Lecture 6: ALists, Resizing, vs. SLists

9/9/2020

A Last Look at Linked Lists

Doubly Linked Lists

- Through various improvements, we made all of the following operations fast:
  - `addFirst, addLast`
  - `getFirst, getLast`
  - `removeFirst, removeLast`

Arbitrary Retrieval

- Suppose we added `get(int i)`, which returns the ith item from the list
- Why would `get` be slow for long lists compared to `getLast()`? For what inputs?
  - Have to scan to desired position. Slow for any `i` not near the sentinel node
  - How to fix this?
  - For now: We'll take a different tack: Using an array instead (no links!)

Naive Array Lists

Random Access in Arrays

- Retrieval from any position of an array is very fast
  - Independent of array size
  - 61C Preview: Ultra fast random access results from the fact that memory boxes are the same size (in bits)

Our Goal: AList.java

- Want to figure out how to build an array version of a list:
  - In lecture we'll only do back operations

```
public class AList {
    private int[] items;
    private int size;

    /** Creates an empty list. */
    public AList() {
        items = new int[100];
        size = 0;
    }

    /** Inserts X into the back of the list */
    public void addLast(int x) {
        items[size] = x;
```

```
        size = size + 1;
    }

    /** Returns the item from the back of the list. */
    public int getLast() {
        return items[size-1];
    }

    /** Gets the ith item from the List (0 is the front) */
    public int get(int i) {
        return items[i];
    }

    /** Returns the number of items in the list. */
    public int size() {
        return size;
    }
}
```

Naive AList Code

- AList Invariants:
  - The position of the next item to be inserted is always `size`
  - `size` is always the number of items in the AList
  - The last item in the list is always in position `size - 1`
- Let's now discuss delete operations

The Abstract vs. the Concrete

- When we `removeLast()`, which memory boxes need to change? To what?
  - User's mental model: {5, 3, 1, 7, 22, -1} -> {5, 3, 1, 7, 22}
- Actual truth:
  - We change the size

Deletion

- When we `removeLast()`, which memory boxes need to change? To what?
  - Only `size`!

```
public class AList {
    private int[] items;
    private int size;

    /** Creates an empty list. */
    public AList() {
        items = new int[100];
        size = 0;
    }

    /** Inserts X into the back of the list */
```

```
    public void addLast(int x) {
        items[size] = x;
        size = size + 1;
    }

    /** Returns the item from the back of the list. */
    public int getLast() {
        return items[size-1];
    }

    /** Gets the ith item from the List (0 is the front) */
    public int get(int i) {
        return items[i];
    }

    /** Returns the number of items in the list. */
    public int size() {
        return size;
    }

    /** Deletes item from back of the list and returns deleted item */
    public int removeLast() {
        int x = getLast();
        items[size-1] = 0; // Not necessary to preserve invariants -> not
necessary for correctness
        size = size - 1;
        return x;
    }
}
```

Resizing Arrays

The Mighty AList

- Key Idea: Use some subset of the entries of an array
- What happens if we insert more than 100 items in AList? What should we do about it?

Array Resizing

- When the array gets too full, e.g. addLast(11), just make a new array:
    - `int[] a = new int[size+1];`
    - `System.arraycopy()`
    - `a[size] = 11;`
    - `items = a; size += 1;`
- We call this process "resizing"

Implementation

- Let's implement the resizing capability

```
public class AList {
    private int[] items;
    private int size;

    /** Creates an empty list. */
    public AList() {
        items = new int[100];
        size = 0;
    }

    /** Resizes the underlying array to the target capacity */
    private void resize(int capacity) {
        int[] a = new int[capacity]
        System.arraycopy(items, 0, a, 0, size);
        items = a;
    }

    /** Inserts X into the back of the list */
    public void addLast(int x) {
        if (size == items.length) {
            resize(size + 1);
        }
        items[size] = x;
        size = size + 1;
    }

    /** Returns the item from the back of the list. */
    public int getLast() {
        return items[size-1];
    }

    /** Gets the ith item from the List (0 is the front) */
    public int get(int i) {
        return items[i];
    }

    /** Returns the number of items in the list. */
    public int size() {
        return size;
    }

    /** Deletes item from back of the list and returns deleted item */
    public int removeLast() {
        int x = getLast();
        items[size-1] = 0; // Not necessary to preserve invariants -> not
necessary for correctness
        size = size - 1;
        return x;
    }
}
```

Basic Resizing Analysis

Runtime and Space Usage Analysis

- Suppose we have a full array of size 100. If we can `addLast` two times, how many total array memory boxes will we need to create and fill?
    - Answer: 203

Array Resizing

- Resizing twice requires us to create and fill 203 total memory boxes
    - Most boxes at any one time is 203

Runtime and Space Usage Analysis

- Suppose we have a full array of size 100. If we call `addLast` until `size = 1000`, roughly how many total memory boxes will we need to create and fill?
    - Answer: 101 + 102 + ... + 1000 = approximately 500000

Resizing Slowness

- Inserting 100,000 items requires rought 5,000,000,000 new containers
    - Computers operate at the speed of GHz
    - No huge surprise that 100,000 items took seconds
- Our resizing for ALists is done in linear time

Making AList Fast

Fixing the Resizing Performance Bug

- How do we fix this?

```
public class AList {
    private int[] items;
    private int size;

    /** Creates an empty list. */
    public AList() {
        items = new int[100];
        size = 0;
    }

    /** Resizes the underlying array to the target capacity */
    private void resize(int capacity) {
        int[] a = new int[capacity]
        System.arraycopy(items, 0, a, 0, size);
        items = a;
    }

    /** Inserts X into the back of the list */
    public void addLast(int x) {
```

```
        if (size == items.length) {
            resize(size * 2); // A subtle fix!!!
        }
        items[size] = x;
        size = size + 1;
    }

    /** Returns the item from the back of the list. */
    public int getLast() {
        return items[size-1];
    }

    /** Gets the ith item from the List (0 is the front) */
    public int get(int i) {
        return items[i];
    }

    /** Returns the number of items in the list. */
    public int size() {
        return size;
    }

    /** Deletes item from back of the list and returns deleted item */
    public int removeLast() {
        int x = getLast();
        items[size-1] = 0; // Not necessary to preserve invariants -> not
necessary for correctness
        size = size - 1;
        return x;
    }
}
```

(Probably) Surprising Fact

- Geometric resizing is much faster: Just how much better will have to wait

```
    public void addLast(int x) {
        if (size == items.length) {
            resize(size * 2); // A subtle fix!!!
        }
        items[size] = x;
        size = size + 1;
    }
```

- This is how Python lists are implemented

Performance Problem #2

- Suppose we have a very rare situation occurs which causes us to:
    - Insert 1,000,000,000 items

  - Then remove 990,000,000 items
- Our data structure will handle this spike of evens as well as it could, but afterwards there is a problem

Memory Efficiency

- An AList should not only be efficient in time, but also efficient in space
  - Define the "usage ratio" R = size / items.length;
  - Typical solution: Half array size when R < 0.25
  - More details in a few weeks
- Later we will consider tradeoffs between time and space efficiency for a variety of algorithms and data structures

Generic AList

Theres a Problem

- Generic arrays are not allowed 😦(
- Here's our fix

```
public class AList<Item> {
    private Item[] items;
    private int size;

    /** Creates an empty list. */
    public AList() {
        items = (Item[]) new Object[100];
        size = 0;
    }

    /** Resizes the underlying array to the target capacity */
    private void resize(int capacity) {
        Item[] a = (Item[]) new Object[capacity]
        System.arraycopy(items, 0, a, 0, size);
        items = a;
    }

    /** Inserts X into the back of the list */
    public void addLast(int x) {
        if (size == items.length) {
            resize(size * 2); // A subtle fix!!!
        }
        items[size] = x;
        size = size + 1;
    }

    /** Returns the item from the back of the list. */
    public Item getLast() {
        return items[size-1];
    }

    /** Gets the ith item from the List (0 is the front) */
```

```
    public Item get(int i) {
        return items[i];
    }

    /** Returns the number of items in the list. */
    public int size() {
        return size;
    }

    /** Deletes item from back of the list and returns deleted item */
    public Item removeLast() {
        int x = getLast();
        items[size-1] = null;
        size = size - 1;
        return x;
    }
}
```

Generic ALists (similar to generic SLists)

- When creating an array of references to Glorps:
  - `(Glorp[]) new Object[cap];`
  - Causes a compiler warning, which you should ignore
- Why not just `new Glorp[cap]`
  - Will cause a "generic array creation" error

Nulling Out Deleted Items

- Unlike integer based ALists, we actually want to null out deleted items
  - Java only destroys unwanted objects when the last reference has been lost
  - Keeping references to unneeded objects is sometimes called loitering
  - Save memory. Don't loiter