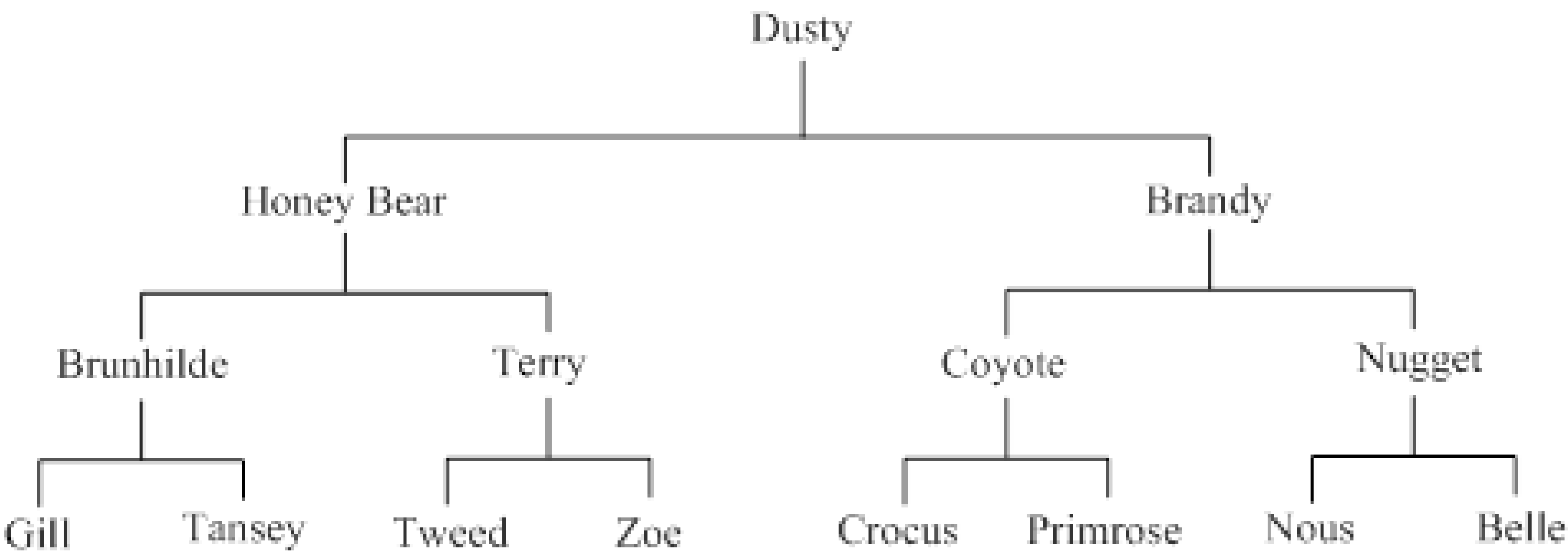
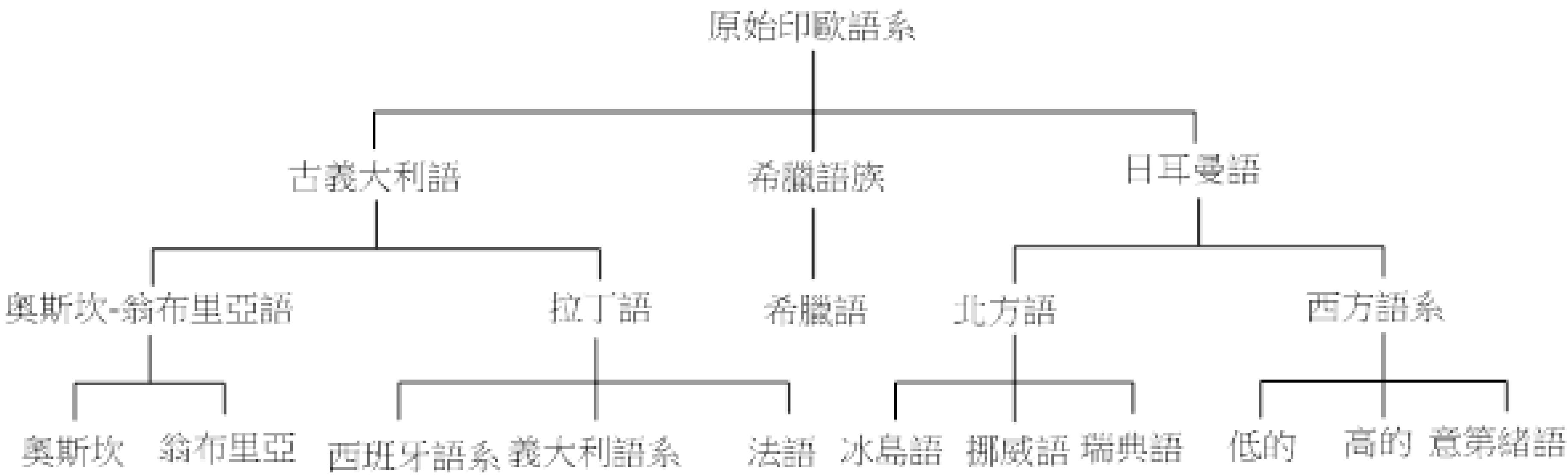
Trees

Pedigree



(a) 血統表

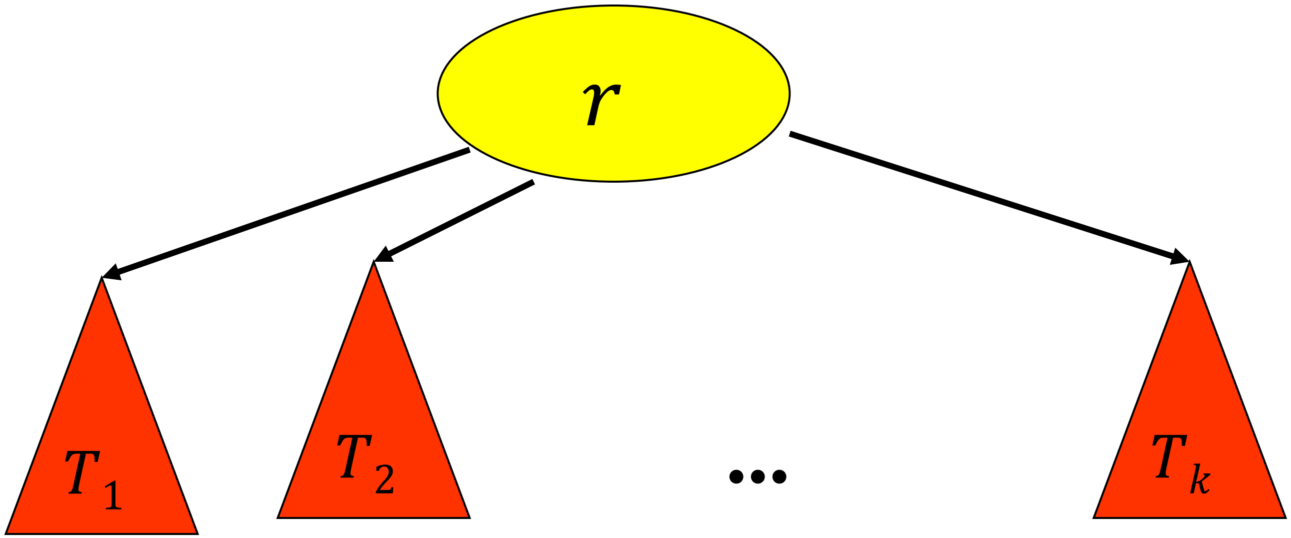
Lineal


原始印歐語系
古義大利語
希臘語族
日耳曼語
奧斯坎-翁布里亞語
拉丁語
希臘語
北方語
西方語系
奧斯坎
翁布里亞
西班牙語系
義大利語系
法語
冰島語
挪威語
瑞典語
低的
高的
意第緒語

(b) 直系表

Trees

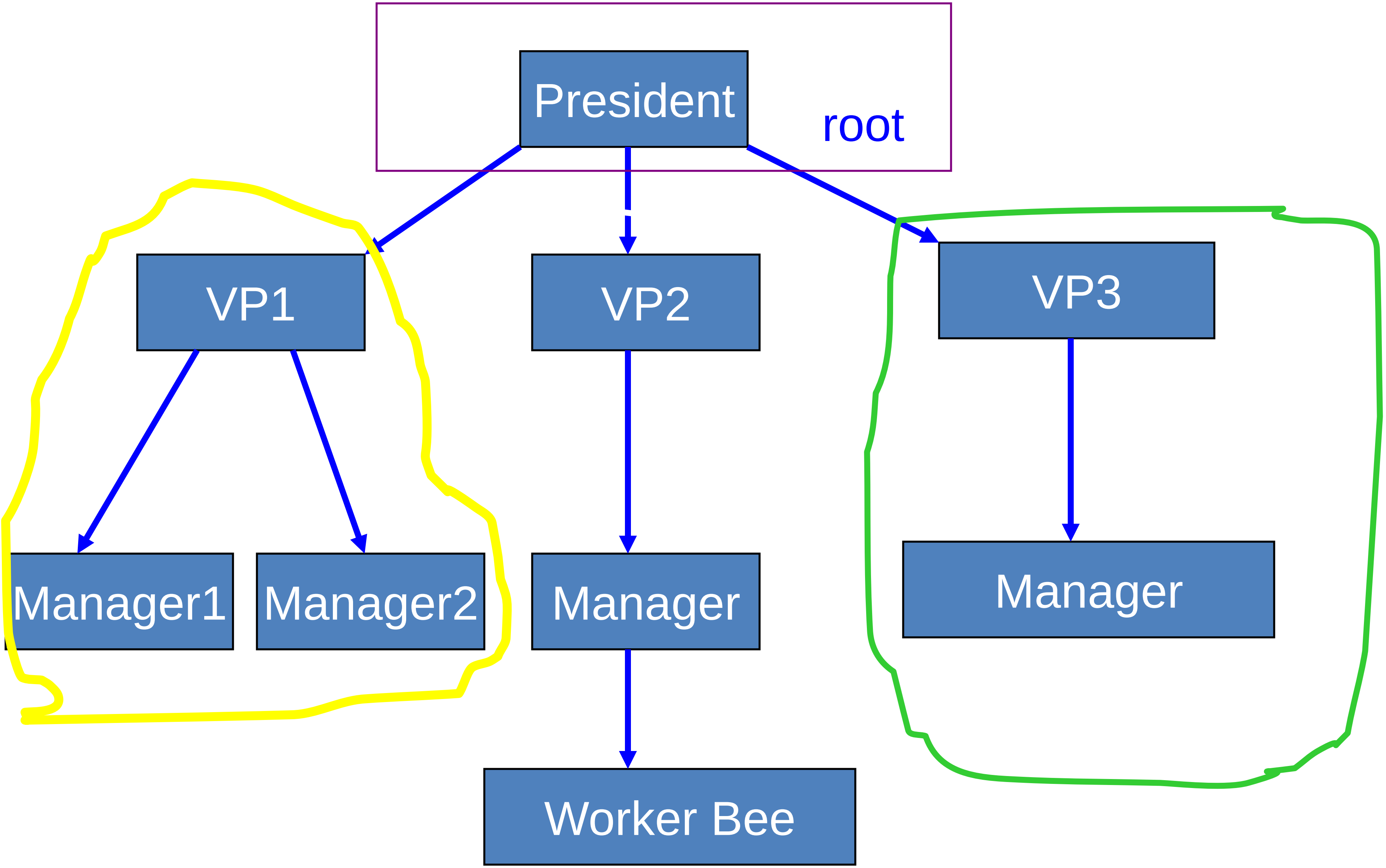
- Tree: a finite set of one or more nodes such that
  - a special node r (root)
  - zero or more nonempty sub(trees) $T_1, T_2, \ldots, T_k$ each of whose roots are connected by a directed edge from r



Definition

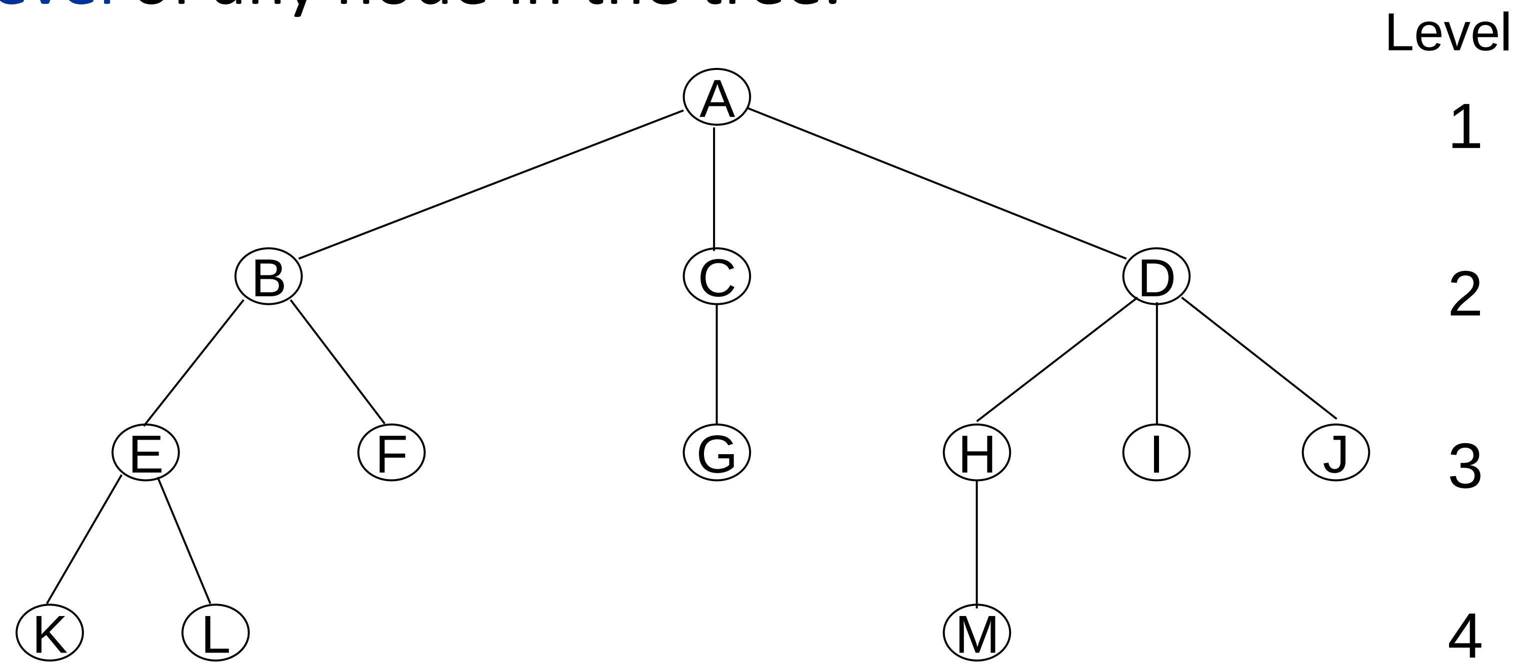
- A tree t is a finite nonempty set of elements.
- One of these elements is called the root.
- The remaining elements, if any, are partitioned into trees, which are called the subtrees of t .

Subtrees

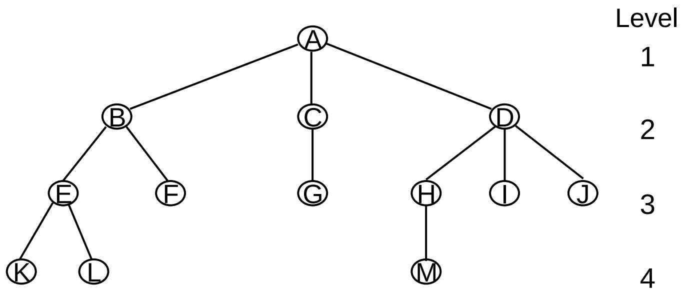

President
root
VP1
VP2
VP3
Manager1
Manager2
Manager
Manager
Worker Bee

A Sample Tree

- The root is at level 1
- The level of a node is the level of the node's parent $+ 1$.
- The height or the depth of a tree is the **maximum** level of any node in the tree.



Terminology



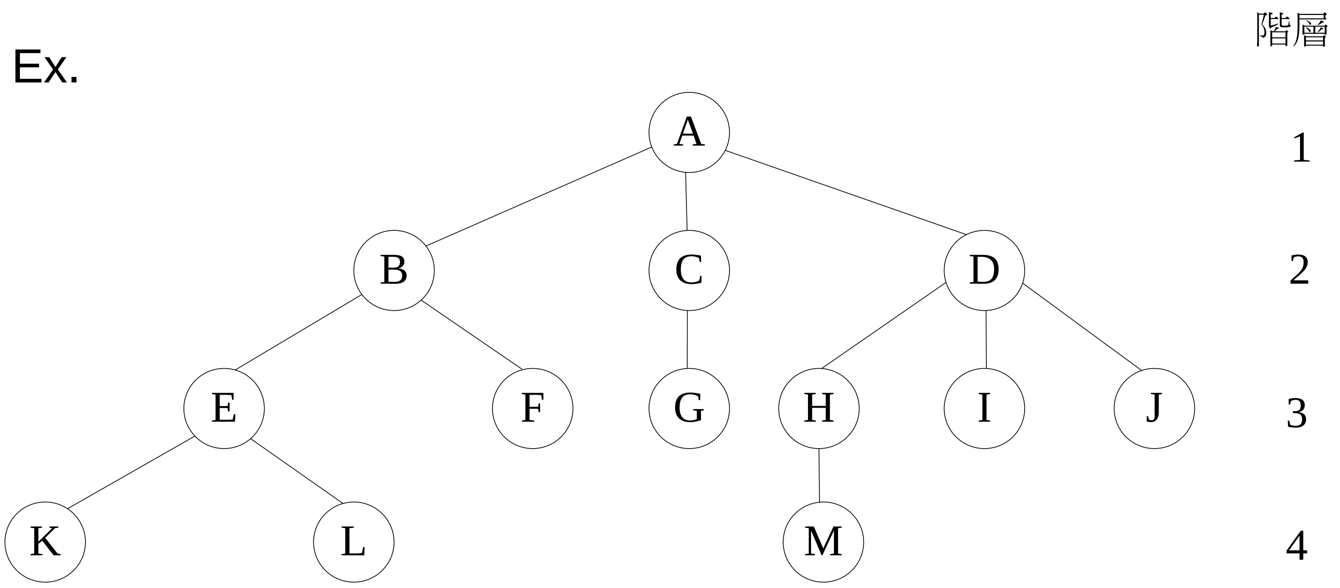
- The **degree** of a node is the number of subtrees of the node
  - The degree of A is 3; the degree of C is 1.
- The node with degree 0 is a **leaf** or **terminal** node.
  - The others are **non-terminal**
- A node that has subtrees is the **parent** of the roots of the subtrees.
  - The roots of these subtrees are the **children** of the node.
- Children of the same parent are **siblings**.
- The **ancestors** of a node are all the nodes long the path from the root to the node.

Representation of Trees

- **List Representation**
  – The root comes first, followed by a list of sub-trees

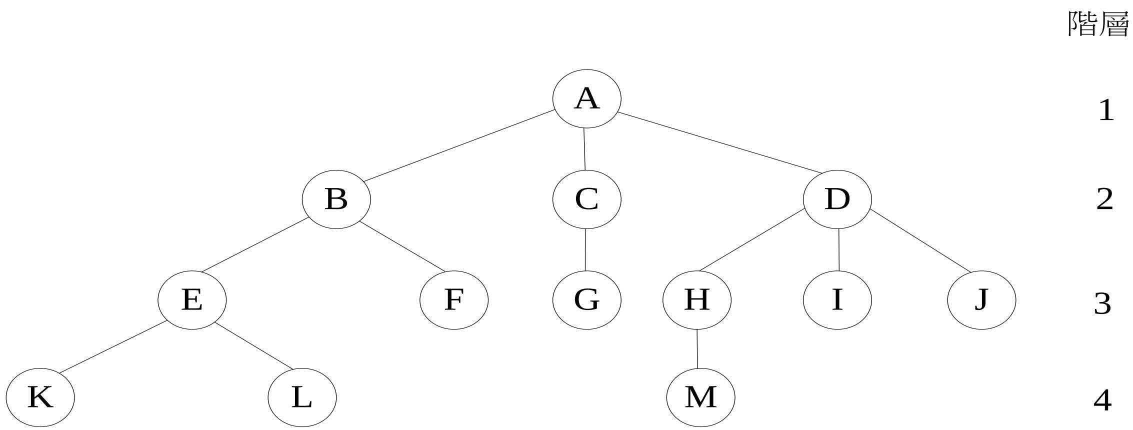
$$T = (root\ (T_1, T_2, \dots, T_n))$$

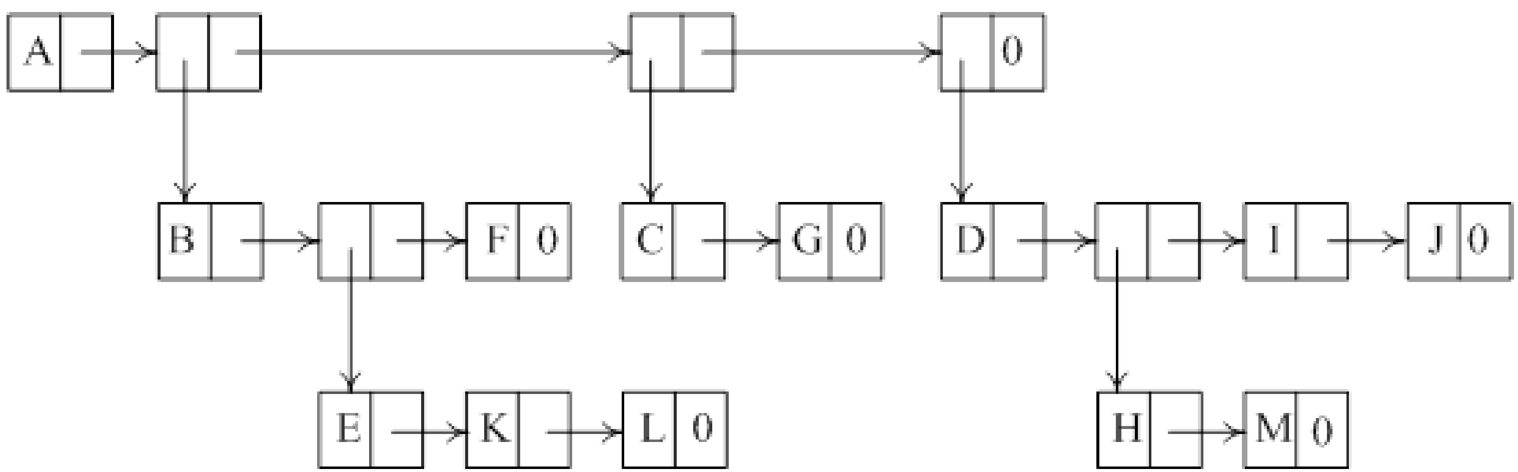
Ex.



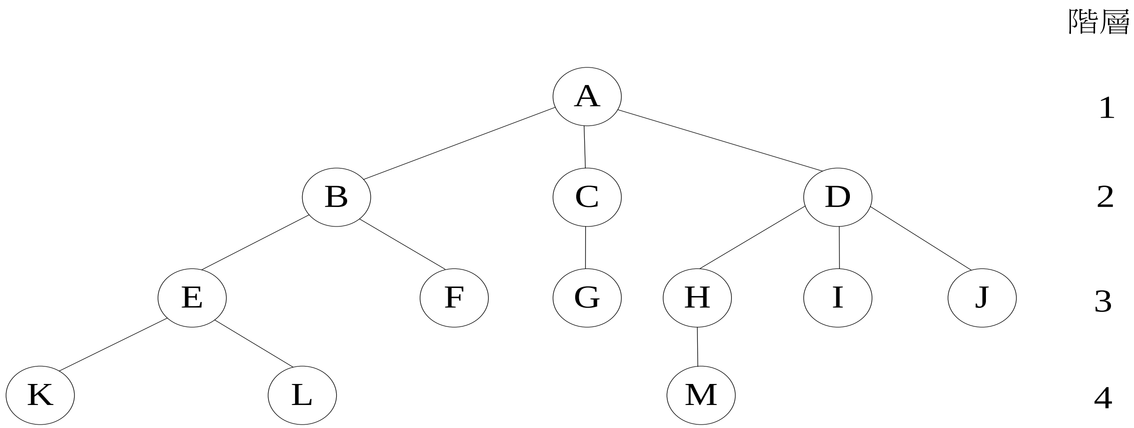
(A (B (E (K, L), F), C (G) , D (H (M), I, J)))

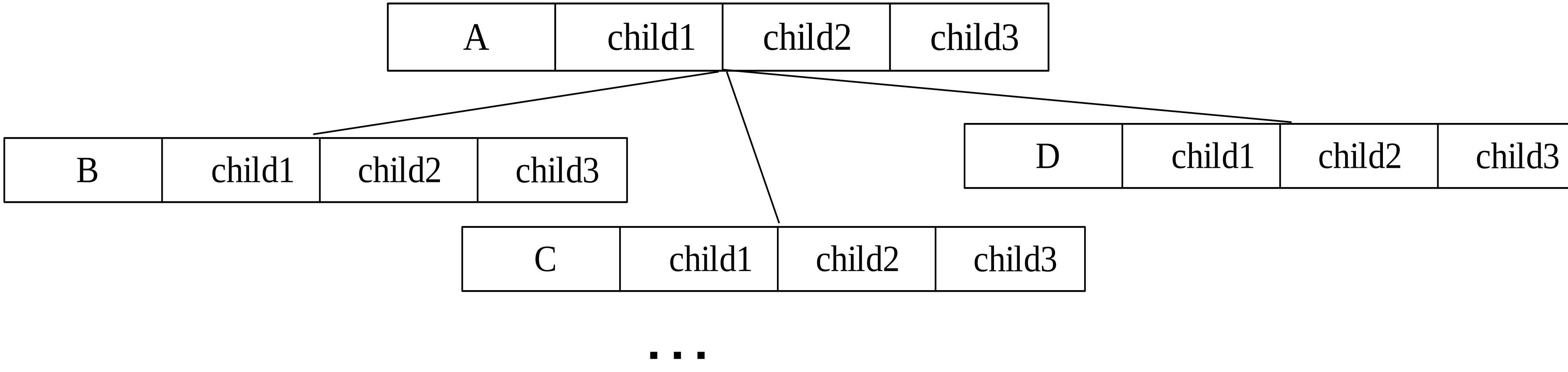
List Representation of Trees





Possible Node Structure for a Tree of Degree k





Possible Node Structure for a Tree of Degree k

- Lemma 5.1: If T is a k-ary tree (i.e., a tree of degree k) with n nodes, each having a fixed size as below, then $n(k-1)\ +\ 1$ of the nk child fields are 0, $n\ \geq\ 1$.

data	child1	child2	…	child k

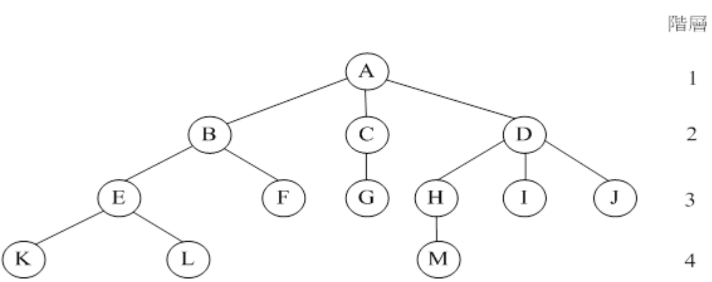
$$nk - (n-1) = n(k-1) + 1$$

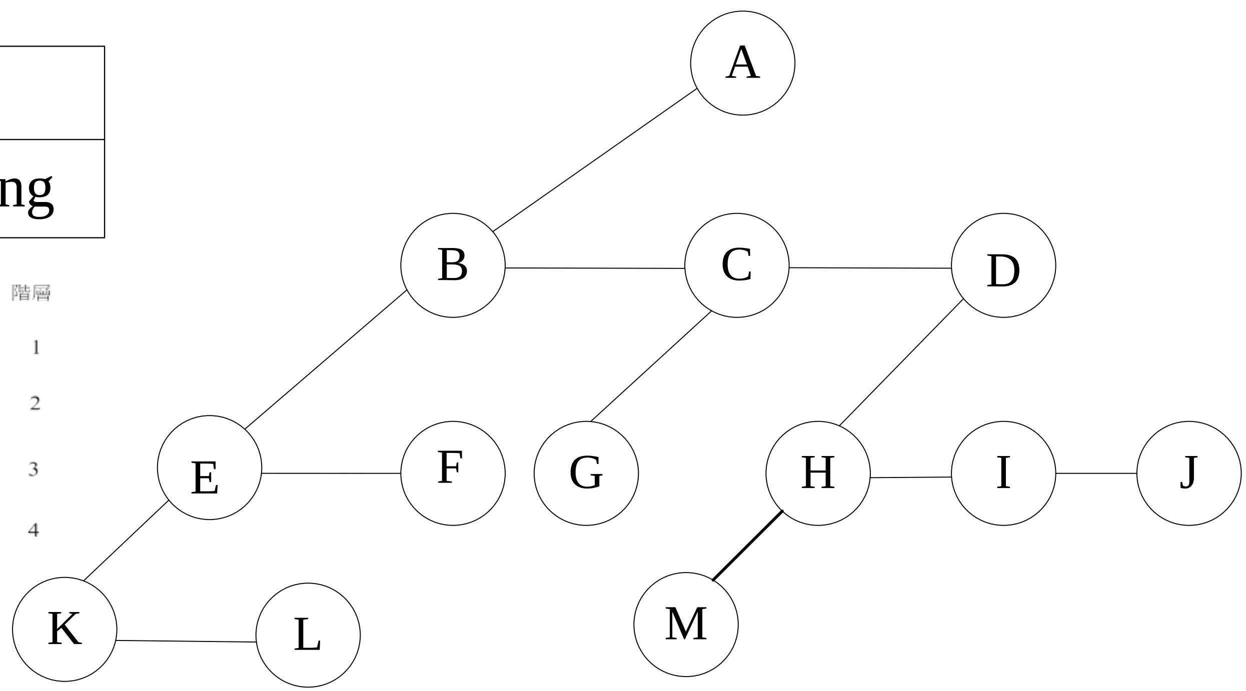
Wasting memory!

Representation of Trees

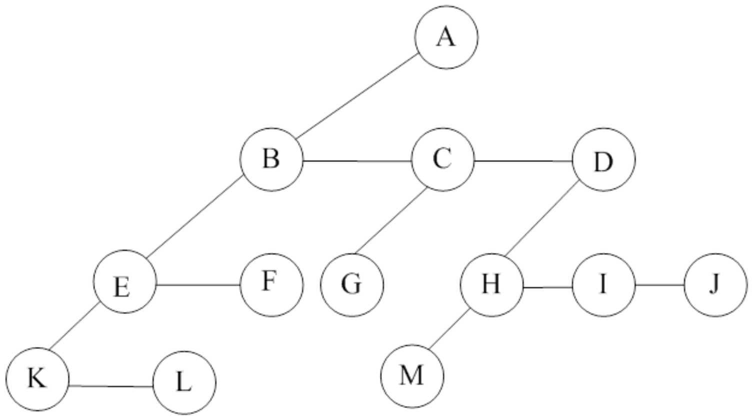
- **Left Child-Right Sibling** Representation
  - Each node has two links (or pointers).
  - Each node only has one leftmost child and one closest sibling.

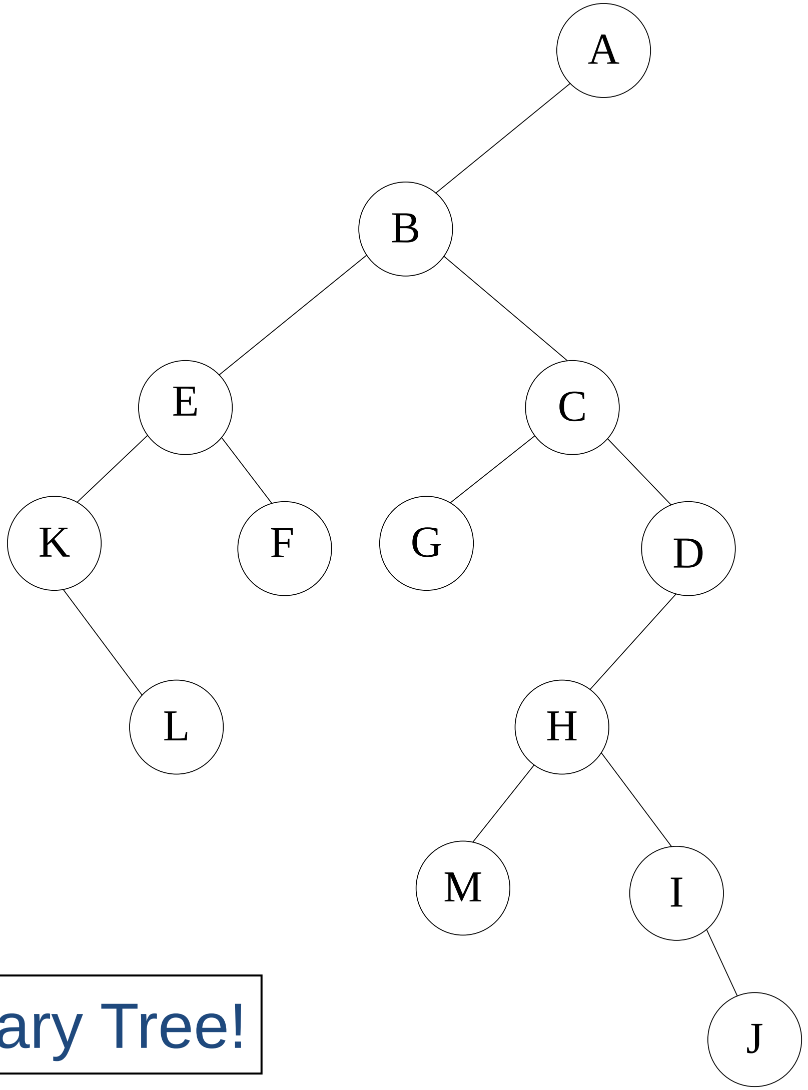
data	
Left child	Right Sibling





Degree Two Tree Representation



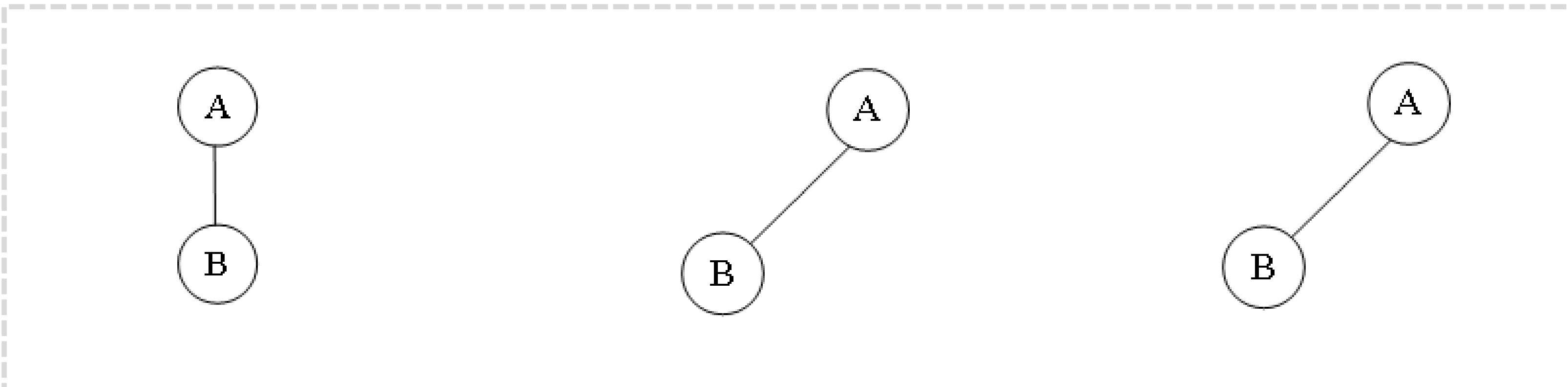


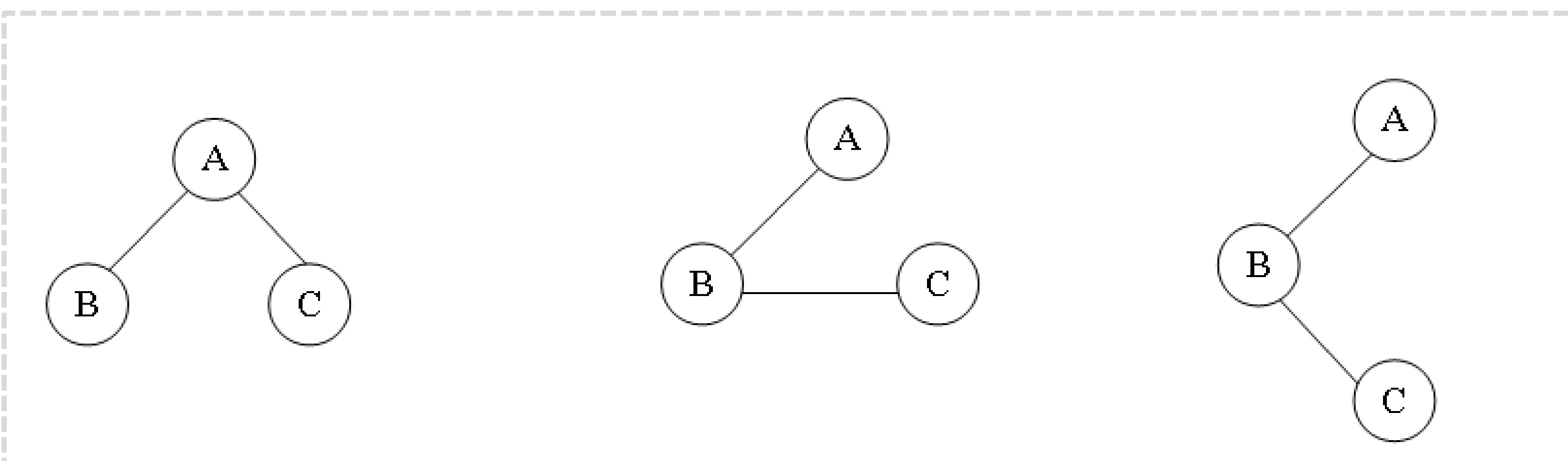
Rotate the right-sibling pointers in a left child-right sibling tree clockwise by 45°

Binary Tree!

Tree Representations

tree | Left child-right sibling | Binary tree





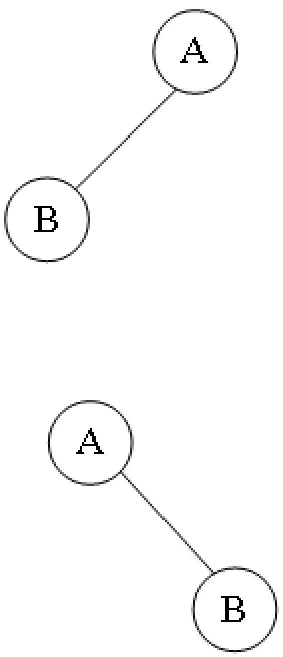
Binary Tree

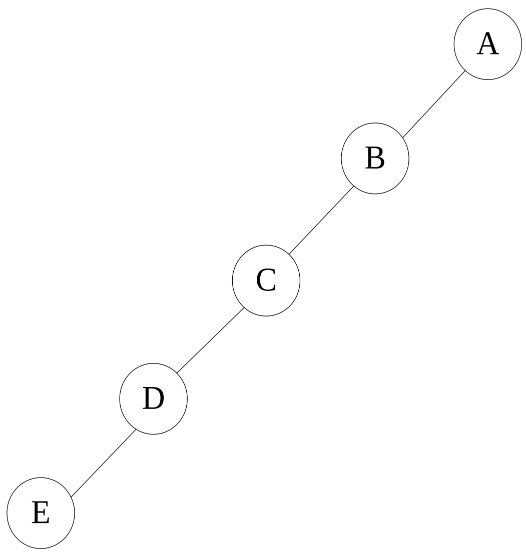
- **Definition**:
  - A binary tree is a finite set of nodes that is either empty or consists of a *root* and two disjoint binary trees called the $left\ subtree$ and the $right\ subtree$.
- There is no tree with zero nodes. But there is an empty binary tree.
- Binary tree distinguishes between the order of the children while in a tree we do not.

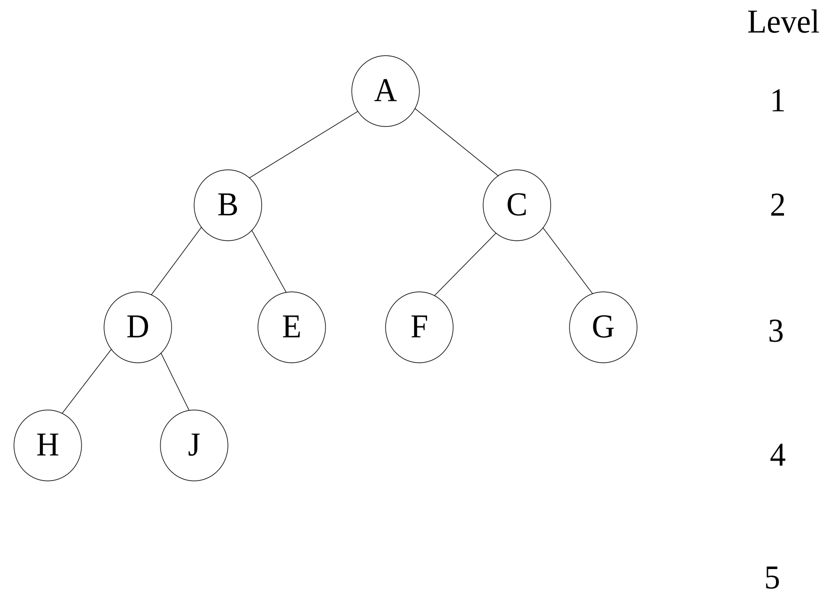
Distinctions between a Binary Tree and a Tree

	Binary tree	Tree
degree	≤ 2	Not limited
order of the subtrees	$\surd$	$\times$
allow zero nodes	$\surd$	$\times$

Binary Tree Examples







Skewed binary tree

Complete binary tree

```
template<class T>
class BinaryTree
{ // object：a finite set of nodes either empty or consisting of a root node,
  // left BinaryTree and rightBinaryTree。
public:
      BinaryTree();
      // creaes an empty binary tree
      bool IsEmpty();
      // return true iff the binary tree ie empty
      BinaryTree(BinaryTree<T>& bt1, T& item, BinaryTree <T>& bt2);
      // creaes an binary tree whose left subtree is bt1, whose right subtree bt2，
      // and whose root node contain item
      BinaryTree<T> LeftSubtree();
      // return the right subtree of *this
      BinaryTree<T> RightSubtree();
      // return the left subtree of *this
      T RootData();
      // return the data in the root node of *this
};
```

The Properties of Binary Trees

- **Lemma 5.2** [Maximum number of nodes]
  1) The maximum number of nodes on level i of a binary tree is $2^{i-1}, i \geq 1$.
  2) The maximum number of nodes in a binary tree of depth k is $2^k - 1, k \geq 1$.
- **Lemma 5.3** [Relation between number of leaf nodes and nodes of degree 2] For any non-empty binary tree, T, if n_0 is the number of leaf nodes and n_2 the number of nodes of degree 2, then

$$n_0 = n_2 + 1.$$

- **Definition**: A full binary tree of depth k is a binary tree of depth k having 2^{k-1}nodes, $k \geq 0$.

Maximum Number of Nodes in Binary Trees

- The maximum number of nodes on level i of a binary tree is $2^{i-1}, i \geq 1$.

  Prove by induction.

  1. Max. no. of node on level $i = 1$ is $2^{1-1} = 1$
  2. Assume the max. no. of node on level $i - 1$ is 2^{i-2}
  3. Since the max. degree of nodes in a binary tree is 2, the max. no. of node on level i is $2 \times 2^{i-2}$=2^{i-1}

- The maximum number of nodes in a binary tree of depth k is $2^{k-1}, k \geq 1$.

$$\sum_{i=1}^{k} 2^{i-1} = 2^{k} - 1$$

Relations between Number of Leaf Nodes and Nodes of Degree 2

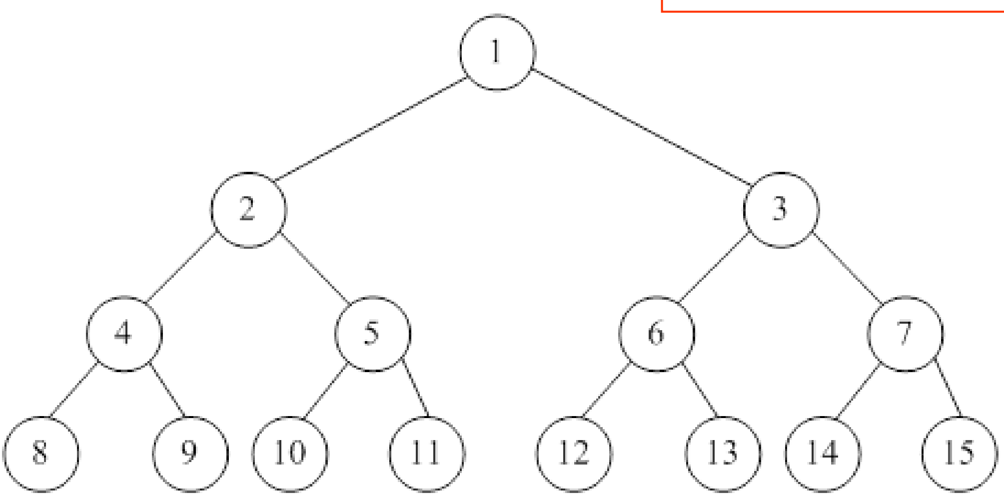
- For any nonempty binary tree, T, if n_0 is the number of leaf nodes and n_2 the number of nodes of degree 2, then $n_0 = n_2 + 1$.
- proof:
  - Let n and B denote the total number of nodes & branches in T.
  - Let n_0, n_1, n_2 represent the nodes with no children, single child, and two children respectively.
  - $n = n_0 + n_1 + n_2, B = n - 1,$

    $B = n_1 + 2n_2$ ==> $n_1 + 2n_2 + 1 = n,$
  - $n_1 + 2n_2 + 1 = n_0 + n_1 + n_2$ ==> $n_0 = n_2 + 1$

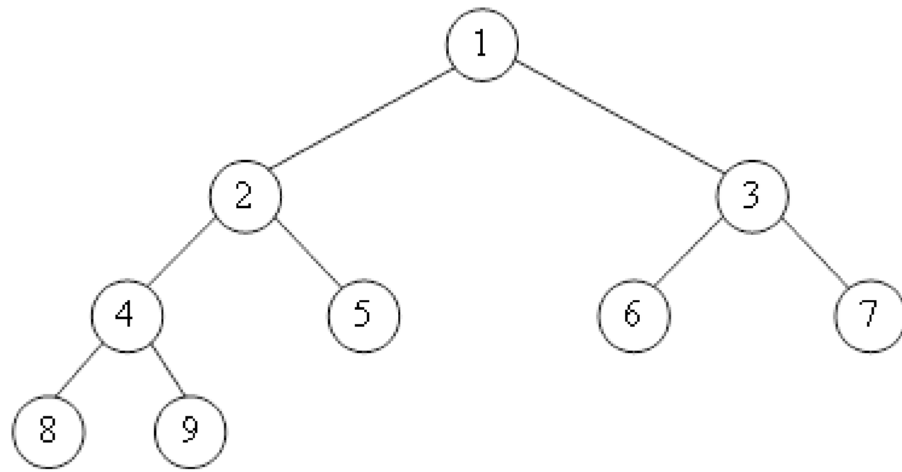
Full BT vs Complete BT

- A full binary tree of depth k is a binary tree of depth k having 2^{k-1} nodes, $k \geq 0$.
- A binary tree with n nodes and depth k is complete iff its nodes correspond to the nodes numbered from 1 to n in the full binary tree of depth k.

Numbering from top to bottom, to left to right



Full binary tree of depth 4

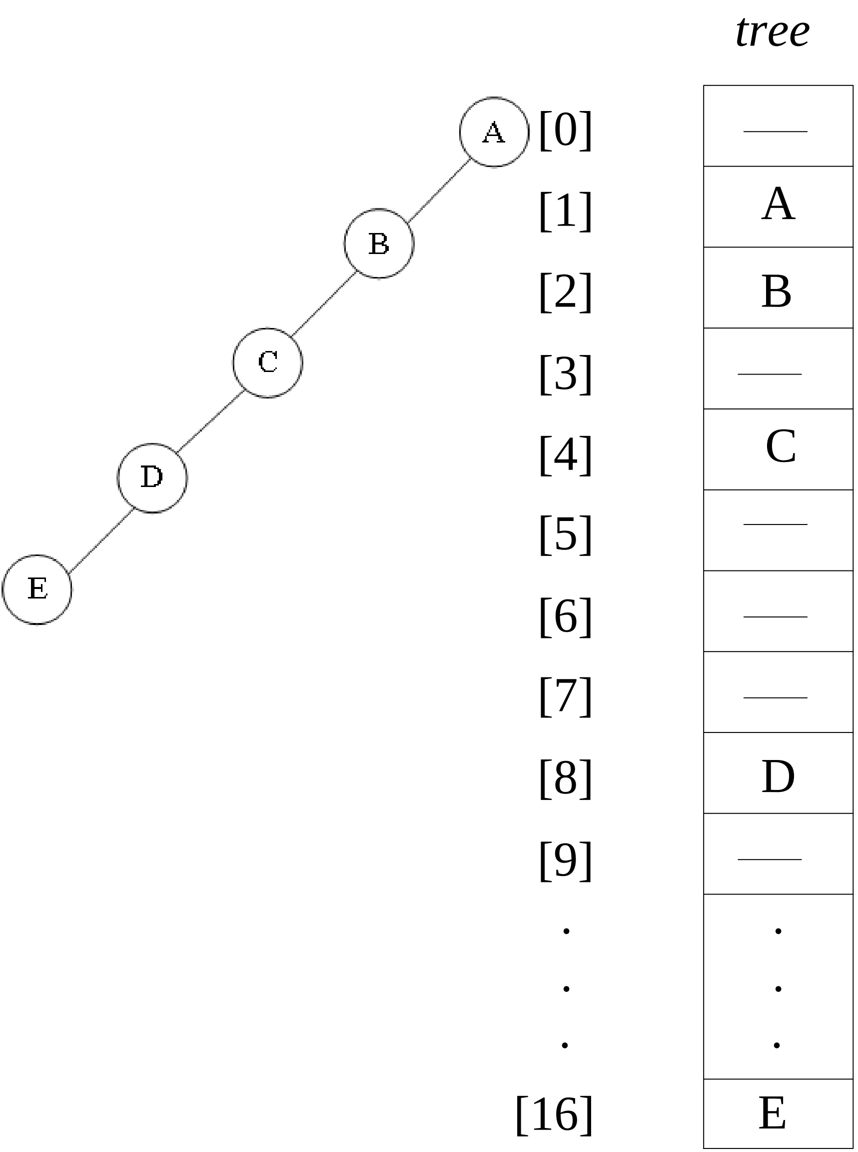


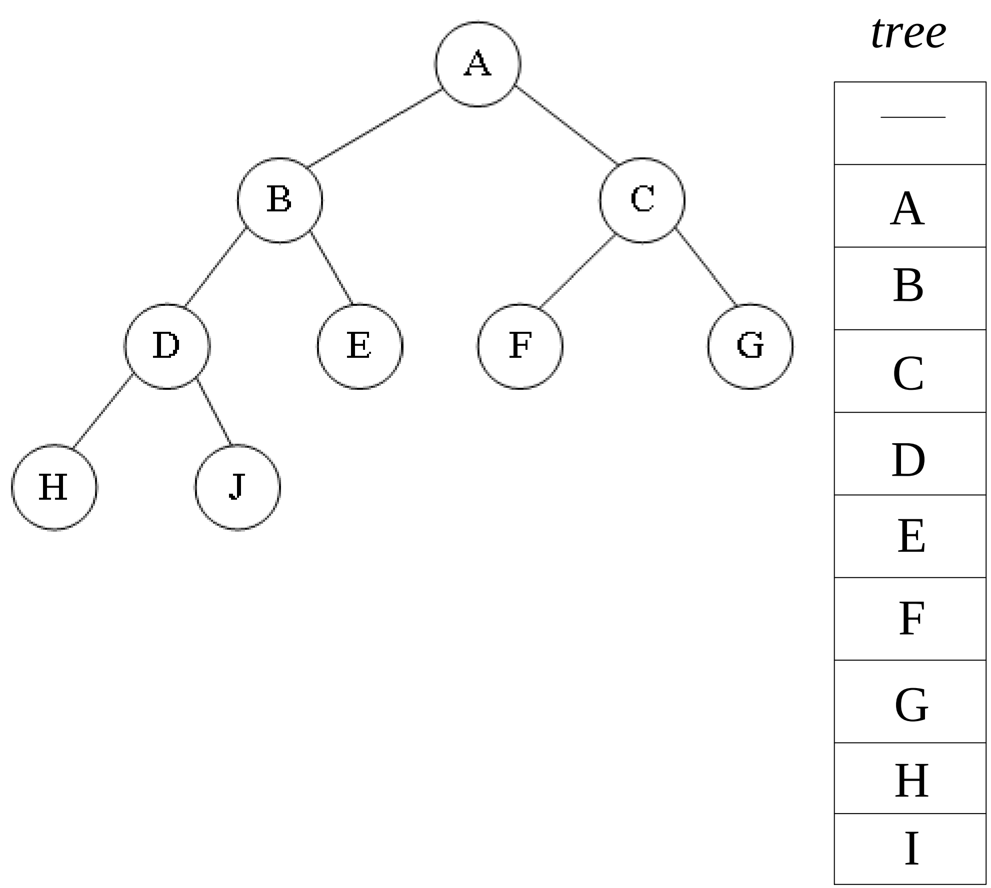
Complete binary tree

Array Representation of a Binary Tree

- **Lemma 5.4**: If a complete binary tree with n nodes is represented sequentially, then for any node with index $i, 1 \leq i \leq n$, we have:
  - $parent(i)$ is at $\lfloor i/2 \rfloor$ if $i \neq 1$. If $i = 1$, i is at the root and has no parent.
  - $left\_child(i)$ is at $2i$ if $2i \leq n$. If $2i > n$, then i has no left child.
  - $right\_child(i)$ is at $2i + 1$ if $2i + 1 \leq n$. If $2i + 1 > n$, then i has no right child.

Sequential Representation



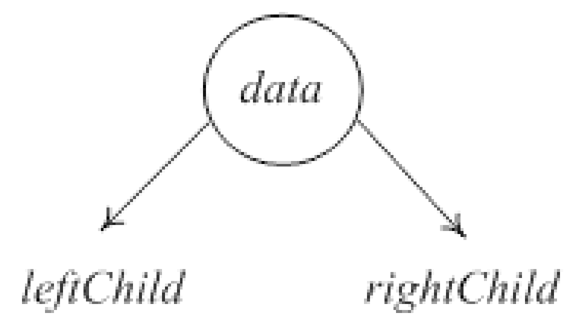


(1) waste space

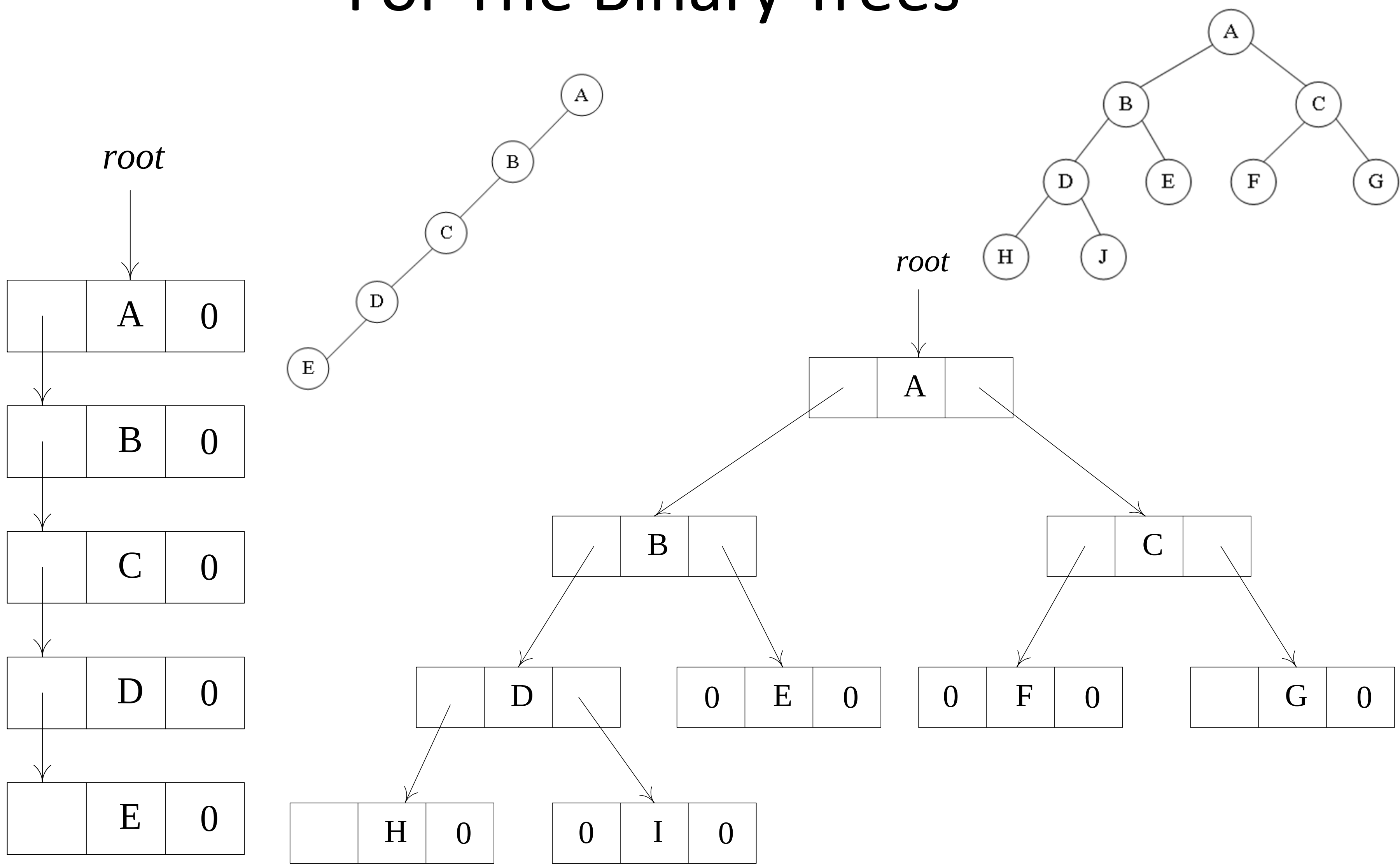
(2) insertion/deletion problem

Linked Representation

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



Linked List Representation For The Binary Trees



Compare Two Binary Tree Representations

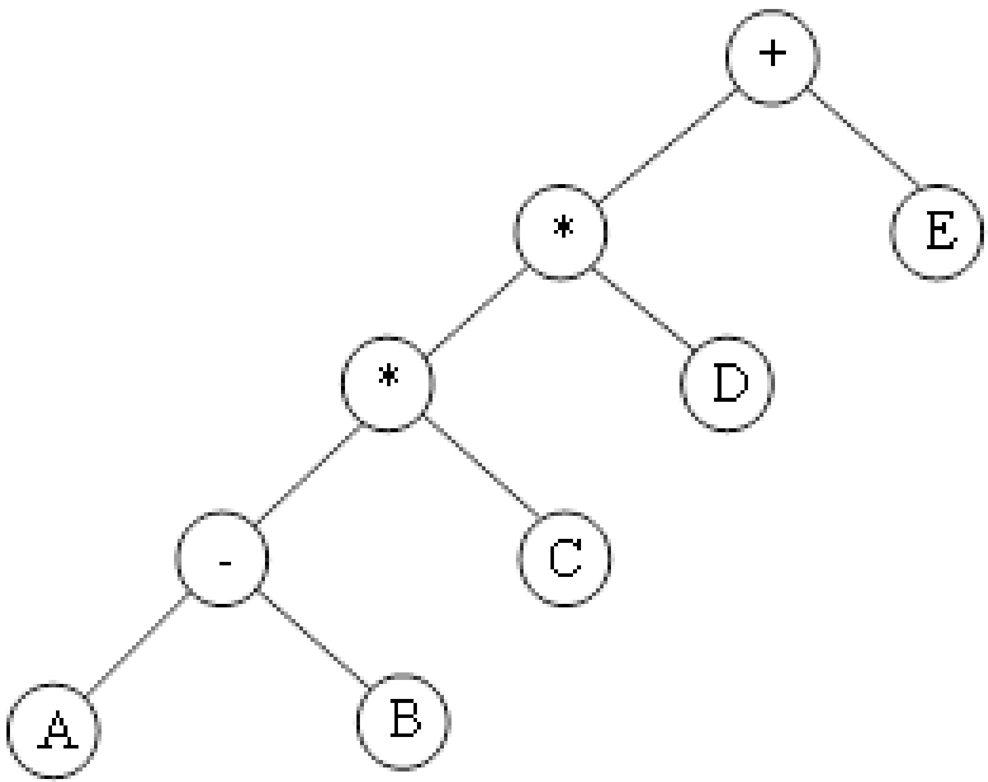
	Array representation	Linked representation
Determination the locations of the parent, left child and right child	Easy	Difficult
Space overhead	Much	Little
Insertion and deletion	Difficult	Easy

Binary Tree Traversals

- Let L, V, and R stand for moving left, visiting the node, and moving right.
  - There are six possible combinations of traversal

    LVR, LRV, VLR, VRL, RVL, RLV
- Adopt convention that we traverse left before right, only 3 traversals remain
  - LVR, LRV, VLR
  - inorder, postorder, preorder

Arithmetic Expression Using BT



inorder traversal

A / B * C * D + E

preorder traversal

+ * * / A B C D E

postorder traversal

A B / C * D * E +

level order traversal

+ * E * D / C A B

Inorder Traversal of A Binary Tree

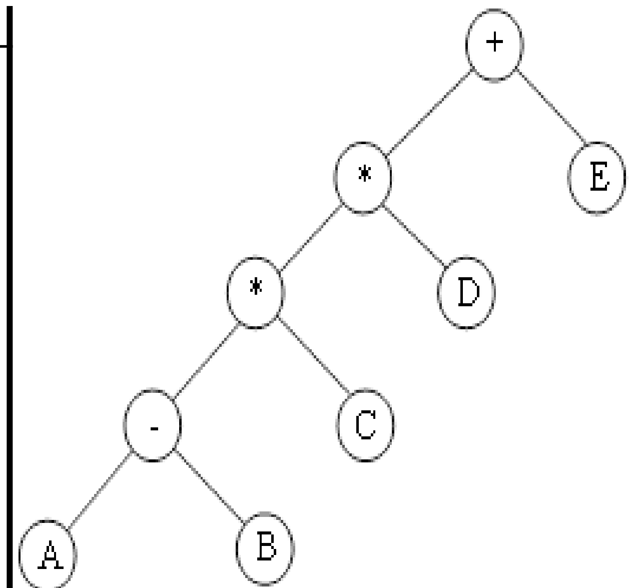
```
template <class T>
void Tree <T>::Inorder()
{ // driver calls workhosrse for traversal of entire tree. The driver
  // is declared as a public member function of Tree
    Inorder(root);
}

template <class T>
void Tree <T>::Inorder(TreeNode <T> *currentNode)
{ // workhorse traverses the subtree rooted at currentNode
  // The workhorse is declared as a private member function of Tree
    if (currrentNode) {
        Inorder(currentNode→leftChild);
        Visit(currentNode);
        Inorder(currentNode→rightChild);
    }
}
```

visit: cout<<current->data;

Trace Operations of Inorder Traversal

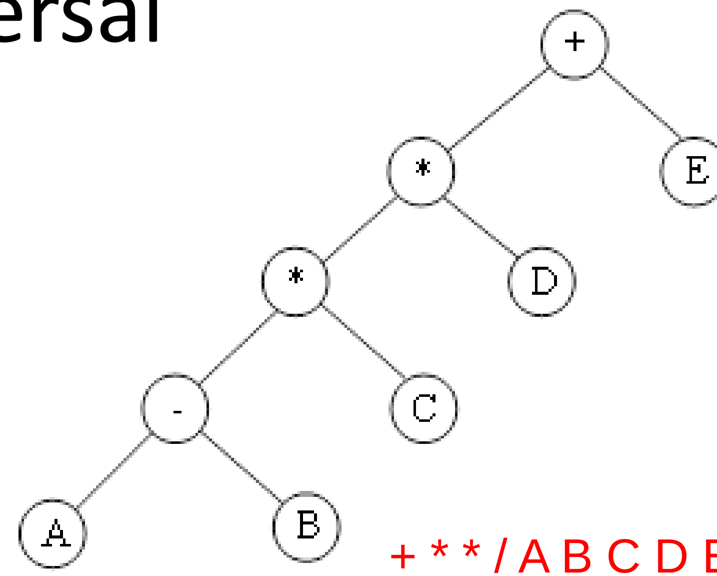
```
    if (currrentNode) {
        Inorder(currentNode→leftChild);
        Visit(currentNode);
        Inorder(currentNode→rightChild);
    }
}
```

Call of inorder	Value in root	Action	Call of inorder	Value in root	
1	+		11	C	
2	*		12	NULL	
3	*		11	C	cout
4	/		13	NULL	
5	A		2	*	cout
6	NULL		14	D	
5	A	cout	15	NULL	
7	NULL		14	D	cout
4	/	cout	16	NULL	
8	B		1	+	cout
9	NULL		17	E	
8	B	cout	18	NULL	
10	NULL		17	E	cout
3	*	cout	19	NULL	



A / B * C * D + E

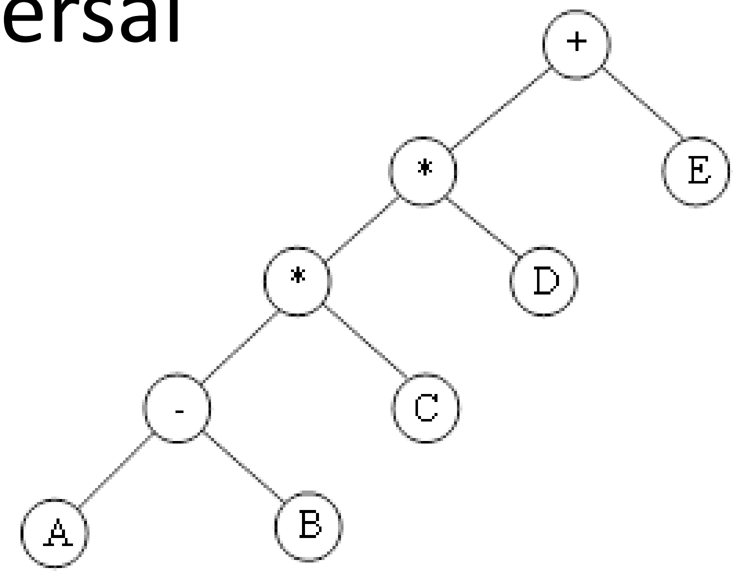
Preorder Traversal



+ * * / A B C D E

```
template < class T>
void Tree <T>::Preorder()
{// driver
        Preorder(root);
}

template < class T>
void Tree <T>::Preorder(TreeNode<T> *currentNode)
{// workhorse
        if (currentNode) {
                Visit(currentNode);
                Preorder(currentNode→leftChild);
                Preorder(currentNode→rightChild);
        }
}
```

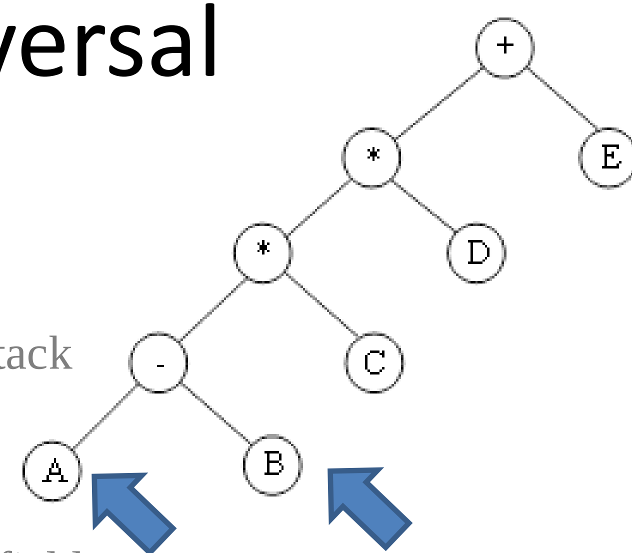
Postorder Traversal



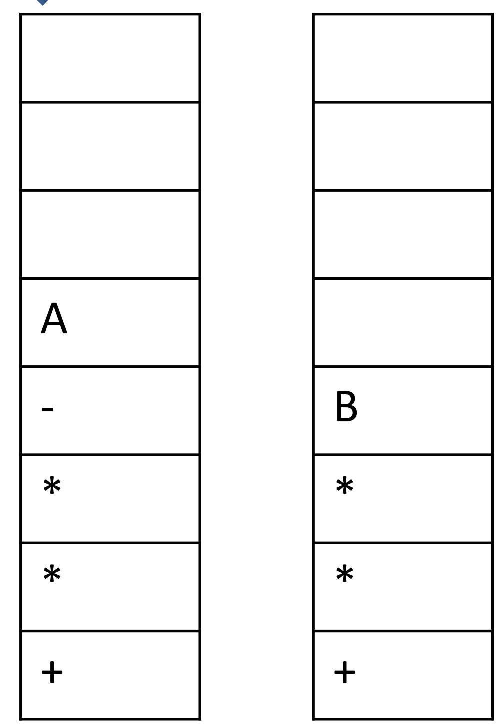
A B / C * D * E +

```
template < class T>
void Tree <T>::Postorder()
{// driver
      Postorder(root);
}

template < class T>
void Tree <T>::Postorder(TreeNode<T> *currentNode)
{// workhorse
      if (currentNode) {
            Postorder(currentNode→leftChild);
            Postorder(currentNode→rightChild);
            Visit(currentNode);
      }
}
```

Iterative Inorder Traversal




```
template < class T>
void Tree <T>::NonrecInorder()
{// Nonrecursiveinorder traversal using a stack
        Stack < TreeNode < T > * > s;    // Declare and init stack
        TreeNode < T > *currentNode = root;
        while(1) {
                while (currentNode) { // move down leftChild fields
                        s.Push(currentNode);
                        currentNode = currentNode→leftChild;
                }
                if (s.IsEmpty()) return;
                currentNode = s.Top();
                s.Pop(); // delete from stack
                Visit(currentNode);
                currentNode = currentNode→rightChild;
        }
}
```



Using Iterator to Get Next Item

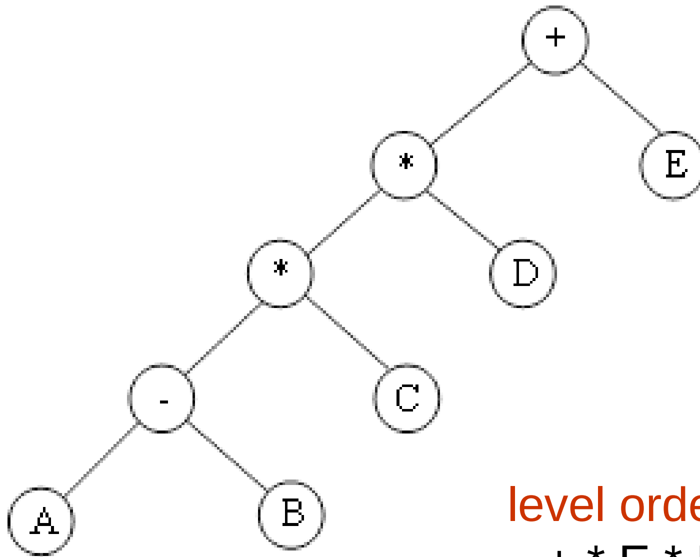
```
class InorderIterator {
public:
        InorderIterator() { currentNode = root; }
        T* Next();
private
        Stack <TreeNode <T>* > s;
        TreeNode<T> *currentNode;
};
```

```
T* InorderIterator::Next()
{
        while (currentNode) {
                s.Push(currentNode);
                currentNode = currentNode→leftChild;
        }
        if (s.Isempty()) return 0;
        currentNode = s.Top();
        s.Pop(); //delete the top
        T& temp = currentNode→data;
        currentNode = currentNode→rightChild;
        return &temp;
}
```

Level-Order Traversal

- All previous mentioned schemes use stacks
- Level-order traversal uses a queue
  - Queue: First In First Out
- Level-order scheme visit the root first, then the root’s left child, followed by the root’s right child
  - Level by level
- All the nodes at a level are visited before moving down to another level

Level-Order Traversal



level order traversal
+ * E * D / C A B

```
template <class T>
void Tree <T>::LevelOrder()
{// traversae 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();
    }
}
```

Some Other Binary Tree Functions

- With the inorder, postorder, or preorder mechanisms, we can implement all needed binary tree functions. e.g.,
  - Copying Binary Trees
  - Testing Equality
    - Two binary trees are equal if their topologies are the same and the information in corresponding nodes is identical.

Copying Binary Trees

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

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

Repeating down the tree

Testing Equality

```
template <class T>
bool Tree<T>::operator = = (const Tree& t) const
{
        return Equal(root, t.root);
}

template <class T>
bool Tree<T>::Equal(TreeNode<T>* a , TreeNode<T>* b)
{ // workhosrse
        if ((!a) && (!b)) return true; //
        return (a && b     // both a and bare non-zero
                    && ( a→data == b→data)    // data is the same
                    && Equal(a→leftChild,b→leftChild)     // left subtrees equal
                    && Equal(a→rightChild,b→rightChild));    // right subtrees equal
}
```

Repeating down the tree

Satisfiability Problem

- Formulas of the Propositional Calculus

  The set of expressions formed by using variables x_1, x_2, x_3, and operators $\neg$ (not), $\wedge$ (and), $\vee$ (or), as well as the following rules

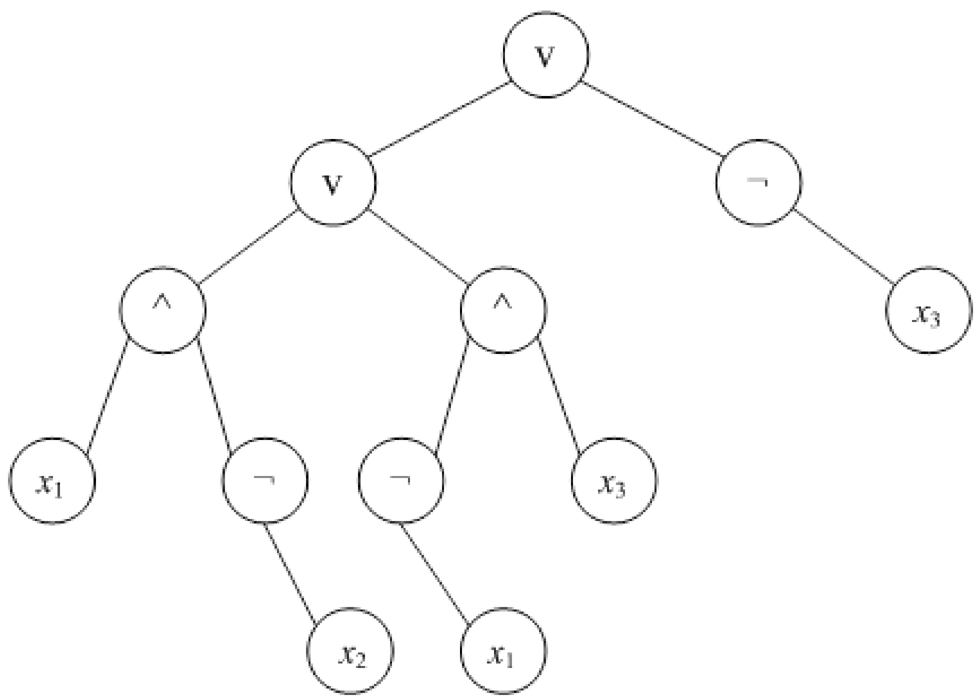
  - A variable is an expression.
  - If x and y are expressions, then $\neg x, x \wedge y$, $x \vee y$ are expressions.
  - Parentheses can be used to alter the normal order of evaluation $(\neg > \wedge > \vee)$.

- Example: $x_1 \vee (x_2 \wedge \neg x_3)$
  - If x_1 and x_3 are $false$ and x_2 is $true$, the value of the expression is $true$

Satisfiability Problem

- **Given a formula :** Is there an assignment to make an expression true?
  - Brute Force: 2^n possible combinations for n variables

$$(x_1 \wedge \neg x_2) \vee (\neg x_1 \wedge x_3) \vee \neg x_3$$



(t, t, t)
(t, t, f)
(t, f, t)
(t, f, f)
(f, t, t)
(f, t, f)
(f, f, t)
(f, f, f)

Perform Formula Evaluation

- To evaluate an expression, we can traverse its tree in postorder.
- To perform evaluation, assume that each node has four fields
  - *leftChild*
  - *first*: operator or value of the variable
  - *Second*: value of the expression of the sub-tree
  - *rightChild*

leftChild	*first*	*second*	*rightChild*

- **enum** *Operator* {*Not, And, Or , True, False*};

First Version of Satisfiability Algorithm

for all 2^n possible truth value combinations for the n variables

{

generate the next combination;

replace the variables by their values;

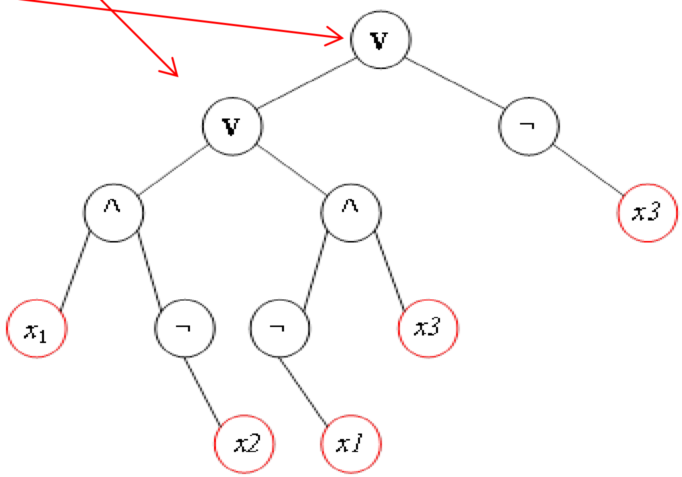
evaluate the formula by traversing the tree it points to in postorder;

if (*fomula.Data*().*second*()) **{ cout** << combination ; **return** ; **}**

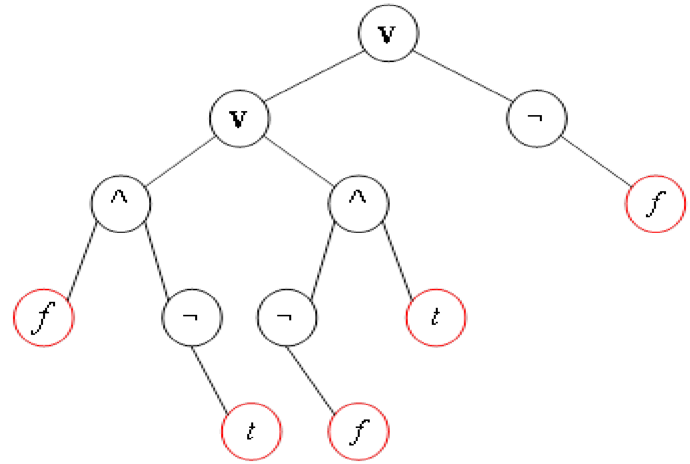
}

cout << "no satisfiable combination";

(t, t, t)
(t, t, f)
(t, f, t)
(t, f, f)
(f, t, t)
(f, t, f)
(f, f, t)
(f, f, f)



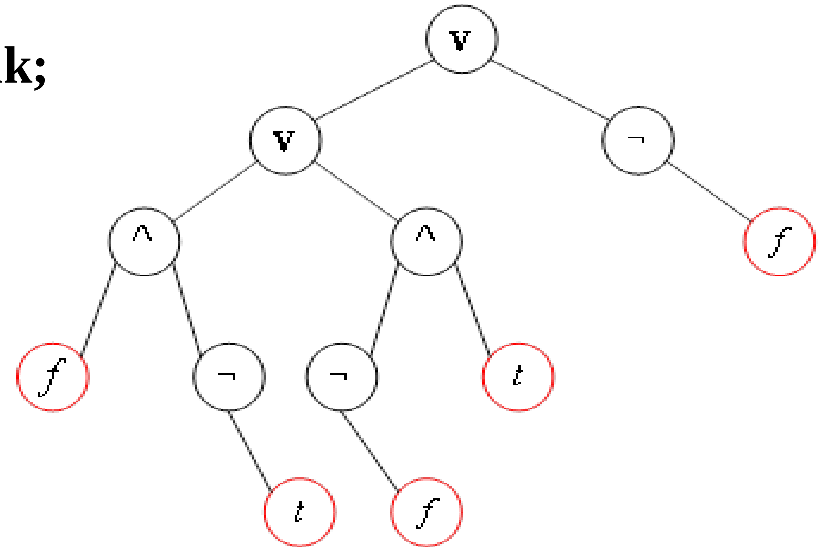
Postorder Traversal




```
template < class T>
void Tree <T>::Postorder()
{// driver
    Postorder(root);
}

template < class T>
void Tree <T>::Postorder(TreeNode<T> *currentNode)
{// workhorse
    if (currentNode) {
        Postorder(currentNode→leftChild);
        Postorder(currentNode→rightChild);
        Visit(currentNode);
    }
}
```

Visit(*currentNode*)

```
switch (p→data.first) {
    case Not: p→data.second = !p→rightChild→data.second; break;
    case And: p→data.second =
            p→leftChild→data.second && p→rightChild→data.second;
            break;
    case Or: p→data.second =
            p→leftChild→data.second || p→rightchild→data.second;
            break;
    case True: p→data.second = true; break;
    case False: p→data.second = false;
}
```



Threaded Binary Trees

- Two many null pointers in current representation of binary trees
  - number of nodes: n
  - number of non-null links: $n - 1$
  - total links: $2n$

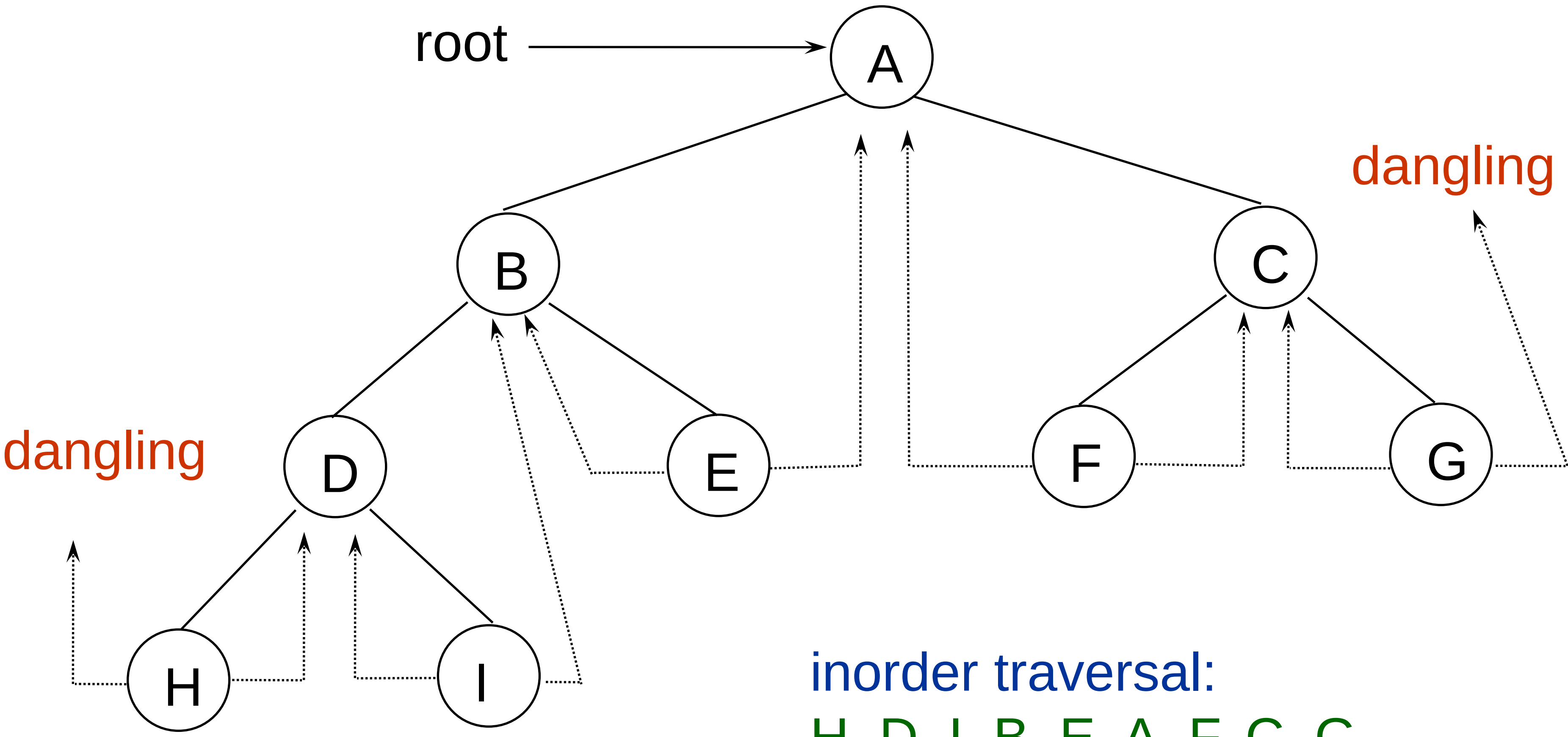
  => null links: $2n - (n - 1) = n + 1$
- For easy of traversal a tree, replace these null pointers with some useful "threads".

Threaded Binary Trees (contd.)

- If $ptr -> left\_child$ is null,
  - replace it with a pointer to the node that would be visited *before* *ptr* in an *inorder traversal*

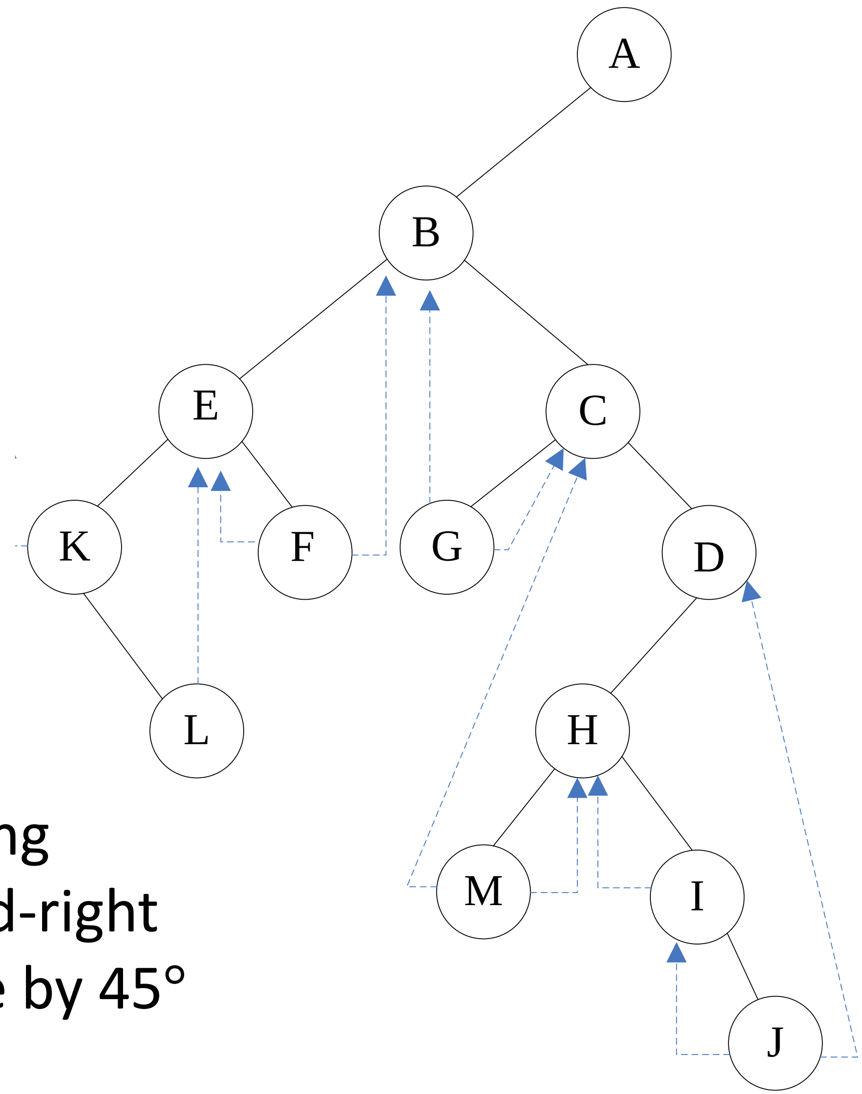
- If $ptr -> right\_child$ is null,
  - replace it with a pointer to the node that would be visited *after* *ptr* in an *inorder traversal*

Example: Threaded Binary Tree



inorder traversal:
H, D, I, B, E, A, F, C, G

Degree Two Tree Representation



Binary Tree!

Rotate the right-sibling pointers in a left child-right sibling tree clockwise by 45°

Threads

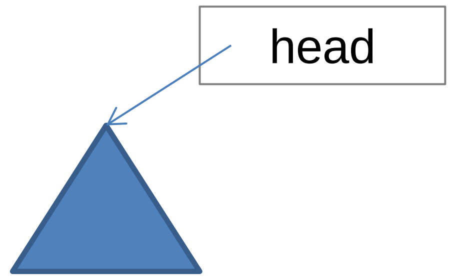
- To distinguish between normal pointers and threads, two boolean fields, $LeftThread$ and $RightThread$, are added to the record in memory representation.
  - $t -> LeftThread\ =\ TRUE$

    => $t -> LeftChild$ is a **thread**
  - $t -> LeftThread\ =\ FALSE$

    => $t -> LeftChild$ is a **pointer** to the left child.

Threads (Cont.)

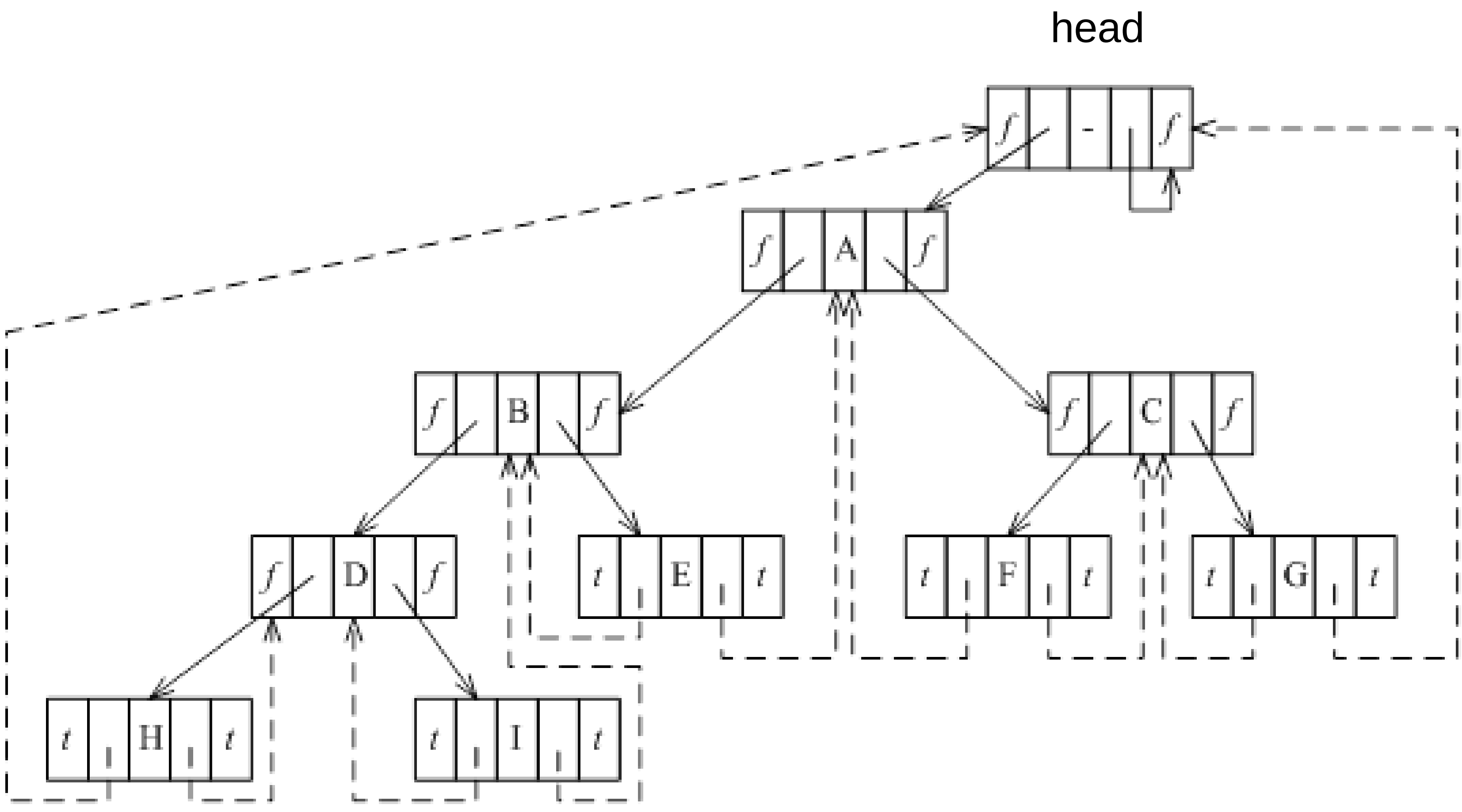
- To avoid dangling threads, a head node is used in representing a binary tree.
- The original tree becomes the left subtree of the head node.



- Empty Binary Tree

leftThread	*leftChild*	*data*	*rightChild*	*rightTread*
true				**false**

Memory Representation of Threaded Tree



f = **false;** *t* = **true**

Find the inorder Successor

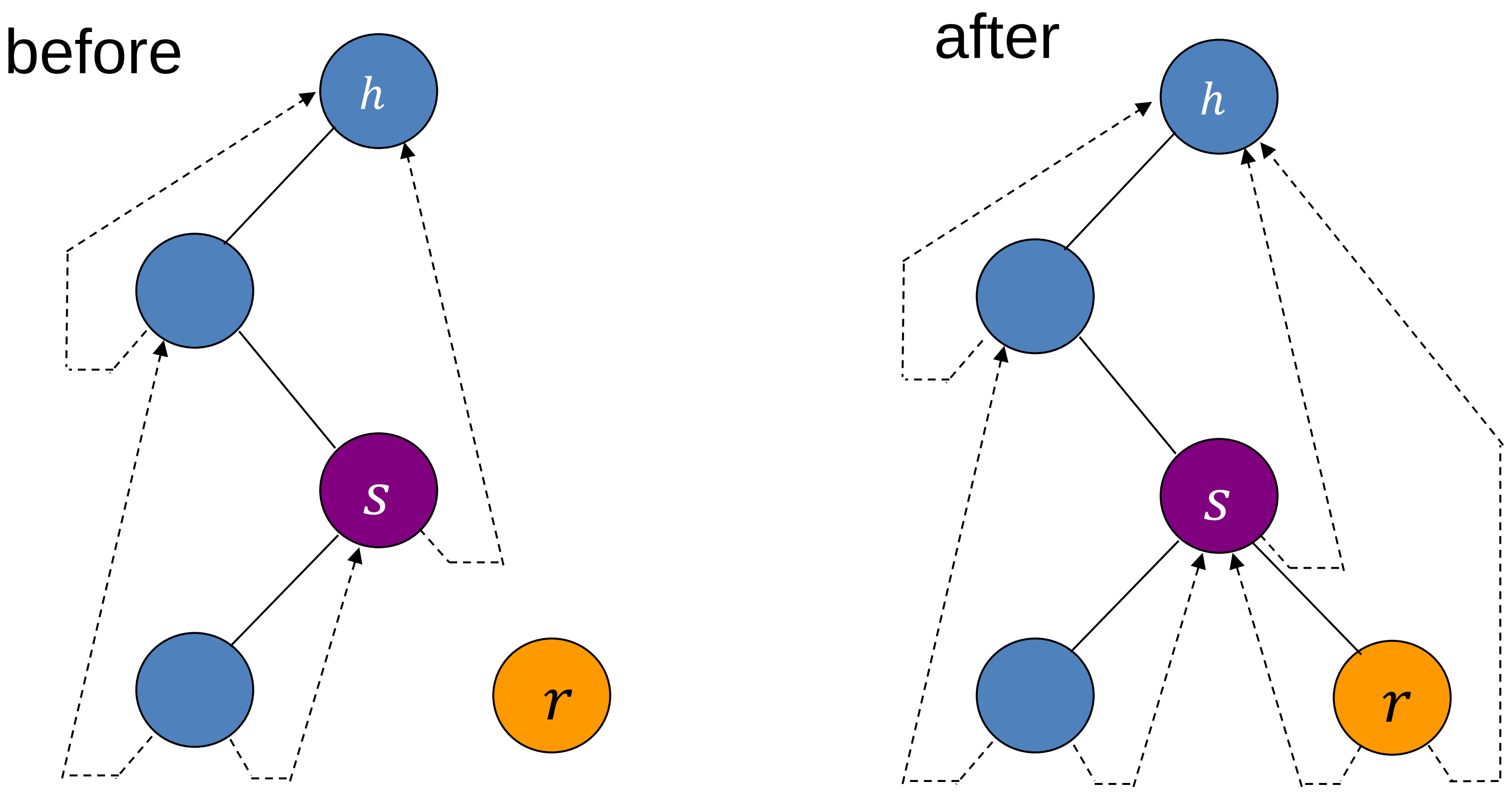
T* $ThreadedInorderIterator$**::**$Next$ ()
{// return the iorder successor of *currentNode* in a threded binary tre

$ThreadedNode$ <T> *$temp = currentNode \rightarrow rightChild$**;**
if (!$currentNode \rightarrow rightThread$)
while (!$temp \rightarrow leftThread$) $temp = temp \rightarrow leftChild$**;**
$currentNode = temp$**;**
if ($currentNode == root$) **return** 0**;**
else return &$currentNode \rightarrow data$**;**
}

Inorder traversal can be performed without stack

Inserting a Node to a Threaded Binary Tree

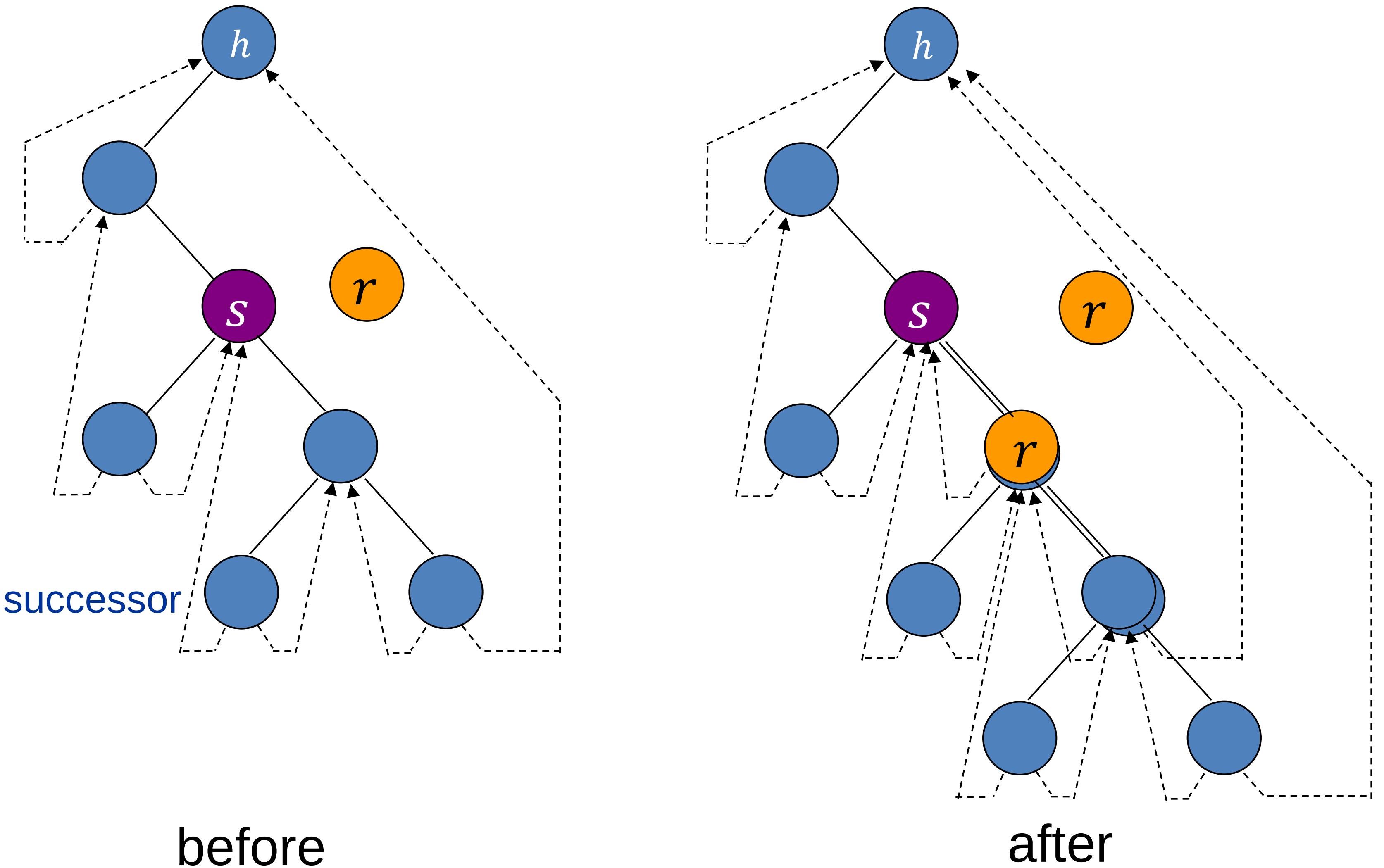
- Inserting a node r as the right child of a node s.
  - If s has an empty right subtree, then the insertion is simple



Inserting a Node to a Threaded Binary Tree

- Inserting a node r as the right child of a node s.
  - If the right subtree of s is not empty, the this right subtree is made the right subtree of r after insertion.
    - When this is done, r becomes the inorder predecessor of a node that has a $LeftThread == TRUE$ field, and consequently there is an thread which has to be updated to point to r.
    - The node containing this thread was previously the inorder successor of s.

Insertion of ***r*** As A Right Child of ***s*** in A Threaded Binary Tree (Cont'd)



Insertion of *r* As A Right Child of *s*

```
template <class T>
void ThreadedTree <T>::InsertRight (ThreadedNode <T> *s,
                                    ThreadedNode <T> *r)
{// insert r as the right son of s
      r → rightChild = s → rightChild;
      r → rightThread = s → rightThread;
r     r → leftChild = s;
      r → leftThread = True; // leftChild is a thread
      s → rightChild = r;
s     s → rightThread = false;
      if (! r → rightThread) {
            ThreadedNode <T> *temp = InorderSucc (r);
successor                            // return the inorder successor of r
            temp → leftChild = r;
      }
}
```

Priority Queues

- In a priority queue, the element to be deleted is the one with highest (or lowest) priority.
- An element with arbitrary priority can be inserted into the queue according to its priority.
- A data structure supports the above two operations is called max (min) priority queue.
- Example
  - Selling machine service
    - Amount of time (min heap): fixed amount per use with different using time
    - Amount of payment (max heap): different amount for per service with the same using time

A Max Priority Queue ADT

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

Operators such as $<, >, ==, =$ are defined with class *T*

Compared with Other Data Structures

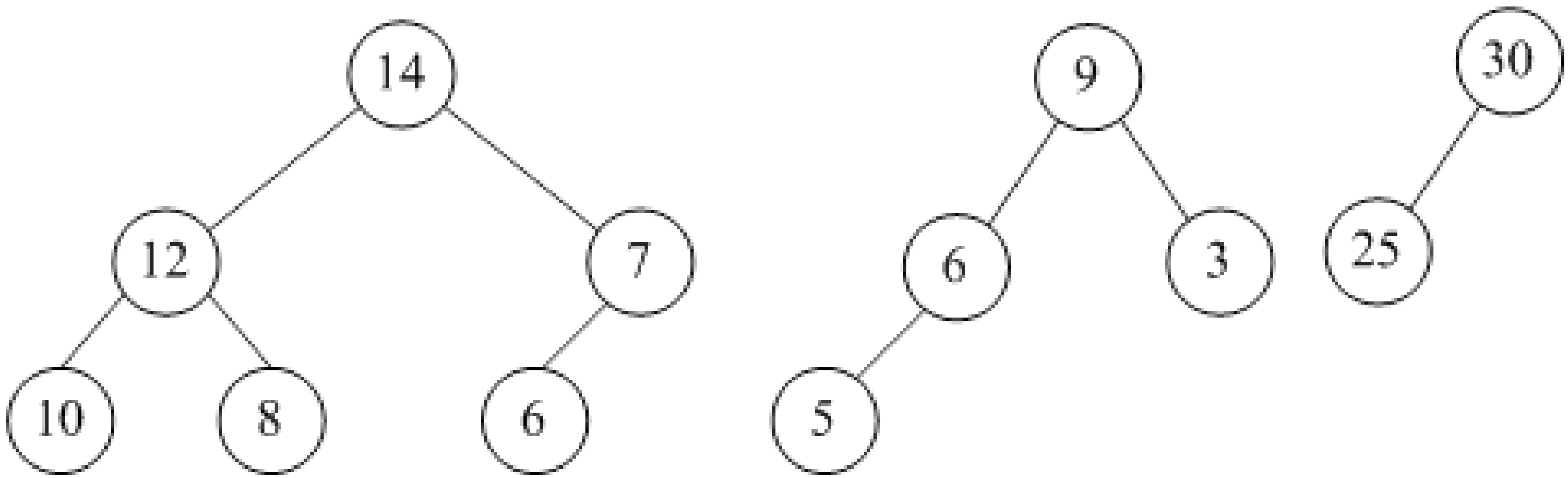
- Unordered linked list
- Unordered array
- Sorted linked list
- Sorted array
- Heap

Representation	Insertion	Deletion
Unordered array	$\Theta(1)$	$\Theta(n)$
Unordered linked list	$\Theta(1)$	$\Theta(n)$
Sorted array	$O(n)$	$\Theta(1)$
Sorted list	$O(n)$	$\Theta(1)$
Max heap	$O(\log_2 n)$	$O(\log_2 n)$

Max (Min) Heap

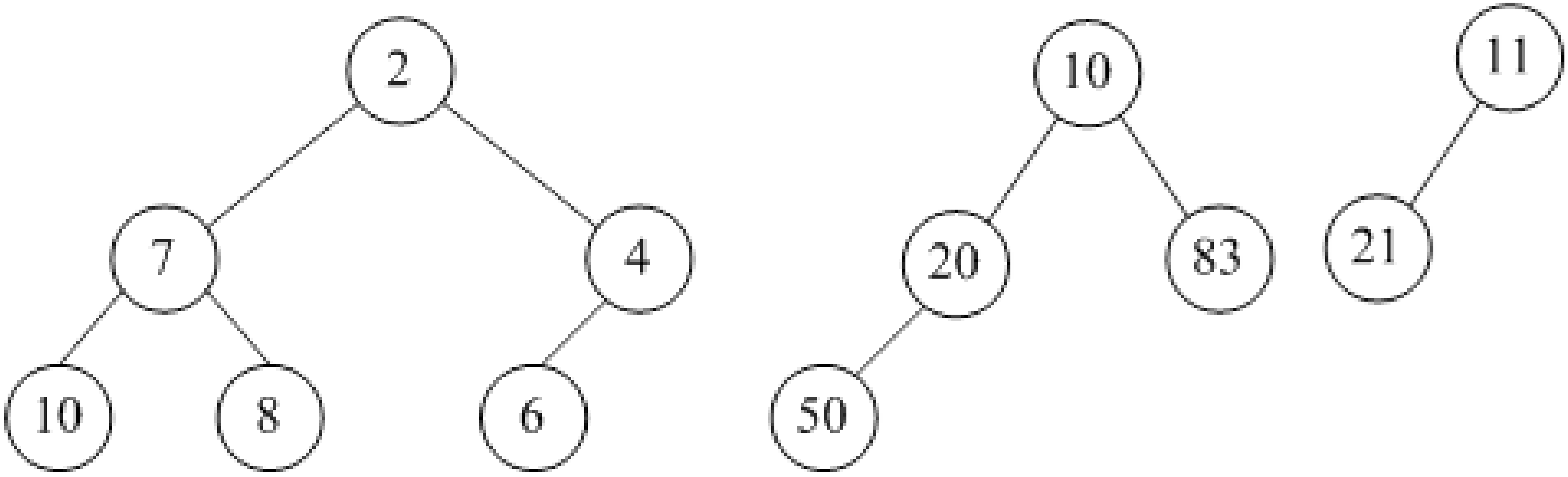
- Heaps are frequently used to implement priority queues. The complexity is $\boldsymbol{O}(\log \boldsymbol{n})$.
- **Definition**
  - A max (min) tree is a tree in which the key value in each node is no smaller (larger) than the key values in its children (if any).
  - A max heap is a complete binary tree that is also a max tree.
  - A min heap is a complete binary tree that is also a min tree.

Example: Max Heap



Property: The root of max heap contains the largest.

Example: Min Heap



Property: The root of min heap contains the smallest.

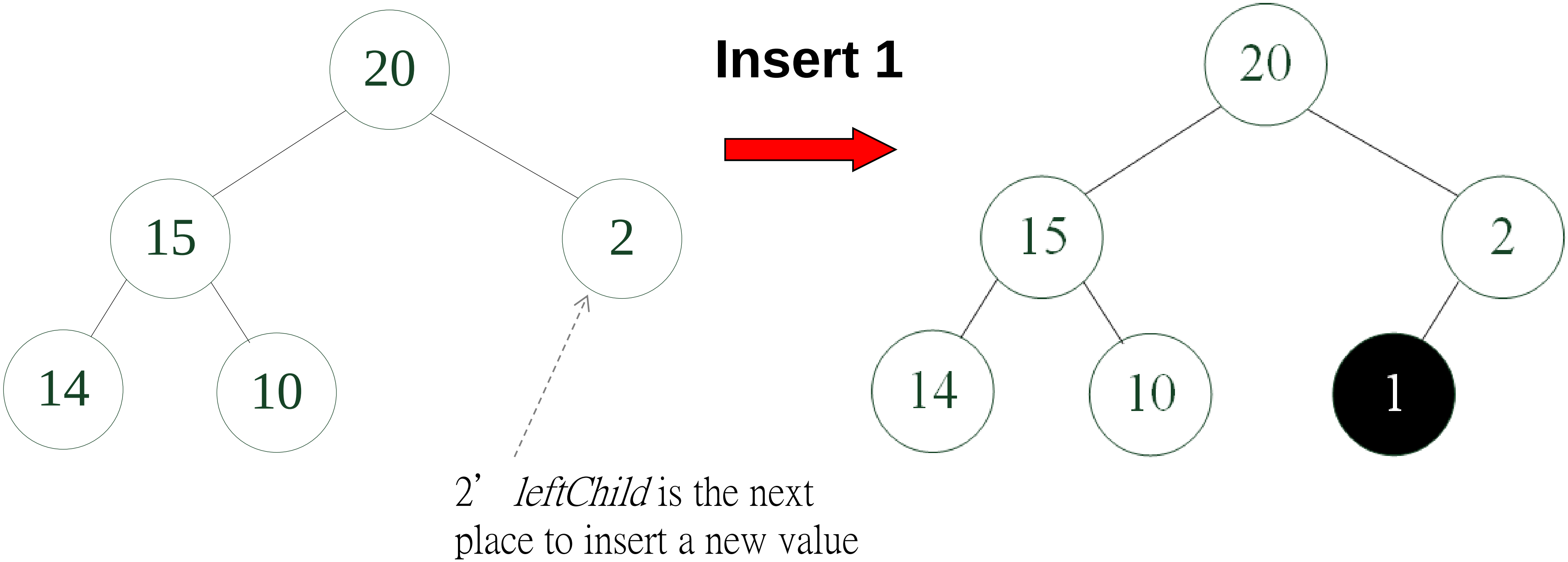
ADT of MaxHeap

```
template <class T>
class MaxPQ {
public:
      virtual ~MaxPQ () {}
      virtual bool IsEmpty () const = 0;
      virtual const T& Top () const = 0;
      virtual void Push(const T&) = 0;
      virtual void Pop () = 0;
private:
     T *heap;          //element array
     int heapSize;     //no. of element in heap
     int capacity;     //size of the array heap

};

template <class T>
MaxHeap<T>::MaxHeap (int theCapacity = 10)
{
      if ( theCapacity < 1 ) throw "Capacity must be >= 1.";
      capacity = theCapacity;
      heapSize = 0;
      heap = new T [capacity + 1]; // heap [0] is not used
}
```

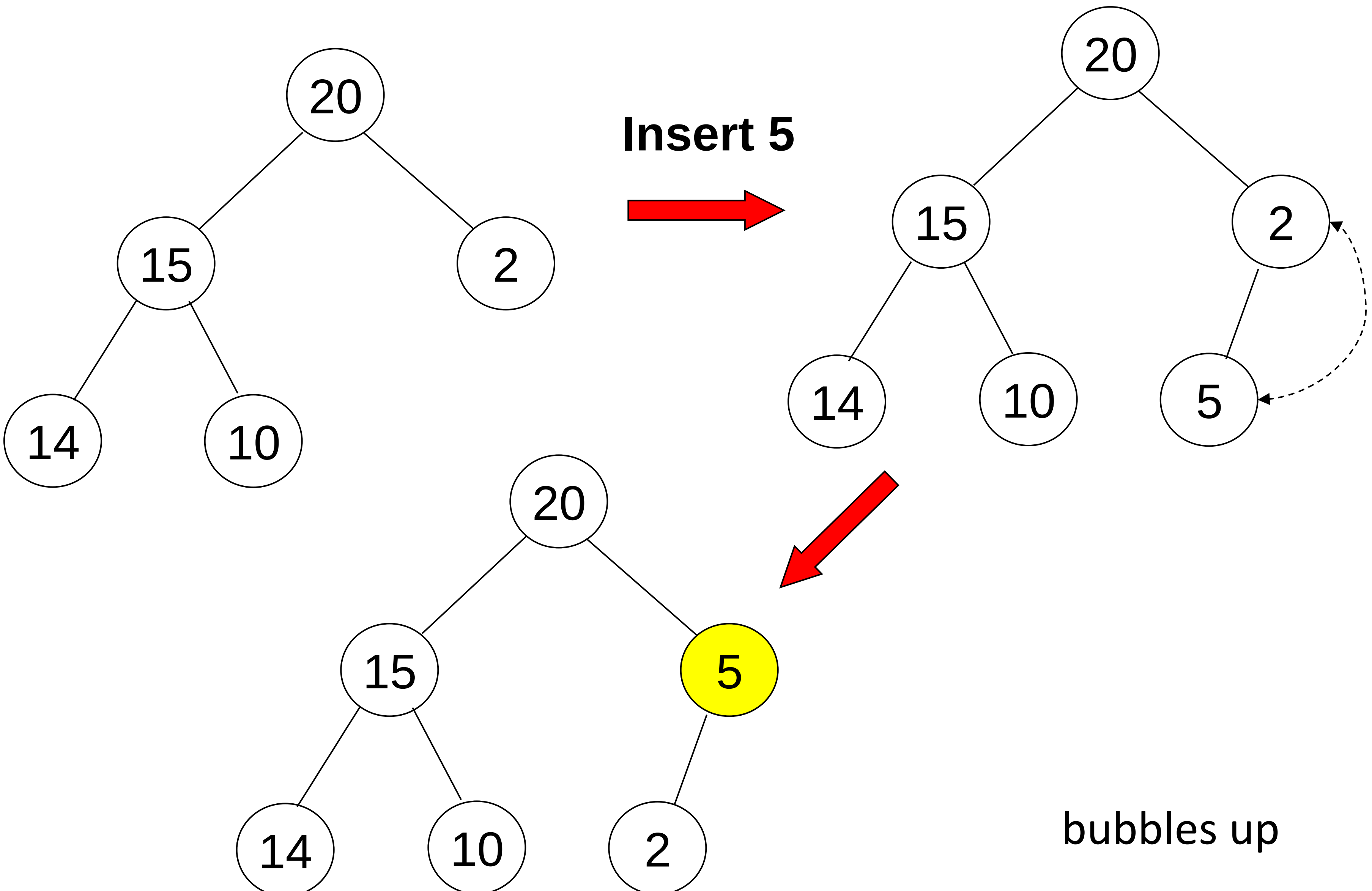
Implement MaxHeap by using an array heap

Insertion Into a Max Heap

- Insertion begins at a leaf of a complete binary tree and bubbles up toward the root to find a correct place

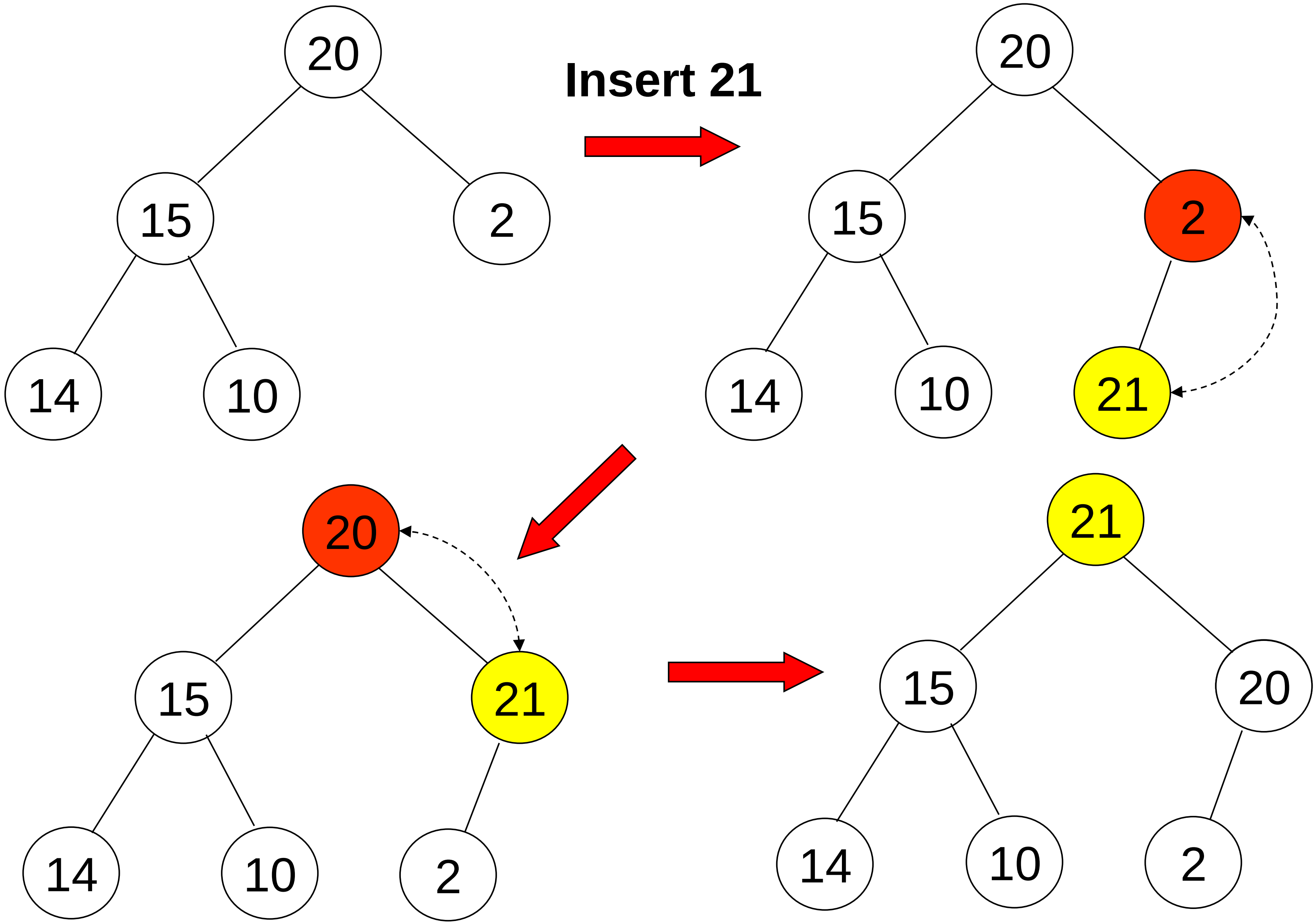


Insertion into a Max Heap (cont'd)



bubbles up

Insertion into a Max Heap (cont'd)



Insertion

$O(\log(n))$

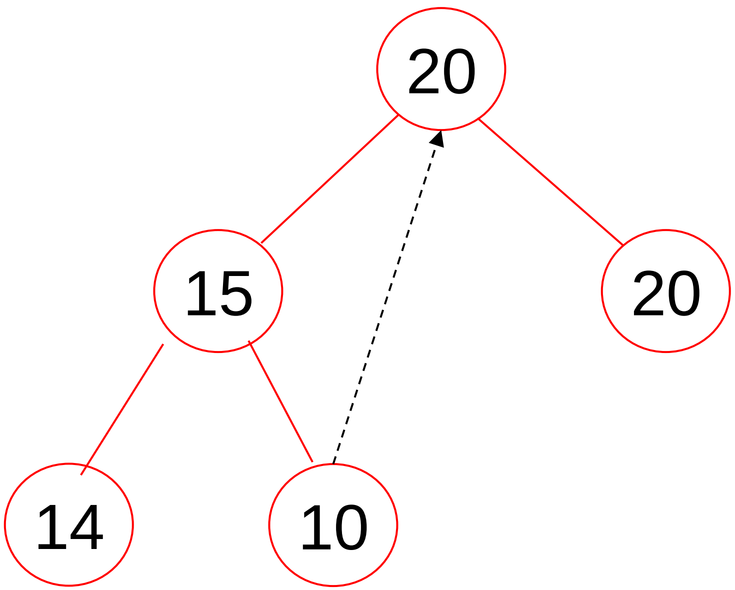
```
Template <class T>
void MaxHeap<T>::Push(const T& e)
{ // inset e to maxHeap
    if (heapSize == capacity) { // double the space
        ChangeSize 1D(heap, capacity, 2*capacity);
        capacity *= 2;
    }
    int currentNode = + +heapSize;
    while (currentNode != 1 && heap[currentNode / 2] < e)
    { // bubble up
        heap[currentNode] = heap[currentNode / 2]; // move pare down
        currentNode /= 2;
    }
    heap[currentNode] = e;
}
```

Index to the next empty location

Index to the parent node

Deletion from a Max Heap (Cont.)

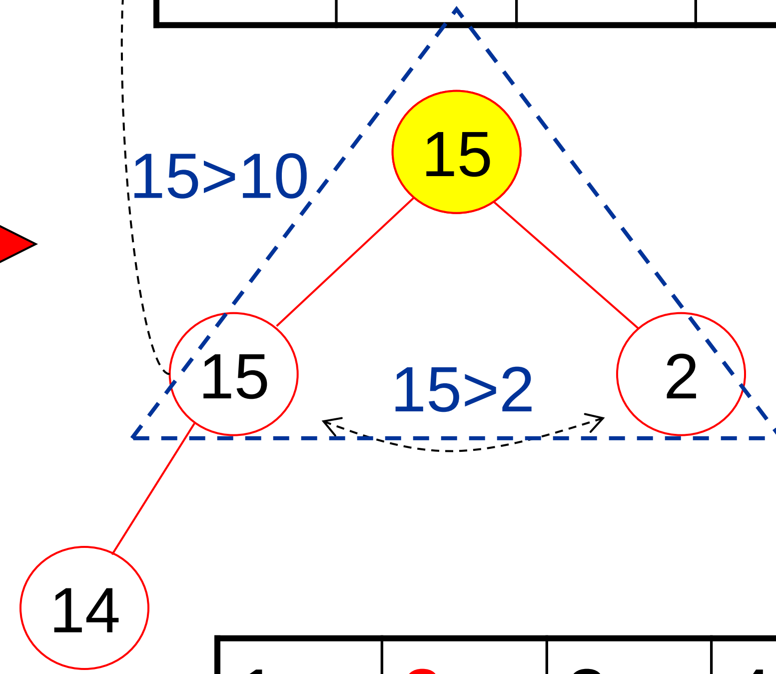
1	2	3	4	5
20	15	2	14	10



delete

lastE =10

1	2	3	4
20	15	2	14

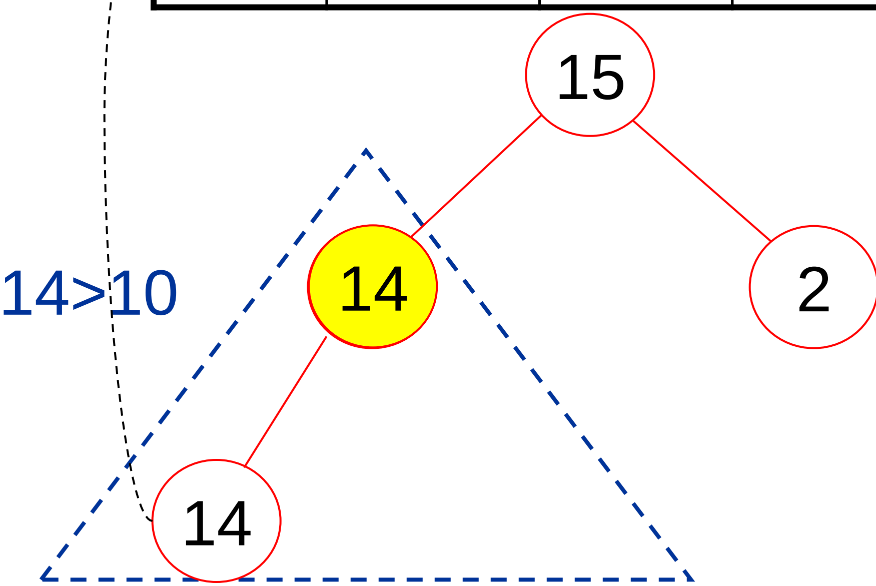


1	2	3	4
15	15	2	14

Deletion from a Max Heap (Cont.)

lastE =10

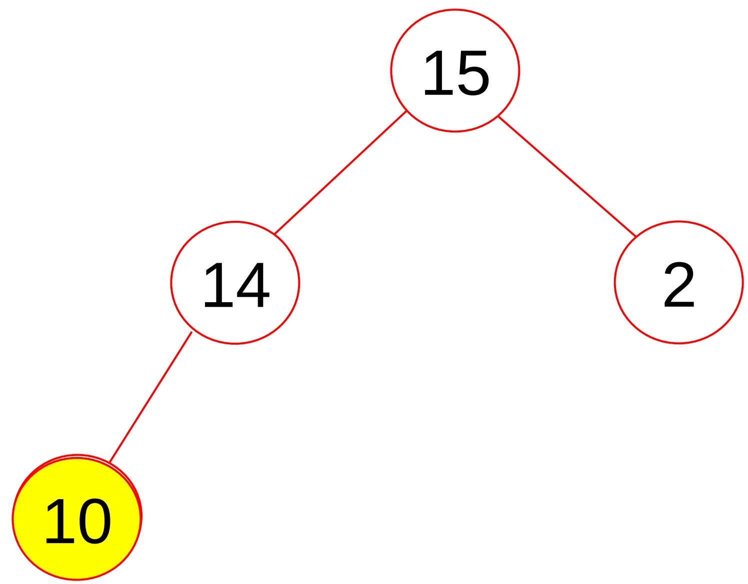
1	2	3	4
15	15	2	14



1	2	3	4
15	14	2	14

lastE =10

1	2	3	4
15	14	2	14



1	2	3	4
15	14	2	10

Deletion from a Max Heap

```
Template <class T>
void MaxHeap<T>::Pop()
{ // delete the max element
    if (IsEmpty ()) throw "Heap is empty. Cannot delete.";
    heap[1]. ~T(); // delete the max element
    // removelast 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;
}
```

Complexity of Heap

- Heap
  - a min (max) element is deleted. $O(\log_2 n)$
  - deletion of an arbitrary element $O(n)$
  - search for an arbitrary element $O(n)$
-

Binary Search Tree

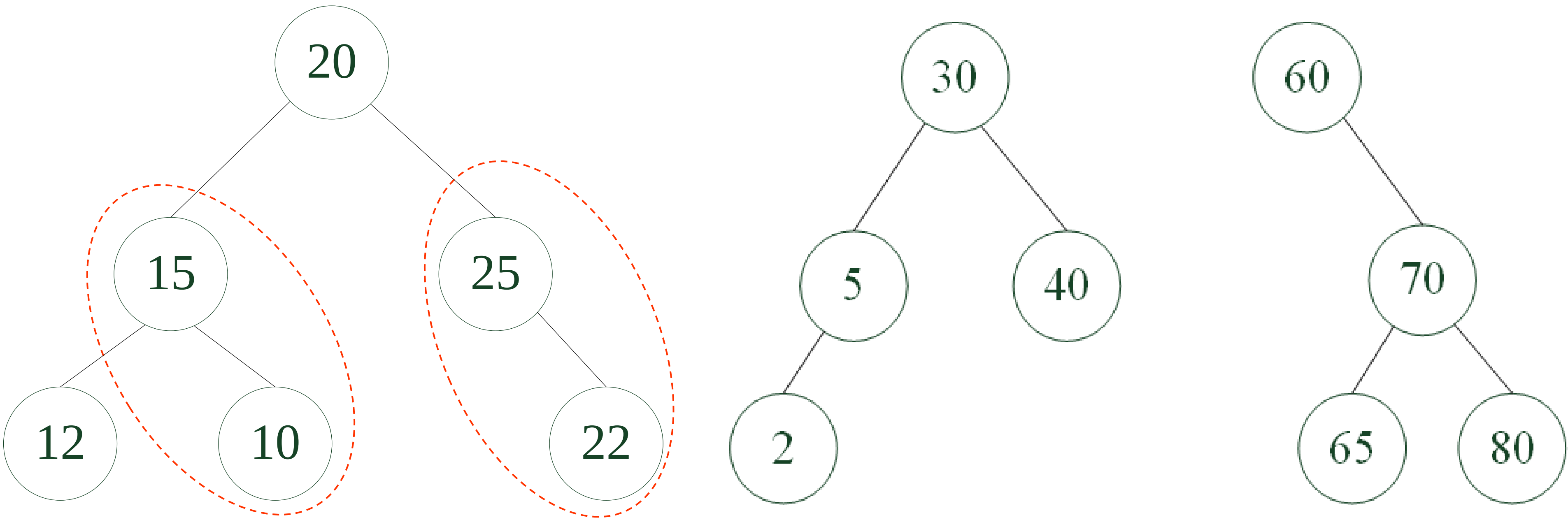
- Dictionary
  - A collection of pairs, each has a key and an element
  - Assume no two pair has the same key

```
template <class K, class E>
class Dictionary {
public:
    virtual bool IsEmptay() const = 0;
        // return true iff the dictionary is empty
    virtual pair <K, E>* Get(const K&) const = 0;
        // return pointer to the pair with specified key; return 0 if no such pair
    virtual void Insert(const pair <K, E>&) = 0;
        // insert the given pair; if key is a duplicate, update associated element
    virtual void Delete(const K&) = 0;
        // delete the pair with specified key
};
```

Binary Search Tree

- Binary search tree: A binary search tree is a binary tree. It may be empty. If it is not empty then it satisfies the following properties:
  - Every element has a unique key.
  - The keys in a nonempty left subtree are smaller than the key in the root.
  - The keys in a nonempty right subtree are larger than the key in the root.
  - The left and right subtrees are also binary search trees.

Binary Trees



Not binary search tree

Binary search trees

Searching a Binary Search Tree

- If the root is null, then this is an empty tree. No search is needed.
- If the root is not null, compare the k with the key of root.
  - If k is equal to the key of the root, then it's done.
  - If k is less than the key of the root, then no elements in the right subtree have key value k. We only need to search the left tree.
  - If k is larger than the key of the root, only the right subtree is to be searched.

Recursive Search

```
template <class K, class E> // driver
pair<K, E>* BST<K, E> :: Get(const K& k)
{ // search the binary tree (*this) for a pair with key k
  // if found, return a pointer to the pair; otherwise, return 0.
        return Get(root, k);
}

template <class K, class E> // workhose
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;
}
```

Iterative Search

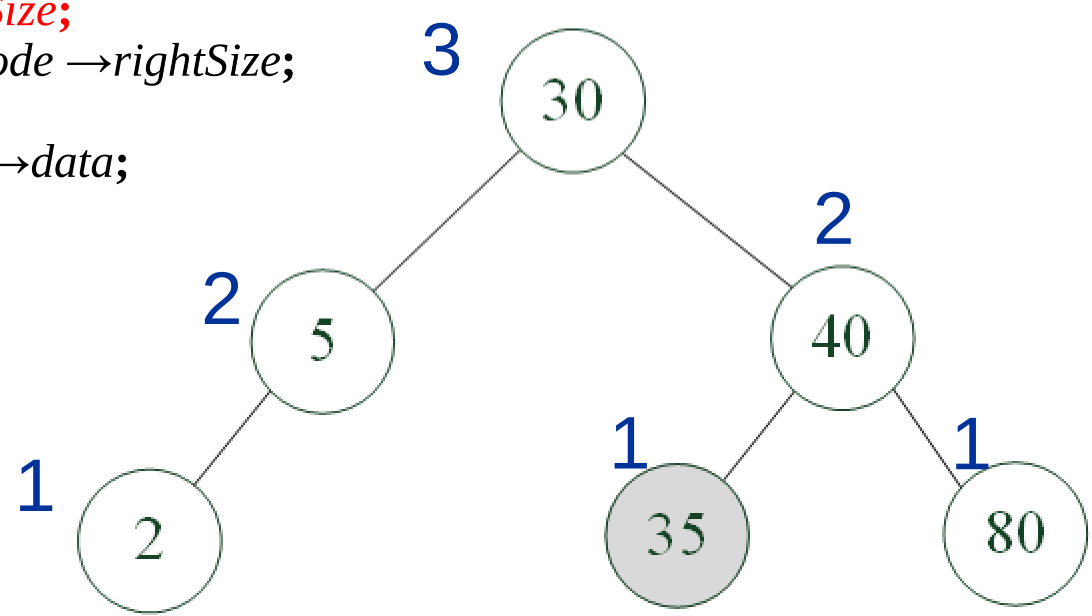
```
template <class K, class E> // Iterative Search
pair<K, E>* BST<K, E> :: Get(const K& k)
{
    TreeNode < pair<K, E> > *currentNode = root;
    while (currentNode) {
        if (k < currentNode →data.first)
            currentNode = currentNode →leftChild;
        else if (k > currentNode →data.first)
            currentNode = currentNode →rightChild;
        else return & currentNode →data;
    }

// no matching pair
return 0;
}
```

Search Binary Search Tree by Rank

- $Rank$ of a node is its position in inorder
- To search a binary search tree by the ranks of the elements in the tree, we need an additional field $LeftSize$.
- $LeftSize$ is **the number of the elements** in the left subtree of a node plus one.
- It is obvious that a binary search tree of height h can be searched by key as well as by rank in $O(h)$ time.
  - What is the range of h?

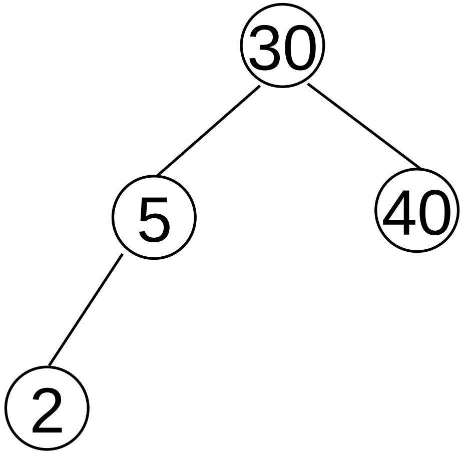
Searching a Binary Search Tree by Rank

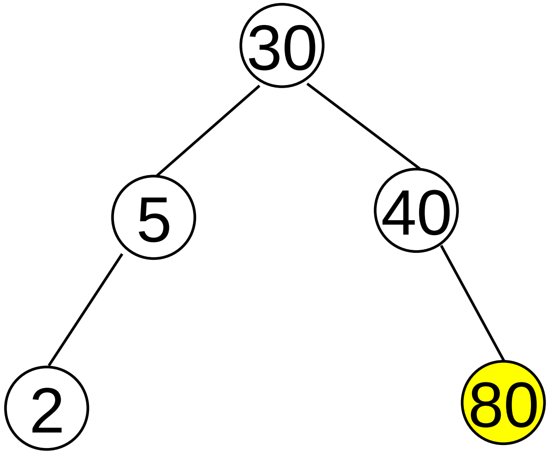
```
template <class K, class E> // search by rank
pair<K, E>* BST<K, E> :: RankGet(int r)
{ // search the binary search tree for the rth smallest pair
    TreeNode < pair<K, E> > *currentNode = root;
    while (currentNode) {
        if (r < currentNode →leftSize) currentNode = currentNode →leftChild;
        else if (r > currentNode →leftSize)
        {
            r −= currentNode →leftSize;
            currentNode = currentNode →rightSize;
        }
        else return & currentNode →data;
    }
return 0;
}
```



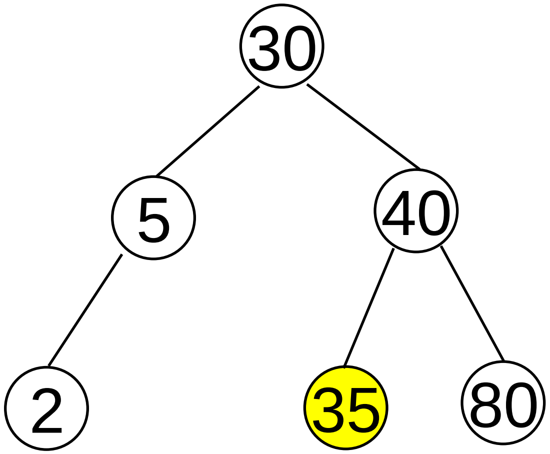
Inserting a Node into a Binary Search Tree

- First, searching with the key
- The searching terminates unsuccessfully => insert the new pair as the right child or right child
- A node if found => simply update the element





Insert 80



Insert 35

BST Insertion

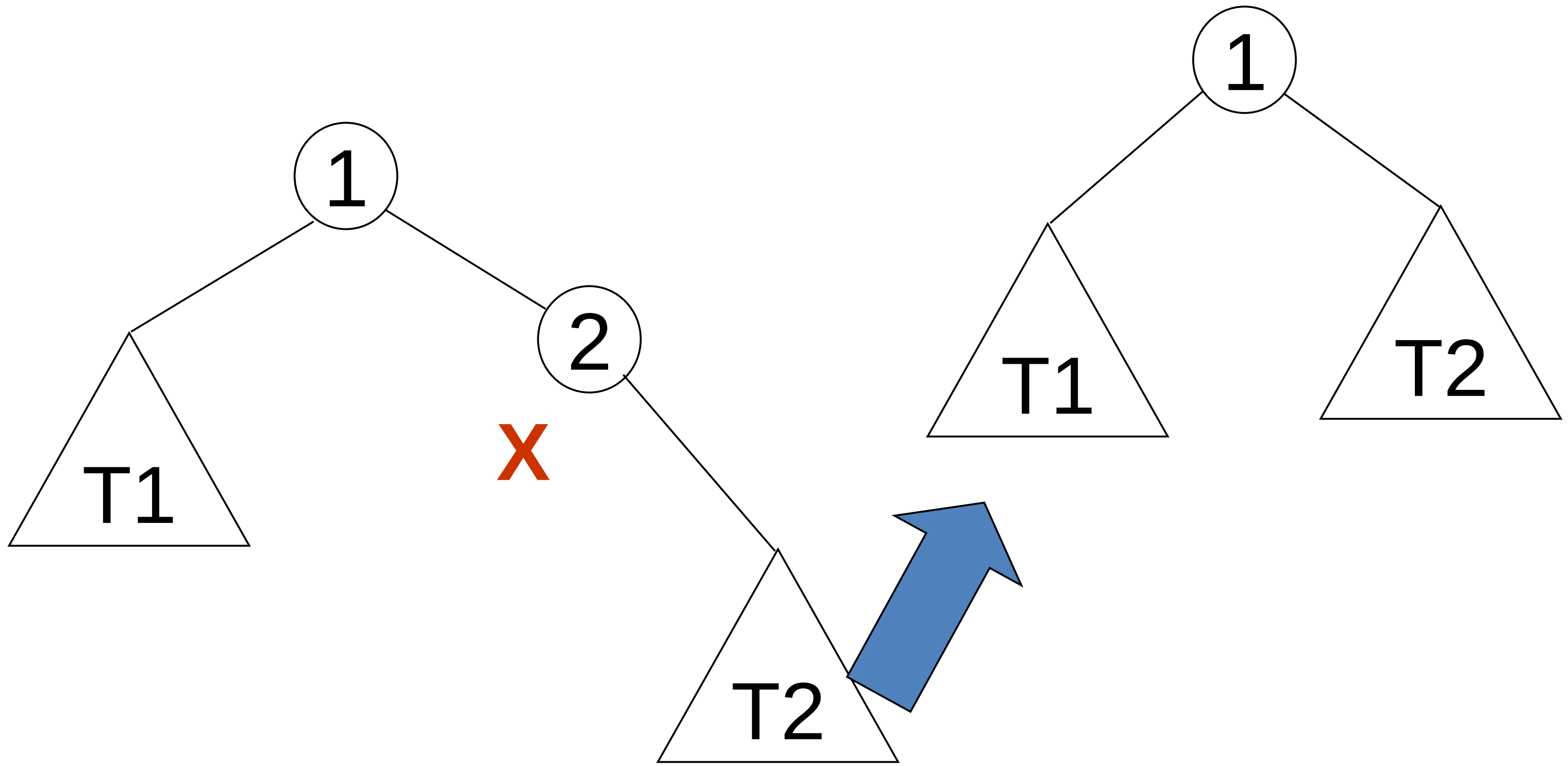
search

insert

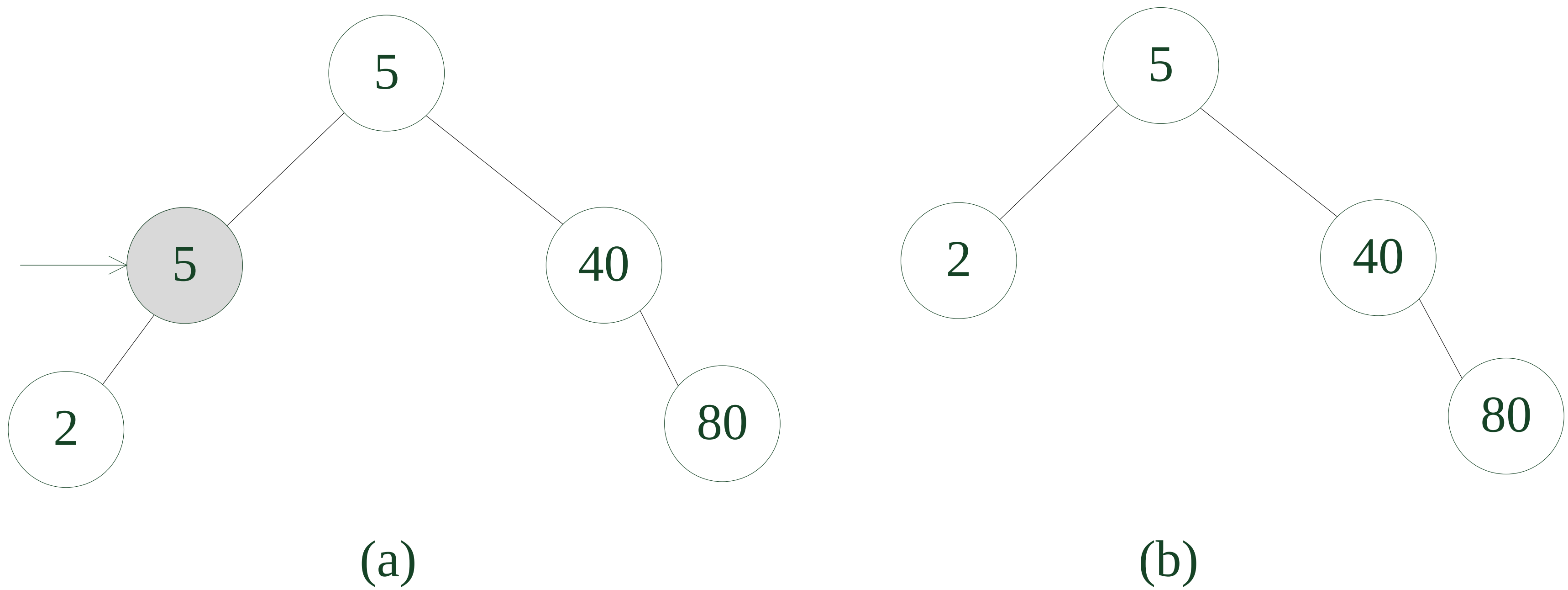
```
template <class K, class E>
void BST<K, E > :: Insert(const pair<K, E >& thePair)
{ // insert thePair into the binary search tree
    // search thePair.first，pp is the parent of   p
    TreeNode < pair<K, E> > *p = root, *pp = 0;
    while (p) {
        pp = p;
        if (thePair.first < p →data.first) p = p →leftChild;
        else if (thePair.first > p →data.first) p = p →rightChild;
        else // dupliacate, update the element
            { p →data.second = thepair.second; return;}
    }
// perform insertion
p = new TreeNode< 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;
}
```

Deletion form a Binary Search Tree

- Leaf Node: directly set the left-child to 0 and dispose it
- Nonleaf Node with a single child: replace it by the child and dispose it

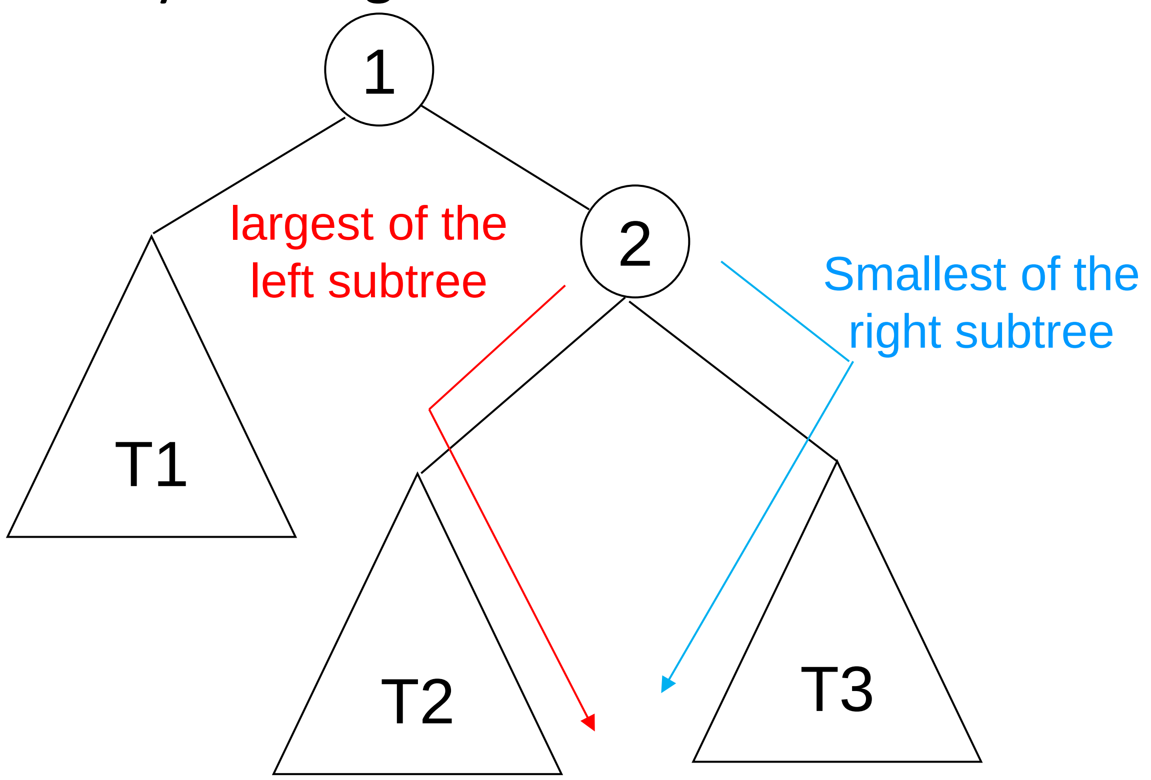


Deletion form a Binary Search Tree

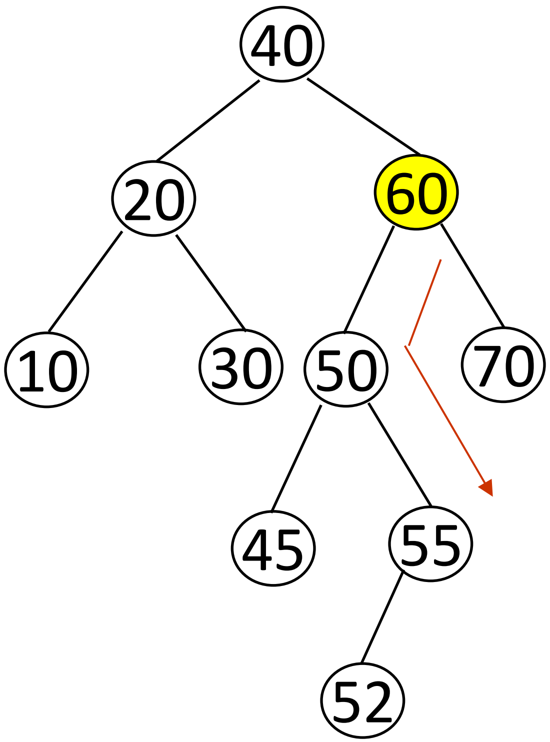


Deletion form a Binary Search Tree

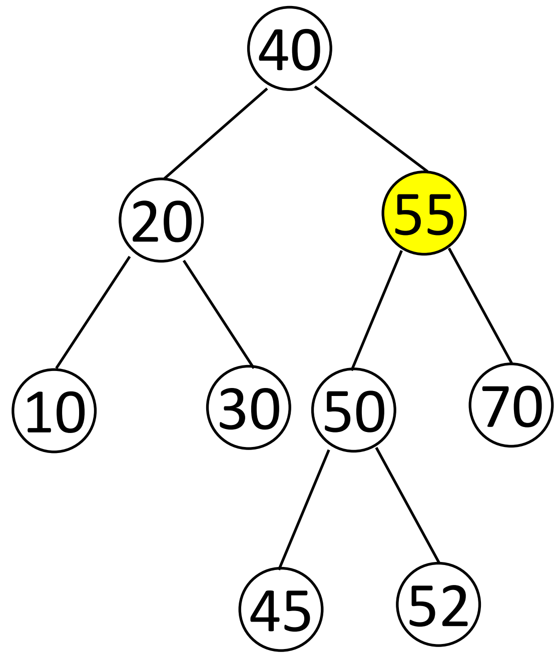
- Nonleaf Node with two child
  - Replace it by the smallest of the right subtree
  - Replace it by the largest of the left subtree



Deletion form a Binary Search Tree



Before deleting 60



After deleting 60

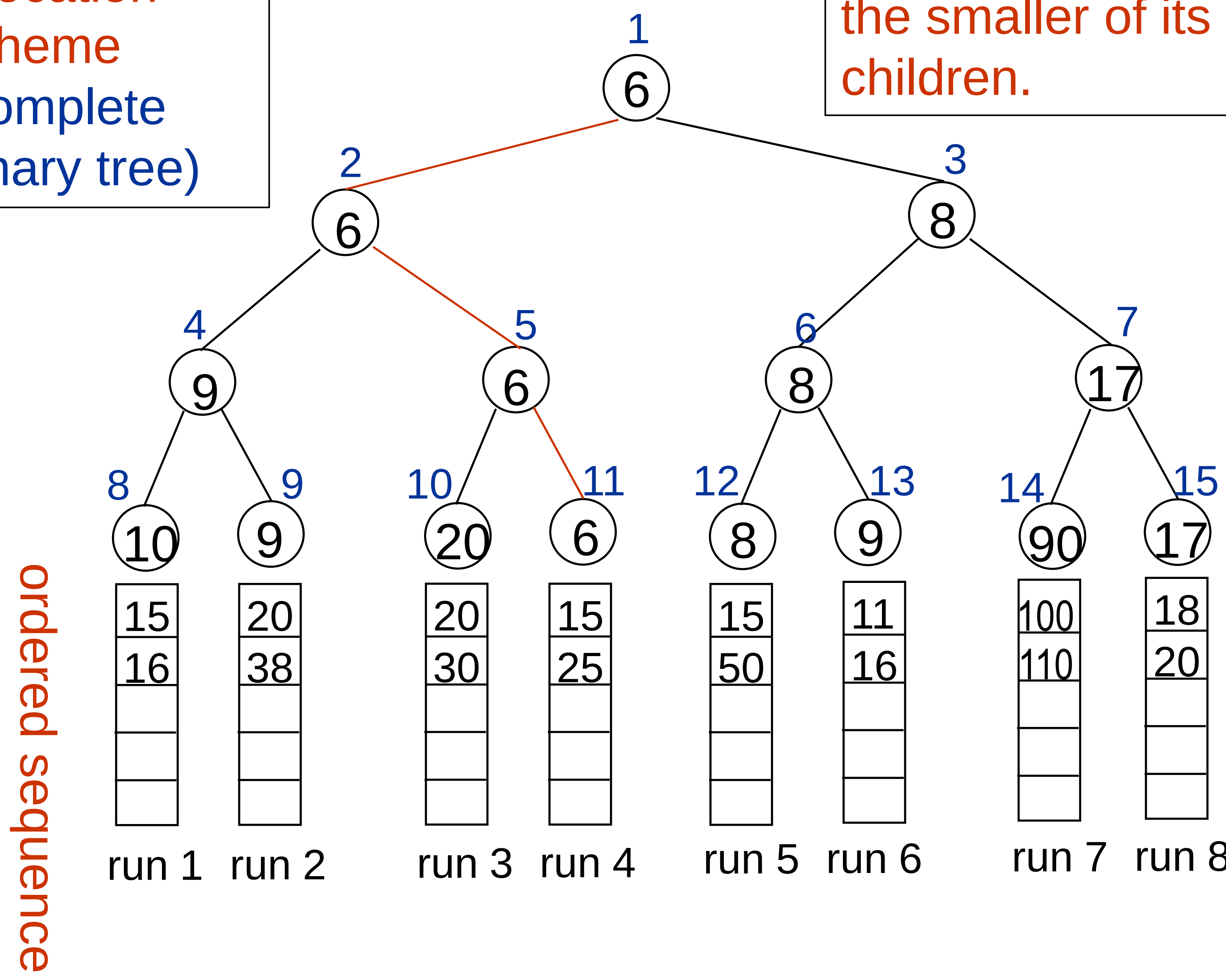
Selection Trees

- To merge k ordered sequences (runs)
  - $K-1$ comparisons are required to determine the next record to output
- To reduce comparisons
  - Winner tree
  - Loser tree

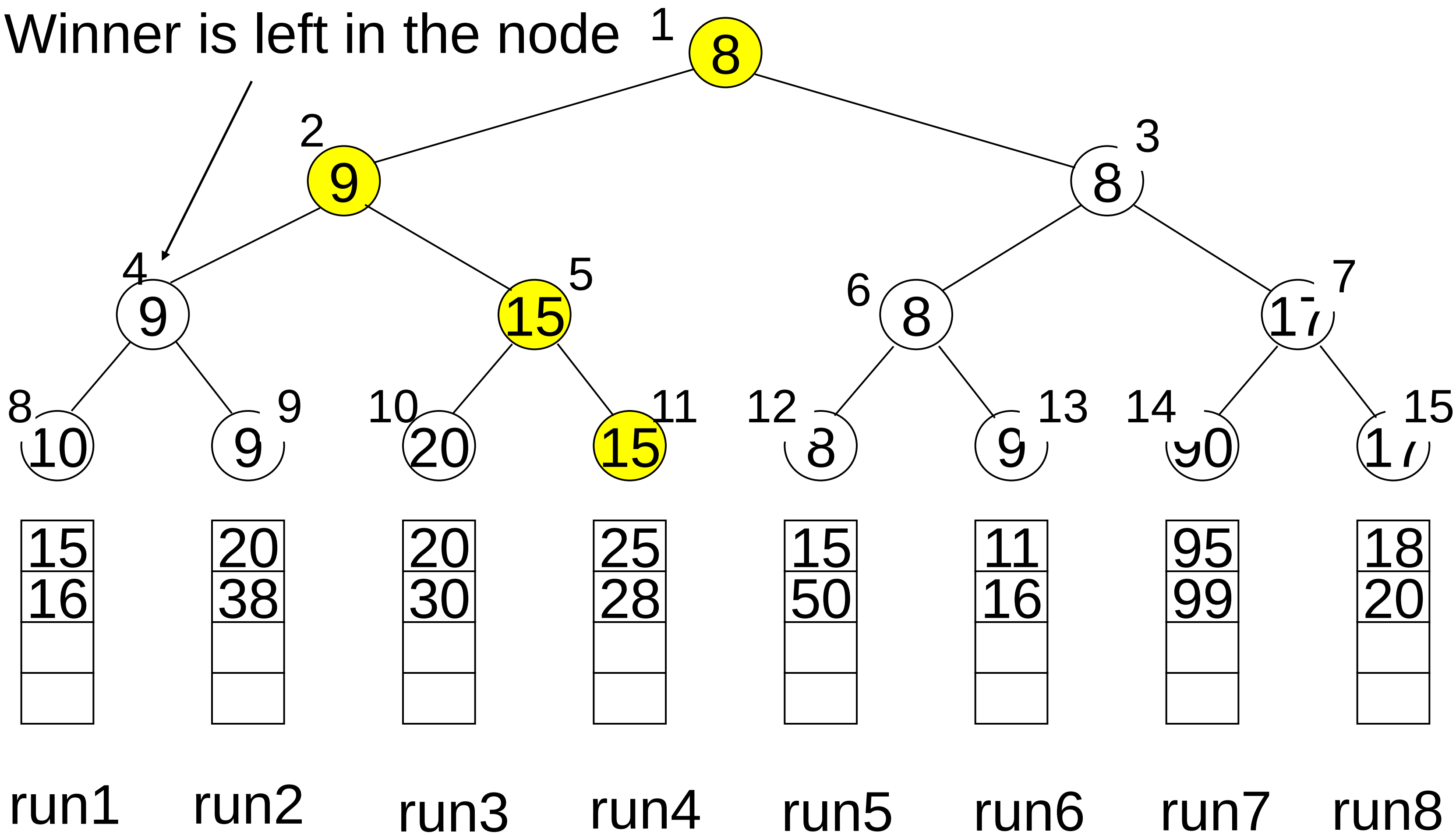
Winner Tree

sequential allocation scheme (complete binary tree)

Each node represents the smaller of its two children.



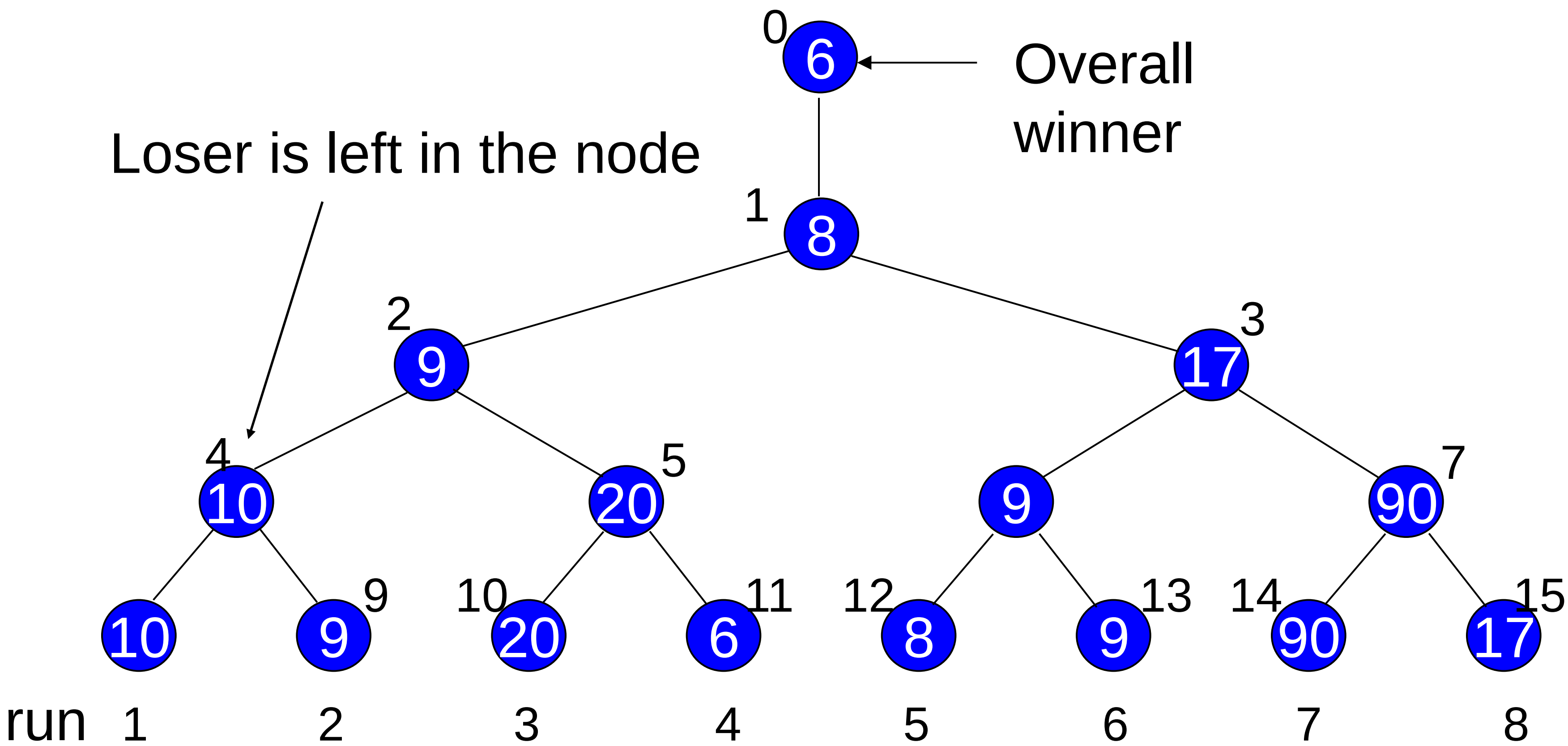
Winner Tree for k = 8



Analysis

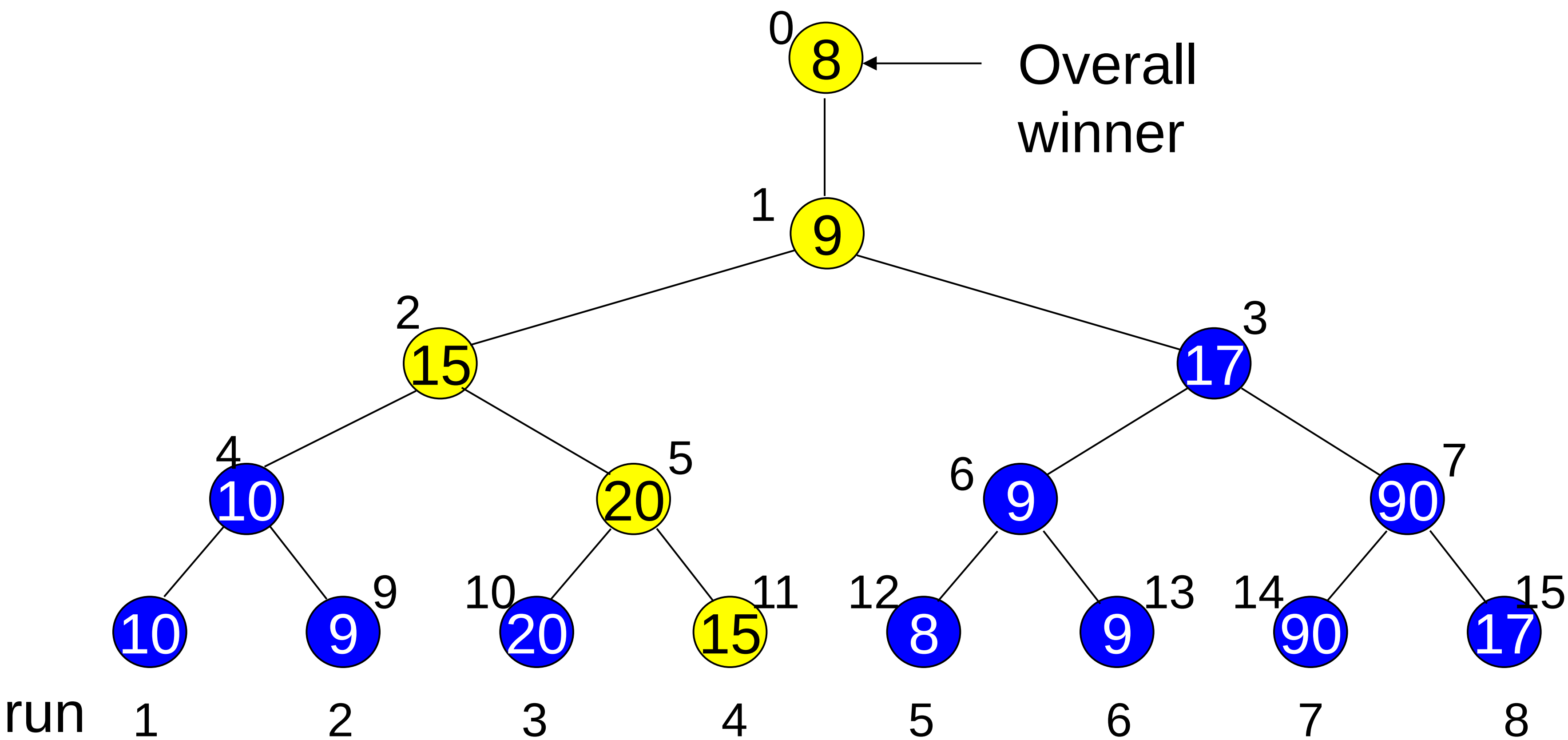
- K: # of runs
- n: # of records
- setup time: $O(K)$ (K-1)
- restructure time: $O(\log_2 K)$ $\lceil \log_2(K+1) \rceil$
- merge time: $O(nlog_2K)$
- **slight** modification: Loser Tree
  - consider the parent node only (vs. sibling nodes)

Loser Tree



Overall winner
Loser is left in the node
0
6
1
8
2
9
3
17
4
10
5
20
9
7
90
10
9
20
6
8
9
90
17
9
10
11
12
13
14
15
run 1
2
3
4
5
6
7
8

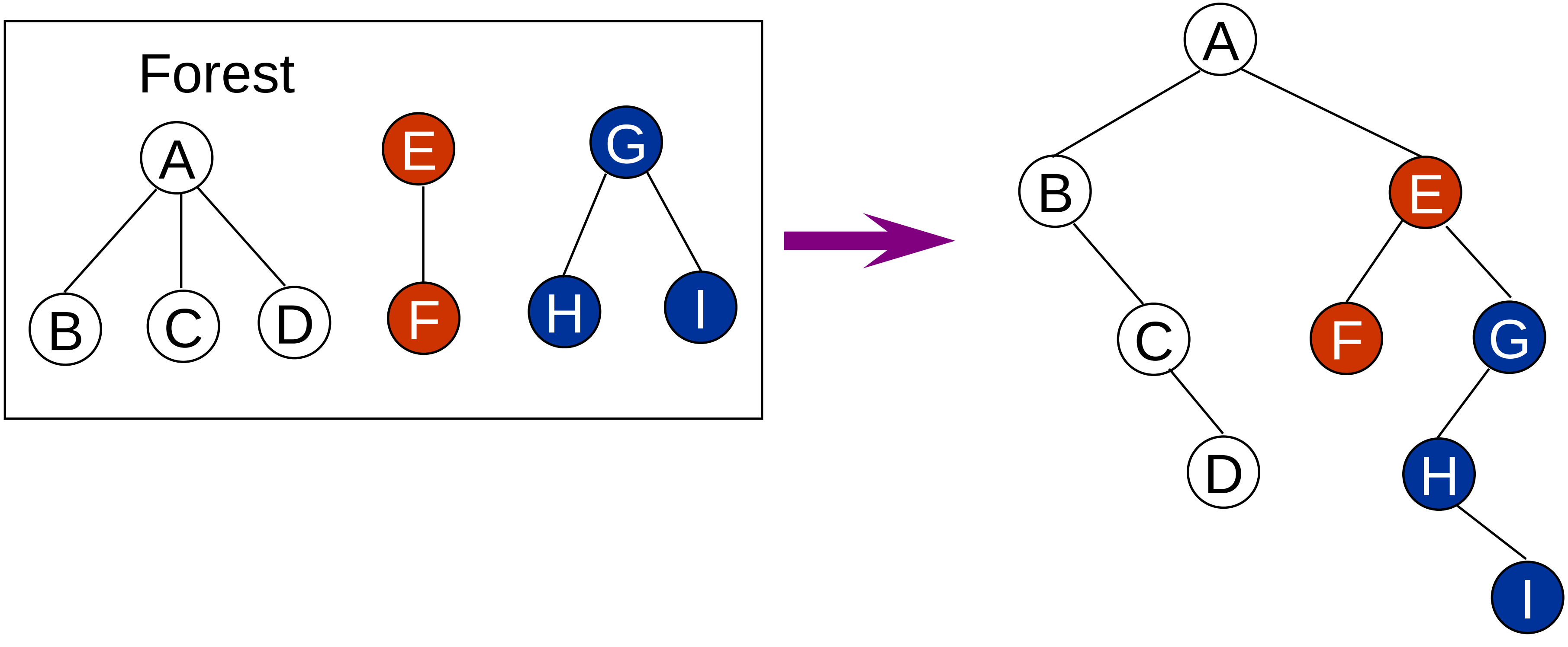
Loser Tree



0
8
Overall winner
1
9
2
15
3
17
4
10
5
20
6
9
7
90
10
9
9
10
20
15
11
12
8
9
13
14
90
17
15
run
1
2
3
4
5
6
7
8

Forest

- A forest is a set of $n \geq 0$ disjoint trees

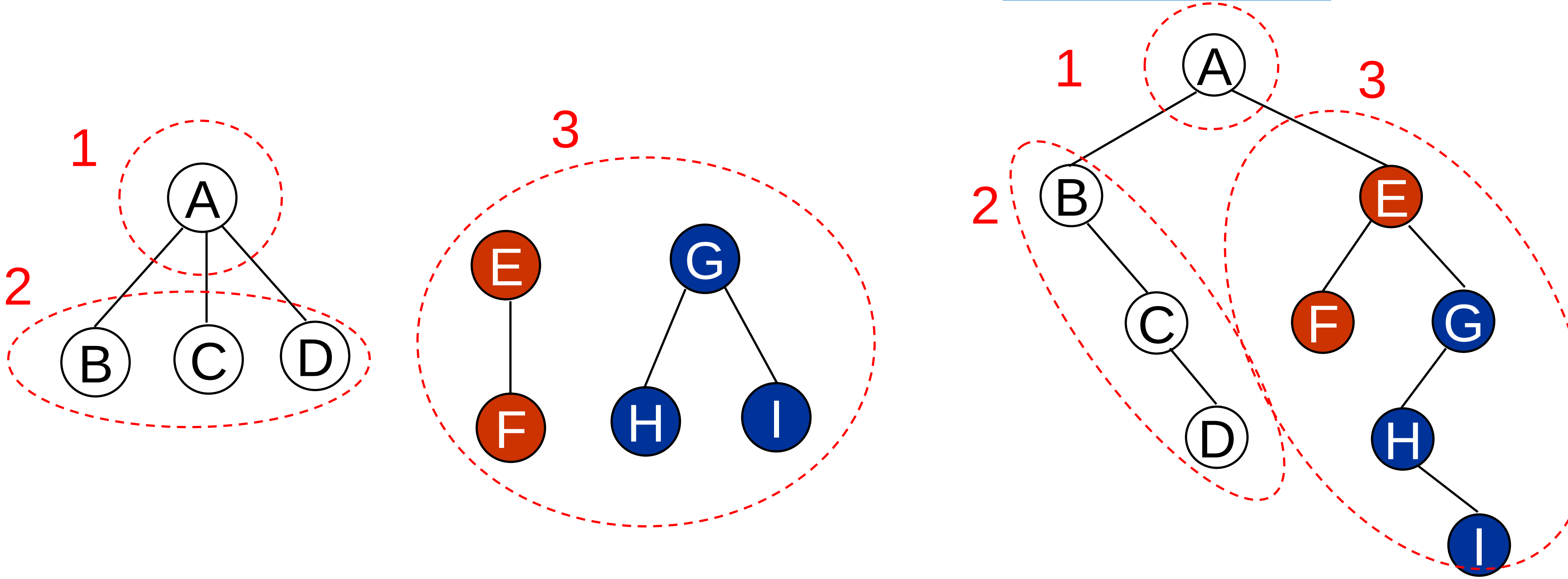


Transform a Forest into a Binary Tree

- $T_1, T_2, \dots, T_n$: a forest of trees
- $B(T_1, T_2, \dots, T_n)$: a binary tree corresponding to this forest
- Algorithm
  - empty, if $n = 0$
  - has root equal to $root(T_1)$; has left subtree equal to $B(T_{11}, T_{12}, \dots, T_{1m})$; where $B(T_{11}, T_{12}, \dots, T_{1m})$ are subtrees of $root(T_1)$; and has right subtree equal to $B(T_2, \dots, T_n)$

Forest Traversals

- 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



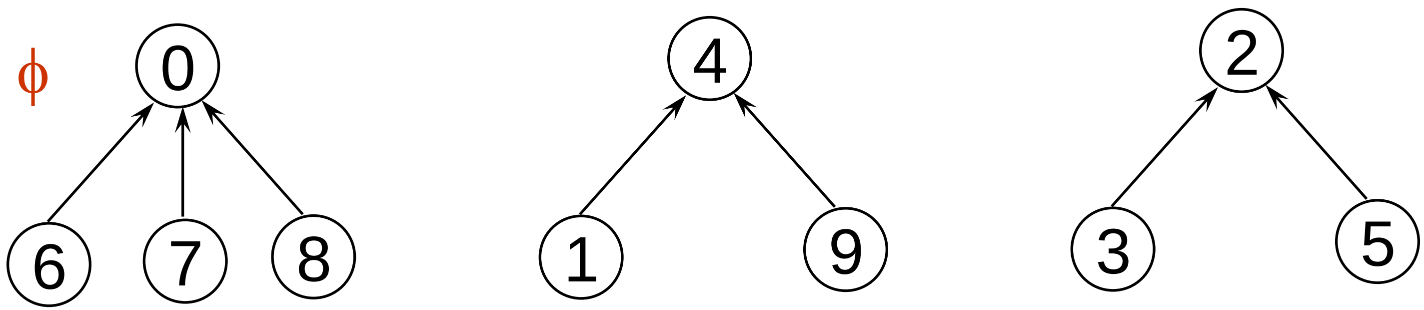
Forest Traversals (contd.)

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

(Disjoint) Set Representation

- $S_1 = \{0, 6, 7, 8\}, S_2 = \{1, 4, 9\}, S_3 = \{2, 3, 5\}$

$S_i \cap S_j = \phi$



- Two operations considered here
  - *Disjoint set union*: $S_1 \cup S_2 = \{0, 6, 7, 8, 1, 4, 9\}$
  - $Find(i)$: Find the set containing the element i.

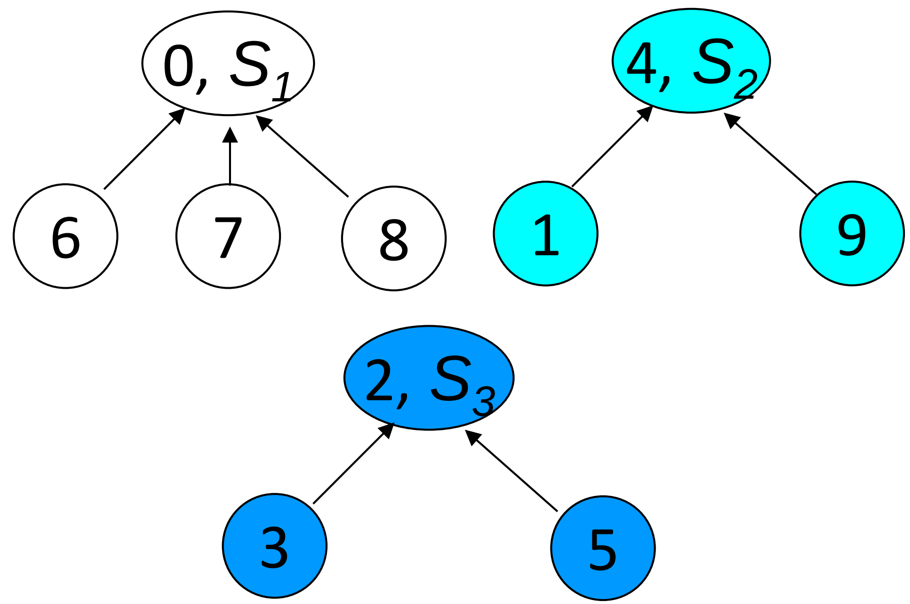
$3 \in S_3, 8 \in S_1$

Array Representation of S_1, S_2, S_3

- We could use an array for the set name.
  - Or the set name can be an element at the root.

set name	pointer
S_1	
S_2	
S_3	
…	

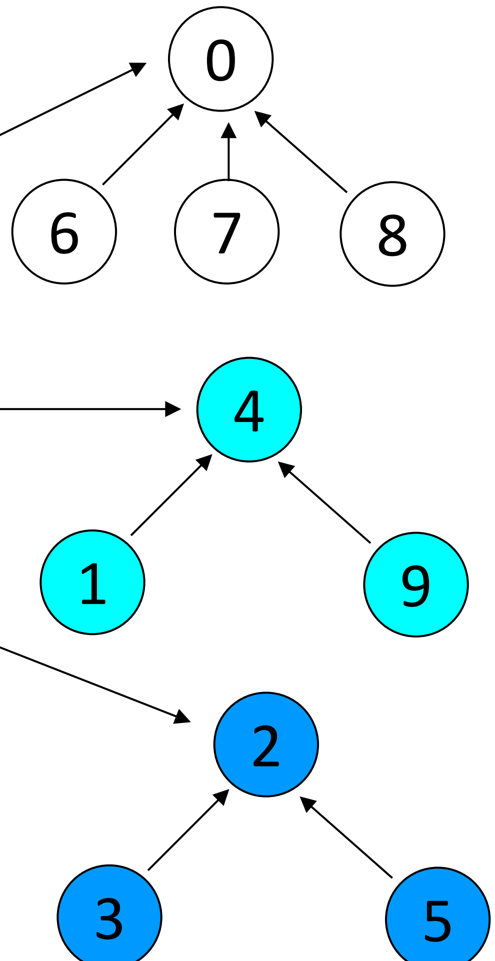
or



Array Representation of S_1, S_2, S_3

- Each root has a pointer to the set name

set name	pointer
S_1	
S_2	
S_3	

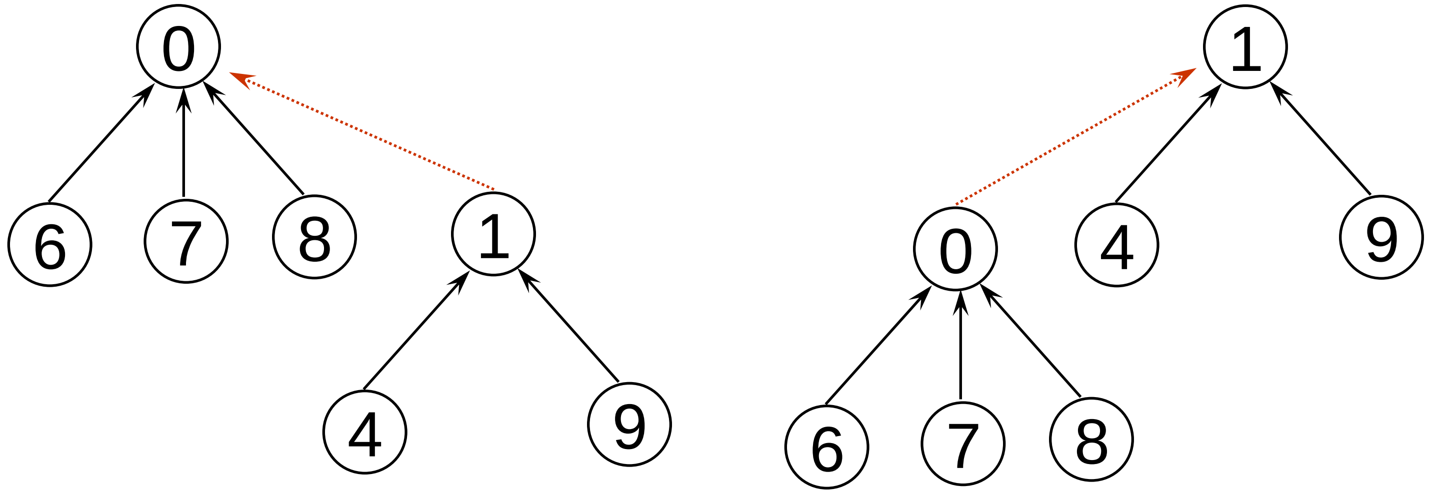


$name[k]$: get the set name

$FindPointer(S_j)$: get the pointer of tree with set name S_j

Disjoint Set $Union$

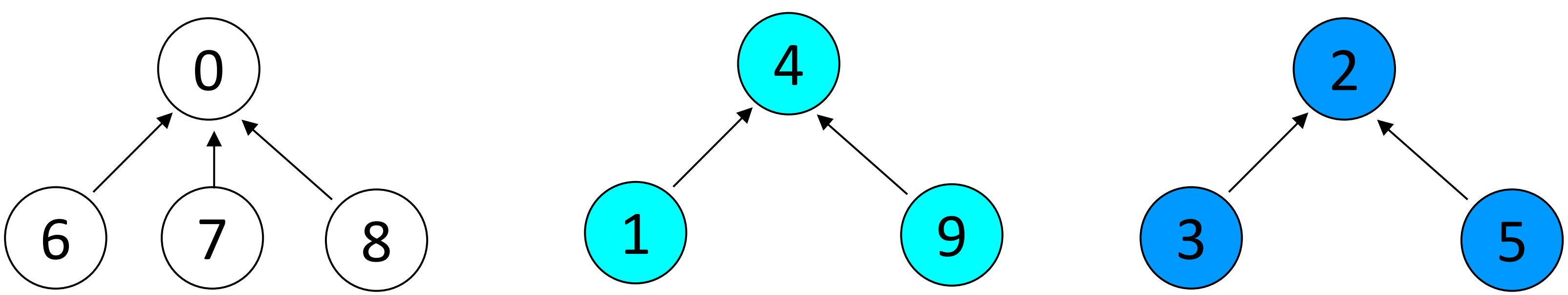
- Make one of trees a subtree of the other
  - Possible representation for $S_1 \cup S_2$



Array Representation of S_1, S_2, S_3

- Assume set elements are numbered 0 through $n-1$

i	[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]
parent	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1



i	[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]
parent	-1	4	-1	2	-1	2	0	0	0	4

root

Array Representation of S_1, S_2, S_3

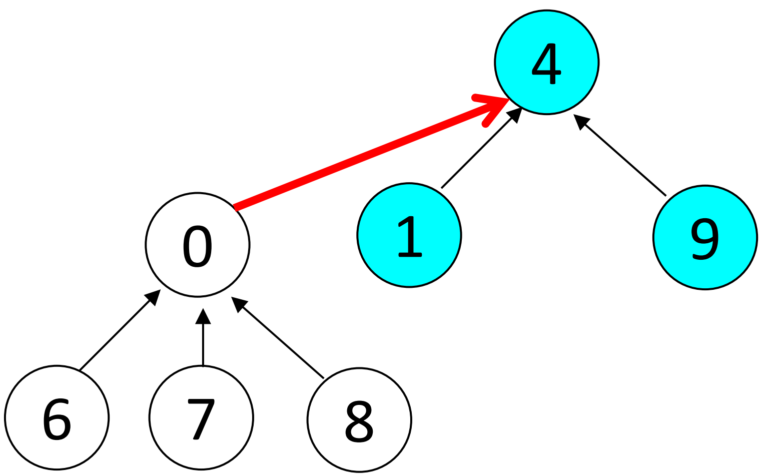
```
class Sets {
public:
    // set operations follow
    …
private:
    int *parent;
    int n; // number of set elements
};

Sets::Sets(int numberOfElements)
{
    if (numberOfElements < 2)
        throw "Must have at least 2 elements.";
    n = numberOfElements;
    parent = new int[n];
    fill(parent, parent + n, -1);
}
```

SimpleUnion

```
void Sets::SimpleUnion(int i, int j)
{// replace the disjoint sets with roots i and j, and i != j with their union
        parent[i] = j;
}
```

i	[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]
parent	-1	4	-1	2	-1	2	0	0	0	4

$$SimpleUnion(0,4)$$



i	[0]	[1]	[2]	[3]	[4]
parent	4	4	-1	2	-1

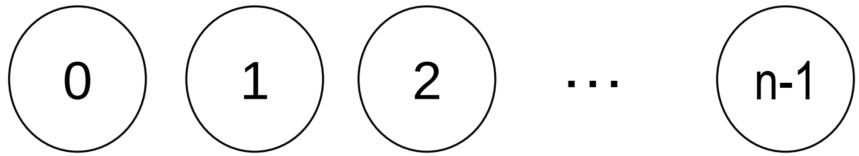
SimpleFind

```
int Sets::SimpleFind(int i)
{ // find the root of the tree containing element i
    while (parent[i] >= 0) i = parent[i];
    return i;
}
```

$$SimpleFind(9) \Rightarrow 4$$

i	[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]
parent	-1	4	-1	2	-1	2	0	0	0	4

Degenerate Tree

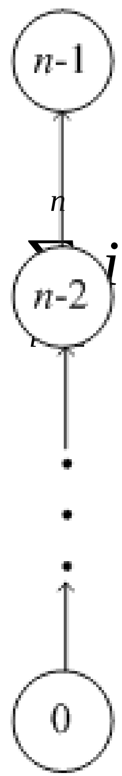


- $union(0,1)$
- $union(1,2)$

  ...

- $union(n-2, n-1)$

- One union operation
  - $O(1)$
- $n-1$ union operations
  - $O(n)$



degenerate tree

- $find(0)$
- $find(1)$

  ...

- $find(n-1)$

- One find operation
  - $O(n)$
- n find operations
  - $O(n^2)$

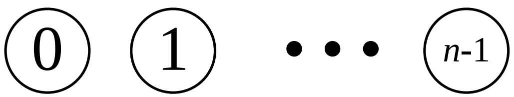
Weighting Rule

- Weighting rule for $union(i, j)$
  - If the number of nodes in the tree with root i is less than the number in the tree with root j, then make j the parent of i; otherwise make i the parent of j.

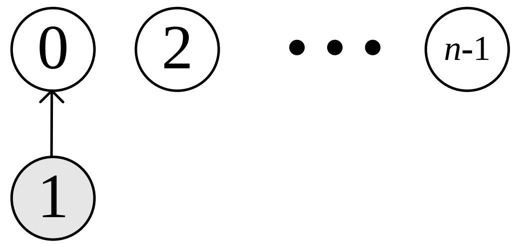
```
void Sets::WeightedUnion(int i, int j)
// union sets with roots iand j, i≠j，using the weighting rule
// parent[i] = -count[i] andparent[j] = -count[j]
{
    int temp = parent[i] + parent[j];
    if (parent[i] > parent[j]) {// i has fewer nodes
        parent[i] = j;
        parent[j] = temp;
    }
    else {// j has fewer nodes or i and j have the same no. of nodes
        parent[j] = i;
        parent[i] = temp;
    }
}
```

Use the weighting rule on the $union$ operation to avoid the creation of degenerate trees.

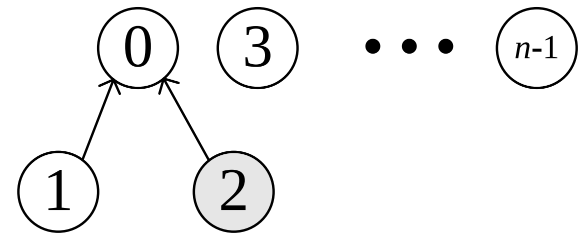
Trees Obtained Using The Weighting Rule



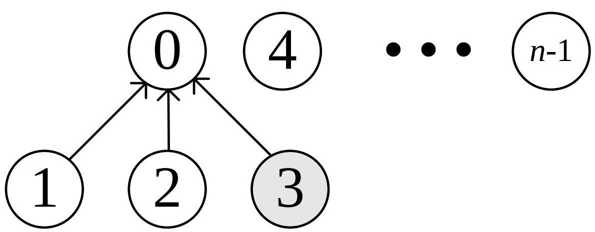
initial



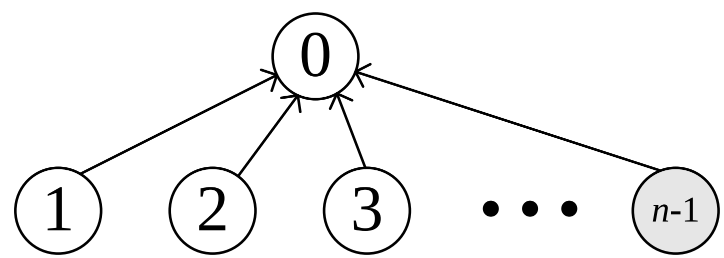
Union (0,1)



Union (0,2)



Union (0,3)



Union (0,n-1)

Weighted Union

- **Lemma 5.5**: Assume that we start with a forest of trees, each having one node. Let T be a tree with m nodes created as a result of a sequence of unions each performed using $WeightedUnion$. The height of T is no greater than $\lfloor \log_2 m \rfloor + 1$.
- Prove by induction.
  - Basis: The lemma is true when *m=1*
  - Hypothesis: Assume that the three The lemma is true for all trees with i nods, $i < m$.

Weighted Union (contd.)

- Induction:
  - Let T be a tree wit m nodes created by $WeightedUnion$ to union two trees j and k.
  - Let tree j and tree k be with a and $m - a$ nodes, respectively.
  - Assume $1 \leq a \leq m/2$. The height of tree T is either

    (1) the same as tree k or

$$\lfloor \log_2 (m-a) \rfloor + 1 \leq \lfloor \log_2 m \rfloor + 1$$

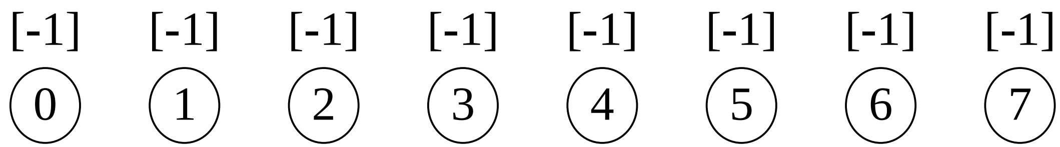
    (2) one more than the height of tree j

$$\lfloor \log_2 a \rfloor + 1 + 1 \leq \left\lfloor \log_2 \frac{m}{2} \right\rfloor + 2 \leq \lfloor \log_2 m \rfloor + 1$$

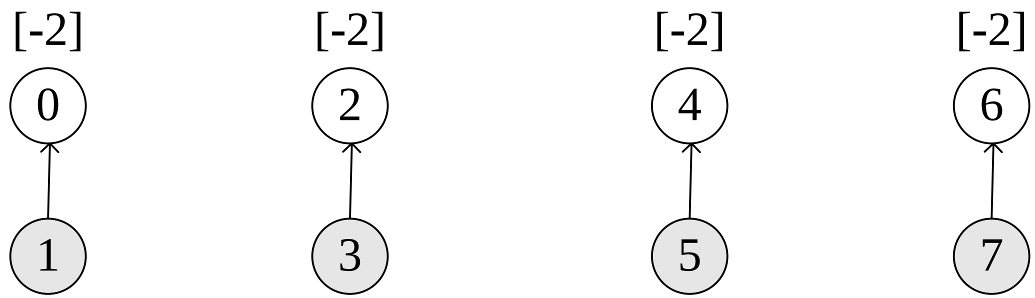
Weighted Union (contd.)

- For the processing of an intermixed sequence of *u–1* unions and f find operations, the time complexity is *O(u+f*log u)*.
- Worst case: Performing $u - 1$ unions first and then performing f find operations
  - $u - 1$ unions: $O(u)$
  - The largest three is of at most u nodes. According lemma 5.5, the height of a tree is at most $O(\log u)$
  - Find operation: $O(\log u)$

Trees Achieving Worst-Case Bound

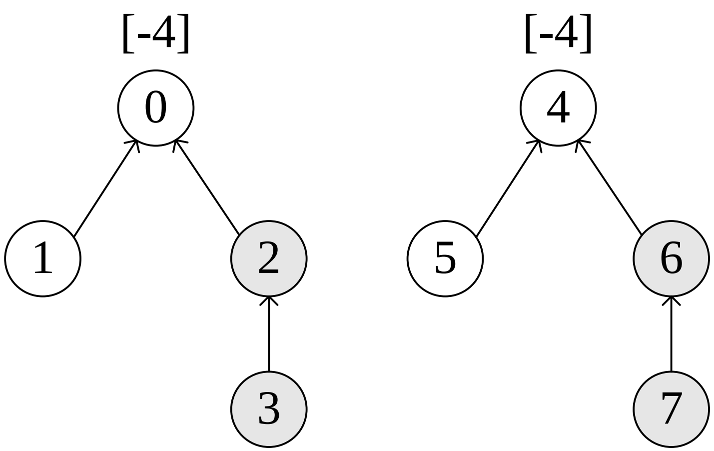


(a) Initial height trees

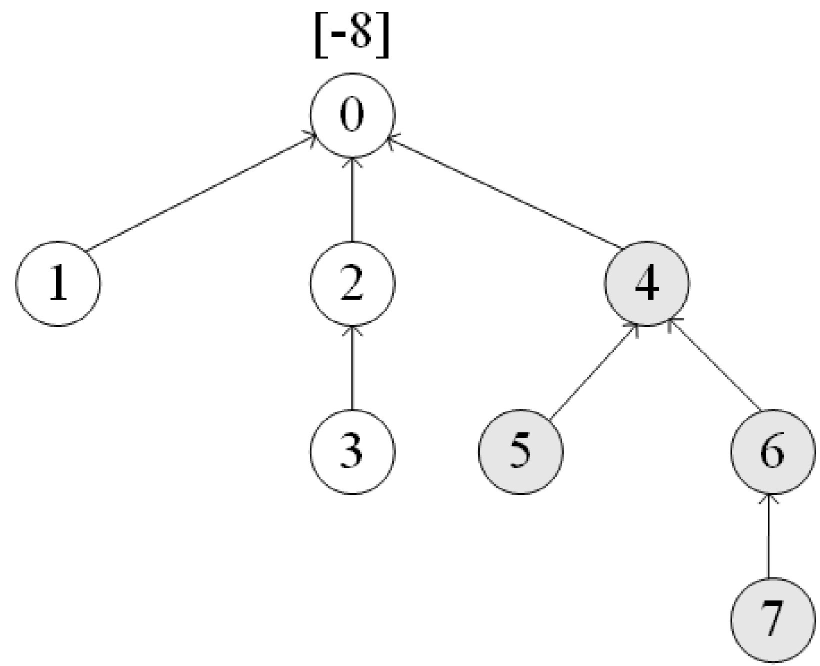


(b) Height-2 trees following union (0, 1), (2, 3), (4, 5), and (6, 7)

Trees Achieving Worst-Case Bound (Cont.)

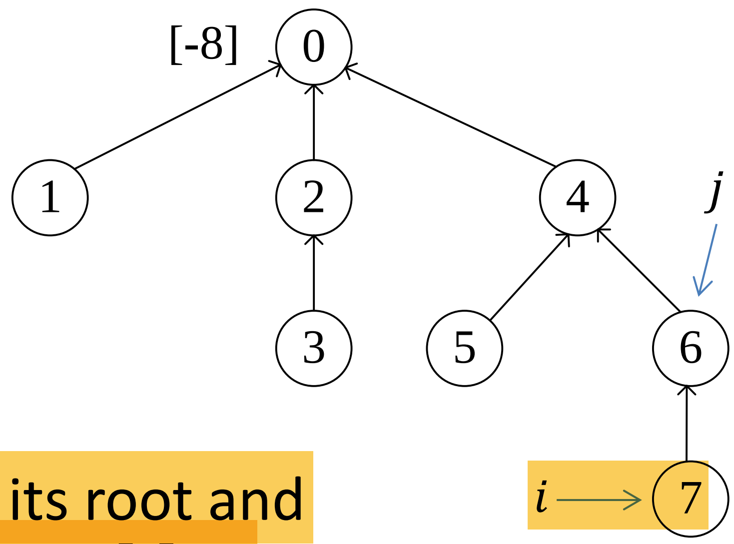


(c) Height-3 trees following union (0, 2), (4, 6)



(d) Height-4 trees following union (0, 4)

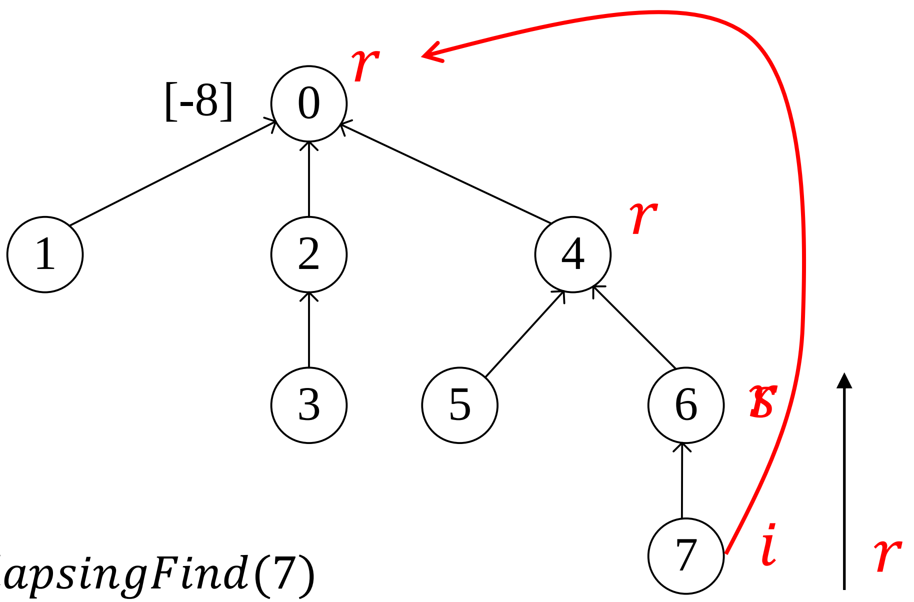
Collapsing Rule



- Collapsing rule:
  - If j is a node on the path from i to its root and $parent[i] \neq root(i)$, then set $parent[j]$ to $root(i)$.
- The first run of find operation will collapse the tree.
  - Each following find operation of the same element only goes up one link to find the root.

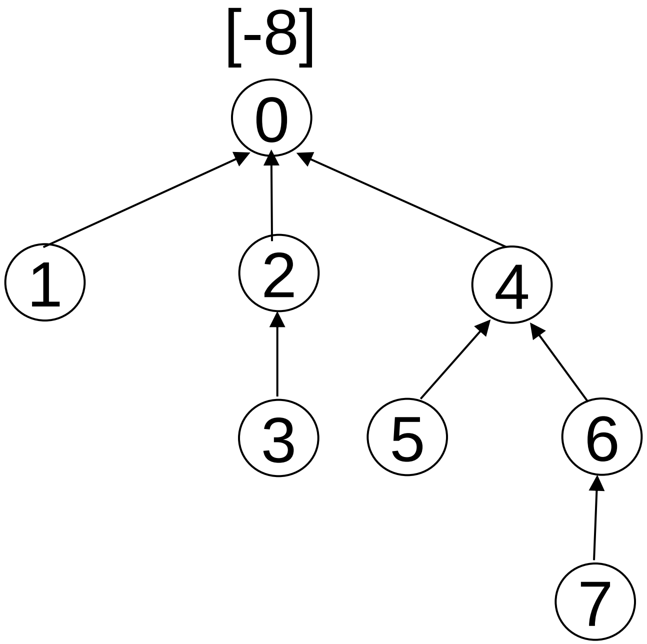
CollapsingFind (contd.)

```
int Sets::CollapsingFind(int i)
{// find the root of the tree containing element i
  // use the collapsing ruleto collapse all nodes from i to the root
      for (int r = i; parent[r] >= 0; r = parent[r]); // find root
      while (i != r) {// collapse
            int s = parent[i];
            parent[i] = r;
            i = s;
      }
      return r;
}
```

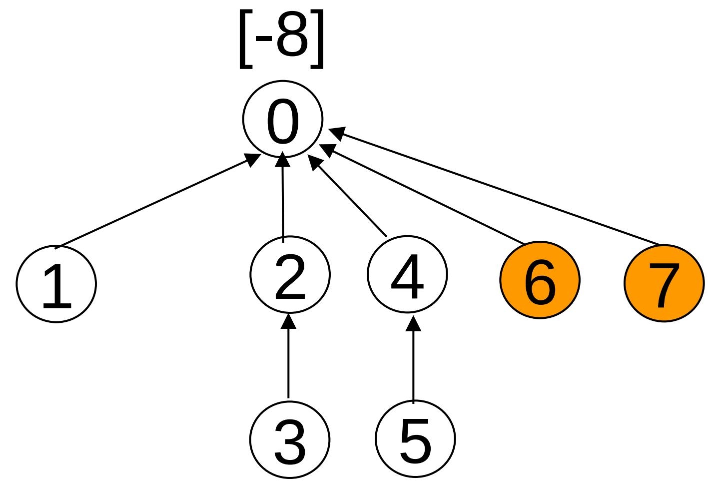


$CollapsingFind(7)$

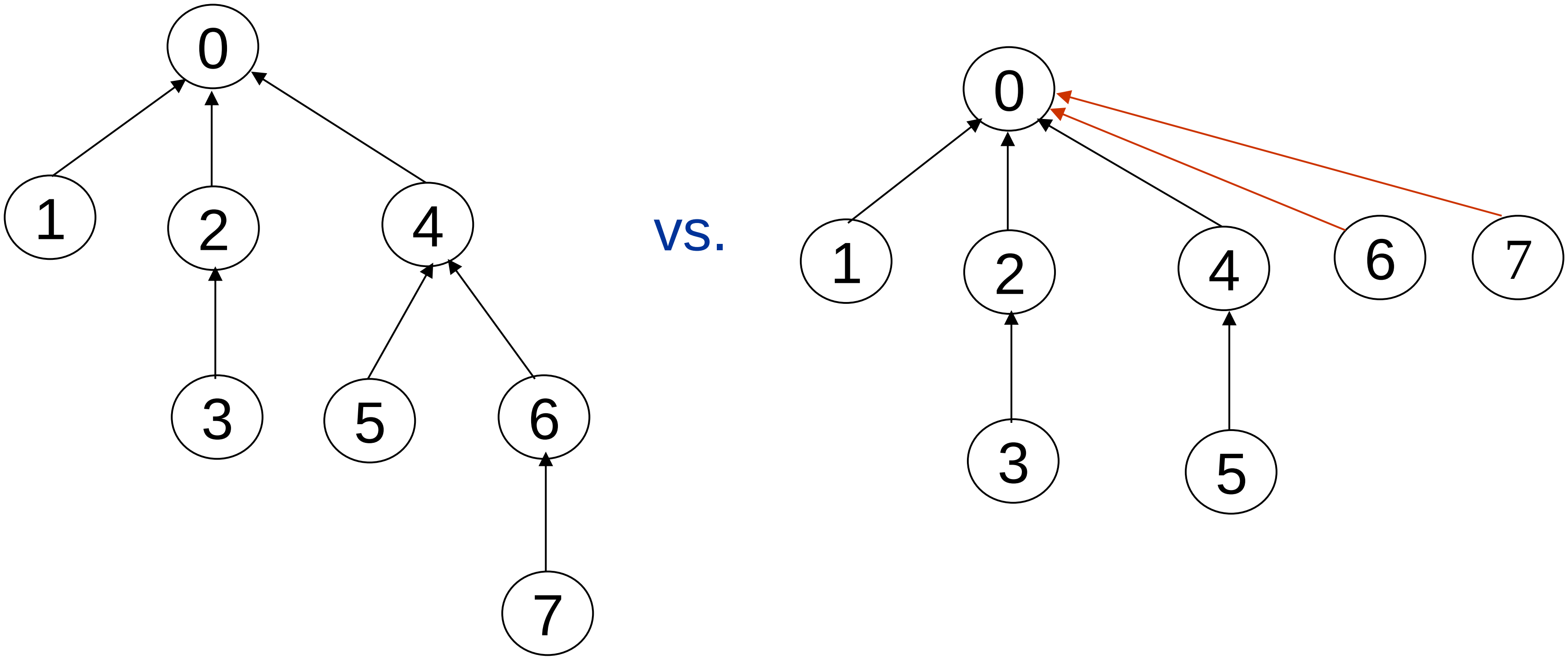
CollapsingFind



Before collapsing



After collapsing



	find(7)	find(7)	find(7)	find(7)	find(7)	find(7)	find(7)	find(7)
go up	3	1	1	1	1	1	1	1
reset	3							

13 moves (vs. 24 moves)

Applications

- Find equivalence class $i \quad j$

(two finds)

- Find S_i and S_j such that $i \in Si$ and $j \in S_j$
  - $S_i = S_j$ do nothing
  - $S_i \neq Sj$ do $union(S_i, S_j)$

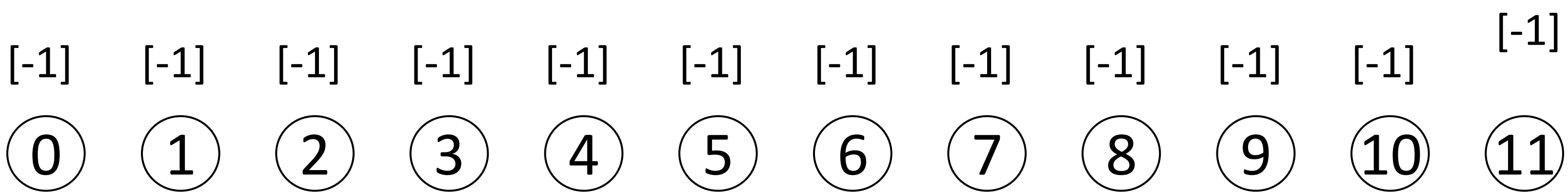
- Example

$$0 \equiv 4, 3 \equiv 1, 6 \equiv 10, 8 \equiv 9, 7 \equiv 4, 6 \equiv 8, 3 \equiv 5, 2 \equiv 11, 11 \equiv 0$$

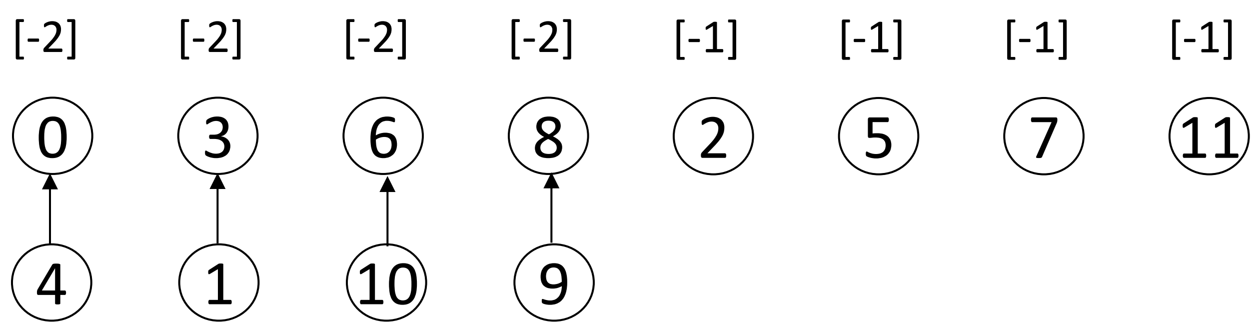
{0, 2, 4, 7, 11}, {1, 3, 5}, {6, 8, 9, 10}

Example 5.5

$0 \equiv 4, 3 \equiv 1, 6 \equiv 10, 8 \equiv 9, 7 \equiv 4, 6 \equiv 8, 3 \equiv 5, 2 \equiv 11, 11 \equiv 0$



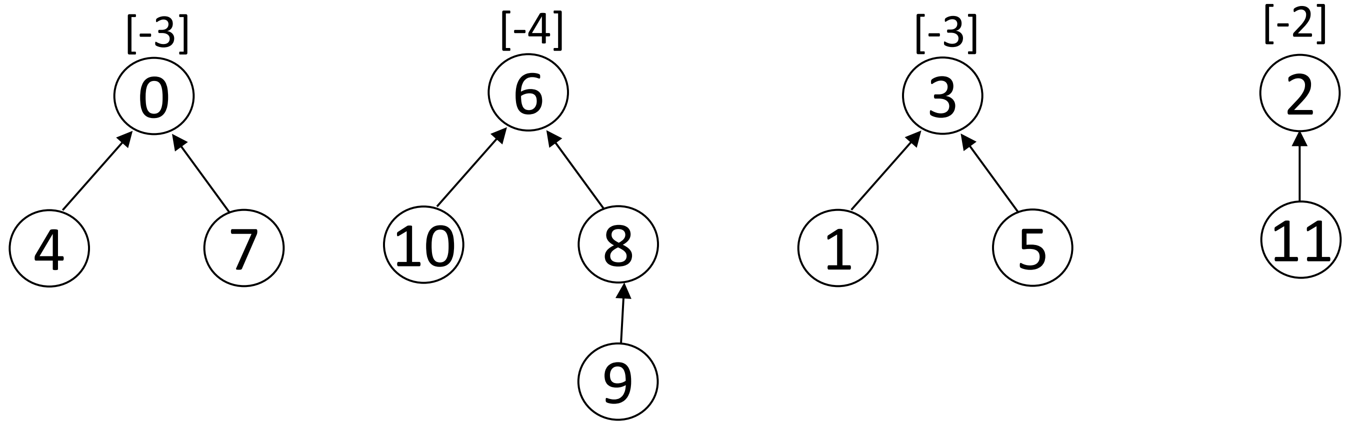
(a) Initial trees



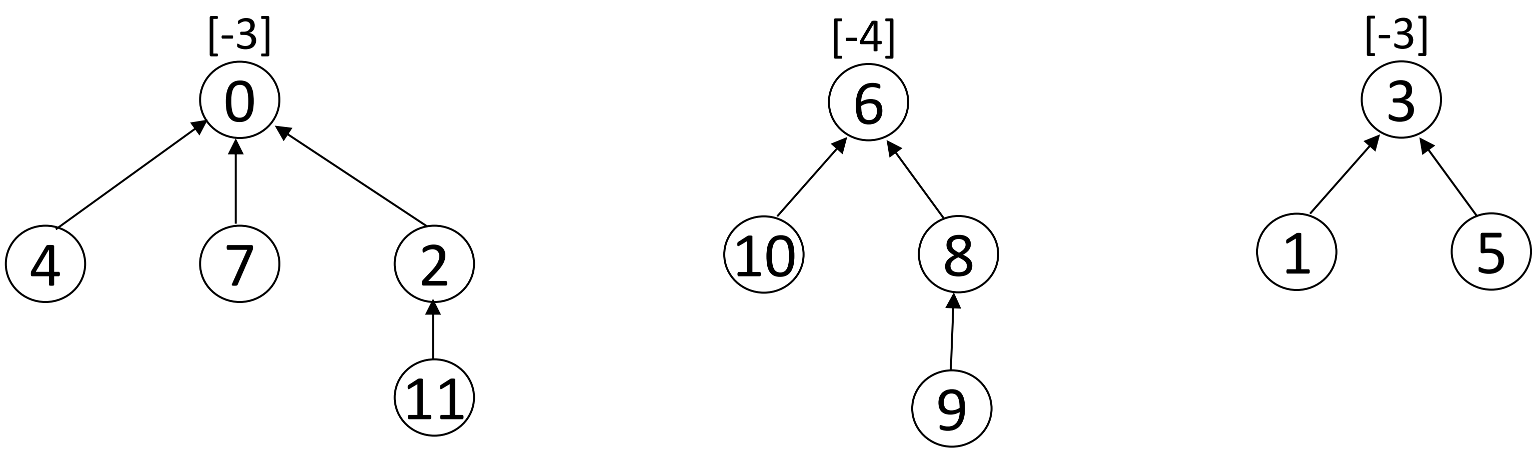
(b) Height-2 trees following 0≡4, 3≡1, 6≡10, and 8≡9

Example 5.5 (Cont.)

$0 \equiv 4, 3 \equiv 1, 6 \equiv 10, 8 \equiv 9, 7 \equiv 4, 6 \equiv 8, 3 \equiv 5, 2 \equiv 11, 11 \equiv 0$



(c) Trees following 7≡4, 6≡8, 3≡ 5, and 2≡11



(d) Trees following 11≡0

Complexity

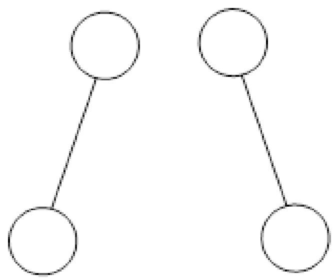
- n number and m equivalence pairs
  - Setup: $O(n)$
  - Find no: $O(2m)$
  - Union no: At most $\min\{n-1, m\}$

Counting Binary Trees

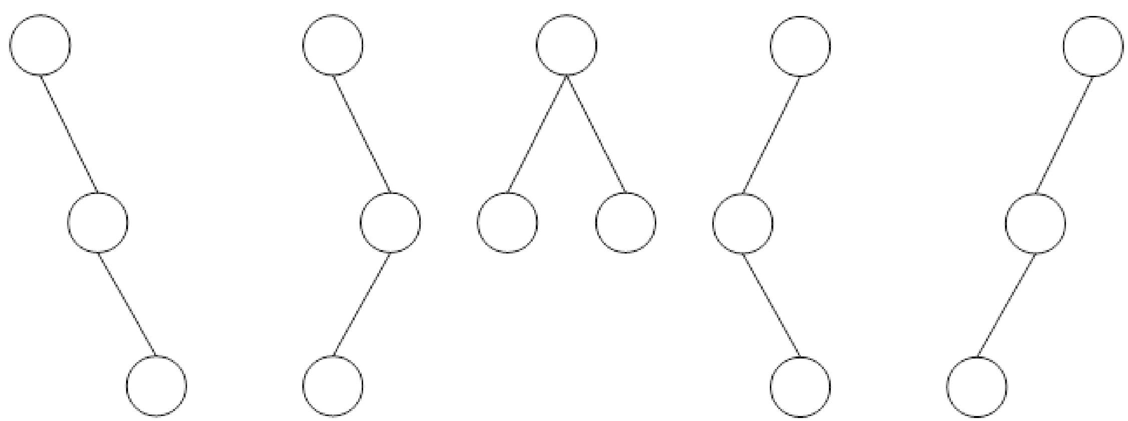
- Problem 1: The no. of distinct trees with n nodes
- Problem 2: The no. of distinct permutations of a the numbers from 1 through n obtaining by a stack
- Problem 3: The no of distinct ways of multiplying $n-1$ matrices

Distinct Binary Trees

- $n = 1$ or $n = 1$ => only one binary tree
- $n = 2$



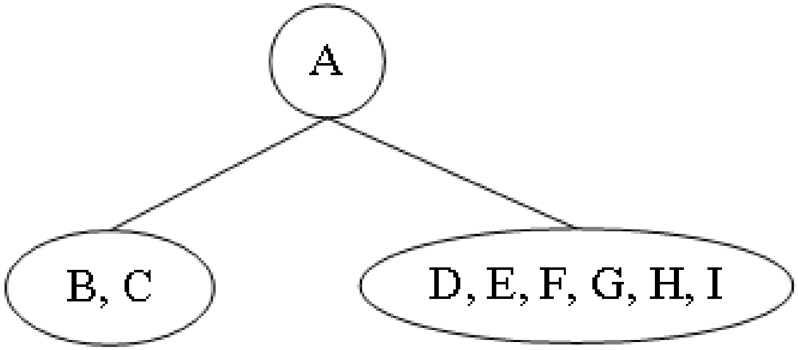
- $n = 3$

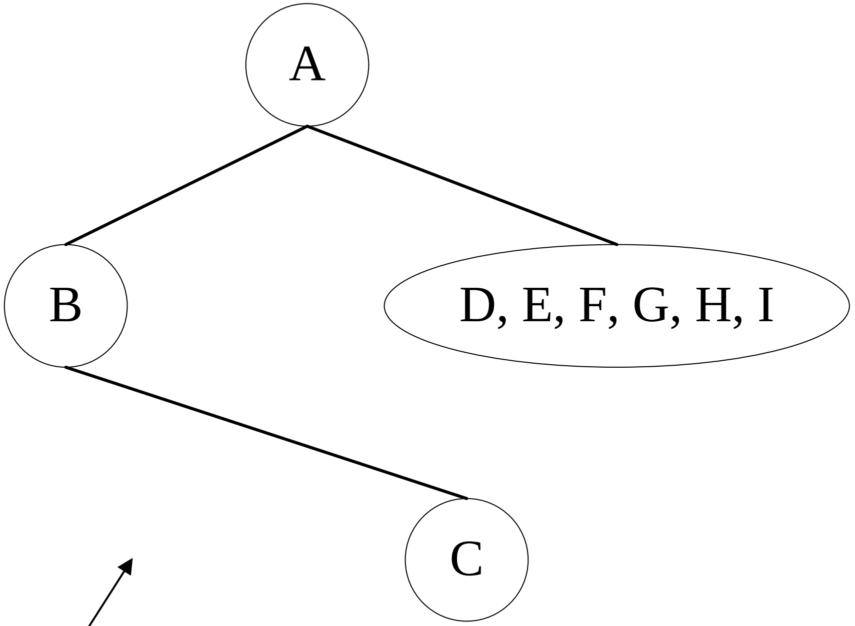


Uniqueness of a Binary Tree

- Suppose that we have the preorder sequence ABCDEFGHI and the inorder sequence BCAEDGHFI of the same binary tree.
- Does such a pair of sequence uniquely define a binary tree?
  - Yes.
  - How to prove it?

Constructing a Binary Tree From Its Preorder and Inorder Sequences

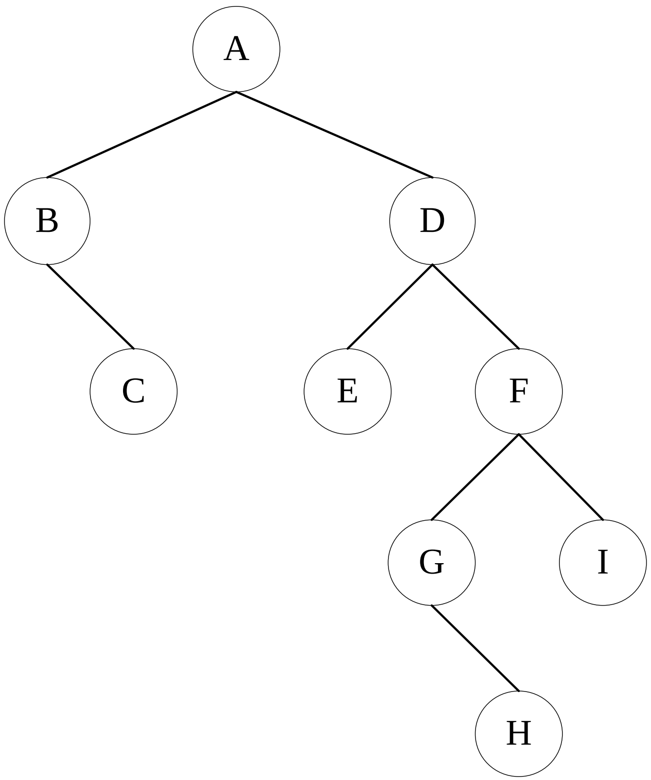


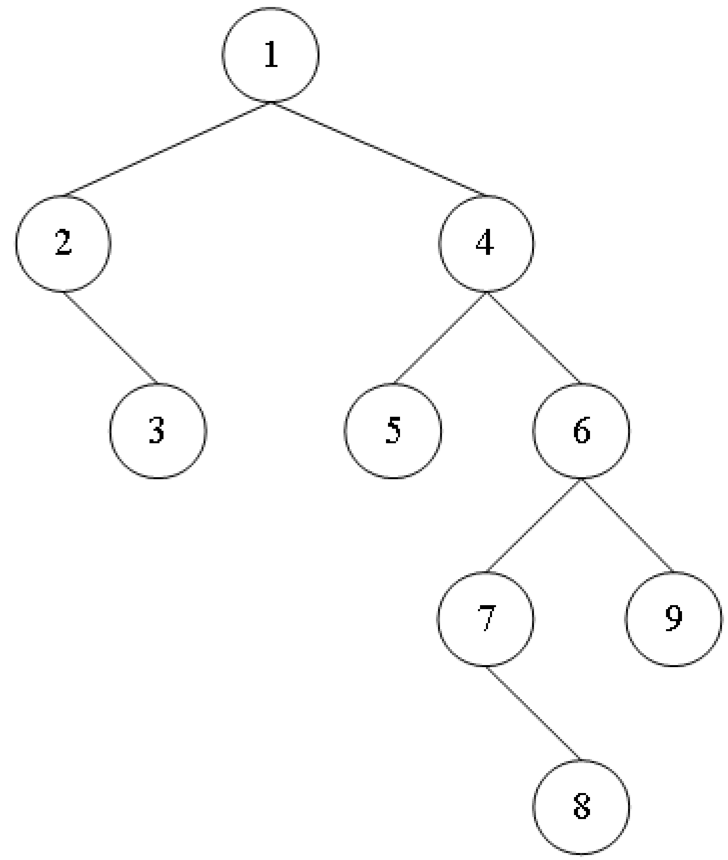


Preorder (VLR): $A\ B\ C\ D\ E\ F\ G\ H\ I$

Inorder (LVR): $B\ C\ A\ E\ D\ G\ H\ F\ I$

Constructing a Binary Tree From Its Preorder and Inorder Sequences (Cont.)





Preorder (VLR): $A\ B\ C\ D\ E\ F\ G\ H\ I$
1 2 3 4 5 6 7 8 9

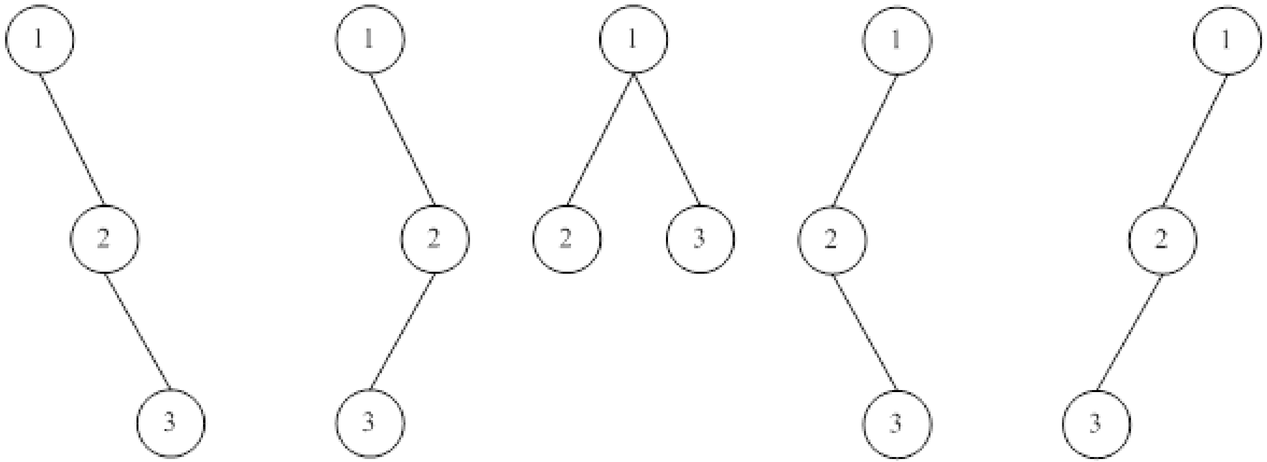
Inorder (LVR): $B\ C\ A\ E\ D\ G\ H\ F\ I$
2 3 1 5 4 7 8 6 9

Distinct binary trees define distinct inorder permutations.

Distinct Binary Trees

- No. of Distinct BT: Distinct permutations obtainable by Passing 1 through n through a stack and deleting in al possible way

preorder: (1, 2, 3)



inorder (1, 2, 3) (1, 3, 2) (2, 1, 3) (2, 3, 1) (3, 2, 1)

(3, 1, 2) is impossible