Data Structures Lab

February 17, 2011
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
  - Tree.h, Tree.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
  ○ Come on by
Assignment 3 - Tips

- Does your tree need to account for double values?
  - That's what the problem is looking for...
  - But do we have to put them in our tree?

- Remember, your tree must support:
  - insert
  - find
  - remove

- How can we deal with doubles but keep our tree simple?
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?

\[
\text{height}(\text{Node}^* \text{ curr}) \{
  \text{h1} = \text{height}(\text{curr} \rightarrow \text{left}) \\
  \text{h2} = \text{height}(\text{curr} \rightarrow \text{right}) \\
  \text{return} \ 1 + \max(\text{h1}, \text{h2})
\}
\]
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

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

```
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

- 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
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 Diagram]

In the image, we have a heap structure with nodes labeled 9, 12, 13, 8, 3, and 2. The heap property is illustrated, where each parent node is greater than or equal to its children.
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
Heap Removal
Heap Removal
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

- Has not been posted yet
  - Will be by the weekend

- Due March 4th
  - Two weeks from tomorrow

- Implement Huffman Compression algorithm
  - File compression algorithm
  - Makes good use of priority queues
Huffman Compression

- How do you encode a file?
  - Assume 64 potential characters
  - Letters, numbers, symbols

- Characters can be identified with unique 6 bit codes
  - a = 000000
  - b = 000001
  - ...
  - y = 011001
  - z = 011010

- An n character file can be encoded in 6n bits
Huffman Compression

- Can't assign codes blindly
- If we encode as follows
  - a = 0
  - b = 1
  - c = 10
- What does 10 mean?
  - ba?
  - c?
- Character codes must contain unique prefixes
Huffman Compression

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

● Still need to adhere to prefix rule
  ○ Each character is a leaf in a binary tree
  ○ Higher frequency characters have lower depth
  ○ Lower frequency characters have greater depth
Huffman Compression

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

\[
\begin{align*}
    a &= 0 \\
    b &= 100 \\
    c &= 101 \\
    d &= 11
\end{align*}
\]
Huffman Compression

- So how can we optimally assign codes?
  - (Assume 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
Huffman Compression

- **a** (0.4)
- **b** (0.2)
- **c** (0.1)
- **d** (0.3)
Huffman Compression
Huffman Compression
Huffman Compression
Huffman Compression
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 a file encoder/decoder
Assignment 4 - Huffman Compression

- Assignment should be posted by this weekend
- This is a big assignment
  - I may provide some code (encoder/decoder)
  - Watch your email
- We'll talk about implementation next week
  - I understand this imposes a time constraint
  - The assignment deadline will be extended if necessary
Our data structures so far have had their data types hardcoded in:
  - CircularLinkedList only takes strings
  - BST only takes int

What if we want to have a generic type?

Templates to the rescue!
C++ Spotlight - Templates

- "template <typename T>"
  - instantiates a generic type T

```cpp
template <typename T>
T max(T a, T b){
    if (a > b)
        return a;
    else
        return b;
}
```

```cpp
int x = max<int>(5, 6);
Node n = max<Node>(Node(5), Node(6));
```
What about a templated class?
   We can do that too!

```cpp
template <typename T>
class Node{
    T data;
    Node* next;
};

template <typename T>
T Node::getData(){
    return this → data;
}
```
C++ Spotlight - Templates

- Each method of a templated class needs a template tag
  - Even if it doesn't use the template

```cpp
template<typename T>
class Node{
    T data;
    Node* next;
};

template<typename T>
int Node::getSize(){
    return (this==NULL) ? 0 : 1+size(next);
}
```