PROJECT 2

SORTING

CSCI 230
DATA STRUCTURE II

MAI PHAM

DEVELOPMENT ENVIRONMENT
MAC OS (xCode)

TABLE OF CONTENTS

Mai Pham
CSCI 230 - Project 2
Prof. T. Vo

PROJECT STATUS

OBJECTIVE:

- Define and implement four sorting algorithms (selection, insertion, merge, and quick sort) on a small and large text file with different data types. Output and compare the data moves, key compares, and time in milliseconds for each sorting algorithm.

STATUS:

- The main project is completed; including all the code and the relevant output base on the requirement.

EXTRA CREDIT:

- I tried to implement the hybrid, shell and radix sort for my extra credit. However, only my shell sort is working completely. For the radix sort, the template function could only work with integer but not string. This is because string can't be divide by the integer position in the for loop. I try to convert my string to integer (atoi) but get error message saying my string is an integer. Therefore, I cannot implement the radix sort for string list. However, it works perfectly for the integer list.

RESULTS & DISCOUSSIONS

SMALL 1K INTEGER			
Sorting	**Key Compares**	**Data Moves**	**Time (ms)**
Selection	499500	3000	4.044
Insertion	247371	739116	22.241
Merge	8711	10030	0.845
Quick	18300	7173	1.374
Shell	798131	22242	5.508
Radix	N/A	N/A	0.544
SMALL 1K STRING			
Sorting	**Key Compares**	**Data Moves**	**Time (ms)**
Selection	499500	3000	24.223
Insertion	244356	730071	32.056
Merge	8733	10030	1.009
Quick	18270	7137	1.846
Shell	798131	23139	44.139
LARGE 100K INTEGER			
Sorting	**Key Compares**	**Data Moves**	**Time (ms)**
Merge	1566353	1700051	180.776
Quick	3091040	1178331	275.239
Shell	258856816	8538312	58424.3
Radix	N/A	N/A	93.517
LARGE 100K STRING			
Sorting	**Key Compares**	**Data Moves**	**Time (ms)**
Merge	1566415	1700051	177.462
Quick	3013837	1174647	297.66
Shell	258856816	8538312	440652

I know for certain that my coding for the sorting algorithms is correct. Therefore, the time in milliseconds should be correct. However, I'm not too sure if I'm computing the data moves and the key compare are correctly. However, based on the above information, it has shown that merge sort and quick sort run much faster compare to selection sort and insertion sort. Shell sort is around the same range as selection and insertion sort. I also notice when sorting in string, the running time is a little bit slower even though the data moves and key compares are the same. Also, radix sort running time is really fast compare to all the other sorting algorithms.

OUTPUT FILE

```
Sorting Method = Selection Sort
Input File Name = small1k.txt
Number of Values Sorted = 1000
Sorted Key Data Type = Integer
Number of Key Compares = 499500
Number of Data Moves = 3000
Times in Milliseconds = 4.044
First 5 pairs:
     (7, 7)         (11, 11)     (15, 15)     (39, 39)     (59, 59)
Last 5 pairs:
     (8163, 8163)         (8167, 8167)        (8175, 8175)        (8183, 8183)
     (8191, 8191)

Sorting Method = Insertion Sort
Input File Name = small1k.txt
Number of Values Sorted = 1000
Sorted Key Data Type = Integer
Number of Key Compares = 247371
Number of Data Moves = 739116
Times in Milliseconds = 22.241
First 5 pairs:
     (7, 7)         (11, 11)     (15, 15)     (39, 39)     (59, 59)
Last 5 pairs:
     (8163, 8163)         (8167, 8167)        (8175, 8175)        (8183, 8183)
     (8191, 8191)

Sorting Method = Merge Sort
Input File Name = small1k.txt
Number of Values Sorted = 1000
Sorted Key Data Type = Integer
Number of Key Compares = 8711
Number of Data Moves = 10030
Times in Milliseconds = 0.845
First 5 pairs:
     (7, 7)         (11, 11)     (15, 15)     (39, 39)     (59, 59)
Last 5 pairs:
     (8163, 8163)         (8167, 8167)        (8175, 8175)        (8183, 8183)
     (8191, 8191)

Sorting Method = Quick Sort
Input File Name = small1k.txt
Number of Values Sorted = 1000
Sorted Key Data Type = Integer
Number of Key Compares = 18300
Number of Data Moves = 7173
Times in Milliseconds = 1.374
First 5 pairs:
```

```
     (7, 7)         (11, 11)     (15, 15)     (39, 39)     (59, 59)
Last 5 pairs:
     (8155, 8155)         (8163, 8163)        (8167, 8167)        (8175, 8175)
     (8191, 8191)

Sorting Method = Shell Sort
Input File Name = small1k.txt
Number of Values Sorted = 1000
Sorted Key Data Type = Integer
Number of Key Compares = 798131
Number of Data Moves = 22242
Times in Milliseconds = 5.508
First 5 pairs:
     (7, 7)         (11, 11)     (15, 15)     (39, 39)     (59, 59)
Last 5 pairs:
     (8163, 8163)         (8167, 8167)        (8175, 8175)        (8183, 8183)
     (8191, 8191)

Sorting Method = Radix Sort
Input File Name = small1k.txt
Number of Values Sorted = 1000
Sorted Key Data Type = Integer
Number of Key Compares = 0
Number of Data Moves = 0
Times in Milliseconds = 0.544
First 5 pairs:
     (7, 7)         (11, 11)     (15, 15)     (39, 39)     (59, 59)
Last 5 pairs:
     (8163, 8163)         (8167, 8167)        (8175, 8175)        (8183, 8183)
     (8191, 8191)

Sorting Method = Selection Sort
Input File Name = small1k.txt
Number of Values Sorted = 1000
Sorted Key Data Type = String
Number of Key Compares = 499500
Number of Data Moves = 3000
Times in Milliseconds = 24.223
First 5 pairs:
     (103, 103)     (1035, 1035)        (1047, 1047)        (1055, 1055)        (1063,
1063)
Last 5 pairs:
     (955, 955)     (959, 959)   (987, 987)   (99, 99)     (995, 995)

Sorting Method = Insertion Sort
Input File Name = small1k.txt
Number of Values Sorted = 1000
Sorted Key Data Type = String
Number of Key Compares = 244356
Number of Data Moves = 730071
Times in Milliseconds = 32.056
First 5 pairs:
     (103, 103)     (1035, 1035)        (1047, 1047)        (1055, 1055)        (1063,
1063)
Last 5 pairs:
     (955, 955)     (959, 959)   (987, 987)   (99, 99)     (995, 995)

Sorting Method = Merge Sort
Input File Name = small1k.txt
Number of Values Sorted = 1000
Sorted Key Data Type = String
```

```
Number of Key Compares = 8733
Number of Data Moves = 10030
Times in Milliseconds = 1.009
First 5 pairs:
     (103, 103)     (1035, 1035)        (1047, 1047)        (1055, 1055)        (1063,
1063)
Last 5 pairs:
     (955, 955)     (959, 959)    (987, 987)    (99, 99)     (995, 995)

Sorting Method = Quick Sort
Input File Name = small1k.txt
Number of Values Sorted = 1000
Sorted Key Data Type = String
Number of Key Compares = 18270
Number of Data Moves = 7137
Times in Milliseconds = 1.846
First 5 pairs:
     (103, 103)     (1035, 1035)        (1047, 1047)        (1055, 1055)        (1063,
1063)
Last 5 pairs:
     (95, 95)       (955, 955)    (959, 959)    (987, 987)    (99, 99)

Sorting Method = Shell Sort
Input File Name = small1k.txt
Number of Values Sorted = 1000
Sorted Key Data Type = String
Number of Key Compares = 798131
Number of Data Moves = 23139
Times in Milliseconds = 44.139
First 5 pairs:
     (103, 103)     (1035, 1035)        (1047, 1047)        (1055, 1055)        (1063,
1063)
Last 5 pairs:
     (955, 955)     (959, 959)    (987, 987)    (99, 99)     (995, 995)

Sorting Method = Merge Sort
Input File Name = large100k.txt
Number of Values Sorted = 1000000
Sorted Key Data Type = Integer
Number of Key Compares = 1566353
Number of Data Moves = 1700051
Times in Milliseconds = 180.776
First 5 pairs:
     (1, 1)         (2, 2)       (3, 3)       (4, 4)       (5, 5)
Last 5 pairs:
     (99996, 99996)       (99997, 99997)      (99998, 99998)      (99999, 99999)
     (100000, 100000)

Sorting Method = Quick Sort
Input File Name = large100k.txt
Number of Values Sorted = 1000000
Sorted Key Data Type = Integer
Number of Key Compares = 3091040
Number of Data Moves = 1178331
Times in Milliseconds = 275.239
First 5 pairs:
     (1, 1)         (2, 2)       (3, 3)       (4, 4)       (5, 5)
Last 5 pairs:
     (99995, 99995)       (99996, 99996)      (99997, 99997)      (99998, 99998)
     (100000, 100000)
```

```
Sorting Method = Shell Sort
Input File Name = large100k.txt
Number of Values Sorted = 1000000
Sorted Key Data Type = Integer
Number of Key Compares = -258856816
Number of Data Moves = 8538312
Times in Milliseconds = 58424.3
First 5 pairs:
     (1, 1)         (2, 2)       (3, 3)       (4, 4)       (5, 5)
Last 5 pairs:
     (99996, 99996)       (99997, 99997)      (99998, 99998)      (99999, 99999)
     (100000, 100000)

Sorting Method = Radix Sort
Input File Name = large100k.txt
Number of Values Sorted = 1000000
Sorted Key Data Type = Integer
Number of Key Compares = 0
Number of Data Moves = 0
Times in Milliseconds = 93.517
First 5 pairs:
     (1, 1)         (2, 2)       (3, 3)       (4, 4)       (5, 5)
Last 5 pairs:
     (99996, 99996)       (99997, 99997)      (99998, 99998)      (99999, 99999)
     (100000, 100000)

Sorting Method = Merge Sort
Input File Name = large100k.txt
Number of Values Sorted = 1000000
Sorted Key Data Type = String
Number of Key Compares = 1566415
Number of Data Moves = 1700051
Times in Milliseconds = 177.462
First 5 pairs:
     (1, 1)         (10, 10)     (100, 100)   (1000, 1000)        (10000, 10000)
Last 5 pairs:
     (99995, 99995)       (99996, 99996)      (99997, 99997)      (99998, 99998)
     (99999, 99999)

Sorting Method = Quick Sort
Input File Name = small1k.txt
Number of Values Sorted = 1000000
Sorted Key Data Type = String
Number of Key Compares = 3013837
Number of Data Moves = 1174647
Times in Milliseconds = 297.66
First 5 pairs:
     (1, 1)         (10, 10)     (100, 100)   (1000, 1000)        (10000, 10000)
Last 5 pairs:
     (99994, 99994)       (99995, 99995)      (99996, 99996)      (99997, 99997)
     (99998, 99998)

Sorting Method = Shell Sort
Input File Name = small1k.txt
Number of Values Sorted = 1000000
Sorted Key Data Type = String
Number of Key Compares = -258856816
Number of Data Moves = 8735352
Times in Milliseconds = 440652
First 5 pairs:
     (1, 1)         (10, 10)     (100, 100)   (1000, 1000)        (10000, 10000)
```

```
Last 5 pairs:
     (99995, 99995)       (99996, 99996)      (99997, 99997)      (99998, 99998)
     (99999, 99999)
```

SOURCE CODE

Main.cpp
Note: Does not include Entry.h and Comparator.h

```
//
//  main.cpp
//  Project_2
//
//  Created by Mai Pham on 4/14/18.
//  Copyright © 2018 Mai Pham. All rights reserved.
//

#include "Entry.h"
#include "Sorting.h"
#include <iostream>
#include <vector>
#include <fstream>
#include <algorithm>
#include <string>
using namespace std;

template <typename E>
void print1st20(vector<E> S) {
    for (int i = 0; i < 20; i++)
        cout << S[i].key() << "  " ;
}

template <typename E>
void printLast20(vector<E> S) {
    int b = S.size();
    int a = b - 21;
    for (int i = a; i < b; i++)
        cout << S[i].key() << "  " ;
}
template <typename E>
void printInfor(ofstream &outputFile, string methor, string fileName, int num, string
data, vector <E>S, Sorting <E>sort)   {
    outputFile << "Sorting Method = " << methor << endl;
    outputFile << "Input File Name = " << fileName << endl;
    outputFile << "Number of Values Sorted = " << num << endl;
    outputFile << "Sorted Key Data Type = " << data << endl;
    outputFile << "Number of Key Compares = " << sort.getCompares()<< endl;
    outputFile << "Number of Data Moves = " << sort.getDataMoves() << endl;
    outputFile << "Times in Milliseconds = " << sort.getMilliSeconds() << endl;
    outputFile << "First 5 pairs:" << endl;
    for (int i = 0; i < 5; i++)
        outputFile << "\t(" << S[i].key() << ", " << S[i].value() << ")   ";

    outputFile << "\nLast 5 pairs:" << endl;
    for (int i = S.size()-5; i < S.size(); i++)
        outputFile << "\t(" << S[i].key() << ", " << S[i].value() << ")   ";
    outputFile << endl << endl;
}
```

```
int main() {
    vector<Entry<int, string>> vect1kInt;
    vector<Entry<string, int>> vect1kString;
    vector<Entry<int, string>> vect100kInt;
    vector<Entry<string, int>> vect100kString;
    Sorting <Entry<int, string>> sortInt;
    Sorting <Entry<string, int>> sortString;

    int key;
    string value;

    ifstream mySmallList;
    mySmallList.open("small1k.txt");
    if(!mySmallList.is_open())
        cout << "No text file found. " << endl;

    while (mySmallList >> key)   {
        value = to_string(key);

        Entry<int, string> entryNum;
        entryNum.setKey(key);
        entryNum.setValue(value);
        vect1kInt.push_back(entryNum);

        Entry<string, int> entryWord;
        entryWord.setKey(value);
        entryWord.setValue(key);
        vect1kString.push_back(entryWord);
    }

    ifstream myLargeList;
    myLargeList.open("large100k.txt");
    if(!myLargeList.is_open())
        cout << "No text file found. " << endl;

    while (myLargeList >> key)   {
        value = to_string(key);

        Entry<int, string> entryNum;
        entryNum.setKey(key);
        entryNum.setValue(value);
        vect100kInt.push_back(entryNum);

        Entry<string, int> entryWord;
        entryWord.setKey(value);
        entryWord.setValue(key);
        vect100kString.push_back(entryWord);
    }

    ofstream outputFile;
    outputFile.open("p2Results.txt");


    // Small 1k Integer Sort
    vector<Entry<int, string>> selectionSort1kInt;
    selectionSort1kInt = vect1kInt;
    sortInt.selectionSort(selectionSort1kInt);
    printInfor(outputFile, "Selection Sort", "small1k.txt", 1000, "Integer",
selectionSort1kInt, sortInt);

    vector<Entry<int, string>> insertionSort1kInt;
```

```
    insertionSort1kInt = vect1kInt;
    sortInt.insertionSort(insertionSort1kInt);
    printInfor(outputFile, "Insertion Sort", "small1k.txt", 1000, "Integer",
insertionSort1kInt, sortInt);

    vector<Entry<int, string>> mergeSort1kInt;
    mergeSort1kInt = vect1kInt;
    sortInt.mergeSort(mergeSort1kInt);
    printInfor(outputFile, "Merge Sort", "small1k.txt", 1000, "Integer",
mergeSort1kInt, sortInt);

    vector<Entry<int, string>> quickSort1kInt;
    quickSort1kInt = vect1kInt;
    sortInt.quickSort(quickSort1kInt);
    printInfor(outputFile, "Quick Sort", "small1k.txt", 1000, "Integer",
quickSort1kInt, sortInt);

    vector<Entry<int, string>> shellSort1kInt;
    shellSort1kInt = vect1kInt;
    sortInt.shellSort(shellSort1kInt);
    printInfor(outputFile, "Shell Sort", "small1k.txt", 1000, "Integer",
shellSort1kInt, sortInt);

    vector<Entry<int, string>> radixSort1kInt;
    radixSort1kInt = vect1kInt;
    sortInt.radixSort(radixSort1kInt);
    printInfor(outputFile, "Radix Sort", "small1k.txt", 1000, "Integer",
radixSort1kInt, sortInt);


    // Small 1k String Sort
    vector<Entry<string, int>> selectionSort1kString;
    selectionSort1kString = vect1kString;
    sortString.selectionSort(selectionSort1kString);
    printInfor(outputFile, "Selection Sort", "small1k.txt", 1000, "String",
selectionSort1kString, sortString);

    vector<Entry<string, int>> insertionSort1kString;
    insertionSort1kString = vect1kString;
    sortString.insertionSort(insertionSort1kString);
    printInfor(outputFile, "Insertion Sort", "small1k.txt", 1000, "String",
insertionSort1kString, sortString);

    vector<Entry<string, int>> mergeSort1kString;
    mergeSort1kString = vect1kString;
    sortString.mergeSort(mergeSort1kString);
    printInfor(outputFile, "Merge Sort", "small1k.txt", 1000, "String",
mergeSort1kString, sortString);

    vector<Entry<string, int>> quickSort1kString;
    quickSort1kString = vect1kString;
    sortString.quickSort(quickSort1kString);
    printInfor(outputFile, "Quick Sort", "small1k.txt", 1000, "String",
quickSort1kString, sortString);

    vector<Entry<string, int>> shellSort1kString;
    shellSort1kString = vect1kString;
    sortString.shellSort(shellSort1kString);
    printInfor(outputFile, "Shell Sort", "small1k.txt", 1000, "String",
shellSort1kString, sortString);
    /*
```

```
    vector<Entry<string, int>> radixSort1kString;
    radixSort1kString = vect1kString;
    sortString.radixSort(radixSort1kString);
    printInfor(outputFile, "Radix Sort", "small1k.txt", 1000, "String",
radixSort1kString, sortString);
    */


    // Large 1k Integer Sort
    vector<Entry<int, string>> mergeSort100kInt;
    mergeSort100kInt = vect100kInt;
    sortInt.mergeSort(mergeSort100kInt);
    printInfor(outputFile, "Merge Sort", "large100k.txt", 1000000, "Integer",
mergeSort100kInt, sortInt);

    vector<Entry<int, string>> quickSort100kInt;
    quickSort100kInt = vect100kInt;
    sortInt.quickSort(quickSort100kInt);
    printInfor(outputFile, "Quick Sort", "large100k.txt", 1000000, "Integer",
quickSort100kInt, sortInt);

    vector<Entry<int, string>> sellSort100kInt;
    sellSort100kInt = vect100kInt;
    sortInt.shellSort(sellSort100kInt);
    printInfor(outputFile, "Shell Sort", "large100k.txt", 1000000, "Integer",
sellSort100kInt, sortInt);

    vector<Entry<int, string>> radixSort100kInt;
    radixSort100kInt = vect100kInt;
    sortInt.radixSort(radixSort100kInt);
    printInfor(outputFile, "Radix Sort", "large100k.txt", 1000000, "Integer",
radixSort100kInt, sortInt);


    // Large 1k String Sort
    vector<Entry<string, int>> mergeSort100kString;
    mergeSort100kString = vect100kString;
    sortString.mergeSort(mergeSort100kString);
    printInfor(outputFile, "Merge Sort", "large100k.txt", 1000000, "String",
mergeSort100kString, sortString);

    vector<Entry<string, int>> quickSort100kString;
    quickSort100kString = vect100kString;
    sortString.quickSort(quickSort100kString);
    printInfor(outputFile, "Quick Sort", "small1k.txt", 1000000, "String",
quickSort100kString, sortString);

    vector<Entry<string, int>> shellSort100kString;
    shellSort100kString = vect100kString;
    sortString.shellSort(shellSort100kString);
    printInfor(outputFile, "Shell Sort", "small1k.txt", 1000000, "String",
shellSort100kString, sortString);
    /*
    vector<Entry<string, int>> raidxSort100kString;
    raidxSort100kString = vect100kString;
    sortString.radixSort(raidxSort100kString);
    printInfor(outputFile, "Radix Sort", "small1k.txt", 1000000, "String",
raidxSort100kString, sortString);
    */

    outputFile.close();
```

```
    return 0;
}
```

Sorting.h

```
//
//  Sorting.h
//  Project_2
//
//  Created by Mai Pham on 4/14/18.
//  Copyright © 2018 Mai Pham. All rights reserved.
//

#ifndef Sorting_h
#define Sorting_h

#include "Comparator.h"
#include "Entry.h"
#include <iostream>
#include <vector>
#include <algorithm>
#include <cstdlib>
#include <ctime>
#include <string>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
using namespace std;

template <typename E>
class Sorting   {
private:
    unsigned int compares;
    unsigned int dataMoves;
    double milliSeconds;
    LeftRight <E>less;
public:
    Sorting()   {
        compares = 0;
        dataMoves = 0;
        milliSeconds = 0;
    }
    int getCompares()
    {   return compares;    }
    int getDataMoves ()
    {   return dataMoves;    }
    double getMilliSeconds ()
    {   return milliSeconds;    }

    void selectionSort(vector<E> &S);
    void insertionSort(vector<E> &S);
    void mergeSort(vector<E> &S);
    void merge(vector<E> &in, vector<E> &out, int b, int m);
    void quickSort(vector<E> &S);
    void quickSortStep(vector<E> &S, int a, int b);
    void findMedian(vector<E> &S);
    void shellSort(vector<E> &S);
    void radixSort(vector<E> &S);
};

template <typename E>
void Sorting<E>::selectionSort(vector<E> &S)   {
```

```
    int t1 = clock();
    compares = dataMoves = 0;
    int min;
    for(int i = 0; i < S.size(); i++)   {
        min = i;
        for(int j = i+1; j < S.size(); j++) {
            if( S[j].key() < S[min].key() )
                min = j;
            compares++;
        }
        swap(S[i], S[min]);
        dataMoves += 3;
    }
    int t2 = clock();
    milliSeconds = (t2-t1)/ (double)CLOCKS_PER_SEC*1000;
}

template <typename E>
void Sorting<E>::insertionSort(vector<E> &S) {
    int t1 = clock();
    compares = dataMoves = 0;
    int i,j;
    for (i = 1; i < S.size(); i++) {
        j = i;
        while (j > 0 && S[j - 1].key() > S[j].key()) {
            compares++;
            swap(S[j-1], S[j]);
            dataMoves += 3;
            j--;
        }
        compares++;
    }
    int t2 = clock();
    milliSeconds = (t2-t1)/ (double)CLOCKS_PER_SEC*1000;
}

template <typename E>                                       // merge-sort S
void Sorting<E>::mergeSort(vector<E> &S) {
    int t1 = clock();
    compares = dataMoves = 0;
    typedef vector<E> vect;
    int n = S.size();
    vect v1(S);  vect* in = &v1;                            // initial input vector
    vect v2(n);  vect* out = &v2;                           // initial output vector
    for (int m = 1; m < n; m *= 2) {                        // list sizes doubling
        for (int b = 0; b < n; b += 2*m) {                  // beginning of list
            merge(*in, *out, b, m);                         // merge sublists
        }
        swap(in, out);                                      // swap input with output
        dataMoves+=3;
    }
    S = *in;                                                // copy sorted array to S
    int t2 = clock();
    milliSeconds = (t2-t1)/ (double)CLOCKS_PER_SEC*1000;
}

template <typename E>
void Sorting<E>::merge(vector<E> &in, vector<E> &out, int b, int m) {
    int i = b;                                              // index into run #1
    int j = b + m;                                          // index into run #2
    int n = in.size();
```

```
    int e1 = min(b + m, n);                                 // end of run #1
    int e2 = min(b + 2*m, n);                               // end of run #2
    int k = b;
    //compares++;
    while ((i < e1) && (j < e2)) {
        //compares++;
        if(!less(in[j].key(), in[i].key()))                 // append smaller to S
            out[k++] = in[i++];
        else
            out[k++] = in[j++];

        dataMoves++;
        compares++;
    }
    //compares++;
    while (i < e1)  {                                       // copy rest of run 1 to S
        out[k++] = in[i++];
        dataMoves++;
        //compares++;
    }
    //compares++;
    while (j < e2){                                         // copy rest of run 2 to S
        out[k++] = in[j++];
        dataMoves++;
        //compares++;
    }
    //compares++;
}

template <typename E>                                       // quick-sort S
void Sorting<E>::quickSort(vector<E> &S) {
    int t1 = clock();
    compares = dataMoves = 0;
    if (S.size() <= 1) return;                              // already sorted
    quickSortStep(S, 0, S.size()-1);                        // call sort utility
    int t2 = clock();
    milliSeconds = (t2-t1)/ (double)CLOCKS_PER_SEC*1000;
}

template <typename E>
void Sorting<E>::quickSortStep(vector<E> &S, int a, int b) {
    if (a >= b) return;                                     // 0 or 1 left? done
    findMedian(S);
    E pivot = S[b];                                         // select last as pivot
    int l = a;                                              // left edge
    int r = b - 1;                                          // right edge
    //compares++;
    while (l <= r) {
        while (l <= r && !less(pivot.key(), S[l].key()))    {
            l++;                // scan right till larger
            compares++;
        }
        compares++;
        while (r >= l && !less(S[r].key(), pivot.key()))    {
            r--;               // scan left till smaller
            compares++;
        }
        compares++;
        if (l < r)  {                                       // both elements found
            swap(S[l], S[r]);
            dataMoves += 3;
```

```
        }
        //compares++;
    }                                                   // until indices cross
    swap(S[l], S[b]);                                   // store pivot at l
    dataMoves+=3;
    quickSortStep(S, a, l-1);                           // recur on both sides
    quickSortStep(S, l+1, b);
}

template <typename E>
void Sorting<E>::findMedian(vector<E> &S) {
    int a = 0;
    int c = S.size()-1;
    int b = S.size()/2;
    compares++;
    if (S[a].key() > S[b].key())    {
        if (S[b].key() > S[c].key())    {
            swap(S[b], S[c]);
            dataMoves += 3;
        }
        else if (S[c].key() > S[a].key())   {
            swap(S[a], S[c]);
            dataMoves += 3;
        }
        compares+=2;
    }
    else    {
        if (S[b].key() < S[c].key())    {
            swap(S[b], S[c]);
            dataMoves += 3;
        }
        else if (S[c].key() < S[a].key())   {
            swap(S[a], S[c]);
            dataMoves += 3;
        }
        compares+=2;
    }
}

template <typename E>
void Sorting<E>::shellSort(vector<E> &S)    {
    int t1 = clock();
    compares = dataMoves = 0;
    for (int i = S.size()/2; i > 0; i /=2)  {
        for (int j = i; j < S.size(); j++)  {
            for (int k = j - i; k >= 0; k -= i)  {
                if (S[k].key() > S[k+i].key())    {
                    swap(S[k+i],S[k]);
                    dataMoves+=3;
                }
                compares++;
            }
        }
    }
    int t2 = clock();
    milliSeconds = (t2-t1)/ (double)CLOCKS_PER_SEC*1000;
}

template <typename E>
void Sorting<E>::radixSort(vector<E> &S)    {
    int t1 = clock();
```

```
    compares = dataMoves = 0;
    E arr[S.size()];

    // get max to compute amount of loop
    E max = S[0];
    for (int i = 1; i < S.size(); i++)
        if (S[i].key() > max.key())
            max = S[i];
    /*
    int num;
    if (isdigit(max.key()))
        num = max.key();
    else
        num = std::atoi(max.key());
    //int num = max.key().length();
     */

    for (int position = 1; max.key()/position > 0; position *= 10)  {
        int count[10] = {0};
        for (int i = 0; i < S.size(); i++)
            count[(S[i].key()/position)%10]++;
        for (int i = 1; i < 10; i++)
            count[i] += count[i - 1];
        for (int i = S.size() - 1; i >= 0; i--) {
            arr [count[(S[i].key()/position) % 10]-1] = S[i];
            count[(S[i].key()/position) % 10]--;
        }

        vector<E> temp;
        for (int i = 0; i < S.size(); i++)
            temp.push_back(arr[i]);
        S = temp;
    }
    int t2 = clock();
    milliSeconds = (t2-t1)/ (double)CLOCKS_PER_SEC*1000;
}

#endif /* Sorting_h */
```