Design and Analysis of Algorithms

Binary Search Trees

Lecture

Instructor: Dr. G P Gupta

Dynamic Sets

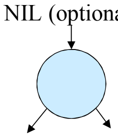
- Next few lectures will focus on data structures rather than straight algorithms
- In particular, ***structures for dynamic sets***
  - » Elements have a *key* and *satellite data*
  - » ***Dynamic sets support queries such as:***
    - ***Search(S, k), Minimum(S), Maximum(S), Successor(S, x), Predecessor(S, x)***
  - » They may also support *modifying operations* like:
    - ***Insert(S, x), Delete(S, x)***

Binary Search Trees

- View today as data structures that can support dynamic set operations.
  - » Search, Minimum, Maximum, Predecessor, Successor, Insert, and Delete.
- Can be used to build
  - » Dictionaries.
  - » Priority Queues.
- Basic operations take time proportional to the height of the tree – $O(h)$.

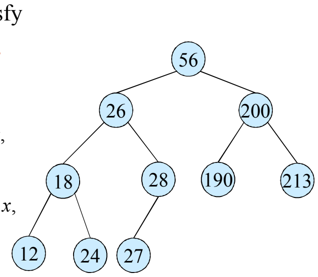
BST – Representation

- Represented by a linked data structure of nodes.
- *root*(T) points to the root of tree T.
- Each node contains fields:
  - » *key*
  - » *left* – pointer to left child: root of left subtree.
  - » *right* – pointer to right child : root of right subtree.
  - » *p* – pointer to parent. $p[root[T]]$ = NIL (optional).

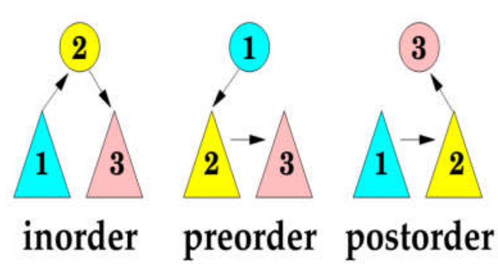


Binary Search Tree Property

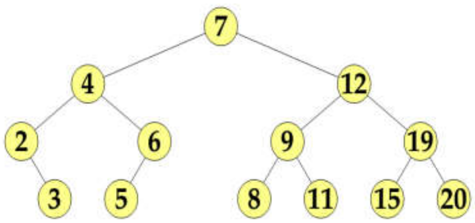
- Stored keys must satisfy the *binary search tree* property.
  - » $\forall$ y in left subtree of x, then $key[y] \le key[x]$.
  - » $\forall$ y in right subtree of x, then $key[y] \ge key[x]$.



Traversal of the Nodes in a BST



Traversal of the Nodes in a BST



Traversal of the Nodes in a BST

Inorder traversal gives: 2, 3, 4, 5, 6, 7, 8 , 9, 11, 12, 15, 19, 20.

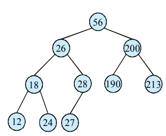
Preorder traversal gives: 7, 4, 2, 3, 6, 5, 12, 9, 8, 11, 19, 15, 20.

Postorder traversal gives: 3, 2, 5, 6, 4, 8, 11, 9, 15, 20, 19, 12, 7.

Inorder Traversal

The binary-search-tree property allows the keys of a binary search tree to be printed, in (monotonically increasing) order, recursively.

Inorder-Tree-Walk (*x*)
1. **if** $x \neq$ NIL
2. **then** Inorder-Tree-Walk(*left*[*p*])
3. print *key*[*x*]
4. Inorder-Tree-Walk(*right*[*p*])



- How long does the walk take?

Inorder Traversal

Theorem 12.1
If x is the root of an n-node subtree, then the call INORDER-TREE-WALK(x) takes $\Theta(n)$ time.

- INORDER-TREE-WALK takes a small, constant amount of time on an empty subtree. T (0)= c for some constant c > 0.
- For n > 0, suppose that INORDER-TREE-WALK is called on a node x whose left subtree has k nodes and whose right subtree has (n-k-1)nodes.
- The time to perform INORDER-TREE-WALK (x) is bounded by T(n) <= T(k) + T(n-k-1) + d for some constant d > 0
- **Use the substitution method to show that T (n) = O(n)**

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

Exercises

- For the set of {1; 4; 5; 10; 16; 17; 21} of keys, draw binary search trees of heights 2, 3, 4, 5, and 6.
- What is the difference between the binary-search-tree property and the min-heap property ?
- Can the min-heap property be used to print out the keys of an n-node tree in sorted order in O (n) time? Show how, or explain why not.

Dynamic-set operations

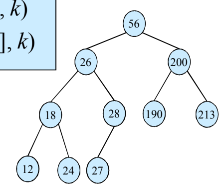
- **the dynamic-set operations :**
- SEARCH, MINIMUM, MAXIMUM, SUCCESSOR, and PREDECESSOR
- ***each one runs in O(h) time*** on a binary search tree of height h.

Dynamic-set operations

- All dynamic-set search operations can be supported in $O(h)$ time.
- $h = \Theta(lg\ n)$ for a balanced binary tree (and for an average tree built by adding nodes in random order.)
- $h = \Theta(n)$ for an unbalanced tree that resembles a linear chain of n nodes in the worst case.

Tree Search

```
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)
```

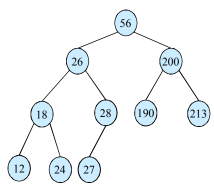


Running time: $O(h)$

Aside: tail-recursion

Iterative Tree Search

```
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
```



The iterative tree search is more efficient on most computers.
The recursive tree search is more straightforward.

Finding Min & Max

- The binary-search-tree property guarantees that:
  - » The minimum is located ***at the left-most node.***
  - » The maximum is located ***at the right-most node.***

```
Tree-Minimum(x)
while left[x] ≠ NIL
    do x ← left[x]
return x
```

```
Tree-Maximum(x)
while right[x] ≠ NIL
    do x ← right[x]
return x
```

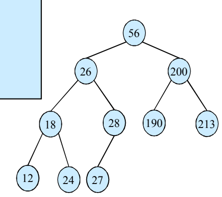
Q: How long do they take?

Predecessor and Successor

- Successor of node x is the node y such that $key[y]$ is the smallest key greater than $key[x]$.
- The successor of the largest key is NIL.
- **Search consists of two cases.**
  - » If node x has a non-empty right subtree, then x's successor is the minimum in the right subtree of x.
  - » If node x has an empty right subtree, then:
    - As long as we move to the left up the tree (move up through right children), we are visiting smaller keys.
    - x's successor y is the node that x is the predecessor of (x is the maximum in y's left subtree).
    - In other words, x's successor y, is the lowest ancestor of x whose left child is also an ancestor of x.

Pseudo-code for Successor

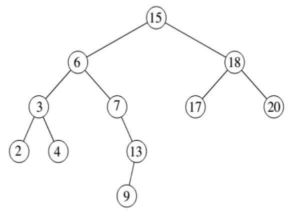
```
Tree-Successor(x)
if right[x] ≠ NIL
    then return Tree-Minimum(right[x])
y ← p[x]
while y ≠ NIL and x = right[y]
do x ← y
    y ← p[y]
return y
```



Code for ***predecessor*** is symmetric.

Running time: $O(h)$

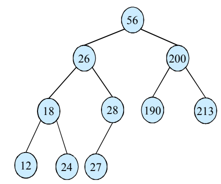
Example



- Find the successor of the node with key value 15. (Answer: Key value 17)
- Find the successor of the node with key value 6. (Answer: Key value 7)
- Find the successor of the node with key value 4. (Answer: Key value 6)
- Find the predecessor of the node with key value 6. (Answer: Key value 4)

BST Insertion – Pseudocode

- Change the dynamic set represented by a BST.
- Ensure the binary-search-tree property holds after change.
- Insertion is easier than deletion.




```
Tree-Insert(T, z)
1.  y ← NIL
2.  x ← root[T]
3.  while x ≠ NIL
4.     do y ← x
5.        if key[z] < key[x]
6.           then x ← left[x]
7.           else x ← right[x]
8.  p[z] ← y
9.  if y = NIL
10.    then root[t] ← z
11.    else if key[z] < key[y]
12.         then  left[y] ← z
13.         else right[y] ← z
```


Example

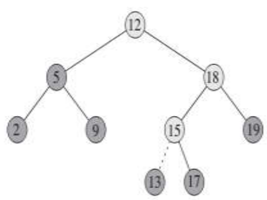


Figure 12.3 Inserting an item with key 13 into a binary search tree. Lightly shaded nodes indicate the simple path from the root down to the position where the item is inserted. The dashed line indicates the link in the tree that is added to insert the item.

Analysis of Insertion

- Initialization: $O(1)$
- While loop in lines 3-7 searches for place to insert z, maintaining parent y. This takes $O(h)$ time.
- Lines 8-13 insert the value: $O(1)$

⇒ TOTAL: $O(h)$ time to insert a node.

```
Tree-Insert(T, z)
1.  y ← NIL
2.  x ← root[T]
3.  while x ≠ NIL
4.     do y ← x
5.        if key[z] < key[x]
6.           then x ← left[x]
7.           else x ← right[x]
8.  p[z] ← y
9.  if y = NIL
10.    then root[t] ← z
11.    else if key[z] < key[y]
12.         then  left[y] ← z
13.         else right[y] ← z
```


Exercise: Sorting Using BSTs

Sort (A)
  for $i \leftarrow 1$ to n
    do tree-insert($A[i]$)
  inorder-tree-walk(*root*)

» What are the worst case and best case running times?

» In practice, how would this compare to other sorting algorithms?

Deletion from BST

Tree-Delete (T, x)

if x has no children ◆ case 0

then remove x

if x has one child ◆ case 1

then make $p[x]$ point to child

if x has two children (subtrees) ◆ case 2

then swap x with its successor

perform case 0 or case 1 to delete it

⇒ TOTAL: $O(h)$ time to delete a node

Deletion – Pseudocode

Tree-Delete(T, z)

```
/* Determine which node to splice out: either z or z's successor. */
if left[z] = NIL or right[z] = NIL
    then y ← z
    else y ← Tree-Successor[z]
/* Set x to a non-NIL child of x, or to NIL if y has no children. */
if left[y] ≠ NIL
    then x ← left[y]
    else x ← right[y]
/* y is removed from the tree by manipulating pointers of p[y]
   and x */
if x ≠ NIL
    then p[x] ← p[y]
/* Continued on next slide */
```

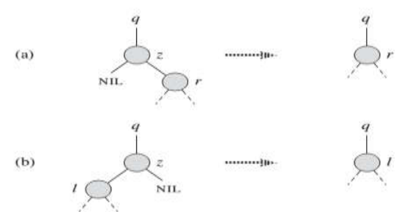

Deletion – Pseudocode

Tree-Delete(T, z) (Contd. from previous slide)

```
if p[y] = NIL
    then root[T] ← x
    else if y ← left[p[i]]
        then left[p[y]] ← x
        else right[p[y]] ← x
/* If z's successor was spliced out, copy its data into z */
if y ≠ z
    then key[z] ← key[y]
        copy y's satellite data into z.
return y
```

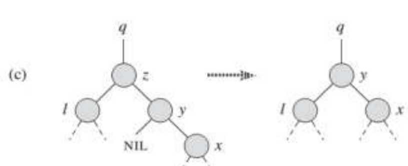

Deletion from BST

- Deleting a node z from a binary search tree. Node z may be the root, a left child of node q, or a right child of q.
- (a) ***Node z has no left child. We replace z by its right child r, which may or may not be NIL.***
- (b) **Node z has a left child l but no right child.**



Deletion from BST

- (c) Node z has two children; its left child is node l, its right child is its successor y, and y's right child is node x. We replace z by y, updating y's left child to become l, but leaving x as y's right child.



Deletion from BST

- (d) Node z has two children (left child *l* and right child r), and its successor y != r lies within the subtree rooted at r.
- We replace y by its own right child x, and we set y to be r's parent. Then, we set y to be q's child and the parent of *l*.