Lists

- A list is a collection of elements of the same type that are stored in a certain linear order.
- An element can be accessed, inserted or removed.
- Two types of lists:
  - Array lists
  - Node lists

Array List ADT

- A list that supports access to its elements by their indices.
- The List ADT extends the notion of array by storing a sequence of arbitrary objects.
- An element can be accessed, inserted or removed by specifying its index.
- An exception is thrown if an incorrect rank is specified (e.g., a negative rank)

Main methods:
- get(i): Return the element with index i.
- set(i, e): Replace with e and return the element at index i.
- add(i, e): Insert a new element e at index i.
- remove(i): Remove the element at index i.

Array-Based Implementation

- Use an array A of size N.
- A variable n keeps track of the size of the list (the number of elements stored).
- Method get(i) is implemented in O(1) time by returning A[i].
- Method set(i, e) is implemented in O(1) time by setting A[i] to e.
In method `add(i, e)`, we need to make room for the new element by shifting forward the $n - i$ elements $A[i], ..., A[n-1]$.

In the worst case ($i = 0$), this takes $O(n)$ time.

```
Algorithm add(i, e)
{
    for (j = n-1, n-2, ..., i)
    A[i]=e;
    n=n+1;
}
```

In method `remove(i)`, we need to fill the hole left by the removed element by shifting backward the $n - i - 1$ elements $A[i+1], ..., A[n-1]$.

In the worst case ($i = 0$), this takes $O(n)$ time.

```
Algorithm remove(i)
{
    e=A[i];
    for (j = i, i+1, ..., n-2)
        A[j]=A[j+1];
    n=n-1;
    return e;
}
```

### A Simple Interface

```java
public interface IndexList<E> {
    /** Returns the number of elements in this list. */
    public int size();
    /** Returns whether the list is empty. */
    public boolean isEmpty();
    /** Inserts an element e to be at index i, shifting all elements after this. */
    public void add(int i, E e) throws IndexOutOfBoundsException;
    /** Returns the element at index i, without removing it. */
    public E get(int i) throws IndexOutOfBoundsException;
    /** Removes and returns the element at index i, shifting the elements after this. */
    public E remove(int i) throws IndexOutOfBoundsException;
    /** Replaces the element at index i with e, returning the previous element at i. */
    public E set(int i, E e) throws IndexOutOfBoundsException;
}
```

### Extendable-Array-Based Array Lists (1/4)

In an `add` operation, when the array is full, instead of throwing an exception, we can replace the array with a larger one.

**How large should the new array be?**

- Incremental strategy: increase the size by a constant $c$.
- Doubling strategy: double the size.

```
public interface IndexList<E> {
    /** Returns the number of elements in this list. */
    public int size();
    /** Returns whether the list is empty. */
    public boolean isEmpty();
    /** Inserts an element e to be at index i, shifting all elements after this. */
    public void add(int i, E e) throws IndexOutOfBoundsException;
    /** Returns the element at index i, without removing it. */
    public E get(int i) throws IndexOutOfBoundsException;
    /** Removes and returns the element at index i, shifting the elements after this. */
    public E remove(int i) throws IndexOutOfBoundsException;
    /** Replaces the element at index i with e, returning the previous element at i. */
    public E set(int i, E e) throws IndexOutOfBoundsException;
}
```
Extendable-Array-Based Array Lists (2/4)

/** Realization of an indexed list by means of an array, which is doubled when the size of the indexed list exceeds the capacity of the array. */

public class ArrayIndexList<E> implements IndexList<E> {

    private E[] A; // array storing the elements of the indexed list
    private int capacity = 16; // initial length of array A
    private int size = 0; // number of elements stored in the indexed list

    /** Creates the indexed list with initial capacity 16. */
    public ArrayIndexList() {
        A = (E[]) new Object[capacity]; // the compiler may warn, but this is ok
    }

    private void checkIndex(int i, int size) throws IndexOutOfBoundsException {
        if (size == capacity) { // an overflow
            capacity *= 2;
            E[] B = (E[]) new Object[capacity];
            for (int i = 0; i < size; i++) B[i] = A[i];
            A = B;
        }
    }

    /** Inserts an element at the given index. */
    public void add(int r, E e) throws IndexOutOfBoundsException {
        checkIndex(r, size++);
        for (int i = size - 1; i >= r; i--)
            A[i + 1] = A[i];
        A[r] = e;
        size++;
    }

    Extendable-Array-Based Array Lists (3/4)

    /** Removes the element stored at the given index. */
    public E remove(int r) throws IndexOutOfBoundsException {
        checkIndex(r, size--);
        E temp = A[r];
        for (int i = r; i < size - 1; i++)
            A[i] = A[i + 1];
        size--;
        return temp;
    }

    Comparison of the Strategies

    - We compare the incremental strategy and the doubling strategy by analyzing the total time $T(n)$ needed to perform a series of $n$ push operations.
    - A push operation is to add an element at the end of the list.
    - We assume that we start with an empty list represented by an array of size 1.
    - We call amortized time of a push operation the average time taken by a push over the series of operations, i.e., $T(n)/n$.

    Incremental Strategy Analysis

    - We replace the array $k = n/c$ times.
    - The total time $T(n)$ of a series of $n$ push operations is proportional to
      \[
      n + c + 2c + 3c + 4c + \ldots + kc
      = n + c(1 + 2 + 3 + \ldots + k)
      = n + ck(k + 1)/2.
      \]
    - Since $c$ is a constant, $T(n)$ is $O(n + k^2)$, i.e., $O(n^2)$.
    - The amortized time of a push operation is $O(n)$.

    Doubling Strategy Analysis

    - We replace the array $k = \log n$ times.
    - The total time $T(n)$ of a series of $n$ push operations is proportional to
      \[
      n + 1 + 2 + 4 + 8 + \ldots + 2^k
      = n + 2^k - 1 = 2n - 1.
      \]
    - $T(n)$ is $O(n)$.
    - The amortized time of a push operation is $O(1)$. 
Position ADT

- The Position ADT models the notion of place within a data structure where a single object is stored.
- It gives a unified view of diverse ways of storing data, such as:
  - A cell of an array.
  - A node of a linked list.
- Just one method:
  - `object element()`: returns the element stored at the position.

Java Interface for the Position ADT

```java
public interface Position<E> {
    /** Return the element stored at this position. */
    E element();
}
```

Node List ADT

- The Node List ADT models a sequence of positions storing arbitrary objects.
- It establishes a before/after relation between positions.
- Generic methods:
  - `size()`, `isEmpty()`,
- Accessor methods:
  - `first()`, `last()`
- Update methods:
  - `set(p, e)`, `addBefore(p, e)`, `addAfter(p, e)`
- `addFirst(e)`, `addLast(e)`
- `remove(p)`

Java Interface for the Node List ADT (1/2)

```java
public interface PositionList<E> {
    /** Returns the number of elements in this list. */
    public int size();
    /** Returns whether the list is empty. */
    public boolean isEmpty();
    /** Returns the first node in the list. */
    public Position<E> first();
    /** Returns the last node in the list. */
    public Position<E> last();
    /** Returns the node after a given node in the list. */
    public Position<E> next(Position<E> p)
        throws InvalidPositionException, BoundaryViolationException;
    /** Returns the node before a given node in the list. */
    public Position<E> prev(Position<E> p)
        throws InvalidPositionException, BoundaryViolationException;
    /** Inserts an element at the front of the list, returning new position. */
    public void addFirst(E e);
    /** Inserts an element at the end of the list, returning new position. */
    public void addLast(E e);
    /** Inserts an element after the given node in the list. */
    public void addAfter(Position<E> p, E e)
        throws InvalidPositionException;
    /** Inserts an element before the given node in the list. */
    public void addBefore(Position<E> p, E e)
        throws InvalidPositionException;
    /** Removes a node from the list, returning the element stored there. */
    public E remove(Position<E> p)
        throws InvalidPositionException;
    /** Replaces the element stored at the given node, returning old element. */
    public E set(Position<E> p, E e)
        throws InvalidPositionException;
}
```

Java Interface for the Node List ADT (2/2)

```java
public interface PositionList<E> {
    /** Returns the number of elements in this list. */
    public int size();
    /** Returns whether the list is empty. */
    public boolean isEmpty();
    /** Returns the first node in the list. */
    public Position<E> first();
    /** Returns the last node in the list. */
    public Position<E> last();
    /** Returns the node after a given node in the list. */
    public Position<E> next(Position<E> p)
        throws InvalidPositionException, BoundaryViolationException;
    /** Returns the node before a given node in the list. */
    public Position<E> prev(Position<E> p)
        throws InvalidPositionException, BoundaryViolationException;
    /** Inserts an element at the front of the list, returning new position. */
    public void addFirst(E e);
    /** Inserts an element at the end of the list, returning new position. */
    public void addLast(E e);
    /** Inserts an element after the given node in the list. */
    public void addAfter(Position<E> p, E e)
        throws InvalidPositionException;
    /** Inserts an element before the given node in the list. */
    public void addBefore(Position<E> p, E e)
        throws InvalidPositionException;
    /** Removes a node from the list, returning the element stored there. */
    public E remove(Position<E> p)
        throws InvalidPositionException;
    /** Replaces the element stored at the given node, returning old element. */
    public E set(Position<E> p, E e)
        throws InvalidPositionException;
}
```

Doubly Linked List Implementation

- A doubly linked list provides a natural implementation of the Node List ADT.
- Nodes implement Position and store:
  - `element`
  - `link to the previous node`
  - `link to the next node`
- Special trailer and header nodes.
Insertion

- We visualize operation addAfter(p, e), which returns position q

![Diagram of addAfter operation]

Insertion Algorithm

Algorithm insertAfter(p, e)
{
  Create a new node v;
  v.setElement(e);
  v.setPrev(p); // link v to its predecessor
  v.setNext(p.getNext()); // link v to its successor
  (p.getNext()).setPrev(v); // link p’s old successor to v
  p.setNext(v); // link p to its new successor, v
  return v; // the position for the element e
}

Deletion

- We visualize remove(p), where p = last()

![Diagram of remove operation]

Deletion Algorithm

Algorithm remove(p)
{
  t = p.element; // a temporary variable to hold the return value
  (p.getPrev()).setNext(p.getNext()); // linking out p
  (p.getNext()).setPrev(p.getPrev());
  p.setPrev(null); // invalidating the position p
  p.setNext(null);
  return t;
}

Implementation of the Position Interface ADT (1/2)

```java
public class DNode<E> implements Position<E> {
  private DNode<E> prev, next; // References to the nodes before and after
  private E element; // Element stored in this position

  /** Constructor */
  public DNode(DNode<E> newPrev, DNode<E> newNext, E elem) {
    prev = newPrev;
    next = newNext;
    element = elem;
  }

  // Method from interface Position
  public E element() throws InvalidPositionException {
    if ((prev == null) && (next == null)) throw new InvalidPositionException("Position is not in a list!");
    return element;
  }

  // Accessor methods
  public DNode<E> getNext() { return next; }
  public DNode<E> getPrev() { return prev; }
  public void setPrev(DNode<E> newPrev) { prev = newPrev; }
  public void setNext(DNode<E> newNext) { next = newNext; }
  public void setElement(E newElement) { element = newElement; }
}
```
Java Implementation of the Node List ADT (1/7)

```java
public class NodePositionList<E> implements PositionList<E> {
    protected int numElts; // Number of elements in the list
    protected DNode<E> header, trailer; // Special sentinels

    /** Constructor that creates an empty list; O(1) time */
    public NodePositionList() {
        numElts = 0;
        header = new DNode<E>(null, null, null); // create header
        trailer = new DNode<E>(header, null, null); // create trailer
        header.setNext(trailer); // make header and trailer point to each other
    }
}
```

Java Implementation of the Node List ADT (2/7)

```java
/** Checks if position is valid for this list and converts it to DNode if it is valid; O(1) time */
protected DNode<E> checkPosition(Position<E> p) throws InvalidPositionException {
    if (p == null)
        throw new InvalidPositionException("Null position passed to NodeList");
    if (p == header)
        throw new InvalidPositionException("The header node is not a valid position");
    if (p == trailer)
        throw new InvalidPositionException("The trailer node is not a valid position");
    try {
        DNode<E> temp = (DNode<E>) p;
        if ((temp.getPrev() == null) || (temp.getNext() == null))
            throw new InvalidPositionException("Position does not belong to a valid NodeList");
        return temp;
    } catch (ClassCastException e) {
        throw new InvalidPositionException("Position is of wrong type for this list");
    }
}
```

Java Implementation of the Node List ADT (3/7)

```java
/** Returns the number of elements in the list; O(1) time */
public int size() {
    return numElts;
}
/** Returns whether the list is empty; O(1) time */
public boolean isEmpty() {
    return (numElts == 0);}
/** Returns the first position in the list; O(1) time */
public Position<E> first() throws EmptyListException {
    if (isEmpty())
        throw new EmptyListException("List is empty");
    return header.getNext();
}
```

Java Implementation of the Node List ADT (4/7)

```java
/** Returns the position before the given one; O(1) time */
public Position<E> prev(Position<E> p) throws InvalidPositionException, BoundaryViolationException {
    DNode<E> v = checkPosition(p);
    DNode<E> prev = v.getPrev();
    if (prev == header)
        throw new BoundaryViolationException("Cannot advance past the beginning of the list");
    return prev;
}
/** Insert the given element before the given position, returning the new position; O(1) time */
public void addBefore(Position<E> p, E element) throws InvalidPositionException {
    DNode<E> v = checkPosition(p);
    numElts++;
    DNode<E> newNode = new DNode<E>(v.getPrev(), v, element);
    v.getPrev().setNext(newNode);
    v.setPrev(newNode);
}
```

Java Implementation of the Node List ADT (5/7)

```java
/** Insert the given element at the beginning of the list, returning the new position; O(1) time */
public void addFirst(E element) {
    numElts++;
    DNode<E> newNode = new DNode<E>(header, header.getNext(), element);
    header.getNext().setPrev(newNode);
    header.setNext(newNode);
}
```

Java Implementation of the Node List ADT (6/7)

```java
/** Remove the given position from the list; O(1) time */
public E remove(Position<E> p) throws InvalidPositionException {
    DNode<E> v = checkPosition(p);
    numElts--;
    DNode<E> vNext = v.getNext();
    DNode<E> vPrev = v.getPrev();
    vPrev.setNext(vNext);
    v.setPrev(vPrev);
    return v.element;
}
```

Java Implementation of the Node List ADT (7/7)

```java
/** Remove the given position from the list; O(1) time */
public E remove(Position<E> p) throws InvalidPositionException {
    DNode<E> v = checkPosition(p);
    numElts--;
    DNode<E> vNext = v.getNext();
    DNode<E> vPrev = v.getPrev();
    vPrev.setNext(vNext);
    v.setPrev(vPrev);
    return v.element;
}
```
Java Implementation of the Node List ADT (7/7)

```java
/** Replace the element at the given position with the new element and return the old element; O(1) time */
public E set(Position<E> p, E element)
    throws InvalidPositionException
{
    DNode<E> v = checkPosition(p);
    E oldElt = v.element();
    v.setElement(element);
    return oldElt;
}
```

Performance

- In the implementation of the List ADT by means of a doubly linked list
  - The space used by a list with \( n \) elements is \( O(n) \).
  - The space used by each position of the list is \( O(1) \).
  - All the operations of the List ADT run in \( O(1) \) time.
  - Operation `element()` of the Position ADT runs in \( O(1) \) time.

Iterators

- An iterator provides a way to access the elements of an aggregate object sequentially without exposing its underlying representation.
- Extends the concept of Position by adding a traversal capability.
- An iterator consists of a sequence \( S \), a current element in \( S \), and a way of stepping to the next element in \( S \) and making it the current element.
- Methods of the Iterator ADT:
  - boolean hasNext(): Test whether there are elements left in the iterator.
  - object next(): Return the next element in the iterator.

Simple Iterators in JAVA (1/2)

```java
public interface PositionList<E>
    extends Iterable<E>
{
    // ...all the other methods of the list ADT ...;
    /** Returns an iterator of all the elements in the list. */
    public Iterator<E> iterator();
}
```

Simple Iterators in JAVA (2/2)

```java
/** Returns a textual representation of a given node list */
public static <E> String toString(PositionList<E> l)
{
    Iterator<E> it = l.iterator();
    String s = "["
    while (it.hasNext())
    {
        s += it.next(); // implicit cast of the next element to String
        if (it.hasNext())
            s += ", ";
    }
    s += "]";
    return s;
}
```

Implementing Iterators

- Two notions of iterator:
  - snapshot: freezes the contents of the data structure at a given time
  - dynamic: follows changes to the data structure
This approach makes a “snapshot” of a collection of elements and iterates over it.

- It would involve storing the collection in a separate data structure that supports sequential access to its elements.
- Uses a cursor to keep track of the current position of the iterator.
- Creating a new iterator involves creating an iterator object that represents a cursor placed just before the first element of the collection, taking O(1) time.
- `next()` returns the next element, if any, and moves the cursor just past this element's position.

This approach iterates over the data structure directly.

- No separate copy of the data structure is needed.

```java
public class ElementIterator<E> implements Iterator<E> {
    protected PositionList<E> list; // the underlying list
    protected Position<E> cursor; // the next position

    /** Creates an element iterator over the given list. */
    public ElementIterator(PositionList<E> L) {
        list = L; cursor = (list.isEmpty())? null : list.first();
    }

    public boolean hasNext() {
        return (cursor != null);
    }

    public E next() throws NoSuchElementException {
        E toReturn = cursor.element();
        cursor = (cursor == list.last())? null : list.next(cursor);
        return toReturn;
    }
}
```

**Returns an iterator of all the elements in the list.**

```java
public Iterator<E> iterator() {
    return new ElementIterator<E>(this);
}
```

1. Chapter 6, Data Structures and Algorithms by Goodrich and Tamassia.