Binary Search Tree

- Can be represented by a linked data structure in which each node is an object. Each node contains fields: *key, left, right and parent*.

- The keys are stored in such a way that satisfy the *binary-search-tree property*:

Let x be a node in a binary search tree. If *y* is a node in the left subtree of x, then *key*[*y*] ≤ *key*[x]. If *y* is a node in the right subtree of x, then *key*[x] ≤ *key*[*y*].

What are BSTs good for?

- Dictionary operations
  - Searching
  - Insertion
  - Deletion
- Other related operations
  - successor, predecessor
  - range search
  - split, merge …

Querying a Binary Search Tree

- All dictionary operations (search, insert and delete) and also "minimum", "maximum", "successor" and "predecessor" can be supported in $O(h)$ time. (h is the height of the tree)

- For a balanced binary tree, $h = \Theta(\log n)$; for an unbalanced tree that resembles a linear chain of n nodes, then $h = \Theta(n)$ in the worst case.

QUERYING A BINARY SEARCH TREE

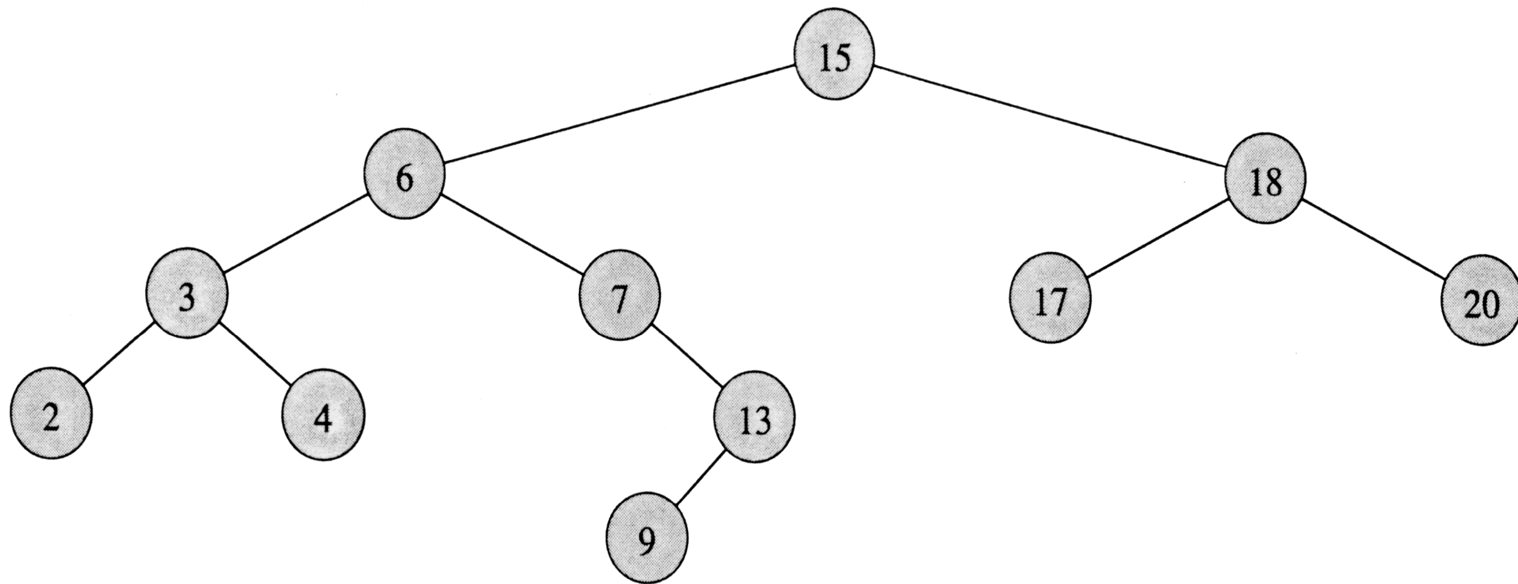


Figure 13.2 Queries on a binary search tree. To search for the key 13 in the tree, the path $15 \to 6 \to 7 \to 13$ is followed from the root. The minimum key in the tree is 2, which can be found by following *left* pointers from the root. The maximum key 20 is found by following *right* pointers from the root. The successor of the node with key 15 is the node with key 17, since it is the minimum key in the right subtree of 15. The node with key 13 has no right subtree, and thus its successor is its lowest ancestor whose left child is also an ancestor. In this case, the node with key 15 is its successor.

Tree-Search(*x*, *k*)

```
1.  if x = NIL or k = key[x]
2.      then return x
3.  if k < key[x]
4.      then return Tree-Search(left[x], k)
5.      else return Tree-Search(right[x], k)
```

Finding Min & Max

Minimum(x)

1. while *left*[x] $\neq$ *NIL*
2. do x $\leftarrow$ *left*[x]
3. return x

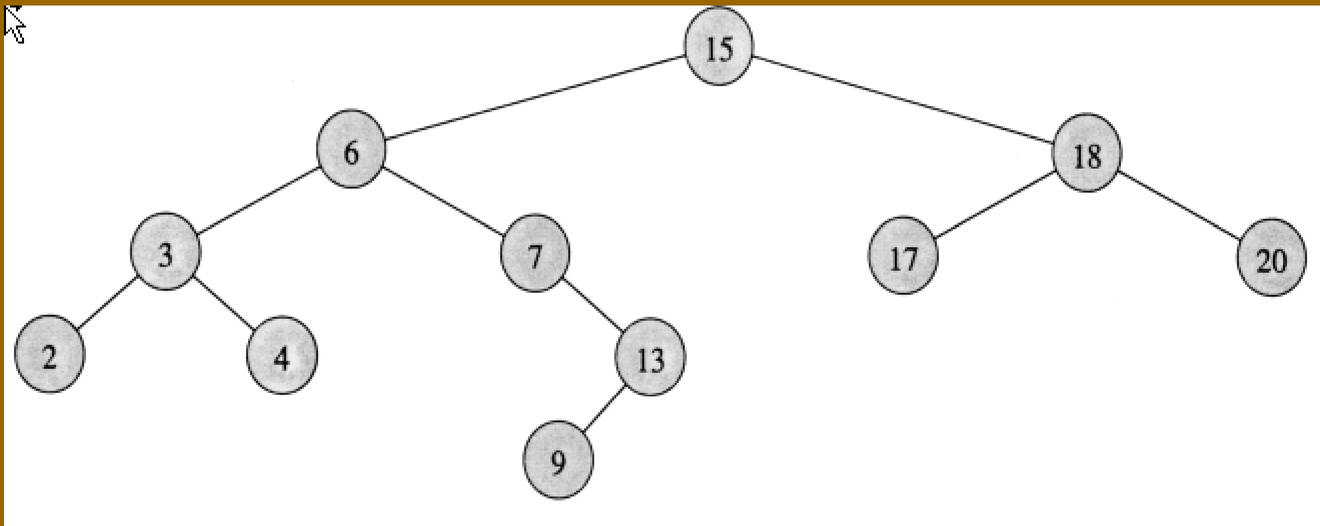
Maximum(x)

1. while *right*[x] $\neq$ *NIL*
2. do x $\leftarrow$ *right*[x]
3. return x

Q: How long do they take?

SUCCESSOR

Find the successor (the next larger value) of a node.



In the above tree, the successor of 13 is 15,

The successor pf 4 is 6 etc.

Successor (TreePtr *T*, *key x*)

// return node containing successor

```
{ if (T == null) return null;
   if (T.key > x) return Successor(T.right, x);
  if (T.key == x) return Minimum(T.right);
  y = Successor(T.left,x);
  if (y != null) return y;
  else return T;
}
```

Node Insertion

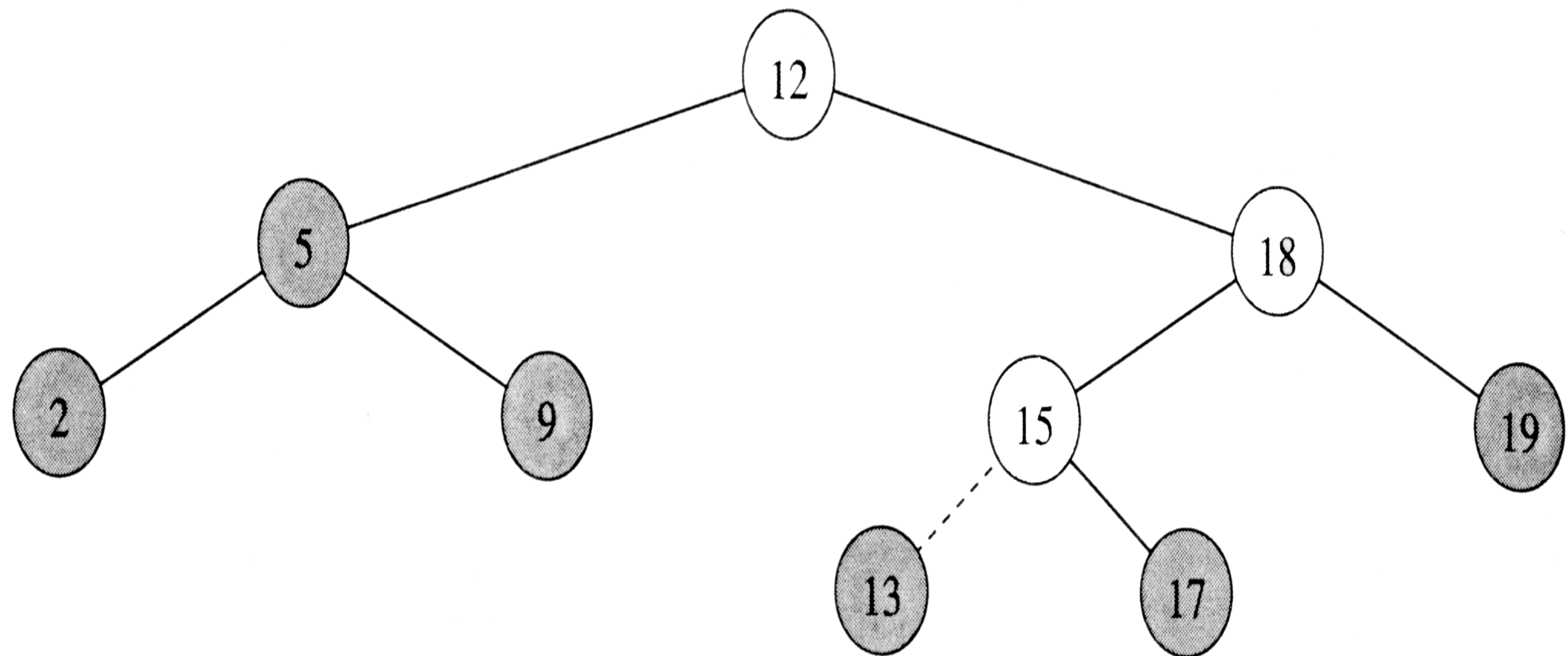


Figure 13.3 Inserting an item with key 13 into a binary search tree. Lightly shaded nodes indicate the path from the root down to the position where the item is inserted. The dashed line indicates the link in the tree that is added to insert the item.

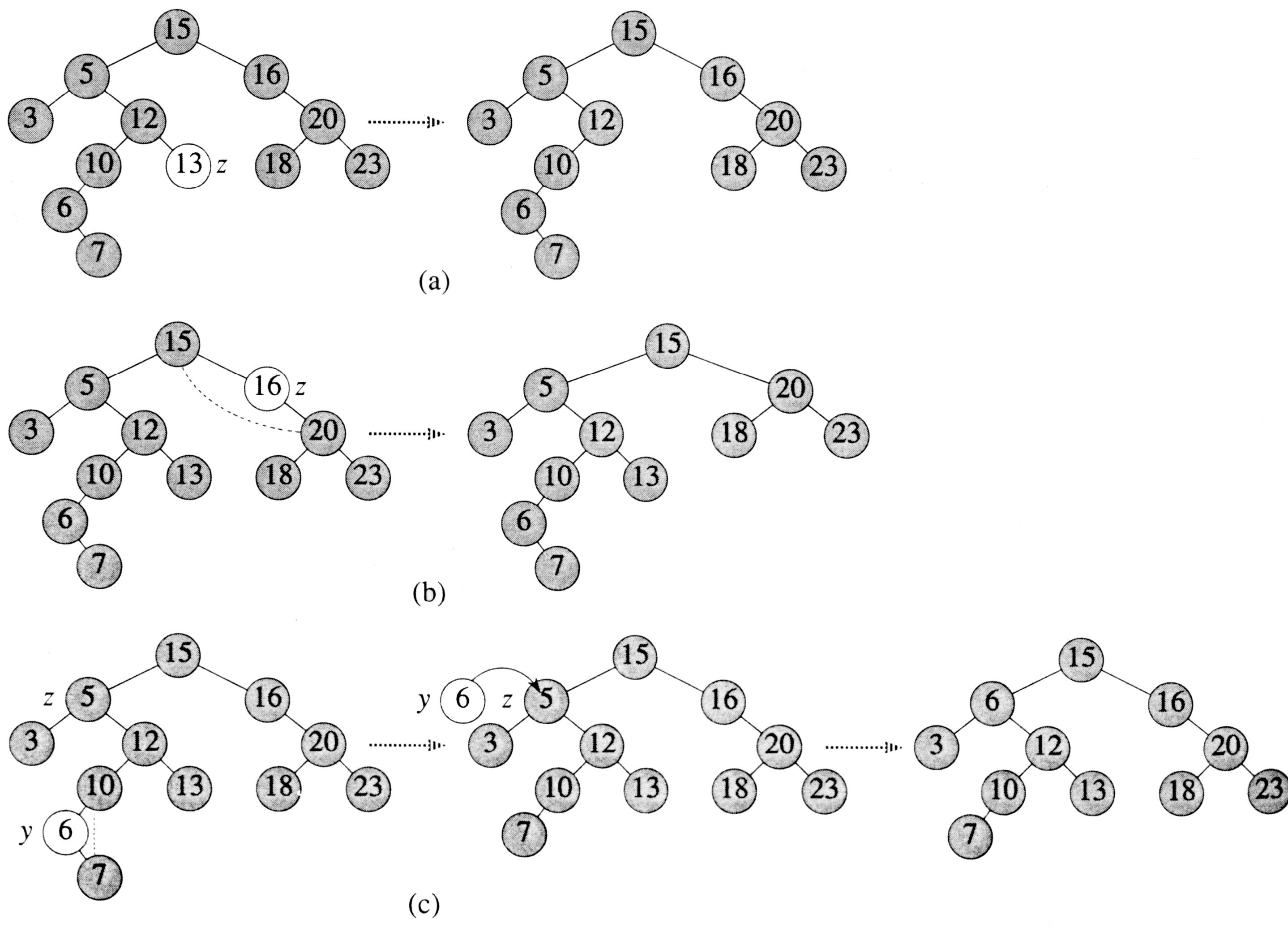
Insertion

```
TreePtr Insert (TreePtr T, key x)
{ if (T == null)
     return makeTree(x);
  if (T.key < x)
    T.right = Insert(T.right, x);
  else if (T.key > x)
    T.left = Insert(T.left, x);
  return T;
}
```

Analysis on Insertion

- Algorithm performs a sequence of key comparisons from the root along a path.
- Worst-case cost = O(h)

Node Deletion



Node Deletion

1. If the tree is NULL, there is nothing to delete.
2. If (T.key < x) continue with right subtree.
3. If (T.key > x) continue with left subtree.

// Now T.key == x.

Case 4.1. x has no children. (set tree to null.)

Case 4.2. x has one child. (set T to T.right.)

Case 4.3. X has both children.

Delete min key from the right tree and replace the node with this key.

Deletion – Recursive Implementation

```
TreePtr Delete(TreePtr T, key x)
{ if (T == null) return T;
   if (T.key < x)
    { T.left = Delete(T.left, x); return T;}
  if (T.key > x)
   {T.right = Delete(T.right, x); return T;}
 if (T.left == null && T.left == null) return null;
 if (T.left == null) return T.right;
 if (T.right == null) return T.left;
 TreePtr y = DeleteMin(T.right);
 T. key = y.key; return T;
}
```

Complexity of BST operations

Search

Successor

Insert

Delete – all take O(h) steps.

Balanced Binary Search Tree

Goal is to make $h = O(\log n)$ so that all the dictionary operations can be performed in $O(\log n)$ basic steps.

AVL tree is one of the simplest height balancing techniques. (oldest as well.)