ASSIGNMENT 2

Serhat Gündem
2200356820
BBM203
03-12-2021

Project description and Project goals:

1)Cover: The project is about creating an "Employee Recording System" application by using OOP(Object Oriented Programming) principles. In this project, you're asked to store and take action on employees that you have to create as an object. There will be 2 kinds of employee, and for each kind of employee, you're asked to design a different data structure. Using these data structures, the employees created by the user will be stored. After storing, they will be asked to be sorted in some of their attributes such as employee number and appointment date to the institution. Two data structures will be designed for this assignment: Circular Linked List with array implementation to store temporary employees in order of their employee numbers and, doubly linked list to store permanent employees in order of their appointment date to the institution. In command section, you'll be asked to take some different actions such as listing all the employees by one of their attribute. Moreover, when comparing two user-defined object, you're asked to overload some operators such as '<' , '>', '='.

2)Introduction: This project's goal is to teach more about OOP principles and algorithms behind a circular linked list with array implementation and doubly linked list. Besides them, you'll be getting familiar with overloading operators, typecasting, pointers etc.

3)Method: My methods to solve this problem was first decide what to do before starting the problem. After that, I've done a depth research about implementing circular array linked list, and gained the algorithm behind it. It was simple but when it is tried to implement using arrays, that's when things get complicated for me. Yet, I figured it out after a while – I'll be explaining the algorithm behind it in later part of the report. After completing implementation of circular array data structure using array, I moved onto implementing doubly linked list structure. These structures are the foundation of this problem. After I am done with them, things get much easier. Then, I created Employee class with the specific attributes belong to it and inherited it and created 2 subclass :Temporary Employee and Permanent Employee. After that, I created a Date class to use inside Employee objects. After all, there was only the commands that I had to deal with. The most challenging one was the 6th command which asks me to sort these employees together by their employee number. Yet, the way the other commands should be implemented was just all about the 6th command, and since I completed it, I could easily implement the other command only modifying some parts of the 6th command.

4) Development:

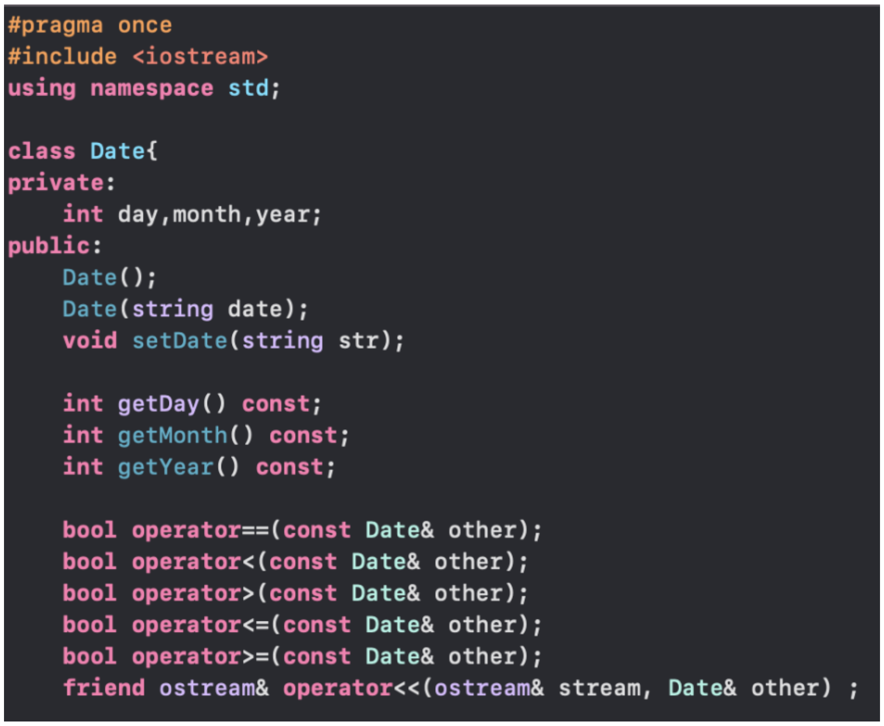
1- I already explained my plan above, but in short, the plan was first implement the data structures that will be required to store employees and create the actual employee class as well as its subclasses. Since we were required to use only Circular linked list using arrays and doubly-linked lists, the first thing to do was to implement them. After all, there was only one thing to do which is reading the commands and applying them.

2- I've first done how can I implement these classes with the most efficient way, the functions of that data structures (insert, remove, getEmployee) have to be efficient. In introduction part, I stated the problem in my words.

3- My solution is actually simple enough, first implement the Circular Array Linked List class and Doubly Linked List class. And, since we're asked to create an employee recording system, we should keep the data of the employees and the only efficient way to do this was to use OOP principle so I decided to create an Employee class with the specific attributes as well as a Date class to keep track of the some dates that belong to an Employee object. And then in commands part, since we're asked to sort all these employees by some of their attributes, I should find an efficient way. So I used a technique that is also used in Merge Sort which is divide and conquer. Yet, the difference was I am not dividing the list from the middle because I cannot do so since this is a user-defined data structure (kind of a linked list) we cannot apply that technique since we cannot go directly to the middle of the linked list, yet in arrays this is possible. So, only difference was this in fact, after that, I've chosen one employee from each list (one is to store permanent employees and one is to store temporary employees) and compare them. Whichever is less than the other is printed first, then we do the same process excluding the employee that is already printed.

I could've implement circular linked list dynamically and that'd be so much easier for me to implement but we're not allowed to do so, that's why, I had to eliminate that method.

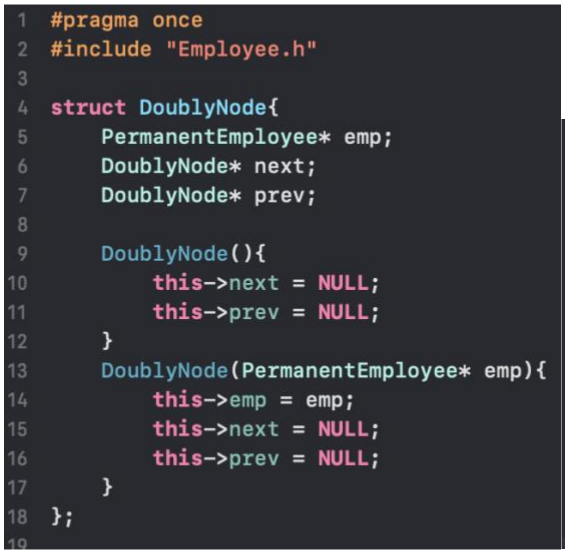
I used 4 interfaces (was not in need of creating separate header files for the subclasses of the employee class since they didn't have much features and since they are inherited from the employee class. These interfaces were: DoubleDynamicLinkedList.h, CircularArrayLinkedList, Date.h, Employee.h.



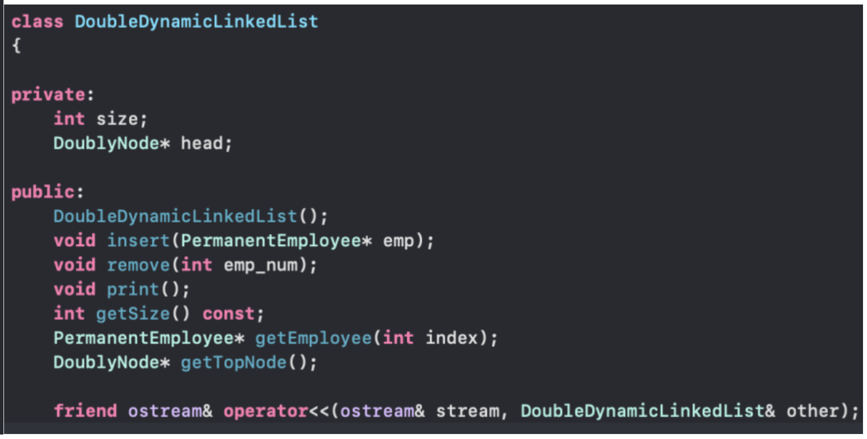
```cpp
#pragma once
#include <iostream>
using namespace std;

class Date{
private:
    int day,month,year;
public:
    Date();
    Date(string date);
    void setDate(string str);

    int getDay() const;
    int getMonth() const;
    int getYear() const;

    bool operator==(const Date& other);
    bool operator<(const Date& other);
    bool operator>(const Date& other);
    bool operator<=(const Date& other);
    bool operator>=(const Date& other);
    friend ostream& operator<<(ostream& stream, Date& other) ;
```

The figure above represents Date.h interface. I declared a class inside it and had its attributes private but since its attributes are set after its created (to avoid complexity of the code in main file) they were not declared as const. In public part, we have 4 method whose 3 of them are const since they don't change anything. And one of them is doing the main thing which is setting the day, month and year attributes of a date object. It takes the date as string and then divides it into part with sub_string method from std library and assign them to the specific variables.

At the end we have the overloaded operators declarations which 5 of them compares 2 date objects and 1 of them just prints that date object.

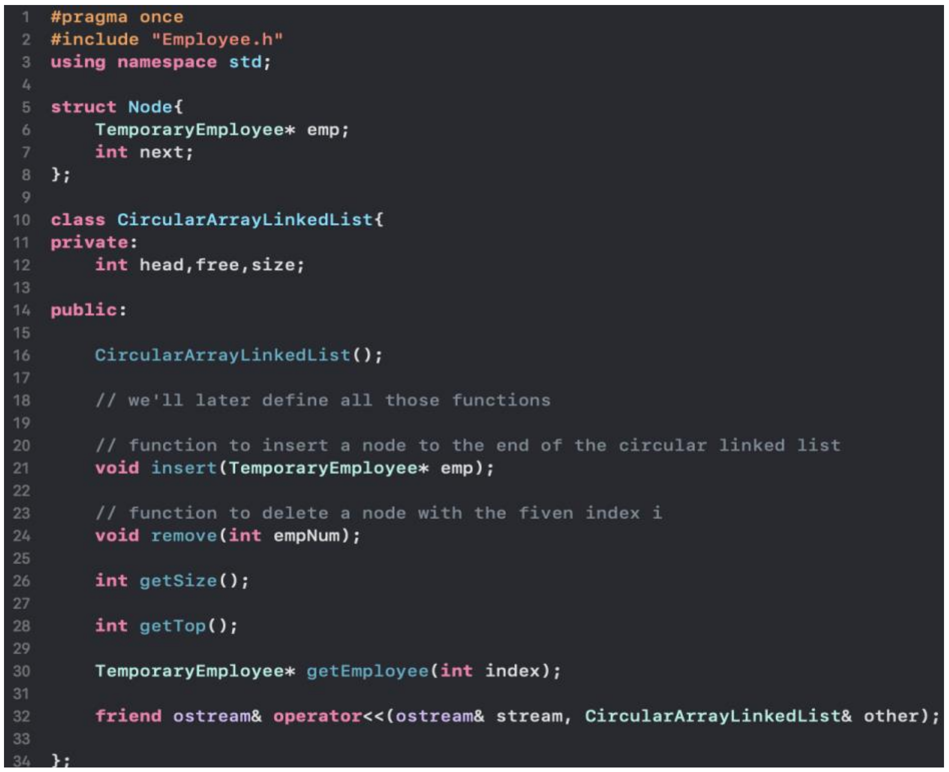


```cpp
#pragma once
#include "Employee.h"

struct DoublyNode{
    PermanentEmployee* emp;
    DoublyNode* next;
    DoublyNode* prev;

    DoublyNode(){
        this->next = NULL;
        this->prev = NULL;
    }
    DoublyNode(PermanentEmployee* emp){
        this->emp = emp;
        this->next = NULL;
        this->prev = NULL;
    }
};
```



```cpp
class DoubleDynamicLinkedList
{

private:
    int size;
    DoublyNode* head;

public:
    DoubleDynamicLinkedList();
    void insert(PermanentEmployee* emp);
    void remove(int emp_num);
    void print();
    int getSize() const;
    PermanentEmployee* getEmployee(int index);
    DoublyNode* getTopNode();

    friend ostream& operator<<(ostream& stream, DoubleDynamicLinkedList& other);
```

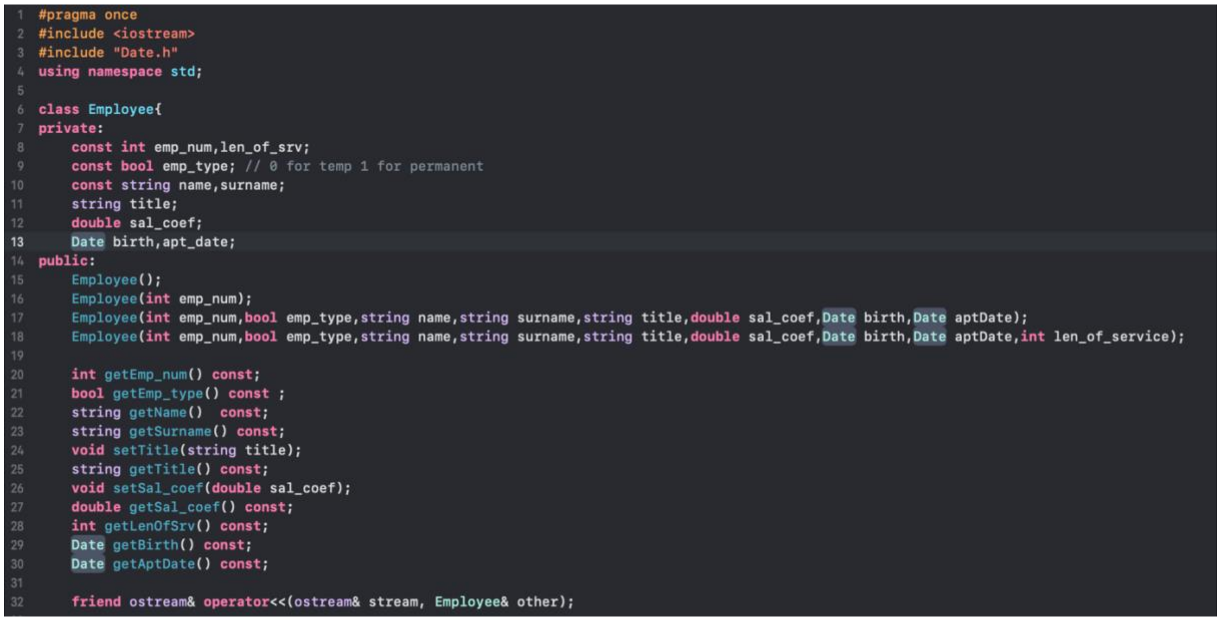
And the code above represents DoubleDynamicLinkedList.h interface. It has a struct node in which we will store our values. Since it is a doubly linked list,every node should keep its previous and next node as well as the data which is the permanent employee pointer in this case.Inside DoubleDynamicLinkedList class declaration, we have our attributes private and some methods in public as well as a constructor. In this class declaration, we also declared insert method which inserts an employee taken as parameter by its attribute date in ascending order to the double dynamic linked list. And we have remove method which removes the employee with the specific employee number that is also taken as parameter. We have a print method that I used to use before having the idea of operator overloading (deleted it anyway). We have getEmployee method which takes an integer number that represent the index of the employee that is asked for, and it just basically returns that employee. Also we have the getoTopNode() method that I used to get the head of the linked list. At the and we have the operator overloading (<<) to print the content of the double dynamic linked list.



```cpp
#pragma once
#include "Employee.h"
using namespace std;

struct Node{
    TemporaryEmployee* emp;
    int next;
};

class CircularArrayLinkedList{
private:
    int head,free,size;

public:

    CircularArrayLinkedList();

    // we'll later define all those functions

    // function to insert a node to the end of the circular linked list
    void insert(TemporaryEmployee* emp);

    // function to delete a node with the fiven index i
    void remove(int empNum);

    int getSize();

    int getTop();

    TemporaryEmployee* getEmployee(int index);

    friend ostream& operator<<(ostream& stream, CircularArrayLinkedList& other);

};
```

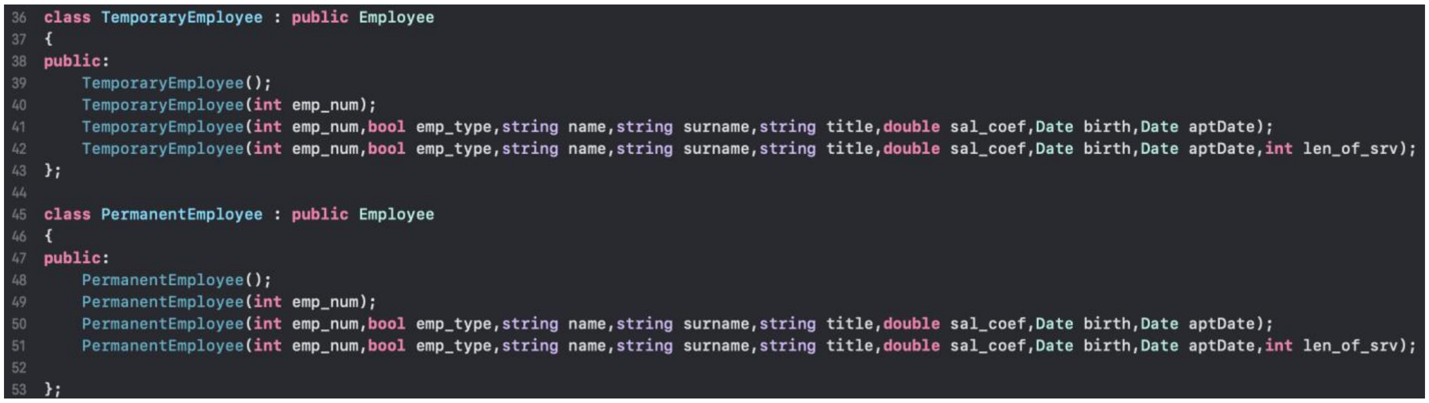
Above code represents the CircularArrayLinkedList.h interface. In this header file, we have a struch node that includes data which is temporary employee and next index of the next node.

In CircularArrayLinkedList class we have 3 attributes as private: head (to keep track of the head of the array), free (to keep track of the next free spot in the array), size (to keep track of the size of the array). In public declarations of the circularArrayLinkedList class we have 5 methods: insert (which inserts the employees to the array in order of their employee number in ascending order), remove (which removes the employee with the specific employee number taken as paramer), getSize()which returns the size of the array, getTop() (which returns the index of the top employee), getEMployee() which returns the employee that is asked to be in the index given as the parameter.



```cpp
#pragma once
#include <iostream>
#include "Date.h"
using namespace std;

class Employee{
private:
    const int emp_num,len_of_srv;
    const bool emp_type; // 0 for temp 1 for permanent
    const string name,surname;
    string title;
    double sal_coef;
    Date birth,apt_date;
public:
    Employee();
    Employee(int emp_num);
    Employee(int emp_num,bool emp_type,string name,string surname,string title,double sal_coef,Date birth,Date aptDate);
    Employee(int emp_num,bool emp_type,string name,string surname,string title,double sal_coef,Date birth,Date aptDate,int len_of_service);

    int getEmp_num() const;
    bool getEmp_type() const ;
    string getName()  const;
    string getSurname() const;
    void setTitle(string title);
    string getTitle() const;
    void setSal_coef(double sal_coef);
    double getSal_coef() const;
    int getLenOfSrv() const;
    Date getBirth() const;
    Date getAptDate() const;

    friend ostream& operator<<(ostream& stream, Employee& other);
```

Above the code represents Employee.h header file. In this file, we have declared the Employee class as well as its attributes and methods. The attributes that are not required to change are set const as well as the methods. Yet, we cannot set title and salary coefficient const since we may have to change them if needed. Inside the public part we have 4 constructors (every of them used in main source file) following by some getter and setter methods (getter methods are also set const since they're not required to change anything. At the end we have a operator overloading to print the content of an employee.

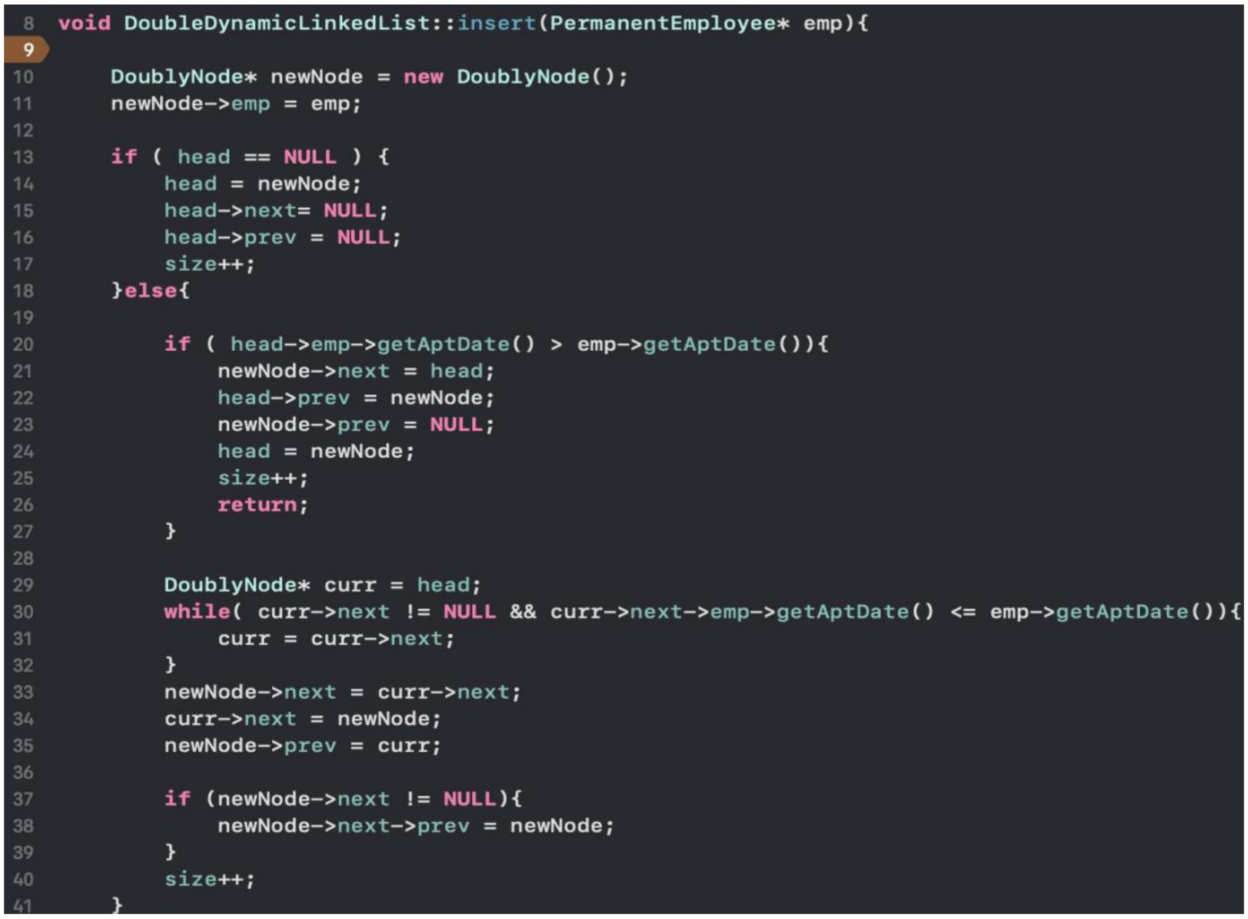


```cpp
class TemporaryEmployee : public Employee
{
public:
    TemporaryEmployee();
    TemporaryEmployee(int emp_num);
    TemporaryEmployee(int emp_num,bool emp_type,string name,string surname,string title,double sal_coef,Date birth,Date aptDate);
    TemporaryEmployee(int emp_num,bool emp_type,string name,string surname,string title,double sal_coef,Date birth,Date aptDate,int len_of_srv);
};

class PermanentEmployee : public Employee
{
public:
    PermanentEmployee();
    PermanentEmployee(int emp_num);
    PermanentEmployee(int emp_num,bool emp_type,string name,string surname,string title,double sal_coef,Date birth,Date aptDate);
    PermanentEmployee(int emp_num,bool emp_type,string name,string surname,string title,double sal_coef,Date birth,Date aptDate,int len_of_srv);

};
```

The above code is the continuation of Employee.h header file. In this part, subclasses of the Employee class are declared by inheriting employee class.

4 – Implementation:

I will first be explaining DoubleDynamicLinkedList class' functions in this report.



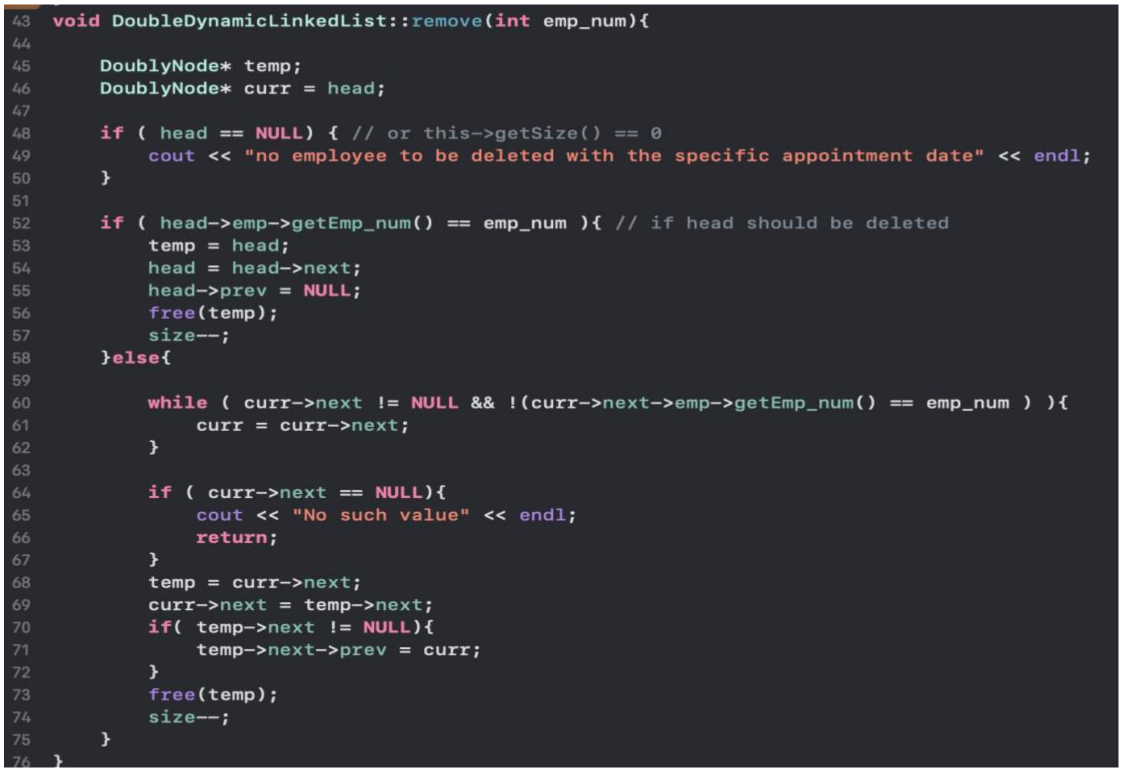
```cpp
void DoubleDynamicLinkedList::insert(PermanentEmployee* emp){

    DoublyNode* newNode = new DoublyNode();
    newNode->emp = emp;

    if ( head == NULL ) {
        head = newNode;
        head->next= NULL;
        head->prev = NULL;
        size++;
    }else{

        if ( head->emp->getAptDate() > emp->getAptDate()){
            newNode->next = head;
            head->prev = newNode;
            newNode->prev = NULL;
            head = newNode;
            size++;
            return;
        }

        DoublyNode* curr = head;
        while( curr->next != NULL && curr->next->emp->getAptDate() <= emp->getAptDate()){
            curr = curr->next;
        }
        newNode->next = curr->next;
        curr->next = newNode;
        newNode->prev = curr;

        if (newNode->next != NULL){
            newNode->next->prev = newNode;
        }
        size++;
    }
```

Insert function takes a permanent employee parameter and inserts it into the doubly linked list. First, a new node is created and set the employee as its data.

If the head of the doubly linked list is null which means the list is empty then we should set the new node as the head of the list as well as setting its next and previous nodes NULL and increase size by 1.

If the list is not empty then check if the appointment date of the employee that will be added is smaller than the head employee's employee number, if it is then, insert the new employee to the head. First set its next as head and head's prev as that new node. Then set new head as that new node and increase size by 1.

If the new employee's emp_num is not smaller than the head's emp_num then traverse through the list and find the correct spot for the new node according to its appointment date. Set curr as the head node and while next node of curr is not equal to NULL and appointment date of the employee that the next node of the curr is less than or equal to the new employees's appointment date, set curr as the next node of curr. Then insert it after curr, at the end, if newNode's next is not NULL, then we should connect them to not lose track of the remaining nodes.



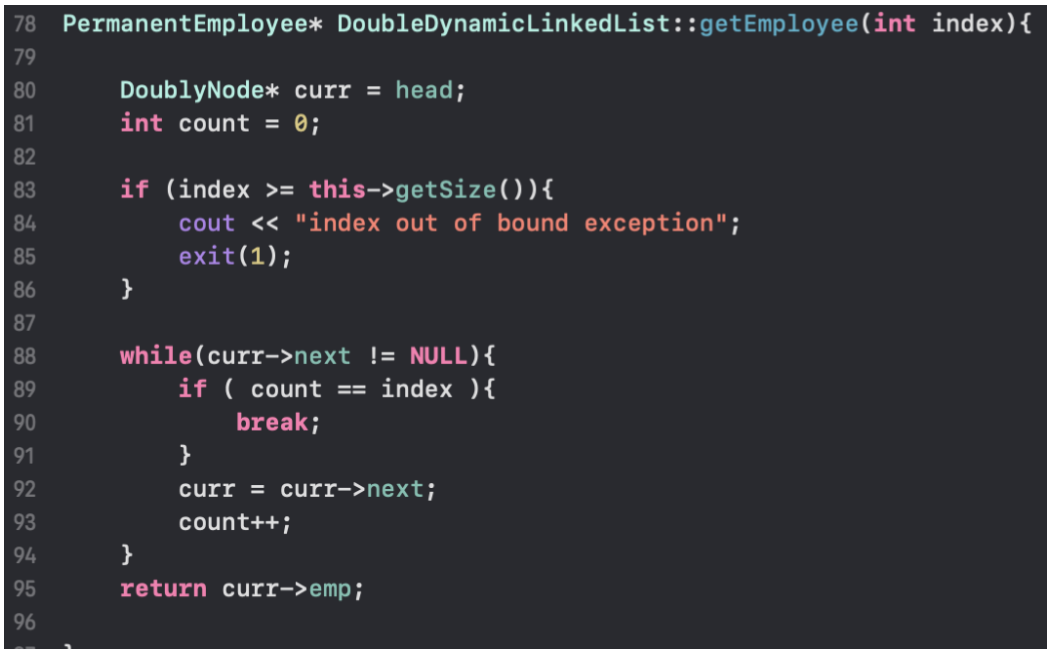
```cpp
void DoubleDynamicLinkedList::remove(int emp_num){

    DoublyNode* temp;
    DoublyNode* curr = head;

    if ( head == NULL) { // or this->getSize() == 0
        cout << "no employee to be deleted with the specific appointment date" << endl;
    }

    if ( head->emp->getEmp_num() == emp_num ){ // if head should be deleted
        temp = head;
        head = head->next;
        head->prev = NULL;
        free(temp);
        size--;
    }else{

        while ( curr->next != NULL && !(curr->next->emp->getEmp_num() == emp_num ) ){
            curr = curr->next;
        }

        if ( curr->next == NULL){
            cout << "No such value" << endl;
            return;
        }
        temp = curr->next;
        curr->next = temp->next;
        if( temp->next != NULL){
            temp->next->prev = curr;
        }
        free(temp);
        size--;
    }
}
```

Remove function takes an integer parameter which represents the employee number that is required to remove. Temp node is created to store the removed node, and curr node is created to traverse through the list starting from the head.

If head of the list is null, then there is no employee to remove so print a message and exit(1), since we are not responsible for catching error, I didn't think about it much. If the employee that will be removed is the head of the list then assign head to temp and head = head's next node, and assign head's prev as NULL (we do not need to do that since when we initialize a node its prev node is set to NULL as default.) After that, free the node the save the memory.

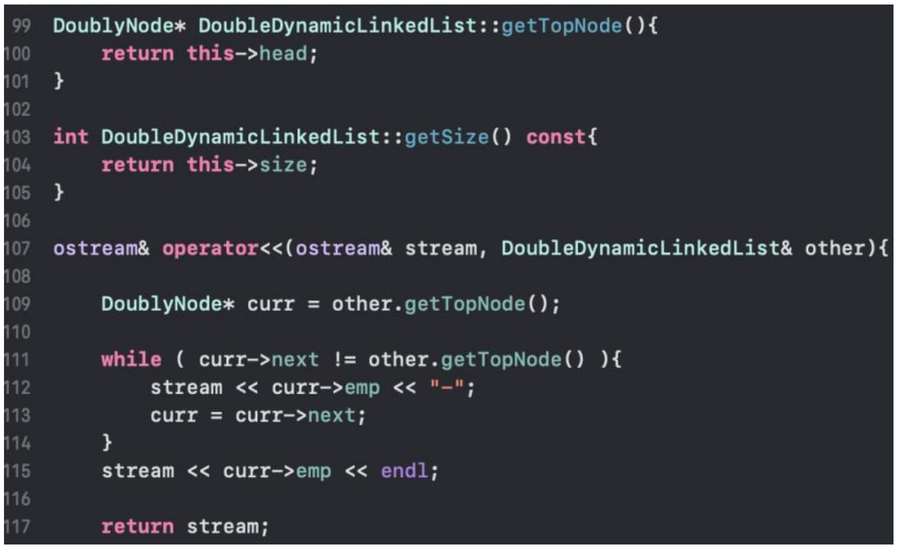
If it's not the head of the list, then we should traverse through the list. We will traverse through the list till next of curr is not null and we don't find the employee we want to delete.

There might be 2 situation that the while loop is terminated. One is curr.next might be NULL, in such case, there will be a message displayed to the terminal as such employee do not exist. In the second case we found the employee that we want to delete which is the next node of the curr. So we first assign that employee to temp and set curr.next as that elements next, and then, if that element's next is not null then we should also set its prev to curr. After all free that employee.



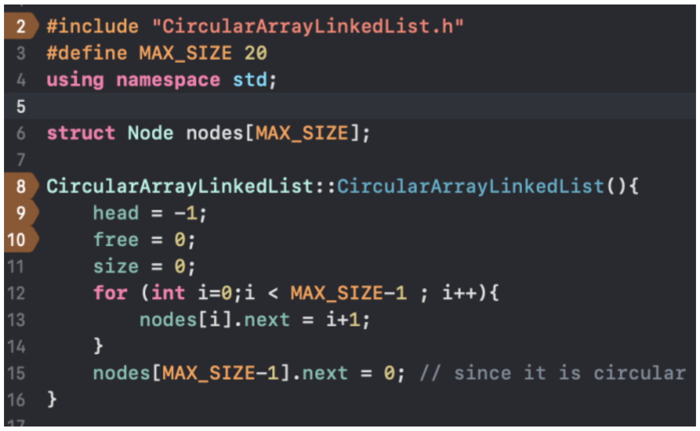
```
PermanentEmployee* DoubleDynamicLinkedList::getEmployee(int index){

    DoublyNode* curr = head;
    int count = 0;

    if (index >= this->getSize()){
        cout << "index out of bound exception";
        exit(1);
    }

    while(curr->next != NULL){
        if ( count == index ){
            break;
        }
        curr = curr->next;
        count++;
    }
    return curr->emp;

```

Get employee method just returns the employee at the index passed as parameter. It traverse through the list and whenever finds the employee at that index returns it.



```
DoublyNode* DoubleDynamicLinkedList::getTopNode(){
    return this->head;
}

int DoubleDynamicLinkedList::getSize() const{
    return this->size;
}

ostream& operator<<(ostream& stream, DoubleDynamicLinkedList& other){

    DoublyNode* curr = other.getTopNode();

    while ( curr->next != other.getTopNode() ){
        stream << curr->emp << "-";
        curr = curr->next;
    }
    stream << curr->emp << endl;

    return stream;
```

getTopNode method returns the head of the list. GetSize method returns the size of the list. And also we have an operator overloading there to print the content of the doubly linked list.



```
#include "CircularArrayLinkedList.h"
#define MAX_SIZE 20
using namespace std;

struct Node nodes[MAX_SIZE];

CircularArrayLinkedList::CircularArrayLinkedList(){
    head = -1;
    free = 0;
    size = 0;
    for (int i=0;i < MAX_SIZE-1 ; i++){
        nodes[i].next = i+1;
    }
    nodes[MAX_SIZE-1].next = 0; // since it is circular
}
```

In curcilar array linked list class, an array nodes is created that takes struct Node objects inside it with a size of MAX_SIZE = 20. In Circular array Linked list constructor, head is set as -1, free is 0 and size is 0 initially. And all the nodes inside the array have been linked to each other one by one. At the end, the last nodes.next is linked to the first index of the array.

In general, the way I implement the circular array linked list is actually a unique implementation since I cannot see it in anywhere but the exam paper of BBM201. First I create an array in node type and with the size of 20. This node structure consists of a data which is Temporary Employee pointer and an integer to keep track of the next node's index. Whenever I want to insert any new element to the list. I create a node and add it to the end of the array. It does not matter if its employee number is greater than or smaller than the last element of the array, because I will already compare them when linking them. So the element are added to the array in an unordered way, but when they're linked when deciding what it's next value to be. So whenever the array is printed it should be traversed through the head's next not in the order of them added to the array.

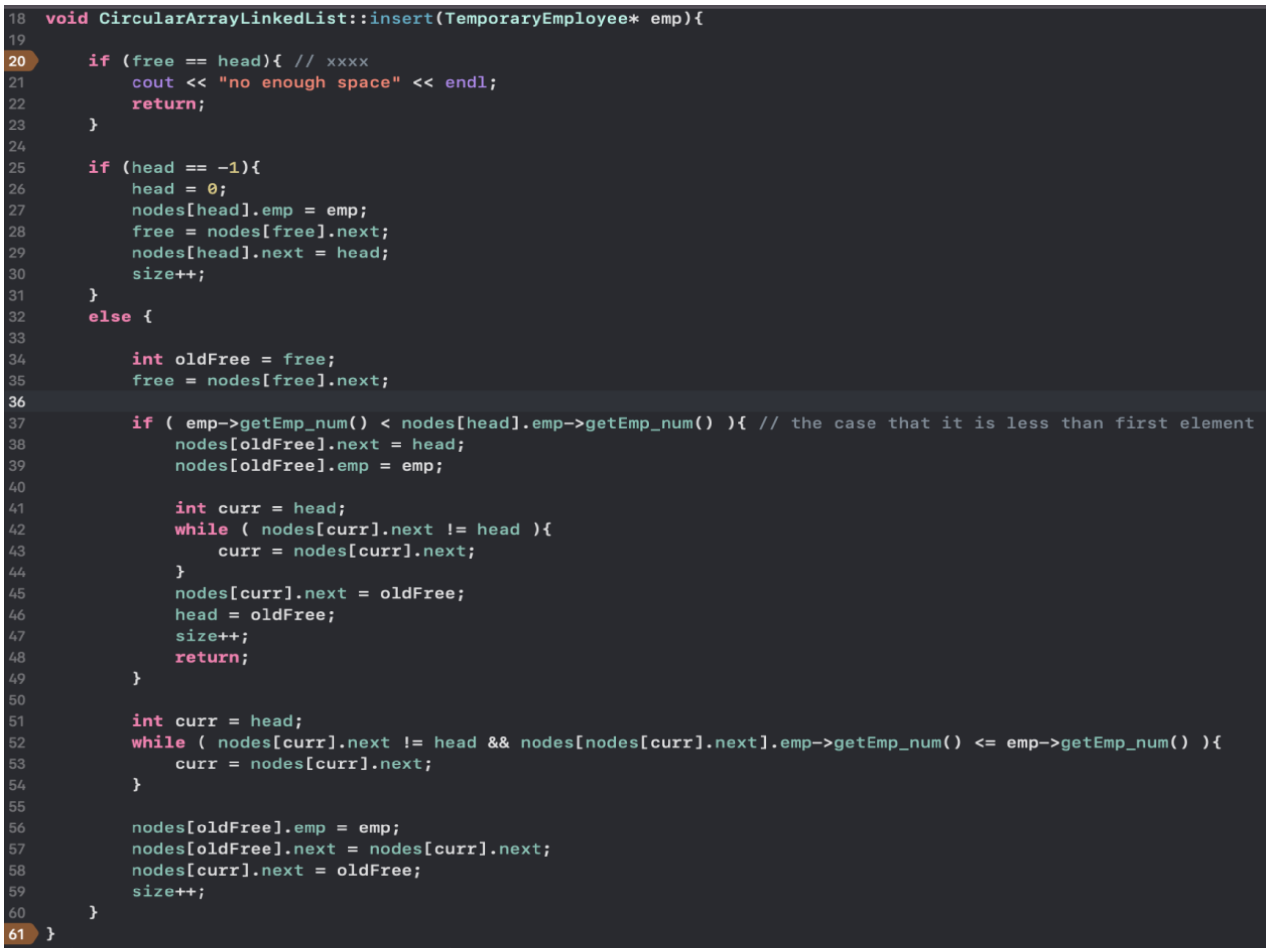
In insert function (the codes is below), we first check if free is equal to the head, if it is it means that the list is full.

If head is equal to -1, then we haven't set the head yet so that will be our first element to be added. set nodes[head] (which is nodes[0] in this case) data as the new employee. Set free to its next, and set head's next to head again since this list should be circular.

If there are already some employees in our list, we should first create and oldFree so that we can keep track of the old free index, after that, we can move free by one.

First check, if the new employee's emp_num is smaller than the head's employee number, if it is then, set nodes[oldFree].next as head and nodes[oldFree].emp = emp and go to the end of the list which is tail and set its next to head. After all set head as the new employee which is kept in index of oldFree in nodes array.

If the new employee will be added inside or the end of the list then, we should traverse through the list till finding a good spot (when traversing we'are setting curr = nodes[curr].next because it is sorted in this way not the actual array way so it would be ridiculous to traverse through the array from 0 to n because we didn't insert the elements in that order). After finding the proper spot, insert it.

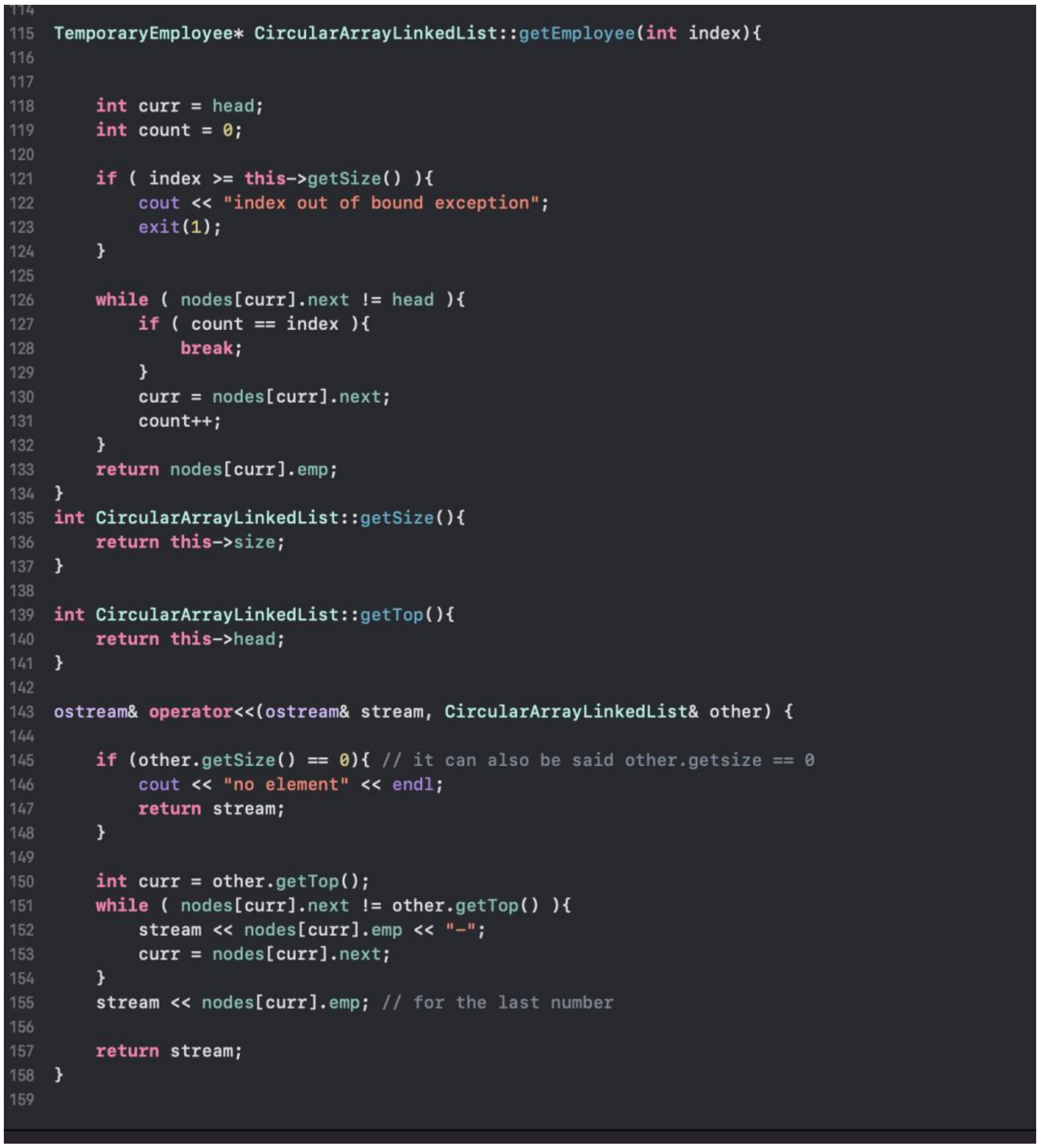


```
void CircularArrayLinkedList::insert(TemporaryEmployee* emp){

    if (free == head){ // xxxx
        cout << "no enough space" << endl;
        return;
    }

    if (head == -1){
        head = 0;
        nodes[head].emp = emp;
        free = nodes[free].next;
        nodes[head].next = head;
        size++;
    }
    else {

        int oldFree = free;
        free = nodes[free].next;

        if ( emp->getEmp_num() < nodes[head].emp->getEmp_num() ){ // the case that it is less than first element
            nodes[oldFree].next = head;
            nodes[oldFree].emp = emp;

            int curr = head;
            while ( nodes[curr].next != head ){
                curr = nodes[curr].next;
            }
            nodes[curr].next = oldFree;
            head = oldFree;
            size++;
            return;
        }

        int curr = head;
        while ( nodes[curr].next != head && nodes[nodes[curr].next].emp->getEmp_num() <= emp->getEmp_num() ){
            curr = nodes[curr].next;
        }

        nodes[oldFree].emp = emp;
        nodes[oldFree].next = nodes[curr].next;
        nodes[curr].next = oldFree;
        size++;
    }
}
```

In remove part (code is below), if head == -1 then there is no element to remove. If the employee that will be deleted is the head employee then we should first go to the tail and set tail's next to the head's next so that we can box off the actual head, and update the free index by setting it the deleted employee's index and set head.next as the new head. To understand that we're not looping infinitely we should check if the curr.next == head because if it is equal to the head and we do not stop the while loop, then it will infinitely loop.

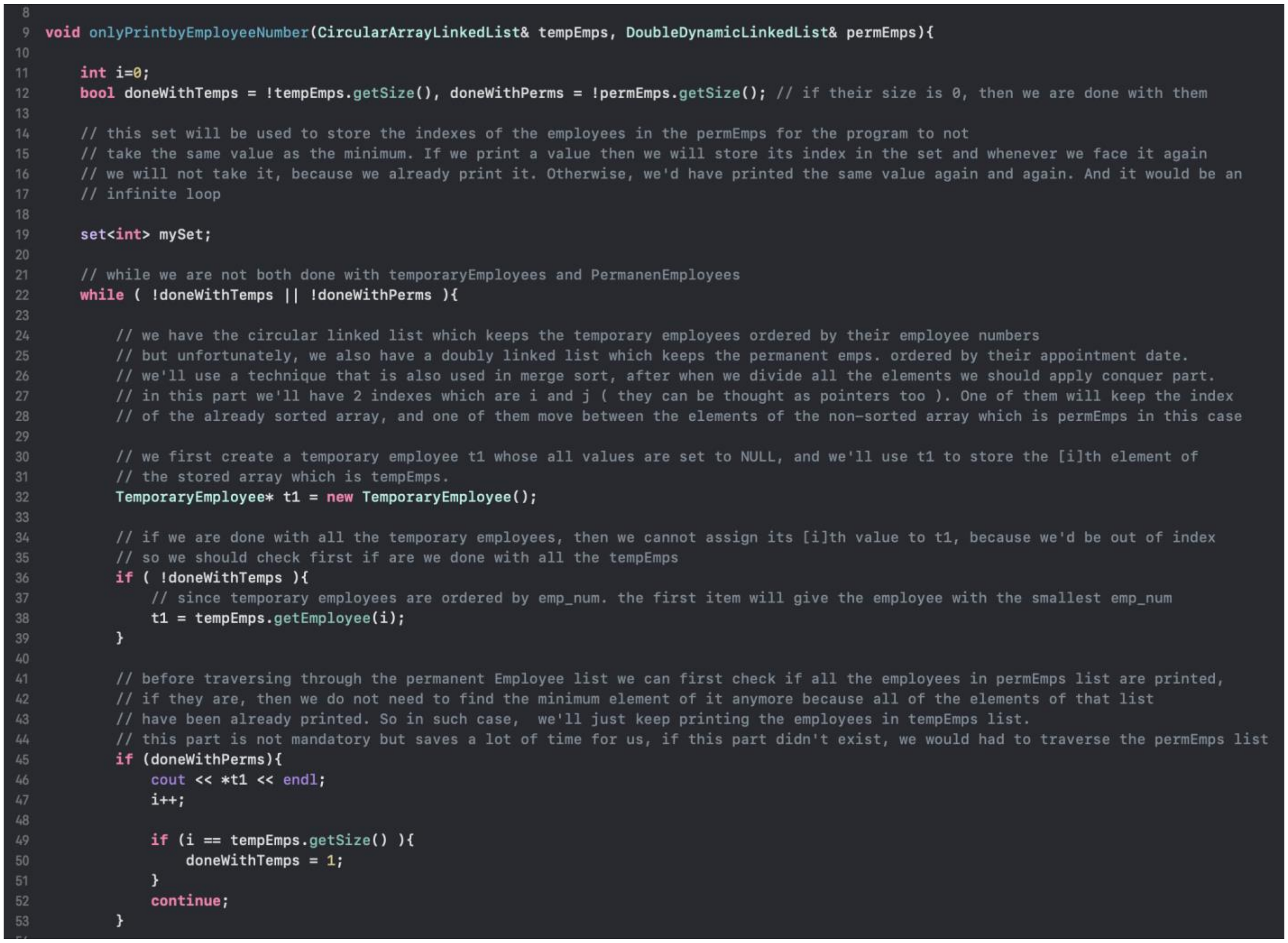
If we want to delete something from the inside of the list, then we should first find the employee that we want to delete in a while loop till the curr.next == the element that we want to delete assign it to a temporary value, update free index and that's it.

Additionally, we have the getEmployee, getSize, getTop methods but I didn't include it in the report to avoid complexity of the pictures. Their logic is the same as in the doubly linked list class. And also, we have an operator << to print the content of the circular linked list using array.



```cpp
TemporaryEmployee* CircularArrayLinkedList::getEmployee(int index){


    int curr = head;
    int count = 0;

    if ( index >= this->getSize() ){
        cout << "index out of bound exception";
        exit(1);
    }

    while ( nodes[curr].next != head ){
        if ( count == index ){
            break;
        }
        curr = nodes[curr].next;
        count++;
    }
    return nodes[curr].emp;
}
int CircularArrayLinkedList::getSize(){
    return this->size;
}

int CircularArrayLinkedList::getTop(){
    return this->head;
}

ostream& operator<<(ostream& stream, CircularArrayLinkedList& other) {

    if (other.getSize() == 0){ // it can also be said other.getsize == 0
        cout << "no element" << endl;
        return stream;
    }

    int curr = other.getTop();
    while ( nodes[curr].next != other.getTop() ){
        stream << nodes[curr].emp << "-";
        curr = nodes[curr].next;
    }
    stream << nodes[curr].emp; // for the last number

    return stream;
}
```

In main function there are exactly 7 functions which most of them are so similar to themselves except a little changes at the end. I will explain onlyPrintByEmployeeNumber function with pictures and the rest of them are inspired from them.



```cpp
void onlyPrintbyEmployeeNumber(CircularArrayLinkedList& tempEmps, DoubleDynamicLinkedList& permEmps){

    int i=0;
    bool doneWithTemps = !tempEmps.getSize(), doneWithPerms = !permEmps.getSize(); // if their size is 0, then we are done with them

    // this set will be used to store the indexes of the employees in the permEmps for the program to not
    // take the same value as the minimum. If we print a value then we will store its index in the set and whenever we face it again
    // we will not take it, because we already print it. Otherwise, we'd have printed the same value again and again. And it would be an
    // infinite loop

    set<int> mySet;

    // while we are not both done with temporaryEmployees and PermanenEmployees
    while ( !doneWithTemps || !doneWithPerms ){

        // we have the circular linked list which keeps the temporary employees ordered by their employee numbers
        // but unfortunately, we also have a doubly linked list which keeps the permanent emps. ordered by their appointment date.
        // we'll use a technique that is also used in merge sort, after when we divide all the elements we should apply conquer part.
        // in this part we'll have 2 indexes which are i and j ( they can be thought as pointers too ). One of them will keep the index
        // of the already sorted array, and one of them move between the elements of the non-sorted array which is permEmps in this case

        // we first create a temporary employee t1 whose all values are set to NULL, and we'll use t1 to store the [i]th element of
        // the stored array which is tempEmps.
        TemporaryEmployee* t1 = new TemporaryEmployee();

        // if we are done with all the temporary employees, then we cannot assign its [i]th value to t1, because we'd be out of index
        // so we should check first if are we done with all the tempEmps
        if ( !doneWithTemps ){
            // since temporary employees are ordered by emp_num. the first item will give the employee with the smallest emp_num
            t1 = tempEmps.getEmployee(i);
        }

        // before traversing through the permanent Employee list we can first check if all the employees in permEmps list are printed,
        // if they are, then we do not need to find the minimum element of it anymore because all of the elements of that list
        // have been already printed. So in such case,  we'll just keep printing the employees in tempEmps list.
        // this part is not mandatory but saves a lot of time for us, if this part didn't exist, we would had to traverse the permEmps list
        if (doneWithPerms){
            cout << *t1 << endl;
            i++;

            if (i == tempEmps.getSize() ){
                doneWithTemps = 1;
            }
            continue;
        }
```

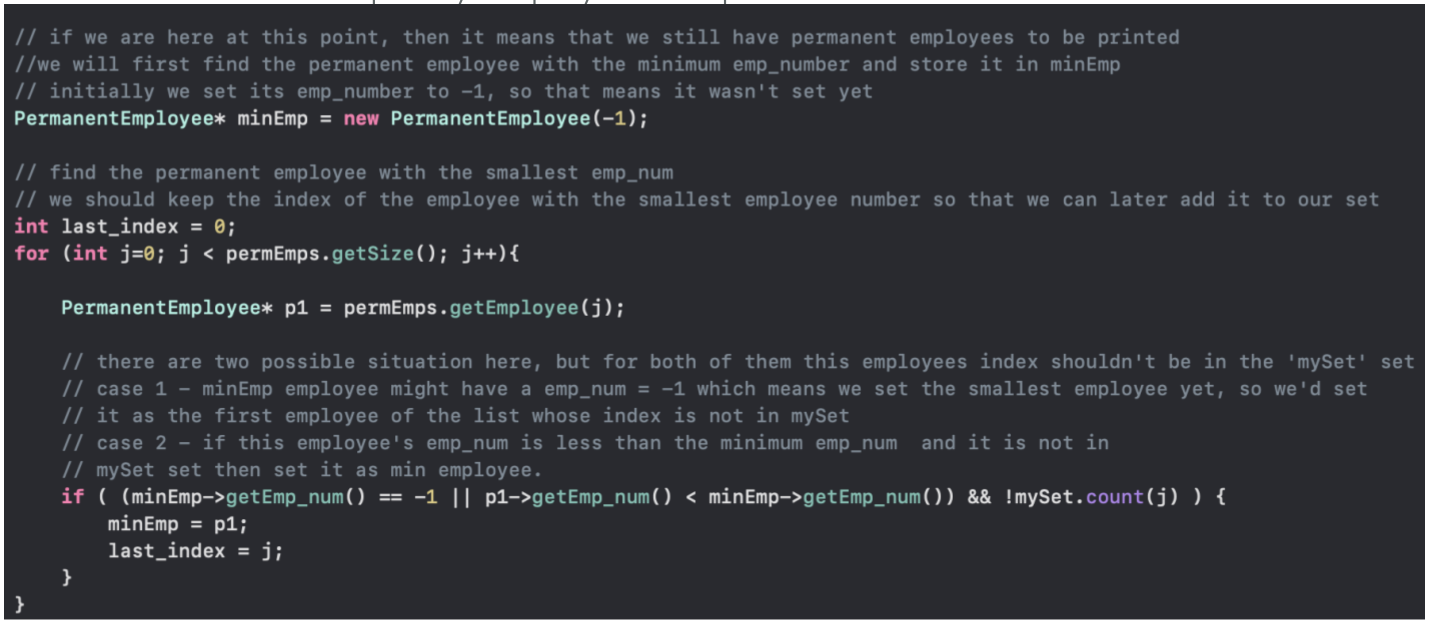
I've actually explained my code in comment lines, but will provide a more detailed version in this report. onlyPrintEmployeeNumber takes 2 parameters which one of them is CircularArrayLinkedList object that stores the temporary employees and DoubleDynamicLinkedList object that stores permanent employees. This function basically will print the employees in order of employee number in ascending order. Now, when we think we already have tempEmps list which is a Circular array linked list so it is already sorted by employee numbers in asceding order. So we do not have to worry about it, to find its employee with the smallest employee number, we just need to get its first element and compare the smallest element of the other list which is not sorted by employee number. So to find its employee with the smallest employee number, we should traverse through the list. And compare both list's smallest item and print the smaller one, and do nothing with the other one. After this is completed, do this till all the elements are printed. Yet, there is a problem right here. Whenever we print an element from tempEmps we can just increase its index i by 1 and we are good to go, yet, in permEmps list we should traverse through the list all the time and when we print one of them, we should do something to not print it again. And that solution comes with a set. Using set, we can store the indexes of permEmps so that we will not check it again and again. That was the general idea of how my algorithm works. If we are to dive into more detail:

We'll print all those elements in a while loop and for this while loop to break, all the employees should be checked. That's why we create 2 boolean value which are doneWithTemps and doneWithPerms that return true if we are done by checking all of the employees.

In while loop, we will create a temporary employee whose all values are set to NULL initially, and we'll use that employee to store the [i]th element of the stored array which is tempEmps.

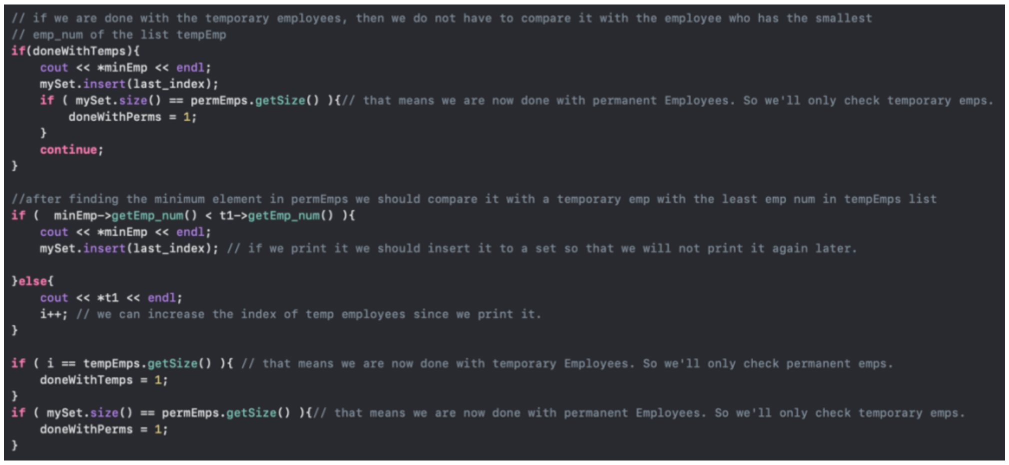
First, it should be checked if we are done with temps or not, if we are not done with temporary employees then we can assign the ith value to it. Otherwise, we'd get an index out of bound error.

Then we should check if we are done with permanent Employees, and we are done we do not need to find its smallest value, we can just print the temporary employees one by one. Inside this if block, after printing temporary employee, the I is increased by 1 and it is checked if all the temporary employees are printed.



```cpp
// if we are here at this point, then it means that we still have permanent employees to be printed
//we will first find the permanent employee with the minimum emp_number and store it in minEmp
// initially we set its emp_number to -1, so that means it wasn't set yet
PermanentEmployee* minEmp = new PermanentEmployee(-1);

// find the permanent employee with the smallest emp_num
// we should keep the index of the employee with the smallest employee number so that we can later add it to our set
int last_index = 0;
for (int j=0; j < permEmps.getSize(); j++){

    PermanentEmployee* p1 = permEmps.getEmployee(j);

    // there are two possible situation here, but for both of them this employees index shouldn't be in the 'mySet' set
    // case 1 - minEmp employee might have a emp_num = -1 which means we set the smallest employee yet, so we'd set
    // it as the first employee of the list whose index is not in mySet
    // case 2 - if this employee's emp_num is less than the minimum emp_num  and it is not in
    // mySet set then set it as min employee.
    if ( (minEmp->getEmp_num() == -1 || p1->getEmp_num() < minEmp->getEmp_num()) && !mySet.count(j) ) {
        minEmp = p1;
        last_index = j;
    }
}
```

Then a PermanentEmployee minEmp is created with an initial value of employee number is set to -1 which indicates that this employee is not set yet for my perspective. I use this simple technique in algorithm problems as well. First set min = -1 and then in an if block inside a for loop it goes into the for loop if min == -1 or a condition happens. We should also keep track of the index of the employee with the minimum employee number to store in 'mySet'.



```cpp
// if we are done with the temporary employees, then we do not have to compare it with the employee who has the smallest
// emp_num of the list tempEmp
if(doneWithTemps){
    cout << *minEmp << endl;
    mySet.insert(last_index);
    if ( mySet.size() == permEmps.getSize() ){// that means we are now done with permanent Employees. So we'll only check temporary emps.
        doneWithPerms = 1;
    }
    continue;
}

//after finding the minimum element in permEmps we should compare it with a temporary emp with the least emp num in tempEmps list
if (  minEmp->getEmp_num() < t1->getEmp_num() ){
    cout << *minEmp << endl;
    mySet.insert(last_index); // if we print it we should insert it to a set so that we will not print it again later.

}else{
    cout << *t1 << endl;
    i++; // we can increase the index of temp employees since we print it.
}

if ( i == tempEmps.getSize() ){ // that means we are now done with temporary Employees. So we'll only check permanent emps.
    doneWithTemps = 1;
}
if ( mySet.size() == permEmps.getSize() ){// that means we are now done with permanent Employees. So we'll only check temporary emps.
    doneWithPerms = 1;
}
```

At the end, after finding the permanent employee with the smallest employee number, we should check first if we are done with Temporary employees, if we are done we do not need to check the ith index of the temporary employees (again it would be an index out of bound exception would rise, of course in java, in this case it would be a segmentation fault). After printing the permanent employee we should add its index to 'mySet' to avoid endless loop.

So if we are not done both of them, we should then compare them and print the one with the smaller employee number.

After all, we should update the Boolean variables so that we can end the loop when needed.

For function 7, all the things are the same as function 6 except this time we should sort by appointment date. And we know that permanent employees are already sorted by appointment date from old to new. And temporary employees are not sorted by appointment date, so the spots should be changed. And also the comparison criteria should be appointment date instead of employee number. The rest is the same as the function 6.

For function 8, all the thing are the same as function 7 except that in the printing part we should this time check if the specific employee has been appointed before a certain date, if yes, then we can print it out. And also we should consider that the printing will be from new to old. So, instead of finding the employee with the older appointment date we should first find the employee with the newest appointment date. That's why we should go from end to the beginning for permanent employees and check for the employee with the biggest appointment date for the temporary employees.

For function 9, all the things are the same as function 7 except that in printing part we should check if the employee assigned to the institution in a given year.

For function 10, all the things are the same as function 6 except that in printing part we should check if the employee was born before a certain date.

For function 11, all the things are the same as function 6 except that in printing part we should check if the employee was born in a particular month.

For function 12, for temporary employees we should go from end to the beginning and check if the specific employee has the given title, if it is store its address in lastAssigned employee pointer and check for all the employees of permanent employee list. At the end print the employee with the highest appointment date.

5) Programmer Catalog: I've spend 1 day to decide how I will go through all the things. Firstly, I planned what will be implemented by reading the assignment sheet and did some research about it. In 2 days I implemented all the classes. I've done the commands part in only 1 day, and it took me 1 day to write this report. Yet, I should state that I study 10 hours a day. Other programmers also use can this code, but they should change it a bit. I also was planning to ake it more clear such us downgrading the number of functions from 7 to 2 but we were not allowed to use 3rd list so I couldn't do that. For other programmers who want to use my code should change its types if they want to store different objects inside those

circular array linked list and doubly linked list since I didn't make them generic. I explained all other things in the previous part of the report.

6) They can use my code since I designed it clear. Yet as I said, they should be changing the type of the variables that the data structures take in.

5) As a result, I want to mention about time complexity of the operations that is done in the main function. The time complexity of merge and sort 2 data structures would be not so efficient since I wanted to design a more clear code blocks. I defined a getEmployee method which takes the ith employee of the list. And in main file, I used a for loop that i goes starts from 0 to the length of the list. And for every i, it runs getEmployee method and in this method the ith element is returned. But the thing is, we could have done this more efficiently by only defining this part in main function. That could have been done easily, but it would look so complicated in the main function. And since the number of items that a list can take is so tenable, I used it that way.

6) References:

https://bilgisayarkavramlari.com/2007/05/03/linked-list-linkli-liste-veya-bagli-liste/
https://www.geeksforgeeks.org/circular-linked-list/
https://www.youtube.com/c/TheChernoProject