Ordered collections
Linked Datastructures
Lists
Stacks
Queues
Ordered Collections

- List
- Stack
- Queue
List: Essential Characteristics

Collection of items

Copies/duplicates of items: relevant

Order: relevant!
List<T>

- addAt(atPosition: int, newItem: T): void
- addFront(newItem: T): void
- addBack(newItem: T): void
- get(position: int): T
- remove(position: int): T
- size(): int
- isEmpty(): boolean

(one possible design)

Common synonym for List is Sequence
Who is at position 1 in this sequence (left to right)?
Who is at position 1?

One-based:
• First element at position 1
• Second element at position 2
• ...

Zero-based:
• First element at position 0
• Second element at position 1
• ...

Like how everyone except computer scientists would count

Like array indexing

Every list ADT gets to make its own choice about this issue. Implementer’s job is to document it. Client’s job is to be aware.
Review: node diagrams

Instead of this object diagram:

```
Node<T>
  next: Node<T>
  data: T

... T
```

DSAJ draws this node diagram:
Implementing a list using linked nodes

(today we don’t bother drawing data objects or pointers)

Except for tail, this is exactly our structure for LinkedBag!
Code Demo: LinkedList
Exercise

1. **Draw** the node diagram of the desired outcome.
(2. **Write** the code to make it happen.)

Goal: add `newNode` at position 3
Edge case 1 of 3. What if list is empty?
Edge case 1 of 3. What if list is empty?
Edge case 2 of 3. Insert at front?

- head
- newNode
- tail
Edge case 2 of 3. Insert at front?
Edge case 3 of 3. Insert at end but not front?
Edge case 3 of 3. Insert at end but not front?
Question

Consider addAt() in LinkedList. In the best case, is its number of reads/writes from memory independent or dependent on the number of items in the list? What about in the worst case?

<table>
<thead>
<tr>
<th>Answer</th>
<th>Best Case</th>
<th>Worst Case</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>Independent</td>
<td>Independent</td>
</tr>
<tr>
<td>B</td>
<td>Independent</td>
<td>Dependent</td>
</tr>
<tr>
<td>C</td>
<td>Dependent</td>
<td>Independent</td>
</tr>
<tr>
<td>D</td>
<td>Dependent</td>
<td>Dependent</td>
</tr>
<tr>
<td>E</td>
<td>Send halp I’m confused</td>
<td></td>
</tr>
</tbody>
</table>
Consider `addAt()` in `LinkedList`. In the best case, is its number of reads/writes from memory independent or dependent on the number of items in the list? What about in the worst case?

<table>
<thead>
<tr>
<th>Option</th>
<th>Votes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Best Case: Independent, Worst Case: Independent</td>
<td>0%</td>
</tr>
<tr>
<td>Best Case: Independent, Worst Case: Dependent</td>
<td>0%</td>
</tr>
<tr>
<td>Best Case: Dependent, Worst Case: Independent</td>
<td>0%</td>
</tr>
<tr>
<td>Best Case: Dependent, Worst Case: Dependent</td>
<td>0%</td>
</tr>
<tr>
<td>Send help I'm confused :)</td>
<td>0%</td>
</tr>
</tbody>
</table>
Efficiency: `LinkedList.addAt()`

**Best case:** insert at head or tail. Running time independent of size of list.

**Worst case:** insert in middle. Very worst: just before tail. Running time dependent on size of list.
Exercise

How would you check whether a list contains any duplicate items?
Write a loop to do it!

Q: is the running time of hasDuplicates() independent or dependent on the number of items?
In LinkedList, both addAt() and hasDuplicates() can be dependent on the number of items in the list. If the list gets really big, will there be a noticeable difference in their running times?

A. Yes, addAt() will always be much slower than hasDuplicates()
B. Yes, hasDuplicates() will always be much slower than addAt()
C. No, they will always run in about the same time
D. It depends on the contents of the list; sometimes one will be slower than the other; other times they will be the same
E. Time is an illusion: I’m confused by the question
In LinkedList, both addAt() and hasDuplicates() can be dependent on the number of items in the list. If the list gets really big, will there be a noticeable difference in their running times?

- Yes, addAt() will always be much slower than hasDuplicates()  
  0%

- Yes, hasDuplicates() will always be much slower than addAt()  
  0%

- No, they will always run in about the same time  
  0%

- It depends on the contents of the list; sometimes one will be slower than the other; other times they will be the same  
  0%

- Time is an illusion: I’m confused by the question  
  0%
Comparison: `addAt()` and `hasDuplicates()`

Add at front and back: never depends on number of items in list

Checking for duplicates: if first two elements are duplicates, never depends on number of items in list

...in **best case** neither depends on size!
In LinkedList, both addAt() and hasDuplicates() can be dependent on the number of items in the list. If the list gets really big, will there be a noticeable difference in their running times?

A. Yes, addAt() will in the **worst case** be much slower than hasDuplicates()

B. Yes, hasDuplicates() will in the **worst case** be much slower than addAt()

C. No, in the **worst case** they will run in about the same time
In LinkedList, both `addAt()` and `hasDuplicates()` can be dependent on the number of items in the list. If the list gets really big, will there be a noticeable difference in their running times?

<table>
<thead>
<tr>
<th>Option</th>
<th>Percentage</th>
</tr>
</thead>
<tbody>
<tr>
<td>Yes, <code>addAt()</code> will in the worst case be much slower than <code>hasDuplicates()</code></td>
<td>0%</td>
</tr>
<tr>
<td>Yes, <code>hasDuplicates()</code> will in the worst case be much slower than <code>addAt()</code></td>
<td>0%</td>
</tr>
<tr>
<td>No, in the worst case they will run in about the same time</td>
<td>0%</td>
</tr>
</tbody>
</table>
Comparison

Let’s run some experiments!
Efficiency

Dependent vs independent of size isn’t the whole story
Another list data structure

Singly:

Doubly:

What is an example of an operation that would be more efficient with doubly-linked lists?
Ordered Collections

- List
- Stack
- Queue
Stack and Queue Operations

- Add item
- Remove and return next item
- Return next item without removing
Stack\(<T>\)

- `push(newItem: T): void`
- `pop(): T`
- `peek(): T`
- `isEmpty(): boolean`

Queue\(<T>\)

- `enqueue(newItem: T): void`
- `dequeue(): T`
- `peek(): T`
- `isEmpty(): boolean`

(one possible design)

<table>
<thead>
<tr>
<th>Operation</th>
<th>Stack</th>
<th>Queue</th>
</tr>
</thead>
<tbody>
<tr>
<td>Add item</td>
<td><code>push</code></td>
<td><code>enqueue</code></td>
</tr>
<tr>
<td>Remove and return next item</td>
<td><code>pop</code></td>
<td><code>dequeue</code></td>
</tr>
<tr>
<td>Return next item without removing</td>
<td><code>peek</code></td>
<td><code>peek</code></td>
</tr>
</tbody>
</table>
Stack and Queue Implementations
As Linked List

Stack

Queue

push and pop from head
dequeue from head
enqueue at tail
Code Demo: LinkedStack and LinkedQueue
Array Bag (fixed capacity)

- add: independent of size
- frequencyOf: dependent on size

```java
void add(T item) {
    // omitted: if no room, throw exception
    items[size] = item;
    size++;
}
```

```java
int frequencyOf(T item) {
    int count = 0;
    for (int i = 0; i < size; i++) {
        if (items[i].equals(item))
            count++;
    }
    return count;
}
```
Array Bag (resizable)

- **add**: worst case, dependent on size; best case, independent
- **frequencyOf**: always dependent on size

```java
void add(T item) {
    // omitted: if no room, copy into bigger array
    items[size] = item;
    size++;
}

int frequencyOf(T item) {
    int count = 0;
    for (int i = 0; i < size; i++) {
        if (items[i].equals(item))
            count++;
    }
    return count;
}
```
Linked Bag

- **add**: always independent of size
- **frequencyOf**: always dependent on size

Efficiency + unbounded capacity was the great win of linked bags over array bags
Linked List

• addAt: best case independent of size; worst case dependent
• hasDuplicates: not only dependent, but runs considerably slower than addAt as size increases [experiment]
How is stair climbing like addAt and hasDuplicates?
Linked List

addAt, worst case, size n

hasDuplicates, worst case, size n