CMPT-225 Assignment 5

Instructor   Jan Manuch

Deadline: Friday, July 21, 2006 at 4pm

The assignment is worth 5% of the final grade. Answer the last 2 questions on the paper and submit them to the drop-in box with a label CMPT-225 located opposite to CS General Office. For the **first** question write a program in Java or C++ and submit using online submission server: **submit.cs.sfu.ca**.

1. (3 points) Write a program in Java or C++. The task is to implement ADT Sorted List as a derived class from ADT List. For the ADT List (with the familiar interface) use generic array based implementation provided (ListInterface.java, ListException.java, ListIndexOutOfBoundsException.java, List.java) with this document. The ADT Sorted List should have the following interface:

```
public boolean isEmpty();

public int size();

public void add(int i, T item);
// Precondition: 1 <= i <= size()+1
// Postcondition: The size of the list is incremented by 1 and the
// the number of elements ‘item’ has increased by 1. The position i
// is a recommended position where to start looking for position
// where to insert ‘item’. It can be ignored.

public T get(int i); // corrected return type to T
// Precondition: 1 <= i <= size()
// Postcondition: The i-th smallest item in the list is returned.

public void removeAll;
// Postcondition: List is empty.

public void remove(int i);
// Precondition: 1 <= i <= size()
// Postcondition: The i-th smallest item in the list is removed
// from the list.

public void remove(T item);
// Precondition: none
// Postcondition: All occurrences of ‘item’ are removed from the list.

public int count(T item);
// Precondition: none
// Postcondition: Returns the number of occurrences of ‘item’
// in the list.
```

Note that all methods except last two are present in ADT List, although some of them have different meanings. For sure, you need to implement the last two methods. For the remaining methods, you might need or might not need to override the method from class List depending on your implementation. The interface for the new class is in the file SortedListInterface.java. Note it contains only the new method and inherits remaining methods from ListInterface. Your class implementing ADT Sorted List should be called SortedList.

Requirements:

- Class SortedList should be derived from class List, should be generic and should implement interface SortedListInterface.
- elements of sorted list have to be stored in underlying (superclass) list. You are not allowed to store elements in some other data member, even if it would be of type List.
- elements should be compared with compareTo() method.
- if you decide to store items in the underlying ADT List in sorted manner (see bellow), implement method add() using the **binary search** algorithm which will start searching not in the middle position of the array but at position i (i is the first parameter to the method).

Recommended implementation: keep the elements in underlying list in sorted manner. That way you don't need to override get() and the original version of remove(), but you have to override add() so that the element is inserted to a proper place. When searching for a proper place where to insert the new element, you should start looking from the position recommended by the first argument of add(). For instance, if you would use binary search to find the position, you could pick the first position you are going to test to be the recommended position.

Other possibility, is to leave add() as it is, i.e., let it insert elements to recommended place, but override methods get() and the original version of remove() so that they will first sort the list and then return the i-th smallest element.

Remarks:

- you can use auxiliary helping methods when needed, as long as they are declared private.
- for Java programmers: all code you write should be located in the file SortedList.java. You should not modify other files, with exception of Tester.java, which will be replaced with another file when testing.
- The recommended position passed as the first parameter to add() does not have to be meaningful. That means no matter strategy you use for implementation, you should first check if the value is in the range 1..size()+1. You should not throw exception in this case. If it's smaller than 1 than replace it with 1, if it's greater than size()+1, replace it with size()+1. Then use this corrected value: in the first strategy as a starting value for binary search, in the second strategy, as an actual parameter you pass to add() of the superclass (i.e., also in the second strategy you have to override add() method, but the code is very simple, just correct the value of the position as described above).

Hints:

- your implementation will be tested using test file similar to Tester.java.

- the declaration of class SortedList should look something like this:

```
public class SortedList<T extends Comparable>
extends List<T> implements SortedListInterface<T> {
```

For C++ programmers:

The supporting code which comes with this assignment is located in the following files: ListException.h, ListIndexOutOfRangeException.h, List.h, SortedList.h, Tester.cpp.

Remarks:

- The implementation for ADT List is located in the header file List.h. In C++, the definitions of the methods of a template class have to be located in the same file as definition of the template class (or if they are in different file, definitions should be prefixed with keyword export, but most of the compilers does not support this feature yet).
- All code you write should be located in the file SortedList.h. In this file, you can uncomment any method inherited from template class List if you need to override it and you should provide implementations (codes) for uncommented methods and new methods after the class definition.
- You can add private section at the end of definition of template class SortedList and add new data member there or any auxiliary methods you want to use.
- When calling a method which is inherited from template class List and is not overridden in template class SortedList, you **have to use** one of the two following ways (otherwise your code will not compile): either access this method via this, or explicitly specify that you are calling a method defined in List with scope resolution operator: List<T>::.

  For example, assume that you haven't overridden method get() in template class SortedList (you didn't uncommented its declaration in the definition of SortedList) and that in some other method of SortedList, if you need to call get() method. You cannot call get() directly:

  ```
  T item = get(index); // compiler error
  ```

  you have to use one of following two statements:

  ```
  T item = this->get(index);
  T item = List<T>::get(index);
  ```

- You cannot substitute int for template parameter T, otherwise the compiler will be not able to distinguish between items and indexes. Therefore, long is used as a template parameter in sample testing file Tester.cpp.
- You can only modify the file SortedList.h and Tester.cpp (if you want to try more complicated tests), but Tester.cpp will be replaced with another file when testing your submission.

Marking details:

2 parts:

- visual inspection of the source code (1 point):

  - (0.5 points) check that the previous code was not modified, mainly check that no new methods were added to the class List and that the data members of List are private (and not protected or public). If a student, didn't submit files with the previous code, automatically give them 0.5 point for this part.
  - (0.5 points) if the implementation of the student is keeping elements in a sorted manner, check that the add() method is using the binary search to find a position where to insert an element. If they use other implementation, automatically give them 0.5 point for this part.
    Remark: to check whether implementation is keeping elements sorted, look at the code of get() method. If it's not overloaded or just calls super. get(), then elements are sorted (i.e., check for binary search in add()). If get() is sorting and searching for k-th smallest element, then the elements are not sorted (i.e., 0.5 points automatically).

- testing (2 points): replace or add Tester.java/Tester.cpp to the collection of files submitted by student and compile. The program should output the points in the last line.
  Remark: some students submitted only SortedList.java/SortedList.h, the only file which needed to be changed for the implementation of the problem. This is complicating the evaluation process. Please, next time submit all files, even those included with the assignment which you are not supposed to modify.

---

2. (4 points) Consider the Quick Sort algorithm presented on the lecture/text book. Assume that pivot in each recursive call is chosen such that the size of the first partition is

a) n-1 if $n<1001$, otherwise 1000;

b) $n/1000$;

where n is the size of the subarray to be partitioned. The second partition has size n-1-(the size of the first partition). For example, consider the strategy **b)**. Assume we run Quick sort on an array of size 30 000. In the first partitioning, the whole array is partitioned to: the first partition of size 30, pivot (1 element) and the second partition of size 29 970. In the next recursive call, the subarray of size 30 is further partitioned to: the first partition of size 0, pivot (1 element) and the second partition of size 29, etc.

For each of the **two** cases,

- specify the recurrent formula expressing the time cost of the quick sort;
- provide an explicit non-recurrent formula (i.e., solve the recurrence) for the time cost; use the big-O notation!

You can assume that the partitioning algorithm takes linear ($O(n)$) time, where n is the size of subarray which is being partitioned. To solve the recurrence:
(1) use repeated substitution to guess the solution; and

(2) prove that your guess is correct using the mathematical induction.

Hint: in case **a)**, you can assume that *T(1000)* is of order *O(1)*.

Marking details:

There are two problems a) and b), each worth 2 points. Distribution of points for each of two problems:

- recurrent formula (0.4 points)
- substitution method (0.7 points)
- guess using O notation (0.2 points)
- proof of the guess using mathematical induction (0.7 points). Remark: it's ok if the student omitted the base case of the induction, as it's obvious that for some small value of n any formula will be true if the constant is set sufficiently high.

The correct solutions:

a) Recurrent formula: for *n<1001*, T(n) is O(1), otherwise T(n)=O(n)+T(1000)+T(n-1001)=O(1)+T(n-1001).
Substitution method: we have that T(n)<=cn+T(n-1001) for some value c>0.
Repeated substitution gives:
T(n)<=cn+c(n-1001)+c(n-2.100)+…+c(n-(k-1).1001)+T(n-k.1001)
= k/2.c.(2n-(k-1).1001)+T(n-k.1001).
If n-k.1001 is a constant, then T(n-k.1001) is O(1). This is the case, if for instance, n-k.1001=1. Then k=(n-1)/1001. Therefore,
T(n)<=(n-1)/2002.c.(2n-(n-1)+1001)+O(1)=O(n^2).
Guess: O(n^2)
Proof by MI: Show that there is m>0 and d>0 such that T(n)<=d.n^2, for all n>=m. The base is always true, just choose d (and/or m) sufficiently large. Inductive step: induction hypothesis is that T(l)<=d.l^2, for all l<n and we need to prove it for l=n.
T(n)<=cn+T(n-1001), by ind. hyp.,
<=cn+d$(n-1001)^2$ = dn^2 + cn + 1001^2 – 2.1001.d
= dn^2 + (c-1001d)n + 1001(1001-d).
Now, choose d such that c-1001d<=0 and 1001-d<=0, i.e., d>=c/1001 and d>=1001, then T(n) <= dn^2 and we are done.

b) Recurrent formula: T(n)=O(n)+T(n/1000)+T(n-1-n/1000).
Substitution method: we have that T(n)<=cn + T(1/1000.n) + T(999/1000.n).
Repeated substitution gives:
T(n) <= cn + T(1/1000.n) + T(999/1000.n)
<= cn + c.1/1000n + T(1/1000^2.n) + T(999/1000^2.n)
+ c.999/1000.n + T(999/1000^2.n) + T(999^2/1000^2.n)
= 2cn + T(1/1000^2.n) + T(999/1000^2.n) + T(999/1000^2.n) + T(999^2/1000^2.n)
etc. Important observation is that if we simultaneously expand all terms (substitute for them using recurrent formula), it will produce another c.n (sum of partitioning times). If we perform the substitution k times, we have k.cn + 2^k terms of type T(…). We need to pick k such that all of them are O(1). The rightmost term decreases slowest and is after k simultaneous substitutions T(999^k/1000^k.n).
Assume that 999^k/1000^k.n = 1, then k=log n / log (1000/999) ≈ 3319 log n.

Although the constant is big, it's still O(log n). Therefore, the time is k.cn=O(n.log n). [Also note that 2^k terms is only O(n) terms and each of them is O(1), so they all together are only O(n). This is expected, as the actual work during sorting is done during partitioning.]
Guess: T(n)=O(n.log n).
Proof by MI: Show that there is m>0 and d>0 such that T(n)<=d.n.log n, for all n>=m. The base is always true, just choose d (and/or m) sufficiently large.
Inductive step: induction hypothesis is that T(l)<=d.l.log l, for all l<n and we need to prove it for l=n.
T(n)<= cn + T(n/1000)+T(999/1000.n), by ind. hyp.,
    <= cn + d.n/1000.log(n/1000) + d.999/1000.n.log(999/1000.n)
     = dn.log n + cn – d.n.log(1000)/1000 – d.n.999/1000.log(1000/999)
Now, it's enough to set d such that, for instance c-d.log(1000)/1000 <=0, i.e., pick d>=1000c/log (1000) ≈ 100c.

---

3. (3 points) Provide **two codes** for iterative version of pre-order and post-order tree traversal algorithms for binary trees. The header of the first algorithm should look like this:

```
static public void preOrder(TreeNode<T> root);
```

The pre-order algorithm should be implemented using one stack (and TreeNode-s having only references to left and right child). The post-order algorithm should be implemented without using stack or any other complex data structure (no array, etc, you can use only variables which can store data of size *O(1)*, for instance constant number of references to TreeNode-s, boolean or int variables, etc), but you can assume the TreeNode-s have reference to the parent node.

Remember, the traversal algorithm should call method void visit(TreeNode<T>) on every node in the tree *exactly* once. In the pre-order traversal, it should call visit() on a node v before it calls visit() on any of its descendants. In the post-order traversal, it should call visit() on a node v after it calls visit() on every of its descendants.

Marking details:

Each of two codes is worth 1.5 points.

Pre-order algorithm idea: at each node: visit the node, push the right child on the stack (if it's not null) and move to the left child. If the current node becomes null, pop() a new node from stack.

Post-order algorithm: traverse tree as described on the lecture, traversing each edge twice (first time down, second time up). Visit the node when coming to it from the right child (or if it doesn't have right child, when coming to it from left child, or if it's a leaf, just visit it).