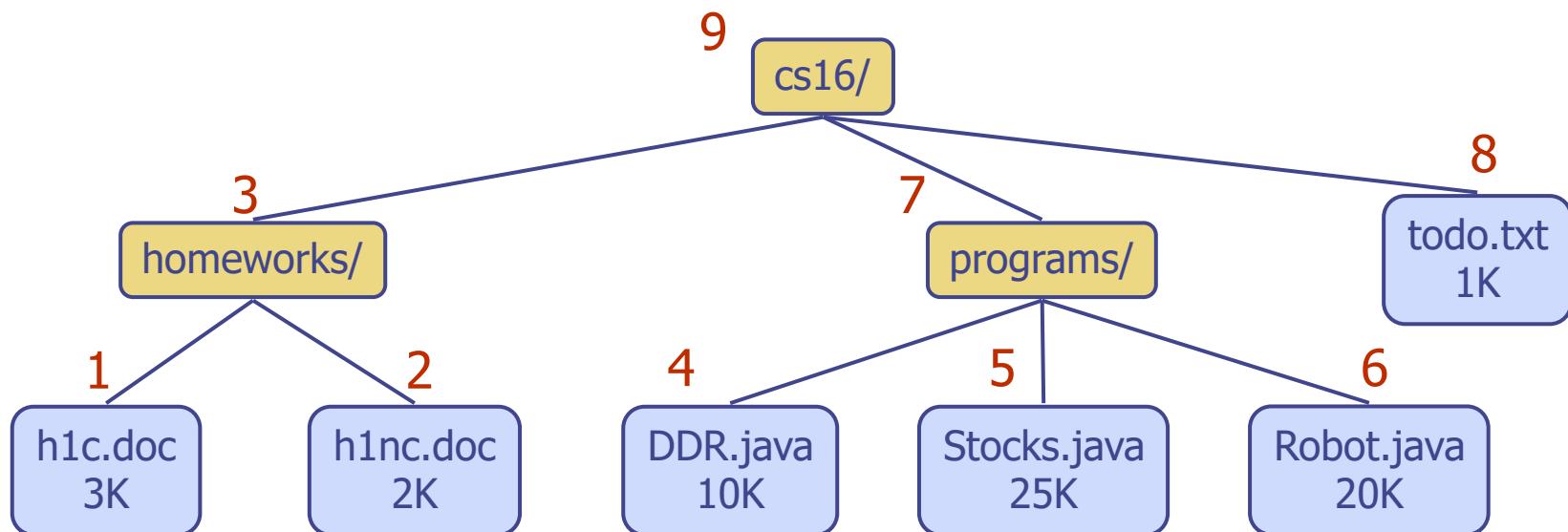
```
Algorithm preOrder(v)
visit(v)
for each child w of v
    preorder (w)
```



Postorder Traversal

- In a postorder traversal, a node is visited after its descendants
- Application: compute space used by files in a directory and its subdirectories

```
Algorithm postOrder(v)
for each child w of v
    postOrder (w)
    visit(v)
```



DFS Traversals

DFS Preorder

Node → Left → Right

DFS Inorder

Left → Node → Right

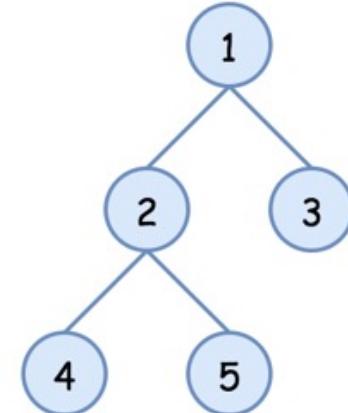
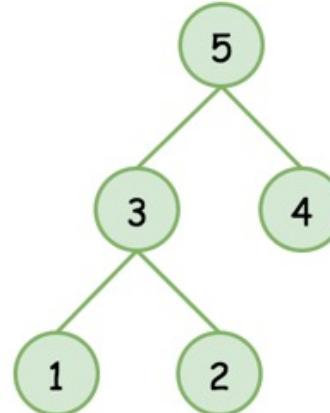
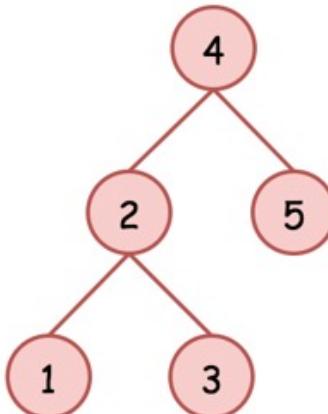
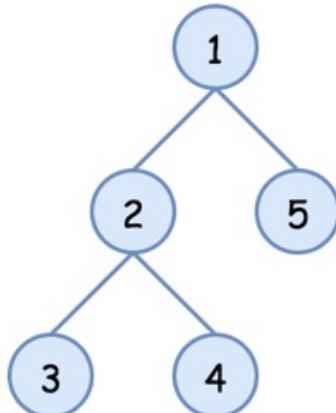
DFS Postorder

Left → Right → Node

BFS

Node → Left → Right

Traversal = [1, 2, 3, 4, 5]



[root.val] +
preorder(root.left) +
preorder(root.right)
if root else []

inorder(root.left) +
[root.val] +
inorder(root.right)
if root else []

postorder(root.left) +
postorder(root.right) +
[root.val]
if root else []

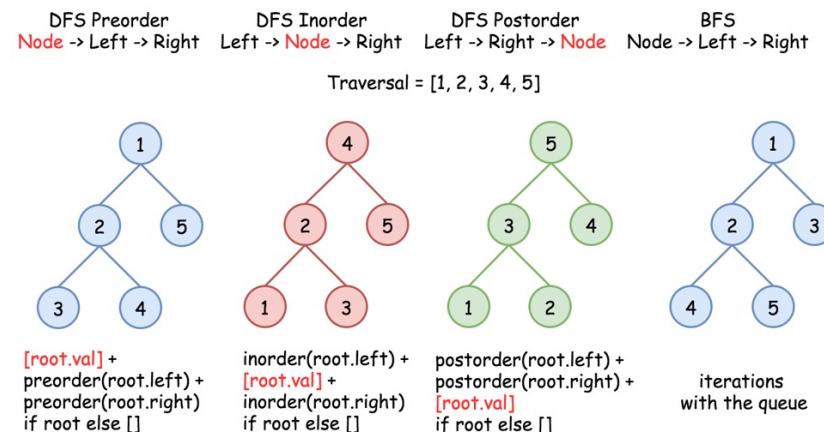
iterations
with the queue

<https://leetcode.com/problems/binary-tree-inorder-traversal/solutions/328601/all-dfs-traversals-in-java-in-5-lines-recursion/>

DFS Traversals (Preorder)

Preorder: Node -> Left -> Right

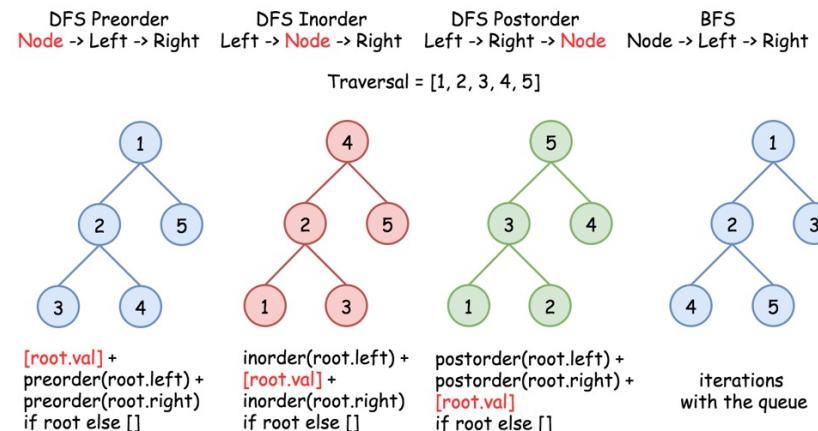
```
public void preorder(TreeNode root,  
List<Integer> nums) {  
    if (root == null) return;  
    nums.add(root.val);  
    preorder(root.left, nums);  
    preorder(root.right, nums);  
}
```



DFS Traversals (Inorder)

Inorder : Left -> Node -> Right

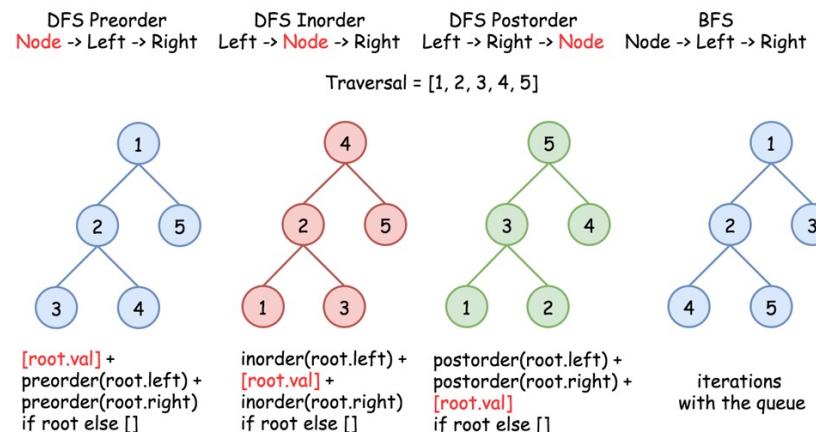
```
public void inorder(TreeNode root,  
List<Integer> nums) {  
    if (root == null) return;  
    inorder(root.left, nums);  
    nums.add(root.val);  
    inorder(root.right, nums);  
}
```



DFS Traversals (Postorder)

Postorder : Left -> Right -> Node

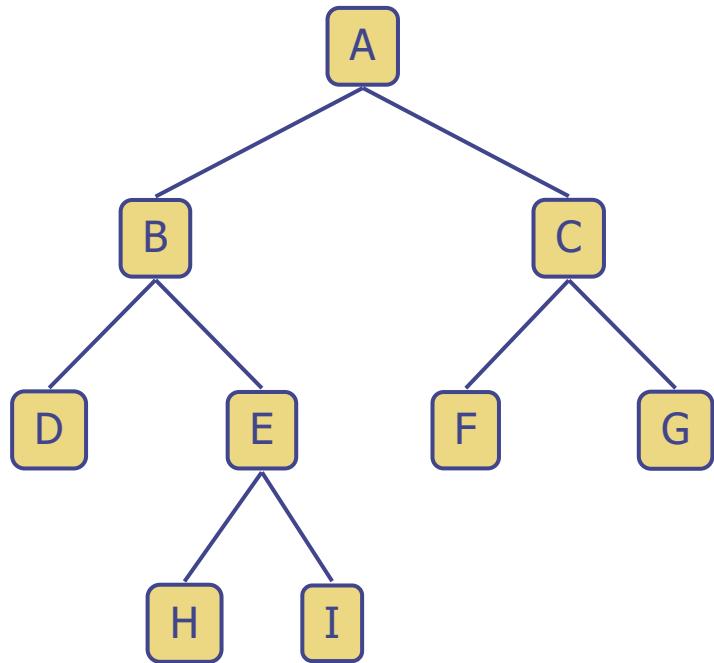
```
public void postorder(TreeNode root,  
List<Integer> nums) {  
    if (root == null) return;  
    postorder(root.left, nums);  
    postorder(root.right, nums);  
    nums.add(root.val);  
}
```



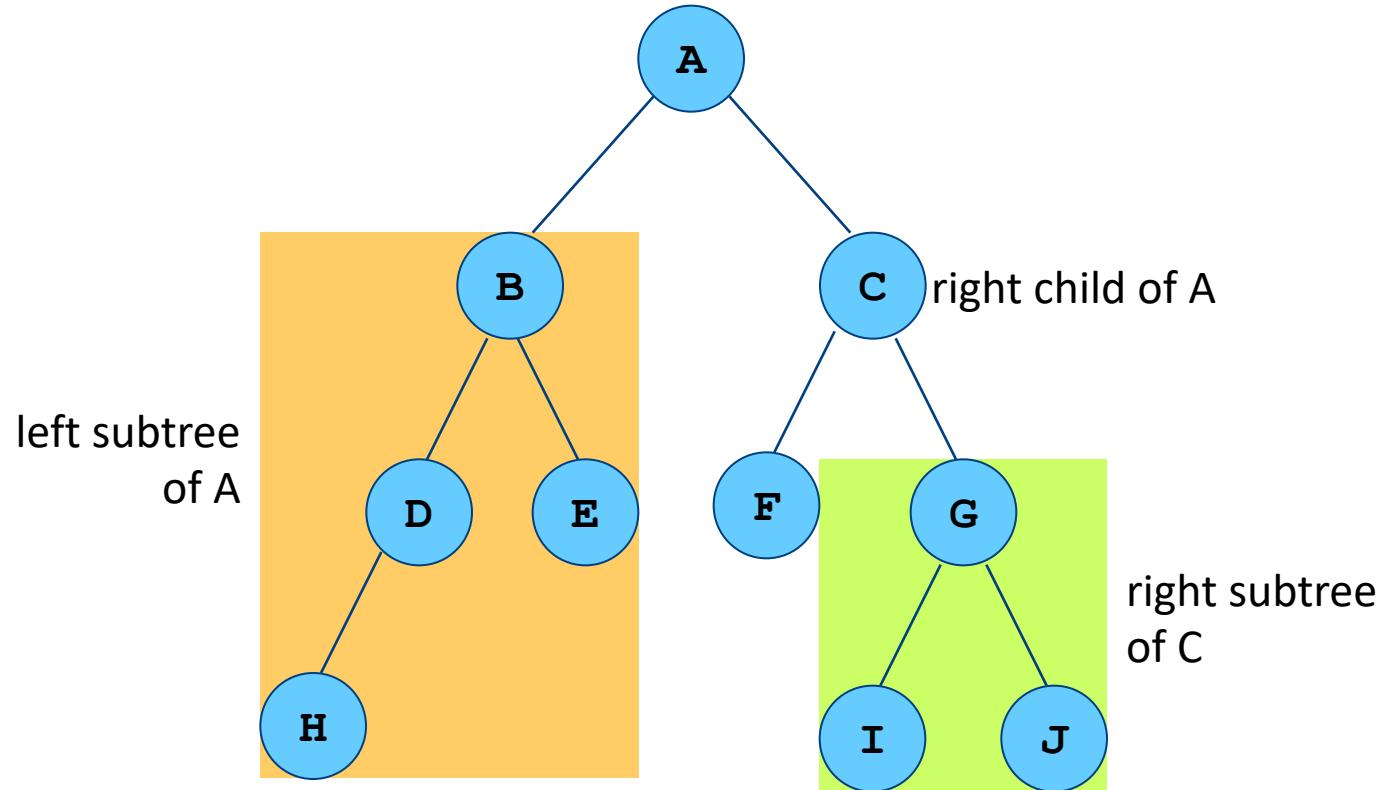
Binary Trees

- A binary tree is a tree with the following properties:
 - Each internal node has at most two children (exactly two for **proper binary trees**)
 - The children of a node are an ordered pair
- We call the children of an internal node **left child** and **right child**
- Alternative recursive definition: a binary tree is either
 - a tree consisting of a single node, or
 - a tree whose root has an ordered pair of children, each of which is a binary tree

- Applications:
 - arithmetic expressions
 - decision processes
 - searching



Binary Tree



Binary Tree Node

Class BinaryNode

{

Object Element; // the data in the node

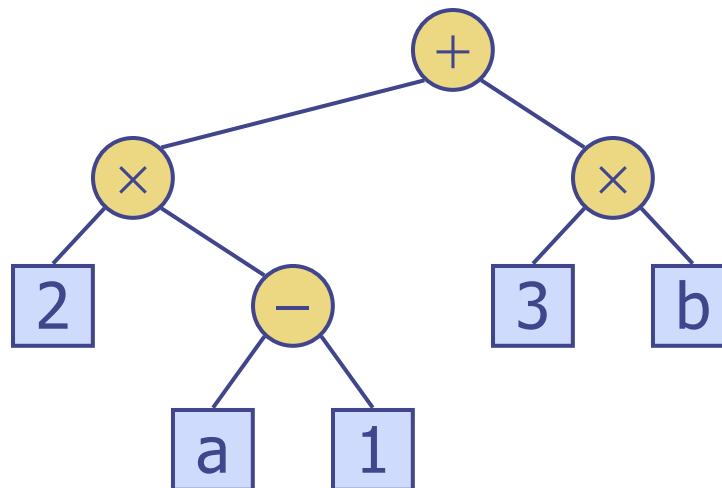
BinaryNode left; // Left child

BinaryNode right; // Right child

}

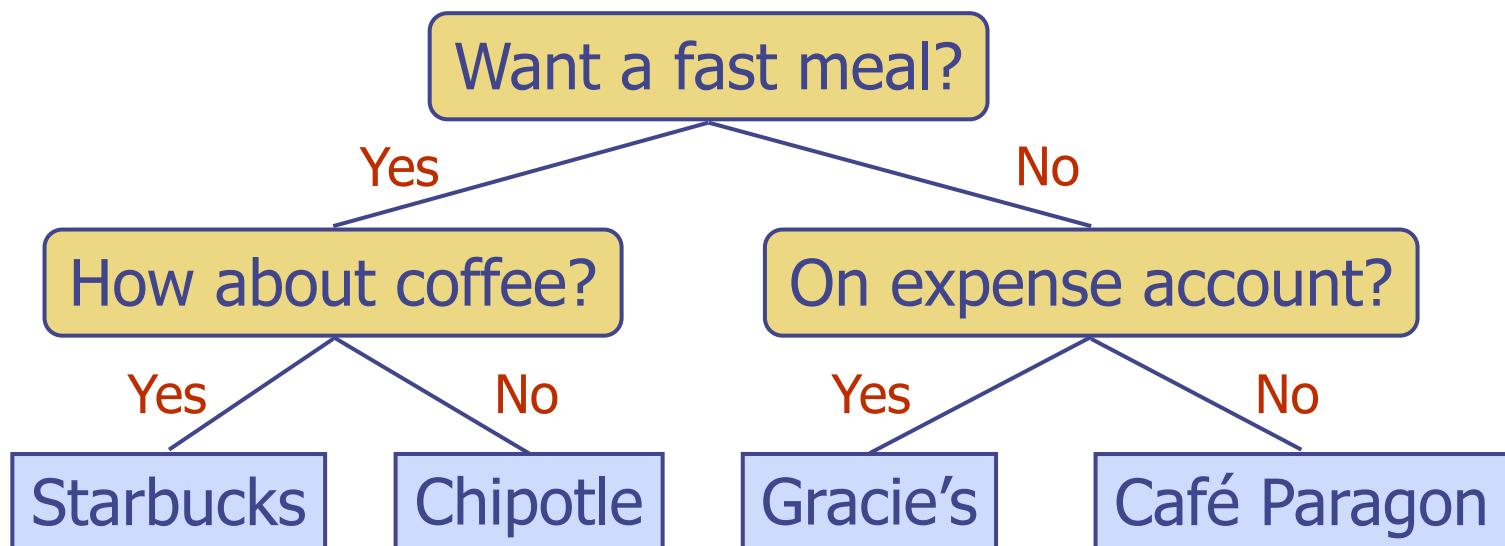
Arithmetic Expression Tree

- ❑ Binary tree associated with an arithmetic expression
 - internal nodes: operators
 - external nodes: operands
- ❑ Example: arithmetic expression tree for the expression $(2 \times (a - 1) + (3 \times b))$



Decision Tree

- ❑ Binary tree associated with a decision process
 - internal nodes: questions with yes/no answer
 - external nodes: decisions
- ❑ Example: dining decision



Properties of Proper Binary Trees

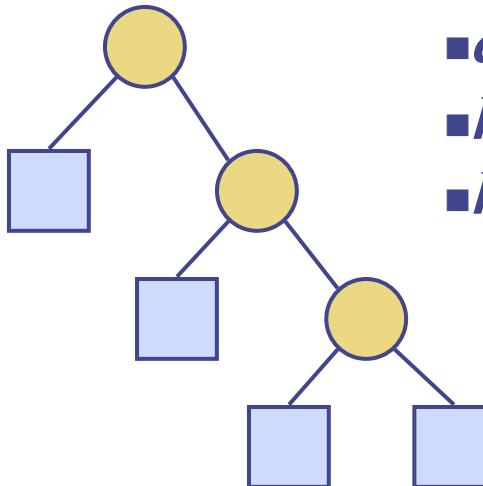
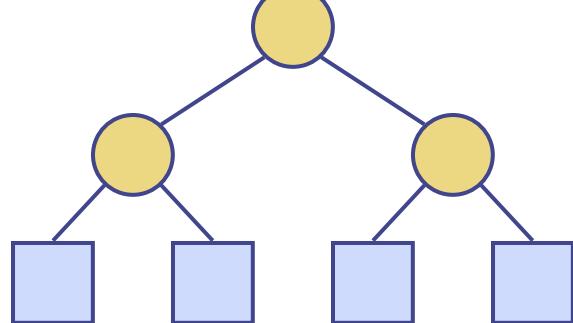
❑ Notation

n number of nodes

e number of external nodes

i number of internal nodes

h height



◆ Properties:

$$e = i + 1$$

$$n = 2e - 1$$

$$h \leq i$$

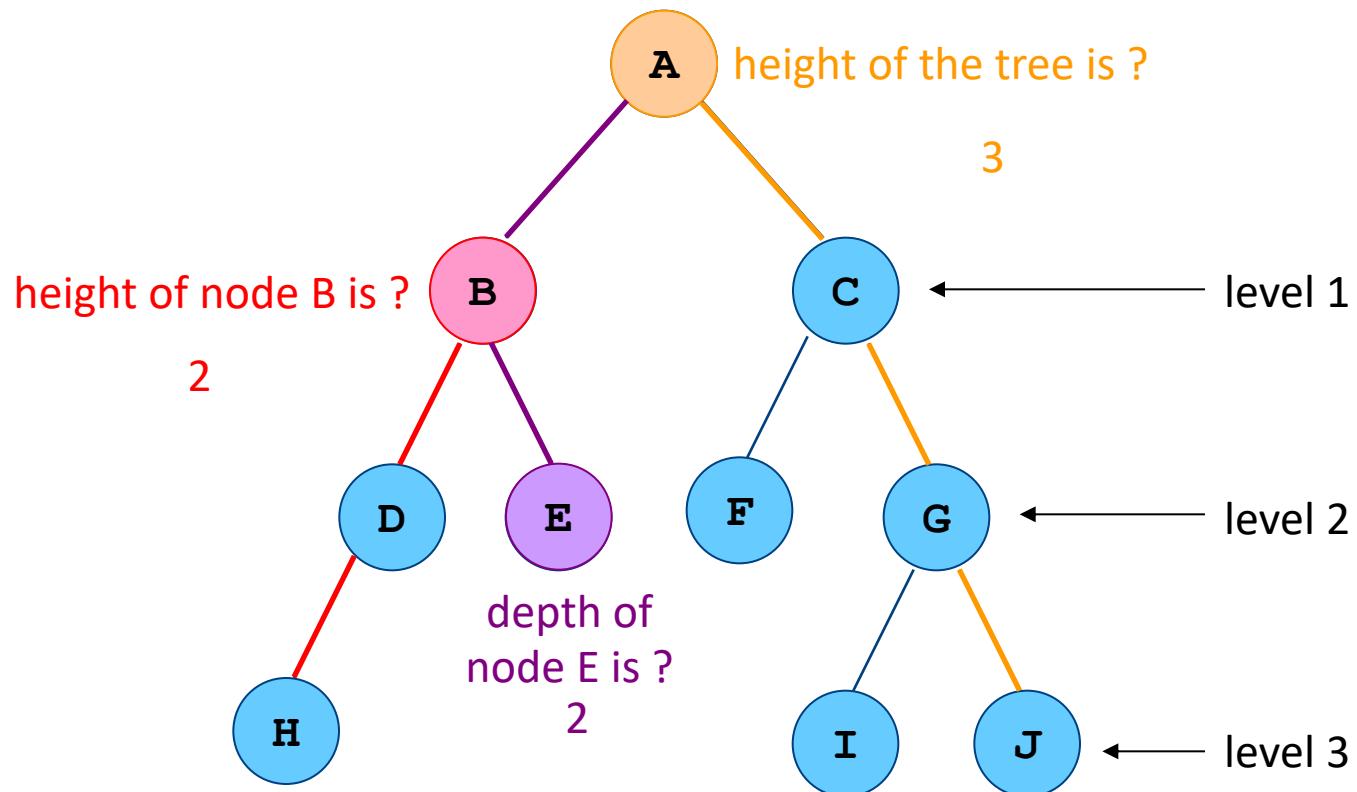
$$h \leq (n - 1)/2$$

$$e \leq 2^h$$

$$h \geq \log_2 e$$

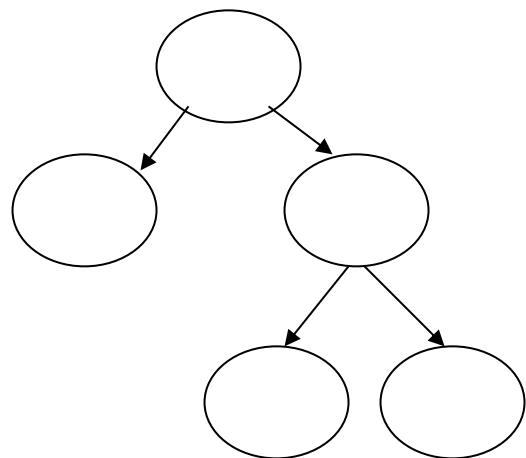
$$h \geq \log_2 (n + 1) - 1$$

Height of a Binary Tree



Full Binary Tree

- A full binary tree is a binary tree in which every node is a **leaf or has two children**.

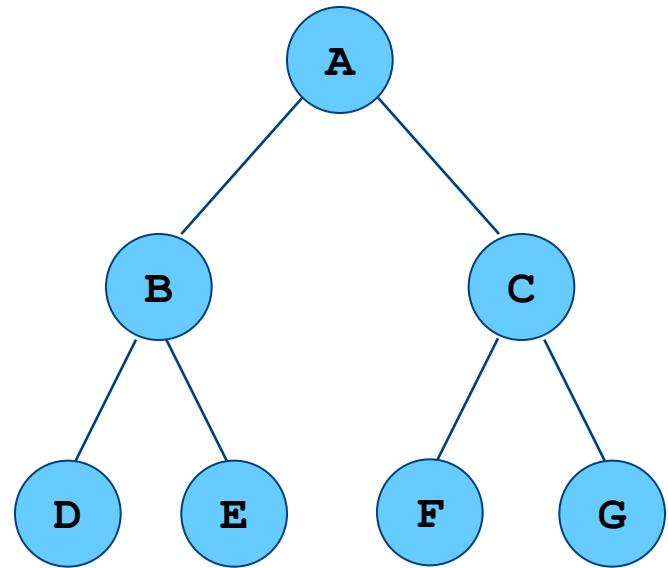


- **Theorem:** A FBT with n internal nodes has $n + 1$ leaves (external nodes).
- Proof by induction.
- Base case:
 - Binary Tree of one node (the root) has:
 - zero internal nodes
 - one external node (the root)
- Inductive Assumption:
 - Assume all FBTs with n internal nodes or less have $n + 1$ external nodes.

- Inductive Step - prove true for a tree with $n + 1$ internal nodes (i.e. a tree with $n + 1$ internal nodes has $(n + 1) + 1 = n + 2$ leaves)
 - Let T be a FBT of n internal nodes.
 - Therefore T has $n + 1$ leaf nodes. (Inductive Assumption)
 - Enlarge T so it has $n+1$ internal nodes by adding two nodes to some leaf. These new nodes are therefore leaf nodes.
 - Number of leaf nodes increases by 2, but the former leaf becomes internal.
 - So,
 - # internal nodes becomes $n + 1$,
 - # leaves becomes $(n + 1) + 2 - 1 = n + 2$

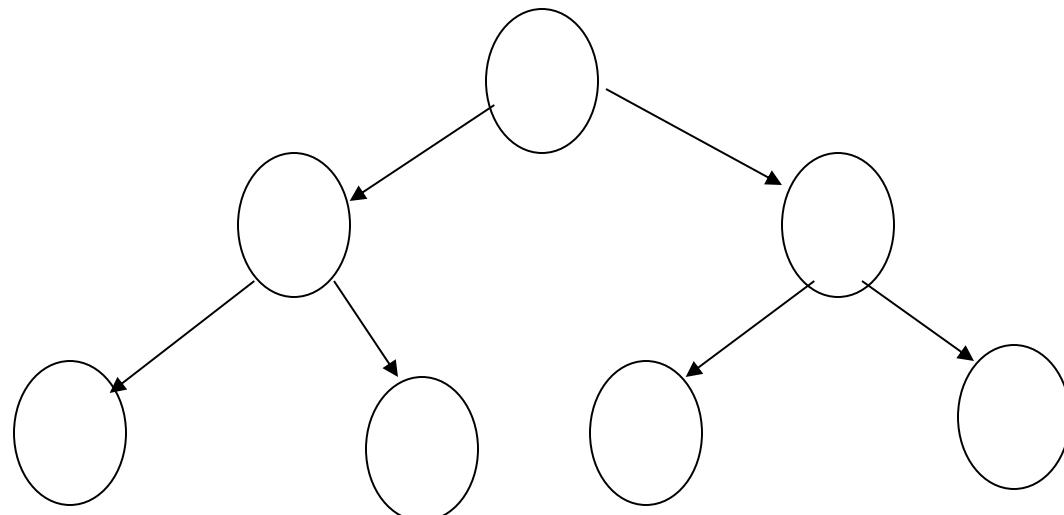
Perfect Binary Trees

- A binary tree is **perfect**, if
 - No node has only one child
 - And all the leaves have the same depth
- A perfect binary tree of height h has how many nodes?
 - $2^{h+1} - 1$ nodes, of which 2^h are leaves



Perfect Binary Tree

- A *Perfect Binary Tree* is a Full Binary Tree in which all leaves have the same depth.

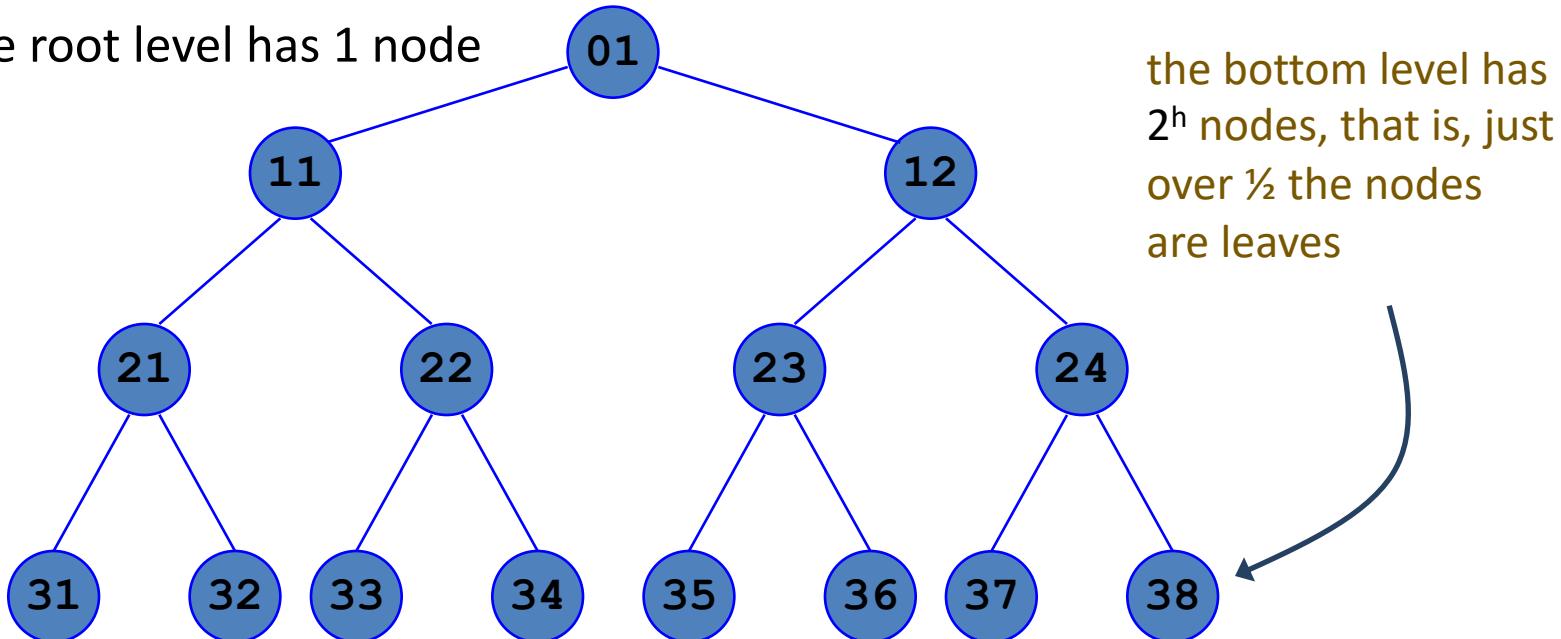


- **Theorem:** The number of nodes in a PBT is $2^{h+1}-1$, where h is height.
- Proof by induction on h , the height of the PBT:
 - Notice that the number of nodes at each level is 2^l . (Proof of this is a simple induction - left to student as exercise). Recall that the height of the root is 0.
 - Base Case:
The tree has one node; then $h = 0$ and $n = 1$ and $2^{(h+1)} - 1 = 2^{(0+1)} - 1 = 2^1 - 1 = 2 - 1 = 1 = n$.
 - Inductive Assumption:
Assume true for all PBTs with height $h \leq H$.

- Prove true for PBT with height H+1:
 - Consider a PBT with height $H + 1$. It consists of a root and two subtrees of height $\leq H$. Since the theorem is true for the subtrees (by the inductive assumption since they have height $\leq H$) the PBT with height $H+1$ has
 - $(2^{(H+1)} - 1)$ nodes for the left subtree
 - $+ (2^{(H+1)} - 1)$ nodes for the right subtree
 - $+ 1$ node for the root
 - Thus, $n = 2 * (2^{(H+1)} - 1) + 1$
 $= 2^{((H+1)+1)} - 2 + 1 = 2^{((H+1)+1)} - 1$

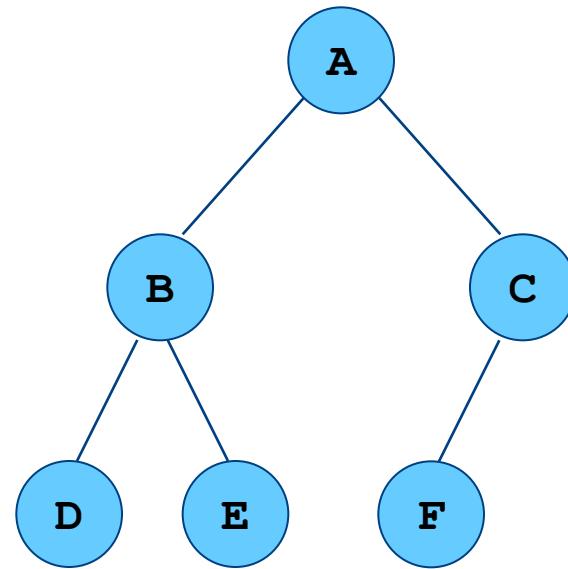
Height of a Perfect Tree

- Each level doubles the number of nodes
 - Level 1 has 2 nodes (2^1)
 - Level 2 has 4 nodes (2^2) or 2 times the number in Level 1
- Therefore a tree with h levels has $2^{h+1} - 1$ nodes
 - The root level has 1 node



Complete Binary Trees

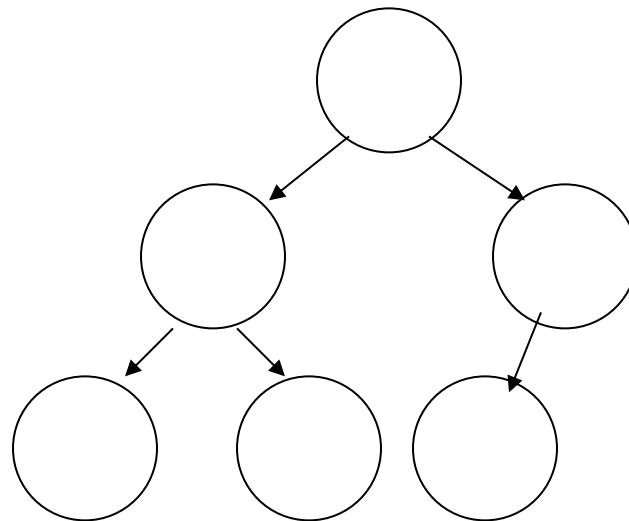
- A binary tree is **complete** if
 - The leaves are on at most two different levels,
 - The second to bottom level is completely filled in, and
 - The leaves on the bottom level are as far to the left as possible



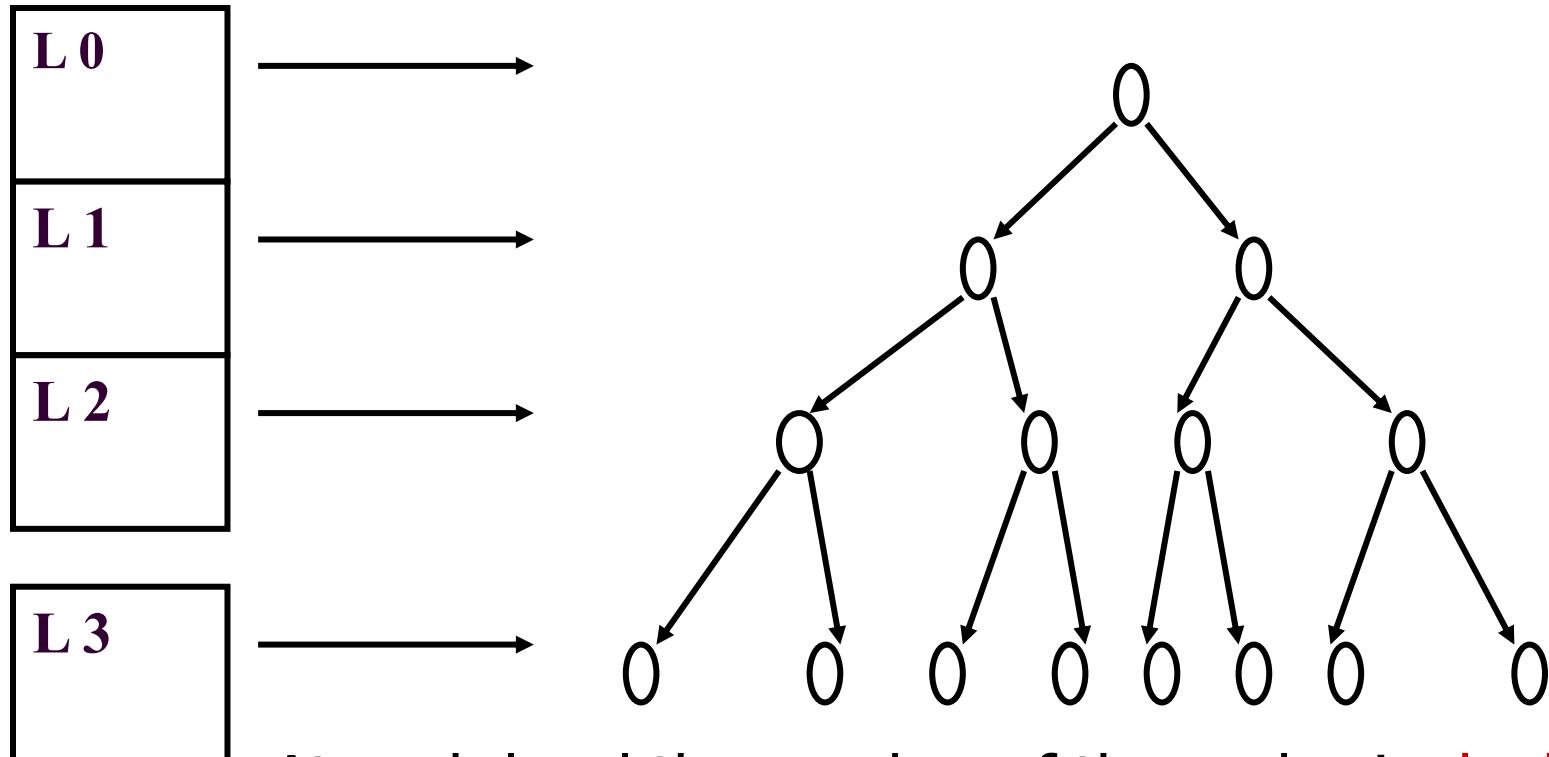
- Perfect trees are also complete

Complete Binary Tree

- A Complete Binary Tree is a binary tree in which **every level is completely filled, except possibly the bottom level** which is filled from left to right.



Complete Binary Tree



At each level the number of the nodes is **doubled**.
total number of nodes:

$$1 + 2 + 2^2 + 2^3 = 2^4 - 1 = 15$$

Complete Binary Tree

Number of the nodes in a tree with M levels:

$$1 + 2 + 2^2 + \dots + 2^M = 2^{(M+1)} - 1 = 2 * 2^M - 1$$

Let N be the number of the nodes.

$$N = 2 * 2^M - 1, \quad 2 * 2^M = N + 1$$

$$2^M = (N+1)/2$$

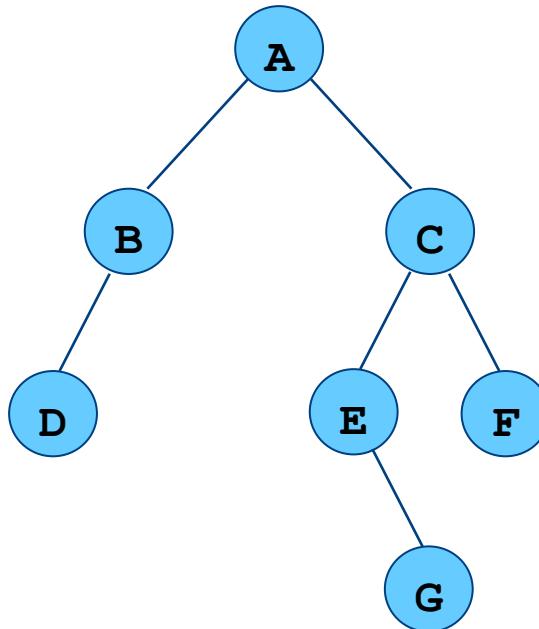
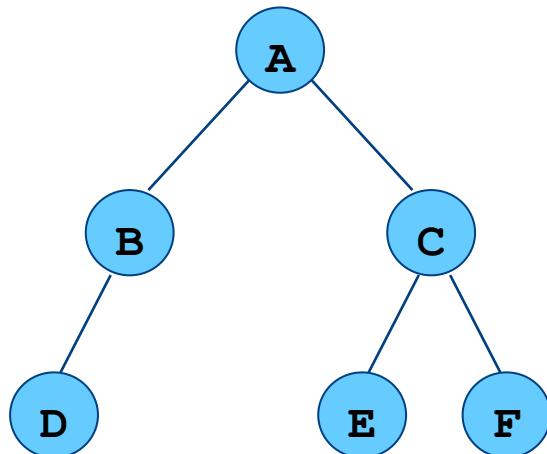
$$M = \log((N+1)/2)$$

N nodes : $\log((N+1)/2) = O(\log(N))$ levels

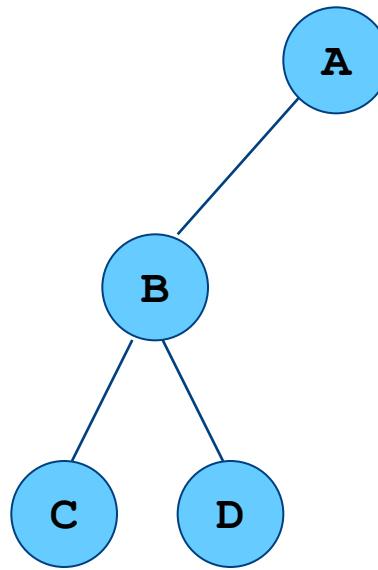
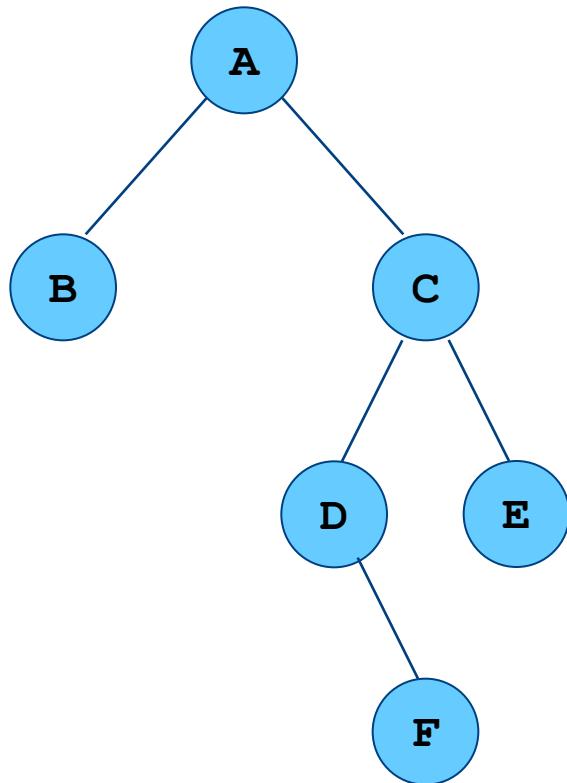
M levels: $2^{(M+1)} - 1 = O(2^M)$ nodes

Balanced Binary Trees

- A binary tree is **balanced** if the height of a node's right subtree is at most one different from the height of its left subtree



Unbalanced Binary Trees



BinaryTree ADT

- ❑ The BinaryTree ADT extends the Tree ADT, i.e., it inherits all the methods of the Tree ADT
- ❑ Additional methods:
 - position **left**(*p*)
 - position **right**(*p*)
 - position **sibling**(*p*)
- ❑ The above methods return **null** when there is no left, right, or sibling of *p*, respectively
- ❑ Update methods may be defined by data structures implementing the BinaryTree ADT

Finding an element in a Binary Tree

- Return a reference to node containing x, return null if x is not found

```
public BinaryNode<AnyType> find(AnyType x)
{
    return find(root, x);
}
private BinaryNode<AnyType> find( BinaryNode<AnyType> node, AnyType x)
{
    BinaryNode<AnyType> t = null;           // in case we don't find it
    if ( node.element.equals(x) )          // found it
        here??
        return node;

    // not here, look in the left subtree
    if(node.left != null)
        t = find(node.left,x);

    // if not in the left subtree, look in the right subtree
    if ( t == null)
        t = find(node.right,x);

    // return reference, null if not found
    return t;
}
```

Is this a full binary tree?

```
boolean  isFBT (BinaryNode<AnyType> t)
{
    // base case - an empty tree is a FBT
    if (t == null) return true;

    // determine if this node is "full"
    // if just one child, return - the tree is not full
    if ((t.left == null && t.right != null)
        || (t.right == null && t.left != null))
        return false;

    // if this node is full, "ask" its subtrees if they are full
    // if both are FBTs, then the entire tree is an FBT
    // if either of the subtrees is not FBT, then the tree is not
    return isFBT( t.right ) && isFBT( t.left );

}
```

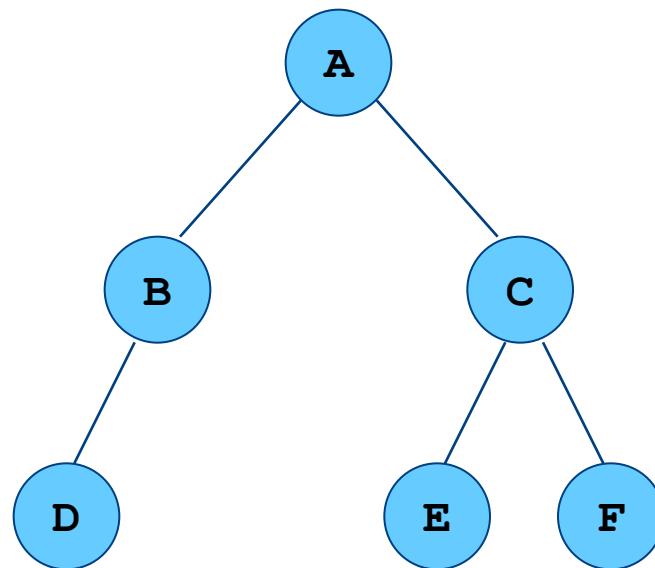
Binary Tree Traversals

- A traversal algorithm for a binary tree visits each node in the tree
 - Typically, it will do something while visiting each node!
- Traversal algorithms are naturally recursive
- There are three traversal methods
 - Inorder
 - Preorder
 - Postorder

InOrder Traversal Algorithm

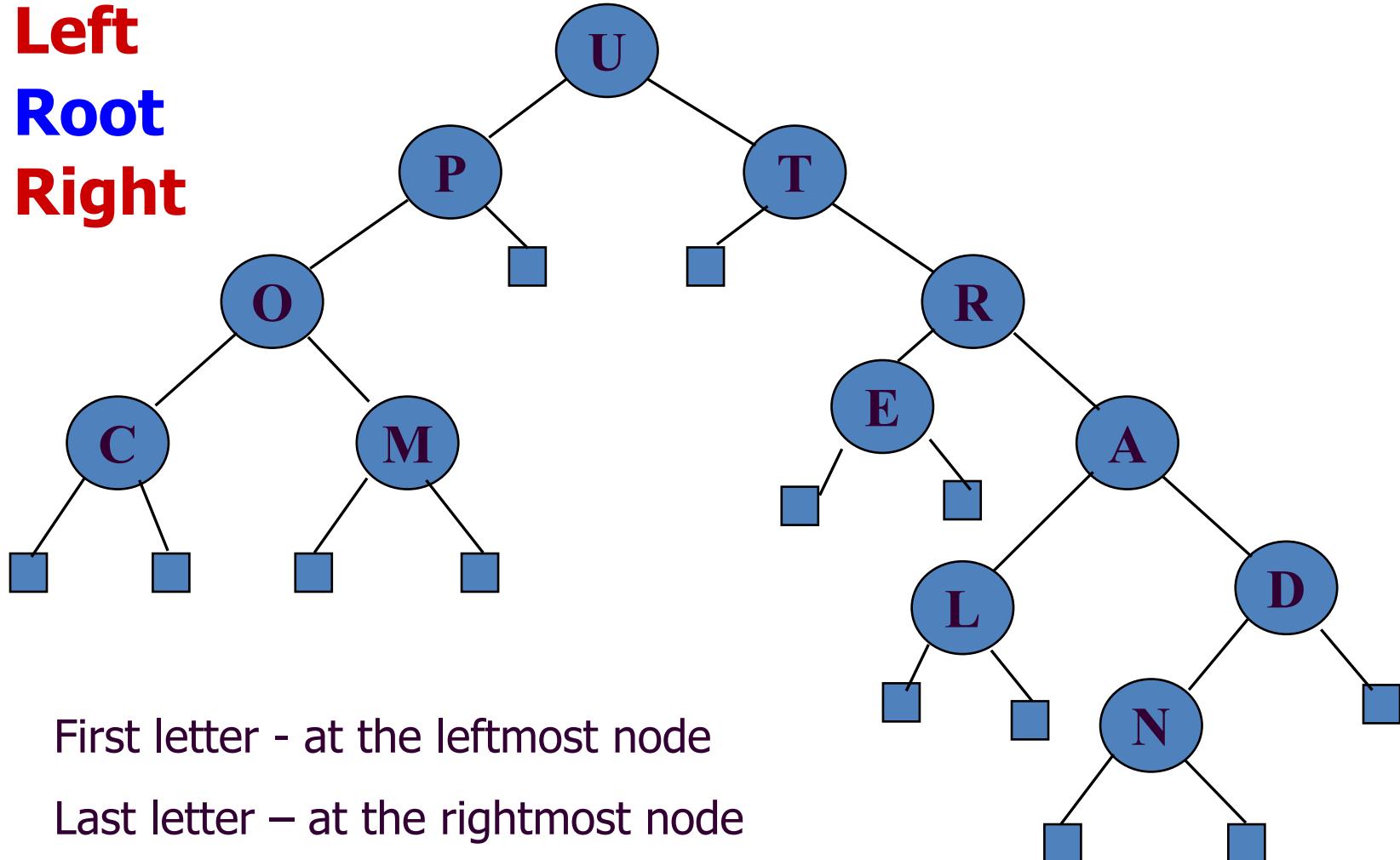
```
// InOrder traversal algorithm
void inOrder(Node n)  {
    if (n != null)  {
        inOrder(n.leftChild) ;
        visit(n) ;
        inOrder(n.rightChild) ;
    }
}
```

InOrder Traversal



Binary Tree – Inorder Traversal

Left
Root
Right



PreOrder Traversal Algorithm

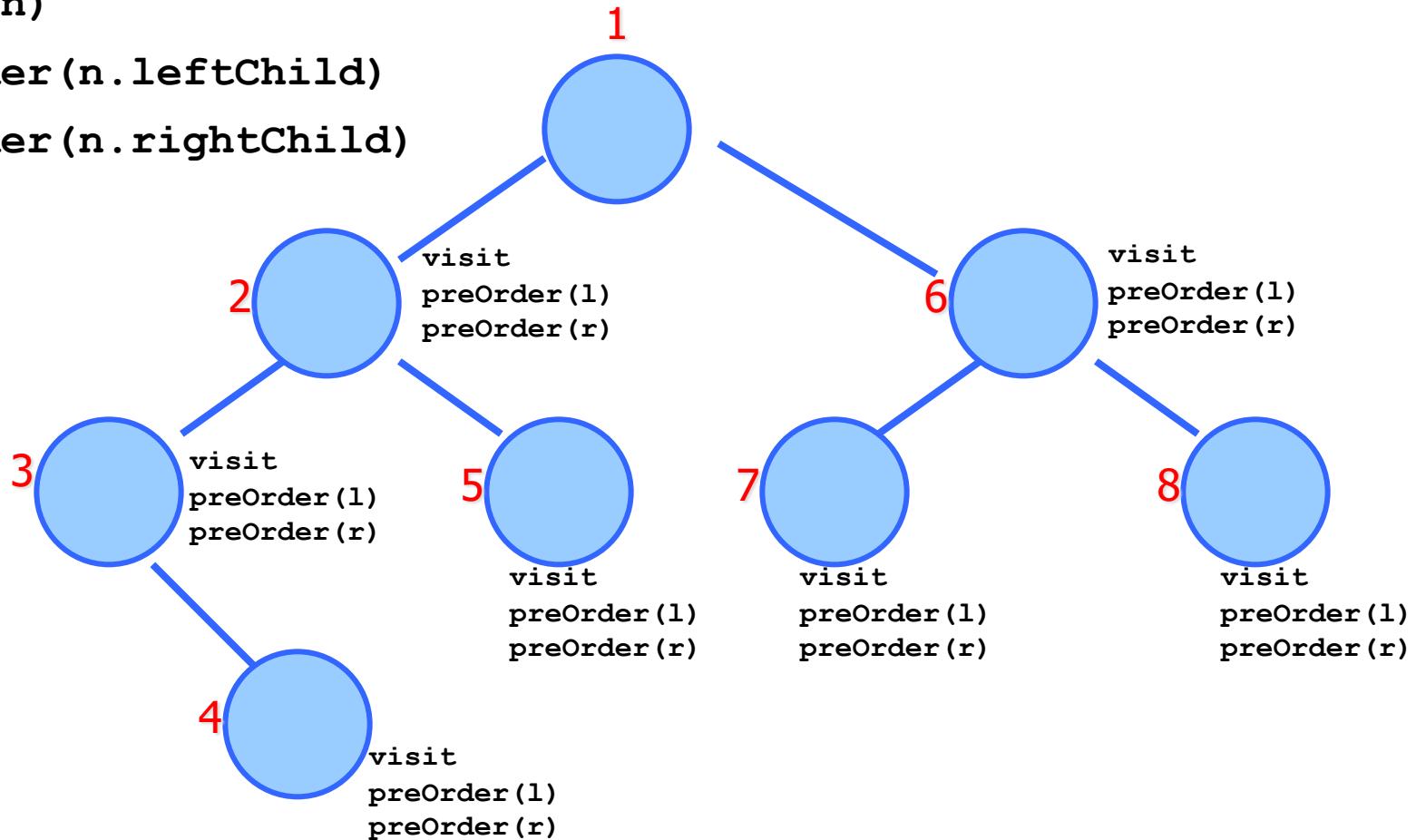
```
// PreOrder traversal algorithm
void preOrder(Node n)  {
    if (n != null)  {
        visit(n);
        preOrder(n.leftChild);
        preOrder(n.rightChild);
    }
}
```

PreOrder Traversal

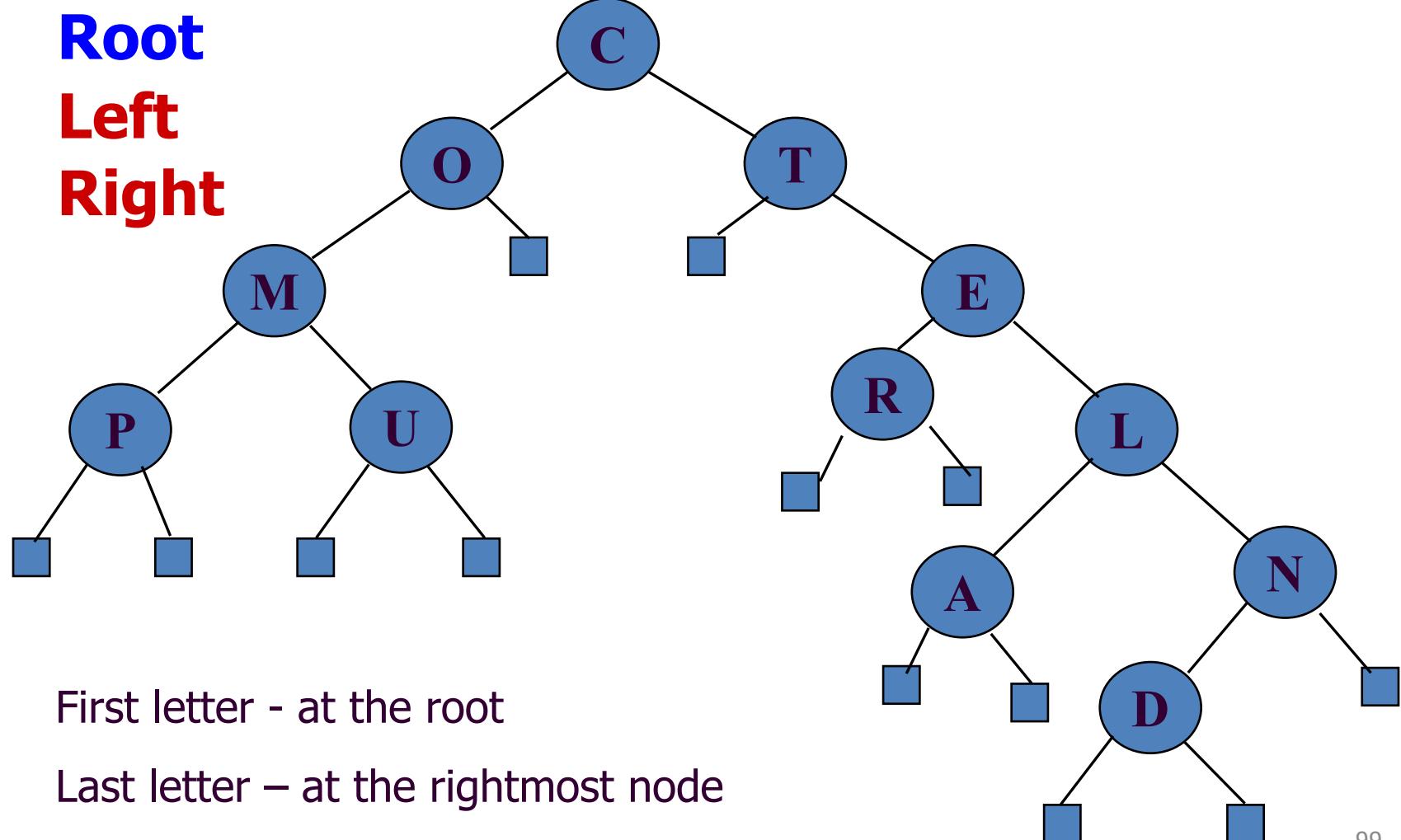
```
visit(n)
```

```
preOrder(n.leftChild)
```

```
preOrder(n.rightChild)
```



Binary Tree – Preorder Traversal



PostOrder Traversal Algorithm

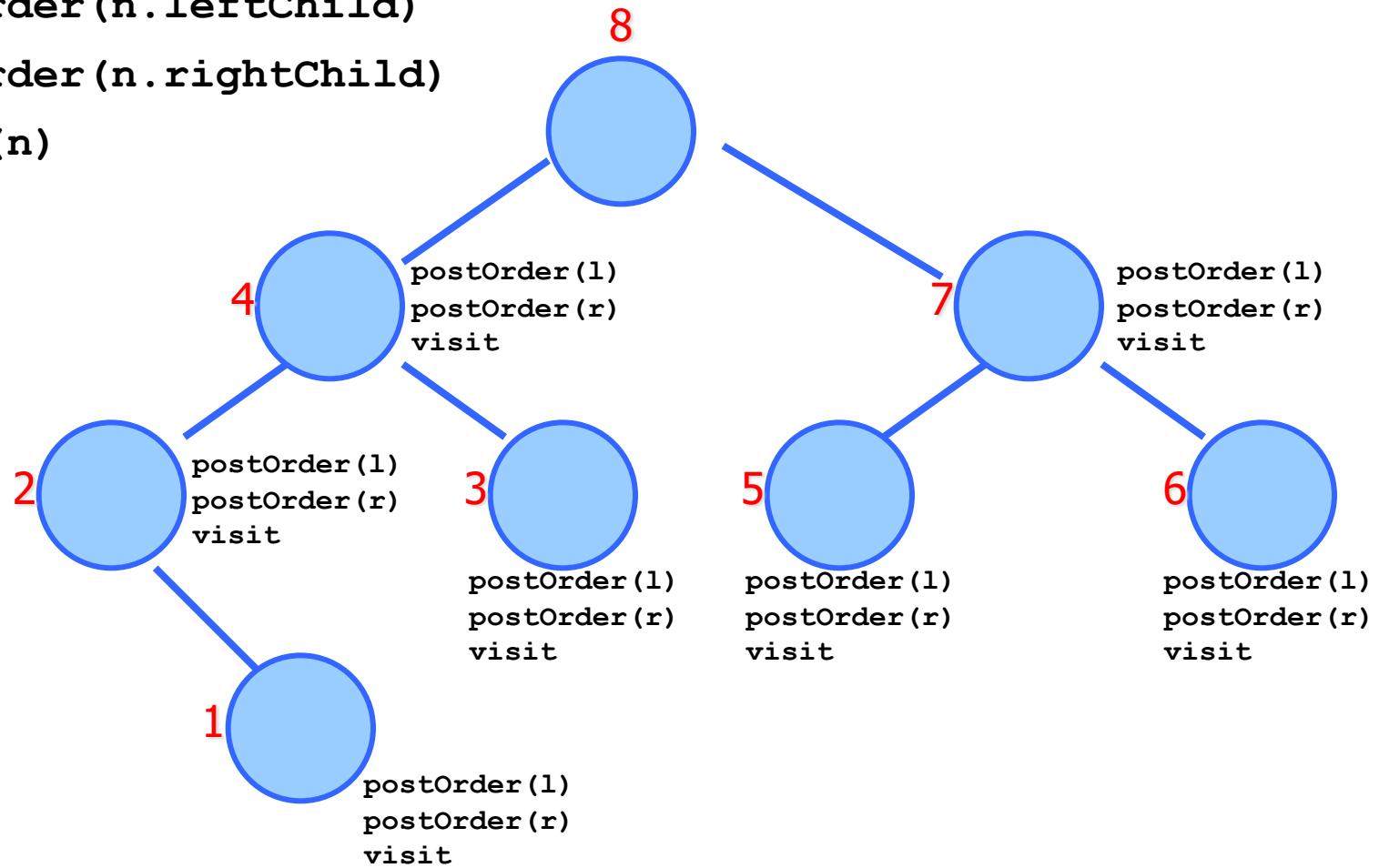
```
// PostOrder traversal algorithm
void postOrder(Node n) {
    if (n != null) {
        postOrder(n.leftChild);
        postOrder(n.rightChild);
        visit(n);
    }
}
```

PostOrder Traversal

`postOrder(n.leftChild)`

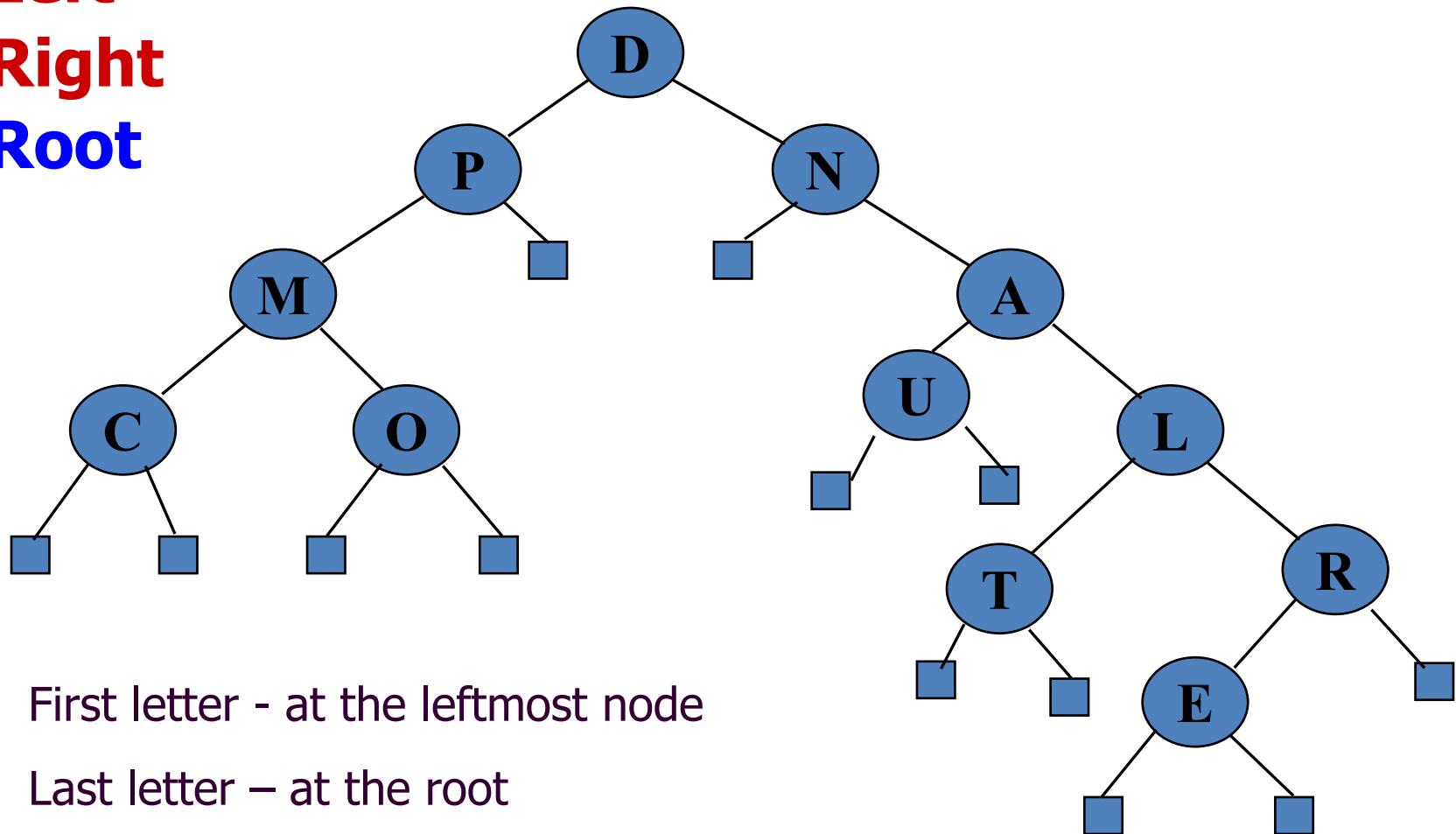
`postOrder(n.rightChild)`

`visit(n)`



Binary Tree – Postorder Traversal

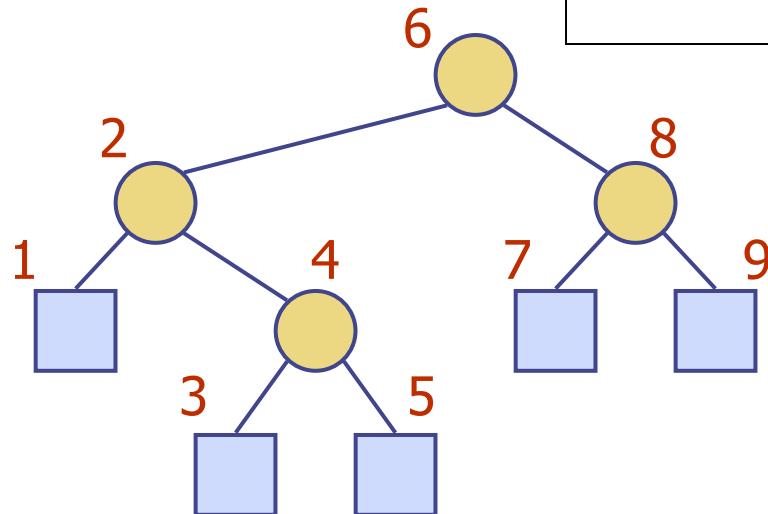
Left
Right
Root



Inorder Traversal

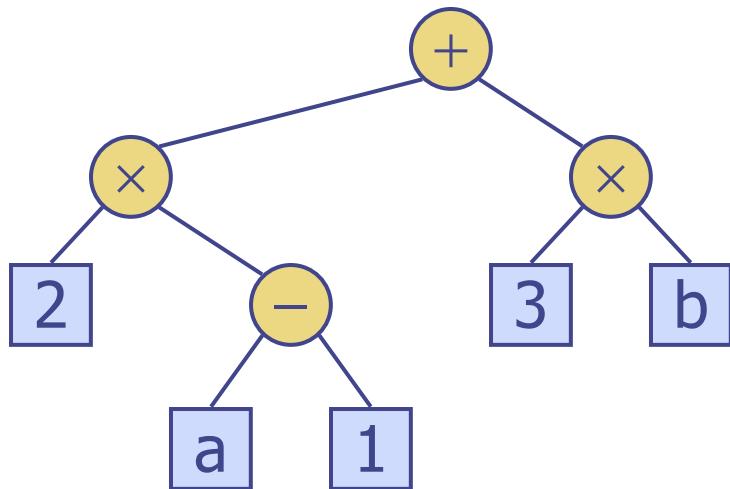
- In an inorder traversal a node is visited after its left subtree and before its right subtree
- Application: draw a binary tree
 - $x(v)$ = inorder rank of v
 - $y(v)$ = depth of v

```
Algorithm inOrder( $v$ )
if left ( $v$ )  $\neq$  null
    inOrder (left ( $v$ ))
    visit( $v$ )
if right( $v$ )  $\neq$  null
    inOrder (right ( $v$ ))
```



Print Arithmetic Expressions

- Specialization of an inorder traversal
 - print operand or operator when visiting node
 - print "(" before traversing left subtree
 - print ")" after traversing right subtree

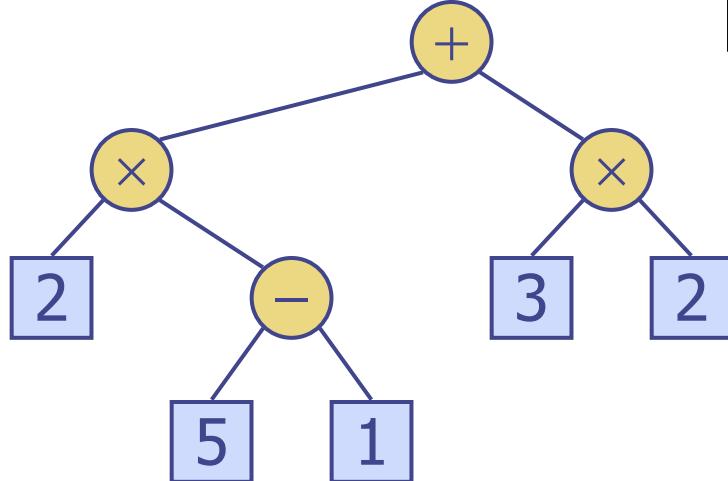


```
Algorithm printExpression(v)
if left (v)  $\neq$  null
    print(“(“)
    inOrder (left(v))
    print(v.element ())
    if right(v)  $\neq$  null
        inOrder (right(v))
    print (“)”)
```

$$((2 \times (a - 1)) + (3 \times b))$$

Evaluate Arithmetic Expressions

- Specialization of a postorder traversal
 - recursive method returning the value of a subtree
 - when visiting an internal node, combine the values of the subtrees



Algorithm *evalExpr(v)*

if *isExternal* (*v*)

return *v.element* ()

else

$x \leftarrow \text{evalExpr}(\text{left}(v))$

$y \leftarrow \text{evalExpr}(\text{right}(v))$

$\diamond \leftarrow \text{operator stored at } v$

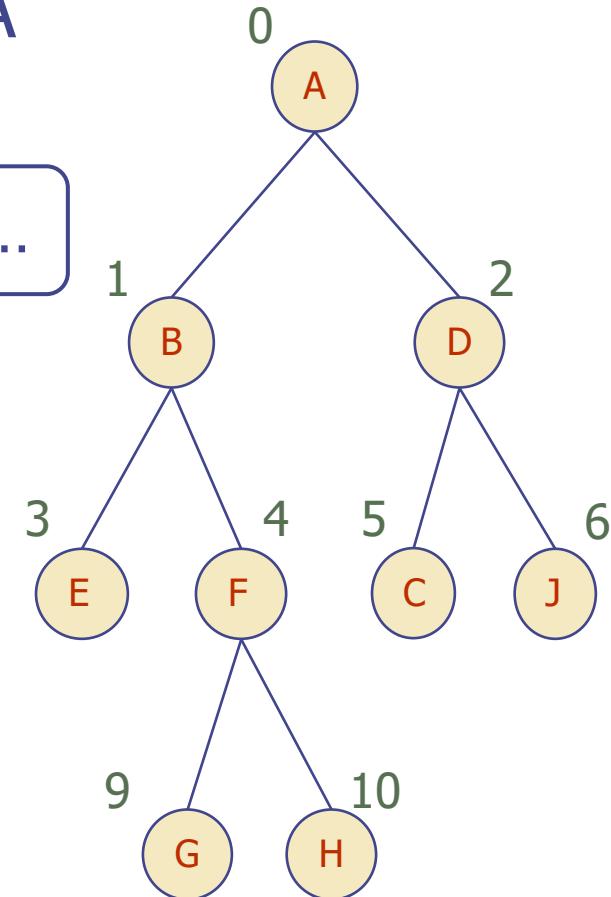
return *x* \diamond *y*

Binary Tree Implementation

- The binary tree can be implemented using a number of data structures
 - Arrays
 - Reference structures (similar to linked lists)

Array-Based Representation of Binary Trees

- Nodes are stored in an array A

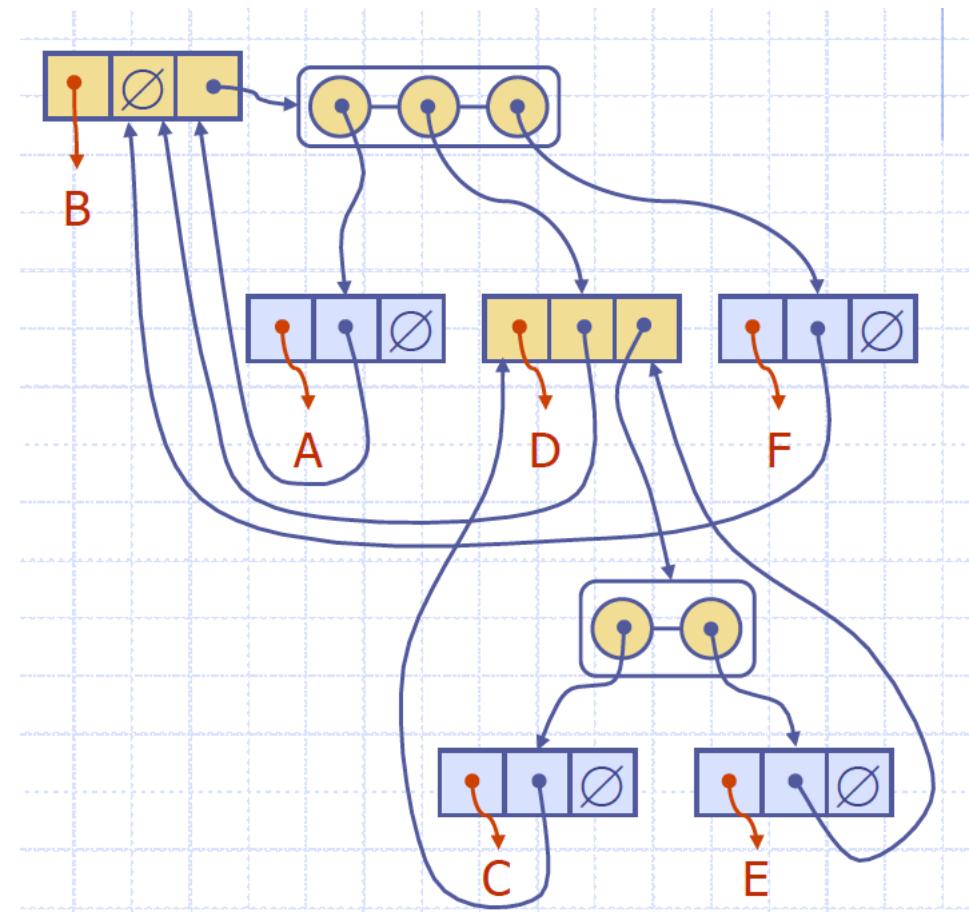
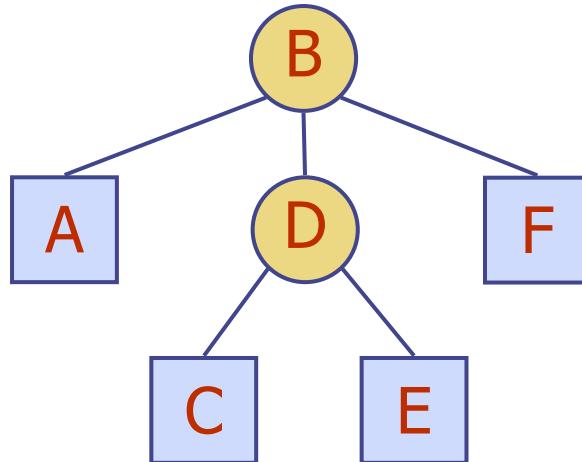


Node v is stored at $A[\text{rank}(v)]$

- $\text{rank}(\text{root}) = 0$
- if node is the left child of $\text{parent}(\text{node})$,
 $\text{rank}(\text{node}) = 2 \cdot \text{rank}(\text{parent}(\text{node})) + 1$
- if node is the right child of $\text{parent}(\text{node})$,
 $\text{rank}(\text{node}) = 2 \cdot \text{rank}(\text{parent}(\text{node})) + 2$

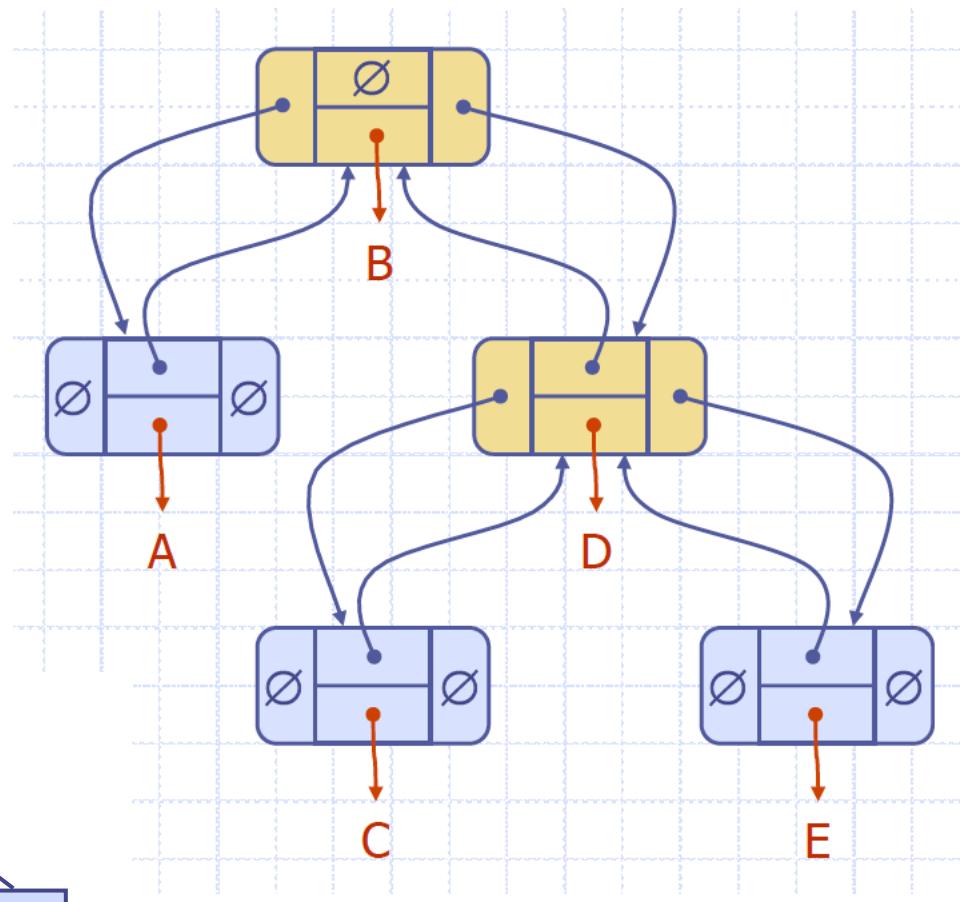
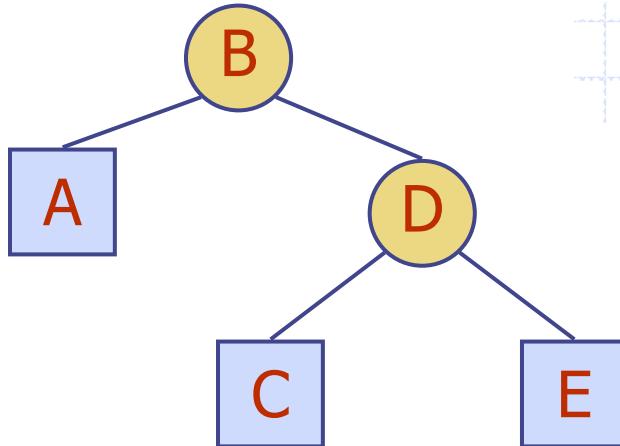
Linked Structure for Trees

- ❑ A node is represented by an object storing
 - Element
 - Parent node
 - Sequence of children nodes
- ❑ Node objects implement the Position ADT



Linked Structure for Binary Trees

- ❑ A node is represented by an object storing
 - Element
 - Parent node
 - Left child node
 - Right child node
- ❑ Node objects implement the Position ADT



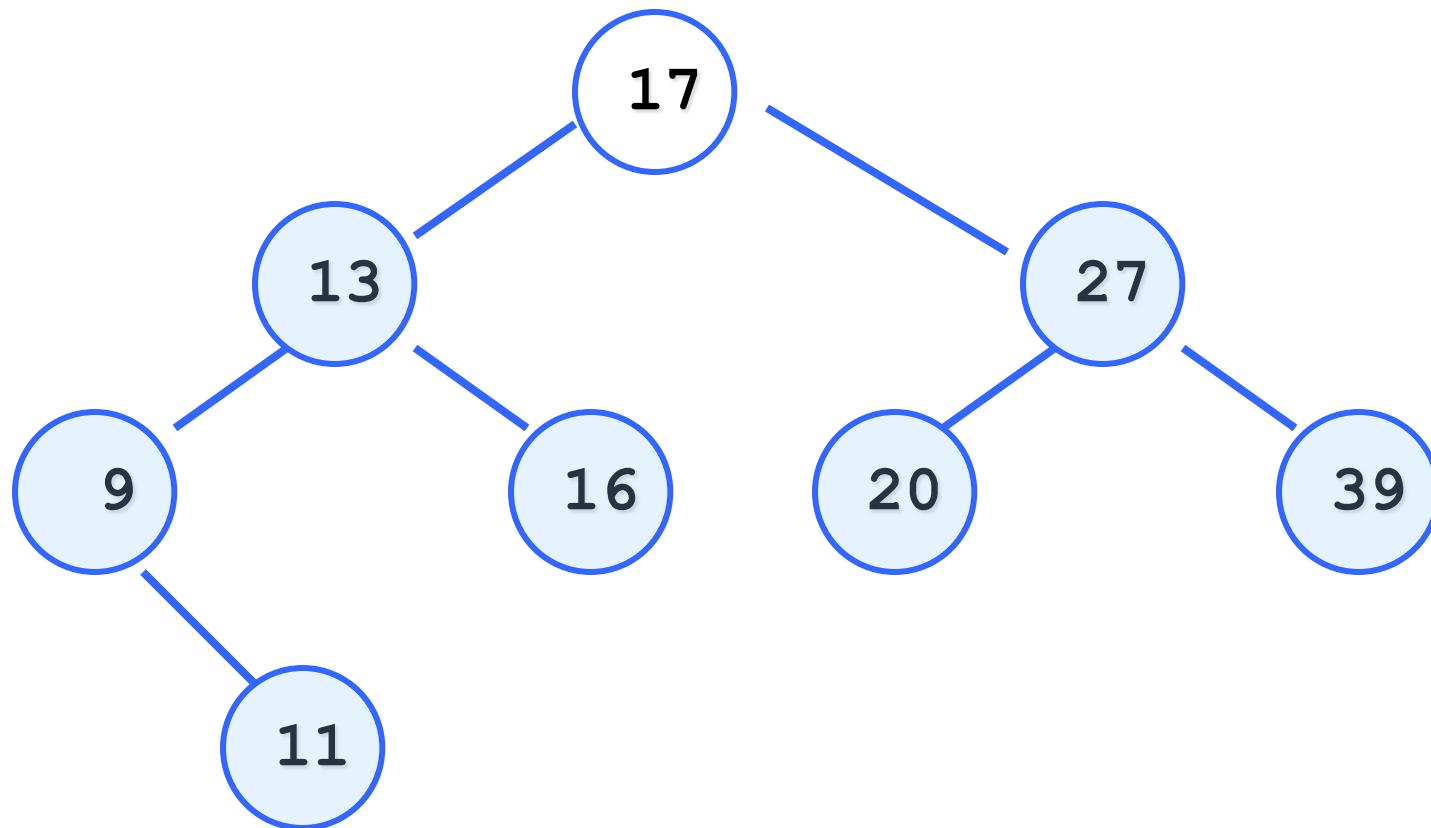
Problem: Accessing Sorted Data

- Consider maintaining data in some order
 - The data is to be frequently searched on the sort key e.g. a dictionary
- Possible solutions might be:
 - A sorted array
 - Access in $O(\log n)$ using binary search
 - Insertion and deletion in linear time
 - An ordered linked list
 - Access, insertion and deletion in linear time
 - Neither of these is efficient

Binary Search Trees

- A binary search tree (BST) is a binary tree with a special property
 - For all nodes in the tree:
 - All nodes in a left subtree have labels **less** than the label of the node
 - All nodes in a right subtree have labels **greater** than or equal to the label of the node
- Binary search trees are fully ordered

BST Example

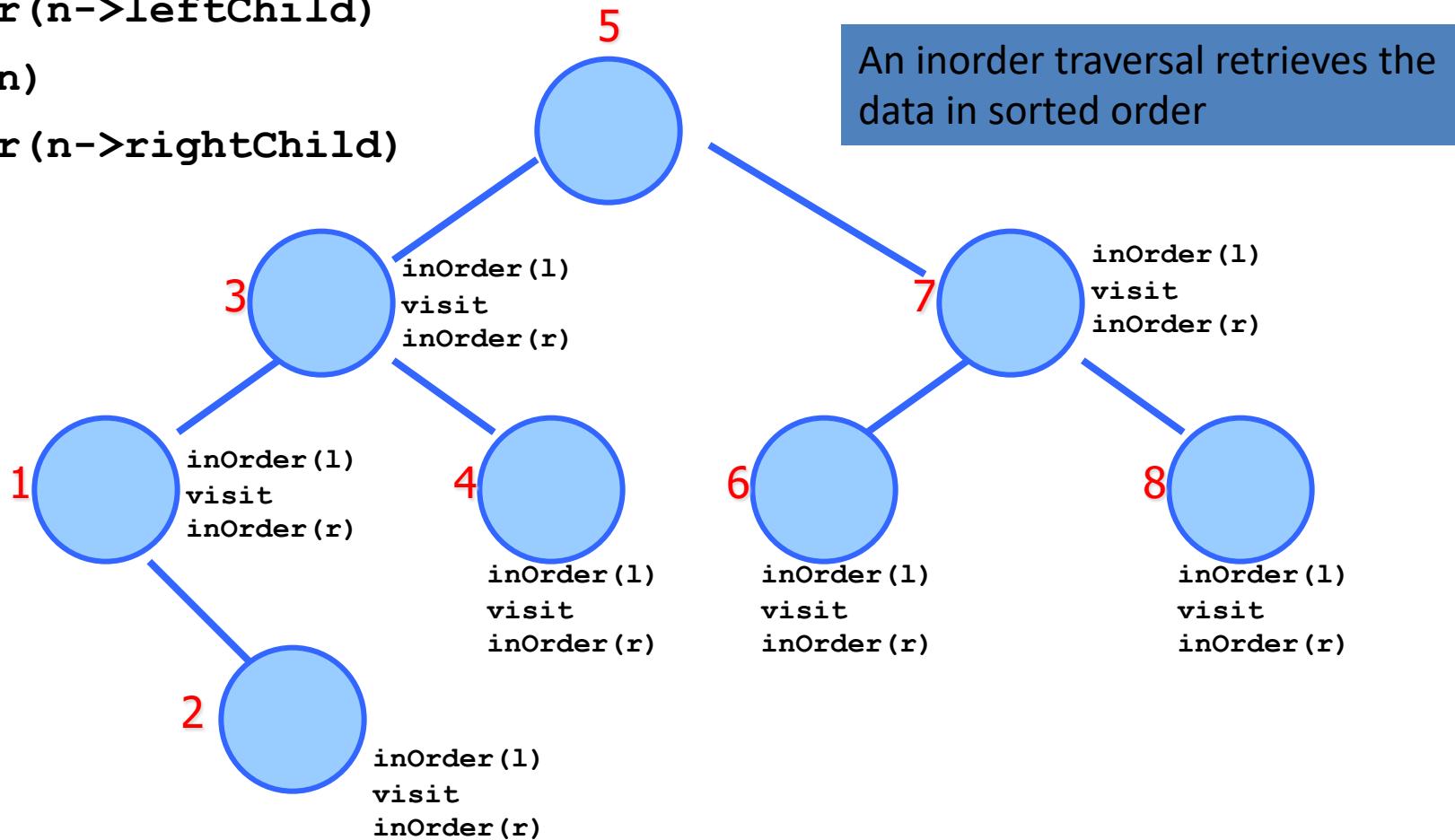


BST InOrder Traversal

```
inOrder(n->leftChild)
```

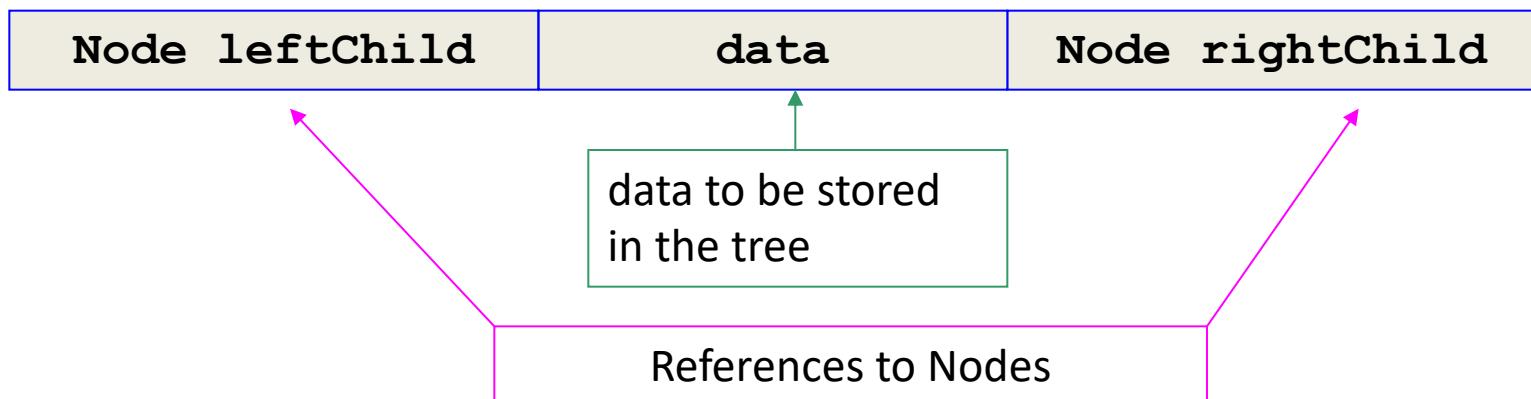
```
visit(n)
```

```
inOrder(n->rightChild)
```



BST Implementation

- Binary search trees can be implemented using a reference structure
- Tree nodes contain data and two pointers to nodes



BST Search

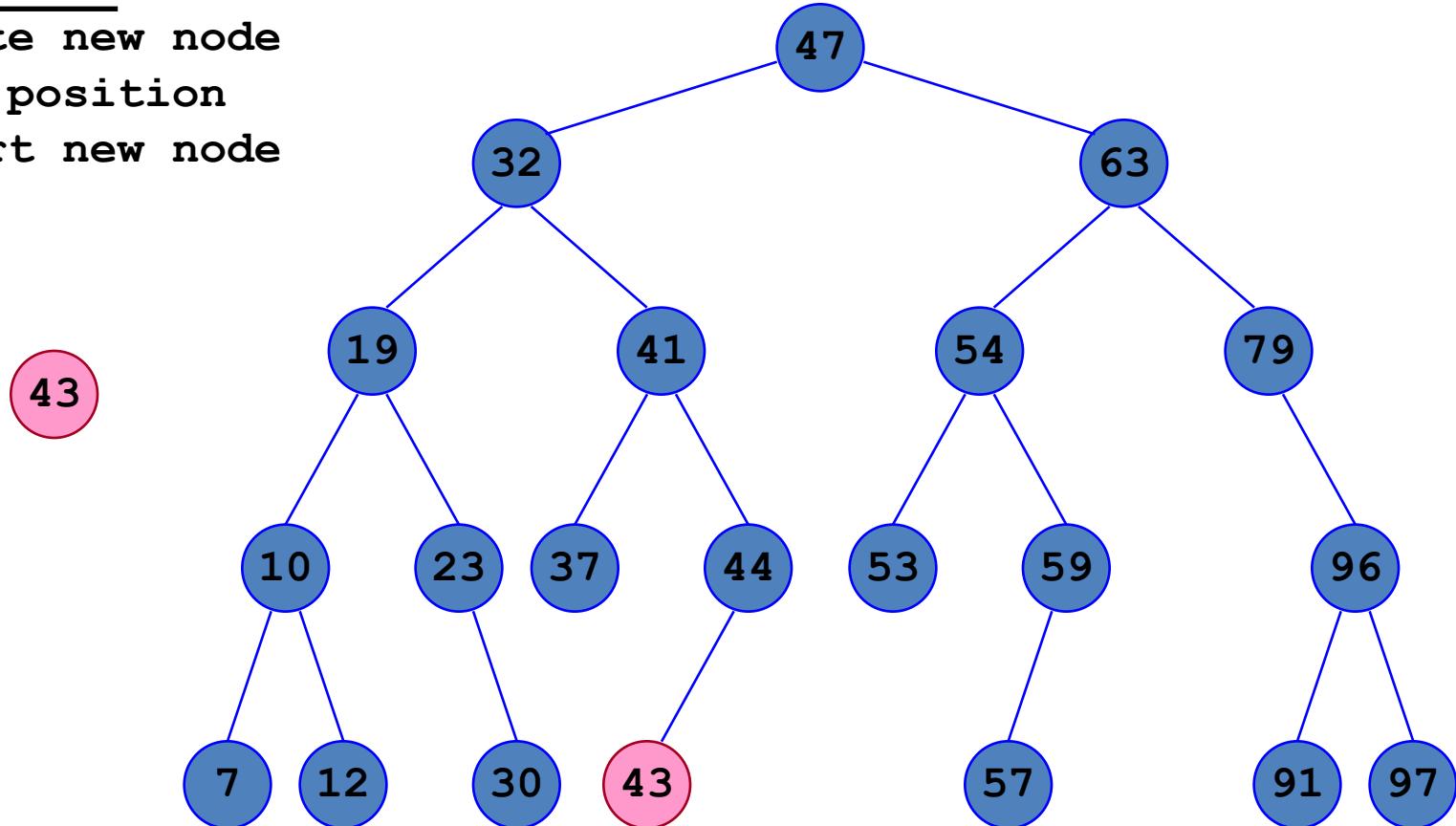
- To find a value in a BST search from the root node:
 - If the target is less than the value in the node search its left subtree
 - If the target is greater than the value in the node search its right subtree
 - Otherwise return true, or return data, etc.
- How many comparisons?
 - One for each node on the path
 - Worst case: height of the tree + 1

BST Insertion

- The BST property must hold after insertion
- Therefore the new node must be inserted in the correct position
 - This position is found by performing a search
 - If the search ends at the (null) left child of a node make its left child refer to the new node
 - If the search ends at the right child of a node make its right child refer to the new node
- The cost is about the same as the cost for the search algorithm, $O(\text{height})$

BST Insertion Example

insert 43
create new node
find position
insert new node



BST Deletion

- After deletion the BST property must hold
- Deletion is not as straightforward as search or insertion
 - So much so that sometimes it is not even implemented!
 - Deleted nodes are marked as deleted in some way
- There are a number of different cases that must be considered

BST Deletion Cases

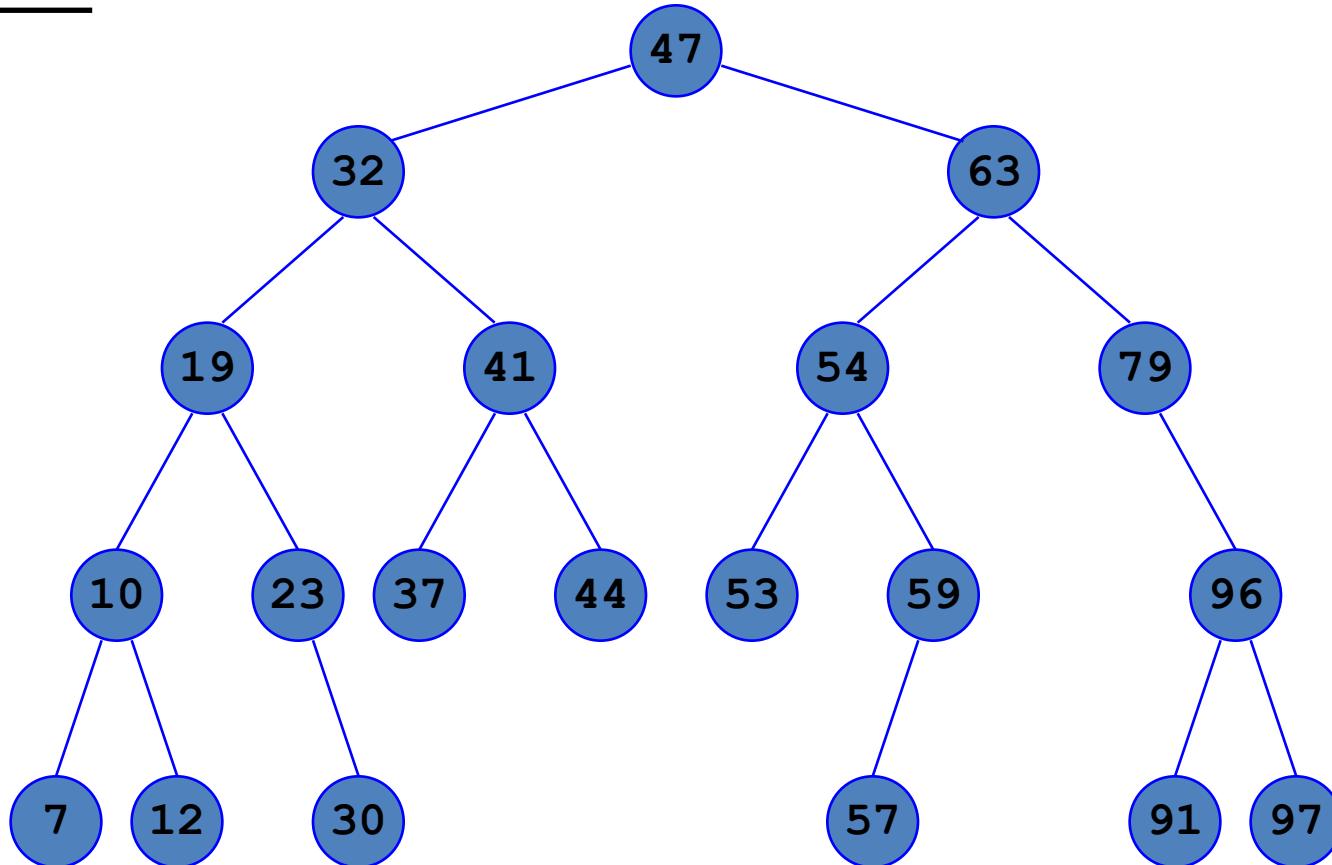
- The node to be deleted has no children
- The node to be deleted has one child
- The node to be deleted has two children

BST Deletion Cases

- The node to be deleted has no children
 - Remove it (assigning null to its parent's reference)

BST Deletion – target is a leaf

delete 30



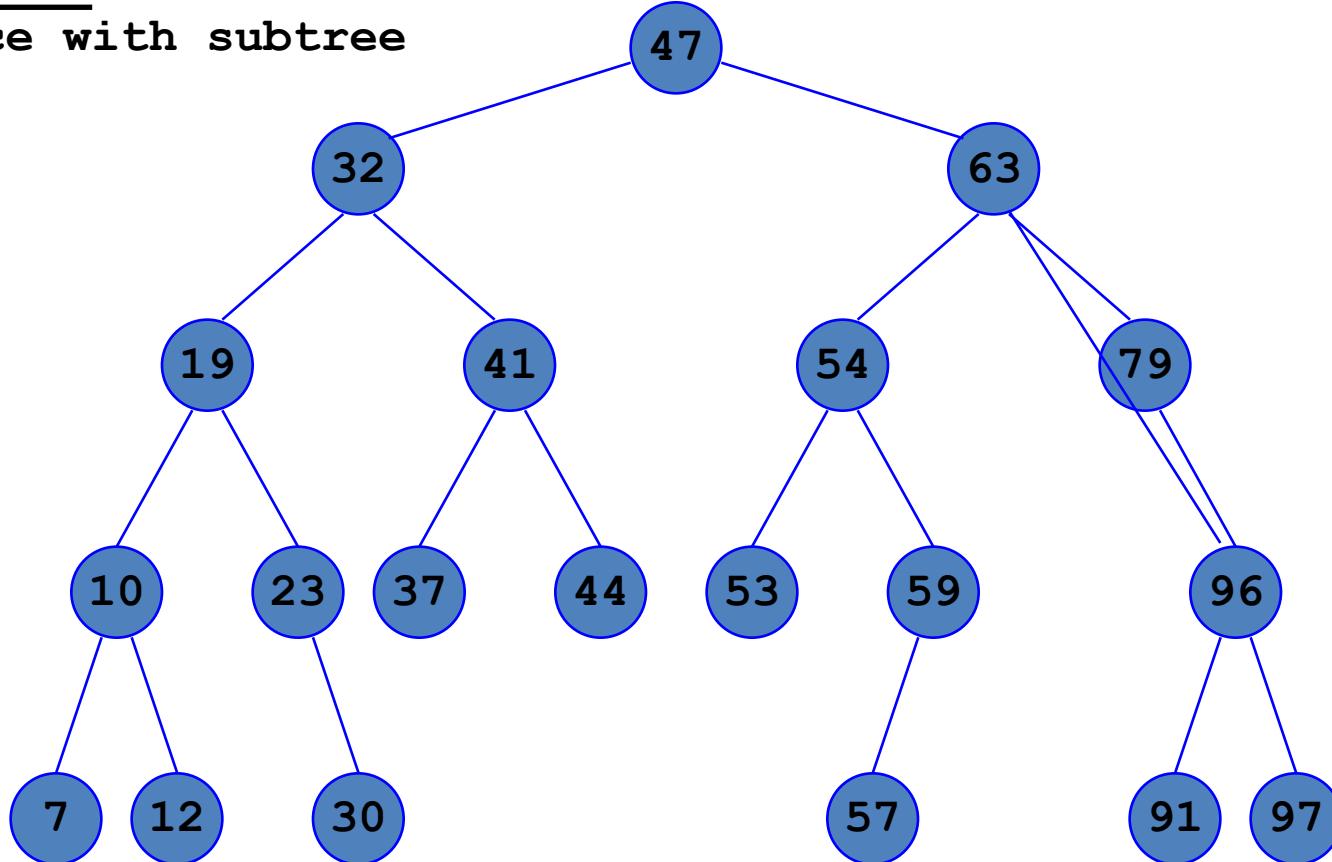
BST Deletion Cases

- The node to be deleted has one child
 - Replace the node with its subtree

BST Deletion – target has one child

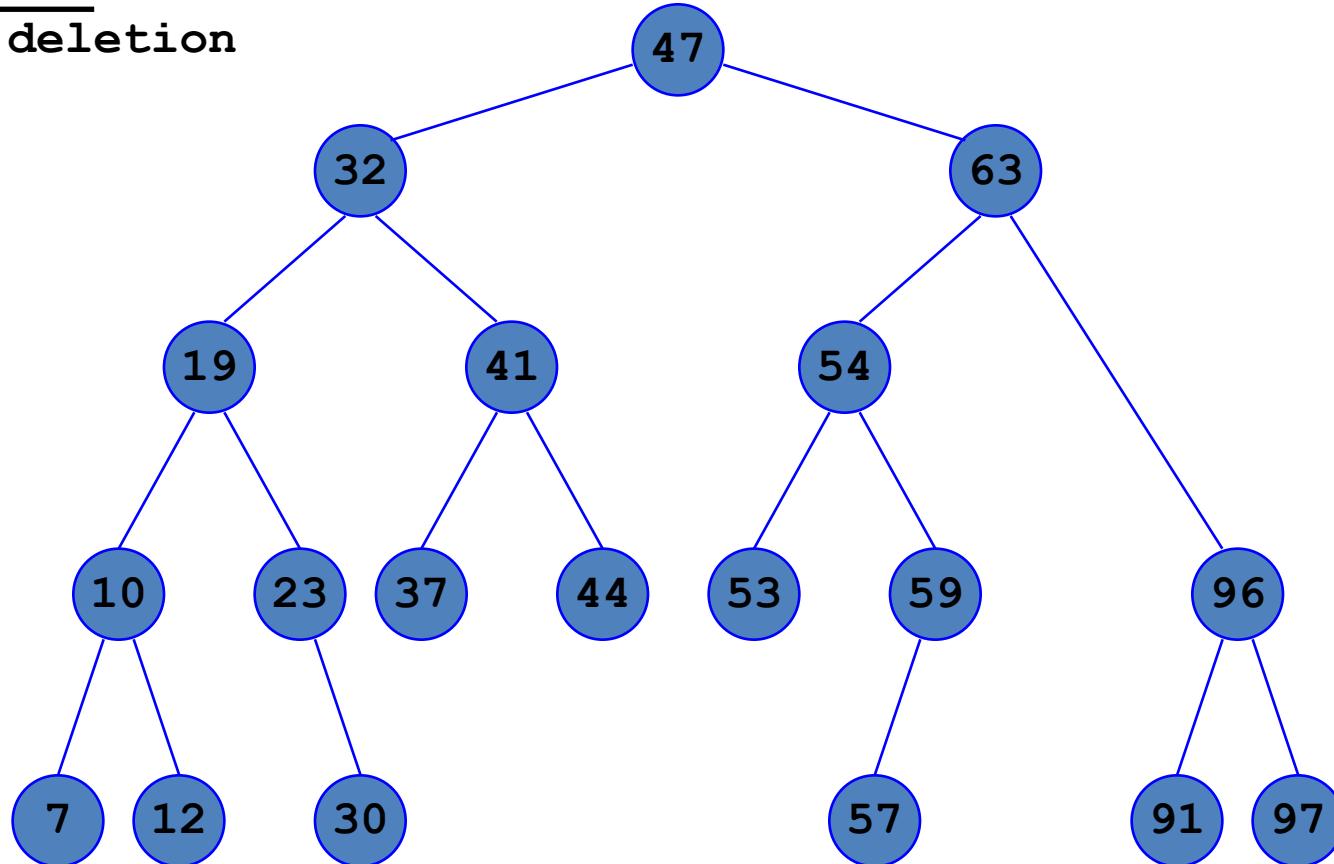
delete 79

replace with subtree



BST Deletion – target has one child

delete 79
after deletion



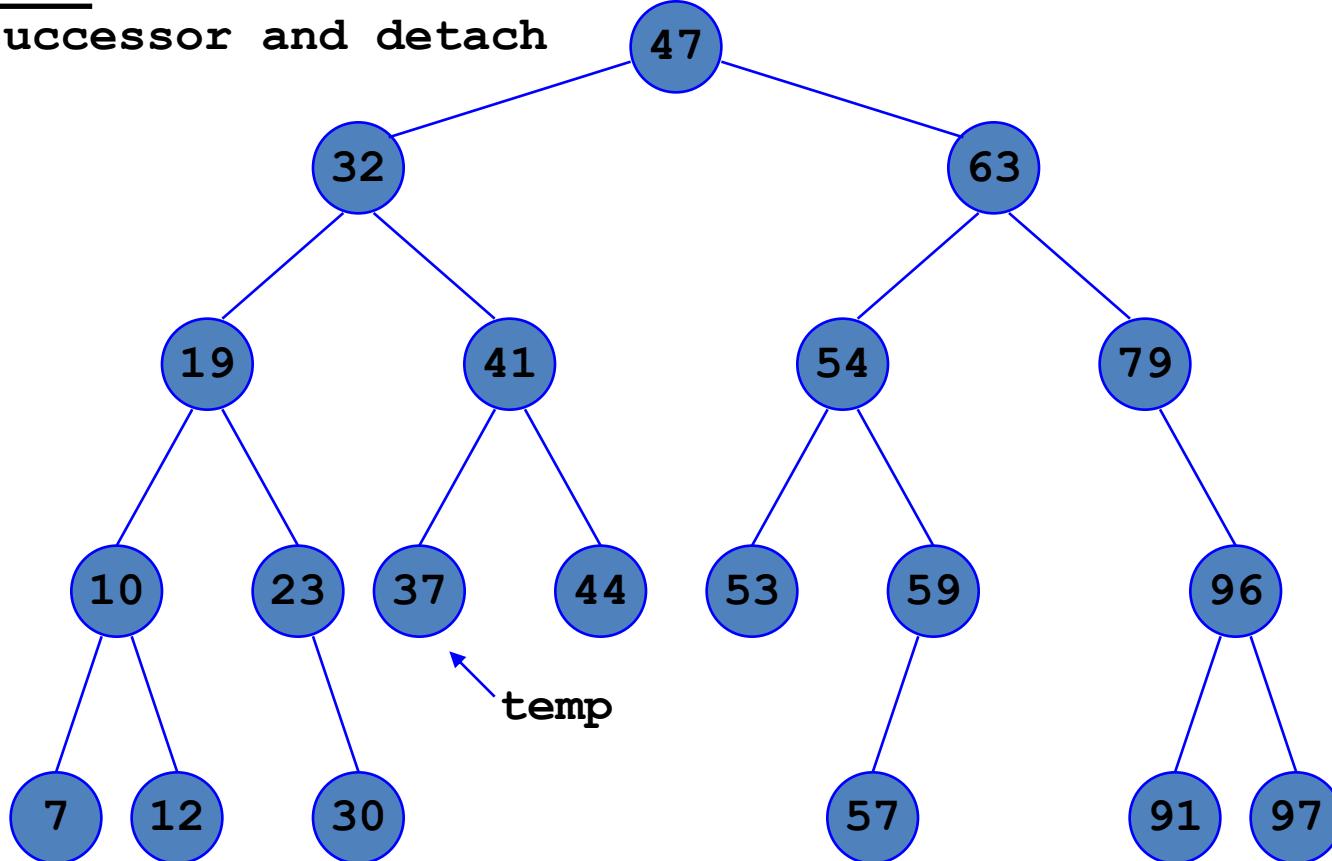
BST Deletion Cases

- The node to be deleted has two children
 - Replace the node with its **successor**, the **left most node** of its **right subtree**
 - It is also possible to replace the node with its **predecessor**, the **right most node** of its **left subtree**
 - If that node has a child (and it can have at most one child) attach it to the node's parent
 - Why can a predecessor or successor have at most one child?

BST Deletion – target has 2 children

delete 32

find successor and detach

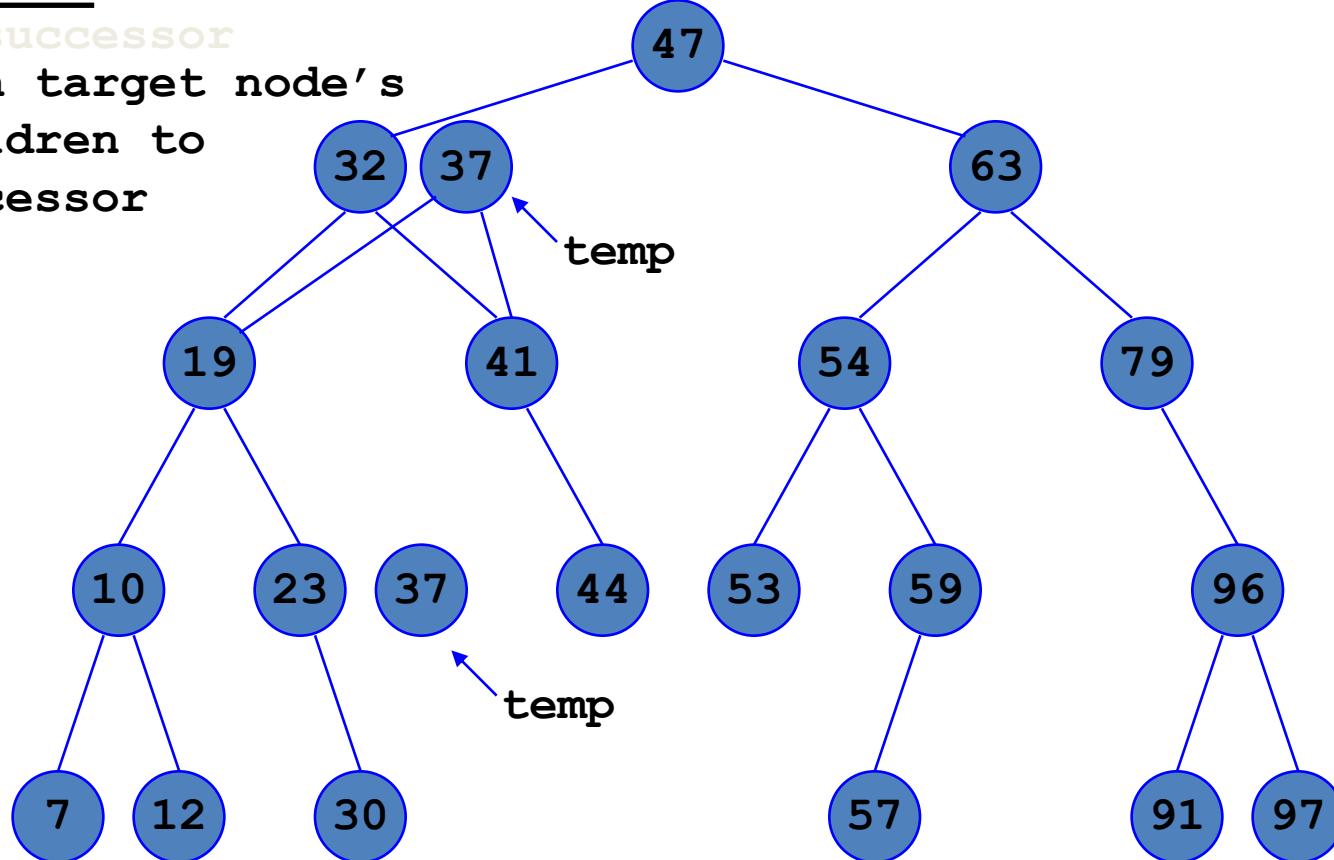


BST Deletion – target has 2 children

delete 32

find successor

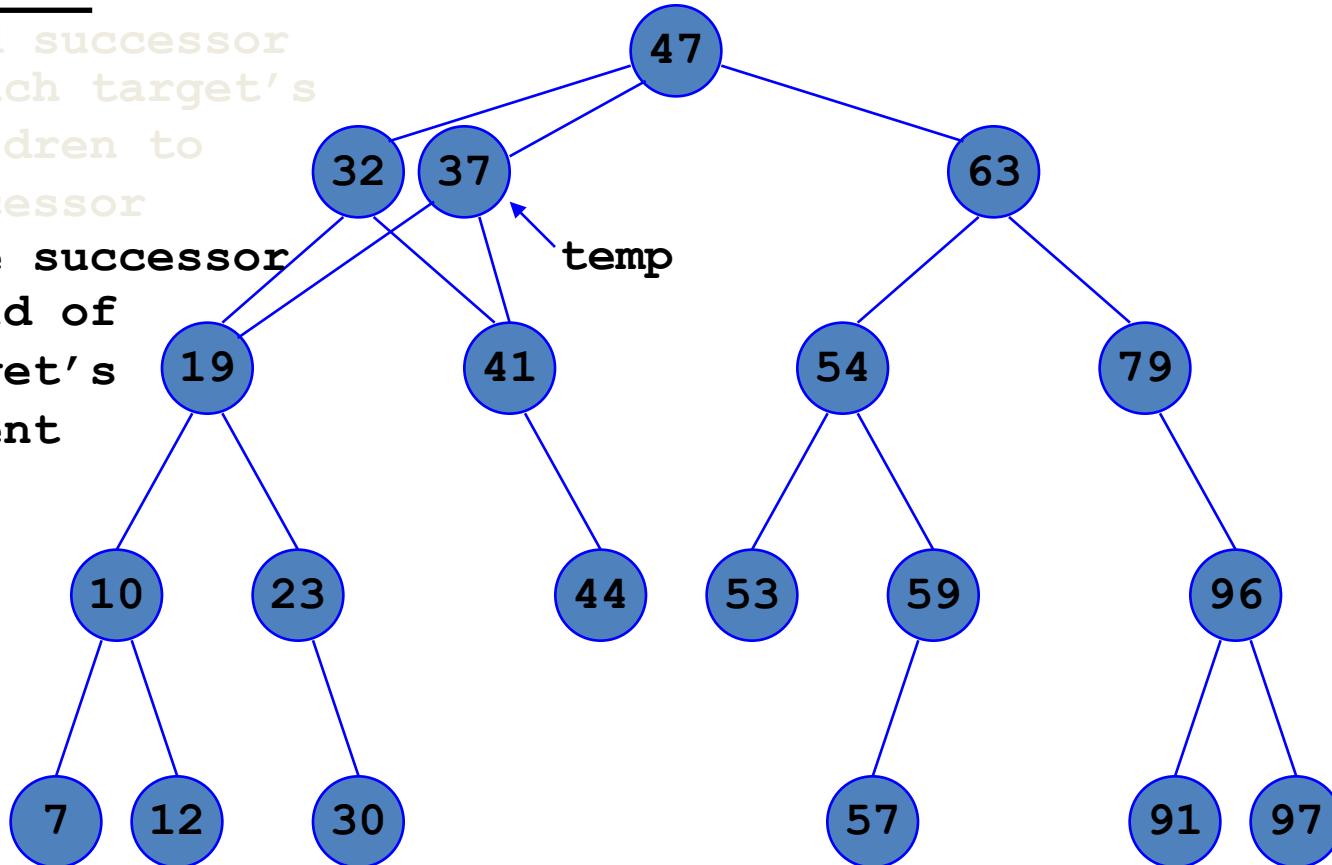
attach target node's
children to
successor



BST Deletion – target has 2 children

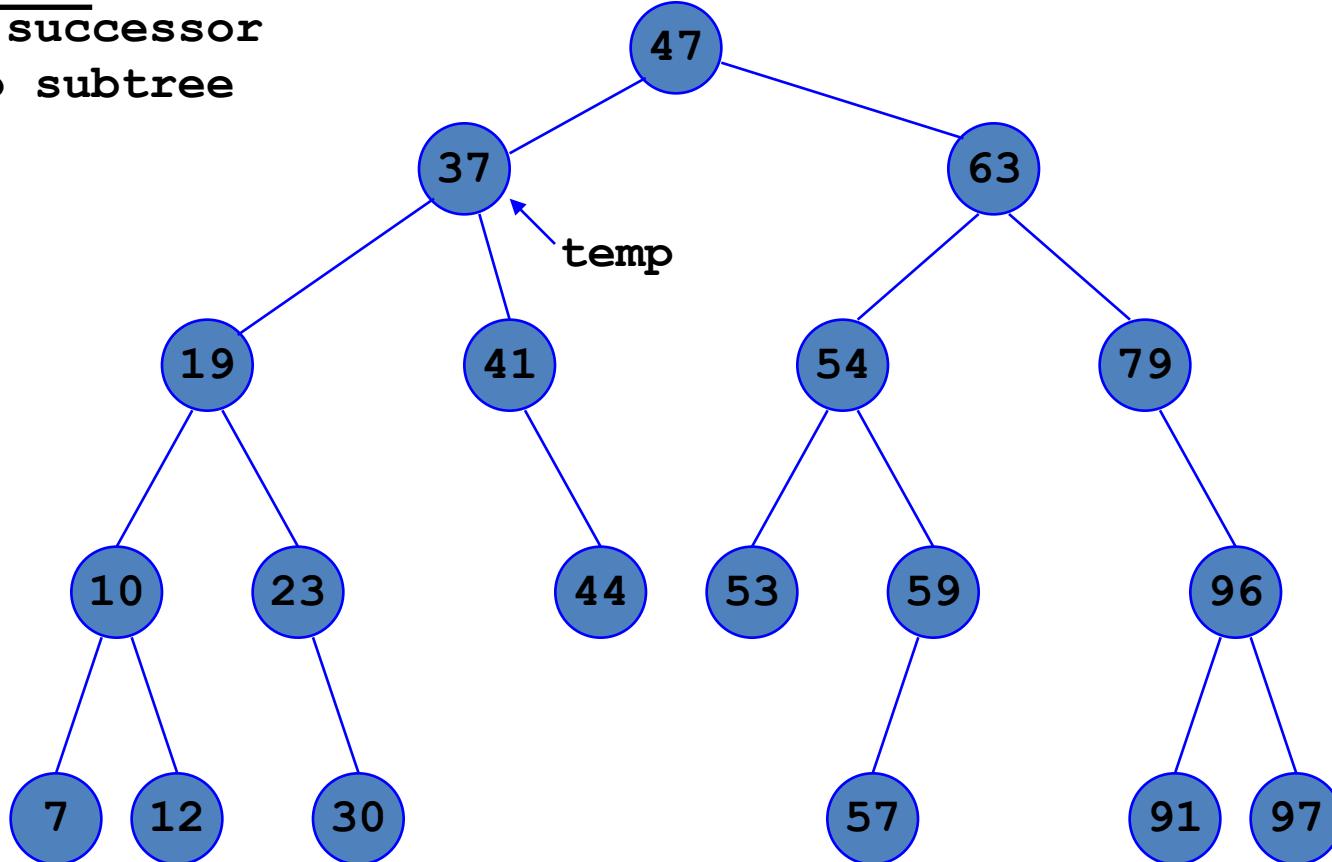
delete 32

- find successor
 - attach target's children to successor
 - make successor child of target's parent



BST Deletion – target has 2 children

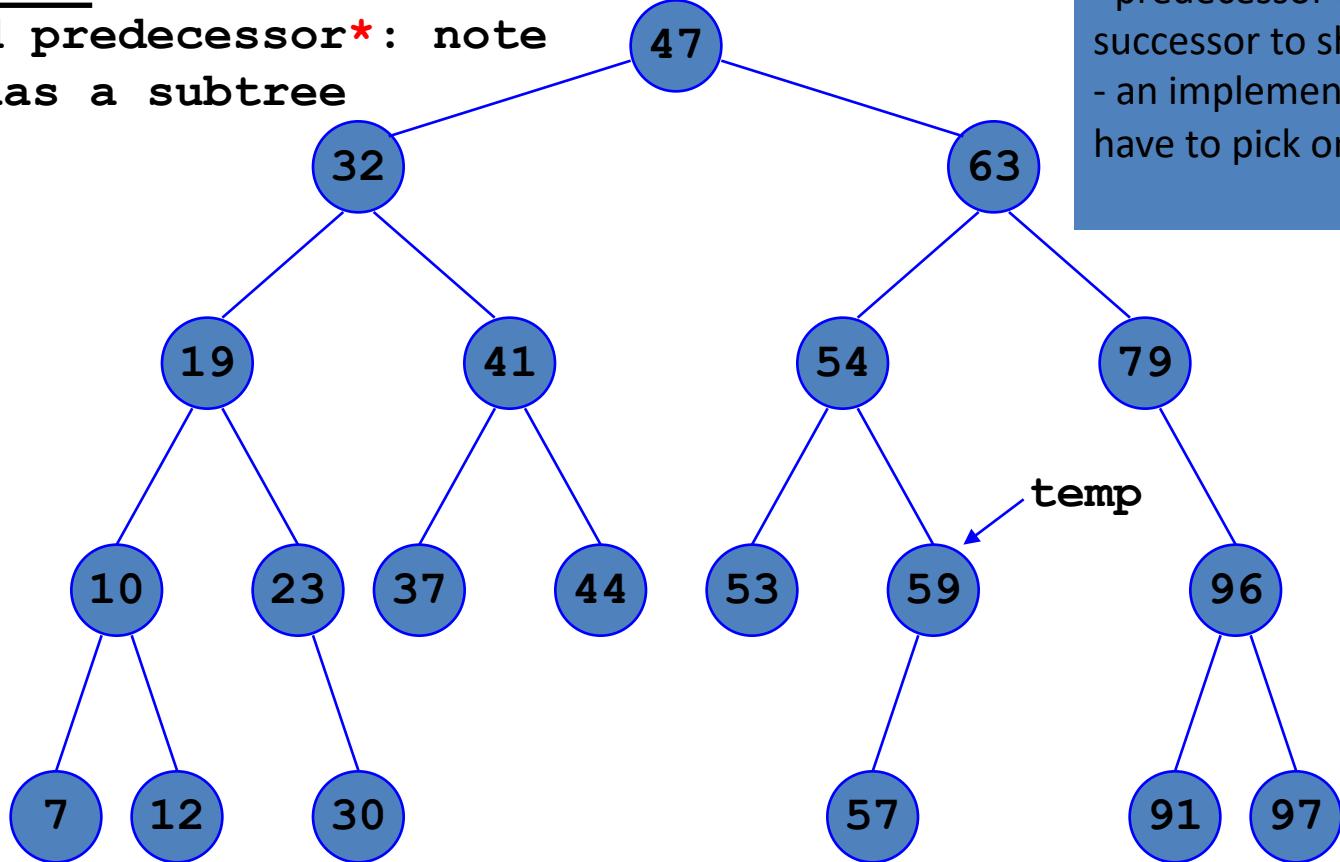
delete 32
note: successor
had no subtree



BST Deletion – target has 2 children

delete 63

- find predecessor*: note
it has a subtree

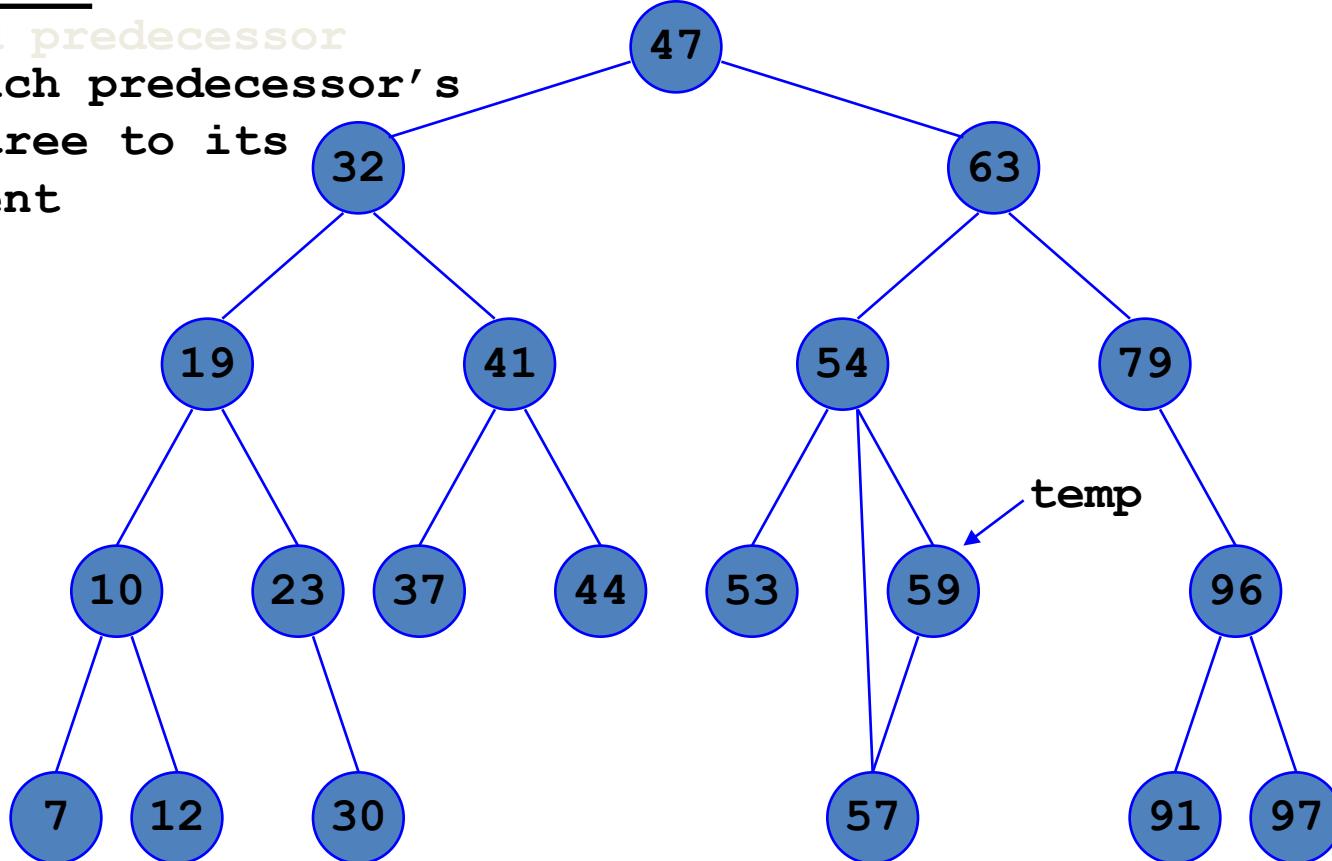


*predecessor used instead of successor to show its location
- an implementation would have to pick one or the other

BST Deletion – target has 2 children

delete 63

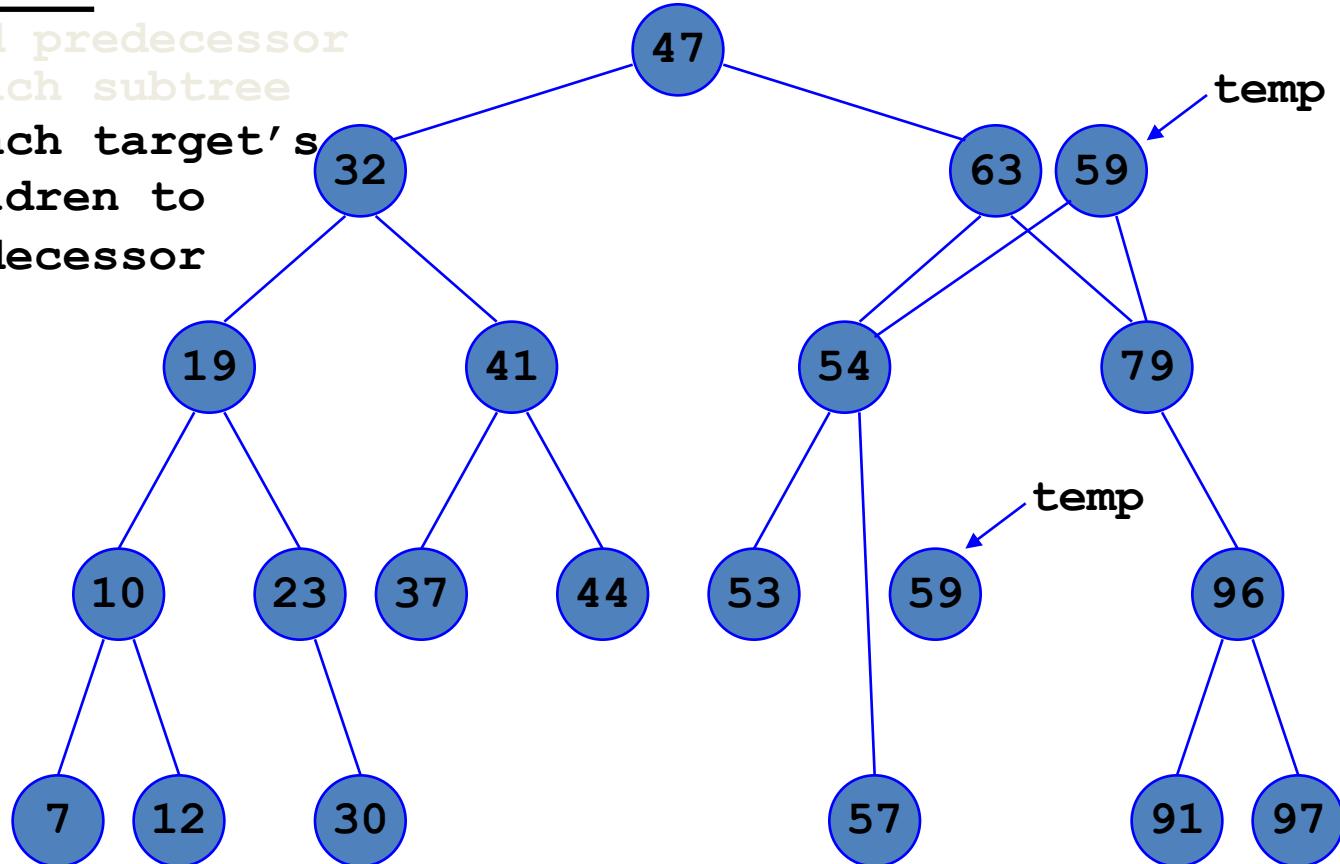
- find predecessor
- attach predecessor's subtree to its parent



BST Deletion – target has 2 children

delete 63

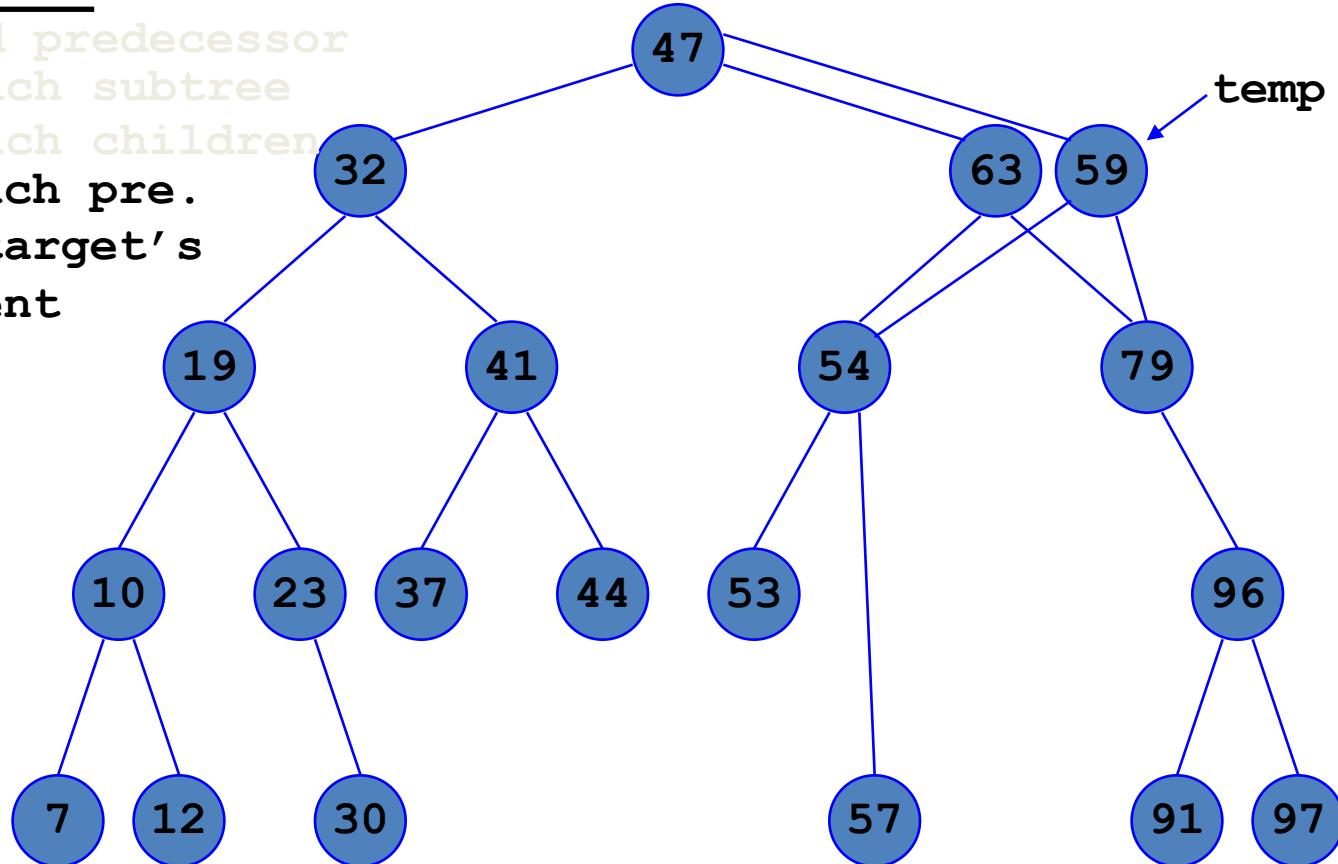
- find predecessor
- attach subtree
- attach target's children to predecessor



BST Deletion – target has 2 children

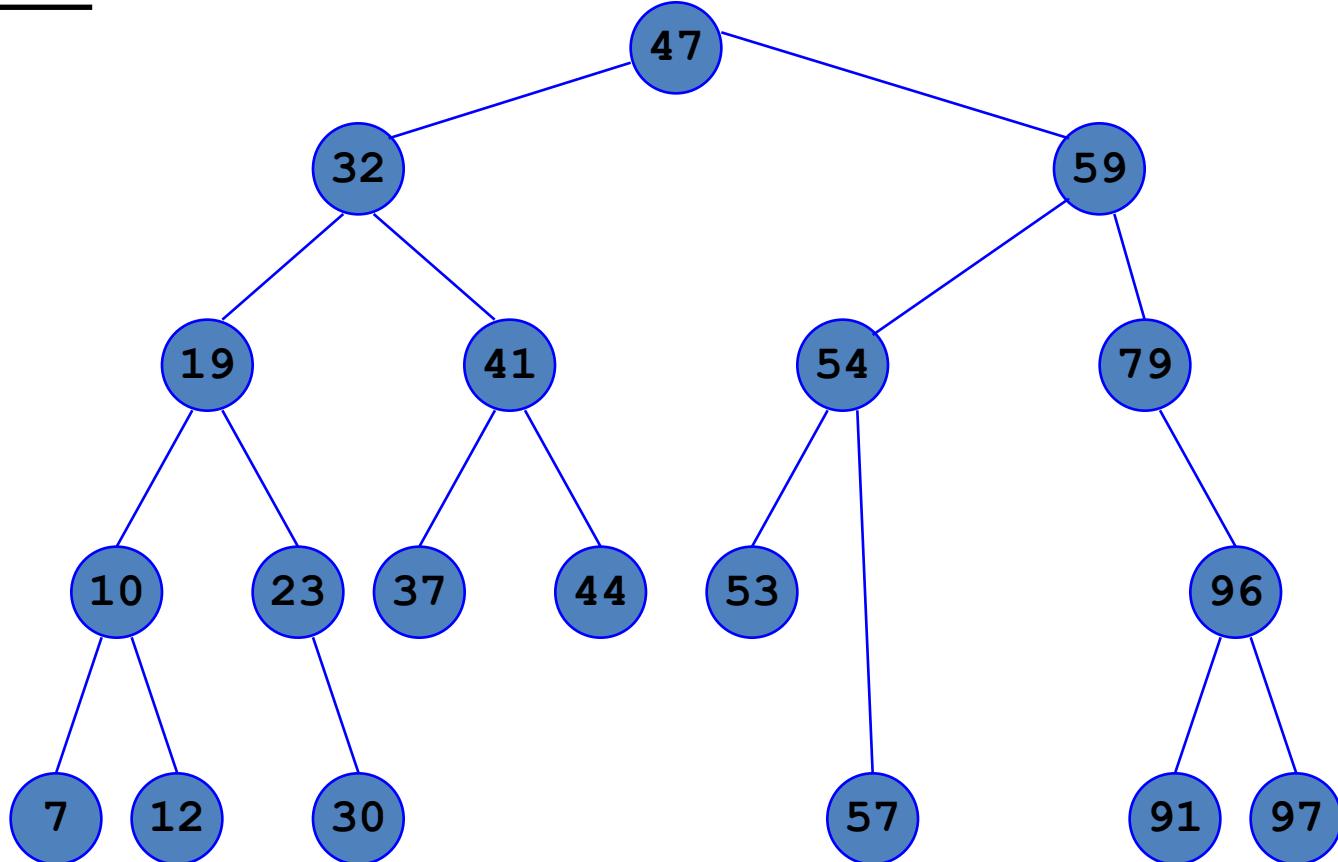
delete 63

- find predecessor
- attach subtree
- attach children
- attach pre.
to target's
parent



BST Deletion – target has 2 children

delete 63

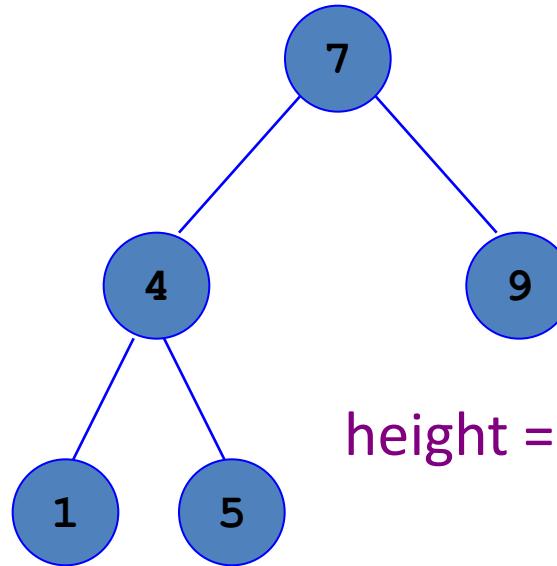


BST Efficiency

- The efficiency of BST operations depends on the **height** of the tree
- All three operations (search, insert and delete) are **$O(\text{height})$**
- If the tree is complete the height is $\lfloor \log(n) \rfloor$
- What if it isn't complete?

Height of a BST

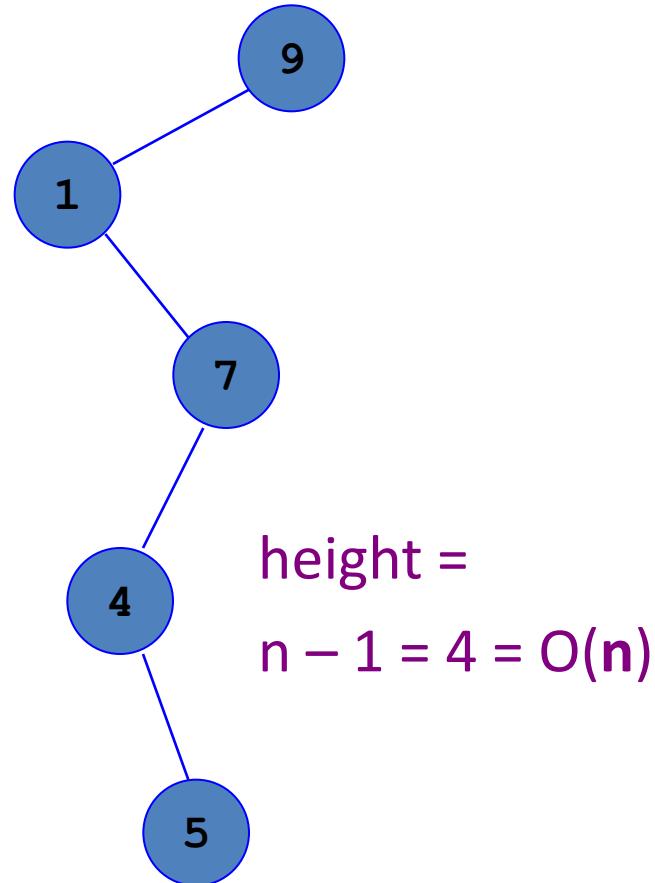
- Insert 7
- Insert 4
- Insert 1
- Insert 9
- Insert 5
- It's a complete tree!



$$\text{height} = \lfloor \log(5) \rfloor = 2$$

Height of a BST

- Insert 9
- Insert 1
- Insert 7
- Insert 4
- Insert 5
- It's a linked list with a lot of extra pointers!



Balanced BSTs

- It would be ideal if a BST was always close to complete
 - i.e. balanced
- How do we guarantee a balanced BST?
 - We have to make the insertion and deletion algorithms more complex. E.g Red-Black or AVL trees

Sorting and Binary Search Trees

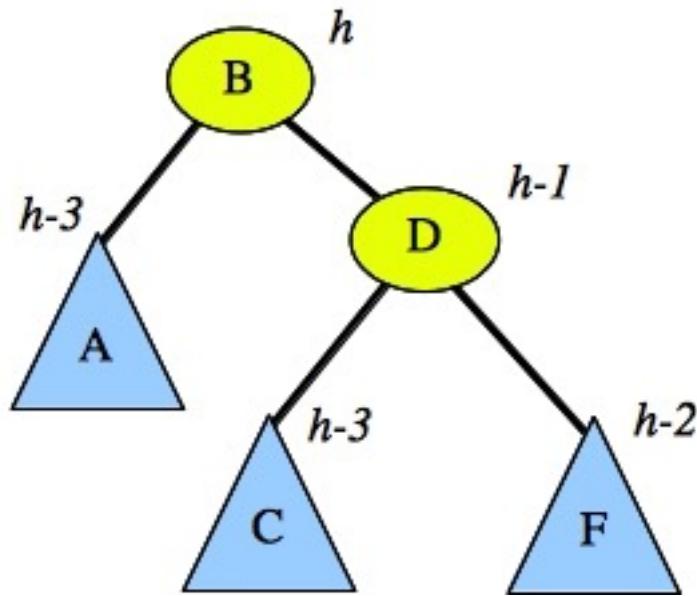
- It is possible to sort an array using a binary search tree
 - Insert the array items into an empty tree
 - Write the data from the tree back into the array using an InOrder traversal
- Running time = $n * (\text{insertion cost}) + \text{traversal}$
 - Insertion cost is $O(h)$
 - Traversal is $O(n)$
 - Total = $O(n) * O(h) + O(n)$, i.e. $O(n * h)$
 - If the tree is balanced = $O(n * \log(n))$

K-ary Tree

- If we know the maximum number of children each node will have, K, we can use an array of children references in each node.

```
class KTreeNode
{
    AnyType element;
    KTreeNode children[ K ];
}
```

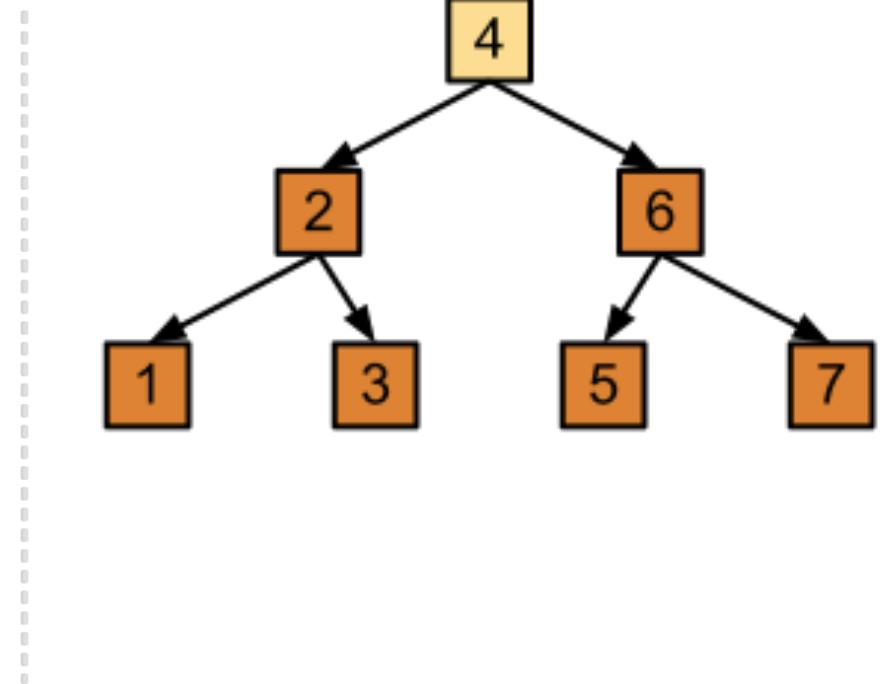
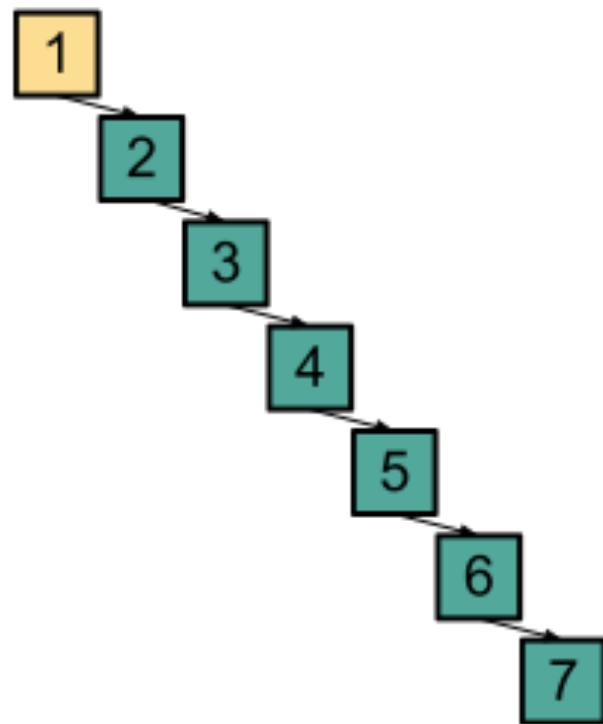
Balanced Binary Search Tree



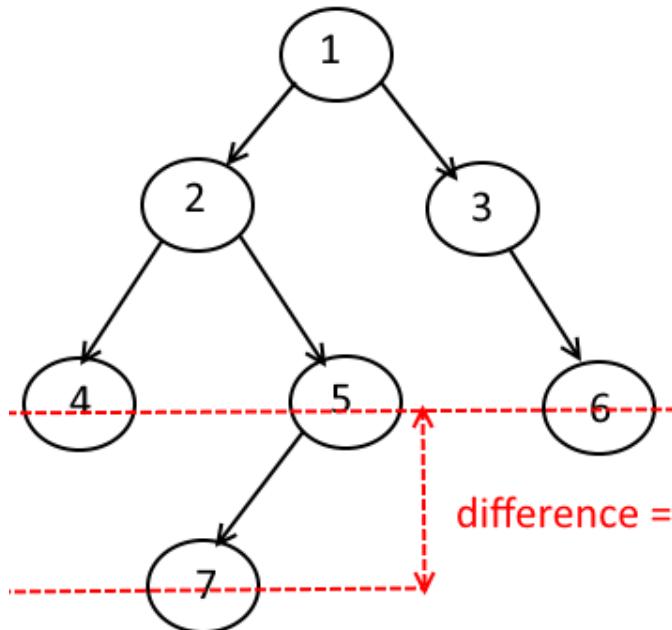
Slides partly from Data Structures and Algorithms in Java (Goodrich, et. al.), and Introduction to Algorithms (Leiserson, , et. al.)

Balanced vs. Non-Balanced

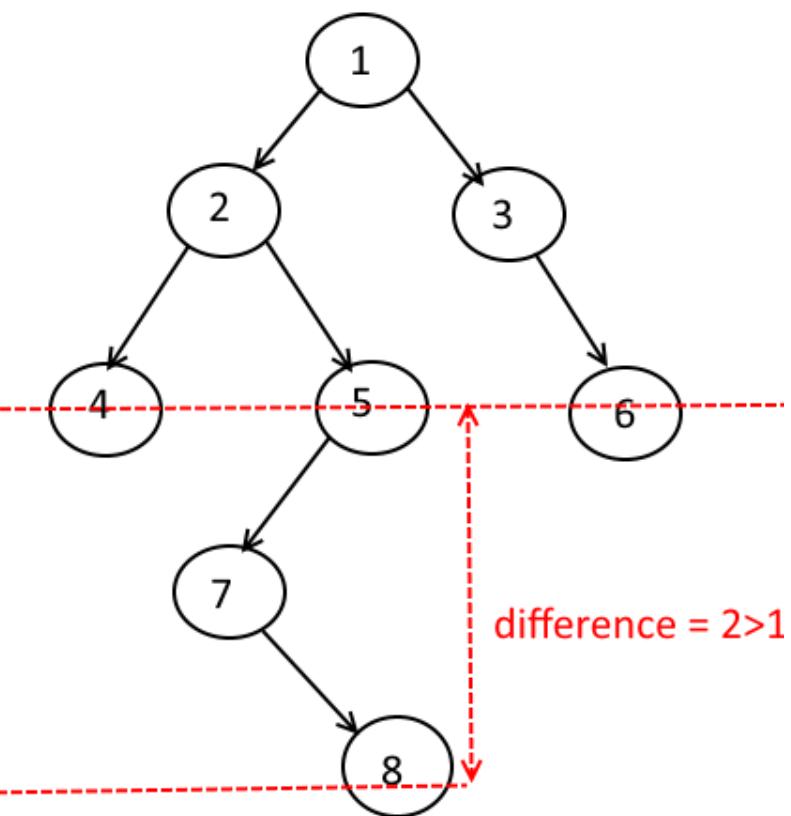
Built from the sequence [1,2,3,4,5,6,7]



Balanced vs. Non-Balanced



BALANCED



NOT BALANCED

Balanced BSTs

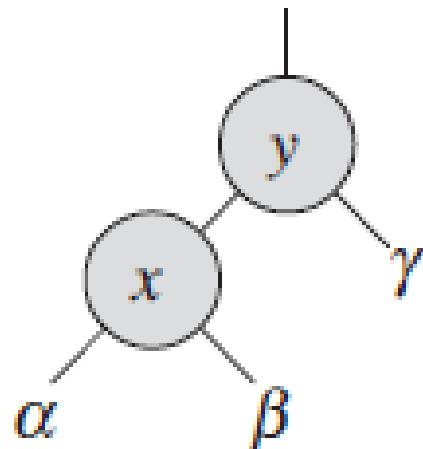
- It would be ideal if a BST was always close to complete
 - i.e. balanced
- How do we guarantee a balanced BST?
 - We have to make the insertion and deletion algorithms more complex. Popular data structures implementing this type of tree include AVL tree, Red-black tree, B-tree, and Splay tree.

Balancing Binary Search Trees

- Inserting or deleting a node from a (balanced) binary search tree can lead to an unbalance
- **Purpose:** To achieve a worst-case runtime of $O(\log n)$ for searching, inserting and deleting
- We perform some **Rotation** operations to rearrange the BST in a balanced form.

Tree Rotations

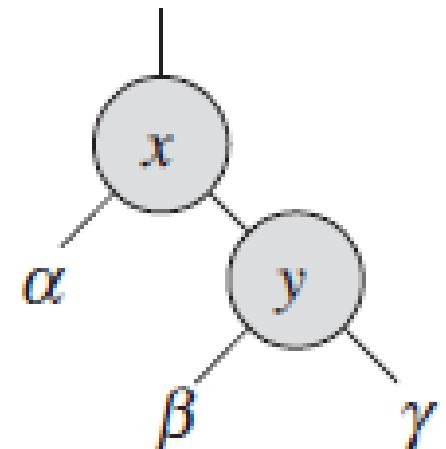
- The basic tree-restructuring operation
- There are left rotation and right rotation. They are inverses of each other
- A rotation operation preserves the BST property



LEFT-ROTATE(T, x)

.....

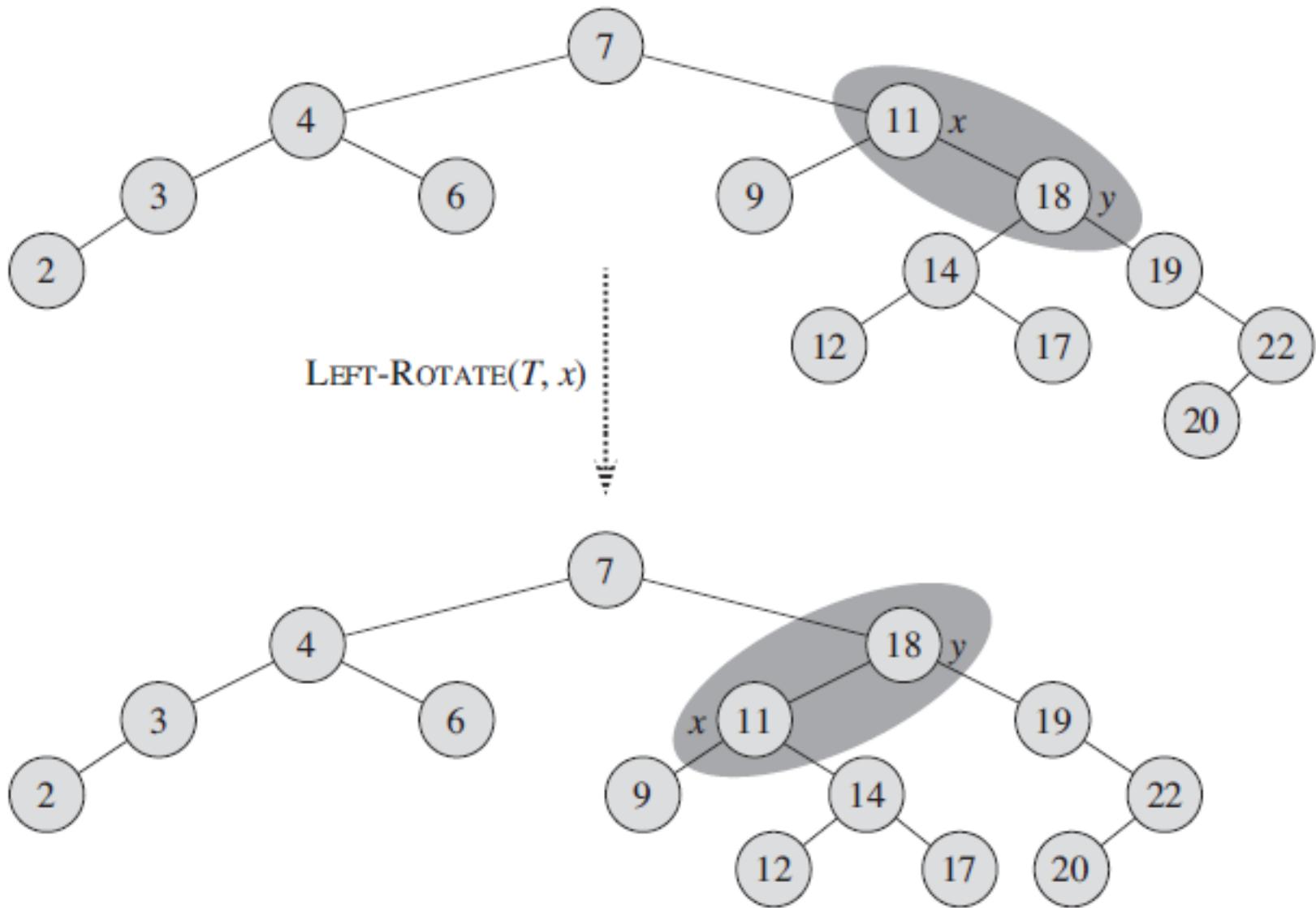
.....
RIGHT-ROTATE(T, y)



Implementing Rotations

LEFT-ROTATE(T, x)

```
1   $y = x.right$                                 // set  $y$ 
2   $x.right = y.left$                             // turn  $y$ 's left subtree into  $x$ 's right subtree
3  if  $y.left \neq T.nil$ 
4       $y.left.p = x$ 
5   $y.p = x.p$                                     // link  $x$ 's parent to  $y$ 
6  if  $x.p == T.nil$ 
7       $T.root = y$ 
8  elseif  $x == x.p.left$ 
9       $x.p.left = y$ 
10 else  $x.p.right = y$ 
11  $y.left = x$                                     // put  $x$  on  $y$ 's left
12  $x.p = y$ 
```



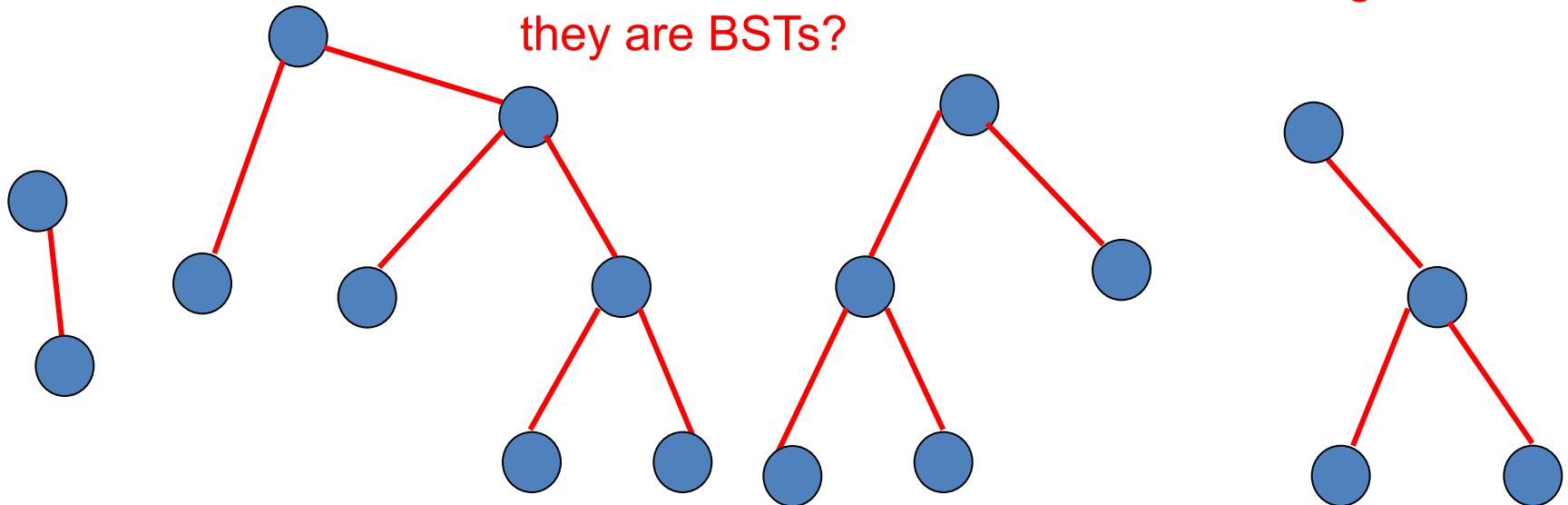
AVL Trees

Invented in 1962 by Adelson-Velski and Landis

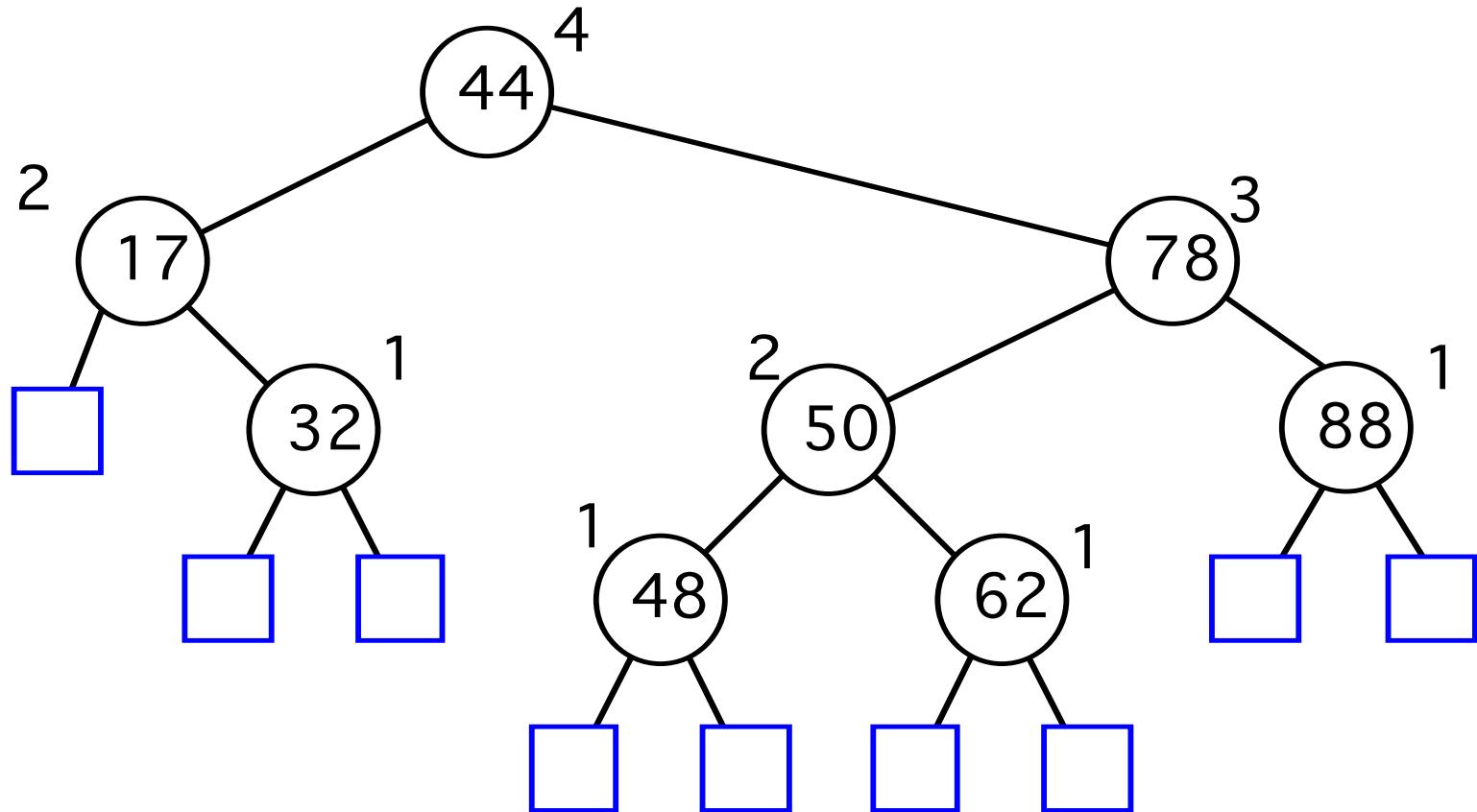
An AVL tree is a binary search tree such that:

- The height of the left and right sub-trees of the root differ by at most 1
- The left and right sub-trees are AVL trees

Which of these are AVL trees, assuming that they are BSTs?



AVL Tree

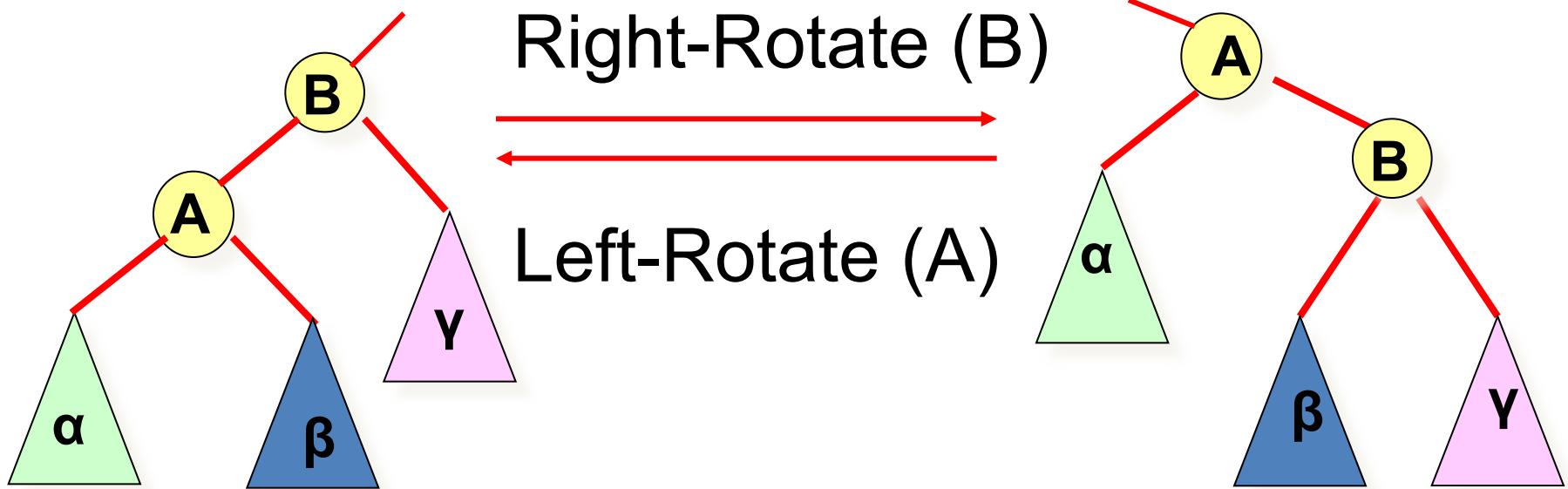


The heights are shown next to the nodes

AVL Trees

- AVL trees are height-balanced binary search trees
- Balance factor of a node
 - $\text{height}(\text{left subtree}) - \text{height}(\text{right subtree})$
- An AVL tree has balance factor calculated at every node
 - For every node, heights of left and right subtree can differ by no more than 1

Rotations

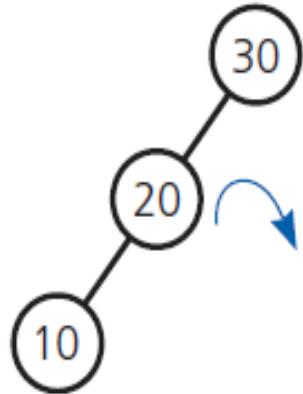


Rotations maintain the ordering property of BSTs.

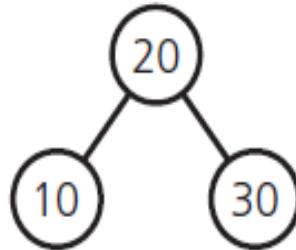
$a \in \alpha, b \in \beta, c \in \gamma$ implies $a \leq A \leq b \leq B \leq c$

A rotation is an O(1) operation

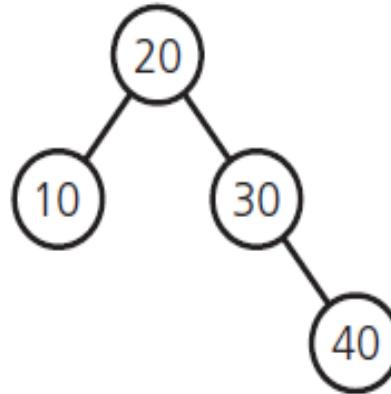
(a) Unbalanced



(b) Balanced due to rotation

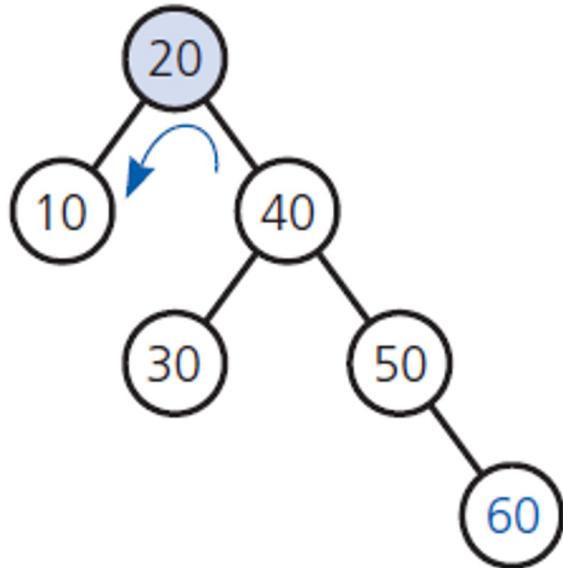


(c) Balanced after addition

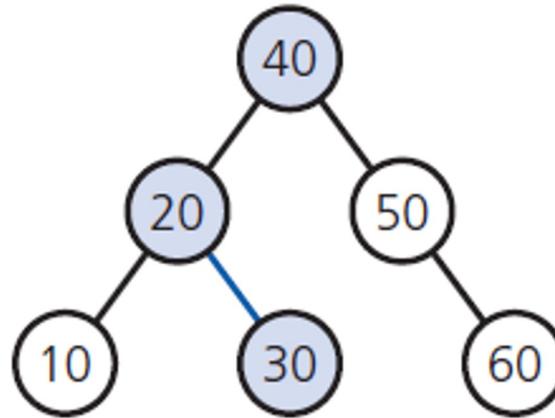


- An unbalanced binary search tree

(a) The addition of 60 to an AVL tree destroys its balance

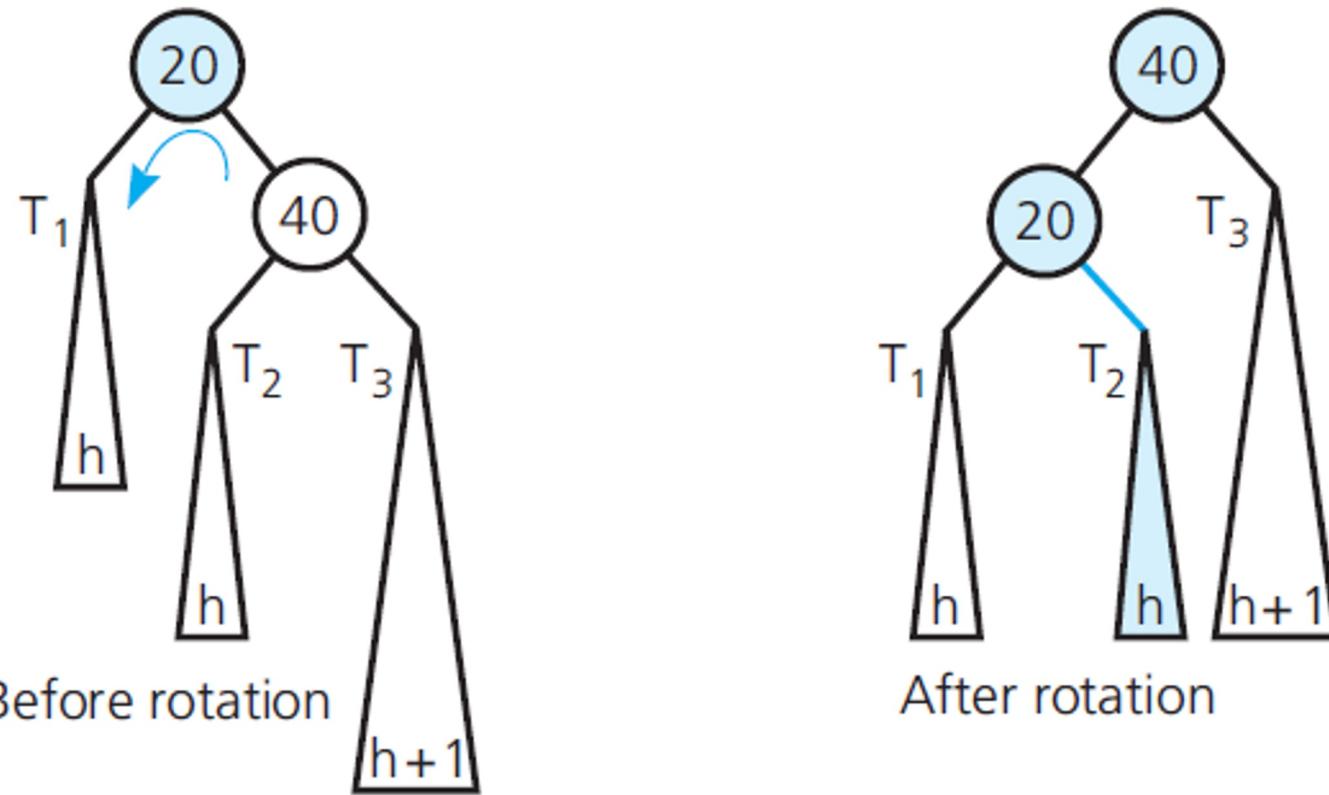


(b) A single left rotation restores the tree's balance



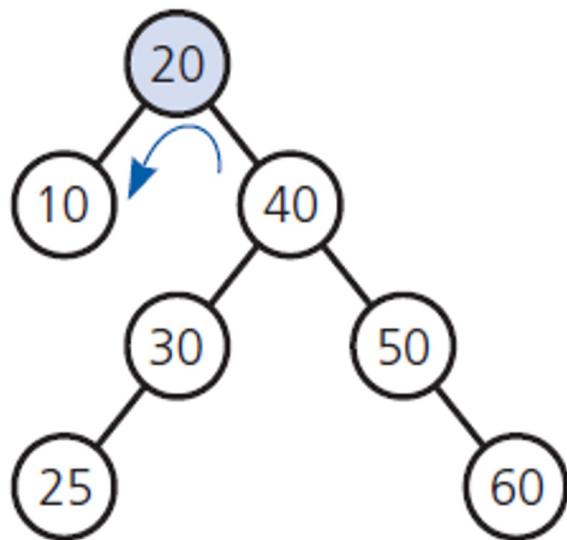
- Correcting an imbalance in an AVL tree due to an addition by using a single rotation to the left

(c) The general case for a single left rotation
in an AVL tree whose height decreases

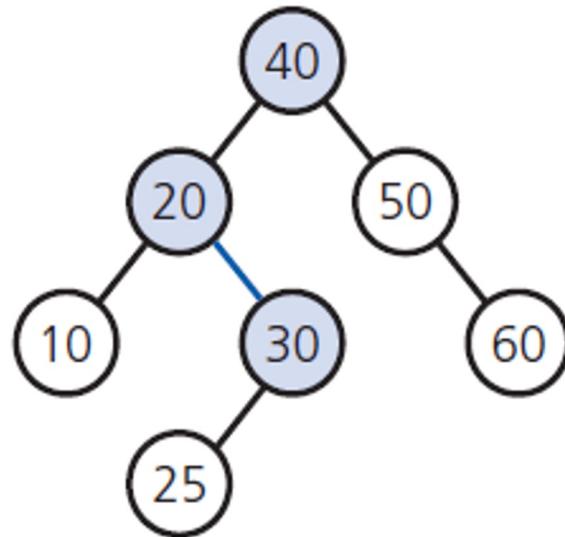


- Correcting an imbalance in an AVL tree due to an addition by using a **single rotation** to the left

(a) Unbalanced

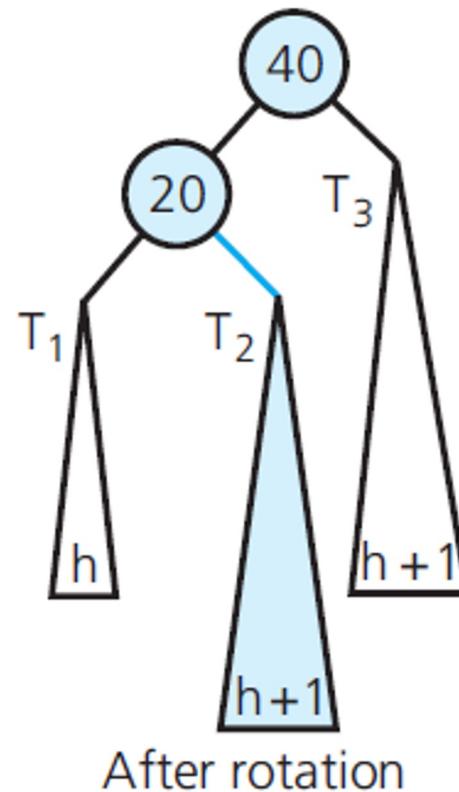
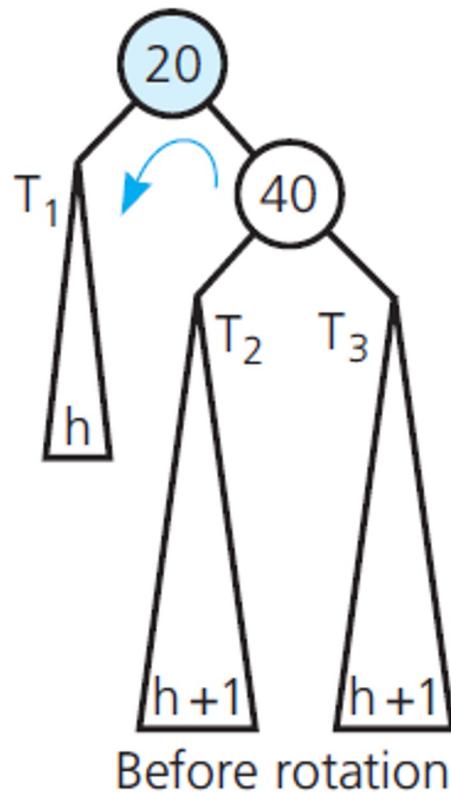


(b) After a single left rotation that restores the tree's balance



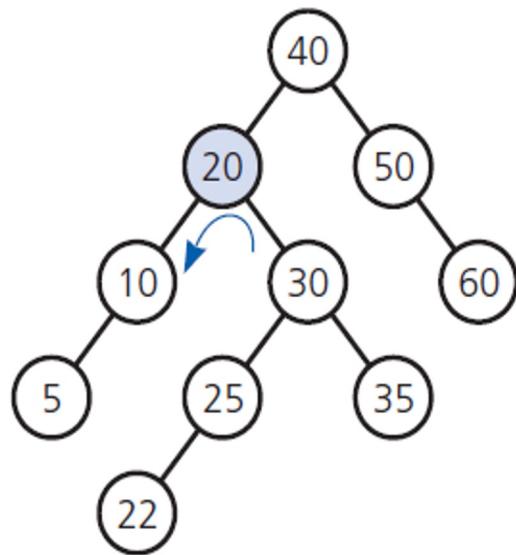
- A single rotation to the left that does not affect the height of an AVL tree

(c) The general case for a single left rotation in an AVL tree whose height is unchanged

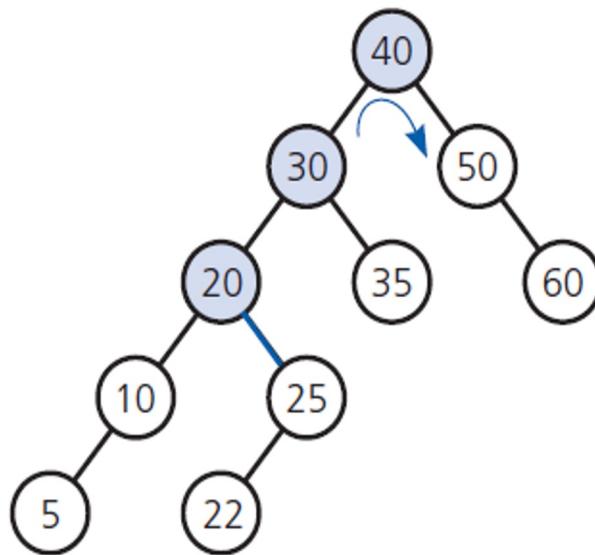


- A single rotation to the left that does not affect the height of an AVL tree

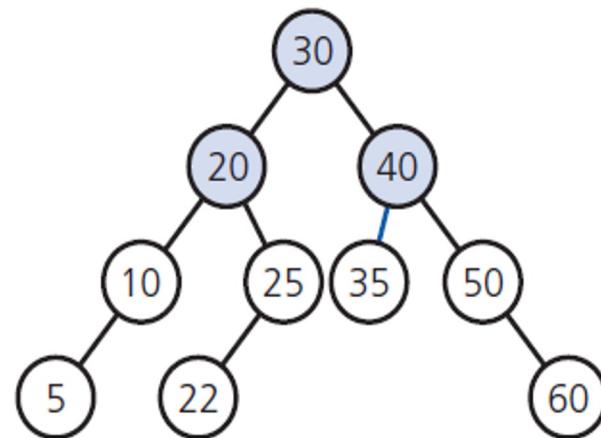
(a) Before the rotation



(b) After a left rotation

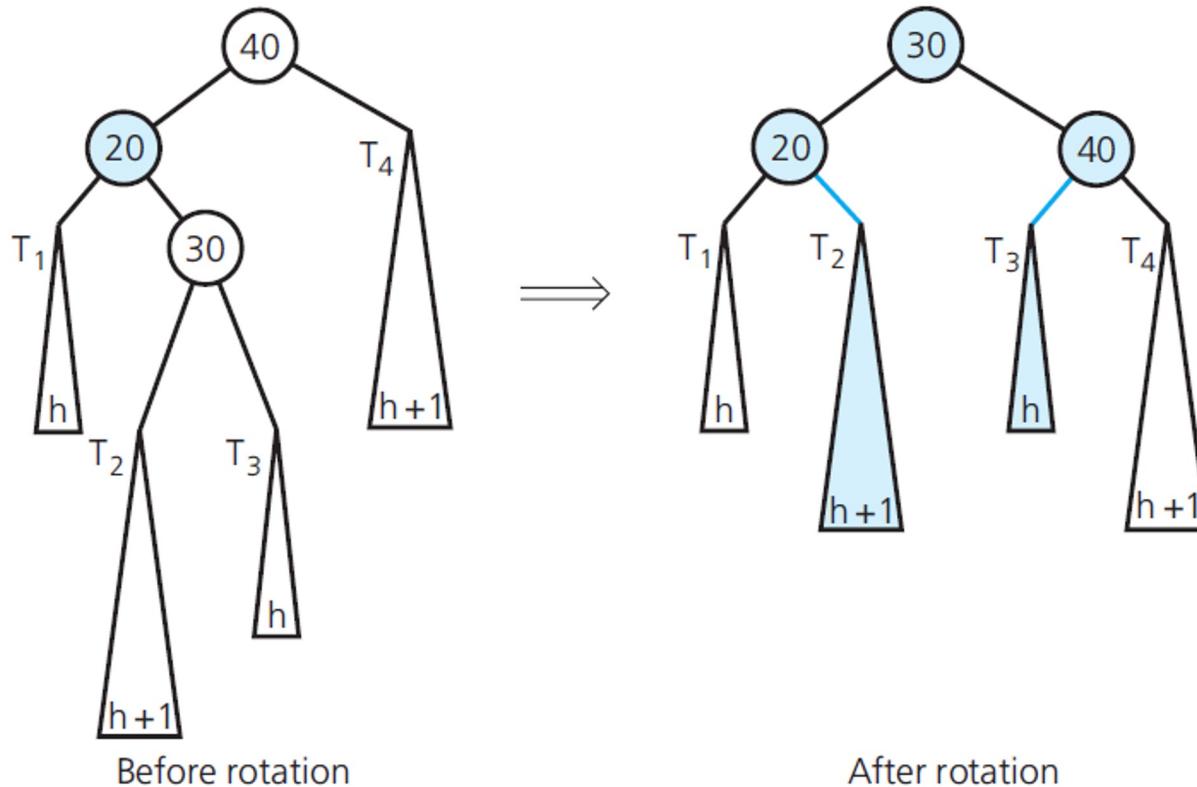


(c) After the right rotation



- A **double rotation** that decreases the height of an AVL tree

(d) The double rotation in general



- A double rotation that decreases the height of an AVL tree

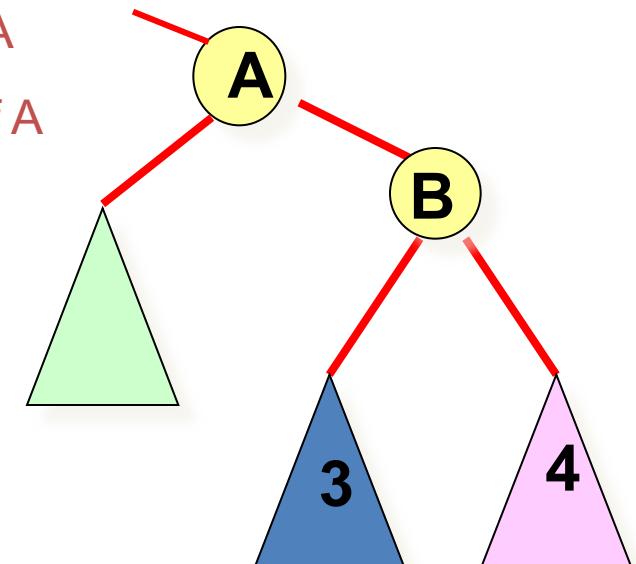
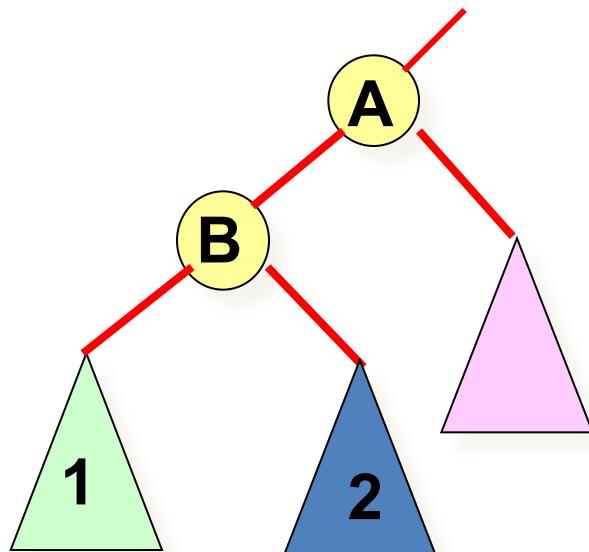
Rotations

- Since an insertion/deletion involves adding/deleting a single node, this can only increase/decrease the height of some subtree by 1
- Thus, if the AVL tree property is violated at a node x , it means that the heights of $\text{left}(x)$ and $\text{right}(x)$ differ by exactly 2.
- Rotations will be applied to x to restore the AVL tree property.

Insertions: 4 Cases

When inserting into a sub-tree of A, there are 4 cases in which a height violation could occur:

1. Inserting in the left sub-tree of the left child of A
2. Inserting in the right sub-tree of the left child of A
3. Inserting in the left sub-tree of the right child of A
4. Inserting in the right sub-tree of the right child of A

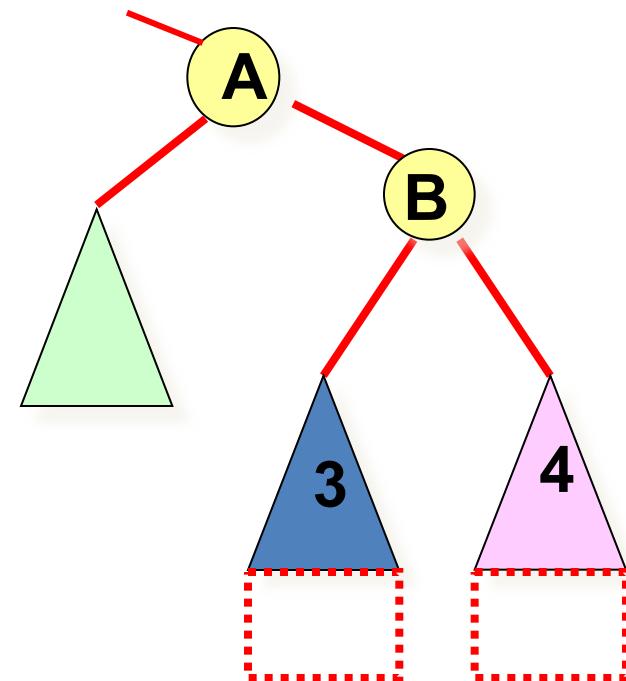
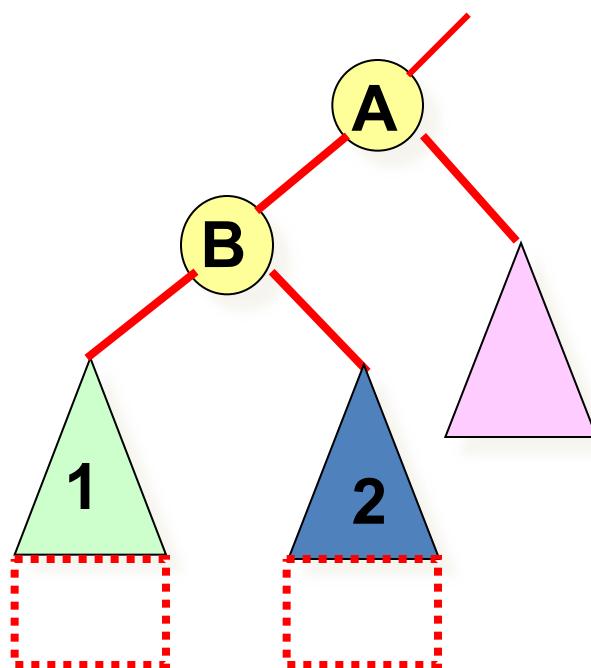


Rotations Required for the 4 Cases

Case 1: Requires a single right rotation to balance

Case 2 and 3: Require double rotations to balance

Case 4: Requires a single left rotation to balance

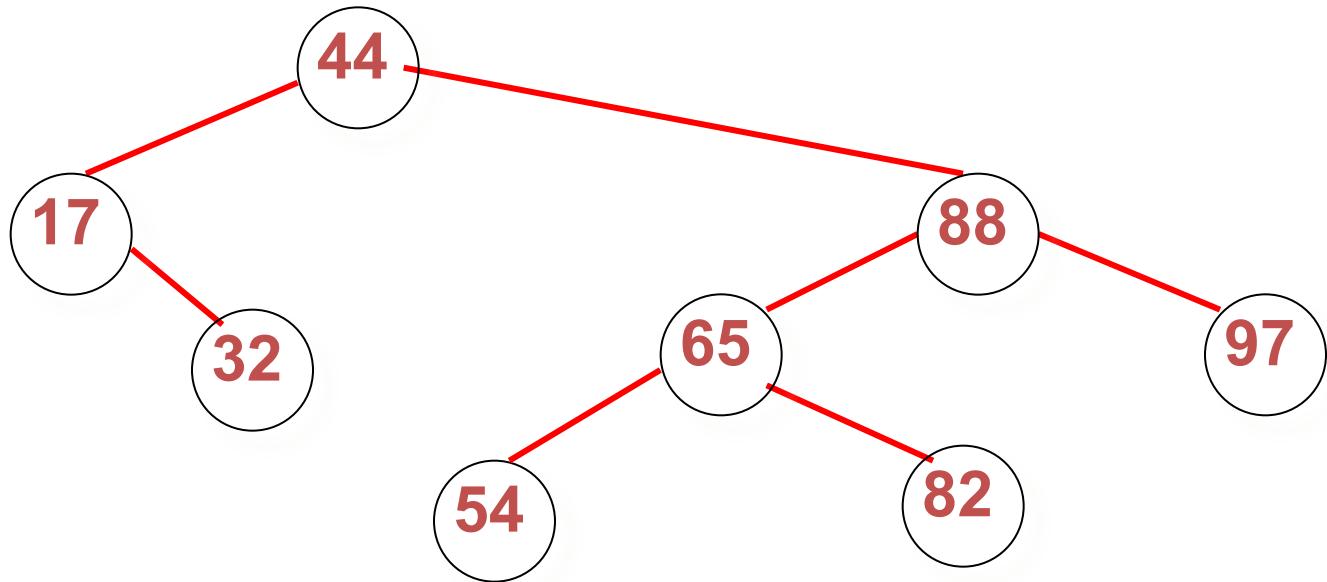


Insertion in an AVL Tree

- First insert node w in AVL tree T as for BST
- Then find the first node x going up from w to the root that is unbalanced (if none, are finished)
- Apply appropriate rotation (single or double), which reduces height of sub-tree rooted at x by 1

Since all nodes in T that became unbalanced were on the path of T from w to the root, restoring the sub-tree rooted at x to its original height rebalances the entire tree.

Insertion in an AVL Tree

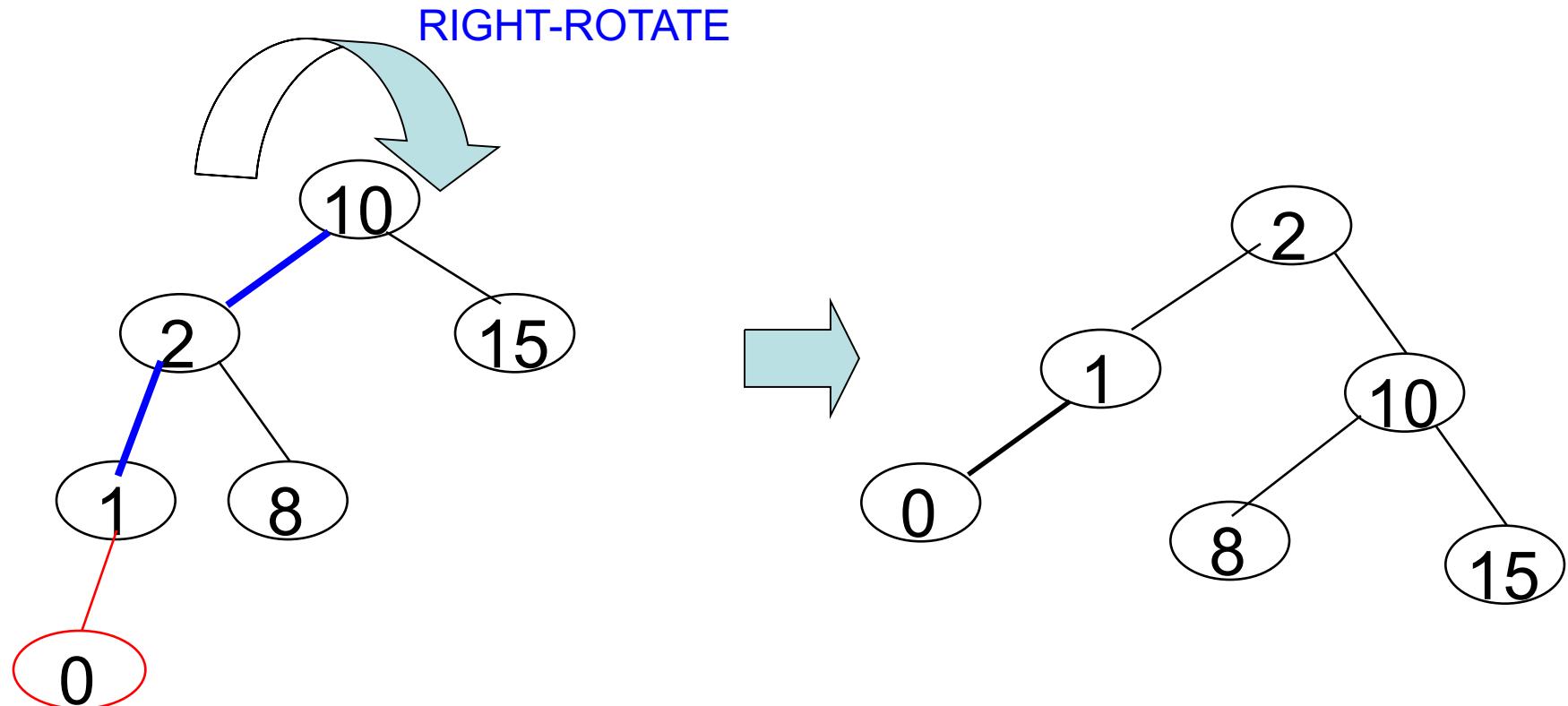


What insertion value would cause a Case 1 rotation? Case 2? 3? 4?

Insertion into an AVL trees

1. place a node into the appropriate place in BST
2. examine height balancing on insertion path:
 1. Tree was balanced ($\text{balance}=0$) => increasing the height of a subtree will be in the tolerated interval $+/-1$
 2. Tree was not balanced, with a factor $+/-1$, and the node is inserted in the *smaller* subtree leading to its height increase => the tree will be balanced after insertion
 3. Tree was balanced, with a factor $+/-1$, and the node is inserted in the *taller* subtree leading to its height increase => the tree is no longer height balanced (the heights of the left and right children of some node x might differ by 2)
 - we have to balance the subtree rooted at x using rotations

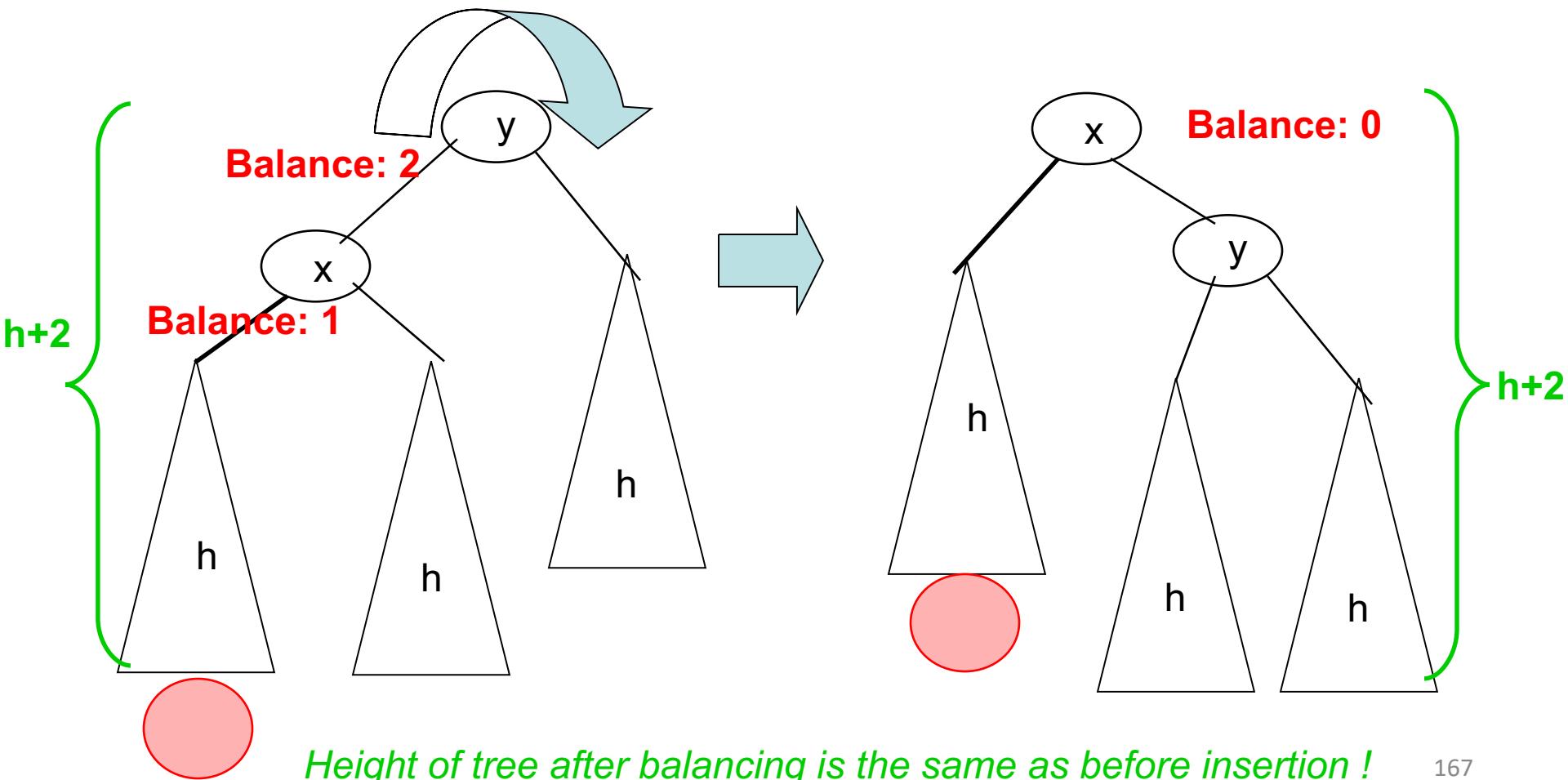
Example – AVL insertions



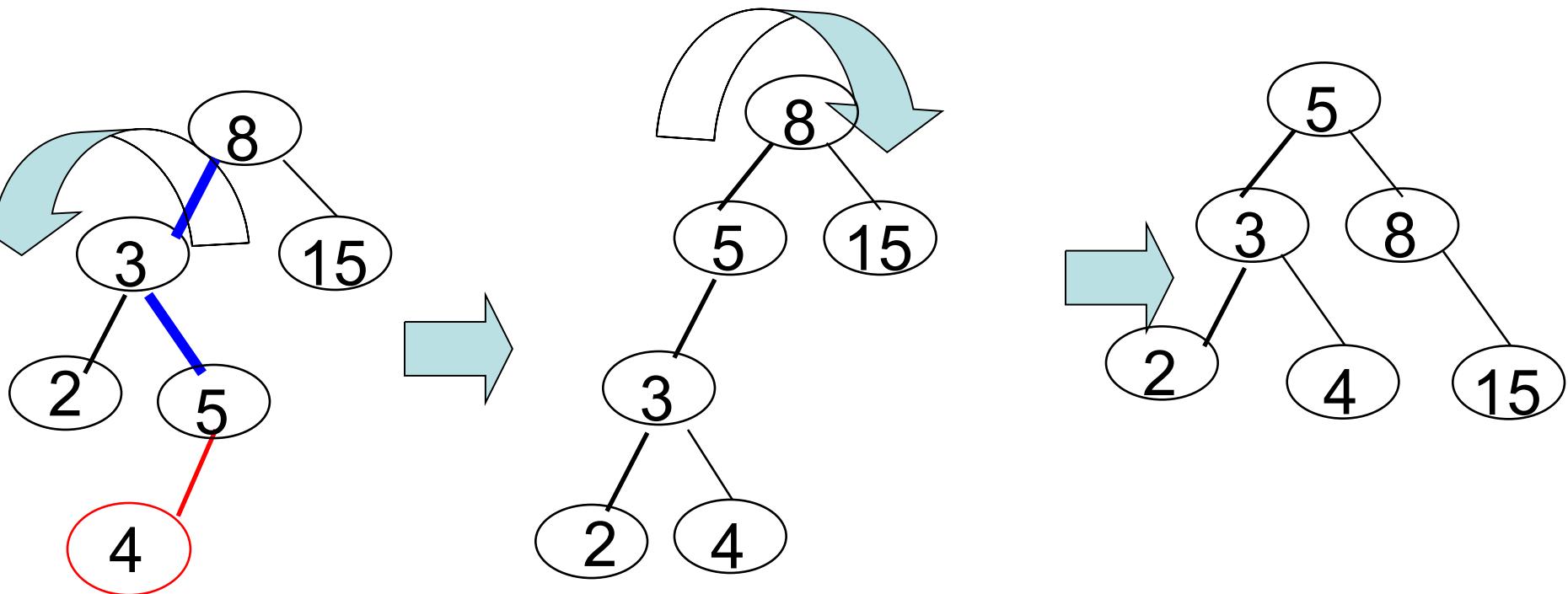
Case 1: Node's **Left – Left** grandchild is too tall

AVL insertions – Right Rotation

Case 1: Node's **Left – Left** grandchild is too tall



Example – AVL insertions

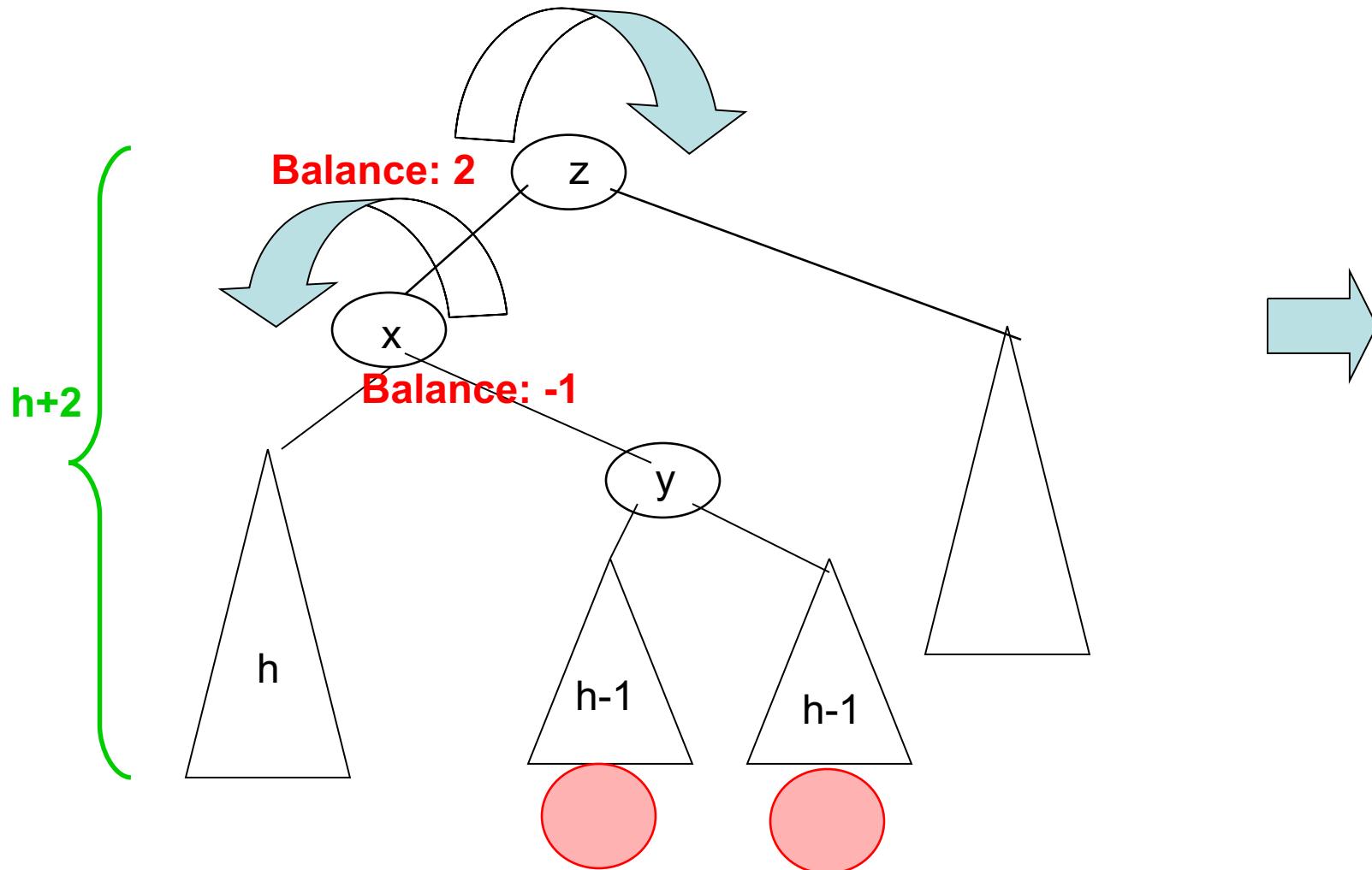


Solution: do a Double Rotation: LEFT-ROTATE and RIGHT-ROTATE

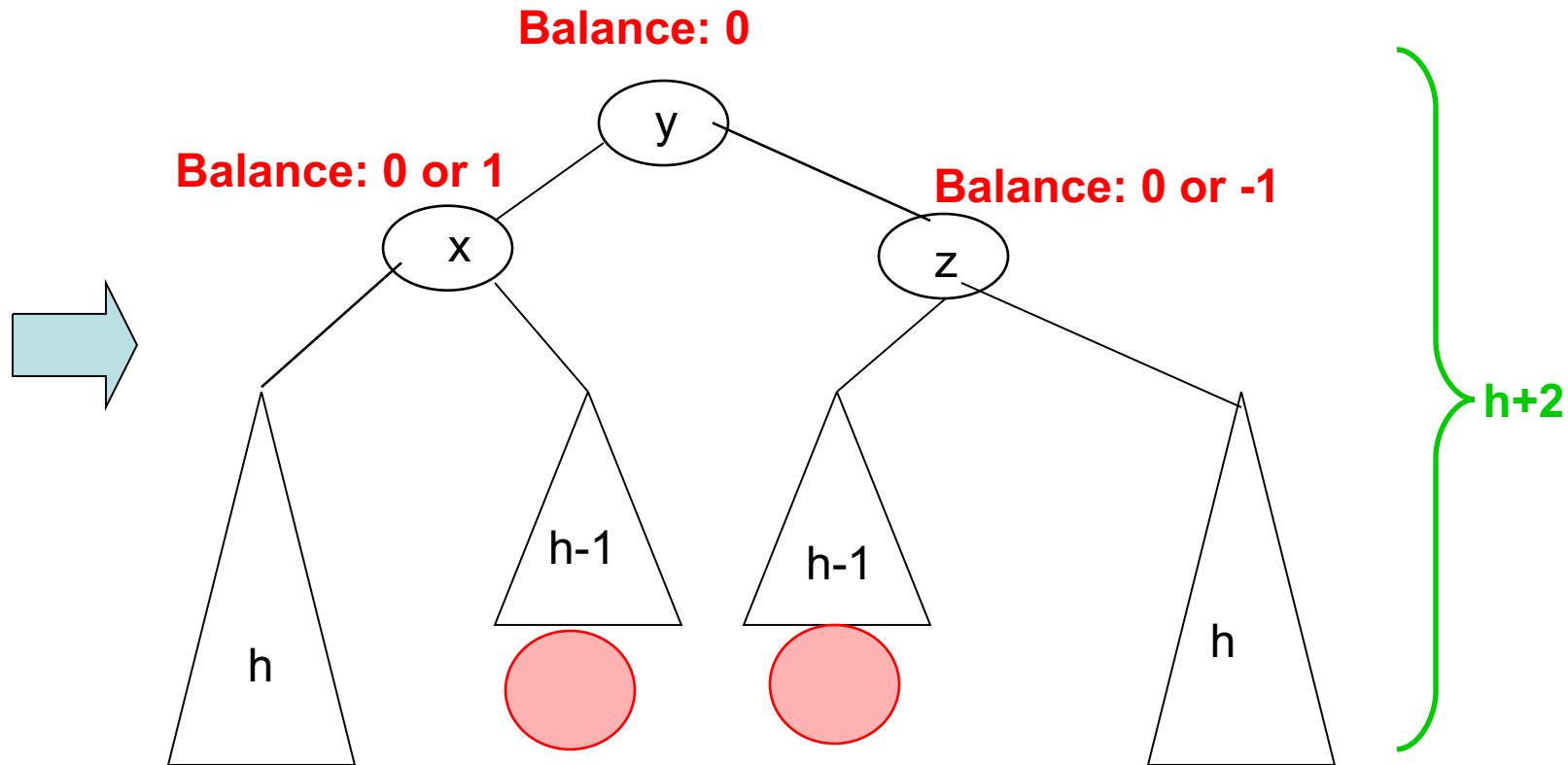
Case 2: Node's **Left-Right** grandchild is too tall

Double Rotation – Case Left-Right

Case 2: Node's **Left-Right** grandchild is too tall

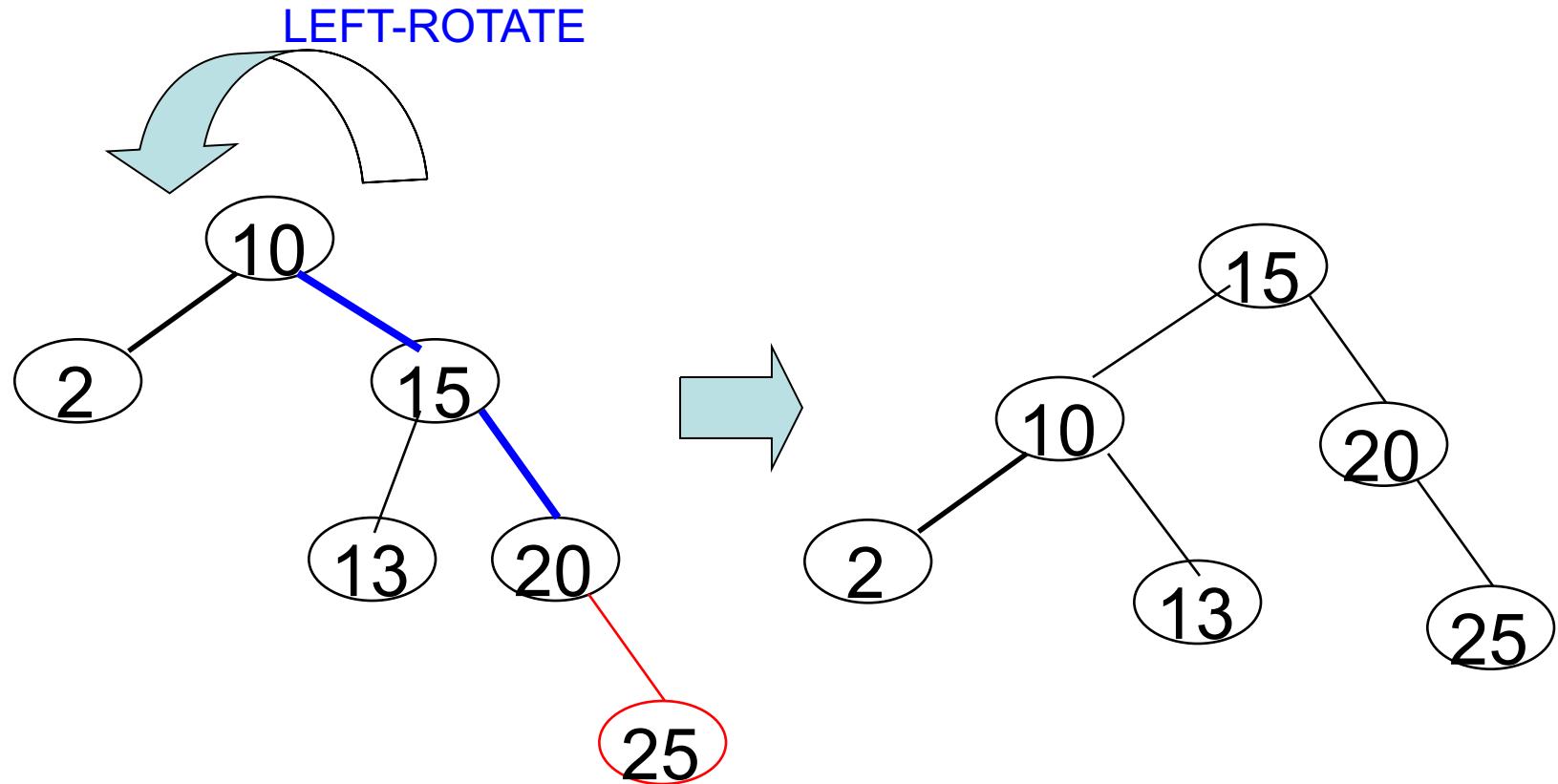


Double Rotation – Case Left-Right



*Height of tree after balancing is the same as before insertion !
=> there are NO upward propagations of the unbalance !*

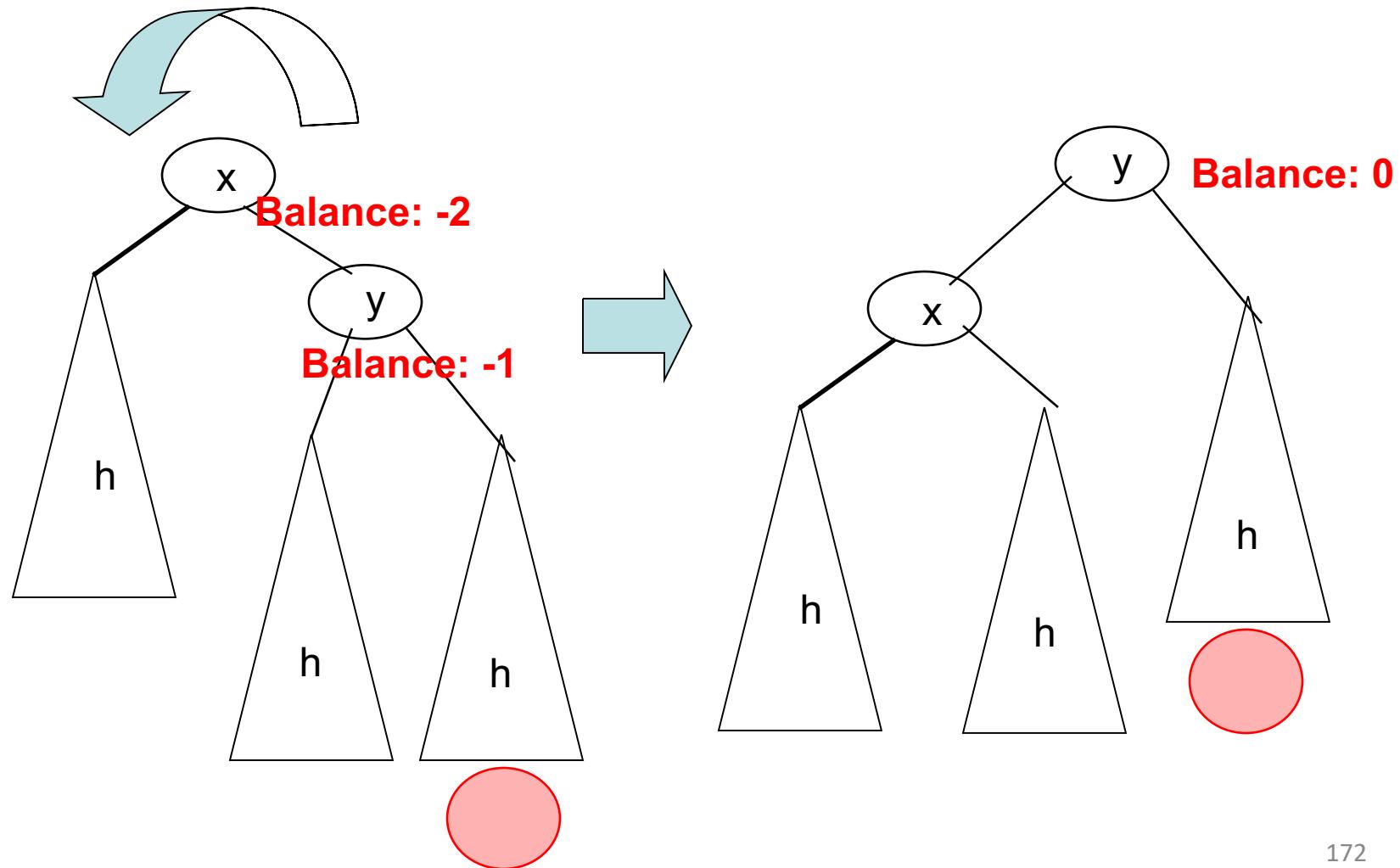
Example – AVL insertions



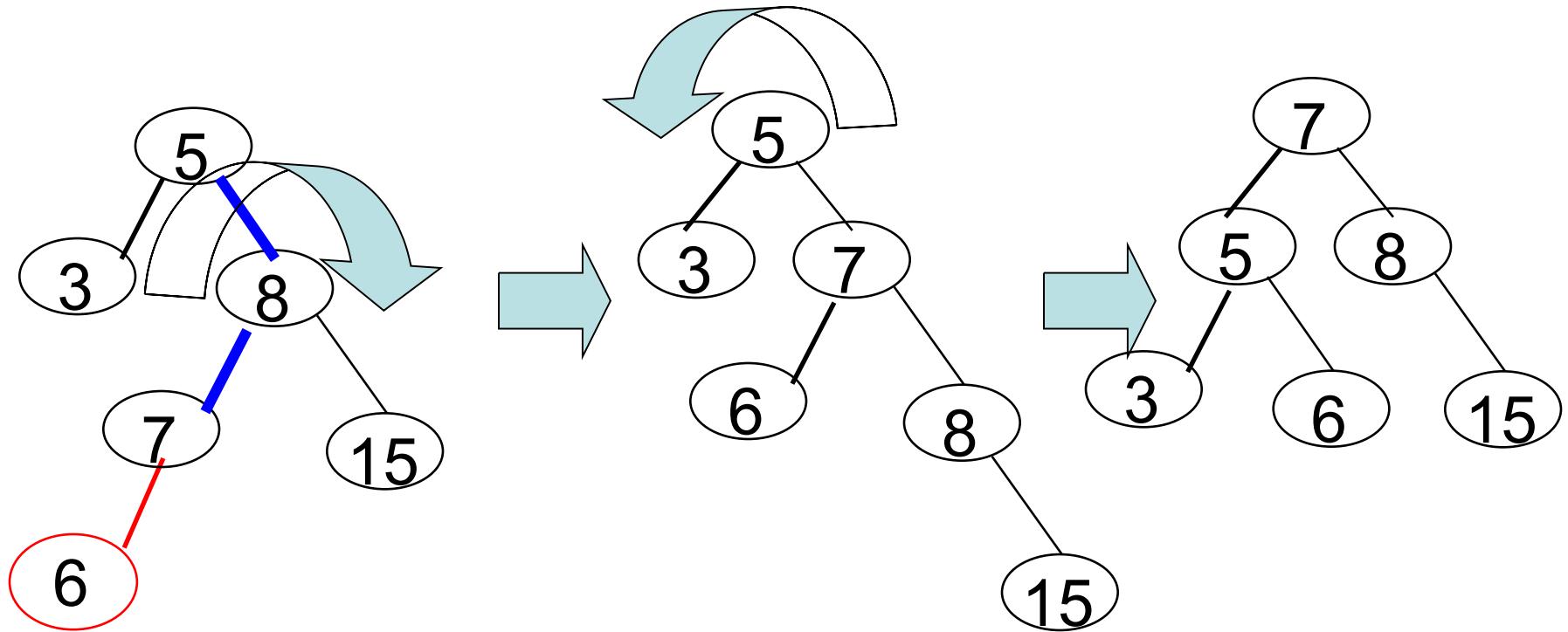
Case 3: Node's **Right – Right** grandchild is too tall

AVL insertions – Left Rotation

Case 3: Node's **Right – Right** grandchild is too tall



Example – AVL insertions

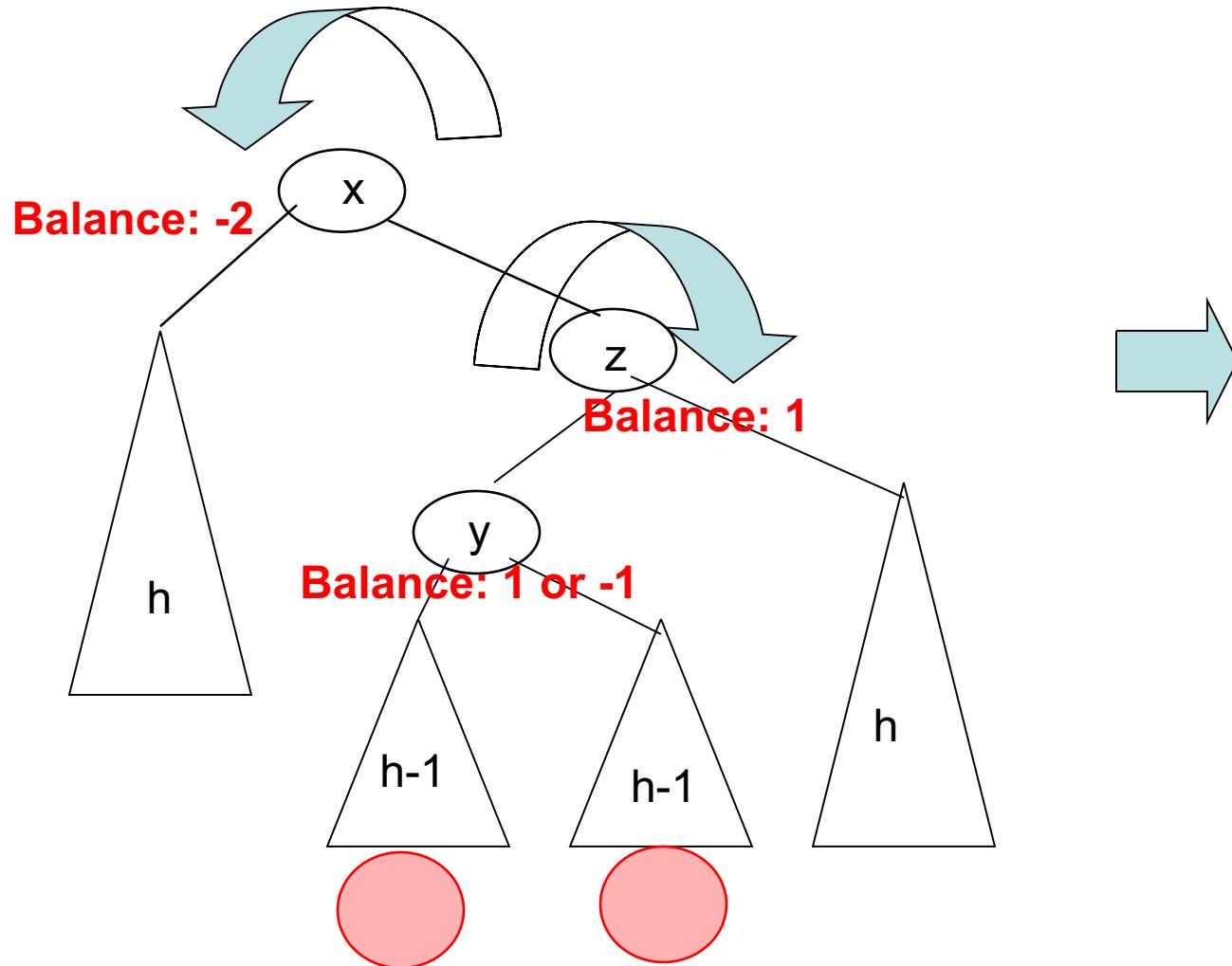


Solution: do a Double Rotation: RIGHT-ROTATE and LEFT-ROTATE

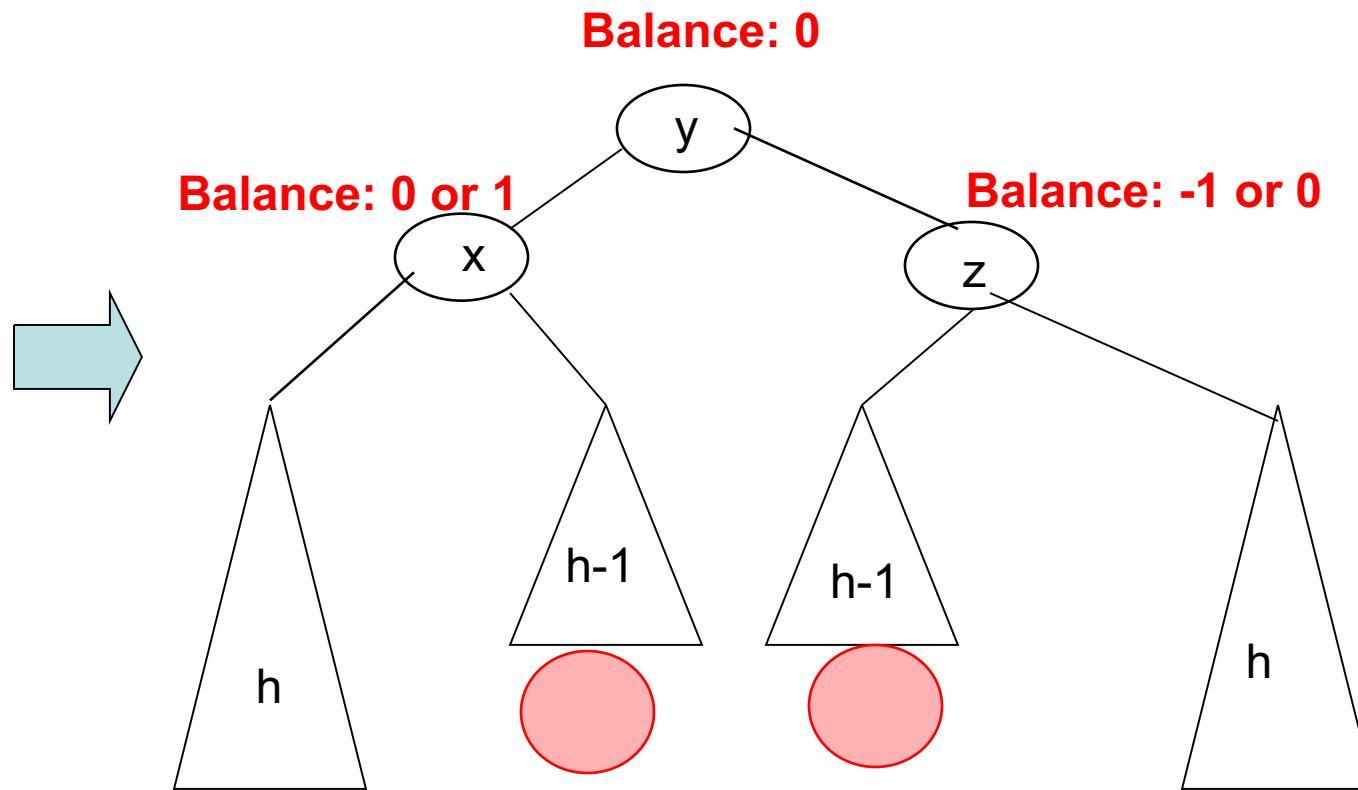
Case 4: Node's **Right – Left** grandchild is too tall

Double Rotation – Case Right-Left

Case 4: Node's **Right – Left** grandchild is too tall



Double Rotation – Case Right-Left



Implementing AVL Trees

- Insertion *needs* information about the height of each node
- It would be highly inefficient to calculate the height of a node every time this information is needed => *the tree structure is augmented with height information that is maintained during all operations*
- An AVL Node contains the attributes:
 - Key
 - *Left, right, p*
 - *Height*

Analysis of AVL-INSERT

- Insertion makes $O(h)$ steps, h is $O(\log n)$, thus Insertion makes $O(\log n)$ steps
- At every insertion step, there is a call to Balance, but *rotations will be performed only once for the insertion of a key*. It is not possible that after doing a balancing, unbalances are propagated , because the BALANCE operation restores the height of the subtree before insertion. => number of rotations for one insertion is $O(1)$
- AVL-INSERT is $O(\log n)$

AVL Delete

- The procedure of BST deletion of a node z:
 - 1 child: delete it, connect child to parent
 - 2 children: put successor in place of z, delete successor
- Which nodes' heights may have changed:
 - 1 child: path from deleted node to root
 - 2 children: path from deleted successor leaf to root
- AVL Tree may need rebalancing as we return along the deletion path back to the root

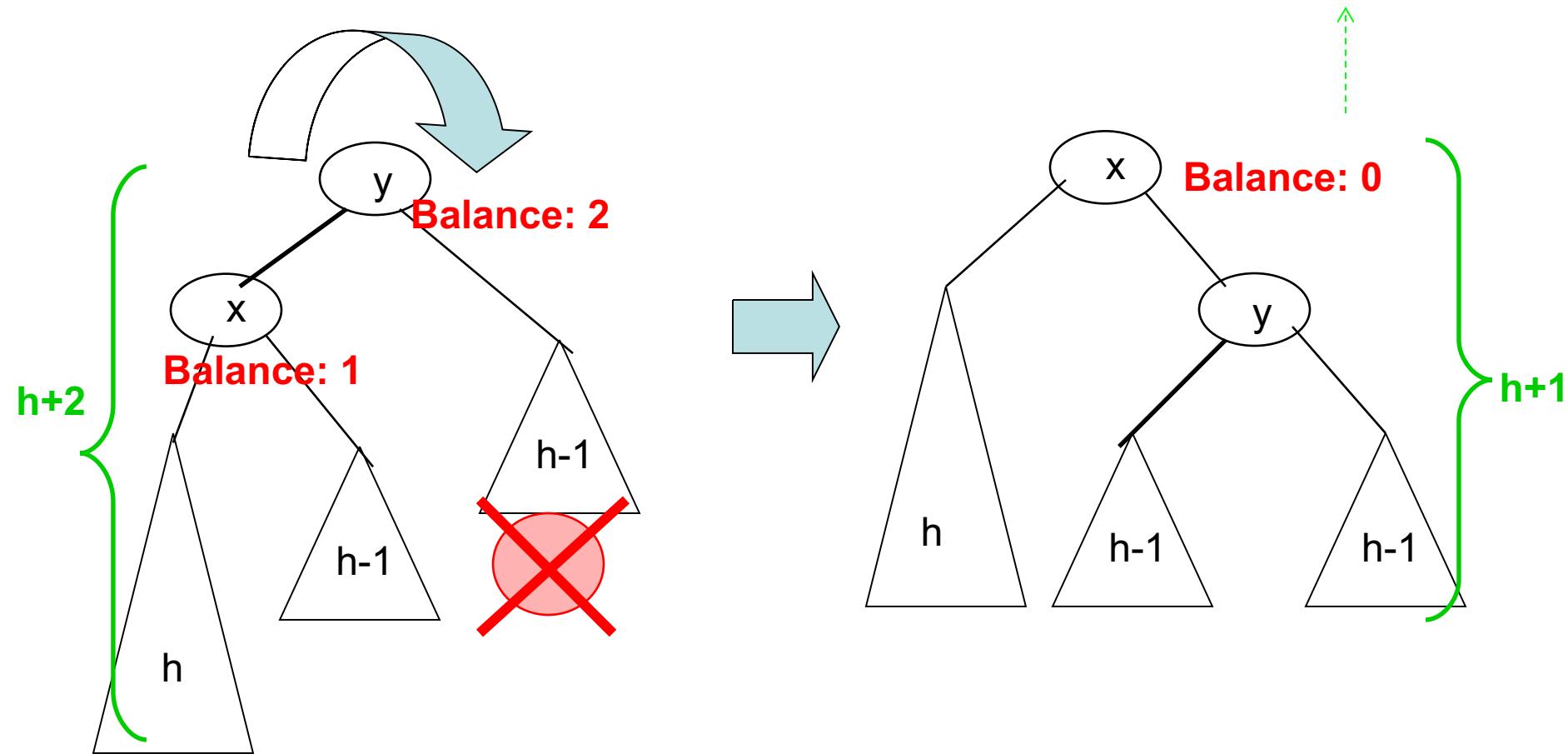
Deletion in an AVL Tree

- First delete node w in AVL tree T as in BST
- Then find the first node x going up from w to the root that is unbalanced (if none, are finished)
- Apply appropriate rotation (single or double), which results either in the sub-tree rooted at x being its original height before the deletion, or in its height being decreased by 1.

Balancing the sub-tree rooted at x may NOT rebalance the entire tree. $O(\log n)$ rotations may be required.

AVL delete – Right Rotation

Case 1: Node's **Left-Left** grandchild is too tall

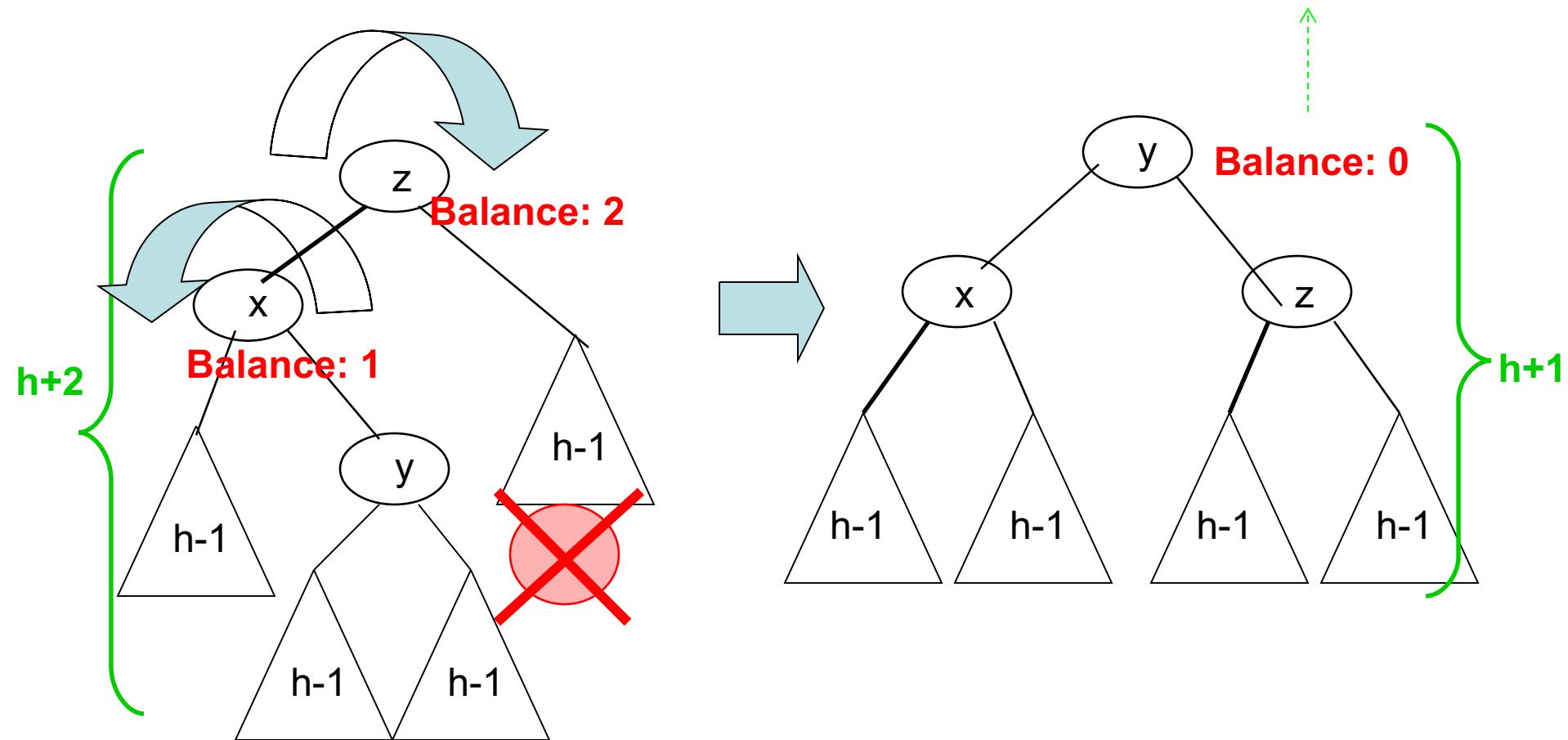


Delete node in right child, the height of the right child decreases

The height of tree after balancing decreases => Unbalance may propagate

AVL delete – Double Rotation

Case 2: Node's **Left-Right** grandchild is too tall

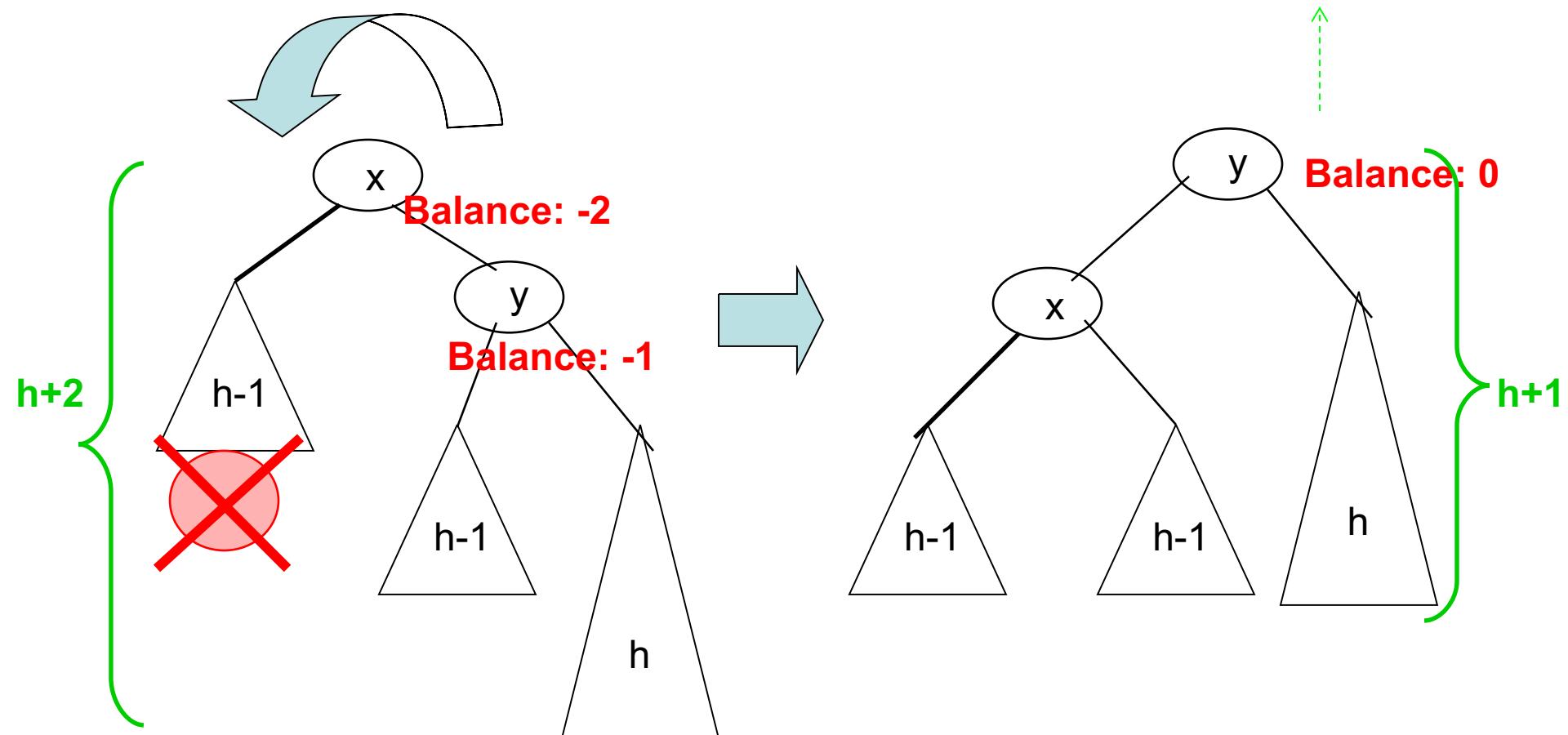


Delete node in right child, the height of the right child decreases

The height of tree after balancing decreases !=> Unbalance may propagate

AVL delete – Left Rotation

Case 3: Node's **Right – Right** grandchild is too tall

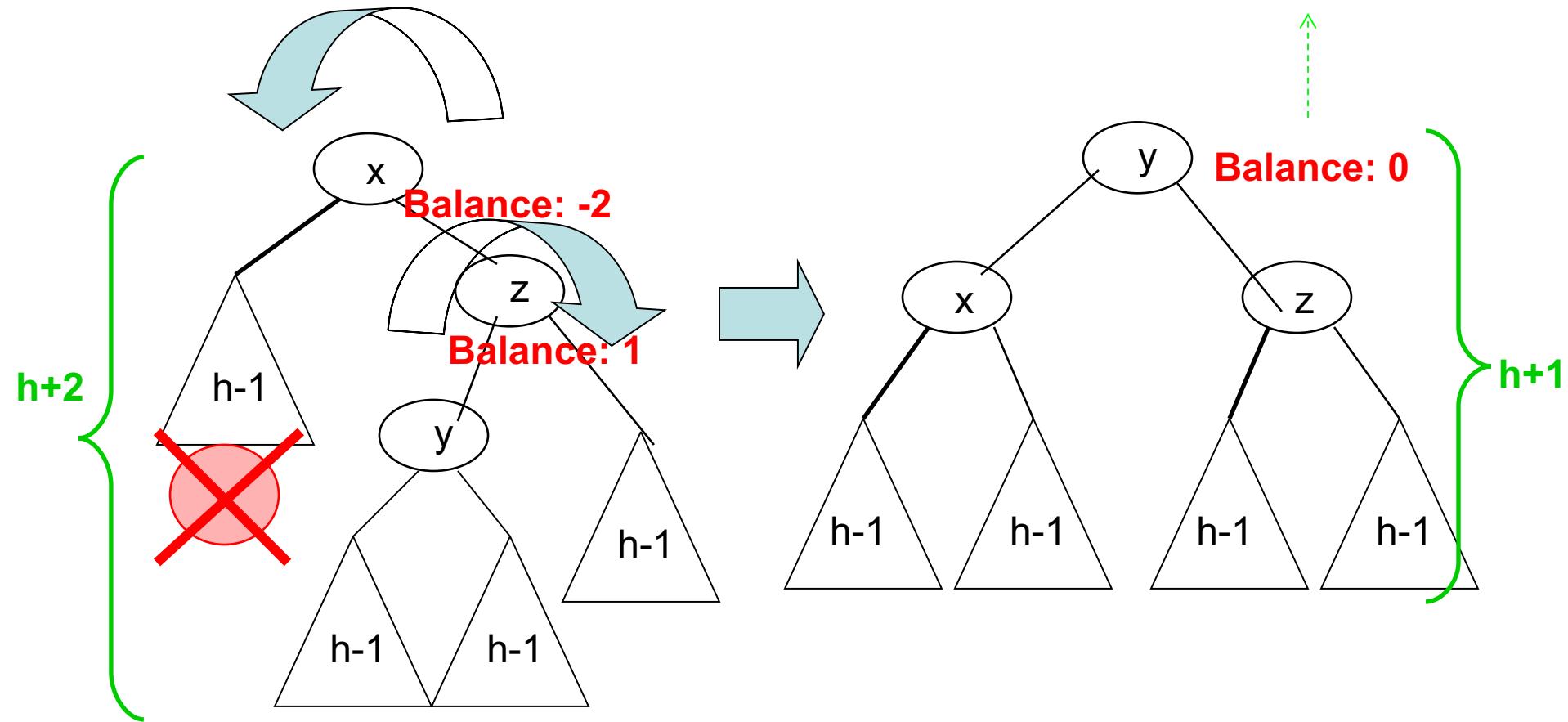


Delete node in left child, the height of the left child decreases

The height of tree after balancing decreases => Unbalance may propagate

AVL delete – Double Rotation

Case 4: Node's **Right – Left** grandchild is too tall



Delete node in left child, the height of the left child decreases

The height of tree after balancing decreases => Unbalance may propagate

Analysis of AVL-DELETE

- Deletion makes $O(h)$ steps, h is $O(\log n)$, thus deletion makes $O(\log n)$ steps
- At the deletion of a node, *rotations may be performed for all the nodes of the deletion path which is $O(h)=O(\log n)$!* In the worst case, it is possible that after doing a balancing, unbalances are propagated on the whole path to the root !

AVL Trees - Summary

- AVL definition of balance: *for each node x, the heights of the left and right subtrees of x differ by at most 1.*
- Maximum height of an AVL tree with n nodes is $h < 1.44 \log_2 n$
- AVL-Insert: $O(\log n)$, Rotations: $O(1)$ (For Insert, unbalances are not propagated after they are solved once)
- AVL-Delete: $O(\log n)$, Rotations: $O(\log n)$ (For Delete, unbalances may be propagated up to the root)

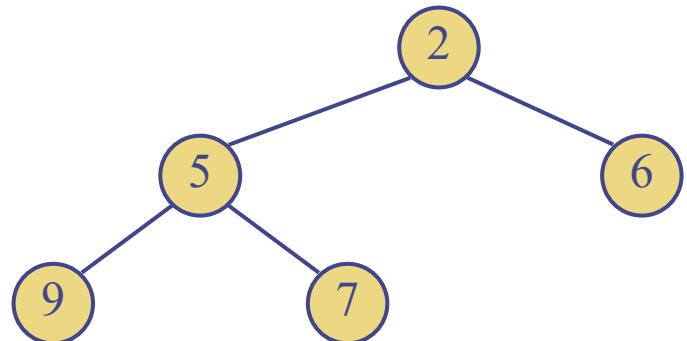
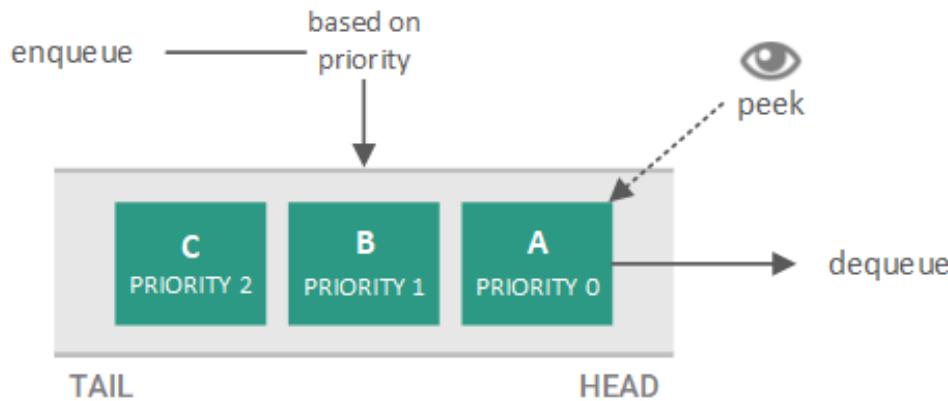
Running Times for AVL Trees

- a single restructure is $O(1)$
 - using a linked-structure binary tree
- find is $O(\log n)$
 - height of tree is $O(\log n)$, no restructures needed
- insert is $O(\log n)$
 - initial find is $O(\log n)$
 - Restructuring up the tree, maintaining heights is $O(\log n)$
- remove is $O(\log n)$
 - initial find is $O(\log n)$
 - Restructuring up the tree, maintaining heights is $O(\log n)$

AVL vs Simple BST

	AVL	Simple BST
Max Height	$1.44 \log n$	n
INSERT	$O(\log n)$	$O(\log(n))$
Rotations at Insert	$O(1)$	
DELETE	$O(\log n)$	$O(\log n)$
Rotations at Delete	$O(\log n)$	

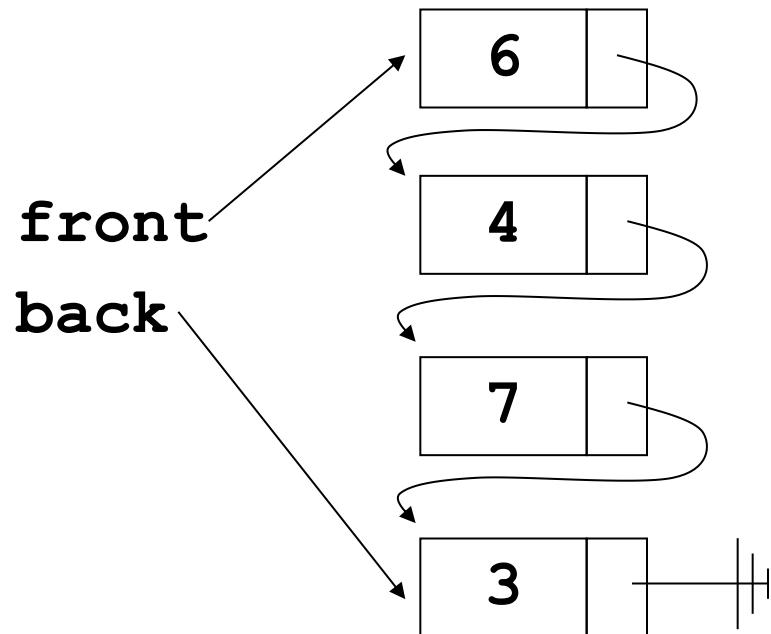
Priority Queues and Heaps



Slides partly from Data Structures and Algorithms in Java
(Goodrich, et. al.), and J. Edgar course at SFU

Queue Implementations

- Either with an array
- Or with a linked list



Priority Queues

- Queues are first-in first-out. Priority queues are a fancier type of queue
 - Maintains an ordering of items in the queue, not necessarily first-in first-out
- Items in a priority queue are given a priority value. The highest priority item is removed first
- Uses include
 - Request scheduling in operation systems
 - Data structure to support shortest path algorithm
 - ...

ADT Priority Queue

- Items in a priority queue have a *priority*
 - Not necessarily numerical
 - Could be lowest first or highest first
- The highest priority item is removed first
- Priority queue operations (Both should be performed in at most $O(\log n)$ time)
 - Insert
 - Remove in priority queue order

Implementing a Priority Queue

- Items have to be removed in priority order
 - This can only be done efficiently if the items are ordered in some way
- One option would be to use a balanced binary search tree
 - Binary search trees are fully ordered and insertion and removal can be implemented in $O(\log n)$ time
 - Some operations (e.g. removal) are complex
 - Although operations are $O(\log n)$ they require quite a lot of structural overhead
- There is a much simpler binary tree solution

Priority Queue ADT

- A priority queue stores a collection of entries as pairs (key, value)
- **insert(k, v)**: inserts an entry with key k and value v
- **removeMin()**: removes and returns the entry with smallest key, or null if the the priority queue is empty
- **min()**: returns, but does not remove, an entry with smallest key, or null if the the priority queue is empty
- **size()**
- **isEmpty()**

Example

A sequence of priority queue methods:

Method	Return Value	Priority Queue Contents
insert(5,A)		{ (5,A) }
insert(9,C)		{ (5,A), (9,C) }
insert(3,B)		{ (3,B), (5,A), (9,C) }
min()	(3,B)	{ (3,B), (5,A), (9,C) }
removeMin()	(3,B)	{ (5,A), (9,C) }
insert(7,D)		{ (5,A), (7,D), (9,C) }
removeMin()	(5,A)	{ (7,D), (9,C) }
removeMin()	(7,D)	{ (9,C) }
removeMin()	(9,C)	{ }
removeMin()	null	{ }
isEmpty()	true	{ }

Entry ADT

- An **entry** in a priority queue is simply a key-value pair
- Priority queues store entries to allow for efficient insertion and removal based on keys
- Methods:
 - **getKey**: returns the key for this entry
 - **getValue**: returns the value associated with this entry
- As a Java interface:

```
public interface Entry<K,V> {  
    K getKey();  
    V getValue();  
}
```

Sequence-based Implementation

- ❑ An unsorted list



- ❑ Performance:

- **insert** takes $O(1)$ time since we can insert the item at the beginning or end of the sequence
- **removeMin** and **min** take $O(n)$ time since we have to traverse the entire sequence to find the smallest key

- ❑ A sorted list



- ❑ Performance:

- **insert** takes $O(n)$ time since we have to find the place where to insert the item
- **removeMin** and **min** take $O(1)$ time, since the smallest key is at the beginning

PQ Sorting



- We use a priority queue
 - Insert the elements with a series of **insert** operations
 - Remove the elements in sorted order with a series of **removeMin** operations
- Running time:
 - Unsorted sequence gives selection-sort: $O(n^2)$ time
 - Sorted sequence gives insertion-sort: $O(n^2)$ time
- **Can we do better?**

Algorithm *PQ-Sort(S, C)*

Input sequence S , comparator C for the elements of S

Output sequence S sorted in increasing order according to C

$P \leftarrow$ priority queue with comparator C

while $\neg S.isEmpty()$

$e \leftarrow S.remove(S.first())$

$P.insert(e, e)$

while $\neg P.isEmpty()$

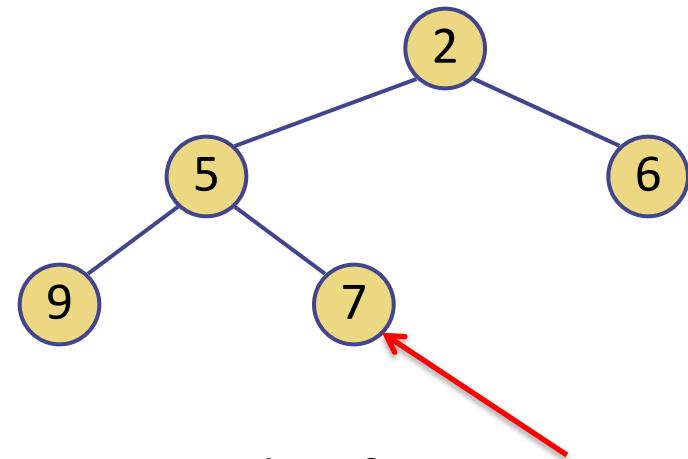
$e \leftarrow P.removeMin().getKey()$

$S.addLast(e)$

Heaps

- A *heap* is **binary tree** with two properties
- Heaps are **complete**
 - All levels, except the bottom, are completely filled in
 - The leaves on the bottom level are as far to the left as possible.
- Heaps are **partially ordered**
 - The value of a node is at least as large as its children's values, for a *max* heap or
 - The value of a node is no greater than its children's values, for a *min* heap

- For all internal node v other than the root, $\text{key}(v) \geq \text{key}(\text{parent}(v))$
- Complete Binary Tree: let h be the height of the heap
 - for $i = 0, \dots, h - 1$, there are 2^i nodes of depth i
 - at depth h , the internal nodes are to the left of the external nodes



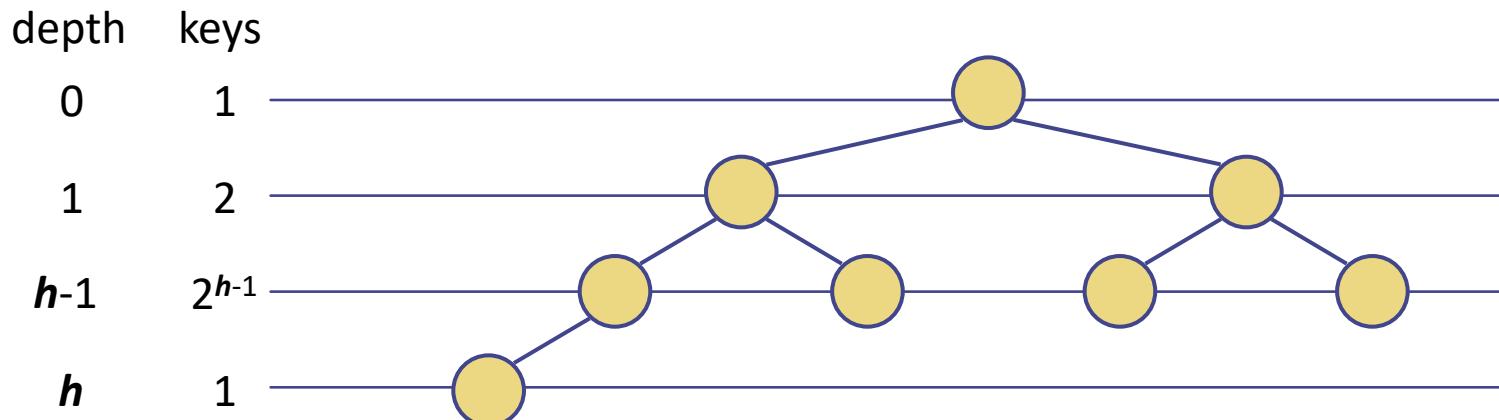
- The last node of a heap is the rightmost node of maximum depth

Height of a Heap

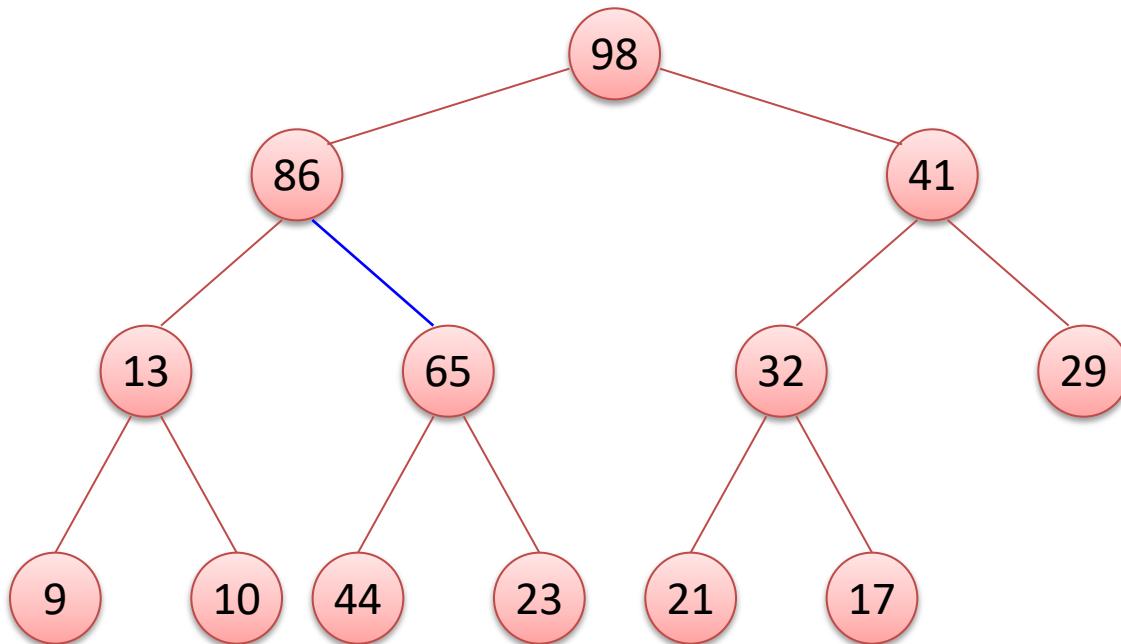
□ Theorem: A heap storing n keys has height $O(\log n)$

Proof: (we apply the complete binary tree property)

- Let h be the height of a heap storing n keys
- Since there are 2^i keys at depth $i = 0, \dots, h - 1$ and at least one key at depth h , we have $n \geq 1 + 2 + 4 + \dots + 2^{h-1} + 1$
- Thus, $n \geq 2^h$, i.e., $h \leq \log n$



Partially Ordered Tree – max heap

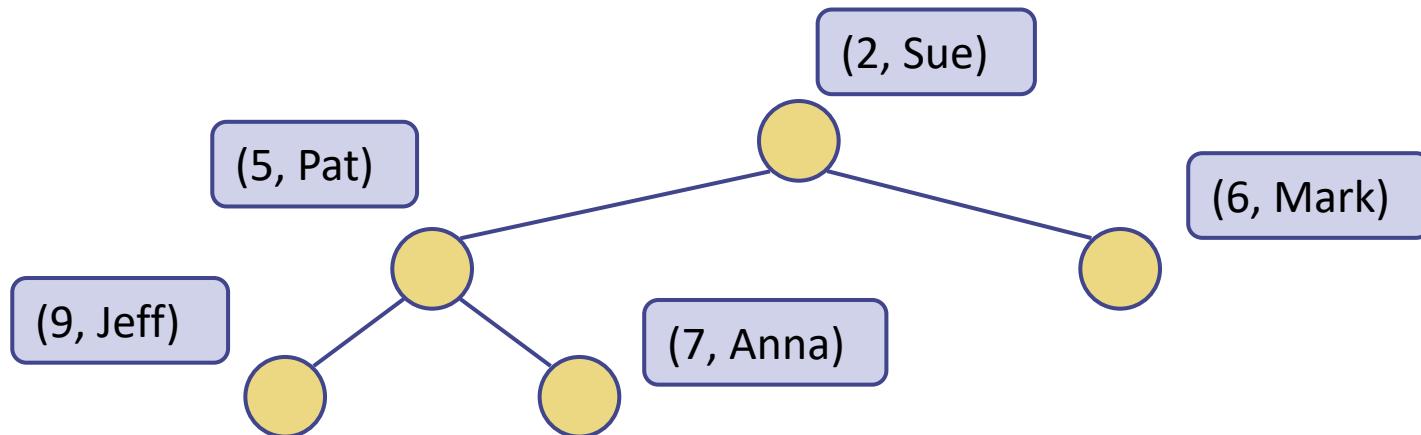


Heaps are not fully ordered, an inorder traversal would result in:

9, 13, 10, 86, 44, 65, 23, 98, 21, 32, 17, 41, 29

Heaps and Priority Queues

- We can use a heap to implement a priority queue
- We store a (key, element) item at each internal node
- We keep track of the position of the last node

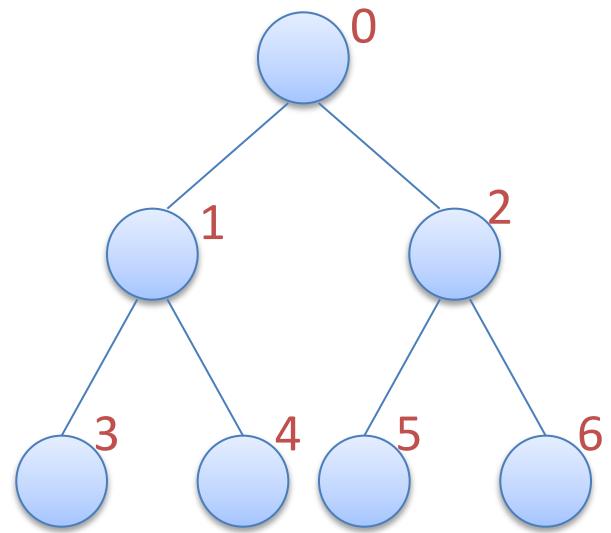


Priority Queues and Heaps

- A heap can implement a priority queue
- The item at the top of the heap must always be the highest priority value
 - Because of the partial ordering property
- Implement priority queue operations:
 - Insertions – insert an item into a heap
 - Removal – remove and return the heap's root
 - For both operations preserve the heap property

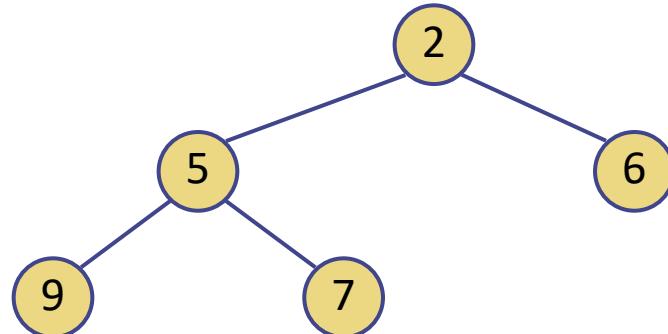
Heap Implementation

- Heaps can be implemented using *arrays*
- There is a natural method of indexing tree nodes
 - Index nodes from top to bottom and left to right as shown on the right
 - Because heaps are *complete* binary trees there can be no gaps in the array



Array-based Heap Implementation

- We can represent a heap with n keys by means of an array of length n
- For the node at rank i
 - the left child is at rank $2i + 1$
 - the right child is at rank $2i + 2$
- Links between nodes are not explicitly stored
- Operation add corresponds to inserting at rank $n + 1$
- Operation remove_min corresponds to removing at rank n
- Yields in-place heap-sort

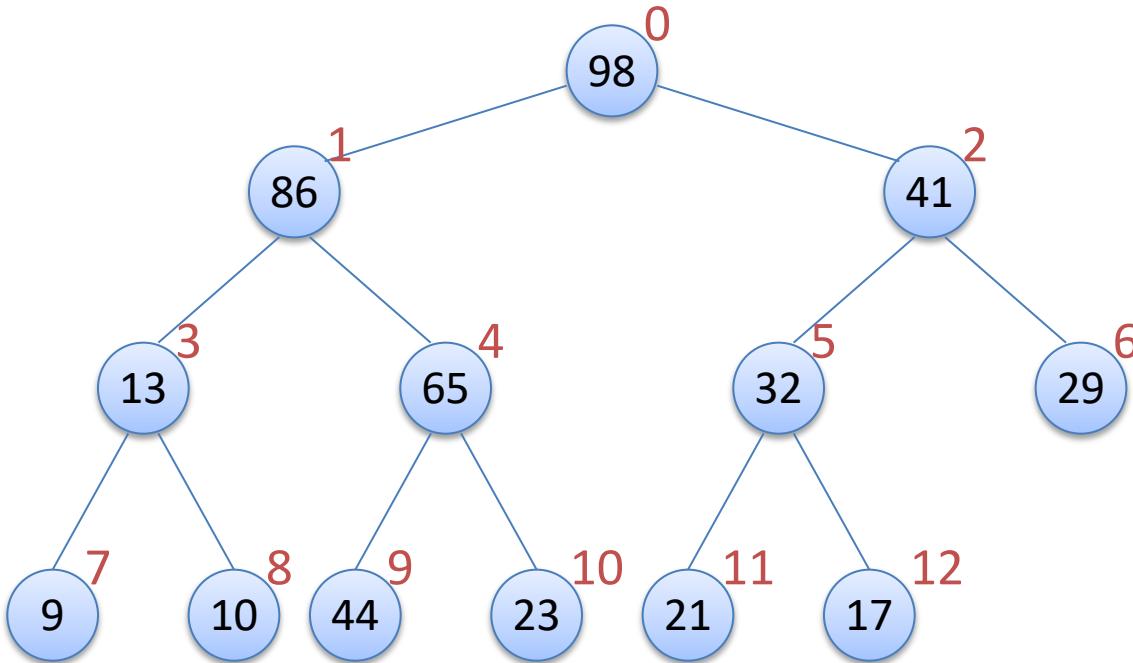


2	5	6	9	7
0	1	2	3	4

Referencing Nodes

- It will be necessary to find the index of the parents of a node
 - Or the children of a node
- The array is indexed from 0 to $n - 1$
 - Each level's nodes are indexed from:
 - $2^{\text{level}} - 1$ to $2^{\text{level}+1} - 2$ (where the root is level 0)
 - The children of a node i , are the array elements indexed at $2i + 1$ and $2i + 2$
 - The parent of a node i , is the array element indexed at $\text{floor}((i - 1) / 2)$

Heap Array Example

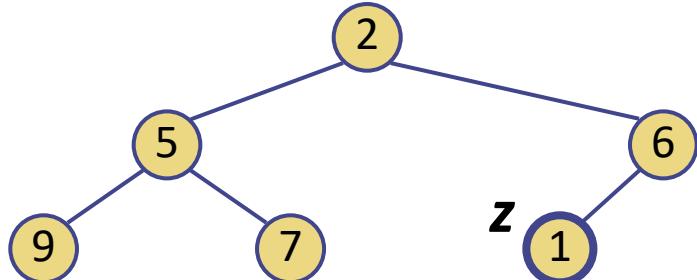
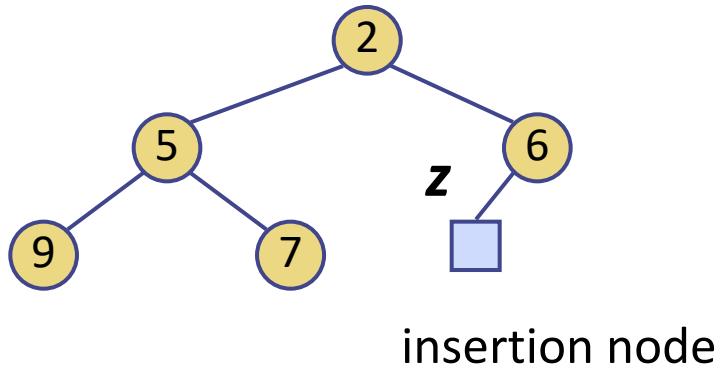


index	0	1	2	3	4	5	6	7	8	9	10	11	12
value	98	86	41	13	65	32	29	9	10	44	23	21	17

Underlying Array

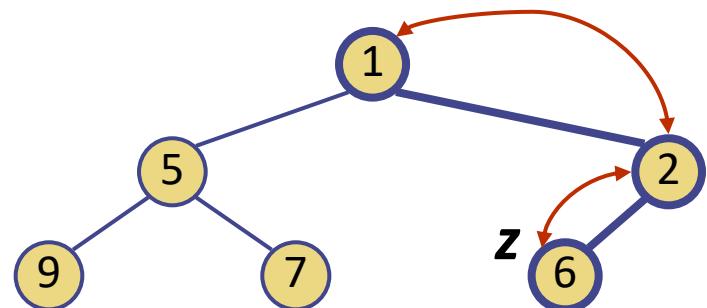
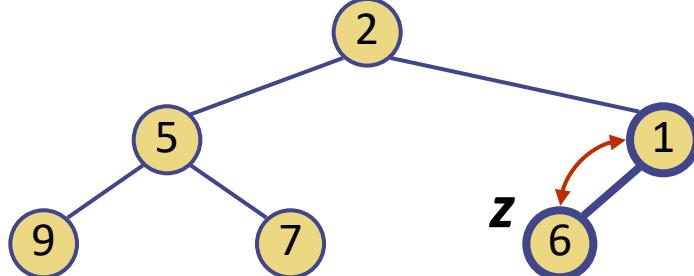
Insertion into a Heap

- Insertion of a key k to the heap
 1. Find the insertion node z
(the new last node)
 2. Store k at z
 3. Restore the heap-order
property (discussed next)



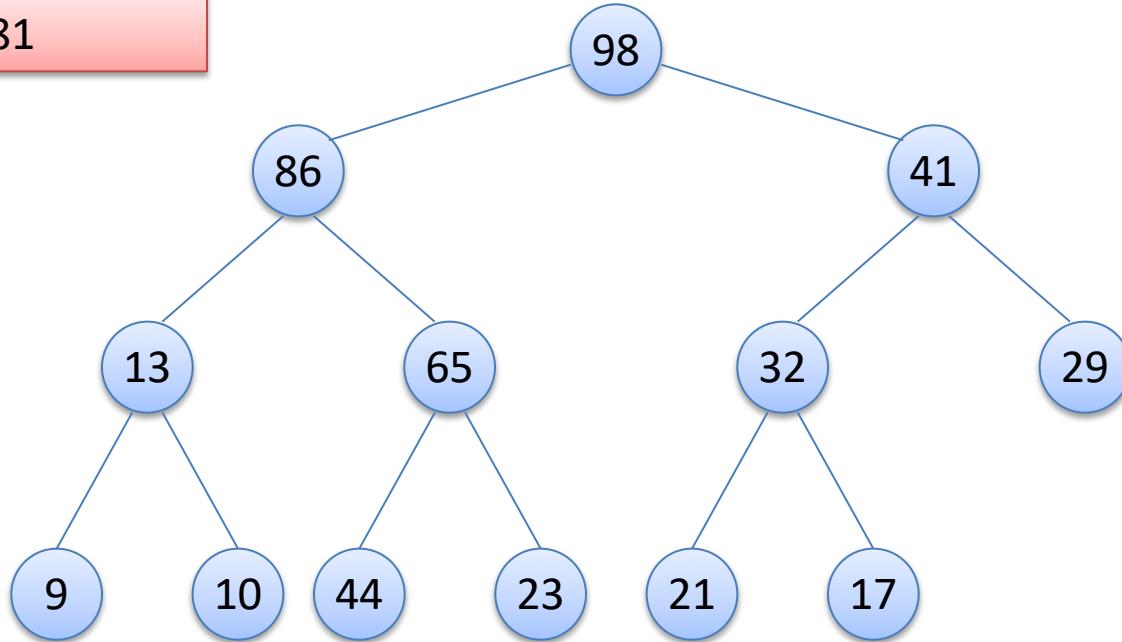
UpHeap

- After the insertion of a new key k , the heap-order property may be violated. Algorithm UpHeap restores the heap-order property by swapping k along an upward path from the insertion node
- UpHeap terminates when the key k reaches the root or a node whose parent has a key smaller than or equal to k
- Since a heap has height $O(\log n)$, UpHeap runs in $O(\log n)$ time



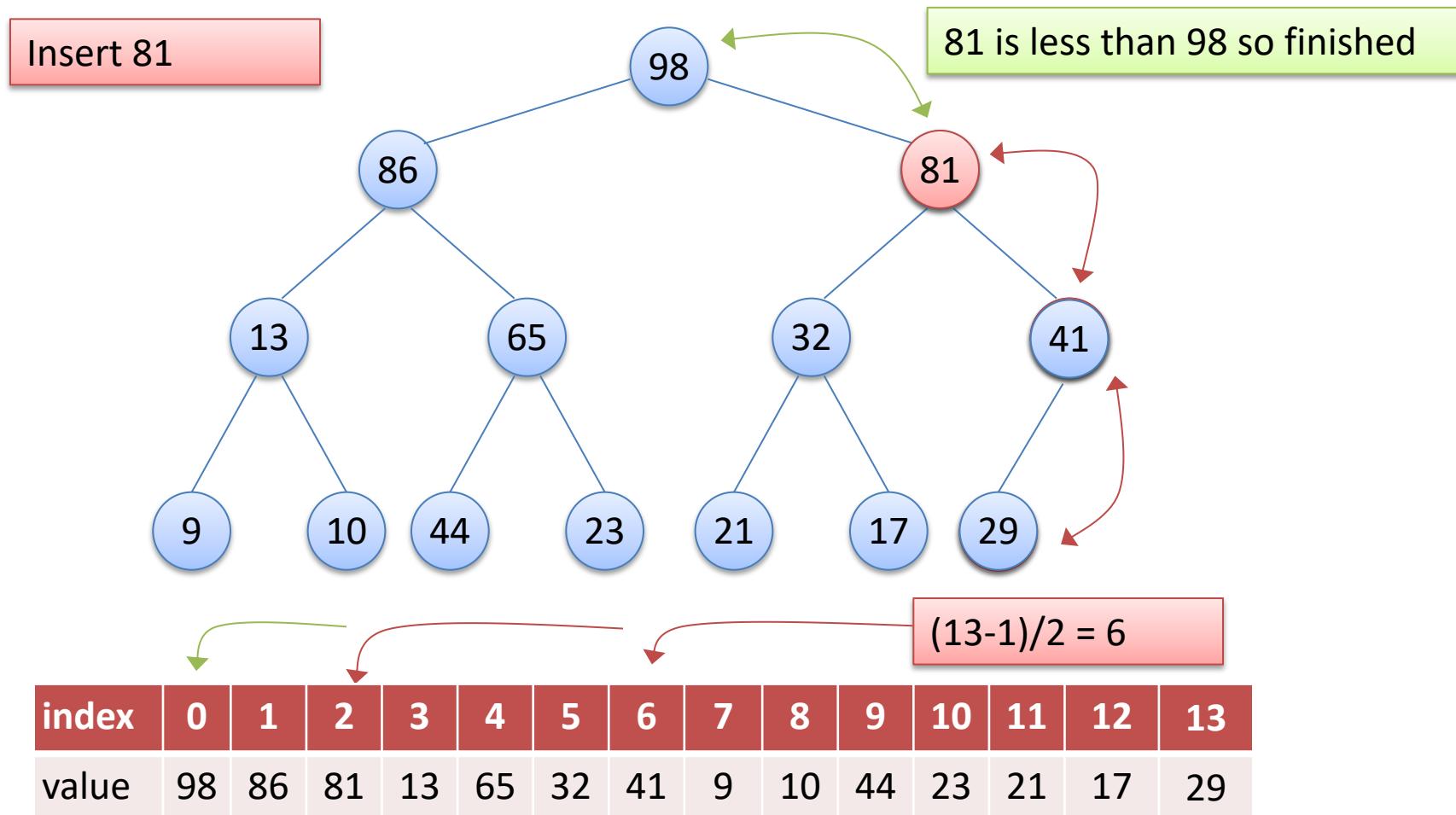
Heap Insertion Example

Insert 81



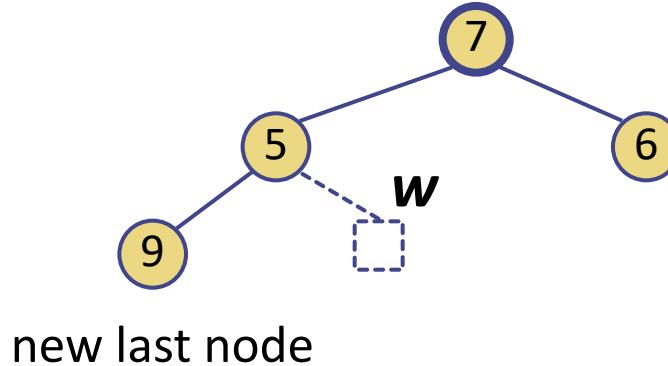
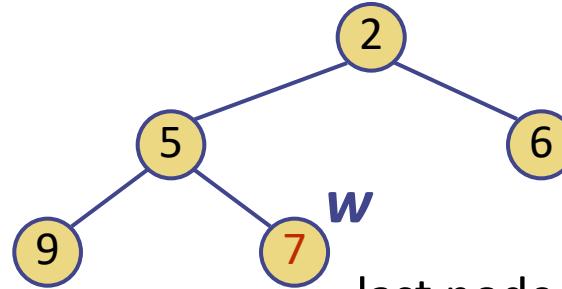
index	0	1	2	3	4	5	6	7	8	9	10	11	12	13
value	98	86	41	13	65	32	29	9	10	44	23	21	17	

Heap Insertion Example



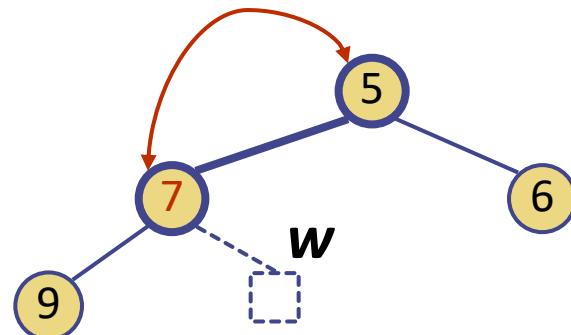
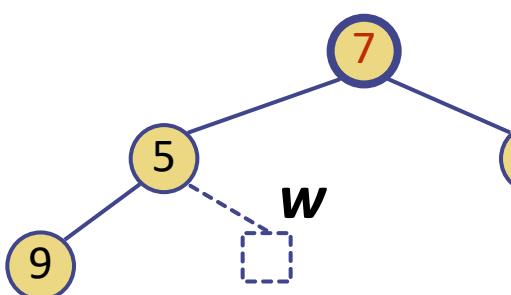
Removal from a Heap

- Removal of the root key from the heap
 - 1. Replace the root key with the key of the last node w
 - 2. Remove w
 - 3. Restore the heap-order property (discussed next)

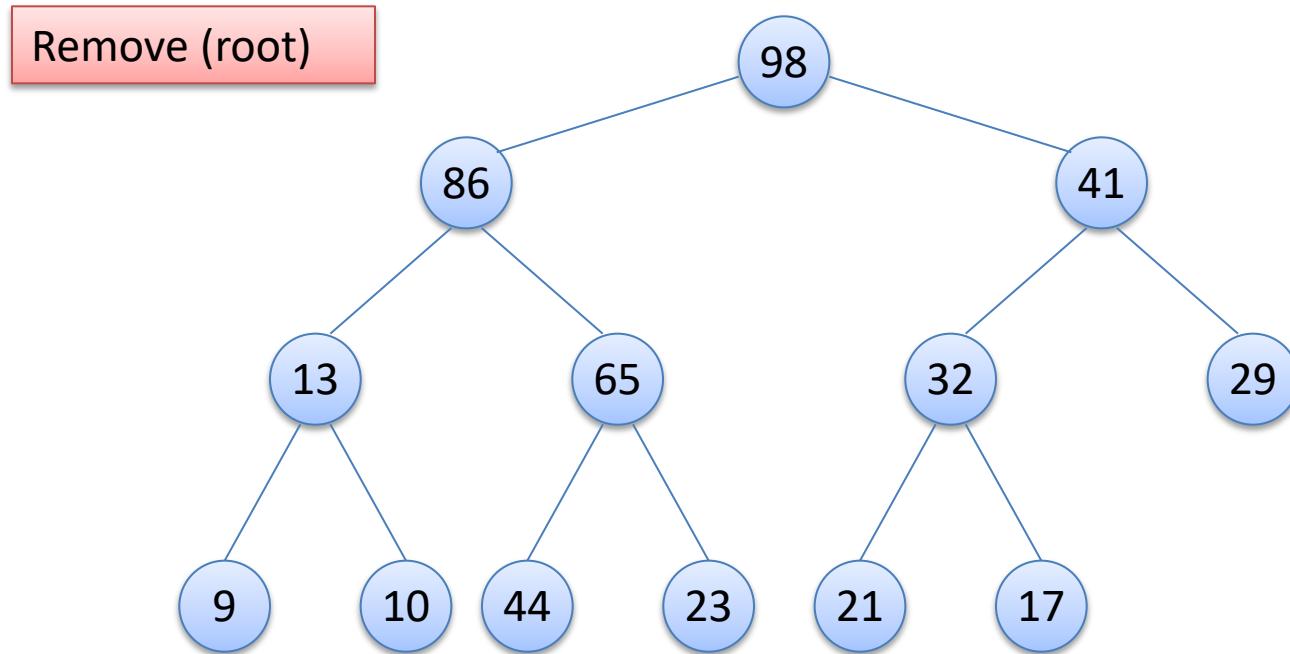


DownHeap

- After replacing the root key with the key k of the last node, the heap-order property may be violated. Algorithm DownHeap restores the heap-order property by swapping key k along a downward path from the root
- DownHeap terminates when key k reaches a leaf or a node whose children have keys greater than or equal to k
- Since a heap has height $O(\log n)$, DownHeap runs in $O(\log n)$ time

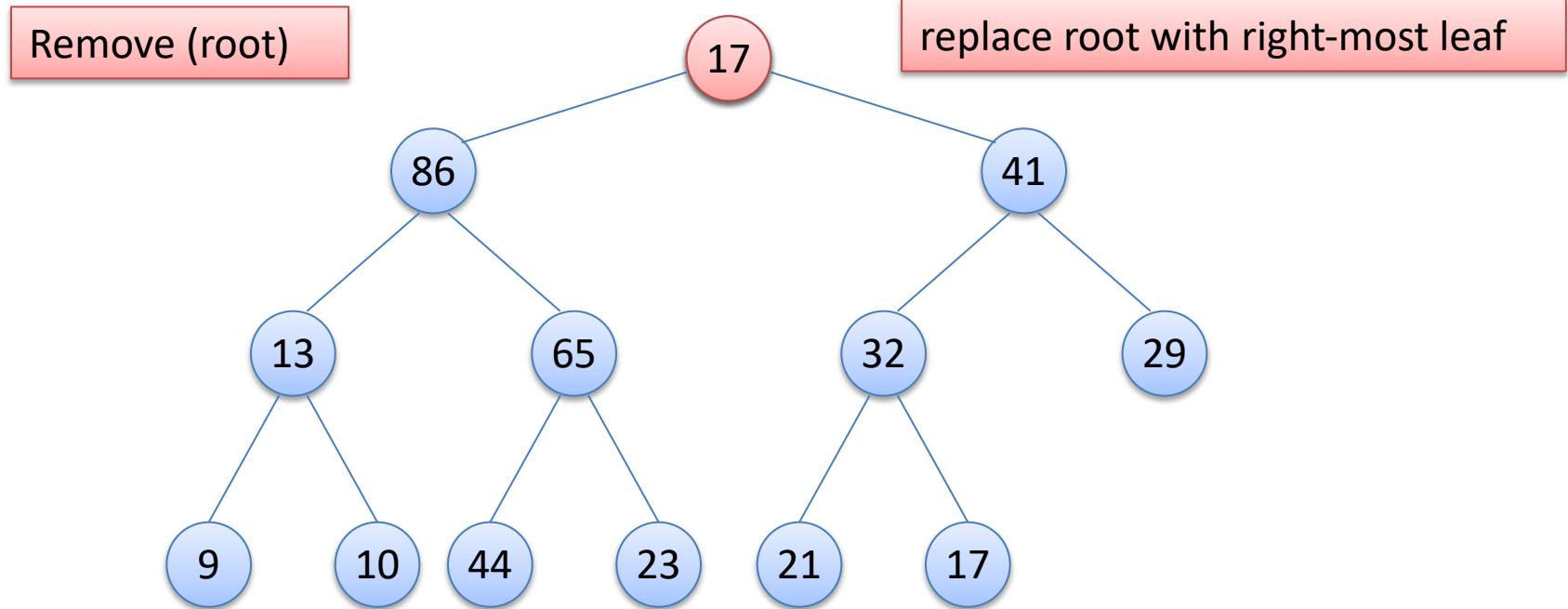


Heap Removal Example



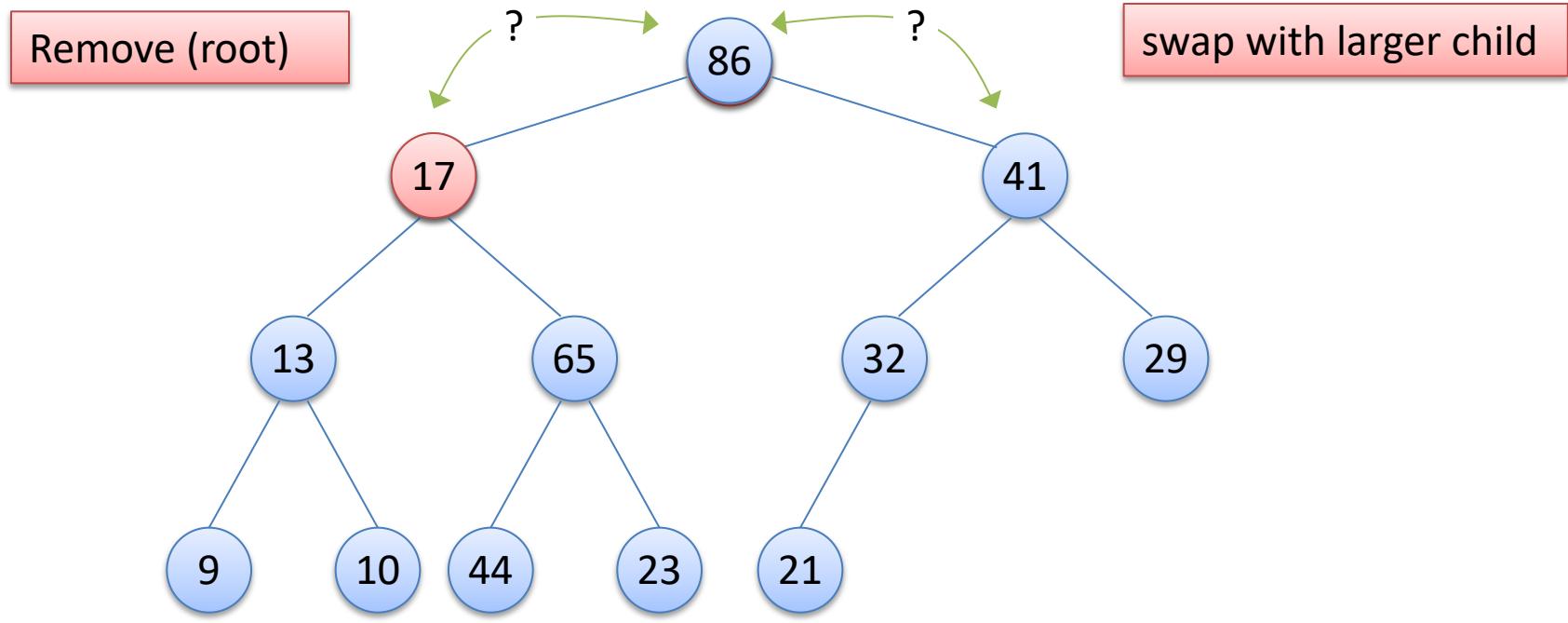
index	0	1	2	3	4	5	6	7	8	9	10	11	12
value	98	86	41	13	65	32	29	9	10	44	23	21	17

Heap Removal Example



index	0	1	2	3	4	5	6	7	8	9	10	11	12
value	17	86	41	13	65	32	29	9	10	44	23	21	

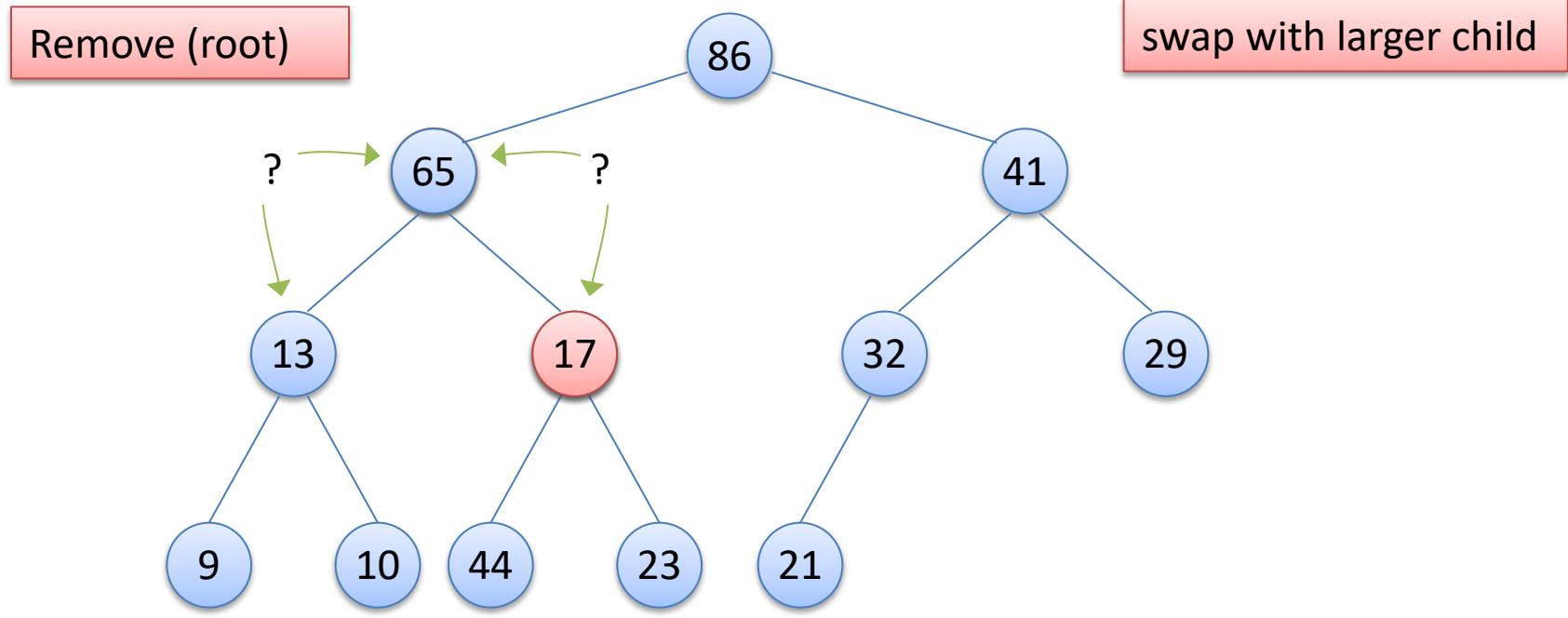
Heap Removal Example



index	0	1	2	3	4	5	6	7	8	9	10	11	12
value	86	17	41	13	65	32	29	9	10	44	23	21	

children of root: $2*0+1, 2*0+2 = 1, 2$

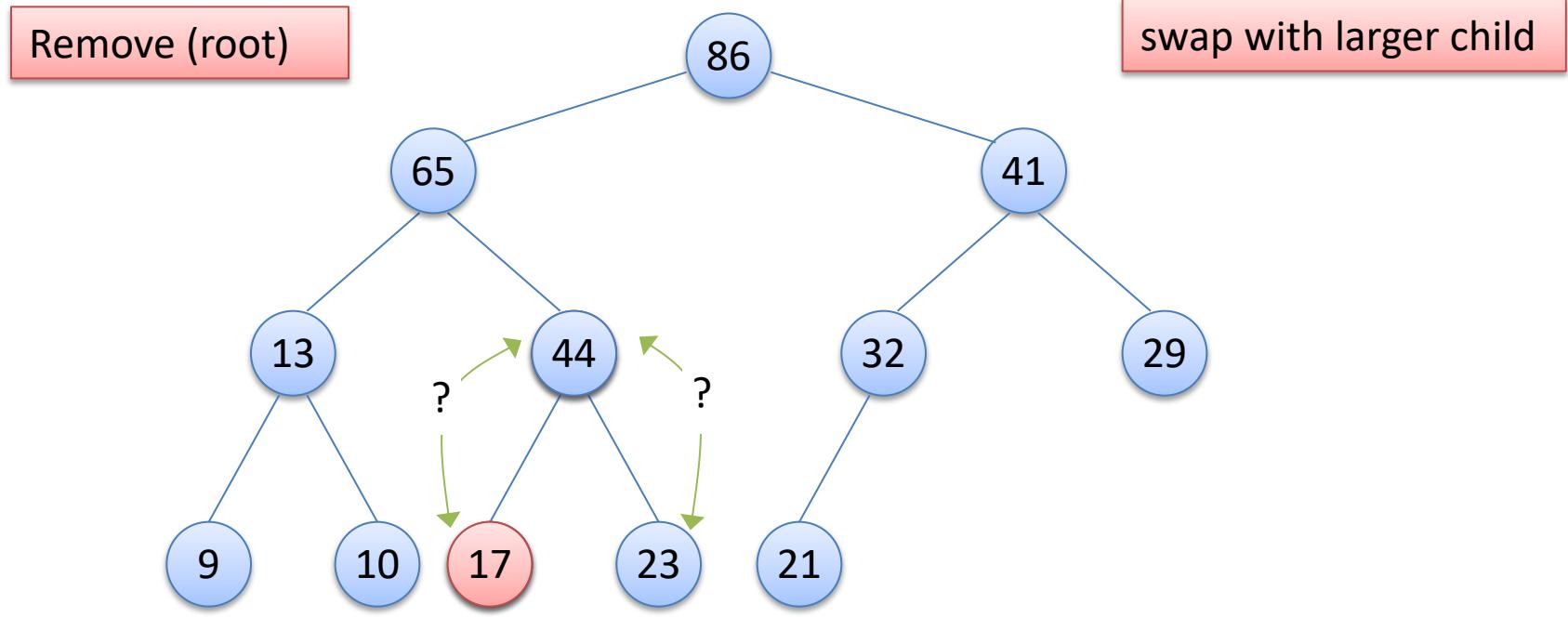
Heap Removal Example



index	0	1	2	3	4	5	6	7	8	9	10	11	12
value	86	65	41	13	17	32	29	9	10	44	23	21	

children of 1: $2 \cdot 1 + 1, 2 \cdot 1 + 2 = 3, 4$

Heap Removal Example



index	0	1	2	3	4	5	6	7	8	9	10	11	12
value	86	65	41	13	44	32	29	9	10	17	23	21	

children of 4: $2 \cdot 4 + 1, 2 \cdot 4 + 2 = 9, 10$

Heap Efficiency

- For both insertion and removal the heap performs at most $height$ swaps
 - For insertion at most $height$ comparisons
 - To UpHeap the array
 - For removal at most $height * 2$ comparisons
 - To DownHeap the array (have to compare two children)
- Height of a complete binary tree is $\lfloor \log_2(n) \rfloor$
 - Both insertion and removal are therefore $O(\log n)$

Sorting with Heaps

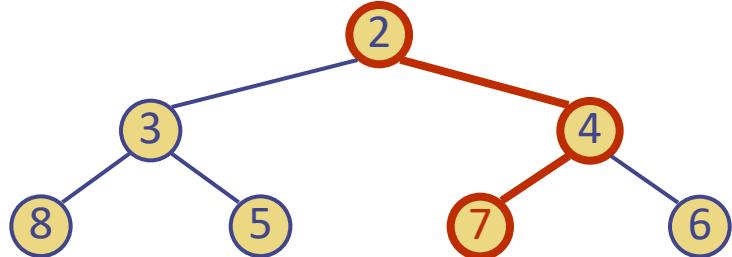
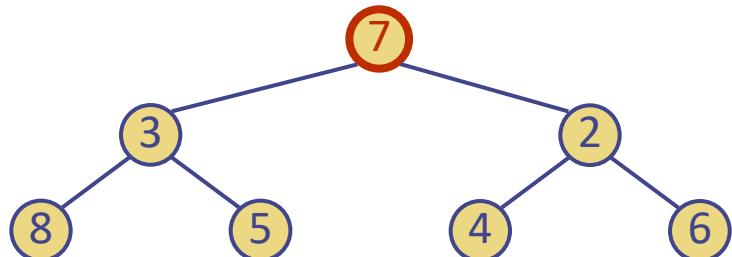
- **Observation 1:** Removal of a node from a heap can be performed in $O(\log n)$ time
- **Observation 2:** Nodes are removed in order
- **Conclusion:** Removing all of the nodes one by one would result in sorted output
- **Analysis:** Removal of *all* the nodes from a heap is a $O(n * \log n)$ operation

But ...

- A heap can be used to return sorted data
 - In $O(n * \log n)$ time
- However, we can't assume that the data to be sorted just happens to be in a heap!
 - **Aha!** But we can *put* it in a heap.
 - Inserting an item into a heap is a $O(\log n)$ operation so inserting n items is $O(n * \log n)$
- But we can do better than just repeatedly calling the insertion algorithm

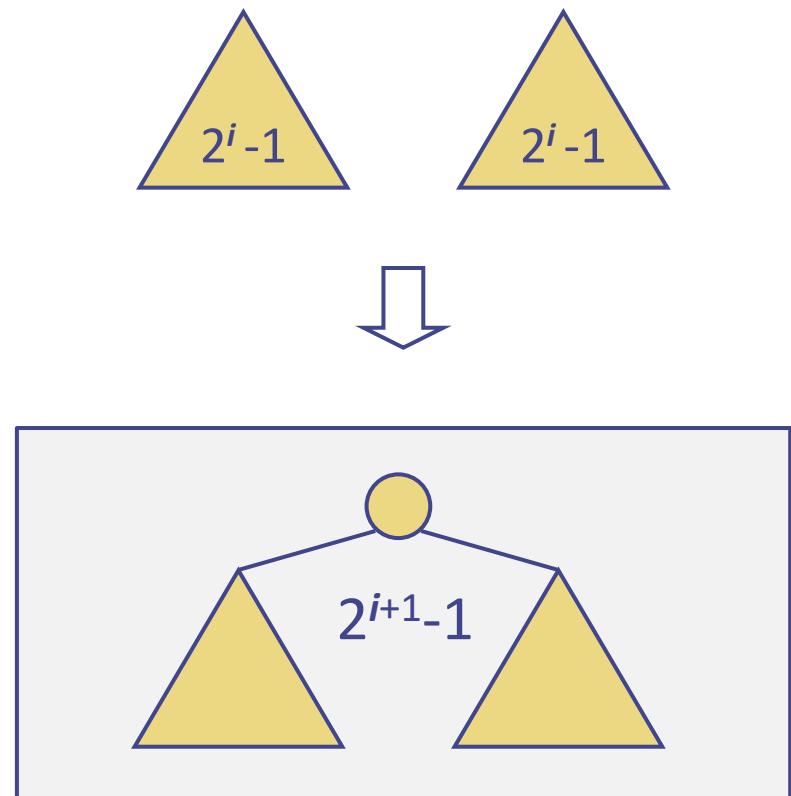
Merging Two Heaps

- We are given two heaps and a key k
- We create a new heap with the root node storing k and with the two heaps as subtrees
- We perform DownHeap to restore the heap-order property

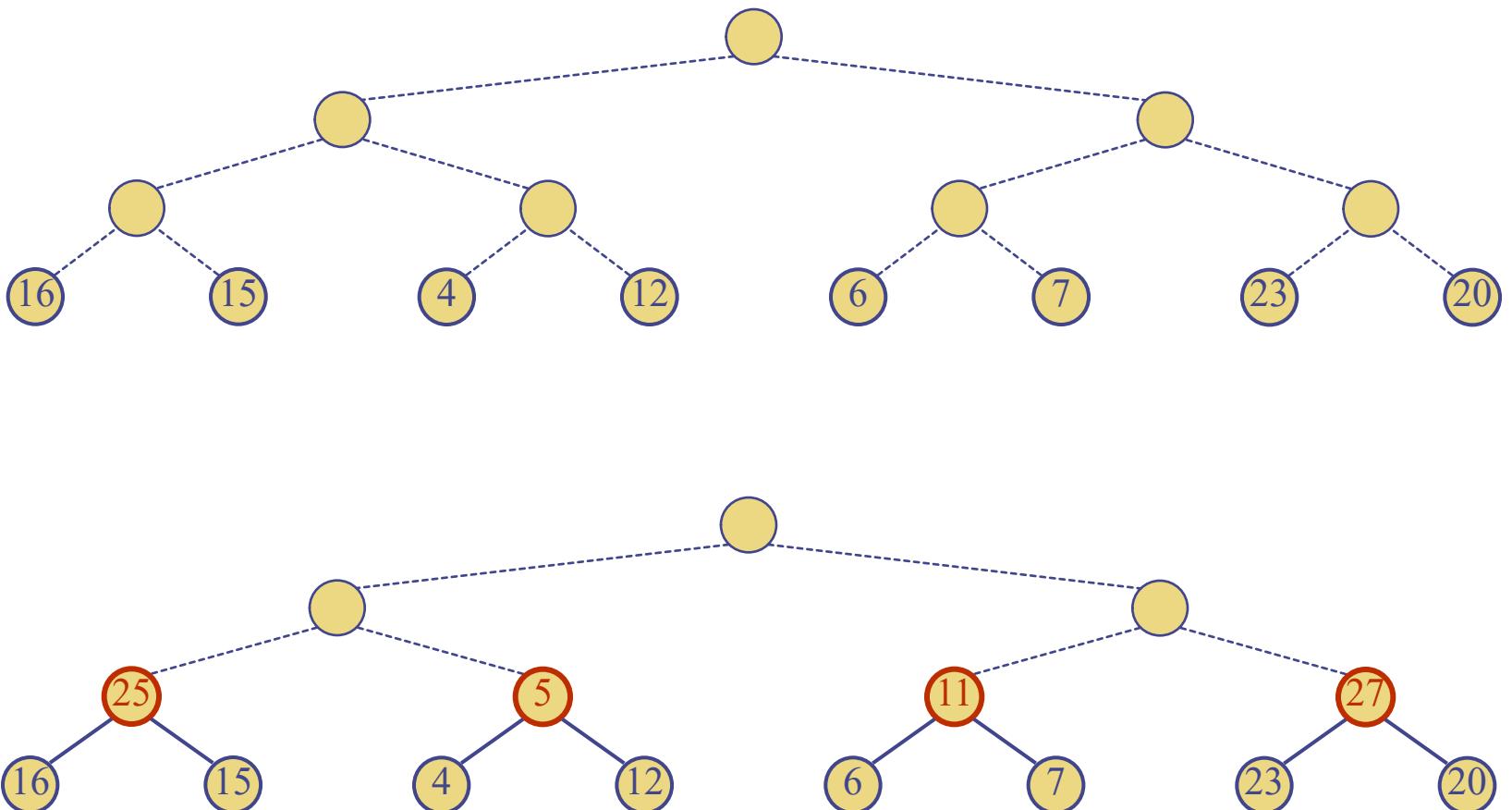


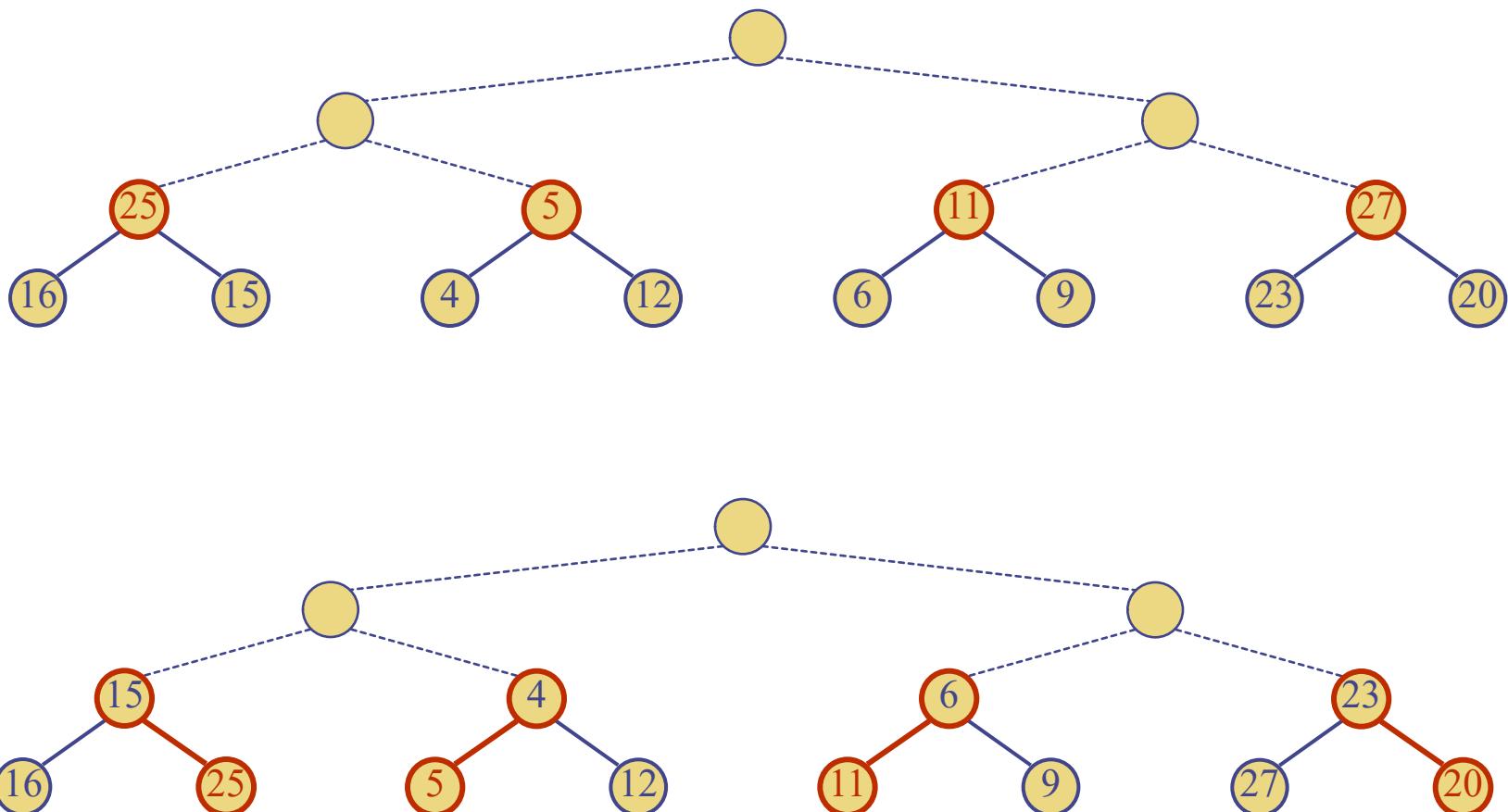
Bottom-up Heap Construction

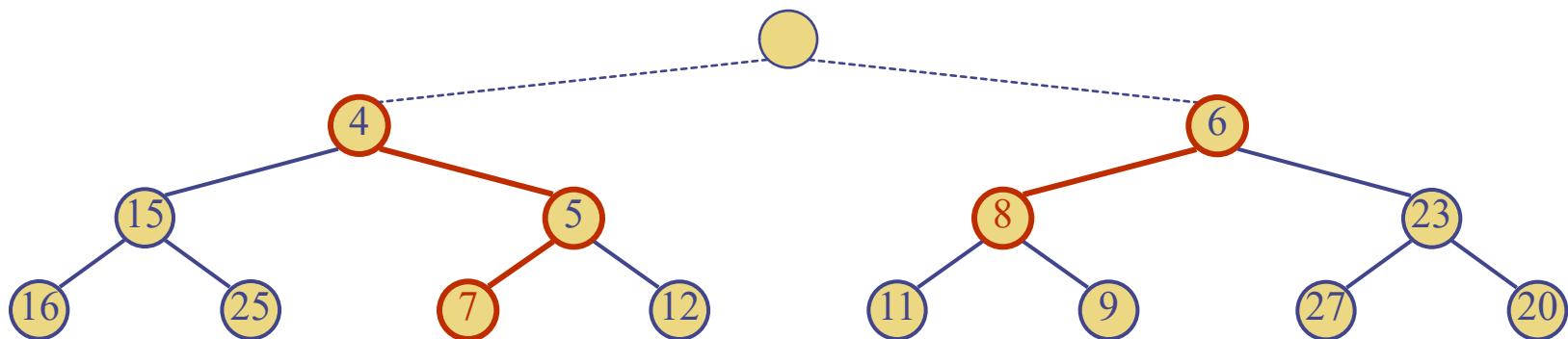
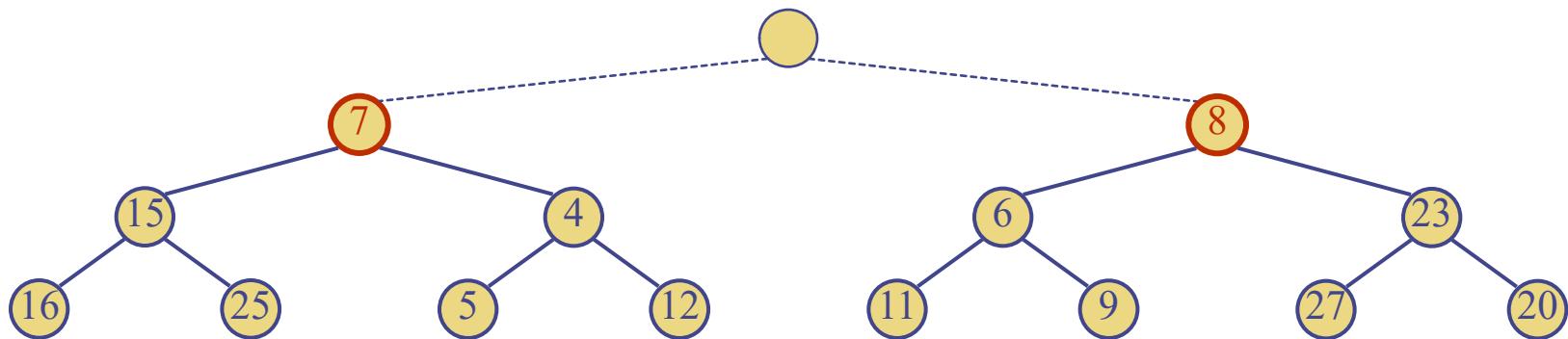
- We can construct a heap storing n given keys in using a bottom-up construction with $\log n$ phases
- In phase i , pairs of heaps with $2^i - 1$ keys are merged into heaps with $2^{i+1} - 1$ keys

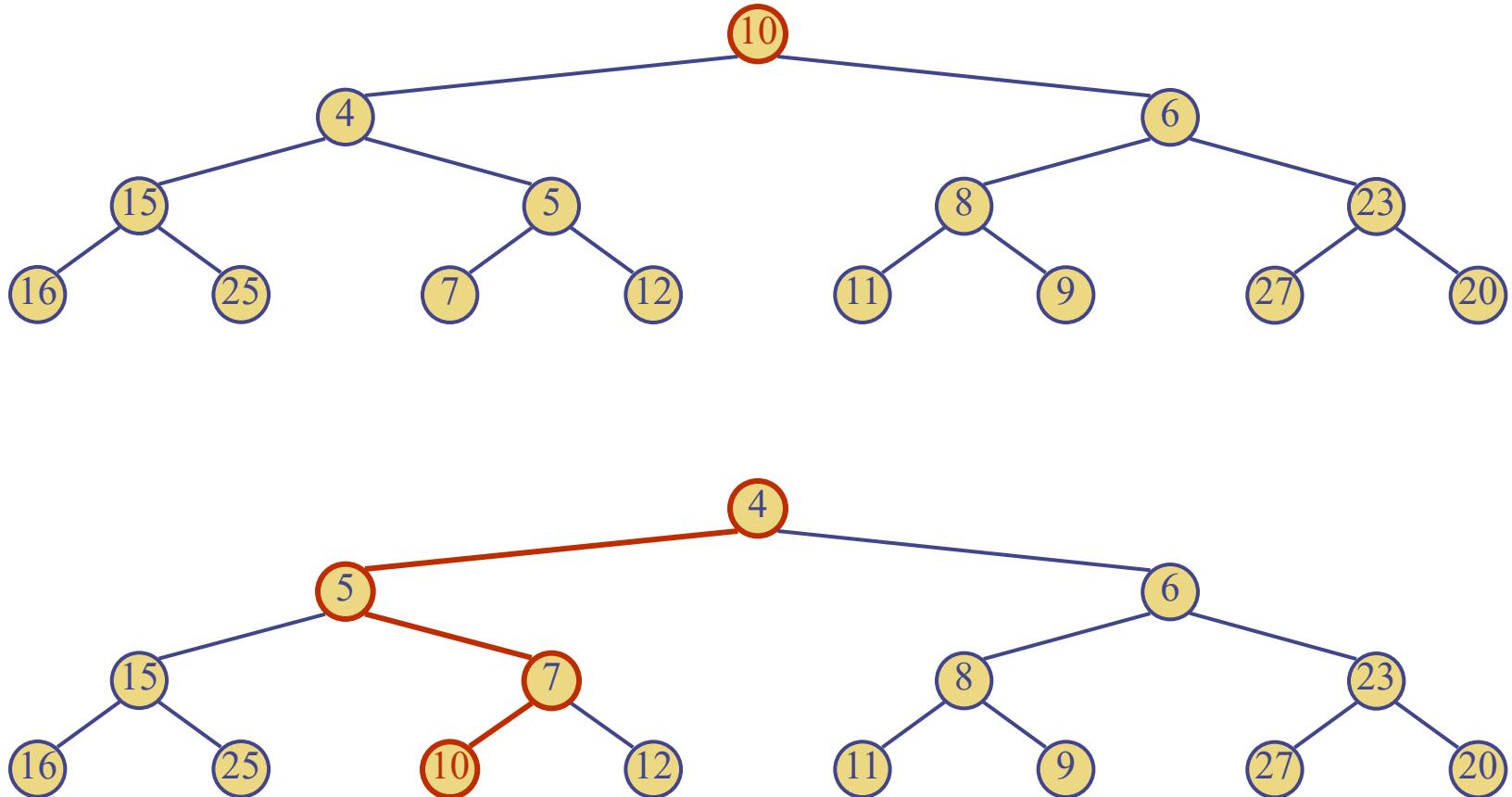


Example









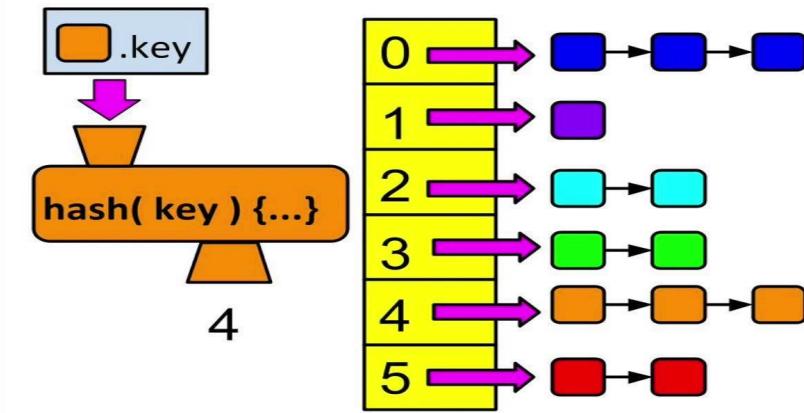
Heap-Sort Algorithm

- Consider a heap-based priority queue with n items
 - the space used is $O(n)$
 - methods `insert` and `removeMin` take $O(\log n)$ time
 - methods `size`, `isEmpty`, and `min` take time $O(1)$ time
- We can sort a sequence of n elements in $O(n \log n)$ time
- Heap-sort is much faster than quadratic sorting algorithms, such as insertion-sort and selection-sort

HeapSort Notes

- The algorithm runs in $O(n * \log n)$ time
 - Considerably more efficient than selection sort and insertion sort
 - The same average case complexity as MergeSort and QuickSort

Maps (Dictionaries) and Hash Tables



Slides partly from Data Structures and Algorithms in Java
(Goodrich, et. al.), and J. Edgar course at SFU

Problem Examples

- What can we do if we want rapid access to individual data items?
 - Looking up the address of someone making a 911 call
 - Checking the spelling of words by looking up each one in a dictionary
 - Looking up data in an air traffic control system
- In each case speed is very important
 - But the data does not need to be maintained in order

Dictionary ADT

- A dictionary is an ADT composed of a collection of (key, value) pairs, such that each possible key appears at most once in the collection.
- Operations
 - Insert (key,value) pair
 - Lookup value for a key
 - Remove (key,value) pair
 - Modify (key,value) pair
- Dictionary ADT also known as Associative Array, Map, or symbol table.

Maps



- A map models a searchable collection of key-value entries
- The main operations of a map are for searching, inserting, and deleting items
- Multiple entries with the same key are not allowed
- Applications: address book, student-record database, etc.

The Map ADT

- ❑ **get(k)**: if the map M has an entry with key k, return its associated value; else, return null
- ❑ **put(k, v)**: insert entry (k, v) into the map M; if key k is not already in M, then return null; else, return old value associated with k
- ❑ **remove(k)**: if the map M has an entry with key k, remove it from M and return its associated value; else, return null
- ❑ **size()**, **isEmpty()**
- ❑ **entrySet()**: return an iterable collection of the entries in M
- ❑ **keySet()**: return an iterable collection of the keys in M
- ❑ **values()**: return an iterable collection of the values in M

Example

<i>Operation</i>	<i>Output</i>	<i>Map</i>
isEmpty()	true	\emptyset
put(5,A)	null	(5,A)
put(7,B)	null	(5,A),(7,B)
put(2,C)	null	(5,A),(7,B),(2,C)
put(8,D)	null	(5,A),(7,B),(2,C),(8,D)
put(2,E)	C	(5,A),(7,B),(2,E),(8,D)
get(7)	B	(5,A),(7,B),(2,E),(8,D)
get(4)	null	(5,A),(7,B),(2,E),(8,D)
get(2)	E	(5,A),(7,B),(2,E),(8,D)
size()	4	(5,A),(7,B),(2,E),(8,D)
remove(5)	A	(7,B),(2,E),(8,D)
remove(2)	E	(7,B),(8,D)
get(2)	null	(7,B),(8,D)
isEmpty()	false	(7,B),(8,D)

Possible Implementations

- Sorted (by search key), array-based
- Sorted (by search key), link-based
- Unsorted, array-based
- Unsorted, link-based

Selecting an Implementation

	<u>Insertion</u>	<u>Removal</u>	<u>Retrieval</u>	<u>Traversal</u>
Unsorted array-based	$O(1)$	$O(n)$	$O(n)$	$O(n)$
Unsorted link-based	$O(1)$	$O(n)$	$O(n)$	$O(n)$
Sorted array-based	$O(n)$	$O(n)$	$O(\log n)$	$O(n)$
Sorted link-based	$O(n)$	$O(n)$	$O(n)$	$O(n)$
Binary search tree	$O(\log n)$	$O(\log n)$	$O(\log n)$	$O(n)$

The average-case order of the ADT dictionary operations for various implementations

Possible Solutions

- Balanced binary search tree
 - Binary search trees allow lookup and insertion in $O(\log n)$ time, which is relatively fast.
 - Binary search trees also maintain data in order, which may be not necessary for some problems.
- Arrays
 - Insertion in constant time, but lookup in linear time.
 - But, if we know the index of a data item lookup can be performed in constant time.
- Can we use an array to insert and retrieve data in constant time? Yes – as long as we know an item's index

Example

- Let's consider storing information about Canadians given their phone numbers
 - Between 000-000-000 and 999-999-9999
- It's easy to convert phone numbers to integers
 - Just get rid of the "-"s
 - The keys range between 0 and 9,999,999,999
- Can we use array to insert and retrieve in constant time?

- If we use Canadian phone numbers as the index to an array how big is the array?
 - 9,999,999,999 (ten billion)
 - That's a really big array!
- Consider that the estimate of the current population of Canada is about 34M
 - That means that we will use around 0.3% of the array
 - That's a lot of wasted space
 - And the array probably won't fit in main memory ...

More Examples

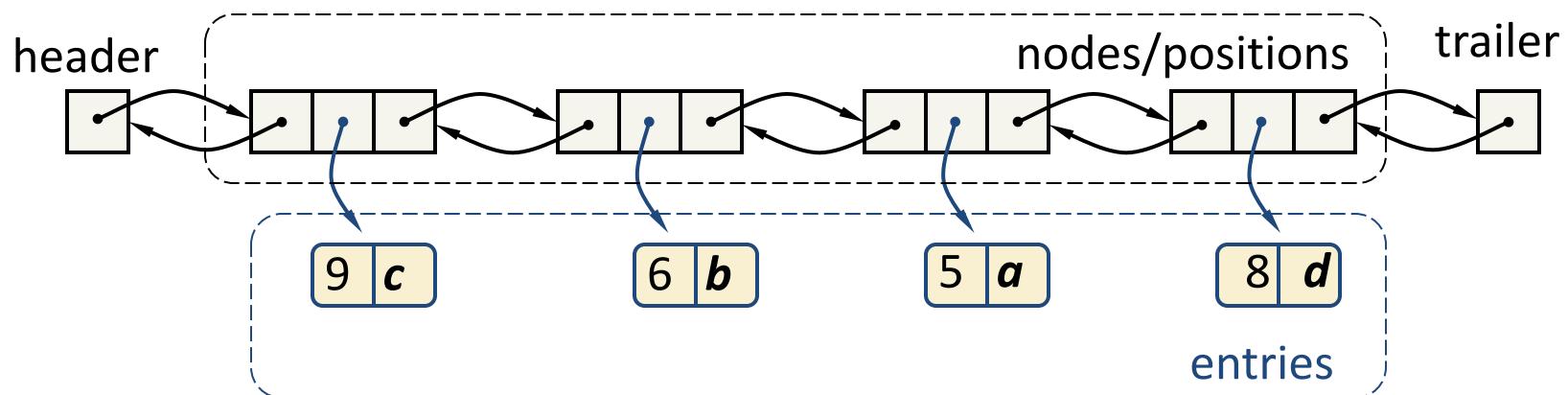
- What if we had to store data by name?
 - We would need to convert strings to integer indexes
- Here is one way to encode strings as integers
 - Assign a value between 1 and 26 to each letter
 - a = 1, z = 26 (regardless of case)
 - Sum the letter values in the string

$$\text{"dad"} = 4 + 1 + 4 = 9$$

$$\text{"add"} = 1 + 4 + 4 = 9$$

A Simple List-Based Map

- We could implement a map using an unsorted list
 - We store the entries of the map in a doubly-linked list S, in arbitrary order
 - S supports the node list ADT



Performance of a List-Based Map

- ❑ Performance:
 - **put** takes $O(1)$ time since we can insert the new item at the beginning or at the end of the sequence
 - **get** and **remove** take $O(n)$ time since in the worst case (the item is not found) we traverse the entire sequence to look for an item with the given key
- ❑ The unsorted list implementation is effective only for maps of small size or for maps in which puts are the most common operations, while searches and removals are rarely performed (e.g., historical record of logins to a workstation)

Finding Unique String Values

- Ideally we would like to have a unique integer for each possible string
- This is relatively straightforward
 - As before, assign each letter a value between 1 and 26
 - And multiply the letter's value by 26^i , where **i** is the position of the letter in the word:
 - "dad" = $4*26^2 + 1*26^1 + 4*26^0 = 2734$
 - "add" = $1*26^2 + 4*26^1 + 4*26^0 = 784$

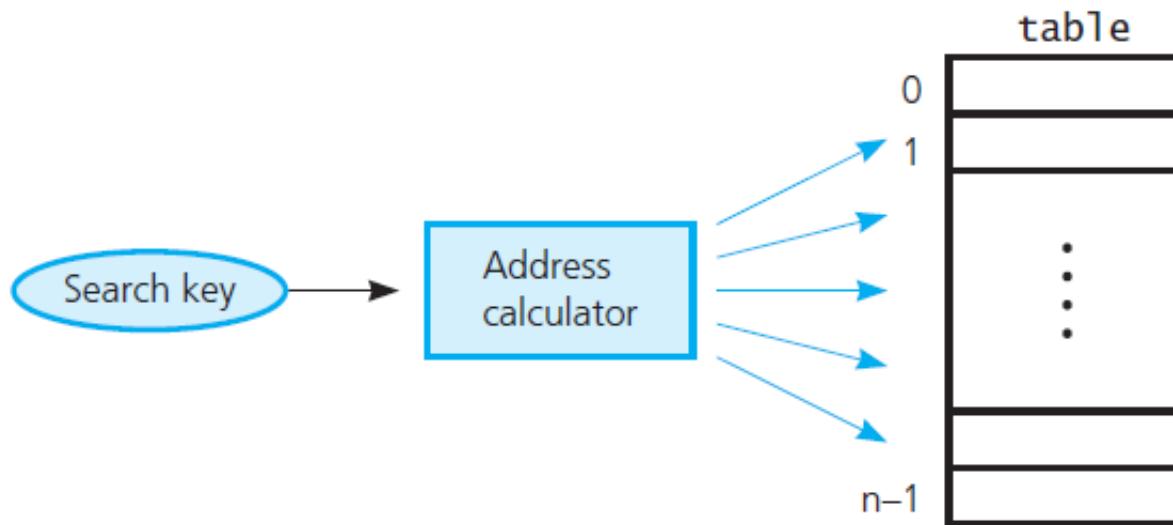
- The proposed system generates a unique number for each string
 - However most strings are not meaningful
 - Given a string containing ten letters there are 2610 possible combinations of letters
 - That is, 141,167,095,653,376 different possible strings
- It is not practical to create an array large enough to store all possible strings
 - Just like the general telephone number problem

Hashing Approach

- We don't need to determine the array size by the maximum possible number of keys but fix the array size based on the amount of data to be stored
 - Map the key value (phone number or name or some other data) to an array element
 - We still need to convert the key value to an integer index using a **hash function**
- This is the basic idea behind **hash tables**

Hashing

- Binary search tree retrieval have order $O(\log_2 n)$. We need a different strategy to locate an item even faster.
- Consider a “magic box” as an address calculator
 - Place/retrieve item from that address in an array



Hash Tables

- A hash table is a data structure that can be used to make map operations faster.
- While worst-case is still $O(n)$, average case is $O(1)$.
- Applications: databases, compilers, browser caches, etc.

Hash Functions and Hash Tables

- A **hash function** h maps keys of a given type to integers in a fixed interval $[0, N - 1]$
- Example: $h(x) = x \bmod N$ is a hash function for integer keys
- The integer $h(x)$ is called the **hash value** of key x
- A **hash table** for a given key type consists of
 - Hash function h
 - Array (called table) of size N
- When implementing a map with a hash table, the goal is to store item (k, o) at index $i = h(k)$



Hash Functions



□ A hash function is usually specified as the composition of two functions:

- Hash code:

h_1 : keys \models integers

- Compression function:

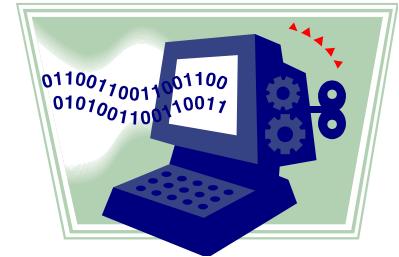
h_2 : integers $\models [0, N - 1]$

The hash code is applied first, and the compression function is applied next on the result, i.e.,

$$\square \quad h(x) = h_2(h_1(x))$$

□ The goal of the hash function is to “disperse” the keys in an apparently random way

Hash Codes



- **Memory address:** We reinterpret the memory address of the key object as an integer (default hash code of all Java objects). Good in general, except for numeric and string keys
- **Integer cast:** We reinterpret the bits of the key as an integer. Suitable for keys of length less than or equal to the number of bits of the integer type (e.g., byte, short, int and float in Java)
- **Component sum:** We partition the bits of the key into components of fixed length (e.g., 16 or 32 bits) and we sum the components (ignoring overflows). Suitable for numeric keys of fixed length greater than or equal to the number of bits of the integer type (e.g., long and double in Java)

Hash Codes (cont.)

- **Polynomial accumulation:** We partition the bits of the key into a sequence of components of fixed length (e.g., 8, 16 or 32 bits) $a_0 a_1 \dots a_{n-1}$. We evaluate the polynomial $p(z) = a_0 + a_1 z + a_2 z^2 + \dots + a_{n-1} z^{n-1}$ at a fixed value z , ignoring overflows.
- Especially suitable for strings (e.g., the choice $z = 33$ gives at most 6 collisions on a set of 50,000 English words)
- Polynomial $p(z)$ can be evaluated in $O(n)$ time using Horner's rule: The following polynomials are successively computed, each from the previous one in $O(1)$ time

$$p_0(z) = a_{n-1}$$

$$p_i(z) = a_{n-i-1} + z p_{i-1}(z) \\ (i = 1, 2, \dots, n - 1)$$

- We have $p(z) = p_{n-1}(z)$

Compression Functions

□ Division: $h_2(y) = y \bmod N$

- The size N of the hash table is usually chosen to be a prime
- The reason is for even distribution of keys and is based on number theory that is beyond the scope of this course

□ Multiply, Add and Divide (MAD): $h_2(y) = [(ay + b) \bmod p] \bmod N$

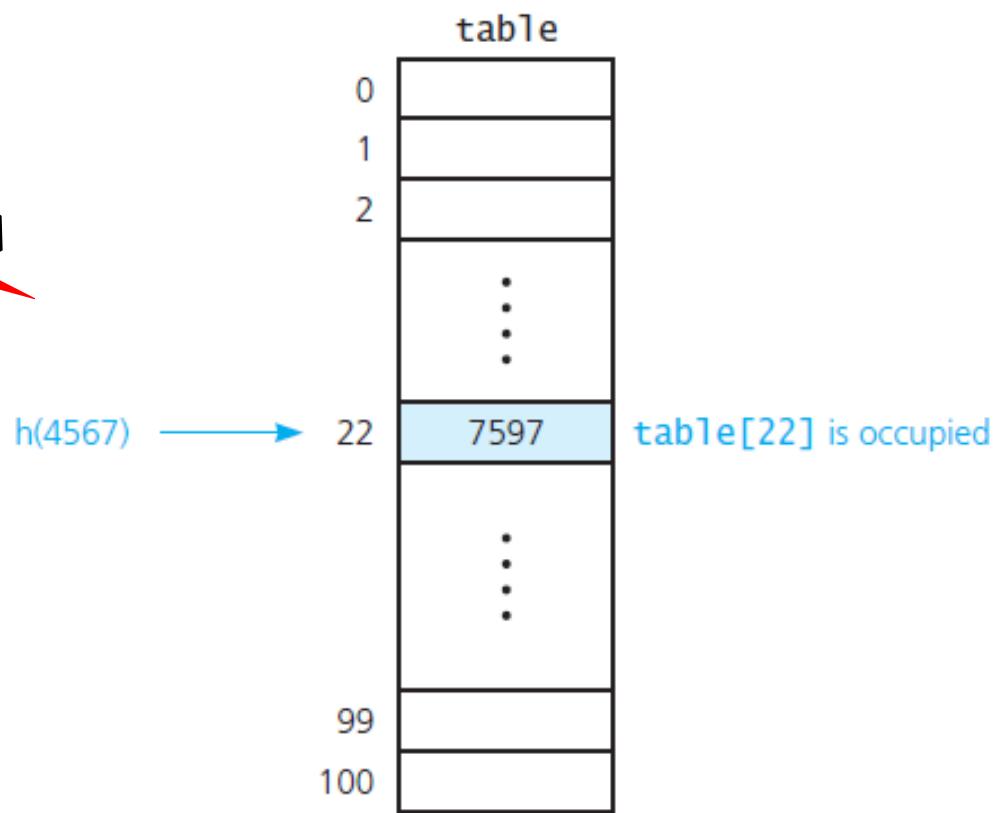
- a and b are nonnegative integers such that p is prime $> N$
- $a > 0$
- Otherwise, every integer would map to the same value b

Scattering Data

- A typical hash function usually results in some collisions
 - A *perfect* hash function avoids collisions entirely
 - Each search key value maps to a different index
 - Only possible when all of the search key values actually stored in the table are known
- The goal is to reduce the number and effect of collisions
- To achieve this the data should be distributed evenly over the table.

Collisions

- ❑ Collisions occur when different elements are mapped to the same cell



Dealing with Collisions

- A collision occurs when two different keys are mapped to the same index
 - Collisions may occur even when the hash function is good
- Two main ways of dealing with collisions
 - Open addressing
 - Separate chaining

Open Addressing

- Idea – when an insertion results in a collision look for an empty array element
 - Start at the index to which the hash function mapped the inserted item
 - Look for a free space in the array following a particular search pattern, known as *probing*
- There are three open addressing schemes
 - Linear probing
 - Quadratic probing
 - Double hashing

Linear Probing

- The hash table is searched sequentially
 - Starting with the original hash location
 - Search $h(\text{search key}) + 1$, then $h(\text{search key}) + 2$, and so on until an available location is found
 - If the sequence of probes reaches the last element of the array, wrap around to $\text{arr}[0]$

Linear Probing Example

- Hash table is size 23
- The hash function, $h = x \bmod 23$, where x is the search key value
- The search key values are shown in the table

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22
						29			32				58								21	

Linear Probing Example

- Insert 81, $h = 81 \bmod 23 = 12$
- Which collides with 58 so use linear probing to find a free space
- First look at 12 + 1, which is free so insert the item at index 13

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22
							29		32			58	81								21	

Linear Probing Example

- Insert 35, $h = 35 \bmod 23 = 12$
- Which collides with 58 so use linear probing to find a free space
- First look at $12 + 1$, which is occupied so look at $12 + 2$ and insert the item at index 14

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22
							29		32			58	81	35							21	

Linear Probing Example

- Insert 60, $h = 60 \bmod 23 = 14$
- Note that even though the key doesn't hash to 12 it still collides with an item that did
- First look at $14 + 1$, which is free

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22
						29			32			58	81	35	60						21	

Linear Probing Example

- Insert 12, $h = 12 \bmod 23 = 12$
- The item will be inserted at index 16
- Notice that “primary clustering” is beginning to develop, making insertions less efficient

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22
							29		32			58	81	35	60	12					21	

Searching

- Searching for an item is similar to insertion
- Find 59 , $h = 59 \bmod 23 = 13$, index 13 does not contain 59 , but is occupied
- Use linear probing to find 59 or an empty space
- Conclude that 59 is not in the table

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22
							29		32			58	81	35	60	12					21	



Linear Probing

- The hash table is searched sequentially
 - Starting with the original hash location
 - Search $h(\text{search key}) + 1$, then $h(\text{search key}) + 2$, and so on until an available location is found
 - If the sequence of probes reaches the last element of the array, wrap around to $\text{arr}[0]$
- Linear probing leads to *primary clustering*
 - The table contains groups of consecutively occupied locations
 - These clusters tend to get larger as time goes on
 - Reducing the efficiency of the hash table

Quadratic Probing

- Quadratic probing is a refinement of linear probing that prevents primary clustering
 - For each successive probe, i , add i^2 to the original location index
 - 1st probe: $h(x)+1^2$, 2nd: $h(x)+2^2$, 3rd: $h(x)+3^2$, etc.

Quadratic Probing Example

- Hash table is size 23
- The hash function, $h = x \bmod 23$, where x is the search key value
- The search key values are shown in the table

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22
						29			32			58									21	

Quadratic Probing Example

- Insert 81, $h = 81 \bmod 23 = 12$
- Which collides with 58 so use quadratic probing to find a free space
- First look at $12 + 1^2$, which is free so insert the item at index 13

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22
							29		32			58	81								21	

Quadratic Probing Example

- Insert 35, $h = 35 \bmod 23 = 12$
- Which collides with 58
- First look at $12 + 1^2$, which is occupied, then look at $12 + 2^2 = 16$ and insert the item there

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22
						29		32			58	81			35						21	

Quadratic Probing Example

- Insert 60, $h = 60 \bmod 23 = 14$
- The location is free, so insert the item

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22
						29		32			58	81	60		35						21	

Quadratic Probing Example

- Insert 12, $h = 12 \bmod 23 = 12$
- First check index $12 + 1^2$,
- Then $12 + 2^2 = 16$,
- Then $12 + 3^2 = 21$ (which is also occupied),
- Then $12 + 4^2 = 28$, wraps to index 5 which is free

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22
					12	29			32			58	81	60		35					21	

The diagram illustrates the quadratic probing sequence for inserting the value 12 at index 5. It shows a 2D array of 23 slots. The first row contains indices 0 through 22. The second row contains the values stored at each index: 12, 29, 32, 58, 81, 60, 35, and 21. Red arrows point from the value 12 in the second row to its position in the first row, then to the value 29, then to 32, then to 58, then to 81, then to 60, and finally to 35. This sequence represents the probe steps: 12+1^2=13 (occupied by 29), 12+2^2=16 (occupied by 32), 12+3^2=21 (occupied by 58), 12+4^2=28 (wraps to index 5, occupied by 81), and 12+5^2=37 (wraps to index 14, occupied by 60), and 12+6^2=48 (wraps to index 22, occupied by 35).

Double Hashing

- In both linear and quadratic probing the probe sequence is independent of the key
- Double hashing produces *key dependent probe sequences*. A second hash function, h_2 , determines the probe sequence
- The second hash function must follow these:
 - $h_2(key) \neq 0$
 - $h_2 \neq h_1$
 - A typical h_2 is $p - (key \bmod p)$ where p is prime

Double Hashing Example

- Hash table is size 23
- The hash function, $h = x \bmod 23$, where x is the search key value
- The second hash function, $h_2 = 5 - (\text{key} \bmod 5)$

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22
						29			32				58								21	

Double Hashing Example

- Insert 81, $h = 81 \bmod 23 = 12$
- Which collides with 58 so use h_2 to find the probe sequence value
- $h_2 = 5 - (81 \bmod 5) = 4$, so insert at $12 + 4 = 16$

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22
							29		32			58				81					21	

Double Hashing Example

- Insert 35, $h = 35 \bmod 23 = 12$
- Which collides with 58 so use h_2 to find a free space
- $h_2 = 5 - (35 \bmod 5) = 5$, so insert at $12 + 5 = 17$

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22
						29			32			58				81	35				21	

Double Hashing Example

- Insert 60, $h = 60 \bmod 23 = 14$

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22
						29		32			58		60		81	35					21	

Double Hashing Example

- Insert 83, $h = 83 \bmod 23 = 14$
- $h_2 = 5 - (83 \bmod 5) = 2$, so insert at $14 + 2 = 16$, which is occupied
- The second probe increments the insertion point by 2 again, so insert at $16 + 2 = 18$

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22
						29		32			58		60		81	35	83			21		

Deletion Example

- Linear probing, $h(x) = x \bmod 23$
- Suppose I want to delete 60
- Any problems?

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22
						29		32			58	81	35		12						21	

Deletions and Open Addressing

- Deletions add complexity to hash tables
 - It is easy to find and delete a particular item
 - But what happens when you want to search for some other item?
 - The recently empty space may make a probe sequence terminate prematurely
- One solution is to mark a table location as either empty, occupied or deleted
 - Locations in the deleted state can be re-used as items are inserted

Deletion Example

- Linear probing, $h(x) = x \bmod 23$
- Suppose I want to delete 60

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22
						29			32			58	81	35	x	12					21	

Deletion Example

- Linear probing, $h(x) = x \bmod 23$
- Search for 12

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	
							29		32			58	81	35	x	12						21	

The diagram shows a linear probing hash table with 23 slots. The slots are indexed from 0 to 22. The values in the slots are: 29 at index 7, 32 at index 8, 58 at index 12, 81 at index 13, 35 at index 14, x at index 15 (the target value), 12 at index 16, and 21 at index 21. Red arrows point from the value 12 at index 15 to the slots at indices 12, 13, 14, 15, and 16, illustrating the probe sequence.

Deletion Example

- Linear probing, $h(x) = x \bmod 23$
- Insert 15

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22
						29		32			58	81	35	15	12						21	

Rehashing

Table size : $M > N$. When $N/M > 0.75$ do
rehashing: Build a second table twice as large as
the original and rehash there all the keys of the
original table.

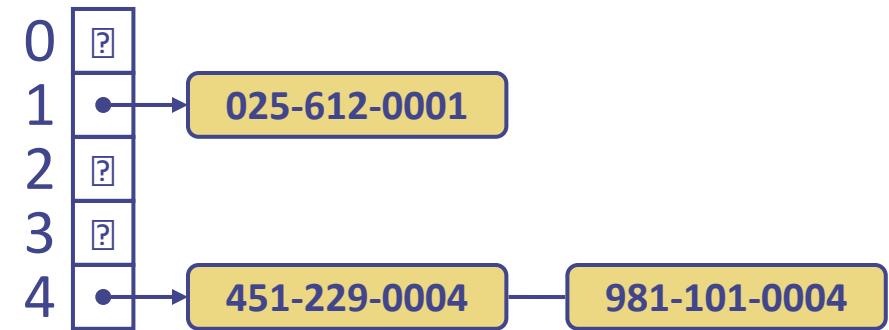
Expensive operation, running time $O(N)$, however,
once done, the new hash table will have good
performance.

Separate Chaining

- A different approach to collisions. Keys hashing to same slot are kept in linked lists attached to that slot.
 - If a collision occurs the new item is added to the end of the list at the appropriate location
- Useful for **highly dynamic situations**, where the number of the search keys cannot be predicted in advance.
- N - number of keys
- M - size of table
- N/M - average length of the lists

Separate Chaining

- Separate Chaining: let each cell in the table point to a linked list of entries that map there



- Separate chaining is simple, but requires additional memory outside the table

Separate Chaining

➤ Advantages

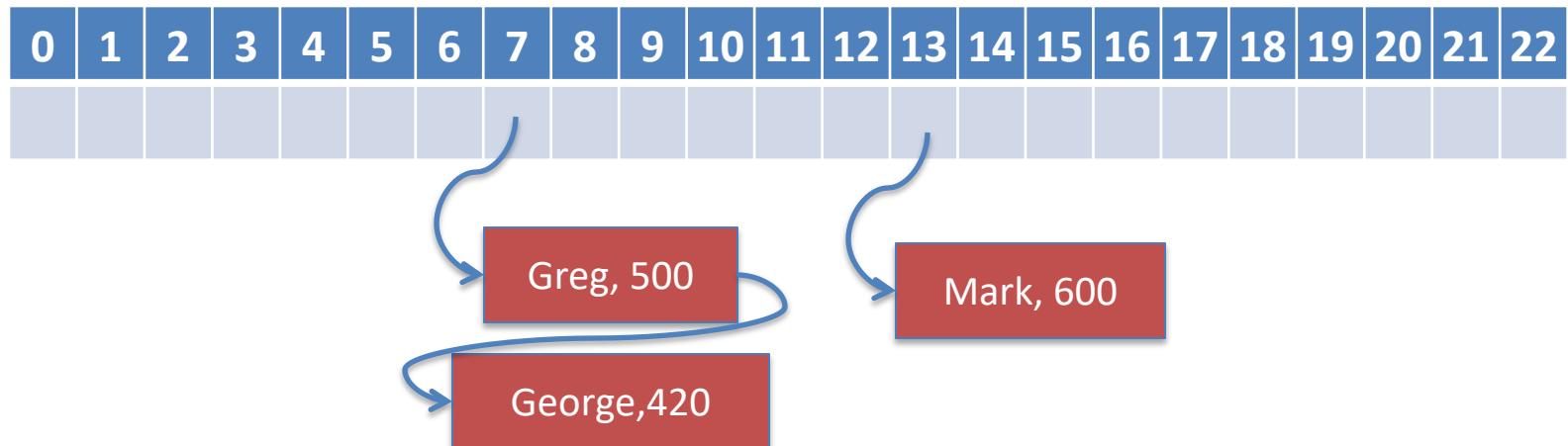
- used when memory space is a concern
- easily implemented

➤ Disadvantages

- unevenly distributed keys – long lists : search time increases, many empty spaces in the table.

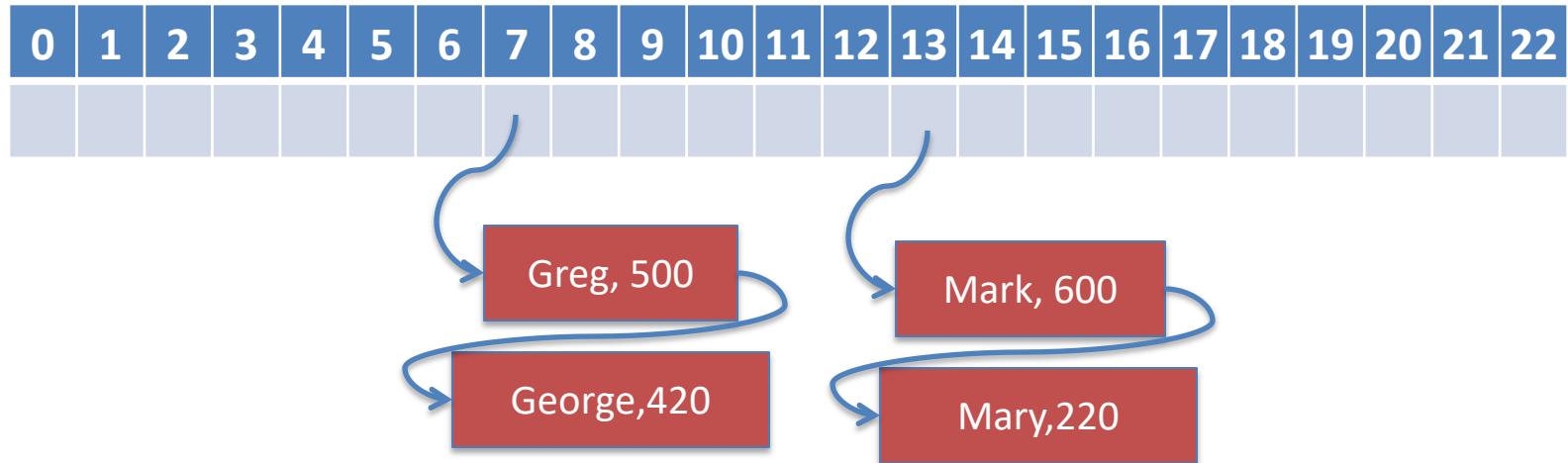
Separate Chaining Example

- Consider Customer data from A3
 - Say we wish to insert $e = \text{Customer}(\text{George}, 420)$
 - Where does it go?
 - $h(e) = 7$ (G is 7th letter in alphabet)



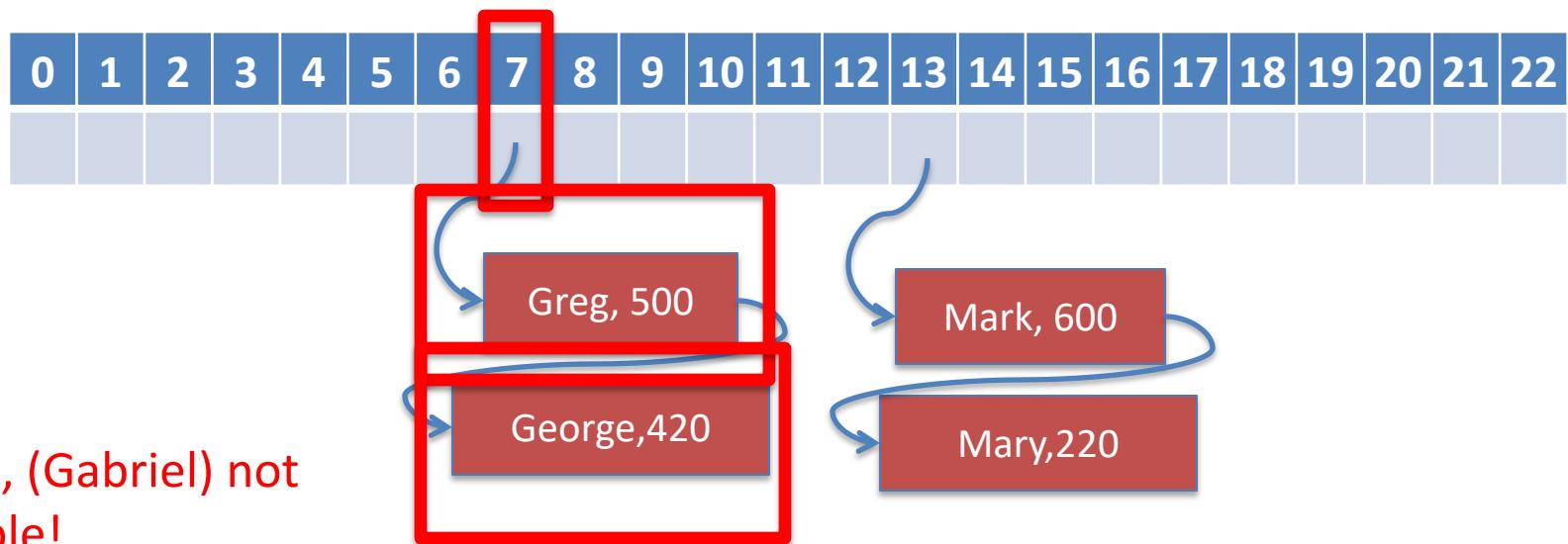
Separate Chaining Example

- Consider Customer data from A3
 - Say we wish to insert $e = \text{Customer}(\text{Mary}, 220)$



Separate Chaining Example

- Consider Customer data from A3
 - Say we wish to find $e = \text{Customer}(\text{Gabriel})$



Hash Table Efficiency

- The load factor $\alpha = \text{number of items} / \text{table size}$.
- As the table fills, α increases, and the chance of a collision occurring also increases. So performance decreases as α increases.
- It is important to base the table size on the largest possible number of items. The table size should be selected so that α does not exceed $2/3$

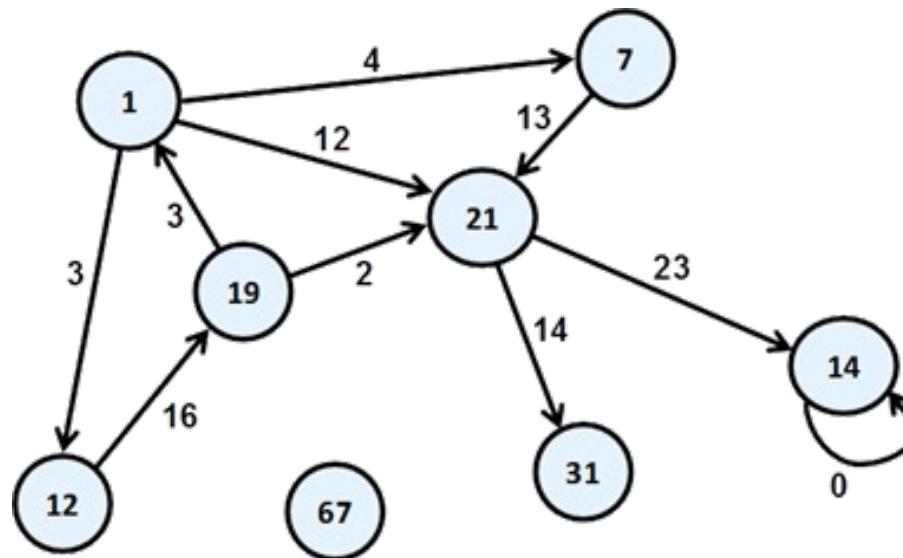
Hash Table Discussion

- If α is less than 0.5 open addressing and separate chaining give similar performance.
- As α increases, separate chaining performs better than open addressing. However, separate chaining increases storage overhead for the linked list pointers.
- In the worst case hash table performance can be poor, if the hash function does not evenly distribute data across the table.
- In the worst case, searches, insertions and removals on a hash table take $O(n)$ time

Performance of Hashing

- The expected running time of all the dictionary ADT operations in a hash table is $O(1)$
- In practice, hashing is very fast provided the load factor is not close to 100%
- Applications of hash tables:
 - small databases
 - compilers
 - browser caches

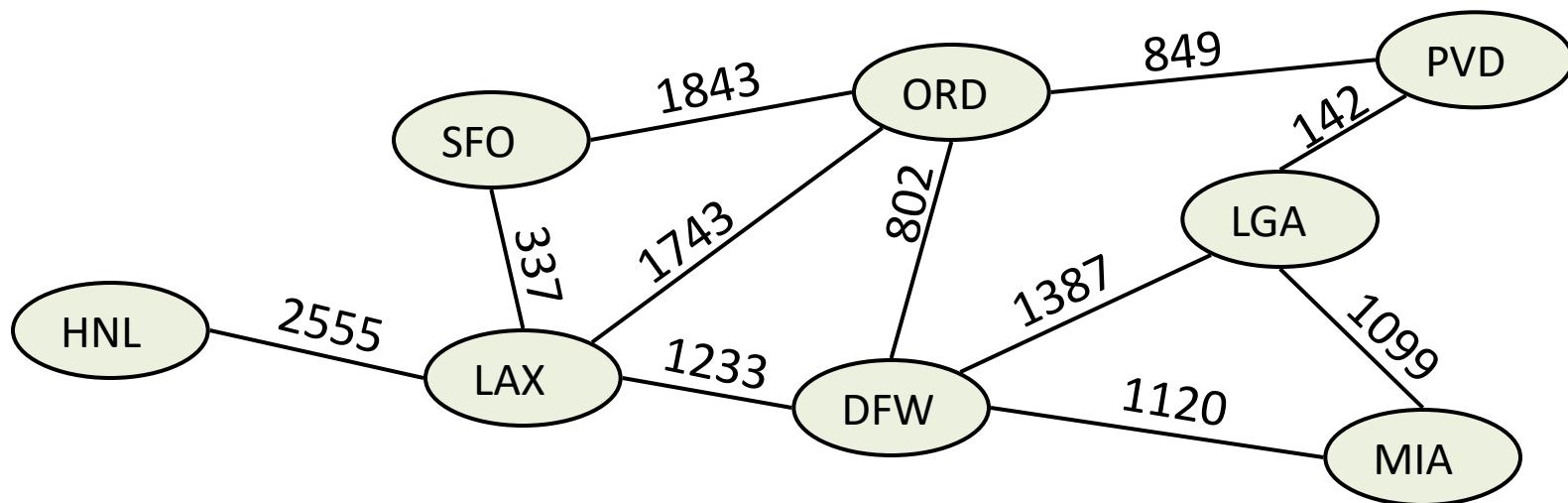
Graph Data Structure



Slides partly from Data Structures and Algorithms in Java (Goodrich, et. al.), Data Structures and Algorithm Analysis in Java (M. A. Weiss)

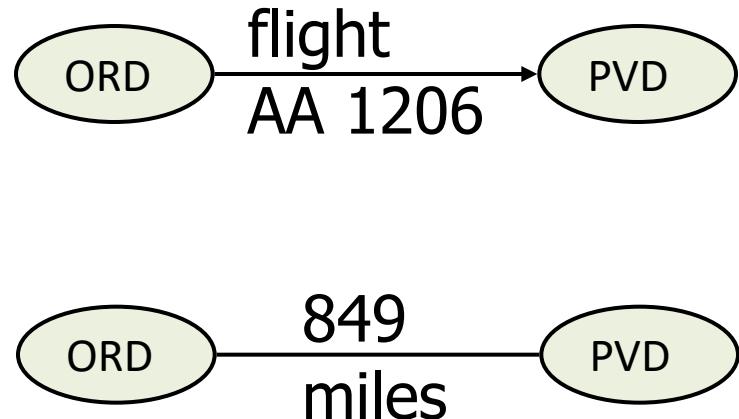
Graphs

- A graph G is a pair (V, E) , where V is a set of nodes, called **vertices**, and E is a collection of pairs of vertices, called **edges**. Vertices and edges can store elements
- Example: A vertex represents an airport and stores the three-letter airport code. An edge represents a flight route between two airports and stores the mileage of the route



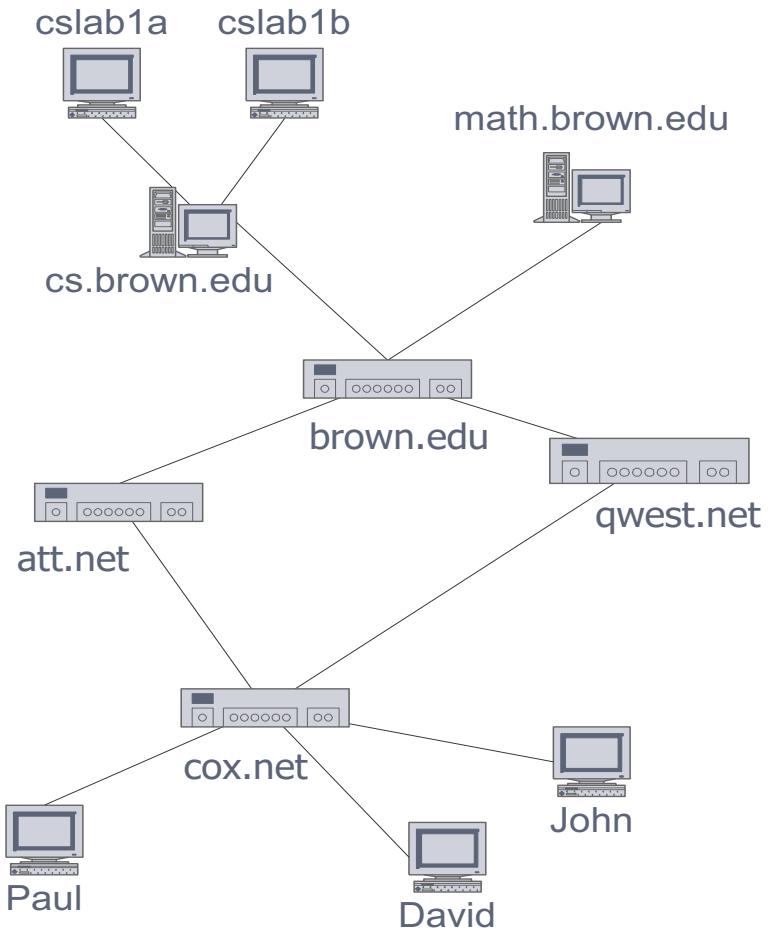
Edge Types

- Directed edge
 - ordered pair of vertices (u,v)
 - first vertex u is the origin
 - second vertex v is the destination
 - e.g., a flight
- Undirected edge
 - unordered pair of vertices (u,v)
 - e.g., a flight route
- Directed graph
 - all the edges are directed
 - e.g., flight network
- Undirected graph
 - all the edges are undirected
 - e.g., route network



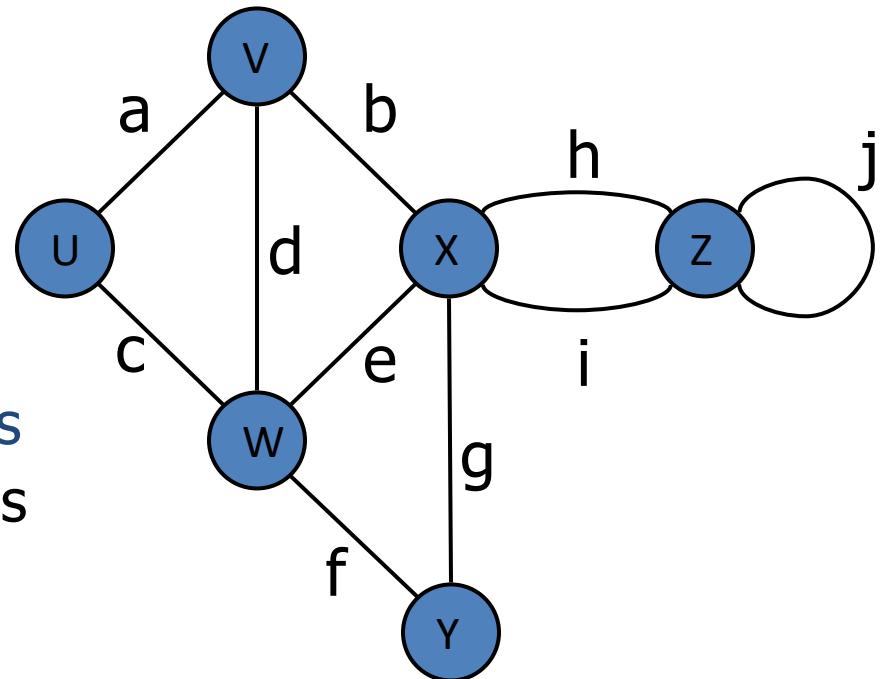
Applications

- Transportation networks
 - Highway network
 - Flight network
- Computer networks
 - Local area network
 - Internet
 - Web
- Electronic circuits
 - Printed circuit board
 - Integrated circuit
- Databases
 - Entity-relationship diagram
- Operations
 - Task scheduling
 - Network management



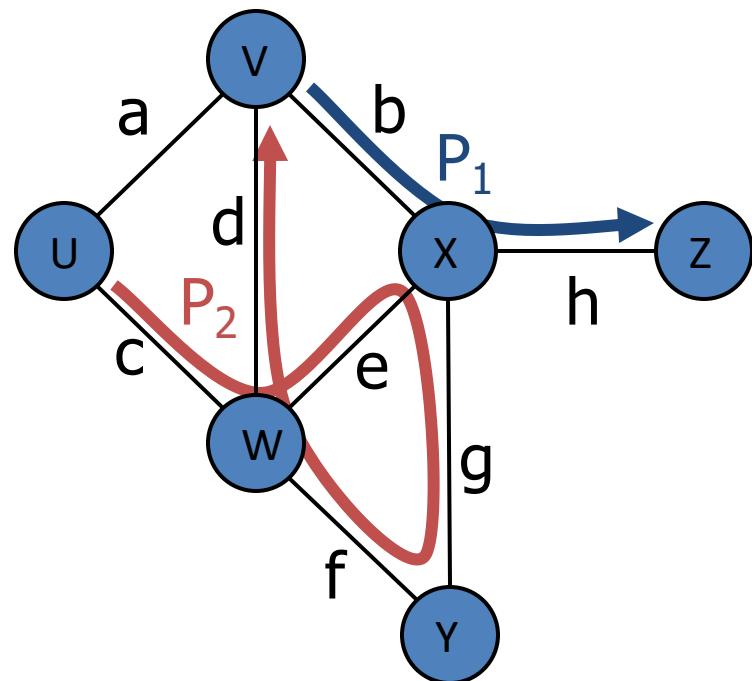
Terminology

- **Adjacent** vertices
 - U and V are adjacent
- **Degree** of a vertex
 - X has degree 5
- **Parallel edges and self-loops**
 - h and i are parallel edges
 - j is a self-loop



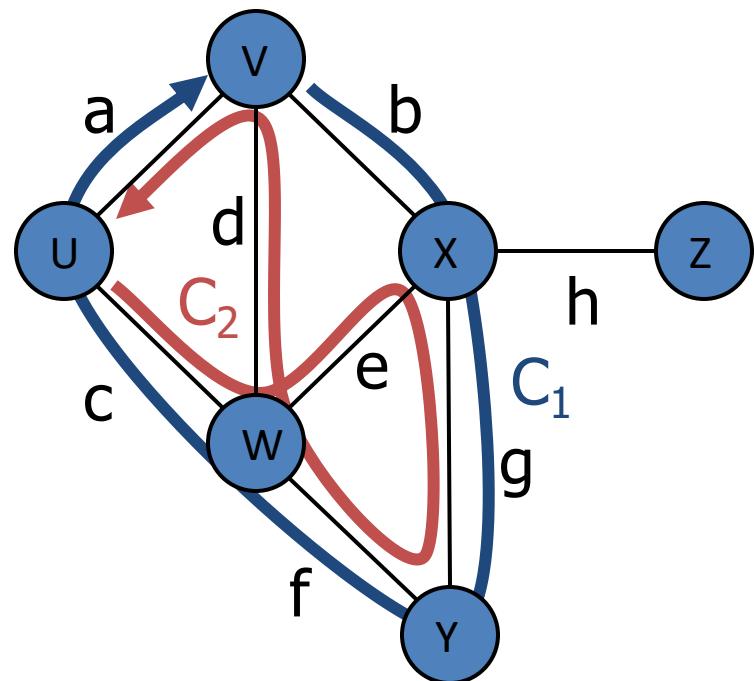
Terminology (cont.)

- Path
 - sequence of alternating vertices and edges
 - begins with a vertex
 - ends with a vertex
 - each edge is preceded and followed by its endpoints
- Simple path
 - path such that all its vertices and edges are **distinct**
 - $P_1=(V,b,X,h,Z)$ is a simple path
 - $P_2=(U,c,W,e,X,g,Y,f,W,d,V)$ is a path that is not simple



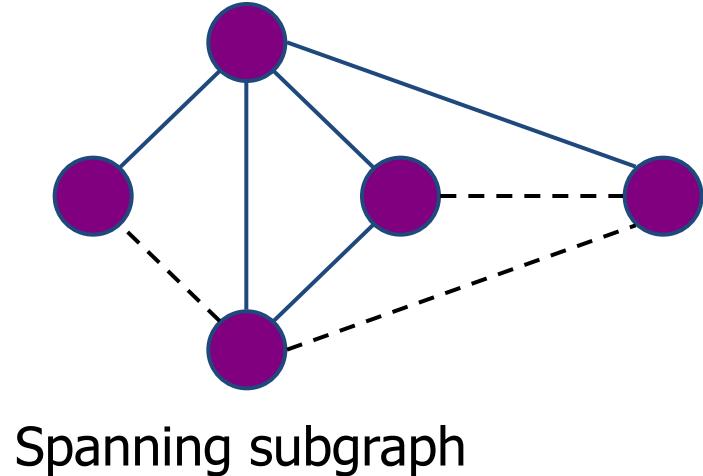
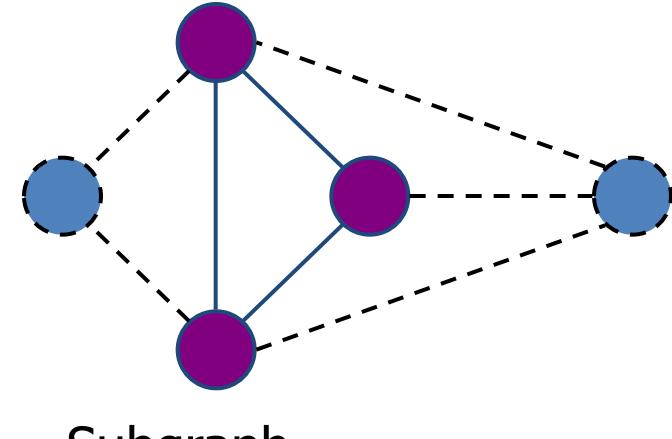
Terminology (cont.)

- Cycle
 - circular sequence of alternating vertices and edges
 - each edge is preceded and followed by its endpoints
 - Simple cycle
 - cycle such that all its vertices and edges are distinct
 - Examples
 - $C_1 = (V, b, X, g, Y, f, W, c, U, a, \leftarrow)$ is a simple cycle
 - $C_2 = (U, c, W, e, X, g, Y, f, W, d, V, a, \leftarrow)$ is a cycle that is not simple



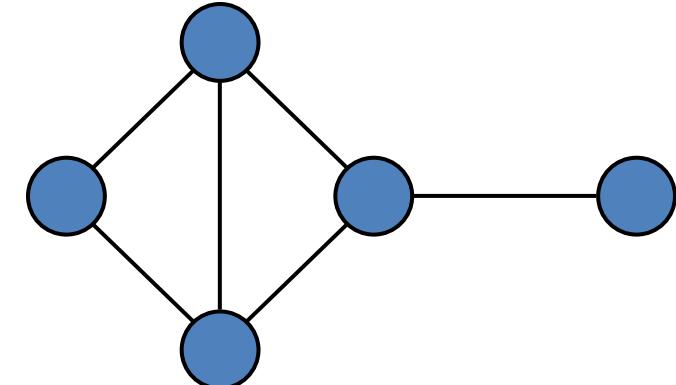
Subgraphs

- A subgraph S of a graph G is a graph such that
 - The vertices of S are a subset of the vertices of G
 - The edges of S are a subset of the edges of G
- A spanning subgraph of G is a subgraph that contains all the vertices of G

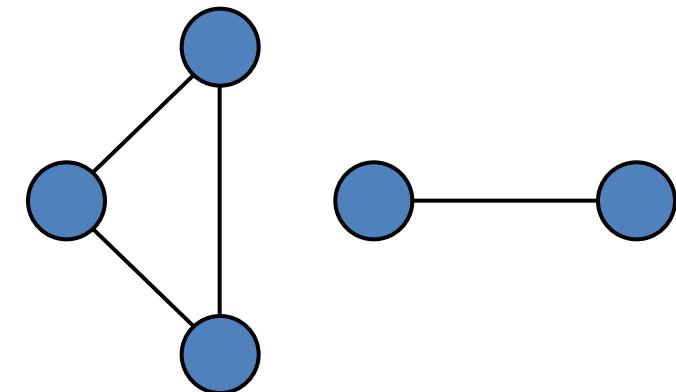


Connectivity

- A graph is connected if there is a path between every pair of vertices
- A connected component of a graph G is a maximal connected subgraph of G



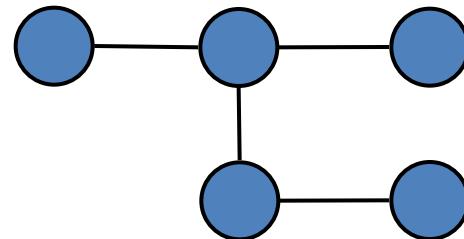
Connected graph



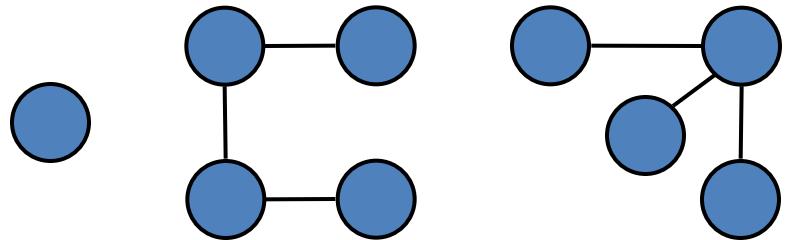
Non connected graph with two connected components

Trees and Forests

- A tree is an undirected graph T such that
 - T is connected
 - T has no cycles
- This definition of tree is different from the one of a rooted tree
- A forest is an undirected graph without cycles
- The connected components of a forest are trees



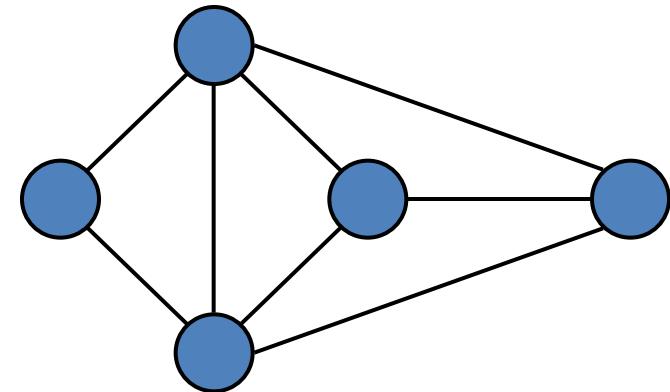
Tree



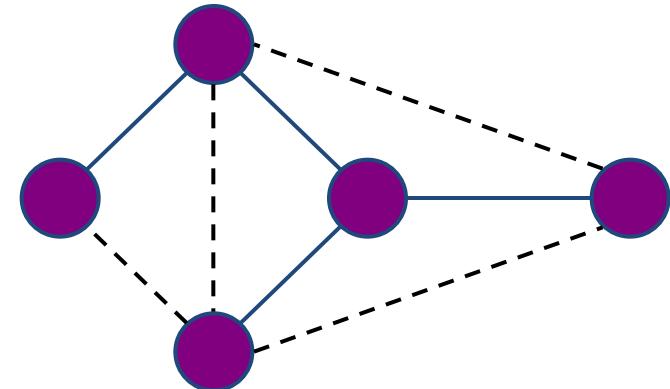
Forest

Spanning Trees and Forests

- A spanning tree of a connected graph is a spanning subgraph that is a tree
- A spanning tree is not unique unless the graph is a tree
- Spanning trees have applications to the design of communication networks
- A spanning forest of a graph is a spanning subgraph that is a forest



Graph



Spanning tree

Properties

Property 1

$$\sum_v \deg(v) = 2m$$

Proof: each edge is counted twice

Notation

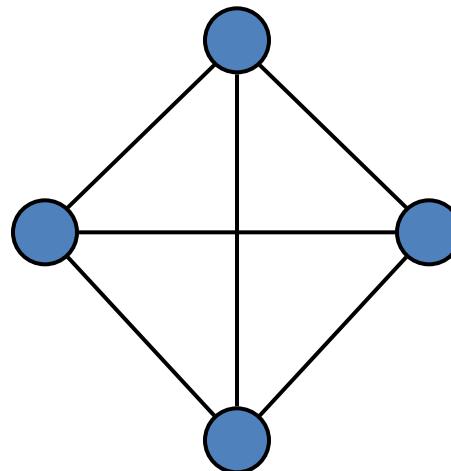
n	number of vertices
m	number of edges
$\deg(v)$	degree of vertex v

Property 2

In an undirected graph with no self-loops and no parallel edges

$$m \leq n(n - 1)/2$$

$$\text{Proof: } (n-1) + (n-2) + \dots + 1$$



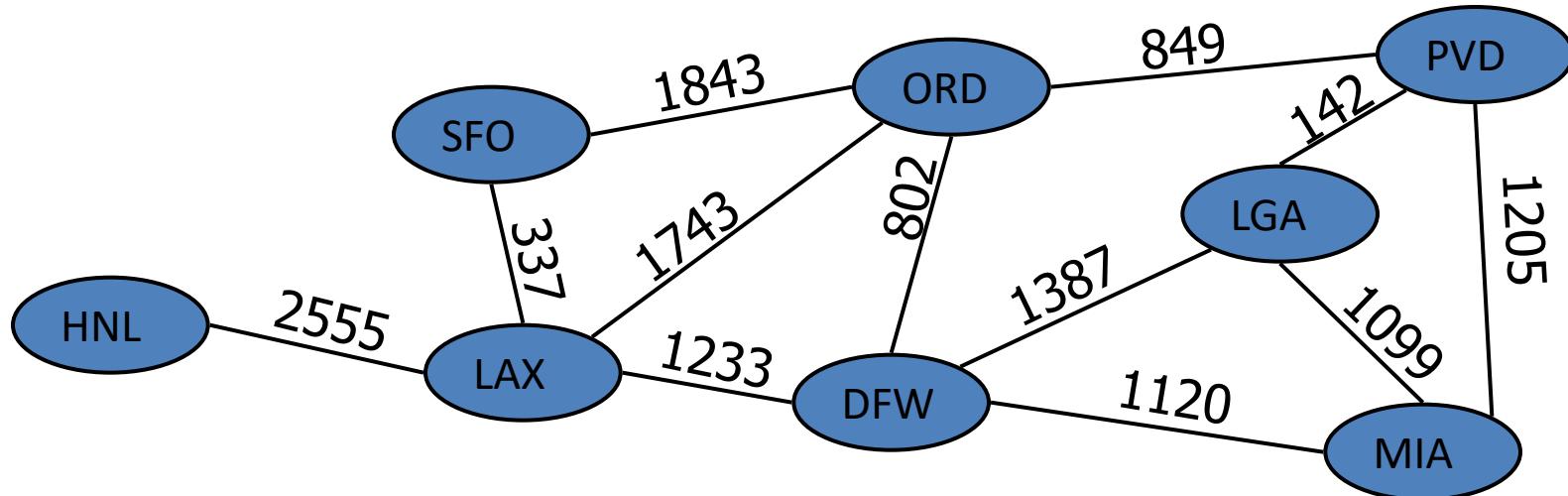
Example

- $n = 4$
- $m = 6$
- $\deg(v) = 3$

What is the bound for a directed graph?

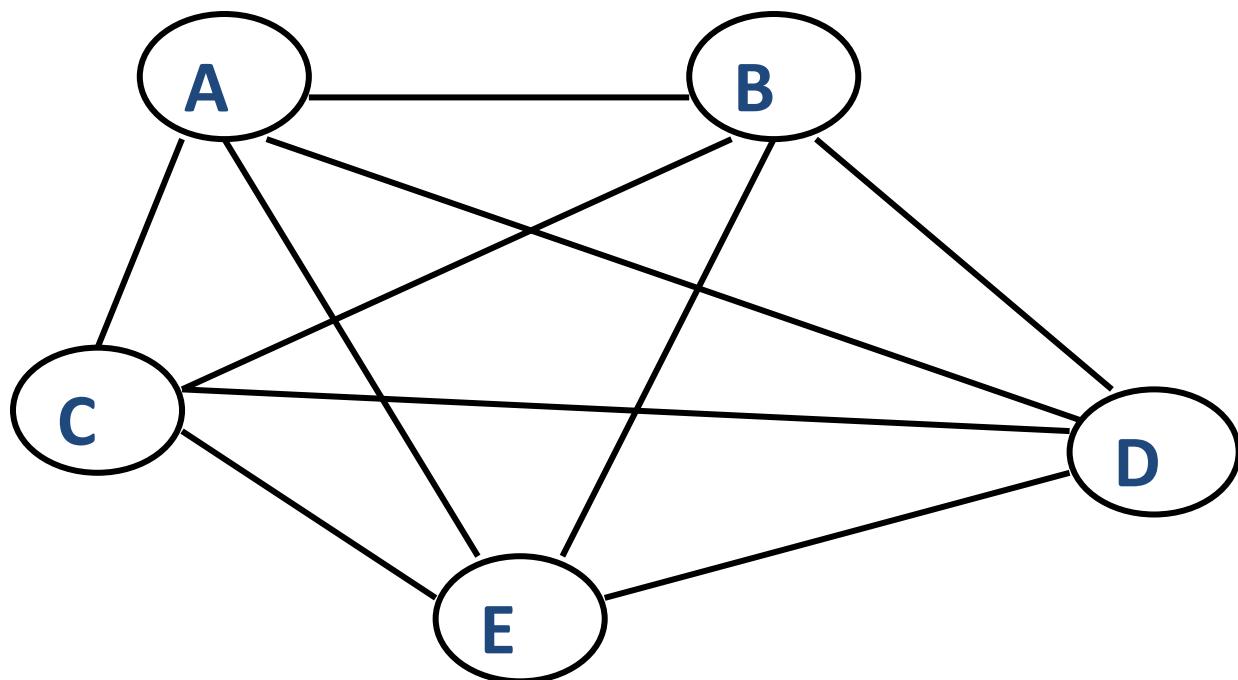
Weighted Graphs

- In a weighted graph, each edge has an associated numerical value, called the weight of the edge
- Edge weights may represent, distances, costs, etc.
- Example:
 - In a flight route graph, the weight of an edge represents the distance in miles between the endpoint airports



Complete graphs

Graphs with all edges present



A complete graph

Dense graphs:
relatively few of
the possible edges
are missing

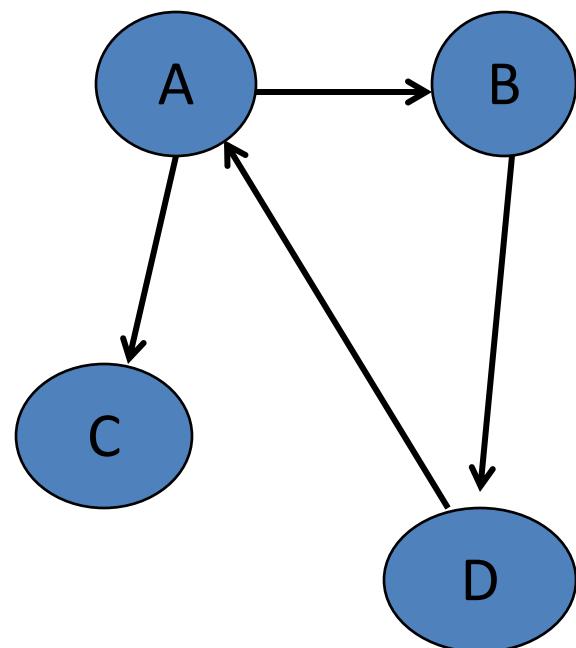
Sparse graphs:
relatively few of
the possible edges
are present

Adjacency matrix

Vertices: A,B,C,D

Edges: AC, AB, BD, DA

	A	B	C	D
A	0	1	1	0
B	0	0	0	1
C	0	0	0	0
D	1	0	0	0



Adjacency lists

Vertices: A,B,C,D

Edges: AC, AB, BD, DA

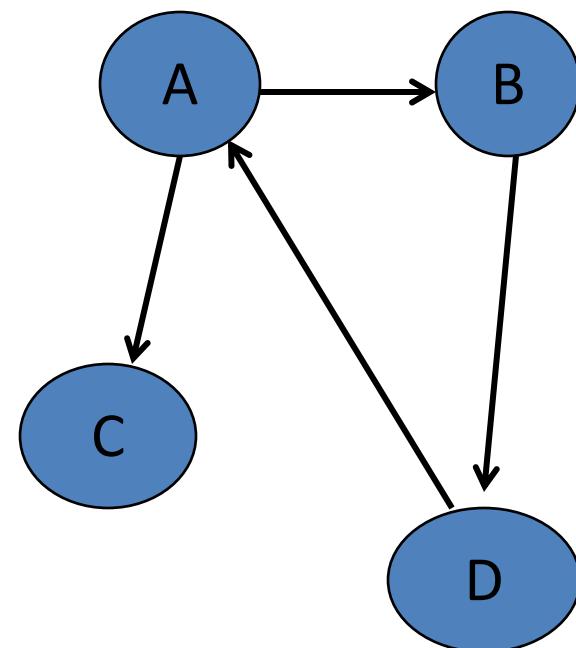
Heads lists

A B C

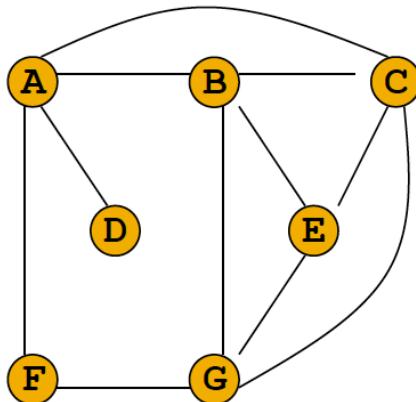
B D

C =

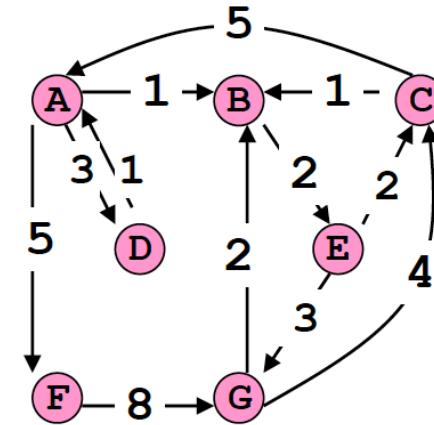
D A



Adjacency List



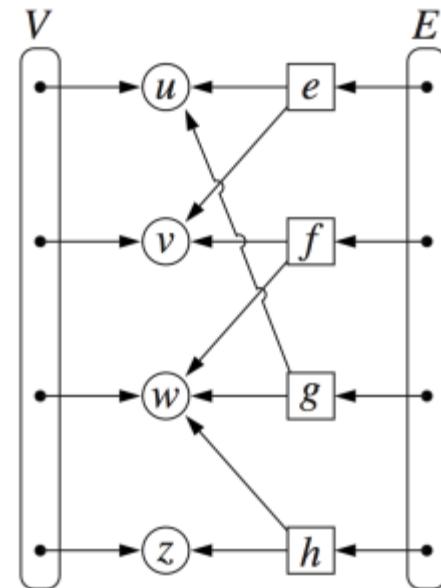
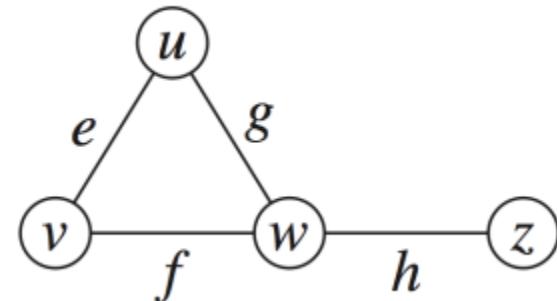
A	B	→	C	→	D	→	F	
B	A	→	C	→	E	→	G	
C	A	→	B	→	E	→	G	
D	A							
E	B	→	C	→	G			
F	A	→	G					
G	B	→	C	→	E	→	F	



A	B	1	→	D	3	→	F	5	
B	E	2							
C	A	5	→	B	1				
D	A	1							
E	C	2	→	G	3				
F	G	8							
G	B	2	→	C	4				

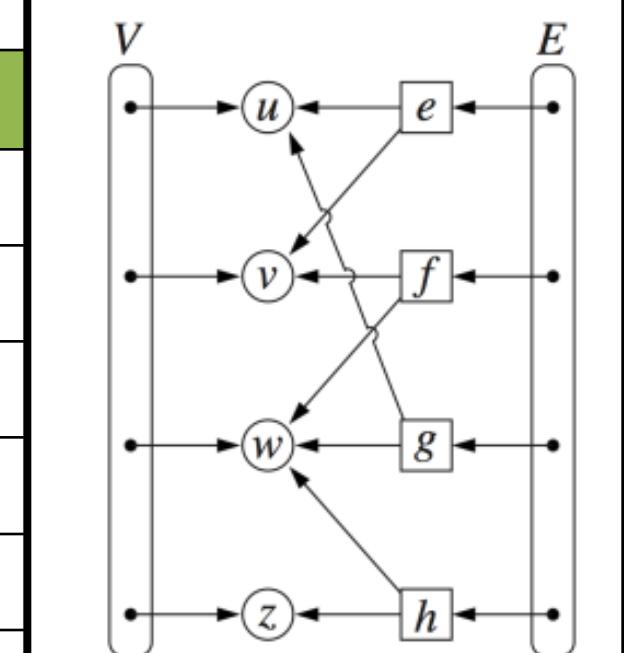
Edge List Structure

- Vertex object
 - element
 - reference to position in vertex sequence
- Edge object
 - element
 - origin vertex object
 - destination vertex object
 - reference to position in edge sequence
- Vertex sequence
 - sequence of vertex objects
- Edge sequence
 - sequence of edge objects



Complexity in big-O

<ul style="list-style-type: none"> ▪ n vertices, m edges ▪ no parallel edges ▪ no self-loops 	Edge List	Adjacency List	Adjacency Matrix
Space	$n + m$		
<code>incidentEdges(v)</code>	m		
<code>areAdjacent (v, w)</code>	m		
<code>insertVertex(v)</code>	1		
<code>insertEdge(v, w, x)</code>	1^*		
<code>removeVertex(v)</code>	m^*		
<code>removeEdge(e)</code>	1^*		

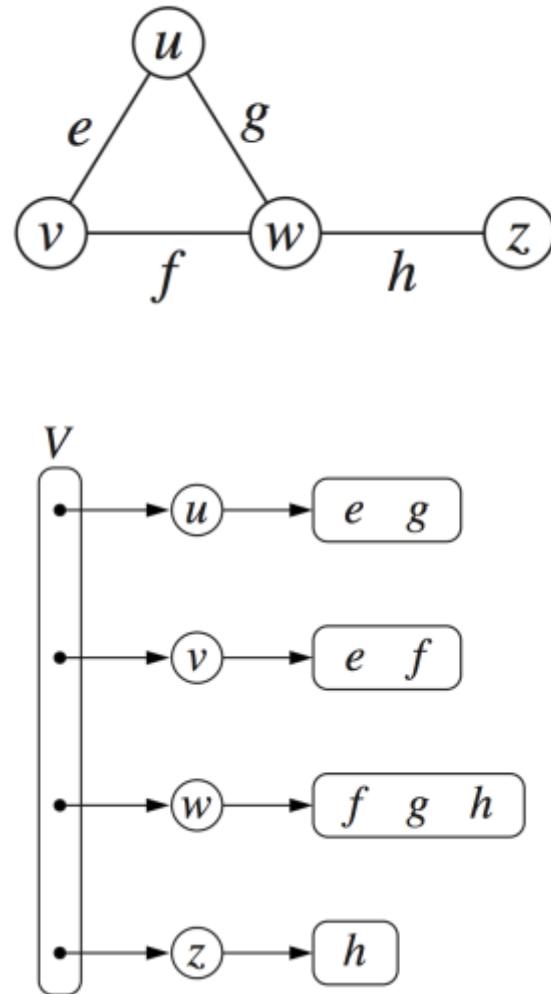


* Assuming locating an edge or vertex is constant time

Adjacency List Structure

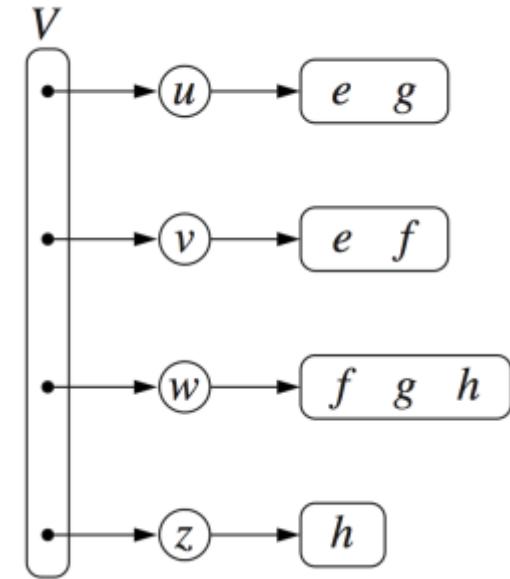
- Incidence sequence for each vertex
 - sequence of references to edge objects of incident edges
- Augmented edge objects
 - references to associated positions in incidence sequences of end vertices

Vertices or edges in each list



Complexity in big

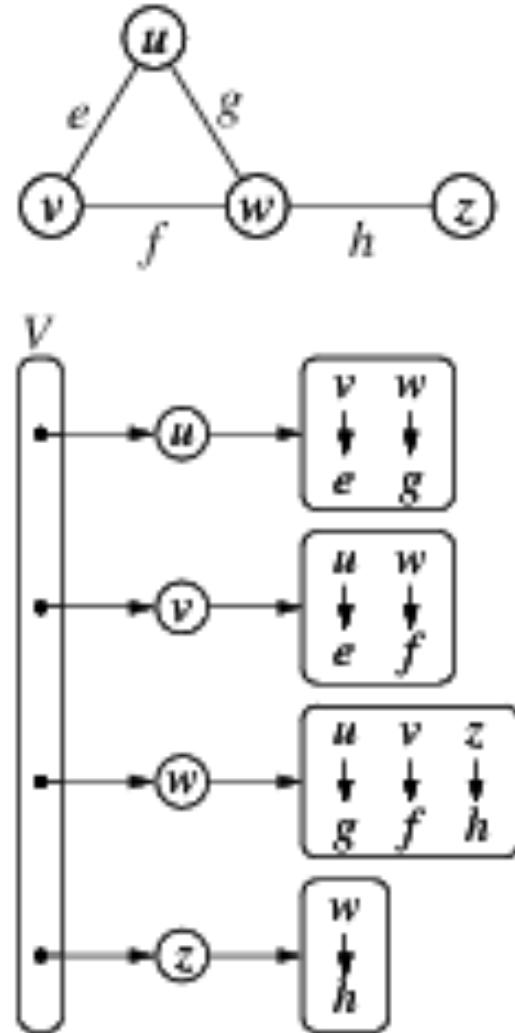
n vertices, m edges ■ $\deg(v)$ is degree of v	Edge List	Adjacency List	
Space	$n + m$	$n + m$	
<code>incidentEdges(v)</code>	m	$\deg(v)$	
<code>areAdjacent (v, w)</code>	m	$\min(\deg(v), \deg(w))$	
<code>insertVertex(v)</code>	1	1	
<code>insertEdge(v, w, x)</code>	1	1	
<code>removeVertex(v)</code>	m	$\deg(v)$	
<code>removeEdge(e)</code>	1	1*	



*Assuming location of edge is known

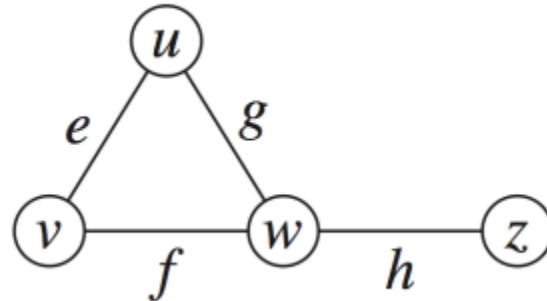
Adjacency Map

- Similar to Adjacency List
 - each list is replaced with a map
 - key is vertex
 - $\text{areAdjacent}(u,v)$ is $O(1)$
 - More space



Adjacency Matrix Structure

- Edge list structure
- Augmented vertex objects
 - Integer key (index) associated with vertex
- 2D-array adjacency array
 - Reference to edge object for adjacent vertices
 - Null for non nonadjacent vertices
- The “old fashioned” version just has 0 for no edge and 1 for edge



	0	1	2	3
<i>u</i>	→ 0	<i>e</i>	<i>g</i>	
<i>v</i>	→ 1	<i>e</i>	<i>f</i>	
<i>w</i>	→ 2	<i>g</i>	<i>f</i>	<i>h</i>
<i>x</i>	→ 3		<i>h</i>	

Complexity in big-O

<ul style="list-style-type: none"> ▪ n vertices, m edges ▪ $\deg(v)$ is degree of v 	Edge List	Adjacency List	Adjacency Matrix																									
Space	$n + m$	$n + m$	n^2																									
<code>incidentEdges(v)</code>		<table border="1" style="margin-left: auto; margin-right: auto;"> <thead> <tr> <th></th> <th>0</th> <th>1</th> <th>2</th> <th>3</th> </tr> </thead> <tbody> <tr> <td>$u \rightarrow 0$</td> <td></td> <td>e</td> <td>g</td> <td></td> </tr> <tr> <td>$v \rightarrow 1$</td> <td>e</td> <td></td> <td>f</td> <td></td> </tr> <tr> <td>$w \rightarrow 2$</td> <td>g</td> <td>f</td> <td></td> <td>h</td> </tr> <tr> <td>$z \rightarrow 3$</td> <td></td> <td></td> <td>h</td> <td></td> </tr> </tbody> </table>		0	1	2	3	$u \rightarrow 0$		e	g		$v \rightarrow 1$	e		f		$w \rightarrow 2$	g	f		h	$z \rightarrow 3$			h		n
	0	1	2	3																								
$u \rightarrow 0$		e	g																									
$v \rightarrow 1$	e		f																									
$w \rightarrow 2$	g	f		h																								
$z \rightarrow 3$			h																									
<code>areAdjacent (v, w)</code>		1																										
<code>insertVertex(v)</code>		n^2																										
<code>insertEdge(v, w, x)</code>		1																										
<code>removeVertex(v)</code>		n^2																										
<code>removeEdge(e)</code>	1	1																										

DAGs and Topological Ordering

- A directed acyclic graph (DAG) is a digraph that has no directed cycles
- A topological ordering of a digraph is a numbering

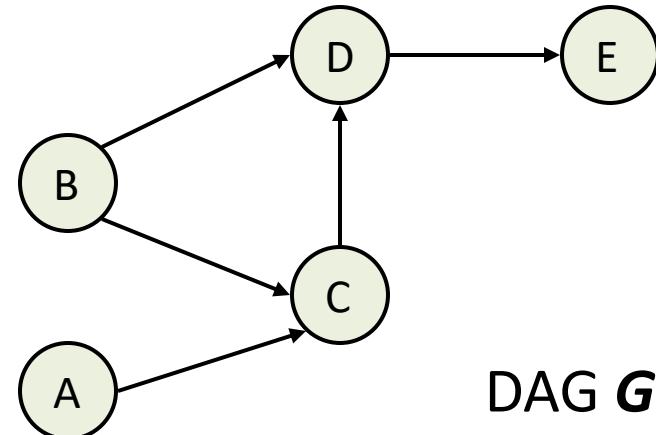
$$v_1, \dots, v_n$$

of the vertices such that for every edge (v_i, v_j) , we have $i < j$

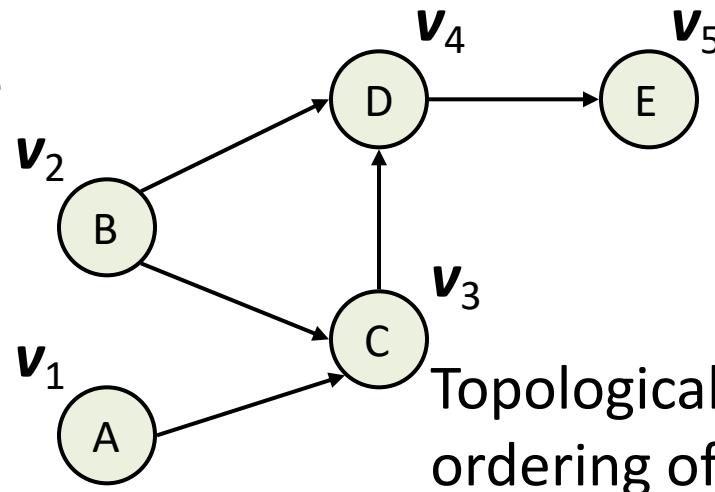
- Example: in a task scheduling digraph, a topological ordering a task sequence that satisfies the precedence constraints

Theorem

A digraph admits a topological ordering if and only if it is a DAG



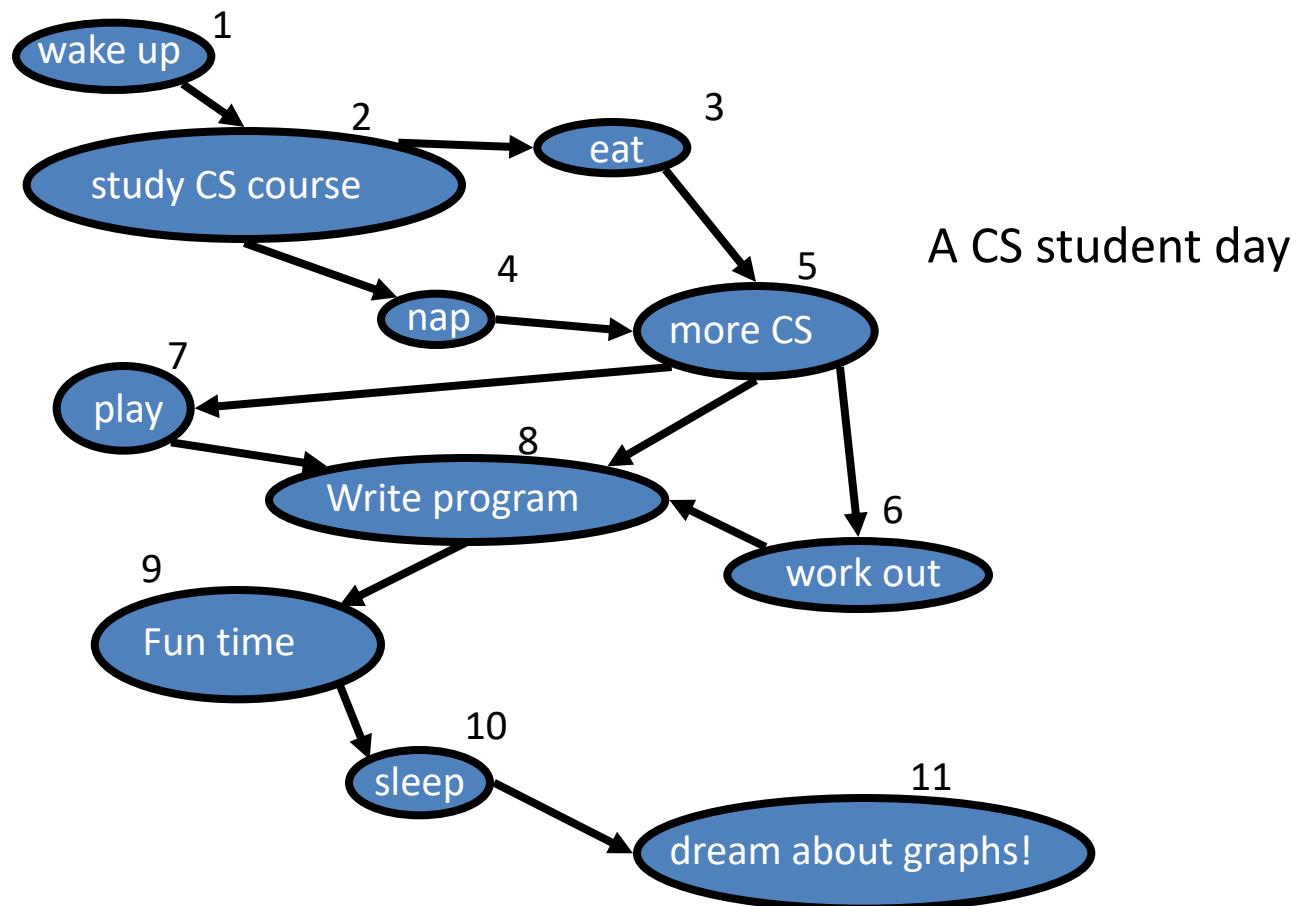
DAG **G**



Topological ordering of **G**

Topological Sorting

- Number vertices, so that (u,v) in E implies $u < v$



Topological Sort

RULE: If there is a path from u to v , then v appears after u in the ordering.

Graphs: directed, acyclic (DAG)

OutDegree of a vertex U : the number of **outgoing** edges

Indegree of a vertex U : the number of **incoming** edges

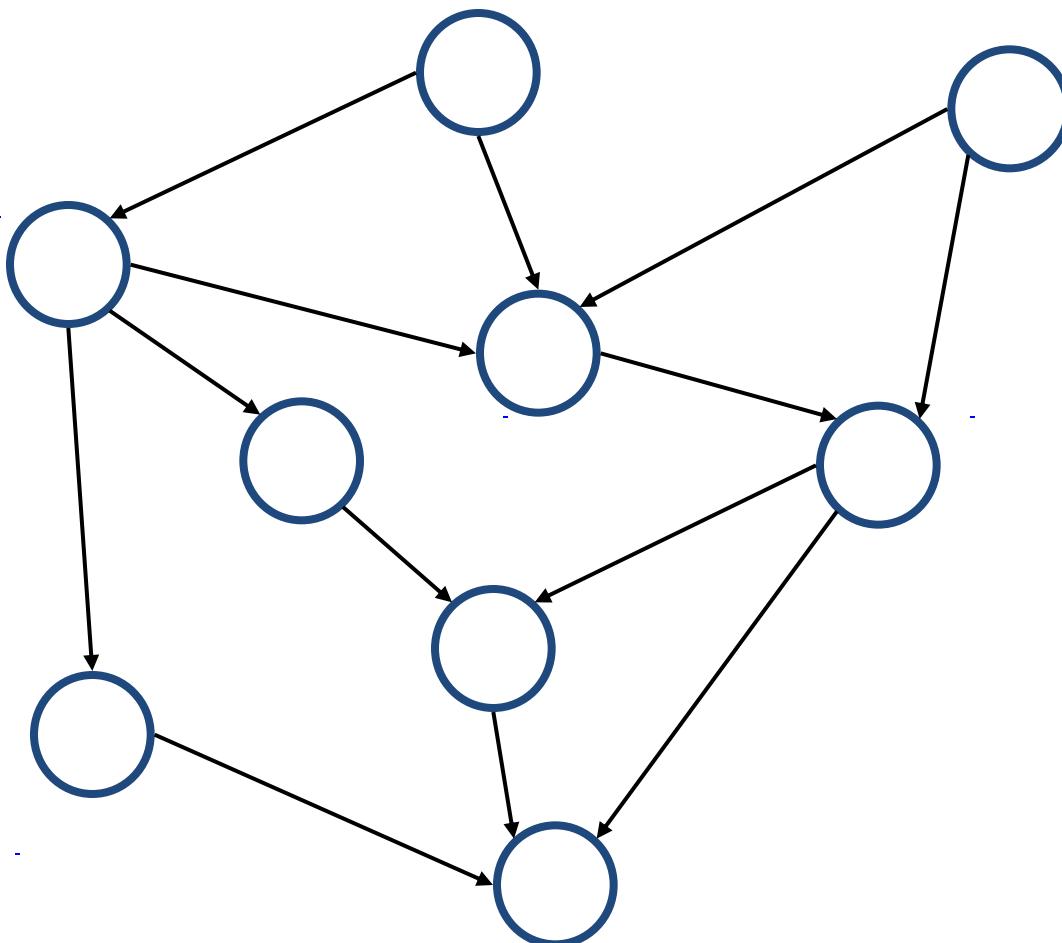
The algorithm for topological sort uses "indegrees" of vertices.

Implementation: with a queue

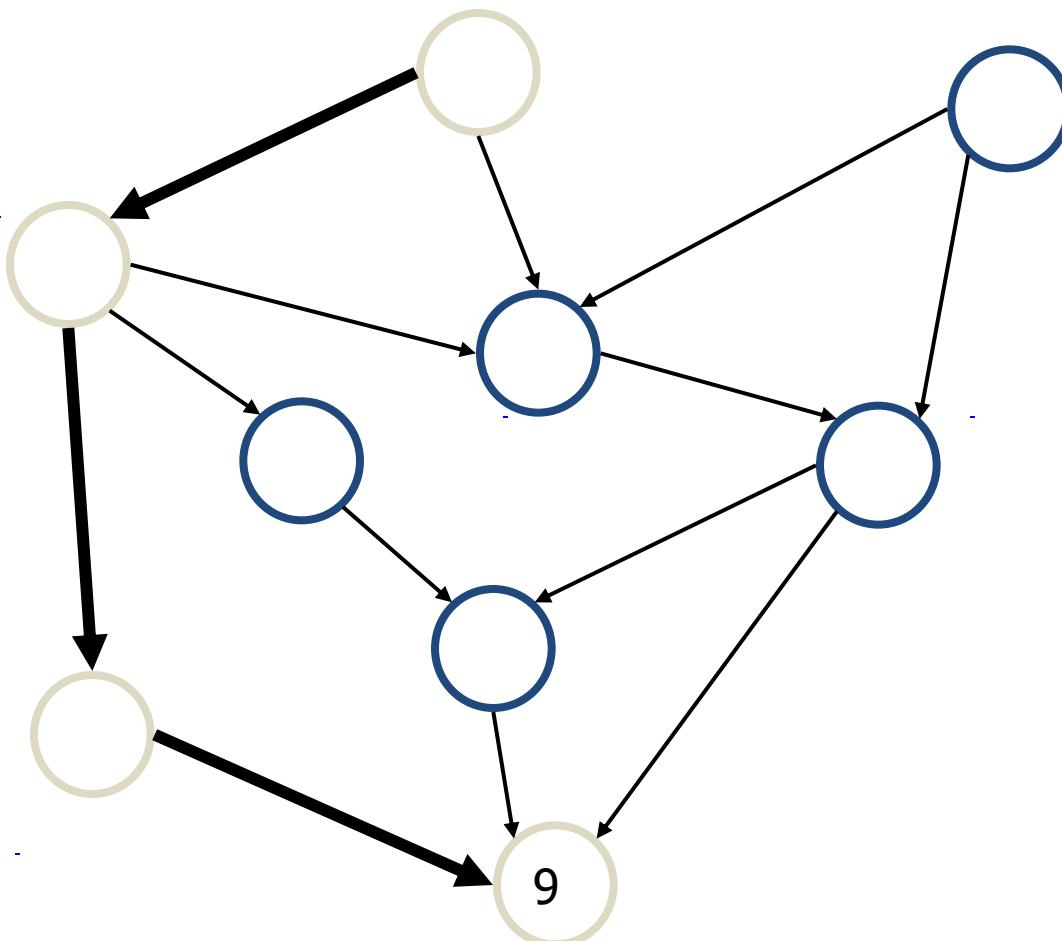
Complexity: Adjacency lists: $O(|E| + |V|)$,

Matrix representation: $O(|V|^2)$

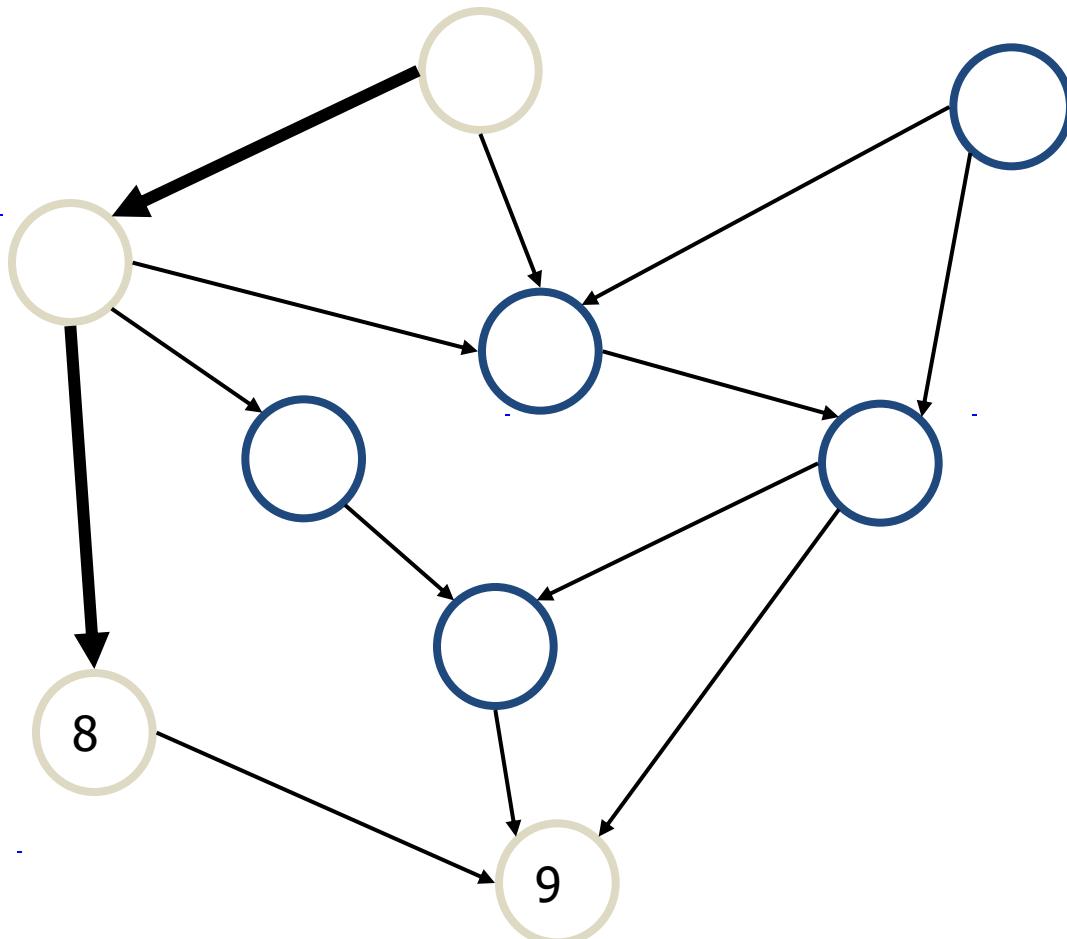
Topological Sorting Example



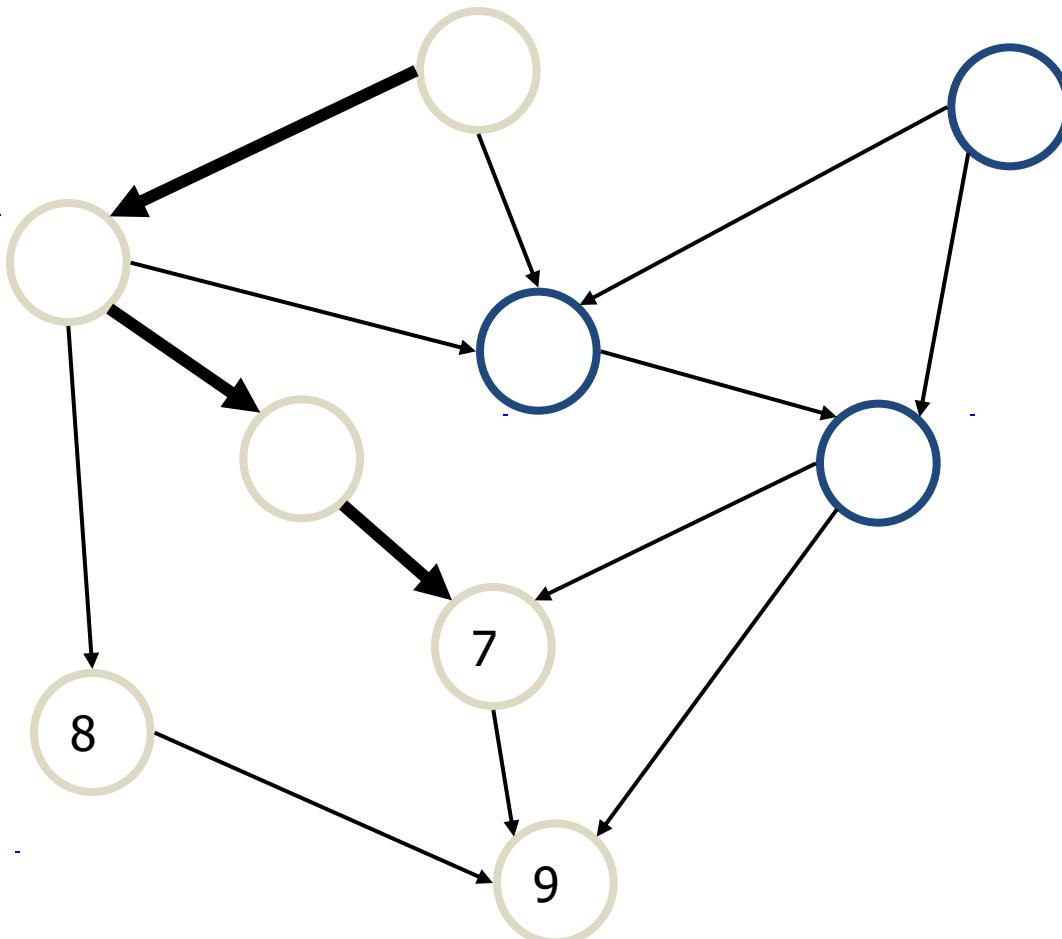
Topological Sorting Example



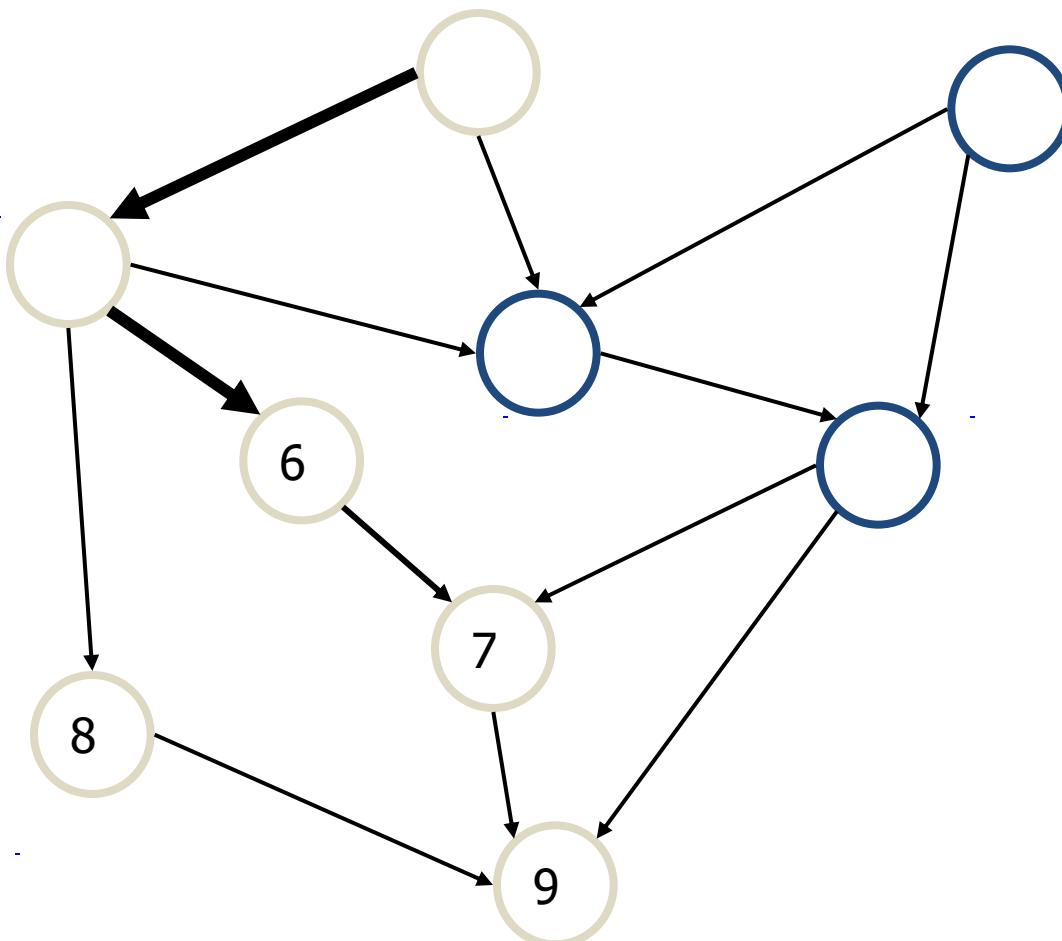
Topological Sorting Example



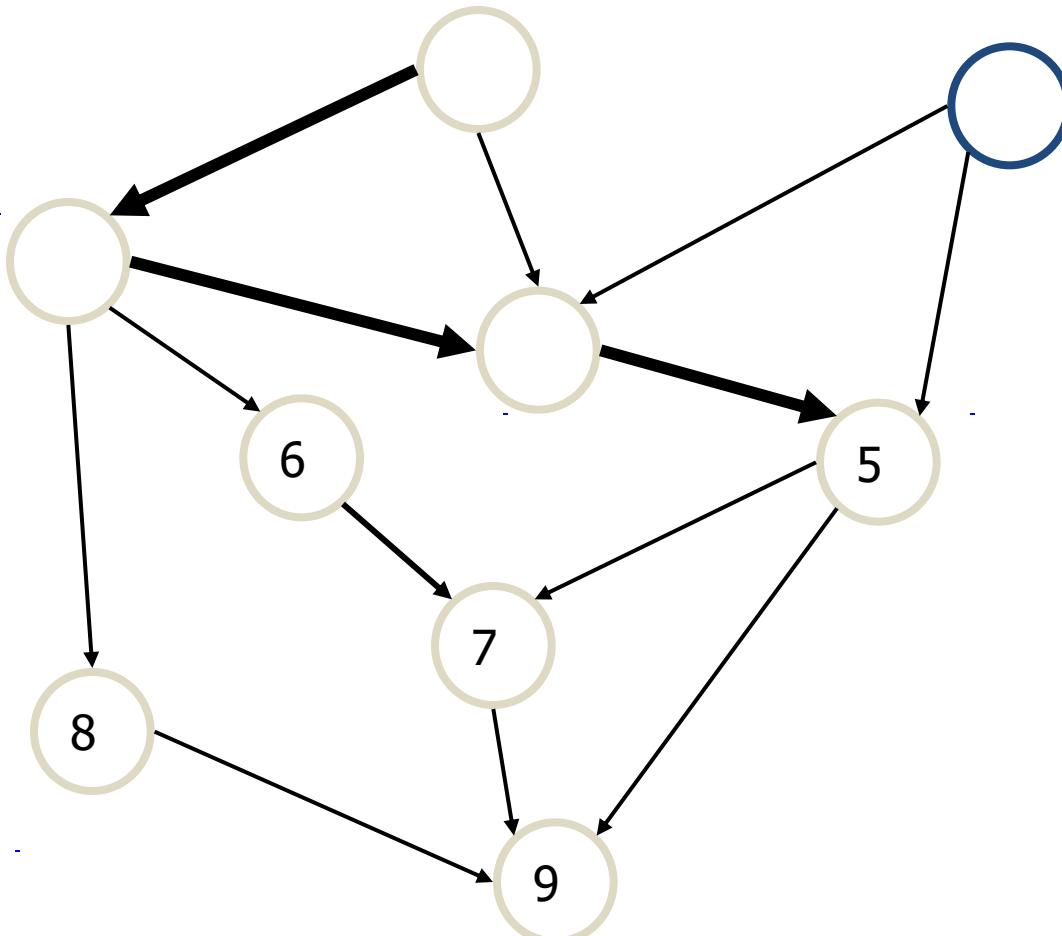
Topological Sorting Example



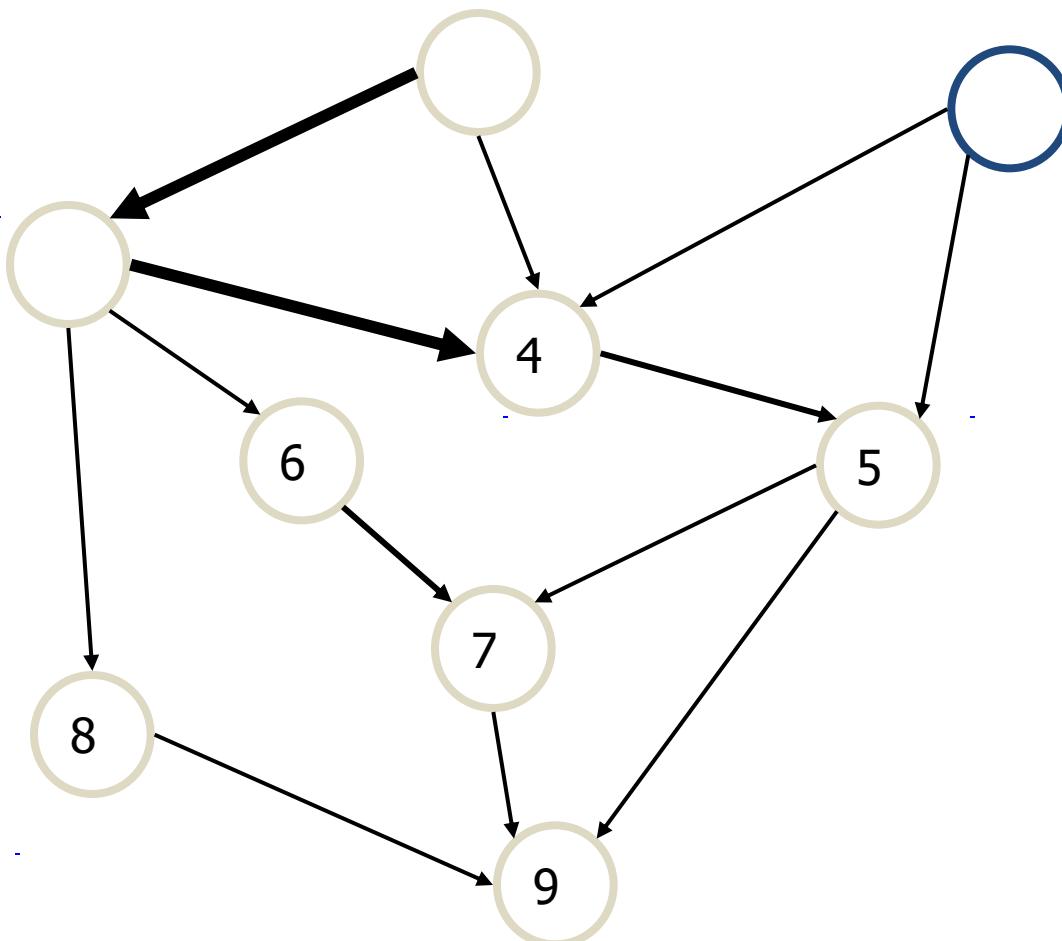
Topological Sorting Example



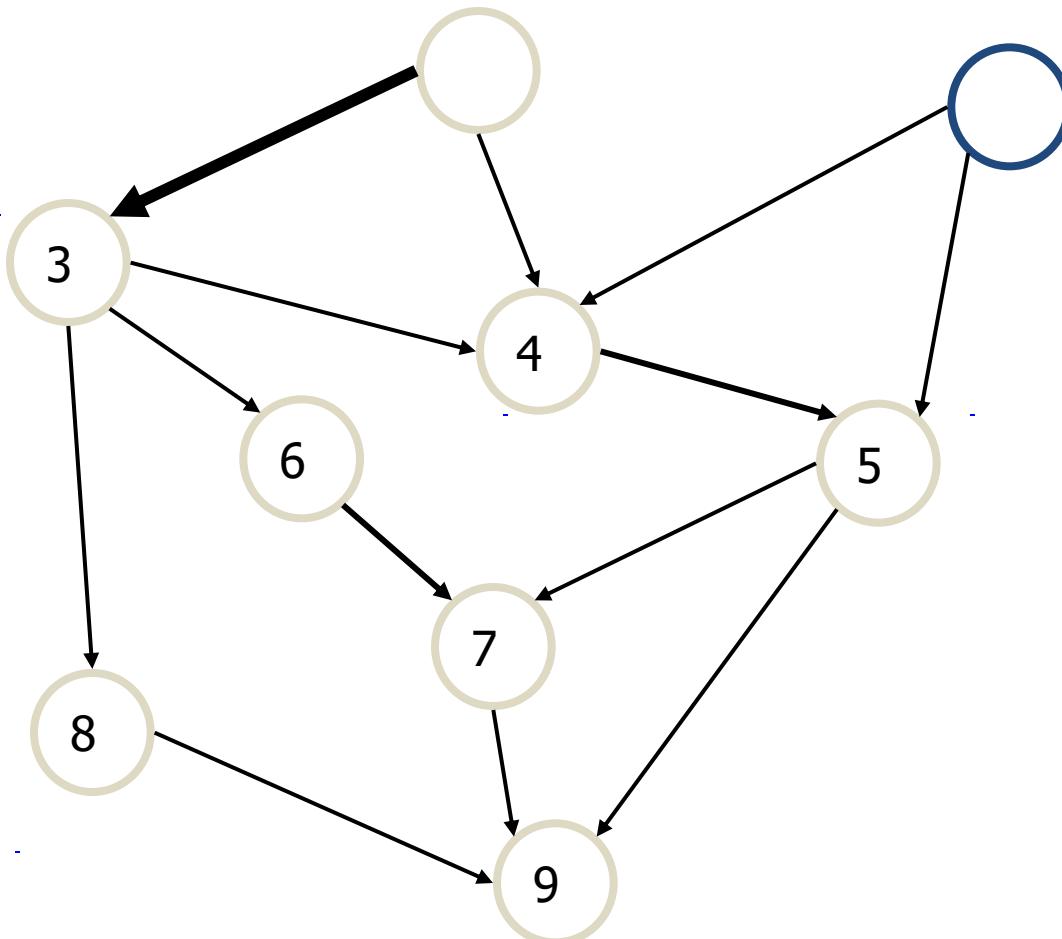
Topological Sorting Example



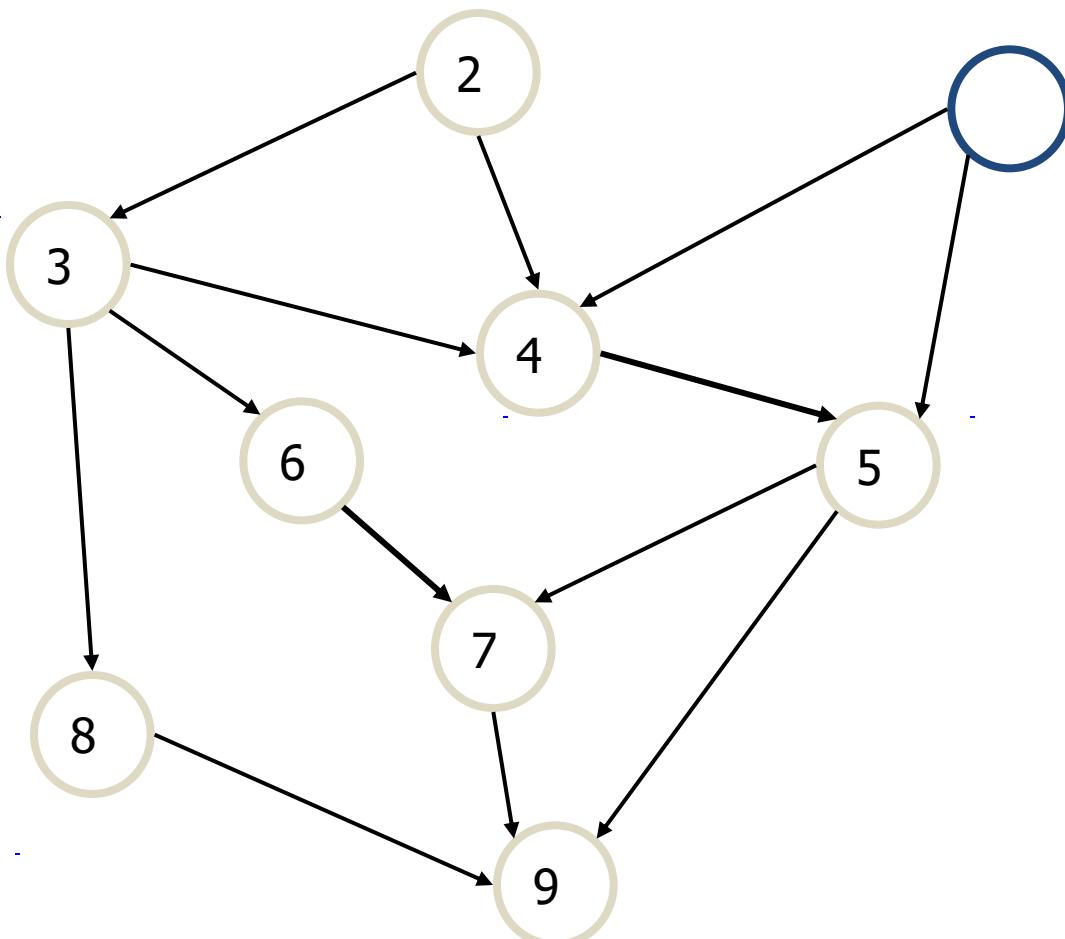
Topological Sorting Example



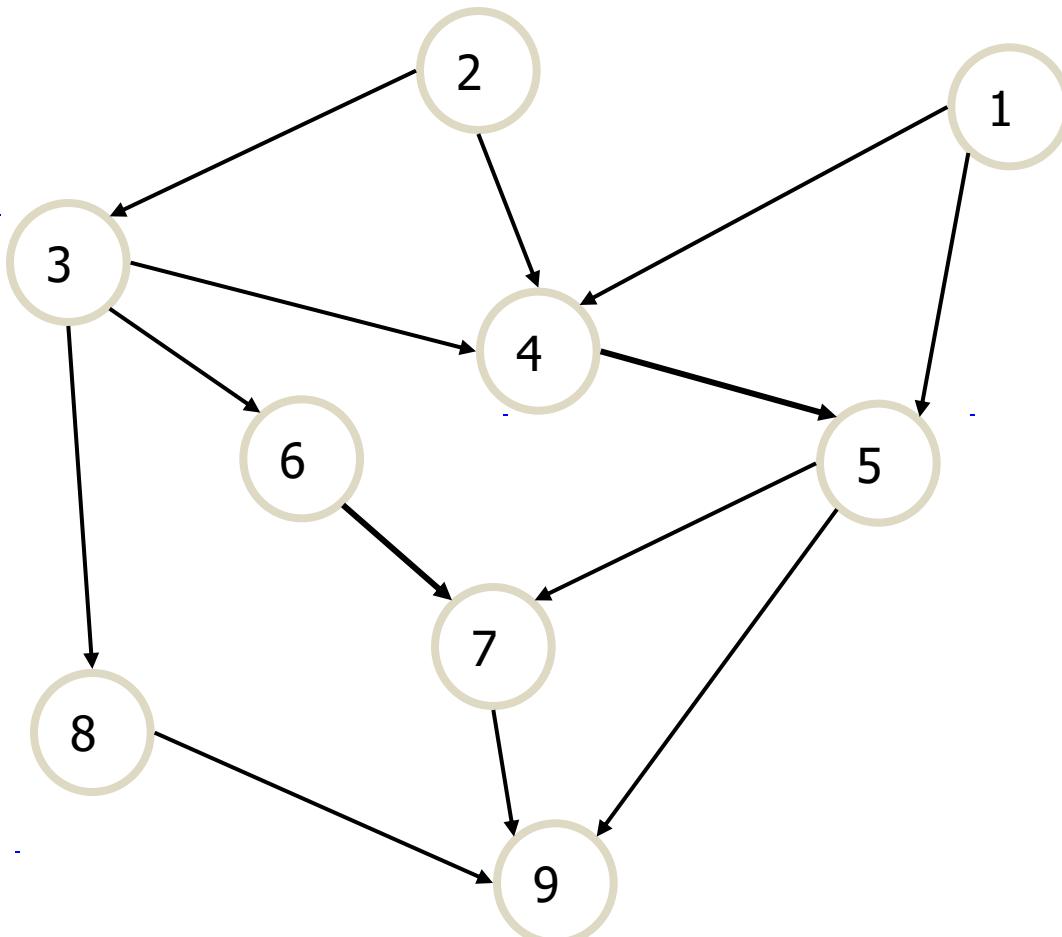
Topological Sorting Example



Topological Sorting Example



Topological Sorting Example



Minimum Spanning Trees

Spanning subgraph

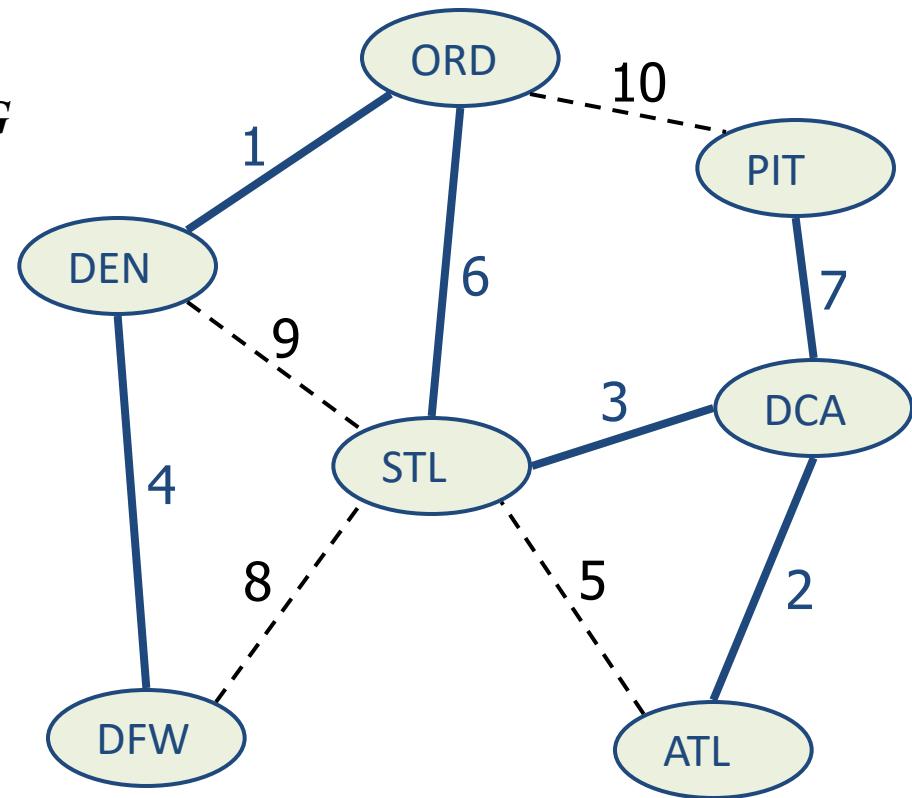
- Subgraph of a graph G containing all the vertices of G

Spanning tree

- Spanning subgraph that is itself a (free) tree

Minimum spanning tree (MST)

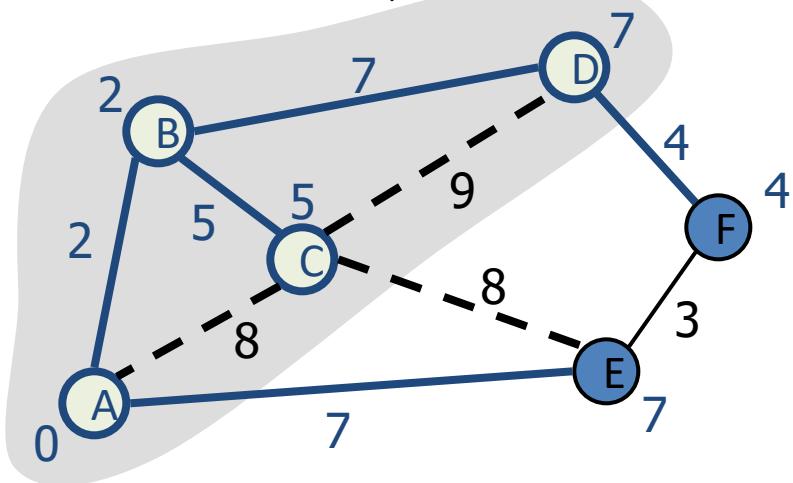
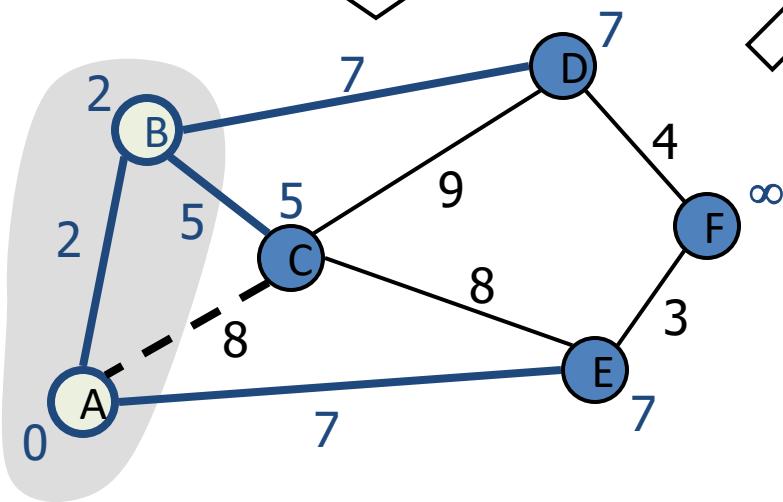
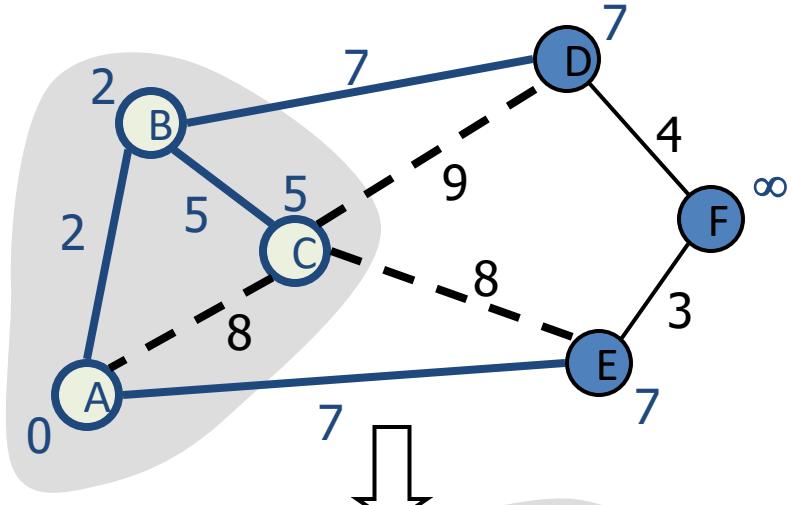
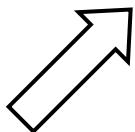
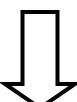
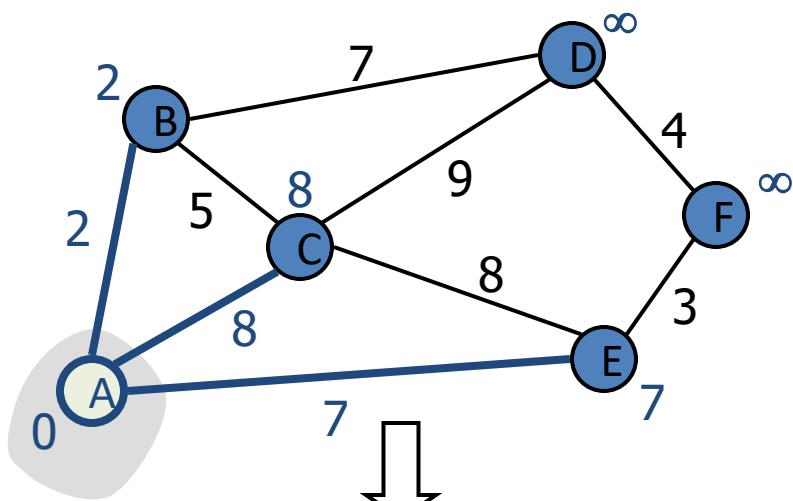
- Spanning tree of a weighted graph with minimum total edge weight
- Applications
 - Communications networks
 - Transportation networks



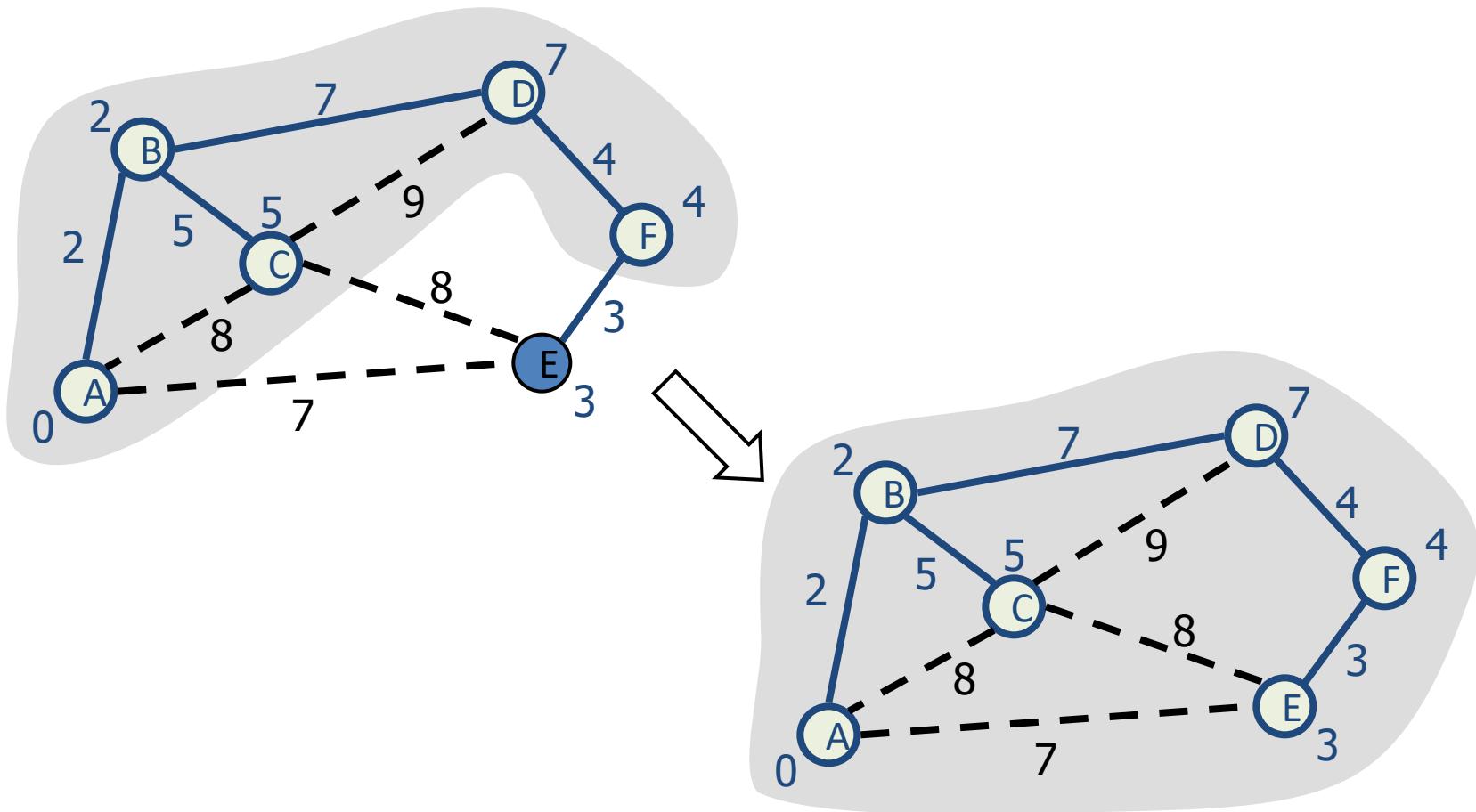
Prim's Algorithm

- Pick an arbitrary vertex s and grow the MST as a cloud of vertices, starting from s
- Store with each vertex v label $d(v)$ representing the smallest weight of an edge connecting v to a vertex in the cloud
- At each step:
 - We add to the cloud the vertex u outside the cloud with the smallest distance label
 - Make sure the result is a tree that is no cycle.
 - We update the labels of the vertices adjacent to u

Example



Example (contd.)



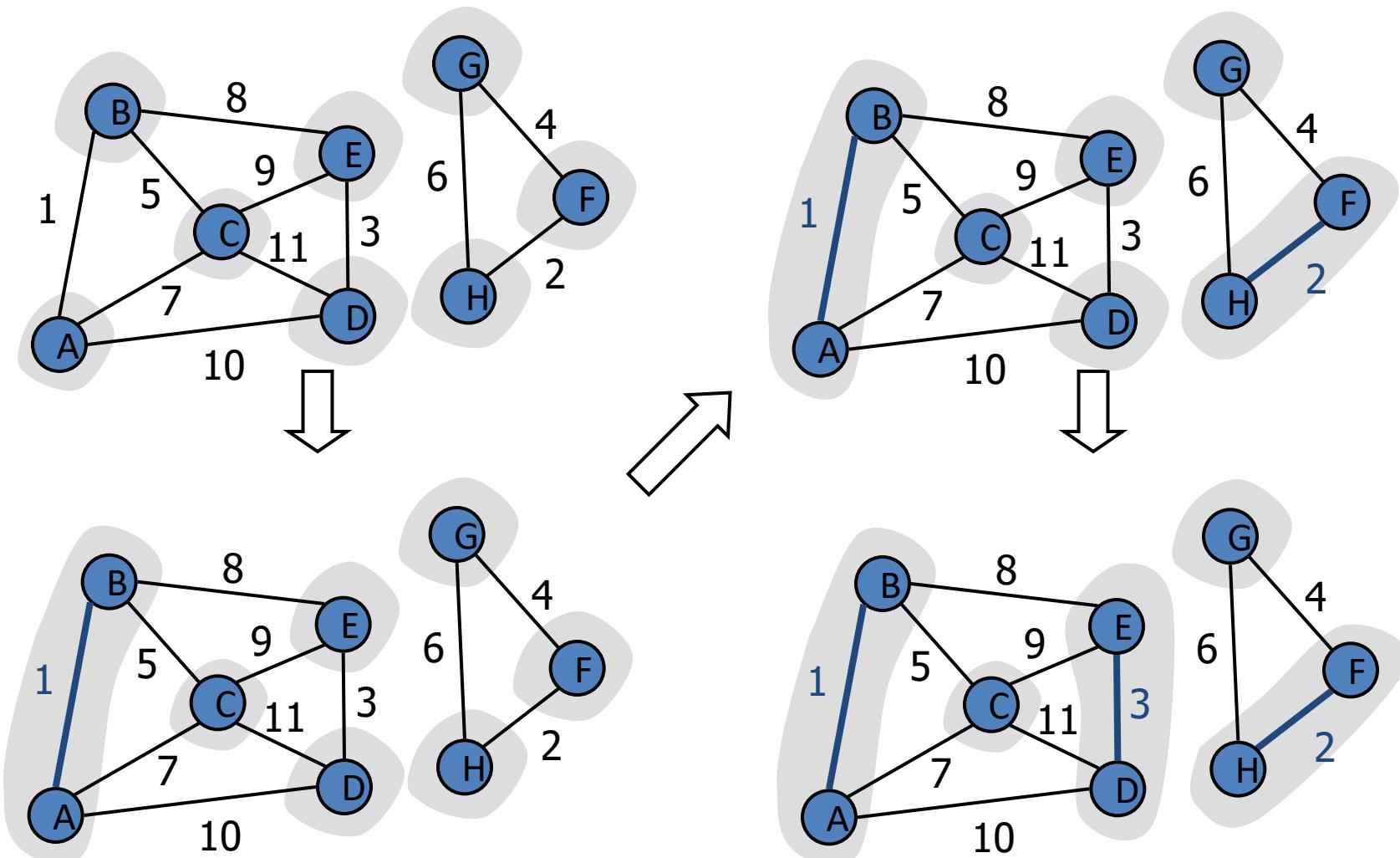
Analysis

- Graph operations
 - We cycle through the incident edges once for each vertex
- Label operations
 - We set/get the distance, parent and locator labels of vertex z $O(\deg(z))$ times
 - Setting/getting a label takes $O(1)$ time
- Priority queue operations
 - Each vertex is inserted once into and removed once from the priority queue, where each insertion or removal takes $O(\log n)$ time
 - The key of a vertex w in the priority queue is modified at most $\deg(w)$ times, where each key change takes $O(\log n)$ time
- Prim's algorithm runs in $O((n + m) \log n)$ time provided the graph is represented by the adjacency list structure. Recall that sum of all degrees equals $2m$. The running time is $O(m \log n)$ since the graph is connected

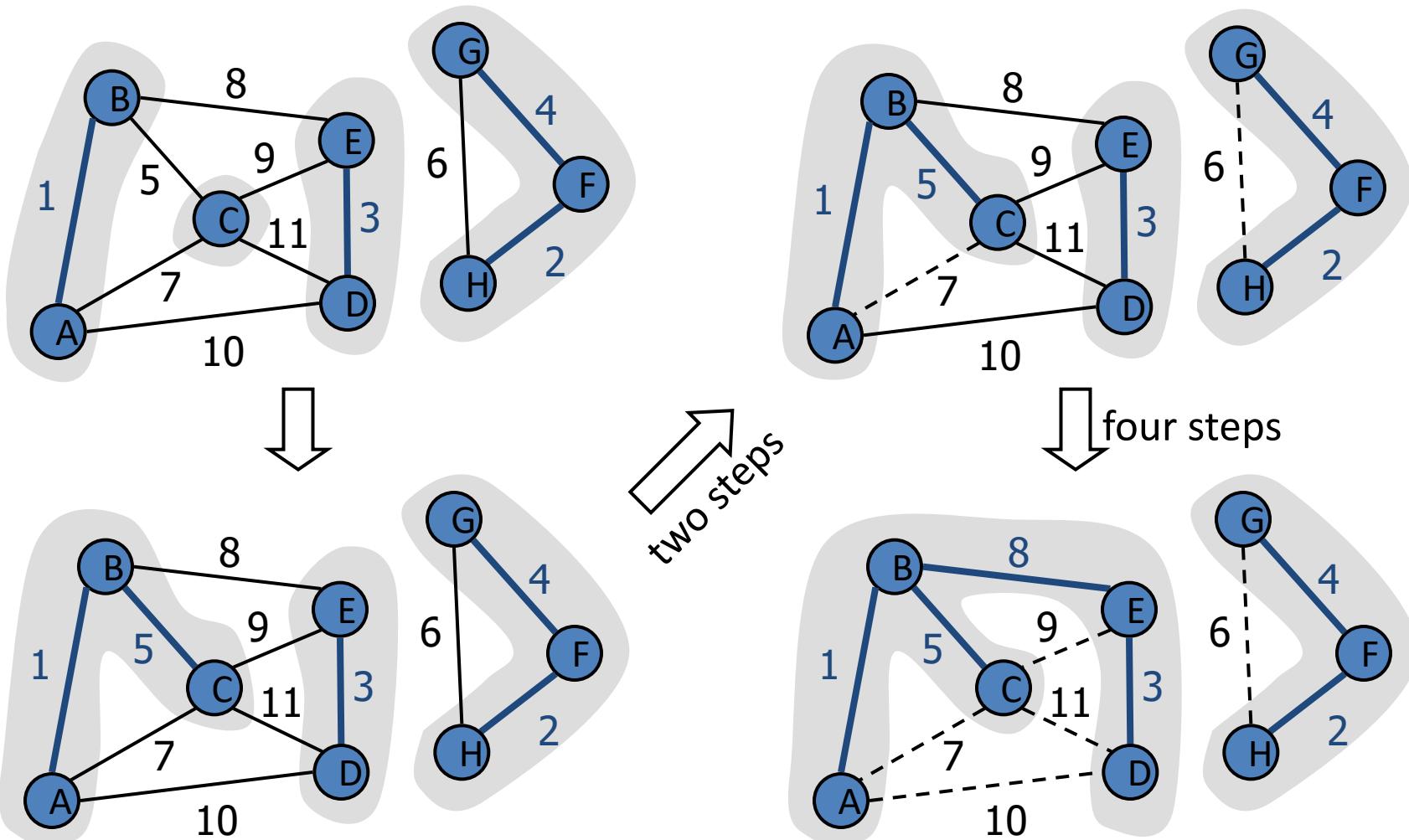
Kruskal's Algorithm

- Maintain a partition of the vertices into clusters
 - Initially, single-vertex clusters
 - Keep an MST for each cluster
 - Merge “closest” clusters and their MSTs
- A priority queue stores the edges outside clusters
 - Key: weight
 - Element: edge
- At the end of the algorithm
 - One cluster and one MST

Example



Example (contd.)

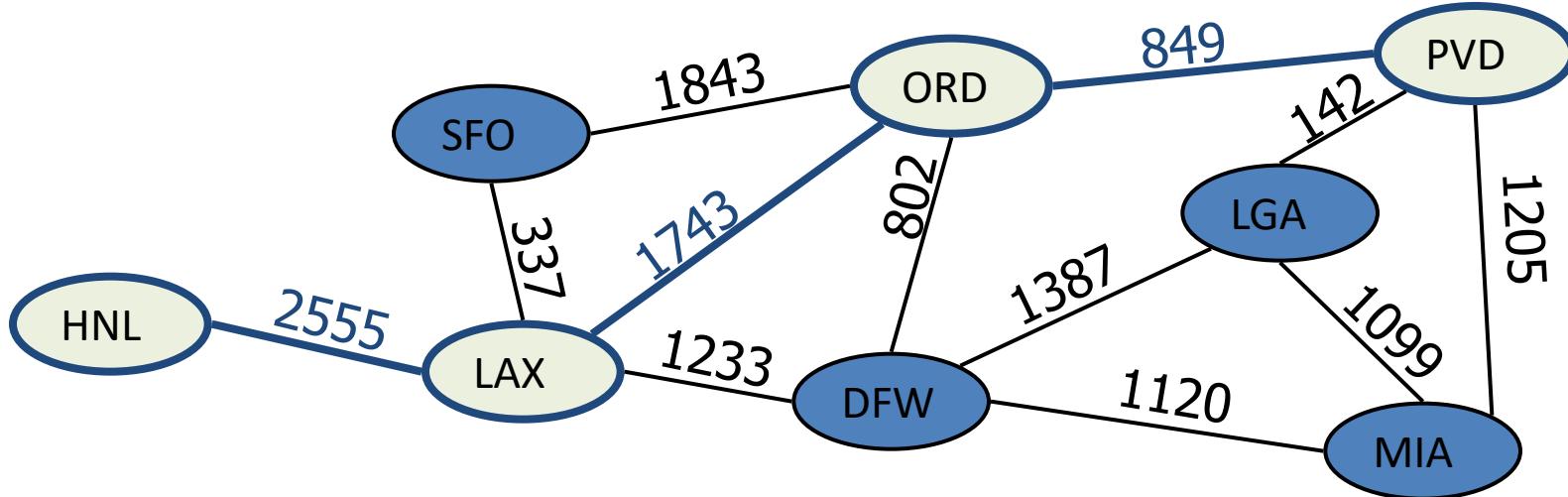


Data Structure for Kruskal's Algorithm

- ❑ The algorithm maintains a forest of trees
- ❑ A priority queue extracts the edges by increasing weight
- ❑ An edge is accepted if it connects distinct trees
- ❑ We need a data structure that maintains a partition, i.e., a collection of disjoint sets, with operations:
 - **makeSet(u)**: create a set consisting of u
 - **find(u)**: return the set storing u
 - **union(A, B)**: replace sets A and B with their union

Shortest Paths

- Given a weighted graph and two vertices u and v , we want to find a path of minimum total weight between u and v .
 - Length of a path is the sum of the weights of its edges.
- Some applications
 - Internet packet routing
 - Flight reservations
 - Driving directions



Shortest Path Properties

Property 1:

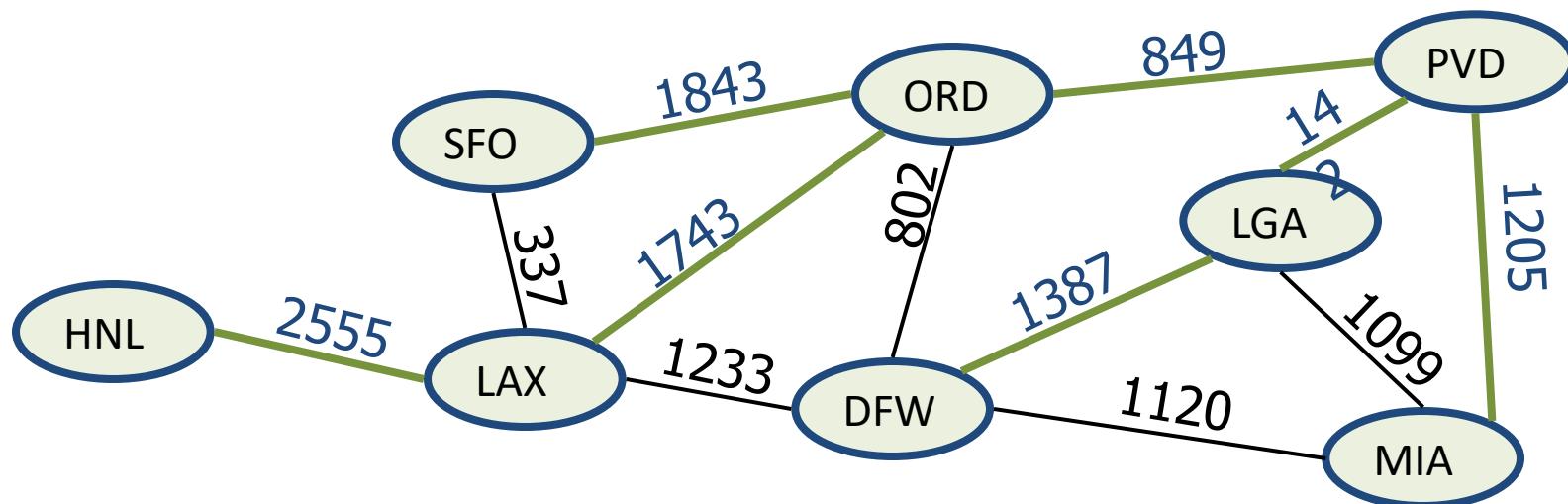
A subpath of a shortest path is itself a shortest path

Property 2:

There is a tree of shortest paths from a start vertex to all the other vertices

Example:

Tree of shortest paths from Providence



Dijkstra's Algorithm

- It computes the shortest distance of all the vertices from a given source vertex s
- Assumptions:
 - the graph is connected
 - the edges are undirected
 - the edge weights are nonnegative
- We grow a **cloud** of vertices, beginning with s and eventually covering all the vertices
- We store with each vertex v a **label $d(v)$** representing the distance of v from s in the subgraph consisting of the cloud and its adjacent vertices
- At each step
 - We add to the cloud the vertex u outside the cloud with the smallest distance label, $d(u)$
 - We update the labels of the vertices adjacent to u

Dijkstra's Algorithm

1. Store s in a priority queue with distance = 0
2. While there are vertices in the queue

 DeleteMin a vertex v from the queue

 For all adjacent vertices w :

 Compute new distance

 Store in / update Distance table

 Insert/update in priority queue

Complexity

$$O(E \log V + V \log V) = O((E + V) \log(V))$$

Each vertex is stored only once in the queue – $O(V)$

DeleteMin operation is : $O(V \log V)$

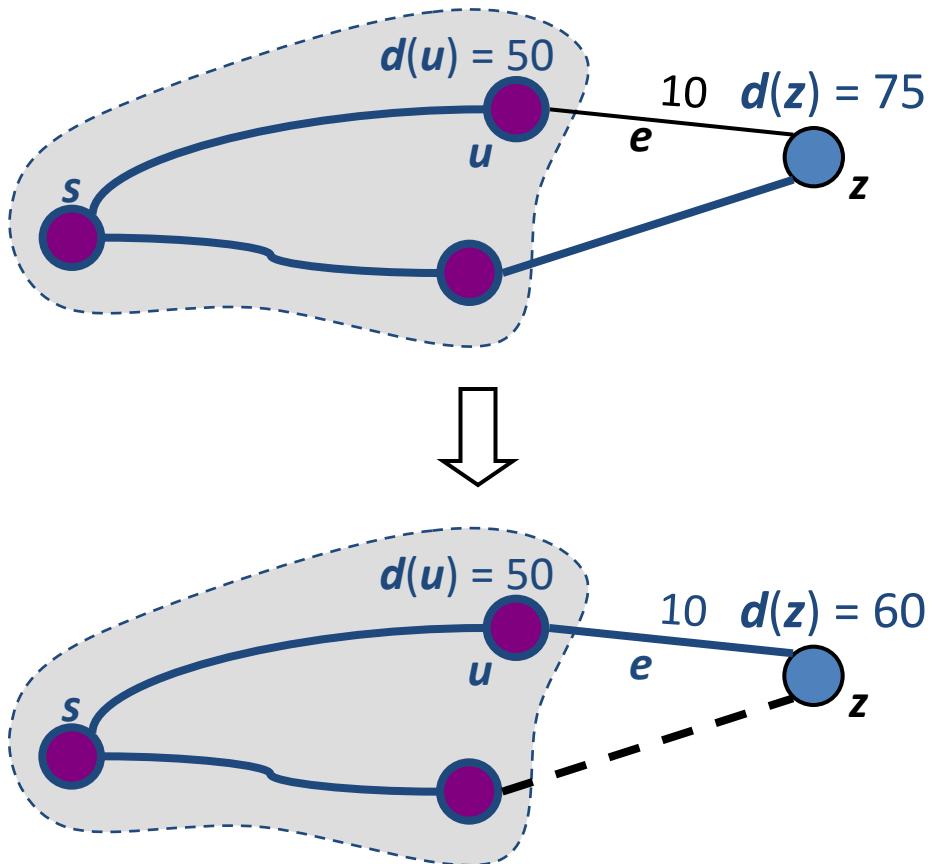
Updating the priority queue –

search and insert: $O(\log V)$

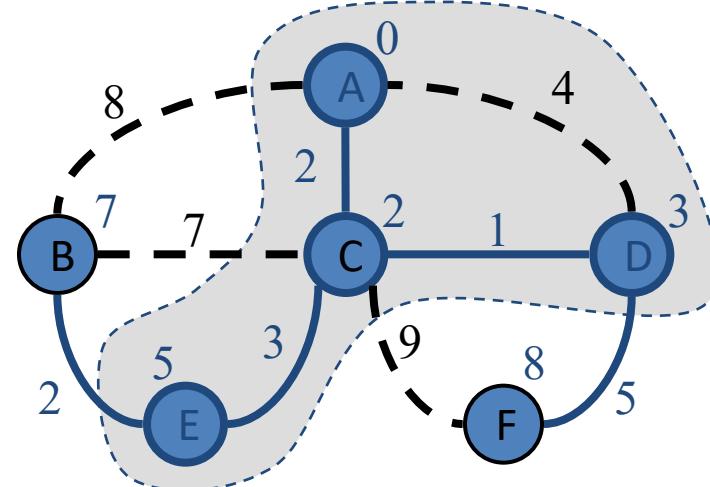
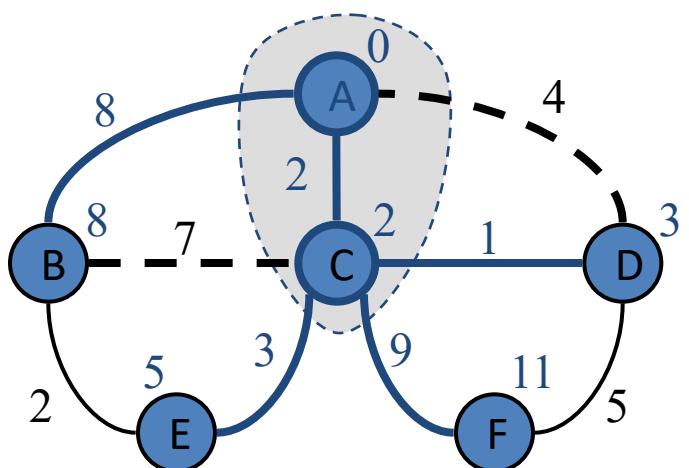
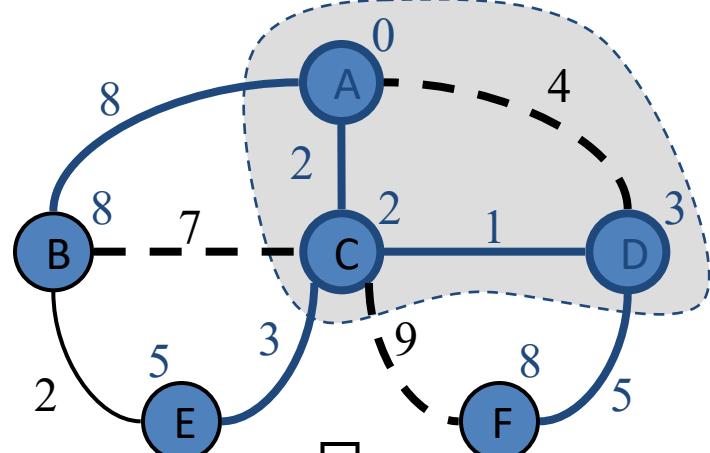
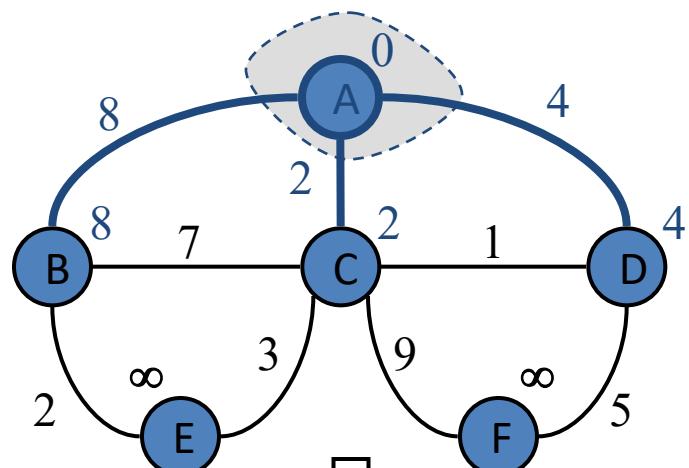
performed at most for each edge: $O(E \log V)$

Edge Relaxation

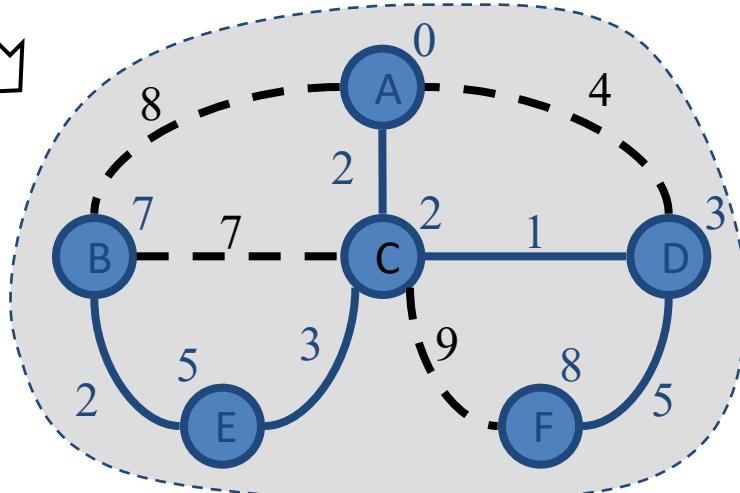
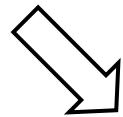
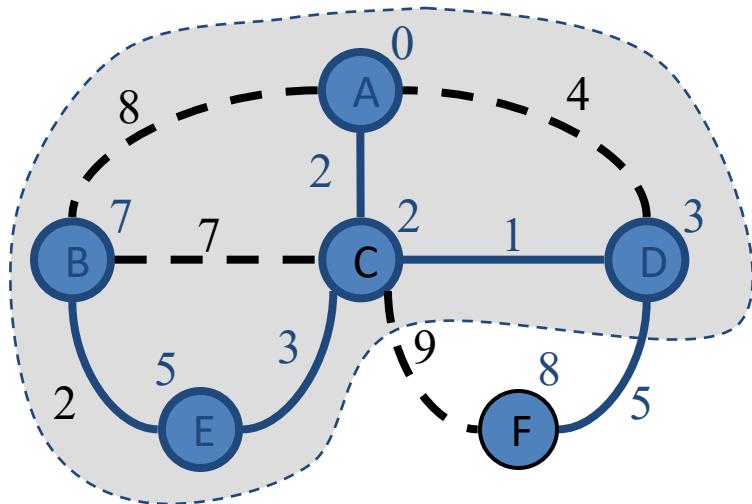
- Consider an edge $e = (u, z)$ such that
 - u is the vertex most recently added to the cloud
 - z is not in the cloud
- The relaxation of edge e updates distance $d(z)$ as follows:
$$d(z) \leftarrow \min\{d(z), d(u) + \text{weight}(e)\}$$



Example



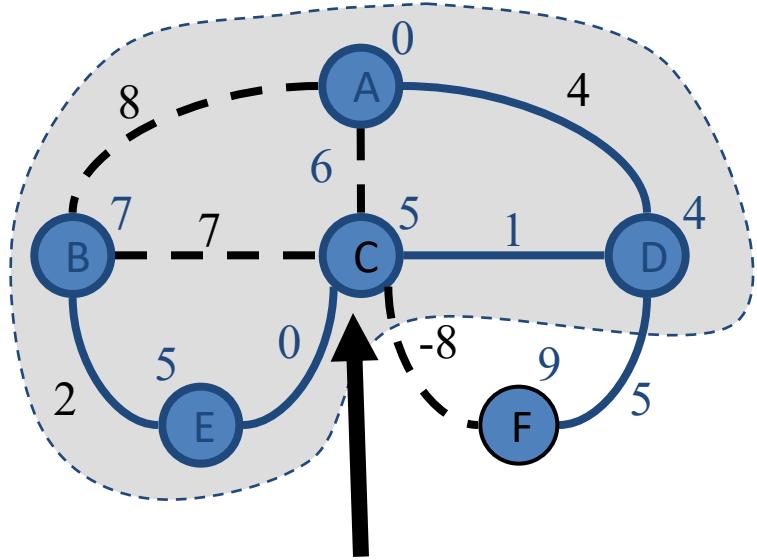
Example (cont.)



Why It Doesn't Work for Negative-Weight Edges

- ◆ Dijkstra's algorithm is based on the greedy method. It adds vertices by increasing distance.

- If a node with a negative incident edge were to be added late to the cloud, it could mess up distances for vertices already in the cloud.

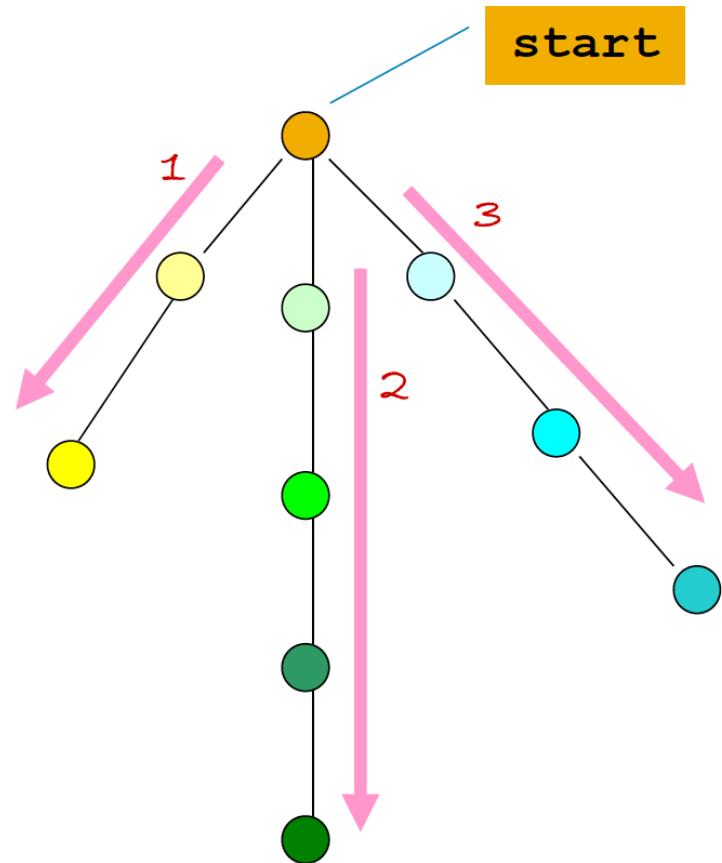


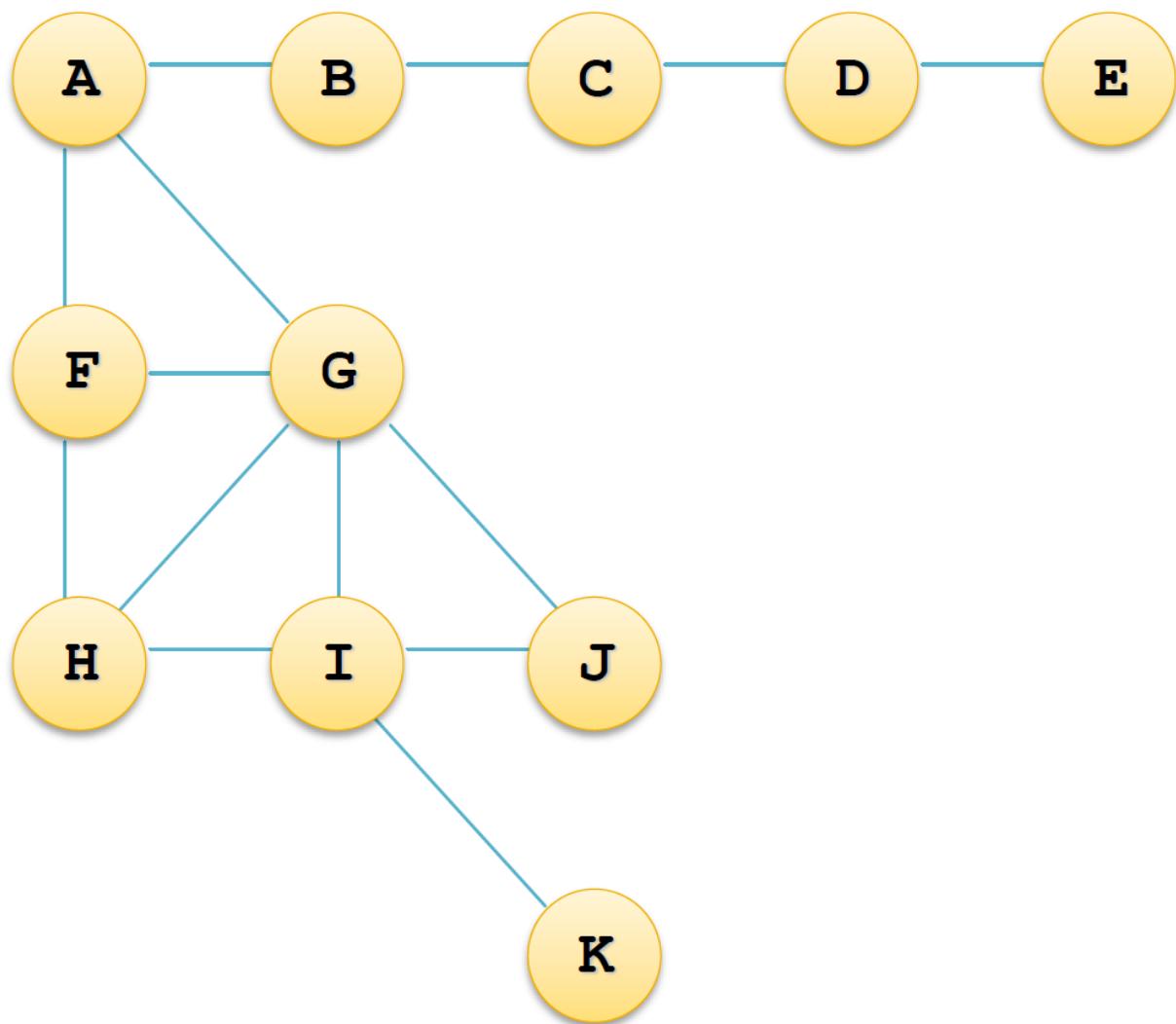
C's true distance is 1, but it is already in the cloud with $d(C)=5$!

Depth-First Search (DFS)

- A general technique for traversing a graph G that
 - Visits all the vertices and edges of G
 - Determines whether G is connected
 - Computes the connected components of G
 - Computes a spanning forest of G
- For a graph with n nodes and m edges it takes $O(n + m)$
- DFS can be extended to solve other graph problems
 - Find a path between two given vertices
 - Find a cycle in the graph

- Visit a vertex, v , move from v as deeply as possible
- Use a stack to store nodes
 - Stacks are LIFO
- DFS:
 - visit and push start
 - while (s not empty)
 - peek at node, nd , at top of s
 - if nd has an unvisited neighbour visit it and push it onto s
 - else pop nd from s





stack

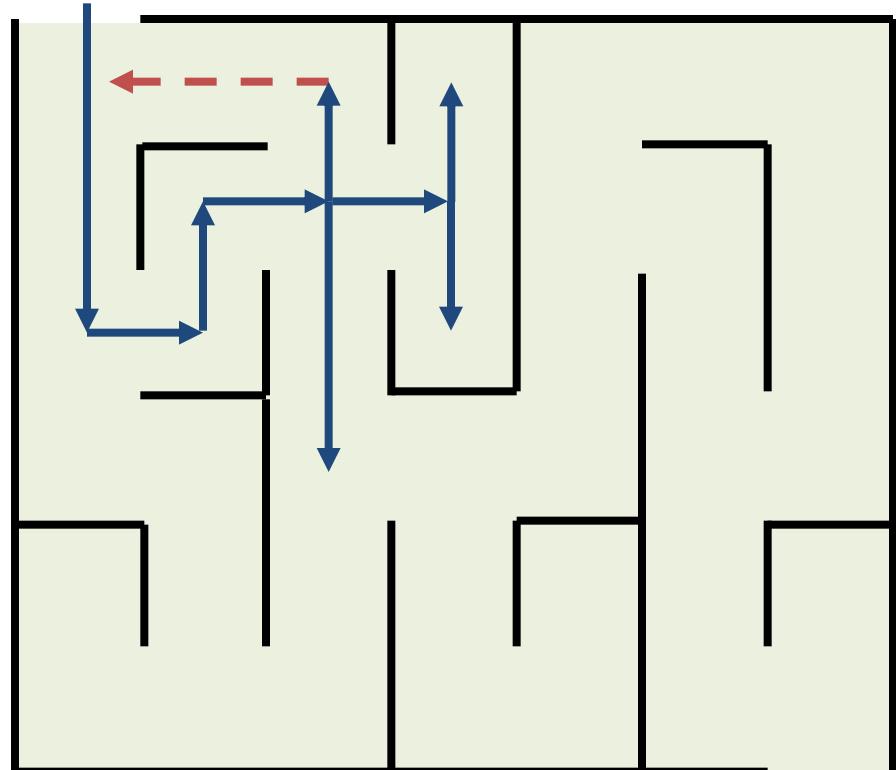
J	K
E	I
D	H
C	G
B	F
A	

visited

A
B
C
D
E
F
G
H
I
J
K

DFS and Maze Traversal

- The DFS algorithm is similar to a classic strategy for exploring a maze
 - We mark each intersection, corner and dead end (vertex) visited
 - We mark each corridor (edge) traversed
 - We keep track of the path back to the entrance (start vertex) by means of a rope (recursion stack)



Path Finding



- We can specialize the DFS algorithm to find a path between two given vertices u and z using the template method pattern
- We call $\text{DFS}(G, u)$ with u as the start vertex
- We use a stack S to keep track of the path between the start vertex and the current vertex
- As soon as destination vertex z is encountered, we return the path as the contents of the stack

```
Algorithm pathDFS(G, v, z)
  setLabel(v, VISITED)
  S.push(v)
  if v = z
    return S.elements()
  for all e ∈ G.incidentEdges(v)
    if getLabel(e) = UNEXPLORED
      w ← opposite(v,e)
      if getLabel(w) = UNEXPLORED
        setLabel(e, DISCOVERY)
        S.push(e)
        pathDFS(G, w, z)
        S.pop(e)
      else
        setLabel(e, BACK)
    S.pop(v)
```



Cycle Finding

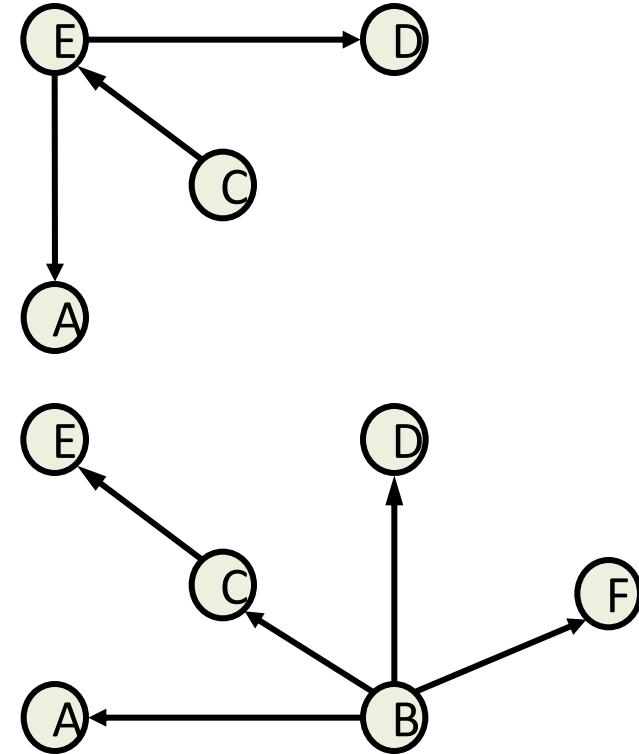
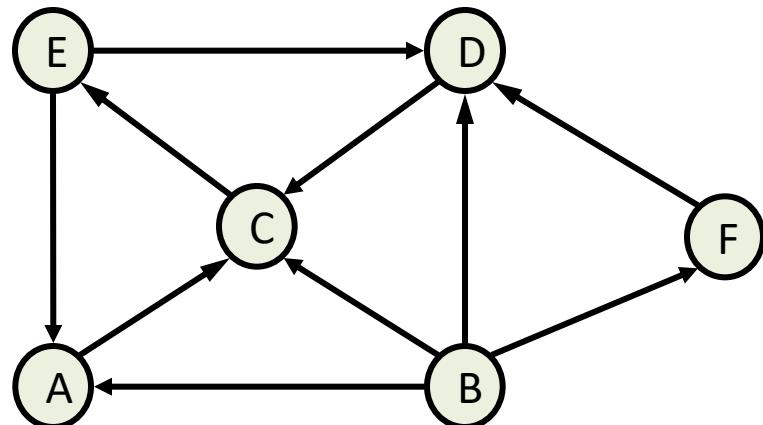
- We can use DFS to find a simple cycle
- We use a stack S to keep track of the path between the start vertex and the current vertex
- As soon as a back edge (v, w) is encountered, we return the cycle as the portion of the stack from the top to vertex w

```
Algorithm cycleDFS( $G, v, z$ )
  setLabel( $v, VISITED$ )
   $S.push(v)$ 
  for all  $e \in G.incidentEdges(v)$ 
    if getLabel( $e$ ) = UNEXPLORED
       $w \leftarrow \text{opposite}(v, e)$ 
       $S.push(e)$ 
      if getLabel( $w$ ) = UNEXPLORED
        setLabel( $e, DISCOVERY$ )
        pathDFS( $G, w, z$ )
         $S.pop(e)$ 
      else
         $T \leftarrow \text{new empty stack}$ 
        repeat
           $o \leftarrow S.pop()$ 
           $T.push(o)$ 
        until  $o = w$ 
      return  $T.elements()$ 
   $S.pop(v)$ 
```

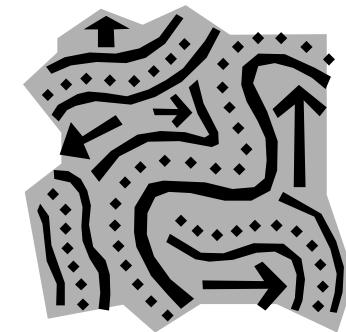
Reachability



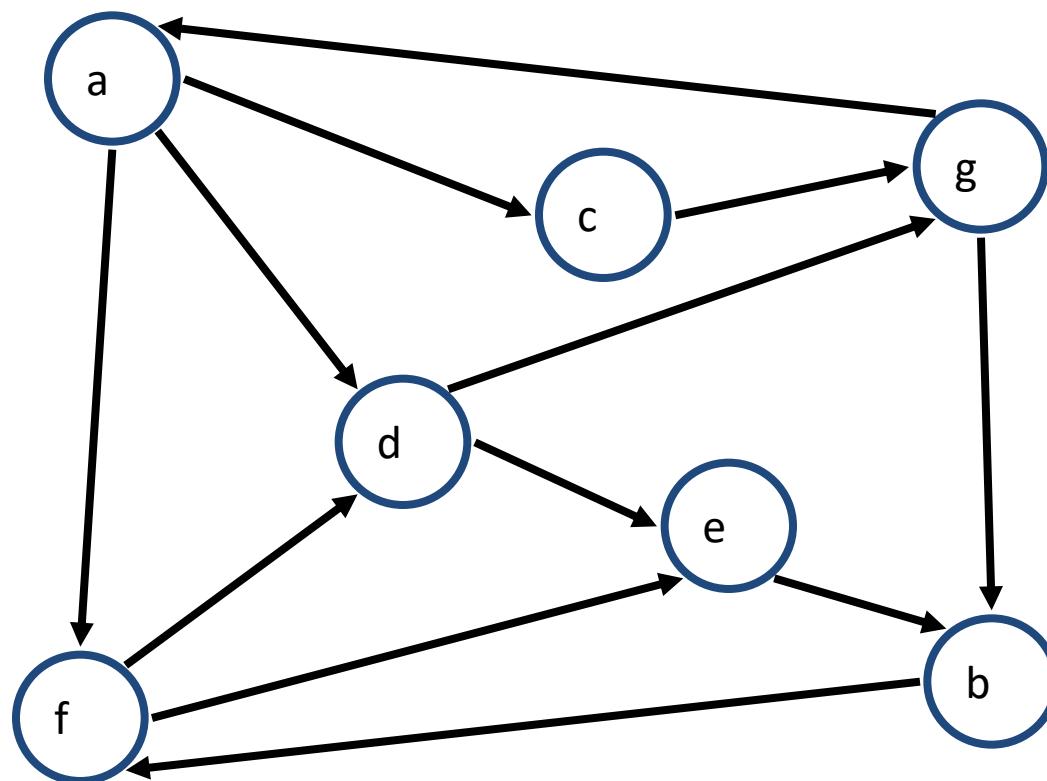
- DFS **tree** rooted at v : vertices reachable from v via directed paths



Strong Connectivity

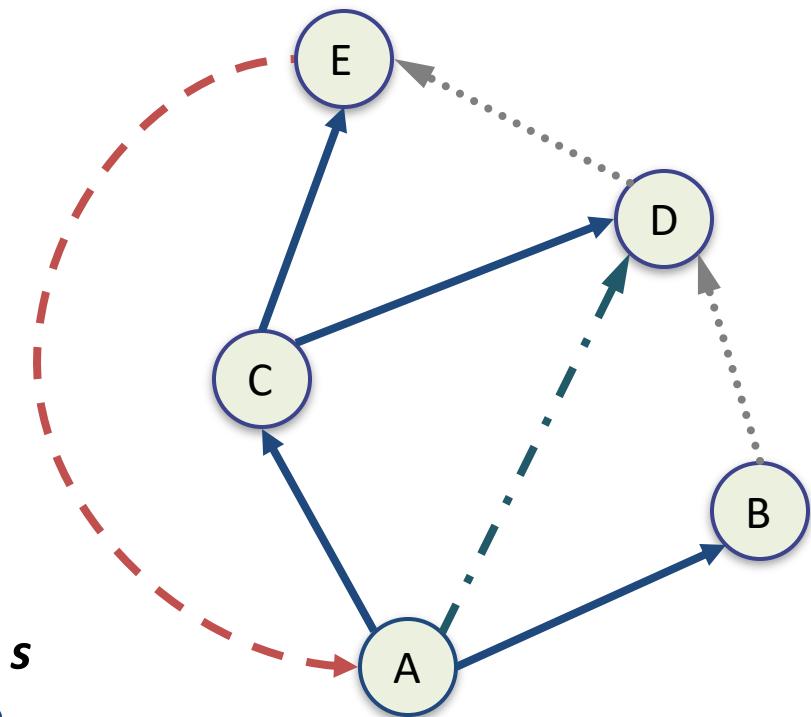


- Each vertex can reach all other vertices



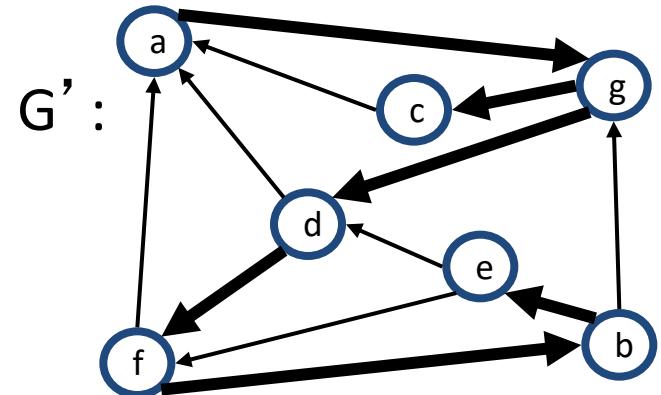
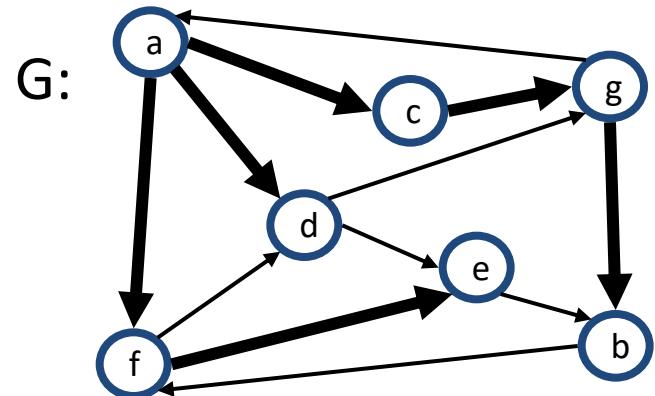
Directed DFS

- ❑ We can specialize the traversal algorithms (DFS and BFS) to digraphs by traversing edges only along their direction
- ❑ In the directed DFS algorithm, we have four types of edges
 - discovery edges
 - back edges
 - forward edges
 - cross edges
- ❑ A directed DFS starting at a vertex s determines the vertices reachable from s



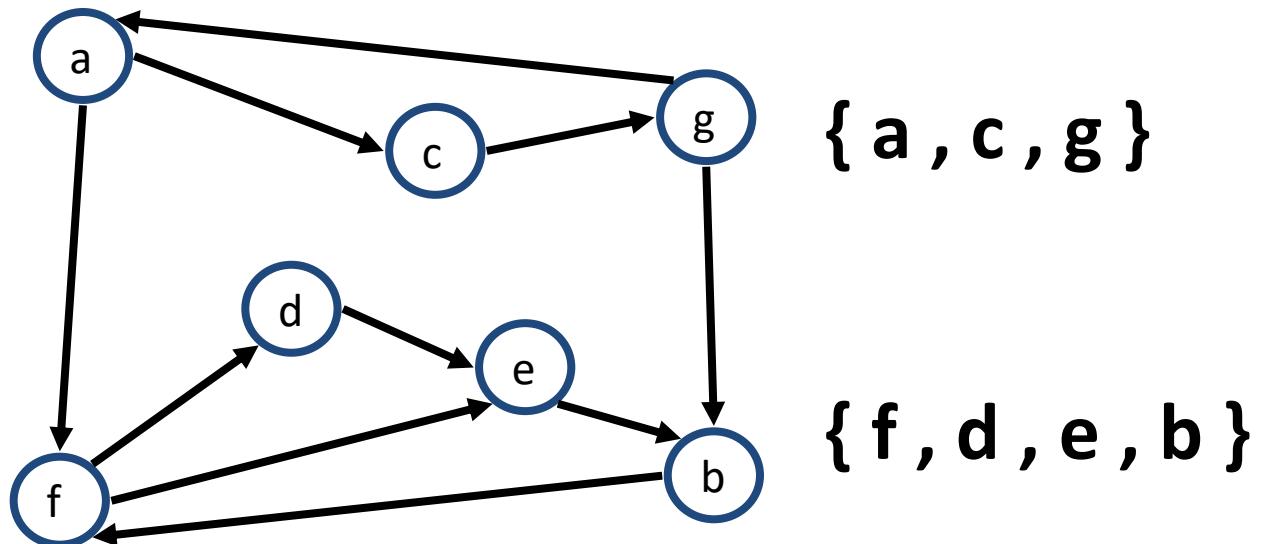
Strong Connectivity Algorithm

- Pick a vertex v in G
- Perform a DFS from v in G
 - If there's a w not visited, print “no”
- Let G' be G with edges reversed
- Perform a DFS from v in G'
 - If there's a w not visited, print “no”
 - Else, print “yes”
- Running time: $O(n+m)$



Strongly Connected Components

- Maximal subgraphs such that each vertex can reach all other vertices in the subgraph
- Can also be done in $O(n+m)$ time using DFS, but is more complicated (similar to biconnectivity).



All Connected Components

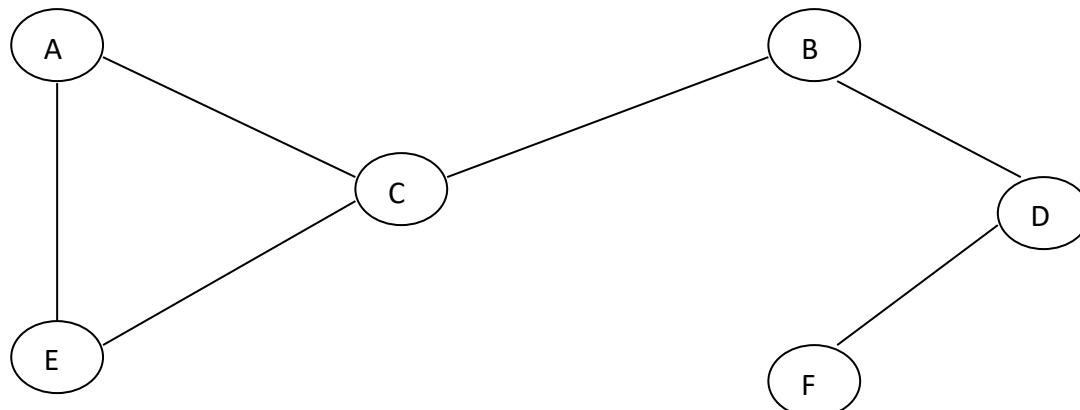
- Loop over all vertices, doing a DFS from each unvisted one.

```
1  /** Performs DFS for the entire graph and returns the DFS forest as a map. */
2  public static <V,E> Map<Vertex<V>,Edge<E>> DFSComplete(Graph<V,E> g) {
3      Set<Vertex<V>> known = new HashSet<>();
4      Map<Vertex<V>,Edge<E>> forest = new ProbeHashMap<>();
5      for (Vertex<V> u : g.vertices( ))
6          if (!known.contains(u))
7              DFS(g, u, known, forest);           // (re)start the DFS process at u
8      return forest;
9  }
```

Bridges and CutPoints

An edge in a graph is called a bridge, if its removal disconnects the graph. A node is a Cut Point if its removal disconnects a graph.

How can we detect them?

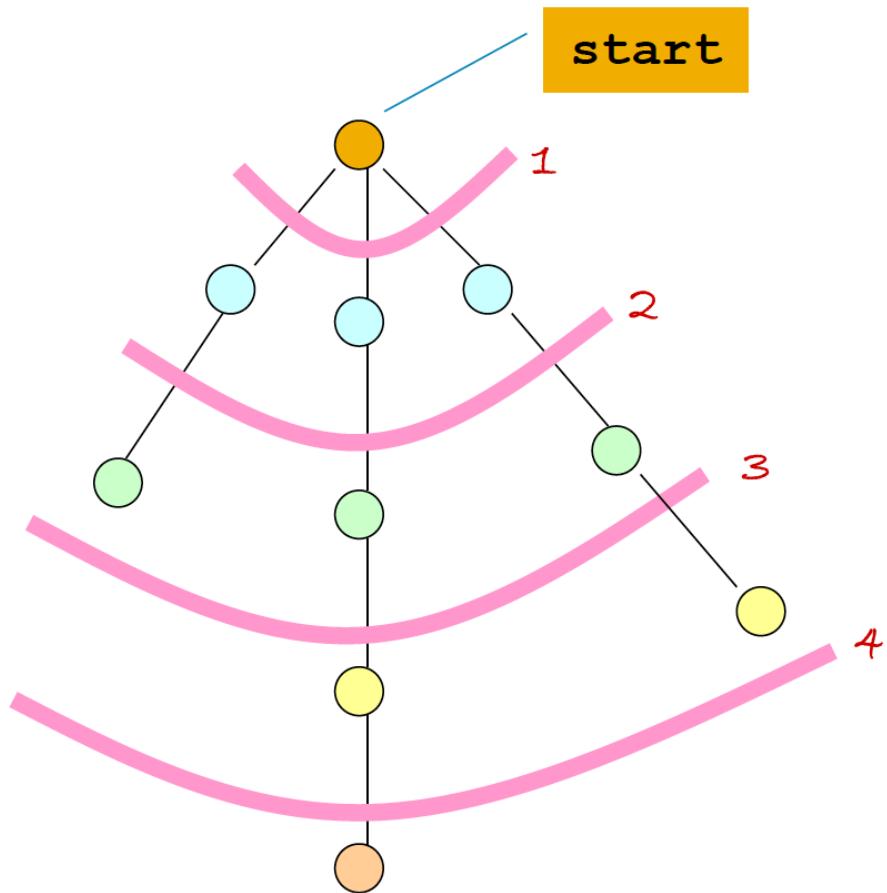


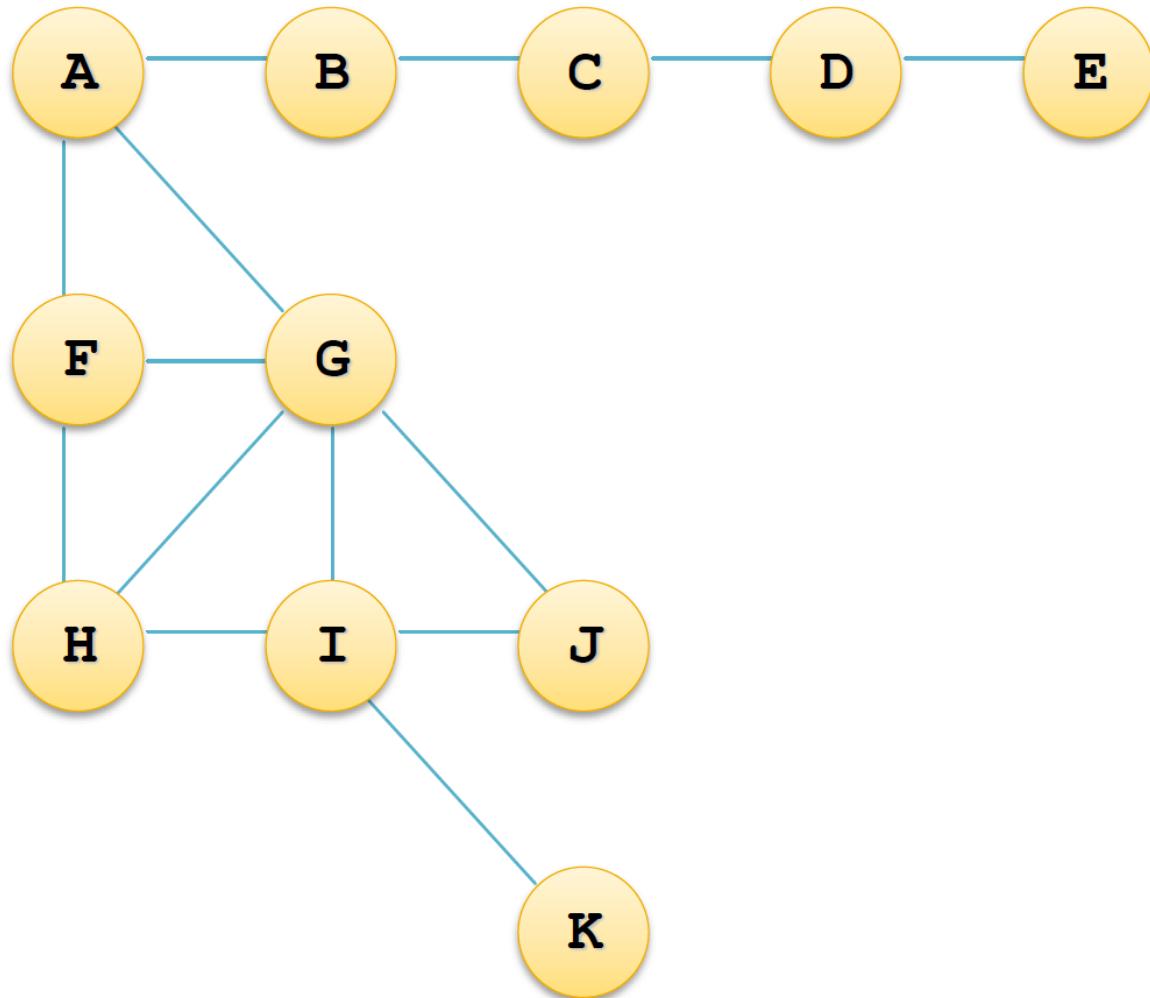
C is a cut point, and (C,B) is a bridge.

Breadth-First Search

- A general technique for traversing a graph G that
 - Visits all the vertices and edges of G
 - Determines whether G is connected
 - Computes the connected components of G
 - Computes a spanning forest of G
- For a graph with n nodes and m edges it takes $O(n + m)$
- BFS can be extended to solve other graph problems
 - Find and report a path with the minimum number of edges between two given vertices
 - Find a simple cycle, if there is one

- After visiting a vertex, v , visit every vertex adjacent to v before moving on
- Use a queue to store nodes
 - Queues are FIFO
- BFS:
 - visit and insert start
 - while (q not empty)
 - remove node from q and make it *current*
 - visit and insert the unvisited nodes adjacent to *current*





queue

A
B
F
G
C
H
I
J
D
K
E

visited

A
B
F
G
C
H
I
J
D
K
E

BFS Algorithm

Algorithm $BFS(G, s)$

```
 $L_0 \leftarrow$  new empty sequence  
 $L_0.addLast(s)$   
setLabel(s, VISITED)  
 $i \leftarrow 0$   
while  $L_i.isNotEmpty()$   
     $L_{i+1} \leftarrow$  new empty sequence  
    for all  $v \in L_i.elements()$   
        for all  $e \in G.incidentEdges(v)$   
            if  $getLabel(e) = UNEXPLORED$   
                 $w \leftarrow opposite(v,e)$   
                if  $getLabel(w) = UNEXPLORED$   
                    setLabel(e, DISCOVERY)  
                    setLabel(w, VISITED)  
                     $L_{i+1}.addLast(w)$   
            else  
                setLabel(e, CROSS)  
     $i \leftarrow i + 1$ 
```

Applications

- Using the template method pattern, we can specialize the BFS traversal of a graph G to solve the following problems in $O(n + m)$ time
 - Compute the connected components of G
 - Compute a spanning forest of G
 - Find a simple cycle in G , or report that G is a forest
 - Given two vertices of G , find a path in G between them with the minimum number of edges, or report that no such path exists

Shortest Path in Unweighted Graphs

Data Structures needed:

Distance Table

Queue

Implementation: Breadth-First search using a queue

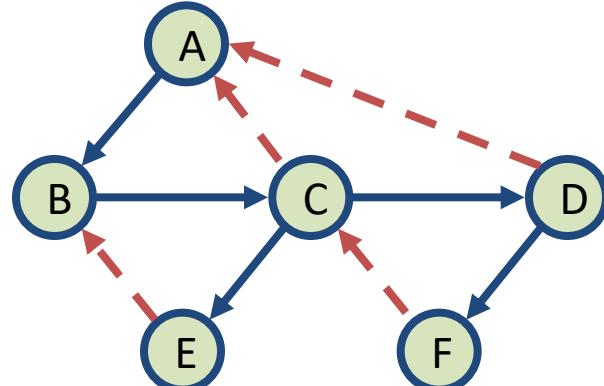
Complexity:

Matrix representation: $O(|V|^2)$

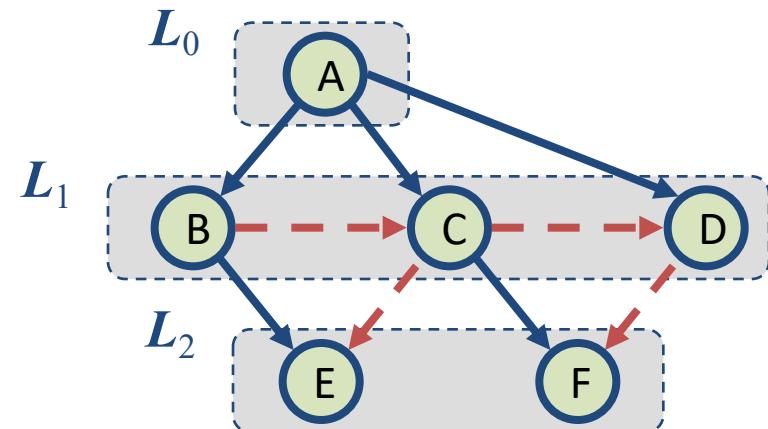
Adjacency lists - $O(|E| + |V|)$

DFS vs. BFS

Applications	DFS	BFS
Spanning forest, connected components, paths, cycles	✓	✓
Shortest paths		✓
Biconnected components	✓	



DFS



BFS