

---

---

CS206

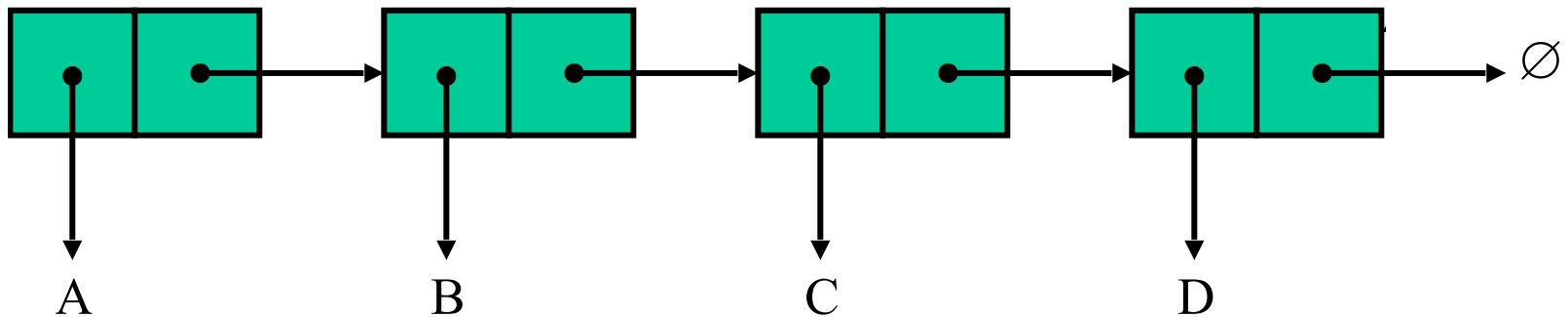
Linked Lists

---

# Linked List

---

- A linked list is a lists of objects.
- The objects form a linear sequence.
- The sequence is unbounded in length.
- Each object leads to the next



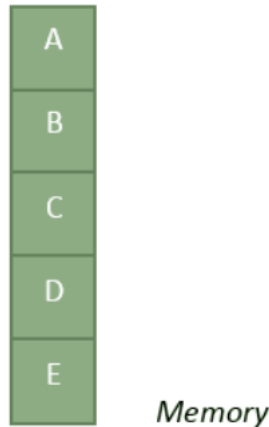
---

# Linked List, Array and ArrayList

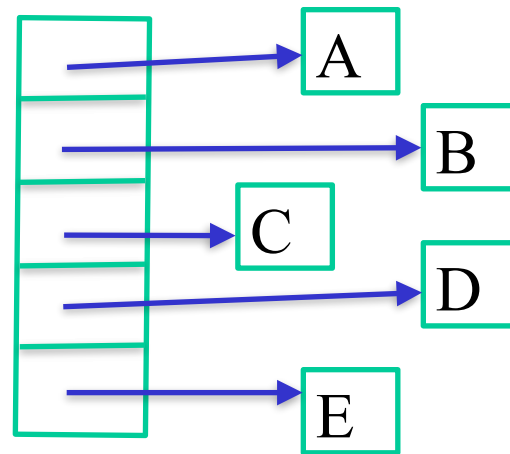
---

- An array is a single consecutive piece of memory, a linked list is made of many disjoint pieces (the linked objects).
- ArrayList is between(ish)

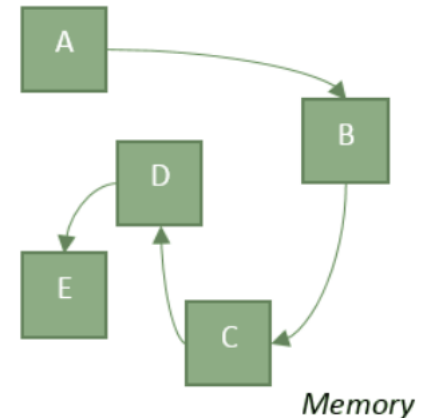
Array



ArrayList



Linked List



---

# Linked List versus Array

---

- Array
  - quick access to any element
  - slow insertion, deletion and reordering (shifting required in general)
- Linked list
  - quick insertion, deletion and reordering of the elements
  - slow access (must traverse list)

---

# Linked List Core

---

- the essential part of a linked list is a “self-referential” structure.
- That is, a class with an instance variable that holds a “reference” to another member of that same class
- For linked lists, this structure is usually called a Node

```
private class Node<J> {  
    public J data;  
    public Node<J> next;  
    public Node(J data, Node<J> next) {  
        this.data = data;  
        this.next = next;  
    }  
}
```

---

# References in Java (Review)

---

- A reference variable holds a memory address to where the referenced object is stored (not the object itself)
- Reference types
  - Anything that inherits from `Object` (including `String`, `Integer`, `Double`, etc)
  - “primitive” types: `int`, `float`, etc are NOT reference types
- A reference is `null` when it doesn't refer/point to any object

---

# References and equality (review)

---

```
public class ReferenceCheck {  
    public static void main(String[] args) {  
        String s1 = new String("abc");  
        String s2 = new String("abc");  
        String s3 = s2;  
        String s4 = "abc";  
        String s5 = "abc";  
  
        System.out.println("s1.equals(s2) " + s1.equals(s2));  
        System.out.println("s1==s2 " + (s1 == s2));  
        System.out.println("s1==s3 " + (s1 == s3));  
        System.out.println("s1==s4 " + (s1 == s4));  
        System.out.println("s2==s3 " + (s2 == s3));  
        System.out.println("s2==s3 " + (s2 == s4));  
        System.out.println("s3==s4 " + (s3 == s4));  
    }  
}
```

The “new” operator returns  
a reference to an object of the  
given type

equals should compare content!  
compareTo should compare content!

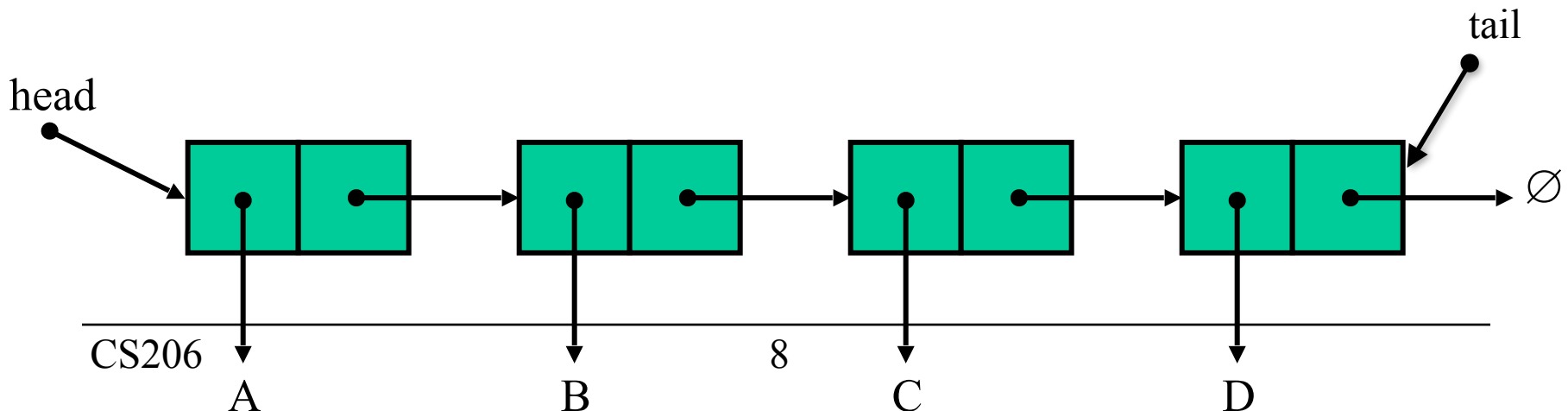
== compares memory location

---

# Heads and Tails

---

- Given that one thing leads to another in a LL, need a place to start
  - referred to as "head"
- If you know where the head is, you can get to everything in LL
  - So, when working with LL there is almost always a value called head (or front, or ...)
- Often it is useful to also have a value tail
  - not required, just really useful
- Q: How do you know when at end of LL?





---

# Linked List interface

---

```
public interface LinkedListInterface<J>
{
    int size();
    boolean isEmpty();
    J first();
    J last();
    void addLast(J c);
    void addFirst(J c);
    J removeFirst();
    J removeLast();
    boolean remove(J r);
}
```

No mention of nodes!!

But this still egregiously violates encapsulation!!

---

# Starting Point

---

```
public class LinkedList<J> implements LinkedListInterface<J>
{
    private class Node<H>
    {
        public H data;
        public Node<H> next;
        public Node(H data)
        {
            this.data = data;
            this.next = null;
        }
    }

    private Node<J> head = null;
```

---

# isEmpty() and first()

---

---

# Size

---

```
public int size() {
    int siz=0;
    Node<J> n = head;
    while (n!=null) {
        siz++;
        n= n.next;
    }
    return siz;
}
```

- Algorithmic Complexity (Big-O)?
- Can we improve? (yes, but you have to cheat)

---

# toString() for Linked List

---

```
public String toString() {
    StringBuffer sb = new StringBuffer();
    for (Node<J> node = head; node != null; node = node.next) {
        sb.append(node.data.toString());
        sb.append("\n");
    }
    return sb.toString();
}
```

---

```
public J last()
```

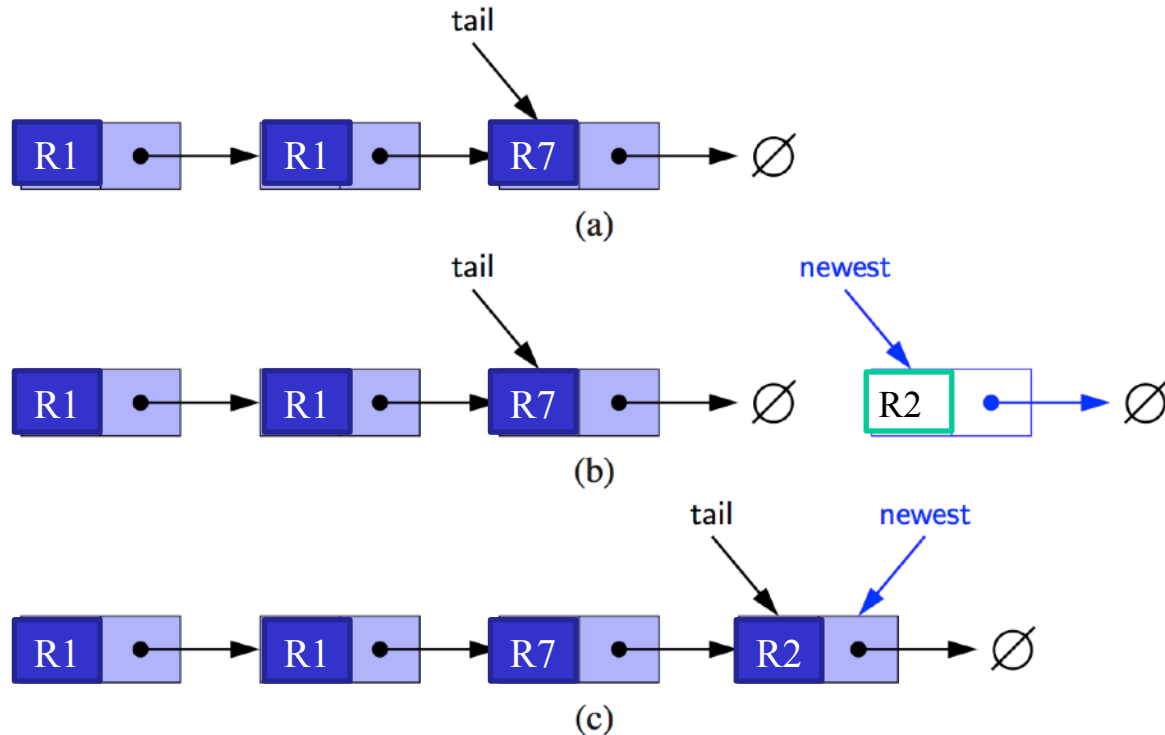
---

- Write in groups

Show my last with private utility function

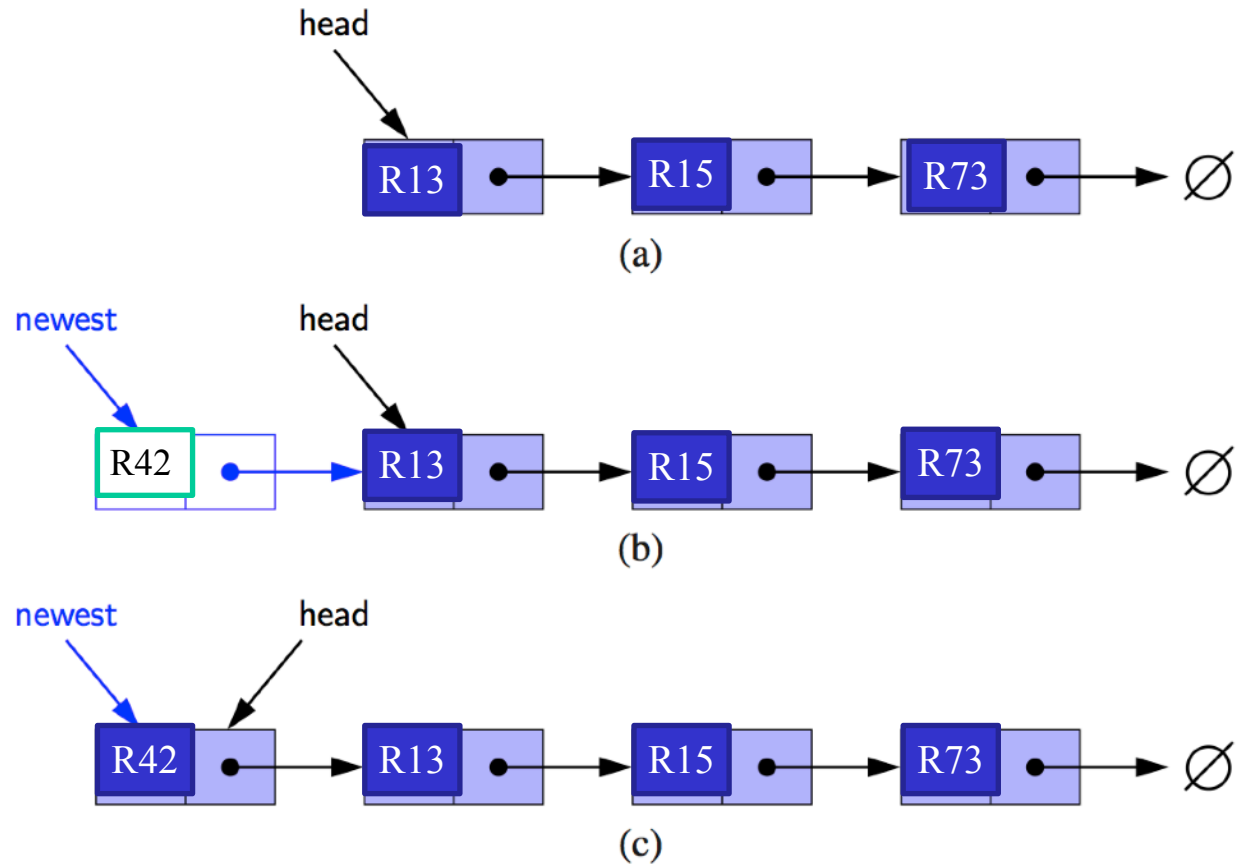
# Inserting at the Tail

1. Get to the end
  1.  $O(n)$
  2. Save time, add an instance variable "tail"
2. Create a new node
3. Have new node point to null
4. have old last node point to new node
5. update tail to point to new node



# Inserting at the Head

1. create a new node
2. have new node point to old head
3. update head to point to new node

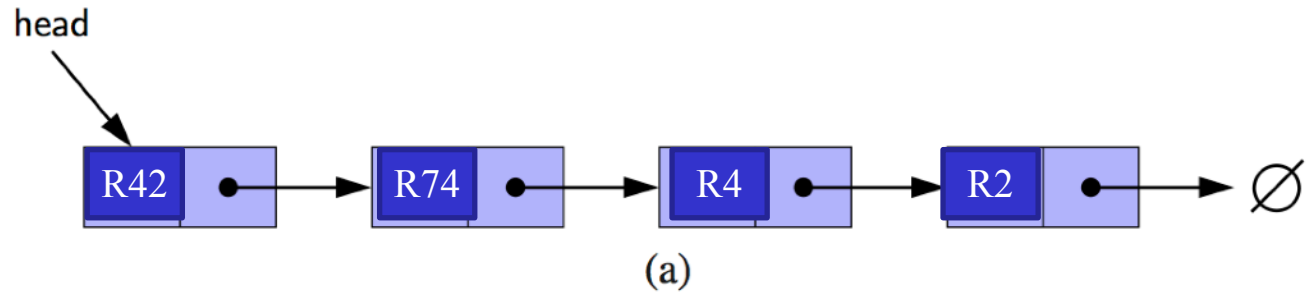


write addFirst at chalkboard

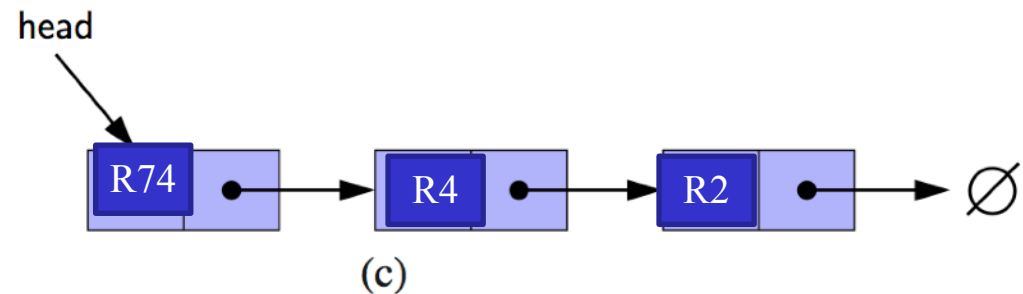
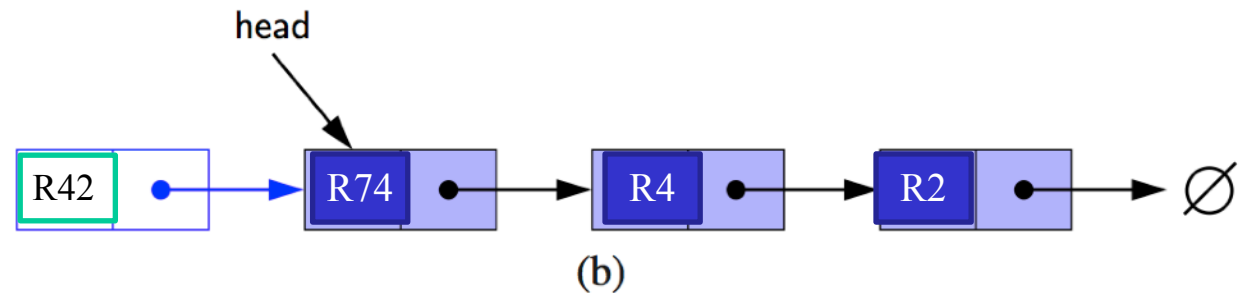


# Removing at the Head

1. update head to point to next node in the list



2. allow "garbage collector" to reclaim the former first node



---

```
void addLast(J c);  
void addFirst(J c);
```

---

Why not take a Node?

Write in groups

---

# Deletion

---

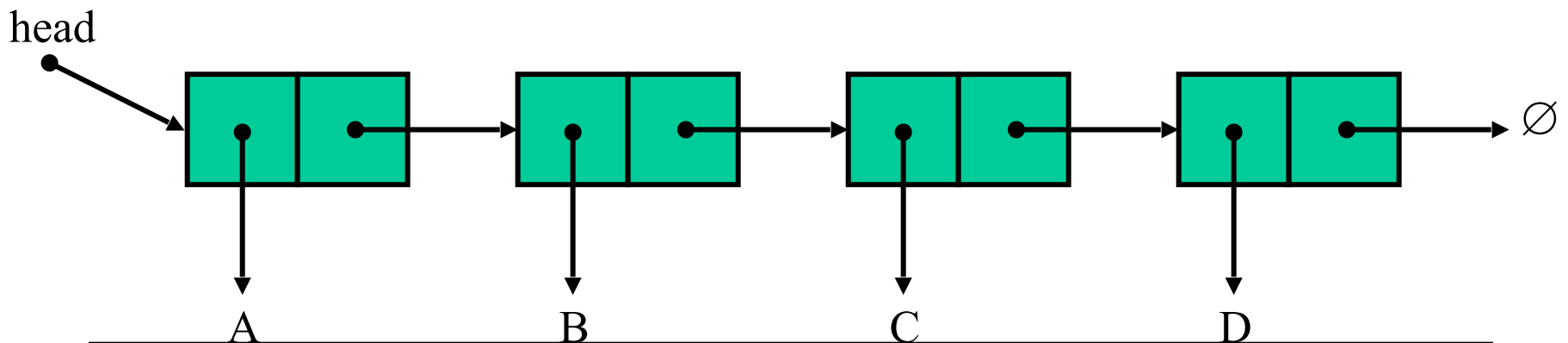
```
public J removeFirst() {  
    if (head == null)  
        return;  
    Node<J> tmp = head;  
    head = head.next;  
    return tmp.data;  
}
```

---

# removeLast()

---

- Problem
  - How do you remove the last
  - Can we use the lastNode utility function?
    - Not exactly, because to remove D we need to do things to C
    - Cannot go backwards!!
  - So, need to search forward in list to find the node before the last node



---

# Remove Last

---

- To find the node before last use two vars: prev and here
- each time in loop
  - prev=here
  - here=here.next

```
public J removeLast() {  
    if (head == null)  
        return null;  
    Node<J> prev = head;  
    Node<J> here = head.next;  
    if (here == null) {  
        // only one item in list  
        head = null;  
        return prev.data;  
    }  
    while (here.next != null) {  
        prev = here;  
        here = here.next;  
    }  
    prev.next = null;  
    return here.data;  
}
```

---

# Lab

---

- Hand write the method below
- This method should search through its linked list for a node containing the object `r` (use `==`).

```
/**  
 * Determine if the list contains the provided object.  
 * @param r the object to be looked for.  
 * @return true iff the object is in the linked list.  
 */  
boolean contains(J r);
```