

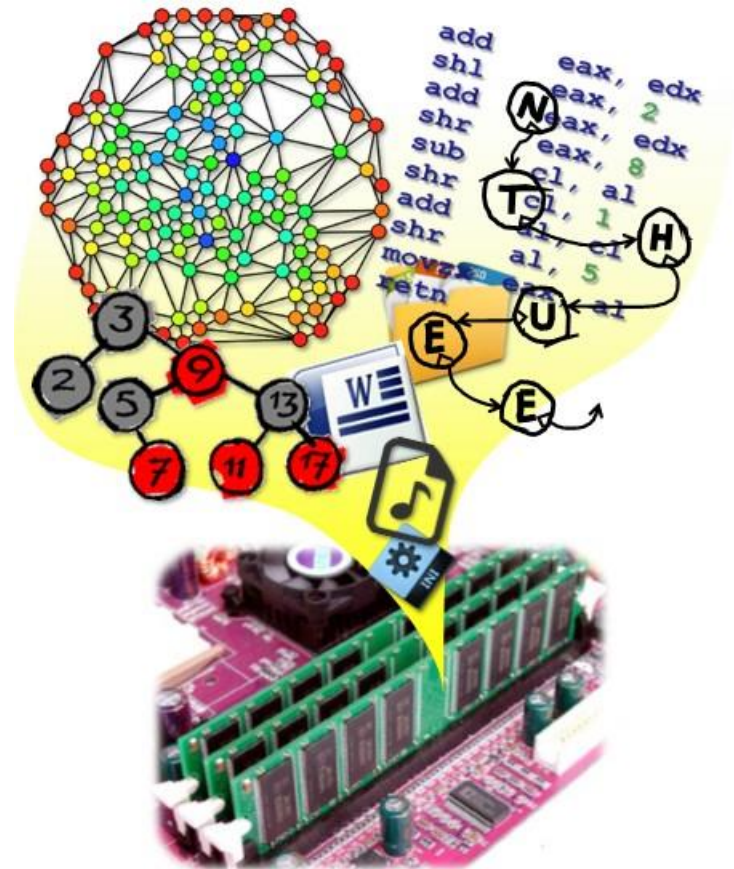
Data Structures

CH5 Trees

Prof. Ren-Shuo Liu

NTHU EE

Spring 2018



為什麼要各種 data structures

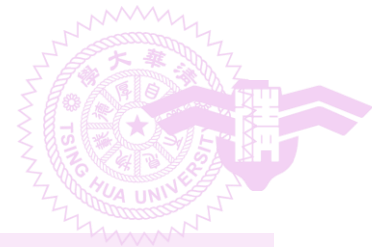


- 舉例：儲存全台灣學生的學號、姓名
 - Array
 - 最節省空間
 - Linked list
 - 新增(insert)、刪除(delete) 速度快
 - Tree
 - 搜尋(search) 速度快
 - Graph
 - 表達學生間的關聯性

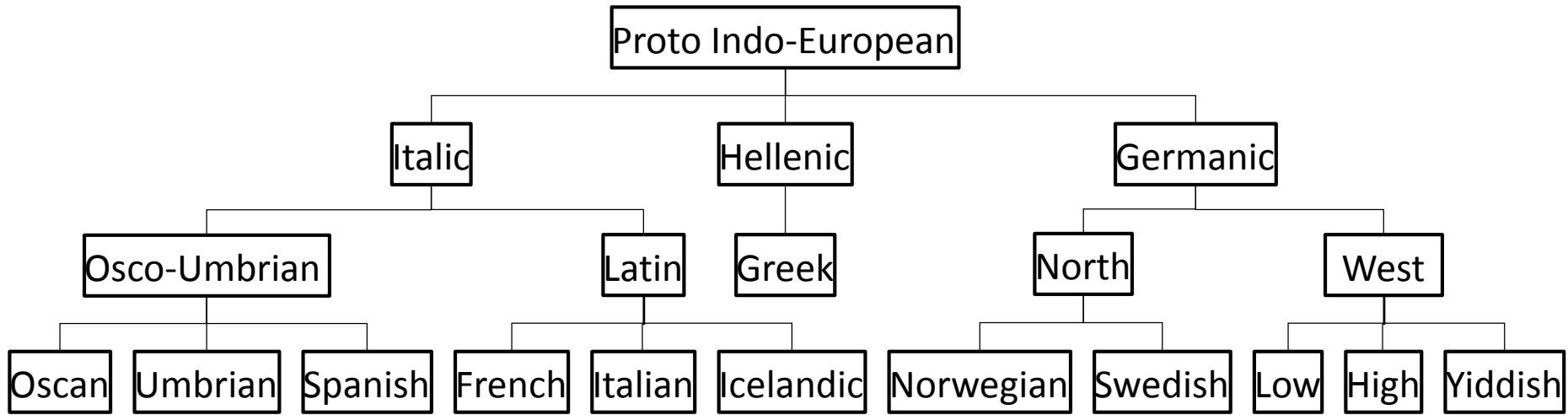
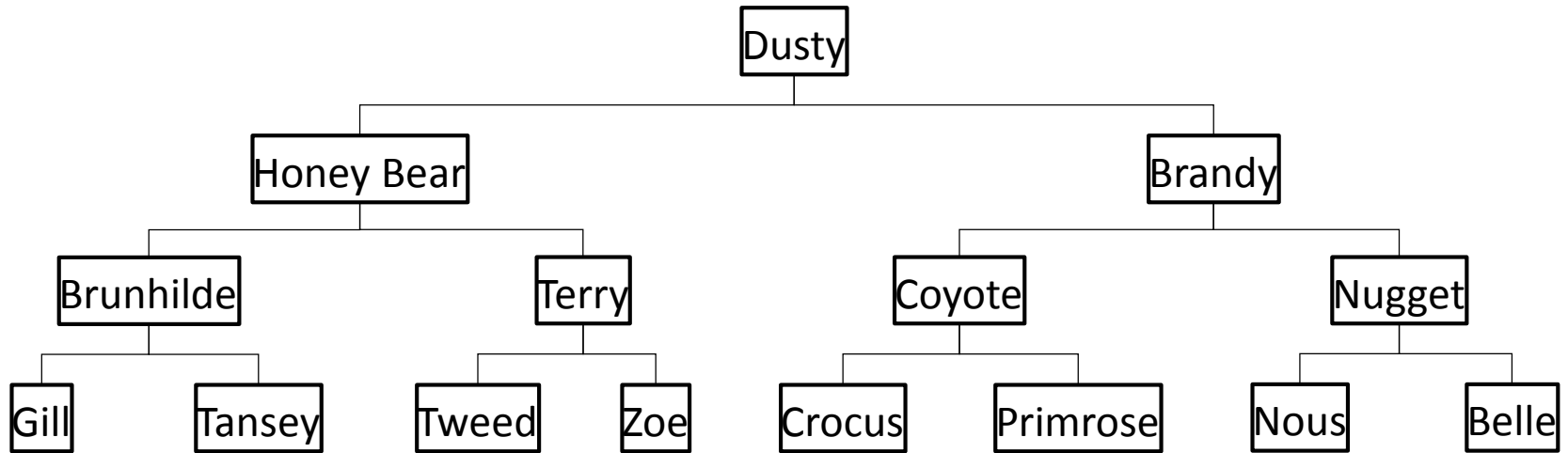


Outline

- **5.1 Introduction**
- 5.2-5.5 Binary trees
- 5.6 Heaps
- 5.7 Binary search trees
- 5.8 Selection trees
- 5.9 Forests
- (5.10 Disjoint sets)
- (5.11 Counting binary trees)



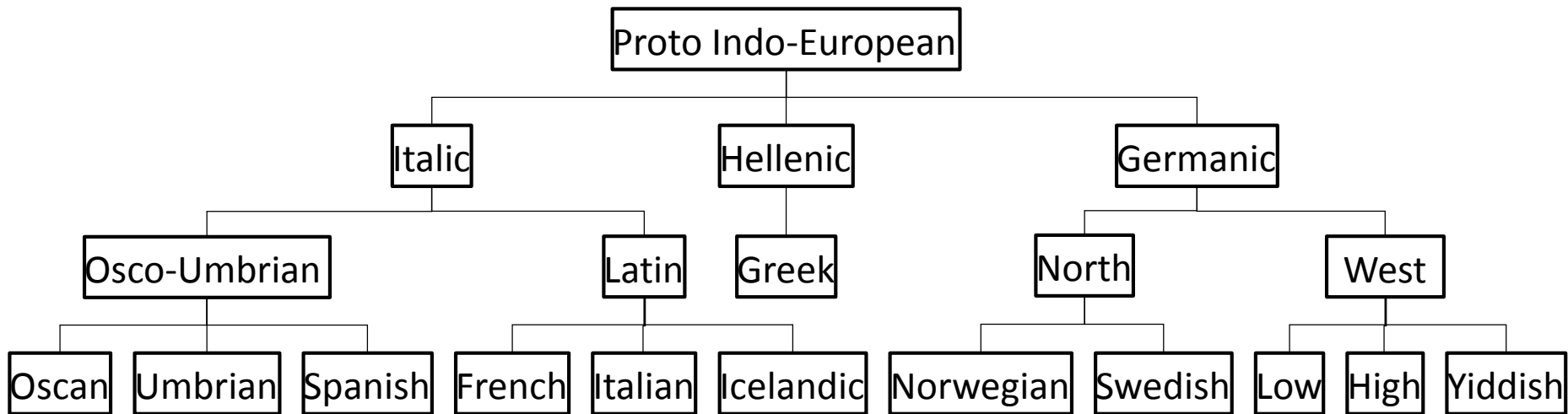
Tree-Type Charts





Tree

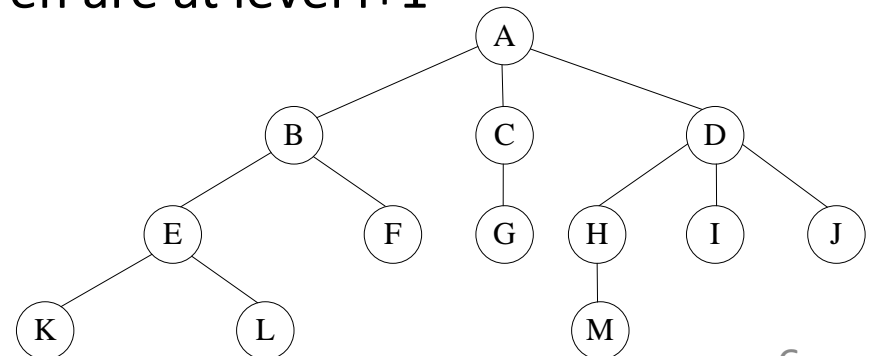
- Definition: a finite set of one or more nodes such that
 - There is a specially designated node called the **root**
 - The remaining nodes are partitioned into $n \geq 0$ disjoint sets, T_1, \dots, T_n , where each of these sets is a tree (i.e., subtree).





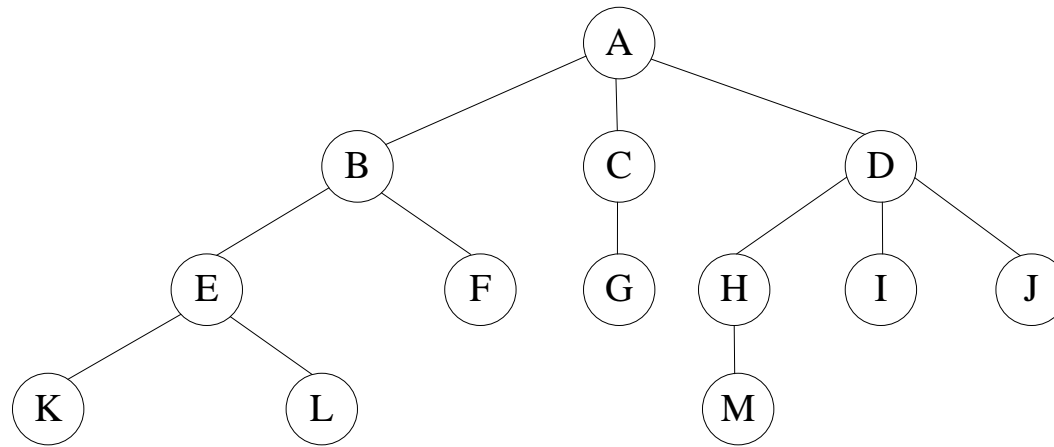
Terminologies

- The number of subtrees of a node is called the node's **degree**
- Nodes that have degree zero are called **leaf or terminal nodes**
- The root of the subtrees of a node X are the **children** of X, and X is the parent of its children
- Children of the same parent are **siblings** (兄弟姊妹)
- The **degree** of a tree is the maximum of the degree of the nodes in a tree
- The **ancestors** of a node are all the nodes along the path from the root to that node
- The **level** of a node is defined by letting the root be at level one. If a node is at level i , then its children are at level $i+1$





List Representation



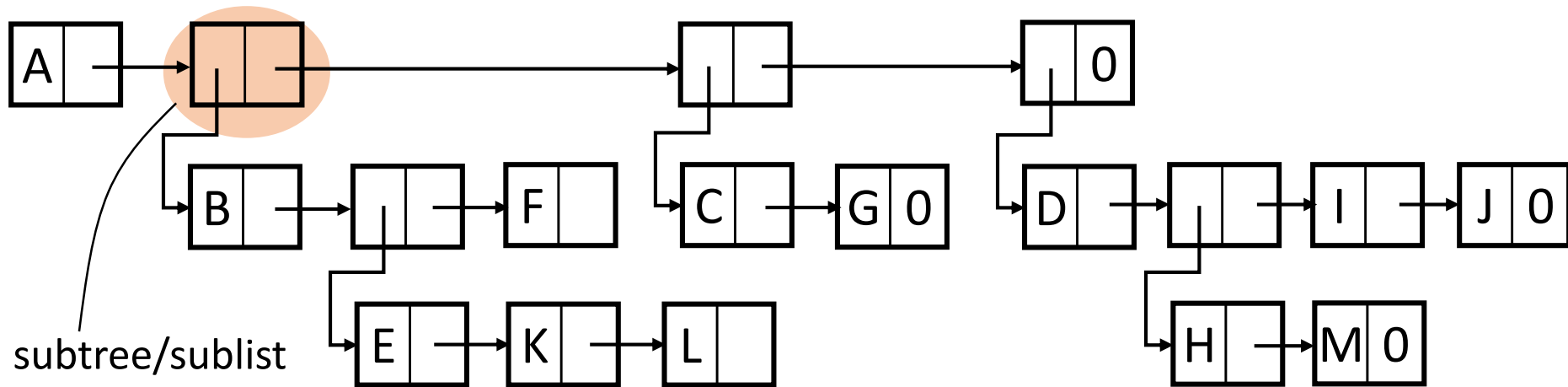
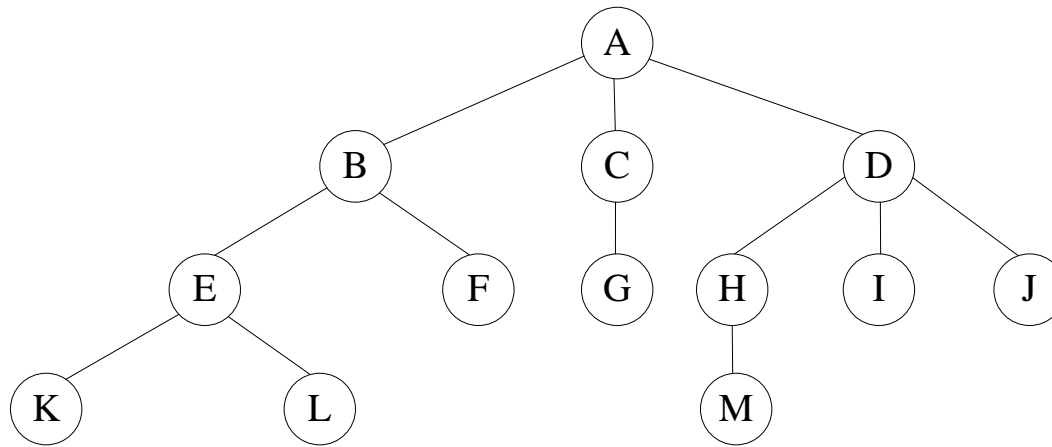
→ (A (B(...), C(...), D(...)))

→ (A (B(E(...), F), C(G), D(H(...), I, J)))

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

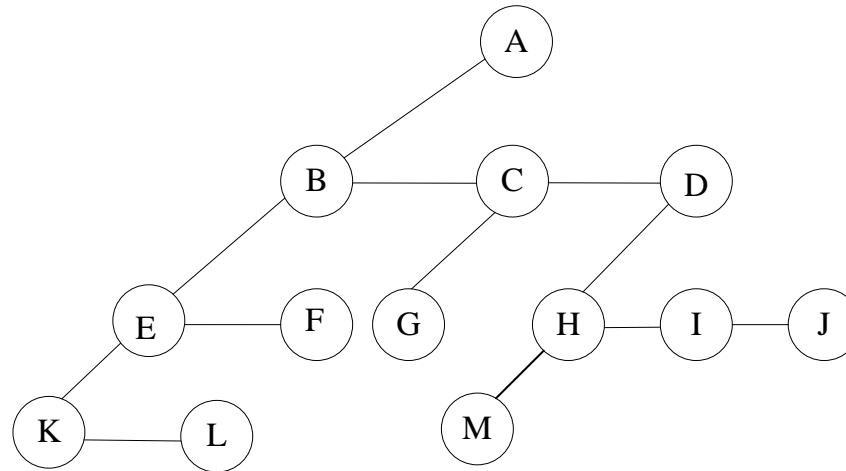
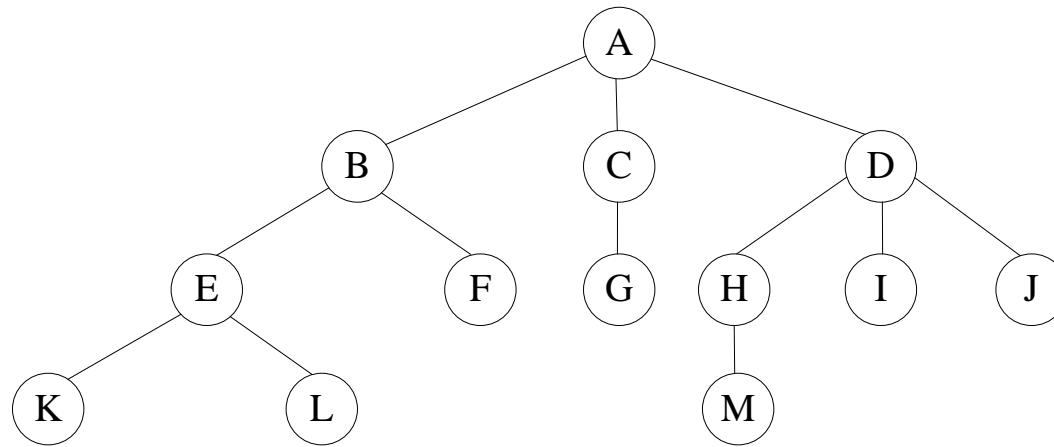


List Representation



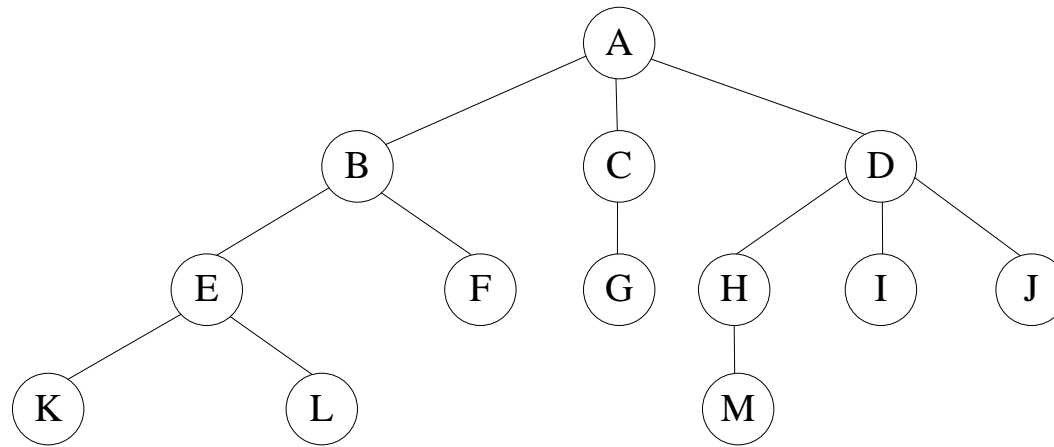


Left Child-Right Sibling Representation

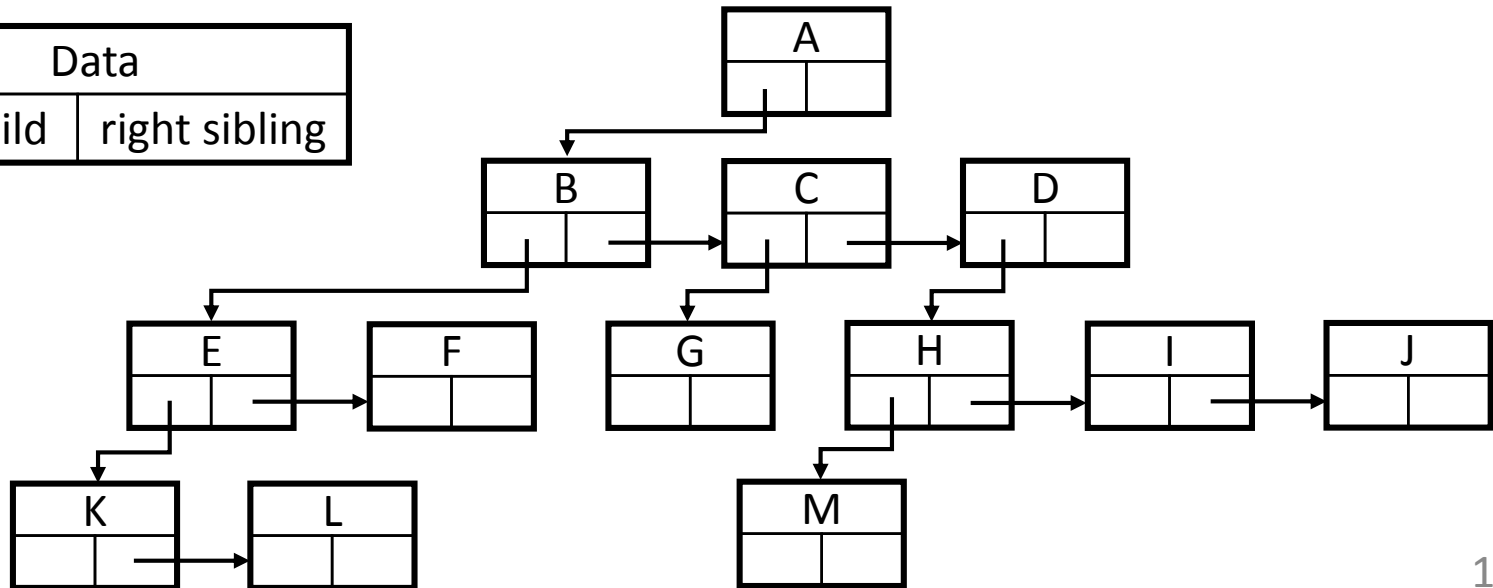




Left Child-Right Sibling Representation



Data	
Left child	right sibling

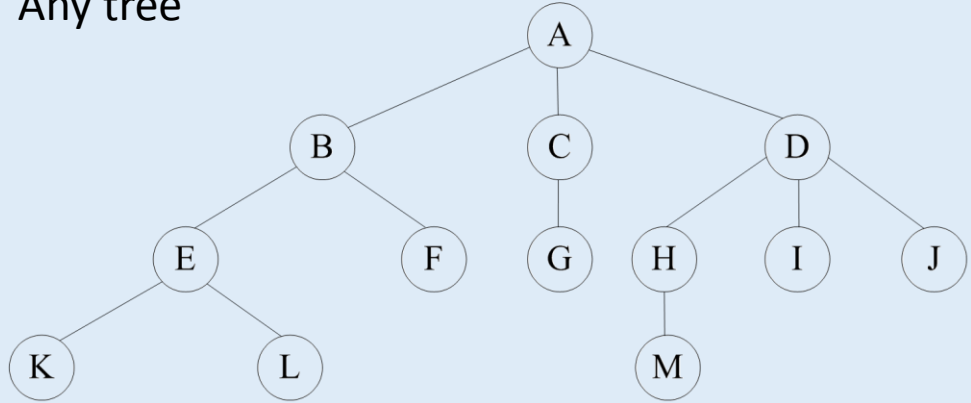


Degree-Two Tree Representation

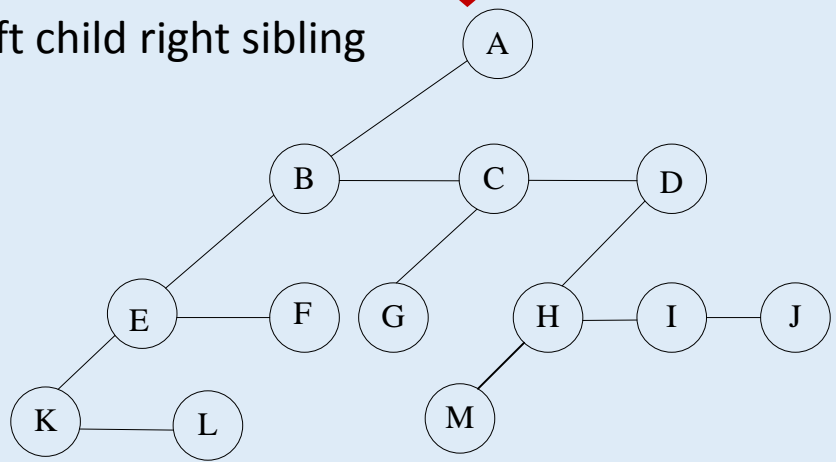


We can represent any tree as a degree-two tree

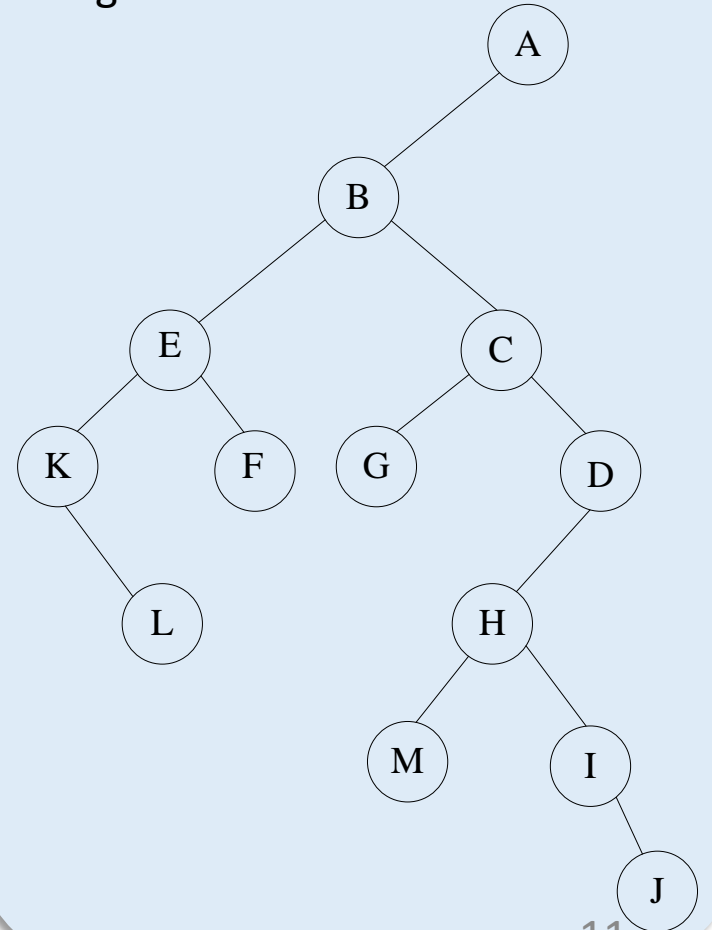
Any tree

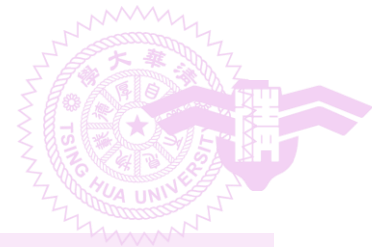


Left child right sibling



Degree two





Outline

- 5.1 Introduction
- **5.2-5.5 Binary trees**
 - **5.2 Basics**
 - 5.3 Traversal and iterators
 - 5.4 Additional operations
 - 5.5 Threaded binary tree



Binary Tree

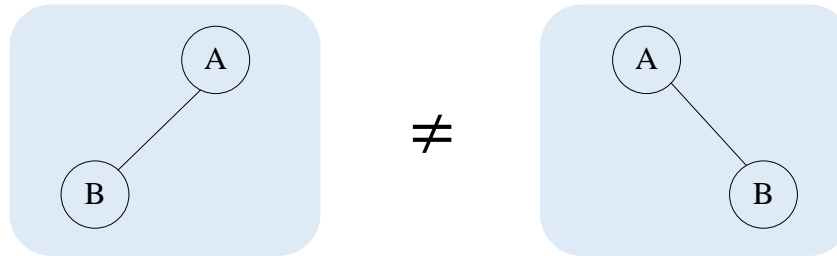
- Definition: A **binary tree** is a finite set of nodes that either is empty or consists of a **root** and two disjoint binary trees called the **left subtree** and the **right subtree**

```
template<class T>
class BinaryTree
{
public:
    BinaryTree(); // constructor for an empty binary tree
    bool IsEmpty();
    // constructor given the root and subtrees
    BinaryTree(BinaryTree<T>& bt1, T& item, BinaryTree <T>& bt2);
    BinaryTree<T> LeftSubtree(); // return the left subtree
    BinaryTree<T> RightSubtree();// return the right subtree
    T RootData(); // return the data in the root
};
```

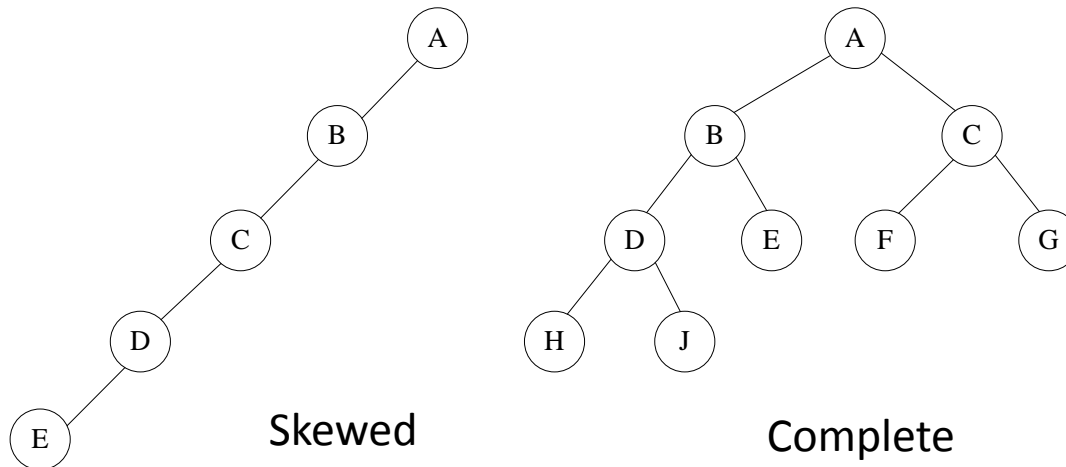


Other Definitions

- Order of children **matters** for binary trees



- Binary trees are allowed to be empty
- Skewed and complete binary tree





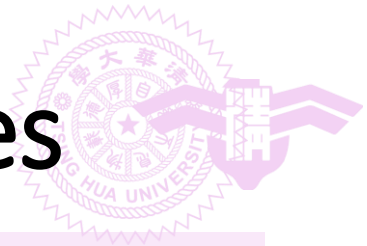
Maximum Number of Nodes

- Properties

- Maximum number of nodes at level i of a binary tree is 2^{i-1}
- Maximum number of nodes in a binary tree of depth k is $2^k - 1, k \geq 1$

- Proof

- **Induction base:** the root is the only node at level 1
- **Induction hypothesis:** maximum number of nodes at level $(i-1)$ is 2^{i-2} , which is true for $(i-1)=1$
- **Induction step:** Each node has at most 2 children. Therefore, the maximum number of nodes at level (i) is $2^{i-2} \times 2 = 2^{i-1}$
- $\sum_{i=1}^k 2^{i-1} = 2^k - 1$



Leaf Nodes vs. Degree-2 Nodes

- Properties

- 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
→ $n_0 = n_2 + 1$

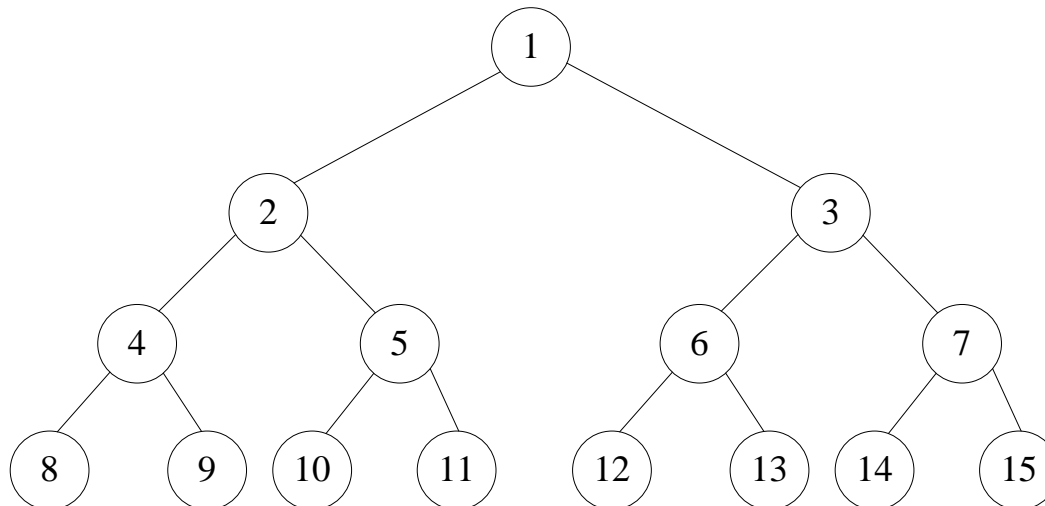
- Proof

- Let n_1 be the number of nodes of degree one and n the total number of nodes
- We have $n = n_0 + n_1 + n_2$
- Each node except the root has a branch leading into it
 - $n-1 = B$, where B is the number of branches
- $B = n_1 + 2n_2$
- $n = B+1 = n_1 + 2n_2 + 1 = n_0 + n_1 + n_2$
→ $n_0 = n_2 + 1$



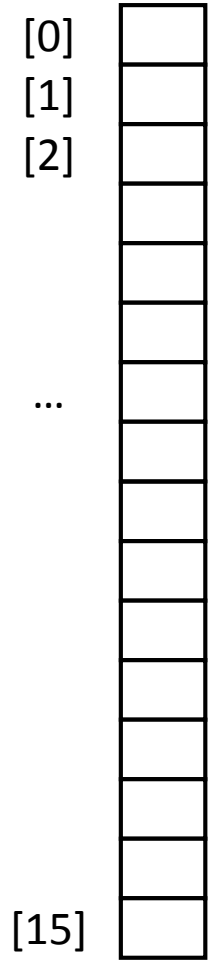
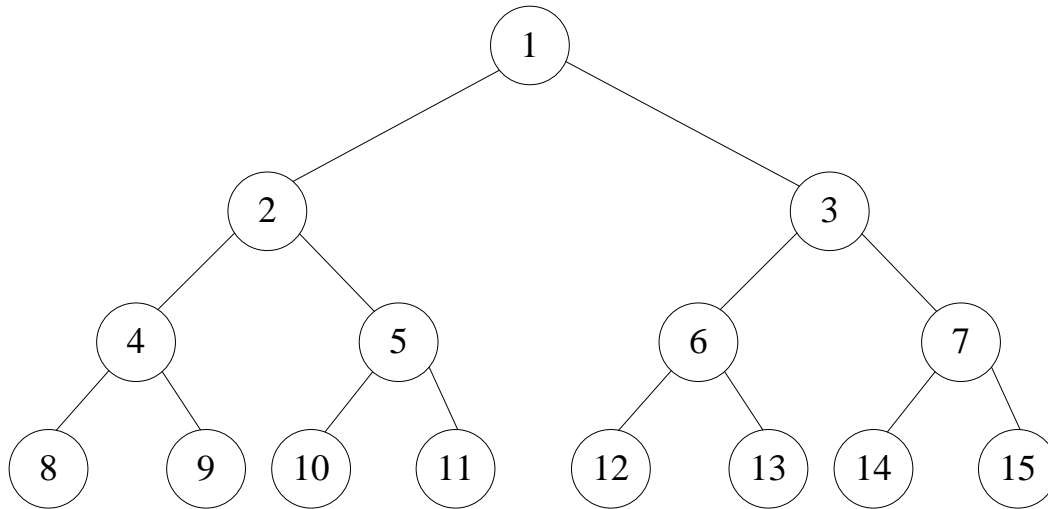
Full & Complete Binary Trees

- Definition: a **full** binary tree of depth k is a binary tree of depth k having $2^k - 1$ nodes, $k \geq 0$
- We number the nodes in the full binary tree level by level, left to right
- 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





Array Representation of a Binary Tree

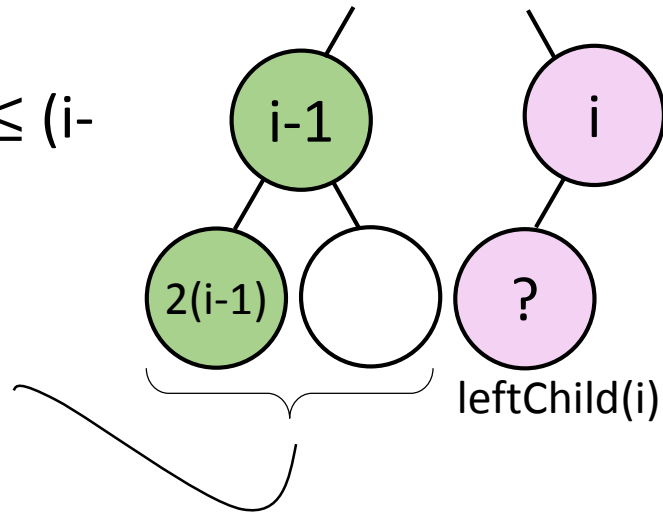


- 2^k -element array for a k -depth binary tree
 - Entry 0 is left unused
- Properties
 1. $\text{parent}(i)$ is at $\lfloor i/2 \rfloor$
 2. $\text{leftChild}(i)$ is at $2i$
 3. $\text{rightChild}(i)$ is at $2i + 1$



Array Representation of a Binary Tree

- Proof of property 2
 - Induction base: $\text{leftChild}(1)$ is at 2
 - Induction hypothesis: for all j , $1 \leq j \leq (i-1)$, $\text{leftChild}(j)$ is at $2j$
 - Induction step:
 - Two nodes immediately preceding $\text{leftChild}(i)$ are right and left children of node $i-1$
 - The left child of node $i-1$ is at $2(i-1)$ according to the hypothesis
 - The right child is thus at $2(i-1)+1 = (2i-1)$
 - $\text{leftChild}(i)$ is thus at $(2i-1)+1 = 2i$
- Property 3 is an immediate consequence of 2
- Property 1 follows from 2 and 3





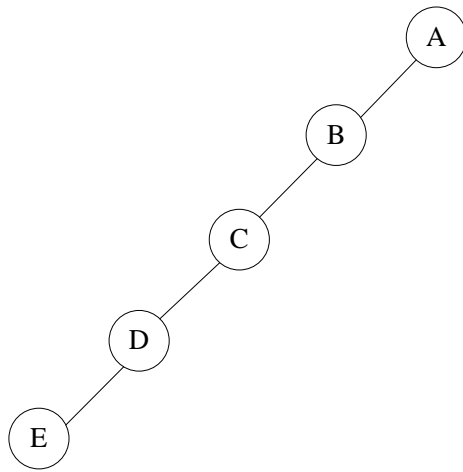
Linked Representation for a Binary Tree

- Drawbacks of the array representation
 - Good for complete/full binary tree but wasteful of space for many other trees
 - Tree manipulation can incur excessive data movement
- Linked representation tackles the problems

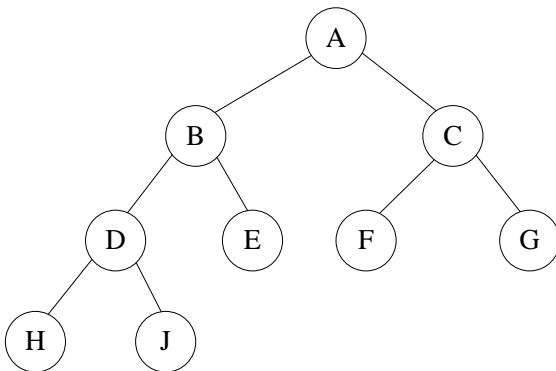
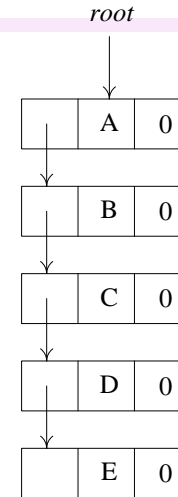
```
template <class T> class Tree;  
  
template <class T>  
class TreeNode {  
friend class Tree <T>;  
private:  
    T data;  
    TreeNode <T> *leftChild;  
    TreeNode <T> *rightChild;  
};  
  
template <class T>  
class Tree{  
public:  
    // tree operations  
    .  
private:  
    TreeNode <T> *root;  
};
```



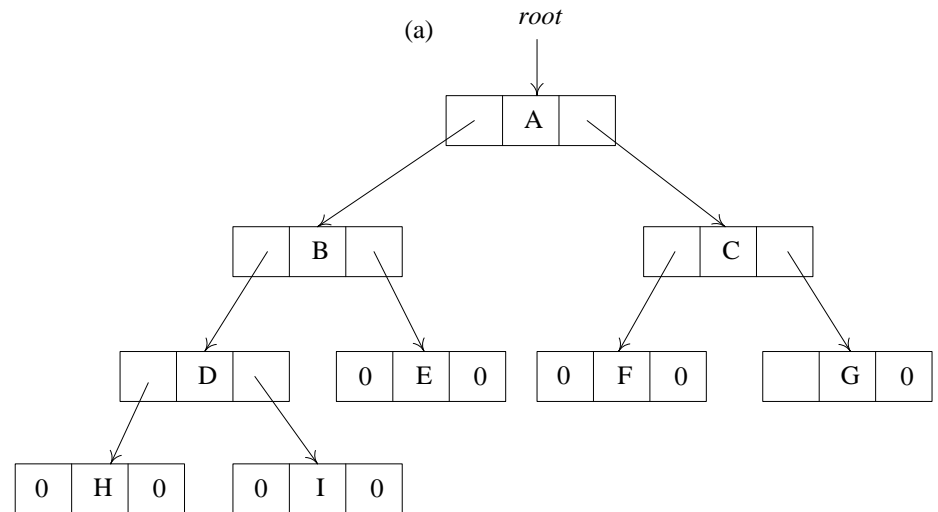
Linked Representation for a Binary Tree



(a)



(b)





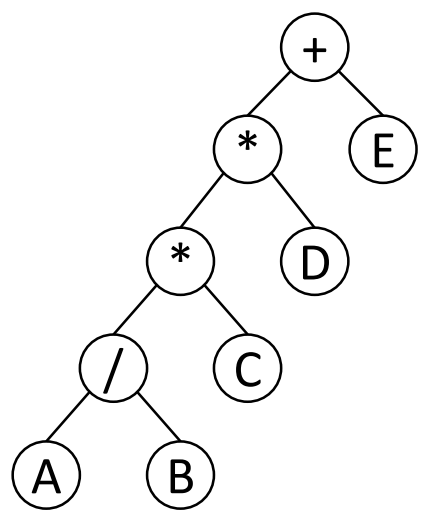
Outline

- 5.1 Introduction
- **5.2-5.5 Binary trees**
 - 5.2 Basics
 - **5.3 Traversal and iterators**
 - 5.4 Additional operations
 - 5.5 Threaded binary tree



Binary Tree Traversal

- Visiting each node in the tree exactly once
 - Operation can be performed on each visited node
- Full traversal produces a **linear order** for the nodes in a tree
 - Linearized sequence of a binary tree can be useful



Level-order traversal	+ * E * D / C A B
Preorder traversal	+ * * / A B C D E
Inorder traversal	A / B * C * D + E (infix)
Postorder traversal	A B / C * D * E + (postfix)

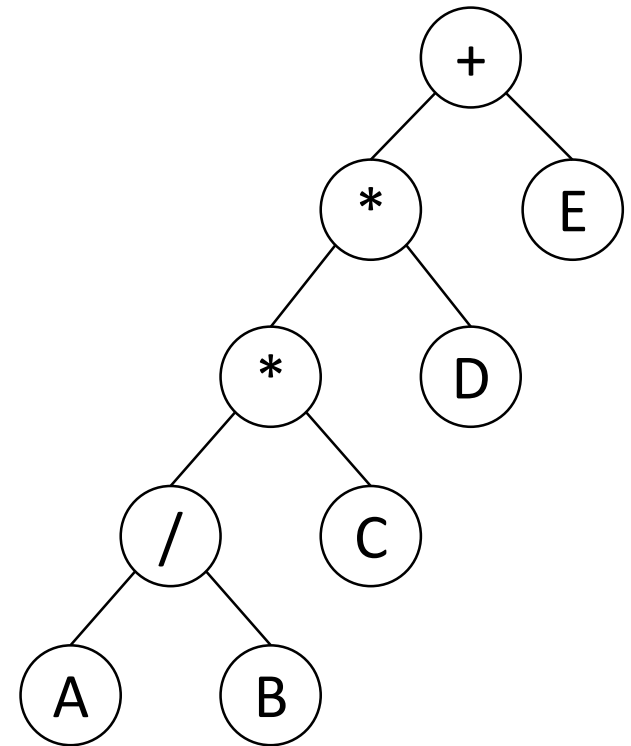


Level-Order Traversal

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



+ * E * D / C A B



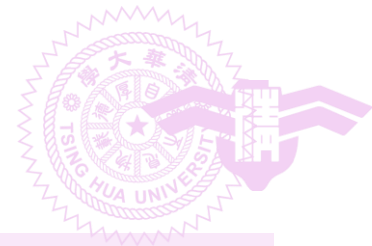
Preorder, Inorder, and Postorder

- Let us define
 - L moving left
 - V visiting the node
 - R moving right
- Possible combinations of traversal
 - Various positions of the V with respect to the L and R

Preorder	VLR
Inorder (V is in between L and R)	LVR
Postorder	LRV

Mirror L & R
VRL
RVL
RLV

mirrored orders are less widely used

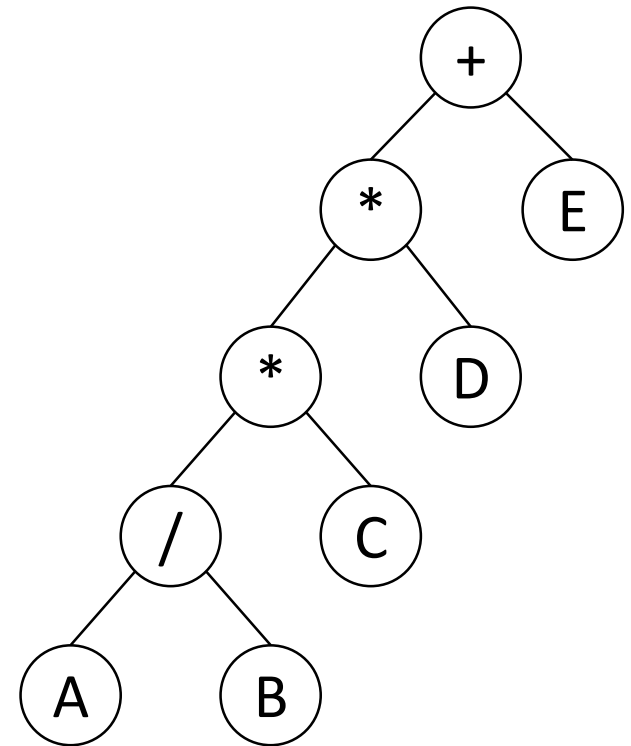


Preorder (VLR)

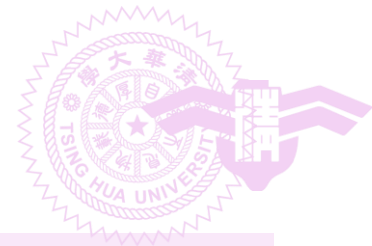
```
template <class T>
void Tree<T>::Preorder()
{
    Tpr(root); //Traverse preorderly
}
```

```
template <class T>
void Tree<T>::Tpr(TreeNode<T> * p)
{
    // this is a recursive function
    visit(p); // **V**
    Tpr(p->leftChild); // L
    Tpr(p->rightChild); // R
}
```

```
template <class T>
void Tree<T>::visit(TreeNode<T> * p)
{
    cout << p->data;
}
```



+ ** / ABCDE

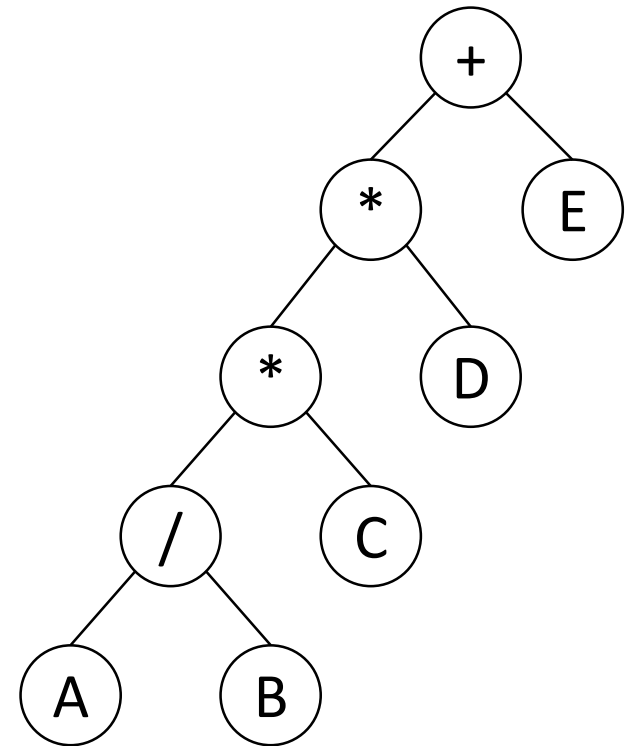


Inorder (LVR)

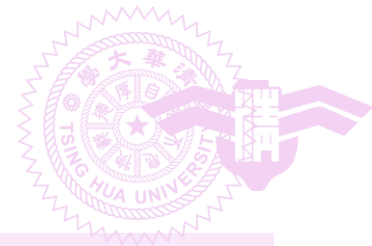
```
template <class T>
void Tree<T>::Inorder()
{
    Tin(root); //Traverse inorderly
}
```

```
template <class T>
void Tree<T>::Tin(TreeNode<T> * p)
{
    // this is a recursive function
    Tin(p->leftChild); // L
    visit(p); // **V**
    Tin(p->rightChild); // R
}
```

```
template <class T>
void Tree<T>::visit(TreeNode<T> * p)
{
    cout << p->data;
}
```



A / B * C * D + E

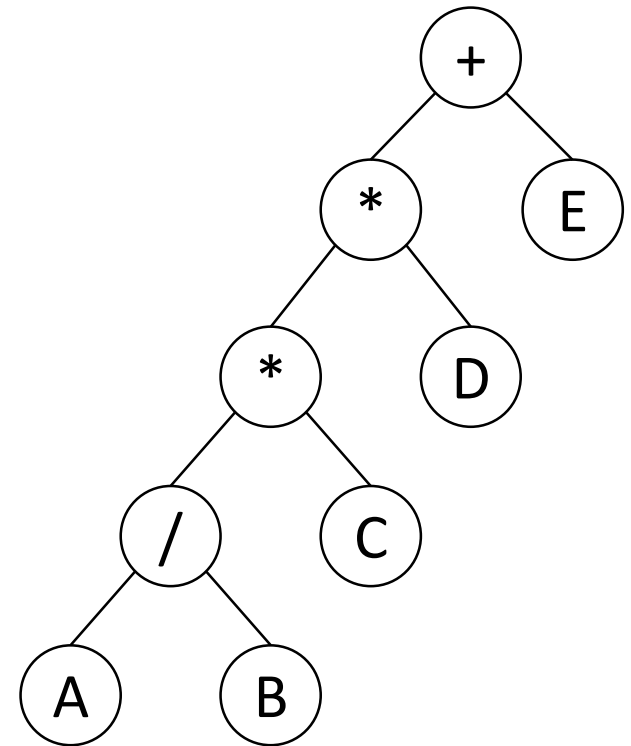


Postorder (LRV)

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

```
template <class T>
void Tree<T>::Tpo(TreeNode<T> * p)
{
    // this is a recursive function
    Tpo(p->leftChild); // L
    Tpo(p->rightChild); // R
    visit(p); // **V**
}
```

```
template <class T>
void Tree<T>::visit(TreeNode<T> * p)
{
    cout << p->data;
}
```

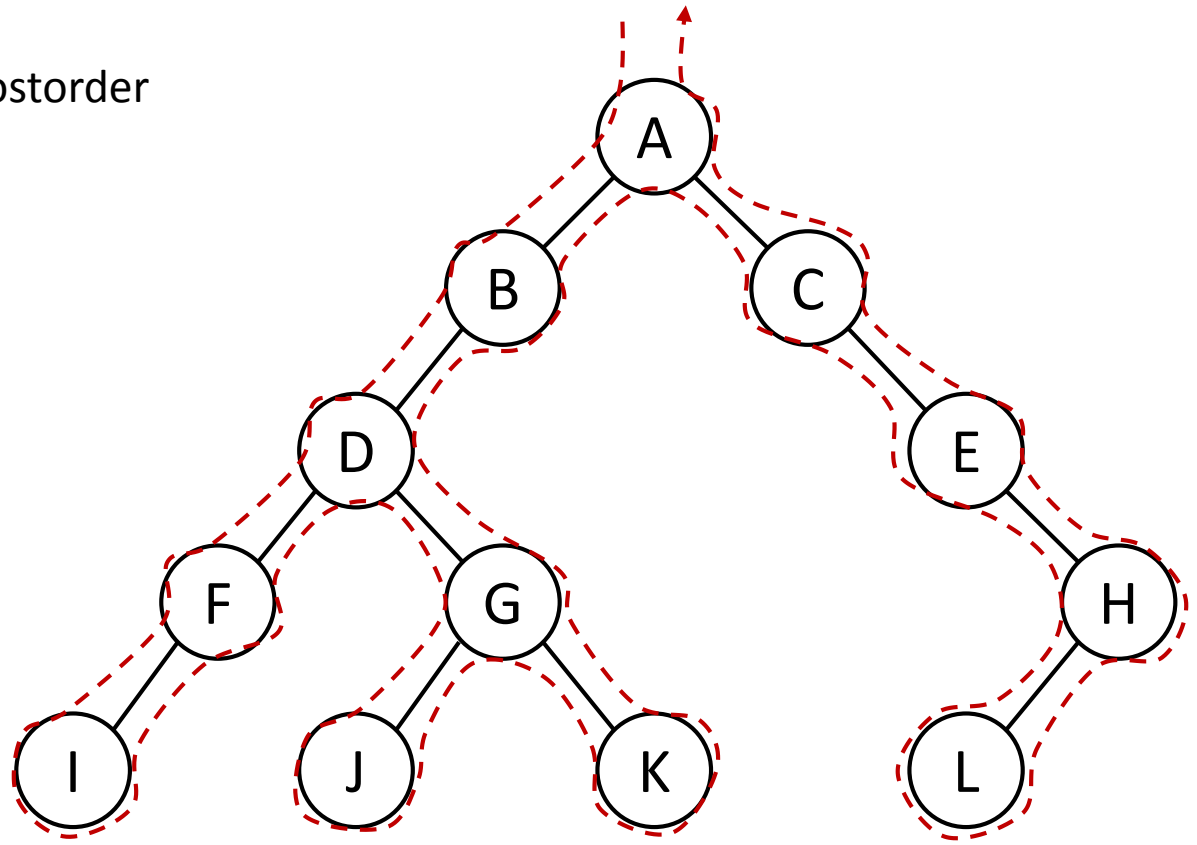
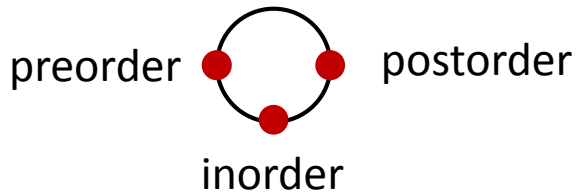


AB/C * D * E +



Tips for Preorder, Inorder, & Postorder

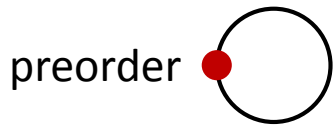
- Attach a point to each node
- Draw the contour of the tree



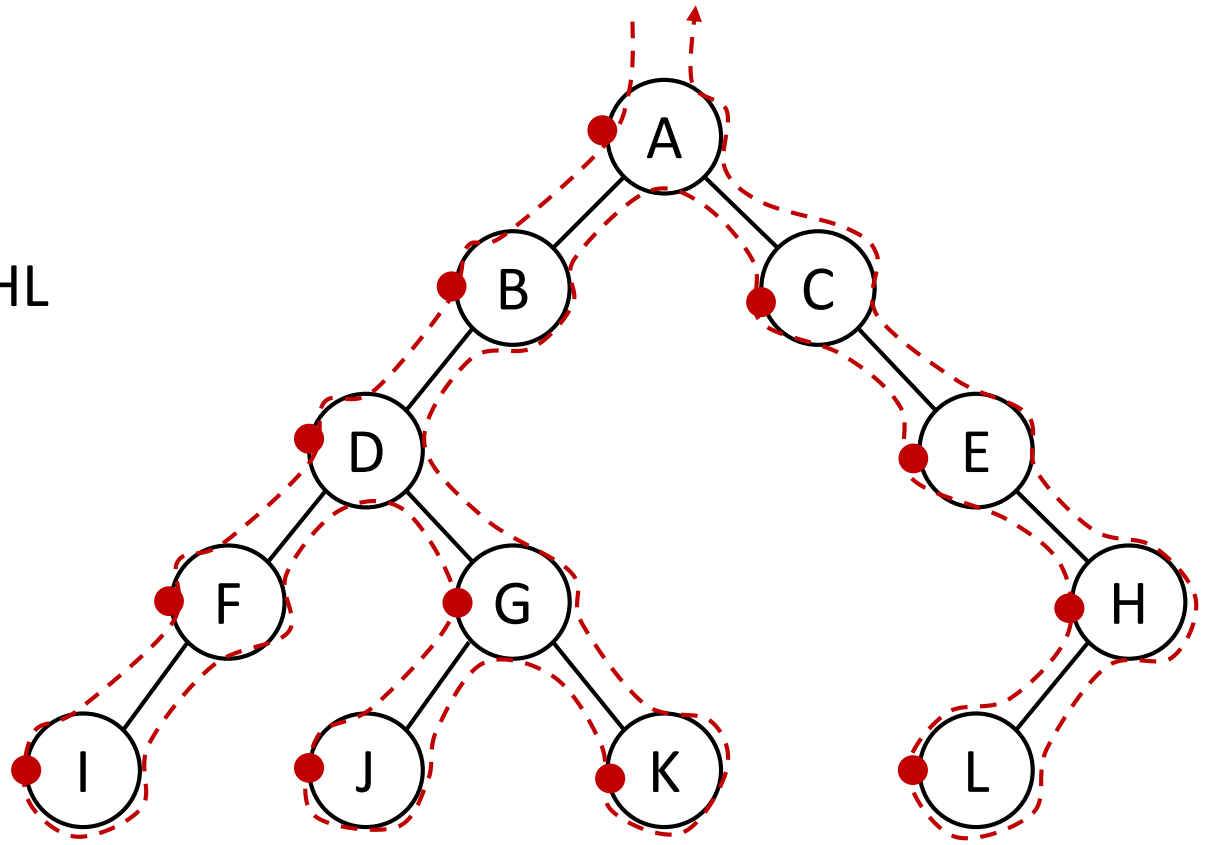


Tips for Preorder, Inorder, & Postorder

- Attach a point to each node
- Draw the contour of the tree



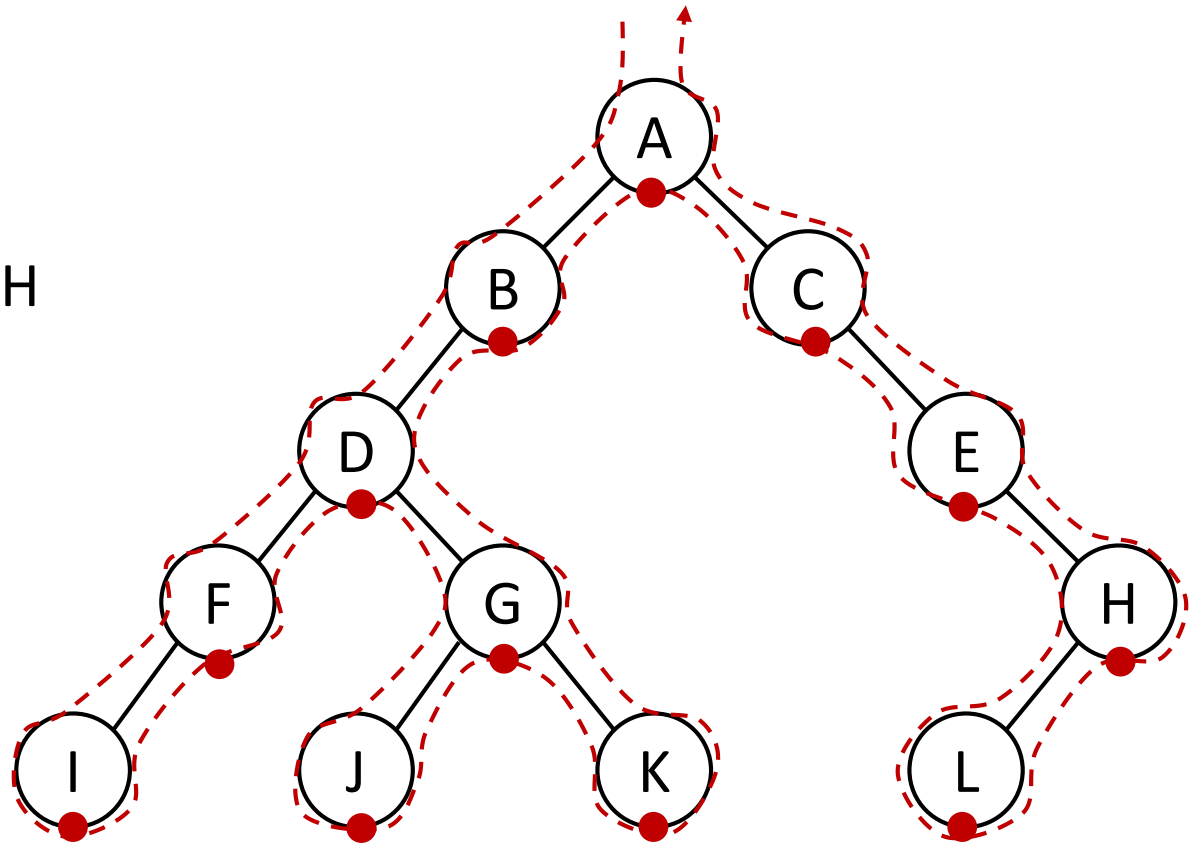
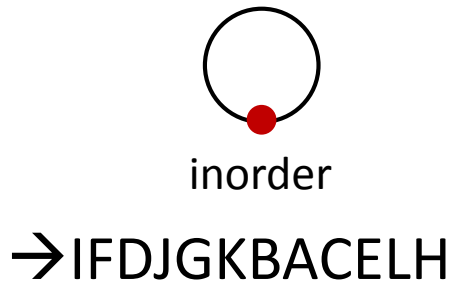
→ ABDFIGJKCEHL





Tips for Preorder, Inorder, & Postorder

- Attach a point to each node
- Draw the contour of the tree



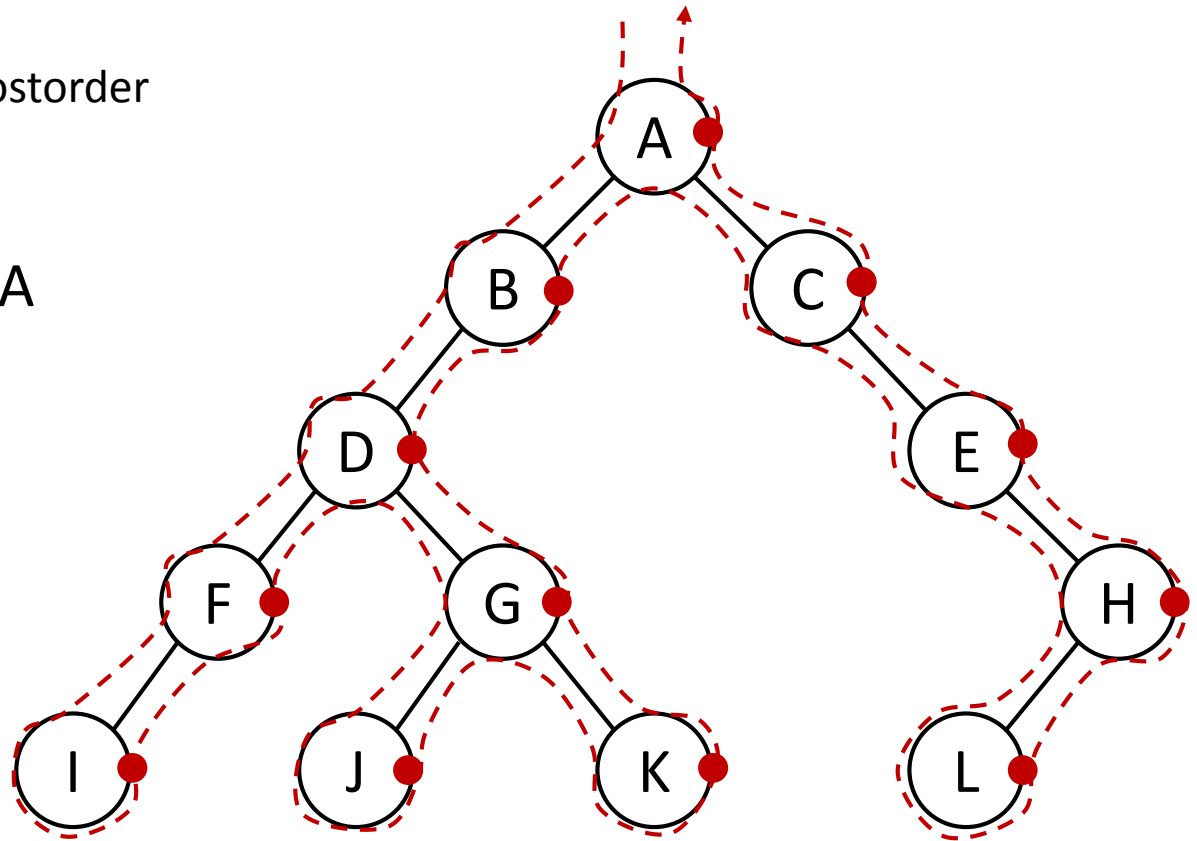


Tips for Preorder, Inorder, & Postorder

- Attach a point to each node
- Draw the contour of the tree



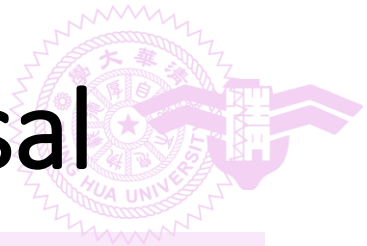
→ IFJKGDBLHECA





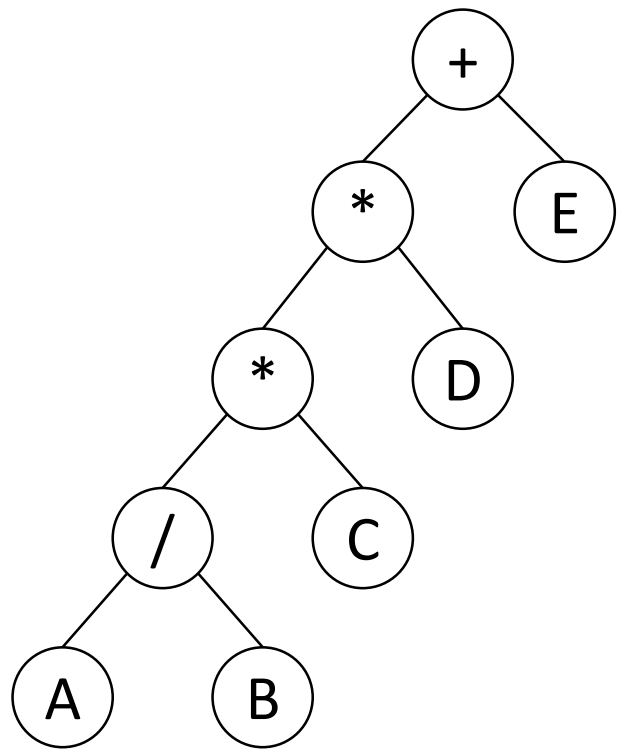
Non-Recursive Traversal Algorithm

- Binary tree is a type of container
- We thus want to implement an **iterator** for a binary tree
 - We need a non-recursive traversal algorithm
 - Such an iterator **USES-A** stack
- Definition:
 - Type X data object **USES-A** data object of Type Y means that
 - Type X object employs the Type Y object **in its member function** to perform a task

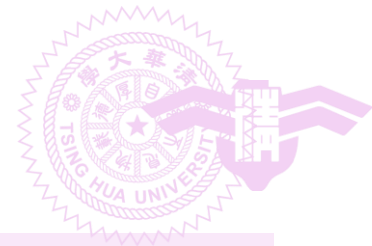


Non-Recursive Inorder Traversal

```
template <class T>
void Tree <T>::NonrecInorder()
{
    Stack <TreeNode<T>*> s;
    TreeNode <T> *currentNode = root;
    while(1) {
        while (currentNode) {
            s.Push(currentNode);
            currentNode = currentNode->leftChild;
        }
        if (s.IsEmpty())
            return;
        currentNode = s.Top();
        s.Pop();
        Visit(currentNode);
        currentNode = currentNode->rightChild;
    }
}
```



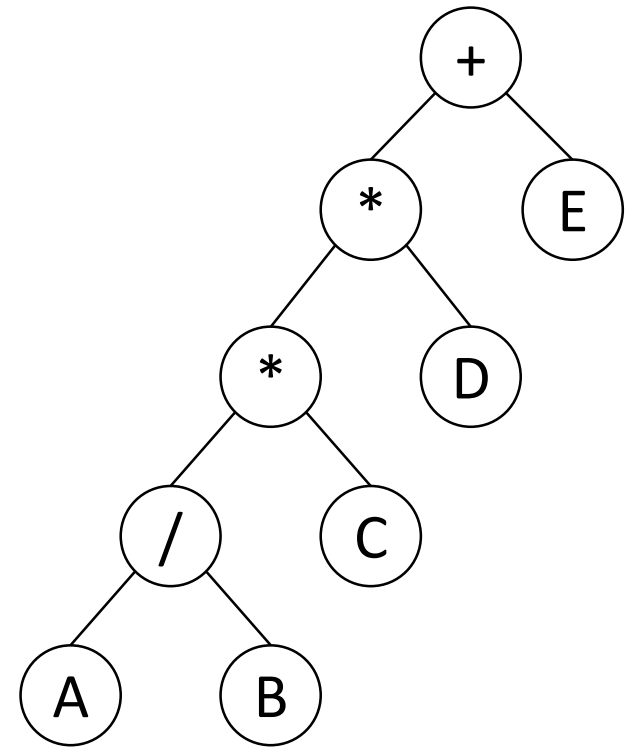
A / B * C * D + E



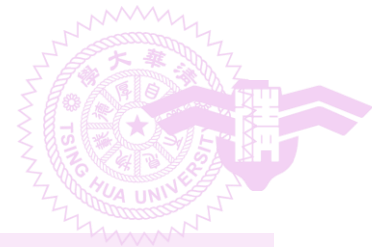
Inorder Iterator

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

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



A / B * C * D + E



Outline

- 5.1 Introduction
- **5.2-5.5 Binary trees**
 - 5.2 Basics
 - 5.3 Traversal and iterators
 - **5.4 Additional operations**
 - **5.4.1 Copying binary trees**
 - **5.4.2 Testing equality**
 - (5.4.3 The satisfiability problem)



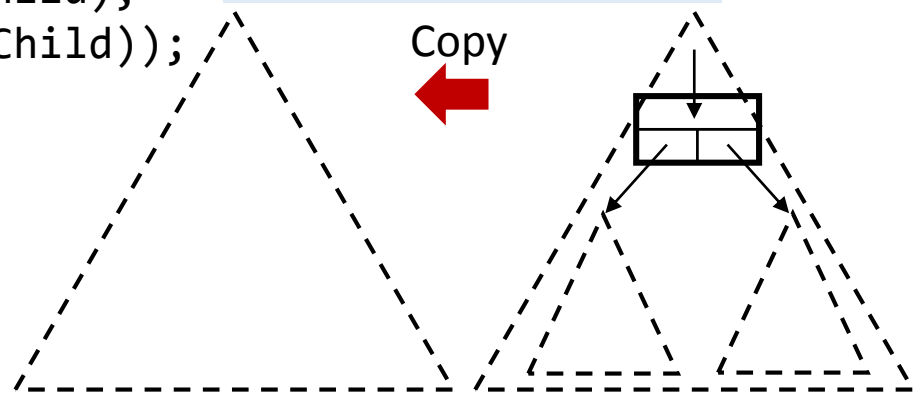
Copying Binary Trees

```
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> * p) // Workhorse
{ // Return a pointer to an exact copy of the tree rooted at p
    if (!p) // a null pointer
        return 0;

    return new TreeNode<T>(p->data,
                           Copy(p->leftChild),
                           Copy(p->rightChild));
}
```

Coping a tree =
+ copying the root node
+ copying the left tree
+ copying the right tree





Copying Binary Trees (Exemplifying Pseudo Code)

```
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> * p) // Workhorse  
{
```

if p points to an empty tree
return 0;

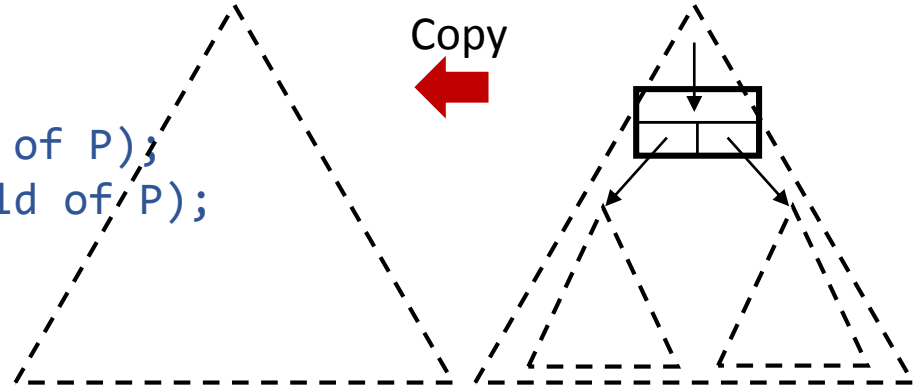
Let P be the node pointed by p;
Create a node N;

data of N = data of P;
leftChild of N = Copy(leftChild of P);
rightChild of N = Copy(rightChild of P);

return the address of N;

```
}
```

Copying a tree =
+ copying the root node
+ copying the left tree
+ copying the right tree





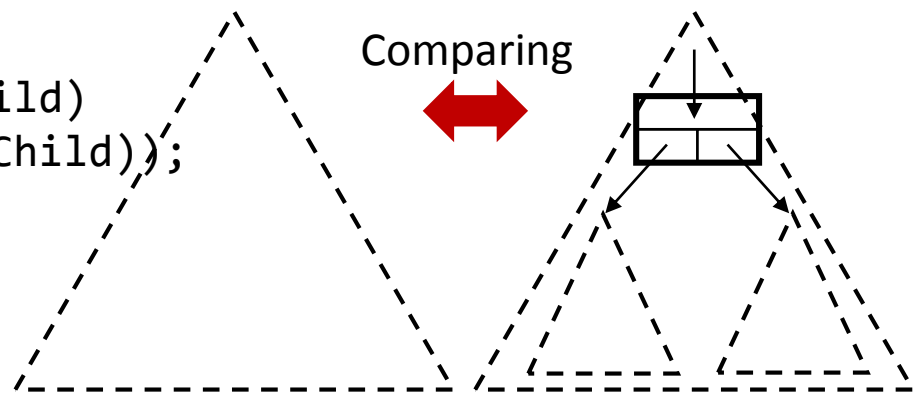
Testing Equality of Binary Trees

```
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) // workhorse
{
    if ((!a) && (!b)) // two empty trees
        return true;
    else if ((!a) || (!b))
        return false;

    return ((a->data == b->data)
    && Equal(a->leftChild,b->leftChild)
    && Equal(a->rightChild,b->rightChild));
}
```

Comparing two trees =
+ comparing two root data
+ comparing the two left trees
+ comparing the two right trees





Testing Equality of Binary Trees (Exemplifying Pseudo Code)

```
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) // workhorse
{
    if two trees are both empty
        return true;
    else if only one tree is empty
        return false;

    if the root nodes of the two trees contain the same data
    && the left subtrees of the two trees are equal
    && the right subtrees of the two trees are equal
        return true;
    else
        return false;
}
```

Comparing two trees =
comparing two root data
+ comparing the two left trees
+ comparing the two right trees



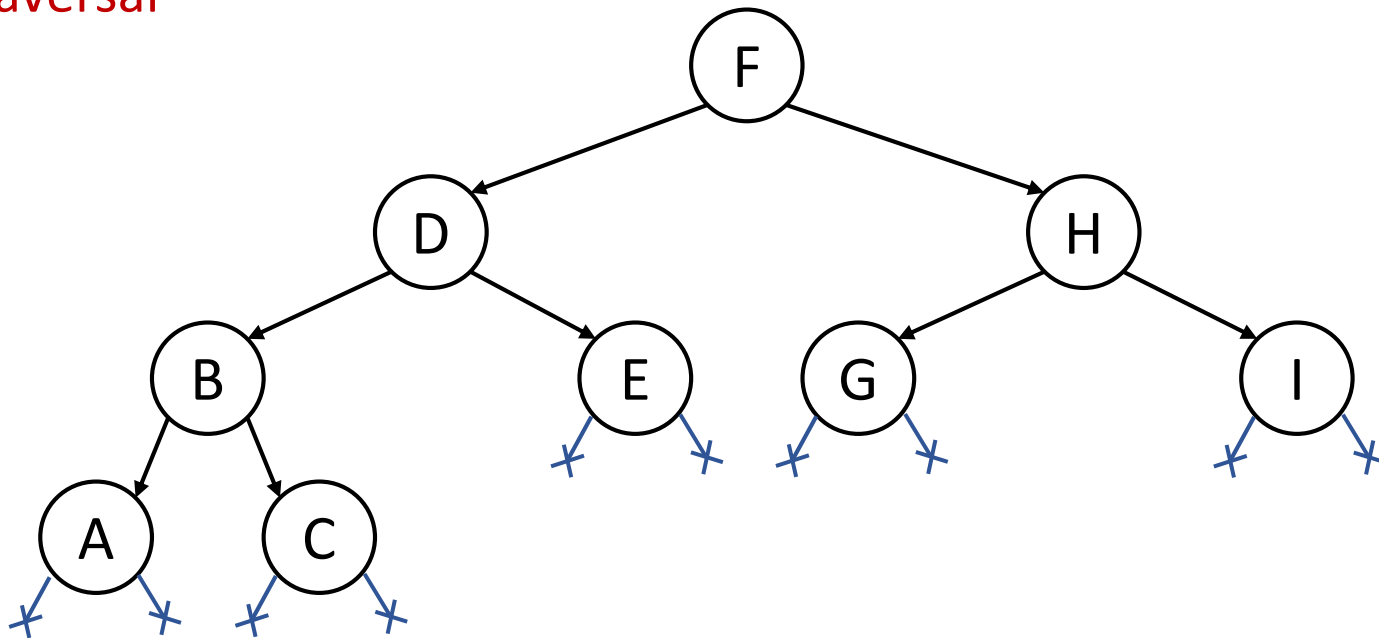
Outline

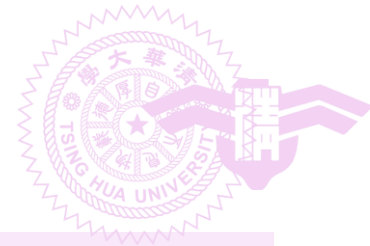
- 5.1 Introduction
- **5.2-5.5 Binary trees**
 - 5.2 Basics
 - 5.3 Traversal and iterators
 - 5.4 Additional operations
 - **5.5 Threaded binary trees**



Motivation

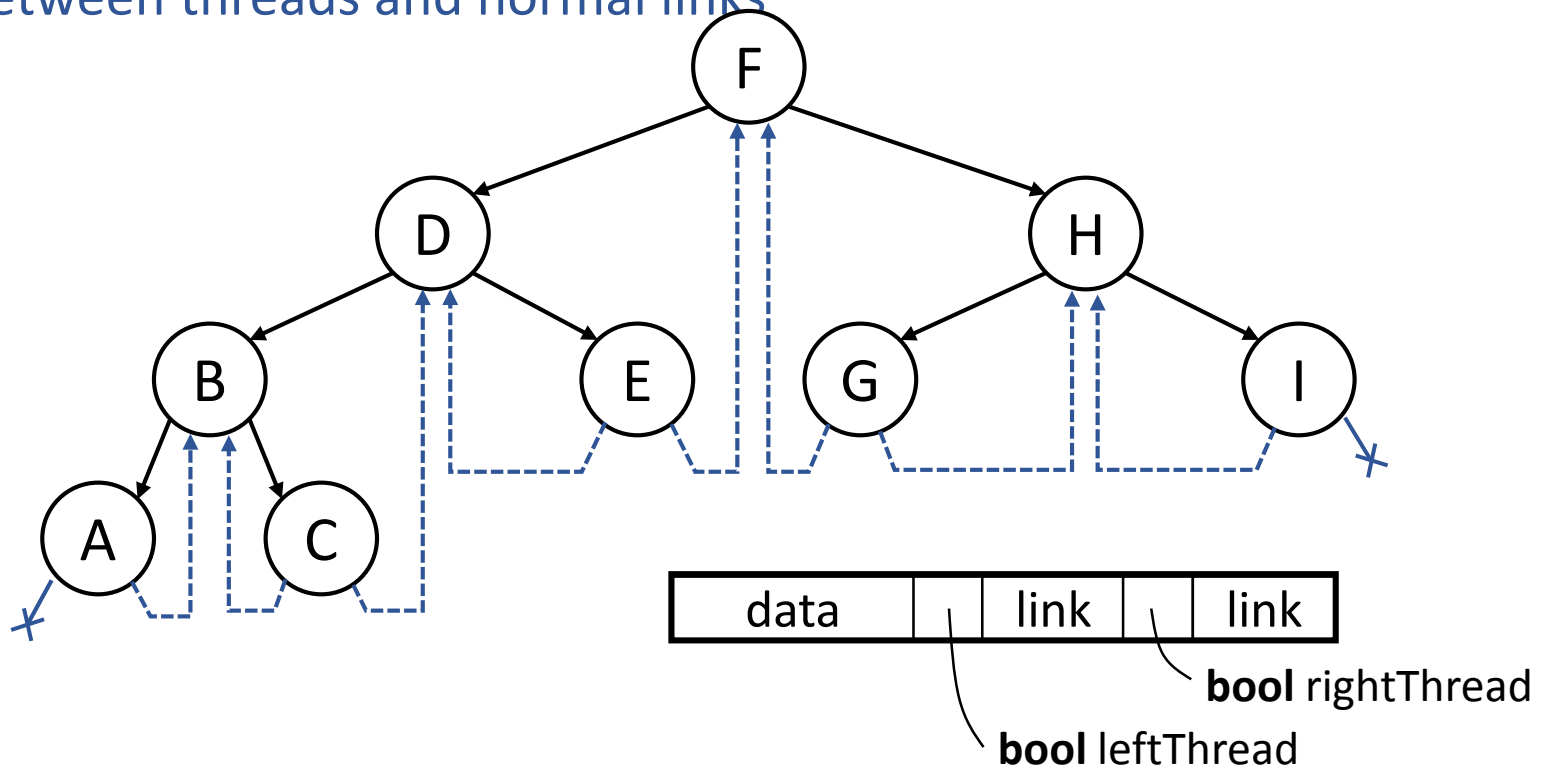
- More than **half** of the link fields of a binary tree store **0s**
 - n nodes contain a total of **$2n$ link fields**
 - n nodes are pointed by a total of **$n-1$ non-zero links**
- We want to make use of these 0 links to assist **inorder tree traversal**





Threads

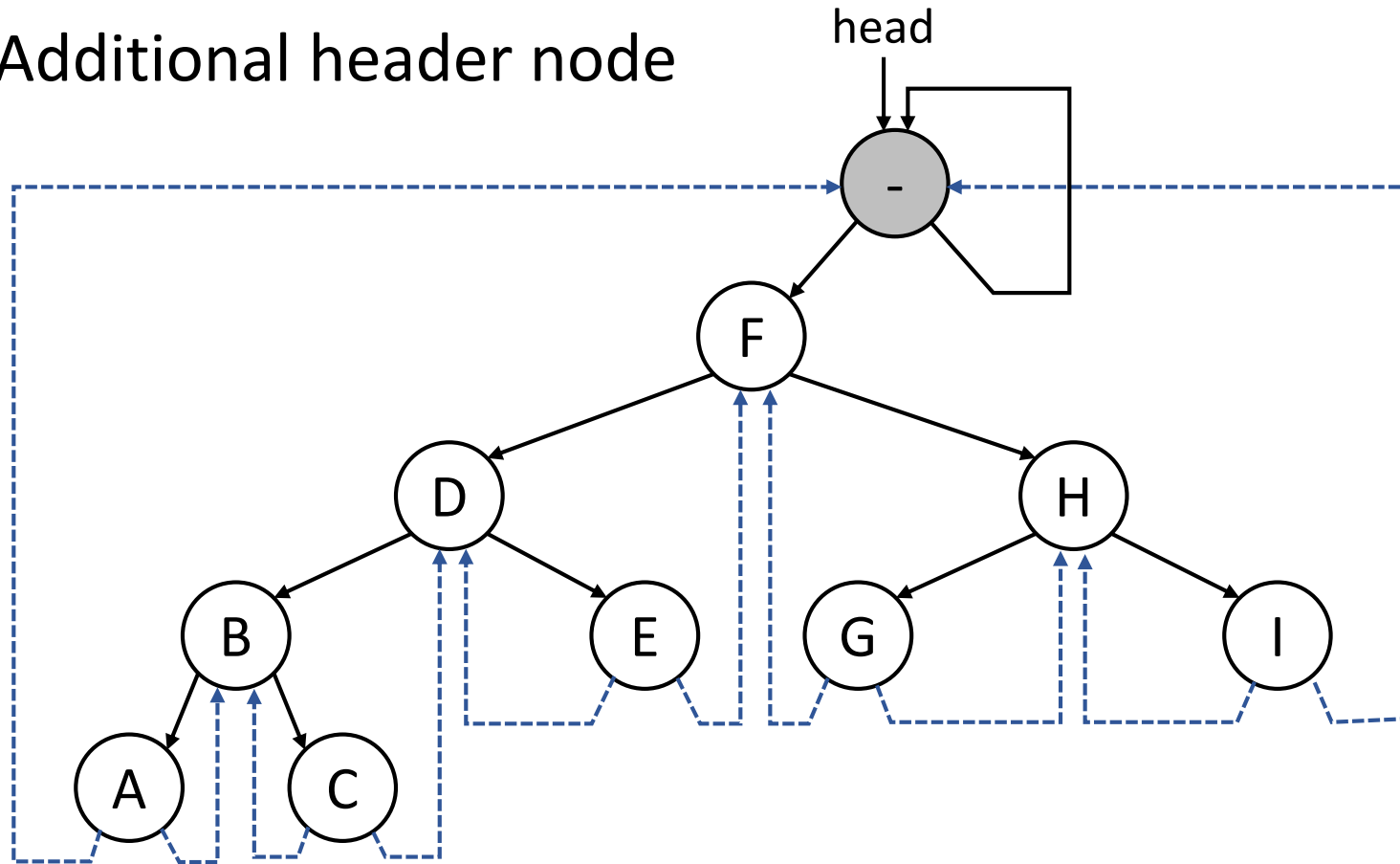
- Threads: **links for inorder traversal**
 - Replace a 0 leftChild link with a link to the **inorder predecessor**
 - Replace a 0 rightChild link with a link to the **inorder successor**
- Each node needs two **additional Boolean fields** to distinguish between threads and normal links





Threaded Binary Trees

- Additional header node

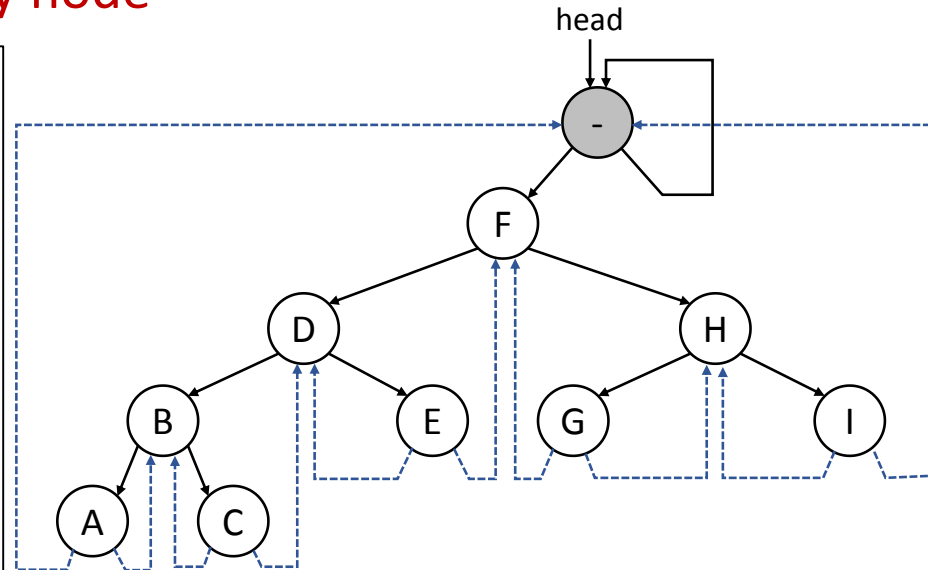




Traverse a Threaded Binary Tree

- Thread binary tree traversal
 - No need of stacks
 - Can begin from any arbitrary node

```
T* ThreadedInorderIterator::Next()
{
  ThreadedNode <T> *temp
    =currentNode->rightChild;
  if(currentNode->rightThread){
    currentNode = temp;
  }else{
    while (!temp->leftThread)
      temp = temp->leftChild;
    currentNode = temp;
  }
  if(currentNode == head)
    return 0;
  else
    return &currentNode->data;
}
```

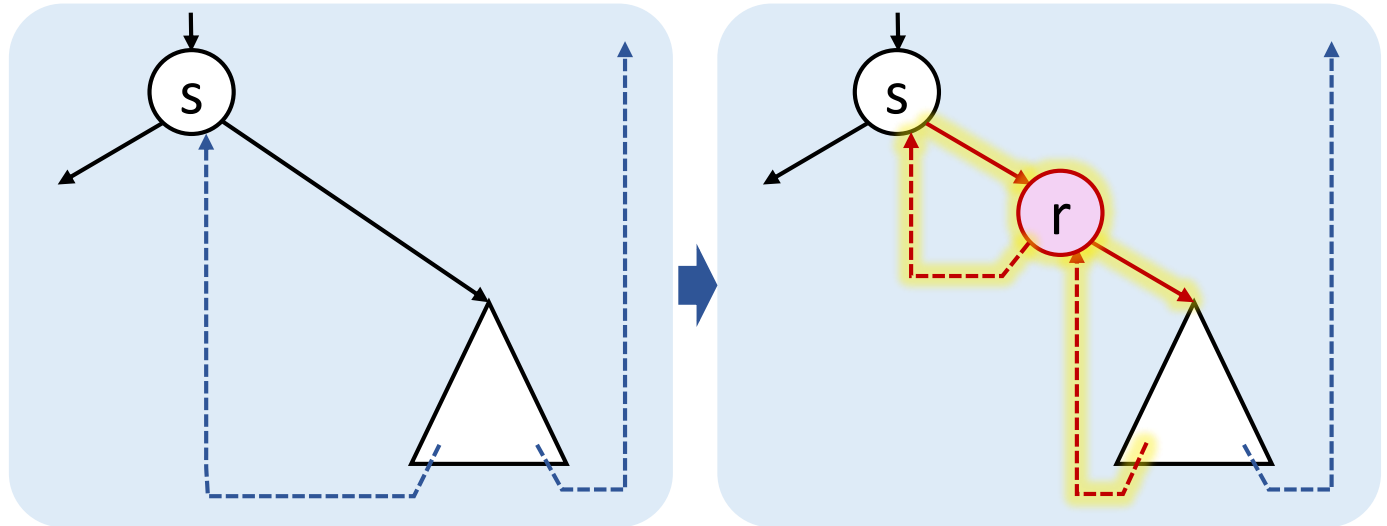




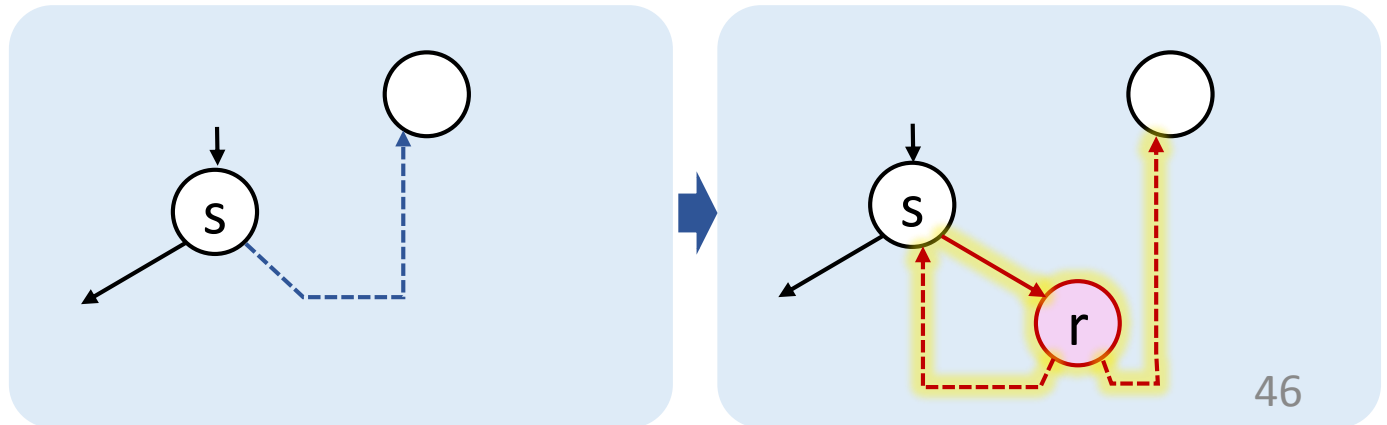
Inserting a Node into a Treaded Tree

- Insert a node as the right child

Case 1
right subtree
exists



Case 2
right subtree
doesn't exist





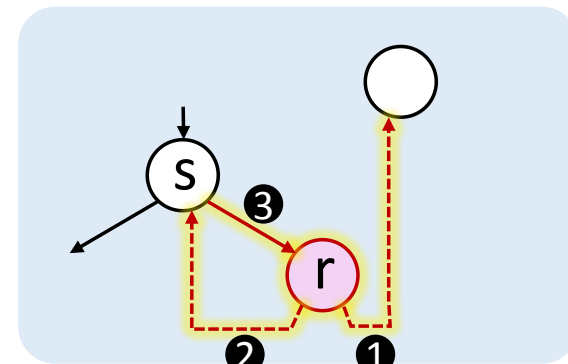
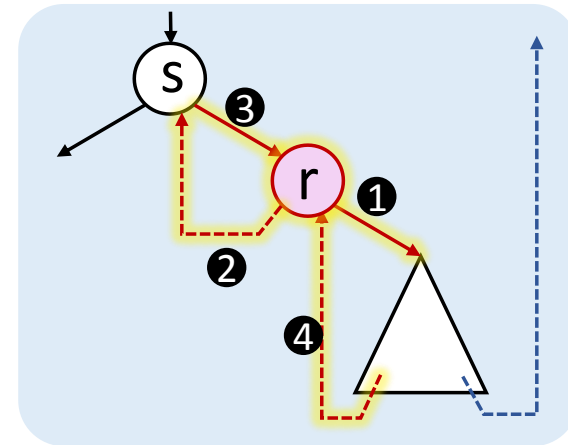
Inserting a Node into a Treaded Tree

```
template <class T>
void ThreadedTree<T>::InsertRight(ThreadedNode <T> *s,
                                  ThreadedNode <T> *r)
{
  // insert r as the right child of s
  r->rightChild = s->rightChild;      ❶
  r->rightThread = s->rightThread;

  r->leftChild = s;                    ❷
  r->leftThread = true; //leftChild is a thread

  s->rightChild = r;                   ❸
  s->rightThread = false;

  if (! r->rightThread) {              ❹
    ThreadedNode <T> *temp = InorderSucc(r);
    temp->leftChild = r;
  }
}
```

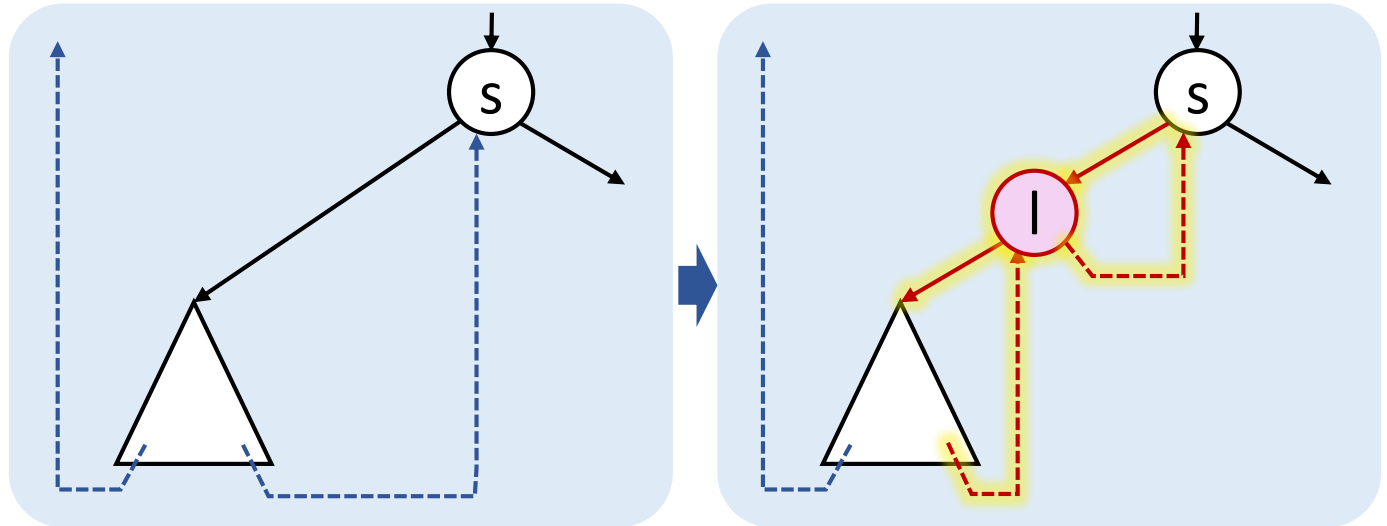




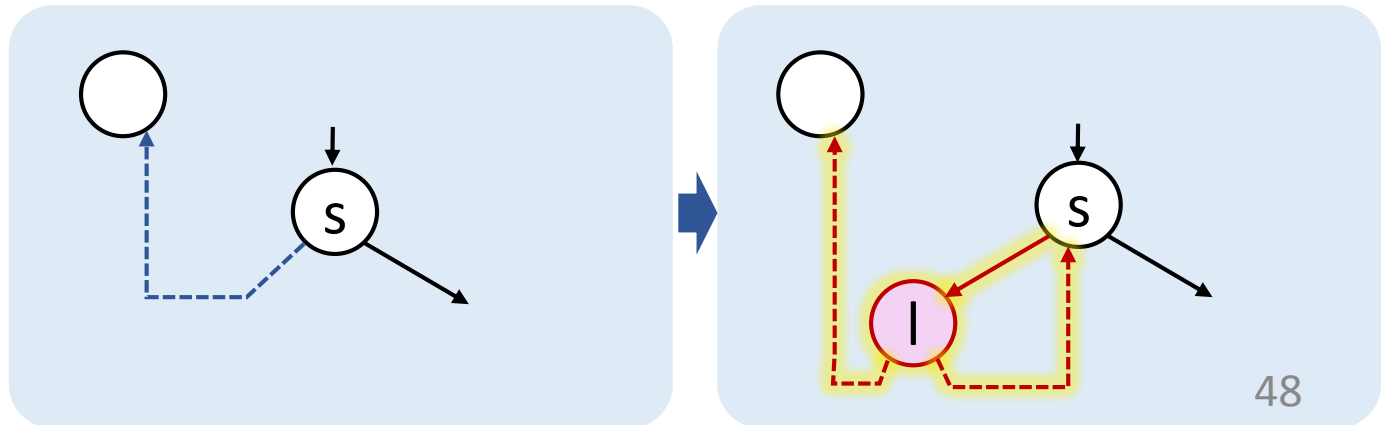
Inserting a Node into a Treaded Tree

- Insert a node as the left child (similar to the right case)

Case 1
left subtree
exists



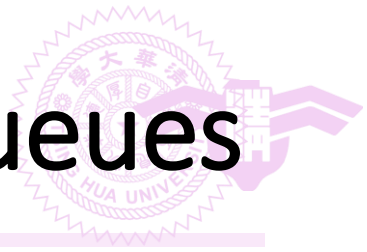
Case 2
left subtree
doesn't exist





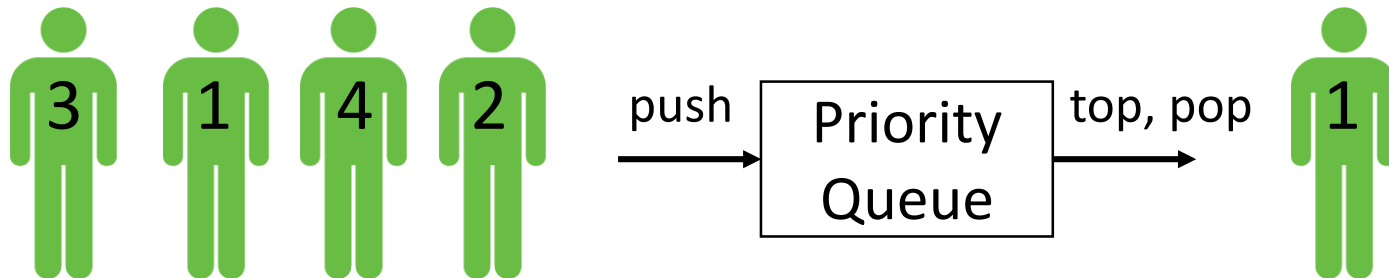
Outline

- 5.1 Introduction
- 5.2-5.5 Binary trees
- **5.6 Heaps → priority queues**
- 5.7 Binary search trees
- 5.8 Selection trees
- 5.9 Forests
- (5.10 Disjoint sets)
- (5.11 Counting binary trees)



Normal Queues vs. Priority Queues

- Normal queues
 - First in first out
 - Also known as first come first serve
- Priority queues
 - High priority first



- 不是只有heap才能實現priority queue，但heap是整體比較有效率的 data structure



Priority Queues

- Max priority queue ADT

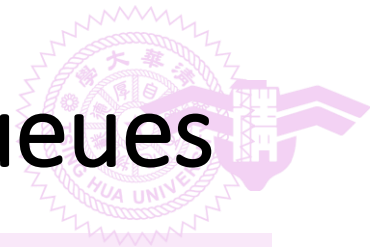
```
template <class T>
class MaxPQ {
public:
    virtual ~MaxPQ() {} // virtual destructor
    virtual bool IsEmpty() const = 0; //return true iff empty
    virtual const T& Top() const = 0; //return reference to the max
    virtual void Push(const T&) = 0;
    virtual void Pop() = 0;
};
```

"= 0" is a notation indicating **pure virtual** functions. It does not mean an assignment.

Exemplifying Use of Priority Queues



- Shortest-job-first task scheduling
 - Maintain a priority queue of all pending tasks
 - Task with the smallest time requirement is performed
- Simulating a system, e.g., a processor
 - Maintain a priority queue of concurrent events
 - E.g., multiple instructions are concurrently executed in a processor
 - Event occurs at the least time is selected to show the instruction execution sequences



Unordered List-Based Priority Queues

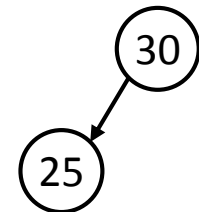
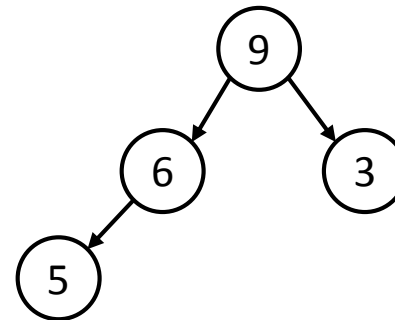
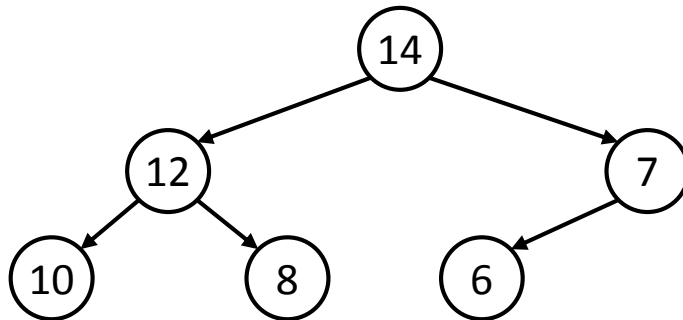
- Concept
 - Use an unordered list (array or chain) to store all elements
 - Scan the entire list for determining the max priority element
- Time complexity

	Unordered List
IsEmpty()	$O(1)$
Top()	$O(n)$
Push()	$O(1)$
Pop()	$O(n)$



Heap-Based Priority Queue

- Definition
 - **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)
 - **Max (min) heap** is a complete binary max (min) tree

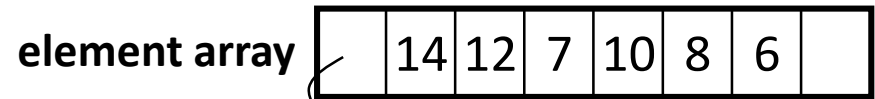
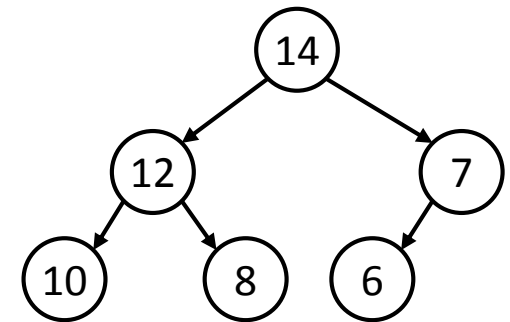




Max Heap

- Publicly derived from MaxPQ

```
template <class T>
class MaxHeap : public class MaxPQ<T> {
public:
    MaxHeap(int theCapacity = 10); // constructor
    bool IsEmpty() const; //return true iff empty
    const T& Top() const; //return reference to the max
    void Push(const T&);
    void Pop();
private:
    T* heap; // element array
    int heapSize; // number of elements in heap
    int capacity; // size of the element array
};
```

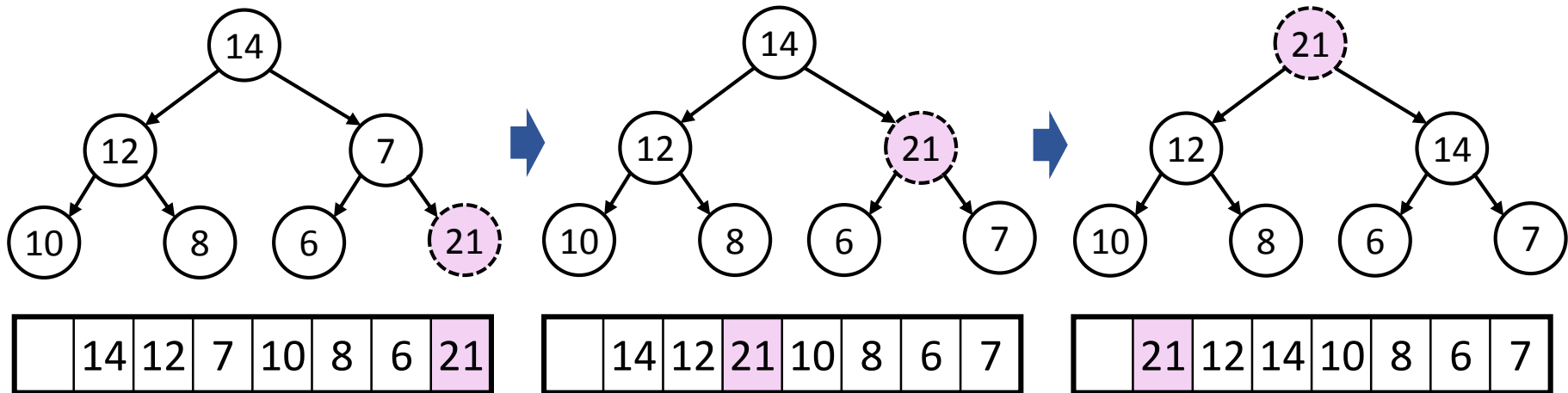


heap[0] is left unused



Insertion into a Max Heap

- Process
 - Enlarging the element array (if necessary)
 - Appending the element to the end
 - **Bubbling up** (if necessary) to maintain the max heap property
- Time complexity = $O(\log(n))$

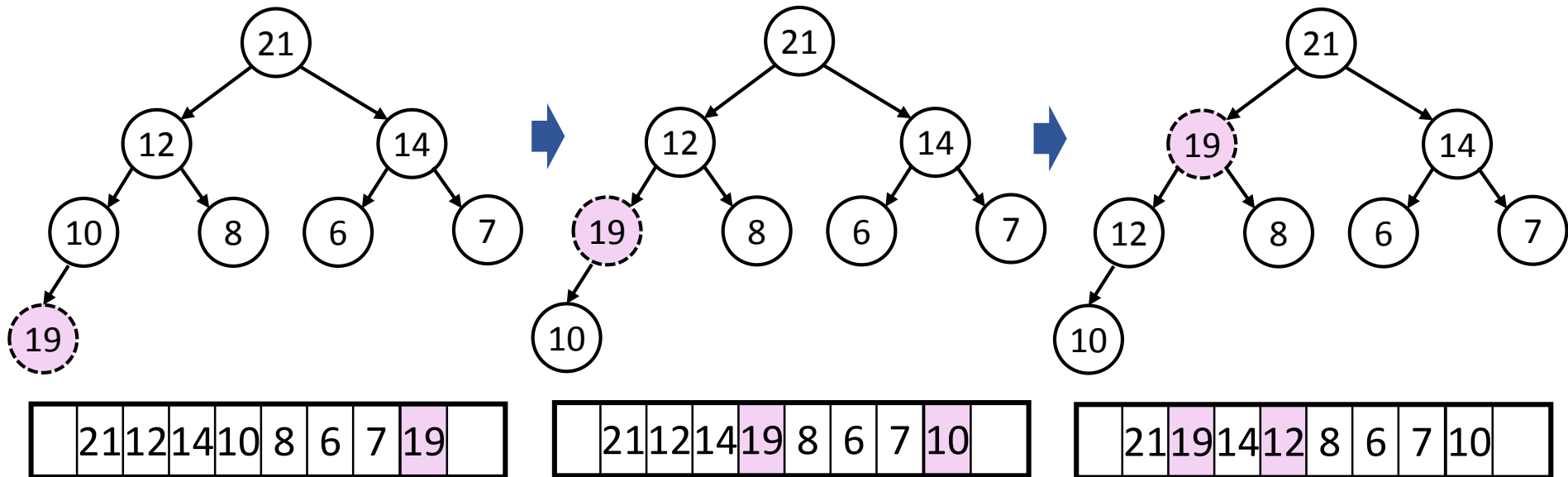


$7/2=3...1 \rightarrow$ "21" 的 parent 在位置3("7")



Insertion into a Max Heap

- Process
 - Enlarging the element array (if necessary)
 - Appending the element to the end
 - **Bubbling up** (if necessary) to maintain the max heap property
- Time complexity = $O(\log(n))$





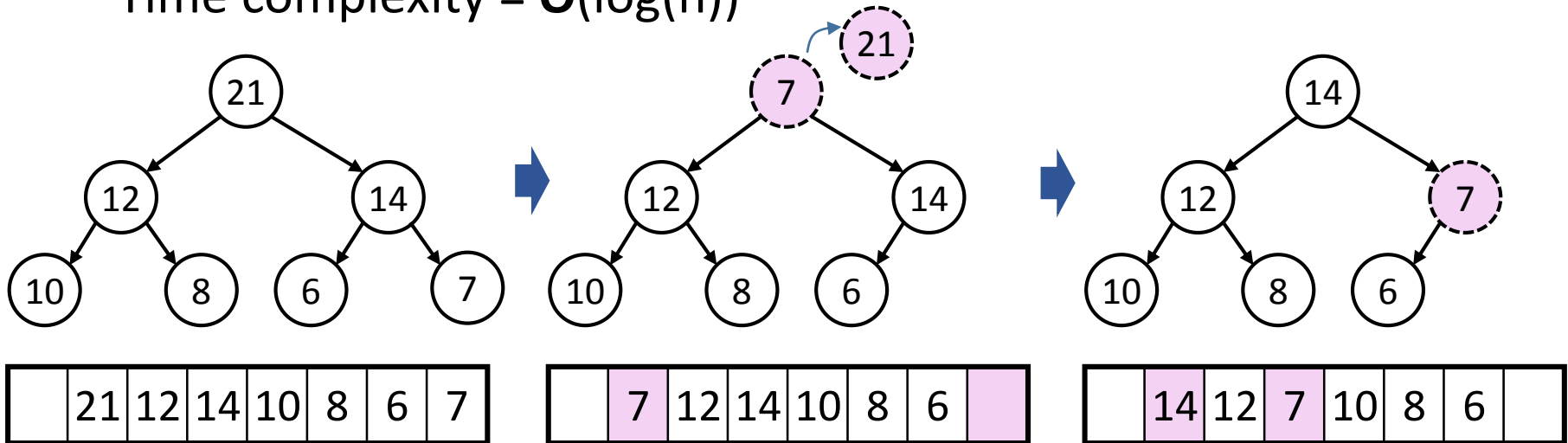
Insertion into a Max Heap

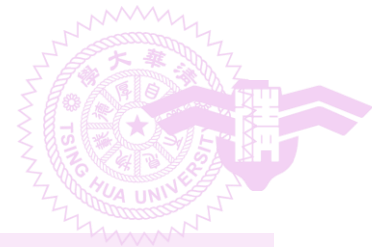
```
Template <class T>
void MaxHeap<T>::Push(const T& e)
{ // add element e to max heap
  if (heapSize == capacity) { // double the capacity
    ChangeSize 1D(heap, capacity, 2*capacity);
    capacity *= 2;
  }
  int currentNode = ++heapSize;
  while (currentNode != 1 && heap[currentNode/2] < e)
  {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



Deletion from a Max Heap

- Process
 - Max Heap deletion always occurs at **the root**
 - Move the **last array element** to **the root** position
 - **Trickling down** (if necessary) to maintain the max heap property
- Time complexity = $O(\log(n))$





Deletion from a Max Heap

```
Template <class T>
void MaxHeap<T>::Pop()
{ // Delete max element
  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 to find a position to place the last element
  int currentNode = 1; // root
  int child = 2; // a child of current node
  while (child <= heapSize){
    // set child to larger child of currentNode
    if (child < heapSize && heap[child] < heap[child + 1]) child++;

    if (lastE >= heap[child]) break;
    heap[currentNode] = heap[child];
    currentNode = child; child *= 2;
  }
  heap[currentNode] = lastE;
}
```



Outline

- 5.1 Introduction
- 5.2-5.5 Binary trees
- 5.6 Heaps
- **5.7 Binary search trees → Dictionary**
- 5.8 Selection trees
- 5.9 Forests
- (5.10 Disjoint sets)
- (5.11 Counting binary trees)



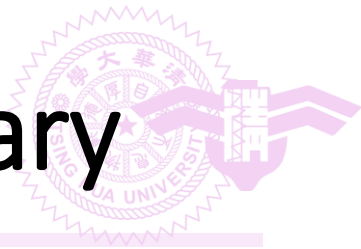
Dictionary

- Definition: A dictionary is
 - A collection of pairs: **key** and associated **element (also knowns value)**
 - No two pairs have the same key
- Operations
 - Test if a dictionary is empty
 - Get the pointer to the pair with a specified key
 - Insert a pair of key and element
 - If key is a duplicate, update the associated element
 - Delete a pair with a specified key
- ADT

```
template <class K, class E>
class Dictionary {
public:
    virtual bool IsEmptay() const = 0;
    virtual pair <K, E>* Get(const K&) const = 0;
    virtual void Insert(const pair <K, E>&) = 0;
    virtual void Delete(const K&) = 0;
};
```

"= 0" is a notation indicating **pure virtual** functions. It does not mean an assignment.

Unordered List-Based Dictionary



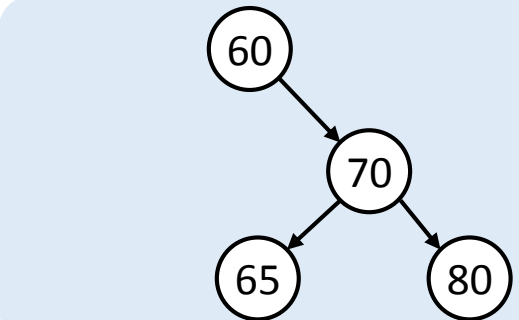
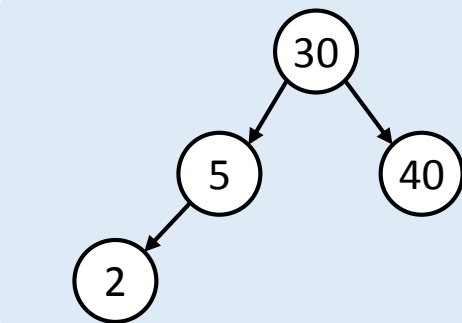
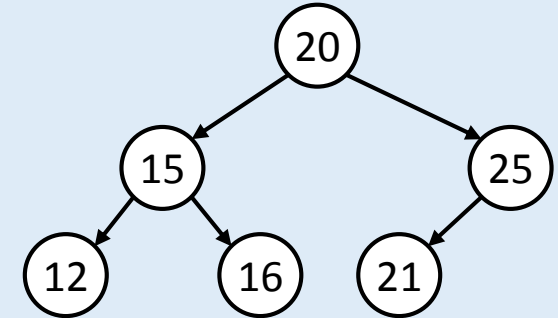
- Concept
 - Use an unordered list (array or chain) to store all elements
 - Scan the entire list for Get(), Insert(), and Delete()
- Time complexity

	Unordered List	Note
IsEmpty()	$O(1)$	
Get(key)	$O(n)$	
Insert(pair)	$O(n)$	Require scanning the list for possible duplicate key
Delete(key)	$O(n)$	



Binary Search Tree (BST)

- Definition: a BST is
 - A binary tree
 - May be empty
 - If a BST is not empty
 - Each element has a **distinct key**
 - Keys (if any) in the **left subtree** are **smaller** than the key in the root
 - Keys (if any) in the **right subtree** are **larger** than the key in the root
 - The left and right subtrees are also BSTs
- Concept
 - Root partitions all elements into two subtrees
 - Recall the binary search algorithm





Searching a Binary Search Tree (Recursive)

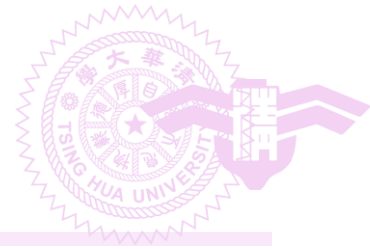
```
template <class K, class E>
pair<K, E>* BST<K, E> :: Get(const K& k)
{ // Driver
    return rGet(root, k);
}

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

The two data members of an STL pair are named as "first" and "second"

It is correct to name the workhorse "Get" as the textbook does (because of function overloading). I change the name to "rGet" for clarity.

Searching a Binary Search Tree (Iterative)

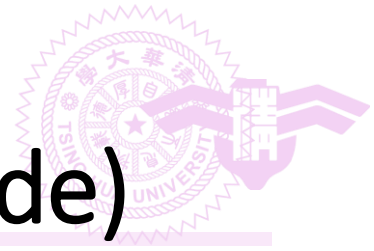


```
template <class K, class E>
pair<K, E>* BST<K, E>::Get(const K& k)
{
    TreeNode < pair<K, E> > * currentNode = root;

    while (currentNode) {
        if (k < currentNode->data.first)
            currentNode = currentNode->leftChild;
        else if (k > currentNode->data.first)
            currentNode = currentNode->rightChild;
        else
            return & currentNode->data;
    }

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

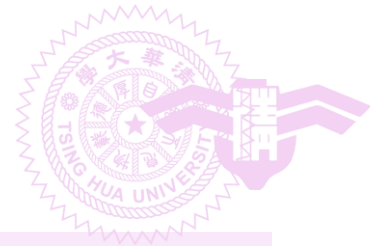
Searching a Binary Search Tree (Iterative) (Exemplifying Pseudo Code)



```
template <class K, class E>
pair<K, E>* BST<K, E>::Get(const K& k)
{
    Let currentNode be a pointer point to the root node;

    while (currentNode is not zero) {
        if (k < the key field pointed by currentNode)
            Let currentNode point to the left subtree;
        else if (k > the key field pointed by currentNode)
            Let currentNode point to the right subtree;
        else
            return the address of the data field pointed by currentNode;
    }

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



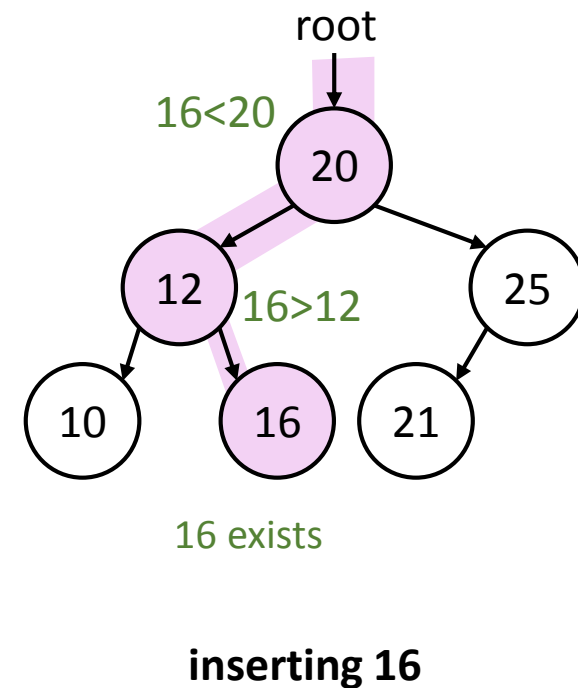
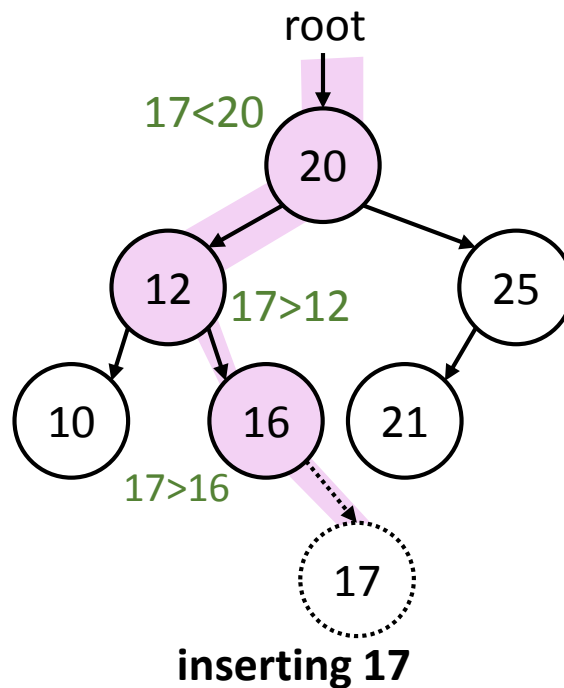
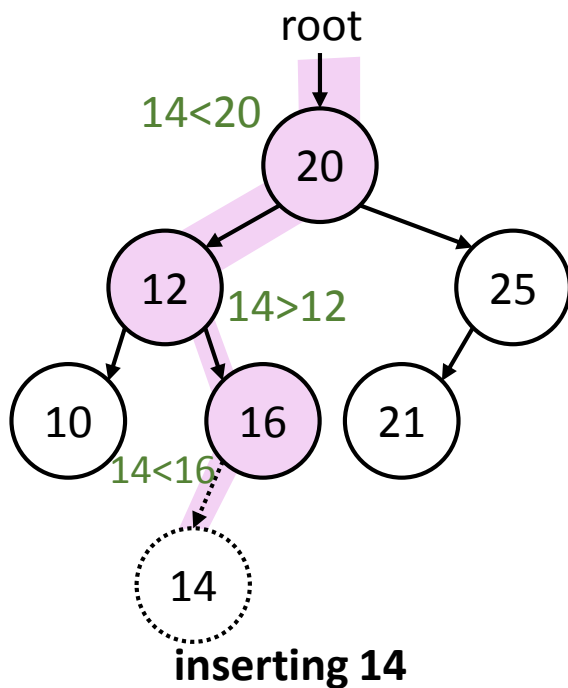
BST Operations

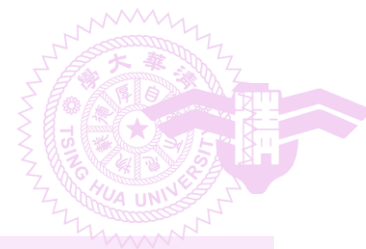
- Searching a BST by a key
- Searching a BST **by rank**
- Insertion
- Deletion
- Joining BSTs
- Splitting a BST



Insertion

- Insertion involves searching for the key
 - BST cannot contain duplicate keys
 - If search succeeds, update the content of the searched node
 - If search does not succeed, insertion takes place at the point the search terminated





Insertion

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

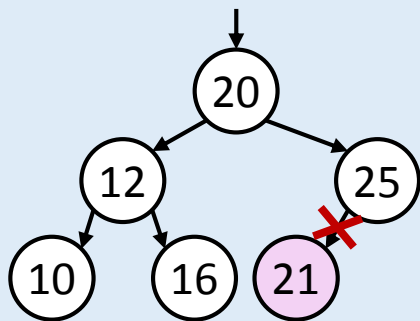
Time complexity: $O(h)$



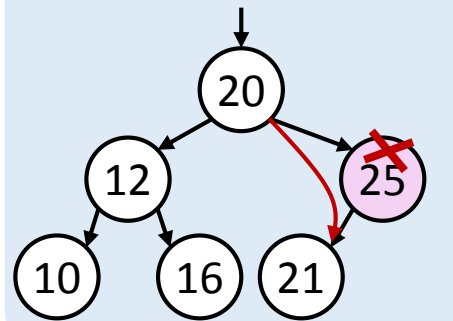
Deletion

- Insertion also involves searching for the key
- If the node to delete does not exist, nothing occurs
- Otherwise, the node to delete can have
 - no children → just delete it
 - single child → bypass the node
 - two children →
 - Replace the node with its successor (or predecessor)
 - Successor is the largest node of the left subtree
 - Handle the deletion of the successor

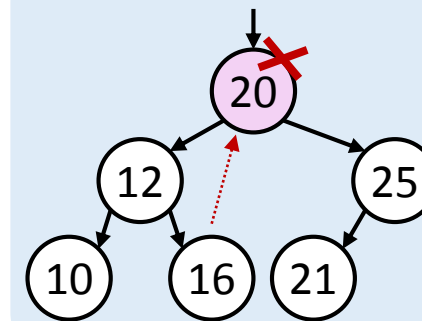
Time complexity: $O(h)$



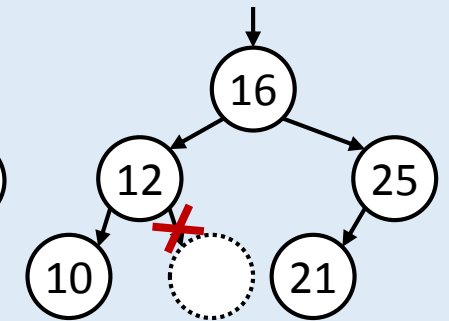
No children



One child



Two children

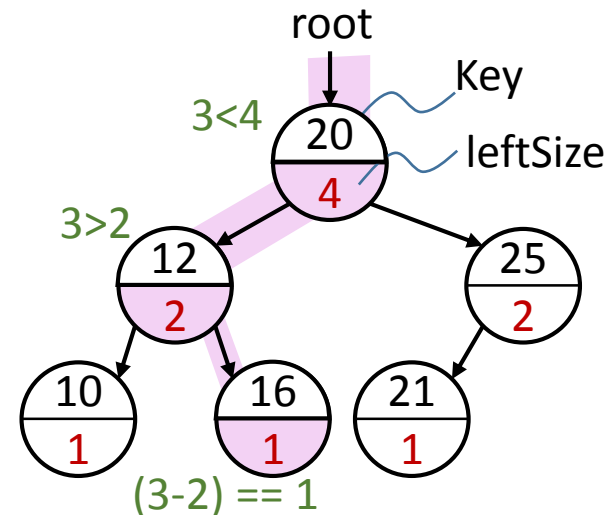




Searching by Rank

- Key concepts
 - Node's **rank** is its **inorder position**
 - Each node equips an **additional field: leftSize**
 - One plus the number of nodes in the left subtree of the node
 - **leftSize** can guide the rank-based search

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



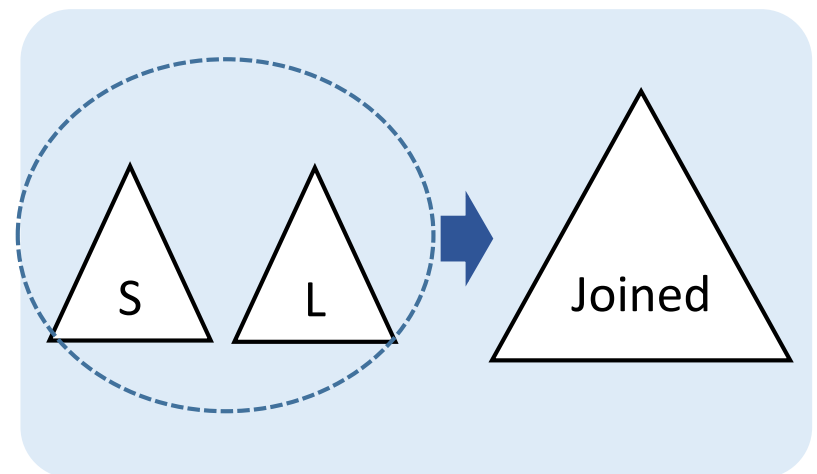
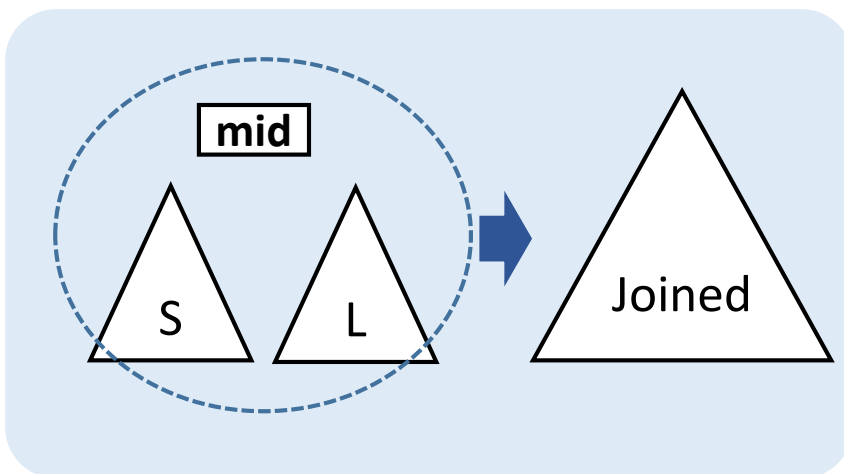
Searching the rank-3 node

Time complexity: $O(h)$



Joining BSTs

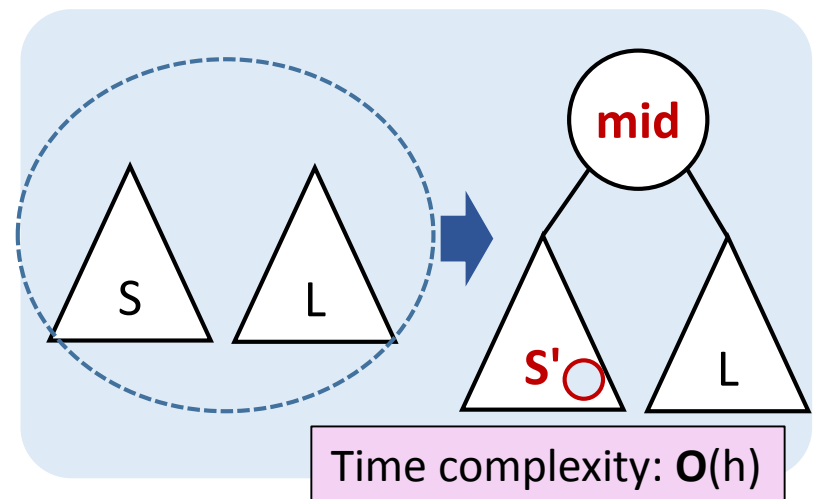
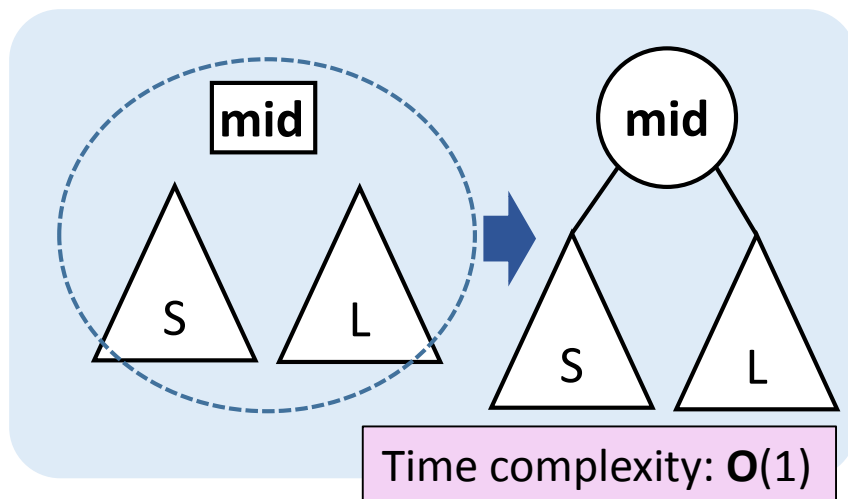
- Three-way joining
 - Given one key called **mid**
 - Given two BSTs named **S** and **L**
 - Each key in **S** is smaller than **mid**
 - Each key in **L** is larger than **mid**
- Two-way joining
 - Given two BSTs named **S** and **L**
 - All keys of **S** are smaller than all keys of **L**





Joining BSTs

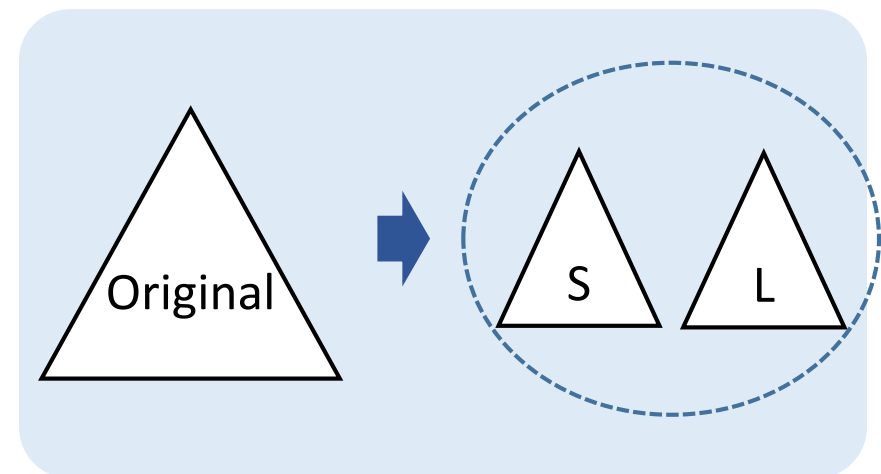
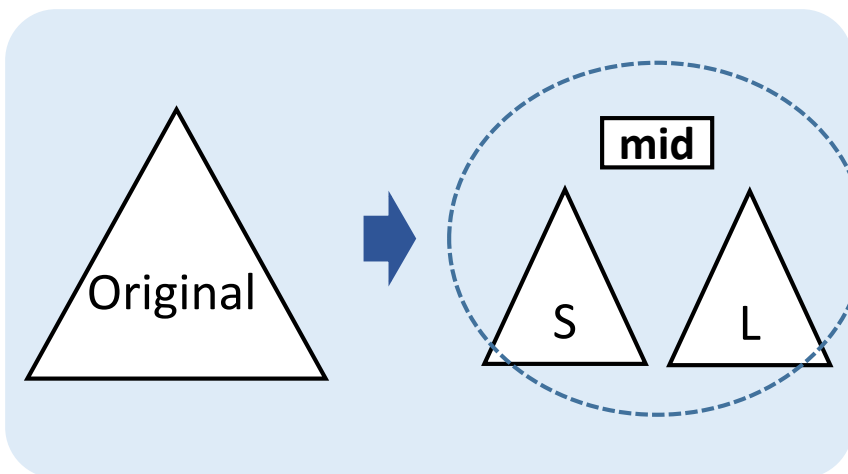
- Three-way joining (easy to perform)
 - Create a root node whose key is **mid**
 - Let **S** and **L** be the left and right subtrees, respectively
- Two-way joining (leverage three-way joining)
 - Remove from **S** the node with the largest key (or from **L** with the smaller key)
 - Let the node be the **mid**
 - Perform three-way joining

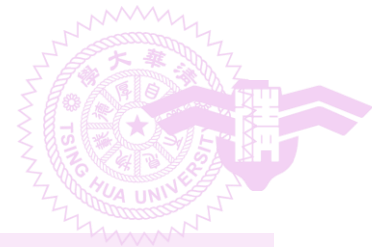




Splitting a BST by a Key

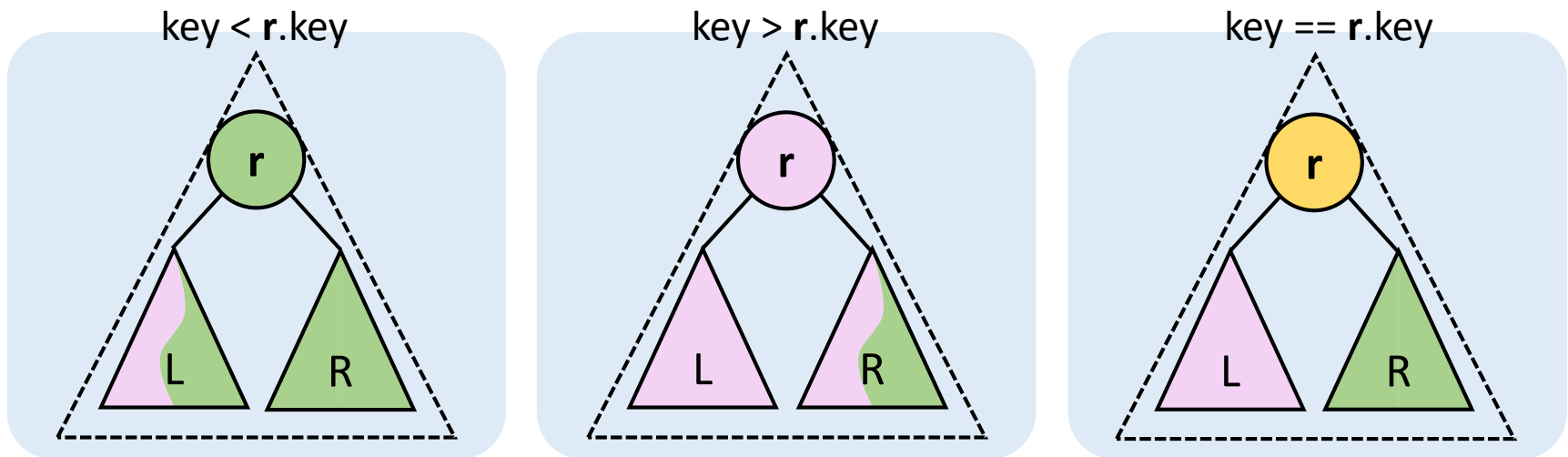
- Given a BST and a **key**
 - Generate two BSTs, **S** and **L**
 - Each key in **S** is **smaller** than the **key**
 - Each key in **L** is **larger** than the **key**
 - Retrieve the pair if the key exists in the original BST





Splitting a BST by a Key

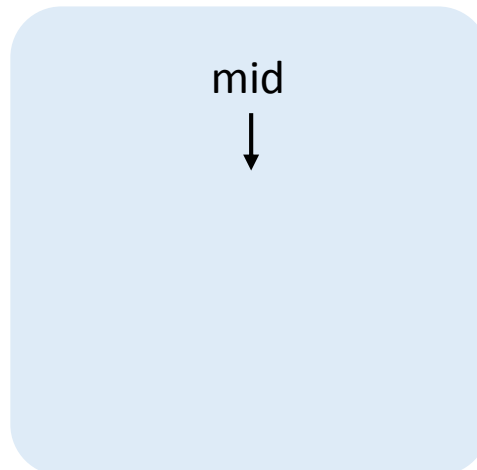
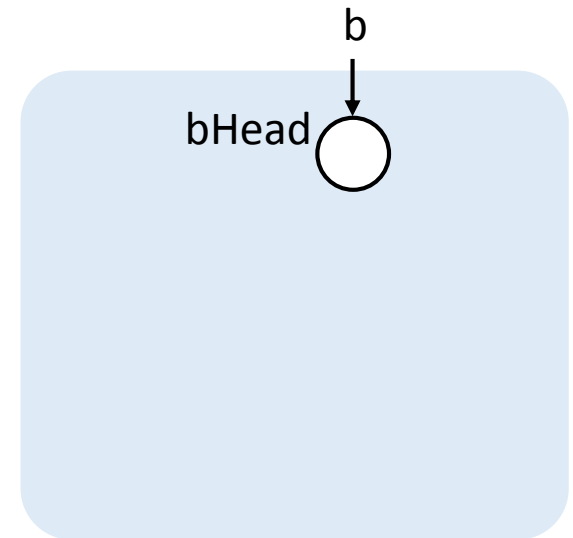
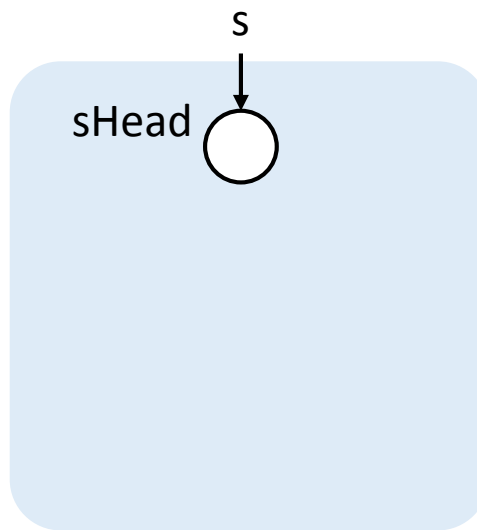
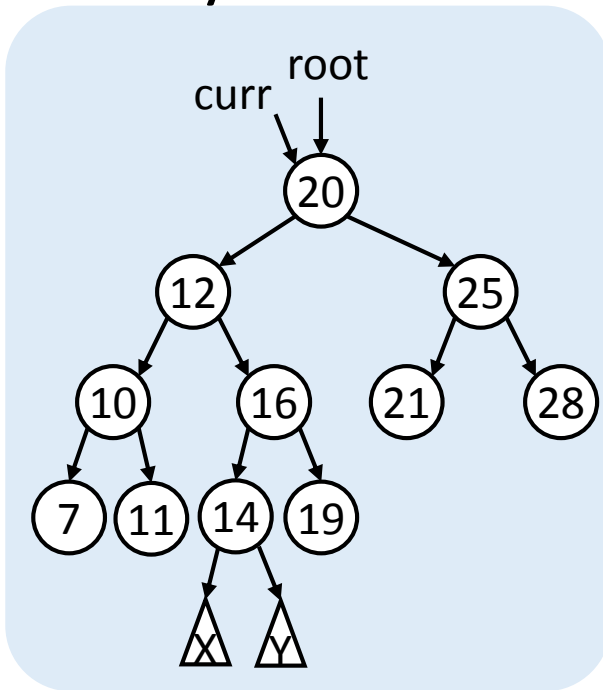
- Observations about splitting at a root node that have subtrees **L** and **R**
 - if ($\text{key} < \text{r.key}$)
 - Node **r** together with subtree **R** is to be in the larger BST
 - **Continue to split** subtree **L**
 - if ($\text{key} > \text{r.key}$)
 - Node **r** together with subtree **L** is to be in the smaller BST
 - **Continue to split** subtree **R**
 - if ($\text{key} == \text{r.key}$)
 - **R** and **L** are to be in the larger and smaller BSTs, respectively





Splitting a BST by a Key

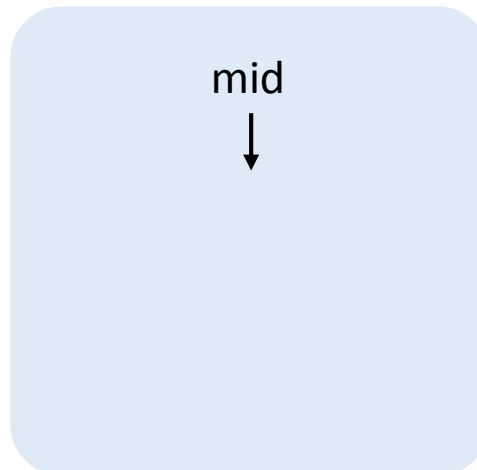
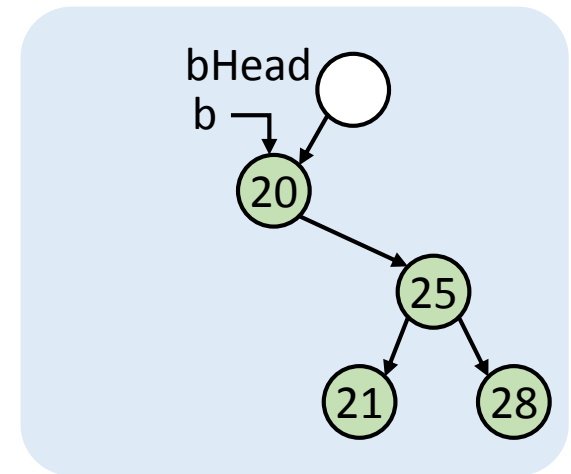
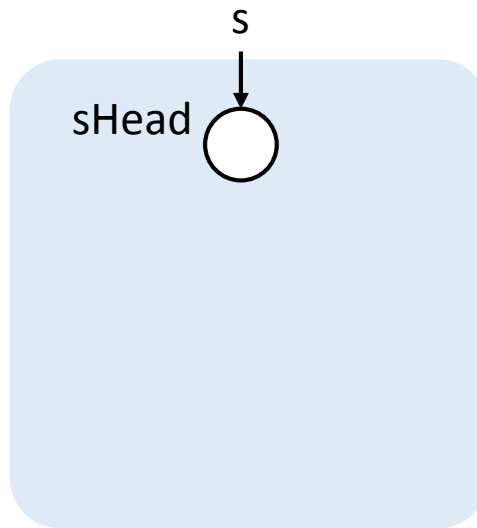
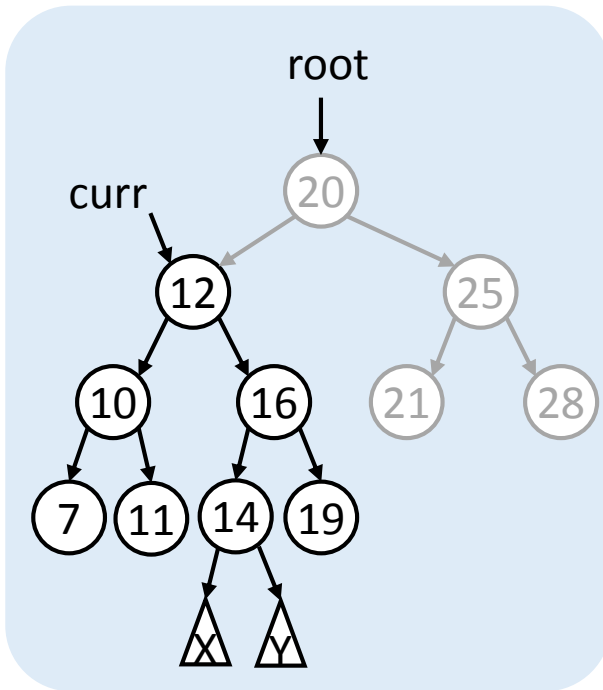
- Key = 14





Splitting a BST by a Key

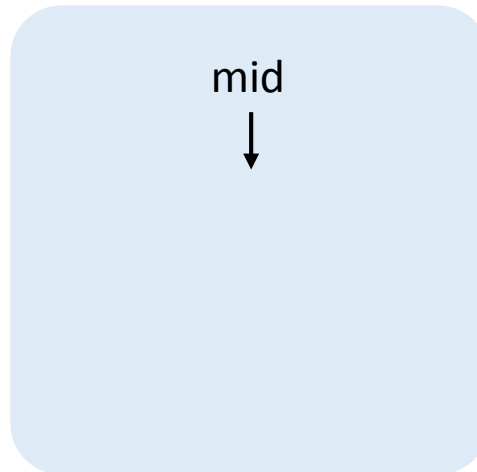
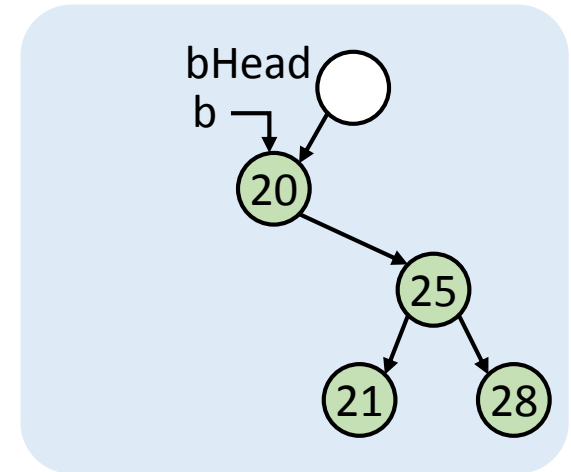
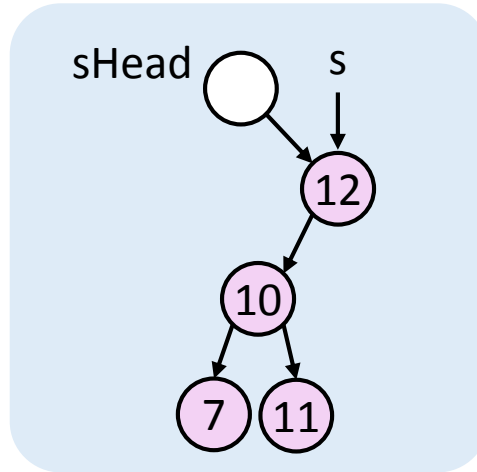
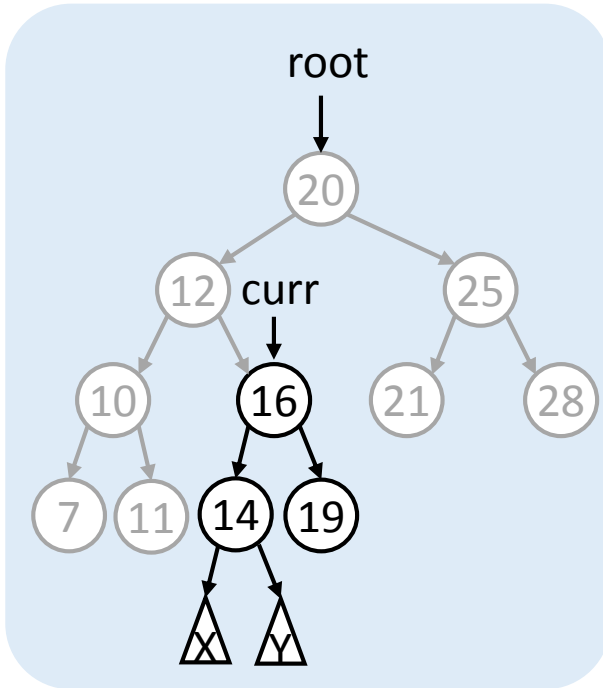
- $14 < 20$





Splitting a BST by a Key

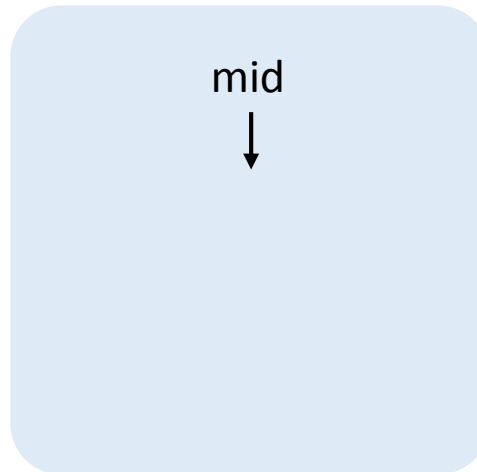
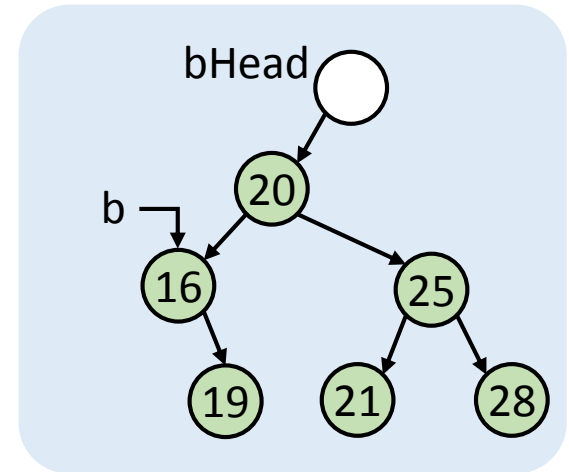
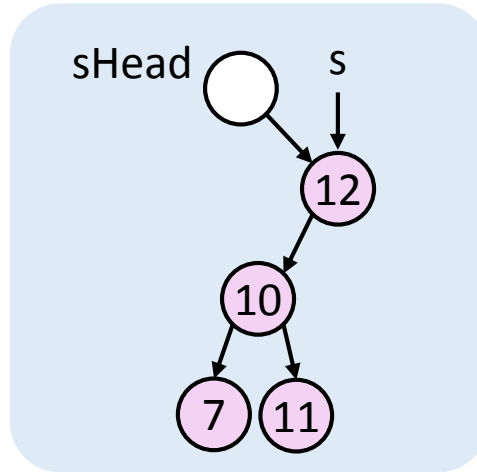
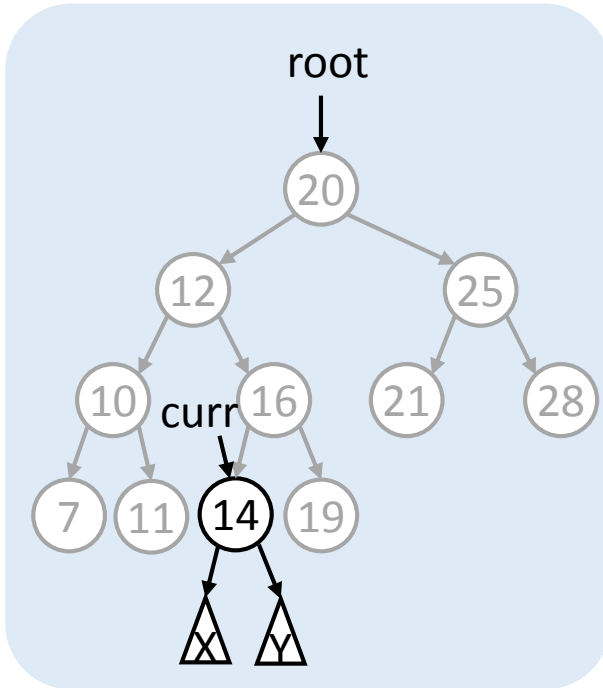
- $14 > 12$





Splitting a BST by a Key

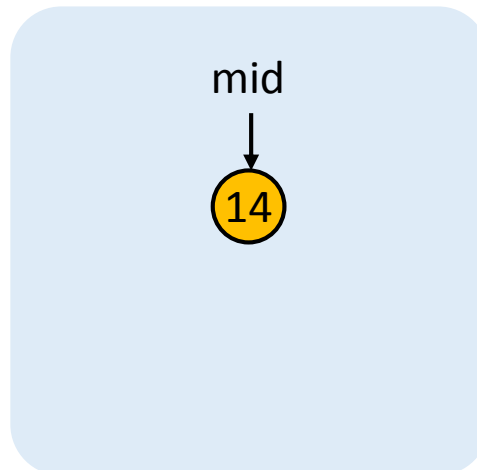
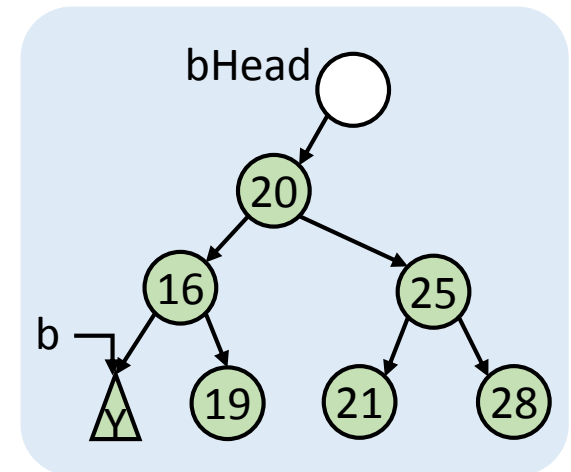
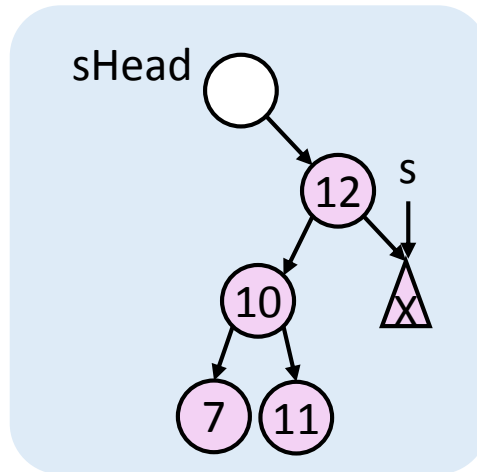
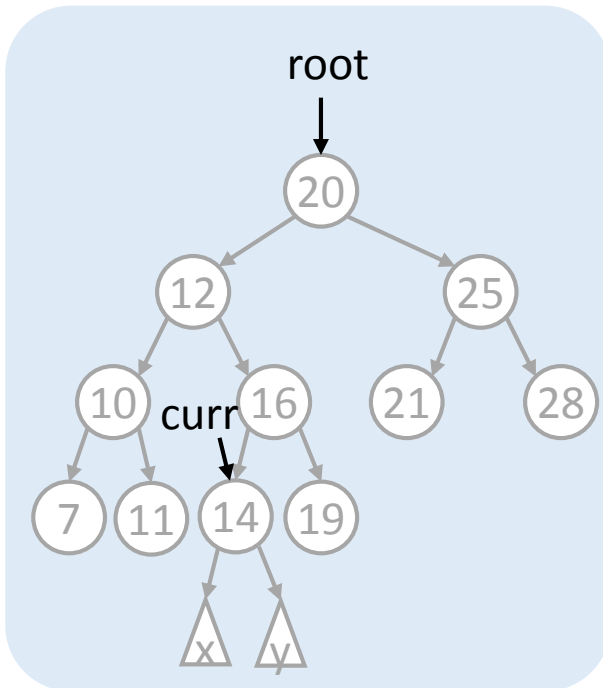
- $14 < 16$

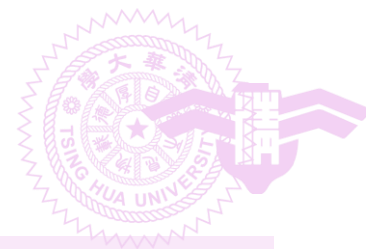




Splitting a BST by a Key

- $14 == 14$





Splitting a BST by a Key

```
template <class K, class E>
void BST<K, E>::Split(const K& k, BST<K, E>& small, pair<K, E>*& mid,
BST<K, E>& big)
{ // Split the BST with respect to key k
  if (!root) {small.root = big.root = 0; return;} // empty tree
  // create temporary header nodes for small and big
  TreeNode<pair<K, E> > *sHead = new TreeNode<pair<K, E> >,
    *s = sHead,
    *bHead = new TreeNode<pair<K, E> >,
    *b = bHead,
    *currentNode = root;

  while (currentNode)
    if (k < currentNode->data.first){ // case 1
      b->leftChild = currentNode;
      b = currentNode; currentNode = currentNode->leftChild;
    }
    else if (k > currentNode->data.first) { // case 2
      s->rightChild = currentNode;
      s = currentNode; currentNode = currentNode->rightChild;
    }
}
```

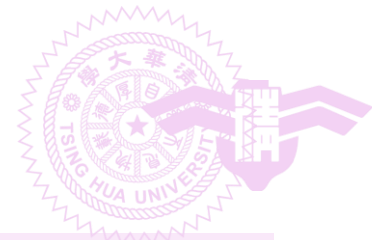


Splitting a BST by a Key

```
else { // case 3
    s->rightChild = currentNode->leftChild;
    b->leftChild = currentNode->rightChild;
    small.root = sHead->rightChild; delete sHead;
    big.root = bHead->leftChild; delete bHead;
    mid = new pair<K, E>(currentNode->data.first,
                        currentNode->data.second);

    delete currentNode;
    return;
}
// no pair with key k
s->rightChild = b->leftChild = 0;
small.root = sHead->rightChild; delete sHead;
big.root = bHead->leftChild; delete bHead;
mid = 0;
return;
}
```

Time complexity: $O(h)$

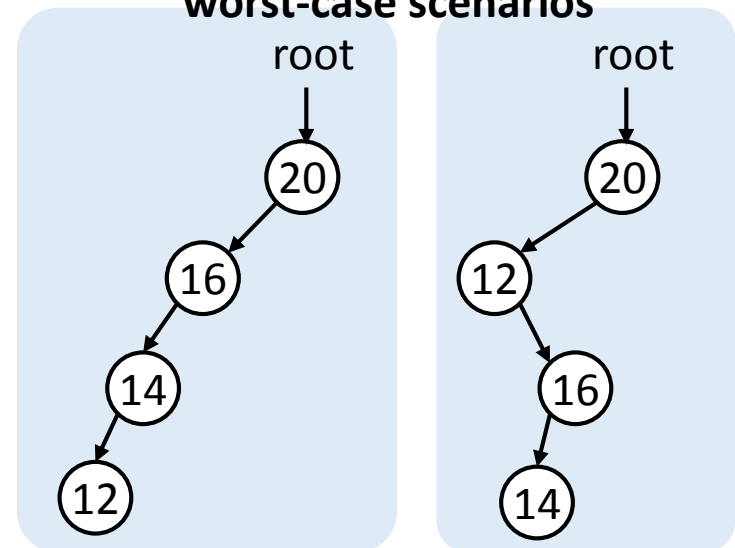


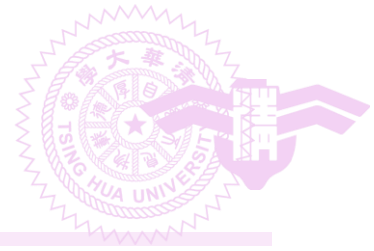
Discussion about BST Height

- Most BST operations are of **$O(\text{height})$** time complexity
- Consider an **n** -element BST
 - In the **worst case**, height = **$O(n)$**
 - If insertions and deletions are made at **random**, height = **$O(\log(n))$** on average (not guaranteed)
- **Balanced search trees** are search trees that can **guarantee worst case height = $O(\log(n))$**
 - Including **AVL trees**, **Red-Black trees** as in Chapter 10 and 11

Operations	Complexity
Searching by a key	$O(\text{height})$
Searching by rank	
Insertion	
Deletion	
Joining BSTs	
Splitting by a key	

worst-case scenarios





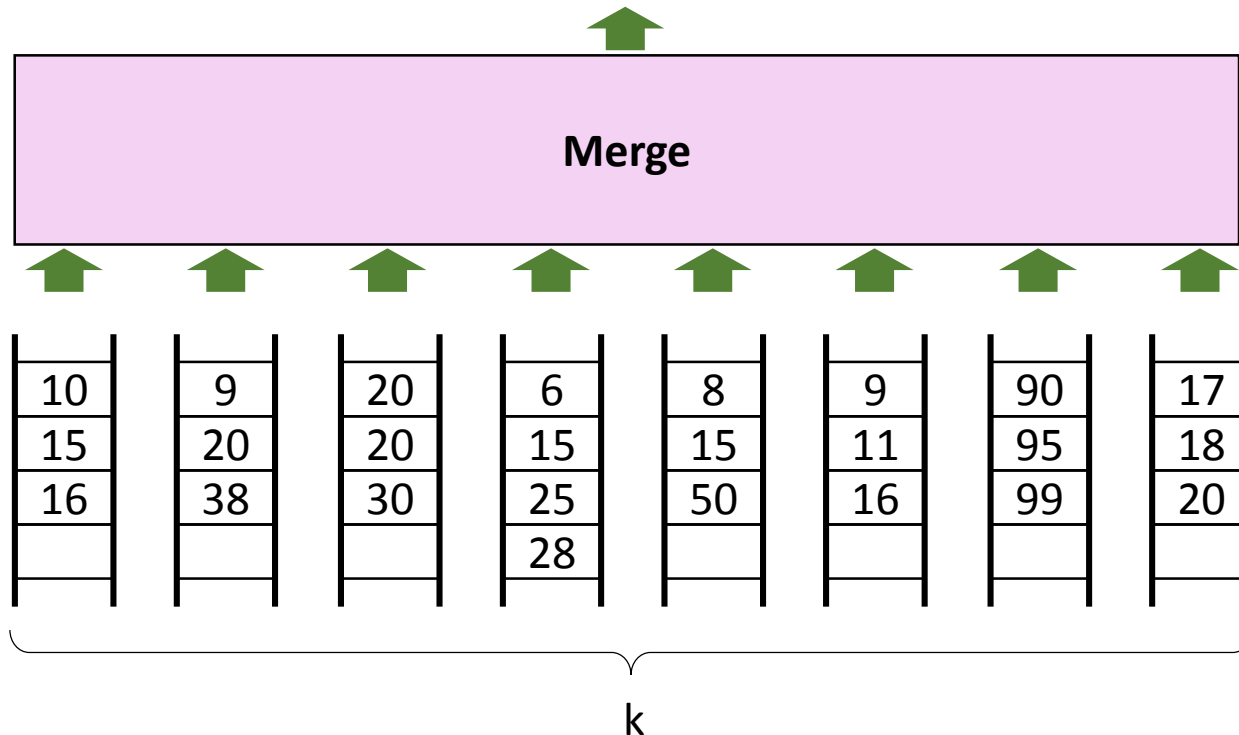
Outline

- 5.1 Introduction
- 5.2-5.5 Binary trees
- 5.6 Heaps
- 5.7 Binary search trees → Dictionary
- **5.8 Selection trees**
- 5.9 Forests
- (5.10 Disjoint sets)
- (5.11 Counting binary trees)



Motivation

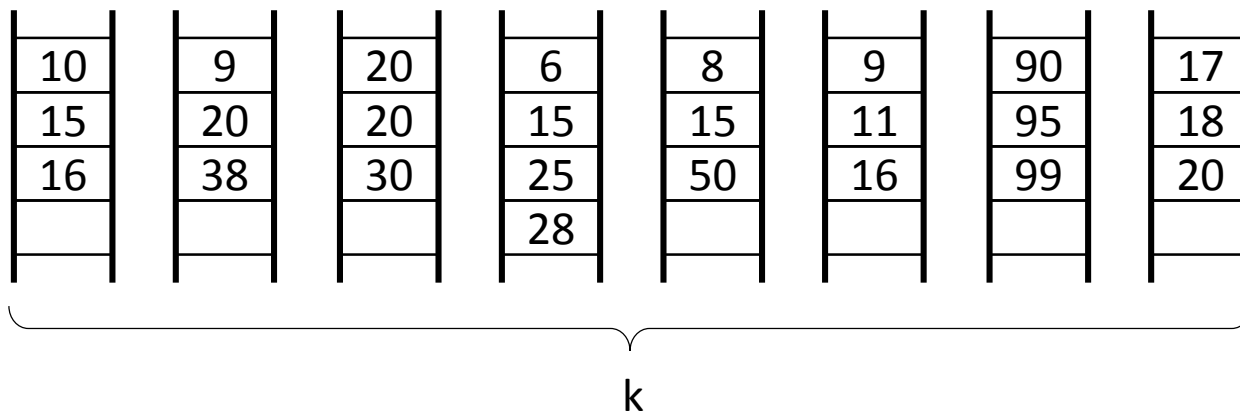
- Sometimes we need to merge k ordered sequences (called runs) with a total of n elements into a single ordered sequence





Algorithms

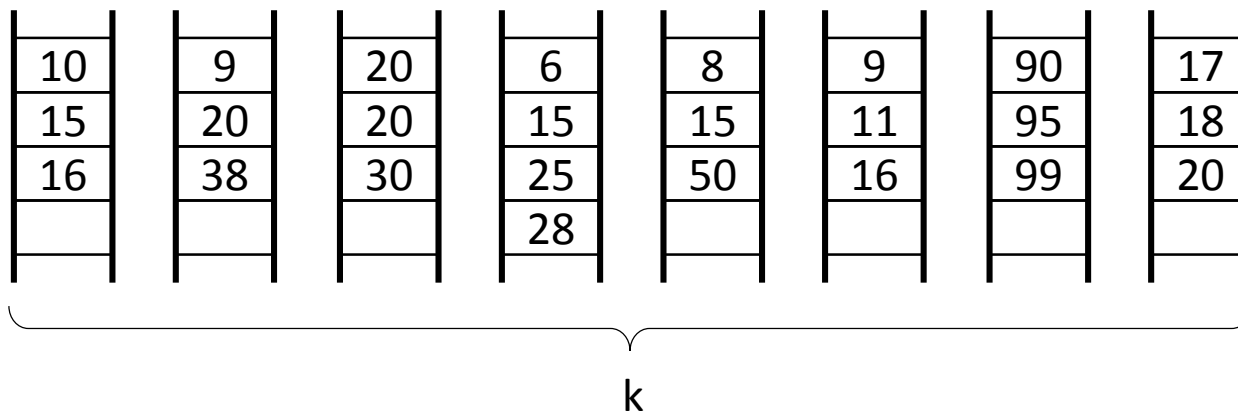
- Naïve algorithm
 - Perform $(k-1)$ comparisons to find the smallest/largest element among the k top elements
 - Repeat the above steps for $n-1$ times
 - Time complexity = $O(nk)$





Algorithms

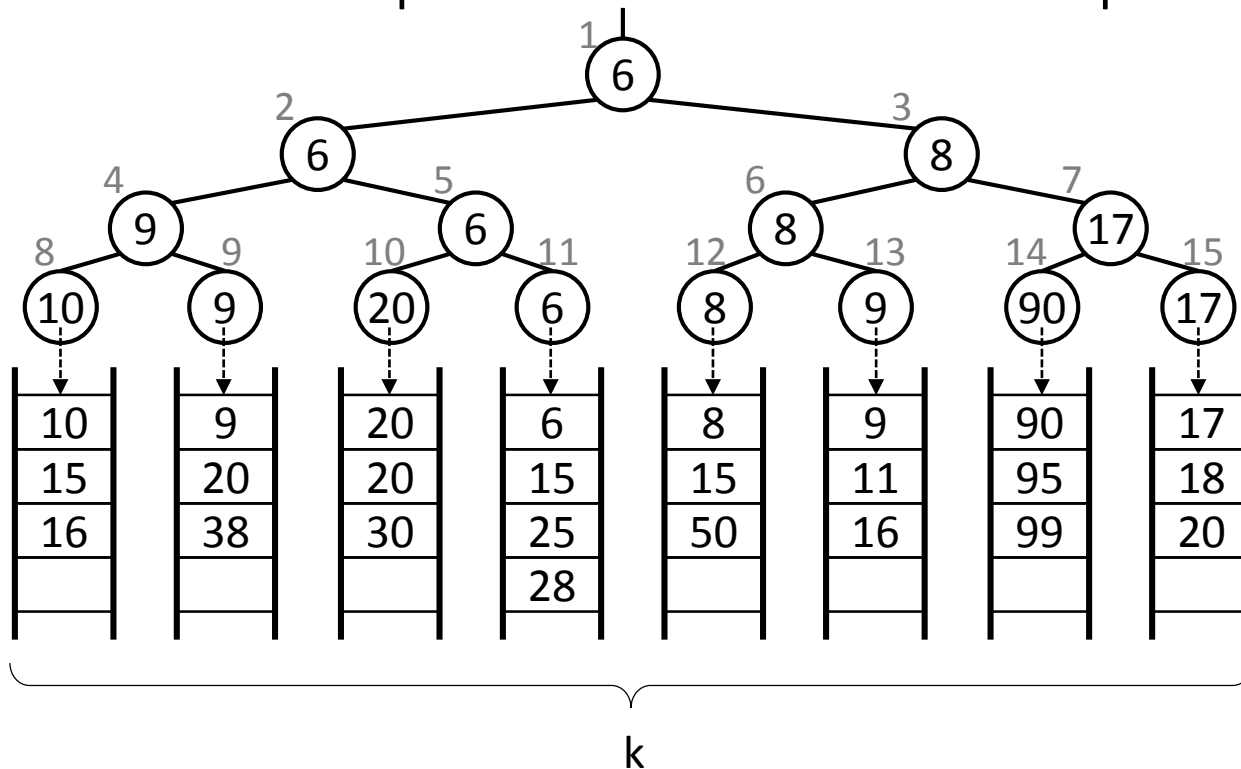
- Selection tree
 - Perform only $O(\log(k))$ comparisons to find the smallest/largest element
 - Time complexity = $O(n \log(k))$
 - Two implementations: winner trees and loser trees





Winner Tree

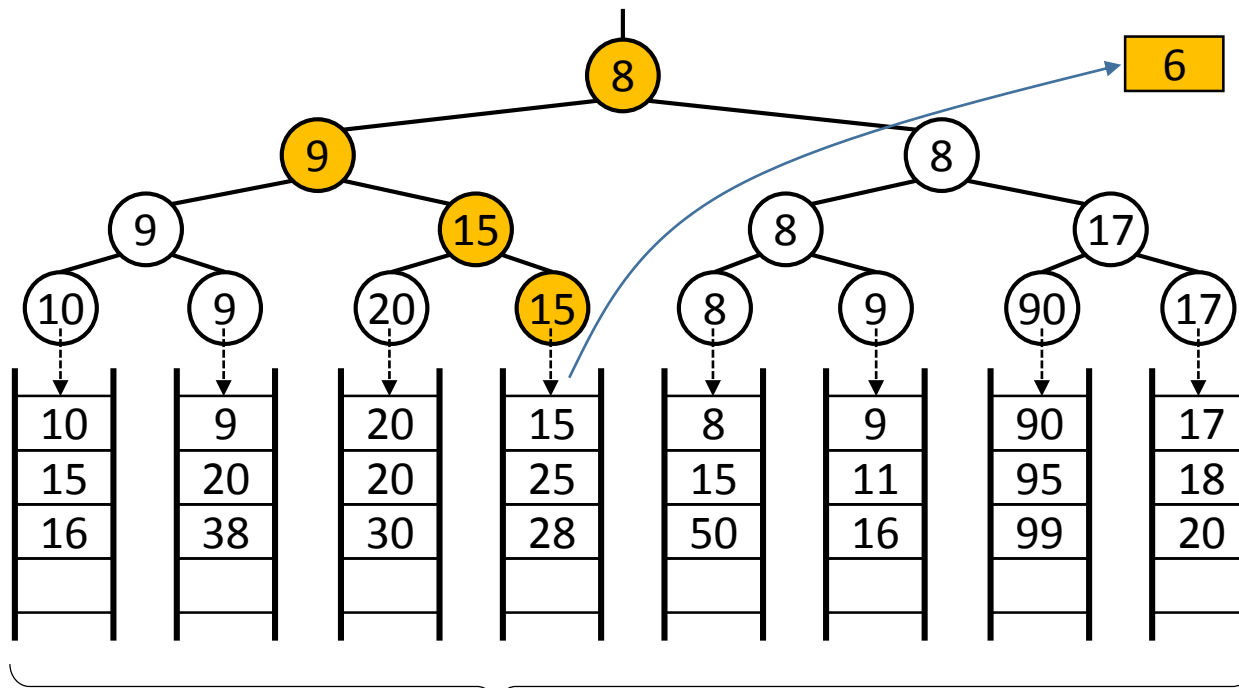
- Complete binary tree in which each node represents to the winner (e.g., **the smaller** in the following example) of its two children
 - Each node uses a pointer to link the record it represents





Merge Sequences

- Repeatedly performs
 - Outputting the record pointed by the root node, which represents the overall winner
 - Replaying the tournament along the path from the output node

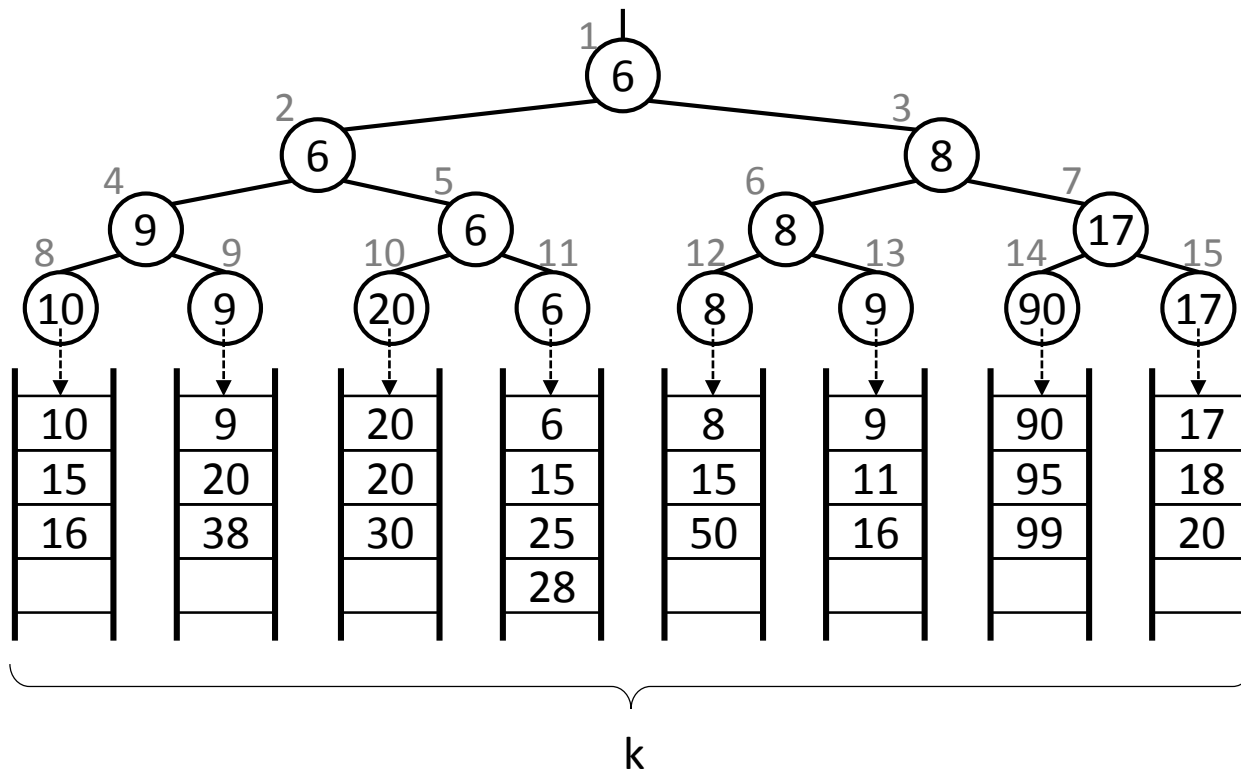


Time complexity of each output: $O(\log(k))$
Time complexity of the entire merge: $O(n \log(k))$



Inefficiency of Winner Trees

- Tree nodes store the duplicate information
 - E.g., there are four 6's and three 8's in the following winner tree
- Need to access the both children to reconstruct a node

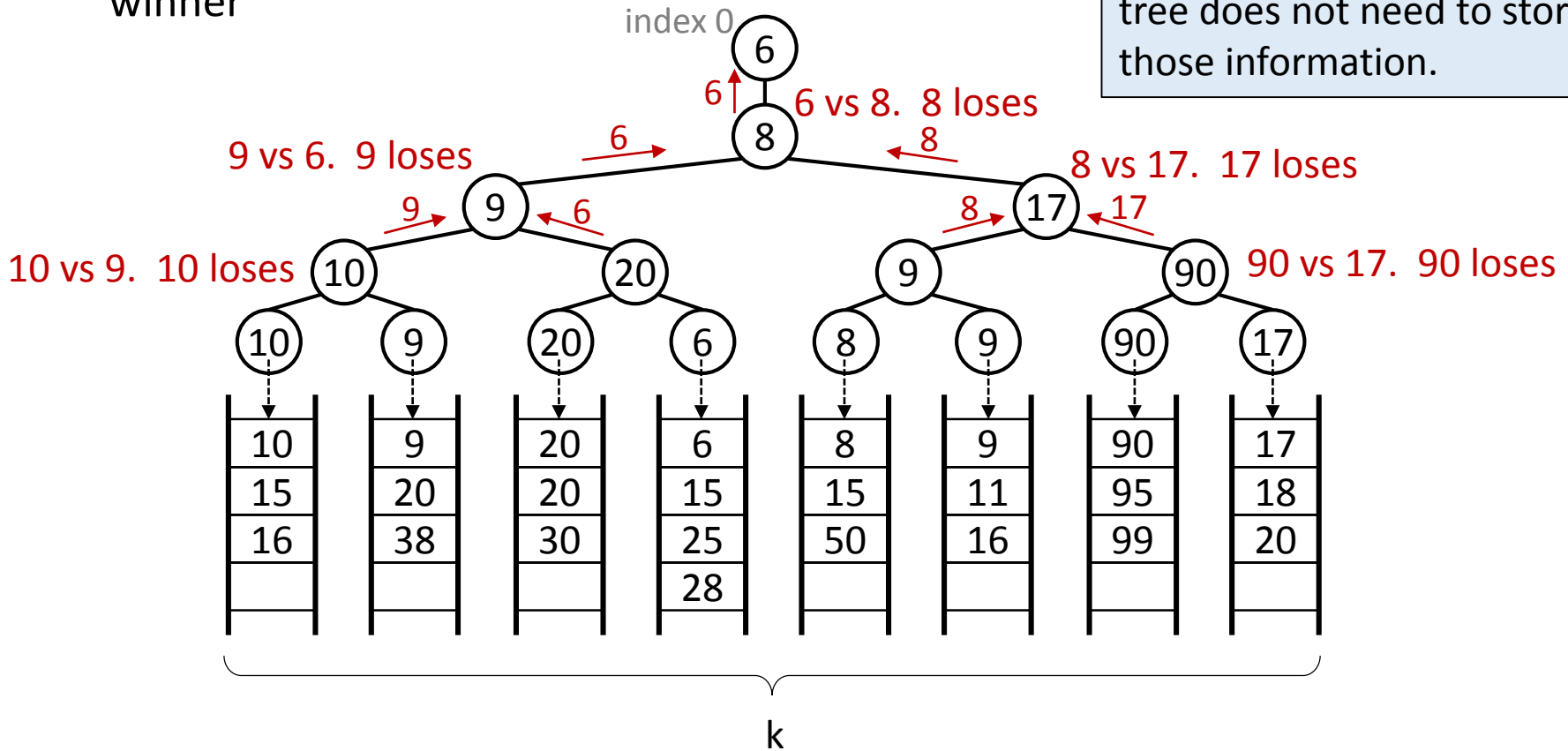




Loser Tree

- Non-leaf nodes represent losers, instead of winners
- Index-0 node represents the overall winner

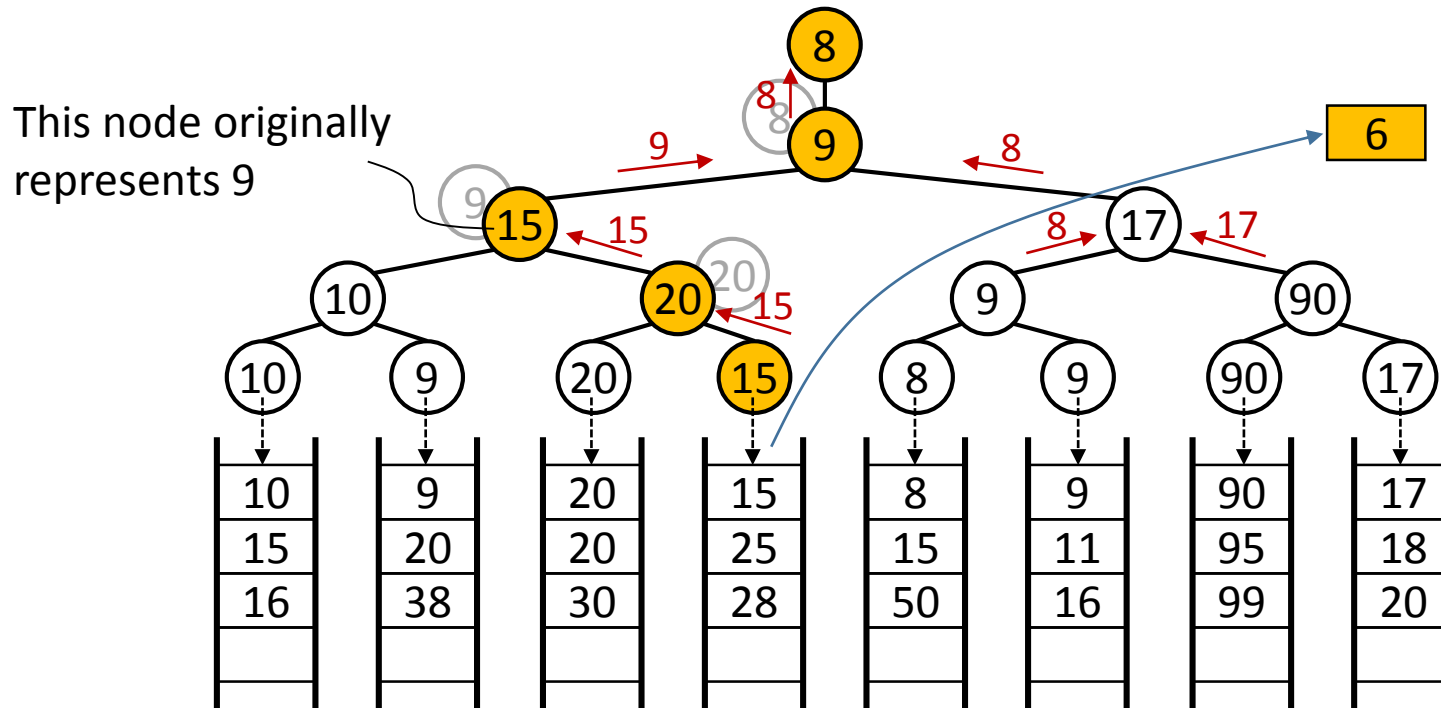
Red arrows and text are used to help understanding the loser tree only. The tree does not need to store those information.





Loser Tree

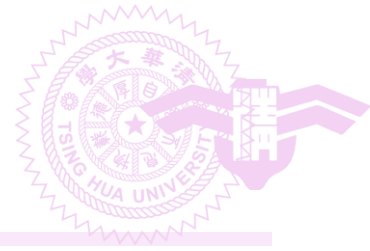
- Non-leaf nodes represent losers
- Index-0 node represents the overall winner



Time complexity of each output: $O(\log(k))$

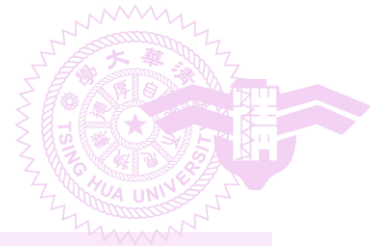
Time complexity of the entire merge: $O(n \log(k))$

Loser trees tend to outperform winner trees in speed because of the reduced constant



Outline

- 5.1 Introduction
- 5.2-5.5 Binary trees
- 5.6 Heaps
- 5.7 Binary search trees → Dictionary
- 5.8 Selection trees
- **5.9 Forests**
- (5.10 Disjoint sets)
- (5.11 Counting binary trees)



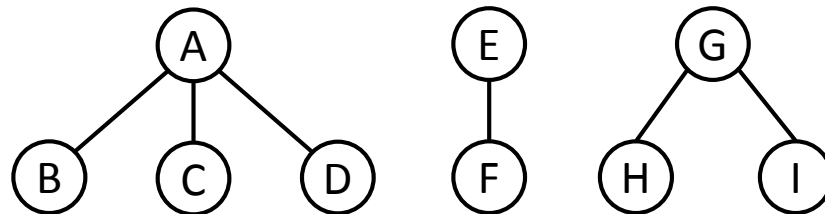
Outline

- 5.1 Introduction
- 5.2-5.5 Binary trees
- 5.6 Heaps
- 5.7 Binary search trees → Dictionary
- 5.8 Selection trees
- **5.9 Forests**
- (5.10 Disjoint sets)
- (5.11 Counting binary trees)



Forest

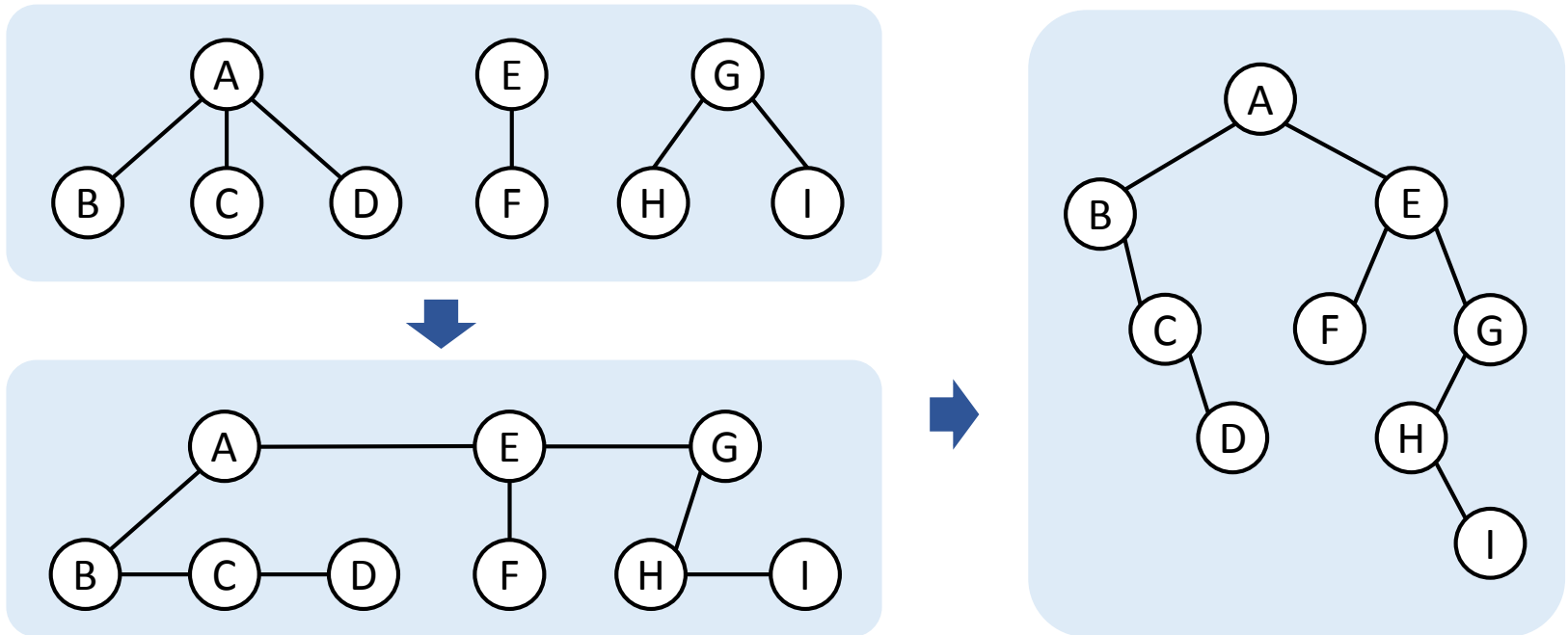
- Definition
 - A forest is a set of $n \geq 0$ disjoint trees
- Removing the root of a tree produces a forest (i.e., the subtrees)
- Three-tree forest example



- Forest operations
 - Transforming a forest into a binary tree
 - Forest traversals

Forest to Binary Tree

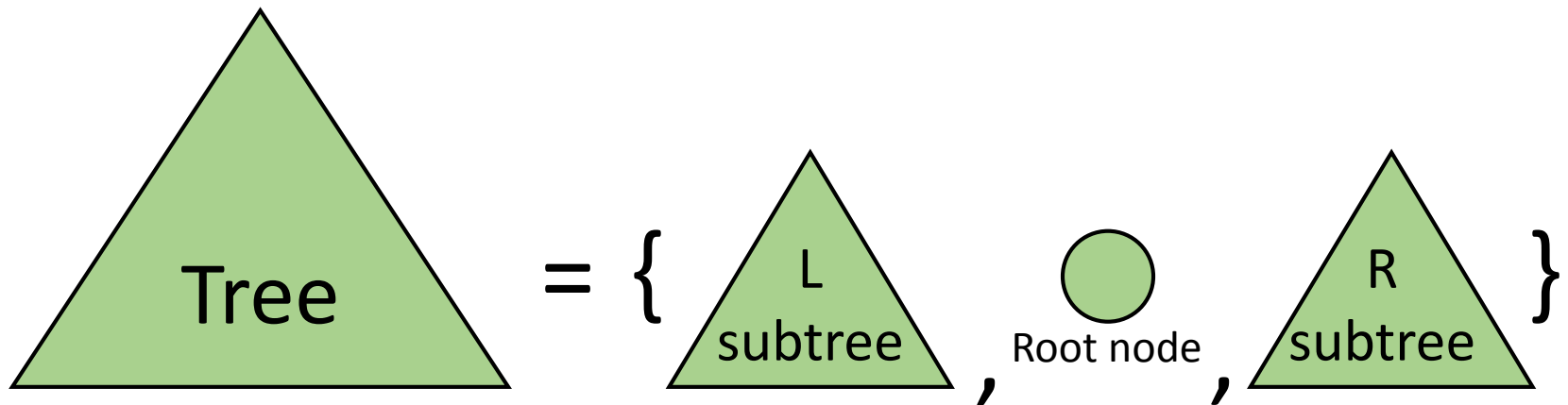
- Left-child-right-sibling approach





Recap Inorder Tree Traversal

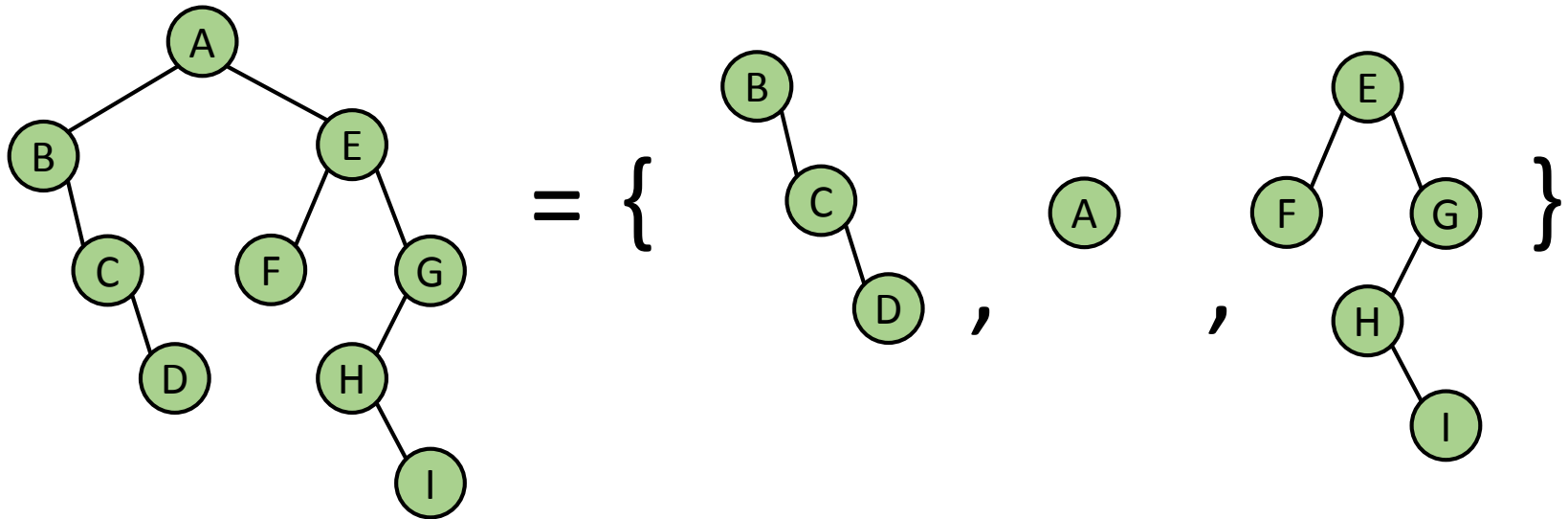
- Recursive definition





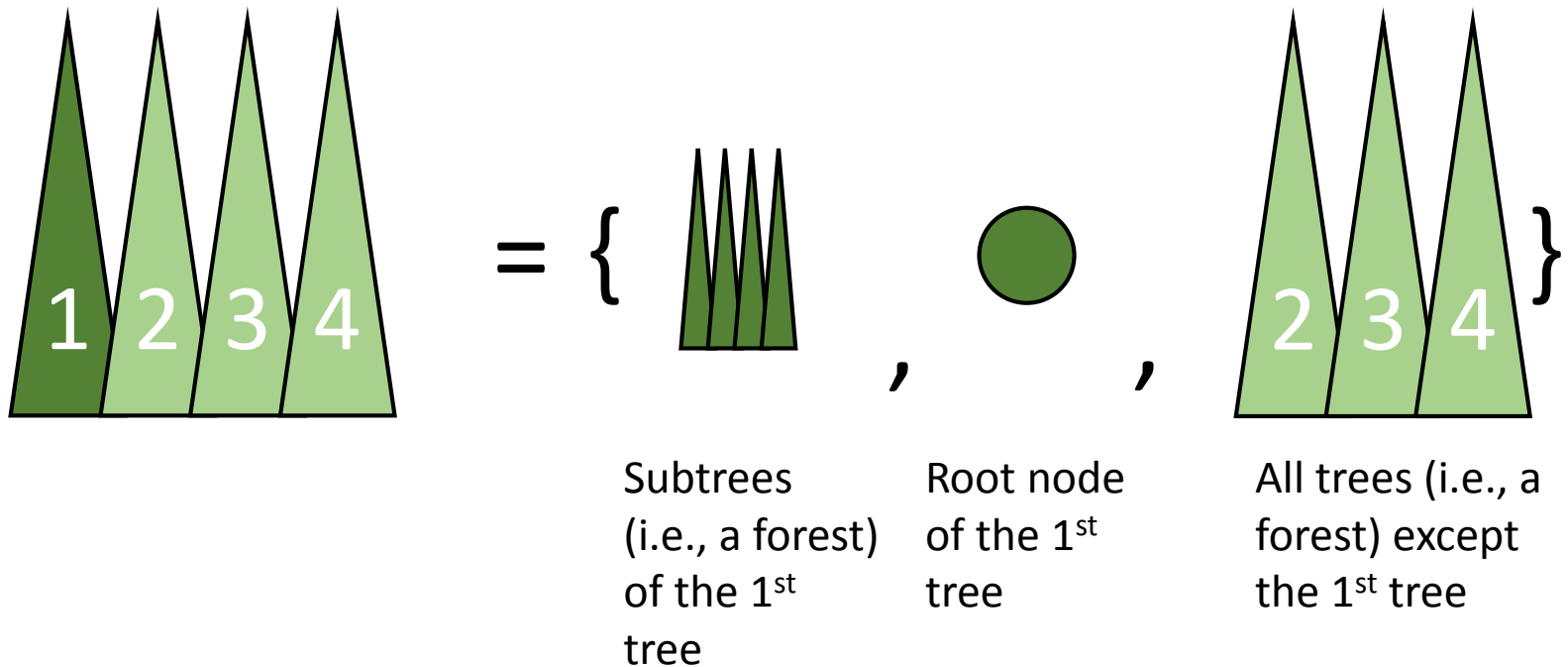
Recap Inorder Tree Traversal

- Example



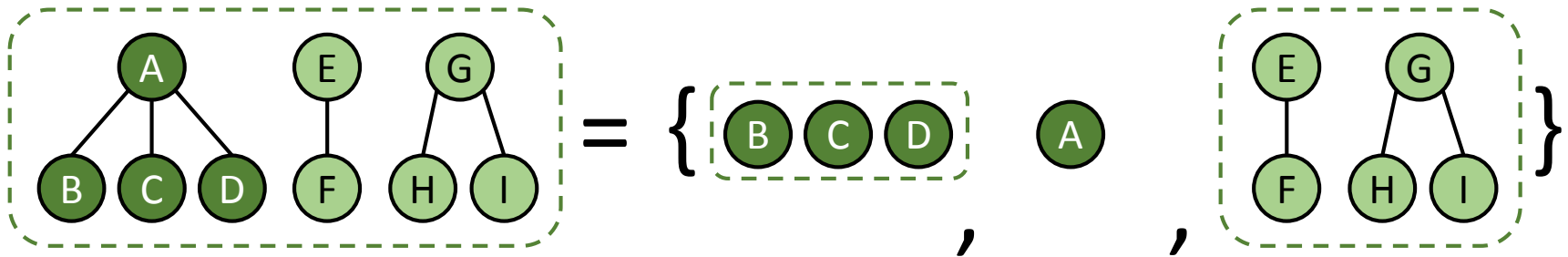
Inorder Forest Traversal

- Recursive definition



Inorder Forest Traversal

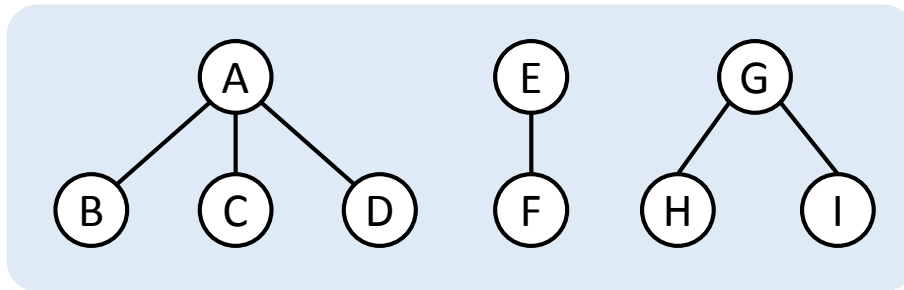
- Recursive definition



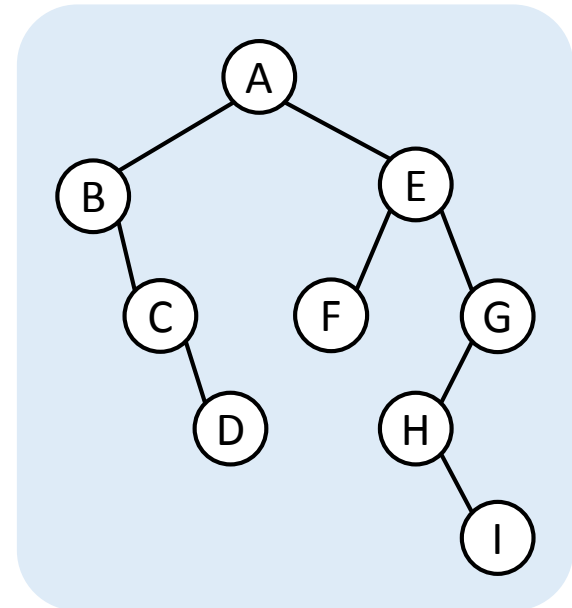


Forest Traversals (Preorder)

- Traverse F in 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



A B C D E F G H I

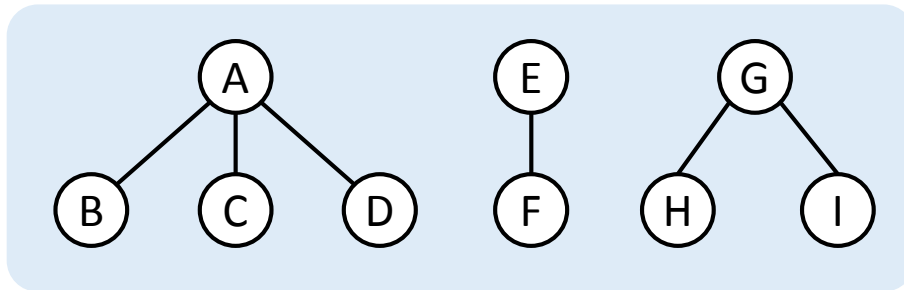


A B C D E F G H I

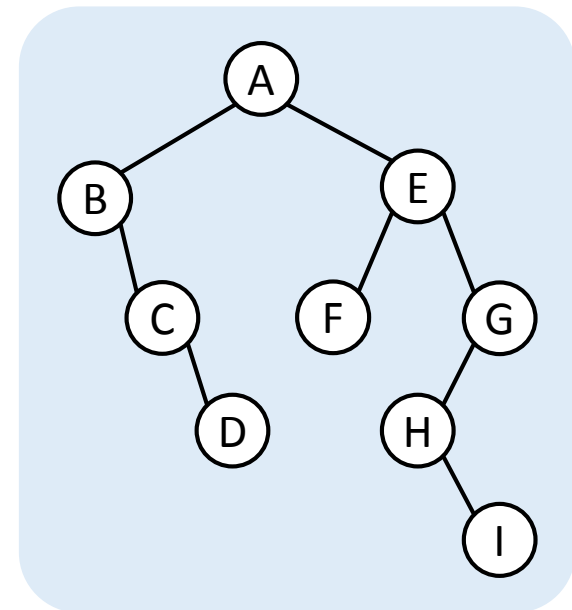


Forest Traversals (Inorder)

- Traverse F in **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**



BCDAFEHIG

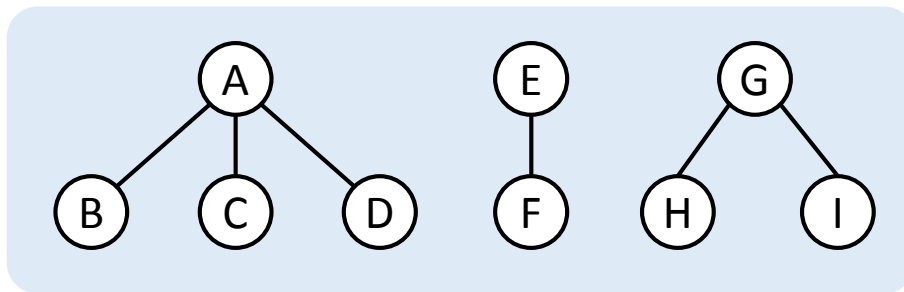


BCDAFEHIG

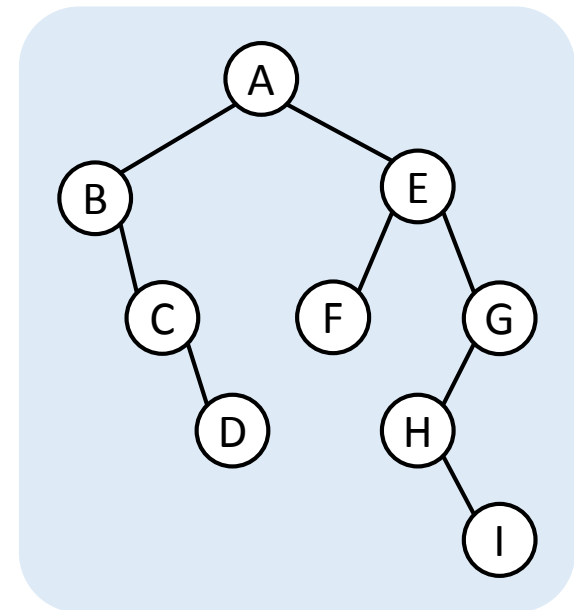


Forest Traversals (Postorder)

- Traverse F in 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



DCBFIHGEA

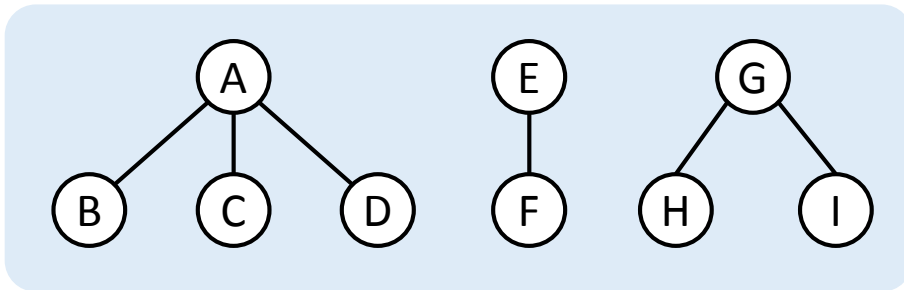


DCBFIHGEA

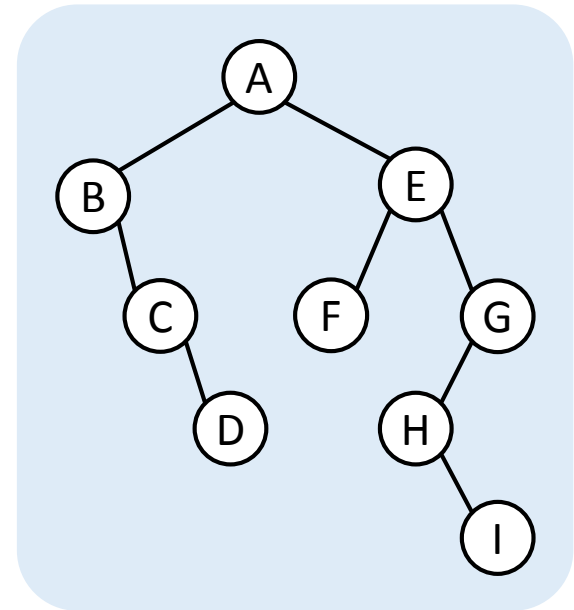


Forest Traversals (Level-Order)

- Traverse F in forest level-order
 - If F is empty then return
 - Nodes are visited by level
 - Begin the traversal from the roots of each tree in the forest
 - Within each level, nodes are visited from left to right



A E G B C D F H I



A B E C F G D H I