Ve 280

Programming and Introductory Data Structures

Linked List

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.

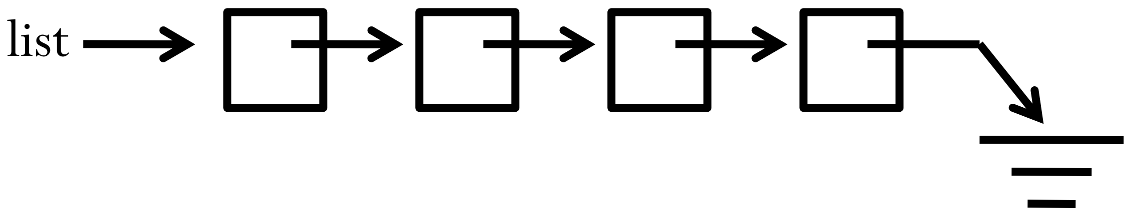
Outline

- Introduction to Linked List
- Implementation of Linked List

Linked Lists

Introduction

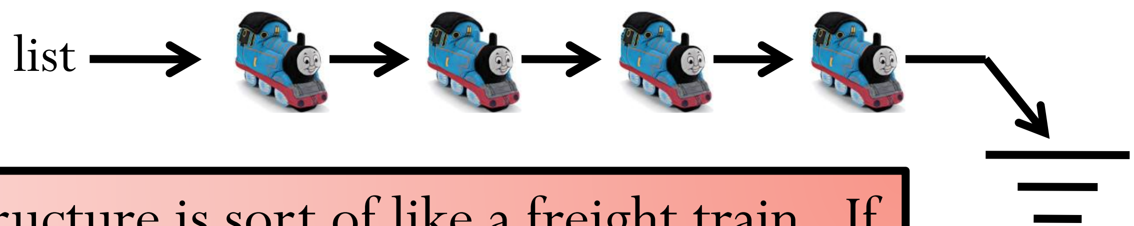
- Expandable arrays are only one way to implement storage that can grow and shrink over time.
- Another way is to use a **linked structure**.
- A linked structure is one with a series of zero or more data containers, connected by pointers from one to another, like:



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

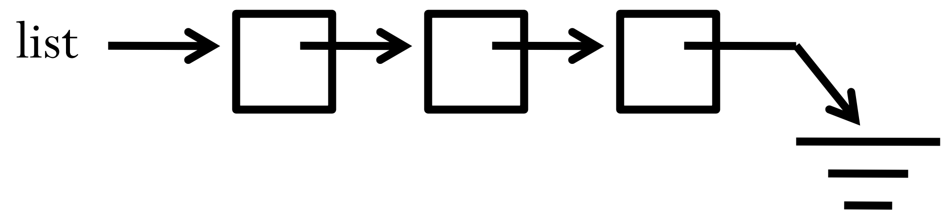
- Introduction to Linked List
- Implementation of Linked List

Linked Lists

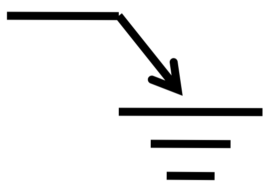
Implementation

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

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



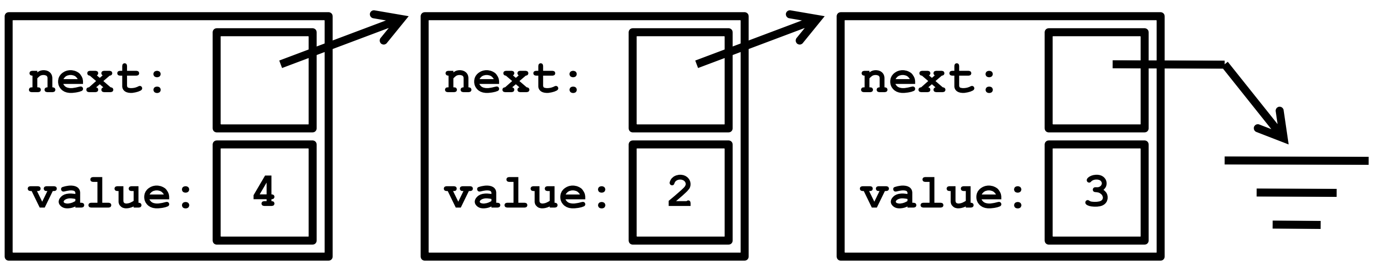
- The invariants on these fields are:
  - The **value** field holds the integer value of this element of the list.
  - The **next** field points to the next node in the list, or NULL if the node is the last one in the list.
- NULL means "pointing at nothing". Its value is "0", written as:



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:

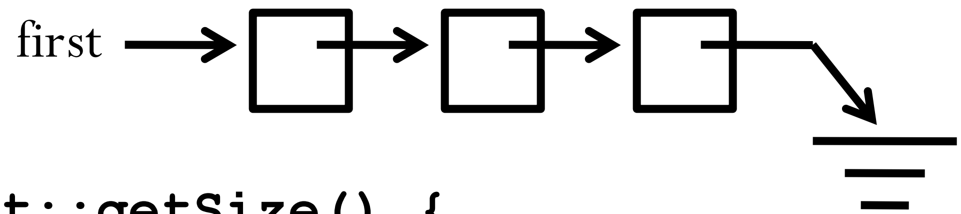
```
class IntList {
   node *first;
 public:
   ...
};
```

```
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:

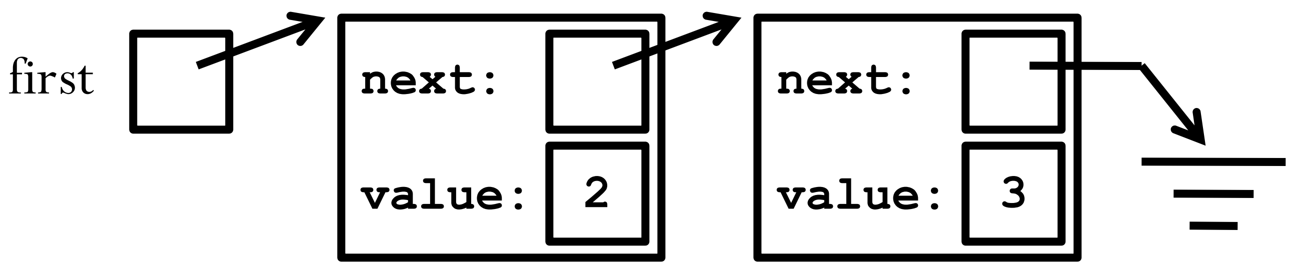
```
void IntList::insert(int v) {
  node *np = new node;
  np->value = v;
  np->next = first;
  first = np;
}
```

Notice that this works no matter what the current list is, as long as the invariant holds (see next slides).

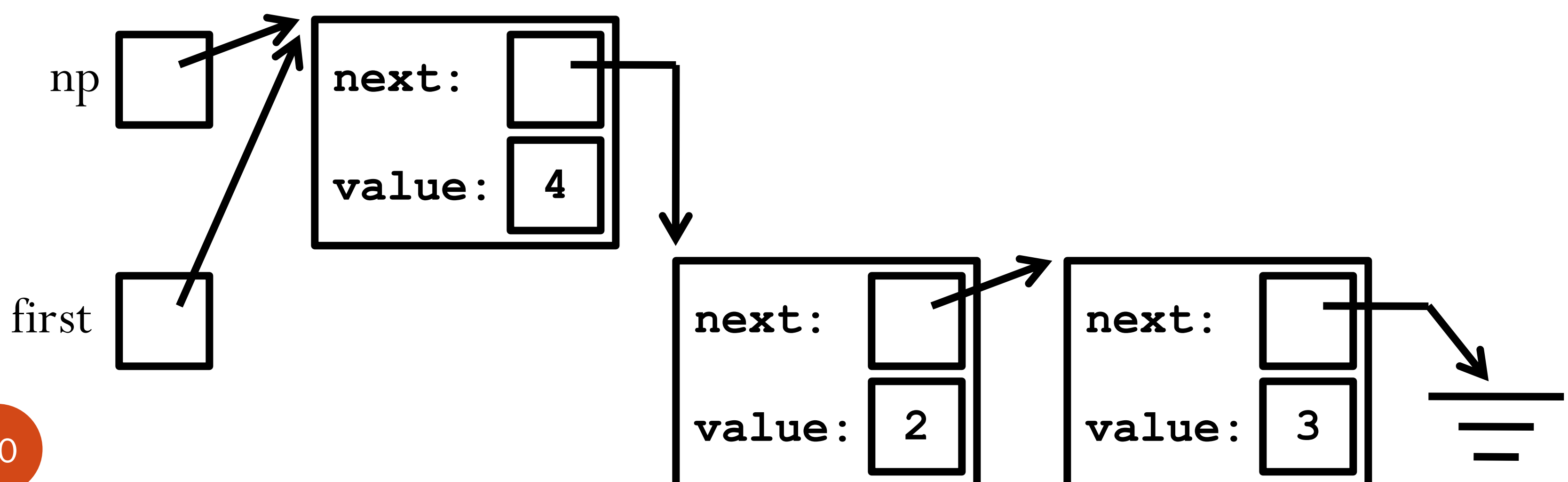
Linked Lists

Example

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



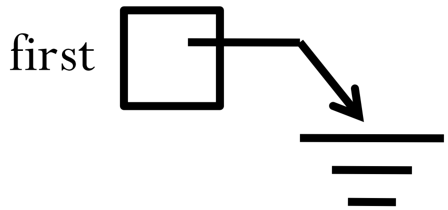
- And then the new list is



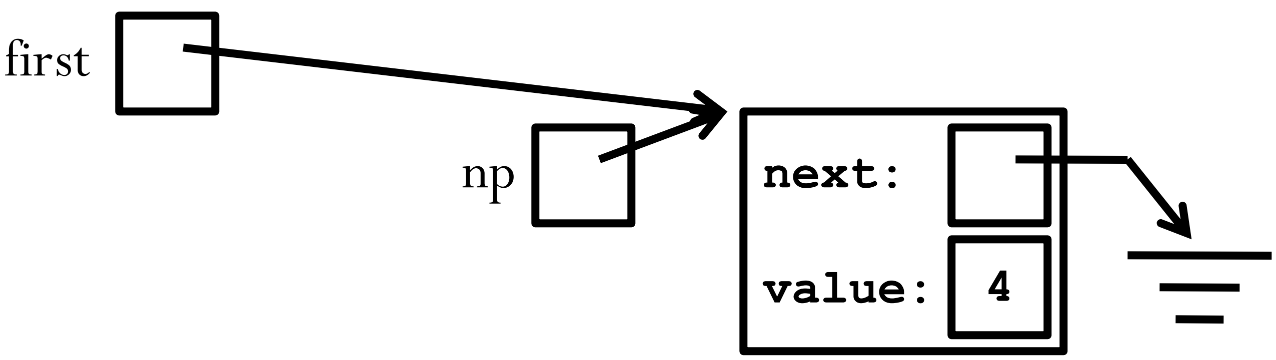
Linked Lists

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:

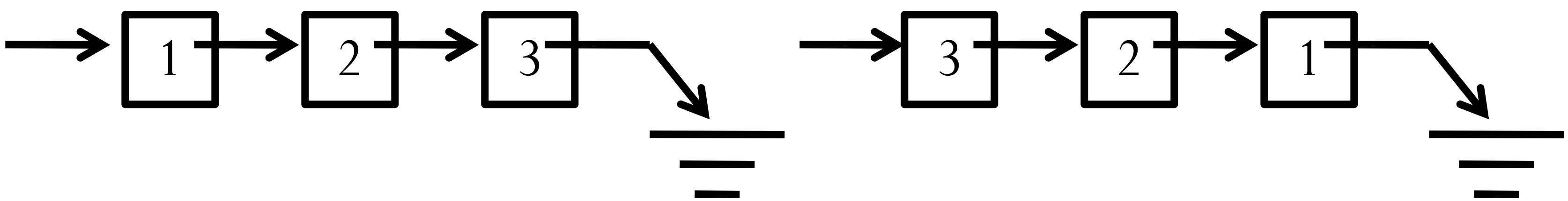
```
void IntList::removeAll() {
   while (!isEmpty()) {
      remove();
   }
}

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);
}
```

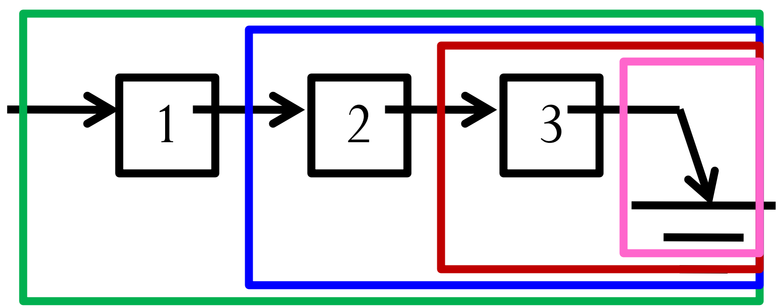
Linked Lists

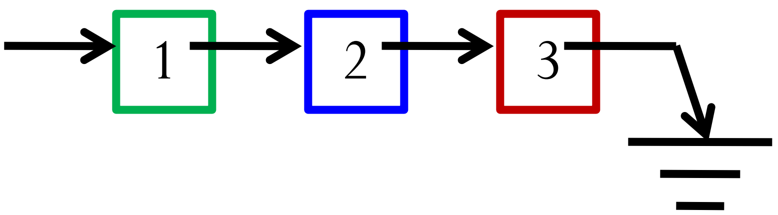
Implementation

```
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;
}
```

Reference

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