CpSc 2120: Algorithms and Data Structures

Instructor: Dr. Brian Dean — Fall 2018
Webpage: `http://www.cs.clemson.edu/~bcdean/` — TTh 12:30-1:45
Handout 9: Quiz 1 In-Class Part. 25 possible points. — Earle 100

1. Debugging (5 points). For each of the three code excerpts below, please either state that the code is correct, or clearly identify one or more "bugs" and show how to fix them. Here, a "bug" is either a specific part of the code or a problem with its general design that causes the program to exhibit undesired behavior, such as potentially crashing, incorrect output, much slower performance than clearly intended, or leaking memory. Line numbers are provided for reference to help you refer to parts of the code, if needed (the line numbers are not themselves part of the code).

```
1. // CODE EXCERPT 1:
2. struct Node {
3.   int val;
4.   Node *next;
5. };
6. // return true if we find a value in a linked list...
7. bool find(Node *L, int value)
8. {
9.    Node *temp = new Node;
10.   if (L = NULL) return false;
11.   temp = L;
12.   while (temp->next != NULL) {
13.      if (temp->val == value) return true;
14.      temp = temp->next;
15.   }
16. }
```

```
1. // CODE EXCERPT 2:
2. // return true if array A[0..n-1] contains two duplicate values
3. // this is intended to run in O(n^2) time
4. bool does_array_have_duplicates(int *A, int n)
5. {
6.   bool found;
7.   for (int i=0; i<n; i++)
8.      for (int j=0; j<n; j++)
9.         if (A[i] == A[j]) found = true;
10.   return found;
11. }
```

```
1. // CODE EXCERPT 3:
2. struct Node {
3.   int val;
4.   Node *left, *right;
5. };
6. // return the depth of a binary tree (number of nodes on longest root-to-leaf path)
7. // the tree may have up to 1 million nodes and may not be balanced
8. int depth(Node *T)
9. {
10.   if (T == NULL) return 0;  // base case
11.   if (depth(T->left) > depth(T->right))
12.      return 1 + depth(T->left);
13.   else
14.      return 1 + depth(T->right);
15. }
```

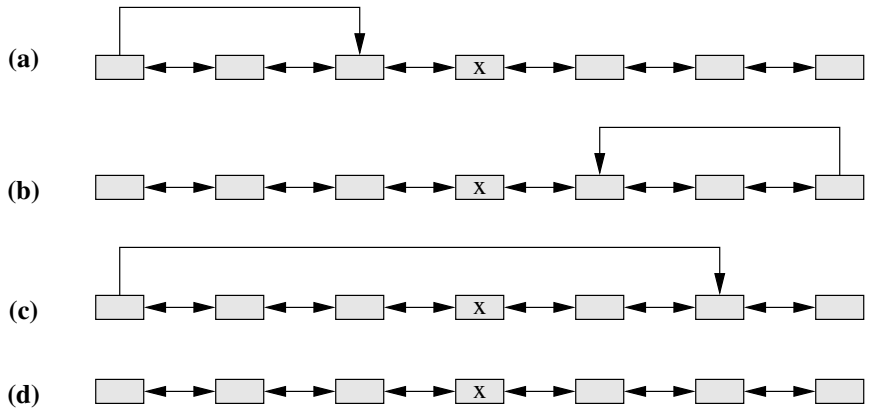

Figure 1: A doubly-linked list (a) with a loop at the beginning that does not contain x, (b) with a loop at the end that does not contain x, (c) with a loop containing x, and (d) with no loops. Note that (c) has a symmetric configuration where the last node points back to a node prior to x.

2. Loopy Lists (4 points). You are given a pointer to some node x in a doubly-linked list, with node structures that look like the following:

```
struct Node {
  int val;
  Node *prev;
  Node *next;
};
```

You want to know if your list contains a loop in any of the configurations shown in Figure 1. Using English, please concisely describe an efficient algorithm for testing which configuration is reality, and describe the algorithm's running time.

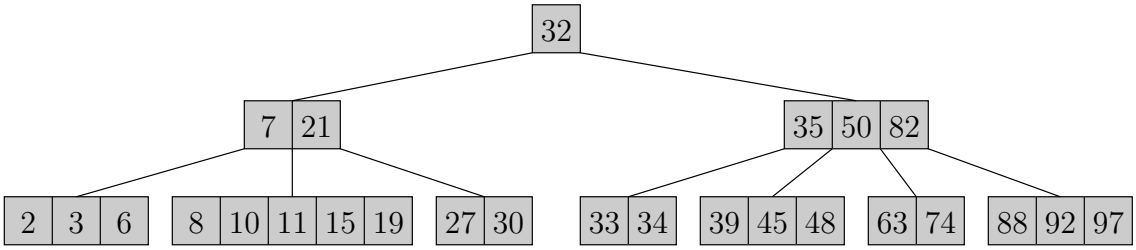

Figure 2: An example of a B-tree.

3. Select on a B-Tree (4 points).

Recall how B-trees generalize binary search trees, where each node has between B and $2B$ children (except possibly the root, which has no lower bound). Suppose each node in a B-tree is defined by the following structure:

```
struct Node {
  int nkeys;  // number of keys stored at this node
  int size;   // number of elements in this node's subtree
              // (including its own keys)
  int *keys;  // array of keys (of size nkeys)
  Node **children;  // array of pointers to child subtrees (of size nkeys+1)
};
```

Our goal in this problem is to implement the *select* function on a B-tree. Please complete the code below and state its running time.

```
// Returns the rth largest key (e.g., r=0 is the minimum) in a B-tree
int select(Node *root, int r)
{
```

4. Rank (4 points). This problem involves balanced binary search trees built from the following node structures:

```
struct Node {
  int val;
  int size; // subtree size
  Node *left, *right, *parent;  // note we have parent pointers
};
```

Please fill in the function below so that it returns the *rank* of an element in the tree in $O(\log n)$ time. Remember that the rank of an element is a number in the range $0 \ldots n-1$ giving its index within an in-order traversal. Your function is not given a pointer to the root of the tree, but note that you can tell if a node is the root if its parent is NULL. As a hint, consider what is the relationship between a node's rank and that of its parent. For full credit, use recursion; a non-recursive solution will earn at most 2/4 points.

```
int rank(Node *x)
{
```

5. Range Queries (2+2 points). You are given a balanced binary search tree with the following node structure:

```
struct Node {
  int key;
  Node *left, *right;
};
```

Please implement the following function *recursively*, which returns the number of nodes with keys between the keys of the lower and upper nodes (inclusive). For example, if your tree contains the keys 1, 2, 3, 4, 10, 11, 12, and lower's key is 3 and upper's key is 11, then the answer should be 4.

Please also describe the running time of your function. For partial credit, give the running time as a function of n, the size of the tree. For full credit, give a more precise running time bound that depends on n as well as the number of elements k returned by the function.

```
int num_elements_in_range(Node *root, Node *lower, Node *upper)
{
```

(b) Suppose your balanced binary search tree now has nodes defined as follows:

```
struct Node {
  int key;
  int size; // subtree size -- this wasn't here before
  Node *left, *right;
};
```

Please describe, in English, how to solve the same problem as in part (a) only in $O(\log n)$ time.

6. Tall People on the Left (4 points). You are given the heights $h_1 \ldots h_n$ of n individuals as input, as well as the locations $x_1 \ldots x_n$ at which they are standing along a one-dimensional number line. All h_i's and x_i's are distinct. Please describe an efficient algorithm that determines, for each person, if someone taller is standing on his/her left. Describe the running time of your algorithm.

This page may be used for scratch work.

This page may be used for scratch work.