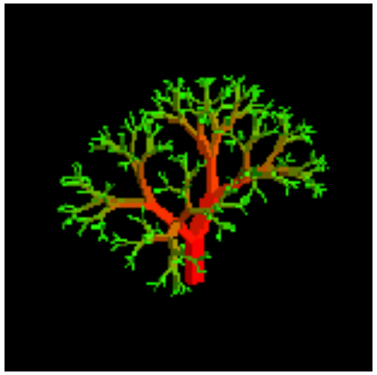
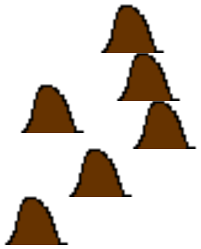
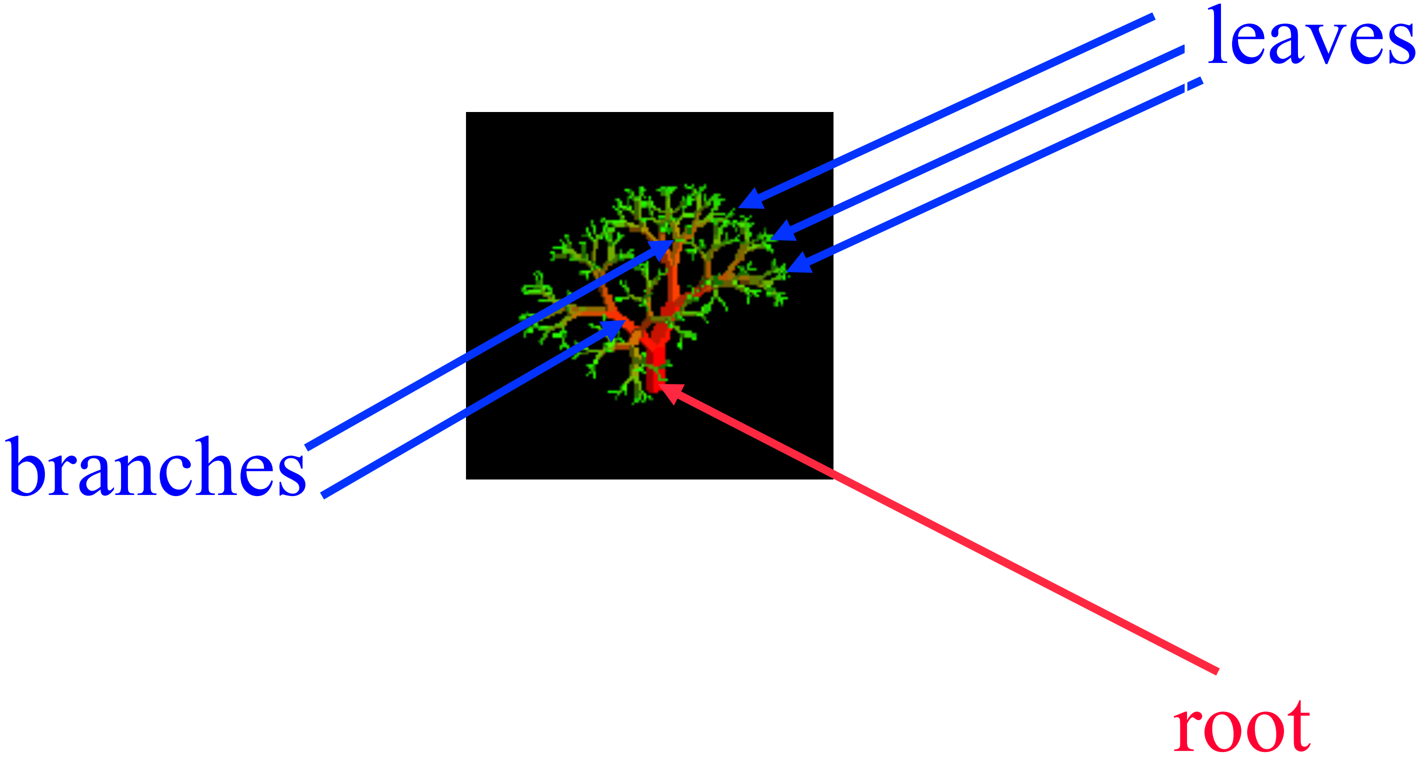
Trees

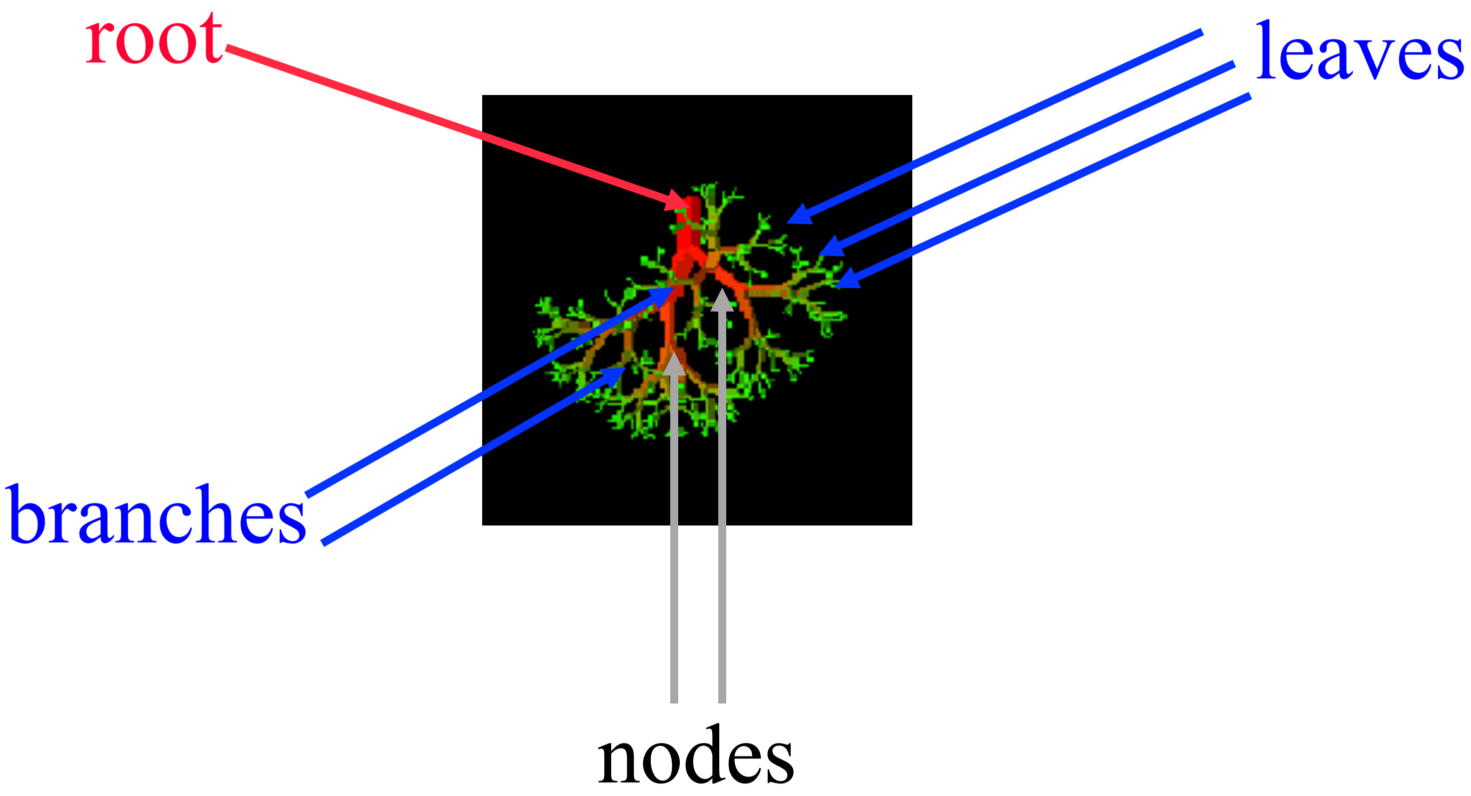




Nature Lover's View Of A Tree



Computer Scientist's View



Chapter 5
Trees

5.1 Introduction

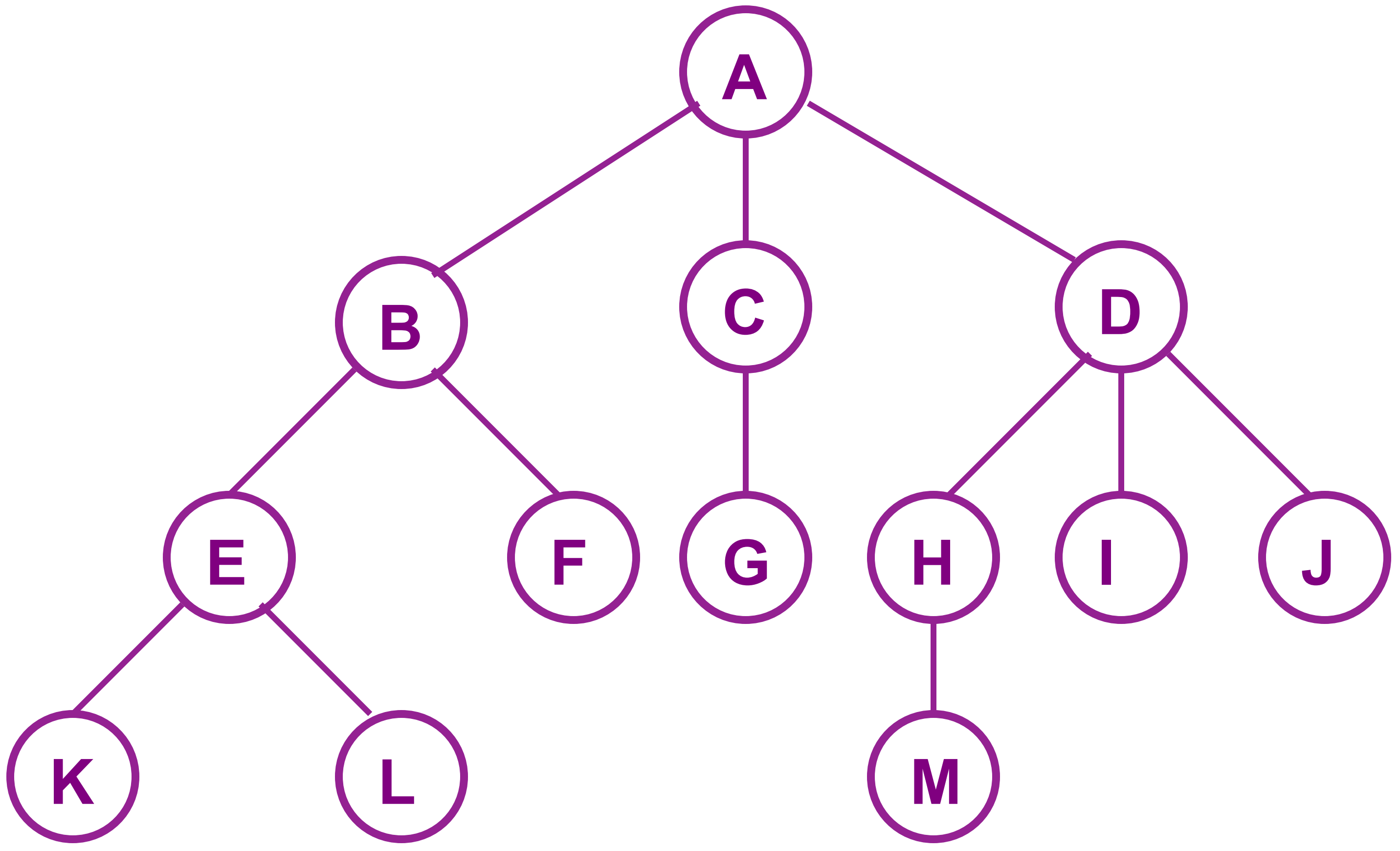
5.1.1 Terminology

Definition: A tree is a finite set of one or more nodes such that

(1) There is a specially designated node called root.

(2) The remaining nodes are partitioned into $n \geq 0$ disjoint sets $T_1, \ldots, T_n$, where each of these sets is a tree.

$T_1, \ldots, T_n$ are called subtrees of the root.


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

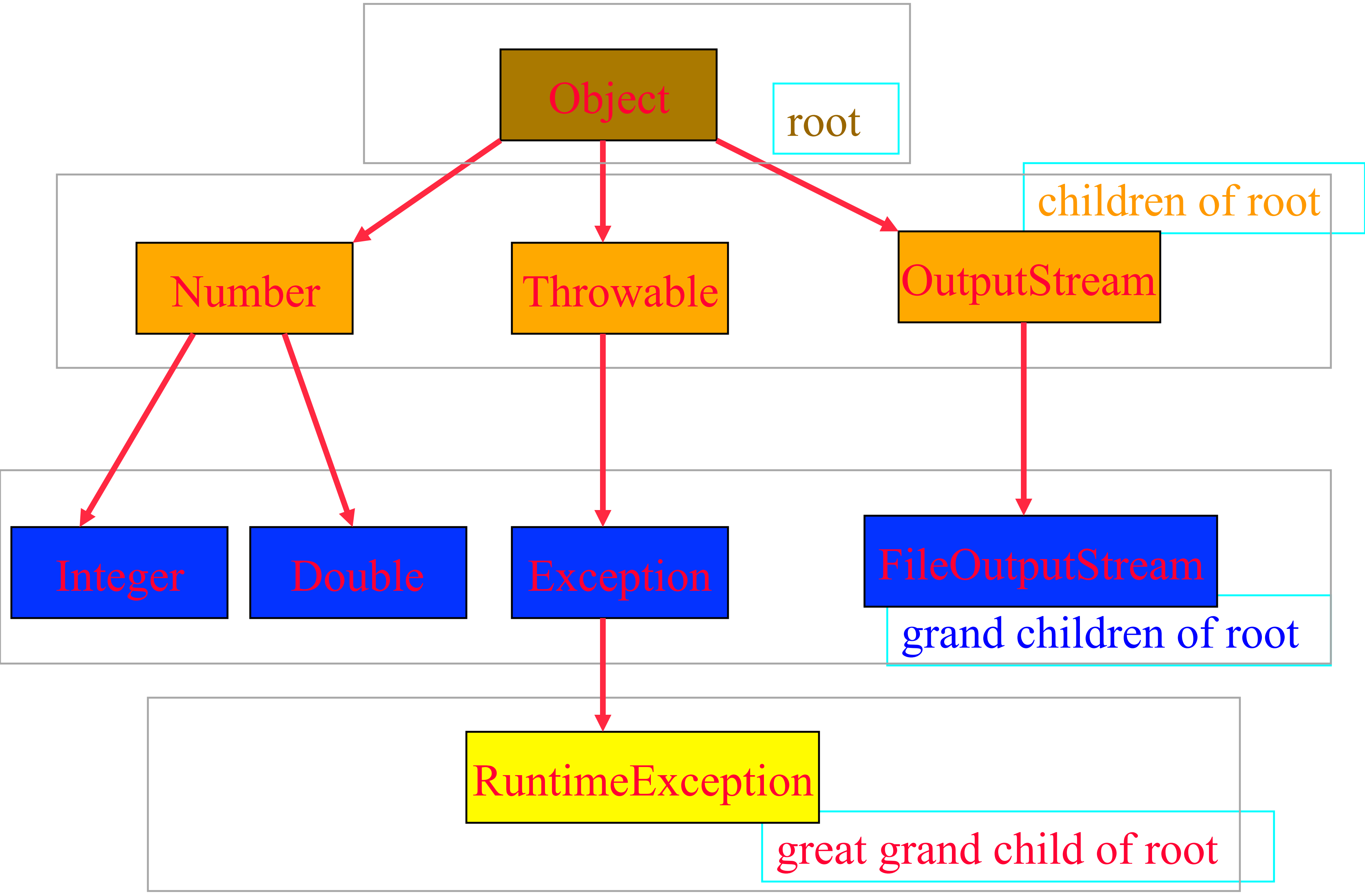
Linear Lists And Trees

- Linear lists are useful for serially ordered data.
  - $(e_0, e_1, e_2, \ldots, e_{n-1})$
  - Days of week.
  - Months in a year.
  - Students in this class.
- Trees are useful for hierarchically ordered data.
  - Employees of a corporation.
    - President, vice presidents, managers, and so on.
  - classes.
    - Object is at the top of the hierarchy.
    - Subclasses of Object are next, and so on.

Hierarchical Data And Trees

- The element at the top of the hierarchy is the root.
- Elements next in the hierarchy are the children of the root.
- Elements next in the hierarchy are the grandchildren of the root, and so on.
- Elements that have no children are leaves.

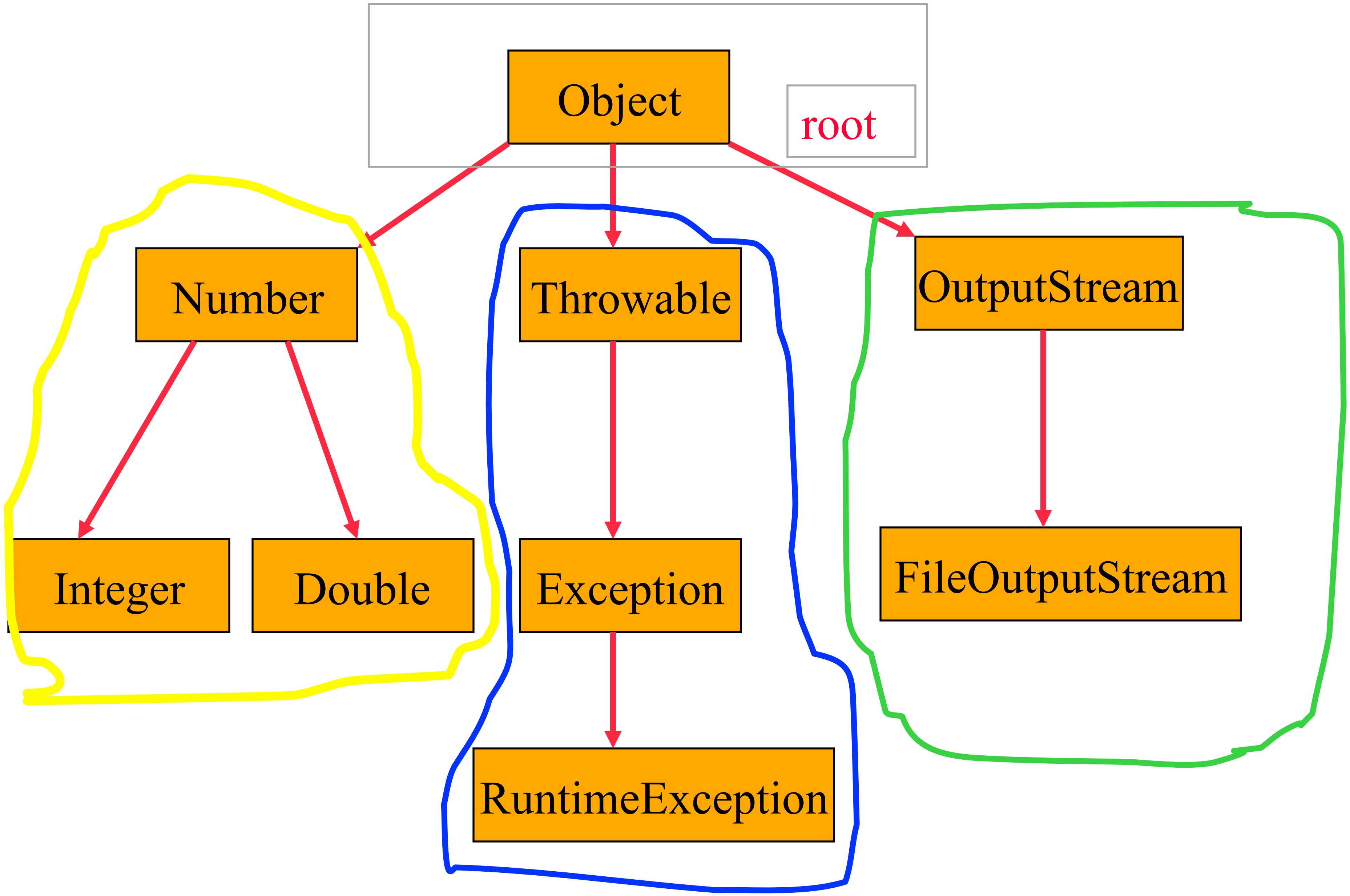
Java's Classes (Part Of Figure 1.1)



Definition

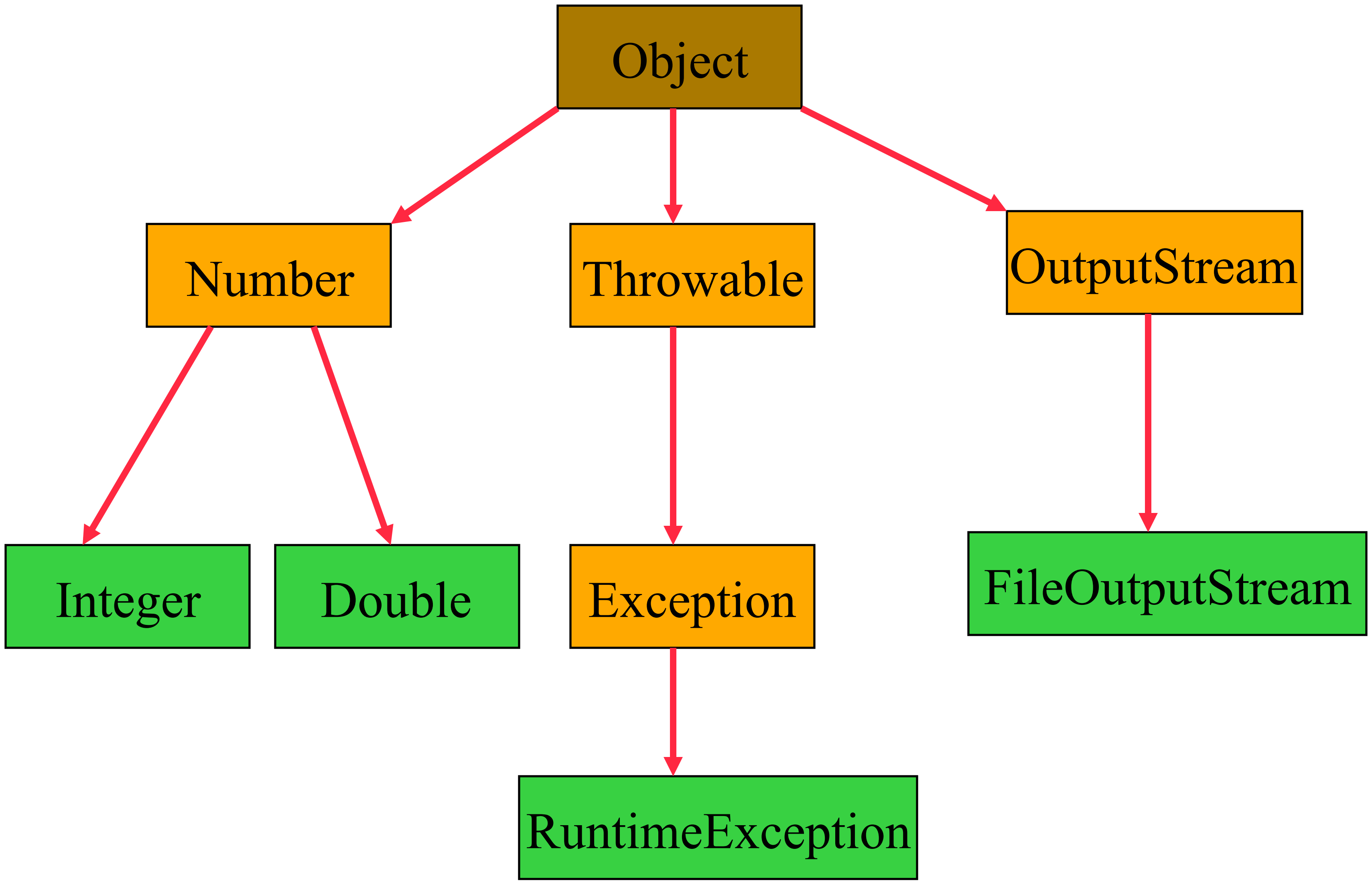
- A tree t is a finite nonempty set of elements.
- One of these elements is called the root.
- The remaining elements, if any, are partitioned into trees, which are called the subtrees of t.

Subtrees

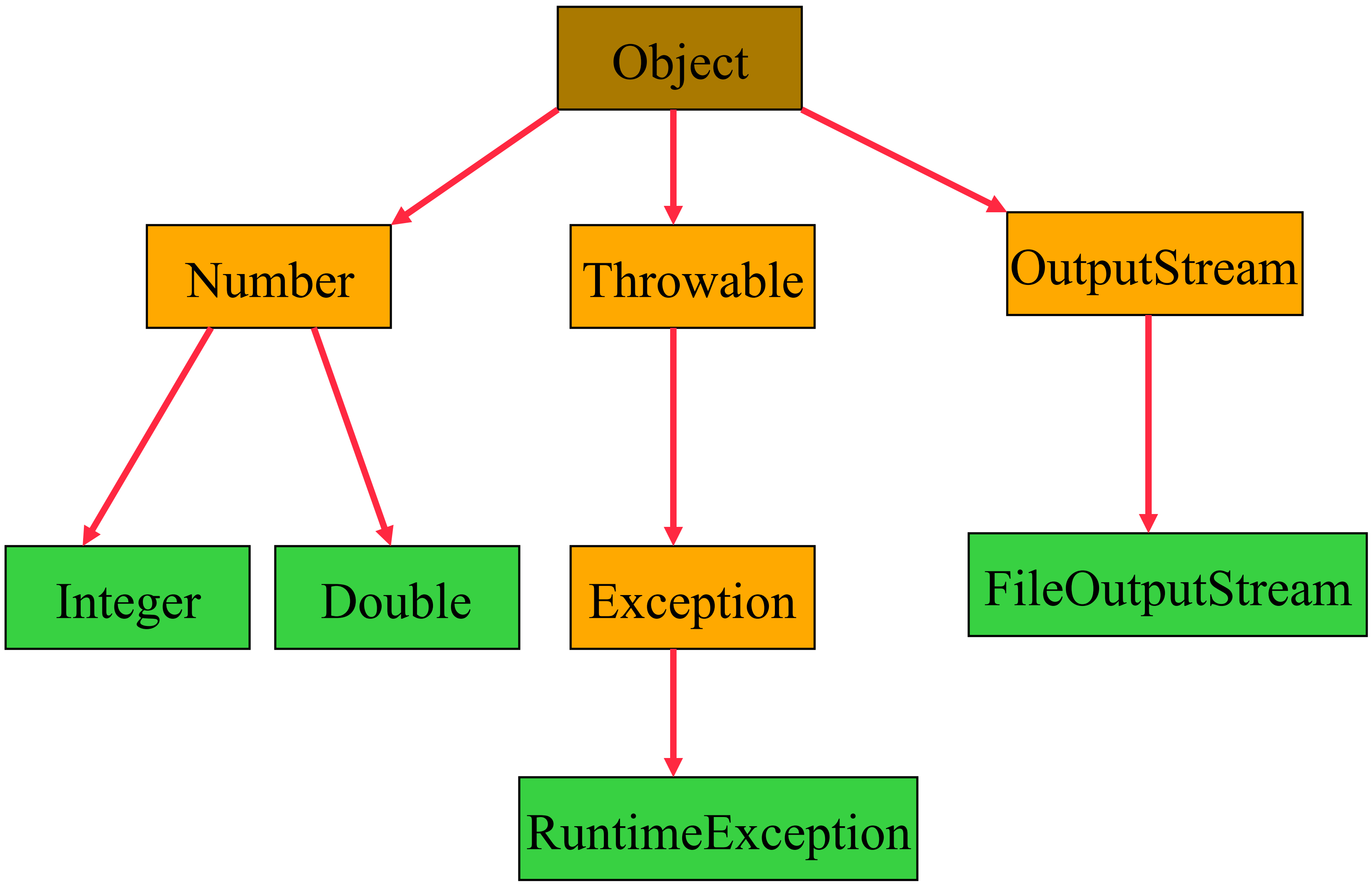


Object
root
Number
Throwable
OutputStream
Integer
Double
Exception
FileOutputStream
RuntimeException

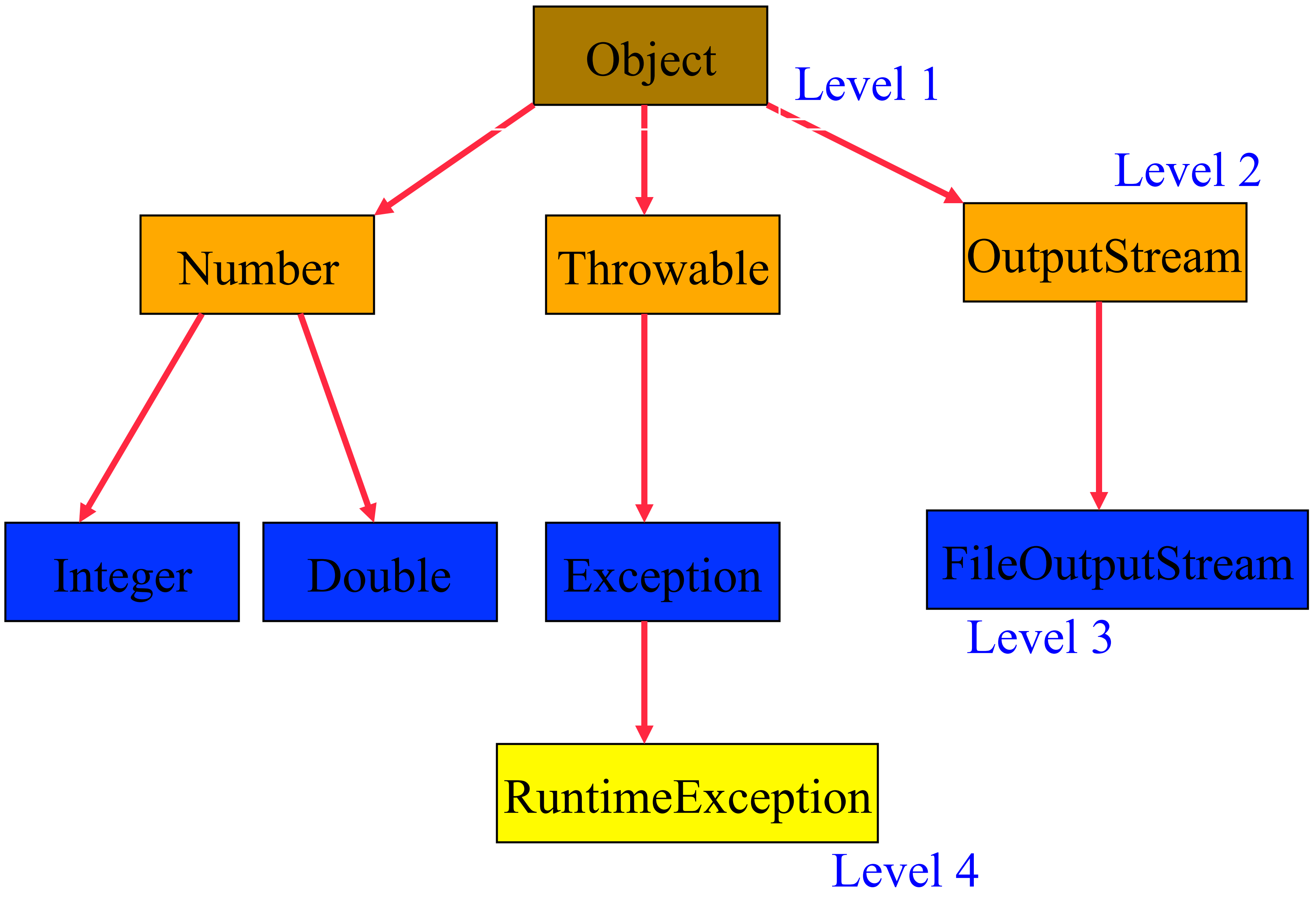
Leaves


Object
Number
Throwable
OutputStream
Integer
Double
Exception
FileOutputStream
RuntimeException

Parent, Grandparent, Siblings, Ancestors, Descendants



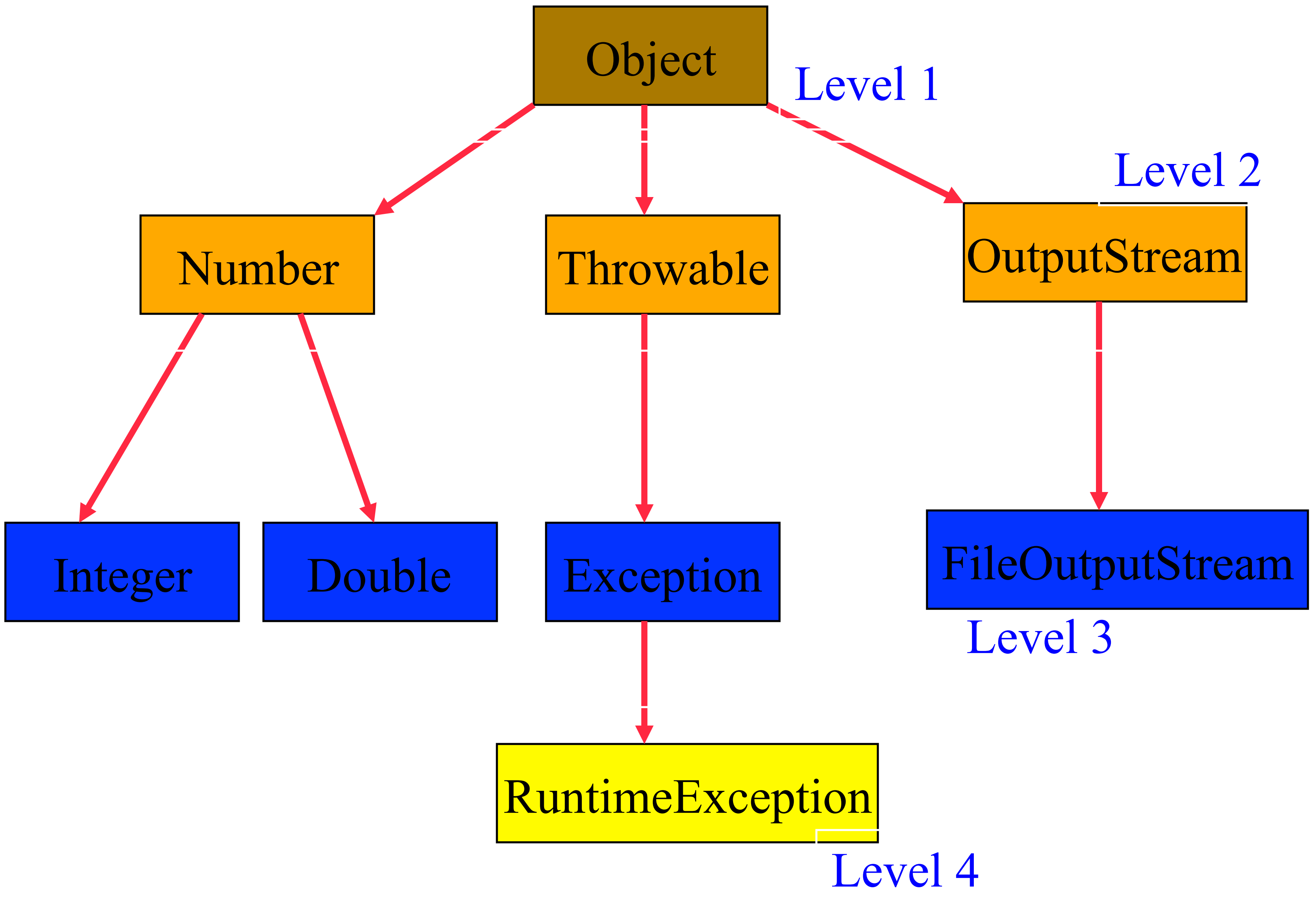
Levels


Object
Level 1
Number
Throwable
OutputStream
Level 2
Integer
Double
Exception
FileOutputStream
Level 3
RuntimeException
Level 4

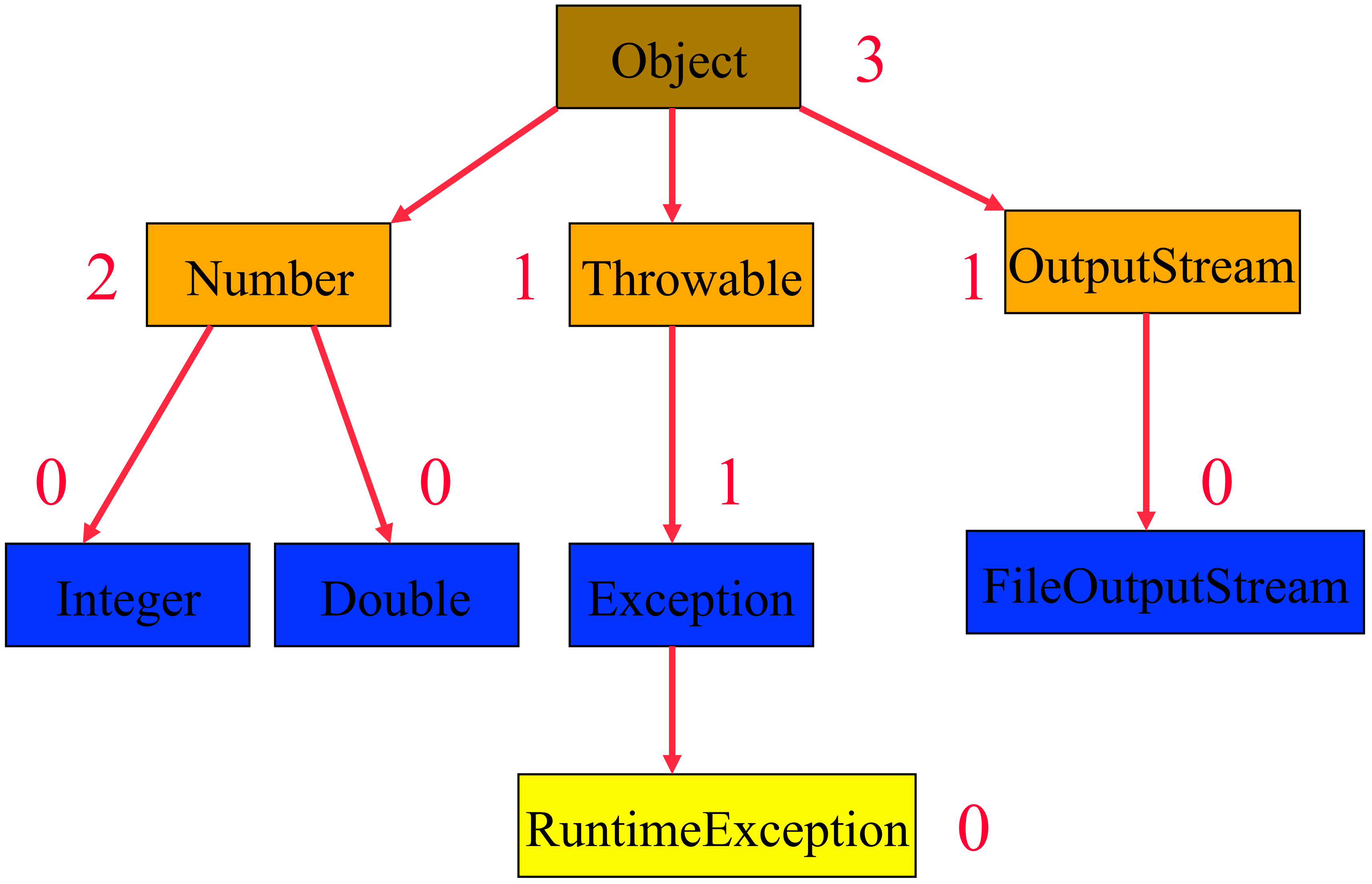
Caution

- Some texts start level numbers at 0 rather than at 1.
- Root is at level 0.
- Its children are at level 1.
- The grand children of the root are at level 2.
- And so on.
- We shall number levels with the root at level 1.

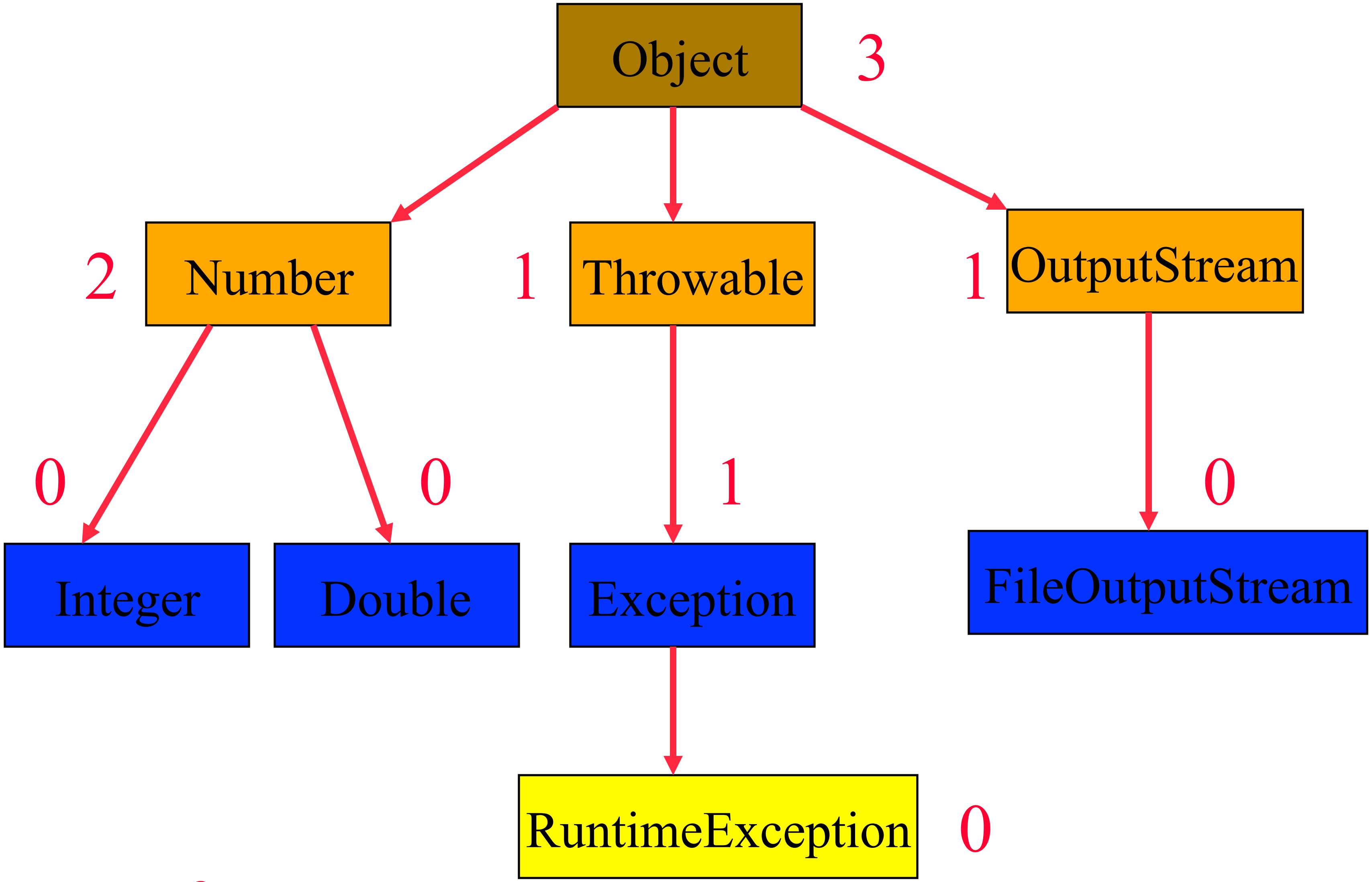
height = depth = number of levels



Node Degree = Number Of Children



Tree Degree = Max Node Degree

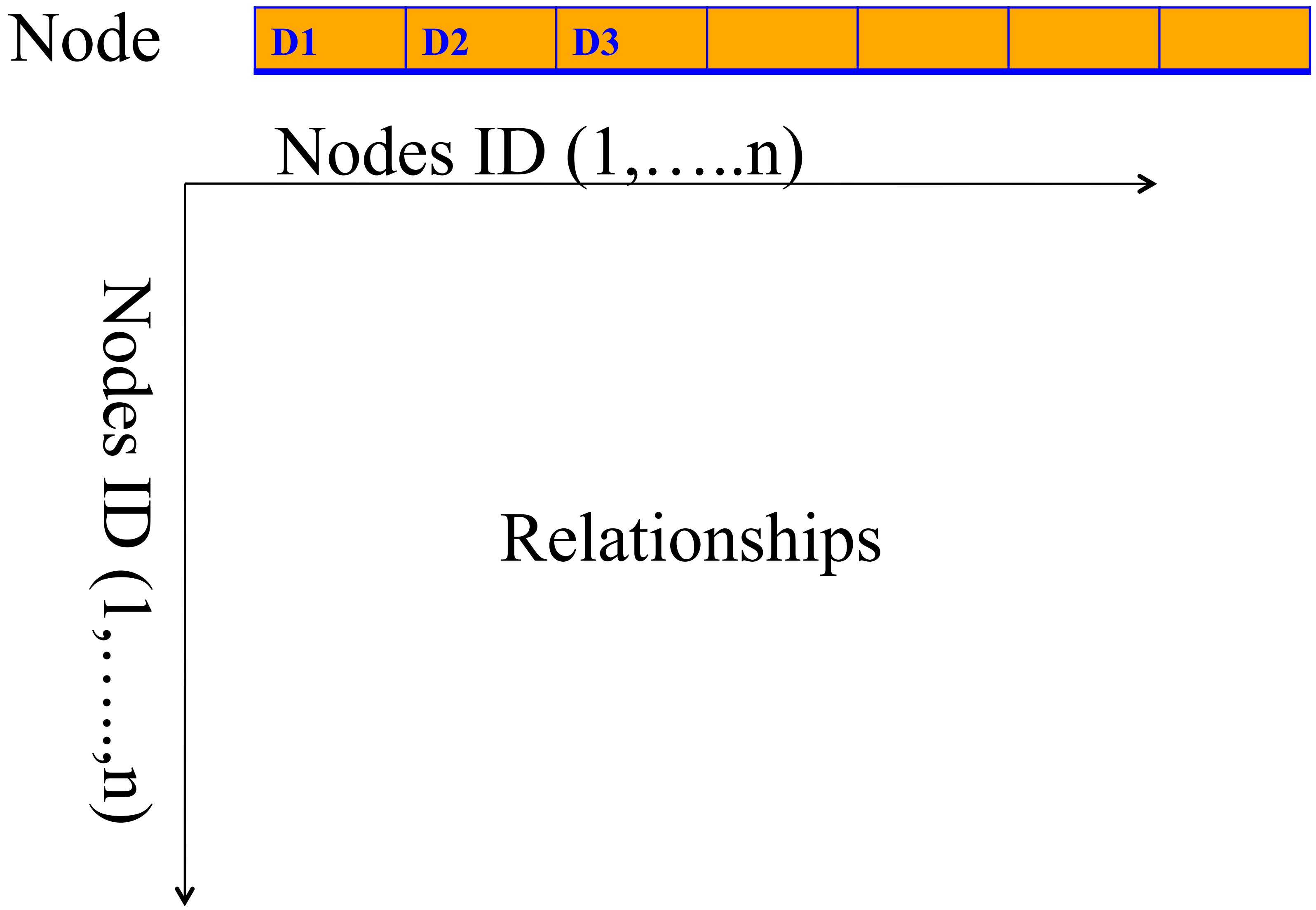


Degree of tree = 3.

Representation

- What to be recorded?
  - Nodes
  - Relationships (Edges)
- Array
- Linked Lists

5.1.2 Representation of Trees



5.1.2 Representation of Trees

For a tree of degree k, we could use a tree node that has fields for data and k pointers to the children

Data	**Child1**	**Child2**	**…**	**Child k**

Possible node structure for a tree c

Waste of space!

Lemma5.1: If T is a k-ary tree with n nodes, each having a fixed size as in Fig.5.4, then n(k-1)+1 of the n*k child fields are 0, n ≥ 1.

Proof:

- **each non-zero child field points to a node**
- **there is exactly one pointer to each node other than the root**
- **the number of non-zero child fields in an n node tree is:**
  - **n-1**
- **the number of zero fields is**
  - **nk-(n-1)=n(k-1)+1.**

Binary Tree

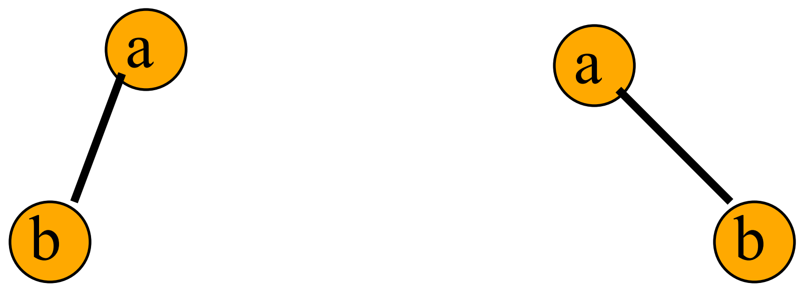
- Finite (possibly empty) collection of elements.
- A nonempty binary tree has a root element.
- The remaining elements (if any) are partitioned into two binary trees.
- These are called the left and right subtrees of the binary tree.

Differences Between A Tree & A Binary Tree

- No node in a binary tree may have a degree more than 2, whereas there is no limit on the degree of a node in a tree.
- A binary tree may be empty; a tree cannot be empty.

Differences Between A Tree & A Binary Tree

- The subtrees of a binary tree are ordered; those of a tree are not ordered.



- Are different when viewed as binary trees.
- Are the same when viewed as trees.

ADT 5.1

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

    bool IsEmpty ();
    // return true iff the binary tree is empty

    BinaryTree(BinaryTree<T>& bt1, T& item,


BinaryTree<T>& bt2);
    // creates a binary tree whose left subtree is bt1,
    // right subtree is bt2, and root node contain item.
    BinaryTree  LeftSubtree();
   // return the left subtree of *this

   T RootData();
   // return the data in the root of *this

   BinaryTree  RightSutree();
   // return the right subtree of *this
};
```

Arithmetic Expressions

- (a + b) * (c + d) + e – f/g*h + 3.25
- Expressions comprise three kinds of entities.
  - Operators (+, -, /, *).
  - Operands (a, b, c, d, e, f, g, h, 3.25, (a + b), (c + d), etc.).
  - Delimiters ((,)).

Operator Degree

- Number of operands that the operator requires.
- Binary operator requires two operands.
  - a + b
  - c / d
  - e - f
- Unary operator requires one operand.
  - + g
  - - h

Infix Form

- Normal way to write an expression.
- Binary operators come in between their left and right operands.
  - a * b
  - a + b * c
  - a * b / c
  - (a + b) * (c + d) + e – f/g*h + 3.25

Operator Priorities

- How do you figure out the operands of an operator?
  - a + b * c
  - a * b + c / d
- This is done by assigning operator priorities.
  - priority(*) = priority(/) > priority(+) = priority(-)
- When an operand lies between two operators, the operand associates with the operator that has higher priority.

Tie Breaker

- When an operand lies between two operators that have the same priority, the operand associates with the operator on the left.
  - a + b - c
  - a * b / c / d

Delimiters

- Subexpression within delimiters is treated as a single operand, independent from the remainder of the expression.
  - $(a + b) * (c - d) / (e - f)$

Infix Expression Is Hard To Parse

- Need operator priorities, tie breaker, and delimiters.
- This makes computer evaluation more difficult than is necessary.
- Postfix and prefix expression forms do not rely on operator priorities, a tie breaker, or delimiters.
- So it is easier for a computer to evaluate expressions that are in these forms.

Postfix Form

- The postfix form of a variable or constant is the same as its infix form.
  - a, b, 3.25
- The relative order of operands is the same in infix and postfix forms.
- Operators come immediately after the postfix form of their operands.
  - Infix = a + b
  - Postfix = ab+

Postfix Examples

- Infix = a + b * c
  - Postfix = a b c * +
- Infix = a * b + c
  - Postfix = a b * c +
- Infix = (a + b) * (c – d) / (e + f)
  - Postfix = a b + c d - * e f + /

Unary Operators

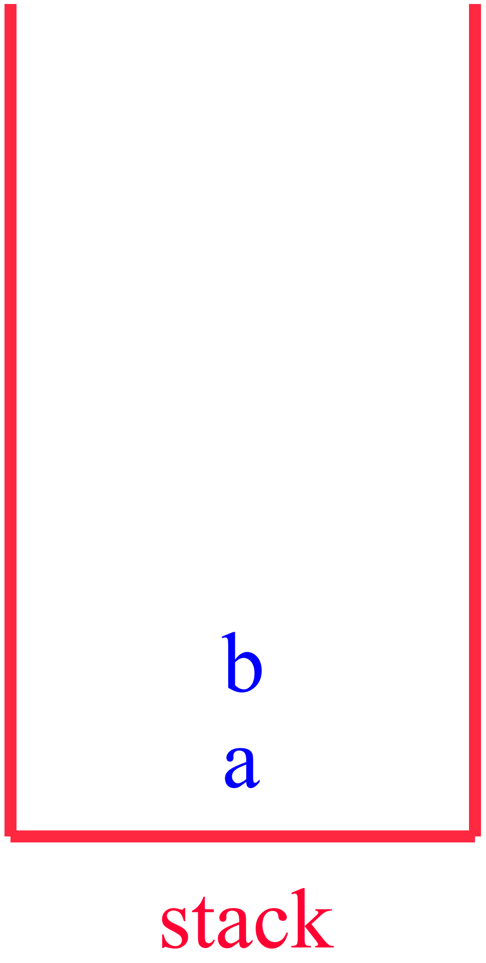
- Replace with new symbols.
  - + a => a @
  - + a + b => a @ b +
  - - a => a ?
  - - a-b => a ? b -

Postfix Evaluation

- Scan postfix expression from left to right pushing operands on to a stack.
- When an operator is encountered, pop as many operands as this operator needs; evaluate the operator; push the result on to the stack.
- This works because, in postfix, operators come immediately after their operands.

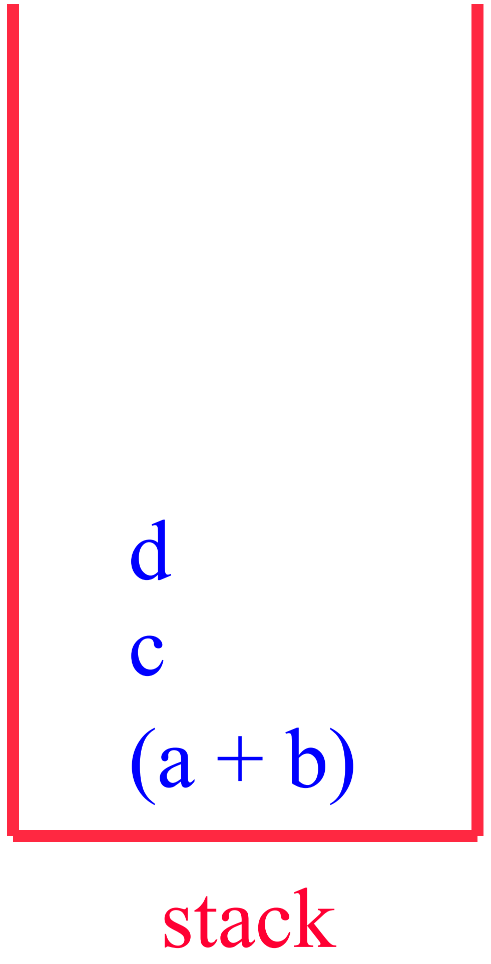
Postfix Evaluation

- (a + b) * (c – d) / (e + f)
- a b + c d - * e f + /
- a b + c d - * e f + /
- a b + c d - * e f + /
- a b + c d - * e f + /



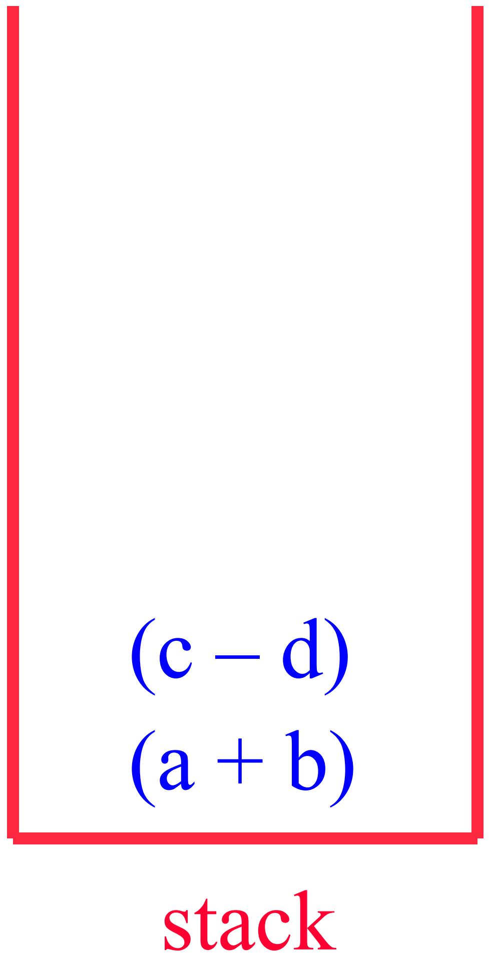
Postfix Evaluation

- (a + b) * (c – d) / (e + f)
- a b + c d - * e f + /
- a b + c d - * e f + /
- a b + c d - * e f + /
- a b + c d - * e f + /
- a b + c d - * e f + /
- a b + c d - * e f + /
- a b + c d - * e f + /



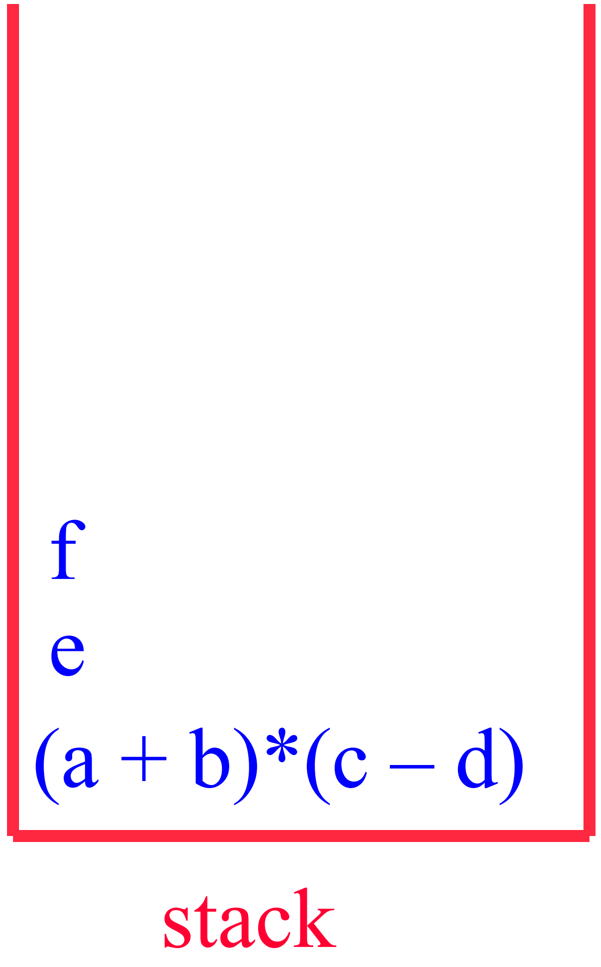
Postfix Evaluation

- (a + b) * (c – d) / (e + f)
- a b + c d - * e f + /
- a b + c d - * e f + /



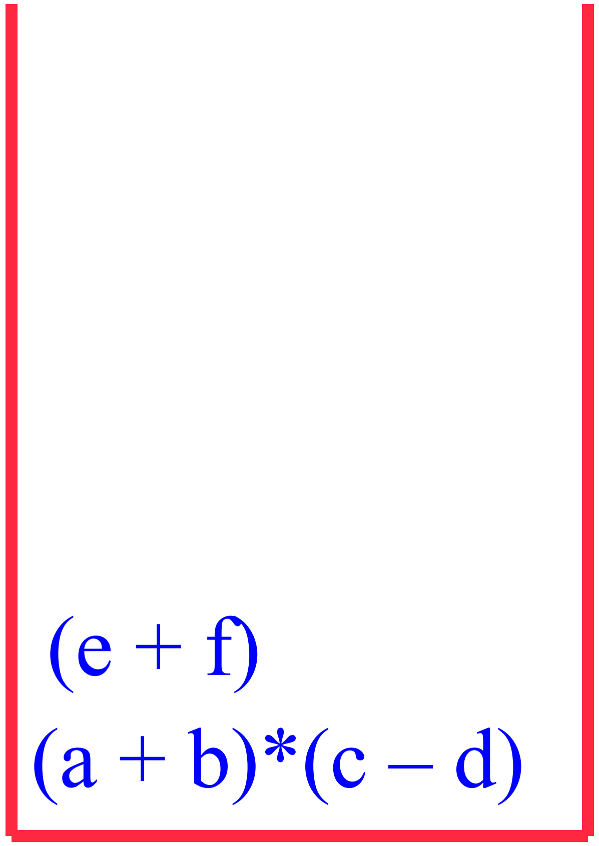
Postfix Evaluation

- (a + b) * (c – d) / (e + f)
- a b + c d - * e f + /
- a b + c d - * e f + /
- a b + c d - * e f + /
- a b + c d - * e f + /
- a b + c d - * e f + /



Postfix Evaluation

- (a + b) * (c – d) / (e + f)
- a b + c d - * e f + /
- a b + c d - * e f + /
- a b + c d - * e f + /
- a b + c d - * e f + /
- a b + c d - * e f + /
- a b + c d - * e f + /



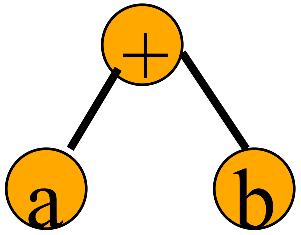
stack

Prefix Form

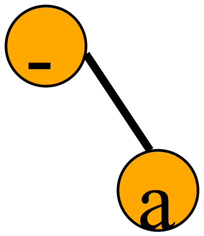
- The prefix form of a variable or constant is the same as its infix form.
  - a, b, 3.25
- The relative order of operands is the same in infix and prefix forms.
- Operators come immediately before the prefix form of their operands.
  - Infix = a + b
  - Postfix = ab+
  - Prefix = +ab

Binary Tree Form

- a + b

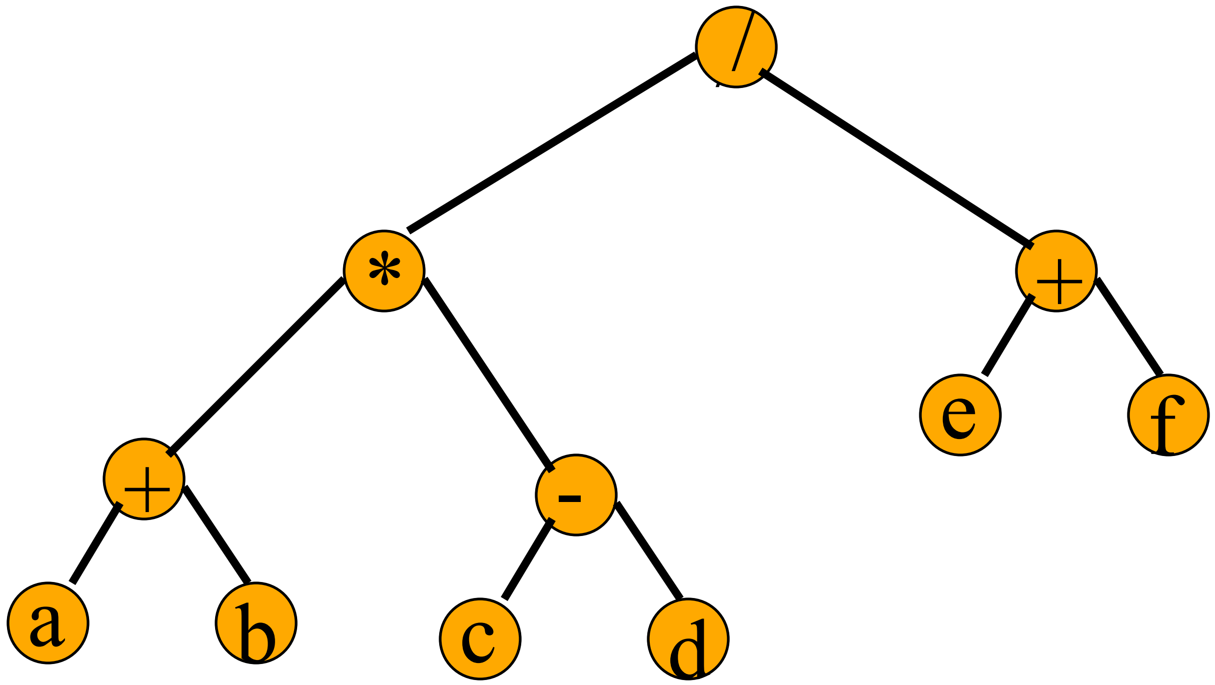


- - a



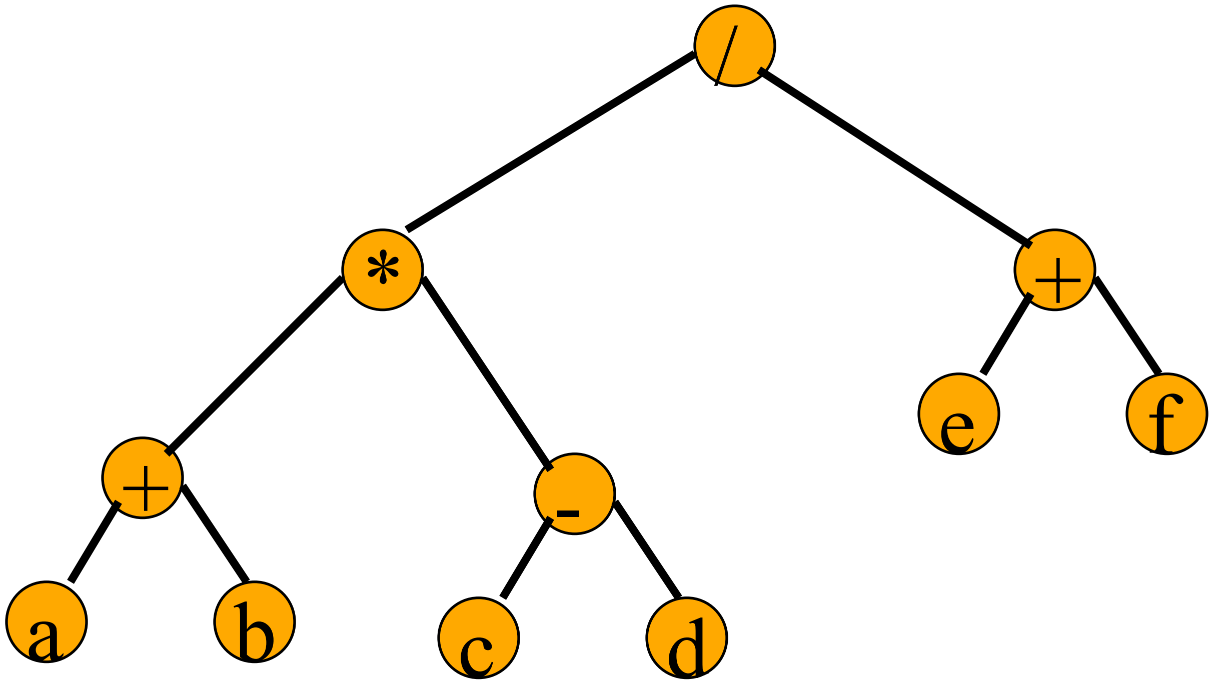
Binary Tree Form

- (a + b) * (c – d) / (e + f)

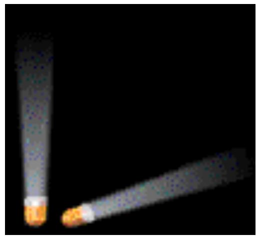
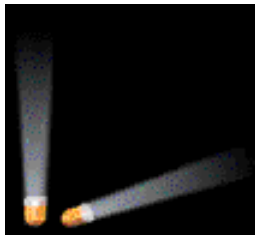
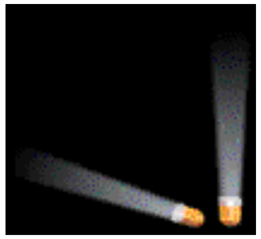
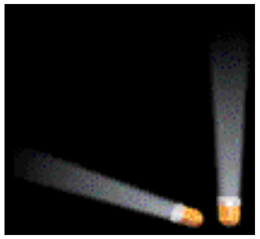
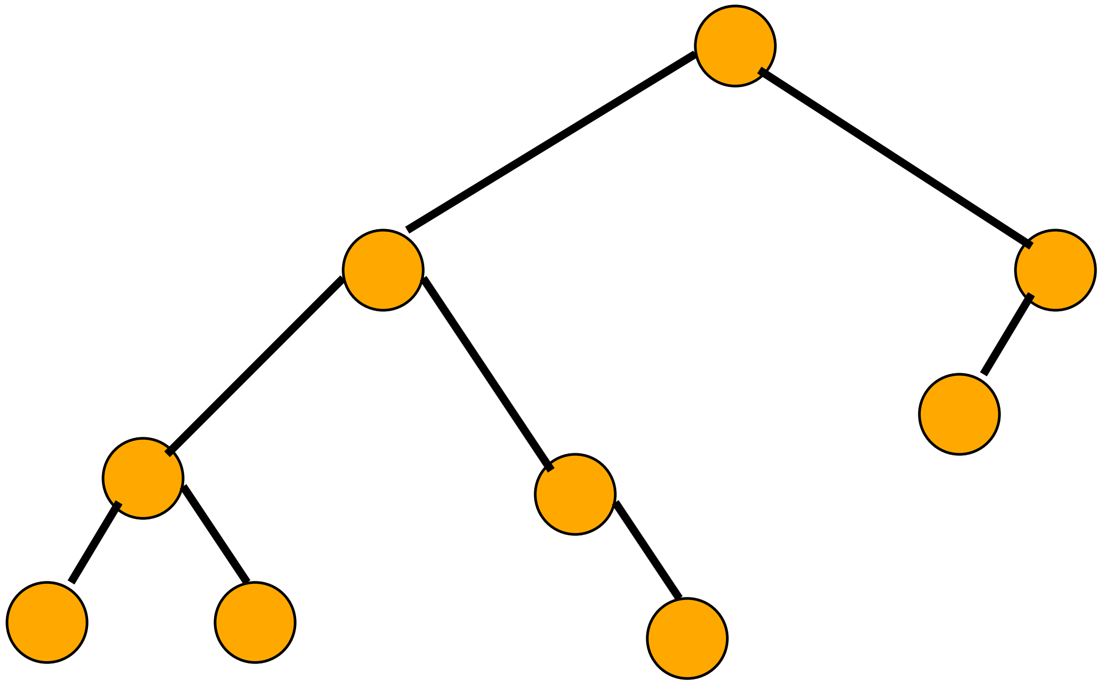


Evaluation of Binary Tree Form

- HOW?
  - Simple recursive evaluation
- Exercise: write an algorithm

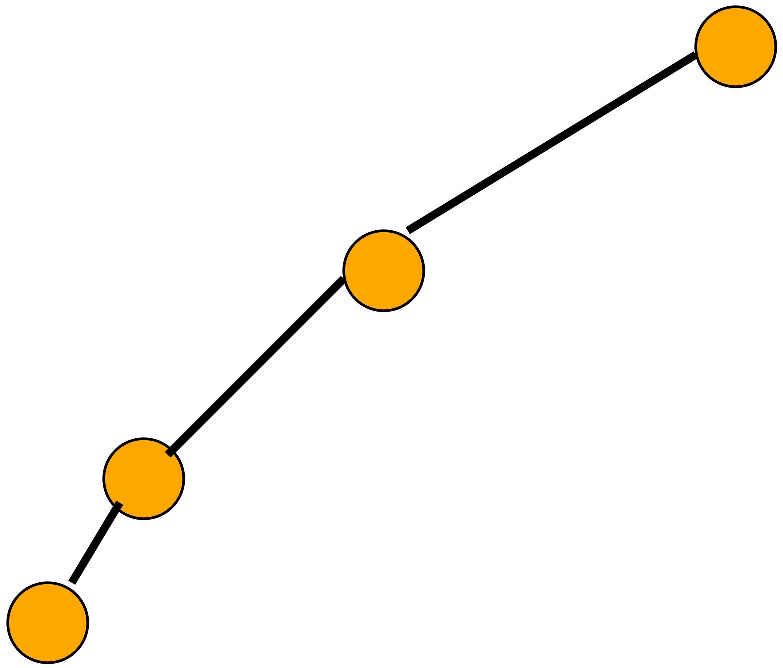


Binary Tree Properties & Representation

Minimum Number Of Nodes

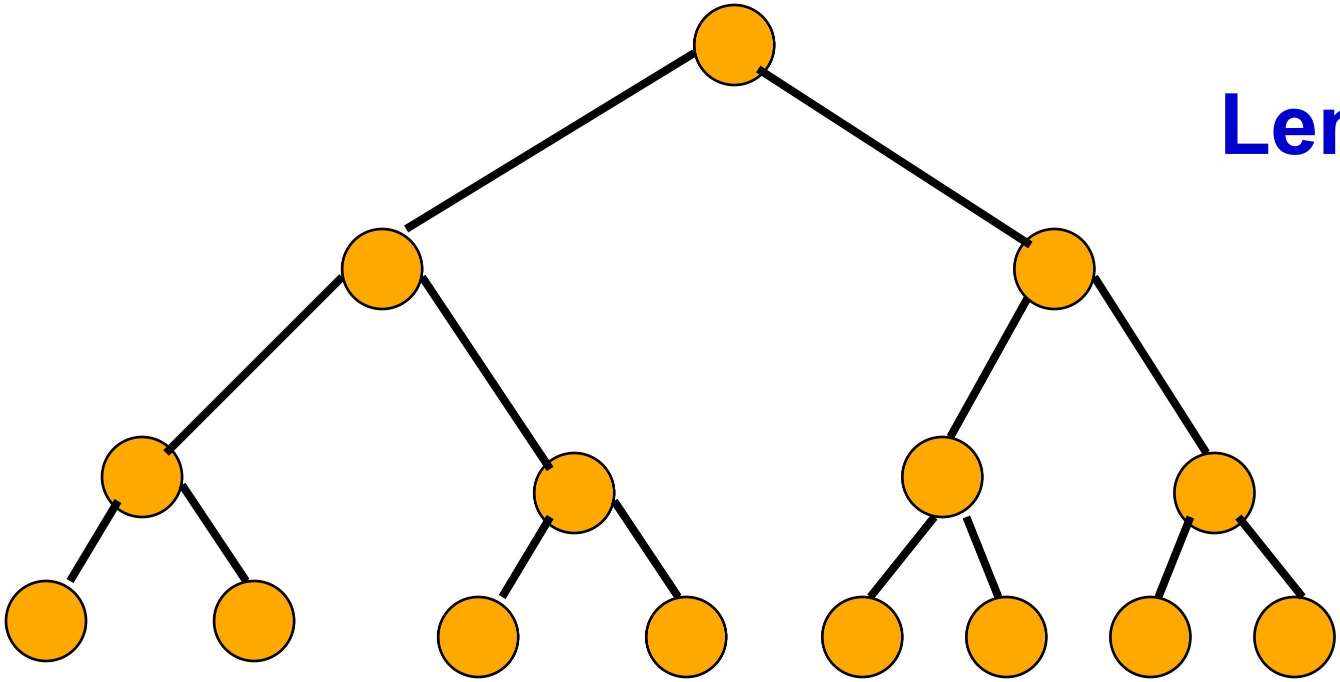
- Minimum number of nodes in a binary tree whose height is h.
- At least one node at each of first h levels.



minimum number of nodes is h

Maximum Number Of Nodes

- All possible nodes at first h levels are present.



Lemma 5.2

Maximum number of nodes

$= 1 + 2 + 4 + 8 + \ldots + 2^{h-1}$

$= 2^h - 1$

Number Of Nodes & Height

- Let n be the number of nodes in a binary tree whose height is h.
- $h <= n <= 2^h - 1$
- $\log_2(n+1) <= h <= n$

Lemma 5.3 [Relation between number of leaf nodes and degree-2 nodes]:

For any nonempty binary tree T, if n_0 is the number of leaf nodes and n_2 is the number of nodes of degree 2, then $n_0 = n_2+1$.

Proof:

Let n_1 be the number of nodes of degree 1 and n the total number of nodes, we have

$$n= n_0+ n_1 + n_2 \qquad (5.1)$$

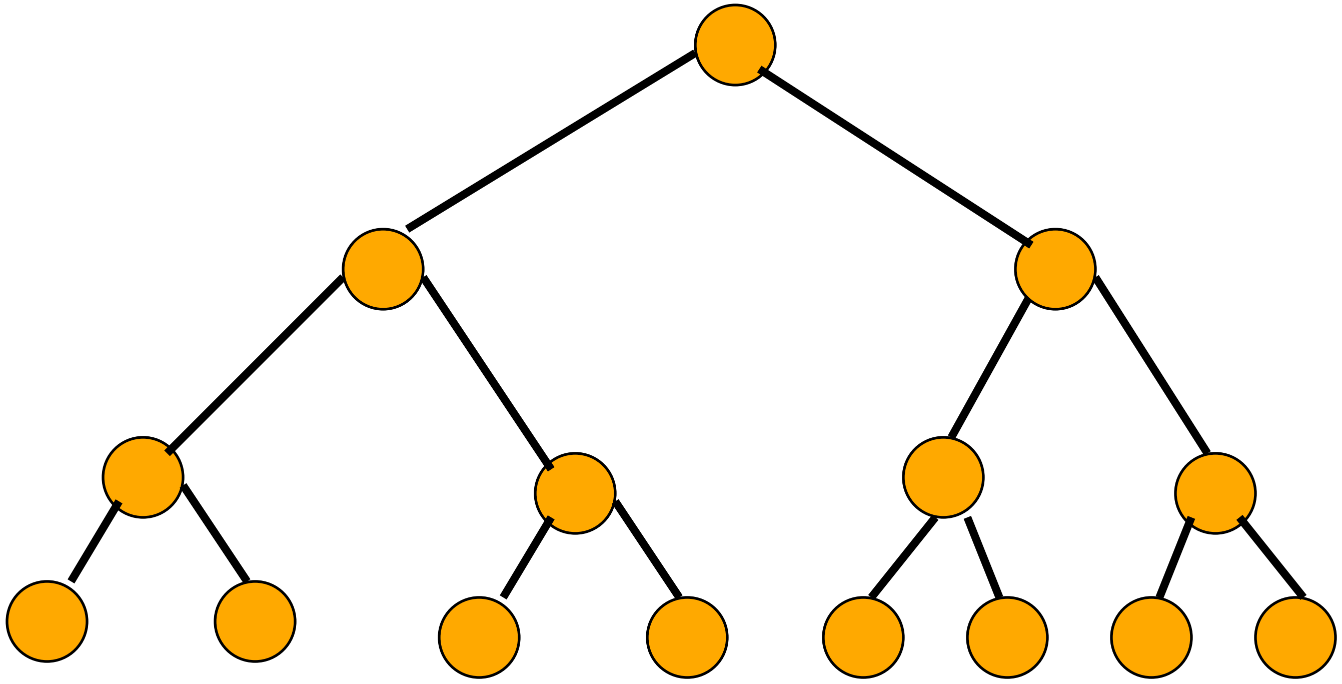
Each node except for the root has a branch leading into it. If B is the number of branches, then $n = B+1$. And also $B = n_1 + 2n_2$, hence

$$n= n_1+ 2n_2 + 1 \qquad (5.2)$$

(5.1) – (5.2): $0= n_0 - n_2 -1$, i.e., $n_0= n_2+1$.

Full Binary Tree

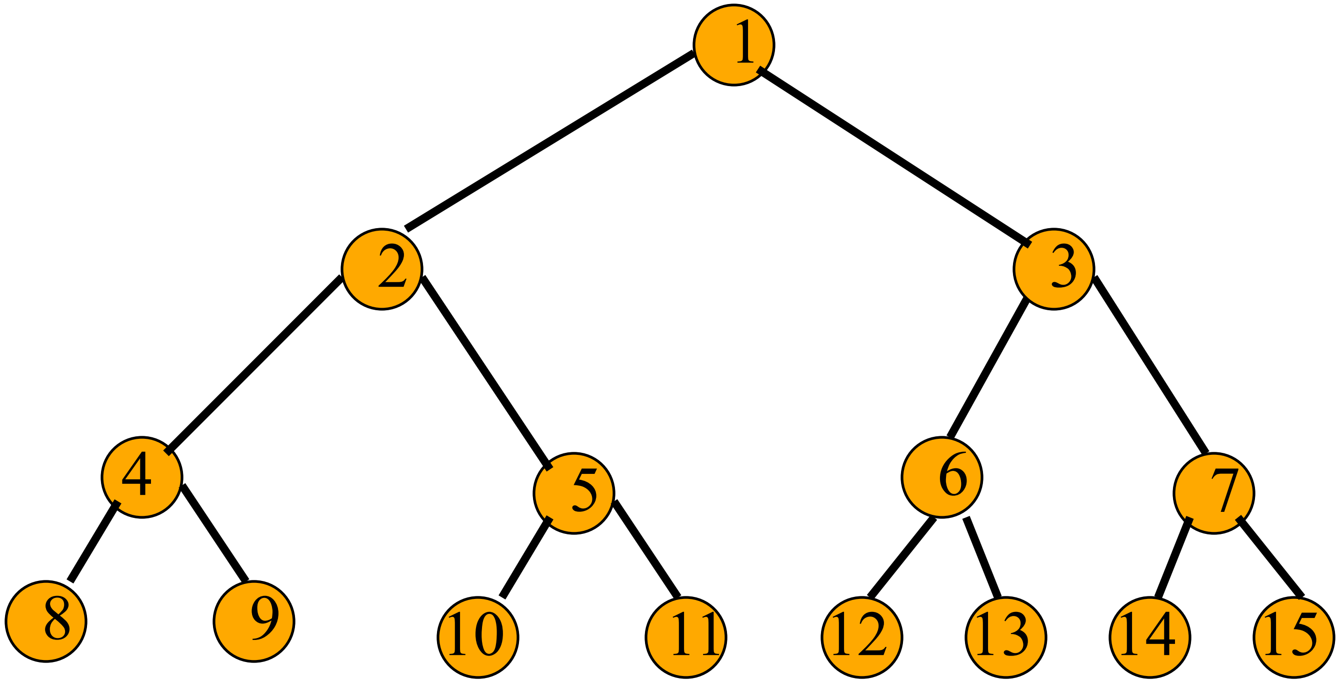
- A full binary tree of a given height h has $2^h - 1$ nodes.



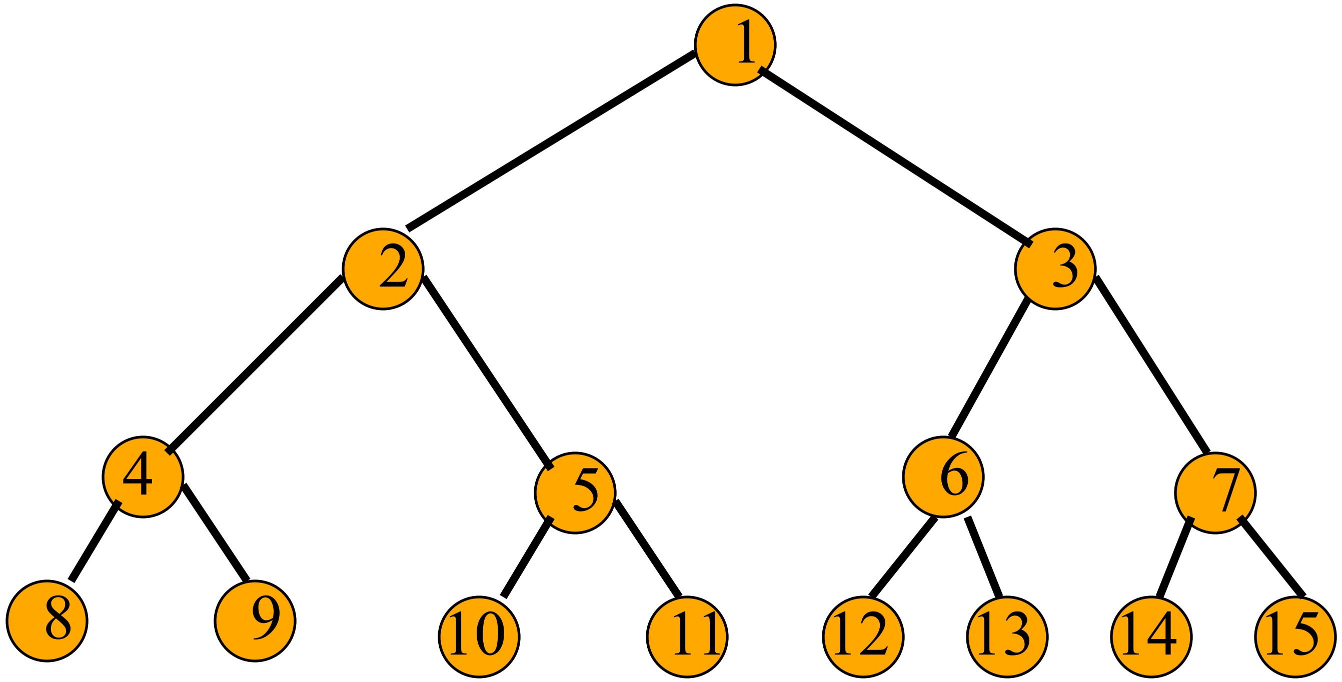
Height 4 full binary tree.

Numbering Nodes In A Full Binary Tree

- Number the nodes 1 through $2^h - 1$.
- Number by levels from top to bottom.
- Within a level number from left to right.

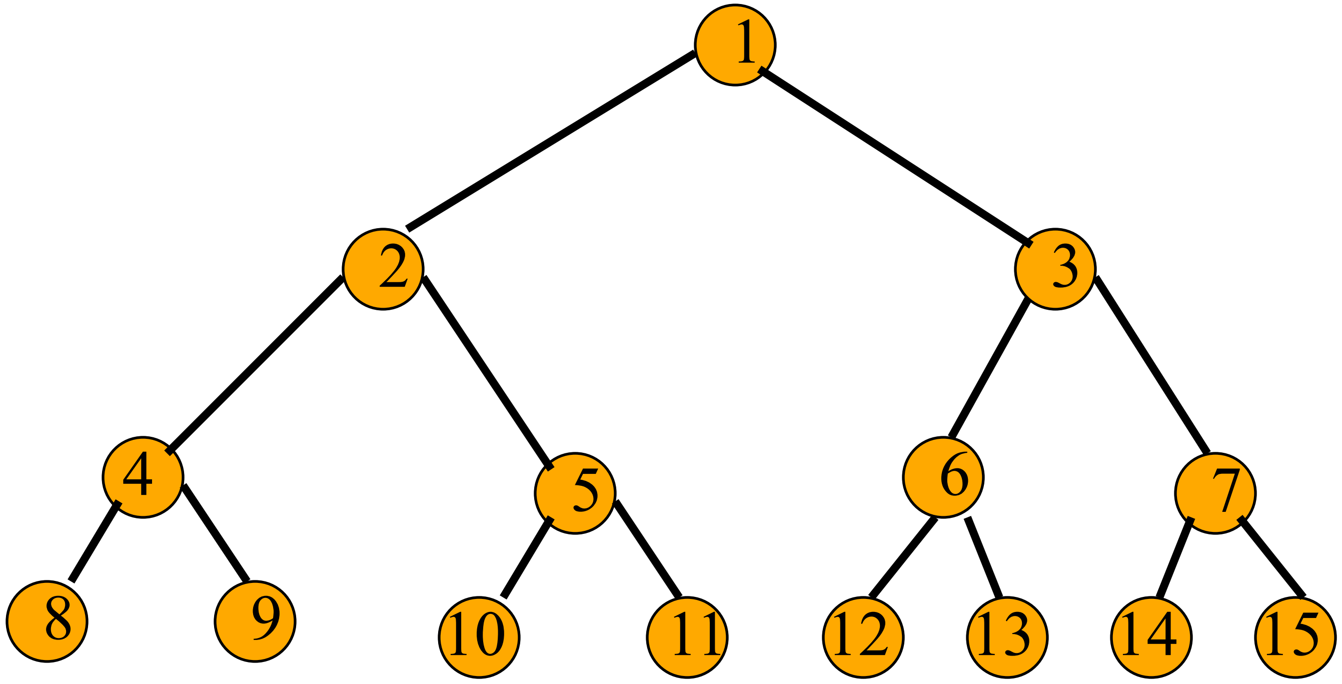


Node Number Properties



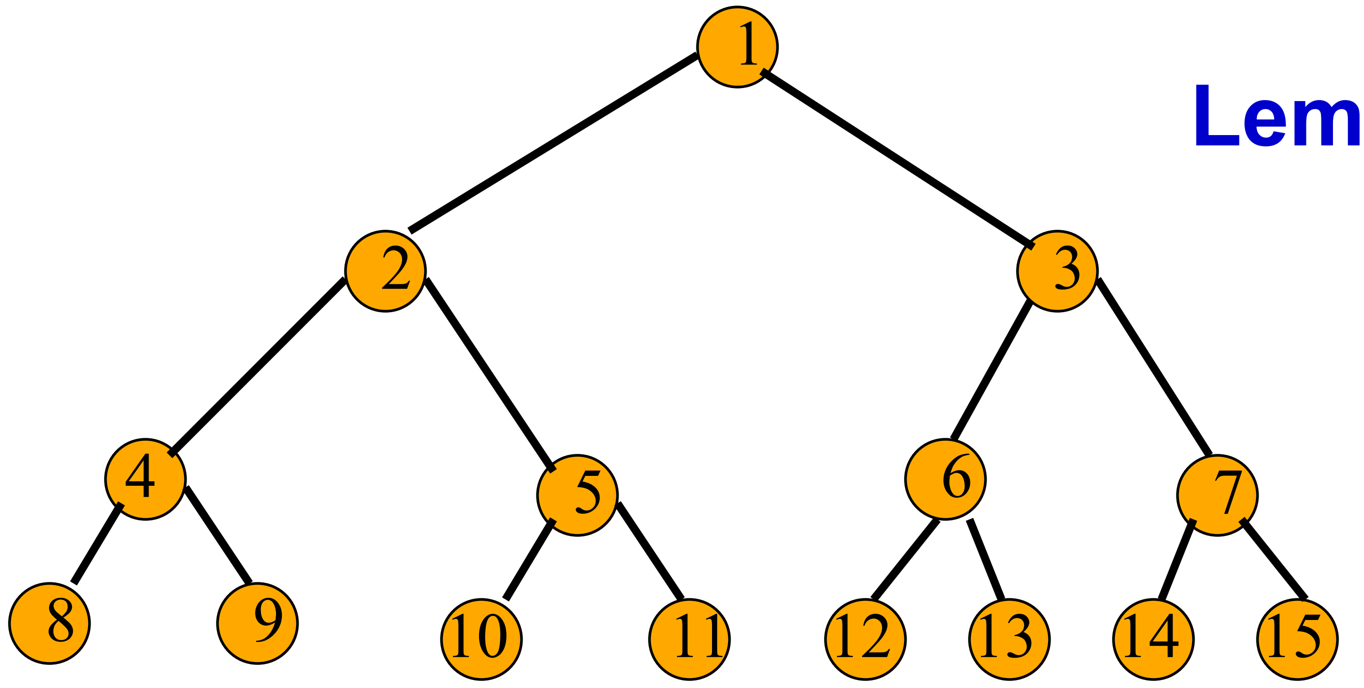
- Parent of node i is node i / 2, unless i = 1.
- Node 1 is the root and has no parent.

Node Number Properties



- Left child of node i is node $2i$, unless $2i > n$, where n is the number of nodes.
- If $2i > n$, node i has no left child.

Node Number Properties



Lemma 5.4

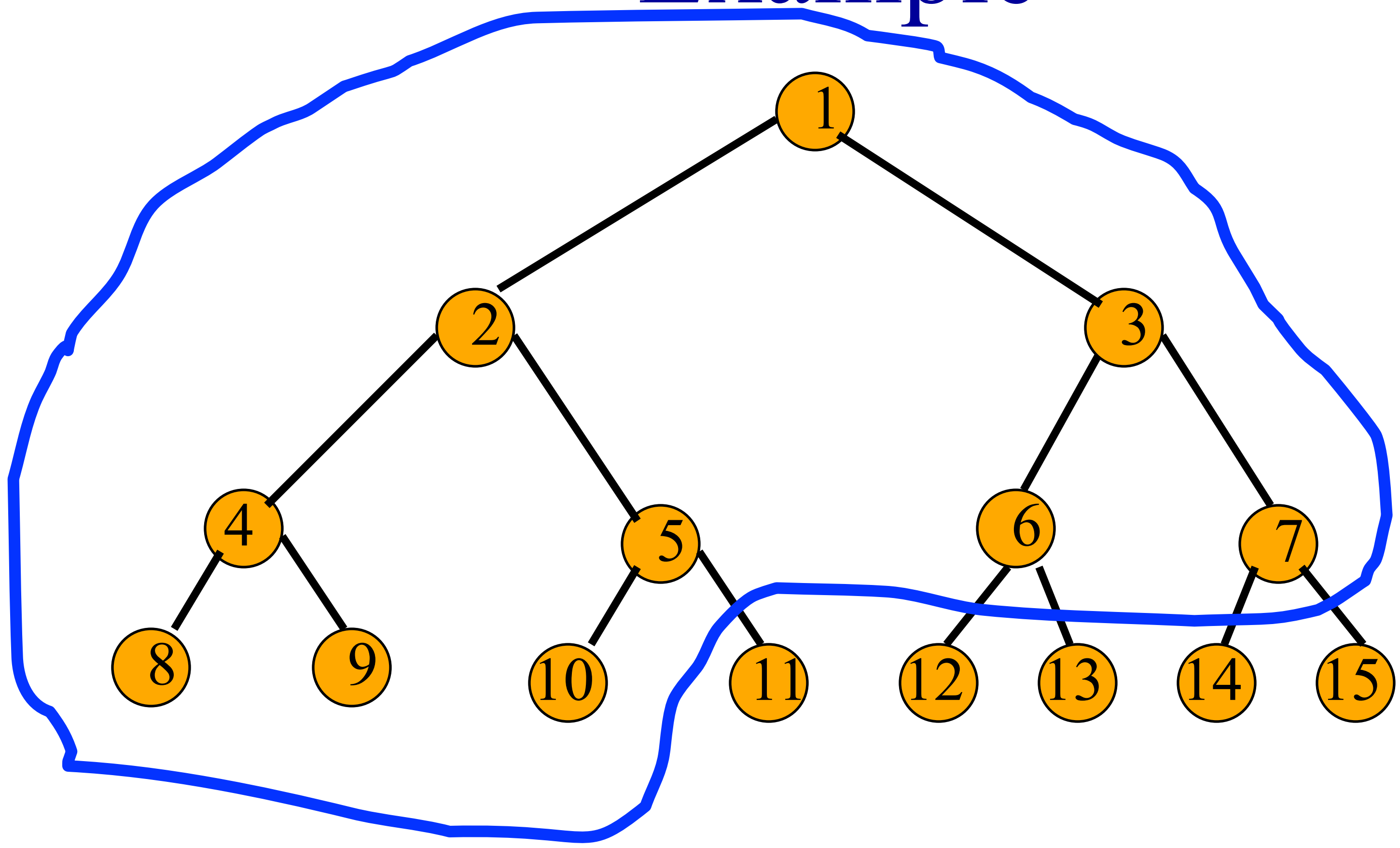
- Right child of node i is node 2i+1, unless 2i+1 > n, where n is the number of nodes.
- If 2i+1 > n, node i has no right child.

Complete Binary Tree With n Nodes

- Start with a full binary tree that has at least n nodes.
- Number the nodes as described earlier.
- The binary tree defined by the nodes numbered 1 through n is the unique n node complete binary tree.

Definition: a binary tree with n nodes and depth k is complete iff its nodes corresponding to the nodes numbered from 1 to n in the full binary tree of depth k.

Example



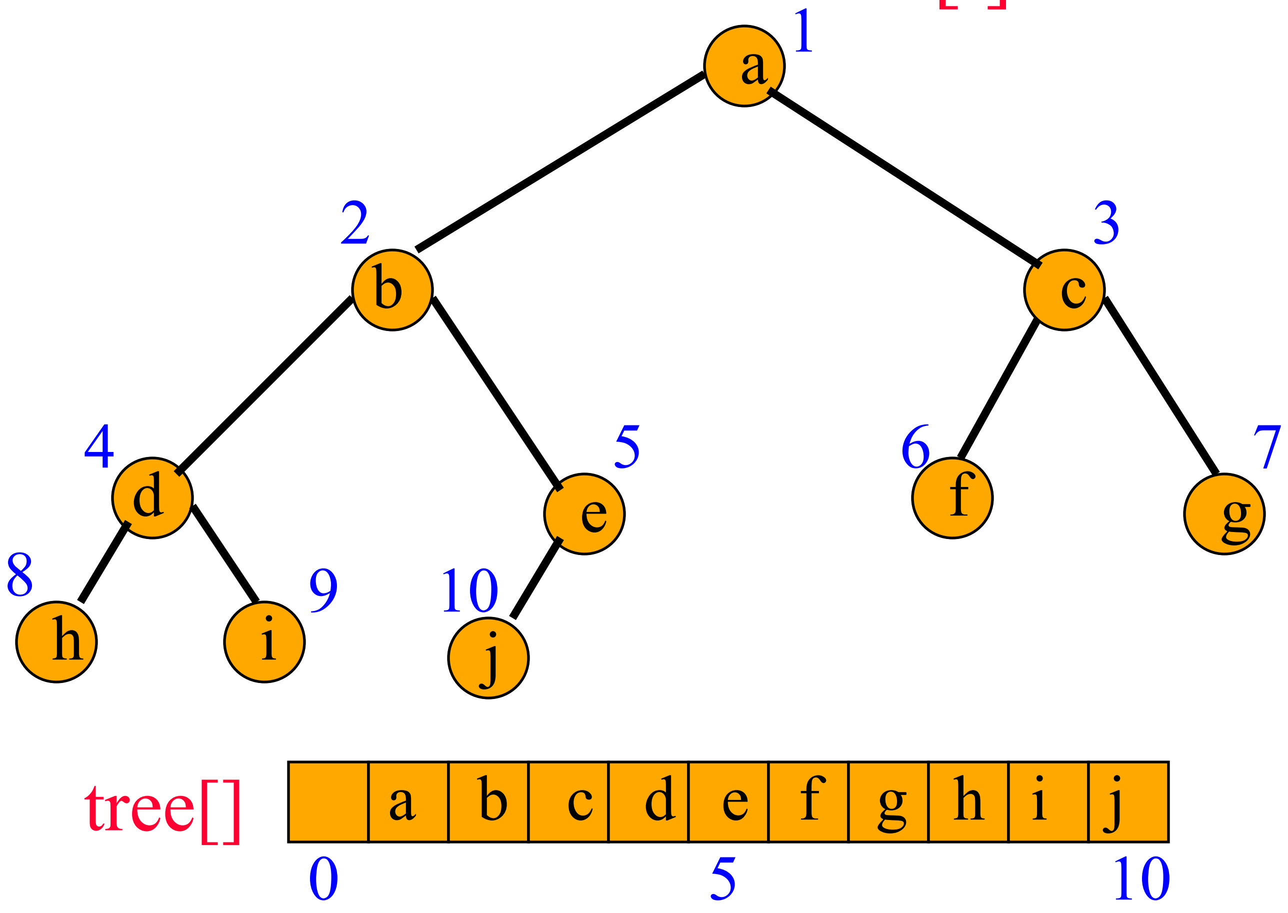
- Complete binary tree with 10 nodes.

Binary Tree Representation

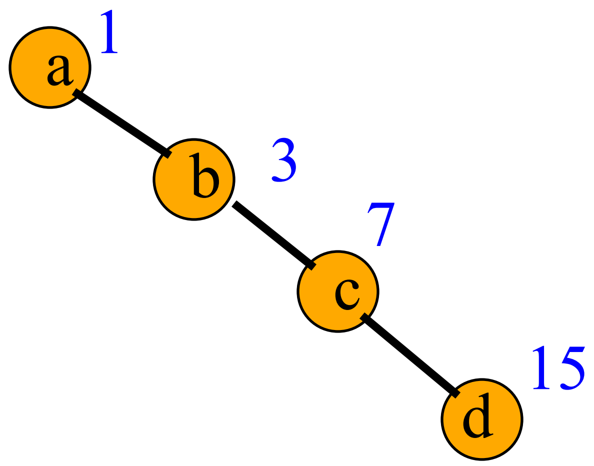
- Array representation.
- Linked representation.

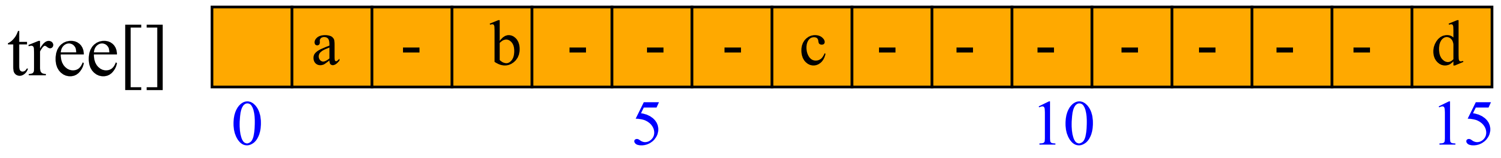
Array Representation

- Number the nodes using the numbering scheme for a full binary tree. The node that is numbered i is stored in tree[i].



Right-Skewed Binary Tree





- An n node binary tree needs an array whose length is between n+1 and 2^n.

Drawback?

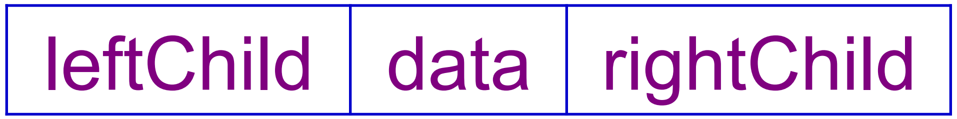
Linked Representation

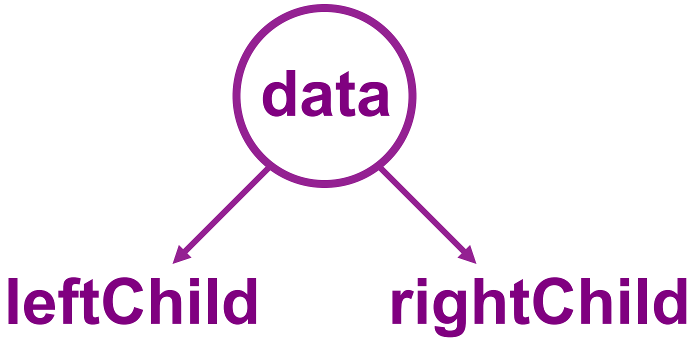
- Each binary tree node is represented as an object whose data type is TreeNode.
- The space required by an n node binary tree is n * (space required by one node).

Exponential v.s. Linear

The Class BinaryTreeNode

- **template <class** T> **class** Tree;
- **class** TreeNode **{**
- **friend class** Tree<T>;
- **public:**
-     TreeNode (T& e, TreeNode<T>* left, TreeNode<T>* right)
-         **{**data=e; leftChild=left; rightChild=right;**}**
- **private:**
-     T data;
-     TreeNode<Y>* leftChild;
-     TreeNode<Y>* rightChild;
- **};**

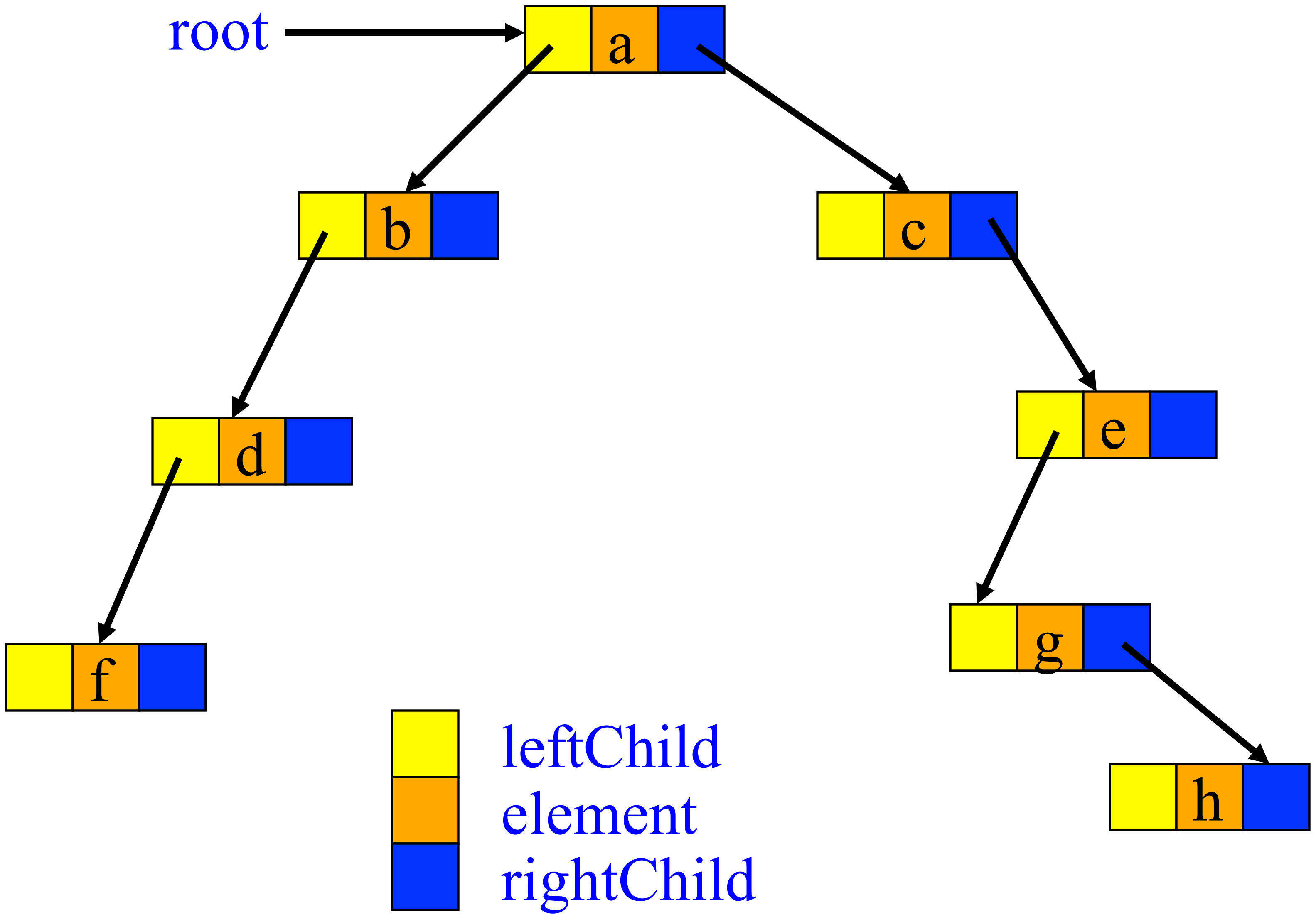




```
template <class T>
class Tree {
public:
    // Tree operations
…
private:
    TreeNode<T>* root;
};
```

If necessary, a 4th field, parent, may be included in the node.

Linked Representation Example



Some Binary Tree Operations

- Determine the height.
- Determine the number of nodes.
- Make a clone.
- Determine if two binary trees are clones.
- Display the binary tree.
- Evaluate the arithmetic expression represented by a binary tree.
- Obtain the infix form of an expression.
- Obtain the prefix form of an expression.
- Obtain the postfix form of an expression.

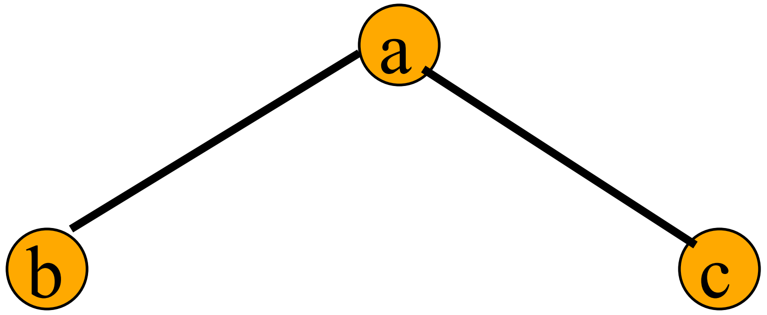
Binary Tree Traversal

- Many binary tree operations are done by performing a traversal of the binary tree.
- In a traversal, each element of the binary tree is visited exactly once.
- During the visit of an element, all action (make a clone, display, evaluate the operator, etc.) with respect to this element is taken.

Binary Tree Traversal Methods

- Preorder
- Inorder
- Postorder
- Level order

Preorder Example (visit = print)

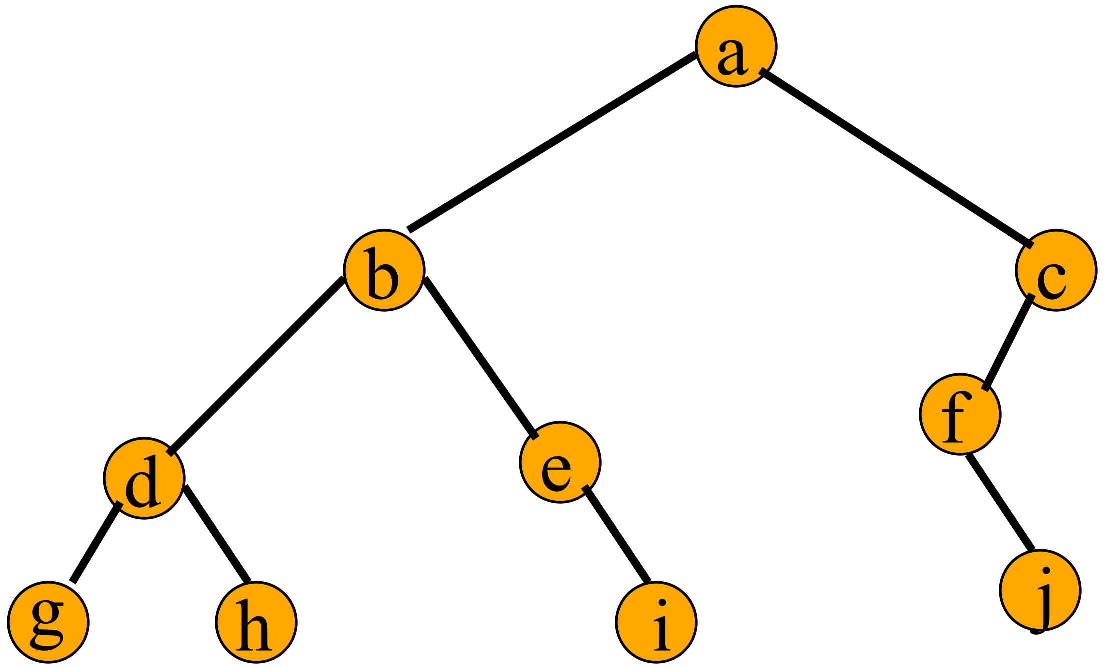


a b c

Preorder Traversal

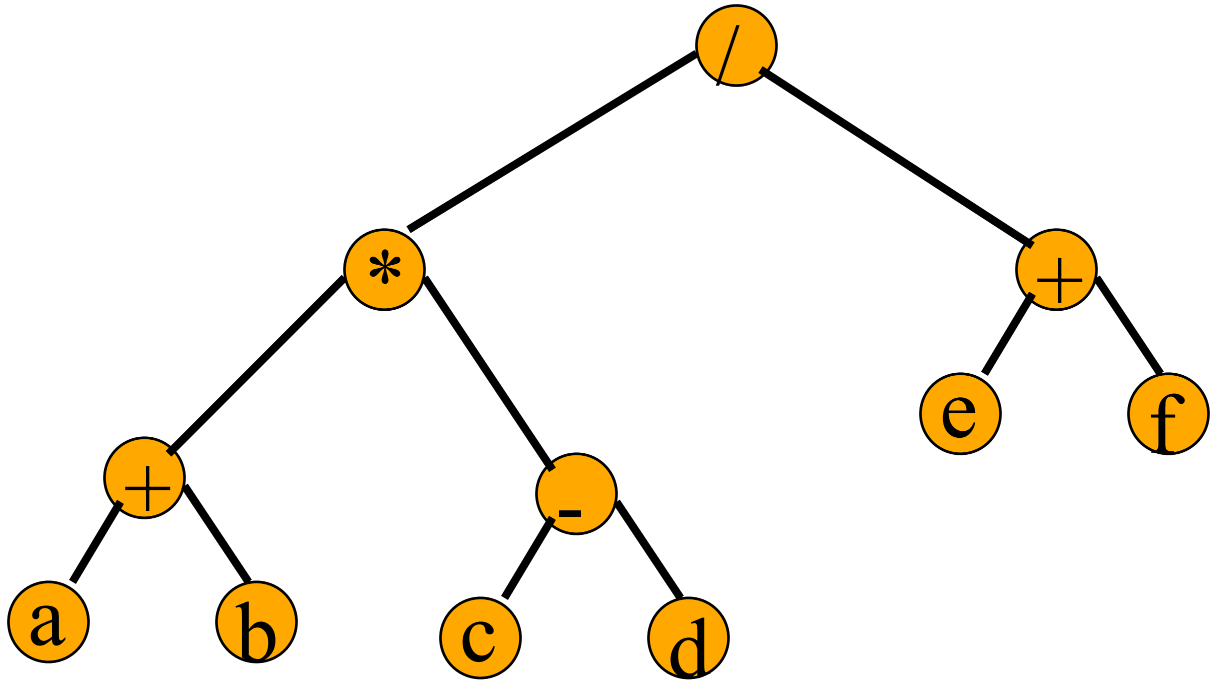
```
• template <class T>
• void Tree<T>::Preorder()
• { // Driver.
•     Preorder(root);
• }
• template <class T>
• void Tree<T>::Preorder(TreeNode<T>* currentNode)
• { // workhorse.
•      if (currentNode) {
•          Visit(currentNode);
•          Preorder(currentNode→leftChild);
•          Preorder(currentNode→rightChild);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

Preorder Example (visit = print)



a b d g h e i c f j

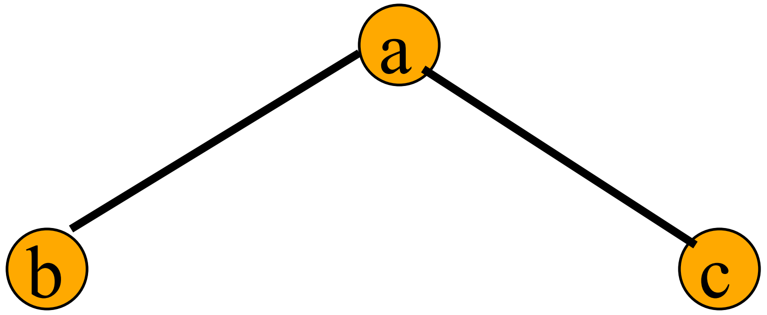
Preorder Of Expression Tree



/ * + a b - c d + e f

Gives prefix form of expression!

Inorder Example (visit = print)

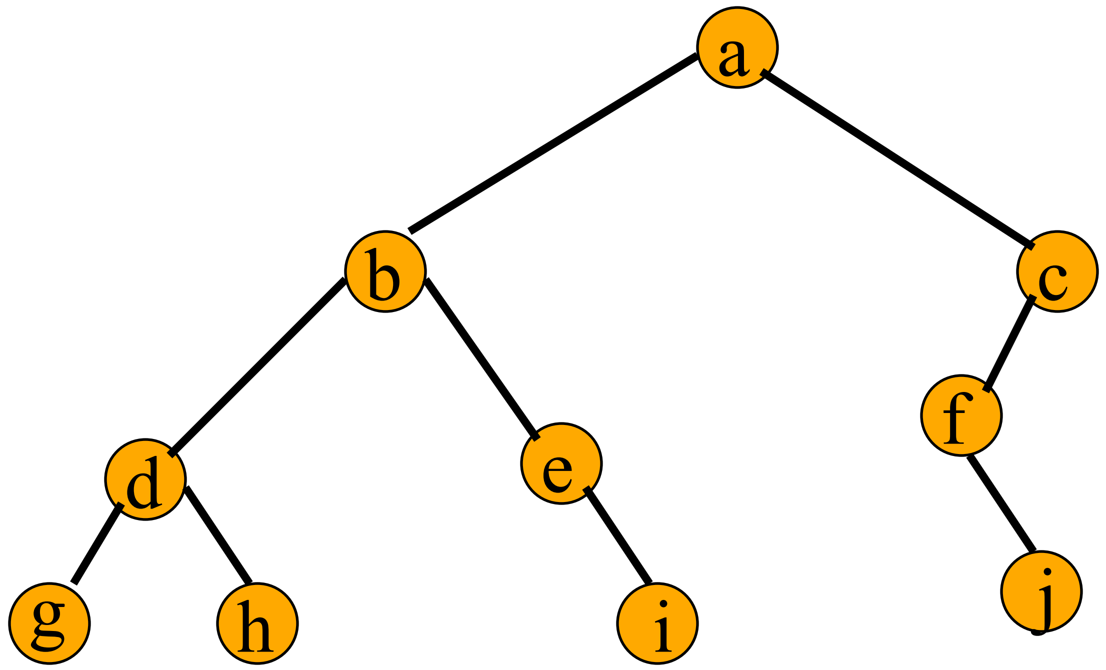


b a c

Inorder Traversal

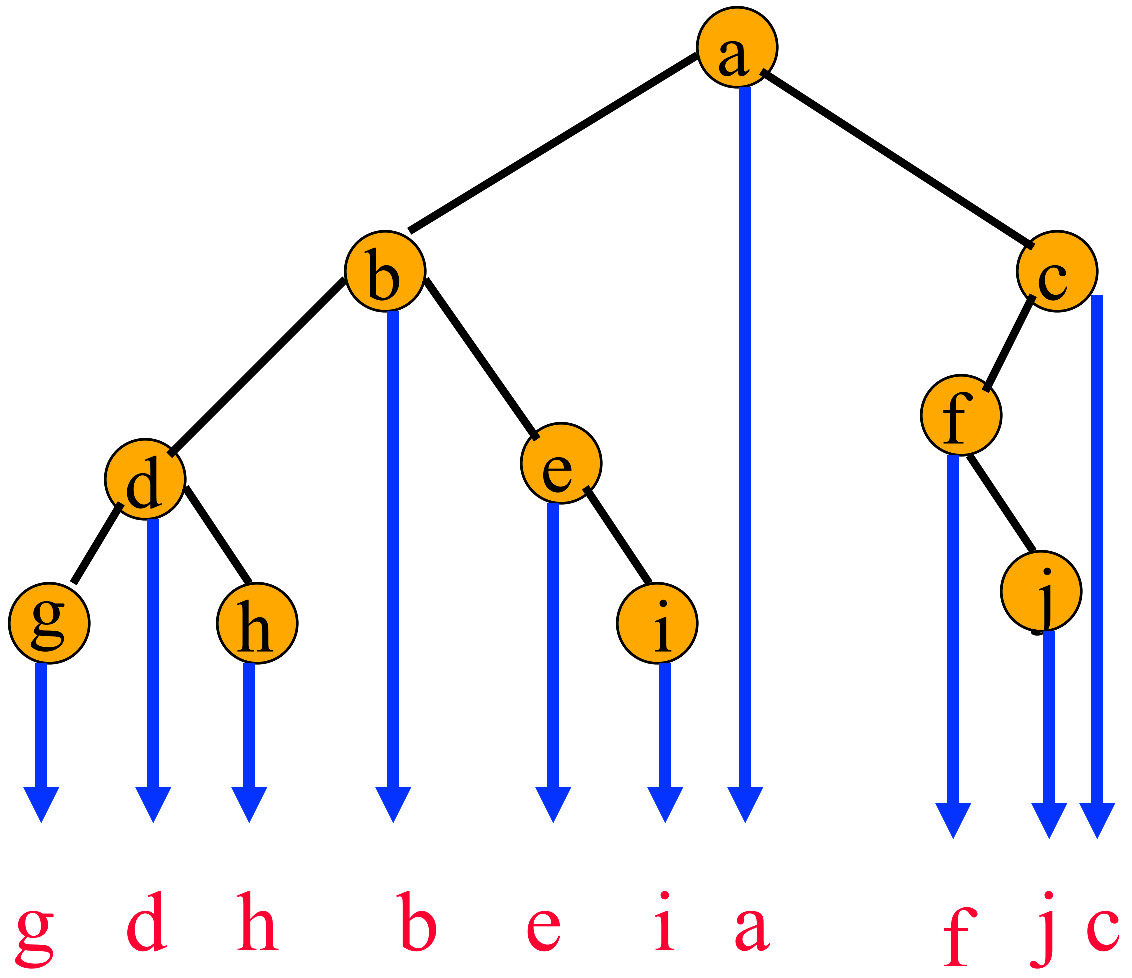
```
template <class T>
void Tree<T>::Inorder()
{ // driver as a public member
   Inorder(root);
}
template <class T>
void Tree<T>::Inorder (TreeNode<T>* currentNode)
{ // workhorse as a private member of Tree
    if (CurrentNode) {
        Inorder(currentNode→leftChild);
        Visit(currentNode)
        Inorder(currentNode→rightChild);
     }
}
```

Inorder Example (visit = print)

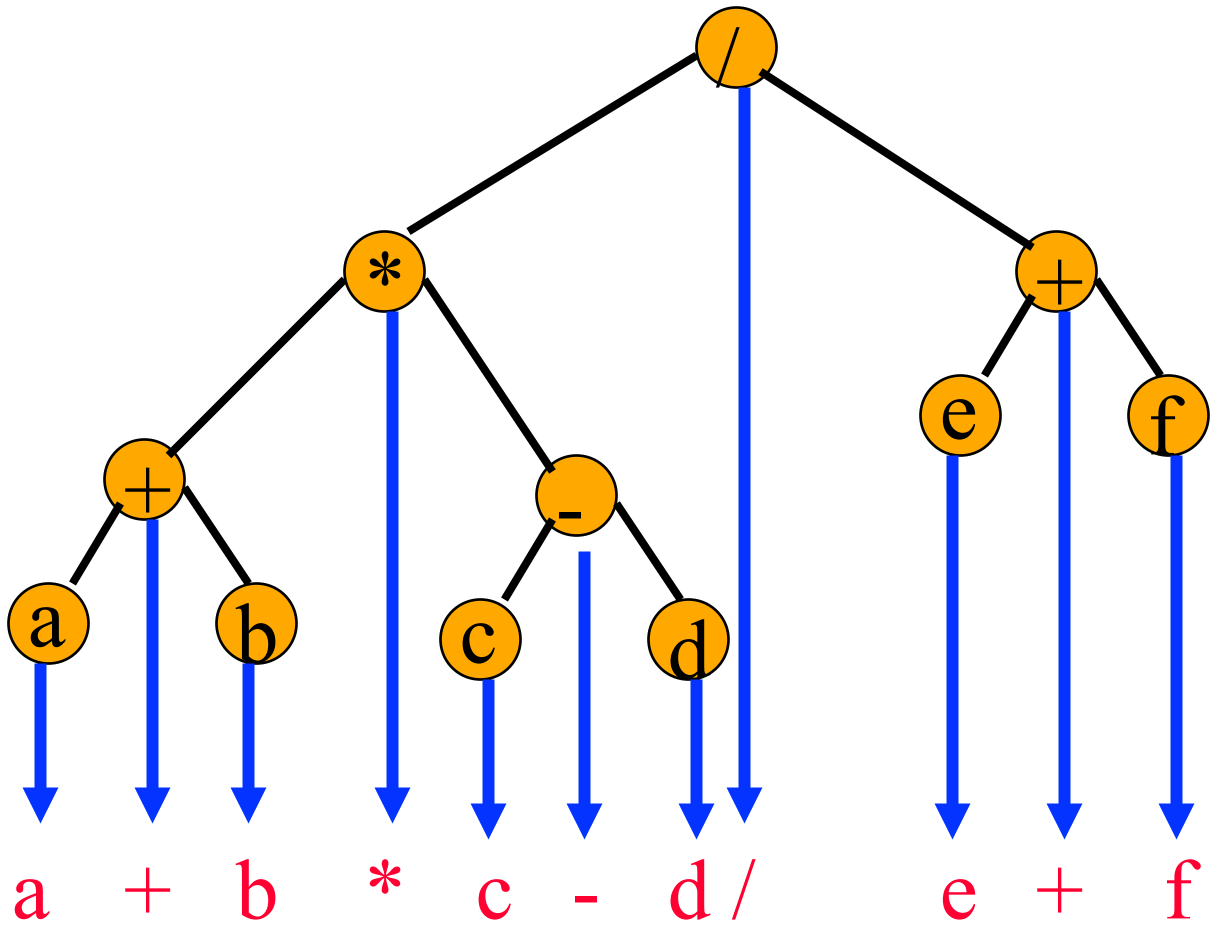


g d h b e i a f j c

Inorder By Projection

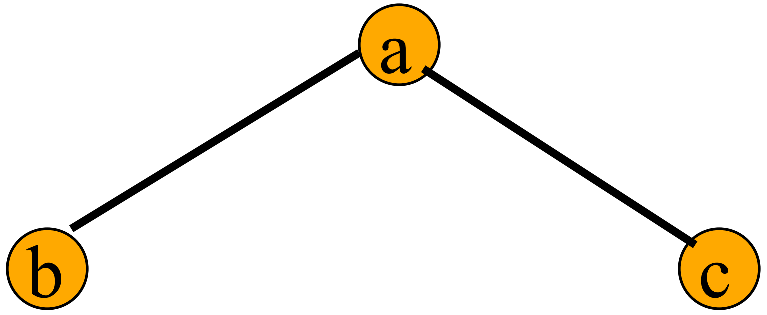


Inorder Of Expression Tree



Gives infix form of expression (sans parentheses)!

Postorder Example (visit = print)

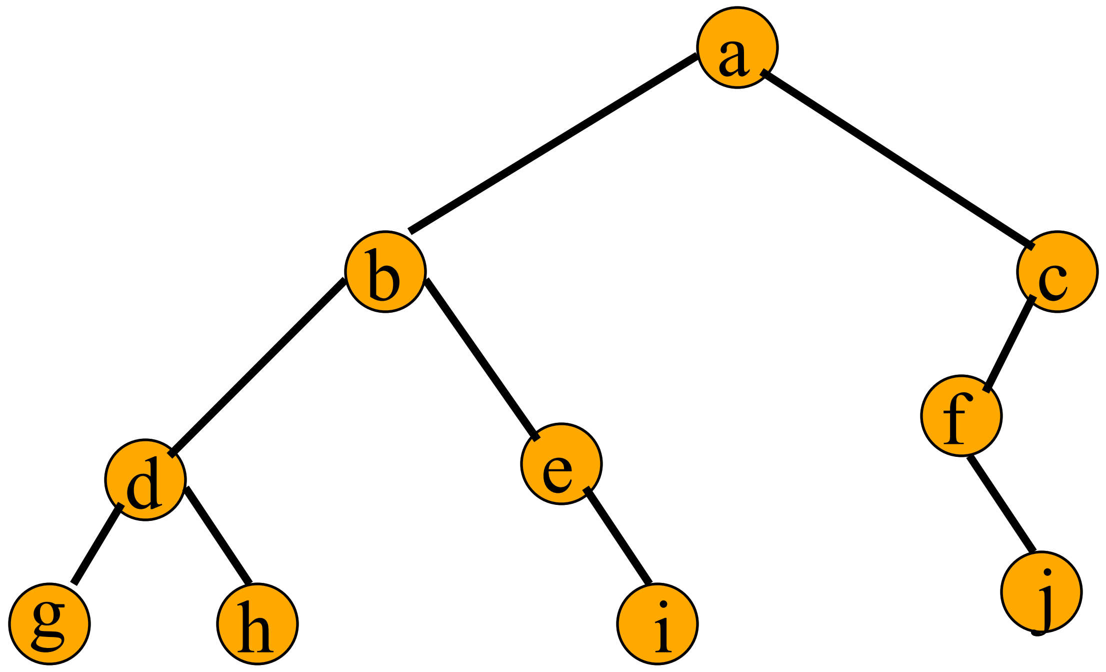


b c a

Postorder Traversal

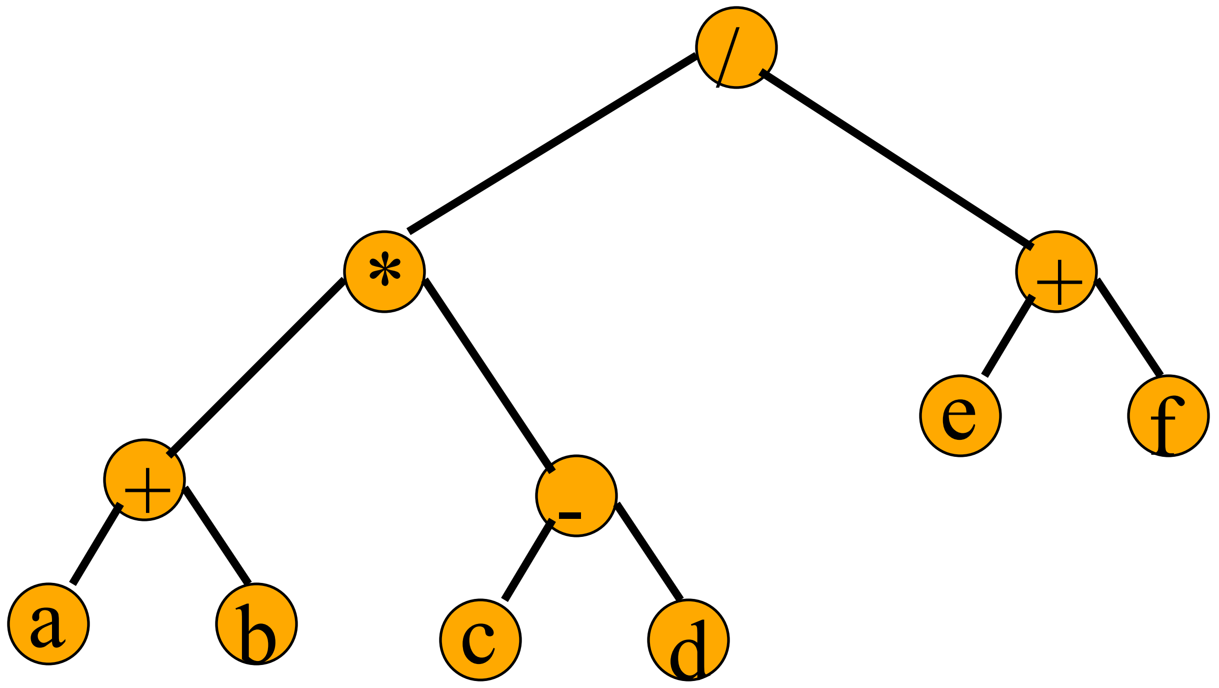
```
template <class T>
void Tree<T>::Postorder()
{ // Driver.
    Postorder(root);
}
template <class T>
void Tree<T>::Postorder (TreeNode<T>* currentNode)
{ // Workhorse.
      if (currentNode) {
         Postorder(currentNode→leftChild);
         Postorder(currentNode→rightChild );
         Visit(currentNode);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

Postorder Example (visit = print)



g h d i e b j f c a

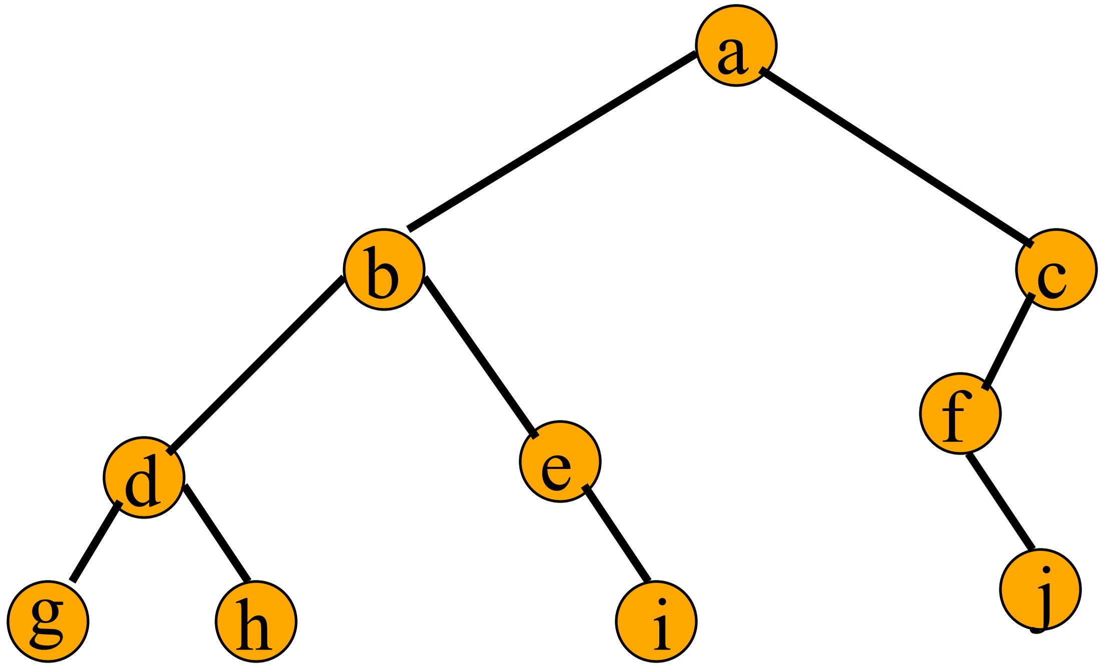
Postorder Of Expression Tree



a b + c d - * e f + /

Gives postfix form of expression!

Traversal Applications

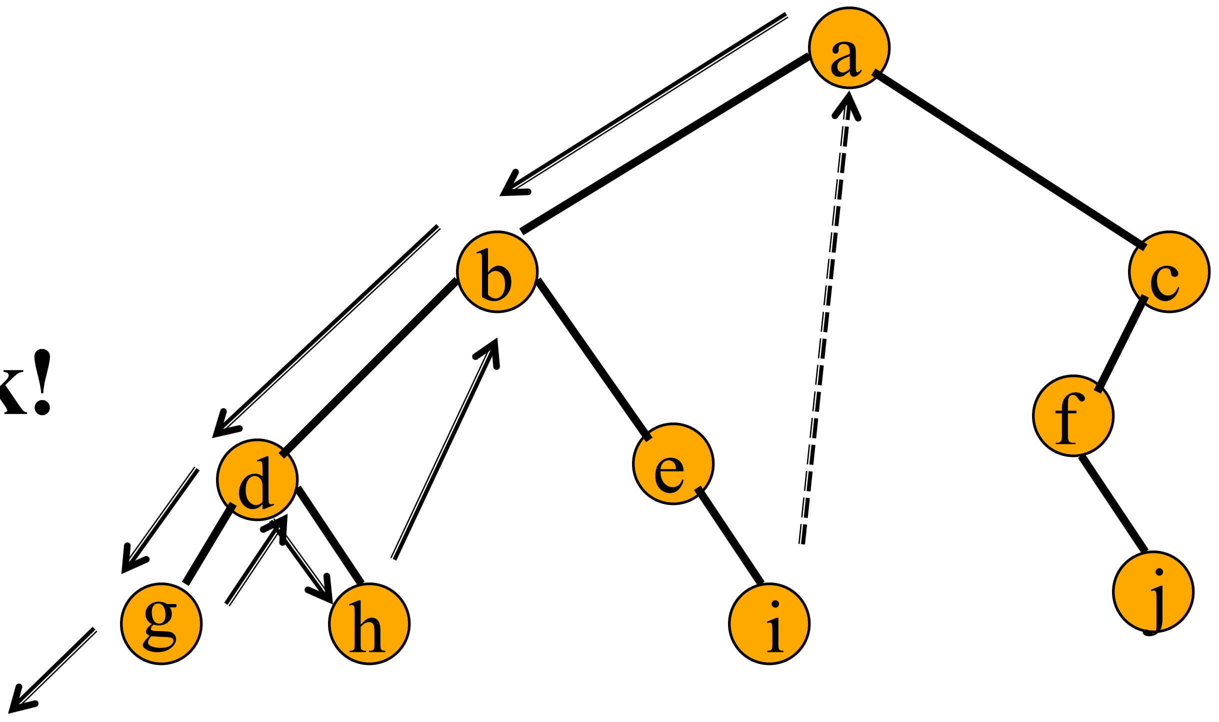


- Make a clone.
- Determine height.
- Determine number of nodes.

- Int h(T * root)
- {
  - If(root == null) return 0;
  - Else
  - {
    - Int hl = h (root->leftchild);
    - Int hr = h(root->rightchild);
    - Return hl + hr + 1;
  - }
- }

Iterative Inorder Traversal

- Tree iterator
    - Access nodes one by one
- Non-recursive tree traversal algorithm
    - Inorder
- Data structure:
    - Stack!



```
template <class T>
void Tree<T>::NonrecInorder()
{ // Nonrecursive inorder traversal using a stack
  Stack<TreeNode<T>*> s; // declare and initialize a stack
  TreeNode<T>* currentNode=root;
  while (1) {
     while (currentNode) { // move down leftChild
        s.Push(currentNode); // add to stack
        currentNode=currentNode→leftChild;
     }
     If (s.IsEmpty()) return;
     currentNode=s.Top();
     s.Pop(); // delete from stack
     Visit(currentNode);
     currentNode=currentNode→rightChild;
  }
}
```

The NonrecInorder USES-A template stack.

Definition: A data object of Type A USES-A data object of Type B if a Type A object uses a Type B object to perform a task. Typically, a Type B object is employed in a member function of Type A.

USES-A is similar to IS-IMPLEMENTED-IN-TERMS-OF, but the degree of using the Type B object is less.

Analysis of NonrecInorder:

- n---the number of nodes in the tree.
- every node is placed on the stack once, line 8, 9 and 11 to 15 are executed n times.
- currenetNode will equal 0 once for every 0 link, which is $2n_0+ n_1=n_0+n_1+n_2+1=n+1$.

The computing time: O(n).

The space required for the stack is equal to the depth of the tree.

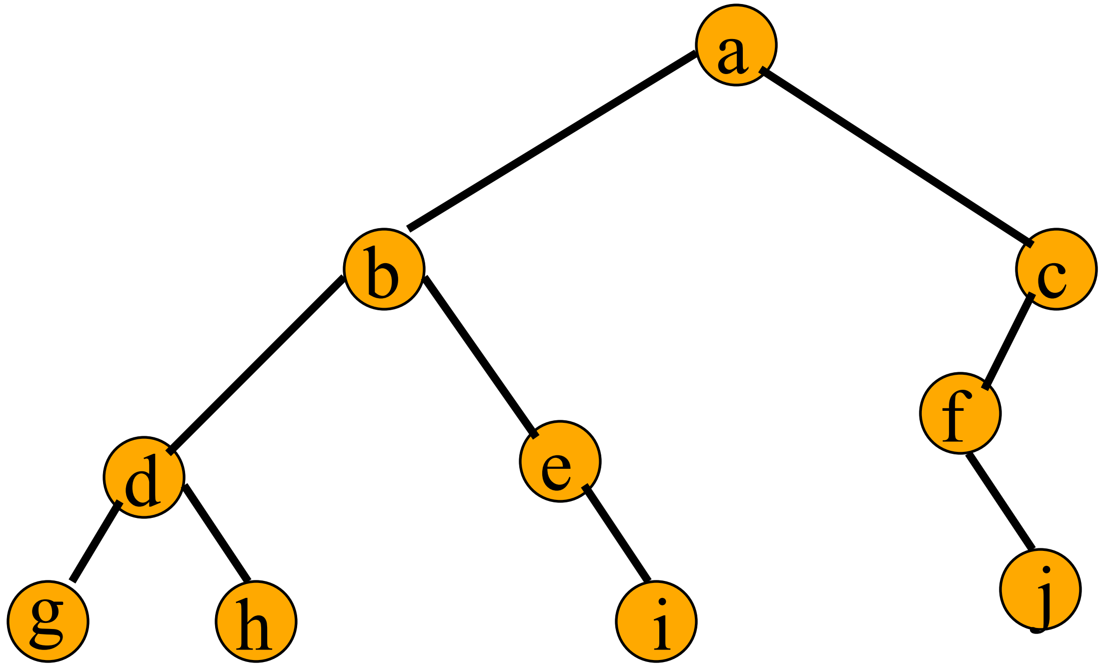
Now we use the function NonrecInorder to obtain an inorder iterator for a tree.

The key observation is that each iteration of the while loop of line 6-16 yields the next element in the inorder traversal of the tree.

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

```
T* InorderIterator::Next()
{
    while (currentNode) {
        s.Push(currentNode);
        currentNode=currentNode→LeftChild;
    }
    if (s.IsEmpty()) return 0;
    currentNode=s.Top();
    T& temp=currentNode→data;
    currentNode=currentNode→rightChild;
    return &temp;
}
```

Level-Order Example (visit = print)



Storage?

a b c d e f g h i j

FIFO Queue

Level Order

```
Let t be the tree root.
while (t != null)
{
    visit t and put its children on a FIFO queue;
    remove a node from the FIFO queue and
    call it t;
    // remove returns null when queue is empty
}
```

```
Q q;
Q.push_back(root);
While(!q.is_empty())
{
    Node * ptr = q.pop_front();
    Visit(ptr);
    If(ptr->left_child) q.push_back(ptr->left_child);
    If(ptr->right_child) q.push_back(ptr->left_right);
}
```

Additional Binary Tree Operations

Copying Binary Trees

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

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

Testing Equality

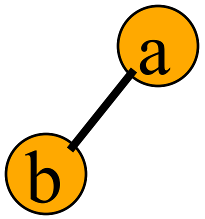
```
template <class T>
bool Tree<T>::operator==(const Tree& t) const
{
   return Equal(root, t.root);
}

template <class T>
bool Tree<T>::Equal(TreeNode<T>* a, TreeNode<T>* b)
{// Workhorse-
   if ((!a) && (!b)) return true; // both a and b are 0
   return (a && b          // both a and b are non-0
      && (a→data == b→data)                 //data is the same
      && Equal(a→leftChild, b→leftChild)           //left equal
      && Equal(a→rightChild, b→rightChild));     //right equal
}
```

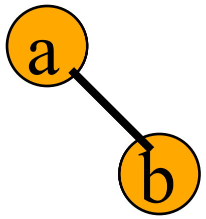
Binary Tree Construction

- Suppose that the elements in a binary tree are distinct.
- Can you construct the binary tree from which a given traversal sequence came?
- When a traversal sequence has more than one element, the binary tree is not uniquely defined.
- Therefore, the tree from which the sequence was obtained cannot be reconstructed uniquely.

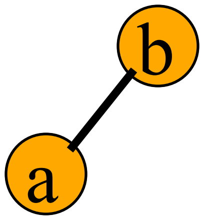
Some Examples

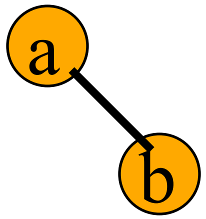
preorder
= ab



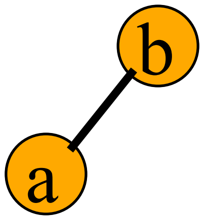


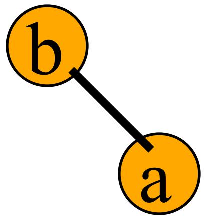
inorder
= ab



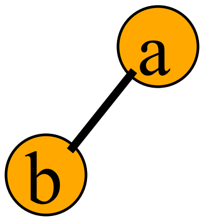


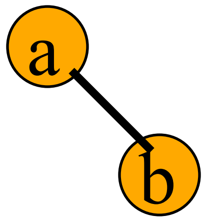
postorder
= ab





level order
= ab





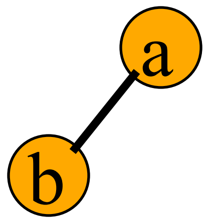
Binary Tree Construction

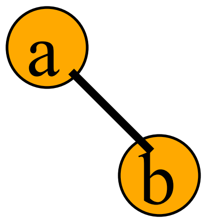
- Can you construct the binary tree, given two traversal sequences?
- Depends on which two sequences are given.

Preorder And Postorder

preorder = ab

postorder = ba

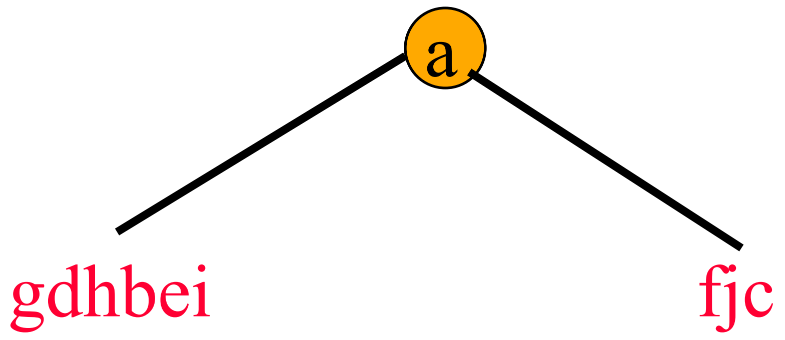




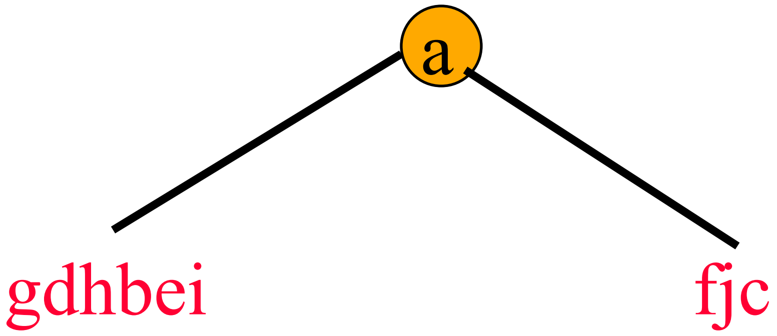
- Preorder and postorder do not uniquely define a binary tree.
- Nor do preorder and level order (same example).
- Nor do postorder and level order (same example).

Inorder And Preorder

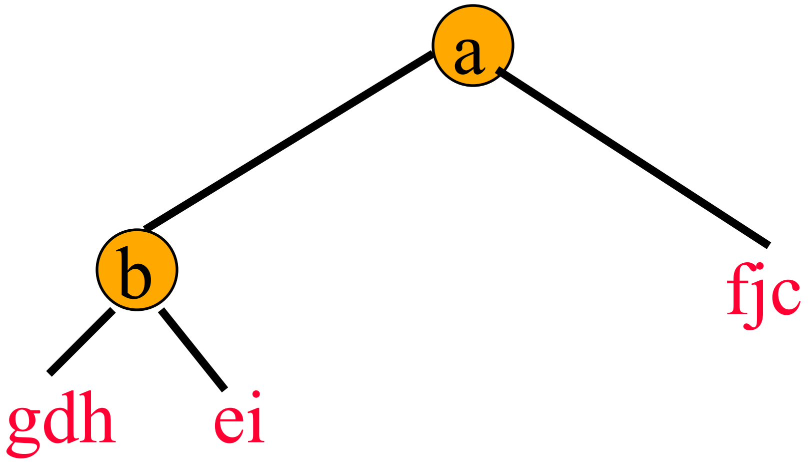
- inorder = g d h b e i a f j c
- preorder = a b d g h e i c f j
- Scan the preorder left to right using the inorder to separate left and right subtrees.
- a is the root of the tree; gdhbei are in the left subtree; fjc are in the right subtree.



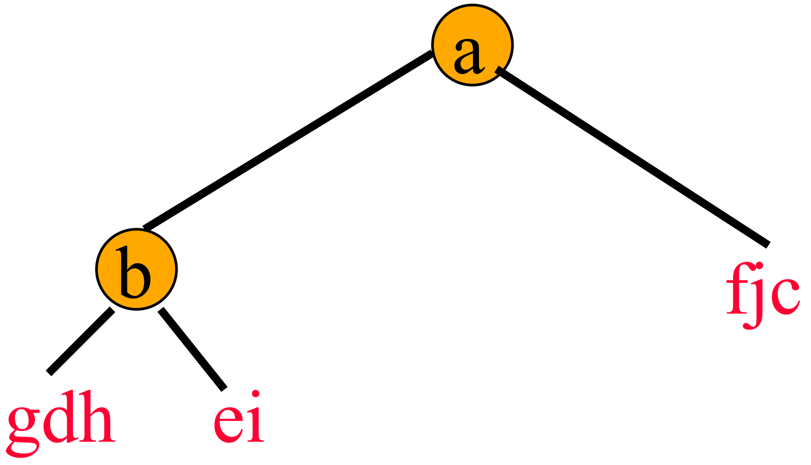
Inorder And Preorder



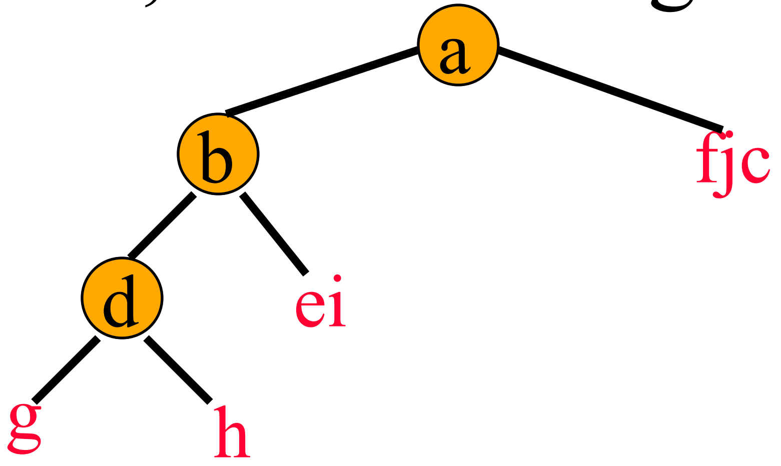
- preorder = a b d g h e i c f j
- b is the next root; gdh are in the left subtree; ei are in the right subtree.



Inorder And Preorder



- preorder = a b d g h e i c f j
- d is the next root; g is in the left subtree; h is in the right subtree.



Inorder And Postorder

- Scan postorder from right to left using inorder to separate left and right subtrees.
- inorder = g d h b e i a f j c
- postorder = g h d i e b j f c a
- Tree root is a; gdhbei are in left subtree; fjc are in right subtree.

Inorder And Level Order

- Scan level order from left to right using inorder to separate left and right subtrees.
- inorder = g d h b e i a f j c
- level order = a b c d e f g h i j
- Tree root is a; gdhbei are in left subtree; fjc are in right subtree.

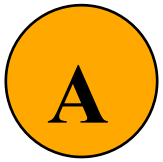
Exercises:

1. A binary tree, inorder = g d h b e i a f j c, postorder = g h d i e b j f c a, construct the tree.

2. A Tree with degree 3, Preorder = ABCDEFGHI, Postorder=BCGHFEIDA, construct the tree.

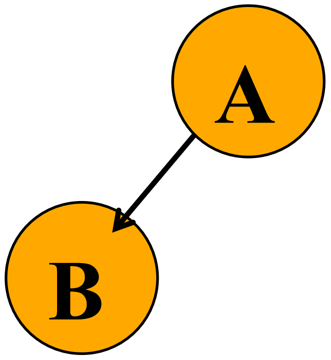
Preorder = ABCDEFGHI,

Postorder=BCGHFEIDA



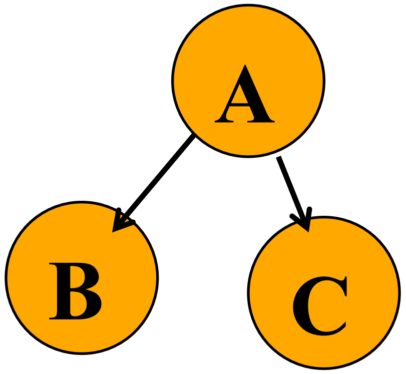
Preorder = ABCDEFGHI,

Postorder=BCGHFEIDA



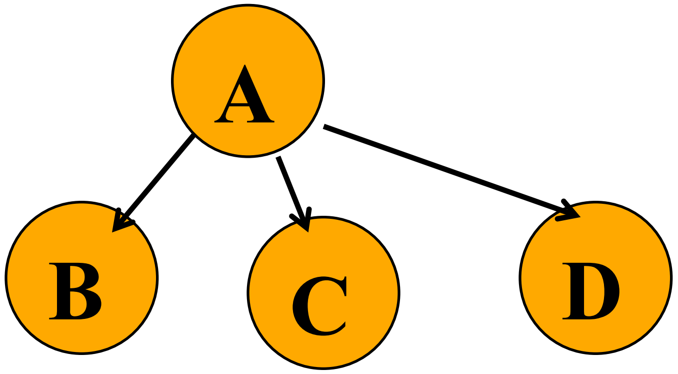
Preorder = ABCDEFGHI,

Postorder=BCGHFEIDA



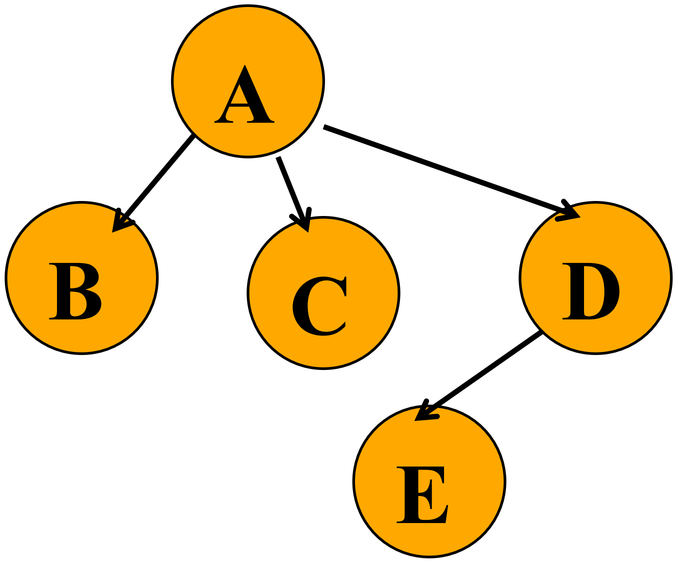
Preorder = ABCDEFGHI,

Postorder=BCGHFEIDA



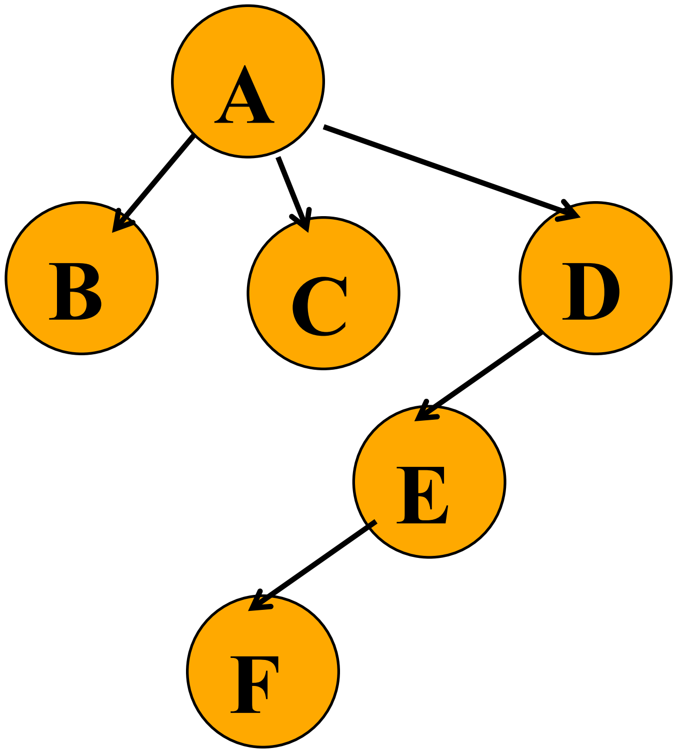
Preorder = ABCDEFGHI,

Postorder=BCGHFEIDA



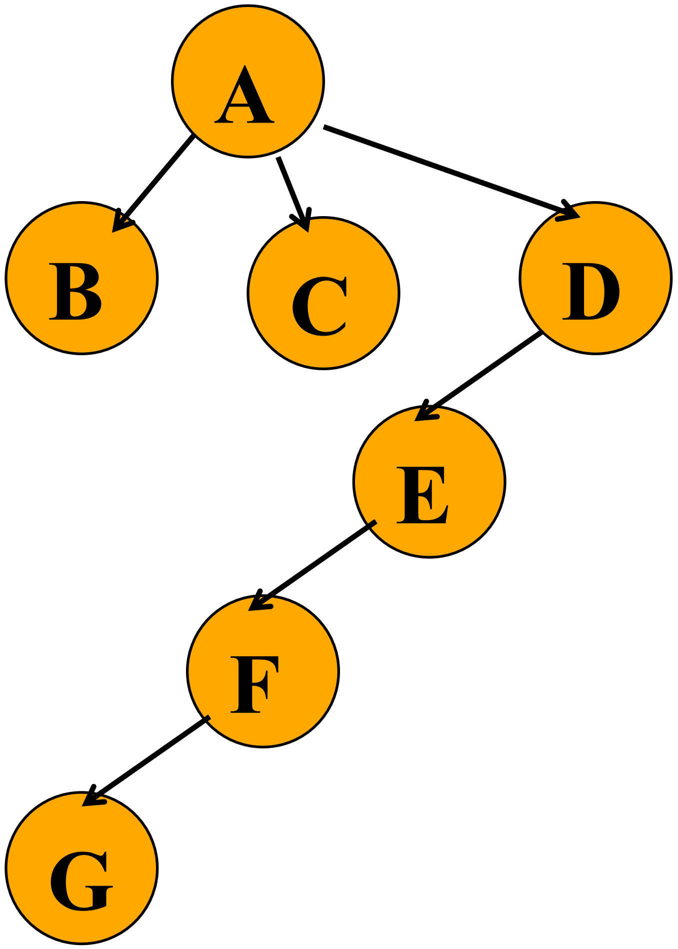
Preorder = ABCDEFGHI,

Postorder=BCGHFEIDA



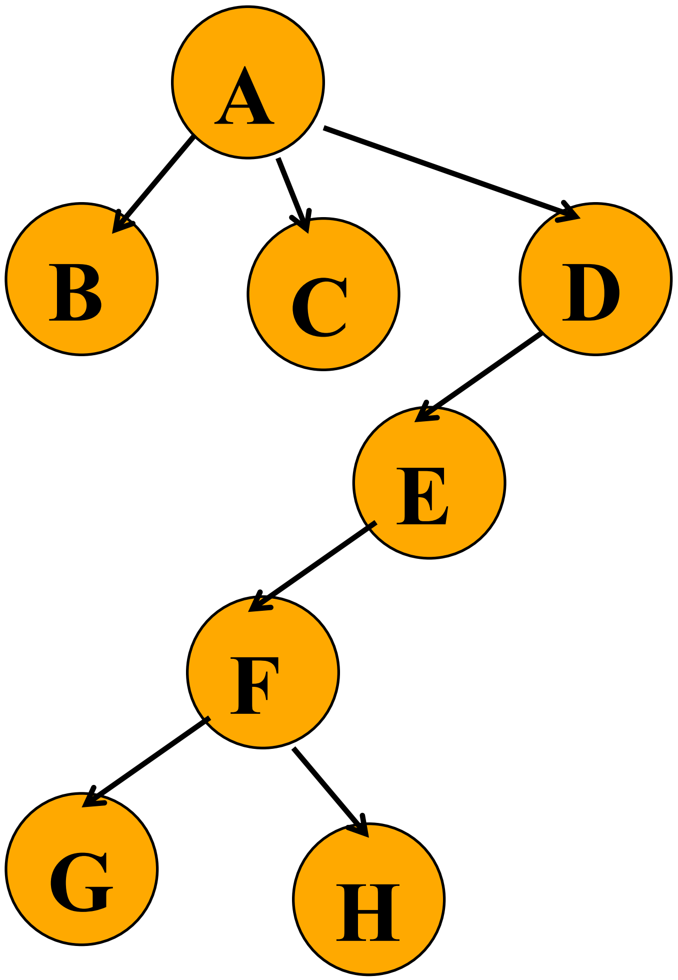
Preorder = ABCDEFGHI,

Postorder=BCGHFEIDA

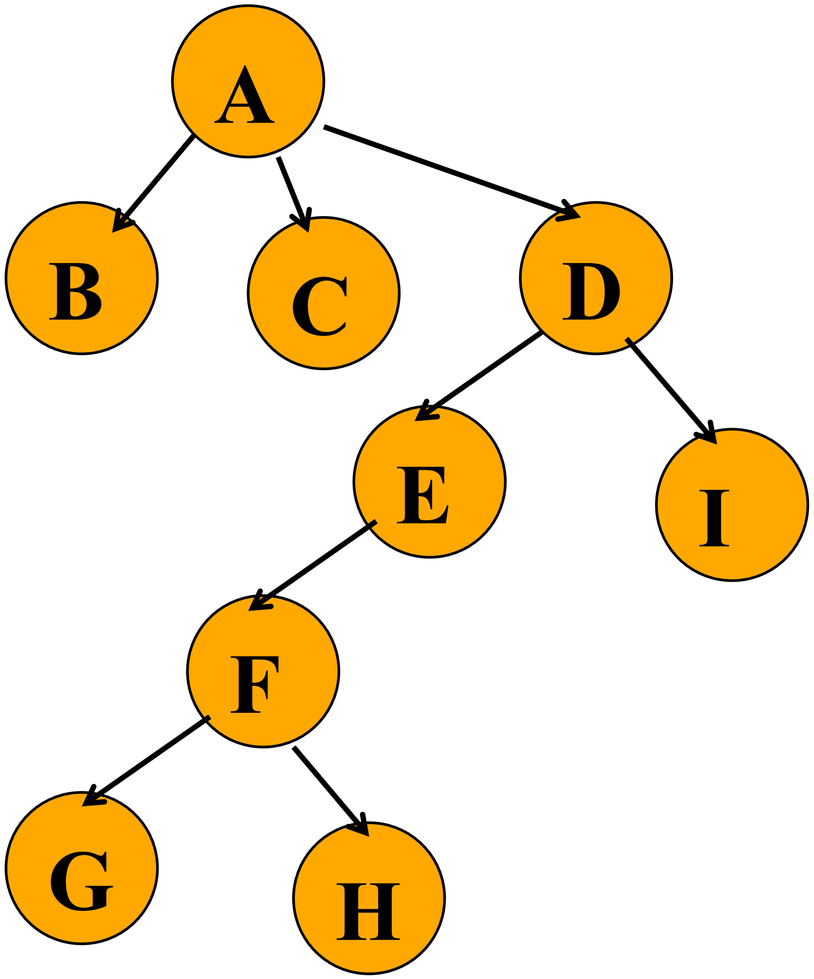


Preorder = ABCDEFGHI,

Postorder=BCGHFEIDA



Preorder = ABCDEFGHI, Postorder=BCGHFEIDA



Exercises: P267-4, P267-6, P272-1, P273-4

Experiment: P267-10

Binary Search Trees

- Dictionary Operations:
  - get(key)
  - put(key, value)
  - remove(key)
- Additional operations:
  - ascend()
  - get(index) (indexed binary search tree)
  - remove(index) (indexed binary search tree)
- **no two pairs have the same key**

ADT 5.3

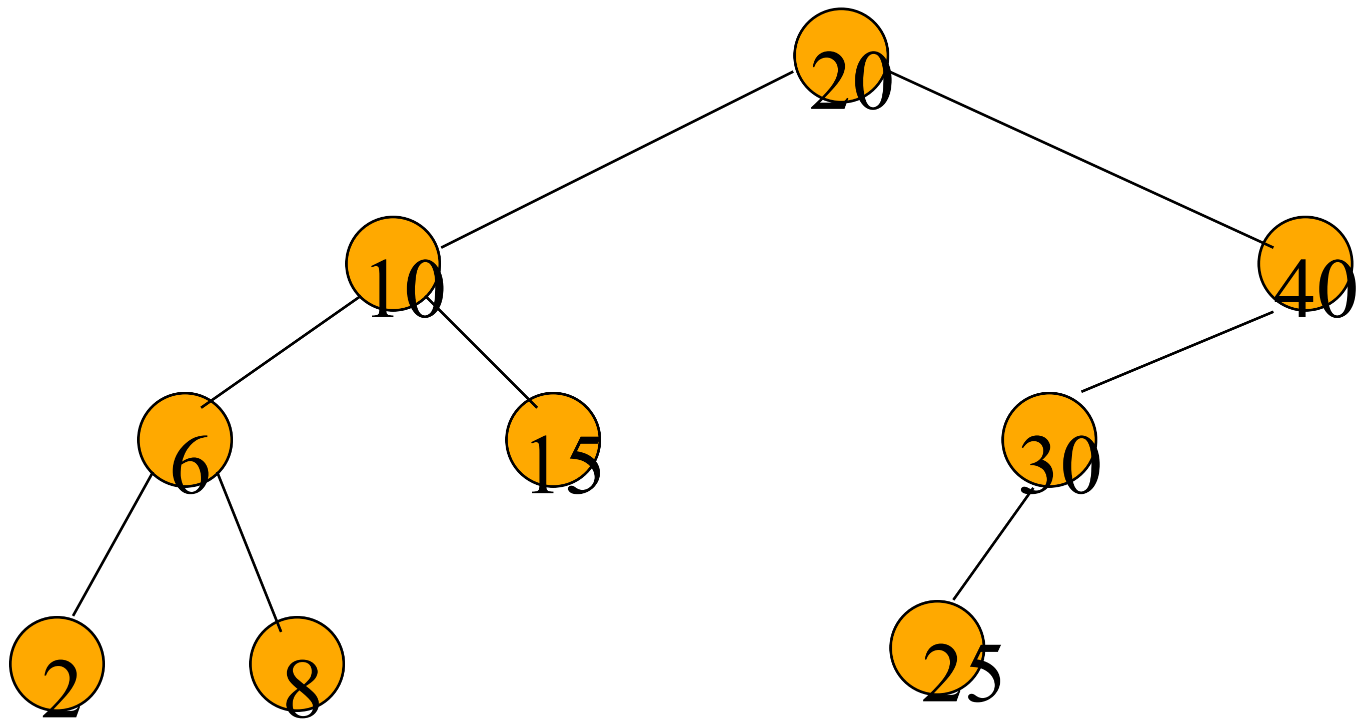
```
template <class K, class E>
class Dictionary {
public:
   virtual bool IsEmpty () const = 0;
      // return true iff the dictionary is empty
   virtual pair<K,E>* Get(const K&) const = 0;
      // return pointer to the pair with specified key;
      // return 0 if no such pair
   virtual void Insert(const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

```
template <class K, class E>
struct pair
{
    K first;
    E second;
};
```

Definition Of Binary Search Tree

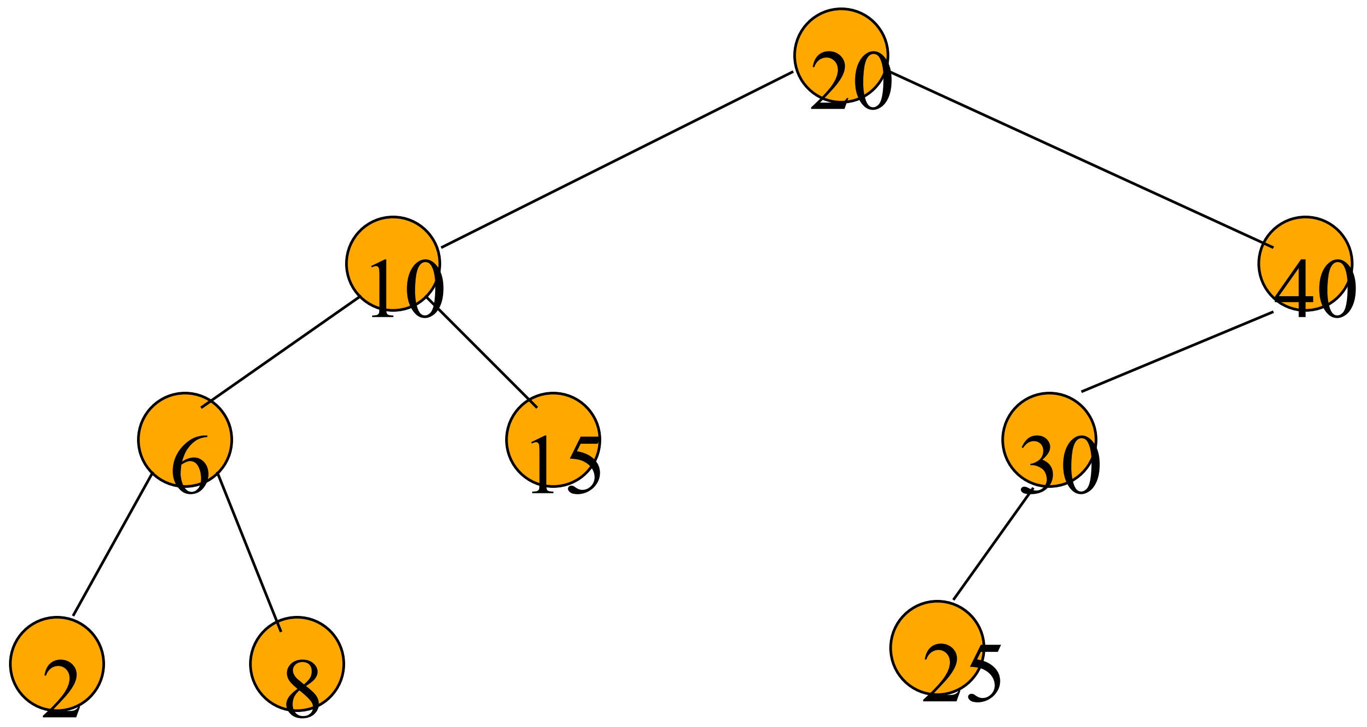
- A binary tree.
- Each node has a (key, value) pair.
- For every node x, all keys in the left subtree of x are smaller than that in x.
- For every node x, all keys in the right subtree of x are greater than that in x.

Example Binary Search Tree



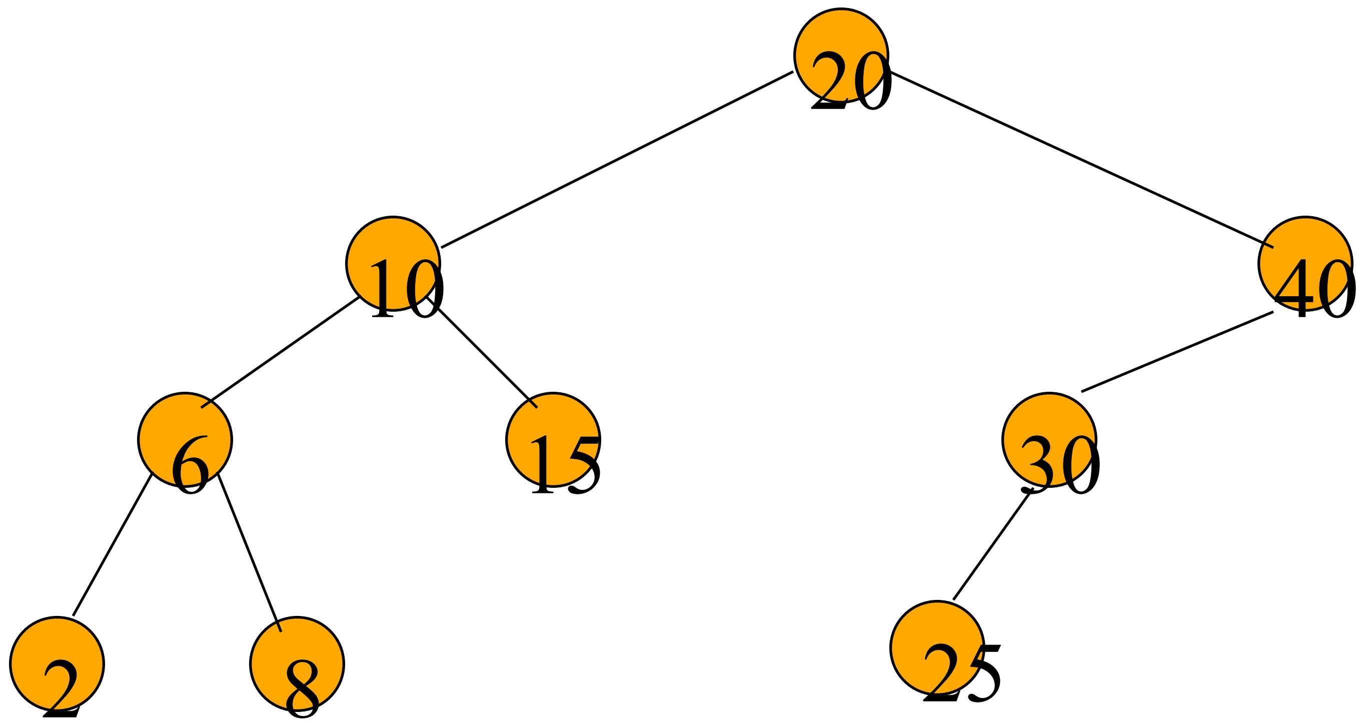
Only keys are shown.

The Operation ascend()



Do an inorder traversal. O(n) time.

The Operation get()



Complexity is O(height) = O(n), where n is number of nodes/elements.

The Operation get()

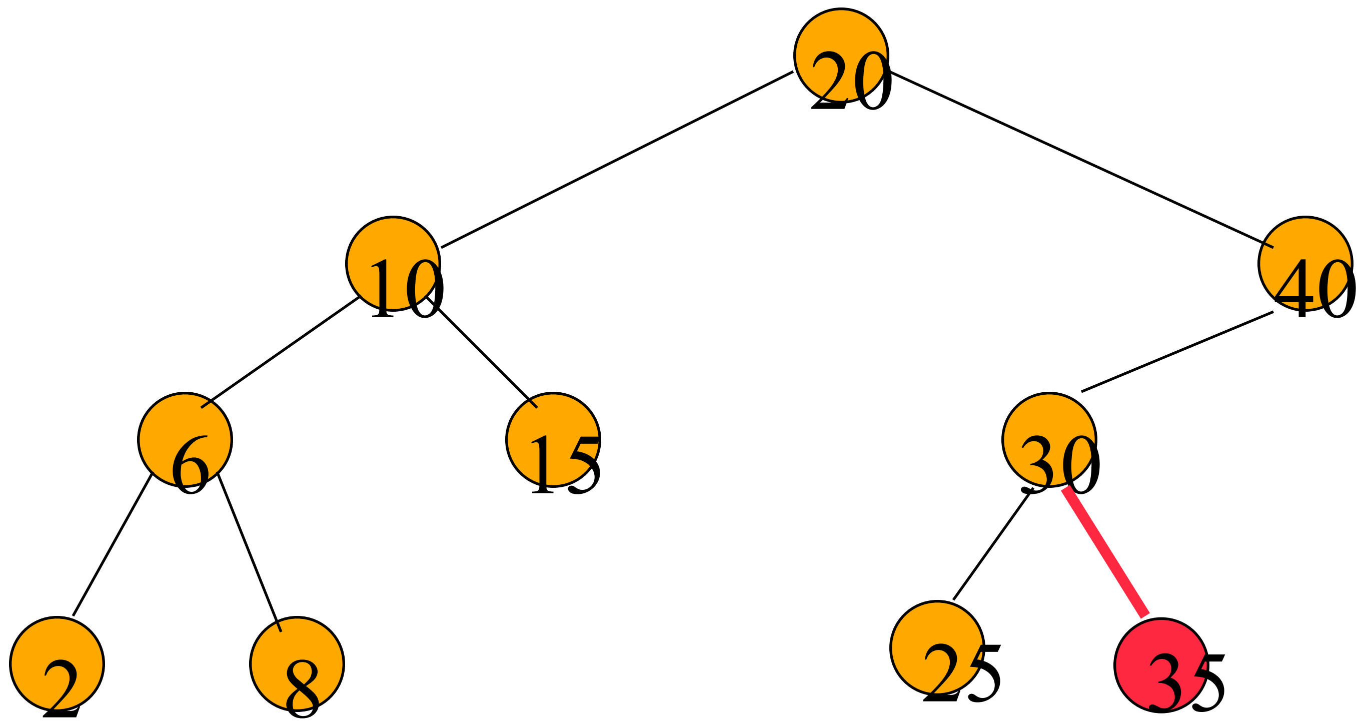
- **Search for an element with key k:**
  - **If k==the key in root, success**
  - **If x<the key in root, search the left subtree**
  - **If x>the key in root, search the right subtree**

The Operation get()

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

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

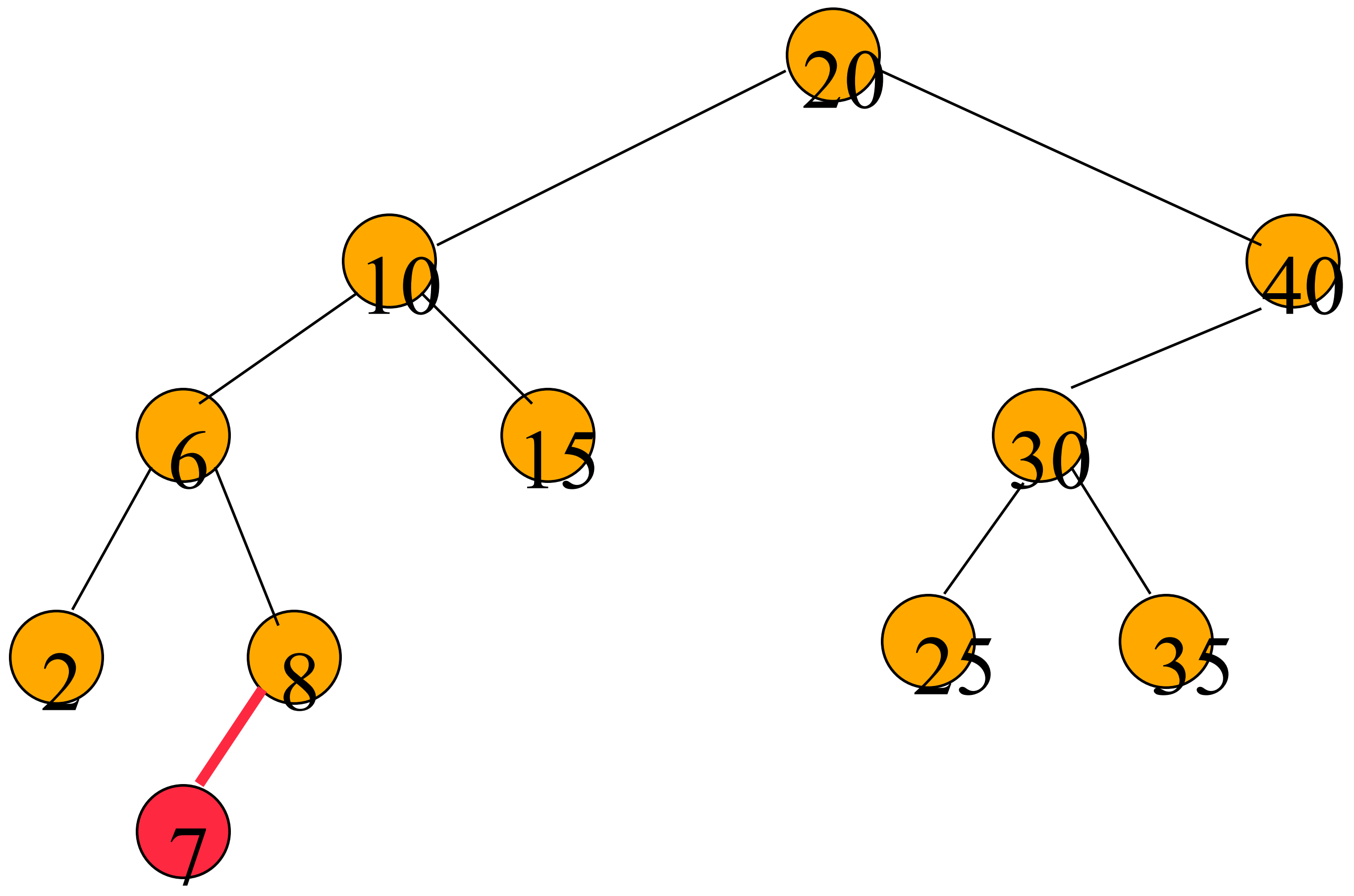
```
template <class K, class E>  // Iterative version
pair<K,E>* BST<K,E>::Get(const K& k)
{
    TreeNode<pair<K,E>>* currentNode = root;
    while (currentNode)
      if (k < currentNode→data.first)
          currentNode = currentNode→leftChild;
      else if (k > currentNode→data.first)
          currentNode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

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

The Operation put()



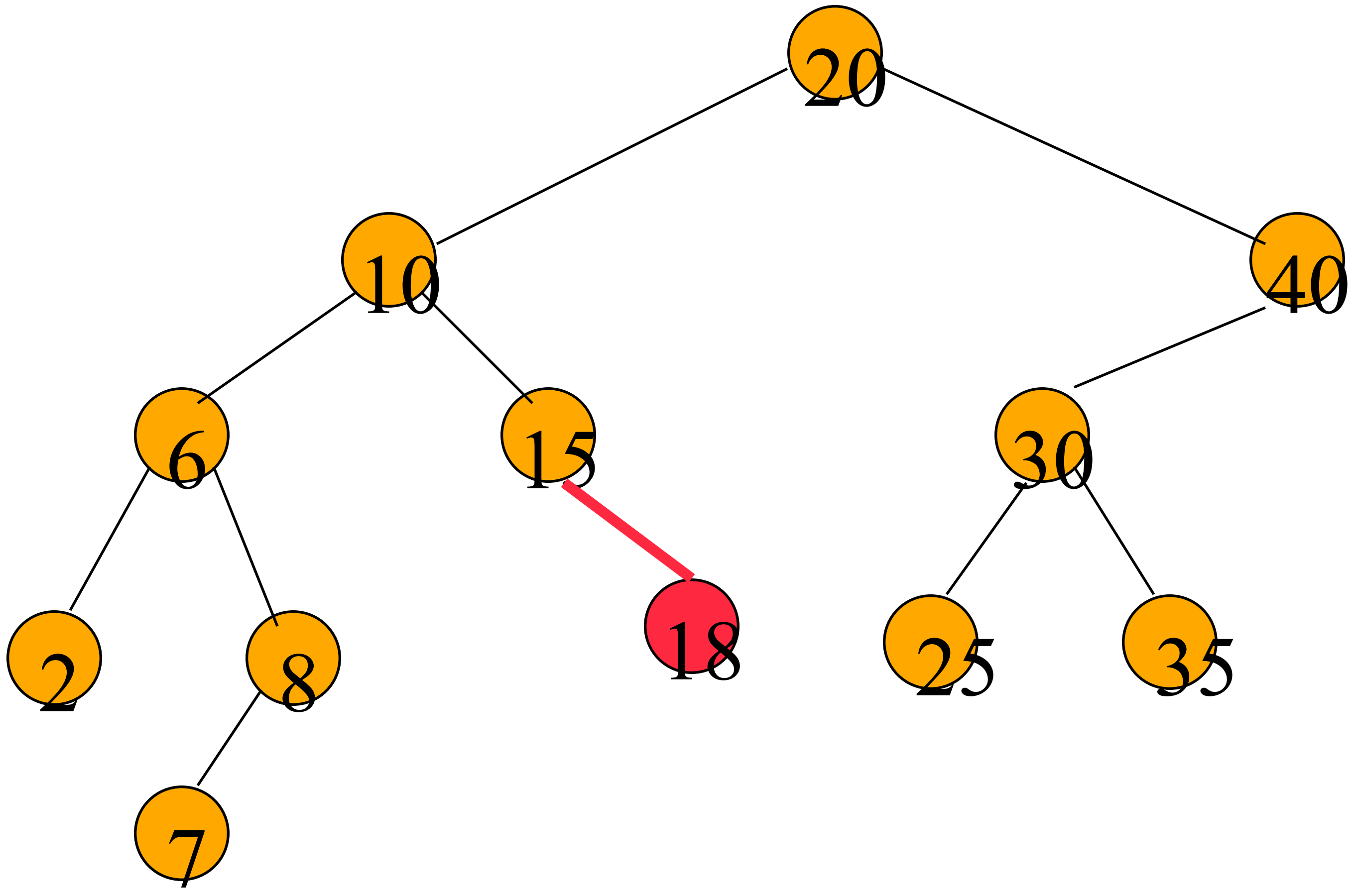
Put a pair whose key is 35.

The Operation put()



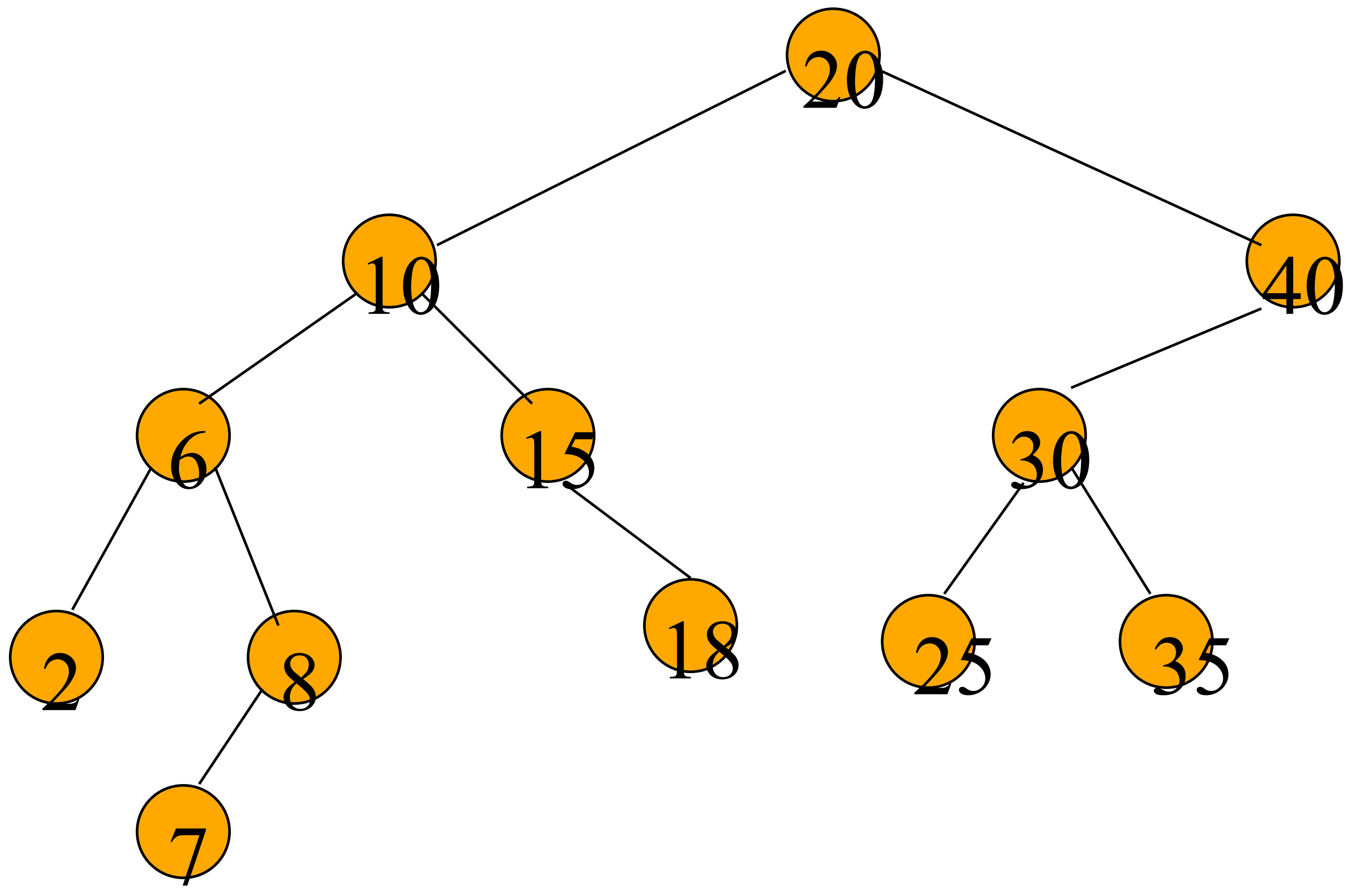
Put a pair whose key is 7.

The Operation put()



Put a pair whose key is 18.

The Operation put()



Complexity of put() is O(height).

When the dictionary already contains a pair with key k

Simply update the element associated with this key to e

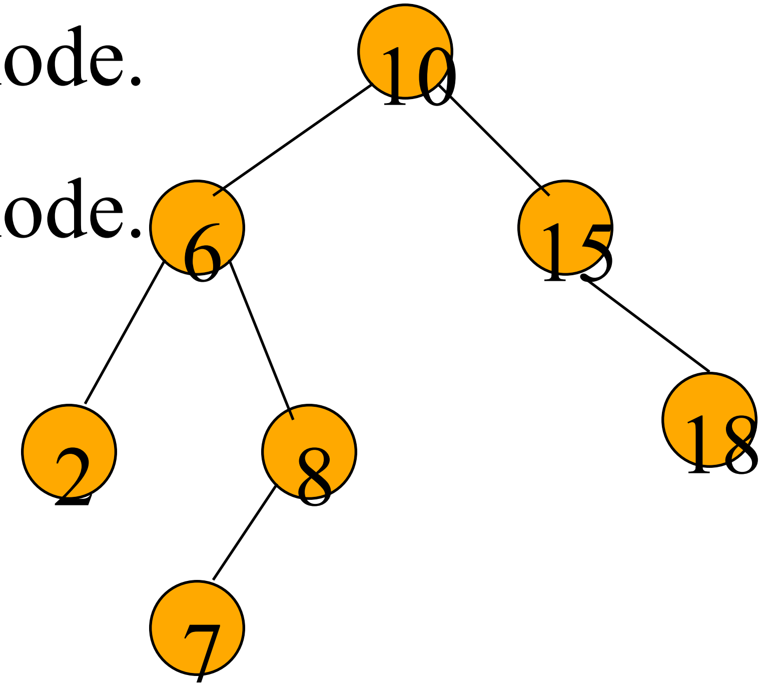
```
template <class K, class E>
void BST<K,E>::Insert(const pair<K,E>& thePair) {
    TreeNode<pair<K,E>> *p=root, *pp=0;
    while (p) {
      pp=p;
      if (thePair.first < p→data.first) p=p→leftChild;
      else if (thePair.first > p→data.first) p=p→rightChild;
      else // duplicate, update associated element
           {p→data.second=thePair.second;return;}
    }
    p=new TreeNode<pair<K,E>>(thePair,0,0);
    if (root) // tree not empty
       if (thePair.first < pp→data.first) pp→leftChild=p;
         else pp→rightChild=p;
    else root=p;
}
```

O(h)

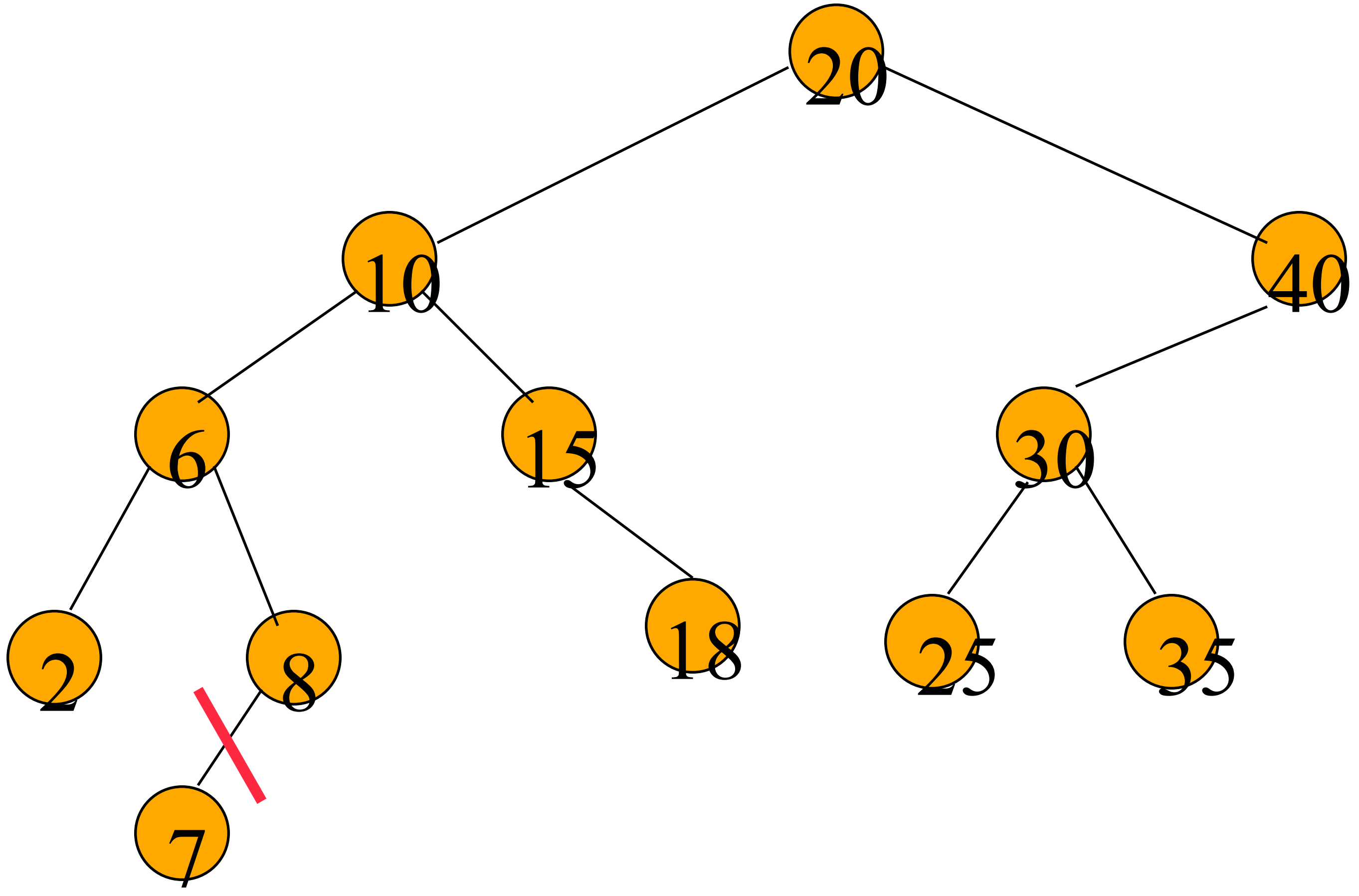
The Operation remove()

Three cases:

- Element is in a leaf.
- Element is in a degree 1 node.
- Element is in a degree 2 node.

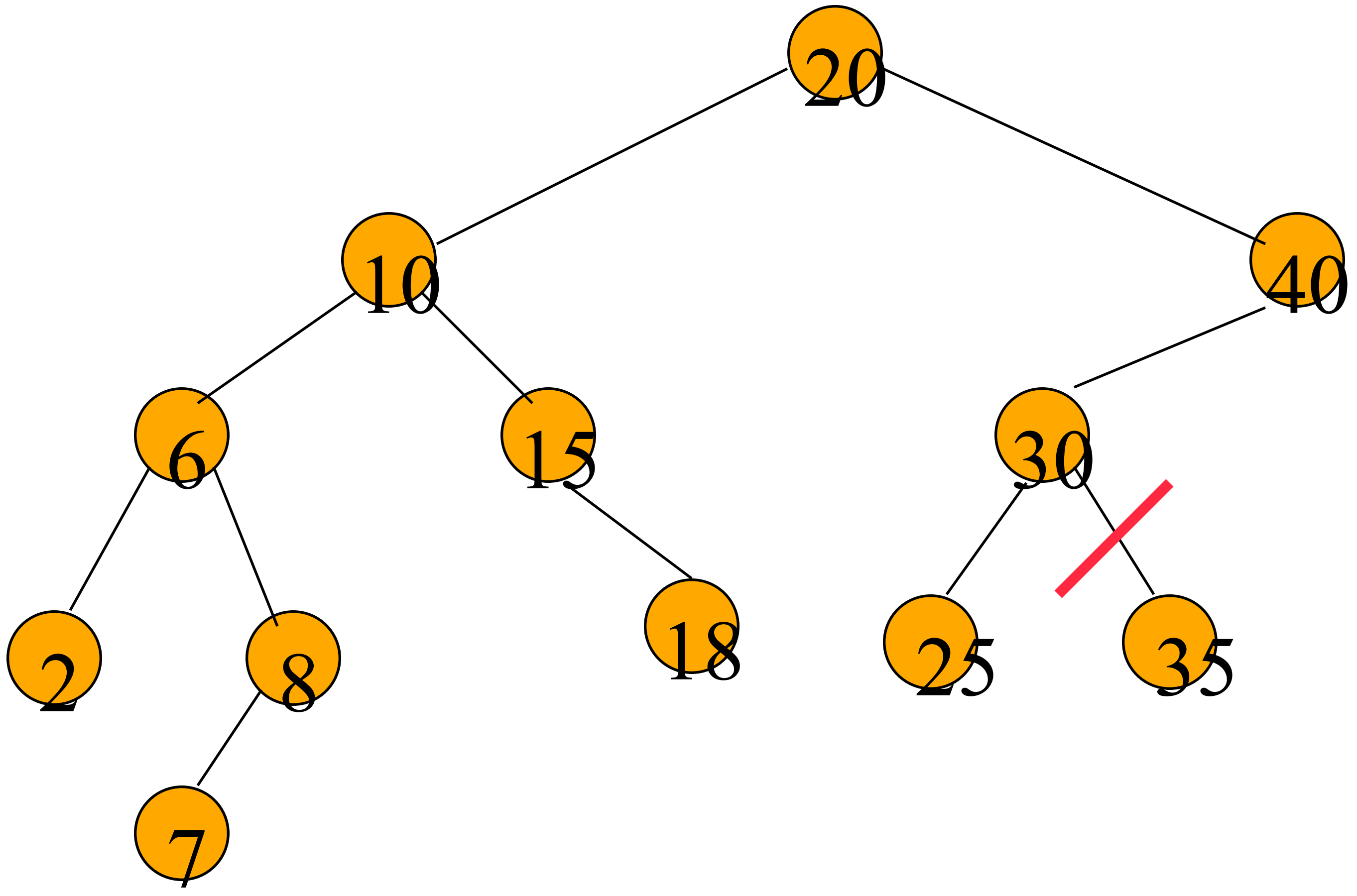


Remove From A Leaf



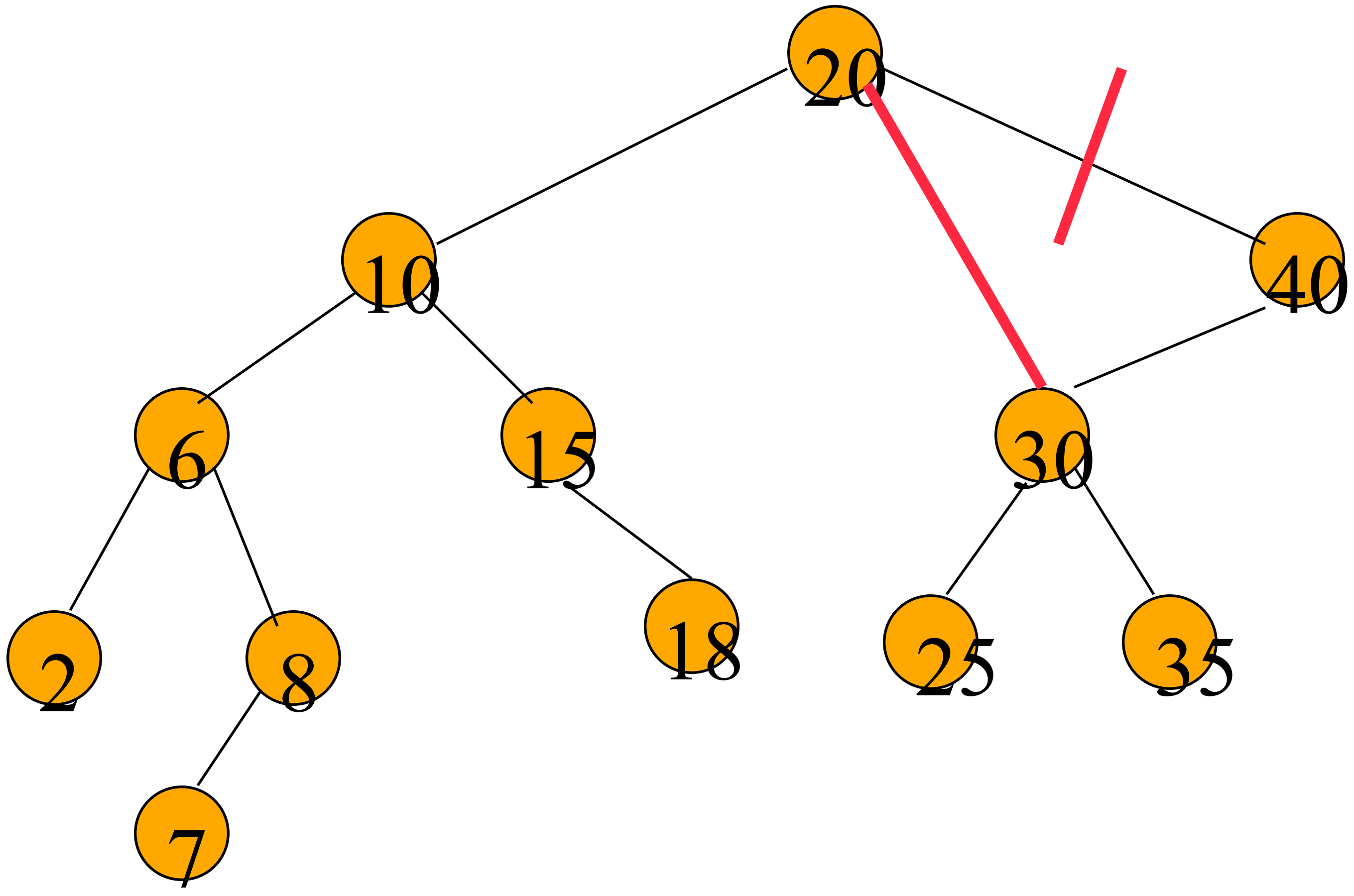
Remove a leaf element. key = 7

Remove From A Leaf (contd.)



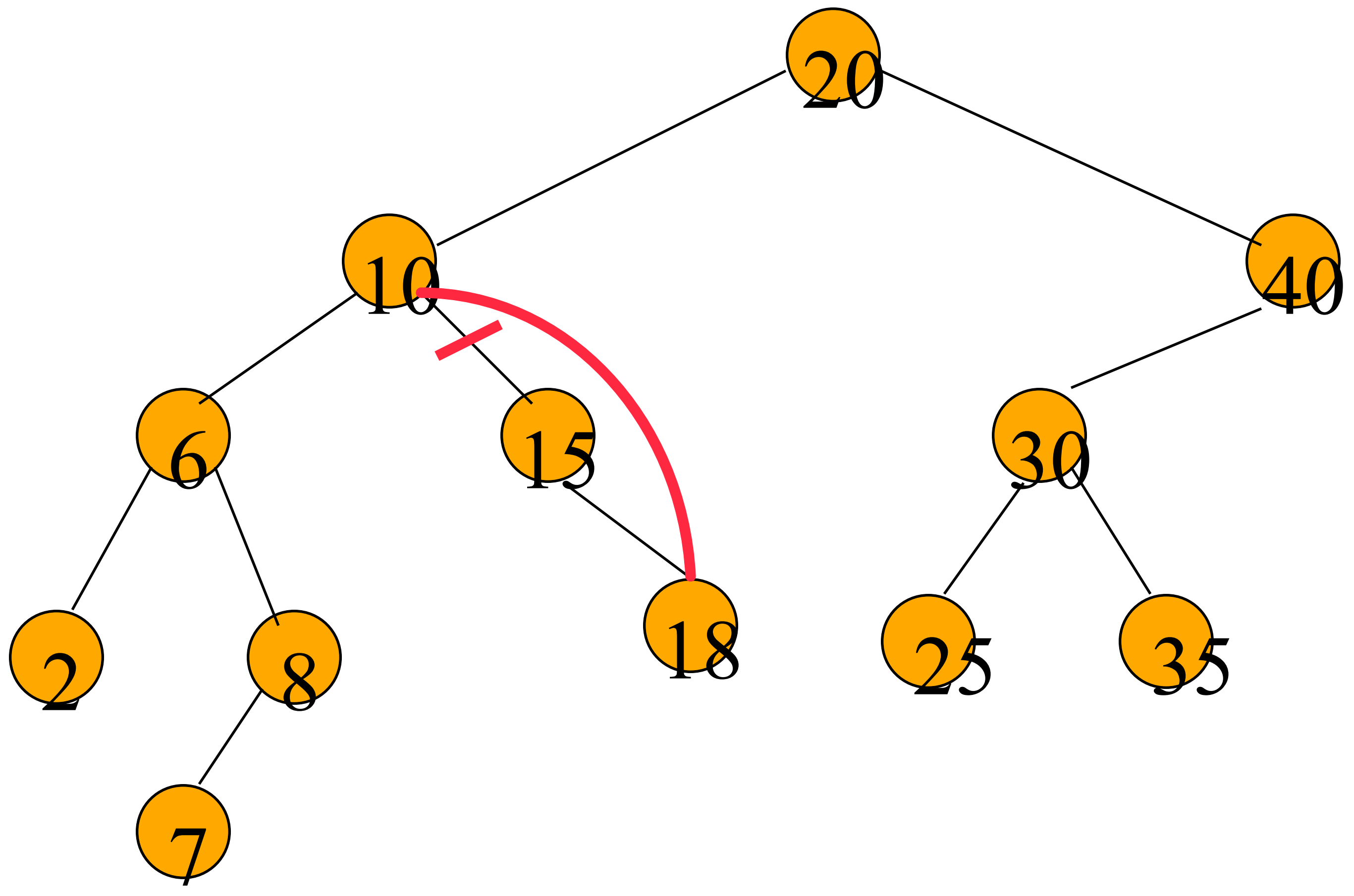
Remove a leaf element. key = 35

Remove From A Degree 1 Node



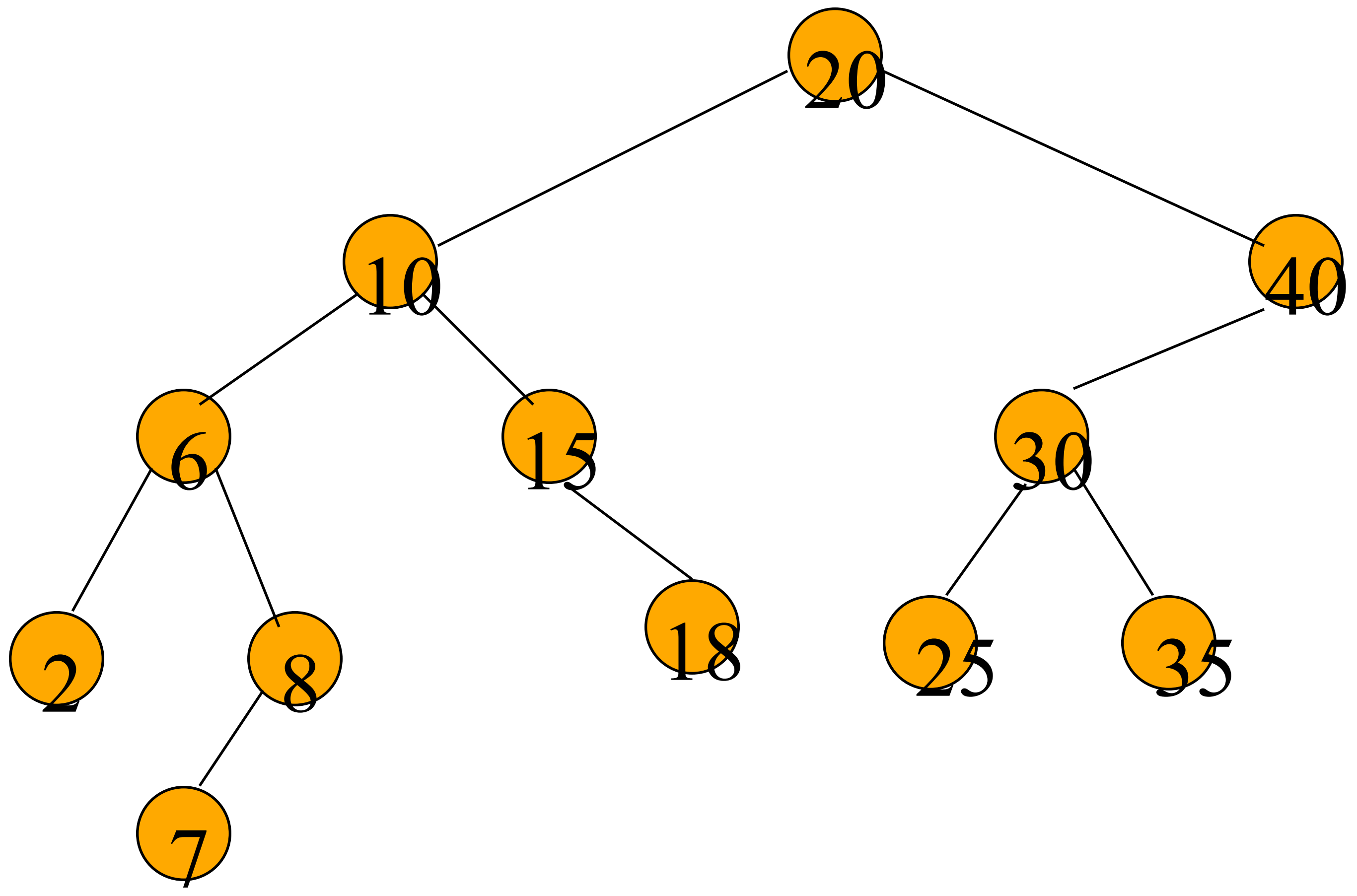
Remove from a degree 1 node. key = 40

Remove From A Degree 1 Node (contd.)



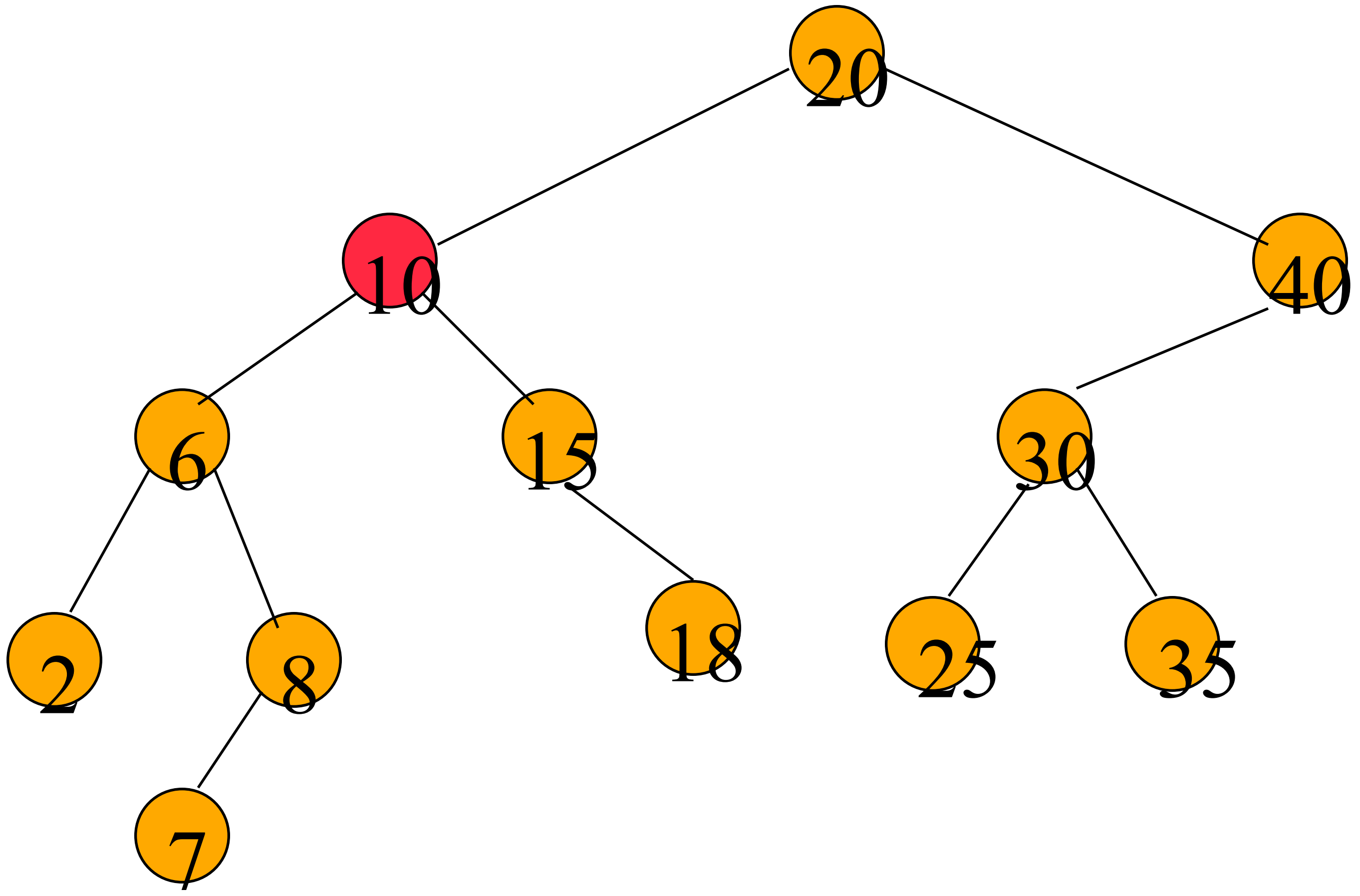
Remove from a degree 1 node. key = 15

Remove From A Degree 2 Node



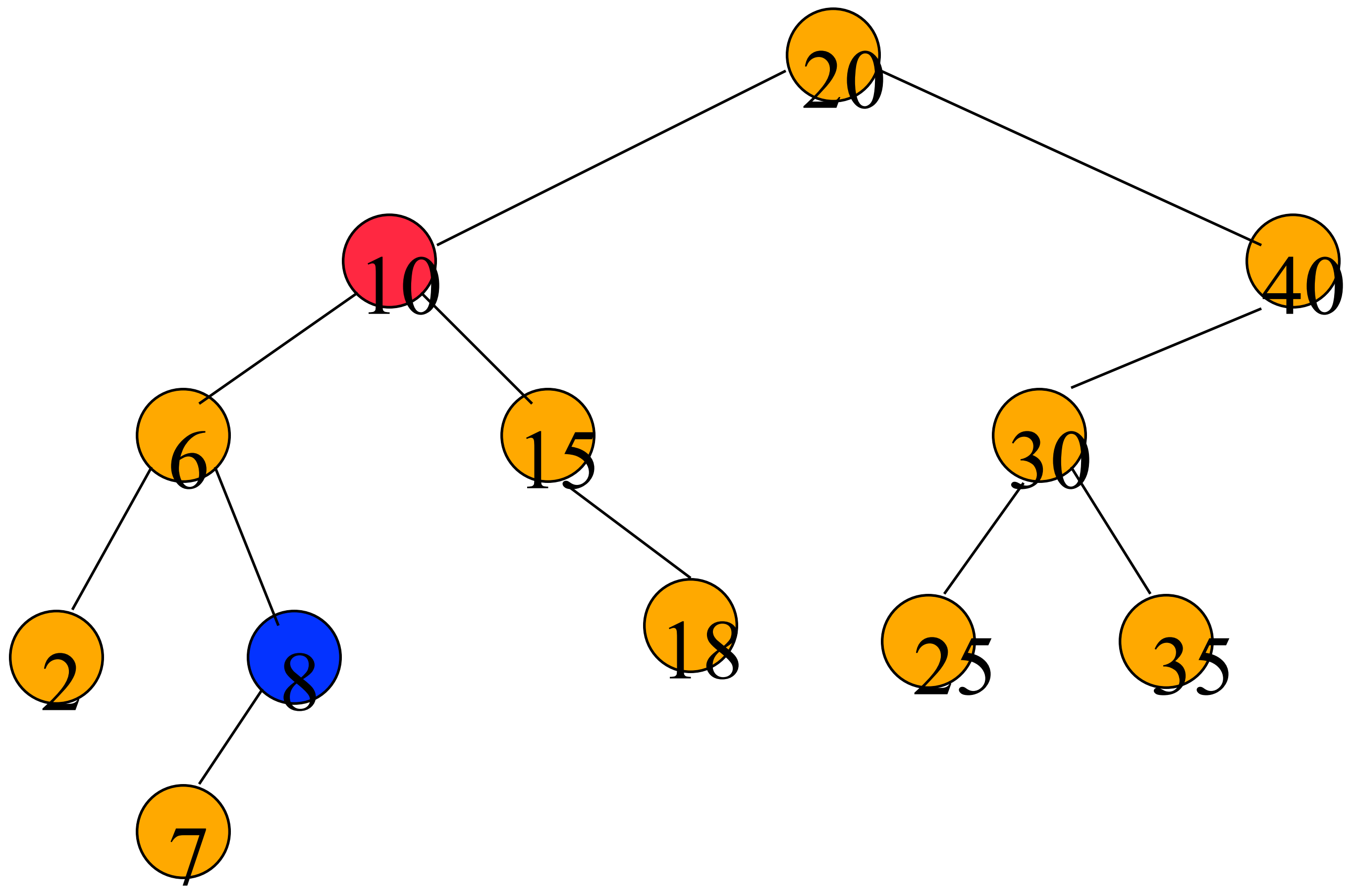
Remove from a degree 2 node. key = 10

Remove From A Degree 2 Node



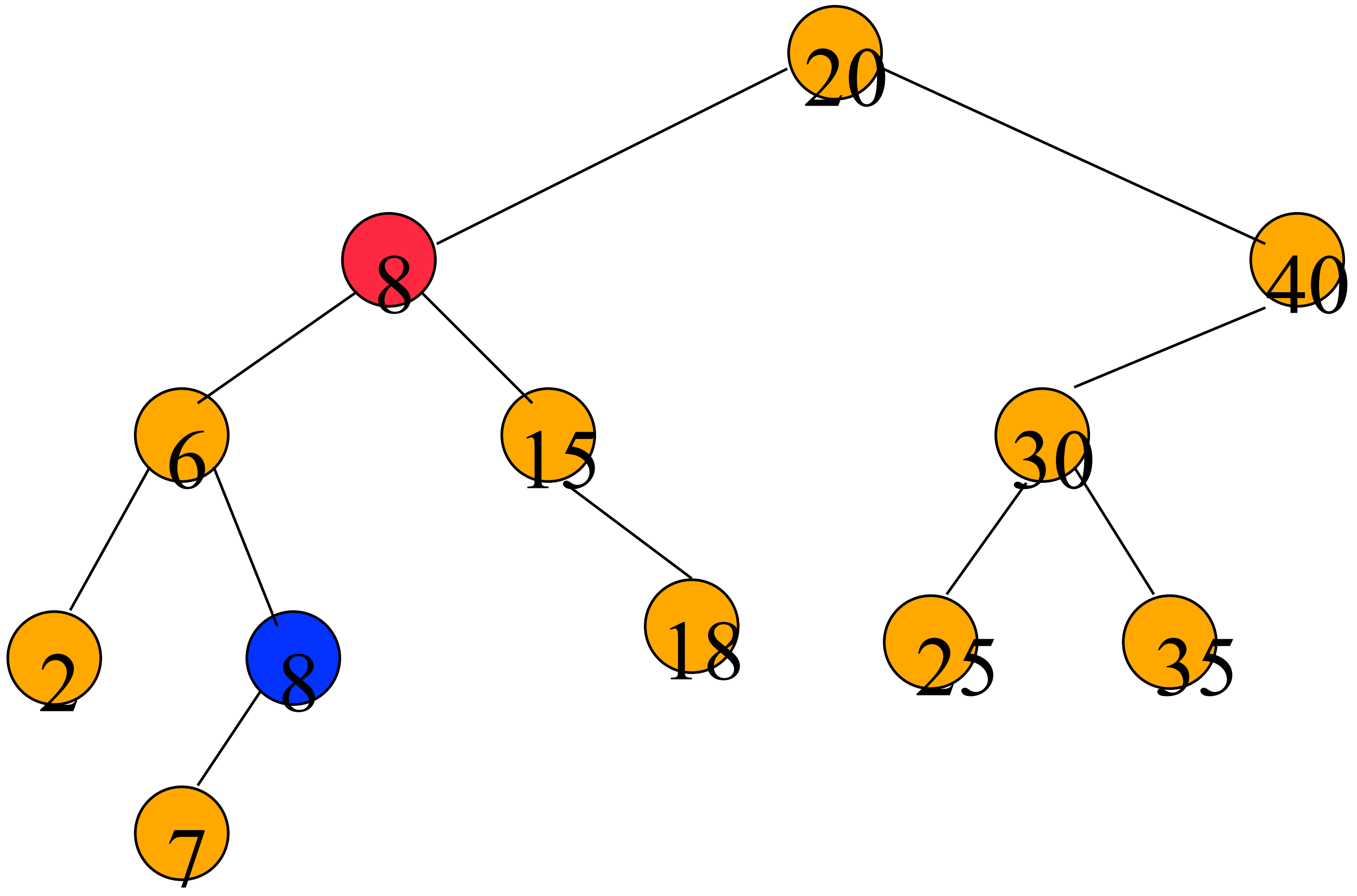
Replace with largest key in left subtree (or smallest in right subtree).

Remove From A Degree 2 Node



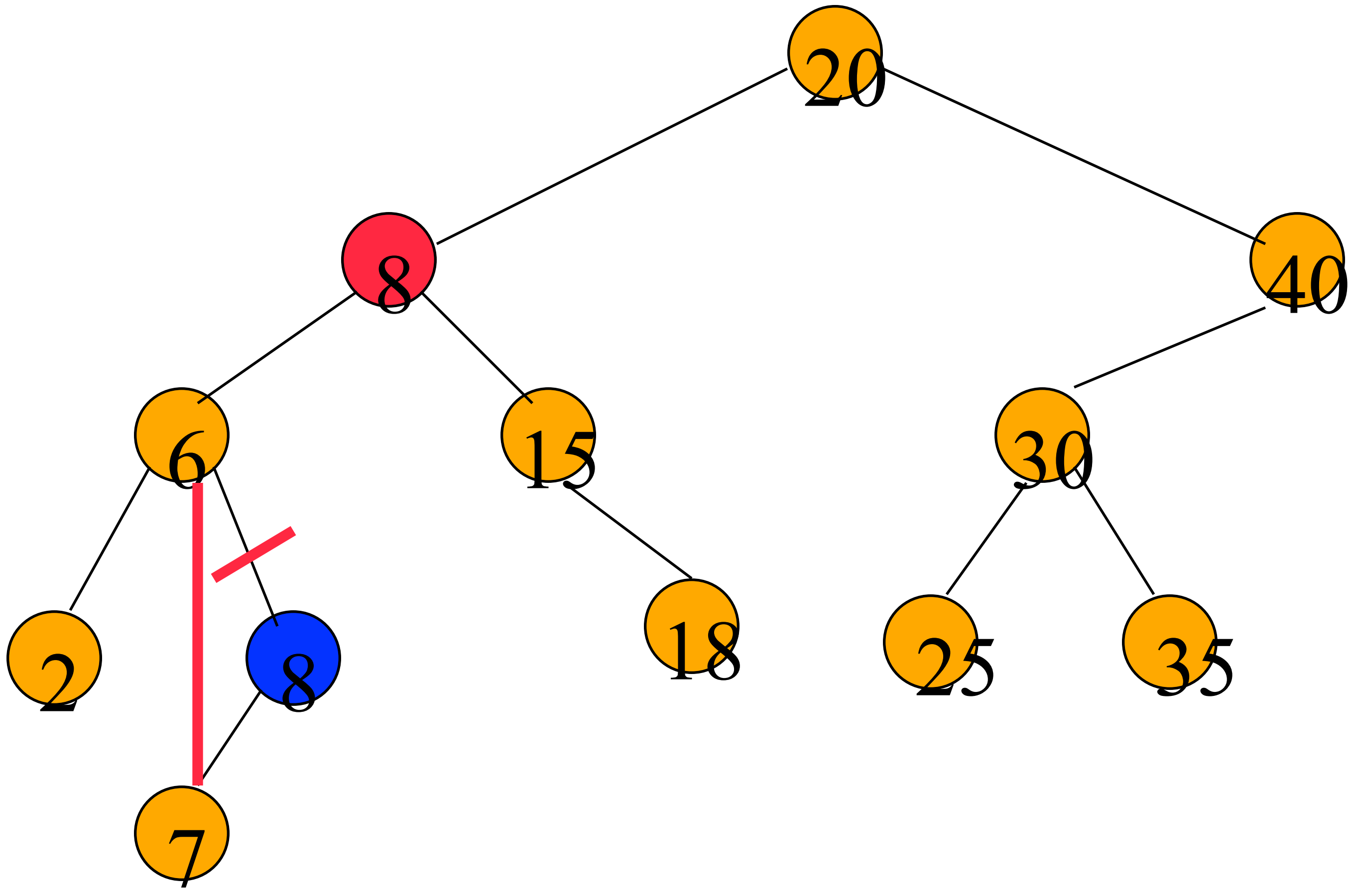
Replace with largest key in left subtree (or smallest in right subtree).

Remove From A Degree 2 Node



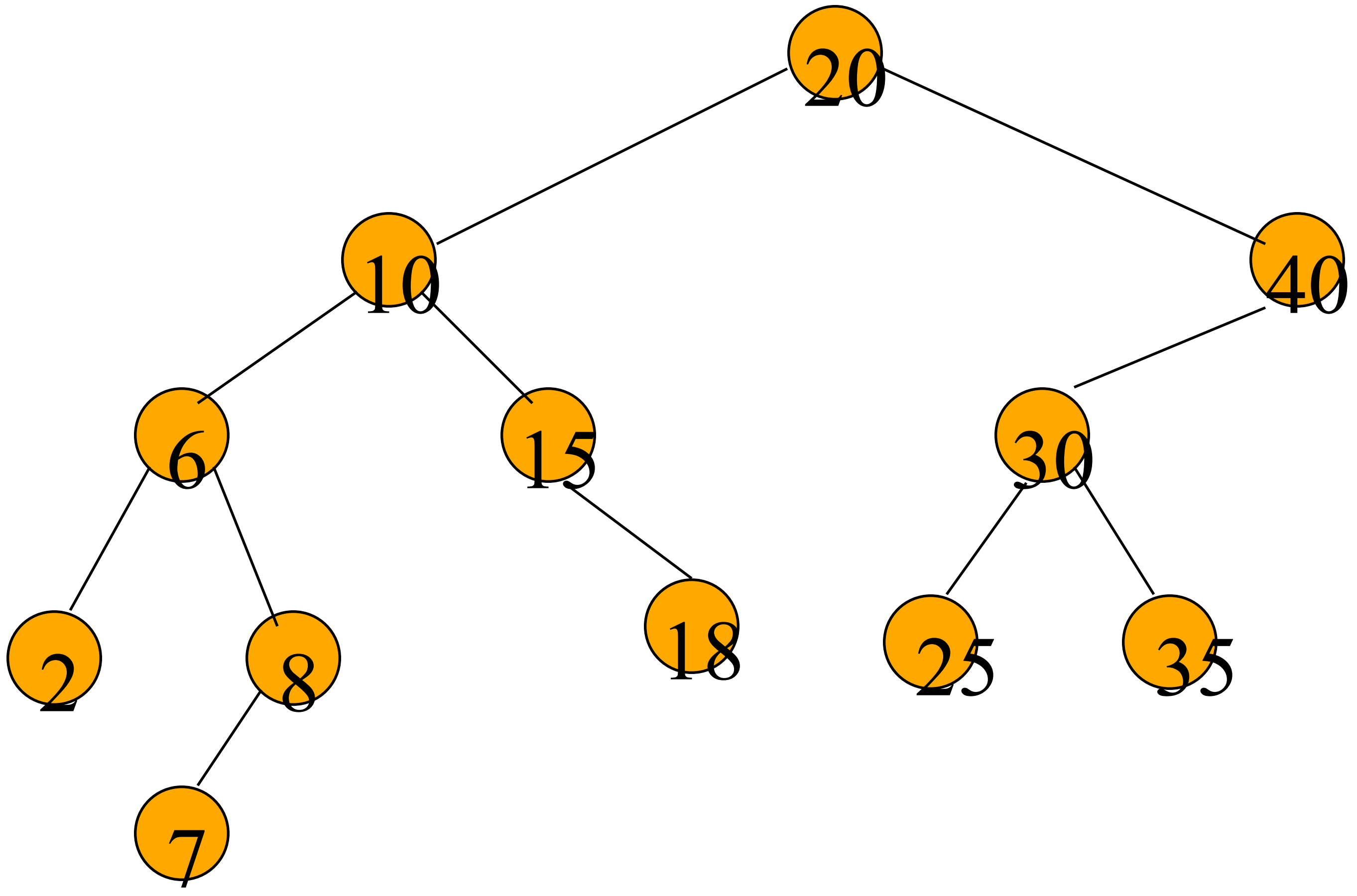
Replace with largest key in left subtree (or smallest in right subtree).

Remove From A Degree 2 Node



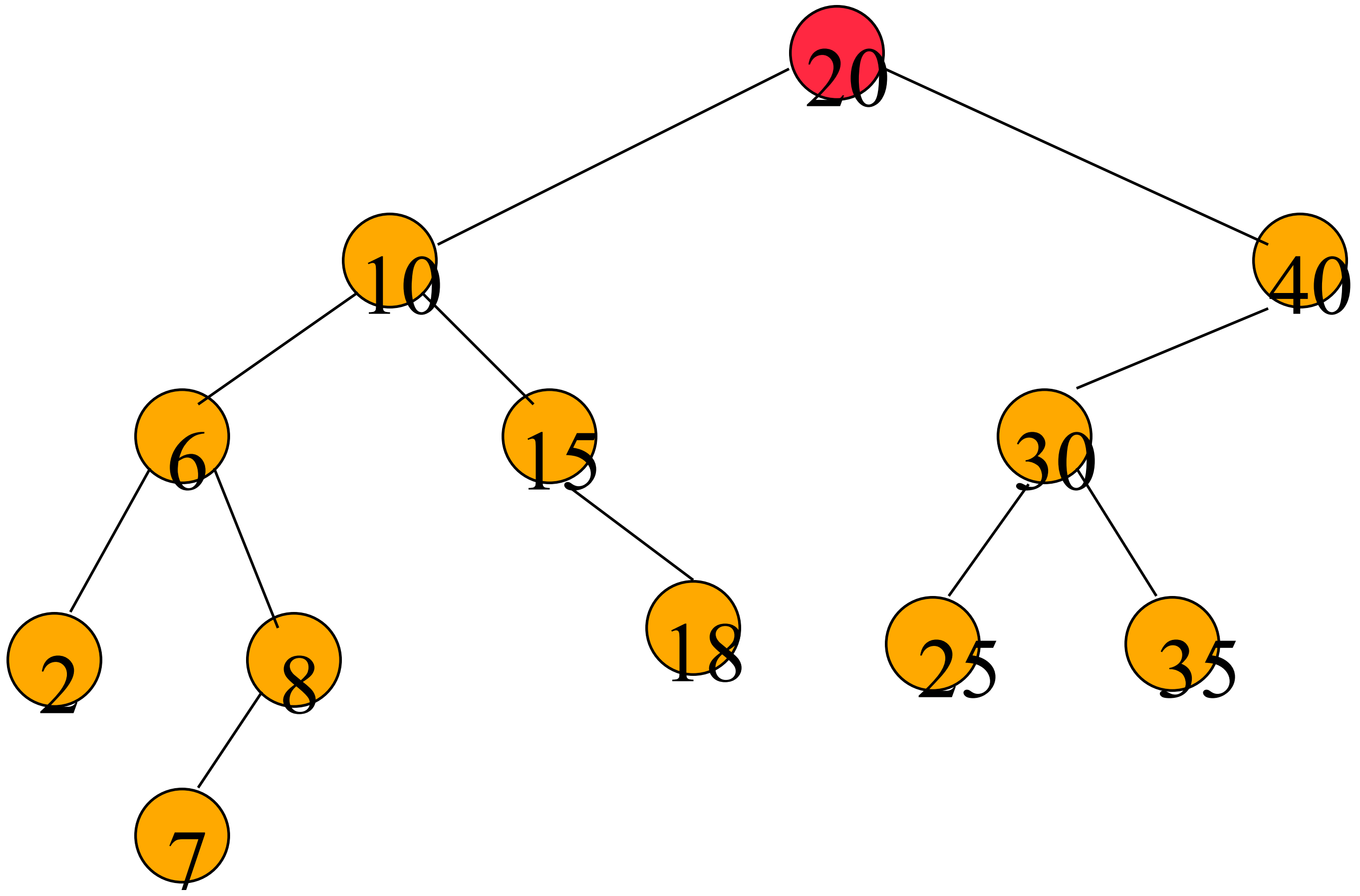
Largest key must be in a leaf or degree 1 node.

Another Remove From A Degree 2 Node



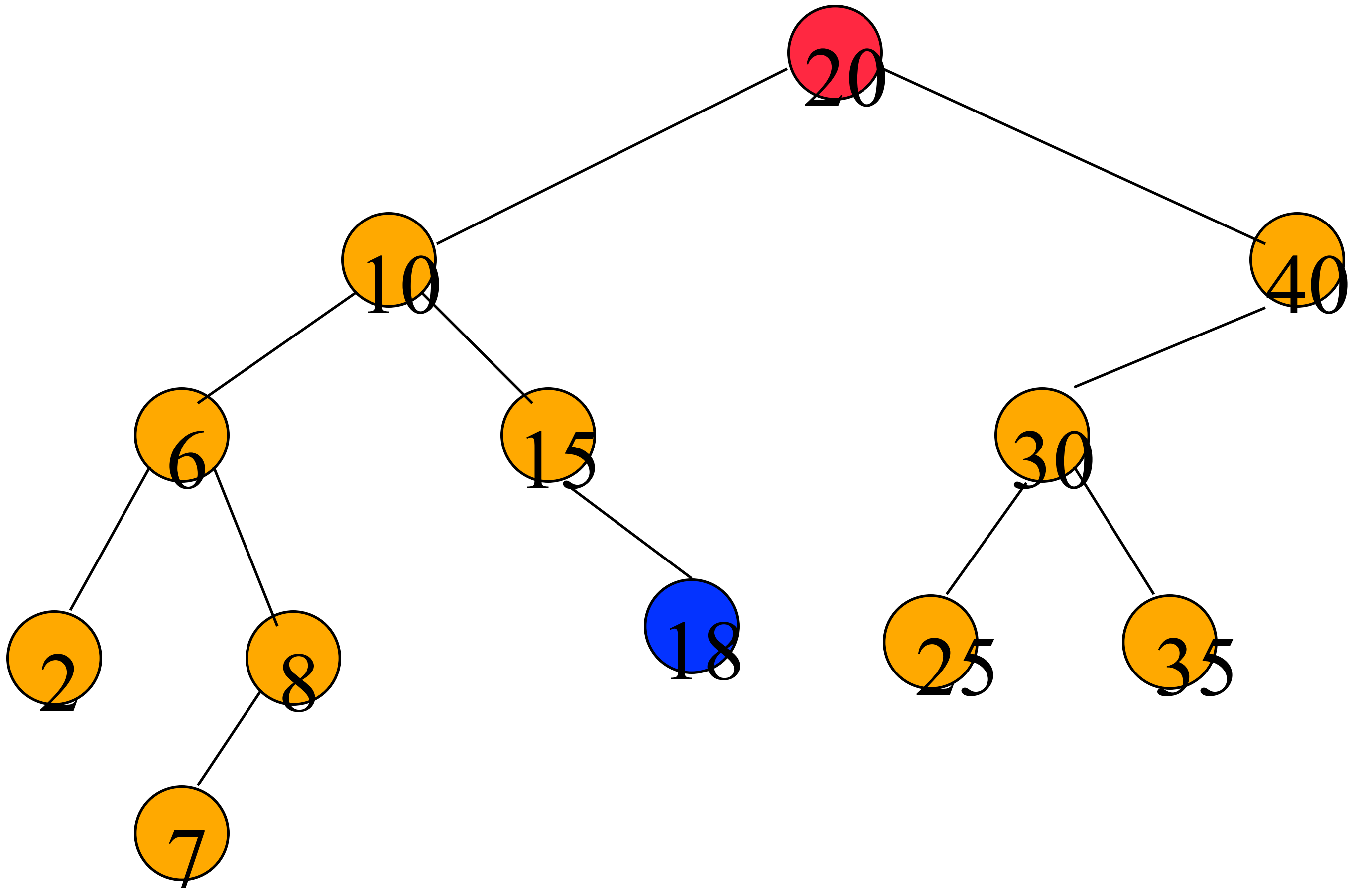
Remove from a degree 2 node. key = 20

Remove From A Degree 2 Node



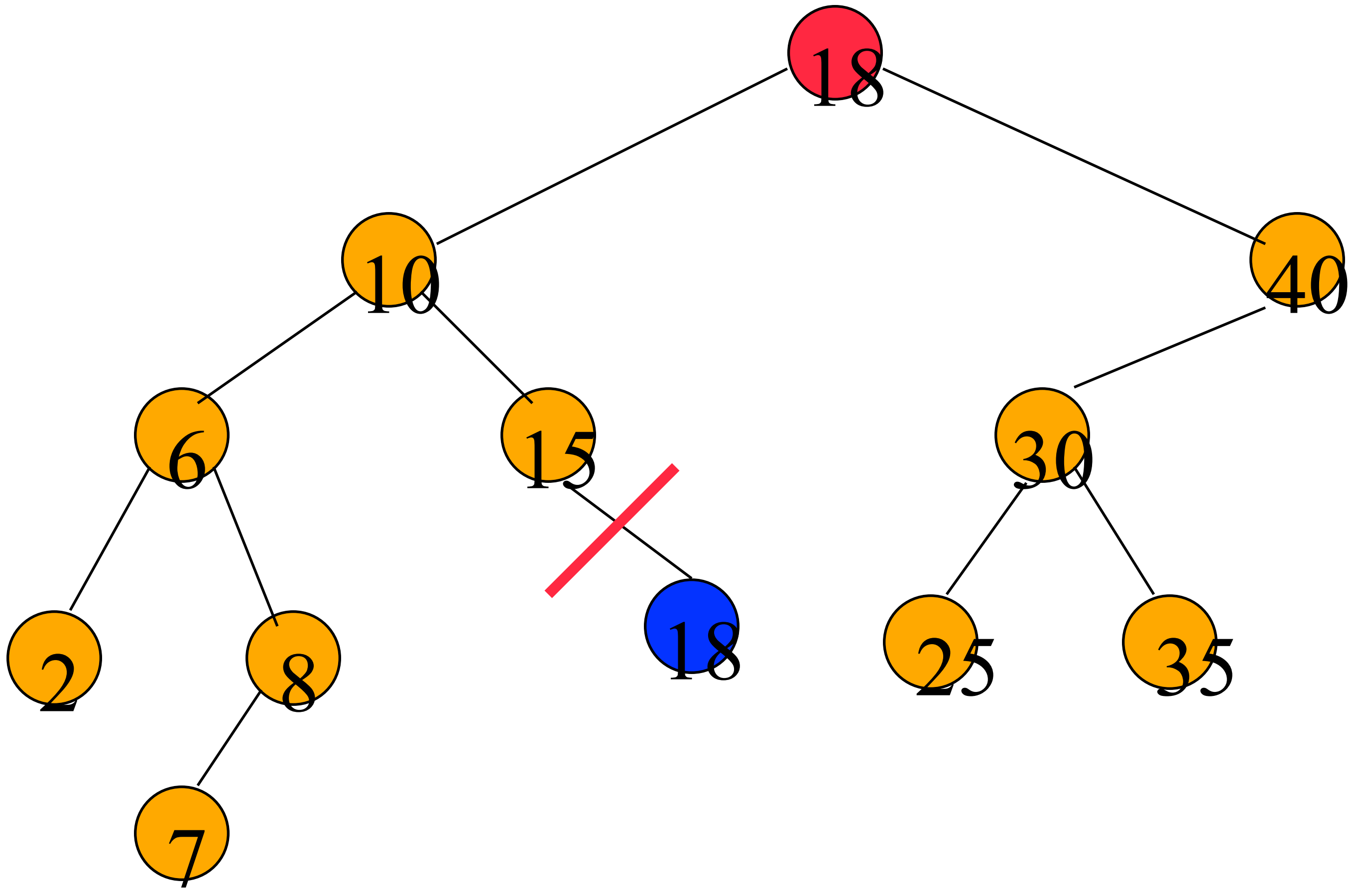
Replace with largest in left subtree.

Remove From A Degree 2 Node



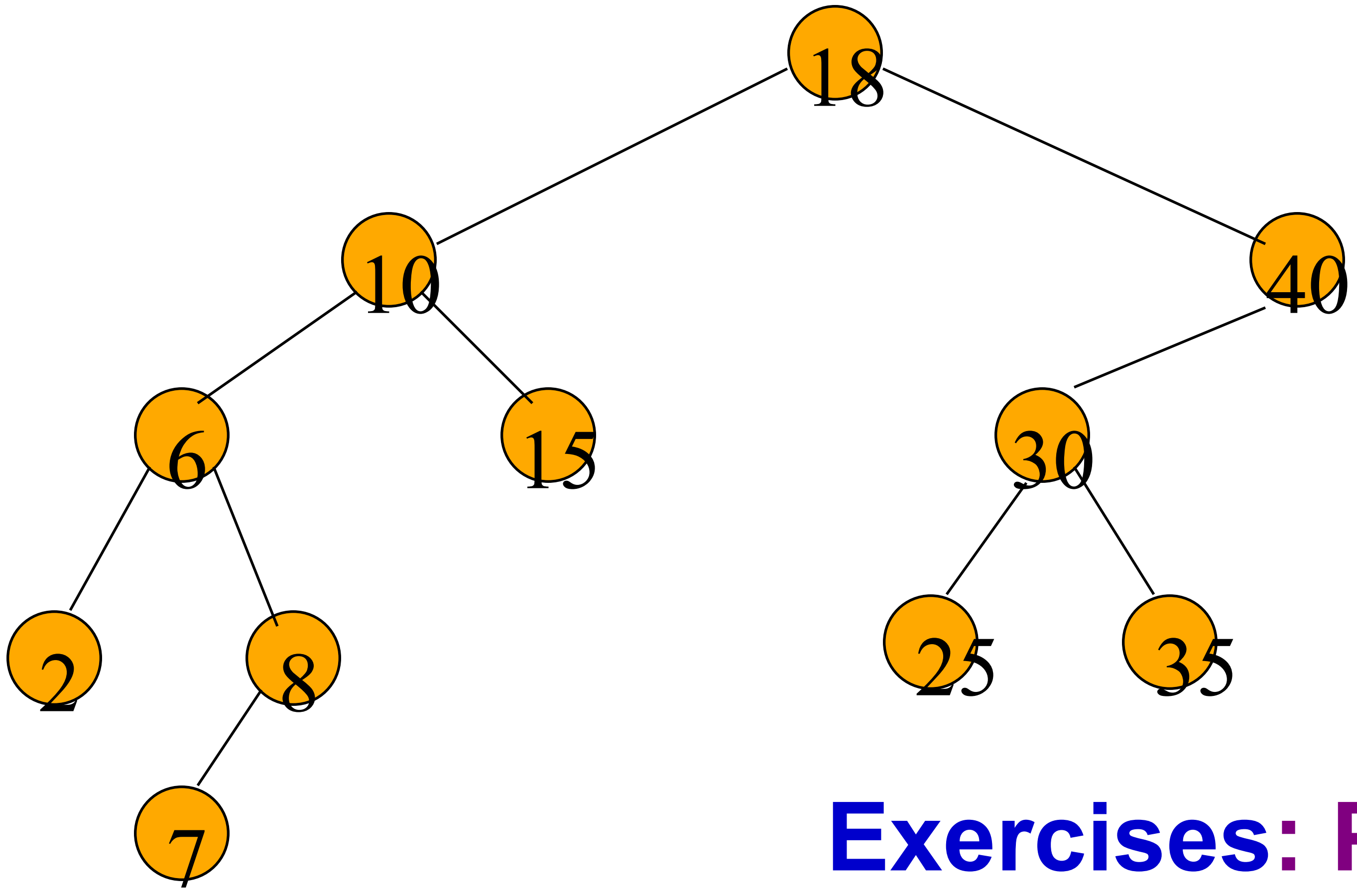
Replace with largest in left subtree.

Remove From A Degree 2 Node



Replace with largest in left subtree.

Remove From A Degree 2 Node



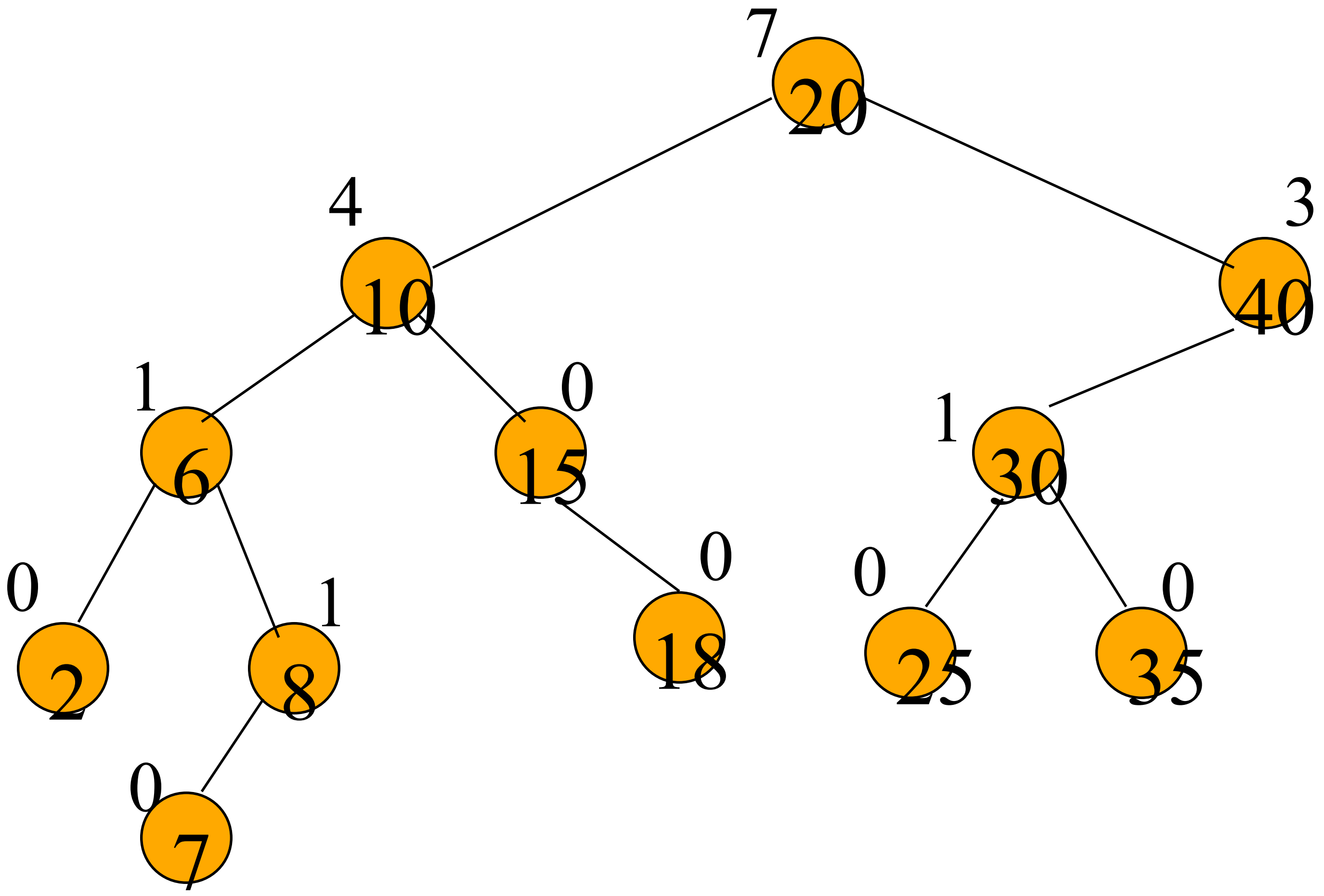
Exercises: P296-1,2

Complexity is O(height).

Indexed Binary Search Tree

- Binary search tree.
- Each node has an additional field.
  - leftSize = number of nodes in its left subtree

Example Indexed Binary Search Tree



leftSize values are out of the circle

leftSize And Rank

Rank of an element is its position in inorder (inorder = ascending key order).

[2,6,7,8,10,15,18,20,25,30,35,40]

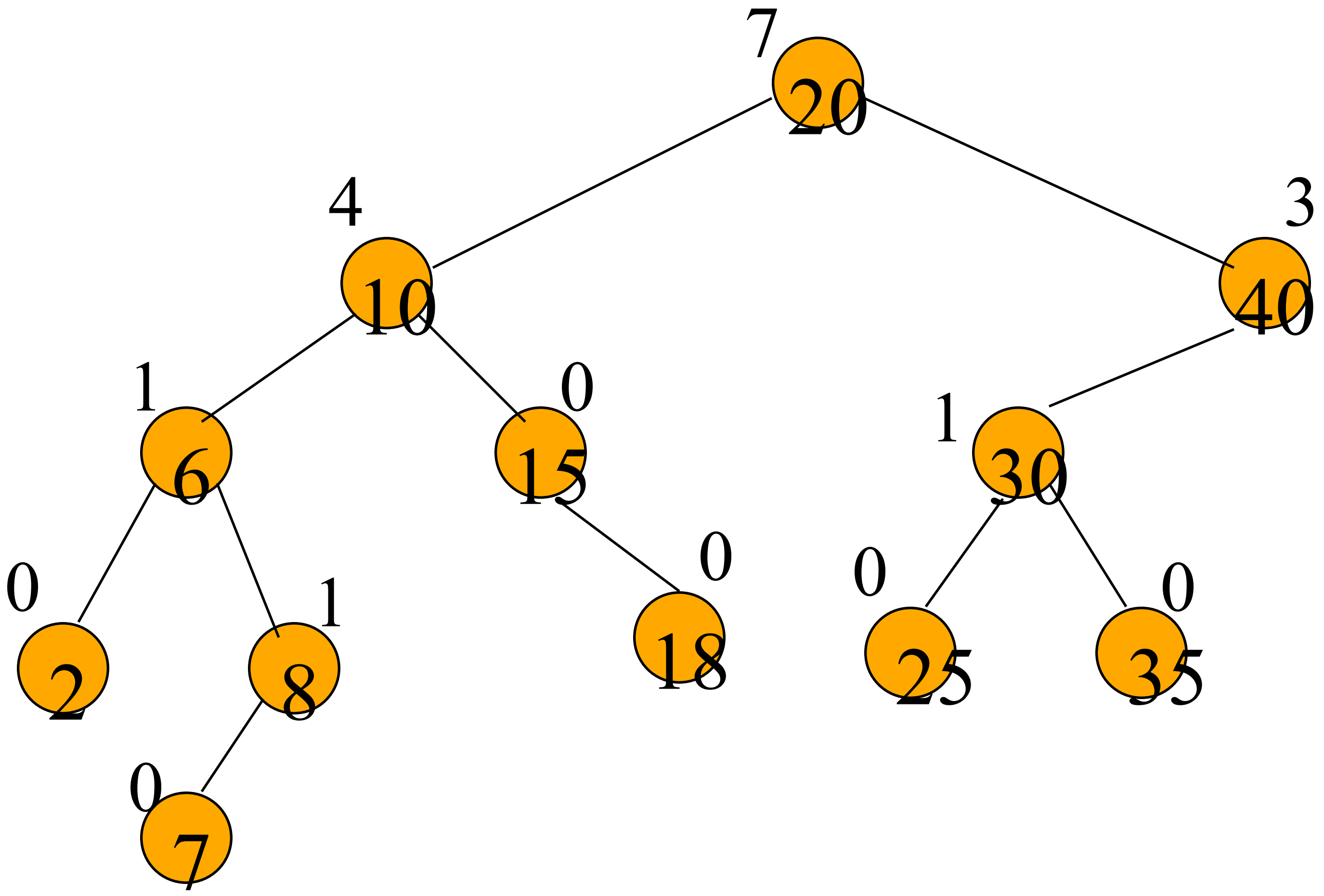
rank(2) = 0

rank(15) = 5

rank(20) = 7

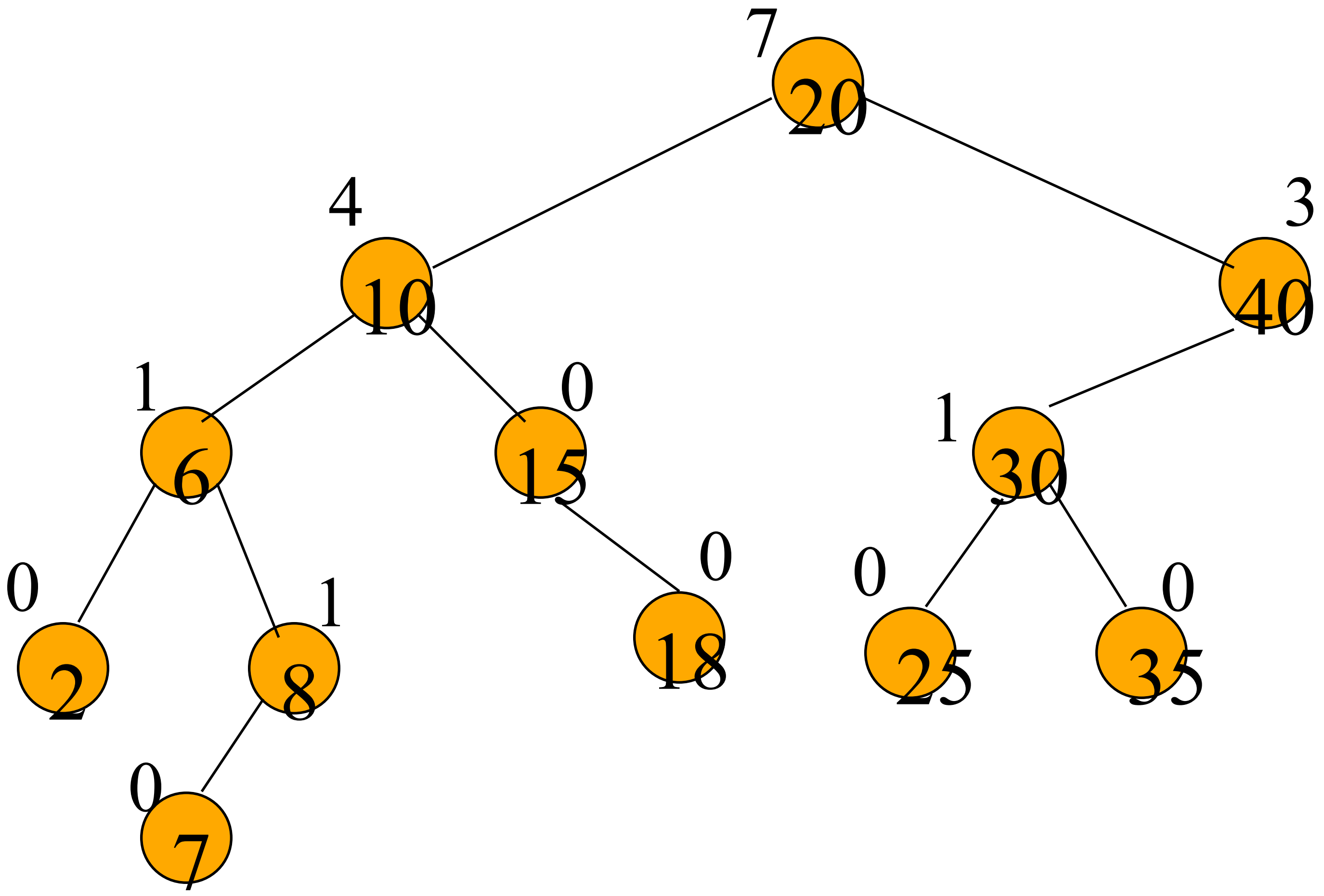
leftSize(x) = rank(x) with respect to elements in subtree rooted at x

leftSize And Rank



sorted list = [2,6,7,8,10,15,18,20,25,30,35,40]

get(index) And remove(index)



sorted list = [2,6,7,8,10,15,18,20,25,30,35,40]

get(index) And remove(index)

- if index = x.leftSize desired element is x.element
- if index < x.leftSize desired element is index'th element in left subtree of x
- if index > x.leftSize desired element is (index - x.leftSize-1)'th element in right subtree of x

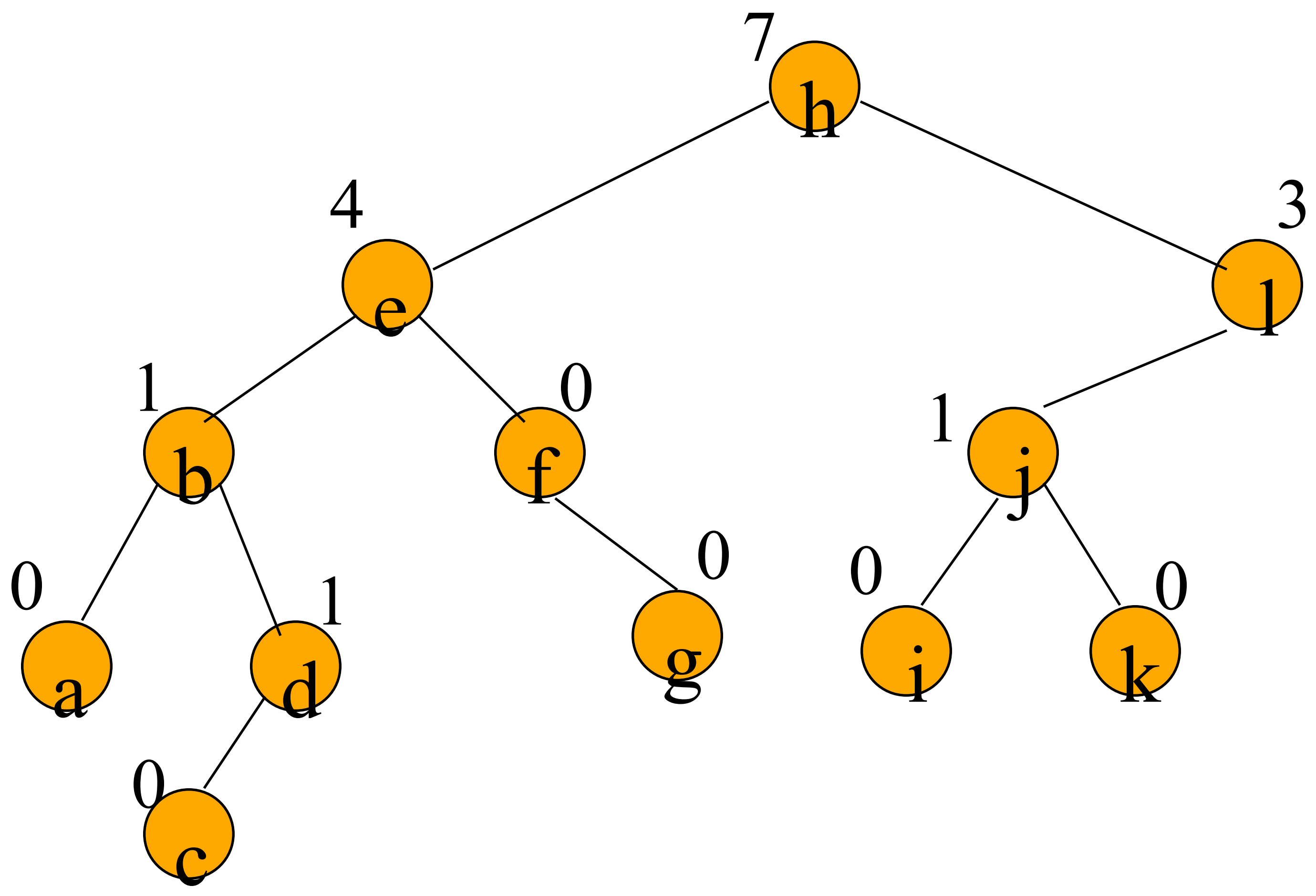
Applications
(Complexities Are For Balanced Trees)

Best-fit bin packing in O(n log n) time.

Representing a linear list so that get(index), add(index, element), and remove(index) run in O(log(list size)) time (uses an indexed binary tree, not indexed binary search tree).

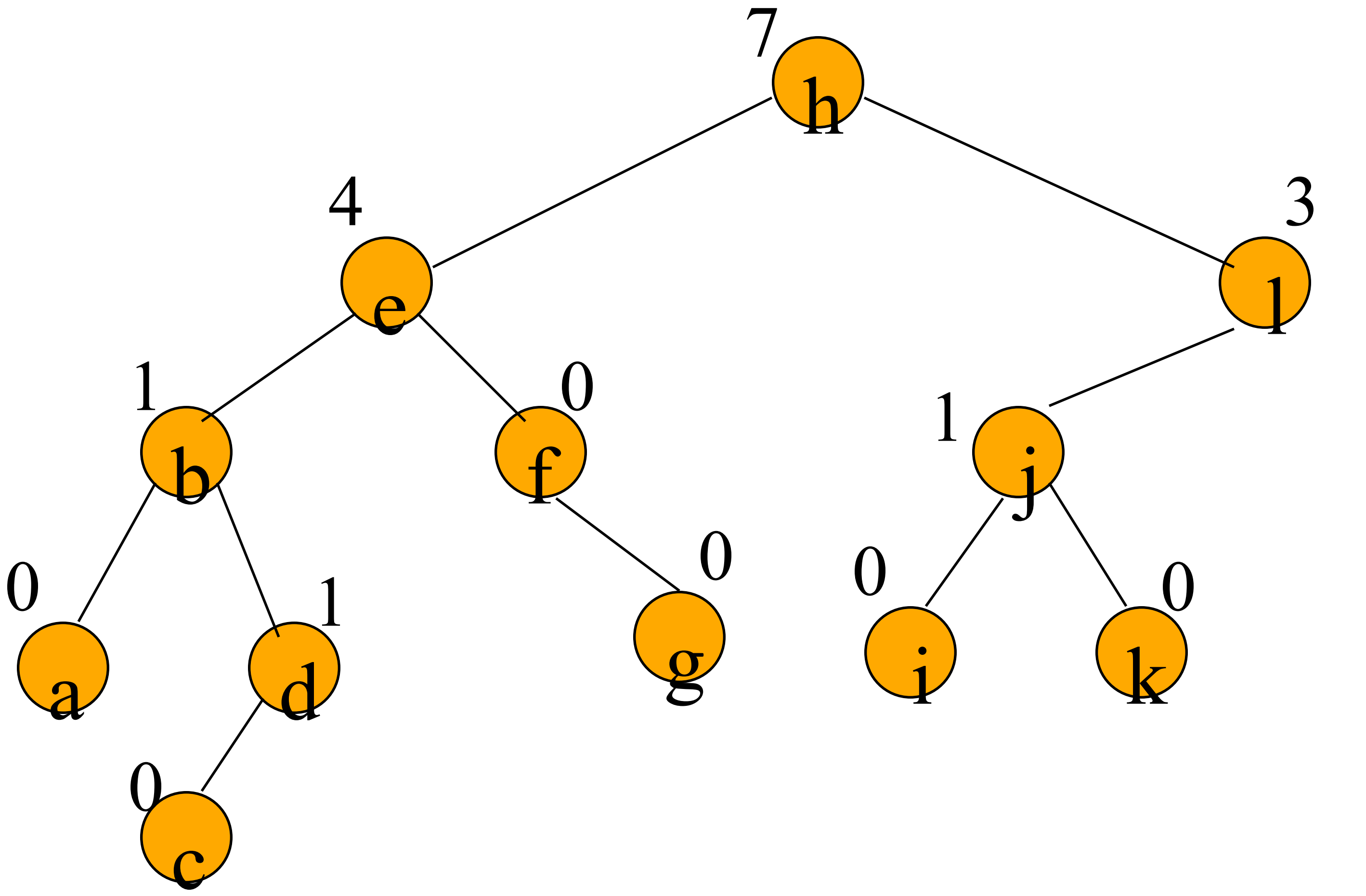
Can't use hash tables for either of these applications.

Linear List As Indexed Binary Tree



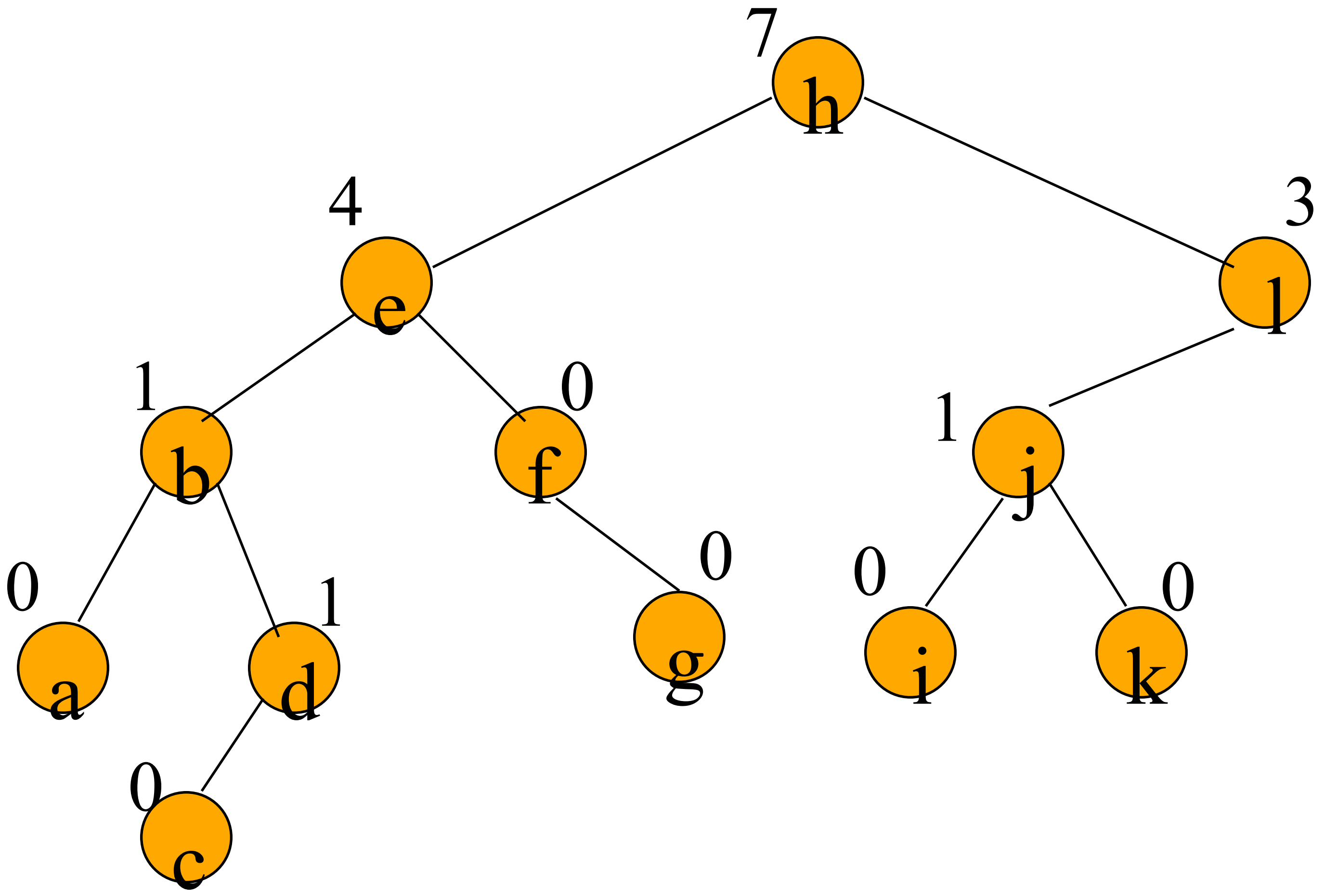
list = [a,b,c,d,e,f,g,h,i,j,k,l]

add(5,'m')



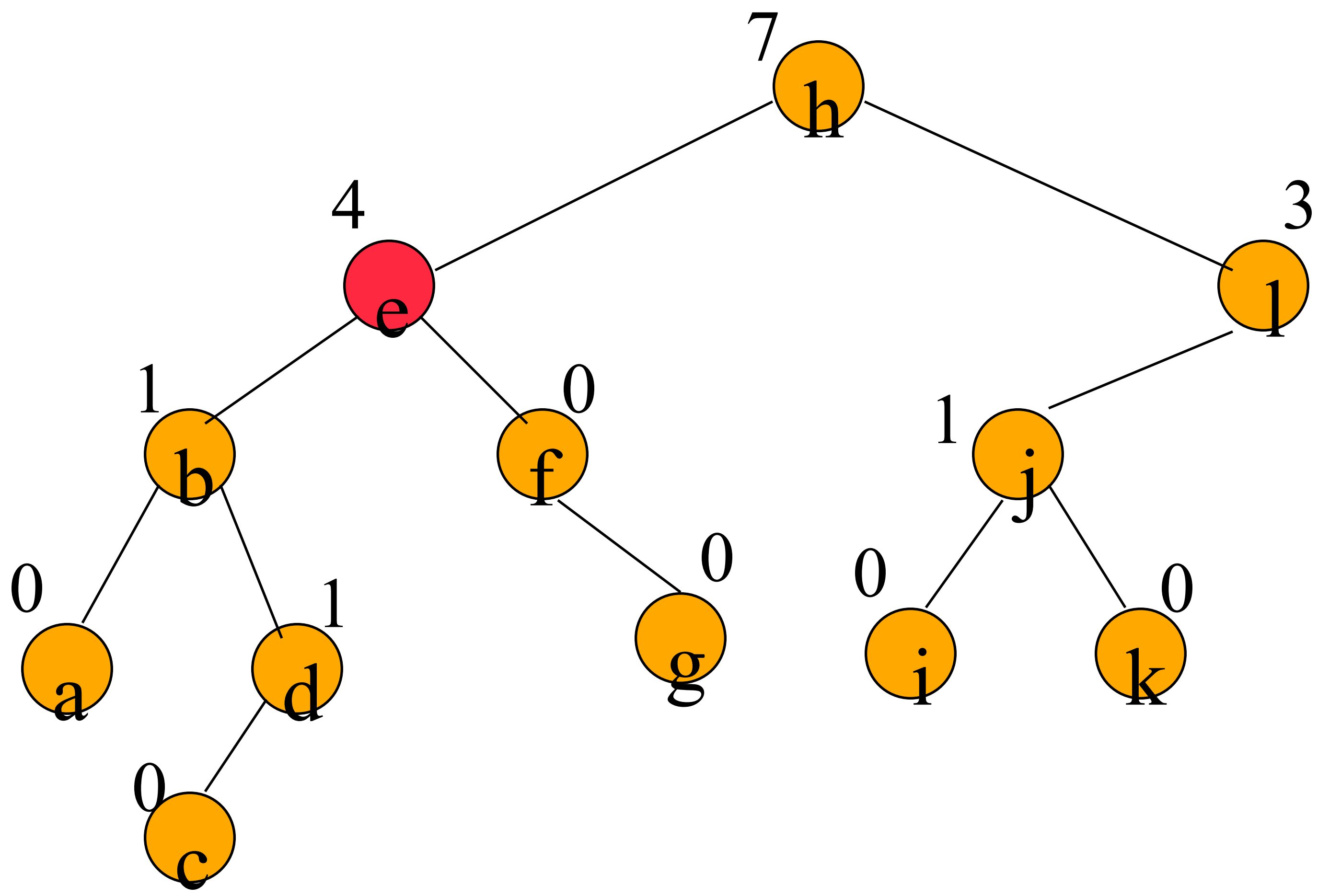
list = [a,b,c,d,e,f,g,h,i,j,k,l]

add(5,'m')



list = [a,b,c,d,e,**m**,f,g,h,i,j,k,l]
find node with element 4 (e)

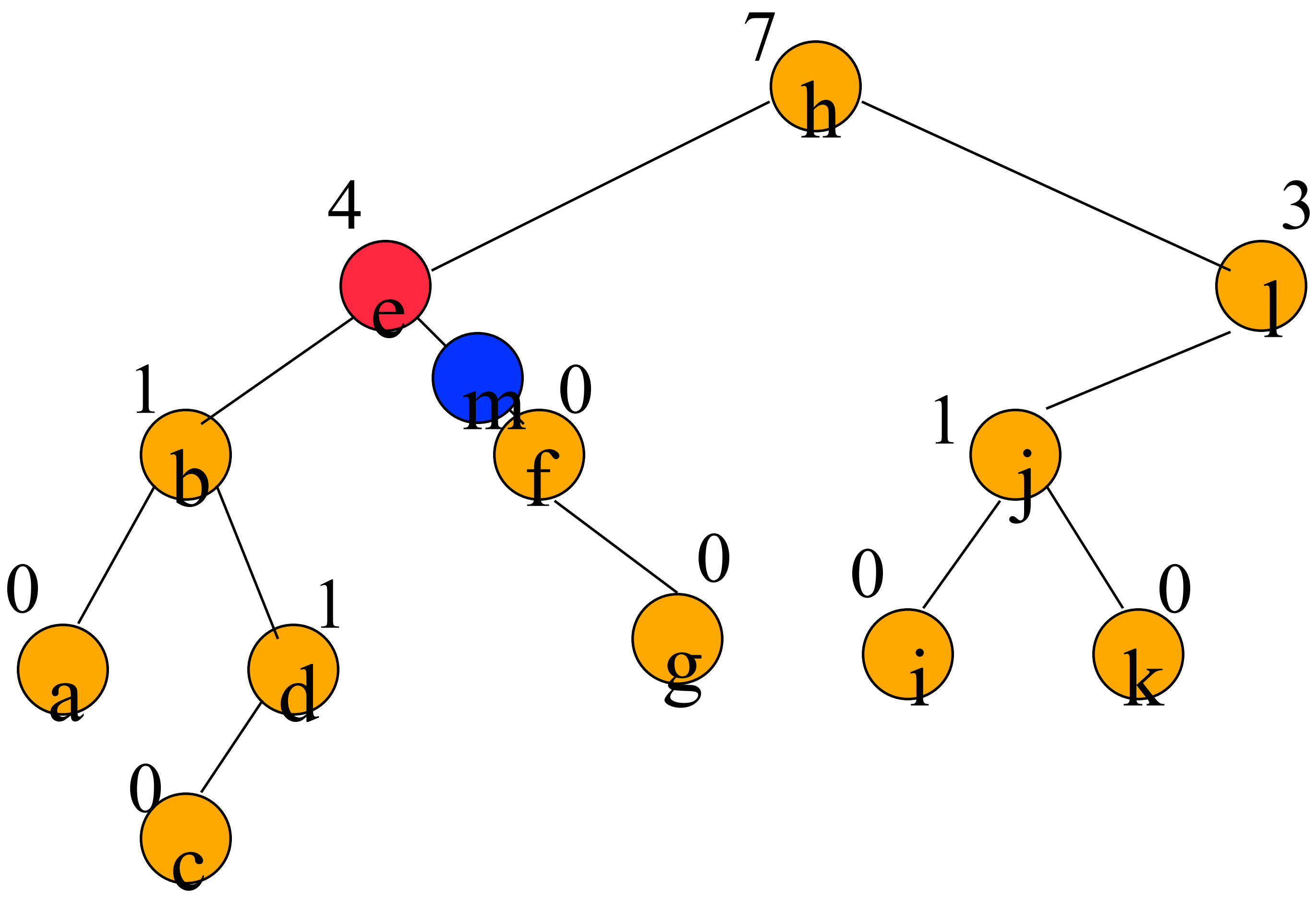
add(5,’m’)



list = [a,b,c,d,e, m,f,g,h,i,j,k,l]

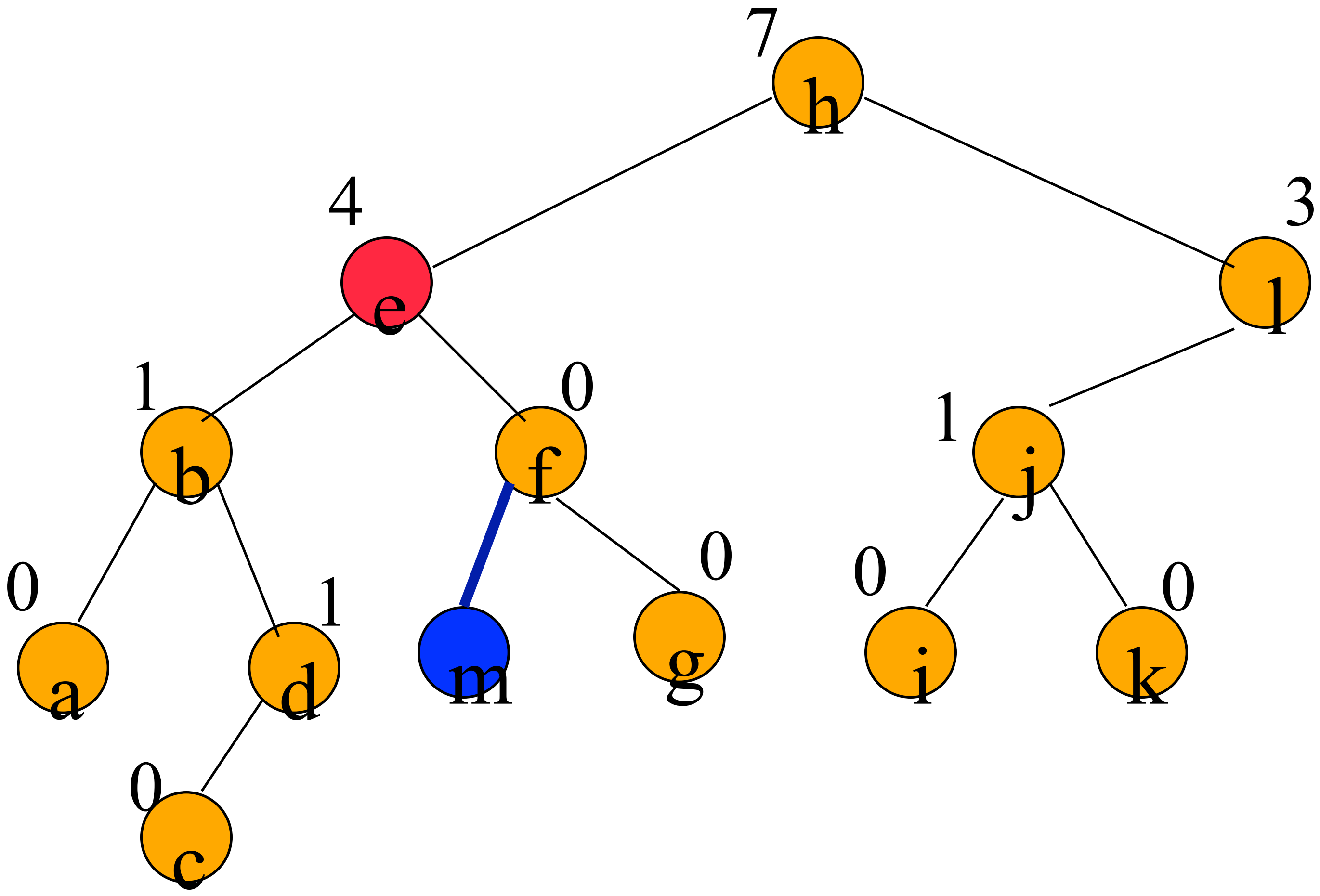
find node with element 4 (e)

add(5,’m’)



add m as right child of e; former right subtree of e becomes right subtree of

add(5,’m’)



add m as leftmost node in right subtree of e

add(5,’m’)

- Other possibilities exist.
- Must update some leftSize values on path from root to new node.
- Complexity is O(height).

- Exercise:
  - Data IndexedBinaryTree::Search(int idx);
  - Bool IndexedBinaryTree::Insert(int idx, int data);

Threaded Trees

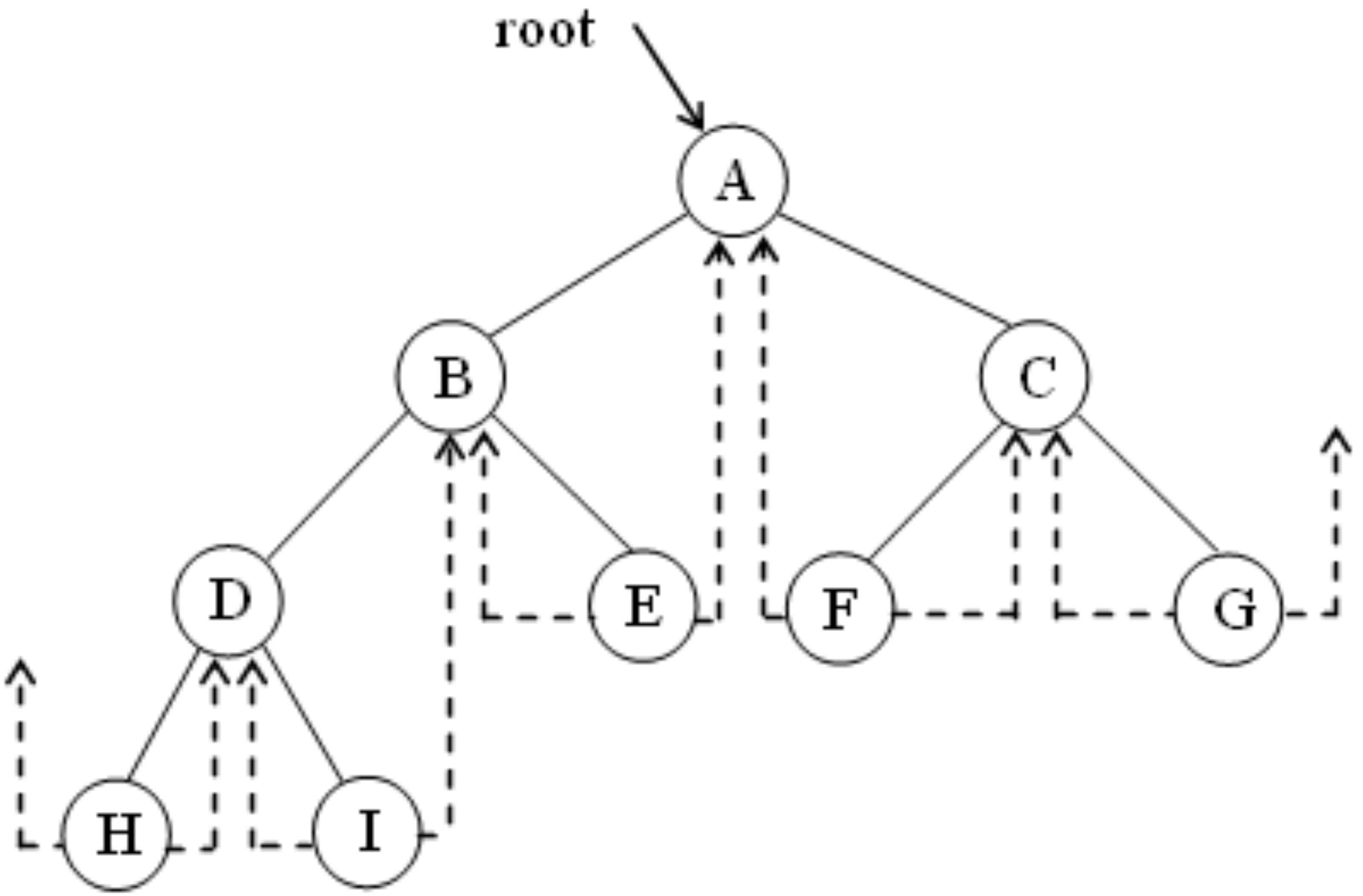
- Binary trees have a lot of wasted space:
  - the leaf nodes each have 2 null pointers
- We can use these pointers to help us in inorder traversals
- We have the pointers reference the next /previous node in an inorder traversal; called *threads*
- We need to know if a pointer is an actual link or a thread, so we keep a boolean for each pointer

The threads are constructed using the following rules:

(1) A 0 **rightChild** field at node p is replaced by a pointer to the inorder successor of p.

(2) A 0 **leftChild** field at node p is replaced by a pointer to the inorder predecessor of p.

The following is a threaded tree, in which node E has a predecessor thread pointing to B and a successor thread to A.



To distinguish between threads and normal pointers, add two bool fields:

- **leftThread**
- **rifgtThread**

If t→leftThread == true, then t→leftChild contains a thread

otherwise a pointer to left child

Similar for t→rightThread.

```
template <class T>
class ThreadedNode {
friend class ThreadedTree;
private:
   bool  leftThread;
   ThreadedNode * leftChild;
   T data;
   ThreadedNode * rightChild;
   bool  rightThread;
};
```

```
template <class T>
class ThreadedTree {
public:
   // Tree operations
…
private:
   ThreadedNode *root;
};
```

Let ThreadedInorderIterator be a nested class of ThreadedTree:

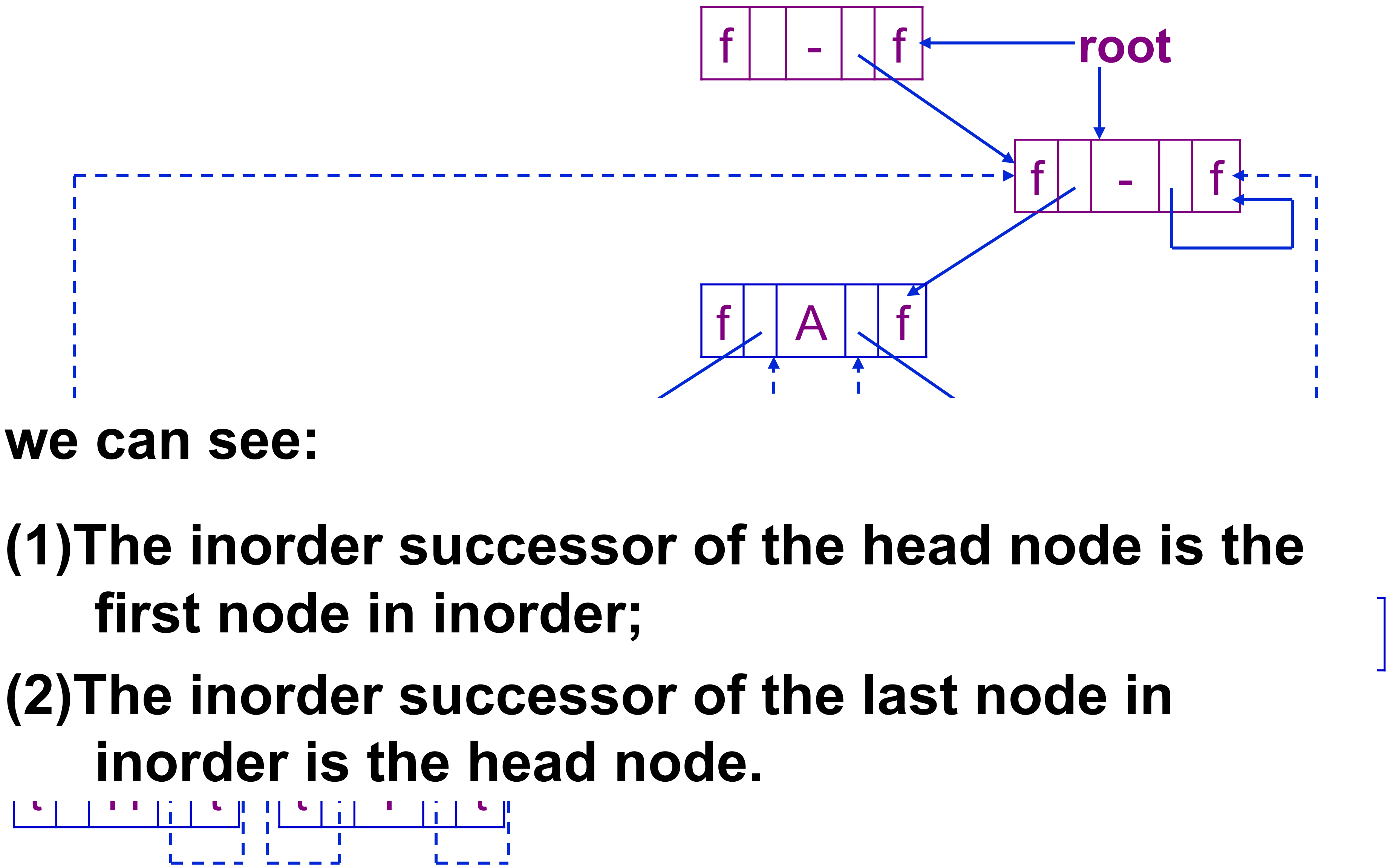
```
class ThreadedInorderIterator {
public:
   T* Next();
   ThreadedInorderIterator()
        { currentNode = root; }
private:
   ThreadedNode<T>* currentNode;
};
```

To make the left thread of the first node in inorder and the right thread of the last node in inorder un-dangle, we assume a head node for all threaded binary tree, let the two threads point to the head.

The original tree is the left subtree of the head, and the rightChild of head points to the head itself.

leftThread	leftChild	data	rightChild	rightThread
true				**false**

An empty threaded binary tree



we can see:

(1)The inorder successor of the head node is the first node in inorder;

(2)The inorder successor of the last node in inorder is the head node.

memory representation of threaded tree

Inorder Traversal of a Threaded Binary Tree

Observe:

(1)If x→rightThread==true

the inorder successor of x is

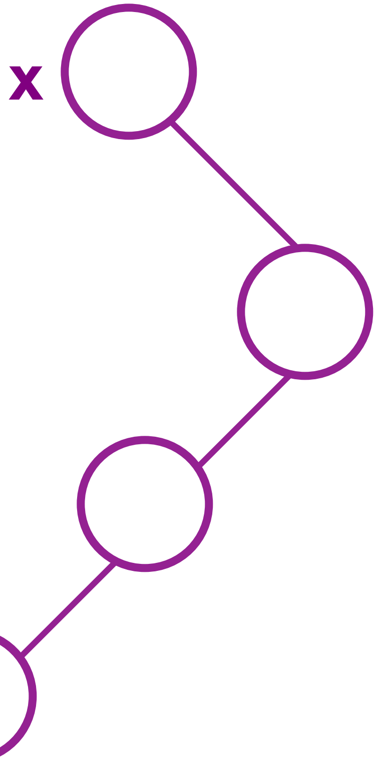
x→rightChild;

(2) If x→rightThread==false

the inorder successor of x is obtained by

following a path of leftChild from the right child of x until

a node with leftThread==true is reached.



Thus we have:

```
T* ThreadedInorderIterator::Next()
{ // Return the inorder successor of currentNode in a
threaded
  //  binary tree
     ThreadedNode<T>* temp=currentNode→rightChild;
     if (! currentNode→rightThread)
         while (!temp→leftThread)
           temp=temp→leftChild;
     currentNode=temp;
     if (currentNode==root)
          return 0; //no next
     else
          return &currentNode→data;
}
```

Note that when currentNode == root, Next() return the 1st node of inorder, thus we can use the following function to do an inorder travesal of a threaded binary tree:

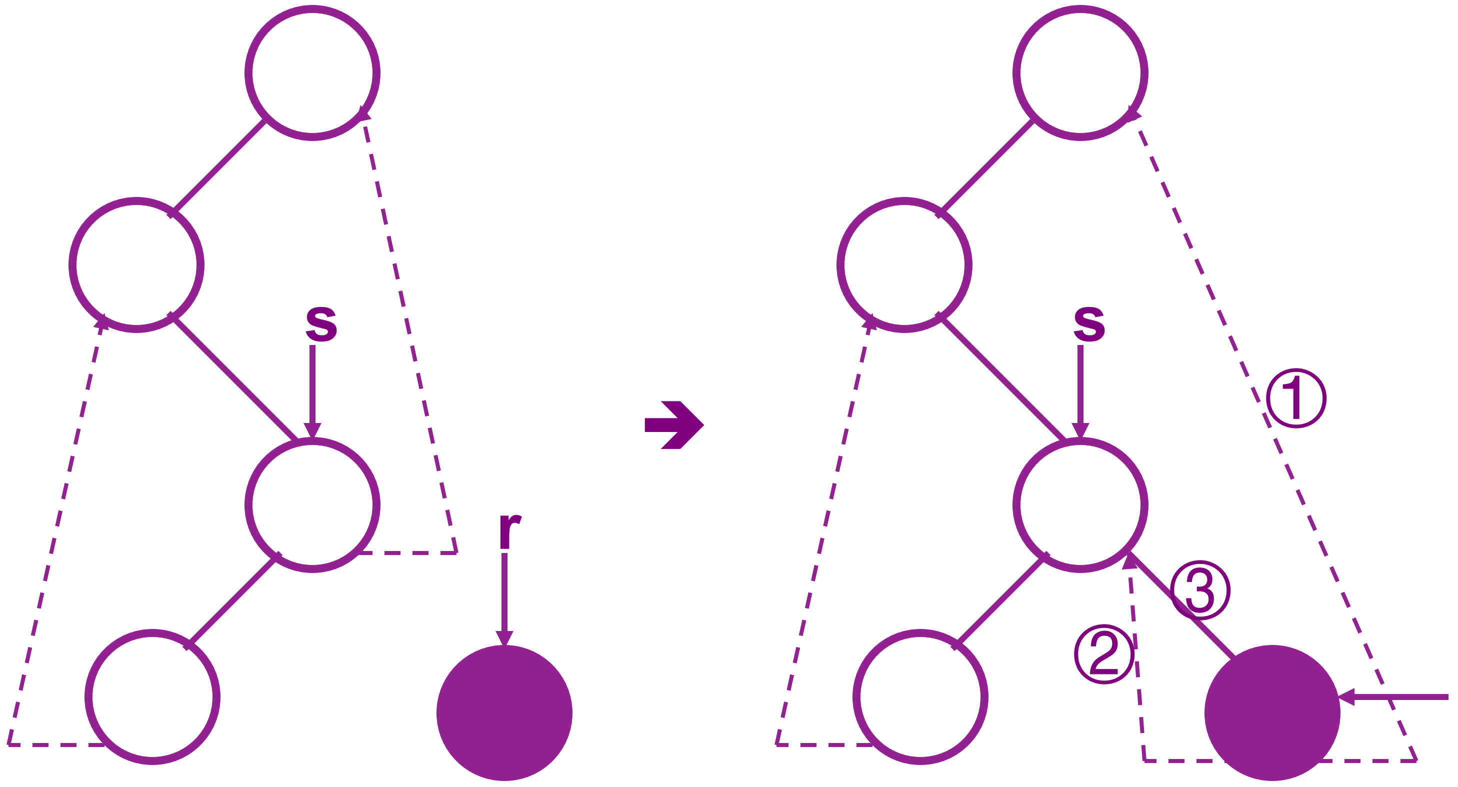
```
template <class T>
void ThreadedTree::Inorder()
{
   ThreadedInorderiterator ti;
   for (T* p = ti.Next(); p; p = ti.Next())
   Visit(*p);
}
```

Inserting a Node into a Threaded Binary Tree

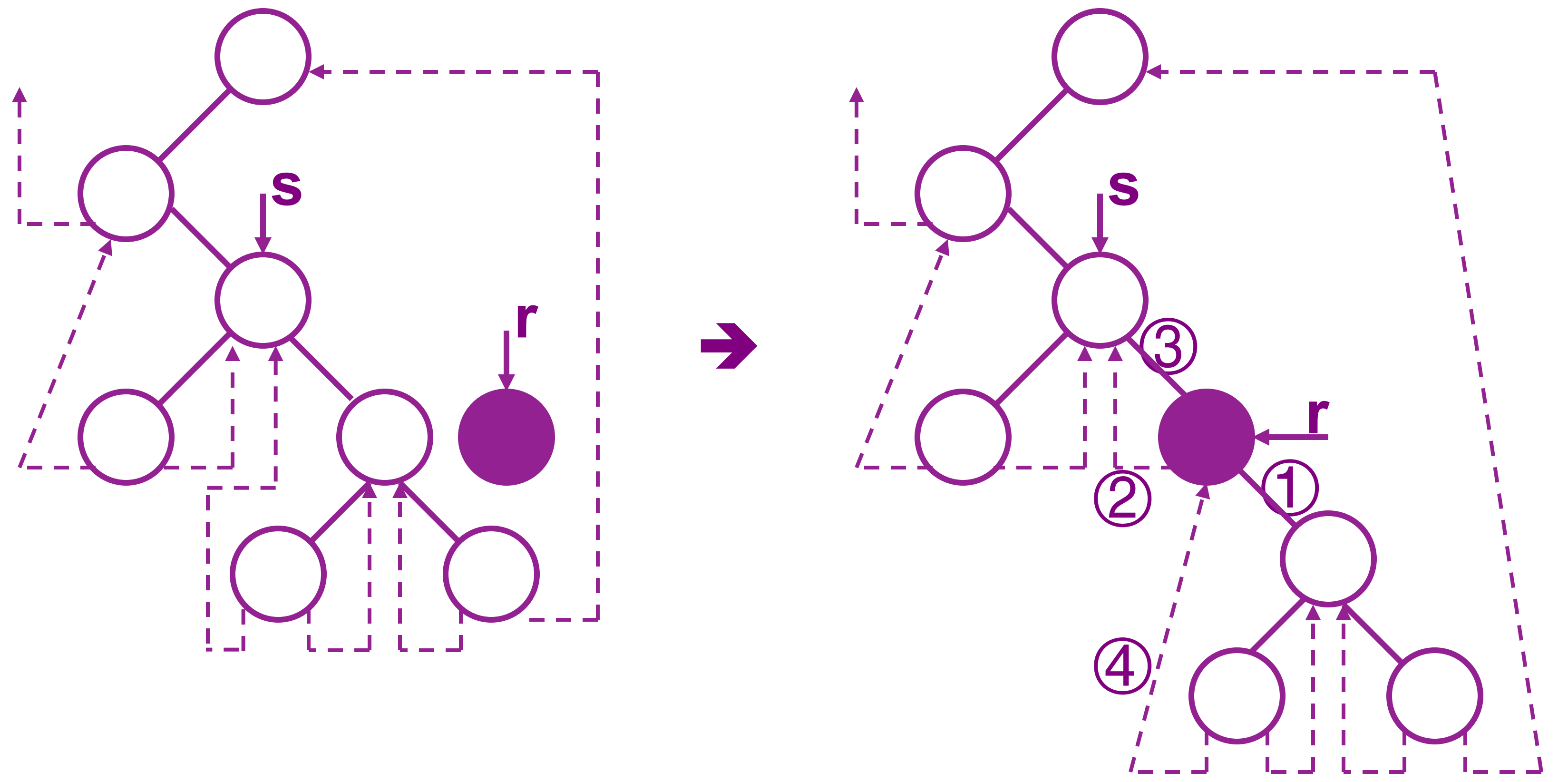
Insertion into a threaded tree provides the function for growing threaded tree.

We shall study only the case of inserting r as the right child of s. The left child case is similar.

(1) If s→rightThread==true, as:



(2) If s→rightThread==false, as:



In both (1) and (2), actions ①,②, ③ are the same,④ is special for (2).

```
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;                    // ①
    r→rightThread=s→rightThread;     // ① note s!=t.root,
    r→leftChild=s;                           // ②
    r→leftThread=true;                       // ②
    s→rightChild=r;                               // ③
    s→rightThread=false;                     // ③
    if (! r→rightThread) {     // case (2)
        ThreadedNode<T>* temp=InorderSucc(r);  // ④
        temp→leftChild=r;                              // ④
    }
}
```

Exercises: P277-1, P278-4

Given a binary tree, make it an inorder threaded binary tree.

Priority Queues

Two kinds of priority queues:

- Min priority queue.
- Max priority queue.

Min Priority Queue

- Collection of elements.
- Each element has a priority or key.
- Supports following operations:
  - isEmpty
  - size
  - add/put an element into the priority queue
  - get element with min priority
  - remove element with min priority

Max Priority Queue

- Collection of elements.
- Each element has a priority or key.
- Supports following operations:
  - isEmpty
  - size
  - add/put an element into the priority queue
  - get element with max priority
  - remove element with max priority

ADT MaxHeap

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

Complexity Of Operations

First idea:

- Linear List
  - Unordered Linear List
  - Ordered Linear List

Complexity

- isEmpty
- Push
- Pop

Complexity Of Operations

Two good implementations are heaps and leftist trees.

isEmpty, size, and get => $O(1)$ time

put and remove => $O(\log n)$ time where n is the size of the priority queue

Applications

Sorting

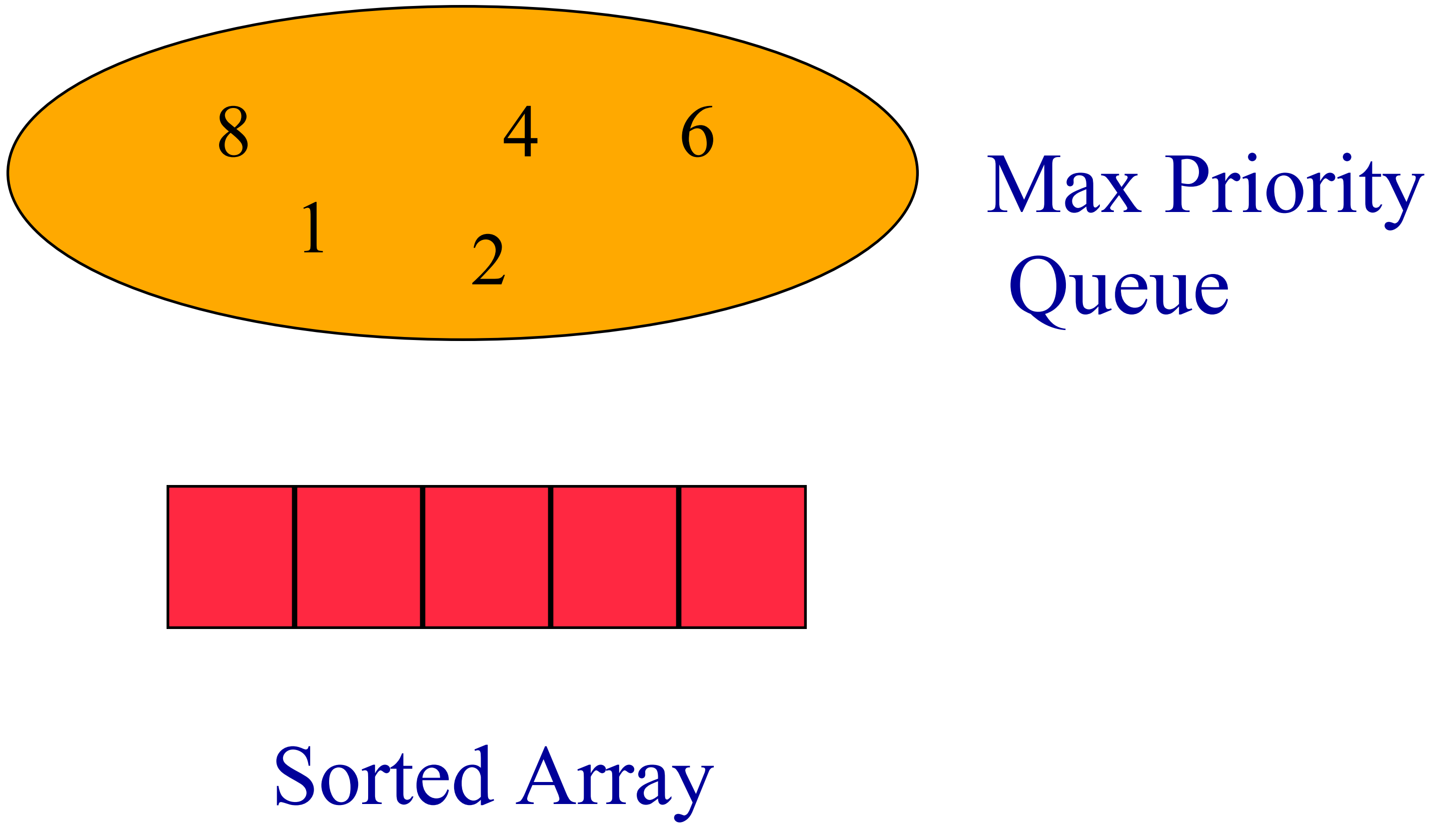
- use element key as priority
- put elements to be sorted into a priority queue
- extract elements in priority order
  - if a min priority queue is used, elements are extracted in ascending order of priority (or key)
  - if a max priority queue is used, elements are extracted in descending order of priority (or key)

Sorting Example

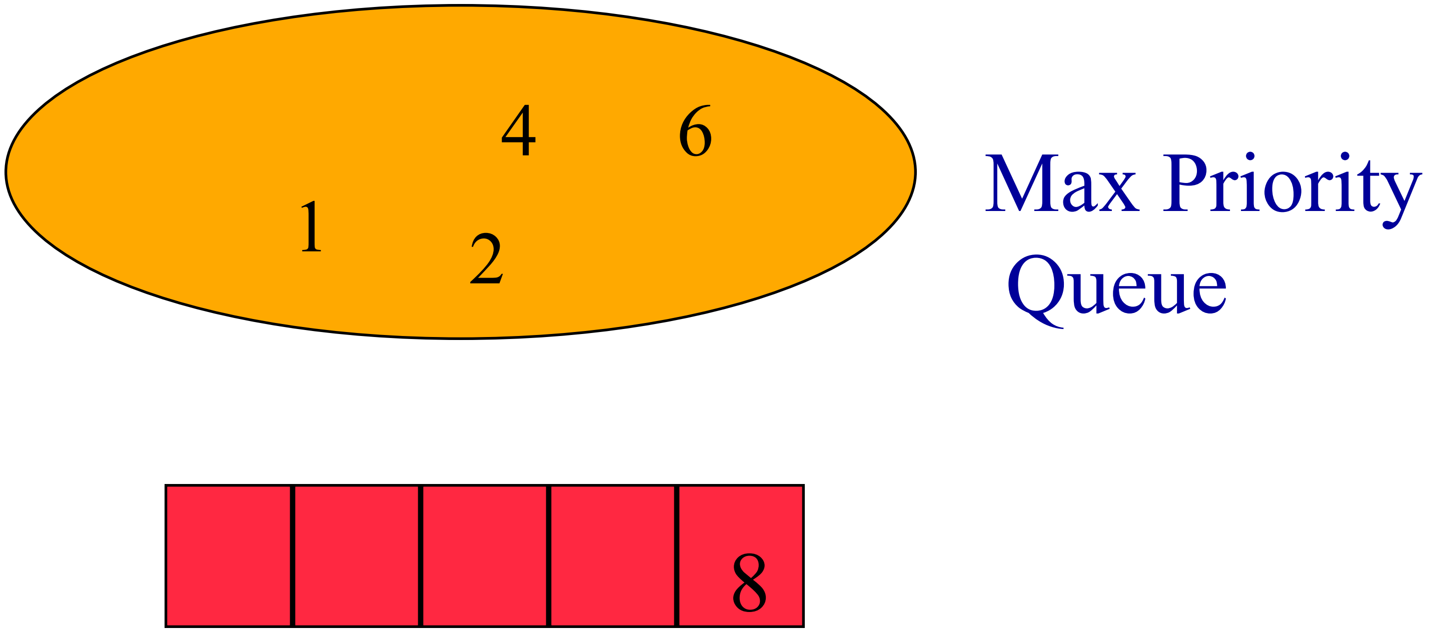
Sort five elements whose keys are 6, 8, 2, 4, 1 using a max priority queue.

- Put the five elements into a max priority queue.
- Do five remove max operations placing removed elements into the sorted array from right to left.

After Putting Into Max Priority Queue

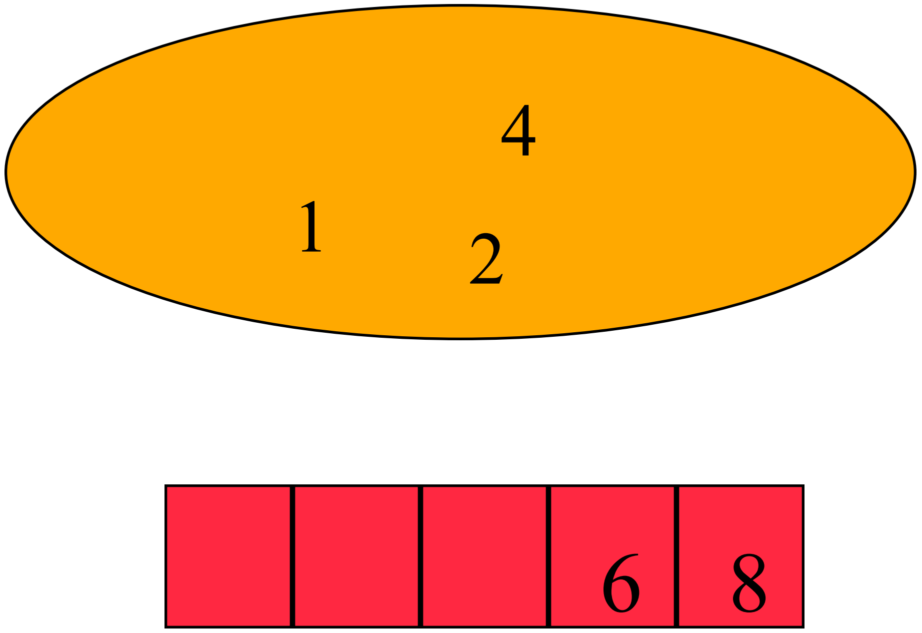


After First Remove Max Operation



Sorted Array

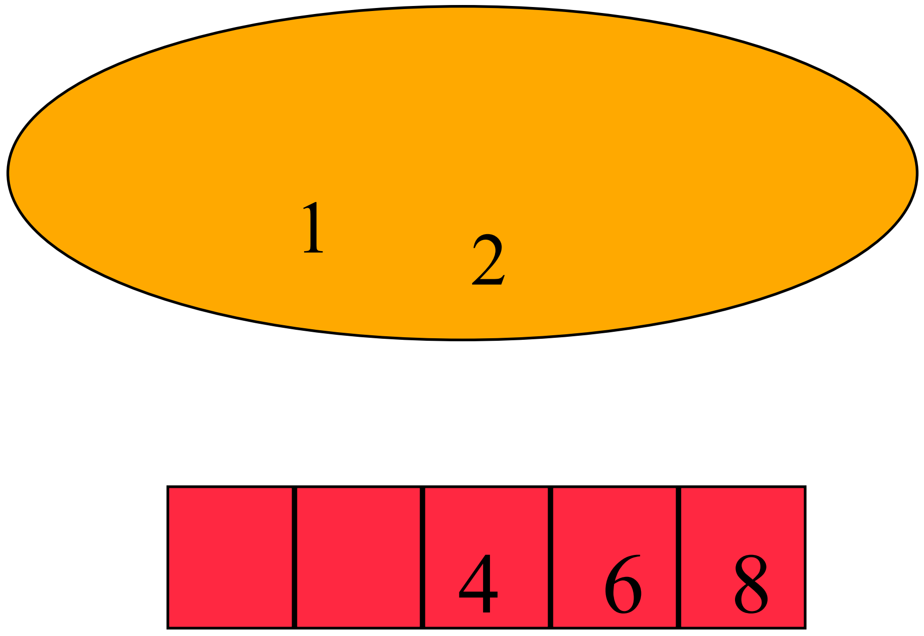
After Second Remove Max Operation



Max Priority Queue

Sorted Array

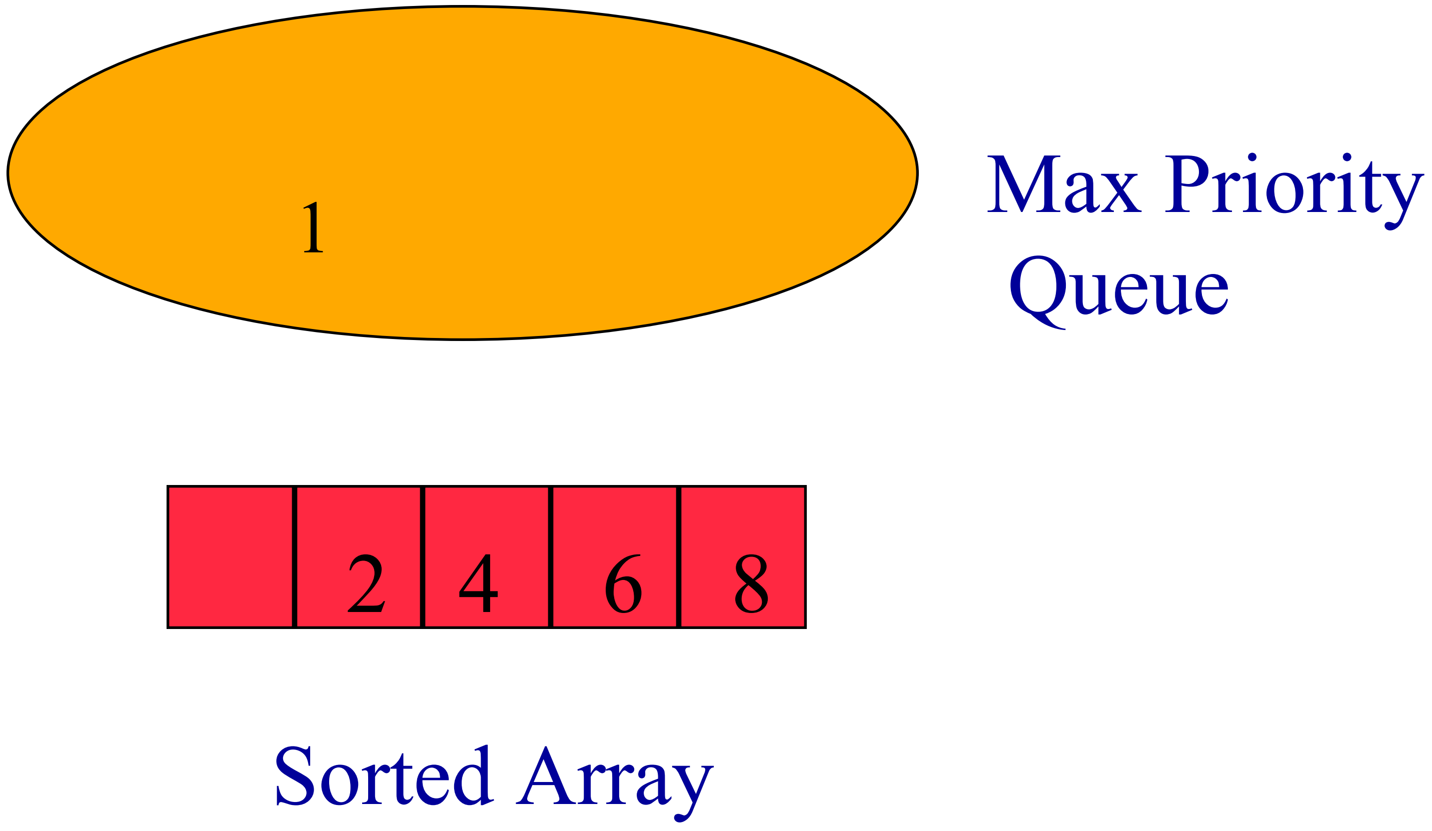
After Third Remove Max Operation



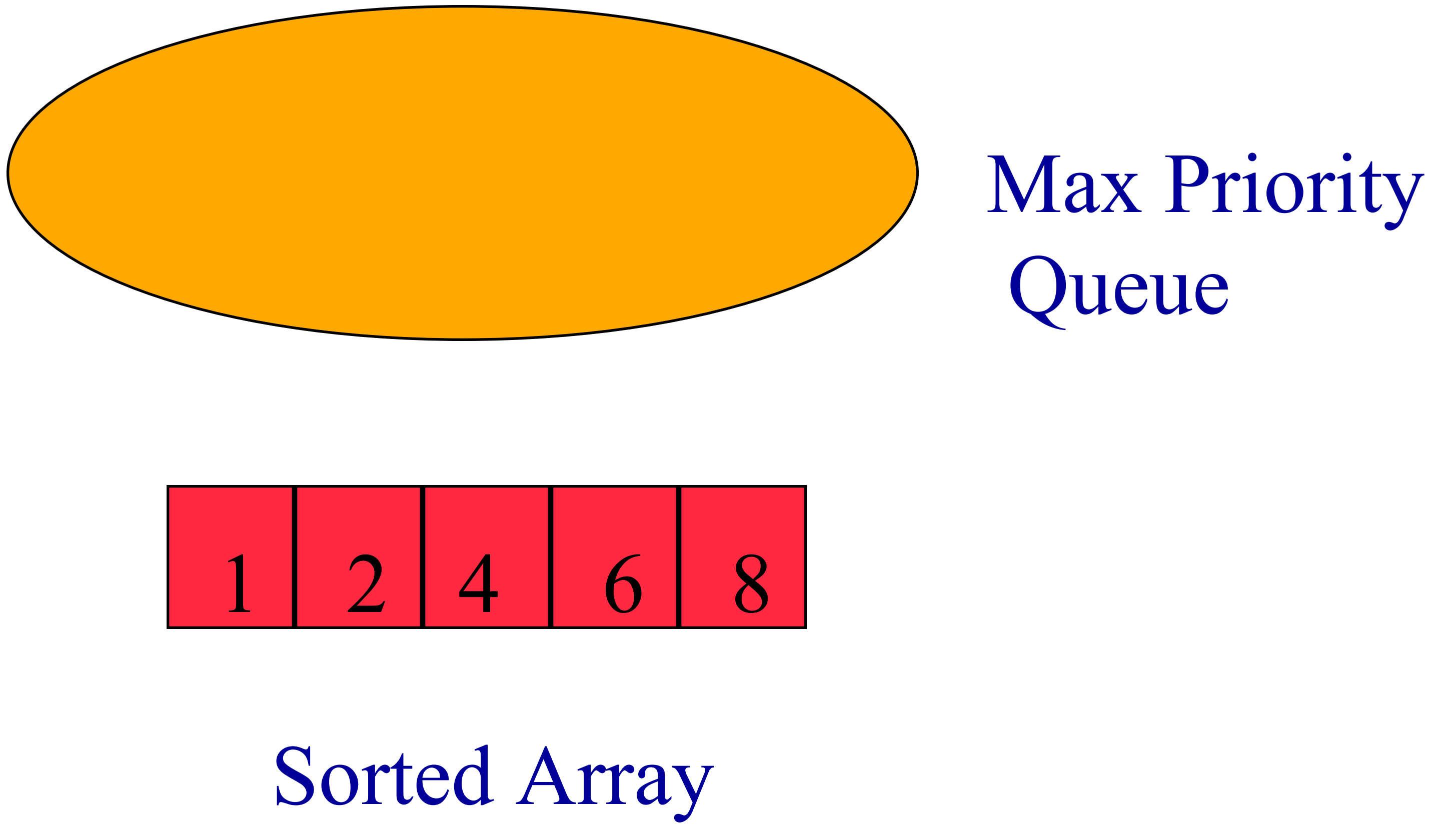
Max Priority Queue

Sorted Array

After Fourth Remove Max Operation



After Fifth Remove Max Operation



Complexity Of Sorting

Sort n elements.

- n put operations => O(n log n) time.
- n remove max operations => O(n log n) time.
- total time is O(n log n).
- compare with sort methods $O(n^2)$*

Machine Scheduling

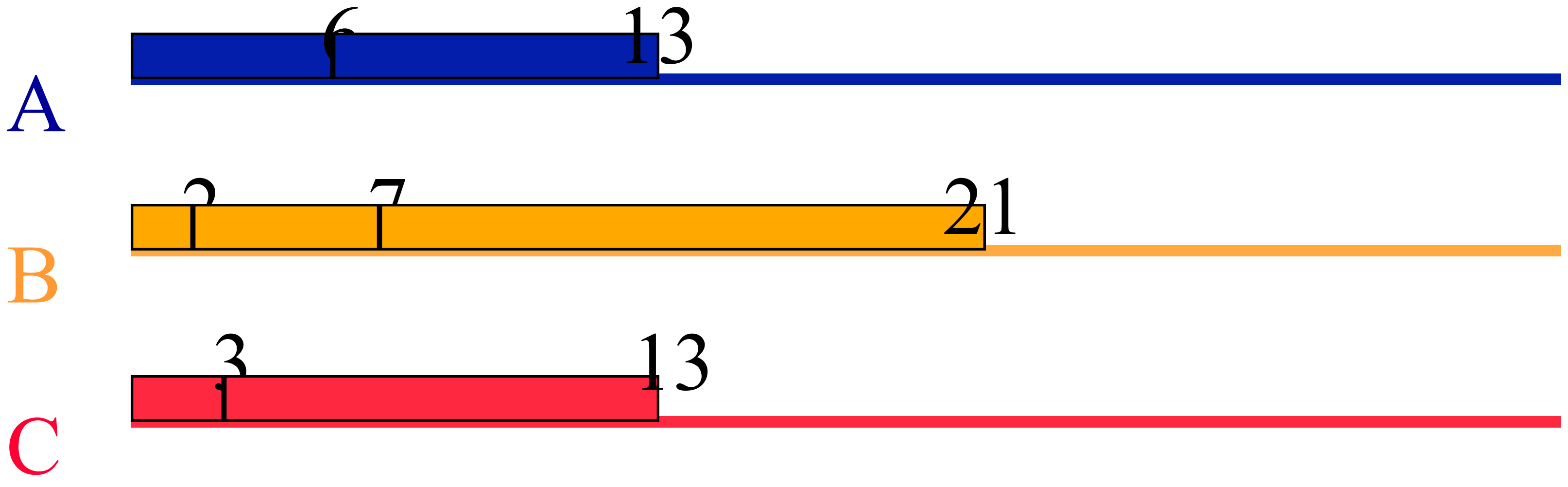
- m identical machines
- n jobs/tasks to be performed
- assign jobs to machines so that the time at which the last job completes is minimum

Machine Scheduling Example

3 machines and 7 jobs

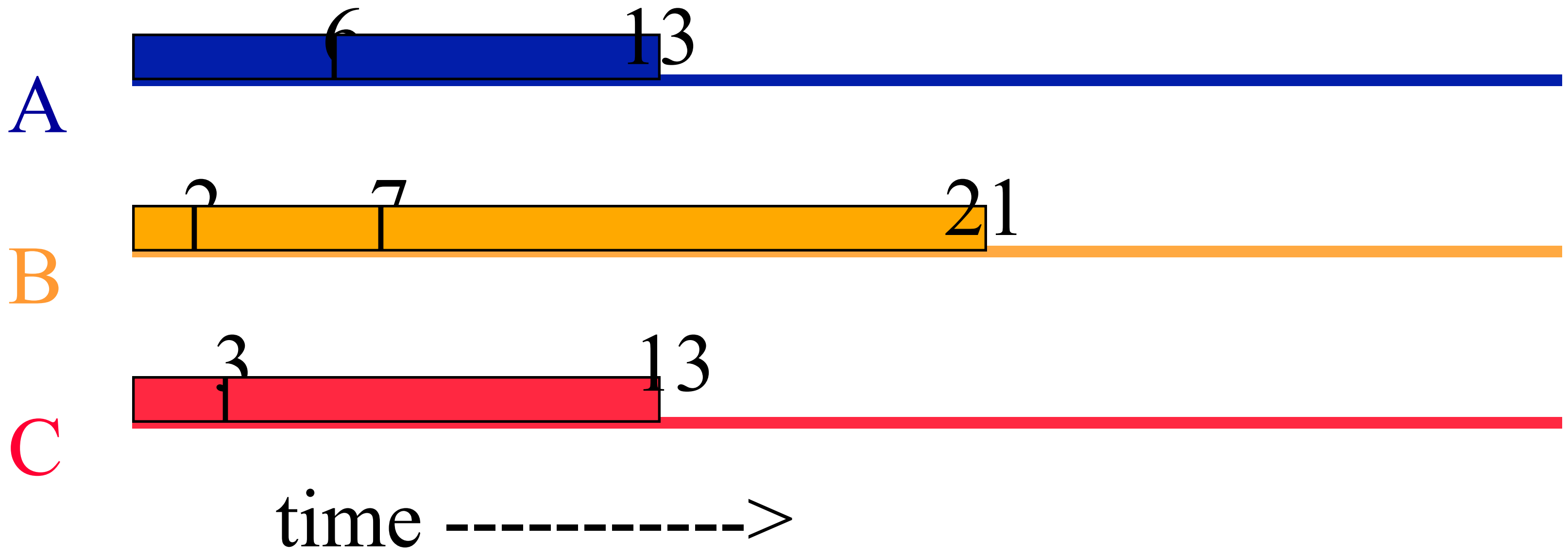
job times are [6, 2, 3, 5, 10, 7, 14]

possible schedule



time ----------->

Machine Scheduling Example



Finish time = 21

Objective: Find schedules with minimum finish time.

LPT Schedules

Longest Processing Time first.

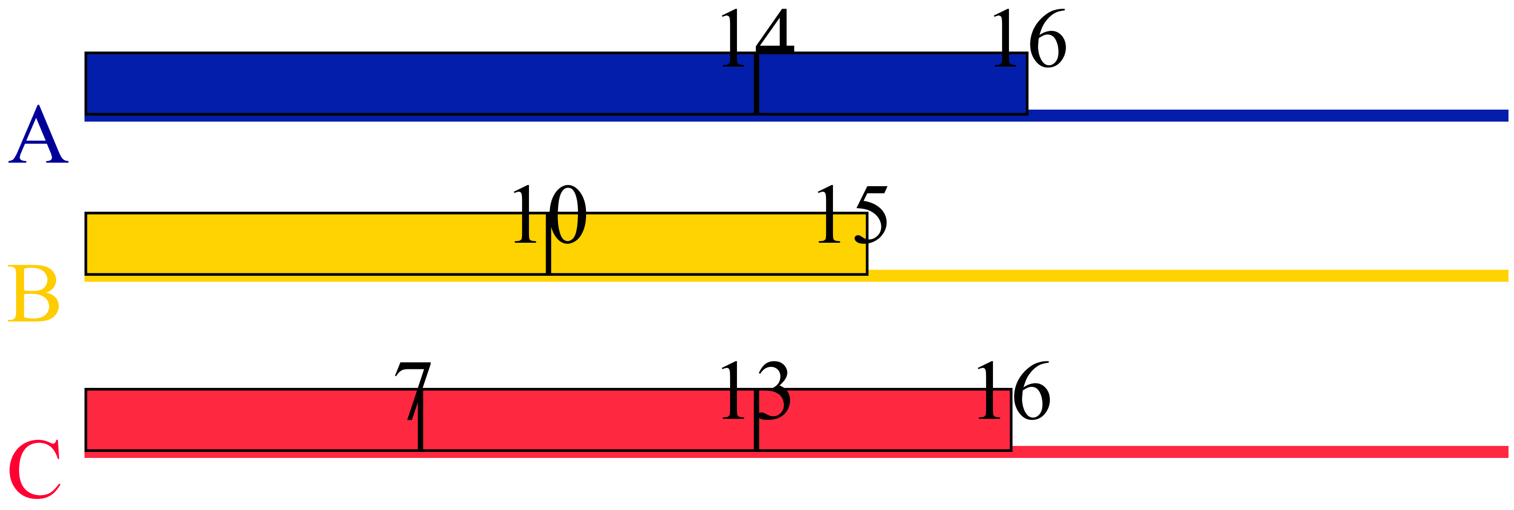
Jobs are scheduled in the order

14, 10, 7, 6, 5, 3, 2

Each job is scheduled on the machine on which it finishes earliest.

LPT Schedule

[14, 10, 7, 6, 5, 3, 2]



Finish time is 16!

LPT Schedule

- LPT rule does not guarantee minimum finish time schedules.
- Usually LPT finish time is much closer to minimum finish time.
- Minimum finish time scheduling is NP-hard.

NP-hard Problems

- Infamous class of problems for which no one has developed a polynomial time algorithm.
- That is, no algorithm whose complexity is $O(n^k)$ for any constant k is known for any NP -hard problem.
- The class includes thousands of real-world problems.
- Highly unlikely that any NP-hard problem can be solved by a polynomial time algorithm.

NP-hard Problems

- Since even polynomial time algorithms with degree $k > 3$ (say) are not practical for large n, we must change our expectations of the algorithm that is used.
- Usually develop fast heuristics for NP-hard problems.
  - Algorithm that gives a solution close to best.
  - Runs in acceptable amount of time.
- LPT rule is good heuristic for minimum finish time scheduling.

Complexity Of LPT Scheduling

- Sort jobs into decreasing order of task time.
  - O(n log n) time (n is number of jobs)
- Schedule jobs in this order.
  - assign job to machine that becomes available first
  - must find minimum of m (m is number of machines) finish times
  - takes O(m) time using simple strategy
  - so need O(mn) time to schedule all n jobs.

Using A Min Priority Queue

- Min priority queue has the finish times of the m machines.
- Initial finish times are all 0.
- To schedule a job, remove machine with minimum finish time from the priority queue.
- Update the finish time of the selected machine and put the machine back into the priority queue.

Using A Min Priority Queue

- m put operations to initialize priority queue
- 1 remove min and 1 put to schedule each job
- each put and remove min operation takes O(log m) time
- time to schedule is O(n log m)
- overall time is

$$O(n \log n + n \log m) = O(n \log (mn))$$

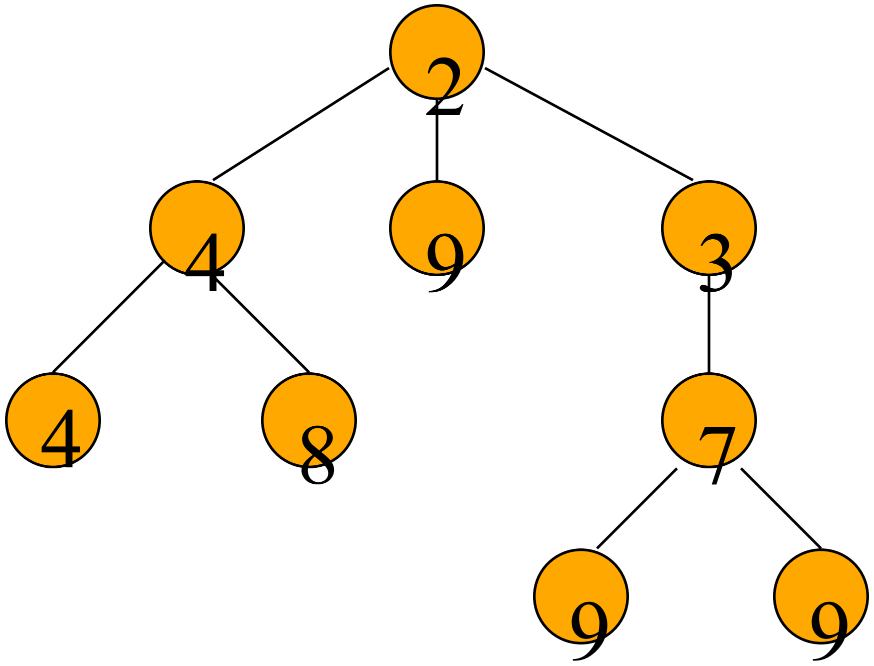
Min Tree Definition

Each tree node has a value.

Value in any node is the minimum value in the subtree for which that node is the root.

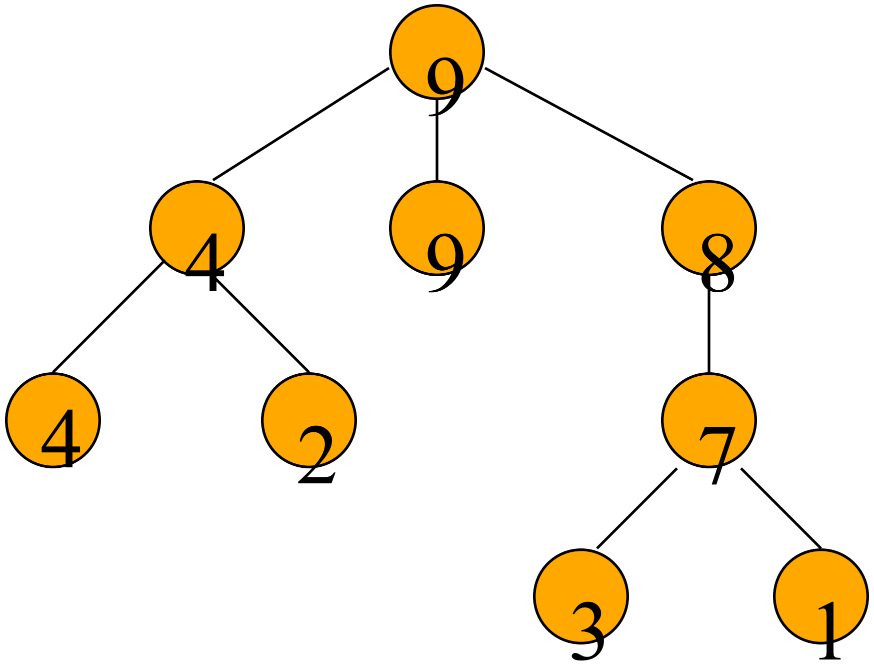
Equivalently, no descendent has a smaller value.

Min Tree Example



Root has minimum element.

Max Tree Example

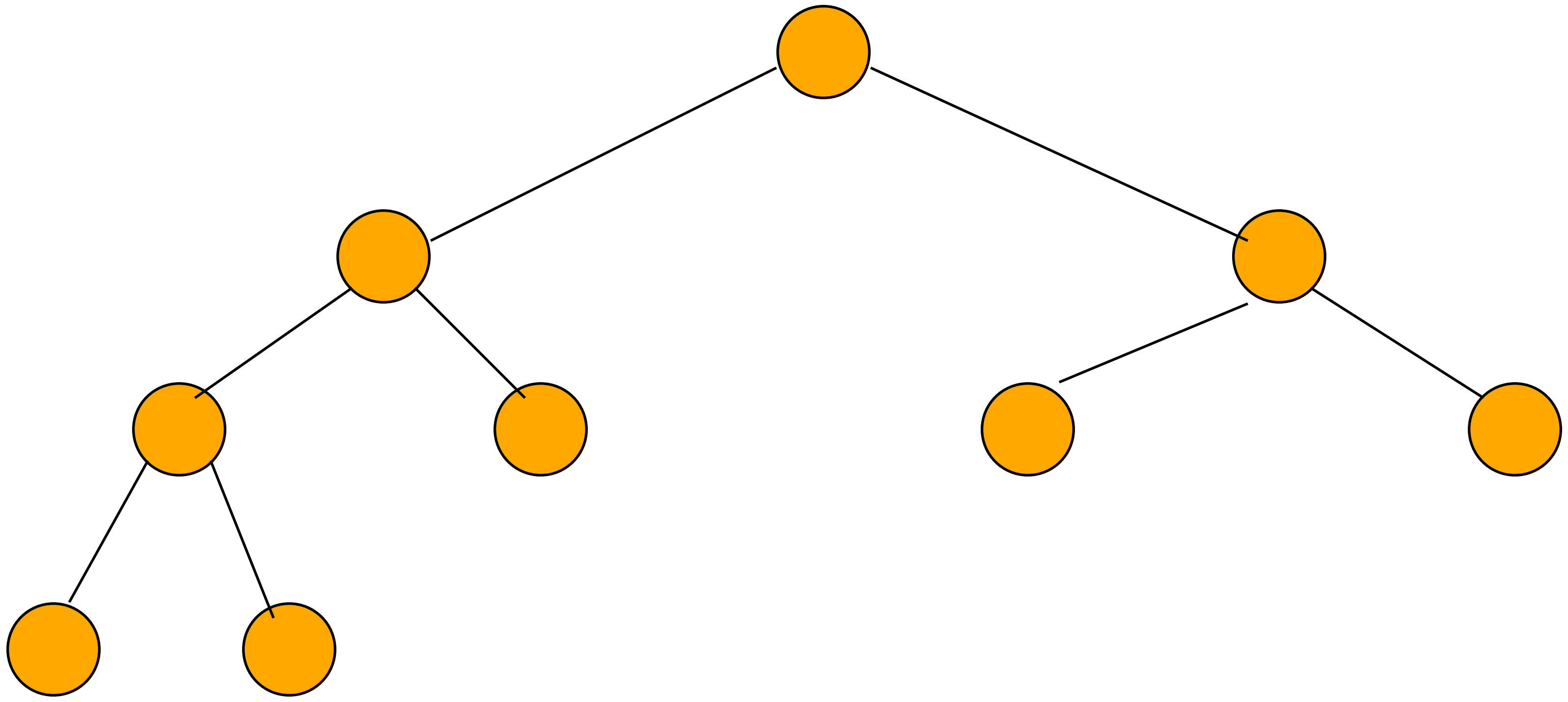


Root has maximum element.

Min Heap Definition

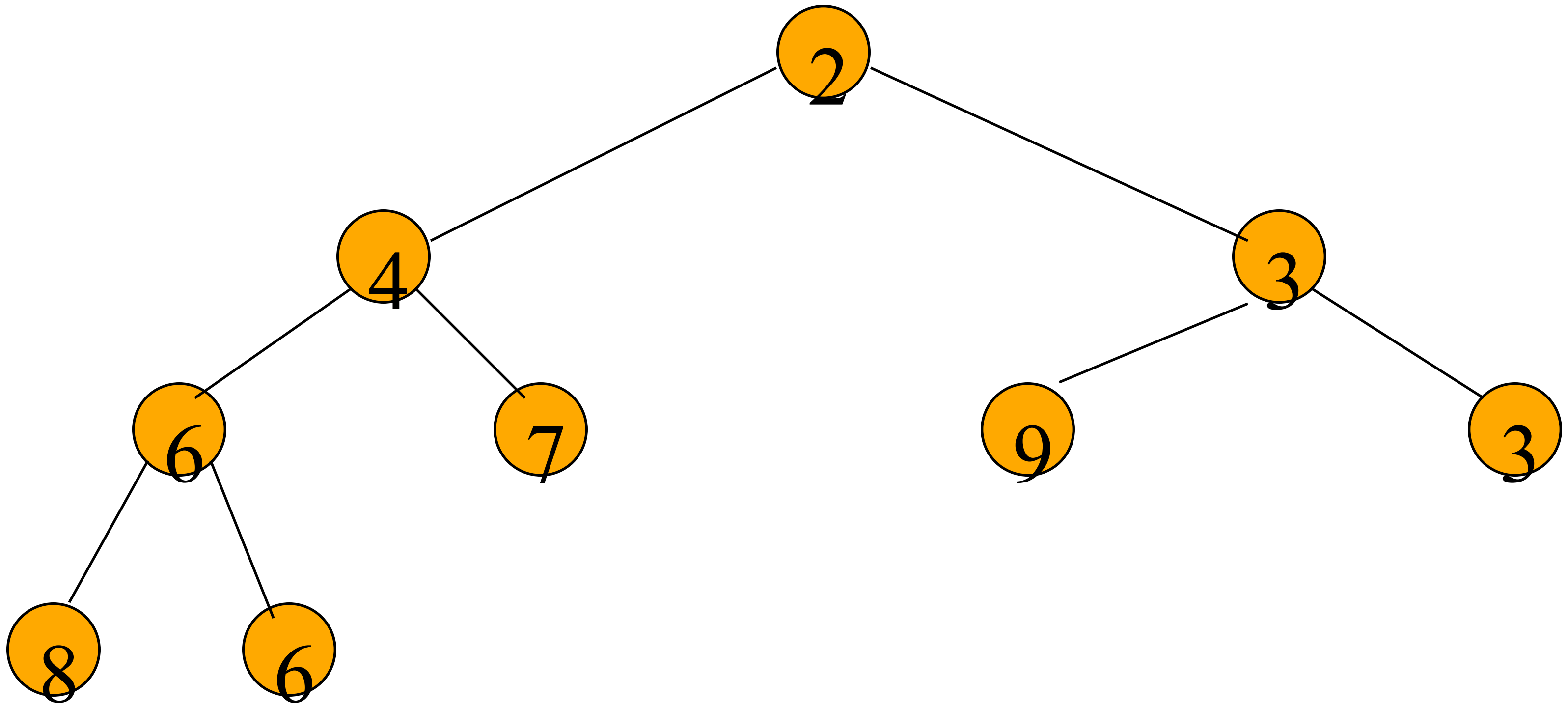
- complete binary tree
- min tree

Min Heap With 9 Nodes



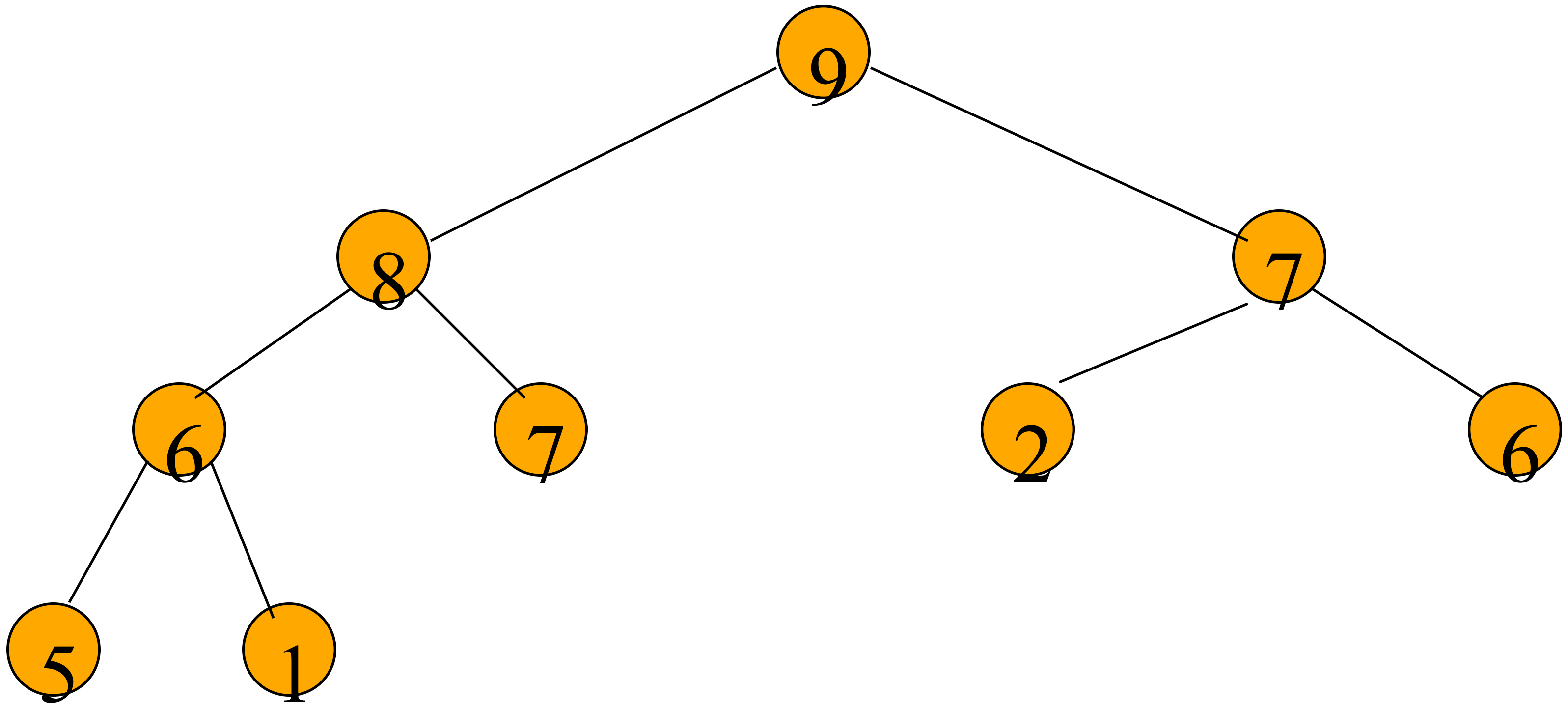
Complete binary tree with 9 nodes.

Min Heap With 9 Nodes



Complete binary tree with 9 nodes that is also a min tree.

Max Heap With 9 Nodes



Complete binary tree with 9 nodes
that is also a max tree.

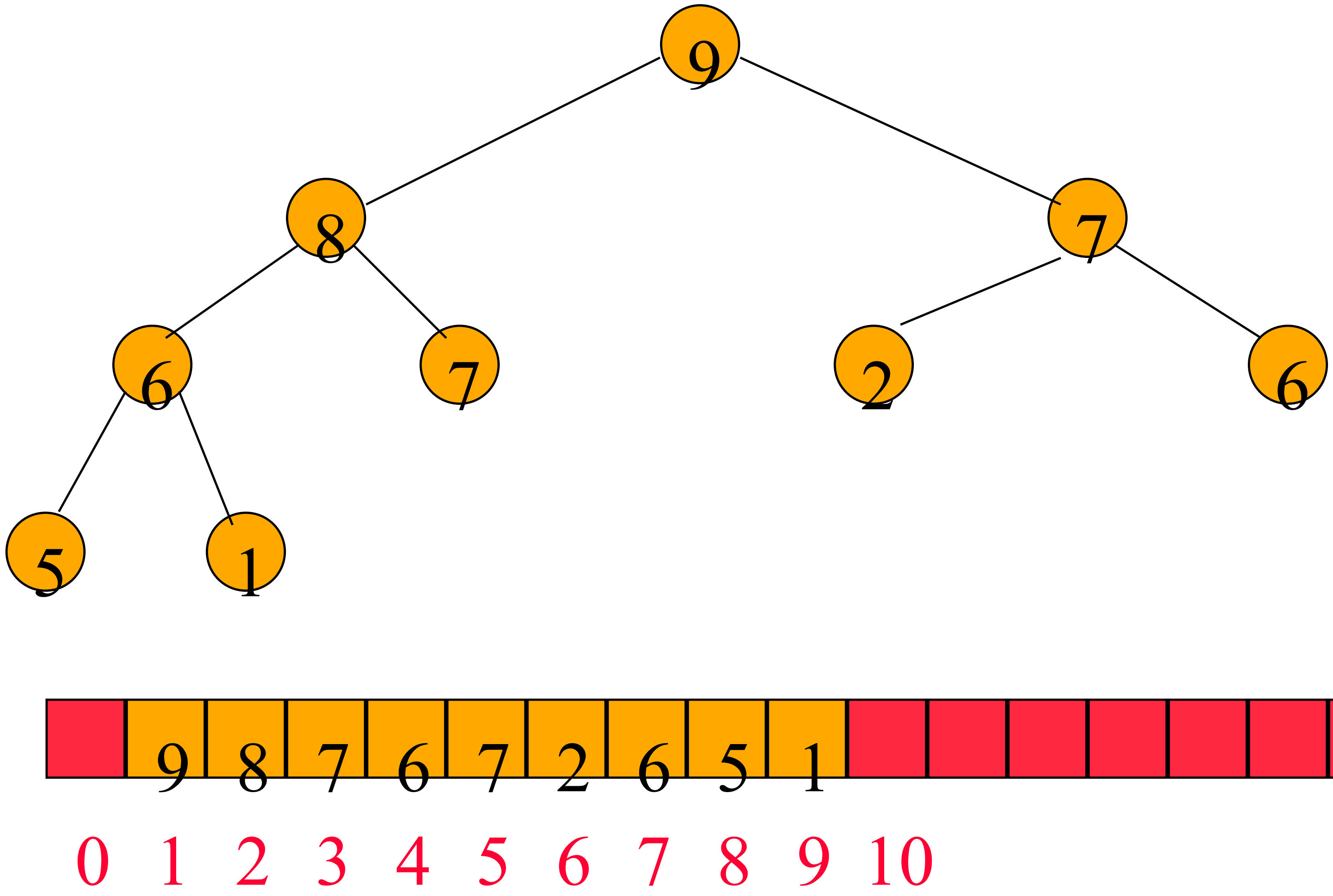
Heap Height

Since a heap is a complete binary tree, the height of an n node heap is $\log_2 (n+1)$.

How to represent a Heap?

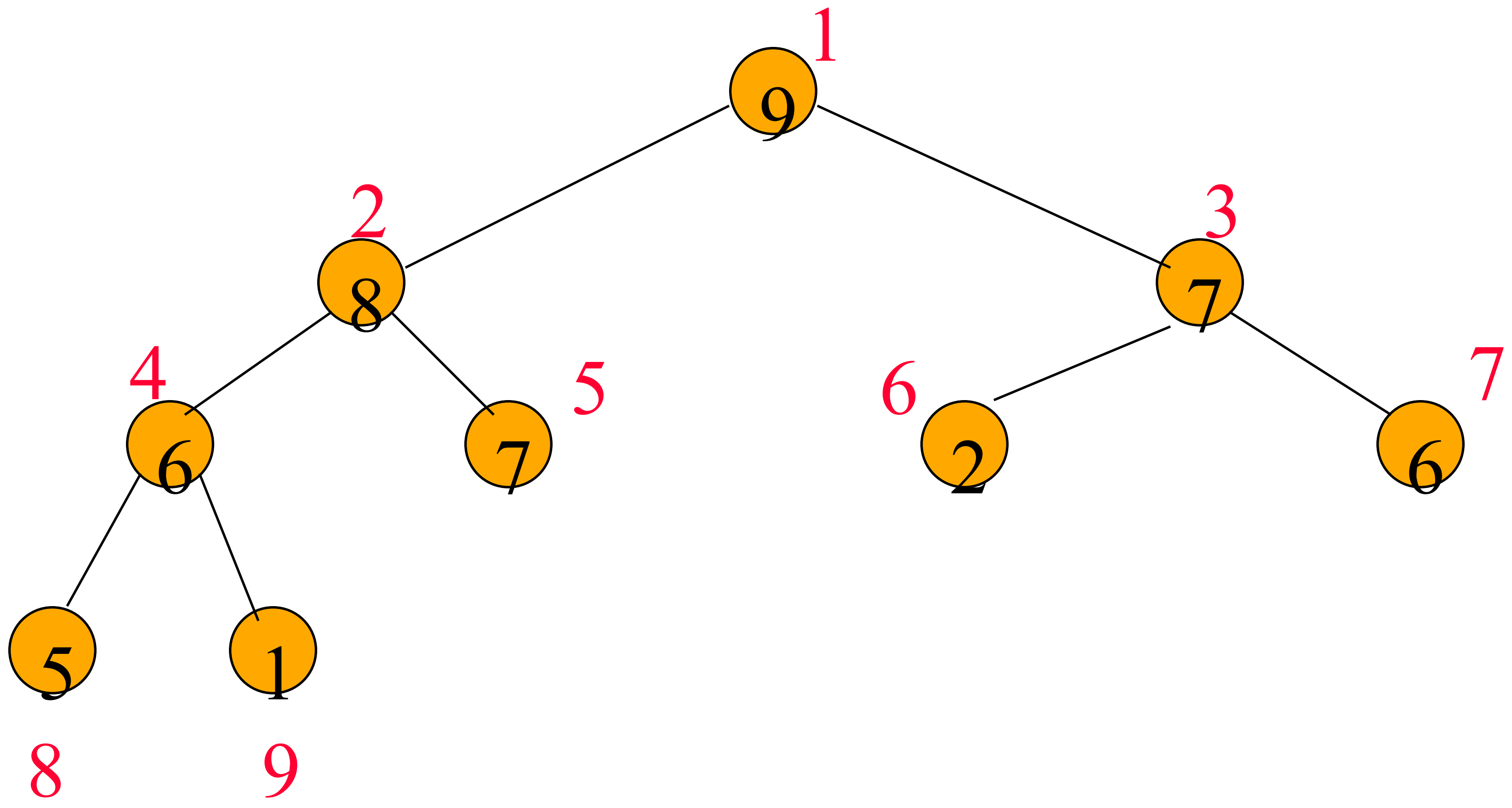
```
template <class T>
class MaxHeap: public MaxPQ <T>
{
public:
    MaxHeap (int theCapacity=10);
    bool IsEmpty () { return heapSize==0;}
    const T& Top() const;
    void Push(const T&);
    void Pop();
private:
   T* heap;              // element array
   int heapSize;           // number of elements in heap
   int capacity;         // size of the array heap
};
```

A Heap Is Efficiently Represented As An Array



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

template <class T>
Inline T& MaxHeap<T>::Top()
{
    if (IsEmpty()) throw “The heap is empty”;
    return heap[1];
}
```

Moving Up And Down A Heap

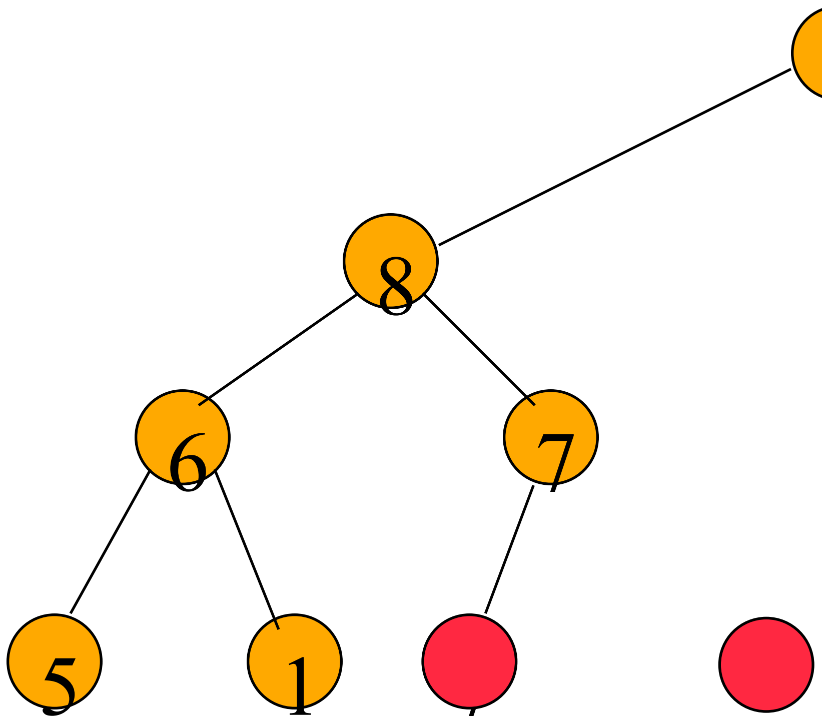


Putting An Element Into A Max Heap

Complete Binary Tree

Max Tree

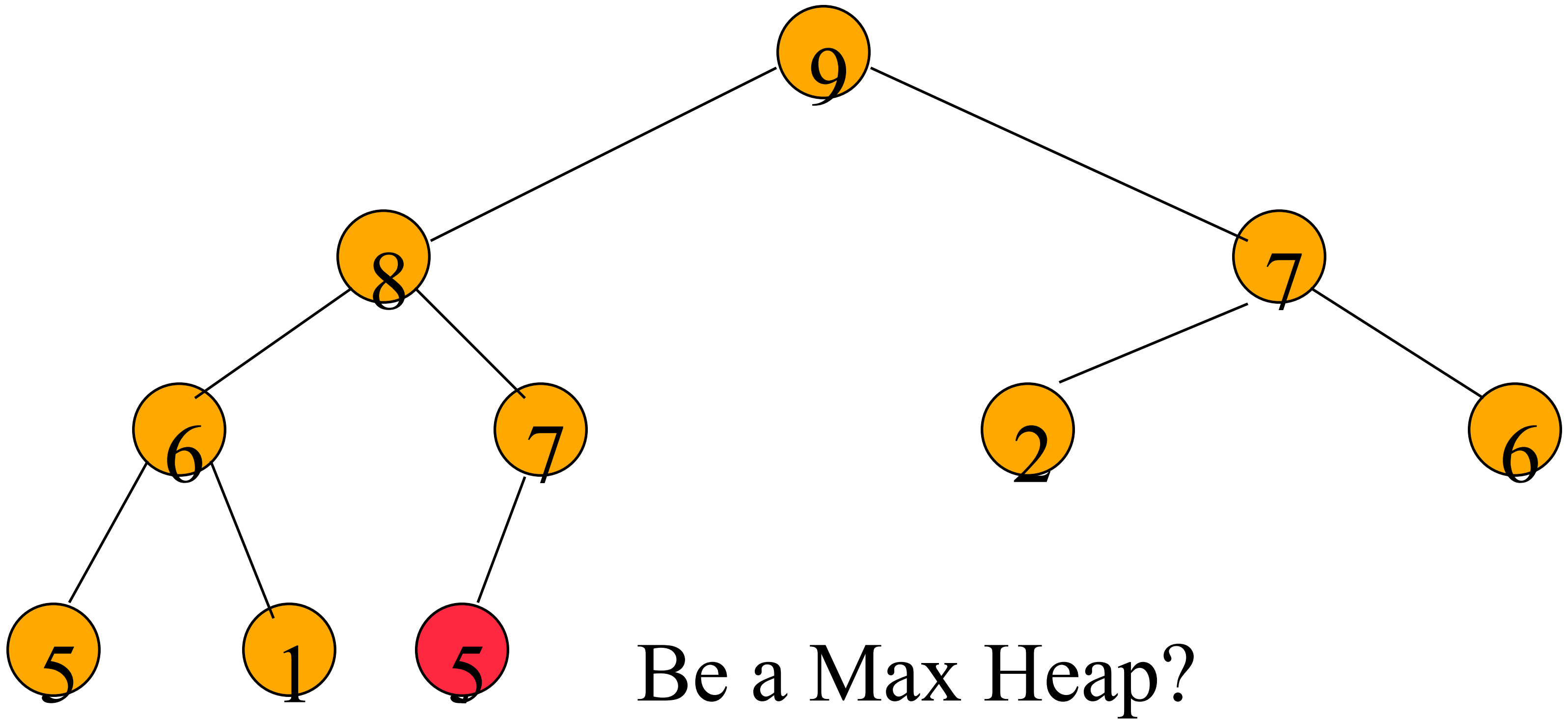
Putting An Element Into A Max Heap



Be a complete binary tree?

Complete binary tree with 10 nodes.

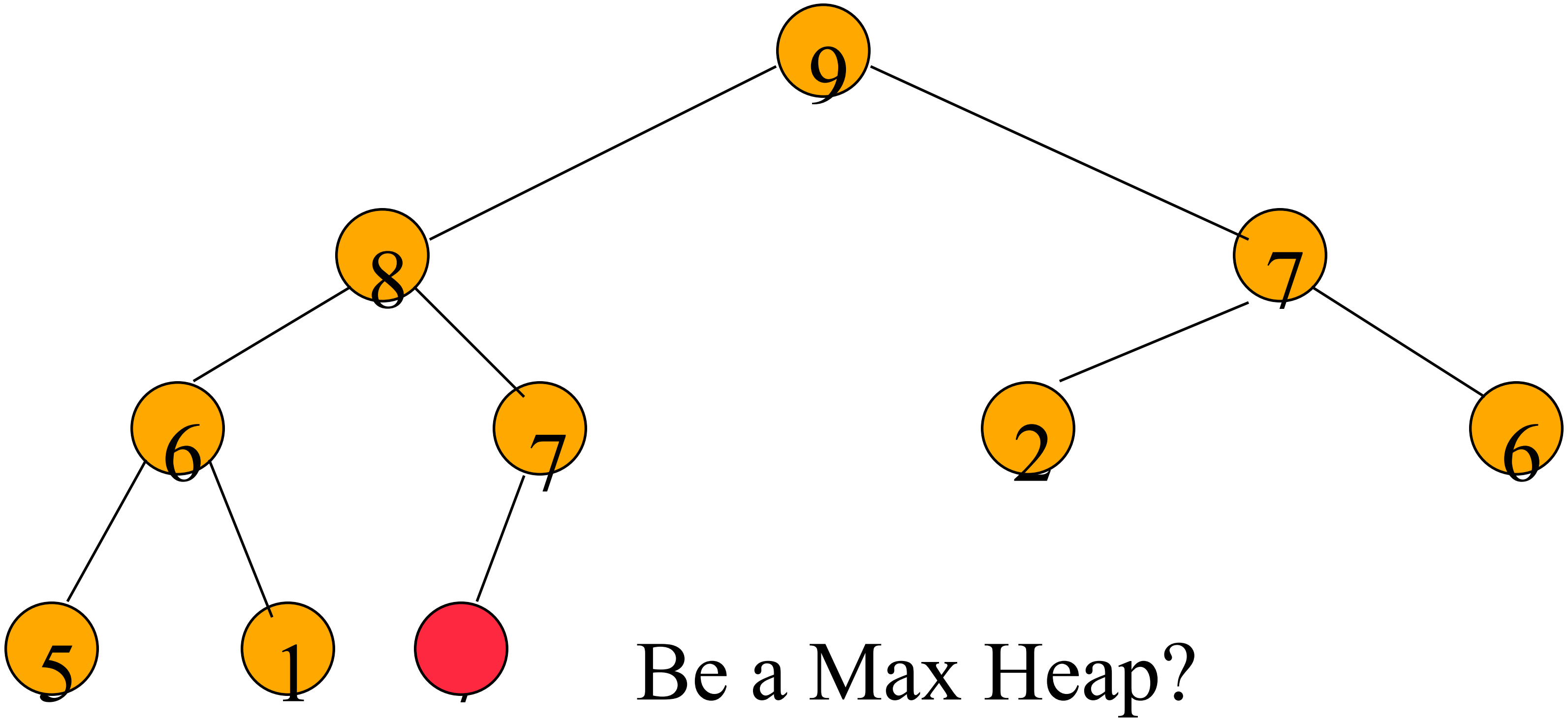
Putting An Element Into A Max Heap



Be a Max Heap?

New element is 5.

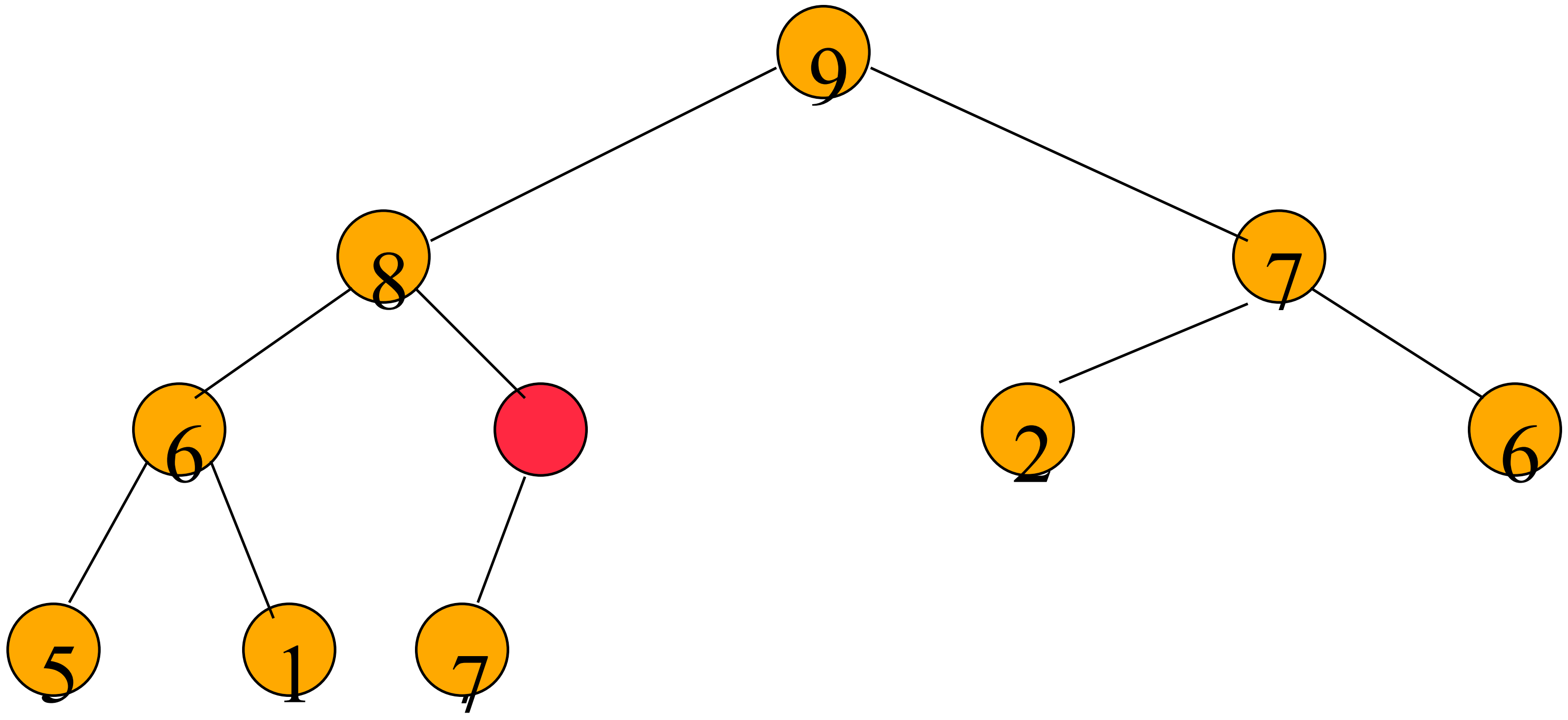
Putting An Element Into A Max Heap



Be a Max Heap?

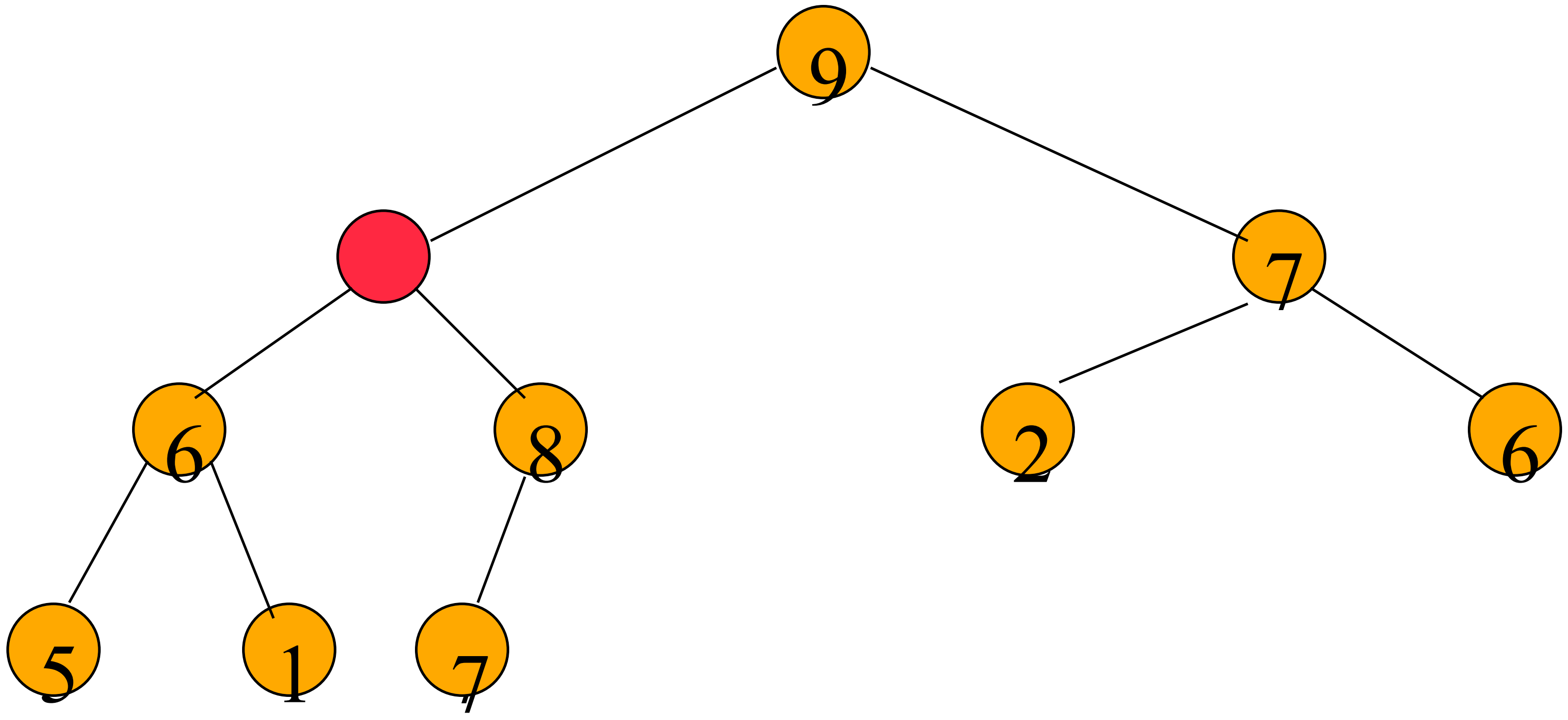
New element is 20.

Putting An Element Into A Max Heap



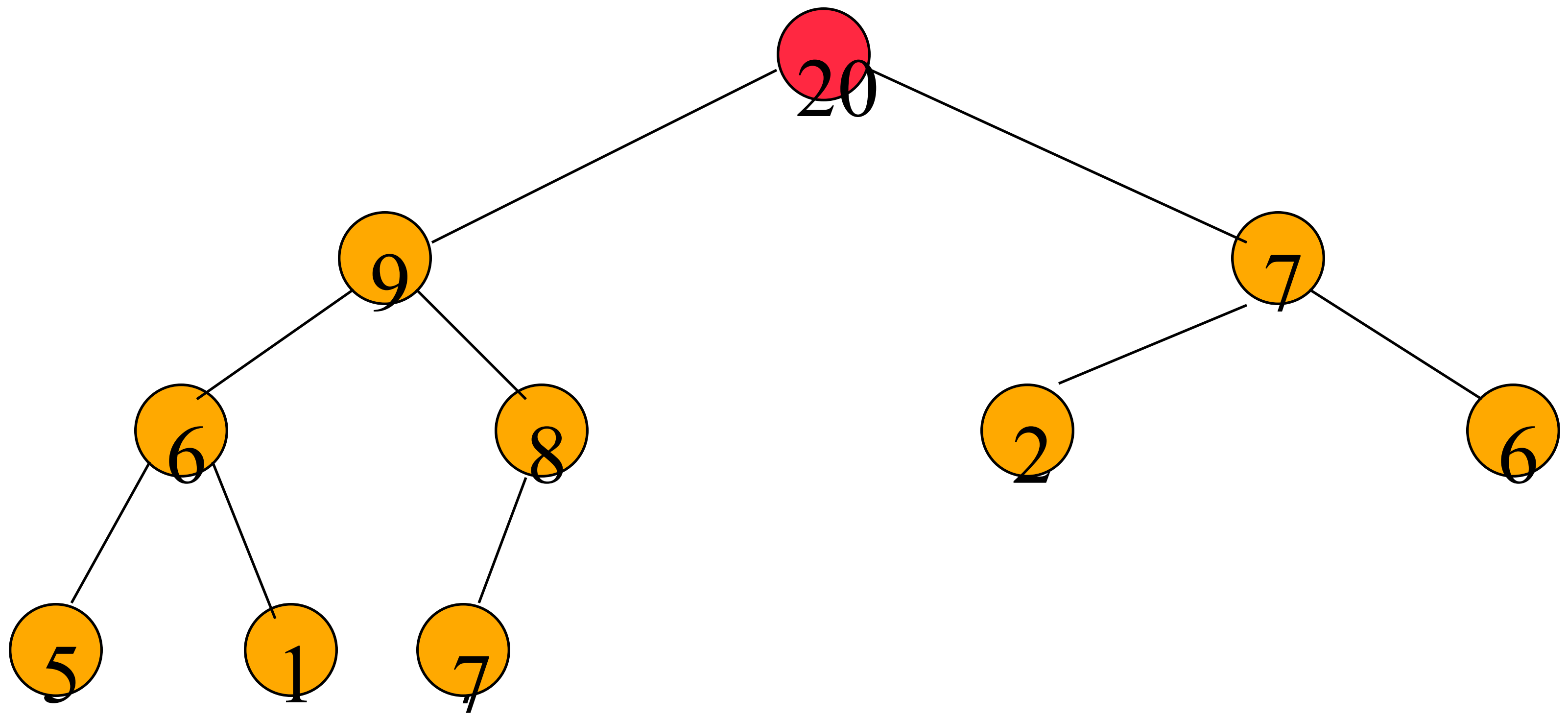
New element is 20.

Putting An Element Into A Max Heap



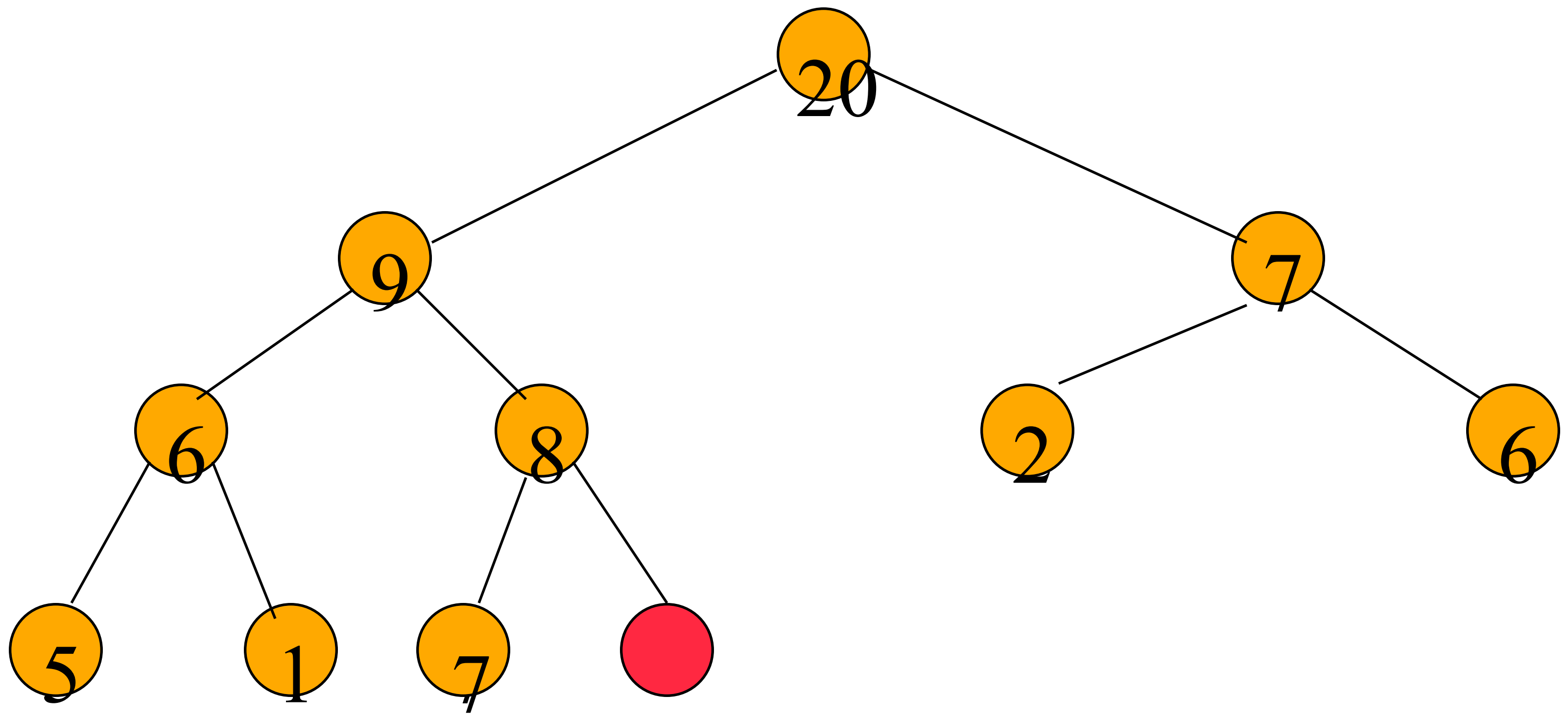
New element is 20.

Putting An Element Into A Max Heap



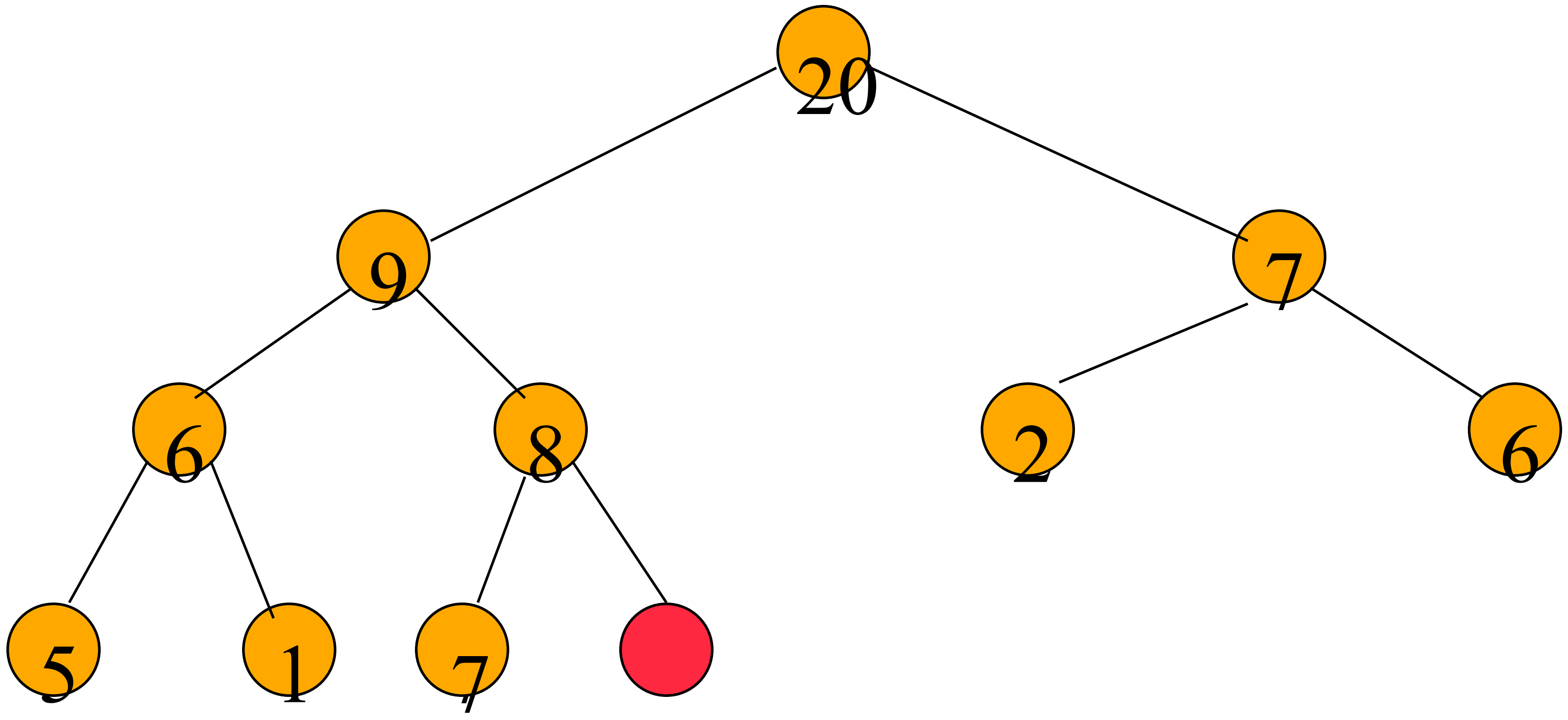
New element is 20.

Putting An Element Into A Max Heap



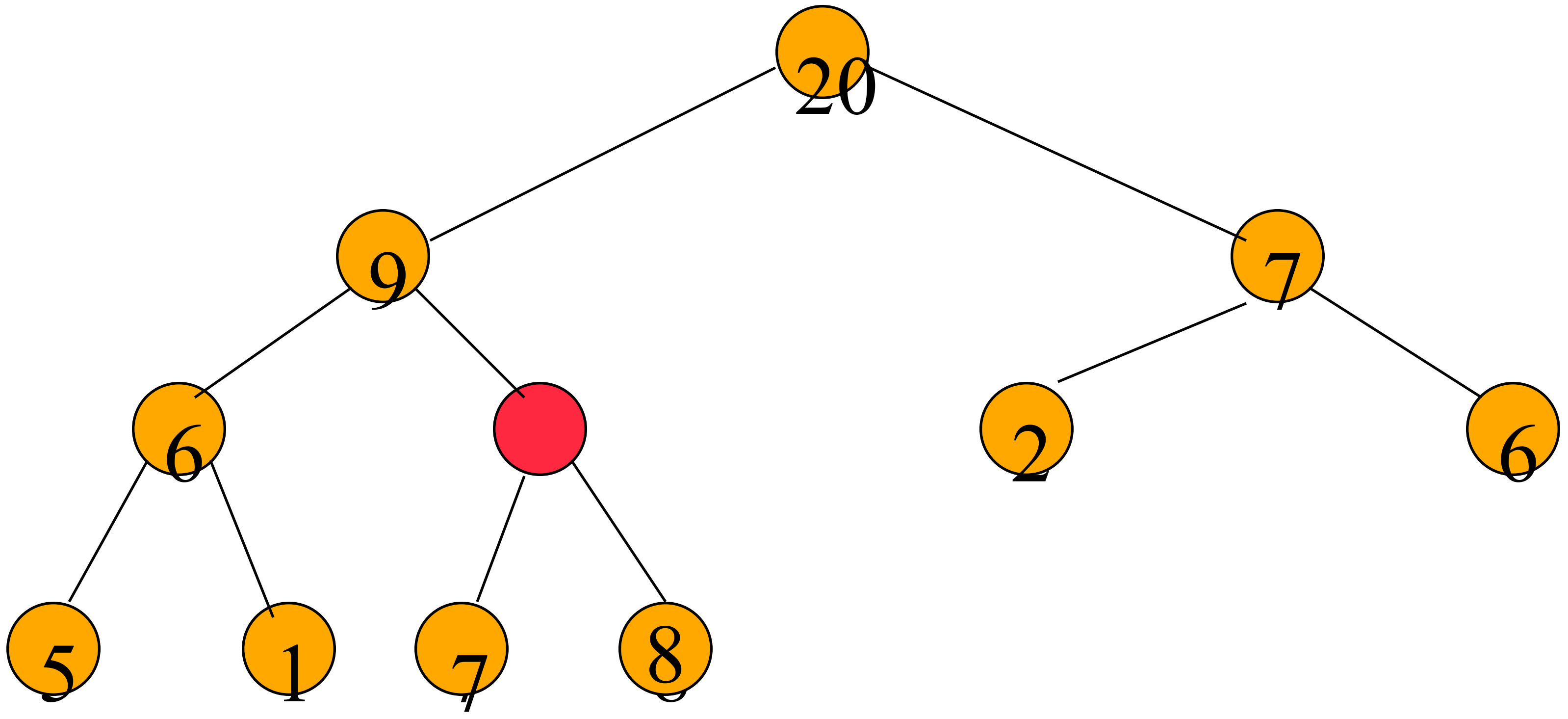
Complete binary tree with 11 nodes.

Putting An Element Into A Max Heap



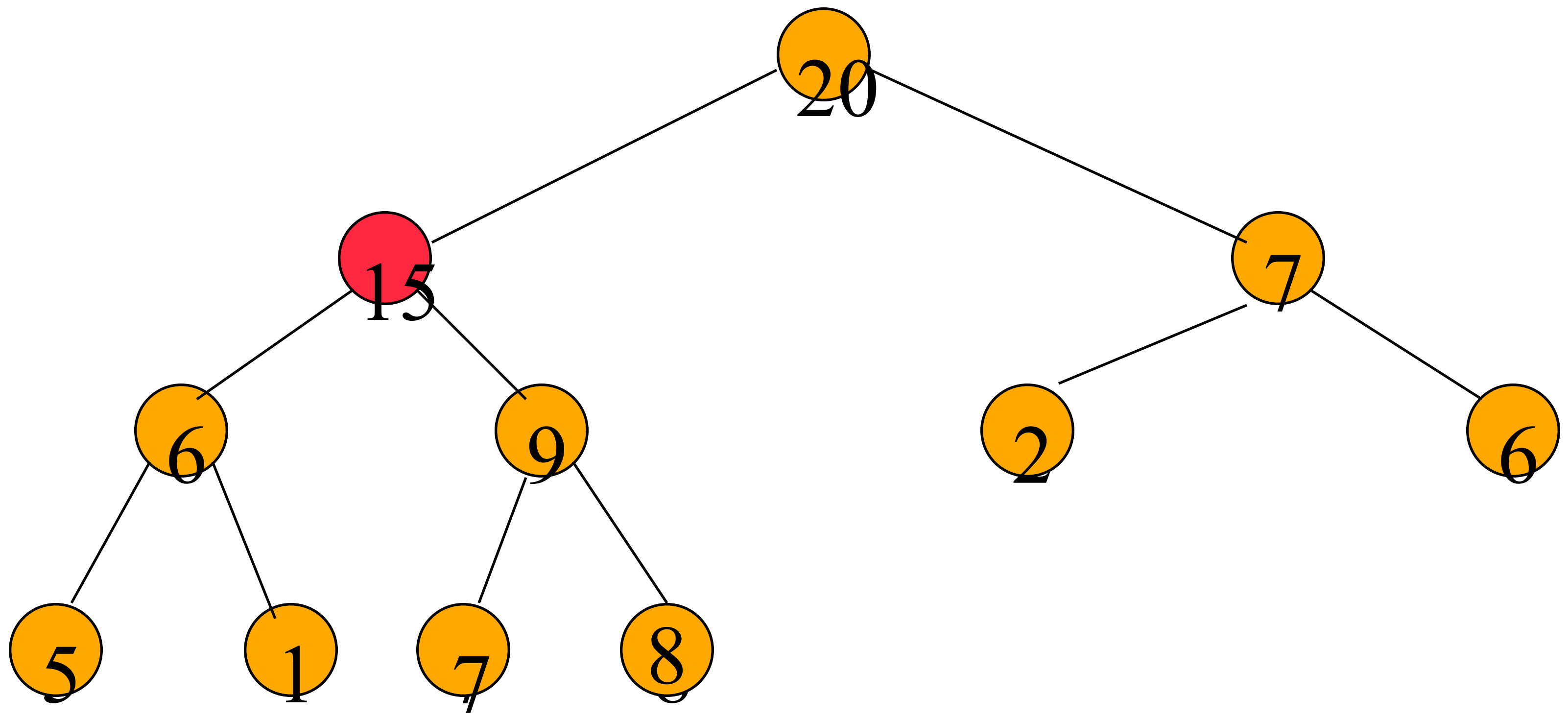
New element is 15.

Putting An Element Into A Max Heap



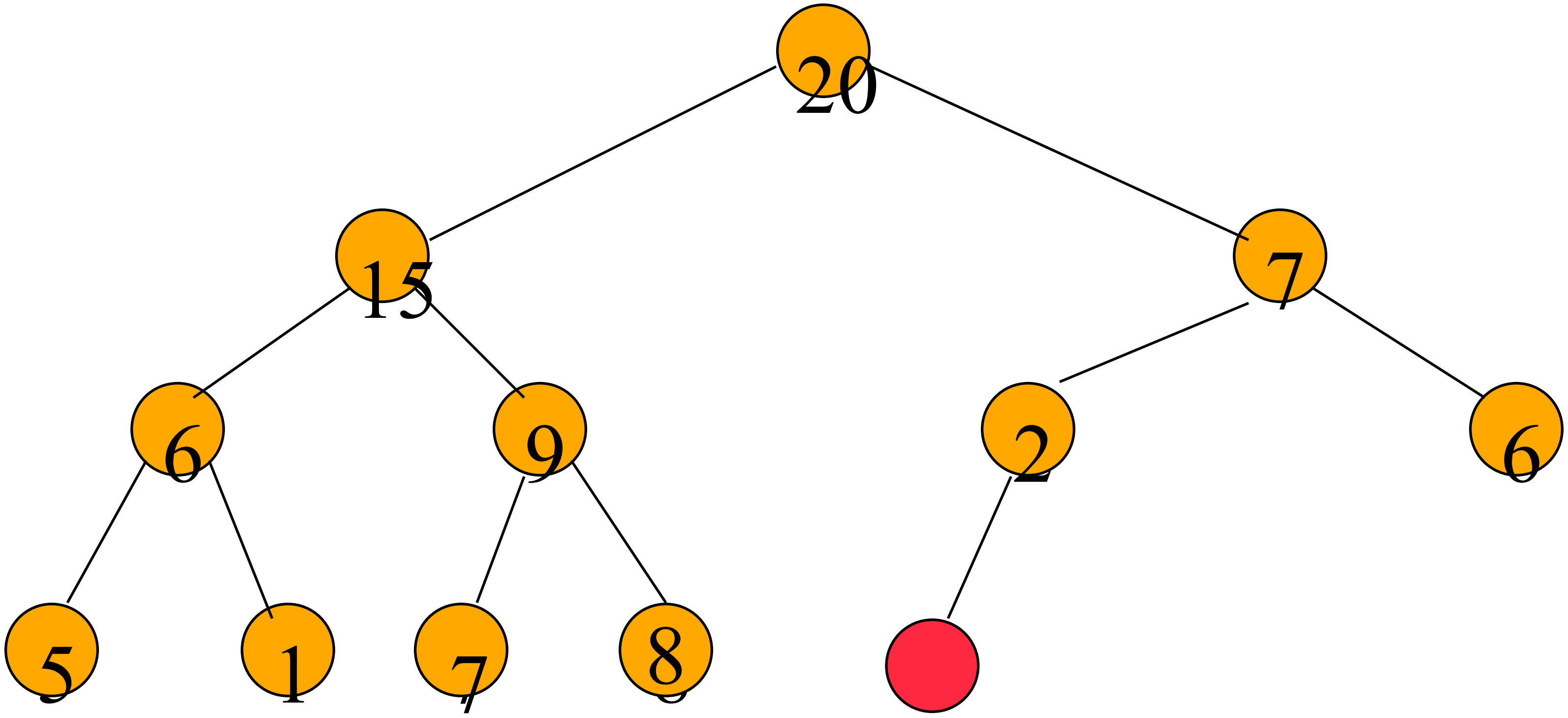
New element is 15.

Putting An Element Into A Max Heap



New element is 15.

Complexity Of Put



Complexity is O(log n), where n is heap size.

```
template <class T>
void MaxHeap<Type>::Push(const T& e)
{ // insert e into the max heap
    if (heapSize == capacity)  {  // double the capacity
        ChangeSize1D(heap, capacity, 2*capacity);
        capacity *= 2;
    }
    int currentNode = ++heapSize;
    while (currentNode != 1 && heap[currentNode/2] < e)
    {  //  bubble up
        heap[currentNode] = heap[currentNode/2];
                currentNode /=2;
        }
       heap[currentNode] = e;
}
```

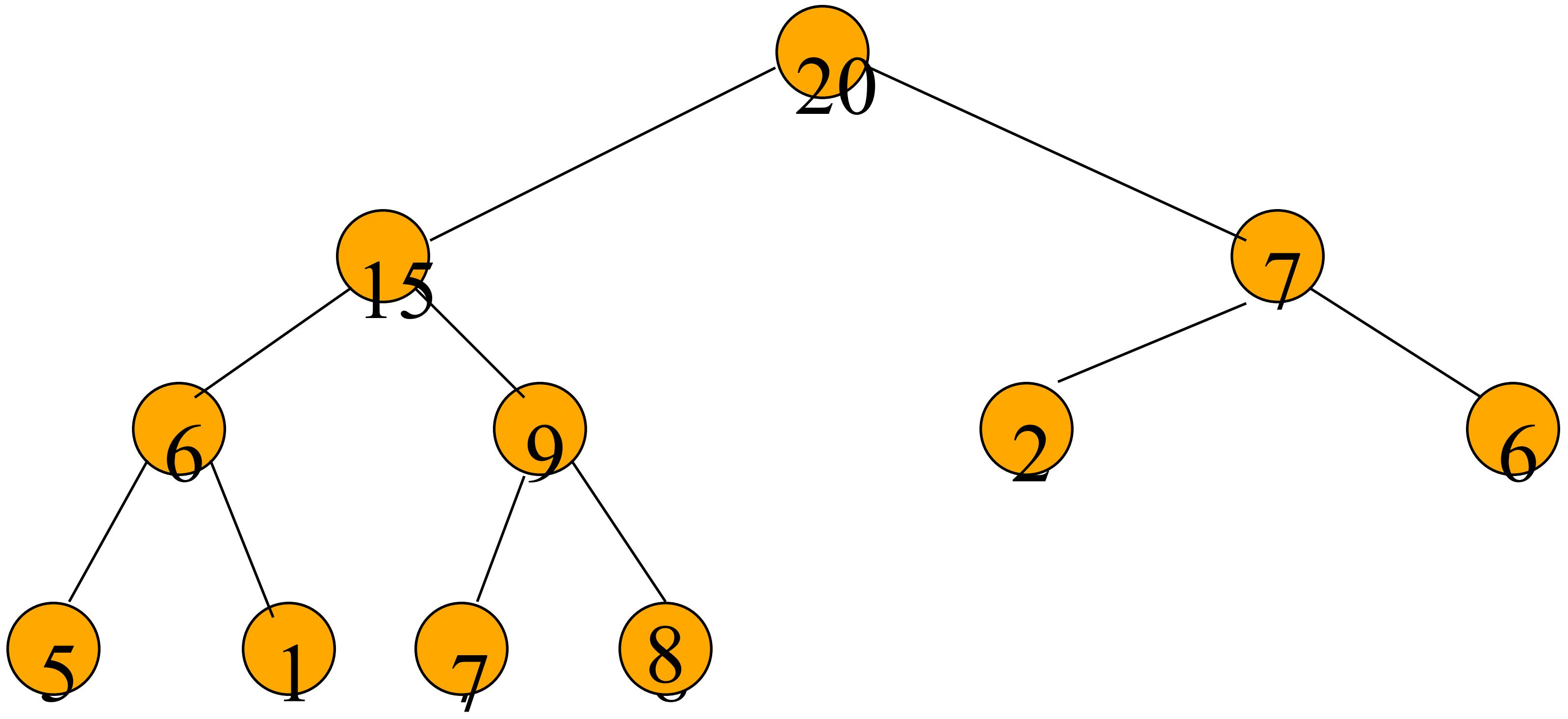
O(log n)

Removing The Max Element

Complete Binary Tree

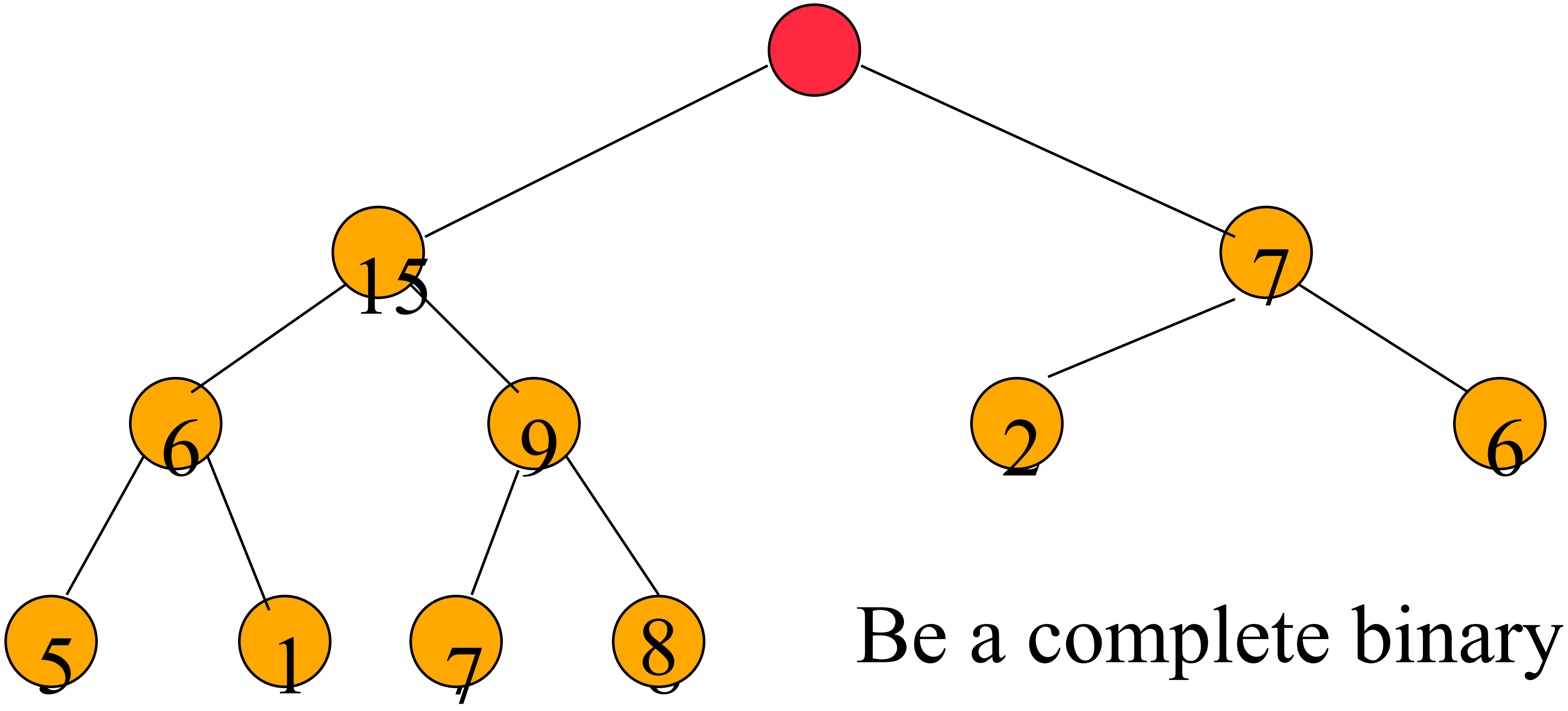
Max Tree

Removing The Max Element



Max element is in the root.

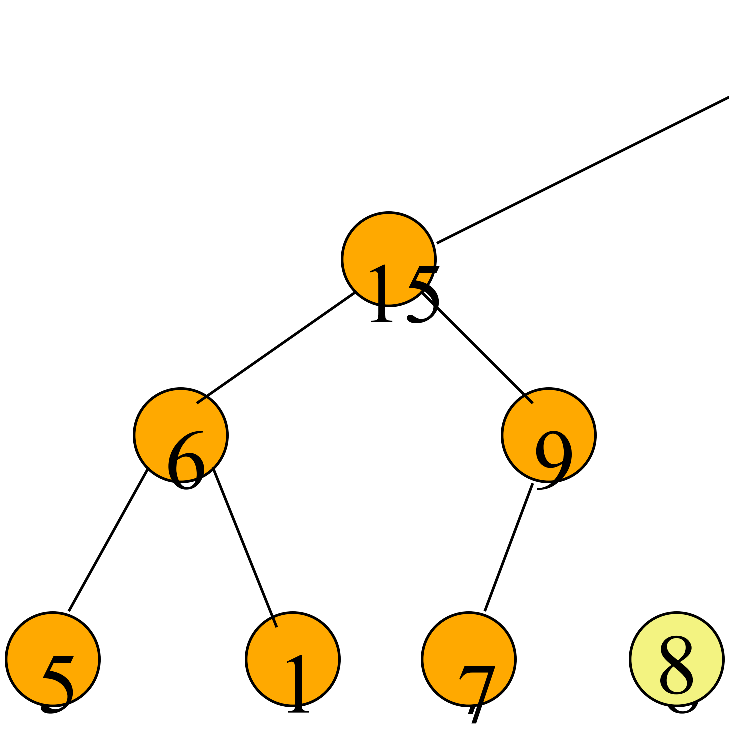
Removing The Max Element



Be a complete binary tree?

After max element is removed.

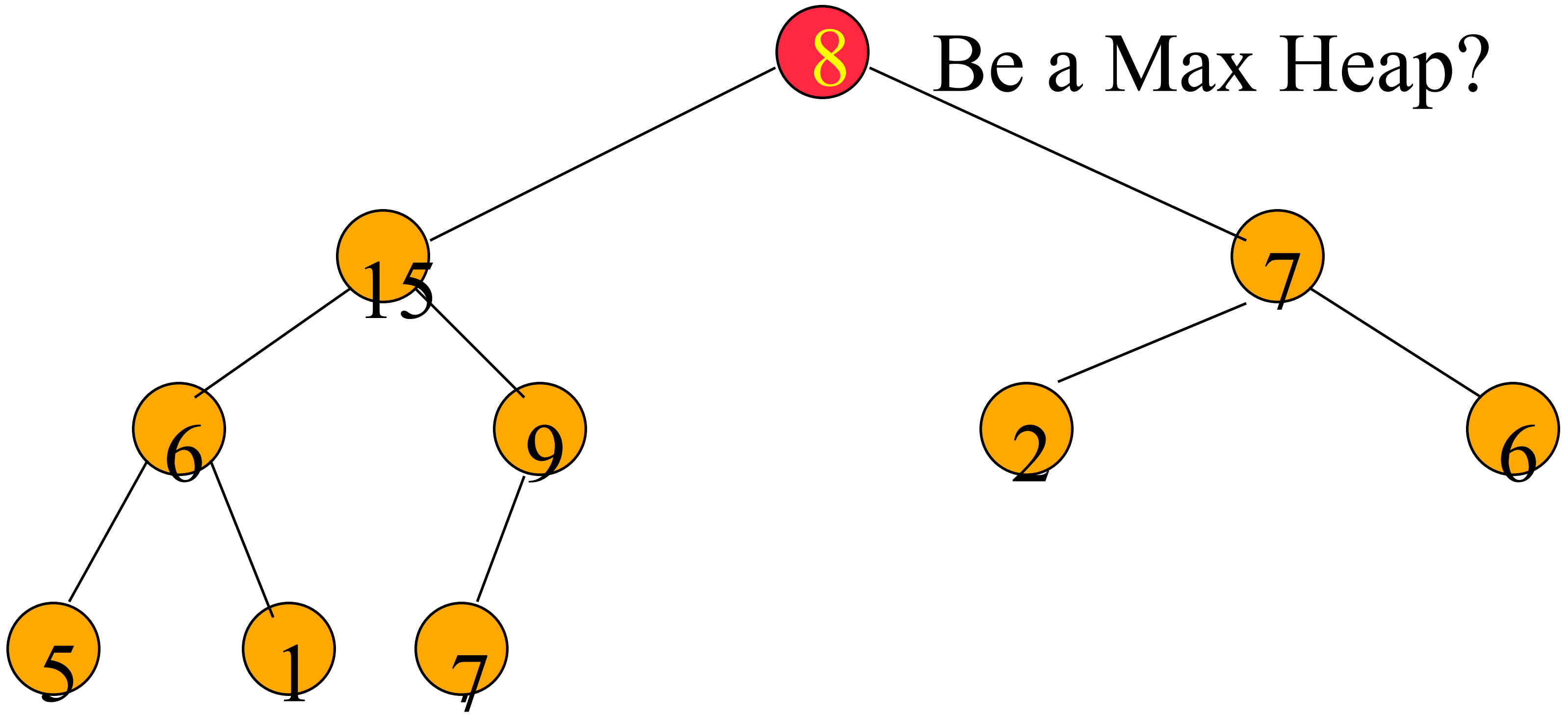
Removing The Max Element



Heap with 10 nodes.

Reinsert 8 into the heap.

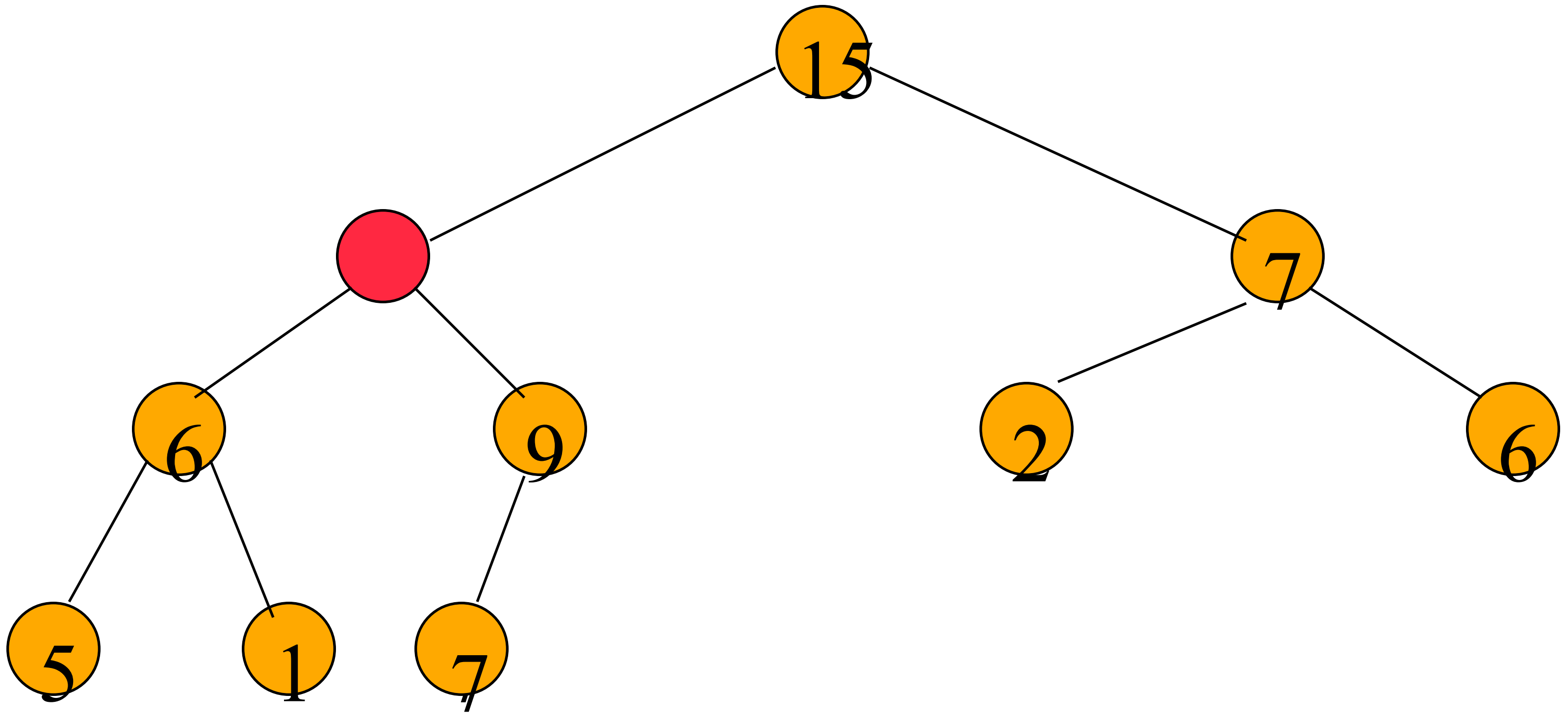
Removing The Max Element



Be a Max Heap?

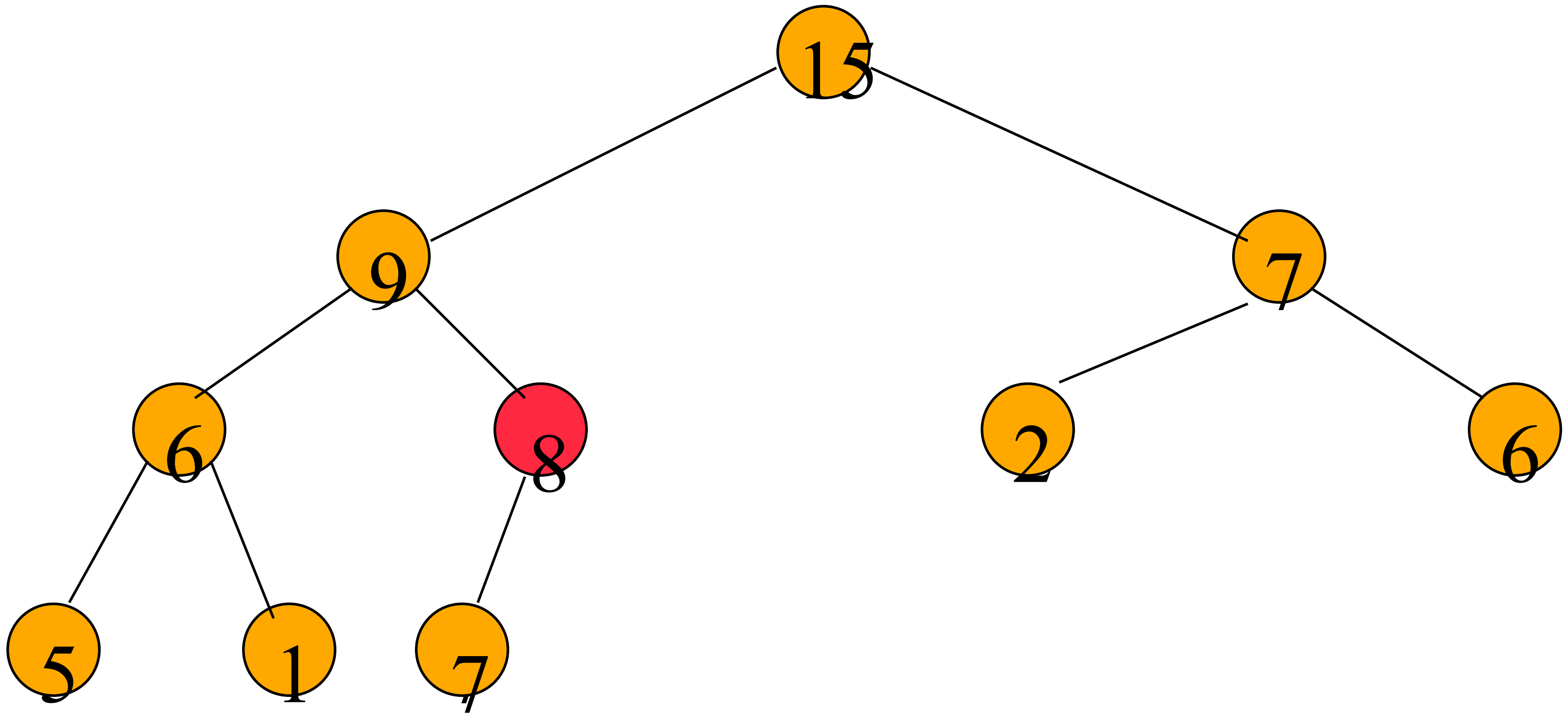
Reinsert 8 into the heap.

Removing The Max Element



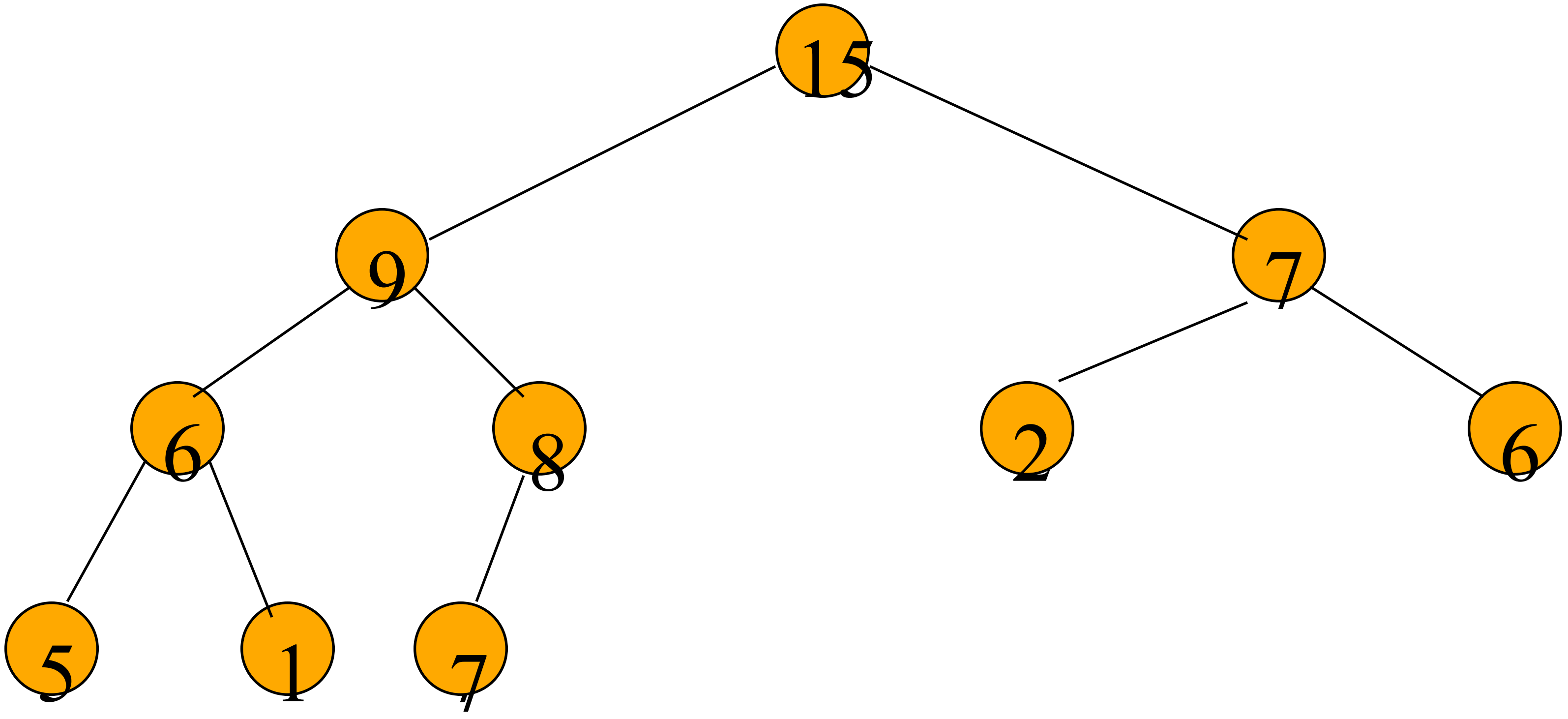
Reinsert 8 into the heap.

Removing The Max Element



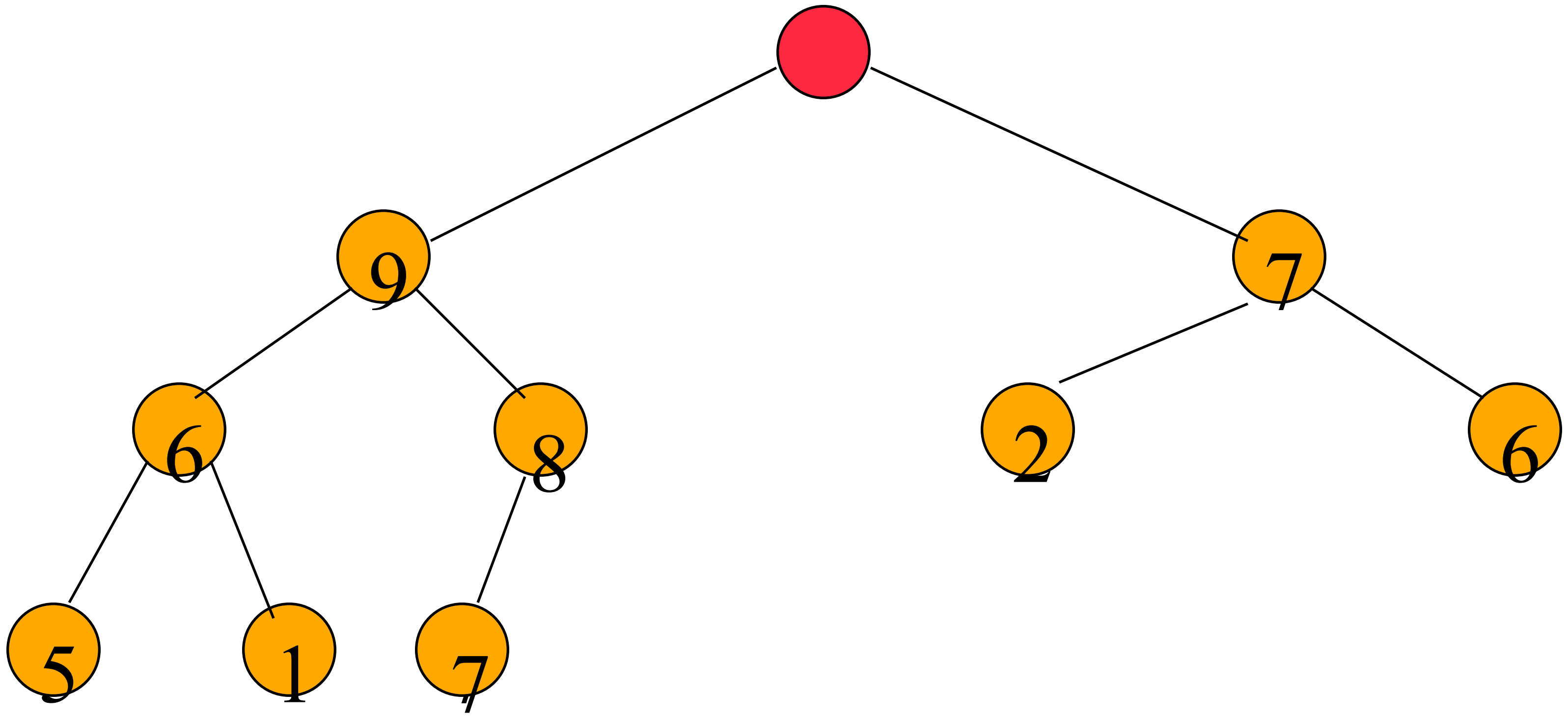
Reinsert 8 into the heap.

Removing The Max Element



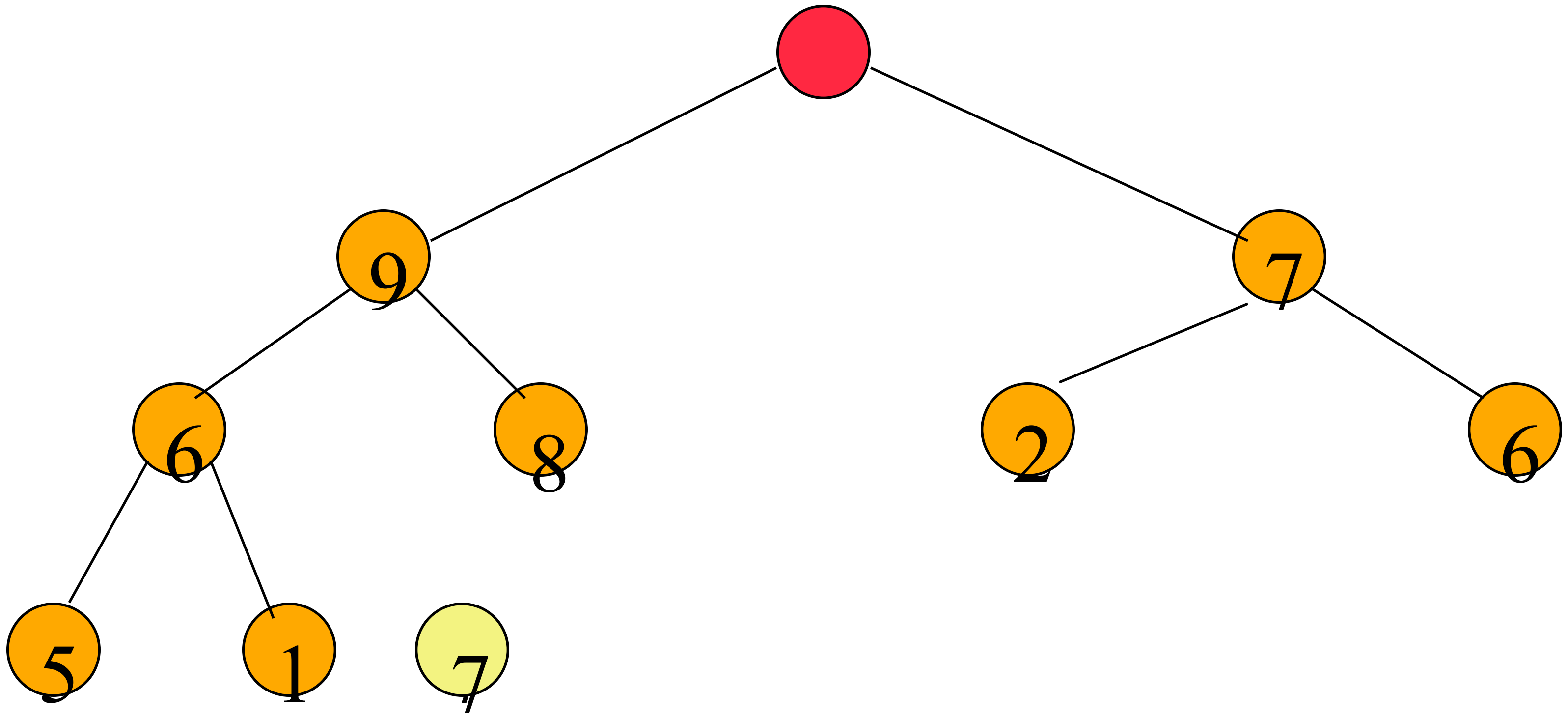
Max element is 15.

Removing The Max Element



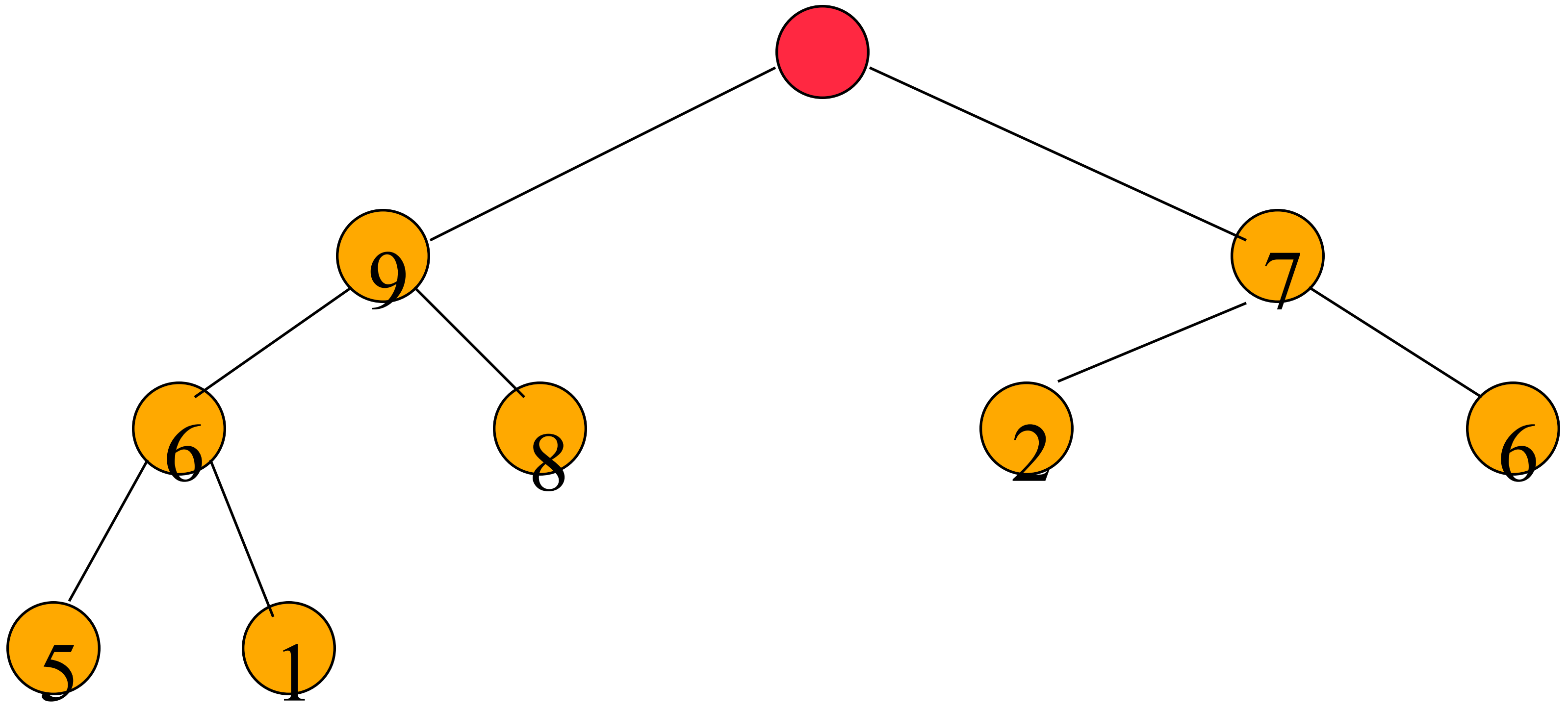
After max element is removed.

Removing The Max Element



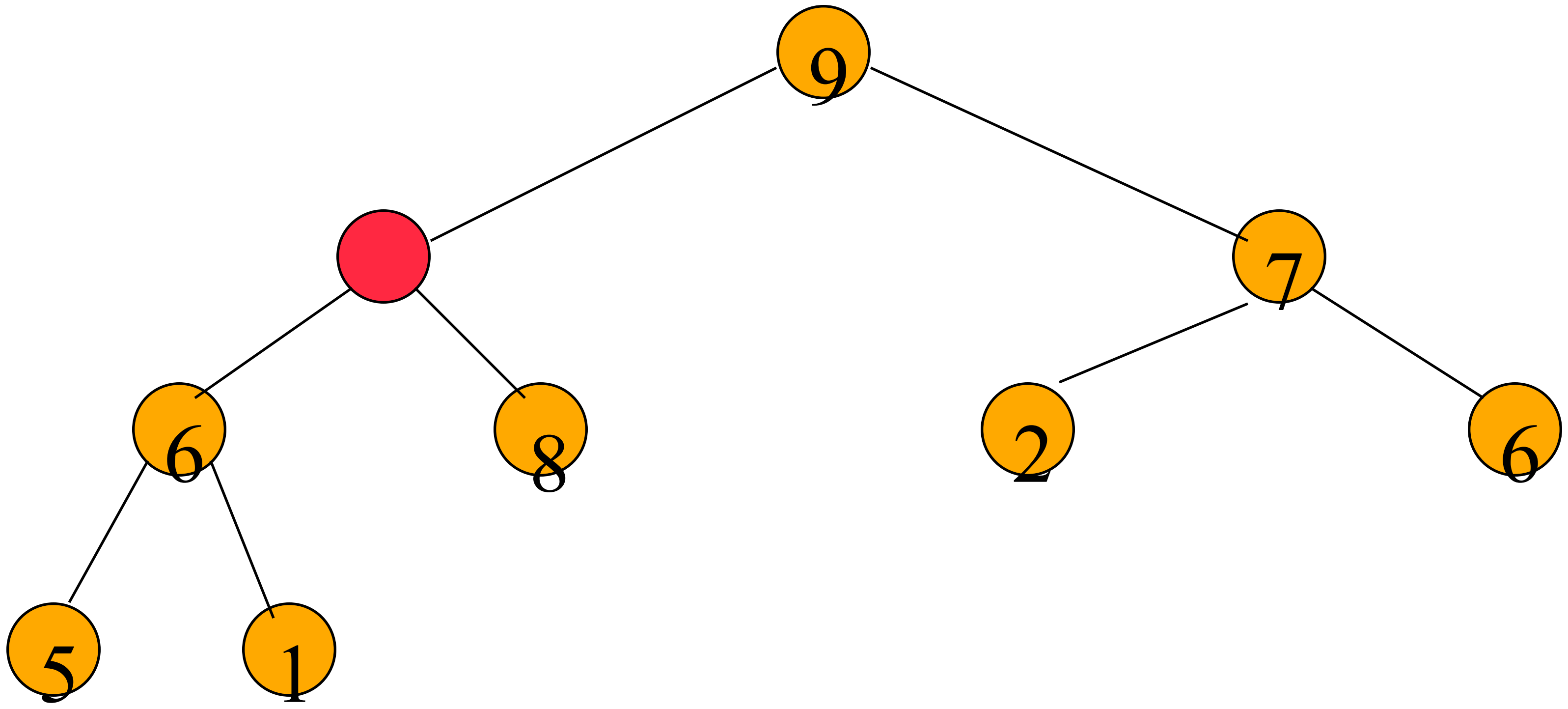
Heap with 9 nodes.

Removing The Max Element



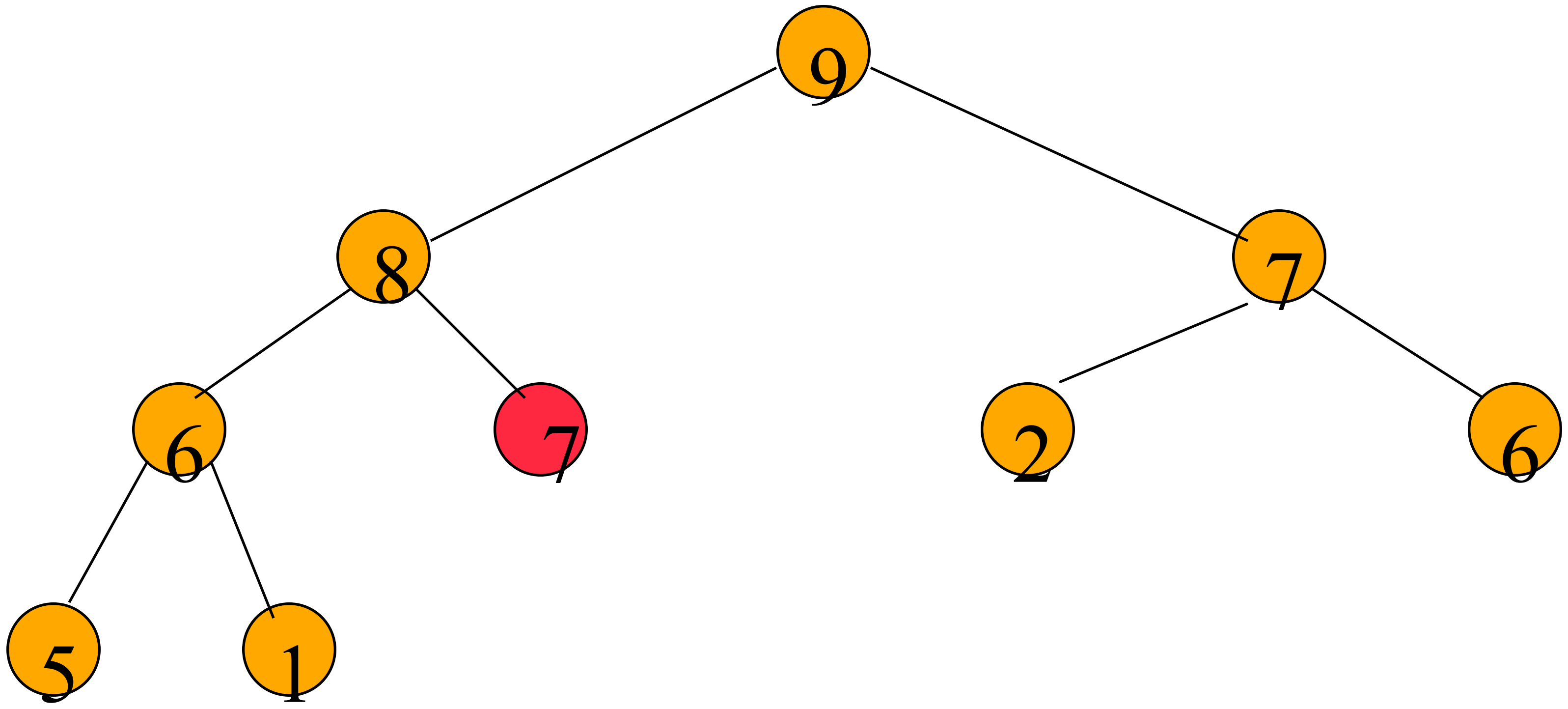
Reinsert 7.

Removing The Max Element



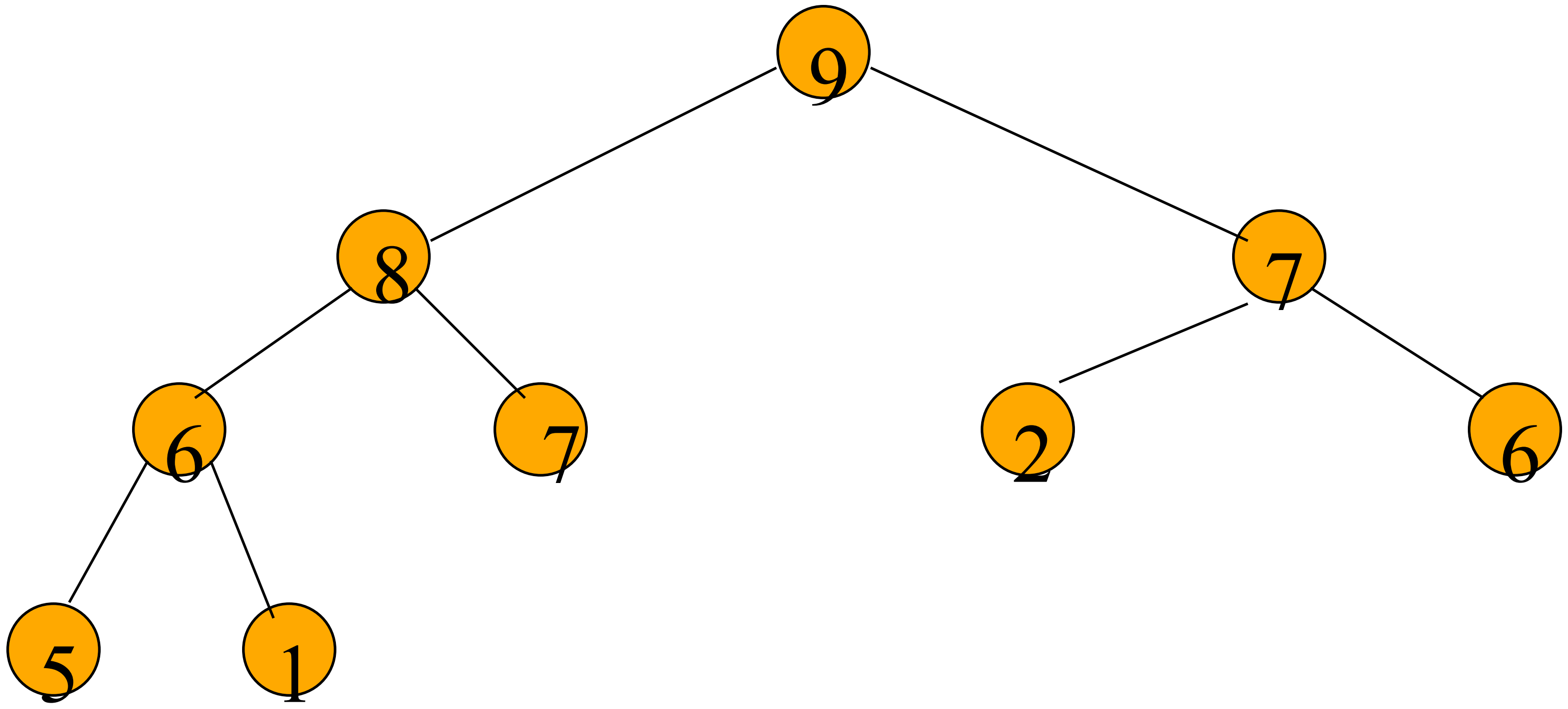
Reinsert 7.

Removing The Max Element



Reinsert 7.

Complexity Of Remove Max Element



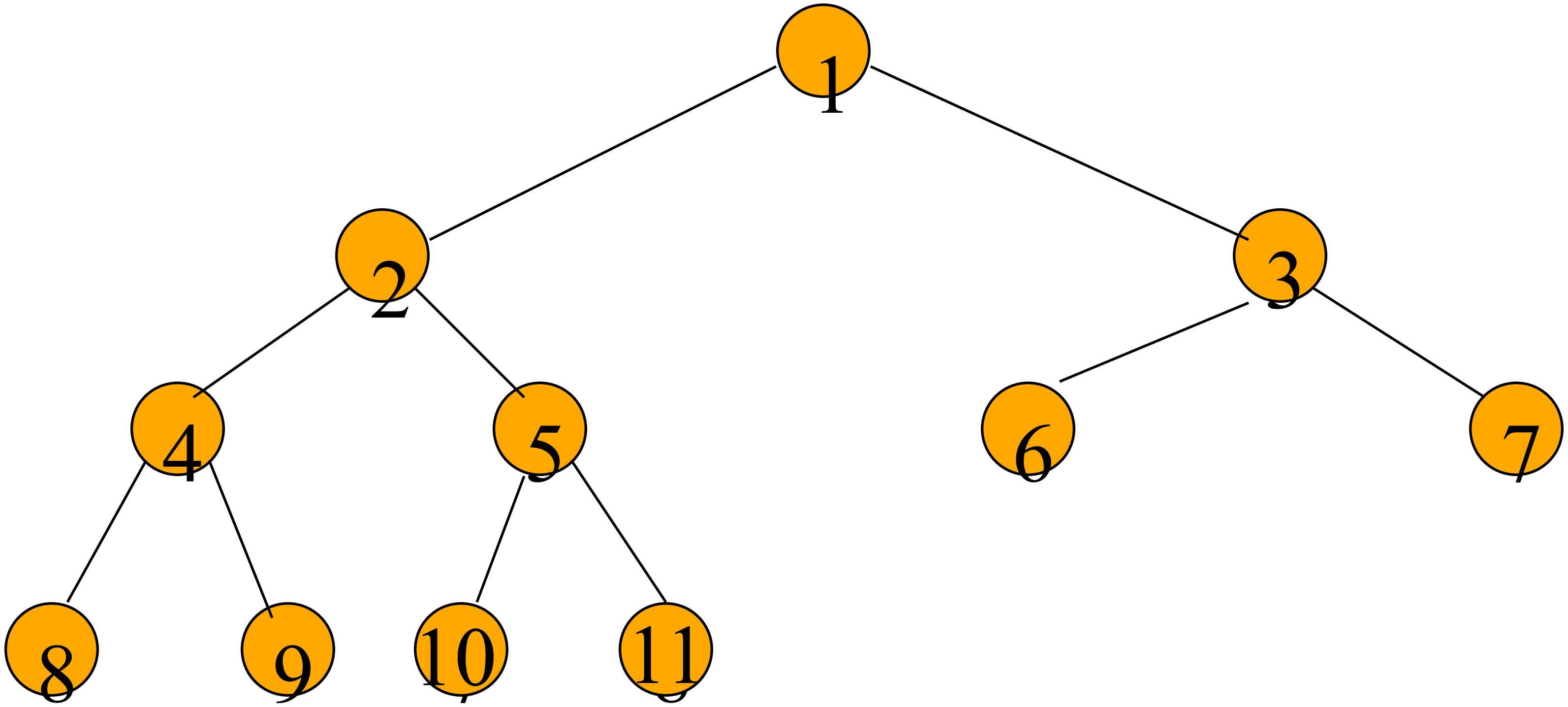
Complexity is O(log n).

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

   // remove the last element from heap
   T lastE = heap[heapSize--];

   // trickle down
   int currentNode = 1;  // root
   int child = 2;        // left child of currentNode
```

```
  while (child <= heapSize)
  {
    //  set child to the larger child of currentNode
    if (child<heapSize && heap[child]<heap[child+1])
child++;

        // can we put lastE in currentNode?
    if (lastE>=heap[child]) break;  // yes

    // no
    heap[currentNode]=heap[child]; //  move child up
    currentNode=child; child*=2; // move down a level
      }
    heap[currentNode]=lastE;
}
```

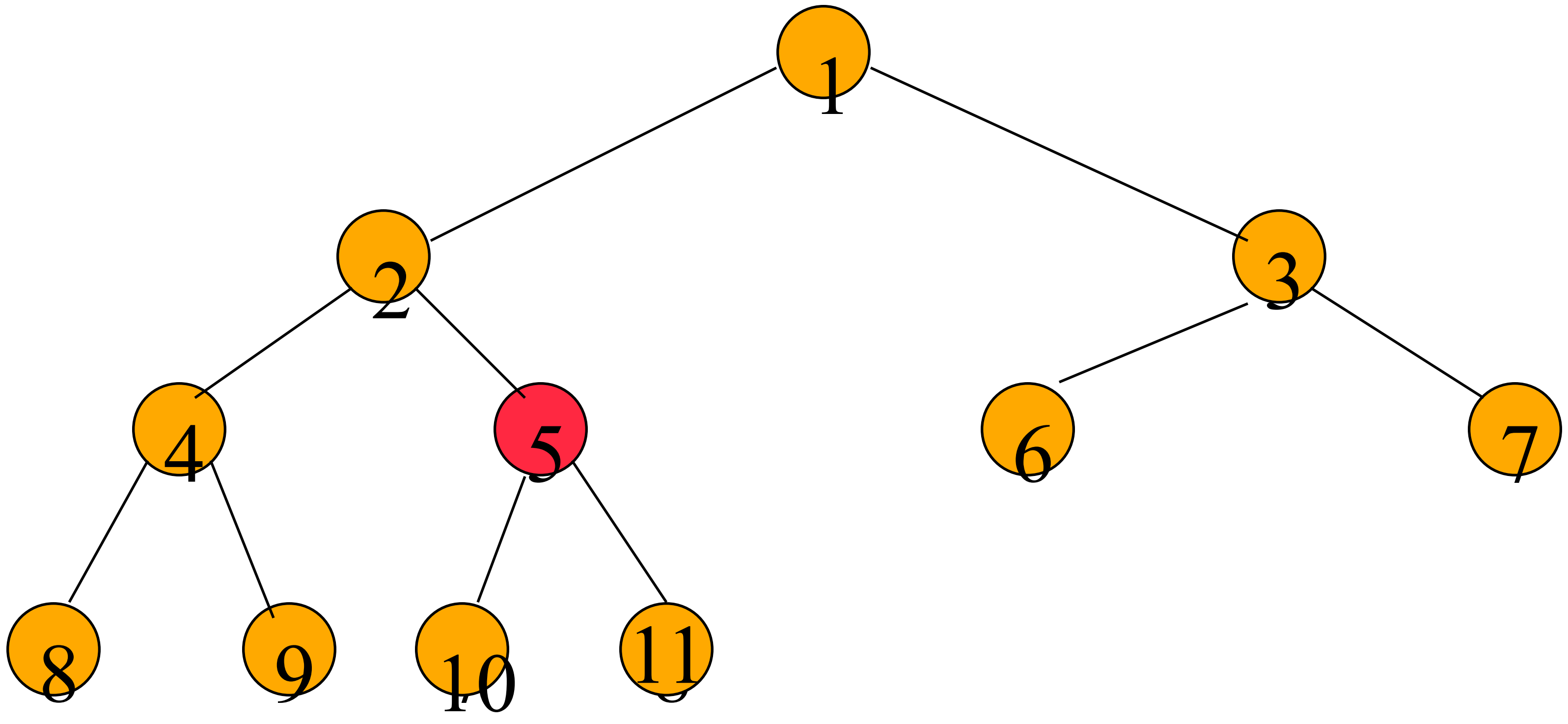
- **Exercises: P287-2, 3**

Initializing A Max Heap



input array = [-, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11]

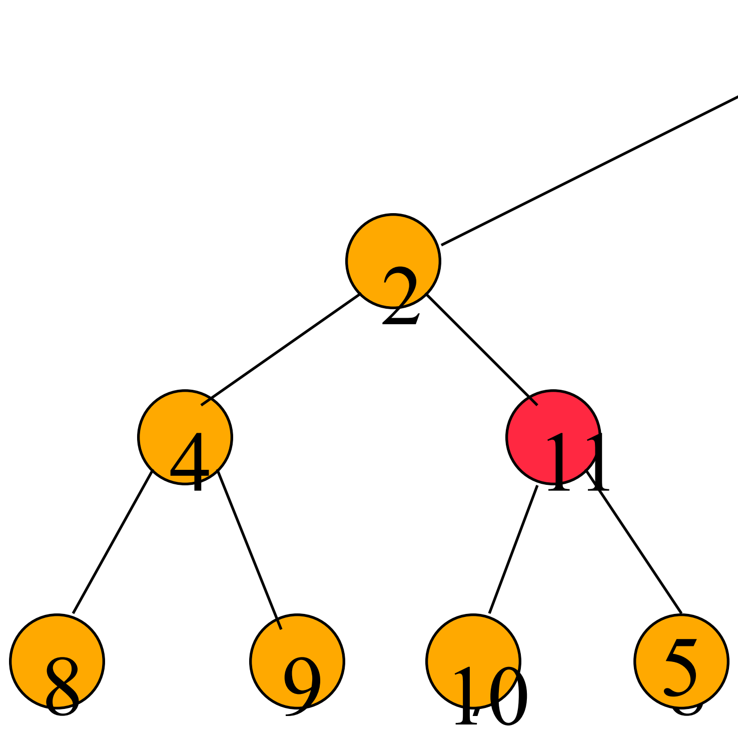
Initializing A Max Heap



Start at rightmost array position that has a child.

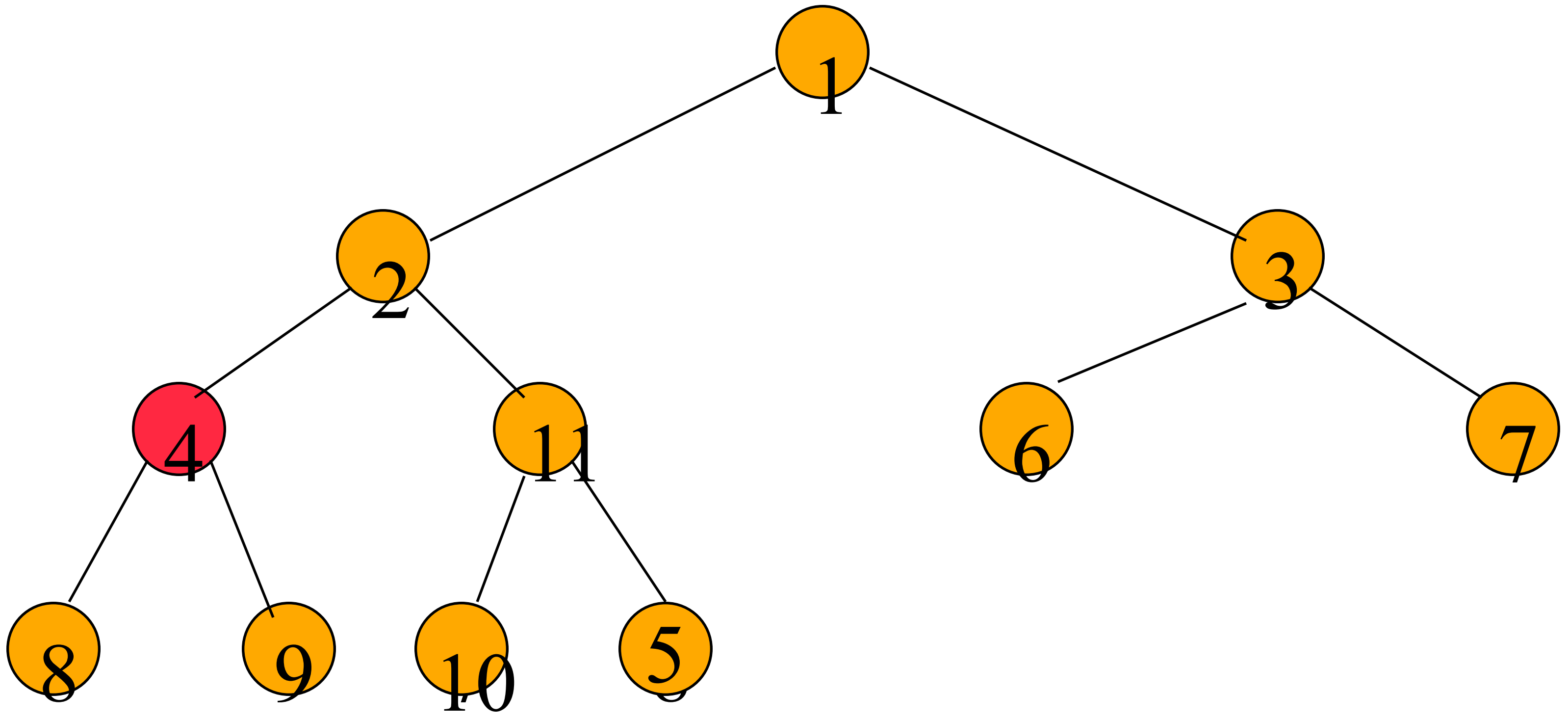
Index is n/2.

Initializing A Max Heap

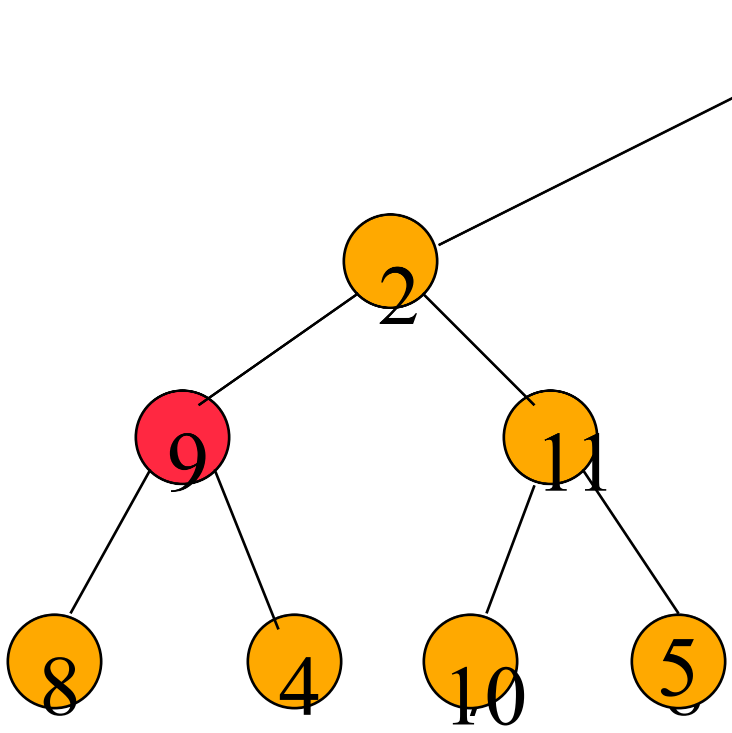


Move to next lower array position.

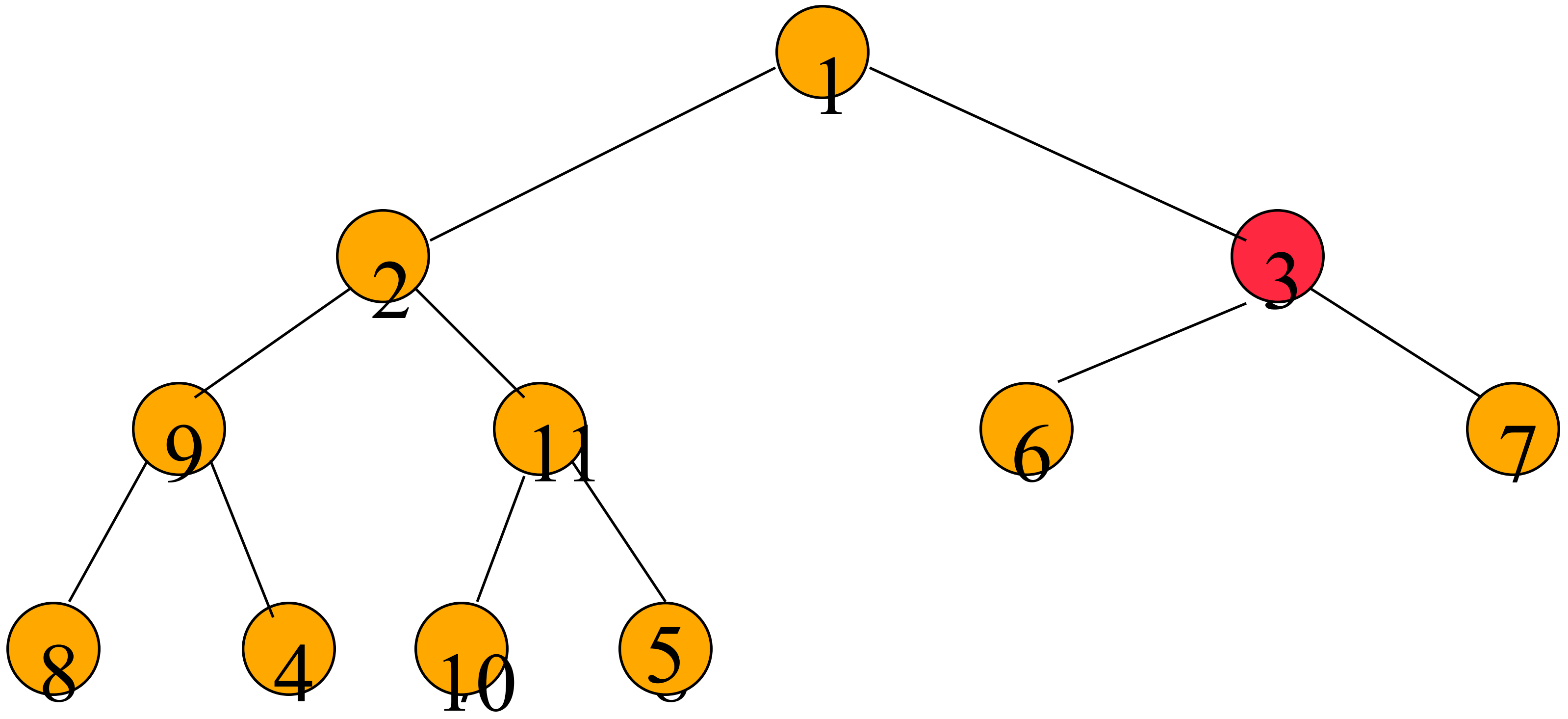
Initializing A Max Heap



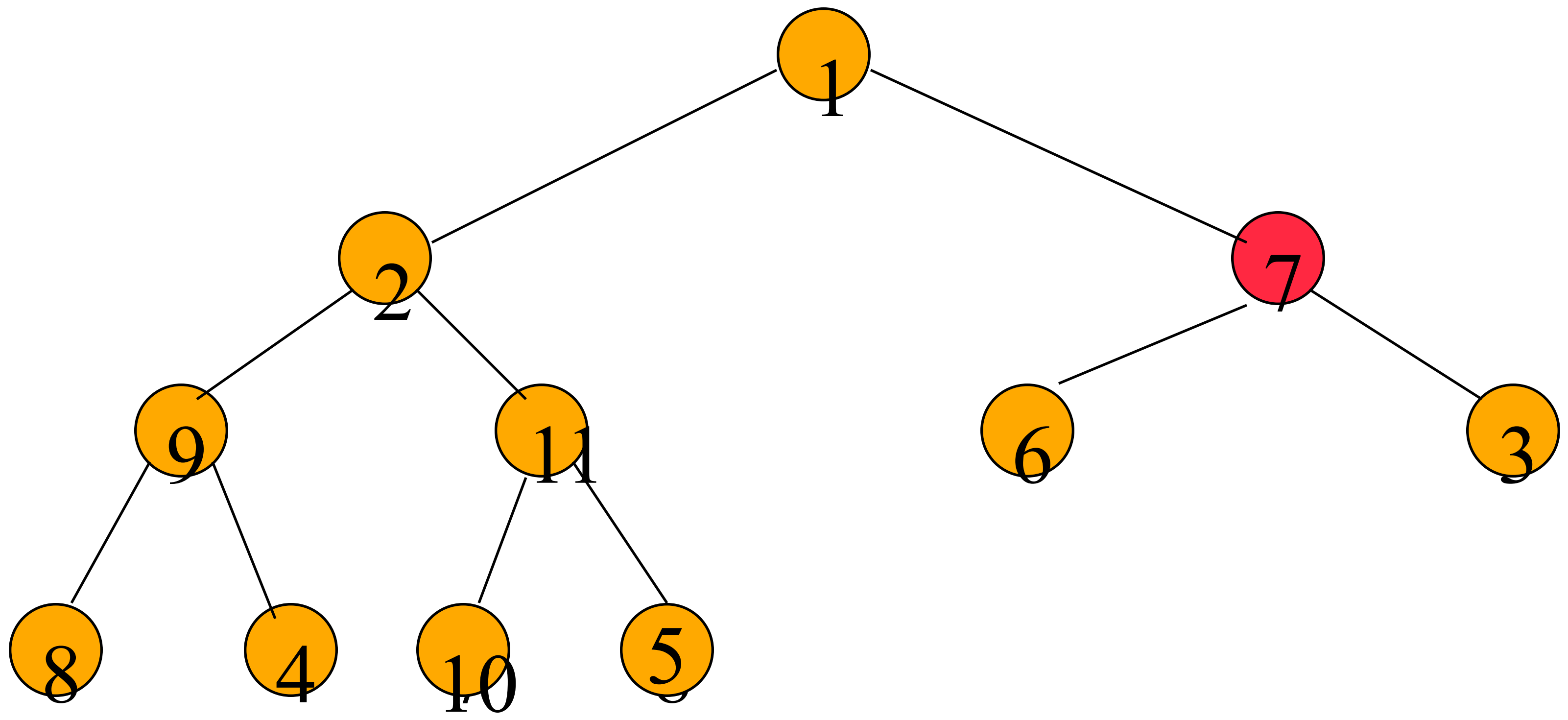
Initializing A Max Heap



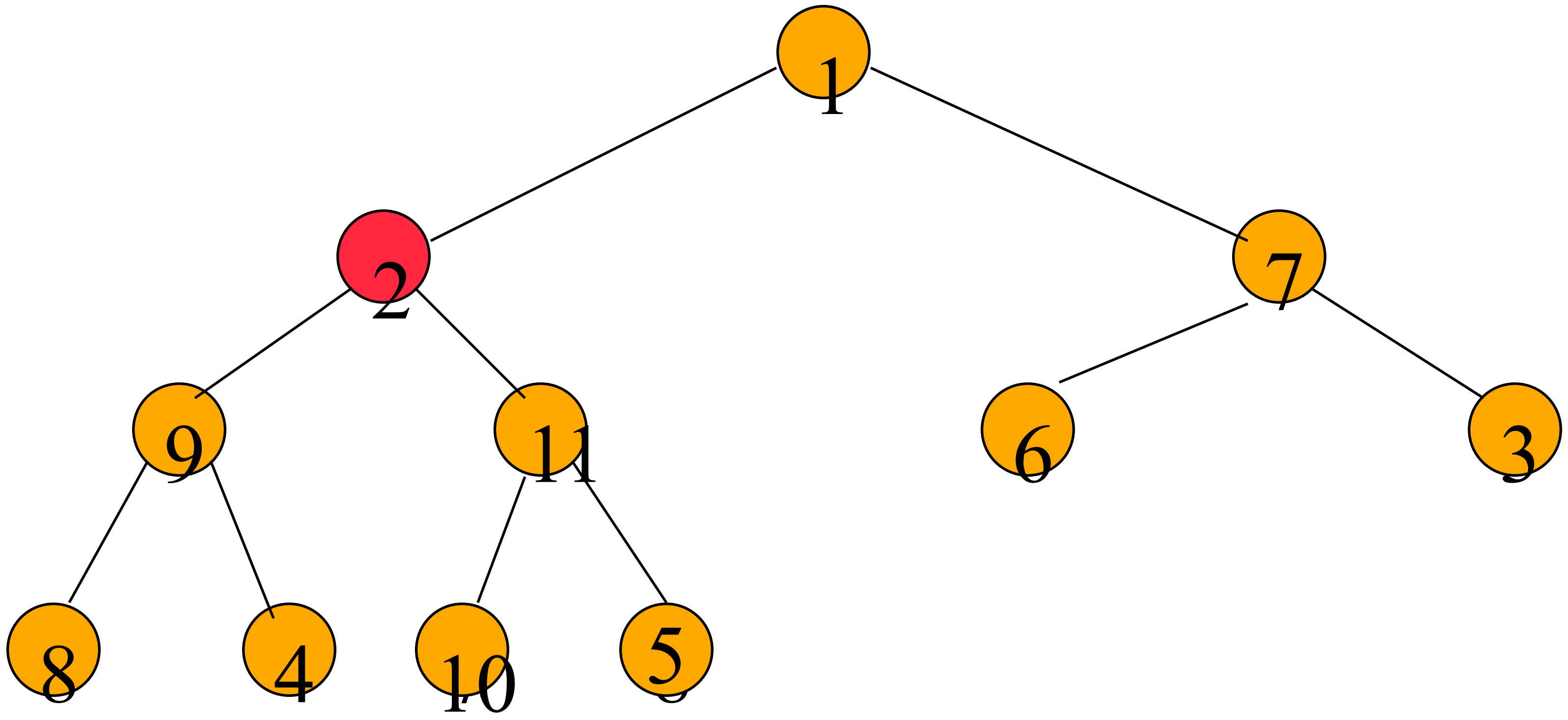
Initializing A Max Heap



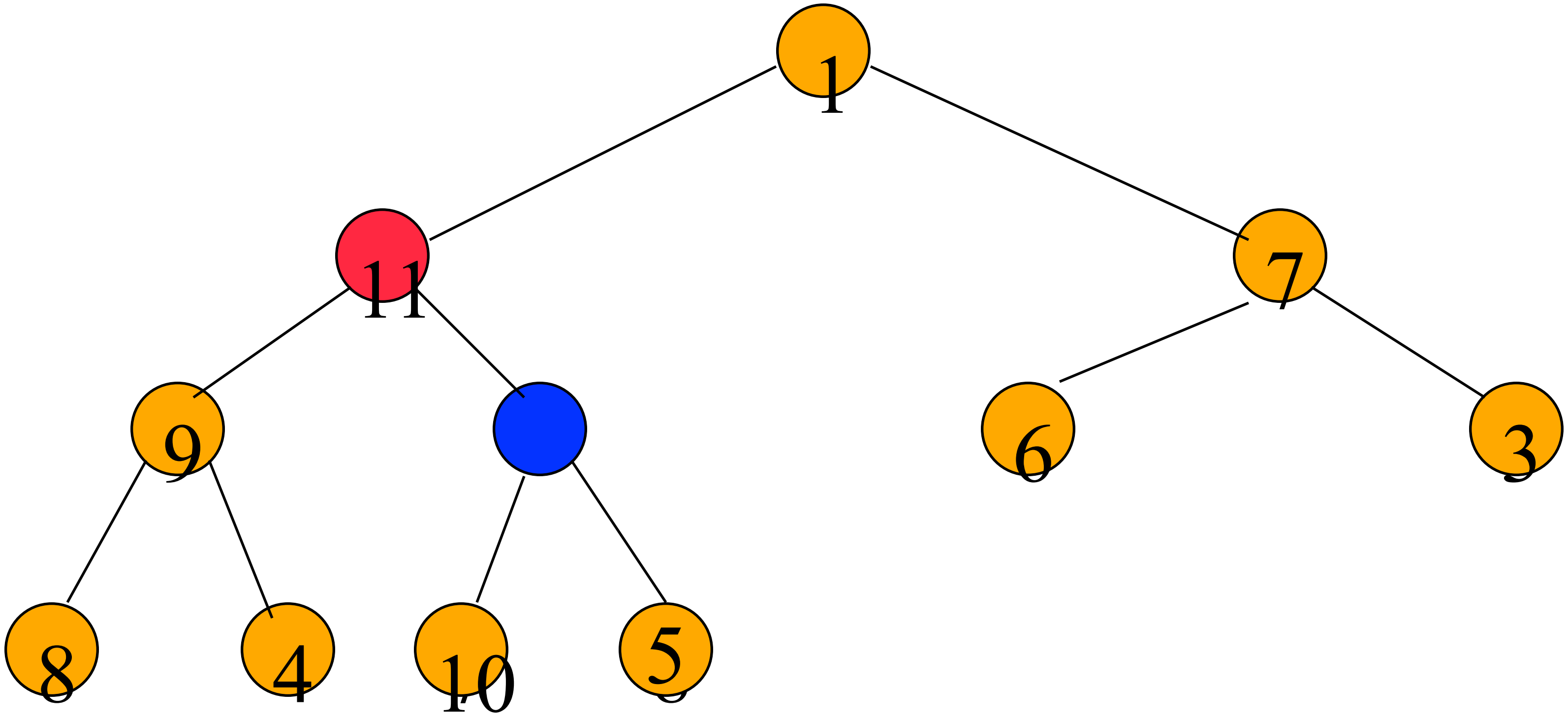
Initializing A Max Heap



Initializing A Max Heap

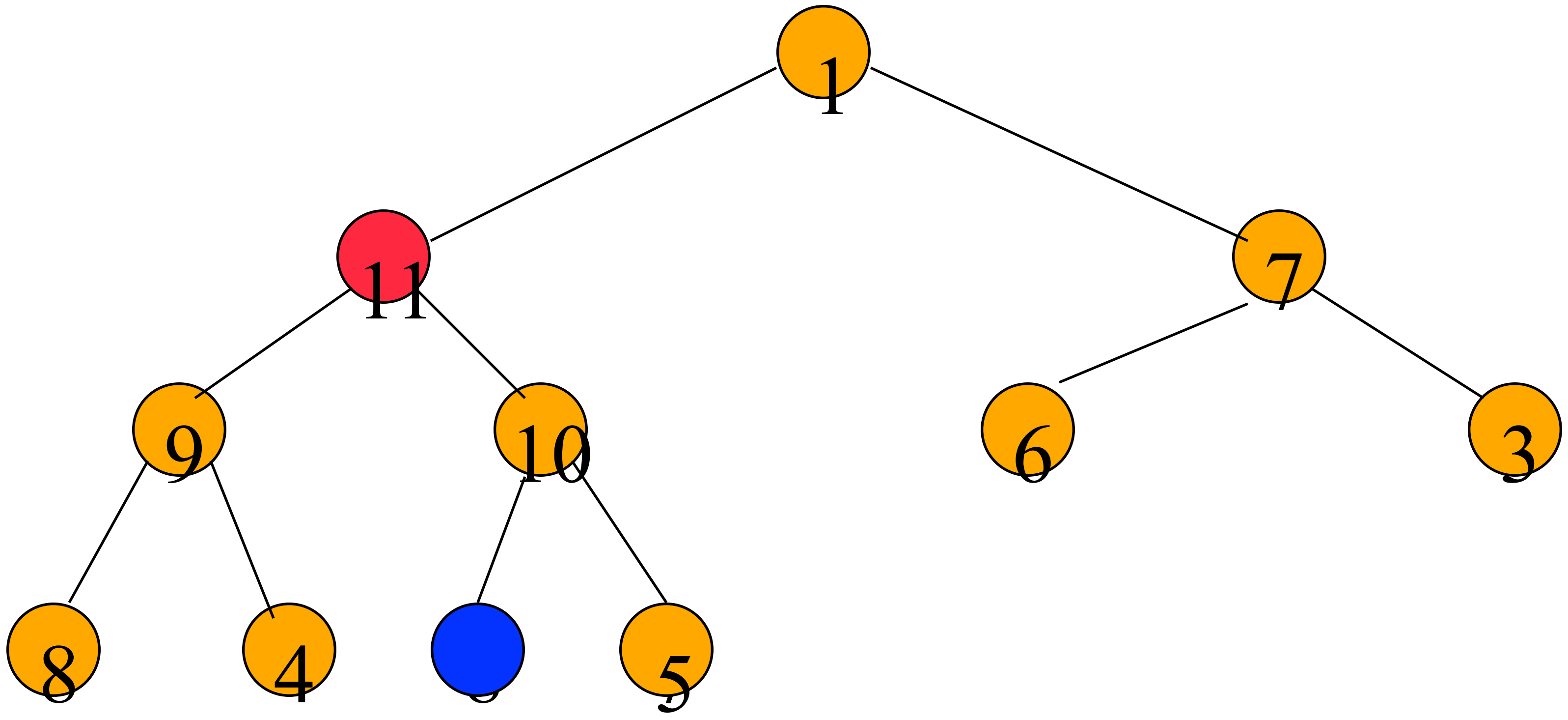


Initializing A Max Heap



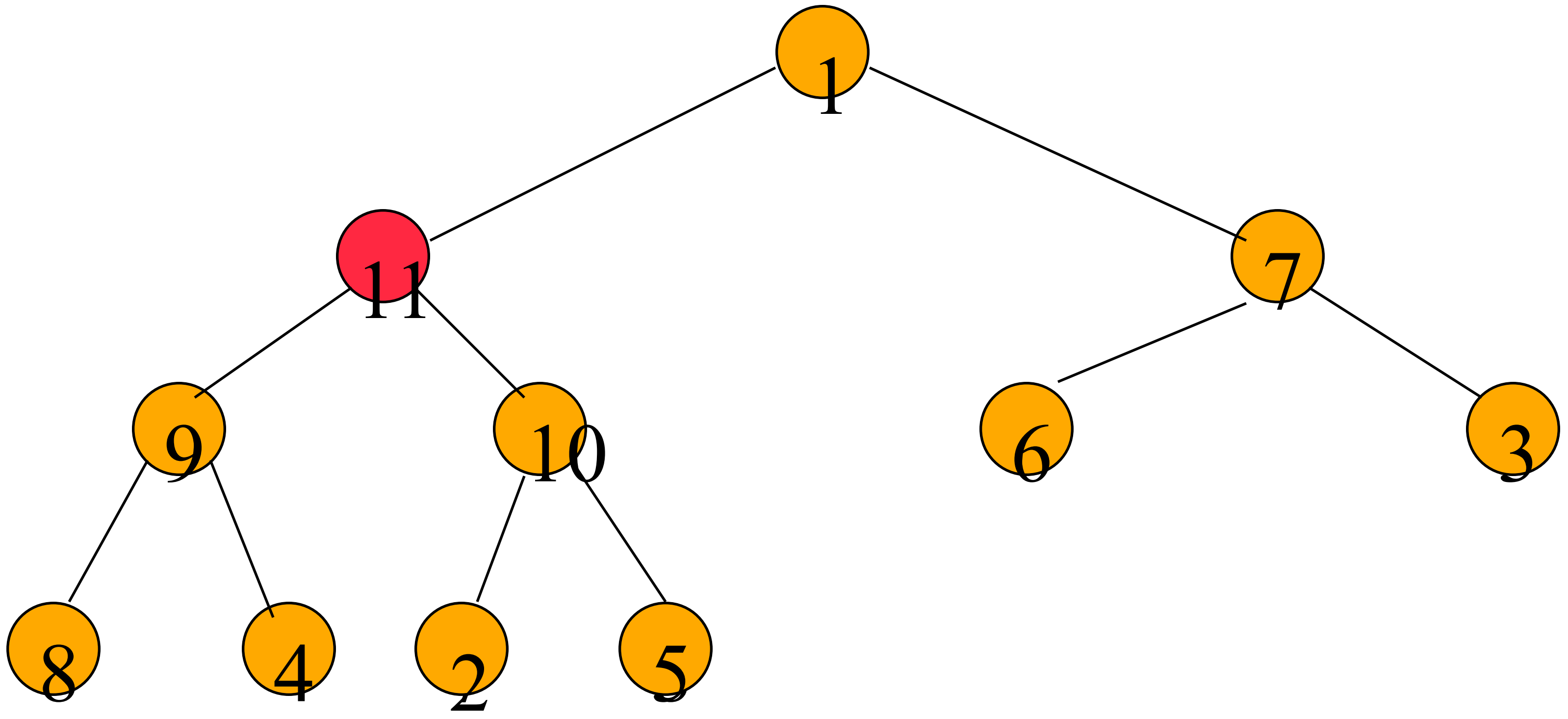
Find a home for 2.

Initializing A Max Heap



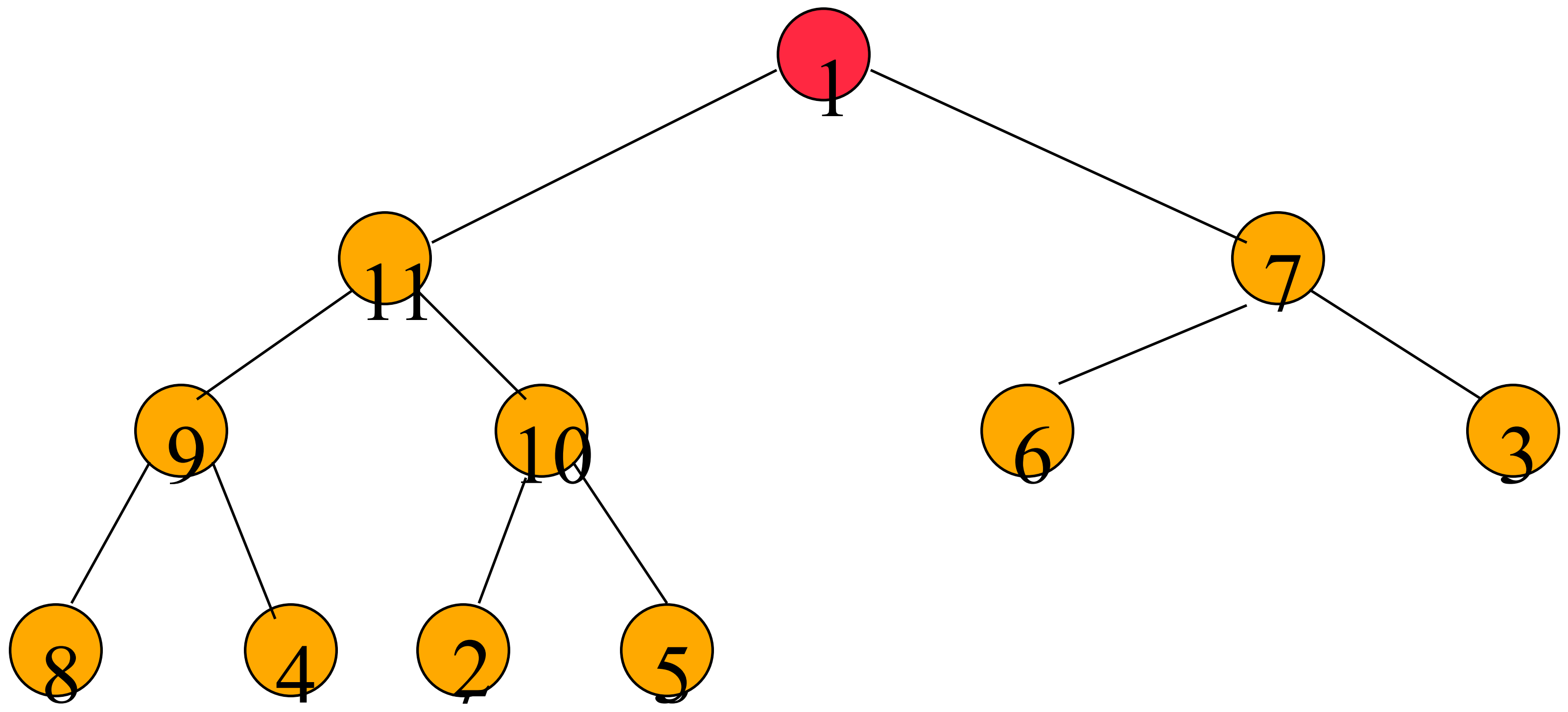
Find a home for 2.

Initializing A Max Heap



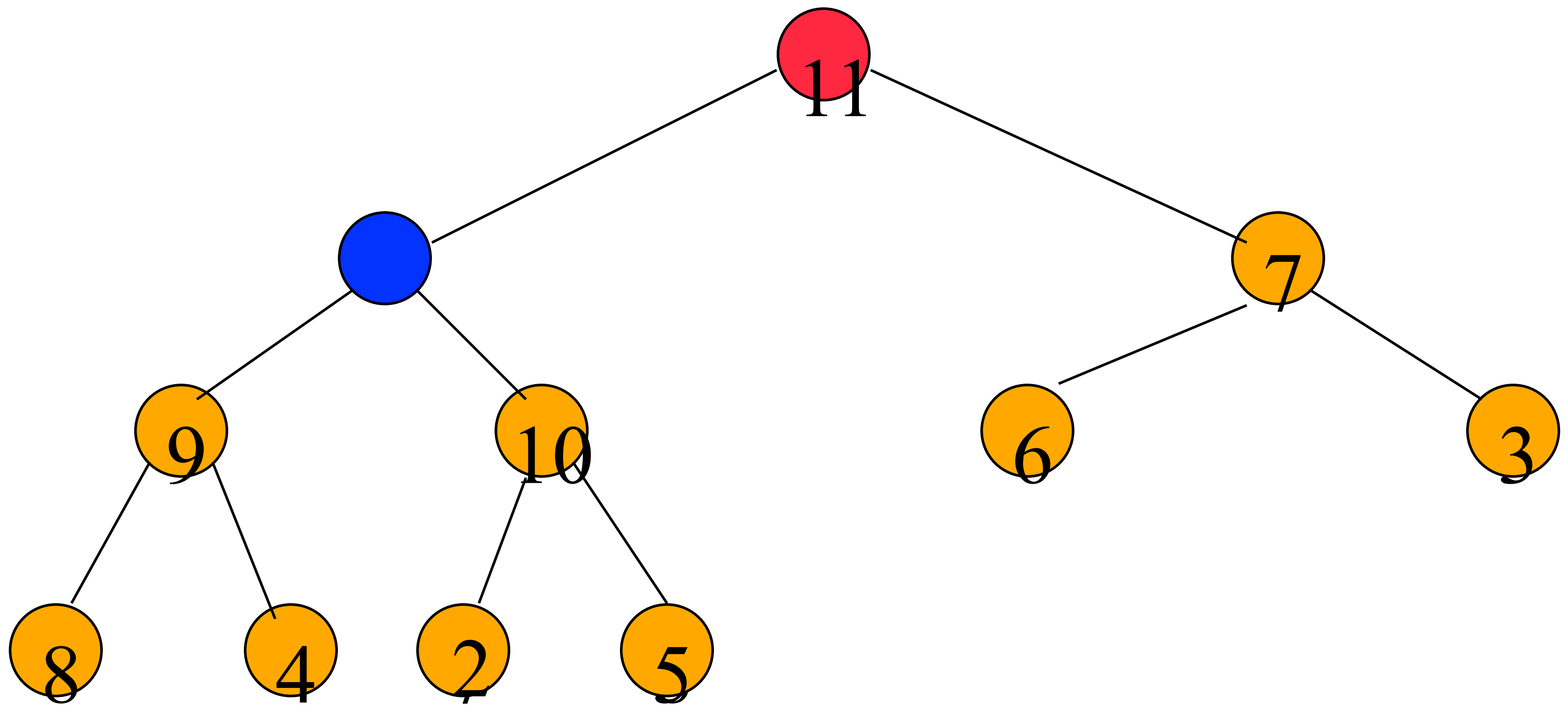
Done, move to next lower array position.

Initializing A Max Heap



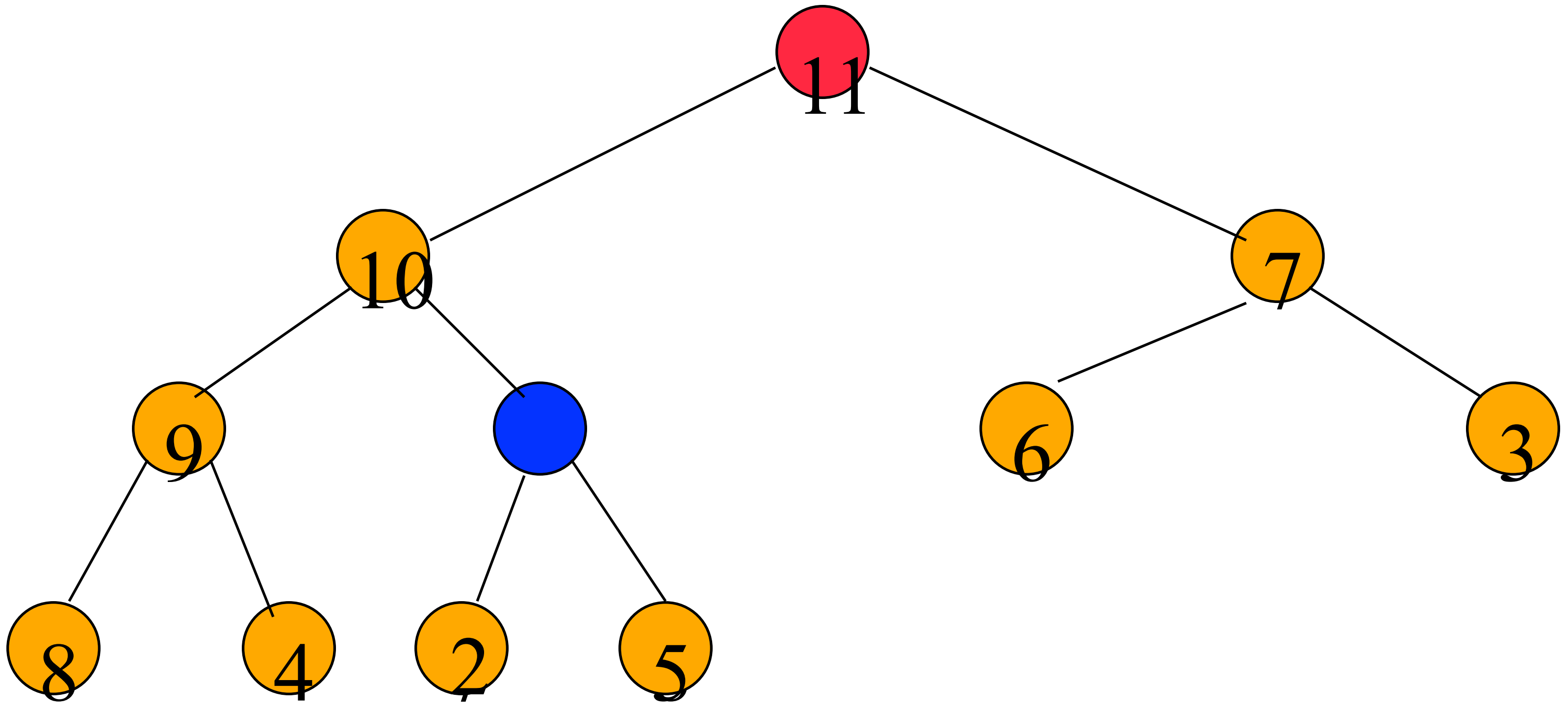
Find home for 1.

Initializing A Max Heap



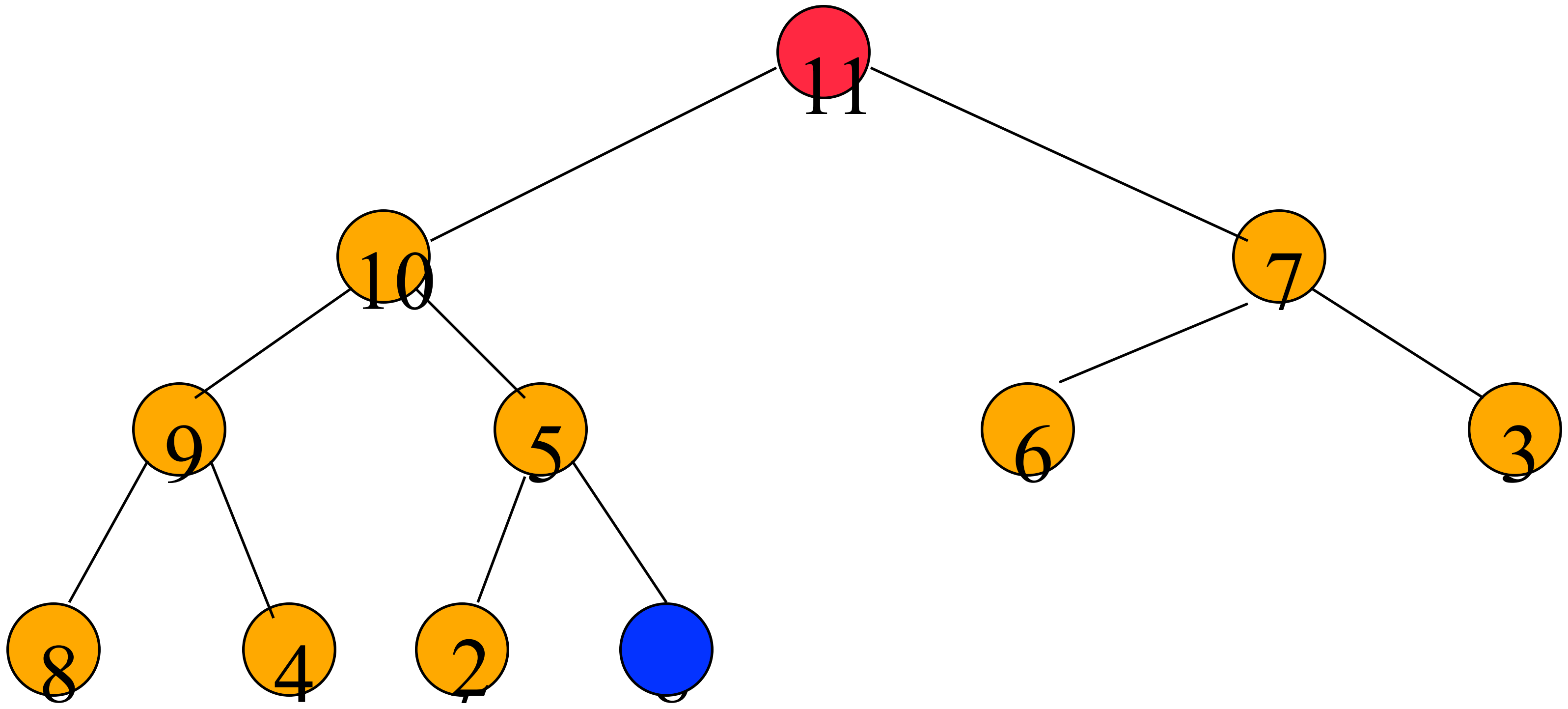
Find home for 1.

Initializing A Max Heap



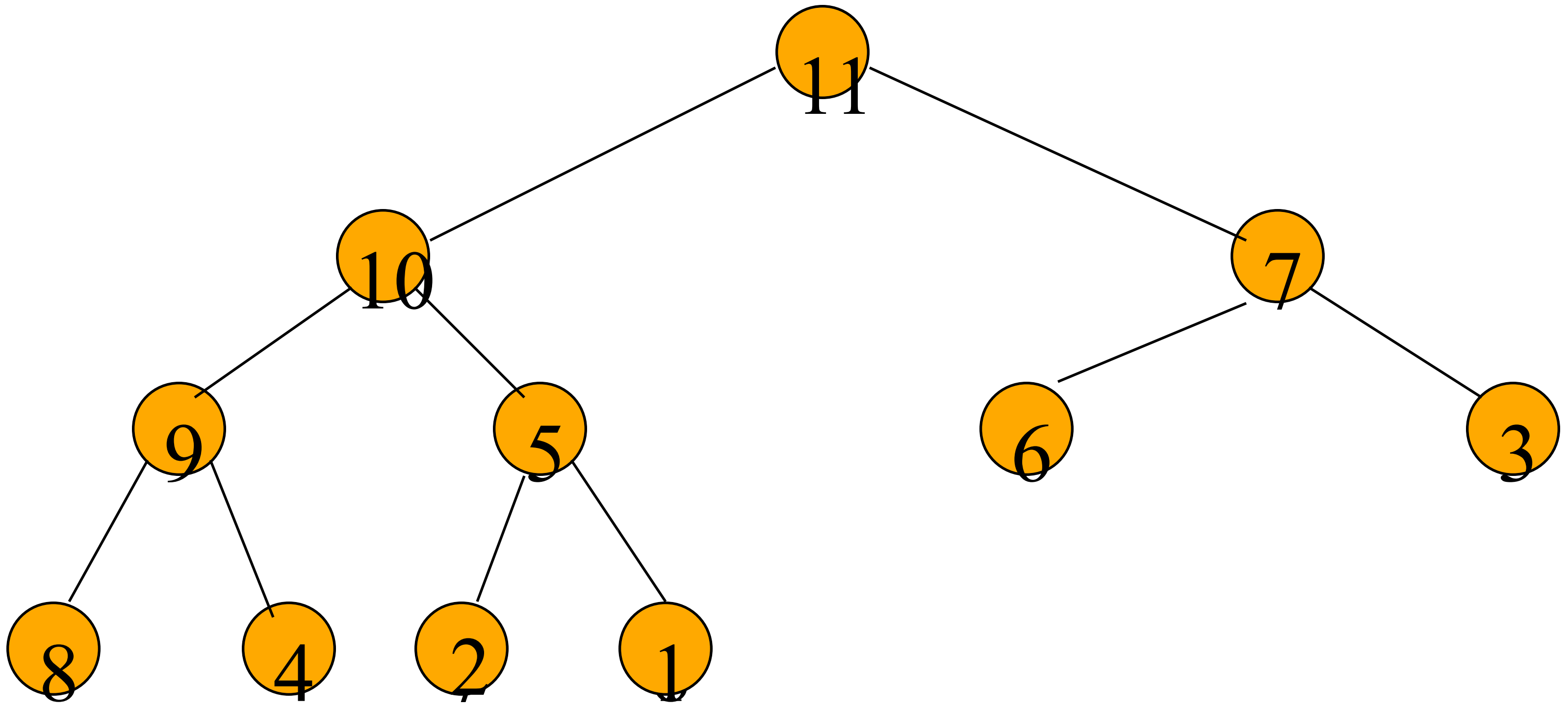
Find home for 1.

Initializing A Max Heap



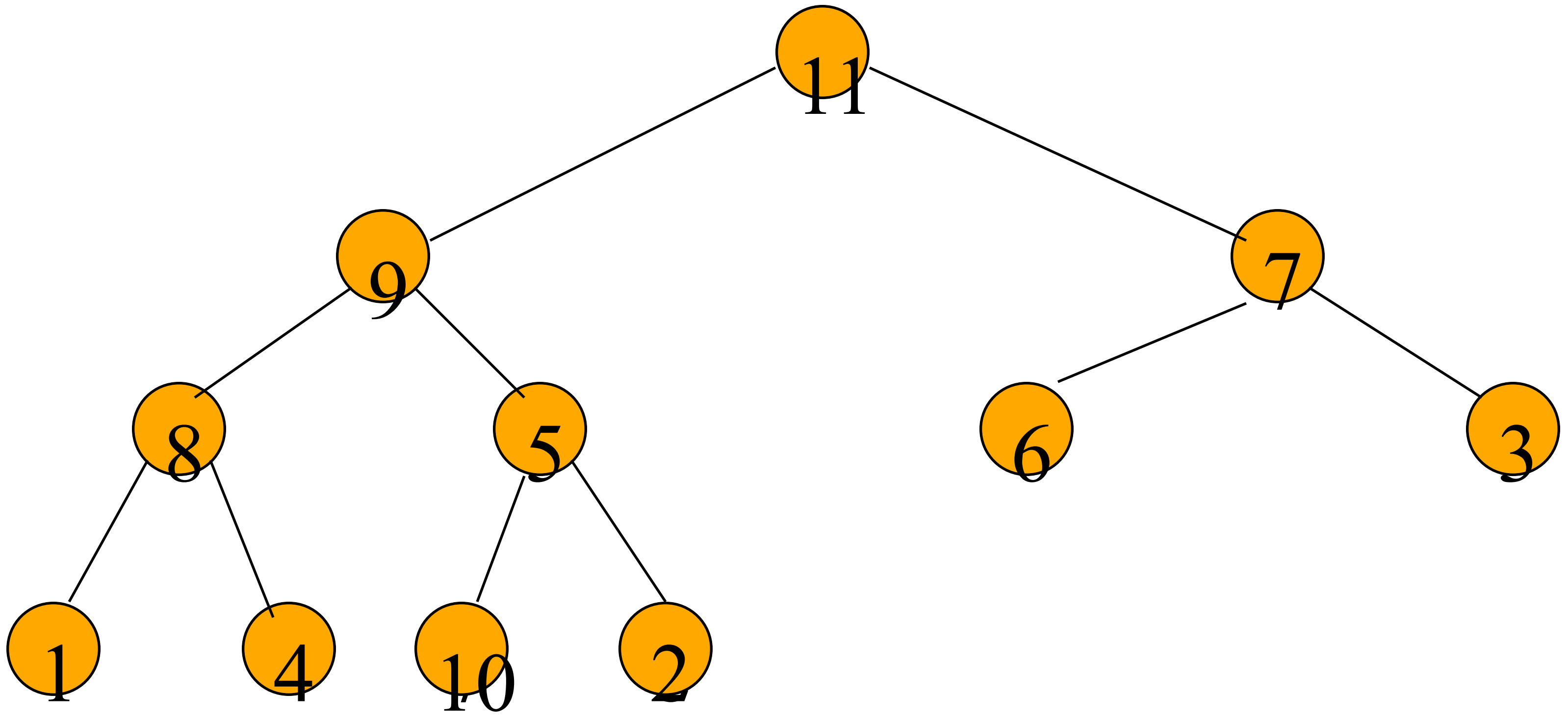
Find home for 1.

Initializing A Max Heap



Done.

Time Complexity



Height of heap = h.

Number of subtrees with root at level j is $<= 2^{j-1}$.

Time for each subtree is O(h-j+1).

Complexity

Time for level j subtrees is $<= 2^{j-1}(h-j+1) = t(j)$.

Total time is $t(1) + t(2) + \ldots + t(h-1) = O(n)$.

Programming:

Write an algorithm to initialize a Max Heap with C++.

Leftist Trees

Linked binary tree.

Can do everything a heap can do and in the same complexity.

- insert
- remove min (or max)
- initialize

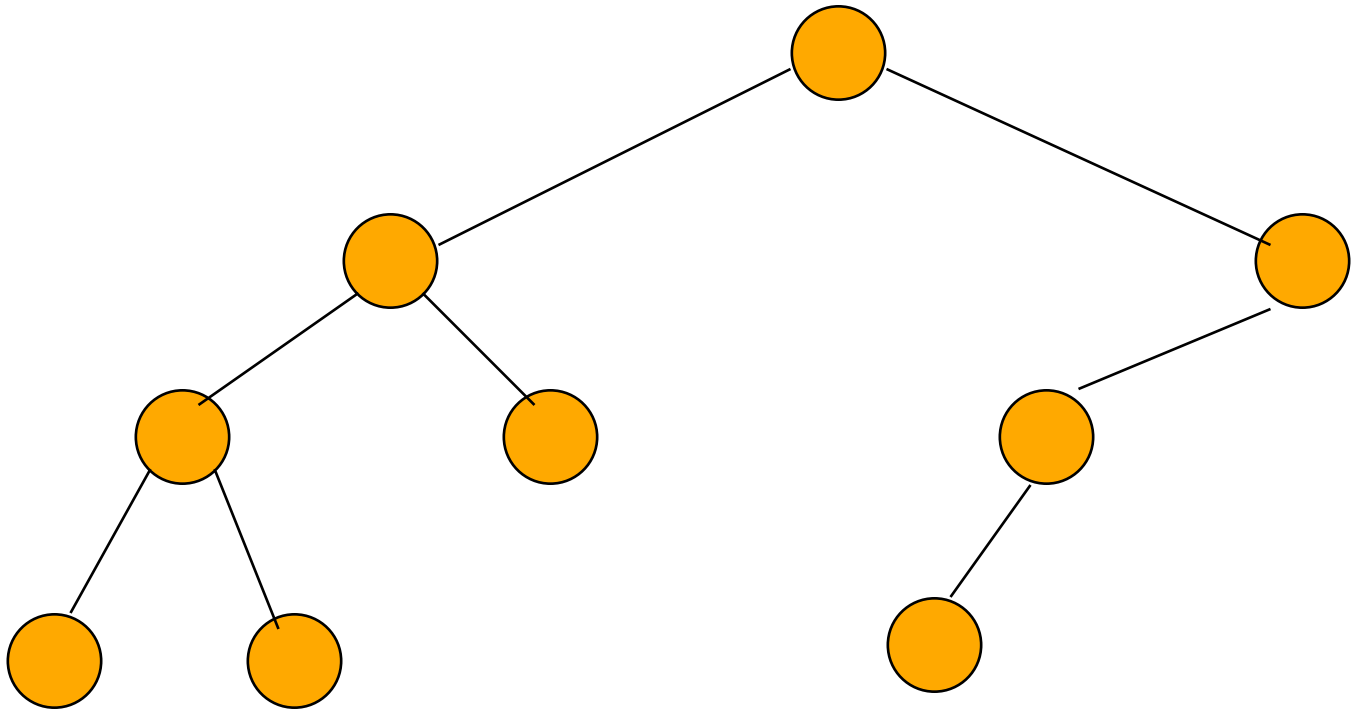
Can meld two leftist tree priority queues in O(log n) time.

Extended Binary Trees

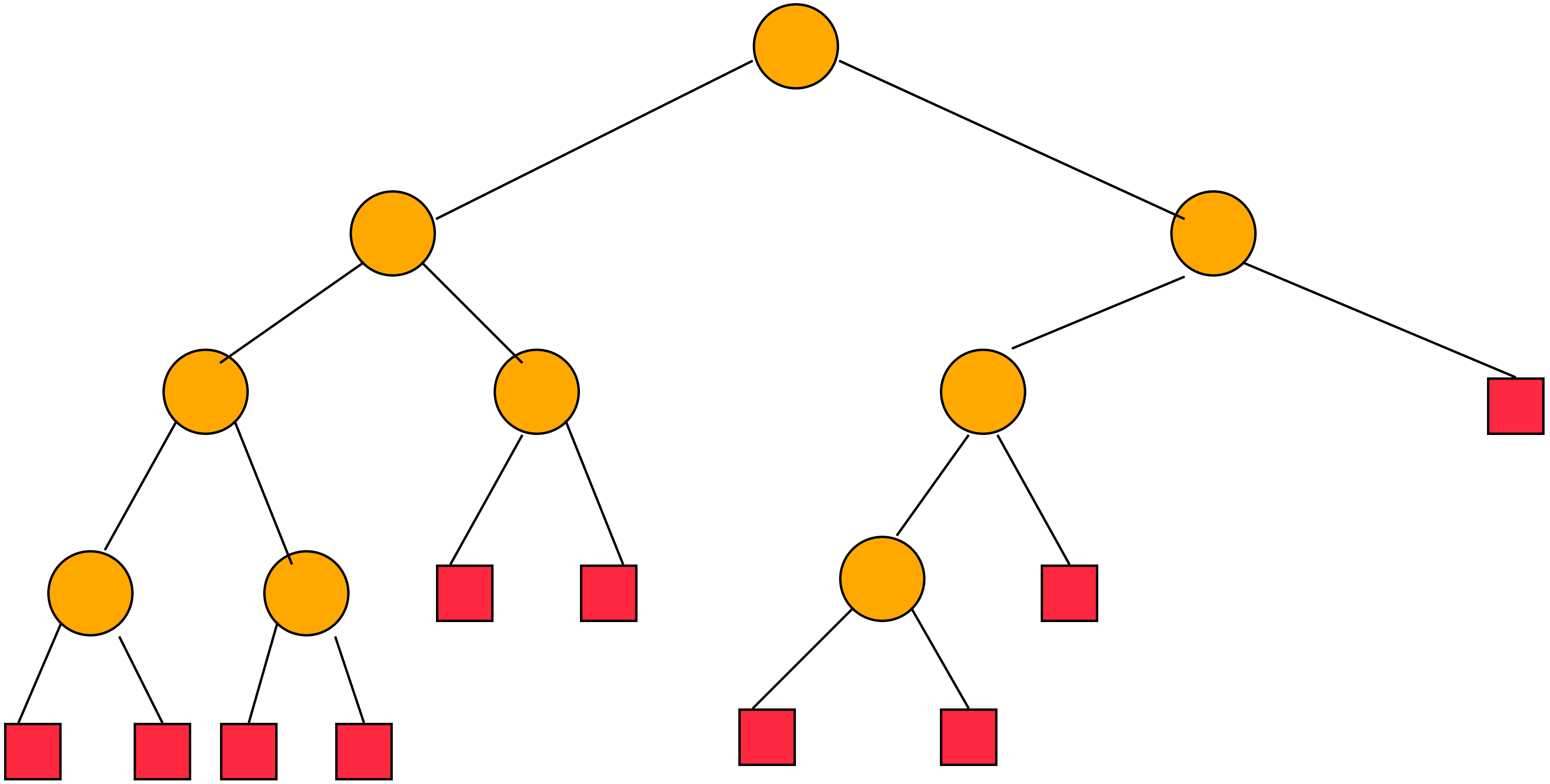
Start with any binary tree and add an external node wherever there is an empty subtree.

Result is an extended binary tree.

A Binary Tree



An Extended Binary Tree

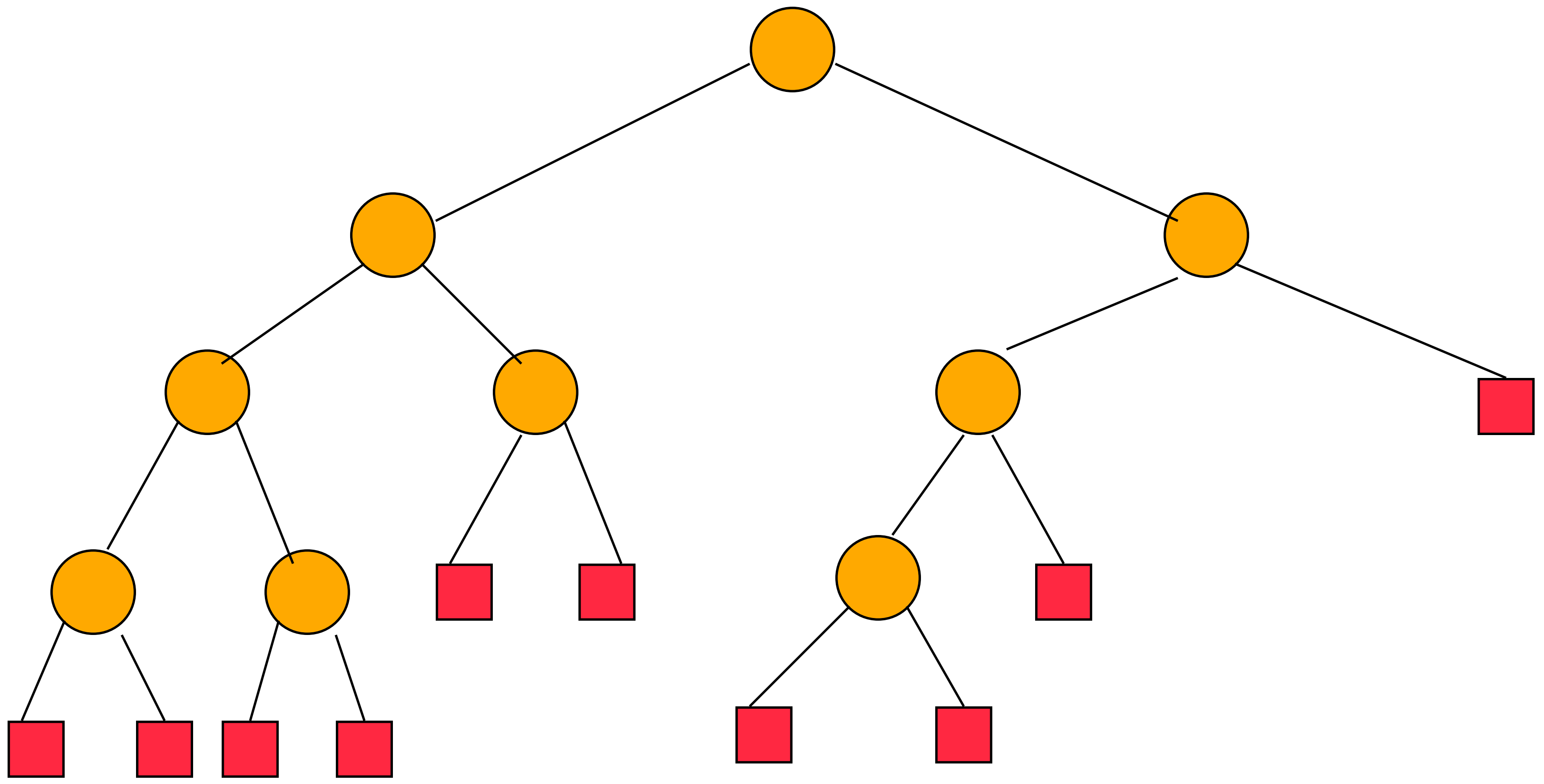


number of external nodes is n+1

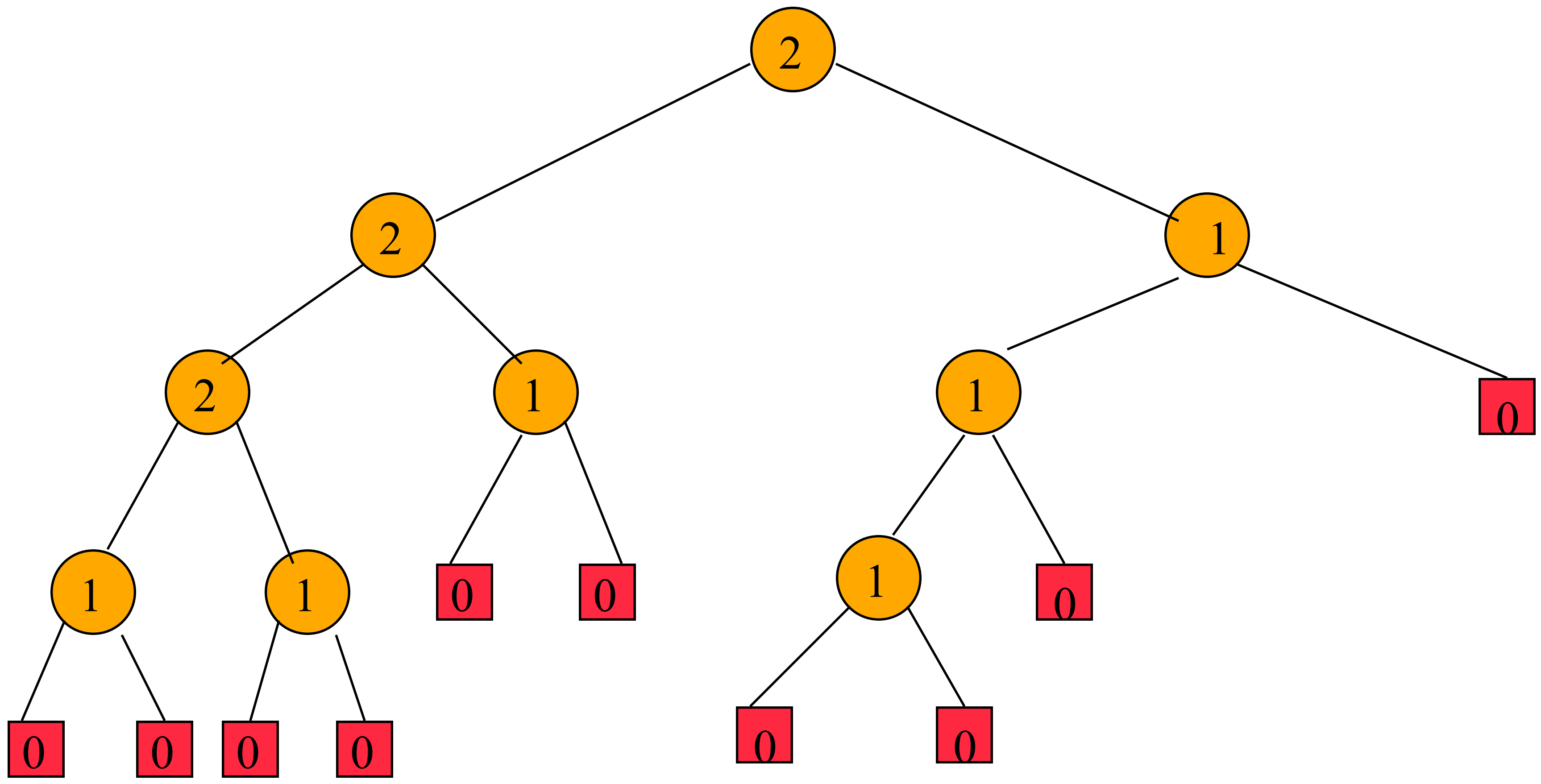
The Function s()

For any node x in an extended binary tree, let s(x) be the length of a shortest path from x to an external node in the subtree rooted at x.

s() Values Example



s() Values Example



Properties Of s()

If x is an external node, then s(x) = 0.

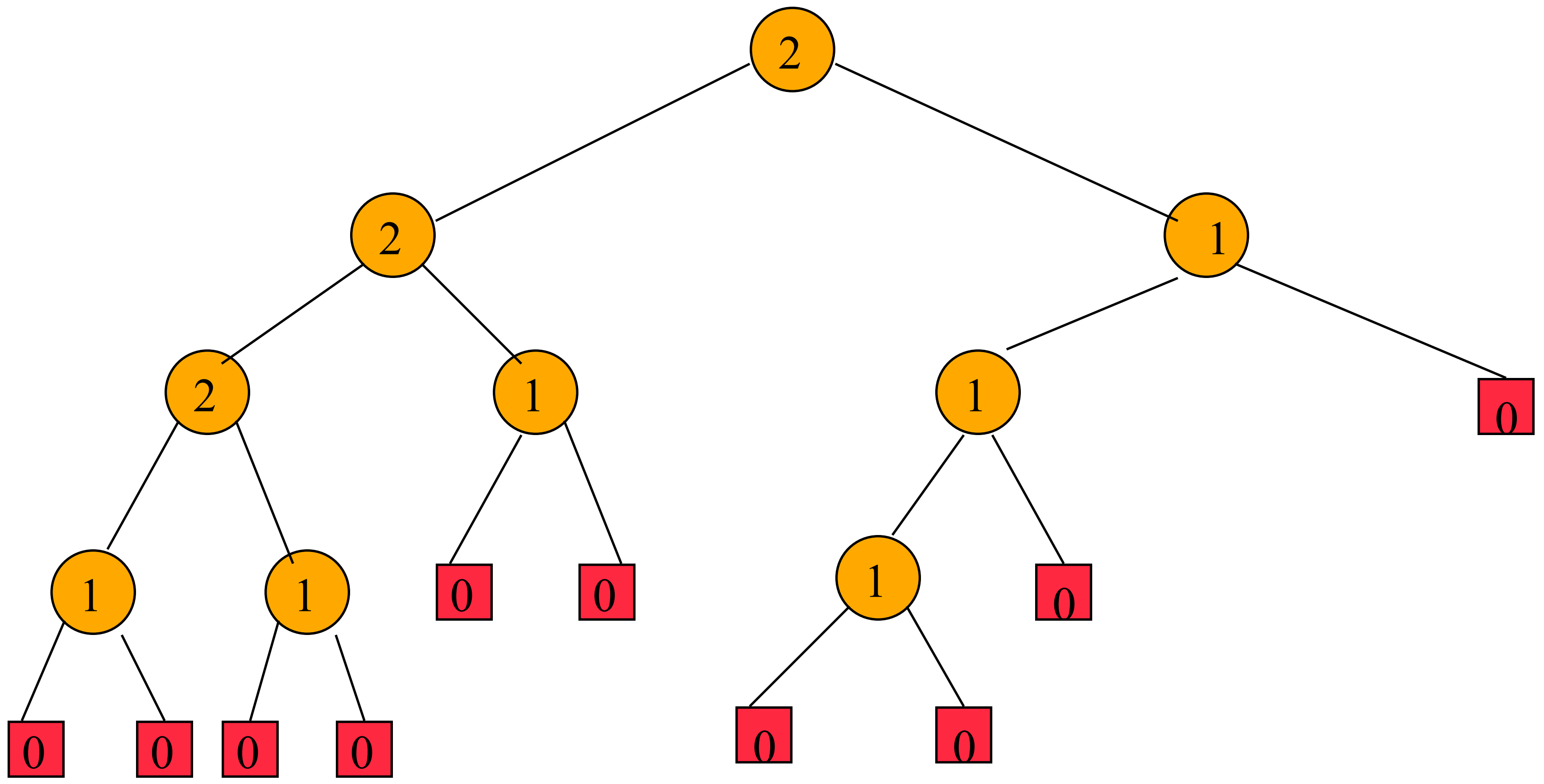
Otherwise,

$$s(x) = \min\{s(\text{leftChild}(x)),\ s(\text{rightChild}(x))\} + 1$$

Height Biased Leftist Trees

A binary tree is a (height biased) leftist tree iff for every internal node x, s(leftChild(x)) >= s(rightChild(x))

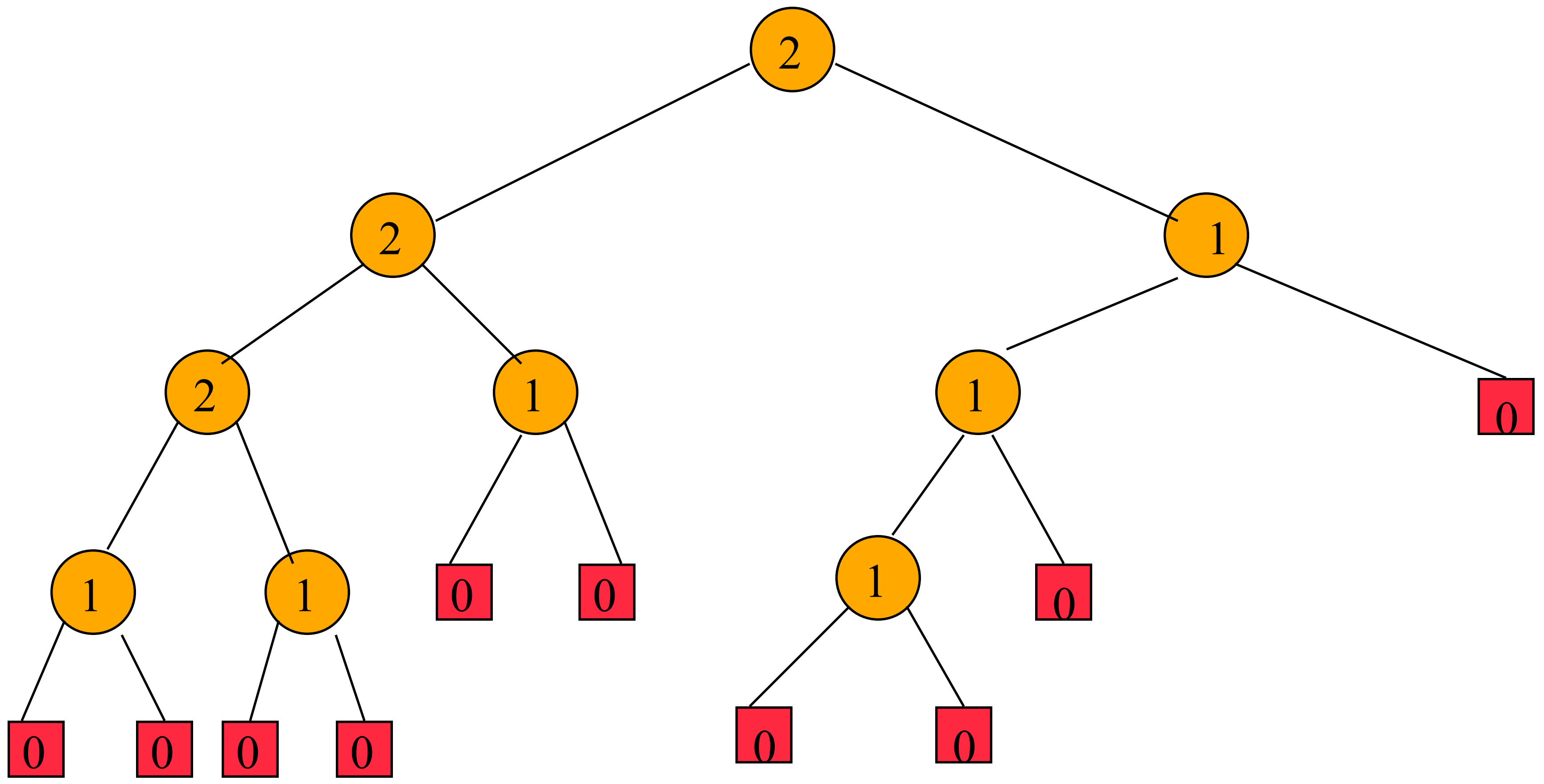
A Leftist Tree



Leftist Trees – Property 1

In a leftist tree, the rightmost path is a shortest root to external node path and the length of this path is s(root).

A Leftist Tree



Length of rightmost path is 2.

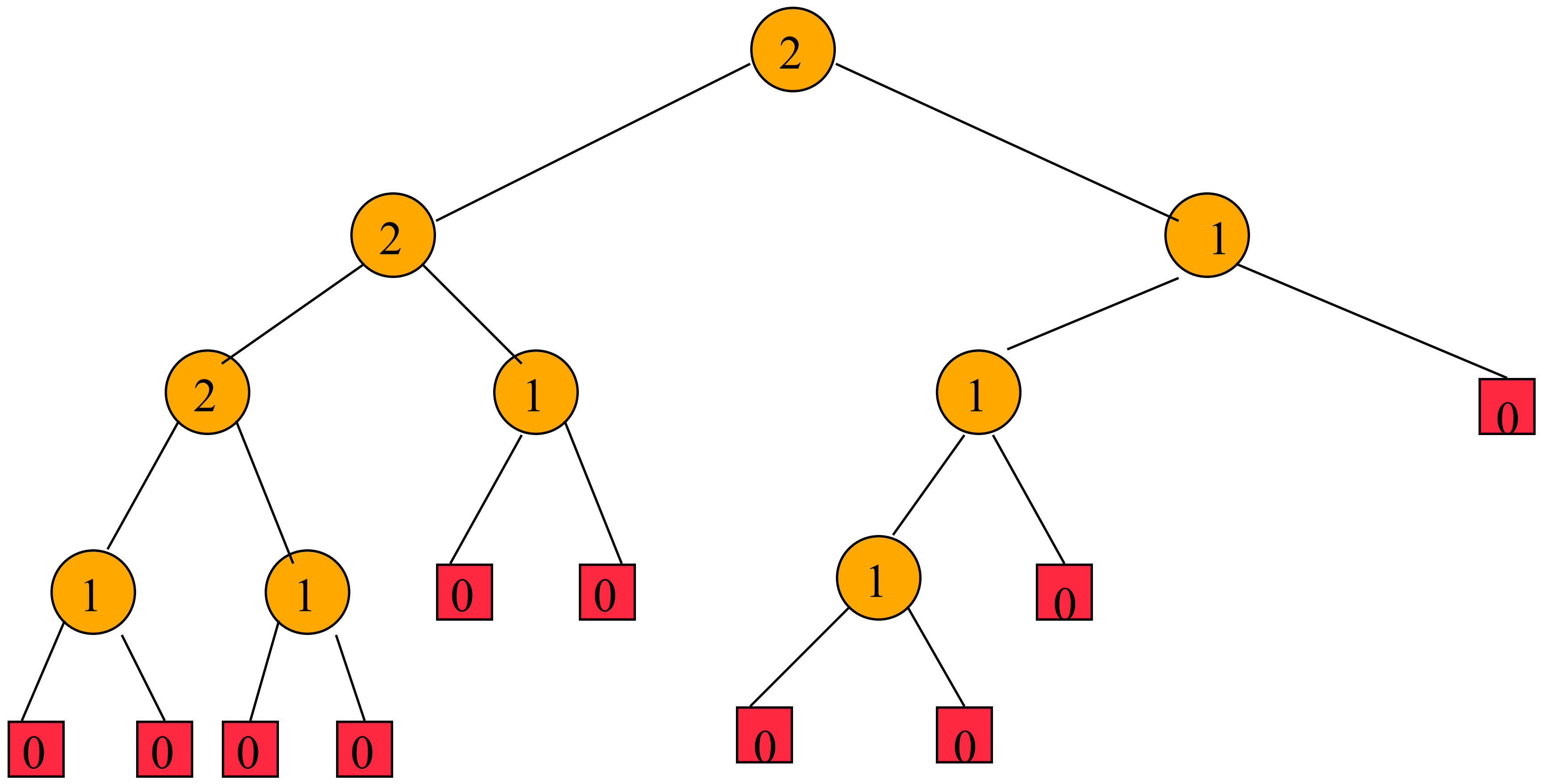
Leftist Trees—Property 2

The number of internal nodes is at least

$2^{s(root)} - 1$

Because levels 1 through s(root) have no external nodes.

A Leftist Tree



Levels 1 and 2 have no external nodes.

Leftist Trees—Property 3

Length of rightmost path is O(log n), where n is the number of (internal) nodes in a leftist tree.

Property 2 =>

- $n >= 2^{s(root)} - 1 => s(root) <= \log_2(n+1)$

Property 1 => length of rightmost path is s(root).

Leftist Trees As Priority Queues

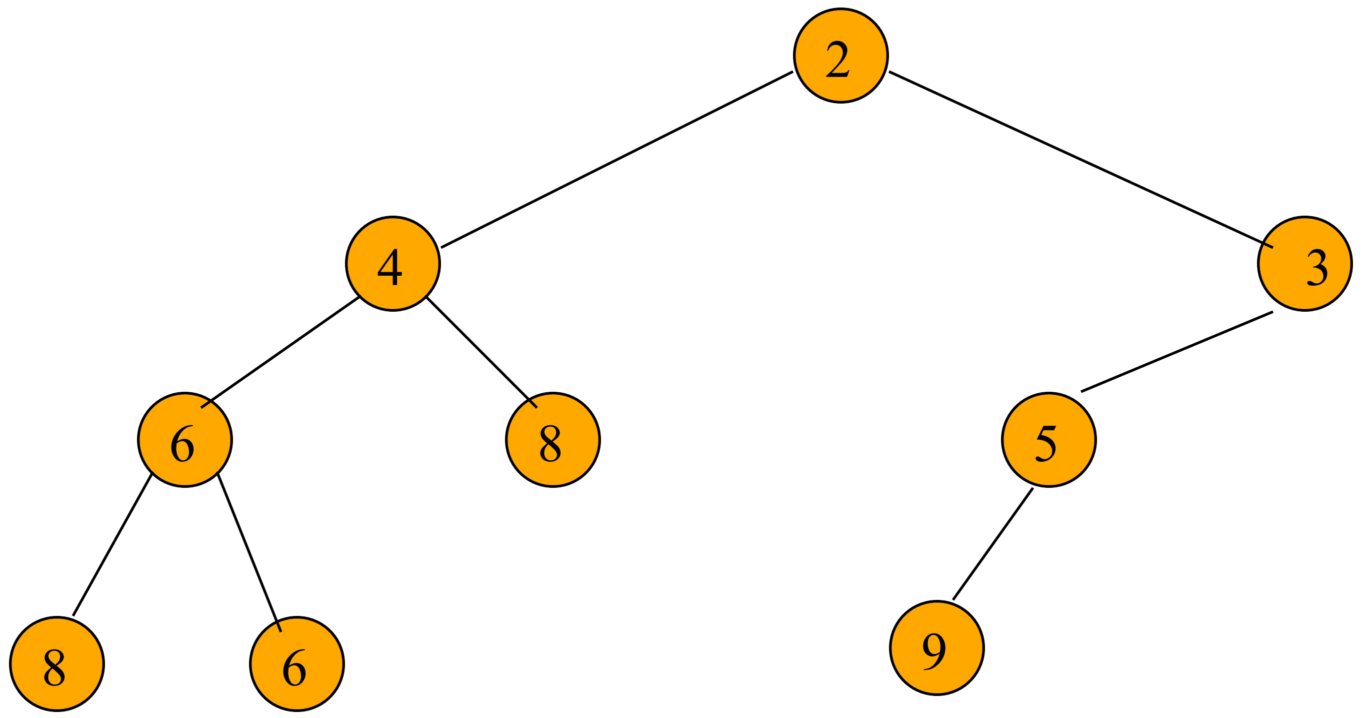
Min leftist tree … leftist tree that is a min tree.

Used as a min priority queue.

Max leftist tree … leftist tree that is a max tree.

Used as a max priority queue.

A Min Leftist Tree



Some Min Leftist Tree Operations

put

removeMin()

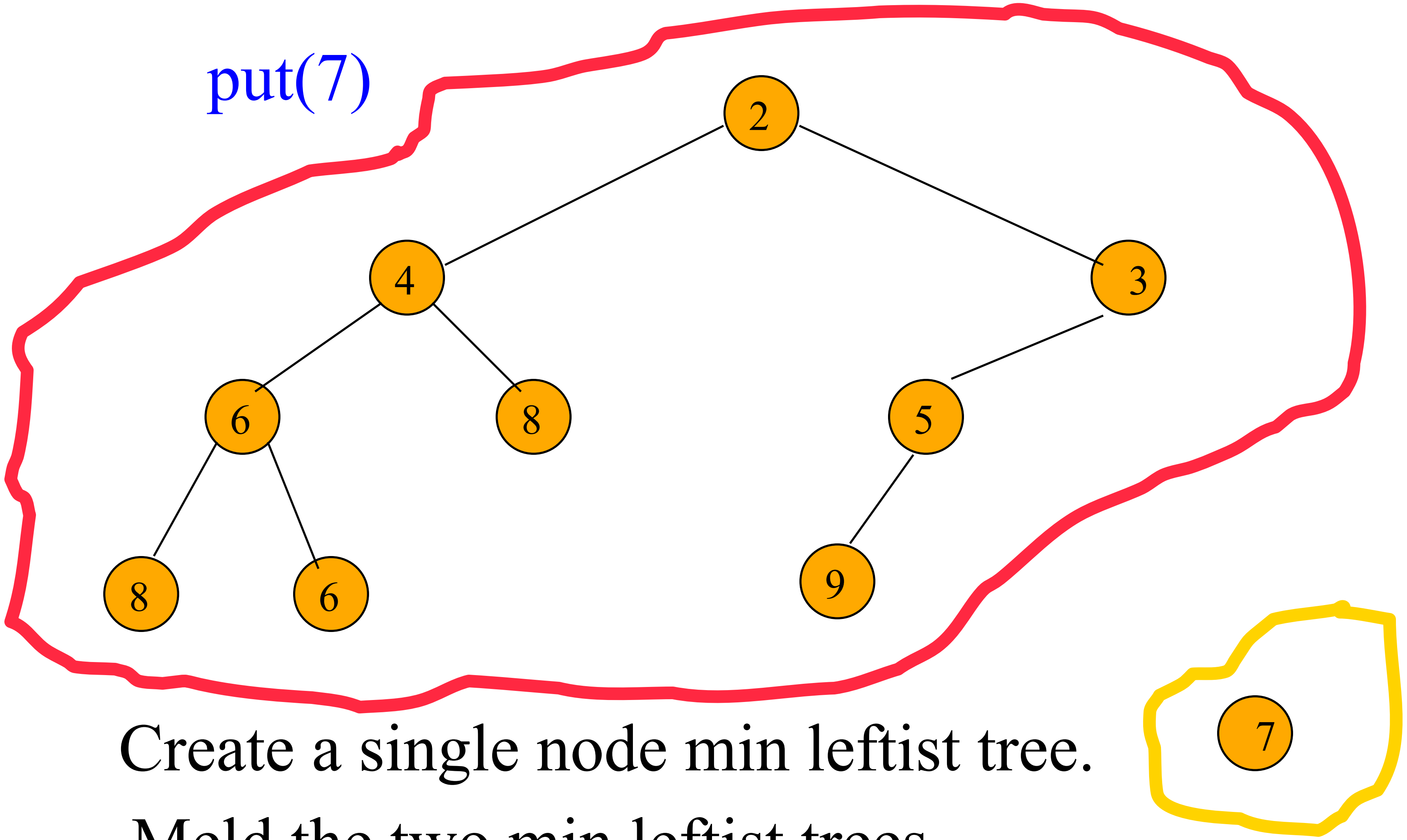
meld()

initialize()

put() and removeMin() use meld().

Put Operation

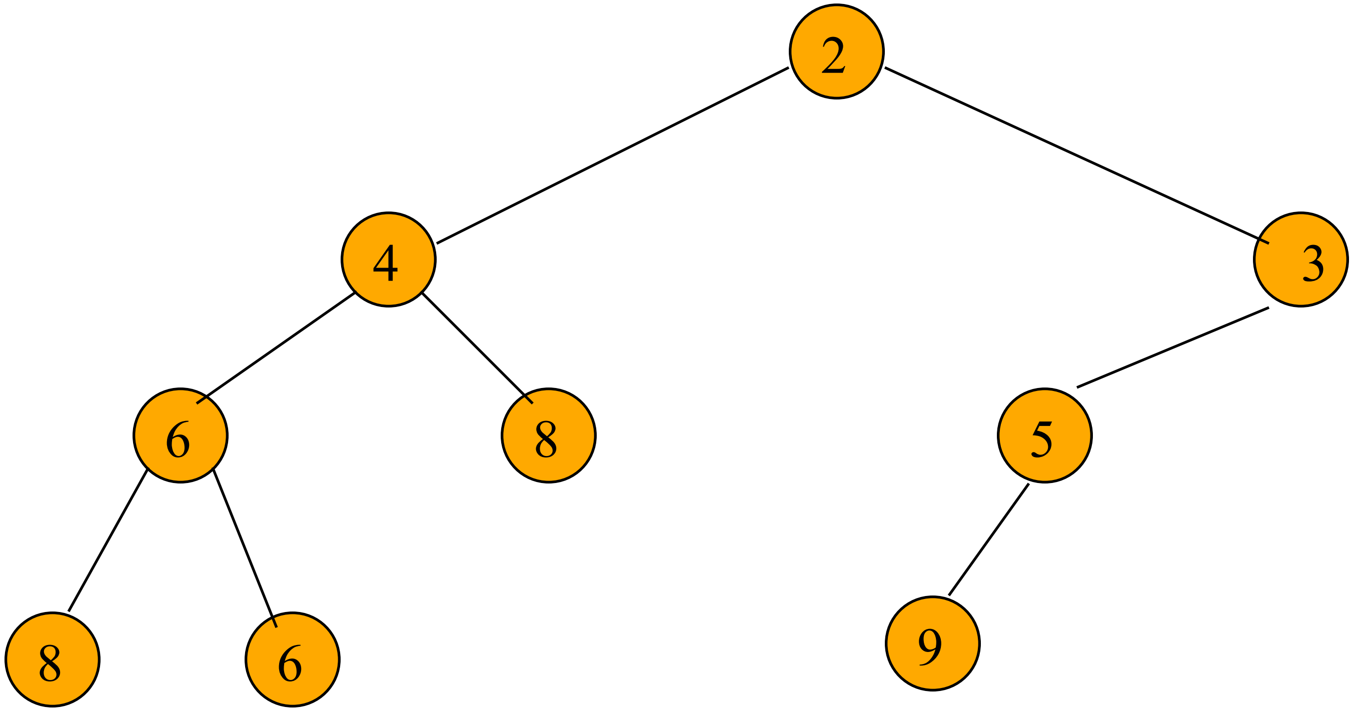
put(7)



Create a single node min leftist tree.

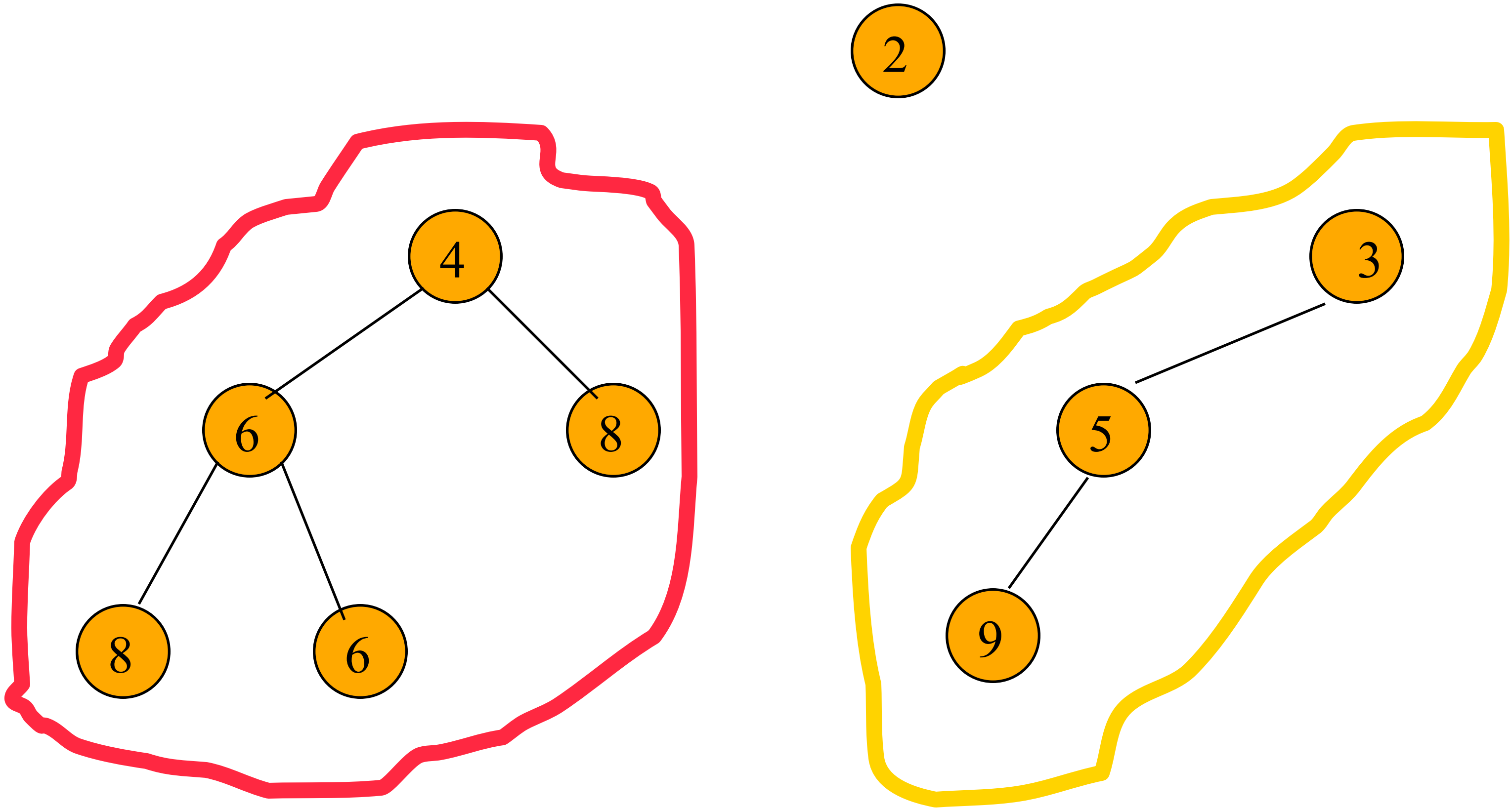
Meld the two min leftist trees.

Remove Min



Remove the root.

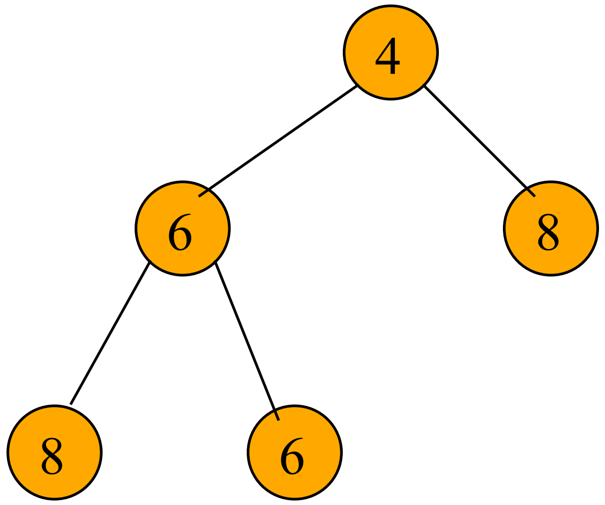
Remove Min

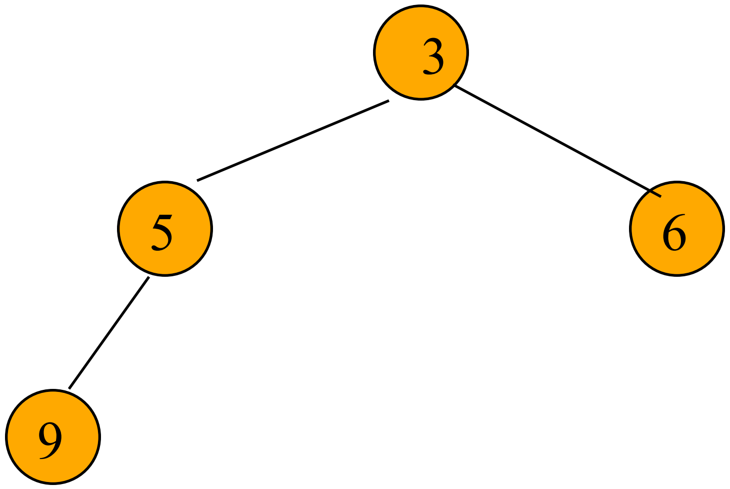


Remove the root.

Meld the two subtrees.

Meld Two Min Leftist Trees

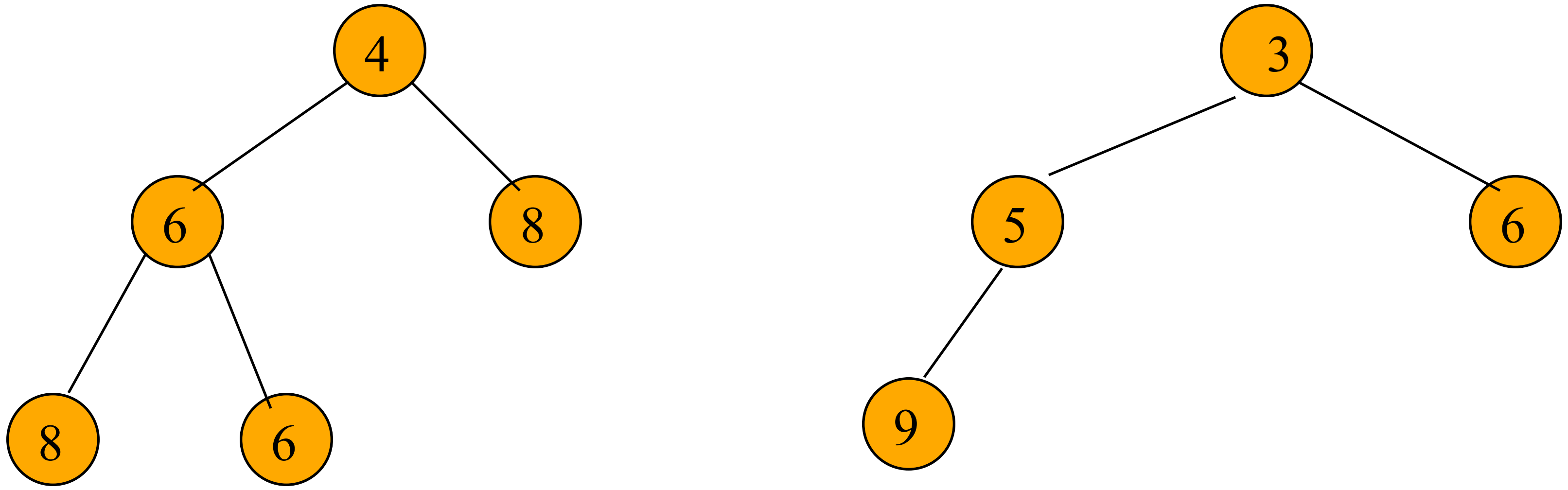




HOW to get logarithmic performance?

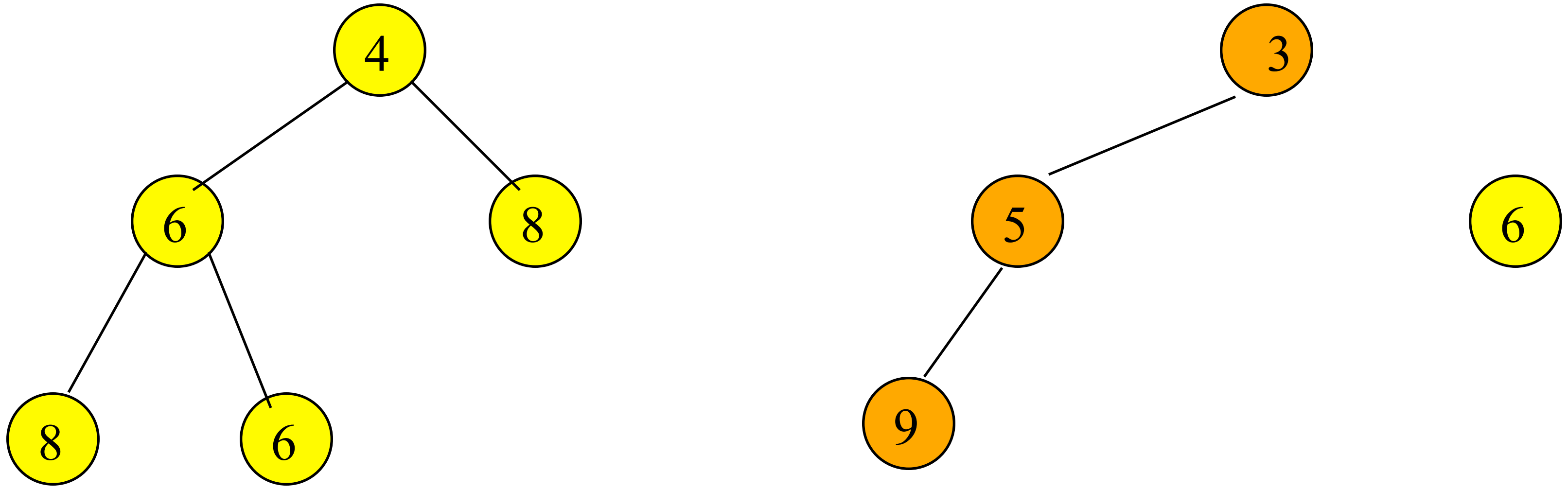
Traverse only the rightmost paths

Meld Two Min Leftist Trees



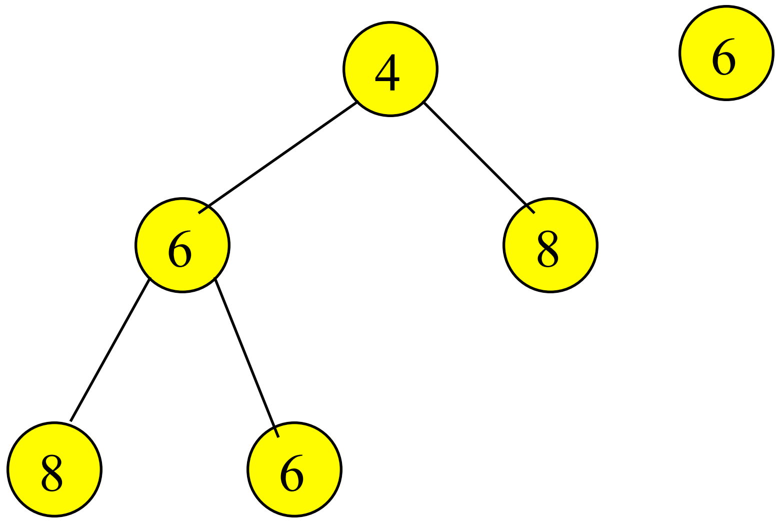
Meld right subtree of tree with smaller root and all of other tree.

Meld Two Min Leftist Trees



Meld right subtree of tree with smaller root and all of other tree.

Meld Two Min Leftist Trees



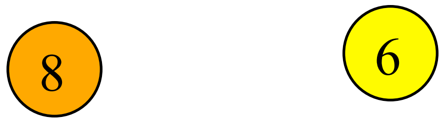
Meld right subtree of tree with smaller root and all of other tree.

Meld Two Min Leftist Trees

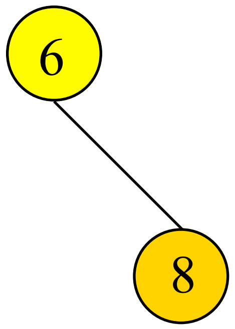
Meld right subtree of tree with smaller root and all of other tree.

Right subtree of 6 is empty. So, result of melding right subtree of tree with smaller root and other tree is the other tree.

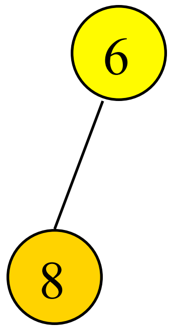
Meld Two Min Leftist Trees



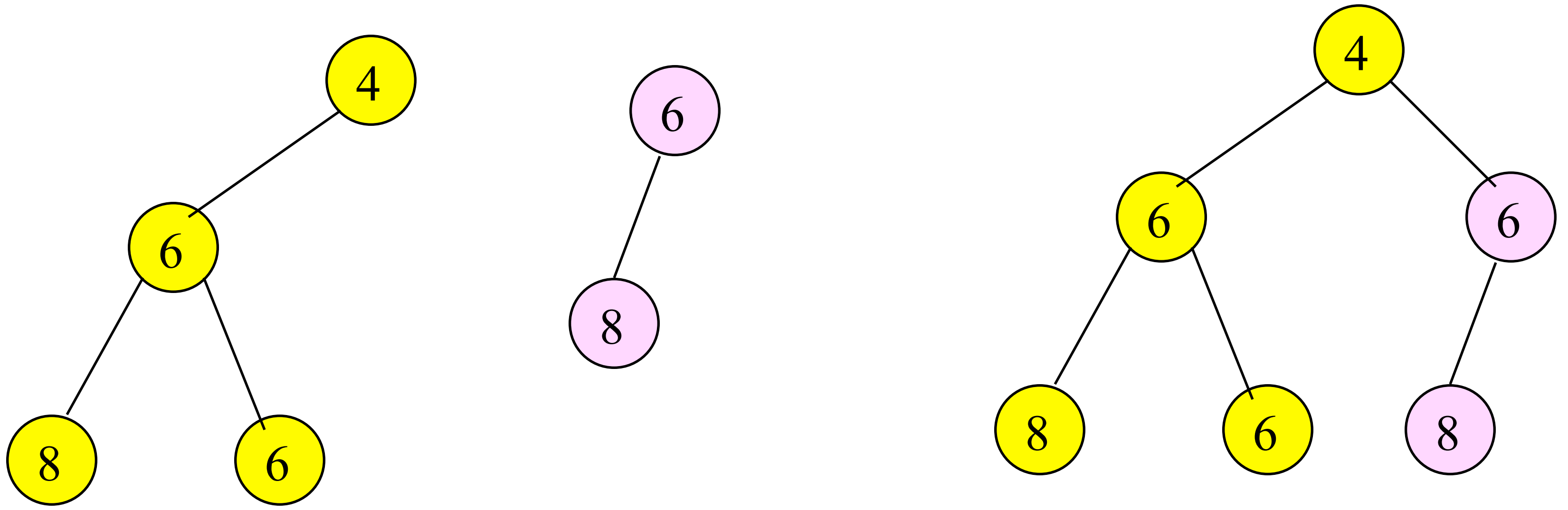
Make melded subtree right subtree of smaller root.



Swap left and right subtree if s(left) < s(right).



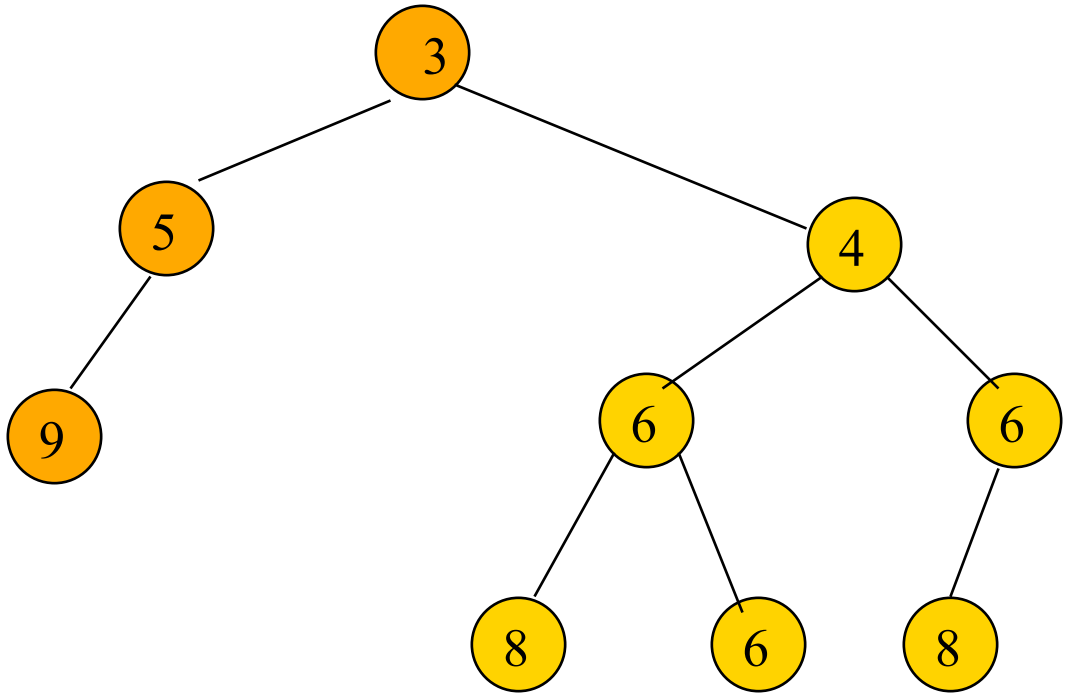
Meld Two Min Leftist Trees



Make melded subtree right subtree of smaller root.

Swap left and right subtree if s(left) < s(right).

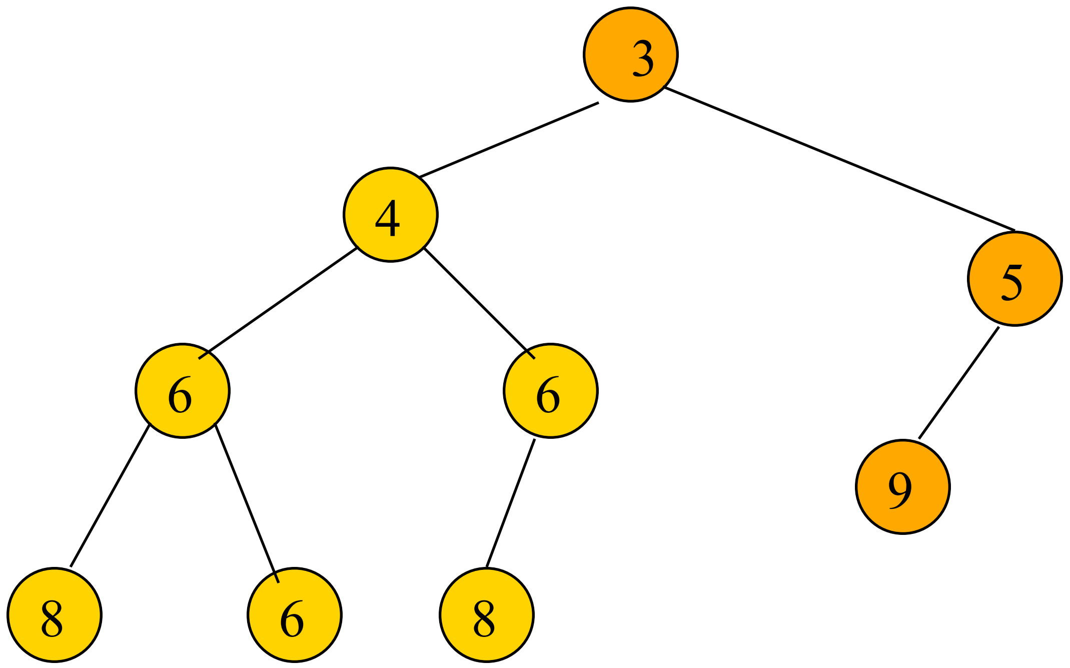
Meld Two Min Leftist Trees



Make melded subtree right subtree of smaller root.

Swap left and right subtree if s(left) < s(right).

Meld Two Min Leftist Trees

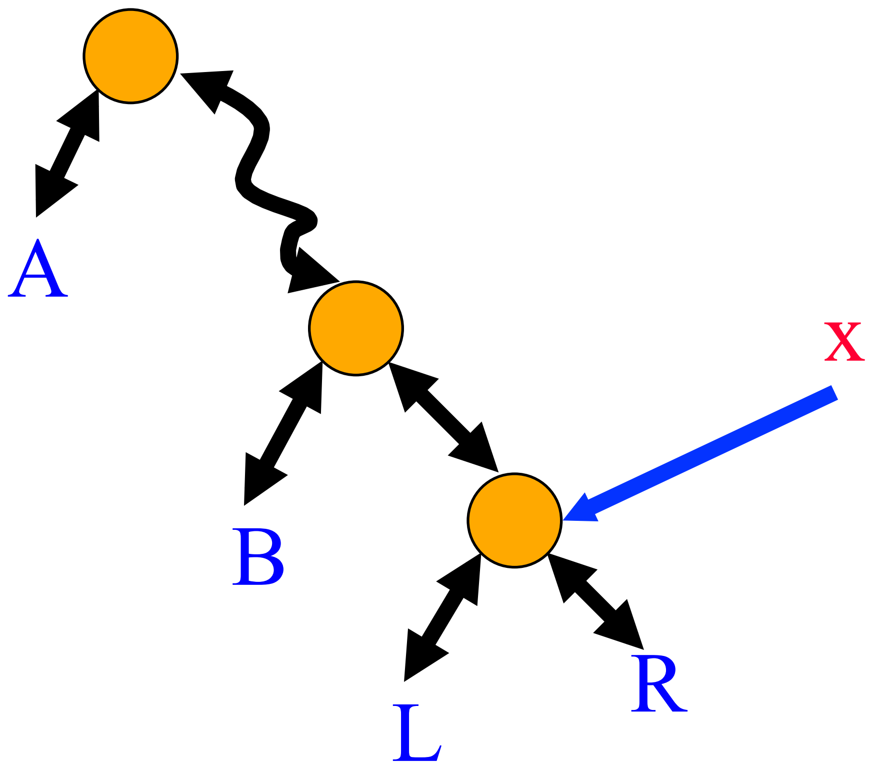


Initializing In O(n) Time

- Create n single-node min leftist trees and place them in a FIFO queue.
- Repeatedly remove two min leftist trees from the FIFO queue, meld them, and put the resulting min leftist tree into the FIFO queue.
- The process terminates when only 1 min leftist tree remains in the FIFO queue.
- Analysis is the same as for heap initialization.

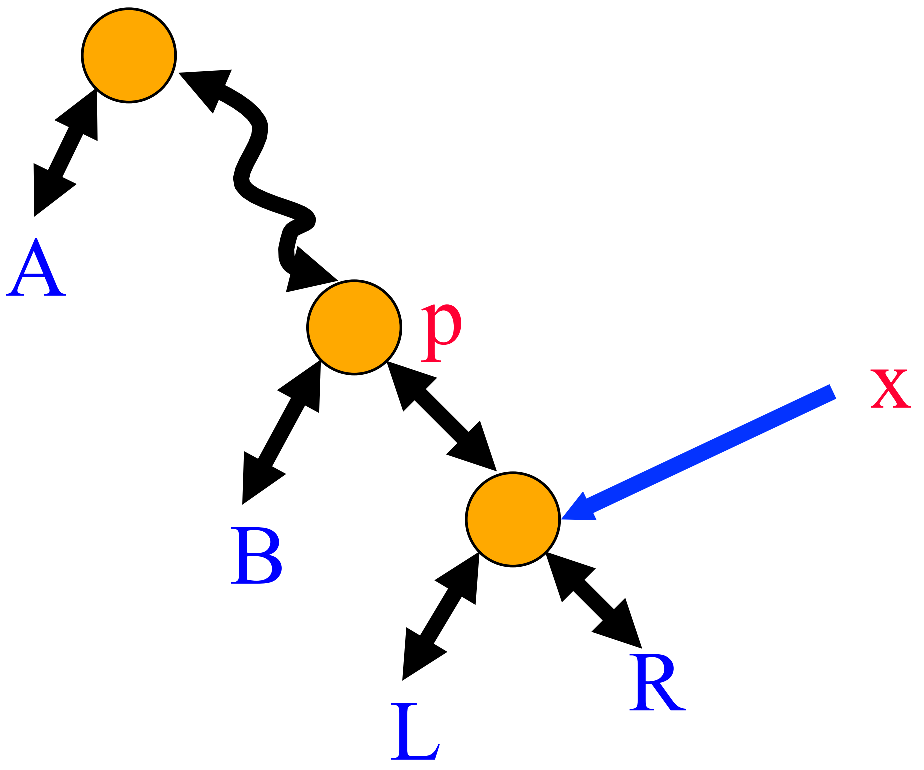
Arbitrary Remove

Remove element in node pointed at by x.



x = root => remove min.

Arbitrary Remove, x != root

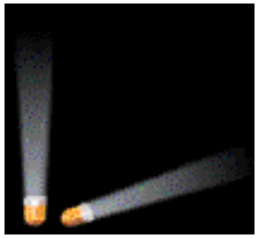


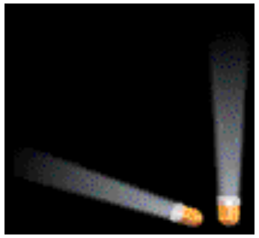
Make L right subtree of p.

Adjust s and leftist property on path from p to root.

Meld with R.

Selection Trees/ Tournament Trees





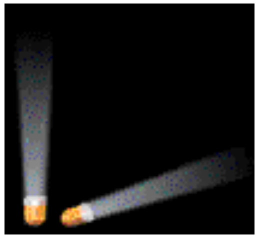
Polynomial addition (no equal items)

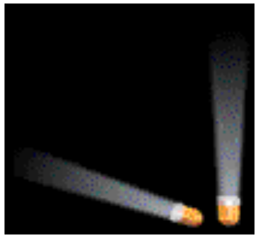
- Merge: 2 sorted lists → one
  - 2 items : smaller one selected

K polynomials?

- Merge: k sorted lists → one
  - k items : smallest one selected
  - HOW?

Selection Trees/ Tournament Trees





Winner trees.

Loser Trees.

World Cup Knockout

16 teams

8 1/8 matches → 8 winners

4 1/4 matches → 4 winners

2 semifinal matches → 2 winners

1 final match → World Cup Championship

Winner Tree: A simulation

Winner Trees

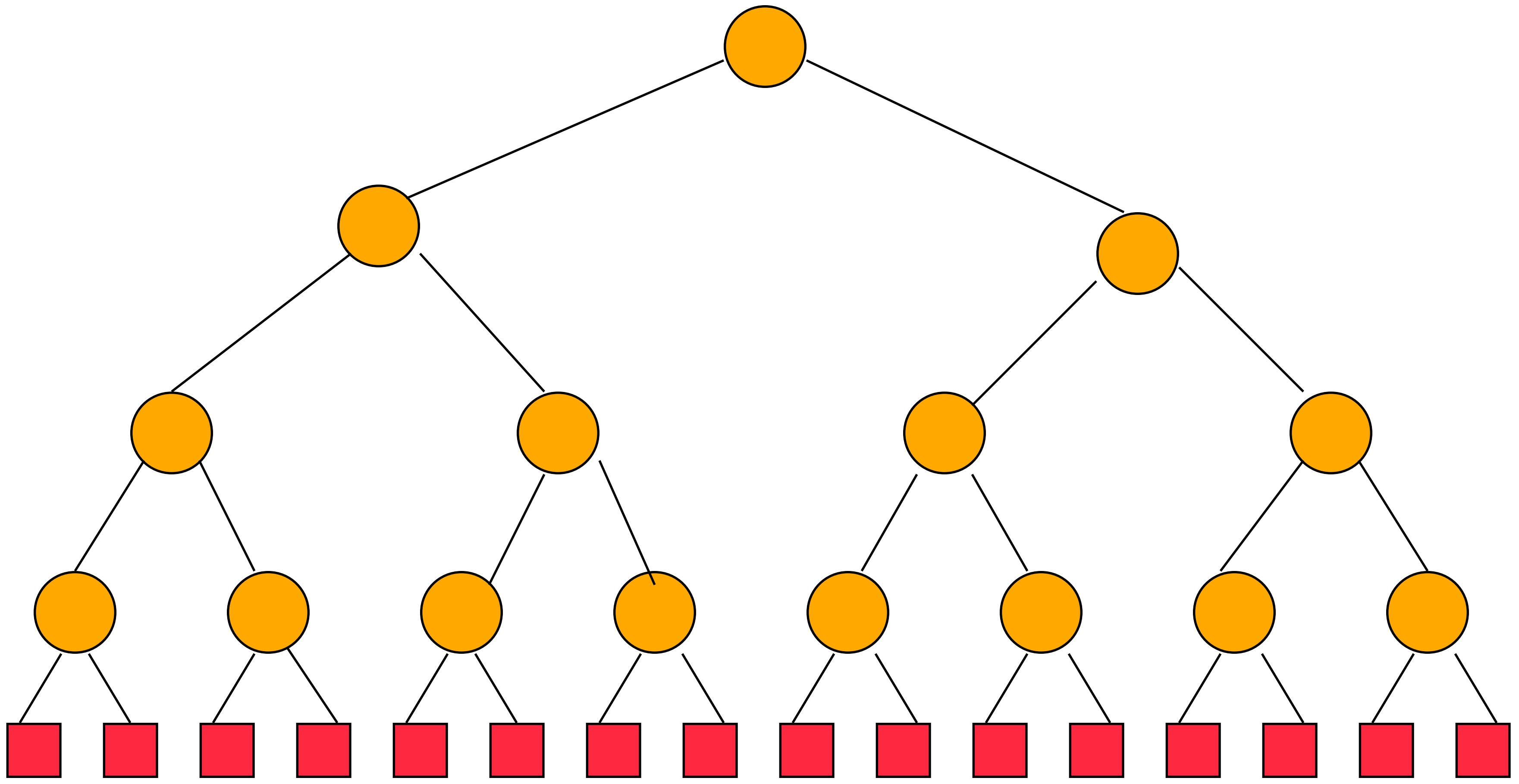
Complete binary tree with n external nodes and n - 1 internal nodes.

External nodes represent tournament players.

Each internal node represents a match played between its two children; the winner of the match is stored at the internal node.

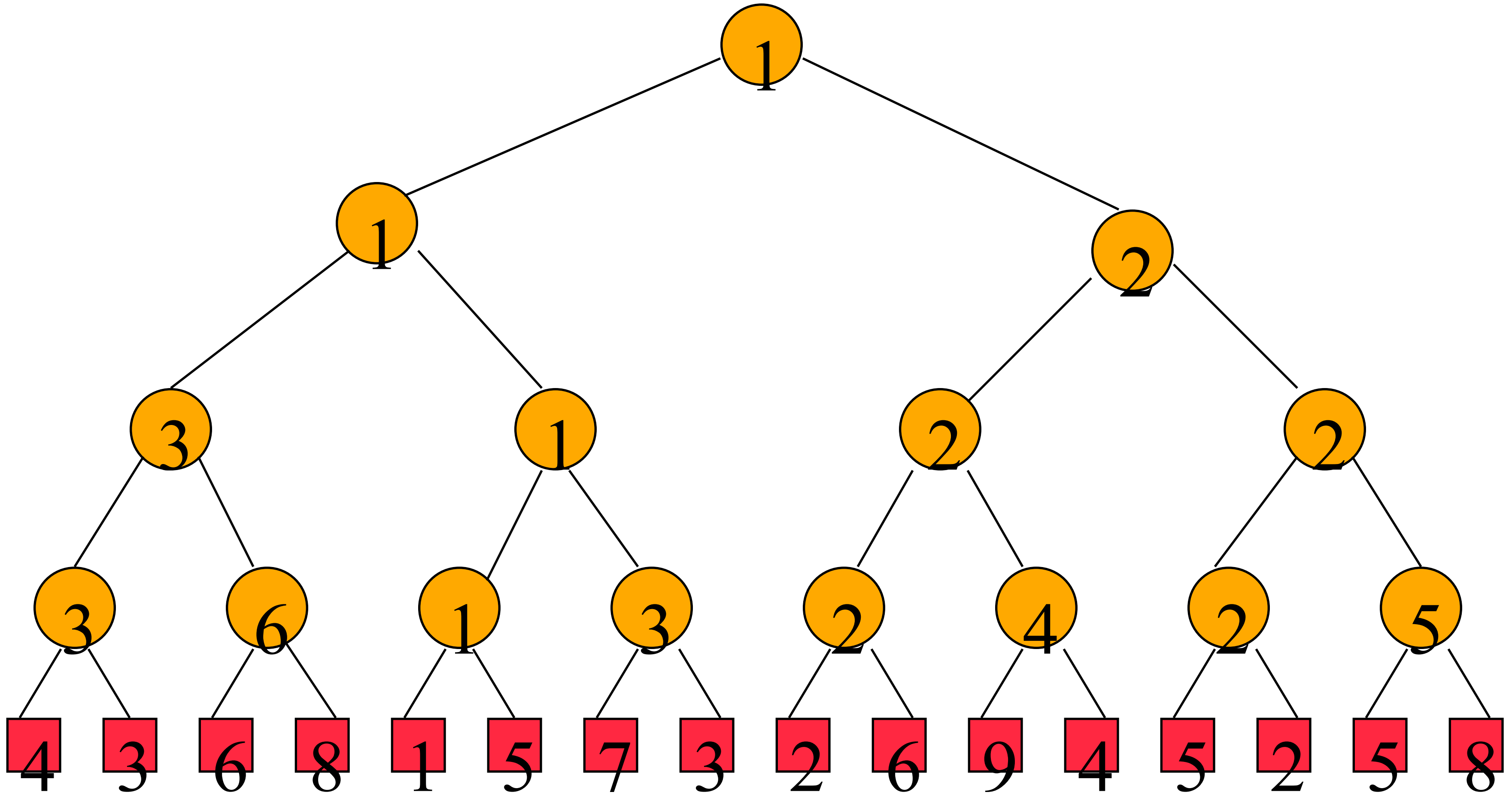
Root has overall winner.

Winner Tree For 16 Players



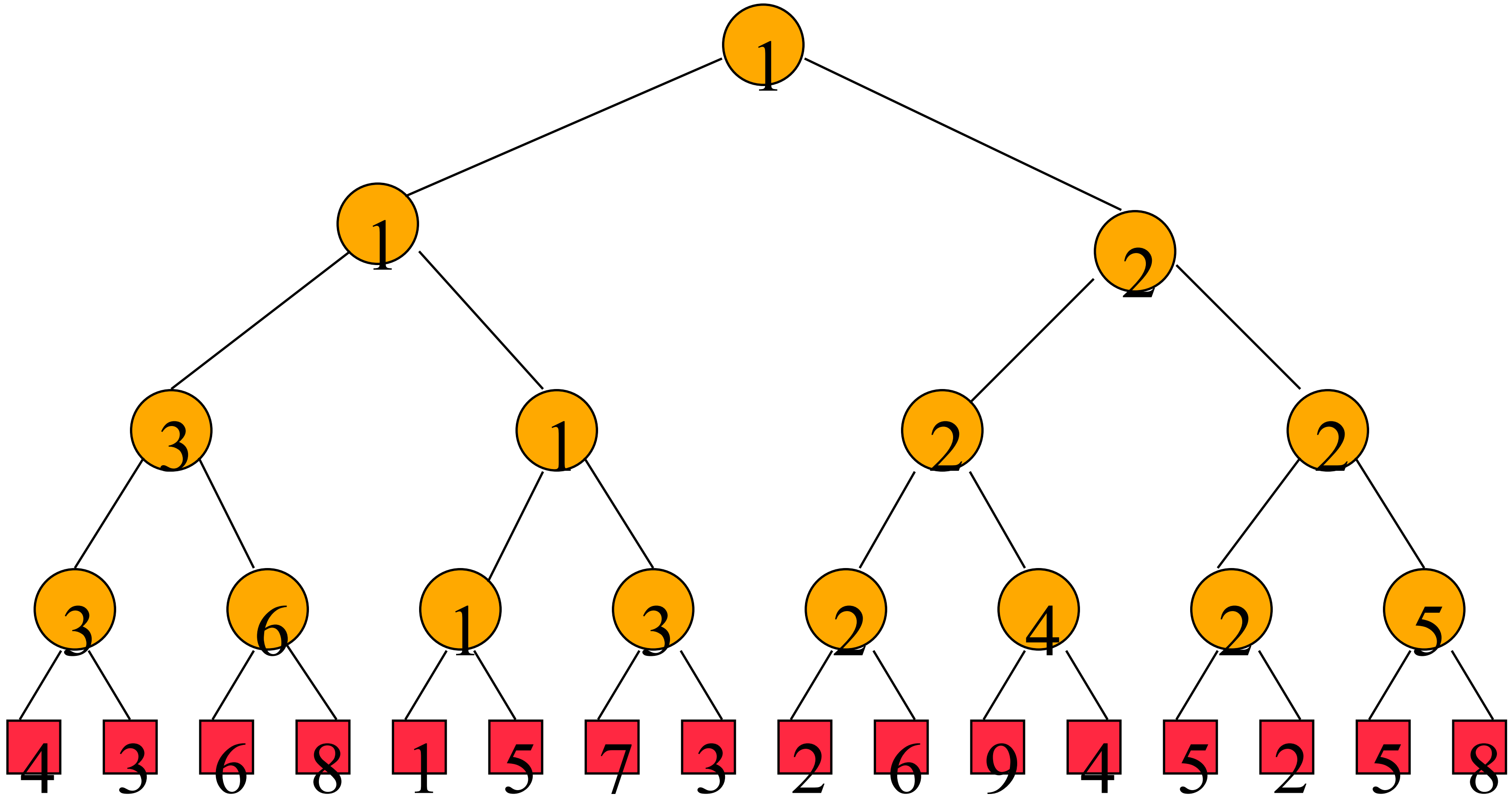
🟥 player ● match node

Winner Tree For 16 Players



Smaller element wins => min winner tree.

Winner Tree For 16 Players



height is $\log_2 n$ (excludes player level)

Complexity Of Initialize

- O(1) time to play match at each match node.
- n - 1 match nodes.
- O(n) time to initialize n player winner tree.

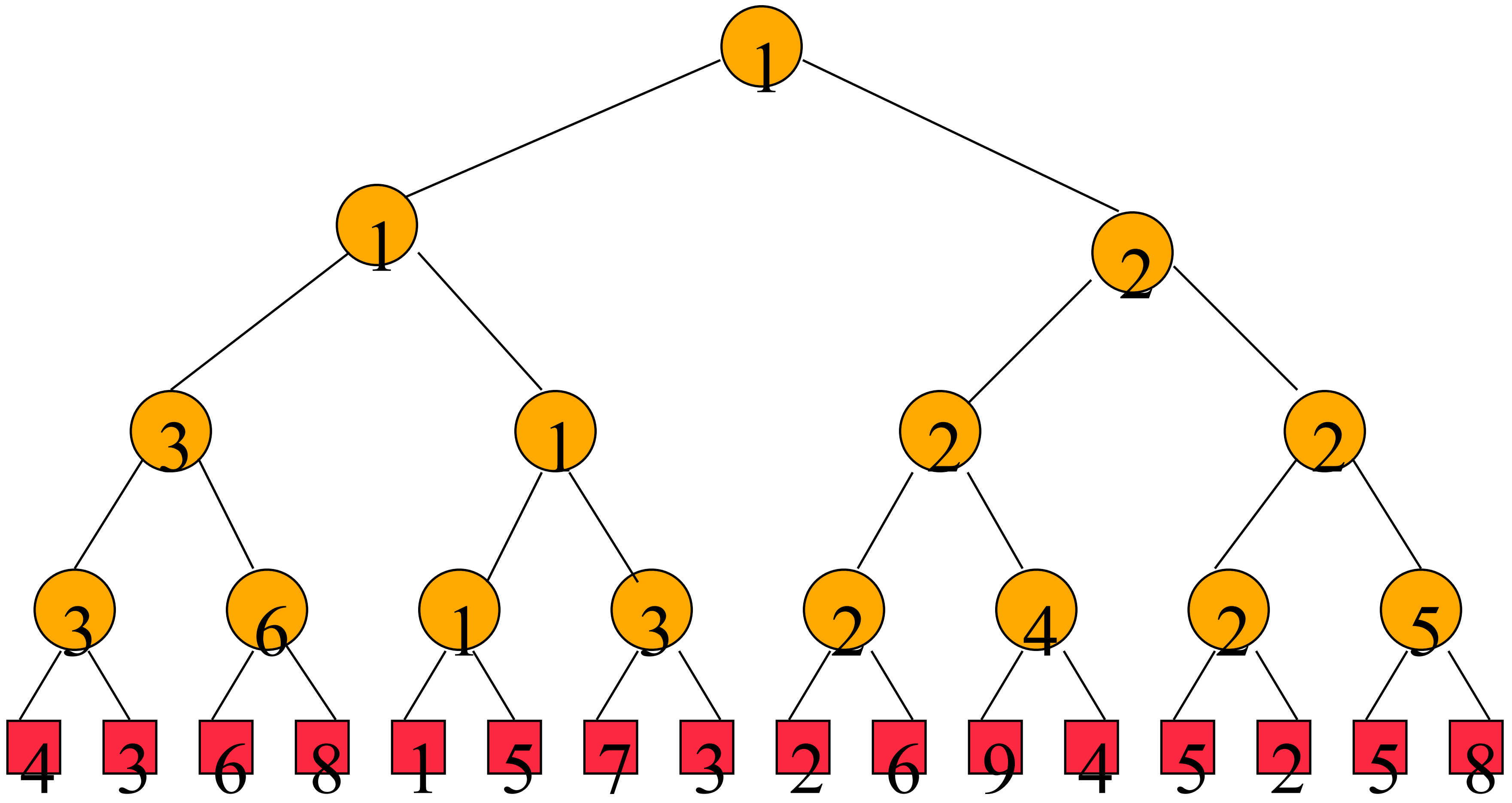
Applications

Sorting.

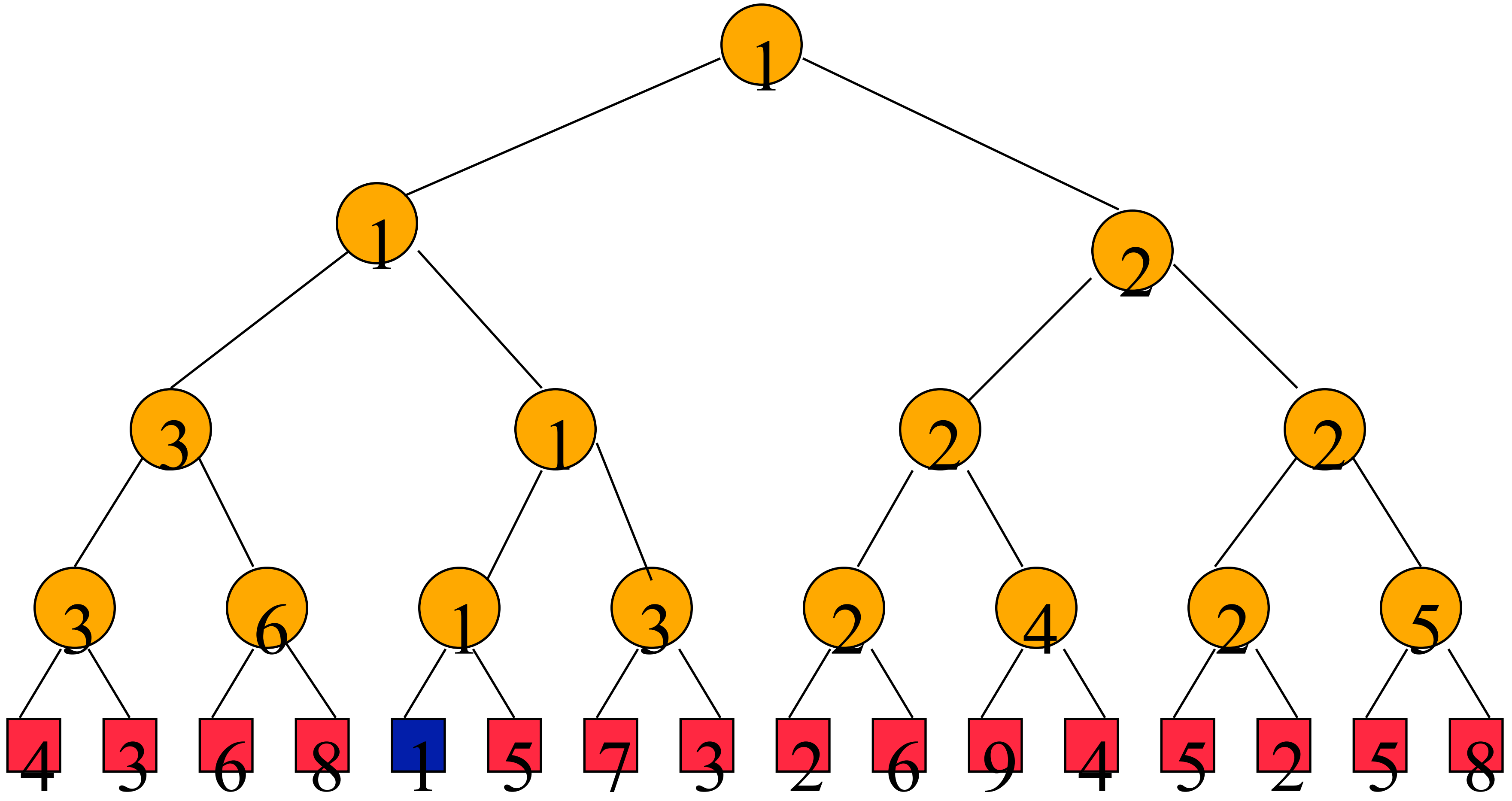
Put elements to be sorted into a winner tree.

Repeatedly extract the winner and replace by a large value.

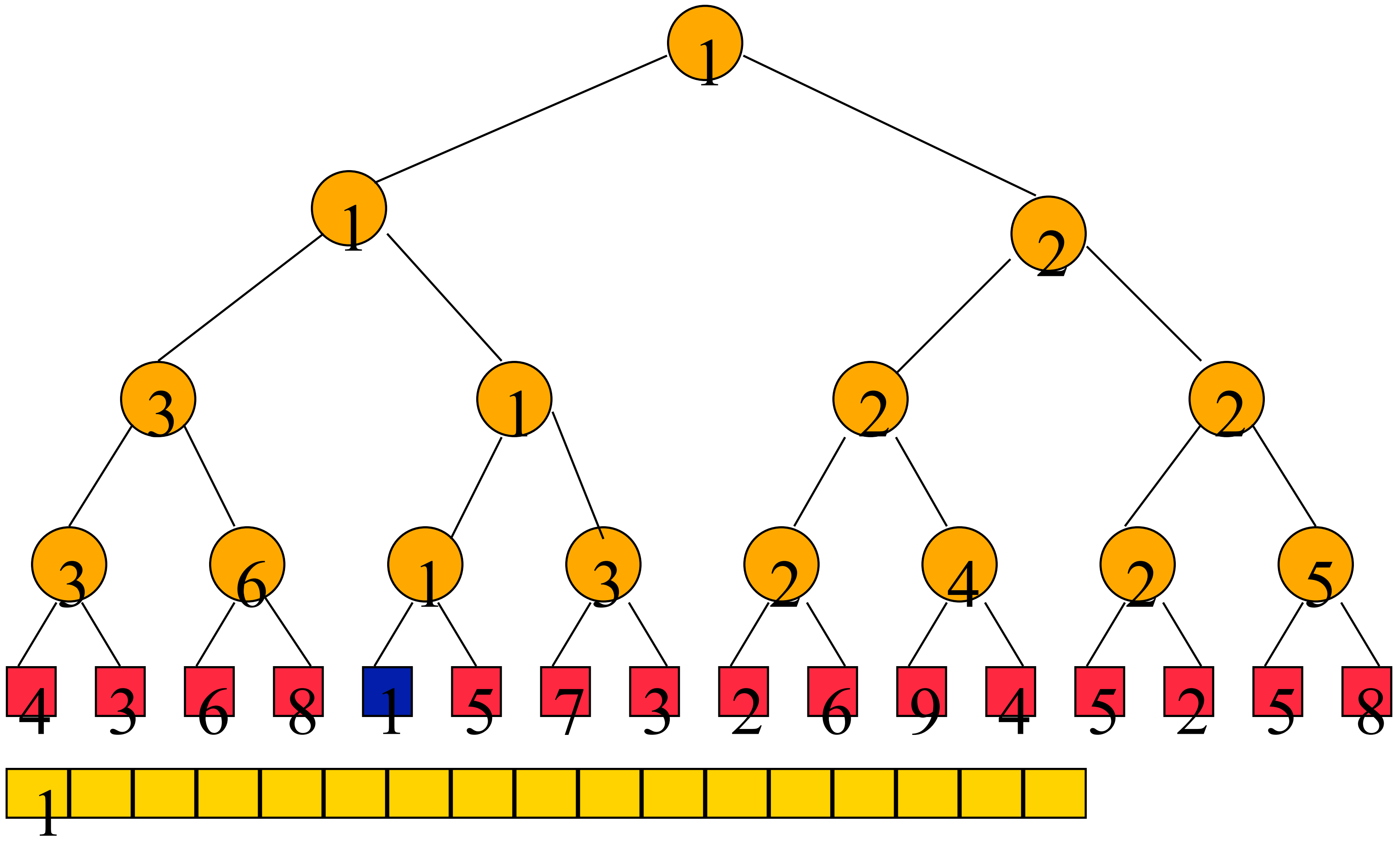
Sort 16 Numbers



Sort 16 Numbers

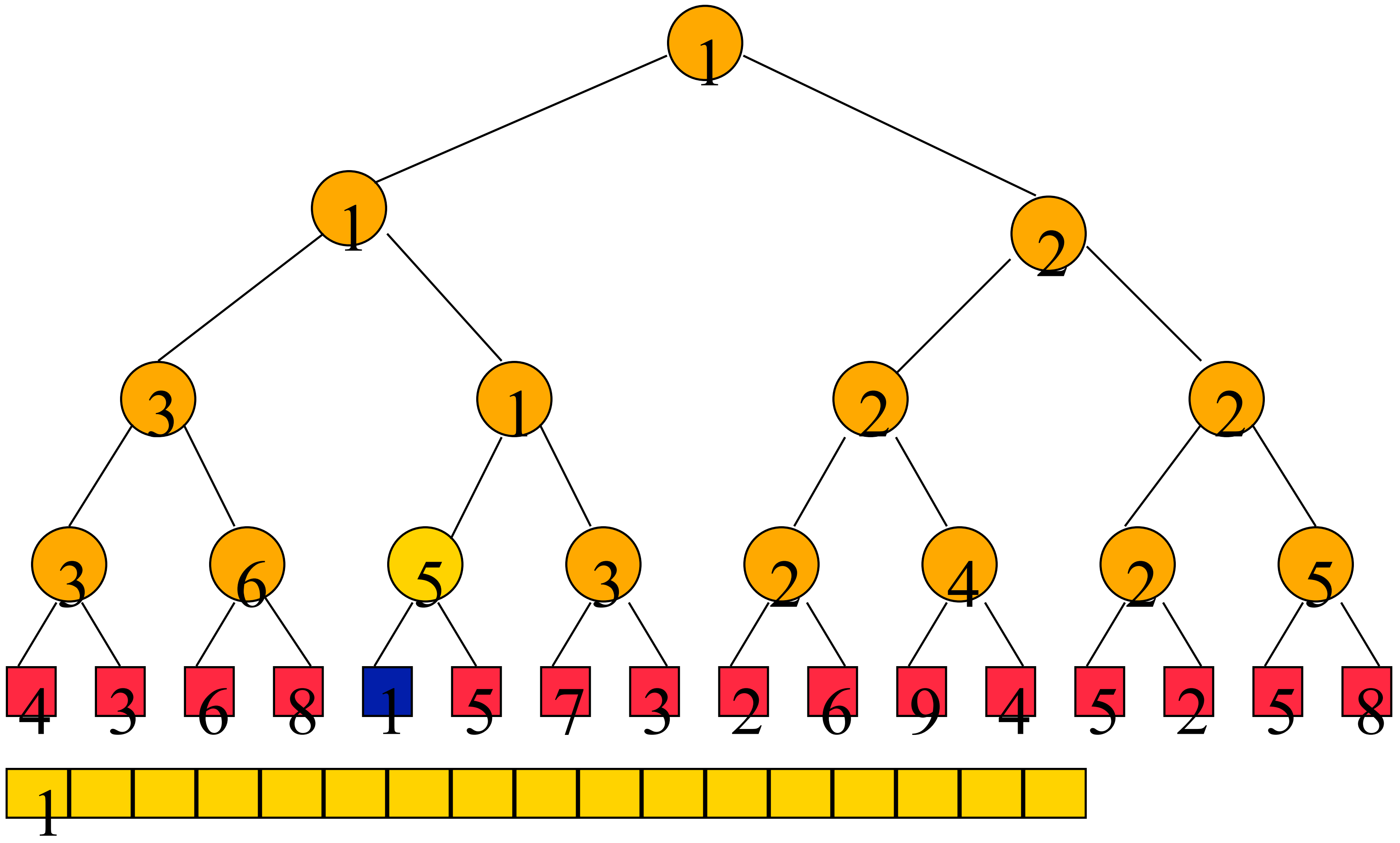


Sort 16 Numbers



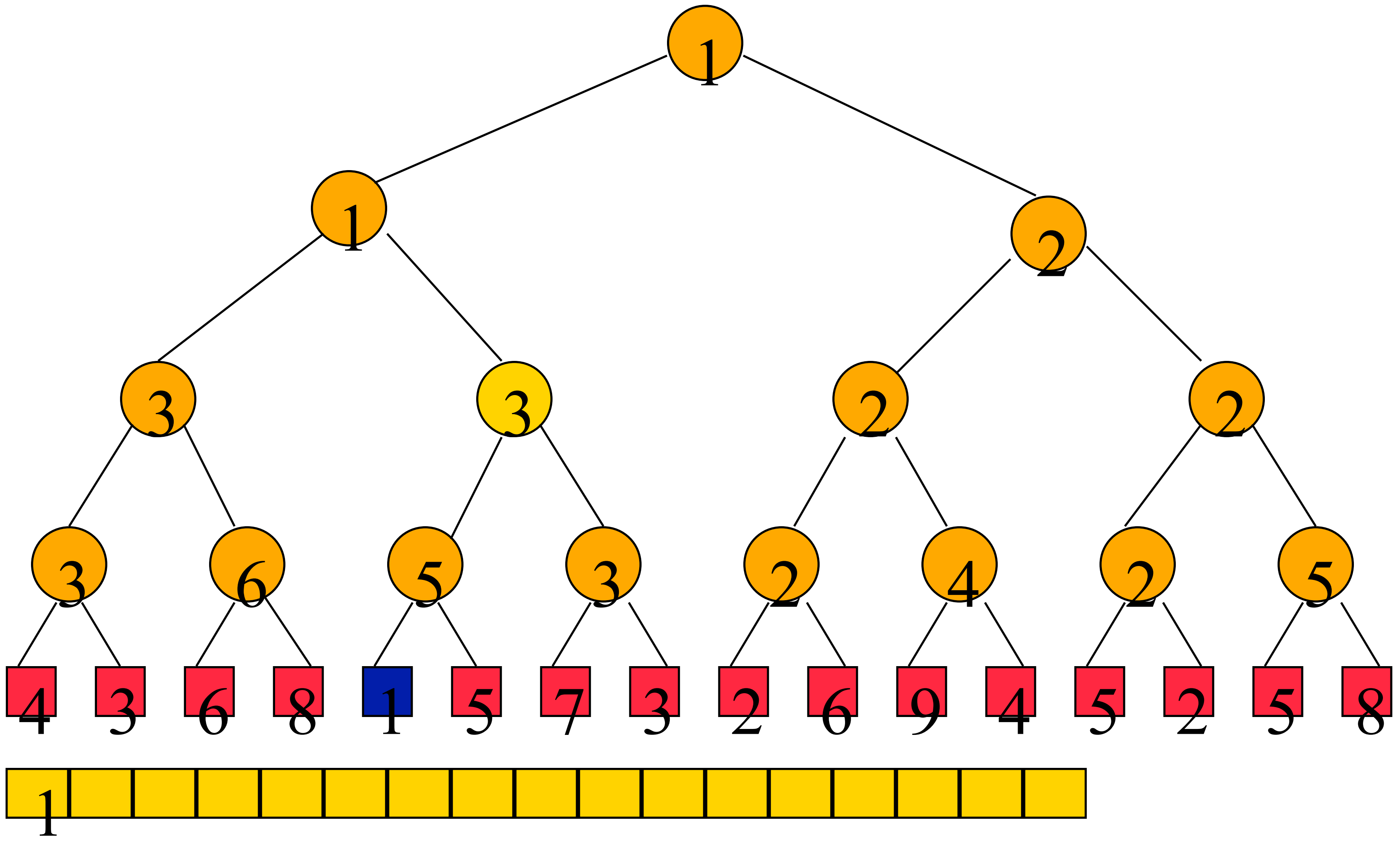
Sorted array.

Sort 16 Numbers



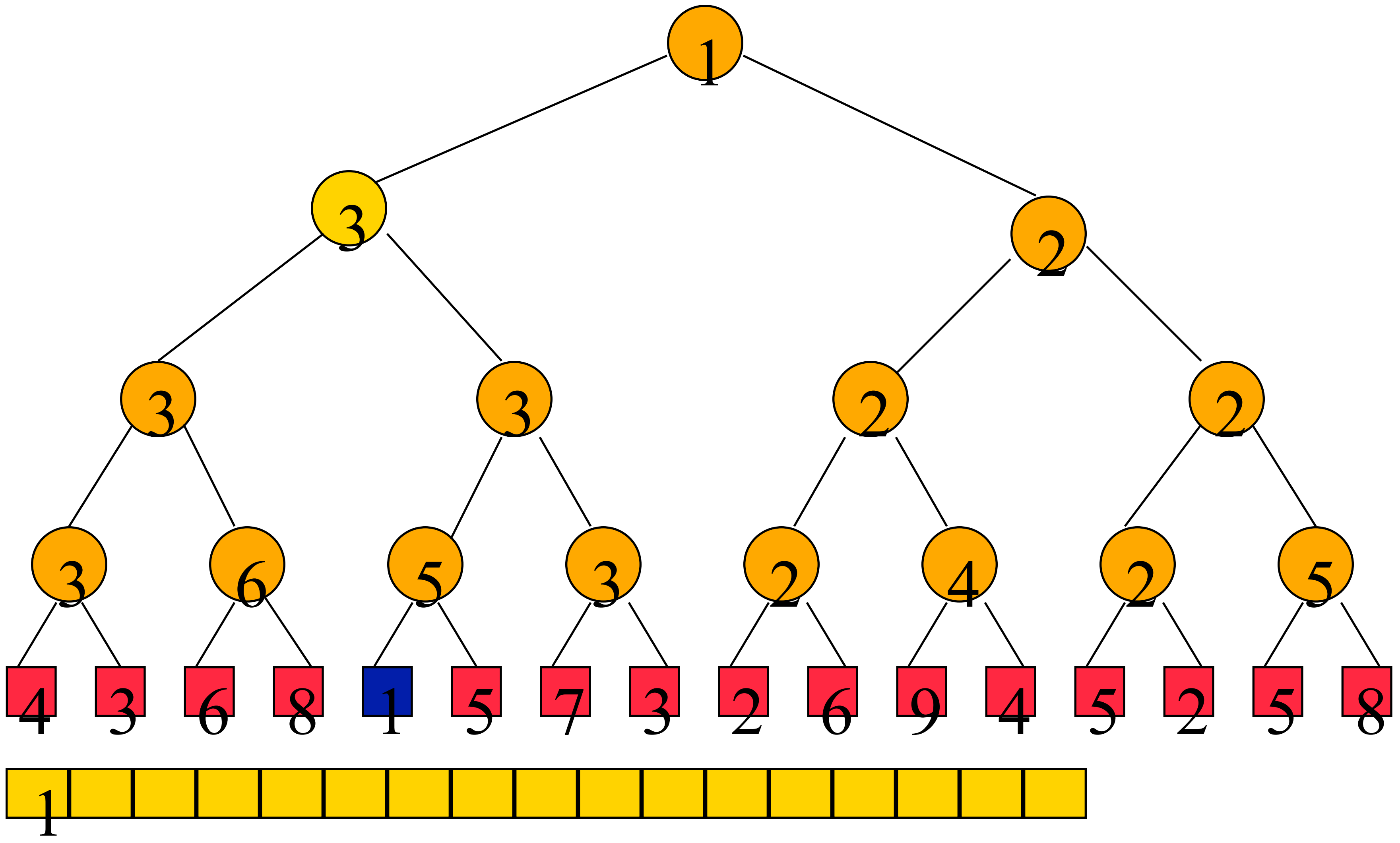
Sorted array.

Sort 16 Numbers



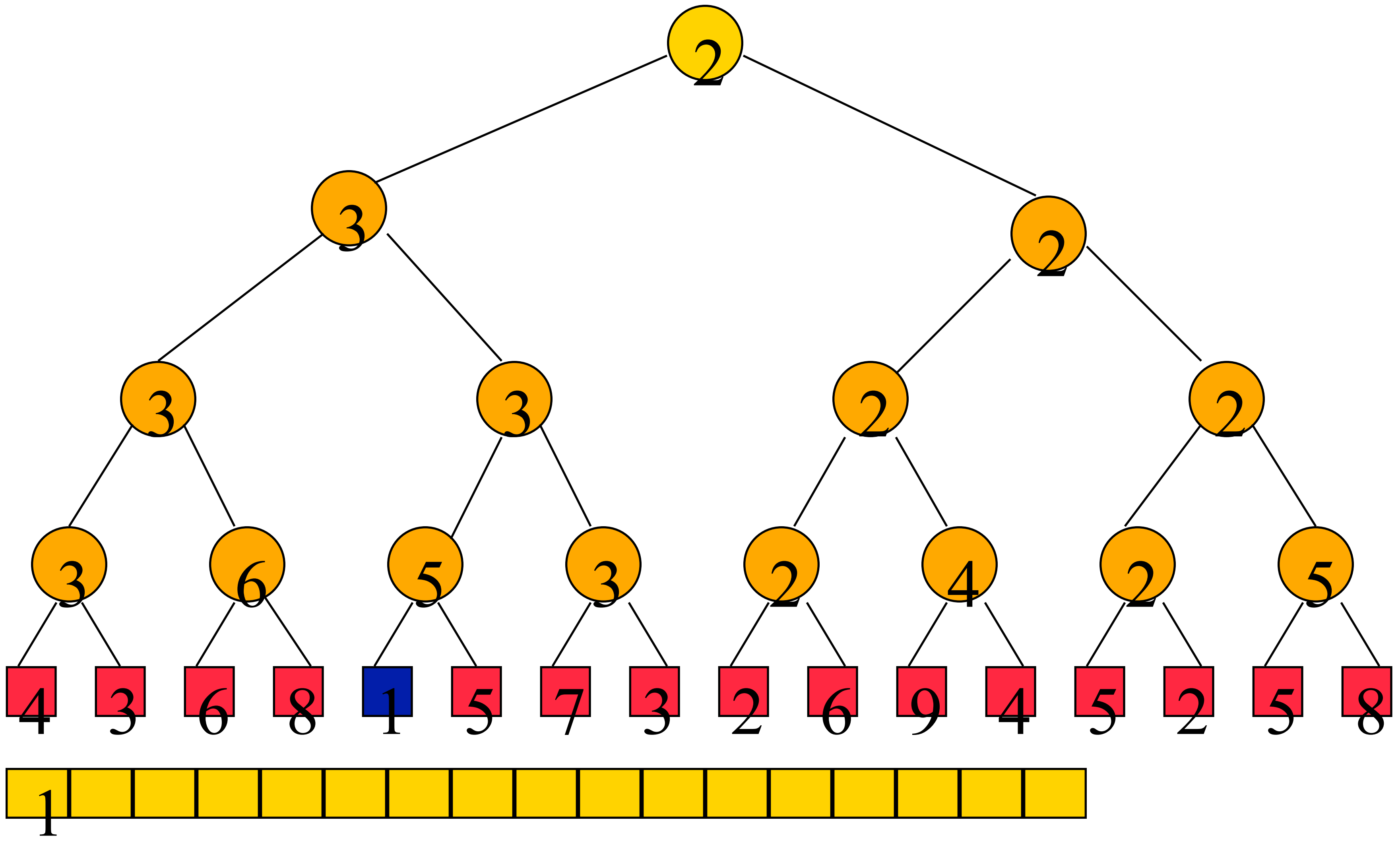
Sorted array.

Sort 16 Numbers



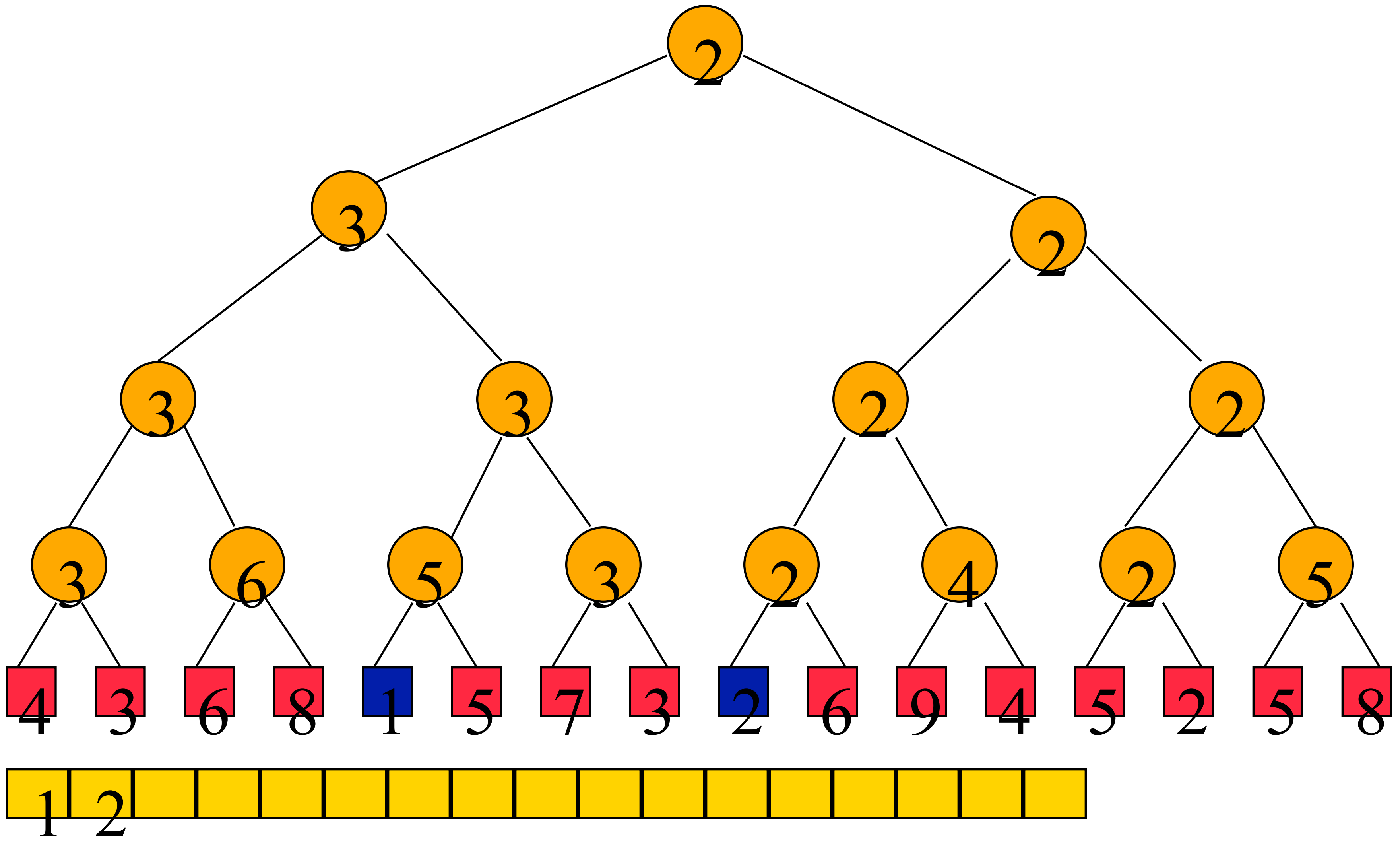
Sorted array.

Sort 16 Numbers



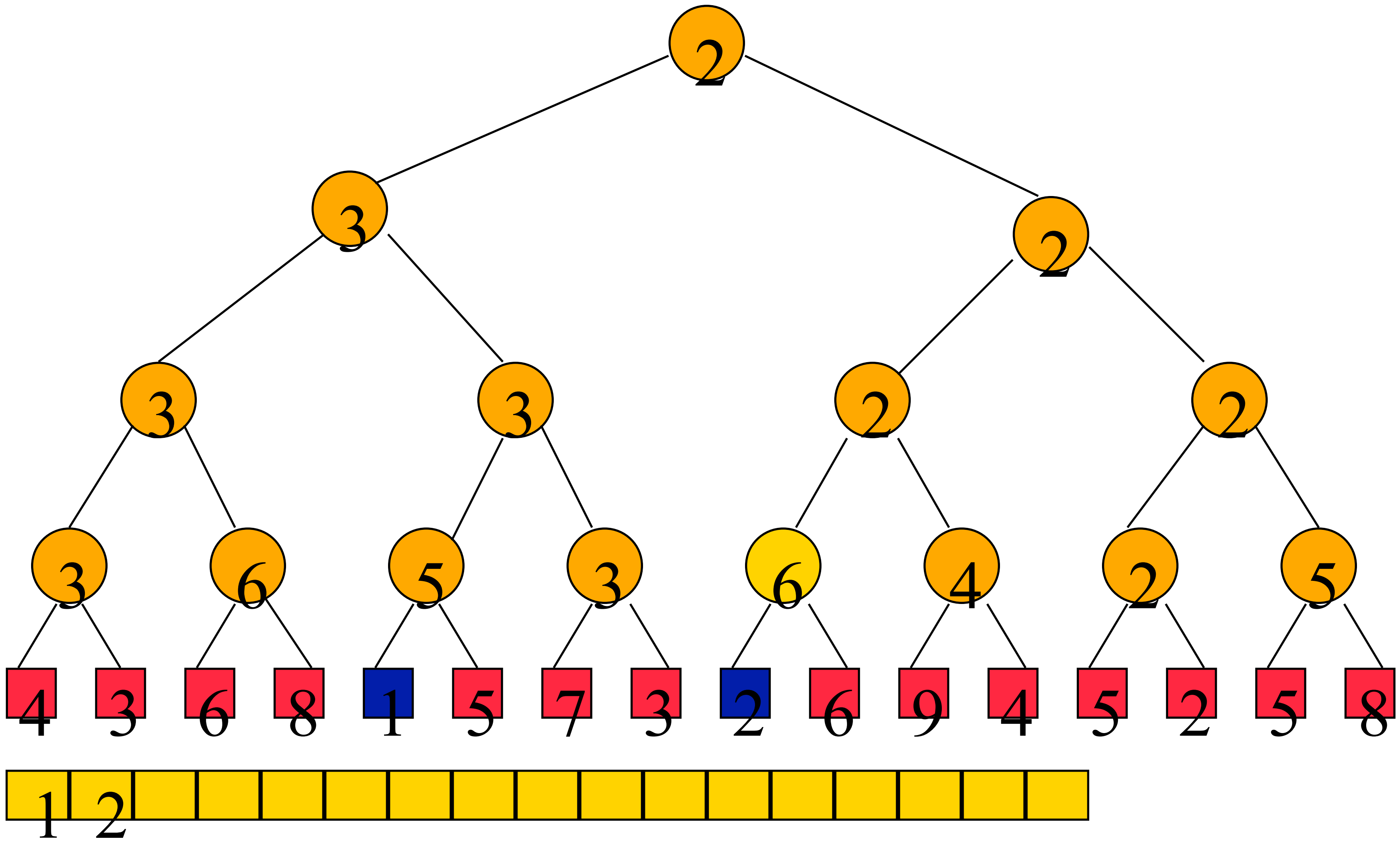
Sorted array.

Sort 16 Numbers



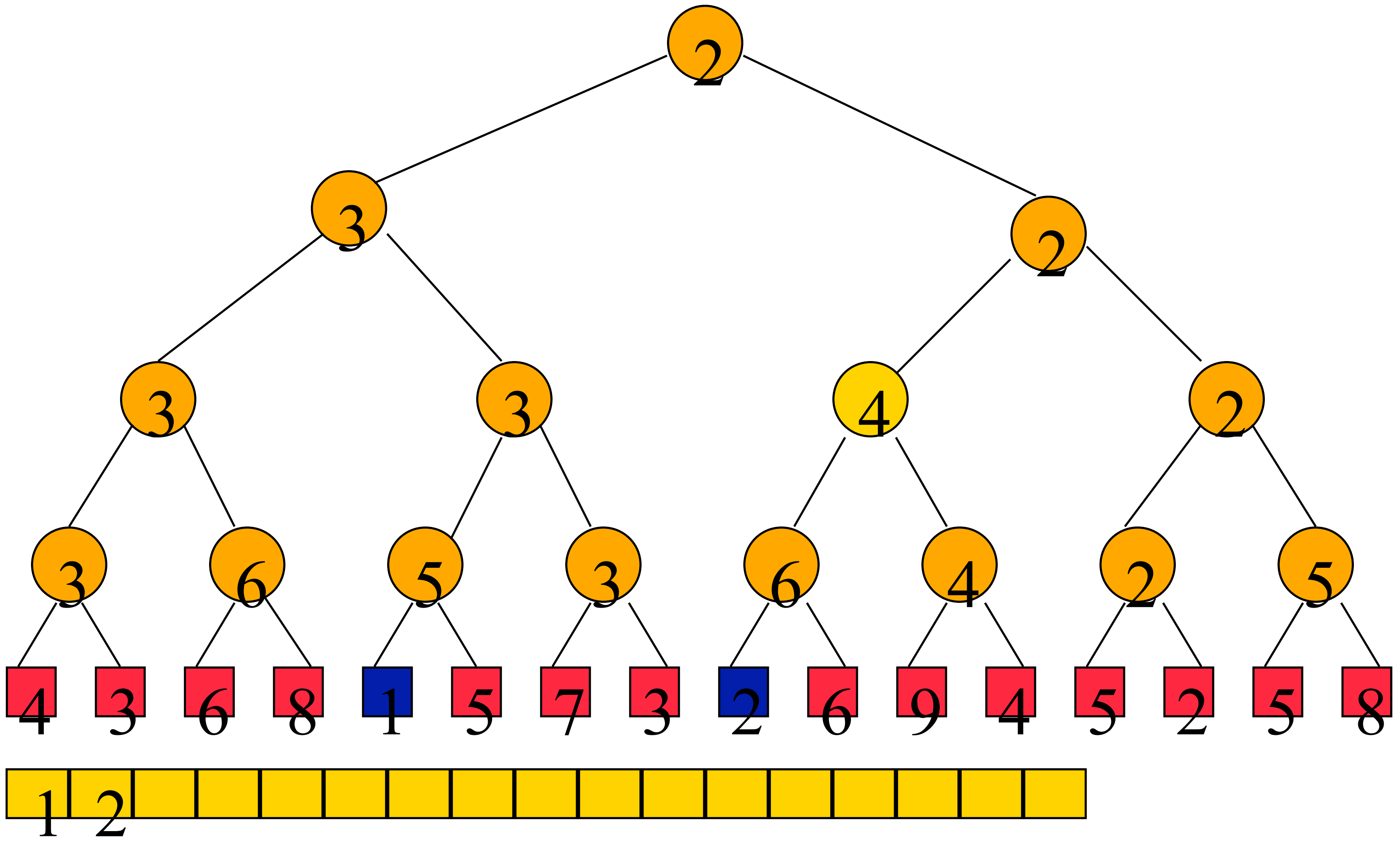
Sorted array.

Sort 16 Numbers



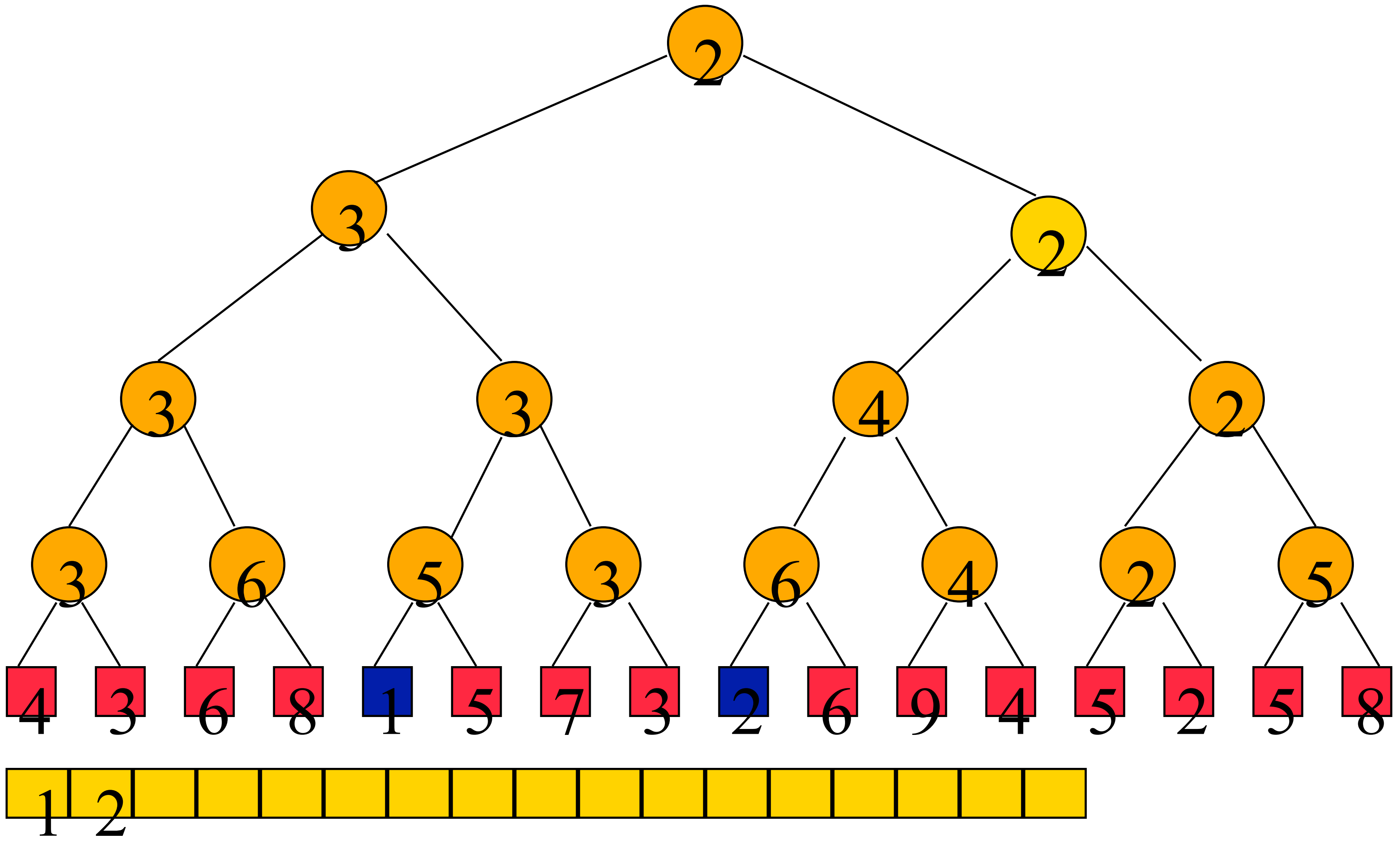
Sorted array.

Sort 16 Numbers



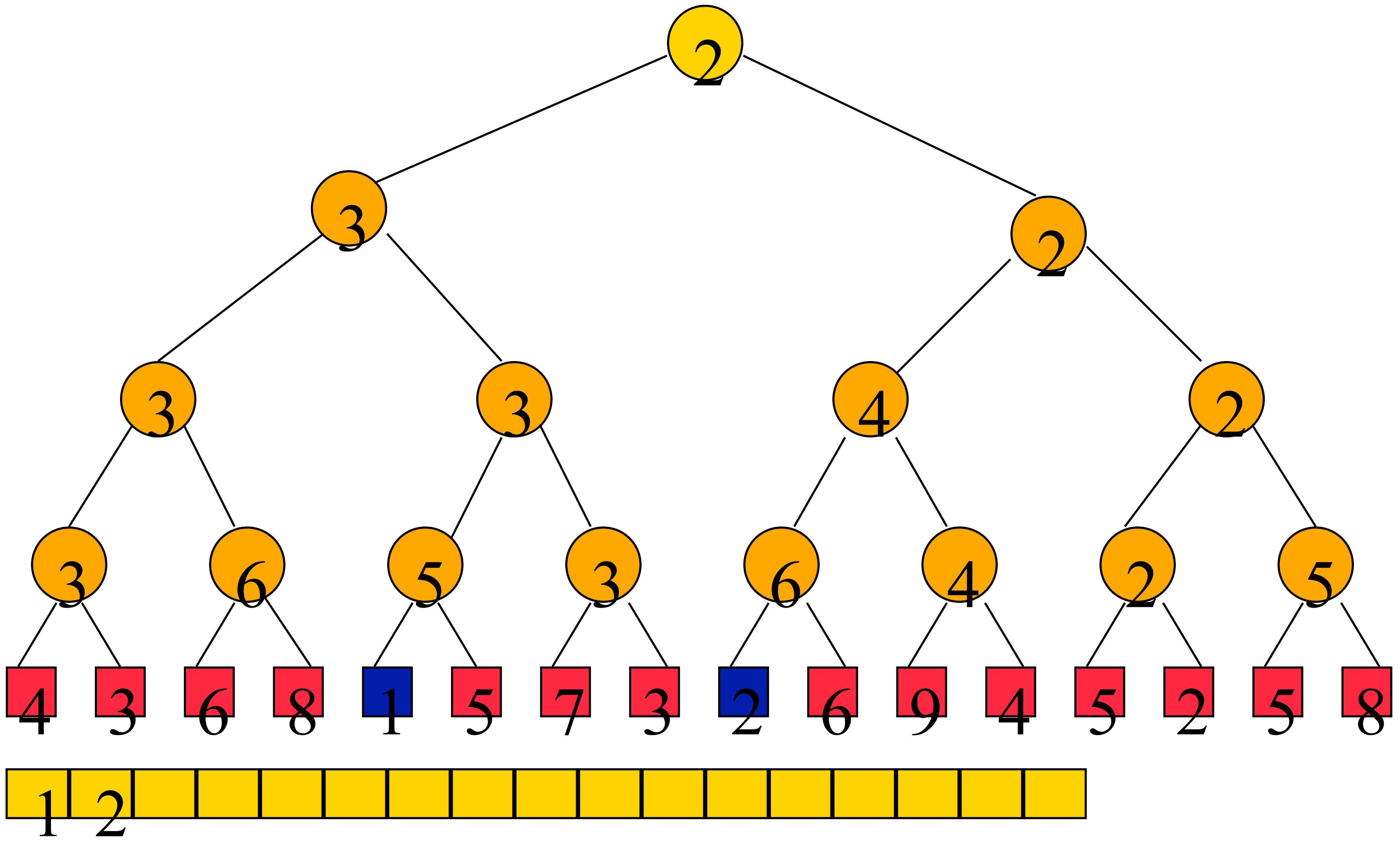
Sorted array.

Sort 16 Numbers



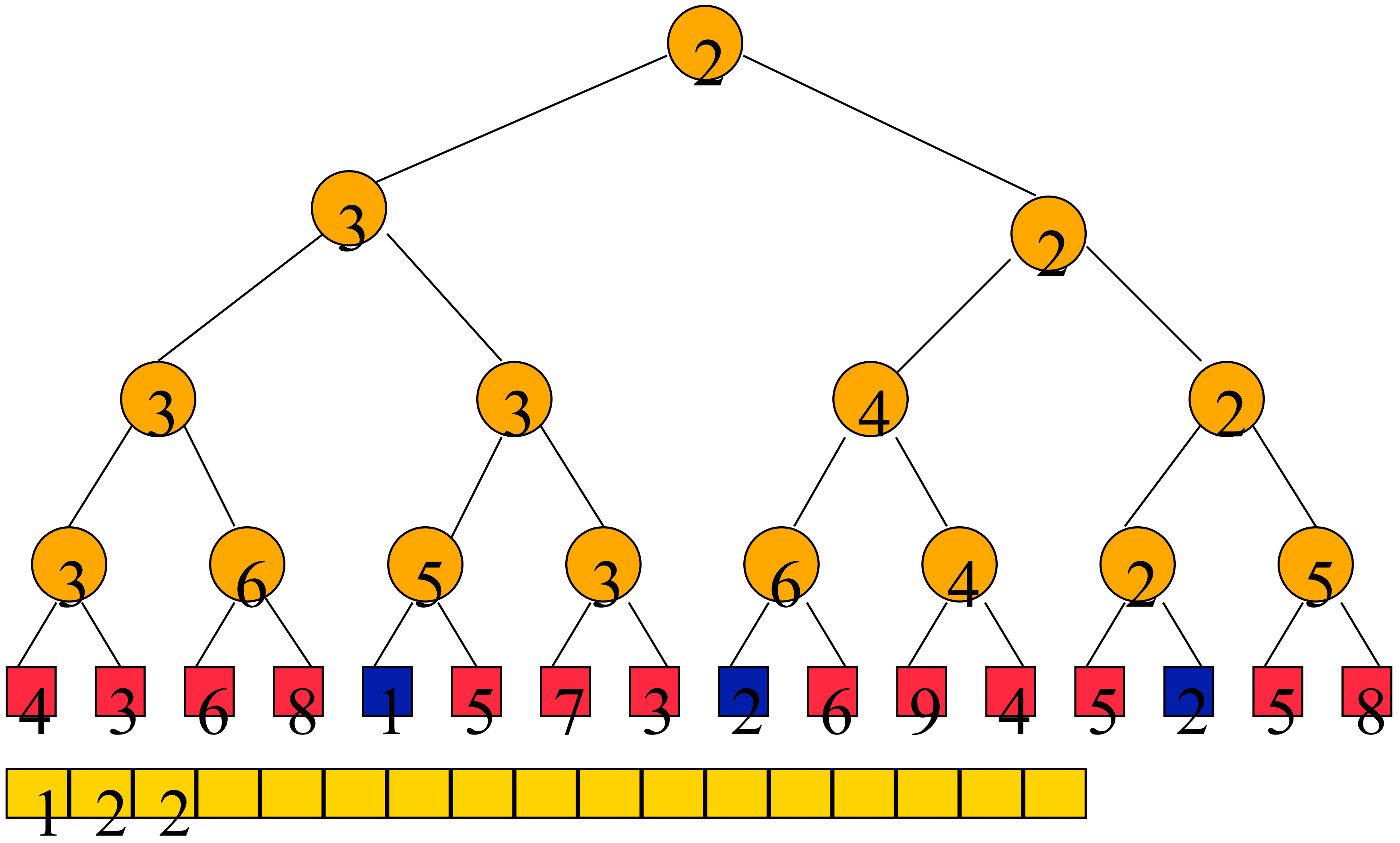
Sorted array.

Sort 16 Numbers



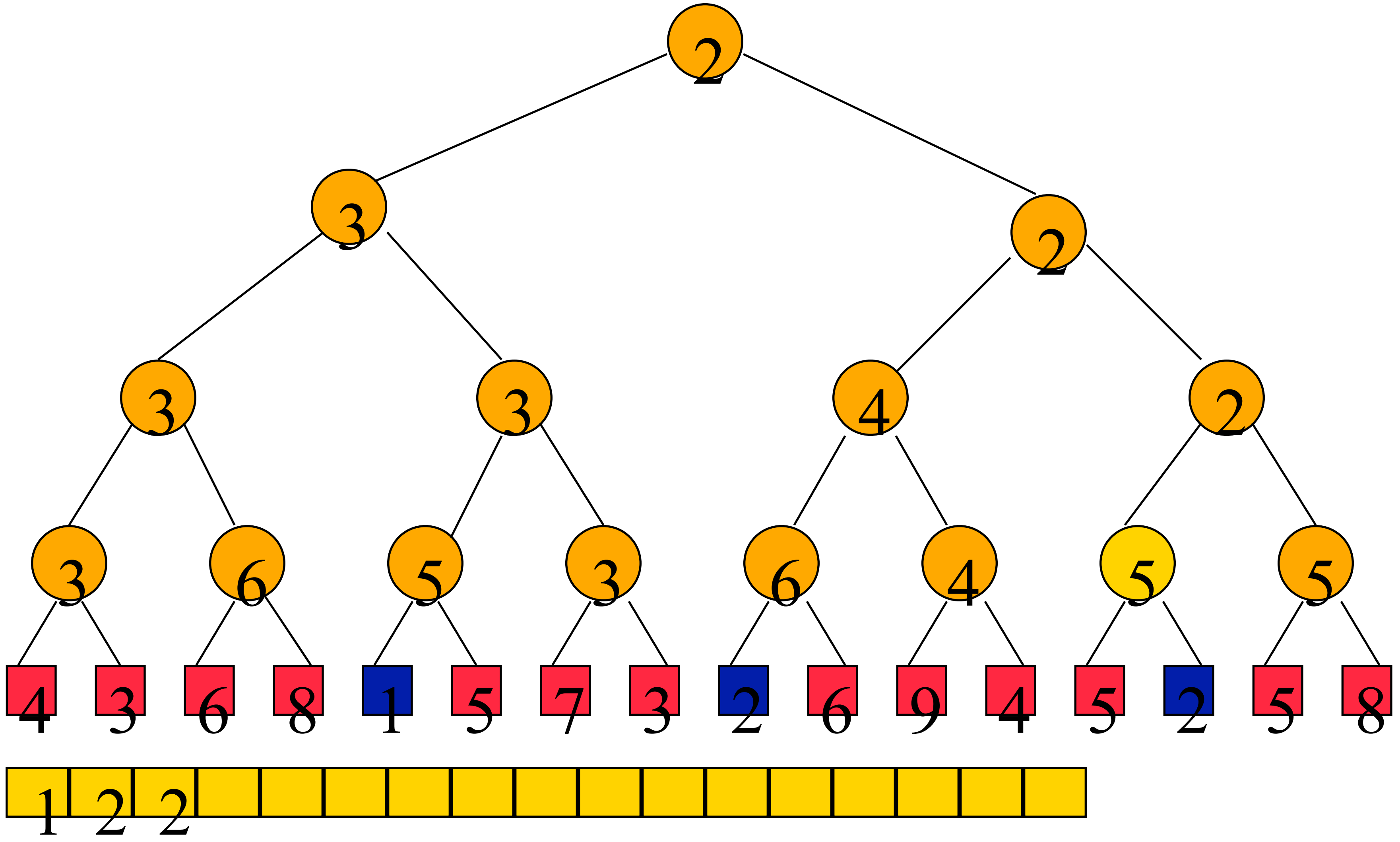
Sorted array.

Sort 16 Numbers



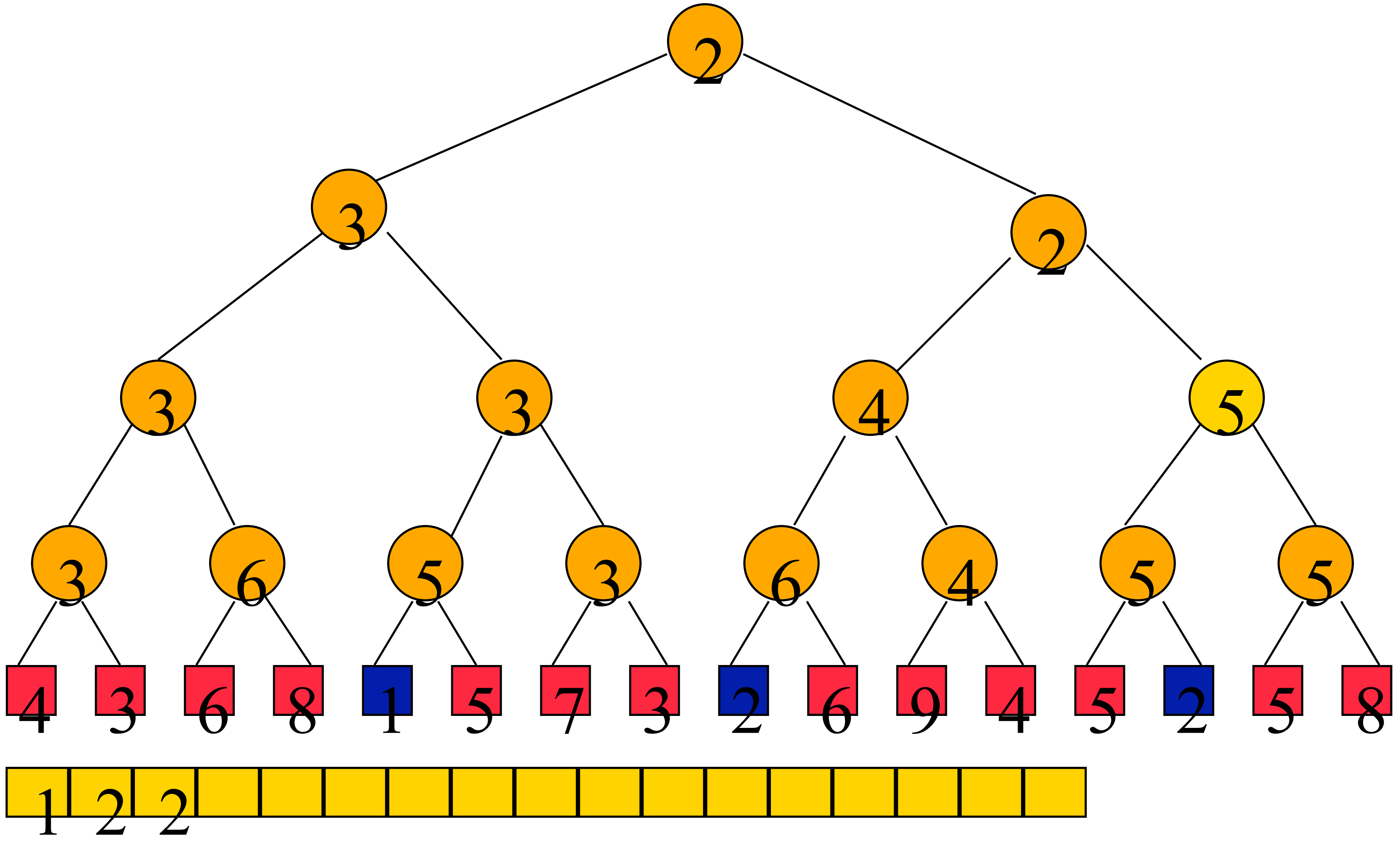
Sorted array.

Sort 16 Numbers



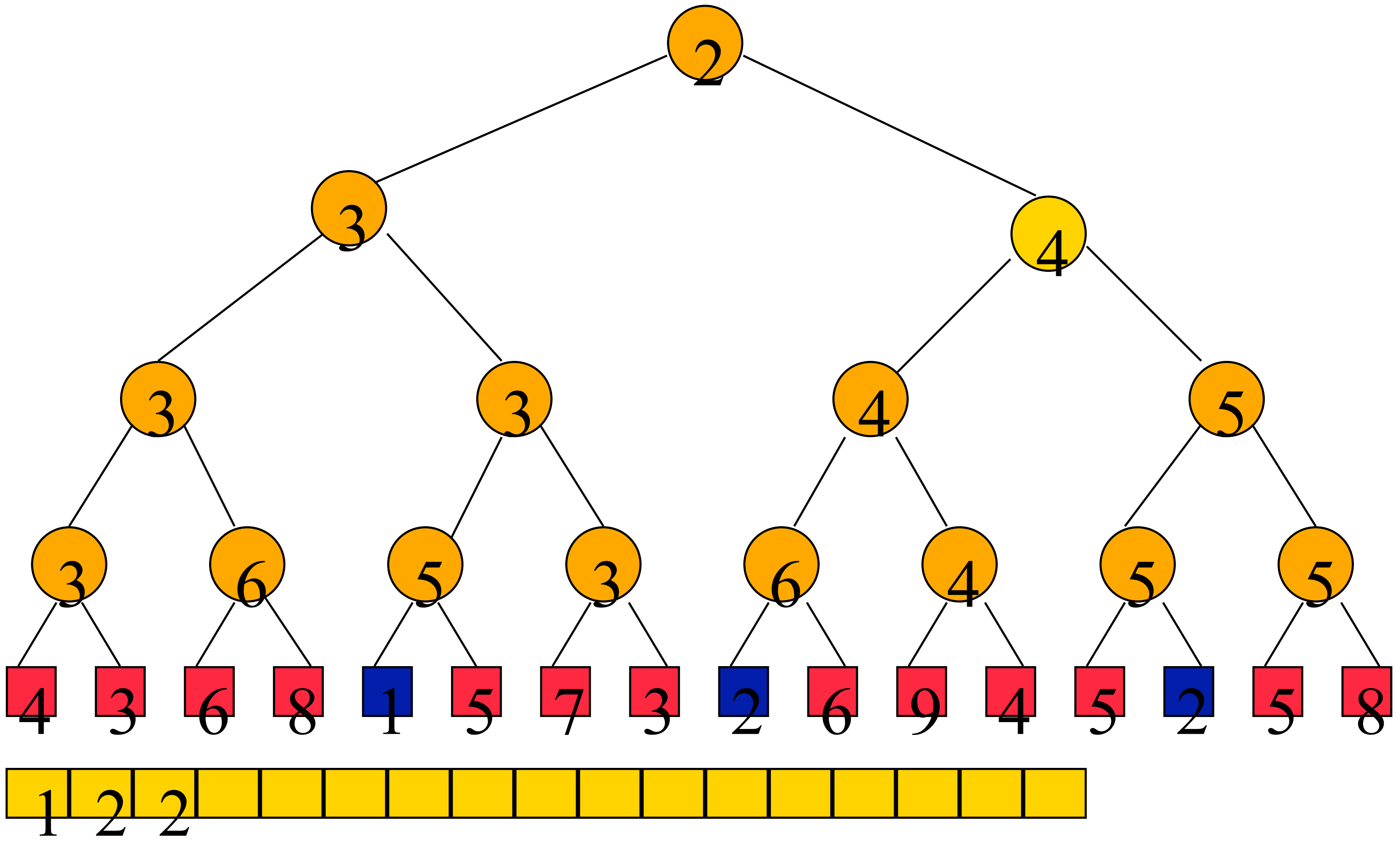
Sorted array.

Sort 16 Numbers



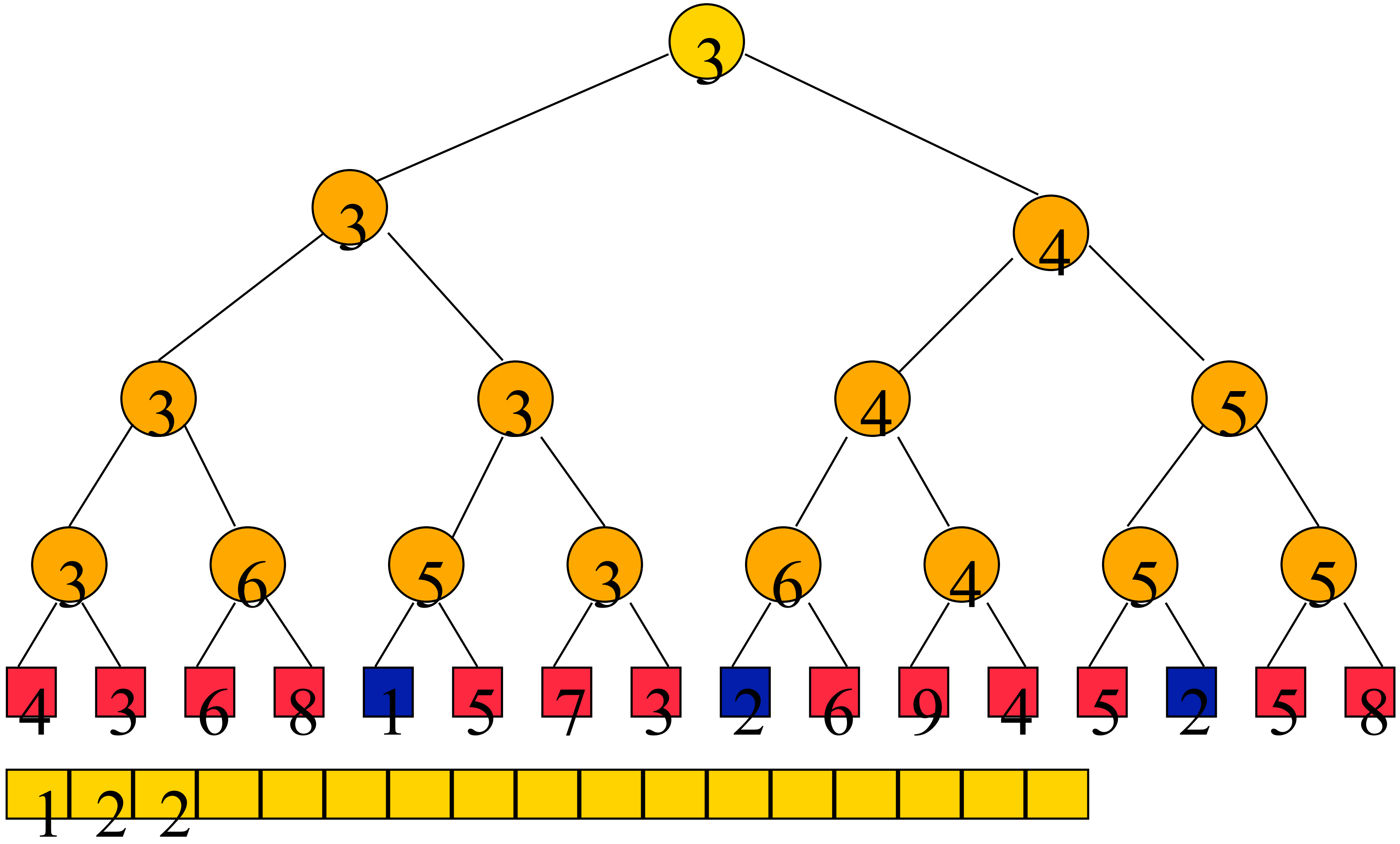
Sorted array.

Sort 16 Numbers



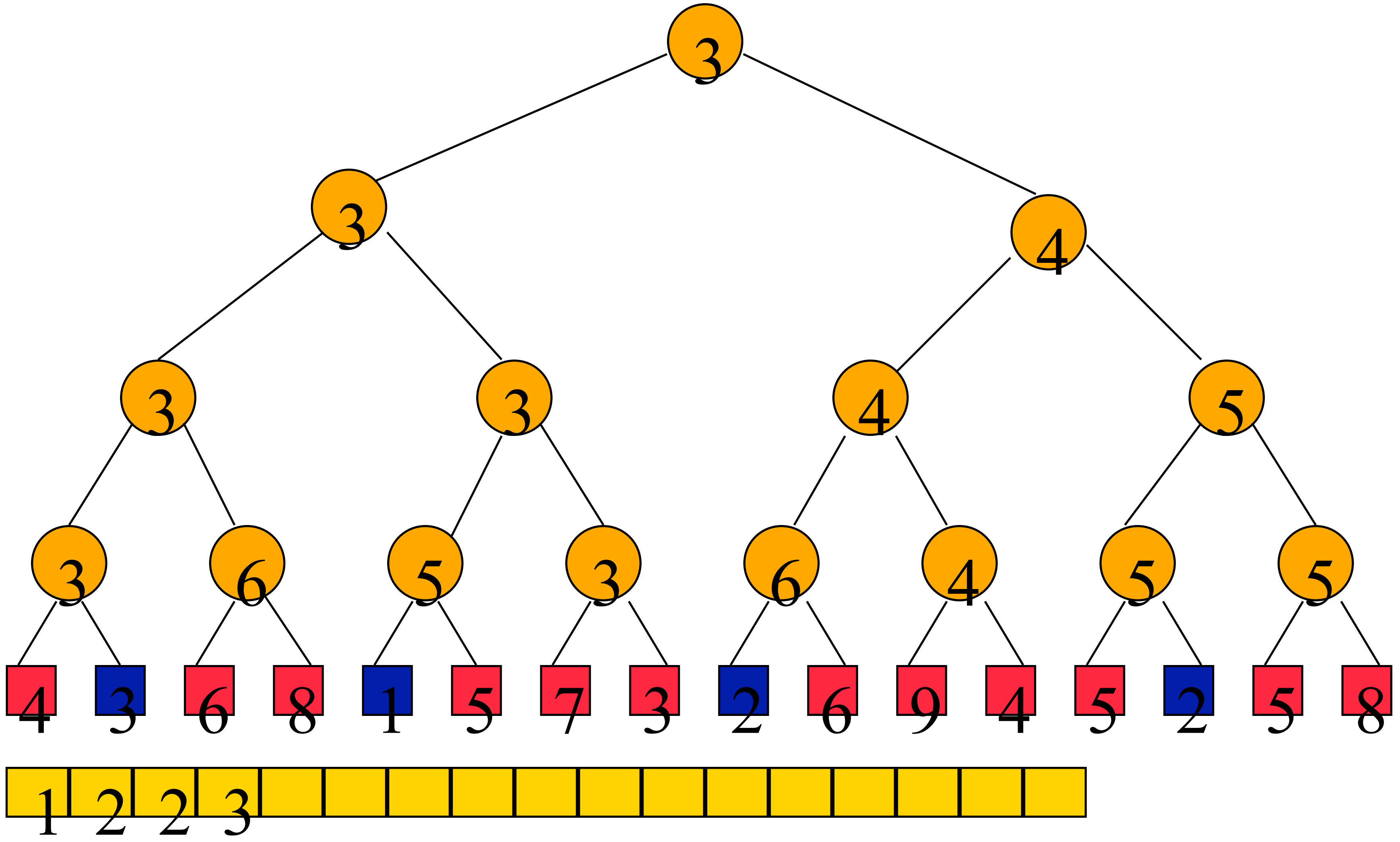
Sorted array.

Sort 16 Numbers



Sorted array.

Sort 16 Numbers



Sorted array.

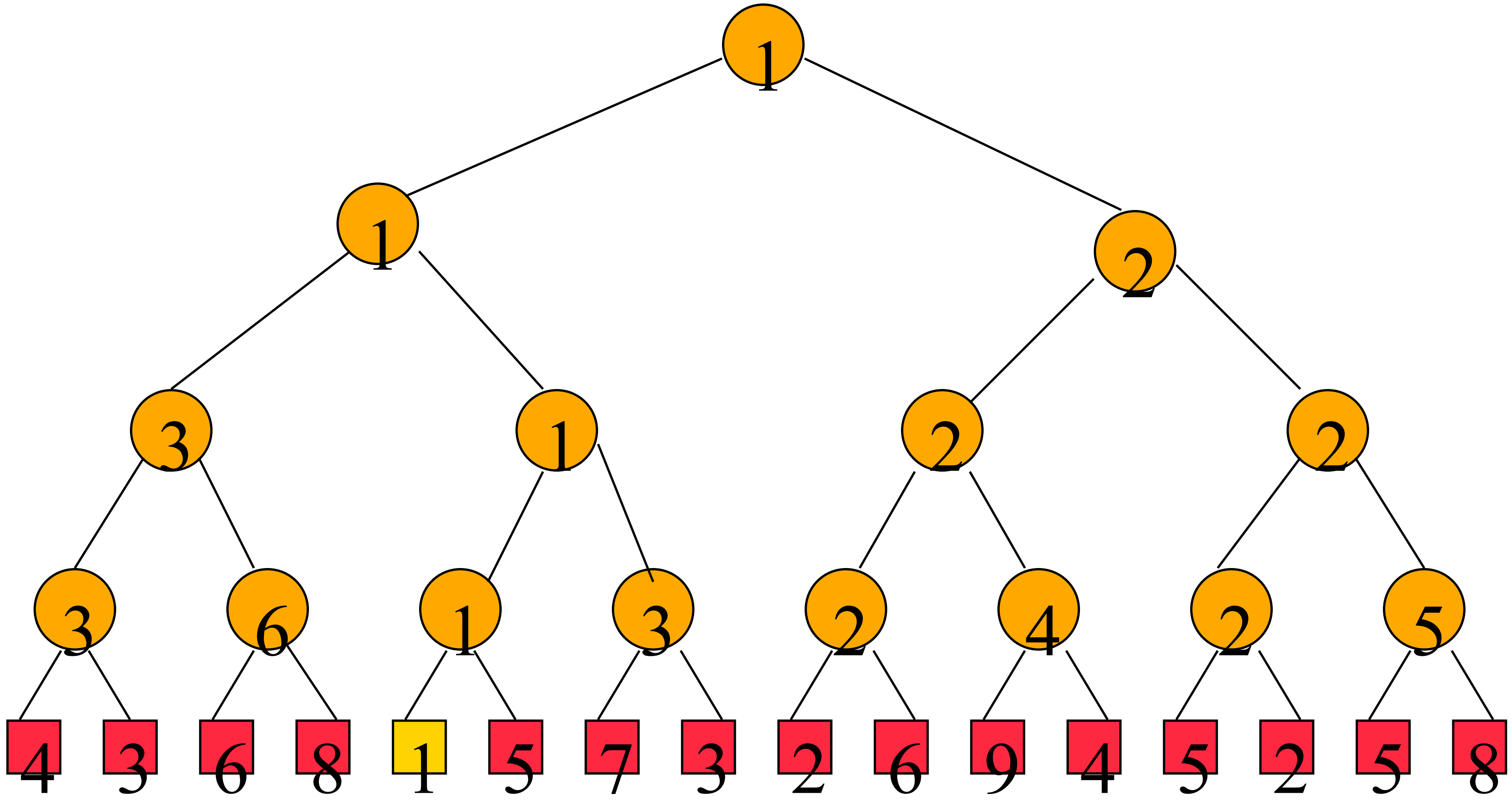
Time To Sort

- Initialize winner tree.
  - O(n) time
- Remove winner and replay.
  - O(log n) time
- Remove winner and replay n times.
  - O(n log n) time
- Total sort time is O(n log n).

Winner Tree Operations

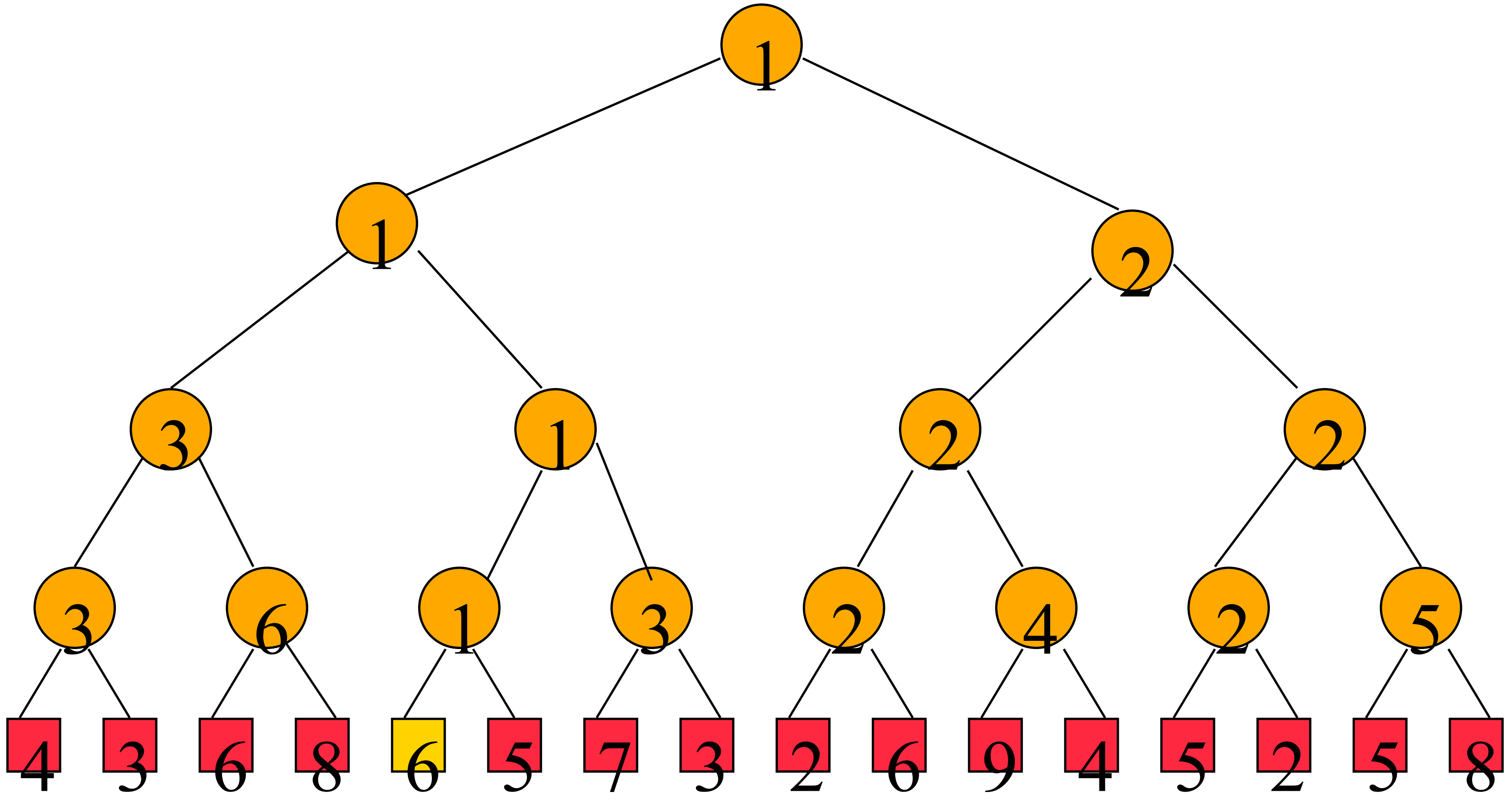
- Initialize
  - O(n) time
- Get winner
  - O(1) time
- Remove/replace winner and replay
  - O(log n) time
  - more precisely Theta(log n)

Replace Winner And Replay



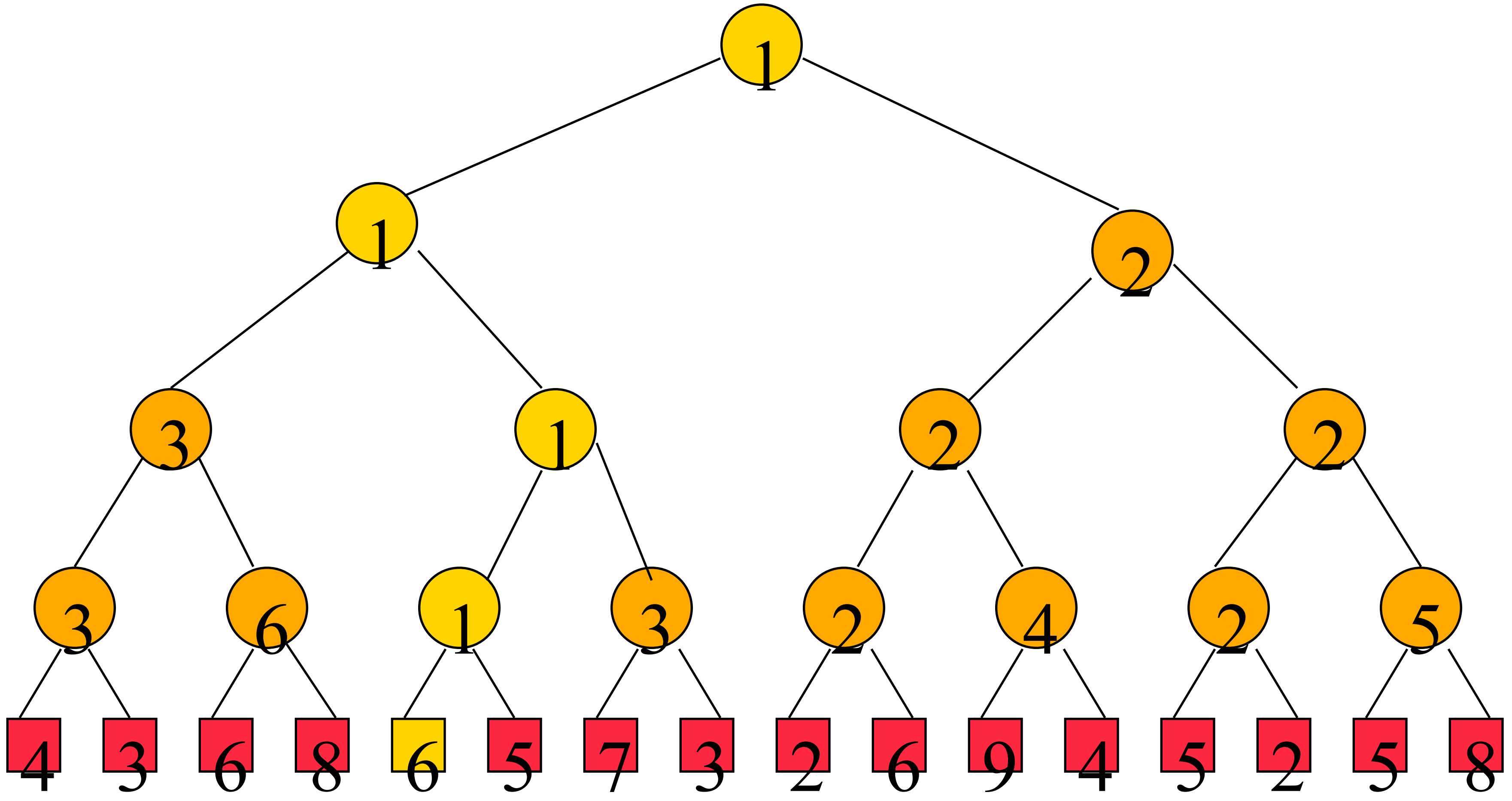
Replace winner with 6.

Replace Winner And Replay



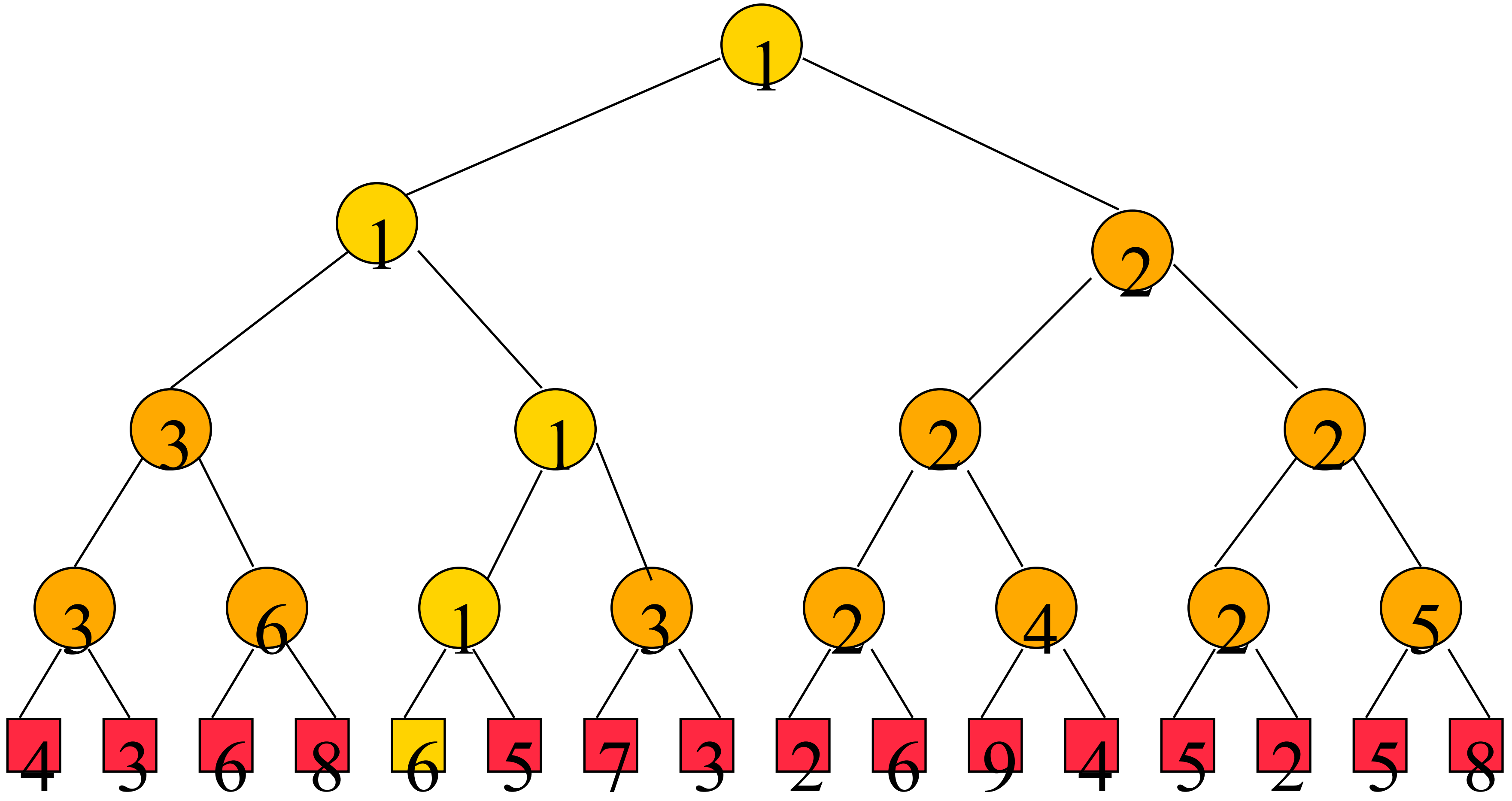
Replay matches on path to root.

Replace Winner And Replay



Replay matches on path to root.

Replace Winner And Replay

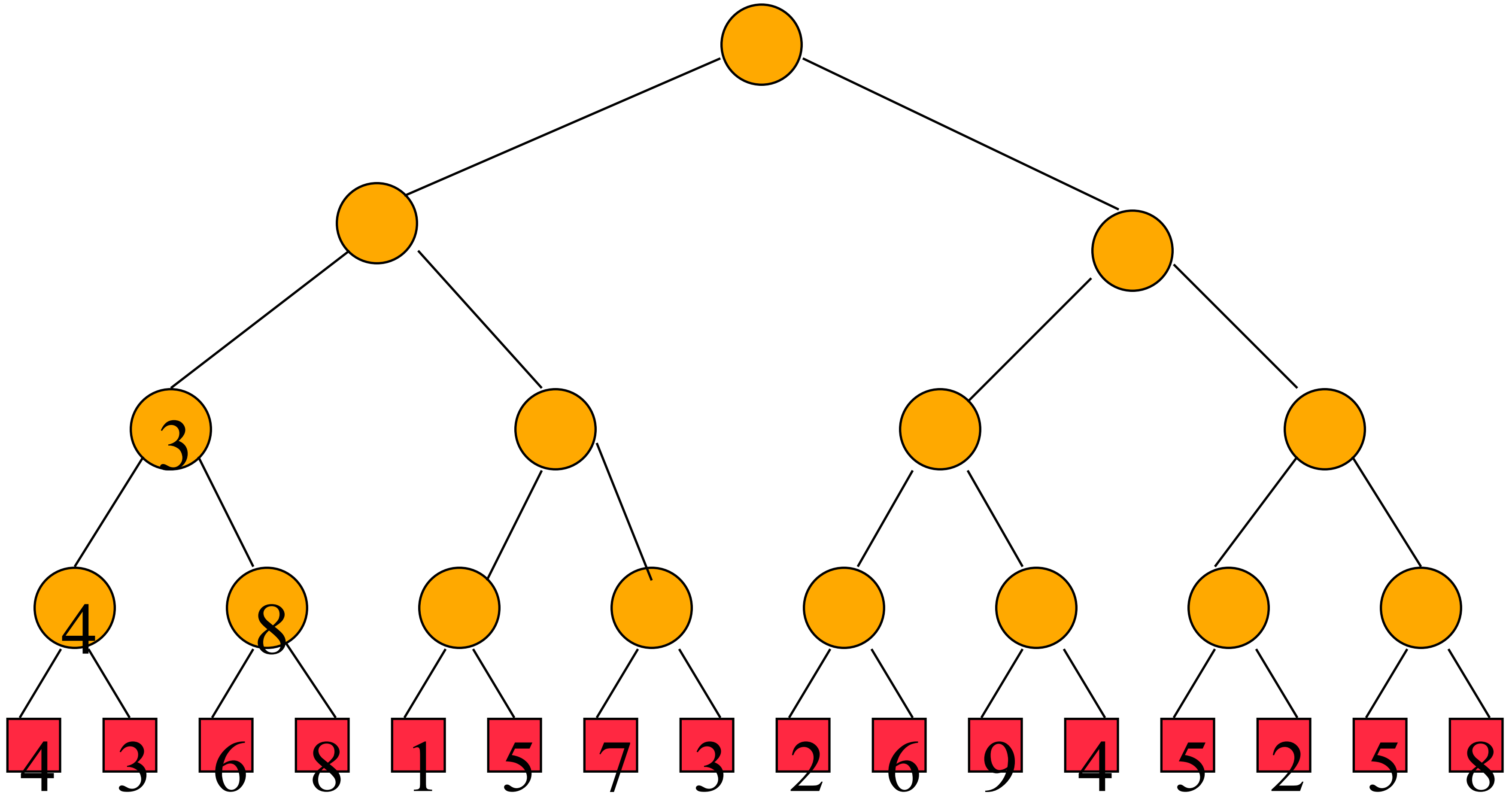


Opponent is player who lost last match played at this node.

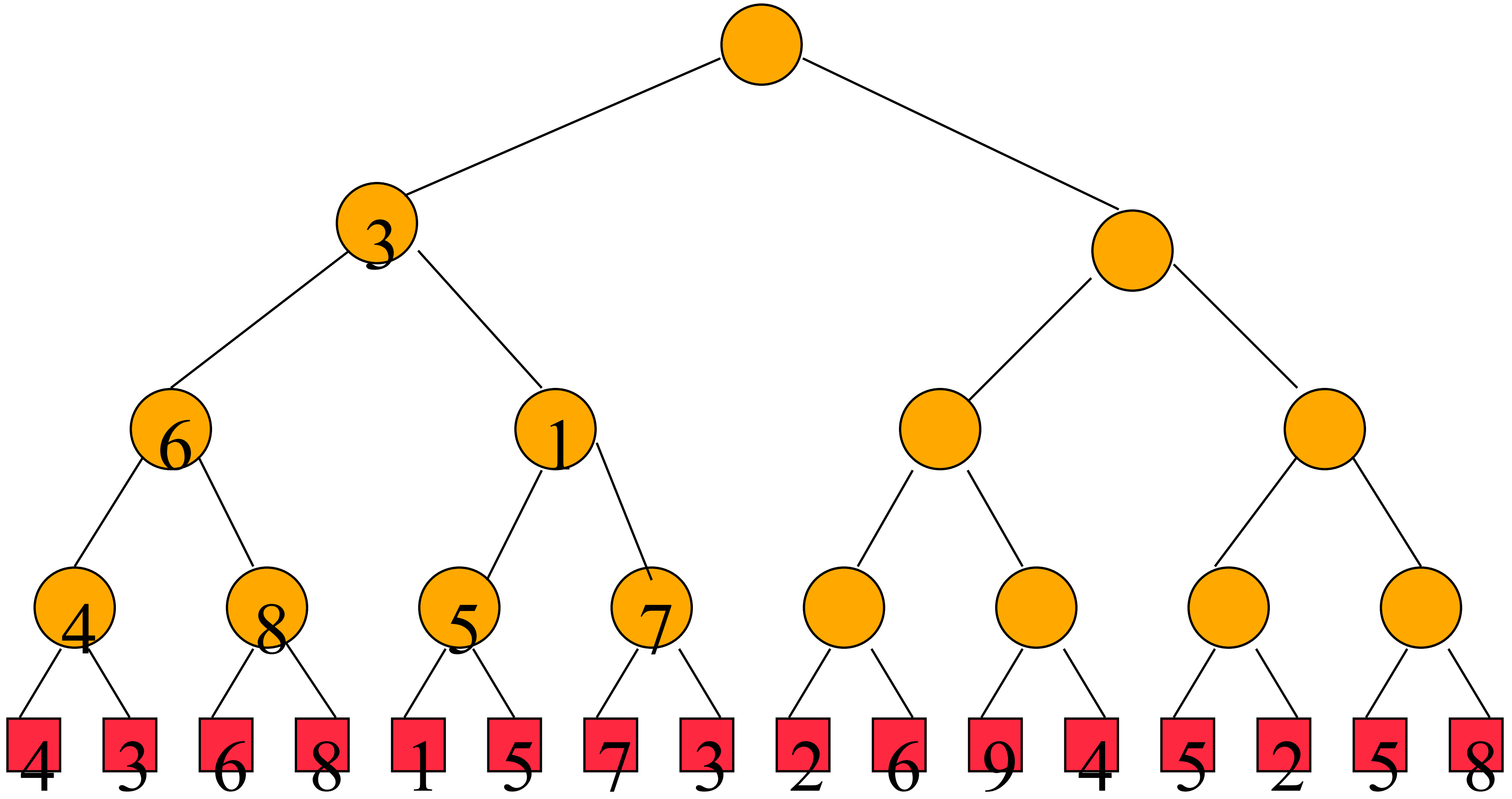
Loser Tree

Each match node stores the match loser rather than the match winner.

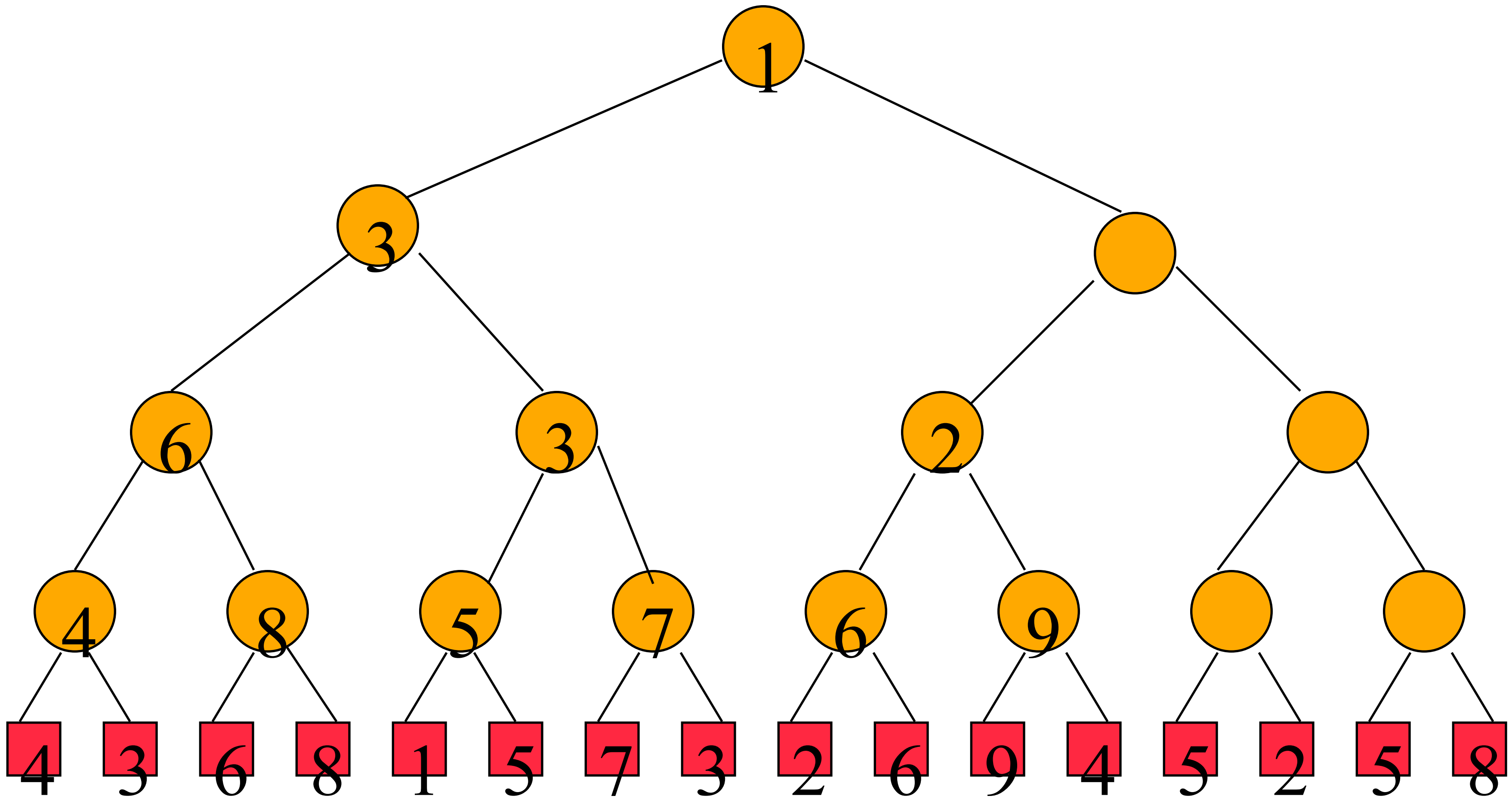
Min Loser Tree For 16 Players



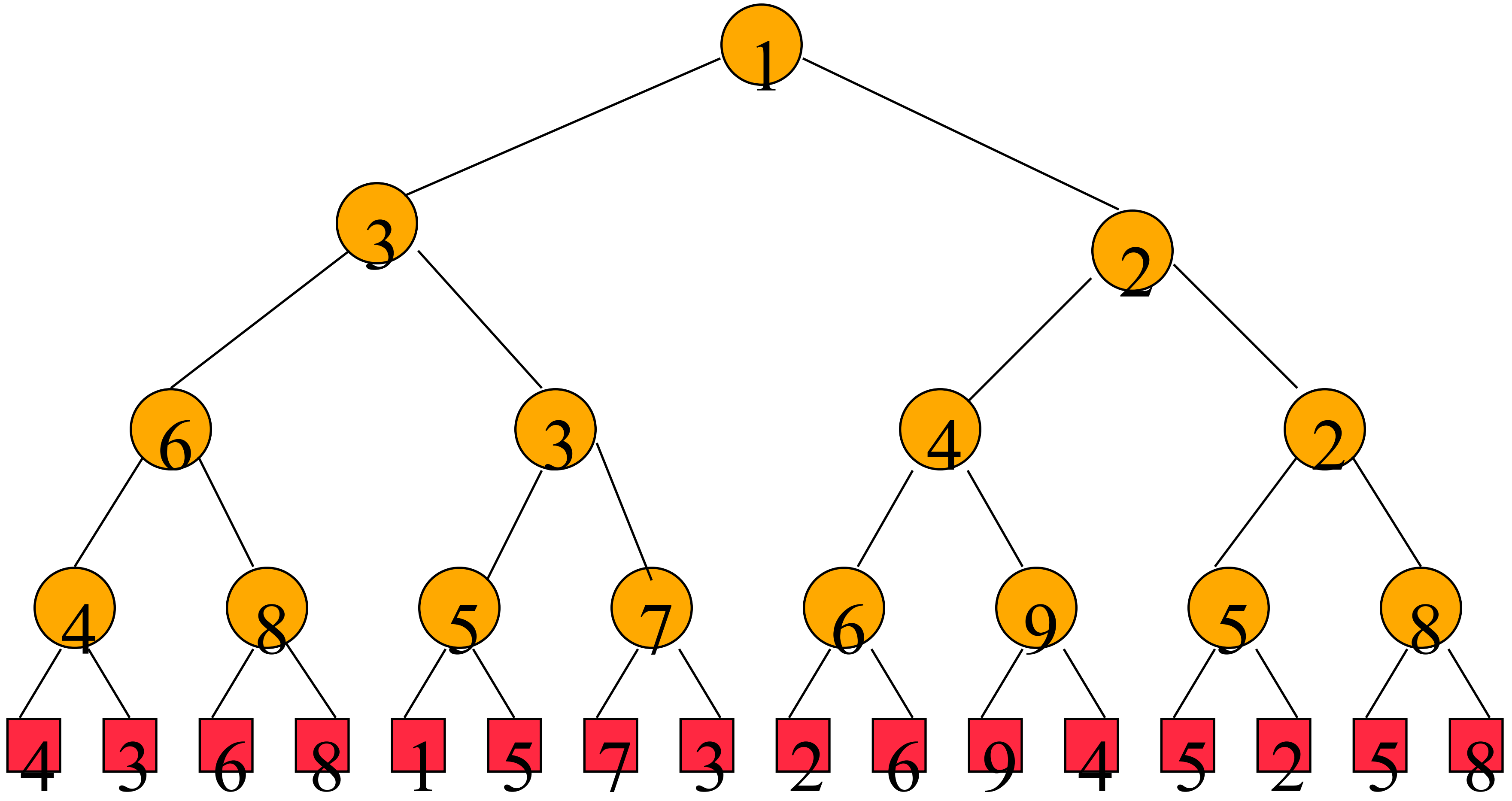
Min Loser Tree For 16 Players



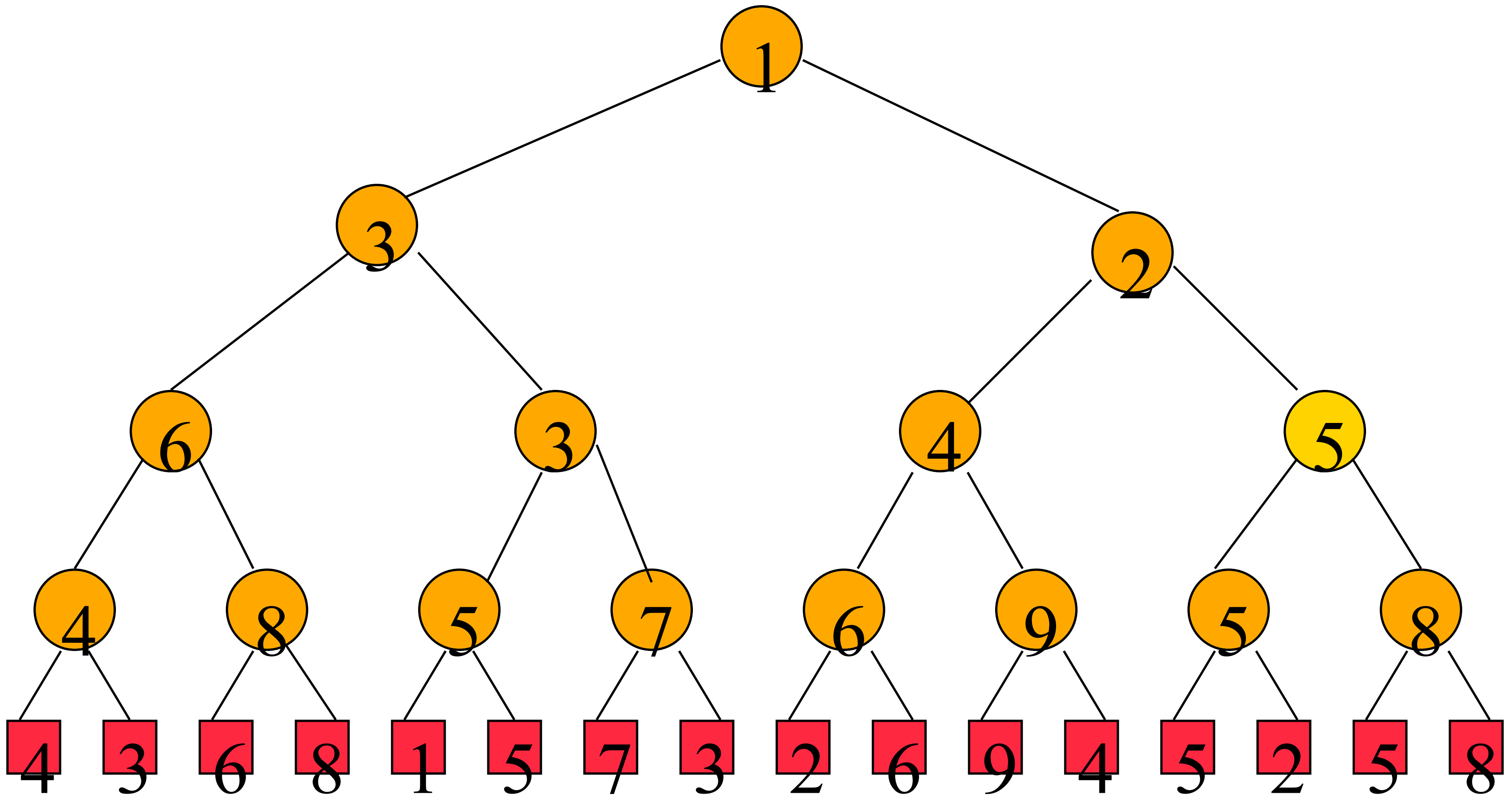
Min Loser Tree For 16 Players



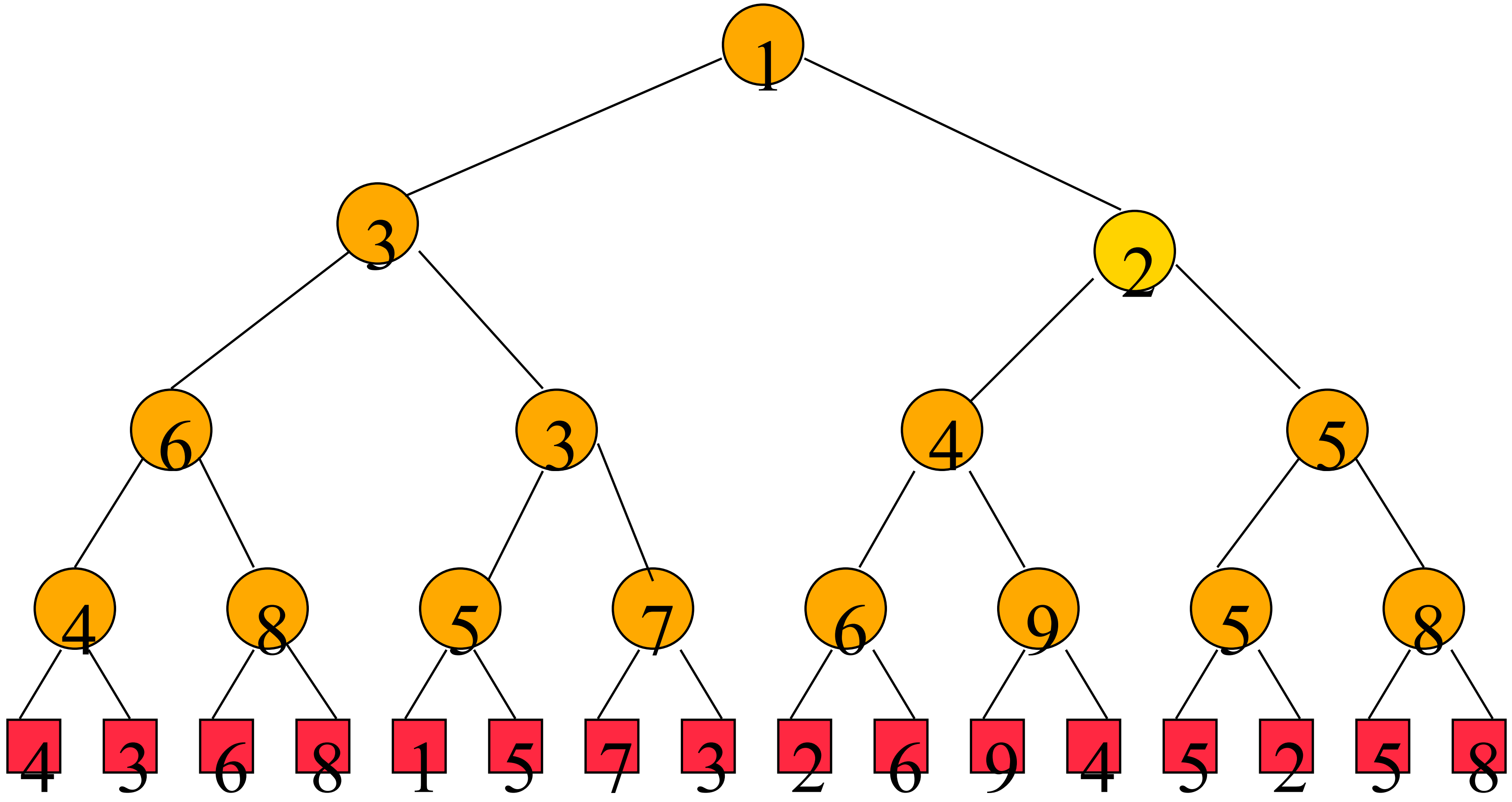
Min Loser Tree For 16 Players



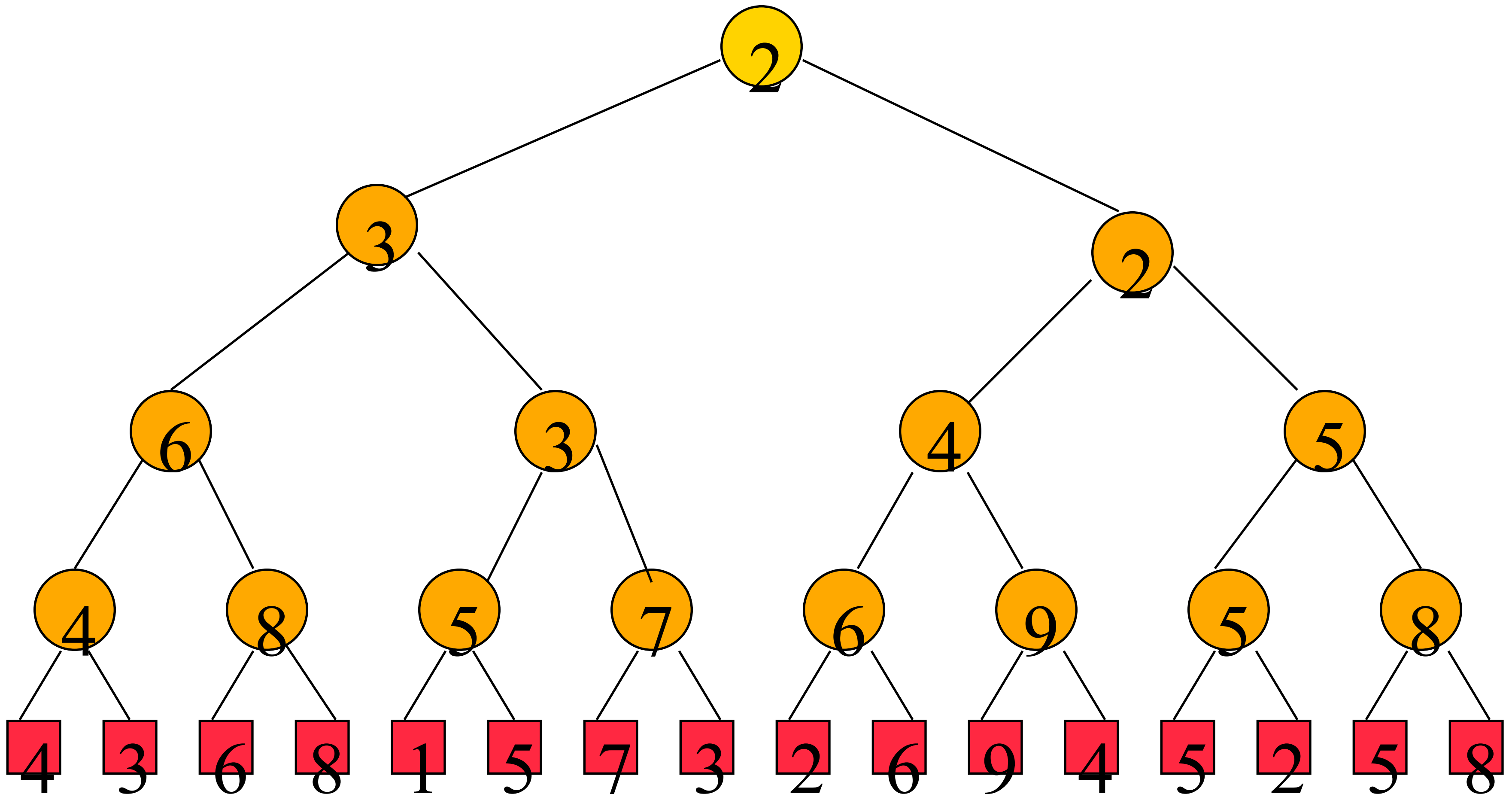
Min Loser Tree For 16 Players

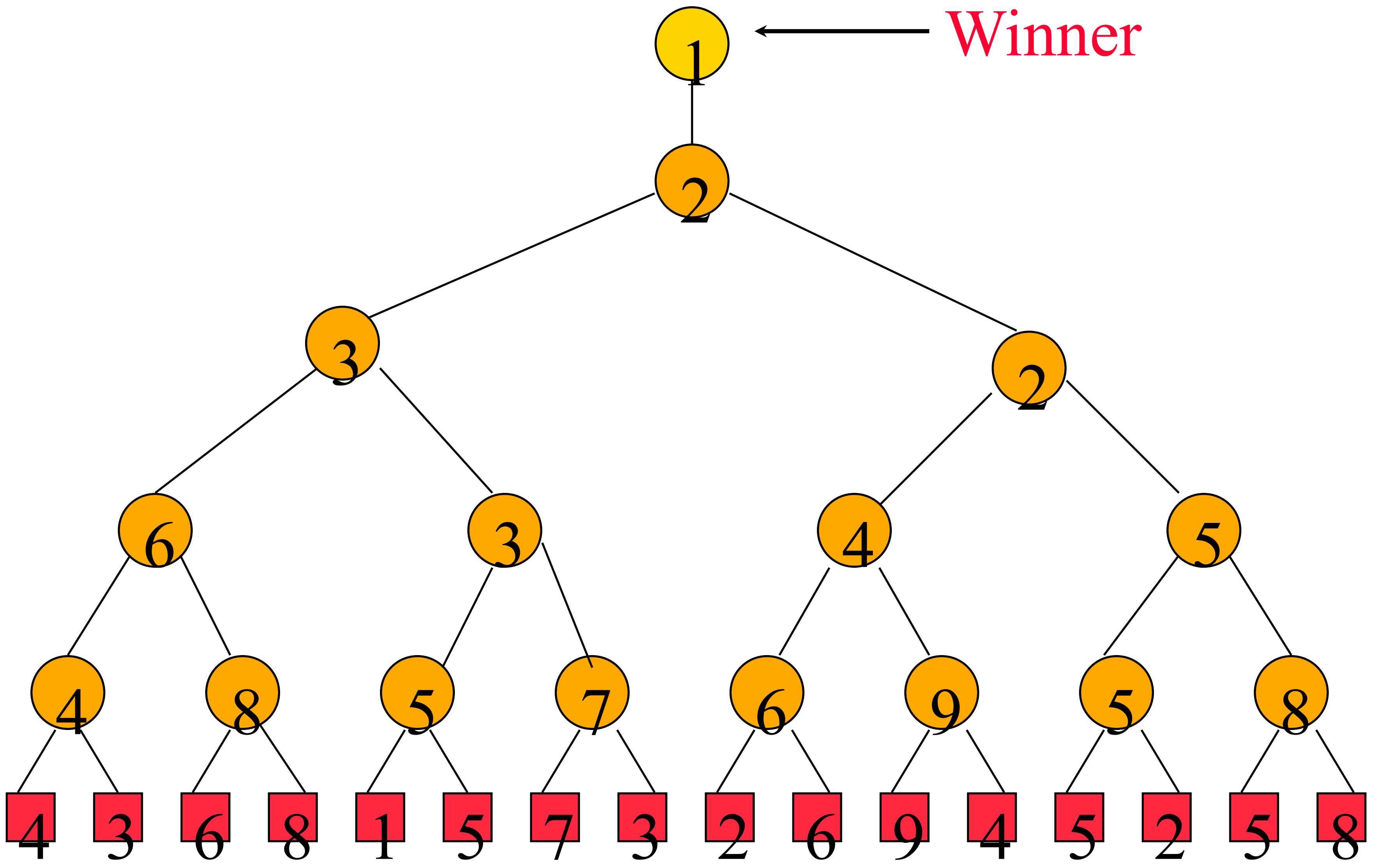


Min Loser Tree For 16 Players



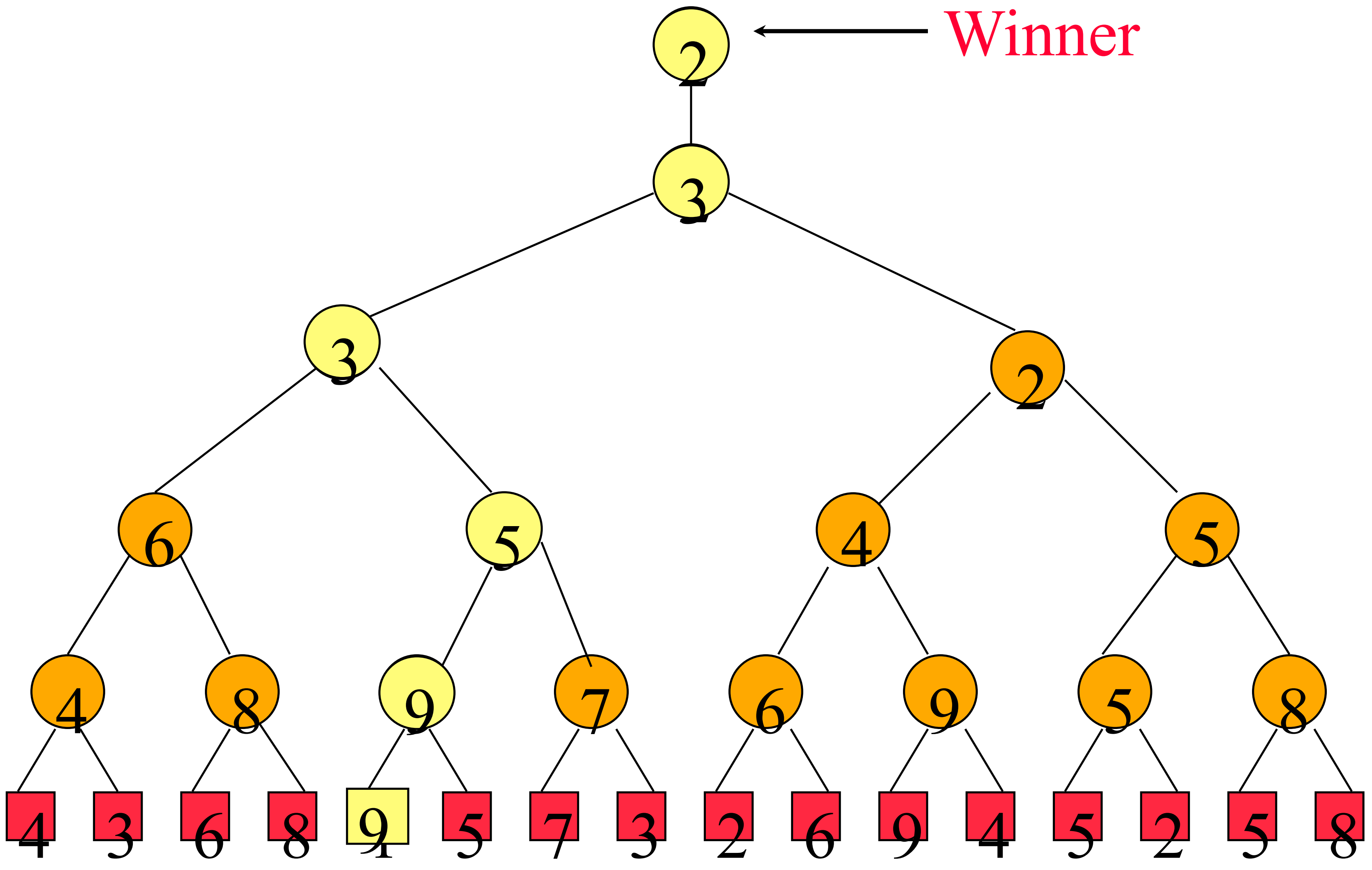
Min Loser Tree For 16 Players




Winner
1
2
3
2
6
3
4
5
4
8
5
7
6
9
5
8
4
3
6
8
1
5
7
3
2
6
9
4
5
2
5
8

Complexity Of Loser Tree Initialize

- One match at each match node.
- One store of a left child winner.
- Total time is O(n).
- More precisely Theta(n).



Replace winner with 9 and replay matches.

Complexity Of Replay

- One match at each level that has a match node.
- O(log n)
- More precisely Theta(log n).

- **class** LoserTree {
- **public:**
-     LoserTree(int k);
-     **void** Build( );
-     …
- **private:**
-     **int** k;
-     **int** *l;
-     Rec *buf;
-     **int** getKey(int i);
-     **int** getIndex(int i);
- };

- LoserTree::LoserTree(**int** k) {
- l = **new int**[k];
- buf = **new** Rec[k];
- }

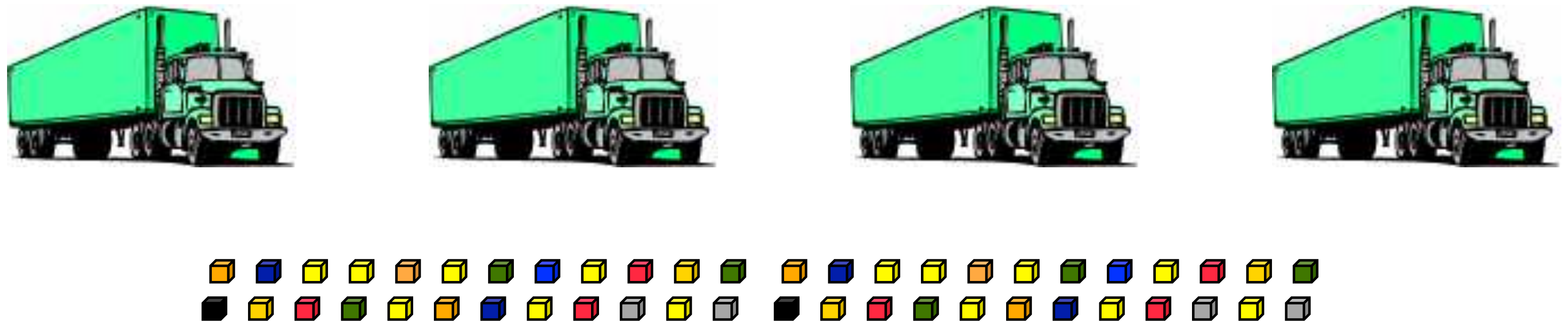
```
void LoserTree::Build( ) {
   int i;
   for (i = k – 1; i > 0; i--)
      if (getKey(2*i) > getKey(2*i + 1)
         l[i] = getIndex(2*i + 1);
      else l[i] = getIndex(2*i);
   l[0] = l[1];
   for (i = 1; i < k; i++)
      if (l[i] == getIndex(2*i) l[i] = getIndex(2*i + 1);
      else l[i] = getIndex(2*i);
}
```

```
int LoserTree::getKey(int i) {
    if (i < k) return buf[l[i]].key; else return buf[i - k].key;
}
int LoserTree::getIndex(int i) {
    if (i < k) return l[i]; else return (i – k);
}
```

More Tournament Tree Applications

- k-way merging of runs during an external merge sort
- Truck loading

Truck Loading



- n packages to be loaded into trucks
- each package has a weight
- each truck has a capacity of c tons
- minimize number of trucks

Truck Loading

$n = 5$ packages

weights $[2, 5, 6, 3, 4]$

truck capacity $c = 10$

Load packages from left to right. If a package doesn't fit into current truck, start loading a new truck.

Truck Loading

n = 5 packages

weights [2, 5, 6, 3, 4]

truck capacity c = 10

truck1 = [2, 5]

truck2 = [6, 3]

truck3 = [4]

uses 3 trucks when 2 trucks suffice

Truck Loading

n = 5 packages

weights [2, 5, 6, 3, 4]

truck capacity c = 10

truck1 = [2, 5, 3]

truck2 = [6, 4]

Bin Packing

- n items to be packed into bins
- each item has a size
- each bin has a capacity of c
- minimize number of bins

Bin Packing

Truck loading is same as bin packing.

- Truck is a bin that is to be packed (loaded).
- Package is an item/element.

Bin packing to minimize number of bins is NP-hard.

Several fast heuristics have been proposed.

Bin Packing Heuristics

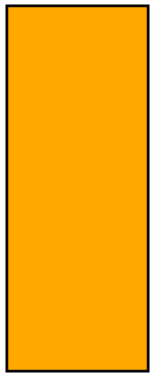
- First Fit.
  - Bins are arranged in left to right order.
  - Items are packed one at a time in given order.
  - Current item is packed into leftmost bin into which it fits.
  - If there is no bin into which current item fits, start a new bin.

First Fit

n = 4

weights = [4, 7, 3, 6]

capacity = 10



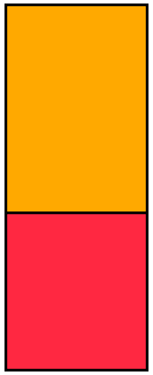
Pack red item into first bin.

First Fit

n = 4

weights = [4, 7, 3, 6]

capacity = 10



Pack blue item next.

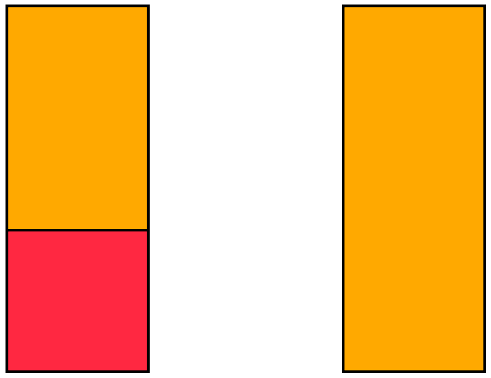
Doesn’t fit, so start a new bin.

First Fit

n = 4

weights = [4, 7, 3, 6]

capacity = 10

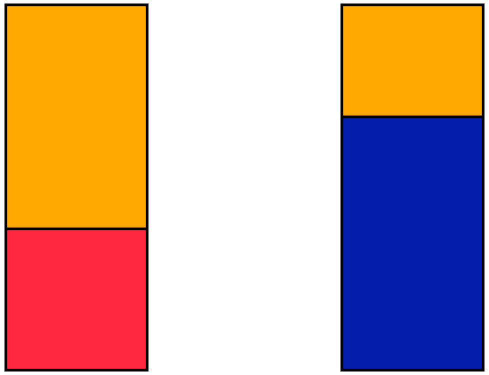


First Fit

n = 4

weights = [4, 7, 3, 6]

capacity = 10



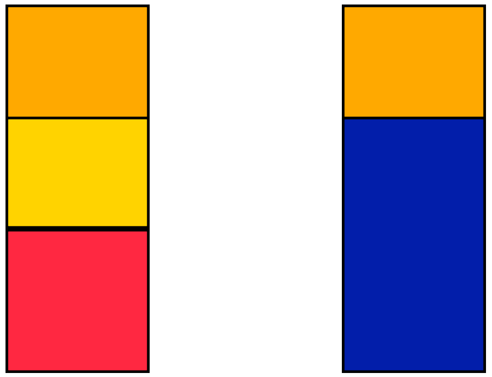
Pack yellow item into first bin.

First Fit

n = 4

weights = [4, 7, 3, 6]

capacity = 10



Pack green item.

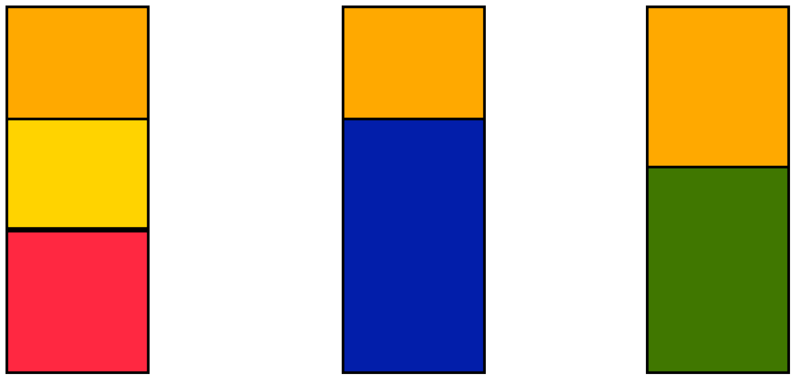
Need a new bin.

First Fit

n = 4

weights = [4, 7, 3, 6]

capacity = 10



Not optimal.

2 bins suffice.

Bin Packing Heuristics

- First Fit Decreasing.
  - Items are sorted into decreasing order.
  - Then first fit is applied.

Bin Packing Heuristics

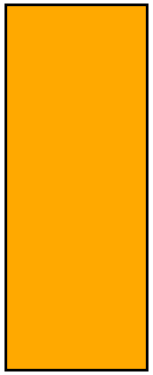
- Best Fit.
  - Items are packed one at a time in given order.
  - To determine the bin for an item, first determine set S of bins into which the item fits.
  - If S is empty, then start a new bin and put item into this new bin.
  - Otherwise, pack into bin of S that has least available capacity.

Best Fit Example

n = 4

weights = **[4, 7, 3, 6]**

capacity = 10



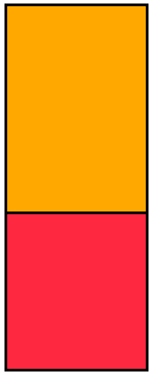
Pack red item into first bin.

Best Fit

n = 4

weights = **[4, 7, 3, 6]**

capacity = 10



Pack blue item next.

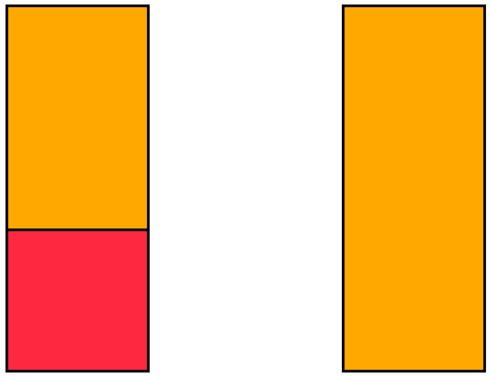
Doesn't fit, so start a new bin.

Best Fit

n = 4

weights = **[4, 7, 3, 6]**

capacity = 10

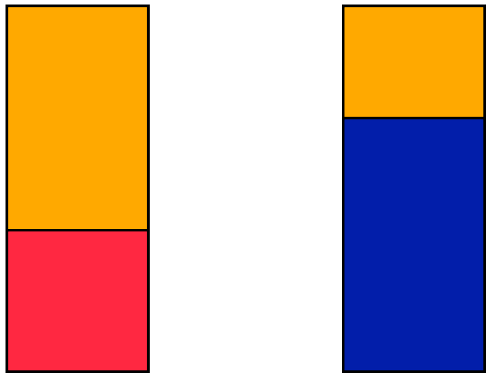


Best Fit

n = 4

weights = **[4, 7, 3, 6]**

capacity = 10



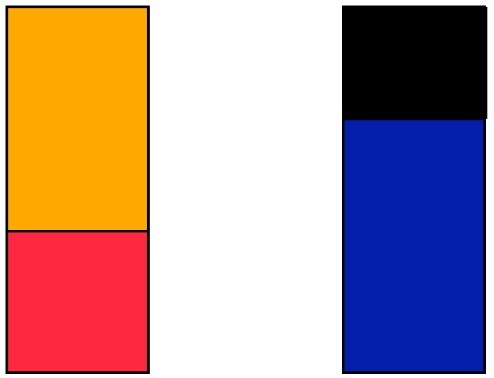
Pack yellow item into second bin.

Best Fit

n = 4

weights = **[4, 7, 3, 6]**

capacity = 10



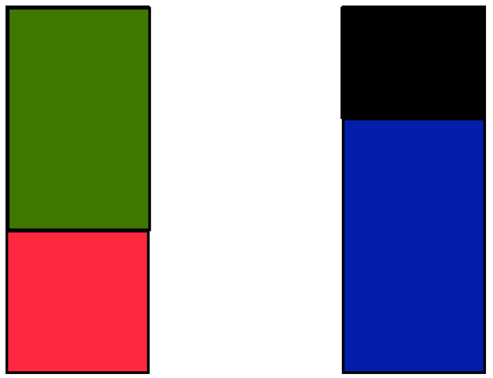
Pack green item into first bin.

Best Fit

n = 4

weights = **[4, 7, 3, 6]**

capacity = 10



Optimal packing.

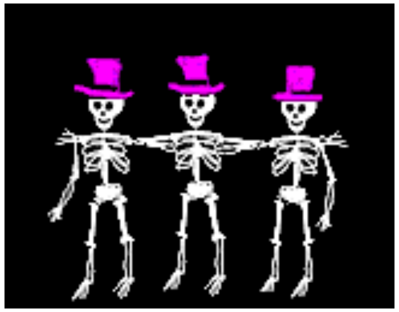
Implementation Of Best Fit

- Use a dynamic dictionary in which the elements are of the form (available capacity, bin index).
- Pack an item whose requirement is s.
  - Find a bin with smallest available capacity >= s.
  - Reduce available capacity of this bin by s.
    - May be done by removing old pair and inserting new one.
  - If no such bin, start a new bin.
    - Insert a new pair into the dictionary.

Bin Packing Heuristics

- Best Fit Decreasing.
  - Items are sorted into decreasing order.
  - Then best fit is applied.

Performance



- For first fit and best fit:

  Heuristic Bins <= (17/10)(Minimum Bins) + 2

- For first fit decreasing and best fit decreasing:

  Heuristic Bins <= (11/9)(Minimum Bins) + 4

Complexity Of First Fit

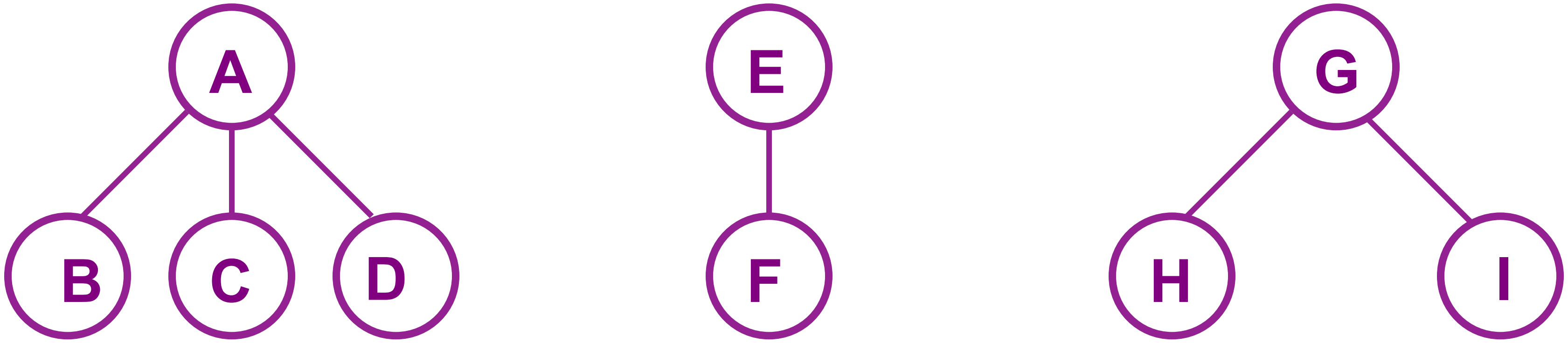
Use a max tournament tree in which the players are n bins and the value of a player is the available capacity in the bin.

O(n log n), where n is the number of items.

- **Exercises: P301-1,4**

Forests

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



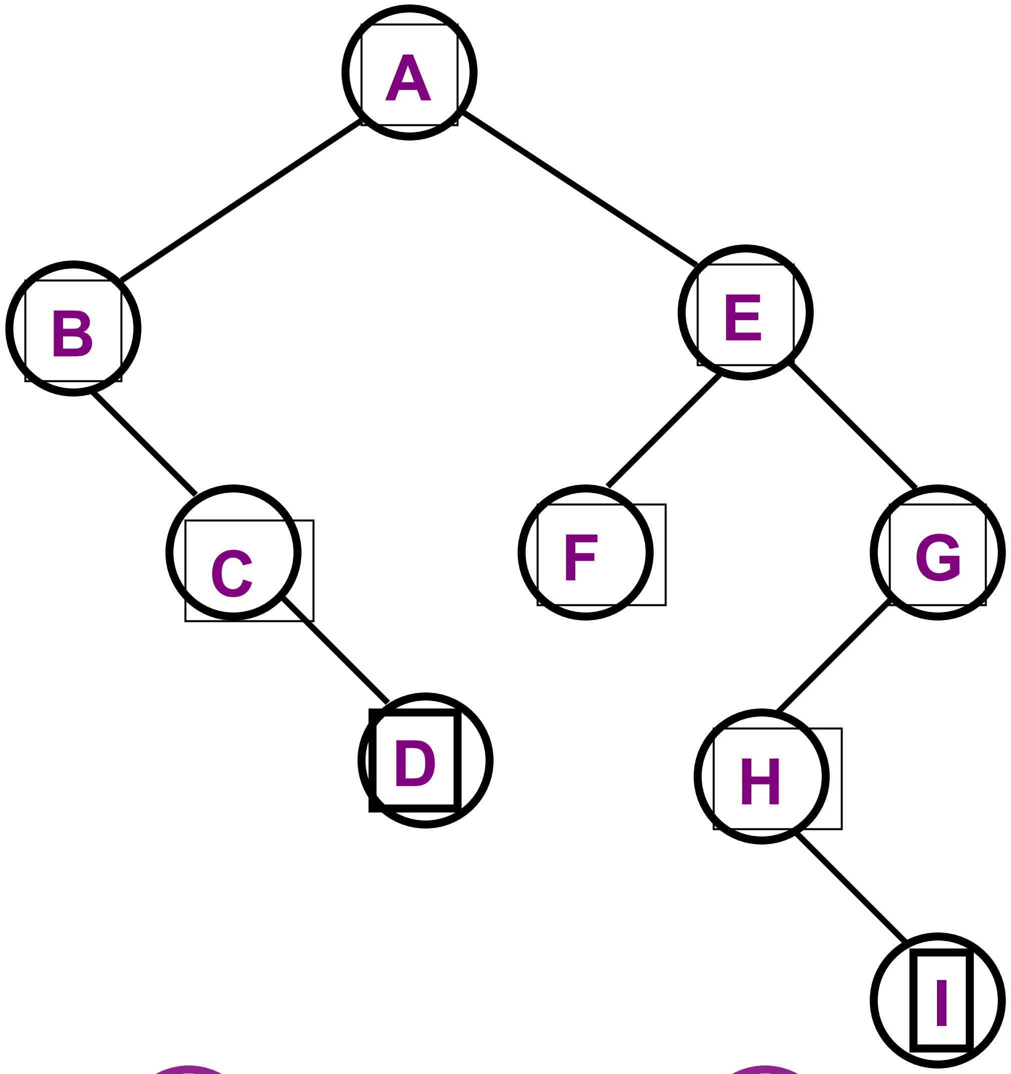
Transforming a Forest into a Binary Tree

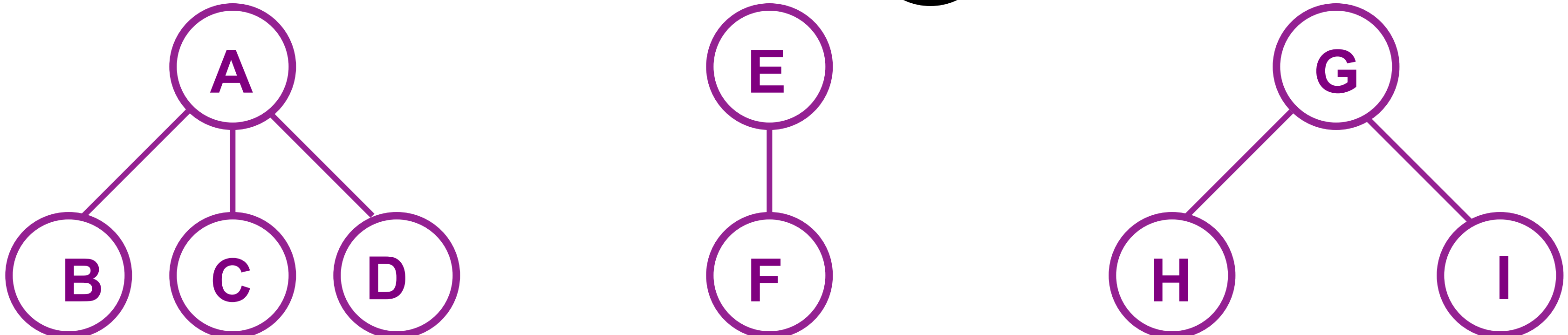
Definition:

If $T_1, \ldots, T_n$ is a forest of trees, then the binary tree corresponding to it , denoted by $B(T_1, \ldots, T_n)$,

(1) is empty if n=0

(2) has root equal to root(T_1); has left subtree equal to $B(T_{11}, \ldots, T_{1m})$, where $T_{11}, \ldots, T_{1m}$ are the subtrees of root(T_1); and has right subtree $B(T_2, \ldots, T_n)$.

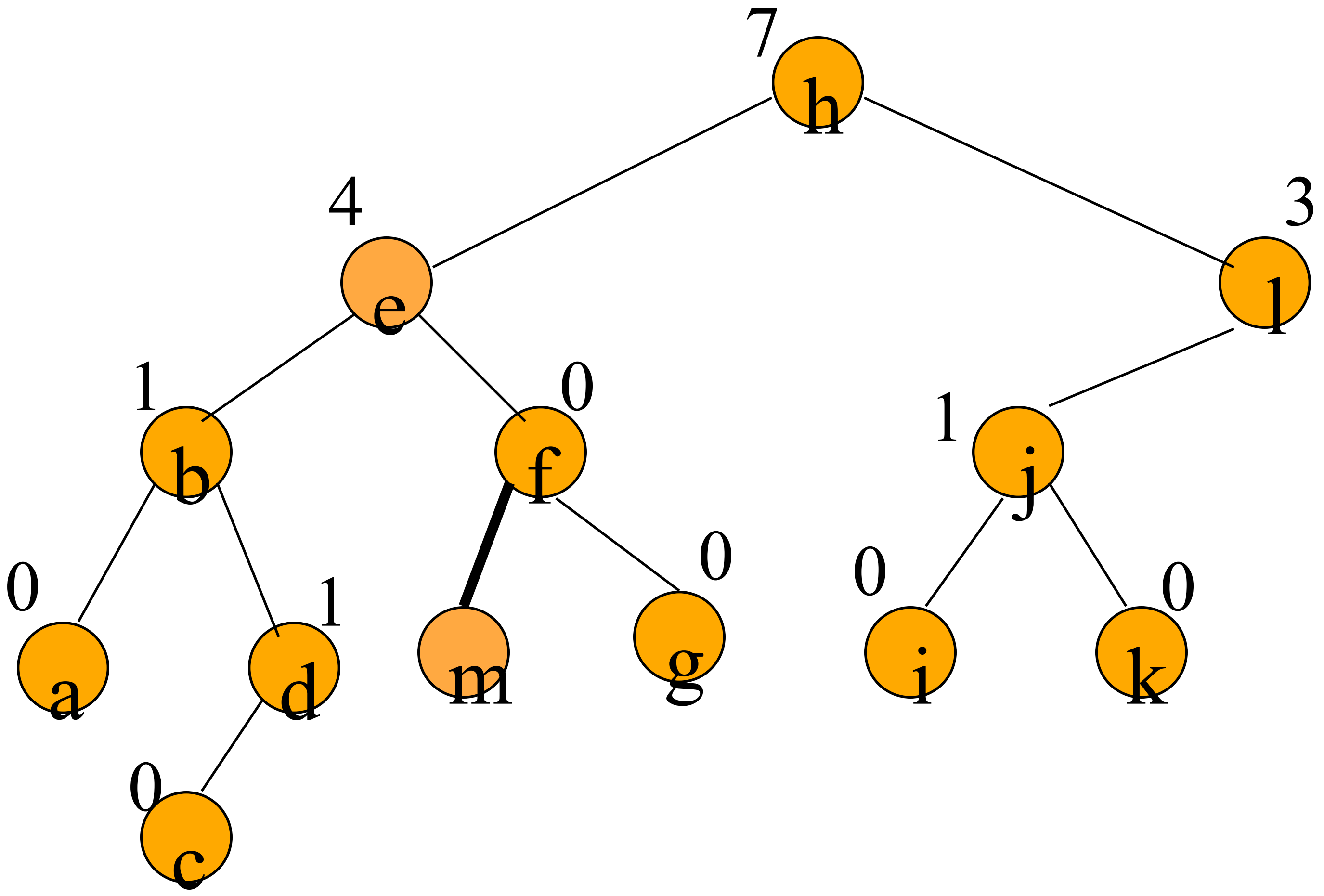

A
B
E
C
F
G
D
H
I


A
E
G
B
C
D
F
H
I

Transforming a Forest into a Binary Tree

- Left child → first child
- Right child→ Sibling

Transform a Binary Tree to a Forest



Forest Traversals

Let T be the corresponding binary tree of a forest F.

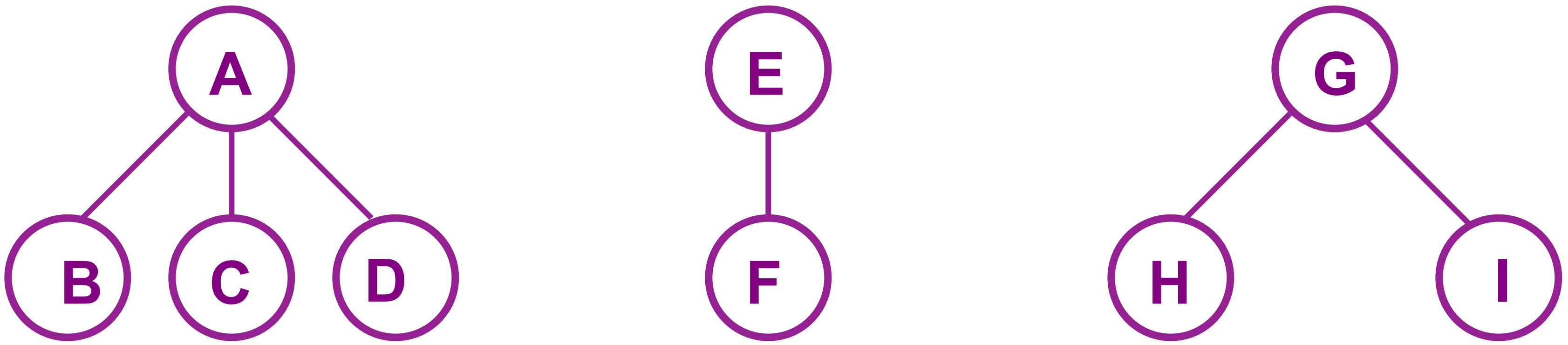
Visiting the nodes of F in forest preorder is defined as:

(1) If F is empty then return.

(2) Visit the root of the first tree of F.

(3) Traverse the subtrees of the first tree in forest preorder.

(4) Traverse the remaining trees of F in forest preorder.


A
B
C
D
E
F
G
H
I

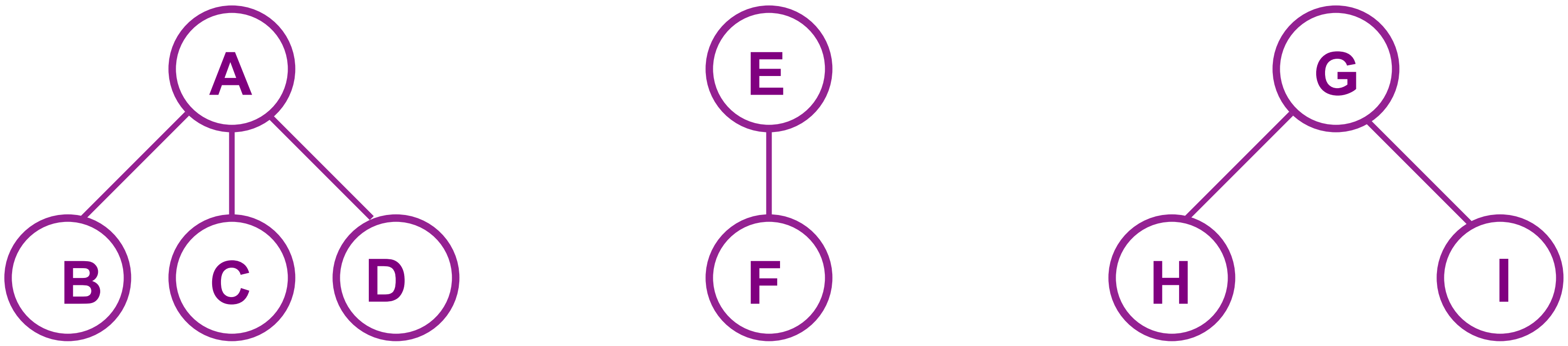
Visiting the nodes of F in forest inorder is defined as:

(1) If F is empty then return.

(2) Traverse the subtrees of the first tree in forest inorder.

(3) Visit the root of the first tree of F.

(4) Traverse the remaining trees of F in forest inorder.


A
B
C
D
E
F
G
H
I

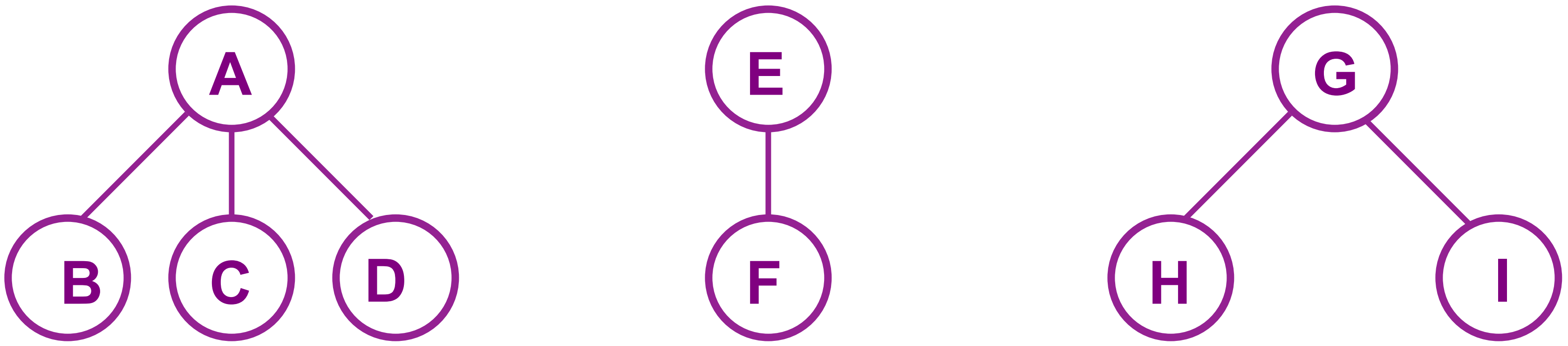
Visiting the nodes of F in forest postorder is defined as:

(1) If F is empty then return.

(2) Traverse the subtrees of the first tree in forest postorder.

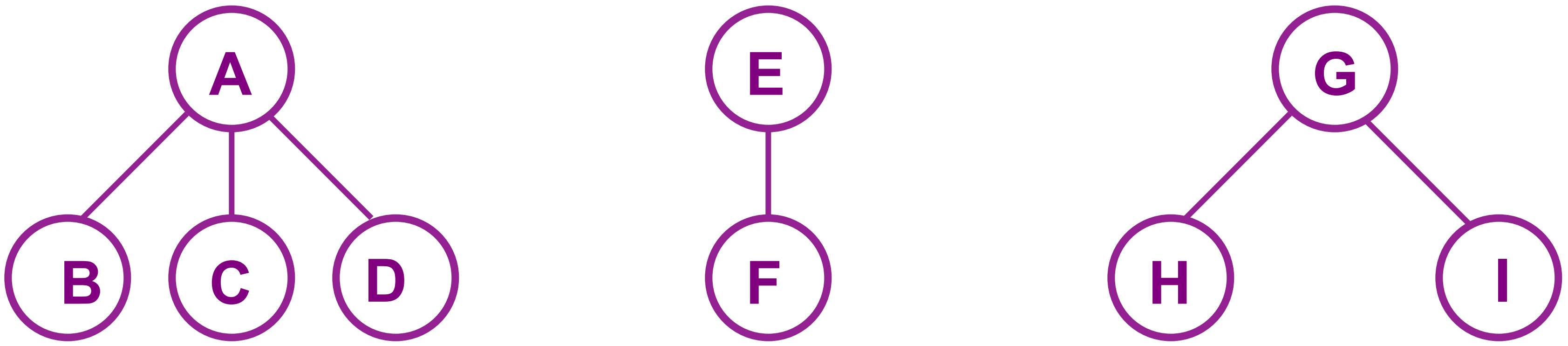
(3) Traverse the remaining trees of F in forest postorder.

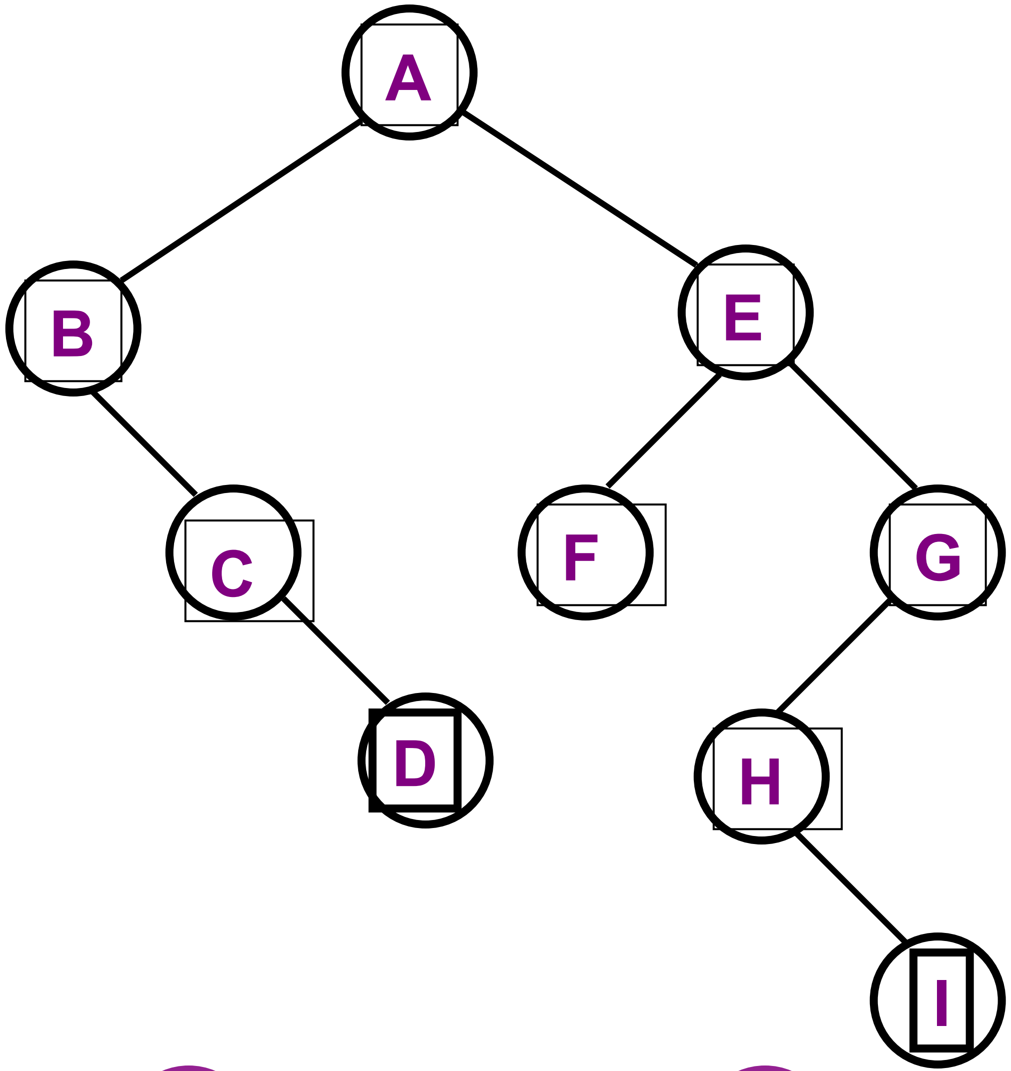
(4) Visit the root of the first tree of F.


A
B
C
D
E
F
G
H
I

In level-order traversal of F

- Nodes are visited by level
- Beginning with the roots of each trees in F
- Within each level, from left to right.

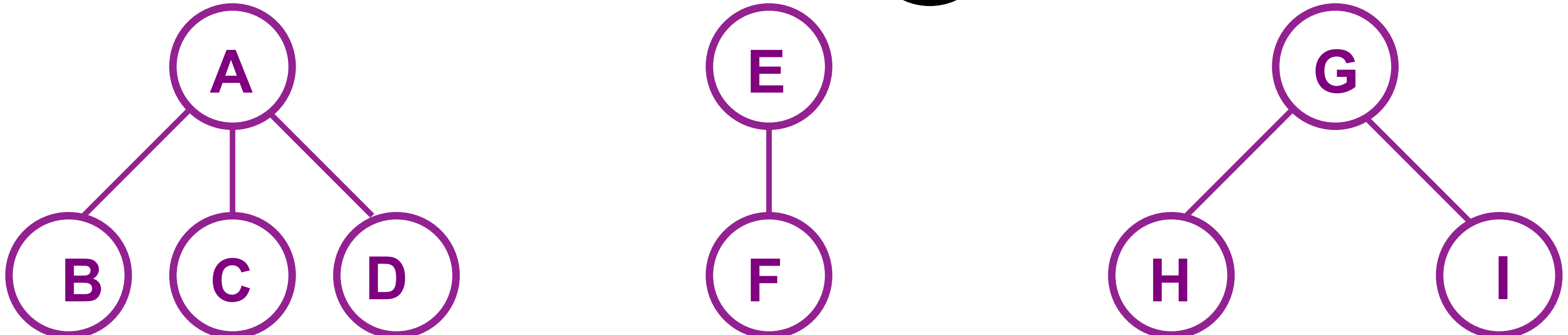

A
B
C
D
E
F
G
H
I



PreOrder

InOrder

PostOrder



Exercises: P304-3.

Set Presentation

- Assume:
- Elements of the sets are the numbers 0, 1, 2, …, n-1 (might be thought as indices).
- For any two sets S_i, S_j, $i \neq j$, $S_i \cap S_j = \varnothing$.
- Operations:
  - Disjoint set union $S_i \cup S_j$.
  - Find(i)---find the set containing i.

Union-Find Problem

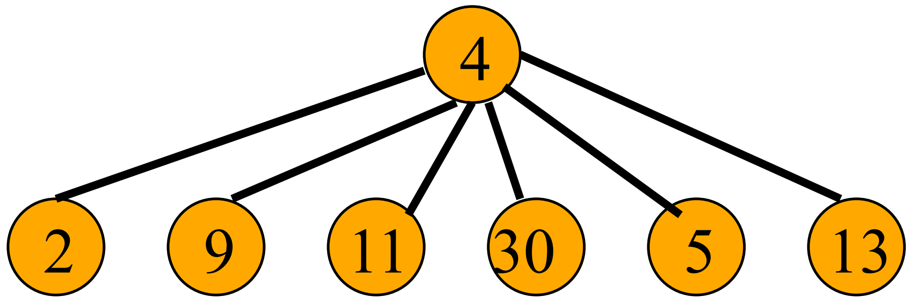
- A union operation combines two sets into one.
  - Each of the n elements is in exactly one set at any time.
- A find operation identifies the set that contains a particular element.

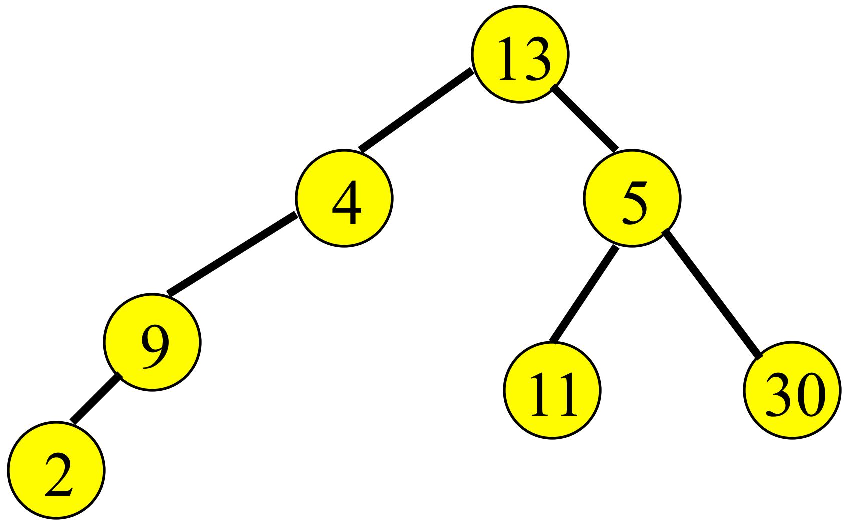
Using Arrays And Chains

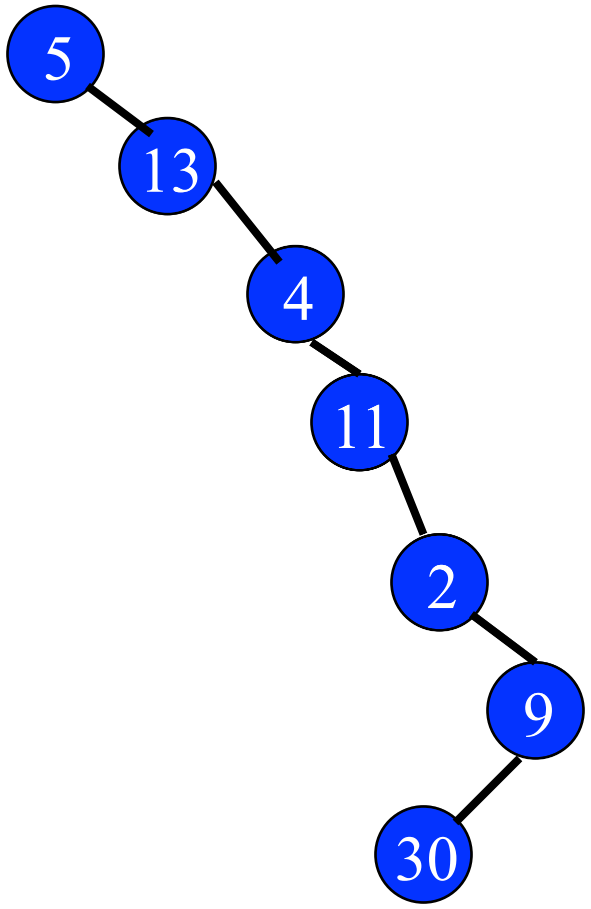
- Array
  - Union
  - Find
- Chains
  - Union
  - Find

A Set As A Tree

- S = {2, 4, 5, 9, 11, 13, 30}
- Some possible tree representations:

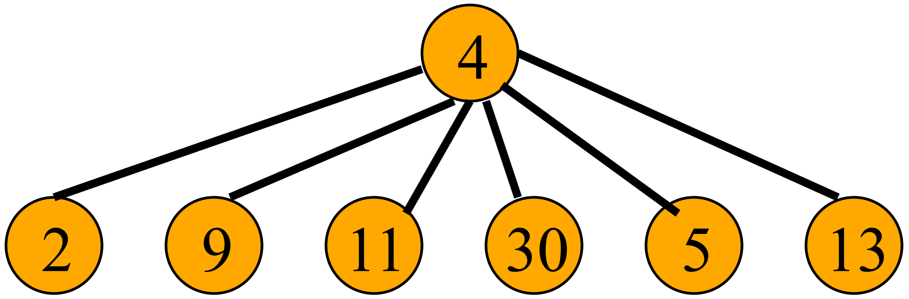






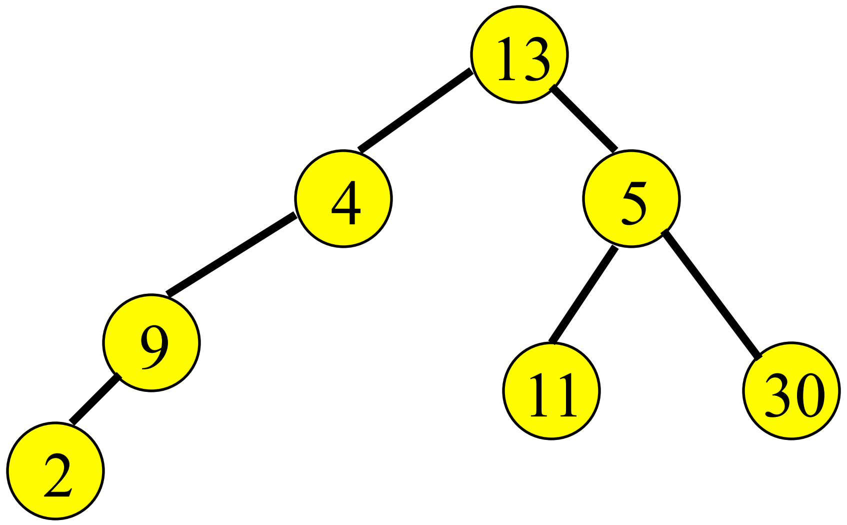
Result Of A Find Operation

- find(i) is to identify the set that contains element i.
- In most applications of the union-find problem, the user does not provide set identifiers.
- The requirement is that find(i) and find(j) return the same value iff elements i and j are in the same set.



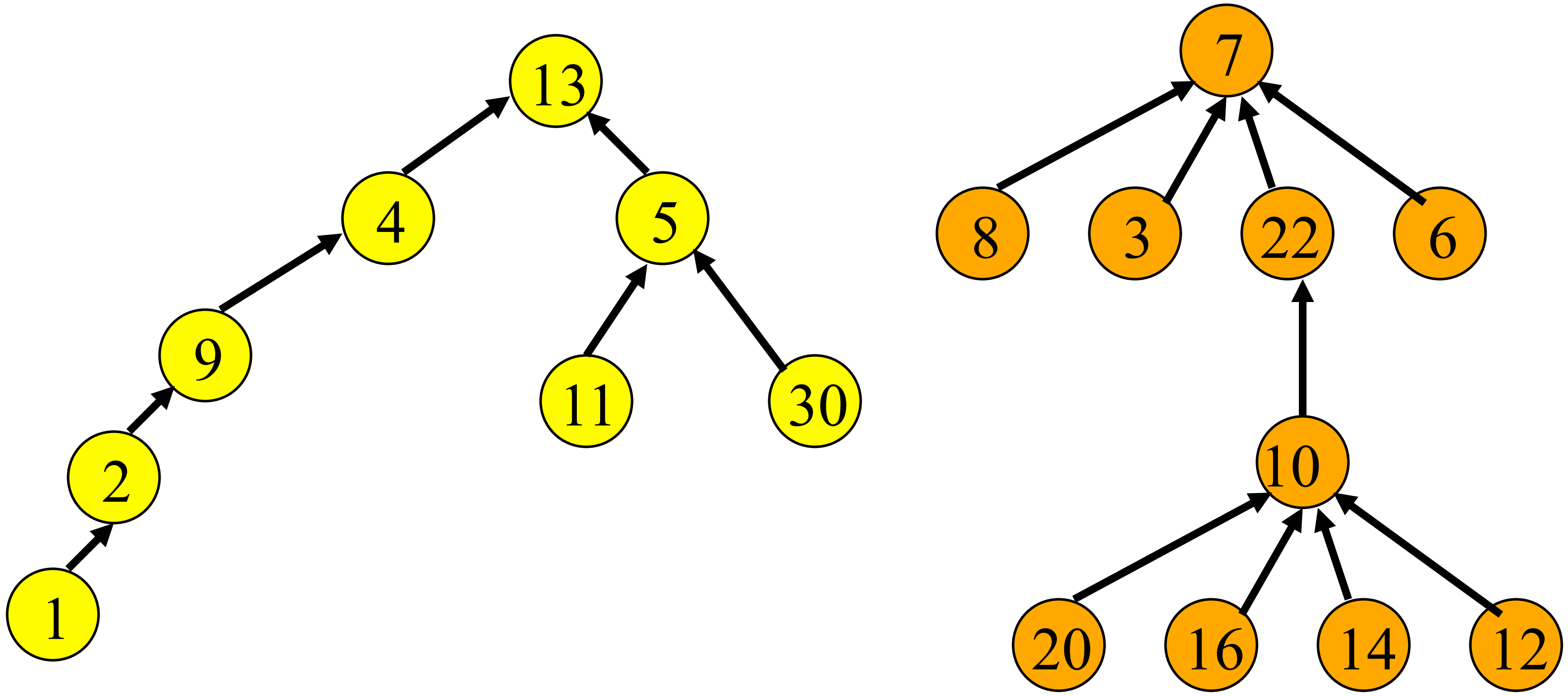
find(i) will return the element that is in the tree root.

Strategy For find(i)



- Tree traversal from the root? O(n)
- Start at the node that represents element i and climb up the tree until the root is reached.
- Return the element in the root.
- To climb the tree, each node must have a parent pointer.

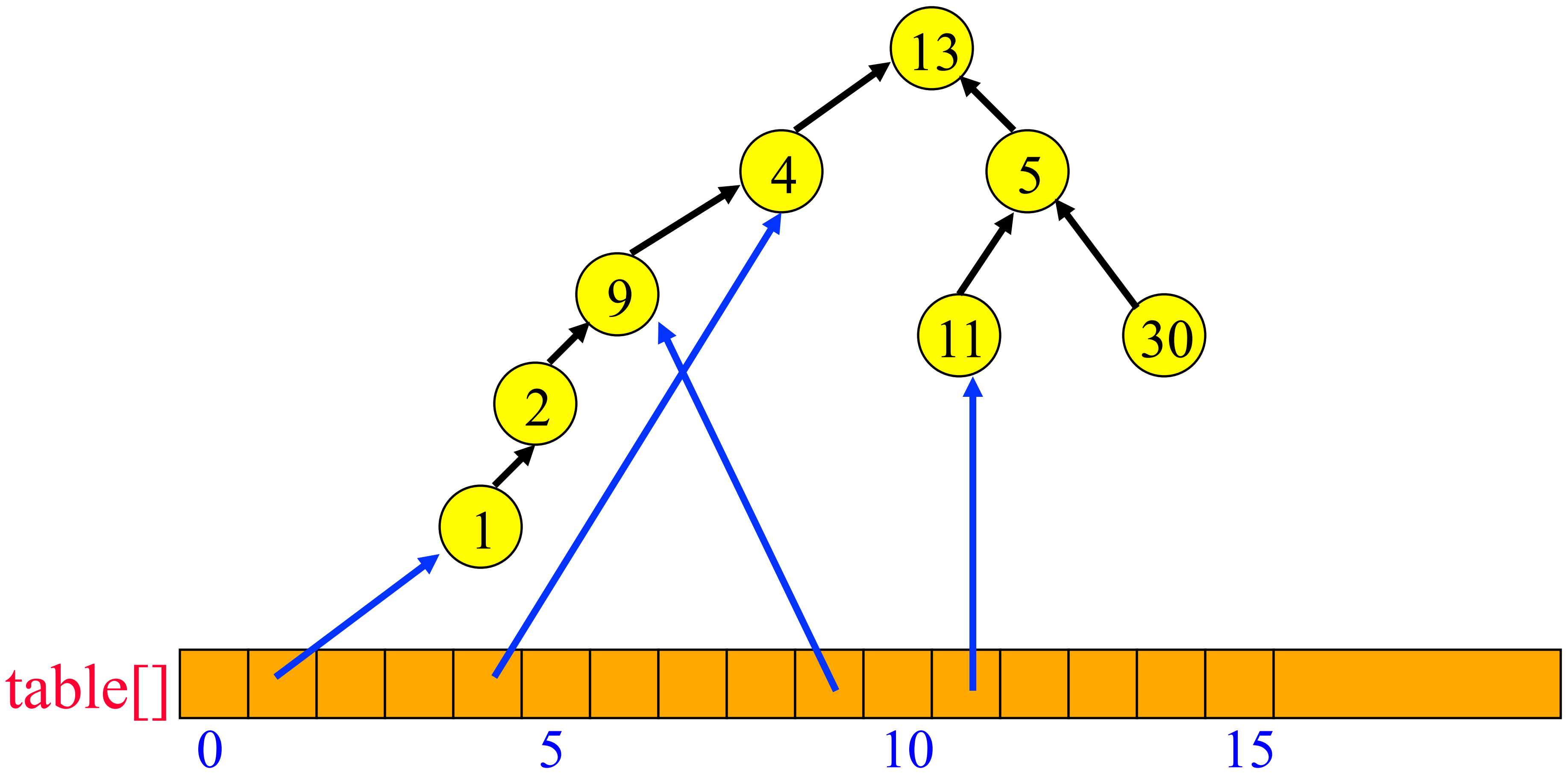
Trees With Parent Pointers



Possible Node Structure

- Use nodes that have two fields: element and parent.
  - Use an array table[] such that table[i] is a pointer to the node whose element is i.
  - To do a find(i) operation, start at the node given by table[i] and follow parent fields until a node whose parent field is null is reached.
  - Return element in this root node.

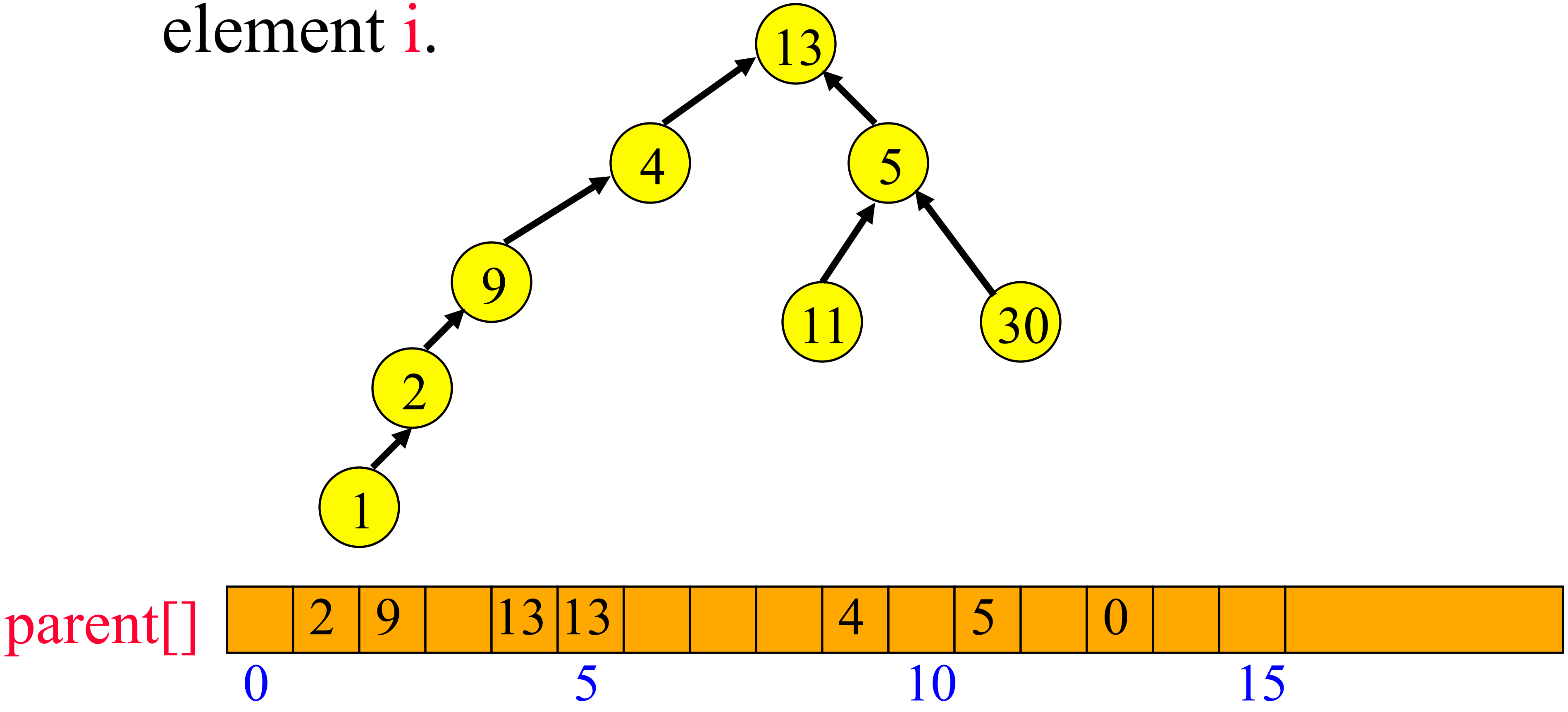
Example



(Only some table entries are shown.)

Better Representation

- Use an integer array parent[] such that parent[i] is the element that is the parent of element i.

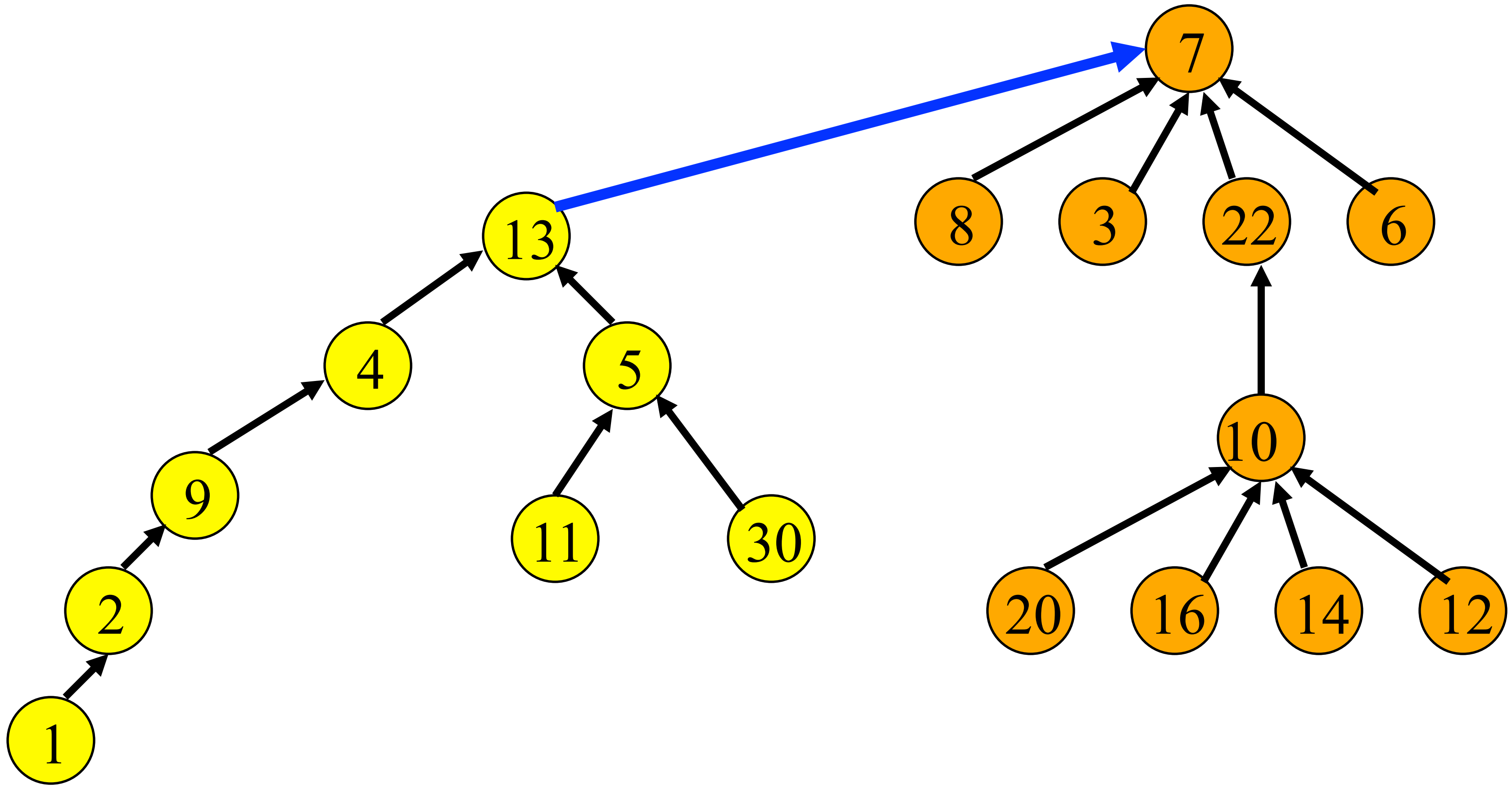


```
class Sets {
public:
    // Set operations
private:
    int  *parent;
    int  n; // number of set elements
};
Sets::Sets (int numberOfElements)
{
    if (numberOfElements < 2) throw “Must have at least 2 elements.”;
    n=numberOfElements;
    parent=new int[n];
    fill(parent, parent+n, -1);
}
```

Union Operation

- union(i,j)
  - i and j are the roots of two different trees, i != j.
- To unite the trees, make one tree a subtree of the other.
  - parent[j] = i

Union Example



- union(7,13)

The Simple Find Method

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

The Simple Union Method

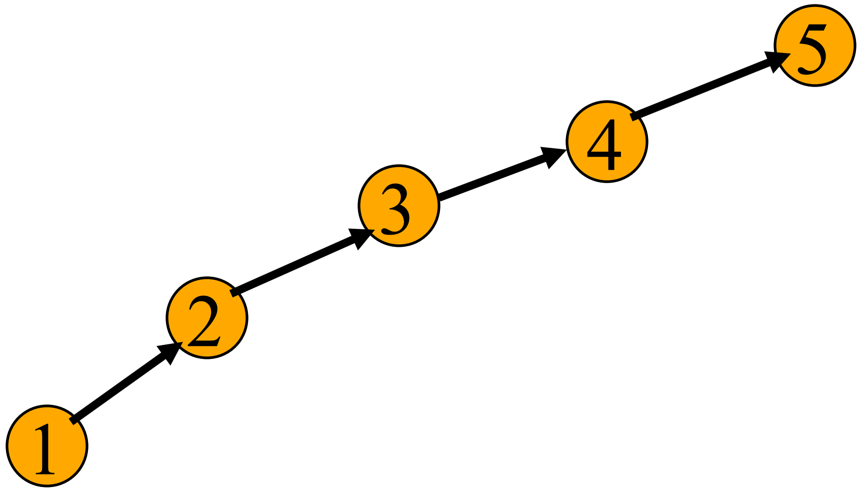
```
void Sets::SimpleUnion (int j, int i)
{ // Replace the disjoint sets with roots i and j,
i!=j with their
  // union
     parent[i] = j;
}
```

Time Complexity Of union()

- O(1)

Time Complexity of find()

- Tree height may equal number of elements in tree.
  - union(2,1), union(3,2), union(4,3), union(5,4)…

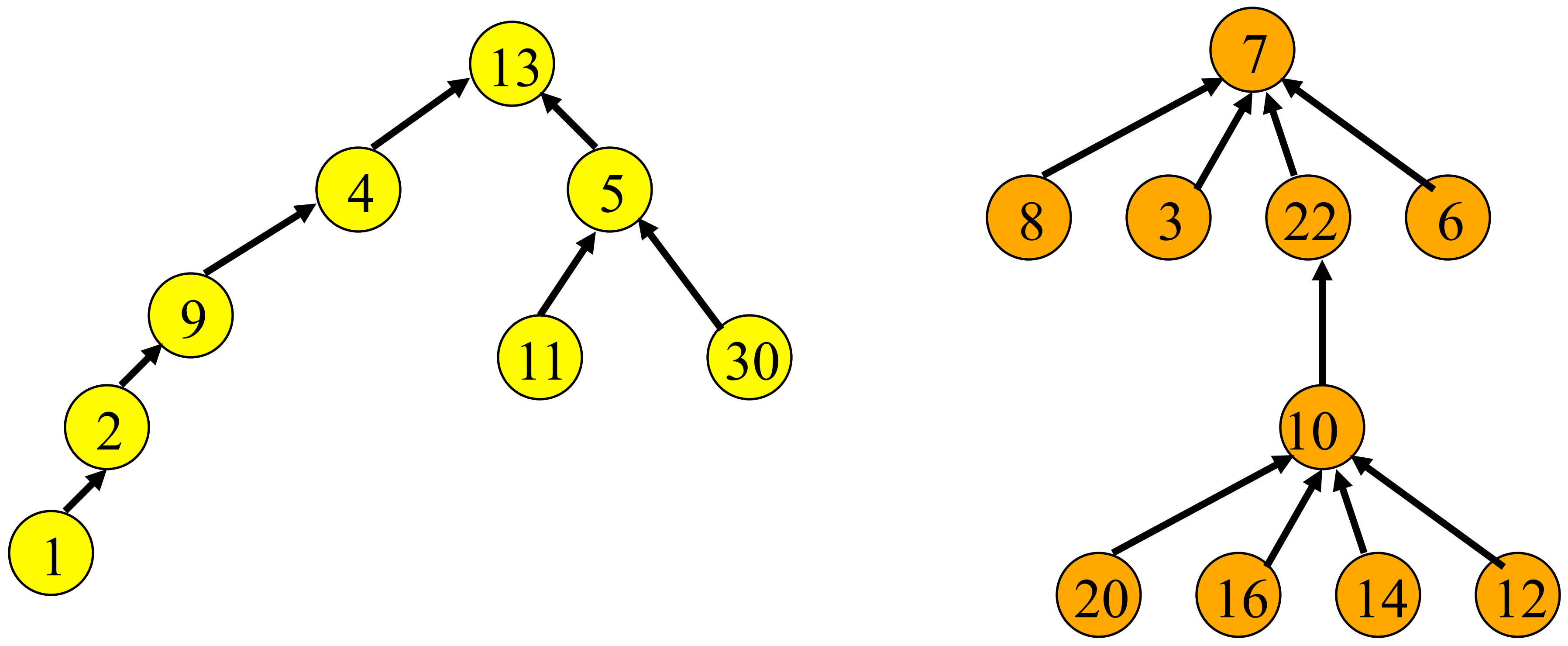


So complexity is O(u).

u Unions and f Find Operations

- O(u + uf) = O(uf)
- Time to initialize parent[i] = 0 for all i is O(n).
- Total time is O(n + uf).

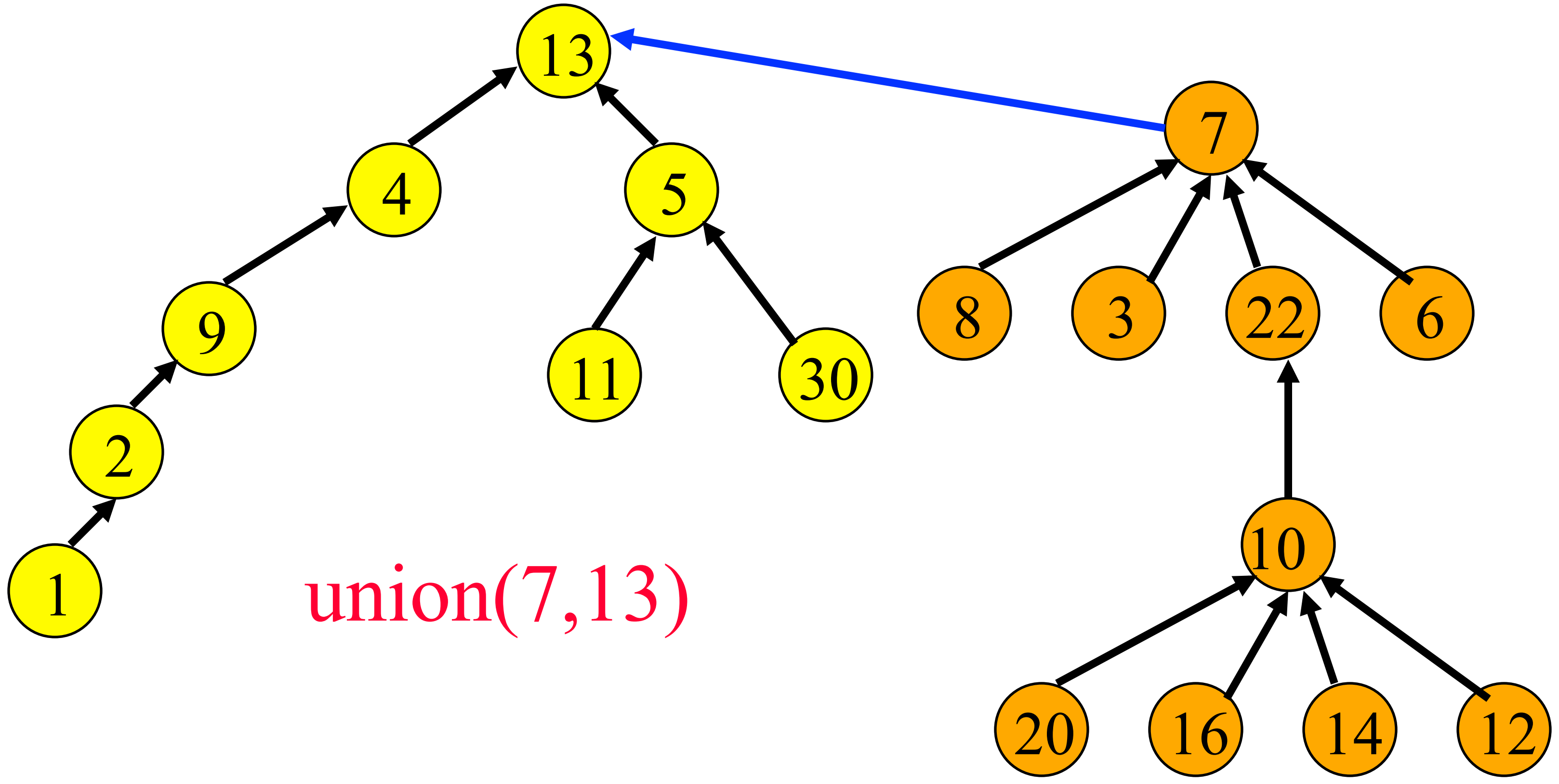
Smart Union Strategies



- union(7,13)
- Which tree should become a subtree of the other?

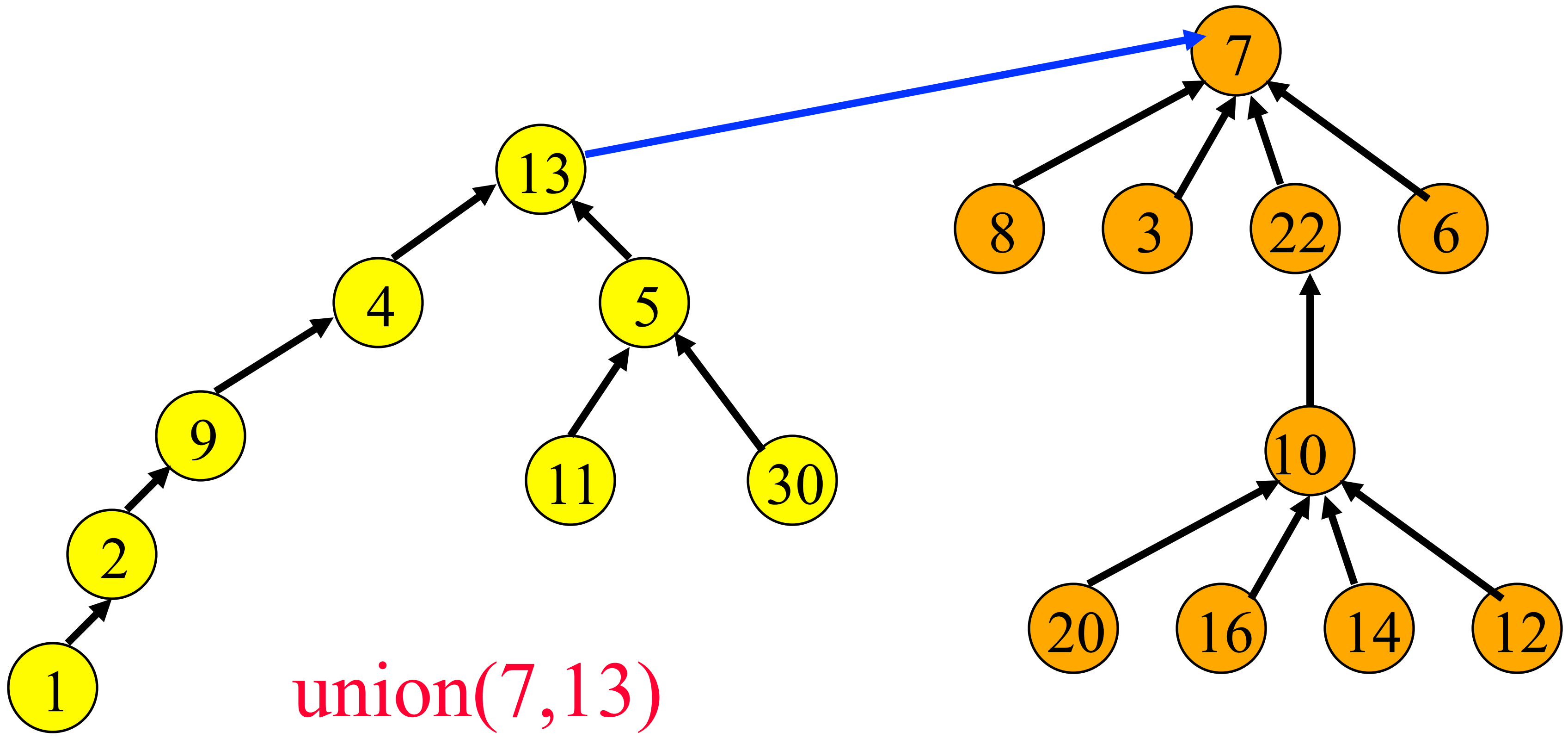
Height Rule

- Make tree with smaller height a subtree of the other tree.



Weight Rule

- Make tree with fewer number of elements a subtree of the other tree.



Implementation

- Root of each tree must record either its height or the number of elements in the tree.
- When a union is done using the height rule, the height increases only when two trees of equal height are united.
- When the weight rule is used, the weight of the new tree is the sum of the weights of the trees that are united.

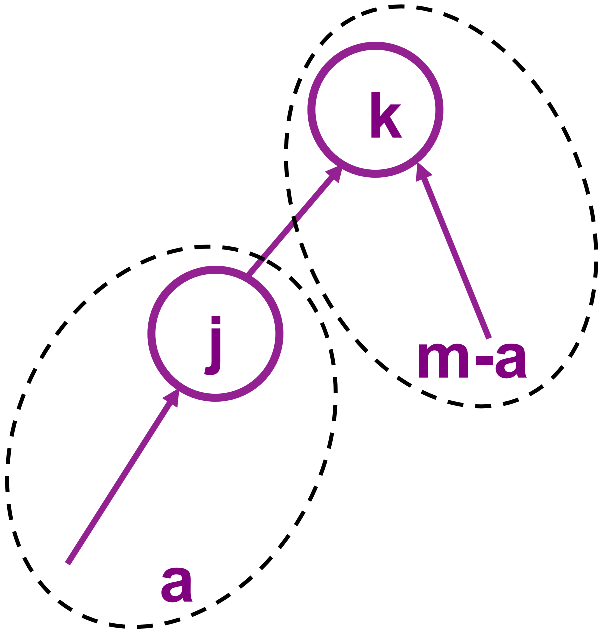
```
void Sets::WeightedUnion (int i, int j)
{ // Union sets with roots i and j, i≠j, weighting rule
  // parent[i] = - count[i] and parent[j] = - count[j]
    int temp = parent[i] + parent[j];
    if (parent[i] > parent[j]) {   // i has fewer nodes
        parent[i] = j;
        parent[j] = temp;
     }
    else {    // j has fewer nodes
        parent[j] = i;
         parent[i] = temp;
    }
}
```

Height Of A Tree

- Suppose we start with single element trees and perform unions using either the height or the weight rule.
- **Lemma 5.5** The height of a tree with m elements is at most floor $(\log_2 m) + 1$.

- **Lemma 5.5** The height of a tree with m elements is at most floor $(\log_2 m) + 1$.
- **Proof by induction:**
  - **m = 1, it is true.**
  - **Assume it is true for all trees with i ≤ m-1 nodes.**
  - **For i = m, let T be a tree with m nodes created by WeightedUnion.**
  - **Consider the last union performed,**
    - **Union(k, j).**

Let a be the number of nodes in tree j and m-a that in tree k. without loss of generality, assume $1 \le a \le m/2$. Then the height of T is either the same as that of k or is 1 + that of j.



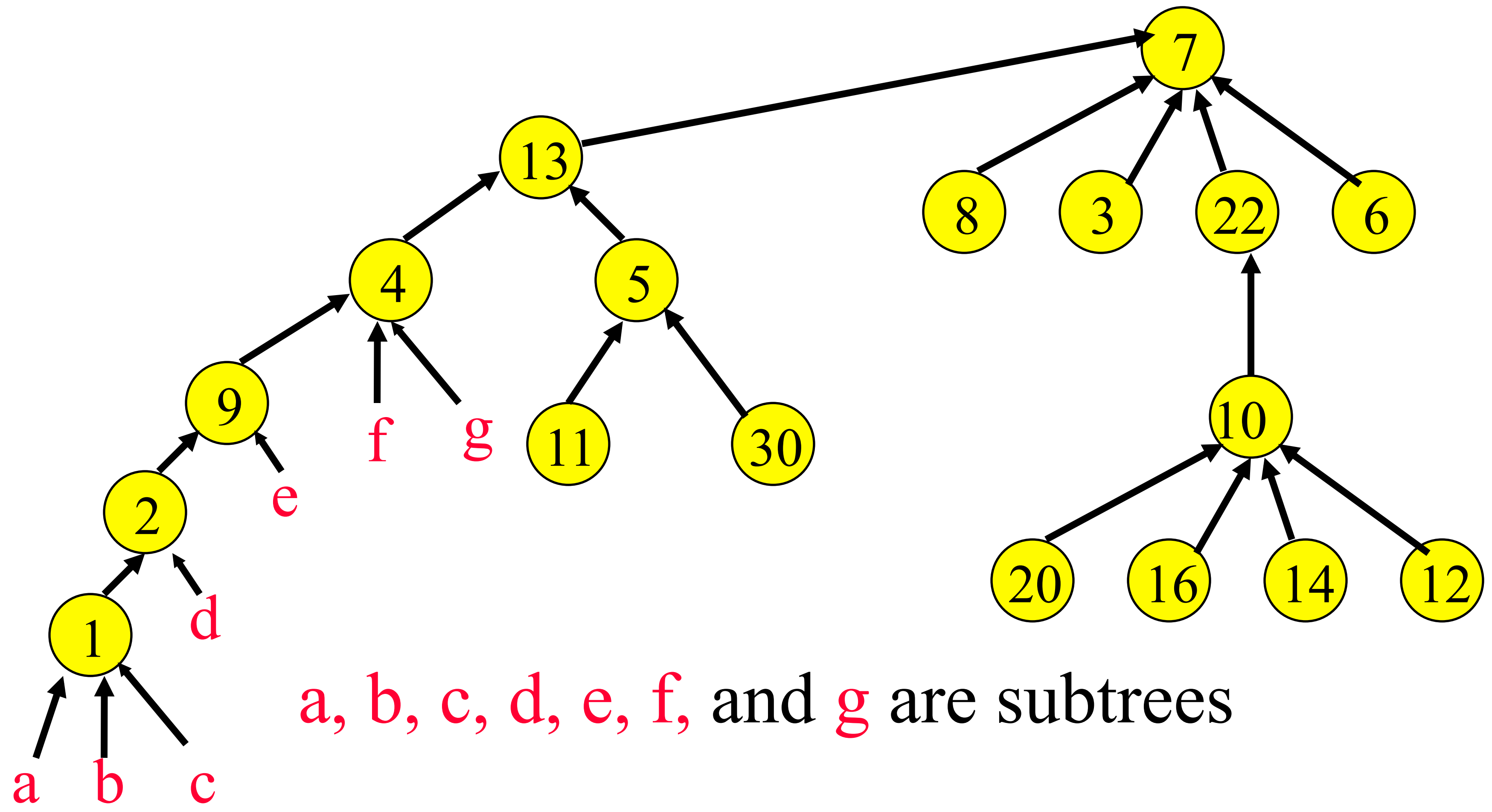
$m\text{-}a \ge m/2 \ge a$

If the former is the case, the height of T $\le \lfloor \log_2 (m-a) \rfloor + 1 \le \lfloor \log_2 m \rfloor + 1$.

If the latter is the case, the height of T $\le \lfloor \log_2 a \rfloor + 2 \le \lfloor \log_2 m/2 \rfloor + 2 \le \lfloor \log_2 m \rfloor + 1$.

- **The time to process a find is at most O(log n) in a tree of n nodes**
- **If an intermixed sequence of u-1 union and f find is to be done**
- **The worst case time is O(u + f log u).**

Sprucing Up The Find Method



a, b, c, d, e, f, and g are subtrees

- find(1)
- Do additional work to make future finds easier.

Path Collapsing/Path Compaction

Further improvement in the find algorithm.

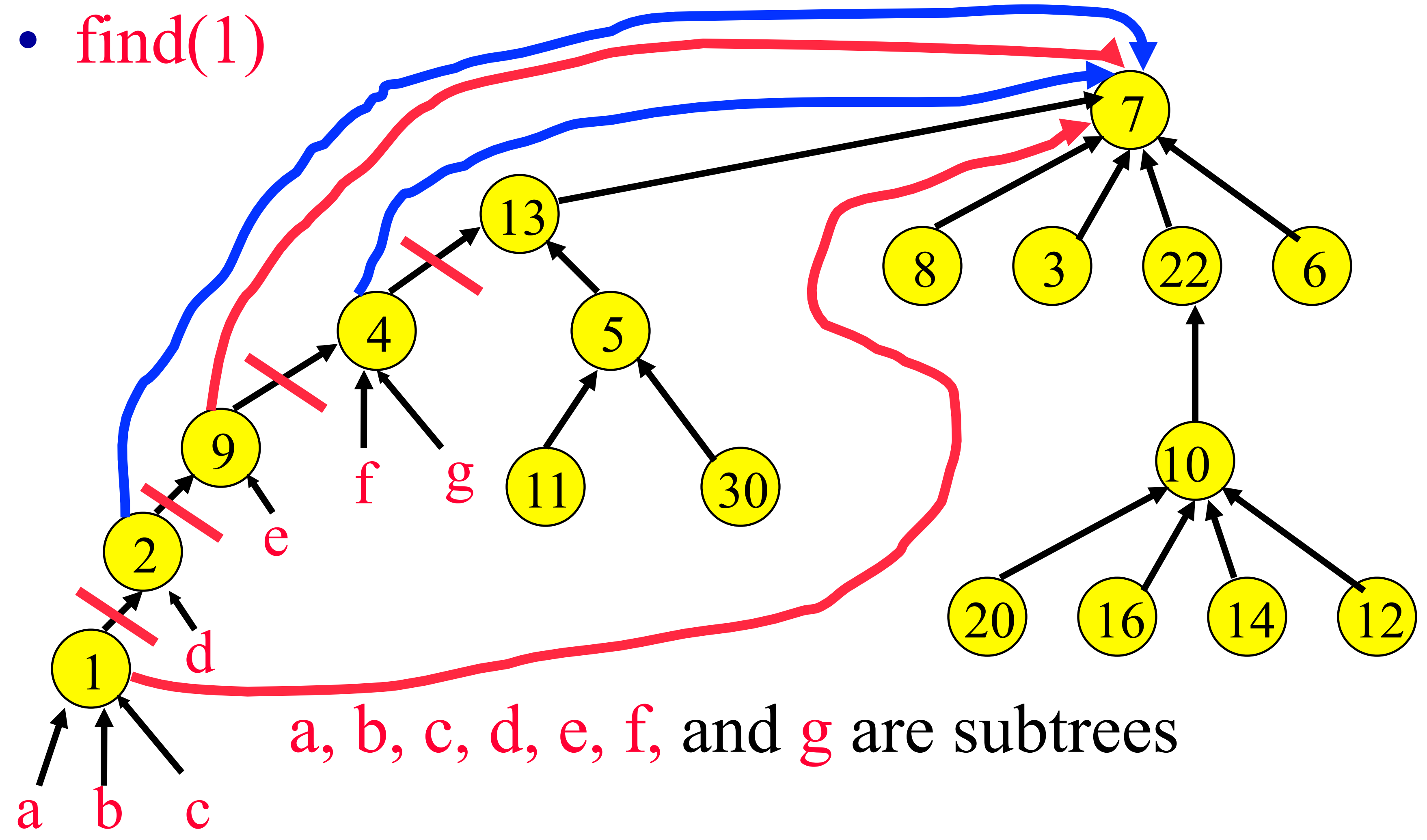
Definition [Collapsing rule] :

If j is a node on the path from i to its root and parent[i] ≠ root(i),

then set parent[j] to root(i).

Path Collapsing/Path Compaction

- Make all nodes on find path point to tree root.
- find(1)



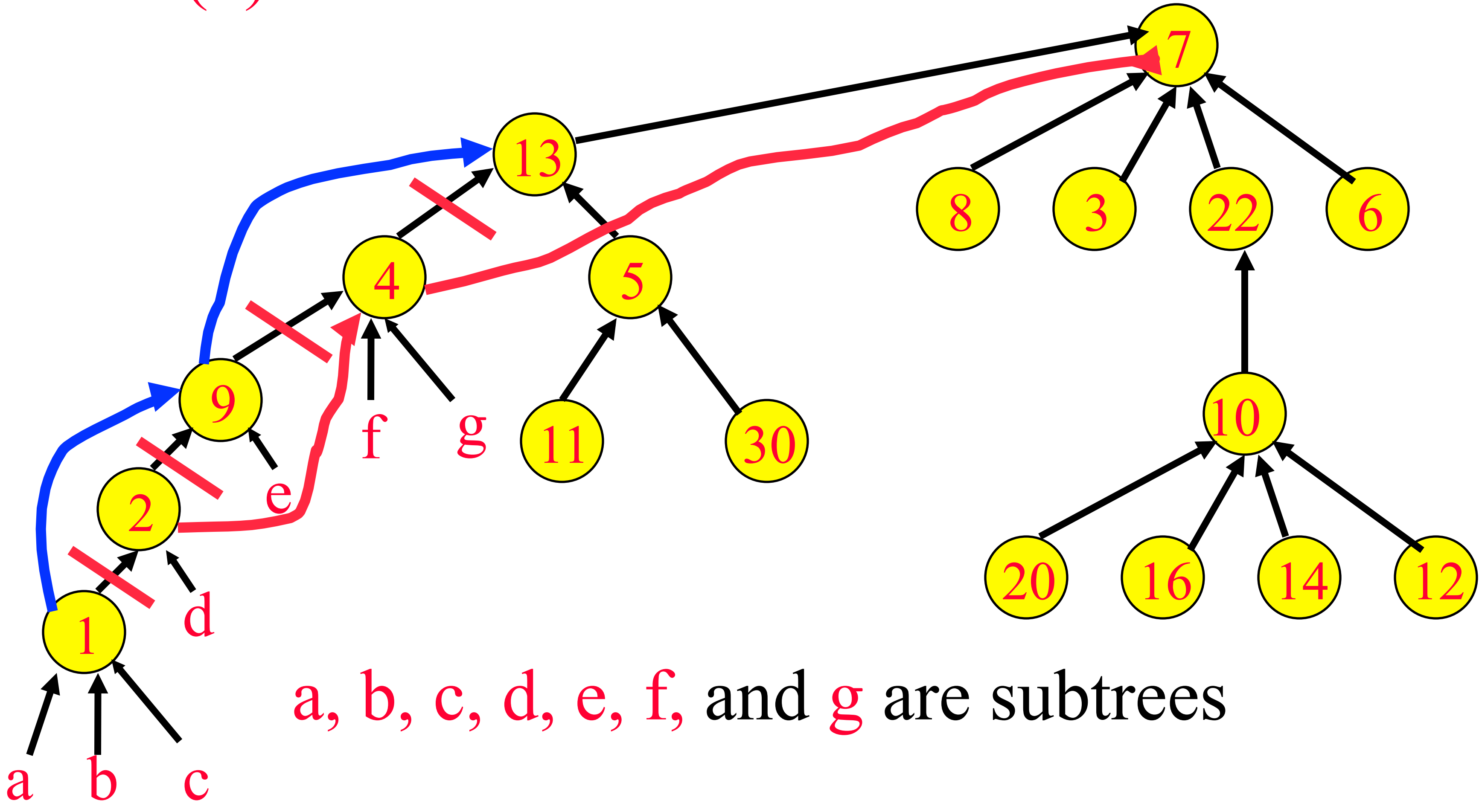
a, b, c, d, e, f, and g are subtrees

Makes two passes up the tree.

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

Path Splitting

- Nodes on find path point to former grandparent.
- find(1)

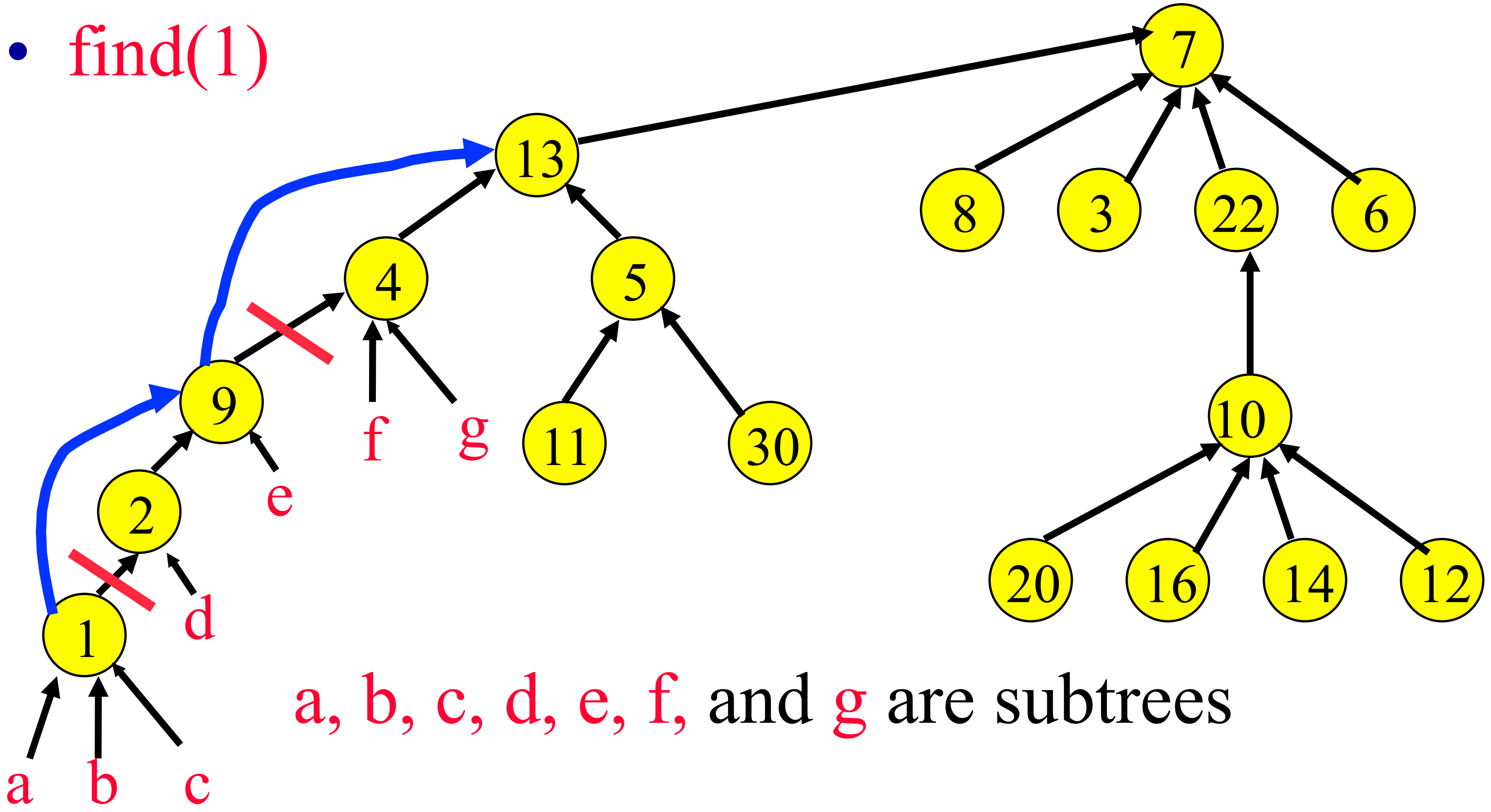


a, b, c, d, e, f, and g are subtrees

Makes only one pass up the tree.

Path Halving

- Parent pointer in every other node on find path is changed to former grandparent.
- find(1)



a, b, c, d, e, f, and g are subtrees

Changes half as many pointers.

Application to Equivalent Classes

equivalence classes $\Leftrightarrow$ disjoint sets

Initially, parent[i] = -1, $0 \le i \le n-1$.

To process $i \equiv j$,

Let x = find(i), y = find(j) --- 2 finds

If $x \neq y$ then union(x, y) --- at most 1 union

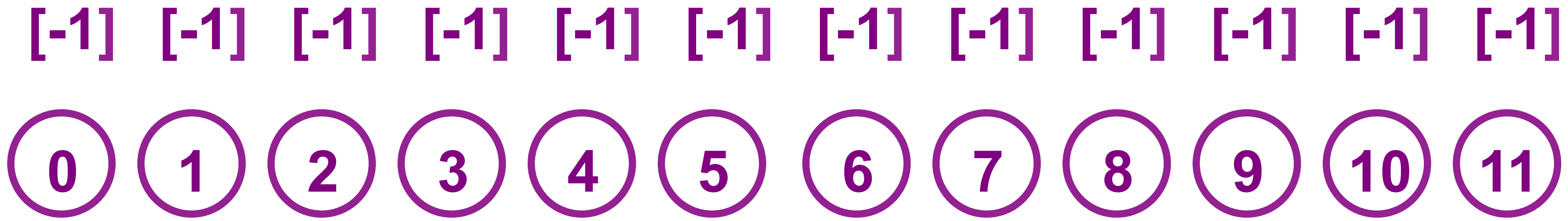
Thus if we have n elements and m equivalence pairs, we needs 2m finds and min {n-1, m} unions. The total time is $O(n+2m\ \alpha(n+2m,n))$.

Example:

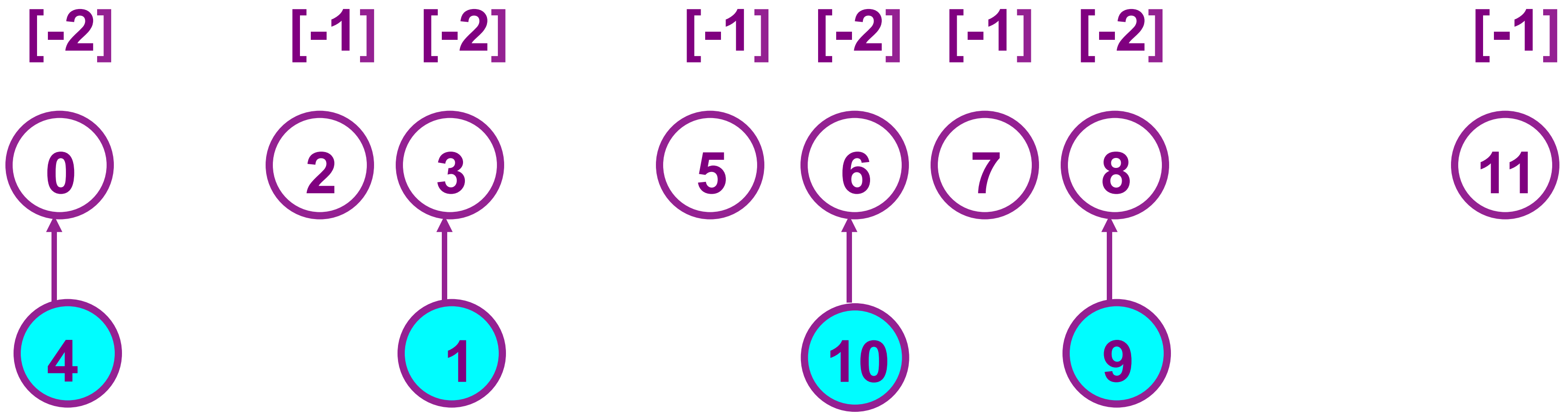
n = 12, process equivalence pairs:

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

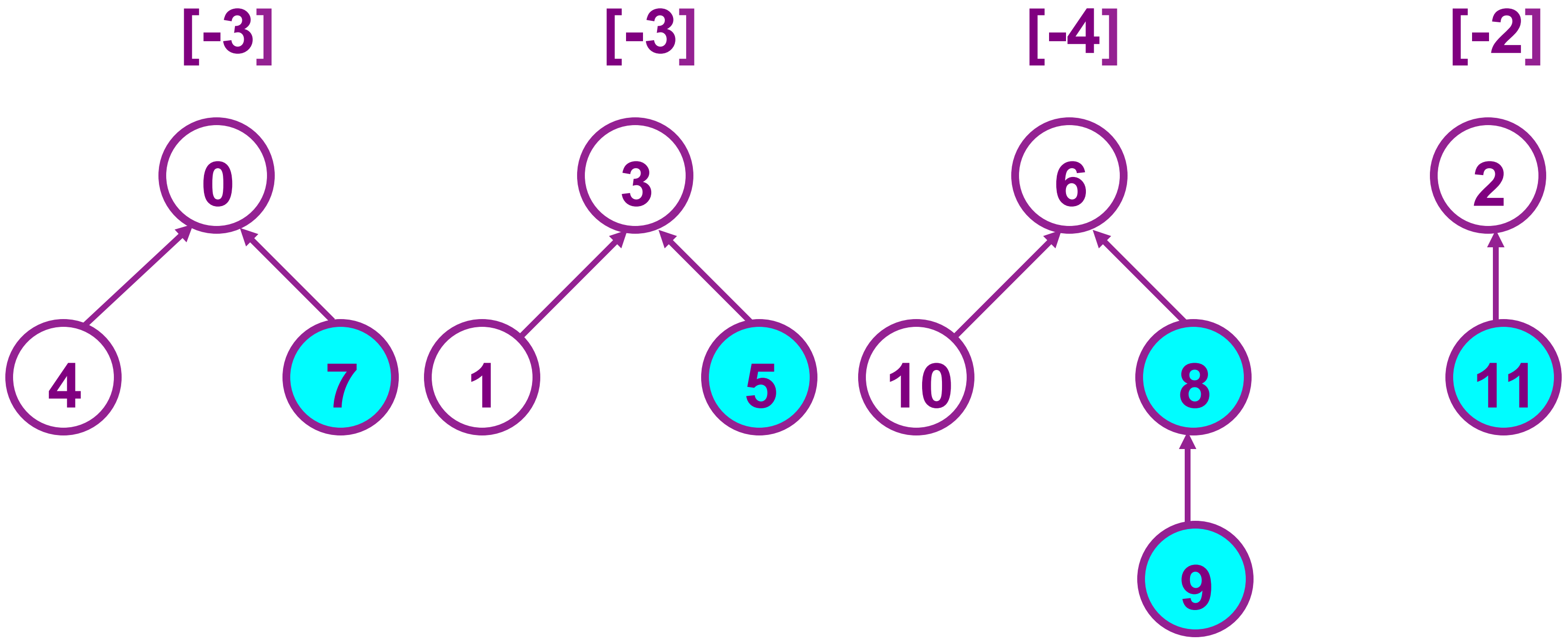
11 ≡ 0



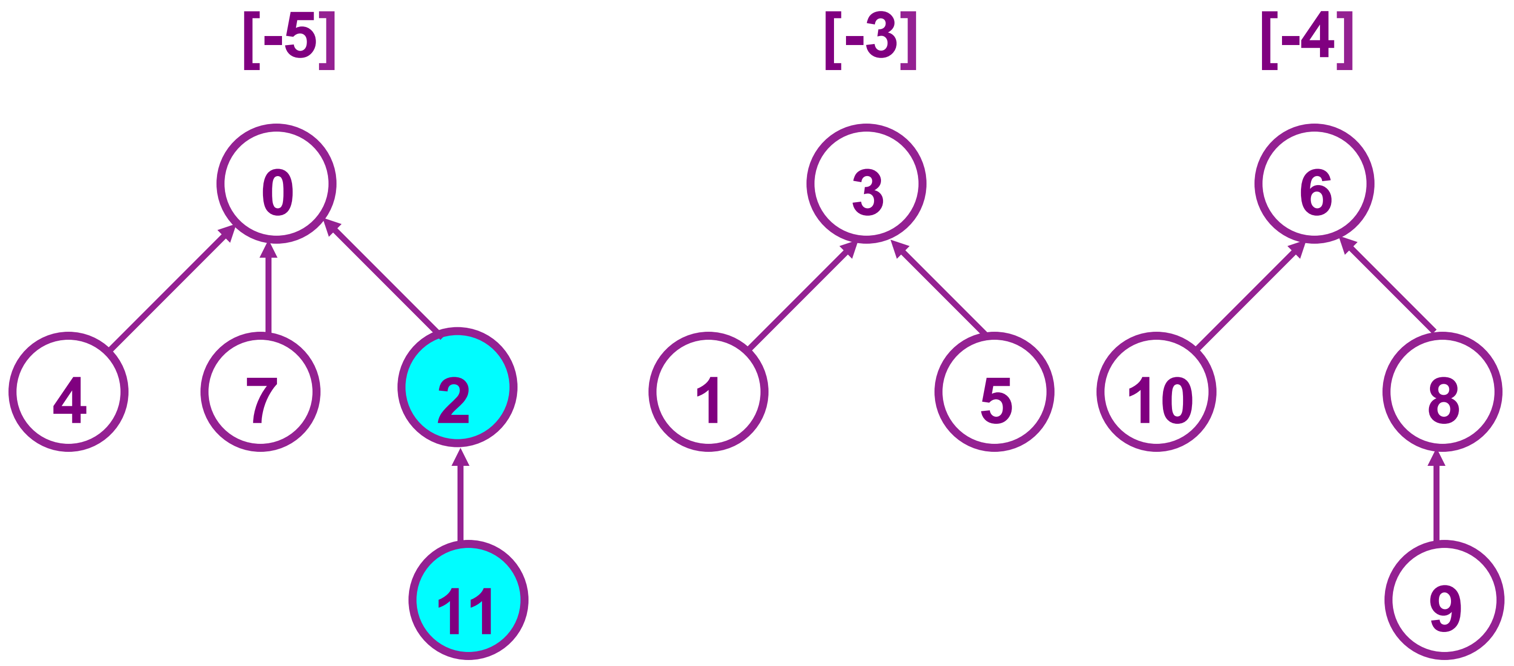
(a) Initial trees



(b) After processing 0 ≡ 4, 3 ≡ 1, 6 ≡10, and 8 ≡ 9



(c) After processing 7 ≡ 4, 6 ≡ 8, 3 ≡ 5, and 2 ≡ 11



(d) After processing 11 ≡ 0

Thinking ……

- Application
  - Coding: msg → 101011100001
- What we have
  - Dictionary of Message: n words
  - Every word w_i has an average frequency f_i
- Requirement
  - For an message, minimize its code length

Problem formulation

- n elements
- Each element has a length l_i
- Each element has a frequency f_i
- Binary coding
  - unfixed length
- Wanted:
  - Larger $f_i \rightarrow$ smaller l_i
  - $\mathrm{Min}(\Sigma(l_i * f_i))$

Huffman Tree

- Binary tree with n leaves
- Frequency: leaf value
- Coding length
  - Distance from the root
- Wanted:
  - Larger f_i → deeper level
  - Min($\Sigma(l_i * f_i)$)
- Word coding……

Huffman Tree

- ADT
- Algorithm
    - Tree Construction
    - Coding
    - decoding

Building a Tree

Scan the original text

Eerie eyes seen near lake.

- What characters are present?

E e r i space
y s n a r l k .

Building a Tree

Scan the original text

Eerie eyes seen near lake.

- What is the frequency of each character in the text?

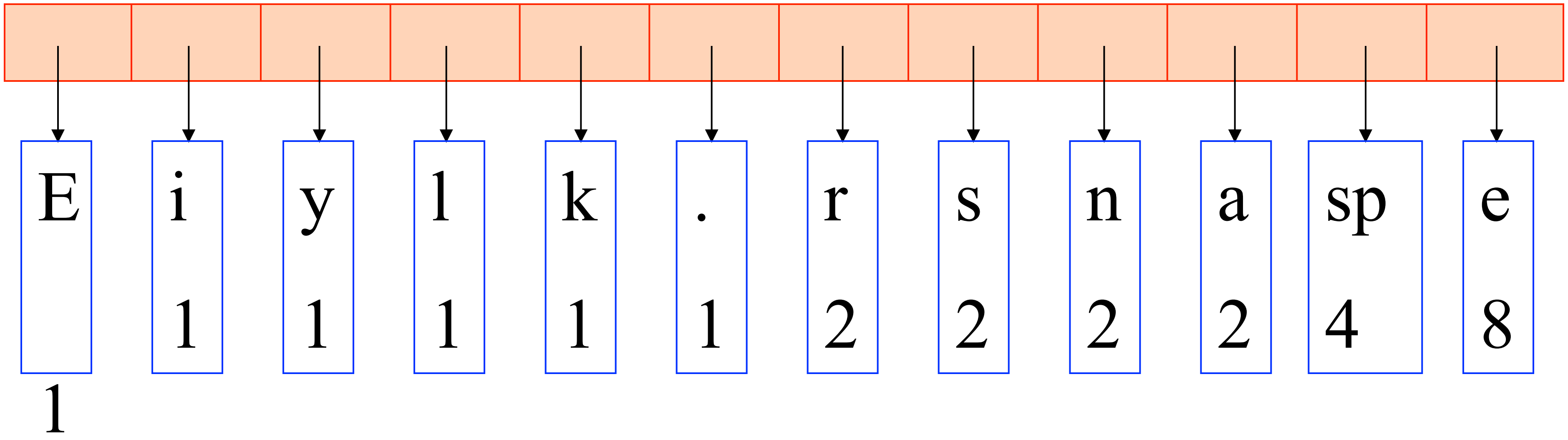
Char	Freq.	Char	Freq.	Char	Freq.
E	1	y	1	k	1
e	8	s	2	.	1
r	2	n	2		
i	1	a	2		
space	4	l	1		

Building a Tree
Prioritize characters

- Create binary tree nodes with character and frequency of each character
- Place nodes in a priority queue
  - The <u>lower</u> the occurrence, the higher the priority in the queue

Building a Tree

- The queue after inserting all nodes

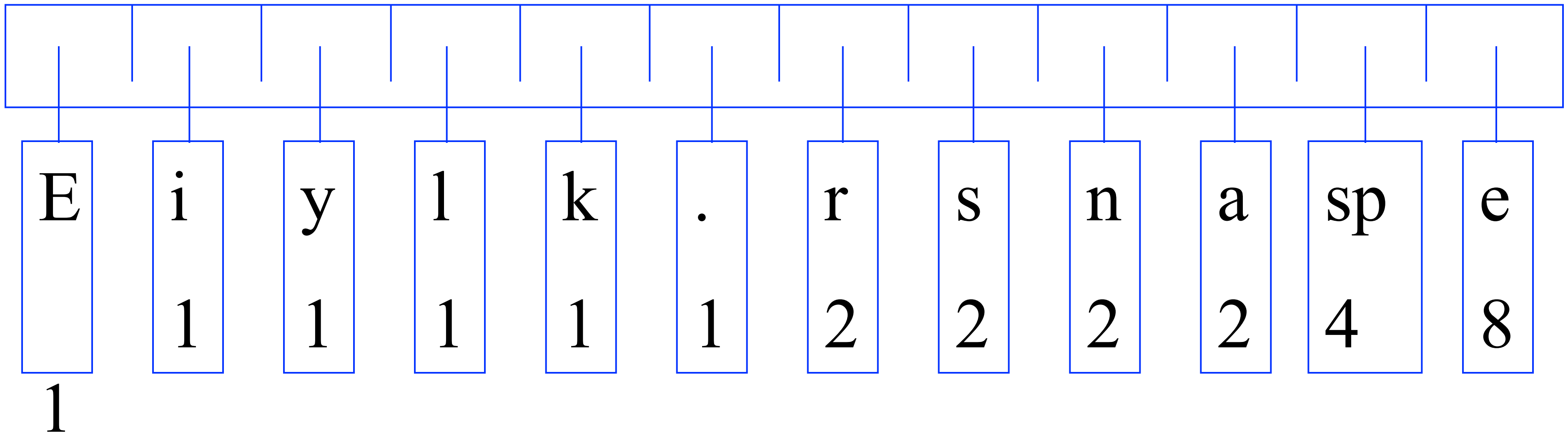


- Null Pointers are not shown

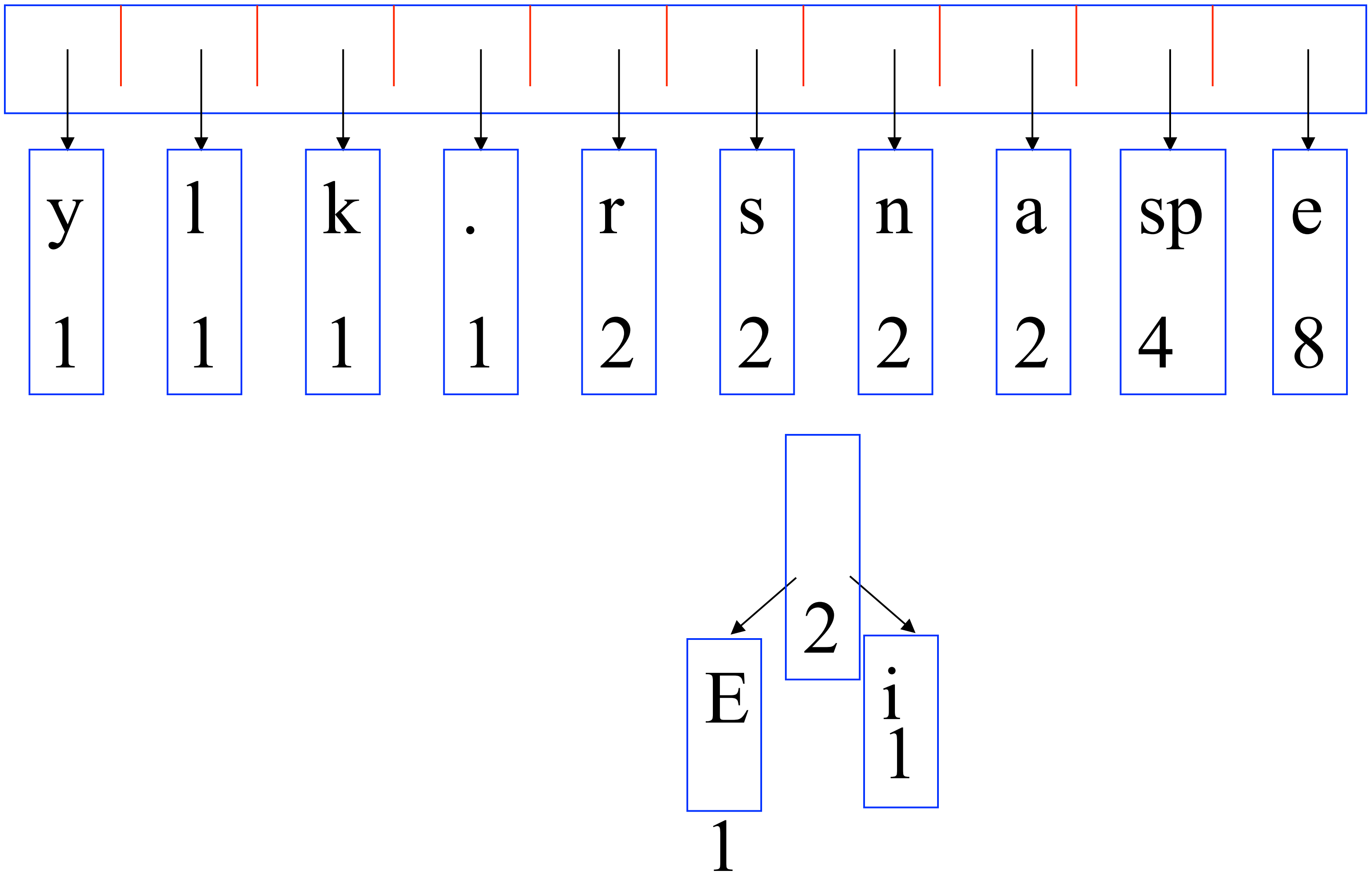
Building a Tree

- While priority queue contains two or more nodes
  - Create new node
  - Dequeue node and make it left subtree
  - Dequeue next node and make it right subtree
  - Frequency of new node equals sum of frequency of left and right children
  - Enqueue new node back into queue

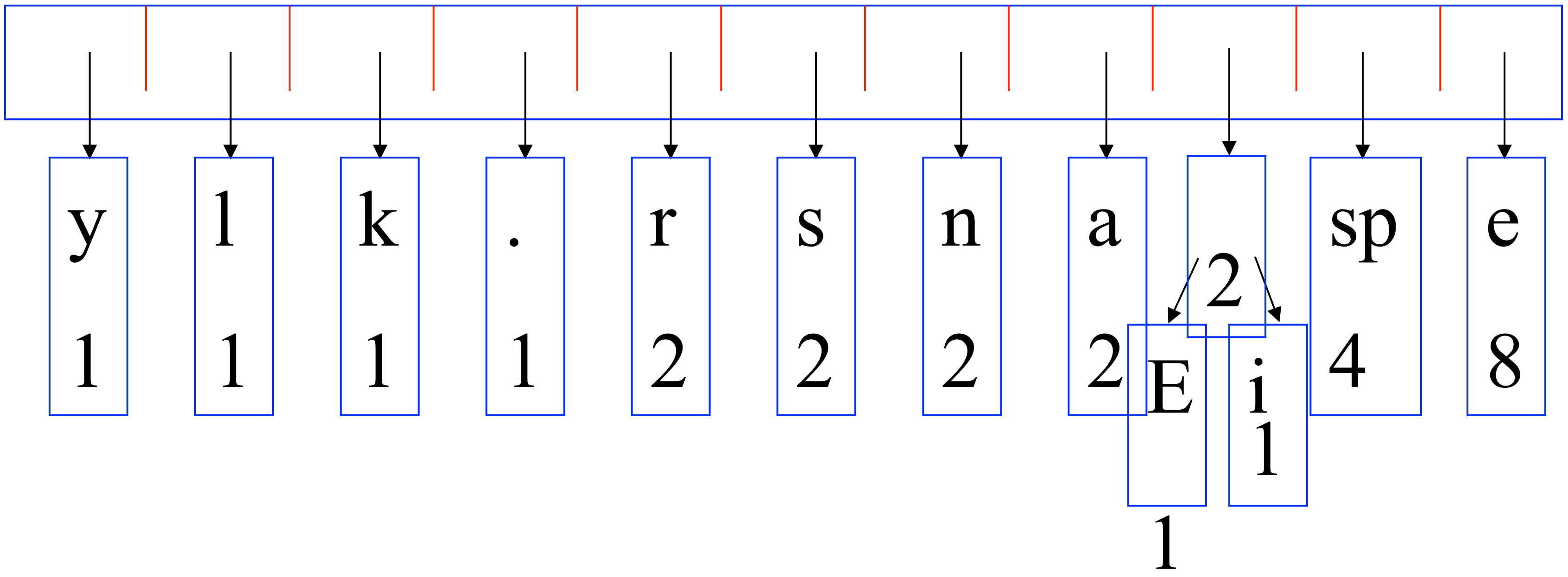
Building a Tree



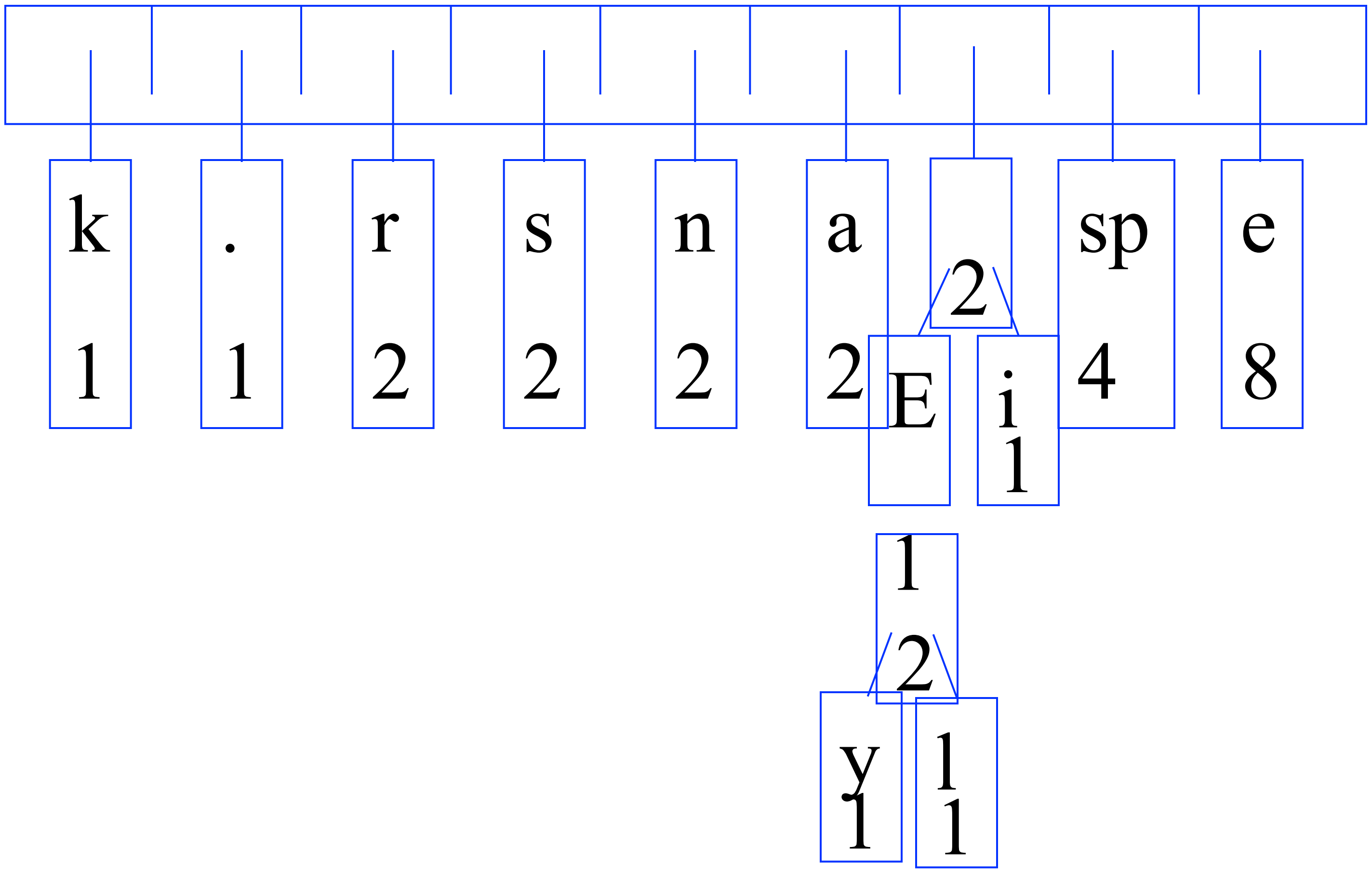
Building a Tree



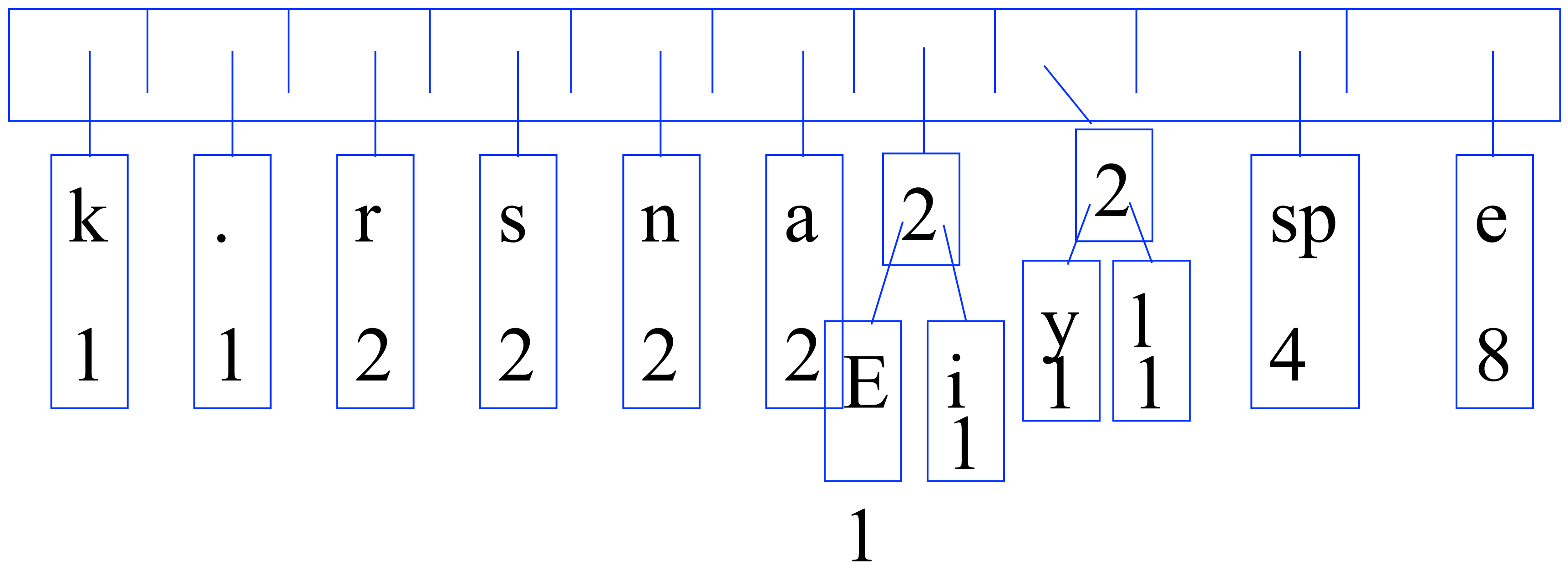
Building a Tree



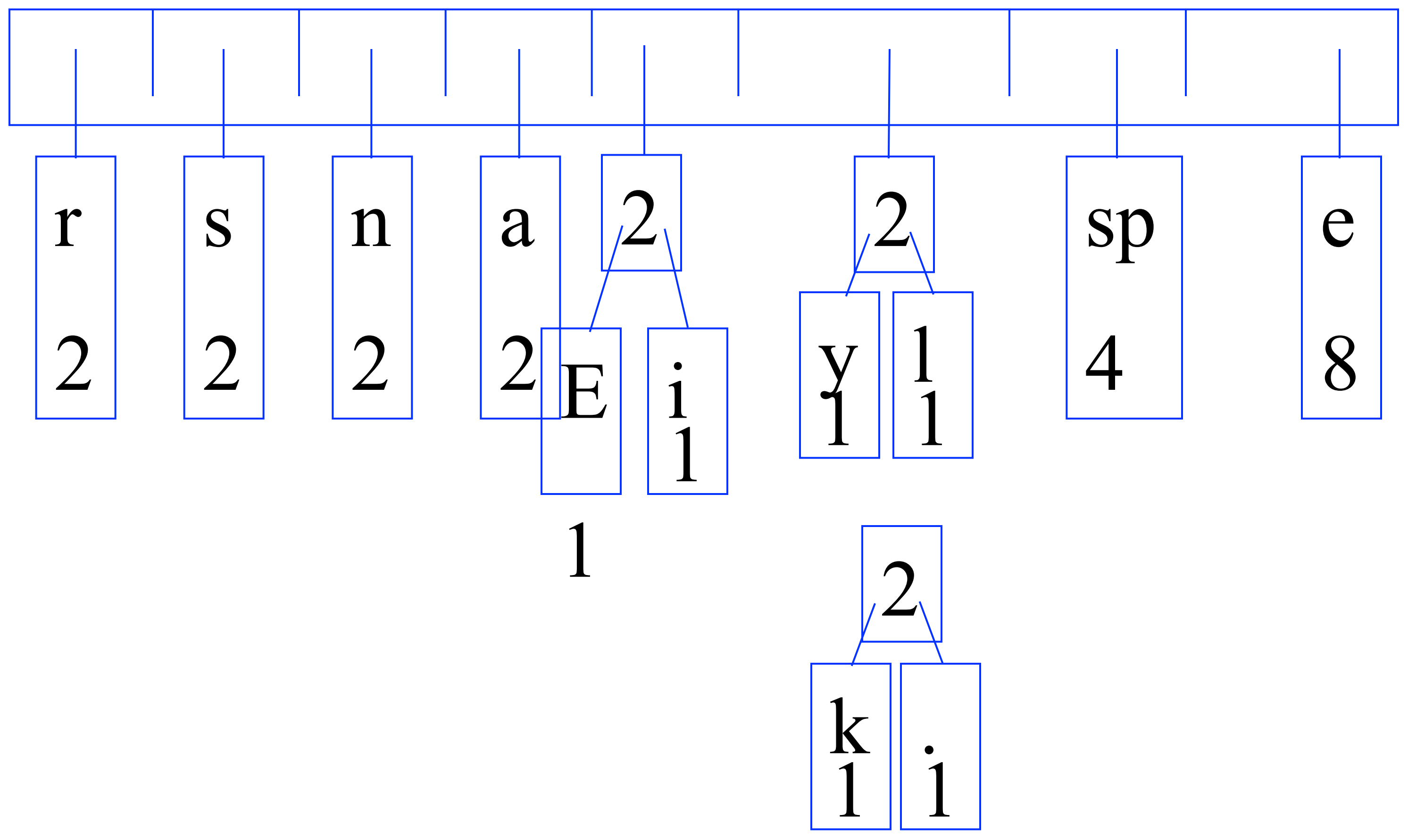
Building a Tree



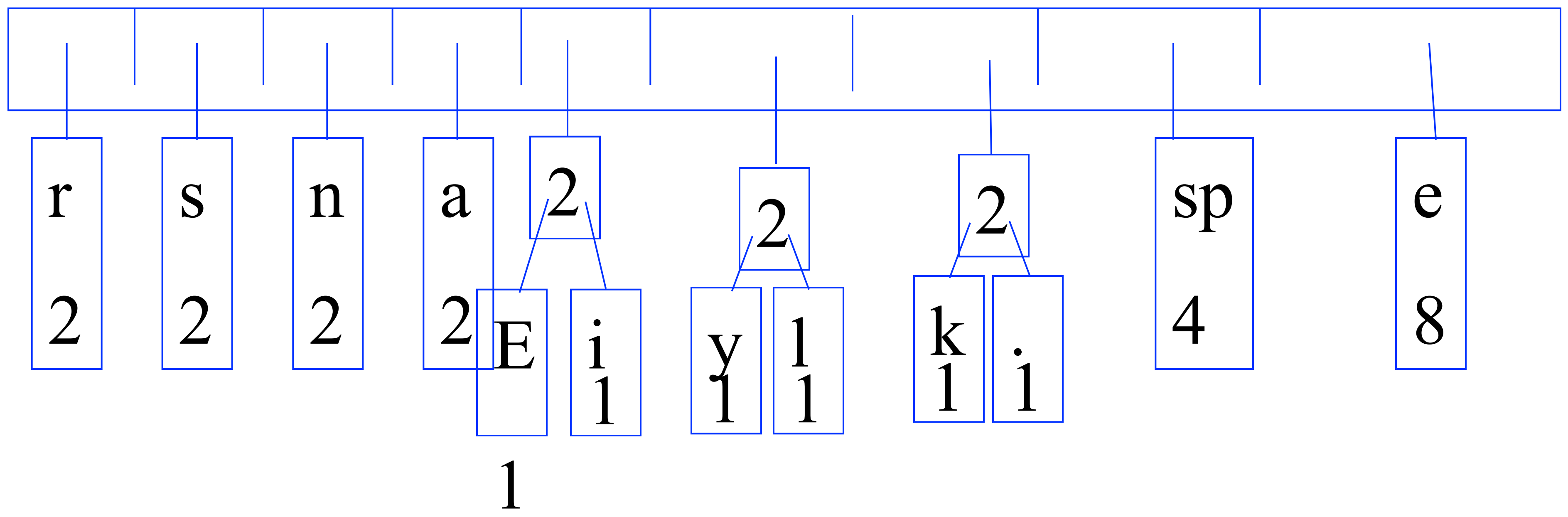
Building a Tree



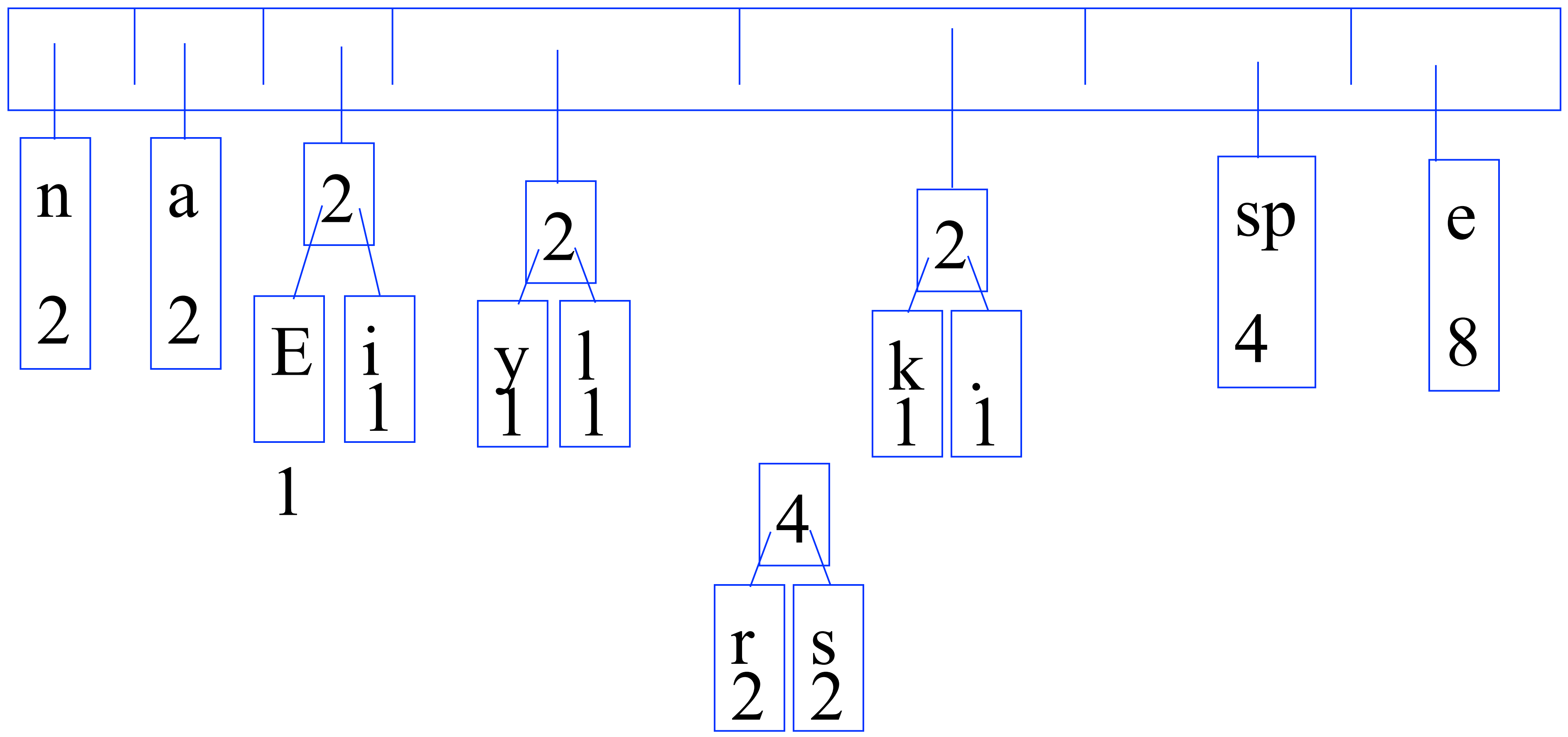
Building a Tree



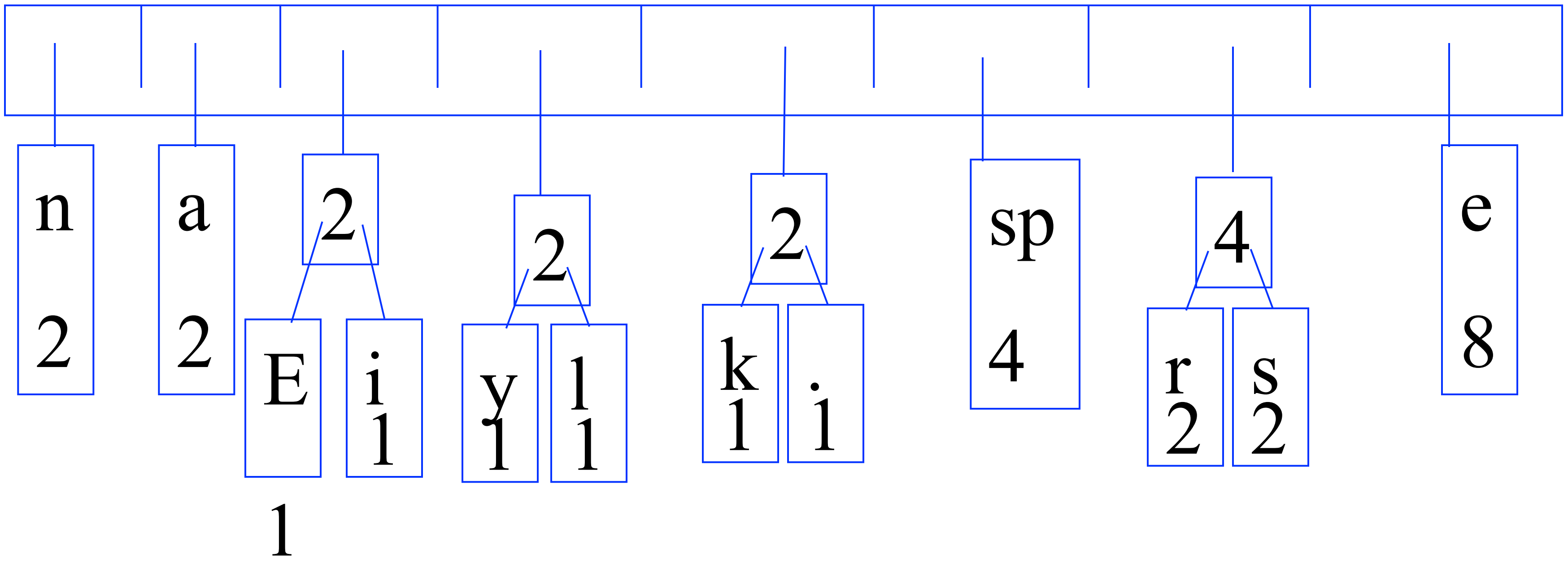
Building a Tree



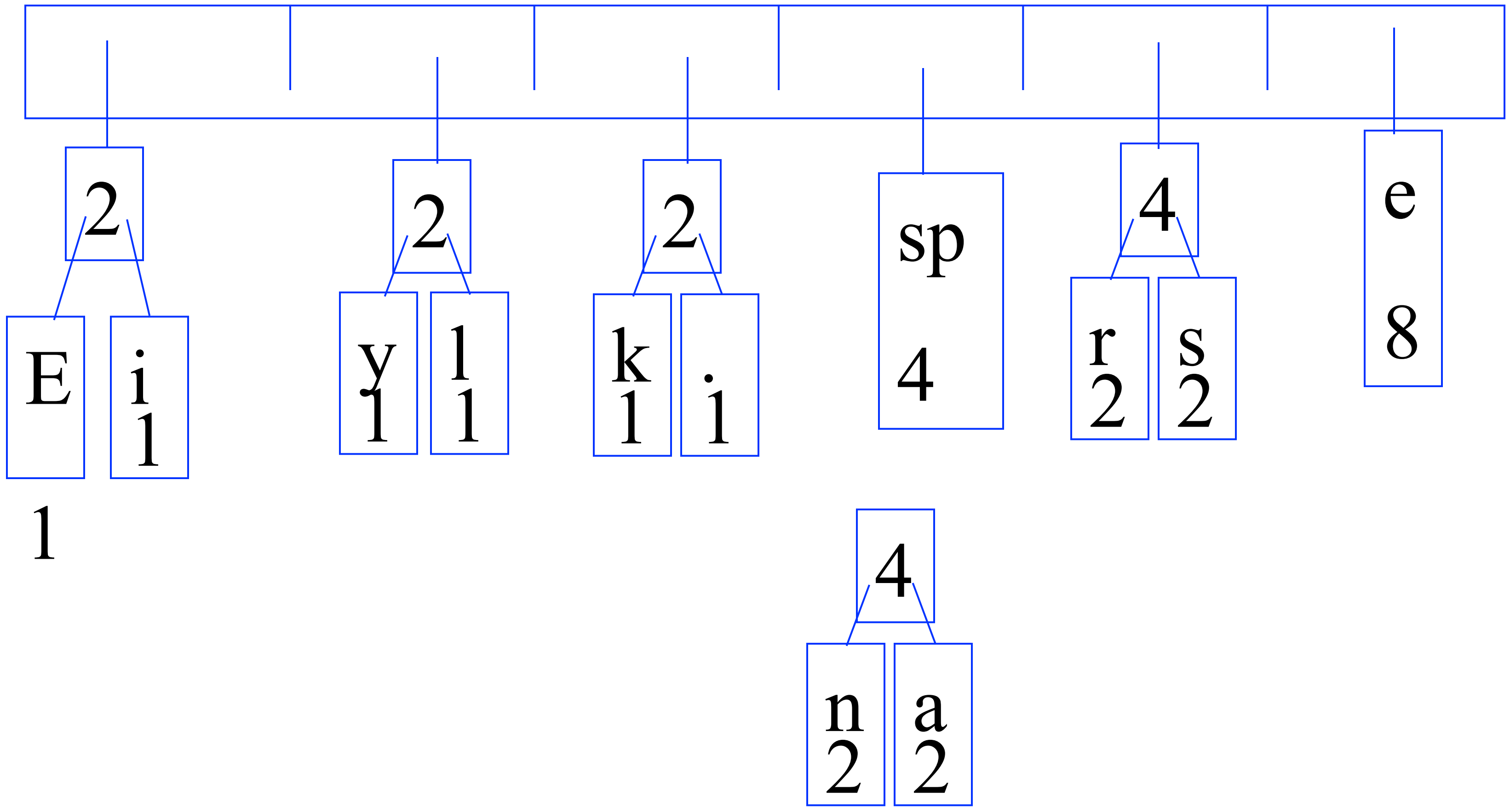
Building a Tree



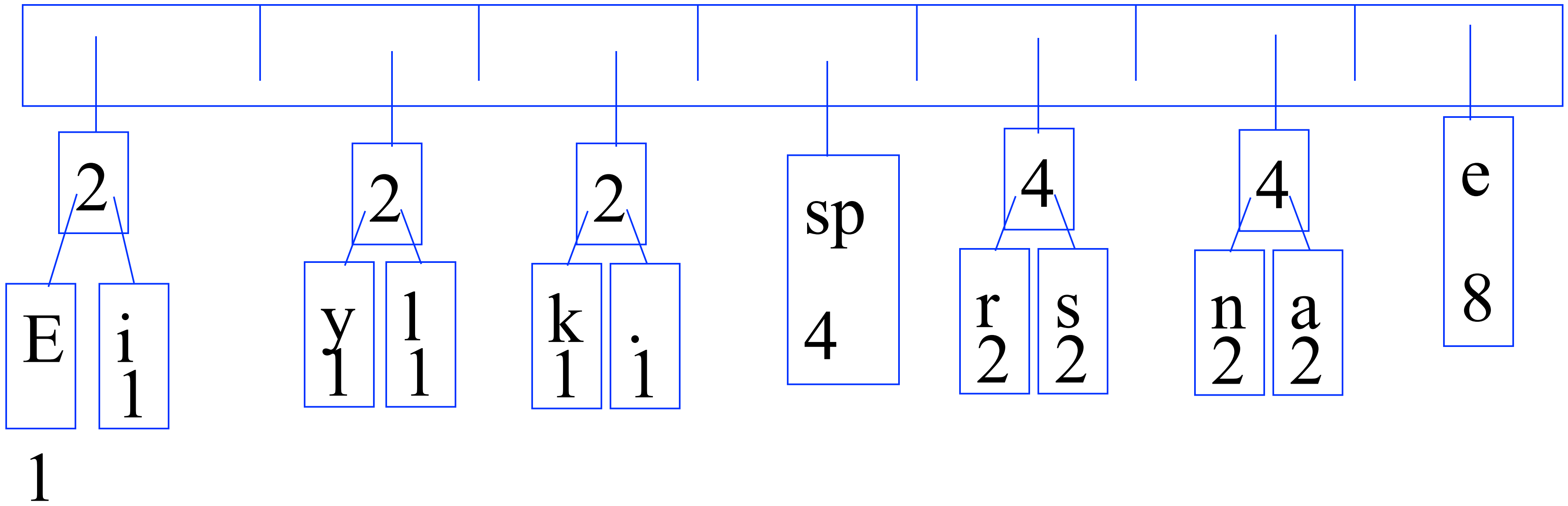
Building a Tree



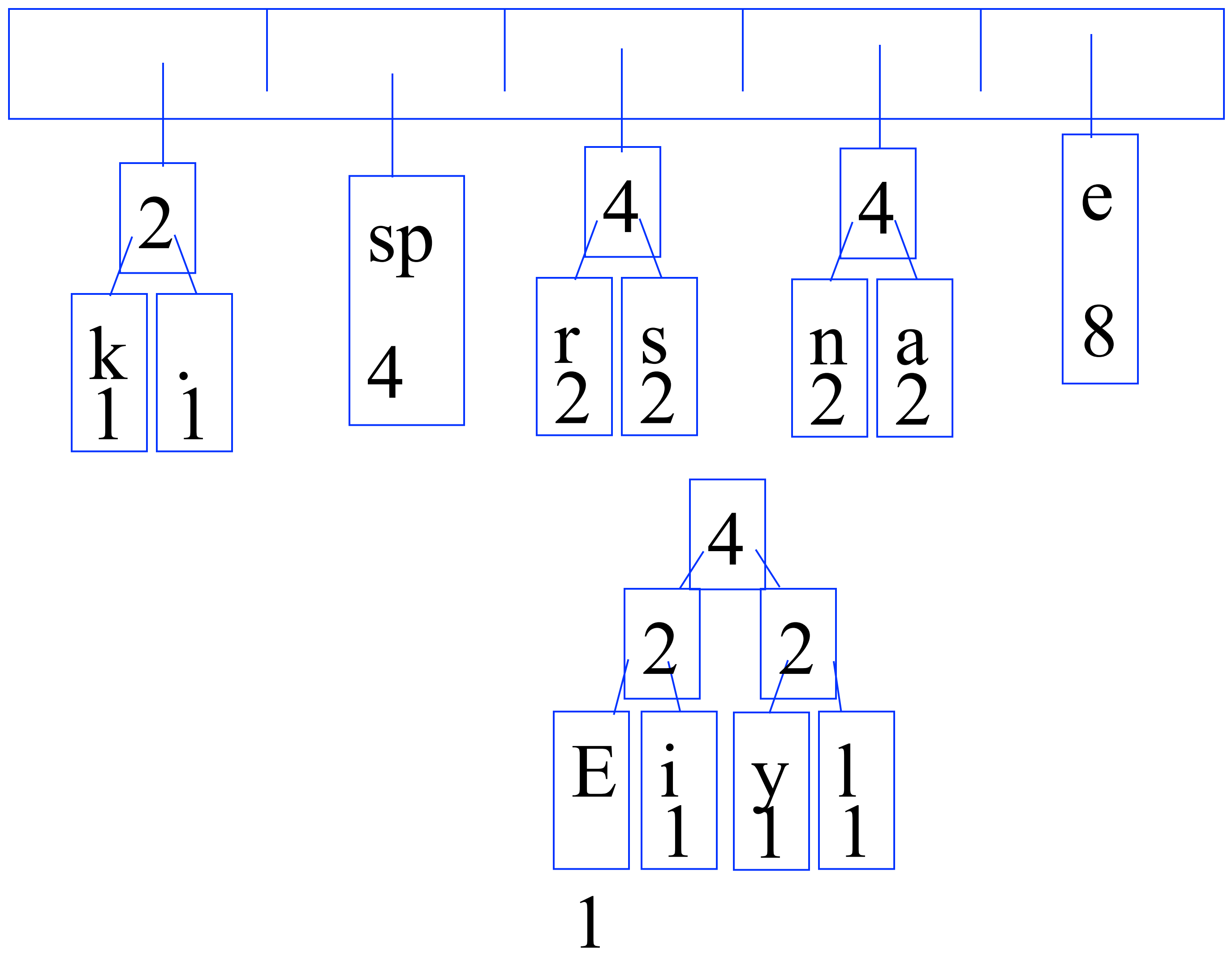
Building a Tree



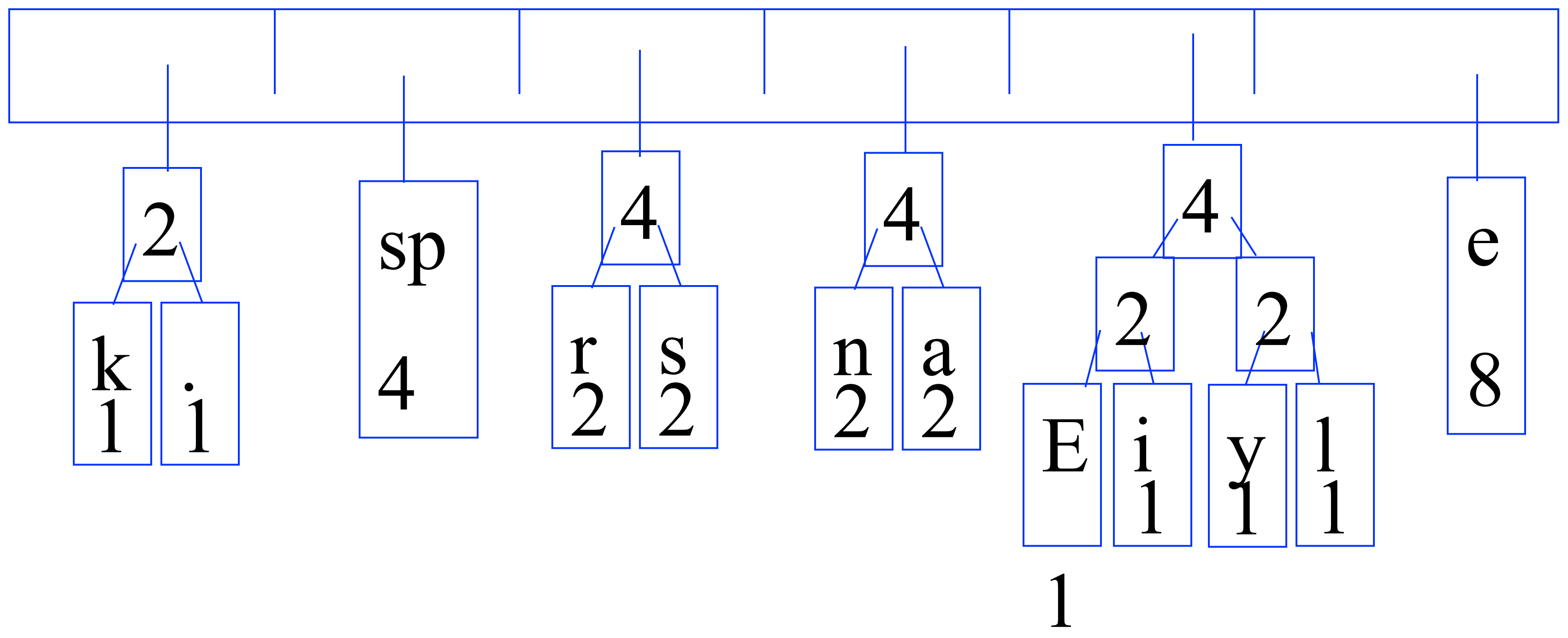
Building a Tree



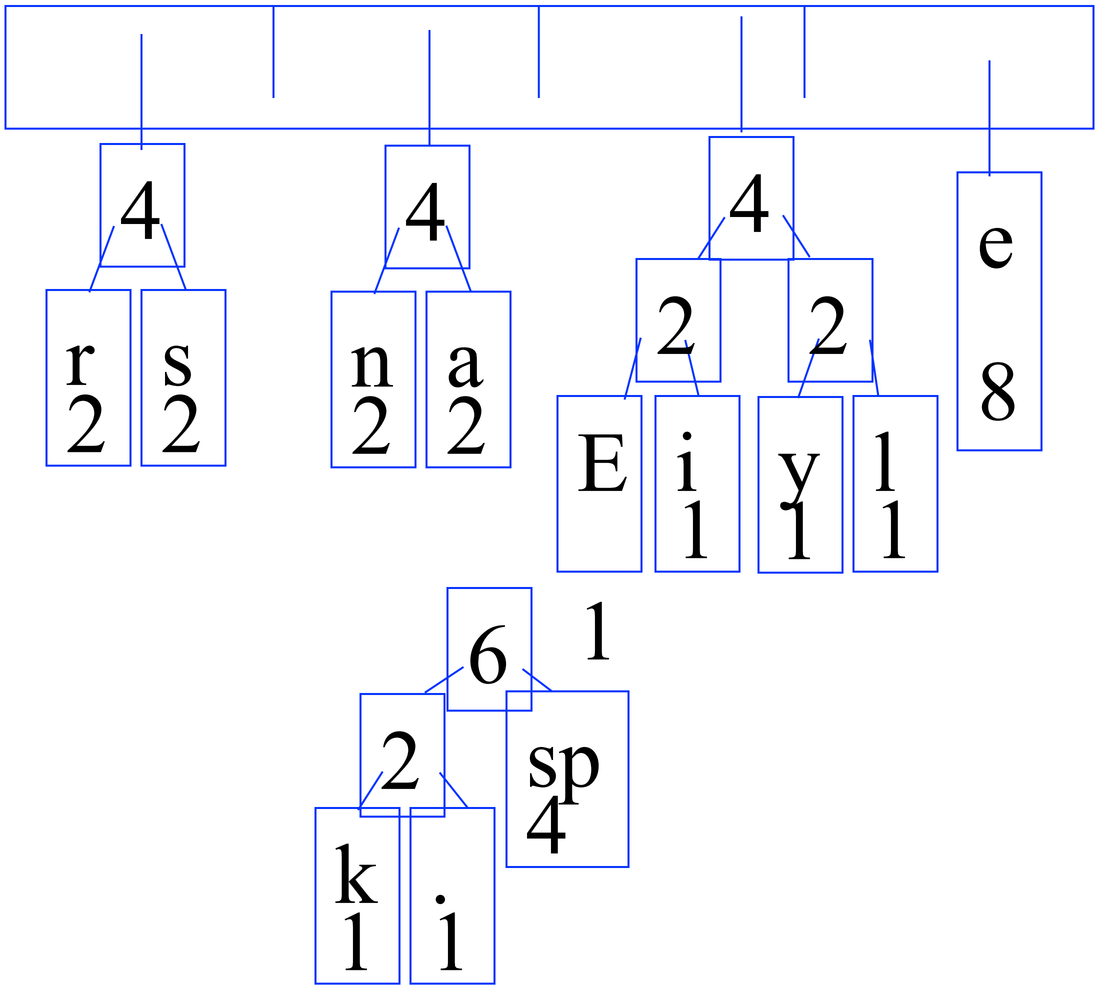
Building a Tree



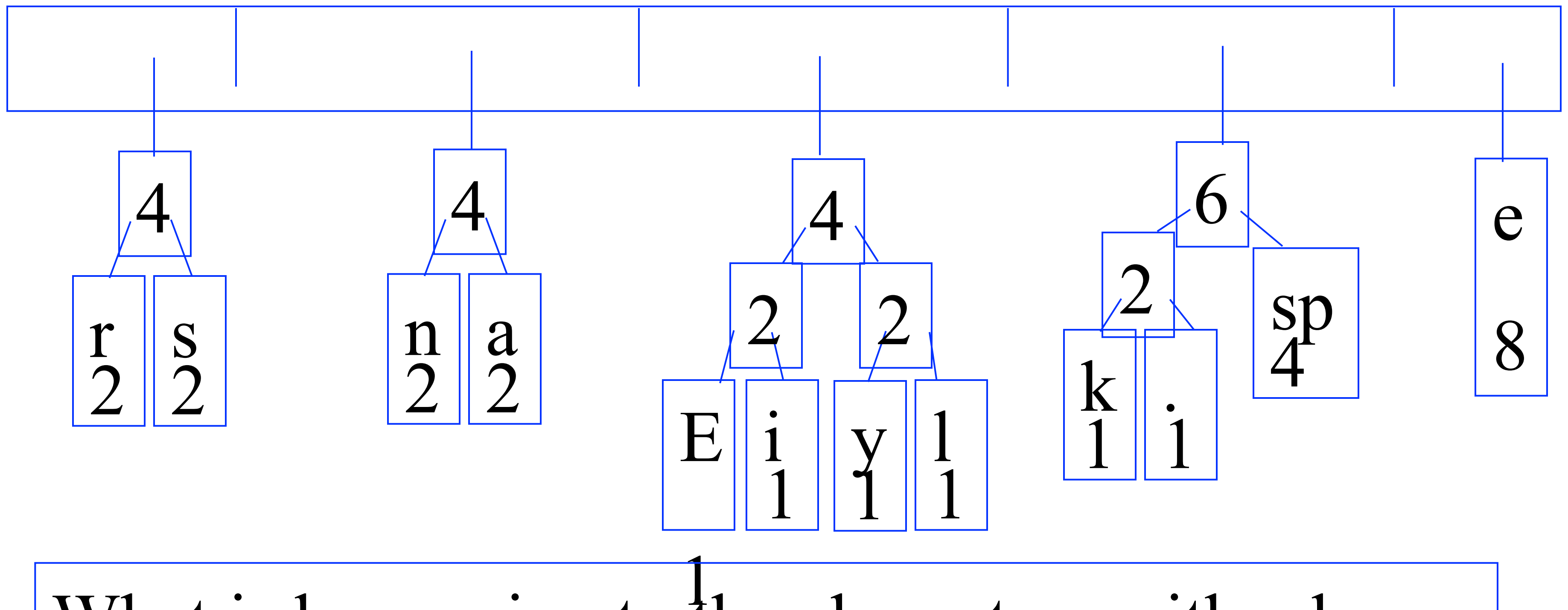
Building a Tree



Building a Tree

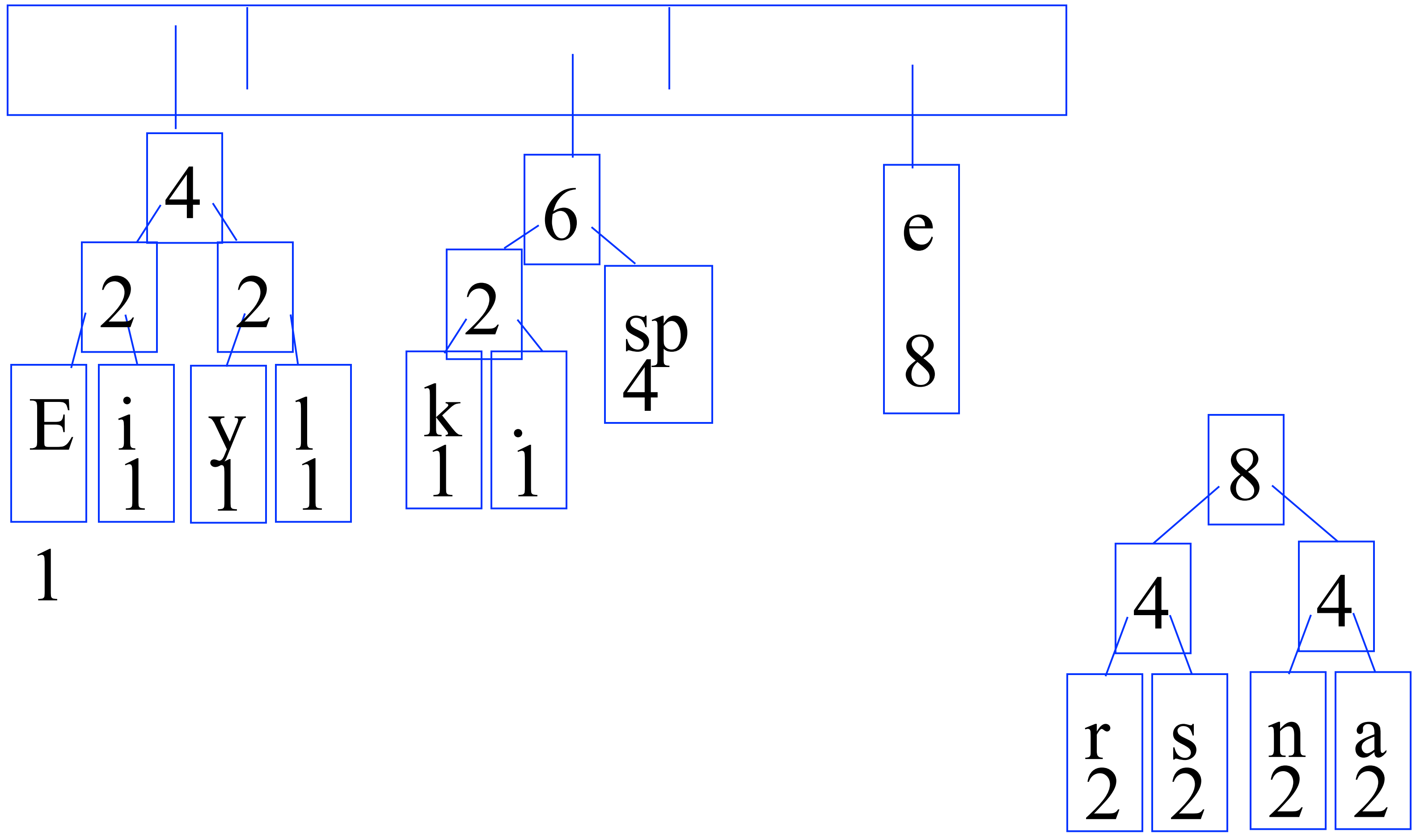


Building a Tree

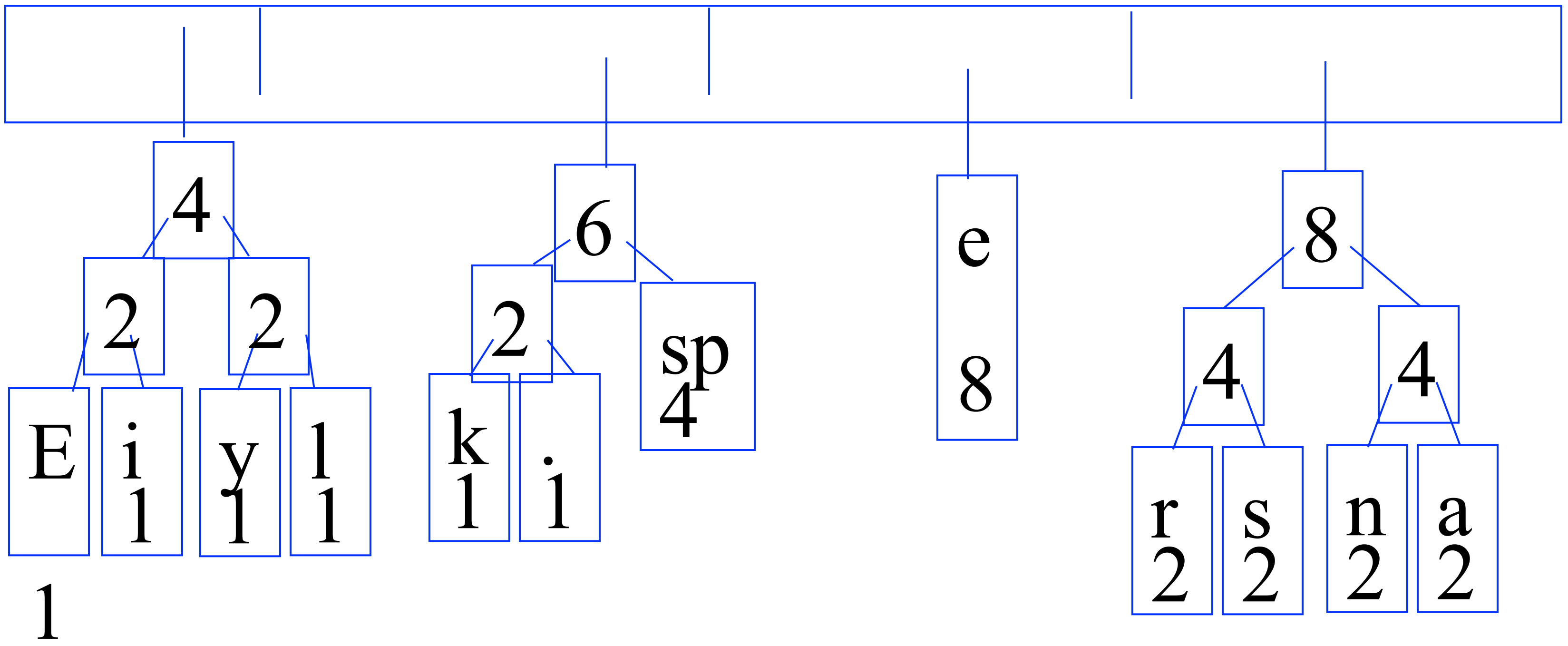


What is happening to the characters with a low number of occurrences?

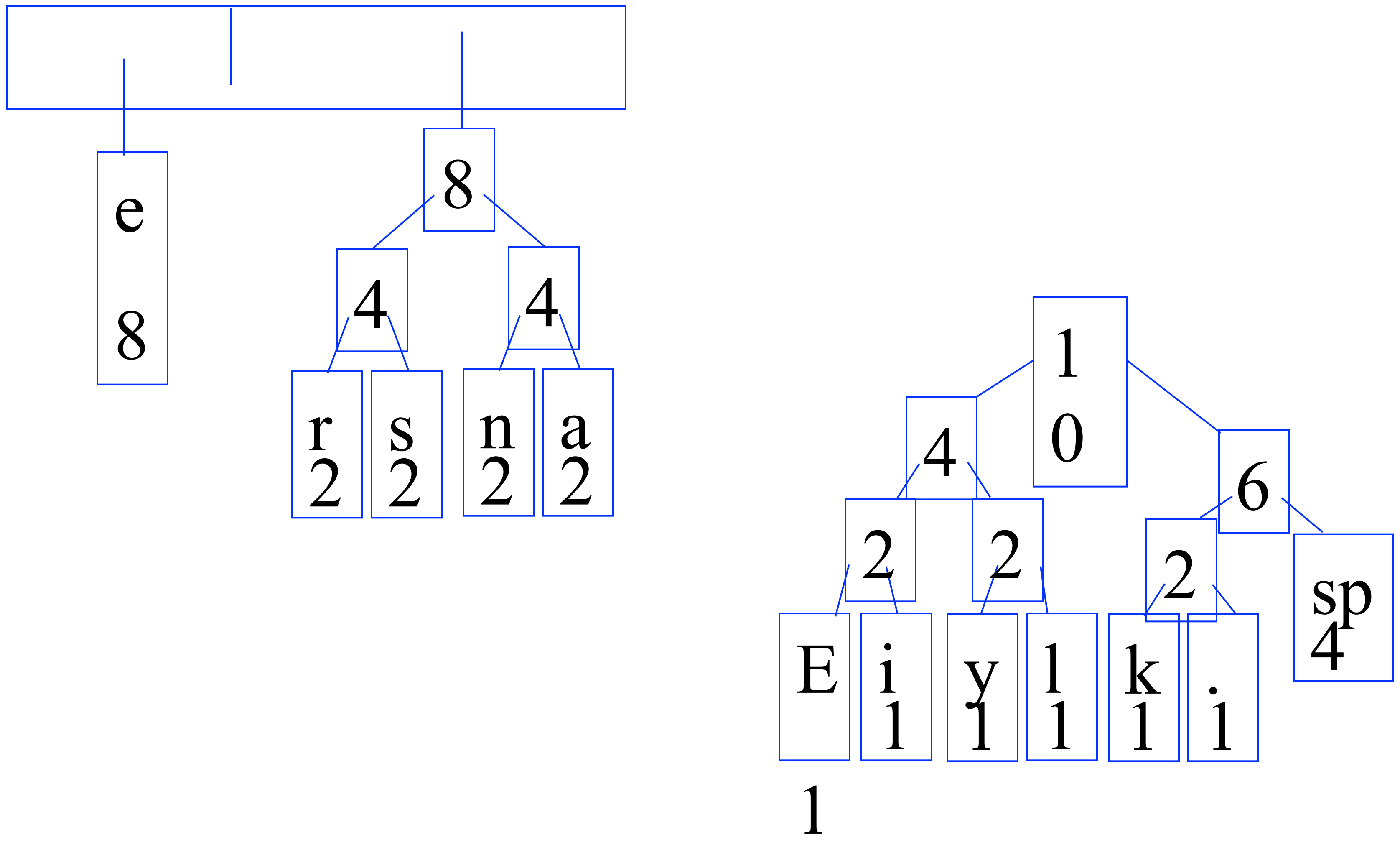
Building a Tree



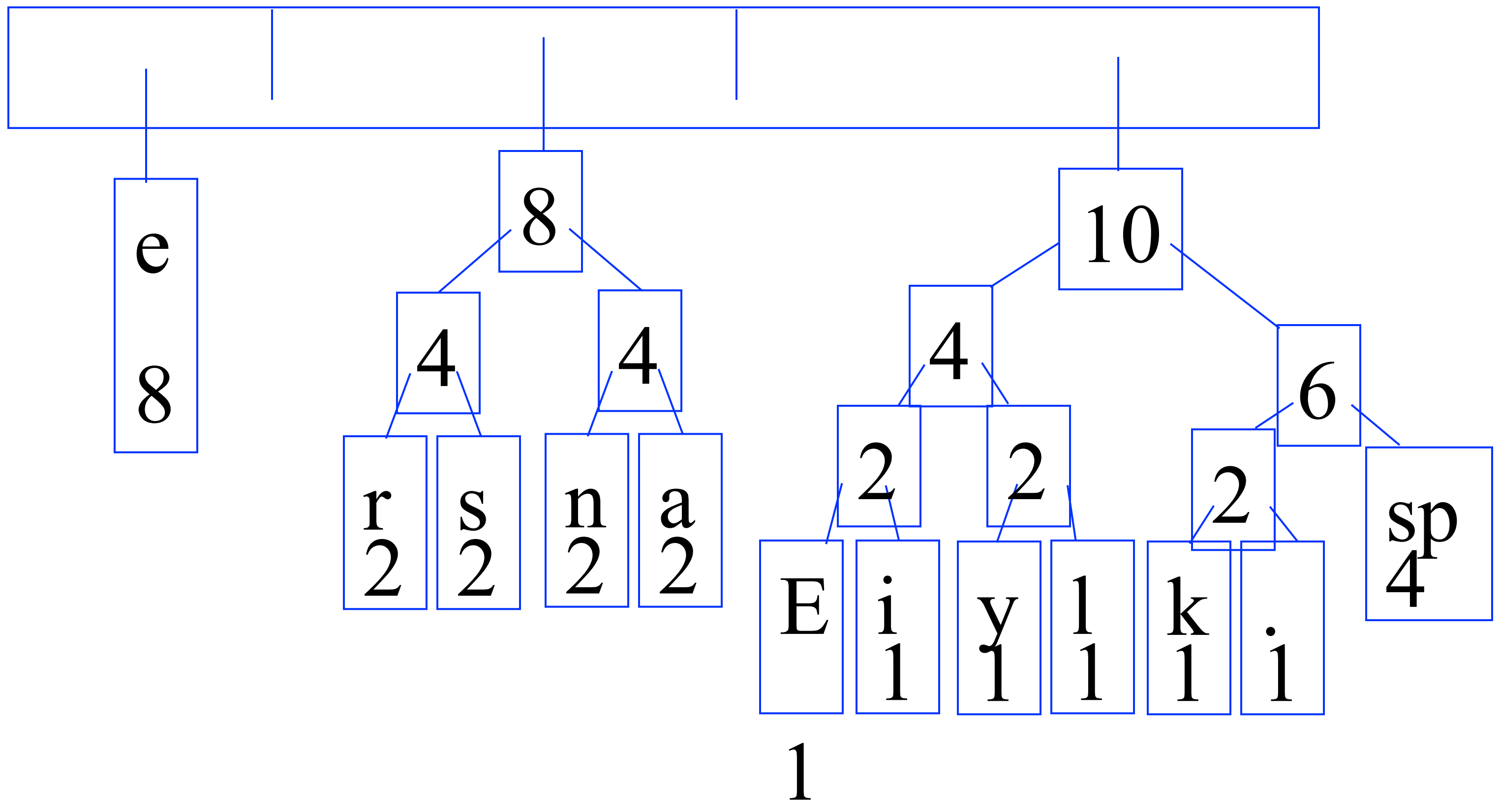
Building a Tree



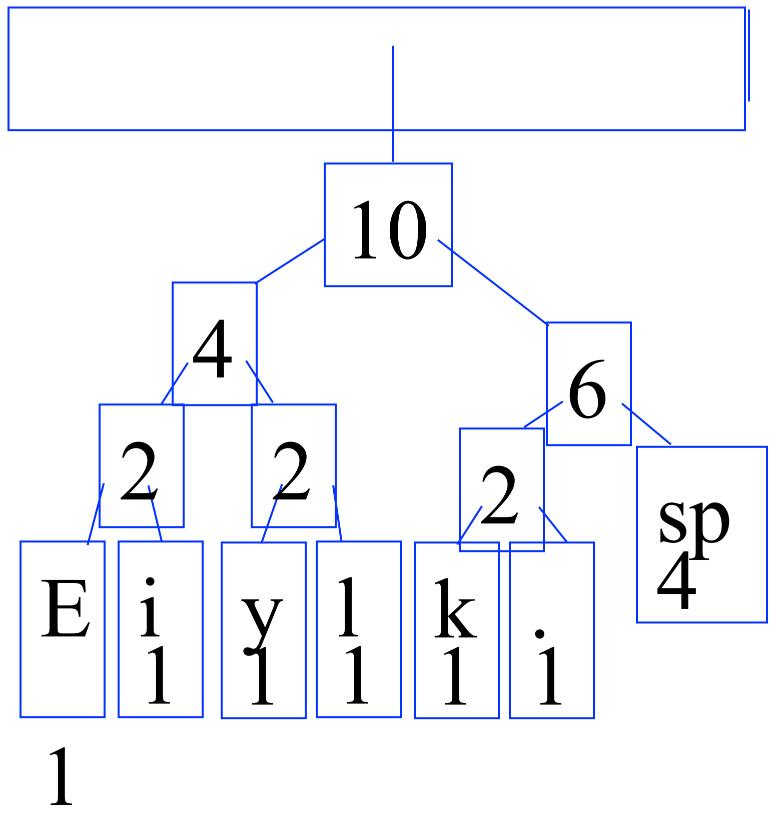
Building a Tree

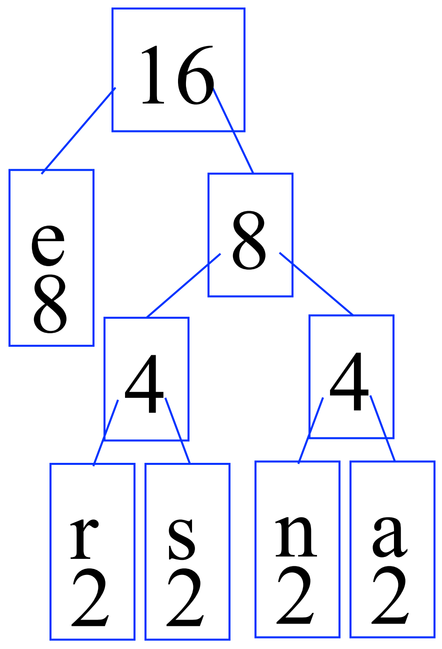


Building a Tree

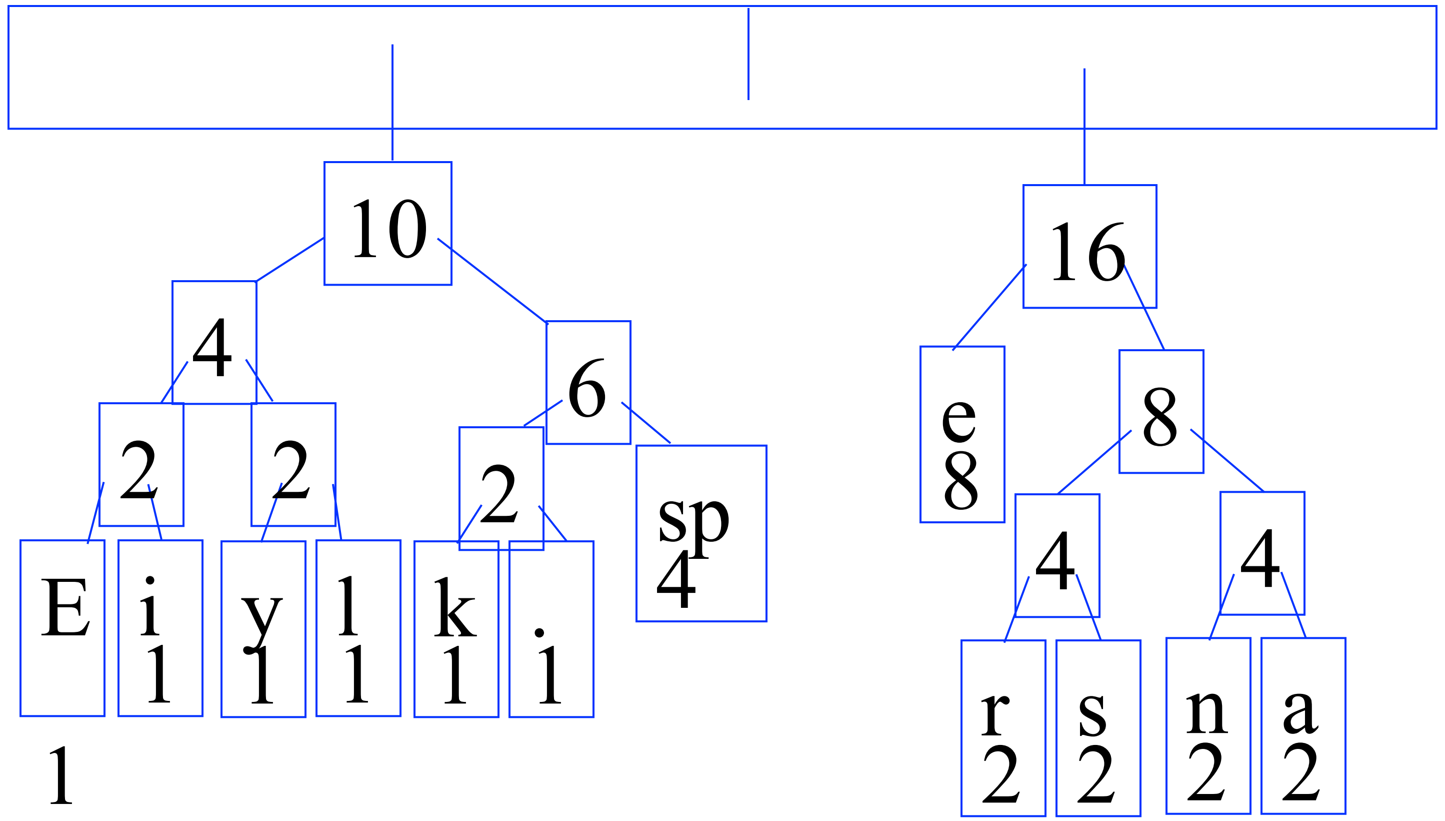


Building a Tree

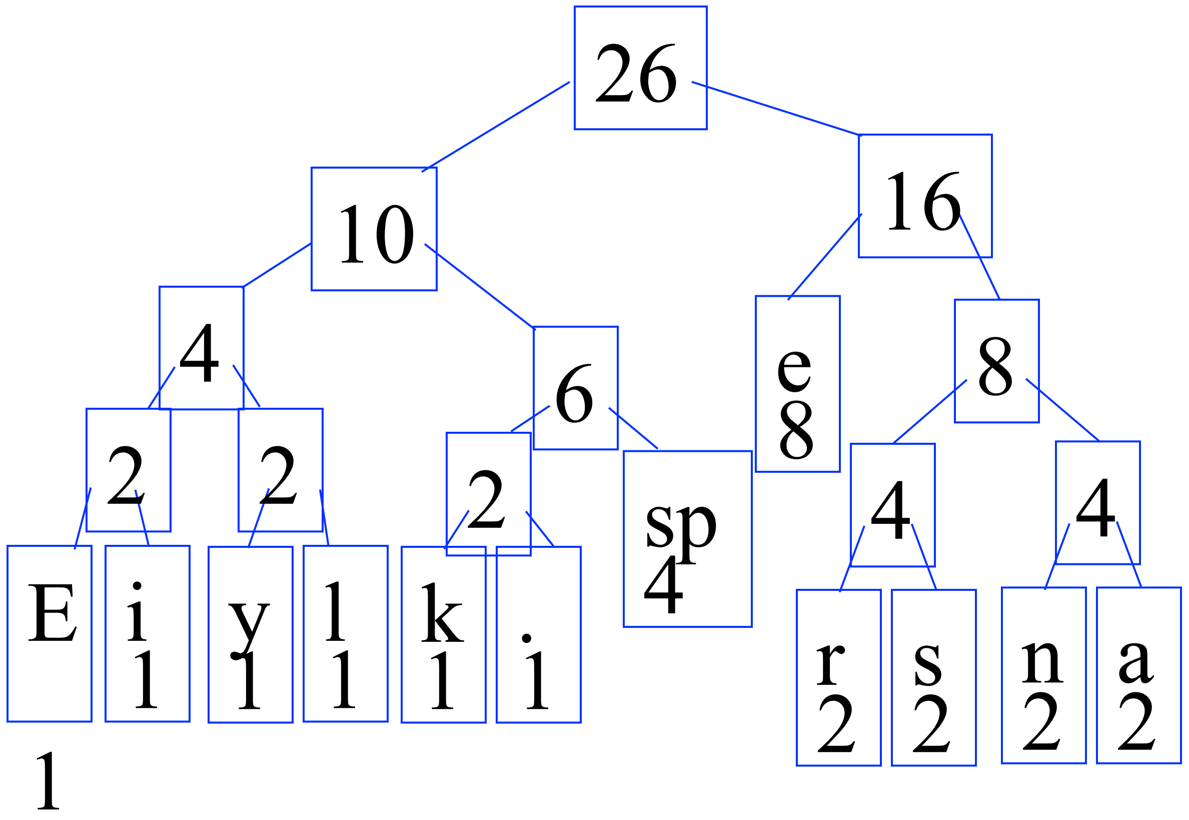




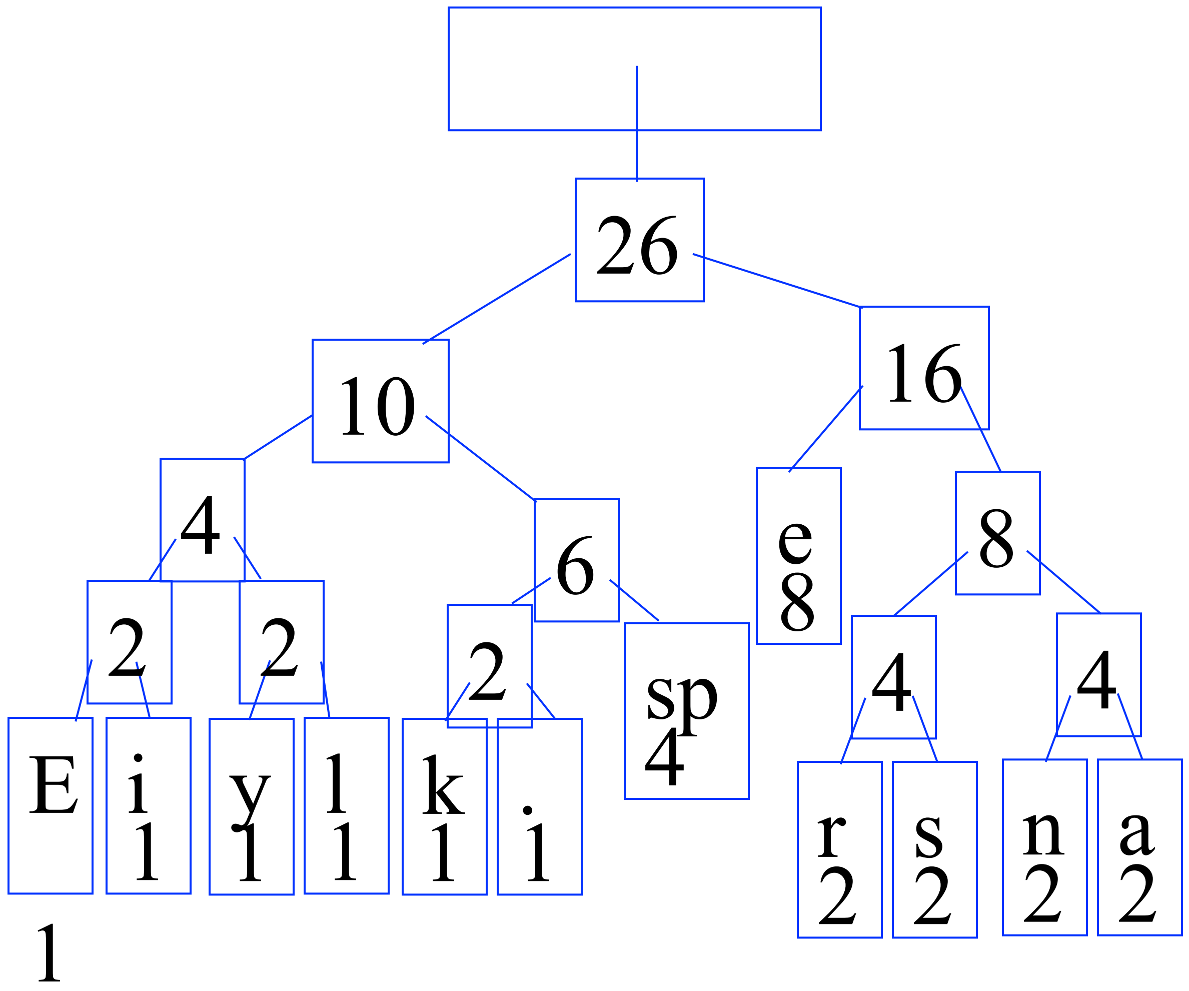
Building a Tree



Building a Tree



Building a Tree



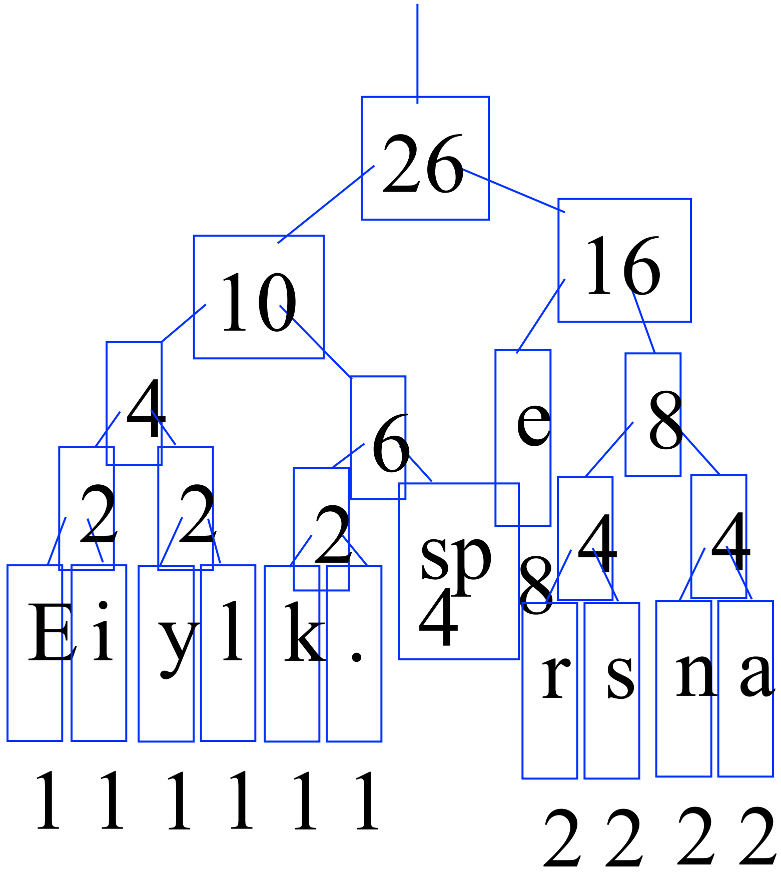
After enqueueing this node there is only one node left in priority queue.

Building a Tree

Dequeue the single node left in the queue.

This tree contains the new code words for each character.

Frequency of root node should equal number of characters in text.



Eerie eyes seen near lake. ✓ 26 characters

- Write Path Splitting /Path Halving algorithms.

- **Exercises: P316-3**