**CHAPTER 3**

**Lists, Stacks, and Queues**

**3.1**

template <typename Object>

void printLots(list <Object> L, list<int> P)

{

typename list < int > ::const\_iterator pIter ;

typename list < Object >::const\_iterator lIter ;

int start = 0;

lIter = L.begin();

for (pIter=P.begin(); pIter != P.end() && lIter != L.end(); pIter++)

{

while (start < \*pIter && lIter != L.end())

{

start++;

lIter++;

}

if (lIter !=L.end())

cout<<\*lIter<<endl;

}

}

This code runs in time P.end()--, or largest number in P list.

**3.2 (a)** Here is the code for single linked lists:

// beforeP is the cell before the two adjacent cells that are to be

// swapped

// Error checks are omitted for clarity

void swapWithNext(Node \* beforep)

{

Node \*p , \*afterp;

p = before->next;

afterp = p->next; // both p and afterp assumed not NULL

p->next = afterp-> next;

beforep ->next = afterp;

afterp->next = p;

}

**(b)** Here is the code for doubly linked lists:

// p and afterp are cells to be switched. Error checks as before

{

Node \*beforep, \*afterp;

beforep = p->prev;

afterp = p->next;

p->next = afterp->next;

beforep->next = afterp;

afterp->next = p;

p->next->prev = p;

p->prev = afterp;

afterp->prev = beforep;

}

**3.3**

template <typename Iterator, typename Object>

Iterator find(Iterator start, Iterator end, const Object& x)

{

Iterator iter = start;

while ( iter != end && \*iter != x)

iter++;

return iter;

}

**3.4**

// Assumes both input lists are sorted

template <typename Object>

list<Object> intersection( const list<Object> & L1,

const list<Object> & L2)

{

list<Object> intersect;

typename list<Object>:: const\_iterator iterL1 = L1.begin();

typename list<Object>:: const\_iterator iterL2= L2.begin();

while(iterL1 != L1.end() && iterL2 != L2.end())

{

if (\*iterL1 == \*iterL2)

{

intersect.push\_back(\*iterL1);

iterL1++;

iterL2++;

}

else if (\*iterL1 < \*iterL2)

iterL1++;

else

iterL2++;

}

return intersect;

}

**3.5**

// Assumes both input lists are sorted

template <typename Object>

list<Object> listUnion( const list<Object> & L1,

const list<Object> & L2)

{

list<Object> result;

typename list<Object>:: const\_iterator iterL1 = L1.begin();

typename list<Object>:: const\_iterator iterL2= L2.begin();

while(iterL1 != L1.end() && iterL2 != L2.end())

{

if (\*iterL1 == \*iterL2)

{

result.push\_back(\*iterL1);

iterL1++;

iterL2++;

}

else if (\*iterL1 < \*iterL2)

{

result.push\_back(\*iterL1);

iterL1++;

}

else

{

result.push\_back(\*iterL2);

iterL2++;

}

}

return result;

}

**3.6** This is a standard programming project. The algorithm can be sped up by setting *Mt* =*M* mod *N*,

so that the hot potato never goes around the circle more than once. If *Mt >N/*2, the potato should

be passed in the reverse direction. This requires a doubly linked list. The worst case running time

is clearly *O(N* min*(M*, *N))*, although when the heuristics are used, and *M* and *N* are comparable,

the algorithm might be significantly faster. If *M* = 1, the algorithm is clearly linear.

#include <iostream>

#include <list>

using namespace std;

int main()

{

int i,j, n, m, mPrime, numLeft;

list <int > L;

list<int>::iterator iter;

//Initialization

cout<<"enter N (# of people) & M (# of passes before elimination):";

cin>>n>>m;

numLeft = n;

mPrime = m % n;

for (i =1 ; i <= n; i++)

L.push\_back(i);

iter = L.begin();

// Pass the potato

for (i = 0; i < n; i++)

{

mPrime = mPrime % numLeft;

if (mPrime <= numLeft/2) // pass forward

for (j = 0; j < mPrime; j++)

{

iter++;

if (iter == L.end())

iter = L.begin();

}

else // pass backward

for (j = 0; j < mPrime; j++)

{

if (iter == L.begin())

iter = --L.end();

else

iter--;

}

cout<<\*iter<<" ";

iter= L.erase(iter);

if (iter == L.end())

iter = L.begin();

}

cout<<endl;

return 0;

}

**3.7** *O*(*N*2). The trim method reduces the size of the array, requiring each add to resize it. The resize takes *O*(*N*) time, and there are *O*(*N*) calls.

// if out of bounds, writes a message (could throw an exception)

Object & operator[]( int index )

{

if (index >=0 && index <size() )

return objects[ index ];

else

cout<<"index out of bounds\n";

return objects[0];

}

const Object & operator[]( int index ) const

{

if (index >=0 && index <size() )

return objects[ index ];

else

cout<<"index out of bounds\n";

return objects[0];

}

**3.8**

iterator insert(iterator pos, const Object& x)

{

Object \* iter = &objects[0];

Object \*oldArray = objects;

theSize++;

int i;

if (theCapacity < theSize)

theCapactiy = theSize;

objects = new Object[ theCapacity ];

while(iter != pos)

{

objects[i]= oldArray[i];

iter += sizeOf(Object);

pos += sizeOf(Object);

i++;

}

objects[pos] = x;

for (int k = pos+1; k < theSize; k++)

objects[k] = oldArray[ k ];

delete [ ] oldArray;

return & objects[pos];

}

**3.9** All the aforementioned functions may require the creation of a new array to hold the data. When

this occurs all the old pointers (iterators) are invalid.

**3.10** The changes are the const\_iterator class, the iterator class and changes to all Vector functions that

use or return iterators. These classes and functions are shown in the following three

**(a)**

class const\_iterator

{

public:

//const\_iterator( ) : current( NULL )

// { } Force use of the safe constructor

const Object & operator\* ( ) const

{ return retrieve( ); }

const\_iterator & operator++ ( )

{

current++;

return \*this;

}

const\_iterator operator++ ( int )

{

const\_iterator old = \*this;

++( \*this );

return old;

}

bool operator== ( const const\_iterator & rhs ) const

{ return current == rhs.current; }

bool operator!= ( const const\_iterator & rhs ) const

{ return !( \*this == rhs ); }

protected:

Object \*current;

const Vector<Object> \*theVect;

Object & retrieve( ) const

{

assertIsValid();

return \*current;

}

const\_iterator( const Vector<Object> & vect, Object \*p ) :theVect (& vect), current( p ) { }

void assertIsValid() const

{

if (theVect == NULL || current == NULL )

throw IteratorOutOfBoundsException();

}

friend class Vector<Object>;

};

**(b)**

class iterator : public const\_iterator

{

public:

//iterator( )

// { } Force use of the safe constructor

Object & operator\* ( ) { return retrieve( ); }

const Object & operator\* ( ) const { return const\_iterator::operator\*( ); }

iterator & operator++ ( )

{

cout<<"old "<<\*current<<" ";

current++;

cout<<" new "<<\*current<<" ";

return \*this;

}

iterator operator++ ( int )

{

iterator old = \*this;

++( \*this );

return old;

}

protected:

iterator(const Vector<Object> & vect, Object \*p ) : const\_iterator(vect, p ) { }

friend class Vector<Object>;

};

**(c)**

iterator begin( )

{ return iterator(\*this ,&objects[ 0 ]); }

const\_iterator begin( ) const

{ return const\_iterator(\*this,&objects[ 0 ]); }

iterator end( )

{ return iterator(\*this, &objects[ size( ) ]); }

const\_iterator end( ) const

{ return const\_iterator(\*this, &objects[ size( ) ]); }

**3.11**

template <typename Object>

struct Node

{

Object data;

Node \* next;

Node ( const Object & d = Object(), Node \*n = NULL ) : data(d) , next(n) {}

};

template <typename Object>

class singleList

{

public:

singleList( ) { init(); }

~singleList() { eraseList(head);}

singleList( const singleList & rhs)

{

eraseList(head);

init();

\*this = rhs;

}

bool add(Object x)

{

if (contains(x))

return false;

else

{

Node<Object> \*ptr = new Node<Object>(x);

ptr->next = head->next;

head->next = ptr;

theSize++;

}

return true;

}

bool remove(Object x)

{

if (!contains(x))

return false;

else

{

Node<Object>\*ptr = head->next;

Node<Object>\*trailer;

while(ptr->data != x)

{ trailer = ptr;

ptr=ptr->next;

}

trailer->next = ptr->next;

delete ptr;

theSize--;

}

return true;

}

int size() { return theSize;}

void print()

{

Node<Object> \*ptr = head->next;

while (ptr != NULL)

{

cout<< ptr->data<<" ";

ptr = ptr->next;

}

cout<<endl;

}

bool contains(const Object & x)

{

Node<Object> \* ptr = head->next;

while (ptr != NULL)

{

if (x == ptr->data)

return true;

else

ptr = ptr-> next;

}

return false;

}

void init()

{

theSize = 0;

head = new Node<Object>;

head-> next = NULL;

}

void eraseList(Node<Object> \* h)

{

Node<Object> \*ptr= h;

Node<Object> \*nextPtr;

while(ptr != NULL)

{

nextPtr = ptr->next;

delete ptr;

ptr= nextPtr;

}

};

private:

Node<Object> \*head;

int theSize;

};

**3.12**

template <typename Object>

struct Node

{

Object data;

Node \* next;

Node ( const Object & d = Object(), Node \*n = NULL ) : data(d) , next(n) {}

};

template <typename Object>

class singleList

{

public:

singleList( ) { init(); }

~singleList(){ eraseList(head);}

singleList( const singleList & rhs)

{

eraseList(head);

init();

\*this = rhs;

}

bool add(Object x)

{

if (contains(x))

return false;

else

{

Node<Object> \*ptr = head->next;

Node<Object>\* trailer = head;

while(ptr && ptr->data < x)

{

trailer = ptr;

ptr = ptr->next;

}

trailer->next = new Node<Object> (x);

trailer->next->next = ptr;

theSize++;

}

return true;

}

bool remove(Object x)

{

if (!contains(x))

return false;

else

{

Node<Object>\*ptr = head->next;

Node<Object>\*trailer;

while(ptr->data != x)

{

trailer = ptr;

ptr=ptr->next;

}

trailer->next = ptr->next;

delete ptr;

theSize--;

}

return true;

}

int size() { return theSize;}

void print()

{

Node<Object> \*ptr = head->next;

while (ptr != NULL)

{

cout<< ptr->data<<" ";

ptr = ptr->next;

}

cout<<endl;

}

bool contains(const Object & x)

{

Node<Object> \* ptr = head->next;

while (ptr != NULL && ptr->data <= x )

{

if (x == ptr->data)

return true;

else

ptr = ptr-> next;

}

return false;

}

void init()

{

theSize = 0;

head = new Node<Object>;

head-> next = NULL;

}

void eraseList(Node<Object> \* h)

{

Node<Object> \*ptr= h;

Node<Object> \*nextPtr;

while(ptr != NULL)

{

nextPtr = ptr->next;

delete ptr;

ptr= nextPtr;

}

};

private:

Node<Object> \*head;

int theSize;

};

**3.13** Add the following code to the const\_iterator class. Add the same code with iterator replacing

const\_iterator to the iterator class.

const\_iterator & operator-- ( )

{

current = current->prev;

return \*this;

}

const\_iterator operator-- ( int )

{

const\_iterator old = \*this;

--( \*this );

return old;

}

**3.14**

const\_iterator & operator+ ( int k )

{

const\_iterator advanced = \*this;

for (int i = 0; i < k ; i++)

advanced.current = advanced.current->next;

return advanced;

}

**3.15**

void splice (iterator itr, List<Object> & lst)

{

itr.assertIsValid();

if (itr.theList != this)

throw IteratorMismatchException ();

Node \*p = iter.current;

theSize += lst.size();

p->prev->next = lst.head->next;

lst.head->next->prev = p->prev;

lst.tail->prev->next = p;

p->prev = lst->tail->prev;

lst.init();

}

**3.16**

The class const\_reverse\_iterator is almost identical to const\_iterator while reverse\_iterator is

almost identical to iterator. Redefine ++ to be − and vice versa for both the pre and post operators

for both classes as well as changing all variables of type const\_iterator to const\_reverse\_iterator

and changing iterator to reverse\_iterator. Add two new members in list for rbegin() and rend().

// In List add

const\_reverse\_iterator rbegin() const

{

return const\_reverse\_iterator itr( tail);

}

const\_reverse\_iterator rend() const

{

const\_reverse\_iterator itr(head);

}

reverse\_iterator rbegin()

{

return reverse\_iterator itr( tail);

}

reverse\_iterator rend()

{

reverse\_iterator itr(head);

}

**3.17** changed functions listed below

For the class const\_iterator

const\_iterator( ) : current( nullptr )

{ }

const Object & operator\* ( ) const

{

assertIsValid();

return retrieve( );

}

const\_iterator & operator++ ( )

{

assertIsValid();

current = current->next;

return \*this;

}

const\_iterator operator++ ( int )

{

assertIsValid();

const\_iterator old = \*this;

++( \*this );

return old;

}

protected:

Node \*current;

const List<Object> \*theList;

const\_iterator(const List<Object> &lst, Node \*p) : theList(&lst),current(p)

{}

void assertIsValid() const

{

if (theList == nullptr || current == nullptr || current == theList->head)

throw IteratorOutOfBoundsException();

}

For the class class iterator : public const\_iterator

Object & operator\* ( )

{

assertIsValid();

return const\_iterator::retrieve( );

}

const Object & operator\* ( ) const

{

assertIsValid();

return const\_iterator::operator\*( );

}

iterator & operator++ ( )

{

assertIsValid();

this->current = this->current->next;

return \*this;

}

iterator operator++ ( int )

{

assertIsValid();

iterator old = \*this;

++( \*this );

return old;

}

protected:

iterator(const List<Object> & lst, Node \*p ): const\_iterator(lst,p)

{ }

For the List class

iterator begin( )

{ iterator itr(\*this, head);

return (++itr );

}

const\_iterator begin( ) const

{ const\_iterator itr(\*this, head);

return ( ++itr );

}

iterator end( )

{ iterator itr(\*this, tail);

return ( itr );

}

const\_iterator end( ) const

{

iterator itr(\*this, tail);

return ( itr );

}

iterator insert( iterator itr, const Object & x )

{

itr.assertIsValid();

if (itr.theList != this)

throw IteratorMismatchException();

Node \*p = itr.current;

theSize++;

return (\*this, p->prev = p->prev->next = new Node( x, p->prev, p ) );

}

iterator insert( iterator itr, Object && x )

{

itr.assertIsValid();

if (iter.theList != this)

throw IteratorMismatchException();

Node \*p = itr.current;

theSize++;

return { p->prev = p->prev->next

= new Node{ std::move( x ), p->prev, p } };

}

iterator erase( iterator itr )

{

Node \*p = itr.current;

itr.assertIsValid();

if (itr.theList != this)

throw IteratorMismatchException();

iterator retVal(\*this, p->next );

p->prev->next = p->next;

p->next->prev = p->prev;

delete p;

theSize--;

return retVal;

}

iterator erase( iterator from, iterator to )

{

for( iterator itr = from; itr != to; )

{

itr.assertIsValid();

if (iter.theList != this)

throw IteratorMismatchException();

itr = erase( itr );

}

return to;

}

**3.18** Add a boolean data member to the node class that is true if the node is active; and false if it is “stale.”

The erase method changes this data member to false; iterator methods verify that the node is not

stale.

**3.19** Without head or tail nodes the operations of inserting and deleting from the end becomes a *O(N)*

operation where the *N* is the number of elements in the list. The algorithm must walk down the

list before inserting at the end. With the head node insert needs a special case to account for when

something is inserted before the first node.

**3.20 (a)** The advantages are that it is simpler to code, and there is a possible saving if deleted keys are

subsequently reinserted (in the same place). The disadvantage is that it uses more space, because

each cell needs an extra bit (which is typically a byte), and unused cells are not freed.

**(b)** Add a int numDeleted to the list (initialized to 0) and a bool deleted to Node (initialized to false)

void garbageCollection()

{

Node \*p = head->next;

Node \*q ;

while (p!= tail)

{

if (p->deleted == true)

{

q = p->prev;

p->prev->next = p->next;

p->next->prev = p->prev;

delete p;

p = q;

}

p = p->next;

}

numDeleted = 0;

}

iterator erase( iterator itr )

{

Node \*p = itr.current;

Node \*q; p->deleted = true;

numDeleted ++;

theSize--;

if (numDeleted >= theSize)

garbageCollection();

iterator retVal(\*this, p->next );

return retVal;

}

3.21 **(b)**

/\*

Weiss Exercise 3\_21 Balanced parenthesis for c++

\*/

#include <iostream>

#include <stack>

#include <string>

#include <fstream>

using namespace std;

int main()

{

string fileName;

stack<char> match;

bool balanced = true;

string line;

ifstream in;

int i;

int lineNumber = 0;

char x;

cout<<"what is the name of the file: ";

cin>>fileName;

in.open(fileName.c\_str());

getline(in, line);

while (in && balanced)

{

lineNumber++;

for ( i= 0; i< line.size(); i++)

{

x = line[i];

if (x == '\"') // skip chars in double quotes

{

i++;

while (x != '\"')

x++;

}

else if (x == '\'') // skip chars in single quotes

{

i++;

while (x != '\'')

x++;

}

else if (x == '\\' && i < line.size()-1 && line[i+1] == '\\')

getline(in, line);

else if (x == '(' || x == '['|| x == '{')

match.push(x);

else if (x == ')')

{

if (match.top()!='(') balanced = false;

match.pop();

}

else if (x == ']')

{

if (match.top()!='[') balanced = false;

match.pop();

}

else if (x == '}')

{

if (match.top()!='{') balanced = false;

match.pop();

}

else if (x == '/' && i < line.size() -1 && line[++i] == '\*')

match.push('/');

else if (x== '\*' && i < line.size() -1 && line[++i] == '/')

{

if (match.top()!='/') balanced = false;

match.pop();

}

}

getline(in, line);

}

if (balanced)

cout<<"the symbols balanced \n";

else

cout<<"mismatched symbol on line "<<lineNumber<<endl;

}

**3.22** The following function evaluates a postfix expression, using +, −, ∗, */* and ^ ending in =. It requires

spaces between all operators and = and uses the stack, string and math.h libraries. It only recognizes

0 in input as 0.0.

double evalPostFix( )

{

stack<double> s;

string token;

double a, b, result;

cin>> token;

while (token[0] != ’=’)

{

result = atof (token.c\_str());

if (result != 0.0 )

s.push(result);

else if (token == "0.0")

s.push(result);

else

switch (token[0])

{

case ’+’ : a = s.top(); s.pop(); b = s.top();

s.pop(); s.push(a+b); break;

case ’-’ : a = s.top(); s.pop(); b = s.top();

s.pop(); s.push(a-b); break;

case ’\*’ : a = s.top(); s.pop(); b = s.top();

s.pop(); s.push(a\*b); break;

case ’/’ : a = s.top(); s.pop(); b = s.top();

s.pop(); s.push(a/b); break;

case ’^’ : a = s.top(); s.pop(); b = s.top();

s.pop(); s.push(exp(a\*log(b))); break;

}

cin>> token;

}

return s.top();

}

**3.23 (a, b)** This function will read in from standard input an infix expression of single lower case

characters and the operators, +, −, */* , ∗, ^ and *(*, *)*, and output a postfix expression.

void inToPostfix()

{

stack<char> s;

char token;

cin>> token;

while (token != ’=’)

{

if (token >= ’a’ && token <= ’z’)

cout<<token<<" ";

else

switch (token)

{

case ’)’ : while(!s.empty() && s.top() != ’(’)

{ cout<<s.top()<<" "; s.pop();}

s.pop(); break;

case ’(’ : s.push(token); break;

case ’^’ : while(!s.empty() && !(s.top()== ’^’ || s.top() == ’(’))

{cout<<s.top(); s.pop();}

s.push(token); break;

case ’\*’ :

case ’/’ : while(!s.empty() && s.top() != ’+’ && s.top() != ‘-‘ && s.top() != ‘(‘)

{cout<<s.top(); s.pop();}

s.push(token); break;

case ‘+’ :

case ‘-‘ : while(!s.empty() && s.top() != ‘(‘ )

{cout<<s.top()<<’’ ‘‘; s.pop();}

s.push(token); break;

}

cin>> token;

}

while (!s.empty())

{cout<<s.top()<<’’ ‘‘; s.pop();}

cout<<ˇ = \nˇ;

}

**(c)** The function converts postfix to infix with the same restrictions as above.

string postToInfix()

{

stack<string> s;

string token;

string a, b;

cin>>token;

while (token[0] != ’=’)

{

if (token[0] >= ’a’ && token[0] <= ’z’)

s.push(token);

else

switch (token[0])

{

case ’+’ : a = s.top(); s.pop(); b = s.top(); s.pop();

s.push("("+ a+" + " + b+")"); break;

case ’-’ : a = s.top(); s.pop(); b = s.top(); s.pop();

s.push("("+a+" - "+ b+")"); break;

case ’\*’ : a = s.top(); s.pop(); b = s.top(); s.pop();

s.push("("+a+" \* "+ b+")"); break;

case ’/’ : a = s.top(); s.pop(); b = s.top(); s.pop();

s.push("("+a+" / " + b+")"); break;

case ’^’ : a = s.top(); s.pop(); b = s.top(); s.pop();

s.push("("+a+" ^ " + b+")"); break;

}

cin>> token;

}

return s.top();

} //Converts postfix to infix

**3.24** Two stacks can be implemented in an y array by having one grow from the low end of the array up,

and the other from the high end down.

**3.25 (a)** Let *E* be our extended stack. We will implement *E* with two stacks. One stack, which we’ll call

*S*, is used to keep track of the *push* and *pop* operations, and the other*M*, keeps track of the minimum.

To implement *E.push (x)*, we perform *S.push (x)*. If *x* is smaller than or equal to the top element in

stack *M*, then we also perform *M.push (x)*. To implement *E.pop*( ) we perform *S.pop*( ). If *x* is equal

to the top element in stack *M*, then we also *M.pop*( ). *E.findMin*( ) is performed by examining the

top of *M*. All these operations are clearly *O*(1).

**(b)** This result follows from a theorem in Chapter 7 that shows that sorting must take (*N* log *N*)

time. *O*(*N*)operations in the repertoire, including *deleteMin*, would be sufficient to sort.

**3.26** Three stacks can be implemented by having one grow from the bottom up, another from the top

down and a third somewhere in the middle growing in some (arbitrary) direction. If the third stack

collides with either of the other two, it needs to be moved. A reasonable strategy is to move it so that

its center (at the time of the move) is halfway between the tops of the other two stacks.

**3.27** Stack space will not run out because only 49 calls will be stacked. However the running time is

exponential, as shown in Chapter 2, and thus the routine will not terminate in a reasonable amount

of time.

**3.28** This requires a doubly linked list with pointers to the head and the tail In fact it can be implemented

with a list by just renaming the list operations.

template <typename Object>

class deque

{

public:

deque() { l();}

void push (Object obj) {l.push\_front(obj);}

Object pop (); {Object obj=l.front(); l.pop\_front(); return obj;}

void inject(Object obj); {l.push\_back(obj);}

Object eject(); {pop\_back(obj);}

private:

list<Object> l;

}; //

**3.29** Reversal of a singly linked list can be done recursively using a stack, but this requires *O (N)* extra

space. The following solution is similar to strategies employed in garbage collection algorithms (*first*

represents the first node in the non-empty node in the non-empty list). At the top of the *while* loop

the list from the start to *previousPos* is already reversed, whereas the rest of the list, from *currentPos*

to the end is normal. This algorithm uses only constant extra space.

//Assuming no header and that first is not NULL

Node \* reverseList(Node \*first)

{

Node \* currentPos, \*nextPos, \*previousPos;

previousPos = NULL;

currentPos = first;

nextPos = first->next;

while (nextPos != NULL)

{

currentPos -> next = previousPos;

perviousPos = currentPos;

currentPos = nextPos;

nextPos = nextPos -> next;

}

currentPos->next = previousPos;

return currentPos;

}

**3.30 (c)** This follows well-known statistical theorems. See Sleator and Tarjan’s paper in Chapter 11 for

references.

**3.31**

template <typename Object>

struct node

{

node () { next = NULL;}

node (Object obj) : data(obj) {}

node (Object obj, node \* ptr) : data(obj), next(ptr) {}

Object data;

node \* next;

};

template <typename Object>

class stack

{

public:

stack () { head = NULL;}

~stack() { while (head) pop();

}

void push(Object obj)

{

node<Object> \* ptr = new node<Object>(obj, head);

head= ptr;

}

Object top()

{return (head->data); }

void pop()

{

node<Object> \* ptr = head->next;

delete head;

head = ptr;

}

private:

node<Object> \* head;

};

**3.32**

template <typename Object>

class queue

{

public:

queue () { front = NULL; rear = NULL;}

~queue() { while (front) deque(); }

void enque(Object obj)

{

node<Object> \* ptr = new node<Object>(obj, NULL);

if (rear)

rear= rear->next = ptr;

else

front = rear = ptr;

}

Object deque()

{

Object temp = front->data;

node<Object> \* ptr = front;

if (front->next == NULL) // only 1 node

front = rear = NULL;

else

front = front->next;

delete ptr;

return temp;

}

private:

node<Object> \* front;

node<Object> \* rear;

}; //

**3.33** This implementation holds maxSize −1 elements.

template <typename Object>

class queue

{

public:

queue(int s): maxSize(s), front(0), rear(0) {elements.resize(maxSize);}

queue () { maxSize = 100; front = 0;

rear = 0;elements.resize(maxSize);}

~queue() { while (front!=rear) deque(); }

void enque(Object obj)

{

if (! full())

{

elements[rear] = obj;

rear = (rear + 1) % maxSize;

}

}

Object deque()

{

Object temp;

if (!empty())

{

temp= elements[front];

front = (front +1 ) % maxSize;

return temp;

}

}

bool empty() {return front == rear;}

bool full() { return (rear + 1) % maxSize == front;}

private:

int front, rear;

int maxSize;

vector<Object> elements ;

}; //

**3.34 (b)** Use two iterators *p* and *q*, both initially at the start of the list. Advance *p* one step at a time,

and *q* two steps at a time. If *q* reaches the end there is no cycle; otherwise, *p* and *q* will eventually

catch up to each other in the middle of the cycle.

**3.35 (a)** Does not work in constant time for insertions at the end

**(b)** Because of the circularity, we can access the front item in constant time, so this works.

**3.36** Copy the value of the item in the next node (that is, the node that follows the referenced node) into

the current node (that is, the node being referenced). Then do a deletion of the next node.

**3.37 (a)** Add a copy of the node in position *p* after position *p*; then change the value stored in position

*p* to *x*.

**(b)** Set p->data = p->next->data and set p->next = p->next->next. Then delete p->next. Note that the

tail node guarantees that there is always a next node.