CpSc 2120: Algorithms and Data Structures

Instructor: Dr. Brian Dean
Webpage: `http://www.cs.clemson.edu/~bcdean/`
Handout 7: Lab #5

Fall 2018
TTh 12:30-1:45
Earle 100

1 Storing a Sequence in a Binary Search Tree

In this lab, we will continue our study of the (balanced) binary search tree by investigating its second major usage: encoding an arbitrary sequence from "left to right" within its inorder traversal.

Recall, as discussed in lecture, that when we encode an arbitrary sequence $a_0 \ldots a_{n-1}$ within a BST, the *find* operation is no longer meaningful, since the traditional "BST property" no longer holds (i.e., elements in the left subtree are not necessarily less than the root, and elements in the right subtree are no longer greater than the root). Instead, we access the ith element of the sequence, a_i, by calling *select* to obtain a pointer to the element of rank i.

Today, we will start with the randomly-balanced BST we developed in the previous week's lab (which, to our advantage, already has a fully-functional *select* routine):

`/group/course/cpsc212/f18/lab05/bst_seq.cpp`

In this code, we have commented out the original *find*, *insert*, *remove*, and *split* functions, since these no longer interact with the BST in the right way.

2 Goal One: Insert Based on Rank

To insert properly into a BST that encodes an arbitrary sequence, we need to build a recursive *insert* function that inserts a new element by rank. The function is defined as follows:

```
Node *insert (Node *T, int v, int r)
```

It takes a pointer to the root of our tree, the value v to insert, and the rank r at which we should insert this value; like the previous version of *insert*, it should return a pointer to the tree that results from the insertion. If our tree encodes a sequence $a_0 \ldots a_{n-1}$, an insertion at rank r should place the new value immediately before a_r, so the new sequence encoded by the tree is $a_0 \ldots, a_{r-1}, v, a_r, \ldots, a_{n-1}$. To insert at the beginning of the sequence, we set $r = 0$. To insert at the end, we use $r = n$. To insert in the very middle, we would use $r = n/2$.

Please fill in the details of the new *insert* function. Since it now operates based on rank, it will be structured much like *select*. Some testing code is included in the *main* function to help you check correctness. Please think carefully about the recursive structure of your code (especially details like whether you should add or subtract one in certain places). It may help to draw a picture as a guide.

3 Goal Two: Maintaining Balance While Inserting Based on Rank

Just like last week, our *insert* function does nothing to maintain balance, and could result in a very imbalanced (and therefore slow) BST. To fix this, last week we also built a more sophisticated version of *insert* based on a randomized balancing mechanism: when inserting the nth element into a BST, it inserted that element at the root with probability $1/n$, and otherwise recursively just as before. Insertion at the root was done by first splitting the tree.

Your task now is to build a version of insertion *by rank* that uses the same randomized balancing mechanism as before. With probability $1/n$ (n being the new size of the tree after the insert), you should insert at the root, and otherwise you should insert recursively just as before. This will work much like your original *insert_keep_balanced* function from the previous section. To insert at the root, however, you will need to modify the *split* function so it splits on rank, rather than on value. As before, please think carefully about the recursive structure of your function, and don't be afraid to draw pictures!

4 Goal Three: Putting Everything Together to Solve a Fun Example Problem

Suppose n football teams, numbered $0 \ldots n-1$, all play each-other in a large tournament, where there are no ties. In your source file, you will find a function called

```
bool did_x_beat_y(int x, int y)
```

which returns true if team x won its game against team y, false otherwise. Your task is to compute an ordering of the teams, say $a_0 \ldots a_{n-1}$, so that team a_0 defeated team a_1, team a_1 defeated team a_2, and so on. Let's call such a sequence *valid*.

Note that there could be many possible valid solutions. For example, if team 0 beat team 1, team 1 beat team 2, etc., and team $n-1$ beat team 0, then any "cyclic rotation" of the sequence $0, 1, 2, \ldots, n-1$ (e.g., $3, 4, 5, \ldots, n-2, n-1, 0, 1, 2$) will be a valid solution.

The algorithm we will use to solve this problem is straightforward (the lab TAs will help explain it). It involves building up a sequence (stored in a balanced BST) to which we add teams $0 \ldots n-1$ one by one. Suppose we have built up a valid sequence of teams $0 \ldots k-1$ and we want to add team k in a position so that the entire sequence is valid. It turns out that there always exists such a position, and that we can find it quickly using our balanced BST.

If team k defeated the first team in our sequence, then we can just add it to the beginning. Likewise, if team k lost to the last team in the sequence, we can add it to the end. If neither of these cases holds, then let us imagine labeling the $k-1$ teams in our current valid sequence with "W" (if it wins against team k) or "L" (if it loses against team k). This gives a string of length $k-1$ containing Ws and Ls, where we know the first character W and the last is L. There must be some location where the string contains an adjacent pair of WL characters, and in between these is a feasible place to put our new team k. Moreover, we can *binary search* for such a position with only $O(\log k) \leq O(\log n)$ calls to *select*. How do we perform the binary search? Look at the middle character in the sequence. If it is "W", then we know the second half of the sequence starts with "W" and ends with "L", so it must contain somewhere a "WL" pair. On the other hand, if the middle character is "L", then the first half of the sequence starts with "W" and ends with "L", so it

must somewhere contain an adjacent "WL" pair. Since each call to *select* takes only $O(\log n)$ time with high probability, this gives a total running time of $O(n \log^2 n)$ to solve the entire problem.

Please implement the algorithm above in the function `order_n_teams` (some of the easy parts are written already for you, just to help get you started). In *main*, we have included testing code to see if your function returns a tree encoding a valid sequence.

5 Grading

For this lab, you will receive 8 points for correctness and 2 points for having well-organized, readable code. Zero points will be awarded for code that does not compile, so make sure your code compiles on the lab machines before submitting!

Final submissions are due by 11:59pm on the evening of Tuesday, October 2. No late submissions will be accepted.