# Honors Data Structures

Lecture 2: Lists.

02/24/2020

Daniel Bauer

# Data Types in Programming Languages

- All languages have data types. Most object-oriented languages include the concept of a **type hierarchy** (different types are related by inheritance).

- Static vs. dynamic typing.

- We will study the type systems in Java and Scala -- other modern OOP languages are similar.

# Data Types

# Data Types

- Basic data types: booleans, bytes, integers, floats, characters…

- Simple abstractions: arrays, `String`

- More complex, structured data types (**this course**): Lists, Stacks, Trees, Sets, Graphs

# Abstract Data Types

- An Abstract Data Type (ADT) is an collection of data together with a set of operations.

- ADT specification *does not mention how* operations are implemented.

- Example:

  - `Set` ADT might provide “add”, “remove”, “contains”, “union”, and “intersection” operations.

# ADTs vs. Data Structures

- A data *type* is a well-defined collection of data with a well-defined set of operations on it.

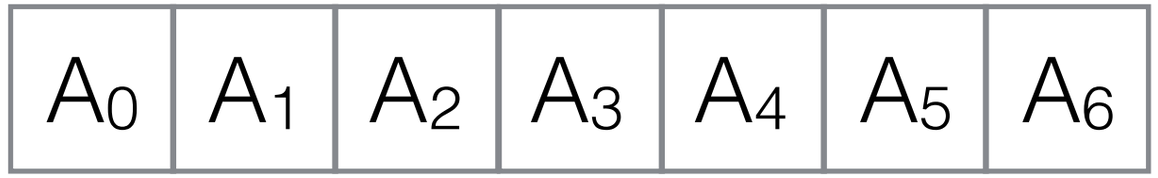
- A *data structure* is an actual implementation of a particular abstract data type.

# The List ADT

- A list *L* is a sequence of *N* objects $A_0$, $A_1$, $A_2$, ..., $A_{N-1}$
- *N* is the length/size of the list. List with length N=0 is called the *empty list.*
- $A_i$ *follows/succeeds* $A_{i-1}$ for i > 0.
- $A_i$ *precedes* $A_{i+1}$ for i < N.

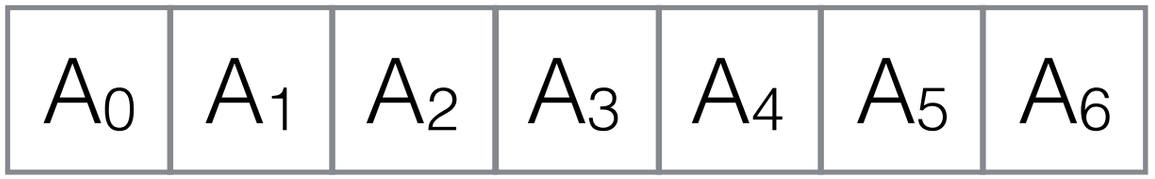
| $A_0$ | $A_1$ | $A_2$ | $A_3$ | $A_4$ | $A_5$ | $A_6$ |
|---|---|---|---|---|---|---|

# Typical List Operations



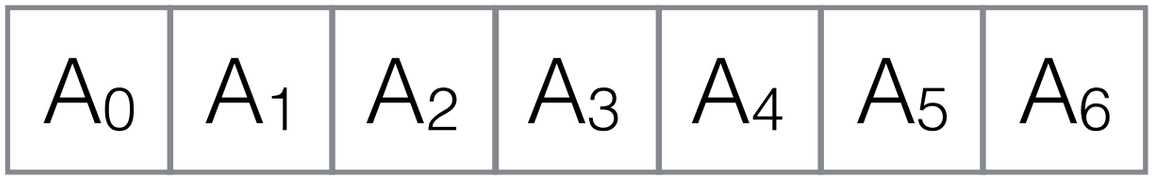
# Typical List Operations

- `void printList()`



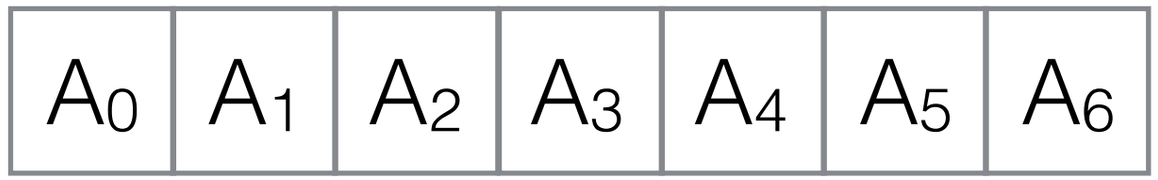
# Typical List Operations

- `void printList()`
- `void makeEmpty()`



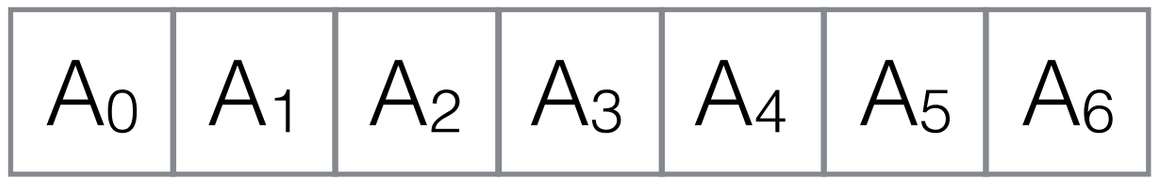
# Typical List Operations

- `void printList()`
- `void makeEmpty()`
- `int size()`



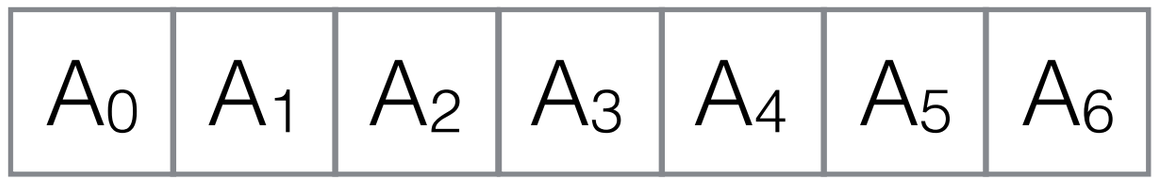
# Typical List Operations

- `void printList()`
- `void makeEmpty()`
- `int size()`
- `Object findKth(k) / get(k)`



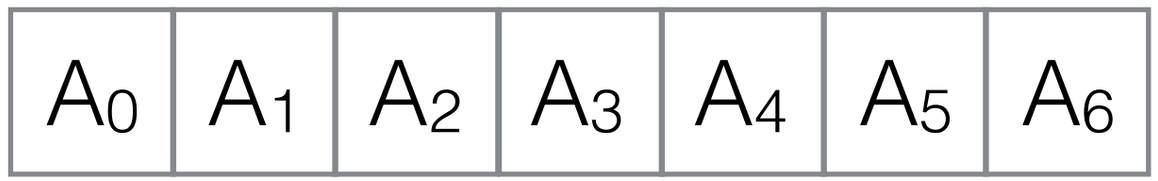
# Typical List Operations

- `void printList()`
- `void makeEmpty()`
- `int size()`
- `Object findKth(k) / get(k)`
- `boolean insert(x, k), append(x)`



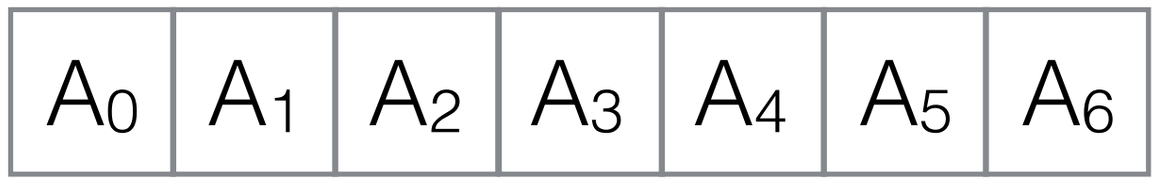
# Typical List Operations

- `void printList()`
- `void makeEmpty()`
- `int size()`
- `Object findKth(k) / get(k)`
- `boolean insert(x, k), append(x)`
- `boolean remove(k)`

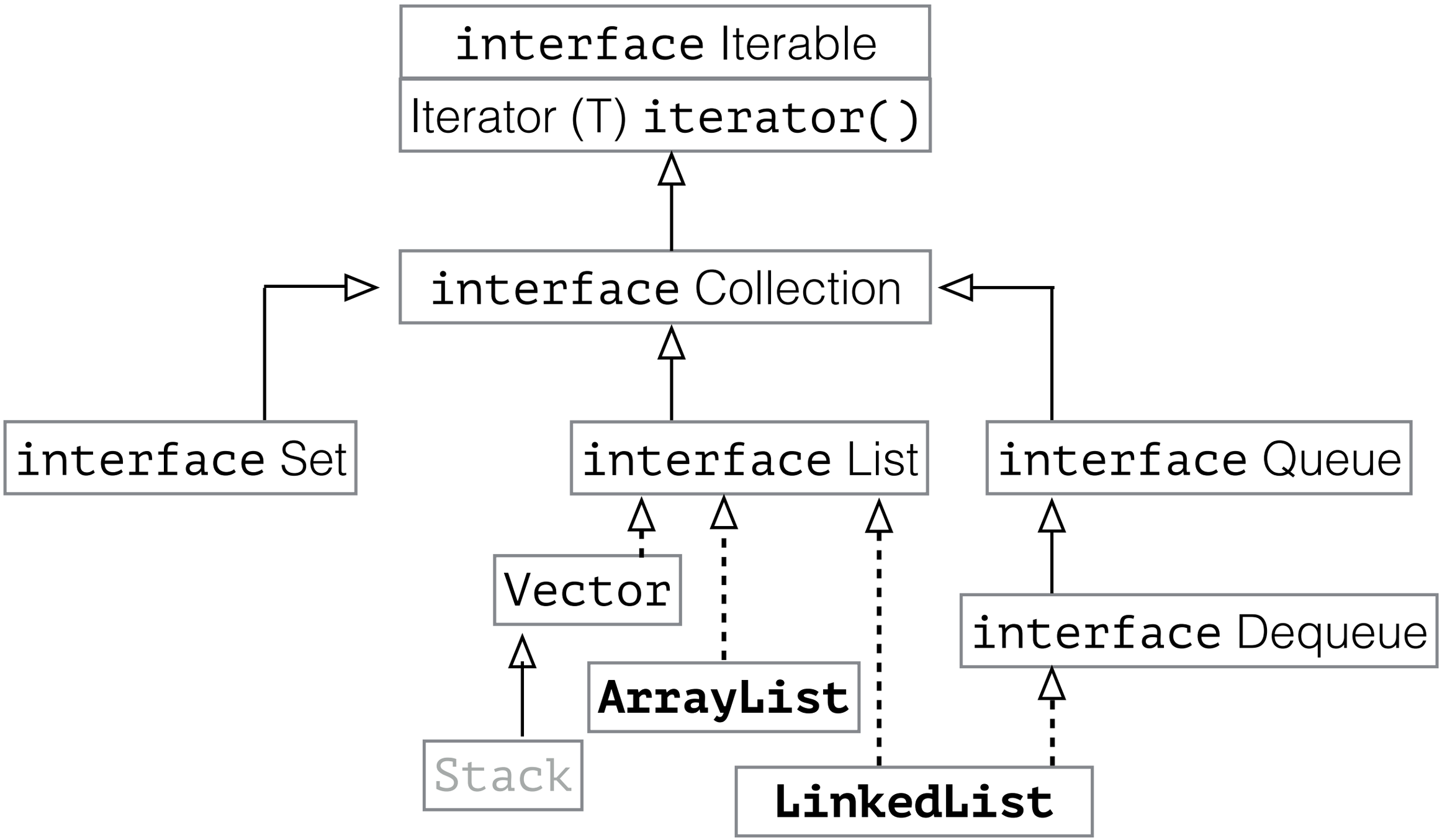


# Typical List Operations

- `void printList()`
- `void makeEmpty()`
- `int size()`
- `Object findKth(k) / get(k)`
- `boolean insert(x, k), append(x)`
- `boolean remove(k)`
- `int find(x) / indexOf(x)`



# Lists in the Java API




https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/util/Collection.html

# The Java Collection API

```
package java.util;

interface Collection<E> extends Iterable<E> {
    boolean add(E e);
    boolean addAll(Collection<? extends E> c);
    void clear();
    boolean contains(Object o);
    boolean containsAll(Collection<?> c);
    boolean isEmpty();
    Iterator<E> iterator(); // via Iterable
    boolean remove(Object o);
    boolean removeAll(Collection<?> c);
    boolean retainAll(Collection<?> c);
    int size();
    Object[] toArray();
    <T> T[] toArray(T[] a);
}
```

https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/util/Collection.html

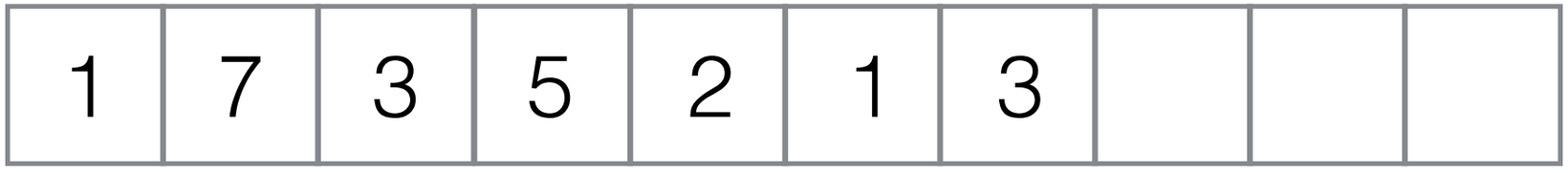
# Array Lists

- Just a thin layer wrapping an array.

```
public class ArrayList {

    public static final int DEFAULT_CAPACITY = 10;
    private int theSize;
    private Integer[] theItems;

}
```



# Generic Classes

- We typically do not know what kind of object to expect in a data structure.

- Java allows to add *type parameters (<> syntax)* to the definitions of classes. Such classes are called *generic classes.*

```
public class MyArrayList<AnyType> {
    private AnyType[] theItems;
    ...
    public AnyType get(int idx) { ... }
    public boolean add(int idx, AnyType x) { ... }
}
```

# Generic Classes (2)

- Type parameters make it possible to create a new instance of data structure that stores objects of specific data types (and their sub-types).

```
List<Integer> l = new MyArrayList<Integer>();
```

# Generic Classes (2)

- Type parameters make it possible to create a new instance of data structure that stores objects of specific data types (and their sub-types).

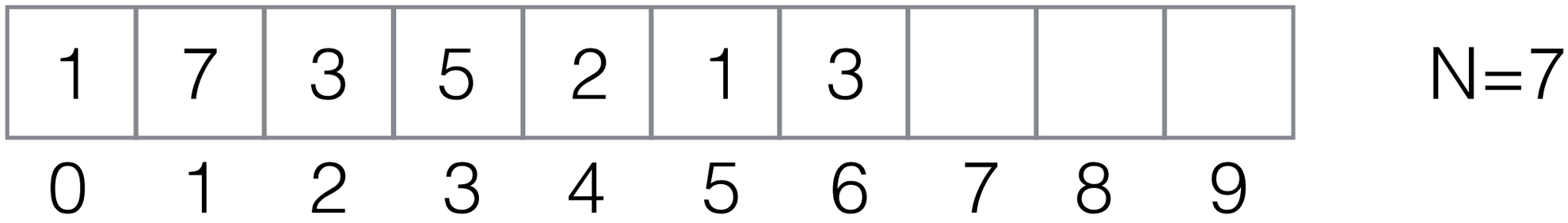
```
List<Integer> l = new MyArrayList<Integer>();
```

- In Java ≥7, this can be simplified using the <> (Diamond) operator:

```
List<Integer> l = new MyArrayList<>();
```

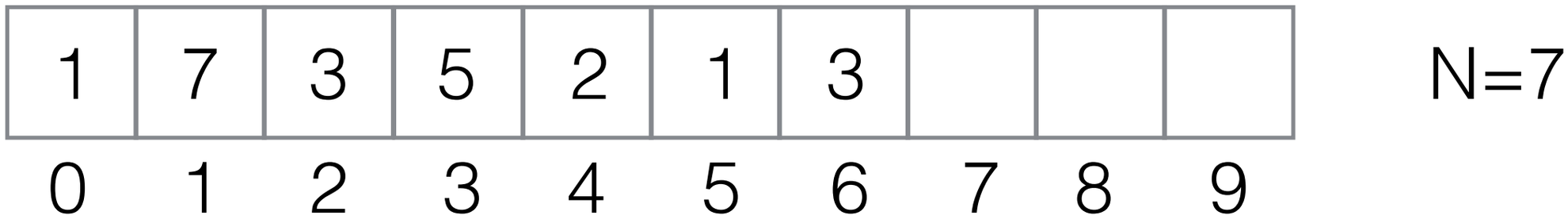
- Type of `l` is inferred automatically.

# Running Time for Array List Operations



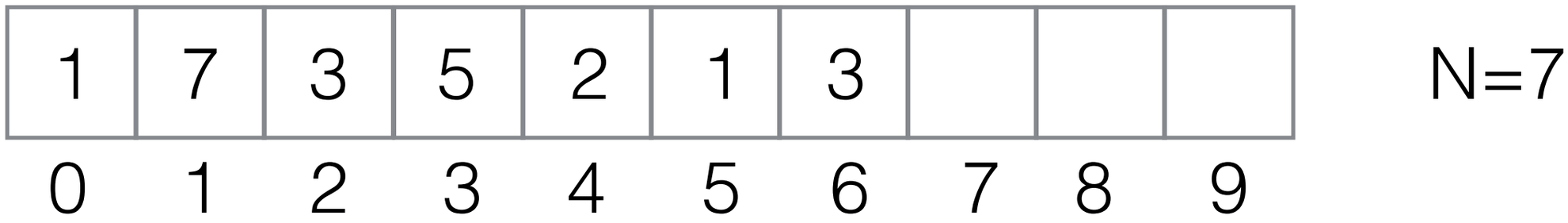

| Operation | Number of Steps |
|---|---|
| printList | |
| find(x) | |
| findKth(k) | |
| insert(x,k) | |
| remove(x) | |

# Running Time for Array List Operations



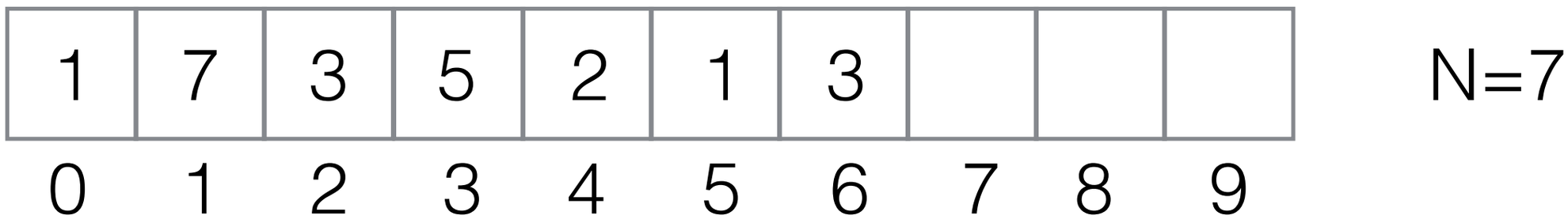

| Operation | Number of Steps |
|---|---|
| printList | N |
| find(x) | N |
| findKth(k) | |
| insert(x,k) | |
| remove(x) | |

# Running Time for Array List Operations



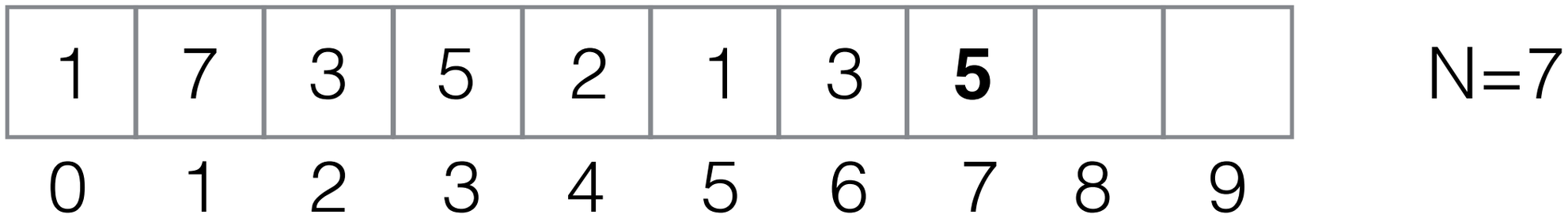

| Operation | Number of Steps |
|---|---|
| printList | N |
| find(x) | N |
| findKth(k) | 1 |
| insert(x,k) | |
| remove(x) | |

# Array List: Insert/Remove




| | |
|---|---|
| insert(x,k) | |
| remove(x) | |

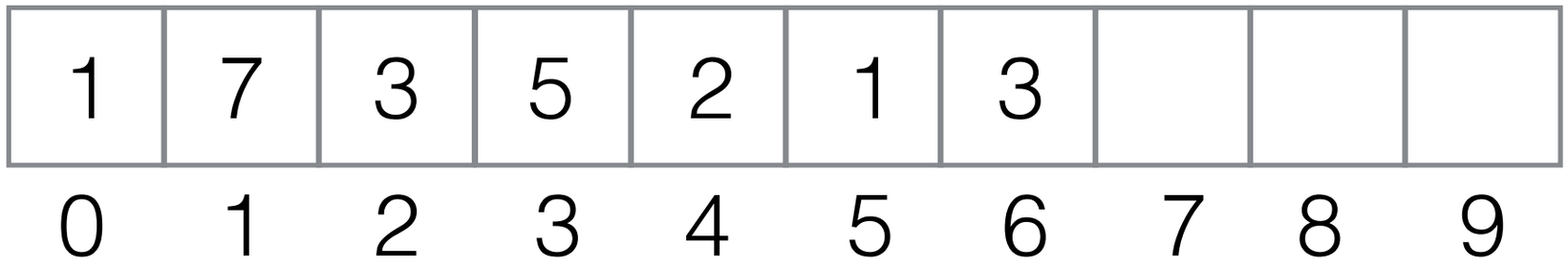
# Array List: Insert/Remove




insert(5,7): 1 step

| | |
|---|---|
| insert(x,k) | |
| remove(x) | |

# Array List: Insert/Remove




N=7

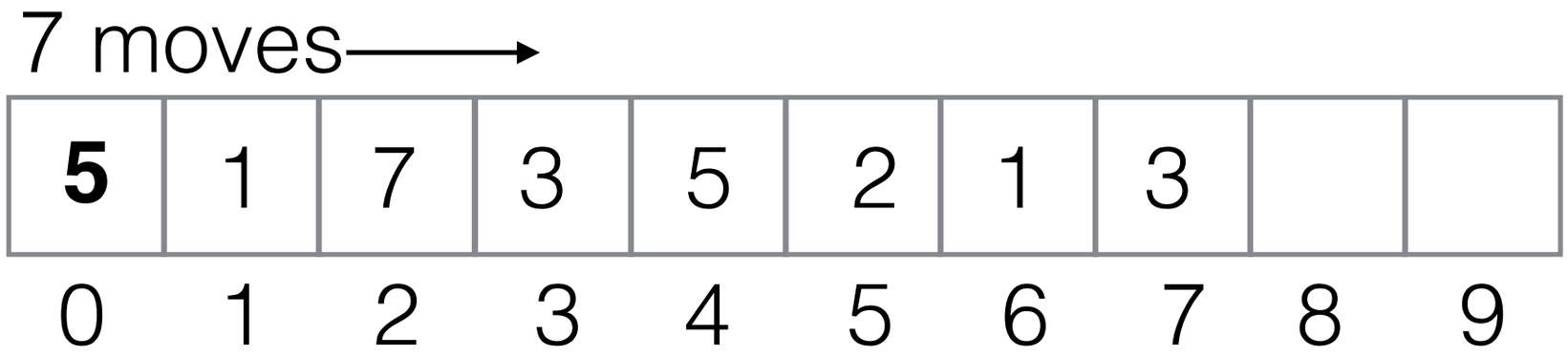
insert(5,7): 1 step

remove(7): 1 step

best case

| | |
|---|---|
| insert(x,k) | |
| remove(x) | |

# Array List: Insert/Remove




insert(5,7): 1 step

remove(7): 1 step

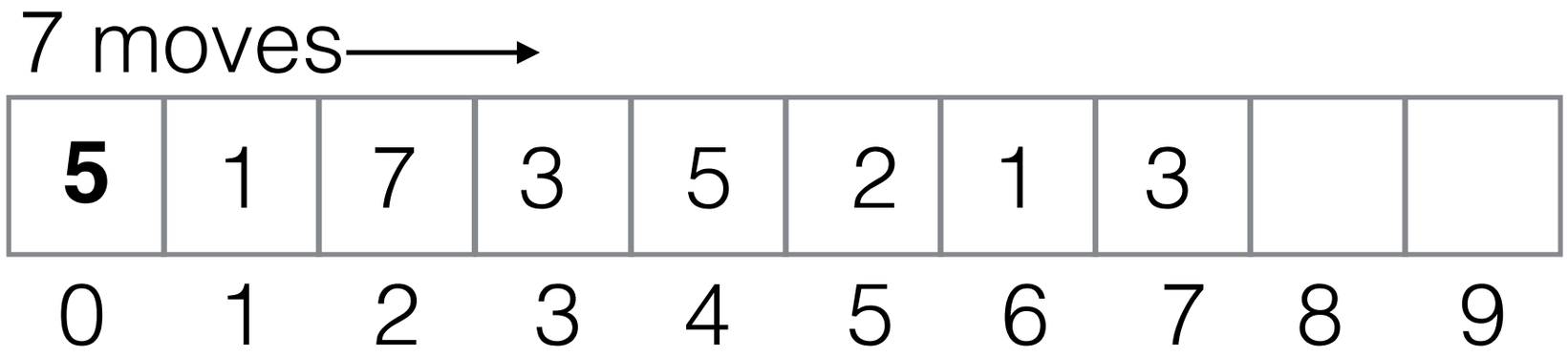
best case

insert(5,0): 7 steps

worst case

| | |
|---|---|
| insert(x,k) | |
| remove(x) | |

# Array List: Insert/Remove




insert(5,7): 1 step

remove(7): 1 step

best case

insert(5,0): 7 steps

remove(0): O(N)

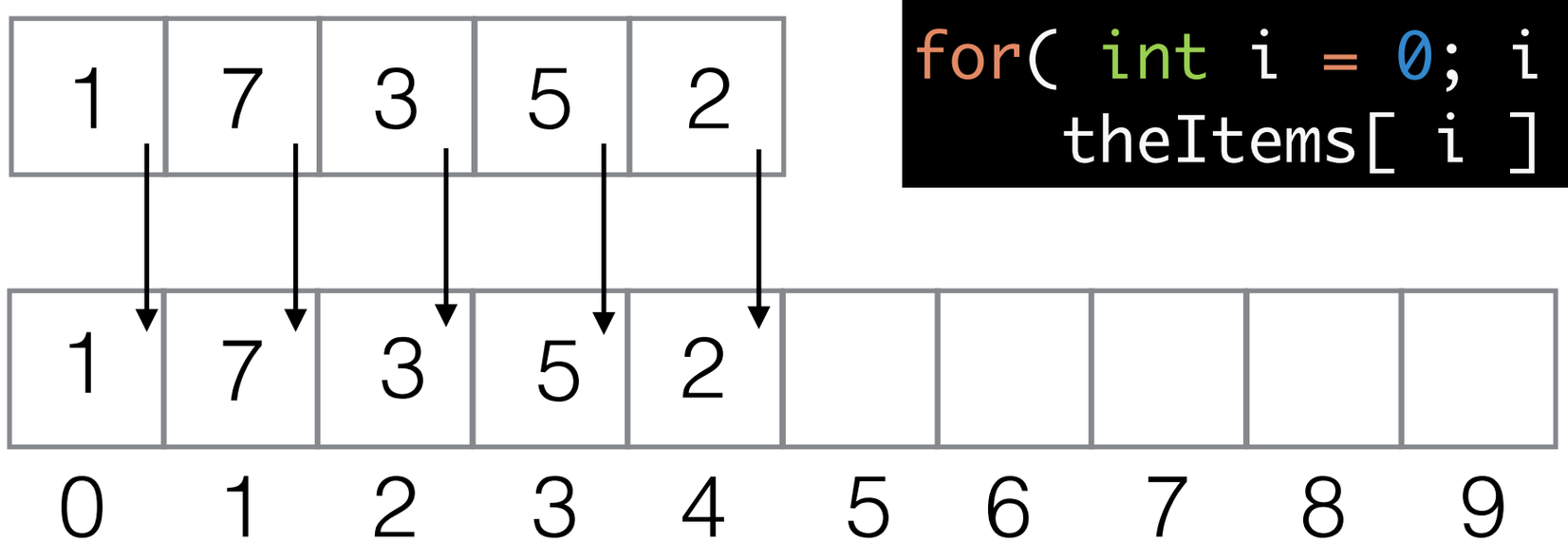
worst case

| | |
|---|---|
| insert(x,k) | N |
| remove(x) | N |

# Expanding Array Lists

- What if we are running out of space during `append/insert`
- first copy all elements into a new array of sufficient size

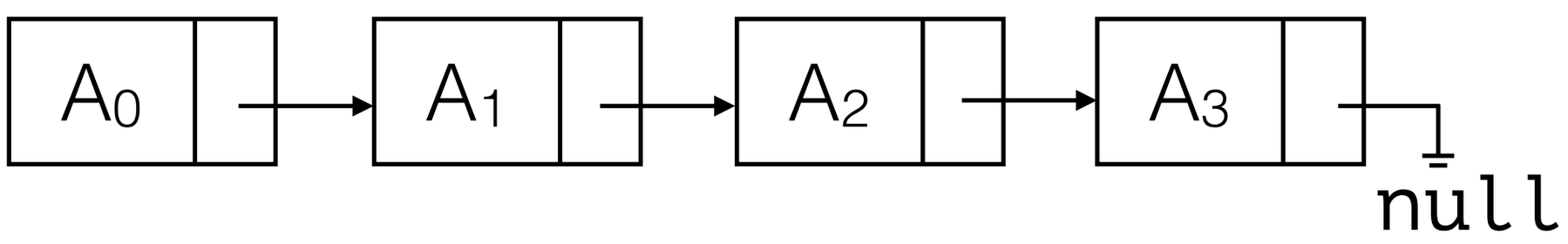
```
newCapacity = arr.length * 2;
Integer[ ] old = theItems;
theItems = new Integer[newCapacity];
for( int i = 0; i < size( ); i++ )
    theItems[ i ] = old[ i ];
```



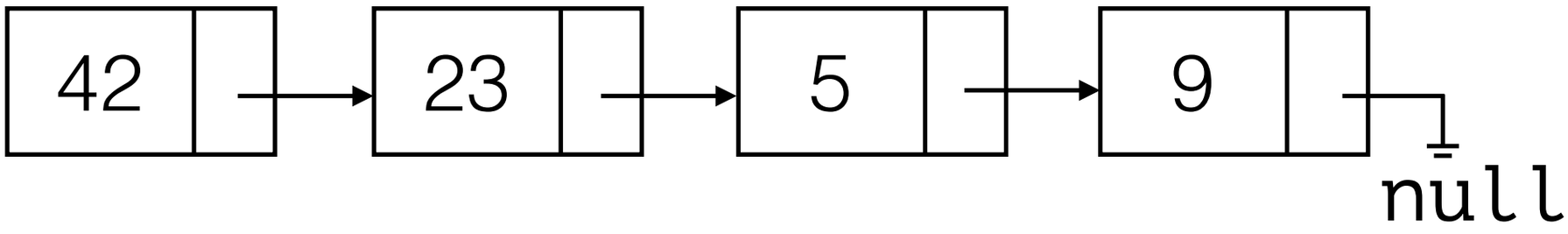
# Simple Linked Lists

- Series of *Nodes*. Each *Node* contains:
  - A reference to the data object it contains.
  - A reference to the next node in the List.

```
private static class Node<T> {
    public T data;
    public Node next;
    public Node(Integer d, Node<T> n) {
        data = d;
        next = n;
    }
}
```

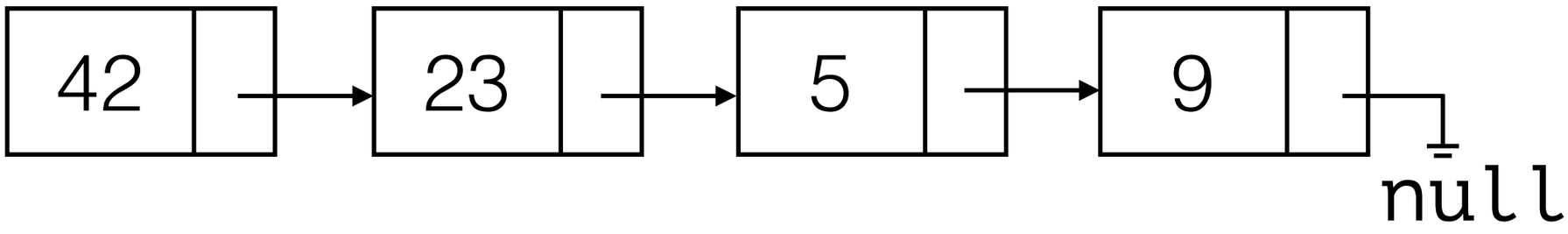


# Running Time for Simple Linked List Operations



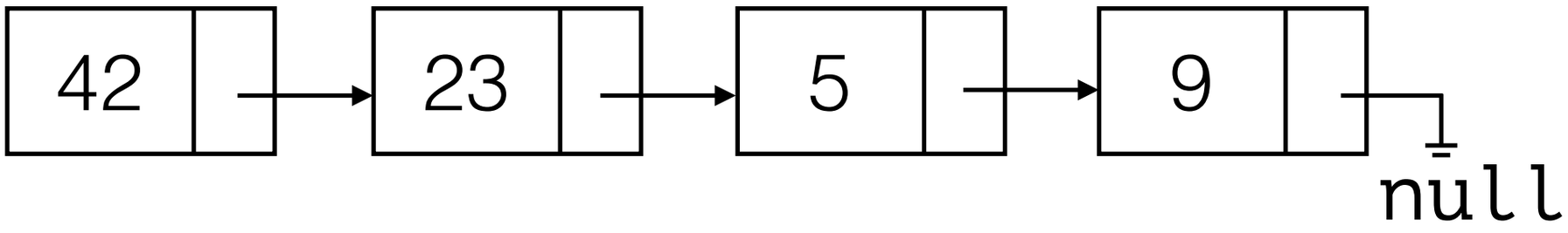

| printList | |
|---|---|
| find(x) | |
| findKth(k) | |
| next() | |

# Running Time for Simple Linked List Operations



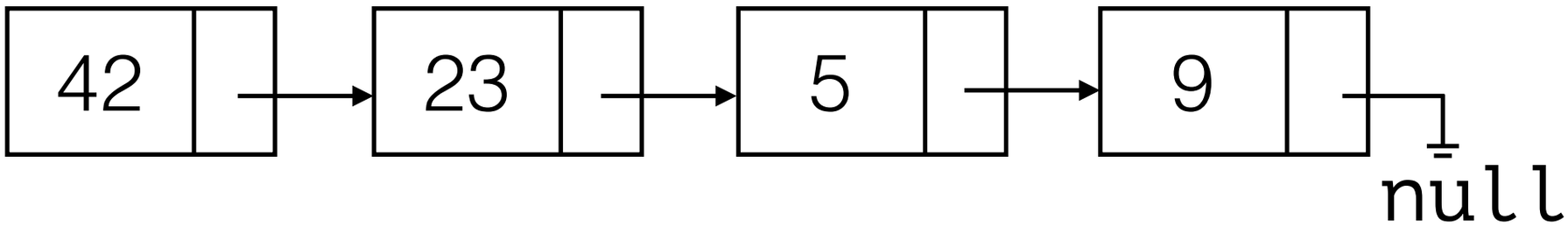

| printList | N |
|---|---|
| find(x) | |
| findKth(k) | |
| next() | |

# Running Time for Simple Linked List Operations



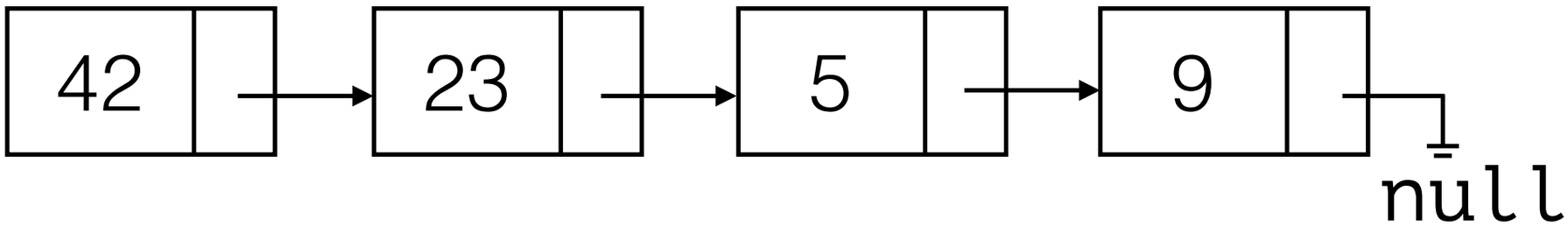

| printList | N |
|---|---|
| find(x) | N |
| findKth(k) | |
| next() | |

# Running Time for Simple Linked List Operations




| printList | N |
|---|---|
| find(x) | N |
| findKth(k) | k |
| next() | |

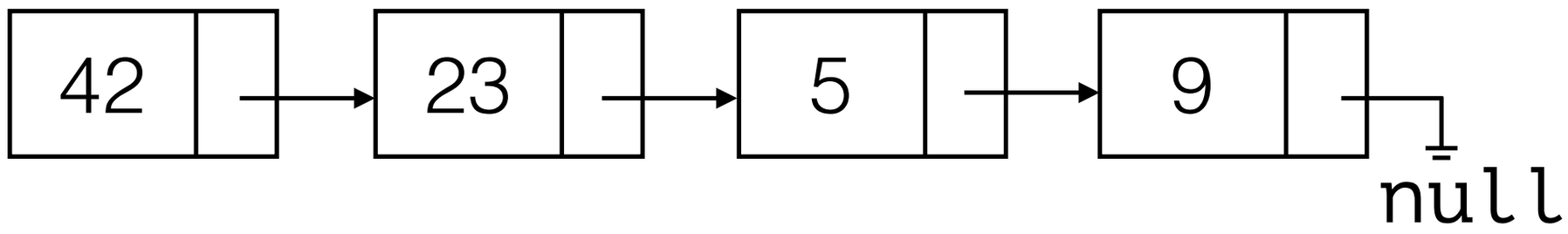
# Running Time for Simple Linked List Operations




| printList | N |
|---|---|
| find(x) | N |
| findKth(k) | k |
| next() | 1 |

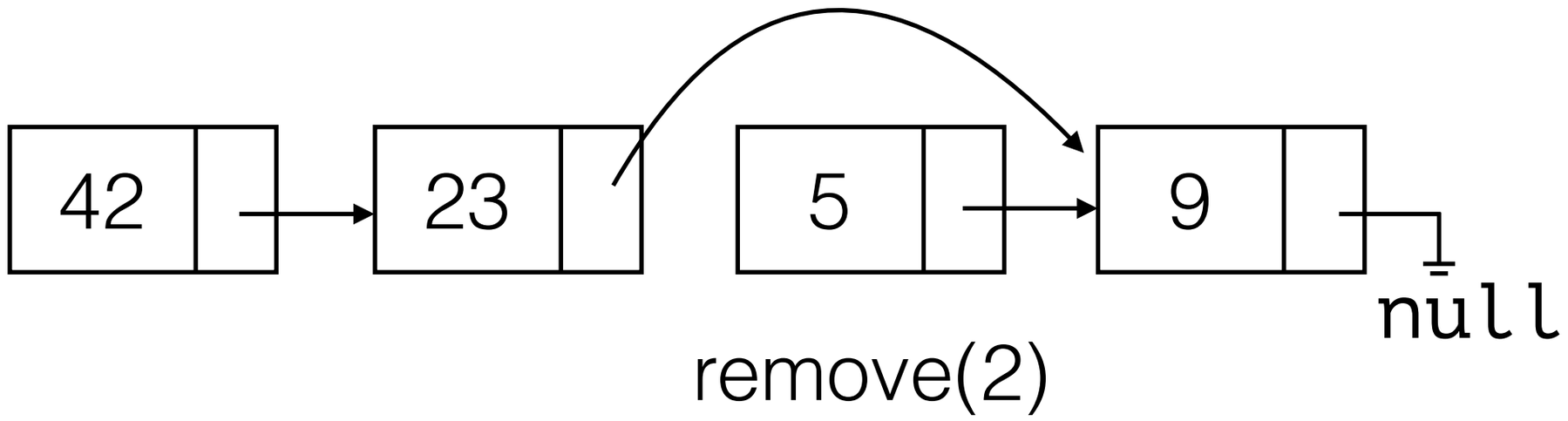
In many applications we can use next() instead of findKth(k).
(for every element in the list do… / filter the list … )

# Simple Linked List Removal



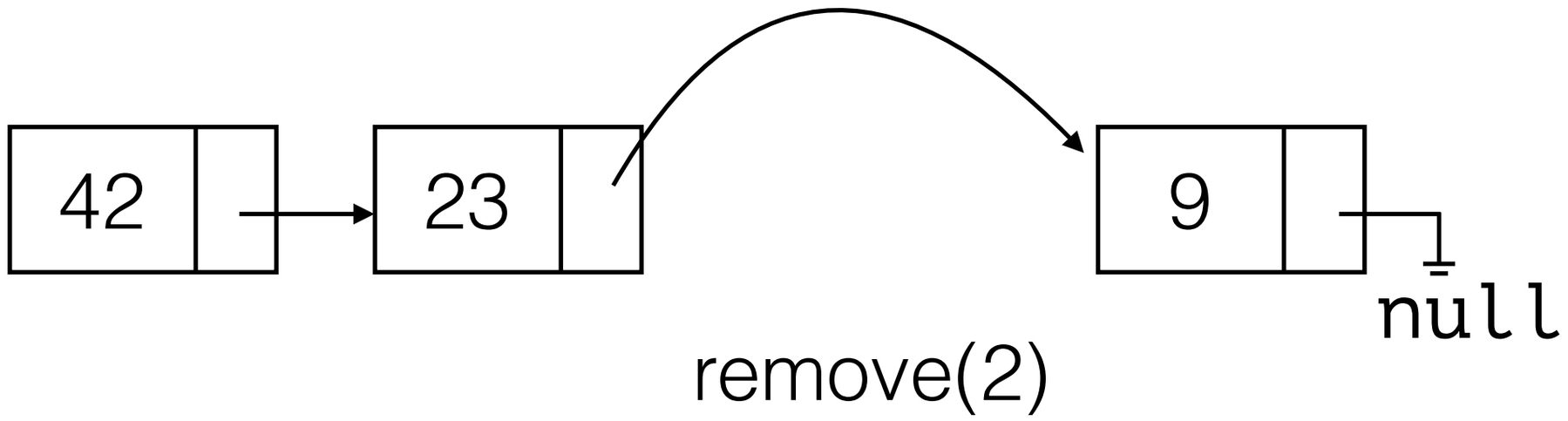

| findKth(k) | k |
|---|---|
| next() | 1 |
| insert(x,k) | ??? |
| remove(k) | |

# Simple Linked List Removal



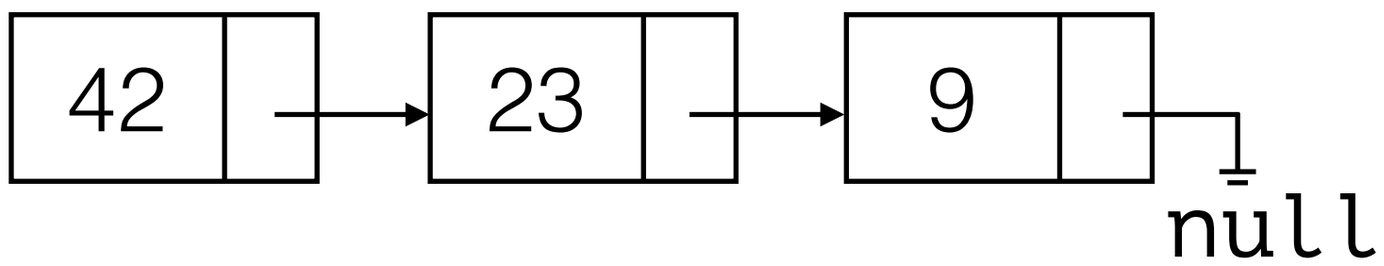

| findKth(k) | k |
|---|---|
| next() | 1 |
| insert(x,k) | ??? |
| remove(k) | search time + 1 |

# Simple Linked List Removal



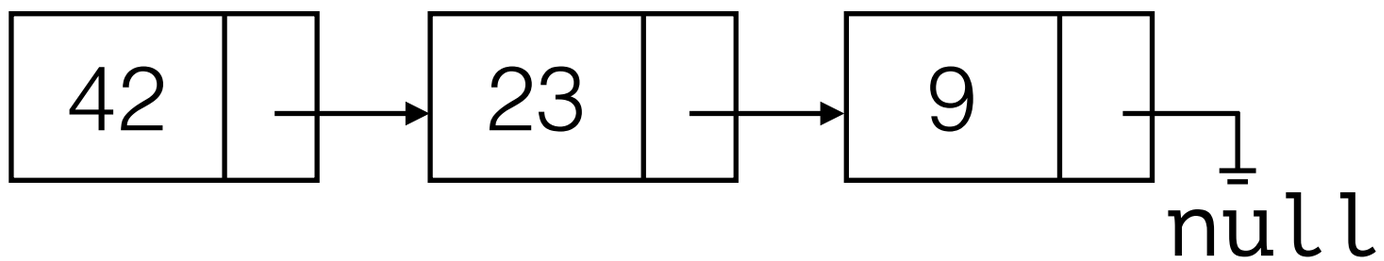

| findKth(k) | k |
|---|---|
| next() | 1 |
| insert(x,k) | ??? |
| remove(k) | search time + 1 |

# Simple Linked List Insertion




| insert(x,k) | |
|---|---|

# Simple Linked List Insertion




insert(5,2)

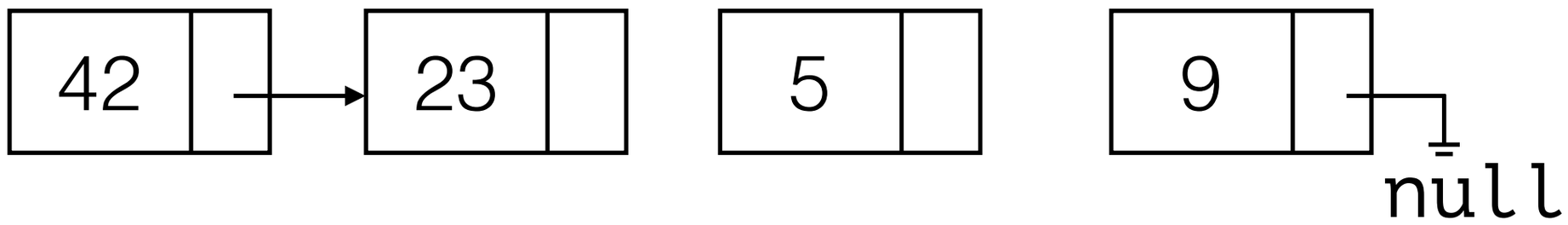
| insert(x,k) | |
|---|---|

# Simple Linked List Insertion


insert(5,2)

| insert(x,k) | |
|---|---|

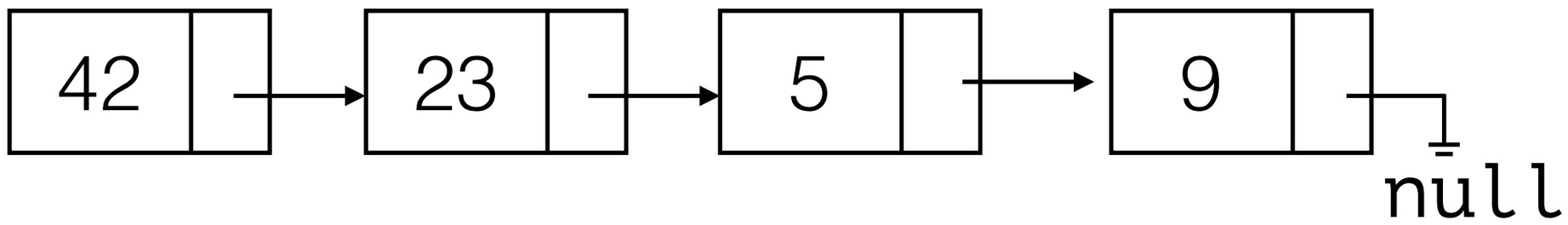
# Simple Linked List Insertion




insert(5,2)

| insert(x,k) | |
|---|---|

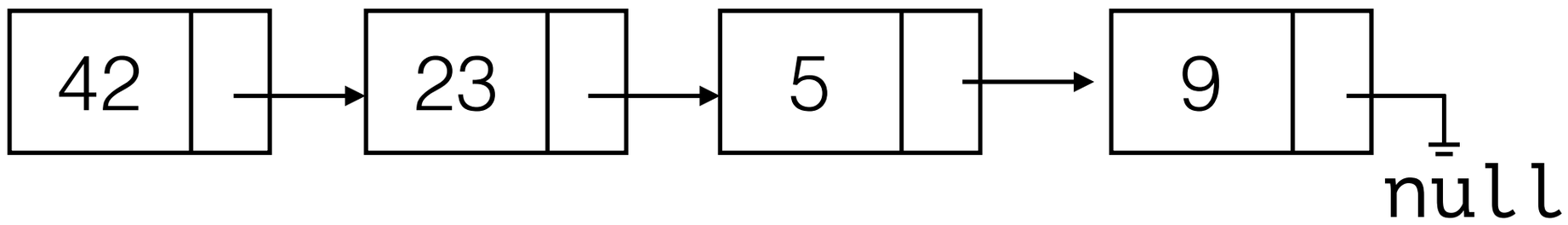
# Simple Linked List Insertion




insert(5,2)

| insert(x,k) | |
|---|---|

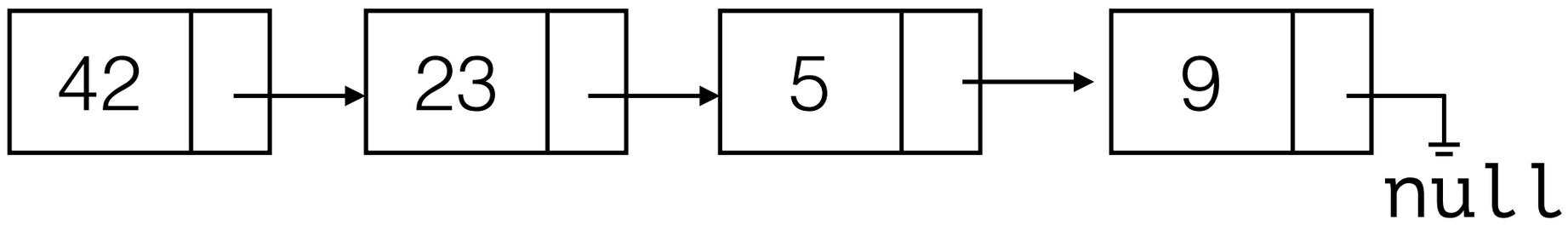
# Simple Linked List Insertion




insert(5,2)

| insert(x,k) | search time +1 |
| --- | --- |

# Simple Linked List Insertion

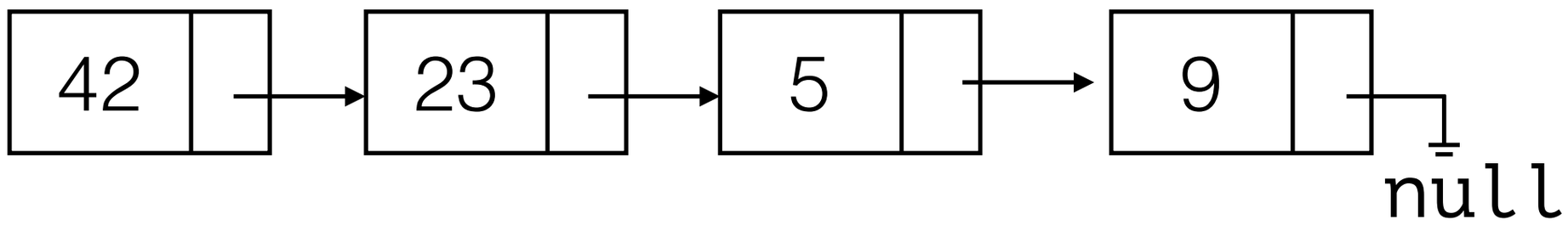



insert(5,2)

| insert(x,k) | search time +1 |
|---|---|

Inserting in position 0?
Inserting in position N-1?

# Simple Linked List Insertion




insert(5,2)

| insert(x,k) | search time +1 |
|---|---|

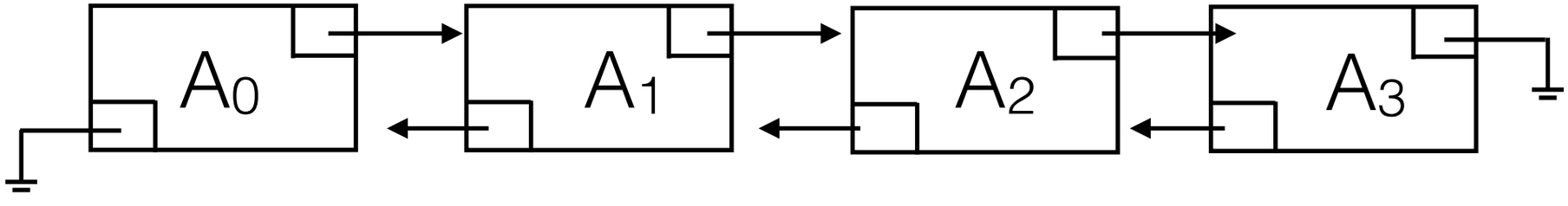
Inserting in position 0?
Inserting in position N-1?

Linked list should remember the first and last object.

# Doubly Linked Lists

- Also maintain reference to previous node in the list.
- Speeds up append at end of list.

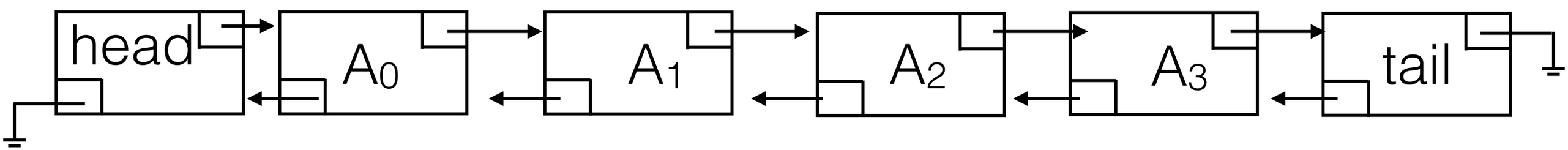
```
private static class Node<AnyType> {
    public AnyType data;
    public Node next;
    public Node prev;
    public Node(Node<AnyType> d, Node n,
                Node<AnyType> p) {
        data = d; next = n; prev = n;
    }
}
```



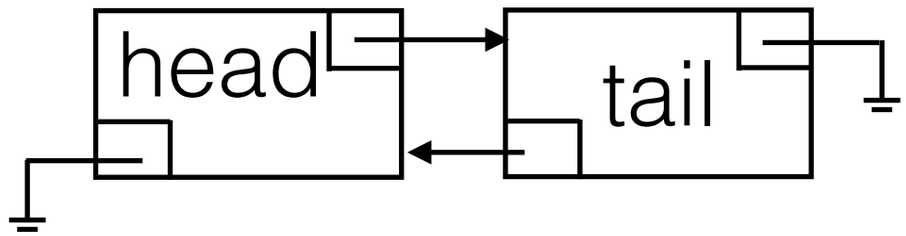
# Doubly Linked List with Sentinel Nodes

- header node, tail node
- make implementation of `next` / `previous` easier.
- Remove other special cases

(e.g. removing first node/last node)



# Empty Doubly Linked List with Sentinel Nodes



# For-Each Loops

- `Iterables` support special Java syntax

```
for (T item : someIterable) {
    System.out.println(item.toString());
}
```

- No need to explicitly get the `Iterator` and call `next()` repeatedly.

- This is also called an enhanced for-loop.

# Java Iterators

```
package java.lang;

interface Iterator<T> {
    boolean hasNext();
    T next();
    void remove();
}
```

Our List implementation should be compatible with the Iterator interface.

https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/util/Iterator.html

# The Iterable Interface

```
package java.lang;

interface Iterable<T> {
    Iterator<T> iterator();
}
```

- Using Iterables and Iterators.

```
Iterator<T> someIterator = someIterable.iterator()

while (someIterator.hasNext()) {
    T nextItem = someIterator.next();
    System.out.println(nextItem.toString());
}
```

https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/lang/Iterable.html

# The Iterable Interface

```
package java.lang;

interface Iterable<T> {
    Iterator<T> iterator();
}
```

- Using Iterables and Iterators.

```
Iterator<T> someIterator = someIterable.iterator()

while (someIterator.hasNext()) {
    T nextItem = someIterator.next();
    System.out.println(nextItem.toString());
}
```

- Never implement Iterable and Iterator in the same class!

https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/lang/Iterable.html

# Nested Classes

- Usually each Java class is defined in its own `.java` file.
- Sometimes classes have a specific purpose (in relation to another class) and are not used anywhere else.

```
class OuterClass {
    ...
    static class StaticNestedClass {
        ...
    }
    class InnerClass {
        ...
    }
}
```

# Inner Classes

```
public class MyArrayList<AnyType> implements Iterable<AnyType>{

 . . .

    public java.util.Iterator<AnyType> iterator( ) {
        return new ArrayListIterator( );
    }

    private class ArrayListIterator implements java.util.Iterator<AnyType> {
        private Node<AnyType> current = 0;
       . . .
    }
}
```

- Instances of inner classes can access instance members of the outer instance that created it.