Algorithms
Chapter 12
Binary Search Trees

Assistant Professor: Ching-Chi Lin
林清池 助理教授
chingchi.lin@gmail.com

Department of Computer Science and Engineering
National Taiwan Ocean University

Outline

- **What is a binary search tree?**
- Querying a binary search tree
- Insertion and deletion

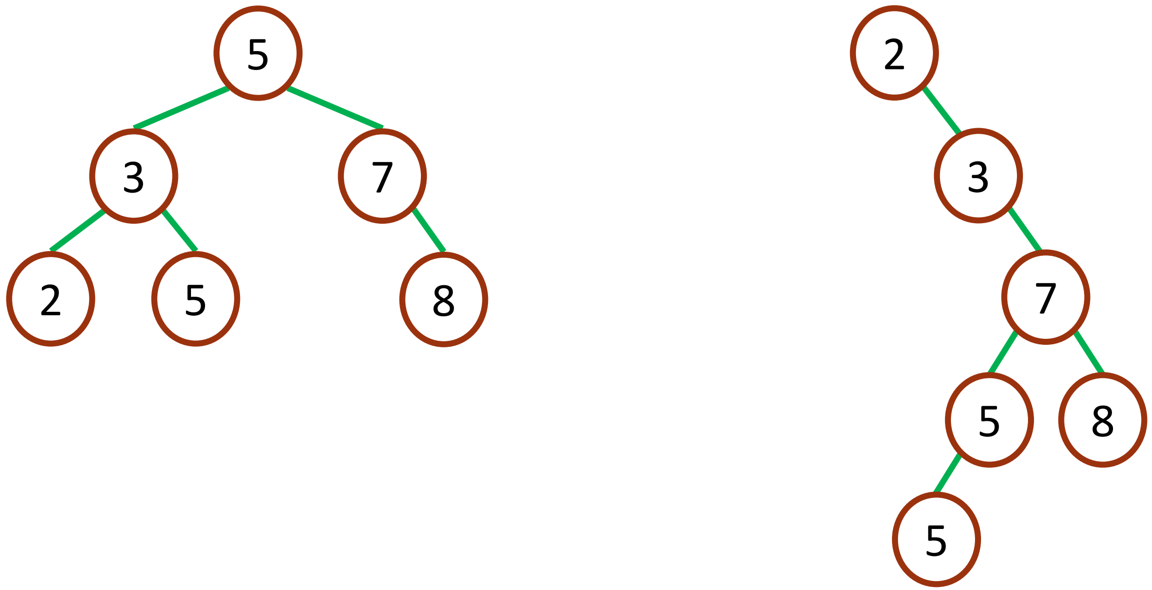
Overview

- **Search trees** are data structures that support many dynamic-set operations.
  - Dynamic-set operations includes SEARCH, MINIMUM, MAXIMUM, PREDECESSOR, SUCCESSOR, INSERT, and DELETE.
- Can be used as both a dictionary and as a priority queue.
- Basic operations take time proportional to the height of the tree, i.e., $\Theta(h)$.
  - For complete binary tree with n nodes: worst case $\Theta(\lg n)$.
  - For linear chain of n nodes: worst case $\Theta(n)$.
- Different types of search trees include **binary search trees**, **red-black trees** (Chapter 13), and **B-trees** (Chapter 18).
- We will cover binary search trees, tree walks, and operations on binary search trees.

Binary search trees

- We represent a binary tree by a linked data structure in which each node is an object.
- Each node contains the fields
  - *key* and possibly other satellite data.
  - *left*: points to left child.
  - *right*: points to right child.
  - *p*: points to parent. $p[root[T]] = \text{NIL}$.
- Stored keys must satisfy the **binary-search-tree property**.
  - If y is in left subtree of x, then $key[y] \leq key[x]$.
  - If y is in right subtree of x, then $key[y] \geq key[x]$.

Figure 12.1 Binary search trees



(a) (b)

- A binary search tree on 6 nodes with height 2.
- A less efficient binary search tree with height 4 that contains the same keys.

Inorder tree walk

- The binary-search-tree property allows us to print keys in a binary search tree in order, recursively.
- Elements are printed in monotonically increasing order.

```
INORDER-TREE-WALK(x)
    if x ≠ NIL
        then INORDER-TREE-WALK(left[x])
             print key[x]
             INORDER-TREE-WALK(right[x])
```

- The inorder tree walk prints the keys in each of the two binary search trees from Figure 12.1 in the order 2, 3, 5, 5, 7, 8.

Properties of binary search trees

- **Theorem** If x is the root of an n-node subtree, then the call INORDER-TREE-WALK(x) takes $\Theta(n)$.

  Proof:

  - $T(0) = c$, as It takes constant time on an empty subtree.
  - Left subtree has k nodes and right subtree has $n-k-1$ nodes.
  - d : the time to execute INORDER-TREE-WALK(x), exclusive of the time spent in recursive calls.
  - Prove by substitution method: $T(n)=(c+d)n+c$.
  - For $n = 0$, we have $(c+d)\cdot 0+c=c=T(0)$.
  - For $n > 0$, $T(n) = T(k) + T(n-k-1)+d$

$$
\begin{aligned}
&= ((c+d)k+c)+((c+d)(n-k-1)+c)+d \\
&= (c+d)n+c-(c+d)+c+d \\
&= (c+d)n+c.
\end{aligned}
$$

Outline

- What is a binary search tree?
- **Querying a binary search tree**
- Insertion and deletion

Operations on binary search trees

- We shall examine SEARCH, MINIMUM, MAXIMUM, SUCCESSOR, and PREDECESSOR operations.
- The running times of these operations are all $O(h)$.

```
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)
```

```
ITERATIVE-TREE-SEARCH(x, k)
    while x ≠ NIL and k ≠ key[x]
        do if k < key[x]
        then x ← left[x]
        else  x ← right[x]
    return x
```

- On most computers, the iterative version is more efficient.
- **Time:** The algorithm visiting nodes on a downward path from the root. Thus, running time is $O(h)$.

Minimum and maximum

- The binary-search-tree property guarantees that
  - the minimum key of a binary search tree is located at the leftmost node, and
  - the maximum key of a binary search tree is located at the rightmost node.
- Traverse the appropriate pointers (*left* or *right*) until NIL is reached.

TREE-MINIMUM(x)
1. **while** $left[x] \neq$ NIL
2. **do** $x \leftarrow left[x]$
3. **return** x

TREE-MAXIMUM(x)
1. **while** $right[x] \neq$ NIL
2. **do** $x \leftarrow right[x]$
3. **return** x

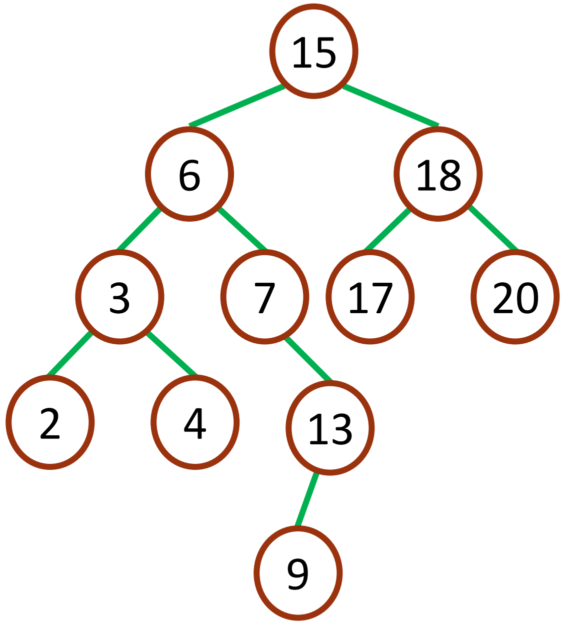
- **Time:** Both procedures visit nodes that form a downward path from the root to a leaf. Both procedures run in $O(h)$ time.

Successor and predecessor$_{1/2}$

- Assuming that all keys are distinct, the successor of a node *x* is the node *y* such that *key*[*y*] is the smallest key > *key*[*x*].
- The structure of a binary search tree allows us to determine the successor of a node without ever comparing keys.
- If *x* has the largest key in the binary search tree, then we say that *x*'s successor is NIL.
- There are two cases:
  - If node *x* has a non-empty right subtree, then *x*'s successor is the minimum in *x*'s right subtree.
  - If node *x* has an empty right subtree and *x* has a successor *y*, then *y* is the lowest ancestor of *x* whose left child is also an ancestor of *x*.

Successor and predecessor$_{2/2}$

```
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
```



- The successor of the node with key 13 is the node with key 15.
- **Time:** Since we either follow a path up the tree or follow a path down the tree. The running time is $O(h)$.
- TREE-PREDECESSOR is symmetric to TREE-SUCCESSOR.

Outline

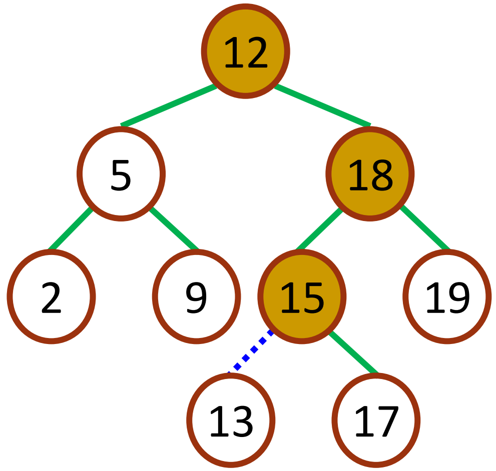
- What is a binary search tree?
- Querying a binary search tree
- **Insertion and deletion**

Insertion and deletion

- The operations of insertion and deletion cause the dynamic set represented by a binary search tree to change.
- The binary-search-tree property must hold after the change.
- Insertion is more straightforward than deletion.

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
```



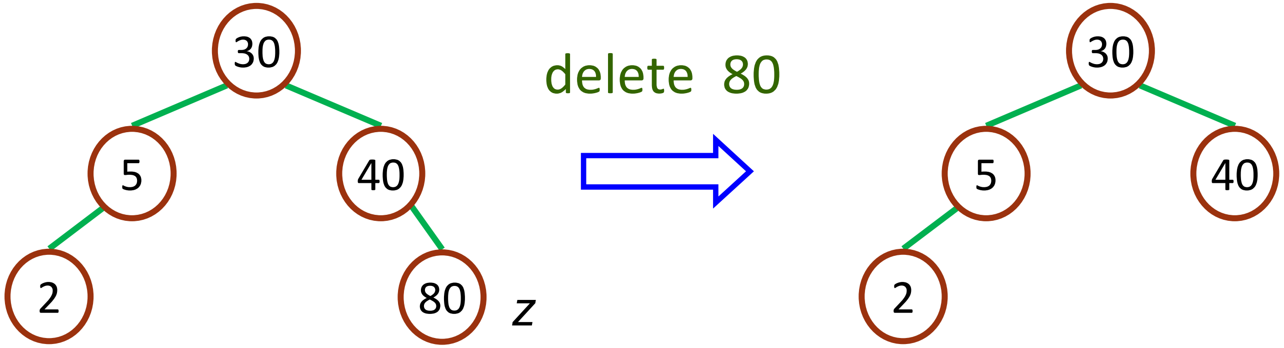
Inserting an item with key 13

- **Time:** Since we follow a path down the tree. The running time is $O(h)$.

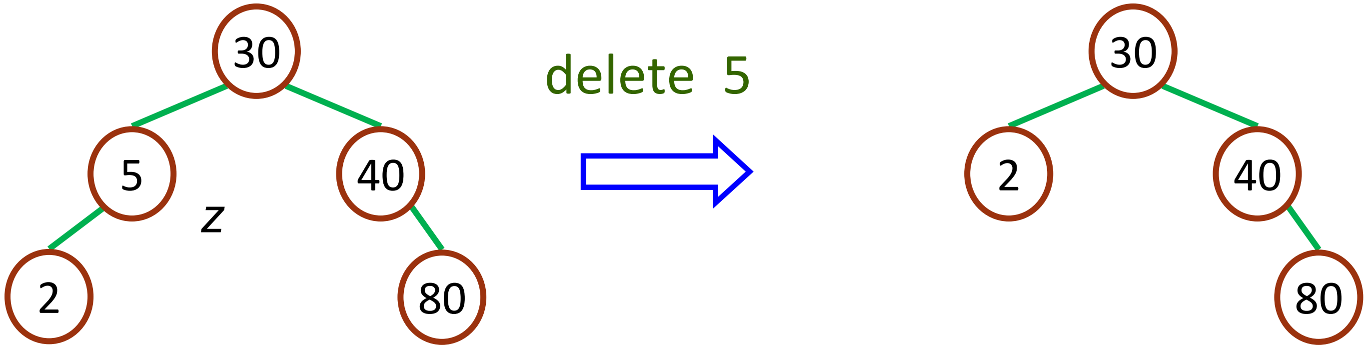
Deletion

- TREE-DELETE is broken into three cases.
- **Case 1:** *z* has no children.
  - Delete *z* by making the parent of *z* point to NIL, instead of to *z*.
- **Case 2:** *z* has one child.
  - Delete *z* by making the parent of *z* point to *z*'s child, instead of to *z*.
- **Case 3:** *z* has two children.
  - *z*'s successor *y* has either no children or one child.
    (*y* is the minimum node with no left child in *z*'s right subtree.)
  - Delete *y* from the tree (via Case 1 or 2).
  - Replace *z*'s key and satellite data with *y*'s.

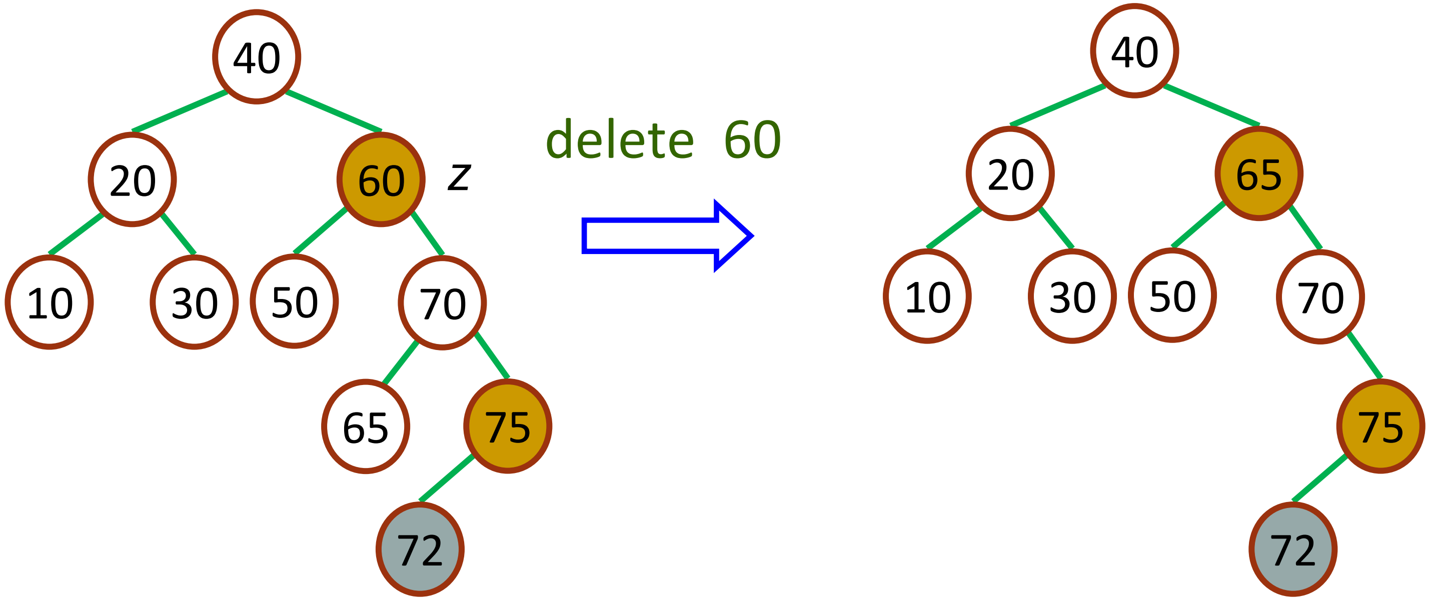
Case 1:



Case 2:



Case 3:



Deletion

```
TREE-DELETE(T, z)
    if left[z] = NIL or right[z] = NIL
       then y ← z
       else  y ← TREE-SUCCESSOR(z)
    if left[y] = NIL
       then x ← left[y]
       else  x ← right[y]
    if x ≠ NIL
       then p[x] ← p[y]
    if p[y] = NIL
       then root[T] ← x
       else if y = left[p[y]]
               then left[p[y]] ← x
               else right[p[y]] ← x
    if y ≠ z
       then key[z] ← key[y]
            copy y's satellite data into z
    return y
```

Time: $O(h)$.