Data Structures Lab

Learning to Prioritize
Assignment 3

- Due tomorrow night

- Focus on implementing a balanced search tree
  - Solving the diamond problem is secondary

- Make sure your files and methods are named correctly
  - BalancedTree.h, BalancedTree.cpp, assn3.cpp
  - insert, find, remove, print
Assignment 3 - Tips

- Build your tree incrementally
  - Start with a print method
  - Test each method as you write it

- Write insert and remove recursively
  - Makes balancing much easier

- Write rotation methods before balance method
  - Remember to test them

- I'll be in Descutes 100 today
  - But I'll be gone on Friday
  - If you have questions, ask early
Tracking Tree Height

● How do we know when our tree is out of balance?
  ○ Need to compute balance factor
  ○ Easy if you know subtree heights
  ○ But how do you compute them?

● How about a recursive solution?

```cpp
height(Node* curr) {
    h1 = height(curr → left)
    h2 = height(curr → right)
    return 1 + max(h1, h2)
}
```
Tracking Tree Height

- Recursively computing height takes $O(n)$
  - All operations should run in $O(\log n)$

- Seems like we're doing a lot of extra work
  - Let's keep track of height as we go
  - Only update when necessary

- Assume each node holds a height variable
  - Finding height is a $O(1)$ lookup
  - Can we update height in $O(1)$ as well?
Tracking Tree Height

- Let's update our recursive code

```cpp
height(Node* curr) {
    h1 = height(curr → left)
    h2 = height(curr → right)
    return 1 + max(h1, h2)
}
```
Let's update our recursive code:

```c
updateHeight(Node* curr) {
    h1 = curr → left → height;
    h2 = curr → right → height;
    curr → height = 1 + max(h1, h2)
}
```

What assumptions does this code make?
Tracking Tree Height

● Which nodes need to be updated?
  ○ Any node along an insert or delete path
  ○ Easy to access if we write insert/delete recursively

● At the end of each call to insert/delete:
  ○ Update the height of your node
  ○ Balance your node if necessary
Tracking Tree Height

Any issues here?

//node has two children
if (temp → left != NULL && temp → right != NULL){

    //find the in-order predecessor
    Node* & temp = getMax(curr->left);

    //swap up value and remove lower node
    curr → value = temp → value;
    remove(temp → value, temp);

    balance(curr);  //balance on the way up
}


Tracking Tree Height

- Also need to update height after rotation
  - Which nodes need to be updated?
  - Does the order matter?
Assignment 3 - Questions
Keeping your priorities straight

- Binary Search Trees maintain tree order
  - Any element can be found in $O(\log n)$
  - But what if you only want specific elements?
  - Can we find them faster?

- Priority Queue
  - Allows easy access to (largest / smallest / best) node
  - Pushes important elements to the front

- Our PQs will prioritize small elements
  - But you could use any comparator
Throw it on the heap

- One PQ implementation is the min-heap
  - Binary Tree
  - Root is the smallest element of the tree
  - Root of each subtree is smallest element in the subtree

- Easy to access the smallest element
  - It's always at the root
  - O(1)

- Hard to access arbitrary elements
  - Heap ordering doesn't facilitate searching
Throw it on the heap

Diagram:
```
  3
 / \
8   6
 / \/ \\/
9  12 13
```
Throw it on the heap
Throw it on the heap
Throw it on the heap
Heap Insertion

- How do we insert into a heap?
  - Blindly insert at the bottom
  - Worry about ordering problems later

- Bubble-up
  - Compare new node against parent
  - If new node is larger, we're done
  - If new node is smaller, swap values and repeat

- Advantages
  - Smallest nodes rise to the top
  - Tree remains balanced
  - Fast insertion time
Heap Insertion
Heap Insertion
Heap Insertion
Heap Insertion
Heap Removal

- How do we remove the root of a heap?
  - Swap it with the bottom node
  - Delete that one instead
  - Worry about ordering problems later

- Bubble-down
  - Compare root against children
  - If root is smaller, we're down
  - Otherwise, swap root with smallest child and repeat

- Inverse of Bubble-up
Heap Removal

```
  2
 / \/
8   3
/ \  / \  
9  12 13 6
```
Heap Removal
Heap Removal

Diagram of a heap structure:

- Node 6
  - Node 8
    - Node 9
    - Node 12
  - Node 3
    - Node 13
Heap Removal

```
  3
 / \
8   6
|
9---12---13
```
Heap Removal
Heap Implementation

- Heap trees are always completely full
  - We can implement this with an array

- Replace pointer manipulation with array arithmetic
  - Kind of the same thing...

- More on that next week
Homework 4

- Will be posted later today
- Due March 9th
  - Two weeks from tomorrow
- Implement Huffman Compression algorithm
  - File compression algorithm
  - Makes good use of priority queues
Huffman Compression

● How do you represent text?

● 8-bit char encoding
  ○ A - 01000001
  ○ B - 01000010
  ○ ...
  ○ Y - 01011001
  ○ Z - 01011010

● Fixed length codes are easy to parse
  ○ 01000011 01001111 01010111
  ○ C          O          W

● How many bits necessary to encode an n char string?
Huffman Compression

● Can we encode smarter?

● Some characters appear more frequently than others
  ○ What if we assigned them shorter codes?
  ○ Give other characters longer codes

● Lots of short codes outweigh occasional longer codes
  ○ A - 1000 (4 bits)
  ○ B - 10110 (5 bits)
  ○ ...
  ○ Y - 100110101 (9 bits)
  ○ Z - 1100101101 (10 bits)

● How many bits to encode ABA? ZZZ?
Huffman Compression

● Variable length encodings are tricky

● Suppose we encode as follows
  ○ A = 0
  ○ B = 1
  ○ C = 10

● What does 10 mean?
  ○ BA?
  ○ C?

● Character codes must have unique prefixes
  ○ Encoding for C can't begin with encoding for B
Huffman Compression

- Represent character codes with a binary tree
  - Characters must be on leaves
  - No character code can be a prefix of another

A = 00
B = 01
C = 10
D = 11
Huffman Compression

- Suppose a is much more frequent than other characters...

A = 0
B = 100
C = 101
D = 11
Huffman Compression

● So how can we optimally assign codes?
  ○ (Assuming we know each character's frequency)

● Huffman's algorithm runs as follows:
  ○ Create a node for each character
  ○ Combine two nodes with smallest frequencies
  ○ Repeat until only one node remains

● Use the resulting tree to produce character codes
Huffman Compression

A (0.4)  B (0.2)  C (0.1)  D (0.3)
Huffman Compression

A (0.4)  B (0.2)  C (0.1)  D (0.3)
Huffman Compression
Huffman Compression
Huffman Compression
Huffman Compression
Huffman Compression

Diagram:

- (1.0)
  - A (0.4)
  - (0.6)
    - (0.3)
      - B (0.2)
      - C (0.1)
    - D (0.3)
Huffman Compression
Huffman Compression

- So where do priority queues come in?
  - We only ever care about the smallest nodes

- PQs make Huffman Compression very efficient
  - Insert all nodes into a PQ
  - Remove the root twice to get the two smallest nodes
  - Insert a new node with their combined probability
  - Repeat until all nodes are gone
Assignment 4 - Huffman Compression

● Implement a Priority Queue
  ○ Elements should be binary tree nodes
  ○ Initially unconnected

● Compute character frequencies
  ○ Read in a file
  ○ Count character occurrences

● Apply Huffman algorithm
  ○ Produce character tree
  ○ Translate into character codes

● Write an encoder/decoder