CSE 332: Data Structures and Parallelism

Section 4: Balanced Trees Solutions

0. The ABC's of AVL Trees

What are the constraints on the data types you can store in an AVL tree? When is an AVL tree preferred over another dictionary implementation, such as a HashMap?

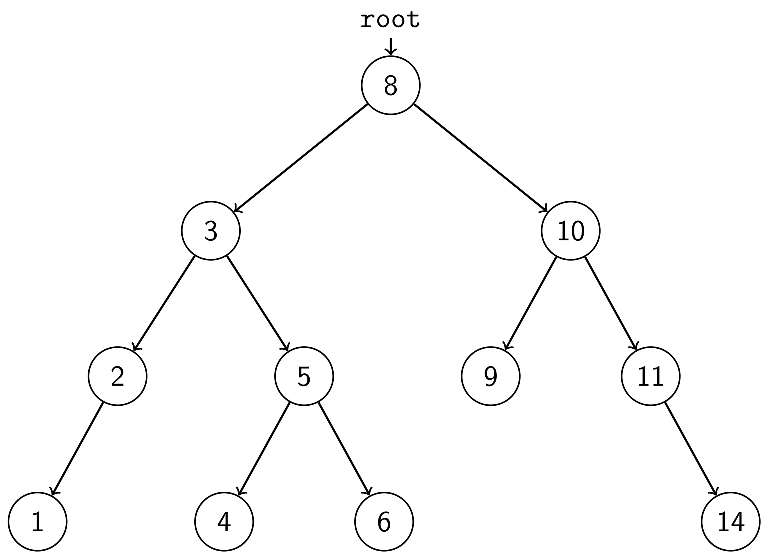
Solution:

AVL trees are similar to TreeMaps. They require that keys be orderable, though not necessarily hashable. The value type can be anything, just like any other dictionary.
A perk over HashMaps is that keys are stored and can be iterated over in sorted order.

1. Let's Plant an AVL Tree.

Insert 10, 4, 5, 8, 9, 6, 11, 3, 2, 1, 14 into an initially empty AVL Tree.

Solution:



2. The ABC's of B-Trees

(a) What properties must a B-tree of n values have with given values for M and L?

Solution:

(a) B-Tree order property:

i. Every subtree between keys a and b contains all data x where $a < x \leq b$

(b) B-Tree structure property:

i. If $n \leq L$, the root is a leaf with n values, otherwise the root is an internal node that must have between 2 and M children

ii. All internal nodes must have between $\left\lceil \frac{M}{2} \right\rceil$ and M children (i.e., half-full)

iii. All leaf nodes must have between $\left\lceil \frac{L}{2} \right\rceil$ and L key-value pairs (i.e., half-full)

(b) Give an example of a situation that would be a good job for a B-tree. Furthermore, are there any constraints on the data that B-trees can store?

Solution:

(a) B-trees are most appropriate for very, very large data stores, like databases, where the majority of the data lives on disk and cannot possibly fit into RAM all at once.

B-trees require orderable keys. B-trees are typically not implemented in Java because because what makes them worthwhile is their precise management of memory.

3. Implement a B-Tree? Nah, Let's Analyze!

Given the following parameters for a B-Tree with a page size of 256 bytes:

- Key Size = 8 bytes
- Pointer Size = 2 bytes
- Data Size = 14 bytes per record (includes the key)

Assuming that M and L were chosen appropriately, what are M and L? Recall that M is defined as the maximum number of pointers in an internal node, and L is defined as the maximum number of values in a leaf node. Give a numeric answer and a short justification based on two equations using the parameter values above.

Solution:

We start by defining the following variables.

- 1 page on disk is p bytes
- Keys are k bytes
- Pointers are t bytes
- Key/Value pairs are v bytes

We know that the amount of memory used by one leaf node is vL and the amount of memory used by one internal node is $tM + k(M - 1)$. We want select values for M and L such that both equations are $\leq p$.

If we solve both equations for M and L, we obtain $M = \left\lfloor \frac{p+k}{t+k} \right\rfloor$ and $L = \left\lfloor \frac{p}{v} \right\rfloor$

Plugging in the given values, we get $M = \left\lfloor \frac{256+8}{2+8} \right\rfloor = 256$ and $L = \left\lfloor \frac{256}{14} \right\rfloor = 18$

4. Oh, B-Trees

Find a tight upper bound on the *worst case runtime* of these operations on a B-tree. Your answers should be in terms of L, M, and n.

(a) Insert a key-value pair

(b) Look up the value of a key

(c) Delete a key-value pair

Solution:

Insertion, Deletion The steps for insert and delete are similar and have the same worst case runtime.

(a) Find the leaf: $\mathcal{O}(\lg(M)\log_M(n))$. (For more details, see the next solution.)

(b) Insert/remove in the leaf – there are L elements, essentially stored in an array: $\mathcal{O}(L)$

(c) Split a leaf/merge neighbors: $\mathcal{O}(L)$

(d) Split/merge parents, in the worst case going up to the root: $\mathcal{O}(M\log_M(n))$

The total cost is then $\lg(M)\log_M(n) + 2L + M\log_M(n)$.

We can simplify this to a worst-case runtime $\mathcal{O}(L + M\log_M(n))$ by combining constants and observing that $M\log_M(n)$ dominates $\lg(M)\log_M(n)$. Note that in the average case, splits for any reasonably-sized B-tree are rare, so we can amortize the work of splitting over many operations.

However, if we're using a B-tree, it's because what concerns us the most is the penalty of disk accesses. In that case, we might find it more useful to look at the worst-case number of disk lookup operations in the B-tree, which is $\mathcal{O}(\log_M(n))$.

Look up (a) We must do a binary search on a node containing M pointers, which takes $\mathcal{O}(\lg(M))$ time, once at each level of the tree.

(b) There are $\mathcal{O}(\log_M(n))$ levels.

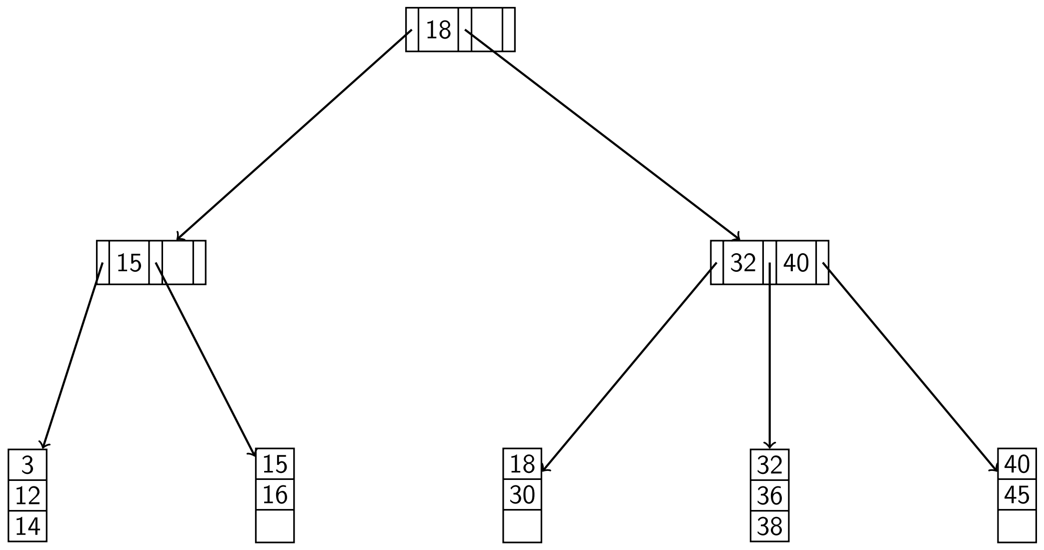
(c) We must do a binary search on a leaf of L elements, which takes $\mathcal{O}(\lg(L))$ time.

(d) Putting it all together, a tight bound on the runtime is $\mathcal{O}(\lg(M)\log_M(n) + \lg(L))$.

5. It's Fun to B-Trees!

(a) Insert the following into an empty B-Tree with $M = 3$ and $L = 3$: $3, 18, 14, 30, 32, 36, 15, 16, 12, 40, 45, 38$.

Solution:



(b) Delete $45, 14, 15, 36, 32, 18, 38, 40, 12$

Solution: