Advanced Object-oriented Programming

Lecture 10

More Lists
We saw that adding to the start of a linked list is straightforward, but that adding to the end of a list required a loop to traverse the list to find the end.

We could improve on this by keeping pointers to both the start and the end of the list.
Start and End of Lists

We saw that adding to the start of a linked list is straightforward, but that adding to the end of a list required a loop to traverse the list to find the end.

We could improve on this by keeping pointers to both the start and the end of the list.
Start and End of Lists

head  tail

head  tail

head  tail

listStart

listEnd
public class List {

    private Node listStart;
    private Node listEnd;

    public void add(BandCard b) {
        listStart = new Node(b, listStart);
        if (listEnd == null) { // list was empty
            listEnd = listStart; // 1 card in list
        }
    }
}
public class List {

    private Node listStart;
    private Node listEnd;

    public void add(BandCard b) {
        listStart = new Node(b, listStart);
        if (listEnd == null) { // list was empty
            listEnd = listStart; // 1 card in list
        }
    }
}
public class List {

    private Node listStart;
    private Node listEnd;

    public void add(BandCard b) {
        listStart = new Node(b, listStart);
        if (listEnd == null) { // list was empty
            listEnd = listStart; // 1 card in list
        }
    }
}
public class List {

    private Node listStart;
    private Node listEnd;

    public void add(BandCard b) {
        listStart = new Node(b, listStart);
        if (listEnd == null) { // list was empty
            listEnd = listStart; // 1 card in list
        }
    }
}
public class List {
    private Node listStart;
    private Node listEnd;

    public void add(BandCard b) {
        listStart = new Node(b, listStart);
        if (listEnd == null) { // list was empty
            listEnd = listStart; // 1 card in list
        }
    }
}
class List v1.1

public class List {

    private Node listStart;
    private Node listEnd;

    public void add(BandCard b) {
        listStart = new Node(b, listStart);
        if (listEnd == null) { // list was empty
            listEnd = listStart; // 1 card in list
        }
    }
}
public void addend(BandCard b) {
    if (listStart == null) {
        add(b);
    } else {
        listEnd.addend(b);
        listEnd = listEnd.tail();
    }
}
public void addend(BandCard b) {
    if (listStart == null) {
        add(b);
    } else {
        listEnd.addend(b);
        listEnd = listEnd.tail();
    }
}

public void addend(BandCard b) {
    if (listStart == null) {
        add(b);
    } else {
        listEnd.addend(b);
        listEnd = listEnd.tail();
    }
}
public void addend(BandCard b) {
    if (listStart == null) {
        add(b);
    } else {
        listEnd.addend(b);
        listEnd = listEnd.tail();
    }
}
Notes

- There is no constructor.
- There are only two fields, both of type `Node`. The default value for any variable of Object type (i.e., belonging to some class) is `null`, which in our case represents the empty list.
- Thus, the fields are initialised by default to the empty list.
- Method `length()` is unchanged, but ...
- Method `tail()` (in class `List`) does need to be changed (exercise!).
There is no constructor. There are only two fields, both of type \texttt{Node}. The default value for any variable of Object type (i.e., belonging to some class) is \texttt{null}, which in our case represents the empty list. Thus, the fields are initialised by default to the empty list.

- \textbf{method} \texttt{length()} is unchanged, but . . .
- \textbf{method} \texttt{tail()} (in class \texttt{List}) \textit{does} need to be changed (exercise!).
Notes

- There is no constructor.
  There are only two fields, both of type `Node`. The default value for any variable of Object type (i.e., belonging to some class) is `null`, which in our case represents the empty list. Thus, the fields are initialised by default to the empty list.

- Method `length()` is unchanged, but …

- Method `tail()` (in class `List`) does need to be changed (exercise!).
Removing the Last Element of a List
Removing the Last Element of a List
Removing the Last Element of a List
Removing the Last Element of a List

```
head tail
listStart
head tail
null
```
Removing the Last Element of a List

```maude
file: removeLast.maude

in List

fmod REMOVE_LAST is

protecting LIST.

op removeLast : List -> List.

var B : BandCard.
var L : List.

eq removeLast(add(B, empty)) = empty.

cq removeLast(add(B, L)) = add(B, removeLast(L))
if L /= empty.

endfm
```
Removing the Last Element of a List

file: removeLast.maude

in List

fmod REMOVE_LAST is

  protecting LIST .

  op removeLast : List -> List .

  var B : BandCard .
  var L : List .

  eq removeLast(add(B, empty)) = empty .
  cq removeLast(add(B, L)) = add(B, removeLast(L)) if L /= empty .

endfm
Removing the Last Element of a List

file: removeLast.maude

\textbf{in} List

\textbf{fmod} REMOVE\_LAST \textbf{is}

\textbf{protecting} LIST.

\textbf{op} removeLast : List -> List.

\textbf{var} B : BandCard.

\textbf{var} L : List.

\textbf{eq} removeLast(add(B, empty)) = empty.

\textbf{cq} removeLast(add(B, L)) = add(B, removeLast(L)) if L \neq empty.

\textbf{endfm}
Removing the Last Element of a List

file: removeLast.maude

in List

fmod REMOVE_LAST is

  protecting LIST.

  op removeLast : List -> List.

  var B : BandCard.

  var L : List.

  eq removeLast(add(B, empty)) = empty.

  cq removeLast(add(B, L)) = add(B, removeLast(L)) if L /= empty.

endfm
Removing the Last Element of a List

One way of doing this is to loop through the list to find the second-last element, then setting its tail field to null, and setting listEnd to (what was) the second-last element. (Exercise!)

Because this is — potentially — a very common operation on lists, an efficient implementation is desirable.

One way of achieving this is through doubly-linked lists: each node has a pointer not just to the following node, but also a pointer to the preceding node.
Removing the Last Element of a List

One way of doing this is to loop through the list to find the second-last element, then setting its tail field to null, and setting listEnd to (what was) the second-last element. (Exercise!)

Because this is — potentially — a very common operation on lists, an efficient implementation is desirable.

One way of achieving this is through doubly-linked lists: each node has a pointer not just to the following node, but also a pointer to the preceding node.
Removing the Last Element of a List

One way of doing this is to loop through the list to find the second-last element, then setting its tail field to null, and setting listEnd to (what was) the second-last element. (Exercise!)

Because this is — potentially — a very common operation on lists, an efficient implementation is desirable.

One way of achieving this is through doubly-linked lists: each node has a pointer not just to the following node, but also a pointer to the preceding node.
Removing the Last Element of a List

One way of doing this is to loop through the list to find the second-last element, then setting its tail field to null, and setting listEnd to (what was) the second-last element. (Exercise!)

Because this is — potentially — a very common operation on lists, an efficient implementation is desirable.

One way of achieving this is through doubly-linked lists: each node has a pointer not just to the following node, but also a pointer to the preceding node.
Nodes with Two pointers

Here’s a picture of a node with two pointers.

![Diagram of a node with two pointers](image)

The class has three fields:

- one *(value)* to store a BandCard;
- two *(prev, tail)* to store (pointers to) nodes

We’ll call these things ‘BiNodes’
Nodes with Two pointers

Here’s a list of length 1:

The values of `prev` and `tail` are both `null` (empty list)
Nodes with Two pointers

here's a list of length 2:

first.tail is the second, and second.prev is the first.
Nodes with Two pointers

And here's a list of length 3:

first.prev is null, as is last.tail
Can we implement **BiNodes** (doubly-linking nodes) as a subclass of **Node**? (I.e., a **BiNode** is a **Node** with an extra pointer to the previous **BiNode**?)

class BiNode extends Node {
    // inherits private Node tail;
    private BiNode prev;
}

No.
Can we implement **BiNodes** (doubly-linking nodes) as a subclass of **Node**? (I.e., a **BiNode** is a **Node** with an extra pointer to the previous **BiNode**?)

```java
class BiNode extends Node {
    // inherits private Node tail;
    private BiNode prev;
}
```

No.
Can we implement BiNodes (doubly-linking nodes) as a subclass of Node?
(I.e., a BiNode is a Node with an extra pointer to the previous BiNode?)

```java
class BiNode extends Node {
    // inherits private Node tail;
    private BiNode prev;
}
```

No.
Can we implement BiNodes (doubly-linking nodes) as a subclass of Node? (I.e., a BiNode is a Node with an extra pointer to the previous BiNode?)

```java
class BiNode extends Node {
    // inherits private Node tail;
    private BiNode prev;
}
```

No.
class BiNode {
    private final BandCard value;
    private BiNode tail;
    private BiNode prev;

    public BiNode(BandCard b) {
        value = b;
    }
}
class BiNode {
    private final BandCard value;
    private BiNode tail;
    private BiNode prev;
    public BiNode(BandCard b) {
        value = b;
    }
}
class BiNode { 
  private final BandCard value;
  private BiNode tail;
  private BiNode prev;
  
  public BiNode(BandCard b) {
    value = b;
  }
}
class BiNode {
    private final BandCard value;
    private BiNode tail;
    private BiNode prev;

    public BiNode(BandCard b) {
        value = b;
    }
}
public BiNode(BandCard b, BiNode t) {
    value = b;
    tail = t;
    if (t != null) {
        t.prev = this;
    }
}

public BandCard head() {
    return value;
}
public BiNode(BandCard b, BiNode t) {
    value = b;
    tail = t;
    if (t != null) {
        t.prev = this;
    }
}

public BandCard head() {
    return value;
}
public BiNode(BandCard b, BiNode t) {
    value = b;
    tail = t;
    if (t != null) {
        t.prev = this;
    }
}

public BandCard head() {
    return value;
}
public BiNode(BandCard b, BiNode t) {
    value = b;
    tail = t;
    if (t != null) {
        t.prev = this;
    }
}

public BandCard head() {
    return value;
}
public void addend(BandCard b) {
    tail = new BiNode(b);
    tail.prev = this;
}
BiNodes

class BiNode contd.

```java
public void addend(BandCard b) {
    tail = new BiNode(b);
    tail.prev = this;
}
```
This last method sets the current node to be the last in the list. I.e., it ‘removes’ (pointers to) any following nodes. The point of this will be clear when we see method `removeLast()` in class `LList`.

```java
public void chop() {
    tail = null;
}
```
This last method sets the current node to be the last in the list. I.e., it ‘removes’ (pointers to) any following nodes. The point of this will be clear when we see method `removeLast()` in class `LList`. 

```java
public void chop() {
    tail = null;
}
```
Doubly-linked Lists

Class **LList** looks exactly like class **List** (v1.1), except

- **BiNode** replaces **Node** everywhere
- we add the method **removeLast()**
Doubly-linked Lists

Class LList looks exactly like class List (v1.1), except

- BiNode replaces Node everywhere
- we add the method removeLast()
Removing the Last Element

removeLast pseudocode

if (the list is empty)
    do nothing
else if (there is only one element in the list)
    make the list empty
else (i.e., there are at least two cards in the list)
    get rid of the last one:
        move pointer back;
        set tail of second-last to null
Removing the Last Element

removeLast pseudocode

if (the list is empty)
   do nothing

else if (there is only one element in the list)
   make the list empty

else (i.e., there are at least two cards in the list)
   get rid of the last one:
      move pointer back;
      set tail of second-last to null
Removing the Last Element

removeLast pseudocode

if (the list is empty)
    do nothing

else if (there is only one element in the list)
    make the list empty

else (i.e., there are at least two cards in the list)
    get rid of the last one:
        move pointer back;
        set tail of second-last to null
Removing the Last Element

removeLast pseudocode

if (the list is empty)
    do nothing
else if (there is only one element in the list)
    make the list empty
else (i.e., there are at least two cards in the list)
    get rid of the last one:
        move pointer back;
        set tail of second-last to null
Removing the Last Element

removeLast pseudocode

if (the list is empty)
    do nothing
else if (there is only one element in the list)
    make the list empty
else (i.e., there are at least two cards in the list)
    get rid of the last one:
        move pointer back;
        set tail of second-last to null
Removing the Last Element

```java
in class LList

public void removeLast() {
    if (listEnd == null) { // list is empty...
        return; // ...do nothing
    }

    // if we’re here, the list is not empty
    if (listEnd.prev() == null) { // only one node
        listStart = null; // make the list empty
        listEnd = null;
    } else { // more than one node in list
        listEnd = listEnd.prev(); // move pointer
        listEnd.chop(); // cut end of list
    }
}
```
Removing the Last Element

in class LList

```java
public void removeLast() {
    if (listEnd == null) { // list is empty...
        return; // ...do nothing
    }

    // if we’re here, the list is not empty
    if (listEnd.prev() == null) { // only one node
        listStart = null; // make the list empty
        listEnd = null;
    } else { // more than one node in list
        listEnd = listEnd.prev(); // move pointer
        listEnd.chop(); // cut end of list
    }
}
```
Removing the Last Element

in class LList

```java
public void removeLast() {
    if (listEnd == null) { // list is empty...
        return; // ...do nothing
    }
    // if we’re here, the list is not empty
    if (listEnd.prev() == null) { // only one node
        listStart = null; // make the list empty
        listEnd = null;
    } else { // more than one node in list
        listEnd = listEnd.prev(); // move pointer
        listEnd.chop(); // cut end of list
    }
}
```
Removing the Last Element

in class LList

public void removeLast() {
    if (listEnd == null) { // list is empty...
        return; // ...do nothing
    }

    // if we're here, the list is not empty
    if (listEnd.prev() == null) { // only one node
        listStart = null; // make the list empty
        listEnd = null;
    } else { // more than one node in list
        listEnd = listEnd.prev(); // move pointer
        listEnd.chop(); // cut end of list
    }
}
Removing the Last Element

```java
public void removeLast() {
    if (listEnd == null) { // list is empty...
        return; // ...do nothing
    }
    // if we’re here, the list is not empty
    if (listEnd.prev() == null) { // only one node
        listStart = null; // make the list empty
        listEnd = null;
    } else { // more than one node in list
        listEnd = listEnd.prev(); // move pointer
        listEnd.chop(); // cut end of list
    }
}
```
Removing the Last Element

in class LList

```java
public void removeLast() {
    if (listEnd == null) { // list is empty...
        return; // ...do nothing
    }
    // if we're here, the list is not empty
    if (listEnd.prev() == null) { // only one node
        listStart = null; // make the list empty
        listEnd = null;
    } else { // more than one node in list
        listEnd = listEnd.prev(); // move pointer
        listEnd.chop(); // cut end of list
    }
}
```
Removing the Last Element

in class LList

```java
public void removeLast() {
    if (listEnd == null) { // list is empty...
        return; // ...do nothing
    }

    // if we’re here, the list is not empty
    if (listEnd.prev() == null) { // only one node
        listStart = null; // make the list empty
        listEnd = null;
    } else { // more than one node in list
        listEnd = listEnd.prev(); // move pointer
        listEnd.chop(); // cut end of list
    }
}
```
Removing the Last Element

in class LList

public void removeLast() {
    if (listEnd == null) { // list is empty...
        return;           // ...do nothing
    }
    // if we’re here, the list is not empty
    if (listEnd.prev() == null) { // only one node
        listStart = null; // make the list empty
        listEnd = null;
    } else { // more than one node in list
        listEnd = listEnd.prev(); // move pointer
        listEnd.chop(); // cut end of list
    }
}
Finding Stuff in the Middle

Suppose we want to work with nodes in the middle of a list e.g., finding nodes, adding/removing nodes . . .

The standard practice is to traverse the list (loop) until we find the node/position we want, then do stuff (add/remove, etc., which usually involves setting the pointers appropriately)

As an example, we’ll look at a method to find a node
Finding Stuff in the Middle

Suppose we want to work with nodes in the middle of a list e.g., finding nodes, adding/removing nodes . . .

The standard practice is to traverse the list (loop) until we find the node/position we want, then do stuff (add/remove, etc., which usually involves setting the pointers appropriately)

As an example, we’ll look at a method to find a node
Finding Stuff in the Middle

Suppose we want to work with nodes in the middle of a list e.g., finding nodes, adding/removing nodes . . .

The standard practice is to traverse the list (loop) until we find the node/position we want, then do stuff (add/remove, etc., which usually involves setting the pointers appropriately)

As an example, we’ll look at a method to find a node
Finding Stuff in the Middle

Suppose we want to work with nodes in the middle of a list e.g., finding nodes, adding/removing nodes . . .

The standard practice is to traverse the list (loop) until we find the node/position we want, then do stuff (add/remove, etc., which usually involves setting the pointers appropriately)

As an example, we’ll look at a method to find a node
in List

fmod IS_IN is

  protecting LIST .

  op isIn : String List -> Bool .

  var B : BandCard .

  var S : String .

  var L : List .

  eq isIn(S, empty) = false .

  eq isIn(S, add(B, L)) = (getName(B) == S) or isIn(S,L) .

endfm
in List

fmod IS_IN is

  protecting LIST.

  op isIn : String List -> Bool.

  var B : BandCard.
  var S : String.
  var L : List.

  eq isIn(S, empty) = false.
  eq isIn(S, add(B, L)) = (getName(B) == S) or isIn(S,L).

endfm
in List

fmod IS_IN is

  protecting LIST .

  op isn : String List -> Bool .

  var B : BandCard .
  var S : String .
  var L : List .

  eq isn(S, empty) = false .
  eq isn(S, add(B, L)) = (getName(B) == S) or isn(S,L) .

endfm
in List

fmod IS_IN is

  protecting LIST .

  op isIn : String List -> Bool .

  var B : BandCard .
  var S : String .
  var L : List .

  eq isIn(S, empty) = false .
  eq isIn(S, add(B, L)) = (getName(B) == S) or isIn(S,L) .

endfm
Finding a Node

pseudocode

if (this is what we’re looking for)
do something
else
move on
Finding a Node

```
if (this is what we’re looking for)
do something
else
  move on
```
Finding a Node

pseudocode

if (this is what we’re looking for)
do something
else
move on
Finding a Node

pseudocode

if (this is what we’re looking for)
do something
else
move on
Finding a Node

pseudocode

if (this is what we’re looking for)
do something
else
move on
More Pseudocode

**pseudocode**

```plaintext
if (list is empty)
  return false
else
  either:
    return true, or
  look at next node
```

**Maude spec**

```plaintext
eq isIn(S, empty) = false .
```

**Maude spec**

```plaintext
eq isIn(S, add(B, L)) = (getName(B) == S) or isIn(S,L) .
```
More Pseudocode

pseudocode

if (list is empty)
  return false
else
  either:
    return true, or
  look at next node

Maude spec

eq isIn(S, empty) = false .

Maude spec

eq isIn(S, add(B, L)) =
  (getName(B) == S) or isIn(S,L) .
pseudocode

if (list is empty)
  return false

else
  either:
    return true, or
  look at next node

Maude spec

\[ \text{eq} \quad \text{isIn}(S, \text{empty}) = \text{false} . \]

Maude spec

\[ \text{eq} \quad \text{isIn}(S, \text{add}(B, L)) = \text{(getName}(B) == S) \text{ or } \text{isIn}(S,L) . \]
More Pseudocode

**Pseudocode**

```plaintext
if (list is empty)  
    return false  
else  
    either:  
        return true, or 
    look at next node
```

**Maude spec**

```plaintext
eq isIn(S, empty) = false .
```

**Maude spec**

```plaintext
eq isIn(S, add(B, L)) = (getName(B) == S) or isIn(S, L) .
```
More Pseudocode

**pseudocode**

```plaintext
if (list is empty)
    return false
else
    either:
        return true, or
    look at next node
```

**Maude spec**

```plaintext
eq isIn(S, empty) = false .
```

**Maude spec**

```plaintext
eq isIn(S, add(B, L)) = (getName(B) == S) or isIn(S, L) .
```
if (list is empty)
    return false
else
    either:
    return true, or
    look at next node

Maude spec

eq isIn(S, empty) = false .

Maude spec

eq isIn(S, add(B, L)) = (getName(B) == S) or isIn(S,L) .
public boolean isIn(String s) {
    if (listStart == null) {
        return false;
    } else {
        BiNode n = listStart;
        while (n != null) { // not at end
            if (n.head().getName().equals(s)) {
                return true; // found it
            }
            n = n.tail();
        }
        return false; // end of list; not found
    }
}
public boolean isIn(String s) {
    if (listStart == null) {
        return false;
    } else {
        BiNode n = listStart;
        while (n != null) { // not at end
            if (n.head().getName().equals(s)) {
                return true; // found it
            }
            n = n.tail();
        }
        return false; // end of list; not found
    }
}
public boolean isIn(String s) {
    if (listStart == null) {
        return false;
    } else {
        BiNode n = listStart;
        while (n != null) { // not at end
            if (n.head().getName().equals(s)) {
                return true; // found it
            }
            n = n.tail();
        } // not at end
        return false; // end of list; not found
    }
}
public boolean isIn(String s) {
    if (listStart == null) {
        return false;
    } else {
        BiNode n = listStart;
        while (n != null) { // not at end
            if (n.head().getName().equals(s)) {
                return true; // found it
            }
            n = n.tail();
        }
        return false; // end of list; not found
    }
}
public boolean isIn(String s) {
    if (listStart == null) {
        return false;
    } else {
        BiNode n = listStart;
        while (n != null) {  // not at end
            if (n.head().getName().equals(s)) {  // found it
                return true;
            }
            n = n.tail();
        }
        return false;  // end of list; not found
    }
}
public boolean isIn(String s) {
    if (listStart == null) {
        return false;
    } else {
        BiNode n = listStart;
        while (n != null) { // not at end
            if (n.head().getName().equals(s)) { // found it
                return true;
            }
            n = n.tail();
        }
        return false; // end of list; not found
    }
}
Arrays, Linked Lists . . .

**Arrays** are good for
- storing fixed quantities
- accessing anywhere (through indices or pointers)

**Arrays** are bad for
- number of things stored changes a lot
- removing/adding things (shuffle, shuffle)

**Linked Lists** are good for
- storing changing quantities of things
- accessing/adding at ends

**Linked Lists** are bad for
- memory usage (lots of nodes)
- accessing at arbitrary points (loops)
Arrays, Linked Lists . . .

**Arrays** are good for
- storing fixed quantities
- accessing anywhere (through indices or pointers)

**Arrays** are bad for
- number of things stored changes a lot
- removing/adding things (shuffle, shuffle)

**Linked Lists** are good for
- storing changing quantities of things
- accessing/adding at ends

**Linked Lists** are bad for
- memory usage (lots of nodes)
- accessing at arbitrary points (loops)
Arrays, Linked Lists . . .

Arrays are good for
- storing fixed quantities
- accessing anywhere (through indices or pointers)

Arrays are bad for
- number of things stored changes a lot
- removing/adding things (shuffle, shuffle)

Linked Lists are good for
- storing changing quantities of things
- accessing/adding at ends

Linked Lists are bad for
- memory usage (lots of nodes)
- accessing at arbitrary points (loops)
Arrays, Linked Lists . . .

**Arrays** are good for
- storing fixed quantities
- accessing anywhere (through indices or pointers)

**Arrays** are bad for
- number of things stored changes a lot
- removing/adding things (shuffle, shuffle)

**Linked Lists** are good for
- storing changing quantities of things
- accessing/adding at ends

**Linked Lists** are bad for
- memory usage (lots of nodes)
- accessing at arbitrary points (loops)
That’s All, Folks!

Summary

- (doubly) linked lists
- loops

Next:

Off-the-shelf classes