# Data Structures

# Lec-5 Trees, Game Trees and Heaps

# Review about Queues and Hashing

## Queue

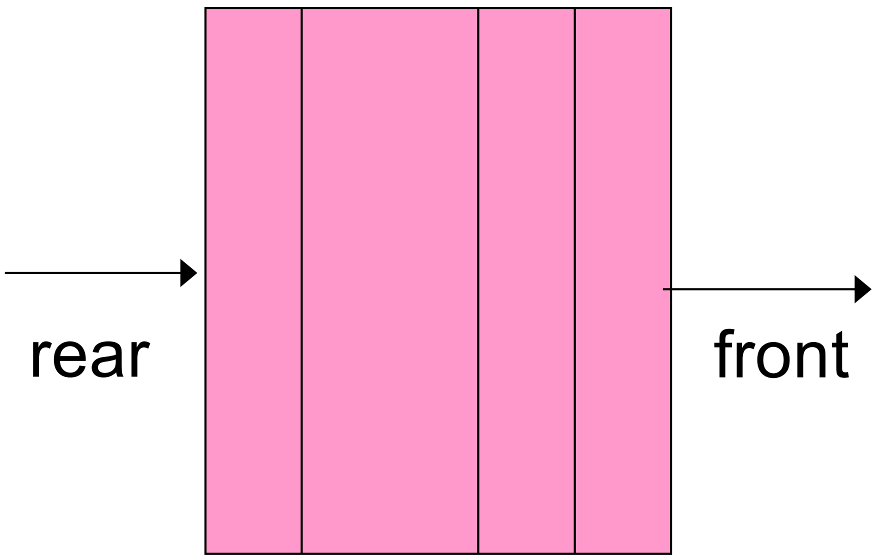
- Queue Abstract Data Type
- Sequential Allocation
- Linked Allocation
- Applications
- Priority Queues

## Hashing

- Sparse Data
- Key Based Data
- Hash Table
- Hash Functions
- Collision Resolution
- Applications

# Review: Queue

- Queue is a list with the restriction that insertions are performed at one end and deletions are performed at the other end of the list
- Also known as: First-in-first-out (FIFO) list



# Review: ADT of Queue

Value:

A sequence of items that belong to some data type ITEM_TYPE

Operations on q:

1. Boolean IsEmpty()

   Postcondition: If the queue is empty, return true, otherwise return false

2. Boolean IsFull()

   Postcondition: If the queue is full, return true, otherwise return false

3. ITEM_TYPE Dequeue() /*take out the front one and return its value*/

   Precondition: q is not empty

   Postcondition: The front item in q is removed from the sequence and returned

4. Void Enqueue(ITEM_TYPE e) /*to append one item to the rear of the queue*/

   Precondition: q is not full

   Postcondition: e is added to the sequence as the rear one

# Review: Implementation of Queue
# Sequential Allocation (Using Array)

```
#define TOTAL_SLOTS 100
class MyQueue
{
	private:
		int front; //the index of the front slot that contains the front item
		int rear;  //the index of the first empty slot at the rear of queue
		int items[TOTAL_SLOTS];
};
```

| | | | | | |
|---|---|---|---|---|---|
| Slot#0 | Slot#1 | Slot#2 | … | Slot#98 | Slot#99 |

Suppose some items are appended into the queue:

| | | | | | |
|---|---|---|---|---|---|
| Slot#0<br>Item A | Slot#1<br>Item B | Slot#2<br>Item C | Slot#3<br>Empty | …<br>Empty | Slot#99<br>Empty |

front    rear

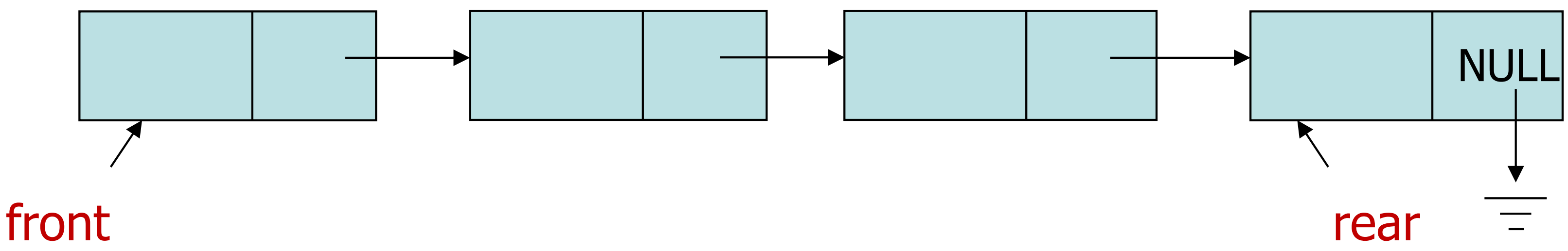
# Review: Implementation of Queue Sequential Allocation (Using Array)

```
bool MyQueue::isEmpty()
{
        return (front==rear);
}
bool MyQueue::isFull()
{
        return((rear+1)%TOTAL_SLOTS==front);
}
```

```
void MyQueue::enqueue(int data)
{
        if(!isFull())
        {
                items[rear]=data;
                rear=(rear+1) %TOTAL_SLOTS;
        }
}
```

```
int MyQueue::dequeue( )
{
        int ret_val;
        if(!isEmpty())
        {
                ret_val=items[front];
                front=(front+1)%TOTAL_SLOTS;
                return ret_val;
        }
}
```

# Review: Implementation of Queue Using Linked List




- Queue can also be implemented with linked list.
- A pointer front points to the first node of the queue.
- A pointer rear points to the last node of the queue.
- If the queue is empty, then front=rear=NULL.
- When will the queue be full?

# Review: Linked Implementation of Queue

```
// Queue.h
#include “stdlib.h”
{
	class Queue
	{
		public:
				Queue( );
				bool IsEmpty();
				void Enqueue(int );
				int Dequeue();
		private:
				ListNode* front;
				ListNode* rear;
				int size;
	};
}
```

```
// Queue.cpp

#include “Queue.h”
Queue::Queue()
{
		size=0;
		front=NULL;
		rear=NULL;
}
bool Queue::IsEmpty()
{
		return (front==NULL);
}
```

# Review: Linked Implementation of Queue

**To insert an item (Enqueue)**

**We have 2 cases:**

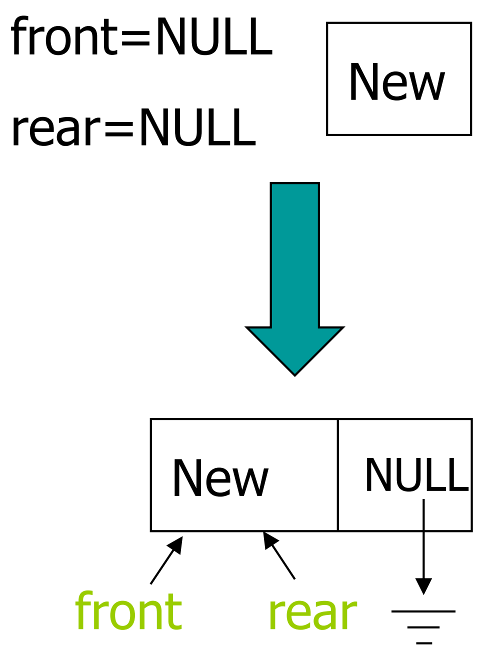
**The queue is empty or not.**

Step 1: Allocate a new slot, **p**, to store the item.

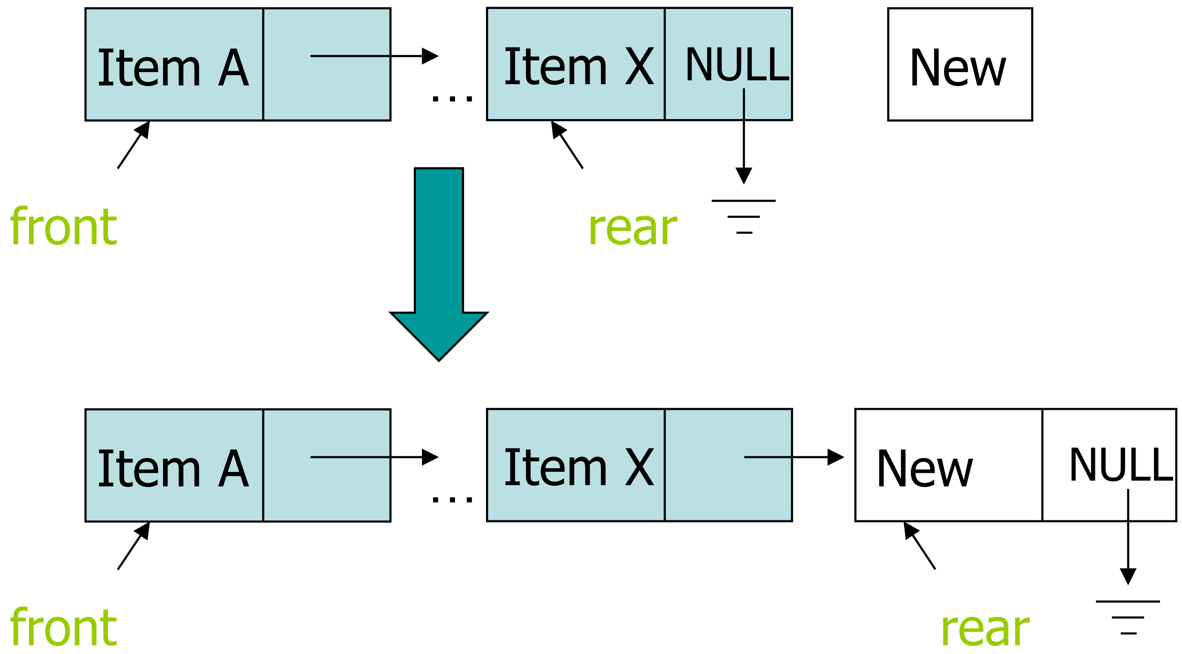
Step 2: Connect **p** to the queue (2 cases).

Step 3: Update the rear pointer to point to **p**.

Case 1: The queue is empty




Case 2: The queue is not empty



# Review: Linked Implementation of Queue

**To insert an item (Enqueue)**

**We have 2 cases:**

**The queue is empty or not.**

Step 1: Allocate a new slot, **p**, to store the item.

Step 2: Connect **p** to the queue (2 cases).

Step 3: Update the pRear pointer to point to **p**.

```
// Queue.cpp

#include "Queue.h"
void Queue::Enqueue(int data)
{
        ListNode *p=new ListNode(data);
        if (IsEmpty())
                front=p;
        else
                rear->next=p;
        rear=p;
}
```

# Review: Linked Implementation of Queue

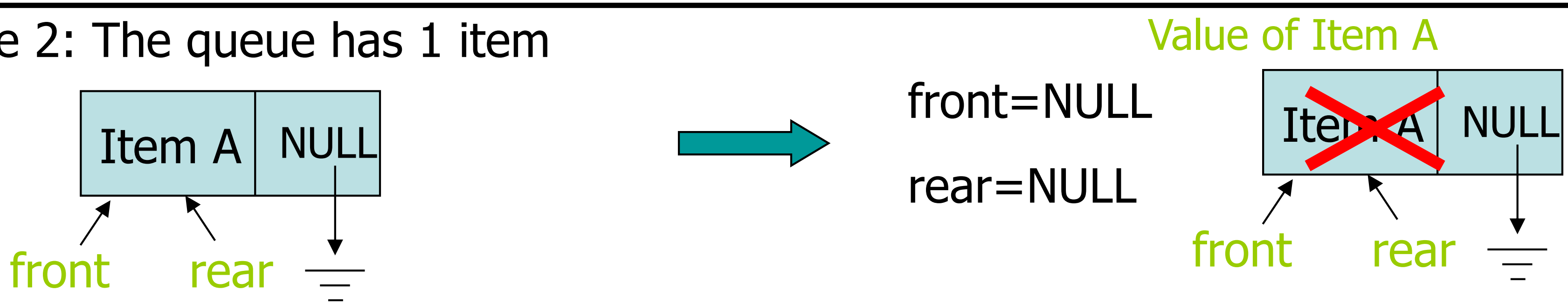
**To delete an item (the front item) and return it**

**We have 3 cases:**

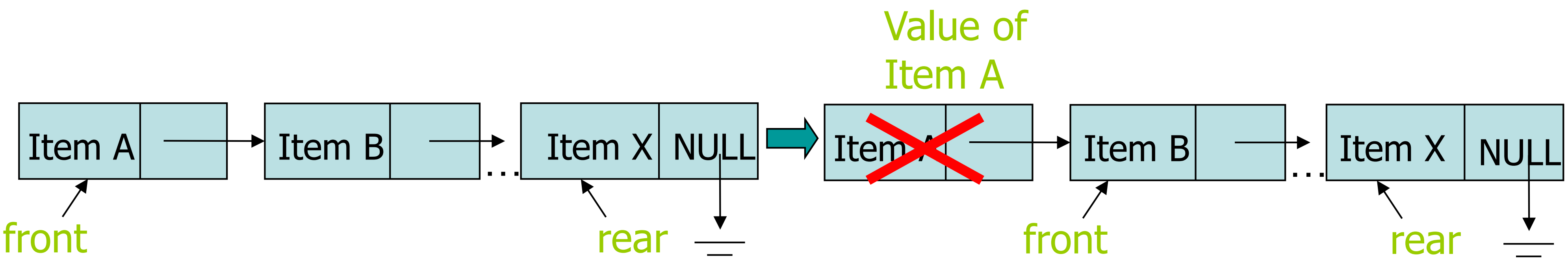
**The queue has 0 item, 1 item or more than one item.**

Case 1: The queue has 0 item → Output error

Case 2: The queue has 1 item




Case 3: The queue has more than one item



# Review: Linked Implementation of Queue

**To delete an item (the front item) and return it**

**We have 3 cases:**

**The queue has 0 item, 1 item or more than one item.**

```
// Queue.cpp

#include "Queue.h"
int Queue::Dequeue()
{
        int ret_value;
        if (!IsEmpty())
        {
                ret_value=front->data;
                front=front->next;
                if(front==NULL)
                        rear=NULL;
        }
        return ret_value;
}
```

# Review: Priority Queue

**Priority Queue**

- The elements in a stack or a FIFO queue are ordered based on the sequence in which they have been inserted.
- In a priority queue, the sequence in which elements are removed is based on the priority of the elements.

**Ordered Priority Queue**

| A<br>Priority=1 | B<br>Priority=2 | C<br>Priority=3 | D<br>Priority=3 |
|---|---|---|---|

(highest priority) (lowest priority)

The first element to be removed.

**Unordered Priority Queue**

| B<br>Priority=2 | C<br>Priority=3 | A<br>Priority=1 | D<br>Priority=3 |
|---|---|---|---|

# Review: Priority Queue

**Priority Queue - List Implementation**

- To implement a priority queue as an **ordered** list.

  Time complexity of the operations :

  (assume the sorting order is from highest priority to lowest)

  **Insertion**: Find the location of insertion. *O(n)*

  Link the element at the found location. *O(1)*

  Altogether: *O(n)*

  **Deletion**: The highest priority element is at the front.

  i.e., Remove the front element takes ***O(1)*** time

# Review: Priority Queue

**Priority Queue - List Implementation**

- To implement a priority queue as an **unordered** list.

  Time complexity of the operations :

  **Insertion**: Simply insert the item at the rear. *O(1)*

  **Deletion**: Traverse the entire list to find the maximum priority element. *O(n)*.

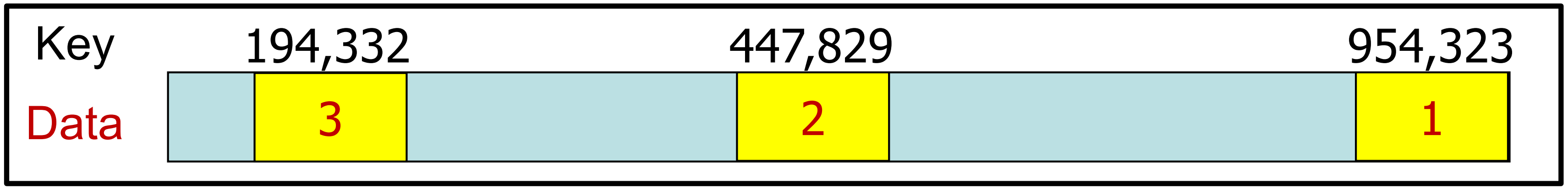
  Copy the value of the element to return it later. *O(1)*

  Delete the node. *O(1)*

  Altogether: *O(n)*

# Review: Sparse data

- How to store those data in the computer so that we can easily get the player's information by their keys?
  - Array:
    - A lot of memory space wasted




  - Linked List:

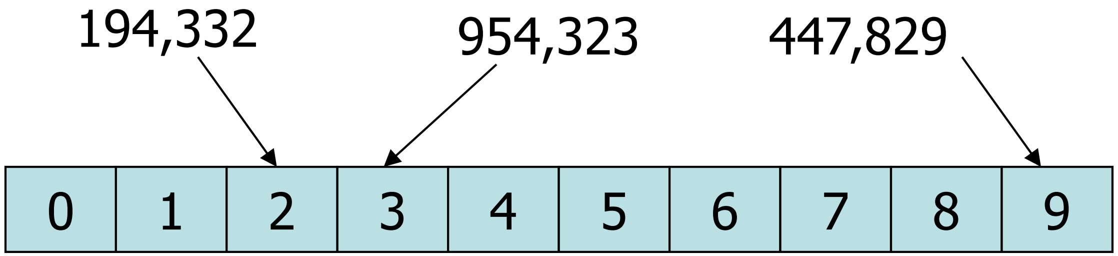
    Hard to search if we have 10,000 players

  - Hash Table

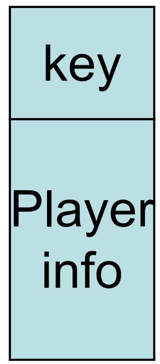
    Best solution in this case!

# Review: Basic Hash Table

- Advantages:
  - Quickly store sparse key-based data in a reasonable amount of space
  - Quickly determine if a certain key is within the table




slot




194,332%10=2 or 194,332≡2 (mod 10)

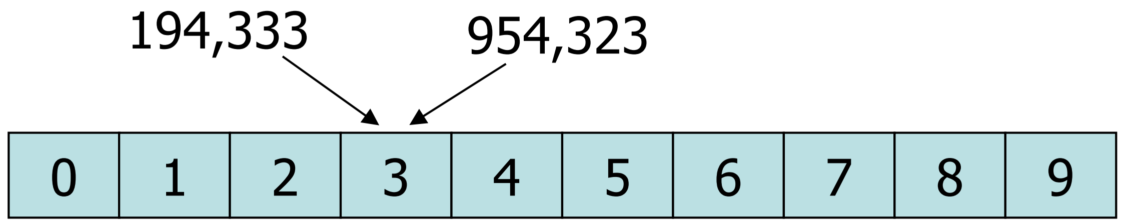
447,879%10=9 or 447,879≡9 (mod 10)

954,323%10=3 or 954,323≡3 (mod 10)

To get the information, we use:
player=table[key%10];

# Review: Collisions

- Two players mapped to the same cell




- Method to deal with collisions
  - Change the table
  - Hash functions
    - ‘Hash’ in the dictionary: chop (meat) into small pieces
    - Here, we ‘Hash’ numbers

# Review: Hash Functions

**Good hash function:**

Fast computation, Minimize collision

**Kinds of hash functions:**

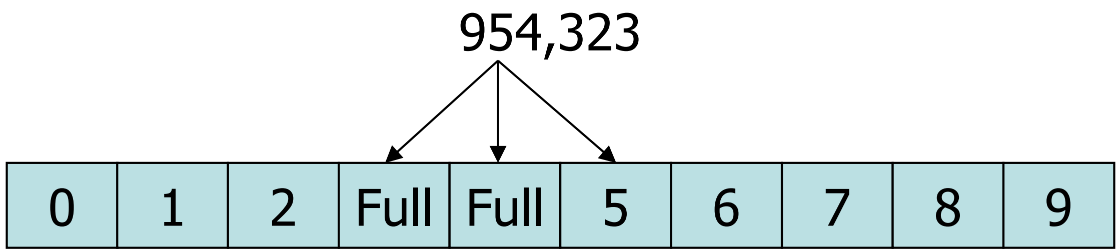
- **Division**: Slot_id = Key % table_size.
- **Others**: eg., Slot_id = (Key$^2$ + Key + 41) % table_size
- **table_size should better be a prime number.**

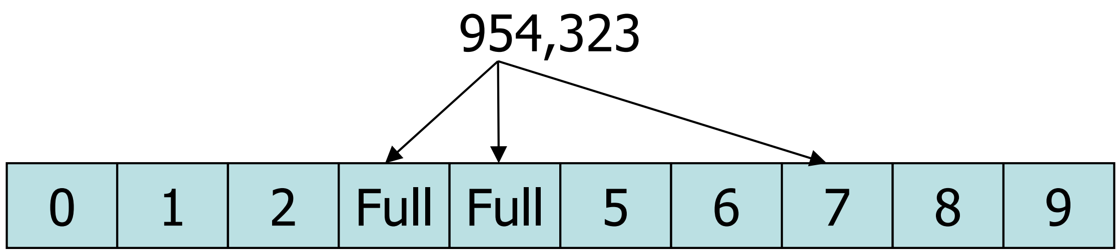
# Review: Combination of Hash Functions

- Collision is easy to happen if we use ***%*** function
- Combination:
  - Apply hash function $h_1$ on key to obtain *mid_key*
  - Apply hash function $h_2$ on *mid_key* to obtain *Slot_id*
- Example:
  - We apply %101 on 12320324111220 and get 79
  - We apply %10 on the result 79 obtained by %101
    - 79 % 10 =9

# Review: Collision Resolution - Open Addressing

- Linear Probing
  - ➢ If collide, try Slot_id+1, Slot_id+2
- Quadratic Probing
  - ➢ If collide, try Slot_id+1, Slot_id+4,…



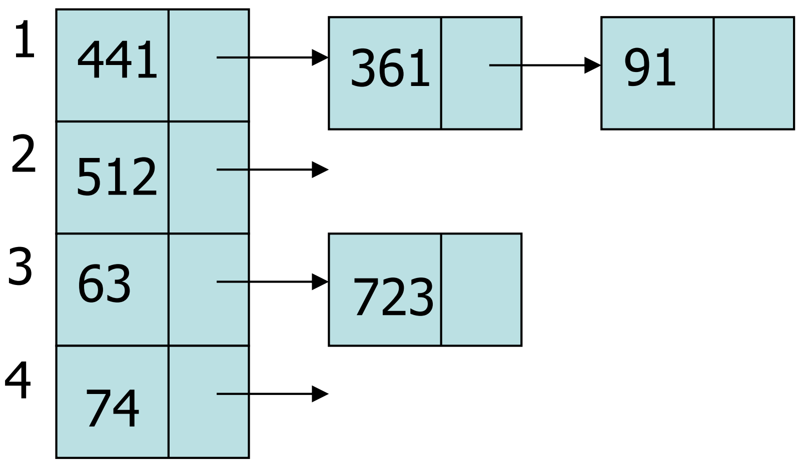



# Review: Collision Resolution - Open Addressing

- Linear Probing
  - If collide, try Slot_id+1, Slot_id+2
- Quadratic Probing
  - If collide, try Slot_id+1, Slot_id+4,…
- Double Hashing
  - If collide, try Slot_id+$h_2(x)$, Slot_id+$2h_2(x)$,… (prime size important)
- General rule: If collide, try other slots in a certain order
- How to find data?
  - If not found, try the next position according to different probing rule
  - Every key has a preference over all the positions
  - When finding them, just search in the order of their preferences

# Review: Collision Resolution - Separate Chaining

- Problems with Open Addressing?
- Using linked list to solve Collision
  - Every slot in the hash table is a linked list
  - Collision→Insert into the corresponding list
  - Find data→Search the corresponding list

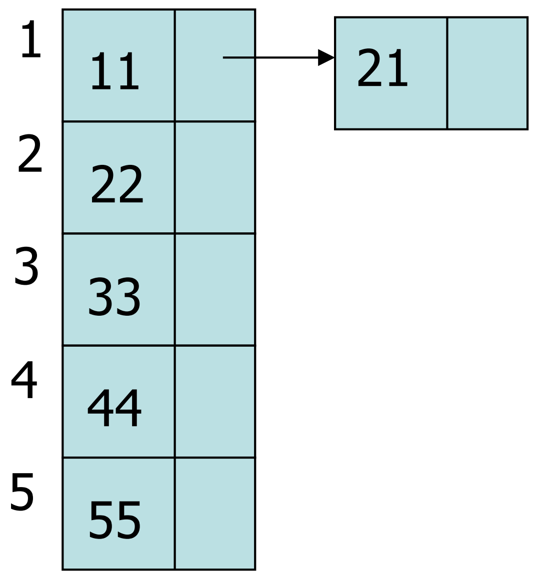


# Review: Collision Resolution

- Example: 11,22,33,44,55,66,77,88,99,21
  - ➢ Using linear probing

| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| 21 | 11 | 22 | 33 | 44 | 55 | 66 | 77 | 88 | 99 |

  - ➢ Using separate chaining



# Review: More on Hash Table Size

Table of prime size is important in the following case:

For quadratic probing, we have the following property:

> If quadratic probing is used and the table size is prime, then a new element can always be inserted if the table is at least half empty.

# Review: Rehashing

- Too many elements in the table
  →Too many collisions when inserting
- **Load factor** = number of slots occupied/total slots
- When half full, rehash all the elements into a double-size table
- In interactive systems, the user who triggers rehashing is unlucky
- In total, only O(n) cost incurred for a hash table of size n

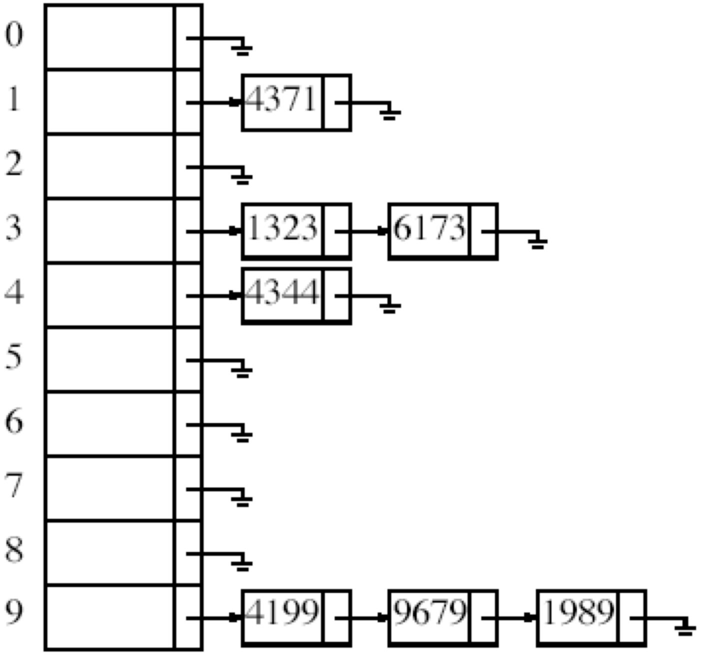
- Example: initial hash table size 2, when the size grows to 32, how many rehashes are done?
  - 2→4 1 number rehashed
  - 4→8 2 numbers rehashed
  - 8→16 4 numbers rehashed
  - 16→32 8 numbers rehashed
  - In total, 15 numbers rehashed, 15<16=32/2

# Exercise 1

Given input {4371, 1323, 6173, 4199, 4344, 9679, 1989} and a hash function $h(x) = x \pmod{()\ 10}$, show the resulting

a. separate chaining hash table
b. hash table using linear probing
c. hash table using quadratic probing
d. hash table with second hash function $h2(x) = 7 - (x \bmod 7)$

# Exercise 1




| | |
|---|---|
| 0 | 9679 |
| 1 | 4371 |
| 2 | 1989 |
| 3 | 1323 |
| 4 | 6173 |
| 5 | 4344 |
| 6 | |
| 7 | |
| 8 | |
| 9 | 4199 |

# Exercise 1

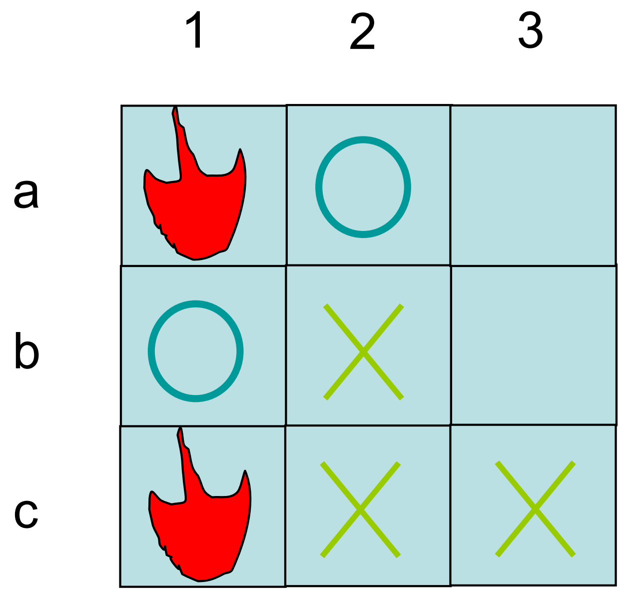
| | |
|---|---|
| 0 | 9679 |
| 1 | 4371 |
| 2 | |
| 3 | 1323 |
| 4 | 6173 |
| 5 | 4344 |
| 6 | |
| 7 | |
| 8 | 1989 |
| 9 | 4199 |

| | |
|---|---|
| 0 | |
| 1 | 4371 |
| 2 | |
| 3 | 1323 |
| 4 | 6173 |
| 5 | 9679 |
| 6 | |
| 7 | 4344 |
| 8 | |
| 9 | 4199 |

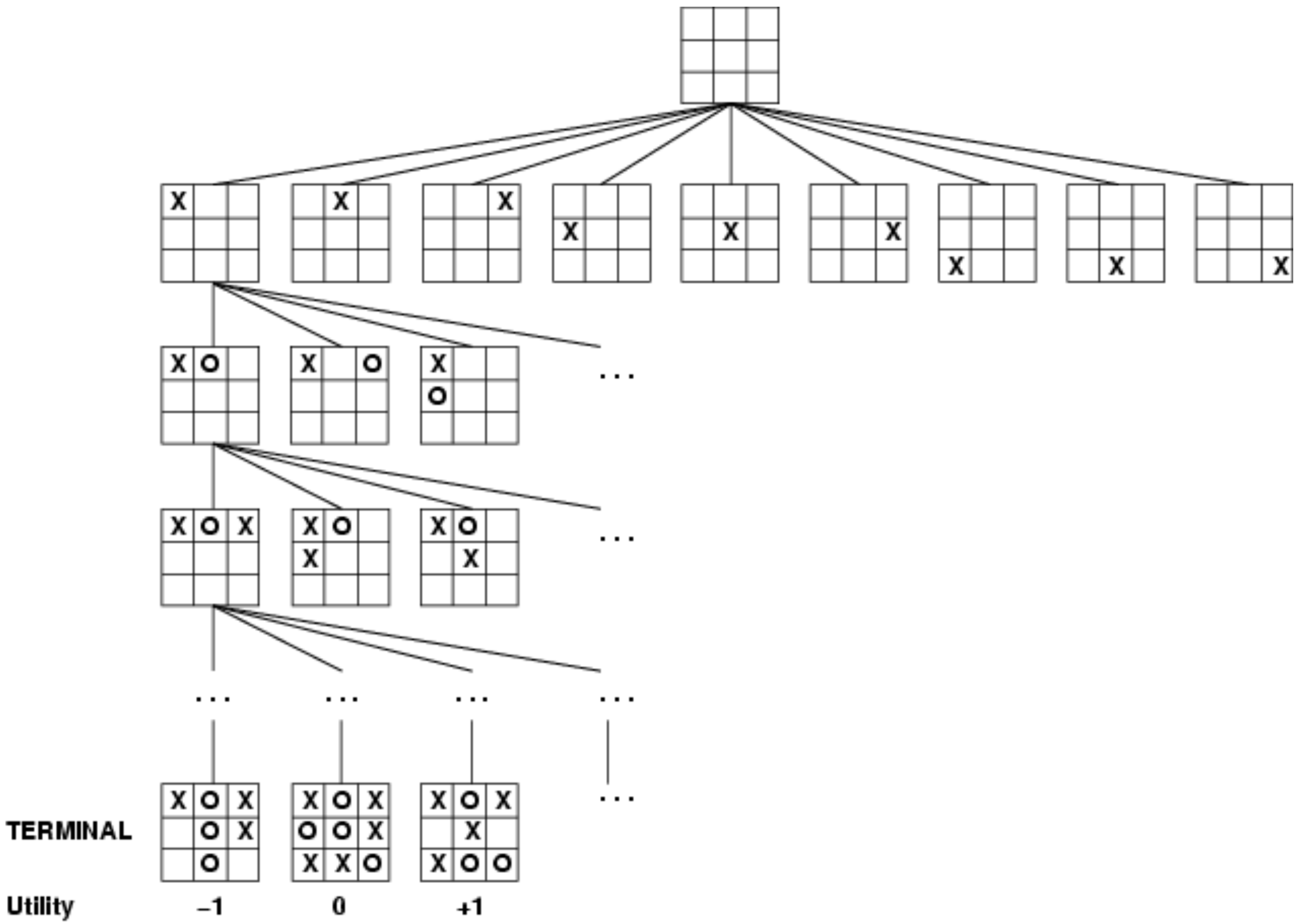
(1989 cannot find a proper slot)

# Tic Tac Toe

- How do you choose strategies when playing games?



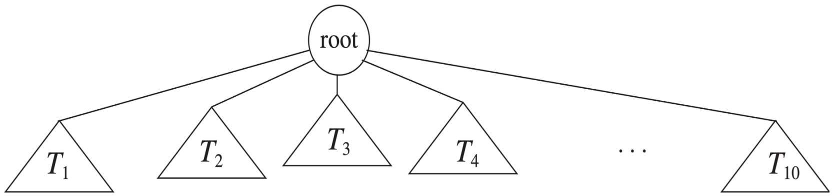

# Game Tree of Tic Tac Toe



# Objective

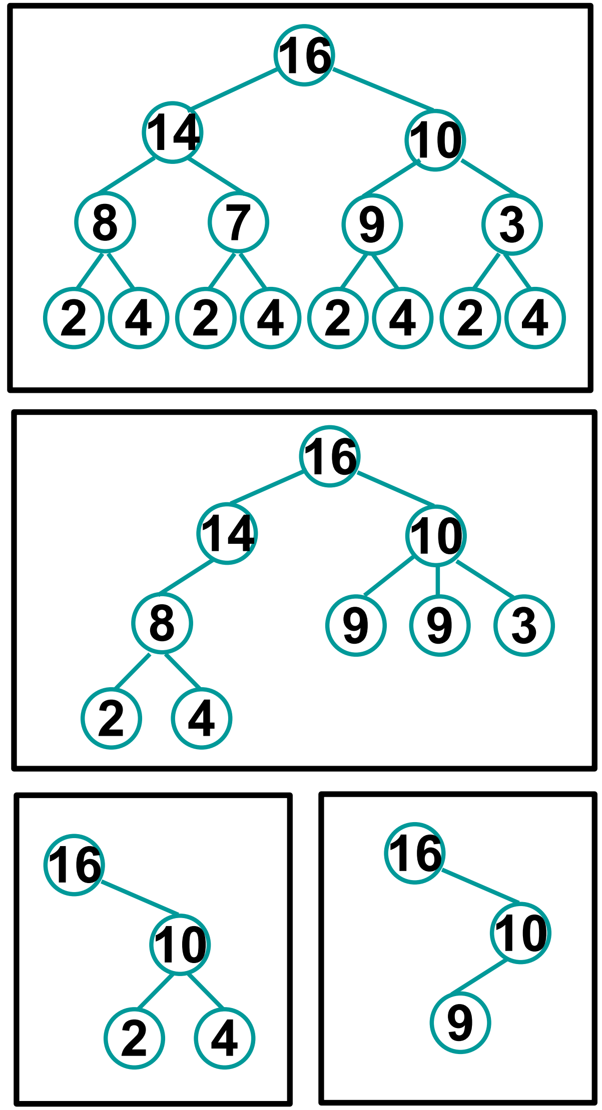
- Definition and Terminology
- Binary Tree
  - Operations
  - Recursive functions
  - Traversal
- Binary Search Tree
  - Insertion, deletion
- Binary Representation of General Tree

# Definition and Terminology




Tree is defined as a finite set *T* of one or more nodes such that:

- a) there is one specially designated node called **the root of the tree**, *root (**T**)* and
- b) the remaining nodes (excluding the root) are partitioned into *m* disjoint sets $\{T_1, T_2, \ldots, T_m\}$ and each of these sets in turn is a tree. The trees $\{T_1, T_2, \ldots, T_m\}$ are called **the subtrees of the root**.




4 examples

For subtrees $T_1, T_2, \ldots, T_m$, each of their roots are connected by a directed edge from the root node.

# Definition and Terminology

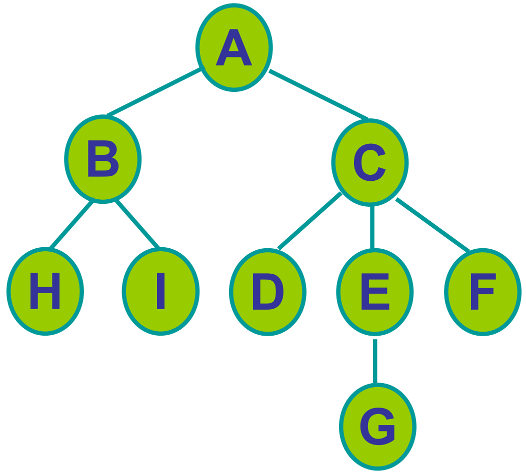
***Terminology:***

| | |
|---|---|
| **Degree** of a node | The number of subtrees of a node |
| **Terminal node** or **leaf** | A node of degree zero |
| **Branch node** or **internal node** | A nonterminal node |
| **Parent** and **Siblings** | Each node is said to be the parent of all roots of its subtrees, and the latter are said to be siblings; they are children of their parent. |
| **A Path from $n_1$ to $n_k$** | a sequence of nodes $n_1, n_2, \ldots n_k$ such that $n_i$ is the parent of $n_{i+1}$ for $0<i<k$. The length of this path is the number of edges on the path |
| **Ancestor** and **Descendant** | If there is a path from $n_1$ to $n_k$, we say $n_k$ is the descendant of $n_1$ and $n_1$ is the ancestor of $n_k$ . |
| **Level** or **Depth** of node | The length of the unique path from root to this node. |
| **Height** of a tree | The maximum level of any leaf in the tree. |

# Definition and Terminology

**Level** of node : State the levels of all the nodes:

A:____, B:____, C:____,
D:____, E:____, F:____,
G:____, H:____, I:____




**Root** of a tree: Root of the tree is: _____

**Height** of a tree: Height of the tree is: _____

**Degree** of a node : State the degrees of:

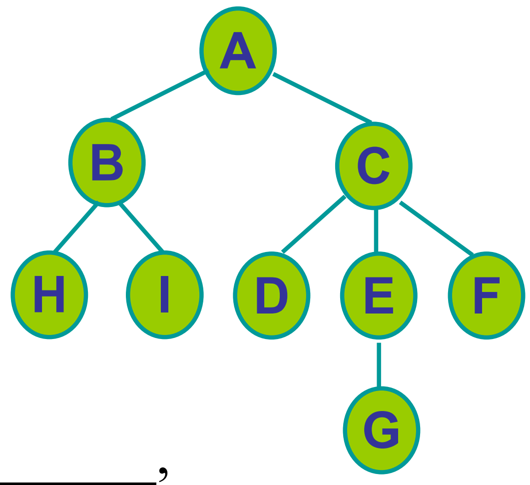
A:____, B:____, C:____,
D:____, E:____, F:____,
G:____, H:____, I:____

**Terminal node** or **leaf**: State all the leaf nodes: ____________________________

**Branch node**: State all the branch nodes: ____________________________

# Definition and Terminology

**Parent** and **Siblings**:




State the parents of: A:___, B:___, C:___, D:___, E:___, F:___, G:___, H:___, I:___

State the siblings of: A:________, B:________, C:________, D:________, E:________, F:________, G:________, H:________, I:________

**Ancestor** and **Descendant**:

State the ancestors of: A:_______, B:_______, C:_______, D:_____________, E:_____________, F:_____________, G:_____________, H:_____________, I:_____________

State the descendants of: A:______________________________, B:_____________, C:_____________, D:____, E:____, F:____, G:____, H:____, I:____

# Binary Tree

## *Definition:*

Binary tree can be defined as a finite set of nodes that either

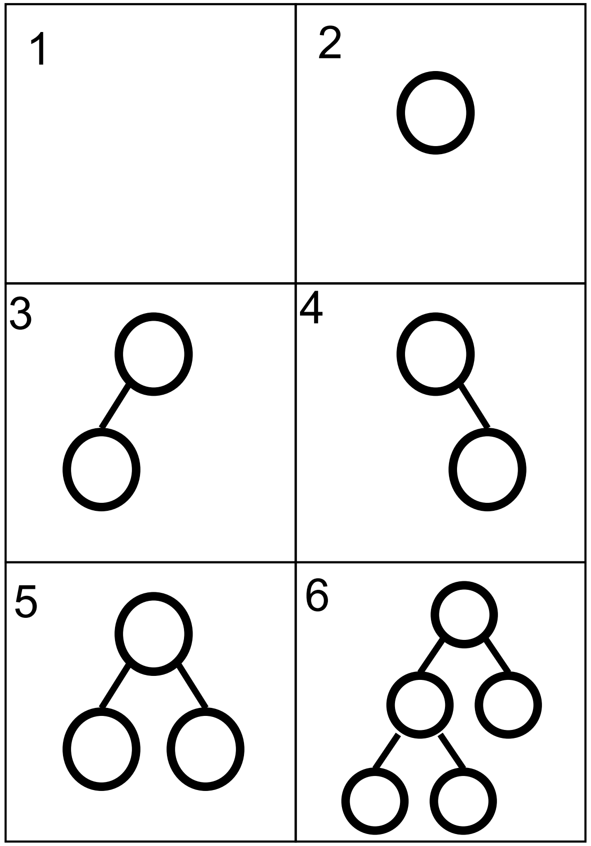
– is empty, or

– consists of

(1) a root, and

(2) the elements of 2 disjoint binary trees called the left and right subtrees of the root.

**6 Examples of Binary tree:**



# Binary Tree

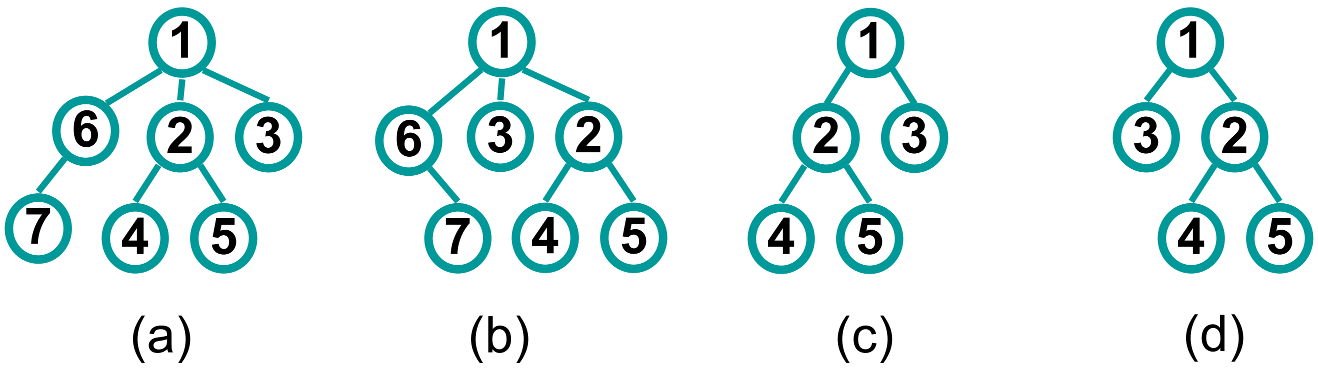
## *Comparison:*

***Tree***

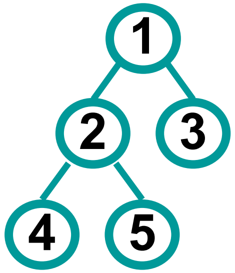
- A tree must have at least 1 node
- Each node has 0, 1, 2, .. or many subtrees.
- We don't distinguish subtrees according to their orders.

***Binary tree***

- A binary tree may be empty
- Each node has 0, 1, or 2 subtrees.
- We distinguish between the left and right subtree.



# Properties of Binary Tree

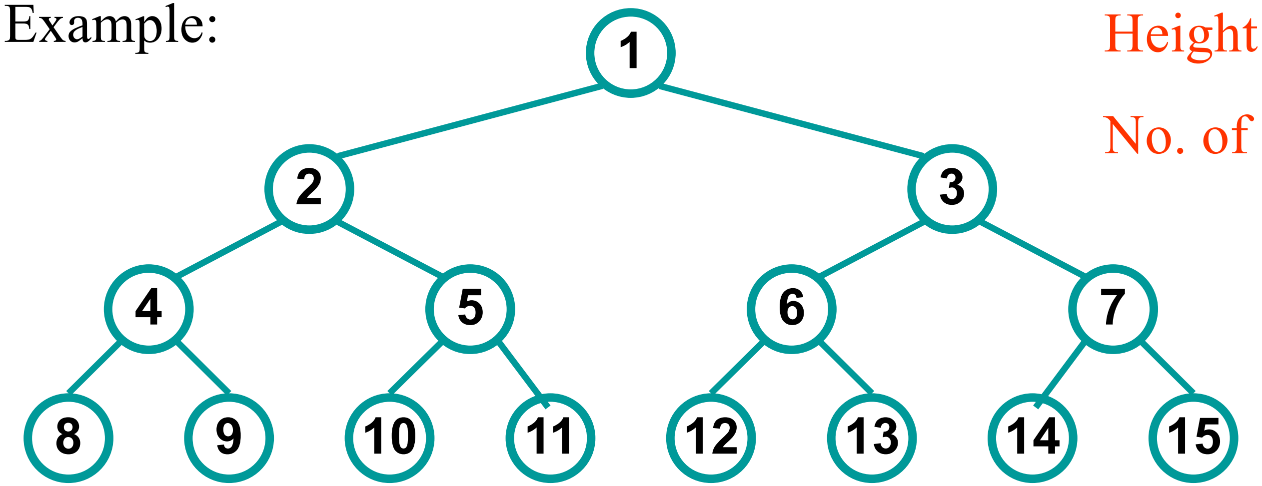



**Maximum number of nodes**

- Consider the levels of a binary tree: level 0, level 1, level 2, ..
- Maximum number of nodes on a level is $\underline{\mathbf{2}^{\textit{level\_id}}}$.
- Maximum number of nodes in a binary tree is $\underline{\mathbf{2}^{\textit{height\_of\_tree}+1} - \mathbf{1}}$.

**Full Binary Tree:** **No. of nodes = $2^{height\_of\_tree+1}-1$**

Example:



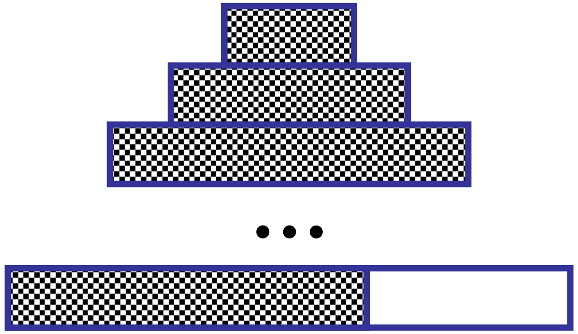

Height of tree = 3

No. of nodes $= 2^{height\_of\_tree+1}-1$

$= 15$

# Properties of Binary Tree

## Complete Binary Tree

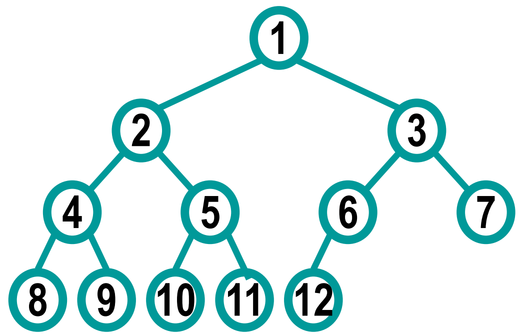


A complete binary tree is like a full binary tree,
But in a complete binary tree,

- Except the bottom level: all are fully filled.
- The bottom level: The filled slots are at the left of the empty slots (if any).

***Definition:*** A binary tree with ***n*** nodes and height ***k*** is **complete** if and only if its nodes correspond to the nodes numbered from 1 to ***n*** in the fully binary tree of height ***k***.

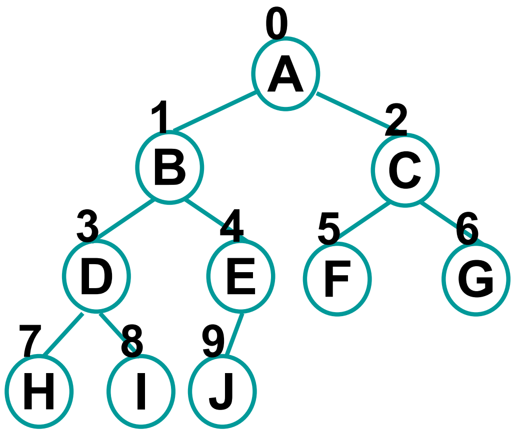
- Each leaf in a tree is either at level ***k*** or level ***k***-1
- Each node has exactly 2 subtrees at level 0 to level ***k***-2



# Array Representation of Binary Tree

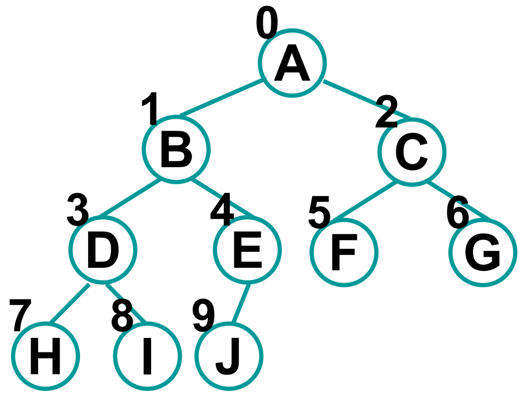
## Array Representation of Binary Tree

A numbering scheme:



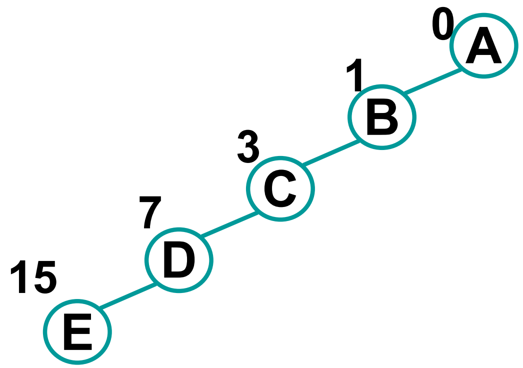

We can **represent binary trees using array** by applying this numbering scheme.

Example 1:



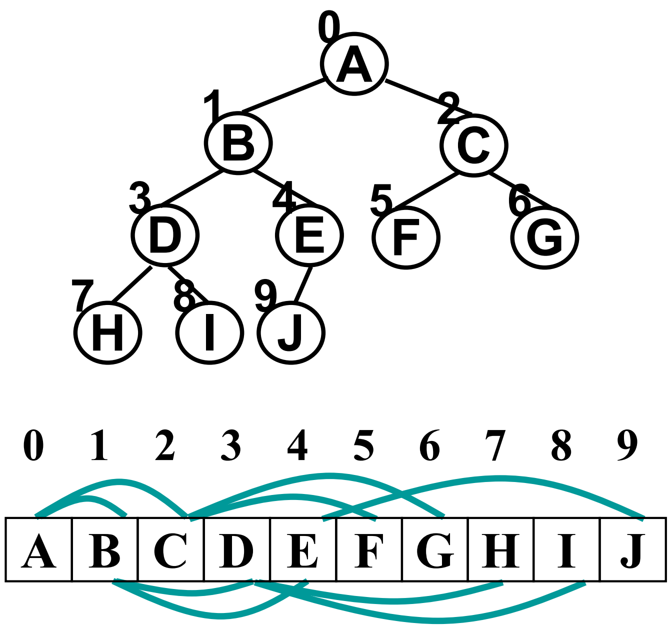

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|---|---|---|---|---|---|---|---|---|---|
| A | B | C | D | E | F | G | H | I | J |

Example 2:




| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | | | | | | | | | | | | | | |

# Array Representation of Binary Tree




Children of a node at slot ***i***:

Left(i) = 2i+1
Right(i) = 2i+2

Parent of a node at slot ***i***:

Parent(i) = $\lfloor (i-1)/2 \rfloor$

**$\lfloor x \rfloor$: "Floor" The greatest integer less than x**

**$\lceil x \rceil$: "Ceiling" The least integer greater than x**

For any slot ***i***,

If ***i*** is odd: it represents a left son.

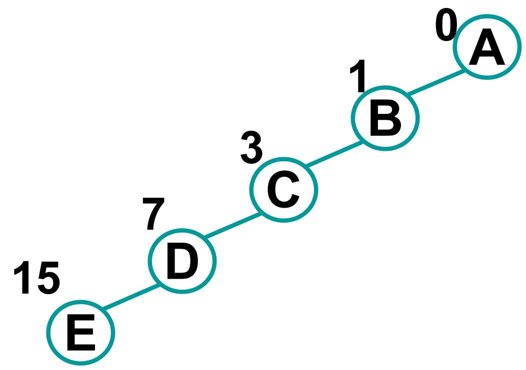
If ***i*** is even (but not zero): it represents a right son.

The node at the right of the represented node of ***i*** (if any), is at ***i***+1.

The node at the left of the represented node of ***i*** (if any), is at ***i***-1.

# Array Representation of Binary Tree

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| A | B | - | C | - | - | - | D | - | - | - | - | - | - | - | E |




**Unused** array elements (not exist or is NULL) must be flagged for non-full binary tree.

***Solutions***: 1. put a special value in the location
2. Add a "used" field (true/false) to each node.

***Advantages and Disadvantages of using array to represent binary tree:***

___ Simpler

___ Save storage for trees known to be almost full.

___ Waste of space (except complete binary tree)

___ Maximum size of the tree is fixed in advance

___ Inadequacy: insertion and deletion of nodes from the middle of a tree require the movement of potentially many nodes

# Array Representation of Binary Tree

- **Application: Find all duplicates in a list of numbers.**

**Method 1**: Compare each number with those before (or after) it.

e.g., to find all duplicates in <*7 4 5 9 5 8 3 3*>, we need to compare:

4 with 7

5 with 7,4

9 with 7,4,5

5 with 7,4,5,9

8 with 7,4,5,9,5

3 with 7,4,5,9,5,8

3 with 7,4,5,9,5,8,3

**Method 2**: Use a special binary tree (Binary Search Tree), T:

- Read number by number.
- Each time compare the number with the contents of T.
- If it is found duplicated, then output, otherwise add it to T.

# Array Representation of Binary Tree

## ❑ Application: Find all duplicates in a list of numbers.

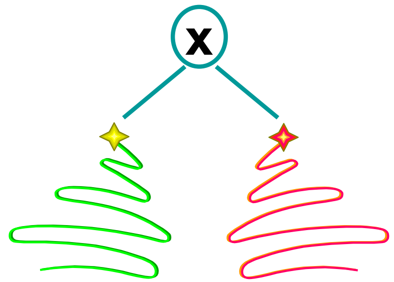
**Method 2: Use a special binary tree (Binary Search Tree), T**:

- Read number by number.
- Each time compare the number with the contents of **T**.
- If it is found duplicated, then output, otherwise add it to **T**.

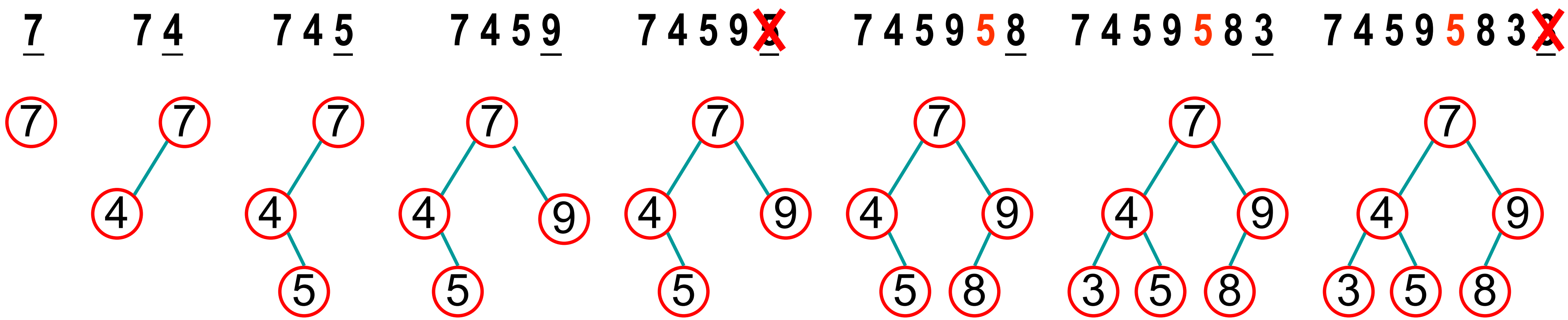
We'll discuss **Binary Search Tree** in a later topic.

Briefly, in a Binary Search Tree,

Each node of the tree contains a number.
The number of each node is
- bigger than the numbers in the left subtree.
- smaller than the numbers in the right subtree.




Example: To find all duplicates in < *7 4 5 9 5 8 3 3*>:

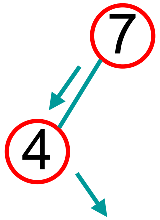


# Array Representation of Binary Tree

- Using Binary Search Tree to Find All Duplicates in a List of Numbers

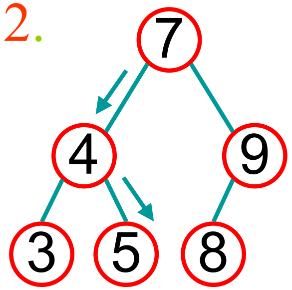
Indeed, searching and insertion are very quick in a binary search tree:

Example 1. The steps to insert a '5':



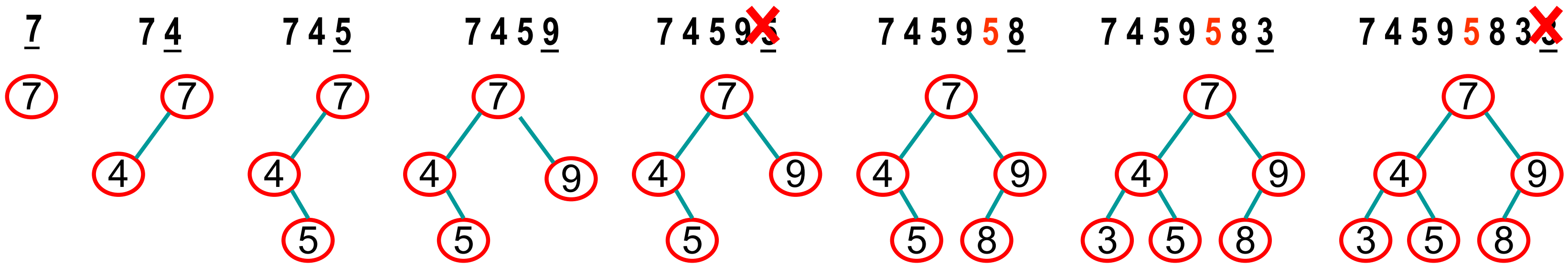

1. Compare '5' with the root. '5' is smaller than the root, so,
2. Go to left subtree, which has root = '4', '5' is larger than '4', so,
3. Go to the right subtree of '4', which is empty. => insert here.

Example 2.




The steps to insert a '5':

1. <same as example 1.>
2. <same as example 1.>
3. Go to the right subtree of '4', which has the root = '5'.
   Found=>no need to insert.



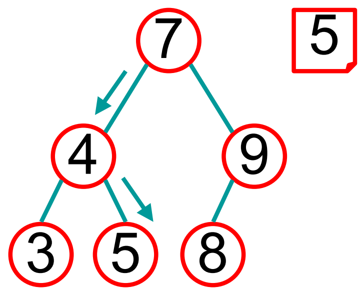

**No. of comparisons required:** 0 1 2 1 3 2 ___ ___

# Array Representation of Binary Tree

➢ Using Binary Search Tree to Find All Duplicates in a List of Numbers

**Method 2 - Use a special binary search tree (Binary Search Tree), T**:

- Read number by number.
- Each time compare the number with the contents of **T**.
- If it is found duplicated, then output, otherwise add it to **T**.




**<u>The algorithm:</u>**

Initialize an empty binary tree.

For each input number:

Traverse the tree from top to bottom.

For each node reached in traversal:

case 1: If the node is empty,
i.e., The input number is not found in the tree. ==> add it.

case 2: If the node content is same as the number,
output the input number as a duplicate.

case 3: If the node content is larger (smaller) than input number,
then prepare for traversal of its left (right) subtree.

The traversal ends due to case 1 or 2, or due to going beyond the array.
If it is due to going beyond the array, then output error message and exit.

# Array Representation of Binary Tree

- Using Binary Search Tree to Find All Duplicates in a List of Numbers

**Method 2 - Use a special binary search tree (Binary Search Tree), T**:

- Read number by number.
- Each time compare the number with the contents of **T**.
- If it is found duplicated, then output, otherwise add it to **T**.

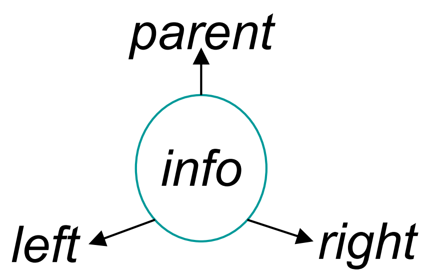
**Exercise:**

Create a binary search tree according to the input sequence:

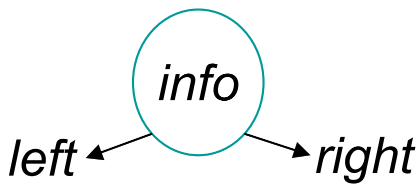
<*3, 5, 0, 2, 7, 9, 6, 8*>

# Linked Representation of Binary Tree

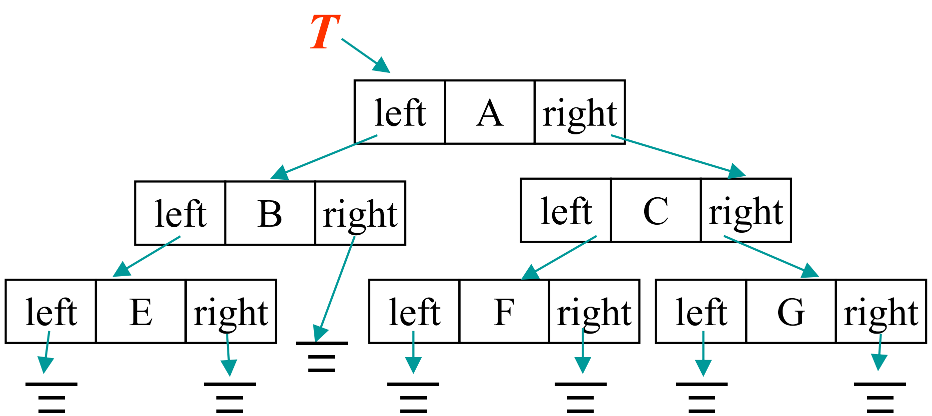
- Each node can contain *info, left, right, parent* fields
- where *left, right, parent* fields are node pointers pointing to the node's left son, right son, and parent, respectively.




- If the tree is always traversed in downward fashion (from root to leaves), the parent field is unnecessary.




```
class TreeNode
{
private:
        int info;
        TreeNode* left;
        TreeNode* right;
};
class Mytree
{
private:
        TreeNode* root;
}
```




- If the tree is empty, root = NULL; otherwise from root you can find all nodes.
- root->left and root->right point to the left and right subtrees of the root, respectively.

# Link Representation of Binary Tree

```
#include <stdlib.h>
#include <stdio.h>

class TreeNode
{
private:
        int info;
        TreeNode* left;
        TreeNode* right;
public:
        TreeNode();

        //create a new left child of a given node
        void SetLeft(int value) {..}

        //create a new right child of a given node
        void SetRight(int value) {..}

        void Insert(int );
}
```

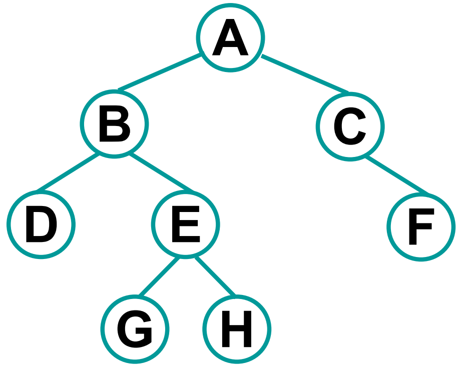
```
class Mytree
{
private:
        TreeNode* root;
public:
        Tree();
        GetHeight();
        Compare(Mytree*);
        void InsertNode(int )
        void PreorderTraversal();
        void PreorderHelper(TreeNode*);
        void InorderTraversal();
        void InorderHelper(TreeNode*);
        void PostorderTraversal();
        void PostorderHelper(TreeNode*);
}
```

# Binary Tree Operations - height

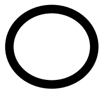
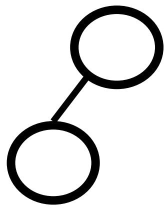
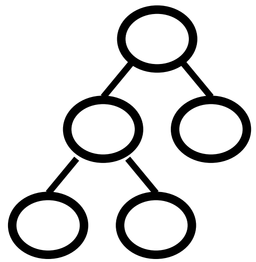
**Review:**

**Depth** of node : The depth of root(T) is zero.
The depth of any other node is one larger than his parent's depth.

**Height** of a tree: The maximum depth of any leaf in the tree.



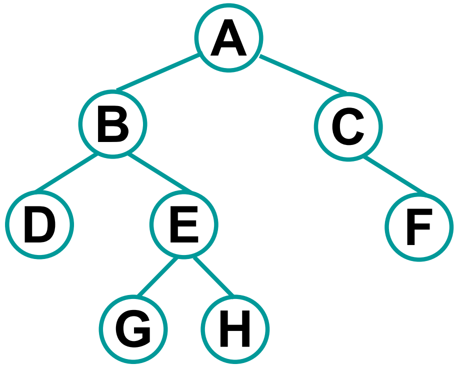

Example:

| | | | |
|---|---|---|---|
| Height of a NULL binary tree is -1 or 0 |  Height of a tree with 1 node is 0 |  Height = 1 |  Height = 2 |

# Binary Tree Operations - height

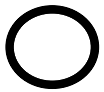
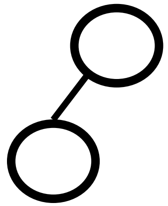
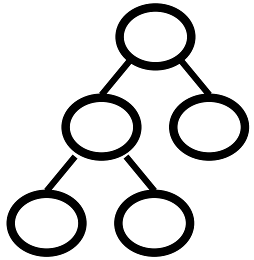
```
//To determine the height of a binary tree
int Mytree::height() {
        return root->height();
}
```

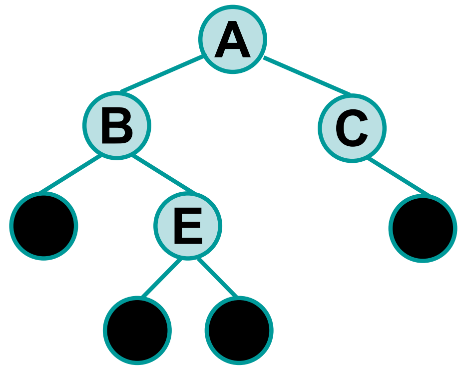
```
int TreeNode::height( )
{       int HeightOfLeftSubTree, HeightOfRightSubTree;
        if (this == NULL)
                return(0);
        if ((this->left == NULL) && (this->right == NULL))
                return(0); // the subroot is at level 0
        HeightOfLeftSubTree = this->left->height();
        HeightOfRightSubTree = this->right->height();
        if (______________________________________)
                return___________________________;
        else
                return___________________________;
}
```



# Binary Tree Operations - countleaves

Example:

| | | | |
|---|---|---|---|
| A NULL binary tree has 0 leaf node |  A tree with 1 node has 1 leaf node |  No. of leaf nodes = 1 |  No. of leaf nodes = 3 |

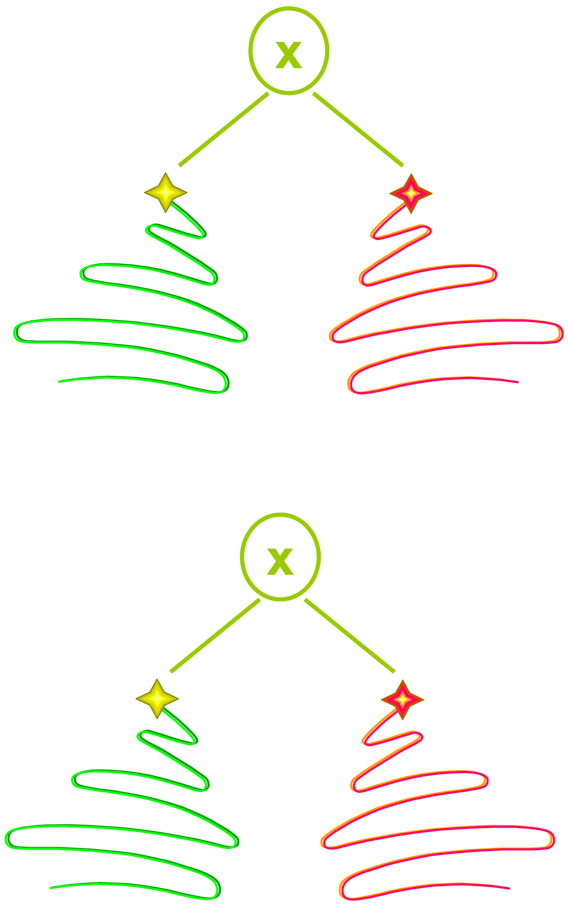



```
//To count the number of leaf nodes
int Mytree::count_leaf(TreeNode* p)
{
	if (p == NULL)
		return(0);
	else if ((p->left == NULL) && (p->right == NULL))
		return(1);
	else
		return(count_leaf(p->left) + count_leaf(p->right));
}
```

# Binary Tree Operations - equal

```
// To compare 2 binary trees
bool Mytree::equal(Mytree* T)
{
        return root->equal(T->root);
}
```

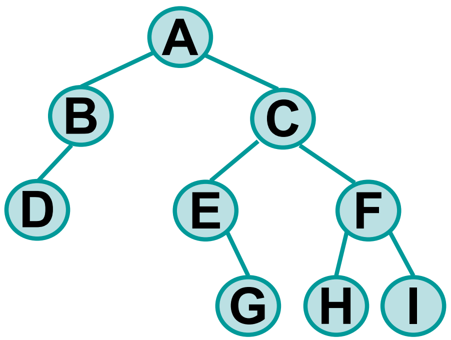
```
bool TreeNode::equal(TreeNode* TN)
{       if ((this == NULL) && (TN == NULL))
                return(true) ;

        if ((this != NULL) && (TN == NULL))
                return(false);
        if ((TN != NULL) && (this == NULL))
                return(false);

        if (this->info == TN->info)
                if (this->left->equal(TN->left) &&
                        this->right->equal(TN->right))
                           return(true);

        return(false);
}
```



# Traversing Binary Tree

**Traversing / walking through**

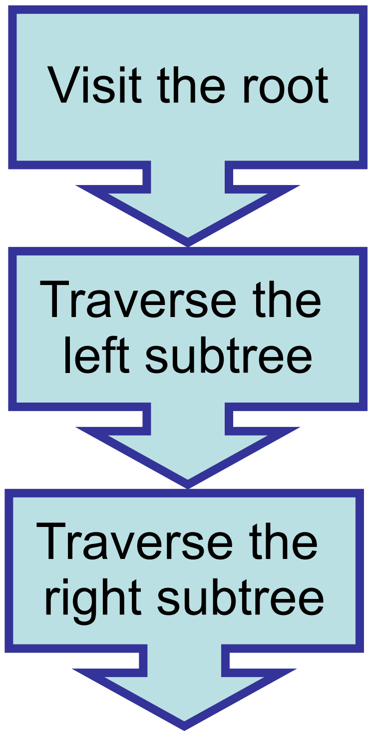
A method of examining the nodes of the tree systematically so that each node is visited exactly once.




**Three principle ways:**

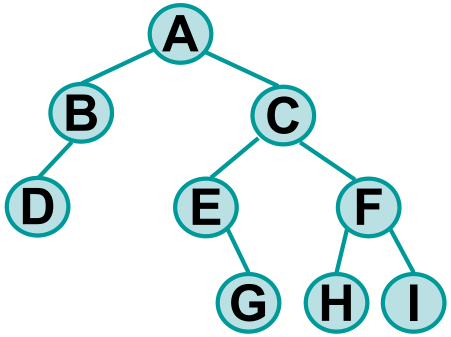
When the binary tree is empty, it is “traversed” by doing nothing, otherwise:

**preorder traversal** **inorder traversal** **postorder traversal**



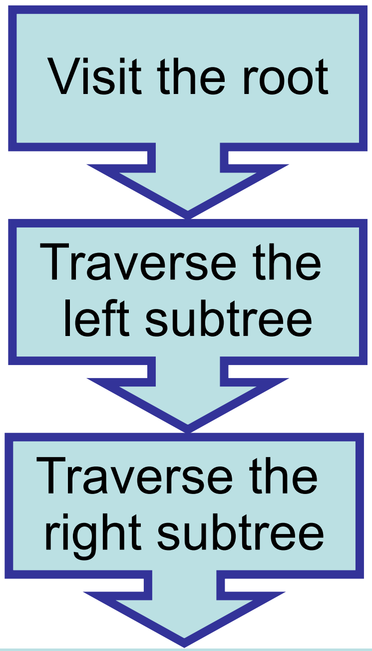

**A B D C E G F H I**

# Traversing Binary Tree




When the binary tree is empty, it is "traversed" by doing nothing, otherwise:

**preorder traversal**




**A B D C E G F H I**

Result:

= A (A's left) (A's right)

= A B **(B's left)** (B's right = NULL) (A's right)

= A B **(B's left)** (A's right)

= A B D (D's left=NULL) (D's right = NULL) (A's right)

= A B D (A's right)

= A B D C **(C's left)** (C's right)

= A B D C **E (E's left=NULL) (E's right)** (C's right)

= A B D C **E (E's right)** (C's right)

= A B D C **E G (G's left=NULL) (G's right = NULL)** (C's right)

= A B D C **E G** (C's right)

= A B D C **E G** F (F's left) (F's right)

= A B D C **E G** F H (H's left=NULL) (H's right =NULL) (F's right)

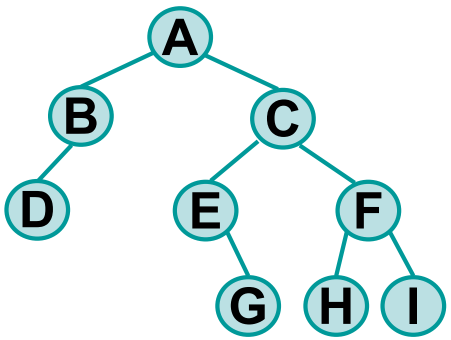
= A B D C **E G** F H I (I's left=NULL) (I's right =NULL)

= A B D C **E G** F H I

# Traversing Binary Tree

**Traversing / walking through**

A method of examining the nodes of the tree systematically so that each node is visited exactly once.




**Three principle ways:**

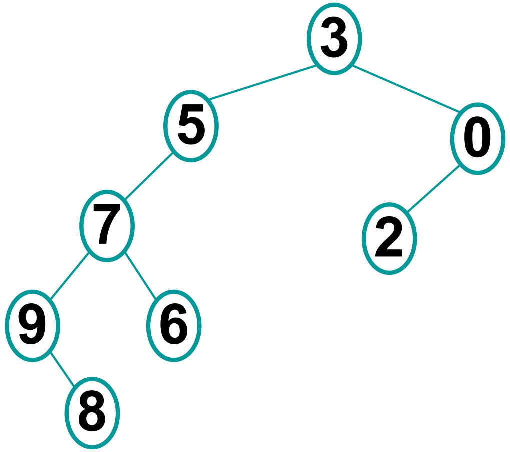
When the binary tree is empty, it is "traversed" by doing nothing, otherwise:

| preorder traversal | inorder traversal | postorder traversal |
|---|---|---|
| Visit the root | Traverse the left subtree | Traverse the left subtree |
| Traverse the left subtree | Visit the root | Traverse the right subtree |
| Traverse the right subtree | Traverse the right subtree | Visit the root |
| **A B D C E G F H I** | **D B A E G C H F I** | **D B G E H I F C A** |

# Traversing Binary Tree

**Exercise:**

1. Examine the preorder, inorder and postorder traversals of the tree:




preorder:

inorder:

postorder:

# Traversing Binary Tree

## Recursive Implementation

```
void Mytree::PreorderTraversal()
{
        PreorderHelper(root);
}
void Mytree::PreorderHelper(TreeNode* node)
{
        if (node!= NULL)
        {
                // visit the root
                cout << node->info;
                PreorderHelper(node->left);    // traverse left subtree
                PreorderHelper(node->right);  // traverse right subtree
        }
}
```

# Traversing Binary Tree

## Non-recursive inorder traversal using a stack

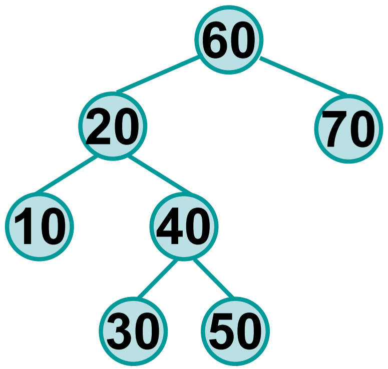
Traversal order: Left subtree => root => right subtree

1. Start with an empty stack, that will store all branch nodes that have been "reached" but itself and its right child pointer are not yet "visited".
2. Use a pointer p to traverse the tree starting from the root.
3. Upon reaching any node, save the address of the node in the stack (so that the node and the node's right child pointer will be traversed later) and then traverse following the left child pointer.
4. Upon reaching any NULL address, pop one from the stack. The popped one is the most recent one waited to be visited, so visit it, then traverse following its right child pointer.
5. The process continues until no more link to follow and no more can be popped (in 4.)

# Traversing Binary Tree

## Non-recursive inorder traversal using a stack

Traversal order: Left subtree => root => right subtree




| curPtr | ->60 | ->20 | ->10 | NULL | ->10 | ->10 | NULL | ->20 | ->20 |
|---|---|---|---|---|---|---|---|---|---|
| Stack | | | ->10 | ->10 | ->10 | | | | |
| | | ->20 | ->20 | ->20 | ->20 | Visit 10 | ->20 | ->20 | Visit 20 |
| | ->60 | ->60 | ->60 | ->60 | ->60 | | ->60 | ->60 | |

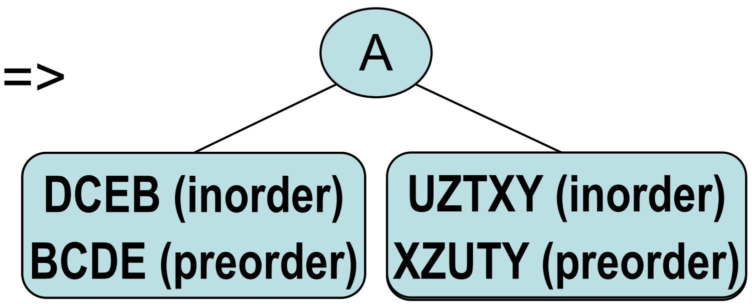
# Traversing Binary Tree

| | |
|---|---|
| **Reconstruction of Binary Tree from its preorder and Inorder sequences** | **Example:** Given the following sequences, find the corresponding binary tree:<br>preorder : ABCDEXZUTY<br>inorder : DCEBAUZTXY |

**Looking at the whole tree:**

- “preorder : **A**BCDEXZUTY”
  ==> A is the root.
- Then, “inorder : DCEB**<u>A</u>**UZTXY”

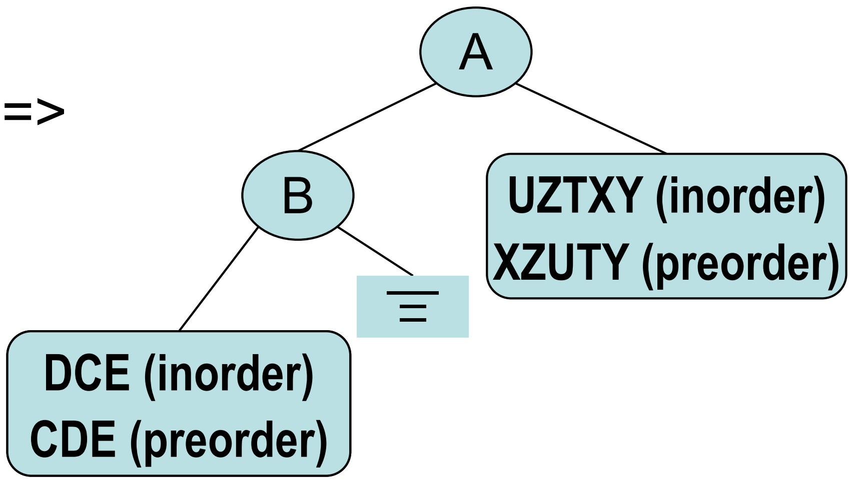
==>




**Looking at the left subtree of A:**

- “preorder : BCDE”
  ==> B is the root
- Then, “inorder: DCE**<u>B</u>**”

=>



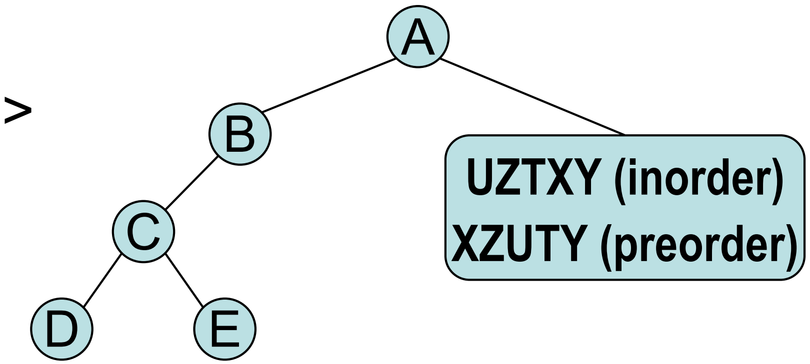
# Traversing Binary Tree

| **Reconstruction of Binary Tree from its preorder and Inorder sequences** | **Example:** Given the following sequences, find the corresponding binary tree:<br>preorder : ABCDEXZUTY<br>inorder : DCEBAUZTXY |
| --- | --- |

**Looking at the left subtree of B:**

- “preorder : CDE”
  ==> C is the root
- Then, “inorder: D<u>**C**</u>E”

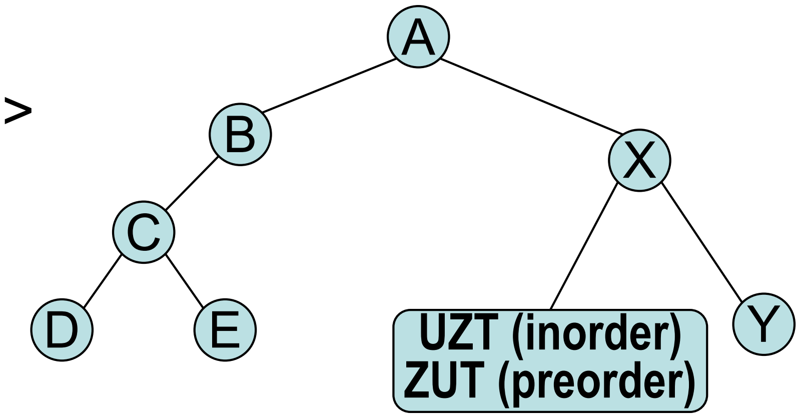
=>




**Looking at the right subtree of A:**

- “preorder : XZUTY”
  ==> X is the root
- Then, “inorder: UZT<u>**X**</u>Y”

=>



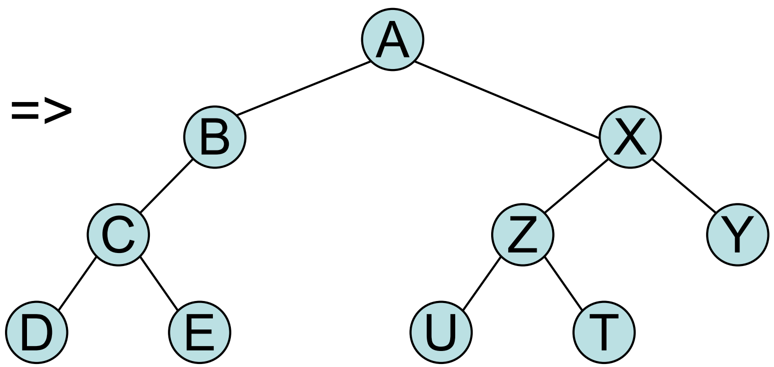
# Traversing Binary Tree

| | | |
|---|---|---|
| **Reconstruction of Binary Tree from its preorder and Inorder sequences** | **Example:** | Given the following sequences, find the corresponding binary tree:<br>preorder : ABCDEXZUTY<br>inorder : DCEBAUZTXY |

**Looking at the left subtree of X:**

- "preorder : ZUT"
  ==> Z is the root
- Then, "inorder: U**Z**T"

=>



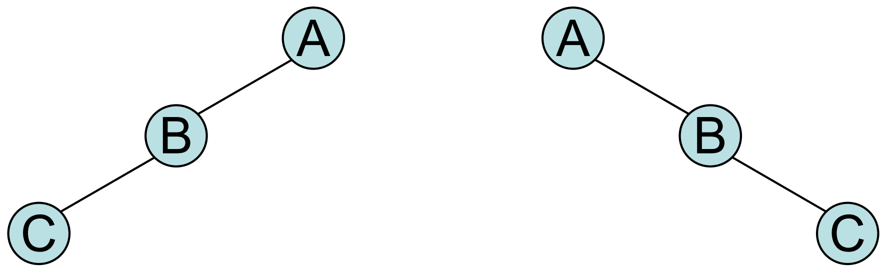
# Traversing Binary Tree

**But:** A binary tree may not be uniquely defined by its preorder and postorder sequences.

**Example:** **Preorder sequence: ABC**

**Postorder sequence: CBA**

We can construct 2 different binary trees:



# Learning Objectives

1. Explain the concept of Tree
2. Able to insert into and delete from a binary search tree
3. Able to do Tree Traversal; Able to reconstruct a tree given two suitable traversal orders
4. Able to write recursive functions on Tree

D:1;    C:1,2;    B:1,2,3;    A:1,2,3,4