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)

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
  - L
  - E
  - F
  - T

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.

- X
  - L
  - E
  - F
  - T
  - PARENT
  - KEY

All leaves in BST are NIL
If X is root of BST, parent is NIL
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.

In fact, every node in the left subtree of a node x has key < x.key and every node in the right subtree of x has key > x.key

**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. T.root = z
12. else if z.key < y.key
13. y.left = z
14. else
15. y.right = z

**BST Insert**

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)

1. while (x != NIL) and (k != x.key) \ base cases
2. return x
3. if k < x.key
4. x = x.left
5. else
6. x = x.right
7. return x

The iterative version is more efficient, in terms of space used, on most computers.

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

Recursive-Tree-Search(x, k)

1. if (x == NIL) or (k == x.key) \ base cases
2. return x
3. if (k < x.key) \ recursive case 1: search left
4. return Recursive-Tree-Search(x.left, k)
5. else \ recursive case 2: search right
6. return Recursive-Tree-Search(x.right, k)

**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)

1. while x.left ≠ NIL
2. x = x.left
3. return x

Tree-Maximum(x)

1. while x.right ≠ NIL
2. x = x.right
3. 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 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
BST Inorder Predecessor

Tree-Predecessor(x)
1. if x.left != NIL then
2. then return Tree-Maximum(x.left)
3. temp = x.parent
4. while temp != NIL and x = temp.left
5. x = temp
6. temp = temp.parent
7. return temp

- If x has a leftchild, then x.predecessor is the largest node in the subtree rooted at x.left.
- If x has no leftchild, then x.predecessor is the nearest ancestor of x whose right child is either an ancestor of x, or x itself.

BST Inorder Successor

Case where x.right is not equal to NIL

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

- Return value?

BST Delete

Input: BST T and node z to be deleted. Three cases:
1) z has no children. Just remove it.
2) z has only one child. Splice out z, by replacing z with z's child.
3) 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.
If y is z's right child, then replace z by y, leaving y's right child as is.
If y is z's right subtree, but is not z's right child, first replace z by its own right child and replace z by y.
Binary Search Trees

Class exercise: Delete the key 6 in BST A and show the resulting tree.
Class exercise: Delete the key 2 in BST B and show the resulting tree.

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.

Postorder Traversal

Postorder traversal is a recursive algorithm used to produce a postfix expression from an expression tree (a binary 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, prints out keys in ascending order.

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

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.