Section 04: Solutions

Heaps

(a) Insert the following values into an initially empty binary max-heap: 10, 7, 3, 6, 15, 12, 20, 32

Solution:

(b) How would you represent the heap above as an array?

Solution:

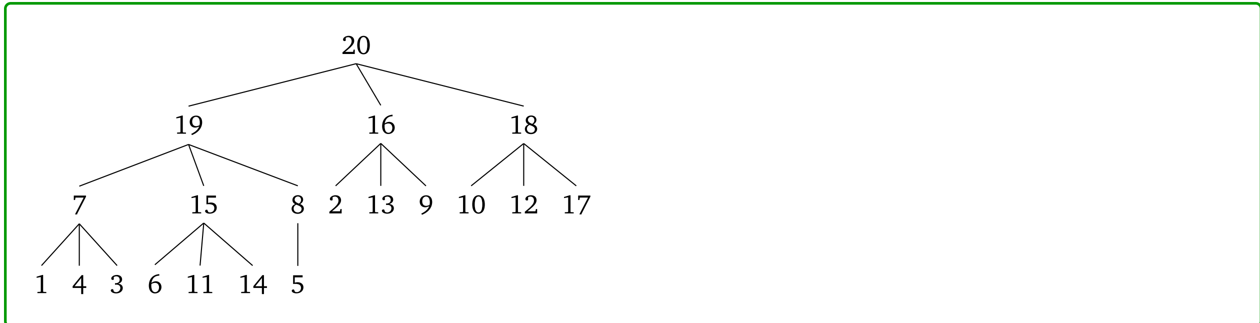
Reading the values at each level across, we get the following array.

[0, 32, 20, 15, 10, 7, 3, 12, 6]

Note that the 0th index is a placeholder to simplify the math. An answer that fills the array starting at index 0 is also acceptable.

(c) Insert the following values into an initially empty **trinary** max heap: 1, 10, 2, 15, 6, 8, 16, 17, 13, 9, 12, 18, 20, 4, 7, 3, 11, 19, 14, 5

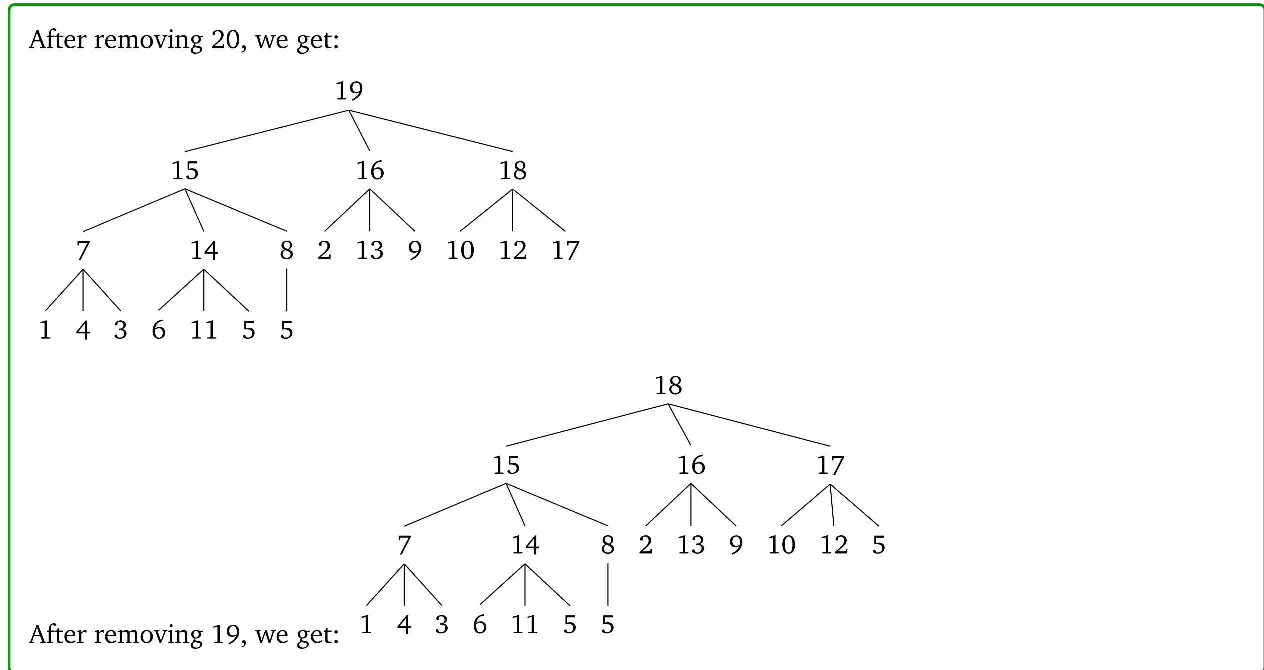
Solution:



(d) What would the heap from part c look like after calling removeMax() twice?

Solution:

After removing 20, we get:



After removing 19, we get:

(e) Consider a binary heap implementation using an array with the root at index 1. What is the index of the parent of the element at index i? What is the index of its left and right children?

Solution:

For a binary heap, the parent is at index $\lfloor \frac{i}{2} \rfloor$.

The left child can be found at index $2 * i$. The right child can be found at index $2 * i + 1$.

(f) Consider a binary heap implementation using an array with the root at index 1. Fill in the blanks for the insert method below:

```
int[] elements = ...;
int size = ...;
public static void insert(int num) {
  int k = size++;
  elements[k] = num;

  while (k > 1 && elements[_________] > elements[_________]) {
    // Swap the element with its parent
    int temp = elements[k];

    elements[k] = elements[_________];

    elements[_________] = temp;

    k = _________;
  }
}
```

Solution:

To access the parent of a node at index k, we can divide the index by 2.

```
int[] elements = ...;
int size = ...;
public static void insert(int num) {
  int k = size++;
  elements[k] = num;
  while (k > 1 && elements[k] > elements[k / 2]) {
    // Swap the element with its parent
    int temp = elements[k];
    elements[k] = elements[k / 2];
    elements[k / 2] = temp;
    k = k / 2;
  }
}
```

Big-O BST, B-Tree, Left-Leaning Red-Black (LLRB) Tree

0.1. Runtime

Write down a tight big-O for each of the following operation. Unless otherwise noted, give a bound in the worst case.

(a) Insert and find in a BST Insert : $O(_______)$ Find : $O(_______)$ One sentence Explanation:

Solution:

Insert : $O(\underline{n})$ Find : $O(\underline{n})$ One sentence Explanation: 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 O (n).

(b) Insert and find in a 2-3 Tree Insert : $O(_______)$ Find : $O(_______)$ One sentence Explanation:

Solution:

Insert : $O(\underline{log(n)})$ Find : $O(\underline{log(n)})$ One sentence Explanation: The main advantage with 2-3 trees is that it is self-balanced as opposed to a BST. Due to this, the worst case time-complexity of operations such as search and insertion is O(log(n)) as the height of a 2-3 tree is O(log(n)).

(c) Insert and find in a LLRB Tree Insert : $O(_______)$ Find : $O(_______)$ One sentence Explanation:

Solution:

Insert : $O(\underline{log(n)})$ Find : $O(\underline{log(n)})$ One sentence Explanation: Red-Black trees are guaranteed to be balanced, so the worst case is we need to traverse the height of the tree.

(d) Find the minimum value in a LLRB tree containing n elements $O(_______)$ One sentence Explanation:

Solution:

$O(_______)$ One sentence Explanation: We can find the minimum value by starting from the root and constantly traveling down the left-most branch.

(e) Find the k-th largest item in a LLRB tree containing n elements $O(_______)$ One sentence Explanation:

Solution:

$O(_______)$ One sentence Explanation: 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. In the worst case, we end up potentially needing to visit every node.

(f) List elements of LLRB tree in sorted otder $O(_______)$ One sentence Explanation:

Solution:

$O(_______)$ One sentence Explanation: We can just perform an in order traversal of the tree once, listing the contents of each node seen in the traversal.

0.2. Dictionaries Comparison

(a) What are the similarities in the constraints on the data type you can store in a B-Tree and Red-Black Tree? **Solution:**

> They require that keys be orderable and no duplicates.

(b) When is using a Red-Black Tree preferred over a BST? **Solution:**

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

(c) When is using a BST preferred over a Red-Black Tree? **Solution:**

> 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 we 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.

(d) When is a Red-Black tree preferred over another dictionary implementation, such as a HashMap? **Solution:**

> A perk over HashMaps is that with Red-Black trees, you can iterate over the keys in sorted order. Red-Black trees have a better worst case bounds for insert, delete, and remove, since they are always balanced; while hashtables can have O (n) operations in the worst case. When the hash function works well, though, hash tables are more efficient (O (1) operations).

0.3. Design Choice

(a) 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 you would 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.

(b) 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.

(c) 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.

(d) 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 $O(log(n))$, 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.

Hashing

0.4. Hash table insertion

For each problem, insert the given elements into the described hash table. Do not worry about resizing the internal array.

(a) Suppose we have a hash table that uses separate chaining and has an internal capacity of 12. Assume that each bucket is a linked list where new elements are added to the front of the list.

Insert the following elements in the EXACT order given using the hash function $h(x) = 4x$:

0, 4, 7, 1, 2, 3, 6, 11, 16

Solution:

To make the problem easier for ourselves, we first start by computing the hash values and initial indices:

key	hash	index (pre probing)
0	0	0
4	16	4
7	28	4
1	4	4
2	8	8
3	12	0
6	24	0
11	44	8
16	64	4

The state of the internal array will be

$6 \to 3 \to 0$	/	/	/	$16 \to 1 \to 7 \to 4$	/	/	/	$11 \to 2$	/	/	/

(b) Suppose we have a hash table that uses linear probing and has an internal capacity of 13.

Insert the following elements in the EXACT order given using the hash function $h(x) = 3x$:

2, 4, 6, 7, 15, 13, 19

Solution:

Again, we start by forming the table:

key	hash	index (before probing)
2	6	6
4	12	12
6	18	5
7	21	8
15	45	6
13	39	0
19	57	5

Next, we insert each element into the internal array, one-by-one using linear probing to resolve collisions. The state of the internal array will be:

13	/	/	/	/	6	2	15	7	19	/	/	4

(c) Suppose we have a hash table that uses quadratic probing and has an internal capacity of 10.

Insert the following elements in the EXACT order given using the hash function $h(x) = x$:

0, 1, 2, 5, 15, 25, 35

Solution:

The state of the internal array will be:

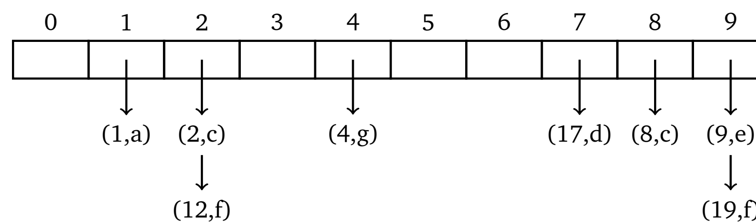
0	1	2	/	35	5	15	/	/	25

(d) Suppose we have a hash table implemented using separate chaining. This hash table has an internal capacity of 10. Its buckets are implemented using a linked list where new elements are appended to the end. Do not worry about resizing.

Show what this hash table internally looks like after inserting the following key-value pairs in the order given using the hash function $h(x) = x$:

$$(1, a), (4, b), (2, c), (17, d), (12, e), (9, e), (19, f), (4, g), (8, c), (12, f)$$

Solution:



0.5. Evaluating hash functions

Consider the following scenarios.

(a) Suppose we have a hash table with an initial capacity of 12. We resize the hash table by doubling the capacity. Suppose we insert integer keys into this table using the hash function $h(x) = 4x$.

Why is this choice of hash function and initial capacity suboptimal? How can we fix it?

Solution:

Notice that the hash function will initially always cause the keys to be hashed to at most one of three spots: 12 is evenly divided by 4.

This means that the likelihood of a key colliding with another one dramatically increases, decreasing performance.

This situation does not improve as we resize, since the hash function will continue to map to only a fourth of the available indices.

We can fix this by either picking a new hash function that's relatively prime to 12 (e.g. $h(x) = 5x$), by picking a different initial table capacity, or by resizing the table using a strategy other then doubling (such as picking the next prime that's roughly double the initial size).

(b) Suppose we have a hash table with an initial capacity of 8 using quadratic probing. We resize the hash table by doubling the capacity.

Suppose we insert the integer keys $2^{20}, 2 \cdot 2^{20}, 3 \cdot 2^{20}, 4 \cdot 2^{20}, \ldots$ using the hash function $h(x) = x$.

Describe what the runtime of the dictionary operations will over time as you keep adding these keys to the table.

Solution:

Initially, for the first few keys, the performance of the table will be fairly reasonable.

However, as we insert each key, they will keep colliding with each other: the keys will all initially mod to index 0.

This means that as we keep inserting, each key ends up colliding with every other previously inserted key, causing all of our dictionary operations to take $\mathcal{O}(n)$ time.

However, once we resize enough times, the capacity of our table will be larger then 2^{20}, which means that our keys no longer necessarily map to the same array index. The performance will suddenly improve at that cutoff point then.