Final Review

A review of all the concepts covered since the midterm

Connor Baker

2019-05-06, Compiled on May 6, 2019 at 11:14am

Final Review

Reading List Notes and Quick References

- Big-O:
  - Chapter 5: Algorithm Analysis
  - Section 5.3: You are not required to prove the theorems, but you need to understand them.
- Lists:
  - Section 6.1-6.3, 6.5: General Lists
  - Chapter 15: Dynamic Array / Array List and Iterator
  - Chapter 17: Linked List
  - Section 6.6: Stack/Queue Concepts
  - Chapter 16.1-16.3: Stack/Queue Implementation
- Trees:
  - Section 18.1: Tree Basics, Binary Trees
  - Section 7.1-7.3: Recursion
  - Section 18.3-18.4: Tree Recursions, Tree Traversals
- Hashing:
  - Section 20.1-20.2: Hashing Basics and Hash Functions
  - Section 20.5: Separate Chaining
  - Section 20.3-20.4: Open Addressing
  - Section 20.6-20.7: Comparisons and Applications
- Graphs:
  - Section 14.1: Graph Basics
  - Section 14.2-14.3: Shortest-path Problem
- Priority Queues / Heap:
  - Section 6.9: Priority Queues
  - Section 21.1-21.3: Binary Heap
  - Section 21.5: Heap Sort
- Binary Search Trees:
  - Section 19.1,19.3: Binary Search Tree Basics
  - Section 19.4: AVL Trees
  - Section 19.5: Red-black Trees
  - Section 19.8: B-Trees
- Disjoint sets:
  - Section 24.1-24.5: Union Find

Sample Questions for Review

NB: Questions before the midterm are the same as they were for the midterm review, which makes sense given that the final is cumulative. Since I published a separate midterm review, these questions are omitted here.

Hashing (Post-Midterm)

45. Compare and contrast the three types of probing we covered for open addressing. Make sure to note the benefits and disadvantages of each.

Well there aren't really *three* different types of probing – rather, there's three ways that we can handle collisions. We have separate chaining and open addressing. Open addressing in turn has two different ways of handling collisions, linear probing and quadratic probing. As a chart, we'd have

1. Separate Chaining
2. Open Addressing
    1. Linear probing
    2. Quadratic probing

Separate Chaining

The benefit of using separate chaining is that collisions just turn into insertions into some data structure (whether it be a binary tree, another `HashMap`, or something else entirely). Generally separate chaining is preferred. However, that means we need to maintain a separate data structure, have another level of indirection, and adding elements to the data structure may require resizing.

Additionally operations like `add(T t)`, `remove(T t)`, and `contains(T t)` have an upper bound of $O(\log(n))$ since we must search whatever data structure we're using for each entry in the hash table. (Using a second `HashMap` keeps the amortized cost at $O(1)$, while using a binary search tree introduces a cost of $O(\log(n))$, for example).

In class we used a linked list as the secondary data structure, which was kind of silly – sure it can grow infinitely and is an easy visual to grasp, but the efficiency just isn't there for the operations that we need (fast `add(T t)`, `remove(T t)`, and `contains(T t)`). For Big-O analysis with separate chaining and linked lists, see my notes from February 28, page 3.

Open Addressing

Linear and quadratic probing search for the next available space in a hash table instead of creating a data structure to handle collisions. As an example, if slot 5 is filled, linear probing will check the sequence $\{5+1, 5+2, 5+3, 5+4, 5+5, \dots\} = \{6, 7, 8, 9, 10, \dots\}$ and so on, while quadratic probing will check $\{5+1, 5+2^2, 5+3^2, 5+4^2, 5+5^2, \dots\} = \{6, 9, 14, 21, 30, \dots\}$.

This introduces a need to use tombstones, markers on each entry that indicate whether it has been removed or not, so that we avoid removing entries which are in search chains.

Linear probing suffers from primary clustering since it continually searches within a small neighborhood of cells. Additionally, it has a load of

$$\frac{1}{2}\left(1+\frac{1}{(1-load)^2}\right)$$

which yields a very steep exponential growth curve for the number of cells checked during an insertion, with respect to the load.

Quadratic probing still needs to use tombstones. However, it doesn't suffer from primary clustering since it checks a more dispersed group of cells. The complexity of quadratic probing isn't known.

A major benefit to use quadratic probing is that if the table size is prime and the load factor stays been 0.5, you are guaranteed to be able to insert and item. Additionally, we get the property that no cell is probed twice.

46. Create a hash function for [strings | numbers | cards | people] which generates relatively unique values.

The wizened people of Oracle have decided on a polynomial hash using base 31, and I see no reason to change the status quo.

Numbers are their own hash functions.

There are 52 cards in a deck, so create a bijection between the cards and the set $\{1, 52\}$. Then the problem reduces to the previous problem involving numbers.

People are (somewhat) uniquely identifiable by their full names, which are in turn strings, so this problem reduces to the first involving strings.

47. Hash [strings | numbers | cards | people] into a hash table of size [some number using [separate chaining | open addressing w/ linear probing | open addressing w/ quadratic probing | open addressing w/ double hashing where `h2(key)= 5 - (key % 5)`] and the hash function you wrote above.

Omitted.

48. After hashing the above [strings | numbers | cards | people] remove three of them, making sure to visually indicate any special states.

Omitted.

49. Rehash your tables above to [a prime number over double the size].

Omitted.

50. Write code for an [add | remove | contains | rehash | getLoad] method in a hash table which uses [separate chaining | open addressing with linear probing | open addressing with quadratic probing | open addressing with double hashing where `h2(key)= 7 - (key % 7)`].

Omitted.

51. Explain how Java [String | Integer | Double | Character] class produces hash codes.

String

```java
/**
 * Returns a hash code for this string. The hash code for a
 * {@code String} object is computed as
 * <blockquote><pre>
 * s[0]*31^(n-1) + s[1]*31^(n-2) + ... + s[n-1]
 * </pre></blockquote>
 * using {@code int} arithmetic, where {@code s[i]} is the
 * <i>i</i>th character of the string, {@code n} is the length of
 * the string, and {@code ^} indicates exponentiation.
 * (The hash value of the empty string is zero.)
 *
 * @return  a hash code value for this object.
 */
public int hashCode() {
    int h = hash;
    if (h == 0 && value.length > 0) {
        hash = h = isLatin1() ? StringLatin1.hashCode(value)
                              : StringUTF16.hashCode(value);
    }
    return h;
}
```

```java
public static int hashCode(byte[] value) {
    int h = 0;
    for (byte v : value) {
        h = 31 * h + (v & 0xff);
    }
    return h;
}
```

```java
public static int hashCode(byte[] value) {
    int h = 0;
    int length = value.length >> 1;
    for (int i = 0; i < length; i++) {
        h = 31 * h + getChar(value, i);
    }
    return h;
}
```

Integer

```java
/**
 * Returns a hash code for this {@code Integer}.
 *
 * @return  a hash code value for this object, equal to the
 *          primitive {@code int} value represented by this
 *          {@code Integer} object.
 */
@Override
public int hashCode() {
    return Integer.hashCode(value);
}

/**
 * Returns a hash code for an {@code int} value; compatible with
 * {@code Integer.hashCode()}.
 *
 * @param value the value to hash
 * @since 1.8
 *
 * @return a hash code value for an {@code int} value.
 */
public static int hashCode(int value) {
    return value;
}
```

Double

```java
/**
 * Returns a hash code for this {@code Double} object. The
 * result is the exclusive OR of the two halves of the
 * {@code long} integer bit representation, exactly as
 * produced by the method {@link #doubleToLongBits(double)}, of
 * the primitive {@code double} value represented by this
 * {@code Double} object. That is, the hash code is the value
 * of the expression:
 *
 * <blockquote>
 *  {@code (int)(v^(v>>>32))}
 * </blockquote>
 *
 * where {@code v} is defined by:
```

```
 *
 * <blockquote>
 *  {@code long v = Double.doubleToLongBits(this.doubleValue());}
 * </blockquote>
 *
 * @return  a {@code hash code} value for this object.
 */
@Override
public int hashCode() {
    return Double.hashCode(value);
}

/**
 * Returns a hash code for a {@code double} value; compatible with
 * {@code Double.hashCode()}.
 *
 * @param value the value to hash
 * @return a hash code value for a {@code double} value.
 * @since 1.8
 */
public static int hashCode(double value) {
    long bits = doubleToLongBits(value);
    return (int)(bits ^ (bits >>> 32));
}
```

Character

```
/**
 * Returns the standard hash code as defined by the
 * {@link Object#hashCode} method.  This method
 * is {@code final} in order to ensure that the
 * {@code equals} and {@code hashCode} methods will
 * be consistent in all subclasses.
 */
public final int hashCode() {
    return super.hashCode();
}
```

Since a character is just an ASCII values, it's essentially calling the `Integer`'s `hashCode()` method.

Graphs

52. Explain the following terms: graph, node, edge, adjacent to, directed edge, weight, $|V|$, $|E|$, path, simple path, path length, cycle, degree/indegree/outdegree of a vertex, DAG.

Graph: The set $G = (V, E)$ where V is set of vertices and E is the set of edges.

Node: Synonymous with vertex.

Edge: A pair $(v, w) \in V$.

Adjacency: A vertex v is said to be *adjacent* to a vertex w if and only if there exists an edge $(v, w) \in E$.

Directed edge: An edge which has direction. By the definition of a graph, all edges are directed (because they're tuples).

Weight: We can consider an edge to have a weight (image the cost of a flight between two cities).

$|V|$: The cardinality of the set of vertices – that is, the number of vertices in the graph.

$|E|$: The cardinality of the set of edges – that is, the number of edges in the graph.

Path: A path π is a non-zero length sequence of edges $\{(v_0, v_1), (v_1, v_2), ... (v_{n-1}, v_n)\}$ which connects two vertices.

Simple path: A path where all the vertices are distinct, except possibly the first and last.

Path length: The length of a path.

Cycle: A path which leads to back to its starting vertex.

Degree: The in-degree plus the out-degree..

In-degree: The number of edges coming into a vertex.

Out-degree: The number of edges leaving a vertex.

Directed Acyclic Graph (DAG): A directed graph without cycles.

53. Explain the difference between the following types of graphs: directed/undirected, weighted/unweighted, cyclic/acyclic, dense/sparse, connected/disconnected.
54. Explain the two ways to store a graph we covered in class. When would you want to use which?
55. Given a graph, draw the corresponding adjacency [matrix | list]. Given an adjacency [matrix | list], draw the corresponding graph.

Graph Algorithms

55. Given a graph and a starting location, perform a [breadth-first | depth-first] traversal showing the steps of the accompanying [queue | stack] data structure. Given a choice of neighbors, chose nodes in numerical order.

56. Write code for a recursive depth-first search. Assume you are given an adjacency matrix `m (int[][])`, the `id` of a node to start from, and an `id` of a node to search for. This should return **`true`**/**`false`** depending on if one can get to the search target from the starting node.
57. Given a graph and a source node id, perform Dijkstra's shortest path algorithm on a given graph showing the steps. Use a table to track the current status, distance, and parent pointer for each node.

Graph Algorithms (Prof. Russel's section only) – Omitted

Priority Queues/ Heaps

NB: The numbering in the PDF I'm follow is weird – it's not me.

56. What are the common operations supported by a priority queue?
57. Compare and contrast different data structures we covered in class to implement a priority queue.
58. Write the code to [enqueue | dequeue] from a priority queue stored as [a sorted dynamic array | an unsorted linked list | a heap]. Make sure to maintain the optimal Big-O for the given queue storage.
59. How can you determine the [min | max] value in a heap? Show the steps to remove the [min | max] value from the heap. Show the steps to insert a value into a heap and keep the heap order.
60. Given an array representation of binary heap, show the corresponding tree structure. Given a binary heap tree representation, show the corresponding array assuming root is at index [0|1].
61. What is the difference between "delete" in a heap and "delete" in other types of trees we discussed in class?
62. Explain how heapsort is $O(n \log(n))$ and can be done "in place".
63. Given an array of numbers, show the steps of the optimal heapify algorithm.

All Trees (Binary, K-ary, Binary-Search, AVL, Red-Black, Heap, B/B+)

64. Draw a valid [tree we covered in class].
65. Given a tree (as either a picture or an array), determine if it is a valid [tree we covered in class] and, if not, determine what rule is violated and where the error is.
66. Explain the "rules" of a [tree we covered in class]. What properties have to be maintained when [adding a node to| removing a node from] the tree?
67. Given a [tree we covered in class] and a value, show the steps of [searching for | inserting | deleting] that value.
68. Compare and contrast the [search | insertion | deletion] times for each of the trees we covered.
69. Given [a scenario] determine which tree you would use, justify your answer. Examples:
    - You need to sort 1000 items.
    - You want to keep track of 10,000 key-value pairs such that you can (a) efficiently print all the items in key-order, (b) have fast "look-up", and (c) relatively fast insertion.
    - You want to index a very large data set that does not fit into memory

Non-Balancing Search Trees (BST)

70. Why is the Big-O of inserting into a BST $O(n)$ and not $O(\log(n))$?
71. How can you determine the [min | max] value in a BST tree?
72. Given a node in a BST tree, how can you find [successor | predecessor] of that node (i.e. find the smallest value larger than the node or find the largest value smaller than the node].
73. Write the code for searching a BST for a given value. Assume a generic `Node<T>` class with a data field and left/right references. Assume the search method is given the root of the tree and a value to search for.
74. Given a BST and a value, show the steps to remove that value from the BST.
75. Write the code for [insert | remove| findMin | findMax | printSubset] in a BST. Assume a generic `Node<T>` class with a data field and `left`/`right` references.

Self-Balancing Search Trees (AVL / Red-Black)

76. Explain the "cases" for inserting into a [AVL | Red-Black] tree.
77. Determine an order of inserting the keys 1, 2, and 3 into an AVL which would require a [single right rotation | single left rotation | left-right double rotation | right-left double rotation].
78. Determine a set of keys and an order to insert them which would produce each of the cases for inserting into a Red-Black tree. Label each case with a meaningful name (not just "case 1", "case 2", etc.).
79. Compare AVL trees and Red-Black trees for pros and cons. Explain when/why you would use an AVL instead of a Red-Black Tree and explain when/why you would use an a Red-Black Tree instead of an AVL.

Union Find / Disjoint Sets

80. Explain the types of problems the union-find data structure is designed to help with. Hint: why is it called "union-find".
81. Given an array representing the results of a series of unions and finds, show the forest (graph form) it represents.
82. Explain the difference between "naive-union" and "rank union".
83. Explain the difference between "naive-find" and "path compression find".
84. Given a series of union and find operations, show the effects on the array which backs the union-find data structure using [naive | rank] union and [naive | path compression] find.
85. What is special about the performance analysis of union-find? Hint: this has to do with path compression and the definition of $log^*(n)$.