

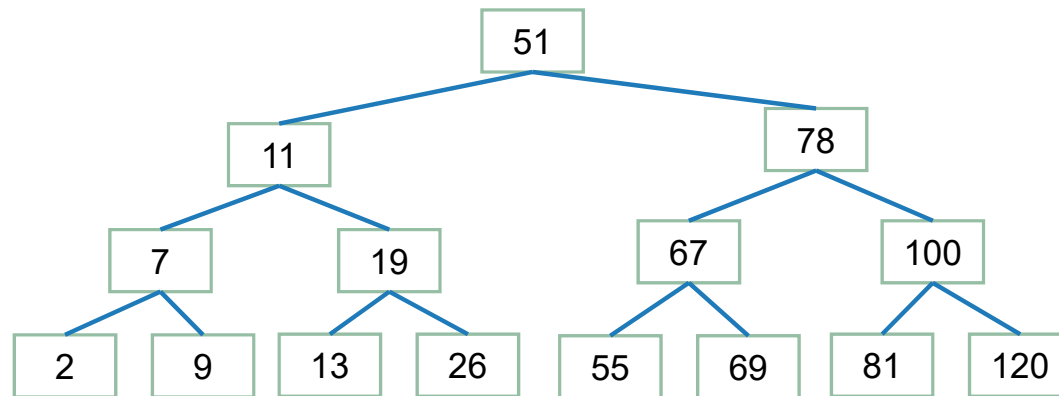


Trees

Yi-Shin Chen
Institute of Information Systems and Applications
Department of Computer Science
National Tsing Hua University
yishin@gmail.com

Tree Structure

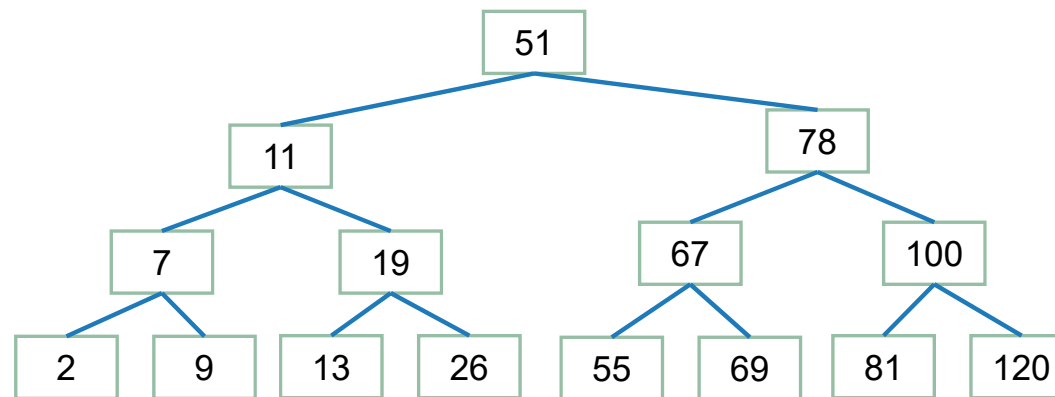
- Data in a tree structure are organized in a **hierarchical** manner



Tree Definition

■ A **tree** is a finite set of one or more nodes

- There is one **root**
- The remaining nodes can be partitioned into n disjoint sets T_1, T_2, \dots, T_n ($n \geq 0$)
- Each subset T_i is a tree (also called **subtrees** of the root)



Terminology

■ Degree of a node

- The number of subtrees
- E.g., $\text{deg}(A) = 3$
- E.g., $\text{deg}(C) = 1$

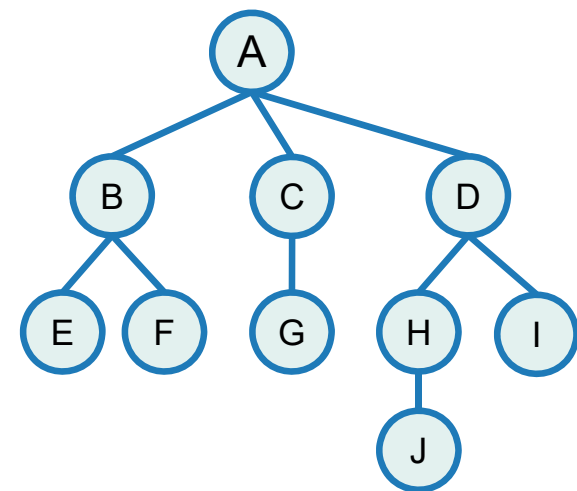
■ Leaf or Terminal nodes

- The node whose degree is 0
- E.g., E, F, G, J, I

■ Non-terminals

■ Degree of a tree

- The maximum degree of the nodes in the tree
- E.g., $\text{Deg. of the tree} = 3$



Terminology (Contd.)

■ Parent / Children

■ Sibling

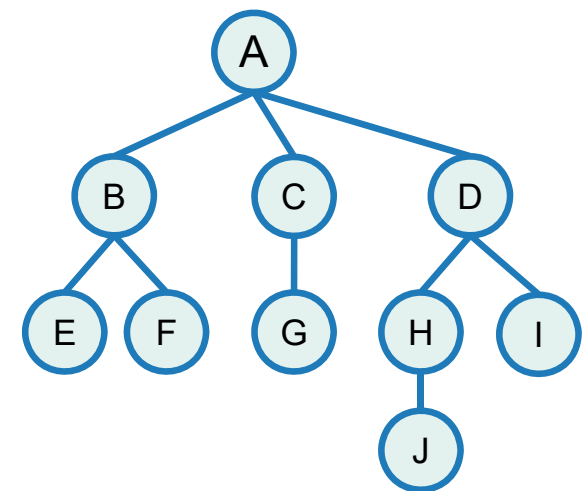
- Children of the same parent
- E.g., B, C, D are siblings

■ Ancestors

- All nodes along the path from the root to that node
- E.g., ancestor of J: H, D, A

■ Descendants

- All nodes in the subtrees



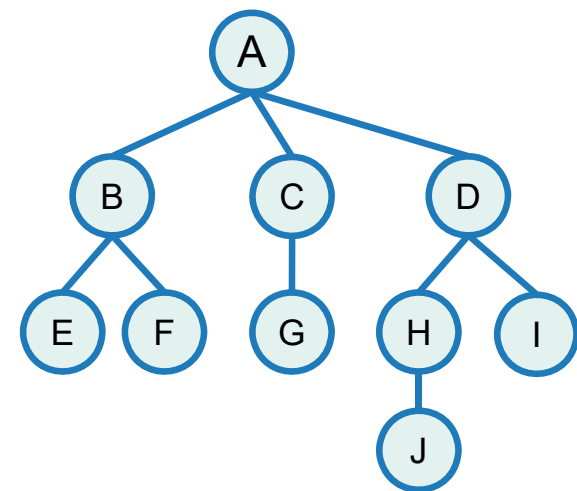
Terminology (Contd.)

■ Level of a node

- $\text{Level}(\text{root}) = 1$
- $\text{Level}(n) = \ell + 1$
 - if level of n 's parent is ℓ
- E.g., $\text{level}(G) =$

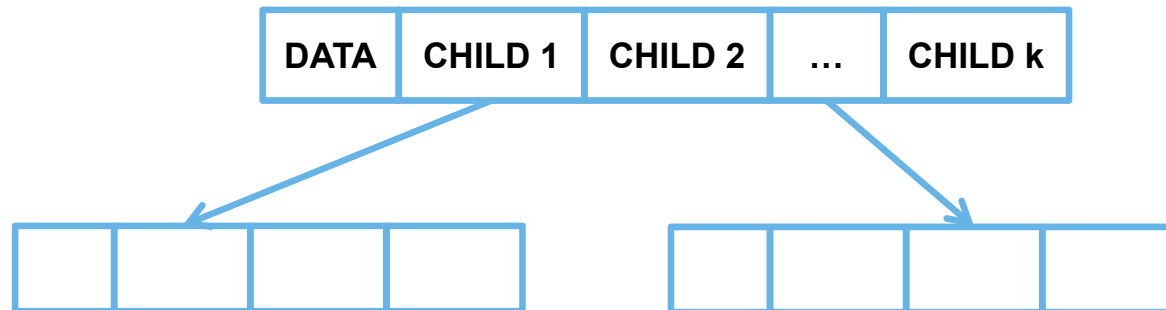
■ Height or depth of a tree

- Maximum level of any node in the tree
- E.g., : Height of the tree = 4



List Representation

- Each tree node holds
 - A **data field**
 - Several **link fields** pointing to subtrees
 - Based on the degree of each node
 - E.g., For tree of **degree k**, allocate **k** link field for each node

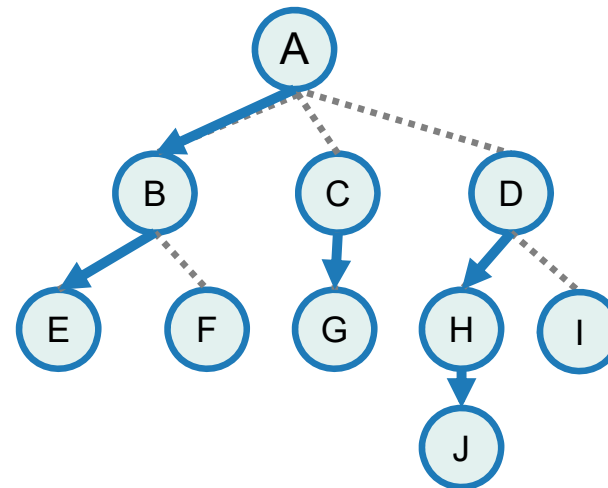


List Representation

- Disadvantage: Wasting the memory!
 - If T is a tree of degree k with n nodes.
 - The total # of link fields are $n*k$
 - Might have a lot of unused links

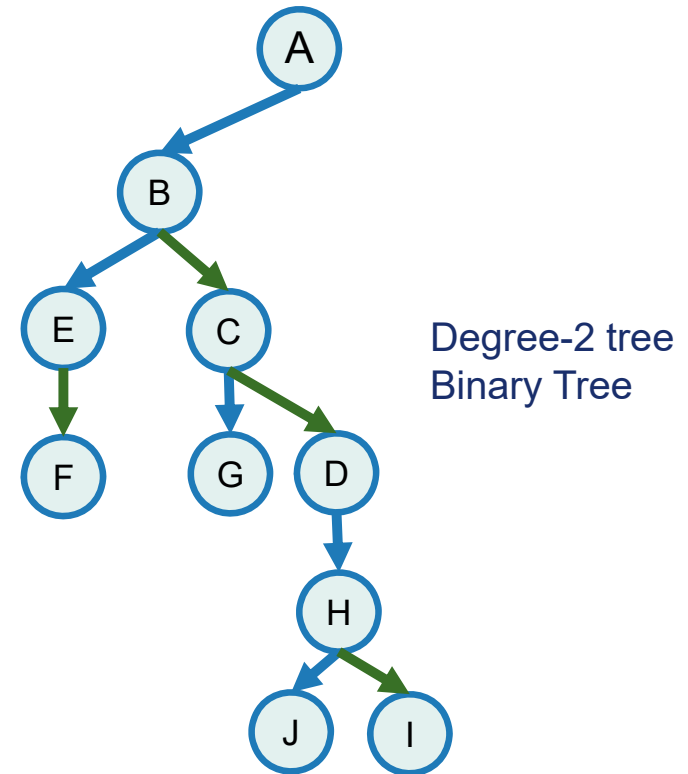
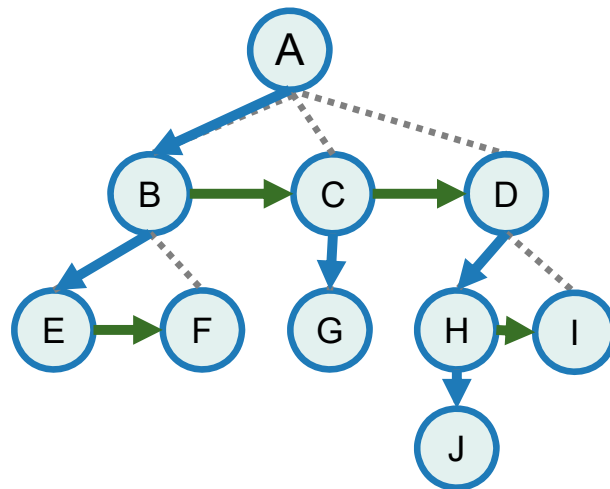
Left Child-Right Sibling Representation

- Each node has exactly two link fields
 - Left link(child): points to leftmost child node
 - Right link(sibling): points to closest sibling node



Left Child-Right Sibling Representation

■ Rotate clockwise 45°

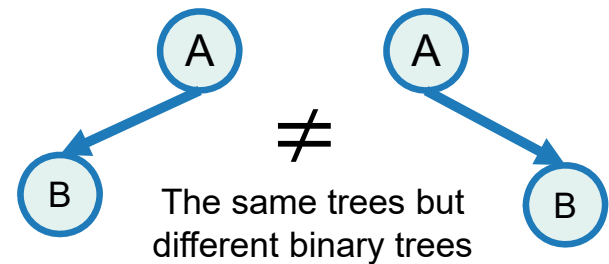


Overview of Binary Trees

■ A binary tree is a finite set of nodes:

- Either is empty
- Or consists of
 - A root
 - Two disjoint binary trees
 - The left subtree
 - The right subtree

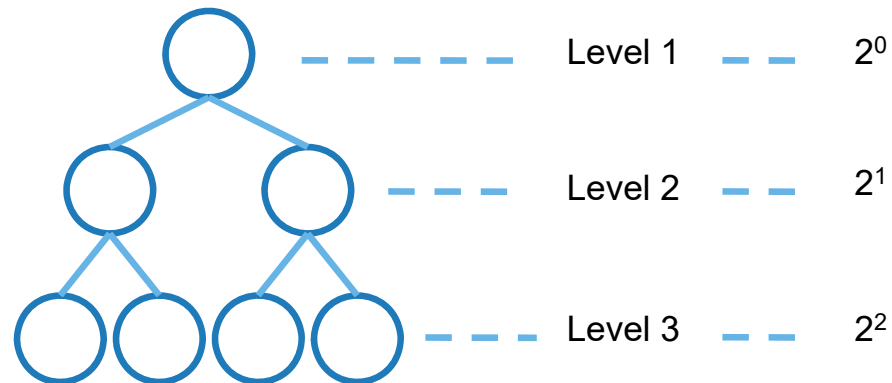
■ Binary tree \neq Regular tree



Properties of Binary Tree

■ [Maximum number of nodes]

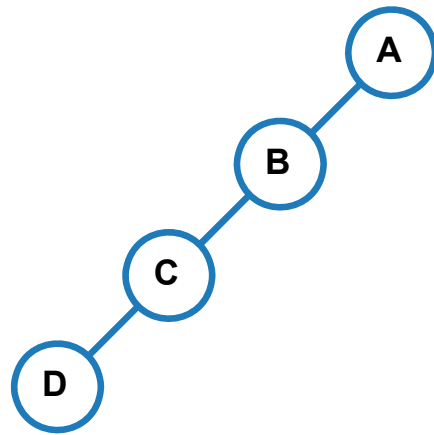
- The max. # of nodes on level i is $2^{(i-1)}$
- The max. # of nodes in a binary tree with depth k is $2^k - 1$



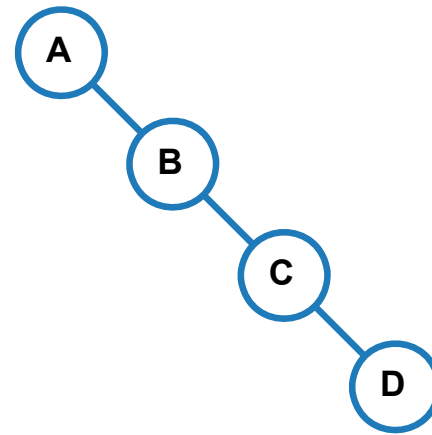
Total # of node is $1 + 2 + 2^2 + 2^3 + \dots + 2^{(k-1)} = 2^k - 1$

Special Binary Trees

■ Skewed tree



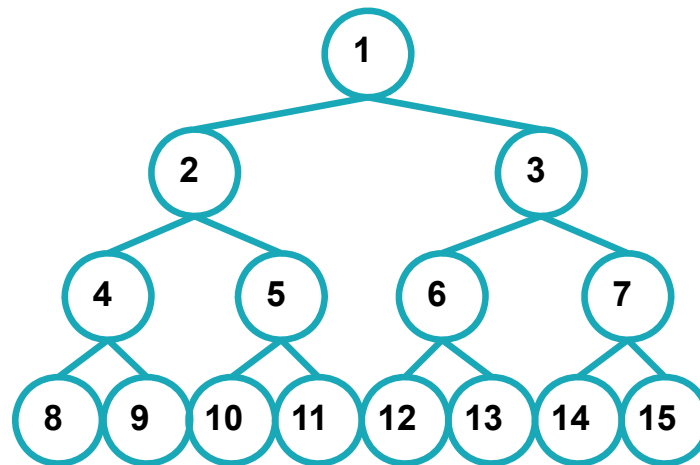
Skewed to the left



Skewed to the right

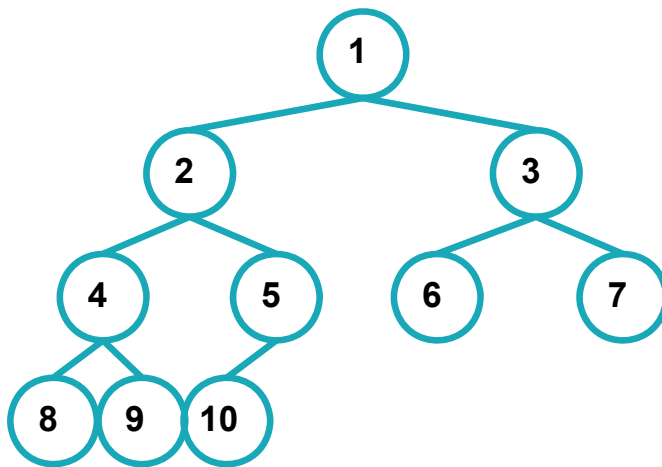
Full Binary Tree

- A binary tree of depth k which has $2^k - 1$ nodes



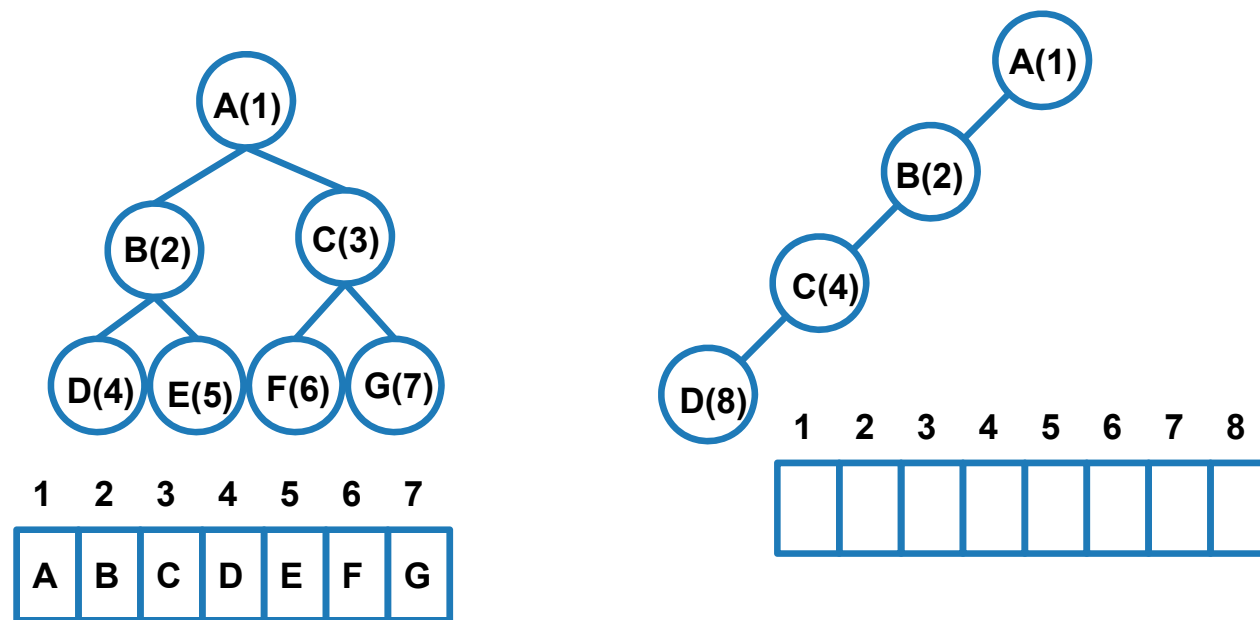
Complete Binary Tree

- A binary tree of depth k with n node is called complete
 - iff its nodes correspond to the nodes numbered from 1 to n in the full binary tree



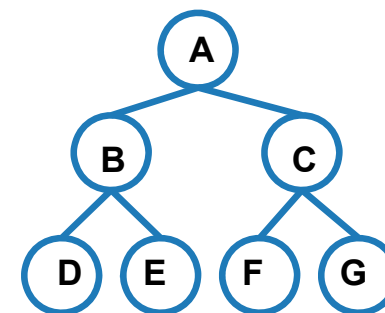
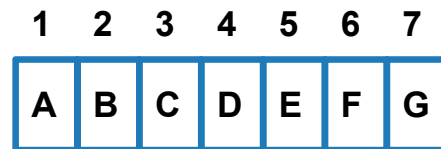
Array Representation

- The numbering scheme suggests to use a 1-D array



Array Representation

- Advantages: Easy to determine the locations of the parent, left child, and right child of any node.
- Let node i be in position i (array[0] is empty)
 - $\text{Parent}(i) = i / 2$ if $i \neq 1$. If $i=1$, i is the root and has no parent
 - $\text{leftChild}(i) = 2i$ if $2i \leq n$. If $2i > n$, the i has no left child
 - $\text{rightChild}(i) = 2i+1$ if $2i+1 \leq n$, if $2i+1 > n$, the i has no right child



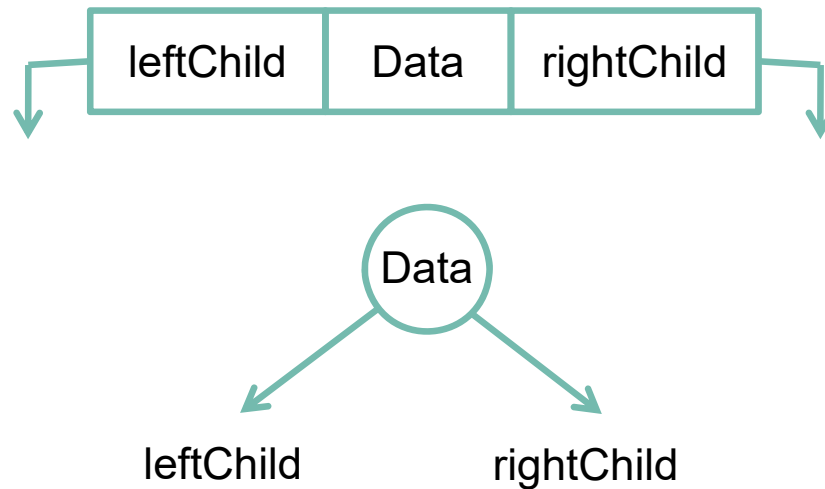
Array Representation (Contd.)

■ Disadvantages:

- Wasteful of space for a skewed tree
- Insertion and deletion of nodes require move a large parts of existing nodes
 - To maintain sorted

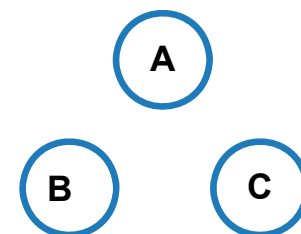
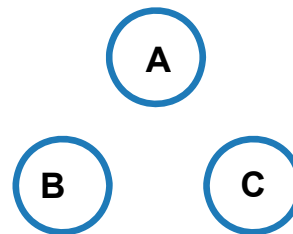
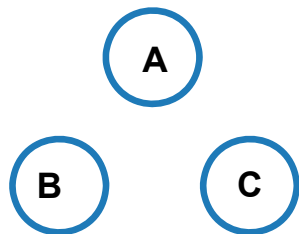
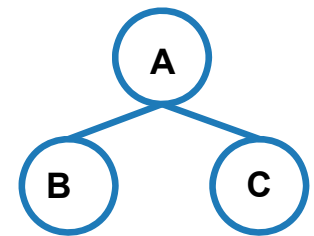
Linked Representation

- Each tree node consists of three fields
 - Data, leftChild, and rightChild



Binary Tree Traversal

- Visit each node in a tree exactly once
- Treat each node and its subtrees in the same fashion
 - Inorder: left -> root -> right
 - Preorder: root -> left -> right
 - Postorder: left -> right -> root

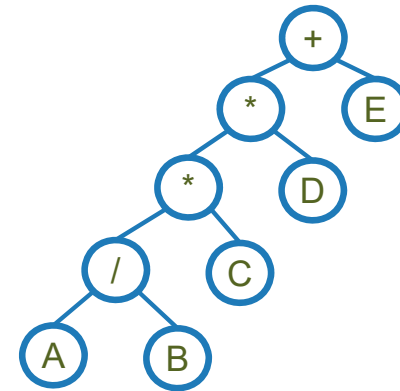


Inorder Traversal

■ Steps of traversal:

- Step1: Moving down the tree toward the **left** until you can go no farther
- Step2: **Visit** the node
- Step3: Move one node to the **right** and continue

■ Use recursion to describe this traversal



Inorder Traversal : Codes

```
template < class T >
void Tree<T>::Inorder()
{ // Start a recursive inorder traversal
  // This function is a public member function of Tree
  Inorder(root);
}

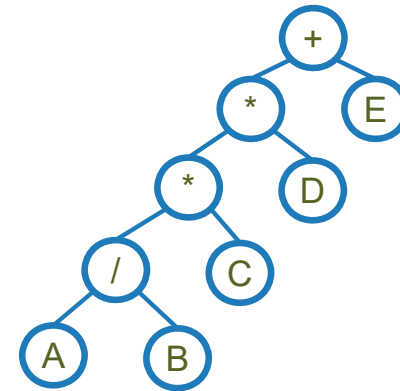
template <class T>
void Tree<T>::Inorder(TreeNode<T>* currentNode)
{ // Recursive inorder traversal function
  // This function is a private member function of Tree
  if(currentNode){
    Inorder(currentNode->leftChild);
    Visit(currentNode); // e.g., printout information
    Inorder(currentNode->RightChild);
  }
}
```

Preorder Traversal

■ Steps of traversal:

- Step1: Visit a node
- Step2: Traverse left, and continue
- Step3: When cannot continue, move right and begin again

■ Use recursion to describe this traversal



Preorder Traversal : Codes

```
template < class T >
void Tree<T>::Preorder()
{ // Start a recursive preorder traversal
  // This function is a public member function of Tree
  Preorder(root);
}

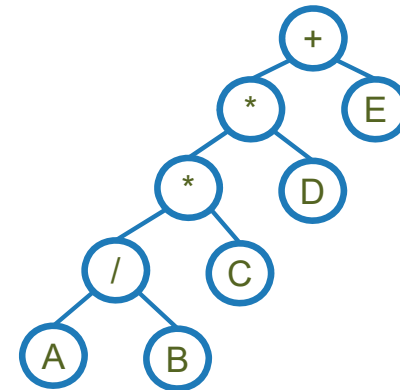
template <class T>
void Tree<T>::Preorder(TreeNode<T>* currentNode)
{ // Recursive preorder traversal function
  // This function is a private member function of Tree
  if(currentNode){
    Visit(currentNode); // e.g., printout information
    Preorder(currentNode->leftChild);
    Preorder(currentNode->RightChild);
  }
}
```


Postorder Traversal

■ Steps of traversal:

- Step1: Moving down the tree toward the left until you can go no farther
- Step2: Move one node to the right
- Step3: Move back to visit the node and go right

■ Use recursion to describe this traversal

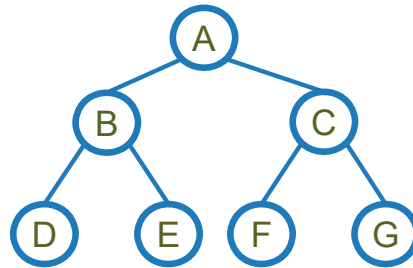


Postorder Traversal : Codes

```
template < class T >
void Tree<T>::Postorder()
{ // Start a recursive postorder traversal
  // This function is a public member function of Tree
  Postorder(root);
}

template <class T>
void Tree<T>::Postorder(TreeNode<T>* currentNode)
{ // Recursive postorder traversal function
  // This function is a private member function of Tree
  if(currentNode){
    Postorder(currentNode->leftChild);
    Postorder(currentNode->RightChild);
    Visit(currentNode); // e.g., printout information
  }
}
```

Exercise



Traversal	Output ordered list
Inorder	
Preorder	
Postorder	

Tree Iterator

- We would like to visit nodes by using **iterator** visit elements in a container
 - Recursive traversal is not suitable
- An **iterative** version
 - Using stack to store non-visited nodes

Non-Recursive Inorder Traversal

```
template < class T >
void Tree<T>::NonrecInorder()
{ // Non recursive inorder traversal using stack
  Stack<TreeNode<T>*> s;    // declare and init a stack
  TreeNode<T>* currentNode = root;
  while(1){
    while(currentNode){      // move down leftChild field
      s.Push(currentNode); // add to stack
      currentNode = currentNode->leftChild;
    }
    if(s.IsEmpty()) return; // all nodes are visited
    currentNode = s.Top();
    s.Pop();
    Visit(currentNode);      // e.g., printout information
    currentNode = currentNode->rightNode;
  }
}
```

Inorder Iterator

```
Class InorderIterator{ // A nested class within Tree
public:
    InorderIterator() { currentNode = root}
    T* Next();
private:
    Stack<TreeNode<T>*> s;
    TreeNode<T>* currentNode;
};

T* InorderIterator::Next()
{
    while(currentNode){ // Move down leftChild field
        s.Push(currentNode); // Add to stack
        currentNode = currentNode->leftChild;
    }
    if(s.IsEmpty()) return NULL; // All nodes are visited
    currentNode = s.Top();
    s.Pop();
    T& temp = currentNode->data;
    currentNode = currentNode->rightNode;
    return &temp;
}
```

Level-Order Traversal

- Visit nodes in a top to down, left to right manner

Level

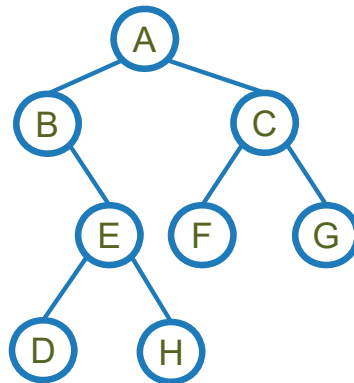
1



2

3

4



A B C E F G D H

Preorder	Inorder	Postorder	Level-Order

Level-Order Traversal : Codes

```
template <class T>
void Tree<T>::LevelOrder()
{ // Traverse the binary tree in level order
  Queue<TreeNode<T>*> q;
  TreeNode<T>* currentNode = root;
  while(currentNode) {
    Visit(currentNode);
    if(currentNode->leftChild) q.Push(currentNode->leftChild);
    if(currentNode->rightChild) q.Push(currentNode->rightChild);
    if(q.IsEmpty()) return;
    currentNode = q.Front();
    q.Pop();
  }
}
```

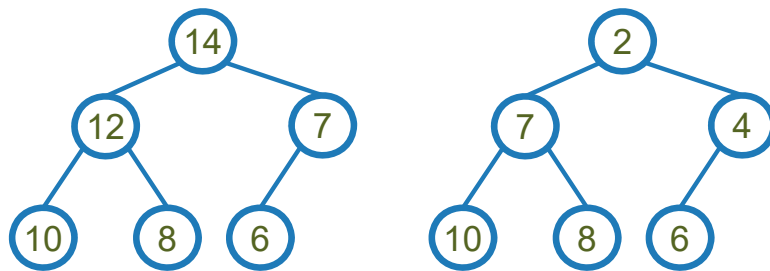

Self-Study Topics

- Binary tree operations
 - Preorder traversal (Non-recursive & iterator)
 - Postorder traversal (Non-recursive & iterator)
 - Copying Binary Trees
 - Testing Equality



Heaps

- A specialized tree-based data structure
 - Satisfy the heap property
 - The value of parent node is either
 - Greater than or equal to the value of child (Max Heap)
 - Less than or equal to the value of child (Min Heap)
 - A complete binary tree



Max Heap : Representation

- Can adopt “**Array Representation**”

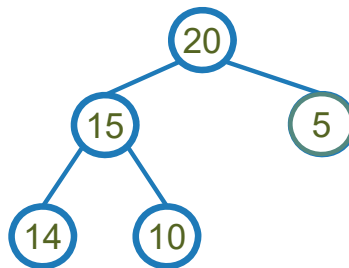
- Since it is a complete binary tree

- Let node i be in position i (array[0] is empty)

- $\text{Parent}(i) = i / 2$ if $i \neq 1$. If $i=1$, i is the root and has no parent
- $\text{leftChild}(i) = 2i$ if $2i \leq n$. If $2i > n$, the i has no left child.
- $\text{rightChild}(i) = 2i+1$ if $2i+1 \leq n$, if $2i+1 > n$, the i has no right child

Max Heap : Insert

- Insert new node
- Make sure it is a complete binary tree
- Check if the new node is greater than its parent
 - If so, swap two nodes



Max Heap : Insert Codes

```
template < class T >
void MaxPQ<T>::Push(const T& e)
{ // Insert e into max heap
  // Make sure the array has enough space here...
  // ...
  int currentNode = ++heapSize;
  while(currentNode != 1 && heap[currentNode/2] < e)
  { // Swap with parent node
    heap[currentNode]=heap[currentNode/2];
    currentNode /= 2; // currentNode now points to parent
  }
  heap[currentNode]=e;
}
```

■ Time Complexity:

Max Heap : Delete

■ Priority Queues

- The element to be deleted is the one with highest priority

■ In priority queues

1. Always delete the root
2. Move the last element to the root (maintain a complete binary tree)
3. Swap with larger and largest child (if any)
4. Continue step 3 until the max heap is maintained (trickle down)

Max Heap : Delete Codes

```
template < class T >
void MaxPQ<T>::Pop()
{ //Delete max element
  if(IsEmpty()) throw "Heap is empty";
  heap[1].~T(); // delete max element (always the root!)
  // Remove the last element from heap
  T lastE = heap[heapSize--];

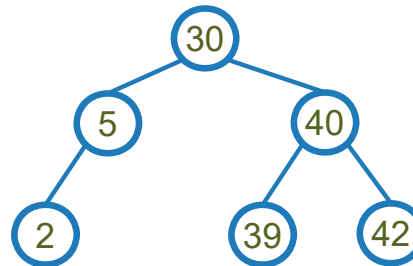
  // trickle down
  int currentNode = 1; // root
  int child = 2; // A child of currentNode
  while(child <= heapSize) {
    // Set child to larger child of currentNode
    if (child < heapSize && heap[child] < heap[child + 1]) child++;

    // Can we put lastE in currentNode?
    if (lastE >= heap[child]) break; // Yes!

    // No!
    heap[currentNode] = heap[child]; // Move child up
    currentNode = child; child *=2; // Move down a level
  }
  heap[currentNode] = lastE;
}
```

Binary Search Tree (BST)

- A binary search tree (BST) is a binary tree which satisfies:
 - Every element has a key
 - No two elements have the same key
 - The keys in the left subtree are smaller than the key in the root
 - The keys in the right subtree are larger than the key in the root
 - The left and right subtrees are also BST



BST : Operations

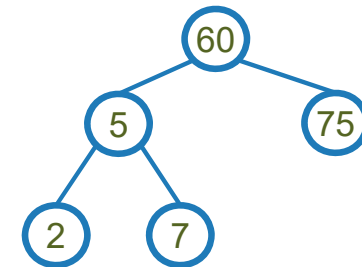
- Search an element in a BST
- Search for the r^{th} smallest element in a BST
- Insert an element into a BST
- Delete max/min from a BST
- Delete an arbitrary element from a BST

BST : Search an Element

■ Search for key 7

■ Search process

1. Start from root
2. Compare the key with root
 - '<' search the left subtree
 - '>' search the right subtree
3. Repeat step 3 until the key is found or a leaf is visited



BST : Recursive Search Codes

```
template < class K, class E >
pair<K,E>* BST<K,E>::Get(const K& k)
{ // Search the BST for a pair with key k
  // If the this pair is found, return a pointer to this
  // pair, otherwise return 0
  return Get(root, k);
}

template < class K, class E >
pair<K,E>* BST<K,E>::Get(TreeNode<pair<K,E>>* p, const K& k)
{
  if(!p) return 0;
  if(k < p->data.first) return Get(p->leftChild, k);
  if(k > p->data.first) return Get(p->rightChild, k);
  return &p->data;
}
```

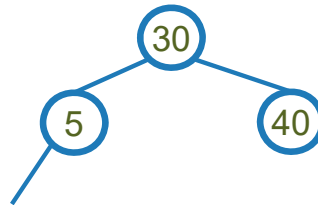
BST : Iterative Search Codes

```
template < class K, class E >
pair<K,E>* BST<K,E>::Get(const K& k)
{
    TreeNode < pair<K, E> > *currentNode = root;
    while (currentNode) {
        if (k < currentNode->data.first)
            currentNode = currentNode->leftChild;
        else if (k > currentNode->data.first)
            currentNode = currentNode->rightChild;
        else return & currentNode->data;
    }
    return NULL; // no match found
}
```

BST : Search an Element by Rank

- Definition of **rank**:

- A **rank** of a node is its position in inorder traversal



BST : Search by Rank Codes

- For each node, we store “leftSize”
 - which is 1 + (# of nodes in the left subtree)

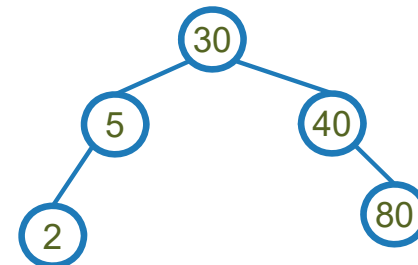
```
template < class K, class E >
pair<K,E>* BST<K,E>::RankGet(int r)
{ // Search BST for the rth smallest pair
  TreeNode<pair<K,E>>* currentNode = root;
  while(currentNode) {
    if(r < currentNode->leftSize)
      currentNode = currentNode->leftChild;
    else if(r > currentNode->leftSize) {
      r -= currentNode->leftSize;
      currentNode = currentNode->rightChild;
    }
    else return &currentNode->data;
  }
  return 0;
}
```

BST : Insert

■ To insert an element with key 80

■ Search process

1. Search for the existence of the element
2. If the search is unsuccessful, then the element is inserted at the point the search terminates

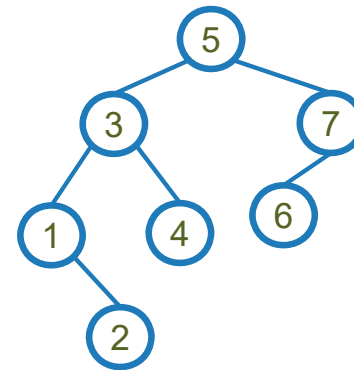
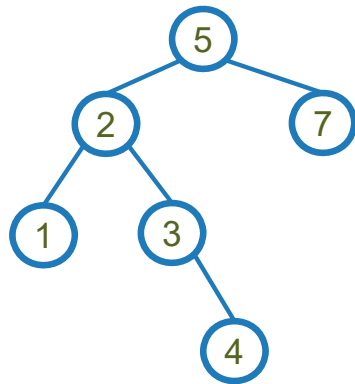


BST : Insert Codes

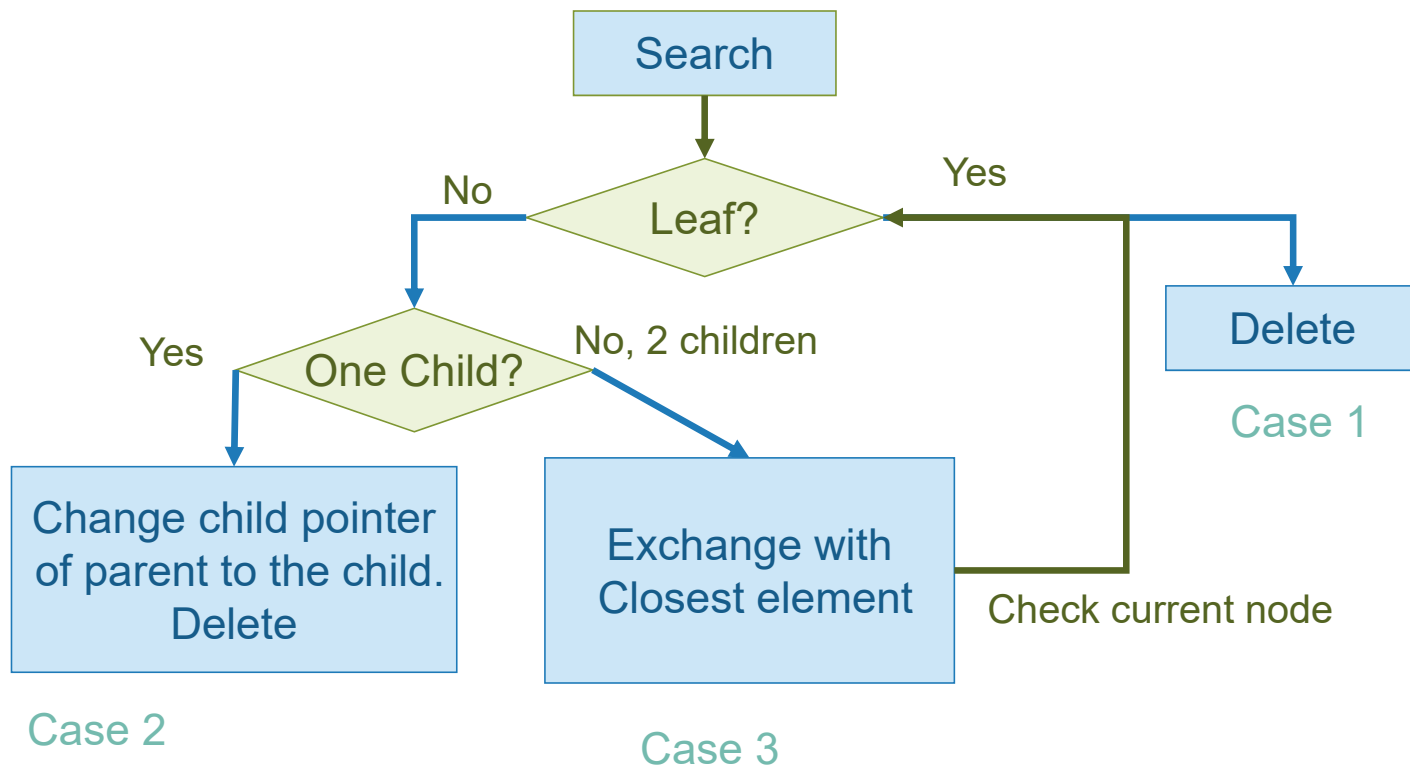
```
template < class K, class E >
void BST<K,E>::Insert(const pair<K,E>& thePair)
{ // Search for key "thePair.first", pp is the parent of p
  TreeNode<pair<K,E>>* p = root, *pp=0;
  while(p){
    pp = p;
    if(thePair.first < p->data.first)
      p = p->leftChild;
    else if(thePair.first > p->data.first)
      p = p->rightChild;
    else // Duplicate, update the value of element
      { p->data.second = thePair.second; return; }
  }
  // Perform the insertion
  p = new pair<K,E>(thePair);
  if(root) // tree is not empty
    if(thePair.first < pp->data.first) pp->leftChild = p;
    else pp->rightChild = p;
  else root = p;
}
```


Min (Max) Element in BST

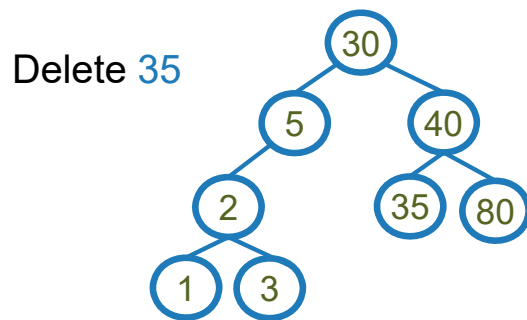
- Min (Max) element is at the leftmost (rightmost) one
- Min or max are not always terminal nodes
- Min or max has at most one child



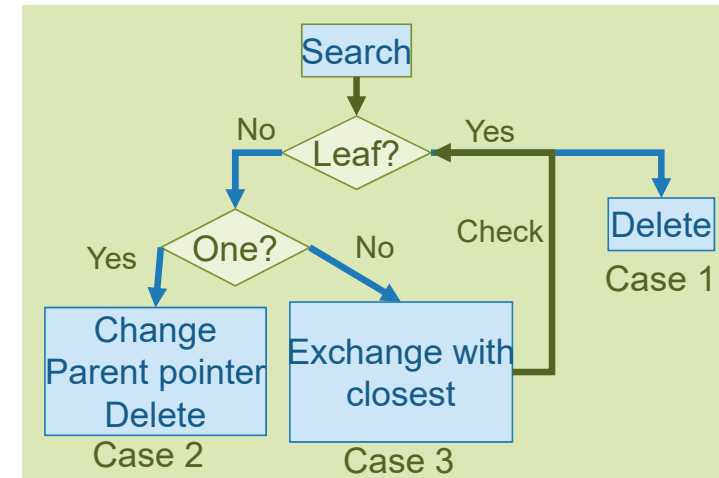
BST: Flow Chart of Deletion



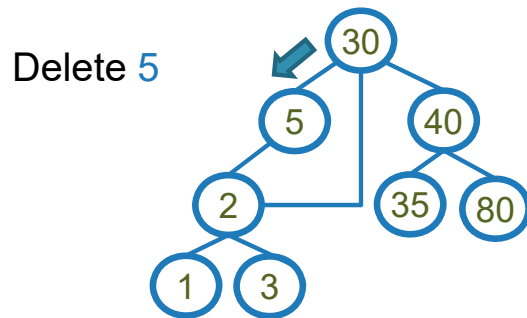
BST: Delete (Case1)



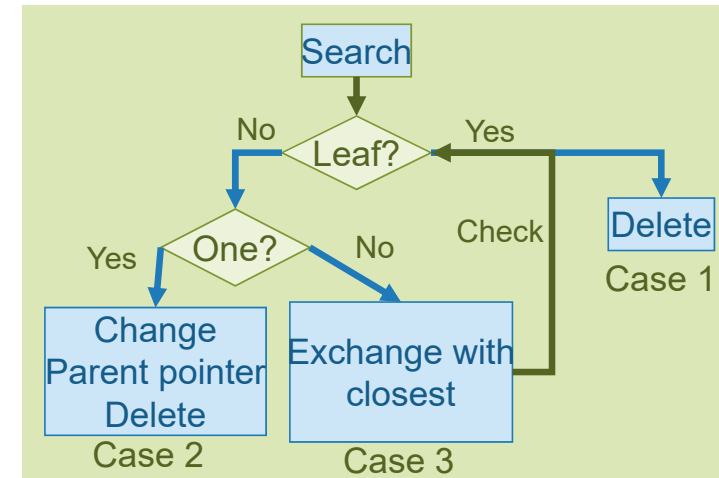
- Case 1 : The element is a leaf node
- The child field of parent node is set to NULL
- Dispose the node



BST: Delete (Case2)

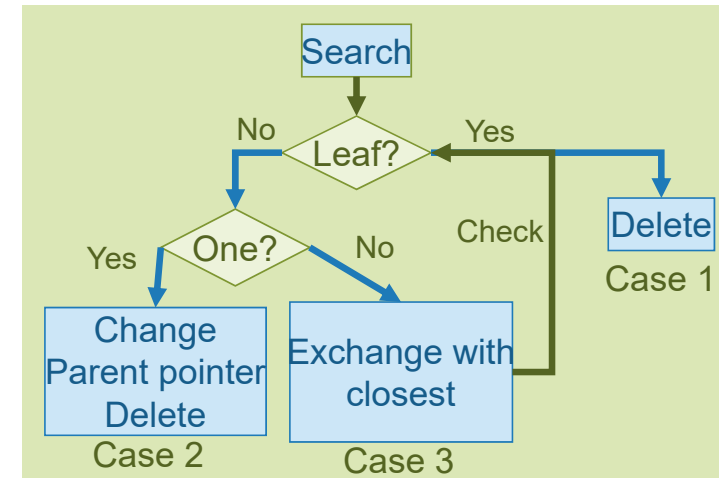
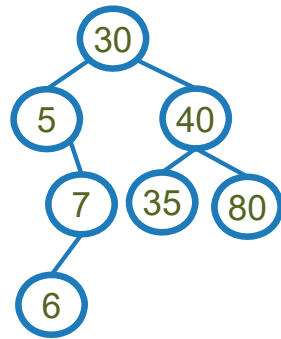


- Case 2 : The element is a non-leaf node with one child
- Change the pointer from the parent node to the single-child node
- Dispose the node



BST: Delete (Case3)

Delete 30



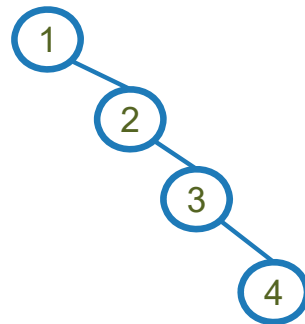
- Case 3 : The element is a non-leaf node with two children
- The deleted element is replaced by the closest one, either
 - The smallest element in right subtree
 - The largest element in left subtree

BST : Time Complexity

- Search, insertion, or deletion takes $O(h)$
- h = Height of a BST

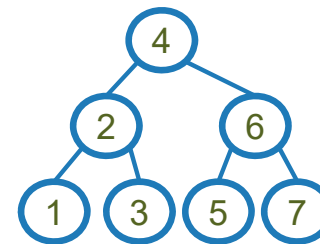
■ Worst case $h=n$

- Insert keys: 1, 2, 3, 4, ..



■ Best case $h=\log_2 n$

- Insert keys : 4, 2, 6, 1, 3, 5, 7



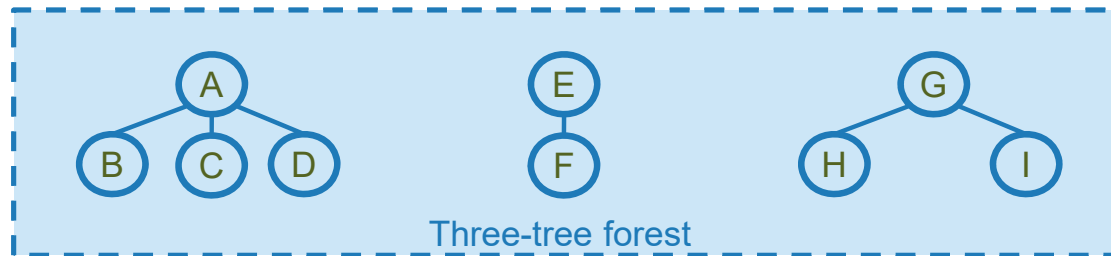
BST depends on how elements are inserted and deleted from the tree

Self-Study Topics

- Write pseudo codes of BST deletion
- Selection trees

Forests

■ Definition : A forest is a set of $n \geq 0$ disjoint trees

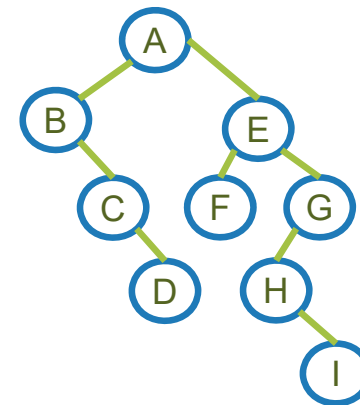
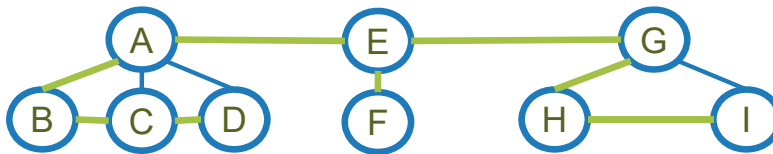


■ Operations :

- Transforming a forest to binary tree
- Forest traversals

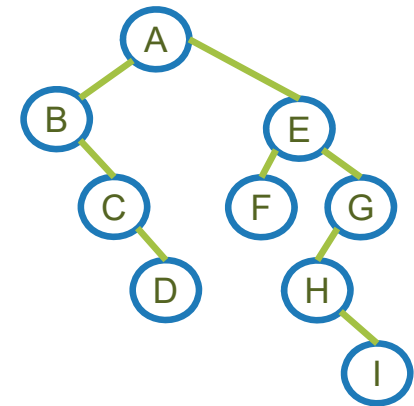
Transforming a Forest to Binary Tree

- Apply left child-right sibling approach
 - Convert each tree into binary tree
 - Connect two binary trees, T_1 and T_2 , by setting the rightChild of root(T_1) to the root(T_2)



Forest Traversals

- Assume we have a forest **F** and binary tree **T**
- The following are equivalent
 - *Preorder* traversal of **T**
 - A B C D E F G H I
 - Visiting the nodes of **F** in *forest preorder*
 - Root: A
 - Left forest: B C D
 - Right forest: E F G H I



Disjoint Sets

- Assume a set S of n integers $\{0, 1, 2, \dots, n - 1\}$ is divided into several subsets S_1, S_2, \dots, S_k
- $S_i \cap S_j = \emptyset$ for any $i, j \in \{1, \dots, k\}$ and $i \neq j$
- Operations:
 - Union disjoint sets: **Union** (S_i, S_j)
 - $S_i = S_i \cup S_j$ or $S_j = S_i \cup S_j$
 - Find the set containing element x : **Find**(x)

Disjoint Sets : Example

- Set

- $S = \{ 0, 1, 2, 3, 4, 5 \}$

- Disjoint subsets

- $S_1 = \{ 0, 2, 3 \}$

- $S_2 = \{ 1 \}$

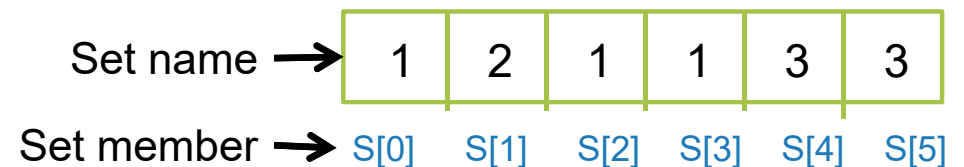
- $S_3 = \{ 4, 5 \}$

- $\text{Union}(S_1, S_2) = \{ 0, 1, 2, 3 \}$

- $\text{Find}(5) = 3$

DS: Array Representation

- $S = \{0, 1, 2, 3, 4, 5\}$ with subsets
 - $S_1 = \{0, 2, 3\}$, $S_2 = \{1\}$ and $S_3 = \{4, 5\}$
- Using a sequential mapping array
 - Index represents set members
 - Array value indicates set name



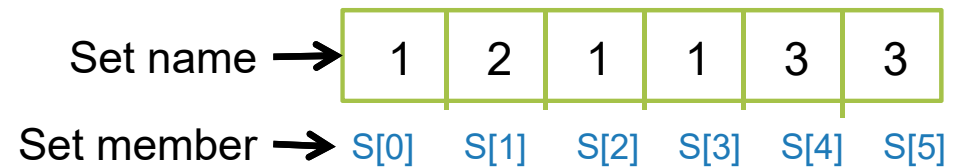
DS Operation: Find(x)

■ Find the set which contains element x is easy

■ Find(5) = S[5] = set 3

Find(3) = S[3] = set 1

■ Complexity = $O(1)$



DS Operation: Union(S_i, S_j)

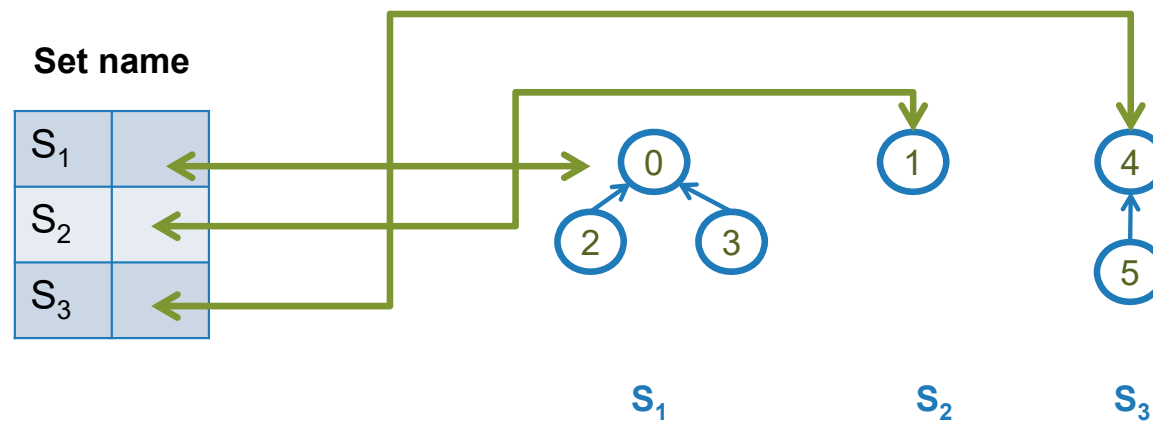


- Assume we always merge the 2nd set to 1st set
 - $S_i = S_i \cup S_j$
- Scan the array and set $S[k]$ to i if $S[k] == j$
 - $S_2 = \text{Union}(S_2, S_3)$



DS: Tree Representation

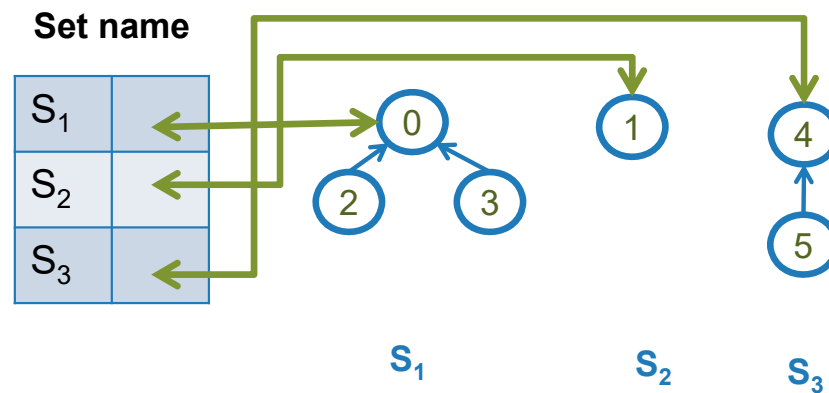
- Link elements of a subset to form a tree
 - Link children to root
 - Link root to set name



DS: Tree Representation

- Use an array to store the tree
- Identify the set by the root of the tree

$S_1 = \{0, 2, 3\}$, $S_2 = \{1\}$ and $S_3 = \{4, 5\}$

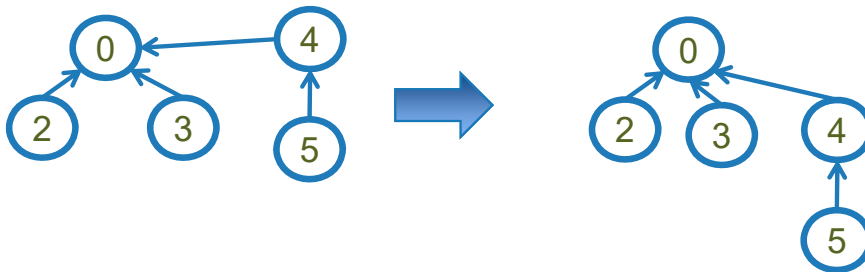


T[0]	-1
T[1]	-1
T[2]	0
T[3]	0
T[4]	-1
T[5]	4

DS Operation: Union(S_i, S_j)

■ Set the parent field of one of the root to the other root

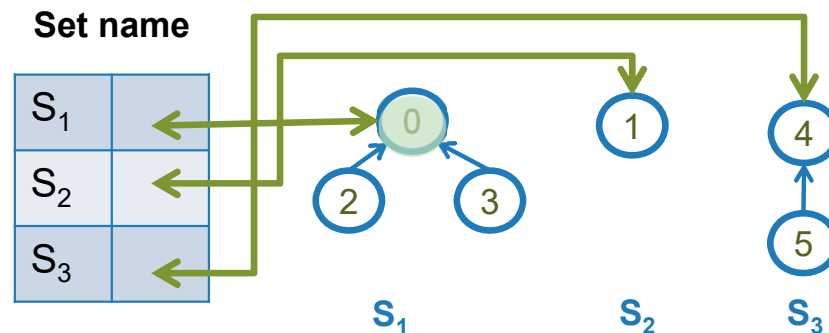
- $S_1 = \text{Union}(S_1, S_3)$
- Time complexity : $O(1)$



T[0]	-1
T[1]	-1
T[2]	0
T[3]	0
T[4]	0
T[5]	4

DS Operation: Find(x)

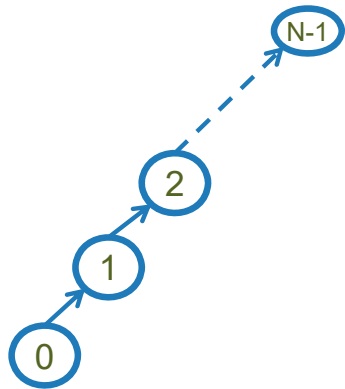
- Following the index starting at x
- Tracing the tree structure
 - Until reaching a node with parent value = -1
- Use the root to identify the set name



T[0]	-1
T[1]	-1
T[2]	0
T[3]	0
T[4]	-1
T[5]	4

DS Time Complexity

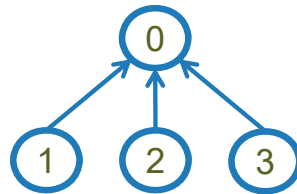
- $S = \{ 0, 1, 2, \dots, n-1 \}$
 - $S_1 = \{ 0 \}, S_2 = \{ 1 \}, S_3 = \{ 2 \}, \dots, S_n = \{ n-1 \}$
- Perform a sequence Union
 - $\text{Union}(S_2, S_1), \text{Union}(S_3, S_2), \dots, \text{Union}(S_n, S_{n-1})$



Followed by a sequence of Find
 $\text{Find}(0), \text{Find}(1), \dots, \text{Find}(n-1)$

Improved Union(S_i, S_j)

- Do not always merge two sets into the first set
- Adopt a **Weighting rule** to union operation
 - $S_i = S_i \cup S_j$, if $|S_i| \geq |S_j|$
 - $S_j = S_i \cup S_j$, if $|S_i| < |S_j|$
- $S = \{ 0, 1, 2, \dots, n \}$
 - $S_1 = \{ 0 \}$, $S_2 = \{ 1 \}$, $S_3 = \{ 2 \}$, \dots , $S_n = \{ n-1 \}$
 - Union (1, 2) \rightarrow Union (1, 3) \rightarrow Union (1, 4)

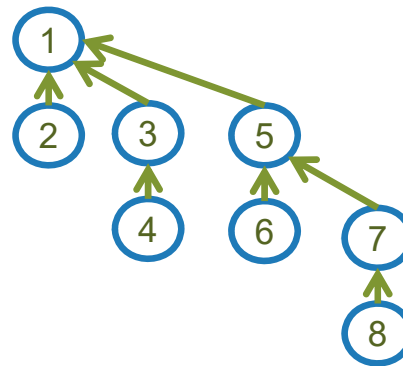


Time Complexity

- The following sequence produces the height of $\log n$



- Union(1, 2)
- Union(3, 4)
- Union(5, 6)
- Union(7, 8)
- Union(1, 3)
- Union(5, 7)
- Union(1, 5)



Improved Find(x)

- Adopt a **Collapsing rule** for find(x)
 - If j is a node on the path from i to the root, set $\text{parent}[j]$ to $\text{root}(i)$

