CSE 250 - Data Structures: Sequences, Arrays, and Lists

Andrew Hughes

SUNY at Buffalo
Computer Science and Engineering

Fall 2019

Outline

Sequences

A common way to store data is in a sequence.

- $2, 4, 6, 8, 10, 12$
- `'h','e','l','l','o','','w','o','r','l','d','!','\0'` (from `"hello world!"`)
- Lines of a file.

Frequently asked of data

- What is the first element?
- What are the elements from start to finish?
- How can I change element i?

What is an ADT?

Definition

An abstract data type (ADT) is a description of usage functionality that should be provided the type.

- Usually specified as an interface for functionality usage (API).
- Specifies the **what**, not the **how**.
- Different implementations may exist varying tradeoffs.

Seq/ListADT

What do we want of a sequence/list?

- `get(index)`
  Get the element at the specified index.
- `set(index,elem)`
  Replaces element at given index with a new value.
- `insert(index,elem)`
  Insert element at given index with a new value.
- `remove(index,elem)`
  Remove element at given index with a new value.

Scala Sequence: Seq

Mutable Sequence ADT

- `apply(i: Int): A`
  Get the element at the specified index.
- `iterator: Iterator[A]`
  Iterator can be used only once.
- `length: Int`
  The length (number of elements) of the sequence.
- `update(idx: Int, elem: A): Unit`
  Replaces element at given index with a new value.

Outline

Array Overview

Array Positives

- Fast/random access.
- Access as sequence (default set/get).

Array Pitfalls

- Fixed size.
- Empty values/entries.

Implement ListADT as FixedArrayList

Consider implementing the ListADT with a fixed length array.

Things to think about:

- What happens when we run out of space?
- How can an array location be empty?
- How are insert and remove handled?

Improve on FixedArrayList

How can we improve on our fixed-size array?

- Increase backing store space.
  - But by how much?

apply and update

Consider the `apply(...)` method:

- Used to retrieve values in array.
- Check if we are out of bounds and raise exception.
- One apply operation on array: $O(1)$ runtime.

Consider the `update(...)` method:

- Used to update values in array.
- Check if we are out of bounds and raise exception.
- One update operation on array: $O(1)$ runtime.

length

Consider the `length` method:

- Returns number of elements in list (not size of internal storage).
- Always maintained as value of `listSize` for $O(1)$ runtime.

Positional Insert and Remove

Positional insertion:

1. Check if position is valid and throw error if not.
2. Check that the backing store isn't full.
   A. If full, double the capacity.
3. From end-1 to position, shift right by one.
4. Assign position the inserted value.
5. Increase size by 1.

Positional removal:

1. Check if position is valid and throw error if not.
2. From position+1 until end, shift left by one.
3. Decrease the size by 1.

Worst case, $O(n)$ runtime.

Reserving Storage

Problem

How does reserving allow for "varying sized" array?

- Reserve more space as insertions/additions occur.
  1. Allocate new array with larger capacity.
     - ★ Usually double the capacity each time.
  2. Copy every element from old array to new array.
  3. Discard old array.

$\Theta(n)$ cost to reserve more space.

- n – current number of elements in storage.

Inserting to the End

Problem

What is the cost to fill an ArrayList?

Suppose we always insert next to end.
What is the cost of each insertion?

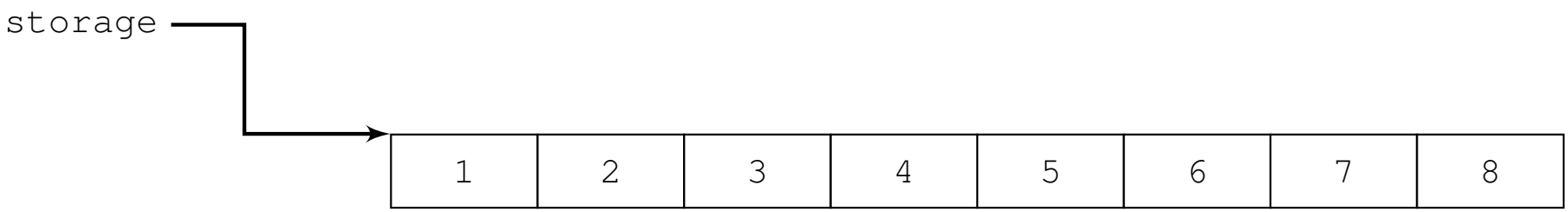
- Worst case, we need to reserve more storage: $O(n)$ runtime.
- This would mean: $O(n^2)$ to fill ArrayList.

Can we do better?

Cost of Multiple Insertions

Consider the following code:

```
val al = new ArrayList[Int]
for(i <- 1 to n) al.insert(i-1,i)
```



Runtime cost for `n` insertions?

- Analyze runtime for insertion as:
  - Cost for adding to end: $O(1)$.
    - ★ Nothing to shift right.
  - Cost for reserving new backing storage.
    - ★ Doesn't occur at every insert. Only when size is 0 or 2^k.

Cost of Multiple Insertions

```
val arrayList = new ArrayList[Int](0)
for(i <- 1 to n) arrayList.insert(i-1,i)
```

```
i = 1:   reserve(1):   0 moved
i = 2:   reserve(2):   1 moved
i = 3:   reserve(4):   2 moved
i = 4:
i = 5:   reserve(8):   4 moved
   ⋮
i = 8:
i = 9:  reserve(16):   8 moved
   ⋮
```

```
i = 16
i = 17: reserve(32): 16 moved
⋮
i = 2^j:
i = 2^j + 1: reserve(2^{j+1}): 2^j moved
⋮
i = n: ??
```
(last reserve: depends on n)

Cost of Multiple Insertions

- Suppose $n = 2^k + 1$, for some $k \in \mathbb{Z}^+$.
  - Worst case n since we double on last insertion.
- Insertion costs: n operations, $O(1)$ each: $\sum_{i=1}^{n} O(1) = \Theta(n)$.
- Doubling costs:
  - Double at every value of `i` where `i` $= 2^j$ for some j.
    - First: $j = 0$.
    - Last: $j = k$, where $k = \max_{j \in \mathbb{N}}\{0, j : 2^j < n\}$.
    - Since $n = 2^k + 1$, $k = \log(n-1)$.
  - Cost of reserve: $\sum_{i=0}^{k} 2^i = 2^{k+1} - 1$.
    - Since $k = \log(n-1)$:
      $$\sum_{i=0}^{\log(n-1)} 2^i = 2^{\log(n-1)+1} - 1 = 2(n-1) - 1 = 2n - 3.$$
    - Reserve runtime: $\Theta(n)$.
- Total cost of inserting n items to end of ArrayList:

$$T(n) = \text{insert cost} + \text{reserve cost} = \Theta(n) + \Theta(n) = \Theta(n)$$

Cost of Multiple Insertions

Runtime is **Amortized** $O(1)$.

- Runtime for a single insertion to end of ArrayList: $O(n)$.
- Runtime for inserting n items to end of ArrayList: $\Theta(n)$.

Amortized describes runtime over the long run.

- `reserve` only called to double size roughly $\log(n)$ times
  - i.e., very infrequently relative to n.
- Not quite the same as average case.
  - Average case is the **expected runtime** over any input.
  - Here, we have shown this **is the runtime**.

Amortized runtime: upfront cost averages out over time.

Outline

Lists

What improvements can we make upon the ArrayList?

- Allocation of storage on demand?
- Simplify implementation?
- Improve insertion/erasure time?

cse250.objects.SNode

We can fit the structure definition of a list all on one line.

```
class SNode[A](var data: A, var next: SNode[A])
```

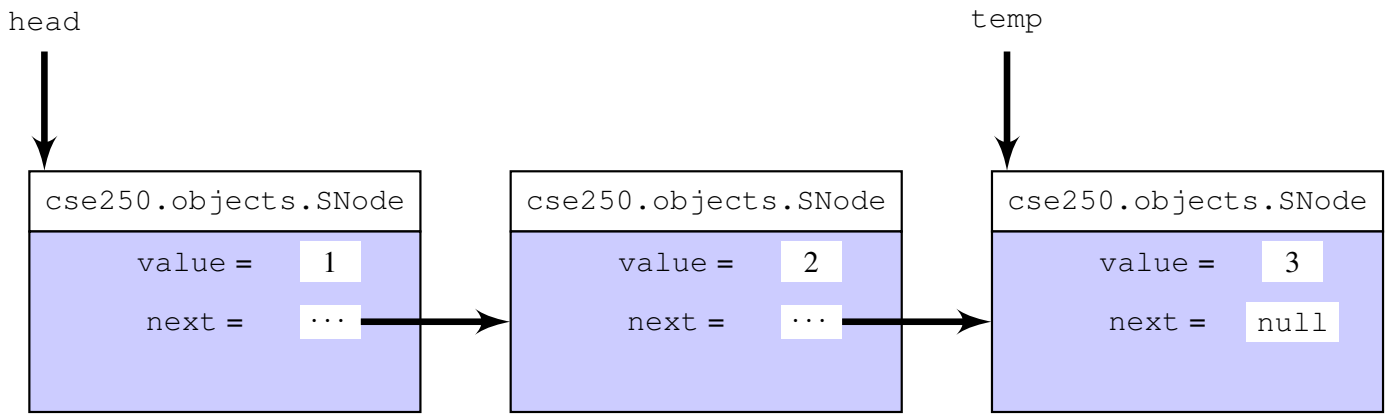
Examples/SNode.scala

- This is used for a singly linked list.
- “Simple” implementation.

cse250.objects.SNode Allocation

Allocate space on demand

```
var head = new cse250.objects.SNode(1,null)
var temp = head
temp.next = new cse250.objects.SNode(2,null)
temp = temp.next
temp.next = new cse250.objects.SNode(3,null)
temp = temp.next
```



cse250.objects.SNode Deallocation

Deallocate space on demand.

```
temp = head->next;
head = temp;
temp = head->next;
head = temp;
temp = head->next;
head = temp;
```

cse250.objects.DNode

What if we want/need the ability to traverse backwards?

```
class DNode[A](var data: A, var prev: DNode[A], var next: DNode[A])
```

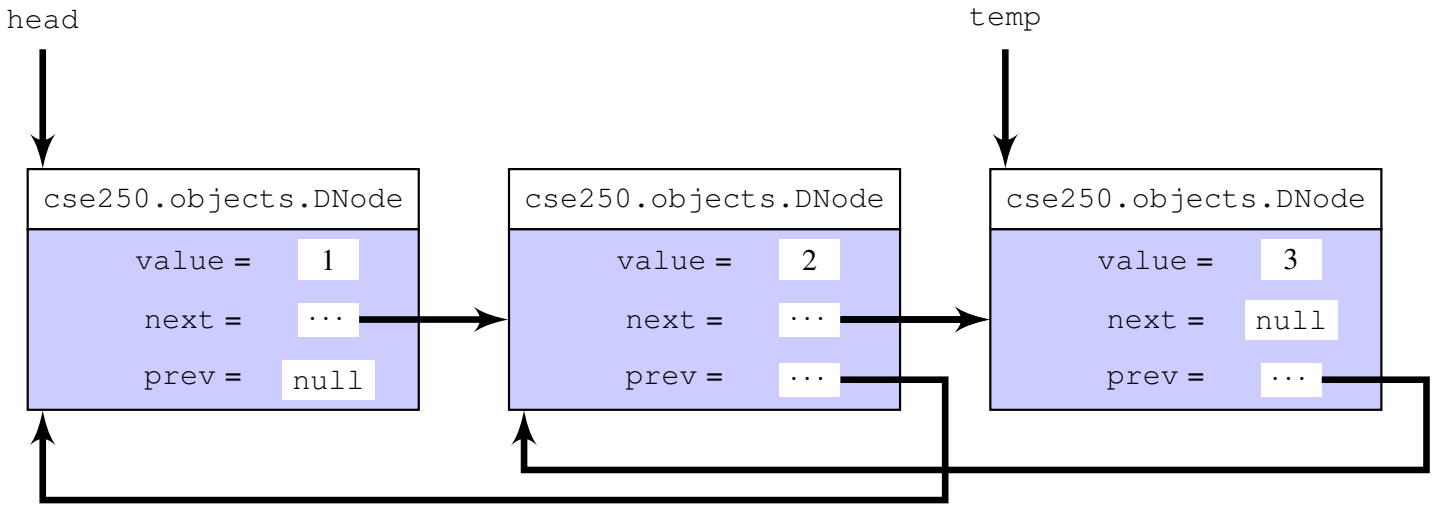
Examples/DNode.scala

- This node with `prev` and `next` is used for a doubly-linked list.
- Tracks the previous and next node in the list.

cse250.objects.DNode Allocation

Allocate space on demand maintaining link forward and backward.

```
var head = new cse250.objects.DNode(1,null,null)
var temp = head
temp.next = new cse250.objects.DNode(2,temp,null);
temp = temp.next;
temp.next = new cse250.objects.DNode(3,temp,null)
temp = temp.next;
```



cse250.objects.DNode Deallocation

Deallocation is the same.

```
temp = head.next;
head = temp;
head.prev = null;
temp = head.next;
head = temp;
head.prev = null;
temp = head.next;
head = temp;
```

temp

Implementing a List Container

- Member variables:
  - `headNode` – pointer to first element in the list.
  - `tailNode` – pointer to last element in the list.
  - `listSize` – number of nodes in the list.
- Positional operations:
  - `insert` – add value at provided position in the list.
  - `remove` – remove value at provided position in the list.
- Access/Modify
  - `apply` – return the i-th element in the list.
  - `update` – update the i-th element in the list.

Overview of Runtime for List Methods

	Runtime	
Function	Unsorted	Positional
`apply`	$\Theta(i)$	$\Theta(i)$
`update`	$\Theta(i)$	$O(1)$
`insert`	$\Theta(i)$	$O(1)$
`remove`	$\Theta(i)$	$O(1)$

How could we improve upon this?

- Provide positional access via Position object.

Outline

ArrayList vs LinkedList: Terminology

ArrayList

- `capacity` – size of backing array.

LinkedList

- `headNode` – first node in the list.
- `tailNode` – last node in the list.

Shared

- `length` – number of elements stored.

ArrayList vs LinkedList: Highlights

ArrayList

- Random access
- Contiguous memory
- Easy to understand implementation

LinkedList

- Efficient positional operations
- Allocate on demand
- Simplified implementation (though harder conceptually)

ArrayList vs LinkedList: Sequence Operations

ArrayList

Iteration (visiting all elements) is $\Theta(n)$.

- Access may be faster: memory techniques makes access more efficient.

LinkedList

Iteration (visiting all elements) is $\Theta(n)$.

- Access may be slower: no guarantee on locality in memory.

Overview of Runtime

Method	Runtime			
	ArrayList		LinkedList	
	Unsorted	Sorted	Unsorted	Sorted
`apply`	$O(1)$	$O(1)$	$O(i)$	$O(i)$
`update`	$O(i)$	–	$O(1)*$	–
`insert`	$O(n)$	$O(n)$	$O(1)*$	$O(n)$
`remove`	$O(n)$	$O(n)$	$O(1)*$	$O(n)$
`insert(n,e)`	Amortized $O(1)$	–	$O(1)$	–

* Position operation runtimes