

Data Structures and Algorithms

Chapter 7

Lists

List ADT

- Defines an ADT that specifies a general list data structure.
- The location of an element is determined by an *index*.
- The index of an element e is the number of elements before e in the list.
- So, the index of the first element is 0 and that of the last element is $n - 1$, assuming that there are n elements in the list.
- The ADT supports the operations in the following slide.

Lists

List ADT

- *size()*: Returns the number of elements currently in the list.
- *isEmpty()*: Returns *true* if the list is empty. Returns *false* otherwise.
- *get(i)*: Returns the element whose index is *i*.
- *set(i, e)*: The element at index *i* is replaced with a new element *e* and the old, replaced element is returned.
- *add(i, e)*: Inserts a new element *e* at location with index *i*, and the element which is currently at index *i* and subsequent elements are moved one index later in the list.
- *remove(i)*: Removes and returns the element at index *i*. The elements that are currently in $[i+1 .. \text{size}() - 1]$ are moved one index earlier in the list.
- An error occurs if *i* is not in the range $[0 .. \text{size}() - 1]$, except for the *add* method, for which a valid range is $[0 .. \text{size}()]$.
- *List.java* (interface)

Lists

List ADT

- Illustration

| Operation | Return Value | List Contents |
|------------|--------------|------------------|
| add(0, 25) | none | (25) |
| add(0, 32) | none | (32, 25) |
| add(2, 12) | none | (32, 25, 12) |
| add(2, 15) | none | (32, 25, 15, 12) |
| get(2) | 15 | (32, 25, 15, 12) |
| get(4) | "error" | (32, 25, 15, 12) |
| size() | 4 | (32, 25, 15, 12) |
| remove(2) | 15 | (32, 25, 12) |
| remove(3) | "error" | (32, 25, 12) |
| size() | 3 | (32, 25, 12) |
| get(1) | 25 | (32, 25, 12) |
| set(0, 10) | 32 | (10, 25, 12) |
| size() | 3 | (10, 25, 12) |
| get(1) | 25 | (10, 25, 12) |
| set(4, 29) | "error" | (10, 25, 12) |

Lists

Array Lists

- A list is implemented using an array as an underlying storage.
- Advantage: direct access to elements
- Disadvantage:
 - Adding or removing elements may require restructuring (shifting of elements) of the array.
 - Size is fixed

Lists

Array Lists with Bounded Array

- ArrayList class

```
1 public class ArrayList<E> implements List<E> {
2     // instance variables
3     public static final int CAPACITY=16;    // default array capacity
4     private E[ ] data;                      // generic array used for storage
5     private int size = 0;                   // current number of elements
6     // constructors
7     public ArrayList() {this(CAPACITY);}
8     public ArrayList(int capacity) {
8         data = (E[ ]) new Object[capacity];
9     }
```

...

Lists

Array Lists with Bounded Array

- **Methods**

```
1  public int size() { return size; }
```

```
2  public boolean isEmpty() { return size == 0; }
```

```
3  public E get(int i) throws IndexOutOfBoundsException {
```

```
4      checkIndex(i, size);
```

```
5      return data[i];
```

```
6  }
```

```
7  public E set(int i, E e) throws IndexOutOfBoundsException {
```

```
8      checkIndex(i, size);
```

```
9      E temp = data[i];
```

```
10     data[i] = e;
```

```
11     return temp;
```

```
12 }
```

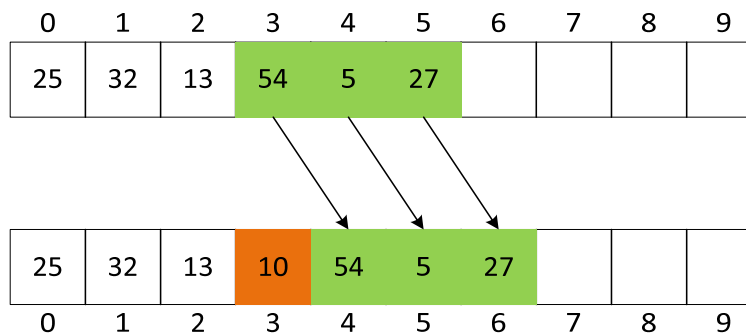
Lists

Array Lists with Bounded Array

- Methods (continued)

```
1 public void add(int i, E e) throws IndexOutOfBoundsException {  
2     checkIndex(i, size + 1);  
3     if (size == data.length)          // not enough capacity  
4         throw new IllegalStateException("Array is full");  
5     for (int k=size-1; k >= i; k--)    // start by shifting rightmost  
6         data[k+1] = data[k];  
7     data[i] = e;                      // ready to place the new element  
8     size++;  
9 }
```

add(3, 10)

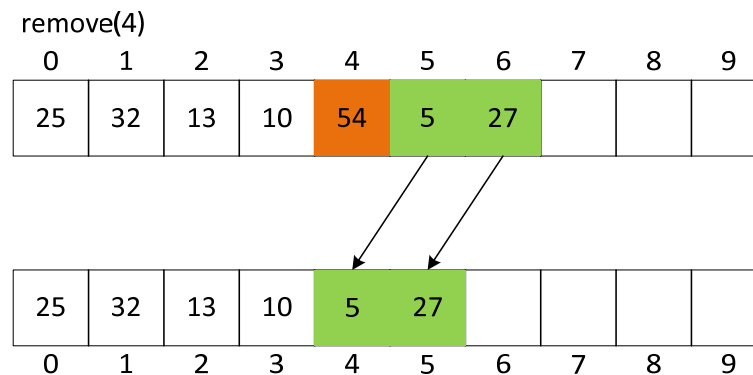


Lists

Array Lists with Bounded Array

- Methods (continued)

```
1 public E remove(int i) throws IndexOutOfBoundsException {  
2     checkIndex(i, size);  
3     E temp = data[i];  
4     for (int k=i; k < size-1; k++)        // shift elements to fill hole  
5         data[k] = data[k+1];  
6     data[size-1] = null;                  // help garbage collection  
7     size--;  
8     return temp;  
9 }
```



Lists

Array Lists with Bounded Array

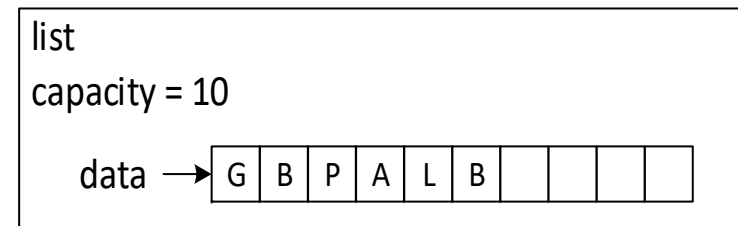
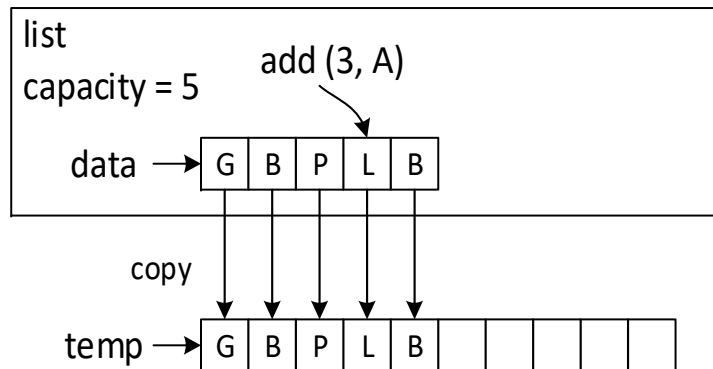
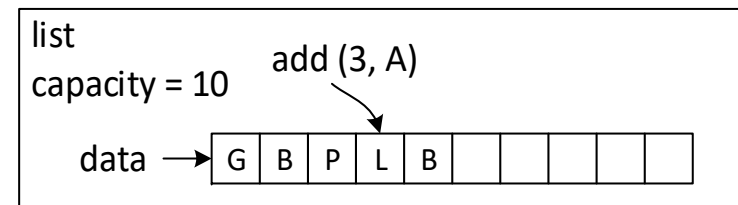
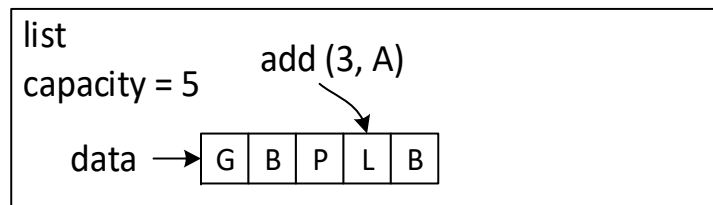
- Running time analysis

| Method | Running Time |
|------------|--------------|
| size() | $O(1)$ |
| isEmpty() | $O(1)$ |
| get(i) | $O(1)$ |
| set(i, e) | $O(1)$ |
| add(i, e) | $O(n)$ |
| remove(i) | $O(n)$ |

Lists

Array Lists with Dynamic Array

- Resize the internal array when the array is full.



Lists

Array Lists with Dynamic Array

- Resize method

```
1  protected void resize(int capacity) {  
2      E[ ] temp = (E[ ]) new Object[capacity];  
3      for (int k=0; k < size; k++)  
4          temp[k] = data[k];  
5      data = temp;  
6  }
```

Lists

Array Lists with Dynamic Array

- Revised add method

```
1  public void add(int i, E e) throws IndexOutOfBoundsException {  
2      checkIndex(i, size + 1);  
3      if (size == data.length)          // not enough capacity, overflow  
4          resize(2 * data.length);      // increase the capacity  
5      for (int k=size-1; k >= i; k--)    // start by shifting rightmost  
6          data[k+1] = data[k];  
7      data[i] = e;                      // ready to place the new element  
8      size++;  
9  }
```

- *ArrayList.java*

Lists

Positional Lists

- Our textbook discusses *positional list*, which uses a notion of *position*. But, we will not discuss this topic in the class. Below is a brief description of positional list.
- A *position* is an abstraction that represents a location of an element in a list.
- A position hides internal nodes (or details) of lists.
- A position allows a user to refer to any element in a list regardless of its location.
- We can perform local operations such as *add before* and *add after*.
- An example: a *cursor* in a text document.

Lists

Java Iterator and Iterable

- An *Iterator* object is an abstraction.
- It provides a uniform way of traversing collections regardless of their internal organizations.
- The *Iterator* interface has the following methods:
 - hasNext(): Returns true if there is at least one additional element in the collection.
 - next(): Returns the next element in the collection.
 - remove(): Removes from the collection the element returned by the most recent call to next(). (optional operation)

Lists

Java Iterator and Iterable

- We create an *Iterator* object by invoking the *iterator()* method that is defined in the *Iterable* interface.
- Example

```
ArrayList<String> stringList = new ArrayList<>( );  
// population of the list omitted  
Iterator<String> stringIterator = stringList.iterator( );  
While (stringIterator.hasNext( ))  
    System.out.println(stringIterator.next( ));
```

- Java *Collection* interface extends the *Iterable* interface so all collection objects can invoke the *iterator()* method to create an iterator.

Lists

Java Iterator and Iterable

- Simpler syntax:

```
for (ElementType variable : collection) {  
    loopBody  
}
```

The previous example is equivalent to:

```
for (String s : stringList) {  
    System.out.println(s);  
}
```

Lists

Java ListIterator

- Java's *ListIterator* interface extends the *Iterator* interface
- Adds bi-directional traversal of a list.
- A list iterator can move forward and backward.
- A list iterator is assumed to be located before the first element, between two consecutive elements, or after the last element.
- A list iterator is obtained by invoking the *listIterator()* method of a *List* interface.
- It inherits all operations of *Iterator* and it also defines additional local update operations.

Lists

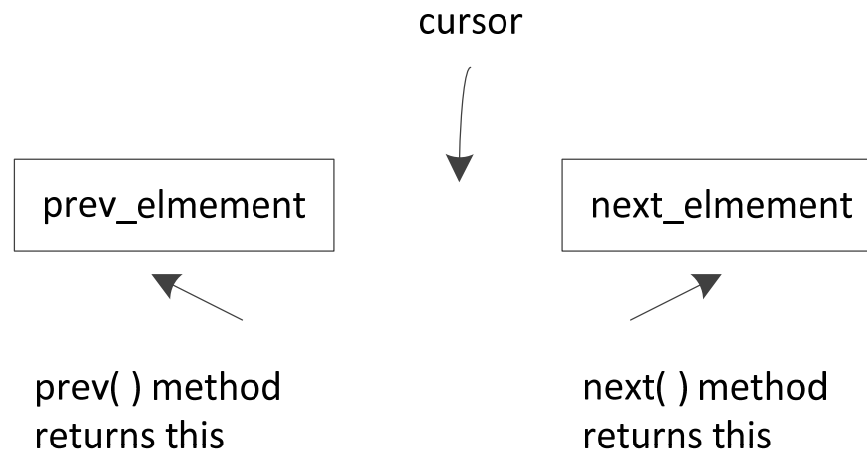
Java ListIterator

- `add(e)`: Inserts the element *e* at the current position of the iterator.
- `hasNext()`
- `hasPrevious()`
- `previous()`: Returns the element *e* before the current iterator position and sets the current position to be before *e*.
- `next()`: Returns the element *e* after the current iterator position and sets the current position to be after *e*.
- `nextIndex()`: Returns the index of the next element.
- `previousIndex()`: Returns the index of the previous element.
- `remove()`: Removes the element returned by the most recent *next* or *previous* operation.
- `set(e)`: Replaces the element returned by the most recent *next* or *previous* operation with *e*.

Lists

Java ListIterator

- Extends the *Iterator* interface
- Allows bidirectional traversal of a list
- *Cursor* is between two elements, say *prev_element* and *next_element*
- *previous()* methods returns *prev_element*
- *next()* methods returns *next_element*



Lists

Java ListIterator

```
LinkedList<Integer> intList = new LinkedList<>();
intList.add(20); intList.add(40); intList.add(60);
ListIterator<Integer> li;
li = intList.listIterator(); // cursor right before the first element
while (li.hasNext()){ // if there is next element
    System.out.print(li.next() + " "); // walk forward
}
System.out.println();
li = intList.listIterator(intList.size()); // cursor right after the last elem.
while (li.hasPrevious()){ // if there is previous element
    System.out.print(li.previous() + " "); // walk backward
}
```

Lists

Java ListIterator

- The out put is:
20 40 60
60 40 20
- If we execute the following statements:
 li = intList.listIterator(2); // cursor is between 2nd and 3rd
 // elements
 li.add(100); // add right before next element
 The list will have: 20 40 100 60
- remove() method removes the last element that was returned by next() or previous()

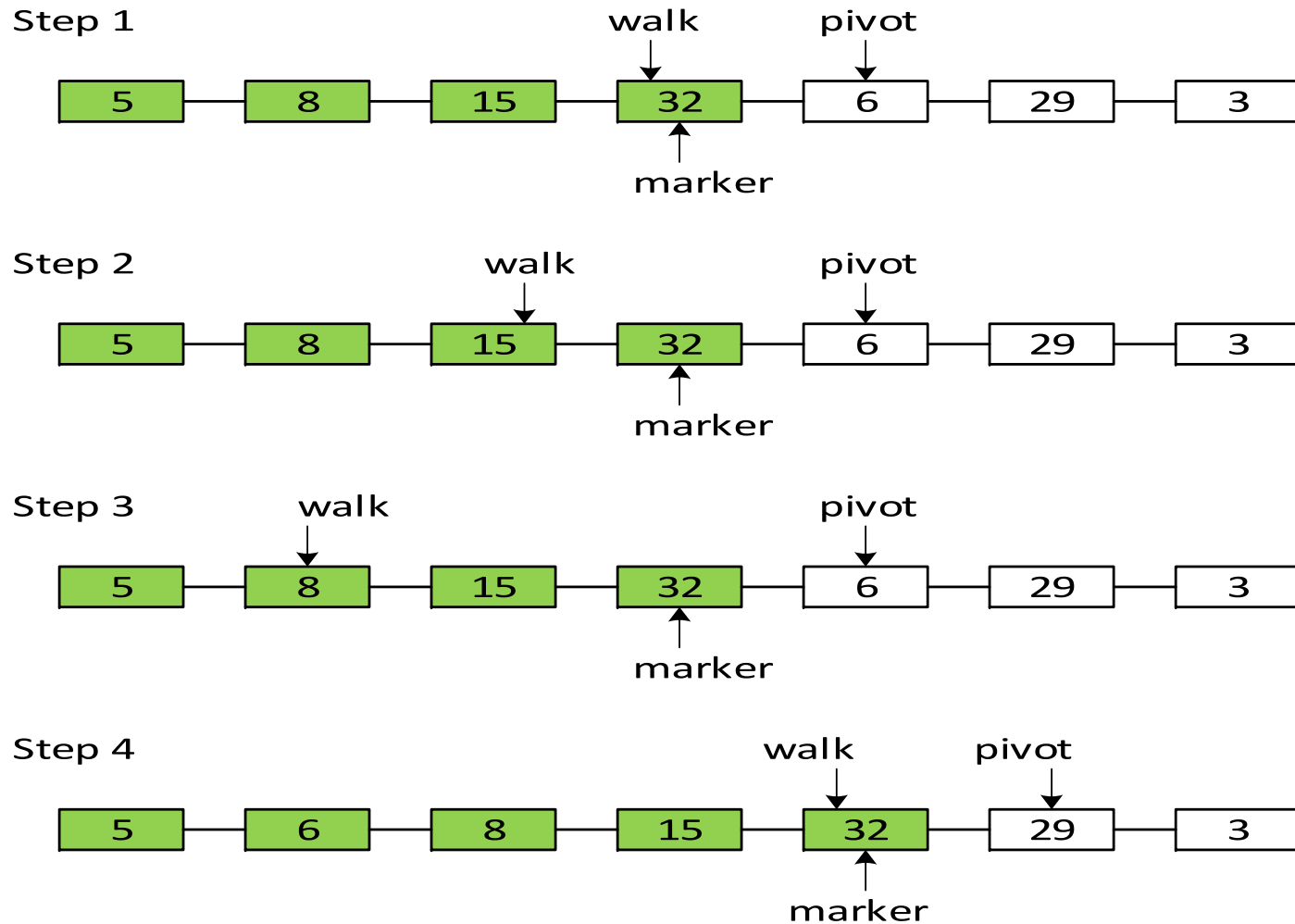
Lists

Sorting a Doubly Linked List

- Sorts an elements in a doubly linked list using the insertion-sort algorithm.
- Uses three variables: *marker*, *pivot*, and *walk*.
- During sorting, the list has two parts.
- One part (on the left): already sorted
- The other part (on the right): has elements not explored
- *marker* is the rightmost node in the already sorted.
- *pivot* is the node of the element to the immediate right of *marker*, and represents the first element in the unsorted part.
- The *walk* is used to traverse the already sorted part of the array to decide the correct location of *pivot*.

Lists

Sorting a Linked List



Lists

Sorting a Linked List

- Refer to *NodeLinkedList.java* and *InsertionSortLinkedList.java*.
- Textbook's code uses *LinkedPositionalList*, which uses *Position*.
- *NodeLinkedList.java* uses *Node*.
- *InsertionSortLinkedList.java* uses *NodeLinkedList*.

References

- M.T. Goodrich, R. Tamassia, and M.H. Goldwasser, “Data Structures and Algorithms in Java,” Sixth Edition, Wiley, 2014.