Midterm II Next Tuesday

Today we’re going to be going over topics that will be covered on the exam. Ask questions if you have them. There will be a review session in Rashid Auditorium this Sunday at 6:30 PM. Take exams and read the lecture notes. Go even if you don’t have questions - someone else will.

**Things that are fair game for the midterm:**
- unbounded arrays
- amortized analysis
- hash tables
- interfaces
- priority queues
- heaps
- restoring invariants
- binary search trees
- avl trees
- polymorphism - à la (void *)
- memory management

**Things that will not be on the midterm**
- tries
- super in-depth Midterm I material

**Loop Invariants**

There are two steps to proving them correct:

1) **Init:** They hold before entering the loop

2) **Preservation:** If your loop invariants hold at the beginning of a loop iteration for assignables $x_1, x_2, \ldots, x_n$, then they hold at the end for $x'_1, x'_2, \ldots, x'_n$

**Interfaces**

We are constructing and manipulating various data structures using primitive datatypes like arrays, structs, ints, etc. Having an interface to these data structures allows us to consider the behavior of the data structure abstractly without worrying about the details of the implementation. For example, a queue can be implemented using a linked list, an array, two stacks, and many other ways.

Why do we consider different implementations of the same data structure?

**Amortized Analysis**

Operations on a data structure might not always take the same amount of time. For example, an operation could be $O(1)$ nine times out of ten, but then that tenth operation takes $O(n)$. We want to be able to talk about the time it takes to do $n$ operations on such a data structure.

Consider the example of a binary counter, where flipping a bit from 0 to 1 or 1 to 0 costs $O(1)$. What is the amortized cost of incrementing our binary counter, in terms of $n$?

**Method 1:** Let’s look at the way we flip bits in a binary counter. How often do we flip the $i^{th}$ bit?

<table>
<thead>
<tr>
<th>$i$</th>
<th>$2^0$</th>
<th>$2^1$</th>
<th>$2^n$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>4</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>5</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>6</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>7</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

From 00…00 to 11…11 we increment $2^n - 1$ times. We flip the $i^{th}$ bit every $\frac{1}{2^n}$ times.

Over $2^n - 1$ increments, bit $i$ is flipped $\frac{2^n}{2^n} - 1$ times.
So we have \[ \sum_{i=0}^{n} \left( \frac{2^n}{2^i} - 1 \right) = \left( \sum_{i=0}^{n} \frac{2^n}{2^i} \right) - n = 2^{n+1} - 2 - n \] flips.

Over \(2^n = 1\) increments, that's \(\frac{2^{n+1} - 2 - n}{2^n - 1} \leq \frac{2(2^n - 1)}{2^n - 1} - \frac{n}{2^n - 1} \in O(1)\) per increment.

Alternatively, how could we reason about this with a banker’s argument?

**Unbounded Arrays**

Unbounded arrays are a data structure with an interface that allows for getting and setting elements at their indices, adding elements to the end, and deleting elements from the end. We resize the data structure in a way that allows for add and delete to have constant amortized cost. How do we do that?

**Hash Tables**

Hash tables allow us to store a large number of **values** but still find them in (expected) constant time. Each value has an associated **key** that can be **obtained from the value**. We use a **hash** function to find where the key-value pair should be in the hash table. It is up to the client to supply the key-from-value and hash functions. We've discussed **linear probing** and **separate chaining** as hashing strategies. We can resize a hashtable when the **load factor**, ratio of stored values to underlying array size is too high.

**Binary Search Trees**

A binary search tree is a tree with left and right children that follow an ordering invariant such that for a tree \(T\):

- \(T\) is **NULL**
  - OR
- \(T\)'s left subtree (child) is either **NULL** or all the data in the ordered left subtree is less than \(T\)'s data.
- \(T\)'s right subtree (child) is either **NULL** or all the data in the ordered right subtree is greater than \(T\)'s data.

Keep in mind that BSTs are not necessarily balanced.

**Priority Queues**

Priority queues allow us to define a partial ordering on elements. We can access elements one by one, where the next element you get has the highest priority in the PQ. The PQ interface supports insertion and deletion, and sometimes just looking at the highest priority element.

In this class, we implement priority queues as min-heaps, which are arrays indexed at 1.

**Restoring Invariants: Heaps and AVL Trees**

When we insert into a heap or an AVL tree, it might be the case that we violate n one of the data structure invariants. That’s okay, as long as we are able to restore the invariants before returning from the function that breaks them.

**Polymorphism**

Polymorphism allows us to use data structures that store **ALL THE THINGS**.

But actually. In C, you can cast anything to (**void**) so if you have a data structure with data of type (**void**), that means you can really store anything. You just need to write functions supporting the data’s underlying type.

**Memory Management**

If you allocate it, you should free it.