```
#include "Sort.h"
#include "Comparator.h"

#include <iostream>
#include <cstdlib>
#include <vector>

#include <ctime> //timing
#include <iomanip>   //for set precision

using namespace std;

//generates best case for quick sort using last element as pivot
//moves the median of each subsequence to the last element
template<template<typename> class Seq>
void median_last(Seq<int> &s, int x, int y) {

    //stop algorithm when subsequence is too small
    if (y-x < 3)
        return;

    //calculate middle element
    int z = (x+y)/2;
    int median = s.elemAtRank(z);

    //move median element to end
    s.removeAtRank(z);
    s.insertLast(median);

    //recurse on subsequences
    median_last(s, x, z-1);
    median_last(s, z, y-1);
}

//populates an input sequence given a size and order of elements
template<template<typename> class Seq>
void generate_sequence(string order, int size, Seq<int> &sequence) {

    if (order == "ordered") {
        for (int i = 0; i<size; ++i) {
            sequence.insertLast(i);
        }
    } else if (order == "reverse") {
        for (int i = size-1; i>=0; --i) {
            sequence.insertLast(i);
        }
    } else if (order == "same") {
        for (int i = 0; i<size; ++i) {
            sequence.insertLast(0);
        }
    } else if (order == "median-last") {
        for (int i = 0; i<size; ++i) {
            sequence.insertLast(i);
        }
        median_last(sequence, 0, size-1);
    } else { //random
        for (int i = 0; i<size; ++i) {
            sequence.insertLast(rand()%size);
        }
    }
}

//averages the running time over _num_itr iterations and returns the average time.
template<template<typename, typename> class Sort, template<typename> class Seq>
double run_test(Sort<int, Comp> s, Seq<int> &original_seq, int _num_itr) {

    //save original sequence order
    Seq<int> seq = original_seq;

    //start timing at new tick
    clock_t k = clock();
    clock_t start;
    do start = clock();
```

```
    while (start == k);

    //sort sequence _num_itr times
    for (int itr=0; itr<_num_itr; ++itr) {
        s.sort(seq);
        seq = original_seq; //reset to original sequence
    }

    //end timing
    clock_t end = clock();

    //show size and iterations
    //cout << seq.size() << ", ";
    //cout << _num_itr << ", ";

    double elapsed_time = double(end - start) / double(CLOCKS_PER_SEC);
    //cout << elapsed_time << ", "; //shows total time
    return elapsed_time / double(_num_itr);
}

//runs a test over a range of sizes
template<template<typename, typename> class Sort, template<typename> class Seq>
void initiate_test(Sort<int, Comp> s, Seq<int> seq2, string order, bool slower) {
    size_t lower = 2;
    size_t upper = 16384;
    if (slower)
        upper/=2;

    int size = lower;
    while(size <= upper) {
        int iterations = int(4*upper/size);

        //initialize and populate sequence
        Seq<int> seq;
        generate_sequence(order, size, seq);

        //test sorting algorithm on sequence
        cout << run_test(s, seq, iterations) << endl;

        size *=2;
    }
}

//choose input senquences and initiate tests for each sort
template<template<typename, typename> class Sort, template<typename> class Seq>
bool TestSort(string name) {

    Sort<int, Comp> s;
    Seq<int> seq;

    bool slower = false; //must avoid biggest sizes when testing slow sorts
    vector<string> orders; //dictates order of input sequence
    vector<string> cases {"best", "worst", "average"};

    if (name == "InsertionSort") {
        orders.push_back("ordered"); //best
        orders.push_back("reverse"); //worst
        orders.push_back("random"); //average
        slower = true;
    } else if (name == "SelectionSort") {
        orders.push_back("ordered"); //best
        orders.push_back("reverse"); //worst
        orders.push_back("random"); //average
        slower = true;
    } else if (name == "HeapSort") {
        orders.push_back("ordered"); //average
        orders.push_back("same"); //average
        orders.push_back("random"); //average
    } else if (name == "MergeSort") {
        orders.push_back("ordered"); //average
        orders.push_back("same"); //average
        orders.push_back("random"); //average
    } else if (name == "QuickSortLast") {
```

```
            orders.push_back("median-last"); //best
            orders.push_back("ordered"); //worst
            orders.push_back("random"); //average
        } else if (name == "QuickSortMedium") {
            orders.push_back("median-last"); //best
            orders.push_back("ordered"); //worst
            orders.push_back("random"); //average
        } else if (name == "QuickSortRandom") {
            orders.push_back("median-last"); //best
            orders.push_back("ordered"); //worst
            orders.push_back("random"); //average
        } else if (name == "RadixSort") {
            orders.push_back("ordered"); //average
            orders.push_back("same"); //average
            orders.push_back("random"); //average
        }

        for (int i=0; i<orders.size(); ++i) {
            cout << '\n' << name << endl;
            cout << cases[i] << " case: ";
            cout << orders[i] << " sequence " << endl;
            cout << "Size | Iter | Avg Time" << endl;
            initiate_test(s, seq, orders[i], slower);
        }

        return true;
    }

    int main() {

        bool passed = true;

        if (!TestSort<InsertionSort, NodeSequence>("InsertionSort"))
            passed = false;
        if (!TestSort<SelectionSort, NodeSequence>("SelectionSort"))
            passed = false;
        if (!TestSort<HeapSort, NodeSequence>("HeapSort"))
            passed = false;
        if (!TestSort<MergeSort, NodeSequence>("MergeSort"))
            passed = false;
        if (!TestSort<QuickSortLast, VectorSequence>("QuickSortLast"))
            passed = false;
        if (!TestSort<QuickSortMedian, VectorSequence>("QuickSortMedium"))
            passed = false;
        if (!TestSort<QuickSortRandom, VectorSequence>("QuickSortRandom"))
            passed = false;
        if (!TestSort<RadixSort, VectorSequence>("RadixSort"))
            passed = false;

        if (passed)
            cout << "All tests passed." << endl;
        else
            cout << "Tests failed." << endl;

        return 0;
    }
```

```
#ifndef SORT_H_
#define SORT_H_

#include <cstdlib>
#include "VectorSequence.h"
#include "NodeSequence.h"
#include "SortedSeqPriorityQueue.h"
#include "UnsortedSeqPriorityQueue.h"
#include "HeapPriorityQueue.h"

////////////////////////////////////////////////////////////////////////////
// Priority Queue sorts
////////////////////////////////////////////////////////////////////////////
template<typename PQ>
class PriorityQueueSort {
private:
    typedef typename PQ::ItemPair::element_type Object;
public:
    void sort(NodeSequence<Object>& s) {
        ////////////////////////////////////////////////////////////////////////
        // Implement PQ sort
        ////////////////////////////////////////////////////////////////////////

        PQ pq; //initialize priority queue

        //transfer elements to priority queue
        while (!s.isEmpty()) {
            pq.insertItem(s.elemAtRank(0), s.elemAtRank(0));
            s.removeAtRank(0);
        }

        //return elements to sorted sequence
        while (!pq.isEmpty()) {
            s.insertLast(pq.minElement());
            pq.removeMin();
        }
    }
};

////////////////////////////////////////////////////////////////////////////
// Currently InsertionSort is identical to SelectionSort
//
// Modify to be an actual insertion sort.
////////////////////////////////////////////////////////////////////////////
template <typename Object, typename Comp>
class InsertionSort :
public PriorityQueueSort<SortedSeqPriorityQueue<Object, Object, Comp> >
{};

template <typename Object, typename Comp>
class SelectionSort :
public PriorityQueueSort<UnsortedSeqPriorityQueue<Object, Object, Comp> >
{};

template <typename Object, typename Comp>
class HeapSort :
public PriorityQueueSort<HeapPriorityQueue<Object, Object, Comp> >
{};

////////////////////////////////////////////////////////////////////////////
// Merge Sort
////////////////////////////////////////////////////////////////////////////
template <typename Object, typename Comp>
class MergeSort {
public:
    void sort(NodeSequence<Object>& S);

protected:
    Comp comp;

    //merge S1 and S2 into S
    void merge(NodeSequence<Object>& S1, NodeSequence<Object>& S2, NodeSequence<Object>& S);
```

```
    //append first element of src onto dst
    void appendFirst(NodeSequence<Object>& src, NodeSequence<Object>& dst) {
        Object obj = src.first().element();
        src.remove(src.first());
        dst.insertLast(obj);
    }
};

template <typename Object, typename Comp>
void
MergeSort<Object, Comp>::
sort(NodeSequence<Object>& S) {
    NodeSequence<Object> S1, S2;
    int n = S.size();
    //0 or 1 elements
    if(n <= 1)
        return;

    //copy first half into S1
    int i;
    for(i = n; i > n/2; i--)
        appendFirst(S, S1);
    //copy second half into S2
    for(; i > 0; i--)
        appendFirst(S, S2);  // put remainder in S2
    sort(S1);
    sort(S2);
    merge(S1, S2, S);
}

//merge S1 and S2 into S
template <typename Object, typename Comp>
void
MergeSort<Object, Comp>::
merge(NodeSequence<Object>& S1, NodeSequence<Object>& S2, NodeSequence<Object>& S) {
    //merge lists by selecting least element
    while(!S1.isEmpty() && !S2.isEmpty()) {
        if(comp(S1.first().element(),S2.first().element()) <= 0)
            appendFirst(S1, S);
        else
            appendFirst(S2, S);
    }
    //copy remainder of s1
    while(!S1.isEmpty())
        appendFirst(S1, S);
    //copy remainder of s2
    while(!S2.isEmpty())
        appendFirst(S2, S);
}

////////////////////////////////////////////////////////////////////////////////
// Quick Sorts
////////////////////////////////////////////////////////////////////////////////
template <typename Object, typename Comp, class Pivot>
class QuickSort {
public:
    void sort(VectorSequence<Object>& S);

protected:
    Comp comp;
    Pivot pivot;

    //recursive helper function
    void quickSortStep(VectorSequence<Object>& S, int leftBound, int rightBound);
};

template <typename Object, typename Comp, class Pivot>
void
QuickSort<Object, Comp, Pivot>::
sort(VectorSequence<Object>& S) {
    //0 or 1 elements
    if(S.size() <= 1)
        return;
```

```
    //call recursive helper
    quickSortStep(S, 0, S.size()-1);
}

//recursive helper function
template <typename Object, typename Comp, class Pivot>
void
QuickSort<Object, Comp, Pivot>::
quickSortStep(VectorSequence<Object>& S, int leftBound, int rightBound) {
    //0 or 1 elements
    if(leftBound >= rightBound)
        return;

    //select the pivot and place at end of the list
    int p = pivot.SelectPivot(S, leftBound, rightBound, comp);
    S.swapElements(S.atRank(p), S.atRank(rightBound));
    Object pivotobj = S.atRank(rightBound).element();

    //partition step
    int leftIndex = leftBound;
    int rightIndex = rightBound - 1;
    while(leftIndex <= rightIndex) {
        //scan right to larger element
        while(leftIndex <= rightIndex &&
              comp(S.atRank(leftIndex).element(), pivotobj) <= 0)
            leftIndex++;
        //scan left to smaller element
        while(rightIndex >= leftIndex &&
              comp(S.atRank(rightIndex).element(), pivotobj) >= 0)
            rightIndex--;
        //swap elements
        if(leftIndex < rightIndex)
            S.swapElements(S.atRank(leftIndex), S.atRank(rightIndex));
    }

    // Shift right the right half of the sequence
    // instead of swapping the last element with the middle element
    int i = rightBound;
    while (i > leftIndex) {
        S.replaceAtRank(i, S.atRank(i-1).element());
        i--;
    }
    // pivot at leftIndex
    S.replaceAtRank(leftIndex, pivotobj);

    //recurse
    quickSortStep(S, leftBound, leftIndex-1);
    quickSortStep(S, leftIndex+1, rightBound);
}

// Various Pivots and specific Quick Sorts

struct PivotLast {
    template<typename Seq, typename Comp>
    int SelectPivot(Seq& s, int leftBound, int rightBound, Comp& c) {
        return rightBound;   // select last as pivot
    }
};

template <typename Object, typename Comp>
class QuickSortLast : public QuickSort<Object, Comp, PivotLast> {};

struct PivotMedian {
template<typename Seq, typename Comp>
    int SelectPivot(Seq& s, int leftBound, int rightBound, Comp& c) {
        ////////////////////////////////////////////////////////////////////
        //Implement Median Pivot Selection Here
        ////////////////////////////////////////////////////////////////////

        //get middle index
        int middle = (rightBound+leftBound)/2;

        //find median of left bound, middle, and right bound
```

```
        if (c(s.elemAtRank(leftBound), s.elemAtRank(middle)) < 0) {
            if (c(s.elemAtRank(leftBound), s.elemAtRank(rightBound)) >= 0) {
                return leftBound;
            } else if (c(s.elemAtRank(middle), s.elemAtRank(rightBound)) < 0) {
                return middle;
            }
        } else {
            if (c(s.elemAtRank(leftBound), s.elemAtRank(rightBound)) < 0) {
                return leftBound;
            } else if (c(s.elemAtRank(middle), s.elemAtRank(rightBound)) >= 0) {
                return middle;
            }
        }
        return rightBound;
    }
};

template <typename Object, typename Comp>
class QuickSortMedian : public QuickSort<Object, Comp, PivotMedian> {};

struct PivotRandom{
    template<typename Seq, typename Comp>
    int SelectPivot(Seq& s, int leftBound, int rightBound, Comp& c) {
        ////////////////////////////////////////////////////////////////////
        //Implement Random Pivot Selection Here
        ////////////////////////////////////////////////////////////////////

        //provide a seed
        srand(leftBound+rightBound);

        //return a random value between right bound and left bound
        return leftBound + rand()%(rightBound-leftBound+1);
    }
};

template <typename Object, typename Comp>
class QuickSortRandom : public QuickSort<Object, Comp, PivotRandom> {};

////////////////////////////////////////////////////////////////////////////////
// Radix Sort
////////////////////////////////////////////////////////////////////////////////
template <typename Object, typename Comp>
class RadixSort {
public:
    void sort(VectorSequence<Object>& S);

protected:
    Comp comp;
};

template <typename Object, typename Comp>
void
RadixSort<Object, Comp>::
sort(VectorSequence<Object>& S) {
    int m = 0, exp = 1;
    int n = S.size();
    VectorSequence<Object> b;

    //fill b with last values (just needs to be initialized to right size)
    //also find max
    for(int i = 0; i < n; ++i) {
        b.insertLast(S.elemAtRank(i));
        if(S.elemAtRank(i) > m)
            m = S.elemAtRank(i);
    }

    //Radix sort
    while(m/exp>0) {
        int bucket[10] = {0};
        for(int i = 0; i < n; ++i)
            bucket[S.elemAtRank(i)/exp%10]++;

        for(int i = 1; i < 10; ++i)
```

```
            bucket[i]+=bucket[i-1];

        for(int i = n-1; i >= 0; --i)
            b.replaceAtRank(--bucket[S.elemAtRank(i)/exp%10], S.elemAtRank(i) );

        for(int i=0;i<n;i++)
            S.replaceAtRank( i, b.elemAtRank(i) );

        exp*=10;
    }
}

#endif
```