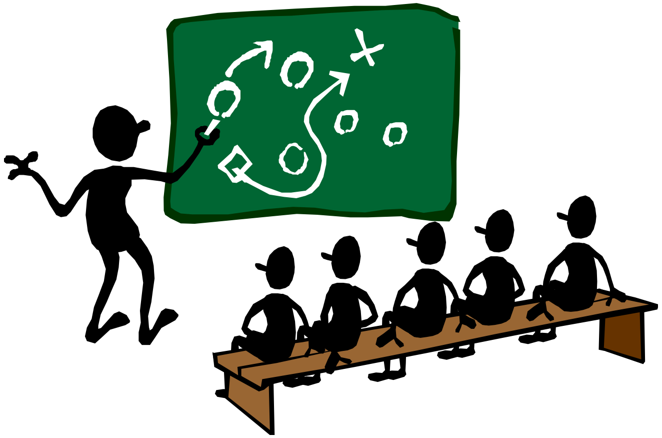
Data Structure

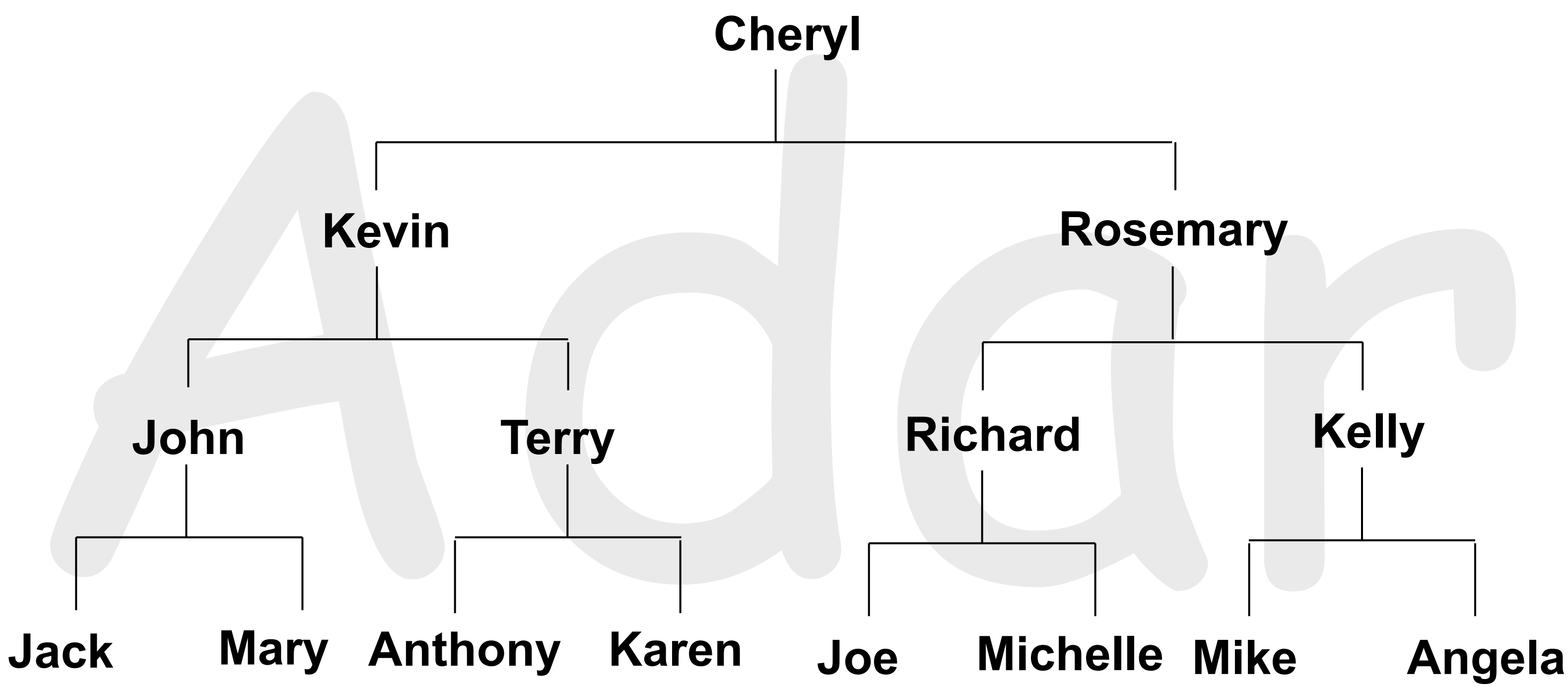
Chapter 5 Trees



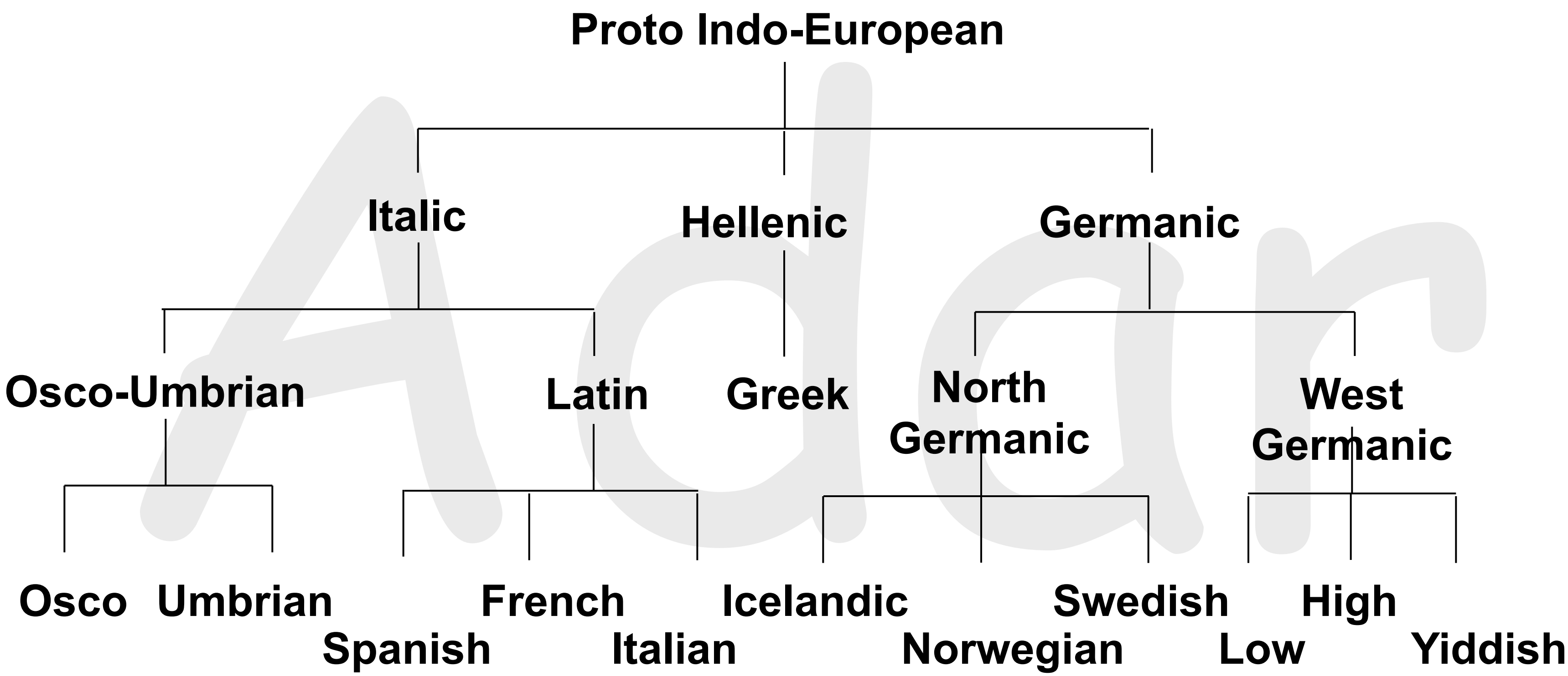
Prof. Mark Po-Hung Lin
Institute of Intelligent Systems, AI College
Institute of Electronics, ECE College
National Yang Ming Chiao Tung University

Most of the lecture slides adapted from Prof. Juinn-Dar Huang

Tree Example - Pedigree



Tree Example - Lineal



Definition (1/3)

- A tree is a finite set of one or more nodes s.t.
  - there is a root node (one and only one)
  - the remaining nodes are partitioned into $n \geq 0$ ***disjoint trees*** $T_1, \ldots, T_n$.
    - subtrees cannot share nodes
  - $T_1, \ldots, T_n$ are subtrees of the root

- This is a recursive definition

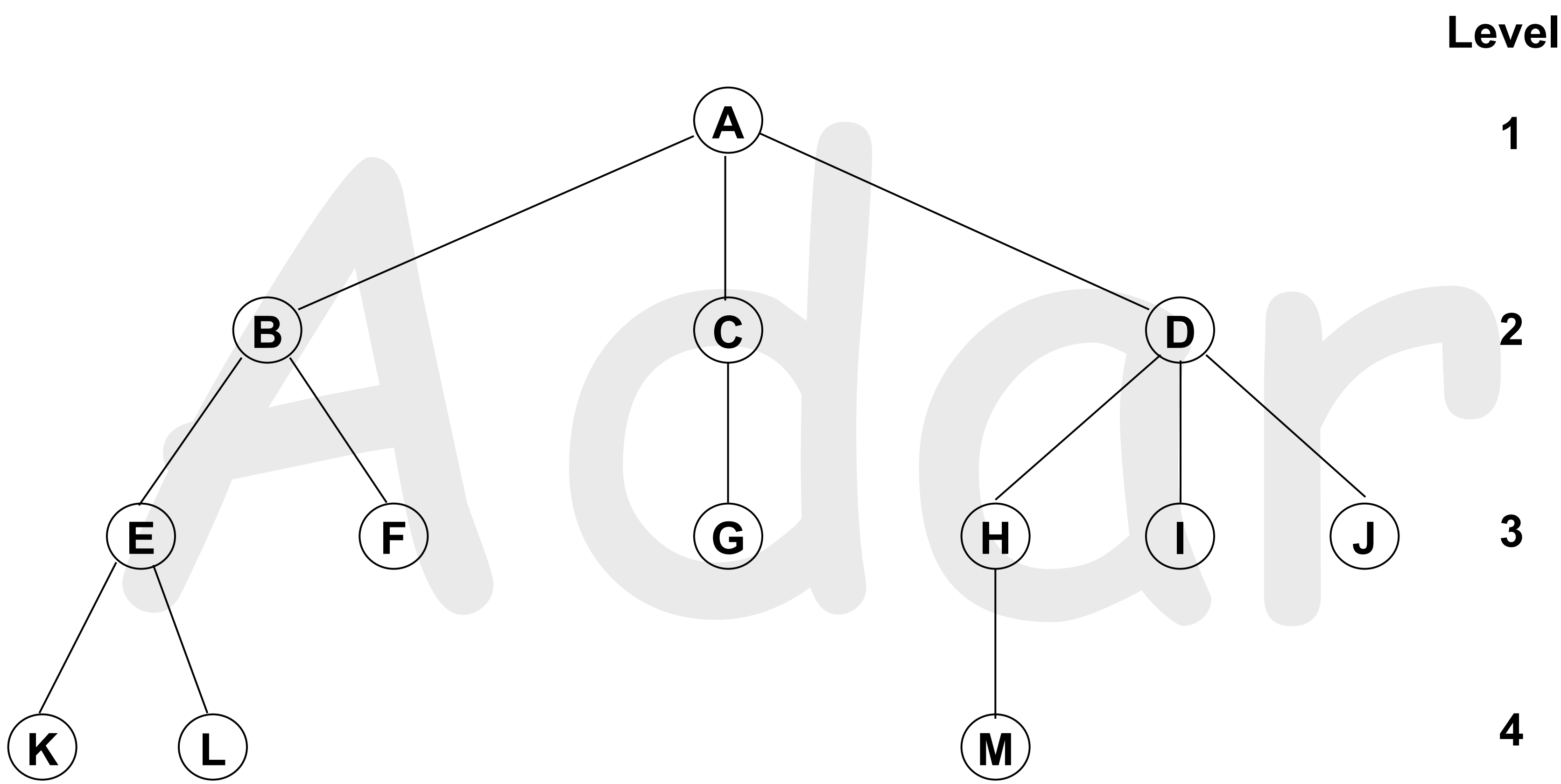
Definition (2/3)

- Number of subtrees of a node ➔ degree
- Nodes with degree 0 ➔ leaf or terminal nodes
- Nodes are not leaf nodes ➔ nonterminal nodes
- The roots of the subtrees of X are the children of X,
- And X is the parent of its children
- Children of the same parent are siblings

Definition (3/3)

- The max degree of the nodes in the tree ➔ degree of the tree
- All the nodes along the path from the root to a specific node ➔ ancestors of that node
- The level of the root node is 1
  - if a node is at level n, its children are at level n+1
- The max level of the nodes in the tree ➔ height or depth of the tree

Illustrated Example



Tree Representation – K-ary Node

- For a tree of degree k, the node structure can be

Data	**Child 1**	**Child 2**	**Child 3**	**Child 4**	**…**	**Child k**

- However, if T is a K-ary tree with n nodes, then $n(k-1)+1$ of the nk child fields are 0, $n \geq 1$

Proof:
1. each mode has k child fields ➔ n*k fields in total
2. only n – 1 nodes are pointed (except the root) ➔ (n – 1) fields actually in use
3. Hence, nk – (n – 1) = n(k – 1) + 1 fields are 0, i.e., null pointers, not in use

- What if K = 1? K = 2?
- What if K is a large number?
  - significant memory waste

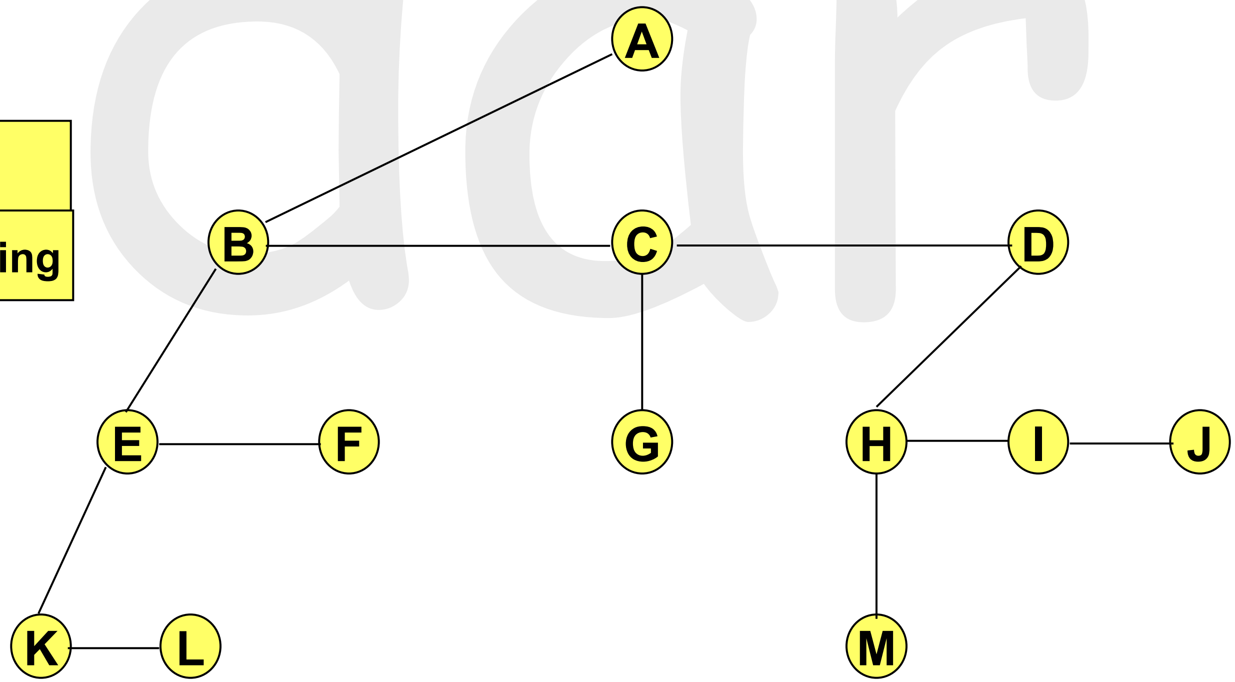
Adapted from Prof. Juinn-Dar Huang

Trees

Left Child – Right Sibling

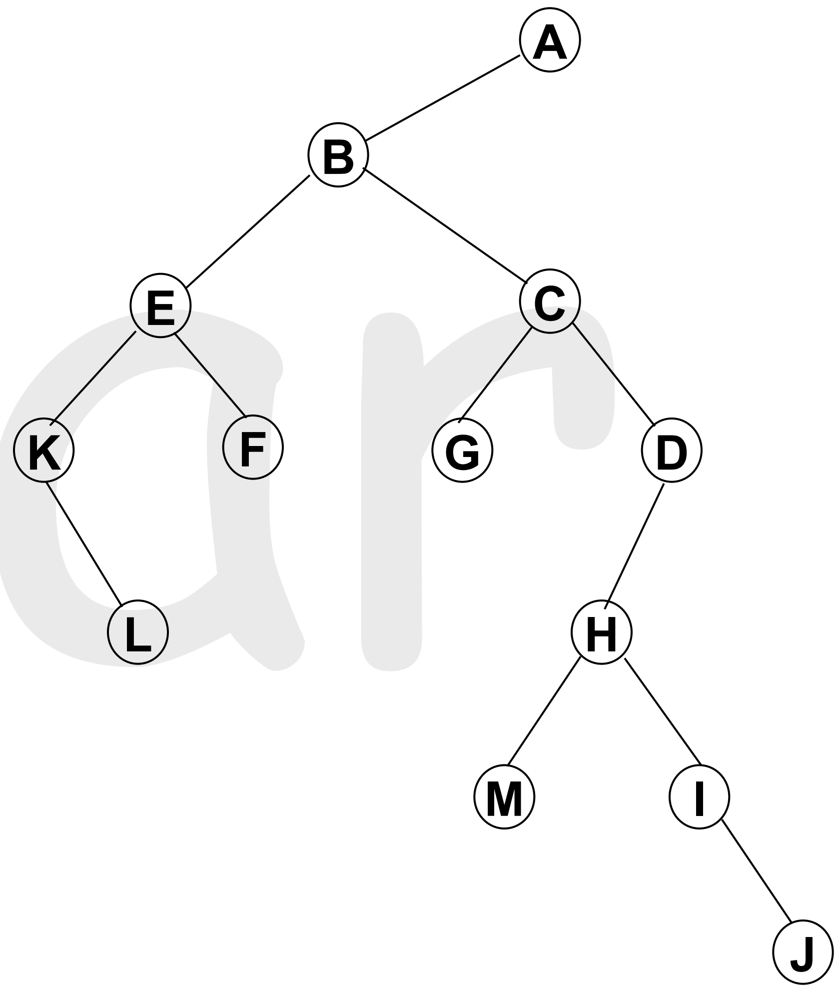
- Left child – right sibling representation
  - each node has 2 pointer fields only
  - "left child" points to its leftmost child if any
  - "right sibling" points to its closet right sibling if any

data	
left child	right sibling



Degree-Two Tree

- Rotate the right sibling 45º clockwise
- Now become left child – right child tree
  - also referred as ***binary tree***
- Degree-two trees
  - why we try to represent an arbitrary tree in a degree-2 tree (or in a binary tree)?
  - hint: memory usage



Binary Tree

- 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

- Again, a recursive definition
- A node can have at most 2 branches (degree ≤ 2)
- left child and right child must be distinguished
- A binary tree can be empty (have 0 node)

Tree vs. Binary Tree

Differences

- No tree has 0 node while a binary tree can be empty
- We distinguish the order of its children in a binary tree while we don't in a tree

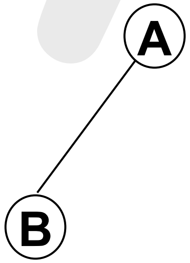


Fig. 1

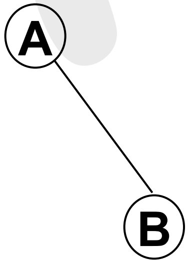
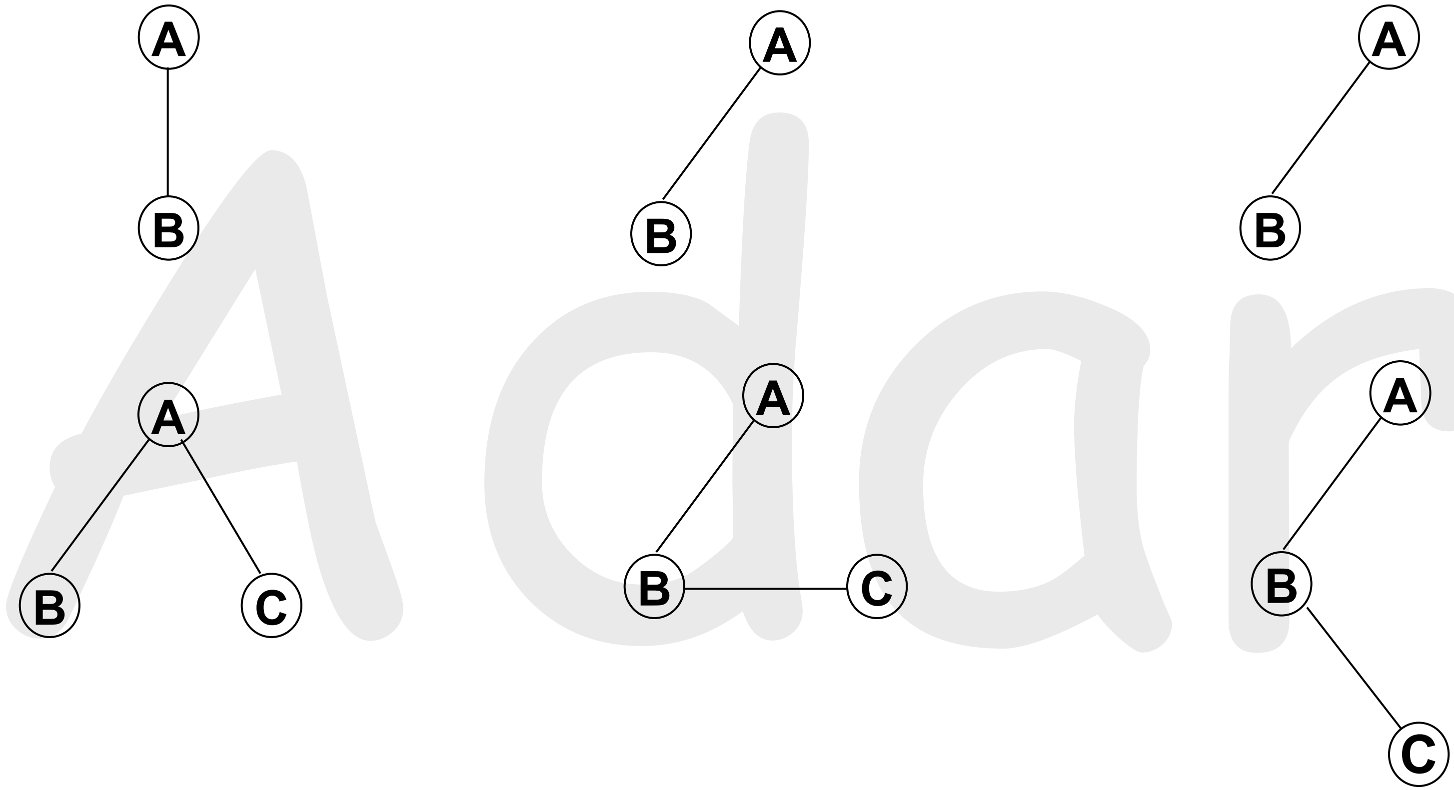


Fig. 2

If Fig. 1 & 2 represent trees,
➔ they represent 2 identical trees

If Fig. 1 & 2 represent binary trees,
➔ they represent 2 different binary trees

Various Representations

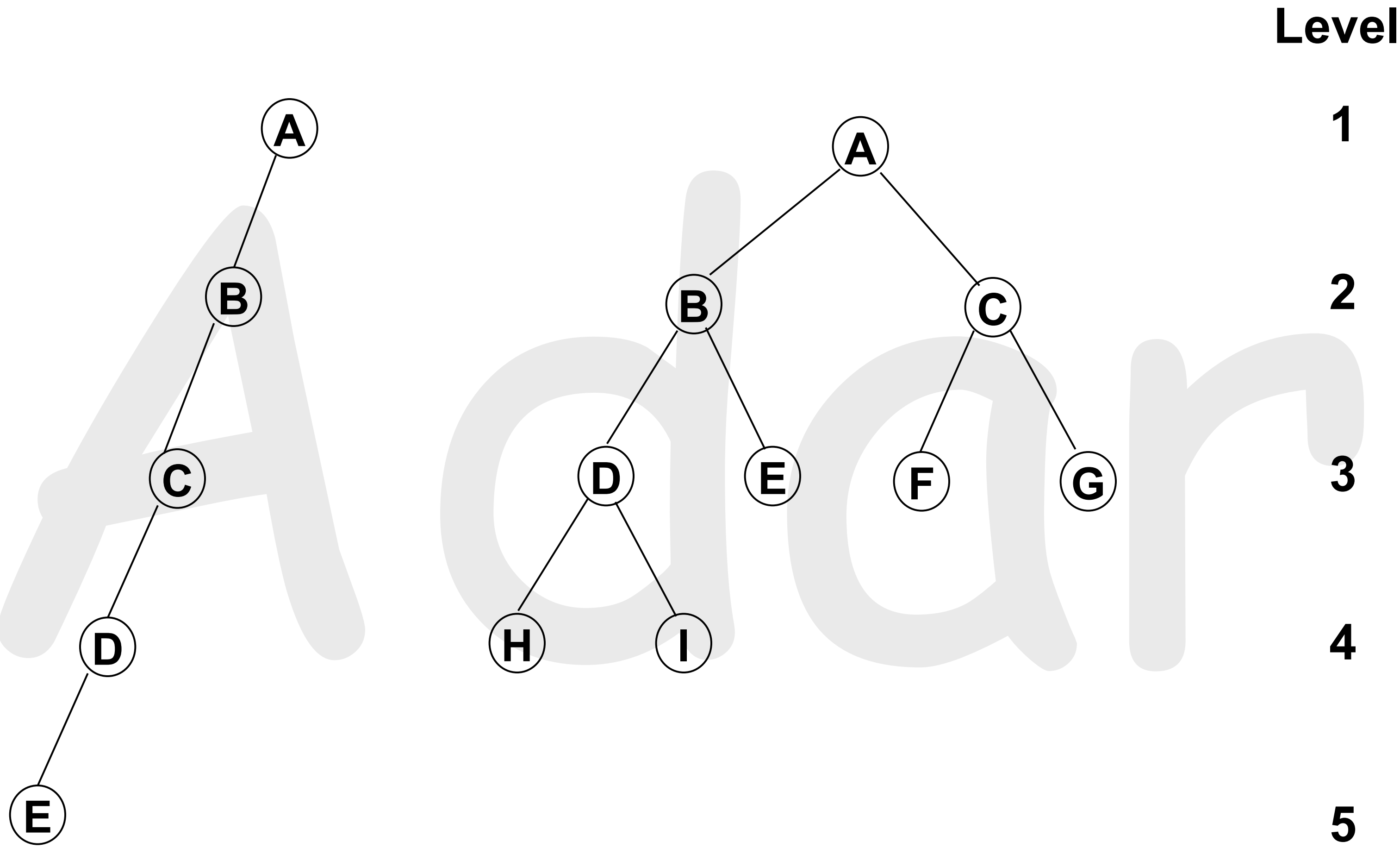


tree

left child - right sibling

binary tree

Skewed and Complete Binary Trees



Skewed binary tree

Complete binary tree

Array Representation

Linked List Representation

Properties of Binary Trees

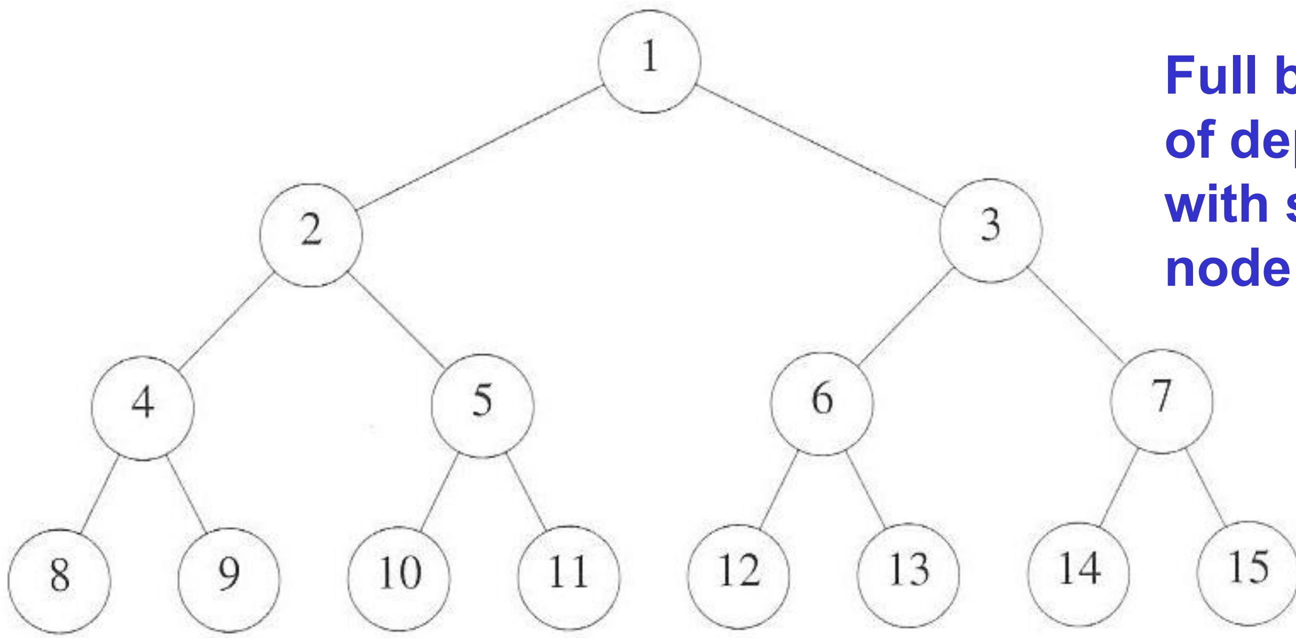
- The max number of nodes at level n is 2^{n-1}, $i \geq 1$
- The max number of nodes in a binary tree of depth k is $2^k - 1$, $k \geq 1$
- For a non-empty binary tree,
  - if n_0 is the number of leaf nodes, and
  - n_2 is the number of nodes of degree 2
  - ➔ $n_0 = n_2 + 1$
  - Proof: Let n_1 is the number of nodes of degree 1

1. Number of node $n = n_0 + n_1 + n_2$
2. Number of used branches $B = n - 1$
3. B is also equal to $2n_2 + n_1$

$n_0 + n_1 + n_2 - 1 = 2n_2 + n_1$ ➔ $n_0 = n_2 + 1$

Full Binary Tree

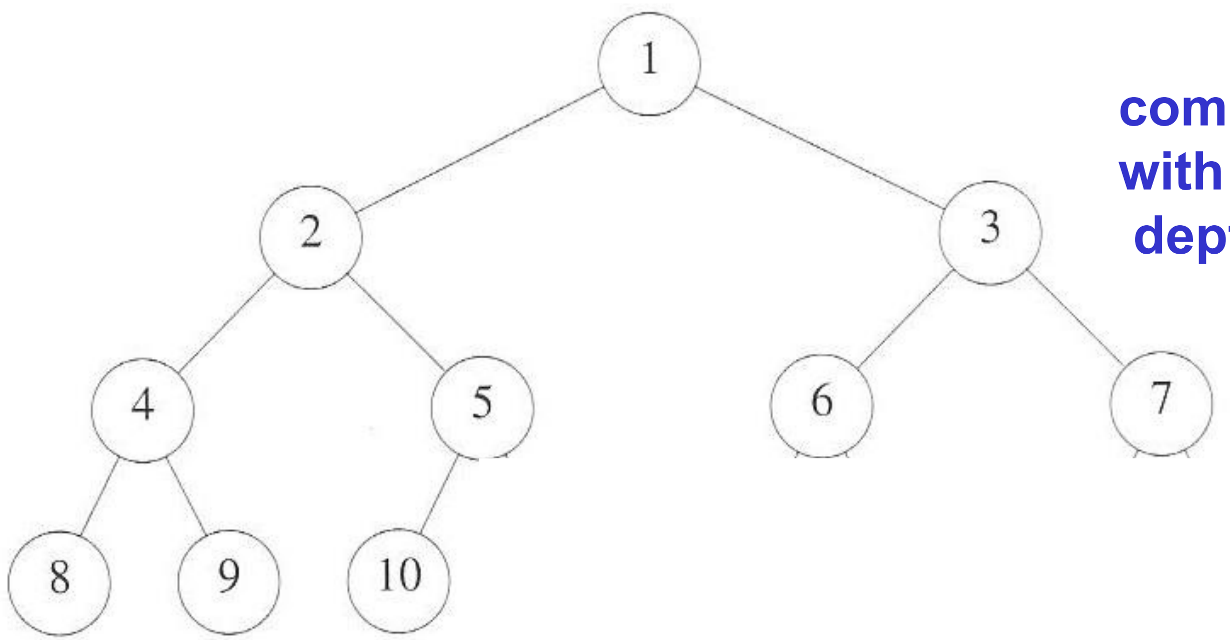
- A **full** binary tree of depth k
  - it has $2^k - 1$ nodes
    - max number of nodes a depth-k binary tree can possibly have



Full binary tree of depth 4 with sequential node numbers

Complete Binary Tree

- 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
- The depth of a complete binary tree with n nodes is
  - $\lceil \log_2(n+1) \rceil$



complete binary tree with 10 nodes and depth 4

Array Representation (1/2)

- Use an array to store nodes
  - nodes are indexed by their unique numbers
- Assume array[1] ~ array[n] are used
  - in C++, it means array[0] is not used intentionally
- Hence, for a node i
  - parent(i) is at $\lfloor i / 2 \rfloor$, if i $\neq$1; if i == 1 → root has no parent
  - leftchild(i) is at 2*i if 2i $\leq$ n; or i has no left child
  - rightchild(i) is at 2*i+1 if 2*i+1 $\leq$ n; or i has no right child

Array Representation (2/2)

[1]	A
[2]	B
[3]	—
[4]	C
[5]	—
[6]	—
[7]	—
[8]	D
[9]	—
⋮	⋮
[16]	E

Skewed tree

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

Complete tree

[Corresponding Trees]

Drawback of Array Representations

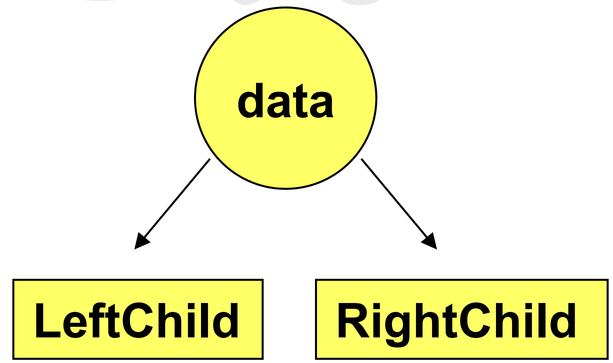
- Again, it's hard to dynamically re-size
- Inefficient memory usage
  - for a skewed binary tree of depth k
  - needs an array of **2^k-1** nodes to store just **k** nodes

We need other alternatives

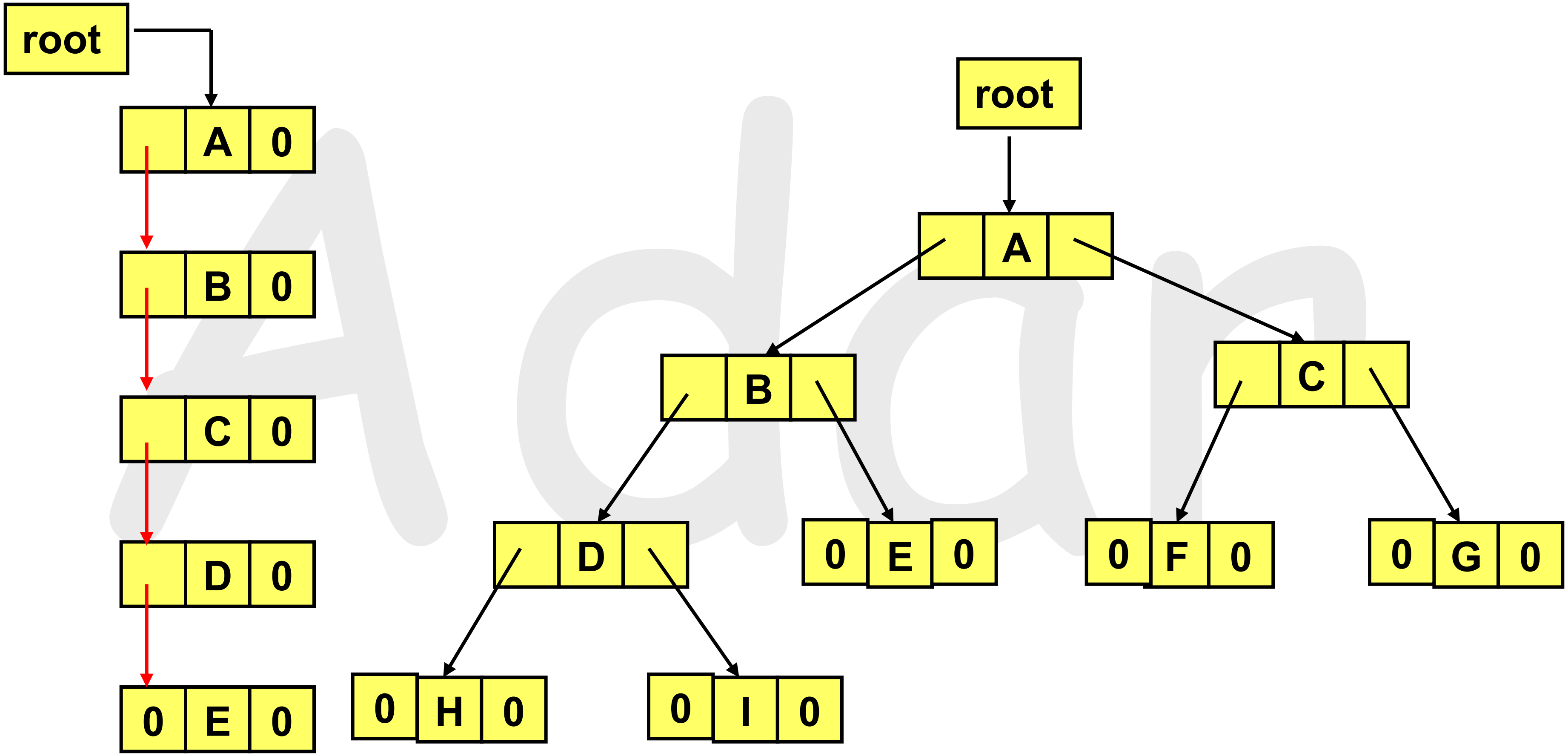
Linked List Representation (1/2)

```
class Tree;

class TreeNode {
friend class Tree;
   char data;
   TreeNode *LeftChild;
   TreeNode *RightChild;
};

class Tree {
   TreeNode *root;
public:
   // operations
};
```

LeftChild	data	RightChild



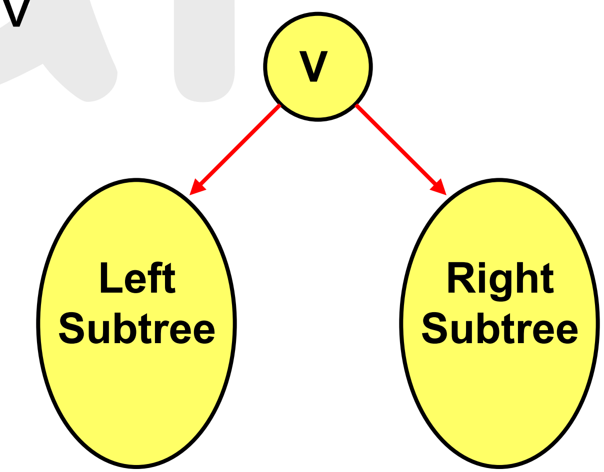
Linked List Representation (2/2)



Corresponding Trees

Binary Tree Traversal (1/2)

- Binary tree traversal
  - visit each node in the tree exactly **once**
- A full traversal produces a **linear** order for the nodes in a binary tree
- There are at least **6** systematic ways to traverse a binary tree
  - VLR, LVR, LRV, VRL, RVL, and RLV
  - sense the smell of recursion?



Binary Tree Traversal (2/2)

- Convention: always traverse left before right
  - 6 ways reduce to 3 ➔ VLR, LVR, LRV

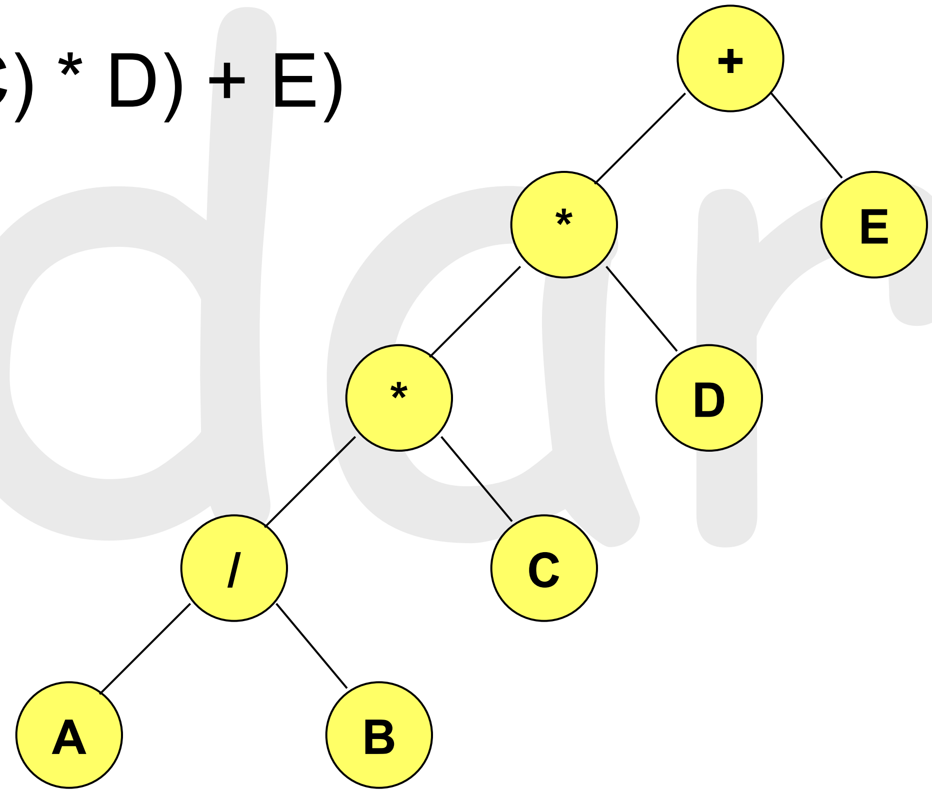
- VLR ➔ preorder
- LVR ➔ inorder
- LRV ➔ postorder

- There is a natural correspondence between pre/in/post-order traversal and pre/in/post-fix forms of expressions
  - remember Chap 3?

Arithmetic Expression

- Binary tree for an arithmetic expression

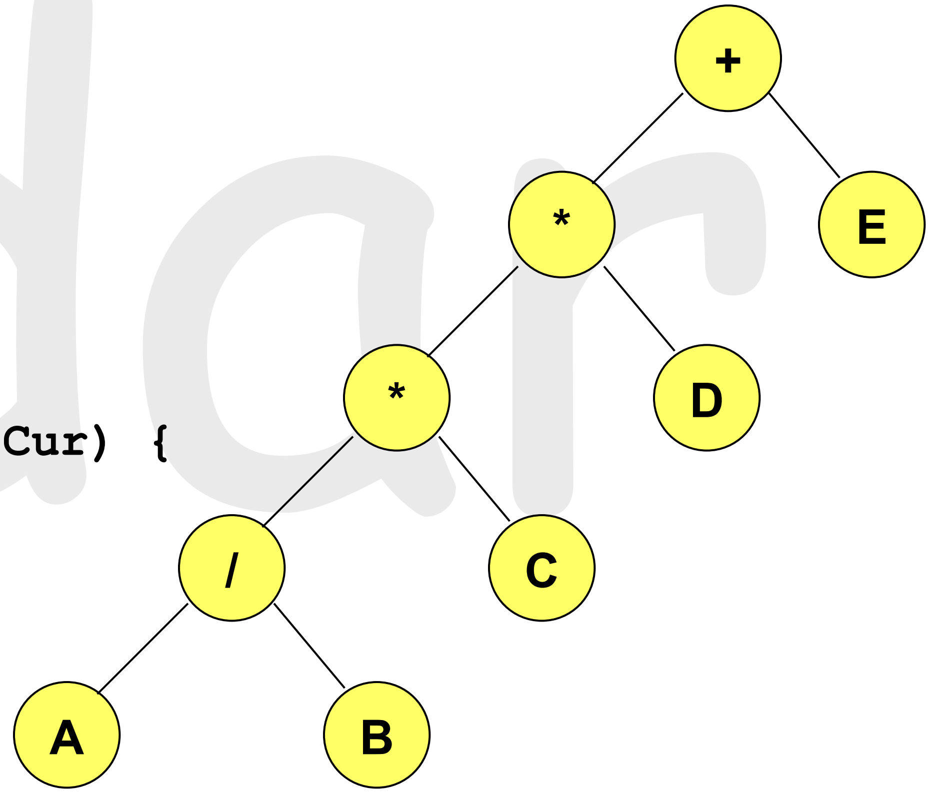
Ans = (((A/B) * C) * D) + E)



Inorder Traversal

- LVR fashion
- Infix expression ➔ A / B * C * D + E

```
void Tree::inorder() {
  inorder(root);
}

// function overloading
void Tree::inorder(TreeNode *Cur) {
  if(Cur) { // not NULL
    inorder(Cur->LeftChild);
    cout << Cur->data;
    inorder(Cur->RightChild);
  }
}  // Recursion
```



Trace Example of Inorder Traversal

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

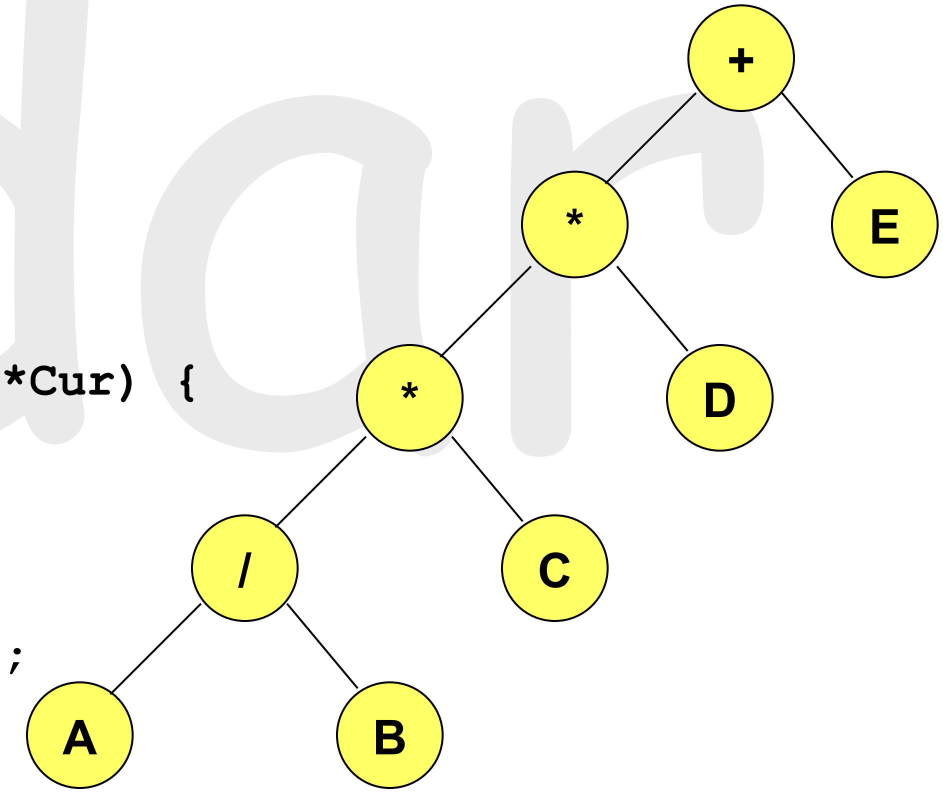
Adapted from Prof. Juinn-Dar Huang

Trees

Preorder Traversal

- VLR fashion
- prefix expression ➔ + * * / A B C D E

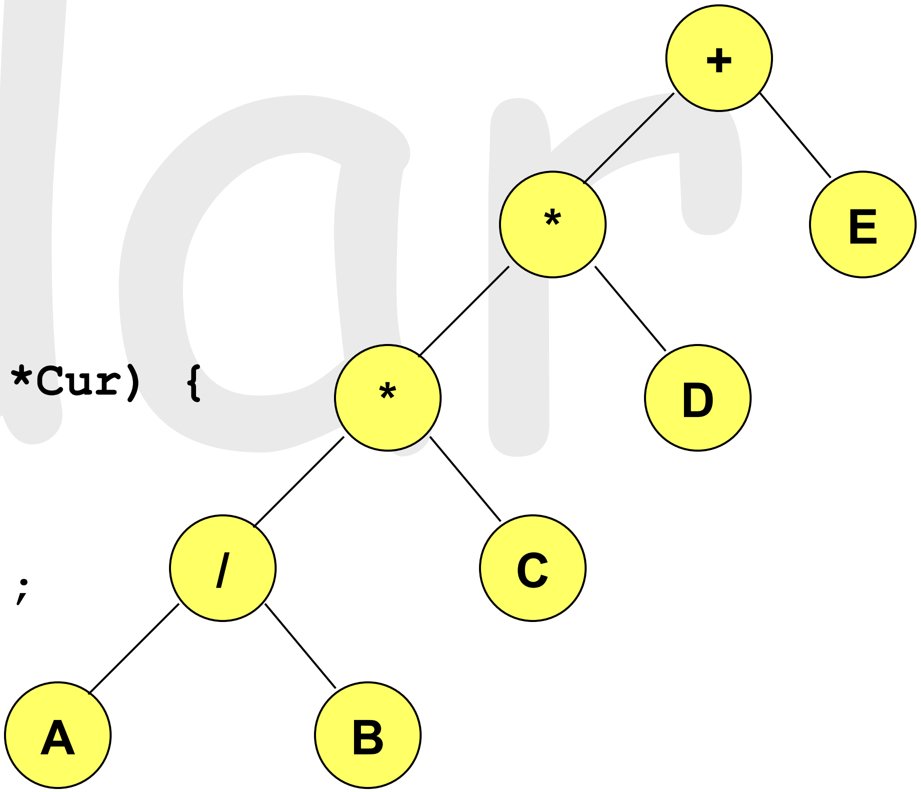
```
void Tree::preorder() {
  preorder(root);
}

// function overloading
void Tree::preorder(TreeNode *Cur) {
  if(Cur) { // not NULL
    cout << Cur->data;
    preorder(Cur->LeftChild);
    preorder(Cur->RightChild);
  }
}  // Recursion
```



Postorder Traversal

- LRV fashion
- postfix expression ➔ A B / C * D * E +

```
void Tree::postorder() {
  postorder(root);
}

// function overloading
void Tree::postorder(TreeNode *Cur) {
  if(Cur) { // not NULL
    postorder(Cur->LeftChild);
    postorder(Cur->RightChild);
    cout << Cur->data;
  }
}  // Recursion
```



Non-Recursive Inorder Traversal

```
void Tree ::NonrecInorder()
// nonrecursive inorder traversal using a stack
{
  Stack <TreeNode *>  s;  // declare and initialize stack
  TreeNode *CurrentNode = root;
  while(1) {
      while (CurrentNode) { // move down LeftChild fields
          s.Add(CurrentNode);  // add to stack
          CurrentNode = CurrentNode →LeftChild;
      }
      if (! s.IsEmpty()) { // stack is not empty
          CurrentNode = *s.Delete(CurrentNode); // delete from stack
          cout << CurrentNode →data << endl;
          CurrentNode = CurrentNode →RightChild;
      }
      else break;
  }
}
```

Time Complexity : O(n)

Space Complexity : O(n)

How about non-recursive preorder ?

How about non-recursive postorder ?

Inorder Iterator Class (1/2)

```
// Assumed to be a friend of class TreeNode and Tree
class InorderIterator {
   const Tree& t;
   Stack<TreeNode *> s;
   TreeNode *Cur;
public:
   char* Next();
   InorderIterator(const Tree& tree)
     :t(tree), Cur(tree.root)  // s(DefultSize)
   { }
};
```

Inorder Iterator Class (2/2)

```
char* InorderIterator::Next() {
  while(Cur) {
    s.Add(Cur);
    cur = cur->LeftChild;
  }
  if(! s.IsEmpty()) {
    s.Delete(Cur);
    char& tmp = Cur->data;
    Cur = Cur->RightChild;
    return &tmp;
  }
  else return 0; // no more elements
}
```

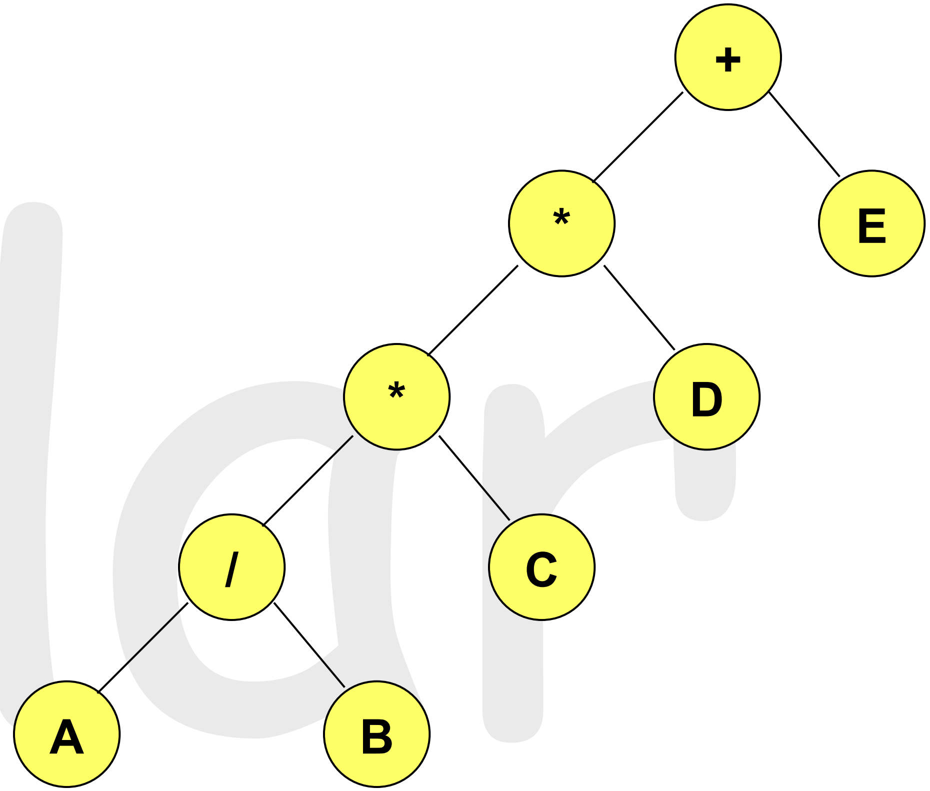
Actually, it's the inner loop of the non-recursive inorder traversal

Level-Order Traversal (1/2)

- For iterative/recursive in/pre/post-order traversal, stacks are required in all cases
- How about traversing a binary tree **level-by-level**?
  - nodes with lower level first
- How to do that? ➔ using **queue** instead of stack

Level-Order Traversal (2/2)

```
void Tree::levelorder() {
  Queue<TreeNode *> q;
  TreeNode *Cur = root;
  while(Cur) {
    cout << Cur->data;
    if(Cur->LeftChild)
      q.Add(Cur->LeftChild);
    if(Cur->RightChild)
      q.Add(Cur->RightChild);

    q.Delete(Cur);  // delete from the head
  }
}
```



Level-Order Traversal ➔ + * E * D / C A B

Duplication

```
// copy ctor
Tree::Tree(const Tree& s) {
  root = copy(s.root);
}

TreeNodes* Tree::copy(TreeNode *orig) {
  if(orig) {
    TreeNode *tmp = new TreeNode;
    tmp->data = orig->data;
    tmp->LeftChild = copy(orig->LeftChild);
    tmp->RightChild = copy(orig->RightChild);
    return tmp;
  }
  return 0; // an empty binary tree
}
```

Equality Test

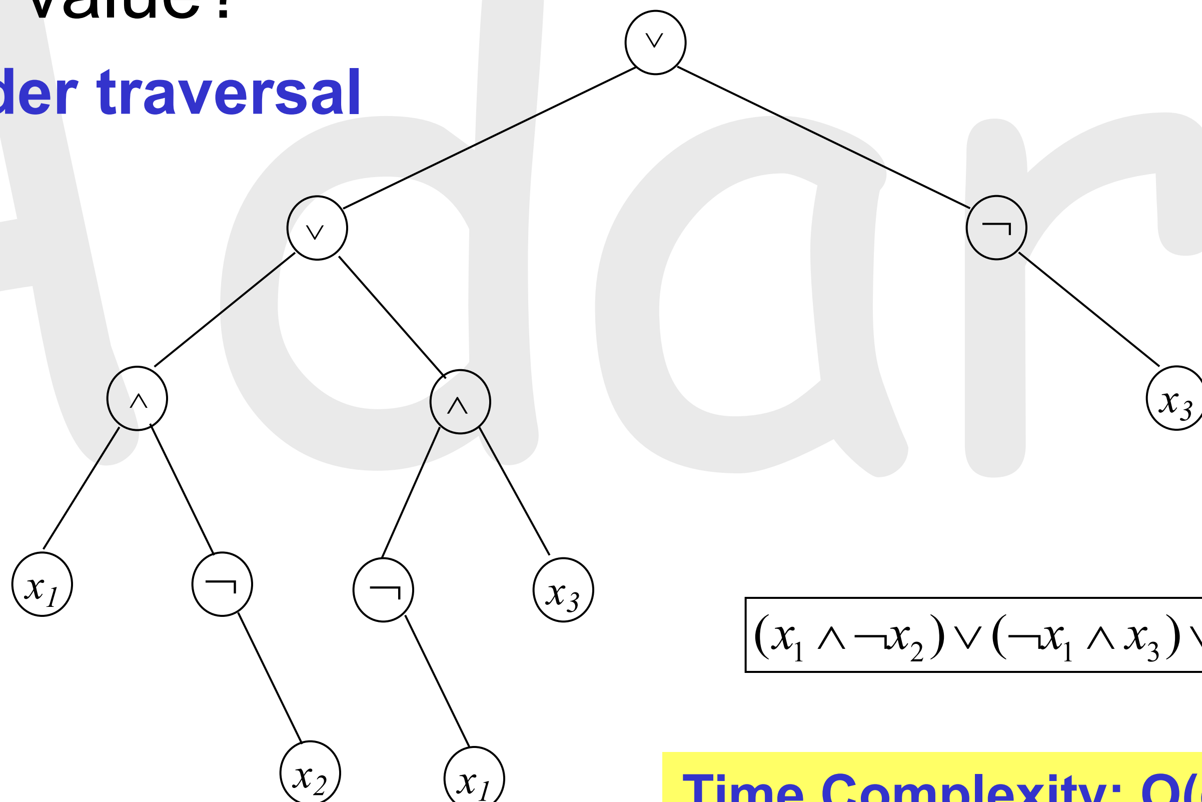
```
// assume the below function is a friend of class Tree
// operator overloading
bool operator==(const Tree& s, const Tree& t)
{  return equal(s.root, t.root);  }

// assume the below function is a friend of class TreeNode
bool equal(TreeNode *a, TreeNode *b) {
  if((! a) && (! b)) return true;  // both a and b are 0

  if(a && b // both a and b are non-0
       && (a->data == b->data) // data is the same
       && equal(a->LeftChild, b->LeftChild) // same left
       && equal(a->RightChild, b->RightChild)) // same right
         return true;
  return false;
}
```

Satisfiability (SAT) Problem (1/4)

- An expression e = x v (y ^ ¬ z)   [x or (y and not z)]
- Find a value combination of x, y, and z such that e is evaluated true
  - positional calculus
  - e.g.,
    x and z are false; y is true ➔ e = F v (T ^ ¬ F) = T

Satisfiability (SAT) Problem (2/4)

- Expression represented in a binary tree
- Given an input combination, how to evaluate the final truth value?
  - **postorder traversal**



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

Time Complexity: $O(g2^n)$

Satisfiability (SAT) Problem (3/4)

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

class SatTree;  // forward declaration
class SatNode {
  friend class SatTree;
  SatNode *LeftChild;
  OpType data;
  bool value;
  SatNode *RightChild;
}
class SatTree {
  SatNode *root;
  void PostOrderEval(SatNode *);
public:
  PostOrderEval();
  void rootvalue() { cout << root->value; }
};
```

Satisfiability (SAT) Problem (4/4)

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

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

Adapted from Prof. Juinn-Dar Huang

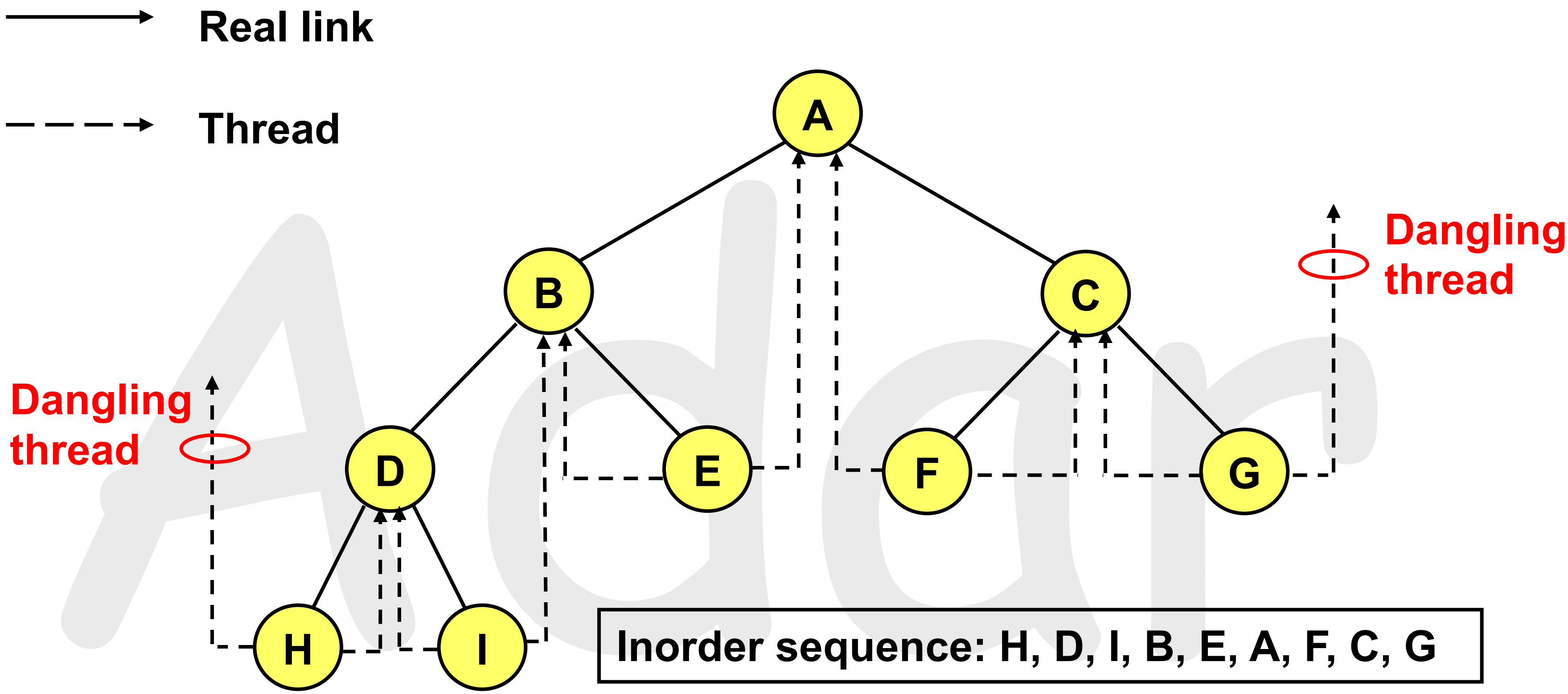
Trees

Threaded Binary Tree (1/3)

- A binary tree with n nodes (n > 0)
  - 2n links in total; n-1 links in use only
- Turn those unused links into threads
  - an original 0 RightChild of Node p re-points to p's inorder successor
  - an original 0 LeftChild of Node p re-points to p's inorder predecessor
- How to distingush a pointer is a real link or just a thread?
  - an extra bool field

LeftThread	LeftChild	data	RightChild	RightThread

Threaded Binary Tree (2/3)



Observation:

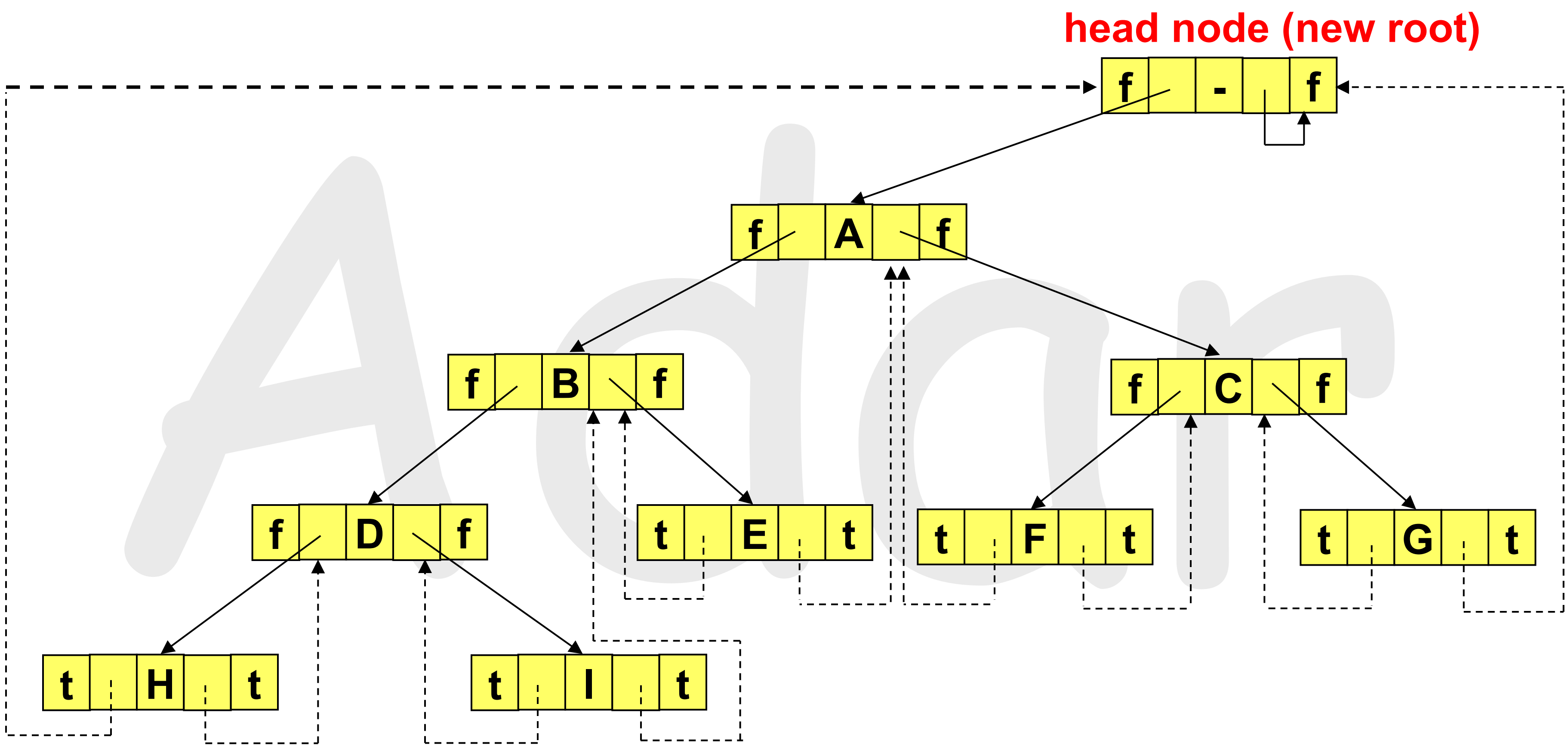
1: If Node p has a **right thread**, the pointer points to p's inorder successor

2: Otherwise, p's inorder successor is obtained by following

a path of left-child links from the right child of p until

a node with a left thread is reached

Threaded Binary Tree (3/3)



Add a dummy head node ➔ no dangling threads

Class Definition (1/2)

```
class ThreadedNode {
  friend class ThreadedTree;
  friend class ThreadedInorderIterator;
  bool LeftThread;
  ThreadedNode *Left;
  char data;
  ThreadedNode *Right;
  bool RightThread;
};

class ThreadedTree {
  friend class ThreadedInorderIterator;
  ThreadedNode *root;
public:
  // ...
};
```

Class Definition (2/2)

```
class ThreadedInorderIterator {
  ThreadedTree& t;
  ThreadedNode* Cur;
public:
  ThreadedInorderIterator(ThreadedTree& tree)
    :t(tree), Cur(tree.root) { }
  Char* Next();
};

char* ThreadedInorderIterator::Next(){
  ThreadedNode *tmp = Cur->Right;
  if(! Cur->RightThread)
    while(! tmp->LeftThread) tmp = tmp->Left;
  cur = tmp;
  if(Cur == t.root) return 0;  // traversal done
  return &Cur->data;
}
```

O(1) space complexity
no stack is required

Adapted from Prof. Juinn-Dar Huang

Trees

Priority Queue

- Priority queue (PQ)
  - each element in a PQ has a priority
  - at any time, an element with arbitrary priority can be inserted into a PQ
  - the element to be deleted is the one with highest priority
    - max PQ
  - the element to be deleted is the one with lowest priority
    - min PQ
- Applications of priority queues ?

ADT MaxPQ

```
template <class T>
struct Element{
  T key;
  // other data members, e.g., int num;
};

template <class T>
class MaxPQ {  // an ABC since it contains pure virtual funcs
public:
  virtual void Insert(const Element<T>&)=0; // pure virtual
  virtual Element<T>* DeleteMax(Element<T>&)=0;
}
```

How to implement a Max PQ?

- Method1: unordered (unsorted) linear list
  - implemented by using either array or list
  - insert time: $\Theta(1)$
  - deletion time: $\Theta(n)$
- Method 2: ordered (sorted) linear list
  - implemented by using either array or list
  - sorted in non-increasing order
  - insert time: $\Theta(n)$
  - deletion time: $\Theta(1)$

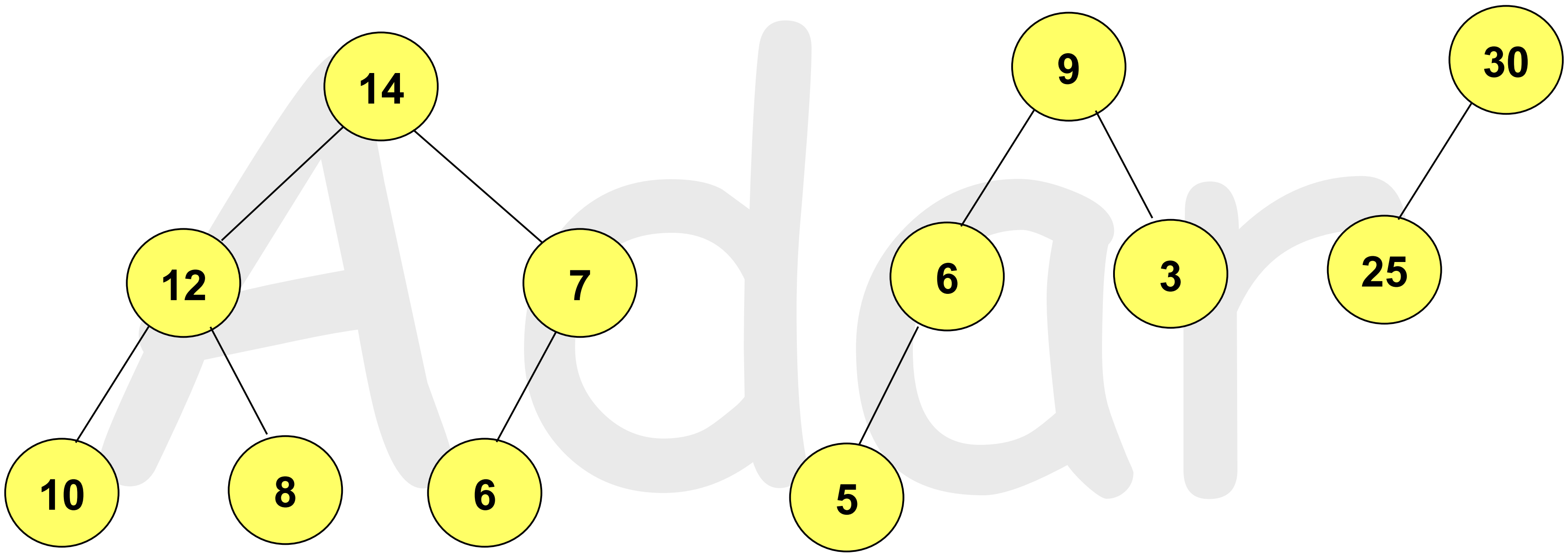
- Any better way? ➔ **Max Heap**

Max Heap

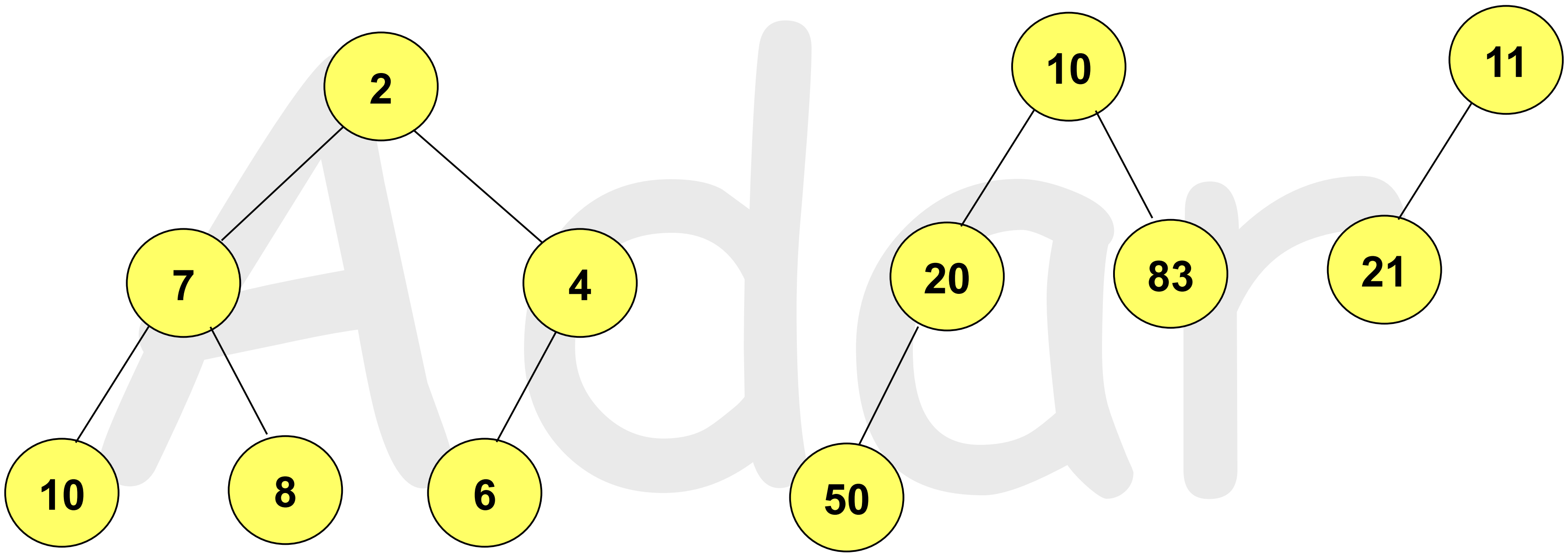
- Max (min) tree
  - a tree in which the key value in each node is no smaller (larger) than the key values in its children (if any)
- Max (min) heap
  - a complete binary tree and also a max (min) tree

- The key in the root of a max (min) tree is the biggest (smallest) key in the tree

Max Heap Examples



Min Heap Examples



Class MaxHeap

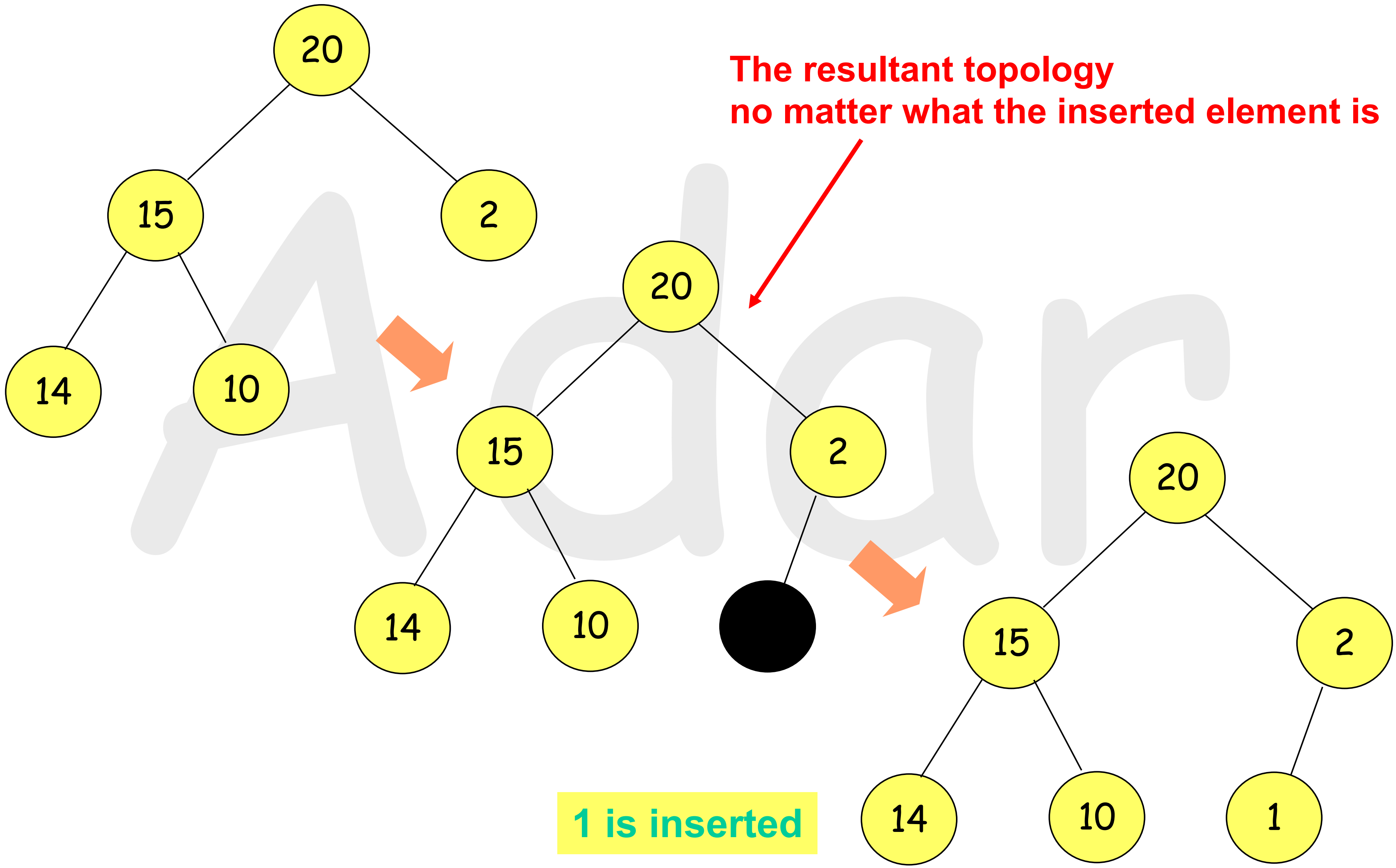
```
template <class T>
class MaxHeap : public MaxPQ<T> {
  Element<T> *heap;
  int n;  // current size
  int MaxSize; // max heap size
public:
  MaxHeap(int sz = DefaultSize);
  // create an empty heap that can hold max sz elements
  bool IsEmpty();
  bool IsFull();
  void Insert(const Element<T>& x);
  // If IsFull() is true then error,
  // else insert x into the heap
  Element<T>* DeleteMax(Element<T>& x);
  // If IsEmpty() is true then return 0,
  // else remove the largest element of the heap,
  // save it to x and return a pointer to x
};
```

How to Store Elements Internally?

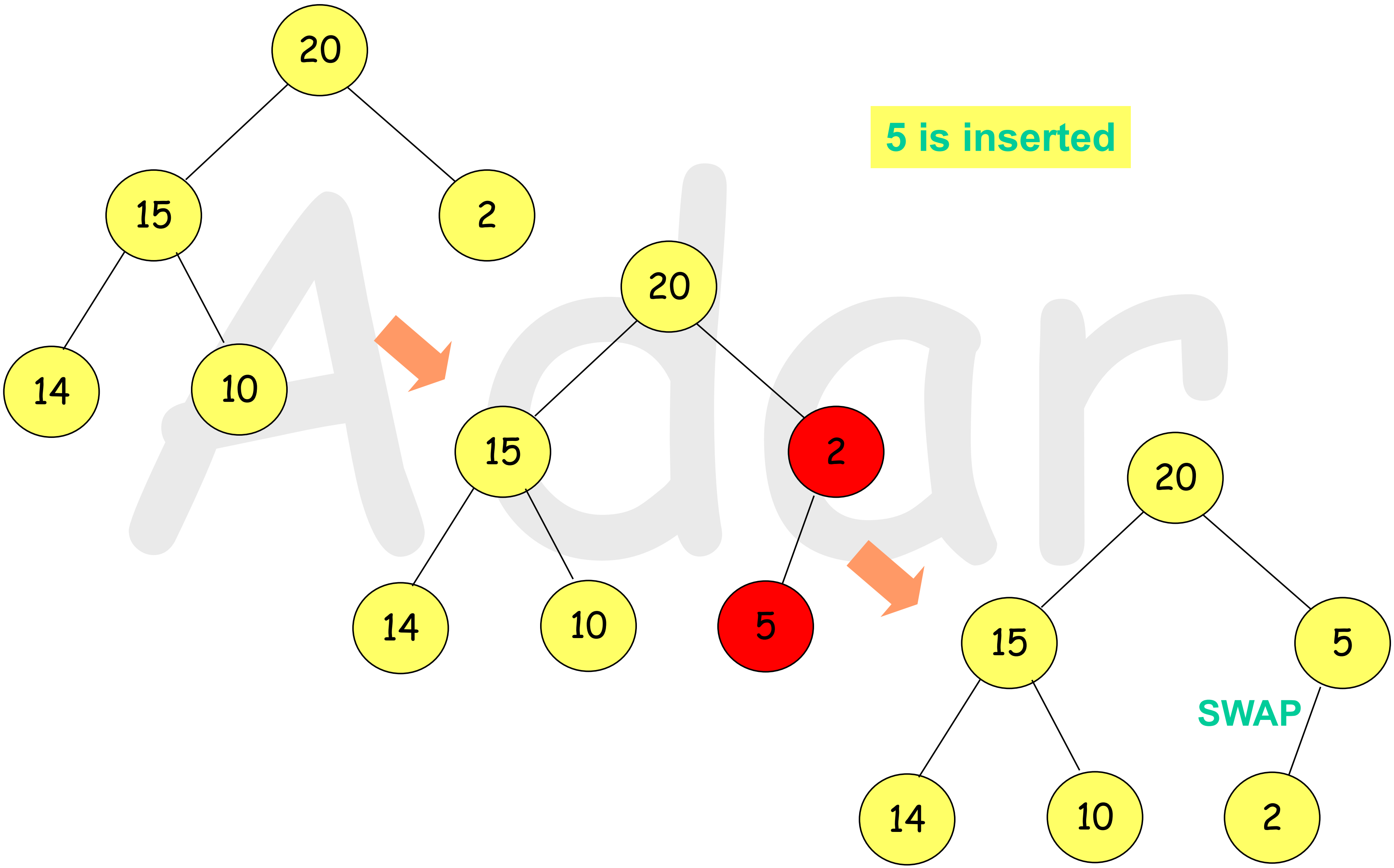
- Heap is a complete binary tree
  - it's OK to use an array to store elements

```
template <class T>
MaxHeap<T>::MaxHeap(int sz)
   :MaxSize(sz), n(0){
  heap = new Element<T>[MaxSize + 1]; // heap[0] is not used
}
```

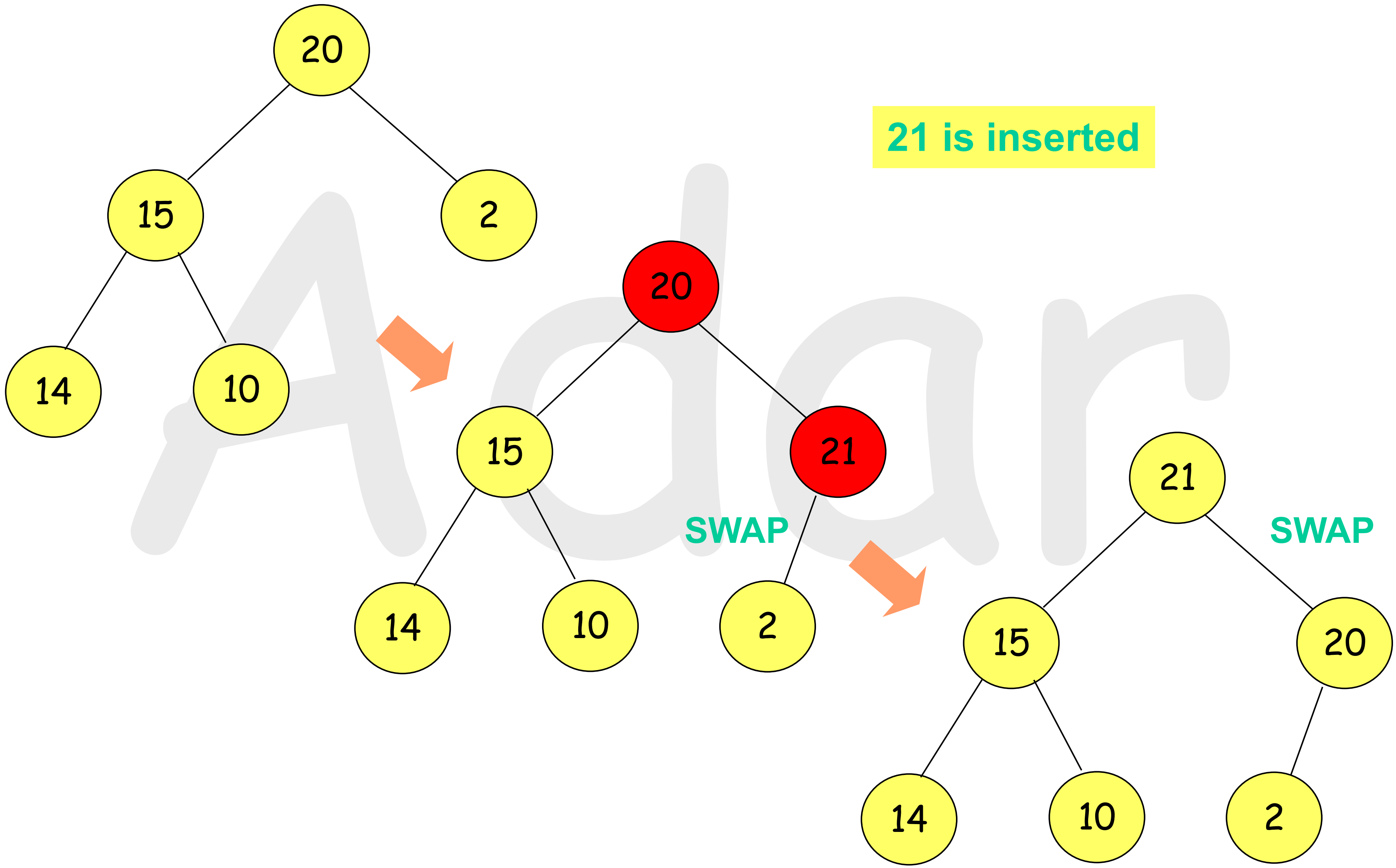
Insertion into a Max Heap (1/4)



Insertion into a Max Heap (2/4)



Insertion into a Max Heap (3/4)

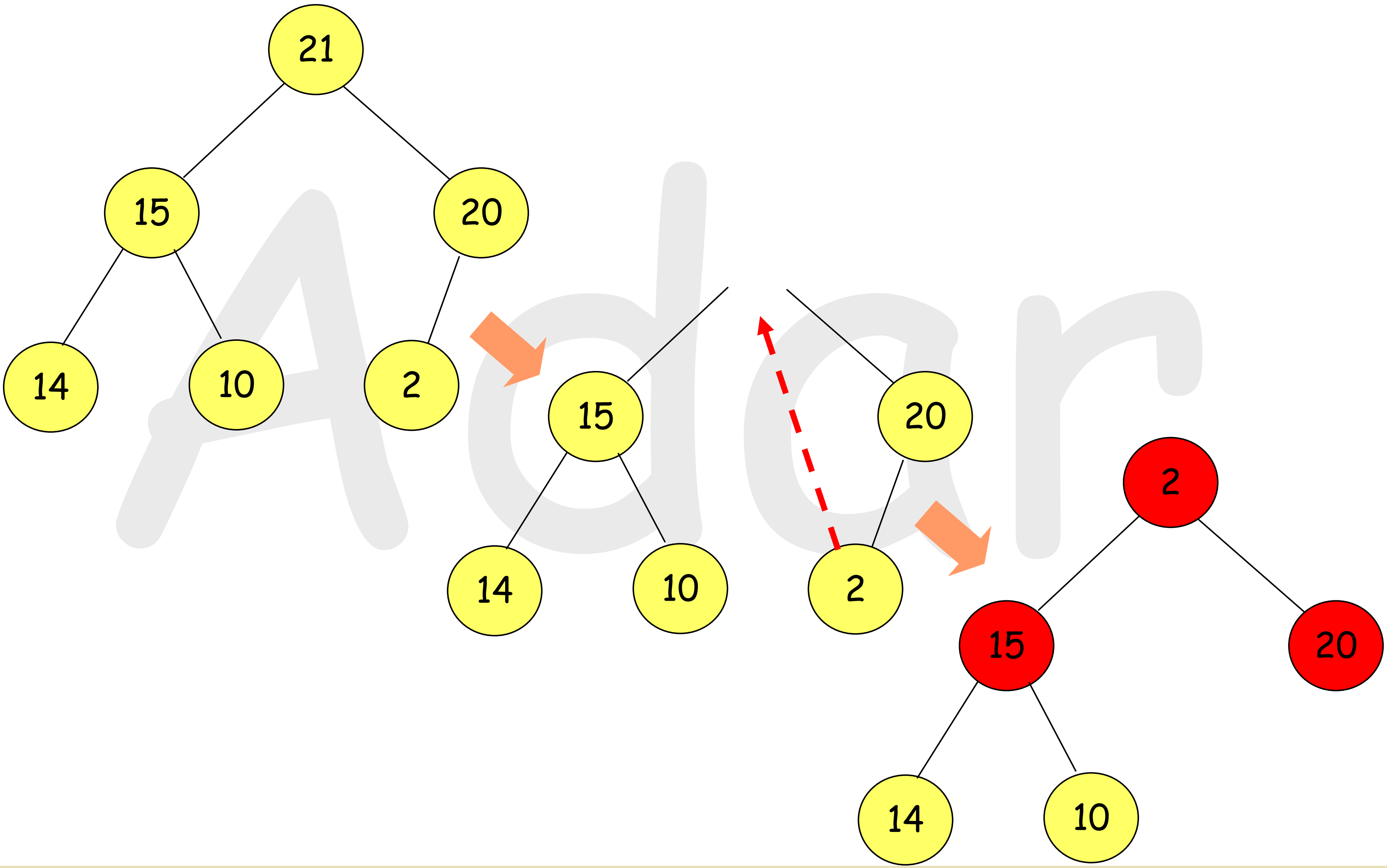


Insertion into a Max Heap (4/4)

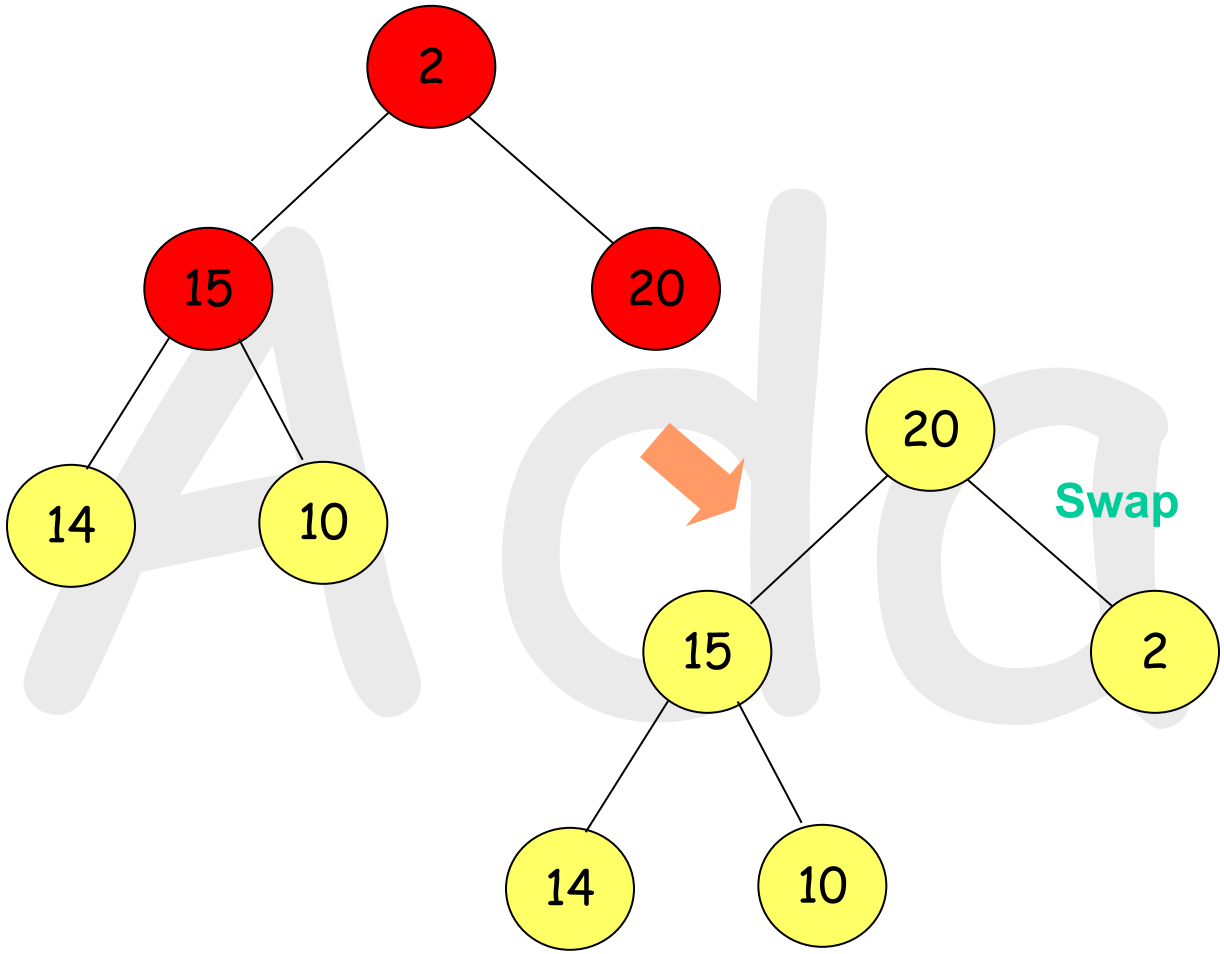
```
template <class T>
void MaxHeap<T>::Insert(const Element<T>& x) {
  if(n == MaxSize)  // heap is already full
  {  HeapFull(); return; }

  ++n;  // increment heap size by 1

  // move down x's ancestors with smaller key
  int i;
  for(i = n; i != 1; i /= 2) {
    if(x.key <= heap[i/2].key) break;  // parent is not smaller
    heap[i] = heap[i/2]; // move down the smaller parent
  }

  heap[i] = x;  // insert x into the right position
}
```

Time Complexity: O(logn)

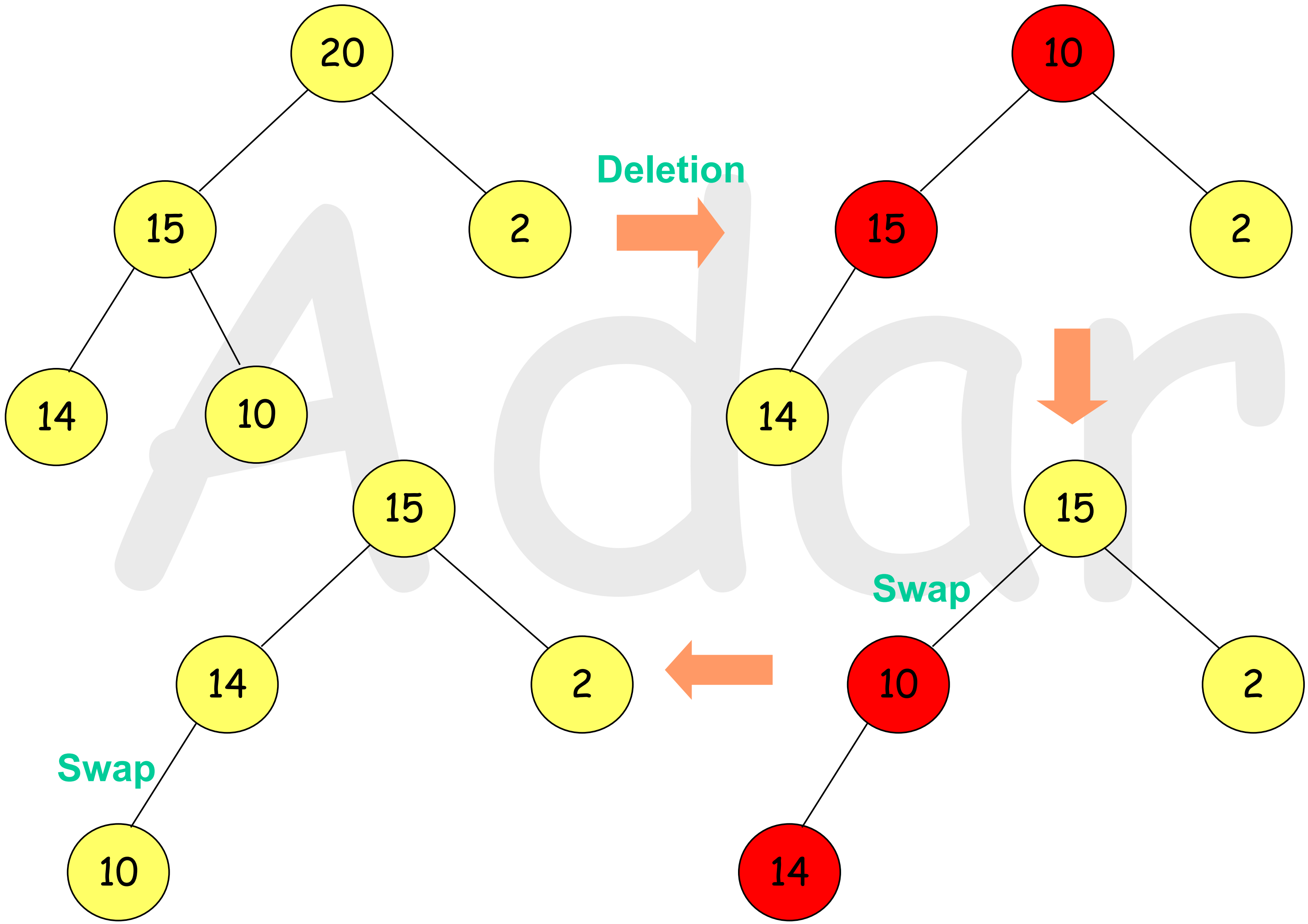
Deletion from a Max Heap (1/4)



Deletion from a Max Heap (2/4)



Deletion from a Max Heap (3/4)



Deletion from a Max Heap (4/4)

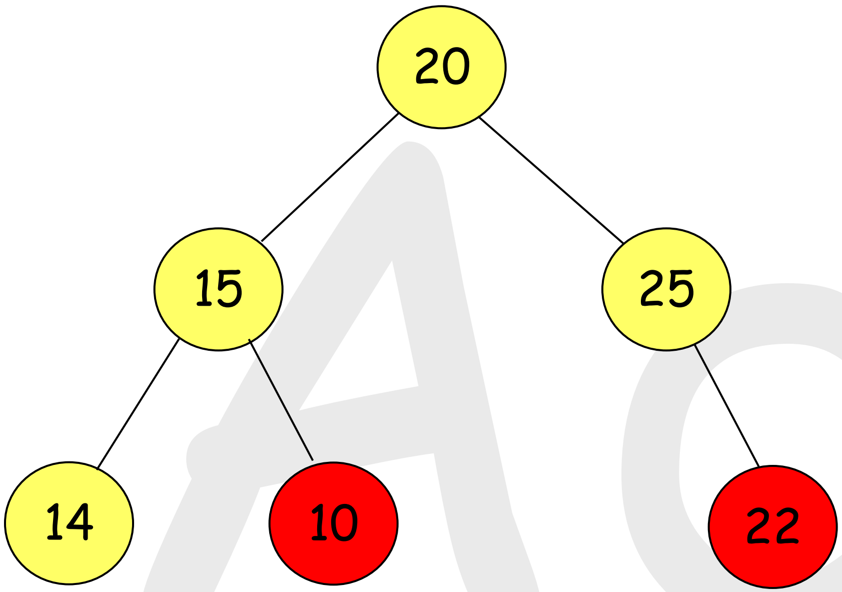
```
template <class T>
Element<T>* MaxHeap<T>::DeleteMax(Element<T>& x) {
  if(! n) // heap is empty
  {  HeapEmpty(); return 0; }
  x = heap[1];
  Element<T>& k = heap[n];
  --n; // decrease the heap size by 1;
  int i;
  for(i = 1, j = 2; j <= n; i = j, j *= 2) {
    if((j < n)  && (heap[j].key < heap[j+1].key))
      ++j; // j points to the bigger child
    if(k.key >= heap[j].key)  break;
    heap[i] = heap[j]; // move child up
  }

  heap[i]= k;
  return &x;
}
```

Time Complexity: O(logn)

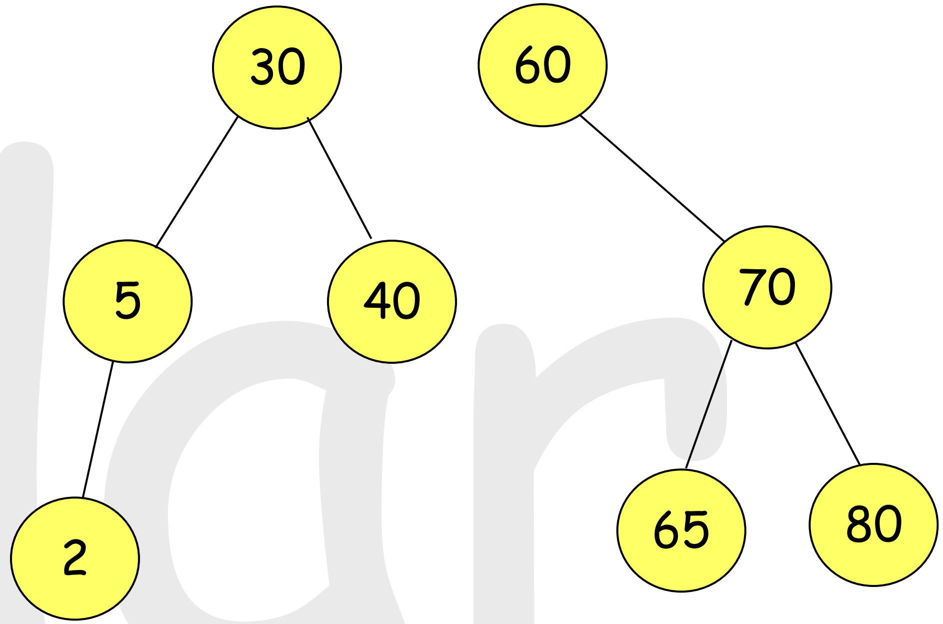
Binary Search Tree (BST)

- Heap is good for priority queues
  - always delete the max/min element
  - cannot delete an element at arbitrary location
  - ➔ used **binary search tree** instead
- Definition
  - is a binary tree
  - may be empty
  - If it's not empty,
    - every element has a key and no 2 elements have the same key
    - keys (if any) in the **left** subtree are **smaller** than the key of the root
    - keys (if any) in the **right** subtree are **bigger** than the key of the root
    - left and right subtrees are also binary search trees (recursive)

BST Examples



Not a BST



BSTs

Apply inorder traversal in a BST
What do you get?

Search in a BST

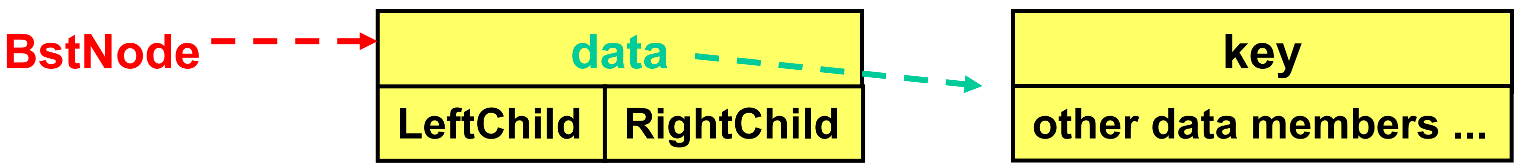
- Search by a given key
  - given key is equal to the key of the current node ➔ found
  - given key is smaller to the key of the current node ➔ left
  - given key is bigger to the key of the current node ➔ right
- Search by a given rank is also fine
  - discuss later

Recursive Search in a BST

```
template <class T>
BstNode<T>* BST<T>::Search(const Element<T>& x) {
  return Search(root, x); // call an overloaded func
}
```

Time Complexity: O(h)

```
template <class T>
BstNode<T>* BST<T>::Search(BstNode<T>* b, const Element<T>& x){
  if(! b) return 0; // not found
  if(x.key == b->data.key) return b; // found
  if(x.key <  b->data.key)
    return Search(b->LeftChild, x);  // search left subtree
  return Search(b->RightChild, x);  // search right subtree
}
```

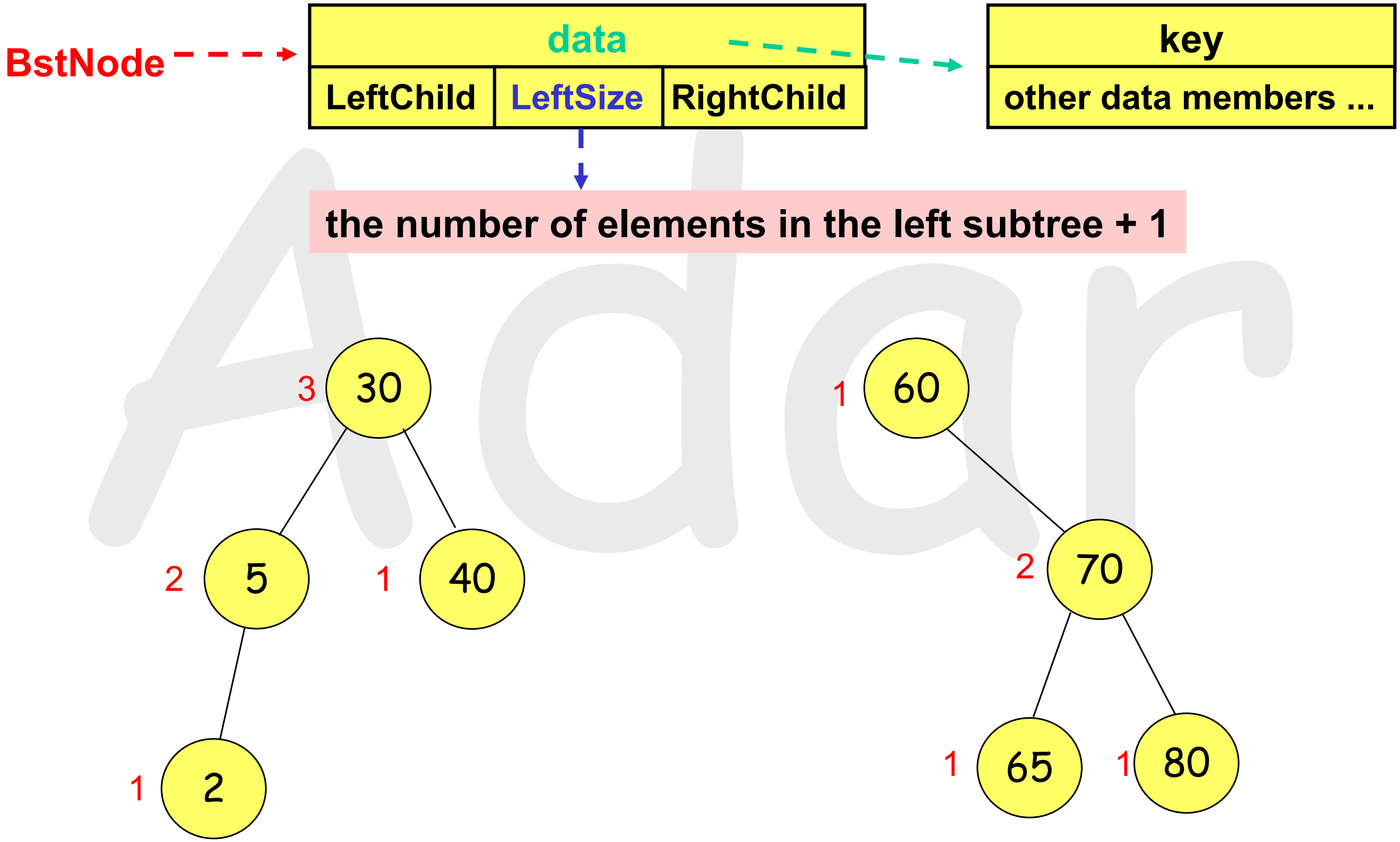


Iterative Search in a BST

```
template <class T>
BstNode<T>* BST<T>::IterSearch(const Element<T>& x) {
  for(BstNode<T>* t = root; t; ) {
    if(x.key == t->data.key) return t; // found
    if(x.key <  b->data.key)
      t = t->LeftChild;  // search left subtree
    else
      t = t->RightChild; // search right subtree
  }
  return 0; // not found
}
```

Time Complexity: O(h)

Search by Rank in a BST (1/2)

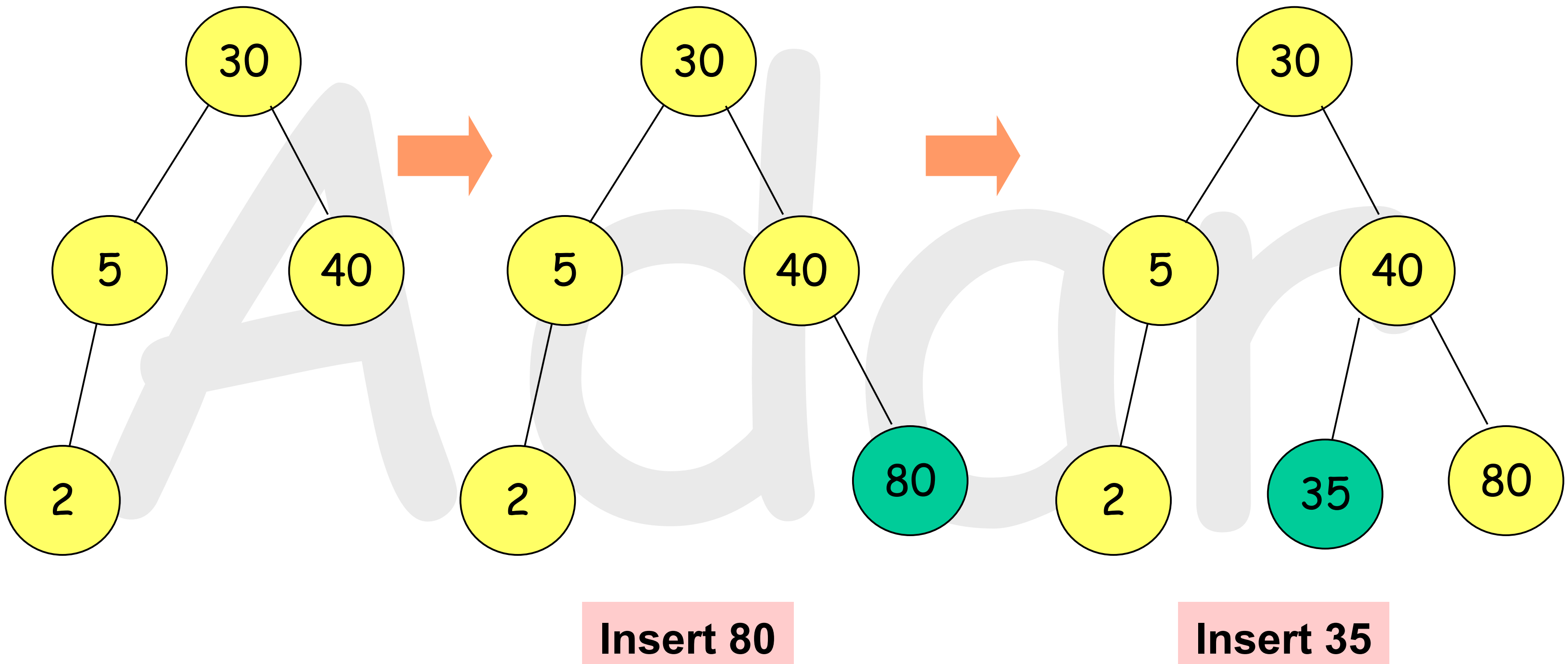


Search by Rank in a BST (2/2)

```
template <class T>
BstNode<T>* BST<T>::Search(int k) {
  for(BstNode<T>* t = root; t; ) {
    if(k == t->LeftSize) return t; // found
    if(k <  t->LeftSize)
      t = t->LeftChild;  // search left subtree
    else {
      k -= t->LeftSize;  // skip the smallest t->LeftSize nodes
      t = t->RightChild; // search right subtree
    }
  }
  return 0; // not found
}
```

Time Complexity: O(h)

Insertion into a BST (1/2)



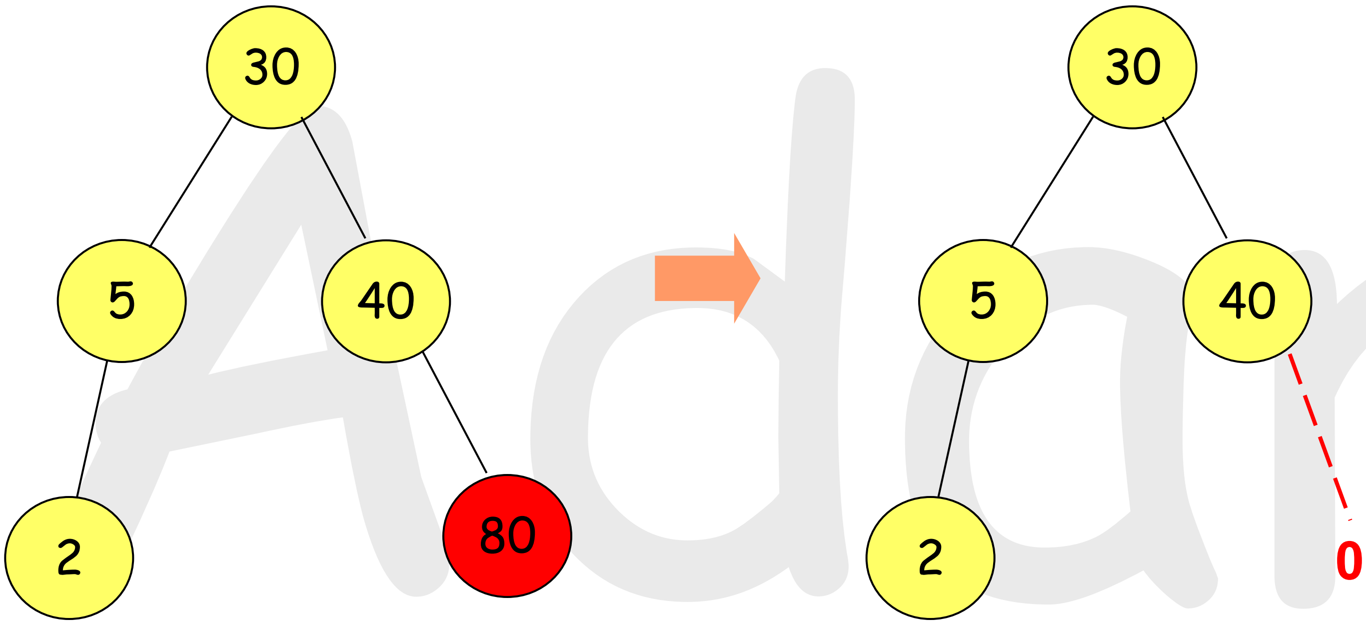
Insertion into a BST (2/2)

Time Complexity: O(h)

```
template <class T>
bool BST<T>::Insert(const Element<T>& x) {
  BstNode<T> *p = root, *q = 0;
  while(p) {
    q = p;
    if(x.key == p->data.key) return false; // an existing key
    if(x.key <  p->data.key)
      p = p->LeftChild;  // move to left subtree
    else
      p = p->RightChild; // move to right subtree
  }
  p = new BstNode<T>;
  p->LeftChild = p->RightChild = 0; p->data = x; // make a copy
  if(! root) root = p;  // an empty BST originally
  else if(x.key < q->data.key) q->LeftChild = p;
  else q->RightChild = p;
  return true;
}
```

Deletion from a BST (1/3)

- Delete a leaf node

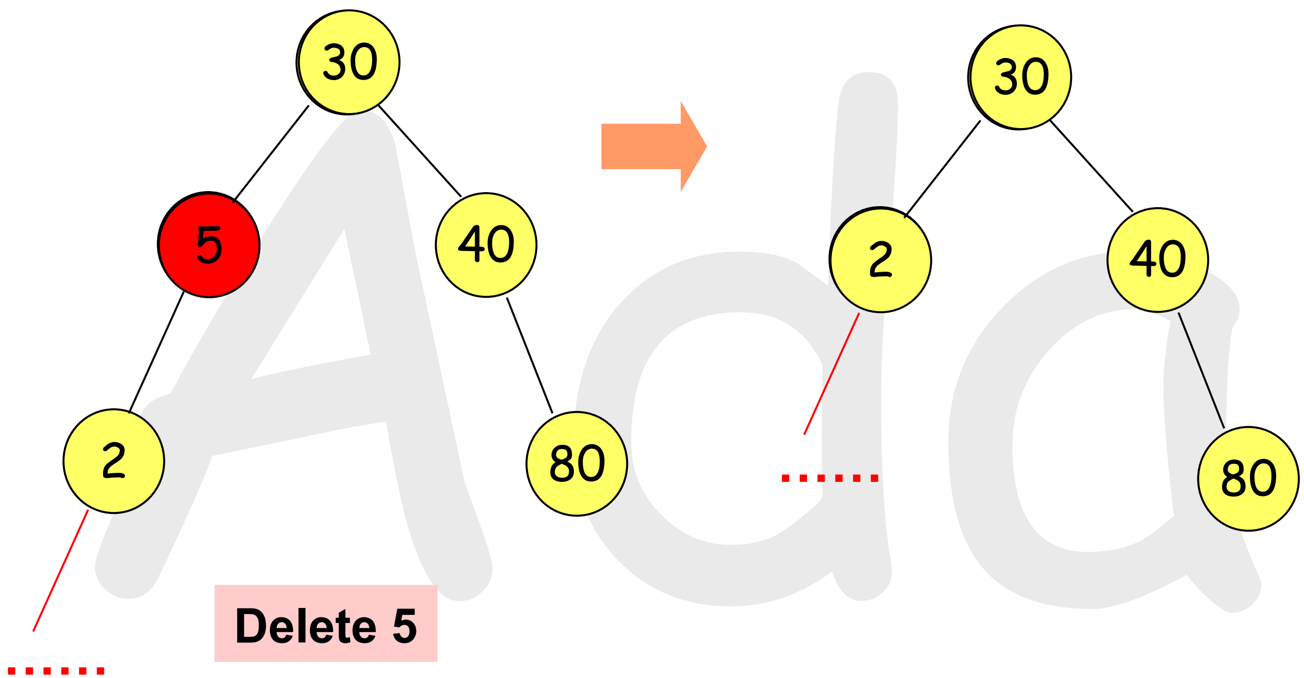


Delete 80

1. Delete the leaf node
2. Set the corresponding link, either LeftChild or RightChild, to 0

Deletion from a BST (2/3)

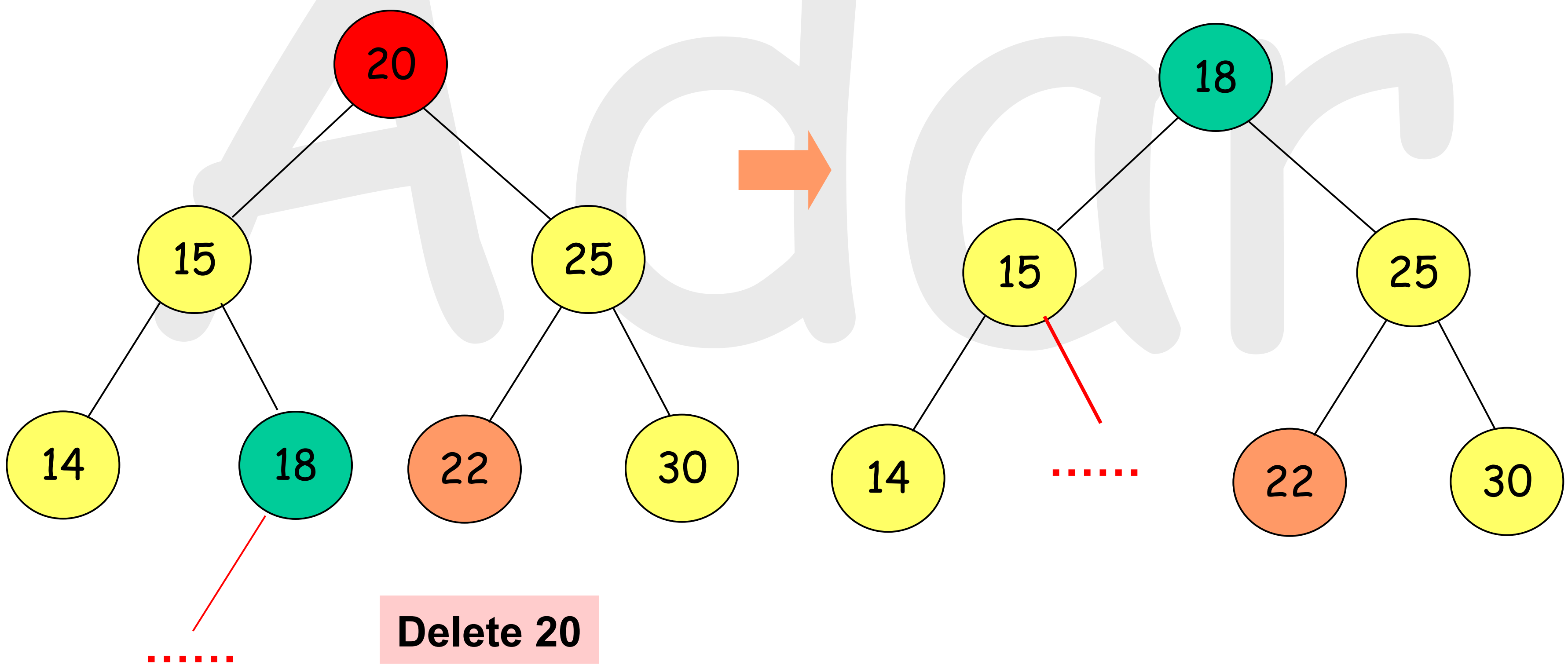
- Delete a node with only one child



1. Delete the specific node
2. Use the single-child, either LeftChild or RightChild, to take place of the deleted node

Deletion from a BST (3/3)

- Delete a node with 2 children
  - replace it by the largest node in its left subtree, or
  - replace it by the smallest node in its right subtree



Height of a BST

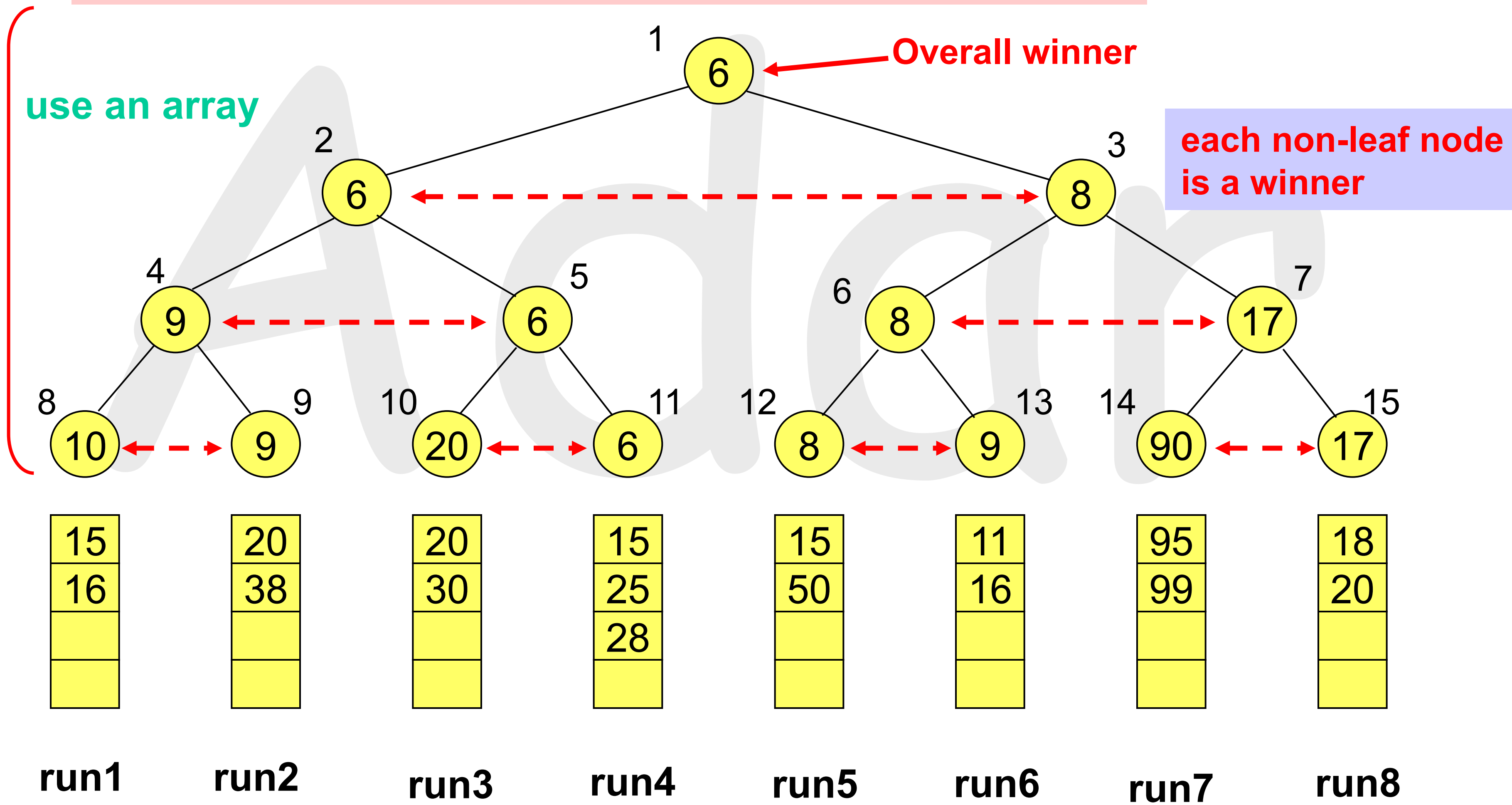
- The height of a BST with n nodes can be as large as n
  - a skewed binary tree
  - how could the worst case happen?
  - degenerate into a linked list
  - time complexity O(h) ➔ O(n)
- In the average case
  - insertions and deletions are made randomly
  - the height of a BST is O(logn) on average
- How to avoid the worst case?
- **Balanced search trees**
  - search trees with a worst-case height of O(logn)
  - such as AVL, 2-3, 2-3-4, red-black, discussed in Chap 10

Selection Trees

- Assume
  - k ordered sequences, named runs, are to be merged into a single ordered sequence
  - each run consists of certain records and is sorted in non-increasing/decreasing key value
  - n is the number of total records in k runs
- Trivial method
  - use k–1 comparisons to get the smallest one
  - repeat the procedure n times
  - time complexity: O(n*k)
- Any better method?
- **Selection tree**
  - two types: winner tree and loser tree

Winner Tree (1/2)

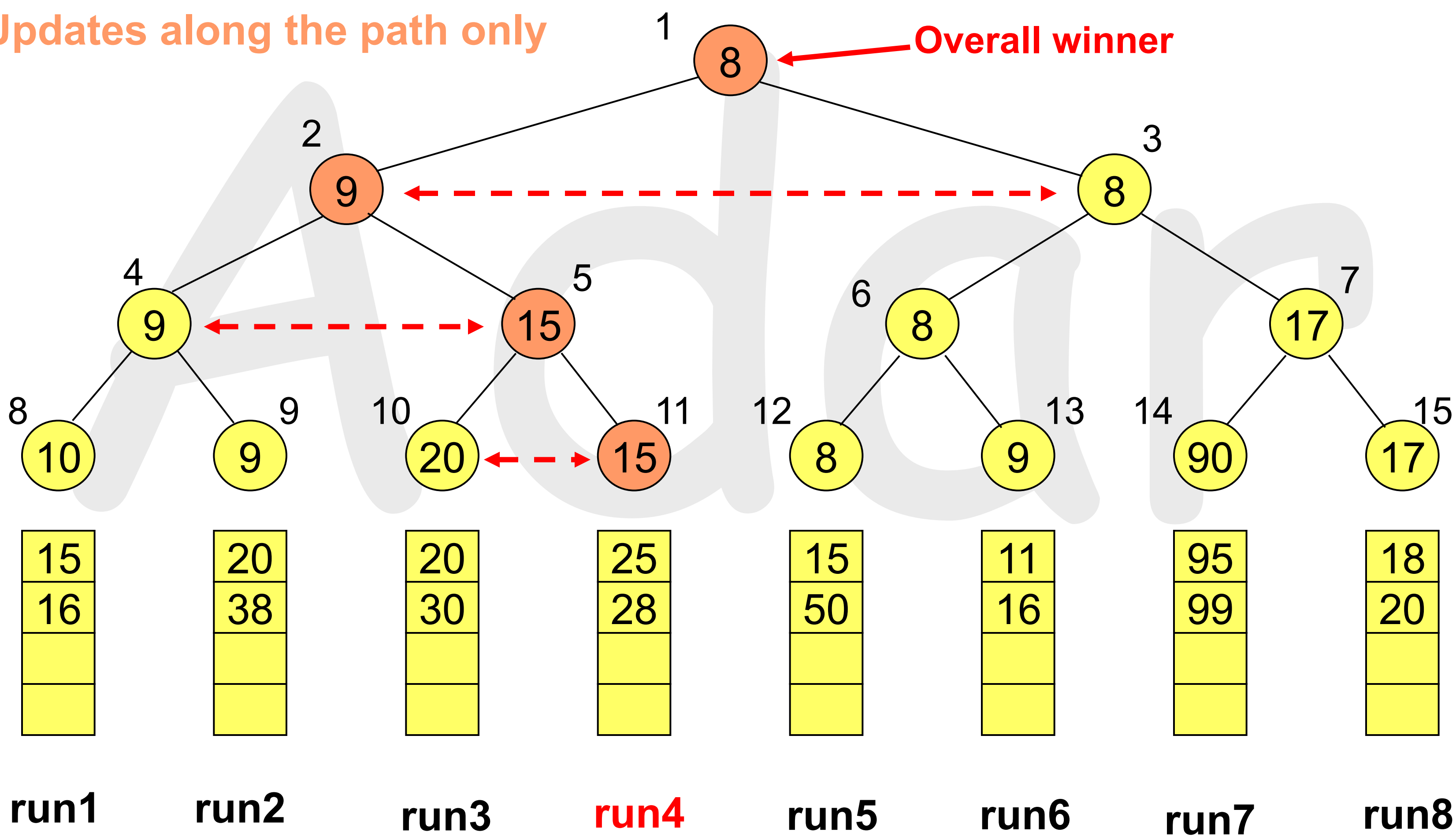
A winner tree is a complete binary tree in which each node represents the smaller of its 2 children



Winner Tree (2/2)

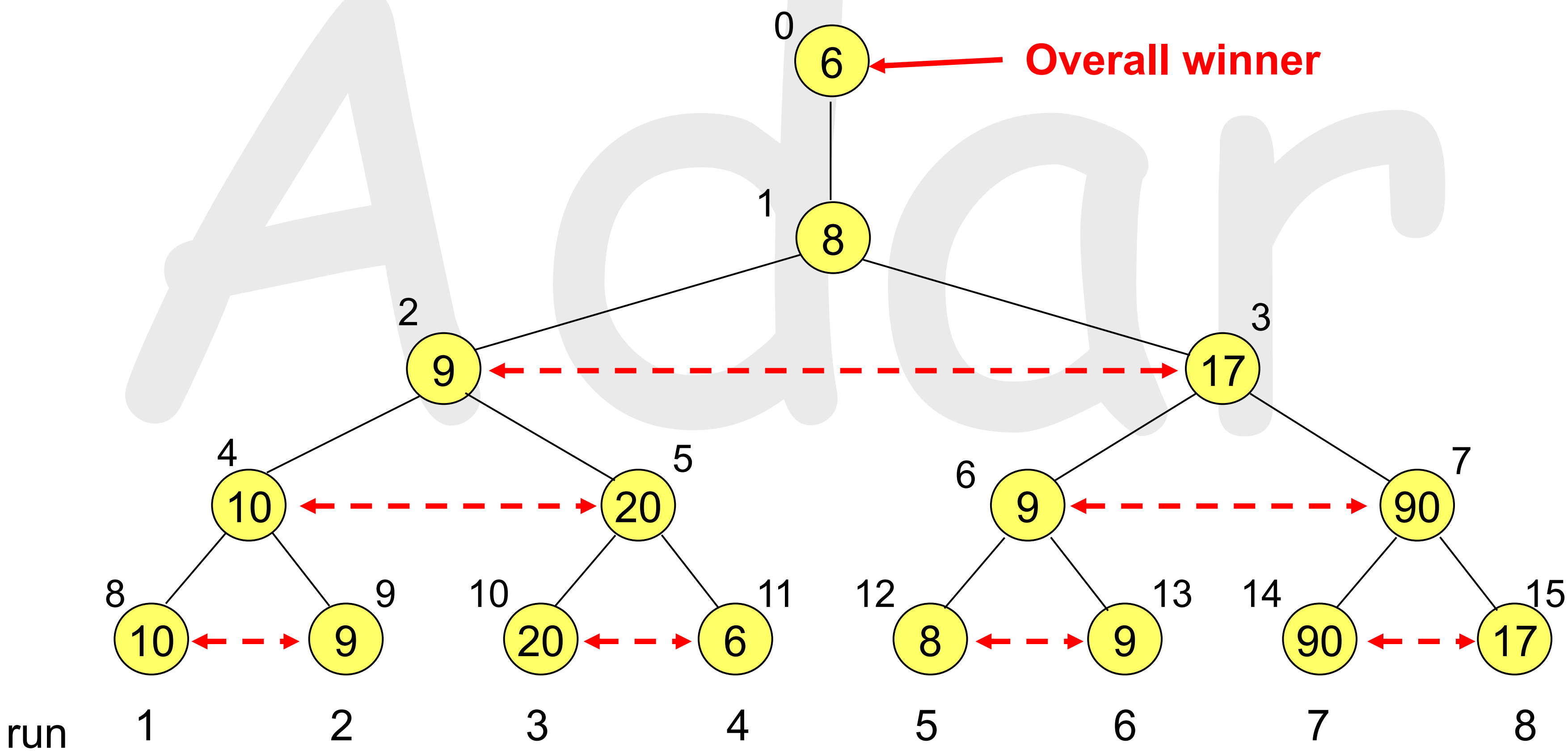
Time Complexity: O(n logk)

Updates along the path only



Loser Tree (1/2)

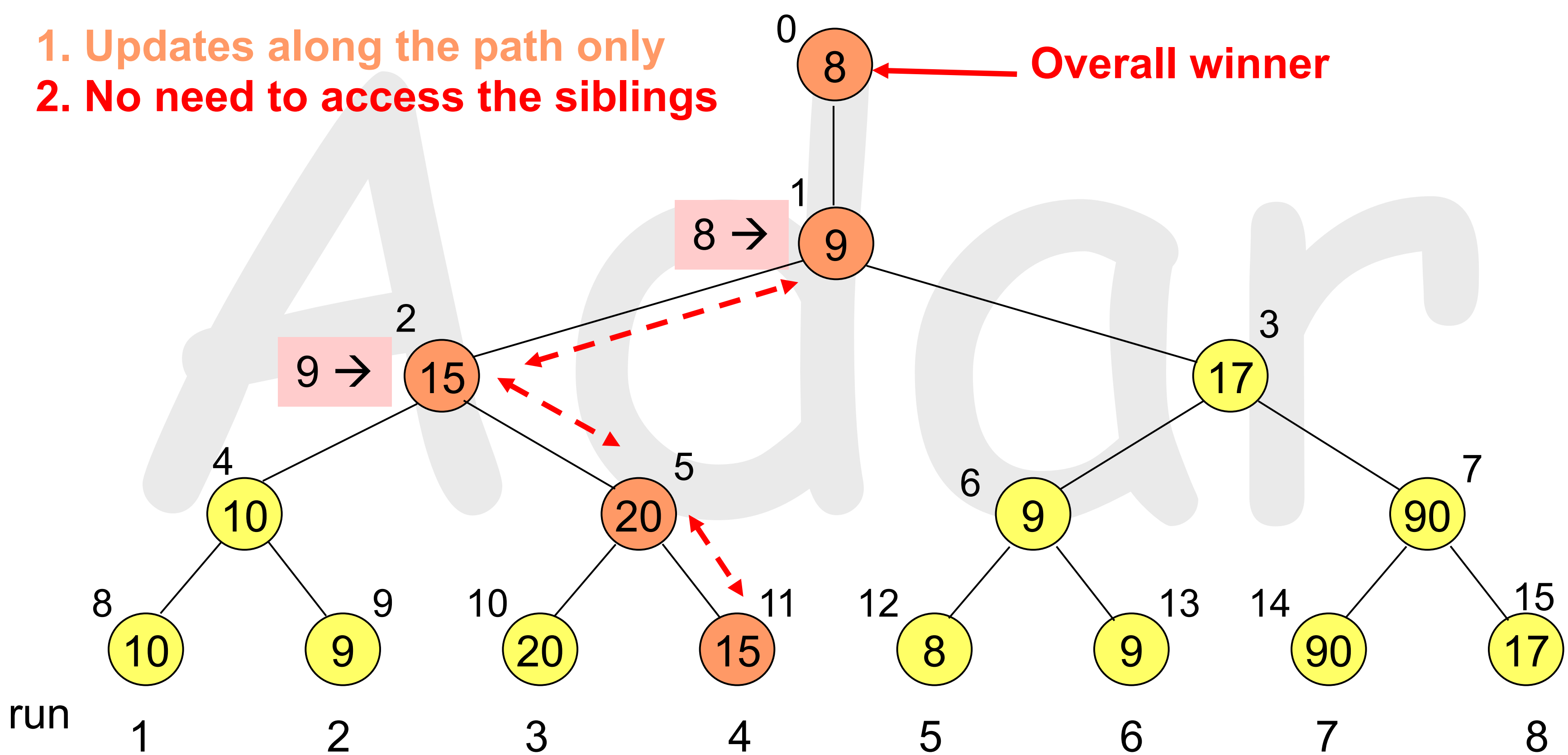
A loser tree is also a **complete** binary tree in which each node retains a pointer to the **loser** of the tournament



Loser Tree (2/2)

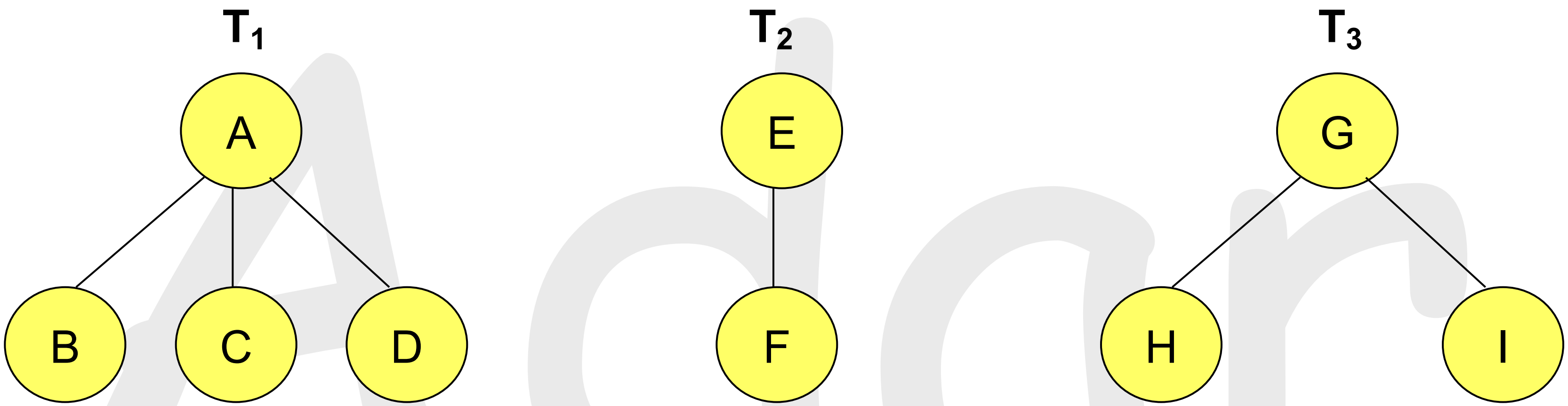
Time Complexity: O(n logk)

1. Updates along the path only
2. No need to access the siblings



Forest

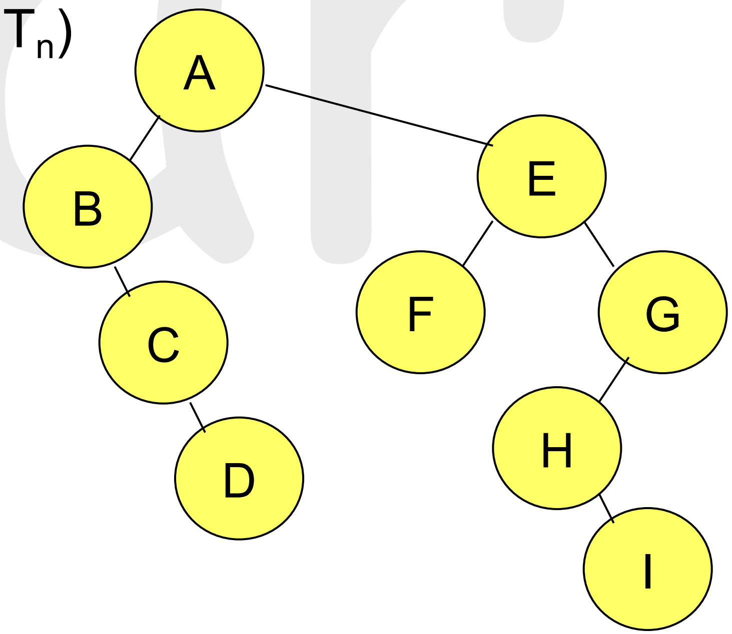
- A forest is a set of n ≥ 0 disjoint trees



3-tree forest

Convert a Forest into a Binary Tree

- If T_1, …, T_n is a forest of trees, the binary tree corresponding to the forest, denoted by B(T_1, …, T_n)
  - is empty if n = 0
  - has a root equal to root(T_1)
  - has a left subtree equal to B(T_{11}, T_{12}, …, T_{1m}), where T_{11}, …, T_{1m} are the subtrees of root(T_1)
  - has a right subtree equal to B(T_2, …, T_n)

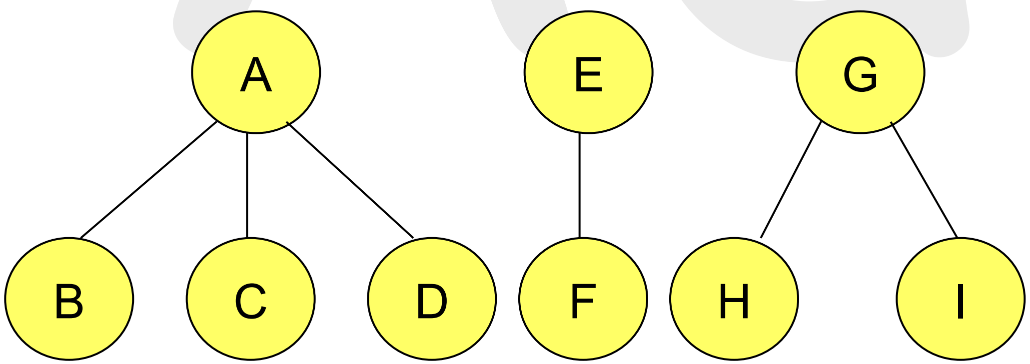


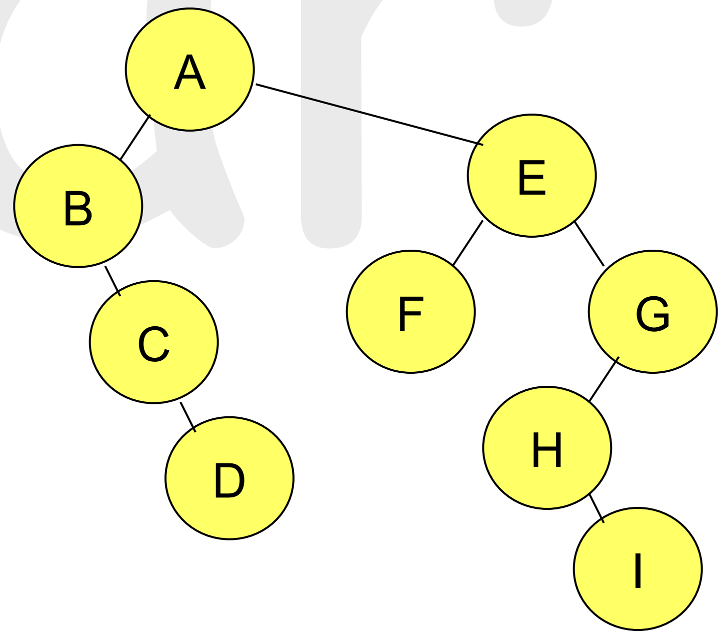
Compare with left child – right sibling representation of a tree

Forest Preorder

Assume a forest F and its corresponding binary tree T

- Preorder traversal of T is equivalent to visiting the nodes of 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 [recursive]
  - traverse the remaining trees of F in forest preorder [recursive]

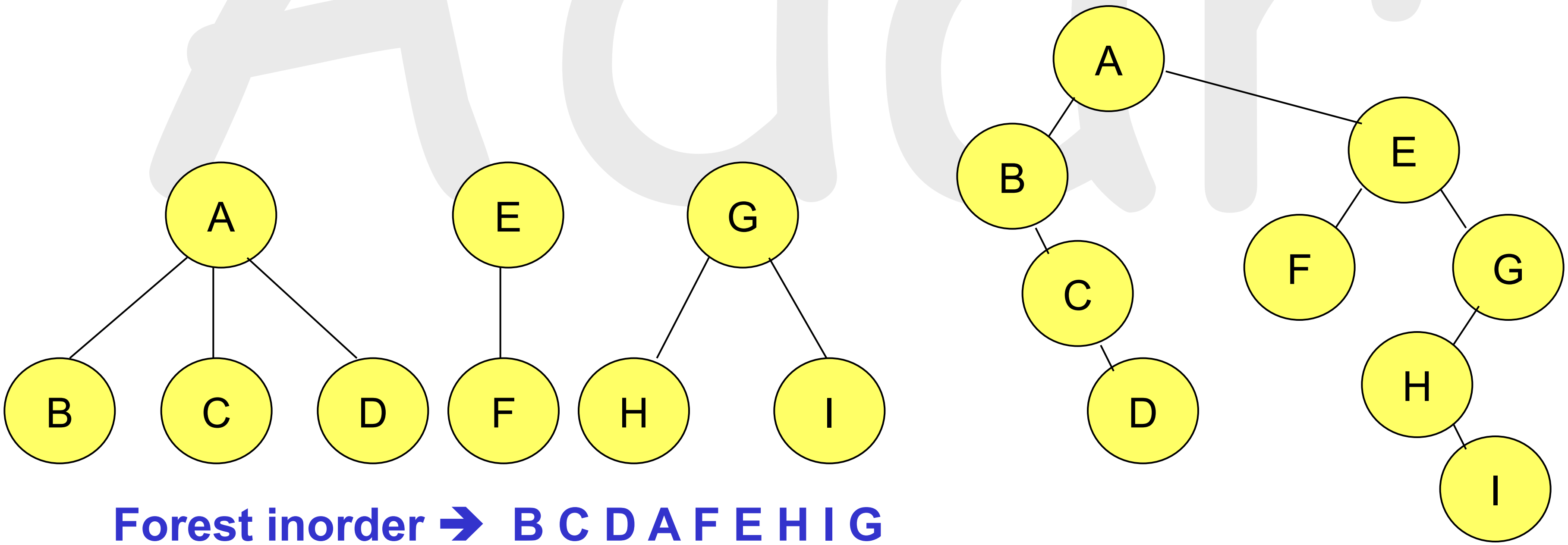




Forest preorder ➔ A B C D E F G H I

Forest Inorder

- Inorder traversal of T is equivalent to visiting the nodes of F in forest inorder
  - if F is empty then return
  - traverse the subtrees of the first tree in forest inorder [recursive]
  - visit the root of the first tree of F
  - traverse the remaining trees of F in forest inorder [recursive]



Forest inorder → B C D A F E H I G

Final Review

- Trees
  - degree, leaf, parent, child, sibling, ancestor, level, height, depth
  - degree-k, left child-right sibling, degree-2 representations
- Binary trees
  - skewed, complete, full binary trees
  - array, linked list representations
  - traversals: preorder(VLR), inorder(LVR), postorder(LRV), level-order
    - recursive or non-recursive; stack or queue
  - operations: tree copy, equality test, SAT
- Threaded binary trees
- Min/max heaps as min/max priority queues: insertion/deletion
- Binary search tree (BST): search/insertion/deletion
- Selection trees: winner tree and loser tree
- Forest

C++ Reference

Containers in C++ STL

- priority_queue
  - is actually a container adaptor
  - default container: vector
- map/multimap : key ➔ data
  - is typically a balanced BST
  - typical implementation: red-black tree
- set/multiset : key only
  - is typically a balanced BST
  - typical implementation: red-black tree