Abstract Data Types - Objects with operations

----------------------------The List ADT------------------------------

A list, we assume will have elements ordered one after the other, but NOT SORTED.

Just arbitrarily ordered one after the other.

We can give this LIST different functions, printList, makeEmpty, find, remove, etc..

-----------ArrayList------------------

These ADT can only have OBJECTS inside them

------------Linked Lists---------------

To avoid the potentially great cost of insertion/deletion, (removing/adding an element at the beginning of the list

we have a LINKED LIST and shifting the rest of the list down/up)

Each linked list contains a series of nodes that might not have to be totally adjacent in memory.

Each node contains the element of the list, and a ink to a node containing its successor ---

that's called the "next" link

The last cell's "next" link references "null"

A DOUBLY LINKED LIST maintains a link to its previous node in the list.

-----------------------------Lists in the Java COLLECTIONS API---------------------------

-----Methods in Collection Interface----------

size - returns number of items in the collection

isEmpty - returns true iff the size of the collection is zero

contains - returns true if "x" is in the collection

add

- these add/remove item "x" from the collection, returning TRUE if it succeeds, FALSE if it fails

remove

----------Iterable-------

The Collection interface extends the ITERABLE interface

Classes that implement the ITERABLE interface can have ENHANCED FOR LOOPS used on them to shift through/view all their items

Iterables (and therefore, iterators), simply allow you to move through a collection.

---------------------------Iterators------------------------------

Collections that implement the Iterable interface must provide a METHOD named "iterator" that returns an object of type Iterator.

ITERATOR is an interface defined also in package java.util

The Iterator interface in package java.util:

public interface Iterator<AnyType>

{

boolean hasNext(); - says if there is a next item

AnyType next(); - each call here gives the next item in the collection (first call gives first item, second call gives second, etc...)

void remove(); - remove the last item returned by "next"

} NOTICE: you cant "remove" again until ANOTHER CALL to "next"

-----PURPOSE of the Iterator: VIA the iterator METHOD, each collection can create, and return an object

implementing the Iterator interface that also stores internally its notion of a CURRENT POSITION (returns an object that uses Iterator)

Ex:

Here, the "print" method is rewritten with Iterator

public static <AnyType> void print( Collection<AnyType> coll )

{

Iterator<AnyType> itr = coll.iterator();

whilte( itr.hasNext() )

{

AnyType item = itr.next();

System.out.println( item );

} }

When the compiler sees the enhanced for loop used on object that is ITERABLE,

it REPLACES enhanced for loop with calls to the "iterator" method to obtain an Iterator, and then calls to "next" and "hasNext"

Ex:

public static final void main(String[] args) {

ArrayList<Integer> a = new ArrayList<Integer>();

a.add(2);

a.add(1);

a.add(3);

a.add(4);

for( Integer i : a) {

System.out.println(i);

}

PRINTS:

2

1

3

4

public static final void main(String[] args) {

ArrayList<Integer> a = new ArrayList<Integer>();

a.add(2);

a.add(1);

a.add(3);

a.add(4);

for( Integer i : a) {

System.out.println(i);

System.out.println();

java.util.Iterator<Integer> n = a.iterator(); - Iterator is generic, so SPECIFY the class - "<Integer>"

while(n.hasNext()) {

Integer i = n.next();

System.out.println(i); } }

PRINTS:

2

1

3

4

2 - Showing you what iterators can do, they come handy when you want to keep track of TWO different locations/speeds

1

3

4

NOTICE: Iterators always start at the beginning. You use them when to go through the WHOLE collection/arraylist

-------------------------------------The List Interface, ArrayList, and LinkedList------------------------------------------

"get" and "set" allow us to access or change an item at the specified position in the list, given by its index "idx"

Index 0 is at the front of the list, size() - 1 represents the last item in the list

Index size() represents the position where a newly addded item can be placed

"add" allows placement of a new item in position "idx"

- "add" at position 0 is adding at the front

- "add" at position size() is adding an item as the new last item

In addition to the standard "remove" that takes AnyType as a parameter,

"remove" is also overloaded to remove an item at a specified position

Also specifies the "listIterator" method that produces a more complicated iterator

public interface Lit<AnyType> extends Collection<AnyType>

{

AnyType get ( int idx );

AnyType set ( int idx, AnyType newVal );

void add ( int idx, AnyType x );

void remove ( int idx );

List Iterator<AnyType> listIterator ( int pos );

}

---------------Implementing the List ADT-------------------

ArrayList - an array implementation of the List ADT

Advantages: "get" and "set" take constant time

Disadvantage: insertion of new items and removal of existing items is expensive, UNLESS these actions are made at the END of the ArrayList

LinkedList - a doubly linked list implementation of the List ADT

Advantages: insertion of new items and removal of existing ones is cheap, AS LONG AS the position of the changes IS KNOWN

- the adds and removes from the front of the list are constant-time operations

- even provides the methods "addFirst" and "removeFirst", "addLast" and "removeLast", and "getFirst" and "getLast" to efficiently add, remove, and access items are both ends of the list

Disadvantage: not easily indexable, calls to "get" expensive UNLESS they are very close to one of the ends of the list (the search can proceed from either the back or the front of the list)

-----Ex: Let's look at the differences

1. Construct a list by adding items at the end

public static void makeList1( List<Integer> lst, int N )

{

lst.clear();

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

lst.add(i);

}

- Whether it's an ArrayList or a LinkedList that's passed as a parameter, the running time of makeList1 is O(N),

because each call to "add", being at the end of the list, takes constant time

2. However, if we construct a List by adding items at the front

public static void makeList2( List<Integer> lst, int N)

{

lst.clear();

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

lst.add( 0, i );

}

- Running time is O(N) for a LinkedList, but O(N^2) for an ArrayList

- In an ArrayList, adding at the front is an O(N) operation - moving all the elements back

3. Computing the sum of the numbers in a List:

public static int sum( List<Integer> lst)

{

int total = 0;

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

total += lst.get( i );

return total;

}

- Running time is O(N) for ArrayList but O(N^2) for a LinkedList.

- In a LinkedList, calls to "get" are O(N) operations

- INSTEAD: use an enhanced "for" loop, which will make the running time O(N) for any List (iterator efficiently advances from one item to the next)

- It uses an ITERATOR which would be able to call the element at "i" immediately as it moves from one item to the the next

Both ArrayList and LinkedList are inefficient for searhes, so calls to the Collection "contains" and "remove" methods (that have parameter AnyType), take LINEAR TIME

NOTE: The ArrayList automatically increases the capacity as needed to ensure that it is AT LEAST as large as the size of the list

"ensureCapacity" - If you can estimate the size BEFOREHAND, "ensureCapacity" can set the capacity to a sufficiently large amount BEFOREHAND to avoid a later expansion of the array capacity

"trimToSize" - trims the ArrayList to the least amount of space needed AFTER all the ArrayList adds are completed

- This avoids wasting space

----------------------------------Example: Using "remove" on a LinkedList--------------------------------------

Removing from almost anywhere in an ArrayList is expensive (we have to shift all the elements every time)

LinkedLists just need a rearranging of some links.

1st Attempt:

public static void removeEvensVer1( List<Integer> lst)

{

int i = 0;

while( i < lst.size() )

if (lst.get( i ) % 2 = 0)

lst.remove( i ); - For an ArrayList, the "remove" method ALSO shifts the elements over ON ITS OWN.

else - For a LinkedList, it goes through the list AGAIN BY ITSELF to get to the position i

i++; - EVEN THOUGH the "get" method was called previous, because this action is INCLUDED WITHIN the "remove" method's function

}

- As explained beforehand, the ArrayList is not efficient here because of the "remove" method

- the LinkedList is not effective because the call to "get" is inefficient and the call to "remove" is inefficient NOT BECAUSE OF SHIFTING, but because of the run-through of the array required to get to the defined position separately

- The runtime is quadratic for both lists

2nd Attempt:

public static void removeEvensVer2( List<Integer> lst )

{

for (Integer x : lst)

if( x % 2 == 0 )

lst.remove( x );

}

- We use an iterator instead of "get" to step through the list, which is efficient.

- BUT: we use the Collections "remove" method to remove the even-valued item.

- This is NOT EFFICIENT because the "remove" method has to SEARCH FOR THE ITEM AGAIN ON ITS OWN, taking linear time

- Which makes the program OVERALL QUADRATIC cause of the iterator referred to by the enhanced "for" loop

- The list ALSO generates an EXCEPTION because when the item is removed, the iterator used by the enhanced for loop is INVALIDATED.

- Basically, the enhanced for loop doesnt understand that it can only proceed if an item is NOT REMOVED

3rd Attempt:

public static void removeEvensVer3( List<Integer> lst )

{

Iterator<Integer> itr = lst.iterator();

while ( itr.hasNext() )

if( itr.next() % 2 == 0 )

itr.remove();

}

- We use an iterator to do everything.

- Removing the element with the iterator is constant time (operation of 1), cause the iterator is already at the node that needs to be removed

- Thus, the entire program takes linear time for a LinkedList

- HOWEVER, even though the iterator is already at the point that needs to be removed, "remove" is still expensive because the array items must be SHIFTED

- Thus, the entire program takes quadratic time for an ArrayList

- Passing in a LinkedList<Integer>, we see that this holds true.

-------------------------------------------ListIterators---------------------------------------

A ListIterator adds more function to the Iterator for Lists

"previous" and "hasPrevious" allows back and forth movement of the list - self-explanatry, follow same logic as "next" and "hasNext"

"add" - places a new item into the list in the current position BETWEEN the values it would call for "next" and "previous"

NOTE: The CURRENT POSITION is the space BETWEEN two values. NOT AT some value.

- Remember, "add" is a constant-time operation for a LinkedList and expensive for an ArrayList (where elements must be shifted)

"set" - Changes the last value seen by the iterator - CONVENIENT for LinkedLists

- Regular iterators can't do anything to the element they're on other than the method commands already given, ("add", "remove", etc..), they can't do "subtract 1 from that element"

-----An implementation of ArrayList in the bank shows a usable, self-constructed ArrayList generic class

----------Generic Array Creation------

An idiom that is required because direct generic array creation is illegal.

Ex: theItems = (AnyType []) new Object[ newCapacity];

Instead, we create an array of the generic type's bound and then use an array cast.

- This will give us an unavoidable compiler warning.

-----------

Two versions of add:

public boolean add(AnyType x) {

add( size(), x); - Trivialy implemented by calling the other version that adds at the specified position

return true;

}

public void add(int idx, AnyType x) {

if (theItems.length == size() )

ensureCapacity( size() \* 2 + 1); - Doubles the capacity, includes the +1 at the end in case the size of the array is zero

for (int i = theSize; i > idx; i-- )

theItems[i] = theItems[ i - 1]; - Computationally expensive, shifts elements that are after the specified position an additional position higher

theItems[ idx ] = x; - Then sets the desired location to the chosen value - thus, "adding" it

theSize++;

}

--------------------------------More Specifics on Implementing LinkedList---------------

To implement the LinkedList, we must provide three classes:

1. The MyLinkedList class itself, containing:

- Links at both ends

- Size of the list

- A host of methods

2. The Node class, likely a private nested class, containing:

- data

- links to the previous and next nodes

- appropriate constructors

3. LinkedListIterator, abstracting the notion of a position and is a private inner class IMPLEMENTING the Iterator interface:

Provides the implementations of:

- next

- hasNext

- remove

Because the iterator classes store a reference to the "current node", and the end marker is a valid position,

we can create an EXTRA NODE at the END of the list to represent the END MARKER

We can also create an extra node at the FRONT of the list, representing the BEGINNING MARKER

SENTINEL NODES: the extra nodes at the front and back

HEADER node = beginning marker = node at the front

TAIL node = end marker = node at the back

------------Advantages------------

In order to remove the first node, we must reset the list's link to the first node.

HOWEVER: because the remove algorithm needs access to the node PRIOR to the node being removed,

we need a "header" node to provide that prior node

An empty list contains header and tail nodes pointed at each other

----------------------------------------------The Code---------------------------------------------

-------Node class--------

At the beginning, we have the private nested Node class, which consists of:

- the stored data

- links to the previous and next Node

- and a constructor

private static class Node<AnyType>

{

public Node( AnyType d, Node<AnyType> p, Node<AnyType> n)

{ data = d; prev = p; next = n; }

public AnyType data; - Usually the data members are private, BUT because the Node class is private, the visibility of the data members is irrelevant.

public Node<AnyType> prev; - Classes outside of MyLinkedList can't see the Node class at all anyways

public Node<AnyType> next;

}

--------Data Members (MyLinkedList)-------------

private int theSize; - Keeping track of the size in a variable so that the "size()" method can be implemented in constant time

- (just needs to return this variable to give size)

private int modCount = 0; - Represents number of changes to the linked list since construction

- Each all to "add", "remove", "doClear()", etc... will update modCount

- Each call to an iterator method (next or remove) will check the stored "modCount" in the iterator with the current "modCount" in the linked list

and will throw a "ConcurrentModificationException" if these two counts don't match

- (??) Ensuring that only the iterator is modifiying the Linked list, and that it's doing it one step at a time??

private Node<AnyType> beginMarker;

private Node<AnyTYpe> endMarker;

--------------------doClear()---------------------------

public void clear()

{ doClear(); }

private void doClear()

{

beginMarker = new Node<AnyType>(null, null, null);

endMarker = new Node<AnyType>(null, beginMarker, null); - NOTE: the previous Node is set to beginMarker.

This makes sense because in an empty list, the tail node and the head node are pointing towards each other

beginMarker.next = endMarker; - Same logic as previously stated is applied here as well

theSize = 0;

modCount++;

}

---------------------Inserting a New Node-------------------------

Let "p" be some node in the LinkedList

Node newNode = new Node(x, p.prev, p); - Even though we set this node to reference "p.prev" and "p", we must still link FROM "p" to the newNode (DOUBLY LINKED LIST)

p.prev.next = newNode; - The "next" reference of the node PREVIOUS to p is SET to the new node

- AKA (p.prev).next = newNode;

p.prev = newNode; - The node in front of newNode, "p" is set to reference new Node as its previous node

----- These steps can be combined ------

Node newNode = new Node(x, p.prev, p);

p.prev = p.prev.next = newNode;

---- These steps can be FURTHER combined ----

p.prev = p.prev.next = new Node(x, p.prev, p);

-----------------------------------------------addBefore----------------------------

private void addBefore( Node<AnyType> p, AnyType x)

{

Node <AnyType> newNode = new Node<>(x, p.prev, p);

newNode.prev.next = newNode; - Same thing as "p.prev.next"

p.prev = newNode;

theSize++;

modCount++;

}

-------------------------------Removing a Node-------------------------------

Only two links change for a node to be disconnected and eligible to be reclaimed by the Virtual Machine

private AnyType remove(Node<AnyType> p)

{

p.prev.next = p.next; - Setting the "next" reference of the node previous to "p", to the node in front of "p"

p.next.prev = p.prev; - Setting the "previous" reference of the next in front of "p", to the node behind "p"

theSize--;

modCount++;

return p.data;

}

BASICALLY, we're using "p" to set the nodes in front and behind it towards each other, thus ignoring "p" and ""disconnecting it from the list""

------------------------------getNode--------------------------------------------

If the Node is in the first half of the list, then the program starts from the BEGINNING of the Linked list and proceeds from there

If the Node is in the later half of the list, then the program starts at the END of the Linked list and works its way backwards (decrementing the loop counter instead)

private Node<AnyType> getNode( int idx )

{

return getNode( idx, 0, size() - 1 );

}

private Node<AnyType> getNode( int idx, int lower, int upper)

{

Node<AnyType> p;

if ( idx < lower || idx > upper)

throw new IndexOutofBoundsException();

if ( idx < size() / 2) - Testing if the Node is in the first half of the linked list

{

p.beginMarker.next; - Starting at first node

for ( int i = 0; i < idx; i++ ) - Proceeding forwards

p = p.next;

}

else

{

p = endMarker; - Starting at the end marker

for ( int i = size(); i > idx; i--) - Proceeding backwards towards the front

p = p.prev;

}

return p;

}

THIS is why the "get" method of Linked Lists are so EXPENSIVE

-----------------------LinkedListIterator-------------------------------------

Maintains a current position

Does the job of an iterator, but calls method(s) from Linked List

----HAS SIGNIFICANT ERROR CHECKING

Stores "modCount" of the Linked list into "expectedModCount" - in order to detect a situation in which the collection has been modified DURING THE ITERATION - ( ANY UNPLANNED SIDE EFFECTS/EDITS )

Uses "okToRemove" to check if the iterator has advanced to the next Node yet, before trying to removing something AGAIN

- It's only set to false after a remove operation has been done

- That's because the remove operation needs a note to act on, it can't remove an EMPTY SPACE

Uses "hasNext" to make the "next" call illegal when "current" is positioned at the endMarker

private class LinkedListIterator implements java.util.Iterator<AnyType>

{

private Node<AnyType> current = beginMarker.next;

private int expectedModCount = modCount;

private boolean okToRemove = false;

public boolean hasNext()

{ return current != endMarker; }

public AnyType next()

{

if (modCount != expectedModCount ) - Checks to make sure that the collection hasn't been modified during the iteration (unknowingly)

throw new java.util.ConcurrentModificationException();

if (!hasNext() ) - If there's no node in front, then "next()" terminates and throws an Error message

throw new java.util.NoSuchElementException();

AnyType nextItem = current.data; - Stores the current position to be returned.

- TO THE USER: it's the data that they haven't gotten yet. The information for the "next" node, where the "next()" is already at the "next" node

current = current.next;

okToRemove = true; - Allows the iterator to remove an element, now that it has proceeded to the next node

return nextItem;

}

public void remove()

{

if ( modCount != expectedModCount )

throw new java.util.ConcurrentModificationException();

if ( !okToRemove )

throw new IllegalStateException();

- We call the remove method of MyLinkedList

MyLinkedList.false.remove( current.prev ); - NOTICE: "current" remains unchanged, because the node that "current" is viewing is NOT AFFECTED by the removal of the prior node

- The iterator would have already proceeded to the next node if this remove method was accessible (AKA if okToRemove is set to true)

- In the ArrayListIterator, we had to shift our items back to compensate for the space after our removal,

so we had to update "current" to the previous node to make sure we were in the node PREVIOUS to the removal

expectedModCount++;

okToRemove = false; - Can't call remove again if we just did it, cause then we've ALREADY disconnected the node

}

}

-----------------------------Iterator and Java Nested and Inner Classes----------------

The ArrayListIterator uses a Java construct known as the "inner class"

----------------Postfix ++ operator------------------

arr[idx++] - Uses "idx" in the array and THEN increments "idx"

-----------------Prefix ++ operator---------------------

arr[++idx] - FIRST Increments "idx" and THEN uses the NEW "idx" in the array

--------Using outside variables in Inner Classes-----------------

------3rd Attempt: SUCCESSFUL

SOLVED: ArrayListIterator is made a NESTED CLASS, and placed inside MyArrayList, which is now its OUTER CLASS

"nested class" - is identified as STATIC to signify that it's nested

- Without "static", it's an INNER CLASS (which is sometimes good sometimes bad, just stick with "nested")

The nested class is also made PRIVATE so that ONLY the OUTER CLASS MyArrayList can access it.

Now, because the "nested" class is a part of the "outer" class, no visibility issues arise.

public class MyArrayList<AnyType> implements Iteraeble<AnyType>

{

private int theSize;

private AnyType [] theItems;

...

public java.util.Iterator<AnyType> iterator() {

return new ArrayListIterator<AnyType>(this);}

private static class ArrayListIterator<AnyType> implements java.util.Iterator<AnyType> - DECLARED private static and PLACED WITHIN MyArrayList

{

private int current = 0;

private MyArrayList<AnyType> theList;

...

public ArrayListIterator( MyArrayList<AnyType> list) {

theList = list;

}

public boolean hasNext()

{ return current <theList.size(); }

public AnyType next()

{ return theList.theItems[current++]; } - No visibility issues now

}

}

-----4th Alternative-----------------Inner classes-----------------------

When we declare an inner class, the compiler implicitly references the OUTER class as "[Outer].this"

Therefore, if ArrayListIterator was declared as an inner class without the STATIC, then "MyArrayList.this" and "theList" would both be referencing the same "MyArrayList"

- "theList would then be redundant and thus removed

USEFUL: when each inner class oboject is associated with exactly one instance of an outer class object

(Cannot exist without having an outer class object with which to be associated first)

Ex: MyArrayList and ArrayListIterator would both point to the same object, IF ArrayListIterator were an INNER CLASS

public class MyArrayList<AnyType> implements Iterable<AnyType>

{

private int theSize;

private AnyType [] theItems;

...

public java.util.Iterator<AnyType> iterator() {

return new ArrayListIterator(); - We don't have to use "<AnyType>" specification, because ArrayListIterator is IMPLICITLY generic, and it's tied to MyArrayList

}

private class ArrayListIterator<AnyType> implements java.util.Iterator<AnyType> - DECLARED private static and PLACED WITHIN MyArrayList

{

private int current = 0;

- "MyArrayList<AnyType> theList" and its corresponding constructor are removed to eliminate redundancy

public boolean hasNext()

{ return current < size(); }

public AnyType next()

{ return theItems[current++]; } - theList.theItems can be replaced with "MyArrayList.this.theItems"

HOWEVER: as long as there is NO OTHER variable named "theItems" inside the INNER CLASS, this can be written SIMPLY as "theItems"

- Same logic applied to "theList.size()"

public void remove() - We can implement the iterator's "remove" by calling MyArrayList's "remove".

{MyArrayList.this.remove( --current );} HOWEVER: because MyArrayList's "remove" would CONFLICT with ArrayListIterator's "remove", we use "MyArrayList.this.remove"

}

}

NOTICE: this code reverts back to the style in Attempt #1

BASICALLY, we can write the code that feels SYNTACTICALLY NATURAL as in ATTEMPT #1 through INNER CLASSES

--------------------------Queues-------------------------------------------

Like stacks, queues are lists.

However, while insertion is done at one end, deletion is done at THE OTHER.

---------Basic Operations-----------

enqueue - inserts an element at the end of the list (the rear)

dequeue - deletes AND RETURNS the element at the start of the list (the front)

-------------------------------------

Like stacks, any list implementation can work for queues.

Both the linkedlist and the array implementations give fast O(1) (constant) running times for every operation.

The linkedlist implementation is very straightforward - just look at the commands it has.

We will instead discuss the array implementation of queues.

-------------------------Array Implementation of Queues--------------------------

theArray - the array we're using

front - the front position in the array

back - the back position in the array (the ends of the queue)

currentSize - the number of elements that are in the queue

------Operations------

To "enque" an element "x":

1. increment "currentSize"

2. increment "back"

3. set "theArray[back] = x"

To "dequeue" an element "x":

1. set return value to "theArray[front]"

2. decrement "currentSize"

3. increment "front"

Other strategies are possible.

----Circular Array Implementation-----

When "front" or "back" gets to the end of the array, it is wrapped around to the beginning.

- This is because even though we might've enqueued a lot to the point that the queue seems full,

we might've also DEQUEUED a lot, so that there were barely any elements in the FRONT of the array

As a result, that's why we implement "circular array" implementation.

Only a minimal amount of code is required to implement this:

If incrementing EITHER "back" or "front" causes it to go past the array, the value is reset to the

first position in the array.

Remember: "back" and "front" travel IN THE SAME DIRECTION along the array. Both move forward