Lists

Checking It Twice

Friday, October 26, 2007

CS111 Computer Programming

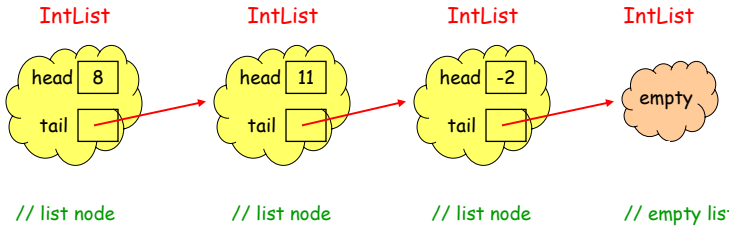
Department of Computer Science
Wellesley College

IntLists represent lists of integers

IntLists are defined recursively:

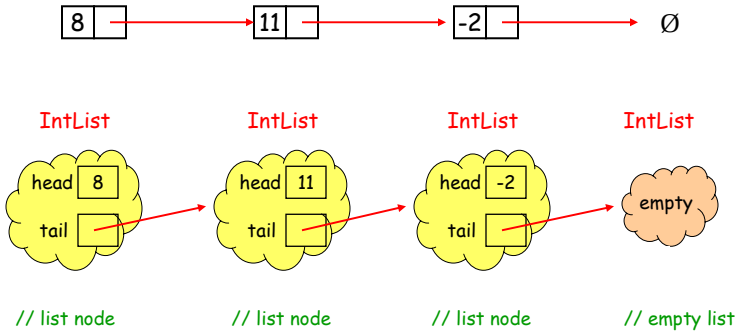
An IntList is either

- the empty list, or
- a (non-empty) list node with two parts:
  1. a head which is an integer, and
  2. a tail which is another IntList.



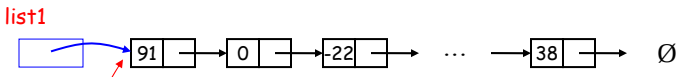
Box-and-pointer notation

We represent list nodes and empty lists using a shorthand known as box-and-pointer notation.

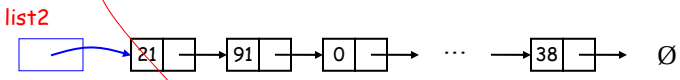


Why are lists useful?

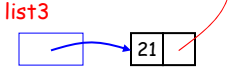
Lists are good for storing many numbers (but you don't know how many in advance).



Lists are also very flexible. E.g., here is a new list which starts with 21, and then the rest are the same as before.



We could create list2 from scratch, or we could use the first list as the tail of a new list.



Five core IntList methods

- Methods in the (Java defined) Math class are static; you invoke them on the Math class:

Class name → `Math` ; Class method → `abs(y)`

```
int x = Math.abs(y);
```

- Similarly all IntList methods are class methods. You invoke them on the IntList class.*

*Not an object.

`public static int head (IntList L)`
Returns the integer that is the head component of the integer list node **L**.
Signals an exception if **L** is empty.

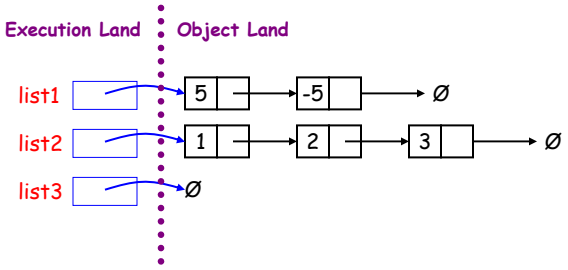
`public static IntList tail (IntList L)`
Returns the integer list that is the tail component of the integer list node **L**.
Signals an exception if **L** is empty.

`public static boolean isEmpty (IntList L)`
Returns **true** if **L** is an empty integer list and **false** if **L** is an integer list node.

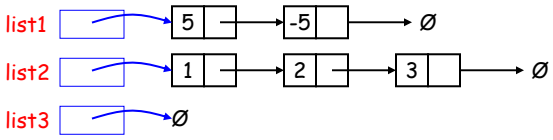
`public static IntList empty()`
Returns an empty integer list.

`public static IntList prepend (int n, IntList L)`
Returns a new integer list node whose head is **n** and whose tail is **L**.

Our working example



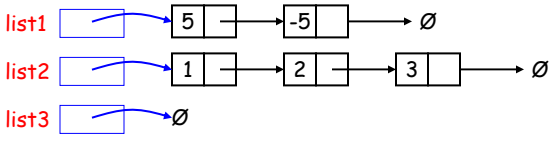
isEmpty method




```
IntList.isEmpty(list1)
IntList.isEmpty(list2)
IntList.isEmpty(list3)
```

`public static boolean isEmpty (IntList L)`
Returns **true** if **L** is an empty integer list and **false** if **L** is an integer list node.

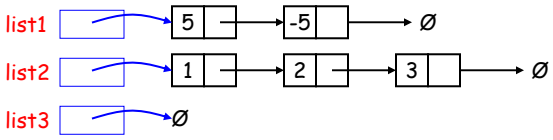
head method




```
IntList.head(list1)
IntList.head(list2)
IntList.head(list3)
```

`public static int head (IntList L)`
Returns the integer that is the head component of the integer list node **L**.
Signals an exception if **L** is empty.

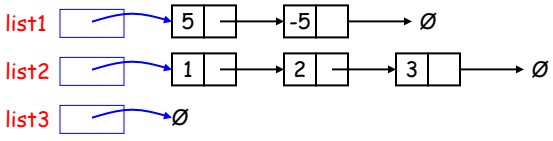
tail method




```
IntList.tail(list1)
IntList.tail(list2)
IntList.tail(list3)
```

`public static IntList tail (IntList L)`
Returns the integer list that is the tail component of the integer list node **L**.
Signals an exception if **L** is empty.

heads and tails




```
IntList.head(IntList.tail(list2))
IntList.head(IntList.tail(IntList.tail(list2)))
```

Constructing IntLists

Two approaches for constructing IntLists:

Using constuctor methods

`public IntList()`
Constructs an empty integer list.

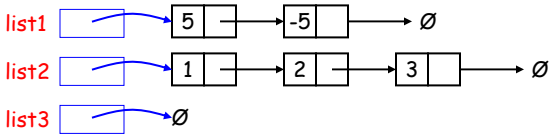
`public IntList(int head, IntList tail)`
Constructs an integer list whose head is **`head`** and whose tail is **`tail`**.

Using `empty()` and `prepend()`

`public static IntList empty()`
Returns an empty integer list.

`public static IntList prepend (int n, IntList L)`
Returns a new integer list node whose head is **`n`** and whose tail is **`L`**.

Constructing IntLists: examples



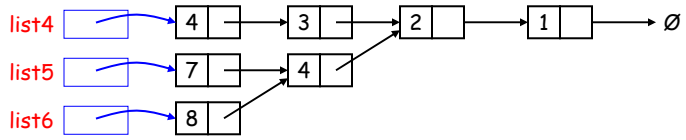
Using constuctor methods

```
IntList list1 = new IntList(5, new IntList(-5, new IntList()));
IntList list2 = new IntList(1, new IntList(2, new IntList(3, new IntList())));
IntList list3 = new IntList();
```

Using `empty()` and `prepend()`

```
IntList list1 = IntList.prepend(5, IntList.prepend(-5, IntList.empty()));
IntList list2 = IntList.prepend(1,
                        IntList.prepend(2, IntList.prepend(3, IntList.empty())));
IntList list3 = IntList.empty();
```

Sharing list nodes




```
IntList list4 = IntList.prepend(4, IntList.prepend(3,
                        IntList.prepend(2, IntList.prepend(1, IntList.empty()))));

IntList list5 = IntList.prepend(7, IntList.prepend(4,
                                              IntList.tail(IntList.tail(list4))));

IntList list6 = IntList.prepend(8, IntList.tail(list5));
```


Recursive list methods

```
// Returns the integer sum of all elements in L
public static int sumList (IntList L) {




}
```

Recursive list methods

```
// Returns the number of elements in list L
public static int length (IntList L) {




}
```


Recursive list methods

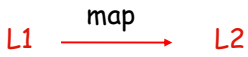
```
// Returns the maximum number in list L
public static int maxList (IntList L) {




}
```

Recursive list methods

```
// Returns true if all the elements in L are positive
// and returns false otherwise
public static boolean areAllPositive (IntList L) {









}
```


Mapping methods

A *mapping* method applies some operation to each element in a list.

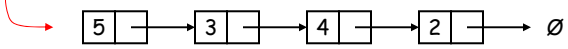


More formally: A *mapping* method generates a new list by applying some function to each element in the original list.

Squaring numbers

- Method mapSquares(IntList L) returns a list, such that each element in the list is the "square" of the corresponding element in "L".
- For example,

```
mapSquares(L)
```



mapSquares(IntList L);

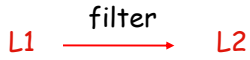
```
// Returns a new list whose elements are the sqaures of
// the corresponding elements in the given list
public static IntList mapSquares(IntList L) {

   if (                    ) {  // base case
       return
   } else {                     // recursive case
       return

   }
}
```

mapSquares(IntList L);

```
// Returns a new list whose elements are the sqaures of
// the corresponding elements in the given list
public static IntList mapSquares(IntList L) {

   if ( InList.isEmpty(L) )  {  // base case
       return empty();
   } else {                     // recursive case
       return IntList.prepend(IntList.head(L)*IntList.head(L),
                              mapSquares(IntList.tail(L)));
   }
}
```


Filter methods

A *filter* method passes some elements through and holds other elements back.

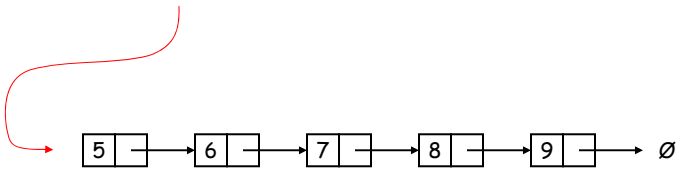


More formally: A *filter* method transforms one list to another by keeping only those elements satisfying a given predicate.

Filtering even numbers

- Method filterEven(IntList L) returns a list containing elements of "L" that are even (i.e., evenly divisable by 2).
- For example,

```
filterEven(L)
```



filterEven(IntList L):

```
// Returns sublist containing elements of L that are even
public static IntList filterEven(IntList L) {

   if (                        )  {              // base case
       return
   } else if (                                ) {  // recursive case
       return

   } else {
       return
  }
}
```

filterEven(IntList L);

```
// Returns sublist containing elements of L that are even
public static IntList filterEven(IntList L) {

   if (   IntList.isEmpty(L)   )  {              // base case
       return IntList.empty();
   } else if (  (IntList.head(L) % 2) == 0  ) {  // recursive case
       return IntList.preprend(IntList.head(L),
                               filterEven(IntList.tail(L)));
   } else {
       return filterEven(IntList.tail(L));
  }
}
```