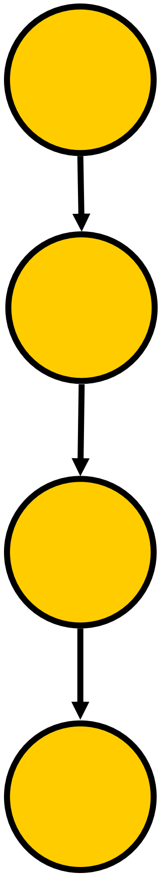
CS2040 Data Structures and Algorithms
Lecture Note #9

Trees

An introduction

Recall

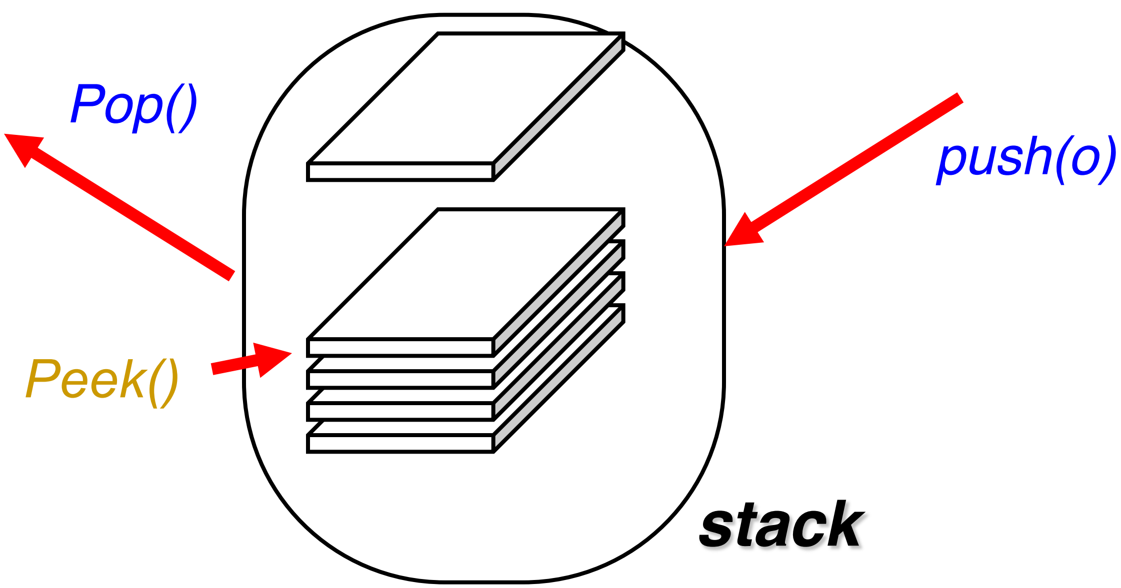
Linked list



Recall

Stack

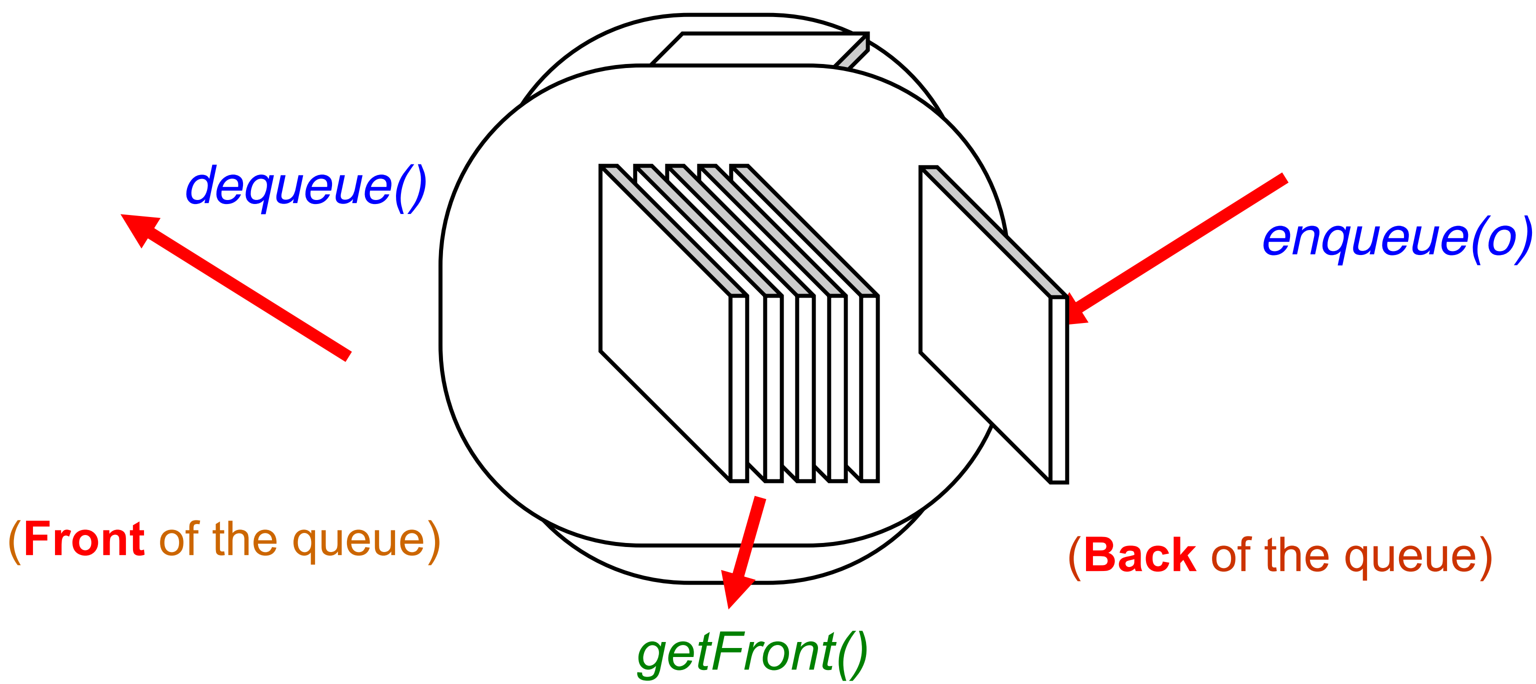
- A Stack is a collection of data that is accessed in a last-in-first-out (LIFO) manner.
- Two operations: ‘push’ and ‘pop’.



Recall

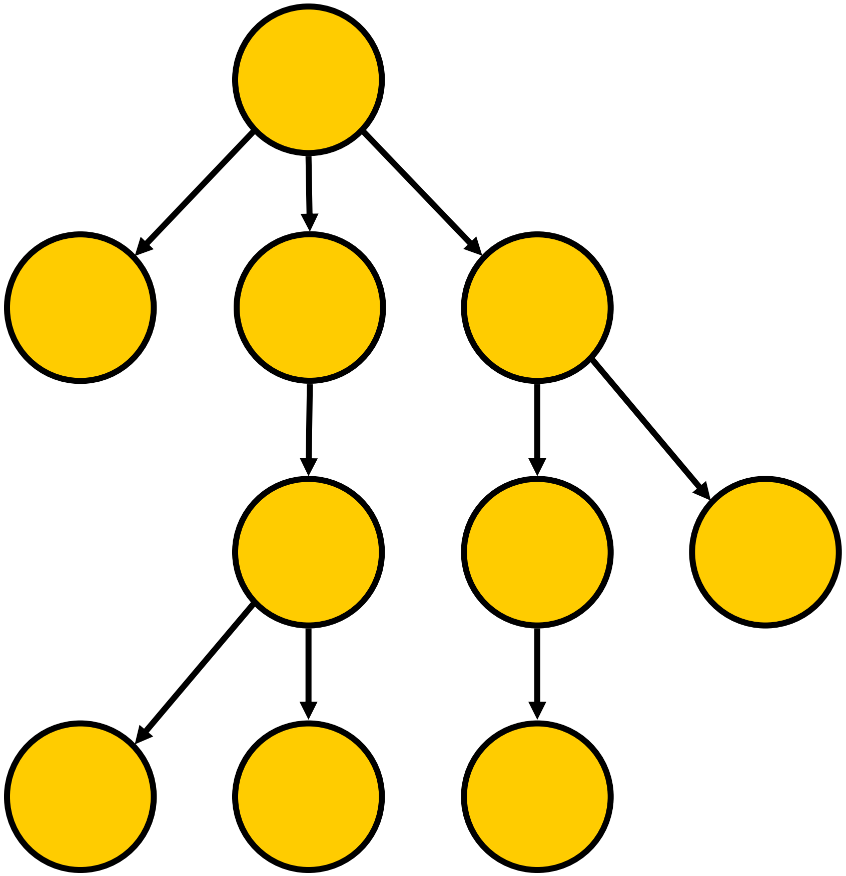
Queue

- A Queue is a collection of data that is accessed in a first-in-first-out (FIFO) manner.
- Two operators: ‘enqueue’ and ‘dequeue’



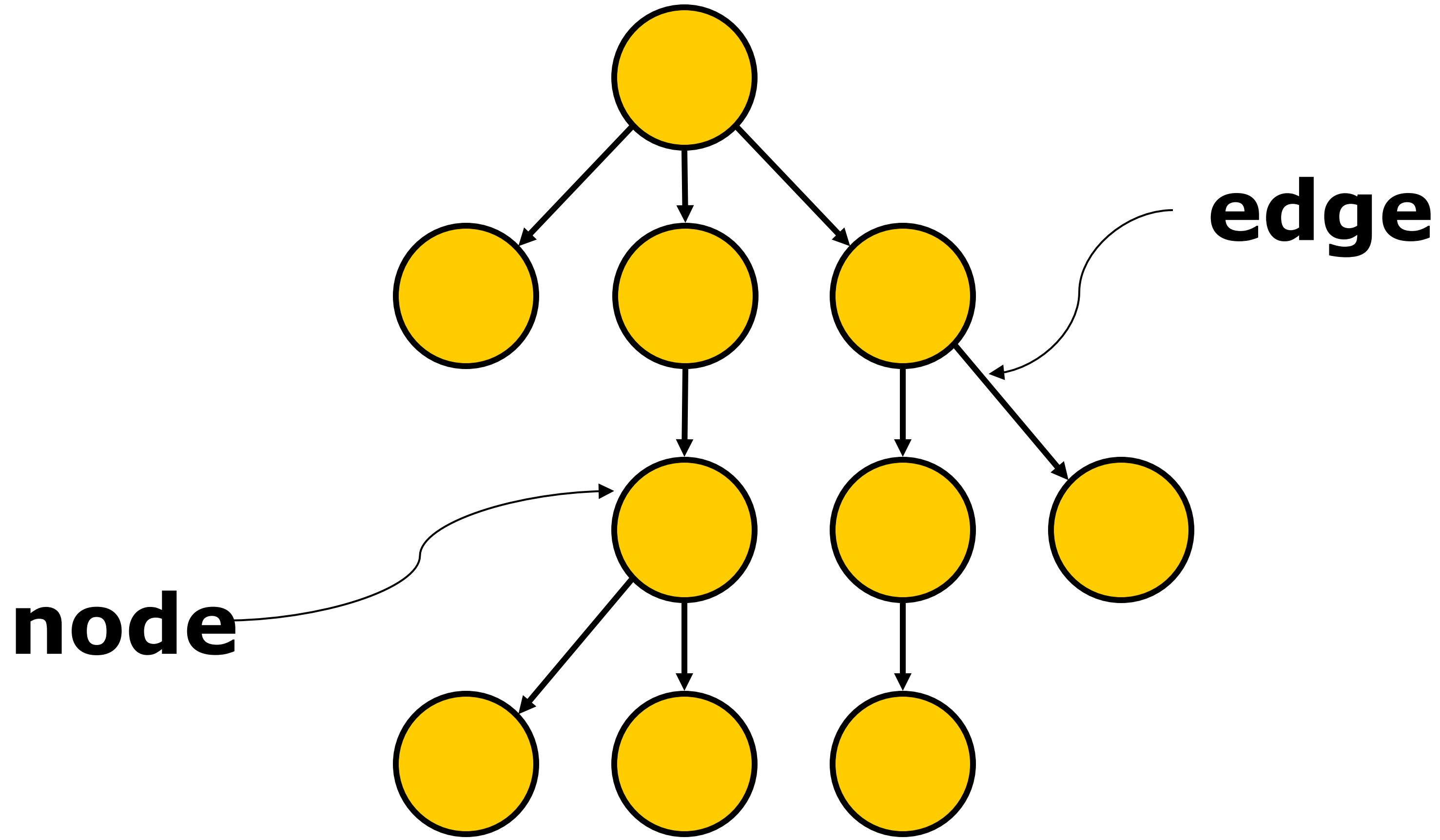
Tree

Tree



Definitions

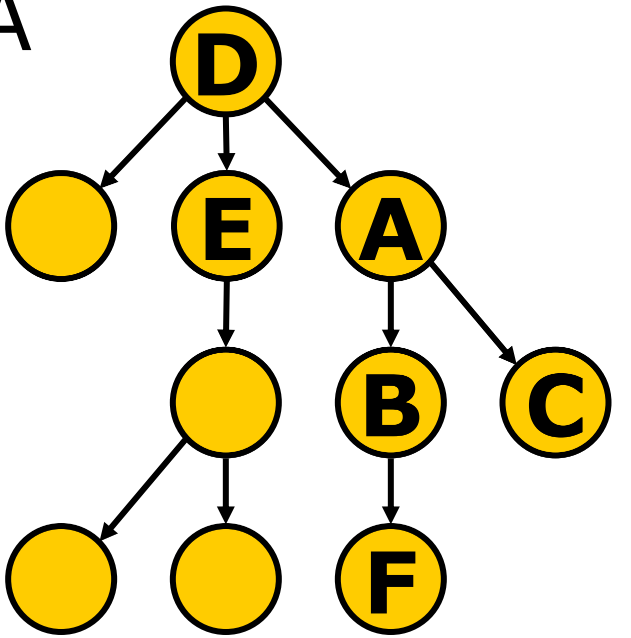
Definitions



Data objects (the circles) in a tree are called nodes (or vertices).
Links between nodes are called edges.

Relationships

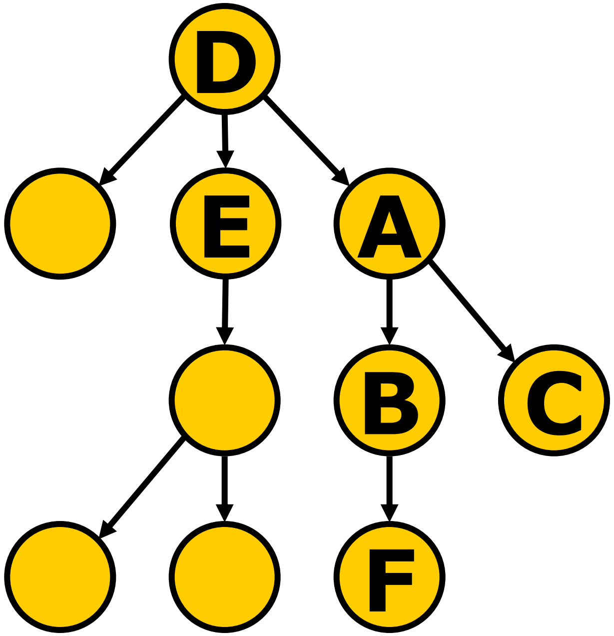
- A is a **parent** of B and C
- B and C are **children** of A
- B and C are **siblings**
  (with the same parent A)



Relationships

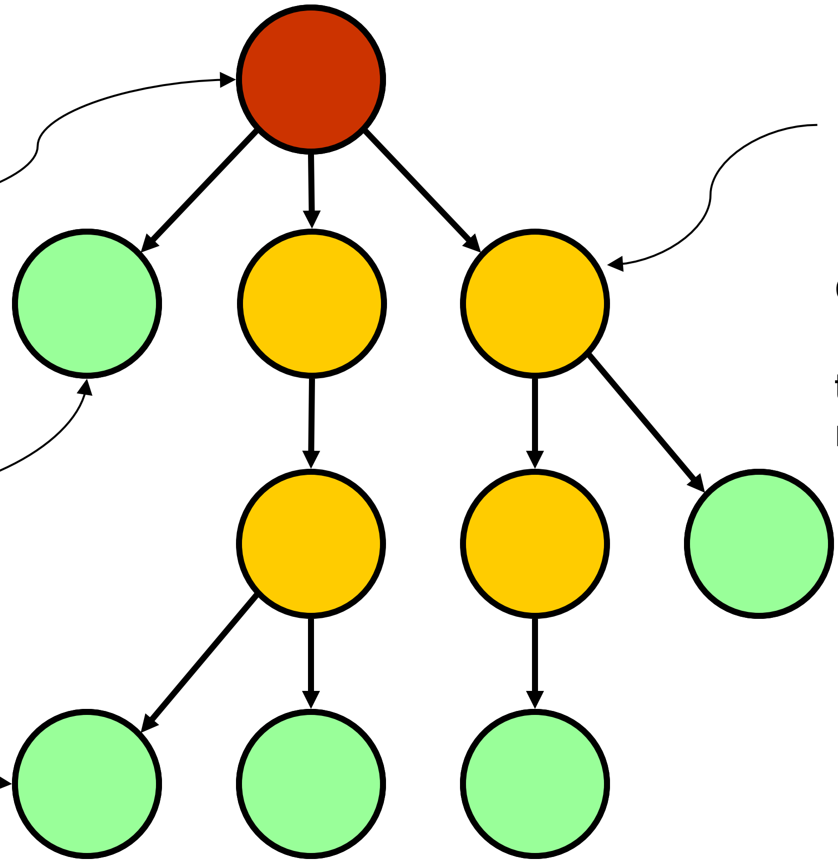
- D is an **ancestor** of B.
- B is a **descendant** of A and D.

- Definition: A is an **ancestor** of B if A is a parent of B, or A is a parent of some C and C is an ancestor of B.



Tree Nodes

root
(has no parent)



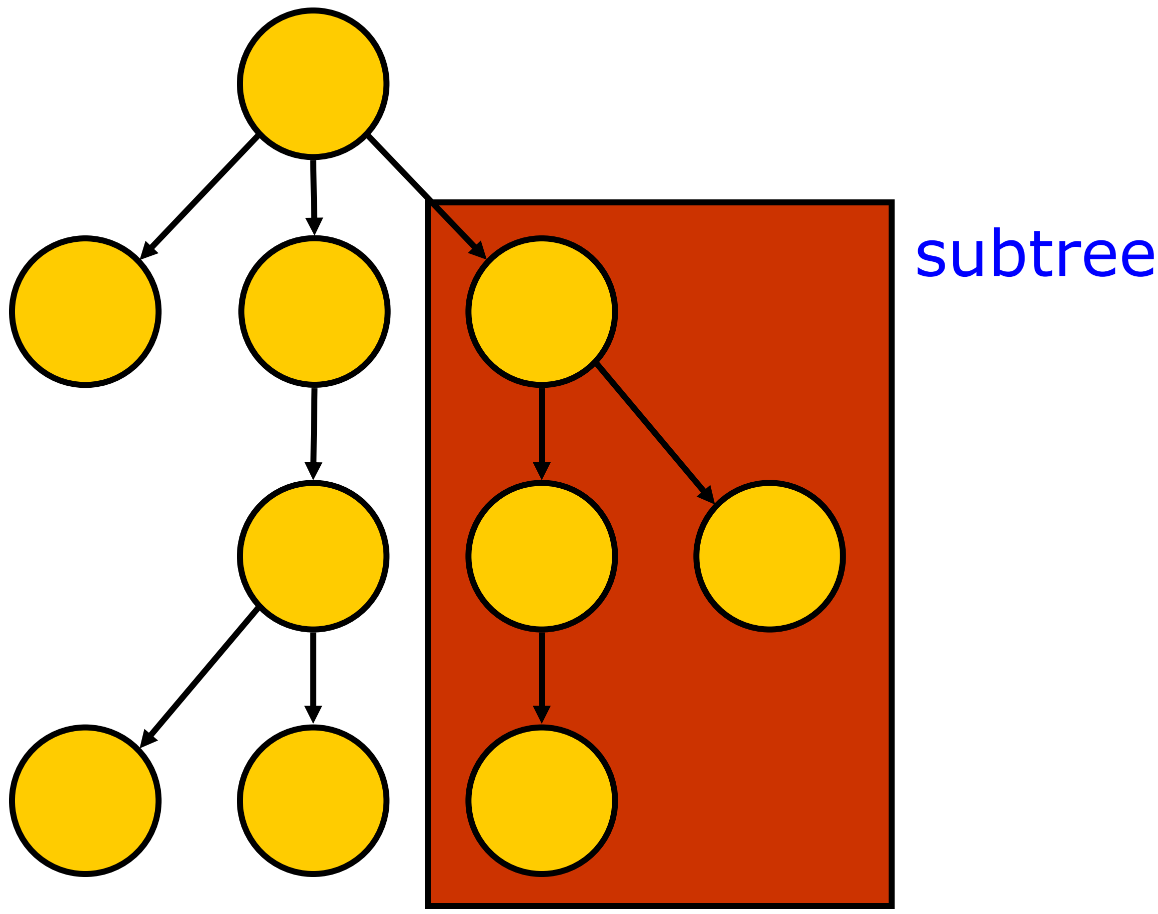
internal nodes
(has one or more children)
Note: Root of this tree is **not** an internal node.

leaves
(or **leaf nodes**,
has no children)

Every node (except the root) of a tree has one parent.

A node with no children is a leaf node.

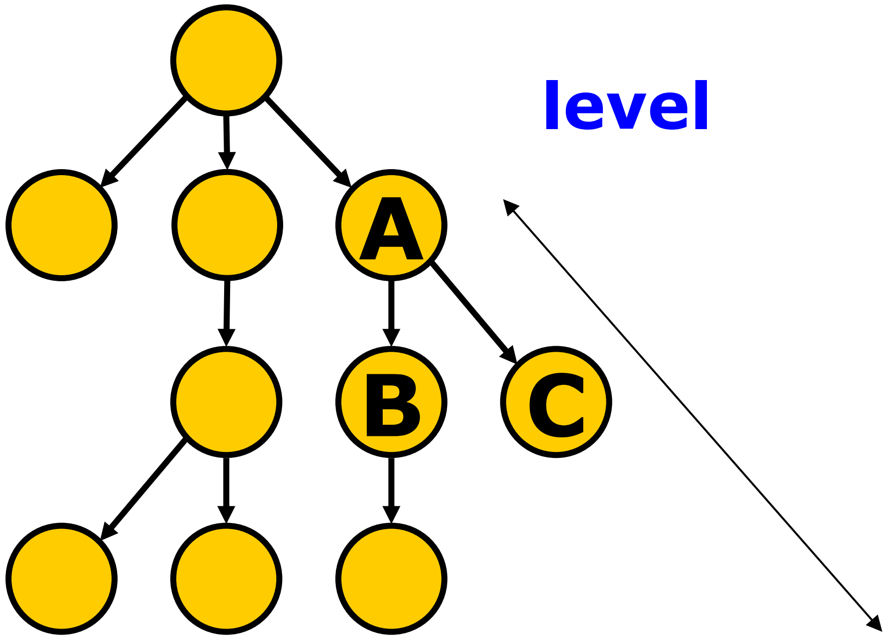
Tree is recursive!



A node and all of its descendants form a subtree

Level of a node

- **Number of nodes on the path from the root to the node (excl. the root)**
  - level of root is 0
  - level of A is 1



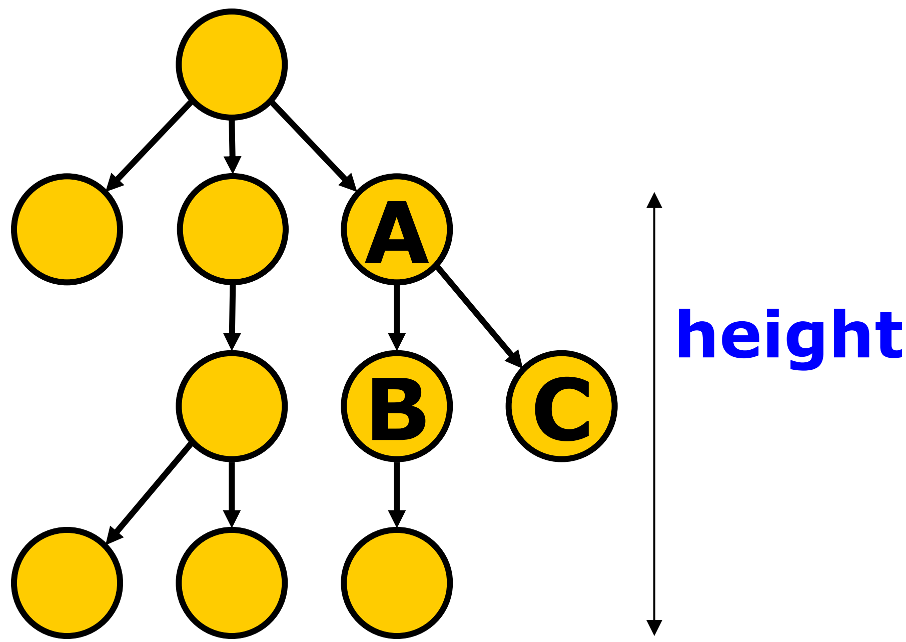
Height of a tree

- **Maximum** level of the nodes in the tree is the **height** of the tree
  - height = 3

What is the height of a tree that has only a root node?

- height = 0

Other books might give you definitions different from what you see here.

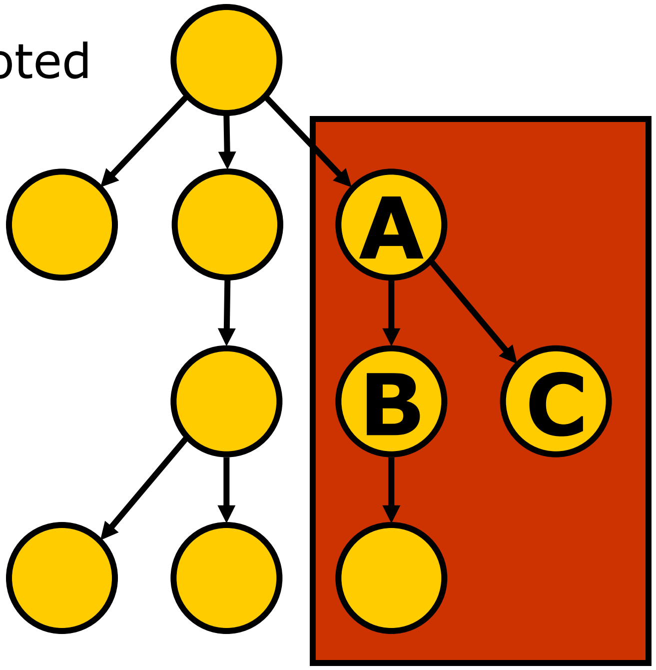


What is the height of a tree that is empty?

- height = -1 (see slide 34)

Size of a tree

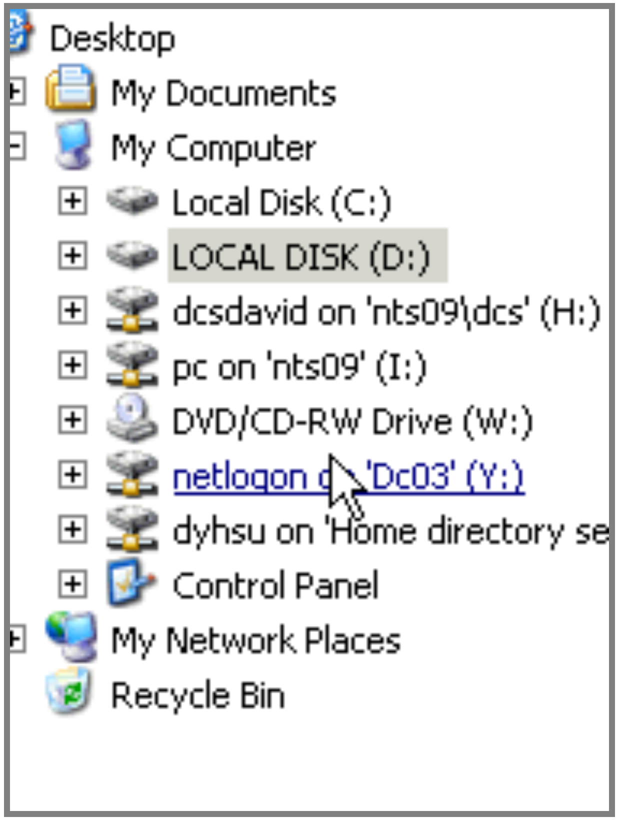
- Number of nodes in the tree is the **size** of the tree
  - The size of this tree is 10.
  - The size of the subtree rooted at A is 4.

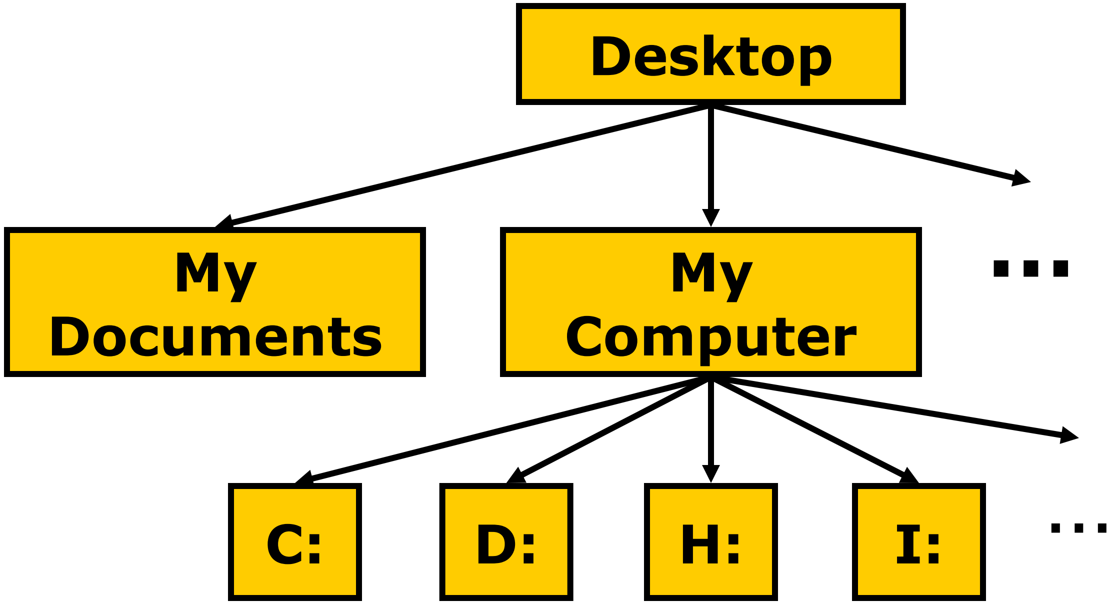


Applications of Trees

A tree can be used to represent data that is hierarchical in Nature.

File systems





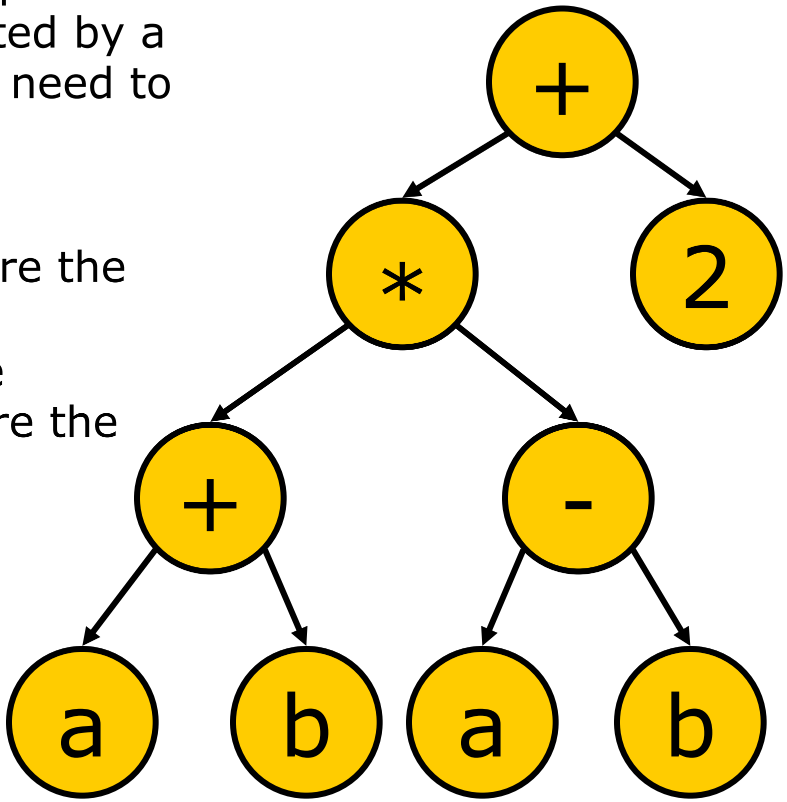
A file system can be represented as a tree, with the top-most directory as the root (in Operating System term, this is called the "root" directory).

Arithmetic Expressions

(a+b) * (a-b) + 2

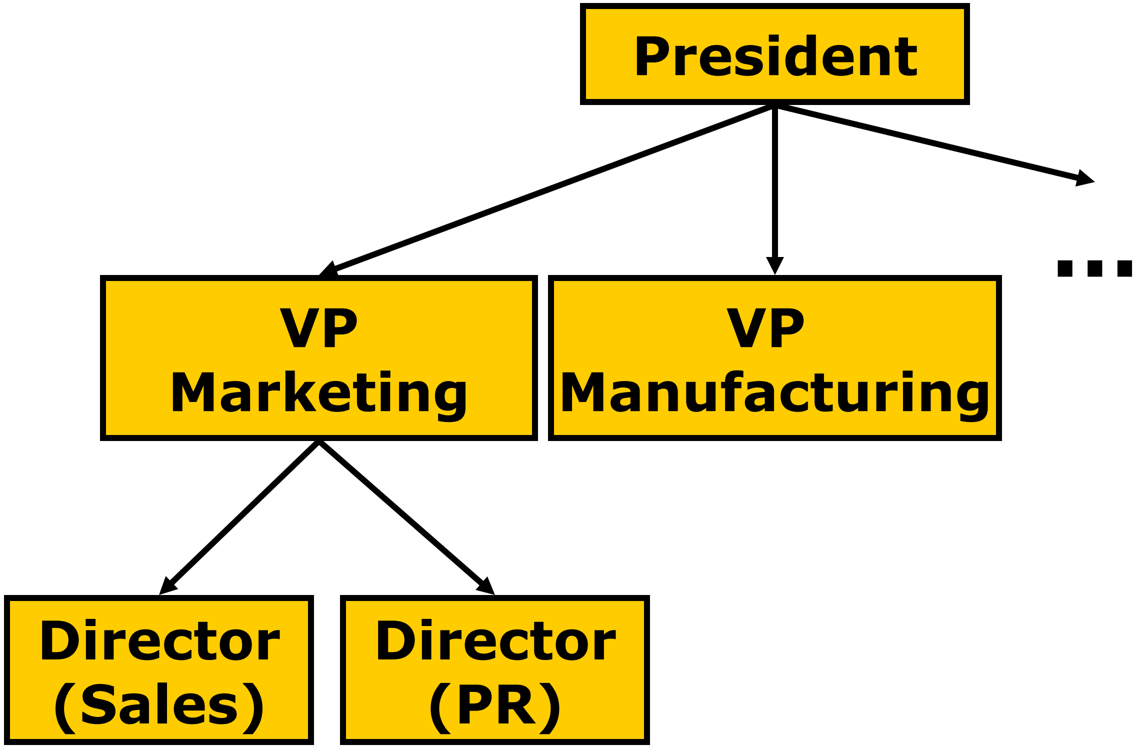
An arithmetic expression can be represented by a tree as well. No need to use brackets.

The leaf nodes are the variables/values (operands). The internal nodes are the operations.



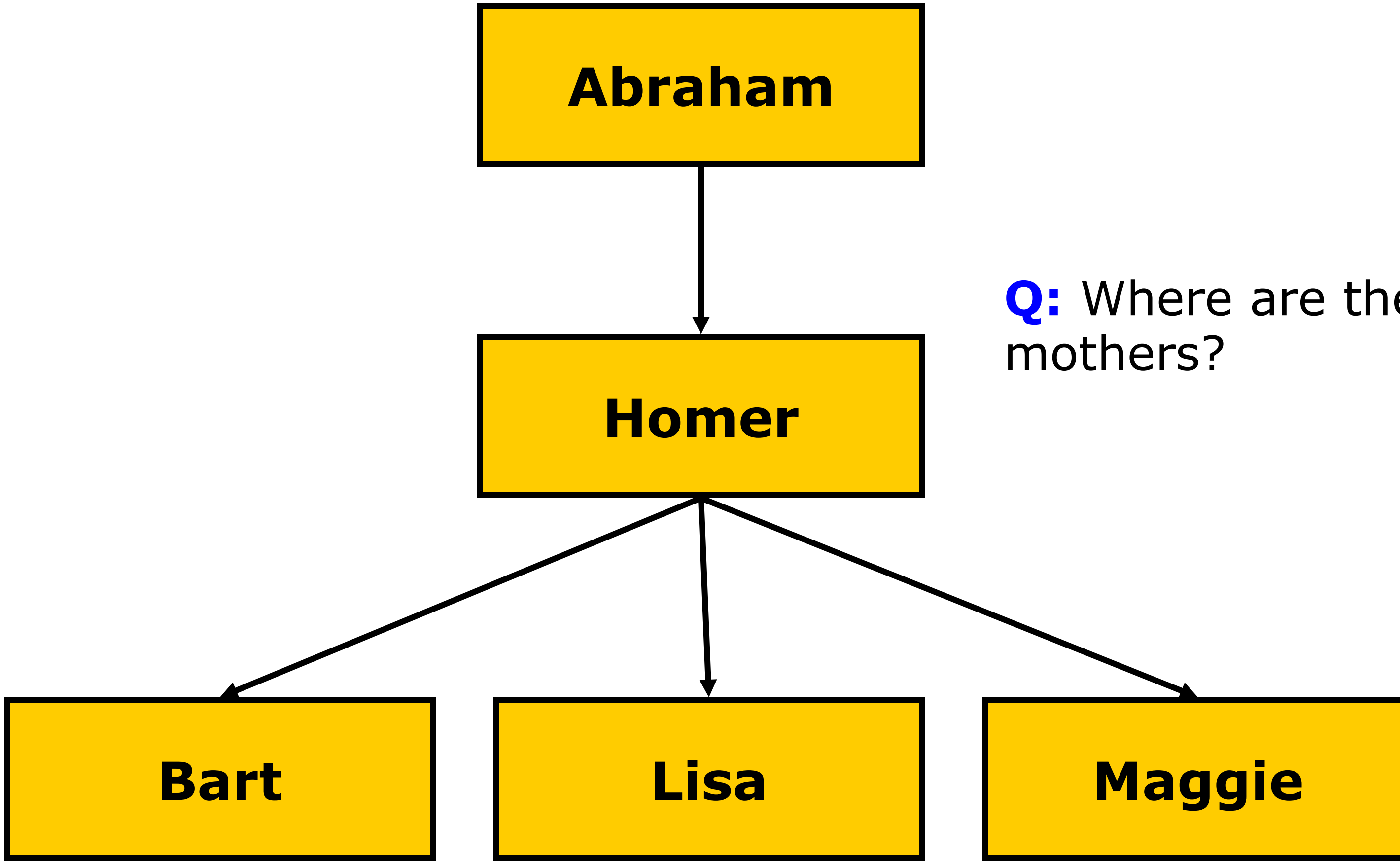
Q: How do we construct such a tree from a given arithmetic expression?

Organization Chart



Each employee (except the president) has one and only one immediate superior.

Family Tree



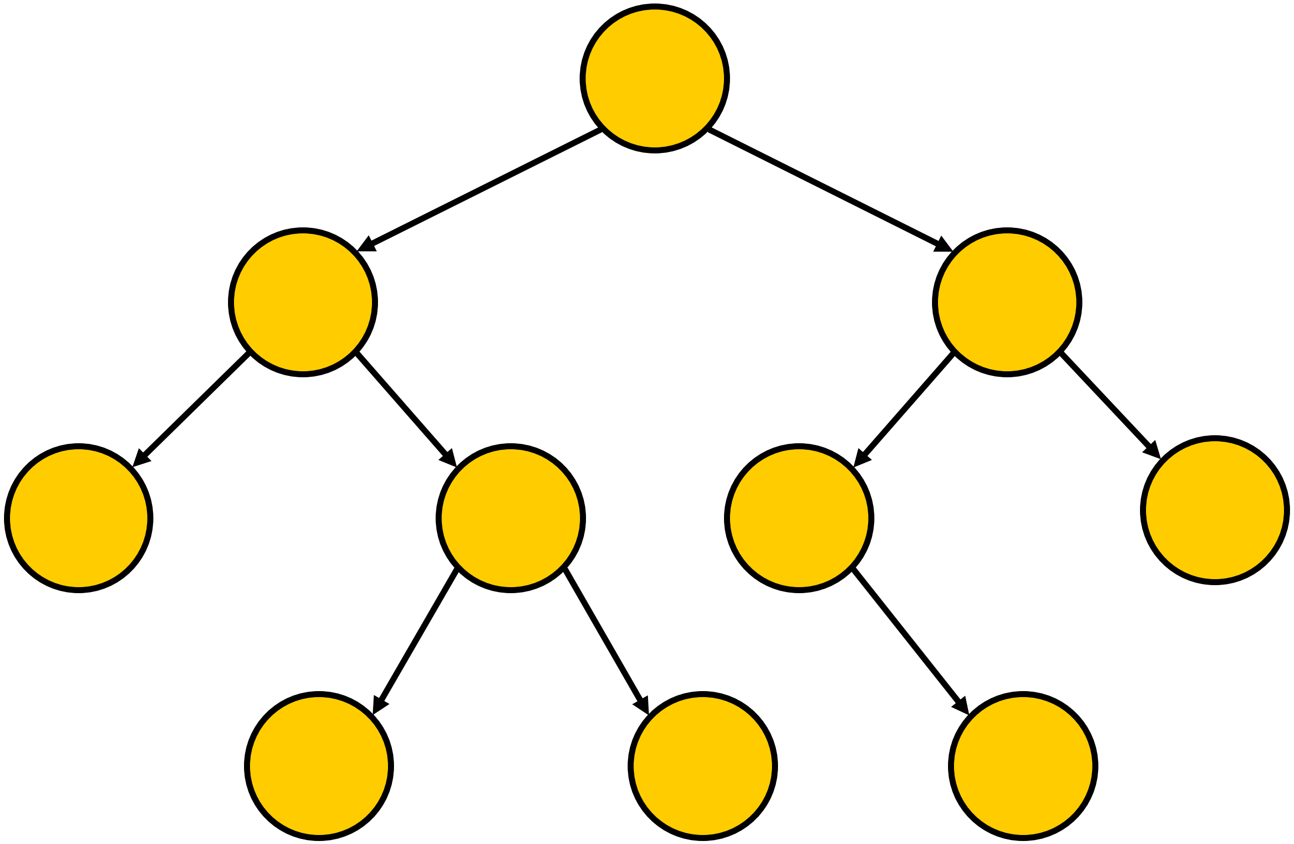
Q: Where are the mothers?

Binary Trees

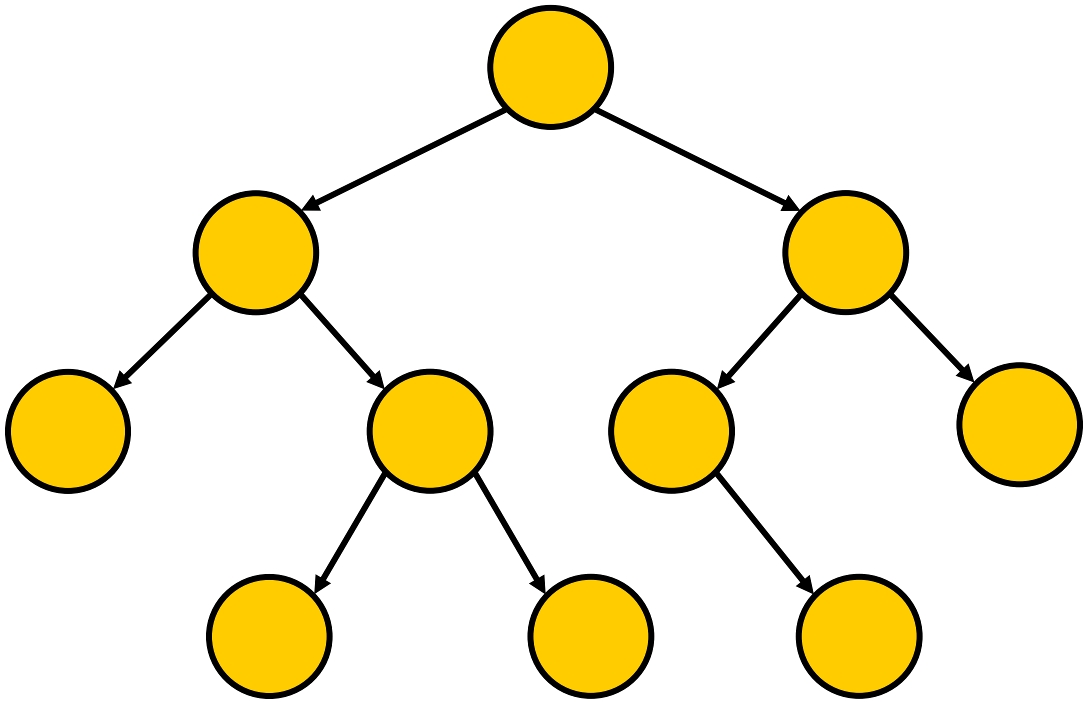
Each node has **at most 2 ordered** children

Q: What is the meaning of "ordered children"?

Binary Tree – each node has at most 2 ordered children.



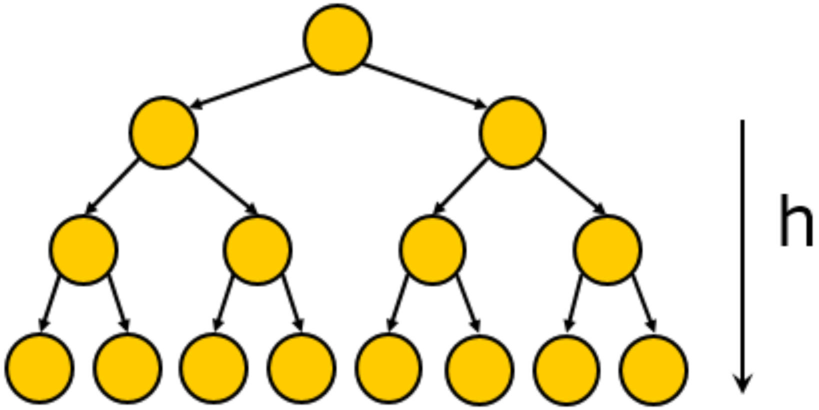
Binary Tree is Recursive



Q: What is the meaning of "recursive" here?

Full Binary Tree

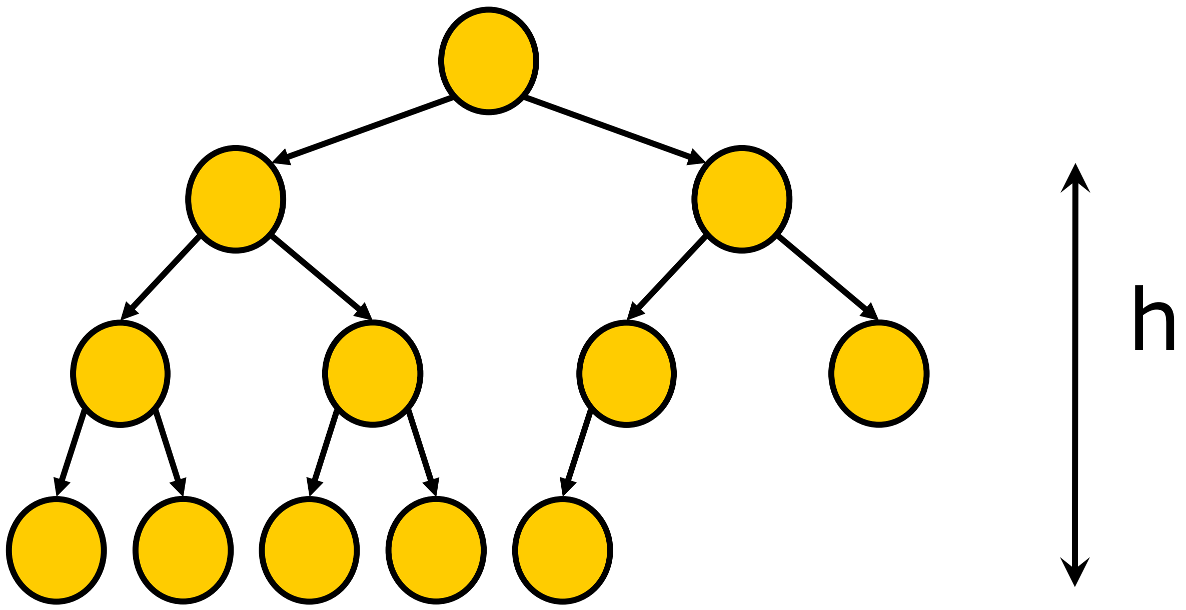
- All nodes at a level < h have two children, where h is the height of the tree.



Question: Is this definition the same as “all nodes except the leaf nodes have 2 children”?
Ans: **No**! **Why**? All leaf nodes may not be of the same level.

Complete Binary Tree

- Full down to level h-1, with level h filled in from left to right.

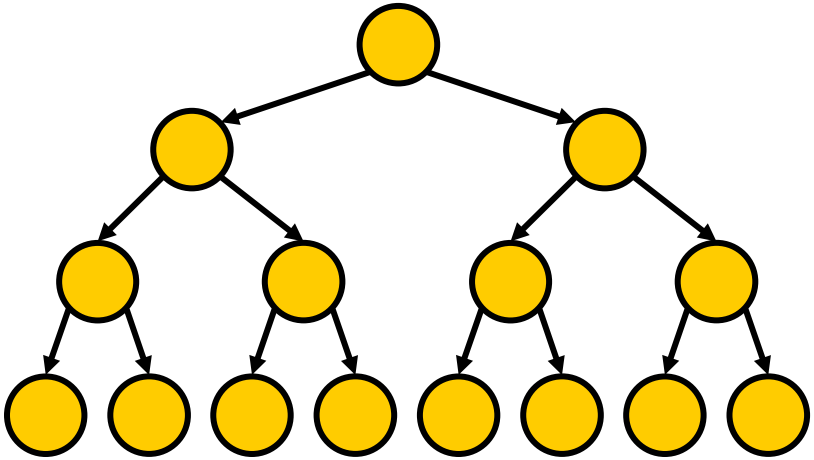


Property

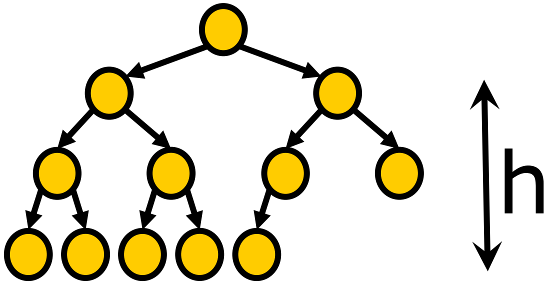
Question: How many nodes in a full binary tree of height h?

Ans: Number of nodes is $\mathbf{2^{h+1} - 1}$. Therefore the height of a full binary tree with N nodes is log N.

Q: How do you prove these 2 results?



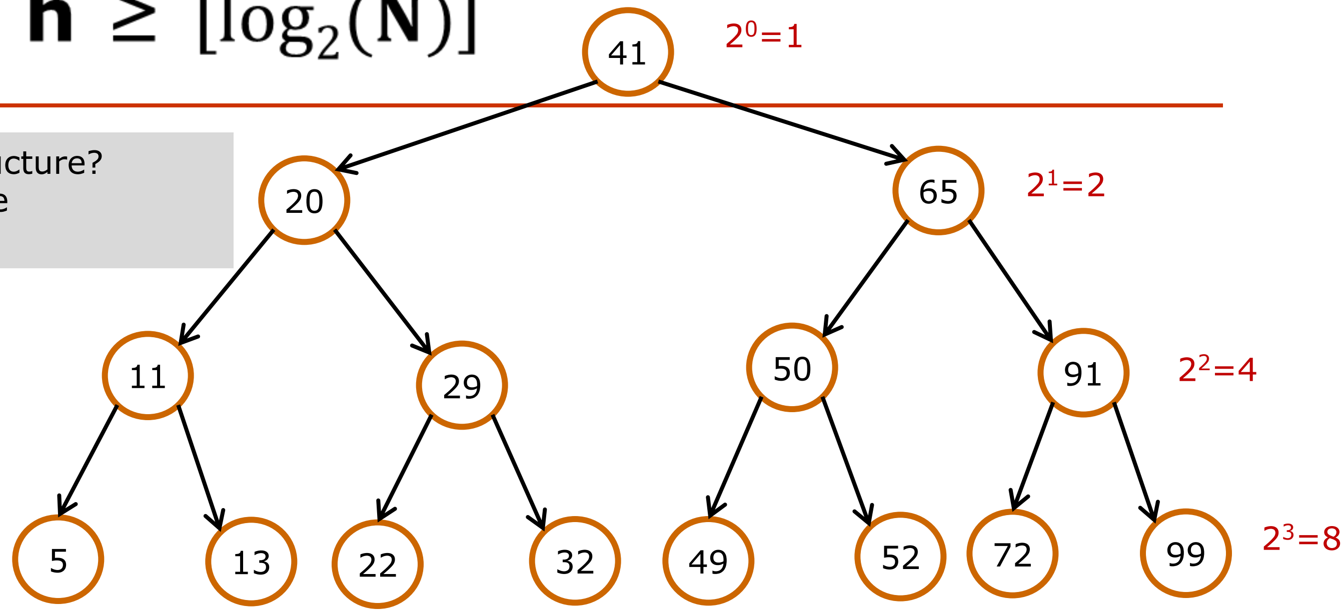
Q: What are the **maximum** and **minimum** numbers of nodes in a complete binary tree of height h?



Most operations take O(**h**) time

Lower bound: **h** $\geq \lfloor \log_2(\mathbf{N}) \rfloor$

Remember this tree structure?
Perfect Binary Tree



$$N = 1 + 2 + 4 + \ldots + 2^h = 2^0 + 2^1 + 2^2 + \ldots + 2^h$$

$$= 2^{h+1} - 1 < 2^{h+1} \text{ (sum of geometric progression)}$$

$$\log_2 (N) < \log_2(2^{h+1})$$

$$\rightarrow \log_2 (N) < (h+1) * \log_2 (2)$$

$$\rightarrow h > \log_2 (N)-1$$

$$\rightarrow h \geq \lfloor \log_2(\mathbf{N}) \rfloor$$

Implementation

A tree can be implemented using **reference based** representation or **array based** representation

Reference Based Implementation

```
class TreeNode
{
   Object item;
   TreeNode left;
   TreeNode right;
   // Methods..
}

class BinaryTree
{
   TreeNode root;
   // Methods
}
```

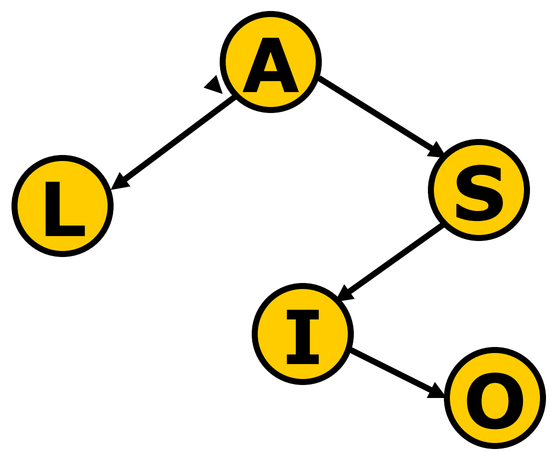
Array Based implementation

```
class TreeNode
{
   Object item;
   int left;
   int right;
   // Methods..
}
```

index	0	1	2	3	4	5
item	L	I	A	S	?	O
left	-1	-1	0	1	-1	-1
right	-1	5	3	-1	-1	-1

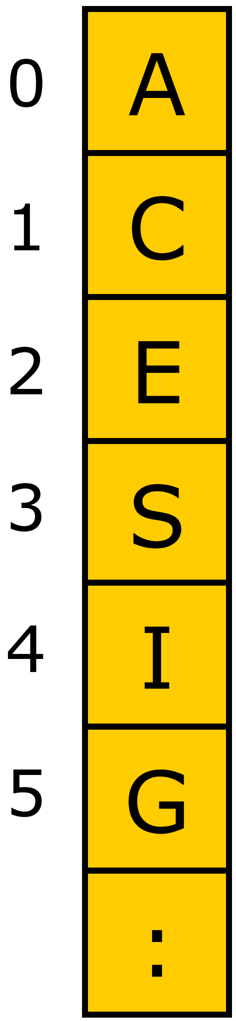
root = 2 **free** = 4

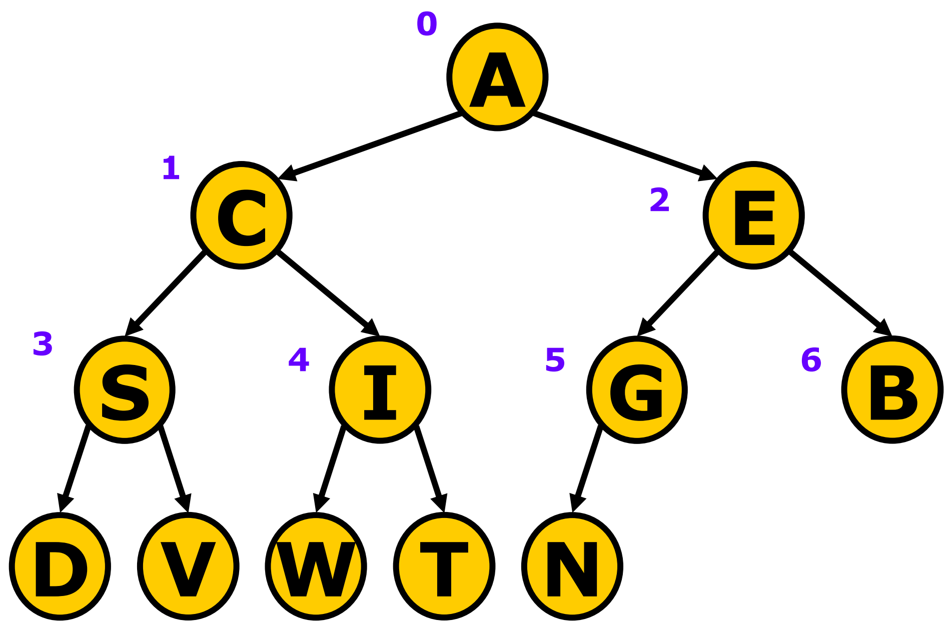
```
class BinaryTree
{
   int root;
   int free;
   TreeNode tree[];
   // Methods
}
```



Q: How to handle free space in an array?

Representing a Complete Tree
- using an array

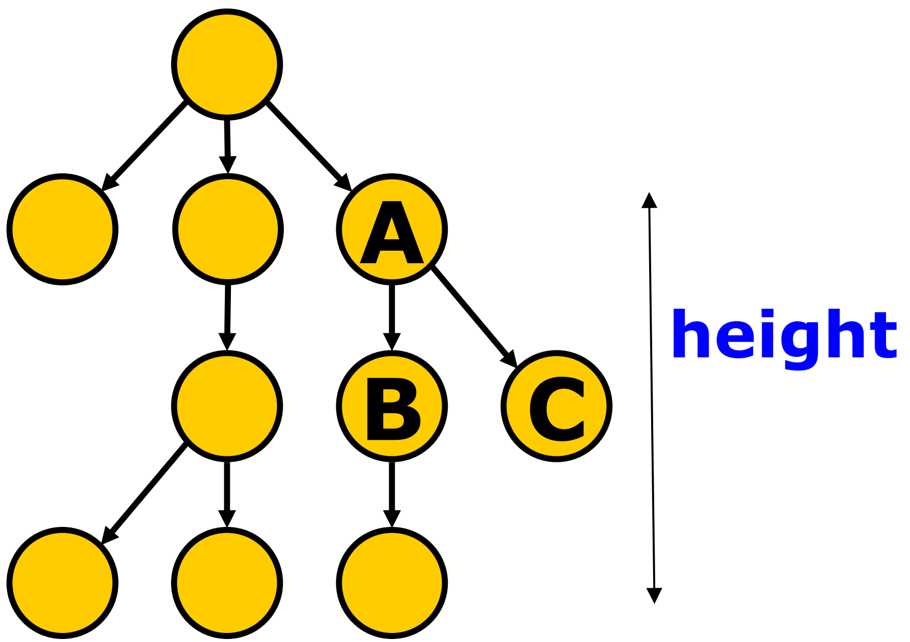




Q: Given that a node is stored in index position i, what are the index positions of its **parent**, **left child**, and **right child**?

Height of a binary tree

- **Maximum level of the nodes in the tree is** called the height of the tree
  - height = 3



Height of a binary tree (cont.)

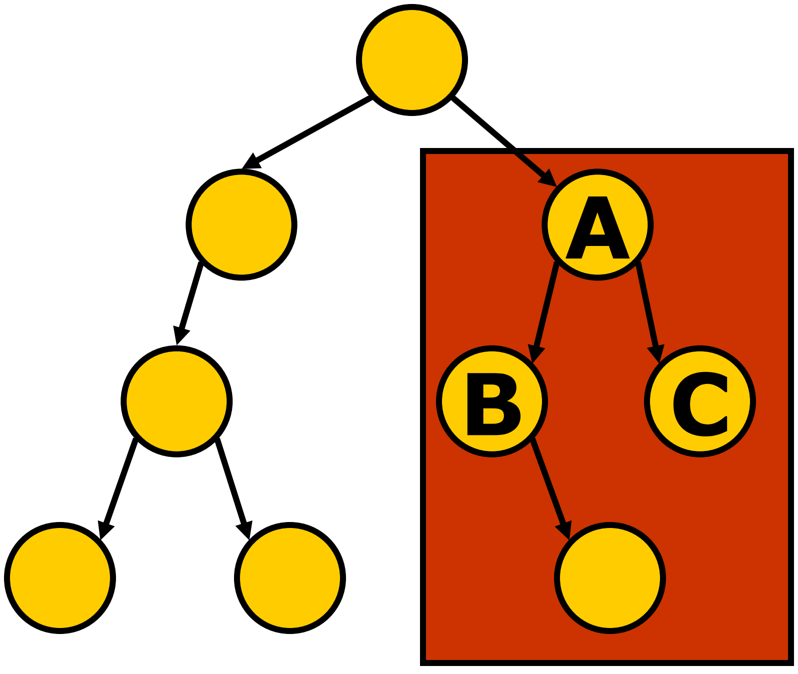
```
height(T)
  if T is empty
    return -1
  else
    return 1 + max (height(T.left), height(T.right))
```

Where T.left and T.right represent the left and right subtrees of the node T respectively

This is a recursive solution, divide and conquer.

Size of a binary tree

- **Number of nodes in the tree**
  - The size of the subtree rooted at A is 4.



Size of a binary tree (cont.)

```
size(T)
   if T is empty
       return 0
   else
       return  1 + size(T.left) + size(T.right)
```

Binary Tree Traversal

Traversing a Binary Tree

- Post-order traversal
- Pre-order traversal
- In-order traversal
- Level-order Traversal

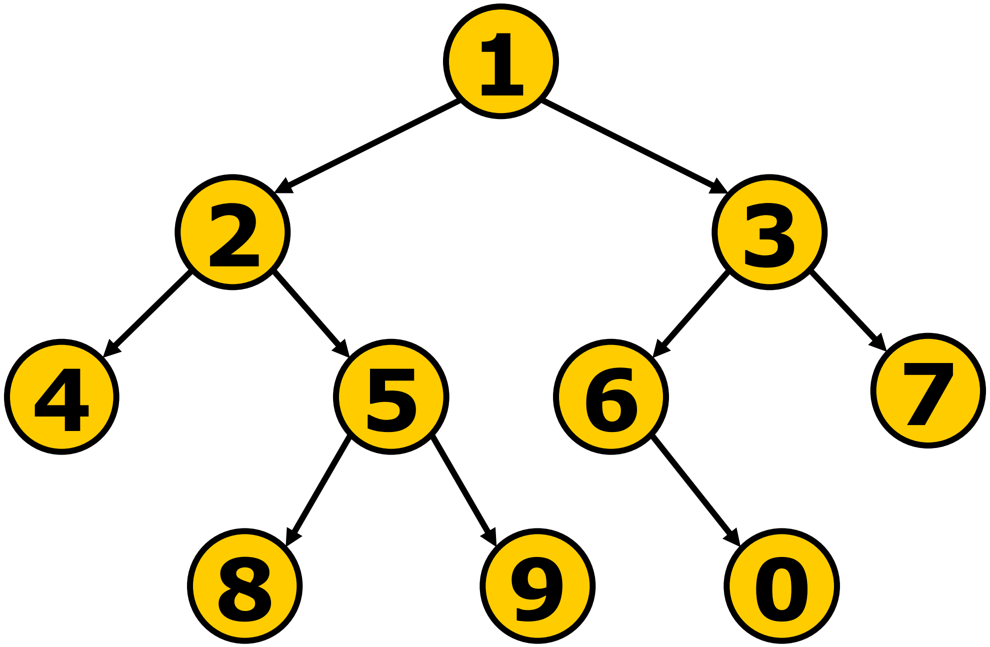
Post-order Traversal

Traverse the **root after** traversing the left and right subtrees.

```
postorder(T)
  if T is not empty then
     postorder(T.left)
     postorder(T.right)
     print T.item
```

Note: This is a recursive solution. Can you give an iterative solution?

Traversal Example



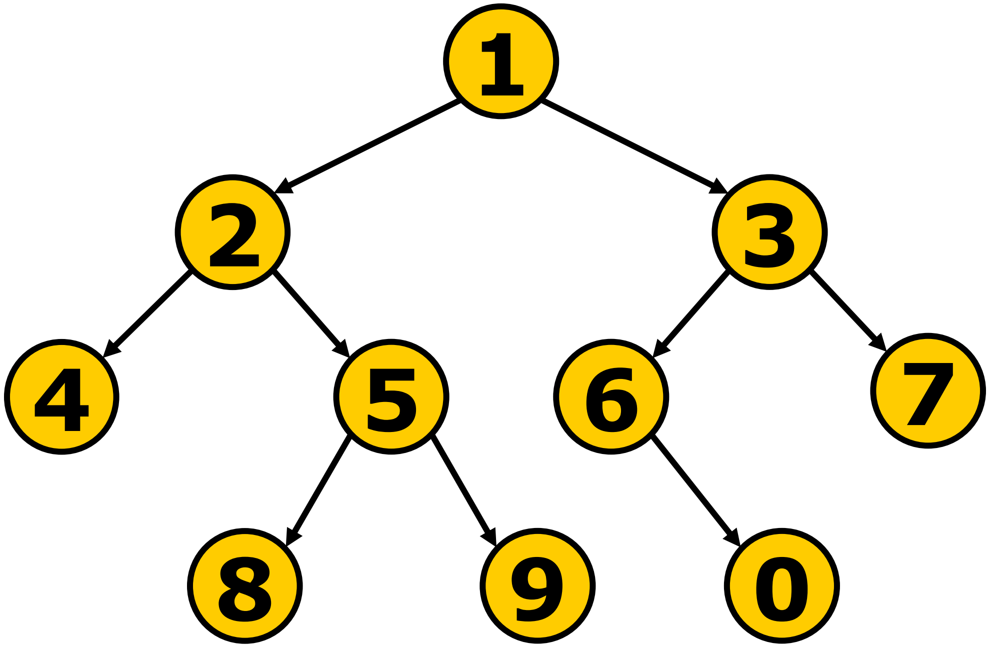
Post-order: 4 8 9 5 2 0 6 7 3 1

Pre-order traversal

Traverse the **root before** traversing the left and right subtrees.

```
preorder(T)
  if T is not empty then
     print T.item
     preorder(T.left)
     preorder(T.right)
```

Traversal Example



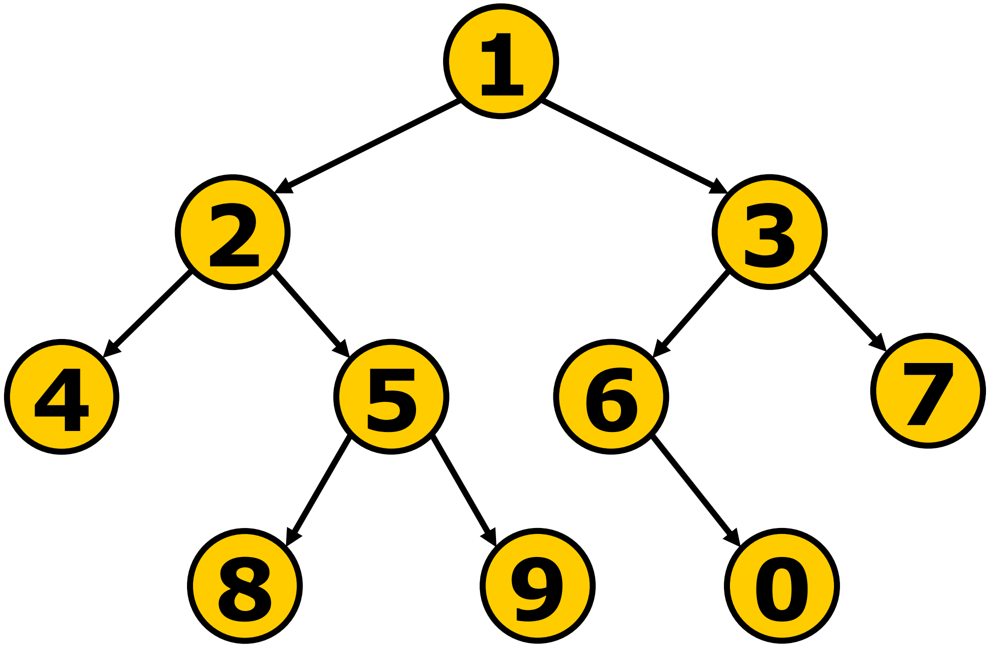
Pre-order: 1 2 4 5 8 9 3 6 0 7

In-order Traversal

Traverse the **root in between** the traversals of left and right subtrees.

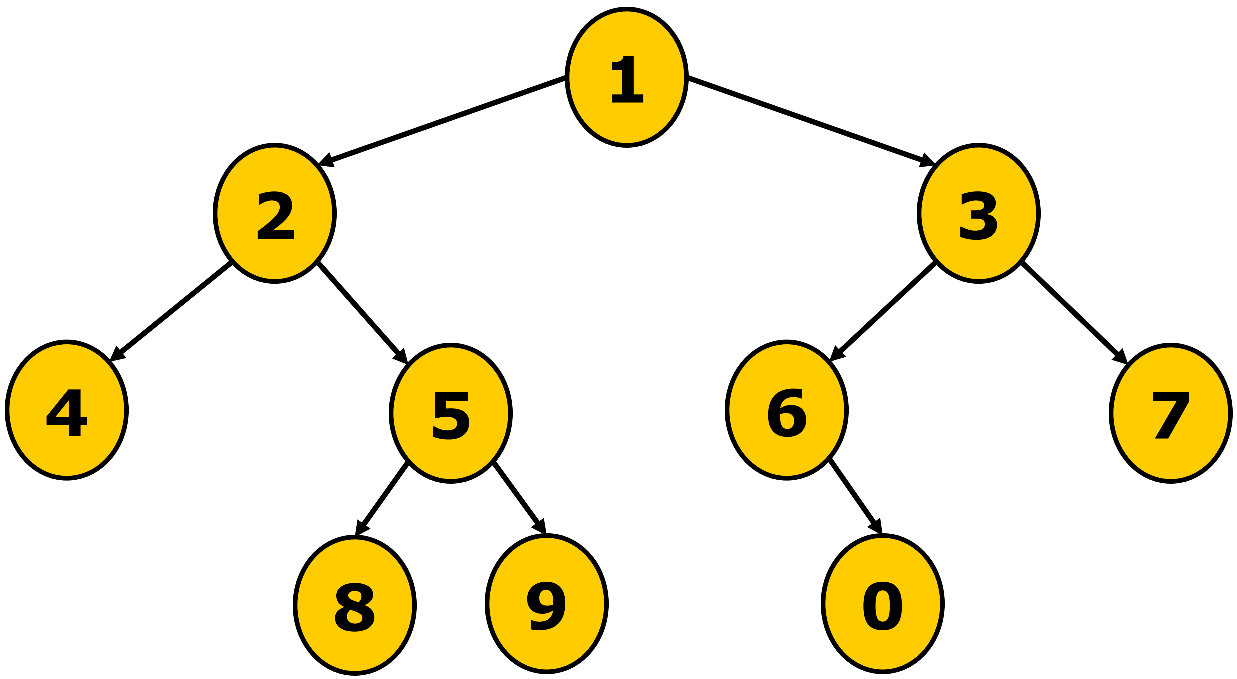
```
inorder(T)
   if T is not empty then
      inorder(T.left)
      print T.item
      inorder(T.right)
```

Traversal Example



In-order: 4 2 8 5 9 1 6 0 3 7

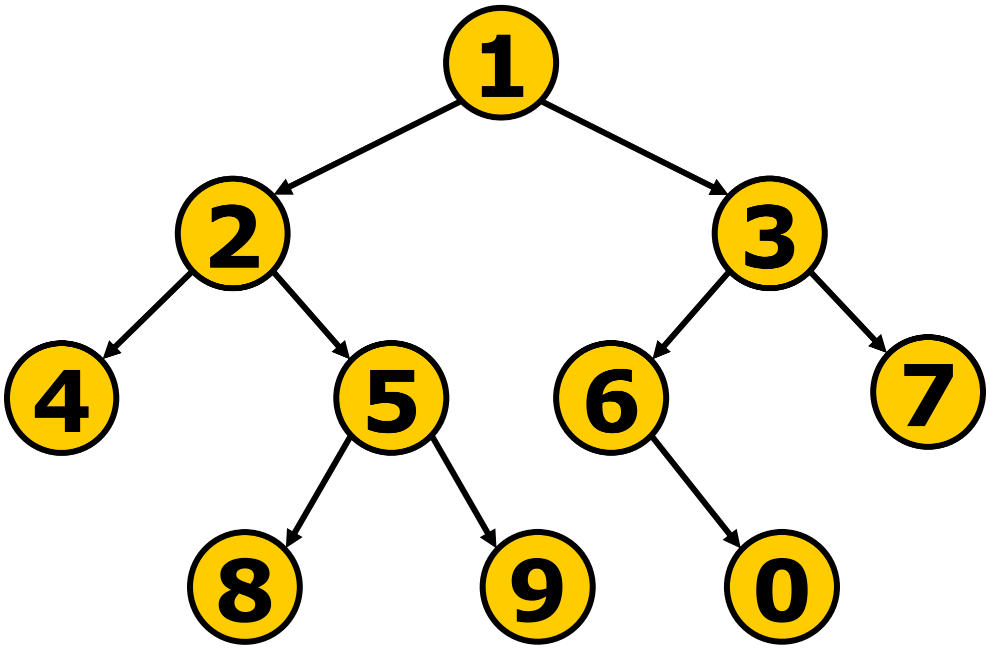
Traversal Example



In-order: 4 2 8 5 9 1 6 0 3 7

Level-order Traversal

Traverse the **tree** **level by level** and from **left to right**.

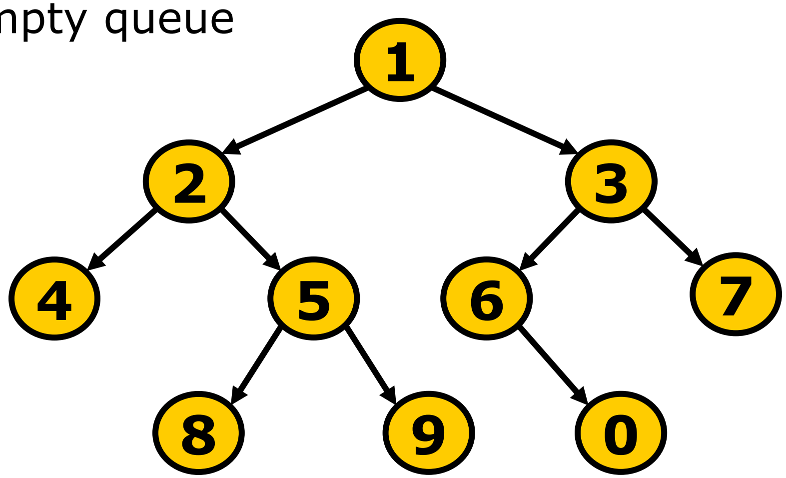


Level-order: 1 2 3 4 5 6 7 8 9 0

Iterative solution

levelOrder(T) - using a queue

```
if T is empty return
Q = new Queue //create an empty queue
Q.enqueue(T) //insert T into Q

while Q is not empty
   curr = Q.dequeue()
   print curr.item
   if curr.left is not empty
       Q.enqueue(curr.left)
   if curr.right is not empty
       Q.enqueue(curr.right)
```



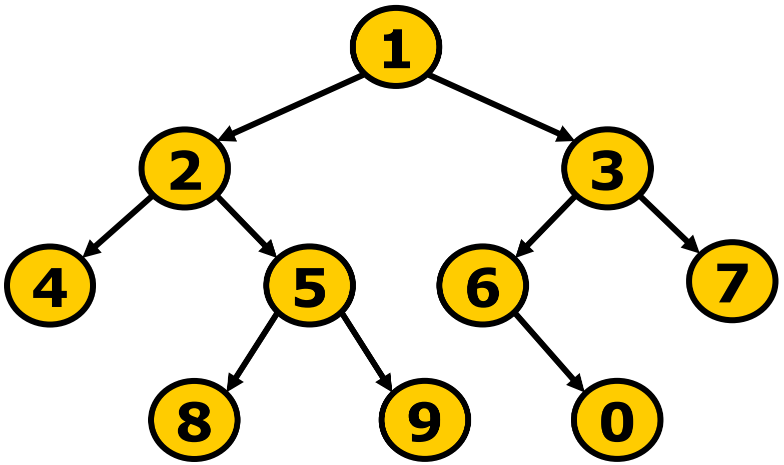
Q: Why do we use a **queue** instead of a **stack**?

levelOrder(T) - Example using a queue

Queue	curr	print
1		
empty	*1*	*1*
2,3		
3	*2*	*2*
3,4,5		
4,5	*3*	*3*
4,5,6,7		
5,6,7	*4*	*4*
5,6,7		
6,7	*5*	*5*
6,7,8,9		
7,8,9	*6*	*6*
7,8,9,0		
8,9,0	*7*	*7*
8,9,0		
9,0	*8*	*8*
9,0		
0	*9*	*9*
0		
empty	*0*	*0*
empty	**end**	

Note: The data in the queue are references to the nodes

dequeue
enqueue, enqueue
dequeue
enqueue, enqueue
...



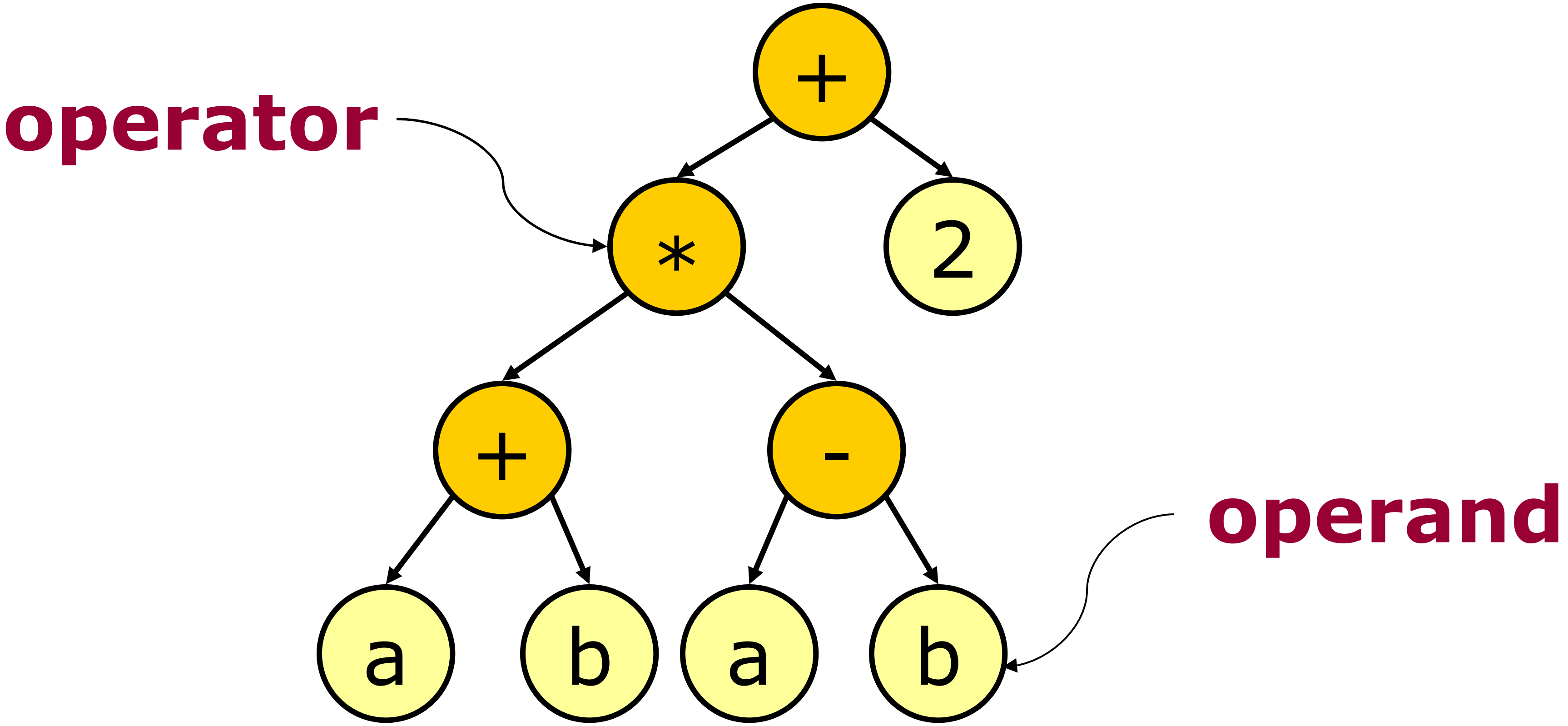
Q: What is the maximum no of nodes in the queue?

Q: What is the main implementation problem of level-order traversal?

Ans: Size of the queue!

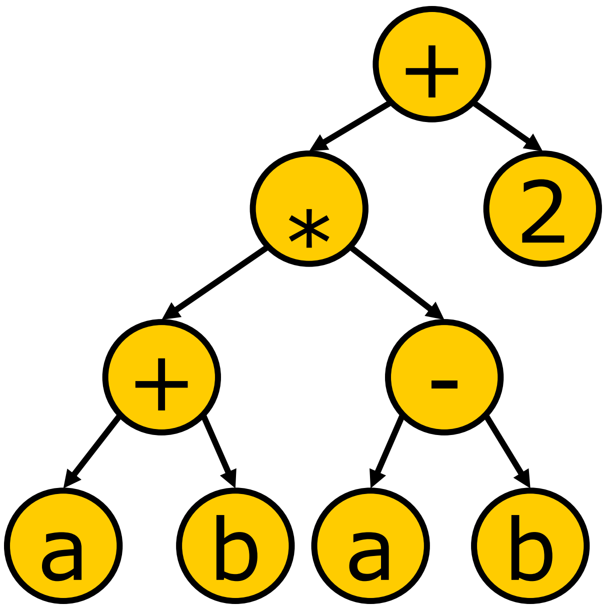
Expression Trees

Evaluating Expression Tree



Leaf nodes (or leaves) store operands.
Internal nodes and root store operators

Traversing Expression Tree



Q2: What is the arithmetic expression of this expression tree?

Post-order traversal: a b + a b - * 2 +

Note: This is the **post**fix expression of the expression tree.

Q1: What are the **infix** and **prefix** expressions of this tree?

Evaluation of Expression Tree

```
eval(T)
  if T is empty
     return 0
  if T is a leaf
    return value of T
  else if T.item is "+"
          return eval(T.left) + eval(T.right)
        else if T.item is "*"
               return eval(T.left) * eval(T.right)
```

Q1: How to handle other arithmetic operators such as **/**, **-**, @, ^ and **unary -** ?

Q2: Do we need to consider the **priorities** of the operators in expression trees ?

Binary Search Tree (BST)

Definition

BST organizes data in a binary tree such that:

- all keys **smaller** than the root are stored in the **left** subtree, and
- all keys **larger** than the root are stored in the **right** subtree.

Q: Can we have nodes with same key values in a BST?

Recall

Tables

- Phone books
- Street directories
- Dictionaries
- Class schedule
- ...

Key	**Data**
Alice	3849-3843
Carl	9493-9349
John	8934-3784

Recall

ADT Table operations

ADT table provides operations to maintain a set of data, each can be uniquely identified by a **key**.
Examples include dictionary, and phonebook.

- data = **search** (key)
- **insert** (key, data)
- **delete** (key)

Recall

Running Times of operations

	Unsorted Array/List	Sorted Array	Sorted LinkedList
Search			
Insert			
Delete			

Recall

Running Times of operations

	Unsorted Array/List	Sorted Array	Sorted LinkedList
Search	O(N)	$O(\log_2 N)$	O(N)
Insert	O(1)	O(N)	O(N)
Delete	O(N)	O(N)	O(N)

Q1: Are unsorted Array/List implementations better than sorted Array?

Q2: Are unsorted Array/List implementations better than sorted LinkedList?

Q3: Can we use a stack or an queue to implement table ADT? Why?

Binary Search Tree (BST)

- insert, delete, and search can be done in

$$O(H)$$

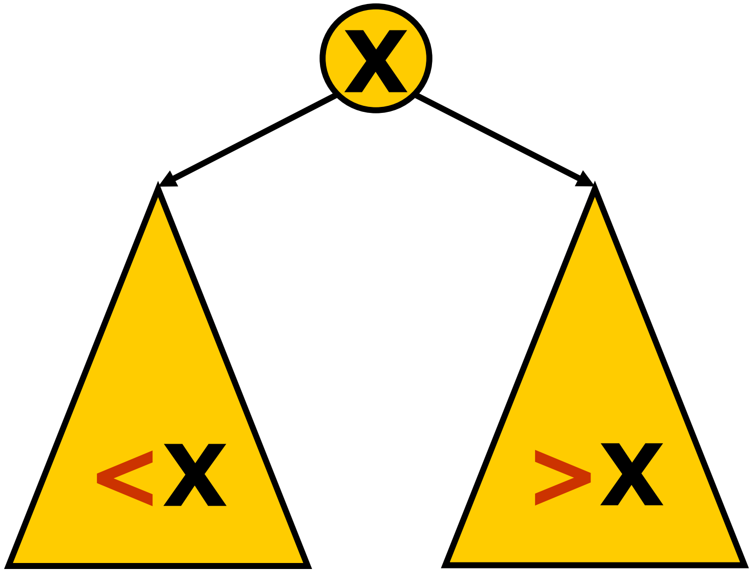
where **H** is the height of the BST**.**

Q1: What is H with respective to N, the no of nodes?

Q2: Are the performances of update operations of BST better than unsorted and sorted array?

Q3: What are the worst cases?

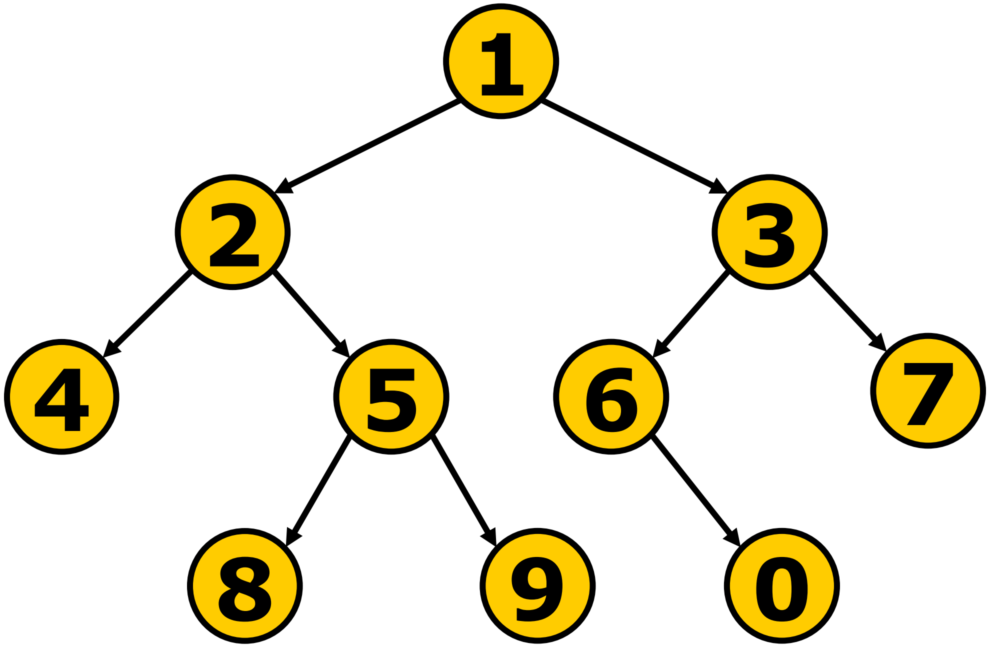
BST Property



BST organizes data in a binary tree such that:

all keys **smaller** than the root are stored in the **left** subtree, and
all keys **larger** than the root are stored in the **right** subtree.

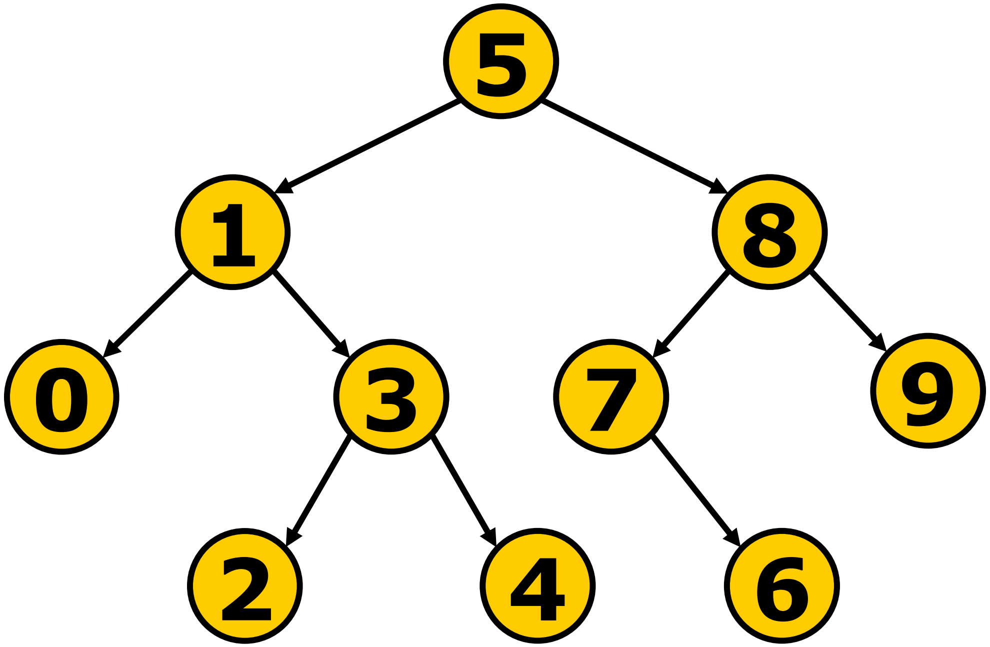
Example



NOT a BST

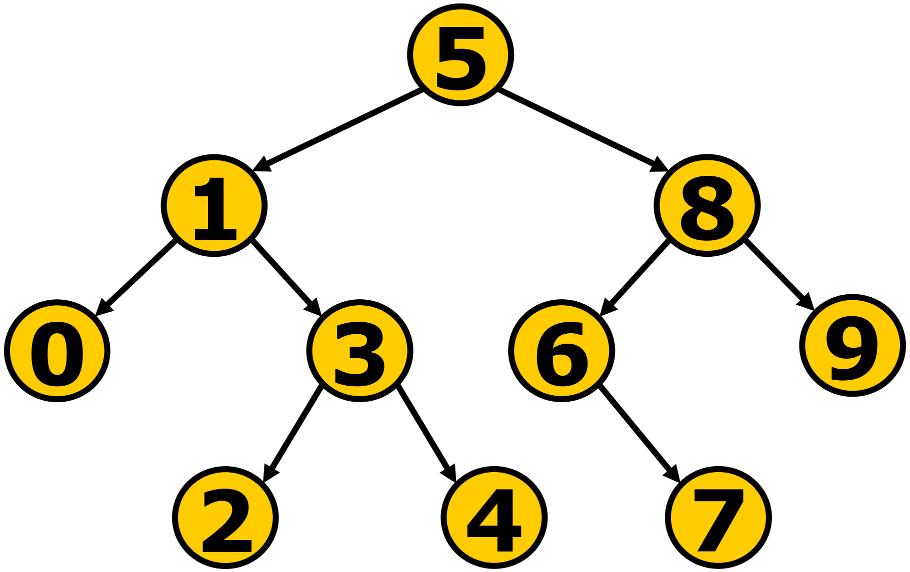
Q: Why?

Example



A BST ? **NO.** Why?

Example



A BST

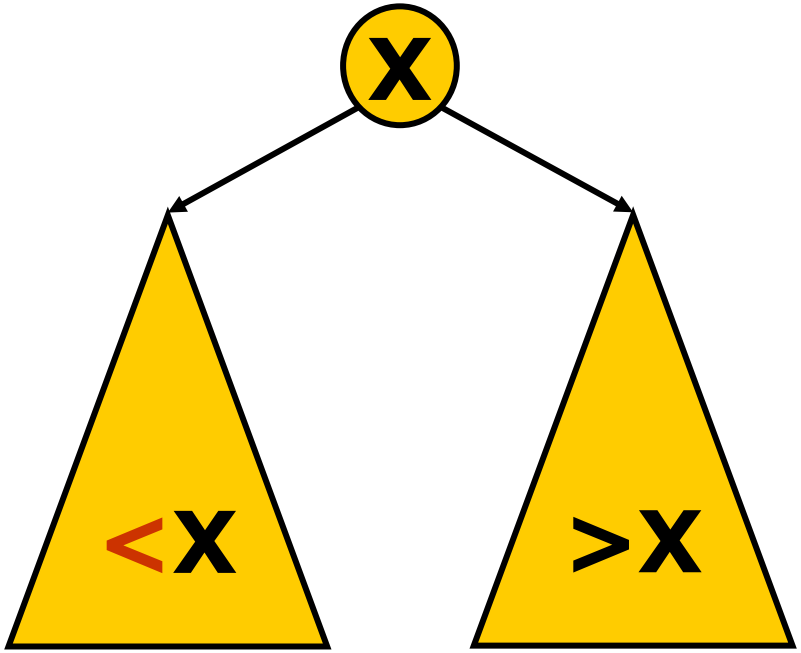
Q: What do you get when you traverse a BST in **in-order**?

Ans: 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 (in **increasing order**).

Why?

Operations on BST

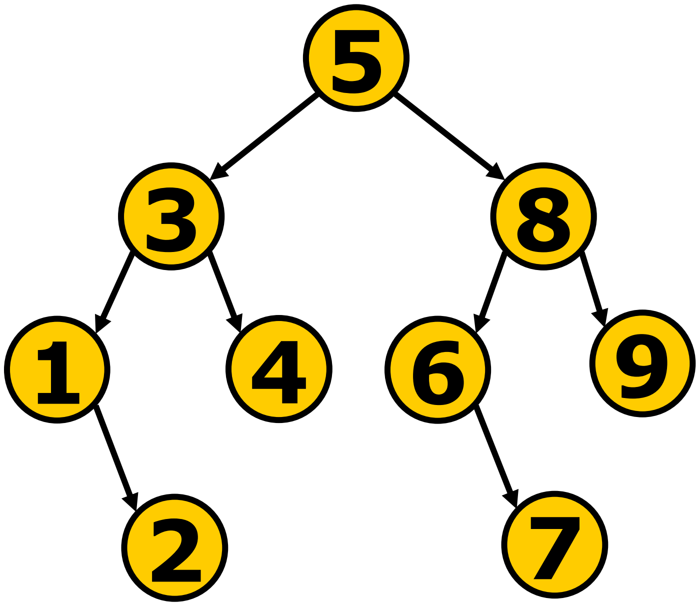
Find Minimum Element in a BST



If X has a left subtree then the minimum element should be in the left subtree, otherwise the minimum element is X.

Finding Minimum Element (cont.)

while T.left is not empty
  T = T.left
return T.item



Q1: How to find **maximum** values?

Q2: How to find **top-k** (or **bottom-k**) values?
e.g. find top-3 values.

Searching x in T (iterative solution)

```
while T is not empty
    if T.item == x then
        return T
    else if T.item > x then
            T = T.left
        else
            T = T.right
return null    // T is empty, so X is not in T
```

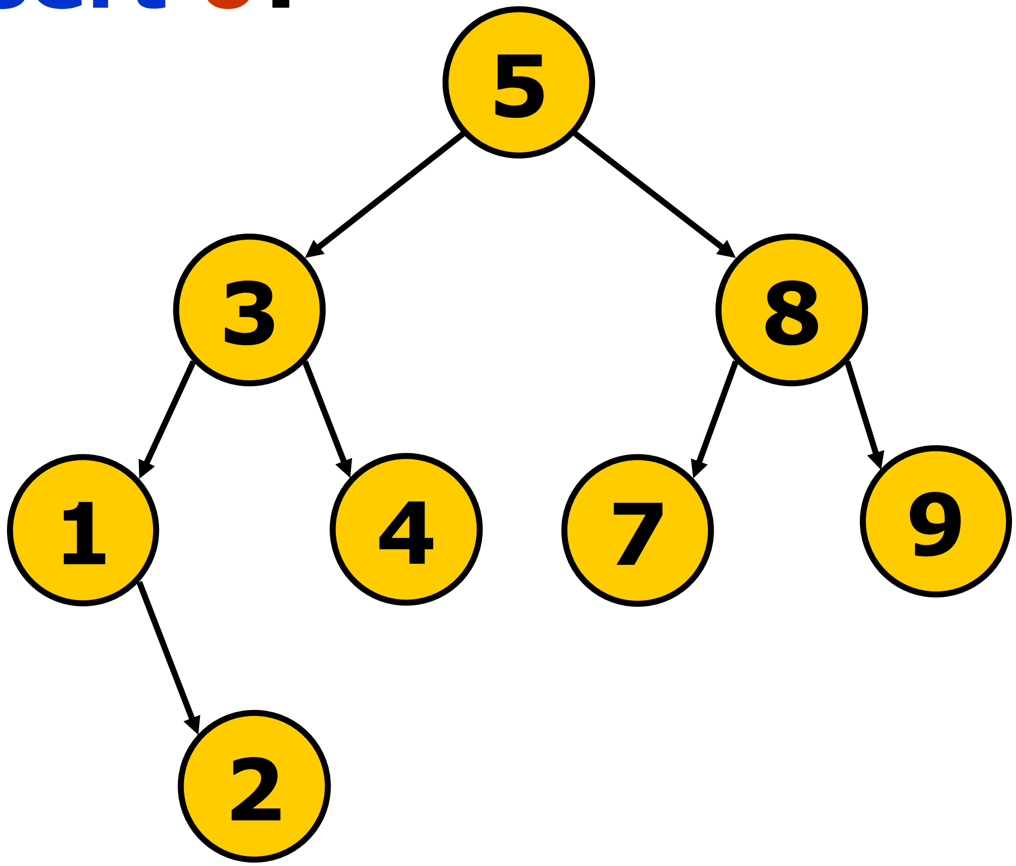
Searching x in T (recursive solution)

```
Search (x, T)
  if T is empty
    return null        // x is not in T
  if x == T.item then
    return T
  else if x < T.item
            return search(x, T.left)
          else
            return search(x, T.right)
```

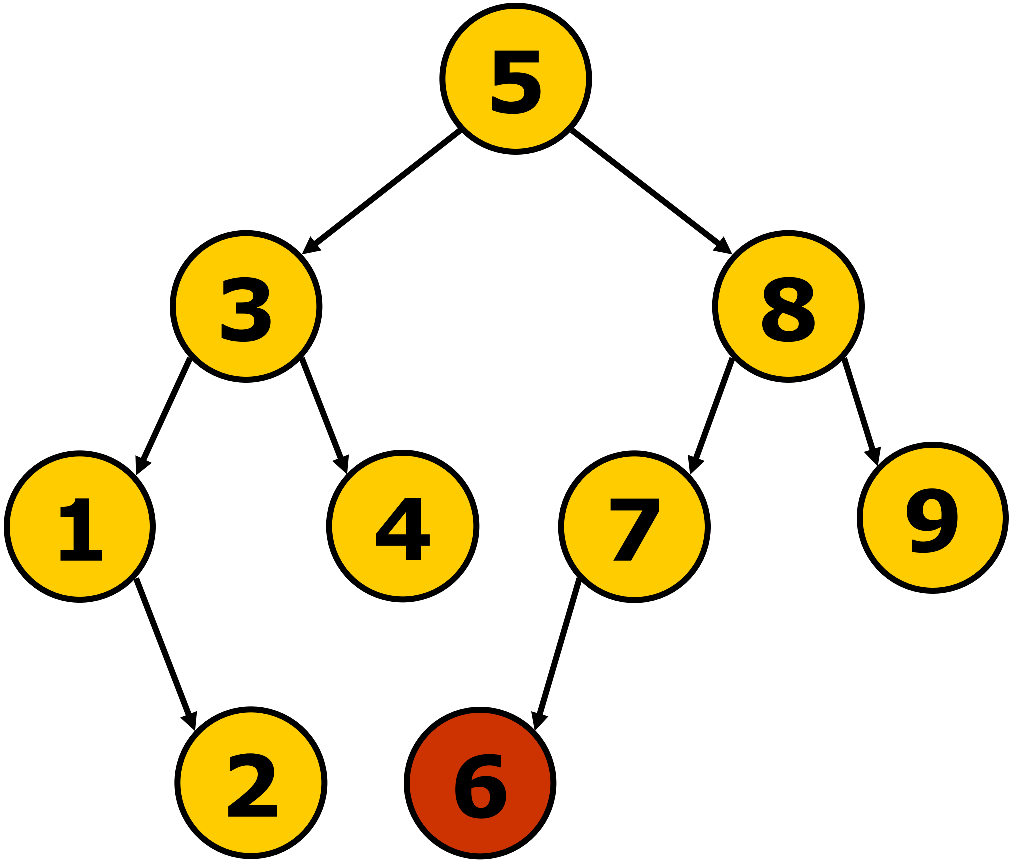
Q: Which solution is faster? Iterative or recursive solution?

Insertions

How to Insert 6?

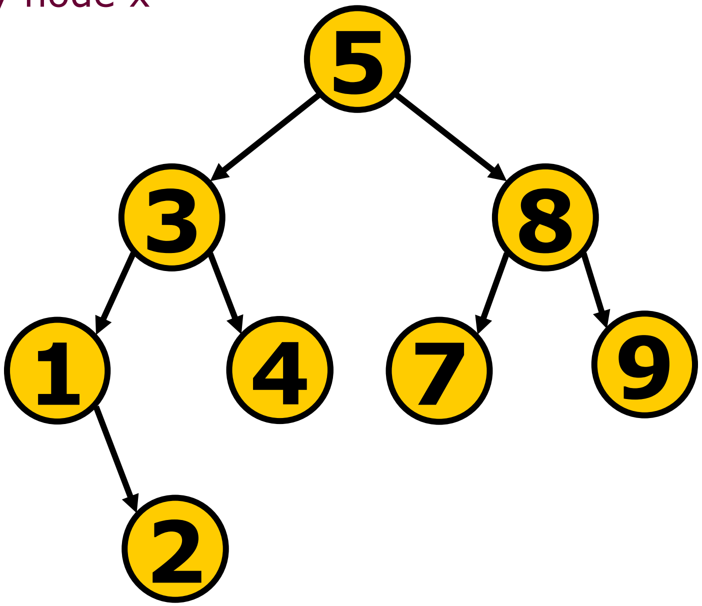


After Inserting 6



insert(x,T)

```
if T is empty
   return new TreeNode(x) // a tree with only node x
else if x < T.item
         T.left = insert(x,T.left)
      else if x > T.item
            T.right = insert(x, T.right)
          else
            return ERROR!
               // X already in T, x=T.item
return T   // return the new tree T
```



This method assumes that we **don't allow duplicate key values** in the BST.

Q: If we allow duplicated key values in the BST, how do you modify the method?
Where to insert it? Before or after the duplicate keys?

Deletions

How to delete?

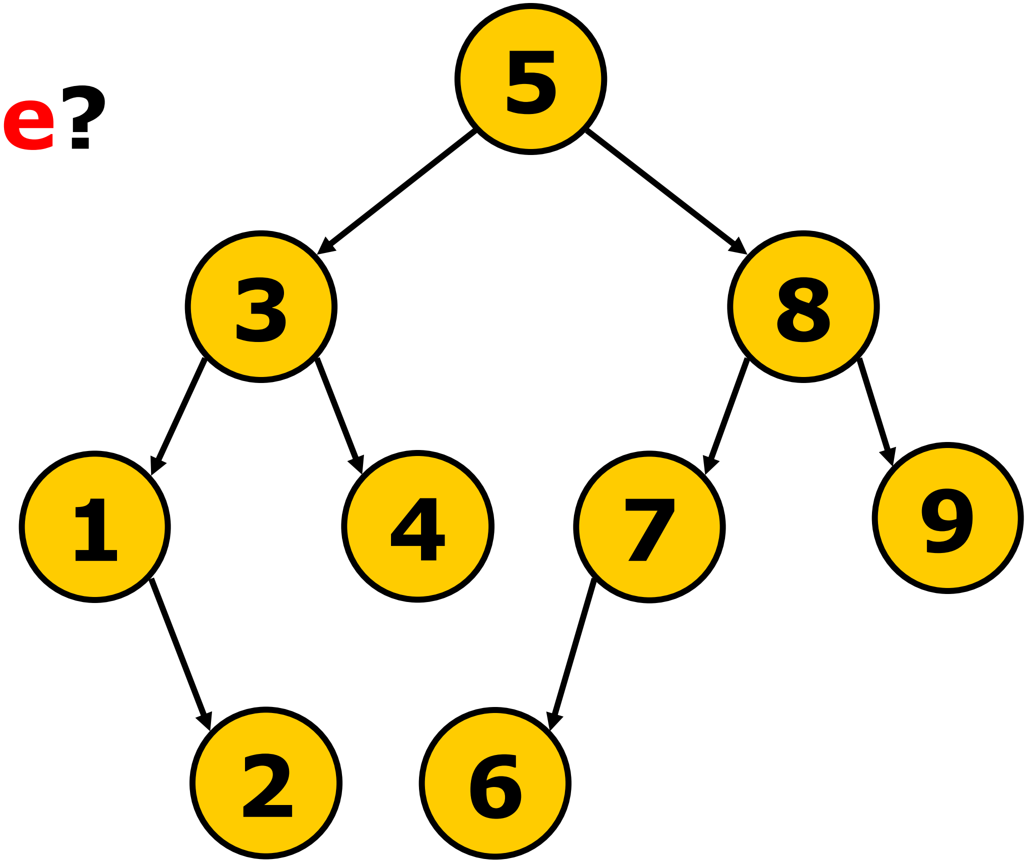


Figure 1. A BST

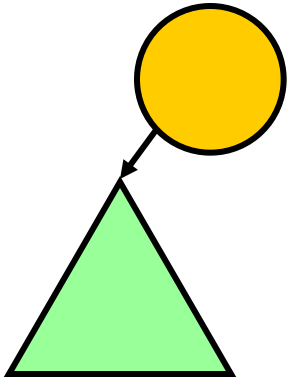
delete(x,T): Case 1

```
if T has no children
    if x == T.item
        return empty tree
    else
        return NOT FOUND
```

E.g: Delete 4 or 3 or 7 in the left figure which only contains a node 4?

delete(x,T): Case 2 (A)

```
if T has only 1 child (left child)
   if x == T.item
         return T.left
   else if x < T.item
                T.left = delete(x,T.left)
           else return NOT fOUND
   return T
```



E.g. delete 4 in the left figure

E.g. delete **10**? **5**?

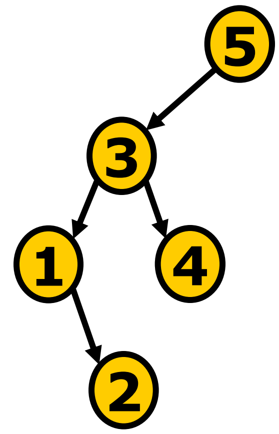
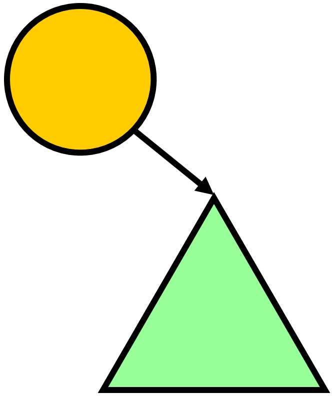


Figure. A BST

delete(x,T): Case 2 (B)

```
if T has only 1 child (right child)
    if x == T.item
        return T.right
    else if x > T.item
            T.right = delete(x, T.right)
        else return NOT FOUND
    return T
```



E.g. delete **6** in the left figure.

E.g. delete **5**? **3**?

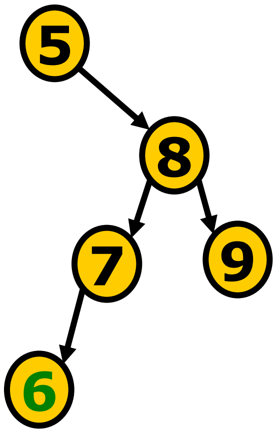
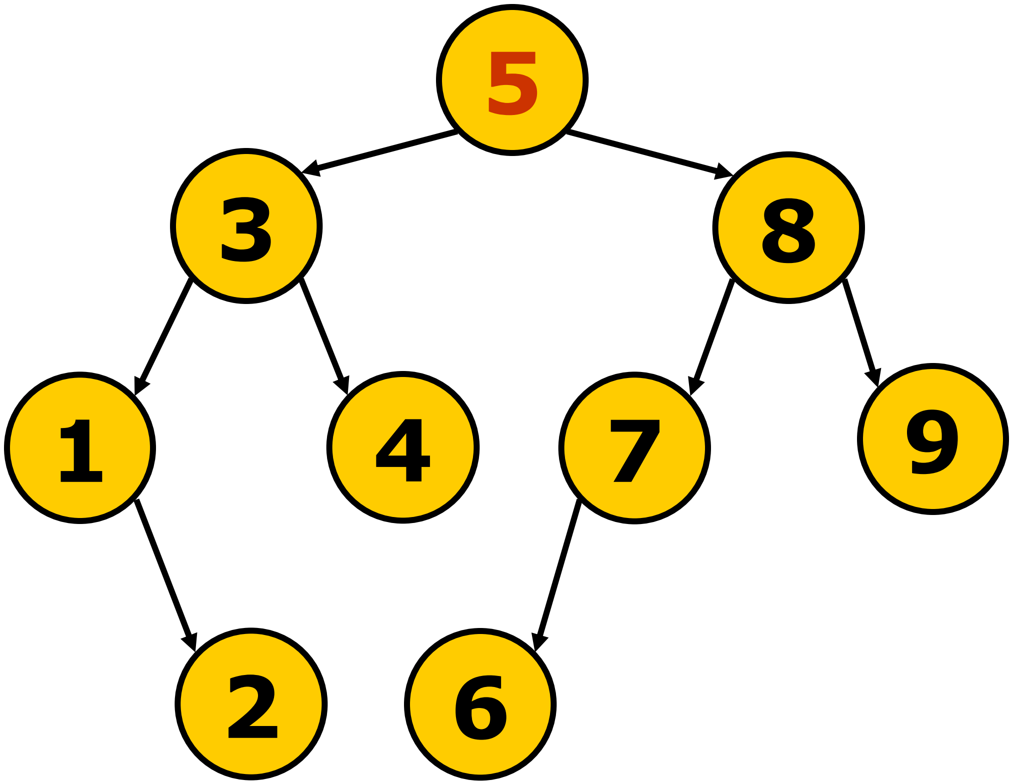


Figure. A BST

delete(x,T): Case 3

Node to be deleted has 2 children

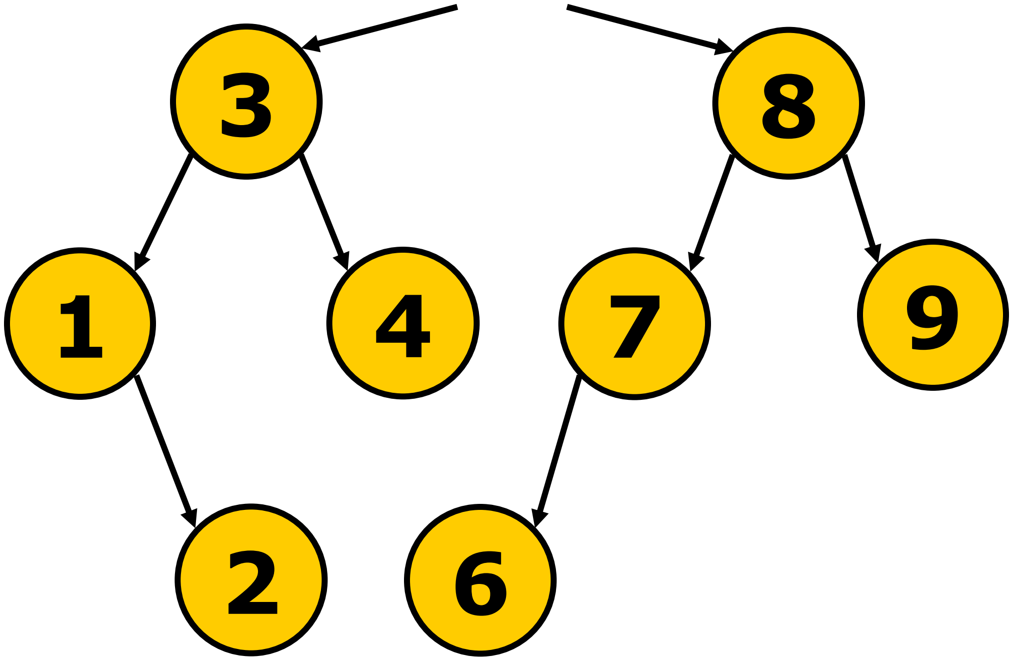
E.g. delete 5



delete(x,T): Case 3

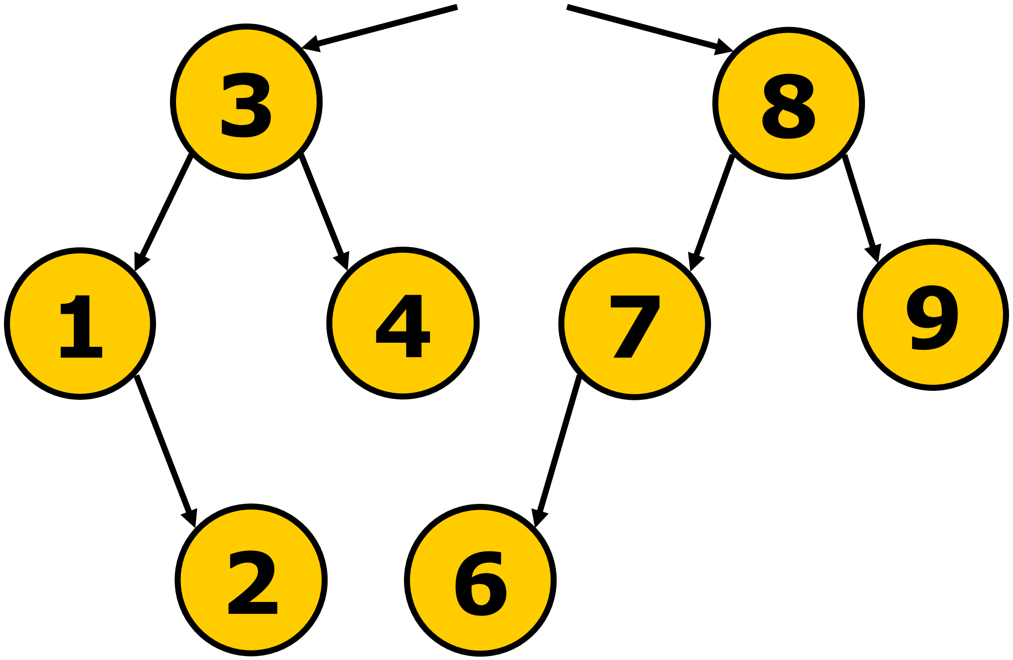
Node to be deleted has 2 children

E.g. delete 5



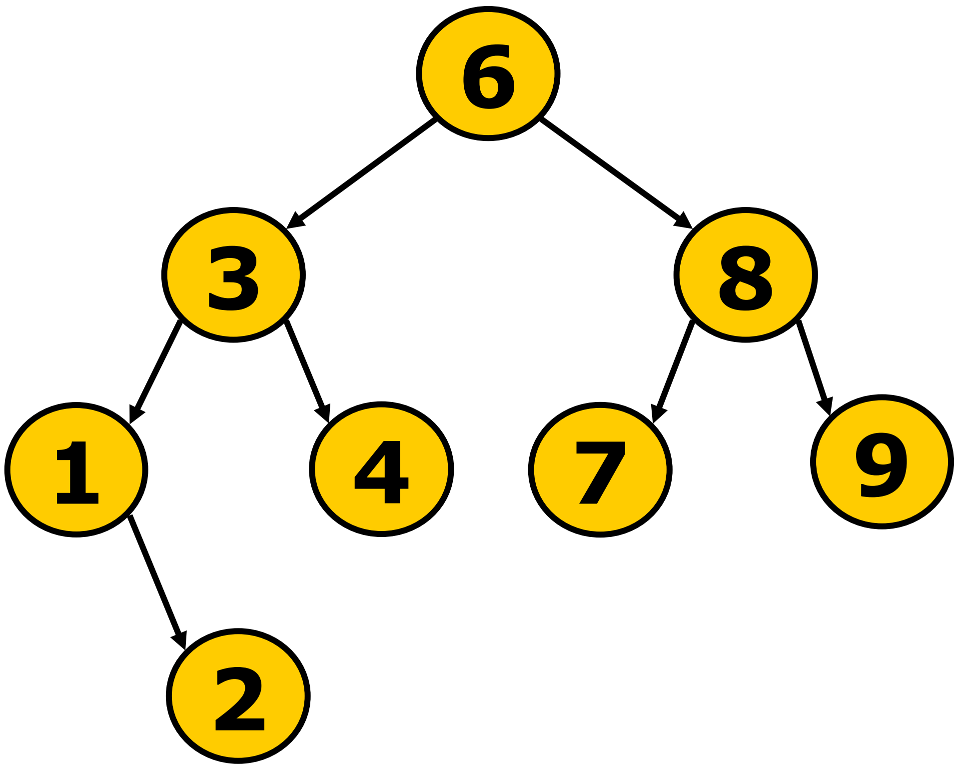
delete(x,T): Case 3

move the smallest node in the right subtree to the position of the deleted node.
Node 5 is deleted



delete(x,T): Case 3

Another way is to move the **largest** node in the left subtree to the position of the deleted node also.



delete(x,T): Case 3

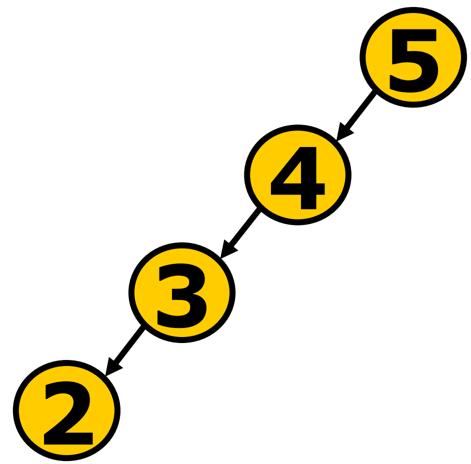
```
if T has two children
   if x == T.item
      T.item = findMin(T.right)
                    // replace T.item by the minimum item of the right subtree

      T.right = delete(T.item, T.right)
                    // delete the original copy of minimum item from the right substree

   else if x < T.item
            T.left = delete(x, T.left)
        else      //  case:  x > T.item
            T.right = delete(x, T.right)
return T
```

Running Time of BST

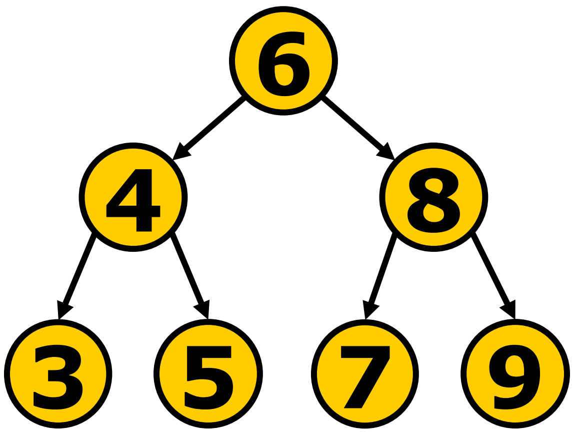
- findMin O(h) where **h** is the height of the BST
- search O(h)
- insert O(h)
- delete O(h)

Running time of BST (cont.)

- But h is not always $O(\log_2 N)$
  where **N** is the total number of nodes in the BST.



Bad ! A **skewed** tree. $h = O(N)$



Good ! A "**balanced**" tree.
$h = O(\log N)$

When you insert nodes in increasing or decreasing order, you get a **skewed** tree and the height h is $O(N)$.