EECS 280 – Lecture 15

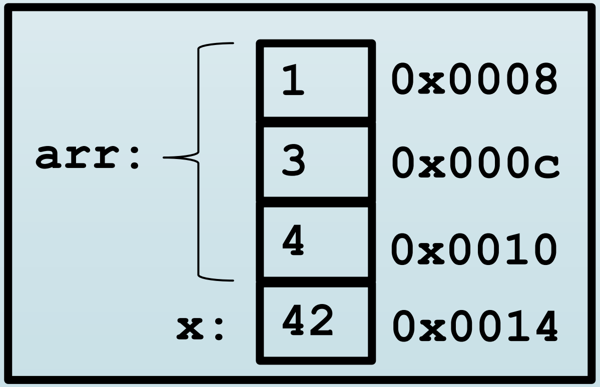
Linked Lists

3/14/2022

Sequential Containers

- Allow for sequential access of elements.
- Maintain the order of elements.

- How can we represent a sequential container?
- One option: store elements contiguously in memory so they are naturally in order.
  - This is how arrays and `std::vector` work.
  - Example:
```
int arr[3] = {1,3,4};
int x = 42;
```



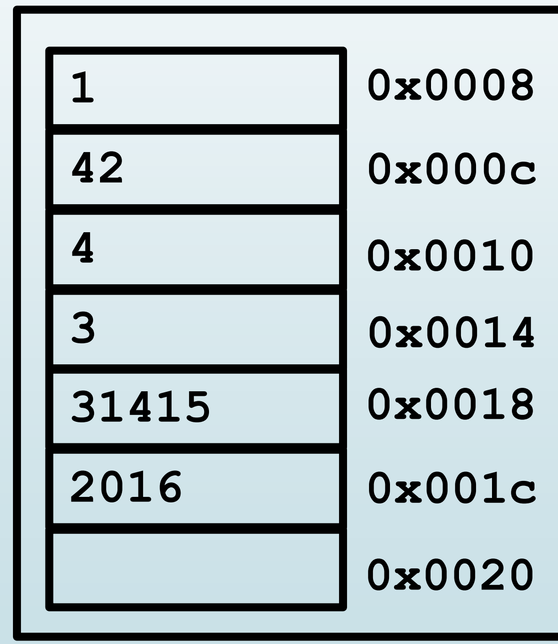
Using Contiguous Memory

- Contiguous memory allows indexing through pointer arithmetic, but it has some drawbacks…

- Inserting a new element into the middle of the sequence requires shifting over elements.
- Increasing the capacity requires allocating an entirely new chunk of memory (e.g. `grow()` for `UnsortedSet`).

Storing Elements Non-Contiguously

- How can we store a sequence without needing a contiguous chunk of memory?
- We can no longer just move forward one space in memory to get to the next element.
- Instead, we must somehow also keep track of the next element at each point in the list.
- Any ideas for how to do this?
  - Pointers!
  - Each "piece" of the list includes a **datum**, but also a **next pointer** containing the address of the next "piece".



Nodes

- Each "piece" of the list includes a **datum**, but also a **next pointer** containing the address of the next "piece".
- We'll call these "pieces" **nodes**.
- Let's use a `struct` to represent each node.
  - Groups together the datum and next pointer.
  - It's "Plain Old Data" (POD). No need for a class.
  - For simplicity, we'll just work with `int`s for now[1].

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

Used to store an element of the list.

Contains the address of the next node in the list.

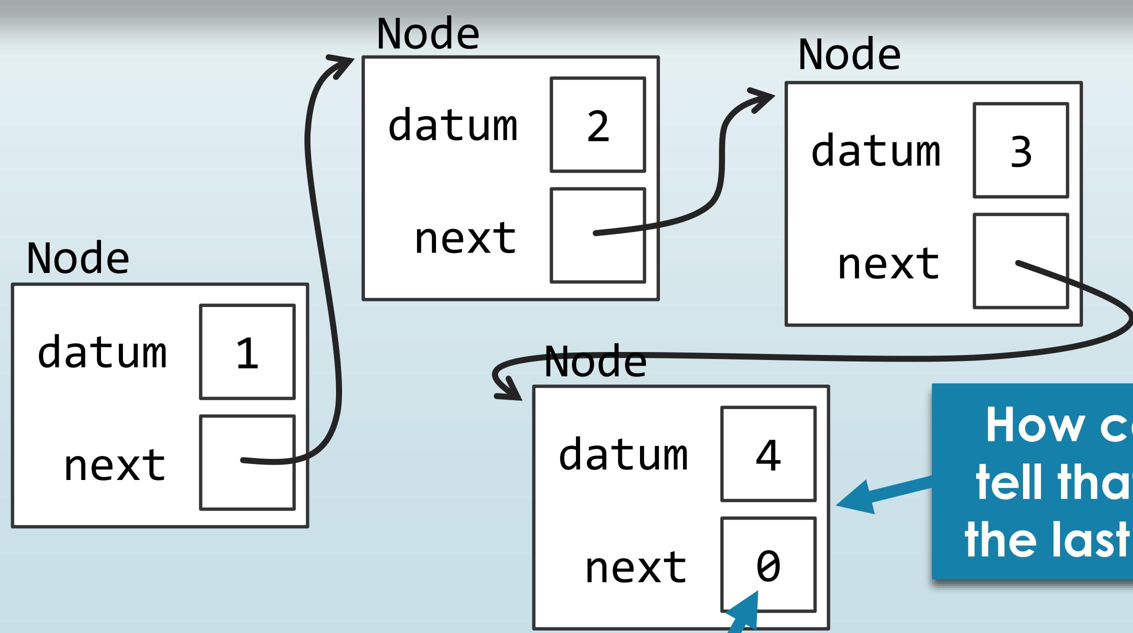
[1] In project 4, you'll write a generic linked list class template.

Nodes

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

Used to store an element of the list.

Contains the address of the next node in the list.



How can we tell that this is the last node?

Use a null pointer (address 0) as a sentinel!

Linked List Data Representation

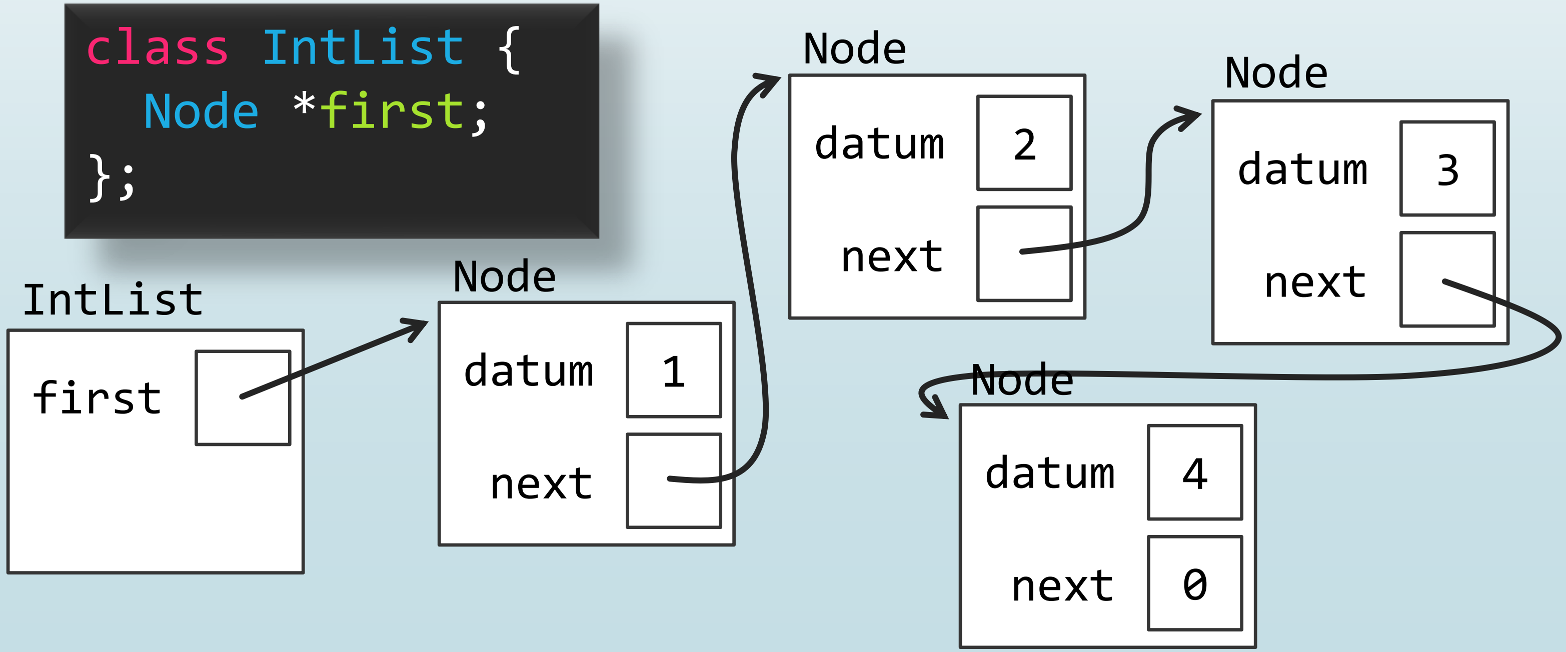
```
struct Node {
  int datum;
  Node *next;
};
```

Used to store an element of the list. (datum)

Contains the address of the next node in the list. (next)

- Let's also use a `class` to represent an entire list.

```
class IntList {
  Node *first;
};
```



The IntList Interface

```
class IntList {
public:
  // EFFECTS: constructs an empty list
  IntList();

  // EFFECTS: returns true if the list is empty
  bool empty() const;

  // REQUIRES: the list is not empty
  // EFFECTS:  Returns (by reference) the first element
  int & front();

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

  // REQUIRES: the list is not empty
  // EFFECTS:  removes the first element
  void pop_front();
  ...
};
```

Using an `IntList`

Question: Does the outside world need to know about Node?

```
int main() {
  IntList list;           // ( )
  list.push_front(1);    // ( 1 )
  list.push_front(2);    // ( 2 1 )
  list.push_front(3);    // ( 3 2 1 )

  cout << list.front(); // 3

  list.front() = 4;       // ( 4 2 1 )

  list.pop_front();       // ( 2 1 )
  list.pop_front();       // ( 1 )
  list.pop_front();       // ( )

  cout << list.empty(); // true (or 1)
}
```

Information Hiding

- Put the **`Node`** struct itself inside the **`IntList`** class.
- **`Node`** can only be used inside the class and its member functions. This is good – it's an implementation detail.

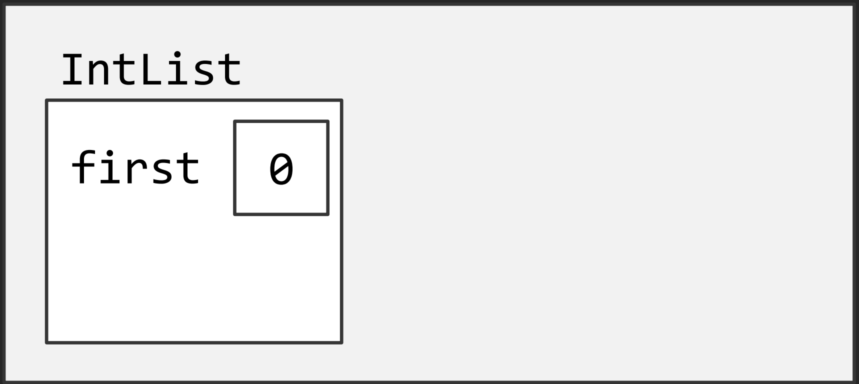
```
struct Node {
  int datum;
  Node *next;
};

class IntList {
private:
  Node *first;
};
```

```
class IntList {
private:

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

  Node *first;
};
```

This is called a "nested" `class` or `struct`.

Implementing `IntList`: Constructor

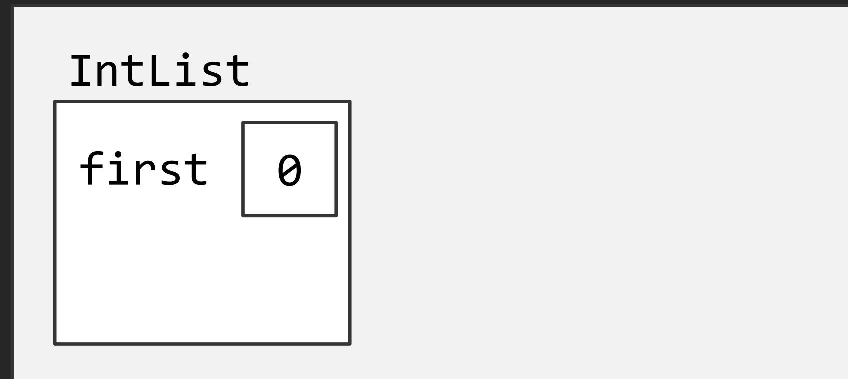
```
class IntList {
private:
  struct Node {
    int datum;
    Node *next;
  };
  Node *first;

public:
  // EFFECTS: constructs an empty list
  IntList() : first(nullptr) { }

  ...
};
```



Sets the `first` pointer to null to indicate an empty list.

Implementing `IntList`: `empty`

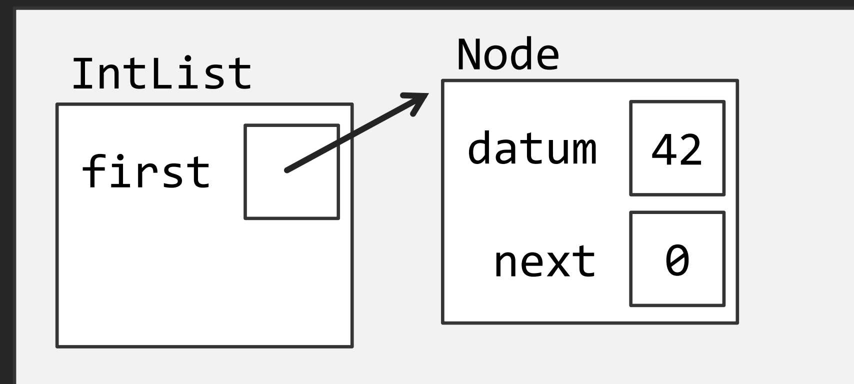
```
class IntList {
private:
  struct Node {
    int datum;
    Node *next;
  };
  Node *first;

public:
  // EFFECTS: returns true if the list is empty
  bool empty() const {
    return first == nullptr;
  }

  ...
};
```



If the list is empty, the `first` pointer will be null.

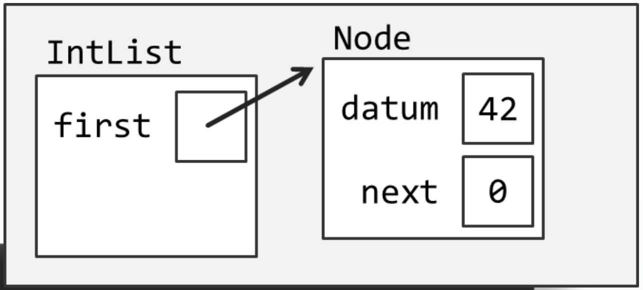
Implementing IntList: front

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

public:
  // REQUIRES: the list is not empty
  // EFFECTS:  Returns (by reference) the first
               element
  int & front() {
    assert(!empty());
    return first->datum;
  }
  ...
};
```



If the list is empty, the first pointer will be null.

Using an `IntList`

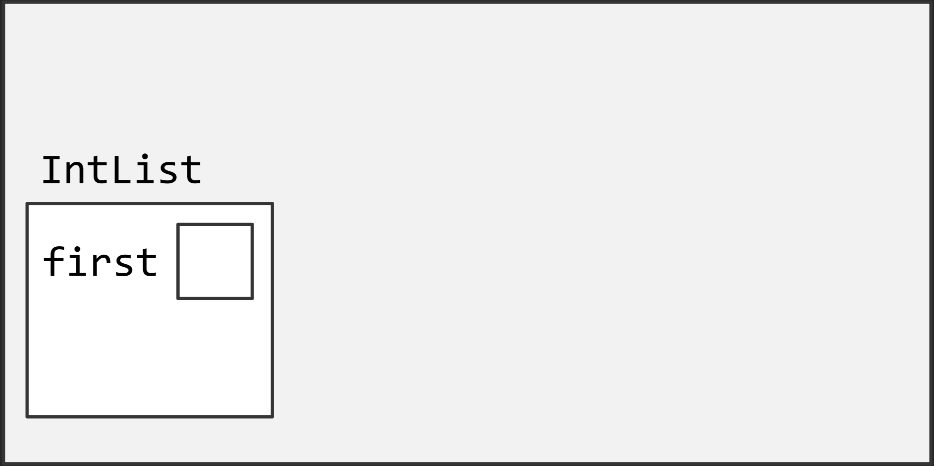



```cpp
int main() {
  IntList list;           // ( )
  list.push_front(1);     // ( 1 )
  list.push_front(2);     // ( 2 1 )
  list.push_front(3);     // ( 3 2 1 )

  cout << list.front(); // 3

  list.front() = 4;       // ( 4 2 1 )

  list.pop_front();       // ( 2 1 )
  list.pop_front();       // ( 1 )
  list.pop_front();       // ( )

  cout << list.empty(); // true (or 1)
}
```

front needs to return an object by reference to support this.

Implementing `IntList`: `push_front`

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

public:
  // EFFECTS: inserts datum at the front of the list
  void push_front(int datum) {
    Node *p = new Node;
    p->datum = datum;
    p->next = first;
    first = p;
  }
  ...
};
```



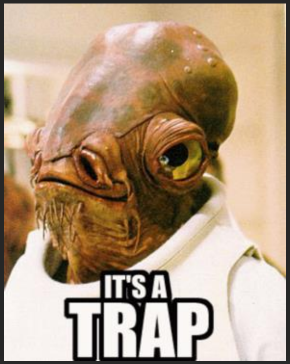
Exercise: pop_front

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

public:
  // REQUIRES: the list is not empty
  // EFFECTS:  removes the first element
  void pop_front() {

    // TODO: YOUR CODE HERE

  }
  ...
};
```

Solution: pop_front

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

public:
  // REQUIRES: the list is not empty
  // EFFECTS:  removes the first element
  void pop_front() {
    assert(!empty());
    delete first;
    first = first->next;
  }
  ...
};
```

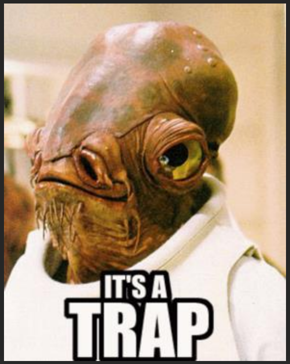
What's wrong with this code?



Solution: pop_front

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

public:
  // REQUIRES: the list is not empty
  // EFFECTS:  removes the first element
  void pop_front() {
    assert(!empty());
    first = first->next;
    delete first;
  }
  ...
};
```

How about this instead?



Solution: pop_front

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

public:
  // REQUIRES: the list is not empty
  // EFFECTS:  removes the first element
  void pop_front() {
    assert(!empty());
    Node *victim = first;
    first = first->next;
    delete victim;
  }
  ...
};
```

Use a temporary variable to keep track of the Node to be destroyed. Now we can safely change first.

Break Time!

We'll start again in five minutes.

Exercise: Traversing a Linked List

- You can use a pointer to traverse a linked list.
  - Start it pointing to the first **`Node`**.
  - Move it to each **`Node`** in turn via **`next`** pointers.
  - At each step, access the **`datum`** of the current **`Node`**.
  - Stop when you get to the null pointer.
- Use this pattern to write a **`print`** function.

```
class IntList {
public:
  // MODIFIES: os
  // EFFECTS:  prints the list to os
  void print(ostream &os) const {
    // TODO: YOUR CODE HERE
  }
  ...
};
```

Solution: Traversing a Linked List

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

public:
  // MODIFIES: os
  // EFFECTS:  prints the list to os
  void print(ostream &os) const {
    for (Node *np = first; np; np = np->next) {
      os << np->datum << " ";
    }
  }
  ...
};
```

Linked Lists and Dynamic Memory

```
void func() {
  IntList list;
  list.push_front(1);
  IntList list2 = list;
  list2.push_front(2);
}
```

- Draw a memory diagram.
  Are there any issues with this code?

Recall: Custom Big Three

- When do we need our own **custom** versions?
  - If you need a deep copy.
  - You need a deep copy if the object owns and manages any resources (e.g. dynamic memory).

- Hints:
  - Check the constructor. If it creates dynamic memory, you probably need the big three.
  - Look at the members. If some of them are pointers, you might need the Big Three.

`IntList` Big Three

- Do we need custom implementations of the Big Three for `IntList`?

- Yes. `IntList` owns and manages `Nodes` dynamically allocated on the heap.

The Big Three

- Destructor
  1. Free resources[1]
- Copy Constructor
  1. Copy regular members from `other`
  2. Deep copy resources from `other`
- Assignment Operator
  1. Check for self-assignment
  2. Free resources
  3. Copy regular members from `rhs`
  4. Deep copy resources from `rhs`
  5. return `*this`

1 The "resource" we often see in 280 is dynamic memory.

The Big Three

How do we avoid code duplication?

- Destructor
  1. Free resources — `pop_all()`
- Copy Constructor
  1. Copy regular members from `other`
  2. Deep copy resources from `other` — `push_all()`
- Assignment Operator
  1. Check for self-assignment
  2. Free resources — `pop_all()`
  3. Copy regular members from `rhs`
  4. Deep copy resources from `rhs` — `push_all()`
  5. return `*this`

pop_all and push_all

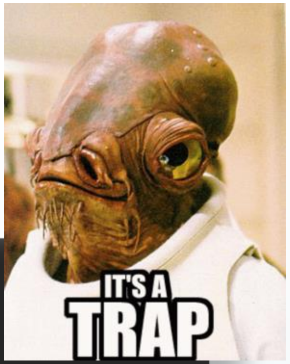
```
class IntList {
...
private:
  ...

  // EFFECTS: removes all nodes from the list
  void pop_all();

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

};
```

Implementing pop_all

```
class IntList {
...
private:
  ...
  // EFFECTS: removes all nodes from the list
  void pop_all() {
    while (!empty()) {
      pop_front();
    }
  }

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

};
```

Implementing push_all

```
class IntList {
...
private:
  ...
  // EFFECTS: removes all nodes from the list
  void pop_all() {
    while (!empty()) {
      pop_front();
    }
  }

  // EFFECTS: copies all nodes from the other list
  //          to this list
  void push_all(const IntList &other) {
    for (Node *np = other.first; np; np = np->next) {
      push_front(np->datum);
    }
  }
};
```



What's wrong with this code?

Implementing push_all

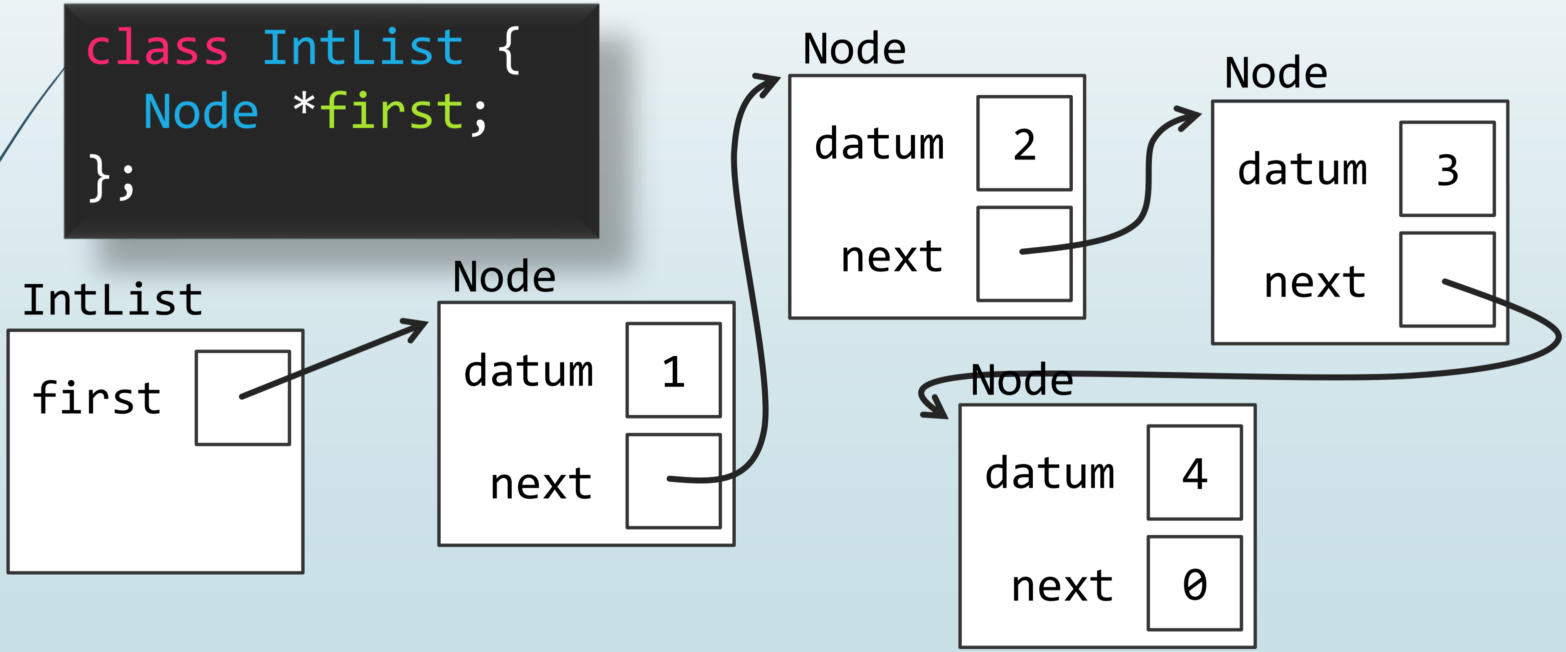
```
class IntList {
...
private:
  ...
  // EFFECTS: removes all nodes from the list
  void pop_all() {
    while (!empty()) {
      pop_front();
    }
  }

  // EFFECTS: copies all nodes from the other list
  //          to this list
  void push_all(const IntList &other) {
    for (Node *np = other.first; np; np = np->next) {
      push_back(np->datum);
    }
  }
};
```

To avoid a backwards copy, we could use a push_back function.

Implementing push_back

- What if we wanted to insert at the end of the list?
  - We have to traverse all the way from the front!
  - Instead, let's change the data representation…

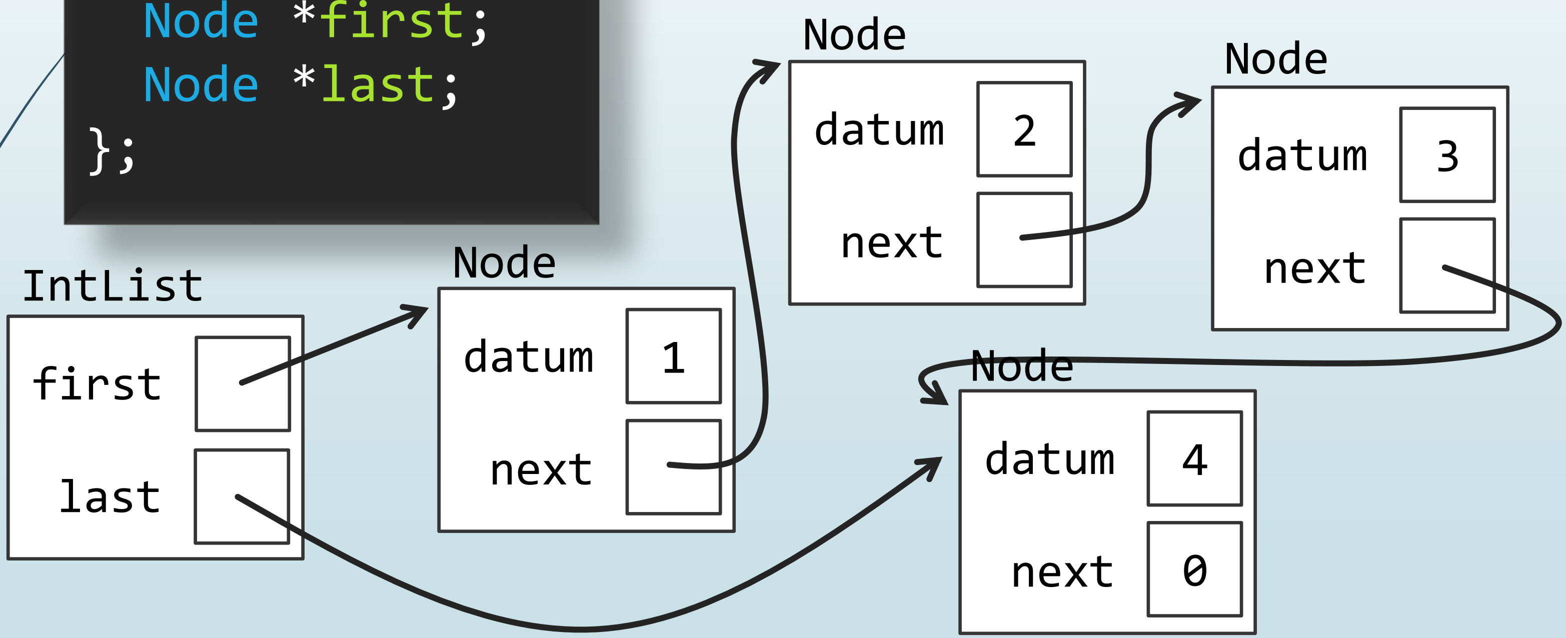
```
class IntList {
  Node *first;
};
```



Implementing push_back

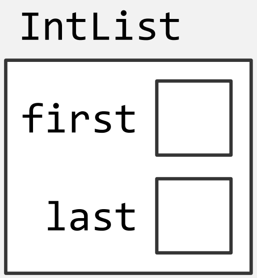
- What if we wanted to insert at the end of the list?
  - We have to traverse all the way from the front!
  - Instead, let's change the data representation…

```
class IntList {
  Node *first;
  Node *last;
};
```

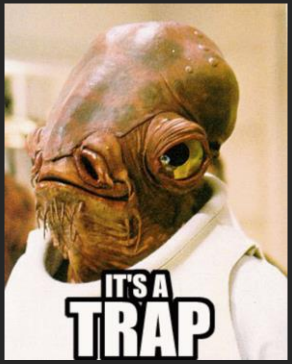


Implementing IntList: push_back

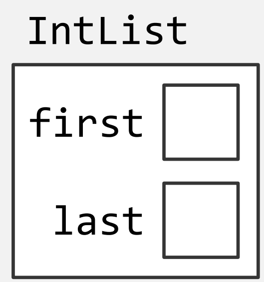
```
class IntList {
private:
  struct Node {
    int datum;
    Node *next;
  };
  Node *first;
  Node *last;
public:
  // EFFECTS: inserts datum at the back of the list
  void push_back(int datum) {
    Node *p = new Node;
    p->datum = datum;
    p->next = nullptr;
    last->next = p;
    last = p;
  }
  ...
};
```



What's wrong with this code?



Implementing IntList: push_back




```
class IntList {
private:
  struct Node {
    int datum;
    Node *next;
  };
  Node *first;
  Node *last;
public:
  // EFFECTS: inserts datum at the back of the list
  void push_back(int datum) {
    Node *p = new Node;
    p->datum = datum;
    p->next = nullptr;
    if (empty()) { first = last = p; }
    else {
      last->next = p;
      last = p;
    }
  }
};
```

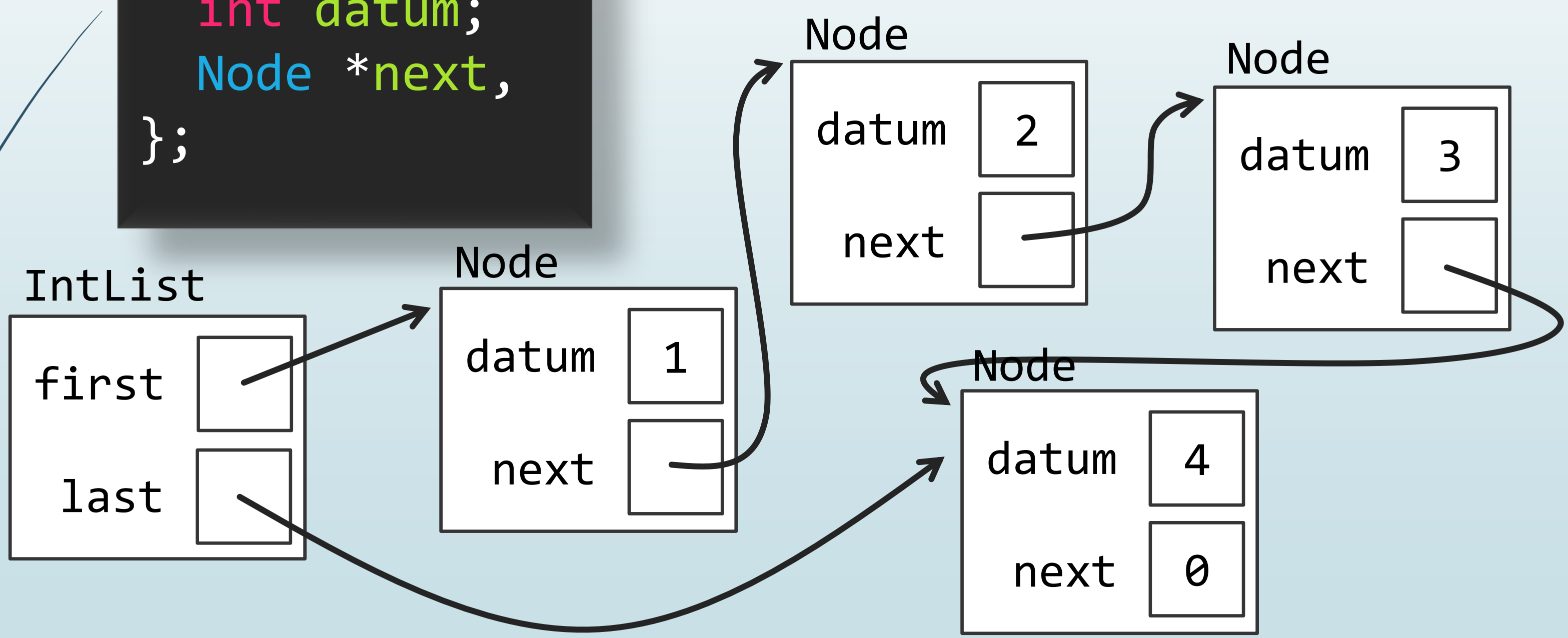
IntList Big Three

```
class IntList {
public:
  ...
  ~IntList() {
    pop_all();
  }

  IntList(const IntList &other)
    : first(nullptr), last(nullptr) {
    push_all(other);
  }

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

Implementing pop_back

- What if we want to remove from the end of the list?
  - We have to traverse all the way from the front!
  - Instead, let's change the data representation…

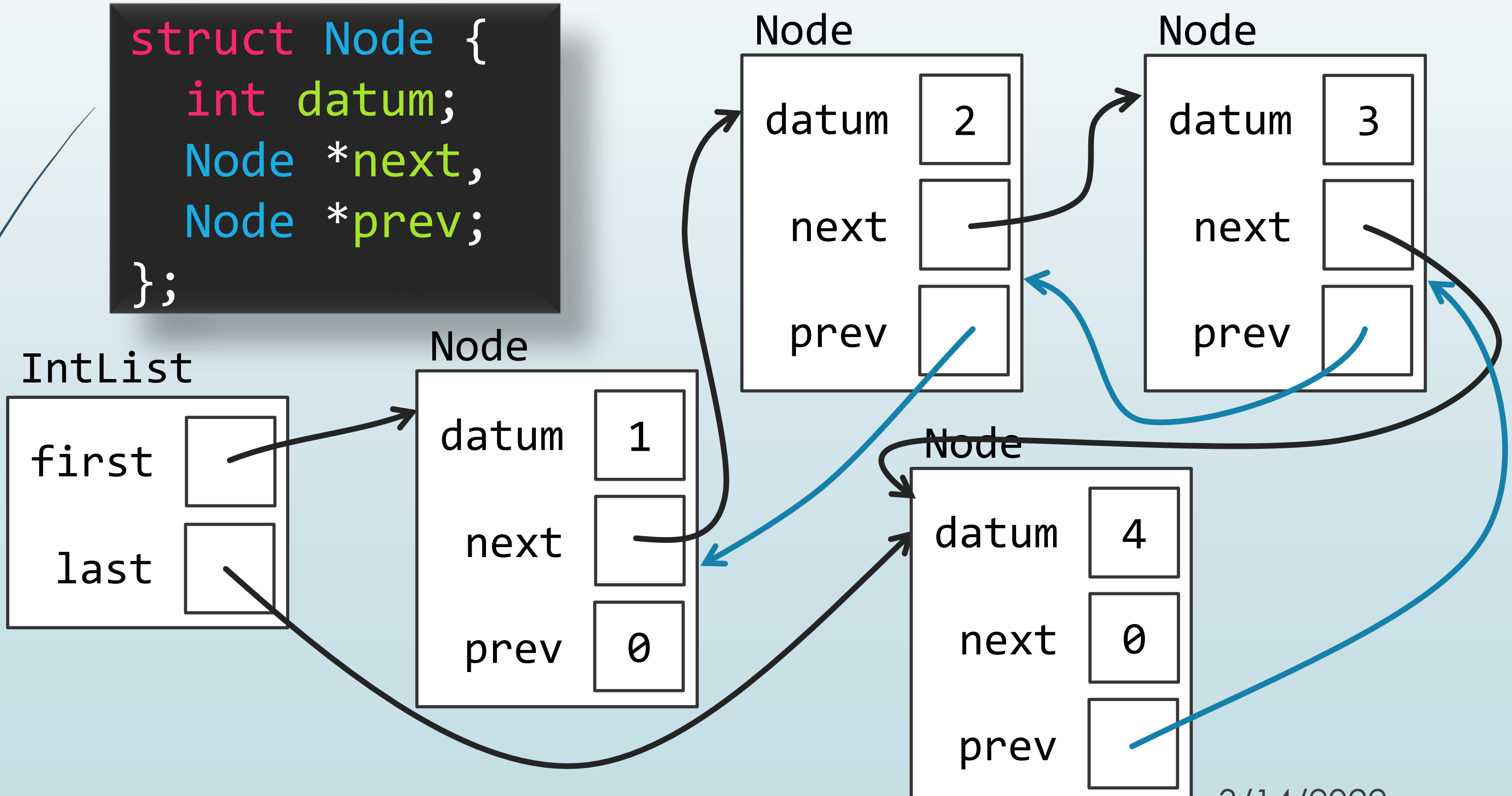
```
struct Node {
  int datum;
  Node *next,
};
```



Implementing pop_back

- What if we want to remove from the end of the list?
  - We have to traverse all the way from the front!
  - Instead, let's change the data representation…

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



Linked List Template

List.h

```
template <typename T>
class List {
public:
  void push_front(T v);
  T& front();
private:
  struct Node {
    T datum;
    Node *next;
  };
  Node *first;
};
```

```
#include "List.h"
int main() {
  List<int> list1;
  List<Duck> list2;
}
```

The compiler instantiates the template as needed according to how it is used in the code.

```
class List<int> {
public:
  void push_front(int v);
  int& front();
private:
  struct Node {
    int datum;
    Node *next;
  };
  Node *first;
};
```

```
class List<Duck> {
public:
  void push_front(Duck v);
  Duck& front();
private:
  struct Node {
    Duck datum;
    Node *next;
  };
  Node *first;
};
```

Member functions of a template must be defined or #included in the .h file, though they need not be defined directly in the class template.