Binary Search Trees

Heaps are good for insertion, deletion, searching.

Priority heaps are good for minimum/maximum.

Binary search trees (BSTs) are a useful data structure to implement dictionary operations, min, max, successor, predecessor.

- basic operations take time O(height tree)
- randomly built BST with n nodes has height $\lg(n)$
- lots of other variants
  - red-black trees (guaranteed to have height $\lg(n)$)
  - AVL trees (also guaranteed to have height $\lg(n)$)
  - B-trees (used extensively in databases; have large outdegree and smaller height)
  - splay trees
  - persistent trees
- The great number of variants is an indication of the importance of BSTs.

The Binary-Search Tree Property

A binary search tree is a binary tree where each node has key, parent, left child, right child

- $p[x] =$ NIL for the root
- $left[x]$, $right[x]$ may be NIL

The keys must satisfy the *binary-search-tree* (BST) property:

> If y is a node in the left subtree of x, then $key[y] \leq key[x]$
> If y is a node in the right subtree of x, then $key[y] \geq key[x]$

Note: This property makes sense only if the keys are totally ordered

Searching a Binary Search Tree

Searching is easy because of the BST property:

TREE-SEARCH(x, k) $\quad$ [x is a pointer to a node]

```
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)
```

- This tells us whether k appears in the subtree rooted at x
- running time: $O(h(x))$, where $h(x)$ is the height of x

Here is a non-recursive version:

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

Min and max are easy: just go all the way to the left/right:

TREE-MINIMUM(x) [x is a pointer to a node]

```
1  while left[x] ≠ NIL
2      do x ← left[x]
3  return x
```

TREE-MAXIMUM(x)

```
1  while right[x] ≠ NIL
2      do x ← right[x]
3  return x
```

Successor and Predecessor

The successor of x is the element with the next-biggest key

- May want successor if you want to list keys in increasing order
- Again, this makes sense only if keys are totally ordered

Where is the successor of x located?

1. If x has a right child, then it's the leftmost node of the subtree rooted at the right child.
   - Clearly this is the successor of x in the subtree rooted at x
   - Work up the tree by induction from x to show that this remains true
2. If x has no right child, and x is the left child of its parent, then the successor is the parent
   - Again, need to argue by induction up the tree that this is right
3. If x is the right child of its parent, find the lowest ancestor of x which is the left child of its parent

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 works the same way
- Both run in time $O(h)$:
  - We either go up the tree or down the tree

Insertion

Inserting z is straightforward:

- We insert z at a leaf
- Figure out which one by starting at the root and making comparisons

TREE-INSERT(T, z)

```
1  y ← NIL
2  x ← root[T]        [y is the parent of x]
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
```

Insertion clearly runs in time $O(h)$

Deletion in BSTs

Deleting z is the trickiest operation. There are three cases:

1. z has no children: easy – just delete z
2. z has one child: easy – delete z; child of z becomes child of z's parent
   - we still maintain the BST property
3. if z has two children
   - Find z's successor z'
     - this will be the leftmost element in the subtree rooted at $right[z]$
   - recursively delete z'
     - this is easy because z' has at most one child (no left child)
   - Replace z by z'
   - This maintains the BST property

TREE-DELETE(T, z)

```
1   if left[z] = NIL or right[x] = NIL
2      then y ← z
3      else y ← TREE-SUCCESSOR[z]
          [y is the node that gets spliced out]
4   if left[y] ≠ NIL
5      then x ← left[y]
6      else  x ← right[y]
          [x is the unique successor of y (or NIL)]
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     key[z] ← key[y]
          [also copy other fields, if there are any]
```

Again, the running time is $O(h)$.

The Height of a Random BST

All the algorithms run in time $O(h)$.

What's h for an n-node tree?

- best case: $\lg(n)$ – if the tree is perfectly balanced
- worst case: $O(n)$ – if the tree is completely unbalanced

What can we expect on average?

Let's assume the tree is built up by starting with an empty tree and inserting n elements.

- it's very hard to analyze what happens if we have inserts + deletes
    - deletes could unbalance a tree—if a node has two children, we delete from the right subtree.

If the n elements are in increasing or decreasing order, then we have a completely unbalanced tree.

- This can be a serious problem in practice
- Running time $O(n)$ is not acceptable
- Red-black trees solve that problem

If all the $n!$ permutations of the trees are equally likely, then the expected height of the tree is $O(\lg n)$.

Using a BST for Sorting

Can sort using a BST by doing an *inorder* traversal

- first left subtree, then root, then right subtree

INORDER-TREE-WALK(x) [walk through subtree rooted at x]

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

Analysis: first need to build the BST by inserting elements to be sorted. This takes expected time

$$O(\lg(1)) + \cdots + O(\lg n) = O(n \lg n)$$

The tree walk then takes time $O(n)$.

Balanced Search Trees

The BSTs just presented only have expected height $O(\lg n)$. There are a number of variants which are *guaranteed* to have height $O(\lg n)$:

- red-black trees (CLR; Chapter 14)
- AVL trees
- ...

Keeping the tree balanced requires (lots of) additional overhead, although the basic ideas remain the same.

Skip Lists

This material is NOT in the text.

- There is a handout

Skip lists support dictionary operations, min, max, successor, predecessor.

- These operations have expected running time $O(\lg n)$
- Worst-case time can be $O(n)$
- Advantages:
  - very simple to code (much simpler than fancy balanced BSTs)
  - algorithm tosses coins, so expected running time is independent of actual list
    - unlike BSTs

Given a set X of elements, a skip list S for X consists of a set $\{S_0, \ldots, S_h\}$ of subsets of X:

- each S_i is implemented as a doubly-linked list
- S_0 consists of all the elements of S, in sorted order, + two special elements $-\infty$ and $+\infty$
- S_{i+1} is a subset of S_i, again in sorted order
  - S_{i+1} must have $+\infty$ and $-\infty$
  - typically S_{i+1} is about half the size of S_i
  - ideally S_{i+1} has every other element in S_i
    - exact size depends on the coin tosses
  - $S_h = \{-\infty, +\infty\}$
    - typically h is about $\lg n$
    - hardly ever $> 3 \lg n$ (can make sure of this)
  - have links up and down from corresponding elements in S_i and S_{i+1}
  - Skip list S has operations *after*, *before*, *above*, *below*
  - $top[S] = h$

Searching a Skip List

Why do we bother repeating the elements in a skip-list?

- Because it makes searching, inserting, deleting, etc. faster!

Idea in searching for k:

- start at the top level (S_h), and find largest $k' \leq k$
- then go down one level and repeat
- if we don't hit k by S_0, it's not there

SkipSearch(S, k)

- returns x in S_0 such that $key[x]$ is greatest key in $S \leq k$
- if $key[x] \neq k$, then k is not in S

```
1  i ← top[S]    [top[S] is highest level of S]
2  x ← tail[S_i]
3  while i ≠ −1
4      do if key[after[x]] ≤ k
5            then x ← after[x]
6            else  if i ≠ 0
7                     then x ← below[x]
8                  i ← i − 1
9  return x
```

Insertion in Skip Lists

Suppose we want to insert item x with key k into the skip list.

Two problems:

1. which lists do we put it into
   - S_0 for sure. How about S_1? S_2?
2. How do we find the right place to put it quickly?

Solutions:

1. Decide probabilistically: toss a coin.
   - If it lands heads, put it in S_1
   - If it lands heads again, put it in S_2
   - ...
   - quit tossing if coin lands tails
2. Do SkipSearch to find the right place quickly

We implement a coin toss by calling RANDOM(), which returns a number in $[0, 1)$

- coin lands heads if RANDOM() $< 1/2$

SkipInsert(S, x)

```
1   k ← key[x]
2   y ← Skip-Search(S, k)
3   Insert x after y in S_0
4   i ← 0
5   while random() < 1/2
6       do while above[y] = nil and key[y] ≠ −∞
7              do y ← before[y]
8          i ← i + 1
9          if  i > top[S]
10             then top[S] ← i
11                  initialize S_i
12         y ← above[y]
13         Insert x after y in S_i
           [fix before, after, above, below]
```

Running time = time of SkipSearch + $O(top[S])$

- Need to show that SkipSearch runs in expected time $O(\lg n)$
- Also need to show that the expected number of backtracks before $above[y] \neq$ nil is constant.

As written, $top[S]$ could grow unboundedly

- this is *extremely* unlikely — requires lots of heads

Could stop $top[S]$ at $3 \lg n$.