15-122: Principles of Imperative Computation

Recitation 21a

Nivedita Chopra, Josh Zimmerman

Exam!

The exam is on Tuesday and there's a review session on Sunday (April 7) from 3:00-5:00PM in GHC 4401 (Rashid, where lecture meets). **Make sure to bring your student ID to the exam** and leave yourself extra time to get where you need to go, especially if you're in Newell-Simon. Here's the floor plan of NSH in case you need it: `https://www.cmu.edu/finance/property-space/floorplan-room/acad-admin/NSH/index.html`

Locations are as follows:

- Recitations A-E will go to Rashid as usual
- Recitation F will go to NSH 3305
- Recitation G students whose last name starts with A-Q will go to NSH 3305
- Recitation G students whose last name starts with R-Z will go to NSH 1507
- Recitation H will go to NSH 1507

Topics that will be covered

We'll emphasize the following topics on the exam, but we'll expect you to be familiar with what we covered earlier in the semester as well (for example, you should still know how to use loop invariants, arrays, etc.)

- Unbounded arrays
- Amortized analysis
- Hash tables
- Interfaces vs. implementations (Do NOT rely on a particular implementation of an interface)
- Binary search trees
- Priority queues
- Heaps (a way of implementing the priority queue interface)
- Restoring invariants (a library function may temporarily violate invariants and restore them before it returns)
- Backtracking search
- C

  Memory management (Both with `malloc` and friends and with local variables on the stack)

  Undefined behavior

  Macros

  `&` operator (address-of), `switch` statements, `structs` without pointers, casting.
- AVL Trees

Unbounded Arrays

Quick Recap

- The costs are amortized
- Cost of insertion and deletion is $O(1)$ amortized
- Double the size of the array when *size* == *limit*
- Don't reduce by half when it is half full - may lead to $O(n)$ operations
- While removing elements, resize when the array is a quarter full

Hash Tables

Quick Recap

- O(1) finding of an element
- Hash function must be randomized and deterministic
- Collisions occur when multiple keys have the same value
- Collisions are resolved by separate chaining or probing (linear/quadratic) - Focus on separate chaining
- Average chain length is the load factor μ
- μ = (number of keys in hash table)/(size of the underlying table)

Amortized Analysis

Quick Recap

- Used when you have different amounts of work at different steps and using Big O analysis doesn't give you a realistic bound
- Expressed as "O(x) amortized"
- Aggregate Method - Calculate the average over n steps and then use $n \to \infty$
- Accounting Method - Allocate tokens for each operation

Important examples include :

- Unbounded Arrays
- Binary Counter
- Queue as two Stacks - which we'll go over if we have time at the end of recitation

Heaps/Priority Queues

Quick Recap

- We usually deal with min-heaps
- The root is the minimum element
- Finding minimum element is $O(1)$
- Deleting minimum element is $O(\log n)$
- Inserting an element is $O(\log n)$
- **Ordering Invariant :** Any element in the heap is smaller than both its children
- **Shape invariant:** An element is inserted in a manner that at any point only the last level is unfilled and elements are filled in this level from left to right

Quick Exercise: Insert elements from 1 to 15 into a heap in such a way to get the smallest possible number in the last level:

Binary Search Trees

Quick Recap

- **Ordering Invariant :** states that every element in the left subtree must be less than the root and every element in the right subtree must be greater than the root.
- There is no shape invariant - so the wort case is a linked list

Quick Exercise: Construct a BST by inserting 4,2,3,1,5,7,6 in the given order

Write code to print out the elements in ascending order
(Assume that you have a print_elem function)

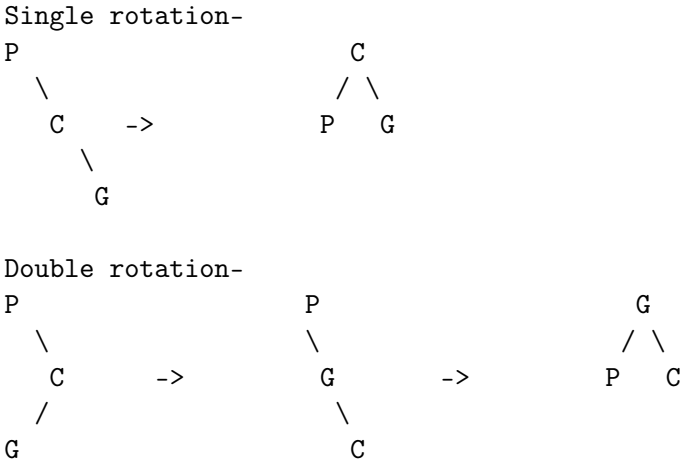
What was the cost of printing the entire tree in ascending order?

So we printed out the elements in sorted order in less than $O(n \log n)$ time!! Did we just come up with a more efficient sorting algorithm?

AVL Trees

Quick Recap

For unbalanced trees, where P is the node at which it is unbalanced:



Quick Exercise:

Is the tree in the previous part balanced? Is it left heavy or right heavy? Perform operations to balance the tree

Bactracking

- **Breadth-first**: Here we traverse a particular level before going to the next level.
  In the tree from the previous sections, this is the order:

- **Depth-first**: Here we simply go as deep as possible in the tree and if we don't find our desired solution, we come back up and follow the other another branches in a similar manner.
  In the tree from the previous sections, this is the order:

- **Best-first**: This follows a *heuristic*, which is a kind of rule that helps us estimate how likely it is that a particular branch will lead us to a solution.

Let's C

Quick Recap

- Most code that you wrote in C_0 is valid C code. So don't panic! There are some exception though, just be careful.
- Strings are represented as `char*` in C and are terminated by `'\0'`. Thus, to allocate memory for a string, you must allocate memory for one extra character.
- You can use macros, which are declared using `#define`
- You can create a pointer using `&` - the address-of operator.
- You can directly access elements of a struct using the dot operator.
- You can use `switch` statements.
- There is a lot of undefined behavior to pay attention to - refer to lecture notes and recitation handouts.
- `void*` is the generic datatype which we usually use to pass pointers around. These need to be cast into appropriate types before use.

Quick Exercise:

What does the following function do?
(Hint: In C, the assignment statement (=) returns the assigned value)

```
void mystery(char* p, char* q){ while (*p++ = *q++); }
```

Memory Management in C

Quick Recap

- You use `malloc` in C and give it a size as the argument
- You must `free` all memory that you allocate
- **The Golden Rule**
  Always `free` what you `malloc`
  Remember that C is not garbage collected
- Don't free memory that you didn't allocate. This can lead to undefined behaviour
- Usually you are only responsible for freeing memory that *you* have allocated.
  I like to think of this as

  number of `mallocs` = number of `frees`