Conrad Horton
CSC382 - September 2016
10-2-2016
Sorting

INTRO: I chose to design and implement versions of the Merge sort, and Bubble sort algorithms to order a generic Doubley Linked List. Merge sort works by splitting the entire list into one or more sublists, ordering them, and joining all sorted lists together. Bubble sorts work by floating the smaller value Nodes to the top of the Linked List.

PUBLIC METHODS:
void doMergeSort() - This works by starting with the and performing a Merge sort on the list. Calls mergeSort(Node<T>*& headRef) passing in the Root Node..

void doBubbleSort() - This works by starting with the rootNode and performing a Bubble sort on the list. Calls bubbleSort(Node<T>** node) passing in the Root Node.

PROTECTED METHODS:
void mergeSort(Node<T>& headRef)* - Recursively goes through the passed in list starting from the passed in Node, and makes sublists. Each sublist is then ordered, and linked together.

Node<T> sortedMerge(Node<T>* listA, Node<T>* listB)* - Recursively attaches the passed in Linked List Nodes and then reattaches them based on sorting order.

void split(Node<T> source, Node<T>*& front, Node<T>*& back)* - Splits the list and sets a reference to the back Node and front Node of the lists.

*void bubbleSort(Node<T>** node)* - Compares each node in the list and swaps their data until the data of least value has bubbled to the top.

Sample use:

```
#include "DoubleLinkList.h"
#include <exception>
#include <ctime>
#include <iostream>

#define QTY 1000

using namespace std;

int main()
  {
  srand (time(NULL));
  long keyArray[QTY];

  /** Different sized lists. */
  DoubleLinkList<int>* dbllBubble1000 = new DoubleLinkList<int>;
  DoubleLinkList<int>* dbllMerge1000  = new DoubleLinkList<int>;

  /** Load the key array with random unordered keys. */
  int k = 0;
  while(k < QTY)
    {
    /** Check to see if the key exists. This is slow since it is iterating over
N elements. */
    bool keyExists = false;
    long key = rand() % 10000 + 1;
    for (int i = 0; i < QTY; i++)
      {
      if(keyArray[i] == key)
        keyExists = true;
      }

    /** Key does not exist. Add it and go to the next element. */
    if(!keyExists)
      {
      keyArray[k] = key;
      k++;
      }
    }

  /** Fill all lists with the same data. */
  for (int j = 0; j < QTY; j++)
    {
    /** Load data into the linked 1000 Node lists. */
    dbllBubble1000->insert(keyArray[j], new int(j));
    dbllMerge1000 ->insert(keyArray[j], new int(j));
    }

  /** Perform sorting algorithms, and calculate execution times. */
  try
    {
    /** Do Bubble sorts. */
```

```
    dbllBubble1000->doBubbleSort();
    dbllBubble1000->printList();

    /** Do Merge sorts. */
    dbllMerge1000->doMergeSort();
    dbllBubble1000->printList();
    }
  catch(exception e)
    {
    cout << e.what() << endl;
    }

  system("pause");
  return 1;
  }
```

Analysis:

Statistics (32-bit processor build)
Merge Sort est. BIG-O Θ(n log(n))
Bubble Sort est. BIG-O Θ(n^2)
Merge time: @10 elements = 0 | @100 elements = 0 | @1000 elements = 1
Bubble time: @10 elements = 0 | @100 elements = 0 | @1000 elements = 5

Comparing the BIG-O notation we can see that the Merge sort is a much more efficient sorting algorithm than the Bubble sort with increasingly larger data sets. While it is more difficult to code up and debug if you are dealing with large datasets, the merge would be a better choice.