CmpSci 187: Programming with Data Structures
Spring 2015

Lecture #18, Using Binary Search Trees

John Ridgway

April 7, 2015

1 Using Binary Search Trees

Review: Balancing a BST

- As we've discussed, finding an element in a binary search tree takes worst-case time proportional to the height of the tree. The same is true of adding or removing an element.
- Ideally a BST with n nodes will be balanced, having a height of $O(\log n)$. But in the worst case (where every node has only one child) the height could be as large as n.

Review: Balancing a Tree

- How can we balance a tree? In CMPSCI 311 you will learn about self-balancing trees, in at least one of the several versions.
- These trees always have a height of $O(\log n)$ when their size is n.
- Adding and removing take $O(\log n)$ time, because there may be a restructuring step of $O(\log n)$ time after each such move to restore the balance.

Review: Balancing Effectively

- The right idea here is to study how we want the new tree to be arranged. Suppose we have used in-order to form the array, so that the array is sorted.
- If the tree is balanced, with an equal number of nodes on each side, the root node will be the middle element of the array. Its left child should be about 1/4 of the way along, and its right child 3/4 of the way, and so forth. This should suggest a recursion:

Code to Balance a BST

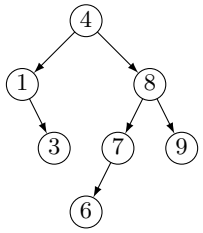
```
private void insertTree(T[] values, int low,
                        int high)
{
  if (low == high) {
    add(values[low]);
  } else if (low + 1 == high) {
    add(values[low]);
    add(values[high]);
  } else {
    int mid = (low + high)/2;
    add(values[mid]);
    insertTree(values, low, mid - 1);
    insertTree(values, mid + 1, high);
  }
}
```

Storing a Tree in an Array

- Remember that in general array-based structures may be faster to use than linked structures because we don't have to create new nodes all the time. We may also save memory by not using space for pointers.
- If our tree has a particular structure, we can store it in an array so that we don't need any explicit pointers at all. In particular, we would like our tree to be as balanced as possible.
- We declare that each node `A[i]` has a left child `A[2i+1]` and a right child `A[2i+2]`.
- So `A[0]` has children `A[1]` and `A[2]`, `A[1]` has `A[3]` and `A[4]`, and so forth. Finding a particular child of a particular node thus involves only arithmetic on the indices.
- But what happens if our tree is not very balanced?

Clicker Question #1

If we store this seven-node BST in an array using the rule just given, with the left child of `A[i]` in `A[2i+1]` and the right child in `A[2i+2]`, what is the address of the node containing 6?



A. `A[6]`

B. `A[7]`

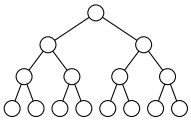
C. `A[11]`

D. `A[12]`

Full and Complete Trees

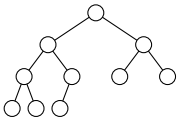
DJW call a binary tree **full** if all its leaves are on the same level. A full binary tree of height h has exactly $2^{h+1} - 1$ nodes, exactly 2^h of which are leaves. The terminology varies.

Full
Binary Tree



DJW call a binary tree **complete** if it is either full or full through its next-to-last level, with the leaves on the last level left-justified.

Complete
Binary Tree



Unbalanced Trees in Arrays

- An array of length n, with our implicit pointers, becomes a complete binary tree with n nodes.
- If we use the implicit pointers to store a tree of height h, we will need an array of size about 2^{h+1} no matter how many nodes are in the tree. The array will use space for all the nodes that are not there, on levels above the bottom level or left of the last node on the bottom level.
- If we wanted to store a BST in an array with implicit pointers, we might benefit from the balancing method in this lecture, though we would have to spend $O(n)$ time to balance a tree of n nodes.
- When we look at **heaps**, starting later in this lecture, we will see that they are always in the form of complete binary trees, and thus can be stored efficiently in arrays.

2 An Application: Word Frequencies

Word Frequencies

- We conclude the chapter on Binary Search Trees by using them to compute the frequencies of words in a text.
- We define a **word** to be a sequence of letters and digits, with a **delimiter** at each end — a space, line break, or punctuation mark. Our goal is to produce an alphabetical list of the words occurring in the text, with the number of occurrences of each word. We ignore the cases of letters.
- We also allow the user to control two parameters of the report.
  - There is a minimum word size so that we don't count frequencies of words below that size.

  - And in our report, we only list those words that occur at least some number of times, called the minimum count to report.

- Our basic object will contain a word and its count of occurrences (so far).

Choosing a Data Structure

- The basic idea is obvious; we will read through our text with a `Scanner` object, identifying each word that we see.
- We need to keep track of each unique word we have seen, each with its count.
- When we read a word, therefore, we need to determine whether we have seen it before, and if so how many times.
- We then either create a new word-count pair for a new word, or replace the existing word-count pair with another, incrementing the count.
- If we keep the word-count pairs in a collection, we need to test whether a given word is represented, retrieve its word-count pair if it is, and insert the correct new word-count pair for that word.
- We also need to output the words from the pairs in the collection in alphabetical order.
- DJW choose to keep the collection as a binary search tree, which can easily be output in order and can be searched quickly as long as it stays balanced.
- Of course our pairs will have to be compared in alphabetical order for the words, without reference to the frequency counts.

The Algorithm for Frequency

- During a run of DJW's program, it is keeping a collection of word-count pairs, and it is about to process the next word in the text file.
- If the word is too short, of course, we can just ignore it (except that we will keep track of the total number of words). But if it is long enough we must either create a new pair for it with count 1, or replace the current pair with another, incrementing the count.
- The `contains` method will tell us whether the word is already there, as long as the `compareTo` method looks only at the strings.
- The `get` method is even better, as it returns us a pointer to the pair in question if it is there.

- The first thing we would think to do is to remove this pair and add a new one with the new count. But we can save significant time by just *changing the count* of the existing pair, without moving the node.
- Changing the count of a pair doesn't affect where it belongs in the tree, since we never have two pairs with the same word.
- Of course if we don't find a pair for the word seen in the text, we know that it is a new word. It's easy to make a new pair with count 1 and insert it into the tree.
- So let's look at the data structure we need for the pairs.

Clicker Question #2

Suppose we feed the algorithm the text "In the beginning God created the heaven and the earth", with a minimum word size of two. What will be the height of the resulting binary search tree? (Remember that the height is the largest level number of any node.)

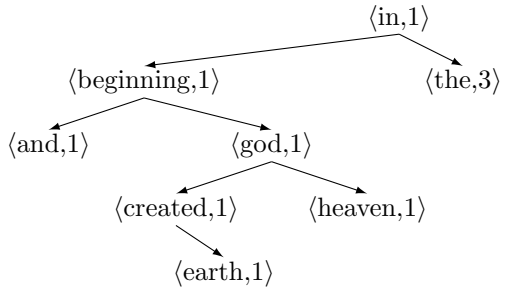
A. 2

B. 3

C. **4**

D. 5

Tree for Clicker Question #2



The `WordFreq` Class

- What are the operations we need for our pair objects?
- We need to create new objects, increment the count of an object, compare two objects alphabetically by word, and produce a formatted string to represent the pair.

Code for `WordFreq`

```
public class WordFreq
    implements Comparable<WordFreq>
{
  private String word;
  private int freq = 0;
  public WordFreq(String newWord) {
    word = newWord; }
  public void inc() { freq +=1; }
  public int compareTo(WordFreq other) {
    return this.word.compareTo(other.word); }
  public String getWord() { return word; }
  public int getFreq() { return freq; }
}
```

`FrequencyList` **Header and Setup**

```
public class FrequencyList {
  public static void main(String[] args)
      throws IOException {
    Scanner conIn = new Scanner(System.in);
    System.out.print("Minimum word size:");
    int minSize = conIn.nextInt();
    String skip = conIn.nextLine();
    System.out.print("Minimum word frequency:");
    int minFreq = conIn.nextInt();
    skip = conIn.nextLine();
    FileReader fin = new FileReader("words.dat");
    Scanner wordsIn = new Scanner(fin);
    wordsIn.useDelimiter("[^a-zA-Z0-9]");
    BSTInterface<WordFreq> tree =
        new BinarySearchTree<WordFreq>();
```

Processing the Text

```
    int numWords = 0, numValidWords = 0;
    while (wordsIn.hasNext()) {
      String word = wordsIn.next();
      // delimiter ensures that we get a word
      numWords += 1;
      if (word.length() >= minSize) {
        numValidWords += 1;
        word = word.toLowerCase();
        WordFreq wordToTry = new WordFreq(word);
        WordFreq wordInTree = tree.get(wordToTry);
        if (wordInTree == null) {
          wordToTry.inc();
          tree.add(wordToTry);
        } else { wordInTree.inc(); }
      } }
```

`FrequencyList`**: The Output**

```
    System.out.printf(
        "Words of length %d and above,\n" +
        "with frequency of %d and above:\n\n",
        minSize, minFreq);
    System.out.println(" Freq Word");
    System.out.println("----- ----------------");
    int numValidFreqs = 0;
    Iterator<WordFreq> iter=tree.inOrderIterator();
    while (iter.hasNext()) {
      WordFreq wordFromTree = iter.next();
      if (wordFromTree.getFreq() >= minFreq) {
        numValidFreqs += 1;
        System.out.printf("%5d %s\n",
            wordFromTree.getFreq(),
            wordFromTree.getWord()); } }

    System.out.println();
    System.out.printf(
        "%d words in the input file.\n" +
        "%d of them are at least %d characters.\n" +
        "%d of these occur at least %d times.\n",
        numWords, numValidWords, minSize, numValidFreqs, minFreq);
    System.out.println("Program completed.");
  }
```

Analyzing `FrequencyList`

- What is the running time of `FrequencyList`? The first question should be "in terms of what?"
- Let's let n be the number of characters in the text file. We will spend $O(n)$ time reading the file, but the rest of the processing depends on various characteristics of the file: how many words; how many distinct words; and so on.
- Let's say that there are $O(n)$ words in the file (since words are usually a constant length, this is probably true). The worst case is actually when the words are all distinct, because then we have $O(n)$ nodes in our BST.
- The unsuccessful searches take $O(\log n)$ time each as long as the BST stays balanced, so we have a total time of $O(n \log n)$.
- If the order of the words is random, the tree will probably stay relatively balanced (as you may see later in CMPSCI 311).

3 Introduction to Heaps

Introduction to Heaps

- We'll close this lecture with a brief introduction to **heaps**, a data structure we will use primarily to implement **priority queues**.

- The basic idea of a heap is to keep a collection of comparable objects in a **semi-sorted** state. The largest element will be at the root of the tree, and larger elements will tend to be nearer the top of the tree.
- Formally, a heap is a complete tree of comparable elements that satisfies the **heap property**: the element in every node is larger than (or equal to) the elements in its children (if any).
- Hence that element must also be larger than the elements in *any of its descendants*. In particular, the largest element must be at the root.

Clicker Question #3

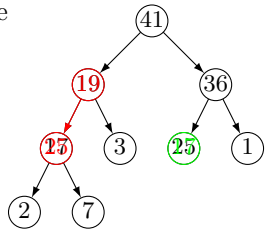
In this heap, which pair of elements could not be switched without violating the heap property?

A. **17 and 25**

B. 19 and 25

C. 3 and 7

D. 1 and 2



- Remember that a complete binary tree has its leaves on one level or on two adjacent levels; in the latter case the leaves on the upper level all exist and those on the lower level are left-justified.
- As we said, a heap is "somewhat sorted". We can find the maximum element quickly, but the farther down we go the less we know about the relative order of elements.

Heaps and Priority Queues

- The key virtue of the heap is that the two basic operations of a **priority queue** can be performed quickly, in $O(\log n)$ rather than $O(n)$ time where n is the number of elements in the heap.
- We need to insert new elements in places such that the heap property still holds.
- When we remove the root element we need to restore the heap property.

Clicker Question #4

Which nodes in a heap could possibly contain the fourth-highest element in the heap, assuming that no two elements have the same value?

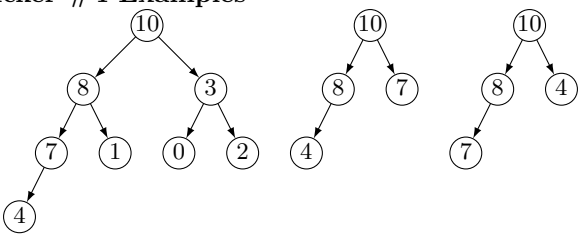
A. anywhere except the root

B. anywhere in level 0, 1, 2, or 3

C. in level 1 or 2, or in the left half of level 3

D. **anywhere in level 1, 2, or 3**

Clicker #4 Examples



Level 3 Level 2 Level 1

Any node on one of these levels could be fourth-high.