Data Structures (CS 102), Professor Yap – Fall 2016

HOMEWORK

November 21, 2016

Homework 6: Due on Wednesday Nov 30, 2016

- This homework is worth 100+5 Points. In each homework, you get the extra 5 points by following all our instructions.
- READ CAREFULLY the homework instructions and policy in `http://cs.nyu.edu/~yap/wiki/class/index.php/DataStruc/Hw-page`.

PART A: WRITTEN ASSIGNMENT (42 Points)

Question A.1 (3+3 Points)
(a) Ex.6.42 (p.467) on Binary Search. Fill in the table.

(b) Suppose I have a list of n elements. If $n = 100$, what is the worst case number of comparisons needed for each of the three algorithms? (Just extend the table in part(a) by a new row for $n = 100$). Do the same for $n = 1000$, $n = 10^6$ and $n = 10^9$.

Question A.2 (3 Points each part)
As usual, even if the answer is a number or a Yes/No, a brief justification is always needed to get full credit.

(a) What is a full binary tree?

(b) If a full binary tree has size 99, how many leaves does it have?
HINT: there is a simple relation between the number L of leaves and number N of internal nodes (i.e., non-leaves) in a full binary tree. First discover this relation.

(c) What is the value of $\sum_{i=0}^{29} 2^i$? ALSO, please give the informal estimate of this number (in terms of units like thousands, millions, etc).
HINT: recall the formula for geometric sums.

(d) Let T be complete binary tree (this is the shape from the heap data structure). If T has size 1000, how many nodes does it have in its last level? HINT: You might want to review Lecture 11.

Question A.3 (3 Points each part)
Consider the Collatz Sequence starting with 13.

(a) Starting from the empty heap, enqueue successive keys from $Collatz(17)$ into this heap. Show us the heap after each enqueue operation.
NOTE: show the heap as a complete binary tree, NOT as an array.

(b) For each heap you draw in part(a), write below it the number of swaps by the enqueue.

(c) Starting from the final heap in the last question, perform successive dequeues until the heap is empty. Draw the heap after each operation.

(d) Again, write below each heap in part(c) the number of swaps performed by the dequeue.

Question A.4 (12 Points)
We discussed heapsort in Lecture 11. Let `values` be an array containing the int's. We want to convert this array into a max-heap using the following method from page 708 of Text:

```
        for (index = SIZE/2 -1; index >=0; index--)
           reheapDown(values[index], index, SIZE -1);
```

code is only the first stage of the **Heapsort** algorithm (read p. 704 and following in Chapter 10). Read about `reheapDown` on p. 627, Chapter 9. This process is sometimes called **heapify** or **buildHeap**.

(a) Simulate the above code on the following initial array of values:

`int[] values = {1,2,3,4,5,6,7,8,9,10}`.

Here `SIZE=10`. Show the resulting complete binary tree after each iteration of the for-loop. For each iteration, also indicate the number of swaps that was done. NOTE: this tree is not necessarily a heap until the last iteration.

(b) Here is another way to heapify:

```
void Heapify(int[] array){
   for (int i=1; i<SIZE; i++)
         enqueue(values,i,values[i]);
}
```

where `enqueue(A,i,k)` means that we add the new key `k` to the heap of size `i` represented by the subarray `A[0..i-1]`.

Simulate this algorithm on the `values` array from part(a). Show the heap as a result of each enqueue operation. For each heap, indicate the number of swaps used by the enqueue operation.

(c) What is the total number of swaps in (a) and in (b)? Which algorithm is better?

PART B: PROGRAMMING ASSIGNMENT (58 Points)

All needed files are found in `hw6_Yap.zip` from Piazza/Resources. They must be included in your `src` folder and should be unmodified unless we allow otherwise. The Makefile has provided all the necessary targets, so there is NO REQUIRED CHANGES from you. To see what these targets are, please type `make help`. Of course, you may create your own targets for your testing, etc.

Question B.1 Timing of Reversal (20 Points)

First study the file `src/rev/Rev.java`. You can test it by calling `>> make run5` with various command line arguments. The class `Rev` has 3 methods for reversing a singly linked list:

```
Node reverse(Node u); // based on recursion
Node iReverse(Node u); // based on iteration
Node iReverse2(Node u); // based on iteration
```

`Rev` is an extension of the class in `src/util/MyList.java` where the method `reverse` has already been implemented. We want you to implement a new class called `MyRev` that extends the class `Rev`. But in `MyRev`, you only need to implement the above two iterative methods. Both iterative solutions use 3 Node pointers: the difference between them is that in `iReverse`, we need to update all 3 pointers in each iteration, while in `iReverse2`, we use a circular array of 3 Nodes, so you only need to update one of the 3 pointers:

```
Node[] pp = new Node[3]; int idx = 0;
```

and use `idx` as index to these pointers. E.g., `pointers[(idx+i)%3]` where `i=0,1,2`.

Besides the java file `MyRev.java`, you have to deliver a total of four text files:

- You must provide three output files called `OUTPUT-reverse-XXX` (where `XXX=100, 1000, 10000`). (Recall how you can use indirection at the terminal to send the output to such files.) We provide a sample file called `SAMPLE-reverse-10` (corresponding to `XXX=10`). Our sample file is produced by the terminal command:

  ```
  ≫ make run5 nn=10
  ```

  Each text file will contains a table of timing comparisons for the 3 reverse methods for various values of `nn`. The method for producing this table has already been written for you. There are 6 columns in a table: the first 3 columns are average times in milliseconds. The last 3 columns are ratios of the timings. All ratios are at least 1. E.g., a ratio of 2.3 means that the method is 2.3 times slower than the best.
- Provide a text file called `COMMENTS-reverse` in which you should write a paragraph to discuss your observations about the efficiency of the 3 methods for reversing a list, based on the data in files `OUTPUT-reverse-XXX`.
- In the file `COMMENTS-reverse` you must further write a paragraph about the following experiment: *Tell us the largest value of `nn` you can use to run the `reverse` method before your system crashes. Also give the largest value of `nn` for the `iReverse` method, but in this case, you are allowed to give up after about 15 minutes and give us a screen shot.* Explain what is going on.

Question B.2 FreeList (20 Points)

You might want to first review Chapter 7.4 in the Text about simulating linked list using arrays, and also Lecture 9. The class in `src/freelist/FreeListz.java` is to implement the generic interface called `FreeList<T>` found in `src/freelist/FreeListInterface.java`. The methods of `FreeListInterface` are explained in the file itself.

We use two arrays: `int[] intArray` and `T[] tArray` where `T` is a generic type. Both arrays have the same length of `MAX`. The pair `(tArray[i]), intArray[i]` (for index `i=0,1,...,MAX-1`) can be viewed as the node, but we call such a pair a **slot**. Like nodes, we can use slots can form linked list. This problem makes you wear two hats:
(a) Wearing the "operating system hat", your task is to maintain a pool of the unused slots, called **free** slots. The free slots are maintained in a linked list called the **free list**, and this name lends itself to our entire data structure.
(b) Wearing the "user hat", your task is to create two linked lists called `oddList` and `evenList` and to build them up by insertions as well as removals from these lists. There are 2 kinds of insertions: `add` and `app` (i.e., "append"). Intuitively, "add" puts the new slots at the head of the list, and "app" puts the new slots at the tail of the list. In order for append to work, we need to maintain an additional pointer to the end of the list.

We have written the main program and various methods already – leaving only the following methods for you to implement:

(a) newSlot, releaseSlot
(b) remove, add, app

In the main program, we test your generic data structure by choosing `T=String`, and these string values are populated with random Zoombini names.

To make sure that you can replicate our solution, we provide an output file called `SAMPLE-freelist-ss111` which you should be able to duplicate by calling

$\gg$ make run4 ss=111 nn=12

if you had correctly implemented the methods. Our main program, with command line arguments of `nn` and `ss`, performs these actions:

(1) First create an instance of `FreeList<String>` with `MAX` set to `nn`. Also, get an instance of `Zoombinis` in order to generate random Zoombini names for our lists.
(2) Initialize two empty lists called `oddList` and `evenList`.
(3) Perform a sequence of `nn` insertions ("add" or "app"): for each insertion, generate a random Zoombini name. If the length of the name is odd, we "add" the name to the head of `oddList`, else "app(end)" the name to the tail of `evenList`.
(4) Now use `showList` to print the size and slots in `oddList, evenList`; also print the number of slots in the free list.
(5) Next perform repeated removals (in any order you like) from the `oddList` until it is empty. Leave the `evenList` alone.
(6) Finally, do the operations of step (4) (showList, etc).

Question B.3 Display of Binary Trees (18 Points)
The class in `src/bst/DisplayBST.java` is to extend the class `src/bst/BSTz.java`

from a previous homework. `BSTz` is a generic class, but we instantiate its generic type using `T=String`. The key in each node is a string that is a Zoombini name. Our goal in this problem is to implement a method called `showBinaryTree()`. The nature of this display is described in the file `DisplayBST.java`, and illustrated here by example:

```
*     Here is a binary tree rooted at label CCC of size 8:
*
*                           CCC
*                        ____|_________________
*                      BBBB                   G
*                    ____|             _______|__
*                 AAAAA               EE       HHH
*                                 _____|____
*                              DDDDDDD   FFFF
*
* When projected to x-axis, we get the string:
*
*                 AAAAABBBB CCCDDDDDDDEE FFFF GHHH
*
* Note that if a label has even length, we append a blank
* space to the label.  For instance, the labels BBBB, EE and FFFF
*      each has a space appended.  After projecting the labels
* to the x-axis, we get the string AA...GHHH which has length 32.
* So we say the 'weight' of the root (CCC) of this tree is 32.
```

The idea of the method is to use two queues (q1 and q2). Suppose q1 stores an entire levels of the binary tree. We will process each node u in q1, by printing some characters (including white spaces) on the screen, and before we go to the next node in q1, we either

(Case 1) Enqueue node u into q2, or

(Case 2) Enqueue any children of u into q2.

When we are done, q1 would be empty, but q2 would contain either the same level as q1 earlier (Case 1), or contain the next level (Case 2). Note that we do not mix up the 2 cases while processing the queue. In the next stage, we can exchange the roles of q1 and q2. There are two methods which you have to implement, called `show1(q1,q2)` and `show2(q1,q2)` corresponding to these 2 cases.

We need to store additional statistics about a node to help us display the binary tree: here, we define the concept of the **weight** of a node u. Each node has **u.label** which is a string to be displayed at that node. The **halflength** of the u is defined to be u`.label.length()/2`. E.g., if the label is `CCC` then the half length is `1`. We allocate $1 + 2\cdot$`halflength` spaces to the label. Then u`.weight` is the sum of all spaces allocated to the nodes below, and including, node u. We also define the **position** of a node u to be the number of spaces that must be allocated to all the labels that must be drawn to the left of the subtree $T(u)$. For instance, if u is the node `G` in the above example. then u`.pos`=13 since `AAAAABBBB CCC` has length 13. To fill in these weight and position information, we define a method called `fillStats`.

WHAT TO IMPLEMENT: In the file `src/bst/DisplayBST.java`, we have left three methods for you to implement:

```
fillStats(...), show1(...), show2(...)
```

When implemented correctly, you can do this on the terminal

$$\gg \text{make run3y nn=13} > \text{OUTPUT-display-13}$$

to produce an output that should be similar to our file `SAMPLE-display-13`.