DSAII: Binary Search Trees

Sattiraju Prabhakar
Source:
Data Structures and Algorithms
Cormen et al
Wichita State University

Topics

- Dynamic Sets
- Binary Search Trees:
  - Organization
  - Visiting all elements
  - Querying binary search tree
  - Insertion and deletion

Dynamic Sets

- Mathematical sets are unchanging
- Algorithmic sets can grow or shrink – they are ***dynamic***.
- A ***dictionary*** is a dynamic set that supports the following operations:
  - Insert, Delete, Test Membership
- Implementation of Dynamic Sets:
  - Depends on operations that must be supported

Elements of a Dynamic Set

- Each object of set has a pointer to it
- ***key***: it is the field of the object

Operations on Dynamic Sets

- Category1: Queries
  - SEARCH(S, k): Returns a pointer x, where *key*[x] = k or Nil.
  - MINIMUM(S): Returns a pointer to the element of S with smallest key
  - MAXIMUM(S): Returns a pointer to the element of S with largest key
  - SUCCESSOR(S, x): Returns the pointer to element immediately larger than *key*[x]
  - PREDECESSOR(S, x): Returns pointer to the element immediately smaller than *key*[x]

Operations (2)

- Category2: Modification
  - INSERT(S, x): Augments set S with the element pointed to by x.
  - DELETE(S, x): Removes the element pointed to by x from S.

Binary Search Trees

- They are data structures
- They support many dynamic set operations
- Basic operations take time proportional to height: $\Theta(\lg n)$

Organization of Binary Search Tree (1)

- Organization can be understood by considering the implementation of the data structure.
- Implementation as a linked-list:
  - Node: Supports Object
    - Field: *key*
    - Field: *left*
    - Field: *right*
    - Field: *parent*

Organization of Binary Search Tree (2)

- Keys in a binary search tree are always stored in 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 key[y] ≤ key[x].
  - If y is a node in the right subtree of x, then key[x] ≤ key[y].

Examples of Binary Search Trees

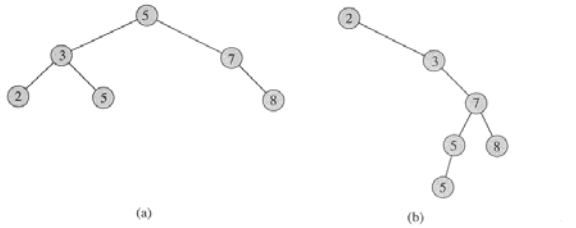

Figure 12.1 Binary search trees. For any node x, the keys in the left subtree of x are at most $key[x]$, and the keys in the right subtree of x are at least $key[x]$. Different binary search trees can represent the same set of values. The worst-case running time for most search-tree operations is proportional 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.

Tutorial

- For the set of keys {1, 4, 5, 10, 16, 17, 21}, draw binary search trees of height 2, 3, 4, 5, and 6.

Printing out all keys

- Inorder Tree Walk:
  - Print left subtree
  - Print root
  - Print right subtree
- Preorder Tree Walk:
  - Print root
  - Print left subtree
  - Print right subtree
- Postorder Tree Walk:
  - Print left subtree
  - Print right subtree
  - Print root

Inorder Tree Walk

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

Running Time = Θ(n)

Tutorial

- For the set of keys {1, 4, 5, 10, 16, 17, 21}, draw binary search trees of height 2, 3, 4, 5, and 6. Then you print out all the keys following inorder walk.

Tree Search

- Searching for a key stored in the tree.
- Given:
  - A pointer to the root of the tree and a key
- Returns:
  - Pointer to a node with key k if one exists, otherwise returns NIL.
- Traces path downward.
- Running Time: O(h), h = height of the tree

Queries on a Binary Search Tree

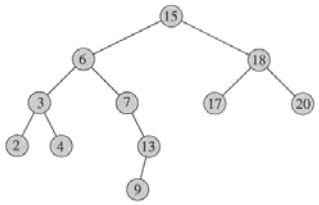

Figure 12.2 Queries on a binary search tree. To search for the key 13 in the tree, we follow the path $15 \to 6 \to 7 \to 13$ from the root. The minimum key in the tree is 2, which can be found by following *left* pointers from the root. The maximum key 20 is found by following *right* pointers from 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.

Recursive Tree Search

```
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)
```


Iterative Tree Search

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


Minimum and Maximum

- Minimum:
  - Follow the left child pointers until NIL is encountered.
- Maximum:
  - Follow the right child until NIL is encountered
- Observe: Both of them are symmetric
- Running time: O(h)

Iterative Tree Minimum

```
TREE-MINIMUM(x)
1  while left[x] ≠ NIL
2        do x ← left[x]
3  return x
```

Recursive Tree Minimum

TREE-MINIMUM(x)

1. if left[x] ≠ NIL
2. return TREE-MINIMUM (left[x])
3. return x

Tutorial

- Find the Minimum and Maximum on various binary trees created from:

  set of keys {1, 4, 5, 10, 16, 17, 21}

Tree Maximum

```
TREE-MAXIMUM(x)
1  while right[x] ≠ NIL
2        do x ← right[x]
3  return x
```


Successor and Predecessor

- Successor:
  - If all keys are distinct, the successor of a node x is the node with the smallest key greater than key[x].
- Predecessor:
  - If all keys are distinct, the successor of a node x is the node with the largest key smaller than key[x].

Successor Operation

- Two Cases:
  - If the right subtree of node x is nonempty
    - Leftmost node in the right subtree
      - By calling TREE-MINIMUM(right[x])
  - If the right subtree of node x is empty
    - Then go up the tree only if the current node is the right child
    - Example: On Page 16, successor of 13 is 15.
- Running time is O(h)

Tree Successor

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


Tree Predecessor

TREE-PREDECESSOR(x)

1. **if** *left*[x] ≠ NIL
   1. **then return** TREE-MAXIMUM(*left*[x])
2. y ←p[x]
3. **while** y≠ NIL and x = *left*[y]
   1. **do** x ← y
   2. y ←p[y]
4. **return** y

Insertion and Deletion

Tree Insert

```
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                ▷ Tree T was empty
11     else if key[z] < key[y]
12             then left[y] ← z
13             else right[y] ← z
```


Tree Delete

```
TREE-DELETE(T, z)
 1  if left[z] = NIL or right[z] = NIL
 2     then y ← z
 3     else y ← TREE-SUCCESSOR(z)
 4  if left[y] ≠ NIL
 5     then x ← left[y]
 6     else x ← right[y]
 7  if x ≠ NIL
 8     then p[x] ← p[y]
 9  if p[y] = NIL
10     then root[T] ← x
11     else if y = left[p[y]]
12             then left[p[y]] ← x
13             else right[p[y]] ← x
14  if y ≠ z
15     then key[z] ← key[y]
16          copy y's satellite data into z
17  return y
```


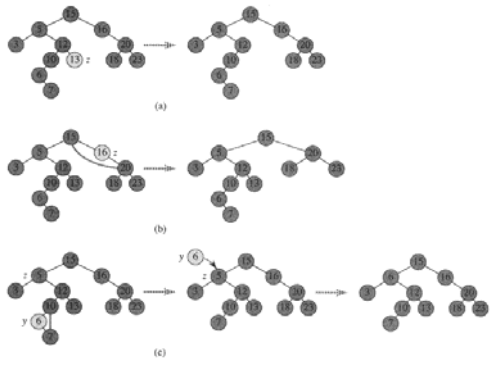

Figure 12.4 Deleting a node z from a binary search tree. Which node is actually removed depends on how many children z has; this node is shown lightly shaded. **(a)** If z has no children, we just remove it. **(b)** If z has only one child, we splice out z. **(c)** If z has two children, we splice out its successor y, which has at most one child, and then replace z's key and satellite data with y's key and satellite data.

Radix Tree Sorting

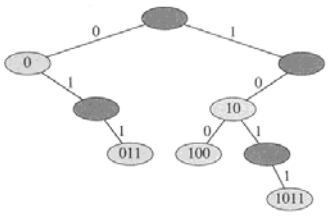

Figure 12.5 A radix tree storing the bit strings 1011, 10, 011, 100, and 0. Each node's key can be determined by traversing the path from the root to that node. There is no need, therefore, to store the keys in the nodes; the keys are shown here for illustrative purposes only. Nodes are heavily shaded if the keys corresponding to them are not in the tree; such nodes are present only to establish a path to other nodes.