CX1107 Data Structures and Algorithms

Trees

College of Engineering

School of Computer Science and Engineering

Dr. Loke Yuan Ren

Lecturer

yrloke@ntu.edu.sg

So Far ...

Dynamic Memory Management

- **`#include <stdlib.h>`**
- **`malloc()`**
- **`free()`**

```
struct _listnode
{
    int item;
    struct _listnode *next;
};
typedef struct _listnode ListNode;
```

1. Display: **`printList()`**
2. Search: **`findNode()`**
3. Insert: **`insertNode()`**
4. Delete: **`removeNode()`**
5. Size: **`sizeList()`**

Linked List vs Array

1. **Display: Both are similar**
2. **Search: Array is better**
3. **Insert and Delete: Linked List is more flexible**
4. **Size: Array is better**

1. Display: printList()
2. Search: findNode()
3. Insert: insertNode()
4. Delete: removeNode()
5. Size: sizeList()

…

```
void printList(ListNode *cur){
    while (cur != NULL){
        printf("%d\n", cur->item);
        cur = cur->next;
    }
}
```

```
int sizeList(ListNode *head){
    int count = 0;
    while (head != NULL){
      count++;
      head = head->next;
    }
    return count;
}
```

```
ListNode *findNode(ListNode* cur, int i){
   if (cur==NULL || i<0)
      return NULL;
   while(i>0){
      cur=cur->next;
      if (cur==NULL)
         return NULL;
      i--;
   }
   return cur;
}
```

```
int insertNode(ListNode **ptrHead, int i, int item){
    ListNode  *pre, *newNode;
    if (i == 0){
        newNode = malloc(sizeof(ListNode));
        newNode->item = item;
        newNode->next = *ptrHead;
        *ptrHead = newNode;
        return 1;
    }
    else if ((pre = findNode(*ptrHead, i-1)) != NULL){
        newNode = malloc(sizeof(ListNode));
        newNode->item = item;
        newNode->next = pre->next;
        pre->next = newNode;
        return 1;
    }
    return 0;
}
```

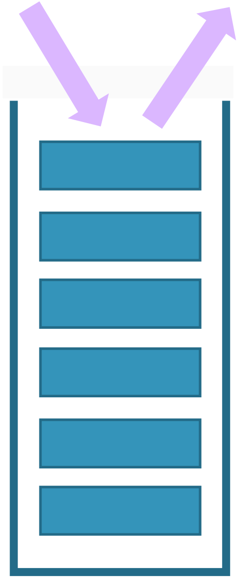
Stacks and Queues

```
typedef struct _linkedlist{
     int size;
     ListNode *head;
} LinkedList;
```

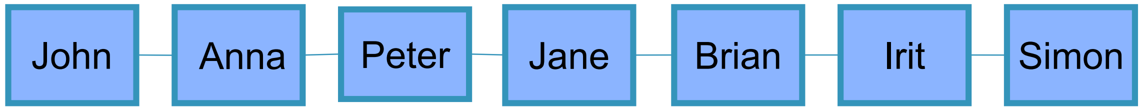
1. **Variations of the linked list**
   - **Doubly Linked List**
   - **Circular Linked List**
   - **Circular Doubly Linked List**
2. **Stacks and Queues**

`All these dynamic data structures are linear data structures`

LIFO

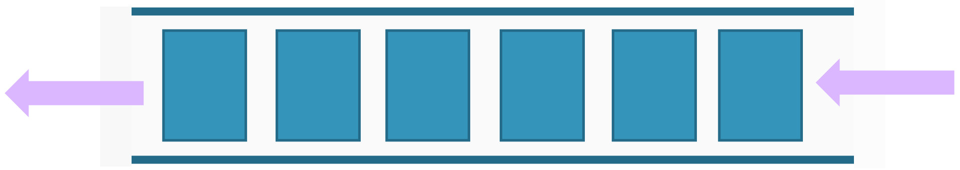


Stack



Linked List

Queue FIFO



```
typedef ListNode StackNode;

typedef LinkedList Stack;
```

Stack

1. Retrieve: `peek()`
2. Insert: `push()`
3. Delete: `pop()`
4. Size: `isEmptyStack()`

```
typedef ListNode QueueNode;
typedef struct _queue{
   int size;
   ListNode *head;
   ListNode *tail;
} Queue;
```

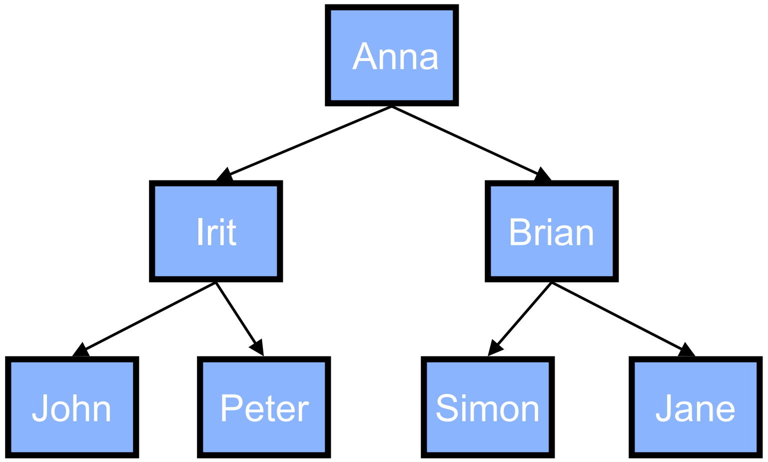
Queue

1. Retrieve: `getFront()`
2. Insert: `enqueue()`
3. Delete: `dequeue()`
4. Size: `isEmptyQueue()`

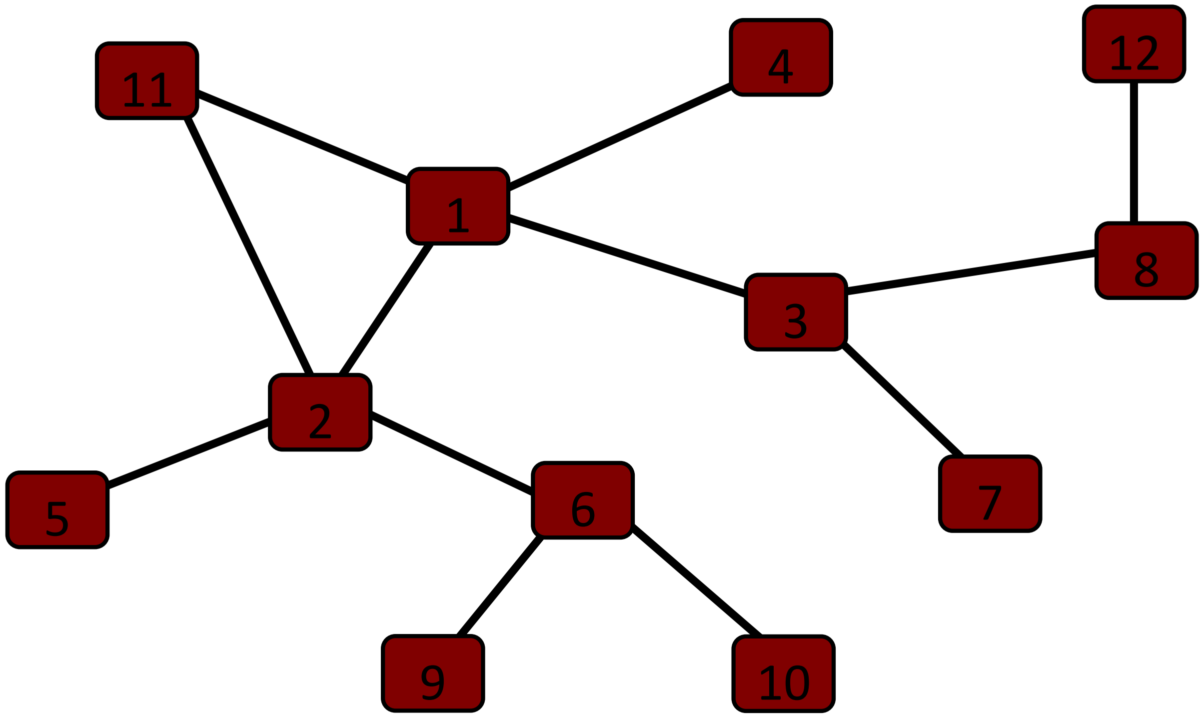
Overview

1. **What are trees?**
2. **Why do you need a tree?**
3. **How to create a tree?**
4. **How to use the tree?**

What are trees?

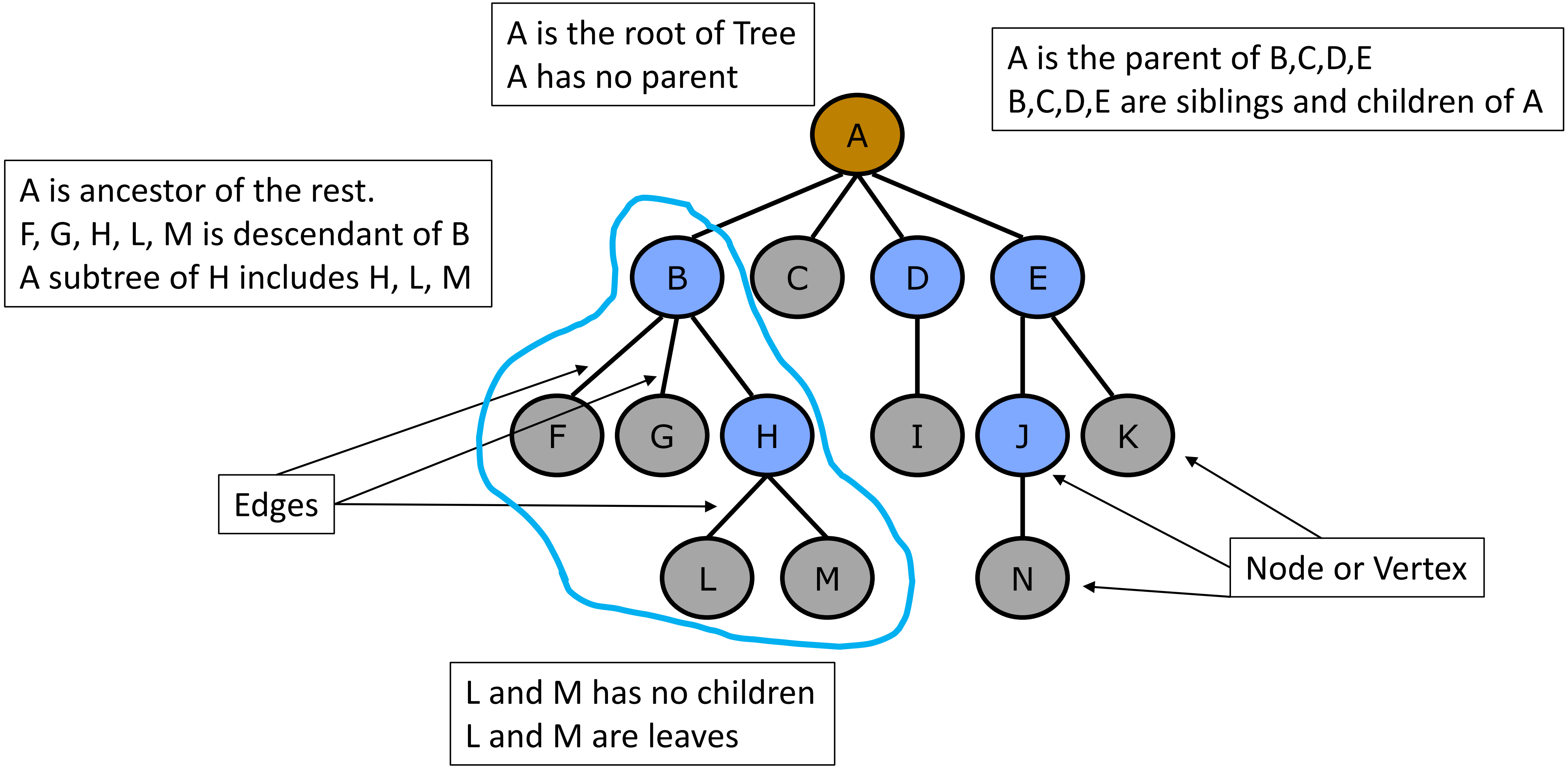


Tree



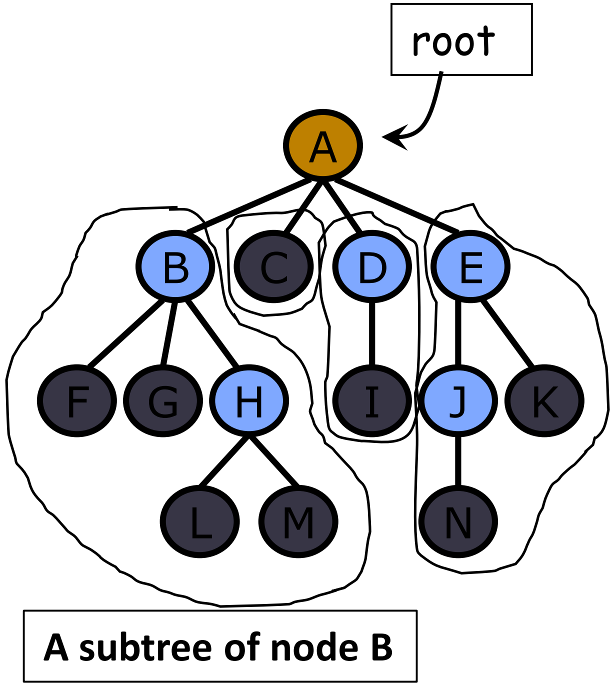
Graph

Terminology In Tree



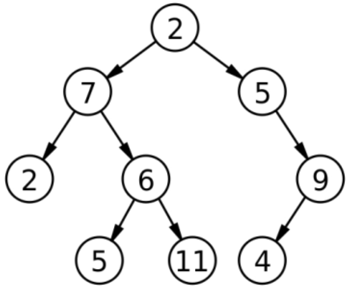
Tree Data Structure

- Similar to family tree concept
- One special node: root
- Each node can has many children
  A has four children: B, C, D, E
- Each node (except the root) has a parent node
- **A is the parent of B, C, D, E**
- Other children of your parent are your siblings
- **B, C, D and E are siblings**
- **Subtree**: Any node in the tree together with all of its descendants for a subtree.

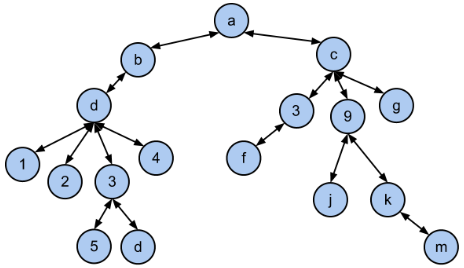


Tree Data Structure

- Tree data structure looks like… a tree: root, branches, leaves
  - Only one root node which has no parent
  - Each node branches out to some number of nodes
    - For binary tree, each node has up to two children (left and right child)
  - Each node has only one “parent” node – the node pointing to it (except the root node)



Binary Tree

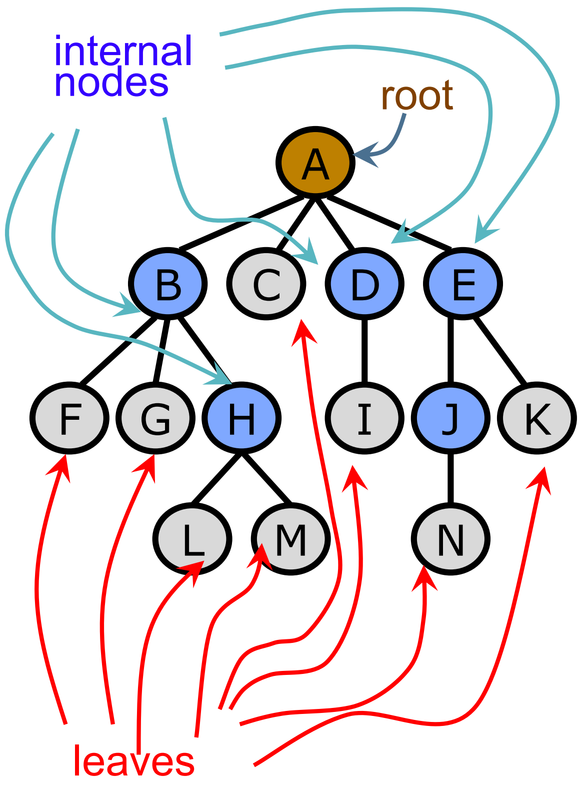


General Tree

- General tree
  - Each node can have links to any number of other nodes

Tree Data Structure

- A tree is composed of nodes
- Types of nodes
  - Root: only one in a tree, has no parent.
  - Internal node(non-leaf): Nodes with children are called internal nodes
  - Leaf (External Node): nodes without children are called leaves



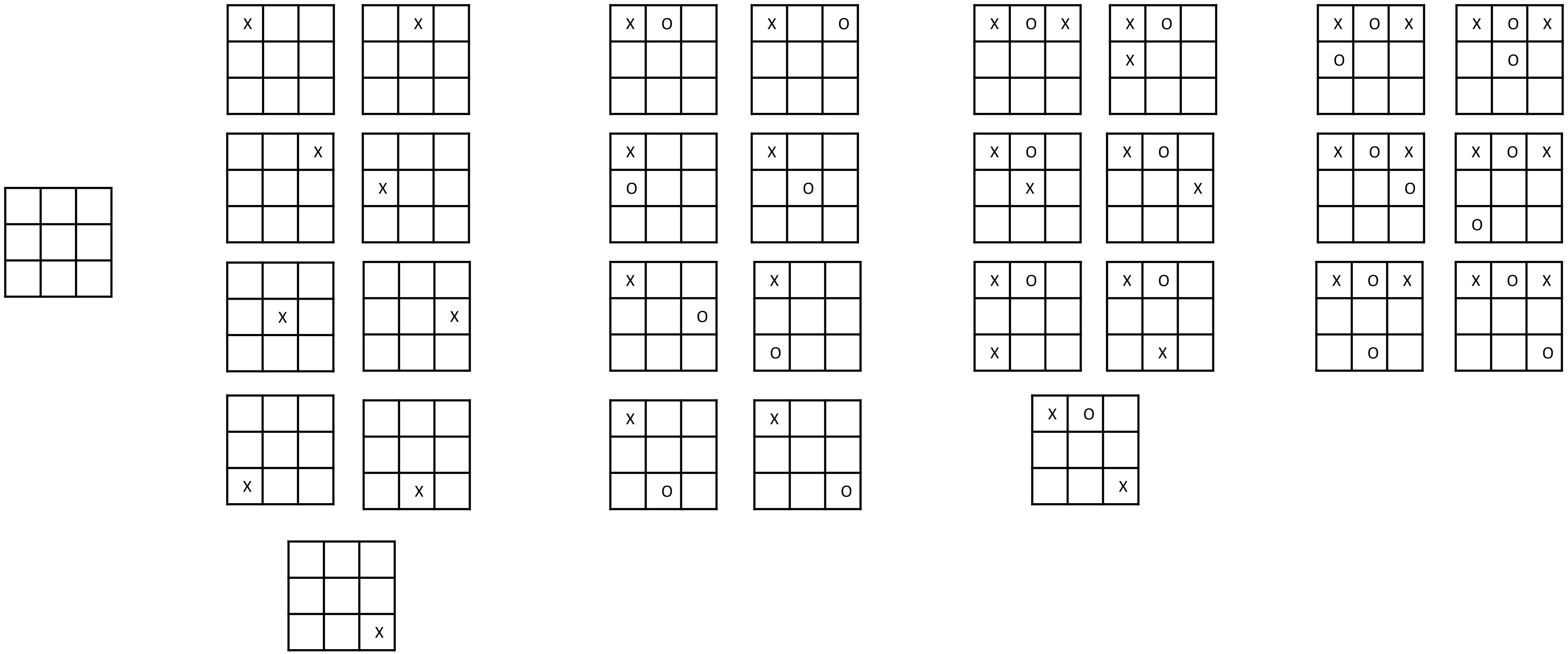
1. What are trees?
2. **Why do you need a tree?**
3. How to create a tree?
4. How to use the tree?

Why Trees?

- Model layouts with hierarchical relationships between items
  - Chain of command in the army
  - Personnel structure in a company
- Optimization problems – Huffman coding (a lossless data compression algorithm. It assigns variable-length codes to input characters based on the usage frequency)
- Permutation, Searching Problems –
  - Eight Queens Problem
  - Gaming eg. Sudoku, Tic-tac-toe

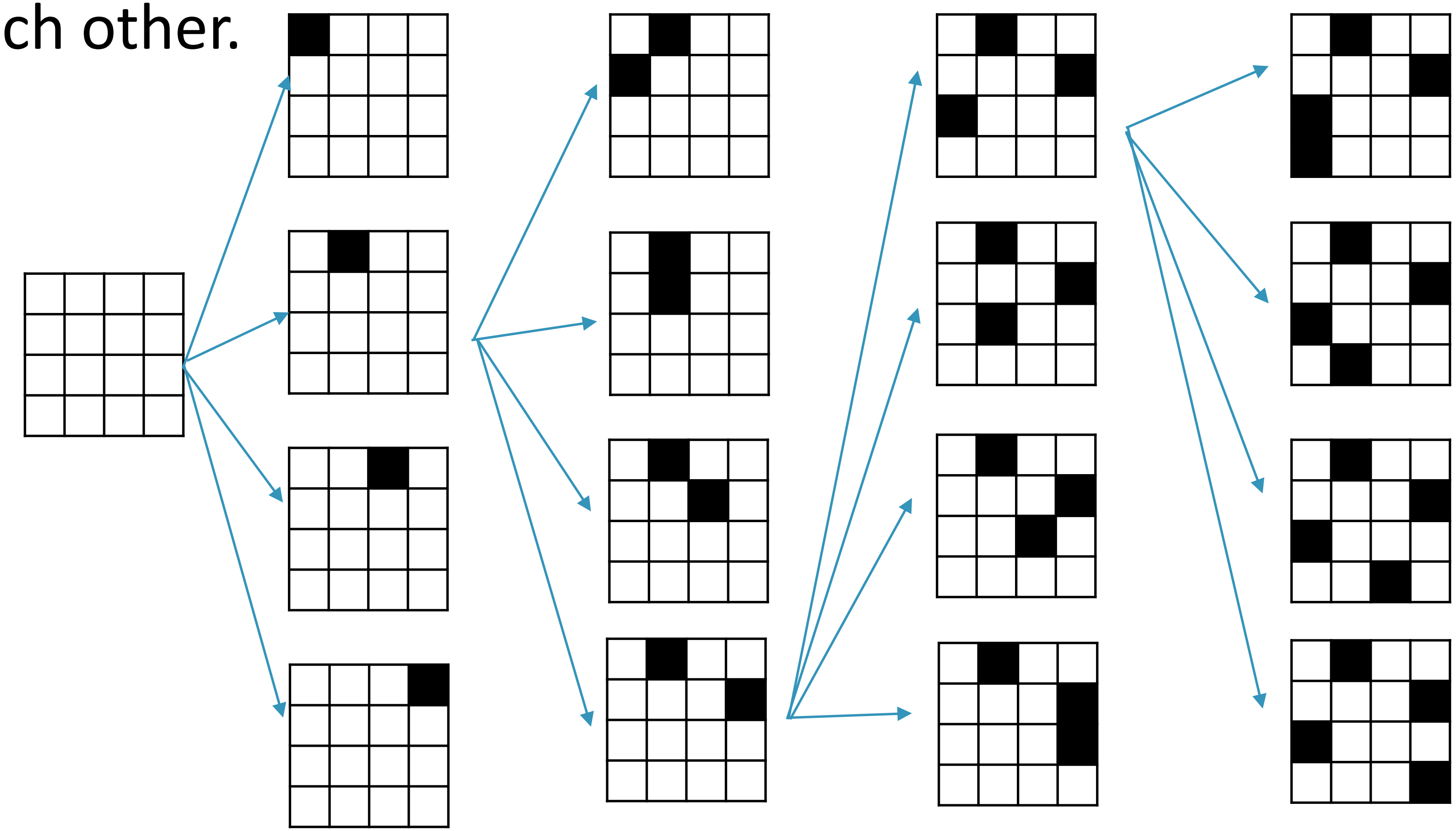
Tic-Tac-Toe

Tic-tac-toe aka noughts and crosses is a paper and pencil game for two players, who take turns marking the spaces in a 3x3 grid. The player who succeeds in placing three of their marks in a horizontal, vertical or diagonal row wins the game

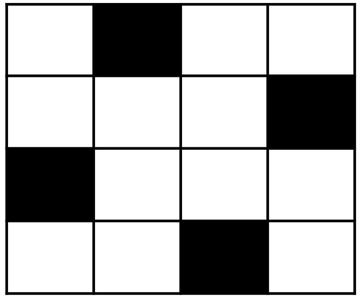


Four Queens Puzzle

Place four queens on a 4x4 chessboard so that no two queens can capture each other.

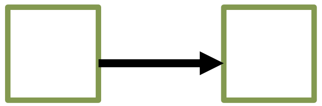


- Each node has 4 children
- The height of the tree is 5

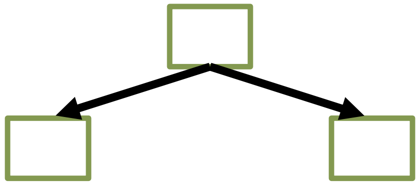


1. What are trees?
2. Why do you need a tree?
3. **How to create a tree?**
4. How to use the tree?

Binary Tree Structure

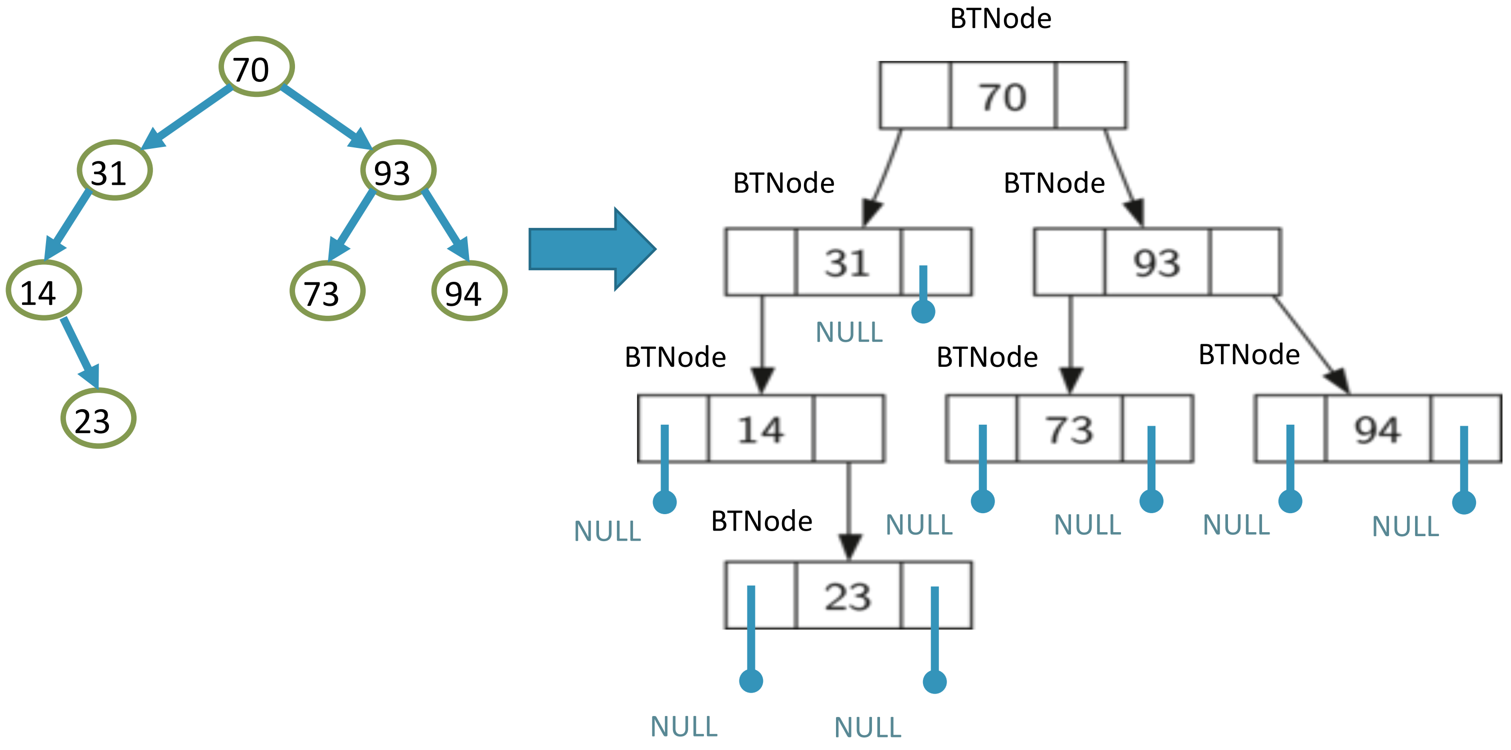


```
typedef struct _listnode{
    int item;
    struct _listnode *next;
}ListNode;
```

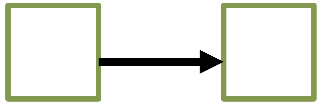


```
typedef struct _btnode{
    int item;
    struct _btnode *left;
    struct _btnode *right;
} BTNode;
```

Example Binary Tree



Tree Traversal Problems

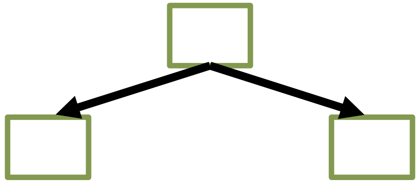


```
typedef struct _listnode{
   int item;
   struct _listnode *next;
}ListNode;
```

Interface Functions

1. Display: printList()
2. Search: findNode()
3. Insert: insertNode()
4. Delete: removeNode()
5. Size: sizeList()

…



```
typedef struct _btnode{
    int item;
    struct _btnode *left;
    struct _btnode *right;
} BTNode;
```

Traversal Problem:
How to systematically travel each node in the tree?

1. What are trees?
2. Why do you need a tree?
3. How to create a tree?
4. **How to use the tree?**
    - **Binary Tree Traversal**

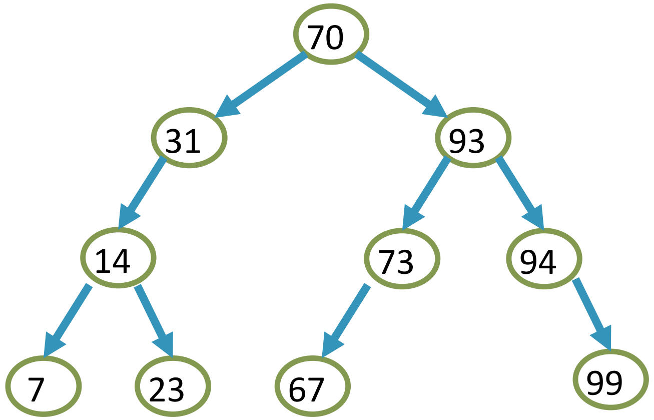
Binary Tree Traversal

```
typedef struct _btnode{
    int item;
    struct _btnode *left;
    struct _btnode *right;
} BTNode;
```

Given a binary tree,

how do you systematically visit every nodes once only?

- **Print the contents of a tree**
- **Search a node**
- **Find the size of a tree**



- **Insert a node**
- **Remove a node**

Binary Tree Traversal

```
typedef struct _btnode{
    int item;
    struct _btnode *left;
    struct _btnode *right;
} BTNode;
```

Traversal Problem:

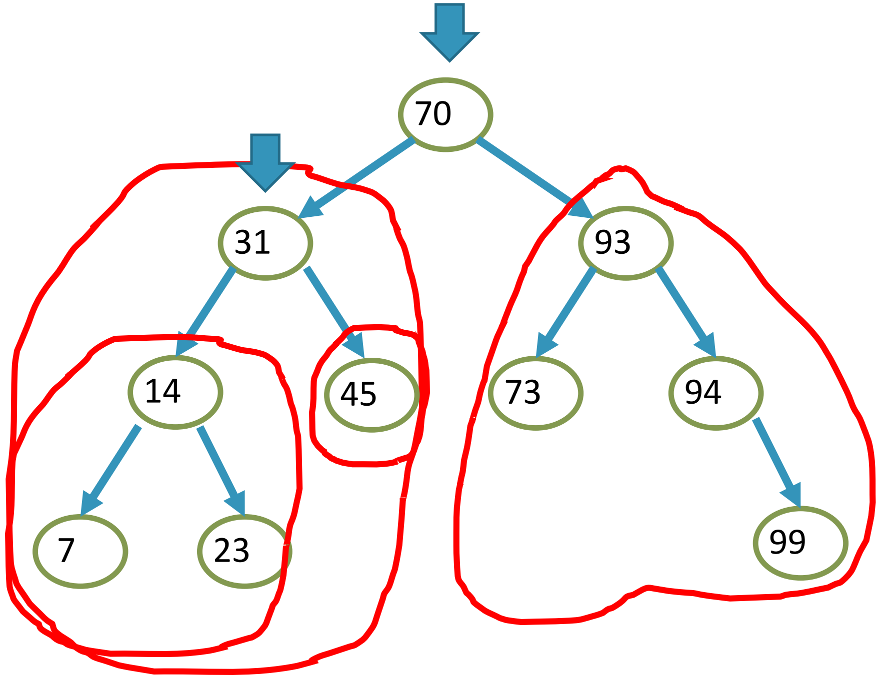
- **Visit root + right subtree + left subtree**
- **Each subtree repeat the same procedure**

  visit root + right subtree + left subtree

- **Until reach the leave**

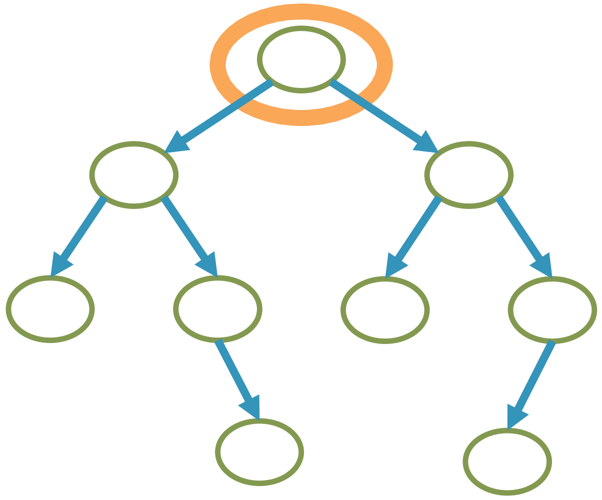
  visit the root (leaf only)

- **It is a recursive problem**



Pseudocode of Binary Tree Traversal

```
TreeTraversal(Node N):
    Visit N;
    If (N has left child)
        TreeTraversal(LeftChild);
    If (N has right child)
        TreeTraversal(RightChild);
    Return; // return to parent
```



This traversal approach is known as pre-order depth first traversal.

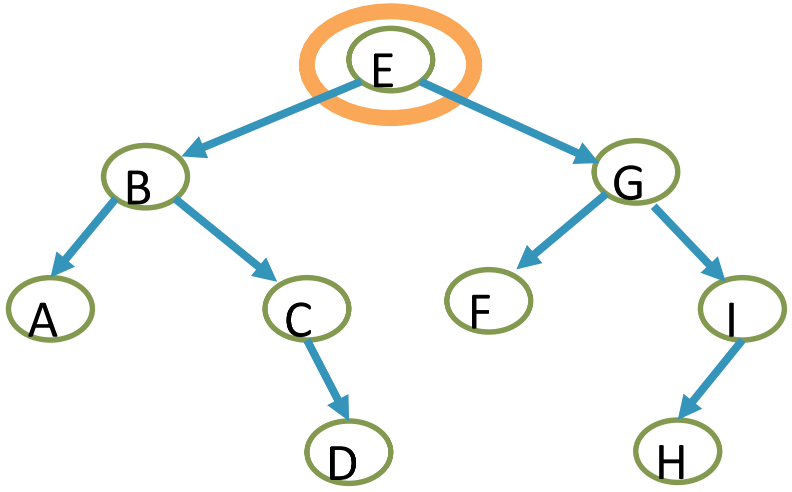
Traversal Approaches on A Binary Tree

- Depth-First Traversal: From the root of a tree, it explores as far as possible. Then it will do the backtracking. There are three traversal orders:
  - Pre-order
  - In-order
  - Post-order

- Breadth-First Traversal: From the root of a tree, it explores each node in level by level.

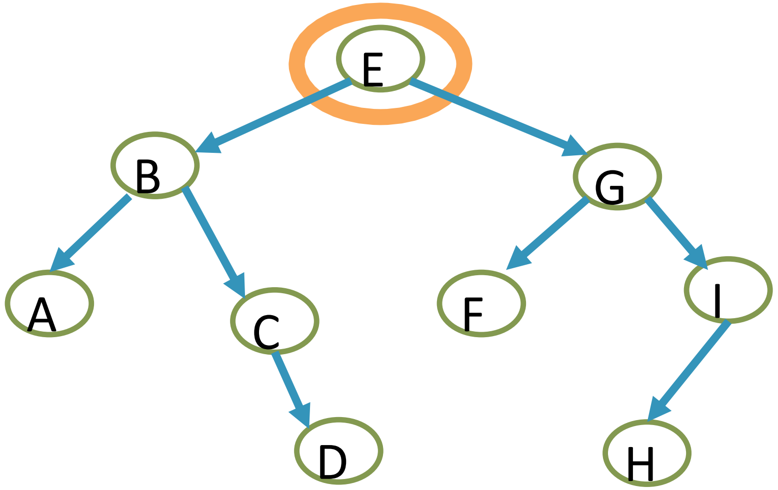
Depth First Traversal: Pre-Order

- **Pre-order**
  - **Process the current node's data**
  - **Visit the left child subtree**
  - **Visit the right child subtree**
- In-order
- Post-order



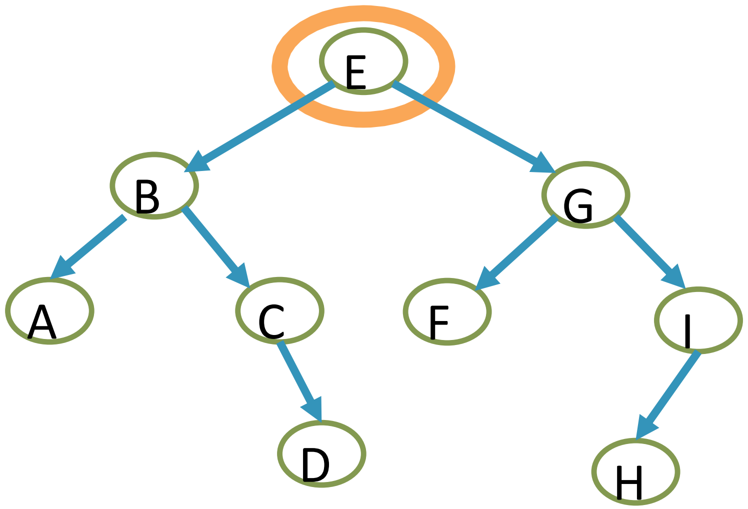
Depth First Traversal: In-Order

- Pre-order
  - Process the current node's data
  - Visit the left child subtree
  - Visit the right child subtree
- **In-order**
  - **Visit the left child subtree**
  - **Process the current node's data**
  - **Visit the right child subtree**
- Post-order



Depth First Traversal: Post-Order

- Pre-order
  - Process the current node's data
  - Visit the left child subtree
  - Visit the right child subtree
- In-order
  - Visit the left child subtree
  - Process the current node's data
  - Visit the right child subtree
- **Post-order**
  - **Visit the left child subtree**
  - **Visit the right child subtree**
  - **Process the current node's data**



Traversal Approaches on A Binary Tree

- Depth-First Traversal: From the root of a tree, it explores as far as possible. Then it will do the backtracking. There are three traversal orders:
  - Pre-order
  - In-order
  - Post-order

- **Breadth-First Traversal**: From the root of a tree, it explores each node in **level by level**.

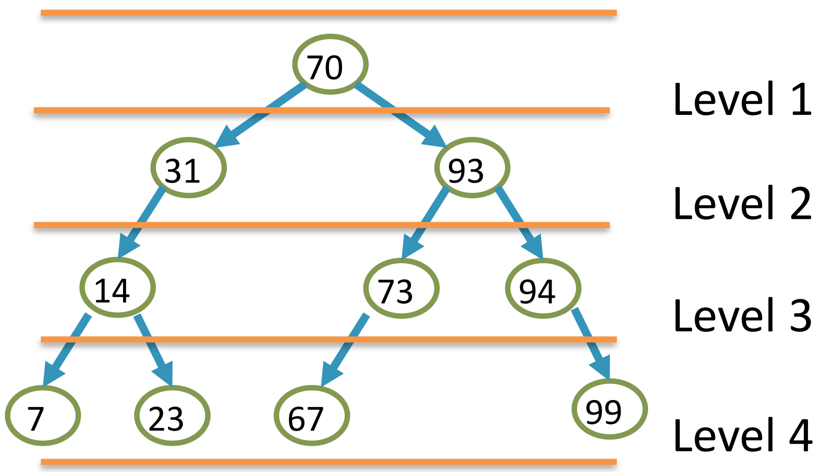
Breadth-First Traversal: Level-by-level

Level-By-Level Traversal:

Visit the root (Level 1)

Visit children of the root (Level 2)

Visit grandchildren of the root (Level3) ...



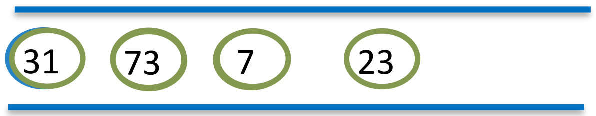
How?

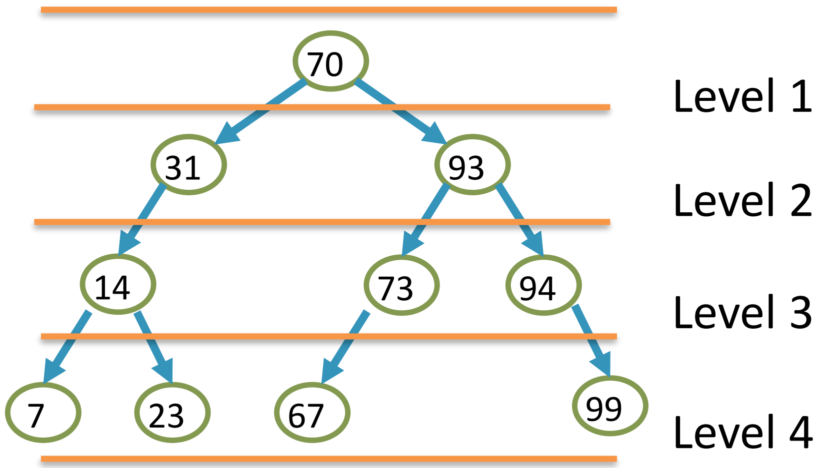
- Visiting the node
- Remember all its children
    - Use a queue (FIFO structure)

Breadth-First Traversal: Level-by-level

Level-By-Level Traversal:

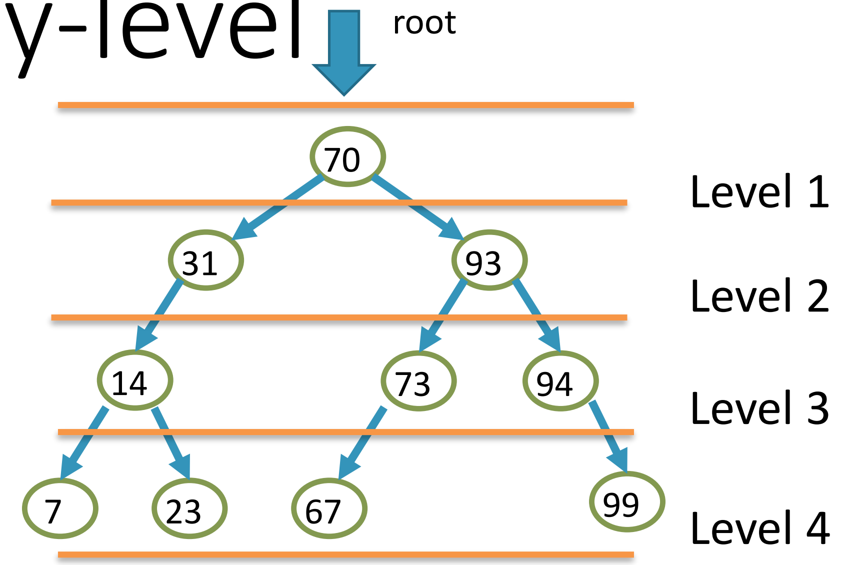
- Visiting the node
- Remember all its children
  - Use a queue (FIFO structure)





1. Enqueue the current node
2. Dequeue a node
3. Enqueue its children if it is available
4. Repeat Step 2 until the queue is empty

Breadth-First Traversal: Level-by-level



Level-By-Level Traversal:

1. Enqueue the current node

2. Dequeue a node

3. Enqueue its children if it is available

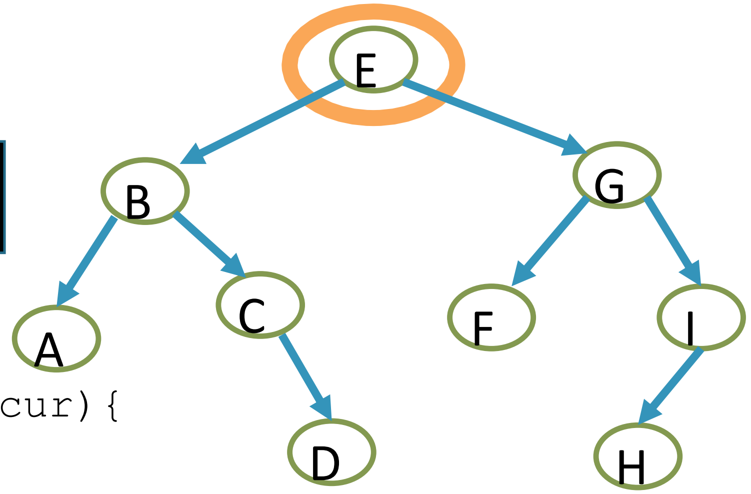
4. Repeat Step 2 until the queue is empty

```
void BFT(BTNode *root){
    Queue *q;
    BTNode* node;
    if(root){
        enqueue(q,root); //data type of item in queue is BTNode*
        while(!isEmptyQueue(*q)){
            node = getFront(*q);dequeue(q);
            if(node->left) enqueue(q,node->left);
            if(node->right) enqueue(q,node->right);
        }
    }
}
```

Tree Traversal Pre-order: Print

Output:

E B A C D G F I H

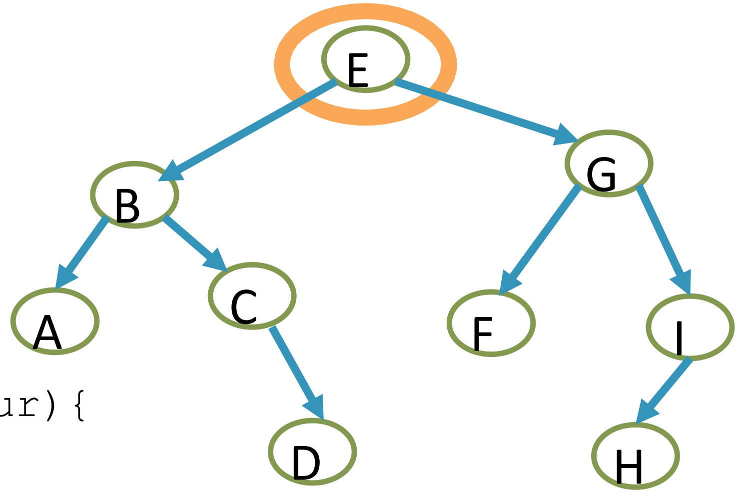



```
void TreeTraversal_pre(BTNode *cur){
    if (cur == NULL)
        return;

    printf("%c  ",cur->item);

    TreeTraversal_pre(cur->left); //Visit the left child node
    TreeTraversal_pre(cur->right);//Visit the right child node
}
```

Tree Traversal In-order: print

Output:

A B C D E F G H I

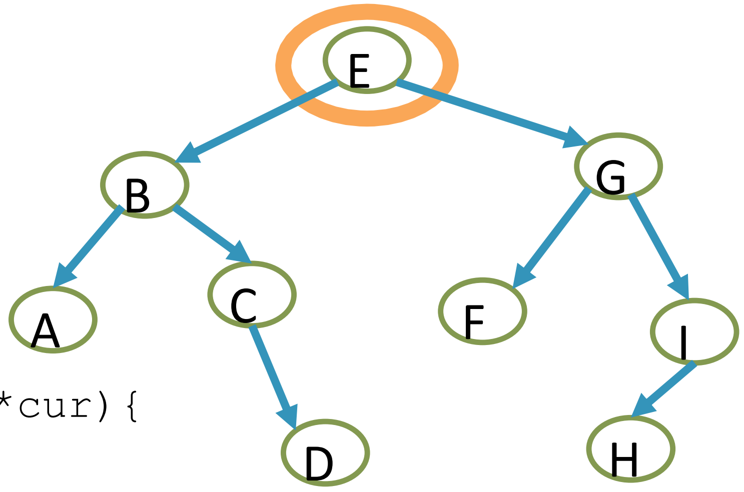



```
void TreeTraversal_in(BTNode *cur){
    if (cur == NULL)
        return;

    TreeTraversal_in(cur->left); //Visit the left child node
    printf("%c  ",cur->item);
    TreeTraversal_in(cur->right);//Visit the right child node
}
```

Tree Traversal Post-order: print

Output:

A D C B F H I G E




```
void TreeTraversal_post(BTNode *cur){
    if (cur == NULL)
        return;

    TreeTraversal_post(cur->left); //Visit the left child node
    TreeTraversal_post(cur->right);//Visit the right child node
    printf("%c  ",cur->item);
}
```

Pre-Order Traversal

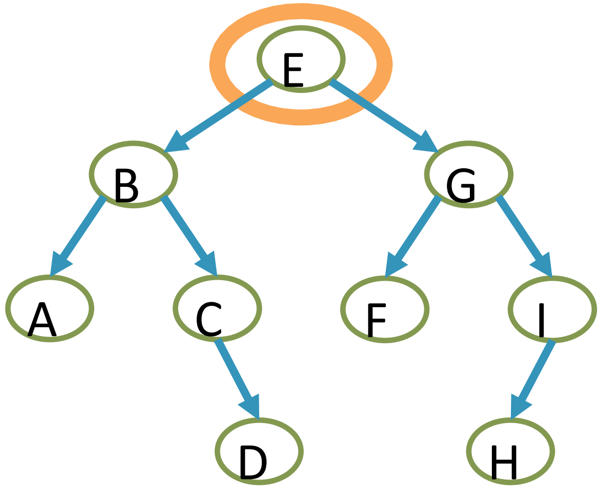
E B A C D G F I H

In-Order Traversal

A B C D E F G H I

Post-Order Traversal

A D C B F H I G E



Count Nodes in a Binary Tree (SIZE)

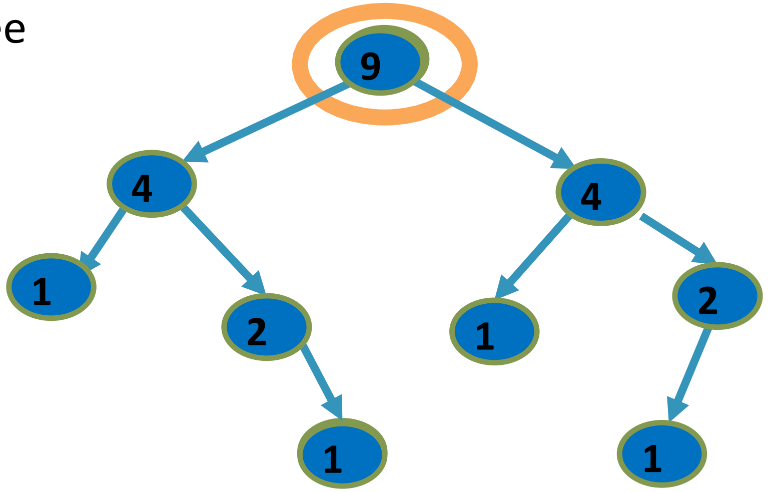
- Recursive definition:
  - Number of nodes in a tree

    = 1

    + number of nodes in left subtree

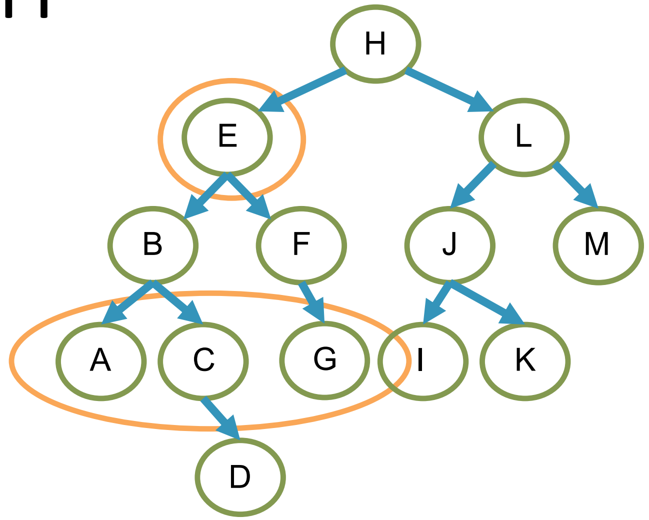
    + number of nodes in right subtree
- Each node returns the number of nodes in its subtree

```
int countNode(BTNode *cur){
    if (cur == NULL)
        return 0;

    return (countNode(cur->left)
          + countNode(cur->right)
          + 1);
}
```



Find the k-level Grandchildren

- Given a node X, find all the nodes that are X's grandchildren
- Given node E, we should return grandchild nodes A, C, and G
- What if we want to find k-level grandchildren?
  - **Need a way to keep track of how many levels down we've gone**




```
void findgrandchildren(BTNode *cur, int c){
    if (cur == NULL) return;
    if (c == k){
        printf("%d ", cur->item);
        return;
    }
    if (c < k){
        findgrandchildren(cur->left, c+1);
        findgrandchildren(cur->right, c+1);
}
```

2-level grandchildren

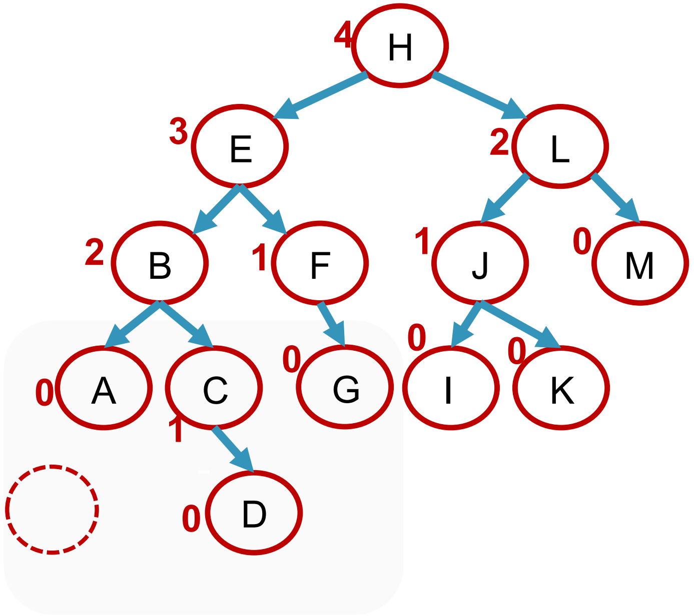
X->left->left
X->left->right
X->right->left
X->right->right

Calculate Height of Every Node

We want each node to report its height

- Leaf node must report 0
- At "null" condition, must report -1

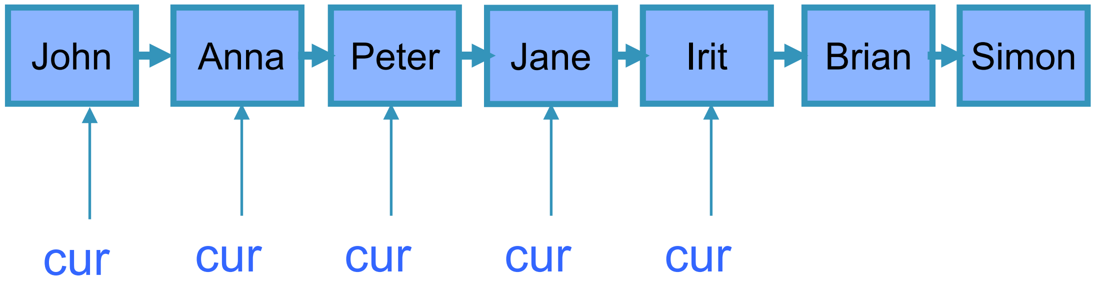
```
int TreeTraversal(BTNode *cur){
    if(cur == NULL)
        return -1;
    int l = TreeTraversal(cur->left);
    int r = TreeTraversal(cur->right);
    int c = max (l, r) + 1;
    return c;
}
```



Sequential Search by a linked list/ array

inefficient

- Given a linked list of names, how do we check whether a given name(e.g., Irit) is in the list?




```
while (cur!=NULL) {
        if cur->item == "Irit"
            found and stop searching;
        else
            cur = cur->next; }
```

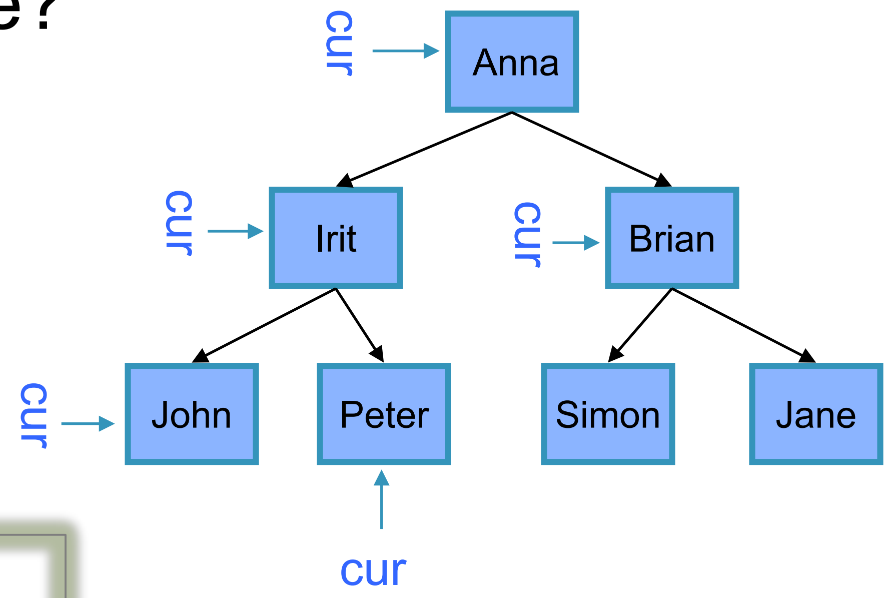
How many nodes are visited during search?
--best case: 1 node (John) => Θ(1)
--worst case: 7 nodes (Simon) => Θ(n)
--avg. case: (1+2+3+…+7)/7=4 nodes => Θ(n)

Sequential Search by a binary tree

inefficient

- Given a binary tree of names, how do we check whether a given name(e.g., Brian) is in the tree?

- **Use the TreeTraversal (Pre-order) template, to check every node**




```
TreeTraversal(Node N)
 if N==NULL return;
 if N.item=given_name return;
 TreeTraversal(LeftChild);
 TreeTraversal(RightChild);
 Return;
```

How many nodes are visited during search?
--best case: 1 node (John) => Θ(1)
--worst case: 7 nodes (Simon) => Θ(n)
--avg. case: (1+2+3+…+7)/7=4 nodes => Θ(n)

Summary

- The difference between linked lists and tree structures (linear and non-linear data structures)
- Overview of Tree
- Tree Traversal
  - Depth-First Traversal
    - Pre-order Traversal
    - In-order Traversal
    - Post-order Traversal
  - Breadth-First Traversal: Level-by-level traversal
- Examples

Make sure that you know the difference among them

SC1007
Data Structures and Algorithms

Binary Search Tree

College of Engineering

School of Computer Science and Engineering

Dr. Loke Yuan Ren

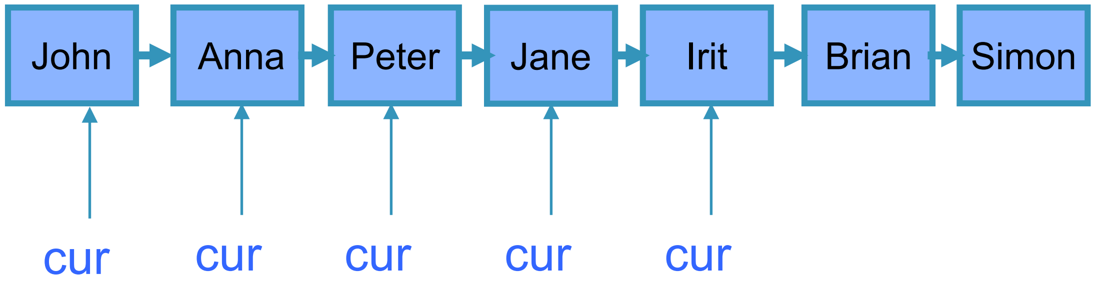
Lecturer

yrloke@ntu.edu.sg

Linear Search by a linked list/ array

inefficient

- Given a linked list of names, how do we check whether a given name(e.g., Irit) is in the list?




```
while (cur!=NULL) {
        if cur->item == "Irit"
            found and stop searching;
        else
            cur = cur->next; }
```

How many nodes are visited during search?
--best case: 1 node (John) => Θ(1)
--worst case: 7 nodes (Simon) => Θ(n)
--avg. case: (1+2+3+…+7)/7=4 nodes => Θ(n)

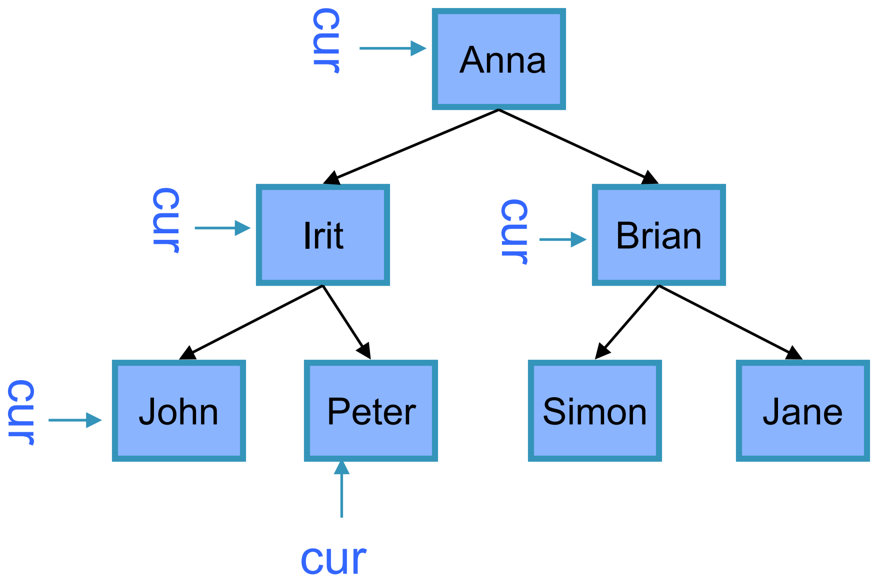
Linear Search by a binary tree

inefficient

- Given a binary tree of names, how do we check whether a given name(e.g., Brian) is in the tree?

- **Use the TreeTraversal (Pre-order) template, to check every node**

How do we insert data into the binary tree?



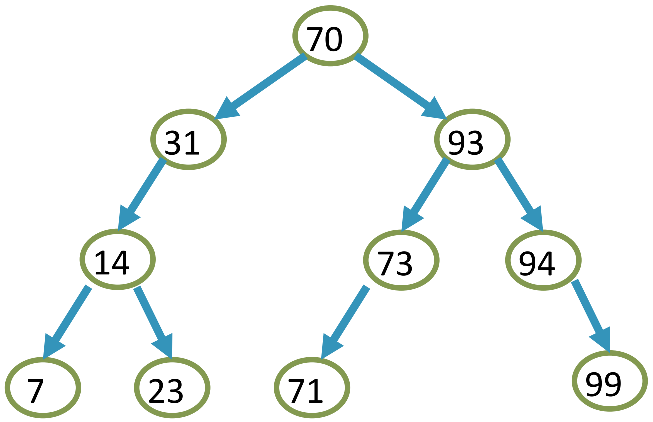

```
TreeTraversal(Node N)
 if N==NULL return;
 if N.item=given_name return;
 TreeTraversal(LeftChild);
 TreeTraversal(RightChild);
 Return;
```

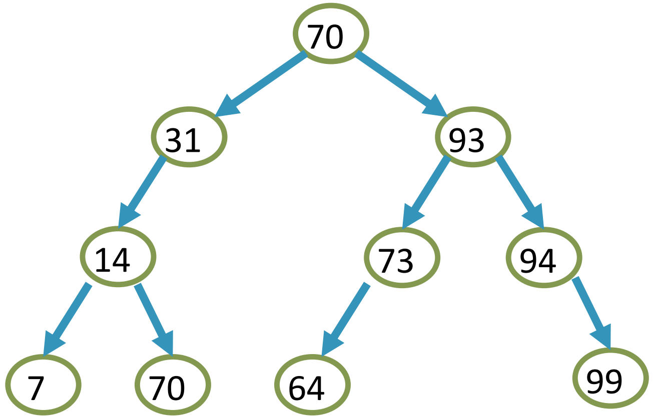
How many nodes are visited during search?
--best case: 1 node (John) => Θ(1)
--worst case: 7 nodes (Simon) => Θ(n)
--avg. case: (1+2+3+…+7)/7=4 nodes => Θ(n)

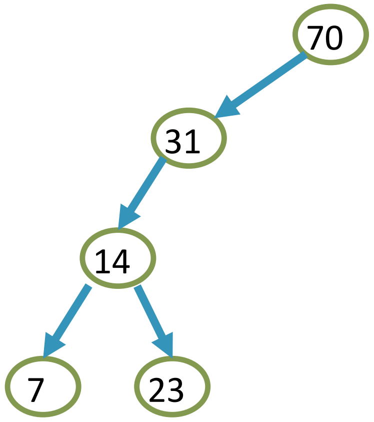
Binary Search Tree

If the given binary tree is a binary search tree (BST), then each node in the tree satisfies the following properties:

1. **Node's value is greater than all values in its left subtree.**
2. **Node's value is less than all values in its right subtree.**
3. **Both subtrees of the node are also binary search trees.**







Binary Search Tree: Search

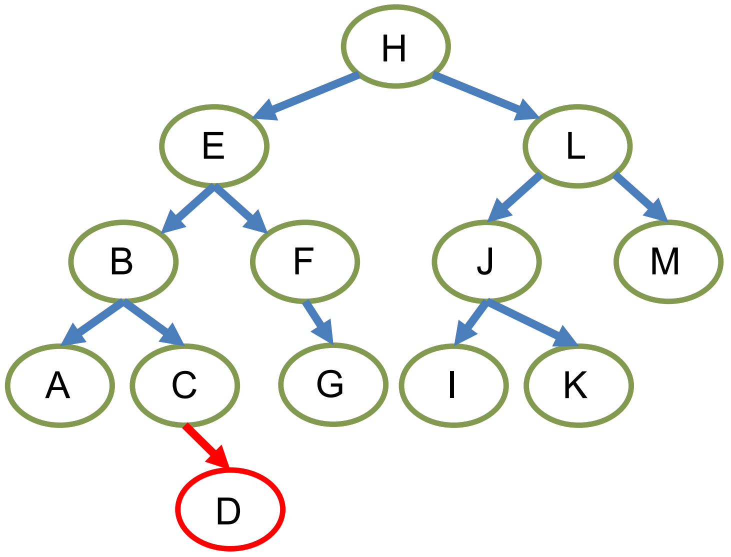
- The approach is a decrease-and-conquer approach
- A problem is divided into two smaller and similar sub-problem, one of which does not even have to be solved
- The method uses the information of the order to reduce the search space.

```
BTNode* findBSTNode(BTNode *cur, char c){
    if (cur == NULL) {
       printf("can't find!");
       return cur;
    }

    if(c==cur->item){
       printf("Found!\n");
       return cur;
    }

    if(c<cur->item)
       return findBSTNode(cur->left,c);
    else
       return findBSTNode(cur->right,c);
}
```

```
void TreeTraversal_pre(BTNode *cur){
    if (cur == NULL) return;

    printf("%c  ",cur->item);

    TreeTraversal_pre(cur->left);
    TreeTraversal_pre(cur->right);
}
```

Binary Search Tree: Insertion

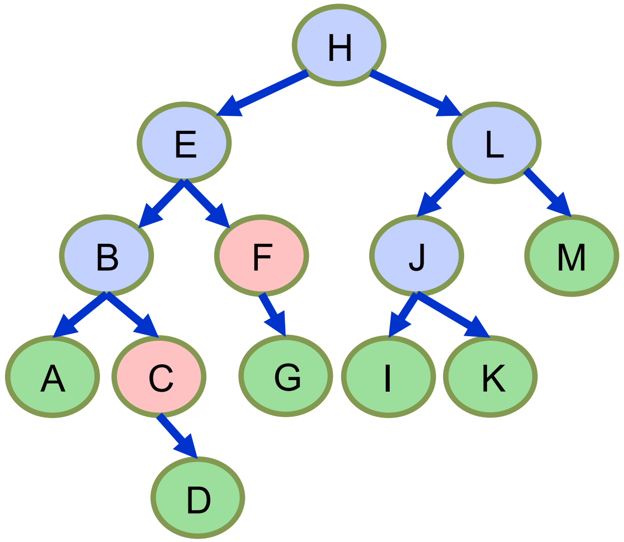
- **After insert a node, the BST must remain as a BST**
- **A duplicate node is not allowed for insertion**
- **A unique position of the given BST will be given to the node**




```
BTNode* insertBSTNode(BTNode* cur, char c)
{
    if (cur == NULL){
     BTNode* node = (BTNode*) malloc(sizeof(BTNode));
     node->item = c;
     node->left = node->right = NULL;
     return node;
     }
    if (c < cur->item)
        cur->left  = insertBSTNode (cur->left, c);
    else if (c > cur->item)
        cur->right = insertBSTNode (cur->right, c);

    return cur;
}
```

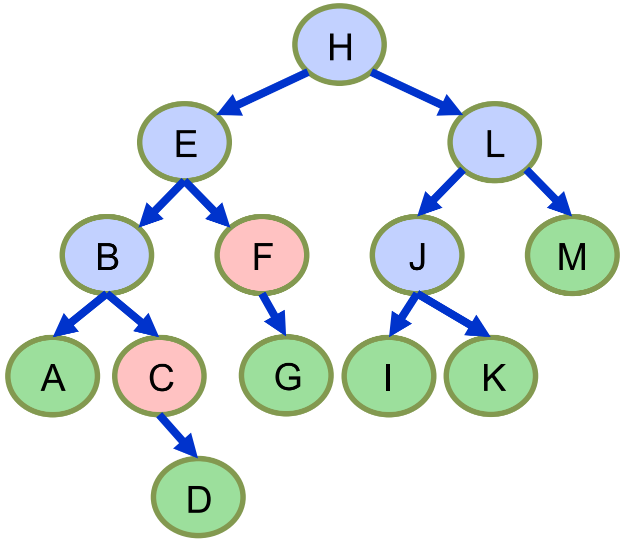
Binary Search Tree: Deletion

- **After remove a node X, the BST must remain as a BST**
- **Deletion operation on a BST is a bit tricky**
- **Three cases to be considered:**
  1. **X has no children**
  2. **X has one child**
  3. **X has two children**



Binary Search Tree: Deletion

- **After remove a node X, the BST must remain as a BST**
- **Deletion operation on a BST is a bit tricky**
- **Three cases to be considered:**
  1. **X has no children**
     - Remove **X**

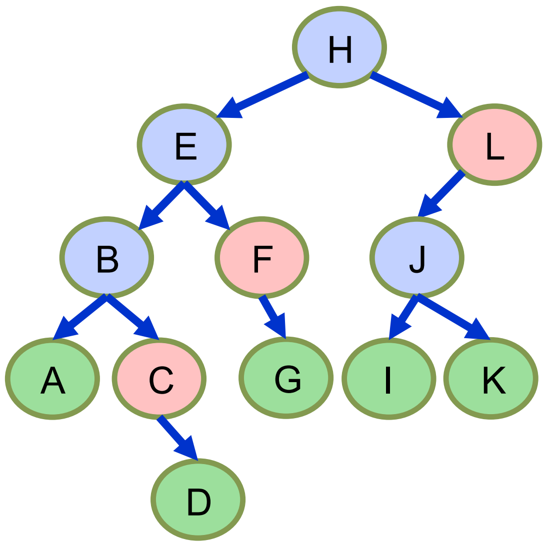


Binary Search Tree: Deletion

- **After remove a node X, the BST must remain as a BST**
- **Deletion operation on a BST is a bit tricky**
- **Three cases to be considered:**

  2. X has one child

  - Replace X with Y
  - Remove X

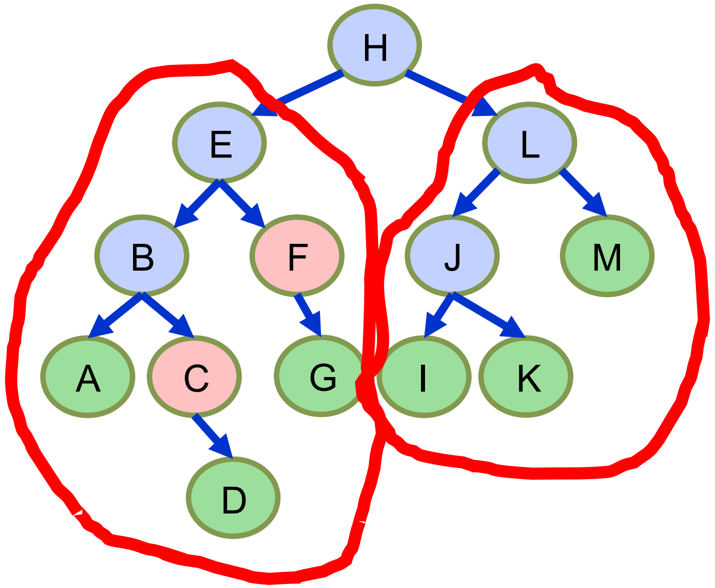


Binary Search Tree: Deletion

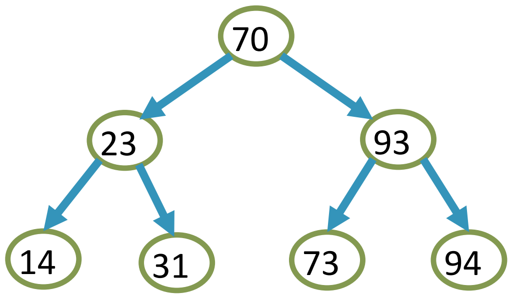
- **After remove a node X, the BST must remain as a BST**
- **Deletion operation on a BST is a bit tricky**
- **Three cases to be considered:**

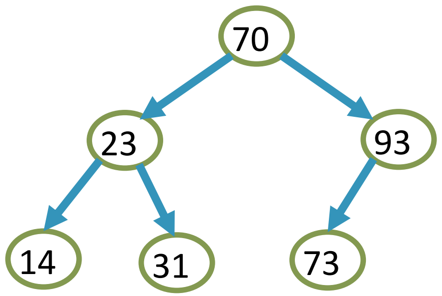
  3. X has two children

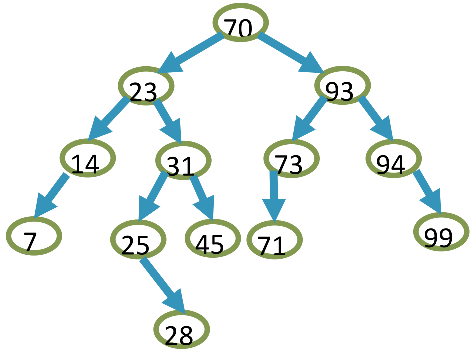
  - **Swap x with successor**
    - the (largest) rightmost node in left subtree
    - the (smallest) leftmost node in right subtree
  - **Perform case 1 or 2 to remove it**



Terminology

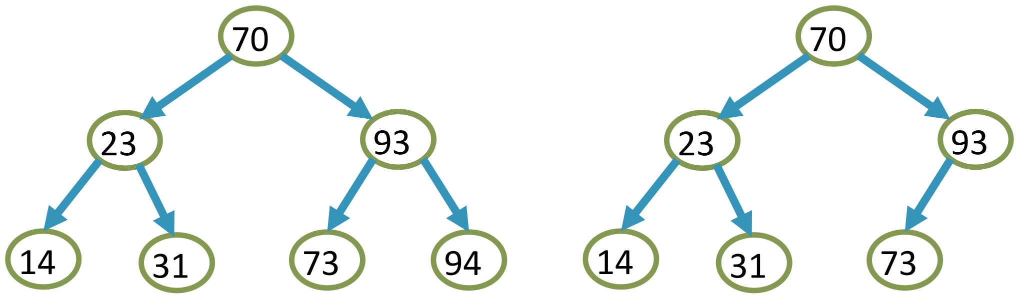






- The Height of a tree: The number of **edges** on the longest path from the root to a leaf
- The Depth/Level of a node: The number of edges from the node to the root of its tree.
- Empty Binary Tree: A binary tree with no nodes. It is still considered as a tree.
- Full Binary Tree: A binary tree of height *H* with no missing nodes. All leaves are at level *H* and all other nodes each have two children
- Complete Binary Tree: A binary tree of height *H* that is full to level *H-1* and has level *H* filled in from left to right
- Balanced Binary Tree: A binary tree in which the left and right subtrees of any node have heights that differ by at most 1

Terminology



- The Height of a tree: The number of **edges** on the longest path from the root to a leaf
- The Depth/Level of a node: The number of edges from the node to the root of its tree.

For a complete binary tree with height *H*, we have:

$$\mathbf{2^H\text{-}1 < \textit{n} \leq 2^{H+1} - 1}$$

where *n* is an integer and the size of the tree

$$\mathbf{2^H \leq \textit{n} < 2^{H+1}} \qquad \textbf{(eg. } 7<n \leq 15 \equiv 8 \leq n<16\textbf{)}$$

$$\mathbf{H \leq \log_2 n < H+1}$$

If H is an integer, H+1 must be the next integer.

Minimal Height = $\lfloor \log_2 n \rfloor$

Binary Search – Worst Case Time complexity

- Assumed that it is a complete binary tree

```
BTNode* findBSTNode(BTNode *cur, char c){
    if (cur == NULL) {
       printf("can't find!");
       return cur;
    }

    if(c==cur->item){
       printf("Found!\n");
       return cur;
    }

    if(c<cur->item)
       return findBSTNode(cur->left,c);
    else
       return findBSTNode(cur->right,c);
}
```

T(n)

Constant c

$\boldsymbol{T((n-1)/2)}$

$\boldsymbol{T((n-1)/2)}$

$$T(n) = T\left(\frac{n-1}{2}\right) + c$$

$$= T\left(\frac{\left(\frac{n-1}{2}\right)-1}{2}\right) + 2c = T\left(\frac{n-1-2}{2^2}\right) + 2c$$

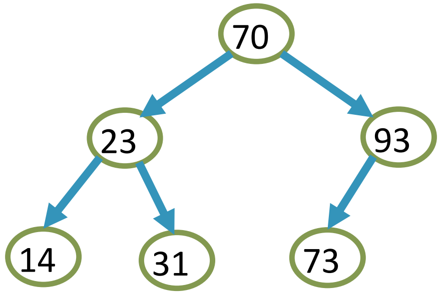
$$= T\left(\frac{\frac{n-1-2}{2^2}-1}{2}\right) + 3c = T\left(\frac{n-1-2-2^2}{2^3}\right) + 3c$$

…

$$= T\left(\frac{n-(1+2\ldots+2^{k-2}+2^{k-1})}{2^k}\right) + kc$$

$$= T\left(\frac{n-2^k+1}{2^k}\right) + kc$$

Binary Search – Worst Case Time complexity




```
BTNode* findBSTNode(BTNode *cur, char c){
    if (cur == NULL) {
       printf("can't find!");
       return cur;
    }

    if(c==cur->item){
       printf("Found!\n");
       return cur;
    }

    if(c<cur->item)
       return findBSTNode(cur->left,c);
    else
       return findBSTNode(cur->right,c);
}
```

T(n)

Constant c

$\boldsymbol{T((n-1)/2)}$

$\boldsymbol{T((n-1)/2)}$

- Assumed that it is a complete binary tree

$$= T\left(\frac{n - 2^k + 1}{2^k}\right) + kc$$

$$0 < \frac{n - 2^k + 1}{2^k} \leq 1$$

$$0 < \frac{n + 1}{2^k} - 1 \leq 1$$

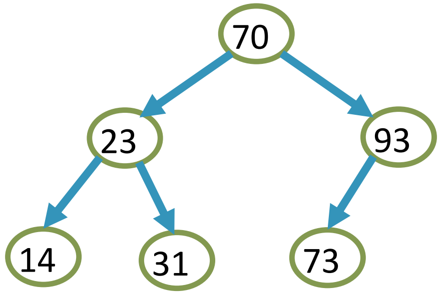
$$1 < \frac{n + 1}{2^k} \leq 2$$

$$2^k < n + 1 \leq 2^{k+1}$$

$$k < \log_2(n + 1) \leq k + 1$$

$$\lceil \log_2(n + 1) \rceil = k + 1$$

Binary Search – Worst Case Time complexity



- Assumed that it is a complete binary tree

```
BTNode* findBSTNode(BTNode *cur, char c){
    if (cur == NULL) {
       printf("can't find!");
       return cur;
    }

    if(c==cur->item){
       printf("Found!\n");
       return cur;
    }

    if(c<cur->item)
       return findBSTNode(cur->left,c);
    else
       return findBSTNode(cur->right,c);
}
```

T(n)

Constant c

$\boldsymbol{T((n-1)/2)}$

$\boldsymbol{T((n-1)/2)}$

$$= T\left(\frac{n - 2^k + 1}{2^k}\right) + kc$$

$$\lceil \log_2(n+1) \rceil = k + 1$$

$$\lfloor \log_2 n \rfloor + 1 = \mathrm{k} + 1$$

$$k = \lfloor \log_2 n \rfloor$$

$$= c + kc$$

$$= (\lfloor \log_2 n \rfloor + 1)c$$

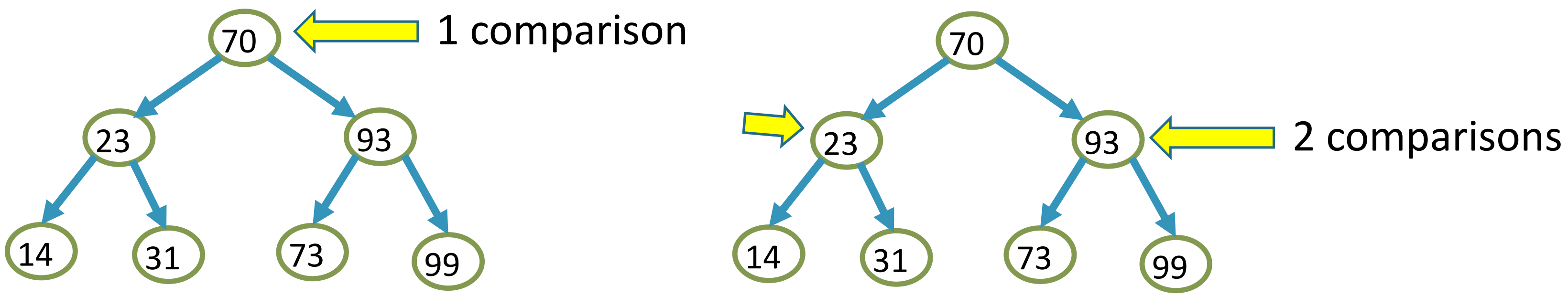
$$= \Theta(\log_2 n)$$

Binary Search – Average Case Time Complexity

- $A_s(n)$: # of comparisons for successful search
- $A_f(n)$: # of comparisons for unsuccessful search (worst case): $\Theta(\log_2 n)$

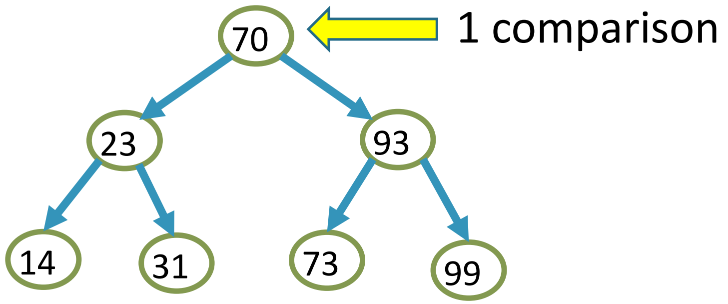
$$A(n) = qA_s(n) + (1-q)A_f(n)$$

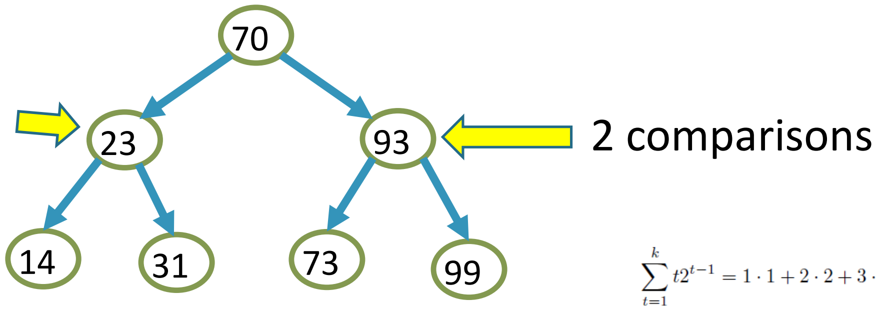
For $A_s(n)$, we assume $n = 2^k - 1$ first



Binary Search – Average Case Time Complexity

$$A(n) = qA_s(n) + (1-q)A_f(n)$$





- For $A_s(n)$, we assume $n = 2^k - 1$ first
- We can observe that:
  - 1 position requires 1 comparison
  - 2 positions requires 2 comparisons
  - 4 positions requires 3 comparisons
  - … …
  - 2^{t-1} positions requires t comparisons
- $n=2^k-1$, we have

$$A_s(n) = \frac{1}{n}\sum_{t=1}^{k} t2^{t-1}$$
$$= \frac{(k-1)2^k + 1}{n}$$
$$= \frac{[\log_2(n+1) - 1](n+1) + 1}{n}$$
$$= \log_2(n+1) - 1 + \frac{\log_2(n+1)}{n}$$

$$\sum_{t=1}^{k} t2^{t-1} = 1\cdot 1 + 2\cdot 2 + 3\cdot 4 + 4\cdot 8 + \ldots + k\cdot 2^{k-1}$$
$$2\sum_{t=1}^{k} t2^{t-1} = \quad 1\cdot 2 + 2\cdot 4 + 3\cdot 8 + \ldots + (k-1)\cdot 2^{k-1} + k\cdot 2^k$$
$$(2-1)\sum_{t=1}^{k} t2^{t-1} = -1\cdot 1 - 1\cdot 2 - 1\cdot 4 - 1\cdot 8 - \ldots - 1\cdot 2^{k-1} + k\cdot 2^k \quad \triangleright \text{eq. 2 - eq. 1}$$
$$\sum_{t=1}^{k} t2^{t-1} = -2^k + 1 + k\cdot 2^k \quad \triangleright \text{geometric series}$$
$$= 2^k(k-1) + 1$$

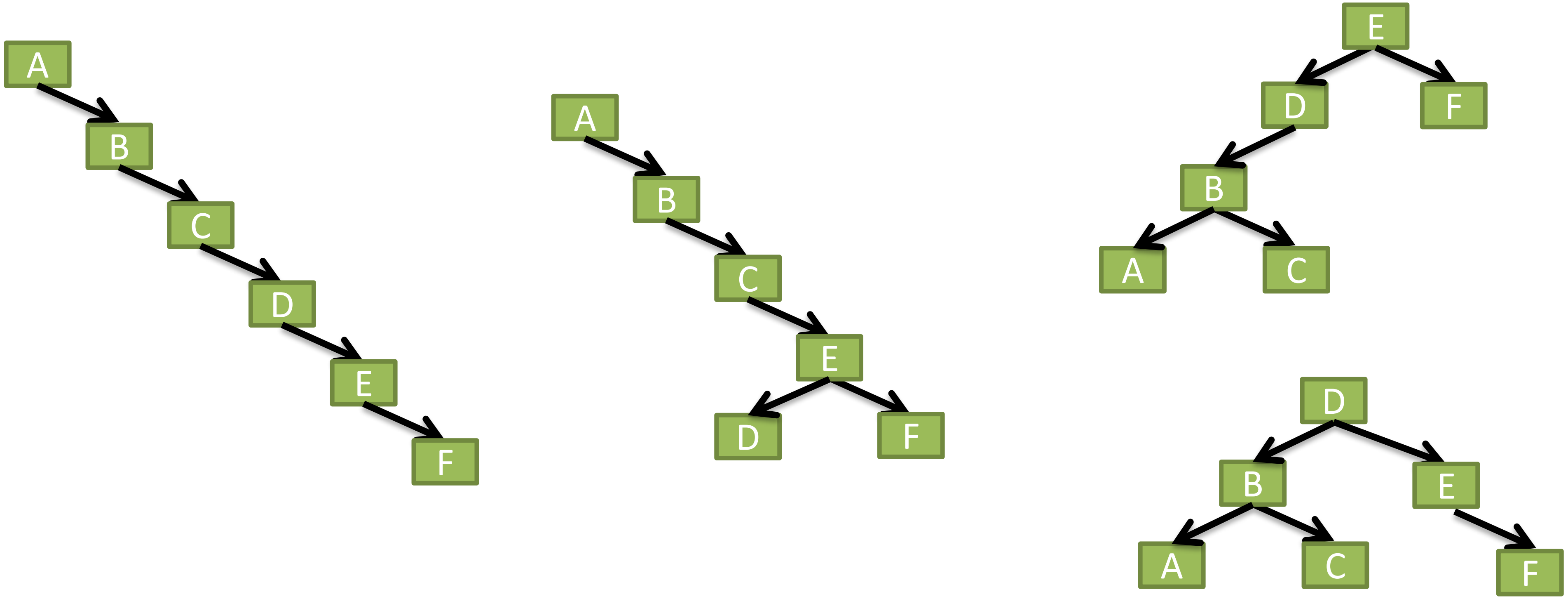
Binary Search – Average Case Time Complexity

- The time complexity is

$$\begin{aligned}
A_q(n) &= qA_s(n) + (1-q)A_f(n) \\
&= q[\log_2(n+1) - 1 + \frac{\log_2(n+1)}{n}] + (1-q)(\log_2(n+1)) \\
&= \log_2(n+1) - q + q\frac{\log_2(n+1)}{n} \\
&= \Theta(log_2(n))
\end{aligned}$$

- Binary search does approximately $\log_2(n+1)$ comparisons on average for n entries.
  - q is probability which is always ≤ 1
  - $\frac{\log_2(n+1)}{n}$ is very small especially when n >> 1

The ‘Good’ and ‘Bad’ Binary Search Trees



Summary

- Linear Search VS Binary Search
- Binary Search Tree: Search, Insertion and Deletion
- Time Complexity of Binary Search
- The importance of having a good tree structure

- Next Lecture
  - Tree Balancing