40/40

```
Printout for cs320-09

// File: QueueA/QueueA.hpp
// Olivia Lara

#ifndef QUEUEA_HPP_
#define QUEUEA_HPP_

#include "ArrayT.hpp"

// ========= QueueA =========
template<class T>
class QueueA {
private:
    ArrayT<T> _data;
    int _head, _tail;

public:
    QueueA(int cap);
    // Post: This queue is allocated with a capacity of cap
    // and initialized to be empty.

    T dequeue();
    // Pre: This queue is not empty.
    // Post: The head value in this queue is removed and returned.

    void enqueue(T const &val);
    // Pre: This queue is not full.
    // Post: val is stored at the tail of this queue.

    T const &headOf() const;
    // Pre: This queue is not empty.
    // Post: The head value from this queue is returned.

    bool isEmpty() const;
    // Post: true is returned if this queue is empty; otherwise, false is returned.

    bool isFull() const;
    // Post: true is returned if this queue is full; otherwise, false is returned.

    void toStream(ostream &os) const;
    // Post: All the items on this queue from tail to head are written to os.
};

// ========= Constructor =========
template<class T>
QueueA<T>::QueueA(int cap) :
    _data(cap + 1),
    _head(0),
    _tail(0) {
}

// ========= dequeue =========
template<class T>
T QueueA<T>::dequeue() {
    if (isEmpty()) {
        cerr << "This list is empty" << endl;
        throw -1;
    }
    T val = _data[_head];
    _head = (_head + 1) % _data.cap();
    return val;
}

// ========= enqueue =========
template<class T>
void QueueA<T>::enqueue(T const &val) {
    if (isFull()) {
        cerr << "The list is full" << endl;
        throw -1;
   }
    _data[_tail] = val;
    _tail = (_tail + 1) % _data.cap();


}

// ========= headOf =========
template<class T>
T const &QueueA<T>::headOf() const {
    if (isEmpty()) {
        cerr << "The list is empty" << endl;
        throw -1;
    }
    return _data[_head];
}

// ========= isEmpty =========
template<class T>
bool QueueA<T>::isEmpty() const {
    return _head == _tail;
}

// ========= isFull =========
template<class T>
bool QueueA<T>::isFull() const {
    return (_head + 1 % _data.cap() - 1 == _tail + 1 % _data.cap());
}

// ========= operator<< =========
template<class T>
ostream &operator<<(ostream &os, QueueA<T> const &rhs) {
    rhs.toStream(os);
    return os;
}

// ========= toStream =========
template<class T>
void QueueA<T>::toStream(ostream &os) const {
    for (int i = _head; i != _tail; i = ((i + 1) % _data.cap())) {
        os << _data[i] << " ";
    }
    os << endl;
}

#endif

// new page
// File: QueueL/QueueL.hpp
// Olivia Lara

#ifndef QUEUEL_HPP_
#define QUEUEL_HPP_

#include <iostream> // ostream.
#include "ListL.hpp"
using namespace std;

// ========= QueueL =========
template<class T>
class QueueL {
private:
    ListL<T> *_listL;

public:
    QueueL();
    // This queue is allocated and initialized to be empty.

    ~QueueL();
    // Post: This queue is deallocated.

    T dequeue();
    // Pre: This queue is not empty.
    // Post: The head value in this queue is removed and returned.

    void enqueue(T const &val);
    // Post: val is stored at the tail of this queue.

    T const &headOf() const;
    // Pre: This queue is not empty.
    // Post: The head value from this queue is returned.

    bool isEmpty() const;
    // Post: true is returned if this queue is empty; otherwise, false is returned.

    void toStream(ostream &os) const;
    // Post: All the items on this queue from tail to head are written to os.
};

// ========= Constructor =========
template<class T>
QueueL<T>::QueueL() {
    _listL = new ListL<T>();
}

// ========= Destructor =========
template<class T>
QueueL<T>::~QueueL() {
    delete _listL;
}

// ========= dequeue =========
template<class T>
T QueueL<T>::dequeue() {
    if (_listL->isEmpty()) {
        cerr << "Cannot dequeue from empty stack." << endl;
        throw -1;
    }
    return _listL->remFirst();
}

// ========= enqueue =========
template<class T>
void QueueL<T>::enqueue(T const &val) {
    _listL->append(val);
}

// ========= headOf =========
template<class T>
T const &QueueL<T>::headOf() const {
    if (_listL->isEmpty()) {
        cerr << "The list is empty" << endl;
        throw -1;
    }
    return _listL->first();
}

// ========= isEmpty =========
template<class T>
bool QueueL<T>::isEmpty() const {
    return _listL->isEmpty();
}

// ========= operator<< =========
template<class T>
ostream &operator<<(ostream &os, QueueL<T> const &rhs) {
    rhs.toStream(os);
    return os;
}

// ========= toStream =========
template<class T>
void QueueL<T>::toStream(ostream &os) const {
    _listL->toStream(os);
}

#endif

// new page
// File: StackA/StackA.hpp
// Olivia Lara


#ifndef STACKA_HPP_
#define STACKA_HPP_

#include "ArrayT.hpp"

// ========= StackA =========
template<class T>
class StackA {
private:
    ArrayT<T> _data;
    int _top;

public:
    StackA(int cap);
    // Post: This stack is allocated with a capacity of cap
    // and initialized to be empty.

    bool isEmpty() const;
    // Post: true is returned if this stack is empty; otherwise, false is returned.

    bool isFull() const;
    // Post: true is returned if this stack is full; otherwise, false is returned.

    T pop();
    // Pre: This stack is not empty.
    // Post: The top value in this stack is removed and returned.

    void push(T const &val);
    // Pre: This stack is not full.
    // Post: val is stored on top of this stack.

    T const &topOf() const;
    // Pre: This stack is not empty.
    // Post: The top value from this stack is returned.

    void toStream(ostream &os) const;
    // Post: All the items on this stack from top to bottom are written to os.
};

// ========= Constructor =========
template<class T>
StackA<T>::StackA(int cap):
    _data(cap),
    _top(-1) {
}

// ========= isEmpty =========
template<class T>
bool StackA<T>::isEmpty() const {
    return _top == -1;
}

// ========= isFull =========
template<class T>
bool StackA<T>::isFull() const {
    return (_top + 1) == _data.cap();
}

// ========= pop =========
template<class T>
T StackA<T>::pop() {
    if (isEmpty()) {
        cerr << "pop precondition violated: Cannot pop from an empty stack." << endl;
        throw -1;
    }
    return _data[_top--];
}

// ========= push =========
template<class T>
void StackA<T>::push(T const &val) {
    if (isFull()) {
        cerr << "the stack is full" << endl;
        throw -1;
    }
    _data[++_top] = val;
}

// ========= topOf =========
template<class T>
T const &StackA<T>::topOf() const {
    if (isEmpty()) {
        cerr << "topOf precondition violated: An empty stack has no top." << endl;
        throw -1;
    }
    return _data[_top];
}

// ========= operator<< =========
template<class T>
ostream &operator<<(ostream &os, StackA<T> const &rhs) {
    rhs.toStream(os);
    return os;
}

// ========= toStream =========
template<class T>
void StackA<T>::toStream(ostream &os) const {
    os << "(";
    for (int i = _top; i > 0; i--) {
        os << _data[i] << ", ";
    }
    if (_top == -1) {
        os << ")";
    }
    else {
        os << _data[0] << ")";
    }
}

#endif

// new page
// File: StackL/StackL.hpp
// Olivia Lara

#ifndef STACKL_HPP_
#define STACKL_HPP_

#include <iostream> // ostream.
#include "ListL.hpp"
using namespace std;

// ========= StackL =========

template<class T>
class StackL {
private:
    ListL<T>* _listL;

public:
    StackL();
    // This stack is initialized to be empty.

    ~StackL();
    // Post: This stack is deallocated.

    bool isEmpty() const;
    // Post: true is returned if this stack is empty; otherwise, false is returned.

    T pop();
    // Pre: This stack is not empty.
    // Post: The top value in this stack is removed and returned.

    void push(T const &val);
    // Post: val is stored on top of this stack.

    T const &topOf() const;
    // Pre: This stack is not empty.
    // Post: The top value from this stack is returned.

    void toStream(ostream &os) const;
    // Post: All the items on this stack from top to bottom are written to os.
};

// ========= Constructor =========

template<class T>
StackL<T>::StackL() {
    _listL = new ListL<T>();
}

// ========= Destructor =========

template<class T>
StackL<T>::~StackL() {
    delete _listL;
}

// ========= isEmpty =========

template<class T>
bool StackL<T>::isEmpty() const {
    return _listL->isEmpty();
}

// ========= pop =========

template<class T>
T StackL<T>::pop() {
    if (_listL->isEmpty()) {
        cerr << "Cannot pop from empty stack." << endl;
        throw -1;
    }
    return _listL->remFirst();
}

// ========= push =========

template<class T>
void StackL<T>::push(const T &val) {
    _listL->prepend(val);
}

// ========= topOf =========

template<class T>
T const &StackL<T>::topOf() const {
    if (_listL->isEmpty()) {
        cerr << "An empty stack has no top." << endl;
        throw -1;
    }
    return _listL->first();
}

// ========= operator<< =========

template<class T>
ostream &operator<<(ostream &os, StackL<T> const &rhs) {
    rhs.toStream(os);
    return os;
}

// ========= toStream =========

template<class T>
void StackL<T>::toStream(ostream &os) const {
    _listL->toStream(os);
}

#endif

// new page
```