# Lists: ArrayList and LinkedList

# List Interfaces Methods

- `add(Elm)`
- `add(index, Elm)`
- `addAll(Coll)`
- `clear()`
- `contains(Obj)`
- `containsAll(Coll)`
- `equals(Obj)`
- `get(index)`
- `hashCode()`
- `indexOf(Obj)`
- `isEmpty()`
- `iterator()`
- `lastIndexOf(Obj)`
- `remove(index)`
- `removeAll(Coll)`
- `remove(Obj)`
- `retainAll(Coll)`
- `set(index, Elm)`
- `size()`
- `subList(from, to)`
- `toArray()`
- `toArray(T[] arr)`

# List Implementations

- Two implementations:
  - `ArrayList`
  - `LinkedList`
- `ArrayList`:
  - Resizable array
    - Essentially a wrapper class around an array
  - Uses Contiguous Allocation
  - Usually has more memory allocated than is strictly necessary
- `LinkedList`
  - Uses Linked Allocation

# ArrayList: add(E)

- Adds the element to the *end* of the list

  `ArrayList<Integer> myAL = newArrayList<>({3, 1, 4});`

| | | | |
|---|---|---|---|
| 3 | 1 | 4 | ∅ |

  `myAL.add(1);`

| | | | |
|---|---|---|---|
| 3 | 1 | 4 | 1 |

- What happens if we now `myAL.add(5);`?
  - List is full
  - Create a new list that is twice as big, copy the data, then add to the end

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 3 | 1 | 4 | 1 | | | | |
| 3 | 1 | 4 | 1 | ∅ | ∅ | ∅ | ∅ |
| 3 | 1 | 4 | 1 | 5 | ∅ | ∅ | ∅ |

- Adding to the end is fast (except every once in a while, don't worry about that)

# ArrayList: `add(index, E)`

- Adds the item at the specified index

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 3 | 1 | 4 | 1 | 5 | ∅ | ∅ | ∅ |

```
myAL.add(1, 10); // Add the number 0 at
index 1
```

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 3 | 1 | 4 | 1 | 5 | ∅ | ∅ | ∅ |
| 3 | ∅ | 1 | 4 | 1 | 5 | ∅ | ∅ |
| 3 | 10 | 1 | 4 | 1 | 5 | ∅ | ∅ |

- Everything from the specified index on must be shifted to the right
  - Worst case, if inserting at index 0, *every* element must be shifted
- This operation is slow
  - The operation will take longer with 1000 elements than with 10

# Other ArrayList Operations

- `get(index)`
  - Does simple math, is fast
- `indexOf(Obj)`
  - Must search through and visit every element (potentially), is slow
- `lastIndexOf(Obj)`
  - Like `indexOf(Obj)`, but in reverse, so is slow
- `remove(index)` or `remove(Obj)`
  - Go to the index (fast ), shift everything left (slow), so slow
- `set(index, E)`
  - Go to the index (fast ), do assignment (fast ), so fast

# ArrayList: Summary

- Fast at/Good at
  - `get(index)`, `set(index, Elm)` (Random Access)
  - Add and Remove at end (usually)

- Slow at/Bad at
  - Add at beginning/middle
  - Remove from beginning/middle
  - Finding elements (searching)
  - Can use up to twice as much memory as needed to hold data

# Linked Lists

- Linked Lists are built of nodes (like a chain)
  - A node consists of (at least) a data field and a reference to the next node in the list (Singly Linked List)
  - The node may also contain a reference to the previous node in the list (Doubly Linked List)

Singly Linked

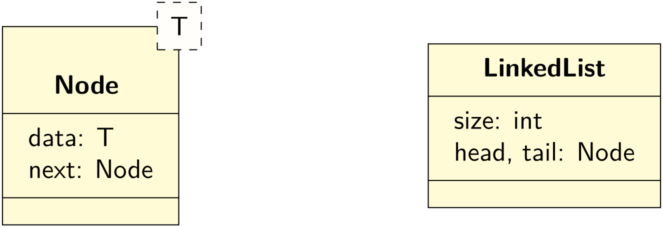
| data |
| --- |
| next |

Doubly Linked

| data | |
| --- | --- |
| prev | next |

- The `LinkedList` maintains a reference to the beginning (`head`) and the end (`tail`) of the list
- All examples in the following slides will demonstrate with a Singly Linked List; it is relatively straightforward to expand your understanding to a Doubly Linked List

# LinkedList Nodes




Note: A doubly linked node would have a `prev` member as well.

# LinkedList

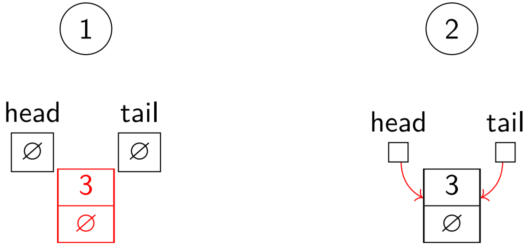
```
LinkedList myLL = new LinkedList<>();
```

| head | tail |
|---|---|
| $\varnothing$ | $\varnothing$ |

```
myLL.add(3); // Add to empty list
```

1. Create a new node with the data and the `next` reference set to null
2. point both `head` and `tail` to point at the new node

# Adding to Empty LinkedList

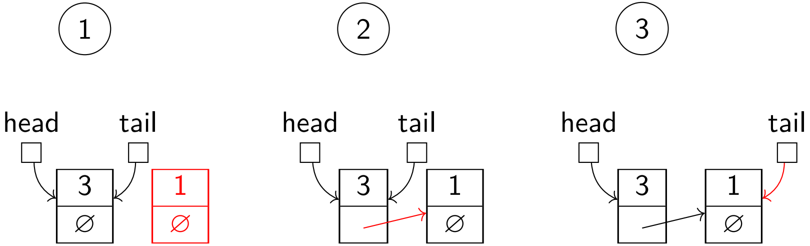


# Adding to End of LinkedList

```
myLL.add(1); // or myLL.addLast(1)
```

1. Create a new node with the data (and a null `next` reference)
2. Point `tail.next` at new node
3. Point `tail` at new node

# Adding to End of LinkedList



# Adding to End of LinkedList

- The number of operations this takes is independent of the number of things in the list (and is small).
- This means the operation is fast
- Similarly, adding at the beginning is fast
  - Replace `tail` with `head` in pseudocode

# LinkedList addLast Pseudocode

```
addLast(data):
    if size == 0:
        head = new Node(data, null)
        tail = head
    else:
        temp = new Node(data, null)
        tail.next = temp
        // if doubly linked list, temp.prev = tail
        tail = temp
    size ++
```

# Random Access: LinkedList

- The only two nodes we know the positions of are `head` and `tail`
- In order to get to a node at a specific index (to either get the value or set the value), we must start at the `head` and iterate through
- The amount of time this can take grows as the list grows, thus it is a slow operation

# Random Access Pseudocode: LinkedList

```
get(index):
    if index >= size:
        throw Exception
    Node it = head
    for (int i=0; i<index; i++)
        it = it.next
    return it.data

set(index, data):
    if index >= size:
        throw Exception
    Node it = head
    for (int i=0; i<index; i++)
        it = it.next
    it.data = data
```

# Add to Middle Pseudocode: LinkedList

```
add(index, data):
    if index > size:
        throw Exception
    if index == 0:
        addFirst(data)
    else if index == size:
        addLast(data)
    else:
        Node it = head
        // iterate to node before index to add at
        for (int i=0; i<index - 1; i++)
            it = it.next
        temp = new Node(data, null)
        temp.next = it.next
        it.next = temp
        size ++
```

# Addition and Removal from Middle of LinkedList

- Removal is similar to addition in the middle of a LinkedList
  - Iterate to node before index to add/remove
  - Link or Unlink node as appropriate
- The linking and unlinking is fast
- Iterating to the node before the node to modify, however, is slow
- Thus, these operations are slow
  - The exception: Removal from the beginning of a Doubly Linked List is fast

# Searching in a LinkedList

- Like the ArrayList, searching in a LinkedList requires visiting possibly all elements
- Thus, operations like `indexOf` are slow

# Comparing ArrayList and LinkedList

| | Random Access | Add/Rm @ Beg | Add/Rm @ Mid | Add @ End | Rm @ End | Search |
|---|---|---|---|---|---|---|
| ArrayList | fast | slow | slow | fast | fast | slow |
| Singly LinkedList | slow | fast | slow | fast | slow | slow |
| Doubly LinkedList | slow | fast | slow | fast | fast | slow |

Note: Arrays are similar to ArrayLists, but are not resizable. They do have arguably more convenient indexing (citation needed).