WIA2005: Algorithm Design and Analysis

Semester 2, Session 2016/17

Lecture 9: Binary Search Tree

Learning objectives

- Know what is Binary Search Tree (BST)
- Know what operation can be done:
  - Search
  - Insertion
  - Deletion

Introduction

- The search tree data structure supports many dynamic-set operations, including SEARCH, MINIMUM, MAXIMUM, PREDECESSOR, SUCCESSOR, INSERT, and DELETE. Thus, we can use a search tree both as a dictionary and as a priority queue.
- Basic operations on a binary search tree take time proportional to the height of the tree.
- For a complete binary tree with n nodes, such operations run in O(lg n) worst-case time.
- If the tree is a linear chain of n nodes, however, the same operations take O(n) worst-case time.

Binary search tree

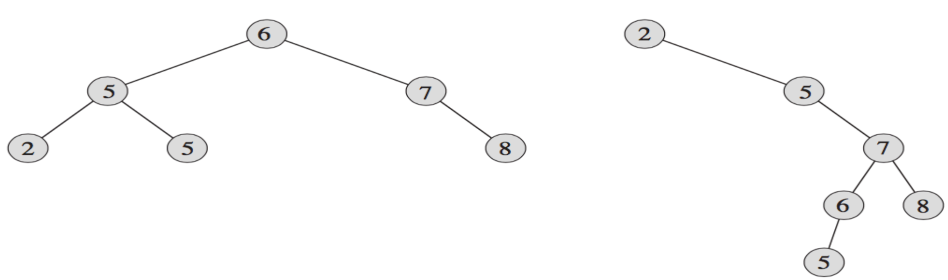
- A binary search tree is organized, as the name suggests, in a binary tree.
- We can represent such a tree by a linked data structure in which each node is an object.
- In addition to a *key* and satellite data, each node contains attributes *left*, *right*, and p that point to the nodes corresponding to its left child, its right child, and its parent, respectively.
- If a child or the parent is missing, the appropriate attribute contains the value NIL.
- The root node is the only node in the tree whose parent is NIL.

Binary Search Tree

- The keys in a binary search tree are always stored in such a way as to 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 y:*key* x:*key*. If y is a node in the right subtree of x, then y:*key* x:*key*.

- For any node x, the keys in the left subtree of x are at most x:*key*, and the keys in the right subtree of x are at least x:*key*.
- Different binary search trees can represent the same set of values.
- The worst-case running time for most search-tree operations is proportional to the height of the tree.
- **(a)** A binary search tree on 6 nodes with height 2.
- **(b)** A less efficient binary search tree with height 4 that contains the same keys.



(a) (b)

Operations

- The binary-search-tree property allows us to print out all the keys in a binary search tree in sorted order by a simple recursive algorithm, called an ***inorder tree walk***.
- This algorithm is so named because it prints the key of the root of a subtree between printing the values in its left subtree and printing those in its right subtree.
  - Similarly, a ***preorder tree walk*** prints the root before the values in either subtree, and a ***postorder tree walk*** prints the root after the values in its subtrees.

```
INORDER-TREE-WALK(x)
if x ≠ NIL
    INORDER-TREE-WALK(x.left)
    print x.key
    INORDER-TREE-WALK(x.right)
```

O(n) time

Analysis of Inorder tree walk

Theorem

If x is the root of an n-node subtree, then the call INORDER-TREE-WALK(x) takes O(n) time.

Querying a binary search tree

- We often need to search for a key stored in a binary search tree.
- Besides the SEARCH operation, binary search trees can support such queries as MINIMUM, MAXIMUM, SUCCESSOR, and PREDECESSOR.

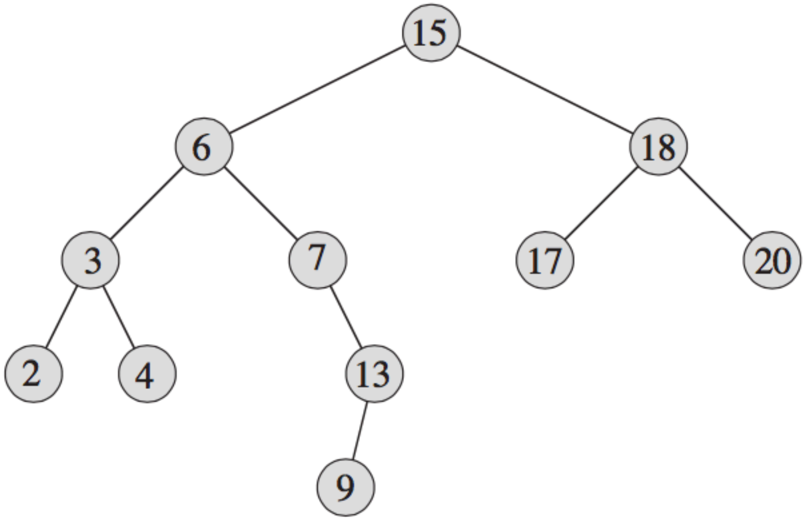
Searching

- Given a pointer to the root of the tree and a key k, TREE-SEARCH returns a pointer to a node with key k if one exists; otherwise, it returns NIL.

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

Searching - Visualised

- To search for the key 13 in the tree, we follow the path 15 -> 6 -> 7 -> 13 from the root.
- The minimum key in the tree is 2, which is 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.



Minimum and Maximum

- We can always find an element in a binary search tree whose key is a minimum by following *left* child pointers from the root until we encounter a NIL.
- The binary-search-tree property guarantees that TREE-MINIMUM is correct.
- The pseudocode for TREE-MAXIMUM is symmetric

```
TREE-MINIMUM(x)
while x.left ≠ NIL
    x = x.left
return x
```

```
TREE-MAXIMUM(x)
while x.right ≠ NIL
    x = x.right
return x
```

Both runs O(h) time on a tree of height h

Successor and predecessor

- Given a node in a binary search tree, sometimes we need to find its successor in the sorted order determined by an inorder tree walk.
- If all keys are distinct, the successor of a node x is the node with the smallest key greater than x:*key*.
- The structure of a binary search tree allows us to determine the successor of a node without ever comparing keys.
- The procedure TREE-PREDECESSOR is symmetric to TREE-SUCCESSOR.

```
TREE-SUCCESSOR(x)
1  if x.right ≠ NIL
2      return TREE-MINIMUM(x.right)
3  y = x.p
4  while y ≠ NIL and x == y.right
5      x = y
6      y = y.p
7  return y
```

Running time: O(h)

The code for TREE-SUCCESSOR can be broken into two cases.

- If the right subtree of node x is nonempty, then the successor of x is just the leftmost node in x's right subtree, which we find in line 2 by calling TREE-MINIMUM(x.*right*).
- On the other hand, if the right subtree of node x is empty and x has a successor y, then y is the lowest ancestor of x whose left child is also an ancestor of x.

Insertion and deletion

- The operations of insertion and deletion cause the dynamic set represented by a binary search tree to change.
- The data structure must be modified to reflect this change, but in such a way that the binary-search-tree property continues to hold.
- As we shall see, modifying the tree to insert a new element is relatively straight- forward, but handling deletion is somewhat more intricate.

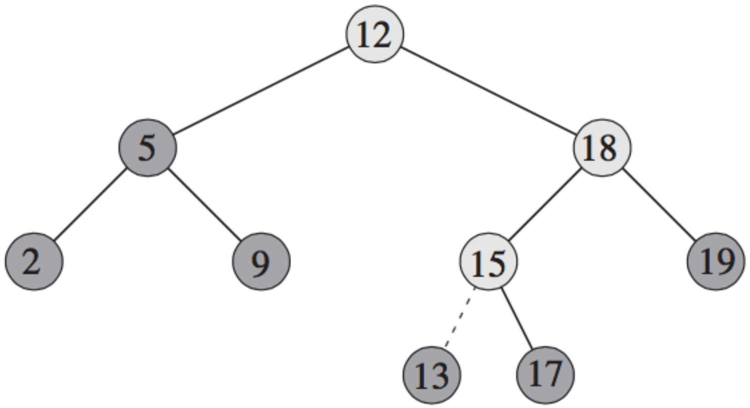
Insertion

- To insert a new value into a binary search tree T , we use the procedure TREE- INSERT.
- The procedure takes a node z for which z:*key* = v, z.*left* = NIL, and z.*right* = NIL.
- It modifies T and some of the attributes of in such a way that it inserts z into an appropriate position in the tree.

```
TREE-INSERT(T, z)
y = NIL
x = T.root
while x ≠ NIL
    y = x
    if z.key < x.key
        x = x.left
    else x = x.right
z.p = y
if y == NIL
    T.root = z        // tree T was empty
elseif z.key < y.key
    y.left = z
else y.right = z
```

Example - Insertion

- Inserting an item with key 13 into a binary search tree.
- Lightly shaded nodes indicate the simple 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.

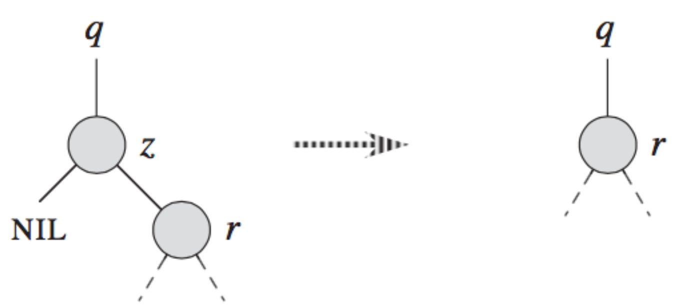


Deletion

- The overall strategy for deleting a node z from a binary search tree T has three basic cases but, as we shall see, one of the cases is a bit tricky.
  - If z has no children, then we simply remove it by modifying its parent to replace z with NIL as its child.
  - If́ 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.
  - If z has two children, then we find z's successor y—which must be in z's right subtree—and have y take z's position in the tree. The rest of z's original right subtree becomes y's new right subtree, and z's left subtree becomes y's new left subtree.
    - This case is the tricky one because, as we shall see, it matters whether y is z's right child.

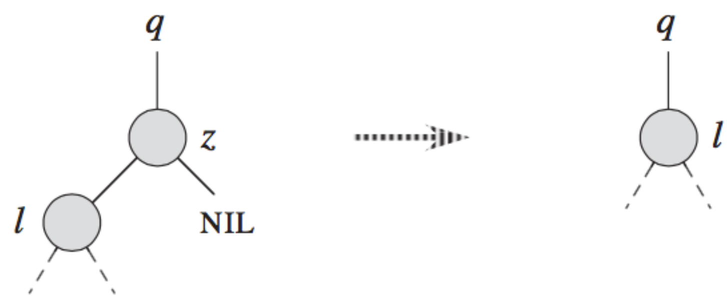
Deletion procedure (1)

- If z has no left child, then we replace z by its right child, which may or may not be NIL.
- When z's right child is NIL, this case deals with the situation in which z has no children.
- When z's right child is non-NIL, this case handles the situation in which z has just one child, which is its right child.



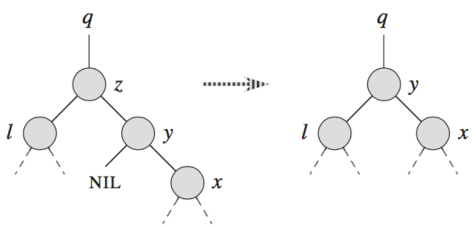
Deletion procedure (2)

- If z has just one child, which is its left child, then we replace z by its left child.

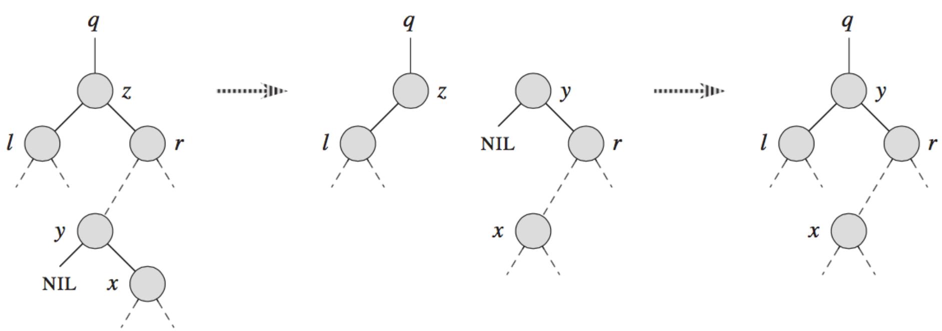


Deletion procedure (3)

- When z has both a left and a right child, we find z’s successor y, which lies in z’s right subtree and has no left child.
- We want to splice y out of its current location and have it replace z in the tree.
  - If y is z’s right child, then we replace z by y, leaving y’s right child alone.



  - Otherwise, y lies within z’s right subtree but is not΄ ’s right child. In this case, we first replace y by its own right child, and then we replace z by y.



Moving subtree

- In order to move subtrees around within the binary search tree, we define a subroutine TRANSPLANT, which replaces one subtree as a child of its parent with another subtree.
- When TRANSPLANT replaces the subtree rooted at node u with the subtree rooted at node , node u’s parent becomes node ’s parent, and u’s parent ends up having as its appropriate child.

```
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
```

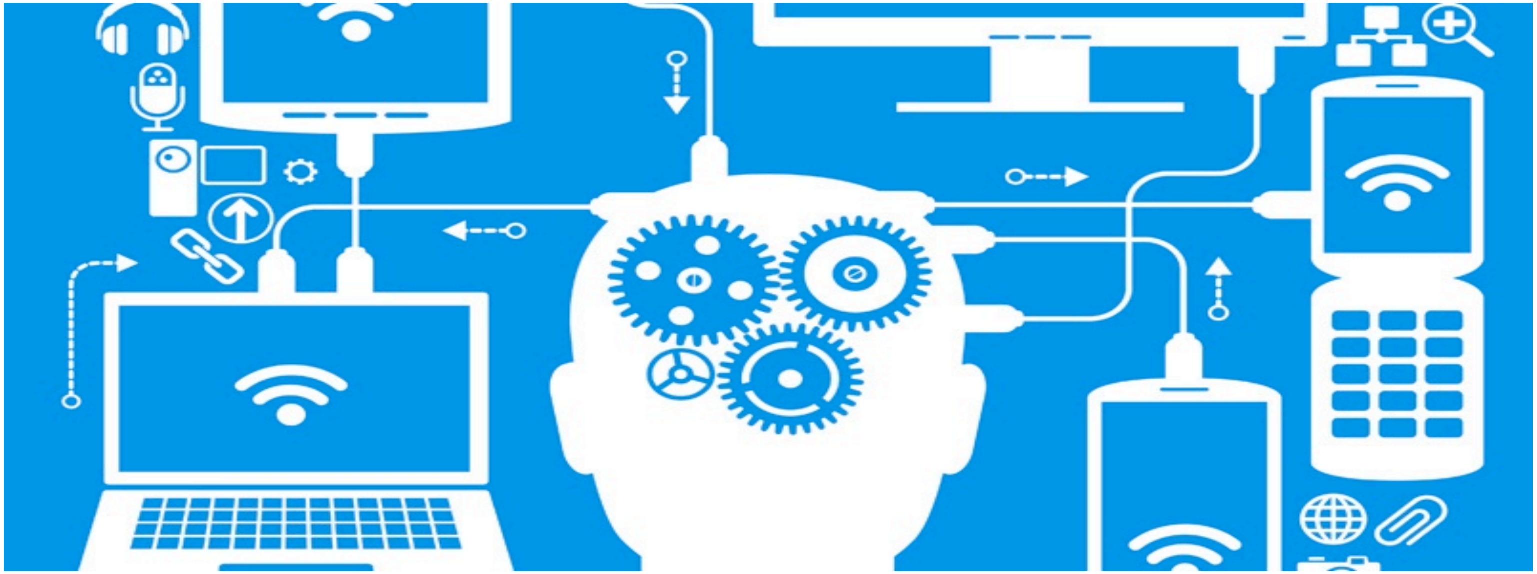
Finally deleting

- With the TRANSPLANT procedure in hand, here is the procedure that deletes node´ from binary search tree T :

```
TREE-DELETE(T, z)
if z.left == NIL
    TRANSPLANT(T, z, z.right)
elseif 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
```

O(h) time on a tree of height h

In the next lecture..



Lecture 10: Dynamic Programmimg