CS 361, Lecture 22

Jared Saia
University of New Mexico

Outline

• Red Black Trees
• Other Balanced Trees

Red-Black Properties

A BST is a red-black tree if it satisfies the RB-Properties

1. Every node is either red or black
2. The root is black
3. Every leaf (NIL) is black
4. If a node is red, then both its children are black
5. For each node, all paths from the node to descendant leaves contain the same number of black nodes

HW Questions

• Are there any questions on the current HW?
• Left-Rotate(x) takes a node x and "rotates" x with its right child
• Right-Rotate is the symmetric operation
• Both Left-Rotate and Right-Rotate preserve the BST Property
• We’ll use Left-Rotate and Right-Rotate in the RB-Insert procedure

Binary Search Tree Property

• Let x be a node in a binary search tree. If y is a node in the left subtree of x, then key(y) ≤ key(x). If y is a node in the right subtree of x then key(y) ≥ key(x)
In-Class Exercise

Show that Left-Rotate(x) maintains the BST Property. In other words, show that if the BST Property was true for the tree before the Left-Rotate(x) operation, then it's true for the tree after the operation.

- Show that after rotation, the BST property holds for the entire subtree rooted at x
- Show that after rotation, the BST property holds for the subtree rooted at y
- Now argue that after rotation, the BST property holds for the entire tree

RB-Insert(T,z)

1. Set left(z) and right(z) to be NIL
2. Let y be the last node processed during a search for z in T
3. Insert z as the appropriate child of y (left child if key(z) ≤ y, right child otherwise)
4. Color z red
5. Call the procedure RB-Insert-Fixup

RB-Insert-Fixup(T,z)

while (color(p(z)) is red) {
  case 1: z’s uncle, y, is red{
    do case 1
  }
  case 2: z’s uncle, y, is black and z is a right child{
    do case 2
  }
  case 3: z’s uncle, y, is black and z is a left child{
    do case 3
  }
}
color(root(T)) = black;

Case 1
### Loop Invariant

At the start of each iteration of the loop:

- Node \( z \) is red
- If parent(\( z \)) is the root, then parent(\( z \)) is black
- If there is a violation of the red-black properties, there is at most one violation, and it is either property 2 or 4. If there is a violation of property 2, it occurs because \( z \) is the root and is red. If there is a violation of property 4, it occurs because both \( z \) and parent(\( z \)) are red.

### Case 2 and 3

![Diagram](image)

### Pseudocode

- Detailed Pseudocode for RB-Insert and RB-Insert-Fixup is in the book, Chapter 13.3
- A detailedproof of correctness for RB-Insert-Fixup in the same Chapter
- Code for \textit{RB-Deletion} is also in Chapter 13

### Other Balanced BSTs

- We'll now \textit{briefly} discuss some other balanced BSTs
- They all implement Insert, Delete, Lookup, Successor, Predecessor, Maximum and Minimum efficiently
AVL Trees

- An AVL tree is height-balanced: For each node \( x \), the heights of the left and right subtrees of \( x \) differ by at most 1
- Each node has an additional height field \( h(x) \)
- Claim: An AVL tree with \( n \) nodes has height \( O(\log n) \)

AVL Trees

- So we have the equation \( n > T(h) \). Let \( \phi = \frac{1 + \sqrt{5}}{2} \). Then:
  \[
  n \geq \frac{1}{\sqrt{5}} (\phi^h - 2)
  \] (1)
  \[
  \log n \geq \log(\frac{1}{\sqrt{5}}) + h \log \phi - 1
  \] (2)
  \[
  \log n - \log(\frac{1}{\sqrt{5}}) + 1 \geq h \log \phi
  \] (3)
  \[
  C \log n \geq h
  \] (4)
- Where the final inequality holds for appropriate constant \( C \), and for \( n \) large enough. The final inequality implies that \( h = O(\log n) \)

AVL Trees

- Claim: An AVL tree with \( n \) nodes has height \( O(\log n) \)
- Q: For an AVL tree of height \( h \), how many nodes must it have in it?
- A: We can write a recurrence relation. Let \( T(h) \) be the minimum number of nodes in a tree of height \( h \)
- Then \( T(h) = T(h-1) + T(h-2) + 1 \), \( T(2) = T(1) \geq 1 \)
- This is similar to the recurrence relation for Fibonacci numbers! Solution:
  \[
  T(h) = \frac{1}{\sqrt{5}} \left( \frac{1 + \sqrt{5}}{2} \right)^h - 2
  \]

AVL Trees

- After insert into an AVL tree, the tree may no longer be height-balanced
- Need to “fix-up” the subtrees so that they become height-balanced again
- Can do this using rotations (similar to case for RB-Trees)
- Similar story for deletions
B-Trees

- B-Trees are balanced search trees designed to work well on disks
- B-Trees are not binary trees: each node can have many children
- Each node of a B-Tree contains several keys, not just one
- When doing searches, we decide which child link to follow by
  finding the correct interval of our search key in the key set
  of the current node.

Disk Accesses

- Consider any search tree
- The number of disk accesses per search will dominate the run time
- Unless the entire tree is in memory, there will usually be a
  disk access every time an arbitrary node is examined
- The number of disk accesses for most operations on a B-tree
  is proportional to the height of the B-tree
- I.e. The info on each node of a B-tree can be stored in main
  memory

B-Tree Properties

- All leaves have the same depth
- Lower and upper bounds on the number of keys a node can
  contain, given as a function of a fixed integer t:
  - Every node other than the root must have $\geq (t-1)$ keys,
    and t children. If the tree is non-empty, the root must
    have at least one key (and 2 children)
  - Every node can contain at most $2t-1$ keys, so any internal
    node can have at most $2t$ children
Note

- The above properties imply that the height of a B-tree is no more than \( \log_t \frac{n+1}{2} \), for \( t \geq 2 \), where \( n \) is the number of keys.
- If we make \( t \), larger, we can save a larger (constant) fraction over RB-trees in the number of nodes examined.
- A (2-3-4)-tree is just a B-tree with \( t = 2 \).

In-Class Exercise

We will now show that for any B-Tree with height \( h \) and \( n \) keys, 
\[ h \leq \log_t \frac{n+1}{2}, \text{ where } t \geq 2. \]

Consider a B-Tree of height \( h > 1 \).

- Q1: What is the minimum number of nodes at depth 1, 2, and 3?
- Q2: What is the minimum number of nodes at depth \( i \)?
- Q3: Now give a lowerbound for the total number of keys (e.g. \( n \geq ???)
- Q4: Show how to solve for \( h \) in this inequality to get an upperbound on \( h \).

Splay Trees

- A Splay Tree is a kind of BST where the standard operations run in \( O(\log n) \) amortized time.
- This means that over \( l \) operations (e.g. Insert, Lookup, Delete, etc), where \( l \) is sufficiently large, the total cost is \( O(l \cdot \log n) \).
- In other words, the average cost per operation is \( O(\log n) \).
- However a single operation could still take \( O(n) \) time.
- In practice, they are very fast.

Skip Lists

- Technically, not a BST, but they implement all of the same operations.
- Very elegant randomized data structure, simple to code but analysis is subtle.
- They guarantee that, with high probability, all the major operations take \( O(\log n) \) time.
- We'll discuss them more next class.
High Level Analysis

Comparison of various BSTs

- RB-Trees: + guarantee $O(\log n)$ time for each operation, easy to augment, – high constants
- AVL-Trees: + guarantee $O(\log n)$ time for each operation, – high constants
- B-Trees: + works well for trees that won’t fit in memory, – inserts and deletes are more complicated
- Splay Trees: + small constants, – amortized guarantees only
- Skip Lists: + easy to implement, – runtime guarantees are probabilistic only

Which Data Structure to use?

- Splay trees work very well in practice, the “hidden constants” are small
- Unfortunately, they can not guarantee that every operation takes $O(\log n)$
- When this guarantee is required, B-Trees are best when the entire tree will not be stored in memory
- If the entire tree will be stored in memory, RB-Trees, AVL-Trees, and Skip Lists are good

Skip List

- Technically, not a BST, but they implement all of the same operations
- Very elegant randomized data structure, simple to code but analysis is subtle
- They guarantee that, with high probability, all the major operations take $O(\log n)$ time

Skip List

- A skip list is basically a collection of doubly-linked lists, $L_1, L_2, \ldots, L_x$, for some integer $x$
- Each list has a special head and tail node, the keys of these nodes are assumed to be $-\text{MAXINT}$ and $+\text{MAXINT}$ respectively
- The keys in each list are in sorted order (non-decreasing)
Skip List

- Every key is in the list \( L_1 \).
- For all \( i > 2 \), if a key \( x \) is in the list \( L_i \), it is also in \( L_{i-1} \). Further there are up and down pointers between the \( x \) in \( L_i \) and the \( x \) in \( L_{i-1} \).
- All the head(tail) nodes from neighboring lists are inter-connected

Example

```
Search(k){
pLeft = L_x.head;
for (i=x;i>=0;i--){
    Search from pLeft in L_i to get the rightmost elem, r, with value <= k;
    pLeft = pointer to r in L_(i-1);
}
if (pLeft==k)
    return pLeft
else
    return nil
}
```