Ve 280

Programming and Introductory Data Structures

Dynamic Resizing;
Linked List

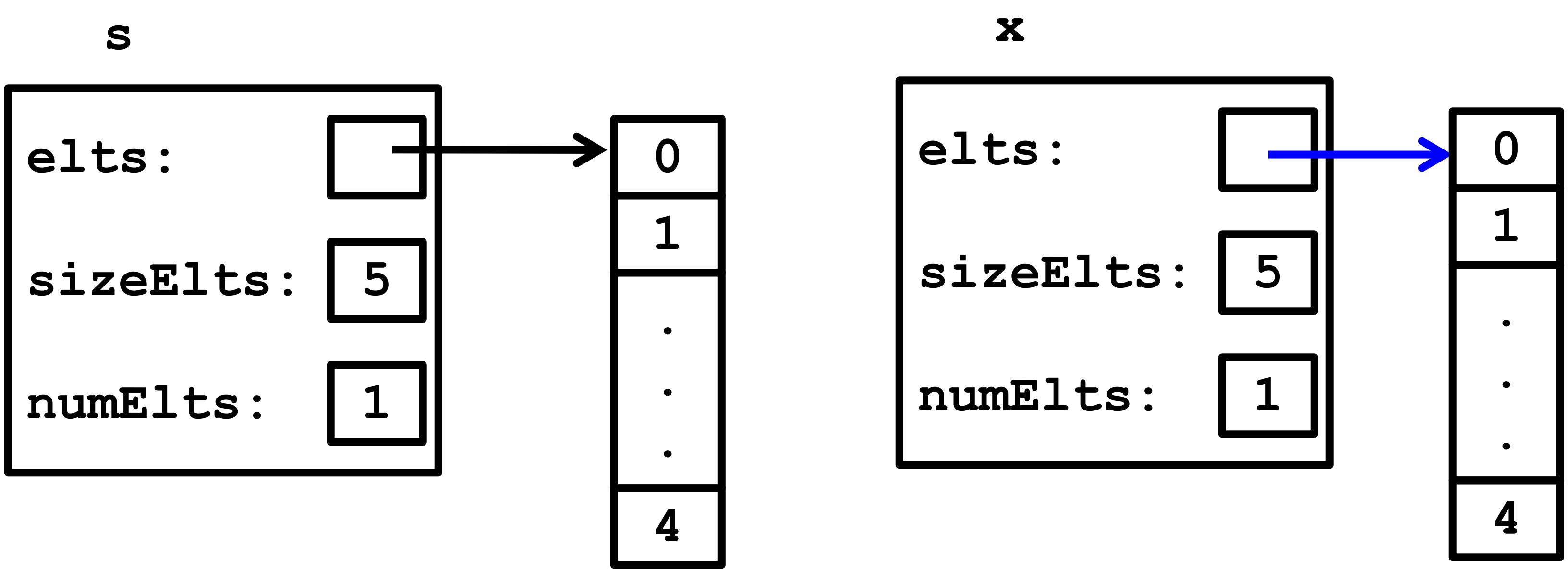
Outline

- Assignment Operator
- Dynamic Resizing
- Introduction to Linked List
- Implementation of Linked List

Review

- Shallow Copy versus Deep Copy
  - We need to copy the dynamic array, not just the array pointer.
- Copy Constructor: When passing arguments by value to a function, copy constructor is called.

```
IntSet(const IntSet &is);
```



Assignment Operators

Basics

- Assignment statement returns a value.
- The value is the **reference** to its left-hand-side object.

- Example

```
x = 4;
(y = x) += 2;
```

  - Are the above statements legal?
  - What is the value of `y`?

Assignment Operators

Basics

- Assignment statements can be "chained". The following is legal in C++:

```
x = y = z;
```

- This is a compound expression. Assignment operators binds **right-to-left**.
- Because "=" binds right-to-left, we first assign `z` to `y`, and this expression yields the (new) value "`y`" so that it can in turn be assigned to `x`.

Assignment Operators

On to overloading

- Now, how do we handle the following code?

```
IntSet s1(5);
IntSet s2(10);
s1 = s2; // assignment of s2 to s1
```

- By default, the compiler will use a shallow copy for the this.
- However, like a copy constructor, assignment must do a **deep copy** of the right-hand-side to the left-hand-side.
- To implement this, we **redefine** the "assignment operator" for `IntSets` by doing **operator overloading**.

Assignment Operators

Operator overloading

- Here's how we **overload** the assignment operator:

```
class IntSet {
   // data elements
   ...
   public:
   // Constructors
   ...
   IntSet &operator= (const IntSet &is);
   ...
};
```

You can overload other operators such as +, *, etc. You need to use the keyword **`operator`**

Assignment Operators

Operator overloading

```
IntSet &operator= (const IntSet &is);
```

- Like the copy constructor, the assignment operator takes a **reference to a const** instance to copy from.
- However, it also **returns** a **reference** to the copied-to object.

- When we call the assignment operator

  `a = b;`

- Essentially, we call the assignment operator of object `a`.
- `b` is the argument to the `operator=()` function.
  - Consider this as `a.operator=(b)`

Assignment Operators

Operator overloading

- The cool thing is that we have written `copyFrom` already:

```
void IntSet::copyFrom(const IntSet &is) {
  if (is.sizeElts != sizeElts) { // Resize array
    delete[] elts;
    sizeElts = is.sizeElts;
    elts = new int[sizeElts];
  }
  // Copy array
  for (int i = 0; i < is.sizeElts; i++) {
    elts[i] = is.elts[i];
  }
  // Establish numElts invariant
  numElts = is.numElts;
}
```

Assignment Operators

Operator overloading

- With `copyFrom`, the assignment operator is (almost) trivial:

```
IntSet &IntSet::operator= (const IntSet &is) {
   copyFrom(is);
   return *this;
}
```

<u>Note</u>: Every method has an implicit local variable "this", which is a pointer to the current instance on which that method operates.

Assignment Operators

Operator overloading

- With `copyFrom`, the assignment operator is (almost) trivial:

```
IntSet &IntSet::operator= (const IntSet &is) {
   copyFrom(is);
   return *this;
}
```

<u>Note</u>: This line dereferences that pointer and then returns a reference to it. We can't just return "this", because "this" is just a pointer, cannot be used as a reference.

Assignment Operators

Operator overloading

- With `copyFrom`, the assignment operator is (almost) trivial:

```
IntSet &IntSet::operator= (const IntSet &is) {
   copyFrom(is);
   return *this;
}
```

<u>**Note**</u>: We must return the reference to the **assigned-to** object, not the **assigned-from** object, i.e., we cannot `return is`.

Assignment Operators

Question

- **<u>Question</u>**: What happens if we do this?

```
IntSet s(50);
s = s;
```

- It is fine! Since their `sizeElts` are equal, no destroying and reallocating are needed.
- However, it is better to modify the code as follows:

```
IntSet &IntSet::operator= (const IntSet &is)
{
    if(this != &is)
        copyFrom(is);
    return *this;
}
```

The Rule of the Big Three

- What we have talked so far can be summarized with a simple rule: **the Rule of the Big Three**.

- Specifically, if you have any **dynamically allocated storage** in a class, you must provide:
  - **A destructor**
  - **A copy constructor**
  - **An assignment operator**

- If you find yourself writing one of these, you almost certainly need all of them.

Outline

- Assignment Operator
- Dynamic Resizing
- Introduction to Linked List
- Implementation of Linked List

Dynamic Resizing

Modifying `Insert()`

- We have modified `IntSet` to allow a client to specify the **capacity** of an `IntSet`.
- However, this doesn't really get around the "big instance" problem, since the caller itself might not know how big the set will grow.
- So, what we **really** want to do is to create an `IntSet` that can **grow** as big as it needs to.
- To do this, we only need to modify the `insert()` method.

Dynamic Resizing

Modifying `Insert()`

- We will use the unsorted representation. We will focus on the action of **resizing**, not the action of inserting.

```
void IntSet::insert(int v) {
  if (indexOf(v) == sizeElts) {
    if (numElts == sizeElts)
      throw sizeElts;
    elts[numElts++] = v;
  }
}
```

We want to modify `throw sizeElts`

Dynamic Resizing

Modifying `Insert()`

- Rather than throw an exception if the array is at maximum capacity, we will instead **grow** the array.

```
void IntSet::insert(int v) {
  if (indexOf(v) == sizeElts) {
    if (numElts == sizeElts)
      grow();
    elts[numElts++] = v;
  }
}
```

Dynamic Resizing

Modifying `Insert()`

- The `grow` method won't take any arguments or return any values.
- It should **never** be called from outside of the class, so add it as a **private** method taking no arguments and returning void.

```
class IntSet {
  // data members ...
  void grow();
  // EFFECTS: enlarge the elts array,
  //          preserving current contents
public:
  // ...
};
```

Dynamic Resizing

Modifying `Insert()`

- `grow` will look like the assignment operator.

- It must perform the following steps:
  1. Allocate a bigger array.
  2. Copy the smaller array to the bigger one.
  3. Destroy the smaller array.
  4. Modify elts/sizeElts to reflect the new array.

Note the order of allocation can destroy. Can we switch this order?

Dynamic Resizing

Modifying `Insert()`

```
void grow() {
   int *tmp = new int[sizeElts + 1];
   for (int i = 0; i < numElts; i++) {
      tmp[i] = elts[i];
   }
   delete [] elts;
   elts = tmp;
   sizeElts += 1;
}
```

1. Allocate a bigger array.
2. Copy the smaller array to the bigger one.
3. Destroy the smaller array.
4. Modify elts/sizeElts to reflect the new array.

Dynamic Resizing

Group Exercise – Modifying `Insert()`

- Unfortunately, we might end up doing a lot of copying.
- Suppose a client creates an `IntSet` of capacity 1, and then inserts N elements into it.
- **<u>Question</u>**: What's the number of integer copies performed by the function `grow` in the worst case?

```
void grow() {
   int *tmp = new int[sizeElts + 1];
   for (int i = 0; i < numElts; i++) {
      tmp[i] = elts[i];
   }
   delete [] elts;
   elts = tmp;
   sizeElts += 1;
}
```

```
void IntSet::insert(int v) {
   if (indexOf(v) == sizeElts) {
      if (numElts == sizeElts)
         grow();
      elts[numElts++] = v;
   }
}
```

Dynamic Resizing

Group Exercise – Modifying `Insert()`

- Suppose a client creates an `IntSet` of capacity 1, and then inserts N elements into it. What's the number of integer copies in the worst case?

__**Answer**__:

- The worst case happens when all the elements inserted are different!
- Before each new insertion, `numElts == sizeElts`.
- We need to call grow each time we insert a new element.
- When we grow an array of size k to one of size k+1, we copied k items.
- We did this for k from 1 to N-1.
- So the total number of copies is:

$$1 + 2 + \ldots + (N-2) + (N-1) = N(N-1)/2$$

Dynamic Resizing

Group Exercise – Modifying `Insert()`

- Suppose a client creates an `IntSet` of capacity 1, and then inserts N elements into it. What's the number of integer copies in the worst case?

<u>Answer</u>:

- N(N-1)/2
- This is a quadratic function in N.
- This means that as the `IntSet` grows, the cost to build the `IntSet` grows much faster.

- **How can we make this better?**

Dynamic Resizing

Optimizing `grow()`

- **How can we make `grow()` better?**

- The intuition is that we aren't buying enough room each time we copy the array:
  - We copy N things, but only buy room for one more slot.

- Instead, we'd like to buy more slots for each N things we copy.
- The new version is only **slightly** different from the old version.
- However, it has **very** different performance characteristics.

Dynamic Resizing

Optimizing `grow()`

```
void grow() {
  int *tmp = new int[sizeElts * 2];
  for (int i = 0; i < numElts; i++) {
    tmp[i] = elts[i];
  }
  delete [] elts;
  elts = tmp;
  sizeElts *= 2;
}
```

Instead of growing the array by one, we double it.

Dynamic Resizing

Group Exercise – Optimizing `grow()`

- Suppose a client creates an `IntSet` of capacity 1, and then inserts N elements into it using the new version of `grow()`.
- **<u>Question</u>**: What's the number of integer copies performed by the function `grow` in the worst case?

```
void grow() {
  int *tmp = new int[sizeElts * 2];
  for (int i = 0; i < numElts; i++) {
    tmp[i] = elts[i];
  }
  delete [] elts;
  elts = tmp;
  sizeElts *= 2;
}
```

```
void IntSet::insert(int v) {
  if (indexOf(v) == sizeElts) {
    if (numElts == sizeElts)
      grow();
    elts[numElts++] = v;
  }
}
```

Dynamic Resizing

Group Exercise – Optimizing `grow()`

Answer:

- After the first `grow`, the capacity is `2`. We copy `1` item.
- After the second `grow`, the capacity is `4`. We copy `2` items.
- After the k-th `grow`, the capacity is 2^k. We copy 2^{k-1} items.

- Suppose $2^m < N \leq 2^{m+1}$
- How many times we need to call grow?
  - m+1 times
- How many copies we perform?
  - $T = 1+2+4+ \ldots +2^m = 2^{m+1} - 1 < 2N$

Dynamic Resizing

Group Exercise – Optimizing `grow()`

Answer:

- `T` (the number of copies) < 2N
- So, instead of copying almost (N-1)N/2 elements, we copy fewer than 2N of them.

Dynamic Resizing

Group Exercise – Optimizing `grow()`

Answer:

- Here's a little table showing what this means:

# elements	(N-1)N/2	2N
1	0	2
8	28	16
64	2016	128
512	130816	1024
2048	2096128	4096

- The "double" implementation is **much** better than the "by-one" implementation.

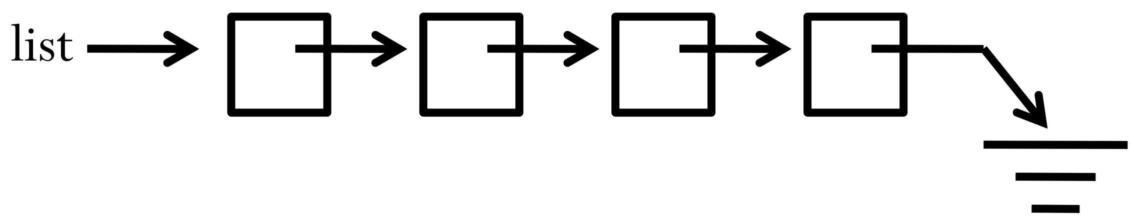
Outline

- Assignment Operator
- Dynamic Resizing
- Introduction to Linked List
- Implementation of Linked List

Linked Lists

Introduction

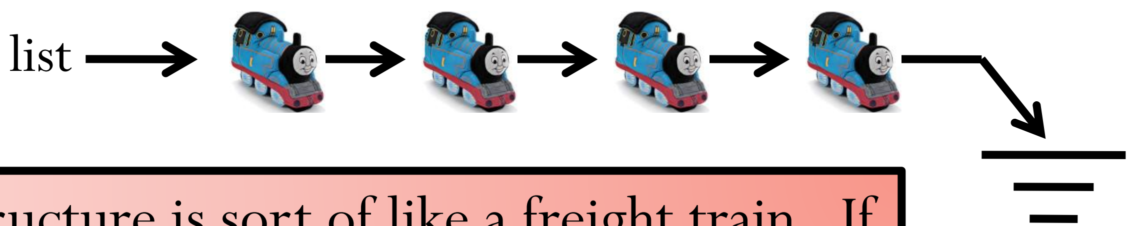
- Expandable arrays are only one way to implement storage that can grow and shrink over time.
- Another way is to use a **linked structure**.
- A linked structure is one with a series of zero or more data containers, connected by pointers from one to another, like:



Linked Lists

Introduction

- Expandable arrays are only one way to implement storage that can grow and shrink over time.
- Another way is to use a **linked structure**.
- A linked structure is one with a series of zero or more data containers, connected by pointers from one to another, like:



A linked structure is sort of like a freight train. If you need to carry more freight, you get a new boxcar, connect it to the train, and fill it. When you don't need it any more, you can remove that boxcar from the train.

Linked Lists

Introduction

- Suppose we wanted to implement an abstract data type for a mutable list of integers, represented as a linked structure.

- This ADT will be similar to the `list_t` type from project two, except that `list_t` is **immutable**:
  - Once a `list_t` object was created, no operations on that list would ever change it.

Linked Lists

Introduction

- There are three operations that the list must support:

```
bool isEmpty();
  // EFFECTS: returns true if list is empty,
  //          false otherwise

void insert(int v);
  // MODIFIES: this
  // EFFECTS: inserts v into the front of the list

class listIsEmpty {}; // An exception class
int remove();
  // MODIFIES: this
  // EFFECTS: if list is empty, throw listIsEmpty.
  //          Otherwise, remove and return the first
  //          element of the list
```

Linked Lists

Introduction

- For example, if the list is (1 2 3), and you `remove()`, the list will be changed to (2 3), and `remove` returns 1.

```
int remove();
   // MODIFIES: this
   // EFFECTS: if list is empty, throw listIsEmpty.
   //          Otherwise, remove and return the
   //          first element of the list
```

- If you then `insert(4)`, the list changes to (4 2 3).

```
void insert(int v);
   // MODIFIES: this
   // EFFECTS: inserts v into the front of the list
```

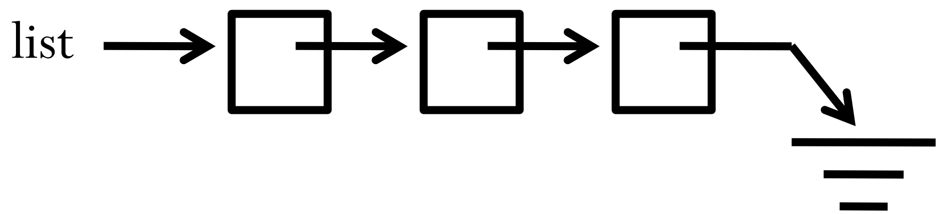
Outline

- Assignment Operator
- Dynamic Resizing
- Introduction to Linked List
- Implementation of Linked List

Linked Lists

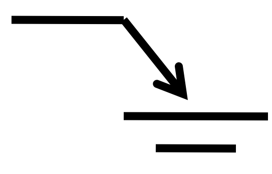
Implementation

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




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

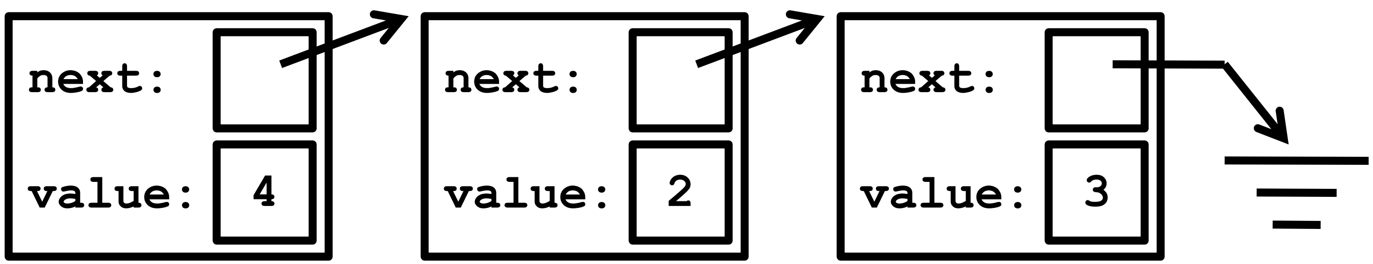
- The invariants on these fields are:
  - The **value** field holds the integer value of this element of the list.
  - The "**next**" field points to the next node in the list, or NULL if the node is the last one in the list.
- NULL means "pointing at nothing". Its value is "0", written as:



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:

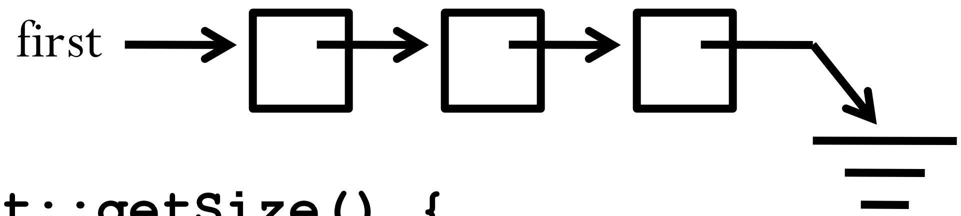
```
class IntList {
  node *first;
 public:
   ...
};
```

```
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:

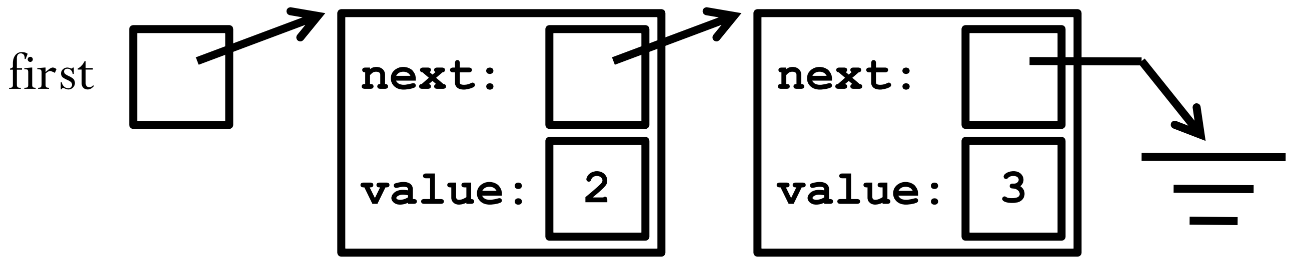
```
void IntList::insert(int v) {
    node *np = new node;
    np->value = v;
    np->next = first;
    first = np;
}
```

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

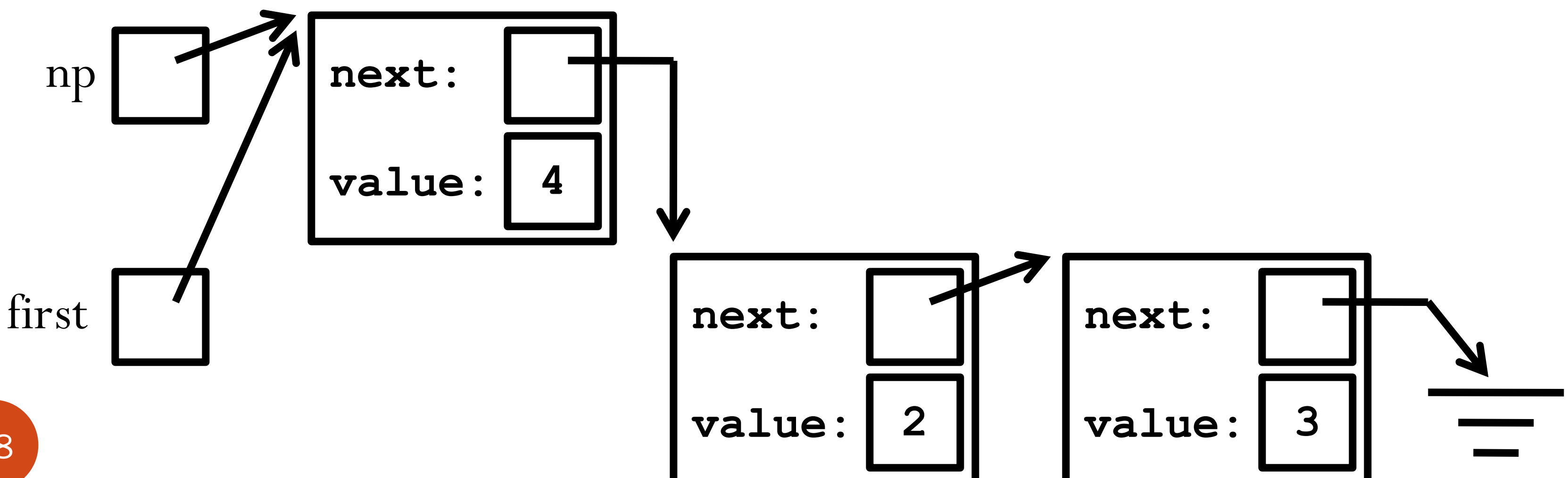
Linked Lists

Example

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



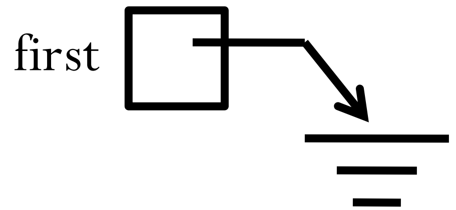
- And then the new list is



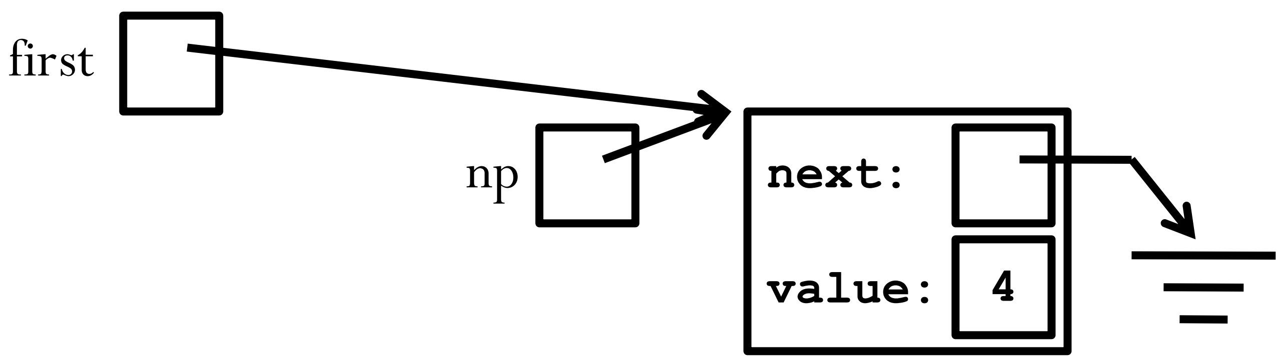
Linked Lists

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`.
- **Question**:

  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;
}
```

Reference

- **Problem Solving with C++ (8th Edition)**, by *Walter Savitch*, Addison Wesley Publishing (2011)
  - Chapter 11.4 Classes and Dynamic Arrays
  - Chapter 13.1 Nodes and Linked Lists