Lists
Computer Science & Engineering 156
Christopher M. Bourke
cbourke@cse.unl.edu

Introduction

- Rare to deal with only one instance
- Often need to deal with collections of objects
- Cannot always anticipate how many things we might need to deal with
- Amount may only be available at runtime
- Languages usually provide support for arrays

In Java:
```java
int n = ...;
Integer ids[] = new Integer[n];
```

Problems With Arrays

- Once created, capacity is fixed
- Cannot dynamically change its size (without rebuilding/copying)
- May not need entire capacity at certain points
- Modifications result in non-contiguous elements
- Must manually handle organization
- Must have a priori knowledge of the size before you create it

Better Solution: A List I

Better Solution: A List ADT
Core Functionality:
- Adding elements
- Retrieving elements
- Removing elements
- Automatic resizing

Better Solution: A List II

Other functionality:
- Determining size and/or emptiness
- Iterating over the elements
- Batch operations (add/delete multiple items)

First Solution: Array Based List for Integers I

- Dynamically increase the size by some increment as needed
- Expose only that which needs to be exposed to the outside
- See MyIntegerArrayList
First Solution: Array Based List for Integers II

Possible additional items:

- boolean isEmpty() convenience method
- Downsizing of array when it gets too “small”
- Generalizing the insert method to any index
- Convenience methods (add/delete at end, start)
- Batch operations
- Tuning of resize parameter

Generalized Array Based List I

- Big Problem: MyIntegerArrayList only holds Integer types
- Following this pattern: a new class for every type!
- Need a way to generalize this class
- Write once and reuse
- Perfect use of (parameterized) polymorphism!
- Java generics

Generalized Array Based List II

- See MyArrayList<T>
- All type references now use T
- Cannot instantiate a generic array:
  arr = new T[n];
- Forced to use a raw type and cast:
  this.arr = (T[])new Object[n];
- Okay in this instance as the interface protects this.arr from invalid types
- Encapsulation in practice!

Linked Lists

Problem: resizing is expensive

- New memory allocation
- Have to copy every element
- Expense: linearly related to number of elements in the list
- Potential for wasted space (or additional expense to downsize)
- Even with larger SIZE steps, solution does not scale

Solution: a linked list

Linked Lists

Key aspects:

- List has a reference to one element (the head)
- Each element has a reference to the next element
- No fixed size, no resizing
- Elements in a linked list are realized as Nodes
- Operations involve only shuffling references around
- No random access
- Need to know where the end (or tail) of the list is
- Need to handle “corner” cases specially

A Linked List

Example

```
head
11 — 42 — 35 — ... — 76 — null
```

Figure: A Linked List
Operations

Find

Find operation:

▶ Find element based on an index
▶ Find (first instance of) a specific element
▶ Find multiple elements

Algorithm:

▶ Start at the head
▶ Traverse the list using each next element
▶ Until find condition is satisfied

Delete

Delete operation

▶ Find the element item
▶ Keep track of the previous element prev (element such that prev.next = item)
▶ Make prev.next point to item.next
▶ Delete item

Add Operation

Add operation:

▶ Find place to insert item
▶ Say between a, b
▶ Set:
  a.next = item
  item.next = b
▶ Inserting at the head or end: may need to treat as special cases

Delete Operation

Example

11 ➤ 42 ➤ 35 ➤ ...

Figure: Deleting element 42

Full Implementation Example I

▶ Full Java implementation: see MyLinkedList
▶ Makes use of generics
▶ Uses a Node container class
▶ Implements the Iterable interface
▶ Supports only two simple operations:
  addElementAtHead(T item)
  removeFirstInstanceOf(T item)

Full Implementation Example II

Exercises: add additional functionality:

▶ Implement a size() method
▶ Insert at an index
▶ Insert at end
▶ Find method (index based)
▶ Batch methods
▶ Add functionality to keep track of a tail element to optimize insert-at-end method
Circularly Linked List

- A linked list is circular if the "last" element points to the first
- Potentially simplifies some code
- Instead of head we could simply have a current node
- Amortized optimization: don’t always need to start at the beginning of the list
- Able to model round robin data structures

Example

```
head
11 42 35 76
```

Figure: A Circularly Linked List

Doubly Linked List

- Nodes can be modified to have a previous as well as a next pointer
- Traversal can be reversed
- Simpler traversal (no need to keep track of previous element)
- Additional amortized optimizations

Example

```
head
null 11 42 35 76 null
```

Figure: A Doubly Linked List

Java Collections List I

Interface

Java provides a basic List interface

- [http://download.oracle.com/javase/6/docs/api/java/util/List.html](http://download.oracle.com/javase/6/docs/api/java/util/List.html)
- Interface, so you don’t use it directly
- Parameterized for your protection
- Sub-interface of Collection<E>
Java Collections List II

Interface

- Insert methods:
  - add(E e)
  - add(int index, E e)
  - addAll(Collection<? extends E> c)
- Check method:
  - contains(Object o)
- Retrieve method:
  - get(int index)
- All collections implement Iterable<E>

```java
for(ElementType element : listName){ ... }
```

Java Collections List III

Interface

- Remove methods:
  - remove(Object o)
  - remove(int index)
- Replace method:
  - set(int index, E e)
- Convenience methods:
  - isEmpty()
  - indexOf(Object o)
  - size()

Java Collections List I

Implementations

ArrayList
- Basic array-based implementation of List
- Allows (duplicate) null values
- Amortized constant time for inserts
- Cost is linear for other operations

LinkedList
- Linear cost for all operations
- Allows (duplicate) null values

Vector
- Provides some thread-safe behavior
- Best-effort fail-fast of iterators
- Synchronized add, get, size, etc, methods
- Still not fully thread safe
- Example:

```java
if(v.isEmpty())
v.add(a);
```

Java Collections: Sets

- Java also has a Set interface
- Duplicate values are not allowed
- At most one null value allowed (depending on implementation)
- Duplicity determined by equals() (and maybe hashCode()) methods

Java Collections: Best Practices

Best Practice:
- Always implement (@Override) the equals() and hashCode() so that collections will work properly
- Should be based on the entire state of the object (standard idioms)
- Code Demo: UserClassA versus UserClassB
- Eclipse tip: Source → Generate hashCode() and equals()
- Take care with mutable elements!