Binary Trees

Lecture 11

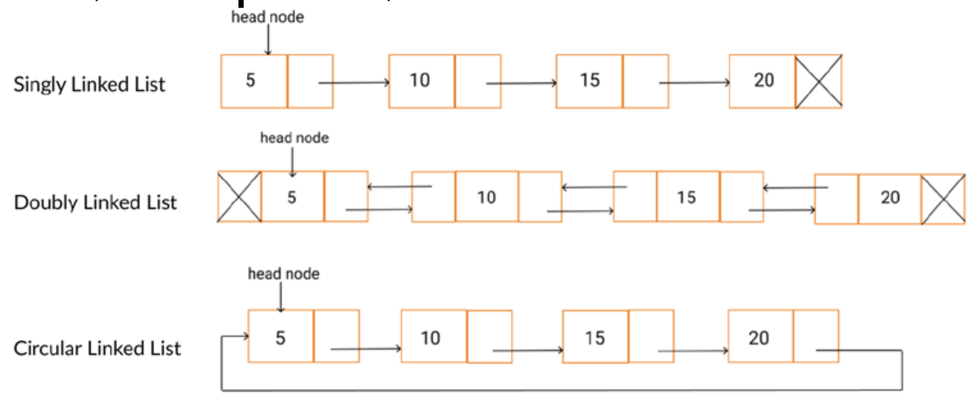
Instructor: Dr. Cong Pu, Ph.D.

cong.pu@okstate.edu

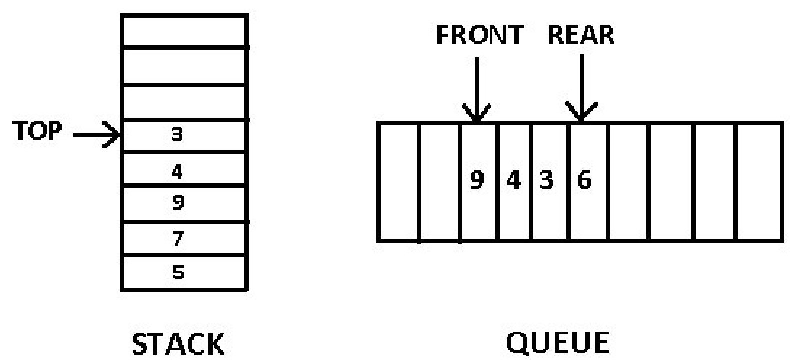
Adapted partially from Data Structures and Algorithms in Java, M.T. Goodrich, R.Tamassia and M. H. Goldwasser, Sixth Edition, Wiley; Data Structures and Algorithms in C++, Adam Drozdek, 4th Edition, Cengage Learning

Trees, Binary Trees, and Binary Search Trees

- **Limitations** of linked lists, stacks, and queues,
  - **Linked lists:**
    - linear in form and cannot reflect hierarchically organized data

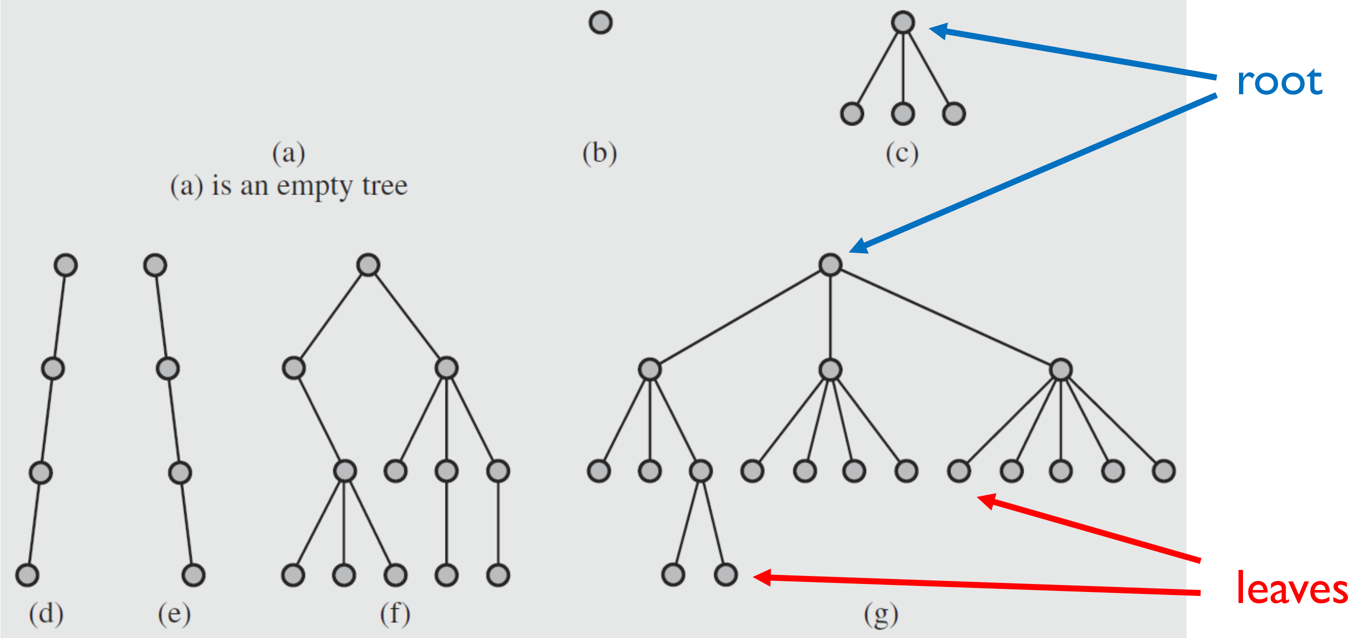


  - **Stacks and queues**
    - one-dimensional structures and have limited expressiveness



Trees, Binary Trees, and Binary Search Trees (cont.)

- A new data structure, the ***tree***,
  - two components, ***nodes*** and ***arcs*** (or ***edges***)
  - the ***root*** at the top, and "grow" down
  - the ***leaves*** of the tree (also called ***terminal nodes***)
    - at the bottom of the tree

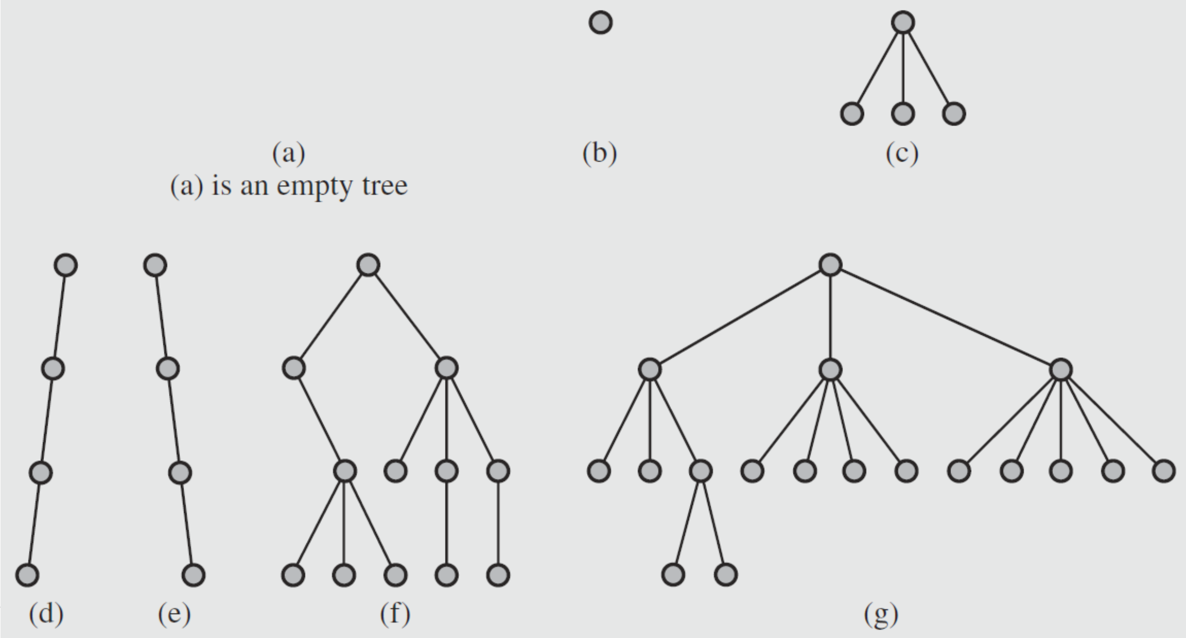


Trees, Binary Trees, and Binary Search Trees (cont.)

- Trees can be defined recursively,
  1. A tree with ***no*** nodes or edges (an *empty structure*) is an **empty tree**
  2. If we have a set $t_1 \cdots t_k$ of disjoint trees, the structure whose root has as its children the roots of $t_1 \cdots t_k$ is also a tree
  3. Only structures generated by rules 1 and 2 are trees
- Every node in the tree must be ***accessible***
  - from the *root* through a **unique sequence** of edges,
  - a ***path***
- The number of edges in the path
  - path's ***length***
- The length of the path from the root to that node **plus 1**
  - a node's ***level*** (or the number of nodes in the path)

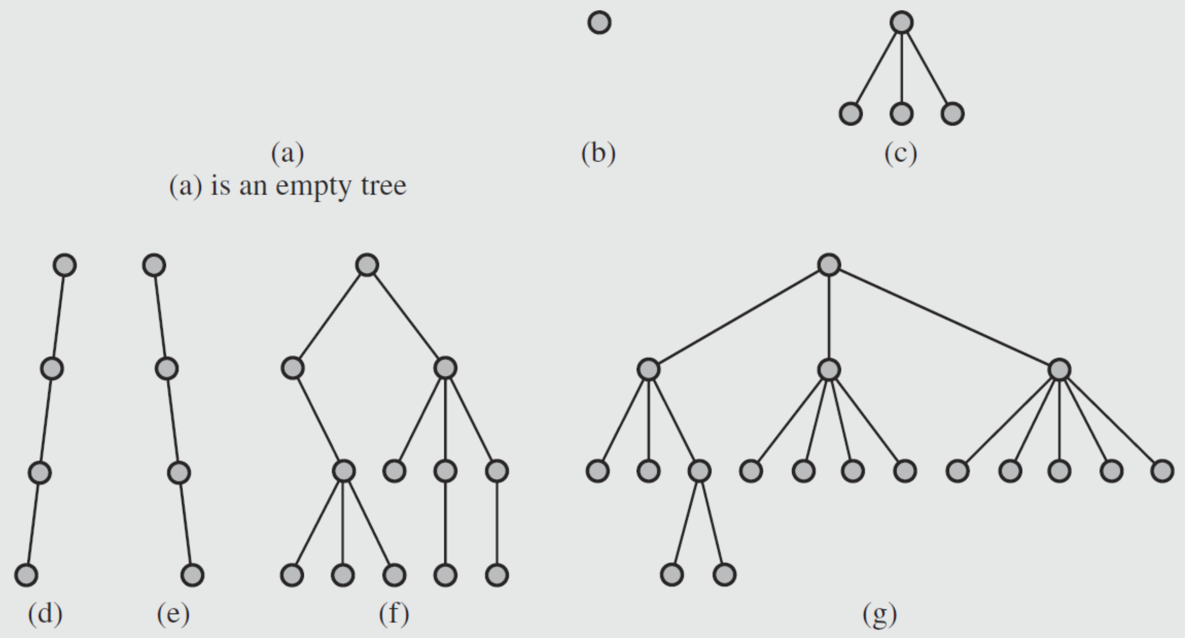
Trees, Binary Trees, and Binary Search Trees (cont.)

- The maximum level of a node in a tree: the tree's ***height***
- An empty tree: **height 0**
- A tree of **height 1**: a single node which is both the **root** and **leaf**
- The level of a node: must be between 1 and the tree's height



Trees, Binary Trees, and Binary Search Trees (cont.)

- The number of children of a given node?
  - can be **arbitrary**
- Using tree to represent hierarchy
- Using trees to improve the process of **searching** for elements??



Trees, Binary Trees, and Binary Search Trees (cont.)

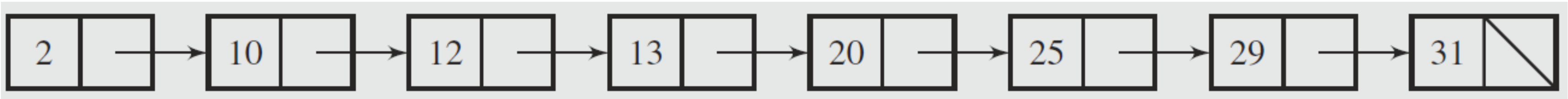
- In order to find a particular element in a **list** of ***n*** elements,
  - examine all nodes
    - search from beginning to end
      - until the element is found
      - or reach the end of list
  - if the list is **ordered?**
    - ***Same idea***: search from beginning to end
  - E.g., 10,000 nodes and the last node is the target
    - all 9,999 of its predecessors have to be traversed

extremely inconvenient!

- If the elements of a list are stored in an ***orderly* tree**??
  - the number of elements that must be looked at can be reduced
    - even when the target is the one farthest way

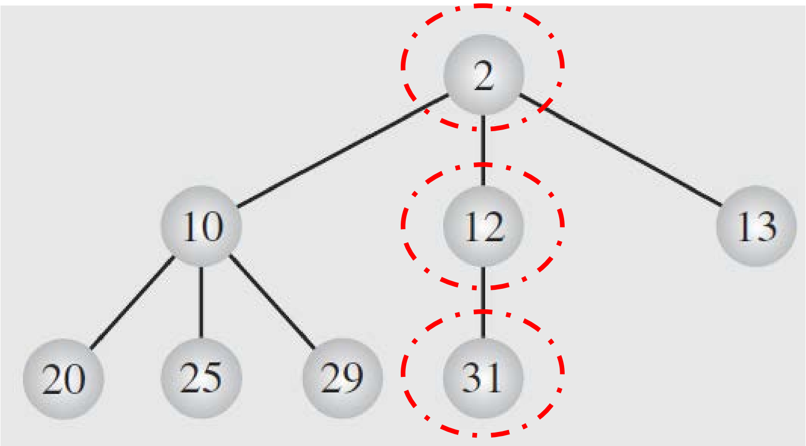
Trees, Binary Trees, and Binary Search Trees (cont.)

- Linked list: search 31 eight tests needed
  - no consideration of searching incorporated into design



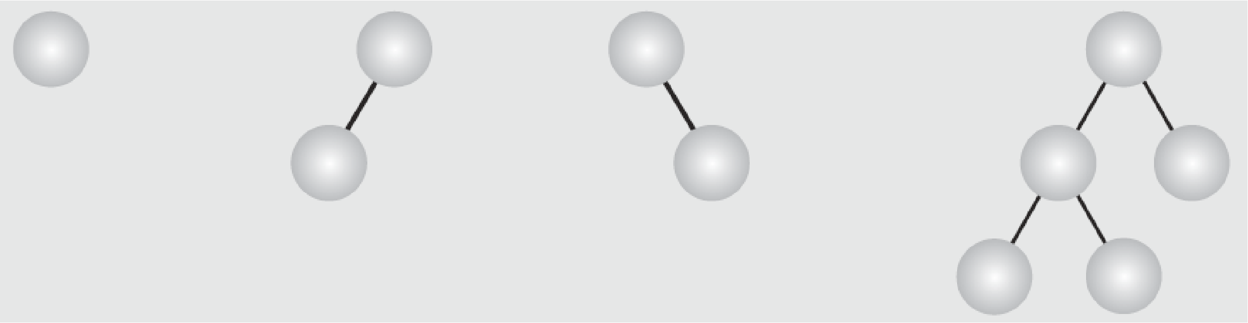
- Tree: search 31
  - considerable savings in searching if a **consistent ordering** to the nodes is applied

elements are ordered from top to bottom, from left to right.



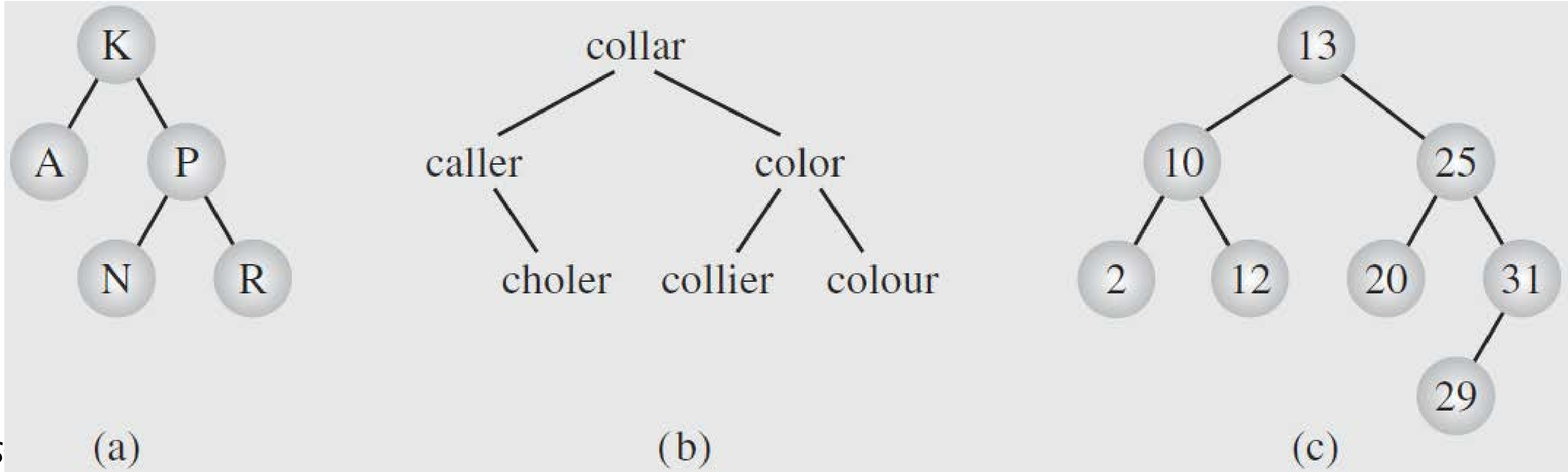
Trees, Binary Trees, and Binary Search Trees (cont.)

- A ***binary tree*** is a tree
  - each node has only **two children:** the ***left child*** and the ***right child***
  - these children can be **empty**

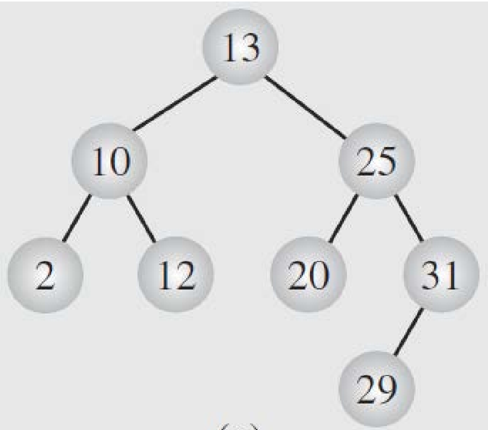


Trees, Binary Trees, and Binary Search Trees (cont.)

- In a ***binary search tree*** (or ***ordered binary tree***),
  - values stored in the **left subtree** of a given node *n* are less than the value stored in node *n*
  - values stored in the **right subtree** of a given node *n* are greater than the value stored in node *n*
  - the values stored are considered **unique**;
  - attempts to store **duplicate values** can be treated as an **error**



Implementing Binary Trees

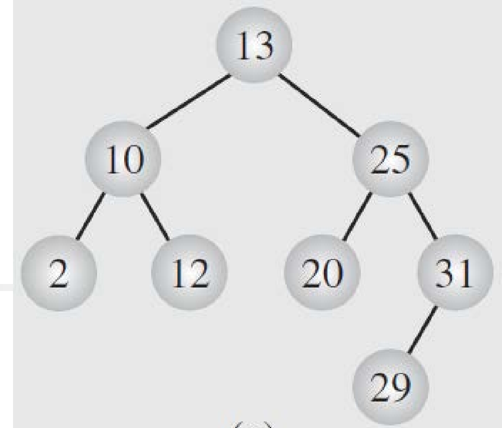


(c)

- Use *arrays* or *linked structures* to implement binary trees
- If using an *array*,
  - an information field
  - two "**pointer**" fields containing the indexes of the array locations of the **left** and **right** children
  - -1, an empty child
- The root of the tree
  - always in the first cell of the array

Index	Info	Left	Right

Implementing Binary Trees



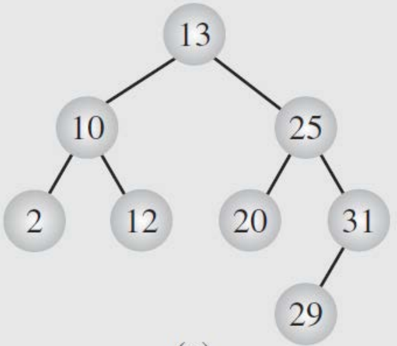
(c)

- Use *arrays* or *linked structures* to implement binary trees
- If using an *array*,
  - an information field
  - two "**pointer**" fields containing the indexes of the array locations of the **left** and **right** children
  - -1, an empty child
- The root of the tree
  - always in the first cell of the array

Index	Info	Left	Right
0	13	4	2
1	31	6	–1
2	25	7	1
3	12	–1	–1
4	10	5	3
5	2	–1	–1
6	29	–1	–1
7	20	–1	–1

Implementing Binary Trees (cont.)

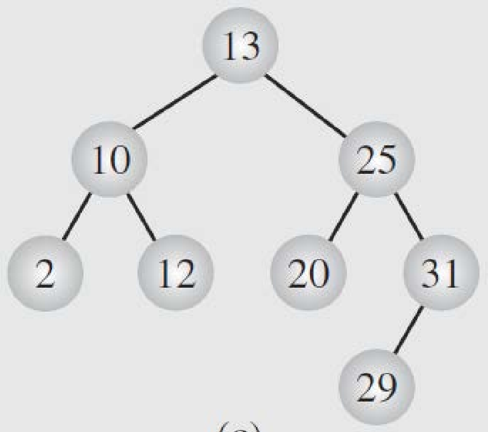
- Drawbacks of **binary tree arrays**
  - need to keep track of the locations of each node,
  - location of children must be known to insert new node
  - **deletion operation**??
    - requiring tag to mark empty cells,
    - moving elements around, or
    - requiring updating values
- Use a **linked implementation**
  - an information data member
  - **two pointer** data members



(c)

Index	Info	Left	Right
0	13	4	2
1	31	6	–1
2	25	7	1
3	12	–1	–1
4	10	5	3
5	2	–1	–1
6	29	–1	–1
7	20	–1	–1

Searching a Binary Search Tree



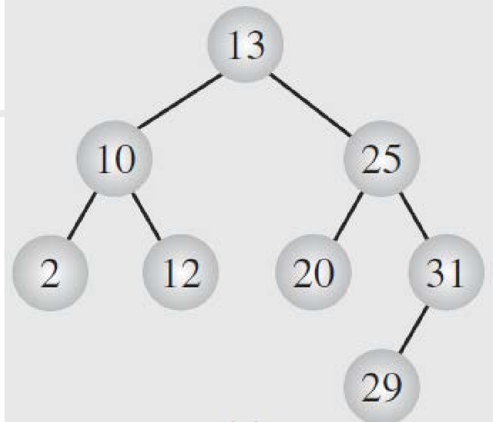
(c)

- Locating a specific value in a binary tree:
  - compare the value to the target value; if match, the search is done
  - If the target is **smaller**, branch to the **left subtree**
  - If the target is **larger**, branch to the **right subtree**
  - If at any point we cannot proceed further,
    - search has failed and the target isn't in the tree

```
template<class T>
T* BST<T>::search(BSTNode<T>* p, const T& el) const {
    while (p != 0)
        if (el == p->el)
            return &p->el;
        else if (el < p->el)
            p = p->left;
        else p = p->right;
    return 0;
}
```

- p: tree
- el: target
- while (p != 0): empty tree?
- if (el == p->el): compare target with node value
- else if (el < p->el): target less than node value; go to left branch; search
- else: target larger than node value; go to right branch; search

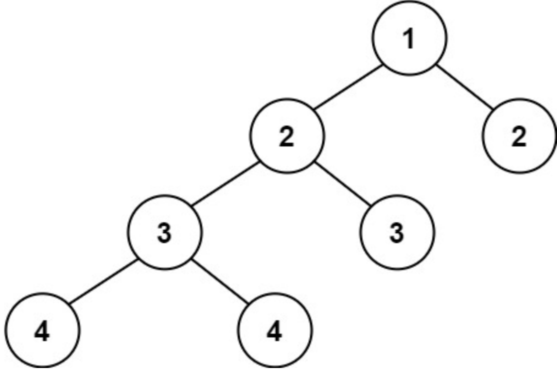
Searching a Binary Search Tree (cont.)



(c)

- Find the value 31??
  - only three comparisons
- Finding (or not finding) the values 26 – 30
  - the maximum of four comparisons;
- Allowing **duplicates** requires additional searches:
  - If there is a duplicate,
    - either locate the first occurrence and ignore the others, or
  - locate each duplicate,
    - search until no path contains another instance of the value
- This search will always terminate at a **leaf node**

Searching a Binary Search Tree (cont.)



balanced binary tree

- The number of comparisons performed during the search
  - determine the **complexity** of the search
  - depend on the number of nodes encountered on the path from the root to the target node
- The complexity??
  - the length of the path plus 1
  - influenced by the shape of the tree and location of the target
- Searching in a binary tree
  - quite efficient, even if it isn't balanced (balanced binary tree)
  - balanced binary tree: a binary tree in which the left and right subtrees of every node differ in height by no more than 1