CHAPTER 5
TREES PART I

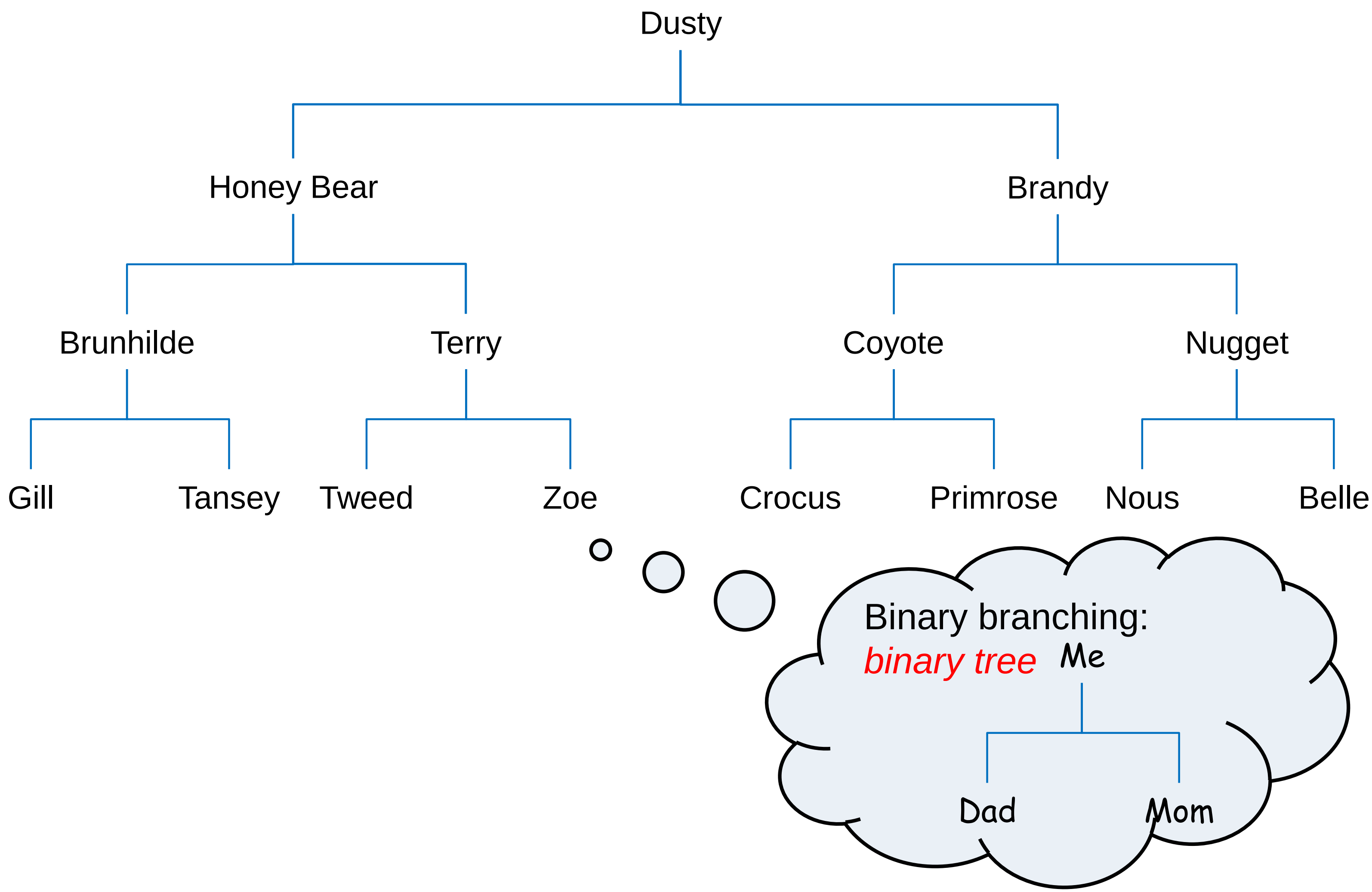
Iris Hui-Ru Jiang

Fall 2008

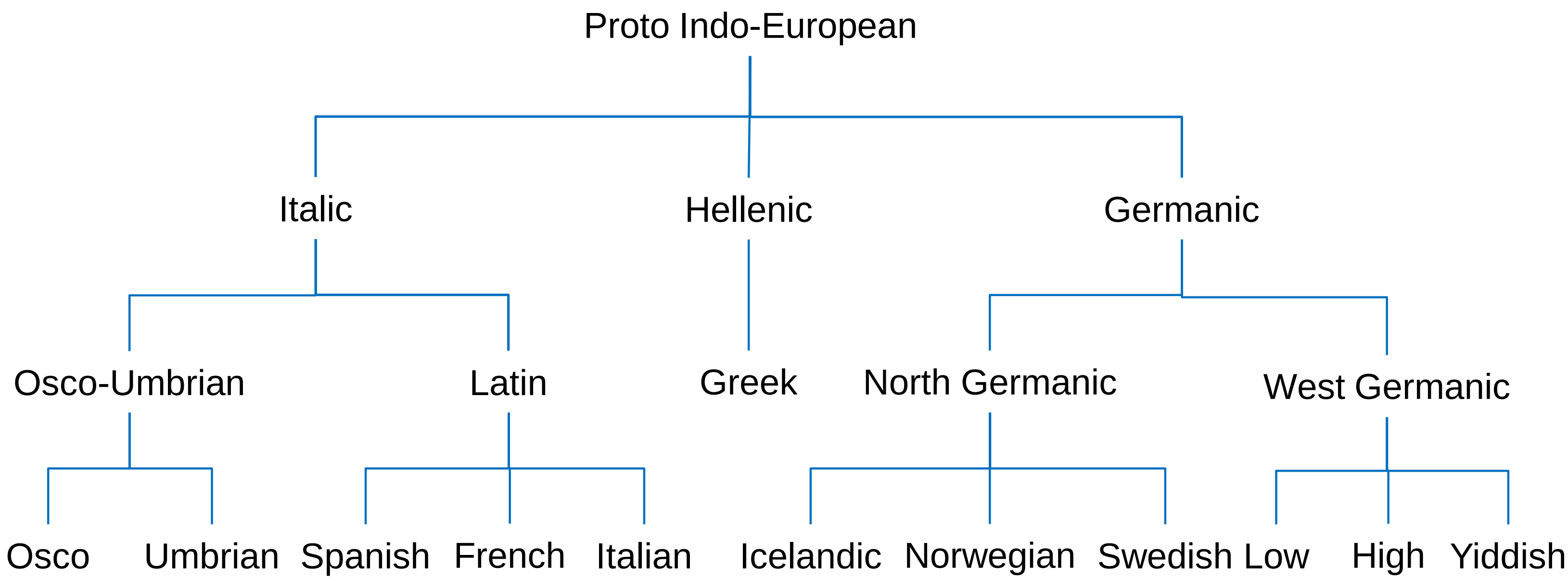
Trees Part I

- **Contents**
  - Trees
  - Binary trees
  - Threaded binary trees
  - Heaps
  - Binary search trees
  - Selection trees
  - Forests
  - Disjoint sets
- **Readings**
  - Chapter 5
  - Section 3.4, 7.6

Genealogical Charts -- Pedigree

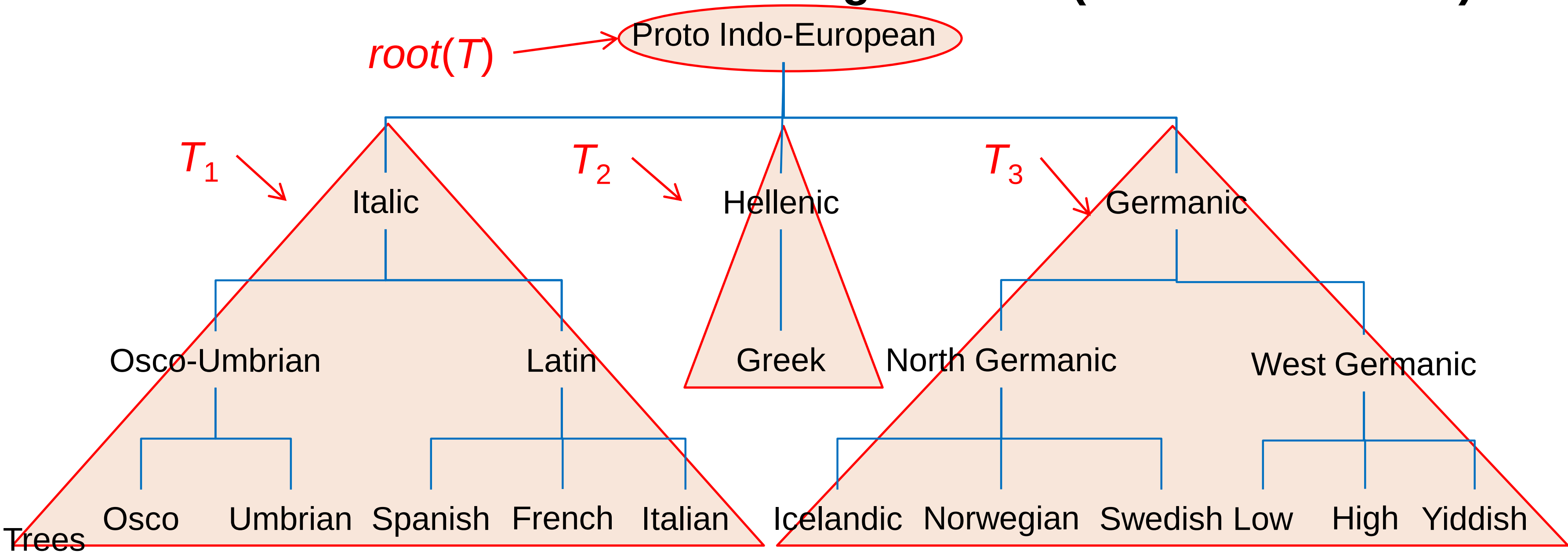


Genealogical Charts -- Lineal

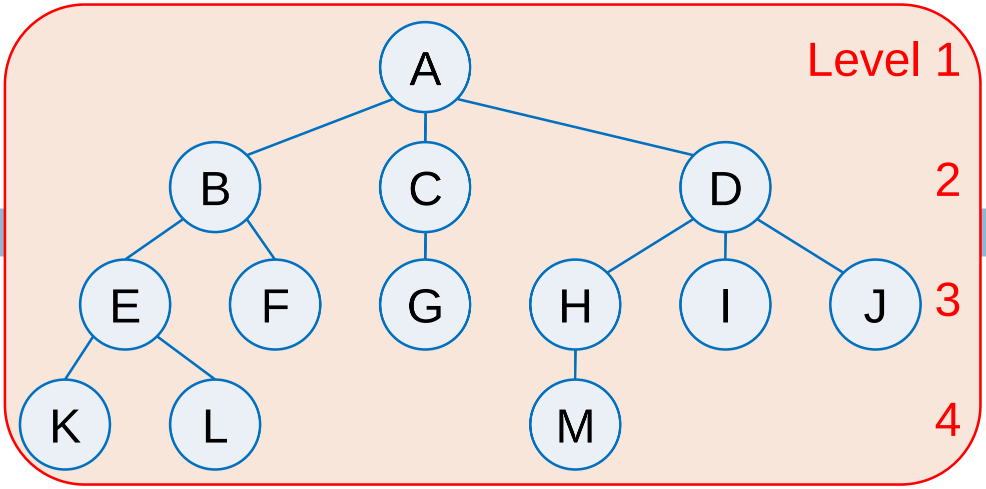


Trees

- **Organize data in a hierarchical manner**
- **Definition: A tree *T* is**
  - A finite set of one or more nodes s.t.
    - There is one root, *root*(*T*)
    - The remaining nodes are partitioned into $n \geq 0$ disjoint sets, T_1, …, T_n,. T_i is also a tree. T_1, …, T_n are the subtrees of *root*(*T*).
- **Recursive definition ⇒ recursive algorithms! (ref. Section 1.5.2)**



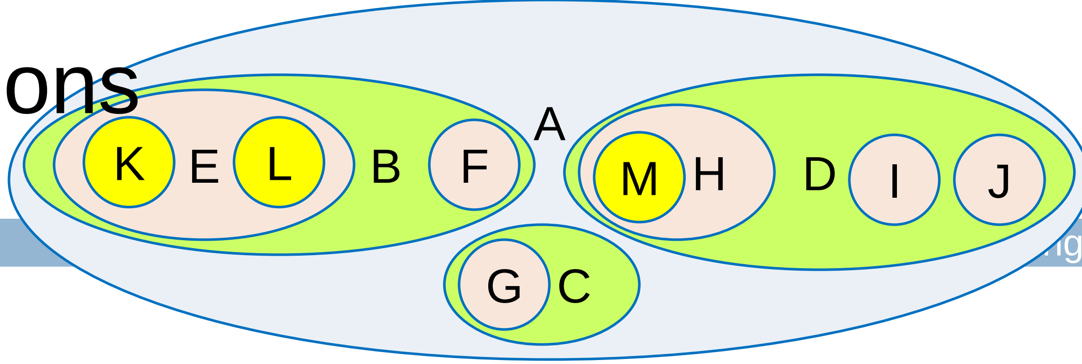
Terminology



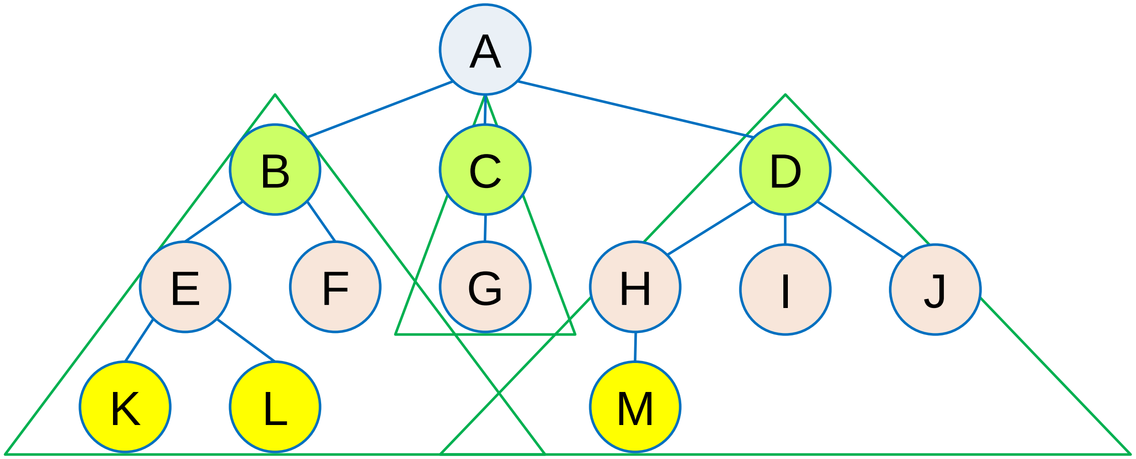
- **# of nodes:** 13
- **root:** *A*
- **degree:** # of subtrees of a node: *degree*(*A*) = 3
- **degree of tree:** max degree of nodes: 3
- **leaf (terminal node):** node of degree 0: {*K*, *L*, *F*, *G*, *M*, *I*, *J*}
- **nonterminal:** not leaf: {*A*, *B*, *C*, *D*, *E*, *H*}
- **parent/children:** *parent*(*D*) = *A*, *children*(*D*) = {*H*, *I*, *J*}
- **siblings:** nodes with same parent: {*H*, *I*, *J*}
- **ancestors:** all the nodes along the path from root to it: *ancestors*(*M*) = {*A*, *D*, *H*}
- **level of root:** 1 (cf. *level*(*root*) = 0 in some books)
- **level of a node:** level of its parent + 1: *level*(*H*) = 3
- **height (depth):** max level of nodes: 4

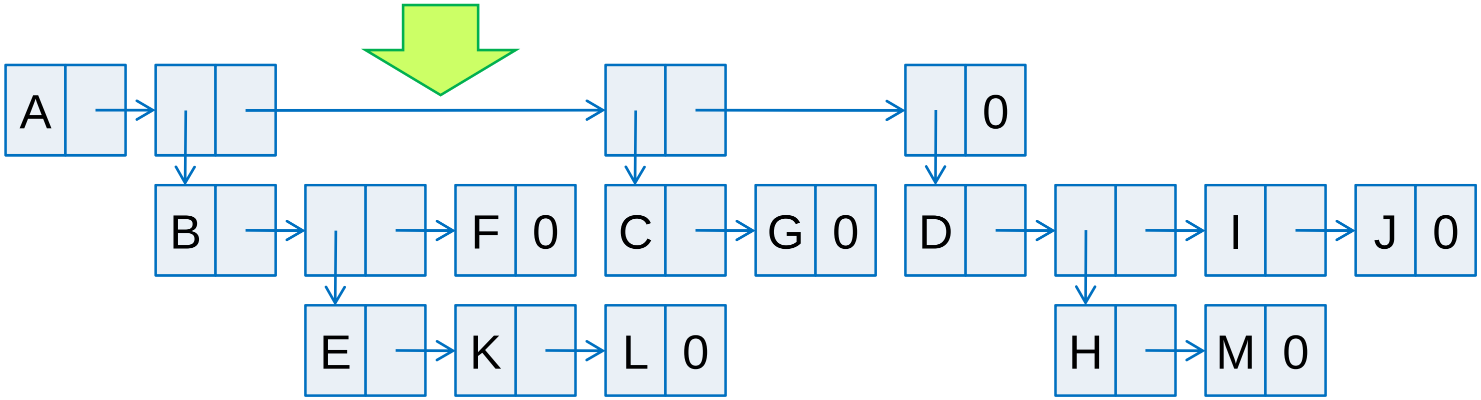
Tree Representations

-- List



- **Parenthesization:**
  - Root comes first, followed by a list of subtrees
  - (*A* (*B* (*E* (*K*, *L*), *F*), *C* (*G*), *D* (*H* (*M*), *I*, *J*)))





Tree Representations
-- *k*-ary Tree Node

_k_-ary tree: tree degree = _k_
A node has at most *k* children

- **For a _k_-ary tree of _n_ nodes, represent each tree node by:**

data	*child* 1	*child* 2	…	*child k*

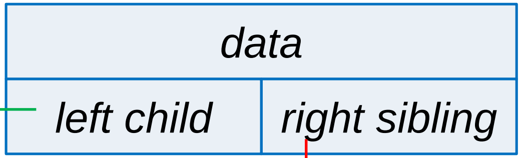
- **Waste memory!** **Why?**
  - **Lemma:** If *T* is a *k*-ary tree with *n* nodes, then $n(k–1)+1$ of the nk child fields are 0, $n \geq 1$
  - **Proof:**

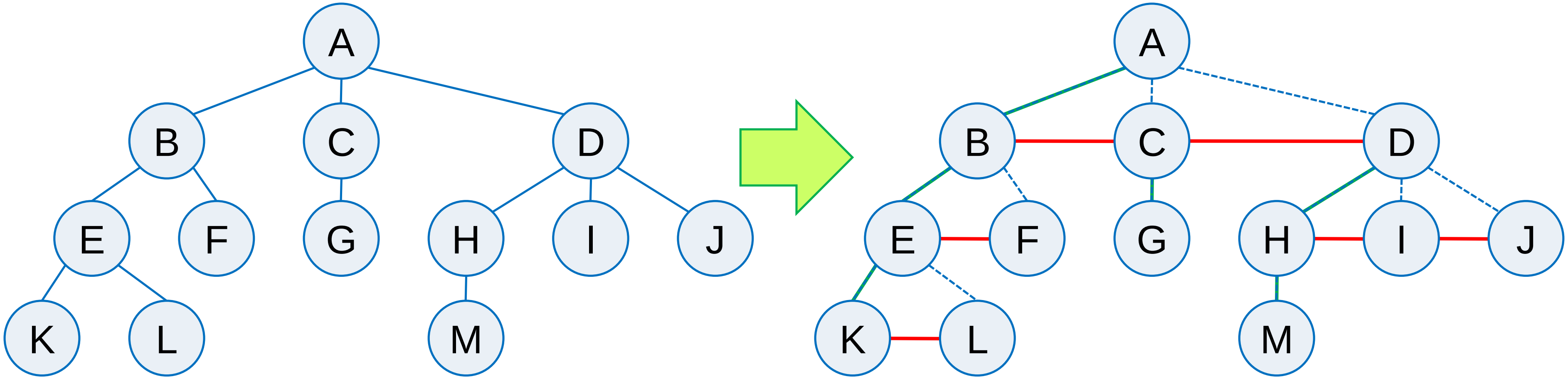
  1) Each node has *k* child fields
     - $\Rightarrow nk$ child fields in total
  2) Except root, each of the rest $n – 1$ nodes is someone's child
     - $\Rightarrow n – 1$ child fields actually in use
  3) $\Rightarrow nk – (n – 1) = n(k–1)+1$ zero fields ■

  - What if k=1? k=2? k is large?

Tree Representations
-- Left Child - Right Sibling

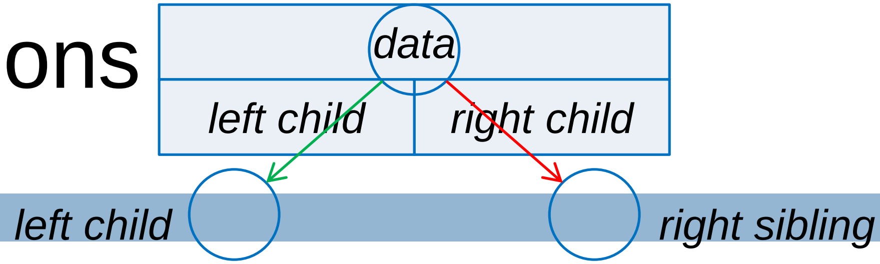
- **Each node has**
  - At most one leftmost child
  - At most one closest right sibling



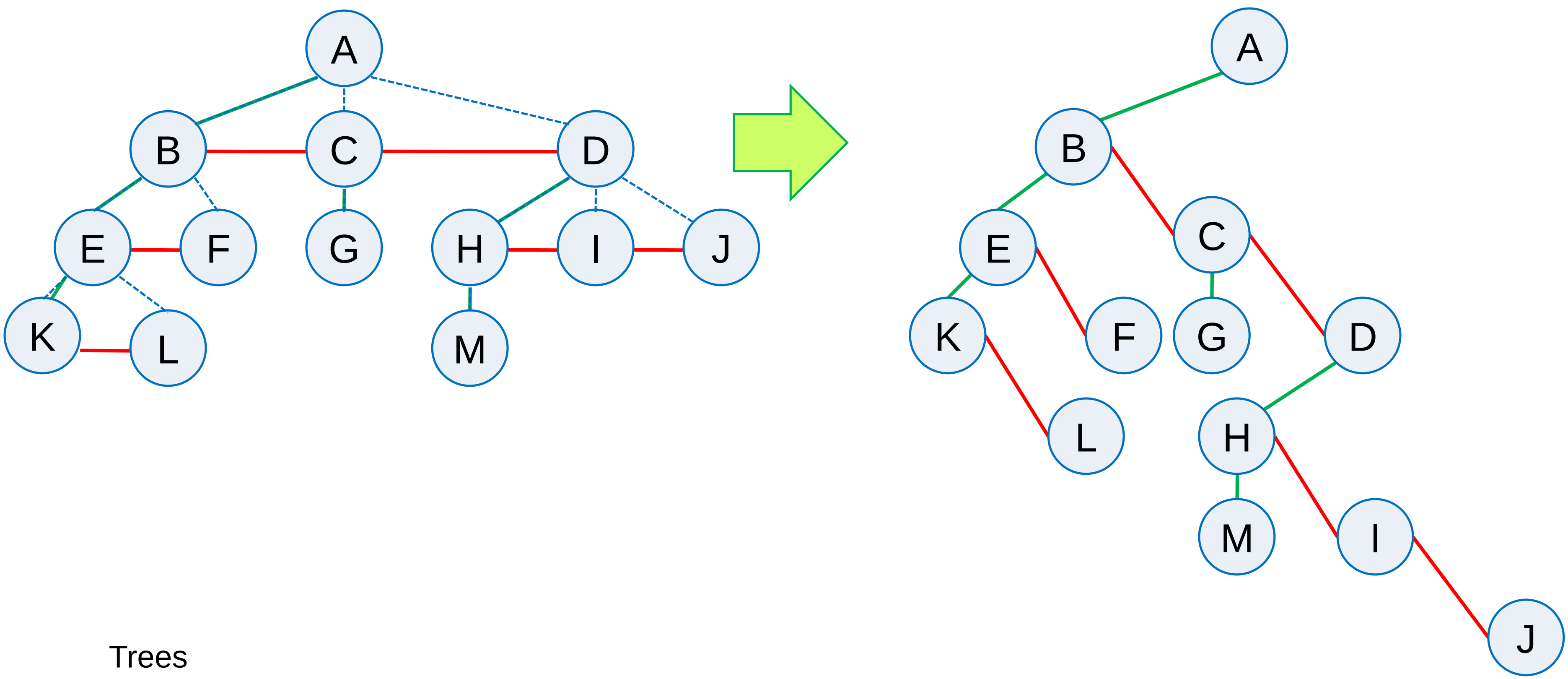


Tree Representations

-- Degree-Two Tree



- **Rotate the right sibling clockwise by 45 degrees**
  - ⇒ Left child-right child, degree-two, binary trees
- **Represent any tree as a binary tree**
  - Why? Regular and efficient!



Binary Trees

Basic concepts

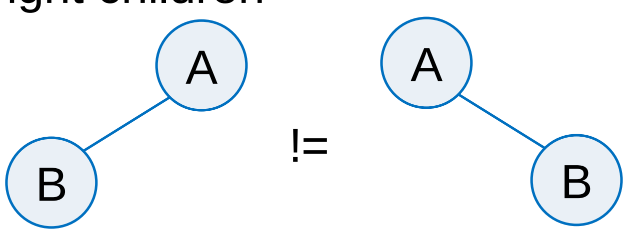
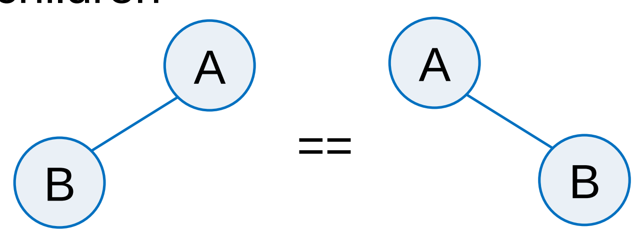
Representations

Traversal

Binary Trees

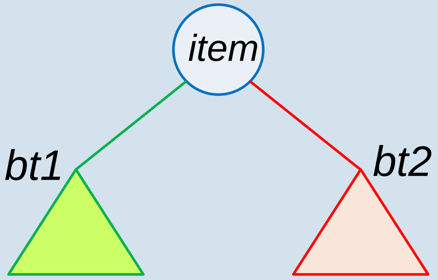
binary tree: tree degree = 2
A node has at most 2 children

- **Definition: A binary tree *T* is**
  - A finite set of nodes that
    - either is empty
    - or consists of one root and two disjoint binary trees
      - the left subtree and the right subtree
- **Recursive definition!**
- **A binary tree is not a special case of tree**

A binary tree	**A tree**
Can have zero nodes	Has at least one node (root)
Distinguishes between left and right children 	Does not care the order of children 

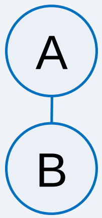
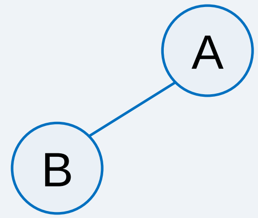
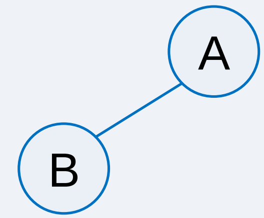
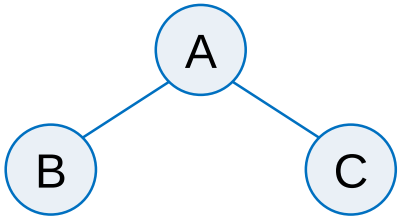
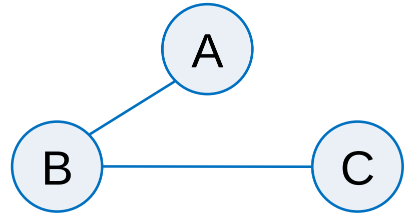
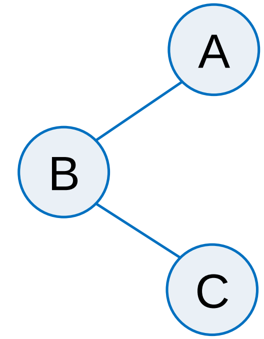
ADT *BinaryTree*


```
template <class T>
class BinaryTree {
// objects:  A finite set of nodes either empty
//           or consisting of a root node, left BinaryTree and right BinaryTree
public:
    BinaryTree ();                      // ctor: creates an empty binary tree
    ~BinaryTree();                      // dtor

    bool IsEmpty();                     // return true iff empty

    BinaryTree(BinaryTree <T>& bt1, T& item, BinaryTree<T>& bt2);
                                        // ctor: creates a binary tree whose root node contains item,
                                        // whose left subtree is bt1, whose right subtree is bt2
```



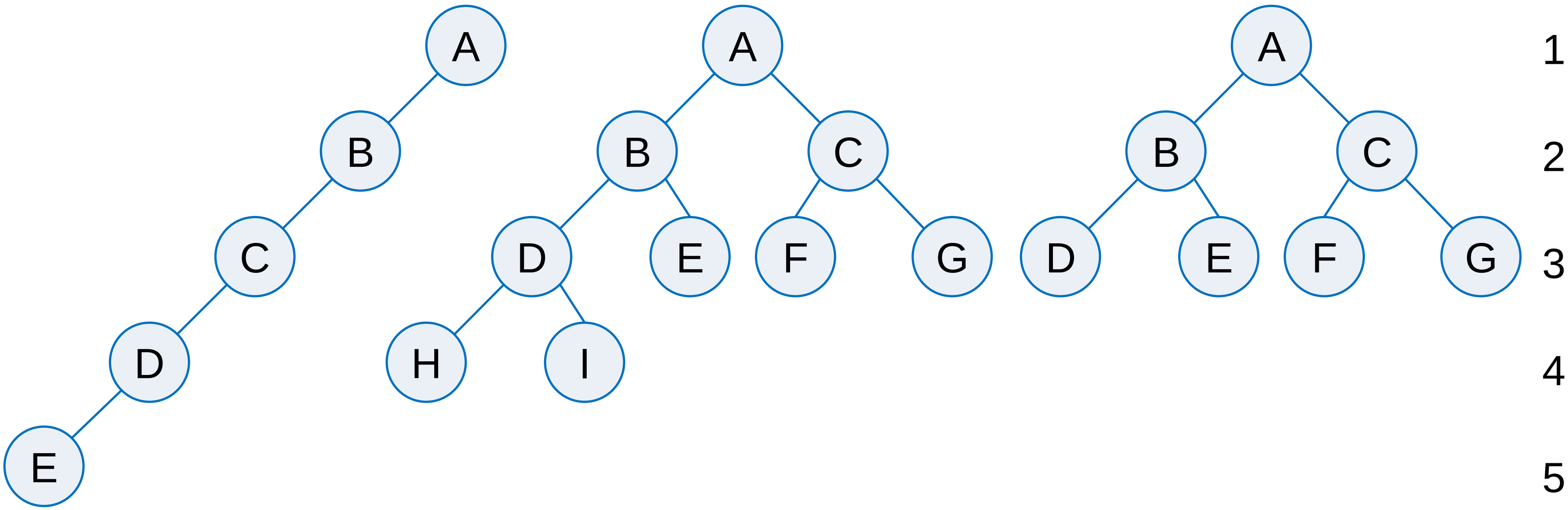

```
    BinaryTree <T> LeftSubtree();       // return the left subtree of *this
    BinaryTree <T> RightSubtree();      // return the right subtree of *this
    T RootData();                       // return the data in the root of *this
};
```

Reusing Tree Representations

Tree	Left child-right sibling tree	Binary tree
		
		

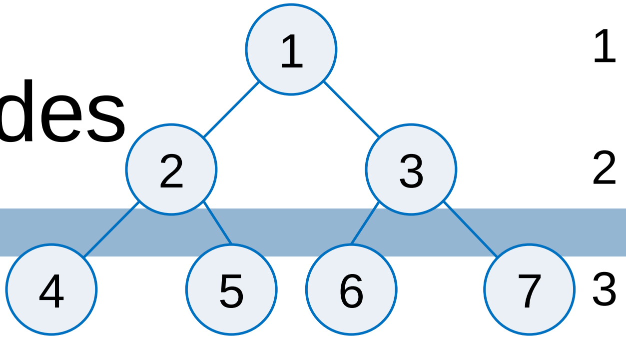
Skewed vs. Complete vs. Full

Skewed	Complete	Full	Level



Skewed	Complete	Full	Level
A tree skews either to left or to right.	Except the last level, each level is fully occupied; the last level is sequentially occupied.	Each level is fully occupied.	

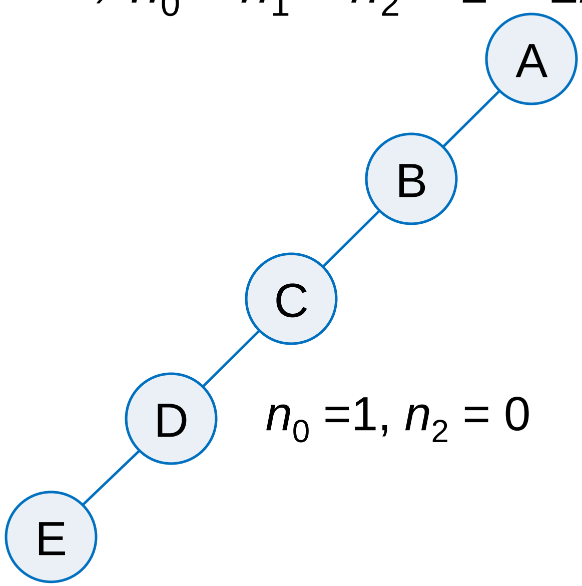
Maximum Number of Nodes



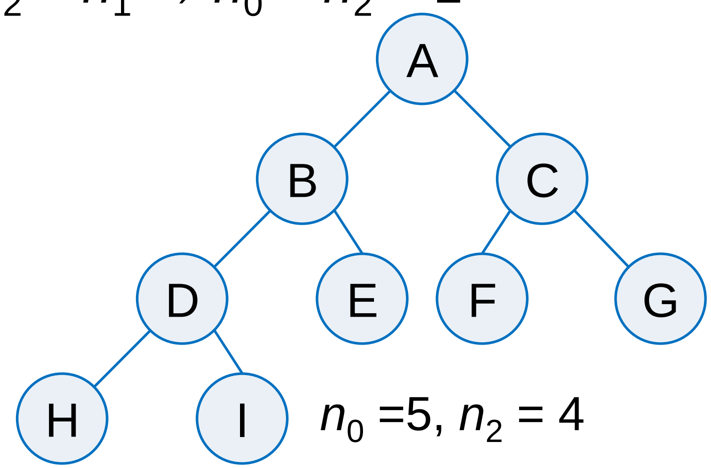
- **Lemma:**
  1. The max # of nodes on level i of a binary tree is 2^{i-1}, $i \geq 1$
  2. The max # of nodes in a binary tree of depth k is $2^k - 1$, $k \geq 1$
- **Proof:**
  1. By induction on i!
     1) Induction base: i=1, trivial!
     2) Induction hypothesis: For $i > 1$, assume the max # of nodes on level $i - 1$ is 2^{i-2}
     3) Induction step: Since each node has at most 2 children, the max # of nodes on level i should be $2*2^{i-2} = 2^{i-1}$ ■
  2. By 1., the max # of nodes in a binary tree of depth k is $\sum_{i=1..k}$(max # of nodes on level i) $= \sum_{i=1..k} 2^{i-1} = 2^k - 1$ ■
- **Q: Consider a binary tree with *n* nodes, what is the range of its depth *d*?**
  - $\lceil \lg (n+1) \rceil \leq d \leq n$

Leaf Nodes and Degree-2 Nodes

- **Lemma:** For any nonempty binary tree, *T*, if n_0 is the # of leaf nodes and n_2 the # of nodes of degree 2, then $n_0 = n_2 + 1$.
- **Proof:**
  1) Let n_1 be the # of nodes of degree 1
  2) # of nodes $n = n_0 + n_1 + n_2$
  3) # of branches $B = n - 1$ (leading into)
  4) $B = 2n_2 + n_1$ (steming from)
  5) $\Rightarrow n_0 + n_1 + n_2 - 1 = 2n_2 + n_1 \Rightarrow n_0 = n_2 + 1$ ■



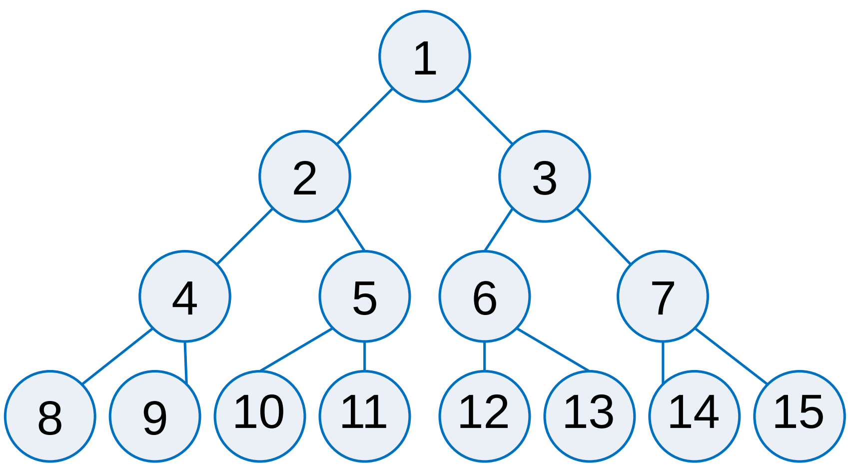
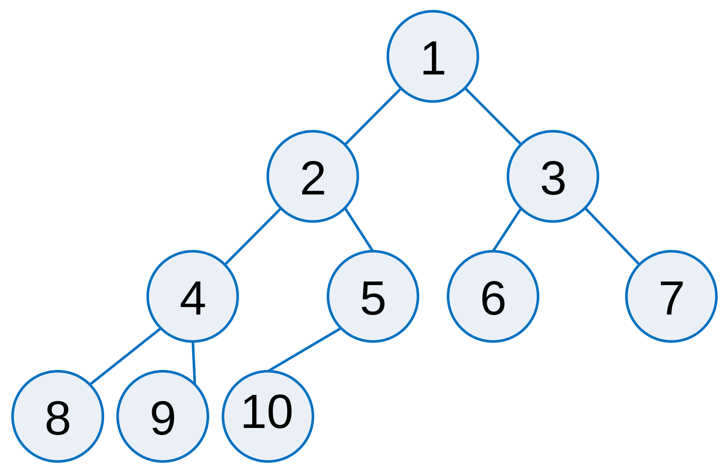
$n_0 = 1$, $n_2 = 0$



$n_0 = 5$, $n_2 = 4$

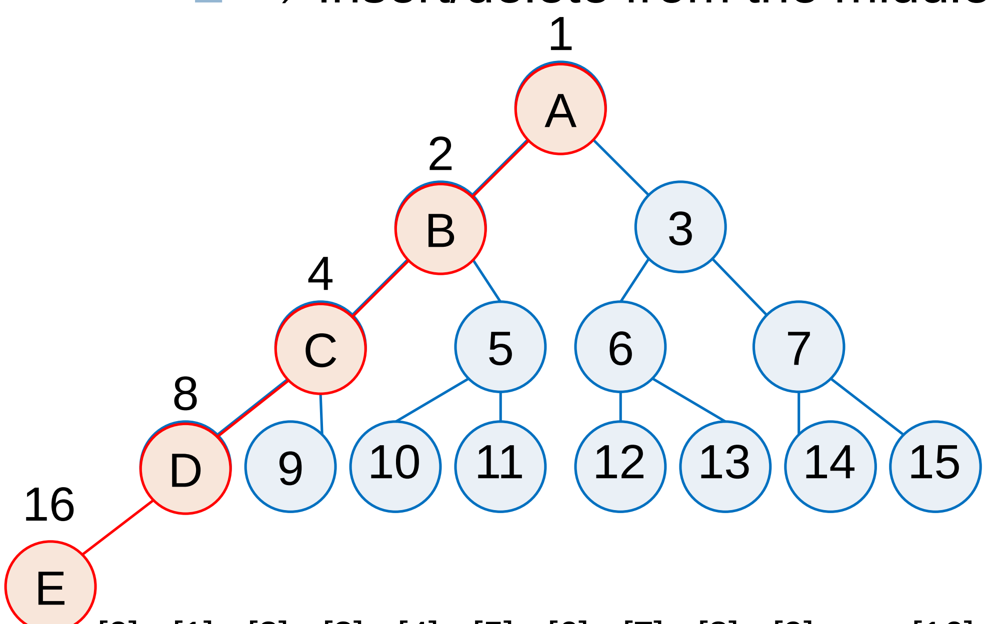
Full and Complete Binary Trees Revisited

- **Definition: A full binary tree of depth *k* is**
  - A binary tree of depth *k* having $2^k - 1$ nodes, $k \geq 0$
- **Definition: A binary tree with *n* nodes & depth *k* is complete iff**
  - Its nodes correspond to the nodes numbered from 1 to *n* in the full binary tree of depth *k*

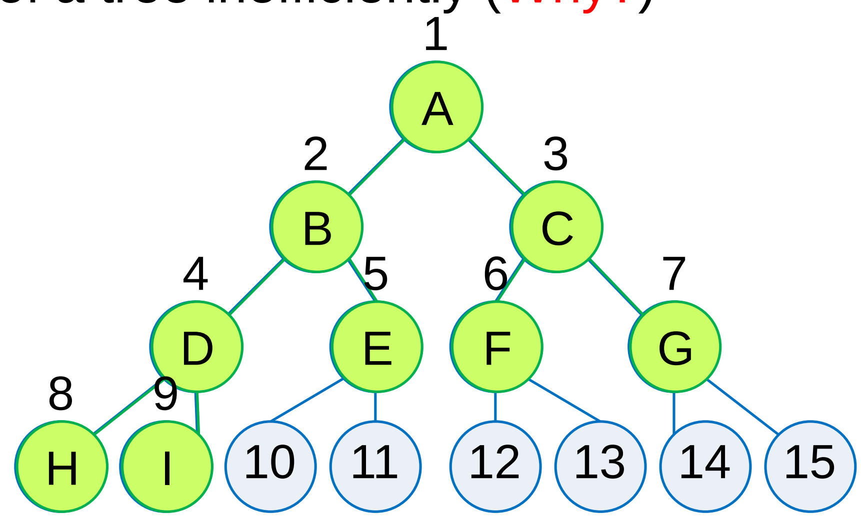
Full	Complete	Level
		1 2 3 4
Each level is fully occupied.	Except the last level, each level is fully occupied; the last level is sequentially occupied.	

Binary Tree Representations
-- Array Representation

- **Use an array to store nodes sequentially**
  - Index nodes with [1..*n*]
  - ⇒ In C++, use *tree*[1] ~ *tree*[*n*] and leave *tree*[0] empty
  - ⇒ Waste memory for incomplete binary trees
  - ⇒ Insert/delete from the middle of a tree inefficiently (Why?)



	[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	…	[16]
tree	-	A	B	-	C	-	-	-	D	-	…	E

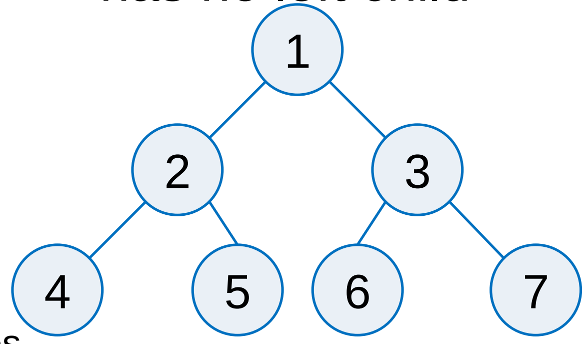


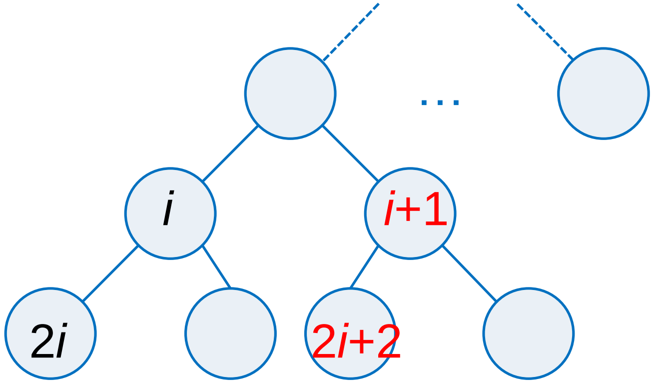
	[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]
tree	-	A	B	C	D	E	F	G	H	I

Trees

Where are Parent & Children in the Array?

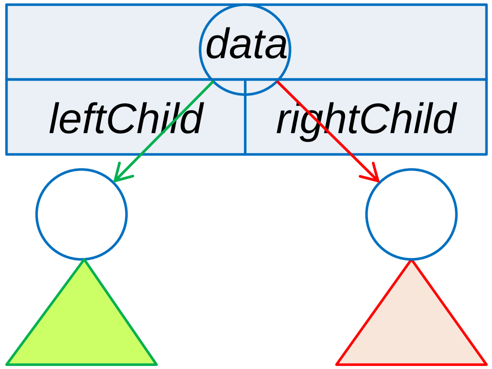
- **Lemma:** If sequentially representing a complete binary tree with n nodes, for any node with index i,
  1. *parent*(i) is at $\lfloor i/2 \rfloor$ if i != 1; if i == 1, i is root and has no parent
  2. *leftChild*(i) is at 2*i if $2i \le n$; or i has no left child
  3. *rightChild*(i) is at 2*i+1 if $2i+1 \le n$; or i has no right child
- **Proof:** 2. ⇒ 3.; 2. & 3. ⇒ 1. Prove 2. by induction!
  1) Induction base: i=1, trivial!
  2) Induction hypothesis: Assume for all j, $1 \le j \le i$, *leftChild*(j) is at $2j$
  3) Induction step: Since *leftChild*(i+1) is after i's children, it is at *leftChild*(i)+2=2i+2=2*(i+1) unless 2(i+1) > n, in which case i+1 has no left child ■



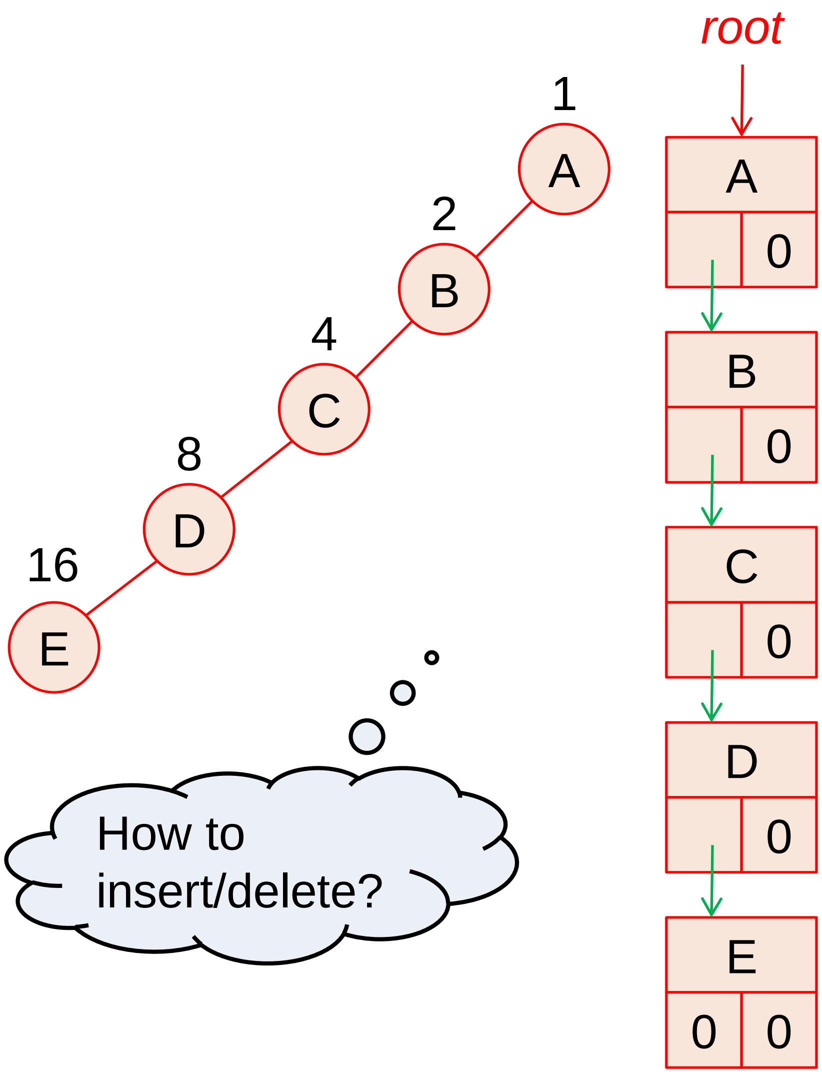


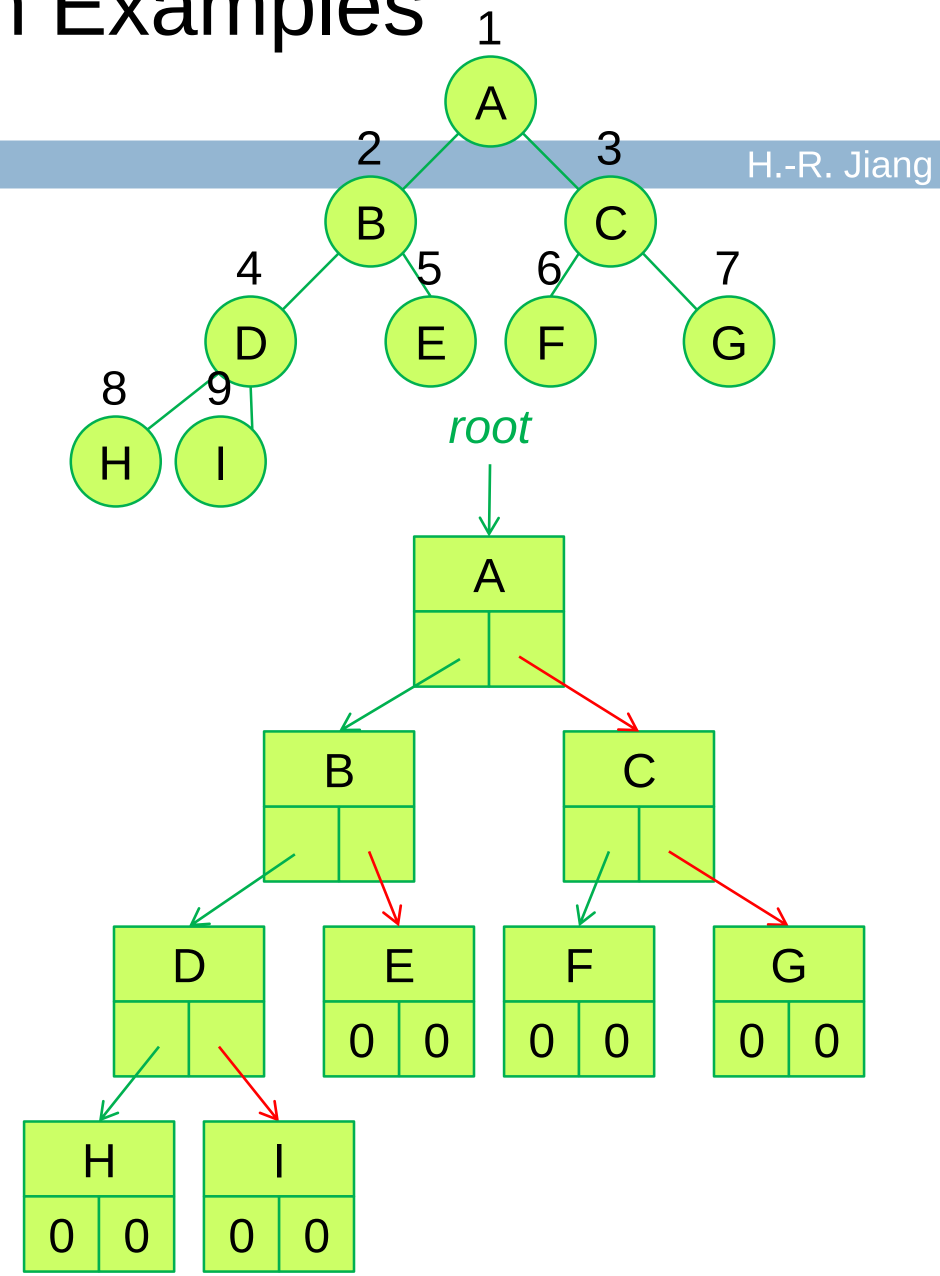
Binary Tree Representations
-- Linked Representation


```
template <class T> class Tree; // forward declaration
class TreeNode {
friend class Tree <T>;
private:
    T data;
    TreeNode <T> *leftChild;
    TreeNode <T> *rightChild;
};

template <class T>
class Tree {
public:
    // Tree operations
    .
private:
    TreeNode <T> *root;
};
```



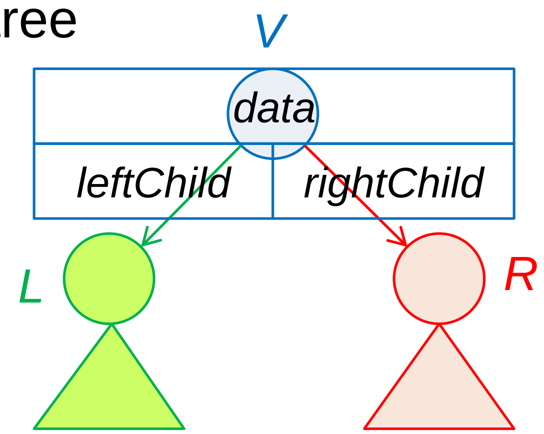
Linked Representation Examples
-- Skewed & Complete Trees





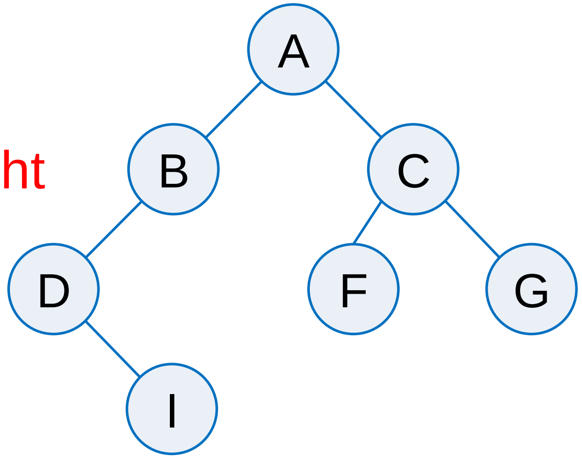
Binary Tree Traversal

- **Visit each node in the tree exactly once**
  - ⇒ Produce a linear order of nodes in a tree
  - 3 actions at a node:
    - *L*: moving left
    - *V*: visiting the node
    - *R*: moving right
  - Have 6 combinations of traversal:
    - *LVR*, *LRV*, *VLR*, *VRL*, *RVL*, *RLV*
  - Adopt the convention: traverse left before right ⇒ 3 remain
    - *V* vs. *L*/*R*
    - **Inorder**: *LVR*
    - **Postorder**: *LRV*
    - **Preorder**: *VLR*
  - Implement in a recursive manner



Binary Tree Traversal
-- Example

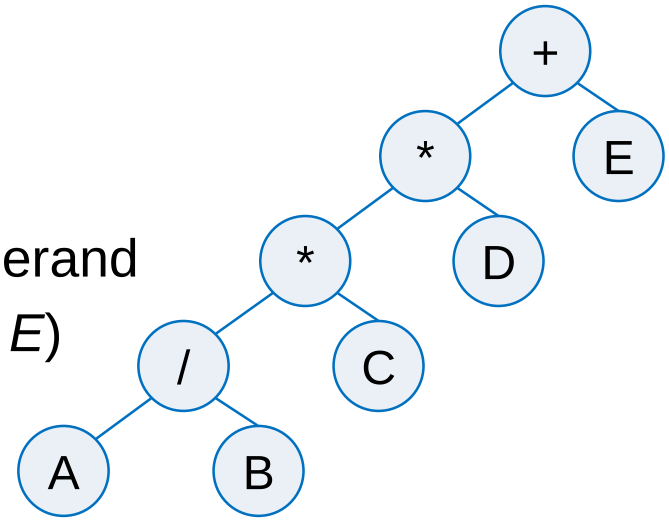
- **A tree traversal is also called a tree walk**
  - Traverse (walk through) a tree
  - e.g., inorder traversal (tree walk)
    1) Move down the tree toward the left until you can go no farther
    2) Then, visit the node
    3) Move one node to the right
    4) And continue



Inorder	Postorder	Preorder
Infix (*LVR*)	**Post**fix (*LRV*)	**Pre**fix (*VLR*)
D I B A F C G	*I D B F G C A*	*A B D I C F G*

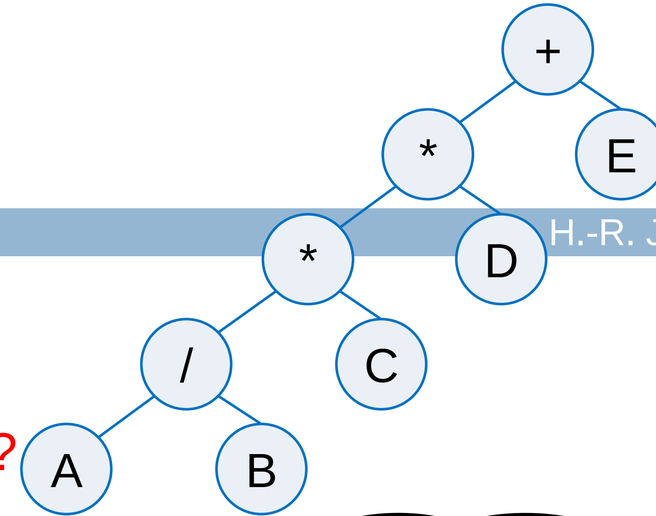
Arithmetic Expression using Binary Tree

- **Ref. Section 3.6**
  - Leaf node: operand
  - Nonleaf nodes: operators
  - Left/right child: left/right operand
- **Example:** ((((*A*/*B*) * *C*) * *D*) + *E*)



Inorder	Postorder	Preorder
Infix (*LVR*)	**Post**fix (*LRV*)	**Pre**fix (*VLR*)
*A / B * C * D + E*	*A B / C * D * E +*	*+ * * / A B C D E*

Inorder Traversal



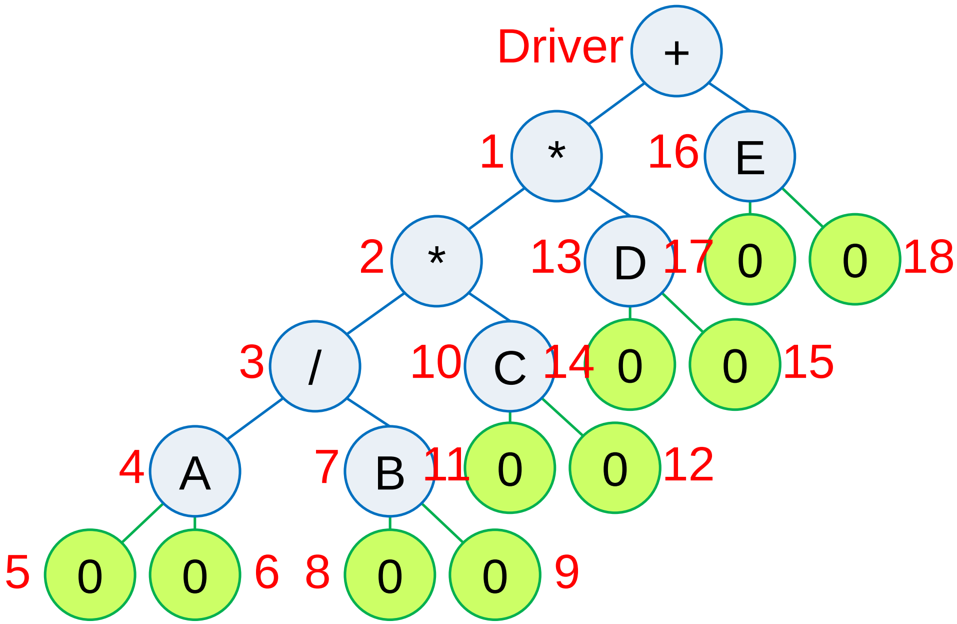
- **Recursion!**
  - e.g., ((((*A*/*B*) * *C*) * *D*) + *E*)
  - Induction basis: terminating condition?
  - Induction hypothesis: recursion?
  - *A / B * C * D + E*

What if postorder and preorder?

```
template <class T>
void Tree <T>::Inorder() { // Driver
// The driver is declared as a public member function of Tree
    Inorder(root);
};

template <class T>
void Tree <T>::Inorder(TreeNode<T> *currentNode) { // Workhorse
// The workhorse is declared as a private member function of Tree
    if (currentNode) { // LVR
        Inorder(currentNode->leftChild);      // L
        Visit(currentNode);                   // V: cout << currentNode->data;
        Inorder(currentNode->rightChild);     // R
    }
};
```

Inorder Traversal
-- Trace Example

H.-R. Jiang



Call of *Inorder*	**Value in *CurrentNode***	**Action**
Driver	+	
1	*	
2	*	
3	/	
4	*A*	
5	0	
4	*A*	**cout** << ‘A’
6	0	
3	/	**cout** << ‘/’
7	*B*	
8	0	
7	*B*	**cout** << ‘B’
9	0	
2	*	**cout** << ‘*’
10	*C*	
11	0	
10	*C*	**cout** << ‘C’
12	0	
1	*	**cout** << ‘*’
13	*D*	
14	0	
13	*D*	**cout** << ‘D’
15	0	
Driver	+	**cout** << ‘+’
16	*E*	
17	0	
16	*E*	**cout** << ‘E’
18	0	

Iterative Inorder Traversal

How to make space complexity O(1)?
- no stack!

- **Time complexity : O(*n*)**
  - Place every node on the stack once ⇒ lines 7, 8, 10—14: O(*n*)
  - *currentNode* == 0 for 0 link ⇒ *n*+1
- **Space complexity : O(*n*)**
  - Max stack size = depth of tree

1. Move down the tree toward the left until you can go no farther
2. Then, visit the node
3. Move one node to the right
4. And continue

```
1.  template <class T>
2.  void Tree <T>::NonrecInorder() {          // Nonrecursive inorder traversal using a stack
3.      Stack<TreeNode <T>*> s;               // declare and initialize stack
4.      TreeNode <T> *currentNode = root;
5.      while (1) {
6.          while (currentNode) {             // move down leftChild fields
7.              s.Push(currentNode);          // add to stack
8.              currentNode = currentNode->leftChild;
9.          }
10.         if (s.IsEmpty()) return;          // stack is empty
11.         currentNode = s.Top();
12.         s.Pop();                          // delete from stack
13.         Visit(currentNode);               // cout << currentNode->data;
14.         currentNode = currentNode->rightChild;
15.     }
16. }
```

Simple Inorder Iterator

Inorder iterator class definition

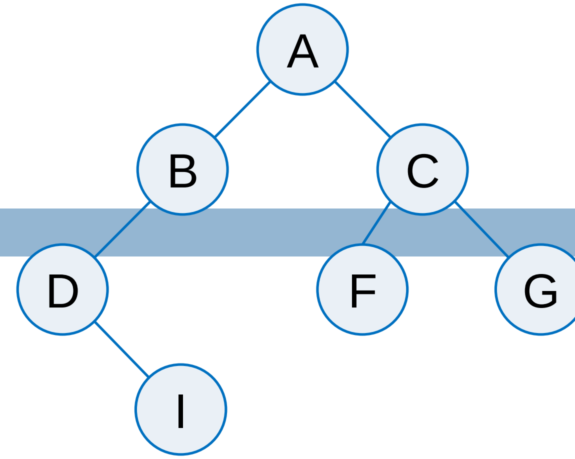
```
// Assumed to be a friend of Tree
class InorderIterator {
public:
    InorderIterator( ) {currentNode = root;};
    T* Next();
private:
    Stack <TreeNode <T>*> s;
    TreeNode <T> *currentNode;
};
```

Obtain the next inorder element

```
T* InorderIterator::Next() { // get next inorder element
    while (currentNode) {
        s.Push(currentNode);
        currentNode = currentNode->leftChild;
    }
    if (s.IsEmpty()) return 0;
    currentNode = s.Top();
    s.Pop();
    T& temp = currentNode->data;
    currentNode = currentNode->rightChild;
    return &temp;
}
```

It's the while loop of *NonrecInorder()*!
- Each iteration of the while loop yields the next element in the traversal
- Return the next element instead of visiting it

Level-Order Traversal



- **Traverse a tree by level**
- **Use a queue instead of a stack**

Inorder	Postorder	Preorder	Level-order
Infix (*LVR*)	**Post**fix (*LRV*)	**Pre**fix (*VLR*)	Level-by-level
D I B A F C G	*I D B F G C A*	*A B D I C F G*	*A B C D F G I*

1. Begin with the root
2. Add its children to the queue, left child first
3. Get the next node from the queue
4. Continue

```
void Tree <T>::LevelOrder() {
   Queue <TreeNode <T>*> q;
   TreeNode <T> *currentNode = root;
   while (currentNode) {
      Visit(currentNode);  // cout << currentNode->data;
      if (currentNode->leftChild) q.Push(currentNode->leftChild);
      if (currentNode->rightChild) q.Push(currentNode->rightChild);
      if (q.IsEmpty()) return;
      currentNode = q.Front();
      q.Pop(); // delete from the head
   }
}
```

Additional Binary Tree Operations

Copy

Equivalence

Satisfiability

Copying Binary Trees

- **Modify any traversal algorithm**
- **Example: in postorder**

```
// copy ctor
template <class T>
Tree <T>::Tree(const Tree<T>& s) { // Driver
    root = Copy(s.root);
};

template <class T>
TreeNode <T>* Tree <T>::Copy(TreeNode<T> *origNode) { // Workhorse
// return a pointer to an exact copy of the binary tree rooted at origNode
    if (!origNode) return 0; // empty tree
    return new TreeNode <T> (origNode->data,
                             Copy(origNode->leftChild),
                             Copy(origNode->rightChild));
};
```

Testing Equality

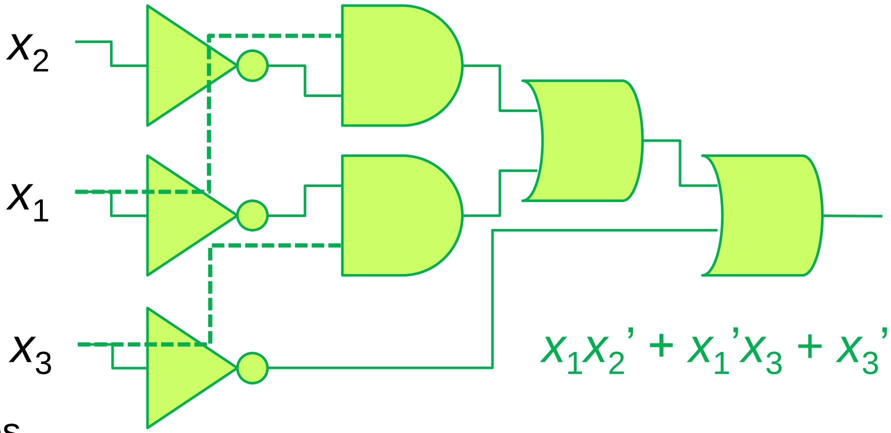
- **Check if both trees have the same linear order by any traversal algorithm**
- **Example: in preorder**

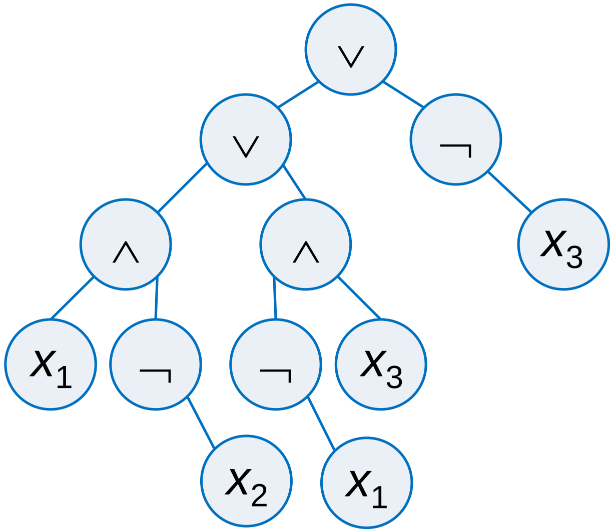
```
// assume to be a friend of Tree
// operator overloading
bool operator== (const Tree& s, const Tree& t) { // Driver
    return Equal(s.root, t.root);
}

// assume to be a friend of TreeNode
bool Tree <T>::Equal(TreeNode <T> *a, TreeNode <T> *b) { // Workhorse
    if ((!a) && (!b)) return true;                 // both a and b are 0
    return (a && b                                  // both a and b are non-zero
            && (a->data == b->data)                 // same data
            && Equal(a->leftChild, b->leftChild)    // same left
            && Equal(a->rightChild, b->rightChild)); // same right
}
```

The Satisfiability Problem

- **Example:** $x_1 \vee (x_2 \wedge \neg x_3) \Rightarrow x_1$ or x_2 and not x_3
- **Propositional calculus:** priority: $\neg > \wedge > \vee$
  - x_1 and x_3 are false; x_2 is true $\Rightarrow F \vee (T \wedge \neg F) = T$
- **The satisfiability (SAT) problem:** (Answer Yes/No)

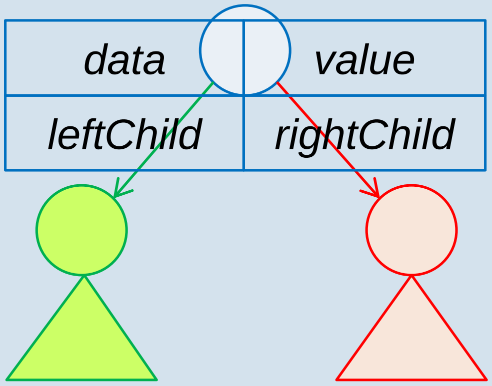
  Is there an assignment making the expression true?
  - Exhaustive test in $O(g\ 2^n)$ time (g: evaluating an assignment)
  - How to evaluate an assignment? $\Rightarrow$ Postorder traversal (Why?)
    - Evaluate a node when its left/right subexpressions are ready
  - e.g., $(x_1 \wedge \neg x_2) \vee (\neg x_1 \wedge x_3) \vee \neg x_3$





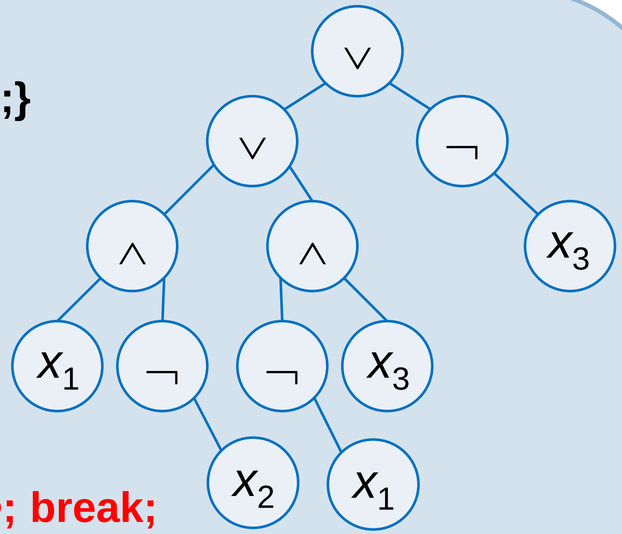
SAT (1/2)


```
enum Operator {Not, And, Or, True, False};

class SatTree; // forward declaration
class SatNode {
    friend class SatTree;
    Operator data;
    bool value;
    SatNode *leftChild;
    SatNode *rightChild;
}

class SatTree {
public:
    void PostOrderEval();
    void rootValue() {cout << root->value;}
private:
    SatNode *root;
    void PostOrderEval(SatNode *);
};
```



SAT (2/2)


```
// Driver
void SatTree::PostOrderEval() {PostOrderEval(root);}

// Workhorse
void SatTree::PostOrderEval(SatNode *s) {
    if (s) { // not null
        PostOrderEval(s->leftChild);
        PostOrderEval(s->rightChild);
        switch(s->data) {
            case Not:   s->value = !s->rightChild->value; break;
            case And:   s->value = s->leftChild->value && s->rightChild->value; break;
            case Or:    s->value = s->leftChild->value || s->rightChild->value; break;
            case True:  s->value = true; break; // terminal node
            case False: s->value = false; // terminal node
        }
    }
}
```



Threaded Binary Trees

Threaded Binary Trees

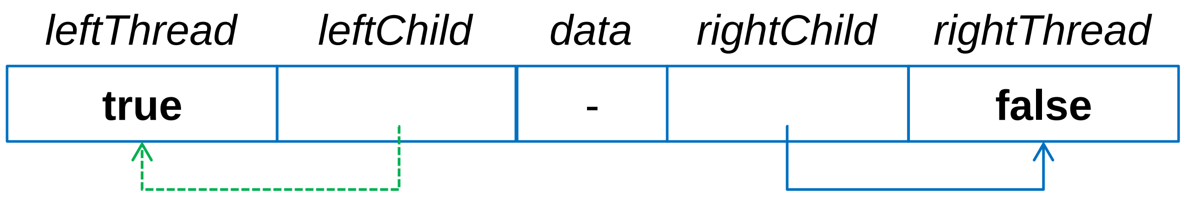
- **Do not waste 0-links in linked representation!**
  - Consider a binary tree with n nodes ($n > 0$)
    - $2n$ links in total
    - Only n-1 links in use
- **Replace 0-links by pointers, called threads**
  - 0 *rightChild* of $p \leftarrow p$'s inorder successor (who right after p)
  - 0 *leftChild* of $p \leftarrow p$'s inorder predecessor (who right before p)
  - $\Rightarrow$ Facilitate tree operations
- **How to distinguish between threads and normal pointers?**
  - Add two extra **bool** fields: *leftThread*, *rightThread*

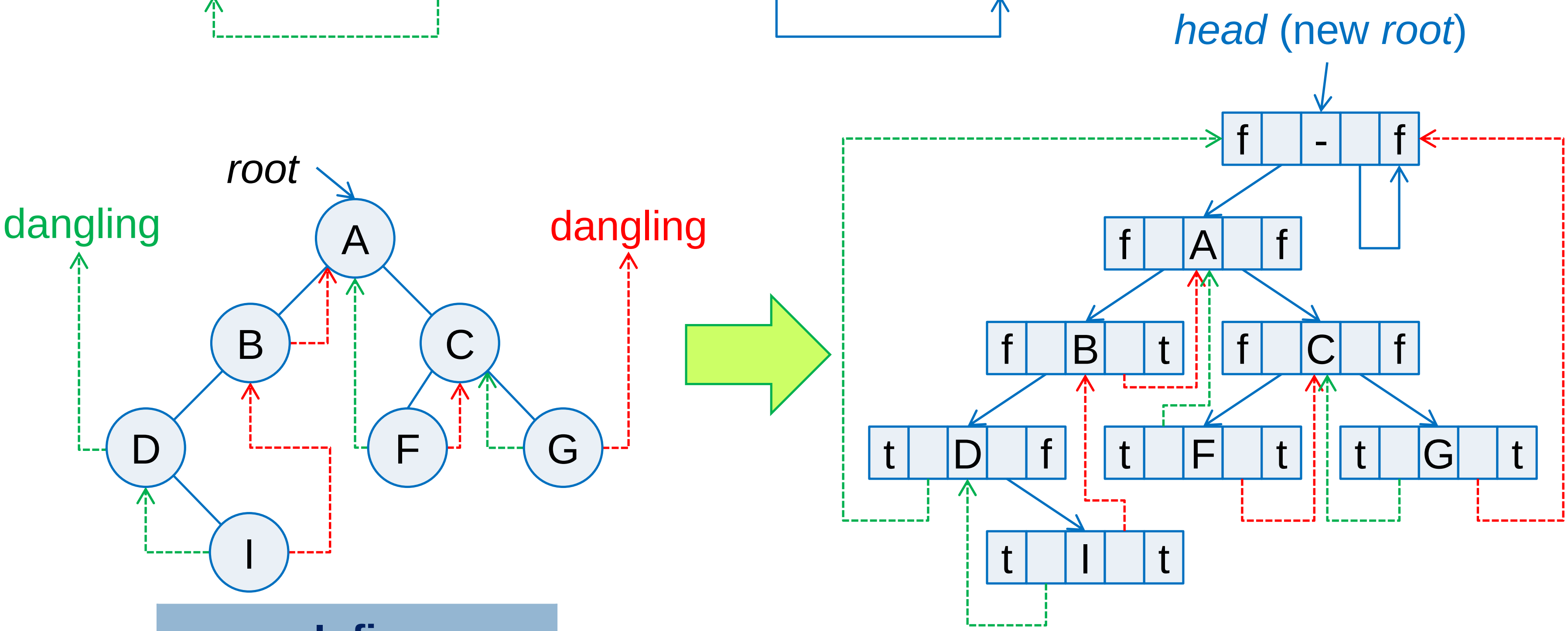
leftThread	*leftChild*	*data*	*rightChild*	*rightThread*

Threaded Binary Trees

-- Example

- **Add a header node to avoid dangling threads**

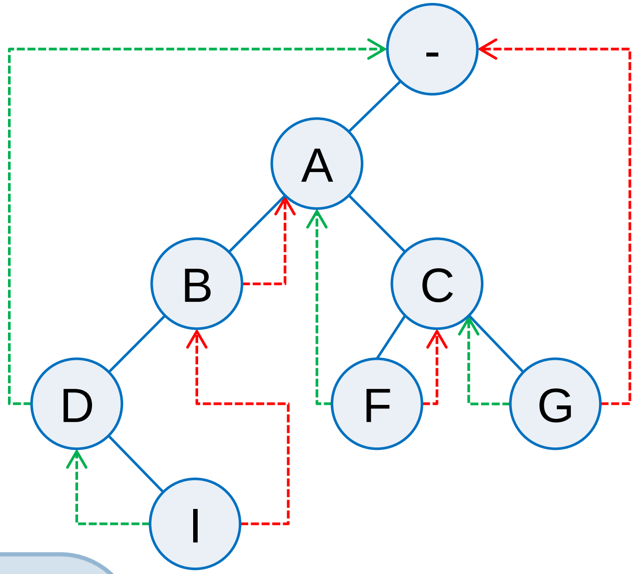




Infix
D I B A F C G

Finding the Inorder Successor

- **Where is the inorder successor of node *p*?**
  - What if *p* has a right thread?
    - Done!
  - What if *p* has no right thread?
    - Check whose predecessor is *p*
    - The leftmost node in *p*'s right subtree



Infix
D I B A F C G

```
T* ThreadInorderIterator::Next() { // return inorder successor
    ThreadNode <T> *temp = currentNode->rightChild;
    if (!currentNode->rightThread)
        while (!temp->leftThread) temp=temp->leftChild;
    currentNode = temp;
    if (currentNode == root) return 0;
    else return &currentNode->data;
}
```

Space complexity: O(1)
- No stack!

How to insert/delete?

CHAPTER 5
TREES PART II

Iris Hui-Ru Jiang **Fall 2008**

Trees Part II

- **Contents**
  - Trees
  - Binary trees
  - Threaded binary trees
  - Heaps
  - Binary search trees
  - Selection trees
  - Forests
  - Disjoint sets
- **Readings**
  - Chapter 5
  - Section 3.4, 7.6

Heaps: Priority Queues

Binary Tree Application

Priority Queue

- **In a priority queue (PQ)**
  - Each element has a priority (key value)
  - Only the element with highest (or lowest) priority can be deleted
    - Max priority queue, or min priority queue
  - An element with arbitrary priority can be inserted into the queue at any time

ADT *MaxPQ*

- **Make an implementation as publicly derived class (ref. Sec. 3.4)**

```
template <class T>
class MaxPQ {
public:
    virtual ~MaxPQ() {} // virtual dtor
    virtual bool IsEmpty() const = 0; // return true iff empty
    virtual const T& Top() const = 0; // return reference to max element
    virtual void Push(const T&) = 0; // add an element
    virtual void Pop() = 0; // delete element with max priority
};
```

What if *MinPQ*?

- **Time complexity comparison on various representations**

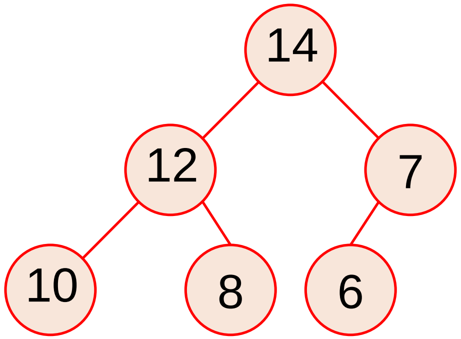
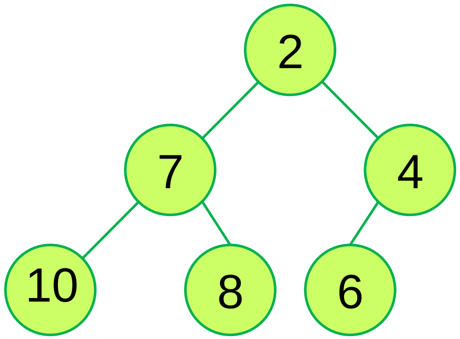
Representation	Insertion	Deletion
Unordered array	$\Theta(1)$	$\Theta(n)$
Sorted array	$\Theta(n)$	$\Theta(1)$
Unordered linked list	$\Theta(1)$	$\Theta(n)$
Sorted linked list	$\Theta(n)$	$\Theta(1)$
Heap	$\Theta(\lg n)$	$\Theta(\lg n)$

Assume blanks are allowed

lg: $\log_2$

Heap

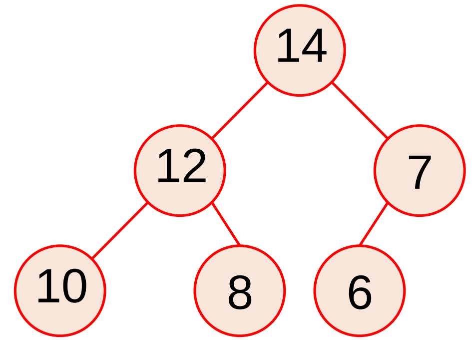
- **Definition: A max (min) heap is**
  - A max (min) tree: *key*[*parent*] >= (<=) *key*[*children*]
  - A complete binary tree
- **Corollary:** Who has the largest (smallest) key in a max (min) heap?
  - Root!
- **Example**

Max heap	Min heap
	

Class *MaxHeap*

- **Implementation?**
  - Complete binary tree ⇒ array representation

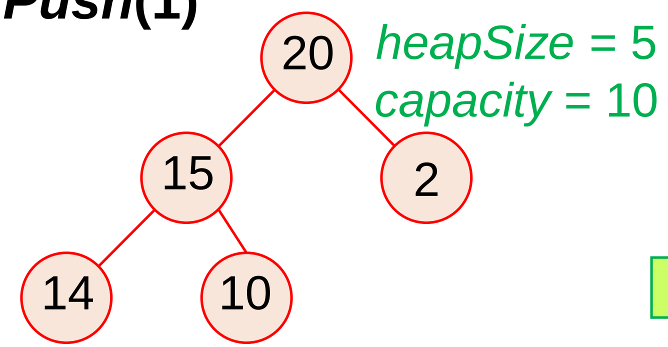
```
template <class T>
class MaxHeap: public MaxPQ<T> {
public:
    MaxHeap (int theCapacity = 10);
    …
private:
    T *heap;          // element array
    int heapSize;     // # of elements in heap
    int capacity;     // size of the array heap
};

template <class T>
MaxHeap<T>::MaxHeap (int theCapacity=10) { // ctor
    if (theCapacity < 1) throw “Capacity must be >= 1”;
    capacity = theCapacity;
    heapSize = 0;
    heap = new T [capacity+1]; // heap[0] unused
}
```



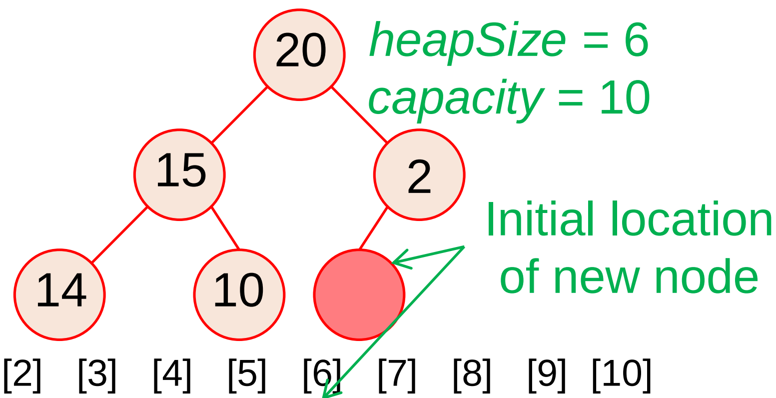
	[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]
heap	-	14	12	7	10	8	6	-	-	-	-

Insertion into a Max Heap (1/3)

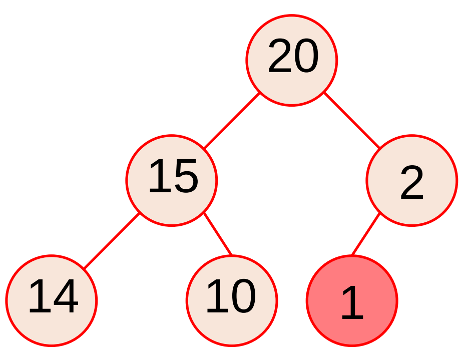
- **Maintain heap property all the times**
- ***Push*(1)**



[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]
-	20	15	2	14	10	-	-	-	-	-



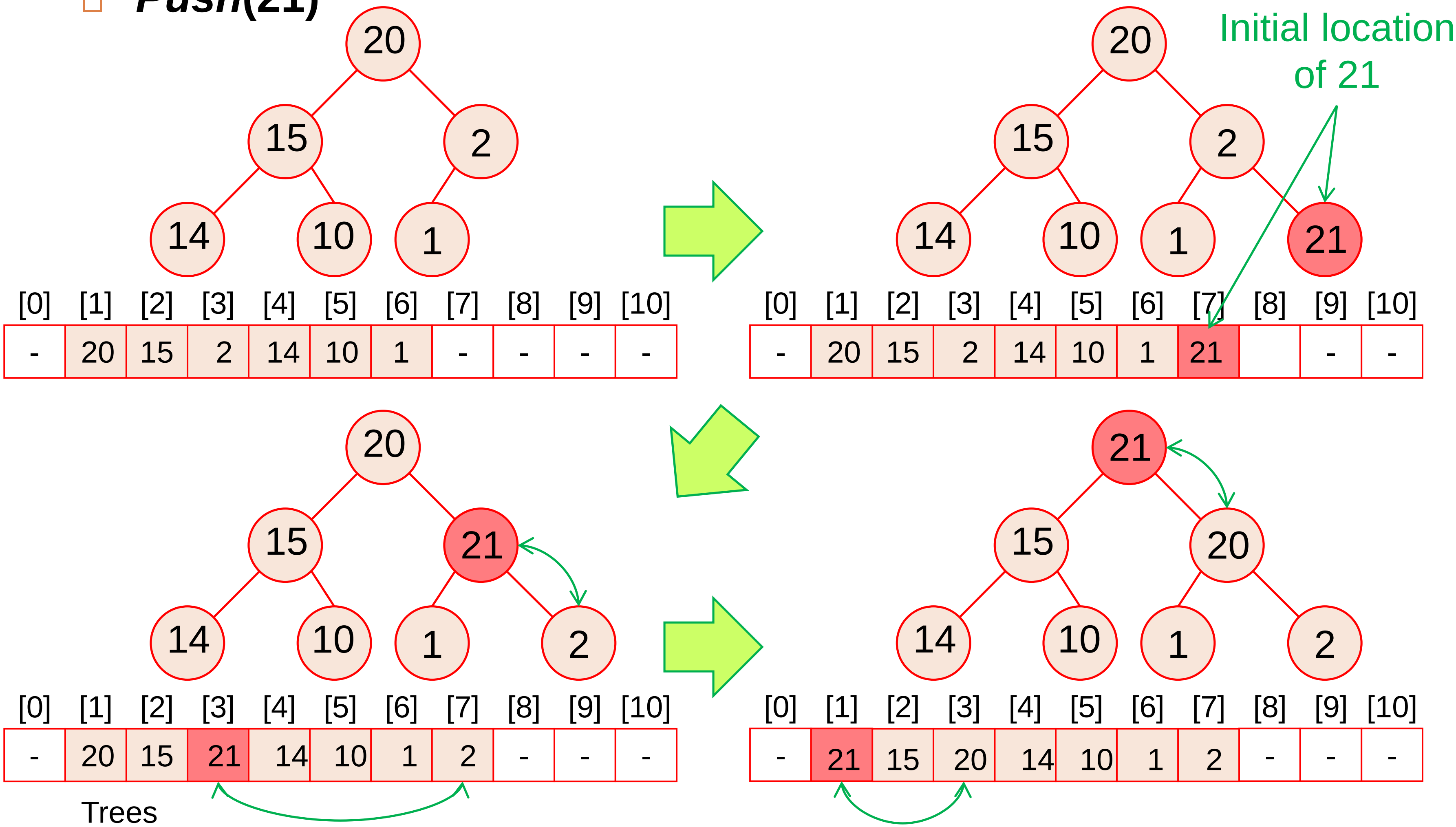
[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]
-	20	15	2	14	10		-	-	-	-



[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]
-	20	15	2	14	10	1	-	-	-	-

Insertion into a Max Heap (2/3)

- **Maintain heap ⇒ bubble up if needed!**
- ***Push*(21)**



Insertion into a Max Heap (3/3)

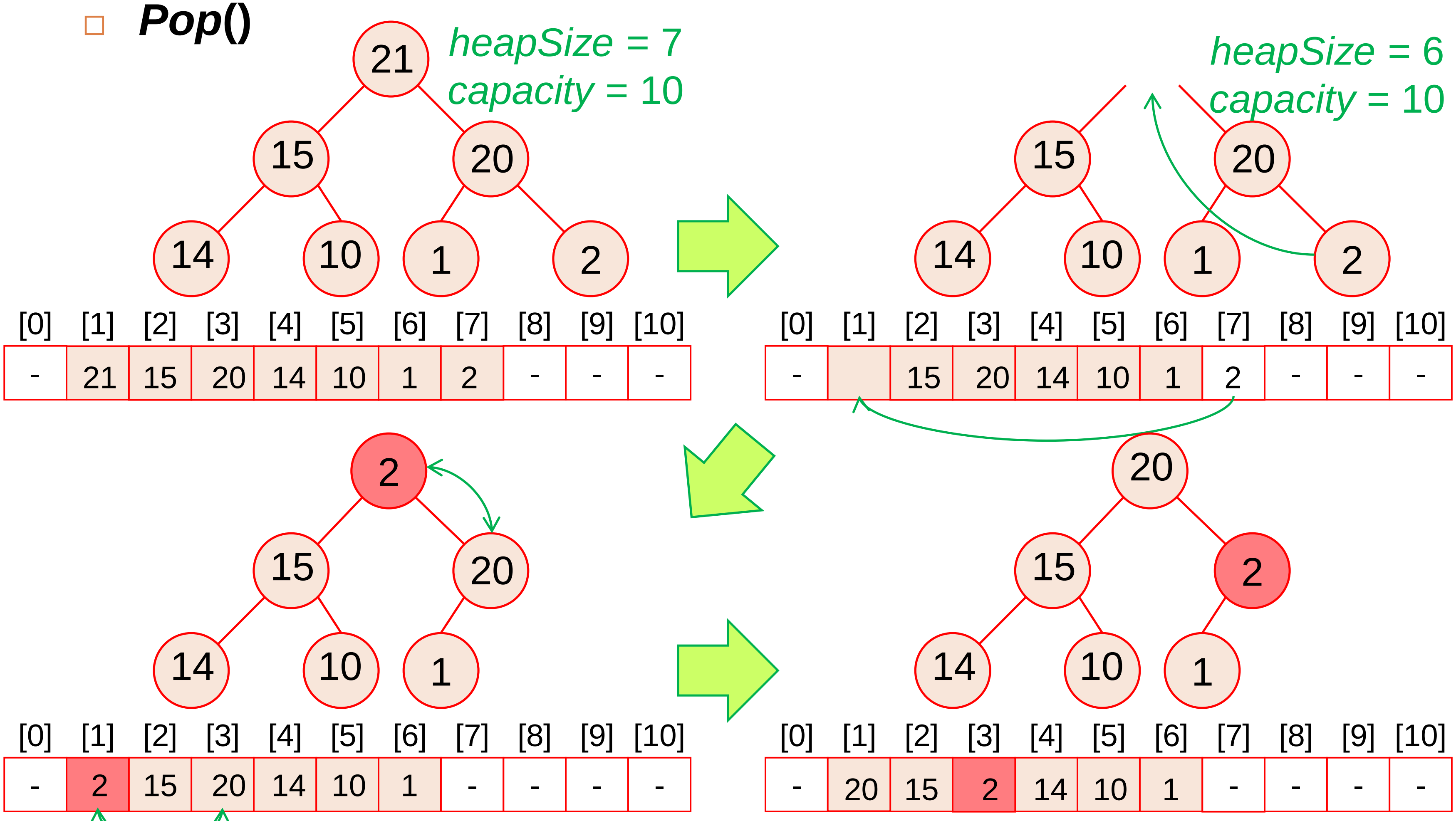
- **Time complexity?**
  - How many times to bubble up in the worst case?
  - Tree height: $\Theta(\lg n)$

```
template <class T>
void MaxHeap<T>::Push(const T& e) {
    if (heapSize == capacity)  { // double capacity
        ChangeSize1D(heap, capacity, 2*capacity);
        capacity *=2;
    }
    int currentNode = ++heapSize;  // initial location of new node

    // bubble up
    while (currentNode != 1 && heap[currentNode/2] < e) {
        heap[currentNode] = heap[currentNode/2];  // move smaller parent down
        currentNode /=2;
    }
    heap[currentNode] = e;
}
```

Deletion from a Max Heap (1/3)

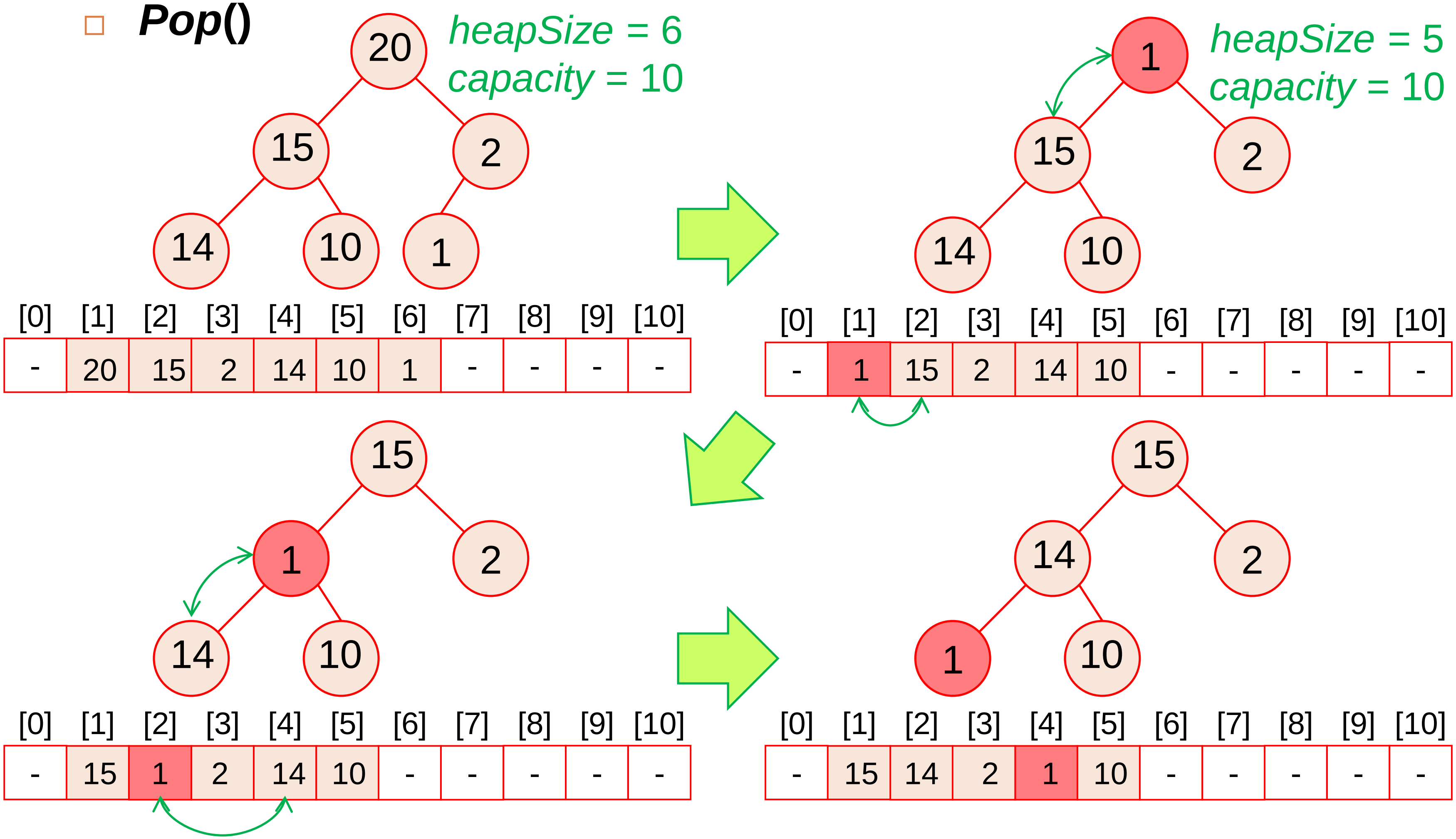
H.-R. Jiang

- **Maintain heap ⇒ trickle down if needed!**
- ***Pop*()**



Deletion from a Max Heap (2/3)

- **Maintain heap ⇒ trickle down if needed!**
- ***Pop()***



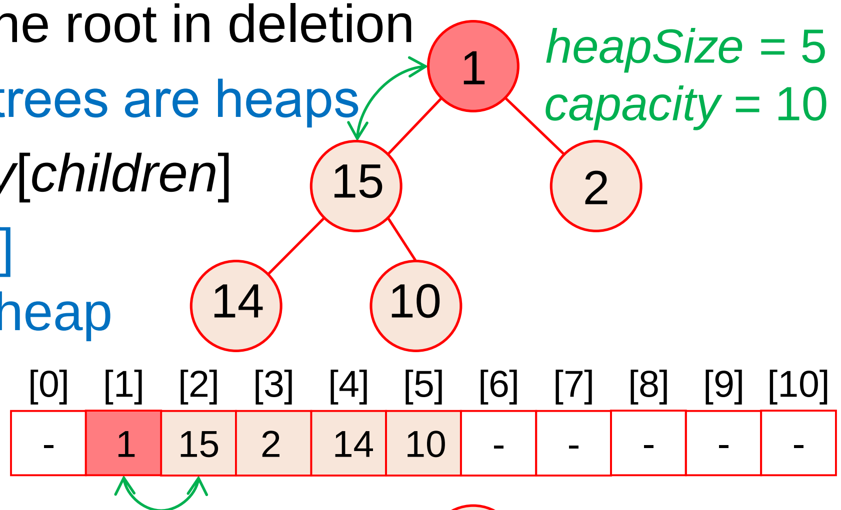
Deletion from a Max Heap (3/3)

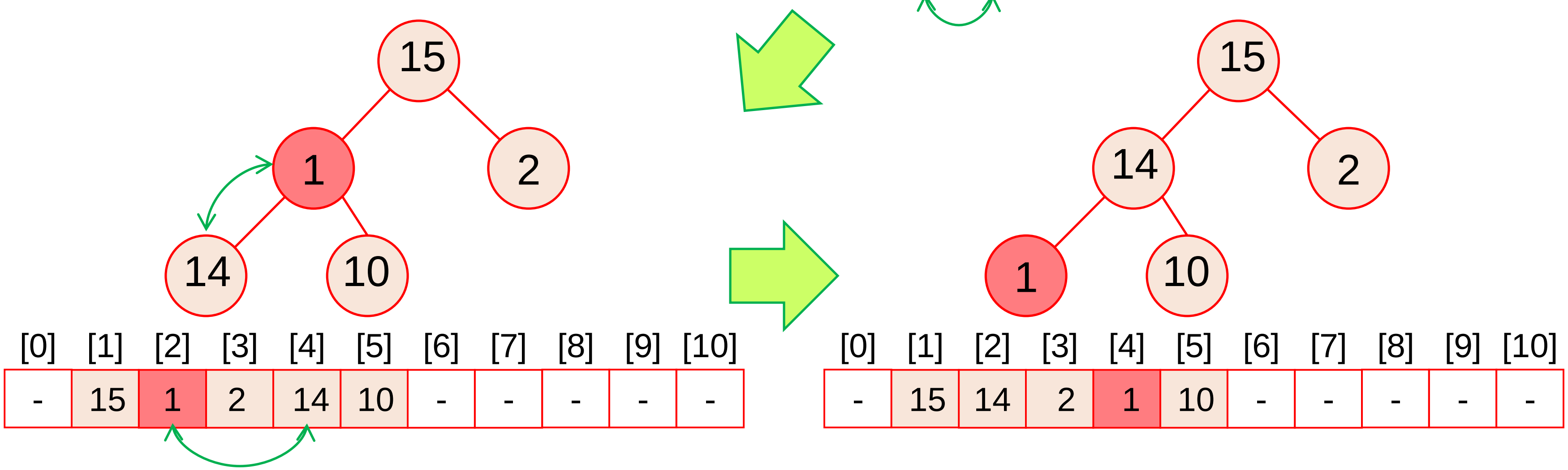
- **Time complexity?**
  - How many times to trickle down in the worst case? $\Theta(\lg n)$

```
template <class T>
void MaxHeap<T>::Pop() {
    if (IsEmpty()) throw "Heap is empty! Cannot delete.";
    heap[1].~T(); // delete max element
    T lastE = heap[heapSize--]; // remove last element from heap

    // trickle down
    int currentNode = 1;  // from root
    int child = 2;  // a child of currentNode
    while (child <= heapSize) {
        if (child < heapSize && heap[child] < heap[child+1])
            child++; // set child to larger child of currentNode
        if (lastE >= heap[child]) break; // we can put lastE in currentNode
        // cannot
        heap[currentNode] = heap[child];  // move child up
        currentNode = child; child *=2; // move down a level
    }
    heap[currentNode] = lastE;
}
```

Max Heapify

- **Max (min) heapify = maintain the max (min) heap property**
  - What we do to trickle down the root in deletion
  - Assume *i*'s left and right subtrees are heaps
    - But *key*[*i*] may be < (>) *key*[*children*]
  - Heapify *i* = trickle down *key*[*i*]
    ⇒ the tree rooted at *i* is a heap

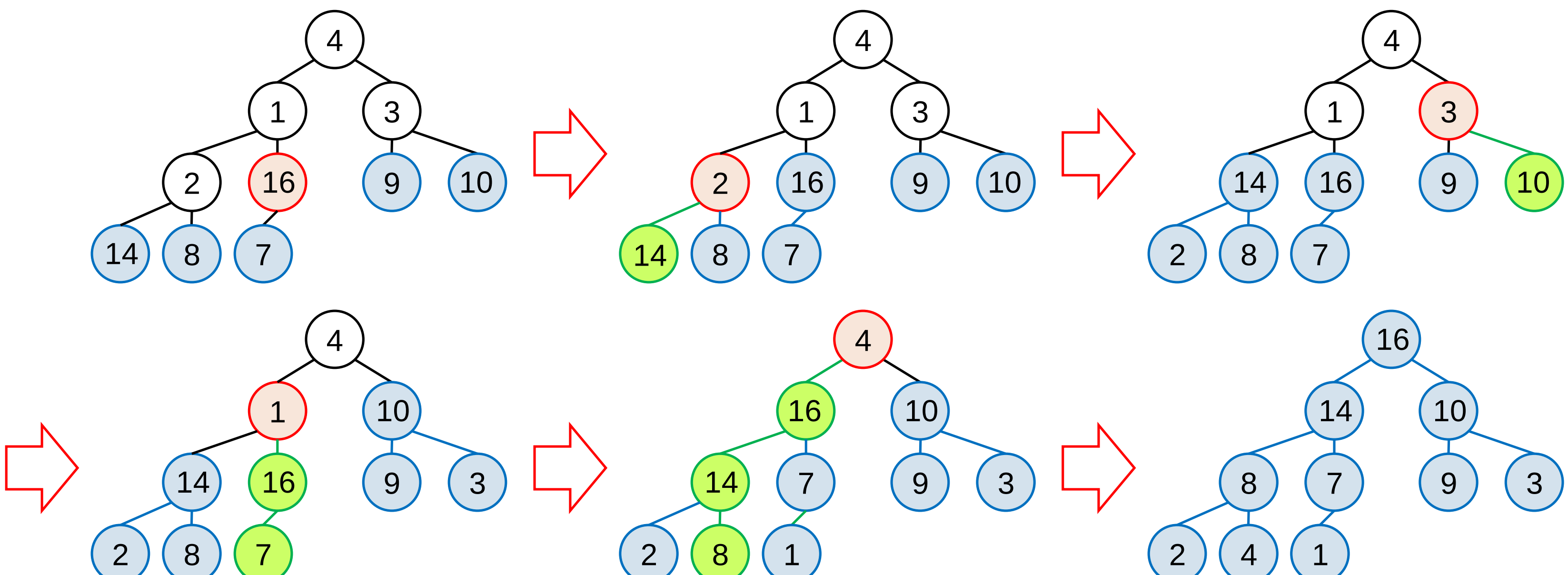




How to Build a Max Heap?

- **How to convert any complete binary tree to a max heap?**
- **Intuition: Max heapify in a bottom-up manner**
  - Induction basis: Leaves are already heaps
  - Induction steps: Start at parents of leaves, work upward till root
  - Time complexity: $O(n \lg n)$

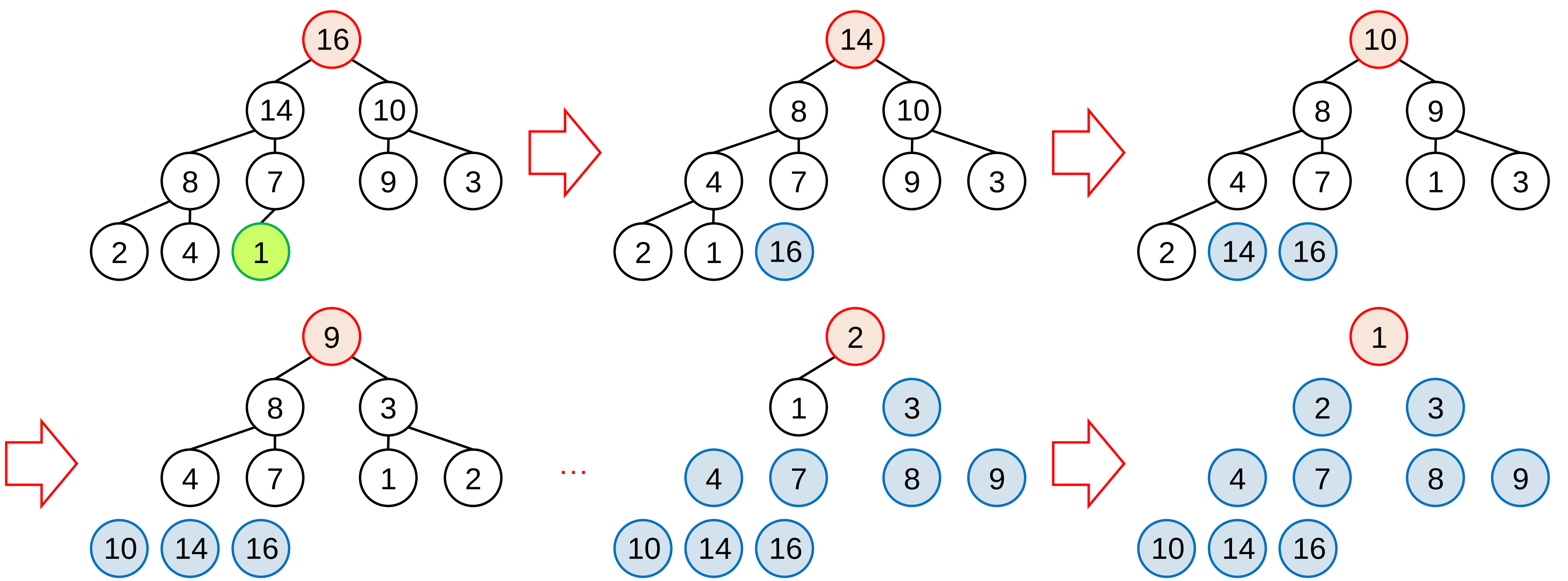
	[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]
a	-	4	1	3	2	16	9	10	14	8	7



Trees

Heap Sort (1/2)

- **How to sort *a*[1:*n*] into nondecreasing order?**
- **Intuition: Build max heap + Sort!**
  - Time complexity: O(*n* lg *n*)
  - Space complexity: O(*n*) for array, in-place



	[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]
a	-	1	2	3	4	7	8	9	10	14	16

Heap Sort (2/2)


```
template <class T>
void Adjust(T *a, const int root, const int n) { // heapify node at root, heapsize n
    T e = a[root];
    for (int j = 2*root; j <= n; j *=2) { // find proper place for e
        if (j < n && a[j] < a[j+1]) j++; // j is max child of its parent
        if (e >= a[j]) break; // e can be parent of j
        a[j/2] = a[j]; // cannot ⇒ move up to j's parent
    }
    a[j/2] = e;
}

template <class T>
void HeapSort(T *a, const int n) { // sort a[1:n] into nondecreasing order
    for (int i = n/2; i >= 1; i--) // build max heap by bottom-up heapifying
        Adjust(a, i, n);

    for (int i = n-1; i >=1; i--) { // sort
        swap(a[1], a[i+1]); // swap first and last element
        Adjust(a, 1, i); // heapify the new root
    }
}
```

while loop in *Pop*()

Summary: Priority Queue using Heap

- **A priority queue is a data structure on sets of keys; a max priority queue supports the following operations:**
  - *Top*(): return the max key in *MaxPQ*
  - *Push*(*e*): insert *e* into *MaxPQ*
  - *Pop*(): delete the max key in *MaxPQ*
  - *Increase*(*e*, *k*): increase the node of key value *e* to new value *k*
- **These operations can be easily supported using a heap**
  - *Top*(): read the 1st element in O(1) time
  - *Push*(*e*): insert the node at the end and fix heap in O(lg *n*) time
  - *Pop*(): delete the 1st element, replace it with the last, decrement the heap size, then heapify in O(lg *n*) time
  - *Increase*(*e*, *k*): traverse a path from the target node upward to *root* to find a proper place for the new key in O(lg *n*) time

Binary Search Trees

Binary Tree Application

Dictionary (Dynamic Set)

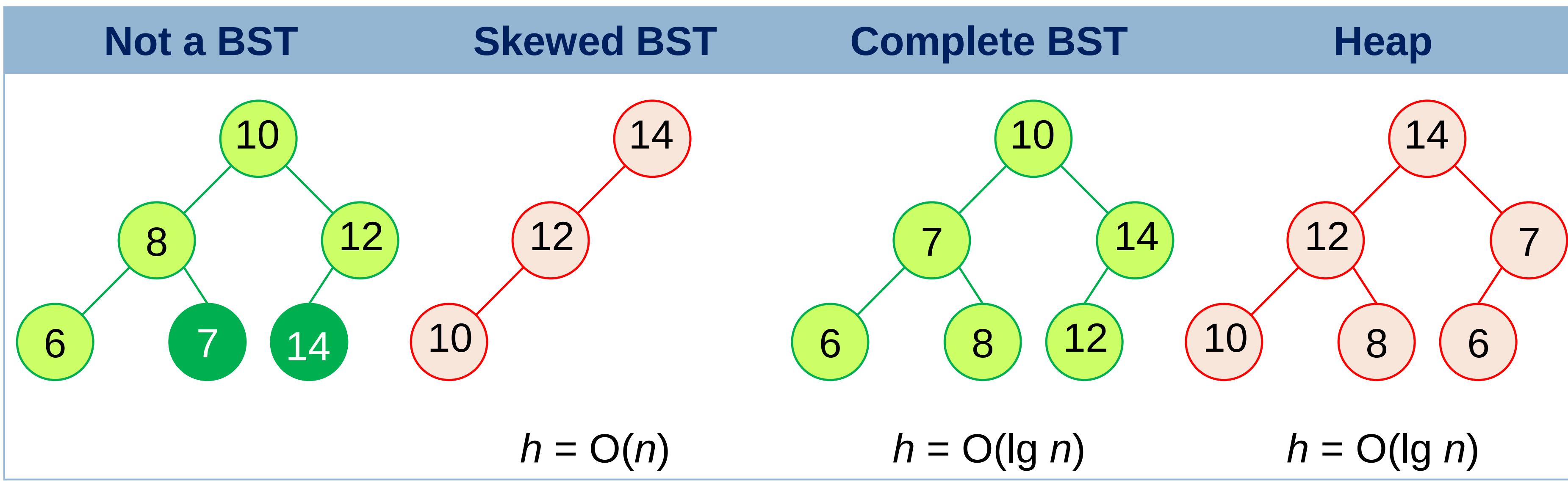
- **Definition: A dictionary is**
  - A collection of pairs: key and an associated element
  - To support operations:
    - Queries: search, min, max, rank, successor, predecessor
    - Modifications: insert, delete
- **Implementation: heap vs. binary search tree**
  - Heap is suitable for a priority queue
  - Binary search tree is good for a dictionary if h is well controlled

Operation	**Min/Max Heap**	**Binary Search Tree**
Search	O(n)	O(h)
Min/Max	O(1)	O(h)
Rank	O(n)	O(h)
Insert	O(h)	O(h)
Delete	O(h): root; O(n): arbitrary	O(h)

Tree size: n; tree height: h

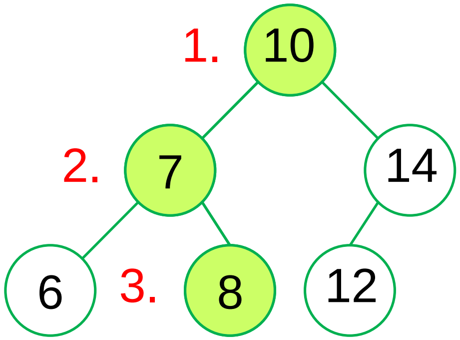
Binary Search Tree

- **Definition: A binary search tree *T* is a binary tree**
  - *T* may be empty, or
  - *T* is not empty,
    - All keys are distinct (this condition can be relaxed)
    - If y in left subtree of x, $key[y] < key[x]$ ($\leq$)
    - If y in right subtree of x, $key[y] > key[x]$ ($\geq$)
    - Left and right subtrees are also binary search trees



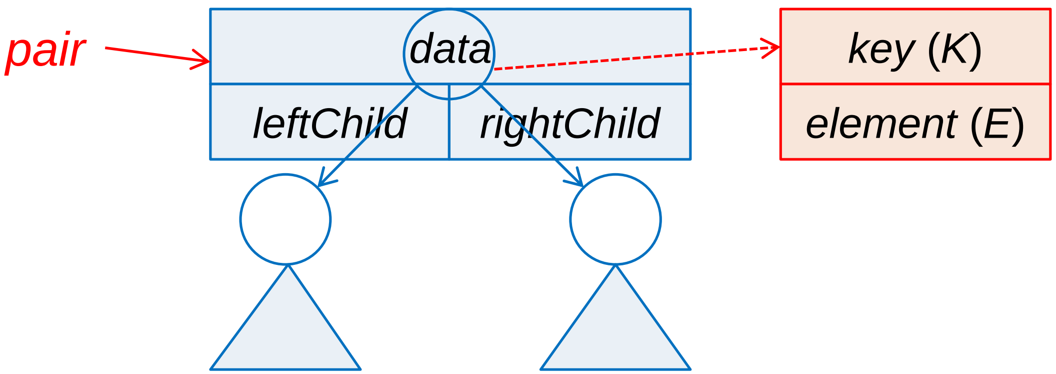
Search

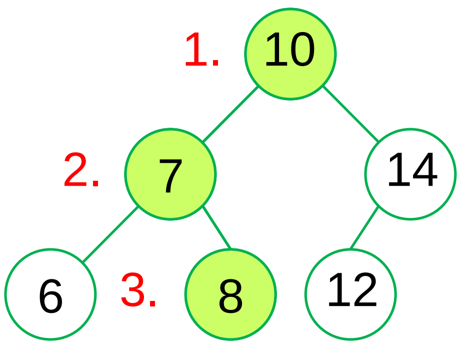
- **How to search for an element with key *k*?**
  - By definition!
  - Begin at the root
  - If the current node *p* == 0, empty and fail!
  - Else, compare *k* with *key*[*p*]
    - *k* == *key*[*p*], found!
    - *k* < *key*[*p*], go left
    - *k* > *key*[*p*], go right
- **Example: search 8**



Recursive Search


```
template <class K, class E> // Driver
pair<K, E>* BST<K, E>::Get(const K& k) { // search *this for key k
    return Get(root, k);
}

template <class K, class E> // Workhorse
pair<K, E>* BST<K, E>::Get(TreeNode <pair<K, E>>* p, const K& k) {
    if (!p) return 0; // not found
    if (k < p->data.key) return Get(p->leftChild); // go left
    if (k > p->data.key) return Get(p->rightChild); // go right
    return &p->data; // found
}
```





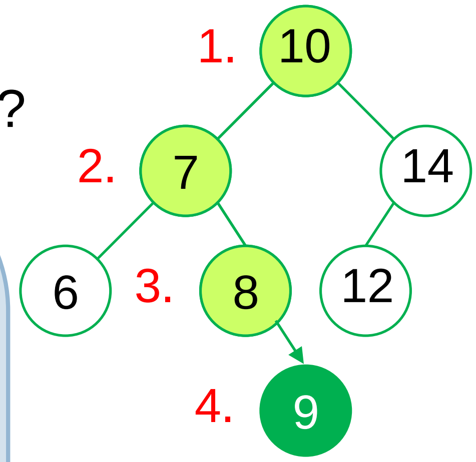
Iterative Search


```
template <class K, class E> // Iterative version
pair<K, E>* BST<K, E>::Get(const K& k) {
    TreeNode <pair<K, E>> *p = root; // current node p begins at root
    while (p)
        if (k < p->data.key) p = p->leftChild;
        else if (k > p->data.key) p = p->rightChild;
        else return &p->data; // found
    return 0; // not found
}
```

Insertion

How to build a BST?
- Worst-case time?
- Best-case time?

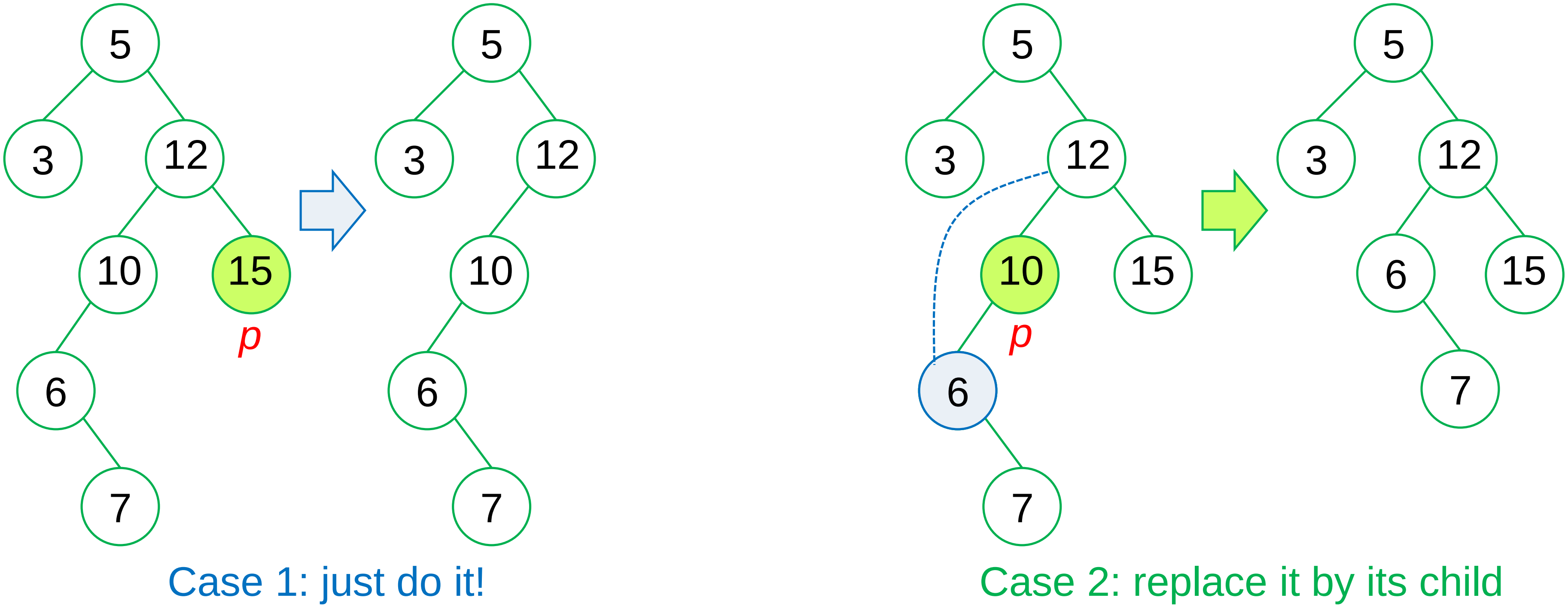
- **How to insert an element with key *k*?**
  - Pretend you are searching key *k*…, e.g., insert 9?




```
template <class K, class E>
void BST<K, E>::Insert(const pair<K, E>& thePair) {
   // 1. find location (search thePair.key), pp is parent of p
   TreeNode <pair<K, E>> *p = root, *pp = 0;
   while (p) {
      pp = p;
      if (thePair.key < p->data.key) p = p->leftChild;
      else if (thePair.key > p->data.key) p = p->rightChild;
      else { // duplicate, consider as an update or something else
         p->data.element = thePair.element; return;}
   }
   // 2. perform insertion
   p = new TreeNode <pair<K, E>> (thePair);
   if (root) // tree nonempty
      if (thePair.key < pp->data.key) pp->leftChild = p;
      else pp->rightChild = p;
   else root = p;
}
```

Deletion (1/2)

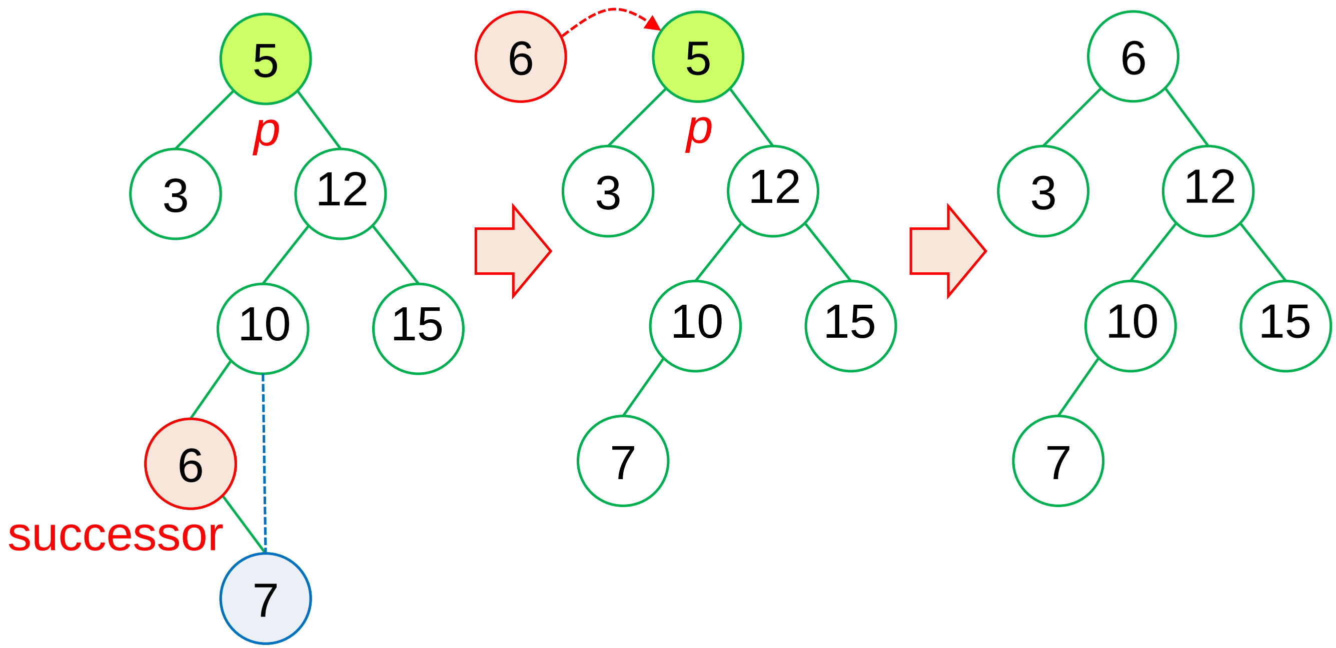
- **How to delete an element *p*?**
  - Case 1: *p* has no children (i.e., a leaf)
  - Case 2: *p* has one child
  - Case 3: *p* has two children



Deletion (2/2)

 H.-R. Jiang

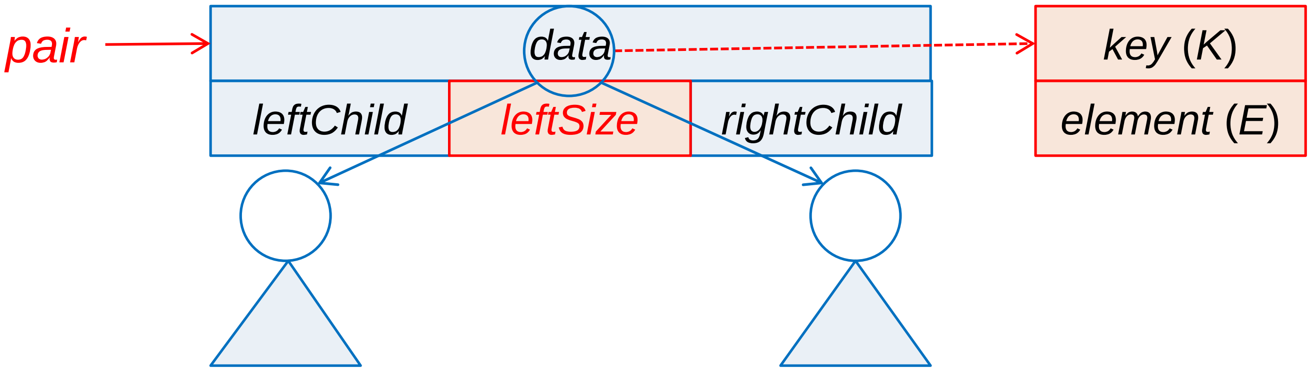
- **How to delete an element *p*?**
  - Case 1: *p* has no children (i.e., a leaf)
  - Case 2: *p* has one child
  - Case 3: *p* has two children



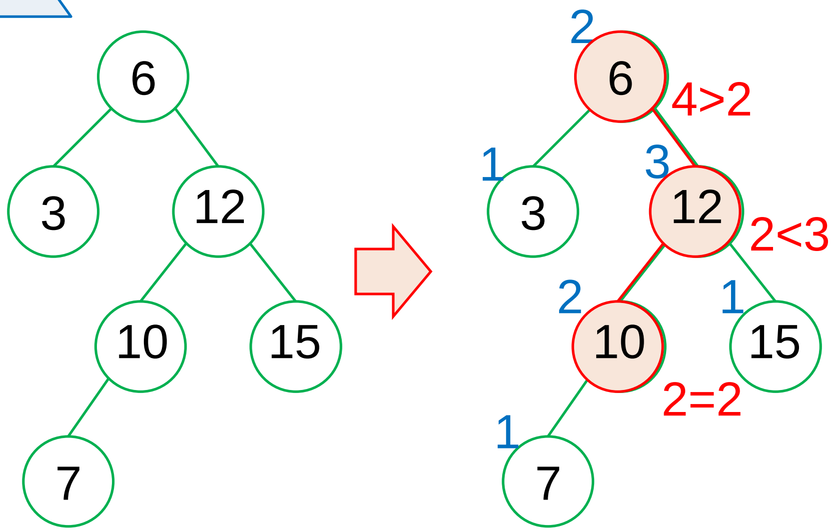
Case 3: replace it by its successor (smallest in right subtree)
or predecessor (largest in left subtree)

Search by Rank (1/2)

- **Augment an extra field into each node**
  - *leftSize*: 1 + # of elements in left subtree
    - i.e., the rank in the tree rooted at itself



- e.g., Who is 4th smallest?
- How to maintain *leftSize*?

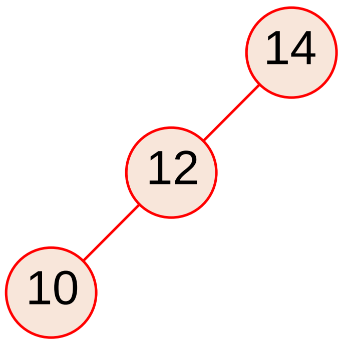
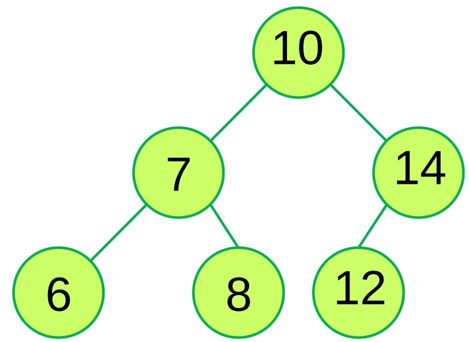


Search by Rank (2/2)


```
template <class K, class E>
pair<K, E>* BST<K, E>::RankGet(int r) { // search rth smallest
    TreeNode <pair<K, E>> *p = root; // current node p begins at root
    while (p)
        if (r < p->leftSize) p = p->leftChild;
        else if (r > p->leftSize) { r -= p->leftSize; p = p->rightChild;}
        else return &p->data; // found
    return 0; // not found
}
```

Tree Height!

- **All operations in a BST can be done in O(*h*) time**

Worst-case: Skewed BST	**Best-case: Complete BST**
	
$h = O(n)$	$h = O(\lg n)$

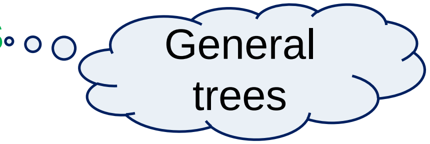
Tree size: *n*; tree height: *h*

- **Wanted: Balanced search trees**
  - e.g., AVL, red/black, 2-3, 2-3-4, B, B+ trees
  - Ref. Chapters 10~11

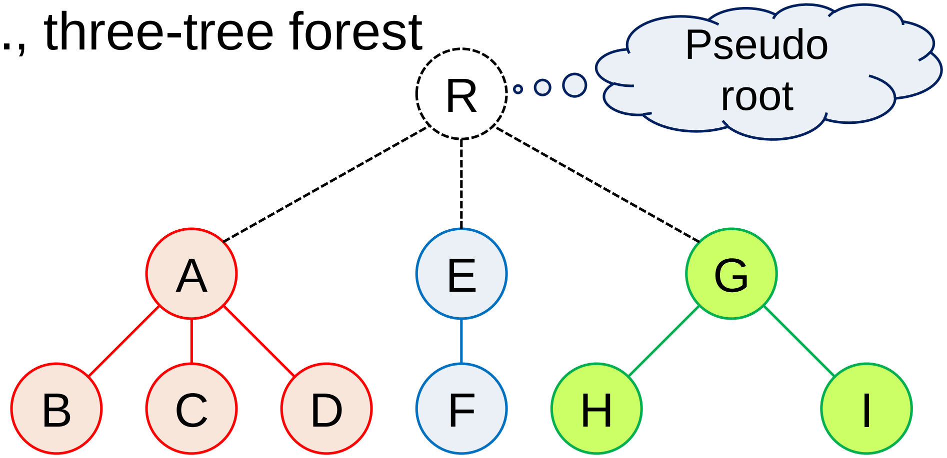
Forests

Forests

- **Definition: A forest is a set of $n \geq 0$ disjoint trees.**

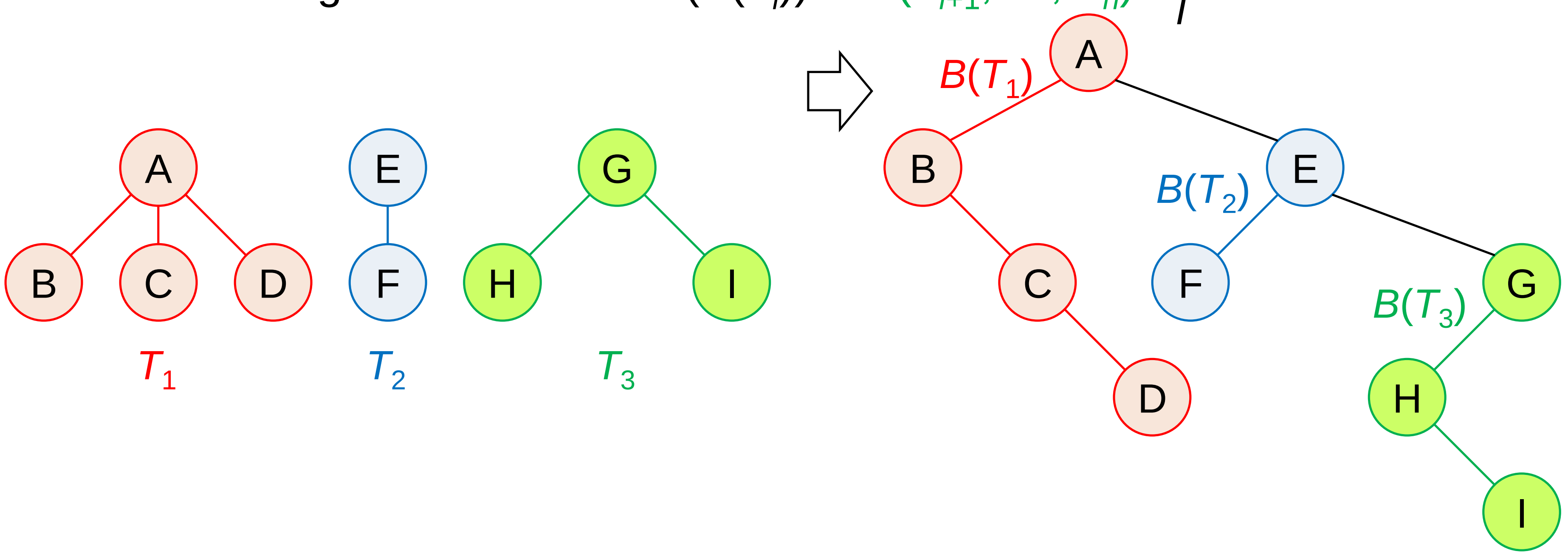


  - Forest vs. tree
    - Remove the root of a tree and obtain a forest, and vice versa
  - e.g., three-tree forest



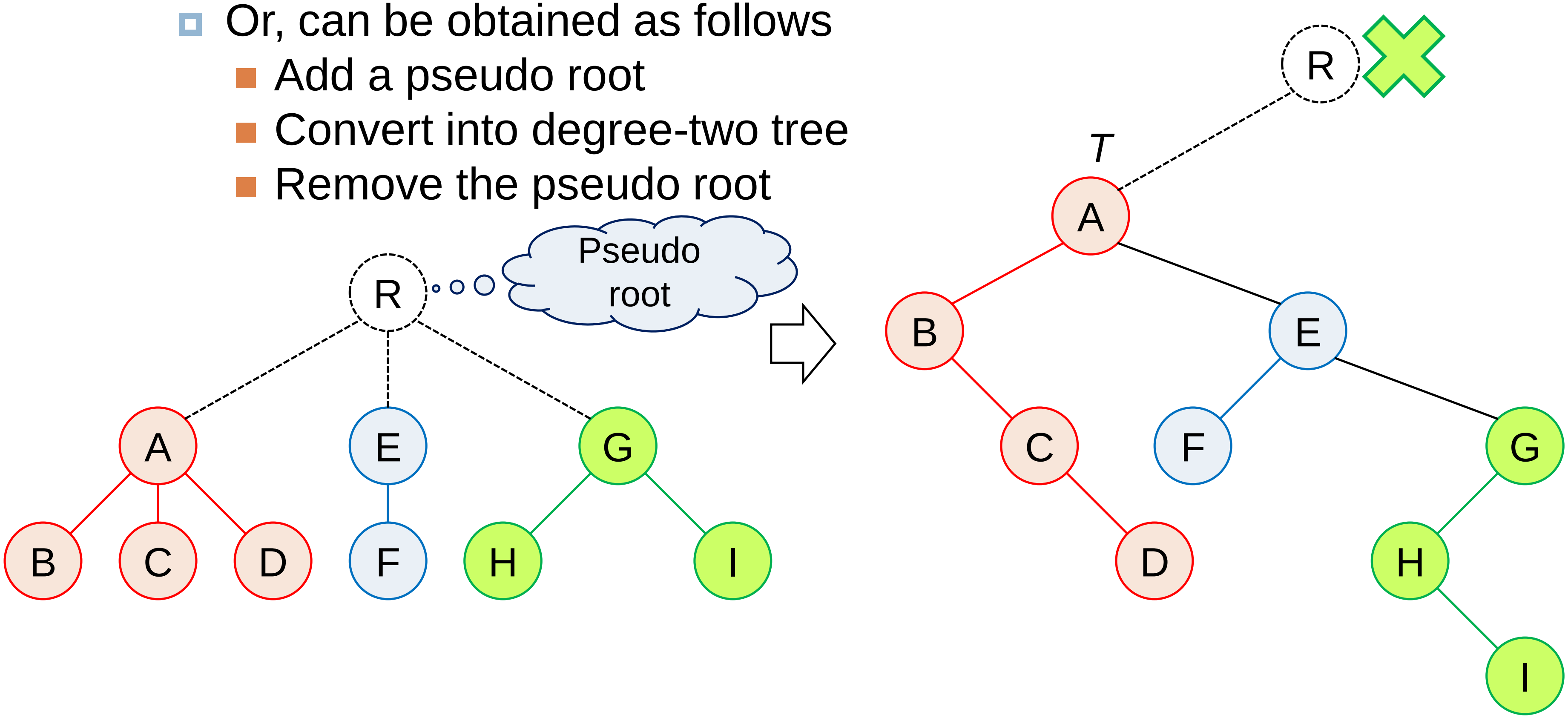
From Forest to Binary Tree (1/2)

- **Definition: If a forest $F = \{T_1, \ldots, T_n\}$, its corresponding binary tree $T = B(T_1, \ldots, T_n)$**
  - Is empty if $n == 0$
  - Is composed of $B(T_1) + \ldots + B(T_n)$, where
    - $B(T_i)$ is T_i's binary tree (ref. Sec. 5.1.2.3, degree-two tree)
    - $root(T) = root(B(T_1))$
    - Right subtree of $root(B(T_i))$ is $B(T_{i+1}, \ldots, T_n)$



From Forest to Binary Tree (2/2)

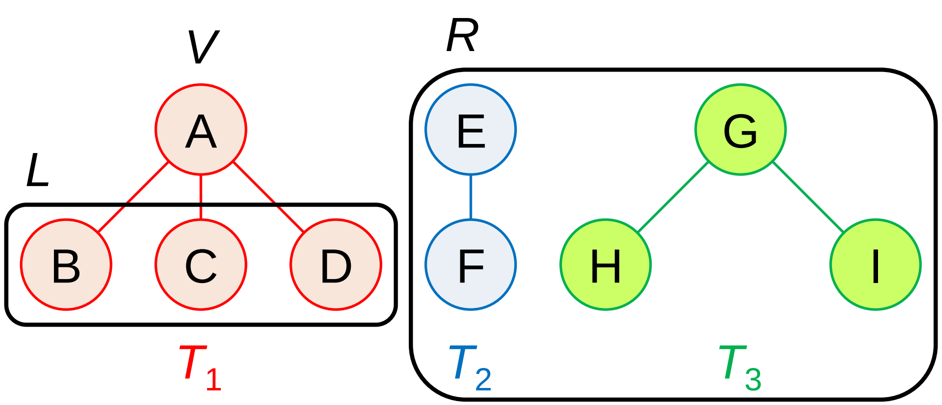
- **Definition: If a forest $F = \{T_1, \ldots, T_n\}$, its corresponding binary tree $T = B(T_1, \ldots, T_n)$**
  - Is empty if $n == 0$
  - Or, can be obtained as follows
    - Add a pseudo root
    - Convert into degree-two tree
    - Remove the pseudo root

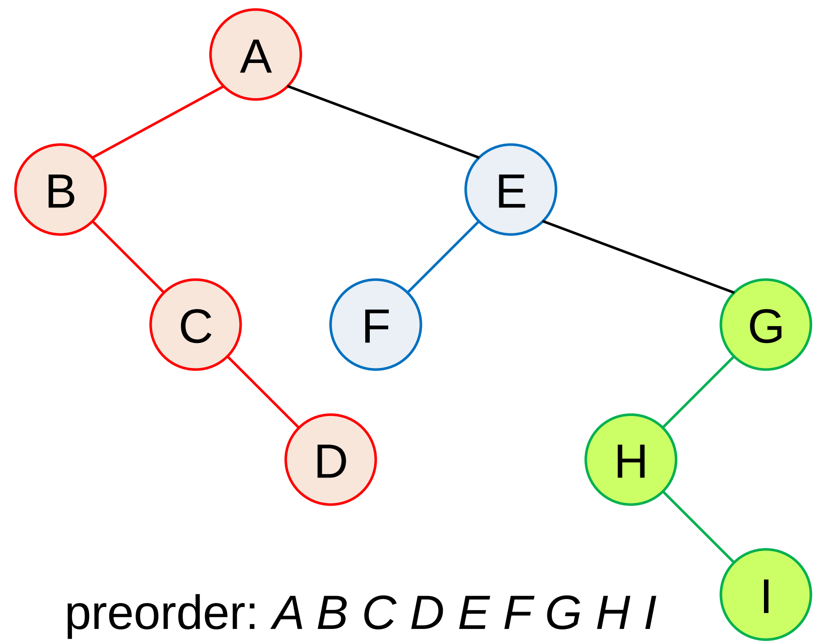


Forest Traversal: Preorder

- **Forest preorder:**
  - If *F* is empty, then return
  - Visit the root of the first tree of *F*
  - Traverse the subtrees of the first tree in forest preorder
  - Traverse the remaining trees of *F* in forest preorder

Forest *F*	**Binary Tree *T***





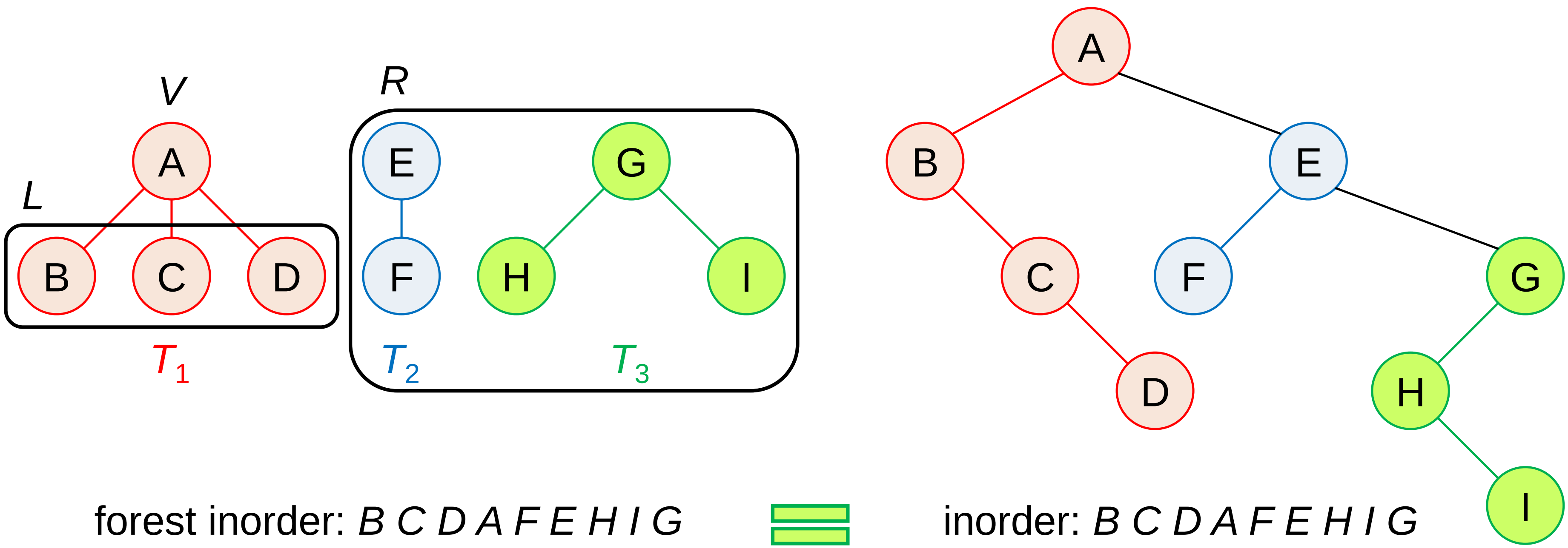
forest preorder: *A B C D E F G H I* = preorder: *A B C D E F G H I*

Forest Traversal: Inorder

 H.-R. Jiang

- **Forest inorder:**
  - If F is empty, then return
  - Traverse the subtrees of the first tree in forest inorder
  - Visit the root of the first tree of F
  - Traverse the remaining trees of F in forest inorder

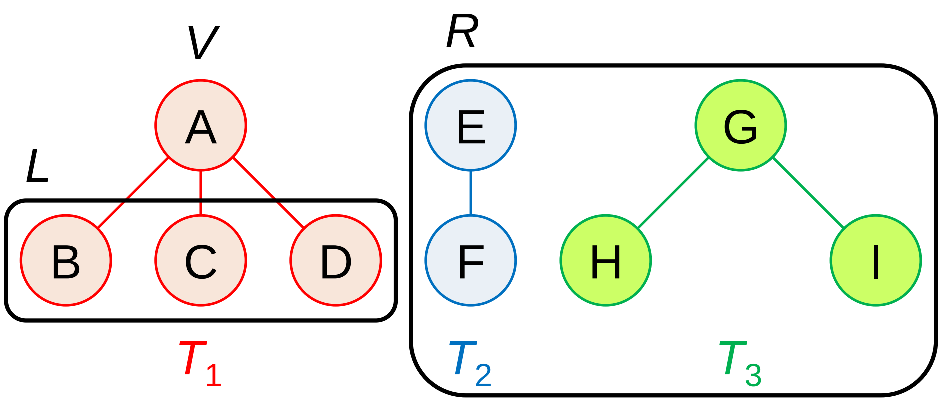
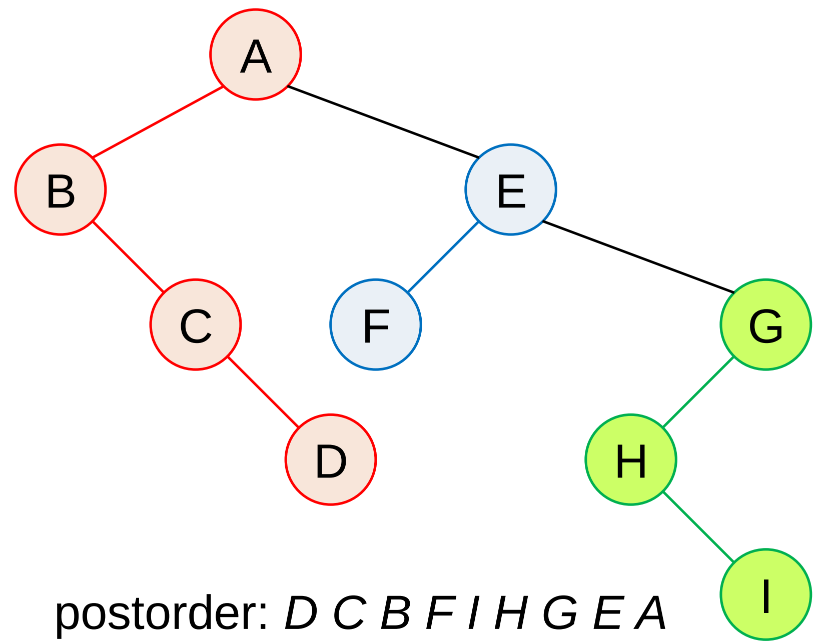
Forest *F*	**Binary Tree *T***



forest inorder: *B C D A F E H I G* = inorder: *B C D A F E H I G*

Forest Traversal: Postorder

- **Forest postorder:**
  - If *F* is empty, then return
  - Traverse the subtrees of the first tree in forest postorder
  - Traverse the remaining trees of *F* in forest postorder
  - Visit the root of the first tree of *F*

Forest *F*	**Binary Tree *T***
	

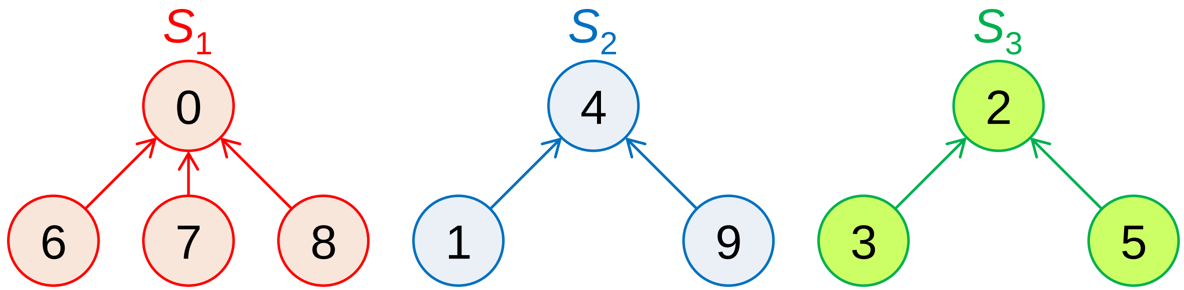
forest postorder: *D C B F I H G E A* = postorder: *D C B F I H G E A*

Disjoint Sets

Tree application

Disjoint Sets

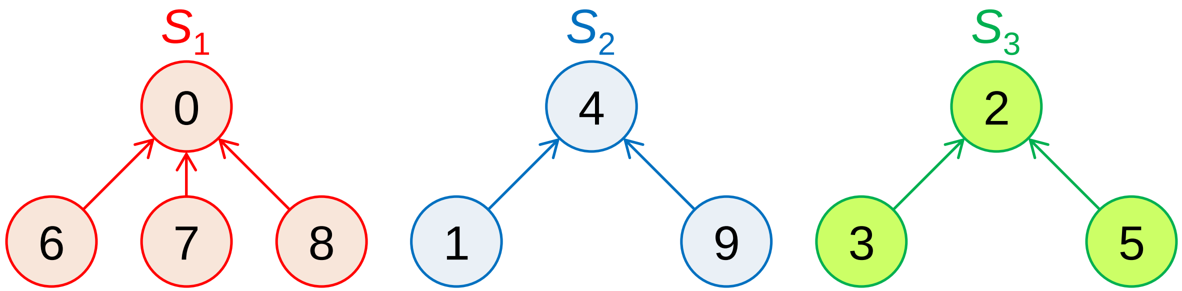
- **Disjoint sets: elements are disjoint**
  - $S_i \cap S_j = \varnothing$
- **Recall: A forest is a set of $n \geq 0$ disjoint trees**
- **$\Rightarrow$ sets $\approx$ trees**
- **Example: $S_1 = \{0, 6, 7, 8\}$, $S_2 = \{1, 4, 9\}$, $S_3 = \{2, 3, 5\}$**



  - Root: representative
    - link: from children to parent

Union & Find

- **Find: Report the set containing the specified element**
  - Example: Find(8) = S_1, Find(3) = S_3



- **Union: Make one of trees a subtree of the other**
  - Example: $S_1 \cup S_2$ = {0, 6, 7, 8, 1, 4, 9}