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 .. *size*( ) – 1] are moved one index earlier in the list.
- An error occurs if *i* is not in the range [0 .. *size*( ) – 1], except for the *add* method, for which a valid range is [0 .. *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

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

. . .
```

Lists
Array Lists with Bounded Array

- Methods

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

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

public E get(int i) throws IndexOutOfBoundsException {
  checkIndex(i, size);
  return data[i];
}

public E set(int i, E e) throws IndexOutOfBoundsException {
  checkIndex(i, size);
  E temp = data[i];
  data[i] = e;
  return temp;
}
```

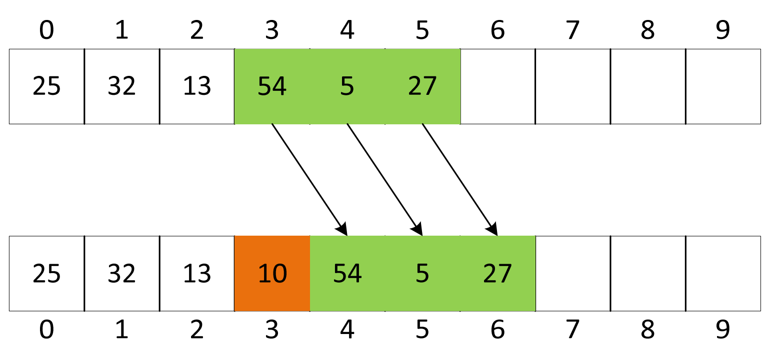
Lists

Array Lists with Bounded Array

- Methods (continued)

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

add(3, 10)

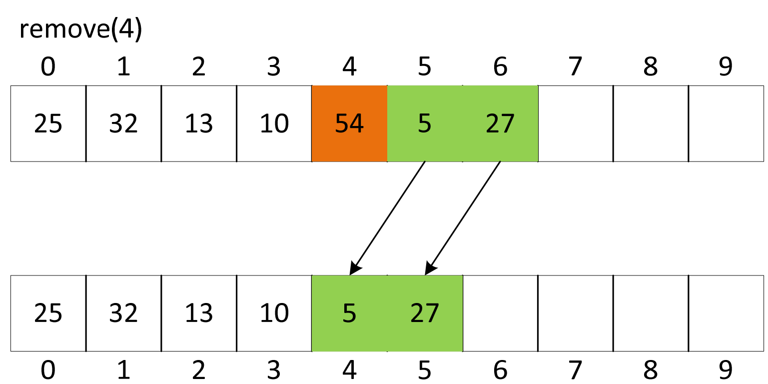


Lists

Array Lists with Bounded Array

- Methods (continued)

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



Lists

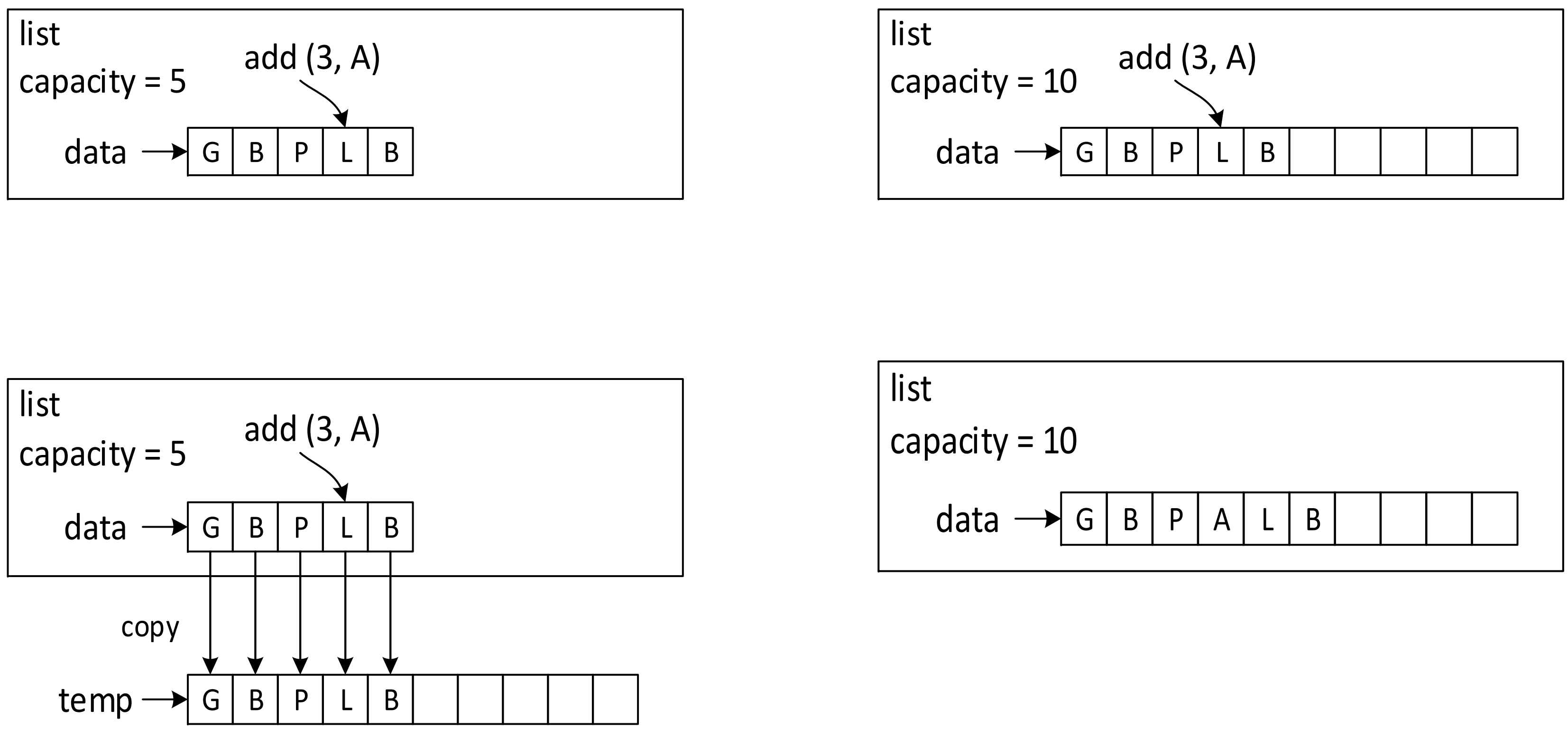
Array Lists with Bounded Array

- 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

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

Lists
Array Lists with Dynamic Array

- Revised add method

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

- *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

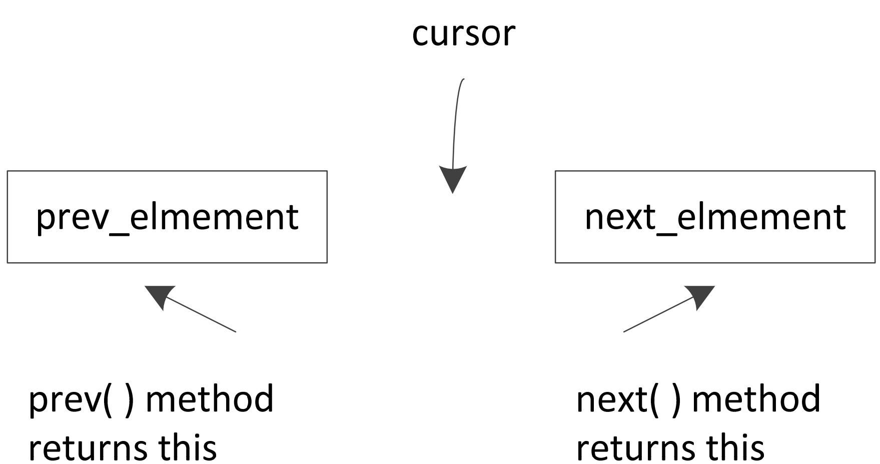
- Java’s *ListIterator* interface extends the *Iterator* interface
- Adds bi-directional traversal of a list.
- A list iterator can move forward and backward.
- A list iterator is assumed to be located before the first element, between two consecutive elements, or after the last element.
- A list iterator 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.
- previoustIndex( ): 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( )

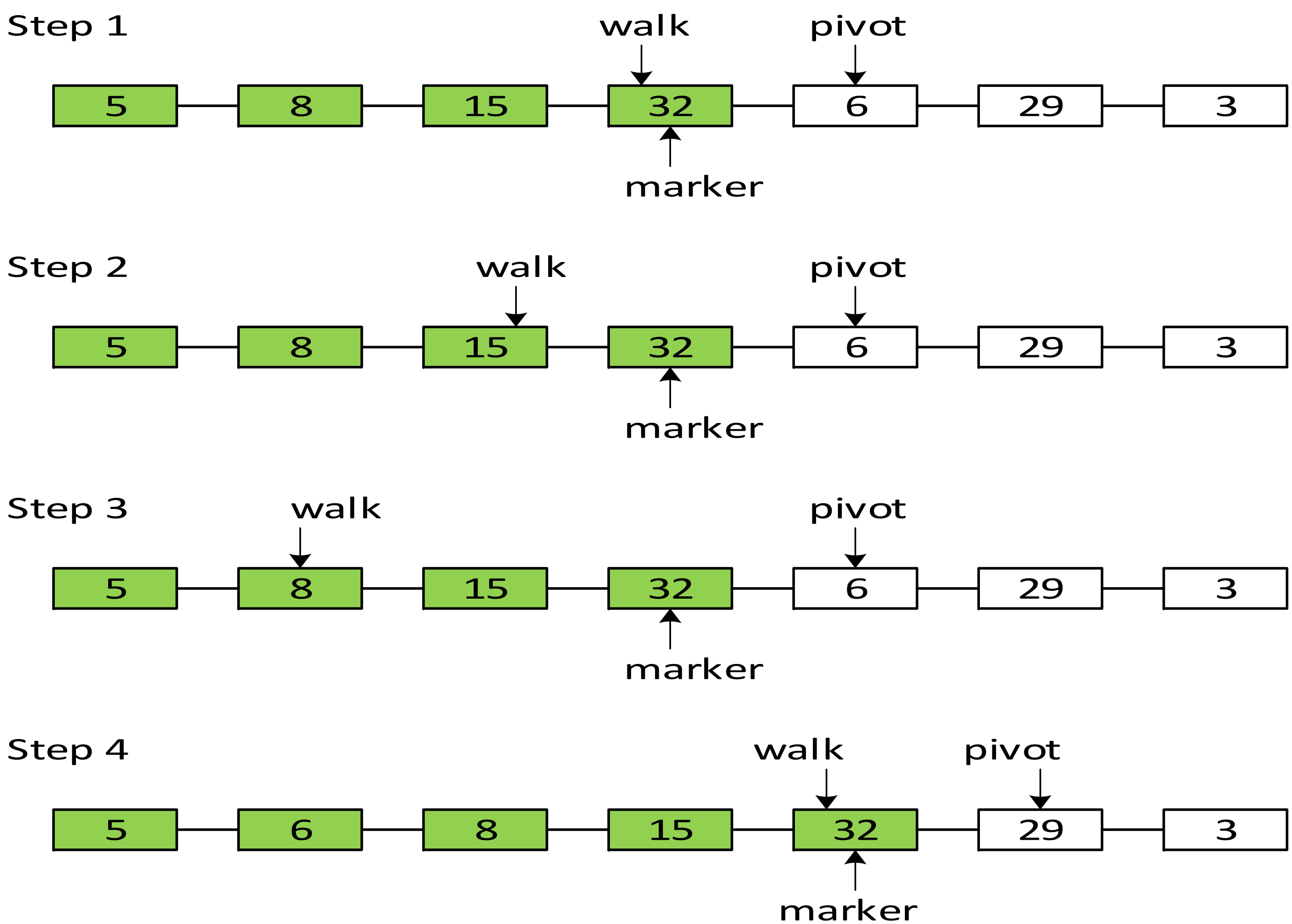
Lists

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 Doubly Linked List



Lists

Sorting a Doubly Linked List

- Textbook's code uses *LinkedPositionalList*, which uses *Position*.
- The code in the next slide uses *ExtendedLinkedList*, which has the same functionality as *LinkedPositionalList* but uses *Node* instead of *Position*.

Lists
Sorting a Doubly Linked List

- Java code

```
public static void insertionSort(ExtendedLinkedList<Integer> list) {
  Node<Integer> marker = list.first(); // last position known to be sorted
  while (marker != list.last()) {
      Node<Integer> pivot = list.after(marker);
      int value = pivot.getElement();      // number to be placed
      if (value > marker.getElement())     // pivot is already sorted
          marker = pivot;
      else {                               // must relocate pivot
          Node<Integer> walk = marker;   // find leftmost item greater than value
          while (walk != list.first() && list.before(walk).getElement() > value)
              walk = list.before(walk);
          list.remove(pivot);              // remove pivot entry and
          list.addBefore(walk, value);     // reinsert value in front of walk
       }
   }
}
```

References

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