#ifndef \_\_DLIST\_H\_\_

#define \_\_DLIST\_H\_\_

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\* Do not modify the class declarations!

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

class emptyList {

};

template <typename T>

class Dlist {

// OVERVIEW: contains a double-ended list of Objects

public:

// Operational methods

bool IsEmpty() const; //1 working

// EFFECTS: returns true if list is empty, false otherwise

void InsertFront(const T &o); //2 working

// MODIFIES this

// EFFECTS inserts o at the front of the list

void InsertBack(const T &o); //3

// MODIFIES this

// EFFECTS inserts o at the back of the list

T RemoveFront(); //4

// MODIFIES this

// EFFECTS removes and returns first object from non-empty list

// throws an instance of emptyList if empty

T RemoveBack(); //5

// MODIFIES this

// EFFECTS removes and returns last object from non-empty list

// throws an instance of emptyList if empty

// Maintenance methods

Dlist(); // 6 ctor working

Dlist(const Dlist &l); // 7 copy ctor

Dlist &operator=(const Dlist &l); // 8 assignment

~Dlist(); // 9 dtor working

private:

// A private type

struct node {

node \*next;

node \*prev;

T o;

};

node \*first; // The pointer to the first node (0 if none)

node \*last; // The pointer to the last node (0 if none)

//How us this different than the doubly linked list in class?

// Utility methods

//10

void MakeEmpty();

// EFFECT: called by constructors to establish empty

// list invariant

void RemoveAll(); //11

// EFFECT: called by destructor and operator= to remove and destroy

// all list nodes

void CopyAll(const Dlist &l); //12

// EFFECT: called by copy constructor/operator= to copy nodes

// from a source instance l to this instance

};

/\*\*\*\* DO NOT MODIFY ABOVE THIS LINE \*\*\*\*\*/

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\*ADD Member function implementations here

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

// START FROM HERE

/\*

PSEUDOCODE

//1 method IsEmpty

if first is not nullptr

return 1

else

return 0

//2 method InsertFront (reference of a node called o)

create new\_node pointer to a new node

set o of new\_node = o

if first is nullptr

set first = new\_node

set last = new\_node

set next of new\_node = nullptr

set prev of new\_node = nullptr

else

first->prev = new\_node;

new\_node->next = first;

new\_node->prev = nullptr;

first = new\_node;

//3 method InsertBack(reference of a node called o)

create a new\_pointer to a new node;

set o of the node of new\_node pointer = o

if the list is not empty

set first = new\_node;

set last = new\_node;

set new\_node->next = nullptr

set new\_node->prev = nullptr

else

set last->next = new\_node

set new\_node->prev = last

set new\_node->next = nullptr

set last = new\_node

\*/

//1

template <class T>

bool Dlist<T>::IsEmpty() const{

return !first?1:0;

}

//2

template <class T>

void Dlist<T>::InsertFront(const T &o){

node\* new\_node = new node;

new\_node->o = o;

if (!first) {

first = new\_node;

last = new\_node;

new\_node->next = nullptr;

new\_node->prev = nullptr;

} else {

first->prev = new\_node;

new\_node->next = first;

new\_node->prev = nullptr;

first = new\_node;

}

}

//3

template <typename T>

void Dlist<T>::InsertBack(const T &o) {

node\* new\_node = new node;

new\_node->o = o;

if (!first) {

first = new\_node;

last = new\_node;

new\_node->next = nullptr;

new\_node->prev = nullptr;

} else {

last->next = new\_node;

new\_node->prev = last;

new\_node->next = nullptr;

last = new\_node;

}

}

//4

template <typename T>

T Dlist<T>::RemoveFront() {

if (IsEmpty()) {

emptyList empt;

throw empt;

}

node \*old\_node = first;

if (first == last) {

first = nullptr;

last = nullptr;

} else {

first = first->next;

first->prev = nullptr;

}

T o = old\_node->o;

delete old\_node;

return o;

}

//5

template <typename T>

T Dlist<T>::RemoveBack() {

if (IsEmpty()) {

emptyList empt;

throw empt;

} else {

node \*last\_node = last;

if (last == first) {

last = nullptr;

first = nullptr;

} else {

last = last->prev;

last->next = nullptr;

}

T o = last\_node->o;

delete last\_node;

return o;

}

}

//6 done

template <class T>

Dlist<T>::Dlist() {

MakeEmpty();

}

//7 done

template <class T>

Dlist<T>::Dlist(const Dlist &l){

MakeEmpty();

CopyAll(l);

}

//8

template <class T>

Dlist<T> &Dlist<T>::operator=(const Dlist &l) {

if (this == &l) return \*this;

  RemoveAll();

  CopyAll(l);

  return \*this;

}

//9 done

template <class T>

Dlist<T>::~Dlist() {

RemoveAll();

}

//10

template <class T>

void Dlist<T>::MakeEmpty() {

first = nullptr;

last = nullptr;

}

//11

template <class T>

void Dlist<T>::RemoveAll() {

while (first) {

node \*toDelete = first;

first = first->next;

toDelete->next = nullptr;

delete toDelete;

}

}

//12

template <class T>

void Dlist<T>::CopyAll(const Dlist &l) {

if (l.first == nullptr) {

first = nullptr;

last = nullptr;

} else {

node\* curr = l.first;

while (curr) {

InsertBack(curr->o);

curr = curr->next;

}

}

}

#endif /\* \_\_DLIST\_H\_\_ \*/

#include <iostream>

#include <string>

#include <stdio.h>

#include "dlist.h"

double IsValidNumber(std::string input) {

int count\_dot = 0;

for (int i = 0; i<input.length();i++) {

if (input.at(i) <= '9' && input.at(i) >= '0' ) {

continue;

} else if (i > 0 && input.at(i) == '.' && (input.at(i+1) <= '9' && input.at(i+1) >= '0' ) && count\_dot == 0) {

count\_dot++;

} else {

return -1;

}

}

return stod(input);

}

template <class T>

int StackLength(Dlist<T> list) {

int count = 0;

try {

while (list.RemoveFront()) {

count++;

}

}

catch (emptyList) {

return count;

}

return count;

}

void GetInput (Dlist<std::string> &input\_string) {

std::string input;

getline(std::cin,input);

while (input != "\0") {

input\_string.InsertBack(input);

getline(std::cin,input);

}

}

void Calculate (Dlist<std::string> input\_string) {

try {

std::string input = input\_string.RemoveFront();

Dlist<double> my\_stack;

double a;

double b;

while (input != "\0") {

a = -1;

b = -1;

if (input == "+"||input == "-"||input == "\*"||input == "/"||input == "n"||input == "d"||input == "r"||input == "p"||input == "c"||input == "a") {

char opt = input.at(0);

switch (opt) {

case '+': {

try {

a = my\_stack.RemoveFront();

b = my\_stack.RemoveFront();

my\_stack.InsertFront(a+b);

}

catch (emptyList) {

if (a != -1) {

my\_stack.InsertFront(a);

}

std::cout << "Not enough operands\n";

}

break;

}

case '-': {

try {

a = my\_stack.RemoveFront();

b = my\_stack.RemoveFront();

my\_stack.InsertFront(b-a);

}

catch (emptyList) {

if (a != -1) {

my\_stack.InsertFront(a);

}

std::cout << "Not enough operands\n";

}

break;

}

case '\*': {

try {

a = my\_stack.RemoveFront();

b = my\_stack.RemoveFront();

my\_stack.InsertFront(a\*b);

break;

}

catch (emptyList) {

if (a != -1) {

my\_stack.InsertFront(a);

}

std::cout << "Not enough operands\n";

}

break;

}

case '/': {

try {

a = my\_stack.RemoveFront();

b = my\_stack.RemoveFront();

if (a == 0) {

std::cout<< "Divide by zero\n";

my\_stack.InsertFront(b);

my\_stack.InsertFront(a);

} else {

my\_stack.InsertFront(b/a);

}

}

catch (emptyList) {

if (a != -1) {

my\_stack.InsertFront(a);

}

std::cout << "Not enough operands\n";

}

break;

}

case 'n': {

try {

a = my\_stack.RemoveFront();

my\_stack.InsertFront((-1)\*a);

}

catch (emptyList) {

std::cout << "Not enough operands\n";

}

break;

}

case 'd': {

try {

a = my\_stack.RemoveFront();

my\_stack.InsertFront(a);

my\_stack.InsertFront(a);

}

catch (emptyList) {

std::cout << "Not enough operands\n";

}

break;

}

case 'r': {

try {

a = my\_stack.RemoveFront();

b = my\_stack.RemoveFront();

my\_stack.InsertFront(a);

my\_stack.InsertFront(b);

}

catch (emptyList) {

if (a != -1) {

my\_stack.InsertFront(a);

}

std::cout << "Not enough operands\n";

}

break;

}

case 'p': {

try {

a = my\_stack.RemoveFront();

my\_stack.InsertFront(a);

std::cout << a << std::endl;

}

catch (emptyList) {

std::cout << "Not enough operands\n";

}

break;

}

case 'c': {

my\_stack.~Dlist();

break;

}

case 'a': {

for (int i = 1; i <= StackLength(my\_stack); i++) {

a = my\_stack.RemoveFront();

my\_stack.InsertBack(a);

if (i == StackLength(my\_stack)) {

std::cout << a <<std::endl ;

} else {

std::cout << a <<" ";

}

}

break;

}

}

} else if (input == "q") {

return ;

} else if (IsValidNumber(input) != -1){

my\_stack.InsertFront(IsValidNumber(input));

} else if (IsValidNumber(input) == -1){

std::cout<< "Bad input\n";

}

input = input\_string.RemoveFront();

}

}

catch (emptyList) {

return;

}

}

int main() {

Dlist<std::string> input\_string;

GetInput(input\_string);

Calculate(input\_string);

return 0;

}

#include <iostream>

#include <string>

#include "dlist.h"

struct Caller {

int time\_stamp;

std::string name;

std::string status;

int duration;

};

// IMPLEMENT FUNCTION GETCALLERINFOR

void GetCallersInfor (int &total\_tick, int event\_number, Dlist<Caller\*> &caller\_list) {

for (int i = 0; i < event\_number; i++) {

Caller\* new\_caller = new Caller;

std::cin >> new\_caller->time\_stamp >> new\_caller->name >> new\_caller->status >> new\_caller->duration;

caller\_list.InsertBack(new\_caller);

//calculate total\_tick;

if (total\_tick == new\_caller->time\_stamp) {

total\_tick = total\_tick + new\_caller->duration;

} else if (total\_tick < new\_caller->time\_stamp){

total\_tick = new\_caller->time\_stamp + new\_caller->duration ;

} else {

total\_tick += new\_caller->duration;

}

}

}

// IMPLEMENT CallerAtCurrentTime

void CallerAtCurrentTime(int curr\_tick, Dlist<Caller\*> &caller\_list) {

try {

Caller\* caller = caller\_list.RemoveFront();

while (caller->time\_stamp == curr\_tick) {

if (caller->status == "none") {

std::cout<<"Call from "<<caller->name<<" a regular member\n";

} else {

std::cout<<"Call from "<<caller->name<<" a "<<caller->status<<" member\n";

}

caller = caller\_list.RemoveFront();

}

caller\_list.InsertFront(caller);

}

catch (emptyList) {

}

}

//

void AddCallerToAnwerList (Caller\* curr, Dlist<Caller\*> &answer\_list) {

Caller\* new\_caller = new Caller;

new\_caller->time\_stamp = curr->time\_stamp;

new\_caller->name = curr->name;

new\_caller->status = curr->status;

new\_caller->duration = curr->duration;

answer\_list.InsertBack(new\_caller);

}

// FUNCTION TO MAKE A LIST OF CALLER TO CALL BACK IN ORDER

void MakeAnswerList(int event\_number, Dlist<Caller\*> caller\_list, Dlist<Caller\*> &answer\_list) {

int ev = event\_number;

//PLATIUM MEMBERS

for ( int i = 0 ; i < event\_number; i++) {

Caller\* curr = caller\_list.RemoveFront();

if ( curr->status == "platium" ) {

AddCallerToAnwerList(curr,answer\_list);

ev--;

} else {

caller\_list.InsertBack(curr);

}

}

//GOLD MEMBERS

event\_number = ev;

for (int i = 0 ; i < event\_number; i++) {

Caller\* curr = caller\_list.RemoveFront();

if ( curr->status == "gold" ) {

AddCallerToAnwerList(curr,answer\_list);

ev--;

} else {

caller\_list.InsertBack(curr);

}

}

//SILVER MEMBERS

event\_number = ev;

for (int i = 0 ; i < event\_number; i++) {

Caller\* curr = caller\_list.RemoveFront();

if ( curr->status == "silver" ) {

AddCallerToAnwerList(curr,answer\_list);

ev--;

} else {

caller\_list.InsertBack(curr);

}

}

//REGULAR MEMBERS

event\_number = ev;

for (int i = 0 ; i < event\_number; i++) {

Caller\* curr = caller\_list.RemoveFront();

if ( curr->status == "none" ) {

AddCallerToAnwerList(curr,answer\_list);

ev--;

} else {

caller\_list.InsertBack(curr);

}

}

}

//FUNCTION TO INPUT CURRENT CALLS

void CurrAnswer(int curr\_tick, Dlist<Caller\*> &answer\_list) {

try {

Caller\* caller = answer\_list.RemoveFront();

if (caller->time\_stamp > curr\_tick) {

CurrAnswer(curr\_tick, answer\_list);

try {

Caller\* next\_caller = answer\_list.RemoveFront();

if (next\_caller->status == "stillcalling") {

answer\_list.InsertFront(caller);

answer\_list.InsertFront(next\_caller);

} else {

answer\_list.InsertFront(next\_caller);

answer\_list.InsertFront(caller);

}

}

catch (emptyList){

answer\_list.InsertFront(caller);

}

} else {

if ( caller->duration == 1 && caller->status != "stillcalling") {

std::cout << "Answering call from " << caller->name<<"\n";

delete caller;

} else if ( caller->duration == 1 && caller->status == "stillcalling" ) {

delete caller;

} else if (caller->duration > 1 && caller->status == "stillcalling") {

caller->duration = caller->duration -1;

answer\_list.InsertFront(caller);

} else {

std::cout << "Answering call from " << caller->name <<"\n";

caller->duration = caller->duration -1;

caller->status = "stillcalling";

answer\_list.InsertFront(caller);

}

}

}

catch (emptyList) {

}

}

//// IMPLEMENT RunSimulator

void RunSimulator (int rem\_tick,Dlist<Caller\*> &caller\_list,Dlist<Caller\*> &answer\_list) {

for (int i = 0; i <= rem\_tick; i++) {

std::cout<< "Starting tick #" <<i <<"\n";

CallerAtCurrentTime(i,caller\_list);

CurrAnswer(i,answer\_list);

}

}

int main() {

Dlist<Caller\*> call\_list;

Dlist<Caller\*> answer\_list;

int total\_tick = 0;

//get event information

int event\_number;

std::cin >> event\_number;

//TEST THE CALL SIMULATOR

GetCallersInfor(total\_tick,event\_number,call\_list);

MakeAnswerList(event\_number,call\_list, answer\_list);

RunSimulator(total\_tick, call\_list, answer\_list);

}