Pointers

Discussion 3

CS 61B // Spring 2022

Announcements

- HW 0, Lab 1, and Lab 2 due 1/31
- HW 1 due 2/1
- Weekly Surveys are worth points + due every Monday
- Topical Review Session on Java this Friday 2-3:30 PM

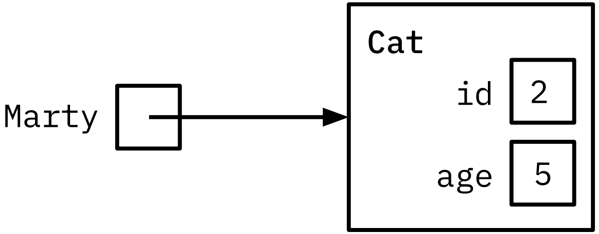
Review

Values & Containers

Simple Containers are named and may contain values or pointers to structured containers.
Structured Containers are anonymous and contain simple containers or objects.

Values are numbers, booleans, and pointers and cannot be *modified* without being *replaced*.

- Numbers → Numbers as we know them (`byte`, `short`, `int`, `double`, `long`, `float`)
- Letters → Characters (`char`)
- Booleans → True or False (`bool`)
- Pointers → Memory address to a spot in memory where a structured container is stored
- Null → Nothing

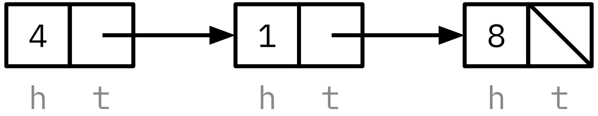


Linked Lists & Arrays

Linked Lists are data structures that consist of structured containers, each containing two simple containers.

`list.head` holds a value

`list.tail` stores a pointer to the next structured container



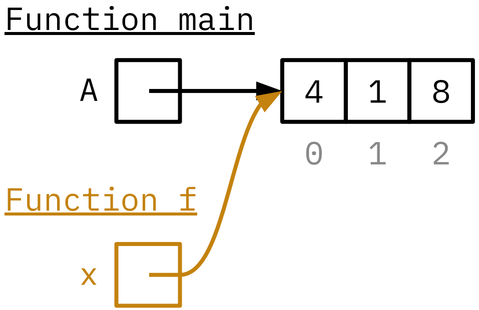
Arrays are data structures which can hold many simple containers of the same type of value.

`arr[i]` holds a value in the ith position of the array

4 1 8
0 1 2
c a t
0 1 2

Destructive & Non-Destructive Operations

Java is **pass-by-value**, so you are passing in a copy of the value of the variable.



```
private static void f(int[] x){ ... }

f(A)
```

Destructive functions alter the structured container or object passed in, causing changes to remain even after we leave the function (i.e. `x[1] = 5`)

Non-Destructive functions don't alter the structured contained passed in (i.e. `x = new int[]{5, 10}`)

CS 61B Pointers
Spring 2022 Exam Prep Discussion 3: January 31, 2022

1 Fill Grid

Given two one-dimensional arrays LL and UR, fill in the program on the next page to insert the elements of LL into the lower-left triangle of a square two-dimensional array S and UR into the upper-right triangle of S, without modifying elements along the main diagonal of S. You can assume LL and UR both contain at least enough elements to fill their respective triangles. (Spring 2020 MT1)

For example, consider

```
int[] LL = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 0, 0 };
int[] UR = { 11, 12, 13, 14, 15, 16, 17, 18, 19, 20 };
int[][] S = {
    { 0, 0, 0, 0, 0},
    { 0, 0, 0, 0, 0},
    { 0, 0, 0, 0, 0},
    { 0, 0, 0, 0, 0},
    { 0, 0, 0, 0, 0}
};
```

After calling fillGrid(LL, UR, S), S should contain

```
{
  { 0, 11, 12, 13, 14 },
  { 1,  0, 15, 16, 17 },
  { 2,  3,  0, 18, 19 },
  { 4,  5,  6,  0, 20 },
  { 7,  8,  9, 10,  0 }
}
```

(The last two elements of LL are excess and therefore ignored.)

```
/** Fill the lower-left triangle of S with elements of LL and the
 *  upper-right triangle of S with elements of UR (from left-to
 *  right, top-to-bottom in each case). Assumes that S is square and
 *  LL and UR have at least sufficient elements. */
public static void fillGrid(int[] LL, int[] UR, int[][] S) {
    int N = S.length;
    int kL, kR;
    kL = kR = 0;

    for (int i = 0; i < N; i += 1) {

        for (int j=0; j<N; j+=1) {

            if (i<j) {  ← Defines UR

                S[i][j] = UR[kR];

                kR += 1;

            } else if (i>j) {  ← Defines LL

                S[i][j] = LL[kL];

                kL += 1;

            }

        ____________________________________________
    }
}
```

```
int[] newArr = new int[N];
newArr[i] = S[i][i];
for (int j=0; j<i; j+=1) {
    newArr[j] = LL[kL];
    kL++;
}
for (int k=i+1; k<N; k+=1) {
    newArr[k] = UR[kR];
    kR++;
}
S[i] = newArr;
```

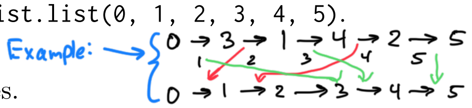
↑

This solution is too long, so use System. arraycopy to copy directly to S[i] and meet line requirements instead of the 2 for loops

2 Even Odd

Implement the method `evenOdd` by *destructively* changing the ordering of a given `IntList` so that even indexed links **precede** odd indexed links.

For instance, if `lst` is defined as `IntList.list(0, 3, 1, 4, 2, 5)`, `evenOdd(lst)` would modify `lst` to be `IntList.list(0, 1, 2, 3, 4, 5)`.

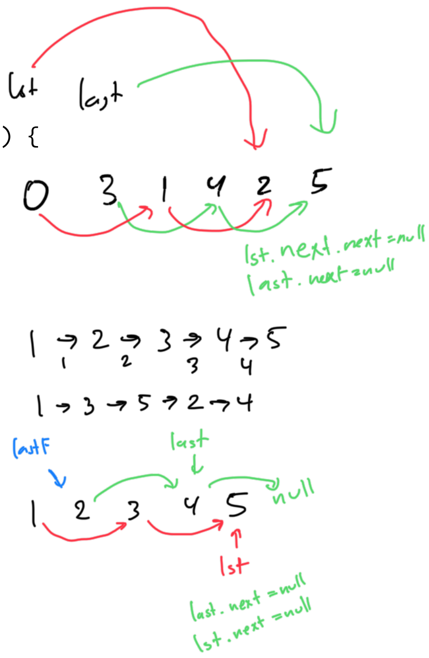


You may not need all the lines.

Hint: Make sure your solution works for lists of odd and even lengths.

```
public class IntList {
    public int first;
    public IntList rest;
    public IntList (int f, IntList r) {
        this.first = f;
        this.rest = r;
    }

    public static void evenOdd(IntList lst) {

      if (lst == null) {
          return;
      }

      IntList last = lst.rest;

      IntList lastFixed = last;

      while (lst.rest != null && lst.rest.rest != null) {

          lst.rest = lst.rest.rest;

          lst = lst.rest;

          last.rest = lst.rest;

          last = last.rest;
      }

      lst.rest = last Fixed;
    }
}
```

↳ Could also check whether length of list requires processing



3 Partition

Implement `partition`, which takes in an `IntList lst` and an integer `k`, and *destructively* partitions `lst` into `k IntLists` such that each list has the following properties:

1. It is the **same** length as the other lists. If this is not possible, i.e. `lst` cannot be equally partitioned, then the later lists should be **one** element smaller. For example, partitioning an `IntList` of length 25 with `k = 3` would result in partitioned lists of lengths 9, 8, and 8.
2. Its ordering is consistent with the ordering of `lst`, i.e. items in earlier in `lst` must **precede** items that are later.

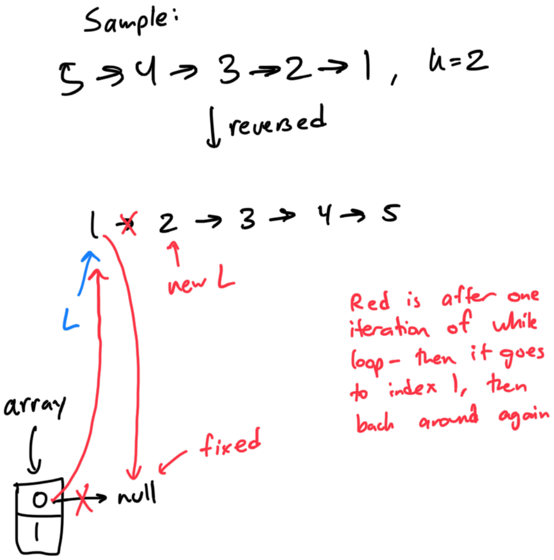
These lists should be put in an array of length `k`, and this array should be returned. For instance, if `lst` contains the elements 5, 4, 3, 2, 1, and `k = 2`, then a **possible** partition (note that there are many possible partitions), is putting elements 5, 3, 2 at index 0, and elements 4, 1 at index 1.

You may assume you have the access to the method `reverse`, which destructively reverses the ordering of a given `IntList` and returns a pointer to the reversed `IntList`. You may not create any `IntList` instances. You may not need all the lines.

Think of adjusting the necessary pointers

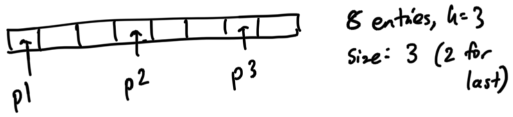
Hint: You may find the % operator helpful.

```
public static IntList[] partition(IntList lst, int k) {
    IntList[] array = new IntList[k];
    int index = 0;
    IntList L = reverse(lst);
    while (L != null) {

        IntList fixed = array[index];

        IntList newL = L.rest;

        array[index] = L;

        array[index].rest = fixed;

        L = newL;

        ____________________________________________

        index = (index+1) % k;
    }
    return array;
}
```


Sample:
5 → 4 → 3 → 2 → 1 , k=2
reversed
1 → 2 → 3 → 4 → 5
new L
L
array
fixed
0
1
null
Red is after one iteration of while loop - then it goes to index 1, then back around again

(or an IntList supporting indexing)

If input was an array, could theoretically build up these lists all at once by predetermining partition sizes and using multiple pointers:


p1
p2
p3
8 entries, k=3
Size: 3 (2 for last)