EE 200 Exam 3

Tufts University

12 December 2019

SOLUTIONS

Question 1: The Big O table

What is the Big-O runtime for each operation below, for each of the data structures listed?

(a) [3 pts] Print the elements in order

Sorted linked list	BST	Heap
O(n)	O(n)	O(n log(n))

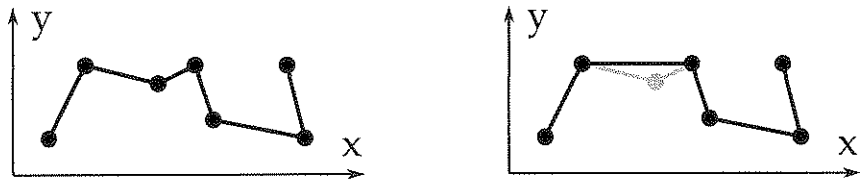
(b) [3 pts] Check if an element is already in the data structure

Sorted linked list	BST	Heap
O(n)	O(log(n))	O(n)

(c) [3 pts] Insert an element

Sorted linked list	BST	Heap
O(n)	O(log(n))	O(log(n))

Suppose you have a path on a 2D plane (e.g., in a graphics or mapping application) which is represented by a sequence of points defined by their (x,y) coordinates.



(d) [2 pts] What is the Big-O runtime to find the total Cartesian length of the path, as a function of the number of nodes?

We have to do a linear-time operation (find the distance to the next node) on each node, so the total time is $O(n)$.

(e) [3 pts] Suppose you want to simplify the path while roughly preserving its shape, by finding the point which can be removed with the smallest change to the overall path length. Describe an algorithm to do this. What is the Big-O runtime of your algorithm?

Scan through the sequence of nodes, and for each node, compute the change in path length if it were removed. This is a linear time operation, since it just requires constant-time math on the previous, current, and next node in the path. Once this is done, scan through and find the minimum-change node to remove and remove it. This is $O(n)$, although it has a much larger constant factor than just computing the distance.

A much worse way to do this would be to create a new path with one node removed, find the length of the path, and repeat. This would have time complexity $O(n^2)$.

Question 2: Choosing data structures

You've been hired to build a navigation app for a single interstate highway[1]. You are given a list of highway exits and their corresponding mileages (i.e., distance from the beginning of the highway), and the app should do two things as efficiently as possible:

- Given an exit, get the mile number of the exit.
- Given a mile number, find the nearest exit.

(a) [4 pts] What data structure(s) would you use to store this information, and why? *There isn't a single correct answer — I'm more interested in your logic for choosing a solution.*

One way to do this would be to have an array of the exits, sorted by distance.

(b) [1 pt] Using your solution, what is the Big-O runtime for looking up a mile number for an exit?

$O(1)$. Just index into the array with the appropriate exit, and read the mileage. If the exits aren't stored as sequential numbers, there could be a hash table mapping exit names to numbers used to index into thh array.

(c) [1 pt] Using your solution, what is the Big-O runtime for looking up the nearest exit given a mile number?

$O(log(n))$. We can do a binary search through the exit array to find the nearest exit.

The standard template library container type `std::set` implements an ordered collection of unique elements. Duplicates are not allowed, and the container is designed for fast insertion and retrieval of values. It is possible to iterate over the values in order, but arbitrary access by index is not allowed.

(d) [2 pts] What data structure would you use to implement `std::set`? Briefly explain your choice.

A hash table seems like a natural choice, but the requirement that the values be ordered rules this out. The next fastest option is a binary tree.

[1] Not a very useful app, but this is related to the more general problem of geocoding and reverse geocoding.

The standard template library type `std::stack` implements a last-in-first-out queue. Items are pushed onto the `stack` and come out in reverse order when popped off (i.e., the most recently pushed item comes off first, like a stack of dinner plates).

(e) [2 pts] Which of the following would be a good candidate to implement a stack? *Mark all that apply.*

- Array or `std::vector`
- Binary search tree
- Hash table
- Heap
- Linked list

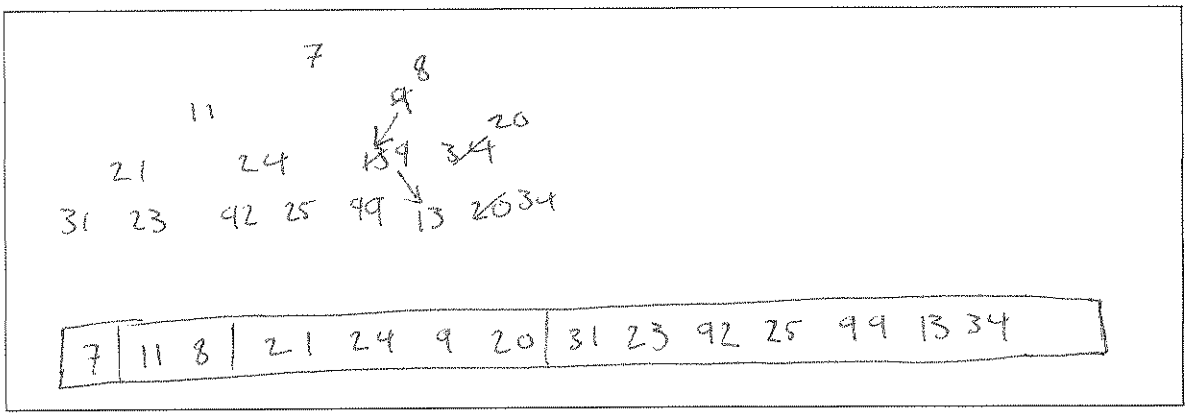
An array or `std::vector` would work well: just append and remove items from the end, and use a counter to keep track of the current top of the stack. A linked list would also work: new items are inserted and removed from the front of the list, which always gives O(1) access time.

Question 3: Heaps

Suppose you have a min-heap with 12 elements, represented by the following array:

7	11	9	21	24	13	34	31	23	92	25	99

(a) [4 pts] Sketch the heap array after the numbers 8 and 20 are added to the heap (in that order).



Question 4: Hashing

(a) [2 pts] Give two examples of why a hash function is useful besides a hash table.

Hash functions are used for:

- Storing passwords in a form that cannot be easily stolen
- Uniquely identifying a file or object (e.g., a git commit)
- Verifying that a file has not been tampered with or corrupted
- Making it hard to mine bitcoin

(b) [2 pts] Give at least two attributes of a good hash function.

- Deterministic, and produces the same hash for identical objects
- Reasonably fast to compute
- Produces large (chaotic) changes in hash value for small perterbations of the object
- Produces an even distribution of hashes for normal input data (i.e., close to a uniform random distribution)

Question 5: Sorting

For each of the sorting algorithms listed below, describe a scenario where you would chose that algorithm instead of the other options (i.e., explain the advantages it has over the others).

(a) [2 pts] Quicksort

Works fast, and works in place (no extra storage needed, other than a tiny bit of stack space for recursive calls). It can also be parallelized across multiple cores after the first pass. Choose this if memory usage is a concern, and the data is usually randomly distributed.

(b) [2 pts] Heapsort

Guaranteed performance of $O(Nlog(N))$, and works in place, avoiding the weaknesses of both Quicksort and Mergesort. However, it has a constant factor roughly $2\times$ slower than the others, so only choose it if the algorithm must work in place and the data is likely to be sorted in a way that makes Quicksort fail.

(c) [2 pts] Mergesort

Guaranteed performance of $O(Nlog(N))$ and can be parallelized across multiple cores. Choose this unless the space requirements are an issue (requires enough space to store the array twice).

(d) [2 pts] Selection sort

Although this is $O(N^2)$, it is a very simple algorithm and may be more efficient than a $O(Nlog(N)$ algorithm for very small N. Might be used as the first pass of mergesort or the last pass of quicksort, where $N < 10$.

Question 6: Threading and locking

Many operations on large arrays can be easily parallelized by simply splitting the operation into smaller tasks and running each on a separate thread. For example, we could use the code below to compute a histogram of characters in a very long string:

BAD WAY:

```
std::mutex mtx;

void worker(char* s, int length, int hist[256])
{

    for(int i = 0; i < length; i++){
        mtx.lock();
        hist[s[i]]++;
        mtx.unlock();
    }

}
```

BETTER WAY:

```
std::mutex mtx;

void worker(char* s, int length, int hist[256])
{
    int blockhist[256] = {0};

    for(int i=0; i<length; i++){
        blockhist[s[i]]++;
    }
    mtx.lock();
    for(int i=0; i<256; i++){
        hist[i] += blockhist[i];
    }
    mtx.unlock();
}
```

```
// For simplicity, we'll assume length is a multiple of MAX_THREADS
// and that the histogram array is already initialized
void compute_histogram(char* s, int length, int hist[265])
{
  int blocksize = length / MAX_THREADS;
  for(int t = 0; t < MAX_THREADS; t++){
    std::thread(worker, s + t*blocksize, blocksize, hist);
  }

  // Wait for all the threads to finish
  // ...
}
```

[4 pts] This code does not work very well. Write code between the lines (or cross out code as necessary) to make this code correctly compute the histogram.

If you need a mutex, you can use the `std::mutex` class, which is unlocked by default, and has three methods: `void lock()`, `void unlock()` and `bool try_lock()` (which attempts to lock the mutex and immediately returns `false` if it could not).

See above.

[2 pts] Bonus: Optimize your code above to get as much advantage from multithreading as possible.

See above.

In class we discussed the `std::atomic_int` type, for which certain operations (including `++` and `--`) are thread-safe. Your friend shows you their `Mutex` class, which is built using `std::atomic_int`:

```
class Mutex
{
```

```
public:
  Mutex() { mLock = 0; }
  lock() {
    while(mLock){} // Just wait until mLock is 0
    mLock++;
  }
  unlock() {
    mLock--;
  }

private:
  std::atomic_int mLock; // Use atomic_int here so the mutex works
}
```

[3 pts] Will this class properly work as a mutex? Why or why not?

> No, because although the increment and decrement operations are atomic, the test and set is not. The processor could be interrupted between leaving the `while(mlock)` loop and incrementing `mLock`.

Question 7: Bonus

[2 pts] What is one piece of advice you would give to a student just beginning EE 200?