CSCI S-Q Lecture #5

7/6/98

• Balanced Trees
  • What does "balanced" mean?
  • Why do the work of balancing?
  • B-Trees
    • ADT Operations
    • Implementation
    • Analysis
  • Splay Trees
    • ADT Operations
    • Implementation
    • Analysis
• Tries
What does "Balance" mean?

- **Full**
  A non-empty binary tree is **full** if it has no nodes with only one child. The number of leaves is one more than the number of nonleaves.

- **Complete**
  All leaves are at depth \(d\) or \(d-1\), **and** all leaves at depth \(d\) are to the left of all leaves at depth \(d-1\).

- **Perfect**
  - A perfect binary tree is a full binary tree in which all leaves have the same depth. A perfect binary tree of height \(h\) has \(2^{h+1}-1\) nodes -- \(2^h\) are leaves, and \(2^h-1\) are non-leaves.
Why Balance Trees?

- Remember the ADT operations on Binary Search Trees:
  - **Expected Case** $O(\log n)$ - because at each node, half of the possibilities are expected to be eliminated
  - **Worst Case** is $O(n)$ - if the tree is "tall" but not "broad", it can behave like a linked list

- How can we prevent the worst case?
  - One method is to maintain balance after all operations, so that $O(\log n)$ is guaranteed.
  - Another approach is to allow average (or amortized) performance of $O(\log n)$, but not require every operation to be $O(\log n)$. 
B-Trees - ADT Operations

• \( v = \text{FIND} \ (k) \)
  
  • Search the tree for the given key, returning its associated value. There must be a distinct value, which we’ll call \( nil \), that denotes a failed search.

• \( \text{INSERT} \ (k, v) \)
  
  • Insert the value with the given key into the tree.

• \( \text{DELETE} \ (k) \)
  
  • If the given key exists in the tree, remove it.

Properties of a B-Tree of Order M

• All internal nodes (except the root) have between \( \lceil \frac{M}{2} \rceil \) and \( M \) children.

• The root is either a leaf or has between 2 and \( M \) children.

• All non-root leaves are at the same depth, and contain \( \lceil \frac{M}{2} \rceil \) to \( M \) keys.

• All data is stored in the leaves (either as the keys themselves, or as pointers to structures containing the keys).
Examples of B-Trees of Order 4

```
+---+ +---+ +---+ +---+
| e | i | m |
+---+ +---+ +---+ +---+
  | a,b,c,d | e,f,g,h | i,j,k,l | m,n,o,p |
+---+ +---+ +---+ +---+
      +---+ +---+ +---+
      | d | g |
+---+ +---+ +---+ +---+
  | a,b,c | d,e,f | g,h,i,j |
+---+ +---+ +---+ +---+
      +---+ +---+ +---+
      | c |
+---+ +---+ +---+ +---+
  | a,b | c,d |
+---+ +---+ +---+ +---+
      +---+ +---+ +---+
      | a,b,c,d |
+---+ +---+ +---+ +---+
      +---+ +---+ +---+
      | a |
+---+ +---+ +---+ +---+
```

Page 5
The following are NOT valid B-Trees of Order 4. Why not?

- b, c, d
- a, b, c, d, e
- a, b, c, d, e
- d, g
- a, b, c, d, e
- j
- g, h, i
- j, k, l
B-Tree FIND (k)

• Since the keys are in order, we know where to find them, i.e. we know how to search. Decide which subtree to investigate.

Example: v = FIND (’n’)

```
          l  o
         /    |
   a,d,h,j  l,m  o,r,s
```

• Start at the Root.

1. If it’s a leaf:
   • Search the ordered list of keys from the beginning, until either
     • a match is found - return the value
     • a key is found that is greater than the key we’re searching for - return nil

2. Otherwise
   • Search the ordered list of keys from the beginning. If the key we are searching for is less than the current key, follow the link to the "previous" child. Go to step 1.
B-Tree INSERT \((k, v)\)

- Follow the path as if doing a FIND. When a leaf is reached:

1. If there is room in the leaf, simply insert the key into the list in order.

2. Otherwise:

   - Split the leaf into two leaves of size \(\text{ceiling}(M/2)\) and \(\text{floor}(M/2)\), placing the keys from the original leaf into these new leaves in order.
   - If there is room in the parent, add the new child to the parent, updating pointers and keys as necessary.
   - Otherwise, split the parent (as with the leaf), and repeat until no more splitting is necessary.
   - If necessary, create a new root with \(\text{ceiling}(M/2)\) children. If this happens, the whole tree will become one level deeper.

Example: Insert \((f)\) in the Order 4 B-Tree below:
**B-Tree DELETE (k)**

- Find the key to be deleted, and remove it.

- If the removal leaves fewer than ceiling(M/2) keys:
  
  - If a neighboring sibling has more than ceiling(M/2) keys, steal a key from that sibling, and update the parent.
  
  - Otherwise, combine the node with a sibling to make one node. The parent loses a node and may need to be combined with a sibling using one of these strategies. Continue moving up the tree until no more nodes need to be combined.

- This may percolate up to the root. If the root needs to be deleted, the tree becomes one level shallower.

Example: Delete (d) from this Order 4 B-Tree:
Analysis of B-Trees

• Given an M-ary B-Tree with N nodes:

  FIND, INSERT and DELETE are all $O(\log N)$

  (M and $\log M$ are constant factors, so we remove them from the analysis. See Weiss for more details on these constant factors.)

• The best choices for M have been shown empirically to be $M = 3$ or $M = 4$ when using B-Trees for in-memory applications. However, B-Trees are often used when saving data to disk. When using this data structure that way, M is usually chosen to allow an interior node to fit into one disk block, and the maximum number of elements in a leaf is also chosen to fit in a single disk block. This strategy minimizes the number of disk accesses, which are usually at least an order of magnitude slower than memory accesses.
AVL Trees

• AVL trees (described in Weiss, section 4.4) keep the tree balanced by imposing a balance condition:

  • The height of the left subtree and the height of the right subtree of every node differ by at most 1

• AVL trees use a technique called rotation to maintain the balance condition.

• We won’t discuss about how AVL trees use rotation - you can read about this in the book, if you are interested. Instead, we’ll use the rotation technique to construct balanced trees using a different algorithm.
Single Rotation

- We can "rebalance" the tree below, keeping it in order.

Use rotation around 'B'.

This method works with the symmetric case, too.
Double Rotation

Q: How can we "balance" the tree below? A single rotation won’t fix the problem.

A: Do two rotations. This is called a double rotation.

Notice that the first rotation might not improve the balance - it fact, it may make it worse!

Double rotation is usually implemented as a single step. We’ve shown it as two single rotations for the sake of clarity.
Splay Trees - Using Amortization

- Splay trees guarantee that any $M$ consecutive tree operations (starting from an empty tree) take at most $O(M \log n)$ time (where $n$ is the number of nodes in the tree).
  - There is no guarantee that a single operation will take $O(\log n)$ time
  - In fact, some of the operations can take $O(n)$ time, and we cannot completely anticipate when these expensive operations will occur.
  - On average, however, each operation will take only $O(\log n)$ time

Splay Trees - ADT Operations

- **FIND** ($k$) - Determine whether $k$ is in the tree.
- **INSERT** ($k$) - Insert $k$ into the splay tree.
- **DELETE** ($k$) - Delete $k$ from the splay tree.

- Every time a key is accessed by one of these operations, its node is moved to the root of the tree by using an operation called **splay** ($k$).
Implementation of Splay Trees

- Assume we have an operation called **splay** (which we’ll define in a few minutes).

- Find (k) - Execute splay (k), and then examine the root to see if it contains k.

- Insert (k) - Insert k as you would in a normal Binary Search Tree, then splay (k).

- Delete (k) - Execute splay (k). If the root doesn’t contain k then you’re done. Otherwise, "concatenate" the two subtrees of the root, $T_L$ and $T_R$:
  - Find the largest element in $T_L$. Splay this element to the top of $T_L$, so that $T_L$ now has no right child.
  - Make $T_R$ the right child of $T_L$.  

Splay (k)

1. Search for k in the Binary Search Tree as you normally would. As you go, keep track of the path, using a stack. X will be the last node inspected. X either contains k, or has an empty child where the search would have continued.

2. Perform a series of rotations, as you travel along the path from P back to the root:
   - Case 1: Parent(X) is the root. (Weiss calls this zig).
     - Perform a single rotation around the parent of P.
   - Case 2: X and Parent(X) are both left children or both right children (Weiss calls this zig-zig).
     - Perform two single rotations in the same direction, first around the grandparent of X and then around the parent of X.
   - Case 3: One of X and Parent(X) is a left child and the other is a right child (Weiss calls this zig-zag).
     - Perform a double rotation around the grandparent of X.
Analysis of Splay Trees

• Why does this work? Why can we say that operations on a splay tree take \( O(\log n) \) on average? (starting from an empty tree)

• (Reference Weiss 11.5.)

• Some definitions:
  
  • The size of node \( X \), denoted \( S(X) \), is the number of descendants of \( X \) (including \( X \) itself).
  
  • The rank of node \( X \), denoted \( R(X) \), is the log of the size of \( X \).

\[
R(X) = \log S(X)
\]

• Some notation:
  
  • Let \( X \) be the node we are accessing. Then \( P \) is the parent of \( X \), and \( G \) is the grandparent of \( X \).

• Notice: The time to access any node \( X \) is equal to 1 (to get to the new root after the splay) plus the number of rotations it took to splay the tree. (We count the zig-zig and the zig-zag as two rotations, and the zig as one.) This is proportional to the number of nodes on the path from the root to the node \( X \).

• Notice: A rotation can change the heights of many nodes in the tree, but only \( X \), \( P \) and \( G \) can have their ranks changed.
Amortized Analysis of Splay Trees (cont’d)

- We’re going to need the following Lemma:
  
  \[
  \text{if } a + b \leq c, \\
  \text{and } a \text{ and } b \text{ are both positive integers, then }
  \]
  
  \[
  \log a + \log b \leq 2 \log c - 2
  \]

- Proof:
  
  - By the arithmetic-geometric mean inequality
    \[
    \sqrt{ab} \leq \frac{a + b}{2}
    \]
  
  - It follows from the assumption that
    \[
    \sqrt{ab} \leq \frac{c}{2}
    \]
  
  - Squaring both sides gives
    \[
    ab \leq \frac{c^2}{2}
    \]
  
  - Taking the logarithm of both sides:
    \[
    \log a + \log b \leq 2 \log c - 2
    \]
Amortized Analysis of Splay Trees (cont’d)

- Theorem: The amortized time to splay a tree with root T at node X is at most $3(R(T) - R(X)) + 1 = O(\log N)$.

- There are four cases to prove:
  - X is the root of the tree.
  - P is the root of the tree (the zig step).
  - the zig-zag step
  - the zig-zig step

- If X is the root of the tree, there are no rotations. The actual time to access the node is 1 - therefore the amortized time is also 1.

- Before we show the other three cases, here’s a bit more notation:
  - Let $R_\alpha(X)$ and $S_\alpha(X)$ be the rank and size of X before the splay step, and $R_\beta(X)$ and $S_\beta(X)$ be the rank of size of X after the splay step.
The zig step...

- Observe: $S_\alpha(P) \geq S_\beta(P)$, therefore $R_\alpha(P) \geq R_\beta(P)$

- For the zig step, the actual time is 1 (for the single rotation), and the potential change is $R_\beta(X) + R_\beta(P) - R_\alpha(X) - R_\alpha(P)$, so $AT_{zig}$ (actual time of the zig step) is

  $AT_{zig} = 1 + R_\beta(X) + R_\beta(P) - R_\alpha(X) - R_\alpha(P)$

- Since $R_\alpha(P) \geq R_\beta(P)$

  $AT_{zig} \leq 1 + R_\beta(X) - R_\alpha(X)$

- Since $S_\beta(X) \geq S_\alpha(X)$, $R_\beta(X) - R_\alpha(X) \geq 0$, we can say

  $AT_{zig} \leq 1 + 3(R_\beta(X) - R_\alpha(X))$

- We’ll use this later...
The zig-zag step...

• Notice four things...:

  (1) \( S_\beta(X) = S_\beta(G) \), therefore \( R_\beta(X) = R_\beta(G) \).
  
  (2) \( S_\alpha(P) \geq S_\alpha(X) \), so \( R_\alpha(P) \geq R_\alpha(X) \).
  
  (3) \( S_\beta(P) + S_\beta(G) \leq S_\beta(X) \).
  
  (4) \( R_\beta(X) \geq R_\alpha(X) \)

• For the zig-zag step, the actual time is 2 (for the double rotation), and the potential change is \( R_\beta(X) + R_\beta(P) + R_\beta(G) - R_\alpha(X) - R_\alpha(P) - R_\alpha(G) \). From observation (1), we can cancel out two terms, so \( \text{AT}_{\text{zig-zag}} \) is

\[
\text{AT}_{\text{zig-zag}} = 2 + R_\beta(P) + R_\beta(G) - R_\alpha(X) - R_\alpha(P)
\]

• Since \( R_\alpha(X) \leq R_\alpha(P) \) (observation (2))

\[
\text{AT}_{\text{zig-zag}} \leq 2 + R_\beta(P) + R_\beta(G) - 2R_\alpha(X)
\]

• Applying the lemma above to observation (3)...
\[
\log S_\beta(P) + \log S_\beta(G) \leq 2 \log S_\beta(X) - 2
\]

- By the definition of rank this becomes:
  \[
  R_\beta(P) + R_\beta(G) \leq 2R_\beta(X) - 2
  \]

- Substituting in we get:
  \[
  AT_{zig-zag} \leq 2R_\beta(X) - 2R_\alpha(X)
  \]
  \[
  AT_{zig-zag} \leq 2(R_\beta(X) - R_\alpha(X))
  \]

- Since \( R_\beta(X) \geq R_\alpha(X) \)
  \[
  AT_{zig-zag} \leq 3(R_\beta(X) - R_\alpha(X))
  \]

We’ll use this later...
The zig-zig step...

- Observe four things...:

  1. $S_\beta(X) = S_\alpha(G)$, therefore $R_\beta(X) = R_\alpha(G)$.
  2. $S_\beta(X) \geq S_\beta(P)$, so $R_\beta(X) \geq R_\beta(P)$.
  3. $S_\alpha(X) + S_\beta(G) \leq S_\beta(X)$.
  4. $R_\alpha(X) \leq R_\alpha(P)$

- For the zig-zig step, the actual time is 2 (for the double rotation), and the potential change is $R_\beta(X) + R_\beta(P) + R_\beta(G) - R_\alpha(X) - R_\alpha(P) - R_\alpha(G)$. From observation (1), we can cancel out two terms, so $AT_{zig-zig}$ is

\[
AT_{zig-zig} = 2 + R_\beta(P) + R_\beta(G) - R_\alpha(X) - R_\alpha(P)
\]

- Since $R_\alpha(X) \leq R_\alpha(P)$ (observation (4))

\[
AT_{zig-zig} \leq 2 + R_\beta(P) + R_\beta(G) - 2R_\alpha(X)
\]

- Applying the lemma above to observation (3)...
\[
\log S_\alpha(X) + \log S_\beta(G) \leq 2 \log S_\beta(X) - 2
\]

- By the definition of rank this becomes:
  \[
  R_\alpha(X) + R_\beta(G) \leq 2R_\beta(X) - 2
  \]
  \[
  R_\beta(G) \leq 2R_\beta(X) - 2 - R_\alpha(X)
  \]

- Substituting in we get:
  \[
  AT_{zig-zig} \leq R_\beta(P) + 2R_\beta(X) - 2R_\alpha(X)
  \]

- Since \(R_\beta(X) \geq R_\alpha(P)\) (from observation (2))
  \[
  AT_{zig-zig} \leq 3(R_\beta(X) - R_\alpha(X))
  \]
In Summary...

• \( AT_{\text{zig}} \leq 1 + 3(R_\beta(X) - R_\alpha(X)) \)
• \( AT_{\text{zig-zag}} \leq 3(R_\beta(X) - R_\alpha(X)) \)
• \( AT_{\text{zig-zig}} \leq 3(R_\beta(X) - R_\alpha(X)) \)

• But as we move \( X \) up the tree, \( R_\beta(X) \) from one splay step becomes \( R_\alpha(X) \) for the next splay step. By adding up the amortized costs of all the rotations (at most one is a zig), the total amortized cost to splay node \( X \) is

\[
3(R_\beta(X) - R_\alpha(X)) + 1
\]

where \( R_\alpha(X) \) is the rank of \( X \) before the first splay step, and \( R_\beta(X) \) is the rank of \( X \) after the last splay step. Since the last splay step leave \( X \) as the root, the amortized bound is

\[
3(R_\beta(T) - R_\alpha(X)) + 1
\]

which is \( O(\log N) \) (from the definition of rank).