Data Structures for Dynamic Sets

Algorithms operate on data, which can be thought of as forming a set S. The data sets manipulated by algorithms are dynamic, meaning they can grow, shrink, or otherwise change over time.

Data Structures are structured ways to represent finite dynamic sets. Different data structures support different kinds of data manipulations, e.g.,

- **dictionary**: insert, delete, test for membership
- **priority queue**: insert, extract-max

Operations on Dynamic Sets

- **INSERT(S, x)** adds element pointed to by x to S
- **DELETE(S, x)** removes element pointed to by x from S
- **SEARCH(S, k)** returns pointer to element x with key[x] = k (or nil)
- **MINIMUM(S)** returns element with smallest key
- **MAXIMUM(S)** returns element with largest key
- **SUCCESSOR(S, x)** returns element with next key > key[x]
- **PREDECESSOR(S, x)** returns element with next key < key[x]

Running Time of a dynamic set operation is usually measured in terms of the size of the set (i.e., number of elements currently in the set).

Elementary Data Structures (Ch. 10)

These elementary data structures should be a review from CS102:

- **arrays** and **linked lists** (singly linked, doubly linked)
- **stacks** (e.g., implemented with arrays and lists)
- **queues** (e.g., implemented with arrays and lists)
- **rooted trees** (e.g., arbitrary trees using pointers, complete d-ary trees using arrays)

Hash Tables (Ch. 11)

Rely on distribution of data, amount of data, and storage mechanism to provide expected O(1) time for many, but not all, dynamic set operations.

Covered in Lecture 9.

Binary Search Trees (Ch. 12)

Every tree node (internal or leaf) contains a unique key.

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)

All dynamic set operations are supported on BSTs

- **INSERT(S, x)**, **DELETE(S, x)**
- **SEARCH(S, k)**, **MINIMUM(S)**, **MAXIMUM(S)**
- **SUCCESSOR(S, x)**, **PREDECESSOR(S, x)**

BST Insert

Input is a BST T and a node z such that
z.left = z.right = NIL
(z is a node with NIL children; aka a leaf)

Every operation on a BST starts at the root and works down to a leaf. Every node is inserted as a leaf.

```latex
\begin{align*}
\text{BST Insert} & \quad T \rightarrow T' \\
1. & \quad y = \text{NIL} \\
2. & \quad x = T.\text{root} \\
3. & \quad \text{while } x \neq \text{NIL} \\
4. & \quad y = x \\
5. & \quad \text{if } z.\text{key} < x.\text{key} \\
6. & \quad x = x.\text{left} \\
7. & \quad \text{else } x = x.\text{right} \\
8. & \quad z.\text{parent} = y \\
9. & \quad \text{if } y = \text{NIL} \\
10. & \quad T.\text{root} = z \\
11. & \quad \text{else if } z.\text{key} < y.\text{key} \\
12. & \quad y.\text{left} = z \\
13. & \quad \text{else } y.\text{right} = z
\end{align*}
```
**BST Insert**

The tree nodes each have three links, each of the same type (tree node):
- `x.left /* leaf nodes are NIL */`
- `x.right`
- `x.parent /* parent of root is NIL */`

**BST Traversals**

- The BST property allows us to print out all keys in sorted order using a simple recursive algorithm called an inorder tree walk. Strategy: visit `x.left`, print `x.key`, visit `x.right`

```
INORDER-TREE-WALK(T.root)
1. if x ≠ NIL
2. INORDER-TREE-WALK(x.left)
3. print x.key
4. INORDER-TREE-WALK(x.right)
```

Running time?

= $\Theta(n)$ (each node must be visited at least once)
(formal proof in chapter 12)

**BST Search**

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.

**BST Min & Max**

The minimum element in a BST can always be found by following left child pointers to a leaf (when 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. y = x.parent
3. while y ≠ NIL and x == y.right
4. x = y
5. y = y.parent
6. return y
```

```
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**

The inorder successor of a node $x$ in BST $T$ is the node with key immediately following $x$ in an inorder traversal of $T$.

- If $x.right ≠ NIL$, then `SUCCESSOR(x)` is the smallest node in the subtree rooted at $x.right$.
- If $x.right = NIL$, then `SUCCESSOR(x)` is the nearest ancestor of $x$ whose left child is either an ancestor of $x$ or $x$ itself.

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

**BST Inorder Successor**

Case where $x.right$ not equal to NIL (successor in red)

Cases where $x.right$ equal to NIL (successor in red)
**BST Inorder Predecessor**

1. **PREDECESSOR(x)**
   1. if x.left != NIL
   2. return Tree-MAXIMUM(x.left)
   3. y = x.parent
   4. while y != NIL and x == y.left
   5. x = y
   6. y = y.parent
   7. return y

The predecessor of a node x in BST T is the node with key immediately preceding key.x in an inorder traversal of T.

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

---

**Deleting a node from a BST**

Conceptually, deleting node z from BST T has 3 cases:

1. If z has no children, just delete it and replace with NIL at z.parent.
2. If z has just one child, then make that child take z’s position in T, dragging the child’s subtree along unchanged.
3. If z has two children, then find z’s successor y and replace z by y in the tree. Node y must be in z’s right subtree and have no left child. 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.

Deletion is tricky because we don’t want to leave any holes in the tree, so we need to “splice” subtrees in certain cases.

---

**Deleting node z from a BST**

TREE-DELETE(T, z) has five cases when deleting node z from bst T:

1. If z has no children, just replace it with NIL.
2. a. If z.left == NIL and z.right != NIL, replace z by z.right:

```
     z
    /   \
   /     \
 z.left   z.right
```

b. If z.right == NIL and z.left != NIL, replace z by z.left:

```
     z
    /   \
   /     \
 z.left   z
```

3. If z has 2 children, find y = SUCCESSOR(z). Node y must be in z’s right subtree and y must have no left child. Goal is to replace z by y, splicing y out of its current location. 2 cases:

   a. If y == z.right, replace z by y and leave y.right as-is:

```
     z
    /   \
   /     \
 z.left   z.right
```

---

**Deleting a node from a BST**

TRANSPLANT is a subroutine used to move subtrees around within T. TRANSPLANT(T, u, v) replaces the subtree rooted at u by the subtree rooted at v:

- Replaces u by v as either left or right child of u.parent, depending on whether u is a left or right child.
- Makes u’s parent (u.parent) become v’s parent (unless u is the root, in which case v becomes the root).
- Leaves update of v.left or v.right up to calling procedure.

```
TRANSPLANT(T, u, v)
1. if u.parent == NIL // u is root
2. T.root = v
3. else if u == u.parent.left
4. u.parent.left = v
5. else u.parent.right = v
6. if v != NIL
7. v.parent = u.parent
```
Deleting node z from a BST

3. If z has 2 children, find \( y = \text{SUCCESSOR}(z) \). Node y must be in z’s right subtree and y must have no left child. Goal is to replace z by y, splicing y out of its current location. 2 cases:

b. Else, y is in z’s right subtree but y is not the root of this subtree. Replace y by y.right, x, dragging along any subtrees of x. Then replace z by y, making y.left = z.left and y.right = z.right.

BST Delete

Input: BST T containing node z to be deleted. Three cases:
1) (lines 1-2) z has no children. Just remove it.
2) (lines 1-4) z has only one child. Splice out z.
3) (lines 5-13) z has two children. Splice out z’s successor y, which has at most one child, swap z and y’s data, and return y.

Minimizing Running Time

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

Solution: Guarantee tree has small height by balancing tree, 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 (however, this extra work is often constant or \( O(\log n) \)).

Red-black, AVL, and 2-3 trees: special cases of binary trees that avoid the worst-case shape of a BST by ensuring that the tree is nearly balanced at all times during construction. To achieve a probabilistically balanced tree, randomize the sequence of key insertions.