Final Review

A review of all the concepts covered since the midterm

Connor Baker

2019-05-06, Compiled on May 7, 2019 at 11:18am

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 – see textbook.

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

String

Oracle's release of `String.java` has the following:

```
/**
 * 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;
}
```

Taking a look at `StringLatin1.java` reveals

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

and `StringUTF16.java` similarly shows

```
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

Oracle's release of `Integer.java` has the following:

```
/**
 * 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

Oracle's release of `Double.java` has the following:

```
/**
 * 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

Oracle's release of `Character.java` has the following:

```java
/**
 * 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.

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

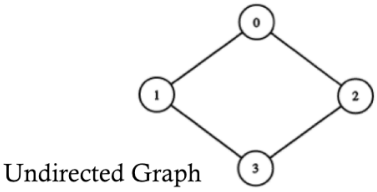
52. Explain the difference between the following types of graphs: directed/undirected, weighted/unweighted, cyclic/acyclic, dense/sparse, connected/disconnected.

- Directed vs. Undirected
  - Presence or absence of edges where the order of the vertices matter
- Weighted vs. Unweighted
  - Presence or absence of edge weights
- Cyclic vs. Acyclic
  - Presence or absence of cycles
- Dense vs. Sparse
  - Presence or absence of a large number of edges relative to the number of vertices
- Connected vs. Disconnected
  - Presence or absence of the property that there is a path from every vertex to every other vertex (when a graph has this property, we call it *strongly connected*)

53. Explain the two ways to store a graph we covered in class. When would you want to use which?

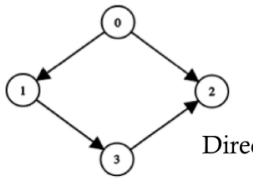
Adjacency Matrix

Assume that all vertices have a unique number identifying them. Then we have an array.



Undirected Graph

	0	1	2	3
0	∞	1	1	∞
1	1	∞	∞	1
2	1	∞	∞	1
3	∞	1	1	∞



Directed Graph

	0	1	2	3
0	∞	1	1	∞
1	∞	∞	∞	1
2	∞	∞	∞	∞
3	∞	∞	1	∞

Figure 1: A comparison of features between undirected and directed graphs

This approach is beneficial when you have memory to spare and want $O(1)$ lookups.

Adjacency List

An adjacency list would be a list of vertices where every vertex points to a list of vertices that it is predecessor of.

This approach is beneficial when you don't have a lot of memory and are okay with $O(n)$ lookups.

54. Given a graph, draw the corresponding adjacency [matrix | list]. Given an adjacency [matrix | list], draw the corresponding graph.

Omitted.

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.

Breadth-First Traversal

Taken from notes on 2019-03-26.

- Given the starting point S
  - Visit all the nodes that are one edge away (S's direct neighbors)
  - Visit all nodes that are two edges away (neighbors of neighbors)
  - Visit all nodes that are three edges away (neighbors of neighbors of neighbors)
  - …
  - Repeat this until all nodes have been visited

Example

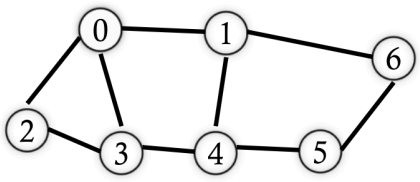


Figure 2: A simple undirected graph

- A breadth-first traversal starting with 0: $\{0\}$

  - Visit all the nodes adjacent to 0: $\{0, 1, 2, 3\}$
  - Visit all the neighbors of those nodes: $\{0, 1, 2, 3, 4, 6\}$
  - Continue the process: $\{0, 1, 2, 3, 4, 6, 5\}$
  - We've reached all the nodes, so we can stop. We've also found that every node in this graph can be reached by a path of at most length 3.

Implementation

- Have we done something similar to this before?
  - Yes, and depending on your progress in Project 3 you might still be doing it.
- We can use a queue
  - Initialize by enqueueing the starting vertex
  - Mark it as visited when we enqueue
  - Process the vertives in a first-in first-out (FIFO) order:
    * Dequeue the first vertex v
    * Enqueue v's neighbor that has not yet been visited or marked

Depth-First Traversal

- Given the starting point S
  - Visit the first neighbor of S
  - Visit the first neighbor of the first neighbor of S
  - Visit the first neighbor fo the first neighbor of the first neighbor of S
  - Repeat this process until there are no more nodes to go, then back, trying the second neighbor of S
  - Repeat that process until we've processed all of the neighbors of S

Example

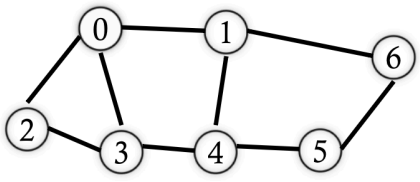


Figure 3: A simple undirected graph

A depth-first traversal starting with 0: $\{0\}$

Pick one neighbor of 0: 1. Then the set of nodes visited is $\{0, 1\}$.

- Pick one neighbor of 1: 4. Then the set of nodes visited is $\{0, 1, 4\}$.
  - Pick one neighbor of 4: 3. Then the set of nodes visited is $\{0, 1, 4, 3\}$.
    - Pick one neighbor of 3: 2. Then the set of nodes visited is $\{0, 1, 4, 3, 2\}$.
      - Pick one neighbor of 2: 2 has no neighbors that we haven't visited already, so we backtrack to 3.
    - Pick one neighbor of 3: 3 has no neighbors that we haven't visited already, so we backtrack to 4. Then the set of nodes visited is $\{0, 1, 4, 3, 2\}$.
  - Pick one neighbor of 4: 5. Then the set of nodes visited is $\{0, 1, 4, 3, 2, 5\}$.
    - Pick one neighbor of 5: 6. Then the set of nodes visited is $\{0, 1, 4, 3, 2, 5, 6\}$.
      - Pick one neighbor of 6: 6 has no neighbors that we haven't visited already, so we backtrack to 5.
    - Pick one neighbor of 5: 5 has no neighbors that we haven't visited already, so we backtrack to 4.
  - Pick one neighbor of 4: 4 has no neighbors that we haven't visited already, so we backtrack to 1.
- Pick one neighbor of 1: 1 has no neighbors that we haven't visited already, so we backtrack to 0.

Pick one neighbor of 0: 0 has no neighbors that we haven't visited already, and we cannot backtrack further, so we're done.

Therefore, our result is $\{0, 1, 4, 3, 2, 5\}$.

Implementation

- Implementation qualms:
  - How do we implement backtracking?
    - With recursion, or equivalently, a stack
  - Is a post-order tree traversal depth-first when applied to graphs? What about a pre-order traversal? What about an in-order traversal?
- Using recursion (or a stack)
  - We push nodes with unvisited neighbors onto the stack
  - Pick an unvisited neighbor to continue
    - If there are no more unvisited neighbors, pop out the node and backtrack

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.

Omitted – see textbook.

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.

Example of Dijkstra's Algorithm

Taken from notes on 2019-03-28.

Suppose we want to find the shortest path given the following graph, and starting at vertex 0.

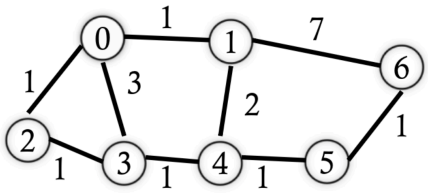


Figure 4: A weighted, undirected graph

Node	0	1	2	3	4	5	6
Path Length	0	∞	∞	∞	∞	∞	∞
Prev	0	-	-	-	-	-	-

- Starting vertex 0
  - Updating for vertices 1, 2, and 3
- Paths: $\{\}$

Node	0	1	2	3	4	5	6
Path Length	0	1	1	3	∞	∞	∞
Prev	0	0	0	0	-	-	-

- Next vertex 1
  - Updating for vertices 4 and 6
- Paths: $\{0, 1\}$

Node	0	1	2	3	4	5	6
Path Length	0	1	1	3	3	∞	8
Prev	0	0	0	0	1	-	1

- Next vertex 2
  - Updating for vertex 3
- Paths: $\{0, 1\}, \{0, 2\}$

Node	0	1	2	3	4	5	6
Path Length	0	1	1	~~3~~ 2	3	∞	8
Prev	0	0	0	~~0~~ 2	1	-	1

- Next vertex 3
  - Updating for vertex 3
    - Reached every node, update nothing
- Paths: $\{0, 1\}, \{0, 2\}, \{0, 2, 3\}$

Node	0	1	2	3	4	5	6
Path Length	0	1	1	~~3~~ 2	3	4	~~8~~ 5
Prev	0	0	0	~~0~~ 2	1	4	~~1~~ 5

- Next vertex 4
  - Updating for vertex 5
- Paths: $\{0, 1\}, \{0, 2\}, \{0, 2, 3\}, \{0, 1, 4\}, \{0, 1, 4, 5\}, \{0, 1, 4, 5, 6\}$

Our final record is then

Node	0	1	2	3	4	5	6
Path Length	0	1	1	~~3~~ 2	3	4	~~8~~ 5
Prev	0	0	0	~~0~~ 2	1	4	~~1~~ 5

- Prev: the previous node in the shortest path starting from 0
- Path length: length of the corresponding shortest path
- What data structure would we use for implementing something like this?
  - A priority queue

Graph Algorithms (Prof. Russel's section only)

Omitted.

Priority Queues/ Heaps

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

56. What are the common operations supported by a priority queue?

    - `add(T t, int p)` and `enqueue(T t, int p)`: enqueue item `t` with priority `p`
    - `peek()` and `findMin()`: return the object with the best priority
        - Per convention, lower is better
        - Symmetric code if higher is better
    - `dequeue()` and `deleteMin()`: remove and return the object with the best priority

57. Compare and contrast different data structures we covered in class to implement a priority queue.

General Performance of Different Implementations

Data Structure	`enqueue()`	`peek()`*	`dequeue()`*	Notes
Unsorted List	$O(1)$	$O(n)$	$O(n)$	best priority can be any location
Sorted Array	$O(n)$	$O(1)$	$O(1)$	best priority at high index
Sorted Linked List	$O(n)$	$O(1)$	$O(1)$	best at head or tail
Multiple Queues	$O(1)$	$O(m)$	$O(m)$	-
Binary Search Tree	$O(\text{height})$	$O(\text{height})$	$O(\text{height})$	min at left-most

- *: assuming best priority
- n: the number of items in a queue
- m: the number of priority levels

Unsorted List

- Pros
    - Easy to maintain/implement
    - Very fast `enqueue` time
- Cons

  - The best priority can be anywhere so it takes $O(n)$ time to find it
  - Relatively slow `peek` and `dequeue` operations
  - Fixed size, though we can copy to a new, larger array (which requires a fair deal of memory to hold both the new and the old arrays at the ready)

Sorted Array

- Pros
  - Easy to implement
  - Very fast `peek` and `dequeue` operations
  - Best priority at the highest index
- Cons
  - Very slow `enqueue` operation time
  - Memory intensive – lots of reshuffling and re-assignment
  - Fixed size, though we can copy to a new, larger array (which requires a fair deal of memory to hold both the new and the old arrays at the ready)

Sorted Linked List

- Pros
  - Easy to maintain/implement
  - Very fast `peek` and `dequeue` operations
  - Can grow infinitely large
  - Best priority at the `head` or `tail` of the linked list
- Cons
  - Insertion is slower because we need to traverse a list so it takes $O(n)$

Multiple Queues

- Pros
  - Fast `enqueue`
  - Relatively fast `peek` and `dequeue`
- Cons
  - Difficult to maintain/implement

Binary Search Tree

- Pros
  - Every operation takes $O(\log_2(\text{height}))$ time
- Cons
  - Difficult to maintain/implement
  - The minimum or maximum priority are at the leftmost or rightmost leaf, respectively

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.

Omitted – see textbook.

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.

Determining the Minimum or Maximum Value in a Heap

Recall the *Heap Order Property*: In a heap, for every node X with parent P, the key in P is smaller than or equal to the key in X (at least, for min-heaps – with max-heaps, the property is inverted).

Therefore, if the heap is a min-heap, the minimum is at the root and the maximum is one of the leaves on the bottom of the tree.

Similarly, if the heap is a max-heap, the maximum is at the root and the minimum is one of the leaves on the bottom of the tree.

Removal

As a fact worth remembering, Recall that our textbook (Weiss) states that

> The `deleteMin` operation is logarithmic in both the worst and average cases.

To remove the minimum value in a min-heap, we pop the most recently added value out, put a hole at the root, and then percolate the hole downwards, placing it wherever we can in a leaf. We then insert the value that we popped in the hole.

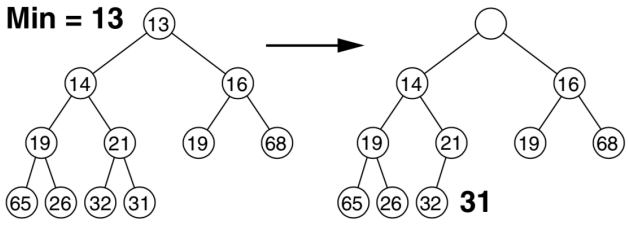


Figure 5: Creation of the hole at the root

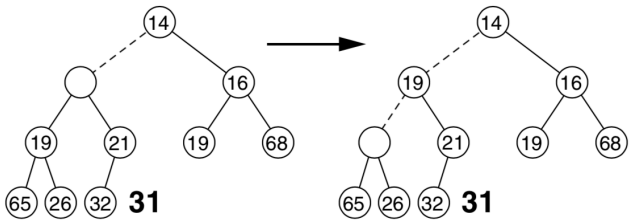


Figure 6: The next two steps in the `deleteMin` operation

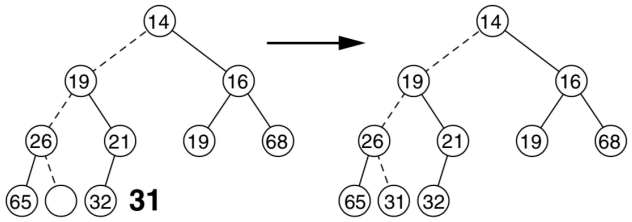


Figure 7: The last two steps in the `deleteMin` operation

To remove the maximum value from a min-heap, find the leaf which holds the maximum value and replace it with the farthest right node possible.

Removal for a max-heap follows similarly to removal for min-heap, albeit with the locations of the minimum and maximum swapped.

Insertion

As a fact worth remembering, Recall that our textbook (Weiss) states that > Insertion takes constant time on average but logarithmic time in the worst case.

To insert a value in a min-heap, we add the value at the next available leaf (since we must maintain a complete tree, we cannot insert it anywhere else). Then, percolate the value up the tree until we have satisfied the Heap Order Property.

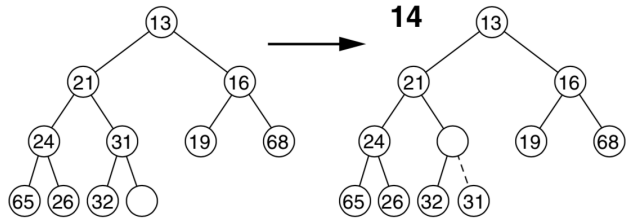


Figure 8: Attempt to insert 14, creating the hole and bubbling the hole up

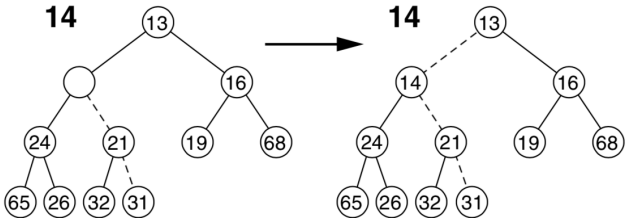


Figure 9: The remaining two steps required to insert 14

Insertion for a max-heap follows similarly.

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].

Recall that the array representation of a binary heap is essentially level-order:

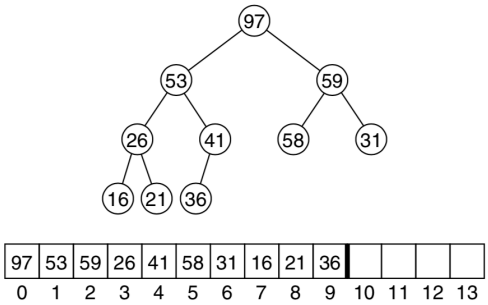


Figure 10: The array representation of a max-heap binary tree

61. What is the difference between “delete” in a heap and “delete” in other types of trees we discussed in class?

With the other trees that we discussed there was this notion of recursively splitting or joining sub-trees when removing nodes. With the removal of a node in a heap, we have a different way of thinking about movement within the tree: percolation.

62. Explain how heapsort is $O(n \log(n))$ and can be done “in place”.

Weiss et. al note (in Section 21.5) that heapsort can be performed by doing the following:

> 1. Tossing each item into a binary heap
> 2. Applying `buildHeap`
> 3. Calling `deleteMin` N times, with the items exiting the heap in sorted order
>
> Step 1 takes linear time total, and step 2 takes linear time. In step 3, each call to `deleteMin` takes logarithmic time, so N calls take $O(N \log(N))$ time. Consequently, we have an $O(N \log(N))$ worst-case sorting algorithm, called heapsort, which is as good as can be achieved by a comparison-based algorithm (see Section

> 8.8).

They note later within the same section that Heapsort can be done in place because it uses the empty parts of the array in which the heap is stored; calling `deleteMin` frees up one cell, which can then be used to shuffle around a value in the heap.

63. Given an array of numbers, show the steps of the optimal heapify algorithm.

Taken from notes on 2019-03-26.

Issue 3: "Heapify"

- We need to be able to convert an existing array into a heap
- We can build the heap bottom up through repeated application of `percolateDown()`
  - Start one level above the bottom
  - Work right to left, bottom up
  - Apply `percolateDown()` for each non-leaf node
    - Compare the non-leaf node with its children
    - Swap if the heap order is violated

Example: Min Heap

Assume we're given the array

```
int[] arr = [92, 47, 21, 20, 12, 45, 63, 61, 17, 55, 37, 25, 64, 83, 73];
```

Then we perform the following steps to convert it to a heap (noting that `percolateDown()` takes as an argument the *index* of the array to percolate):

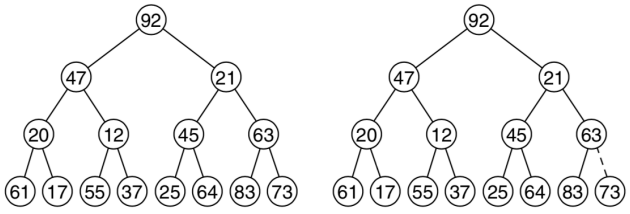


Figure 11: Left: Initial heap; Right: after `percolateDown(7)`

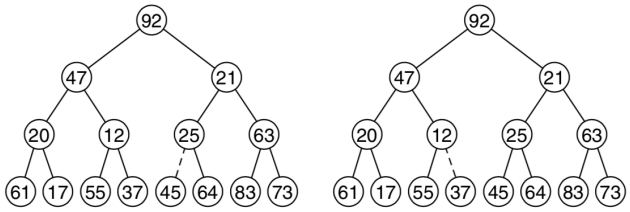


Figure 12: Left: After `percolateDown(6)`; Right: After `percolateDown(5)`

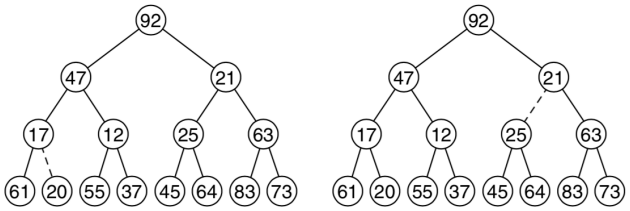


Figure 13: Left: After `percolateDown(4)`; Right: After `percolateDown(3)`

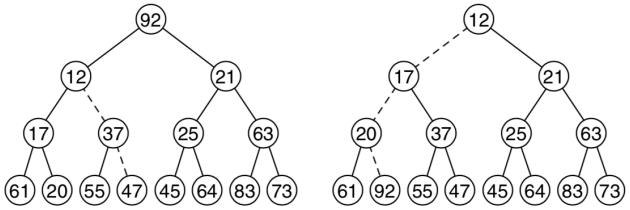


Figure 14: Left: After `percolateDown(2)`; Left: After `percolateDown(1)` and `buildHeap` terminate

Heapify Implementation

```
public void buildHeap() {
  for (int i = parent(this.size); i >= root(); i--) {
    this.percolateDown(i);
  }
}
```

Heapify Complexity

- Assume the tree height is h, and count the work as the number of comparisons/swaps done at each level
  - At the bottom (level 0) there are (at most) 2^h nodes
    - We do not do anything, so the work is zero

  - Level 1 has 2^{h-1} nodes
    - Each might move down (at most) 1 level
  - Level 2 has 2^{h-2} nodes
    - Each might move down (at most) 2 levels
  - Level i is the ith from the bottom and has 2^{h-i} nodes
  - Level h is the root, has $2^{h-h} = 2^0 = 1$ node
- Each level i node can move at most i steps down, so

$$\text{moves} = \sum_{i=1}^{h} i \times 2^{h-i} = \sum_{i=1}^{\log_2(n)} i \times 2^{\log_2(n-i)} = \sum_{i=1}^{\log_2(n)} i \times \frac{2^{\log_2(n)}}{2^i} = n \sum_{i=1}^{\log_2(n)} \frac{i}{2^i}.$$

  Since

$$\sum_{i=1}^{\infty} \frac{i}{2^i} \to 2,$$

  we know that

$$n \sum_{i=1}^{\log_2(n)} \frac{i}{2^i} \leq n \times 2$$

  and

$$n \times 2 \in O(n).$$

  Therefore

$$\text{moves} \in O(n).$$

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

64. Draw a valid [tree we covered in class].

Omitted.

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.

Omitted.

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 $log^*(n)$.