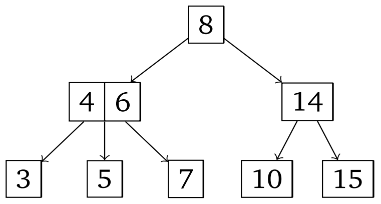
Section 04: Solutions

1. B-Trees

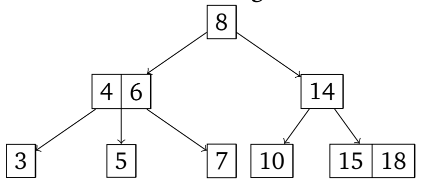
(a) Draw what the following 2-3 tree would look like after inserting 18, 38, 12, 13, and 20.



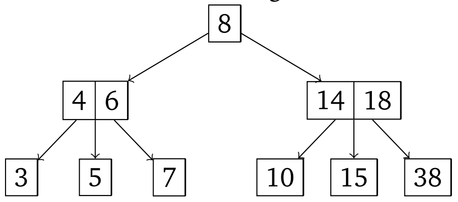
Solution:

The tree is redrawn after each insertion, but only the final tree matters in this case. Remember that doing something like this to show your work can help you earn partial credit in an exam setting!

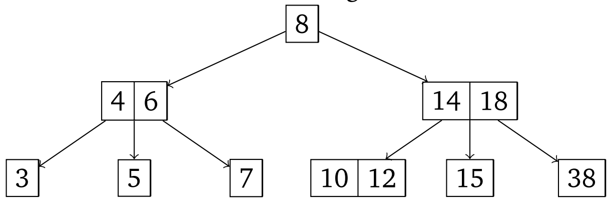
After inserting 18...



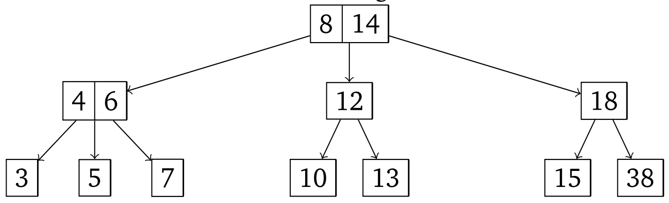
After inserting 38...



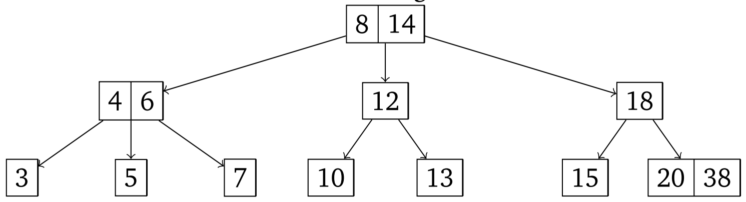
After inserting 12...



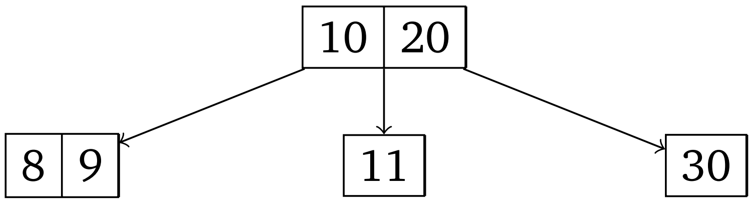
After inserting 13...



After inserting 20...

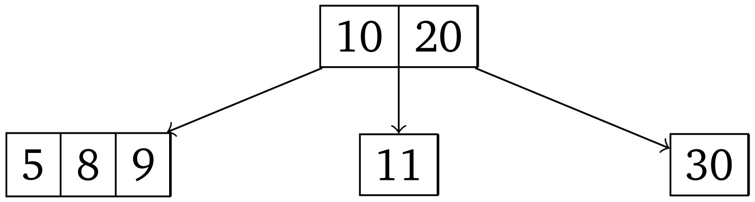


(b) Given the following initial 2-3-4 tree, draw the result of performing each operation.



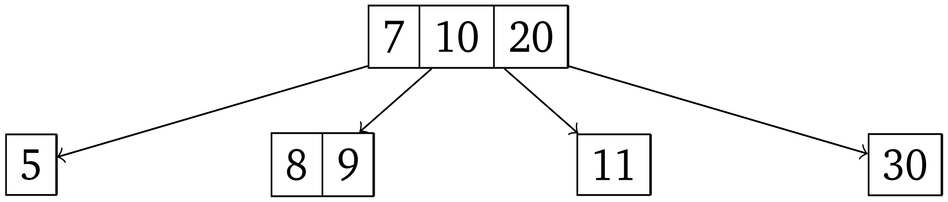
(i) Insert 5 into this tree.

Solution:



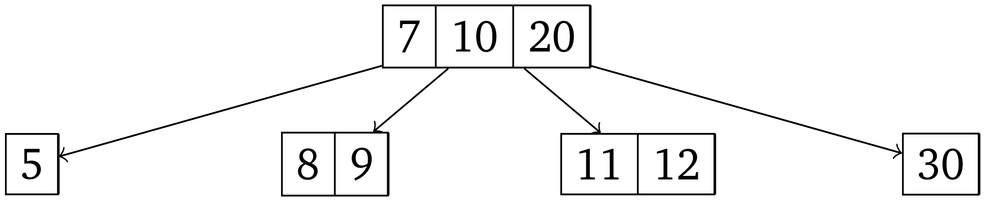
(ii) Insert 7 into the resulting tree.

Solution:



(iii) Insert 12 into the resulting tree.

Solution:



(c) Suppose the keys 1, 2, 3, 4, 5, 6, 7, 8, 9, and 10 are inserted sequentially into an initially empty 2-3-4 tree. Which insertions cause a split to take place?

Solution:

4, 6, 8, 10

2. Big-Θ: Red-Black Trees and BSTs

What is the worst-case big-Θ bound for runtime for each of the following?

(a) Insert and find in a BST.

Solution:

$\Theta(n)$ and $\Theta(n)$, respectively. This is unintuitive, since we commonly say that `find()` in a BST is "`log(n)`", but we're asking you to think about *worst-case* situations. The worst-case situation for a BST is that the tree is a linked list, which causes `find()` to reach $\Theta(n)$.

(b) Insert and find in a Red-Black tree.

Solution:

$\Theta(\log(n))$ and $\Theta(\log(n))$, respectively. Red-Black trees are guaranteed to be balanced, so the worst case is we need to traverse the height of the tree.

(c) Finding the minimum value in a Red-Black tree containing n elements.

Solution:

$\Theta(\log(n))$. We can find the minimum value by starting from the root and constantly traveling down the left-most branch. As in the previous problem, Red-Black trees are guaranteed to be balanced, so the worst case is we need to traverse the height of the tree.

(d) Finding the k-th largest item in a Red-Black tree containing n elements.

Solution:

With a standard Red-Black tree implementation, it would take $\Theta(n)$ time. If we're located at a node, we have no idea how many elements are located on the left vs right and need to do a traversal to find out. We end up potentially needing to visit every node.

(e) Listing elements of a Red-Black tree in sorted order

Solution:

$\Theta(n)$. We can just perform an in order traversal of the tree once, listing the contents of each node seen in the traversal.

3. Analyzing dictionaries

(a) What are the constraints on the data types you can store in a Red-Black tree? When is a Red-Black tree preferred over another dictionary implementation, such as a HashMap?

Solution:

Red-Black 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 with Red-Black trees, you can iterate over the keys in sorted order. Red-Black trees have better worst case bounds for insert, delete, and remove, since they are always balanced; while hash tables can have $\mathcal{O}(n)$ operations in the worst case. When the hash function works well, though, hash tables are more efficient ($\mathcal{O}(1)$ operations).

(b) When is using a BST preferred over a Red-Black tree?

Solution:

One of Red-Black's advantages over BST is that it has an asymptotically efficient `find()` even in the worst-case.

However, if you know that `find()` won't be called frequently on the BST, or if you know the keys you receive are sufficiently random enough that the BST will stay balanced, you may prefer a BST since it would be easier to implement. Since you also don't need to worry about performing rotations and keeping track of red/black links, using a BST could be a constant factor faster compared to using a Red-Black tree.

4. Design

Imagine a database containing information about all trains leaving the Washington Union station on Monday. Each train is assigned a departure time, a destination, and a unique 8-digit train ID number.

What data structures would you use to solve each of the following scenarios? Depending on scenario, you may need to either (a) use multiple data structures or (b) modify the implementation of some data structure.

Justify your choice.

(a) Suppose the schedule contains 200 trains with 52 destinations. You want to easily list out the trains by destination.

Solution:

One solution would be to use a dictionary where the keys are the destination, and the value is a list of corresponding chains. Any dictionary implementation would work.

Alternatively, we could modify a separate chaining hash table so it has a capacity of exactly 52 and does not perform resizing. If we then make sure each destination maps to a unique integer from 0 to 51 and hash each train based on this number, we could fill the table and simply print out the contents of each bucket.

A third solution would be to use a BST or Red-Black tree and have each train be compared first based on destination then based on their other attributes. Once we insert the trains into a tree, we can print out the trains sorted by destination by doing an in-order traversal.

(b) In the question above, trains were listed by destination. Now, trains with the same destination should further be sorted by departure time.

Solution:

Regardless of which solution we modify, we would first need to ensure that the train objects are compared first by destination, and second by departure time.

We can modify our first solution by having the dictionary use a sorted set for the value, instead of a list. (The sorted set would be implemented using a BST or a Red-Black tree).

We can modify our second solution in a similar way by using specifically a BST or a Red-Black tree as the bucket type.

Our third solution actually requires no modification: if the trains are now compared first by destination and second by departure time, the Red-Black and BST tree's iterator will naturally print out the trains in the desired order.

(c) A train station wants to create a digital kiosk. The kiosk should be able to efficiently and frequently complete look-ups by train ID number so visitors can purchase tickets or track the location of a train. The kiosk should also be able to list out all the train IDs in ascending order, for visitors who do not know their train ID.

Note that the database of trains is not updated often, so the removal and additions of new trains happen infrequently (aside from when first populating your chosen DS with trains).

Solution:

Here, we would use a dictionary mapping the train ID to the train object.

We would want to use either a Red-Black tree or a BST, since we can list out the trains in sorted order based on the ID.

Note that while the Red-Black tree theoretically is the better solution according to asymptotic analysis, since it's guaranteed to have a worst-case runtime of $\mathcal{O}(\log(n))$, a BST would be a reasonable option to investigate as well.

big-O analysis only cares about very large values of n, since we only have 200 trains, big-O analysis might not be the right way to analyze this problem. Even if the binary search tree ends up being degenerate, searching through a linked list of only 200 element is realistically going to be a fast operation.

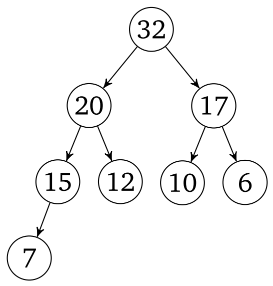
What's actually best will depend on the libraries you already have written, what hardware you actually run on, and how you want to balance code that will be sustainable if you get more trains vs. code that will be easy to understand and check for bugs right now.

5. Heap Insertions

(a) Insert the following sequence of numbers into a *max heap*:

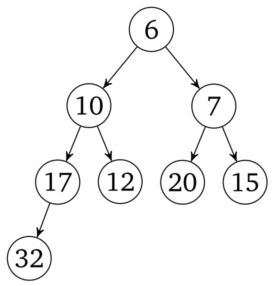
[10, 7, 15, 17, 12, 20, 6, 32]

Solution:



(b) Now, insert the same values into a *min heap*.

Solution:



6. Heaps: Sorting and Reversing

(a) Suppose you have an array representation of a heap. Must the array be sorted?

Solution:

No, $[1, 2, 5, 4, 3]$ is a valid min-heap, but it isn't sorted.

(b) Suppose you have a sorted array (in increasing order). Must it be the array representation of a valid min-heap?

Solution:

Yes! Every node appears in the array before its children, so the heap property is satisfied.

(c) You have an array representation of a min-heap. If you reverse the array, does it become an array representation of a max-heap?

Solution:

No. For example, $[1, 2, 4, 3]$ is a valid min-heap, but if reversed it won't be a valid max-heap, because the maximum element won't appear first.