Assignment 3: Basic Data Structures

UMaine COS 350: Data Structures and Algorithms

Assigned: 21 February 2020 Due: 6 March 2020

Each of the following has instructions about what to turn in for the problem. You will turn in a single zip file containing all of these files, and *only* these files (Mac users, I'm looking at you). Your file **must** be named using the convention "*lastnamef*.zip", where *lastname* is your last name, no capitalization, and *f* is your first initial. Since I will likely be grading portions of these automatically, it is critical to follow these directions and to ensure that your zipfile, when run through unzip, will produce a directory (folder) structure like the following (assuming the zipfile is called `turnerr.zip`):

```
turnerr/
   quicksort.py
   counting_sort.py
   ...
```

1. Warm-up:

   (a) Using Python, implement quick sort.

   - Function name: `quicksort`
     - one parameter, a Python list or tuple containing numbers;
     - returns a sorted array based on the input
   - Turn in:
     - a plain text file named "`quicksort.py`" containing well-documented Python code
     - a plain text file named "`quicksort_output"` showing your program running on sample input

   (b) Write a counting sort using the algorithm in the book (and slides).

   - Function name: `countSort`
     - one parameter, a Python array or tuple containing numbers
     - returns a sorted array based on the input
   - Turn in:
     - a plain text file named "`counting_sort.py`" containing well-documented Python code
     - a plain text file named "`counting_sort_output"` showing sample runs

   (c) Compare your quicksort and counting sort programs. Gather data from both for a variety of values of n, with multiple runs each such value.

   - Turn in: a PDF file named "`comparison.pdf`" that describes what you determined.
     - This file should contain a plot showing the behavior of the two programs.
     - Any conclusions you make comparing the two programs should be backed up with statistics.

- Any conclusions you make about the running time of either program should be backed up by a ratio or power test, statistics, etc.

2. B-trees.

   (a) Implement a general B-tree Python class named `b_tree` based on the discussion in the book.
   - Your class should support inserting, deleting, searching, and in-order traversal.
   - Make the max number of children per node be a parameter that can be set when creating a new instance of `b_tree`.
   - Your book discusses B-trees and their implementation pretty thoroughly, including pseudocode for most (but not all!) operations.
   - Note, however, that the discussion is in the context of B-trees organizing data on disk, not necessarily in memory. Keep the calls to `DiskRead` and `DiskWrite`, but have them just increment counters in the B-tree class so that you can analyze the behavior that would have resulted had this been dealing with disk access.
   - To run your program, the following should work to create a B-tree with 7 children per node and sort an array of numbers using it:
     ```
     nums = [10, 1, 5, 15, ...] # some array of numbers
     b = b_tree(7)
     b.insert_list(nums)
     b.inorder()
     ```
     The result should be a sorted version of `nums`.
   - Turn in:
     - a plain text file named "`b_tree.py`" containing well-documented Python code
     - a plain text file named "`b_tree_output`" showing sample runs of your program
     - a PDF file named "`b_tree.pdf`" that discusses the performance of your program, including how it would perform if the disk reads and writes were real; can you also say anything about how your program (if the disk reads/writes were real) would interact with the computer's virtual memory system?

   (b) Implement a subclass of `b_tree` called "`b_2_4_tree`" that works for 2-4 trees.
   - To run your program, the following should work to create a 2-4 tree and sort an array of numbers using it:
     ```
     nums = [10, 1, 5, 15, ...] # some array of numbers
     b = b_2_4_tree()
     b.insert_list(nums)
     b.inorder()
     ```
     The result should be a sorted version of `nums`.
   - Turn in:
     - a plain text file named "`b_2_4_tree.py`" containing well-documented Python code
     - a plain text file named "`b_2_4_tree_output`" showing sample runs of your program
     - a PDF file named "`b_2_4.pdf`" that discusses the performance of your program and what changes had to be made (if any) to implement 2-4 trees. Do you think the disk read/write calls are necessary for 2-4 trees? If not, why? If so, under what conditions?