

Homework 2

Due February 28

Problem 2.1

A binary tree is **proper** if each node has either zero or two children. A binary tree that is not proper is **improper**. What are the minimum and maximum numbers of internal and external nodes in an improper binary tree with n nodes?

Solution: Consider a binary tree in which there are no left children. It is essentially a linked list. Such a tree would have $n - 1$ internal nodes and 1 external node (the bottom-most node). A tree must have at least one external node, so the minimum number of external nodes is 1 and the maximum number of internal nodes is $n - 1$.

Consider building a binary tree one node at a time. Each time we add a node, it will be an external node and it is added as a child to either a node with no children (an external node) or a node with one child (an internal node). In the former case its parent becomes an internal node, so the net effect is to increase the number of internal nodes by one. In the latter case the parent remains an internal node, so the net effect is to increase the number of external nodes by one. Thus to maximize the number of external nodes we should add a node as a child of a node that is already an internal node whenever possible. However, for the tree to be improper there must be at least one node with exactly one child. (Call such a node an “improper” node.) But if there are more than two improper nodes then we have needlessly missed an opportunity to add to an internal node. Adding a node always changes the number of improper nodes, so we must oscillate between having one and two improper nodes. When we have one improper node, we must add a node to an internal node, and when we have two improper nodes, we must add a node to an external node. Thus we alternate between increasing the number of external nodes by one and increasing the number of internal nodes by one. For even n there are an equal number of internal and external nodes. An improper three-node binary tree has two internal nodes and one external node, so in the case of odd n , there is one more internal node than there are external nodes. In general, the maximal number of external nodes is $\lfloor \frac{n}{2} \rfloor$ and the minimal number of internal nodes is $\lceil \frac{n}{2} \rceil$.

Problem 2.2

Provide an algorithm (in pseudocode) for computing the number of descendants of each node of a binary tree. The algorithm should be based on the Euler tour traversal. A leaf has no descendants (i.e., a node is not its own descendant).

Solution: See Algorithm 2.2.1

Algorithm 2.2.1 eulerDescendentCount(x)

```
count := 0
if  $x$  has a left child  $u$  then
    count := count + 1 + eulerDescendentCount( $u$ )
end if
if  $x$  has a right child  $v$  then
    count := count + 1 + eulerDescendentCount( $v$ )
end if
annotate  $x$  as having  $count$  descendents
return  $count$ 
```

Problem 2.3

(a) You've got a nonnegative integer n and a *sorted* (smallest to largest) array of real numbers $a[0] \dots a[n-1]$. You'd like search for the number p in the list (i.e., you'd like to find an index j with $a[j] = p$, or report *NOT IN LIST*). This corresponds to having a sorted hand of cards in a card game, and asking yourself "Do I have the 9 of hearts?". Describe (i) a linear-time algorithm for solving this problem, (ii) a $\log(n)$ time algorithm for solving the problem.

For part (ii), you may assume that n is a power of 2 (i.e., $n = 2^r$ for some integer $r > 0$).

Solution:

- (i) Iterate sequentially through each element of the array, checking to see if it is the item you are searching for.
- (ii) Write an algorithm for searching a range (specified by indices l and u for $l \leq u$) of a sorted array for a particular element p as follows:
 - if the range has length 0, (i.e., $l = u$), return **false**
 - examine the element in the middle of the range (i.e., $a[l + \lfloor \frac{u-l}{2} \rfloor]$)
 - if the middle element is the element you are looking for (p), return **true**
 - otherwise the middle element is either larger or smaller than p . In the former case, make a recursive call to the search algorithm, giving the sub-range up to (but not including) the middle element (i.e., from l to $l + \lfloor \frac{u-l}{2} \rfloor - 1$). In the latter case, make the recursive call on the sub-range from $l + \lfloor \frac{u-l}{2} \rfloor + 1$ to u .

The initial call to the algorithm will use the full range of the array (i.e., 0 to $n - 1$).

(b) You're given *two* sorted arrays $a[0] \dots a[n-1]$ and $b[0] \dots b[k-1]$ of real numbers, ordered smallest to largest. No number appears twice in either

list, and no number appears in both lists (i.e., they contain, in total, $n + k$ distinct numbers). We're going to find the r th smallest number. More formally, the TWOSELECT problem is this:

Given two sorted (in increasing order) arrays of real numbers of size n and k , with no duplicates among the $n + k$ numbers, and given a number r between 0 and $n + k - 1$, find the r th smallest number among the $n + k$ numbers in **a** and **b**.

If r is zero, the answer will be the lesser of **a**[0] and **b**[0], for instance.

(i) Freddie argues that this problem is at least as hard as the SELECT problem, in which you have to find the r th smallest entry in a list **a**. Here's his argument:

If you had a good algorithm for TWOSELECT, you could apply it to the case where **a** was your input array, and **b** had no elements at all. So if you can solve TWOSELECT faster than $O(n)$, then you can solve SELECT faster than $O(n)$. And if you *can't* do selection faster than $O(n)$, then you've got no hope of doing TWOSELECT faster than $O(n)$.

Explain what's wrong with Freddie's reasoning by carefully examining the hypotheses for the SELECT and TWOSELECT problems.

Solution: TWOSELECT requires that the arrays be sorted, which need not be the case for the SELECT problem.

(ii) Write an algorithm for TWOSELECT with running time $O(\log(n) + \log(k))$. Hint: part (a) of the problem will help you do this.

Note: this is "the hard problem" on this homework; it'll take you a while to come up with a solution. A deck of cards can be remarkably helpful in approaching the problem (sort the cards in increasing order, and then divide them randomly into two piles, i.e., deal 2 cards to the left, one to the right, 3 to the left, 1 to the right, 1 to the left, 3 to the right, etc.). It may be tempting, having seen RANDSELECT in class, to approach the problem using randomization. Resist the temptation; this problem is best solved without it, and the analysis to establish that the running time is $O(\log(n) + \log(k))$ will be much easier without having to worry about average running times.

Solution: First, consider solving the problem in which you know the answer will be found in array **a**. To solve this problem (TWOSELECT_HINT), do a binary search through **a** for the index i such that **a**[i] is bigger than exactly $r - i$ elements in **b** (a constant-time check because **b** is sorted). If such an index is found, return **a**[i].

To solve TWOSELECT, try the algorithm for TWOSELECT_HINT. If it fails, swap **a** and **b** and run the algorithm again. (This time, it's guaranteed to succeed.) In the worst case, the algorithm will be run twice, each time taking \log time in the length of the array where we are assuming the answer is.

Problem 2.4

XML and HTML are useful languages, but their plain-text representations can be difficult to read (for humans). *Pretty printing* is the process of formatting a structured plain-text language with line breaks and indentation so that it is easier to read. In this problem you are to give pseudocode for transforming a simplified XML into an indented text file. The simplified XML consists of three kinds of tokens: opening tags of the form `<html>` (where “html” may be replaced by other tag names like “head” or “h1”), closing tags of the form `</html>`, and plain text (also called character data, or CDATA). Assume there is a `getItem()` operation which gives you the next token in the stream. The token it returns has an `isTag` field (true for tags, false for CDATA), a `tag` field (contains the tag, if any) and a `cdata` field (which contains the character data, if any. Tags have an `isOpening` field, which is true for tags like `<html>` but false for their closing tags, i.e. `</html>`). You may also assume that there’s a function `match(tag1, tag2)` that returns `true` if tag2 is the closing tag that matches tag1, and false otherwise. Each element should be indented one tab more than its parent.

Your algorithm should take input like this:

```
<html><head><title>My Website</title></head><body><h1>My Website</h1>
<p>On the internet</p></body></html>
```

and output this:

```
html
  head
    title
      "My Website"
  body
    h1
      "My Website"
    p
      "On the Internet"
```

You may use the output primitives `output(token)` (which includes quotation marks around plain text), `tab()` which generates a single indentation, and `newline()`.

Problem 2.5

Give pseudocode descriptions of algorithms for performing the following three methods of a node list ADT, assuming the list is implemented as a doubly linked list:

addFirst(*e*) – inserts a new element *e* into the list as the first element

addLast(e) – inserts a new element e into the list as the last element

addBefore(p,e) – inserts a new element e into the list before position p

makeFirst(p) – moves element at position p so that it is the first element, leaving the relative ordering of the rest of the elements unchanged (must run in $O(1)$ time)

Also, draw pictures illustrating each of the major steps in each of the algorithms you provide.

Problem 2.6

Let L be a data structure that maintains a list of n items, ordered by decreasing access count. Suppose that all n items in L initially have access count 0. Describe a sequence of $O(n^2)$ accesses that will reverse L . An “access” amounts to invoking the operation “access(i)”, where i is a number from 0 to $n - 1$, indicating which element, in the current ordering of the list, you’d like to access. Thus if you started with

```
access(n-1)
access(1)
access(n-1)
```

you would have moved the last element to the front (in the first step), increased its access count (in the second step), and then moved the second-to-last item to be second in the list.

Problem 2.7

In class, Spike talked about the notation $\operatorname{argmin}_i b[i]$, which took an array b of real numbers, indexed by $i = 0, 1, \dots, n - 1$ and found which was the smallest, i.e., it represents an index p with the property that

$$a[p] \leq a[i] \text{ for all } i.$$

Write pseudocode (or code) that implements argmin . For example, you might produce code that looks like this:

```
int argmin(double b[])
{
    if b.length is zero, throw an exception.

    <some code here perhaps>
    for (int i = 0; i < b.length; i++){
        if (b[i] ... )
            <more code here>
    }
}
```

```
    <maybe more code>
}

return <something>;
}
```