CSE 241 Class 16

Jeremy Buhler

October 26, 2015

Today: B-Trees Part Deux

1 B-Tree Search

Finding a key in a B-tree is easy

- Start at root
- If current node contains desired key, return it.
- Otherwise, determine which subtree would have key and recur on it
- Looks at only $O(h)$ nodes

Try it on example tree (find H, S, and A)

2 B-Tree Insert and Splitting

Insertion and deletion in a B-tree are interesting because we must maintain the min- and max-degree invariants.

- What's natural **insert**(k)?
- Find leaf where k belongs and put it there
- What's wrong with simple algorithm? [**wait**]
- Leaf may already be full ($2t - 1$ keys) – adding another would violate max-degree invariant.

We can try to fix insertion by **splitting**. Splitting turns a full node into two non-full nodes.

```
Split(x)
    k ← k_t(x)                                        ▷ median key
    create node x_ℓ from keys k_1(x) ... k_{t-1}(x)
    create node x_r from keys k_{t+1}(x) ... k_{2t-1}(x)
    move k into parent of x
    place pointers to x_ℓ and x_r to left and right of k
```

Example of splitting:

Can we always split a node x?

- What if x's parent is full?
- Would be nowhere to put median key of x! So, let's ensure this bad case does not happen
- What if x is the root?
- Can create a new root x of size 1 to hold x's median key
- (B-trees grow *up* from the root!)
- How do we find x's parent? (it has no parent pointer)
- Will assume parent is cached at time of split (OK for insert, delete below)

3 Insertion Algorithm

- To avoid complications, want to visit each node on path to insertion point only once.
- Implies only one disk read per node on path, or $O(h)$ total.
- We split *preemptively* to avoid backtracking.
- Algorithm uses recursive subroutine DoInsert(x, k)
- Will maintain following invariant (*):

> When we call DoInsert(x, k), either x is the root, or x's parent is not full.

```
INSERT(T, k)
   DOINSERT(root[T], k)

DOINSERT(x, k)
   if x is full
      m ← k_t(x)                         ▷ median key of x
      SPLIT(x)                              ▷ create x_ℓ, x_r
      if k < m
         x ← x_ℓ
      else
         x ← x_r

   if leaf(x)
      place k into x
   else
      y ← correct child of x
      DOINSERT(y, k)
```

Examples? [work from the sheet]
Correctness? Argue inductively that *split never fails*. Conclude that k can be inserted at end of algorithm because we can create a non-full node if needed.

- Prove by induction on number of calls to DoInsert.
- **Base**: Invariant (*) holds at first call to DoInsert, since call is made on root node.
- **Ind**: Suppose invariant (*) holds after m calls; show that it will hold after $m + 1$.
- Invariant (*) holds at start of DoInsert, so split will succeed if it happens (always room for median in parent, or new root created).
- If we call DoInsert(y, k), y's parent has been split if it was full, so invariant (*) is maintained.
- When we try to insert k into x, x has just been split if it was full. Hence, x is not full, and insert succeeds. QED

4 Deletion

B-tree deletion is cute but difficult to code. We'll only sketch it here.

- Two problems with removing an arbitrary key k from tree.
- First, what happens to subtrees to left and right of k?
- Second, k's node might have only $t - 1$ keys – removal could violate min-degree invariant.
- As for insert, will have a recursive DoDelete(x, k). Initially call DoDelete(root[T], k).
- To keep balance, will maintain following invariant (**):

When we call DoDelete(x, k), either x is the root, or x contains at least t keys.

- Invariant (**) implies that when we do remove k from some node, it will be the root or will still have at least $t-1$ keys after the deletion.

Assume invariant (**) is true when DoDelete(x, k) is called. Must consider three cases:

1. If x is a leaf ...
   - Simply remove k from x.
   - x has no children, so no subtrees to deal with.
   - Invariant (**) guarantees that x has at least t keys before deleting k.
2. If x contains k (and is not a leaf) ...
   - Let y and z be left and right child nodes of k in x.
   - **(a)** If y has at least t keys, replace k with **pred**(k) (largest key in subtree rooted at y).
   - Now, must recursively remove **pred**(k) – call DoDelete(y, **pred**(k))
   - **(b)** Else if z has at least t keys, replace k with **succ**(k) and remove **succ**(k) from subtree rooted at z.
   - **(c)** Otherwise, both y and z have exactly $t-1$ keys.
   - Hence, can **unsplit** y, z to form a new node w!
   - k becomes median key of w (OK to remove k from x because by invaiant (**), x has at least t keys).

- Now recursively DoDelete(w, k)

3. If x does not contain k (and is not a leaf) …

- Want to delete k from appropriate subtree of x, rooted at some node z.
- **(a)** If z has at least t keys, just call DoDelete(z, k)
- If z has only $t-1$ keys, how do we maintain invariant (**)?
- **(b)** If z's left neighbor y has at least t keys, steal its rightmost key (by rotation), then call DoDelete(z, k)
- Else if z's right neighbor has at least t keys, steal its leftmost key (by rotation), then call DoDelete(z, k)
- **(c)** Else both z and some neighbor y have exactly $t-1$ keys.
- Hence, unsplit z and y into a new node w and call DoDelete(w, k)