CSE 2320 Lab Assignment 1

Due February 16, 2010

Goal:

Implementation of bottom-up mergesort using linked list ideas.

Requirements:

1. Write a Java program that will read an input sequence into an array and then produce an array of links that gives the values in ascending order. The first line of the input will be the length of the sequence (`n`) and each of the remaining `n` lines will be a non-negative integer. The following outputs should be produced:

   a. If `n` ≤ 50, then the original input (and initial links) should be printed.
   b. For each of the bottom-up rounds, the number of sublists that are input to the round should be printed.
   c. If `n` ≤ 50, then the original input (and modified links) should be printed after each round. The subscript of the first list element should also be printed
   d. After the entire bottom-up mergesort finishes, the result should be verified:

   ```
   // Verify the output
   int count=0;
   for (i=oldStart;
        link[i]!=LISTNULL && (key[i]<key[link[i]]
                          || i<link[i] && key[i]==key[link[i]]);
        i=link[i])
     count++;
   if (link[i]==LISTNULL && count==n-1)
     System.out.print("Verified\n");
   else
     System.out.format("Error at %d to %d with count %d\n",i,link[i],
       count);
   ```

2. Email your program (as an attachment) to `fawaz.bokhari@mavs.uta.edu` by 10:45 a.m. on February 16, 2010.

Getting Started:

1. The input is easily read by importing `java.util.Scanner` and using code like:

   ```
   Scanner sc=new Scanner(System.in);
   int n=sc.nextInt();
   ```

   Do not prompt for a file name!

2. The `link` values are initialized according to the original input order. Use `-1` to indicate a grounded pointer (`LISTNULL`).

3. Instead of viewing the initial list as `n` sublists, each with one element, we will take advantage of any ordering that may be present in the input. Before processing any merges, a simple scan of the input will allow determining the initial number of sublists. This scanning should not be performed again for later merge rounds.

4. It will be useful to have a function to find the end of a sublist when given the start of a sublist.

5. Your merge routine should be written according to the general pattern discussed in class:

   a. Initialize for the two input sublists and the output sublist (e.g. by modifying links).
   b. A processing loop that continues until one of the two input sublists is exhausted.
   c. Clean-up code for each of the two cases of the input sublists being exhausted.

   Unlike most merge situations, you may take advantage of the fact that both input sublists have at least one element.

6. A round of merging consists of merging each consecutive pair of sublists. If the number of sublists is odd, then the last sublist will be carried forward to the next round. It is unacceptably costly to code this program to iteratively merge just the first two sublists.

7. If an input value appears more than once, their elements should be ordered by subscripts in the final list, i.e. your sort code will be stable.

8. Consider the following input:

```
10
1
0
2
1
1
1
1
0
0
1
```

The output will be:

```
 i key link
 0   1    1
 1   0    2
 2   2    3
 3   1    4
 4   1    5
 5   1    6
 6   1    7
 7   0    8
 8   0    9
 9   1   -1
Round starting with 4 lists
First element at subscript 1
 i key link
 0   1    2
 1   0    0
 2   2    7
 3   1    4
 4   1    5
 5   1    6
 6   1    9
 7   0    8
 8   0    3
 9   1   -1
Round starting with 2 lists
First element at subscript 1
 i key link
 0   1    3
 1   0    7
 2   2   -1
 3   1    4
 4   1    5
 5   1    6
 6   1    9
 7   0    8
 8   0    0
 9   1    2
Verified
```

Notice that the input sequence ordering has not changed.