CS311 Data Structures
Lecture 03 — List

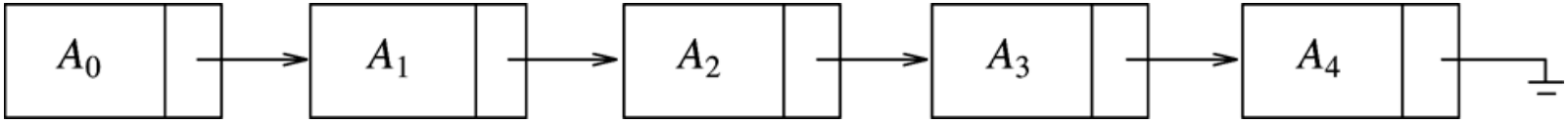
Jyh-Ming Lien

June 6, 2018

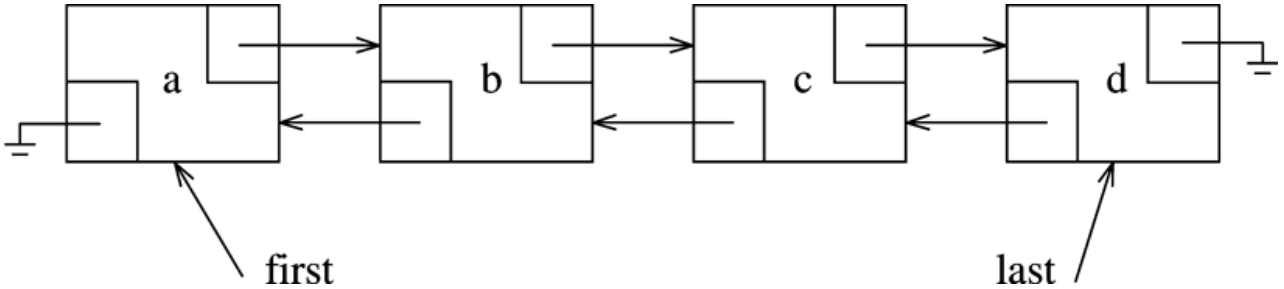
What are lists?

- A list is an abstract data structure that implements an ordered collection of values, where the same value may occur more than once — wikipedia
- Array list, e.g. int A[100]; or `ArrayList<E>` and `Vector<E>` from Java Collections framework, which implements **expandable array**

- Linked list



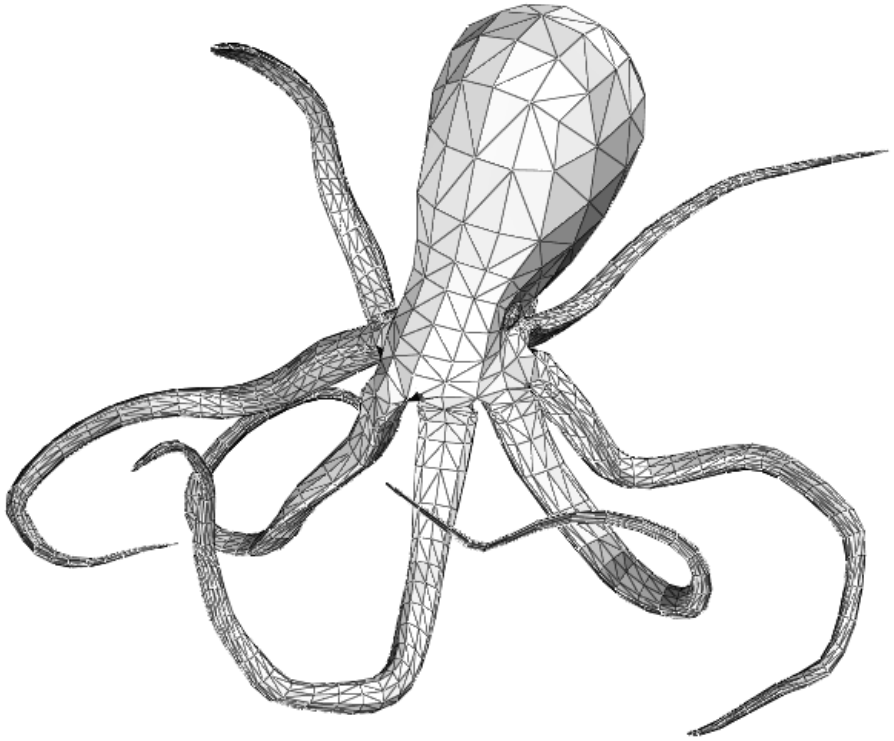
- Doubly linked list (e.g. `LinkedList<E>` in Java Collections)



Basic operations

- Common operations for both Array and Linked lists
  - add(E o), add(int $index$, E o)
  - clear
  - get(int $index$)
  - indexOf(Object o)
  - remove(int $index$)
  - isEmpty()
  - size()
  - listIterator(int $index$)
- Special operations for linked list
  - addLast(E o), addFirst(E o)
  - removeFirst(), removeLast()
  - poll(), peek(), offer(E o)

Geometry representation



Wavefront OBJ file format

v -0.5 -0.5 0.5 v 0.5 -0.5 0.5
v -0.5 0.5 0.5 v 0.5 0.5 0.5
v -0.5 0.5 -0.5 v 0.5 0.5 -0.5
v -0.5 -0.5 -0.5 v 0.5 -0.5 -
0.5 f 1 2 3 f 2 4 3 f 3 4 5 f
4 6 5 f 5 6 7 f 6 8 7 f 7 8 1
f 8 2 1 f 2 8 4 f 8 6 4 f 7 1
5 f 1 3 5

- To draw the geometry (e.g., in OpenGL), we need to:
  - get each triangle
  - get each triangle, get the coordinates of all three vertices
- What data structures should we use to
  - store vertices
  - store triangles

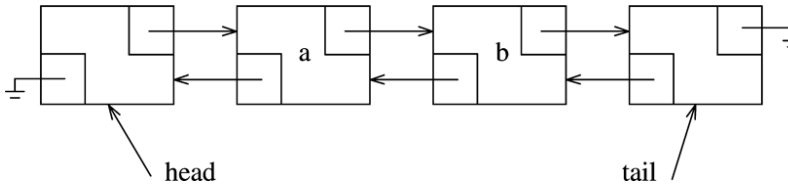
```
1  public class MyArrayList<AnyType> implements Iterable<AnyType>
2  {
3      private static final int DEFAULT_CAPACITY = 10;
4
5      private int theSize;
6      private AnyType [ ] theItems;
7
8      public MyArrayList( )
9        { clear( ); }
10
11     public void clear( )
12     {
13         theSize = 0;
14         ensureCapacity( DEFAULT_CAPACITY );
15     }
16
17     public int size( )
18       { return theSize; }
19     public boolean isEmpty( )
20       { return size( ) == 0; }
21     public void trimToSize( )
22       { ensureCapacity( size( ) ); }
23
24     public AnyType get( int idx )
25     {
26         if( idx < 0 || idx >= size( ) )
27             throw new ArrayIndexOutOfBoundsException( );
28         return theItems[ idx ];
29     }
30
31     public AnyType set( int idx, AnyType newVal )
32     {
33         if( idx < 0 || idx >= size( ) )
34             throw new ArrayIndexOutOfBoundsException( );
35         AnyType old = theItems[ idx ];
36         theItems[ idx ] = newVal;
37         return old;
38     }
39
40     public void ensureCapacity( int newCapacity )
41     {
42         if( newCapacity < theSize )
43             return;
44
45         AnyType [ ] old = theItems;
46         theItems = (AnyType []) new Object[ newCapacity ];
47         for( int i = 0; i < size( ); i++ )
48             theItems[ i ] = old[ i ];
49     }
```

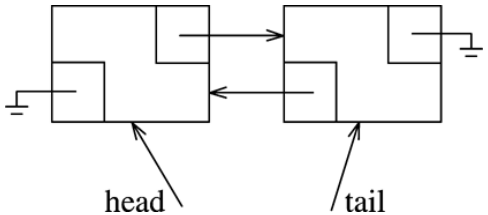
```
50      public boolean add( AnyType x )
51      {
52          add( size( ), x );
53          return true;
54      }
55
56      public void add( int idx, AnyType x )
57      {
58          if( theItems.length == size( ) )
59              ensureCapacity( size( ) * 2 + 1 );
60          for( int i = theSize; i > idx; i-- )
61              theItems[ i ] = theItems[ i - 1 ];
62          theItems[ idx ] = x;
63
64          theSize++;
65      }
66
67      public AnyType remove( int idx )
68      {
69          AnyType removedItem = theItems[ idx ];
70          for( int i = idx; i < size( ) - 1; i++ )
71              theItems[ i ] = theItems[ i + 1 ];
72
73          theSize--;
74          return removedItem;
75      }
76
77      public java.util.Iterator<AnyType> iterator( )
78        { return new ArrayListIterator( ); }
79
80      private class ArrayListIterator implements java.util.Iterator<AnyType>
81      {
82          private int current = 0;
83
84          public boolean hasNext( )
85            { return current < size( ); }
86
87          public AnyType next( )
88          {
89              if( !hasNext( ) )
90                  throw new java.util.NoSuchElementException( );
91              return theItems[ current++ ];
92          }
93
94          public void remove( )
95            { MyArrayList.this.remove( --current ); }
96      }
97  }
```

Linked list

```
1     private static class Node<AnyType>
2     {
3         public Node( AnyType d, Node<AnyType> p, Node<AnyType> n )
4           { data = d; prev = p; next = n; }
5
6         public AnyType data;
7         public Node<AnyType>   prev;
8         public Node<AnyType>   next;
9     }
```

```
1      /**
2       * Change the size of this collection to zero.
3       */
4      public void clear( )
5      {
6          beginMarker = new Node<AnyType>( null, null, null );
7          endMarker = new Node<AnyType>( null, beginMarker, null );
8          beginMarker.next = endMarker;
9
10         theSize = 0;
11         modCount++;
12     }
```





Linked list

```
1       /**
2        * Gets the Node at position idx, which must range from 0 to size( ).
3        * @param idx index of node being obtained.
4        * @return internal node corresponding to idx.
5        * @throws IndexOutOfBoundsException if idx is not between 0 and size().
6        */
7       private Node<AnyType> getNode( int idx )
8       {
9           Node<AnyType> p;
10
11          if( idx < 0 || idx > size( ) )
12              throw new IndexOutOfBoundsException( );
13
14          if( idx < size( ) / 2 )
15          {
16              p = beginMarker.next;
17              for( int i = 0; i < idx; i++ )
18                  p = p.next;
19          }
20          else
21          {
22              p = endMarker;
23              for( int i = size( ); i > idx; i-- )
24                  p = p.prev;
25          }
26
27          return p;
28      }
```

Linked list

```
1   /**
2    * Adds an item to this collection, at specified position p.
3    * Items at or after that position are slid one position higher.
4    * @param p Node to add before.
5    * @param x any object.
6    * @throws IndexOutOfBoundsException if idx is not between 0 and size(),.
7    */
8   private void addBefore( Node<AnyType> p, AnyType x )
9   {
10      Node<AnyType> newNode = new Node<AnyType>( x, p.prev, p );
11      newNode.prev.next = newNode;
12      p.prev = newNode;
13      theSize++;
14      modCount++;
15  }
```

```
1     /**
2      * Removes the object contained in Node p.
3      * @param p the Node containing the object.
4      * @return the item was removed from the collection.
5      */
6     private AnyType remove( Node<AnyType> p )
7     {
8         p.next.prev = p.prev;
9         p.prev.next = p.next;
10        theSize--;
11        modCount++;
12
13        return p.data;
14    }
```

```
1  public class MyLinkedList<AnyType> implements Iterable<AnyType>
2  {
3      private static class Node<AnyType>
4        { ... }
5
6      public MyLinkedList( )
7        { clear( ); }
8
9      public void clear( )
10       { /* Figure 3.25 */ }
11     public int size( )
12       { return theSize; }
13     public boolean isEmpty( )
14       { return size( ) == 0; }
15
16     public boolean add( AnyType x )
17       { add( size( ), x );  return true; }
18     public void add( int idx, AnyType x )
19       { addBefore( getNode( idx ), x ); }
20     public AnyType get( int idx )
21       { return getNode( idx ).data; }
22     public AnyType set( int idx, AnyType newVal )
23     {
24         Node<AnyType> p = getNode( idx );
25         AnyType oldVal = p.data;
26         p.data = newVal;
27         return oldVal;
28     }
29     public AnyType remove( int idx )
30       { return remove( getNode( idx ) ); }
31
32     private void addBefore( Node<AnyType> p, AnyType x )
33       { ... }
34     private AnyType remove( Node<AnyType> p )
35       { ... }
36     private Node<AnyType> getNode( int idx )
37       { ... }
38
39     public java.util.Iterator<AnyType> iterator( )
40        { return new LinkedListIterator( ); }
41     private class LinkedListIterator implements java.util.Iterator<AnyType>
42       { ... }
43
44     private int theSize;
45     private int modCount = 0;
46     private Node<AnyType> beginMarker;
47     private Node<AnyType> endMarker;
48 }
```

Text

```
1       private class LinkedListIterator implements java.util.Iterator<AnyType>
2       {
3           private Node<AnyType> current = beginMarker.next;
4           private int expectedModCount = modCount;
5           private boolean okToRemove = false;
6
7           public boolean hasNext( )
8             { return current != endMarker; }
9
10          public AnyType next( )
11          {
12              if( modCount != expectedModCount )
13                  throw new java.util.ConcurrentModificationException( );
14              if( !hasNext( ) )
15                  throw new java.util.NoSuchElementException( );
16
17              AnyType nextItem = current.data;
18              current = current.next;
19              okToRemove = true;
20              return nextItem;
21          }
22
23          public void remove( )
24          {
25              if( modCount != expectedModCount )
26                  throw new java.util.ConcurrentModificationException( );
27              if( !okToRemove )
28                  throw new IllegalStateException( );
29
30              MyLinkedList.false.remove( current.prev );
31              okToRemove = true;
32          }
33      }
```

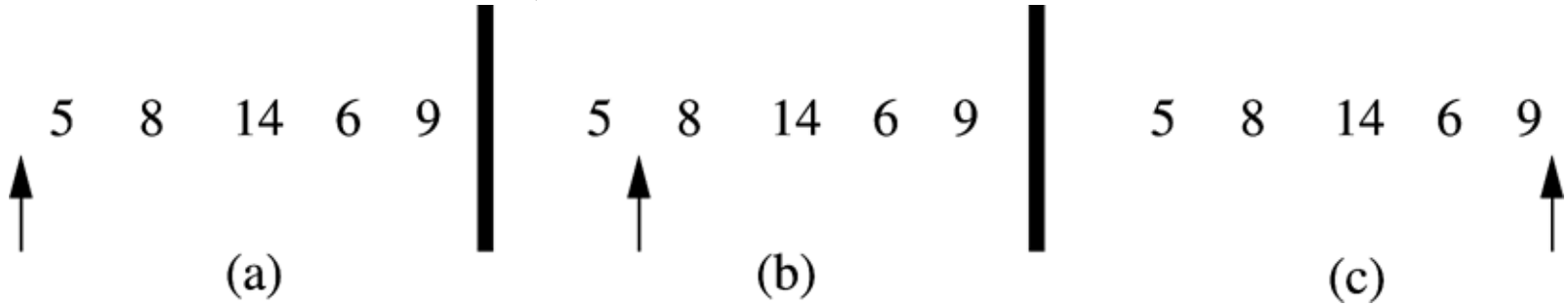
Major differences

	Array list	Linked list
add(E o)		
add(int $index$, E o)		
clear		
get(int $index$)		
indexOf(Object o)		
remove(int $index$)		
removeLast()		
isEmpty()		
ListIterator.remove()		

- be very careful about which list to use

Iterators

- Iterators are pointers to the object in the list
- Java Collection creates/provides an iterator



- operators
  - Use `next()/previous()` to move
  - `next()/previous()` returns element "moved over"
  - `remove()` removes element that was returned from last `next()/previous()`
  - Illegal to remove w/o first calling `next()/previous()`
  - `add(x)` before whatever `next()` would return
  - set(E o)

Iterators are In Between Elements

List Iterators have slightly complex semantics: *between* list elements

Next/Previous

```
LL l = new LL([A, B, C, D])
itr = l.iterator()
                [ A B C D ]
                  ^
itr.next()      [ A B C D ]
A                   ^
itr.next()      [ A B C D ]
B                       ^
itr.previous()  [ A B C D ]
B                   ^
itr.previous()  [ A B C D ]
A                 ^
itr.previous()
--> NoSuchElementException
                [ A B C D ]
                  ^
```

Add/Remove

```
LL l = new LL([A, B, C, D])
itr = l.iterator()
                [ A B C D]
                  ^
itr.add(X)      [ X A B C D ]
                    ^
itr.next()      [ X A B C D ]
A                       ^
itr.next()      [ X A B C D ]
B                           ^
itr.remove()    [ X A C D ]
                        ^
itr.remove()    [ X A C D ]
--> ERROR               ^
itr.previous()  [ X A C D ]
A                   ^
itr.add(Y)      [ X Y A C D ]
                        ^
```

Exercise: Remove even elements from an integer list. Think about how you would use array list or linked list in these implementations.

```
1       public static void removeEvensVer1( List<Integer> lst )
2       {
3           int i = 0;
4           while( i < lst.size( ) )
5               if( lst.get( i ) % 2 == 0 )
6                   lst.remove( i );
7               else
8                   i++;
9       }
```

```
1       public static void removeEvensVer2( List<Integer> lst )
2       {
3           for( Integer x : lst )
4               if( x % 2 == 0 )
5                   lst.remove( x );
6       }
```

```
1       public static void removeEvensVer3( List<Integer> lst )
2       {
3           Iterator<Integer> itr = lst.iterator( );
4
5           while( itr.hasNext( ) )
6               if( itr.next( ) % 2 == 0 )
7                   itr.remove( );
8       }
```