Department of Computer Science
Trent University

COIS 3020H: Data Structures and Algorithms II

Assignment 2: Ropes

(100 marks)

(due Sunday, March 10, 2024 at 23:59 EST)

Note

Late assignments will be deducted 10% per day up to five days (including weekends).

Background

A string, by definition, is a linear sequence of characters representing a word, sentence, or body of text. Not surprisingly, strings are an integral and convenient part of most high-level programming languages. In C#, strings are supported by the `String` and `StringBuilder` library classes which include standard methods to concatenate two strings, return a substring, find the character at a given index, find the index of a given character, among others. Intuitively, each string is implemented as a linear array of characters which for the most part works reasonably well. For methods that retrieve characters or substrings, the linear array is ideal. But as a length of the string grows considerably longer as in the case of text, methods that require a wholesale shift or copy of characters are slowed by their linear time complexity. The question then arises: Can we do better overall for very long strings?

The **rope** data structure is one answer to this question. It is an augmented, full binary tree and is constructed in the following way. Suppose a string is broken down into substrings up to a fixed length. Each substring is stored in a node that is augmented with its length. The nodes are combined in pairs where each internal root of the tree stores the total length of the substrings represented by its left and right subtrees. Note that the substrings themselves are only stored at leaf nodes. If the tree is well-balanced, most methods can be implemented in $O(\log n)$ time which pays dividends when a string is very long.

One of the big advantages of the rope data structure is its support of immutable strings. An immutable string is one that cannot be modified. Therefore, any modifications result in the creation of a new string as in C#. However, until a modification is done, only one copy of the string is necessary. As soon as the string is modified, the changes are made and a new string is created. This is known as copy-to-write. The rope data structure can maintain the various versions of a string as an overlay which saves a great deal on space. For this assignment though, the string represented by the Rope class can be modified and is therefore, not immutable. But it is interesting to think about how a new string (rope) can be built upon a previous version without destroying the original.

Requirements

Your task is to design, implement, test, and document the following methods for the Rope class. The notation $S[i, j]$ is used to represent the substring that begins at index i and ends at index j. Assume as well that the maximum size of a leaf substring is 10.

The public methods of the Rope class closely mirror those of the `String` class of C#.

Rope(string S) : Create a balanced rope from a given string S (5 marks).

void Insert(string S, int i) : Insert string S at index i (5 marks).

void Delete(int i, int j) : Delete the substring $S[i, j]$ (5 marks).

string Substring(int i, int j) : Return the substring $S[i, j]$ (6 marks).

int Find(string S): Return the index of the first occurrence of S; -1 otherwise (9 marks).

char CharAt(int i) : Return the character at index i (3 marks).

int IndexOf(char c) : Return the index of the first occurrence of character c (4 marks).

void Reverse() : Reverse the string represented by the current rope (5 marks).

int Length() : Return the length of the string (1 mark).

string ToString() : Return the string represented by the current rope (4 marks).

void PrintRope() : Print the augmented binary tree of the current rope (4 marks).

The public methods are strongly supported by the following (and indispensible) private methods.

Node Build(string s, int i, int j) : Recursively build a balanced rope for $S[i, j]$ and return its root (part of the constructor).

Node Concatenate(Node p, Node q) : Return the root of the rope constructed by concatenating two ropes with roots p and q (3 marks).

Node Split(Node p, int i) : Split the rope with root p at index i and return the root of the right subtree (9 marks).

Node Rebalance( ) : Rebalance the rope using the algorithm found on pages 1319-1320 of Boehm et al. (9 marks).

Additional Optimizations

The ideal rope maintains a height of $O(\log n)$, but that can be quite a challenge. To help limit the height of the tree, try to implement the following optimizations.

1. After a Split, compress the path back to the root to ensure that binary tree is full, i.e. each non-leaf node has two non-empty children (4 marks).

2. Combine left and right siblings into one node whose total string length is 5 or less (4 marks).

Grading Scheme

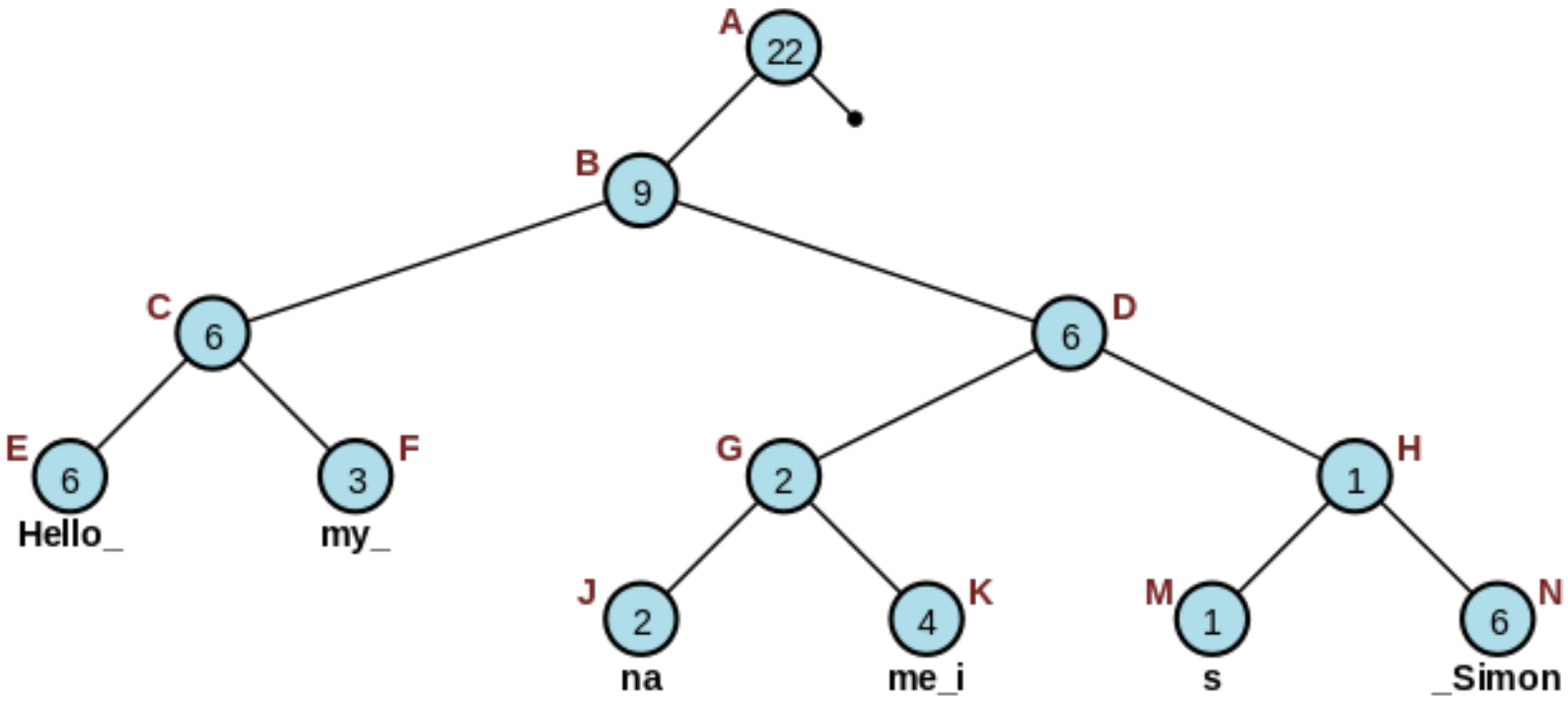
Rope methods	72
Additional Optimizations	8
Testing	15
Documentation	5

References

Hans-J. Boehm, Russ Atkinson, and Michael Plass, *Ropes: an Alternative to Strings*, Software - Practice and Experience, Volume 25(12), December 1995, pp 1315-1330

Dale King, *Ropes - Fast Strings*, January 2, 2017

Rope (data structure), *Wikipedia*, retrieved on February 12, 2024


A
22
B
9
C
6
D
6
E
6
Hello_
F
3
my_
G
2
H
1
J
2
na
K
4
me_i
M
1
s
N
6
_Simon