11 Balanced Trees

Abstract

In this chapter we describe two programming projects that involve trees. The projects in question are improvements of the classes mentioned earlier in the text: the priority queue (pg. 433 Proj. 5) and the set class (similar to the bag class). We will also analyze the time performance of tree algorithms.

B-Trees

Section 10.5 implements the bag class with binary search trees, however their efficiency can sometimes go awry. We will analyze the problem and show one way to fix it.

The Problem of Unbalanced Trees

Let's assume that we have a binary tree that's left-aligned and populated with the values 1 through 5. This is troublesome since this implementation is essentially no better than a linked list; continue adding values to the left branch of the tree and the problem exacerbates. We can solve this problem in several ways, for example we could periodically balance the search trees (explained in Ch.10 pg.538). Another method is to use a B-Trees, where the leaves cannot become deep (relative to the number of nodes).

The B-Tree Rules

Every B-Tree depends on a positive constant integer $MINIMUM$ which serves as the number of entries held in a single node.

- The root may have as few as one entries while every other node has at least $MINIMUM$ entires.
- The maximum number of entries in a node is twice the value of $MINIMUM$.
- The entries of each B-Tree node are stored in a partially filled array, sorted from smallest to largest.
- The number of subtrees n below a nonleaf node is always $n = m + 1$ where m is the number of entires in the node.

Example 11.1. Consider a node that has 42 entries. This node will have 43 children according to our prevoius statement. Each subtree will be referred to as subtree 0, subtree 1,$\cdots$, subtree 42. $\diamond$

- For any nonleaf node the subtrees are ordered from least to greatest value

Example 11.2. Consider a nonleaf node that contains two integer entries 93 and 107. This nonleaf node must have three child nodes based upon example 11.1. subtree0 must be less than the value 93, subtree1 must have a value between 93 and 107, and subtree2 must be greater than 107. ◇

– Every leaf in a B-Tree has the same depth.

The Set ADT with B-Trees

The rest of this section discusses methods for implementing B-Trees. I will refer to the text as much as possible to avoid typing out huge chunks of code.

Invariant for the Set Class Implemented with a B-Tree

1. The items of the set are stored in a B-Tree that comply with the aforementioned rules.
2. The number of entries in the tree's root are stored in hte member variable data_count. The number of subtrees of the root is stored in hte member variable child_count.
3. The root's entries are stored in data[0] through data[data_count-1].
4. If the root has subtrees, then these subtrees are stored in sets pointed to by the pointers subset[0] through subset[child_count-1].

Searching for an Item in a B-Tree

The set class (defined on pg. 551) has a member function (count) htat determines whether an item called *target* appears in the set. Remember that count always returns 0 if *target* is not found, and 1 if it is.

We search B-Trees in much the same way we search binary search trees. The basic algorithm is as follows:

– Check the root, if $count = 1$ then we're done!

– If the root has no children, the search is also done.

– Make a recursive call to the children to check for *target*

We justify the recursive call by arguing that the subset of the root *has* to be smaller than the entire set.

Inserting an Item into a B-Tree

The *insert* member function defined in the set class adds a new item to the B-Tree. We'll attempt to implement a loose insertion which *may* result in an illegal tree.

The Loose Insertion into a B-Tree

Our initial condition is that the B-Tree is valid. Our postconditions are as follows:

- If the value is already set, leave the B-Tree unchanged.
- Otherwise add the entry to the set. The B-Tree remains valid, however it's possible that the number of entries in the root of the set may be higher than $MAXIMUM$.

So a basic algorithm for this insertion seems fairly straightforward, and begins much like the search algorithm: find the location of the root's entires that is not less than the new entry.

- Set local variable i equal to the first index, such that data[i] is less than the entry.
- If we find the value of i at data[i] return; there's nothing to do!
- If the root has no children, add a new entry to the root.
- Otherwise, save the value from the recursive call subset[i]→loose_insert(entry);

A Private Member Function to Fix an Excess in a Child

To fix a child with $MAXIMUM + 1$ entries, the child node is split into two nodes that each contain $MINIMUM$ entries. This leaves one extra entry, which is passed upward to the parent.

Back to the Insert Member Function

By using the loose insertion function defined above, we have only two steps to ensure a smooth insertion.

1. If loose insert returns false, we have nothing to do.
2. Otherwise, if the maximum value has been exceeded for the root, fix the tree with out fix_excess function.

The last step can be performed in two parts. We first copy all the pointers of the child nodes and clear the root, essentially bumping the whole tree down one node. We've changed the scope of the problem to our old root, and this is much simpler to fix. We split the old root node into a subset and shift the middle value up to our new root. Refer to page 562 for a pictoral representation of this shift.

Employing top Down Design

Essentially, top down design means we implement functions in such a way that each successive function call solves a smaller and smaller problem. Through this methodology it becomes easier to determine points of failure.

Removing an Item from a B-Tree