Ve 280

Programming and Introductory Data Structures

Linked Lists

Course Evaluation

- For instructor to improve the teaching, students' feedbacks are very important.

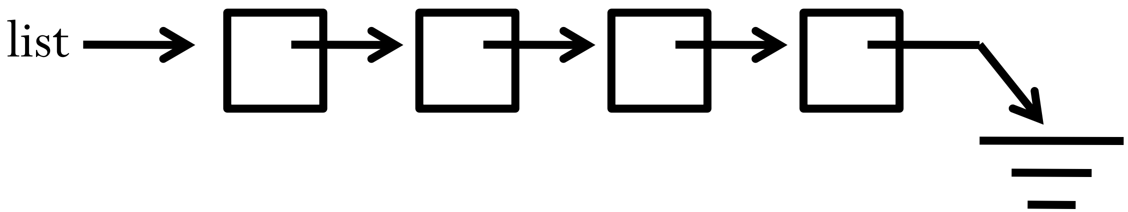
- JI uses an online evaluation system called "IDEA".
  - It sends an email to your **university email account**, which gives you instructions. Please check your university email.
  - All responses are **anonymous**.
  - I encourage you to let me know your feedback.

Outline

- Implementation of Linked List
- Double-Ended Linked Lists

Review

- Introduction to Linked List

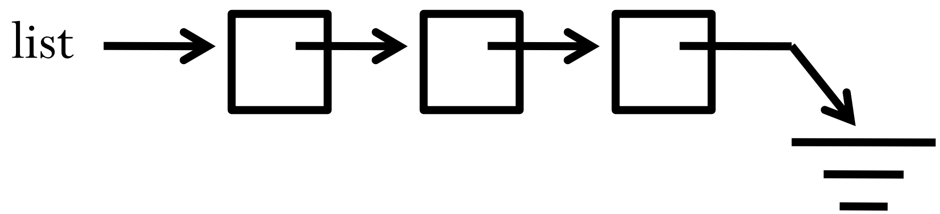


- Method
  - `bool isEmpty();`
  - `void insert(int v); // insert at the front`
  - `int remove(); // remove and return the first`
    `// element of the list;`
    `// if empty, throw listIsEmpty`

Linked Lists

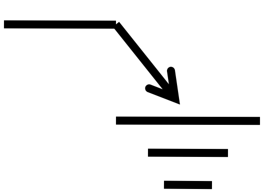
Implementation

- To implement linked list, we need to pick a concrete representation for the node in the list.




```
struct node {
  node *next;
  int   value;
};
```

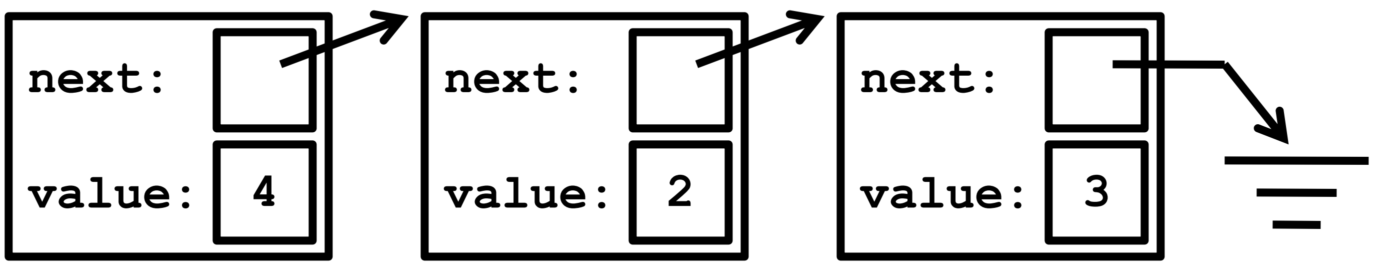
- The invariants on these fields are:
  - The **value** field holds the integer value of this element of the list.
  - The “**next**” field points to the next node in the list, or NULL if the node is the last one in the list.
- NULL means "pointing at nothing". Its value is "0", written as:



Linked Lists

Implementation

- The concrete representation of the list (4 2 3) is:



- The basic idea of implementation is that each time an `int` is inserted into the list, we'll create a new node to hold it.
- Each time an `int` is removed from the (non-empty) list, we'll save the value of the first node, **destroy** the first node, and return the value.

Linked Lists

Implementation

- We'll use the following (private) data members:

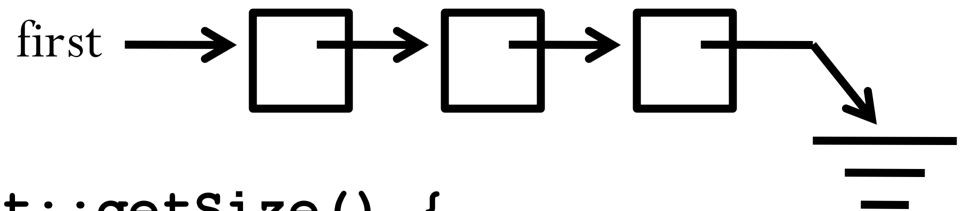
```
class IntList {
  node *first;
 public:
   ...
};
```

```
struct node {
  node *next;
  int   value;
};
```

- The rep invariant is that "first" points to first node of the sequence of nodes representing this `IntList`, or NULL if the list is empty.

Linked List Traversal

- With the “first” pointer, we can traverse the linked list.




```
int IntList::getSize() {
// Effect: return # of items in this list
  int count = 0;
  node *current = first;
  while(current){
    count++;
    current = current->next;
  }
  return count;
}
```

Traverse through the list.

Linked Lists

Implementation

- Here are the public methods we have to implement:

```
class IntList {
    node *first;
  public:
    bool isEmpty();
    void insert(int v);
    int remove();
    IntList();                          // default ctor
    IntList(const IntList& l);  // copy ctor
    ~IntList();                         // dtor
    // assignment
    IntList &operator=(const IntList &l);
};
```

Linked Lists
Implementation

- We will implement the "operational" methods first, assuming that the representation invariants hold.
- After that, we'll go back and implement the default constructor and the **Big Three** to make sure that:
  - The invariants hold during object creation.
  - All dynamic resources are accounted for.

- A list is empty if there is no node in the list, or `first` is NULL:

```
bool IntList::isEmpty() {
   return !first;
}
```

Linked Lists

Implementation

- When we insert an integer, we start out with the "first" field pointing to the current list:
  - That list might be empty, or it might not, but in any event “first” **must** point to a valid list thanks to the rep invariant.

- The first thing we need to do is to create a new node to hold the new "first" element:

```
void IntList::insert(int v) {
  node *np = new node;
  ...
}
```

<u>Question</u>: Can we declare a **<u>local</u>** object instead of a **<u>dynamic</u>** one? I.e., declare:

```
node n;
```

Linked Lists

Implementation

- Next, we need to establish the invariants on the new node.
- This means setting the value field to `v`, and the next field to the "rest of the list" – this is precisely the start of the current list:

```
void IntList::insert(int v) {
    node *np = new node;
    np->value = v;
    np->next = first;
    ...
}
```

Linked Lists

Implementation

- Finally, we need to reestablish the representation invariant: `first` currently points to the **second** node in the list, and must point to the first node of the new list instead:

```
void IntList::insert(int v) {
    node *np = new node;
    np->value = v;
    np->next = first;
    first = np;
}
```

We have accomplished the work of the method, and all invariants are now true, so we are done.

Linked Lists

Implementation

- Finally, we need to reestablish the representation invariant: `first` currently points to the **second** node in the list, and must point to the first node of the new list instead:

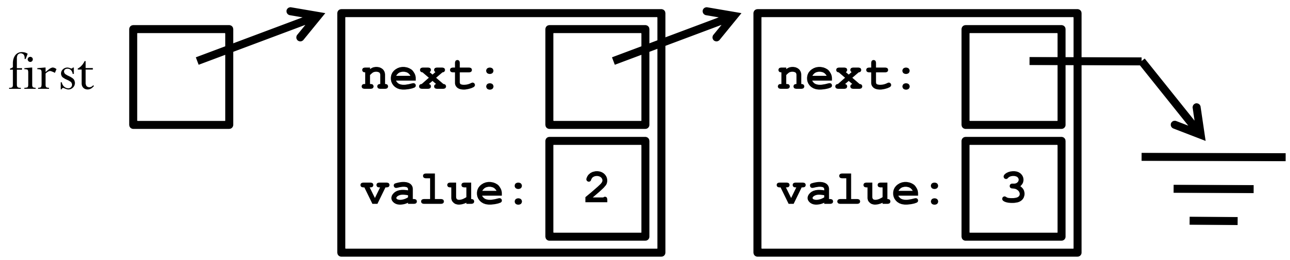
```
void IntList::insert(int v) {
  node *np = new node;
  np->value = v;
  np->next = first;
  first = np;
}
```

Notice that this works no matter what the current list is, as long as the invariant holds.

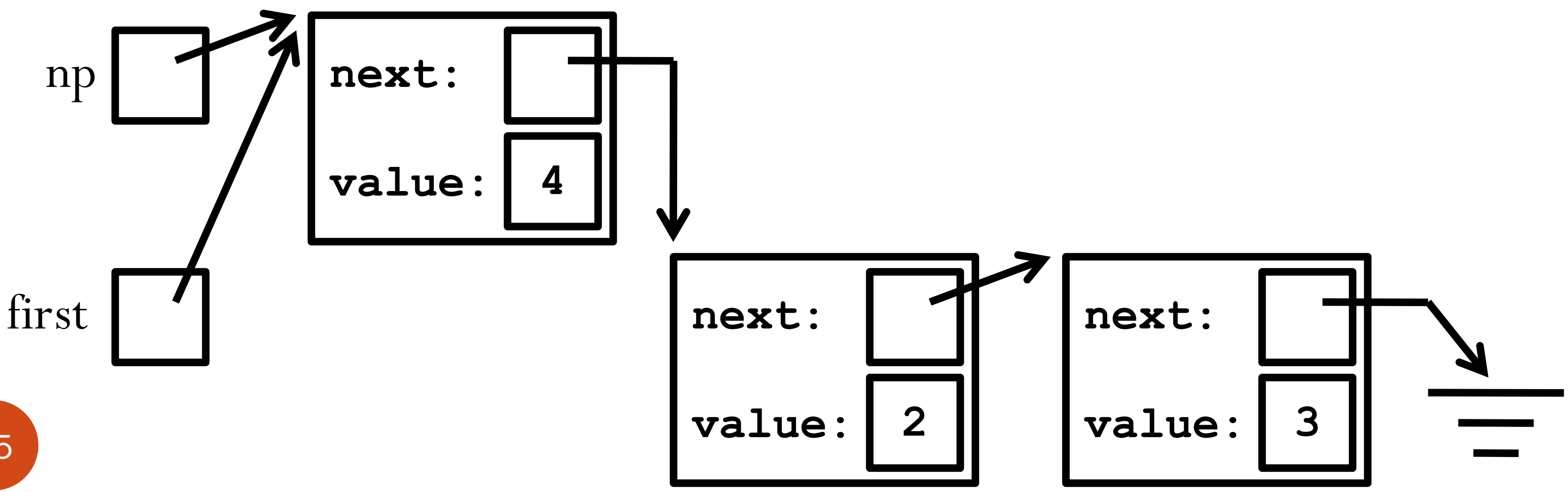
Linked Lists

Example

- Suppose we are inserting a 4.
- The list might already have elements:



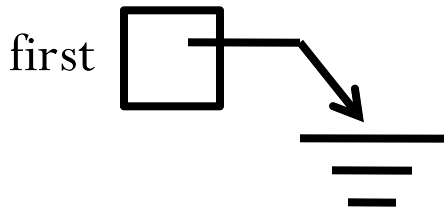
- And then the new list is



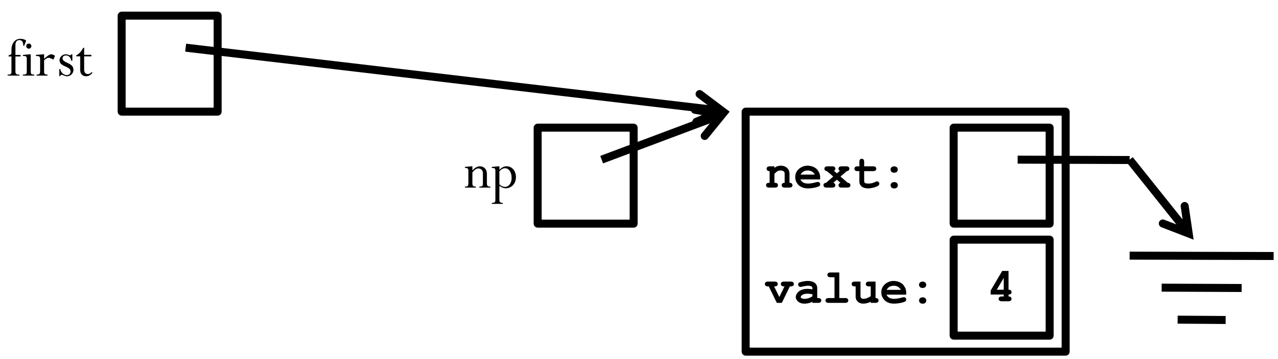
Linked Lists

Example

- Suppose we are inserting a 4.
- The list might be empty:



- And the new list is



Linked Lists

Implementation

- Removal is a bit trickier since there are lots of things we need to accomplish, and they have to happen in precisely **the right order**.
- If the first item is removed, this violates the invariant on "first", which we have to fix:

```
int IntList::remove() {
    ...
    first = first->next;
    ...
}
```

Linked Lists

Implementation

```
first = first->next;
```

- If we are removing the first node, we must delete it to avoid a memory leak.
- Unfortunately, we **can't** delete it before advancing the "first" pointer (since first->next would then be undefined).
- But, **after** we advance the "first" pointer, the node to be removed is an orphan, and can't be deleted.

- We solve this by introducing a local variable to remember the "old" first node, which we will call the `victim`.

Linked Lists

Implementation

- After creating the `victim`, we can then delete the node **after** it is skipped by first.

```
int IntList::remove() {
   node *victim = first;
   ...
   first = victim->next;
   ...
   delete victim;
   ...
}
```

Note: equivalent to
`first = first->next;`

Linked Lists

Implementation

- However, removing the first node is only half of the work.
- We must also return the value that was stored in the node.
- This is also tricky:
  - We can't return the value first and then delete the node, since then the delete wouldn't happen.
  - Likewise, if we delete the node first, the contained value is lost.

- So, we use **another** local variable, `result`, to remember the result that we will eventually return.

Linked Lists

Implementation

- Now that we have the `result` variable, the method becomes:

```
int IntList::remove() {
  node *victim = first;
  int result;
  ...
  first = victim->next;
  result = victim->value;
  delete victim;
  return result;
}
```

Linked Lists
Implementation

- Finally, we need to cope with an empty list, and throw an exception if we have one:

```
int IntList::remove() {
  node *victim = first;
  int result;
  if (isEmpty()) {
    listIsEmpty e;
    throw e;
  }
  first = victim->next;
  result = victim->value;
  delete victim;
  return result;
}
```

Linked Lists

Exercise

- Note that for victim, we initialize it when it is declared, but we don't for `result`.
- **<u>Question</u>**:

  Why didn't we initialize `result` to `victim->value`?

```
int IntList::remove() {
  node *victim = first;
  int result;
  if (isEmpty()) {
    listIsEmpty e;
    throw e;
  }
  first = victim->next;
  result = victim->value;
  delete victim;
  return result;
}
```

Linked Lists

Implementation

- Now let's work on the maintenance methods:
  - Constructors
  - Assignment operator
  - Destructor
- The default constructor is easy:
  - We just have to establish the representation invariant for an empty list:

```
IntList::IntList()
: first(0)
{}
```

Linked Lists

Implementation

- Likewise, the destructor is easy.
- We have to destroy each node in the list before the list itself is destroyed.
- Actually, we already have a mechanism to destroy a single node – it's a side effect of `remove()`.

- So, we call `remove()` until the list is empty, ignoring `remove()`'s result.
- We put this functionality into another private method, called `removeAll()`.

Linked Lists

Implementation

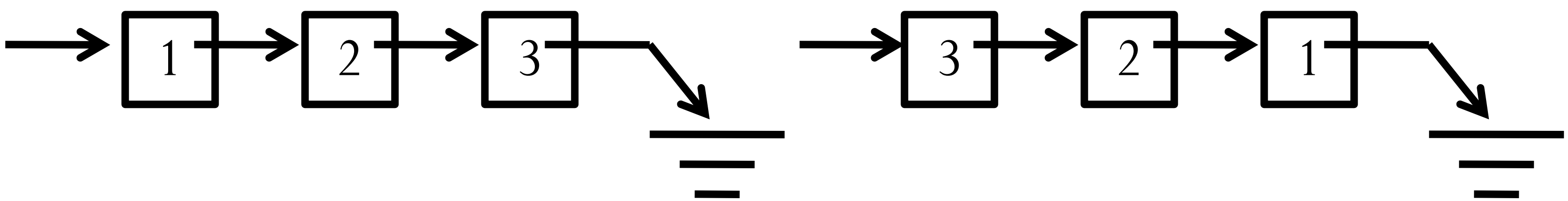
- Here is the destructor and its helper:

```
void IntList::removeAll() {
  while (!isEmpty()) {
    remove();
  }
}

IntList::~IntList() {
  removeAll();
}
```

Linked Lists

Implementation

- The copy constructor is tricky.
- The naive approach would be to walk the list from front to back, and insert each element that we find into the list.
- However, this gives us a list **in reverse order**, because we always insert a new element at the beginning of the list.



- What we would prefer is to be able to walk the list **backward**.

Linked Lists

Implementation

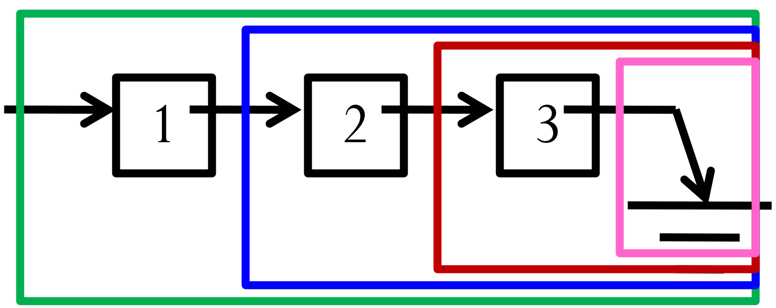
- Since there's no convenient way to walk the list backwards, we'll instead write a helper function that will **recursively** walk the list till the end.
- When we unwind the recursion, we can insert the elements from "back" to "front", which gives us the right answer:

```
void IntList::copyList(node *list) {
  if (!list) return; // Base case

  copyList(list->next);
  insert(list->value);
}
```

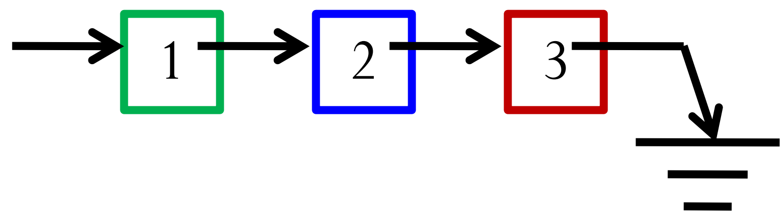
Linked Lists

Implementation

```
void IntList::copyList(node *list) {
  if (!list) return; // Base case

  copyList(list->next);
  insert(list->value);
}
```



Assuming the current list is empty



- `copyList()` must be a private method, since it deals with the concrete representation, not the abstraction.

Linked Lists

Implementation

- With `copyList()`, the copy constructor and assignment operator are pretty easy.
- For the copy constructor, make sure we start with an empty list, and then call `copyList()`:

```
IntList::IntList(const IntList &l)
: first (0)
{
   copyList(l.first);
}
```

Linked Lists

Implementation

- The assignment operator ensures that there is no self-assignment, destroys the current list, then copies the new one:

```
IntList &IntList::operator=
        (const IntList &l) {
  if (this != &l) {
    removeAll();
    copyList(l.first);
  }
  return *this;
}
```

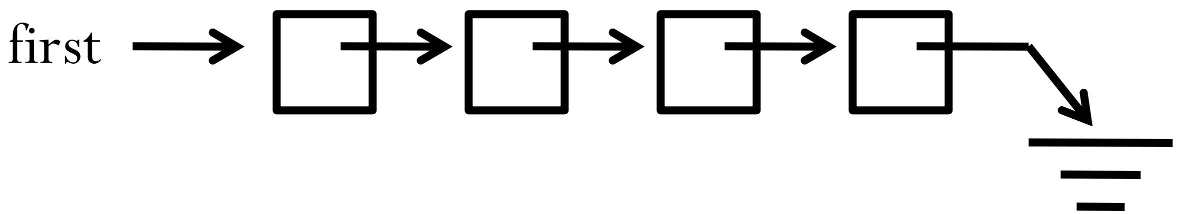
Outline

- Implementation of Linked List
- Double-Ended Linked Lists

Linked Lists

Double-ended list

- What if we wanted to insert something at the end of the list?
- Intuitively, with the current representation, we need to walk down the list until we found "the last element", and then insert it there.



- That's not very efficient, because we have to go through every element to insert something at the tail.
- Instead, we'll change our concrete representation to track both the front and the back of our list.

Linked Lists

Double-ended list

- The new representational invariant has **two** node pointers:

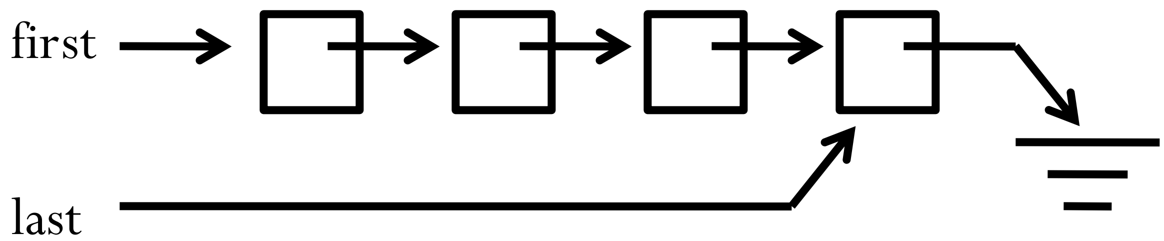
```
class IntList {
    node *first;
    node *last;
  public:
    ...
};
```

- The invariant on `first` is unchanged.
- The invariant on `last` is:
  - `last` points to the last node of the list if it is not empty, and is NULL otherwise.

Linked Lists

Double-ended list

- So, in an empty list, both `first` and `last` point to NULL.
- However, if the list is non-empty, they look like this:



- Question: Adding this new data member, what methods should be changed?
    - Answer: remove, insert, and constructors should be re-written
- In lecture, we'll only write a new method, `insertLast`, which inserts a node at the tail of the linked list

Linked Lists

Double-ended list

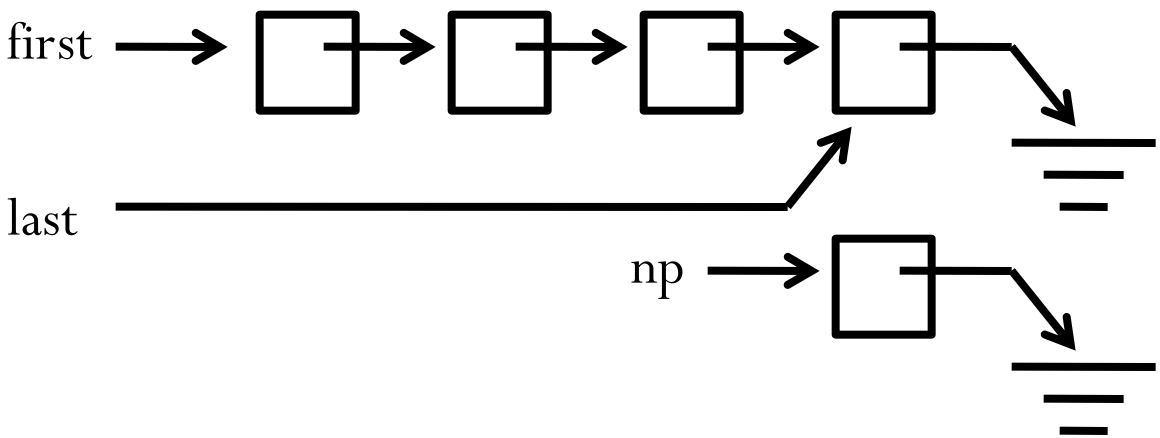
- First, we create the new node, and establish its invariants:

```
void IntList::insertLast(int v) {
   node *np = new node;
   np->next = NULL;
   np->value = v;
   ...
}
```

Linked Lists

Double-ended list

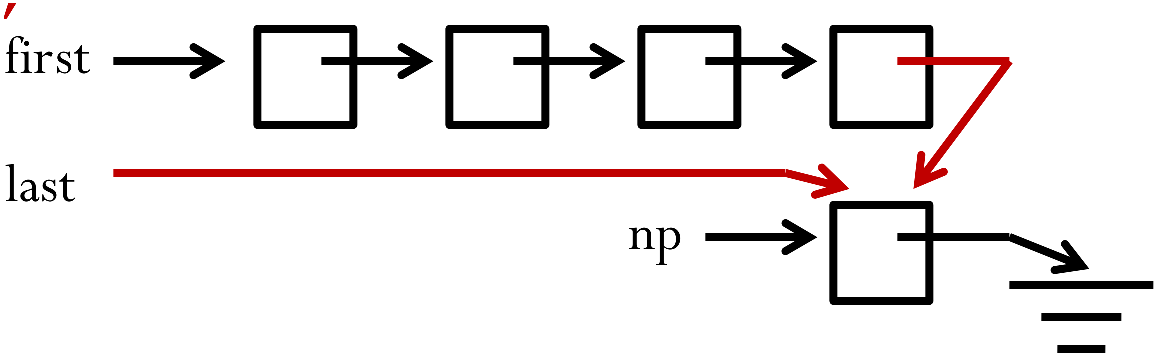
- To actually insert, there are two cases:
  - If the list is empty, we need to reestablish the invariants on `first` **and** `last` (the new node is both the first and last node of the list)
  - If the list is **not** empty, there are two broken invariants. The "old" `last->next` element (incorrectly) points to NULL, and the `last` field no longer points to the last element.



Linked Lists

Double-ended list

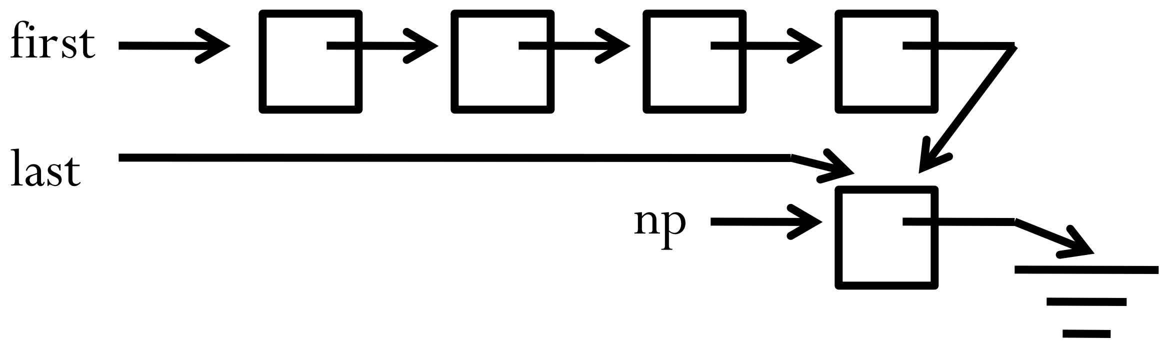
```
void IntList::insertLast(int v) {
   node *np = new node;
   np->next = NULL;
   np->value = v;
   if (isEmpty()) {
      first = last = np;
   }
   else {
      last->next = np;
      last = np;
   }
}
```



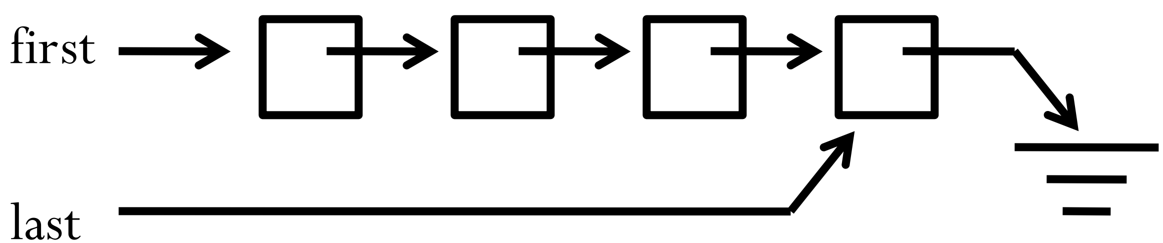
Linked Lists

Double-ended list

- This is efficient, but only for insertion.



- **<u>Question</u>**: Is removal **from the end** efficient or not? Why?



Linked Lists

Double-ended list

- To make removal from the end efficient, as well, we have to have a **doubly-linked** list, so we can go forward **and** backward.
- To do this, we're going to change the representation again.
- In our new representation, a node is:

```
struct node {
   node *next;
   node *prev;
   int   value;
};
```

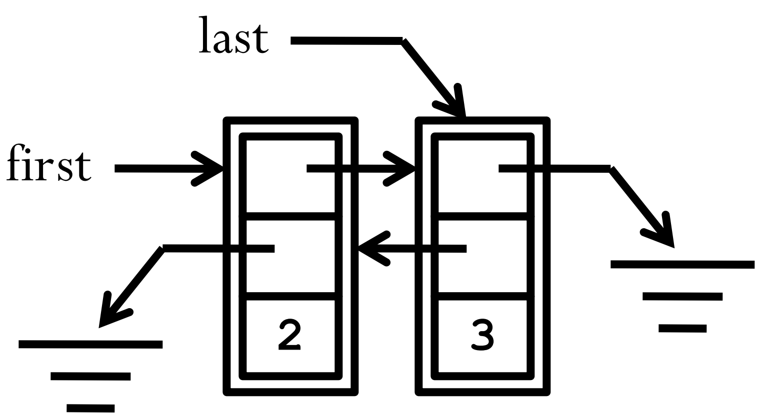
- The `next` and `value` fields are the same as before.
- The `prev` field's invariant is:
  - The `prev` field points to the previous node in the list, or NULL if no such node exists (e.g., the current node is the first node).

Linked Lists

Double-ended list

- With this representation, an empty list is unchanged: both "first" and "last" are NULL.

- While the list (2, 3) would look like this:



- We will implement each method in project five.

Reference

- **Problem Solving with C++ (8th Edition)**, by *Walter Savitch*, Addison Wesley Publishing (2011)
  - Chapter 13.1 Nodes and Linked Lists