CS 61B Final Review

Albert Ye

December 11, 2022

1 Midterm 1

remember what's here LMAO

2 Midterm 2

2.1 Dynamic Method Selection

For an object `x` such that `A x = new B();`

- `A` is static type, which is what `x` is declared to be.
- `B` is dynamic type, which is what `x` is initialized as.

Casting changes the **static type**, but not the **dynamic type**. If casting makes the dynamic type above the static type, it throws a runtime error but not a compilation error. When an object is cast, it's viewed by the compiler in its static type and not its dynamic type (obviously). Casts cannot be performed by turning a class into a class that is neither above nor below the static type, and this would cause a compile error.

For overridden methods, if method is static choose from `A`, if non-static choose from `B`. For non-static methods, parameters are pulled from `A`. If there are no functions in `B` with the same parameters, we use the function in `A`. Parameters must be exact; if there is a function in `B` that takes in a subclass of the parameter used for `A`, we still use the method in `A`.

2.2 Iteration

Any class that extends the interface `Iterable` must have a function `iterator` that returns an `Iterator`. An iterator basically gives the current value and a way of getting to the next value until an end value.

Although practically this should be done to iterate through the entire DS, we will need to look at how it's actually implemented in order to check how it actually behaves.

When iterating, the `next` function doesn't reset. Running the for all loop creates a new iterator and loops through it. Slightly confusing to work with compared to python, where `list(generator)` starts from the current position of the generator.

2.3 Asymptotics

Only consider the largest term, ignore coefficients. $O(f(n))$ works if $f(n)$ is any upper bound for the runtime, and $\Omega(f(n))$ works if $f(n)$ is any lower bound for the runtime, so like $\Omega(1)$ always works.

If the upperbound O is equal to the lowerbound Ω, then we can use Θ.

If we want to find asymptotics in terms of N, we try to find the complexity for N approaching infinity. So any "edge cases" for small N will not cut it

2.4 DSU

2.5 Binary Search Trees

2.6 Hash Table

2.7 Heap

Heaps are also represented as priority queues, but we can cast that aside for a quick second...

Definition 1 (Min-Heap Property)
The root of a tree/subtree is the minimum value, and other values are all less. Max-Heap property is pretty much the same thing, but replace min with max.

In Java, we store the min-heap as an array, `[x, 1, 2, 4, 5, 3, 7, 8]`. Tree is traversed in level-order (which is just breadth first). So we do level 1 (root), then level 2 (children of root), etc. So `parent[i] = i / 2;`, `leftChild(i) = 2*i`, `rightChild(i) = 2*i+1`.

2.7.1 Insertion

Insert the new element to the end of the array, which is the bottom-most layer & left-most element. To make it agree with the principle, we bubble-up by swapping it with any parent that is larger than it.

Because heaps are bushy by construction, the complexity is logarithmic.

2.7.2 Deletion

Delete the root node, and switch it with the bottom-most level & right-most element (last element). TO make it agree with the principle, we bubble-down by swapping it with its smaller child.

Again, the complexity is logarithmic.

3 Post-Midterm 2

3.1 Sorting

Bubble sort is too easy I guess, and takes up to quadratic time.

3.1.1 Insertion Sort

Insert into the sorted component at the right position. Takes quadratic time. Can be spotted by looking for an increasing sorted component

3.1.2 Selection Sort

3.1.3 Merge Sort

3.1.4 Quick Sort

Literally worse than merge sort in that it's not stable *and* potentially slower given on our choice of pivot. I propose we call this gaslighting sort.

3.2 Binary Search Trees

3.3 Graphs

3.3.1 DFS/BFS

what is there to explain. $O(V + E)$.

3.3.2 Dijkstra

Single-source shortest path, finding the shortest path from the source vertex s to all destinations d. We should end up with a shortest path tree with all shortest paths from s to every node d. We do best first search, finding the shortest edge and relaxing every edge $v \to w$ at vertex v.

This is like BFS but with priority queue instead of regular queue. Complexity is $\Theta(E \log V + V \log V)$ because of the E priority changes, V adds and removes, all of which take $\Theta(\log V)$ time.

3.3.3 A* Search

We create another heuristics function $h(v, t)$, which is the estimate of distance between the current vertex v and the destination vertex t. Instead of visiting in order of $d(s, v)$ we visit in order of $d(s, v) + h(v, t)$.

If we run our heuristic incorrectly, we run the risk of not finding the correct shortest path. If h is too far from the actual distance from v to t. Our heuristic must be admissible and consistent, meaning that $h(v, d) \leq d(t, v)$ and for each neighbor of w, $h(v, t) \leq d(v, w) + h(w, t)$. Rest is out of scope.

3.4 Minimum Spanning Trees

3.4.1 Kruskal's Algorithm

3.4.2 Prim's Algorithm