Department of Computer Science
Trent University

COIS 3020H: Data Structures and Algorithms II

Assignment 3: Branching, Lazing and Converting

(100 marks)

(due Sunday, April 7, 2024 at 23:59 EDT)

Notes

1. Late assignments will be deducted 10% per day up to five days (including weekends).
2. A discussion board has been set up to help you find new teammates.
3. Include references to:
   (a) any code that you found elsewhere or
   (b) any help, such as ChatGPT, that you used to produce code.

Part A: Point Me in the Right Direction (15 marks)

A point in d-dimensional space is defined as an d-tuple $(x_1, x_2, ..., x_d)$ where x_i represents the i^{th} coordinate of the point. For example, a point in 2-dimensional space is defined as (x_1, x_2) and often rewritten more familiarly as (x, y).

Suppose that d-dimensional points are stored in a variation of the quadtree called the point quadtree. When a new point is inserted, it is compared against the corresponding coordinates of the current point to decide which one of 2^d branches it should descend. For example, when a 2-dimensional point (x, y) is compared against the current point (u, v) then it can descend one of four branches corresponding to one of four possible cases.

$x < u$ and $y < v$ $\quad\quad$ $x < u$ and $y > v$ $\quad\quad$ $x > u$ and $y < v$ $\quad\quad$ $x > u$ and $y > v$

Note that the number of cases (branches) increases exponentially as d increases.

If branches are implemented as an array $B[0..2^d - 1]$ of nodes, write (12 marks) and test (3 marks) a short program that compares two d-dimensional points and maps the result to an index of B.

Hints:

1. Represent a point as an array of coordinates.
2. Think ”binary” when calculating the index.
3. Only one FOR loop and one IF statement is required.

Part B: Lazy Binomial Heaps (35 marks)

Each public method (`Add`, `Remove` and `Front`) of the binomial heap seen in class takes $O(\log n)$ time. However, if we relax the condition that at most one instance of each B_k can be part of the root list then the time to perform `Add` is reduced to a constant since no `Merge` is required. A binomial tree B_0 representing the new item is simply created and added to the root list of the current binomial heap.

The downside is that the `Remove` method takes $O(n)$ time after n items are inserted. That said, the `Remove` method can also be entrusted to clean up the mess left behind by `Add`. It does so by coalescing (combining) binomial trees so that the binomial heap is returned to having at most one instance of each B_k.

To support the `Coalesce` method, the data structure of the binomial heap is redefined as an array B of binomial nodes where $B[k]$ is a reference to a list of binomial tree(s) B_k (only). Also, to ensure that `Front` can execute in constant time, an additional data member maintains a reference to the item with the overall highest priority.

Re-implement the methods:

1. `Add` (2 marks),
2. `Front` (2 marks),
3. `Remove` (6 marks), and
4. `Print` (5 marks)

for the lazy binomial heap. Also eliminate the methods `FindHighest`, `Merge`, `Union`, and `Consolidate` and replace with a single private method called `Coalesce` (9 marks).

Note: By re-implementing the binomial heap in this way, the amortized cost of `Remove` is $O(\log n)$ and the worst case behaviour of `Add` and `Front` is reduced to $O(1)$. Just by being lazy!

Other marks:

1. Testing (8 marks)
2. Source code documentation (3 marks)

Part C: 2-3-4 Trees to Red Black Trees (50 marks)

The red-black tree is another variation of the binary search tree. It uses augmented data (colour) to balance the tree and ensure that the methods to insert, remove, and find an item are performed in $O(\log n)$ time in the worst case. To meet these objectives, a red-black tree satisfies the following properties.

1. Every node is either red or black.
2. The root node is black.
3. All NIL nodes are considered black.
4. A red node does not have a red child. Therefore, no two red nodes will appear consecutively along any path from the root.
5. Every path from a given node to any of its descendant NIL nodes goes through the same number of black nodes. Therefore, if a node N has exactly one child, the child must be red. Otherwise, if the child were black, its NIL descendants would sit at a different black depth than N's NIL child, violating property 4.

For this part of the assignment, however, your task is **not** to implement the red-black tree, as we shall see.

In class, we learned that a 2-3-4 tree is a B-tree where $t = 2$. But more interestingly, for every 2-3-4 tree, there exists exactly one red-black tree with data elements in the same order. Hence, 2-3-4 trees and red-black trees are considered equivalent data structures. You can transform one into the other and vice versa.

Your task for this part is to demonstrate the equivalence between 2-3-4- trees and red-black trees in a separate document (5 marks) and to define a generic class for the 2-3-4 tree that implements the following public methods. Other private methods may be defined.

TwoThreeFourTree( ) which initializes an empty 2-3-4 tree. (2 marks)

bool Insert(T k) which returns true if key `k` is successfully inserted; false otherwise. (6 marks)

bool Delete(T k) which returns true if key `k` is successfully deleted; false otherwise. (10 marks)

bool Search(T k) which returns true if key `k` is found; false otherwise (4 marks).

BSTforRBTree<T> Convert() which builds and returns the equivalent red-black tree. For this assignment, the red-black tree is represented as an instance of the class BSTforRBTree. The code for BSTforRBTree is found on Blackboard under Assignments. Remember to remove the class `Program` before using in your code. (8 marks)

void Print() which prints out the keys of the 2-3-4 tree in order. (4 marks)

Other marks:

1. Testing (8 marks)
2. Source code documentation (3 marks)