Practical 4 – Vector Searching and Sorting

Download the ***aeda2021_p04.zip*** file from the moodle page and unzip it (contains the ***lib*** folder, the ***Tests*** folder with files ***carPark.h, carPark.cpp***, ***sequentialSearch.h, insertionSort.h*** e ***tests.cpp***, e os ficheiros ***CMakeLists*** e ***main.cpp***)

- Note that some unit tests for this project are commented (*b* and *f*). Remove the comments when you implement the tests.
- You must perform the exercise respecting the order of the questions.

Exercise

1. Consider the problem of managing a car park, already stated in practical 1. The **InfoCard** and **CarPark** classes are shown below, with new members now:

```
class InfoCard {
public:
   string name;
   int frequency;  //new data member
   bool present;
};

class CarPark {
  unsigned freePlaces;
   const unsigned capacity;
   vector<InfoCard> clients;
   const unsigned numMaxClients;
public:
   CarPark(unsigned cap, unsigned nMaxCli);
   unsigned getNumPlaces() const;
   unsigned getNumMaxClients() const;
   unsigned getNumOccupiedPlaces() const;
   unsigned getNumClients() const;
   vector<InfoCard> getClients() const;
   bool addClient(const string & name);
   bool removeClient(const string & name);
   bool enter(const string & name);
   bool leave(const string & name);
   int clientPosition(const string & name) const;
   int getFrequency(const string &name) const;
   InfoCard getClientAtPos(unsigned p) const;
   void sortClientsByFrequency();
   void sortClientsByName();
   vector<string> clientsBetween(unsigned f1, unsigned f2);
   friend ostream & operator<<(ostream & os, const CarPark & cp);
};
```

a) Reimplement the method:

```
int CarPark::clientPosition(const string &name) const
```

which returns the client index of name *name* in the clients vector. If the client does not exist, it returns -1. To perform the search in the clients vector, use the **sequential search method** studied in class (code in *sequentialSearch.h*).

b) In the **InfoCard** class, the new data member *frequency* stores the number of times the client has used the car park. Modify the methods already implemented in order to conveniently update this new data member. Also implement the method:

```
int CarPark::getFrequency(const string &name) const
```

which returns the number of times the client named *name* used the car park. If the client does not exist, it throws an exception of type *ClientDoesNotExist*.

Implement the *ClientDoesNotExist* class and **note that** the handling of this exception makes a call to the method *getName()* which should return the name of the client that does not exist and originated the exception.

c) Implement the method:

```
void CarPark:: sortClientsByFrequency()
```

which sorts the clients vector in decreasing order of frequency of use of the car park, disambiguating (customers with the same frequency) in ascending order of name. Use the **insertion sorting method** studied in class (code in *insertionSort.h*).

d) Implemente the method:

```
vector<string> CarPark::clientsBetween(unsigned f1, unsigned f2)
```

which returns a vector with the name of all clients who used the park> = *f1* times and <= *f2* times (note: the vector to be returned must be ordered according to the criteria stated in c).

e) It is sometimes useful to obtain information from clients sorted by name. Using a sorting algorithm of your choice, implement the method:

```
void CarPark::sortClientsByName()
```

which sorts the clients vector in ascending order of name.

f) Implement the operator << :

```
ostream & operator << (ostream &os, const CarPark &cp)
```

This function prints on the monitor information about all clients, showing the name of the client, whether he is present at the car park or not and the number of times he has used the car.

Also implement the method:

```
InfoCard CarPark::getClientAtPos(unsigned p) const
```

which returns the client (*InfoCard*) existing in the *p* index of the *clients* vector. If there is no such client, throws an exception of type *PositionDoesNotExist*.

Implement the *PositionDoesNotExist* class and **note that** the handling of this exception makes a call to the method *getPosition()* that should return the invalid position that originated the exception.

2. Analyse the temporal and spatial complexity of the following code:

a)
```
void imprime_matriz(int largura, int altura, int ntabs) {
    num = 1;
    for(int a = 1 ; a <= altura ; a++) {
        cout << "[";
        for(int l = 1 ; l <= largura ; l++) {
            cout << num++;
            for(int t = 1 ; t <= ntabs ; t++) cout << "\t";
        }
       cout << "]" << endl;
    }
    cout << endl;
}
```

b)
```
int pesquisa (int v[], int size, int x) {
    int left = 0;
    int right = size-1;
    while (left <= right) {
        int middle = (left + right) / 2;
        if (x == v[middle])
            return middle; // encontrou
        else if (x > v[middle])
            left = middle + 1;
        else
            right = middle -1;
    }
    return -1;
}
```