List ADT

CIT 5940

Abstract Data Types

- An ADT defines a data type and a set of operations on that type
  - An ADT does not specify how the data type is implemented
- **Interfaces** define ADTs in Java, **Classes** define their implementations
  - A data structure is the implementation for an ADT
  - The **List** is an ADT, **LinkedList** and **ArrayList** are implementations

The List ADT

- A List is a **finite, ordered sequence of data items (all of a single type) known as *elements.***
    - Finite: specific size, although the size may change over time
    - Ordered: each element has a position in the list called an **index**
- The operations: (well, some of them)
    - Append
        - `public boolean add(E e);`
    - Insert
        - `public boolean add(int index, E e);`
    - Get
        - `public E get(int index);`
    - Remove
        - `public E remove(int index);` and `public boolean remove(Object o);`

Java's List Interface & Implementation

- The java.util.List interface contains a ton of other methods
  - Won't discuss them here, but make sure you remember that you can use them!
- The two common implementations of Lists:
  - Array-based lists
    - Store an array internally to contain all of the elements, resize as needed
    - In Java, this is the **ArrayList** class
  - Linked lists
    - Each element is stored in a Node, linking the nodes defines the order of the List
    - In Java, this is the **LinkedList** class

List Check-in

- Write a method that collects all elements of an integer array above a value **k**.

```
public List<Integer> takeAbove(int[] array, int k) {
    return null;
}
```

List Check-in

- Write a method that collects all elements of an integer array above a value **k**.

```
public List<Integer> takeAbove(int[] array, int k) {
    List<Integer> filtered = new LinkedList<>();
    for (int number : array) {
        if (number > k) {
            filtered.add(number);
        }
    }
    return filtered;
}
```

List Check-in

- Write a method that collects all elements of an integer array above a value **k**.

```
public List<Integer> takeAbove(int[] array, int k) {
    List<Integer> filtered = new LinkedList<>();
    for (int number : array) {
        if (number > k) {
            filtered.add(number);
        }
    }
    return filtered;
}
```

Could have used an ArrayList, too!

Linked Lists

What is a Linked List?

- A ***doubly-linked list*** is a data structure for implementing a list ADT, where each node has
  - Data
  - a pointer to the next node
  - a pointer to the previous node.
- The list structure typically has pointers to the list's first node (the **head**) and last node (the **tail**).

Linked List Instance Variables & Constructor

- Singly or doubly linked, the LinkedList should keep track of:
  - the head, the tail
  - (for convenience) the # of elements contained
- The constructor creates an empty List, so set the head and tail to be null.

```
public class LinkedList<E> implements List<E> {

    private Node head;
    private Node tail;
    private int size;

    public LinkedList() {
        head = null;
        tail = null;
    }

...
}
```

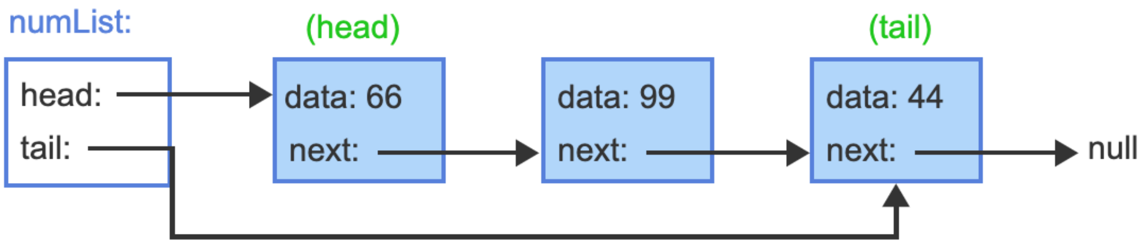
Appending to a Singly Linked List: add(E e)

numList = new List

ListAppend(numList, node 99)

ListAppend(numList, node 44)

ListPrepend(numList, node 66)



Appending to a Singly Linked List: add(E e)

- Handle an empty list specially:
  - head & tail should be updated to point to the new, single Node
- If the list isn't empty:
  - Point the tail node to the new node
  - Update the tail variable to point to the new node

```java
@Override
public boolean add(E e) {
    Node newNode = new Node(e);
    if (head == null) {
        head = newNode;
        tail = newNode;
    } else {
        tail.next = newNode;
        tail = newNode;
    }
    size++;
    return true;
}
```

Appending to a Singly Linked List: add(E e)

- Handle an empty list specially:
  - head & tail should be updated to point to the new, single Node
- If the list isn't empty:
  - Point the tail node to the new node
  - Update the tail variable to point to the new node

What would be different in a Doubly Linked List?

```
@Override
public boolean add(E e) {
    Node newNode = new Node(e);
    if (head == null) {
        head = newNode;
        tail = newNode;
    } else {
        tail.next = newNode;
        tail = newNode;
    }
    size++;
    return true;
}
```

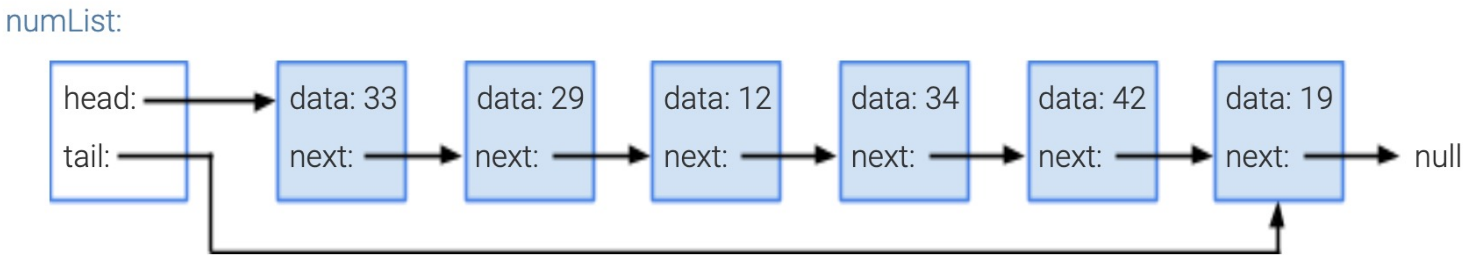
Appending to a Doubly Linked List: add(E e)

- Algorithm is the same!
- Handle an empty list specially:
  - head & tail should be updated to point to the new, single Node
- If the list isn't empty:
  - Point the tail node to the new node **and the new node to the previous tail!**
  - Update the tail variable to point to the new node

```
@Override
public boolean add(E e) {
    Node newNode = new Node(e);
    if (head == null) {
        head = newNode;
        tail = newNode;
    } else {
        tail.next = newNode;
        newNode.previous = tail;
        tail = newNode;
    }
    size++;
    return true;
}
```

Working up to Insert, Remove, Get

- Each of these three abstract methods behaves basically like **linear search**
  - add(int index, E e), get(int index), remove(int index) traverse the list to a specific index
  - remove(Object o) searches for **o** and tries to remove it
- A simple linked list traversal algorithm:
  - Start at the list's head node,
  - Follow next pointers **index** many times
  - Return a reference to the node



Activity

- Return the (non-null) Node at the specified element index.
- Main idea: start from the head and follow pointers until we're at the desired Node.

```
Node<E> node(int index) {


}
```

Activity

- Return the (non-null) Node at the specified element index.

- O(n) solution!
    - For loop runs i times, i ranges from 0 to n

```
Node<E> node(int index) {

    Node<E> x = first;
    for (int i = 0; i < index; i++)
        x = x.next;
    return x;

}
```

Insert: add(int index, E e)

```
public void add(int index, E element) {
    if (index < 0 || index > size()) {
        throw new IllegalArgumentException();      O(1)
    }

    if (index == size)
        add(element);                              O(1)
    else
        linkBefore(element, node(index));          ???
}
```

Insert: add(int index, E e)

```
public void add(int index, E element) {
    if (index < 0 || index > size()) {
        throw new IllegalArgumentException();
    }

    if (index == size)
        add(element);
    else
        linkBefore(element, node(index));
}
```

O(1)

O(1)

???

Insert: add(int index, E e)

```java
public void add(int index, E element) {
    if (index < 0 || index > size()) {
        throw new IllegalArgumentException();
    }

    if (index == size)
        add(element);
    else
        linkBefore(element, node(index));
}
```

```java
void linkBefore(E e, Node<E> succ) {
    final Node<E> pred = succ.prev;
    final Node<E> newNode = new Node<>(pred, e,
succ);
    succ.prev = newNode;
    if (pred == null)
        first = newNode;
    else
        pred.next = newNode;
    size++;
    modCount++;
}
```

linkBefore moves a lot of pointers, but just O(1)

Insert: add(int index, E e)

```
public void add(int index, E element) {
    if (index < 0 || index > size()) {
        throw new IllegalArgumentException();
    }

    if (index == size)
        add(element);
    else
        linkBefore(element, node(index));
}
```

O(1)

O(1)

O(n)

Activity: improving node(int index)

- Can’t do better than O(n) solution in general.
- Right now, O(n) for getting node n. Bad!
- How can we minimize the number of iterations needed to get to any particular location in a **doubly linked list**?

```
Node<E> node(int index) {

        Node<E> x = first;
        for (int i = 0; i < index; i++)
            x = x.next;
        return x;

}
```

Activity: improving node(int index)

- Can't do better than O(n) solution in general.
- Right now, O(n) for getting node n. Bad!
- How can we minimize the number of iterations needed to get to any particular location in a **doubly linked list**?

```
Node<E> node(int index) {
    if (index < (size / 2)) {
        Node<E> x = first;
        for (int i = 0; i < index; i++)
            x = x.next;
        return x;
    } else {
        Node<E> x = last;
        for (int i = size - 1; i > index; i--)
            x = x.prev;
        return x;
    }
}
```

Get and Remove Work Basically the Same!

```java
public E get(int index) {
    checkElementIndex(index);
    return node(index).item;
}
```

```java
public E remove(int index) {
    checkElementIndex(index);
    return unlink(node(index));
}
```

(both are O(n), with optimization to help search from the closer side)

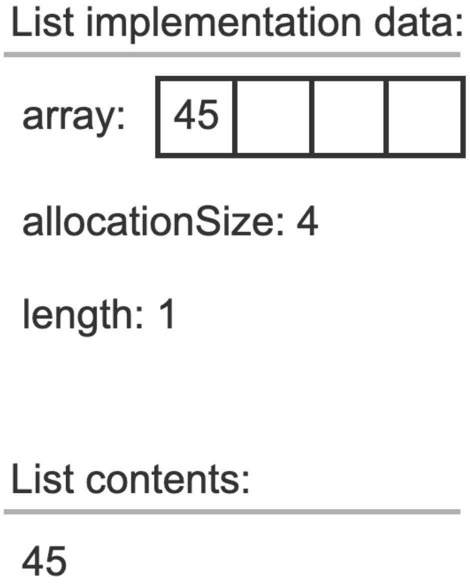
Linked List Summary

- Each element is stored in a **node**
- The position of an element in the list is determined by how many nodes away from the head it is
- Can’t access an element in the list directly, must follow pointers
- **Adding** to the head/tail of a linked list is constant time
- **Adding, getting, and removing** within the linked list is linear time
    - **Doubly linked lists** allow starting from the closer end; not faster asymptotically but still useful in practice.

Array Lists

What is an Array List?

- An ***array-based list*** is a list ADT implemented using an array.
- The implementation usually needs to track:
  - The array
  - The allocation size
  - The current size of the list
- In Java, arrays are fixed in size!
  - We'll need to implement our own **dynamic allocation**
  - Let's come back to this later



Structure of an Array List

```
public class FixedArrayList<E> implements List<E>{

    Object[] array;
    int allocationSize;
    int size;

    final int DEFAULT_CAPACITY = 10;

    public FixedArrayList(int initialCapacity) {
        allocationSize = initialCapacity;
        size = 0;
        array = new Object[initialCapacity];
    }
    ....
}
```

Activity: Adding to a Fixed-Length Array List

- Given instance variables **array**, **allocationSize**, and **size**, write a function that
  - Adds an element to the end of the list if there's space and returns **true**
  - Return **false** and do nothing otherwise.

```
@Override
public boolean add(E e) {







}
```

Activity: Adding to a Fixed-Length Array List

- Given instance variables **array**, **allocationSize**, and **size**, write a function that
  - Adds an element to the end of the list if there's space and returns **true**
  - Return **false** and do nothing otherwise.

```
@Override
public boolean add(E e) {
    if (size == allocationSize) {
        return false;
    }
    array[size] = e;
    size++;
    return true;
}
```

Understanding add(int index, E e)

- If we want to **add(4, 17),** we can't just do **array[4] = 17;**
  - Clobbers whatever is there already!
- To insert at a specific index, we'll need to:
  - Copy all elements at and after this index one place to the right
  - Then, place the desired element at **index**

list:

array:	91	45	84	36	12	78	51	
	0	1	2	3	4	5	6	7

allocationSize: 8
length: 7

Expected output:
91 45 84 36 17 12 78 51

Understanding remove(int index)

- If we want to **remove(3),** we can't just do **array[3] = null;**
  - Leaves an unused spot in the array!
- To remove from a specific index, we'll need to:
  - Copy all elements after this index one place to the left

list:

array:

91	45	84	36	12	78	51	
0	1	2	3	4	5	6	7

allocationSize: 8
length: 7

Expected output:
91 45 84 17 12 78 51

Adding/Removing within a list is an O(n) operation*!

*Worst case

```java
public boolean add(int index, E e) {
    if (size == allocationSize) {
        return false;
    }
    for (int i = size; i > index; i--) {
        array[i] = array[i - 1];
    }
    array[index] = e;
    size++;
    return true;
}
```

```java
public E remove(int index) {
    E toRemove = (E) array[index];
    for (int i = index; i < size - 1; i++) {
        array[i] = array[i + 1];
    }
    array[size - 1] = null;
    size--;
    return toRemove;
}
```

These loops run once for each of the elements after the target index!

Fixed Size to Resizing

- One small tweak, and we have resizing Array Lists

```
public boolean add(E e) {
    if (size == allocationSize) {
        return false;
    }
    array[size] = e;
    size++;
    return true;
}
```

```
public boolean add(E e) {
    if (size == allocationSize) {
        grow();
    }
    array[size] = e;
    size++;
    return true;
}
```

Fixed Size to Resizing

- **grow()** is clearly an O(n) operation, but with some *amortized analysis magic*, we can prove that resizing array lists still have **O(1)** append!

```
private void grow() {
    Object[] newArray = new Object[allocationSize * 2];
    for (int i = 0; i < allocationSize; i++) {
        newArray[i] = array[i];
    }
    array = newArray;
    allocationSize = allocationSize * 2;
}
```

Amortized Analysis

- An algorithm analysis technique that looks at the total cost for a series of operations and amortizes this total cost over the full series.
- Useful when the individual analysis of the worst case cost might lead to an overestimate for the total cost of the series.

Proof!

- We'll use mathematical induction:
- Base case:
  - Array List is empty, so size = 0.
  - Just do **array[0] = e**
- Induction hypothesis:
  - Assume that the average cost of appending n-1 elements is in $\mathbf{O((n - 1) / (n - 1)) \rightarrow O(1)}$

Proof!

- Need to show that the average cost is **O(1)** for n elements
- Appending the nth element leads to two cases:
  - Case 1: array is not full, and so we have a constant time **array[i] = e.** Average cost of adding the first n-1 is **O(n-1),** cost of the next one is **O(1)**, so average cost is **O(n – 1 + 1) / n → O(1).**
  - Case 2: the array is full, and so we pay **O(n)** for the grow operation and **O(1)** for the actual append.
    - The total cost is now **O(n + n + 1)**, averaged over **n** elements
    - **O(2n + 1) / n → O(1)**
- So, the average cost of appending an element to a resizing array-based list is **O(1)!**

Runtime Analysis Summary

Operation	Array List	Singly Linked List	Doubly Linked List
`append`	O(1)*	O(1)	O(1)
`insert`	O(n)	O(n)	O(n)**
`delete`	O(n)	O(n)	O(n)**
`getValue`	O(1)	O(n)	O(n)

*amortized!
**faster in practice than the O(n) for singly LLs

Space Complexity

- **Overhead** refers to all information stored by a data structure aside from the actual data
  - Smaller overhead means better space complexity
- For Array Lists
  - Size must be predetermined before the array can be allocated
  - Unused space (overhead) if the array contains few elements
  - No overhead when array is full
- For Linked Lists
  - Only need space for the elements in the list
  - Needs space for `next` and/or `prev` pointers (overhead)

Which to choose?

Given :

n the number of elements in the list

P the size of a pointer

E the size of a data element

D the maximum number elements that can be stored in the array

Space complexity

- Array List: DE
- Linked Lists: $n(P+E)$

Break-Even Point

$$(1)\ n > DE/(P+E)$$

Solving *(1)* for n gives us the break-even point beyond which the array-based implementation is more space efficient

If we assume $P = E$ then break-even point is $D/2$ (array half full)

Rule of Thumb

- Linked Lists are more space efficient when the number of elements varies widely or is unknown
- Array Lists are more space efficient when you know the eventual size of the list in advance.