CS32 Discussion
Section 1B
Week 2

TA: Zhou Ren

Agenda

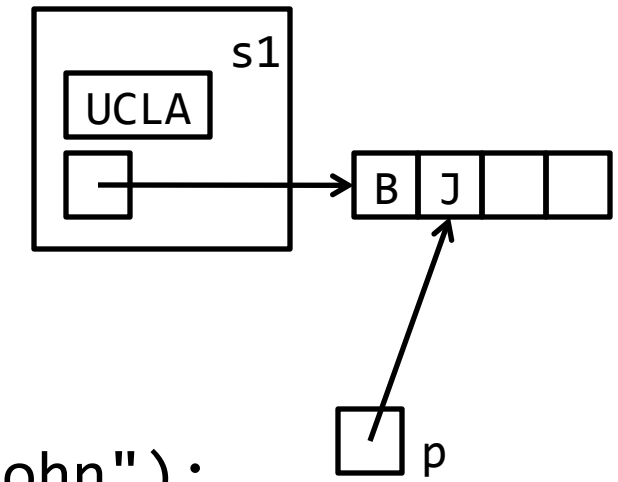
- Copy Constructor
- Assignment Operator Overloading
- Linked Lists

Copy Constructors - Motivation

```
class School
{
    public:
          School(const string &name);
          string getName() const;                                  // accessor
          void setName(const string &name);                        // modifier
          void addStudent(const Student &student);                 // modifier
          Student *getStudent(const string &name) const;           // accessor
          bool removeStudent(const string &name);                  // modifier
          int getNumStudents() const;                              // accessor
    private:
          string m_name;              // Name of the school.
          Student *m_students;        // Dynamic array of students.
          int m_numStudents;          // Number of students.
};
```

Copy Constructors - Motivation

```
Student st1("Brian");
Student st2("John");
School s1("UCLA");
s1.addStudent(st1);
s1.addStudent(st2);
Student *p = s1.getStudent("John");
```



We want to create a new `School` called `s2`, with exactly the same content as `s1`. In other words, we want to clone `s1`.

Copy Constructors - Motivation

- Candidate 1: Use an assignment.

```
School s2("");
s2 = s1;
```

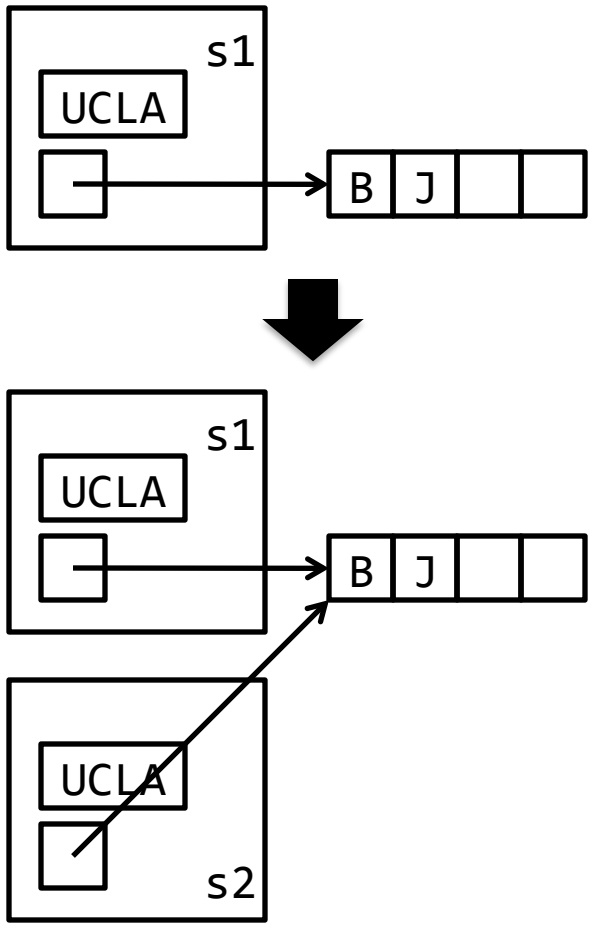
  - What are the issues with this method?

Copy Constructors - Motivation

- Candidate 1

```
School s2("");
s2 = s1;
```

  - Correctness: Every member variable gets copied – even the pointers (but not the pointees).
  - Efficiency: It will first call the default constructor of s2, initialize members with default values, and then copy the values.



Copy Constructors - Motivation

- Candidate 2: Just grab values out of `s1` and manually copy them into `s2`.

```
School s2("");
s2.setName(s1.getName());
…
```

  – What are the limits to this approach?

Copy Constructors - Motivation

- Candidate 2

```
School s2();
s2.setName(s1.getName());
// how do I get students out of s1?
```

  - We may not have accessors and modifiers to all member variables!
  - It is often not desirable to have the user (of a class) know all the internals.
  - Too long to write!

Copy Constructors

```
public:
    School(const School &aSchool);
```

- This is a constructor that is used to copy values from one instance to another.
- Why do you think the parameter is a constant reference?

School Copy Constructor

```
School::School(const School &aSchool)
{
```

School Copy Constructor

```
School::School(const School &aSchool)
: m_name(aSchool.m_name),
  m_numStudents(aSchool.m_numStudents),
  m_students(new Students[m_numStudents])
{
  for (int i = 0; i < m_numStudents; i++)
    m_students[i] = aSchool.m_students[i];
}
```

- Why is it that you don't have to use the accessors?
- If there are dynamically allocated objects, you allocate new memory and manually copy them over.

School Copy Constructor

- With the copy constructor defined, you can now use:

```
School s2(s1);
```

or equivalently,

```
School s2 = s1;
```

Pass-by-Value & Copy Constructor

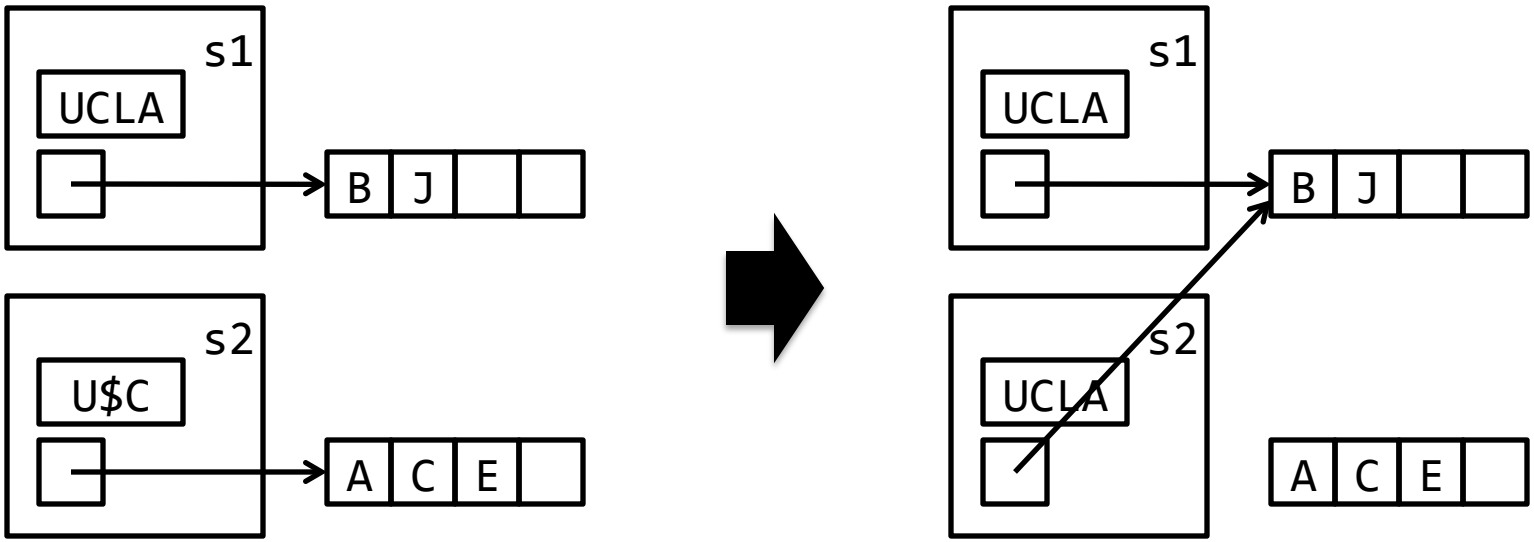
- Copy Constructor is sometimes called for you!

```
void foo(School aSchool)
{
   ...
}
```

- Here, a School instance is passed by value. `foo()` will work with a copy of `aSchool`, and the copy constructor will be used to create that copy.

But how about assignments?

```
s2 = s1;
```



Assignment Operator Overloading

```
s2 = s1;
```

- Overload the operator (in this case, we overload the assignment operator).

```
public:
    School& operator=(const School &aSchool)
```

Assignment Operator Overloading

```
School& School::operator=(const School &aSchool)
{
```

I assume we have = operator properly defined in Student class.

Assignment Operator Overloading

```
School& School::operator=(const School &aSchool)
{
  m_name = aSchool.m_name;
  m_numStudents = aSchool.m_numStudents;
  m_students = new Students[m_numStudents];
  for (int i = 0; i < m_numStudents; i++)
    m_students[i] = aSchool.m_students[i];

  return *this;          // don't forget this!
}
```

I assume we have = operator properly defined in Student class.

Assignment Operator Overloading

```
School& School::operator=(const School &aSchool)
{
  m_name = aSchool.m_name;
  m_numStudents = aSchool.m_numStudents;
  delete[] m_students;
  m_students = new Students[m_numStudents];
  for (int i = 0; i < m_numStudents; i++)
    m_students[i] = aSchool.m_students[i];

  return *this;          // don't forget this!
}
```

I assume we have = operator properly defined in Student class.

Assignment Operator Overloading

```
School& School::operator=(const School &aSchool)
{
  if (this != &aSchool)
  {
    m_name = aSchool.m_name;
    m_numStudents = aSchool.m_numStudents;
    delete[] m_students;
    m_students = new Students[m_numStudents];
    for (int i = 0; i < m_numStudents; i++)
      m_students[i] = aSchool.m_students[i];
  }
  return *this;          // don't forget this!
}
```

I assume we have = operator properly defined in Student class.

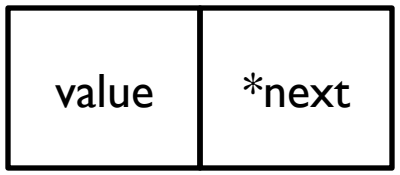
Before we talk about linked lists…

- CS32 is all about *organizing* data. We call an organization scheme a **data structure**. For every data structure, we must define:
  - rules for organizing data items (e.g., array with integers stored in a nondecreasing order),
  - a method to add a new data item without breaking any of the rules,
  - a method to remove a data item without breaking any of the rules, and
  - most importantly, how to search for an item

- We will examine various data structures and algorithms, pros and cons of each, as well as their efficiency.

Linked Lists

- A key component of a linked list is a **node**, which is a single unit of data.

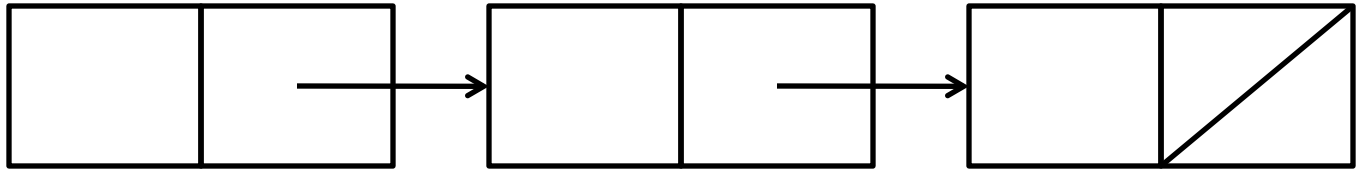


- The first box carries a *value*, and the second is a *pointer* to another node.
- Here is an example node definition in the form of a C++ struct:

```
typedef int ItemType;
struct Node
{
      ItemType value;
      Node *next;
};
```

Linked Lists

- Linked list is a series of nodes, each pointing to the next one.



- The last node's next pointer is NULL.
- What is the information you need to complete the picture?

Head Pointer!

- Obviously, you need to know where it begins.

- We keep a pointer that points to the first item and call it the **head pointer**.
- e.g.

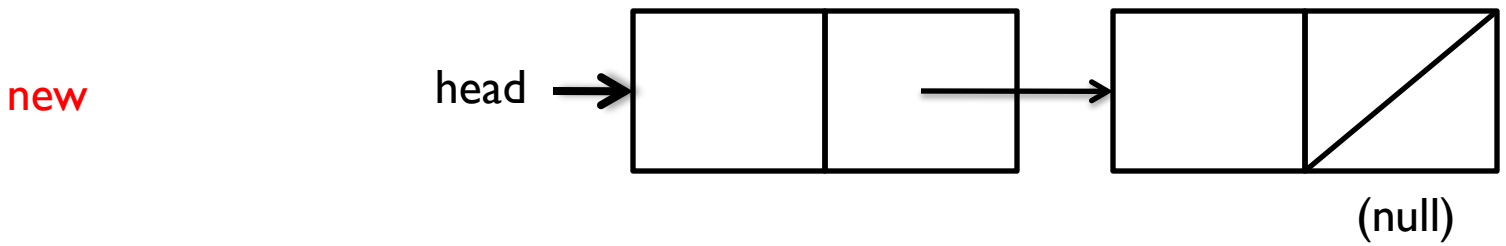
```
Node *head;
```

Linked Lists (Min. Requirements)

- You need a description of a node, which must contain a next pointer.
- You need a head pointer that points to the first node.
- The list must be loop-free (unless it is a *circularly linked list*, in which case one (and only one) loop must exist).

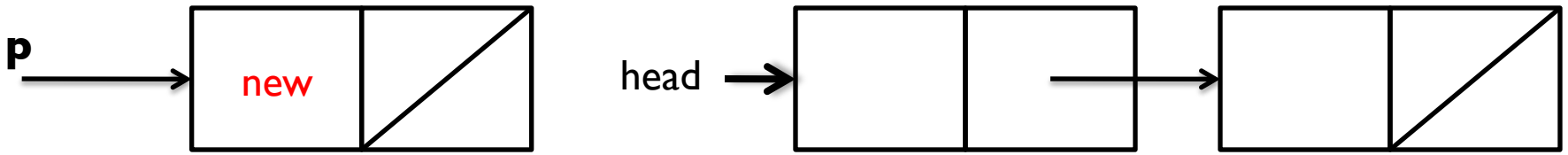
Linked Lists (Insertion)

- Adding a new value to the list.



Linked Lists (Insertion)

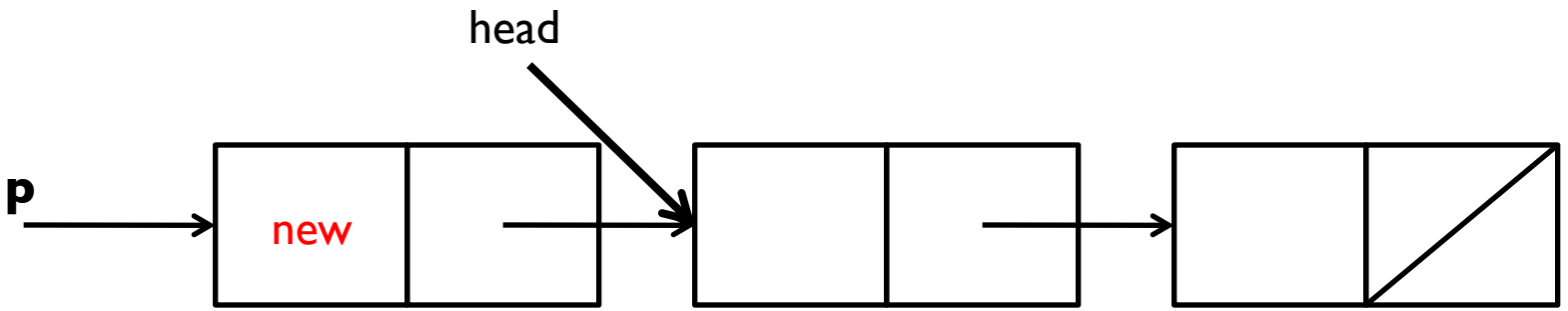
1. Create a new node. Call the pointer to it **p**.



Linked Lists (Insertion)

2. Make its *next* pointer point to the first item.

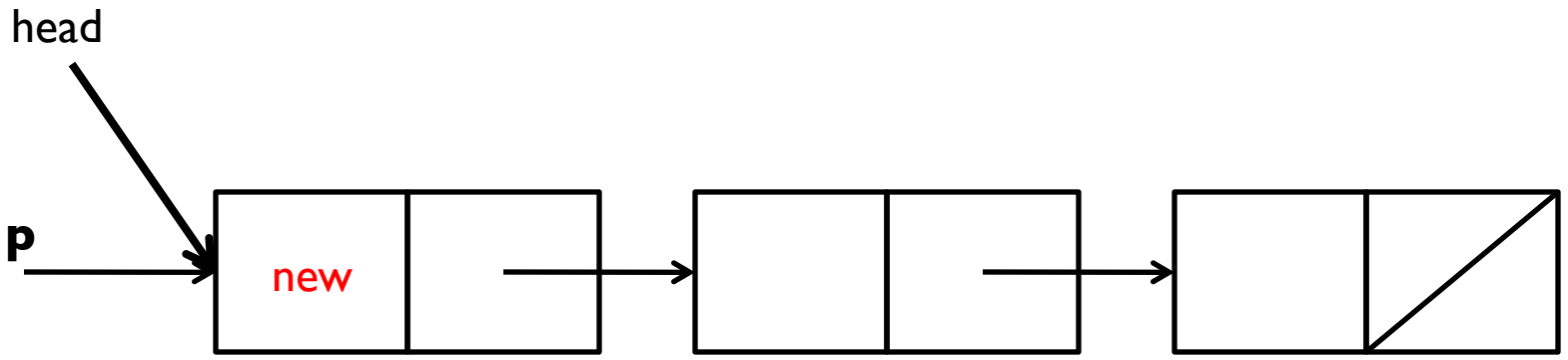
```
p->next = head;
```



Linked Lists (Insertion)

3. Make the head pointer point to the new node.

```
head = p;
```



Linked Lists (Insertion)

- Sanity Check
  - Does it work with an empty list?

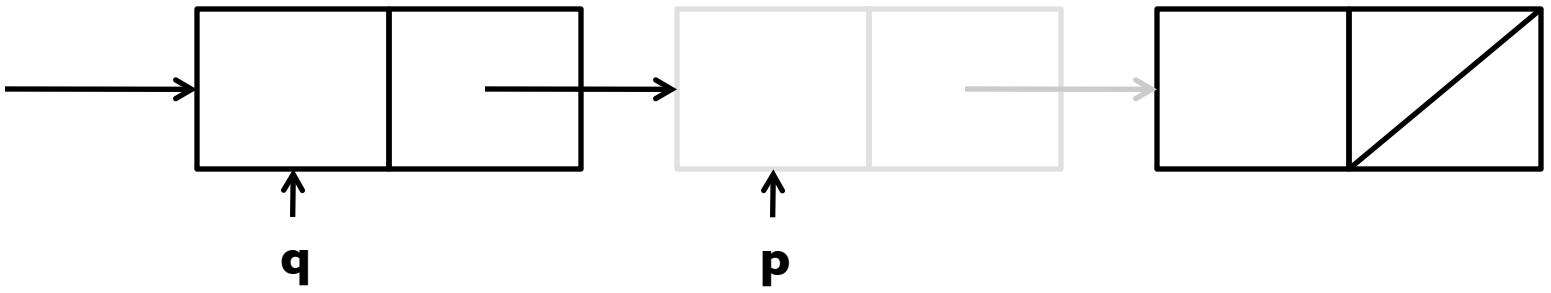
```
p->next = head;
head = p;
```

Linked Lists (Search)

- Isn't it too obvious?

Linked Lists (Removal)

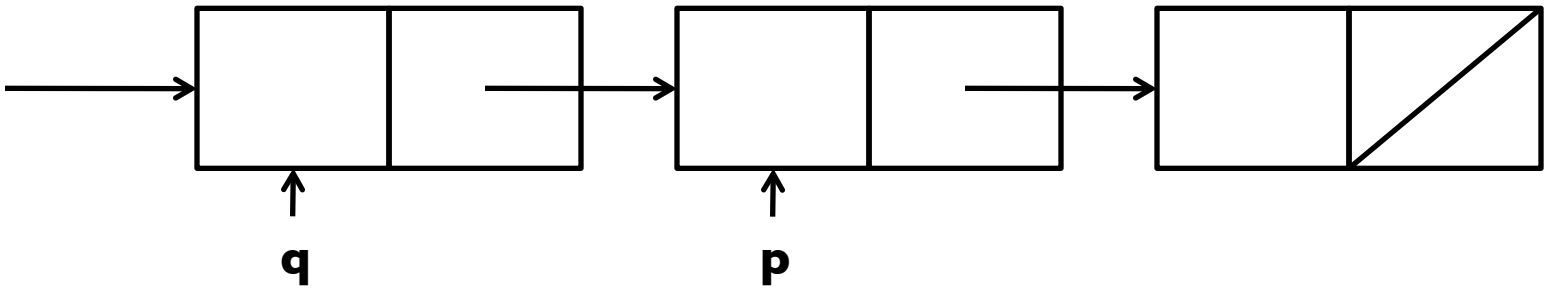
- Suppose there is an item that you want to remove, and it is pointed by a pointer, say **p**.
- Can I just do "delete p;"?



- We need to set the previous node's (**q**) *next* pointer to point to the next node of **p**!

Linked Lists (Removal)

- When looking up **p**, keep the pointer to the previous node (**q**).

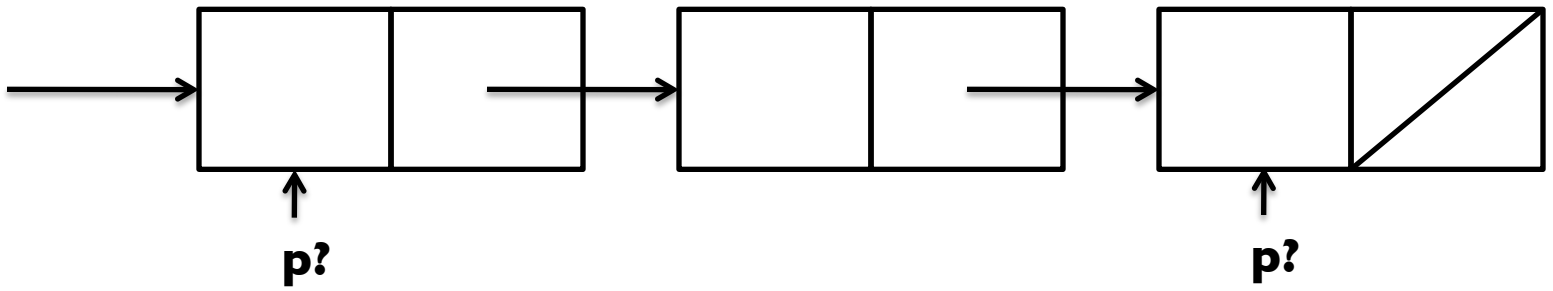


- Then …

```
q->next = p->next;
delete p;
```

Linked Lists (Removal)

- Sanity Checks
  - Does it work if p == head?
  - Does it work if p points to the last one?




```
q->next = p->next;
delete p;
```

Linked Lists (Removal)

- If `p == head`, there is no "previous" node to p.
- Make an exception for this.
  - We need to reset the head pointer.

```
head = p->next;
delete p;
```

Linked Lists (Removal -- Summary)

```
remove(valToRemove)
    p = head, q = NULL
    while p != NULL:
        if p->value == valToRemove:
            break
        q = p
        p = p->next

    if p == NULL:  // no valToRemove in the list
        return
    if p == head (or equivalently, q == NULL):
        head = p->next
    else:
        q->next = p->next

    delete p
```

Rule: Data Removal

- When removing something from a structure with pointers…
  - **Fix the pointers first!**
  - Then delete the data from memory.

What's nice about linked lists

- Very efficient insertion
- Flexible memory allocation
  - Think about what you should do if you have to grow/shrink a dynamically allocated array.
  - And yes, there is a little overhead, but that's the price we pay.
- Simple to implement

What's not so nice about linked lists

- Slow search (i.e. accessing a certain element, e.g. "get the 4237th item")
  - Usually, search is the operation that matters more than insertion or removal.

Variations

- Sorted Linked Lists
  - Make changes to the insertion method.
- Doubly Linked Lists
  - Each node has *prev* and *next* pointers.
  - A *tail* pointer is kept to point to the last node.
  - Why do you think this is useful?
- Circularly Linked Lists
  - The last node's next pointer points to the first one.
  - Essentially, there is no "first" node.