

EE 200 Exam 3

Tufts University

13 December 2018

SOLUTIONS

Question	Points
Sorting	10
Choosing data structures	10
Heaps	5
Binary search trees	14
Threading and mutexes	6
Middle-pointer lists	6
Total	51
Bonus	2+2

Question 1: Sorting

There are three leading sorting algorithms with $N \log(N)$ runtime. Describe in one or two sentences how each algorithm works.

- (a) [2 pts] Mergesort

Works recursively from the bottom up. Each iteration splits the array, mergesorts each half, and then merges the two sorted subarrays.

- (b) [2 pts] Quicksort

Works recursively from the top down. Picks a pivot value and swaps elements so that all values on the left are less than the pivot and all values on the right are greater, then recursively quicksorts each half.

- (c) [1 pt] Heapsort

Turns the array into a min-heap, and then repeatedly pulls the minimum values off the heap to create the sorted array.

What are the disadvantages of each one? (i.e., why wouldn't you always choose one of them and ignore the others?)

- (d) [2 pts] Mergesort

Requires a separate array to put the sorted data in (i.e, it doesn't work in place and therefore requires $2\times$ the memory).

(e) [2 pts] Quicksort

Becomes $O(N)$ for particularly bad combinations of pivot choice and input data.

(f) [1 pt] Heapsort

Has higher constant factors than Mergesort or Quicksort — roughly $2\times$ slower.

Question 2: Choosing data structures

Suppose you're building the “contacts” application for a phone. You have a set of contacts with names and phone numbers, and you want to be able to do two things as quickly as possible:

- Search for a contact's phone number by name. That is, given a name, find the person's phone number.
- Do reverse lookup for caller ID. That is, given a phone number, find the name of the person calling.

(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.*

There are several options:

- Use a hash table mapping names to numbers. This is $O(1)$ for lookup, and $O(N)$ for reverse lookup.
- Use a hash table mapping numbers to names. This is $O(N)$ for lookup, and $O(1)$ for reverse lookup.
- Use two hash tables which are kept in sync. This gives $O(1)$ access both ways, but requires double the storage and bookkeeping.
- Use a tree organized by characters or digits, which allows for quick auto-completion and $\log(N)$ lookup.

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

(c) [1 pt] Using your solution, what is the Big-O runtime for finding a name given a phone number?

The standard template library type `std::unordered_set` implements an unordered collection of unique elements (like a Python `set`). That is, duplicates are not allowed, and items can only be inserted and removed by value.

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

This is a perfect application for a hash table: there is no need to order items, and a hash table provides $O(1)$ access. In fact, if you read the documentation for `unordered_set`, there are a few methods which expose details of the underlying hash table implementation.

The standard template library type `std::stack` implements a last-in-first-out queue. Items are pushed into 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).

(d) [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

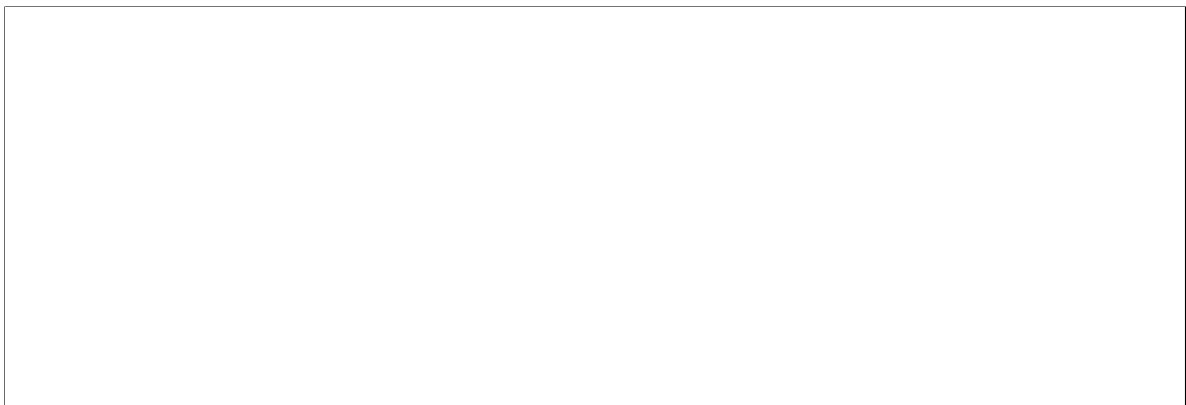
An array or `std::vector` would work well: just append and remove items from the end, and use a counter to keep track of the current top of the stack. A linked list would also work: new items are inserted and removed from the front of the list, which always gives $O(1)$ access time.

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) [2 pts] Draw the conceptual structure of the heap as a tree.



(b) [1 pt] How many elements do you need to examine to determine the minimum value in the heap?

Since the minimum value is at the root of the min-heap, we only need to examine a single element: the root.

(c) [2 pts] How many elements do you need to examine to determine the *maximum* value in the heap?

One approach would be to examine every item. However, we can do better, because any node in the heap which has children must be less than its children. Said another way, we need to examine all of the leaf nodes, which are 31, 23, 92, 25, 99, **and 34** — six in total.

Question 4: Binary search trees

A binary search tree class has the following (partial) declaration:

```
class BTree
{
public:
    // Destructor, which cleans up all memory
    ~BTree();

    // Other methods...

    // Assignment operator, which makes a deep copy of the tree
    const BTree& operator=(const BTree& rhs);

    class Node {
    public:
        Node(int val) { value = val; left = nullptr; right = nullptr; }
        int value;
        Node* left;
        Node* right;
    };

private:
    Node* mRoot;
    int mSize;
};
```

- (a) [4 pts] Write the destructor for the BTree class, which cleans up all of the memory from dynamically-allocated Nodes. You're welcome to write one or more helper functions if it makes your code simpler.

```
void delete_helper(BTree::Node* location)
{
    if(location){ // If this node isn't NULL, we have to clean it up
        delete_helper(location->left); // Do a postorder traversal
        delete_helper(location->right);
        delete location;
    }
}

BTree::~~BTree()
{
    delete_helper(mRoot);
}
```

- (b) [10 pts] Write the assignment operator for the BTree class, which makes a deep copy of the tree. Again, you're welcome to use helper functions (including any you wrote above) if it makes things easier. *Hint: one clean way to do this is with a recursive function that modifies a pointer – meaning you'll need to pass a pointer to a pointer. Hint 2: Don't get so caught up in the binary search tree details that you forget to do the assignment operator details correctly.*

```
void copy_helper(BTree::Node** dst, BTree::Node* src)
{
    if(src){
        *dst = new BTree::Node(src->value); // Pointers are null by default
        copy_helper(&((*dst)->left), src->left);
        copy_helper(&((*dst)->right), src->right);
    }
}

const BTree& BTree::operator=(const BTree& rhs)
{
    // Prevent self-assignment
    if(this == &rhs){
        return *this;
    }

    // Free the current tree, if it exists
    delete_helper(mRoot);
    mRoot = nullptr; // Not strictly necessary, but a good idea

    copy_helper(&mRoot, rhs.mRoot);

    // Finally, copy the size over
    mSize = rhs.mSize;
    return *this; // So that multiple assignment works
}
```

Question 5: Threading and mutexes

You are writing a multithreaded web server, with the skeleton code shown below:

```
typedef struct
{
    std::queue<PageRequest> requestQ; // Queue of page requests to be handled

    int requestsServed; // Count of the number of requests we've served
} ServerState;

void* serve_pages(void* s)
{
    ServerState* server = (ServerState*)s;

    while(1){
```

```

    if(server->requestQ.empty()){ // No requests at the moment
        sched_yield(); // Tell the OS to let another thread run
    }
    else{ // There is some request to handle

        PageRequest req = server->requestQ.front(); // Get the next request
        server->requestQ.pop(); // Remove it from the queue

        if(serve_web_page(req)){ // Try to serve the request

            // If the web page was successfully served, increment the total
            server->requestsServed++;

        }

    } // END else
} // END while(1)
}

int main(int argc, char* argv[])
{
    ServerState state; // Shared data for all the server threads
    pthread_t threads[16];

    for(int i = 0; i < 16; i++){
        pthread_create(threads+i, NULL, serve_pages, (void *)&state);
    }

    // Watch the network and put requests into the queue.
    get_requests(&state); // Like the serve_pages threads, this never returns.
}

```

- (a) [2 pts] Circle any lines or sections of code that could cause problems with multithreading.

See above. We need to protect both requestQ and requestsServed.

- (b) [4 pts] Write code between the lines to make sure that multiple serve_pages threads don't cause problems. Assume that someone else fixes get_requests to use the same locks you do.

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.

- (c) *Bonus* [2 pts] Assuming that the bulk of the time is spent in the serve_web_page() function, optimize your mutex code to get as much advantage from multithreading as possible.

To maximize performance, we need to make sure `serve_web_page()` is not part of the critical section. Otherwise only one thread could be serving a web page at a time, which would defeat the whole point of multithreading!

Since `requestQ` and `requestsServed` are independent, we can also use separate mutexes for them to reduce contention.

Question 6: Middle-pointer lists

Sorted linked lists have fairly poor performance for insertion and retrieval, so your friend proposes an improvement: a singly-linked list with a “middle pointer”, which is like a tail pointer but points to somewhere in the middle of the list.

When searching for items in the sorted list, the “middle-pointer list” can check the search key against the middle item. If the key is less, the search begins at the beginning of the list; if it is more, the search begins in the middle, saving a significant fraction of the search time.

What is the Big-O for the following operations on a sorted “middle-pointer list”?

- (a) [2 pts] Insertion into the sorted list

We have to search through up to half the list, assuming the middle pointer is really in the middle. This is $O(N/2)$, which without the constant factor is just $O(N)$.

- (b) [2 pts] Checking whether an item is in the list

Just like above, we half to linearly scan half the list, so this is also $O(N)$.

- (c) [2 pts] Retrieving the smallest item

The list is sorted, so the smallest item is at the head, with $O(1)$ access.

Question 7: Bonus

Write a poem or joke about any topic in the course. [2 pts]

Deadlock

Each of two threads got a lock;
The OS noticed the clock.
It switched the thread,
and now we're dead,
Because both threads are blocked.