DIT181: Data Structures and Algorithms

The Collections API & Linked Lists

Gül Calikli
Email: calikli@chalmers.se

Recommended Reading

- The collections API
  - Iterator pattern and basic iterator design (pp. 226-230)
  - Iterator interface (pp. 232-238)
  - The list interface (pp. 244-254)
  - Stacks and queues (pp. 254-257)
- Linked List Implementations of Stacks and queues
  - Stacks (pp. 578-581)
  - Queues (pp. 581-584)
  - Comparison of 2 methods (pp. 585)
  - Java.util.Stack class (pp. 585)

The Collections API

- a supporting library
- resides in `java.util`
- provides a collection of data structures and some generic algorithms (e.g., sorting)

Collections API (The Interface)

A generic protocol that many of the data structures tend to follow:

```
public interface SimpleContainer protocol<AnyType>
{      void insert(AnyType x);
       void remove(AnyType x);
       AnyType find(AnyType x);

       boolean isEmpty();
       void makeEmpty(); }
```

- We do not use the above protocol directly in any code.
- An inheritance based hierarchy of data structures could use this class as a starting point.

The Iterator Pattern

- The Collections API makes heavy use of a common technique known as *iterator pattern*.
- An *iterator* object controls iteration of a collection (i.e., it is used to traverse a collection of objects.)

Example:

```
for(int i = 0; i < v.length; i++)
    System.out.println(v[i]);
```

i is an iterator object

The Iterator Pattern

- The Collections API makes heavy use of a common technique known as *iterator pattern*.
- An *iterator* object controls iteration of a collection.

Example:

```
for(int i = 0; i < v.length; i++)
    System.out.println(v[i]);
```

i is an iterator object

In-Class Exercise 6.1: How does using i as an iterator constraint the design?

Basic Iterator Design

- Container class
  - is required to provide an `iterator` method.
  - `iterator` returns an appropriate iterator for the collection.

```
public class My Container
{     Object[] items;
      int size;

      public MyContainerIterator iterator() {
            return new MyContainerIterator(this);
      }
}
```

Basic Iterator Design

- The iterator class has to methods:
  - `hasNext` returns `true` if iterator has not been exhausted
  - `next` returns next item in the collection.

```
public class My ContainerIterator
{       private int current = 0;
        private MyContainer container;

        MyContainerIterator(MyContainer c) {
                container = c;}

        public boolean hasNext() {
                return current < container.size; }

        public Object next() {
                return container.items[current++]; }
}
```

Basic Iterator Design

```
public static void main(String [] args)
{ MyContainer v = new MyContainer();
  v.add("3");
  v.add("2");
  System.out.println("Container contains:");
  MyContainerIterator itr = v.iterator();
       while(itr.hasNext())
              System.out.println(itr.next());}}
```

```
public class My Container
{ Object[] items;
  int size;
  public MyContainerIterator iterator() {
    return new MyContainerIterator(this);}}
```

```
public class My ContainerIterator
{ private int current = 0;
  private MyContainer container;

  MyContainerIterator(MyContainer c) {
     container = c;}
  public boolean hasNext() {
     return current < container.size; }
  public Object next() {
     return container.items[current++]; }}
```

Basic Iterator Design

```
public static void main(String [] args)
{ MyContainer v = new MyContainer();
  v.add("3");
  v.add("2");
  System.out.println("Container contains:");
  MyContainerIterator itr = v.iterator();
       while(itr.hasNext())
              System.out.println(itr.next());}}
```

```
public class My Container
{ Object[] items;
  int size;
  public MyContainerIterator iterator() {
    return new MyContainerIterator(this);}}
```

current keeps the current position in the container

```
public class My ContainerIterator
{ private int current = 0;
  private MyContainer container;

  MyContainerIterator(MyContainer c) {
     container = c;}
  public boolean hasNext() {
     return current < container.size; }
  public Object next() {
     return container.items[current++]; }}
```

Basic Iterator Design

```
public static void main(String [] args)
{ MyContainer v = new MyContainer();
  v.add("3");
  v.add("2");
  System.out.println("Container contains:");
  MyContainerIterator itr = v.iterator();
       while(itr.hasNext())
              System.out.println(itr.next());}}
```

```
public class My Container
{ Object[] items;
  int size;
  public MyContainerIterator iterator() {
    return new MyContainerIterator(this);}}
```

A reference to the container

```
public class My ContainerIterator
{ private int current = 0;
  private MyContainer container;

  MyContainerIterator(MyContainer c) {
     container = c;}
  public boolean hasNext() {
     return current < container.size; }
  public Object next() {
     return container.items[current++]; }}
```

Basic Iterator Design

```
public static void main(String [] args){
MyContainer v = new MyContainer();
  v.add("3");
  v.add("2");
  System.out.println("Container contains:");
  MyContainerIterator itr = v.iterator();
       while(itr.hasNext())
              System.out.println(itr.next());}}
```

In-Class Exercise 6.2:
This iterator design is quite limited . Please explain why?

```
public class My Container
{ Object[] items;
  int size;
  public MyContainerIterator iterator() {
    return new MyContainerIterator(this);}}
```

```
public class My ContainerIterator
{ private int current = 0;
  private MyContainer container;

  MyContainerIterator(MyContainer c) {
     container = c;}
  public boolean hasNext() {
     return current < container.size; }
  public Object next() {
     return container.items[current++]; }}
```

Basic Iterator Design

```
public static void main(String [] args){
MyContainer v = new MyContainer();
  v.add("3");
  v.add("2");
  System.out.println("Container contains:");
  MyContainerIterator itr = v.iterator();
       while(itr.hasNext())
              System.out.println(itr.next());}}
```

```
public class My Container
{ Object[] items;
  int size;
  public MyContainerIterator iterator() {
    return new MyContainerIterator(this);}}
```

```
public class My ContainerIterator
{ private int current = 0;
  private MyContainer container;

  MyContainerIterator(MyContainer c) {
     container = c;}
  public boolean hasNext() {
     return current < container.size; }
  public Object next() {
     return container.items[current++]; }}
```

In-Class Exercise 6.3: `items` and `size` are package visible rather than being private. Could you explain, why?

Basic Iterator Design

```
public static void main(String [] args){
MyContainer v = new MyContainer();
  v.add("3");
  v.add("2");
  System.out.println("Container contains:");
  MyContainerIterator itr = v.iterator();
       while(itr.hasNext())
              System.out.println(itr.next());}}
```

```
public class My Container
{ Object[] items;
  int size;
  public MyContainerIterator iterator() {
    return new MyContainerIterator(this);}}
```

```
public class My ContainerIterator
{ private int current = 0;
  private MyContainer container;

  MyContainerIterator(MyContainer c) {
     container = c;}
  public boolean hasNext() {
     return current < container.size; }
  public Object next() {
     return container.items[current++]; }}
```

In-Class Exercise 6.4:
Why is `MyContainerIterator` is package visible?

Inheritance based Iterators and factories

```
public class MyContainer
{ Object[] items;
  int size;
  public Iterator iterator() {
    return new MyContainerIterator(this);}
  // other methods not shown}
```

```
public interface Iterator
{ boolean hasNext();
  Object next(); }
```

```
public class My ContainerIterator implements Iterator
{ private int current = 0;
  private MyContainer container;

  MyContainerIterator(MyContainer c) {
     container = c;}
  public boolean hasNext() {
     return current < container.size; }
  public Object next() {
     return container.items[current++]; }}
```

The `Collection` Interface

- All containers support the following operations:
- `boolean isEmpty()`
  - Returns `true` if the container contains no elements and false otherwise.
- `int size()`
  - Returns number of elements in the container
- `boolean add(AnyType x)`
  - Adds item x to the container. Returns `true` if the operation succeeds and `false` otherwise.
- `boolean contains(Object x)`
  - Returns `true` if x is in the container and `false` otherwise.
- `boolean remove(Object x)`
  - Removes an item x from the container. Returns `true` if x was removed and `false` otherwise.
- `void clear()`
  - Makes the container empty.

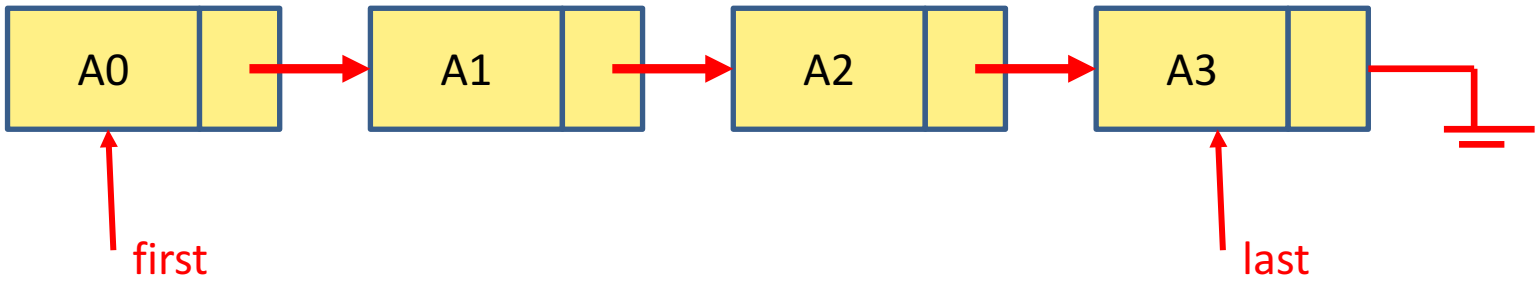
The `Collection` Interface

- All containers support the following operations:
- `Object[] toArray()`
- `<OtherType> OtherType [] toArray(OtherType [] arr)`
  - Returns an array that contains references to all items in the container.
- `java.util.Iterator<AnyType> iterator()`
  - Returns an iterator that can be used to begin traversing all locations in the container.
- In-Class Exercise 6.5: Which one should be used if the collection is to be accessed several times or via a nested loop? How about if the collection is to be accessed only once? Please, explain why?

`Iterator` Interface

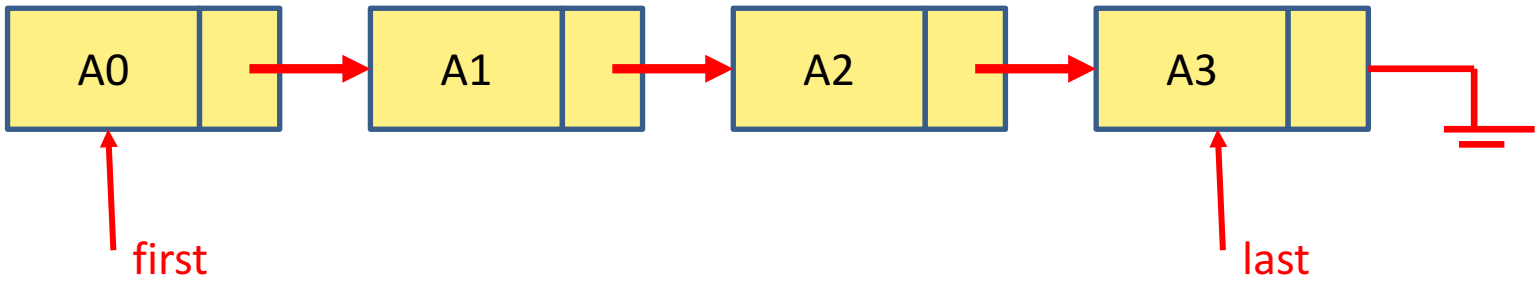
- Iterator interface in the Collections API is small and contains only three methods:
- `Boolean hasNext()`
  - Returns true if there are more items to view in this iteration.
- `AnyType next()`
  - Returns a reference to the next object not yet seen by its iterator. The object becomes seen and thus advances the iterator.
- `void remove()`
  - Removes the last item viewed by `next`. This can be done only once between two next operations.
- **IMPORTANT:** When a container is modified while the iteration is in progress `ConcurrentModificationException` will be thrown as exception by one of the iterator methods.

`LinkedList` class



- Operations:
  - `add (at the end)`
  - `add (at the front)`
  - `remove (at the end)`
  - `remove (at the end)`
  - `get and set`
  - `contains`

LinkedList class



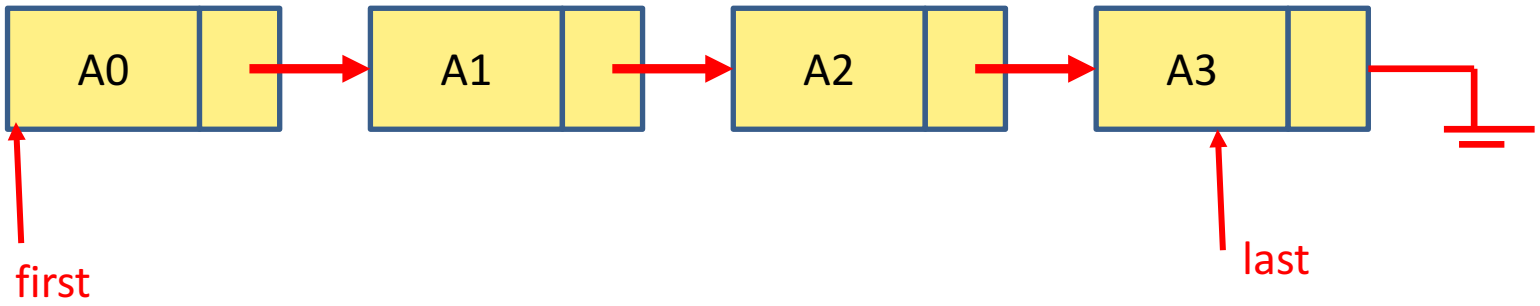
- A typical node looks like:

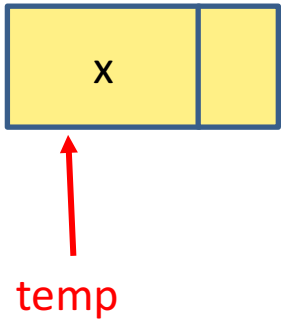
```
class ListNode {
      Object data;
      Listnode next; }
```

- Adding a new last item:

```
last.next = new ListNode();
last = last.next;
last.data = x;
last.next = null;
```

LinkedList class





- A typical node looks like:

```
class ListNode {
      Object data;
      Listnode next; }
```

- Adding a new first item:

```
temp = new ListNode();
temp.data = x;
temp.next = first;
first = temp;
```

Costs for LinkedList vs. ArrayList

- In Class Exercise 6.6: Fill in the single operation costs in terms of Big-O complexities for `ArrayList` and `LinkedList` below for the best case. Explain, why?

	ArrayList	LinkedList
Add/remove at end		
Add/remove at front		
get/set		
contains		

Costs for LinkedList vs. ArrayList

- In Class Exercise 6.7: Using the following code, you construct a `List` by adding items at the end. What is running time (the Big-O complexity) of the following code if:
  - `ArrayList` is passed as parameter
  - `LinkedList` is passed as parameter

```
public static void makeList(List<Integer> lst, int N) {
        lst.clear();
        for(int i = 0; i < N; i++)
                lst.add(i);
}
```

Costs for LinkedList vs. ArrayList

- In Class Exercise 6.8: Using the following code, you construct a `List` by adding items at the front. What is running time (the Big-O complexity) of the following code if:
  - `ArrayList` is passed as parameter
  - `LinkedList` is passed as parameter

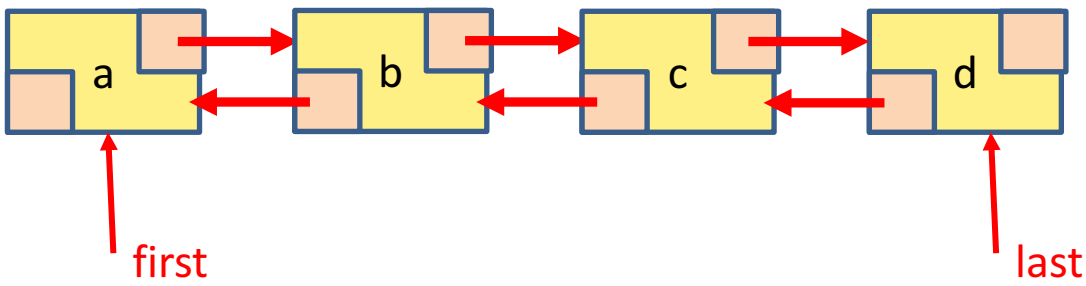
```
public static void makeList2(List<Integer> lst, int N) {
        lst.clear();
        for(int i = 0; i < N; i++)
                lst.add(0, i);
}
```

Costs for LinkedList vs. ArrayList

- In Class Exercise 6.9: The following code cattempts to compute the sum of the numbers in a List. What is running time (the Big-O complexity) of the following code if:
  - `ArrayList` is passed as parameter
  - `LinkedList` is passed as parameter

```
public static int sum(List<Integer> lst) {
        int total = 0;
        for(int i = 0; i < N; i++)
                total += lst.get(i);
}
```

Doubly Linked list



	Doubly LinkedList
Add/remove at end	
Add/remove at front	
get/set	
contains	