Assessment 2: Assignment 1

William Hall

***this document contains a lot of pseudocode and miniature descriptions of implementations*

Task Outline

This task is to implement an altered dataframe with certain constraints in C++. Dataframes are traditionally 2-dimensional labelled data structures, this can have differing types as the data and can be labelled on both columns and rows. Traditionally you can perform arithmetic operations on either rows or columns. The constraints for this dataframe alter a dataframe and make all data within the dataframe the same type. Only the columns will be labelled, and you can perform arithmetic operations on only columns.

The dataframe class is required to be implemented as a generic(e.g., template<typename T>) to accommodate for differing types.

Implementation

Class Dataframe

The dataframe is implemented as a template class and has three private attributes: columnNames, which is a vector of strings storing the names of each column. Data_vec which is a vector of vectors storing the data. And contSize which tracks the current container size. The methods for this class are outlined below which is just .

resize_containers();

Adjusts the size of the containers to the current contSize. It does this via the built in function inside of std::vector<T>.

Pseudocode:

```
data_vec -> resize(contSize)

column_names -> resize(contSize)
```

Complexity: O(n) where n is the size of the resized vector

indexcheck(int)

Checks if the index will be inside of the containers

Pseudocode:

```
indexcheck(int i)
If( i < 0 or i > contSize-1 )
return false
else
return true
```

Complexity: O(1);

printData()

Prints the contents of both containers to the screen

Pseudocode:

```
For(nameItem in columnNames and row in data_vec){
	print(nameItem)
	For(dataItem in row of data_vec){
		print(dataItem)
	}
}
```

Complexity: O(n^2) where n is each item

DataFrame()

Initializes an empty dataframe

Pseudocode:

```
DataFrame(){
	contSize = 0;
	data_vec = new data_vec
	columnNames = new columnNames
}
```

Complexity: O(n) where n is the size of the array although it can be assumed to be O(1) due to n = 0

DataFrame(vector<vector>)

Initializes the dataframe with a vector of vectors.

Pseudocode:

```
DataFrame(vector<vector> vec){
	contSize = vec.size()
	resize_containers()
	For(i = 0 to contSize -1)){
		Data_vec[i].resize(size(vector[i])
		For(j = 0 to size(vector[i]) -1)){
			Data_vec[i][j] = Vector[i][j]
		}
	}
}
```

Complexity: O(n^2) where n is each entry in the vec

load_data(vector<vector> vec)

Erases old data and copies this data into the dataframe

Pseudocode:

```
load_data(vector<vector> vec){
	data_vec->clear()
	columnNames->clear()
	resize_containers()
	For(i = 0 to contSize -1)){
		Data_vec[i].resize(size(vector[i])
		For(j = 0 to size(vector[i]) -1)){
			Data_vec[i][j] = Vector[i][j]
```

```
            }
      }
}
```

Complexity: O(n^2) where n is each entry in the vec

add_data(vector<T>)

this adds a single vector to the existing dataframe

```
add_data(vector<T> vec){
      contSize++
      resize_containers()
      iter = data_vec->end()-1
      for(i =  0 to vector.size()-1){
            it->push_back(vector[i])
      }
}
```

Complexity: O(n) where n is the entries in the vec

add_data(vector<vector>)

adds a vector of vectors into the existing dataframe

```
add_data(vector<vector> vec){
      contSize = contSize+vec.size();
      resize_containers()
      For(i = contSize-vec.size() to contSize -1)){
            Data_vec[i].resize(size(vector[i))
            For(j = 0 to size(vector[i]) -1)){
                  Data_vec[i][j] = Vector[i][j]
            }
      }
}
```

Complexity: O(n^2) where n is each entry in the vec

add_data(vector<T>, string)

adds a vector to the existing dataframe with a columnName

```
add_data(vector<T> vec, string columnName){
	Add_data(vec)
	columnNames[columnNames->size()-1] = columnName
}
```

Complexity: O(n) this is due to add_data(vector<T>) taking O(n)

add_data(vector<vector>, string[])

adds a vector of vectors with corresponding strings, must be equal vectors and strings

```
add_data(vector<vector> vec, string[] columnNames){
	try{
		add_data(vectors)
		for(i = old_contsize : contSize){
			add columnName[i] to columnNames
		}
	}
	Exception{
		Print(must be equal strings and vectors)
	}
}
```

Complexity: O(n^2) due to add_data(vector<vector>)

set_columns(vector<string>, int)

set all the column names in column_names to the values of columnNames

```
set_columns(column_names[], size(column_names)){
```

```
	check that size(column_names) == size(columnNames)
	for(i = old_contsize : contSize){
		set columnNames[i] = column_names[i]
	}
}
```

Complexity: O(n) where n is each entry in column_names

update_column(string, string)

sets the second string to the location of the second string

```
update_column(string old, string new){
	bool buff = find(old) in columnNames
	if (buff){
		set new to location of old
	}else{
		Print(Error: bad name)
	}
}
```

Complexity: O(n) is the complexity of find

update_column(int, string)

at index ‘int’ set the column name to ‘string’

```
update_column(int index, string name){
	if(index is in the bounds of columnNames){
		columnNames[index] = name
	}else{
		Print(Error: bad index)
	}
}
```

Complexity: O(1) simple checks and setting a value

update_columnval(int, vector<T>)

at index ‘int’ in the data_vec set the vector to ‘vector<T>’

```
update_columnval(int index, vector<T> vec){
	if(index is in the bounds of data_vec){
		data_vec[index] = name
	}else{
		Print(Error: bad index)
	}
}
```

Complexity: O(1) simple checks and setting a value

update_columnval(string, vector<T>)

Does the same as the method above although finds index via find() function by using a string

delete_column(int) and delete_column(string), *same function although index is found via different methods*

finds given index and deletes column associated with that index along with column_name

```
delete_column(int index){
	while(index != last_index){
		swap current vector with vector at index+1
	}
	destroy(data_vec->last_index)
	contSize = contSize-1
	resize_containers()
}
```

Complexity: O(n) where n is the amount of swaps done to move the vector to the last index

operator [](string) and operator [](int)

find the vector at the given location and returns it as a class Vec

```
operator [](string column_name){
	int index = find(column_name in columnNames)
	if(index){
		return Vec(data_vec->at(index))
	}else{
		Print(Error)
	}
}
```

Complexity: O(n) is the complexity of find

size()

Returns the number of columns in the data_vec

```
size(){
	int count = 0
	for(column in data_vec){
		count = count + size(column);
	}
	Return count
}
```

Complexity: O(n) where n is each column

shape()

Returns the number of columns in the data_vec

```
shape(){
	return size(data_vec)
}
```

Complexity: O(1)

Class Vec

Mentioned in the overloaded operator methods within the dataframe class is the Vec class. This class extended the std::vector STL class and adds new methods, such as min, max, mean, median, mode and summary. These methods perform statistical operations on the vector.

Min()

Finds the smallest element in the vector using the min_element function.

```
Min(){
        Return min_element(container.begin(), container.end())
}
```

Complexity: O(n-1) comparisons

Max()

Finds the largest element in the vector using the max_element function.

```
Max(){
        Return max_element(container.begin(), container.end())
}
```

Complexity: O(n-1) comparisons

Mean()

Finds the mean of the vector (Accumulate calculates the sum between two iterators)

```
Mean(){
        If(numberOfElements !=0){
                Return Accumulate(container.begin, container.end(), 0.0)/numberOfElements
        }else{
                Return 0
        }
}
```

Complexity: O(N) is the complexity of accumulate

Median()

Calculates the median of the vector using the nth element method

```
Median(){
        Mid = size(container)/2
        Nth_element(container.begin, container.begin()+mid, container.end())
        contAtN = container[mid]
        if(container is not even){
                nth_element(container.begin(), container.begin()+mid-1, container.end())
                return contAtN+container[mid-1]/2
        }else{
                Return contAtN
        }
}
```

Complexity: O(n) as this is the complexity of the nth element method

Mode()

Calculates and returns the mode of the vector

```
Mode(){
        Mp = map of int to int
        For(I = 0 to size(container)){
                Mp[container[i]]++ (Add one to each occurrence in the map)
        }
        Return max_element of the second items being the comparison
}
```

Complexity: O(2n) ~ O(n) as looping through container takes n and max element takes n

Discussion/Possible Improvements

Possible improvements on this could be to make the data_vec container inside Dataframe a vector of Vec this would reduce copying when calling the overloaded operator this was not done as it increases readability by reducing this additionally Vec is conceptually restrained to be either a float or integral type. By not doing this the dataframe can be made of strings or other datatypes.

Order was maintained when deleting a column from the data_vec as such a large amount of swaps occurred. If order was not required this would reduce complexity. Although this would make the dataframe more confusing as columns would move unknowingly to the user when deleting.

The implementation of the Vec class is somewhat not recommended as it uses inheritance to extend the std::vector class. This is difficult to overcome as C++ does not directly allow extending of classes. The class also must be inherited as public as private and protected prevent the user from performing normal methods on the vector.