

---

# CS2040C Data Structures and Algorithms

---

## Trees

# Outline

- Binary trees
- Implementation
- Binary Tree Traversal
- Binary Search Trees
- STL sets/maps
- STL search algorithms

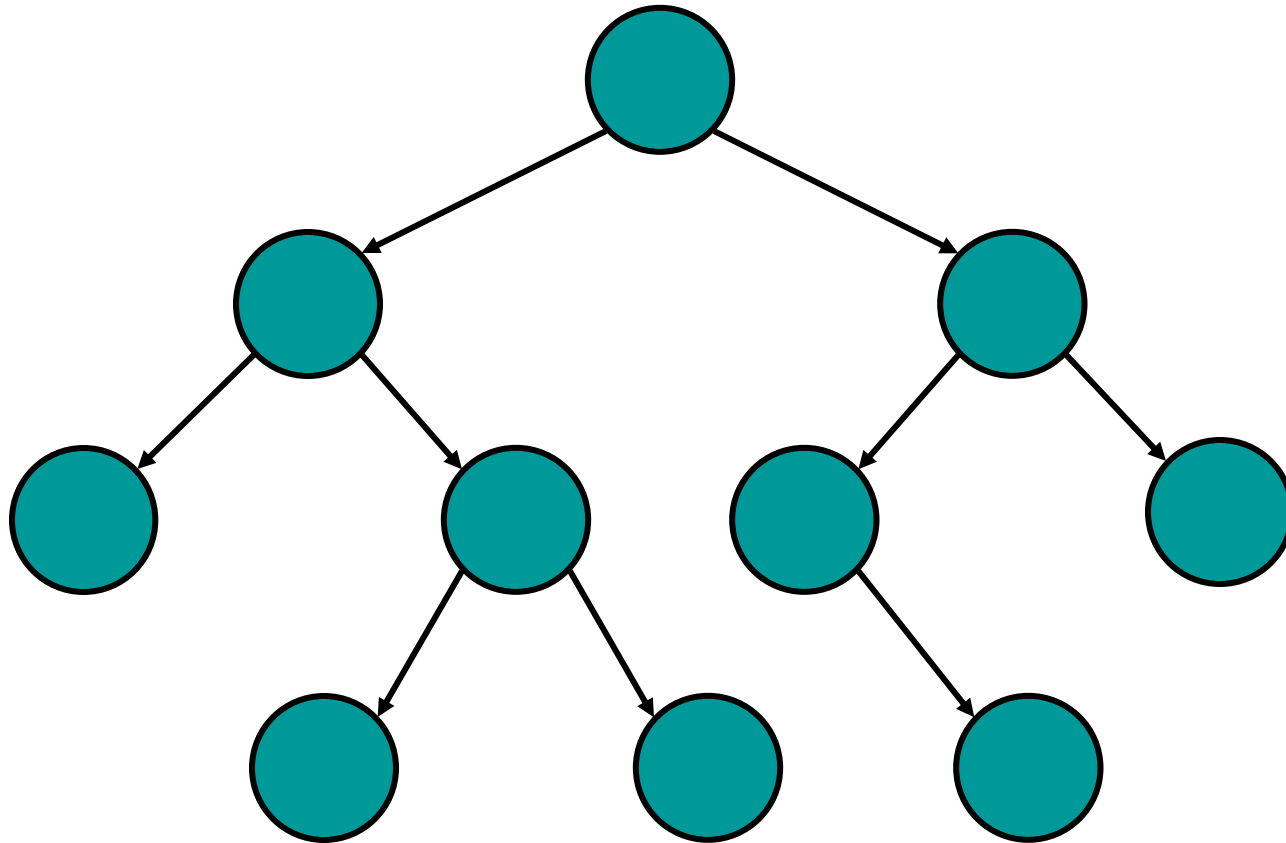
# Binary Trees

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

# Binary Tree

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

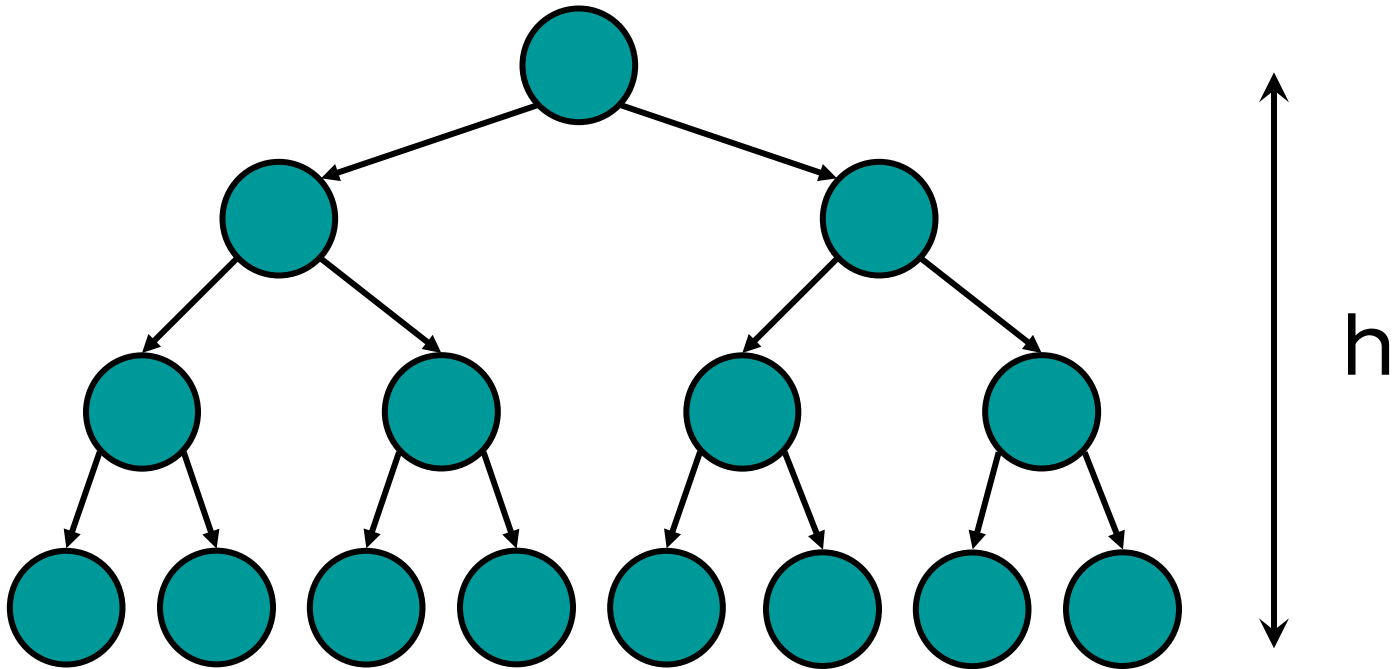
Binary Tree has a recursive structure



**Note:** a degree of a node is the number of subtrees it has

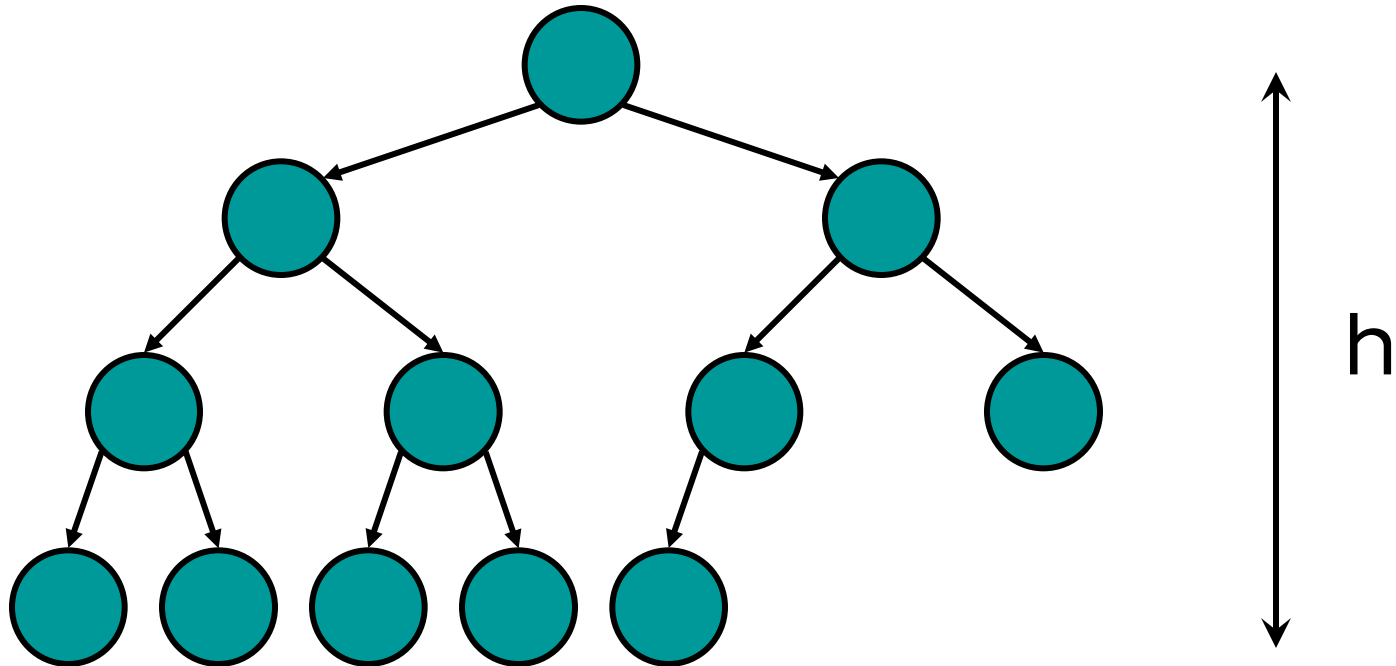
# Full Binary Tree

- All nodes at a level  $< h$  have two children (where  $h$  is the height of the tree)



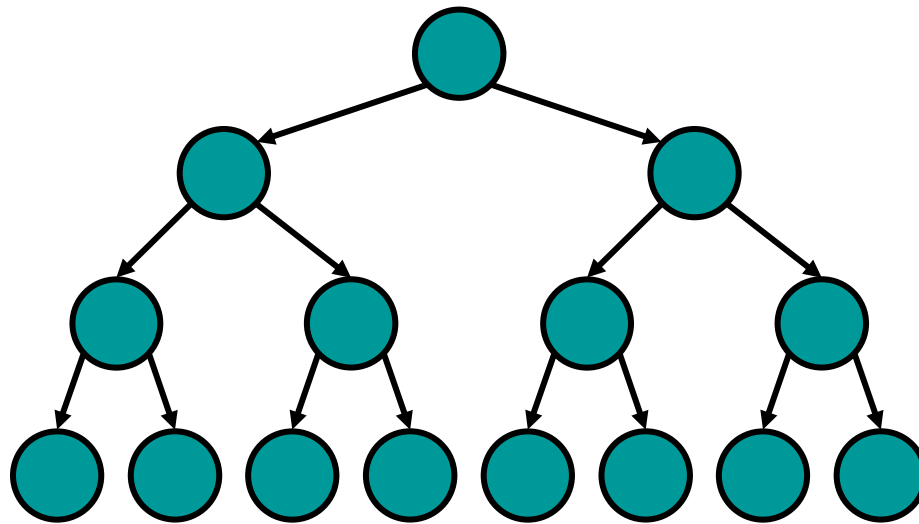
# Complete Binary Tree

- Full down to level  $h-1$
- level  $h$  filled in from left to right



# Full Binary Tree Property

- Number of nodes in a full binary tree of height  $h$  is  $2^h - 1$
- Therefore the height of a full binary tree is  $O(\log N)$



**Q:** How many nodes in a **complete** binary tree of height  $h$ ?

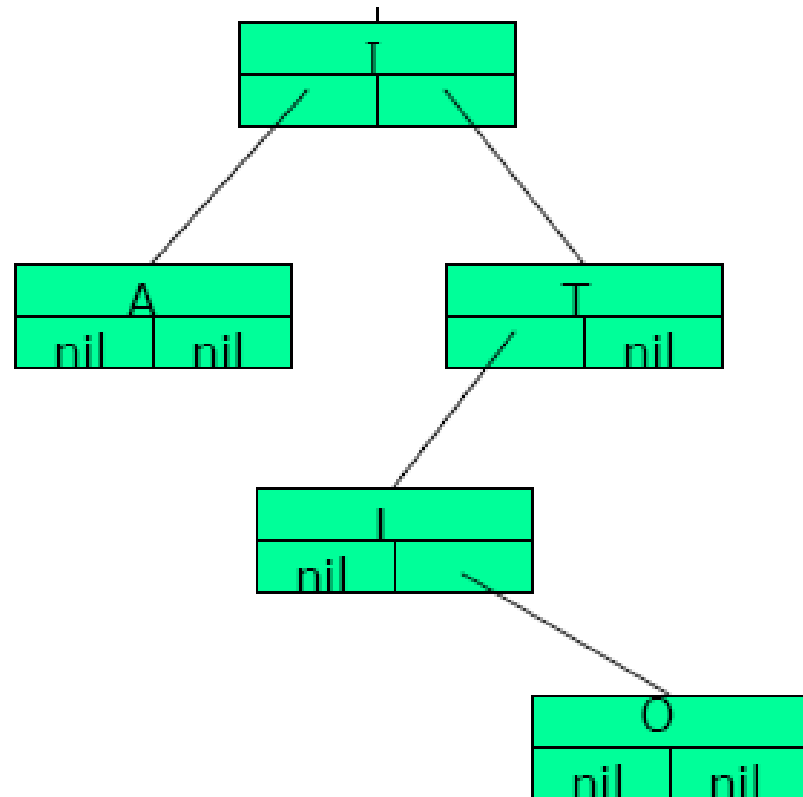
# Implementation

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



# Reference Based

```
class TreeNode {  
private:  
    TreeltemType item;  
    TreeNode *left;  
    TreeNode *right;  
    // More definitions...  
    friend class BinaryTree;  
};  
class BinaryTree {  
private:  
    TreeNode root;  
    // More definitions  
};
```



(nil means “does not point to node”)

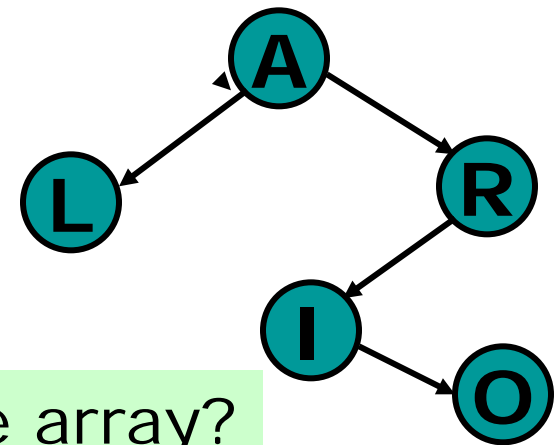
# Array Based

```
class TreeNode {  
private:  
    TreeItemType item;  
    int left;  
    int right;  
    // More definitions...  
    friend class BinaryTree;  
};  
class BinaryTree {  
private:  
    TreeNode[...] tree;  
    int root;  
    int free;  
};
```

tree[0]  
tree[1]

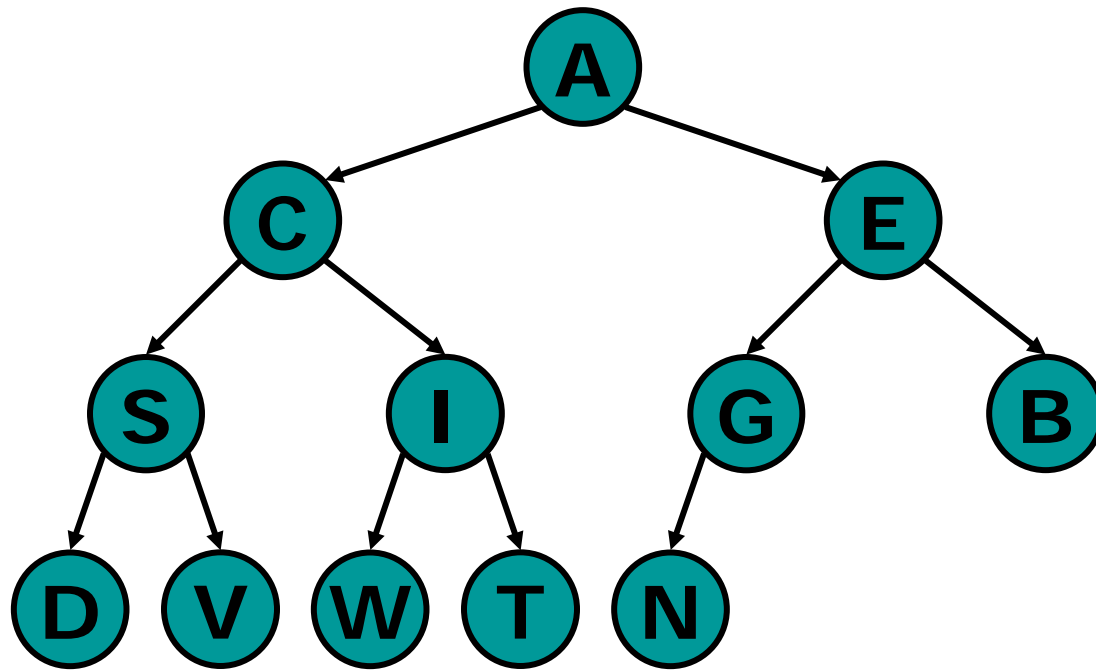
|       |    |    |   |    |    |    |
|-------|----|----|---|----|----|----|
| item  | L  | I  | A | R  | ?  | O  |
| left  | -1 | -1 | 0 | 1  | -1 | -1 |
| right | -1 | 5  | 3 | -1 | -1 | -1 |

root = 2      free = 4



**Q:** How to handle free space in the array?

# Representing a Complete Tree (using array)

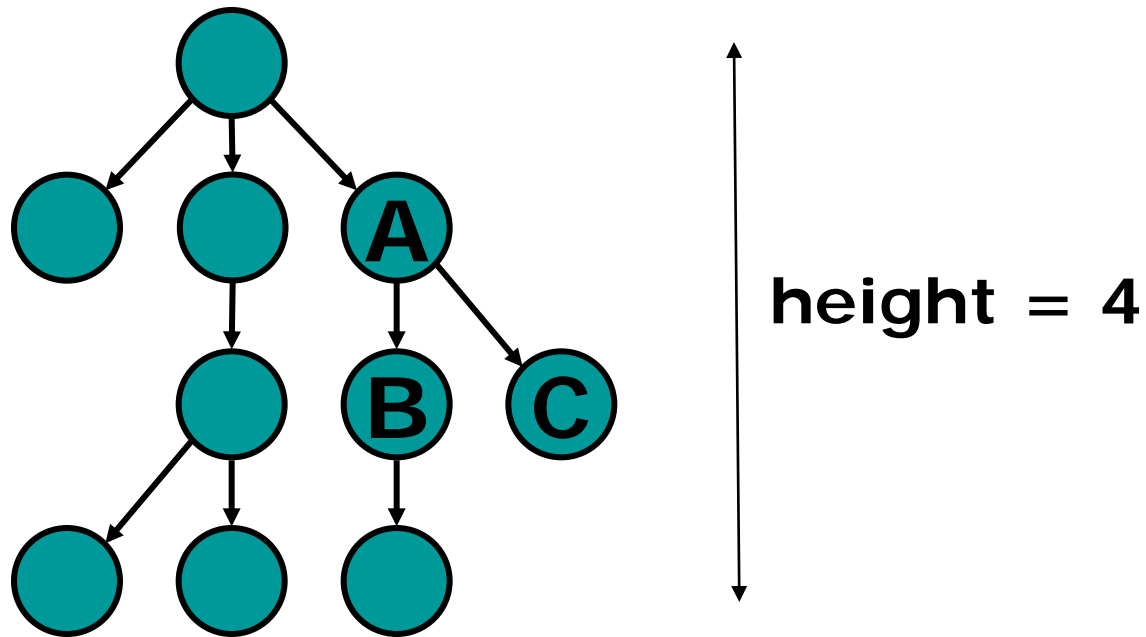


|   |   |
|---|---|
| 0 | A |
| 1 | C |
| 2 | E |
| 3 | S |
| 4 | I |
| 5 | G |
|   | : |

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

# Height of a tree

- **Maximum level** of the nodes in the tree



# Height of a tree (cont'd)

**height(T)**

if T is empty

return 0

else

return 1 + max (height(T.left), height(T.right))

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

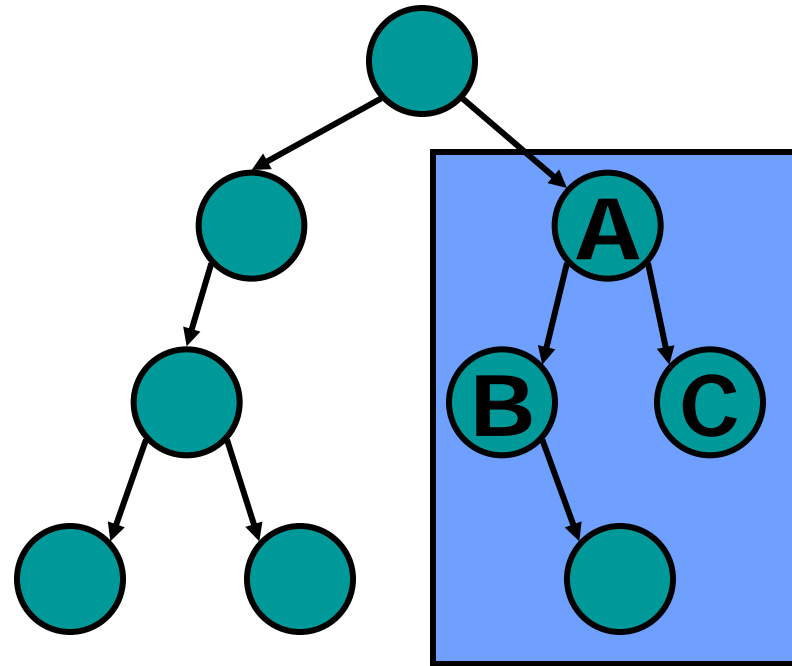
# Balanced Binary Trees

- **Balanced** binary trees
  - A binary tree is **balanced** if the height of any node's right subtree differs from the height of the node's left subtree by no more than 1
- Full binary trees are complete
- Complete binary trees are balanced

# Size of a tree

## ■ Number of nodes in the tree

- The **size** of the subtree rooted at A is 4.



# Size of a Tree (cont'd)

**size(T)**

if T is empty

return 0

else

return 1 + size(T.left) + size(T.right)



---

# Binary Tree Traversal

---

# Traversing a Tree

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

# Post-order Traversal

Traverse the subtrees first before processing the node

`postorder(T)`

**if** T is not empty **then**

`postorder(T.left)`

`postorder(T.right)`

**process T.item**

# Traversal Example

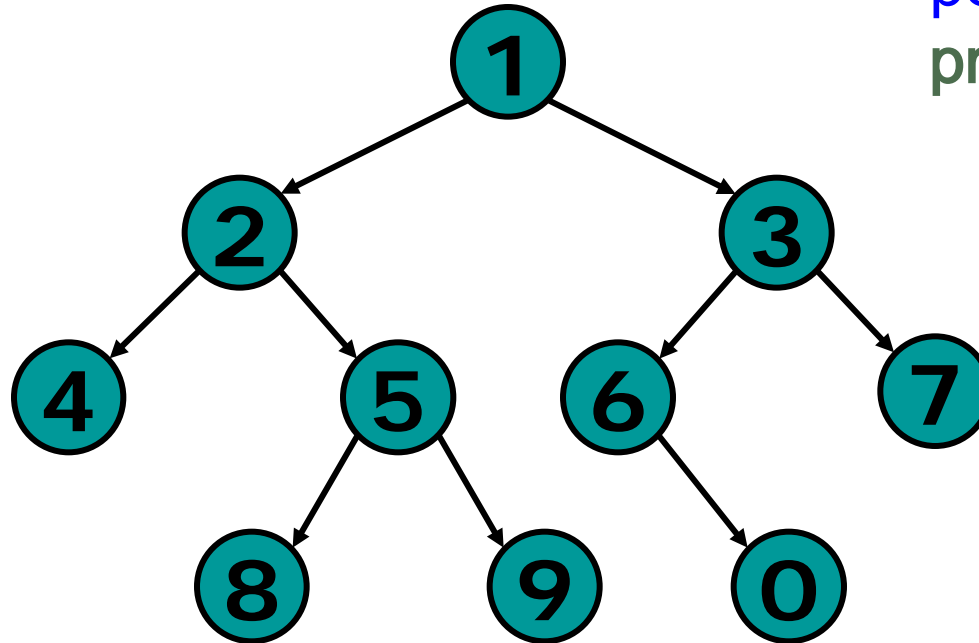
`postorder(T)`

if T is not empty then

`postorder(T.left)`

`postorder(T.right)`

`process T.item`



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

# Pre-order traversal

Process the node first before traversing the subtrees

**preorder**(T)

**if** T is not empty **then**

**process** T.item

**preorder**(T.left)

**preorder**(T.right)

# Traversal Example

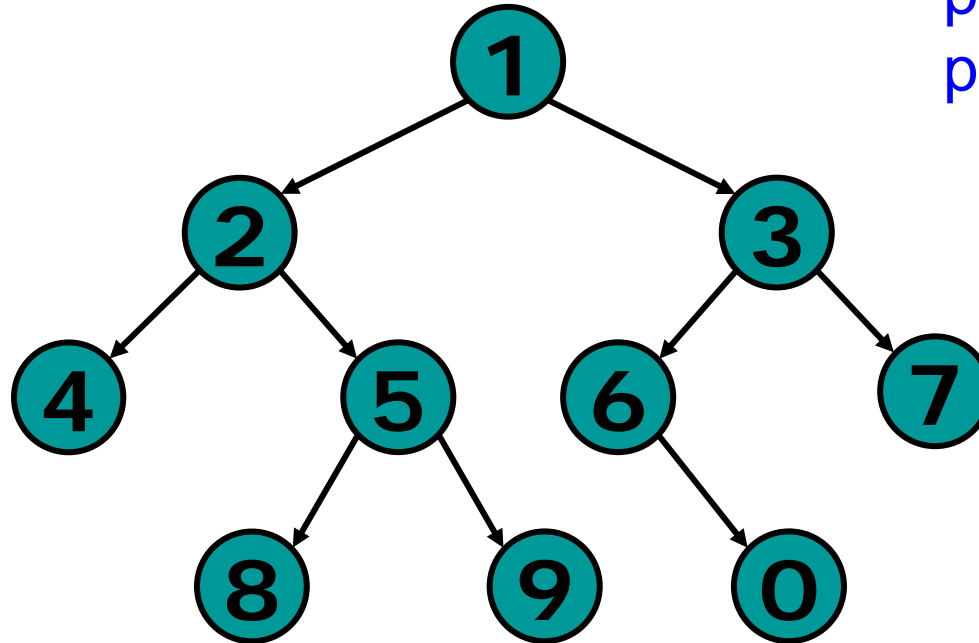
`preorder(T)`

if T is not empty then

  process T.item

`preorder(T.left)`

`preorder(T.right)`



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

# In-order Traversal

**inorder(T)**

**if** T is not empty **then**

**inorder**(T.left)

**process** T.item

**inorder**(T.right)

# Traversal Example

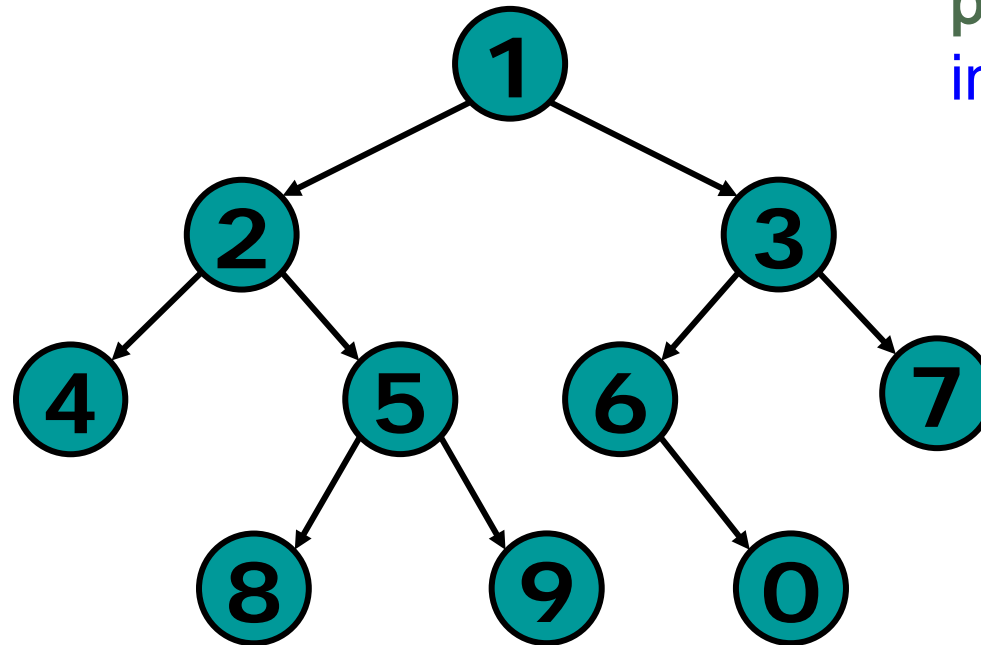
`inorder(T)`

if T is not empty then

`inorder(T.left)`

process T.item

`inorder(T.right)`

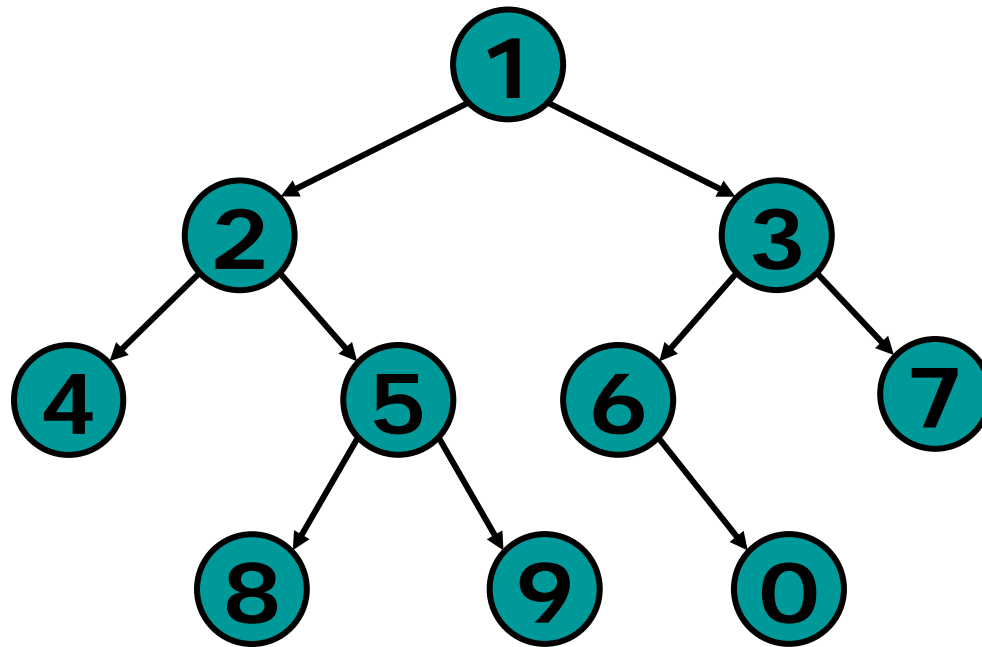


**In-order: 4 2 8 5 9 1 6 0 3 7**



# 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**

# levelOrder(T) (using a queue)

**if** T is empty **return**

Q = **new** Queue

Q.enqueue(T)

**while** Q is not empty

curr = Q.dequeue()

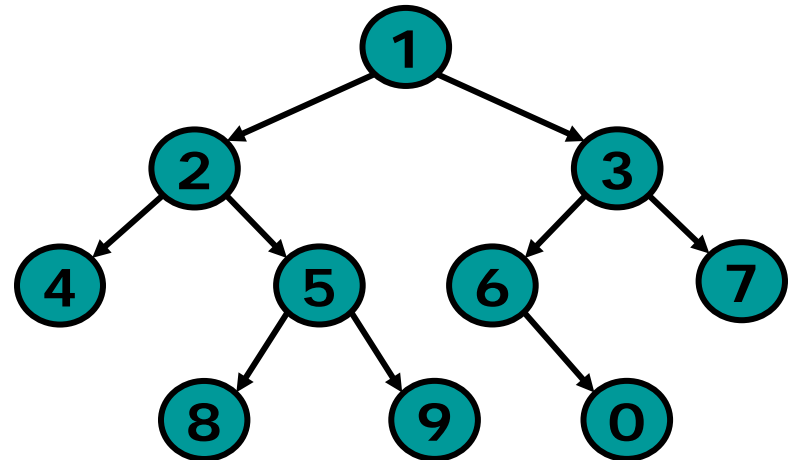
**process curr.item**

**if** curr.left is not empty

Q.enqueue(curr.left)

**if** curr.right is not empty

Q.enqueue(curr.right)

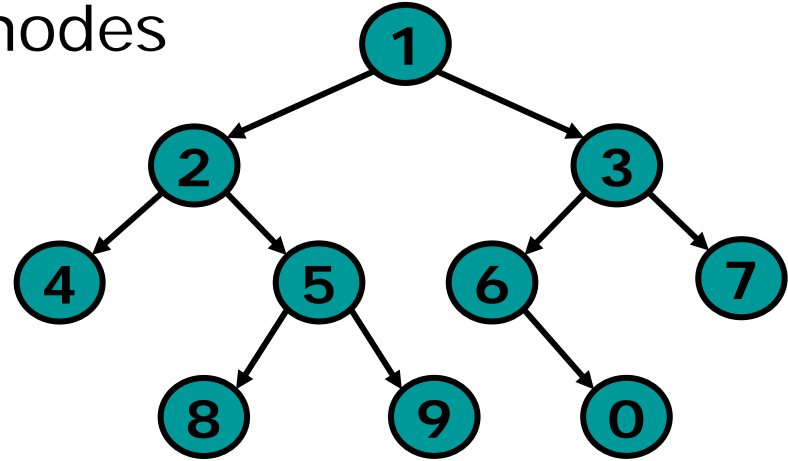


# levelOrder(T) (using a queue, cont'd)

**Note:** The numbers are references to the nodes

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 |   |



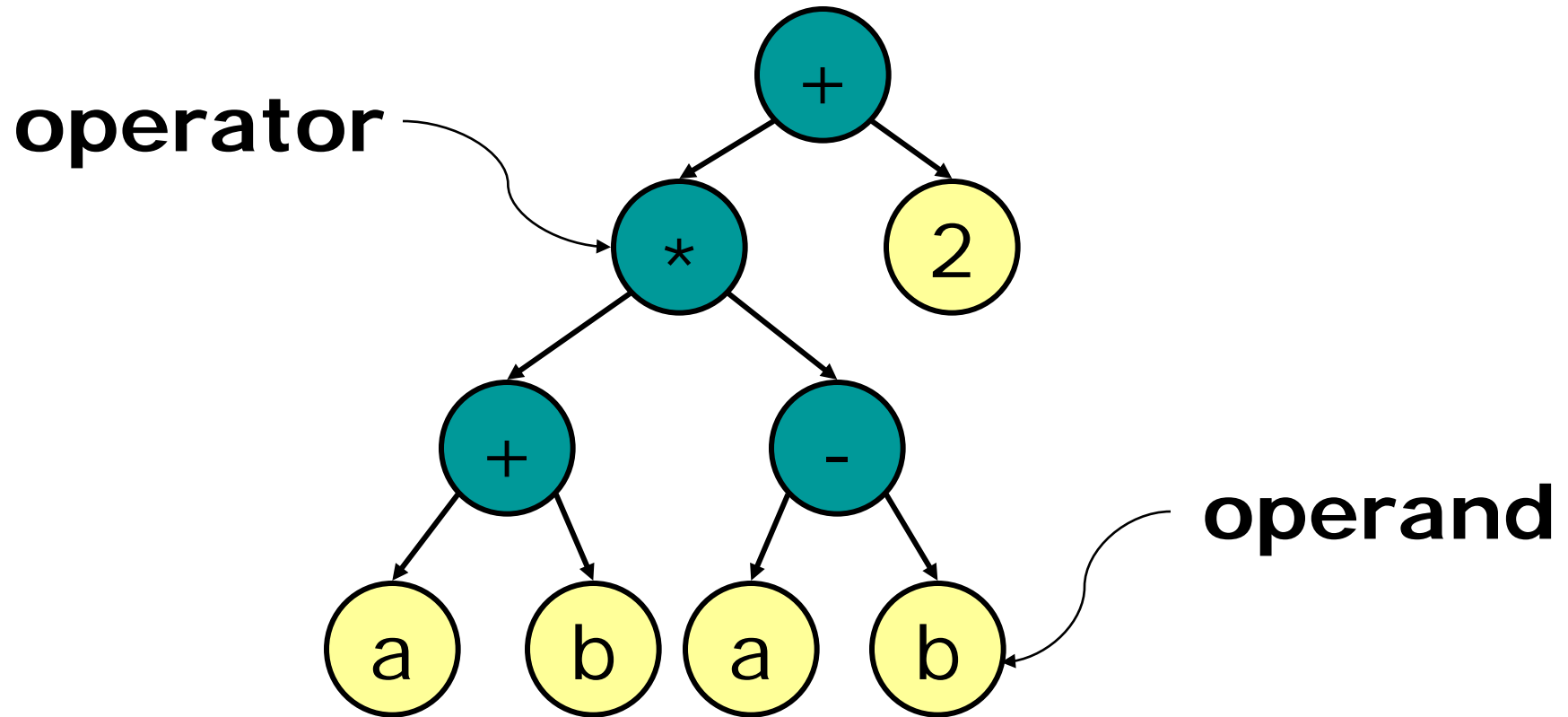
if T is empty return  
Q = new Queue  
Q.enqueue(T)

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

# Expression Trees

---

# Evaluating Expression Tree



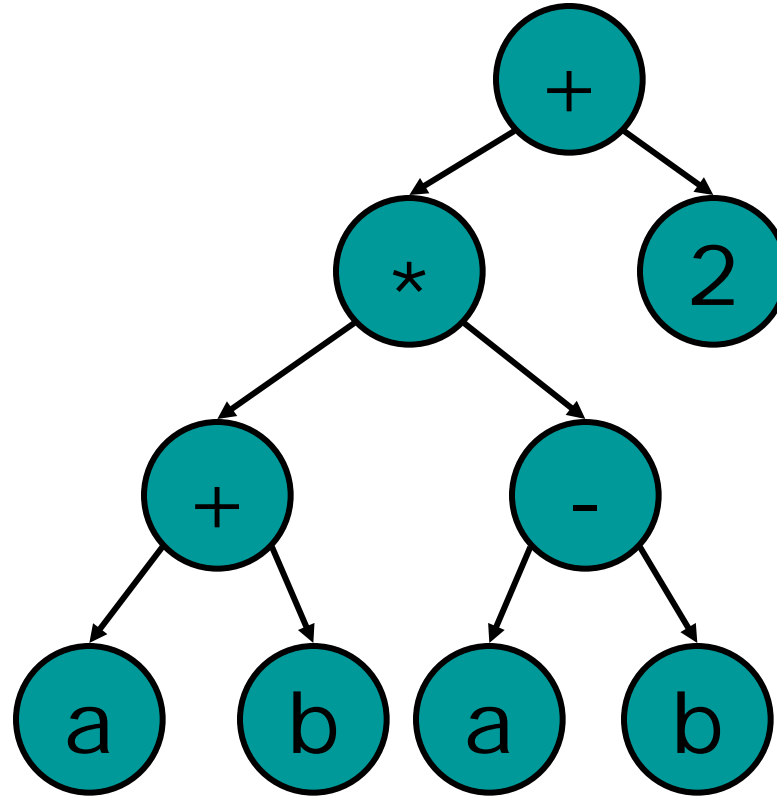
**Leaf nodes** (or **leaves**) store operands.  
**Internal nodes** and **root** store operators

# Traversing Expression Tree

Different traversal order of the expression tree can give different expression formats

- Inorder traversal = infix format
- Preorder traversal = prefix format
- Postorder traversal = postfix format

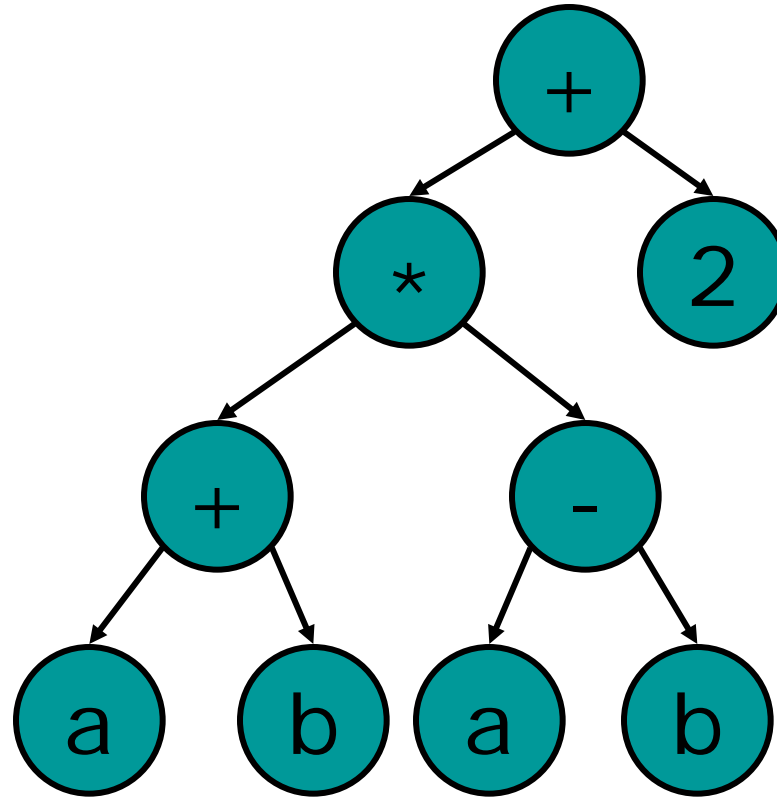
# Traversing Expression Tree (infix)



**in-order:**  $((a + b) * (a - b)) + (2)$

**Note:** produces the usual representation of algebraic expression

# Traversing Expression Tree (postfix)



**Post-order:** (((a b +) (a b -) \*) 2 +)

**Note:** Brackets can be omitted, i.e. **a b + a b - \* 2 +**  
(also called Reverse Polish notation)



# Construction of Expression Tree

Can be constructed by using the postfix expression and a stack:

- loop through the expression for each character

  - If character is operand push into stack

  - If character is operator, pop two values from stack and make them its children and push node into stack

- At the end, only element of stack will be root of expression tree

# Example: $a\ b\ +\ a\ b\ -\ *\ 2\ +$

a push a

b push b

+ pop b then a

make tree with + as root

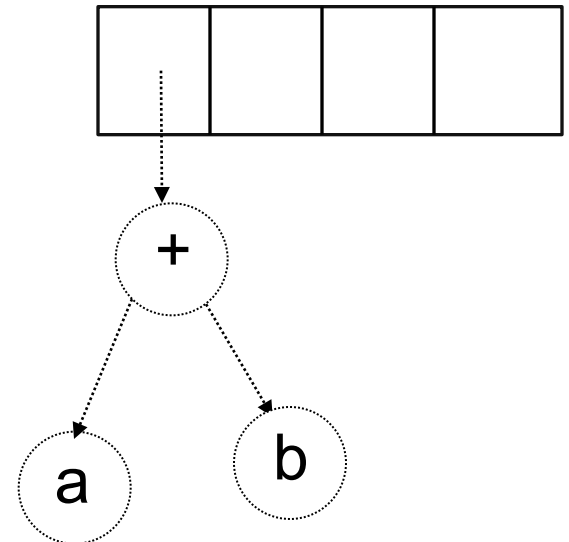
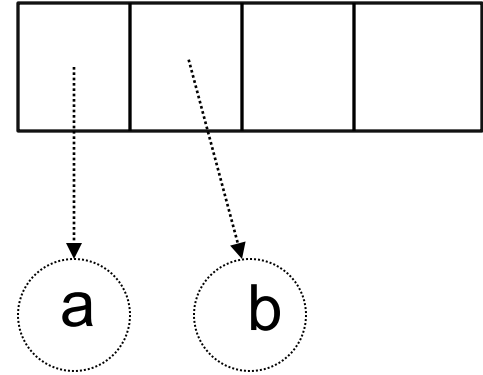
and a and b as left subtrees

push root

a push a

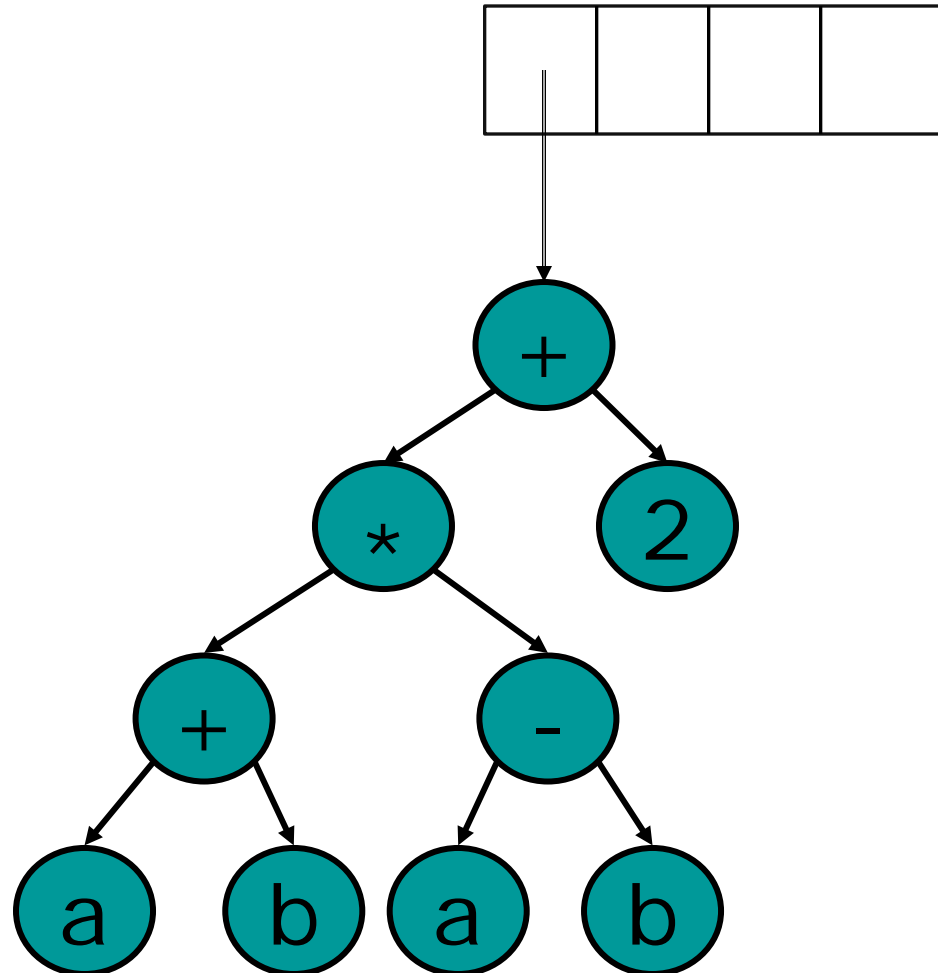
b push b

etc....



# Example: $a b + a b - * 2 +$

■ result



# 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 “\*”

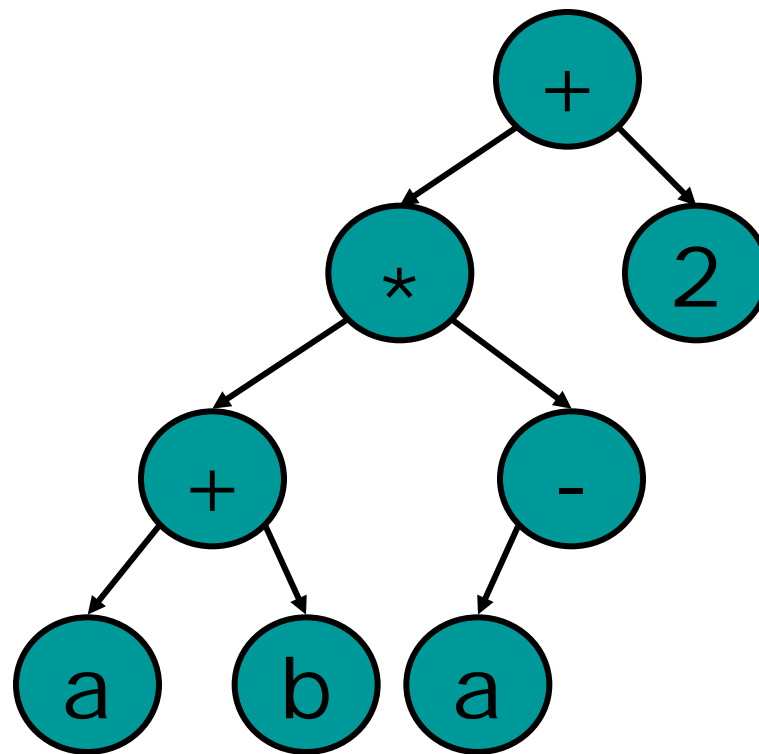
return eval(T.left) \* eval(T.right)

**Q:** How to handle operators  $/$ ,  $-$ , and **unary -** ?

**Q:** Do you need to consider the **priorities** of the operators?

# Unary operator

Expression tree for  $((a + b) * (-a)) + (2)$



# Binary Search Tree (BST)

# Tables

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

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

# Table ADT operations

A table ADT provides operations to maintain a set of data, each can be uniquely identified by a **key**.

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



# Running Time of operations

|        | Unsorted<br>Array/List | Sorted<br>Array | Sorted<br>Linked<br>List | Hash<br>Table |
|--------|------------------------|-----------------|--------------------------|---------------|
| insert | $O(1)$                 | $O(N)$          | $O(N)$                   | $O(1)$        |
| delete | $O(N)$                 | $O(N)$          | $O(N)$                   | $O(1)$        |
| search | $O(N)$                 | $O(\lg N)$      | $O(N)$                   | $O(1)$        |

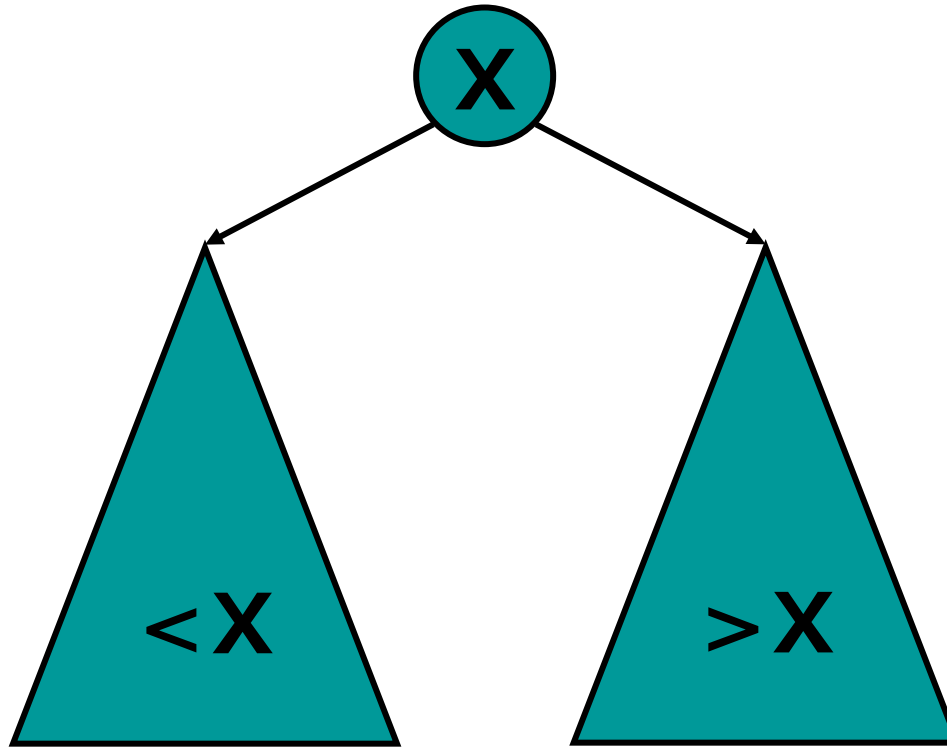
# Binary Search Tree (BST)

- insert, delete, and search can usually be done in

$$O(\log_2 N)$$

**Q:** So, are the update operations' performances of BST better than unsorted and sorted array?

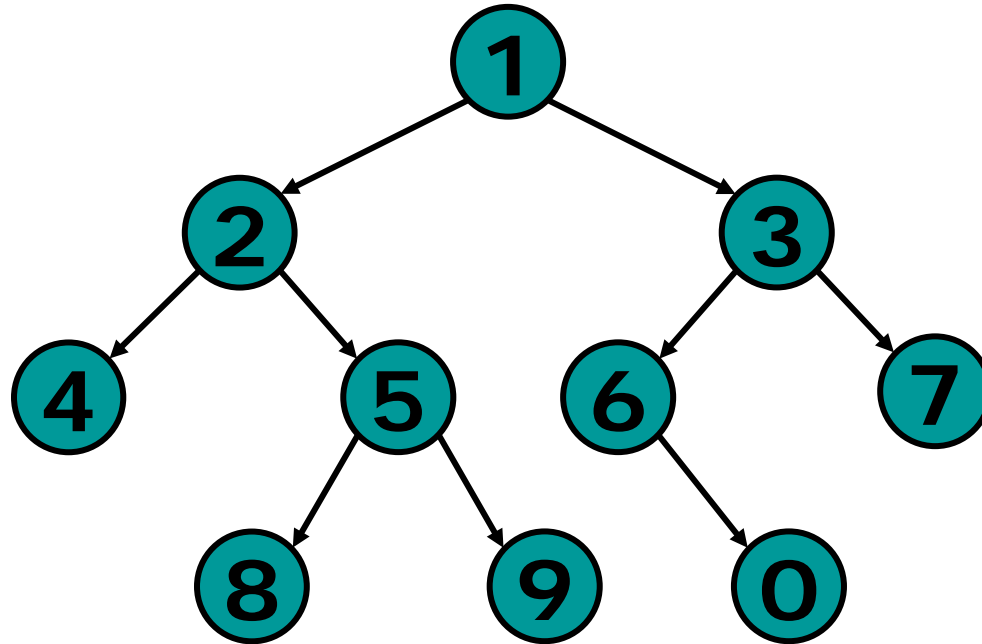
# BST Property



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.

**Q:** Can we have the **same key values** in a BST?

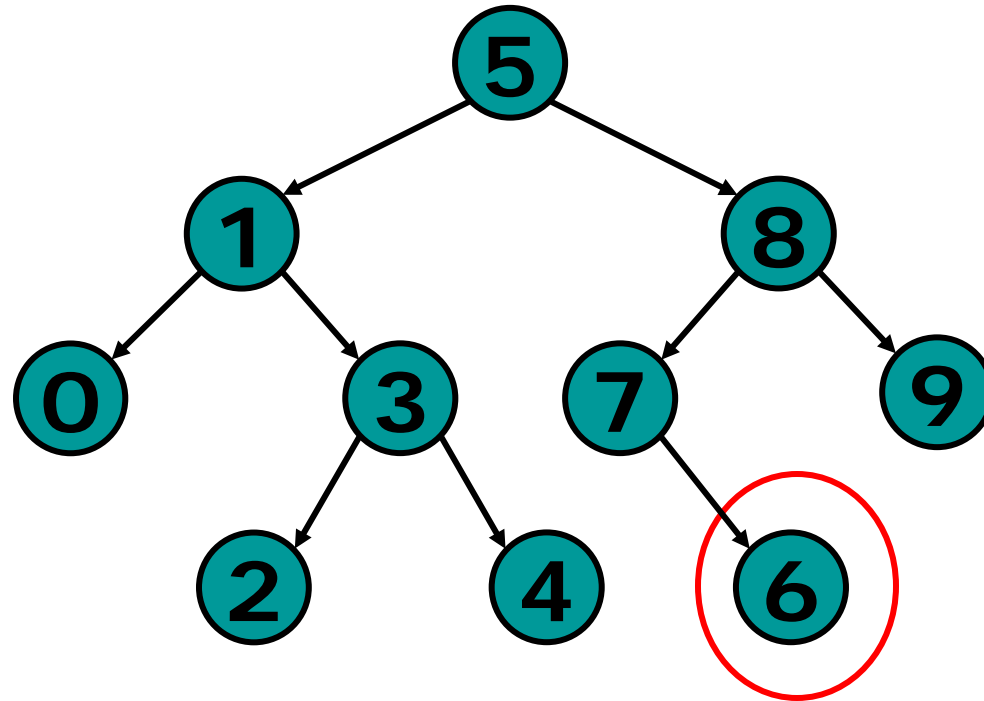
# Example



**NOT** a **BST**. **Why?**

Keys in Left subtree of root > root

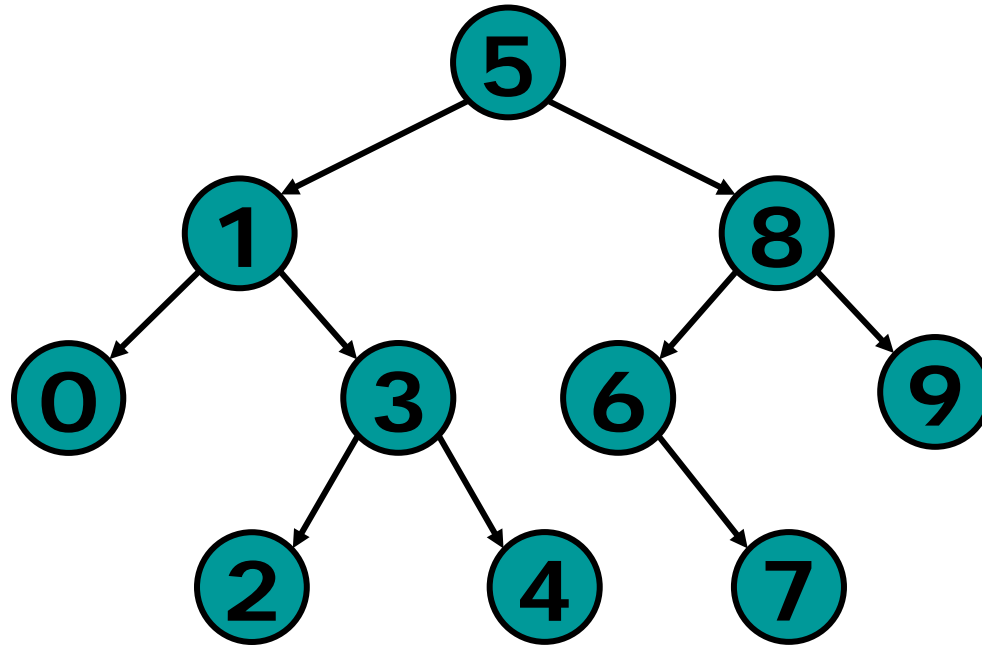
# Example



**A BST ?**

**NO. Why?**

# Example



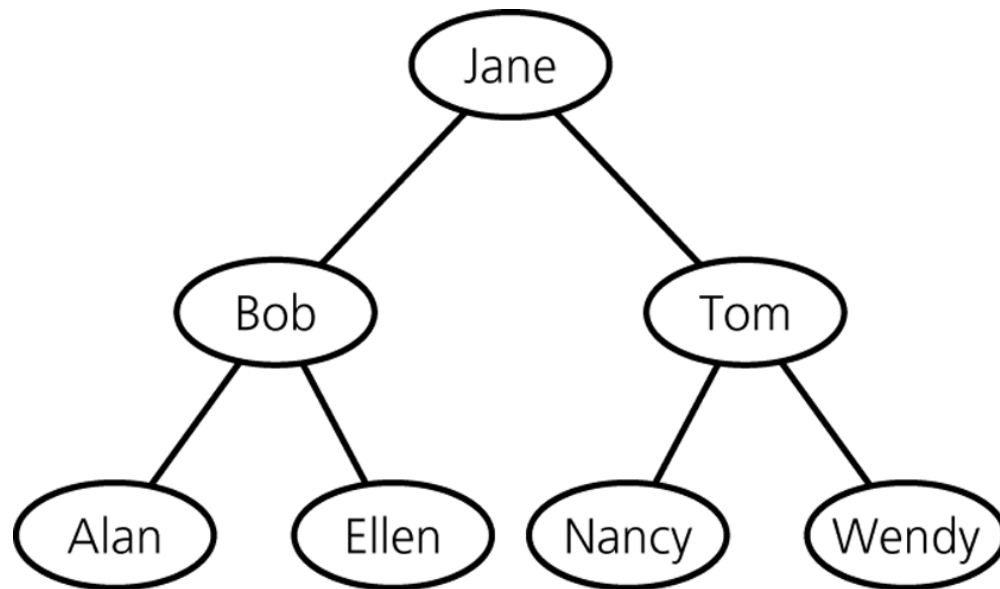
**A BST**

**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).

# Example

BST of names:



# Compare heap with BST

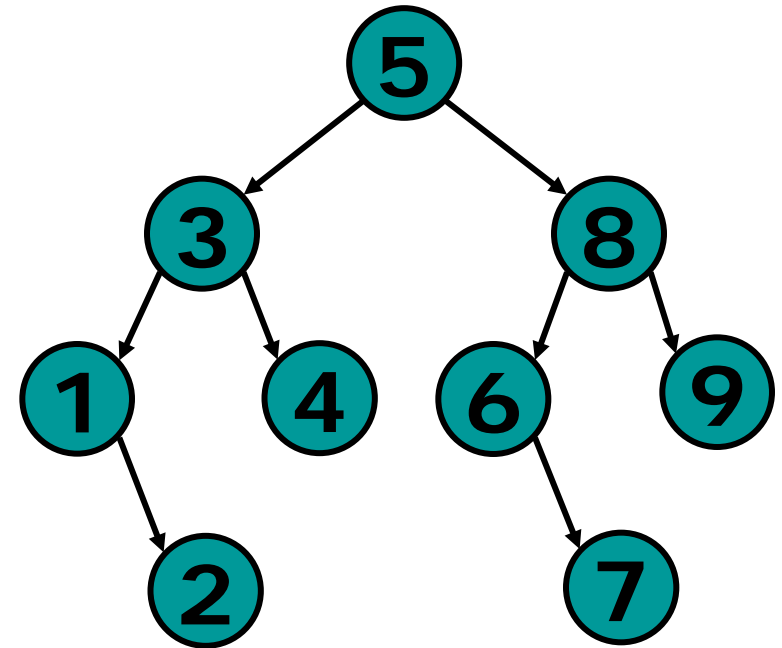
- Both are binary trees
- Difference
  - Heap maintains **heap property**
    - It is not a search tree
  - BST maintains **BST property**
    - It is a search tree



# Operations on BST

# Finding Minimum Element

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



Q: How to find maximum values?

Q: How to find **top-k** (or **bottom-k**) values?

e.g. find top-3 values.

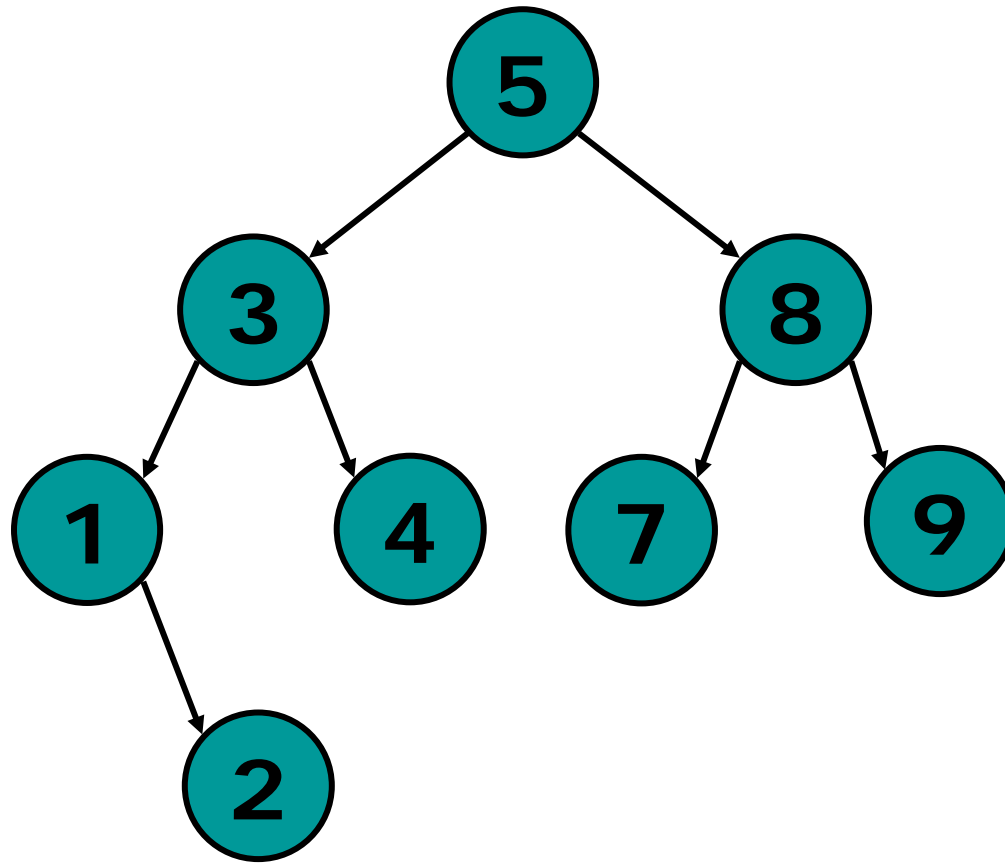
# Searching $x$ in $T$ (iterative solution)

```
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
```

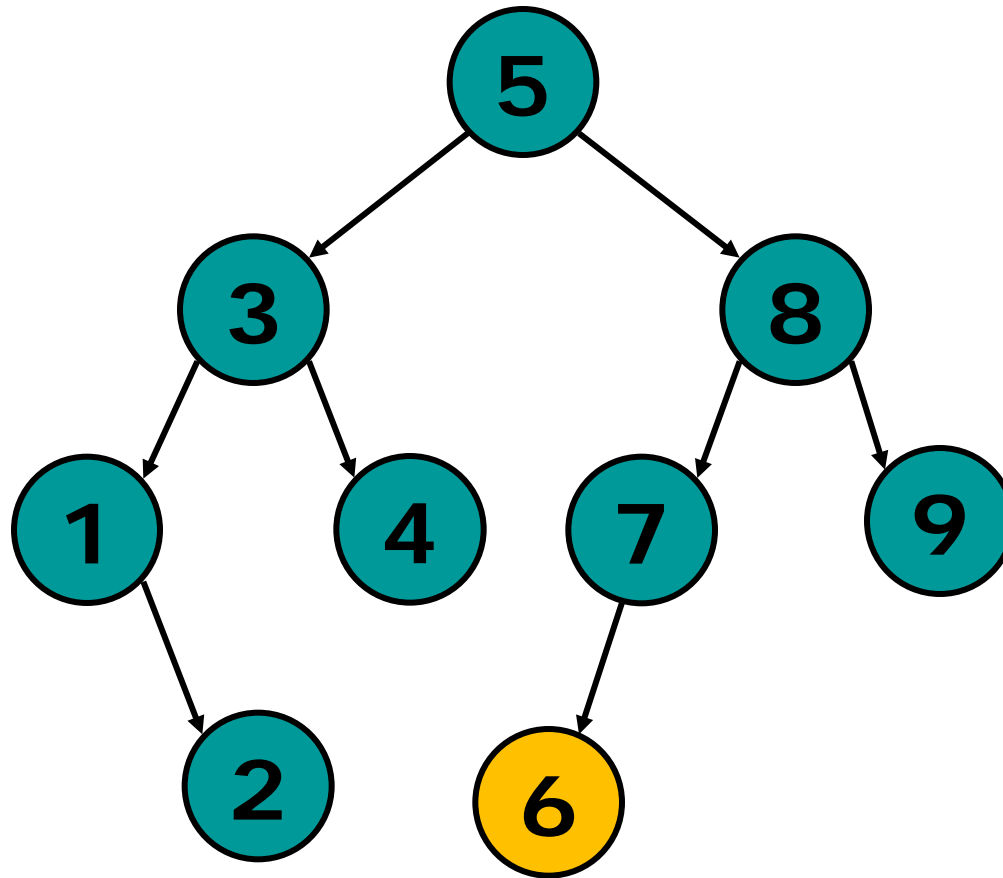
# Searching x in T (recursive solution)

```
function search(x, T)
  if T is empty
    return null
  if x == T.item then
    return T
  else if x < T.item
    return search(x, T.left)
  else
    return search(x, T.right)
```

# 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**

**ERROR!** // X already in T

**return** T // return the new tree T

# How to delete?

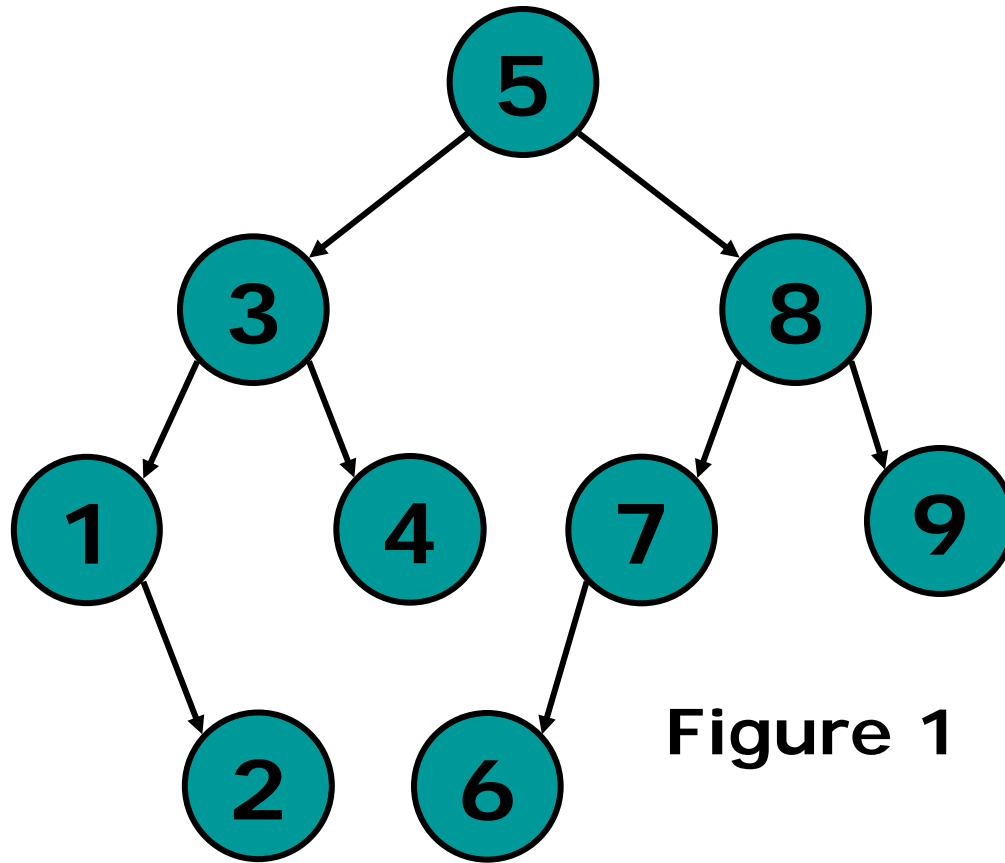


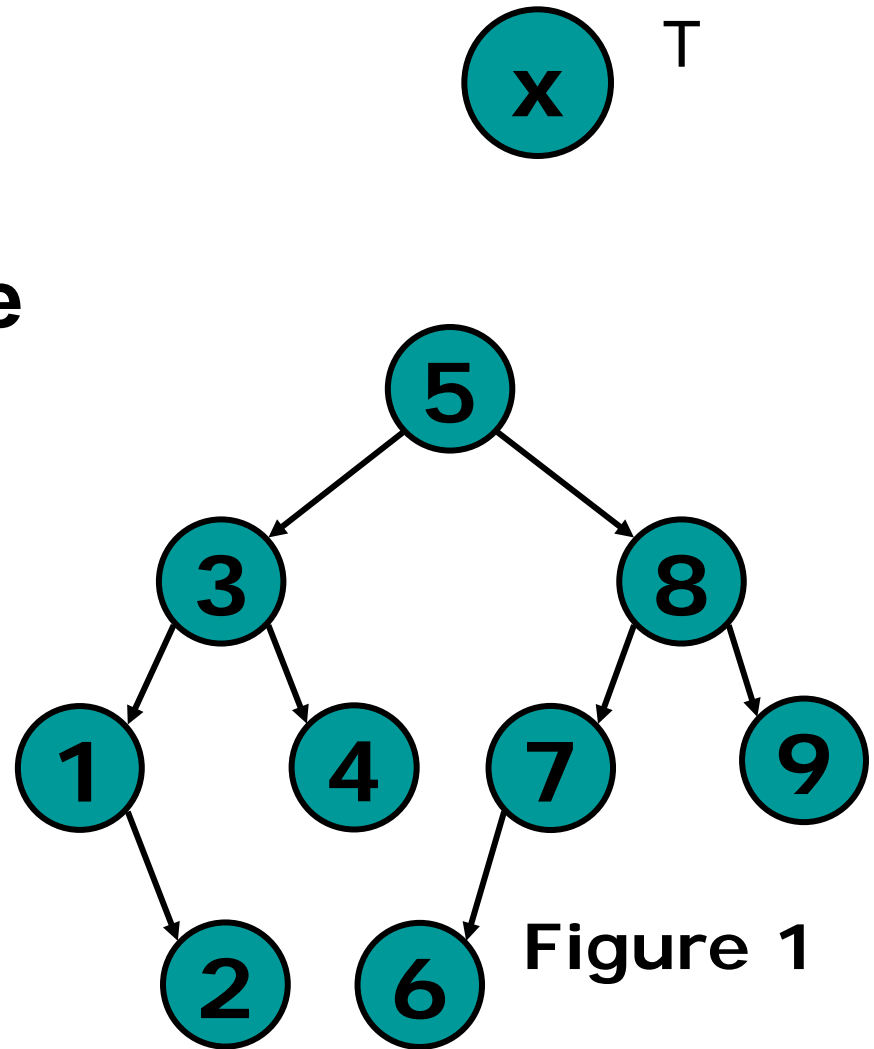
Figure 1



# delete(x,T): Case 1

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

e.g. Delete 4 in Figure 1



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

if T has only 1 child (left)

if  $x == T.item$

**return T.left**

**else**

T.left = delete(x,T.left)

**return T**

e.g. delete 7 in Figure 1

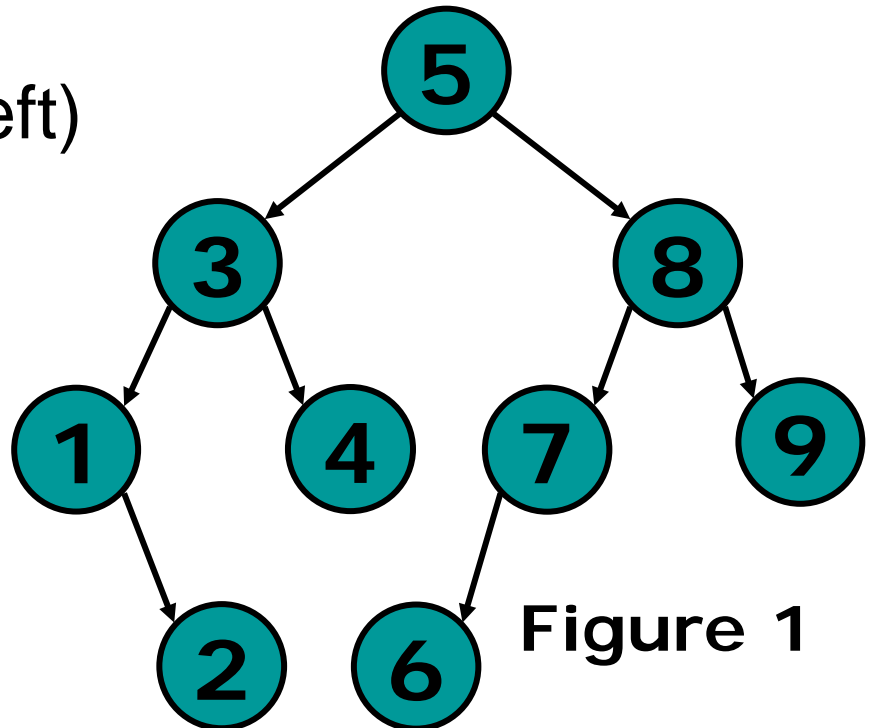
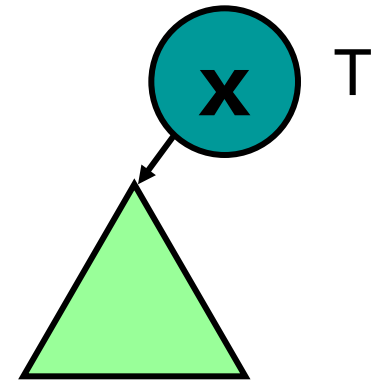


Figure 1

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

if T has only 1 child (right)

if  $x == T.item$

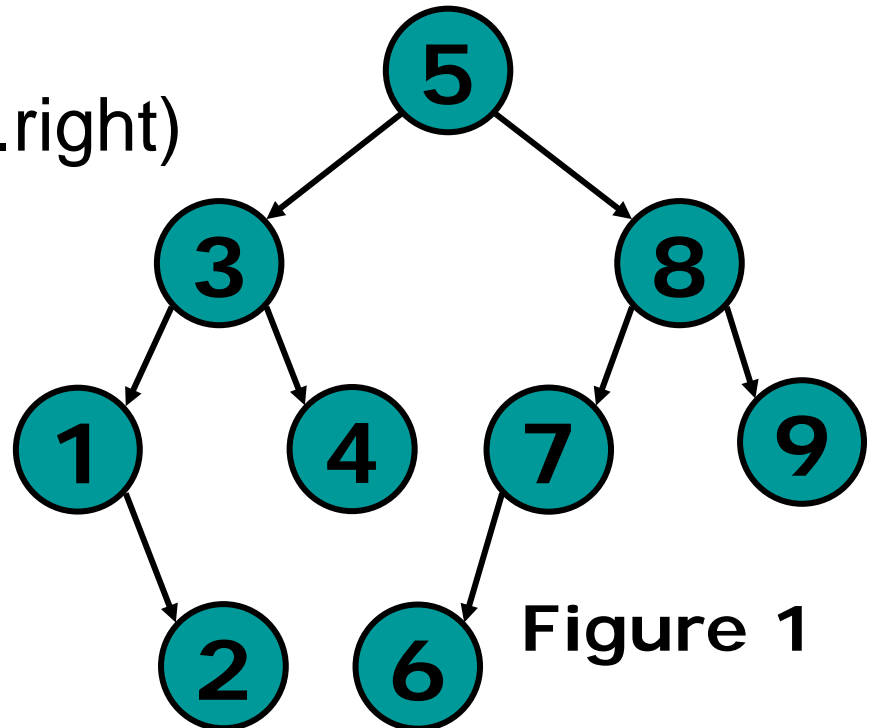
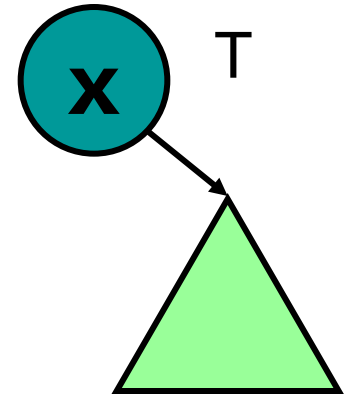
**return** T.right

**else**

T.right = delete(x, T.right)

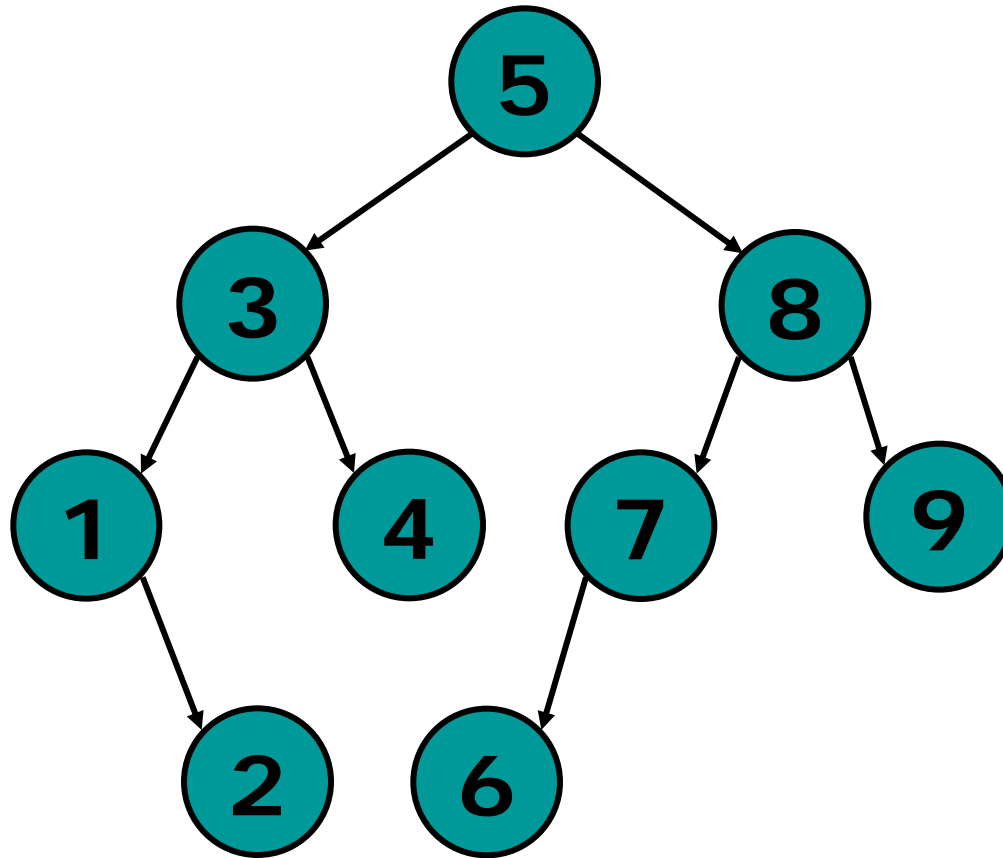
**return** T

e.g. delete 1 in Figure 1



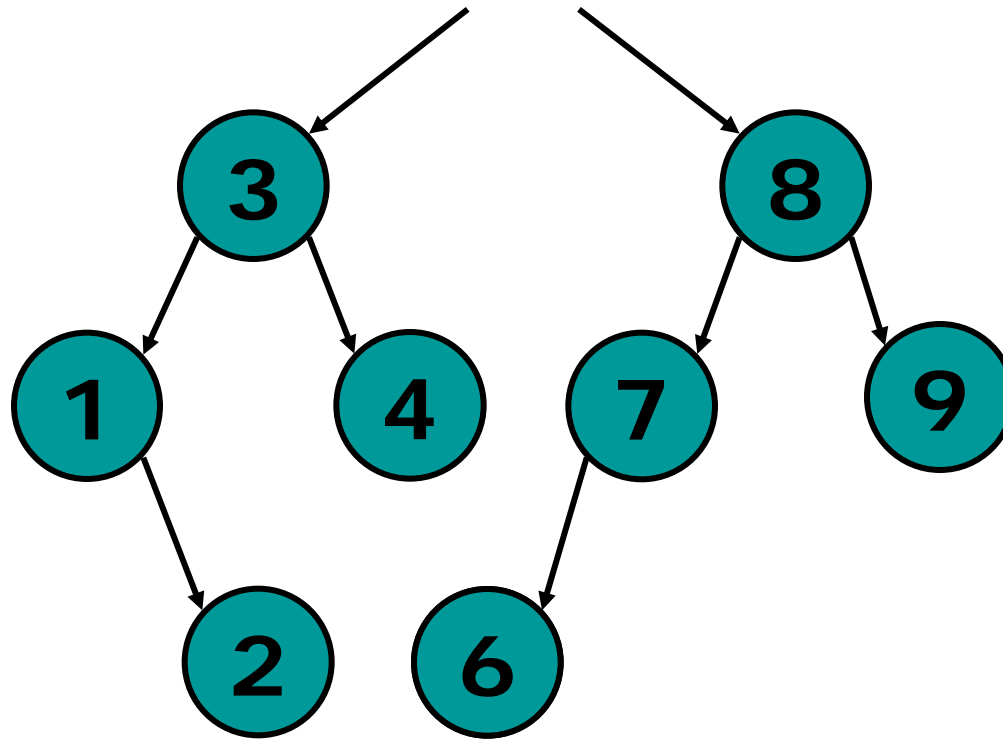
# delete(x,T): Case 3

Node to be deleted has 2 children  
e.g. delete 5



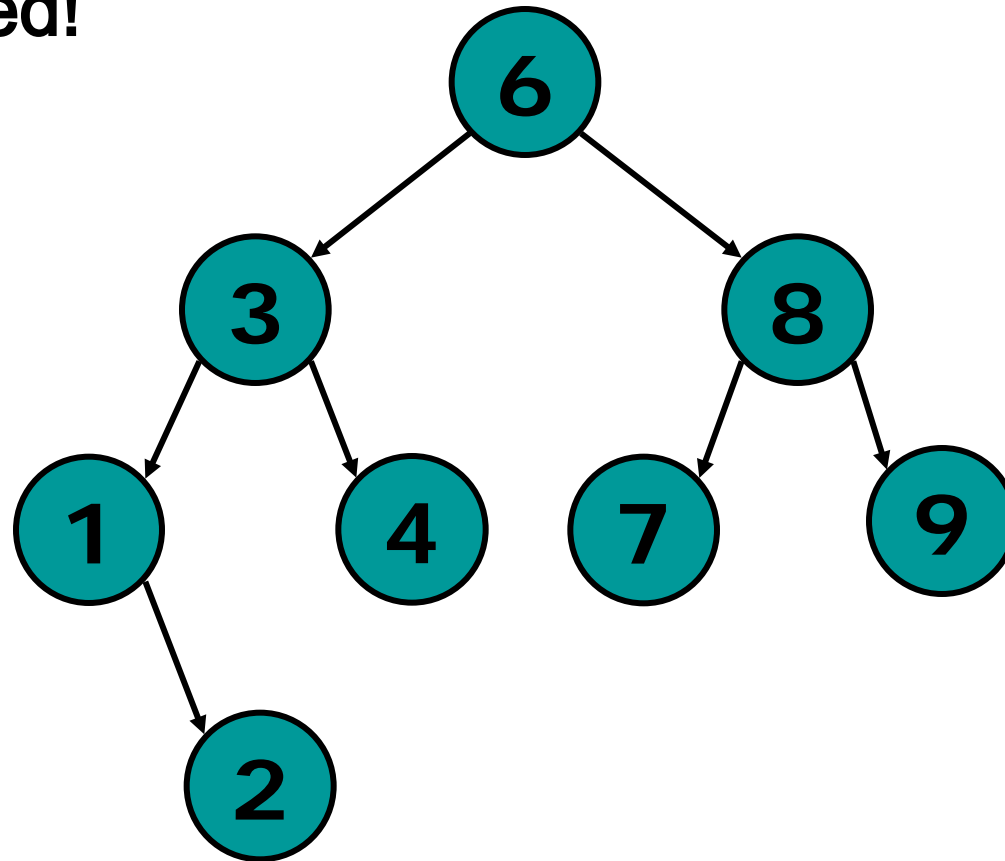
# delete(x,T): Case 3

e.g. delete 5



# delete(x,T): Case 3

**5 deleted!**



## delete(x,T): Case 3

**if** T has **two** children

**if**  $x == T.item$

$T.item = \text{findMin}(T.right)$  // replace T.item by  
        // the min. item of the right subtree

$T.right = \text{delete}(T.item, T.right)$

        // delete **x (i.e. T.item)** from the right subtree

**else if**  $x < T.item$

$T.left = \text{delete}(x, T.left)$

**else**

$T.right = \text{delete}(x, T.right)$

**return T**

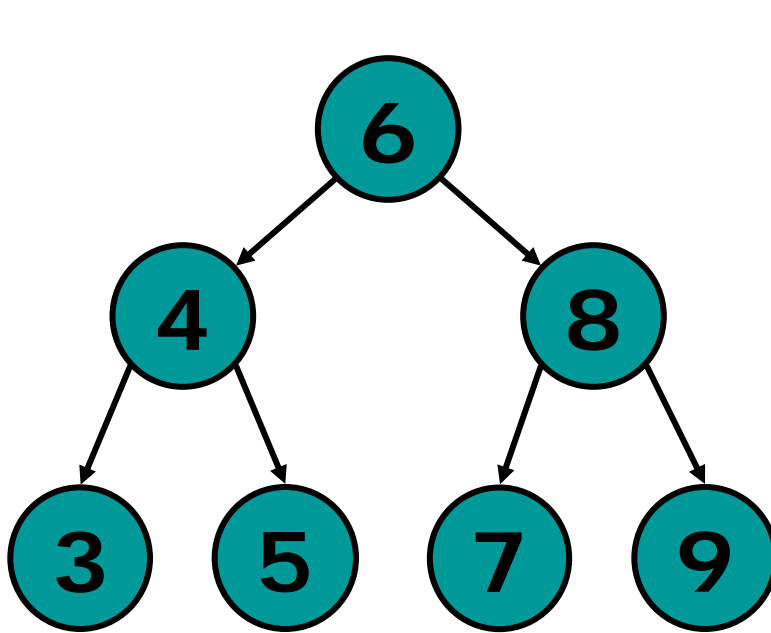
# Running time of BST

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



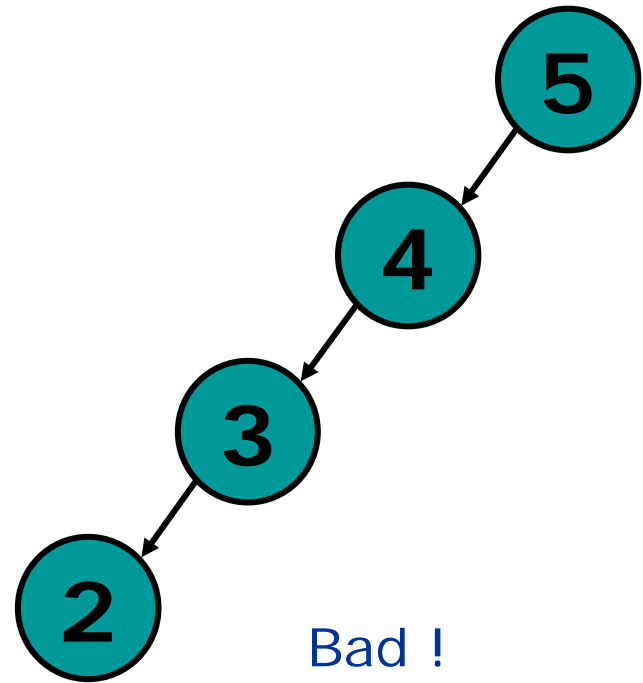
# Running time of BST (cont'd)

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



Good !

$$h = O(\log_2 N)$$



Bad !

$$h = O(N)$$

When you insert nodes in **increasing** or **decreasing** order, you get a **skewed** tree

# Applications of BST

## ■ Treesort

- Uses binary search tree to sort an array of records into search-key order
  - Average case:  $O(n * \log n)$
  - Worst case:  $O(n^2)$

# Applications of BST

- Algorithms for saving a binary search tree
  - Saving a binary search tree and then restoring it to its original shape
    - Uses preorder traversal to save the tree to a file
  - Saving a binary tree and then restoring it to a balanced shape
    - Uses inorder traversal to save the tree to a file
    - Can be used if the data is sorted and the number of nodes in the tree is known

# STL set (ordered)

- set is a type of associative containers where each key is unique.
- The key cannot be modified once it is added to the set, though it is possible to remove and add the modified key of that element.
- Usually implemented as binary search trees
- <https://en.cppreference.com/w/cpp/container/set>

# STL map (ordered)

- Maps are associative containers that store elements in a mapped fashion. Each element has a key value and a mapped value.
- No two mapped values can have same key values.
- Usually implemented as binary search trees
- <https://en.cppreference.com/w/cpp/container/map>

# The STL Search Algorithms for Sorted Ranges

- *binary\_search*

- Returns true if a specified value appears in the sorted range

- *lower\_bound*

- Returns an iterator to the first element that is not less than key; (or to past-the-end iterator if not found)

- *upper\_bound*

- Returns an iterator to the first element greater than the given key (past-the-end iterator returned if not found)

- *equal\_range*

- Returns a pair of iterators that indicate the first and one past the last occurrence of a value

# Example: binary\_search

```
#include <iostream>
#include <algorithm>
using namespace std;
```

```
int main() {
    int nums[] = { -242, -1, 0, 5, 8, 9, 11 };
    int start = 0;
    int end = 7;
    for( int i = 0; i < 10; i++ ) {
        if( binary_search( nums+start, nums+end, i ) ) {
            cout << "nums[] contains " << i << endl;
        } else {
            cout << "nums[] DOES NOT contain " << i << endl;
        }
    }
    return 0;
}
```

# Example: lower/upper bound

```
// map::lower_bound/upper_bound
```

```
#include <iostream>
```

```
#include <map>
```

```
int main () {
```

```
    std::map<char,int> mymap;
```

```
    std::map<char,int>::iterator itlow, itup;
```

```
    mymap['a'] = 20; mymap['b'] = 40;
```

```
    mymap['c'] = 60; mymap['d'] = 80;
```

```
    mymap['e'] = 100;
```



## Example: lower/upper bound (cont'd)

```
itlow = mymap.lower_bound ('b'); // itlow points to b
itup = mymap.upper_bound ('d'); // itup points to e (not d!)
mymap.erase(itlow,itup);        // erases [itlow,itup) i.e. b to d

// print content: keys a and e left
for (std::map<char,int>::iterator it=mymap.begin();
      it!=mymap.end(); ++it)
    std::cout << it->first << " => " << it->second << '\n';
return 0;
}
```

# Example: lower\_bound

```
#include <algorithm>
```

```
#include <vector>
```

```
using namespace std;
```

```
int main() {
```

```
    vector<int> nums;
```

```
    nums.push_back( -242 );
```

```
    nums.push_back( -1 );
```

```
    nums.push_back( 0 );
```

```
    nums.push_back( 5 );
```

```
    nums.push_back( 8 );
```

```
    nums.push_back( 8 );
```

```
    nums.push_back( 11 );
```

# Example: lower\_bound (cont'd)

```
    cout << "Before nums is: ";
    for ( unsigned int i = 0; i < nums.size(); i++ ) {
        cout << nums[i] << " ";
    }
    cout << endl;
    vector<int>::iterator result;
    int new_val = 7;
    result = lower_bound( nums.begin(), nums.end(), new_val );
    nums.insert( result, new_val );
    for ( unsigned int i = 0; i < nums.size(); i++ ) {
        cout << nums[i] << " ";
    }
    return 0;
}
```

# Example: equal\_range

```
#include <algorithm>
using namespace std;
int main() {
    // data declared in the previous example
    pair<vector<int>::iterator, vector<int>::iterator> result;
    int new_val = 8;
    result = equal_range( nums.begin(), nums.end(), new_val );
    cout << "The first place that "
        << new_val
        << " could be inserted is before "
        << *result.first
        << ", and the last place that it could be inserted is before "
        << *result.second << endl;
    return 0;
}
```

# Summary

- Binary Trees are recursive
- Binary Tree traversal – pre/post/in
- Expression Trees – created using Reverse Polish notation and stack
- Binary Search Trees
- STL map/set/algorithms for sorted range

# Programming Exam

- Date: 3 Nov 2018 Saturday (week 11)
- Time: 1pm to 3pm (2 hours)
- Venues: PL1 and PL2
- Lab allocation will be uploaded in IVLE later
- Format: 2 problems with subtasks
- Coverage: all topics up to Trees/AVL Trees