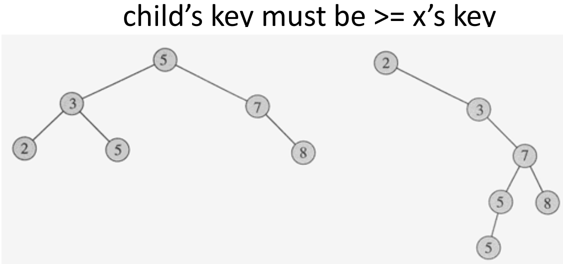
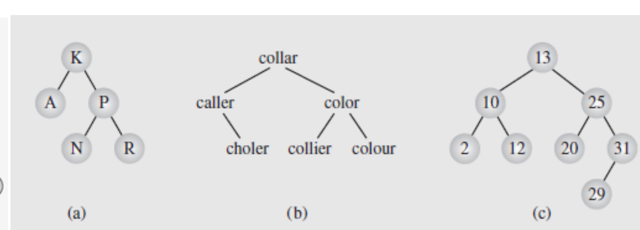
Binary Search Trees

What is a Binary Search Tree?

- A binary tree where each node is an object
  - Each node has a key value, left child, and right child (might be empty)
- Each node satisfies the binary search tree property
  - Let x be a node in the BST. The left child's key must be <= x's key. The right child's key must be >= x's key





(a) (b) (c)

Implementing Binary Trees

- We can use arrays or linked structures to implement binary trees
- If using an array, each element stores a structure that has an information field and two "pointer" fields containing the indexes of the array locations of the left and right children
- The root of the tree is always in the first cell of the array, and a value of -1 indicates an empty child

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

3

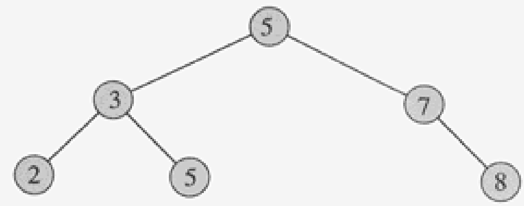
Implementing Binary Trees (continued)

- Implementing binary tree arrays does have drawbacks
  - We need to keep track of the locations of each node, and these have to be located sequentially
  - Deletions are also awkward, requiring tags to mark empty cells, or moving elements around, requiring updating values
- Consequently, while arrays are convenient, we'll usually use a linked implementation
- In a linked implementation, the node is defined by a class, and consists of an information data member and two pointer data members
- The node is manipulated by methods defined in another class that represents the tree

4

Searching a BST

```
TREE-SEARCH(x, k)
if x = NIL or k = key[x]
    then return x
if k < key[x]
    then return TREE-SEARCH(left[x], k)
    else return TREE-SEARCH(right[x], k)
```



Runs in O(h) time but this could be O(n) in the worst case!
O(lgn) if the tree is balanced!

Finding min and max?

Tree Traversal

- ***Tree traversal*** is the process of visiting each node in a tree data structure exactly one time
- This definition only specifies that each node is visited, but does not indicate the order of the process
- Hence, there are numerous possible traversals; in a tree of *n* nodes there are *n*! traversals
- Two especially useful traversals are ***depth-first traversals*** and ***breadth-first traversals***

6

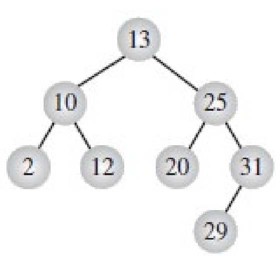
Tree Traversal (continued)

- Breadth-First Traversal
  - Breadth-first traversal proceeds level-by-level from top-down generally visiting each level's nodes left-to-right
  - This can be easily implemented using a queue
  - If we consider a top-down, left-to-right breadth-first traversal, we start by placing the root node in the queue
  - We then remove the node at the front of the queue, and after visiting it, we place its children (if any) in the queue
  - This is repeated until the queue is empty

Breadth-First Traversal (continued)

- The following diagram shows a traversal of the tree from Figure 6.6c, using the queue-based breadth-first traversal

Tree



Queue				Output
13				
10	25			13
25	2	12		13, 10
2	12	20	31	13, 10, 25
12	20	31		13, 10, 25, 2
20	31			13, 10, 25, 2, 12
31				13, 10, 25, 2, 12, 20
29				13, 10, 25, 2, 12, 20, 31
				13, 10, 25, 2, 12, 20, 31, 29

Tree Traversal (continued)

- Breadth-First Traversal (continued)
  - An implementation of this is shown in Figure 6.10

```
template<class T>
void BST<T>::breadthFirst() {
    Queue<BSTNode<T>*> queue;
    BSTNode<T> *p = root;
    if (p != 0) {
        queue.enqueue(p);
        while (!queue.empty()) {
            p = queue.dequeue();
            visit(p);
            if (p->left != 0)
                 queue.enqueue(p->left);
            if (p->right != 0)
                 queue.enqueue(p->right);
        }
    }
}
```


Depth-First Traversal

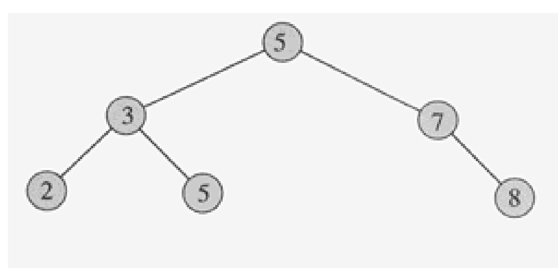
- Depth-first traversal proceeds by following left- (or right-) hand branches as far as possible
- The algorithm then backtracks to the most recent fork and takes the right- (or left-) hand branch to the next node
- It then follows branches to the left (or right) again as far as possible
- This process continues until all nodes have been visited
- While this process is straightforward, it doesn’t indicate at what point the nodes are visited; there are variations that can be used
- We are interested in three activities: traversing to the left, traversing to the right, and visiting a node
  - These activities are labeled L, R, and V, for ease of representation

Depth-First Traversal (continued)

- Based on earlier discussions, we want to perform the traversal in an orderly manner, so there are six possible arrangements:
  - VLR, VRL, LVR, LRV, RVL, and RLV
- Generally, we follow the convention of traversing from left to right, which narrows this down to three traversals:
  - VLR – known as ***preorder traversal***
  - LVR – known as ***inorder traversal***
  - LRV – known as ***postorder traversal***
- These can be implemented with a small amount of code!

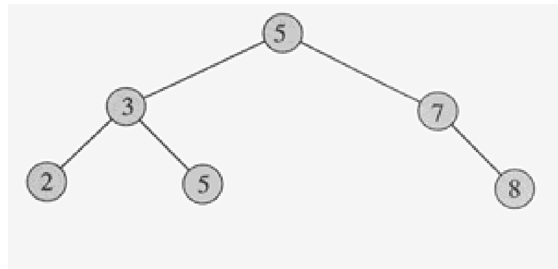
Depth First Search Implementations

```
template<class T>
void BST<T>::inorder(BSTNode<T> *p) {
    if (p != 0) {
        inorder(p->left);
        visit(p);
        inorder(p->right);
    }
}
```



Depth First Search Implementations

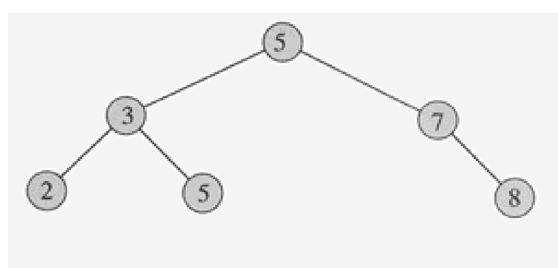
```
template<class T>
void BST<T>::preorder(BSTNode<T> *p) {
    if (p != 0) {
        visit(p);
        preorder(p->left);
        preorder(p->right);
    }
}
```



13

Depth First Search Implementations

```
template<class T>
void BST<T>::postorder(BSTNode<T>* p) {
    if (p != 0) {
        postorder(p->left);
        postorder(p->right);
        visit(p);
    }
}
```

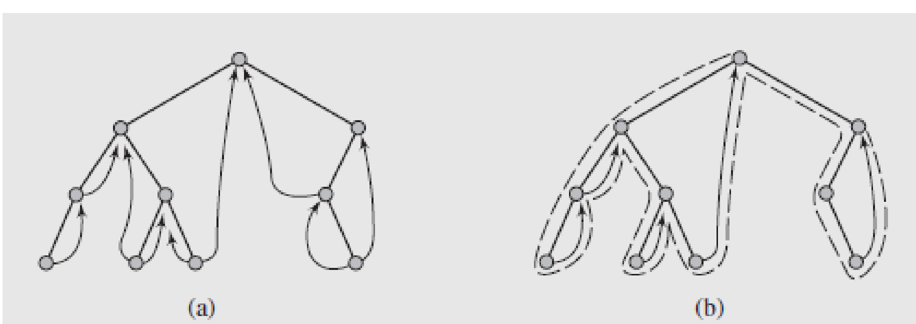


14

Depth-First Traversal (continued)

- While the code is simple, the power lies in the recursion supported by the run-time stack, which can place a heavy burden on the system
- A non-recursive implementation of the traversal algorithms is possible but we'd generally have to manage our own stack
- It is also possible to incorporate the "stack" into the design of the tree itself
  - Done using **threads**, pointers to the predecessor and successor of a node based on an inorder traversal
  - Trees with threads are called **threaded trees**

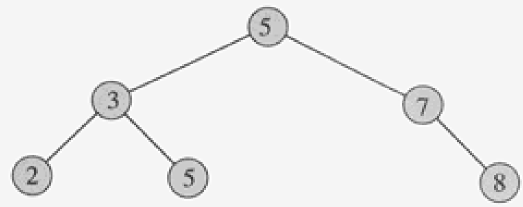
Stackless Depth-First Traversal: Threaded Trees (continued)



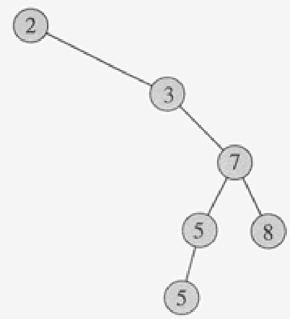
(a) A threaded tree and (b) an inorder traversal's path in a threaded tree with right successors only

Successor

- Finding the node with the next largest (or equal) value




```
TREE-SUCCESSOR(x)
if right[x] ≠ NIL
   then return TREE-MINIMUM(right[x])
y ← p[x]
while y ≠ NIL and x = right[y]
     do x ← y
        y ← p[y]
return y
```

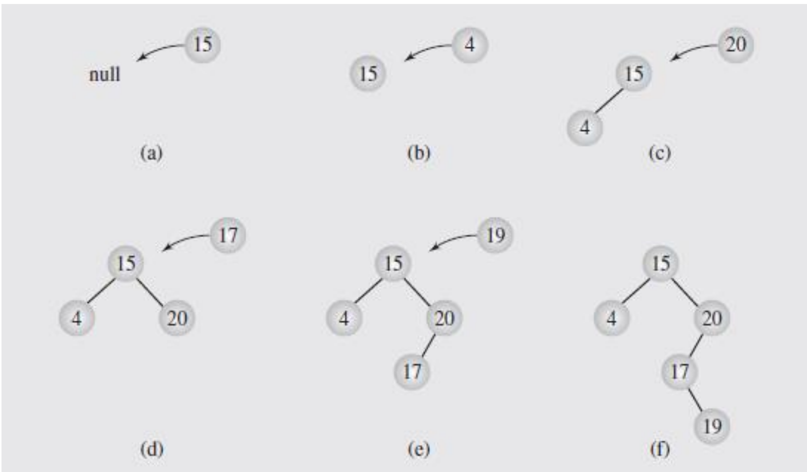


O(h) runtime

Insertion

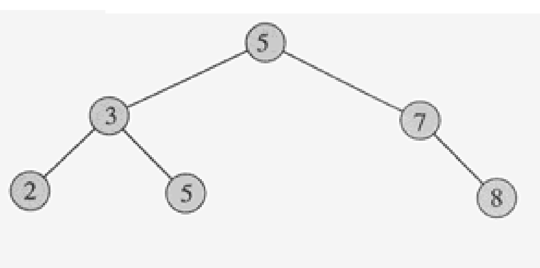
- Searching a binary tree does not modify the tree
- Traversals may temporarily modify the tree, but it is usually left in its original form when the traversal is done
- Operations like insertions, deletions, modifying values, merging trees, and balancing trees do alter the tree structure
- We'll look at how insertions are managed in binary search trees first
- In order to insert a new node in a binary tree, we have to be at a node with a vacant left or right child
- This is performed in the same way as searching:
  - Compare the value of the node to be inserted to the current node
  - If the value to be inserted is smaller, follow the left subtree; if it is larger, follow the right subtree
  - If the branch we are to follow is empty, we stop the search and insert the new node as that child

Insertion (continued)



Insertion

```
TREE-INSERT(T, z)
y ← NIL
x ← root[T]
while x ≠ NIL
      do y ← x
         if key[z] < key[x]
            then x ← left[x]
            else x ← right[x]
p[z] ← y
if y = NIL
   then root[T] ← z                    ▷ Tree T was empty
   else if key[z] < key[y]
           then left[y] ← z
           else right[y] ← z
```



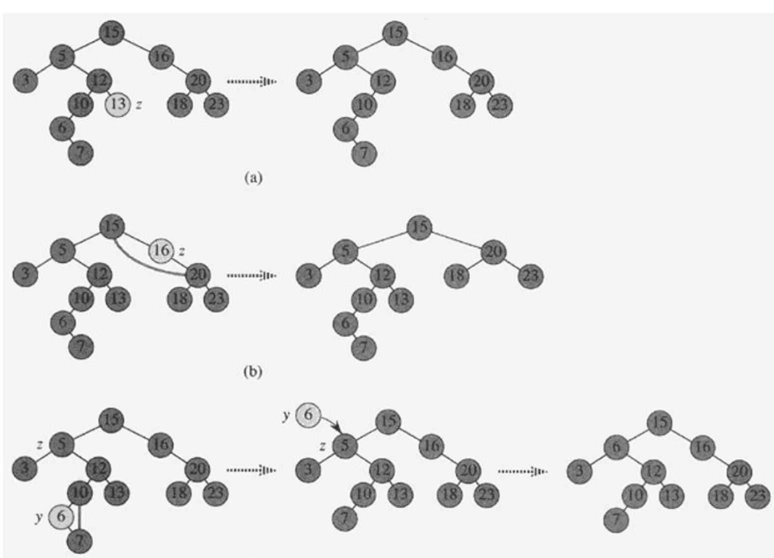
O(h) runtime

Deletion

- Deleting a node z from a BST T

1. If z has no children the simply remove it by modifying its parent to replace z with nil as its child
2. If z has just one child then we elevate that child to take z's position in the tree by modifying z's parent to replace z by z's child
3. If z has two children then: (deletion by copying)
   - Find z's successor y – which must be in z's right subtree – and have y take z's position in the tree
   - As a successor y in the right subtree, y has at most one child. Remove y using rule 2
   - The rest of z's original right subtree becomes y's right subtree and z's left subtree becomes y's left subtree

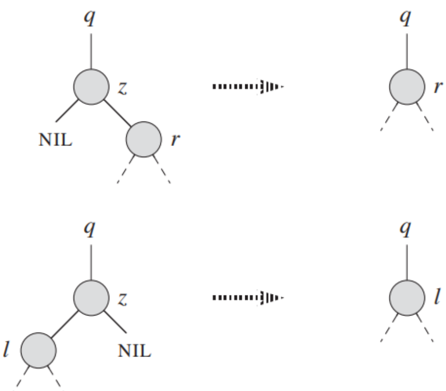
Delete Examples



Deletion

(a)

(b)



Delete node z
First two cases handled by:

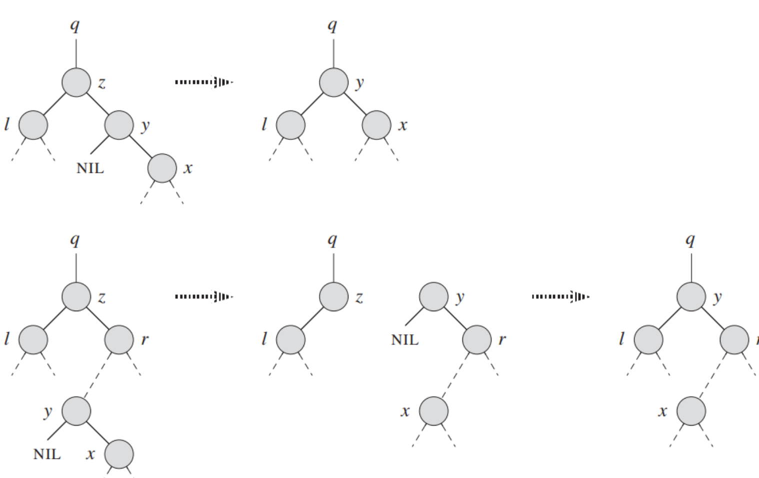
If left child is nil, transplant with right child

If right child is nil, transplant with left child

Delete with two children

(c)

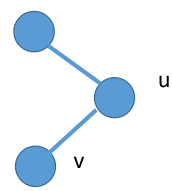
(d)



Transplant

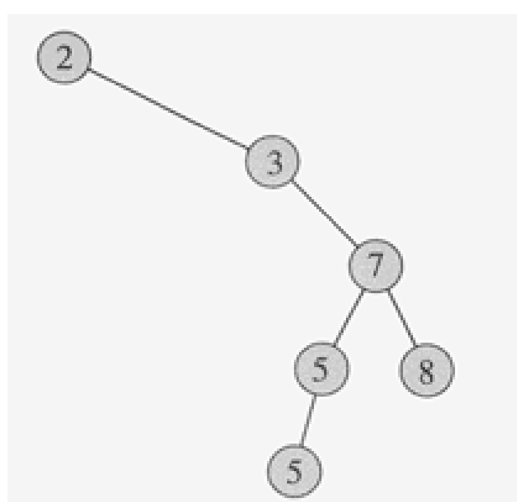
- Helper function for deletion
  - Node v is a child of node u (v could be nil)
  - Replaces u with v and updates parent of u to have v as left or right child
- Handles first two cases; partially handles third case but need to update children of v

```
TRANSPLANT(T, u, v)
if u.p == NIL
    T.root = v
elseif u == u.p.left
    u.p.left = v
else u.p.right = v
if v ≠ NIL
    v.p = u.p
```



Deletion

```
Tree-Delete(T,z)
if z.left == NIL
        Transplant(T, z, z.right)
else if z.right == NIL
        Transplant(T, z, z.left)
else
        y = Tree-Minimum(z.right)
        if y.p != z
                Transplant(T,y,y,right)
                y.right = z.right
                y.right.p = y
        Transplant(T, z, y)
        y.left = z.left
        y.left.p = y
```



BST

- Worst case?
- Best case?
- Expectation for randomly built BST?