Data Structures & Algorithms

CS216 Midterm Review
Spring 2022

Time complexity

- A (finite) set of elementary operations is a choice of operations to consider.
  - E.g. comparison (<=, ==, etc.) and/or arithmetic (+, -, *, etc.)
  - Individual elementary operations should have bounded run-time.
- The time complexity of an algorithm is a count of the number of elementary operations that need to be executed.
  - Typically dependent on the “size” of the input.
  - E.g. larger arrays are harder to search and larger integers are harder to factorize.
  - Often difficult (and unnecessary) to get an exact count. Frequently use big-oh notation instead.

Definition: $f(n) = O(g(n))$ means $f(n)$’s order of magnitude is at most $g(n)$’s order of magnitude.

Arrays

- Data structure of fixed size whose elements are stored in contiguous memory.
- Naturally unordered, but can be coupled with insertion and deletion methods which maintain order.
- Unordered time complexities
  - Insert: O(1)*; search and delete: O(n)
  - Insert at next available slot unless* expanding. Expansion costs O(n).
- Ordered time complexities
  - Insert and delete: O(n); search O(log n)

Question: Can you characterize a generic situation where arrays are acceptable despite the poor time complexities of most of their associated operations?

Linked lists

- Data structure of dynamic size whose elements are contained in nodes which link together. Singly linked and doubly linked versions.
- Naturally unordered, but can be coupled with insertion and deletion methods which maintain order.
  - Binary search is more efficient in an array. Why?
  - Typically used for unordered data.
- Time complexities
  - Insert O(1); search and delete O(n).
  - No penalty for expansion, unlike the array.

Question: Can you characterize generic situations where you would use a linked list over an array or vice versa?

Stacks, queues, and deques

- The stack is a first-in-last-out (FILO) list structure.
    - Can use a singly linked list as a backing data structure.
    - What needs to be added/renamed, if anything?
- The queue is a first-in-first-out (FIFO) list structure.
    - Same comment and question from above.
- The deque is a FIFO list structure from both ends.
    - Can use a doubly-linked list as a backing data structure.
    - front/back (en/de)queue.

Priority queues

- Queue-like structure which prioritizes the dequeue of the “most important” element or “most urgent” request.
- Efficiently implemented using a min-heap as the backing data structure.
  - Min-heap is a (binary) tree whose nodes satisfy the min-heap property: each parent is prioritized over its children. I.e. children are heavier than their parents.
    - Min-heaps often use an array (dynamic) as a backing data structure.
    - Parent at index *i* has children, if any, at indices *2i+1* and *2i+2*.
  - The fundamental operations of the min-heap involve bubble-up and bubble-down, which are used by enqueue (insert) and dequeue (removeMin) methods, respectively.
- Time complexities
  - Enqueue: O(log n). Place at next available space in array. Bubble-up. Tree has height O(log n).
  - Dequeue: O(log n). Promote last element in array to top of tree. Bubble-down.

Hash tables

- Unordered data structure which places elements in buckets via a hash function, where individual buckets typically have a low population.
- Sub-components: hash function, array (holding buckets), linked lists (buckets).
  - Hashing an object has time complexity O(1).
  - For a linked list with bounded size... insert, search, and delete have time complexity O(1).
- Time complexities
  - Search, insert, and delete: O(1) assuming buckets are not astronomically unbalanced. Statistically speaking, you should never worry about this when using a robust hash function.
  - Expansion: O(n). Create new containers and rehash data. Typically occurs when load factor – ratio of elements (keys) to buckets – exceeds $\boldsymbol{\alpha}_{max} = 0.75$.
  - Contraction: O(n). Similar to expansion. Typically when load factor dips below $\boldsymbol{\alpha}_{max} / 4$.

AVL trees

- Ordered data structure utilizing nodes organized within a balanced BST.
  - Nodes are height-balanced if the height of their children does not differ by more than one; the tree is height-balanced if all of its nodes are height-balanced.
  - Nodes satisfy the BST property if all of its left descendents are “lesser” and all of its right descendents are “greater”; the tree is BST if all of its nodes satisfy the BST property.
  - AVL trees are height-balanced BST with mechanisms for self-balance after insertion and deletion.
- Big idea: moving arbitrarily from a parent to child essentially reduces the search space by a factor of one-half.
- Time complexities
  - Insert, search, and delete: O(log n)
  - Search is essentially the binary search; see comment above on halving the search space.
  - Insertion requires a search for insertion location, insertion of node, and a call to balance along each node back through the search path. Note: balancing a single node is O(1). Tree has height O(log n)

Pointed review & exercise

- **Exercise 1:** Code a node-linked queue from scratch, with bare minimum in terms of methods. (E.g. queues don't necessarily need search or traversal methods.)
- **Exercise 2:** Consider a min-heap with ArrayList as a backing data structure. Code the recursive method bubbleDown(int idx) which
    - assumes that the idx-th member of the min-heap is *the reason* the heap potentially violates the min-heap property, fixable by bubbling down, and
    - takes the idx-th member of the array and bubbles it down far enough to satisfy the min-heap.
- **Exercise 3:** Code a fixed-sized hash table using ArrayList and LinkedList for backing structures. Should support insertion, search, and deletion.
- **Exercise 4:** Code an iterative version of the insertion method for AVL trees. (Note that we did this recursively in class.)
- **Exercise 5:** Explain exactly how/when rotations occur during AVL self-balance.