CS F211 Data Structures & Algorithms
I Semester 2021-2022
Lab Sheet 4

1.

a) Consider the following scenario:

Suppose an online video on demand service allows two types of customers: regular and premium. Both types of customers have the following attributes common:

- **Username**
- **Phone Number**
- **Watched shows**

Regular customers can only watch upto 32 hrs of content in a single language: English, while the premium customers can watch unlimited content in either English or their preferred language (hence an extra attribute).

Design the classes for the given scenario, with given member functions:

- **watch(Show show):** Checks whether the customer can watch the show. If yes, add it to the watched list and deduct from watched hours.
- **displayCustomer():** Display customer's details
- **displayShowIds():** Display watched shows

b) Assume that the customers can pool in and share an account, i.e., a single account can have multiple customers, but of same type (all regular or all premium). Further, an account can be shared by 5 regular customers and 10 premium customers. Create a template Account, that can hold customers of the required type. Create the functions to:

- add a customer to the account
- display the customer details

Note: Throw error or print error message wherever necessary.

```
#include<string.h>
#include<stdio.h>
#include<iostream>
using namespace std;

enum Languages{
	ENGLISH, FRENCH, GERMAN
};

struct Show{
	int id, duration, language;
	Show(){}
	Show(int sid, int sduration, int slanguage){
		id=sid;
		duration=sduration;
		language=slanguage;
	}
};

class Customer{
public:
	string userName, phoneNumber;
	int watchHours;
	Show watchedShows[1000];
	int index;
	Customer(){}
	Customer(string uName, string pNumber){
		userName = uName;
		phoneNumber = pNumber;
		watchHours=32;
		index=0;
	}

	virtual void watch(Show show){
		if(watchHours-show.duration >= 0 && show.language==ENGLISH){
			watchHours = watchHours - show.duration;
			watchedShows[index++] = show;
			cout<<userName<<" watched show id: "<<show.id<<endl;
		}
		else if(show.language!=ENGLISH){
			cout<<userName<<": Only English language is allowed for current subscription\n";
		}
		else{
			cout<<userName<<": Exceeds watch limit\n";
		}
	}

	virtual void displayCustomer(){
		cout<<"UserName: "<<userName<<"\nPhone: "<<phoneNumber<<"\n\n";
	}

	void displayWatchedShowIds(){
		for(int i=0;i<index;i++){
			cout<<watchedShows[i].id<<" ";
		}
		cout<<endl;
	}

};
```

```
class PremiumCustomer: public Customer{
public:
	int preferredLanguage;
	PremiumCustomer(): Customer(){}
	PremiumCustomer(string uName, string pName, int language):Customer(uName,
                                                                        pName){
		preferredLanguage= language;
		watchHours = -1;
	}

	void watch(Show show){
		/*update watchedShows and display appropriate messages, consult watch
function of Customer to get a better idea*/
	}

	void displayCustomer(){
		string lang;
		switch(preferredLanguage){
			case FRENCH:
				lang = "French";
				break;
			case GERMAN:
				lang = "German";
				break;
			default:
				lang = "English";
		}
		cout<<"UserName: "<<userName<<"\nPhone: "<<phoneNumber<<endl;
		cout<<"Preferred Language: "<<lang<<"\n\n";
	}
};

template<class T, int max>
class Account{
public:
	T customers[max];
	int maxCustomers;
	int index;

	Account(){
		maxCustomers=max;
		index=0;
	}

	void addCustomer(T cust){
		/*add the customer to the array customers*/
	}

	void displayCustomers(){
		/*diplay each customer's details*/
	}

};

int main(){
	Account<Customer, 5> acc1;
	acc1.addCustomer(Customer("User1", "985555"));
	acc1.addCustomer(Customer("User2", "986666"));
```

```
        acc1.displayCustomers();
        Account<PremiumCustomer, 10> acc2;
        acc2.addCustomer(PremiumCustomer("PUser1", "987777", GERMAN));
        acc2.addCustomer(PremiumCustomer("PUser2", "988888", FRENCH));
        acc2.displayCustomers();
        Show shows[] = {Show(1, 7, ENGLISH), Show(2, 7, FRENCH), Show(3, 10, GERMAN), Show(4, 17, ENGLISH)};
        acc1.customers[0].watch(shows[3]);
        acc1.customers[0].watch(shows[2]);
        for(int i=0; i<3; i++){
                acc2.customers[0].watch(shows[3]);
        }
        cout<<"Watched shows for "<<acc1.customers[0].userName<<":\n";
        acc1.customers[0].displayWatchedShowIds();
        cout<<"Watched shows for "<<acc2.customers[0].userName<<":\n";
        acc2.customers[0].displayWatchedShowIds();
}
```

---

2. Create a custom class for a doubly linked list with header and trailer sentinels that stores integer values at its data and allows the following operations:

- **push(x):** push a node with value x at the end of the list**,**
- **insertNode(pos, x):** insert a node at position pos with value x**,**
- **deleteNode(pos):** deletes the node at position pos**,**
- **print():** prints the entire list
- **goForward(x):** Shift the first node with value x forward by one place. So, if the list is 1<=>2<=>3<=>2<=>4 and x is 2, then the new list should be 1<=>3<=>2<=>2<=>4
- **goBackward(x):** Similar to goForward(x), but in the backward direction.

For goForward and goBackward, handle the appropriate edge cases by throwing error. If there are multiple x, shift only the first x. Also note to shift the node itself and not just the values.

```
#include <iostream>
using namespace std;

struct Node{
   int data;
   struct Node *next, *prev;
   Node(int val){
      data=val;
      next=NULL;
```

```
        prev=NULL;
    };
};

class DoublyLinkedList
{
private:
    struct Node* head, *tail;
public:
    DoublyLinkedList()
    {
        head = NULL;
        tail = NULL;
    }

    void push(int x)
    {
        struct Node *newNode=new Node(x);
        if(head==NULL)
        {
            newNode->prev=NULL;
            newNode->next=NULL;
            head=newNode;
            tail=newNode;
            return;
        }
        newNode->prev = tail;
        newNode->next = NULL;
        tail->next = newNode;
        tail = newNode;
    }

    void insertNode(int position, int x)
    {
        struct Node *newNode=new Node(x);
        if(!newNode)
        {
            throw runtime_error("Memory limit exceeded");
        }
        if(position==1)
        {
            newNode->prev=NULL;
            newNode->next=head;
            if(head)
                head->prev=newNode;
            head=newNode;
        }
        else
        {
            struct Node *temp=head;
            int k=1;
            while(k<(position-1) && (temp->next)!=NULL) //IMPORTANT!!!!
            {
                k++;
                temp=temp->next;
            }
            if(k!=(position-1)) //IMPORTANT!!!!
            {
                throw runtime_error("Desired position does not exist");
```

```
        }
        if(temp->next == NULL)
        {
            tail = newNode;
        }
        newNode->next=temp->next;
        newNode->prev=temp;
        if(newNode->next)
            newNode->next->prev=newNode;
        temp->next=newNode;
    }
}

void deleteNode(int position)
{
    if(head==NULL)
    {
        throw runtime_error("List is empty");
    }
    if(position==1)
    {
        struct Node *temp=head;
        head=head->next;
        if(head->next)
            head->prev=NULL;
        delete(temp);
        return;
    }
    else
    {
        struct Node *temp=head;
        int k=1;
        while(k<(position) && temp!=NULL)
        {
            k++;
            temp=temp->next;
        }
        if(temp==NULL)
        {
            throw runtime_error("Desired position does not exist");
        }
        if(temp->next==NULL)
        {
            tail = temp->prev;
        }
        struct Node *temp2=temp->prev;
        temp2->next=temp->next;
        if(temp->next)
            temp->next->prev=temp2;
        free(temp);
    }
}

void goForward(int element)
{

//Write blocks of code here for forward operation.
//For doing this you need to traverse the list to find if the element is present in the list and starting from that point you go forward one node
```

```
    }

    void goBackward(int element)
    {
       //write blocks of code here for backward operation.
       //same as the forward operation but here you will have to write code for backward operation.
    }

    void print()
    {
      struct Node *p=head;
      while(p)
      {
         cout<<p->data<<" ";
         p=p->next;
      }
      cout<<"\n";
    }
};

int main()
{
    DoublyLinkedList myList;
    for(int i=0;i<10;i++){
       myList.push(i+1);
    }
    myList.print();
    cout<<"Inserting 11 at pos 99\n";
    myList.insertNode(99,11);
    myList.print();
    cout<<"Insering 99 at pos 12\n";
    myList.insertNode(12,99);
    myList.print();
    cout<<"Insering 56 at start\n";
    myList.insertNode(1,56);
    myList.print();
    cout<<"Deleting last element\n";
    myList.deleteNode(13);
    myList.print();
    cout<<"Deleting first element\n";
    myList.deleteNode(1);
    myList.print();
    cout<<"Deleting position 4\n";
    myList.deleteNode(4);
    myList.print();
}
```

---

3. Write a C++ program for the inserting the content of given file into a linked list. Each node of the linked list has the following attributes:

- **a) str : a character array that stores the word. You may convert the all the words to lowercase**
- **b) count: an integer that stores the number of times str appears in the document**
- **c) next: a node pointer, pointing to the node corresponding to the next word in the document**

Input (passed as command line arguments) : name of the input file which has to be read, and name of the output file into which the linked list will be printed.

Sample Input: input.txt output.txt

Content of input.txt:
This is a file. The file contains the code. The code is written in C++

Sample Output: The content of output.txt should look like:

this 1
is 2
a 1
file 2
the 3
file 2
contains 1
the 3
code 2
the 3
code 2
is 2
written 1
in 1
c++ 1

```
#include <stdio.h>
#include <iostream>
#include <stdlib.h>
#include <string.h>
#include <ctype.h>

using namespace std;

struct Node
{
   char *str;
   int count;
   struct Node *next;
   Node(char* word, int freq)
   {
      str = (char*)malloc(strlen(word)+1);
      strcpy(str, word);
      count = freq;
      next = NULL;
   }
};

class WordList
{
private:
   struct Node *head, *tail;
   int updateFrequencies(char* word)
   {
      /*this function should update the frequencies
               of all the nodes having the current word,
        and return the total number of times the word had
        appeared so far
      */
   }
public:
   WordList(){
      head = NULL;
      tail = NULL;
   }

   //the function returns 1 if the node was added successfully, else 0
   int pushWord(char* word)
   {
      int freq = updateFrequencies(word);
      struct Node* newNode = new Node(word, freq+1);
      if(newNode == NULL)
      {
         return 0;
      }
      //base case when the list is empty
      if(tail==NULL)
      {
         head = newNode;
         tail = newNode;
      }
      else
      {
         tail->next = newNode;
         tail = tail->next;
```

```
    }
    return 1;
  }

  void printToFile(char* filename)
  {
    FILE *myOutput = fopen (filename, "w+");
    if (myOutput == 0)
      printf ("output file not opened \n");
    struct Node* ptr = head;

    while (ptr != NULL)
    {
      fprintf (myOutput, "%s %d \n", ptr->str, ptr->count);
      ptr = ptr->next;
    }
  }
};

int main (int argc, char *argv[])
{

  WordList myList;
  char *delim = ". ,:;\t\n\0";
  FILE *myFile;
  FILE *myOutput;
  char *filename = argv[1];
  char *outputfile = argv[2];
  if (argc != 3)
  {
    fprintf (stderr, "error: insufficient input”);
    return 1;
  }
  myFile = fopen (filename, "r+");
  if (myFile == 0)
  {
    printf ("input file not opened\n");
    return 1;
  }

  /*start reading file character by character when word has been detected;    push the word to linked list */

  char ch = 0; int word = 1, k = 0;
  char thisword[100];
  while ((ch = fgetc(myFile)) != EOF)
  {
    if (strchr (delim, ch))
    {
      /* insert the code here to push the word into the list and handle              appropriate edge cases */
    }
    else
    {  /* if not delim, add char to string, set word 1 */
      word = 1;
      thisword[k++] = tolower (ch);
    }
  }

  /* handle non-POSIX line-end */
  if (word == 1 && k>0)
```

```
    {
        thisword[k] = '\0';
        int status = myList.pushWord(thisword);
        if(status!=1)
        {
            fprintf (stderr, "error: adding the word failed.\n");
        }
    }

    myList.printToFile(outputfile);
    return 0;
}
```
