IB9N7 C++ for Quantitative Finance
Worksheet 14

Sorting and searching
Hints and solutions

18 February 2016
(Week 20)

Objectives for this lab session

By the end of this session, you should have completed the following:

- Look at some examples of sorting and searching.

Exercises

Exercise 1: Set using vectors

The aim of this exercise is to create an object which represents an accumulating set of `int`s. It allows us to remember whether a number has been seen before. It should have two public member functions (i.e. these functions form its interface).

```
void add(int a); //add a to the set
bool isPresent(int a) const; //return whether a is in the set.
```

(a) Create a class `Set1` which implements this interface, by storing all the values which have been added in a `std::vector<int>` data member. `add` should just `push_back` into this member.

(b) Create a class `Set2` which implements this interface, by storing all the values which have been added in a `std::vector<int>` data member which is sorted in ascending order. `add` should insert the new value into the right place, and `isPresent` should check efficiently. Hint: implement both of these functions using `lower_bound`.

Here is both done together.

```
#include<algorithm>
#include<iostream>
#include<vector>

class Set1{
    std::vector<int> m_data;
public:
    void add(int a){
        m_data.push_back(a);
    }
    bool isPresent(int a) const{
        for(int i : m_data){
            if(i == a){
                return true;
            }
        }
```

```
        return false;
    }
};
class Set2{
    std::vector<int> m_data;
public:
    void add(int a){
        auto i = std::lower_bound(m_data.begin(),m_data.end(),a);
        //the following check is optional, but there is no need here to duplicate elements
        if(i==m_data.end() || *i != a){
            m_data.insert(i,a);
        }
    }
    bool isPresent(int a) const{
        auto i = std::lower_bound(m_data.begin(),m_data.end(),a);
        //note how the order of the checks on the next line matters.
        return i!=m_data.end() && *i == a;
    }
};
int main(){
    std::vector<int> a {3,4,5,7,57,12,342};
    Set1 s1; Set2 s2;
    for(auto i : a){
        s1.add(i);
        s2.add(i);
    }
    std::cout<<s1.isPresent(6)<<","<<s1.isPresent(7)<<"\n";
    std::cout<<s2.isPresent(6)<<","<<s2.isPresent(7)<<"\n";
}

```

Exercise 2: Vectors and lists

(a) Setup a vector which contains random integers between 0 and 20 inclusive by copying (and adapting if you want) this:

```
#include<random>
#include<vector>
int main(){
    std::vector<int> a;
    std::mt19937 gen;
    std::uniform_int_distribution<int> uid(0,20);
    for(int i=0; i<200; ++i){
        a.push_back(uid(gen));
    }
}
```

(b) Sort your vector into ascending order.

```
std::sort(a.begin(),a.end());
```

(c) Calculate the median of your vector. (Hints: it is easier to do this without using iterators to begin with. It helps that your vector is already in sorted order.)

Having an even number of elements means that there is more than one definition of median. Here I take the average of the middle pair.

```
double median = 0.5 * (a.at(99)+a.at(100));
```

Extra: consider how you might do this if the vector was not originally sorted. There are several ways. I could use `std::nth_element` to locate the lower of the middle pair, and then find the minimum of the upper half. There is an algorithm `min_element` for finding the minimum value in a range.

```
auto lowerIter = a.begin()+99;
std::nth_element(a.begin(),lowerIter,a.end());
double median1 = 0.5 * (*lowerIter + *std::min_element(lowerIter+1,a.end()));
```

(d) Now that your vector is sorted, insert the number '10' in the appropriate place to maintain the sorted invariant.

(Note that you should wrap this feature into a function instead of doing it directly in your `main` function.) The function is

```
void insertValueInSortedPosition(std::vector<int>& a, int newValue){
    a.insert(std::lower_bound(a.begin(),a.end(),newValue),newValue);
}
```

and we will call

```
insertValueInSortedPosition(a,10);
```

(e) Remove all odd numbers from your vector.

Take care with respect to iterator invalidation.

Hint: The slow but safe way to do this is to restart your search at the beginning of the vector once you have removed an element. Once you have done that, try to find a more efficient but equally valid method. You could call the function you wrote for this in the "extra" version of worksheet 6 (week 8 of last term), for example this

```
void remove_odd_elements(std::vector<int>& v){
    size_t number_skipped = 0u;
    for(size_t source=0;source<v.size();++source){
        if(v.at(source)%2 == 1){
            ++number_skipped;
        }else{
            v.at(source-number_skipped) = v.at(source);
        }
    }
    v.resize(v.size()-number_skipped);
}
```

In fact, the STL provides a feature which can make doing this a bit simpler. It is called the *erase-remove idiom* and looks like this. It has its own wikipedia page.

```
bool isOdd(int v){
    return v%2;
}
void remove_odd_elements(std::vector<int>& v){
    v.erase(std::remove_if(v.begin(),v.end(), isOdd), v.end());
}
```

This relies on the version of `erase` which takes two iterators, which removes a whole range of elements from a vector.

(f) Determine whether the numbers 15 or 20 each appear in your vector (just "yes" or "no" will do). You may experiment with the `std::find` algorithm if you want.

```
bool has15 = std::find(a.begin(),a.end(),15) != a.end();
bool has20 = std::find(a.begin(),a.end(),20) != a.end();
```

(g) If the above numbers appear, multiply **all** instances of them by 2, replacing the original copies in the vector.

```
for(int &i : a)
    if(i==15 || i==20)
        i *= 2;
```

(h) Now try to do all of the above using a list instead of a vector.

When would you use a list instead of a vector?

See below for a full listing of both solutions. Note that the only significant differences are that I use the `sort` member of the list, not the `std::sort` algorithm, and that I have to do some iterator arithmetic to find the 100th and 101st elements.

As per the lecture: list is easier when frequent insertions or removals are required, especially at positions other than just the back. Equally, if it is necessary to splice or merge.

Otherwise, if memory is constrained and you don't know how many elements you need ahead of time, you can either use a vector of pointers (where the extra memory use will be smaller), or a list where there is no redundant memory allocated (c.f. vectors where the capacity is usually greater than the size).

Performance of accessing elements may differ too, due to memory locality and cache behaviour.

Iterators for lists are more stable than those for vectors and so may be more convenient.

Which is easiest or most appropriate for calculating the median?

It is easier to do this with a vector, but it is not impossible to do so with a list, as shown. Finding the middle element(s) takes effort with a list, but is trivial with a vector.

```
#include<random>
#include<iostream>
#include<algorithm>
#include<vector>
#include<list>

void insertValueInSortedPosition(std::vector<int>& a, int newValue){
    a.insert(std::lower_bound(a.begin(),a.end(),newValue),newValue);
}

bool isOdd(int v){
    return v%2;
}
void remove_odd_elements(std::vector<int>& v){
    v.erase(std::remove_if(v.begin(),v.end(), isOdd), v.end());
}

void insertValueInSortedPosition(std::list<int>& a, int newValue){
    a.insert(std::lower_bound(a.begin(),a.end(),newValue),newValue);
}
void remove_odd_elements(std::list<int>& v){
    v.erase(std::remove_if(v.begin(),v.end(), isOdd), v.end());
}

void doVector(){
    std::vector<int> a;
    std::mt19937 gen;
    std::uniform_int_distribution<int> uid(0,20);
    for(int i=0; i<200; ++i){
        a.push_back(uid(gen));
    }
    std::sort(a.begin(),a.end());
    double median = 0.5 * (a.at(99)+a.at(100));

    insertValueInSortedPosition(a,10);

    bool has15 = std::find(a.begin(),a.end(),15) != a.end();
    bool has20 = std::find(a.begin(),a.end(),20) != a.end();
    for(int &i : a)
        if(i==15 || i==20)
            i *= 2;
}
void doList(){
```

```
    std::list<int> a;
    std::mt19937 gen;
    std::uniform_int_distribution<int> uid(0,20);
    for(int i=0; i<200; ++i){
        a.push_back(uid(gen));
    }
    a.sort();
    auto a99 = a.begin();
    for(int count = 0; count<99; ++count){
        ++a99;
    }
    auto a100 = a99;
    ++a100;
    double median = 0.5 * (*a99 + *a100);

    insertValueInSortedPosition(a,10);

    bool has15 = std::find(a.begin(),a.end(),15) != a.end();
    bool has20 = std::find(a.begin(),a.end(),20) != a.end();
    for(int &i : a)
        if(i==15 || i==20)
            i *= 2;
}

int main(){
    doVector();
    doList();
}
```