Slides by Andrew DeOrio

EECS 280

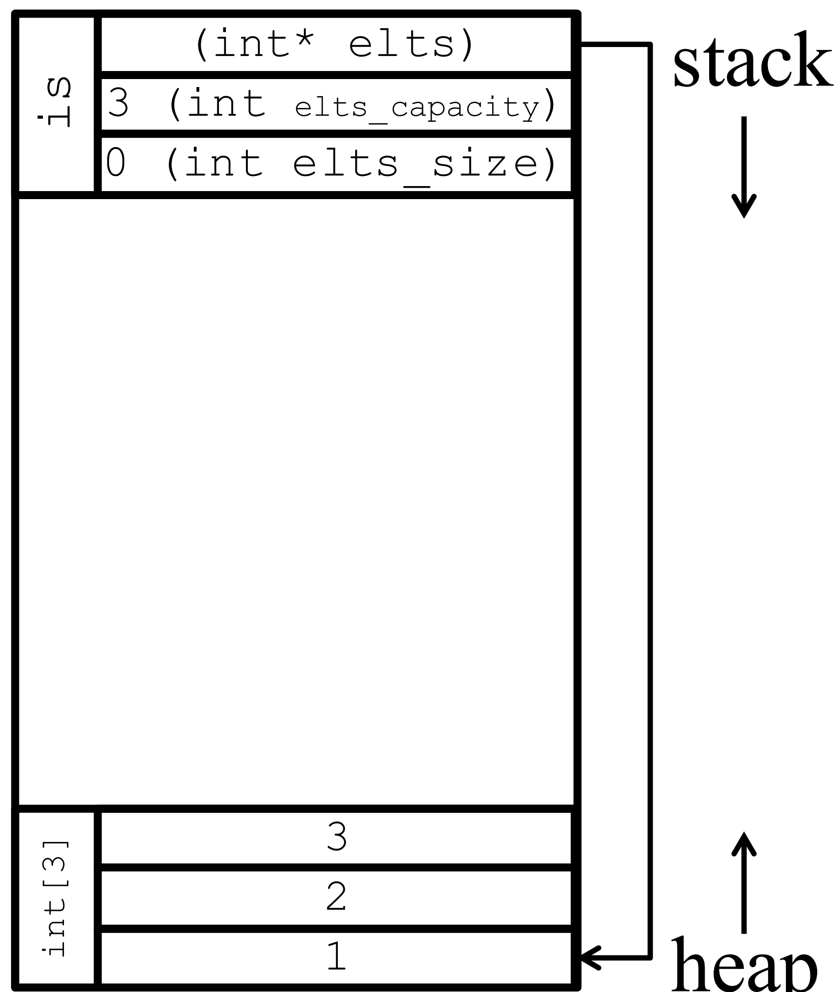
Programming and Introductory Data Structures

Linked Lists

Array-based structures

- In recent lectures, we implemented an array-based container ADT
- Values stored in the container were located "next door" to each other in an array on the heap

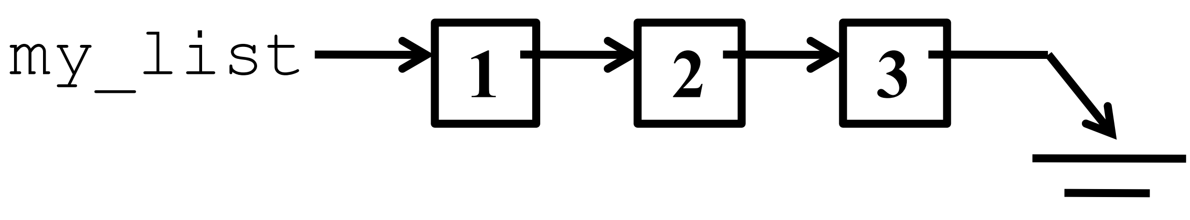
```
class IntSet {
  int *elts; //ptr to dynamic array
  int elts_size;
  int elts_capacity;
  //...
};

int main() {
  IntSet is(3);
  is.insert(1);
  is.insert(2);
  is.insert(3);
}
```



Linked structures

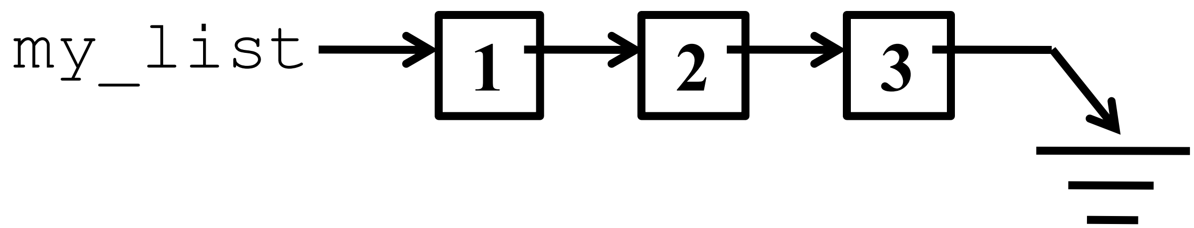
- Another way to implement a container uses pointers to connect one piece of data to the next
- Example: `list_t` from project 2

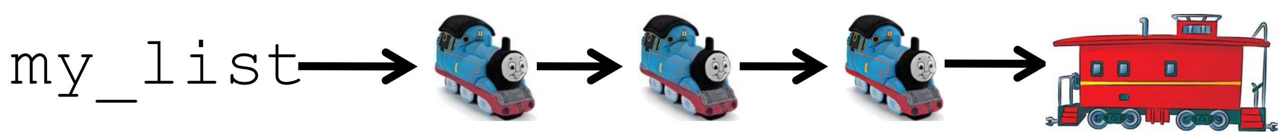
```
int main() {
  list_t my_list;                 // ( )
  l = make_list(3, my_list); // ( 3 )
  l = make_list(2, my_list); // ( 2 3 )
  l = make_list(1, my_list); // ( 1 2 3 )
}
```



Linked structures

- You can think of linked structures like a freight train
- Each piece of data is like a car in the train
- At the end of the train you will find a caboose

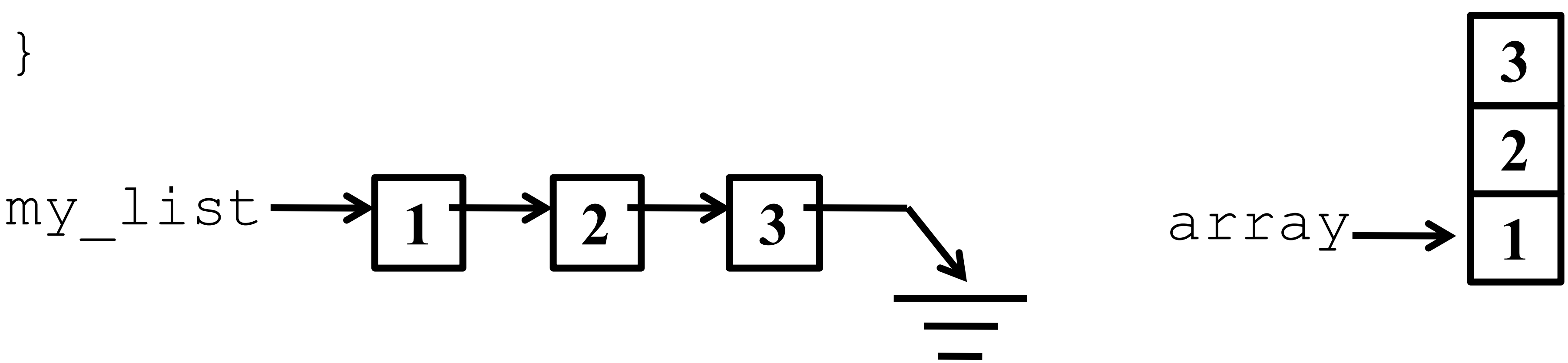




Sequence containers

- Array-based structures and linked structures have something in common: they are both *sequence containers*
- A sequence container allows us to access items sequentially
  - The order in which they are in the container

```
int main() {
  list_t my_list;
  int array[3] = {1, 2, 3};
}
```



Mutable vs. Immutable

- Today we will create a list abstract data type similar to the `list_t` type from project 2

- Key difference: `list_t` was *immutable*
  - You couldn't change items already in the list, only "glue more on"

- Our list today will be *mutable*
  - List items can be added, removed and modified

`IntList` abstraction

- Let's call our list `IntList`, since it will hold integers

```
class IntList {
  //OVERVIEW: a singly-linked list
  //...
```

`IntList` abstraction

- We will start with 4 operations

```
class IntList {
public:
  //EFFECTS:  returns true if the list is empty
  bool empty() const;

  //REQUIRES: list is not empty
  //EFFECTS: Returns a reference to the first element
  //  in the list
  int & front() const;

  //EFFECTS:  inserts datum into the front of the list
  void push_front(int datum);

  //REQUIRES: list is not empty
  //EFFECTS:  removes the item at the front of the list
  void pop_front();
```

Using `IntList`

- For example, we can use an `IntList` like this:

```
int main() {
  IntList l;              // ( )
  l.push_front(1);        // ( 1 )
  l.push_front(2);        // ( 2 1 )
  l.push_front(3);        // ( 3 2 1 )
  cout << l.front();      // 3
  l.pop_front();          // ( 2 1 )
  l.pop_front();          // ( 1 )
  l.pop_front();          // ( )
  l.empty();              // TRUE
  return 0;
}
```

List nodes

- We need to pick a concrete representation that stores a list in dynamically allocated `Node` variables

```
struct Node {
  Node *next;
  int   datum;
};
```

- Invariant: the `datum` field holds the integer datum of an element in the list
- Invariant: the `next` field points to the next `Node` in the list, or `0` (AKA `NULL`) if no such `Node` exists

List nodes

- Question: does any code outside the class need to know about the `Node` type?

```
struct Node {
  Node *next;
  int   datum;
};
```

List nodes

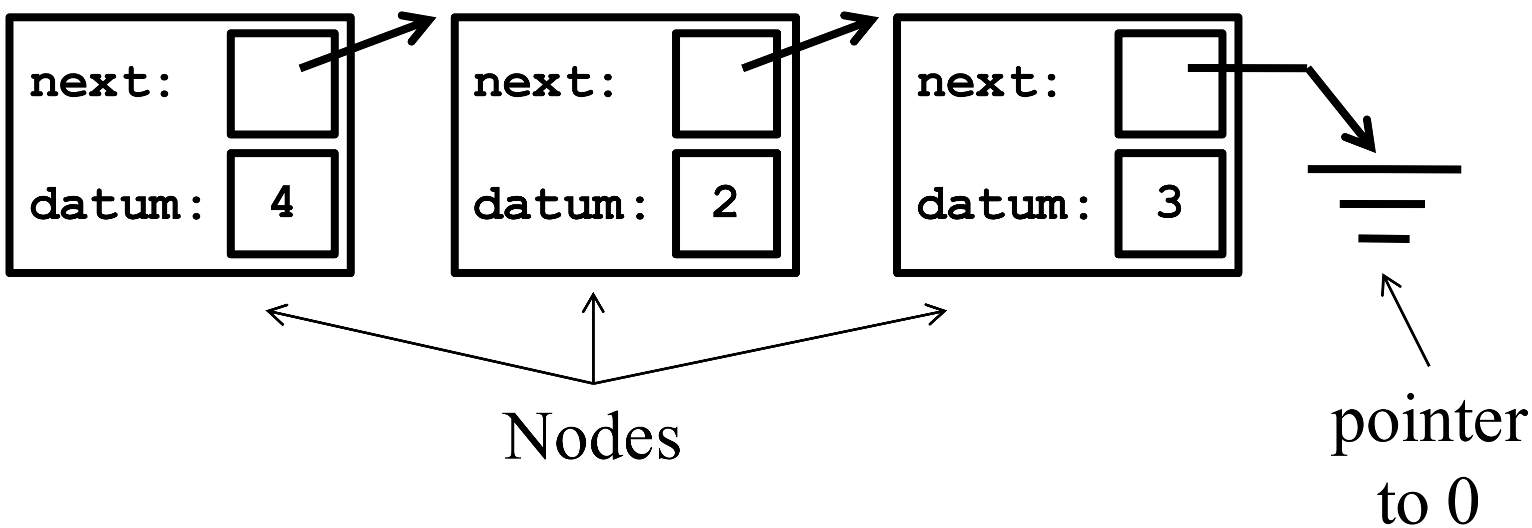
- Question: does any code outside the class need to know about the `Node` type?
- Answer: no, so make it `private`
- Types can be `private`, just like variables

```
class IntList {
  //...
 private:
  struct Node {
    Node *next;
    int   datum;
  };
};
```

Another way to think about it: we're not giving this class a `Node` variable, but rather describing what a future `Node` variable would look like

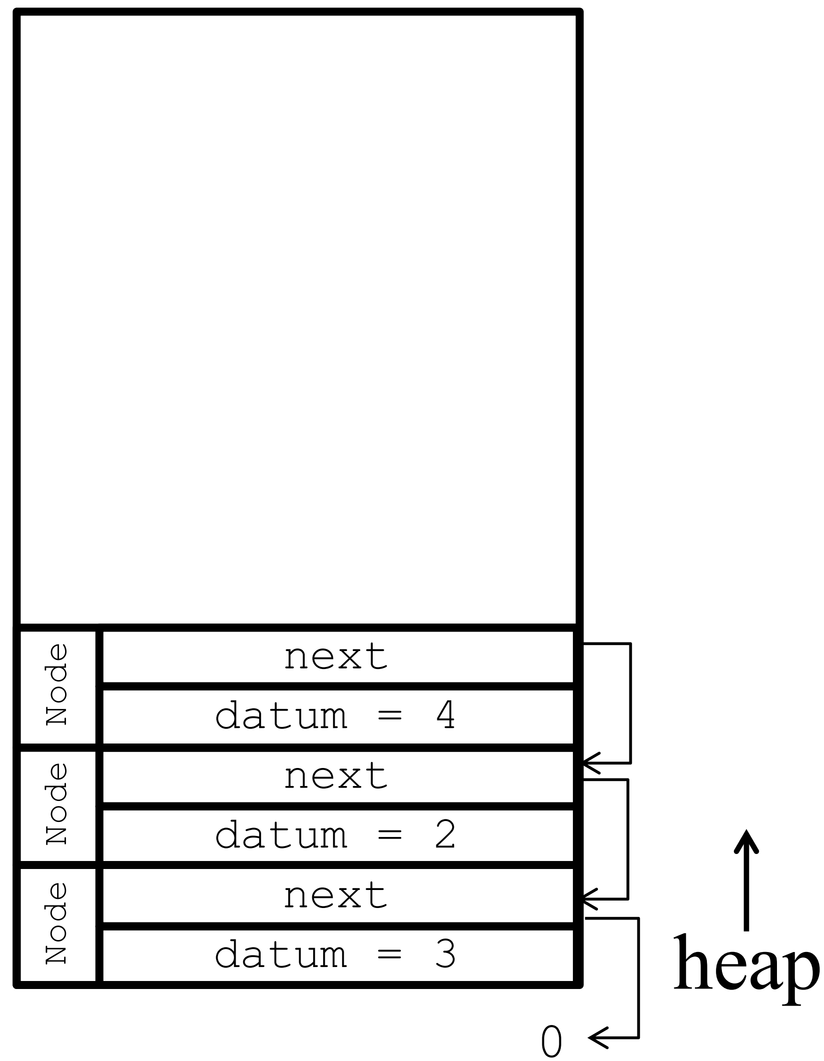
List nodes

- 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 list, we'll destroy the node that held it
- The concrete representation of the list (4 2 3) is:



List nodes

- In memory, the list (4 2 3) looks like this

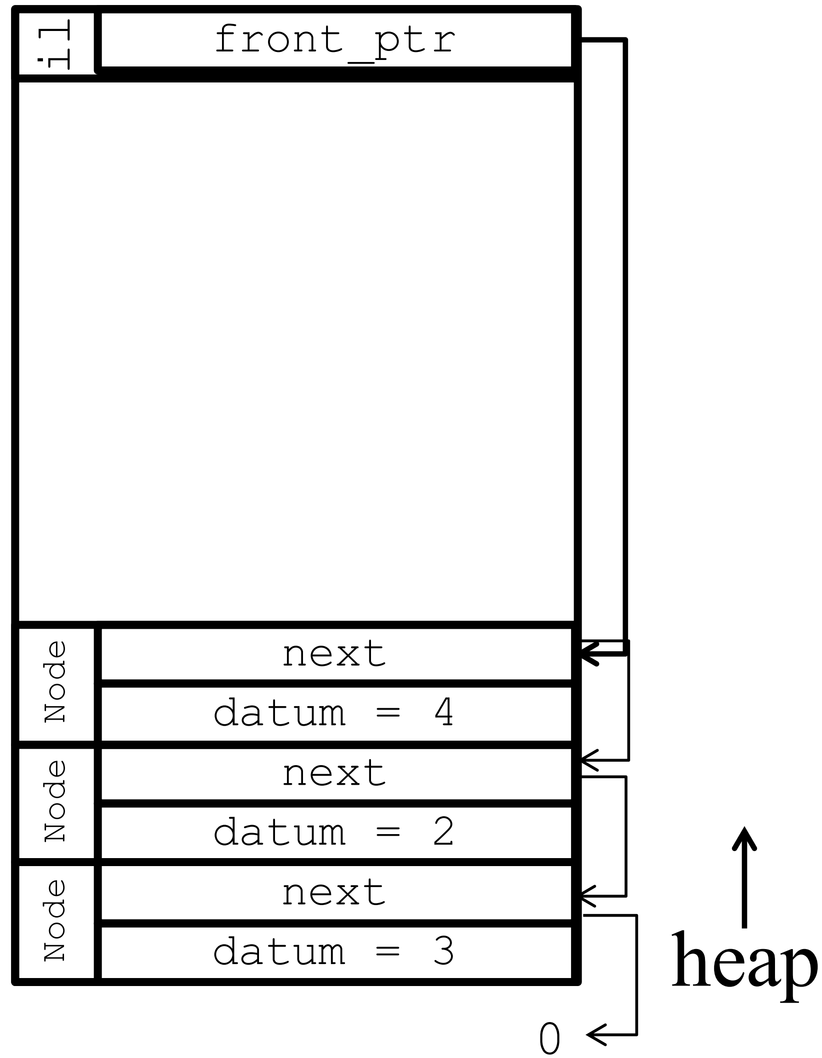


Member variables

- Now, we need a way to find the first node in list
- Store a pointer in a member variable

```
class IntList {
  //...
 private:
  struct Node {
    Node *next;
    int   datum;
  };
  Node *front_ptr;
};
```

- Representation invariant: `front_ptr` points to the first node in the sequence of nodes representing this `IntList`, or 0 if the list is empty



Implementing `empty()`

- Now, let's implement our member functions
- Recall the representation invariant
  - `front_ptr` points to the first node in the sequence of nodes representing this `IntList`, or 0 if the list is empty

```
bool IntList::empty() const {
  return front_ptr == 0;
}
```

Implementing `front()`

- Implement this function
- Use an assert statement to verify the REQUIRES clause

```
//REQUIRES: list is not empty
//EFFECTS: Returns a reference to the first
//  element in the list
int & front() const;
```

Implementing `front()`

- Implement this function
- Use an assert statement to verify the REQUIRES clause

```
//REQUIRES: list is not empty
//EFFECTS: Returns a reference to the first
//  element in the list
int & front() const {
  assert(!empty());
  return front_ptr->datum;
}
```

Using `front()`

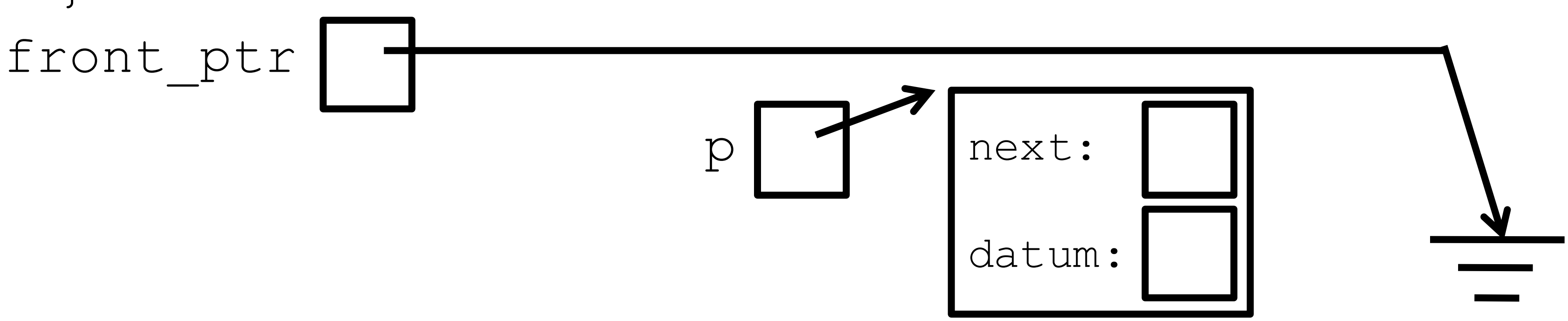
- We can both inspect and modify the item inside a node with `front()` because it returns a reference

```
int main() {
  IntList il;
  il.push_front(1);
  cout << il.front() << endl;  // 1
  il.front() = 17;
  cout << il.front() << endl;  // 17
}
```

Implementing `push_front()`

- When we insert an integer, we start out with the `front_ptr` field pointing to the current list
- The current list might be empty, or not
- The first thing we need to do is to create a new node to hold the new first element

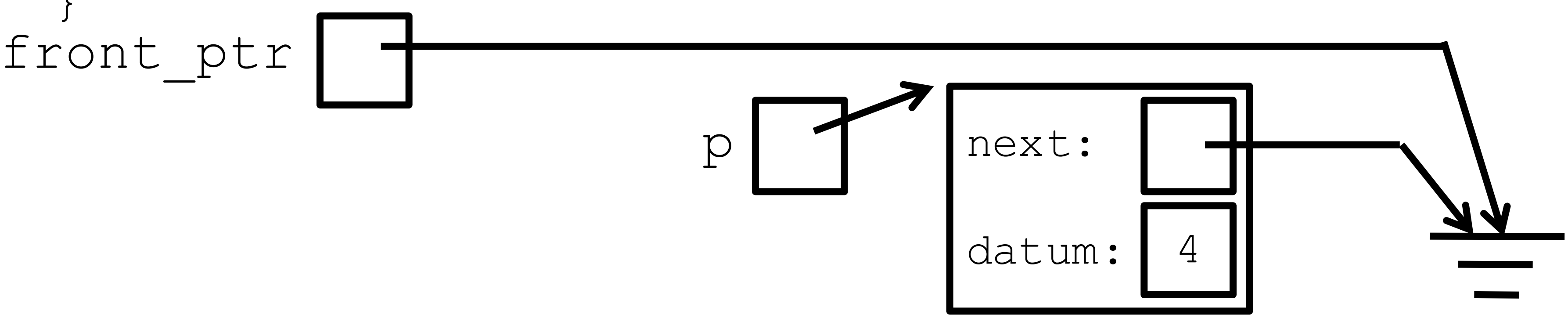
```
void IntList::push_front(int datum) {
  Node *p = new Node;
  //...
}
```



Implementing `push_front()`

- Next, we need to establish the invariants on the new node
  - Set `datum` field to new element
  - Set `next` field to the "rest of the list" (the beginning of the current list)

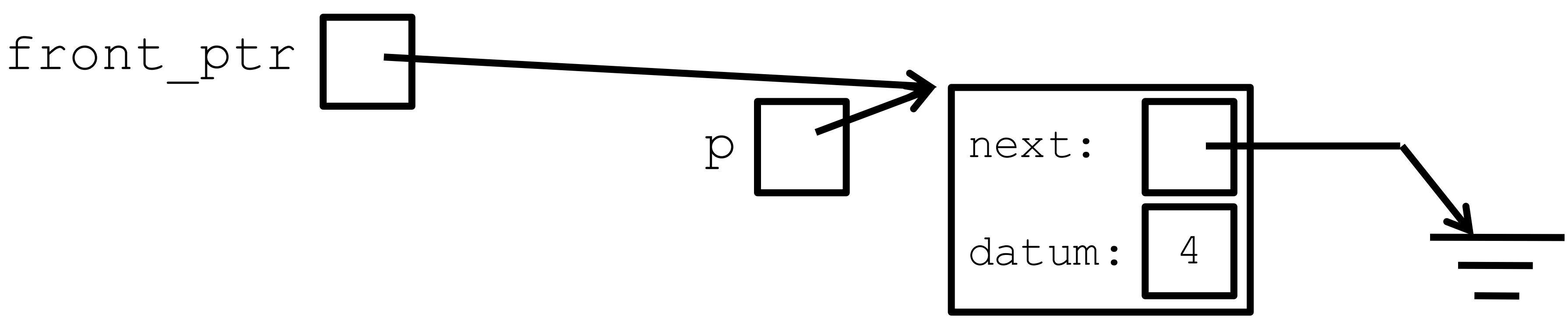
```
void IntList::push_front(int datum) {
  Node *p = new Node;
  p->datum = datum;
  p->next = front_ptr;
  //...
}
```



Implementing `push_front()`

- Finally, we need to fix the representation invariant
- `front_ptr` currently points to the wrong place
- Fix this

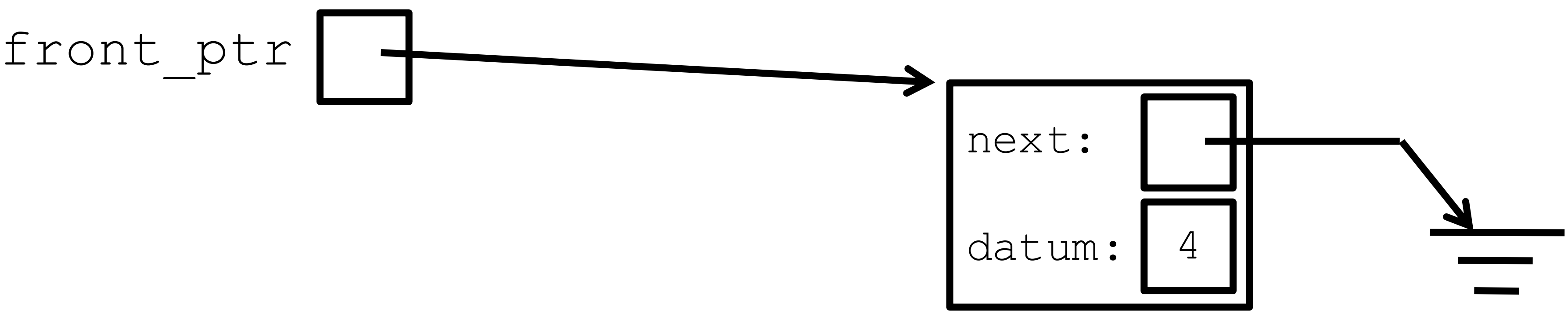
```
void IntList::push_front(int datum) {
  Node *p = new Node;
  p->datum = datum;
  p->next = front_ptr;
  front_ptr = p;
}
```



Implementing `push_front()`

- When the member function returns, `p` goes out of scope
- `front_ptr` does not, because it's a member variable
- The `Node` does not, because it's a dynamic variable

```
void IntList::push_front(int datum) {
  Node *p = new Node;
  p->datum = datum;
  p->next = front_ptr;
  front_ptr = p;
} //p goes out of scope
```



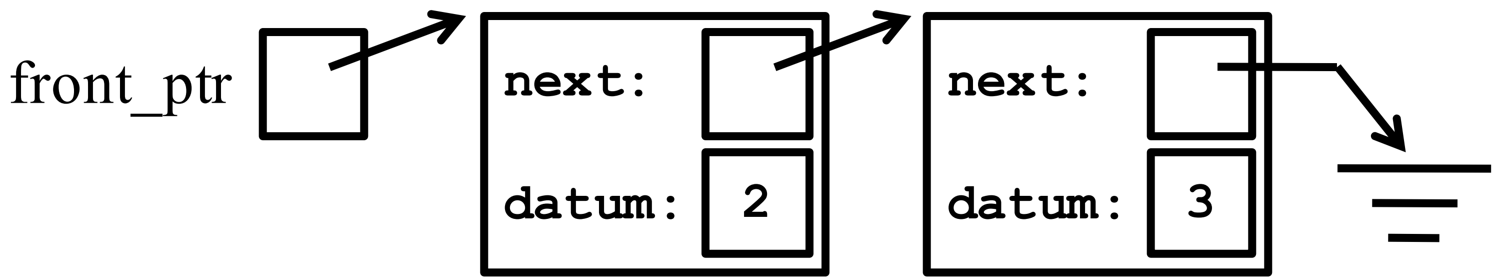
`push_front()` example

- Draw two diagrams for this code:
  - “Simple” diagram with boxes-and-arrows
  - Memory diagram showing the stack and the heap

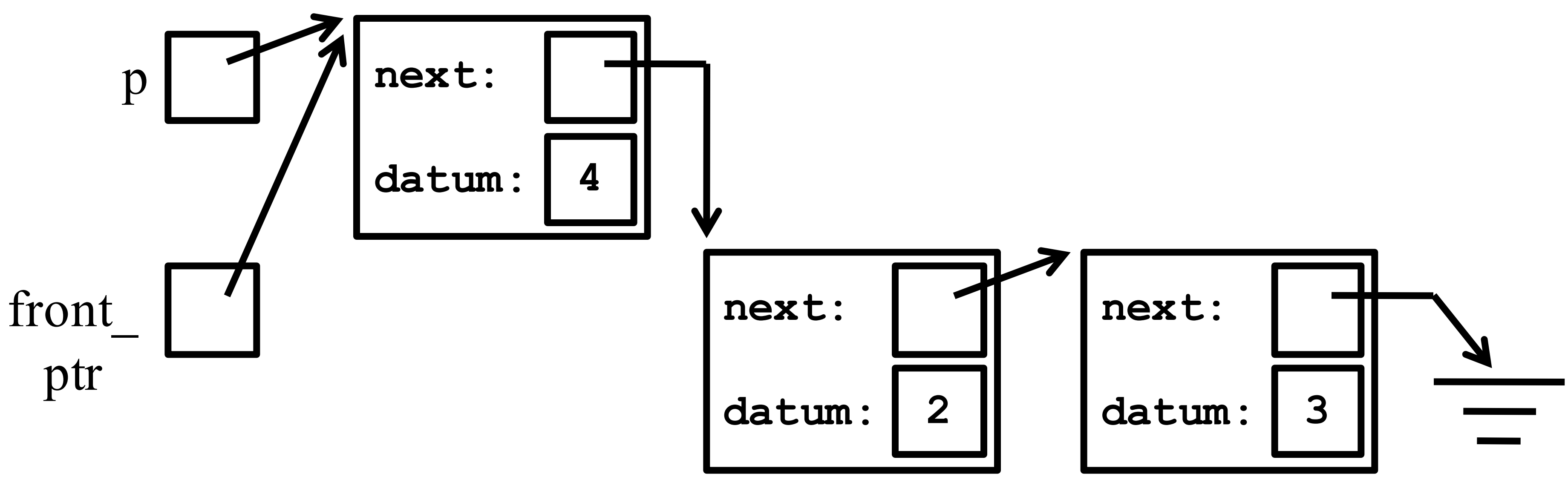
```
int main() {
  IntList il;
  il.push_front(3);
  il.push_front(2);
  il.push_front(4);
  return 0;
}
```

`push_front()` example

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

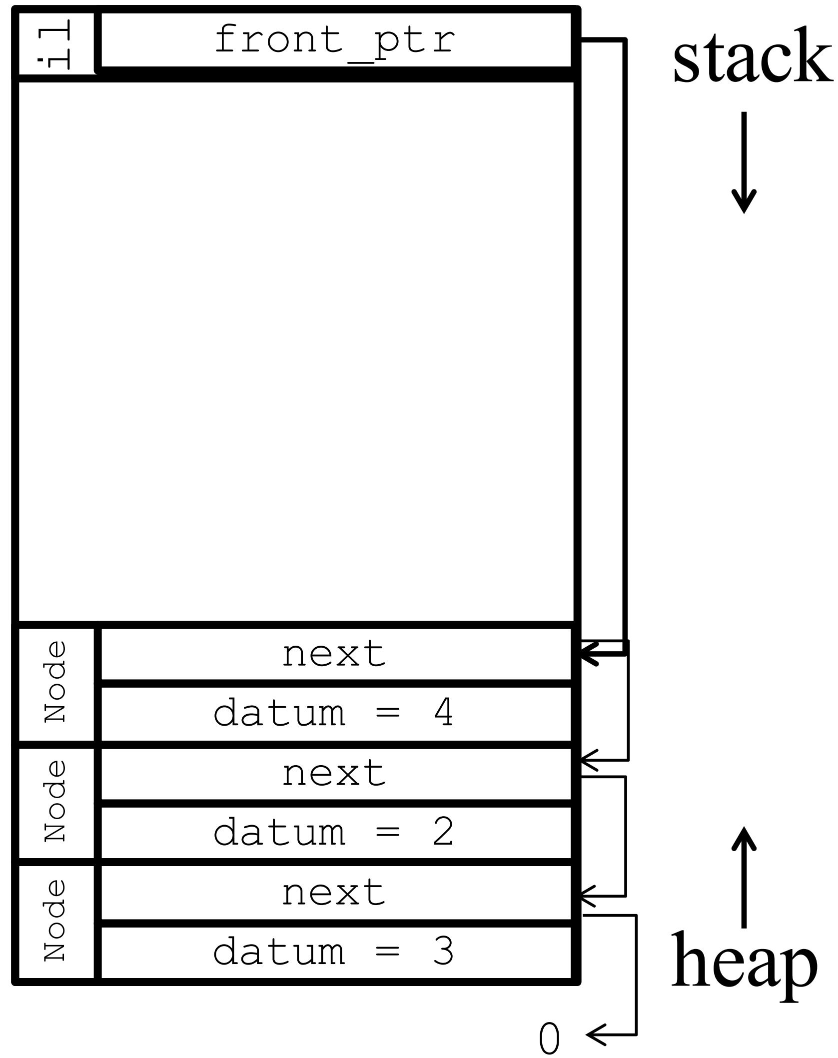


- And then the list's invariant :



`push_front()` example

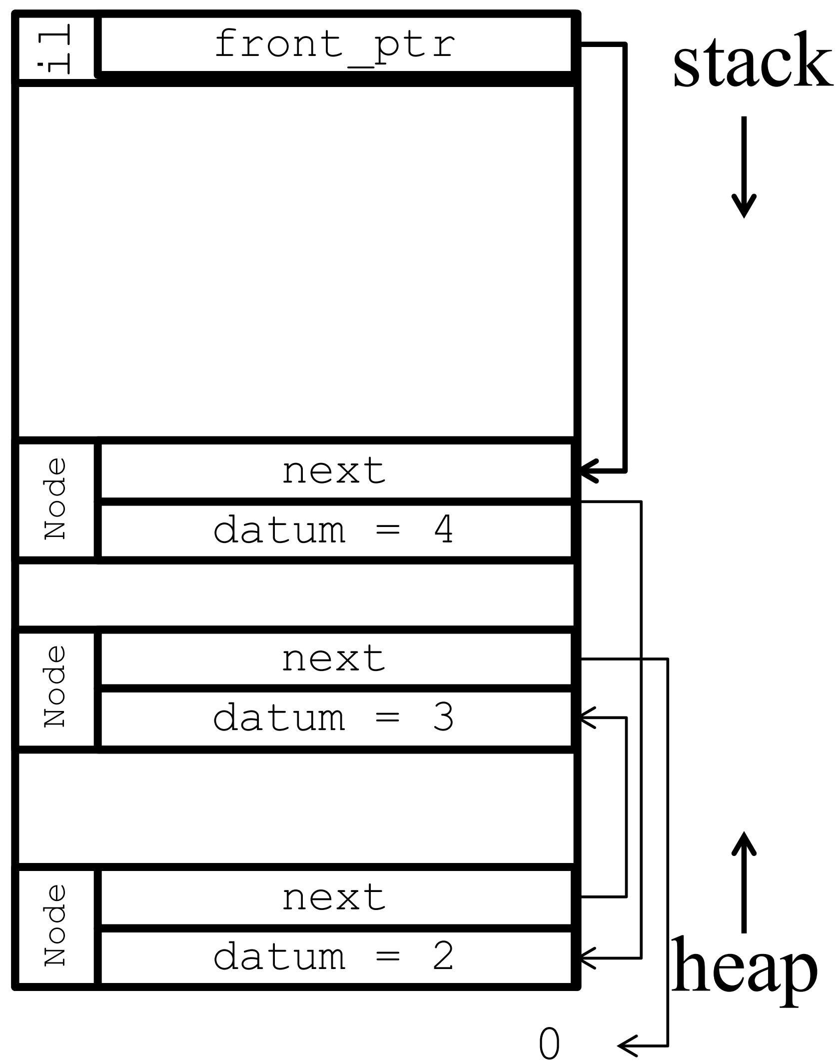
```
int main() {
  IntList il;
  il.push_front(3);
  il.push_front(2);
  il.push_front(4);
  return 0;
}
```



`push_front()` example

- This works too ☺

```
int main() {
  IntList il;
  il.push_front(3);
  il.push_front(2);
  il.push_front(4);
  return 0;
}
```



Implementing `pop_front()`

- Next, we'll implement `pop_front()` which removes one item from the list
- First, check the REQUIRES clause
- It doesn't make any sense to pop from an empty list!

```
//REQUIRES: list is not empty
//MODIFIES: this
//EFFECTS: removes the item at the front of the list
void IntList::pop_front() {
  assert(!empty());
  //...
}
```

Implementing `pop_front()`

- If we are removing the front node, we must delete it to avoid a memory leak
- We also need to advance the front_ptr to the next node in the list

```
void IntList::pop_front() {
  assert(!empty());
  delete front_ptr;
  front_ptr = front_ptr->next;
}
```

- What is **wrong** with this code?

Implementing `pop_front()`

```
void IntList::pop_front() {
  assert(!empty());
  delete front_ptr;
  front_ptr = front_ptr->next; //undefined!
}
```

- OK, let's try and fix it this way:

```
void IntList::pop_front() {
  assert(!empty());
  front_ptr = front_ptr->next;
  delete front_ptr; //we just deleted the new front
}                   //node, not the old one!
```

Implementing `pop_front()`

- If we are removing the front node, we must delete it to avoid a memory leak
- Unfortunately, we can't delete it before advancing the `front_ptr` pointer (since `front_ptr->next` would then be undefined)
- But, after we advance `front_ptr`, the removed node 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`

Implementing `pop_front()`

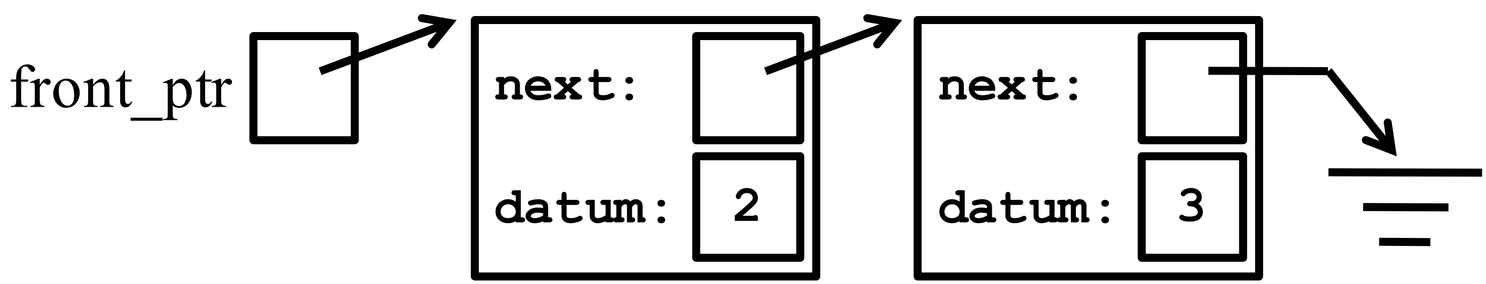
- Store pointer in `victim`, then delete the node **after** `front_ptr` is updated

```
void IntList::pop_front() {
  assert(!empty());
  Node *victim = front_ptr;
  front_ptr = front_ptr->next;
  delete victim; victim=0;
}
```

`pop_front()` example

```
void IntList::pop_front() {
  assert(!empty());
  Node *victim = front_ptr;
  front_ptr = front_ptr->next;
  delete victim; victim=0;
}
```

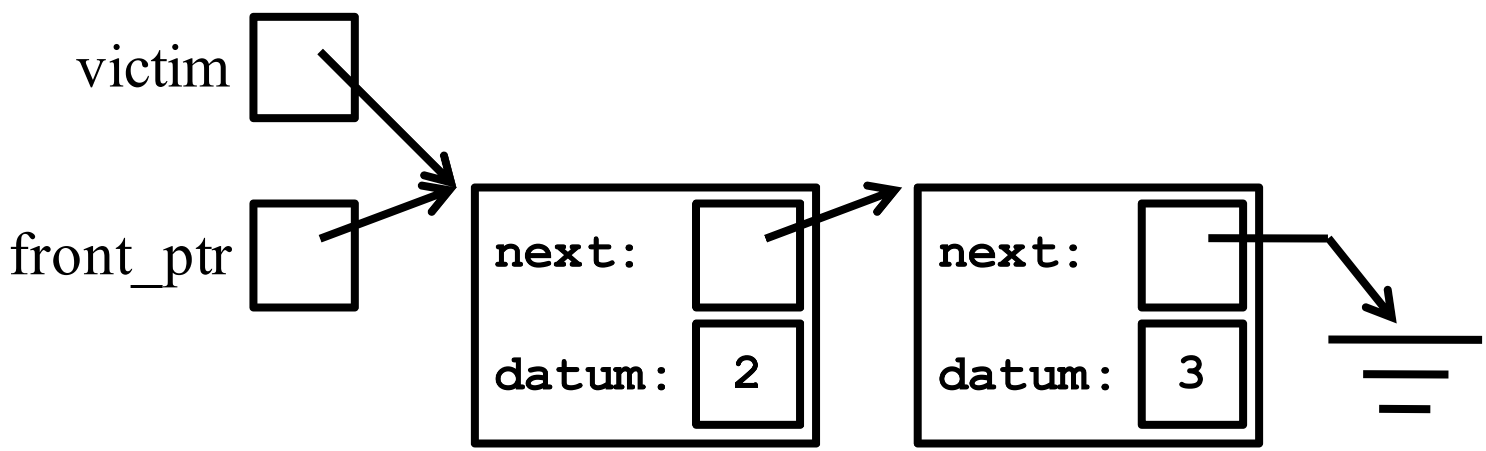
- Suppose the list has elements and `pop_front()` is called



`pop_front()` example

```
void IntList::pop_front() {
  assert(!empty());
  Node *victim = front_ptr;
  front_ptr = front_ptr->next;
  delete victim; victim=0;
}
```

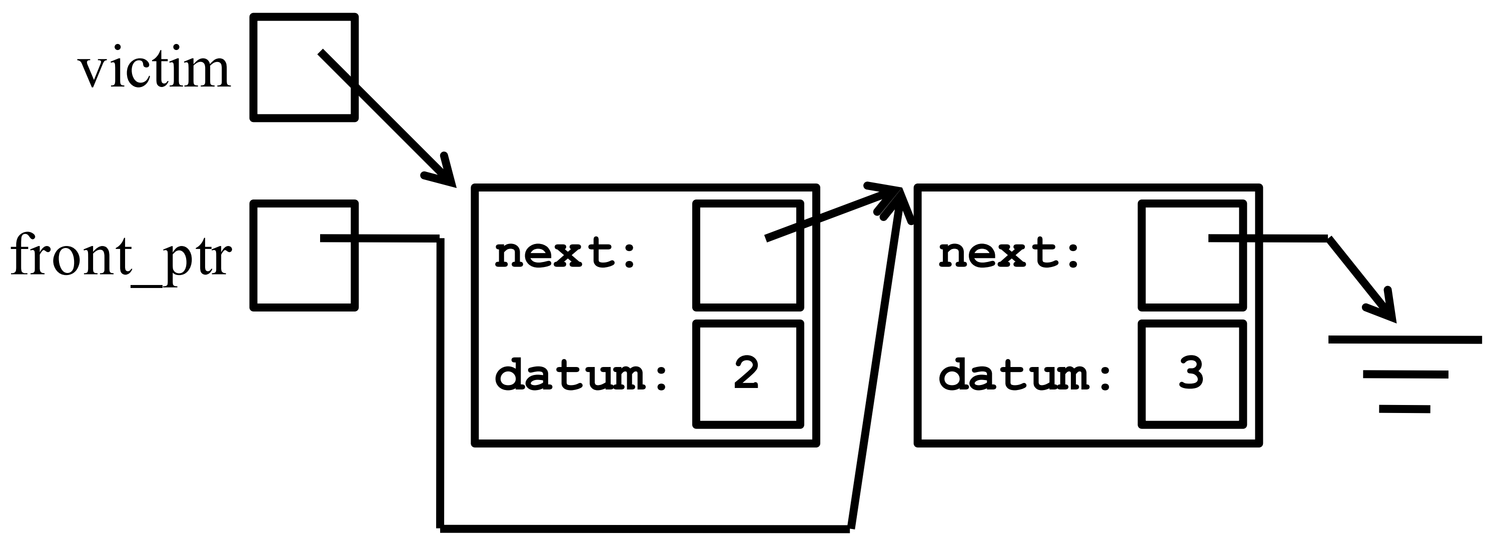
- Set the victim pointer to the node to be removed



`pop_front()` example

```
void IntList::pop_front() {
  assert(!empty());
  Node *victim = front_ptr;
  front_ptr = front_ptr->next;
  delete victim; victim=0;
}
```

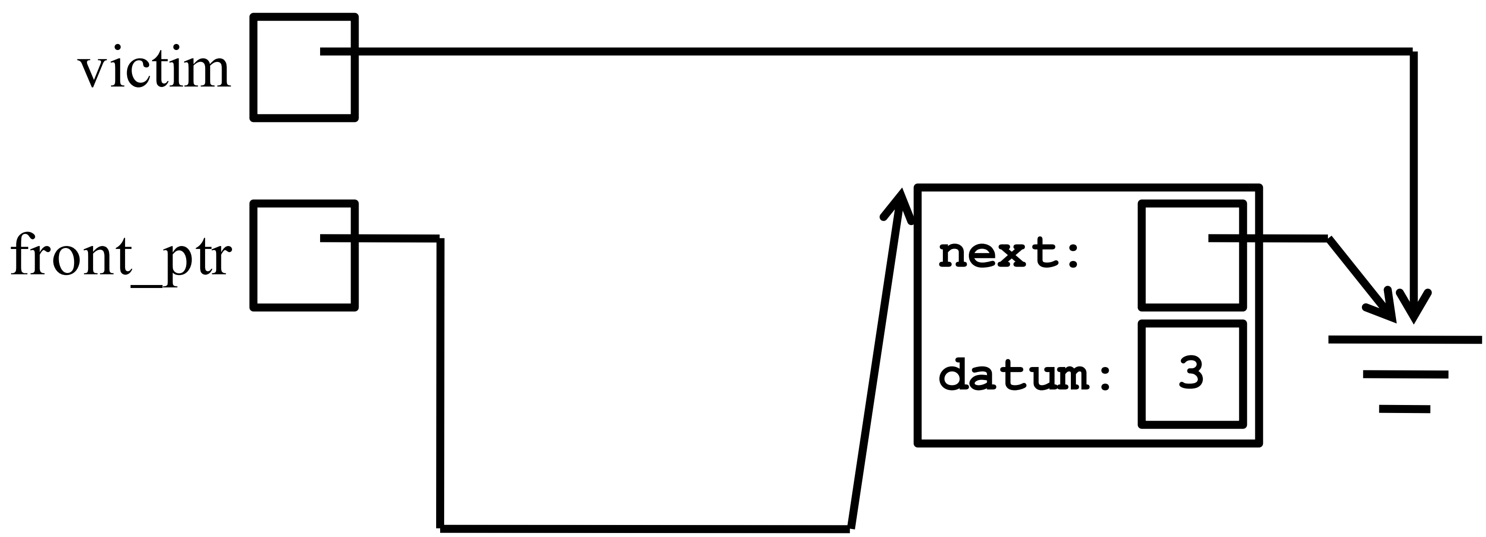
- Move the front pointer



`pop_front()` example

```
void IntList::pop_front() {
  assert(!empty());
  Node *victim = front_ptr;
  front_ptr = front_ptr->next;
  delete victim; victim=0;
}
```

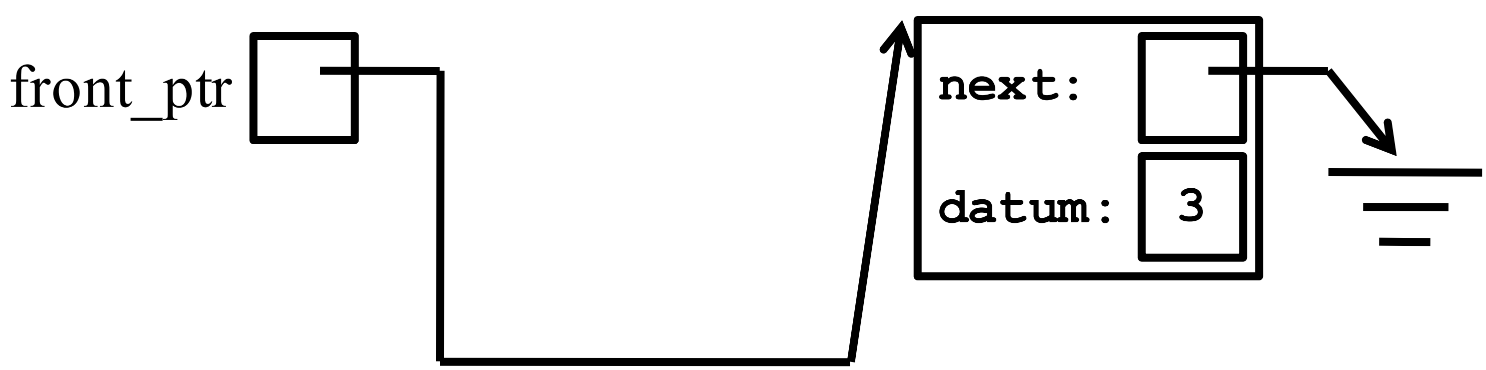
- Delete the victim node and set the pointer to zero



`pop_front()` example

```
void IntList::pop_front() {
  assert(!empty());
  Node *victim = front_ptr;
  front_ptr = front_ptr->next;
  delete victim; victim=0;
}
```

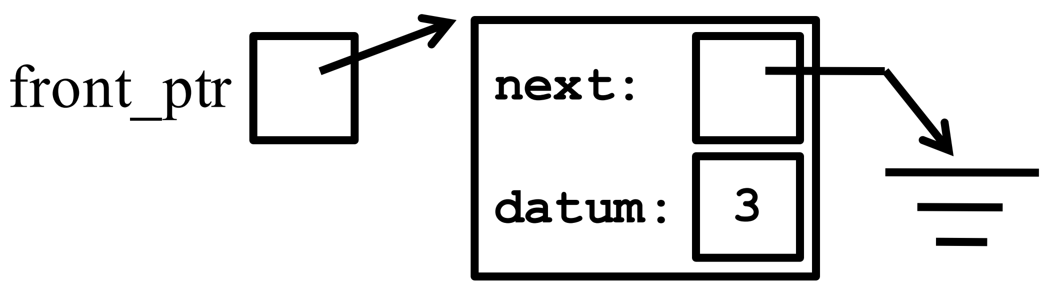
- victim pointer goes out of scope



`pop_front()` example

```
void IntList::pop_front() {
  assert(!empty());
  Node *victim = front_ptr;
  front_ptr = front_ptr->next;
  delete victim; victim=0;
}
```

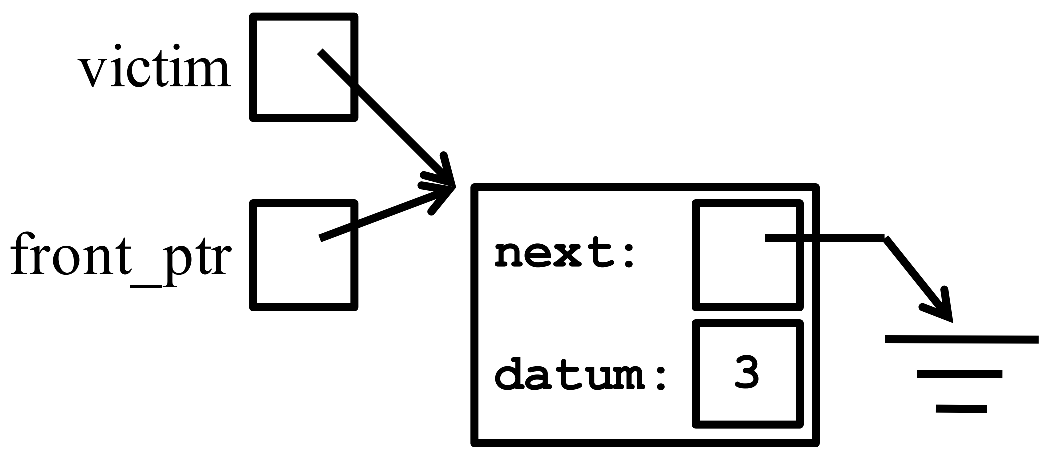
- Likewise, if the list had only a single element



`pop_front()` example

```
void IntList::pop_front() {
  assert(!empty());
  Node *victim = front_ptr;
  front_ptr = front_ptr->next;
  delete victim; victim=0;
}
```

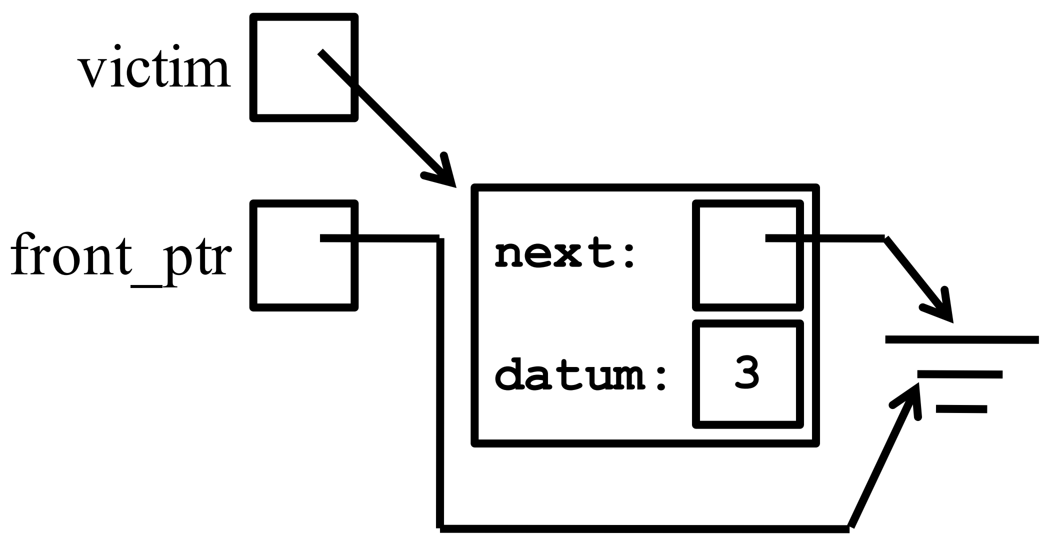
- Set the victim pointer to the node to be removed



`pop_front()` example

```
void IntList::pop_front() {
  assert(!empty());
  Node *victim = front_ptr;
  front_ptr = front_ptr->next;
  delete victim; victim=0;
}
```

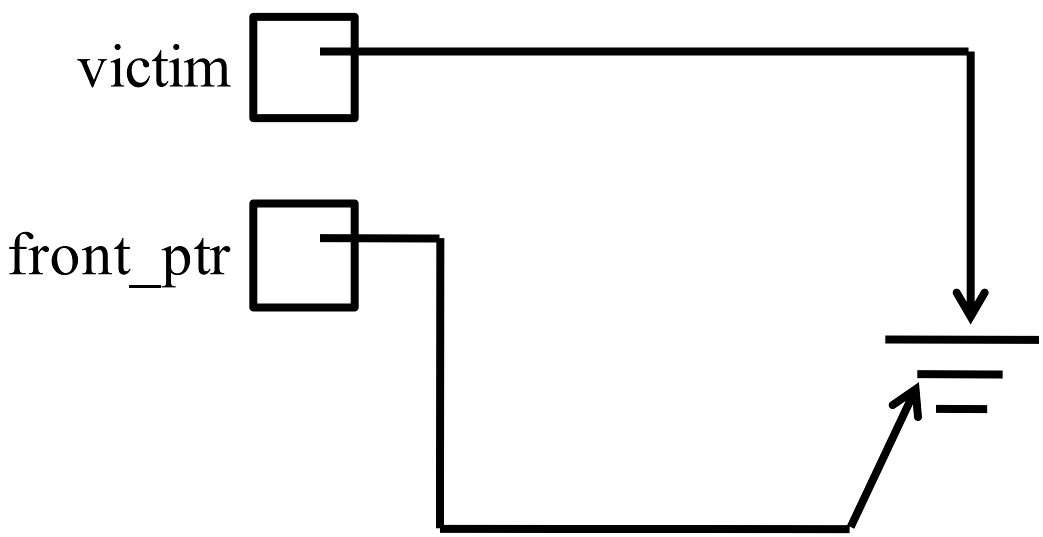
- Move the front pointer



`pop_front()` example

```
void IntList::pop_front() {
  assert(!empty());
  Node *victim = front_ptr;
  front_ptr = front_ptr->next;
  delete victim; victim=0;
}
```

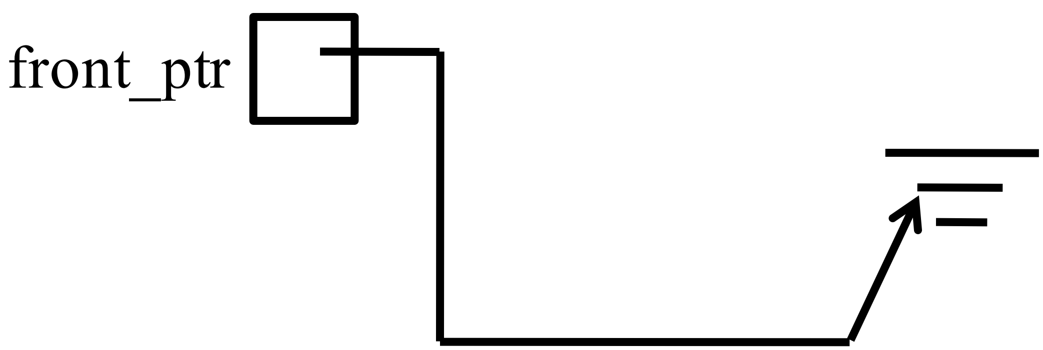
- Delete the victim node and set the pointer to zero



pop_front() example

```
void IntList::pop_front() {
  assert(!empty());
  Node *victim = front_ptr;
  front_ptr = front_ptr->next;
  delete victim; victim=0;
}
```

- victim pointer goes out of scope



Default constructor

- Now, we need a default constructor
- The default constructor should establish the representation invariant for an empty list
- Recall the representation invariant: `front_ptr` points to the first node in the sequence of nodes representing this `IntList`, or `0` if the list is empty

- Write the default constructor

Default constructor

```
class IntList {
public:
  IntList();
  //...
}

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

Walking the list

- We can use pointers to visit each node in a list
- Implement this member function using a for loop

```
class IntList {
public:
  //EFFECTS: prints the list to stdout
  //  e.g., "1 2 3 "
  void print() const;
  //...
}
```

The Big Three

- Question: do we need the Big Three? Why or why not?

- Destructor
- Copy constructor
- Overloaded assignment operator

The Big Three

- Here's what we need each function to do:

- Destructor
  1. Remove all nodes
- Copy constructor
  1. Initialize member variables
  2. Copy all nodes from other list
- Overloaded assignment operator
  1. Remove all nodes from this list
  2. Copy all nodes from other list

The Big Three

- Create private member functions to avoid copy-pasted code

- Destructor
  1. Remove all nodes
- Copy constructor
  1. Initialize member variables
  2. Copy all nodes from other list
- Overloaded assignment operator
  1. Remove all nodes from this list
  2. Copy all nodes from other list

`pop_all()`

`push_all(...)`

The Big Three

```
class IntList {
  //...
 private:
  //MODIFIES: this
  //EFFECTS:  removes all nodes
  void pop_all();

  //MODIFIES: this
  //EFFECTS:  copies all nodes from other list
  //  to this list
  void push_all(const IntList &other);
};
```

The Big Three

- We already have a function to remove one node, `pop_front()`, let's reuse it

```
void IntList::pop_all() {
  while (!empty()) {
    pop_front();
  }
}
```

The Big Three

- We already have a function to insert one node, `push_front()`
- What happens if we try to reuse this function to implement `push_all()` ?

```
void IntList::push_all(const IntList &other) {
  for (Node *p=other.front_ptr; p!=0; p=p->next) {
    push_front(p->datum);
  }
}
```

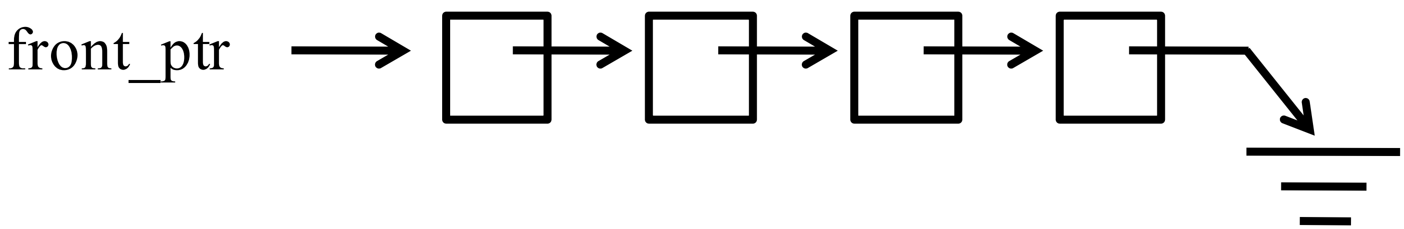
The Big Three

- We could easily write `push_all()` if we had a function `push_back()`

```
void IntList::push_all(const IntList &other) {
  for (Node *p=other.front_ptr; p!=0; p=p->next)
    push_back(p->datum);
}
```

Implementing `push_back()`

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



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

Implementing `push_back()`

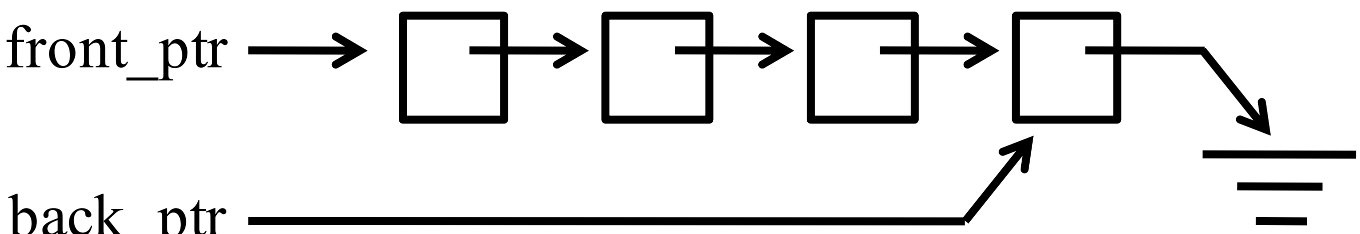
- The new representational invariant has **two** node pointers:

```
class IntList {
//...
private:
  Node *front_ptr;
  Node *back_ptr;
};
```

- The invariant for `front_ptr` is unchanged
- The invariant for `back_ptr:` `back_ptr` points to the last node of the list if it is not empty, and `0` otherwise

Implementing `push_back()`

- So, in an empty list, both data members point to `0`
- However, if the list is non-empty, they look like this:



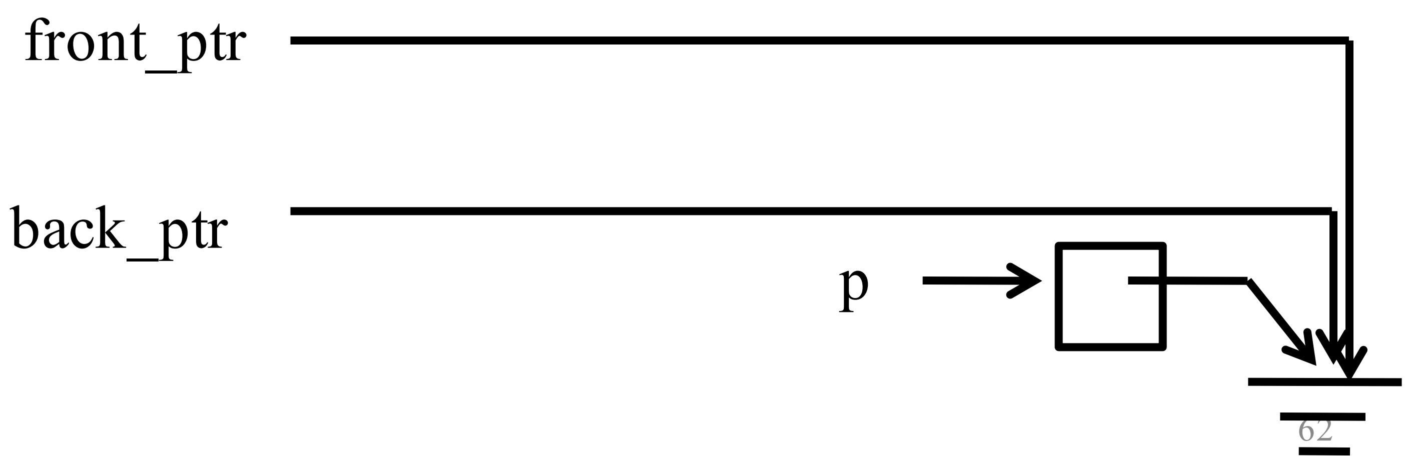
- Note: Adding this new data member requires that we modify `push_front()` and `pop_front()`
- In lecture, we'll only write `push_back()`

Don't worry, you'll get to do this in Project 5 ☺

Implementing `push_back()`

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

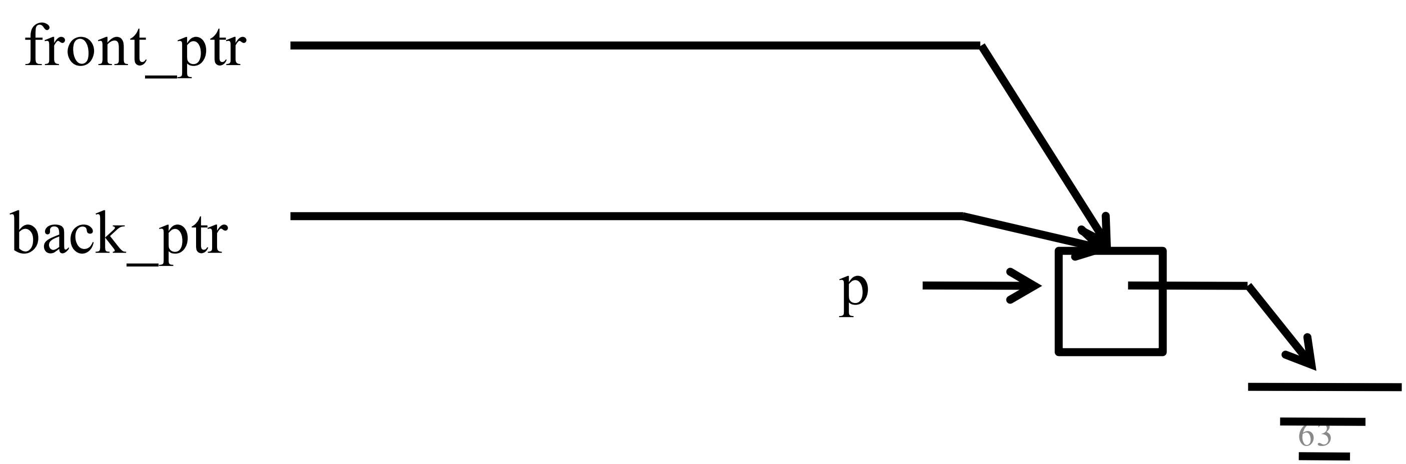
```
void IntList::push_back(int datum) {
  Node *p = new Node;
  p->datum = datum;
  p->next = 0;
  //...
}
```



Implementing `push_back()`

- Next, handle the empty list case

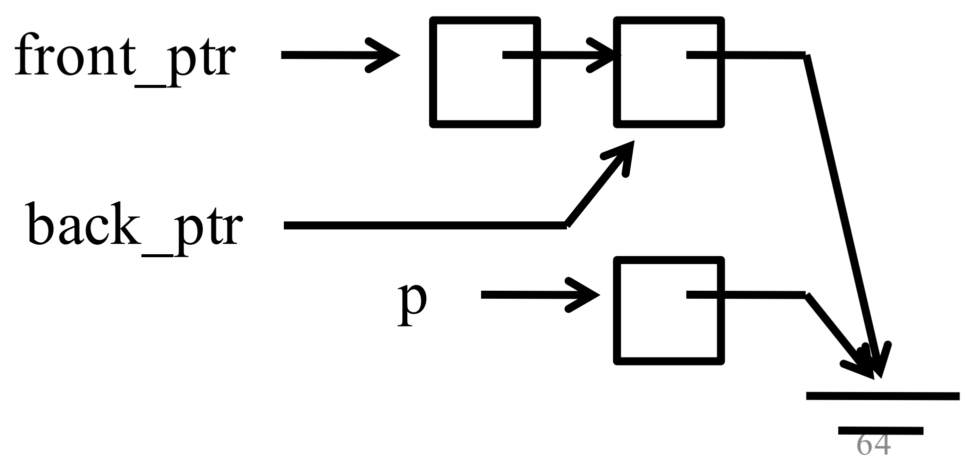
```
void IntList::push_back(int datum) {
  Node *p = new Node;
  p->datum = datum;
  p->next = 0;
  if (empty()) {
    front_ptr = back_ptr = p;
  } else {
    //...
  }
}
```



Implementing `push_back()`

- Finally, handle the non-empty list case

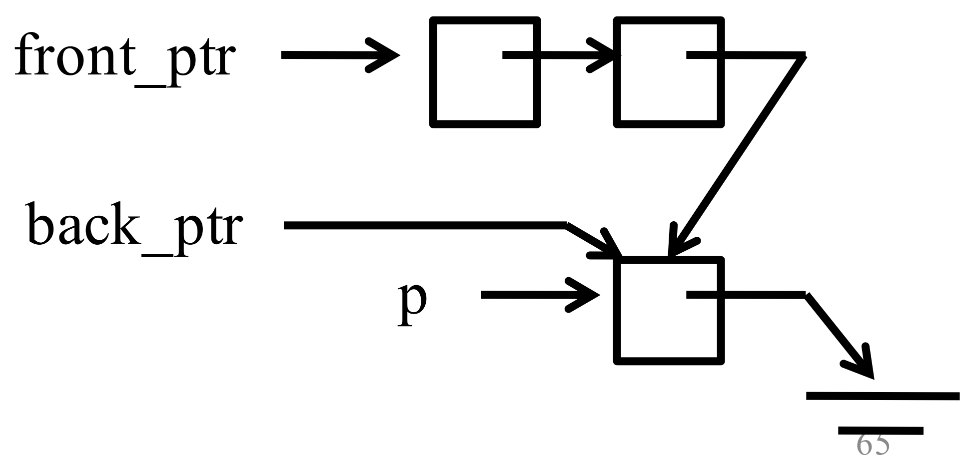
```
void IntList::push_back(int datum) {
  Node *p = new Node;
  p->datum = datum;
  p->next = 0;
  if (empty()) {
    front_ptr = back_ptr = p;
  } else {
    back_ptr->next = p;
    back_ptr = p;
  }
}
```



Implementing `push_back()`

- Finally, handle the non-empty list case

```
void IntList::push_back(int datum) {
  Node *p = new Node;
  p->datum = datum;
  p->next = 0;
  if (empty()) {
    front_ptr = back_ptr = p;
  } else {
    back_ptr->next = p;
    back_ptr = p;
  }
}
```



The Big Three

- Now that `push_back()` is finished, `push_all()` works too

```
void IntList::push_all(const IntList &other) {
  for (Node *p=other.front_ptr; p; p=p->next)
    push_back(p->datum);
}
```

The Big Three

- Back to the Big Three: we'll use `pop_all()` and `push_all()` to implement these methods

- Destructor
  1. Remove all nodes
- Copy constructor
  1. Initialize member variables
  2. Copy all nodes from other list
- Overloaded assignment operator
  1. Remove all nodes from this list
  2. Copy all nodes from other list

Destructor

- The destructor

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

```
void IntList::pop_all() {
  while (!empty()) {
    pop_front();
  }
}
```

```
void IntList::pop_front() {
  assert(!empty());
  Node *victim = front_ptr;
  front_ptr = front_ptr->next;
  delete victim; victim=0;
}
```

Copy constructor

- For the copy constructor, we need to do two things:
  1. Initialize member variables
  2. Copy all nodes from other list

```
IntList::IntList(const IntList &other)
  : front_ptr(0), back_ptr(0) {
  push_all(other);
}
```

Assignment operator

- The assignment operator must
  1. Remove all nodes from this list
  2. Copy all nodes from other list
- Also, it has to ensure that there is no self assignment (this could cause problems)

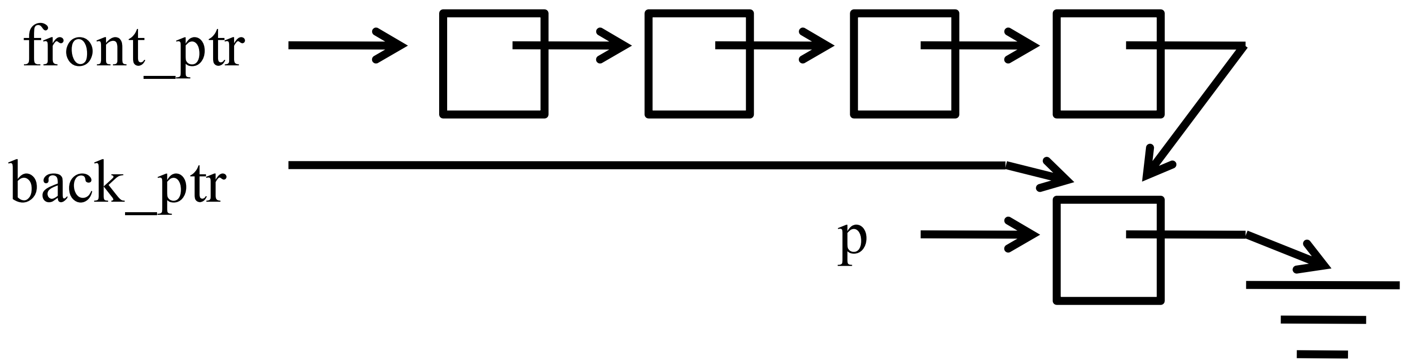
```
IntList & IntList::operator= (const IntList &rhs) {
  if (this == &rhs) return *this;
  pop_all();
  push_all(rhs);
  return *this;
}
```

Singly linked vs. doubly linked lists

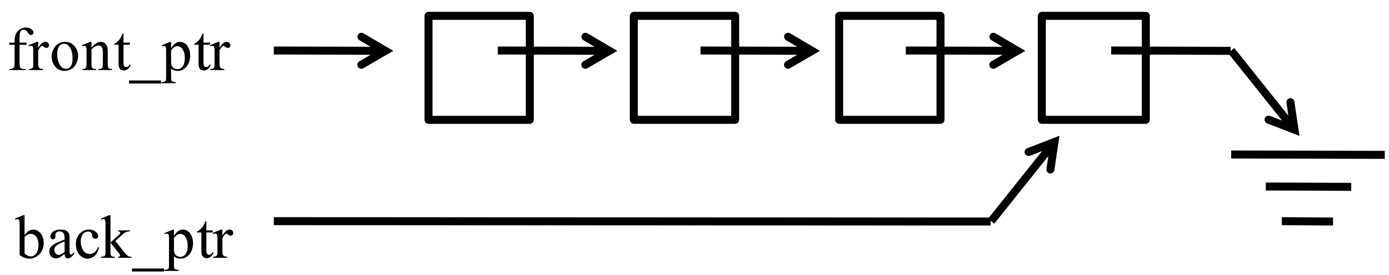
- We have now implemented three member functions, plus constructors and the Big Three
  - `push_front()`, `push_back()` and `pop_front()`
- What about `pop_back()` ?

Singly linked vs. doubly linked lists

- `push_back()` is efficient, but only for insertion

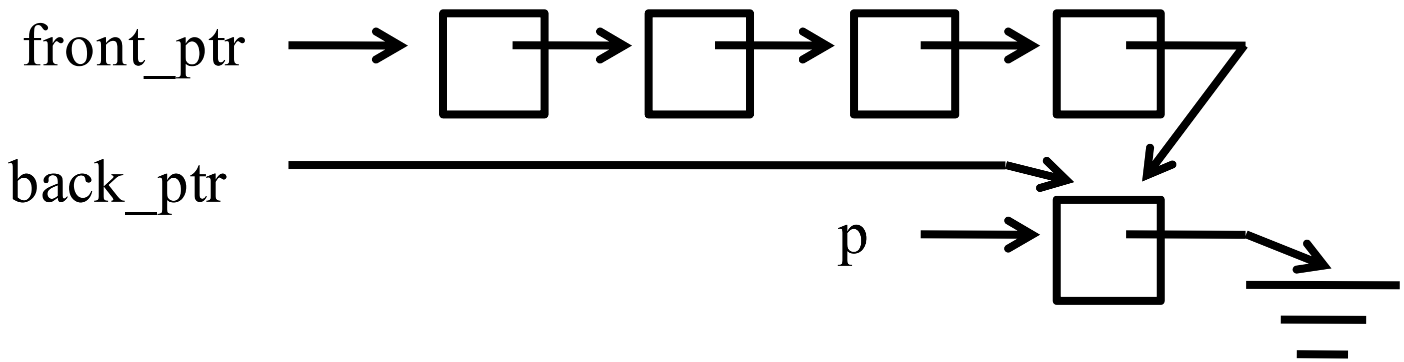


- Why is removal from the end expensive?

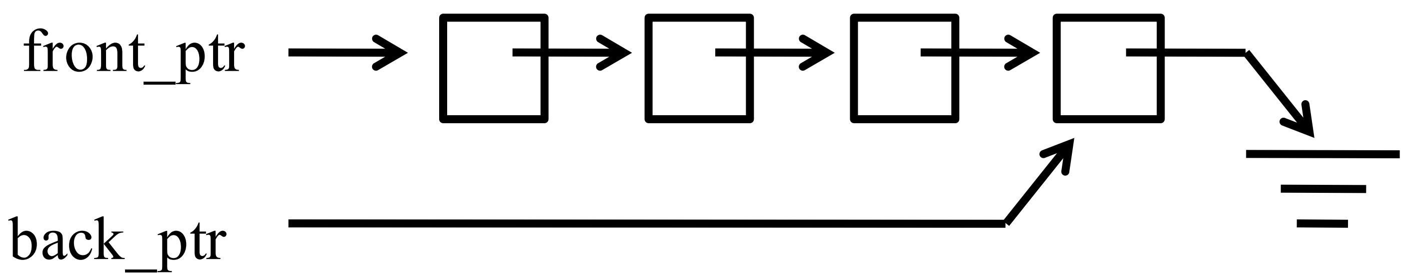


Singly linked vs. doubly linked lists

- `push_back()` is efficient, but only for insertion



- Why is removal from the end expensive? We have to inspect every element to set the new address for `back_ptr`.



Singly linked vs. doubly linked lists

- 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 yet again.
- In our new representation, a node is:

```
struct Node {
  Node *next;
  Node *prev;
  int   datum;
}
```

- The `next` and `datum` fields stay the same.
- The `prev` field's invariant is:
  - `prev` points to the previous node in the list, or `0` if no such node exists

Singly linked vs. doubly linked lists

- With this representation, an empty list is unchanged.

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