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 11: Quiz 1 In-Class Part (Solutions). — Earle 100

1. Debugging (5 points). For each of the code excerpts below, please either state that the code is correct, or clearly identify one or more "bugs" and show how to fix them.

EXCERPT 1: Line 9 shouldn't have "new Node", line 10 should have == instead of =, line 12 shouldn't have "next", and "return false" should appear at the end.
EXCERPT 2: Found should be initialized to false, and the "if" statement should also check that i and j differ, since otherwise it will always return true when comparing A[i] to itself.
EXCERPT 3: The code is very slow (exponential worst-case time!) since it potentially recurses twice on the same child. There is a possibility of overflowing the stack. A non-recursive implementation would be better.

2. Loopy Lists (4 points). Scan forward through the list, storing the address of each node you encounter in a hash table. If you encounter node x again, you are in case (c), and if you encounter a node already in the hash table, you are in case (b). Otherwise, stop when you reach a NULL pointer and run the same algorithm in reverse, to detect if you are in cases (c) or (a). If the reverse algorithm also reaches NULL, you are in case (d). Running time is $O(n)$ anticipated with any reasonable hash function.

3. Select on a B-Tree (4 points). There are several ways to code this, most of which look something like the following, which runs in $O(B \log_B n)$ time (just as with most other operations on a B-tree).

```
int select(Node *root, int r)
{
  for (int index = 0; ; index++) {
    int childsize = children[index] ? children[index]->size : 0;
    if (r < childsize) return select(children[index], r);
    if (r == childsize) return key[index];
    r = r - childsize - 1;
  }
}
```

4. Rank (4 points).

```
int rank(Node *x)
{
   int left_size = x->left ? x->left->size : 0;
   if (x->parent == NULL) return left->size;
   if (x == x->parent->right) return rank(x->parent) + left_size + 1;
   else return rank(x->parent) - x->size + left_size;
}
```

5. Range Queries (2+2 points). The following runs in $O(k + \log n)$ time since it visits k nodes inside the range and up to at most $O(\log n)$ nodes outside the range:

```
int num_elts(Node *root, Node *lower, Node *upper)
{
  int answer = root->key>=lower->key && root->key<=upper->key; // 1 if root in range
  if (lower->key < root->key) answer += num_elts(root->left, lower, upper);
  if (upper->key > root->key) answer += num_elts(root->right, lower, upper);
  return answer;
}
```

For part (b), you can split on `lower->key-1` and `upper->key` to isolate the elements in the range into one subtree, then consult the `size` field on the root of this subtree (joining the subtrees back together afterwards to be polite). You could also just return `rank(upper) - rank(lower) + 1`.

6. Tall People on the Left (4 points). Sort on x using merge sort. Then scan through the people in sorted order keeping track of the tallest person seen so far. This lets us easily determine, for each individual we reach, whether they have someone taller on their left. The total running time is $O(n \log n)$ for the sort and $O(n)$ for the scan, or $O(n \log n)$ total.