

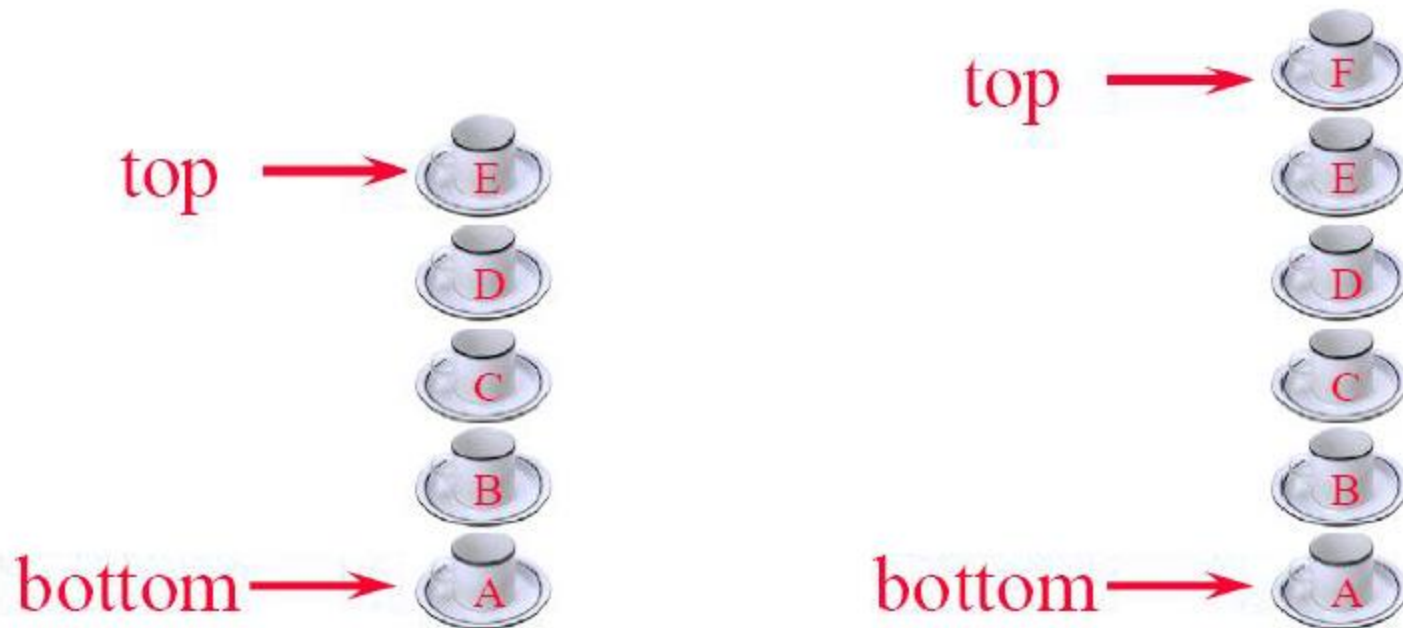
# DS221: Introduction to Scalable Systems

## Topic: Algorithms and Data Structures



# L3: More Basic Data Structures: *Stack, Queue, Trees*

# Stacks



- Add a cup to the stack.
- Remove a cup from new stack.
- A stack is a **LIFO** list: *Last in, First out*



# Stacks

- Container of objects that are inserted and removed according to the LIFO principle
- Objects can be inserted at any time, but only the last object can be removed.
  - Inserting : “pushing”
  - Removing : “Popping”

# Stacks - ADT

- **New()** creates a new stack
- **Push(item)** inserts the *item* onto top of stack
- item **Pop()** removes and returns the top *item* of stack
- item **Top()** returns (but retains) the top *item* of stack (sometimes called **Peek()**)
- int **Size()** returns number of objects in stack
- Invariants
  - $S.\text{Pop}(S.\text{Push}(v))$  has the same stack state as  $S$
  - $S.\text{Top}(S.\text{Push}(v))$  returns the value  $v$

# Parenthesis Matching

- Problem: Match the left and right parentheses in a character string
- $(a*(b+c)+d)$ 
  - Left parentheses: positions 0 and 3
  - Right parentheses: positions 7 and 10
  - Left at position 0 matches with right at position 10
- $(a+b)) * ((c+d)$ 
  - (0,4) match
  - (8,12) match
  - Right parenthesis at 5 has no matching left parenthesis
  - Left parenthesis at 7 has no matching right parenthesis

# Parenthesis Matching

$(( (a+b)*c+d-e)/(f+g)-(h+j)*(k-1))/(m-n)$

- Output pairs (u,v) such that the left parenthesis at position u is matched with the right parenthesis at v.
- (2,6) (1,13) (15,19) (21,25) (27,31) (0,32) (34,38)

- **How do we implement this using a stack?**

1. Scan expression from left to right
2. When a left parenthesis is encountered, add its position to the stack
3. When a right parenthesis is encountered, remove matching position from the stack



# Example

■  $(a * (b + c) + d)$

0	1	2	3	4	5	6	7	8	9	10
(	a	*	(	b	+	c	)	+	d	)

0
---

3
0

0
---

--

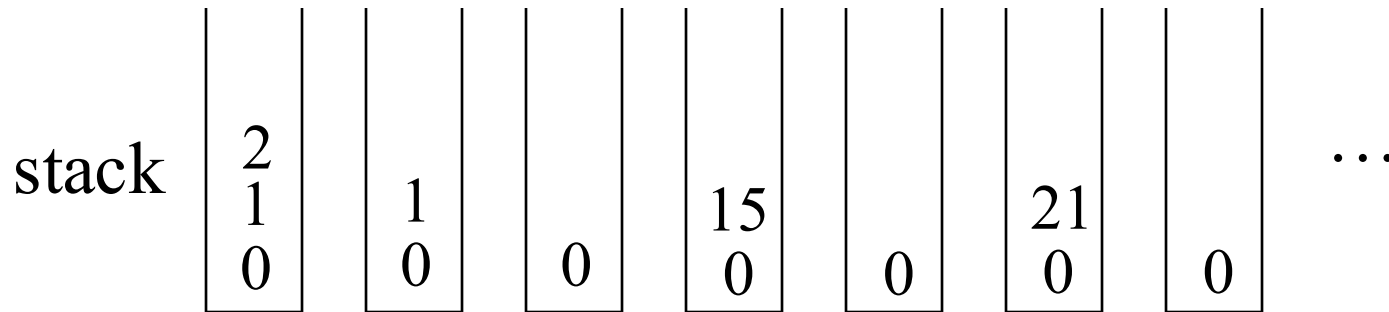
3,7

0,10



# Example

$$(((a+b)*c+d-e)/(f+g)-(h+j)*(k-1))/(m-n)$$



output      (2,6) (1,13)      (15,19)      (21,25)...

# Pseudocode

**Input:** string  $s$

**Output:** List of all pairs  $(u,v)$  such that the left parenthesis at position  $u$  in string  $s$  is matched with the right parenthesis at position  $v$

**function** match\_parentheses(string  $s$ ):

$A \leftarrow$  empty stack

    result  $\leftarrow$  empty array

    for  $i$  from 0 to length( $s$ ) - 1

        if  $s[i] = '('$

$A.push(i)$

        else if  $s[i] = ')'$

            if  $A$  is not empty

$j = A.pop()$

                result.append( $(j, i)$ )

**Try yourself!**

Implement stack ADT yourself in C++  
using a doubly linked list.

# Stacks - ADT

- **New()** creates a new stack
- **Push(item)** inserts the *item* onto top of stack
- item **Pop()** removes and returns the top *item* of stack
- item **Top()** returns (but retains) the top *item* of stack (sometimes called **Peek()**)
- int **Size()** returns number of objects in stack
- Invariants
  - $S.\text{Pop}(S.\text{Push}(v))$  has the same stack state as  $S$
  - $S.\text{Top}(S.\text{Push}(v))$  returns the value  $v$



- **FIFO Principle:** *First in, First Out*
- Elements **inserted only at rear** (enqueued) end and **removed from front** (dequeued)
  - Also called “**Head**” and “**Tail**”





# Queue -Methods

- queue **New()** – Creates and returns an empty queue
- **Enqueue**(item *v*) – Inserts object *v* at the *rear* of the queue
- item **Dequeue()** – Removes the object from *front* of the queue. Error occurs if the queue is empty
- item **Front()** – Returns, but does not remove the front element. An error occurs if the queue is empty (also called **Peek()**)
- int **Size()** – number of items in queue

## Queue – Invariants

- $\text{Front}(\text{Enqueue}(\text{New}(), v))$  *returns the value  $v$*
- $\text{Dequeue}(\text{Enqueue}(\text{New}(), v))$  *has same queue state as  $\text{New}()$*
- $\text{Front}(\text{Enqueue}(\text{Enqueue}(Q, w), v))$  *returns the same value as  $\text{Front}(\text{Enqueue}(Q, w))$*
- $\text{Dequeue}(\text{Enqueue}(\text{Enqueue}(Q, w), v))$  *has same queue state as  $\text{Enqueue}(\text{Dequeue}(\text{Enqueue}(Q, w)), v)$*

# Array Implementation of Queue

- Using array in *circular* fashion
  - **Wraparound** using mapping function (recollect from List ADT discussion)
- A max size  $N$  is specified
- $Q$  consists of an  $N$  element array and 2 integer variables having array index:
  - $f$ : index of the front element (head, for dequeue)
  - $r$ : index of the element after the rear one (tail, for enqueue)





# Array Implementation of Queue

Q



- What does  $f=r$  mean ?

# Array Implementation of Queue



- What does  $f=r$  mean ?
- Resolve Ambiguity:
  - We will never add  $n^{\text{th}}$  element to Queue (declare full if the size of queue is  $N-1$ ) .
  - *...or maintain a separate counter...*



# Pseudo Code

## **int front()**

If  $\text{size()} == 0$  then Return `QueueEmptyException`  
Else Return  $Q[f]$

## **int Dequeue()**

If  $\text{isEmpty}()$  then Return `QueueEmptyException`  
 $v = Q[f]$   
 $Q[f] = \text{null}$   
 $f = (f+1) \bmod N$   
Return  $v$

## **Enqueue(v)**

If  $\text{size()} == N-1$  then Return `QueueFullException`  
 $Q[r] = v$   
 $r = (r+1) \bmod N$

## **int size()**

Return  $(N - f + r) \bmod N$

***Pros? Cons?***

*Compute Complexity?*

*Storage Complexity?*



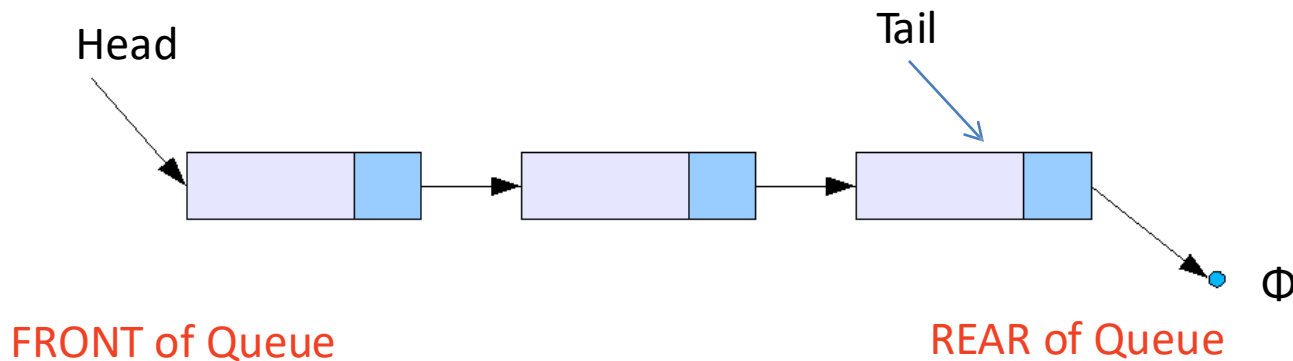
# Queue using a Linked List

- Problem with array: Requires the number of elements (capacity) *a priori*.



# Queue using a Linked List

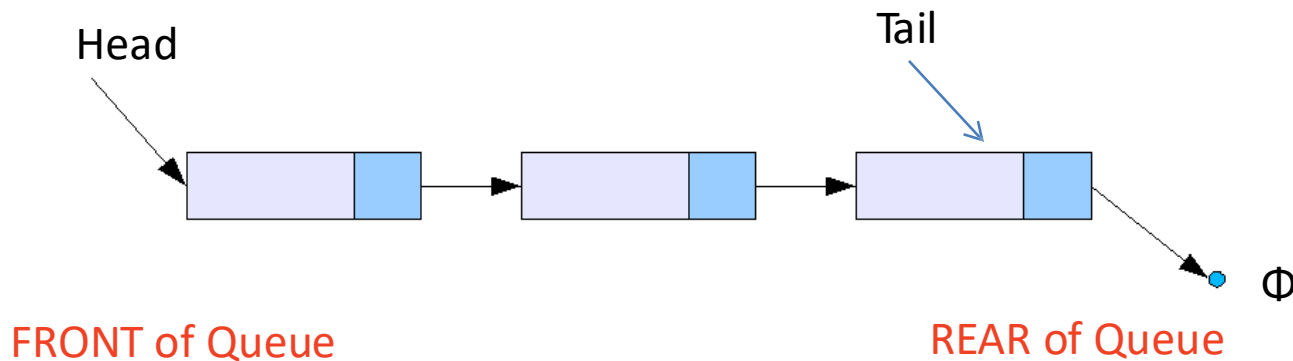
**Nodes (data, pointer)** connected in a chain by links



- Maintain two pointers, to head and tail of linked list.
- The head of the list is FRONT of the queue, the tail of the list is REAR of the queue.

# Queue using a Linked List

**Nodes (data, pointer)** connected in a chain by links



- Maintain two pointers, to head and tail of linked list.
- The head of the list is FRONT of the queue, the tail of the list is REAR of the queue.
- **Why not the opposite?**



# Priority Queues

- A queue where elements are inserted in any order BUT returned in their “priority” order
- Each item has a priority order decided by its *priority key*
  - The key may be explicitly specified, e.g., as a pair  $\langle \text{priority\_key}, \text{item} \rangle$
  - Or it may be implicit, derived from the item itself
    - Natural ordering, e.g.,  $2 < 7 \Rightarrow \text{key}(2) < \text{key}(7)$
- Keys satisfy a **total order relation** (denoted by  $\leq$ )
  - **Reflexivity:**  $k \leq k$
  - **Antisymmetric:**  $k_1 \leq k_2$  and  $k_2 \leq k_1 \Rightarrow k_1 = k_2$
  - **Transitivity:**  $k_1 \leq k_2$  and  $k_2 \leq k_3 \Rightarrow k_1 \leq k_3$



# Priority Queue

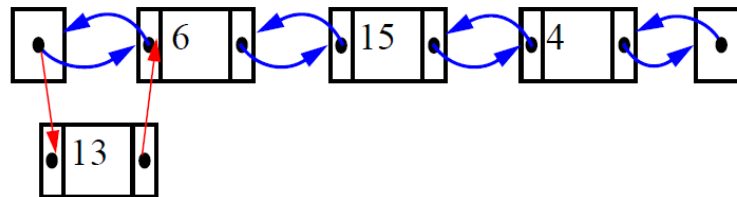
- Usually, we treat smaller keys as having higher priority
- **insert**(*k*, *i*)
- **removeMin**()
- Example
  - insert: (2,A), (4,B), (1,C), (3,B)
  - remove: (1,C), (2,A), (3,B), (4,B)
- *Items are removed in sorted order of key*



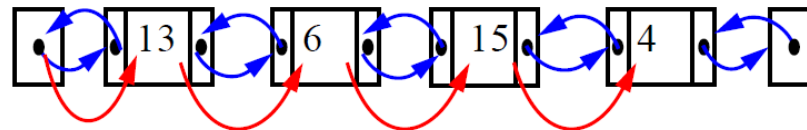
# List-based implementation

## ■ Unsorted linked list

- Insert at the start of the list,  **$O(1)$**
- Search for smallest key when removing from list,  **$O(n)$**



**insert(13)**



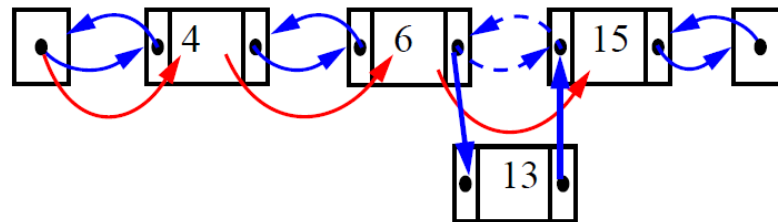
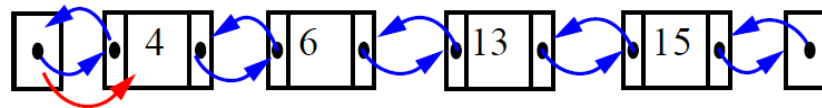
**removeMin()**

*Find smallest key*

# List-based implementation

## ■ Sorted linked list

- Insert at the correct sorted position in the list,  $O(n)$
- Return the first item from the list,  $O(1)$
- What's the time to insert `n` items?



insert(13)

Find sorted position of key

# Priority Queue is a form of Sorting!

- Insertion into **unsorted** priority queue
- **Select** and remove items in sorted order

	Sequence $S$	Priority Queue $P$
Input	(7, 4, 8, 2, 5, 3, 9)	()
Phase 1:		
(a)	(4, 8, 2, 5, 3, 9)	(7)
(b)	(8, 2, 5, 3, 9)	(7, 4)
...	...	...
(g)	()	(7, 4, 8, 2, 5, 3, 9)
Phase 2:		
(a)	(2)	(7, 4, 8, 5, 3, 9)
(b)	(2, 3)	(7, 4, 8, 5, 9)
(c)	(2, 3, 4)	(7, 8, 5, 9)
(d)	(2, 3, 4, 5)	(7, 8, 9)
(e)	(2, 3, 4, 5, 7)	(8, 9)
(f)	(2, 3, 4, 5, 7, 8)	(9)
(g)	(2, 3, 4, 5, 7, 8, 9)	()

## Time Complexity?

- $n$  steps to insert
- $n + (n-1) + (n-2) + \dots + 1$  steps to remove
- $O(n^2)$  is total time to sort

# Priority Queue is a form of Sorting!

- Insertion into correct position in **sorted** priority queue
- Remove items sequentially

	Sequence $S$	Priority Queue $P$
Input	(7, 4, 8, 2, 5, 3, 9)	()
Phase 1:		
(a)	(4, 8, 2, 5, 3, 9)	(7)
(b)	(8, 2, 5, 3, 9)	(4, 7)
(c)	(2, 5, 3, 9)	(4, 7, 8)
(d)	(5, 3, 9)	(2, 4, 7, 8)
(e)	(3, 9)	(2, 4, 5, 7, 8)
(f)	(9)	(2, 3, 4, 5, 7, 8)
(g)	()	(2, 3, 4, 5, 7, 8, 9)
Phase 2:		
(a)	(2)	(3, 4, 5, 7, 8, 9)
(b)	(2, 3)	(4, 5, 7, 8, 9)
...	...	...
(g)	(2, 3, 4, 5, 7, 8, 9)	()

## Time Complexity?

- $n + (n-1) + (n-2) + \dots + 1$  to insert
- $n$  steps to remove
- $O(n^2)$  is total time to sort



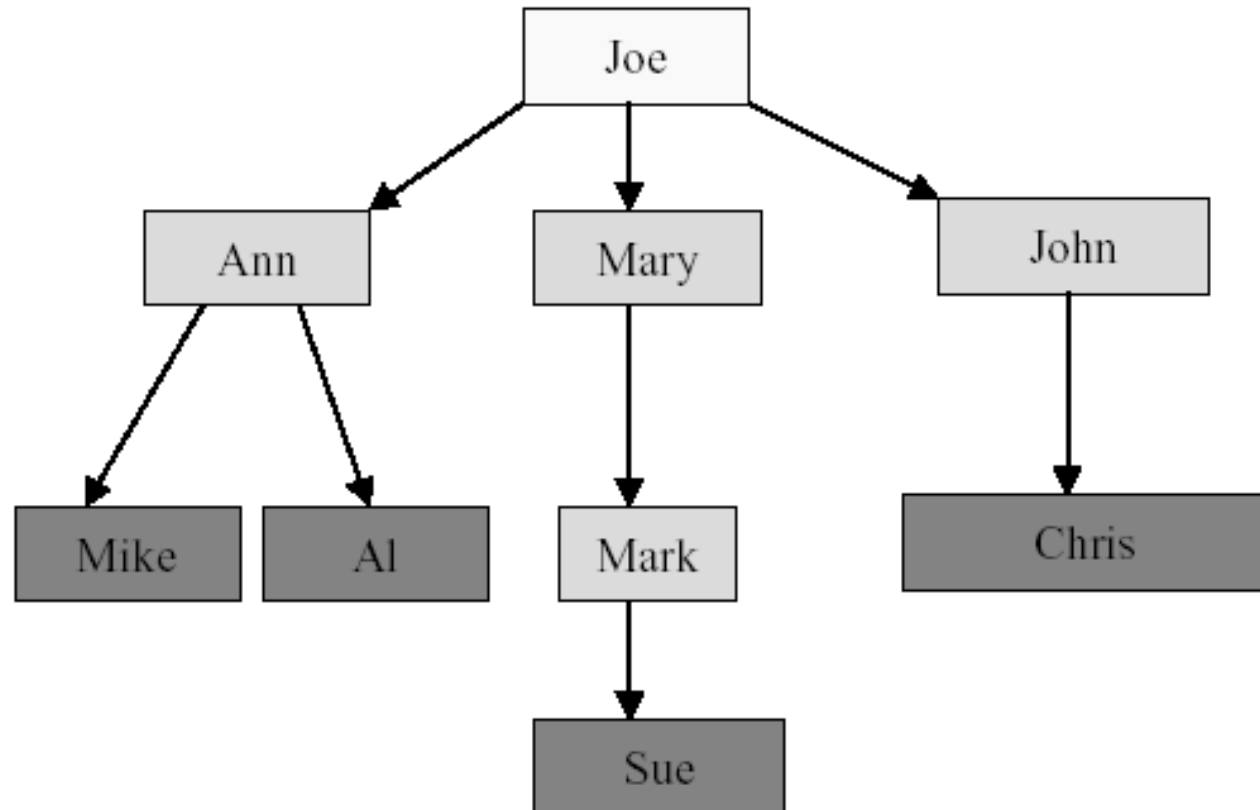
# Priority Queue as Heaps

- *Revisit after Trees*



# Linear Lists vs. Trees

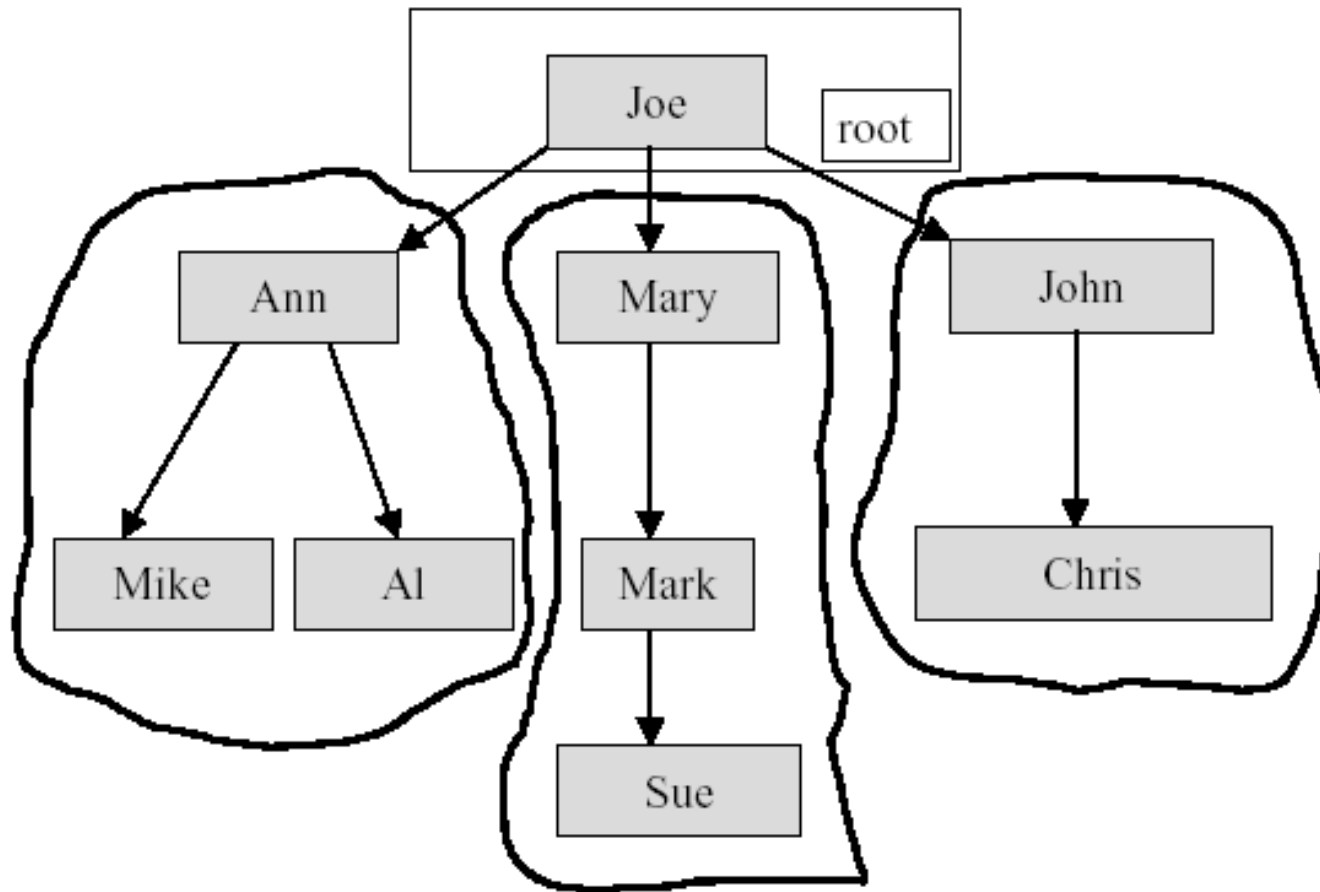
- Linear lists are useful for serially ordered data
  - $(e_1, e_2, e_3, \dots, e_n)$
  - Days of week
  - Months in a year
  - Students in a class
- Trees are useful for hierarchically ordered data
  - Joe's descendants
  - Corporate structure
  - Government Subdivisions
  - Software structure



# Definition of Tree

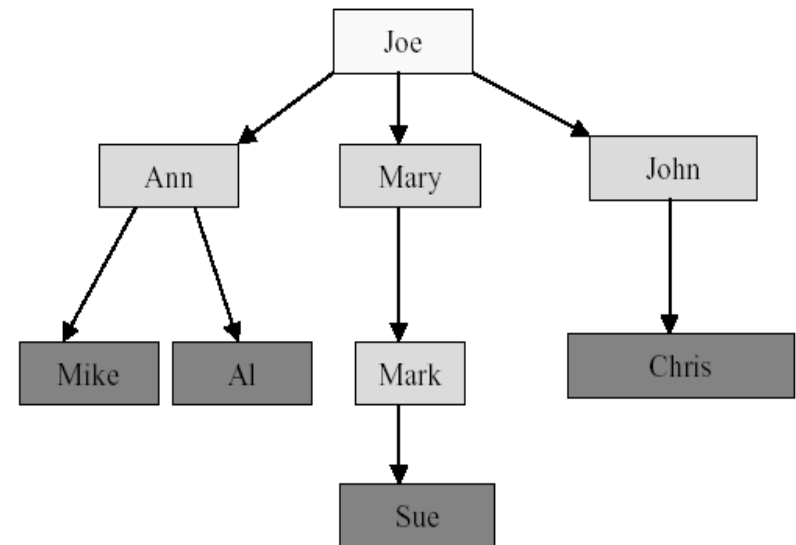
- A **tree**  $t$  is a finite non-empty set of elements
- One of these elements is designated as the **root** of tree  $t$
- The set of remaining elements, if non-empty, is uniquely partitioned into disjoint subsets, each of which constitutes a **subtree** of  $t$

# Subtrees



# Tree Terminology

- The element at the top of the hierarchy is the **root**.
- Elements next in the hierarchy are the **children** of the root.
- Elements next in the hierarchy are the **grandchildren** of the root and so on.
- Elements at the lowest level of the hierarchy are the **leaves**.



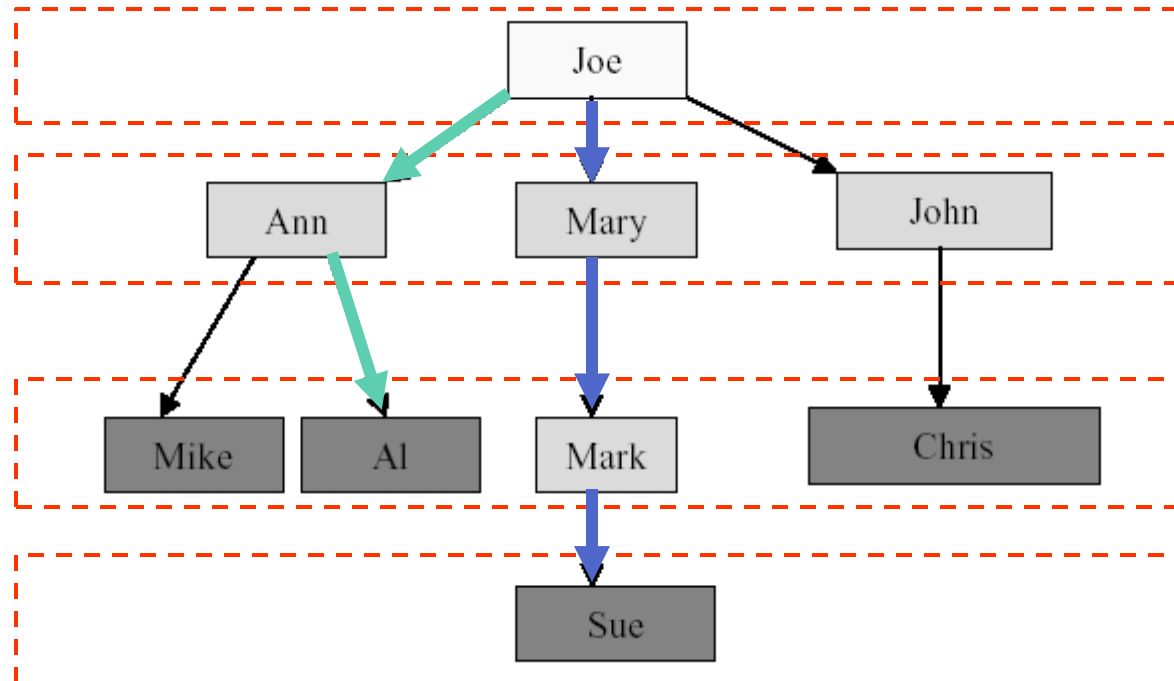


# Tree Terminology

- **Depth** of Node = No. of edges from the root to that node
- **Height** of Tree = No. of edges from root to farthest leaf
- Number of **Levels** of a Tree = Height + 1
- Node **degree** is the number of children it has

Depth(Joe) = 0

Depth(Al) = 2



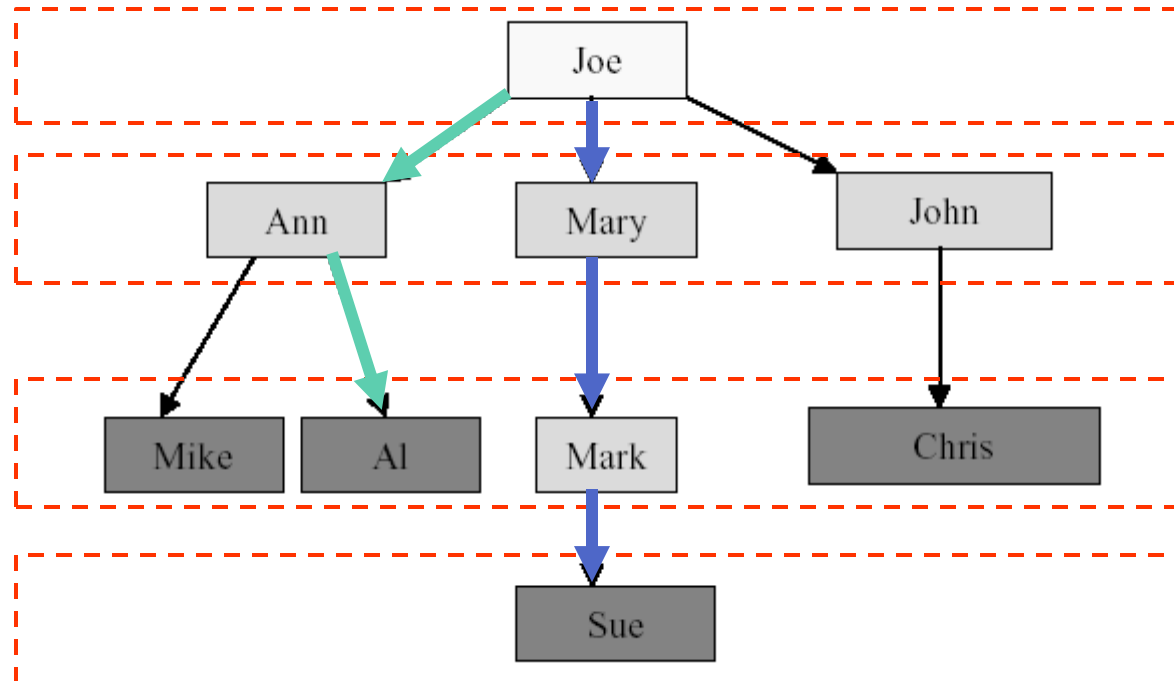
# Tree Terminology

- **Depth** of Node = No. of edges from the root to that node
- **Height** of Tree = No. of edges from root to farthest leaf
- Number of **Levels** of a Tree = Height + 1
- Node **degree** is the number of children it has

Height = 3

Depth(Joe) = 0

Depth(Al) = 2





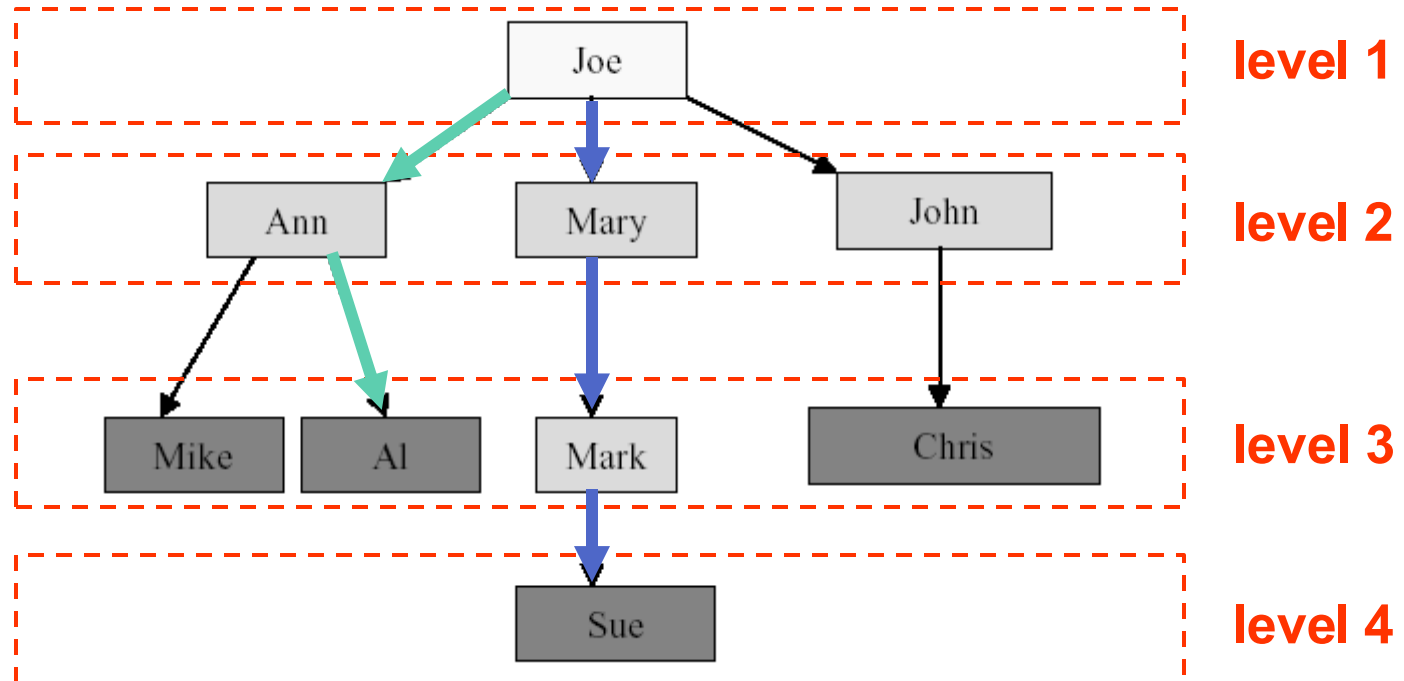
# Tree Terminology

- **Depth** of Node = No. of edges from the root to that node
- **Height** of Tree = No. of edges from root to farthest leaf
- Number of **Levels** of a Tree = Height + 1
- Node **degree** is the number of children it has

Height = 3

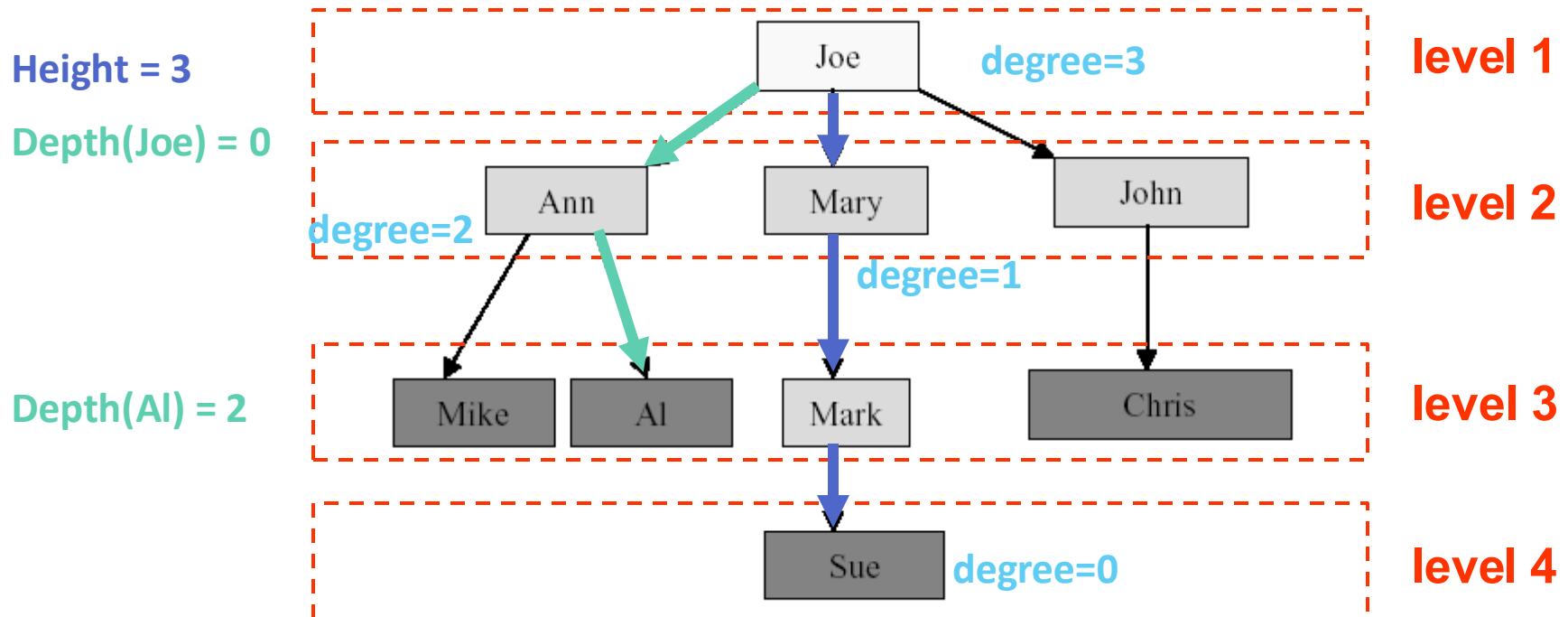
Depth(Joe) = 0

Depth(Al) = 2



# Tree Terminology

- **Depth** of Node = No. of edges from the root to that node
- **Height** of Tree = No. of edges from root to farthest leaf
- Number of **Levels** of a Tree = Height + 1
- Node **degree** is the number of children it has

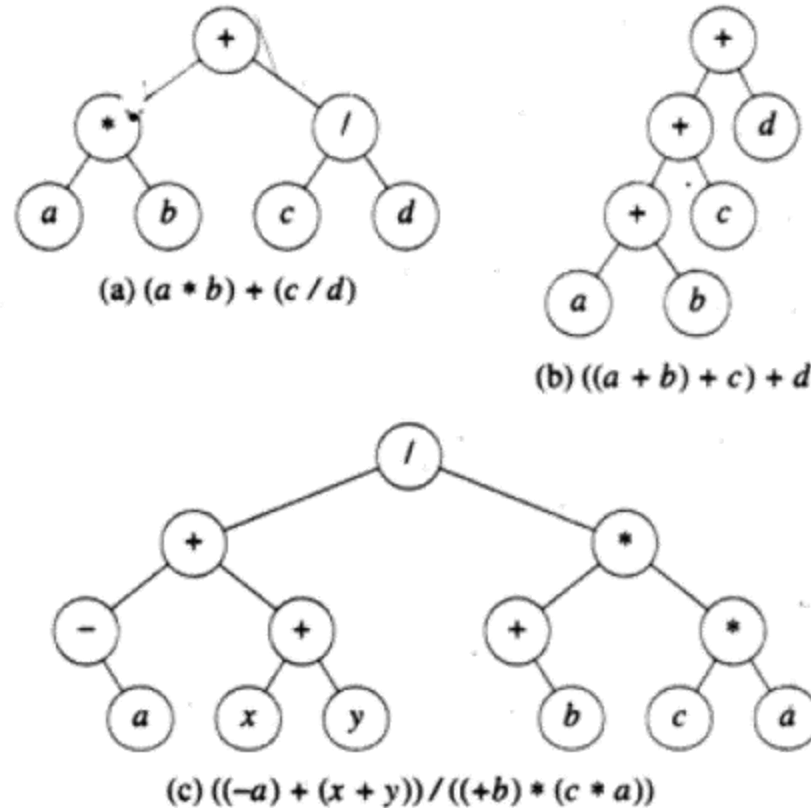




# Binary Tree

- A finite (possibly empty) collection of elements
- A **non-empty binary tree** has a **root** element and the remaining elements (if any) are partitioned into **two binary trees**
- They are called the **left** and **right sub-trees** of the binary tree

# Binary Tree for Expressions



**Figure 11.5** Expression trees



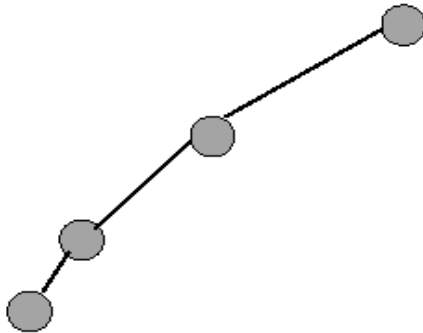
# Binary Tree Properties

1. The drawing of every binary tree with  $n$  elements,  $n > 0$ , has exactly  $n-1$  edges.
  - Each node has exactly 1 parent (except root)
2. A binary tree of height  $h$ ,  $h \geq 0$ , has at least  $h+1$  and at most  $2^{h+1}-1$  elements in it.
  - ▶  $h+1$  levels; at least 1 element at each level  $\rightarrow$  #elements =  $h+1$
  - ▶ At most  $2^{i-1}$  elements at  $i$ -th level  $\rightarrow \sum 2^{i-1} = 2^{h+1} - 1$ 
$$a + ar^1 + ar^2 + \dots + ar^n = a(r^{n+1} - 1)/(r - 1)$$

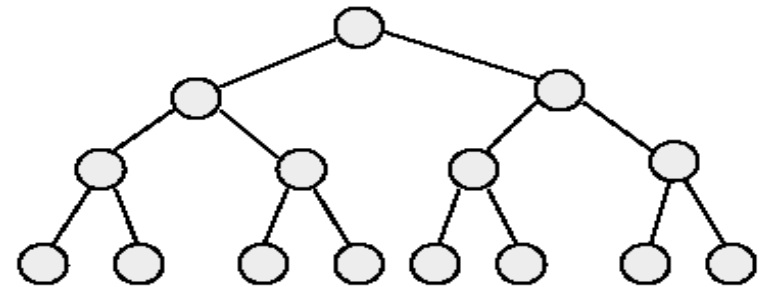
Note: Some tree definitions differ between computer science & discrete math

# Binary Tree Properties

3. The **height** of a binary tree that contains  $n$  elements,  $n \geq 0$ , is **at least**  $\lfloor \log_2 n \rfloor$  and **at most**  $n-1$ .
- At least one element at each level  $\rightarrow h_{\max} = \# \text{elements} - 1$
  - From prev:  $h_{\min} = \text{ceil}(\log(n+1))$



minimum number of elements

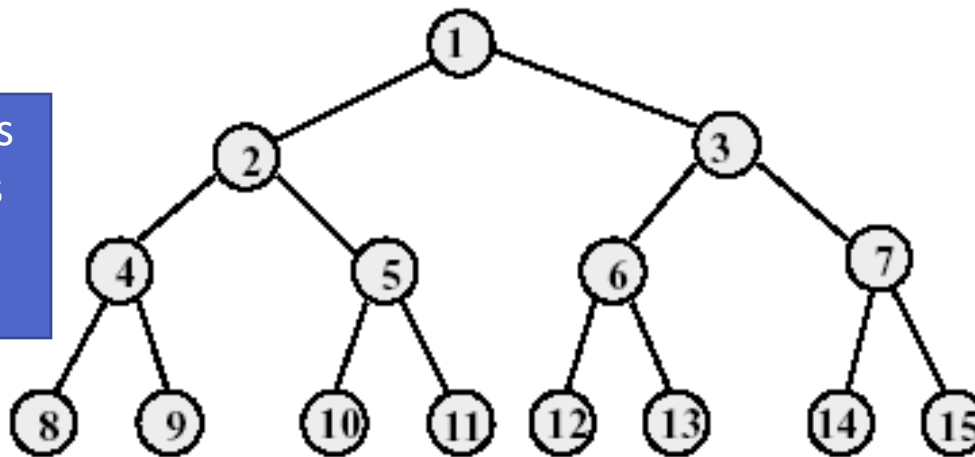


maximum number of elements

# Full Binary Tree

- A full binary tree of height  $h$  has exactly  $2^{h+1}-1$  nodes
- Numbering the nodes in a full binary tree
  - Number the nodes 1 through  $2^{h+1}-1$
  - Number by levels from top to bottom
  - Within a level, number from left to right

Note: Some definitions of full, complete trees are NOT consistently used everywhere

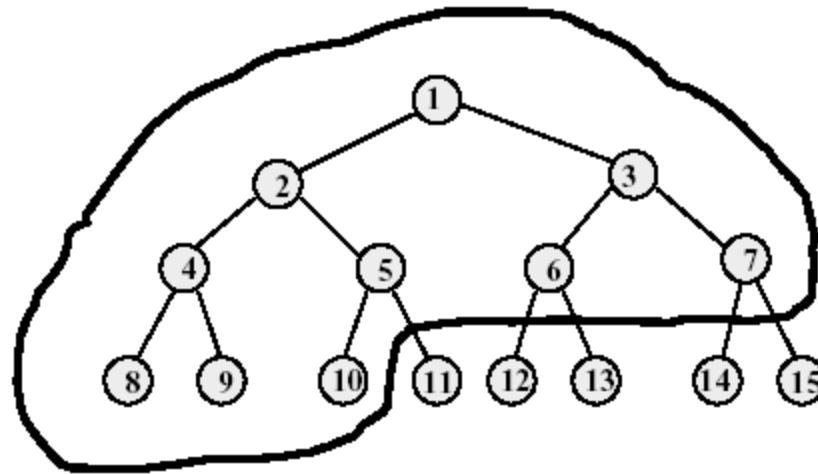




# Complete Binary Tree with N Nodes

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

# Complete Binary Tree



- Complete binary tree with 10 nodes.
- Same node number properties (as in full binary tree) also hold here.

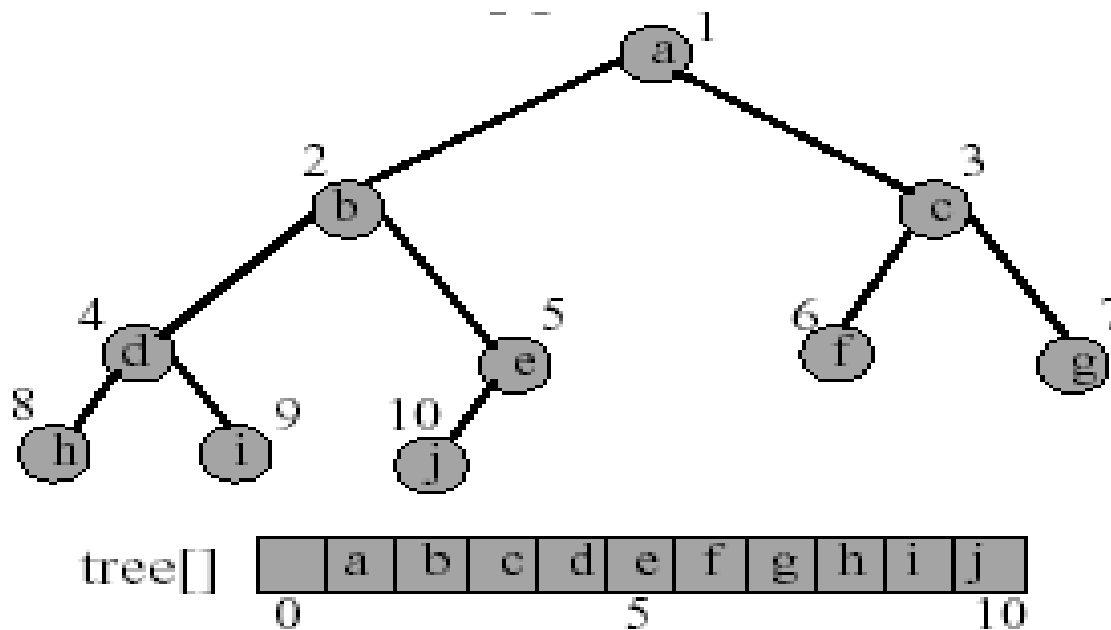


# Binary Tree Representation

- Array representation
- Linked representation

# Array Representation

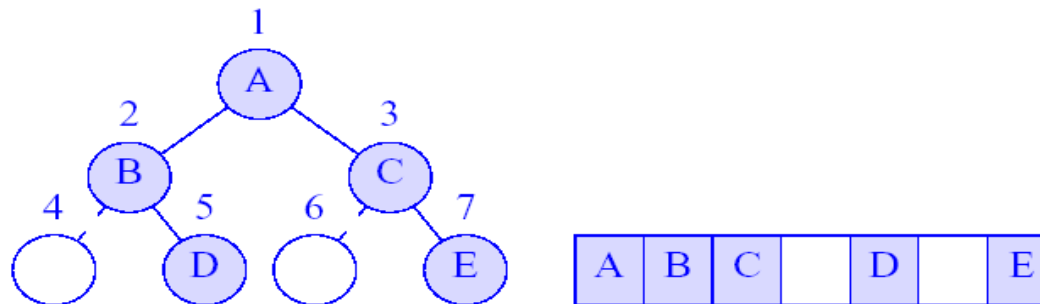
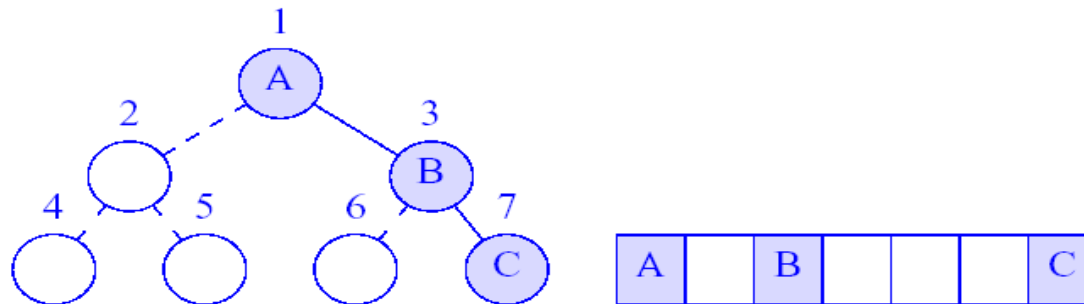
- The binary tree is represented in an array by storing each element at the array position corresponding to the number assigned to it.



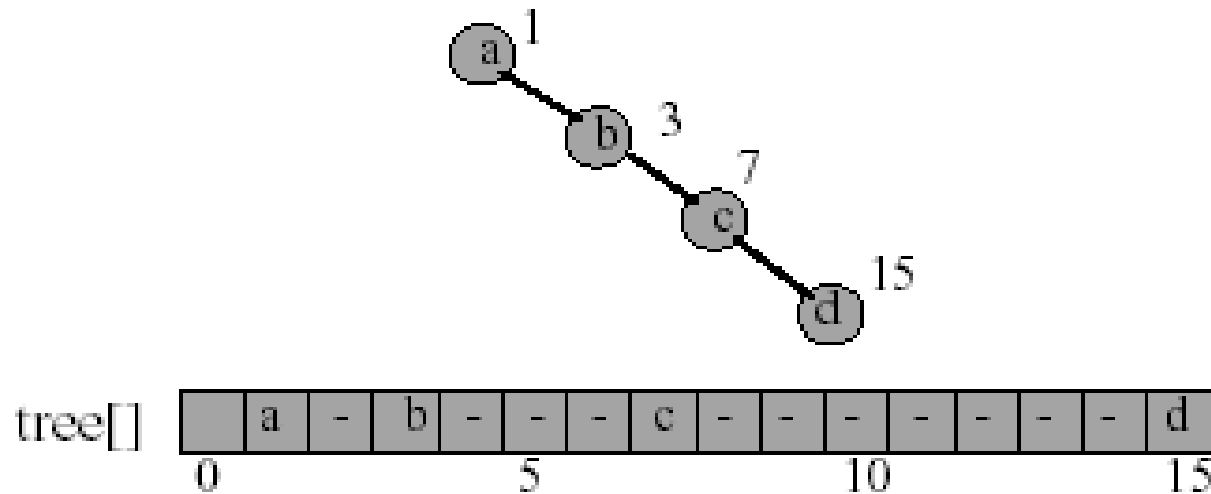


# Incomplete Binary Trees

Complete binary tree with some missing elements



# Right-Skewed Binary Tree



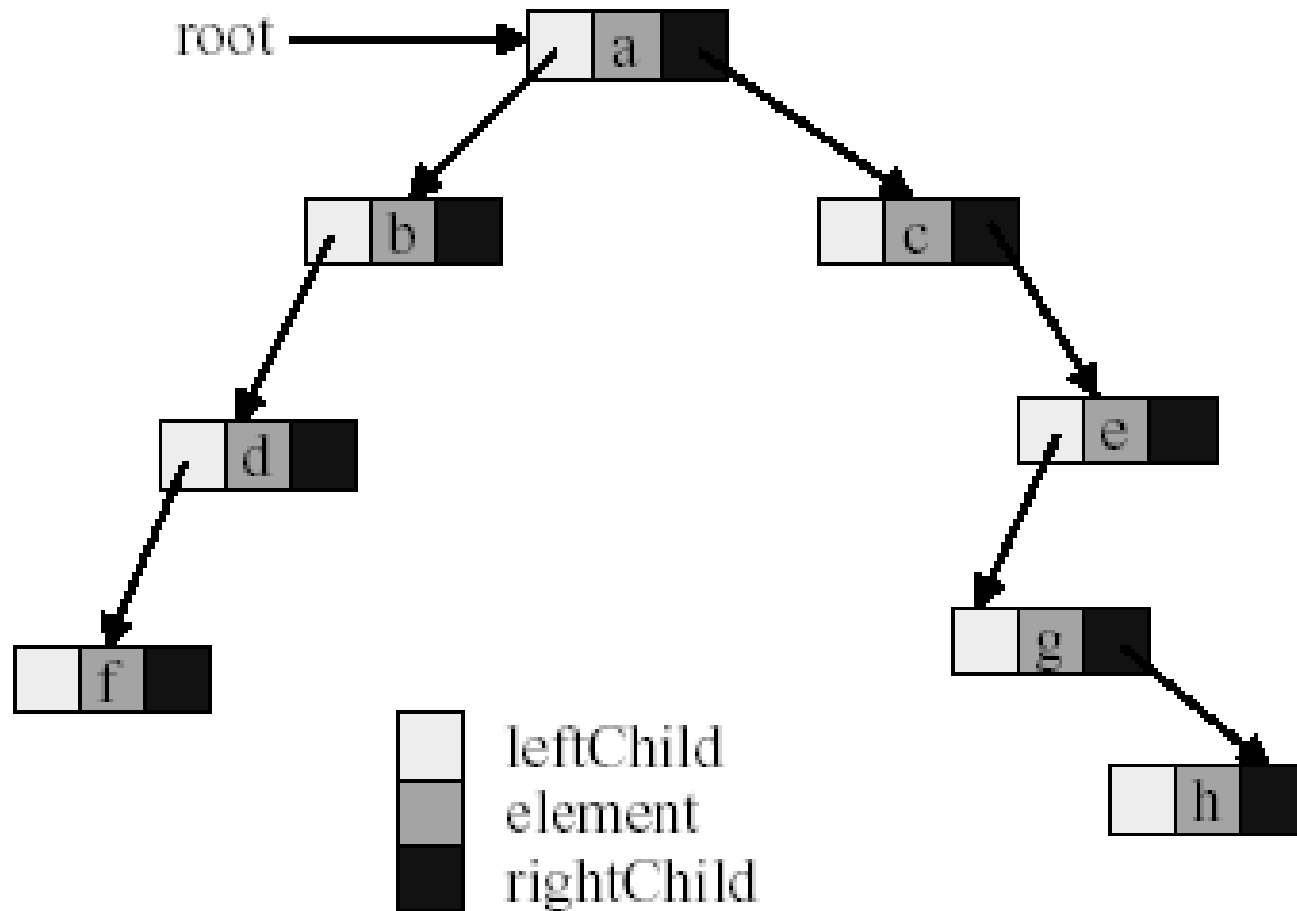
- An  $n$  node binary tree needs an array whose length is between  **$n+1$**  and  **$2^n$** .
- Right-skewed binary tree wastes the most space
- What about left-skewed binary tree?
  - *Equally bad, though with trailing blanks that could be trimmed if known ahead*



## Linked Representation

- The most popular way to present a binary tree
- Each element is represented by a node that has two link fields (`leftChild` and `rightChild`) plus an `item` field
- Each binary tree node is represented as an object whose data type is `BinTreeNode`
- The space required by an  $n$  node binary tree is  $n * \text{sizeof}(\text{BinTreeNode})$

# Linked Representation





## Node Class For Linked Binary Tree

```
class BinTreeNode {  
    int item;  
    BinTreeNode *left, *right;  
  
    BinTreeNode() {  
        left = right = NULL;  
    }  
}
```

# Binary Tree Traversal

- Many binary tree operations are done by performing a **traversal** of the binary tree
- In a traversal, each element of the binary tree is **visited exactly once**
- During the visit of an element, all **actions** (*make a copy, display, evaluate the operator, etc.*) with respect to this element are taken

# Binary Tree Traversal Methods

## ■ Preorder

- The **root** of the subtree is processed **first** before going into the **left then right subtree** (root, left, right)

## ■ Inorder

- After the complete processing of **the left subtree first** the **root** is processed followed by the processing of the complete **right subtree** (left, root, right)

## ■ Postorder

- The **left and right subtree** are completely processed, before the **root** is processed (left, right, root)

## ■ Level order

- The tree is processed one level at a time
- First all nodes in level  $i$  are processed from left to right
- Then first node of level  $i+1$  is visited, and rest of level  $i+1$  processed



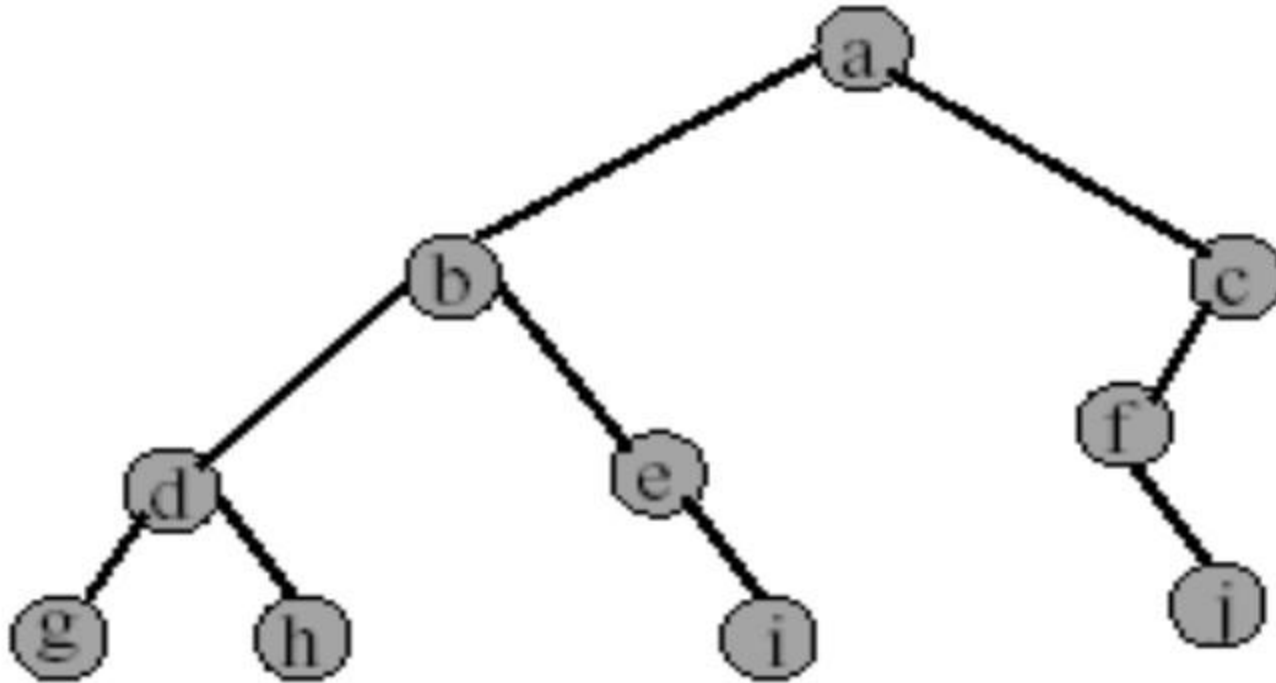
# Preorder Traversal

```
void preOrder(BinTreeNode *t) {  
    if (t != NULL) {  
        visit(t);           // Visit root 1st  
        preOrder(t->left);  // Left Subtree  
        preOrder(t->right); // Right Subtree  
    }  
}
```



# Preorder Example

*(visit action = print)*



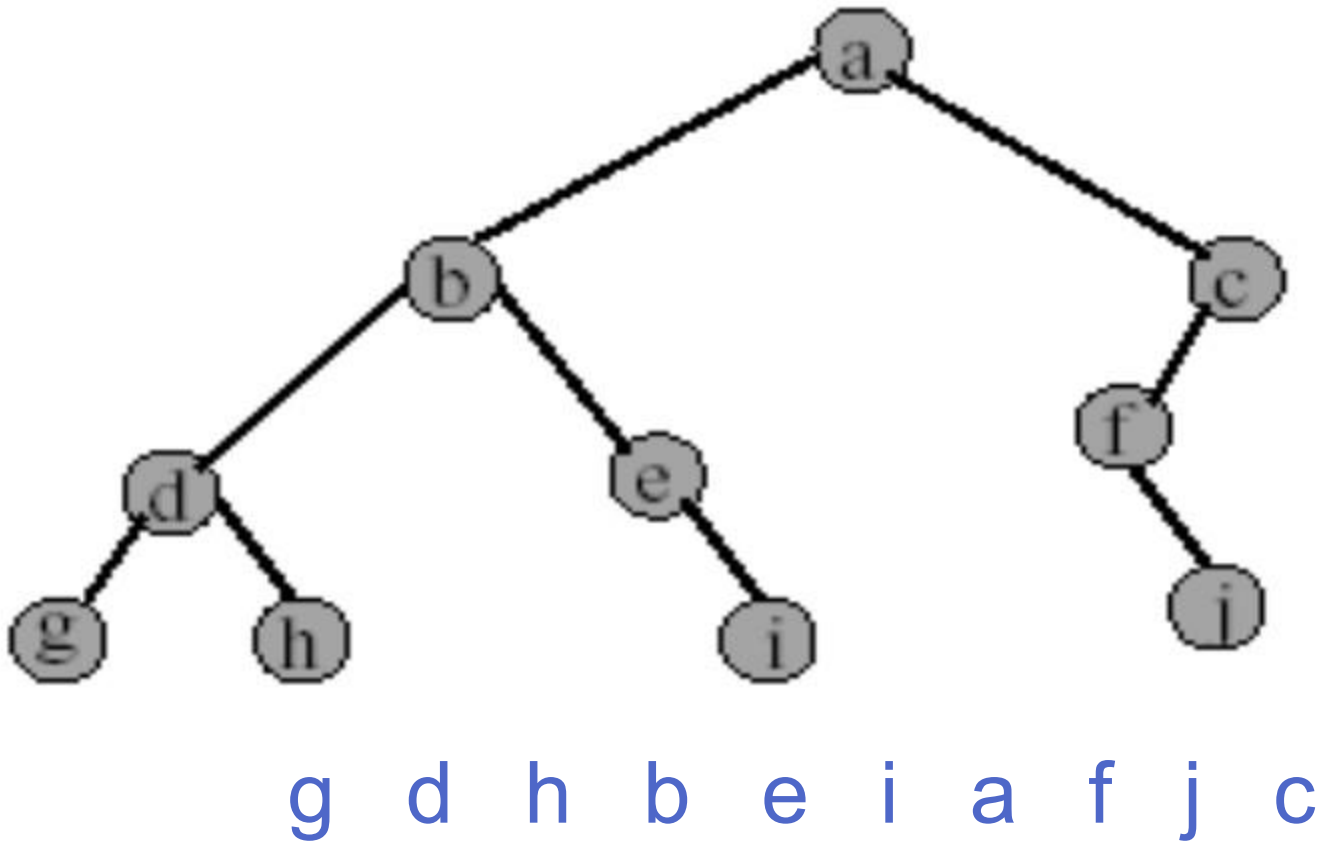
a b d g h e i c f j



# Inorder Traversal

```
void inOrder(BinTreeNode *t) {  
    if (t != NULL) {  
        inOrder(t->left);    // Left Subtree 1st  
        visit(t);            // Visit root  
        inOrder(t->right);   // Right Subtree last  
    }  
}
```

# Inorder example

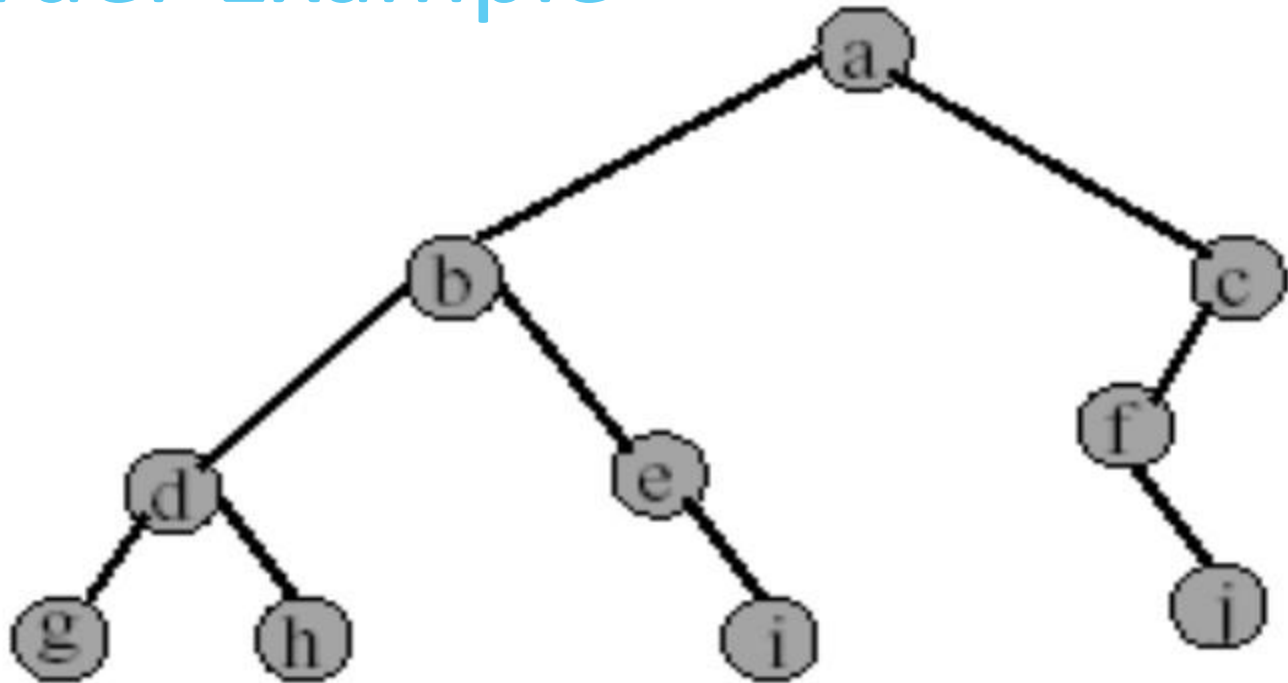




## Postorder Traversal

```
void postOrder(BinTreeNode *t) {  
    if (t != NULL) {  
        postOrder(t->left); // Left Subtree 1st  
        postOrder(t->right); // Right Subtree  
        visit(t);           // Visit root last  
    }  
}
```

# Postorder Example



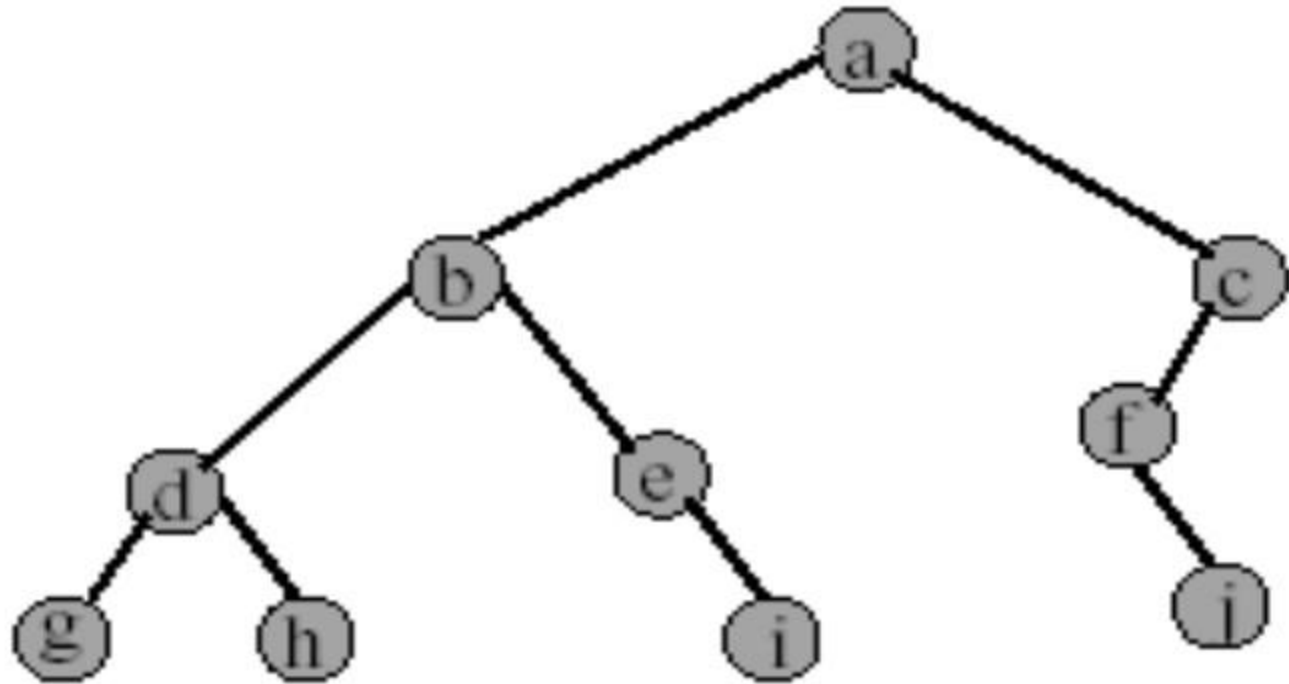
g h d i e b j f c a



## Level Order Traversal

```
void levelOrder(BinTreeNode *root){  
    Queue<BinTreeNode*> q  
    q.enqueue(root)  
    while (q is not empty) {  
        node=q.dequeue()  
        visit(node)  
        // push children to queue  
        if (node->left) q.enqueue(node->left)  
        if (node->right) q.enqueue(node->right)  
    }  
}
```

## Level Order Example



- Add and delete nodes from a queue
- Output: **a b c d e f g h i j**



# When are these traversals useful?

- Expression evaluation
- Expression printing
- Searching



# Space and Time Complexity

- The **space complexity** of each of the four traversal algorithms is  **$O(n)$** 
  - Why not  $\Theta(n)$ ? Size of recursion stack/level queue is variable.
  - Explicit Stacks vs. Recursion...*recursion dangers!*
- The **time complexity** of each of the four traversal algorithm is  **$\Theta(n)$** 
  - Each node visited only one

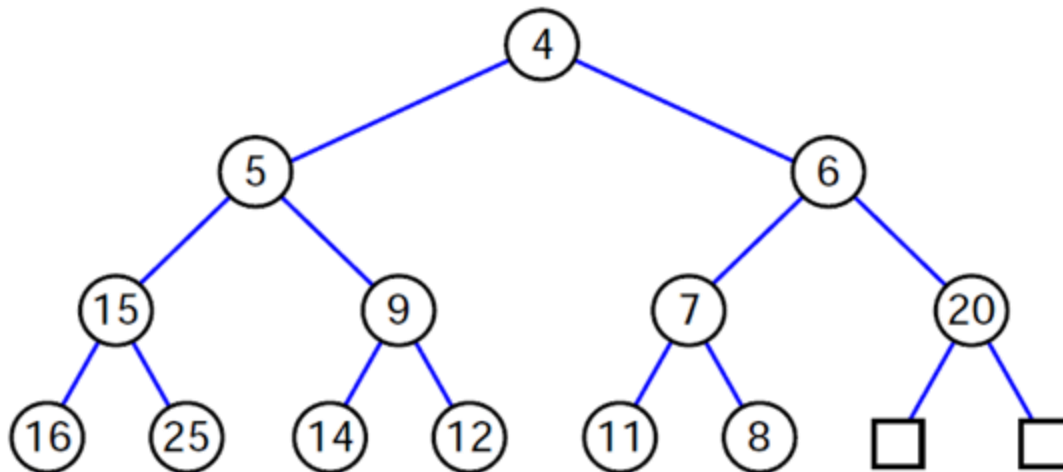


# Priority Queue as Heap

## ■ Heap data structure

- ▶ **Structural property:** Complete binary tree, i.e., every level is full, except last level which may have empty items on the right
- ▶ **Relationship property:** Key at each node is smaller or equal to the keys of its children

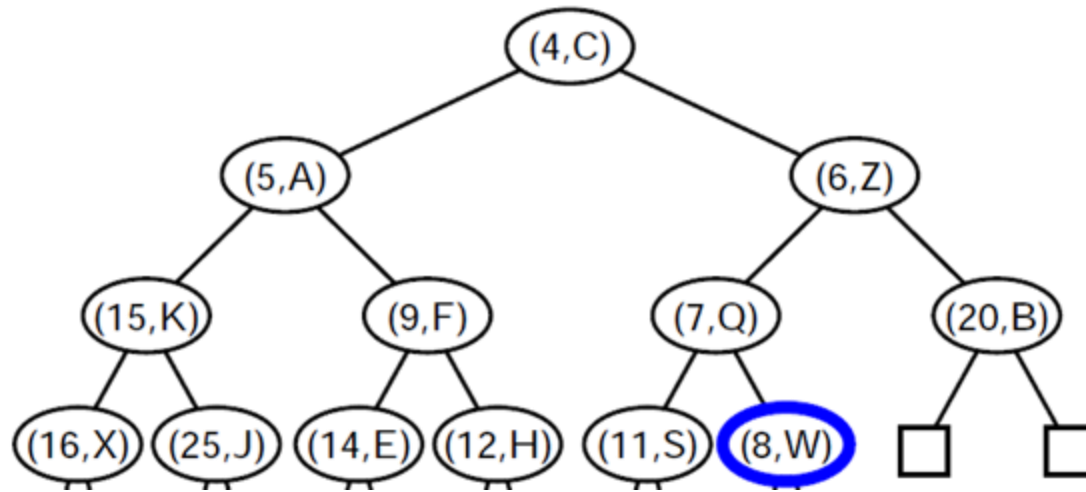
## ■ *Is there something special about the root?*





# Insertion into Heap

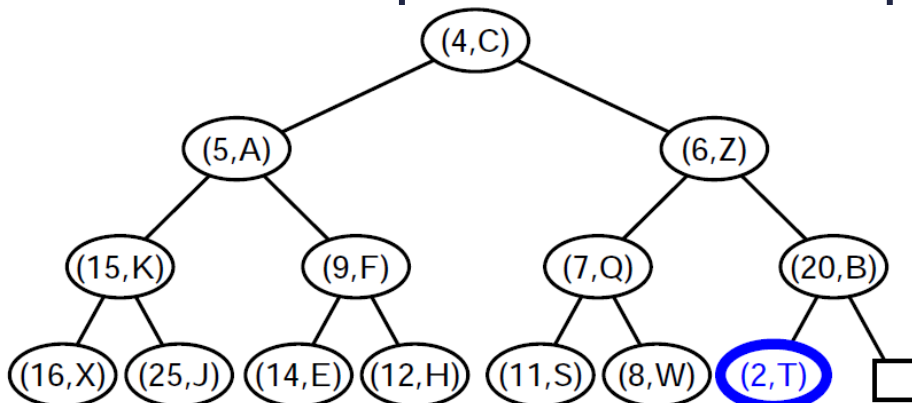
- Start by inserting at the right-most leaf of the tree



**Ideal case.** Inserted position is the correct position

# Insertion into Heap

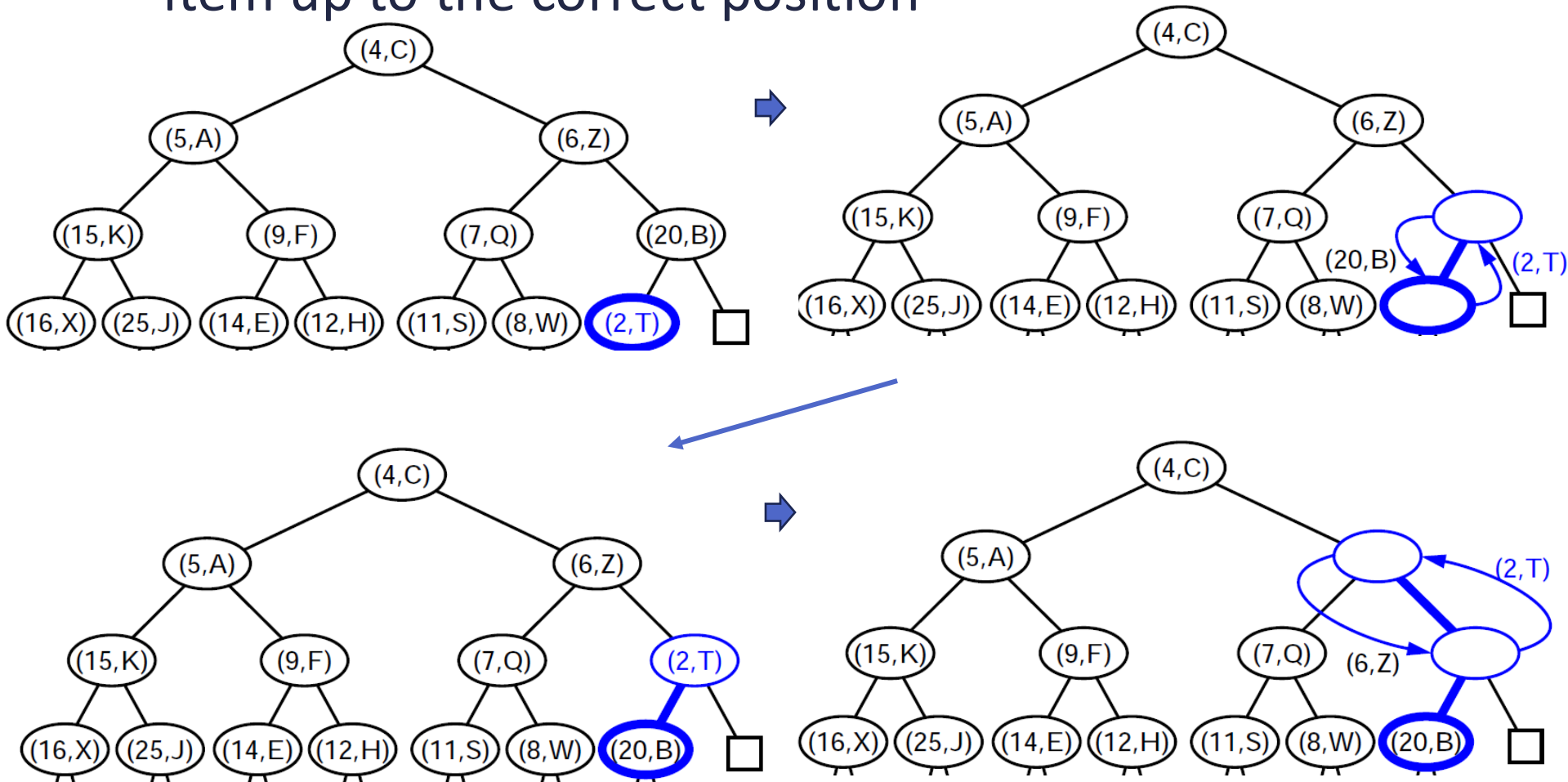
- If heap property is not maintained, “bubble” the item up to the correct position

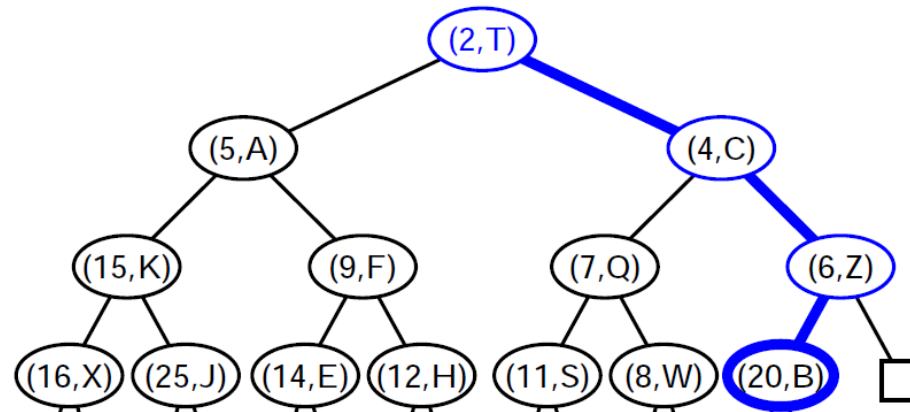
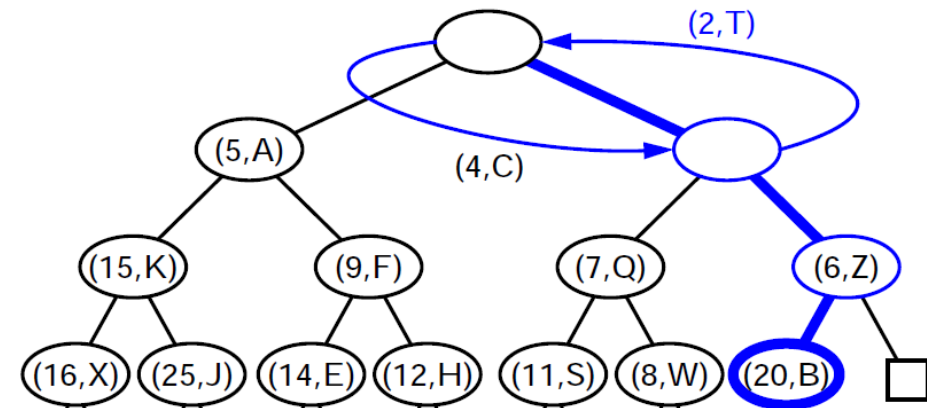
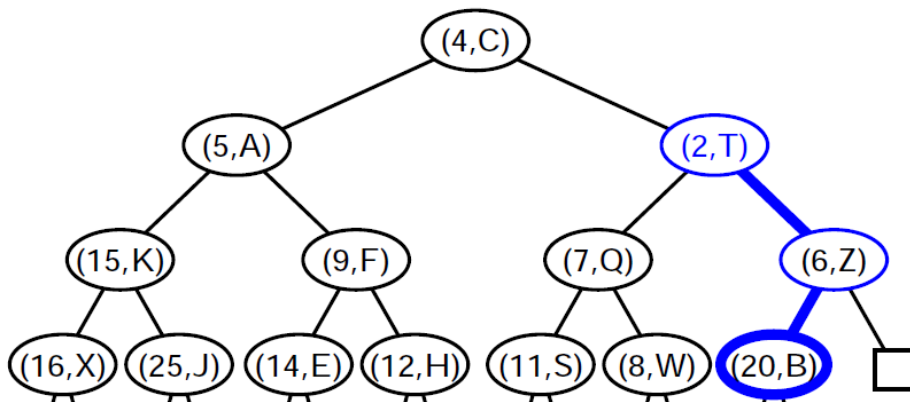




# Insertion into Heap

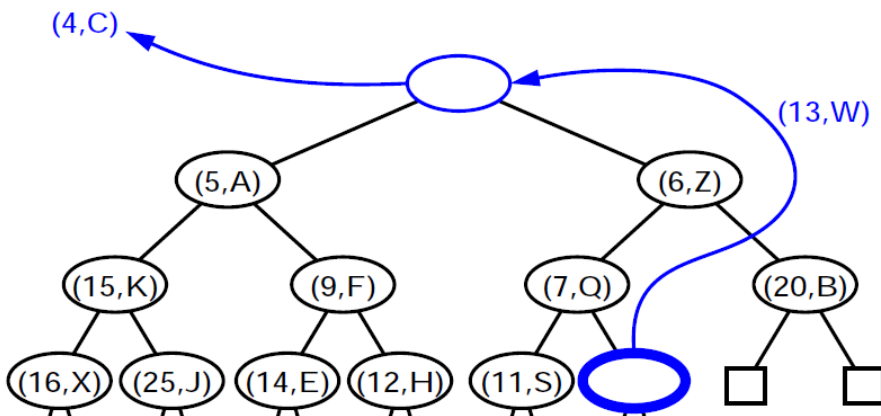
- If heap property is not maintained, “bubble” the item up to the correct position





# Removal from a Heap

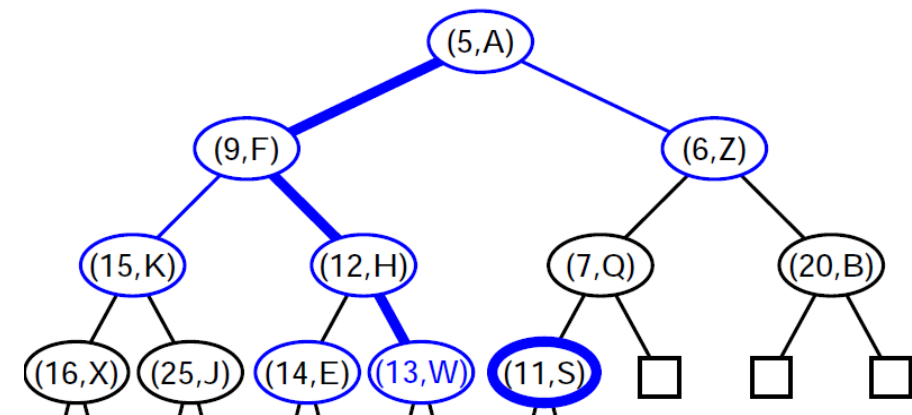
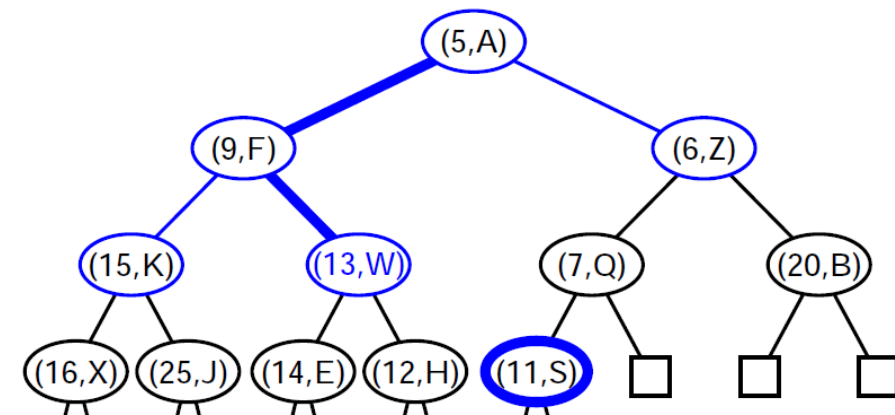
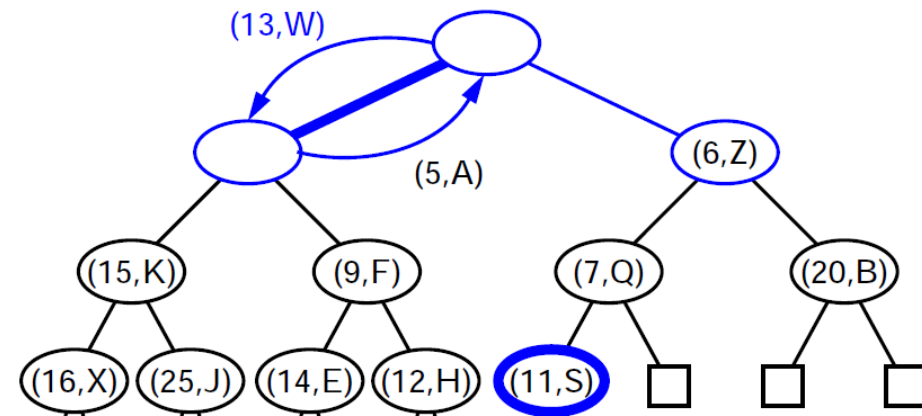
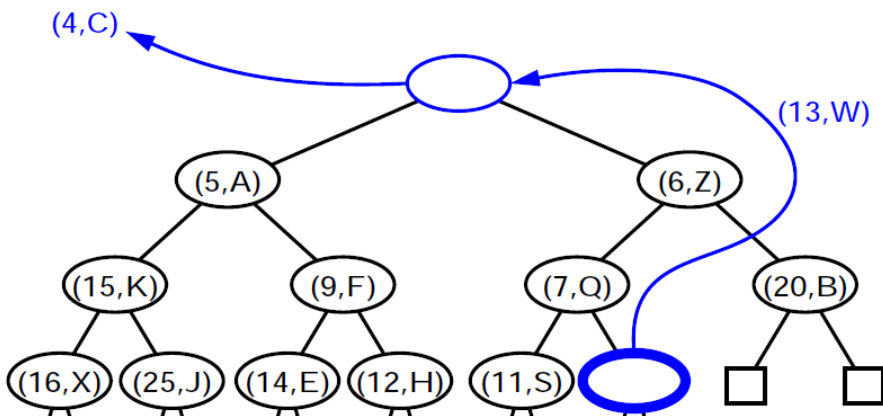
- Move in right-most leaf as root and push down to correct position





# Removal from a Heap

- Move in right-most leaf as root and push down to correct position





# Recall: Priority Queue is a form of Sorting!

- How much time does it take to create a heap of  $n$  unsorted numbers?
- How much time does it take to remove all elements from the heap?



# Recall: Priority Queue is a form of Sorting!

- How much time does it take to create a heap of  $n$  unsorted numbers?
  - Insert numbers one by one into an initially empty heap.
  - Every insertion takes  $O(\log n)$   $\rightarrow$  Total time  $O(n \log n)$
  - [Optional self study]  
Design an  $O(n)$  heap construction algorithm
- How much time does it take to remove all elements from the heap?
  - Repeatedly extract the min. Total time  $O(n \log n)$
  - [Optional self study]  
Is there a faster  $O(n)$  time algorithm?



## Live Demo: Heap sort vs default C++ sort

- Let us compare the performance of two sorting implementations
  - Our implementation based on priority queue (heap)
  - Implementation in C++ standard library (`std::sort`)

[https://github.com/cjain7/DS221-Chirag-LiveDemos/tree/main/Lecture\\_4](https://github.com/cjain7/DS221-Chirag-LiveDemos/tree/main/Lecture_4)



# Complexity Tradeoffs for Priority Queue

- Unsorted List
  - Sorted List
  - Heap
- 
- Insert vs. Remove...*which needs to be faster?*



# Tasks

- Self study (Sahni Textbook)
  - Chapter 8, Stacks
  - Chapter 9, Queues from textbook
  - Chapter 11.0-11.6, Trees & Binary Trees from textbook