**Binary Trees (Ch. 12)**

Binary trees are an efficient way of storing data so that searches can be O(lgn).

Used when we need a data structure that supports dynamic set operations, e.g., for tree S and key x (searching is primary purpose of BST)

- INSERT(S, x)
- Search(S, k)
- MINIMUM(S), MAXIMUM(S)
- SUCCESSOR(S, x), PREDECESSOR(S, x)
- Preorder, Inorder and Postorder traversal help to evaluate expressions

---

**Binary Search Trees**

Requirements for Binary Search Tree (BST):
1. Must be a binary tree.
2. All keys must be unique (key values are like individual ID numbers).
3. Each node in tree is the root of a BST such that:
   - All nodes to left of root will have keys < root.key, and
   - all nodes to right will have keys > root.key.

Main advantage of BST is rapid search and low memory use; memory use is dependent only on size of data set.

Time of search = O(depth of BST) = maximum number of nodes on root to leaf path.

---

**Binary Search Trees**

Every binary tree node (internal or leaf) contains a key. These keys are unique.

**Binary Search Tree Property:** For every node x in tree,

- y.key < x.key for every y in x.left (left subtree of x)
- y.key > x.key for every y in x.right (right subtree of x)

BST has stronger requirements than heaps do.

These dynamic set operations are supported on BST S and key x

- INSERT(S, x), DELETE(S, x)
- SEARCH(S, x), MINIMUM(S), MAXIMUM(S)
- SUCCESSOR(S, x), PREDECESSOR(S, x)
- InorderTraversal( S, x ) (to list or print sorted set)

---

**Binary Search Trees**

A BST can be implemented as either a hierarchical list or as a sequential array.

The hierarchical list representation is better in terms of storage required, because only the amount of storage needed is used. Most algorithms are written for a hierarchical tree with fields for key, right, and left subtrees (and a parent pointer if needed).

The sequential array implementation has better performance when the BST is complete—otherwise there are holes in the array = wasted memory.


parent of A[i] is at A[floor of i/2]

---

**Structure of bst nodes**

Each bst node contains fields left, right, and key. Each bst node is the root of a binary search tree. Assume all keys are unique and all leaves are NIL (singly-linked bst)

```
    X
   /   
  E     R
 / \    / \ 
F  T  G  H  
```

keys < x.key or NIL  keys > x.key or NIL
Structure of bst nodes
A doubly-linked bst node contains fields left, right, parent and key. Each bst node is the root of a binary search tree.

keys < x.key or NIL
keys > x.key or NIL

Binary Search Trees
The BST's total ordering does the heap's partial ordering one better; not only is there a relationship between a BST node and its children, but there is also a relationship between the children, i.e. the value of a node's left child is always less than the value of its right child.

BST Insert
Insert(T, z)
1. y = NIL
2. x = T.root
3. while x ≠ NIL
4. y = x
5. if z.key < x.key
6. x = x.left
7. else
8. x = x.right
9. z.parent = y
10. if y = NIL
11. then x = NIL & parent of z is set to y
12. else if z.key < y.key
13. then y.left = z
14. else y.right = z
15. return x

Input is a BST T and a node z such that z.left = z.right = z.parent = NIL.
All leaves in T are NIL.
Every node is inserted as a leaf.

BST Search
Iterative-Tree-Search(x, k) Recursion-Tree-Search(x, k)
1. if (x == NIL) or (k == x.key) return x
2. if (k < x.key) return Recursion-Tree-Search (x.left, k)
3. else return Recursion-Tree-Search (x.right, k)

Both have running times of O(h), where h is the height of the tree.

BST Min & Max
The minimum element in a BST can always be found by following left child pointers to a leaf (until a NIL left child pointer is encountered). Likewise, the maximum element can be found by following right child pointers to a leaf.

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 have running times of O(h), where h is the height of the tree.

BST Inorder Successor
In a BST, the Inorder Successor of x can also be defined as the node with the smallest key greater than the key x. This algorithm is used when deleting a node from a BST.

1. if x has a right child, then x.successor is the smallest node in the subtree rooted at x.right.
2. if x has no right child, then x.successor is the nearest ancestor of x whose left child is either an ancestor of x or x itself.

Tree-Successor(x)
1. if x.right ≠ NIL
2. return Tree-Minimum(x.right)
3. temp = x.parent
4. while temp ≠ NIL and x == temp.right
5. x = temp
6. temp = temp.parent
7. return temp

Both have running times of O(h), where h is the height of the tree.
**BST Inorder Successor**

Case where \( x.\text{right} \) is not equal to NIL

Tree-Successor(x)

1. if \( x.\text{right} \neq \text{NIL} \)
2. return Tree-Minimum(\( x.\text{right} \))

**BST Inorder Predecessor**

Tree-Predecessor(x)

1. if \( x.\text{left} \neq \text{NIL} \)
2. then return Tree-Maximum(\( x.\text{left} \))
3. temp = \( x.\text{parent} \)
4. while temp \( \neq \text{NIL} \) and \( x = \text{temp.\,right} \)
5. \( x = \text{temp} \)
6. temp = temp.\,parent
7. return temp

**BST Delete**

Input: BST T and node z to be deleted. Three cases:

1. \( x \) has no children. Just remove it.
2. \( x \) has only one child. Splice out \( x \), by letting \( x.\text{child} \) replace \( x \).

**BST Delete**

1. if \( z.\text{left} \equiv \text{NIL} \) or \( z.\text{right} \equiv \text{NIL} \)
2. \( y = z \)
3. else
4. \( y = \text{Tree-Successor}(z) \)
5. if \( y.\text{left} \equiv \text{NIL} \)
6. \( x = y.\text{right} \)
7. else
8. \( x = y.\text{left} \)
9. if \( x \equiv \text{NIL} \)
10. \( x.\text{parent} = y.\text{parent} \)
11. \( \text{T\,root} = x \)
12. else
13. if \( y \equiv y.\text{parent.\,left} \)
14. \( y.\text{parent.\,right} = x \)
15. else
16. \( x = y.\text{parent.\,right} \)
17. if \( x \neq \text{NIL} \)
18. swap key[x] and key[y]
19. return y

**Cases where x.right = NIL**

1. if \( x.\text{right} \neq \text{NIL} \)
2. \( \text{return} \) Tree-Minimum(\( x.\text{right} \))
3. temp = \( x.\text{parent} \)
4. while temp \( \neq \text{NIL} \) and \( x = \text{temp.\,right} \)
5. \( x = \text{temp} \)
6. temp = temp.\,parent
7. \( \text{return} \) temp

**Cases where x.left = NIL**

1. if \( x.\text{left} \neq \text{NIL} \)
2. then return Tree-Maximum(\( x.\text{left} \))
3. temp = \( x.\text{parent} \)
4. while temp \( \neq \text{NIL} \) and \( x = \text{temp.\,left} \)
5. \( x = \text{temp} \)
6. temp = temp.\,parent
7. \( \text{return} \) temp

**Cases where x.left not equal to NIL**

1. if \( x.\text{left} \neq \text{NIL} \)
2. then return Tree-Maximum(\( x.\text{left} \))
3. temp = \( x.\text{parent} \)
4. while temp \( \neq \text{NIL} \) and \( x = \text{temp.\,left} \)
5. \( x = \text{temp} \)
6. temp = temp.\,parent
7. \( \text{return} \) temp

**Input:** BST T and node z to be deleted. Three cases:

1. \( z \) has two children. Find \( z \)'s successor \( y \), which has at most one child.
   - Since \( y \) is \( z \)'s successor, then \( y \) can have no left child, but it may have a right child.
2. If \( y \) is \( z \)'s right child, then replace \( z \) by \( y \), leaving \( y \)'s right child as is.
3. If \( y \) is in \( z \)'s right subtree, but is not \( z \)'s right child, first replace \( y \) by its own right child and replace \( z \) by \( y \).
**BST Transplant**

Replaces one subtree (rooted at u) with another subtree (rooted at v).

When Transplant replaces the subtree rooted at node u with the subtree rooted at node v, node u's parent becomes node v's parent and node u's parent sets node v as its appropriate child.

This procedure simplifies the code for deleting a node from a BST.

**BST Tree-Delete with Transplant**

Tree-Delete(T, z)
1. if z.left == NIL             ;; Case 1 or 2
2. Transplant(T, z, z.right)   ;; Case 2
3. else if z.right == NIL      ;; Case 3
4. Transplant(T, z, z.left)
5. else
6. y = Tree-Minimum(z.right)   ;; Case 3
7. if y.p != z
8. Transplant(T, y, y.right)   ;; Case 3
9. y.right.p = y
10. Transplant(T, z, y)        ;; Case 3
11. y.left = z
12. y.left.p = y

Case 1: z is a leaf
Case 2: z has one child
Case 3: z has two children

**Postorder Traversal**

Postorder traversal is a recursive algorithm used to produce a postfix expression from an expression tree. All leaf nodes have NIL left and right children.

Postorder-Tree-Walk(x)  /** start at root x **/
1. if x != NIL
2. Postorder-Tree-Walk(x.left)
3. Postorder-Tree-Walk(x.right)
4. visit the root

Running time = \( \Theta(n) \) (each node must be visited at least once)

**Preorder Traversal**

Preorder traversal is a recursive algorithm that is used to get a prefix expression from an expression tree. All leaf nodes have NIL left and right children.

Preorder-Tree-Walk(x)  /** start at root x **/
1. if x != NIL
2. visit the root
3. Preorder-Tree-Walk(x.left)
4. Preorder-Tree-Walk(x.right)

Running time = \( \Theta(n) \) (each node must be visited at least once)

**Inorder Traversal**

Inorder-Tree-Walk(x)  /** start at root x **/
1. if x != NIL
2. Inorder-Tree-Walk(x.left)
3. visit the root
4. Inorder-Tree-Walk(x.right)

In a BST, inorder-Tree-Walk(root) visits the keys in ascending order.

Running time = \( \Theta(n) \) (each node must be visited at least once)
Expression Trees

A binary expression tree is a specific kind of a binary tree used to represent arithmetic expressions. An inorder traversal yields an infix arithmetic expression.

Minimizing Running Time

Problem: worst case for binary search tree height is $\Theta(n)$ - no better than a linked list.

Solution: Guarantee tree has small height by making sure it is balanced so that $h = O(\log n)$.

Method: restructure the tree if necessary. No extra work for searching, but requires extra work when inserting or deleting.

Red-black, AVL and 2-3 trees: special cases of binary trees that avoid the worst-case behavior by ensuring that the tree is nearly balanced at all times.