Solutions to Fundamental Algorithms and Data Structures

Data Structures

1. What is the time complexity of adding an element to the end of a dynamic array? What is the time complexity of adding an element to the front of a dynamic array?

Adding an element to the end of a dynamic array takes worst-case time O(n) and amortized time O(1) (any sequence of n appends takes time O(n), though not all operations take the same amount of time). Adding to the front takes time O(n), though you could conceivably build the data structure to make this also run in amortized time O(1) if you use a circular buffer.

2. Give one example of an operation on a doubly-linked list that is faster than the corresponding operation on a singly-linked list.

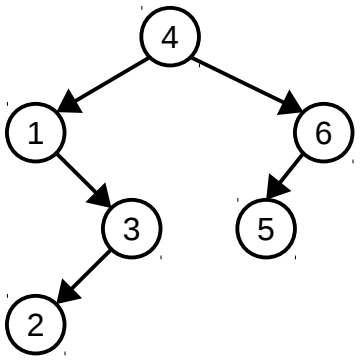
Deleting an element from a doubly-linked list takes time O(1) if you just have a pointer to the element to remove. This can take time O(n) in a singly-linked list because you need to rescan the list to find the node one step before the node to remove.

3. Give an example of when it would be useful to store head and tail pointers in a linked list.

If you were implementing a queue using a linked list, storing a head and tail pointer can make enqueue, dequeue, and front all run in time O(1), since you know where to look to enqueue the element.

4. Draw the binary search tree that results from inserting the elements 4, 1, 3, 2, 6, 5 into an empty tree with no rebalancing. What is the height of this tree?

Here's the tree:



Its height is 3.

5. What is the maximum possible height of a binary search tree containing n elements? What is the minimum possible height?

The maximum possible height is $n - 1$, which happens if the tree degenerates to a linked list. The minimum possible height is O(log n), which happens if it's perfectly balanced.

6. What data structures might you use to implement a set? List some tradeoffs between each implementation.

Here are a few options:

- BST: Supports all operations in worst-case O(log n) time. Requires that stored elements be comparable to one another. Locality of reference isn't the best due to cells being scattered around in RAM. Can iterate in sorted order.
- Hash table: Supports all operations in expected amortized O(1) time. Requires that elements have a hash function and be equality comparable. Locality of reference depends on the type of hash table, but is probably good if we use linear probing or another open addressing scheme. In the worst case, operations degenerate to time O(n) with a bad hash function. Cannot iterate in sorted order.
- Trie: Supports all operations in O(L) time, where L is the length of the string under consideration. Locality of reference is okay but not great. Memory usage can be a bit high depending on how child pointers are stored. Only works on string data or data encoded as strings. Can iterate in sorted order and check whether a string is a prefix of some object in the set.
- Bitvector: Supports all operations in O(1) time. Locality of reference is excellent. Memory usage is O(U), where U is the maximum possible value that can be stored. Cannot efficiently iterate across elements. Only works on fixed-size integer data or data encoded as fixed-sized integers.

7. What data structures might you use to implement a stack? List some tradeoffs between each implementation.

Here are a few options:

- Dynamic array: All operations run in amortized O(1) time and with good wall-clock performance. Excellent locality of reference. Certain operations might run in time O(n) when resizing must occur.
- Linked list: All operations run in worst-case O(1) time and with okay wall-clock performance. Poor locality of reference.

8. What is the average-case time complexity of looking up an element in a hash table? What is the worst-case time complexity of looking up an element in a hash table?

On average, a hash table lookup takes time O(1). In the worst case, this can degenerate to time O(n) if there are lots of hashing collisions.

9. What happens if you enqueue a series of n numbers into a priority queue and then dequeue them all?

They come back in sorted order.

Algorithms

1. What are the best-case and worst-case runtimes of binary search on a sorted array of length n?

In the best case, we find what we're looking for immediately in the middle of the array, which takes time O(1). In the worst case, the element isn't present and we spend O(log n) time trying to find it.

2. What are the best-case and worst-case runtimes for quicksort on an array of length n?

In the best case, we get a perfect split each time, giving runtime O(n log n). In the worst case, we get awful splits the whole way, giving runtime O(n^2).

3. Give one advantage of mergesort over quicksort and vice-versa.

Mergesort is a stable sort and has worst-case O(n log n) runtime. Quicksort is faster in practice due to locality of reference and can be implemented to use O(log n) memory, compared to O(n) memory for mergesort.

4. Which graph algorithm would you use to find the shortest path in a graph between two nodes? Does your answer rely on any assumptions about the graph?

If the graph has no edge weights, BFS will find shortest paths in a graph. If there are edge weights and they're nonnegative, BFS might not work, but Dijkstra's would. If there are edge weights that can be both positive or negative, Bellman-Ford would be appropriate.

Algorithms and Data Structure Design

1. You are given a file containing n numbers in no particular order along with a value k. Design an efficient algorithm for finding the kth largest number out of that file. See if you can solve the problem in time $O(n \log k)$.

One option is to maintain a balanced BST holding the k largest elements seen so far. Start by reading k elements from the file into the BST. Then, repeatedly insert the next element from the file into the BST and then delete the smallest element from the BST. Once the file has been processed this way, the BST will contain the k largest elements of the file, since an element is only removed from the BST if there are at least k elements larger than it. Finally, return the smallest element in the BST.

Since the BST never has more than k elements in it, each operation on it takes time $O(\log k)$. We perform n total operations on it, so the runtime is $O(n \log k)$. The space usage is only $O(k)$, which we need to hold the BST.

2. You would like to design a data structure supporting the following operations: *insert*(x), which adds x into the data structure (assuming it's not already there); *contains*(x), which reports whether x is in the collection; *delete*(x), which removes x from the collection, and *list-in-order*, which lists the element of the collection in the order in which they were inserted. Design the most efficient data structure that you can for this setup.

Intuitively, we can store everything in a hash table with a doubly-linked list threaded through the elements in the order in which they were inserted. Each hash table entry will store both the value and a node for a doubly-linked list, and globally we'll store a head and tail pointer in the list. Whenever we *insert* an element, we look it up in the hash table and, if it's not already present, we add it to the hash table and append the cell to the doubly-linked list. When we *delete* an element, we remove it from the hash table and splice it out of the doubly-linked list. We can perform a *contains* operation by doing a hash table lookup, and we can do *list-in-order* by tracing the linked list from the head to the tail.

Each operation except for *list-in-order* requires O(1) hash table operations and O(1) linked list operations, so each runs in expected, amortized time O(1). *list-in-order* requires time $O(n)$, and overall only $O(n)$ space is needed.

(This data structure is sometimes called a *linked hash set*, by the way.)

3. Design a data structure that models a stack of integers with the normal stack operations of *push* and *pop*, but which also supports the operation *find-min*, which returns the minimum element in the stack. How efficient can you make these operations?

One option is to keep a normal stack where each element has two fields – the actual value, plus the minimum value of all the entries in the stack at or below the current point. Whenever we *push* an element, we look at the stack top to determine the minimum value so far. We then push a new pair consisting of the newly-added value and the minimum of that value and the global minimum. To *pop*, we just pop the top. To do a *find-min*, we just read the global minimum off the top of the stack. (This works because the only way the min changes is if we pop it off the stack or if we push a smaller element on top.)

Each operation runs in time O(1).

4. You are given a social network and a person in that network. Find everyone in that network who is not a friend of that person, but who is a friend of a friend of that person.

Run one round of BFS for two layers and look at all the people you find. Assuming we store the graph as an adjacency list, this runs in time O(F + FF), where F is the number of friends of the original person and FF is the number of friends of friends of that person.)

5. You are given a collection of n strings of total length L. Design an algorithm that determines whether any of those strings are prefixes of one another.

Add all the strings to a trie, one after the other. If during an insertion step we reach a node that's marks the end of a word, we return *true* because the word encoded by that node is a prefix of the current word. Similarly, if we insert a word without creating any new nodes in the trie, we can return *true* because we are tracing a prefix of some existing string. If neither case happens, we return *false*.

This runs in time O(n + L), since we process each character once and each string once.

6. (Challenge problem!) Solve problem (1) in time O(n) and space O(k).

This one's tricky and requires some content only covered in CS161. See if you can figure this one out for yourself! As a hint, look up selection algorithms.