Exercise Set 5: Algorithm Analysis: Stacks, Queues, Binary Search Trees

Exercise 5.3 modified Wednesday, May 7 2003 at 6:09 pm.

Exercise 5.1 The goal of this exercise is to run some tests on two different implementations of stack interface, to determine their expected running time. The Java `Stack` interface has the following methods:

- `void push(Object obj);`
- `Object pop();`
- `Object peek();`
- `boolean empty();`
- `int search(Object obj);`

The `pop()` method decreases the size of the stack by one, or, when called with empty stack, it raises `EmptyStackException`. Look up the Java documentation for further details.

1. Implement the `Stack` interface as a singly linked list of `City` data in the class `LinkedStack`.
2. Implement the `Stack` interface as an array of `City` data in a class `ArrayStack`. If the original size of the array is not sufficient, the `ArrayStack` class should define a new array that has double the length of the original array, copy all data into it, and make it the new buffer. Additionally, if the size of the data in the array is less than 1/3 of its capacity, decrease the size of the array by 1/2 (again copying the data into a new location.
3. Design an algorithm tester class, which generates a given number of random test requests: push, pop, peek for the given stack and records the time needed to complete the test suite. (*Note: the sequence of the requests is generated at random, the data to be pushed is retrieved from the file using the* `FileRange` *iterator.*)
4. Use this class to run a series of timing tests which compares the two implementations for 100, 1000, 10000, and 30000 requests.

Exercise 5.2 The goal of this exercise is to run some tests on two different implementations of queue interface, to determine their expected running time.

We define the Queue interface to consist of the following methods:

- `void enqueue(Object obj); // insert object at the end of the queue`

- `Object dequeue(); // remove object from the front of the queue`
- `boolean empty(); // determine whether the queue is empty`
- `int size(); // the size of the queue`

1. Implement the `Queue` interface as a doubly linked list of `City` data in the class `LinkedQueue`.
2. Implement the `Queue` interface as an array of `City` data with circular buffer in the class ArrayQueue. If the original size of the array is not sufficient, the `ArrayQueue` class should define a new array that has double the length of the original array, copy all data into it, and make it the new queue. Additionally, if the size of the data in the array is less than 1/3 of its capacity, decrease the size of the array by 1/2 (again copying the data into a new location.
3. Design an algorithm tester class, which generates a given number of random test requests: enqueue, dequeue for the given stack and records the time needed to complete the test suite. (*Note: the sequence of the requests is generated at random, the data to be pushed is retrieved from the file using the* `FileRange` *iterator.*)
4. Use this class to run a series of timing tests which compares the two implementations for 100, 1000, 10000, and 30000 requests.

Exercise 5.3 The goal of this exercise is to run learn how to work with linked implementation of binary trees.

A binary search tree `BST` class contains one member data object root of the class `TreeNode`. `TreeNode` is a structure of `City data`, and the `left` and `right` `BST`. The `BST` class has the following API:

- `BST(Comparator c) // constructor is told how to compare`
- `void insert(Object obj) // insert an object into the BST`
- `boolean empty() // determine whether the tree is empty`
- `String toString() // convert BST to a printable string`
- `boolean find(Object obj) // determine whether this object appears in the tree`
- `boolean remove(Object obj) // remove given object from the BST; return false if not found`

The code for the constructor and the first three methods is given to you.

1. Study the design of the class `BST` and the `TreeNode` class.

2. Design the method `remove`, which removes the given object from the tree.

3. Design the method `countNodes`, which counts the number of nodes in the tree.

4. Design the method `height`, which determines the height of the binary tree (the largest sequence of tree branches).

5. Design a test class, which creates a given number of BST's of the given size, and computes the average and maximum height of the trees. The input data for the sample trees comes from the file.

6. Run tests on trees of size 100, 1000, 10000 and print your results.