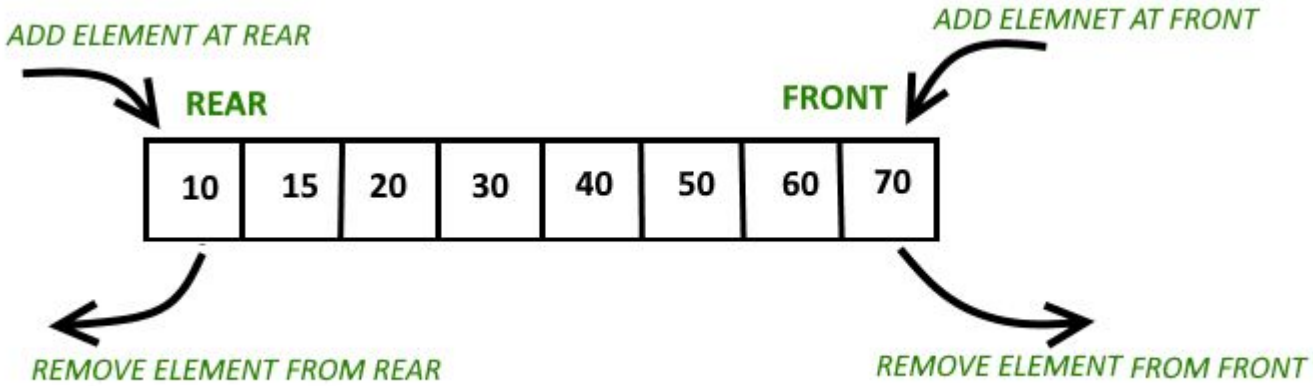
Discussion 4: Project 2

Kyle Hackett

Problem 1

Problem 1. (*Deque*) A double-ended queue or deque (pronounced "deck") is a generalization of a stack and a queue that supports adding and removing items from either the front or the back of the data structure. Create a generic, iterable data type called `LinkedDeque` that uses a doubly-linked list to implement the following deque API:



LinkedDeque	
`LinkedDeque()`	constructs an empty deque
`boolean isEmpty()`	returns `true` if this deque empty, and `false` otherwise
`int size()`	returns the number of items on this deque
`void addFirst(Item item)`	adds `item` to the front of this deque
`void addLast(Item item)`	adds `item` to the back of this deque
`Item peekFirst()`	returns the item at the front of this deque
`Item removeFirst()`	removes and returns the item at the front of this deque
`Item peekLast()`	returns the item at the back of this deque
`Item removeLast()`	removes and returns the item at the back of this deque
`Iterator<Item> iterator()`	returns an iterator to iterate over the items in this deque from front to back
`String toString()`	returns a string representation of this deque

Use a doubly-linked list `Node` to implement the API — each node in the list stores a generic `item`, and references `next` and `prev` to the next and previous nodes in the list

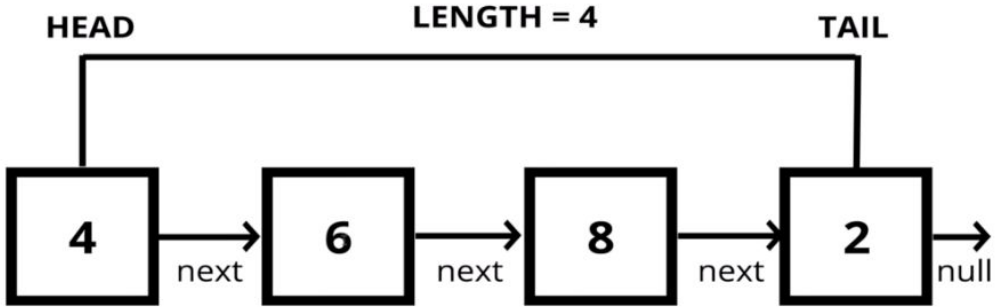
$$\texttt{null} \leftarrow \boxed{item_1} \leftrightarrow \boxed{item_2} \leftrightarrow \boxed{item_3} \leftrightarrow \cdots \leftrightarrow \boxed{item_n} \rightarrow \texttt{null}$$

Instance variables:

- Reference to the front of the deque, `Node first`.
- Reference to the back of the deque, `Node last`.
- Size of the deque, `int n`.

d Lists

Building it from Scratch: Using Nodes



`LinkedDeque()`

- Initialize instance variables to appropriate values.

`boolean isEmpty()`

- Return whether the deque is empty or not.

What instance variable would be useful?

Adding to Deque

```
void addFirst(Item item)
```

- Add the given item to the front of the deque.
- Increment `n` by one.

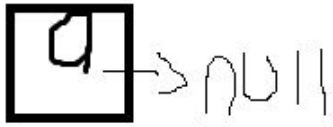
```
void addLast(Item item)
```

- Add the given item to the back of the deque.
- Increment `n` by one.

Node.item defaults to null

Node.next defaults to null

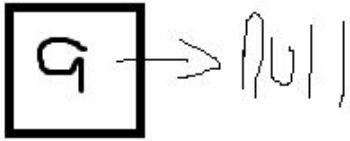
Basic Link list insertion



item:null
next: null

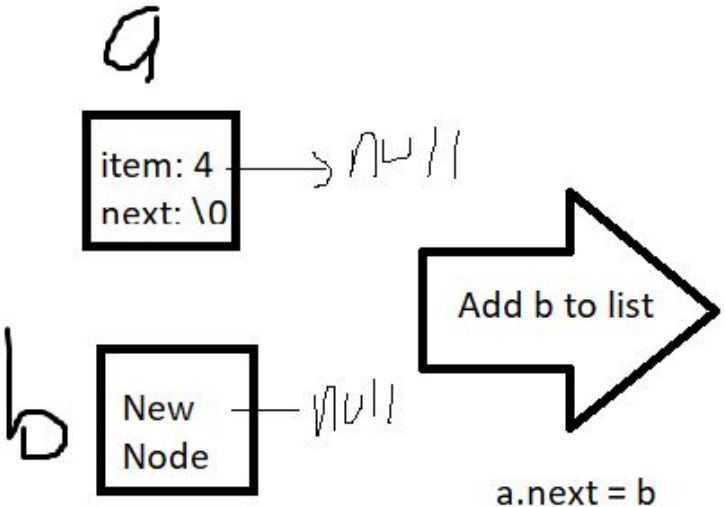
Empty linked list

a.item = 4



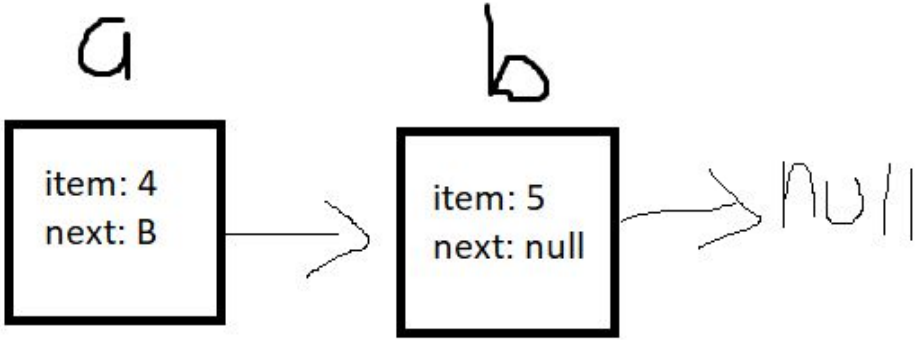
item: 4
next: null

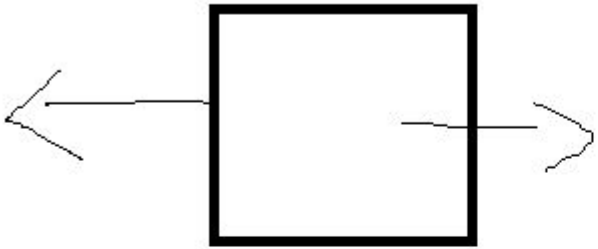
Single node in list



item: 5
next:null

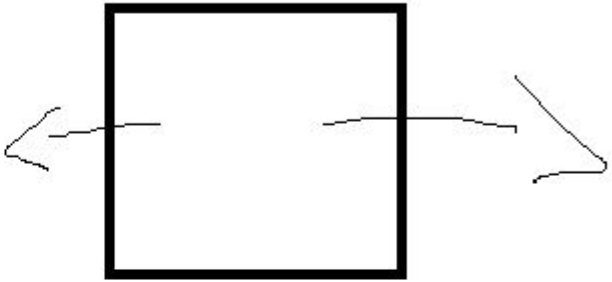
a.next = b





Node first:
item: null
next: null
prev: null

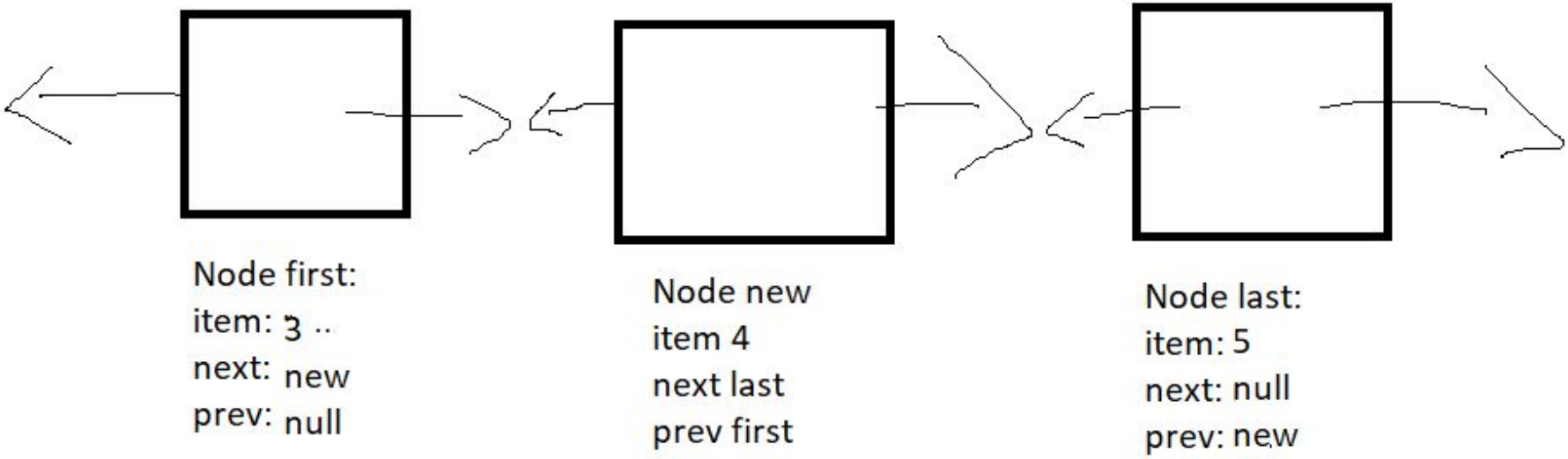
=



Node last:
item: null
next: null
prev: null

Deque starts with first and last nodes being the same

When you add a 2nd element, first and last are no longer the same


Node first:
item: 3 ..
next: new
prev: null
Node new
item 4
next last
prev first
Node last:
item: 5
next: null
prev: new

Peek

```
Item peekFirst()
```

- Return the item at the front of the deque.

```
Item peekLast()
```

- Return the item at the back of the deque.

Remove first or last

Make the first or last elements the next element in line. Set appropriate next and previous.

`Iterator<Item> iterator()`

- Return an object of type `DequeIterator`.

`LinkedDeque :: DequeIterator.`

- Instance variable:
  - Reference to current node in the iterator, `Node current`.
- `DequeIterator()`
  - Initialize instance variable appropriately.
- `boolean hasNext()`
  - Return whether the iterator has more items to iterate or not.
- `Item next()`
  - Return the item in `current` and advance `current` to the next node.

Problem 2

Implement a program called Sort.java that accepts strings from standard input, stores them

in a LinkedDeque data structure, sorts the deque, and writes the sorted strings to standard output

Tools you'll need to use

- StdIn
- Deque
- Basic Intuitive sorting
  - Least -> Greatest
  - Left -> right
- Stack is imported for a reason
  - Good tool to use during your sorting

Problem 3

Problem 3. (*Random Queue*) A random queue is similar to a stack or queue, except that the item removed is chosen uniformly at random from items in the data structure. Create a generic, iterable data type called `ResizingArrayRandomQueue` that uses a resizing array to implement the following random queue API:

`ResizingArrayRandomQueue`	
`ResizingArrayRandomQueue()`	constructs an empty random queue
`boolean isEmpty()`	returns `true` if this queue is empty, and `false` otherwise
`int size()`	returns the number of items in this queue
`void enqueue(Item item)`	adds *item* to the end of this queue
`Item sample()`	returns a random item from this queue
`Item dequeue()`	removes and returns a random item from this queue
`Iterator<Item> iterator()`	returns an independent† iterator to iterate over the items in this queue in random order
`String toString()`	returns a string representation of this queue

Constructor + Instance Variables

Instance variables:

- Array to store the items of queue, `Item[] q`.
- Size of the queue, `int n`.

We are using Item so it can hold any data type

`ResizingArrayRandomQueue()`

- Initialize instance variables appropriately — create `q` with an initial capacity of 2.

Empty? Size?

`boolean isEmpty()`

- Return whether the queue is empty or not.

`int size()`

- Return the size of the queue.

Enqueue -> Adding to the Resizing Array

```
void enqueue(Item item)
```

- If `q` is at full capacity, resize it to twice its current capacity.
- Insert the given item in `q` at index `n`. (Basically add it to the end)
- Increment `n` by one.

Takes new size

```
// Resizes the underlying array.
private void resize(int max) {
    Item[] temp = (Item[]) new Object[max];
    for (int i = 0; i < n; i++) {
        if (q[i] != null) {
            temp[i] = q[i];
        }
    }
    q = temp;
}
```

Dequeue

- `Item dequeue()`
  - Save `q[r]` in `item`, where `r` is a random integer from the interval `[0, n)`.
  - Set `q[r]` to `q[n - 1]`, and `q[n - 1]` to `null`.
  - If `q` is at quarter capacity, resize it to half its current capacity.
  - Decrement `n` by one.
  - Return `item`.

Sample

`Item sample()`

– Return `q[r]`, where `r` is a random integer from the interval `[0, n)`.

STDRANDOM!

`ResizingArrayRandomQueue :: RandomQueueIterator()`

- Instance variables:
  - Array to store the items of `q`, `Item[] items`.
  - Index of the current item in `items`, `int current`.
- `RandomQueueIterator()`
  - Create `items` with capacity `n`.
  - Copy the `n` items from `q` into `items`.
  - Shuffle `items`. Stdrandom?
  - Initialize `current` appropriately.
- `boolean hasNext()`
  - Return whether the iterator has more items to iterate or not.
- `Item next()`
  - Return the item in `items` at index `current` and advance `current` by one.

`Iterator<Item> iterator()`

- Return an object of type `RandomQueueIterator`.

terator

Problem 4

Problem 4. (*Sampling Integers*) Implement a program called `Sample.java` that accepts lo (int), hi (int), k (int), and *mode* (String) as command-line arguments, uses a random queue to sample k integers from the interval $[lo, hi]$, and writes the samples to standard output. The sampling must be done with replacement if *mode* is "+", and without replacement if *mode* is "-". You may assume that $k \leq hi - lo + 1$.

The program should run in time $T(k, n) \sim kn$ in the worst case (sampling without replacement), where k is the sample size and n is the length of the sampling interval.

- Accept lo (int), hi (int), k (int), and $mode$ (String) as command-line arguments.
- Create a random queue q containing integers from the interval $[lo, hi]$. *Cough Prob 3*
- If $mode$ is "+" (sampling with replacement), sample and write k integers from q to standard output.
- If $mode$ is "-" (sampling without replacement), dequeue and write k integers from q to standard. output

Sampling with replacement:
Sample q, look at it, put it back in the bag. Sample again

Allows you to get repeats

Sampling without replacement:
Sample q, look at it, throw it away.
Sample again