Lab 07: The List ADT and the Sieve of Eratosthenes

CS 0445: Data Structures

TAs: Jon Rutkauskas
Brian Nixon
http://db.cs.pitt.edu/courses/cs0445/current.term/

October 28, 2019
University of Pittsburgh, Pittsburgh, PA

The List Data Structure

- Like arrays:
  - Ordered
  - Allows duplicates
  - Allows arbitrary access to elements
  - Access elements by indexing
- Unlike arrays:
  - No fixed capacity
  - Grows dynamically as elements are inserted
  - Allows arbitrary insertions without manual shifting
  - No guarantee of random access (fast indexing)

List Methods – Creating and Deleting Entries

- `void add(E newEntry)` – Adds `newEntry` to the end of the list
- `void add(int newPosition, E newEntry)` – Adds `newEntry` at `newPosition`, shifting everything else down the list
- `E remove(int position)` – Removes and returns the item at `position`
- `void clear()` – Clears everything from the list

List Methods – Retrieving and Modifying Entries

- `E set(int position, E newEntry)` – Sets the entry at position to `newEntry`, returning the old item.
- `E get(int position)` – Returns the entry at position

List Methods – Additional Methods

- `boolean contains(E entry)` – Returns **`true`** if the list contains `entry`, **`false`** otherwise
- `int indexOf(E entry)` – Returns the index of `entry`
- `int getSize()` – Returns the number of items in the list
- `boolean isEmpty()` – Returns **`true`** if there are no items in the list, **`false`** otherwise
- `E[] toArray()` – Returns an array of all entries in the list
  – This method runs in O(n) time and O(n) memory, copying to a new array. Try to use other list methods instead unless this is absolutely needed.

Demonstration: List Methods

- Let's say we have a list named `myList`, and perform the following operations on it:

```
myList.add(A)
myList.add(B)
myList.add(C)
myList.add(D)
myList.add(2,F)
myList.set(3,G)
myList.remove(1)
```

Demonstration: List Methods

```
myList.add(A)
```

0)	A

Demonstration: List Methods

`myList.add(A)`

0)	A

`myList.add(B)`

0)	A
1)	B

Demonstration: List Methods

`myList.add(A)`

0) A

`myList.add(B)`

0) A
1) B

`myList.add(C)`

0) A
1) B
2) C

Demonstration: List Methods

`myList.add(A)`	`myList.add(B)`	`myList.add(C)`	`myList.add(D)`
0) A	0) A	0) A	0) A
	1) B	1) B	1) B
		2) C	2) C
			3) D

Demonstration: List Methods

`myList.add(A)`

0) A

`myList.add(B)`

0) A
1) B

`myList.add(C)`

0) A
1) B
2) C

`myList.add(D)`

0) A
1) B
2) C
3) D

`myList.add(2,F)`

0) A
1) B
2) F
3) C
4) D

Demonstration: List Methods

`myList.add(A)`

0) A

`myList.add(B)`

0) A
1) B

`myList.add(C)`

0) A
1) B
2) C

`myList.add(D)`

0) A
1) B
2) C
3) D

`myList.add(2,F)`

0) A
1) B
2) F
3) C
4) D

`myList.set(3,G)`

0) A
1) B
2) F
3) G
4) D

Demonstration: List Methods

`myList.add(A)`

0) A

`myList.add(B)`

0) A
1) B

`myList.add(C)`

0) A
1) B
2) C

`myList.add(D)`

0) A
1) B
2) C
3) D

`myList.add(2,F)`

0) A
1) B
2) F
3) C
4) D

`myList.set(3,G)`

0) A
1) B
2) F
3) G
4) D

`myList.remove(1)`

0) A
1) F
2) G
3) D

Implementing Lists

- **Array** List

```
public class ArrayList<E> implements ListInterface<E> {

    private E[] list;
    private int size;

    …
}
```

0	1	2	3	4	5	6
24	26	27	28	29	30	24

Implementing Lists

- **Linked** List

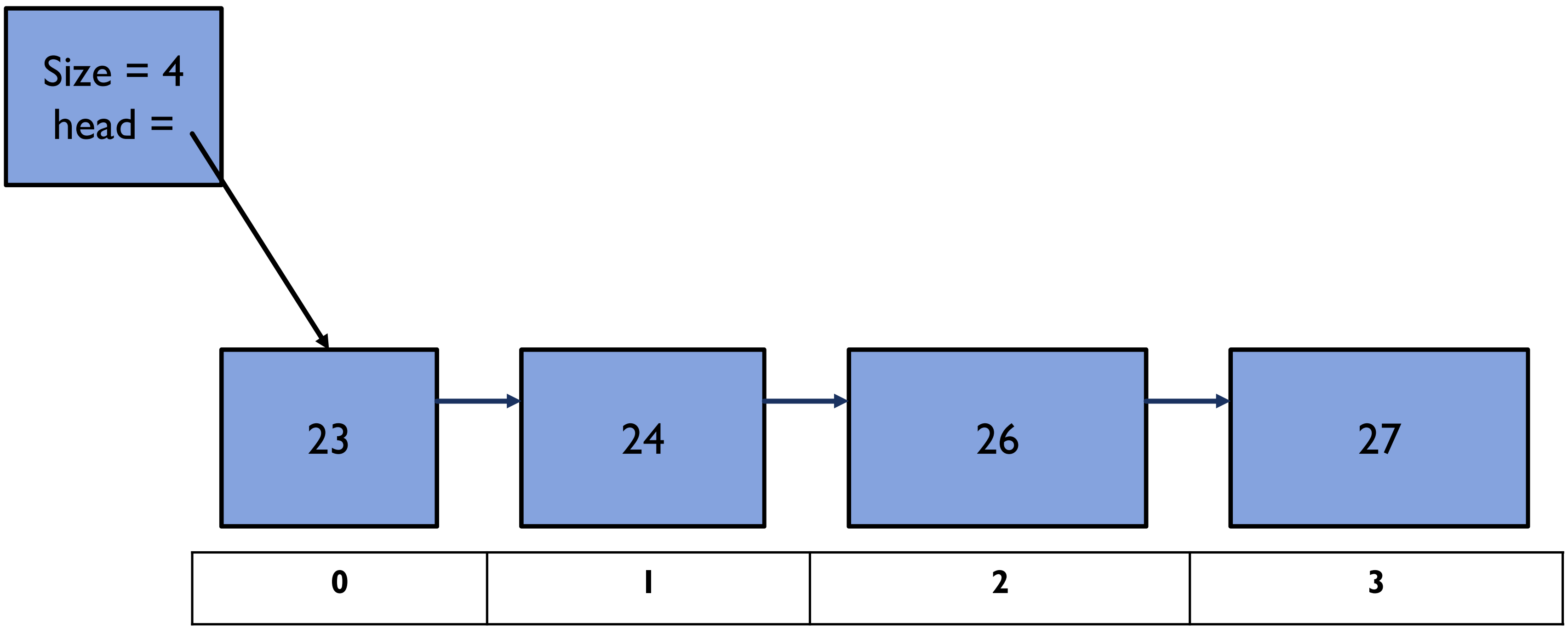
```
public class LinkedList<E> implements ListInterface<E> {

      private Node head;
      private int size;

      public LinkedList() {
         head = null;
         size = 0;
      }
      …
}
```

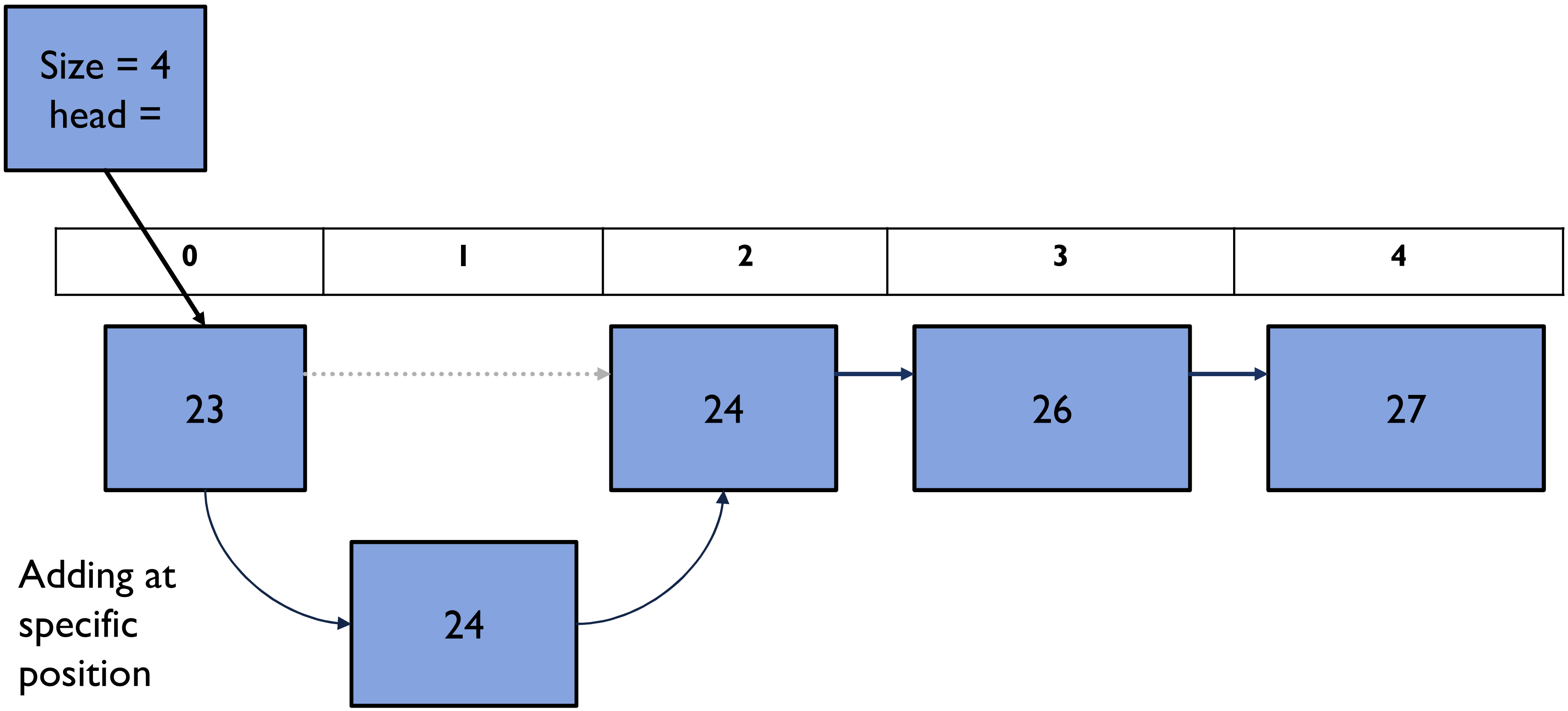
Implementing Lists

- **Linked** List



Implementing Lists

- **Linked** List



Implementing Lists – Differences in Implementation

Array List

- Contiguous in memory

+ O(1) get and set

■ Insertions and deletions require shifting

- Need to resize the array when full and have empty spaces at end of array for holding new entries

Linked List

+ Does not require contiguous memory

- O(n) get and set

■ Insertions require traversal to location, but no shifting

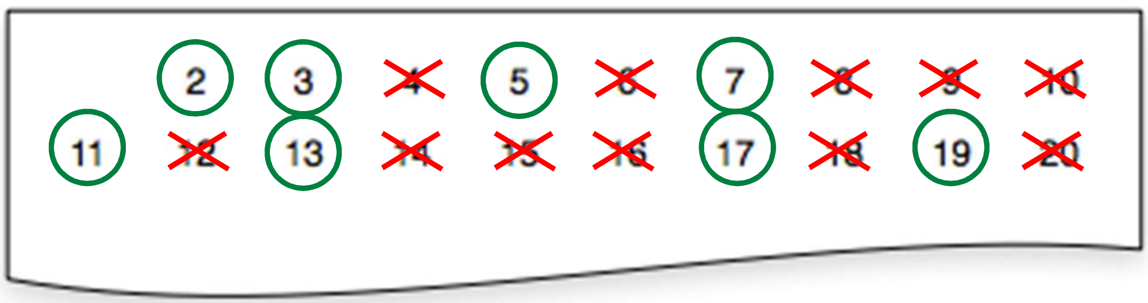
+ No resizing

- Requires additional memory to store `next` reference

Which is better? – Depends on performance and memory requirements and specific usage

The Sieve of Eratosthenes

- Algorithm to find the prime numbers less than a given number n.
  - When a prime is found, cross out the multiples of that prime
  - Only need to test multiples up to $\sqrt{n}$, but need to have enumerated all numbers up to *n*.



The Sieve of Eratosthenes

- Example: Primes less than 25
  - Only need to test multiples up to 5 since $\sqrt{25} = 5$

2	3	4	5	6	7	8	9
10	11	12	13	14	15	16	17
18	19	20	21	22	23	24	25

Write down all numbers 2 – 25

The Sieve of Eratosthenes

- Example: Primes less than 25
  - Only need to test multiples up to 5 since $\sqrt{25} = 5$

2	3	4	5	6	7	8	9
10	11	12	13	14	15	16	17
18	19	20	21	22	23	24	25

Start at the first number, cross out all of its multiples

The Sieve of Eratosthenes

- Example: Primes less than 25
  - Only need to test multiples up to 5 since $\sqrt{25} = 5$

2	3	~~4~~	5	~~6~~	7	~~8~~	9
~~10~~	11	~~12~~	13	~~14~~	15	~~16~~	17
~~18~~	19	~~20~~	21	~~22~~	23	~~24~~	25

Start at the first number, cross out all of its multiples

The Sieve of Eratosthenes

- Example: Primes less than 25
  - Only need to test multiples up to 5 since $\sqrt{25} = 5$

Move to the next non-crossed-out number and cross-out all of its multiples

2	3	~~4~~	5	~~6~~	7	~~8~~	~~9~~
~~10~~	11	~~12~~	13	~~14~~	~~15~~	~~16~~	17
~~18~~	19	~~20~~	~~21~~	~~22~~	23	~~24~~	25

The Sieve of Eratosthenes

- Example: Primes less than 25
  - Only need to test multiples up to 5 since $\sqrt{25} = 5$

Continue to the next non-crossed-out number and cross-out all of its multiples

2	3	~~4~~	5	~~6~~	7	~~8~~	~~9~~
~~10~~	11	~~12~~	13	~~14~~	~~15~~	~~16~~	17
~~18~~	19	~~20~~	~~21~~	~~22~~	23	~~24~~	~~25~~

The Sieve of Eratosthenes

- Example: Primes less than 25
  - Only need to test multiples up to 5 since $\sqrt{25} = 5$

2	3	~~4~~	5	~~6~~	7	~~8~~	~~9~~
~~10~~	11	~~12~~	13	~~14~~	~~15~~	~~16~~	17
~~18~~	19	~~20~~	~~21~~	~~22~~	23	~~24~~	~~25~~

Stop after $\sqrt{n}$. All numbers that have not been crossed out are prime

The Sieve of Eratosthenes - Approach

- 2 general approaches using lists:
  1. Create a list of numbers, **adding** all numbers up to n, and **removing** the multiples as you determine they are not prime
  2. Create a list of n Booleans. A *true* at position 11 would mean that 11 is prime, while a *false* at position 25 would mean 25 is composite, for example. So, set positions to *false* as you determine that number is composite. You would need to do extra work at the end to convert this into a list of prime integers.
- Think about which approach would be more efficient and why?

Lab Instructions

- Download the Lab 7 instructions and Provided Code from the course website
  - http://db.cs.pitt.edu/courses/cs0445/current.term/
- Complete Sieve of Eratosthenes on paper to determine the primes under 60
- Implement `primesUpTo(int max)` using an ArrayList and either of the approaches we discussed
  - ArrayList.java is already implemented and given to you