Nested Structures

Arrays of arrays, lists of lists

Friday, November 17, 2006

CS111 Computer Programming

Department of Computer Science
Wellesley College

Nested Structures

In general, an object can have other objects as components (i.e., nested within it). Examples:

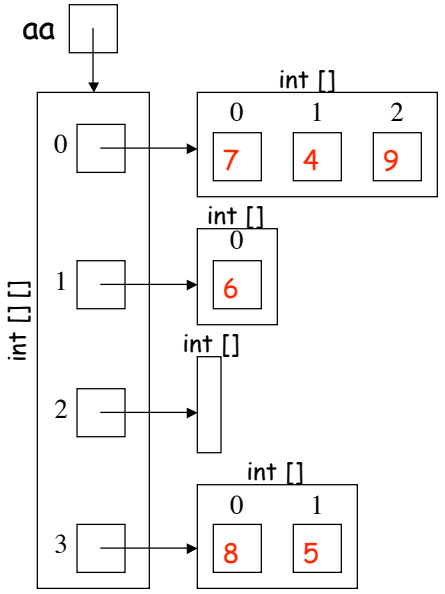
o Buggles have Point, Direction, and Color objects as components.
o StringList nodes have a String and a StringList as components.

Objects can be nested in arbitrary ways. E.g., can have:

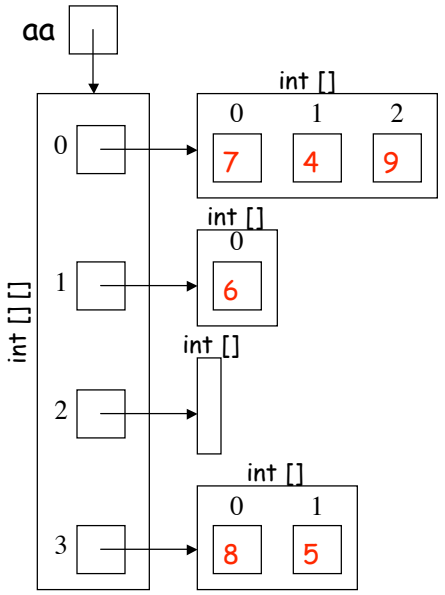
o Arrays of arrays, arrays of arrays of arrays, …
o Lists of list, lists of lists of lists, …
o Arrays of lists, arrays of arrays of lists, arrays of lists of arrays, …
o Lists of arrays, lists of lists of arrays, lists of arrays of lists, …

Can use these to represent regular collections (e.g., colors, bagels, walls in BuggleWorld grid) and irregular collections (e.g., number of candies per trick-or-treater at houses in neighborhood).

An Array of Arrays



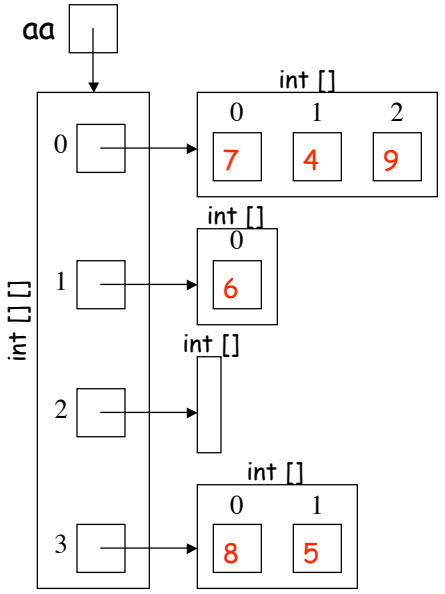
An Array of Arrays: Manipulation



What do the following statements do to the array diagram?

```
aa[0][1] = aa[1][0];
aa[2] = aa[0];
aa[0][2] = aa[2][0];
aa[3][1] = aa[1][0] + aa[2][2];
```

An Array of Arrays: Creation



How can we create this array?

Way 1:

```
int [] [] = {{7,4,9},{6},{},{8,5}}
```

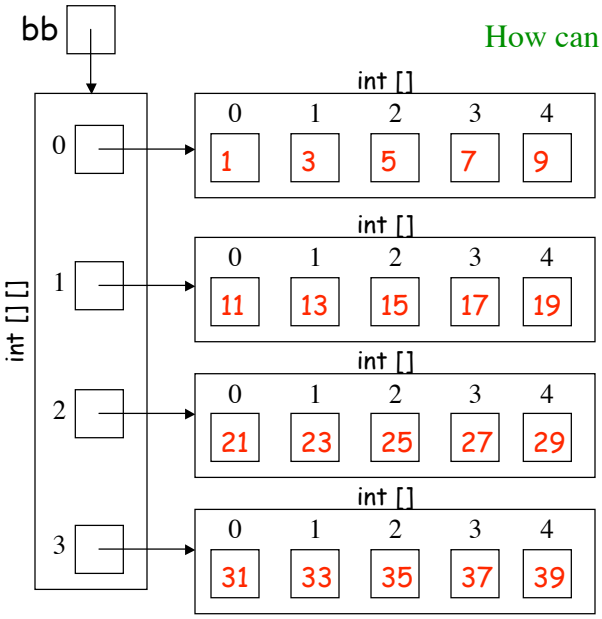
Way 2:

```
int [] a0 = {7,4,9};
int [] a1 = {6};
int [] a2 = {};
int [] a3 = {8,5};
int [] [] aa = new int [4] [];
aa[0] = a0;
aa[1] = a1;
aa[2] = a2;
aa[3] = a3;
```


An Array of Arrays: Summation

```
// Sum all the integers in an array of array of ints
public static int sumArrayofArrays (int [] [] aai) {
  int sum = 0;
  for (int outer = 0; outer < aai.length; outer++) {
    for (int inner = 0; int < aai[outer].length; inner++) {
      sum = sum + aai[outer][inner];
    }
  }
  return sum;
}
```

Another Array of Arrays



How can you create this array?

Ans:

```
// Create array of
// array of zeroes:
int[][] bb = new int[4][5];

// Initialize slots:
for (int i=0; i<4; i++) {
  for (int j=0; j<5; j++) {
    bb[i][j] = (10*i)+(2*j)+1;
  }
}
```


NOTE: WE DID NOT COVER LISTS OF LISTS IN LECTURE THIS SEMESTER!

Review: What's a List?

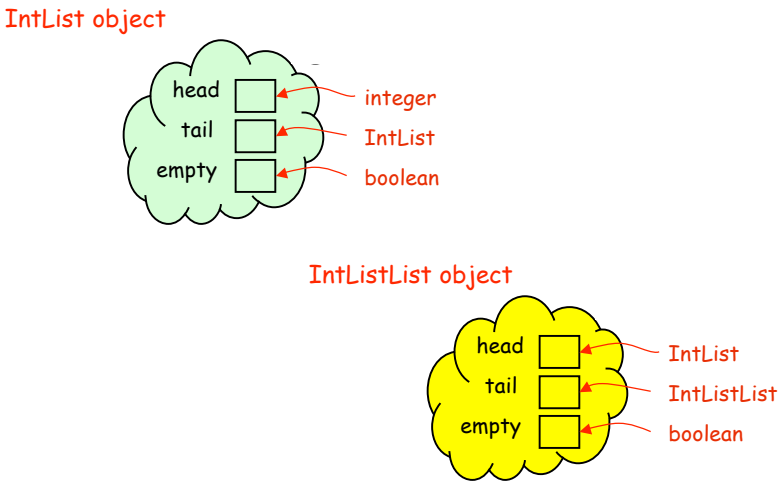
- The empty list (∅), or
- A list node that contains:
  - A *head* containing whatever data element we're storing in the list and
  - A *tail* which is a reference to the rest of the list.

Lists of lists

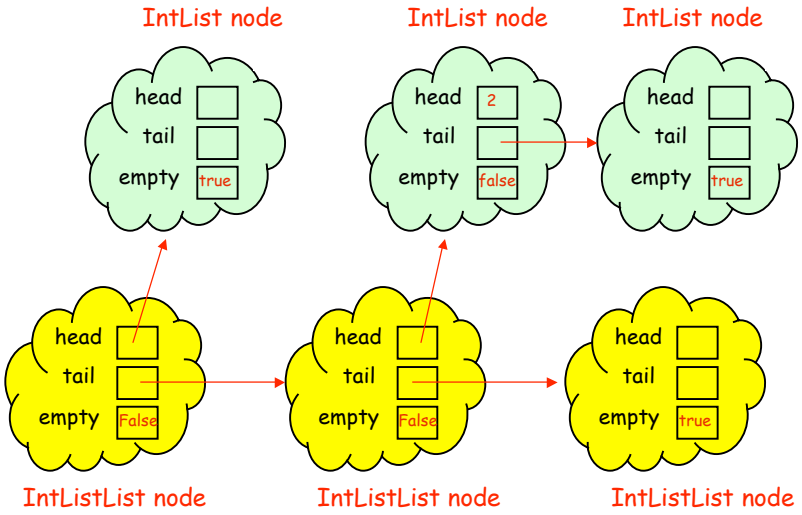
Sometimes we be need keep more than one list at a time
Multiple lists might organized as "lists of lists."

- List of tails of the list [7, 2, 4]:
  [[7, 2, 4], [2, 4], [4], []]
- Permutations of [1,2,3]:
  [[1,2,3], [2,1,3], [2,3,1], [1,3,2], [3,1,2], [3,2,1]]
- Powerset of [1,2,3] = list of all "subsets" of the "set" [1,2,3]:
  [[], [1], [2], [3], [1,2], [2,3], [1,3], [1,2,3]]

IntListList and IntLists have much in common



A short IntListList: [[],[2]]



Five core IntListList methods

`public static IntList head (IntListList L)`
Returns the integer list that is the head component of the list of integer lists node **`L`**.

`public static IntListList tail (IntListList L)`
Returns the list of integer lists that is the tail component of the list of integer lists node **`L`**.

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

`public static IntListList empty()`
Returns an empty list of integer lists.

`public static IntListList prepend (IntList x, IntListList L)`
Returns a new list of integer lists node whose head is **`x`** and whose tail is **`L`**.

Our working examples

list 1
[[], [1], [2], [3], [1,2], [2,3], [1,3], [1,2,3]]
list2
[]
list3
[[]]

IntListList isEmpty() method

list 1
[[], [1], [2], [3], [1,2], [2,3], [1,3], [1,2,3]]
list2
[]
list3
[[]]

```
IntListList.isEmpty(list1);
IntListList.isEmpty(list2);
IntListList.isEmpty(list3);
```

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

IntListList head() method

list 1
[[], [1], [2], [3], [1,2], [2,3], [1,3], [1,2,3]]
list2
[]
list3
[[]]

```
IntListList.head(list1);
IntListList.head(list2);
IntListList.head(list3);
```

`public static IntList head (IntListList L)`
Returns the integer list that is the head component of the list of integer lists node `L`.

IntListList tail() method

list 1
[[], [1], [2], [3], [1,2], [2,3], [1,3], [1,2,3]]
list2
[]
list3
[[]]

```
IntListList.tail(list1);
IntListList.tail(list2);
IntListList.tail(list3);
```

`public static IntListList tail (IntListList L)`
Returns the list of integer lists that is the tail component of the list of integer lists node `L`.

IntListList prepend() method

list 1
[[], [1], [2], [3], [1,2], [2,3], [1,3], [1,2,3]]
list2
[]
list3
[[]]

```
IntListList.prepend(IntList.fromString("[7,2,4]"), list1);
IntListList.prepend(Intlist.empty(), list2);
IntListList.prepend(IntList.fromString("[3,2]"), list3);
```

```
public static IntListList prepend (IntList x, IntListList L)
```
Returns a new list of integer lists node whose head is **x** and whose tail is **L**.

An IntListList generator

- The method `tails(IntList L)` returns the list of tails of the IntList L.
- For example,

```
tails(IntList.fromString("[7,2,4]"));
```

returns

[[7, 2, 4], [2, 4], [4]. []]

Parameter and return type?

```
// Returns a list of the successive tails of L

public static ___________ tails ( _____ L)
{
   if (             )  {
                                        // base case


   } else {
                                        // recursive case

   }
}
```


Parameter and return type?

```
// Returns a list of the successive tails of L

public static IntListList tails ( IntList L)
{
   if (             )  {
                                        // base case


   } else {
                                        // recursive case

   }
}
```

tails()

```
// Returns a list of the successive tails of L

public static IntListList tails (IntList L)
{
   if (IntList.isEmpty(L)) {
        // tricky!
        return IntListList.prepend(IntList.empty(),
                                   IntListList.empty());
   } else {
        return IntListList.prepend(L,
                                   tails(IntList.tail(L)));
   }
}
```


appendAll() glues IntListLists back into IntLists

- For example,

```
IntList L1 = IntList.fromString("[7,2,4]");
appendAll(tails(L1));
```

returns

[7, 2, 4, 2, 4, 4]

* tails(L1) → [[7,2,4], [2,4], [4], []]

appendAll()

```
// Returns the IntList that results from appending
// all the element lists in the input together
 public static IntList appendAll (IntListList LL)
 {
     if (IntListList.isEmpty(LL))
          return IntList.empty();
     else
          return IntList.append(IntListList.head(LL),
                                appendAll(IntListList.tail(LL)));
 }
```


Another example*

```
// Returns an IntList the same length as the given IntListList
// in which each integer element in the output is the sum of
// the elements in the corresponding integer list element in
// the input.

public static IntList mapSumList (IntListList LL)
{

}
```

*If LL is our old friend, [[7,2,4], [2,4], [4], []] , what is returned?

mapSumList()

```
// Returns an IntList the same length as the given IntListList
// in which each integer element in the output is the sum of
// the elements in the corresponding integer list element in
// the input.

public static IntList mapSumList (IntListList LL)
{
     if (IntListList.isEmpty(LL))
          return IntList.empty();
     else
          return IntList.prepend(
                        IntList.sumList(IntListList.head(LL)),
                         mapSumList(IntListList.tail(LL)));
 }
```


Yet another example

```
// Returns an IntListList the same length as the given IntListList
// in which the specified integer has been prepended to every
// element in the input list
public static IntListList mapPrepend (int n, IntListList ill)
{

}
```

mapPrepend()

```
// Returns an IntListList the same length as the given IntListList
// in which the specified integer has been prepended to every
// element in the input list
public static IntListList mapPrepend (int n, IntListList ill)
{
   if (IntListList.isEmpty(ill)) return ill;

   return IntListList.prepend(
             IntList.prepend(n, IntListList.head(ill),
                             mapPrepend(n, IntListList.tail(ill)));
}
```


Making Change

Amount	Denoms	Ways to make change
5	[5,1]	[[5], // 1 nickel [1,1,1,1,1] // 5 pennies]
16	[10,5,1]	[[10,5,1], // dime, nickel, penny [10,1,1,1,1,1,1], // dime, 6 pennies [5,5,5,1], // 3 nickels, 1 penny [5,5,1,1,1,1,1,1], [5,1,1,1,1,1,1,1,1,1,1,1,1], [1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1]]

Break the Problem into Subproblems

o Use the largest coin denomination if possible. New problem: smaller amount, same list of denominations.
o Do not use the largest coin denomination. New problem: same amount, smaller list of denominations.

Base cases are tricky!

General Case

Compute partial result: all the ways to make change for the same amount with all but the largest coin.

If the amount is smaller than largest coin, this is the result.

Otherwise, find all the ways to make change for (amount - largest coin), prepend the largest coin to each of those ways, and return this result combined with the above partial result.

Base Cases

If the amount is 0, …

If the denomination list is empty, …