// stack1.cpp

// stacks using STL

#include <iostream>

#include <string>

#include <stack>

using namespace std;

int main()

{

stack<string> stringStack; // stack is LIFO

string myString="first string";

string myString2="second String";

cout << myString << endl;

cout << myString2 << endl;

stringStack.push(myString);

stringStack.push(myString2);

string outputString = stringStack.top();

stringStack.pop();

cout << "popped value is " << outputString << endl;

outputString = stringStack.top();

stringStack.pop();

cout << "popped value is " << outputString << endl;

return 0;

}

output

first string

second String

popped value is second String

popped value is first string

// stack2.cpp

**#ifndef** H\_StackType

**#define** H\_StackType

**template** <**class** **Type**>

**class** stackType

{

**public**:

**bool** **isEmptyStack**();

//Function returns true if the stack is empty;

//otherwise, it returns false.

**bool** **isFullStack**();

//Function returns true if the stack is full;

//otherwise, it returns false.

**void** **destroyStack**();

//Remove all elements from the stack

//Post: top = 0

**void** **push**(**const** **Type**& newItem);

//Add the newItem to the stack

//Post: stack is changed and the newItem

// is added to the top of stack

**void** **pop**(**Type**& poppedItem);

//Remove the top element of the stack

//Post: Stack is changed and the top element

// is removed from the stack. The top element

// of the stack is saved in poppedItem.

**stackType**(**int** stackSize = 100);

//constructor

//Create an array of size stackSize to hold the

// stack elements. The default stack size is 100

//Post: The variable list contains the base

// address of the array, top = 0 and

// maxStackSize = stackSize

**stackType**(**const** stackType<**Type**>& otherStack);

//copy constructor

**~stackType**();

//destructor

//Remove all elements from the stack

//Post: The array (list) holding the stack

// elements is deleted

**private**:

**int** maxStackSize; //variable to store the maximum stack size

**int** top; //variable to point to the top of the stack

**Type** \*list; //pointer to the array that holds

//the stack elements

};

**#include** <iostream>

**using** **namespace** std;

**template**<**class** **Type**>

**void** **stackType<Type>::destroyStack**()

{

top = 0;

}//end destroyStack

**template**<**class** **Type**>

**bool** **stackType<Type>::isEmptyStack**()

{

**return**(top == 0);

}//end isEmptyStack

**template**<**class** **Type**>

**bool** **stackType<Type>::isFullStack**()

{

**return**(top == maxStackSize);

} //end isFullStack

**template**<**class** **Type**>

**stackType<Type>::stackType**(**int** stackSize)

{

**if**(stackSize <= 0)

{

cout<<"Size of the array to hold the stack must "

<<"be positive."<<**endl**;

cout<<"Creating an array of size 100."<<**endl**;

maxStackSize = 100;

}

**else**

maxStackSize = stackSize; //set the stack size to

//the value specified by

//the parameter stackSize

top = 0; //set top to 0

list = **new** **Type**[maxStackSize]; //create the array to

//hold the stack elements

}//end constructor

**template**<**class** **Type**>

**stackType<Type>::~stackType**() //destructor

{

**delete** [] list; //deallocate memory occupied by the array

}//end destructor

**template**<**class** **Type**>

**void** **stackType<Type>::push**(**const** **Type**& newItem)

{

list[top] = newItem; //add newItem at the top of the stack

top++; // increment the top

}//end push

**template**<**class** **Type**>

**void** **stackType<Type>::pop**(**Type**& poppedItem)

{

top--; //decrement the top

poppedItem = list[top]; //copy the top element of

//the stack into poppedItem

cout << "Popped item is " << poppedItem << endl;

}//end pop

**#endif**

// another array implementation.cpp

//Program to test the various operations of a stack

**#include** <iostream>

**using** **namespace** std;

**int** **main**()

{

stackType<**int**> stack(50);

**int** poppedInt;

stack.push(23);

stack.push(45);

stack.push(38);

stack.pop(poppedInt);

stack.pop(poppedInt);

stack.pop(poppedInt);

stackType<**float**> floatStack; // floatStack is object of class Stack<float>

**float** poppedFloat;

floatStack.push(1111.1); // push 3 floats, pop 3 floats

floatStack.push(2222.2);

floatStack.push(3333.3);

floatStack.pop(poppedFloat);

floatStack.pop(poppedFloat);

floatStack.pop(poppedFloat);

stackType<**long**> longStack; // longStack is object of class Stack<long>

**long** poppedLong;

longStack.push(123123123L); // push 3 longs, pop 3 longs

longStack.push(234234234L);

longStack.push(345345345L);

longStack.pop(poppedLong);

longStack.pop(poppedLong);

longStack.pop(poppedLong);

}

Output

Popped item is 38

Popped item is 45

Popped item is 23

Popped item is 3333.3

Popped item is 2222.2

Popped item is 1111.1

Popped item is 345345345

Popped item is 234234234

Popped item is 123123123

// stack3.cpp

**#ifndef** H\_StackType

**#define** H\_StackType

**#include** <iostream>

**using** **namespace** std;

//Definition of the node

**template** <**class** **Type**>

**struct** nodeType

{

**Type** info;

nodeType<**Type**> \*link;

};

**template**<**class** **Type**>

**class** linkedStackType

{

**public**:

**const** linkedStackType<**Type**>& **operator=**

(**const** linkedStackType<**Type**>&);

//overload the assignment operator

**void** **initializeStack**();

//Initialize the stack to an empty state.

//Post condition: Stack elements are removed; top = NULL

**bool** **isEmptyStack**();

//Function returns true if the stack is empty;

//otherwise, it returns false

**bool** **isFullStack**();

//Function returns true if the stack is full;

//otherwise, it returns false

**void** **push**(**const** **Type**& newItem);

//Add the newItem to the stack.

//Pre condition: stack exists and is not full

//Post condition: stack is changed and the newItem

// is added to the top of stack. top points to

// the updated stack

**void** **pop**(**Type**& poppedElement);

//Remove the top element of the stack.

//Pre condition: Stack exists and is not empty

//Post condition: stack is changed and the top

// element is removed from the stack. The top

// element of the stack is saved in poppedElement

**void** **destroyStack**();

//Remove all elements of the stack, leaving the

//stack in an empty state.

//Post condition: top = NULL

**linkedStackType**();

//default constructor

//Post condition: top = NULL

**linkedStackType**(**const** linkedStackType<**Type**>& otherStack);

//copy constructor

**~linkedStackType**();

//destructor

//All elements of the stack are removed from the stack

**private**:

nodeType<**Type**> \*top; // pointer to the stack

};

**template**<**class** **Type**> //default constructor

**linkedStackType<Type>::linkedStackType**()

{

top = NULL;

}

**template**<**class** **Type**>

**void** **linkedStackType<Type>::destroyStack**()

{

nodeType<**Type**> \*temp; //pointer to delete the node

**while**(top != NULL) //while there are elements in the stack

{

temp = top; //set temp to point to the current node

top = top->link; //advance top to the next node

**delete** temp; //deallocate memory occupied by temp

}

}// end destroyStack

**template**<**class** **Type**>

**void** **linkedStackType<Type>:: initializeStack**()

{

destroyStack();

}

**template**<**class** **Type**>

**bool** **linkedStackType<Type>::isEmptyStack**()

{

**return**(top == NULL);

}

**template**<**class** **Type**>

**bool** **linkedStackType<Type>:: isFullStack**()

{

**return** 0;

}

**template**<**class** **Type**>

**void** **linkedStackType<Type>::push**(**const** **Type**& newElement)

{

nodeType<**Type**> \*newNode; //pointer to create the new node

newNode = **new** nodeType<**Type**>; //create the node

newNode->info = newElement; //store newElement in the node

newNode->link = top; //insert newNode before top

top = newNode; //set top to point to the top node

} //end push

**template**<**class** **Type**>

**void** **linkedStackType<Type>::pop**(**Type**& poppedElement)

{

nodeType<**Type**> \*temp; //pointer to deallocate memory

poppedElement = top->info; //copy the top element into

//poppedElement

cout << "Popped item is " << poppedElement << endl;

temp = top; //set temp to point to the top node

top = top->link; //advance top to the next node

**delete** temp; //delete the top node

}//end pop

**template**<**class** **Type**> //copy constructor

**linkedStackType<Type>::linkedStackType**(**const** linkedStackType<**Type**>& otherStack)

{

nodeType<**Type**> \*newNode, \*current, \*last;

**if**(otherStack.top == NULL)

top = NULL;

**else**

{

current = otherStack.top; //set current to point to the

//stack to be copied

//copy the top element of the stack

top = **new** nodeType<**Type**>; //create the node

top->info = current->info; //copy the info

top->link = NULL; //set the link field of the

//node to null

last = top; //set last to point to the node

current = current->link; //set current to point to the

//next node

//copy the remaining stack

**while**(current != NULL)

{

newNode = **new** nodeType<**Type**>;

newNode->info = current->info;

newNode->link = NULL;

last->link = newNode;

last = newNode;

current = current->link;

}//end while

}//end else

}//end copy constructor

**template**<**class** **Type**> //destructor

**linkedStackType<Type>::~linkedStackType**()

{

nodeType<**Type**> \*temp;

**while**(top != NULL) //while there are elements in the stack

{

temp = top; //set temp to point to the current node

top = top ->link; //advance first to the next node

**delete** temp; //deallocate the memory occupied by temp

}//end while

}//end destructor

**template**<**class** **Type**> //overloading the assignment operator

**const** linkedStackType<**Type**>& **linkedStackType<Type>::operator=**

(**const** linkedStackType<**Type**>& otherStack)

{

nodeType<**Type**> \*newNode, \*current, \*last;

**if**(**this** != &otherStack) //avoid self-copy

{

**if**(top != NULL) //if the stack is not empty, destroy it

destroyStack();

**if**(otherStack.top == NULL)

top = NULL;

**else**

{

current = otherStack.top; //set current to point to

//the stack to be copied

//copy the top element of otherStack

top = **new** nodeType<**Type**>; //create the node

top->info = current->info; //copy the info

top->link = NULL; //set the link field of the

//node to null

last = top; //make last point to the node

current = current->link; //make current point to

//the next node

//copy the remaining elements of the stack

**while**(current != NULL)

{

newNode = **new** nodeType<**Type**>;

newNode->info = current->info;

newNode->link = NULL;

last->link = newNode;

last = newNode;

current = current->link;

}//end while

}//end else

}//end if

**return** \***this**;

}//end operator=

**#endif**

//linkstack.cpp

//This program tests the various operations of a linked stack

**#include** <iostream>

**using** **namespace** std;

**void** **testCopy**(linkedStackType<**int**> OStack);

**int** **main**()

{

linkedStackType<**int**> stack;

linkedStackType<**int**> otherStack;

linkedStackType<**int**> newStack;

**int** num;

stack.push(34);

stack.push(43);

stack.push(27);

newStack = stack;

cout<<"After the assignment operator, newStack: "<<**endl**;

**while**(!newStack.isEmptyStack())

{

newStack.pop(num);

cout<<num<<**endl**;

}

otherStack = stack;

cout<<"Testing the copy constructor"<<**endl**;

testCopy(otherStack);

cout<<"After the copy constructor, otherStack: "<<**endl**;

**while**(!otherStack.isEmptyStack())

{

otherStack.pop(num);

cout<<num<<**endl**;

}

linkedStackType<**int**> intStack;

**int** poppedInt;

intStack.push(23);

intStack.push(45);

intStack.push(38);

intStack.pop(poppedInt);

intStack.pop(poppedInt);

intStack.pop(poppedInt);

linkedStackType<**float**> floatStack; // floatStack is object of class Stack<float>

**float** poppedFloat;

floatStack.push(1111.1); // push 3 floats, pop 3 floats

floatStack.push(2222.2);

floatStack.push(3333.3);

floatStack.pop(poppedFloat);

floatStack.pop(poppedFloat);

floatStack.pop(poppedFloat);

linkedStackType<**long**> longStack; // longStack is object of class Stack<long>

**long** poppedLong;

longStack.push(123123123L); // push 3 longs, pop 3 longs

longStack.push(234234234L);

longStack.push(345345345L);

longStack.pop(poppedLong);

longStack.pop(poppedLong);

longStack.pop(poppedLong);

**return** 0;

}

**void** **testCopy**(linkedStackType<**int**> OStack) //function to test the

// copy constructor

{

**int** num;

cout<<"Stack in the function testCopy:"<<**endl**;

**while**(!OStack.isEmptyStack())

{

OStack.pop(num);

cout<<num<<**endl**;

}

}

output

After the assignment operator, newStack:

Popped item is 27

27

Popped item is 43

43

Popped item is 34

34

Testing the copy constructor

Stack in the function testCopy:

Popped item is 27

27

Popped item is 43

43

Popped item is 34

34

After the copy constructor, otherStack:

Popped item is 27

27

Popped item is 43

43

Popped item is 34

34

Popped item is 38

Popped item is 45

Popped item is 23

Popped item is 3333.3

Popped item is 2222.2

Popped item is 1111.1

Popped item is 345345345

Popped item is 234234234

Popped item is 123123123

//5.12

vector<string> getHtmlTags() { // store tags in a vector

vector<string> tags; // vector of html tags

while (cin) { // read until end of file

string line;

getline(cin, line); // input a full line of text

int pos = 0; // current scan position

int ts = line.find("<", pos); // possible tag start

while (ts != string::npos) { // repeat until end of string

int te = line.find(">", ts+1); // scan for tag end

tags.push\_back(line.substr(ts, te-ts+1)); // append tag to the vector

pos = te + 1; // advance our position

ts = line.find("<", pos);

}

}

return tags; // return vector of tags

}

//5.13

bool isHtmlMatched(const vector<string>& tags) {

LinkedStack S; // stack for opening tags

typedef vector<string>::const\_iterator Iter;// iterator type // iterate through vector

for (Iter p = tags.begin(); p != tags.end(); ++p) {

if (p->at(1) != '/') // opening tag?

S.push(\*p); // push it on the stack

else { // else must be closing tag

if (S.empty()) return false; // nothing to match - failure

string open = S.top().substr(1); // opening tag excluding '<'

string close = p->substr(2); // closing tag excluding '</'

if (open.compare(close) != 0) return false; // fail to match

else S.pop(); // pop matched element

}

}

if (S.empty()) return true; // everything matched - good

else return false; // some unmatched - bad

}

//5.14

int main() { // main HTML tester

if (isHtmlMatched(getHtmlTags())) // get tags and test them

cout << "The input file is a matched HTML document." << endl;

else

cout << "The input file is not a matched HTML document." << endl;

}

//queue1.h

// queue::push/pop

**#include** <iostream> // std::cin, std::cout

**#include** <queue> // std::queue

**int** **main** ()

{

std::queue<**int**> myqueue;

**int** myint;

std::cout << "Please enter some integers (enter 0 to end):\n";

**do** {

std::cin >> myint;

myqueue.push (myint);

} **while** (myint);

std::cout << "myqueue contains: ";

**while** (!myqueue.empty())

{

std::cout << ' ' << myqueue.front();

myqueue.pop();

}

std::cout << '\n';

**return** 0;

}

Output

Please enter some integers (enter 0 to end):

4

7

8

3

0

myqueue contains: 4 7 8 3 0

//queue2.h

//queue2.h

**#ifndef** H\_QueueAsArray

**#define** H\_QueueAsArray

**#include** <iostream>

**using** **namespace** std;

**template**<**class** **Type**>

**class** queueType

{

**public**:

**const** queueType<**Type**>& **operator=**(**const** queueType<**Type**>&);

// overload the assignment operator

**void** **initializeQueue**();

**void** **destroyQueue**();

**int** **isEmptyQueue**();

**int** **isFullQueue**();

**void** **addQueue**(**Type** queueElement);

**void** **deQueue**(**Type**& deqElement);

**queueType**(**int** queueSize = 100);

**queueType**(**const** queueType<**Type**>& otherQueue);

// copy constructor

**~queueType**();

//destructor

**private**:

**int** maxQueueSize;

**int** count;

**int** front;

**int** rear;

**Type** \*list; //pointer to the array that holds the queue elements

};

**template**<**class** **Type**>

**void** **queueType<Type>::initializeQueue**()

{

front = 0;

rear = maxQueueSize - 1;

count = 0;

}

**template**<**class** **Type**>

**void** **queueType<Type>::destroyQueue**()

{

front = 0;

rear = maxQueueSize - 1;

count = 0;

}

**template**<**class** **Type**>

**int** **queueType<Type>::isEmptyQueue**()

{

**return**(count == 0);

}

**template**<**class** **Type**>

**int** **queueType<Type>::isFullQueue**()

{

**return**(count == maxQueueSize);

}

**template**<**class** **Type**>

**void** **queueType<Type>::addQueue**(**Type** newElement)

{

rear = (rear + 1) % maxQueueSize; // use mod operator to advance rear

//because array is circular

count++;

list[rear] = newElement;

}

**template**<**class** **Type**>

**void** **queueType<Type>::deQueue**(**Type**& deqElement)

{

deqElement = list[front];

count--;

front = (front + 1) % maxQueueSize; // use mod operator to advance // rear because the array is circular

}

**template**<**class** **Type**>

**queueType<Type>::queueType**(**int** queueSize) //constructor

{

**if**(queueSize <= 0)

{

cout<<"Size of the array to hold the queue must "

<<"be positive."<<**endl**;

cout<<"Creating an array of size 100."<<**endl**;

maxQueueSize = 100;

}

**else**

maxQueueSize = queueSize; //set maxQueueSize to queueSize

front = 0; //initialize front

rear = maxQueueSize - 1; //initiaize rear

count = 0;

list = **new** **Type**[maxQueueSize]; //create the array to

//hold queue elements

}

**template**<**class** **Type**>

**queueType<Type>::~queueType**() //destructor

{

**delete** [] list;

}

**template**<**class** **Type**>

**const** queueType<**Type**>& **queueType<Type>::operator=**

(**const** queueType<**Type**>& otherQueue)

{

cout<<"Write the definition of the function "

<<"to overload the assignment operator"<<**endl**;

}

**#endif**

//queue1.cpp

//Test Program Queue as Array

**#include** <iostream>

**using** **namespace** std;

**int** **main**()

{

queueType<**int**> queue;

**int** x, y;

queue.initializeQueue();

x = 4;

y = 5;

queue.addQueue(x);

queue.addQueue(y);

queue.deQueue(x);

queue.addQueue(x + 5);

queue.addQueue(16);

queue.addQueue(x);

queue.addQueue(y - 3);

cout<<"Queue Elements: ";

**while**(!queue.isEmptyQueue())

{

queue.deQueue(y);

cout<<" "<<y;

}

cout<<**endl**;

**return** 0;

}

output

Queue Elements: 5 9 16 4 2

//5.18

typedef string Elem; // queue element type

class LinkedQueue { // queue as doubly linked list

public:

LinkedQueue(); // constructor

int size() const; // number of items in the queue

bool empty() const; // is the queue empty?

const Elem& front() const throw(QueueEmpty); // the front element

void enqueue(const Elem& e); // enqueue element at rear

void dequeue() throw(QueueEmpty); // dequeue element at front

private: // member data

CircleList C; // circular list of elements

int n; // number of elements

};

//5.19

LinkedQueue::LinkedQueue() // constructor

: C(), n(0) { }

int LinkedQueue::size() const // number of items in the queue

{ return n; }

bool LinkedQueue::empty() const // is the queue empty?

{ return n == 0; }

// get the front element

const Elem& LinkedQueue::front() const throw(QueueEmpty) {

if (empty())

throw QueueEmpty("front of empty queue");

return C.front(); // list front is queue front

}

//5.20

void LinkedQueue::enqueue(const Elem& e) {

C.add(e); // insert after cursor

C.advance(); // ...and advance

n++;

}

// dequeue element at front

void LinkedQueue::dequeue() throw(QueueEmpty) {

if (empty())

throw QueueEmpty("dequeue of empty queue");

C.remove(); // remove from list front

n--;

}

//queue3.h

**#ifndef** H\_linkedQueue

**#define** H\_linkedQueue

**#include** <iostream>

**using** **namespace** std;

//Definition of the node

**template** <**class** **Type**>

**struct** nodeType

{

**Type** info;

nodeType<**Type**> \*link;

};

**template**<**class** **Type**>

**class** linkedQueueType

{

**public**:

**const** linkedQueueType<**Type**>& **operator=**

(**const** linkedQueueType<**Type**>&);

// overload the assignment operator

**bool** **isEmptyQueue**();

**bool** **isFullQueue**();

**void** **destroyQueue**();

**void** **initializeQueue**();

**void** **addQueue**(**const** **Type**& newElement);

**void** **deQueue**(**Type**& deqElement);

**linkedQueueType** (); //default constructor

**linkedQueueType**(**const** linkedQueueType<**Type**>& otherQueue);

//copy constructor

**~linkedQueueType**(); //destructor

**private**:

nodeType<**Type**> \*front; //pointer to the front of the queue

nodeType<**Type**> \*rear; //pointer to the rear of the queue

};

**template**<**class** **Type**>

**linkedQueueType<Type>::linkedQueueType**() //default constructor

{

front = NULL; // set front to null

rear = NULL; // set rear to null

}

**template**<**class** **Type**>

**bool** **linkedQueueType<Type>::isEmptyQueue**()

{

**return**(front == NULL);

}

**template**<**class** **Type**>

**bool** **linkedQueueType<Type>::isFullQueue**()

{

**return** **false**;

}

**template**<**class** **Type**>

**void** **linkedQueueType<Type>::destroyQueue**()

{

nodeType<**Type**> \*temp;

**while**(front != NULL) //while there are elements left in the queue

{

temp = front; // set temp to point to the current node

front = front ->link; // advance front to the next node

**delete** temp; // deallocate memory occupied by temp

}

rear = NULL; // set rear to null

}

**template**<**class** **Type**>

**void** **linkedQueueType<Type>::initializeQueue**()

{

destroyQueue();

}

**template**<**class** **Type**>

**void** **linkedQueueType<Type>::addQueue**(**const** **Type**& newElement)

{

nodeType<**Type**> \*newNode;

newNode = **new** nodeType<**Type**>; //create the node

newNode->info = newElement; //store the info

newNode->link = NULL; //initialize the link field to null

**if**(front == NULL) //if initially queue is empty

{

front = newNode;

rear = newNode;

}

**else** //add newNode at the end

{

rear->link = newNode;

rear = rear->link;

}

}//end addQueue

**template**<**class** **Type**>

**void** **linkedQueueType<Type>::deQueue**(**Type**& deqElement)

{

nodeType<**Type**> \*temp;

deqElement = front->info; //copy the info of the first element

temp = front; //make temp point to the first node

front = front->link; //advance front to the next node

**delete** temp; //delete the first node

**if**(front == NULL) //if after deletion the queue is empty

rear = NULL; //set rear to NULL

}//end deQueue

**template**<**class** **Type**>

**linkedQueueType<Type>::~linkedQueueType**() //destructor

{

nodeType<**Type**> \*temp;

**while**(front != NULL) //while there are elements left in the queue

{

temp = front; //set temp to point to the current node

front = front ->link; //advance first to the next node

**delete** temp; //deallocate memory occupied by temp

}

rear = NULL; // set rear to null

}

**template**<**class** **Type**>

**const** linkedQueueType<**Type**>& **linkedQueueType<Type>::operator=**

(**const** linkedQueueType<**Type**>& otherQueue)

{

//Write the definition of to overload the assignment operator

}

//copy constructor

**template**<**class** **Type**>

**linkedQueueType<Type>::linkedQueueType**(**const** linkedQueueType<**Type**>& otherQueue)

{

//Write the definition of the copy constructor

}//end copy constructor

**#endif**

//queue3.cpp

//Test Program linked queue

**#include** <iostream>

**#include** "queue3.h"

**using** **namespace** std;

**int** **main**()

{

linkedQueueType<**int**> queue;

**int** x, y;

queue.initializeQueue();

x = 4;

y = 5;

queue.addQueue(x);

queue.addQueue(y);

queue.deQueue(x);

queue.addQueue(x + 5);

queue.addQueue(16);

queue.addQueue(x);

queue.addQueue(y - 3);

cout<<"Queue Elements: ";

**while**(!queue.isEmptyQueue())

{

queue.deQueue(y);

cout<<" "<<y;

}

cout<<**endl**;

**return** 0;

}

output

Queue Elements: 5 9 16 4 2

//Deque1.cpp

// deque::front

**#include** <iostream>

**#include** <deque>

**int** **main** ()

{

std::deque<**int**> mydeque;

mydeque.push\_front(77);

mydeque.push\_back(20);

mydeque.front() -= mydeque.back();

std::cout << "mydeque.front() is now " << mydeque.front() << '\n';

std::cout << "mydeque.back() is now " << mydeque.back() << '\n';

**return** 0;

}

Output

mydeque.front() is now 57

mydeque.back() is now 20

// 5.21

typedef string Elem; // deque element type

class LinkedDeque { // deque as doubly linked list

public:

LinkedDeque(); // constructor

int size() const; // number of items in the deque

bool empty() const; // is the deque empty?

const Elem& front() const throw(DequeEmpty); // the first element

const Elem& back() const throw(DequeEmpty); // the last element

void insertFront(const Elem& e); // insert new first element

void insertBack(const Elem& e); // insert new last element

void removeFront() throw(DequeEmpty); // remove first element

void removeBack() throw(DequeEmpty); // remove last element

private: // member data

DLinkedList D; // linked list of elements

int n; // number of elements

};

// 5.22

// insert new first element

void LinkedDeque::insertFront(const Elem& e) {

D.addFront(e);

n++;

}

// insert new last element

void LinkedDeque::insertBack(const Elem& e) {

D.addBack(e);

n++;

}

// remove first element

void LinkedDeque::removeFront() throw(DequeEmpty) {

if (empty())

throw DequeEmpty("removeFront of empty deque");

D.removeFront();

n--;

}

// remove last element

void LinkedDeque::removeBack() throw(DequeEmpty) {

if (empty())

throw DequeEmpty("removeBack of empty deque");

D.removeBack();

n--;

}

//3.23

class DLinkedList { // doubly linked list

public:

DLinkedList(); // constructor

~DLinkedList(); // destructor

bool empty() const; // is list empty?

const Elem& front() const; // get front element

const Elem& back() const; // get back element

void addFront(const Elem& e); // add to front of list

void addBack(const Elem& e); // add to back of list

void removeFront(); // remove from front

void removeBack(); // remove from back

private: // local type definitions

DNode\* header; // list sentinels

DNode\* trailer;

protected: // local utilities

void add(DNode\* v, const Elem& e); // insert new node before v

void remove(DNode\* v); // remove node v

};

//3.24

DLinkedList::DLinkedList() { // constructor

header = new DNode; // create sentinels

trailer = new DNode;

header->next = trailer; // have them point to each other

trailer->prev = header;

}

DLinkedList::~DLinkedList() { // destructor

while (!empty()) removeFront(); // remove all but sentinels

delete header; // remove the sentinels

delete trailer;

}

//3.25

bool DLinkedList::empty() const // is list empty?

{ return (header->next == trailer); }

const Elem& DLinkedList::front() const // get front element

{ return header->next->elem; }

const Elem& DLinkedList::back() const // get back element

{ return trailer->prev->elem; }

//3.26

// insert new node before v

void DLinkedList::add(DNode\* v, const Elem& e) {

DNode\* u = new DNode; u->elem = e; // create a new node for e

u->next = v; // link u in between v

u->prev = v->prev; // ...and v->prev

v->prev->next = v->prev = u;

}

void DLinkedList::addFront(const Elem& e) // add to front of list

{ add(header->next, e); }

void DLinkedList::addBack(const Elem& e) // add to back of list

{ add(trailer, e); }

//3.27

void DLinkedList::remove(DNode\* v) { // remove node v

DNode\* u = v->prev; // predecessor

DNode\* w = v->next; // successor

u->next = w; // unlink v from list

w->prev = u;

delete v;

}

void DLinkedList::removeFront() // remove from font

{ remove(header->next); }

void DLinkedList::removeBack() // remove from back

{ remove(trailer->prev); }

//5.23

typedef string Elem; // element type

class DequeStack { // stack as a deque

public:

DequeStack(); // constructor

int size() const; // number of elements

bool empty() const; // is the stack empty?

const Elem& top() const throw(StackEmpty); // the top element

void push(const Elem& e); // push element onto stack

void pop() throw(StackEmpty); // pop the stack

private:

LinkedDeque D; // deque of elements

};

//5.24

DequeStack::DequeStack() // constructor

: D() { }

// number of elements

int DequeStack::size() const

{ return D.size(); }

// is the stack empty?

bool DequeStack::empty() const

{ return D.empty(); }

// the top element

const Elem& DequeStack::top() const throw(StackEmpty) {

if (empty())

throw StackEmpty("top of empty stack");

return D.front();

}

// push element onto stack

void DequeStack::push(const Elem& e)

{ D.insertFront(e); }

// pop the stack

void DequeStack::pop() throw(StackEmpty)

{

if (empty())

throw StackEmpty("pop of empty stack");

D.removeFront();

}