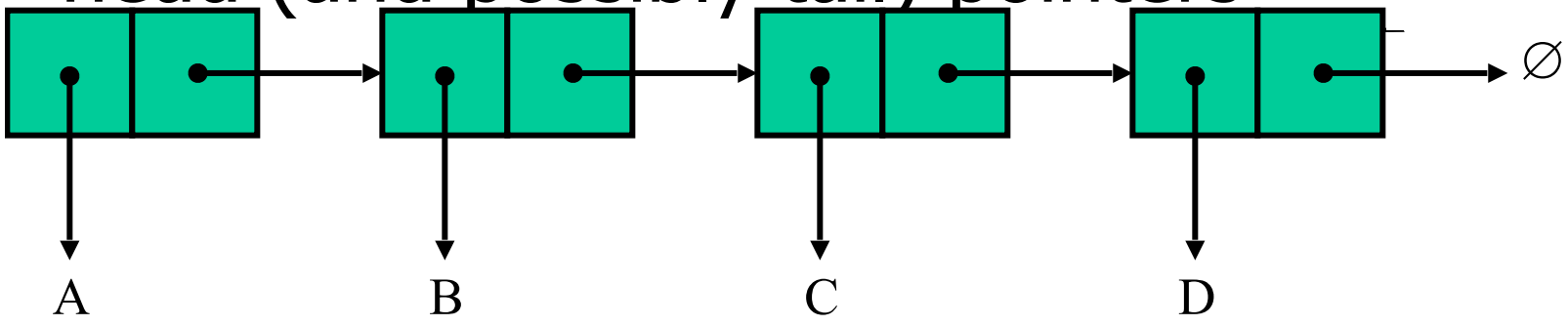
# CS206

## Linked Lists

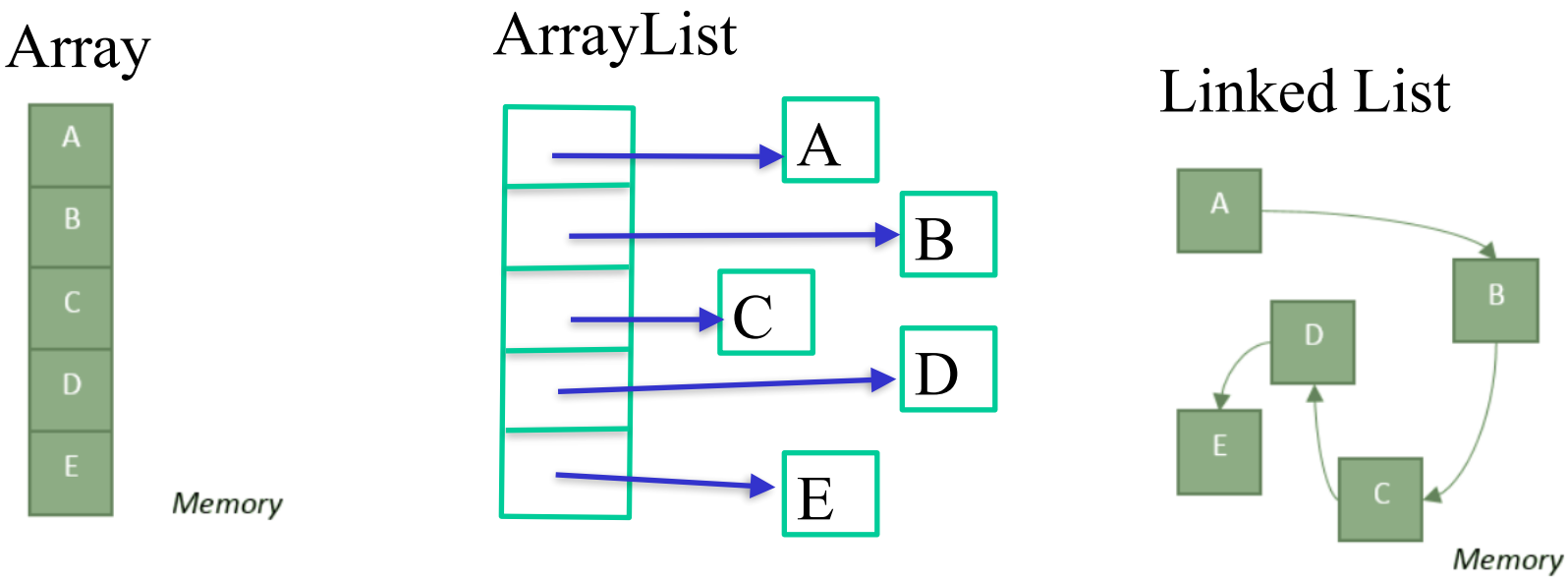
# Linked List

- A linked list is a lists of objects.
- The objects form a linear sequence.
- The sequence is unbounded in length.
- Need a way to get at elements
  - head (and possibly tail) pointers



# Linked List versus Array

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



# 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
  - Multi-dimensional arrays are similarly self-referential
- For linked lists, this structure is usually referred to as a Node

```
private class Node<J> {
  public J data;
  public Node next;
  public Node(J data, Node 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";

        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 a reference

Equals should compare content
compareTo should compare content

== compares memory location

# 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!!

# Starting Point

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

# Size

```
public int size() {
    int siz=0;
    for (Node n=head; n!=null; n=n.getNext())
    {
        siz++;
    }
    return siz;
}
```

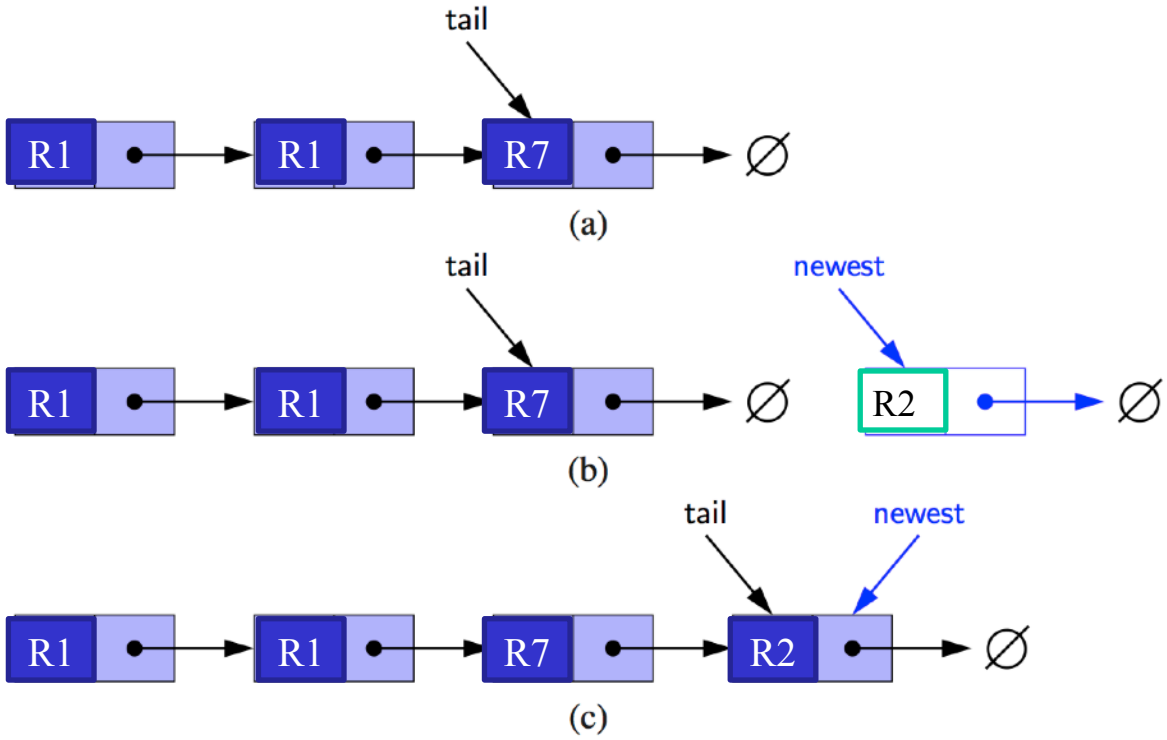
- Algorithmic Complexity (Big-O)?
- Can we improve?

# Print a Linked List

```
public String toString() {
    StringBuffer s = new StringBuffer();
    for (Node n=head; n!=null; n=n.getNext())
    {
        s.append( n.data.toString());
        if (n != tail)
        {
            s.append("\n");
        }
    }
    return s.toString();
}
```

# 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



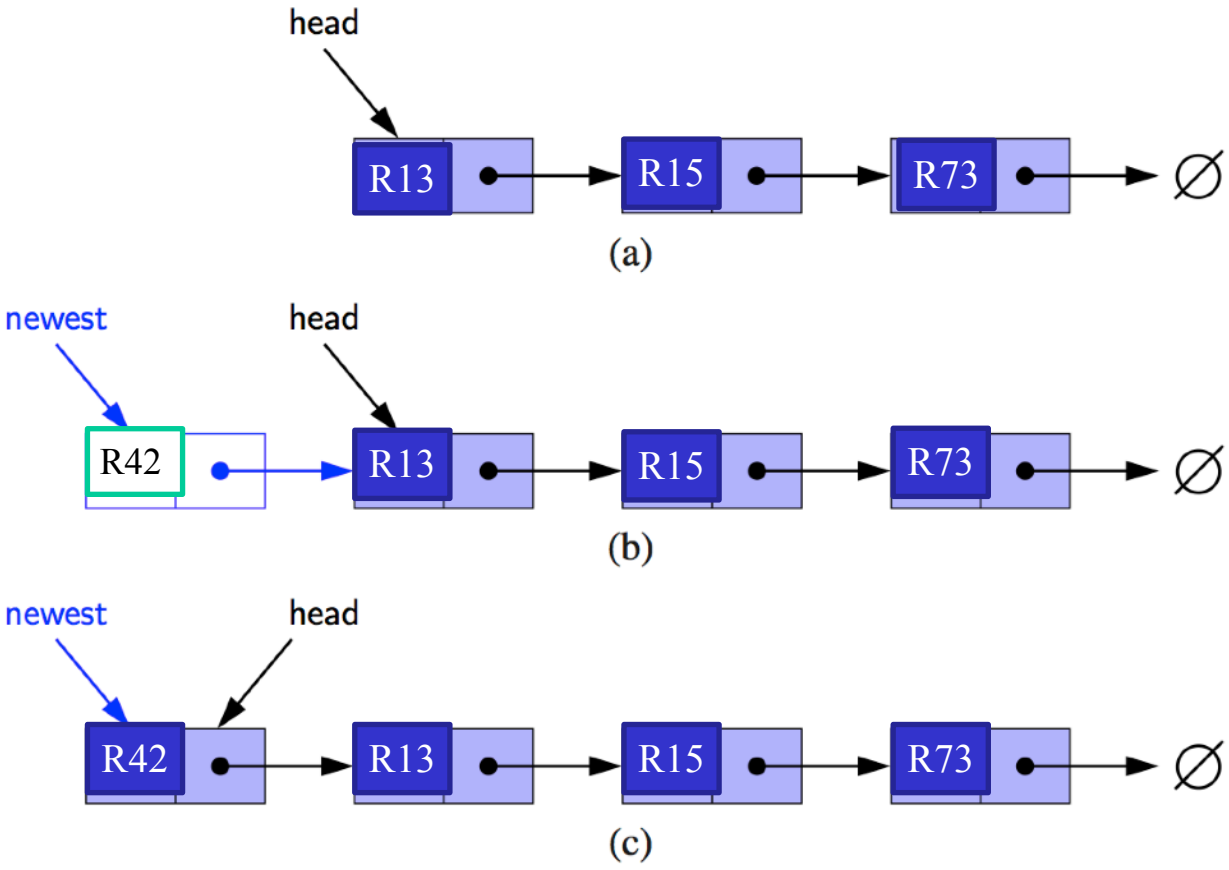
# Insertion

```
public void addLast(J c)
{
    Node newest = new Node(c, null);
    if (isEmpty())
   { head = newest;}
    else
    {
       tail.next=newest;
    }
    tail = newest;
    size++;
  }
```

Why not take a Node?

# Inserting at the Head

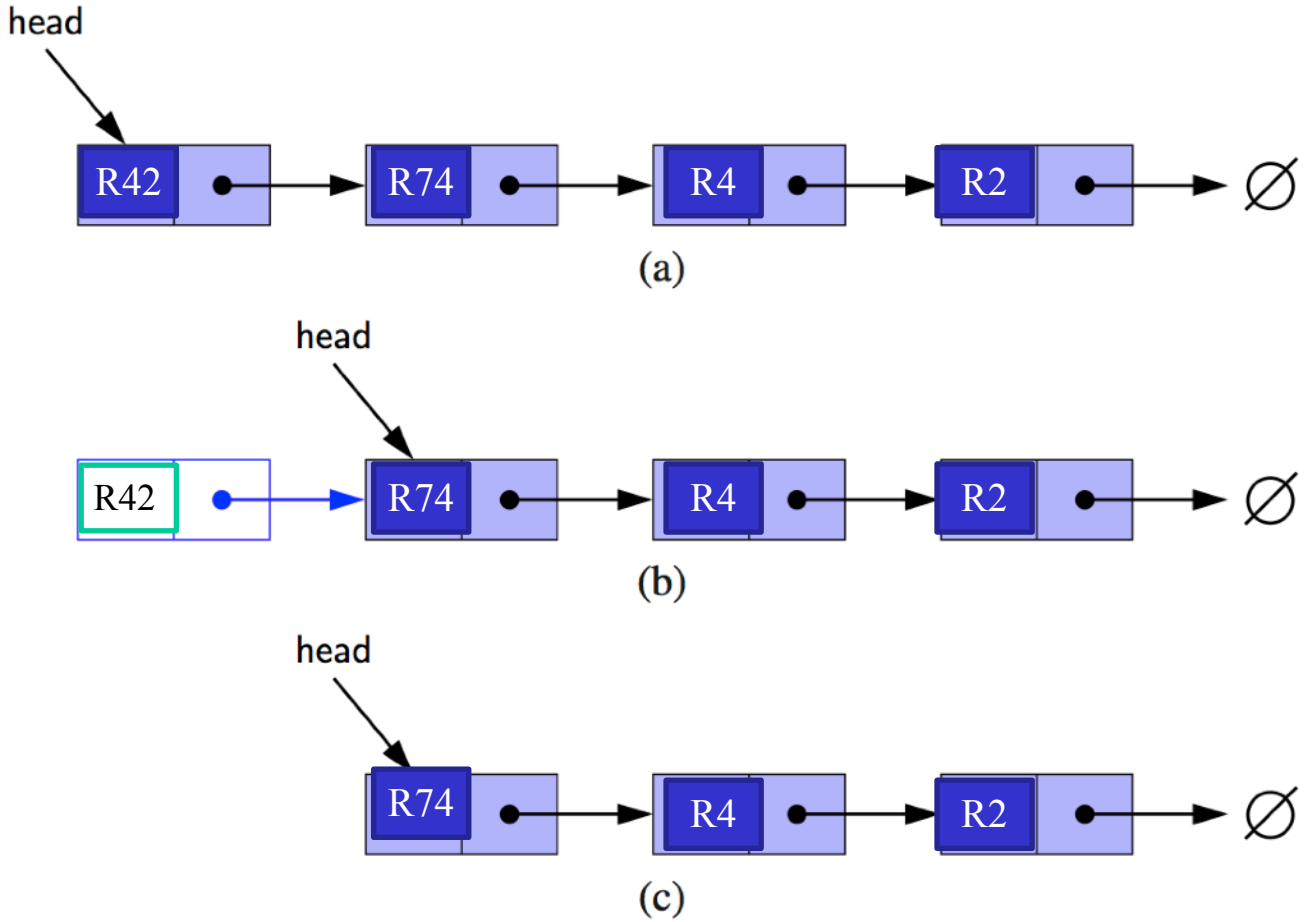
1. create a new node
2. have new node point to old head
3. update head to point to new node




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



# Deletion

```
public J removeFirst()
{
   if (isEmpty()) {return null;}
   J target = head.data;
   head = head.next;
   size--;
   if (isEmpty()) {tail = null;}
   return target;
}
```

# removeLast()

1. If you have a tail pointer
2. If you do not have a tail pointer

# Mini-Lab

- Hand write the method below

- This method should search through its linked list for a node containing the object j (use ==).

```
/**
     * Remove a node containing the provided object.
     * If not found, return false
     * If found, remove from the linked list the node containing r
     * and return true.
     * @param r the object to be removed.
     * @return true iff the object is in the linked list.
     */
    boolean remove(J r);
```