CSCI 2270

Data Structures & Algorithms

Gabe Johnson

Lecture 20 March 4, 2013

Treaps

Lecture Goals

1. B-Tree Postmortem
2. Treaps
3. Function Pointers

Upcoming Homework Assignment

HW #6 **Due: Friday, Mar 8**

Priority Queues

Priority queues are a special type of Queue data structure in which values have an associated priority. When inserting, we place new nodes in the priority queue according to their priority. When removing, we always remove the highest priority node.

This week you'll need to design the struct yourself, and write test code to make sure it works. After B-Trees you should be able to do this in your sleep. *You will not get a driver implementation for this assignment*.

B-Tree Redux

Now that B-Trees are done, I want to know:

* Did you actually write your own test code?
* Did you do whiteboard/paper design?
* Did you do it in bitesize pieces?
* Did you collaborate?
* Did you read the web about how to implement?
* Did you find example source code?

... and if so, did these things help you substantially?

The B-Tree Head Trip

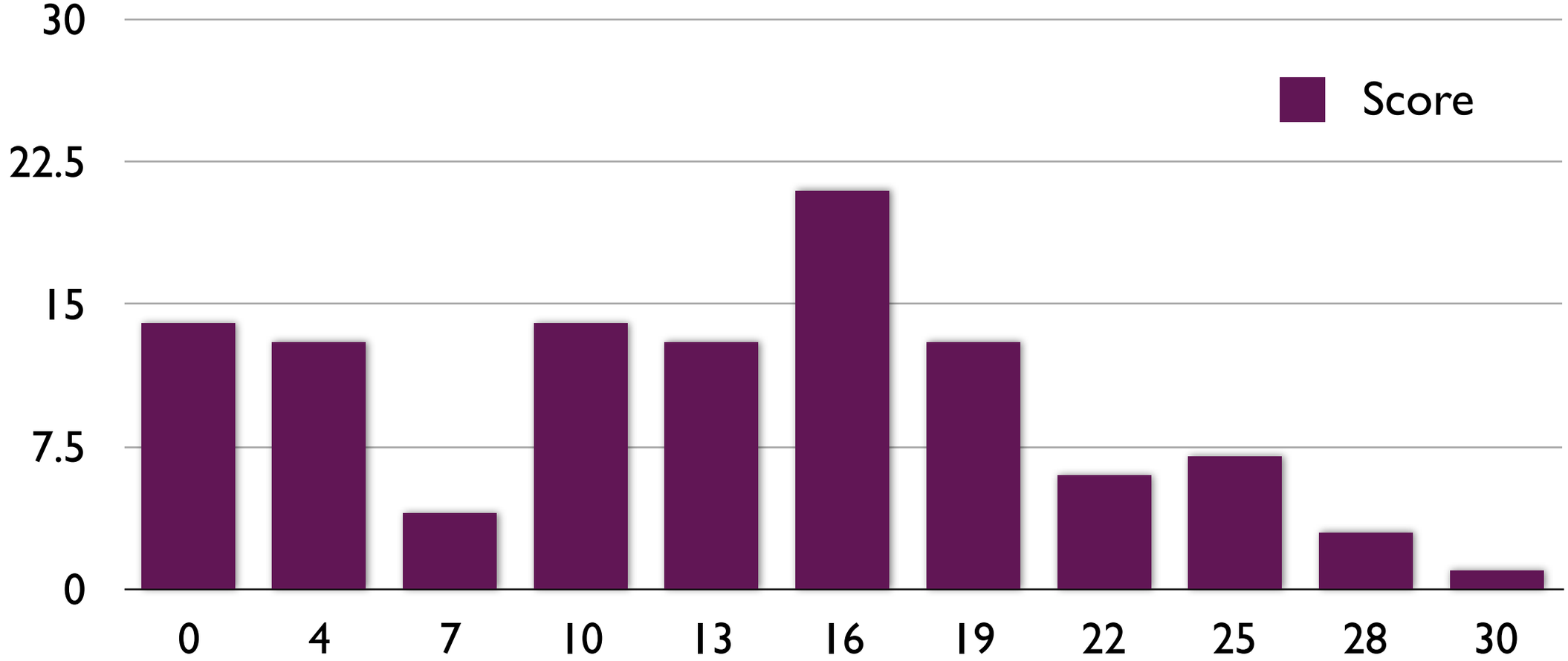
This wasn't really about B-Trees. I don't really care about that particular data structure, aside from it being really tricky to implement.

The part I care about, and the part I hope you learned, is that doing anything beyond moderate complexity requires advanced planning, testing, and a methodical process.

Of course you can mess around and (given enough time) come up with the solution. But in the real world, time is a scarce resource. You have to learn how to *design*.

B-Tree Scores

53 of you broke past 100%. The rest of you have until the final to get 100% (that's 15 points). Also: about 50 of you didn't turn anything in. Interesting.



Treaps

A Treap is a special kind of tree called a **randomized binary search tree**. The idea is that a normal binary search tree can be lopsided, with some paths from root to leaf being really long.

We've seen Red-Black trees, which use node coloring and a tricky sequence of rotations to keep the tree balanced. It turns out we can get almost the same efficiency by using random numbers, and combining BST semantics with Heap semantics.

Treaps.pdf

See Treaps.pdf on GitHub. Currently this is in HW 6, on Priority Queues. While you could implement a priority queue with a treap, a plain heap would probably be a better solution.

For this reason, this isn't really the best spot for this file and I'll move it somewhere later. **Mystery!**

Function Pointers

Say we want to do something that involves sorting numbers, but at runtime we might want to sort either in non-ascending or non-descending order, as we would in a priority queue.

We have those two functions defined: `get_larger` and `get_smaller`. They both take two integers and return an integer.

Function Pointers

```
// cs2270/code/func_pointers.cpp

int get_larger(int a, int b) {
  if (a > b) {
    return a;
  }
  return b;
}

int get_smaller(int a, int b) {
  if (a < b) {
    return a;
  }
  return b;
}
```

Function Pointers

The format to declare and assign a pointer to a function with that signature is like this:

```
int (*whatever) (int, int) = the_function_name;
 a      b          c    d              e
```

- **a**: the return type of the function.
- **b**: the name of our pointer to the function.
- **c**, **d**: a parenthetic comma-separated list of types the function accepts as input parameters.
- **e**: the name of the function we point to. Note that we OMIT the parens after the function name.

Functions As Params

```
int compare_with_func_pointer(int a, int b, int (*comparator) (int, int)) {
    // we get ‘normal’ integers a and b,
    // and a function pointer called ‘comparator’.
    // we know from the sig that it returns an int and accepts two integers.
    // we can call it like any other function:
  return comparator(a, b);
}
```

Here’s how you pass something to that function:

```
compare_with_func_pointer(30, 20, get_larger);
```

Silly String

Say we have a function like this:

```
string add_numbers_and_say_hi(int one, int two, int& result) {
  result = one + two;
  return "Hi!";
}
```

Here's how you would create and use a function pointer to it:

```
string (*silly_string) (int, int, int&) = add_numbers_and_say_hi;
  int sum = 0;
  string reply = silly_string(8, 13, sum);
  cout << "The silly string replied with " << reply
       << " and gives sum: " << sum << endl;
```

Hi!
21