CS 61B Course Summary

Matthew Signorotti

May 2018

1 Data Structures

- **List**
  - **Linked list**: The simplest list data structure. It features constant-time accessing from and inserting to the front and back of the list. Unfortunately, accessing from and inserting to the middle are linear time.
  - **Array list** (resizing array): An array which can be inserted to and deleted from any position. Accessing from any point in the list is constant time, and resizing is amortized (average) constant time if array size can only double or half.
    - Suffers when adding to the middle, which requires shifting all the elements to the right by one, or to the front (except in an array-dequeue or other implementation in which the list can actually wrap around to the back of the internal array).
- **Queue**
  - A more specific, bare-bones interface which only requires operations to add, remove, and examine elements.
  - **Linked list**: See above; it supports adding and removing from not only the front, but also the end.
  - **Priority queue**
    - **Minimum heap**: A type of tree data structure which is the most common implementation of a priority queue, and which must obey two properties: completeness and the min-heap property. To be complete, every non-leaf node has the maximum number of children, and the nodes on the bottom level are positioned as much to the "left" side of the tree as possible. To obey the min-heap property, every node is greater than or equal to its parents.
      - Used in the `java.util.PriorityQueue` implementation.

    - **Maximum heap**: The opposite of a minimum heap. It obeys the same properties, except every node is less than or equal to its parents. Useful if you want to obtain
    - What makes the priority queue so great? Why not collect a jumbled set of data and sort it?
        - Adding an element is $\Theta(\log N)$, and roughly by that logic, constructing the queue is $\Theta(N \log N)$ — the same, asymptotically, as adding everything to a collection and then sorting it at the end. In this case, either approach is valid, and you would want to run efficiency tests to determine a strategy.
        - However, priority queues are truly advantageous for large sets of N elements, of which we only care about a small subset, say K elements:
            1. We only need to store $\Theta(K)$ memory.
            2. "Sorting" through all the information will take $\Theta(N \log K)$ time, not $\Theta(N \log N)$ (the min-heap tree will be maximum $\log K$ height, which is much less than $\log N$).
  - The `ArrayDeque` encountered in CS 61B was an array-implemented queue which could add to both the back and the front of the queue, hence "double-ended queue."

- **Stack**
  - A queue which is last-in, first-out (LIFO), rather than first-in, first-out (FIFO).

- **Map**
  - Hash map
    - Chaining (buckets) hash table: Used the most in practice. An array maps `hashCode % buckets.length` to linked lists of elements.
    - Linear probing hash table: No buckets; the internal array contains at most one item, directly. If a spot is filled, linearly "probe" the list forward until an open spot is found, or else the operation fails.
    - Advantage: amortized $\Theta(1)$ (constant-time) addition and retrieval of elements, provided an evenly distributed `hashCode` function.
    - The main benefit of hashing over using a primitive list of key-value pairs is not having to do $\Theta(N)$ `compareTo` computations.
  - Tree map
    - Binary search tree: Very simple implementation which, for a balanced tree, takes $\Theta(\log N)$ for addition and checking for elements. Removal is accomplished by "promoting" the minimum

element in the right sub-tree, or the maximum element in the left sub-tree.
    - Unfortunately, this implementation is less ideal because empirically, after adding and removing from a tree, the tree's height tends toward $\Theta(\sqrt{N})$.
* B-trees: Guaranteed to be balanced. A b-tree of magnitude 3, or 2-3 tree, has nodes of up to 2 items and up to 3 children. Objects are inserted into the leaf nodes, and if a node is "overstuffed" (exceeds capacity), the tree "splits" upward. When splitting, only the middle node, or left-middle if there are two middle nodes, goes upward.

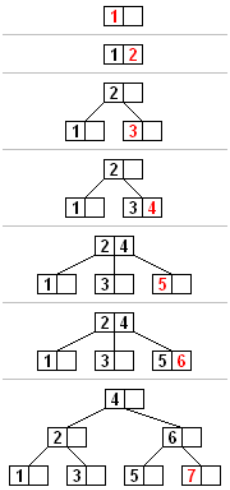

Figure 1: Inserting into a left-leaning b-tree of magnitude 3 (2-3 tree)

* (Left-leaning) Red-black trees: Unlike b-trees, these perform well and can be searched just like a binary search tree. To construct one, start with a b-tree. Then, for every node with multiple items, keep the rightmost item as the root node, and each item to the left becomes the left child of the current node.

– Tries
* A tree in which each node represents a character or digit of a greater expression. Stopping at a certain node tells whether the sequence of characters from the root to that point is contained in the map. In most trie implementations, the character each node represents is the character represented by the mapping to that node in its parent's `children` map/array.
* Ternary search trie: A less conventional idea in which nodes of equivalent depth and with the same parent as an existing node are added to that node's `left` or `right` attribute.

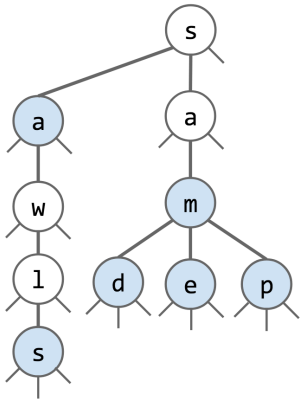

Figure 2: Ternary search trie after inserting “a”, “awls”, “sam”, “sad”, “sap”, and “same”

  - ∗ Particularly effective to retrieve items which share a prefix, or for digit-by-digit problems such as radix sort. Used primarily for its rapid prefix matching.
  - ∗ Highly space efficient, more so than a hash set, but potentially less time efficient to retrieve items with non-constant time `hashCode` functions.
  - ∗ Unlike other abstract data types, lacks a widely used implementation in `java.util`.

- **Set**
  - Hash sets: See above.
  - Tree sets: See above.
  - Tries: See above.
  - Any map implementation can be reduced to a set implementation.
- **Graph**
  - Vertices typically have numbers, and are looked up with something like a map from strings to integers.
  - Adjacency matrix (initial attempt): Two-dimensional matrix telling whether two vertices are connected. Not as great: $\Theta(V^2)$ space and $\Theta(V)$ time to look up adjacent vertices.
  - Edge set: Hash set of pairs of integers representing connections.
  - Adjacency list: A map or array records a list of “children” for every node.

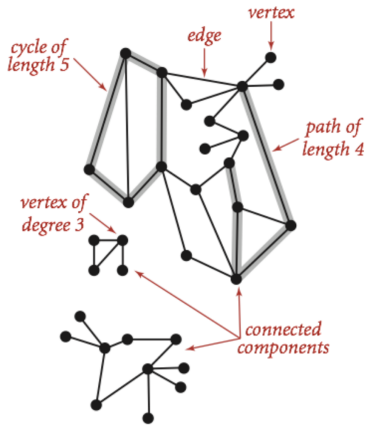

Figure 3: Graph terminology

  - A similar representation was used in Project 3, except we went further by creating node objects which contain sets of children.

- **Disjoint set**
  - Support two operations: `connect` and `isConnected`.
  - How we designed the data structure:
    - Attempt #1, `QuickFindDS`: Optimize `isConnected`. Simply map item numbers to an integer representing the number of the set each is part of. This has constant time `isConnected`, but `connect` is a different story.
    - Attempt #2, `QuickUnionDS`: Array-based trees (array-based since we need to find any node quickly) represent each set. Connect trees to connect elements; trace items to their roots to see if they are connected.
    - Attempt #3, `WeightedQuickUnionDS`: Same as attempt # 2, except to connect sets, always connect the tree of fewer nodes to the root of the other one.
    - Path compression: A very efficient optimization. Whenever we call `connect`, we call a `find` helper function on both arguments to find the roots of their respective trees. The path compression optimization attaches each parent node of `find`'s argument directly to the root. `find` will initially still take logarithmic time, but called again on the same argument, it will take constant time.

2 Algorithms

Tree (and Graph) Traversals

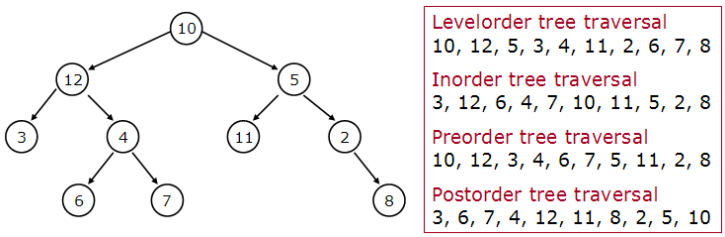

Figure 4: The three depth-first tree traversals

- Depth-first traversals
  - Implemented recursively or with a stack data structure, as used below.
  - Pre-order traversal: Add or print the current node, and the children to the stack, and retrieve the next node from the stack.
  - In-order traversal: Add the left child to the stack, add or print the current node, and then add the right child to the stack.
  - Post-order traversal: Add the children to the stack and then add or print the current node.
- Breadth-first traversal: Use a queue to choose the next node. Add or print the current node, add all its children to the queue, and dequeue the next item.
- The same traversals can be extended to graph algorithms. For graphs, which have loops, make sure to mark each node so as not to reach the same node multiple times and encounter an infinite loop.

Graph Algorithms

- **Search algorithms**
  - $\Theta(V)$ space
  - Breadth-first search: See above. $\Theta(V + E)$ time.
  - Depth-first search: See above. $\Theta(V + E)$ time.
  - Dijkstra's algorithm: Used to find shortest paths to a point or every point from a source. Using a priority queue, expand the graph vertex by vertex, choosing vertices of minimum distance from the source. All edge weights should have the same sign, i.e. non-negative. Implemented with a min-heap, is $O(E \log V)$.

  - A* search algorithm: Same as Dijkstra's, except vertices are ordered by some estimate heuristic representing the minimum amount of work to reach the target. Used in artificial intelligence, Google Maps; highly efficient to find the shortest path from one node to another. Best runtime, but no clear asymptotic complexity without knowing the heuristic.

- **Minimum spanning tree**
  - Cut property: For all edges connecting two sets of vertices, the minimum spanning tree will contain just one of those edges, and it will be the smallest one.
  - Prim's algorithm: Starting with a tree of just one node, create a priority queue to sort edges by weight. Adding edges touched by the tree, and using a disjoint set to prevent cycles, add $N-1$ edges.
  - Kruskall's alogirhtm: Sort the edges into a priority queue. Add $N-1$ edges which, according to a disjoint set, do not create a cycle.
  - Maximum spanning tree: Multiply the weights by -1; run either algorithm.

- **Topological sorting**: Generating a valid ordering of vertices in which, for each vertex, its "parents" are before it in the ordering. This is achieved by a reverse post-order, or simply the post-order traversal of a list, reversed. A breadth-first search also works if the graph has only one source node.

Sorting

	Time	**Memory**	**Usage**
Quick sort	$\Theta(N \log N)$ to $\Theta(N^2)$	$\Theta(\log N)$	Fastest, but unstable
Merge sort	$\Theta(N \log N)$	$\Theta(N)$[1]	Best stable sort
Heap sort	$\Theta(N$ to $N \log N)$[2]	$\Theta(1)$	Bad caching (hardware)
Insertion sort	$\Theta(N+K)$[3] to $\Theta(N^2)$	$\Theta(1)$	Almost-sorted arrays
Radix sort	$\Theta(L(N+R))$		Bounded-length data

Comparison Sorts

- **Quick sort**: The best performance empirically. Pick an element to partition around, preferably randomly so that the algorithm can't be "broken," and throw everything either right or left of it. Then sort the left and right sides of that element.

[1] Common implementations do not sort in place, leading to $\Theta(N)$ memory.
[2] $\Theta(N)$ is possible for an array of all duplicates.
[3] K inversions (pairs of items in which the left is greater than the right).

- **Merge sort**: Sort two halves, then merge them into one whole. Alternatively, can be implemented iteratively from the "bottom up" — sort every two elements, then every four, etc.
- **Heap sort**: From right to left, "heapify" the input array (this $\Theta(N)$ bottom-up approach beats $\Theta(N \log N)$ top-down, due to some fancy math — see Discussion 13 for a proof). Then empty the heap, reinserting the items at the end. Despite its great asymptotic complexity, rarely used due to its bad caching performance.
- **Insertion sort**: For each element which is less than the element next to it, swap it leftward until it is greater than the adjacent element.
- **Selection sort**: Avoid! Has $\Theta(N^2)$ runtime. For each index, it finds the minimum element left of that index, and swaps it into the position.

Comparison sorts such as quick sort and merge sort are believed to have achieved *ideal* asymptotic runtime for a sort which uses `compareTo` to order items. This is because for N items, the sorting problem can be reduced to a decision tree with $N!$ leaf nodes, each representing an arrangement of the N objects, and a height of $\Theta(\log N!)$. (This assumes the tree has a constant branching factor throughout, which is fairly intuitive, but not entirely justified.) The number of decisions, $\Theta(\log N!)$, is a runtime equivalent to $\Theta(N \log N)$ because of the following proof:

$$\log N! = \log N + \log(N-1) + \cdots + \log 1 \longrightarrow N \log N \in \Omega(\log N!)$$
$$N! > (N/2)^{N/2} \longrightarrow \log N! > \log(N/2)^{N/2} \longrightarrow \log N! \in \Omega(N \log N)$$
$$\log N! \in \Theta(N \log N)$$

Radix Sorts

The above "ideal" runtime is for sorts based on comparing items, but could there be a different type of sort with better runtime?

- **Counting sort** (subroutine for radix sort): Count the frequencies of each of R digits in the input. Use these frequencies to generate starting indices to place objects in the output array. Iterate through the input array and place each item at its starting index in the output array, updating the starting index.
- **Radix sort**: For each digit position in the input data, perform a counting sort. LSD (least significant digit) sort starts with the least significant, typically rightmost, digit. MSD sort does the opposite, and it also has to subdivide/group the subproblems by digit in the most recently sorted index.

- Runtime $\Theta(L(N + R))$ is proportional to the number of digits L, the number of counting sorts required. Each counting sort iterates over N indices and must create a frequency arrays of length R, to accommodate every possible digit.
- Performs best only if L is bounded by a constant. For instance, integers or fixed-length strings have a maximum size.