Simple Data Structures

Lecture 3 – Programming and Data Structures

Dr Jonas Lundberg, office B3024

Jonas.Lundberg@lnu.se

Slides and examples are available in Moodle

1 februari 2019

Simple Data Structures

Today ...

- Overview (A few common data structures)
- Sequential Data Structures
- What is a data structure?
- Iterators
- Array-based Data Structures
- Linked Data Structures
- Javadoc Documentation

Data Structures – Introduction

- We often need to handle large sets of data
- A *data structure* is a model for storing/handling such data sets
- Scenarios where data structures are needed
  1. Students in a course
  2. Measurements from an experiment
  3. Queue to get an apartment at our campus
  4. Telephone numbers in Stockholm
- Different scenarios require different data structure properties
  - Data should be ordered
  - Not the same element twice
  - Important that look-up is fast
  - In general: Important that operations X,Y,Z are fast
- Selecting data structure is a design decision $\Rightarrow$ might affect performance, modifiability, and program comprehension.

A Few Common Data Structures

List A *sequential* collection where each *element* has a position. In principal: a growing/flexible array

Queue A sequential collection with add and remove at different sides $\Rightarrow$ a *FiFo* (First in, First out)

Stack A sequential collection with add and remove at the same side $\Rightarrow$ a *LiFo* (Last in, First out)

Deque A sequential collection with add and remove at both sides (Deque = Double-Ended Queue)

Set A non-ordered collection not containing the same element twice $\Rightarrow$ Trying to add `X` twice $\Rightarrow$ the second attempt is ignored

Map (or *Table*) A set of *key/value* pairs
Operations: `put(key,value), get(key) --> value`

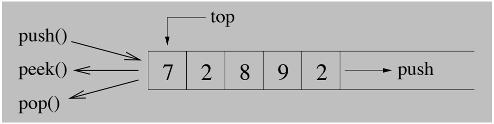
Tree Data ordered as a tree with a root (Will be presented later)
Example: The file system on your hard drive

Graph Data (nodes) with binary relations (edges)
Example: A road map with cities (nodes) and roads (edges) between them. Not part of this course.

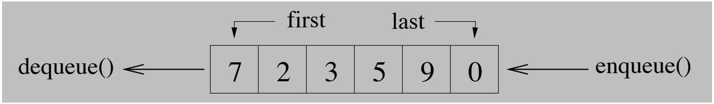
Sequential Data Structures

Sequential ⇒ a sequence where each element has a position.

Stack (Last in, first out) Add and remove at one side only.



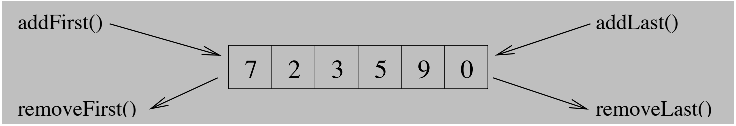
Queue (First in, first out) Add at one side, remove at the other.



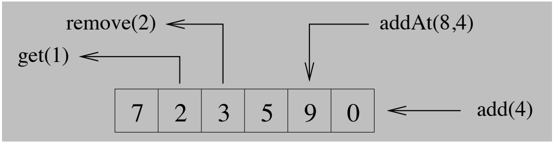
Note: Stack and Queue has a very limited set of operations
⇒ they are the most simple data structures

Deque and List

Deque (Double-ended Queue) Add and remove at both sides.



List (Add and remove everywhere)



- Note: List is the most general sequential data structure
- Q: Why not always use a list?
    - A specialized data structure can be more efficient (time, memory).
    - A specialized structure provides a more precise model
      ⇒ easier to understand for someone who reads the code

Data Structures – Definition

A data structure is defined by:

1. a name
2. the type of data that can be stored
3. a number of operation definitions

Note: What type of implementation that is used is not a part of the definition.

Example:

- Name: StringStack
- Data type: Strings
- Operations
  - push: Add a string at the top of the stack
  - pop: Return (and remove) the string at the top of the stack
  - peek: Return (without removing) the string at the top of the stack
  - size: Returns the current number of strings in the stack
  - ...

  Note: This is more of a *description* rather than a formal definition. We can use mathematics (so-called formal specifications) to properly define the semantics of each operation.

Data Structures in Java

- Interfaces (+ documentation) are often used to define a data structure

```
public interface StringStack {
    int size();                   // current stack size
    boolean isEmpty();            // true if stack is empty
    void push(String str);        // add string at top of stack
    String pop();                 // return and remove top string.
                                  // Throws StackException if stack is empty.
    String peek();                // return (without removing) top element.
                                  // Throws StackException if stack is empty.
}
```

- It provides a name (StringStack), a data type (String) and operations (pop, push, ...)
- Different data types:
  - A fix type (e.g., `String`, `int`, `Student`)
  - The `Object` type ⇒ everything can be stored, access requires down-casting
  - A generic type `<T>` ⇒ type decided when data structure created

```
ArrayList<String> names = new ArrayList<String>();
```

Implementing Sequential Data Structures

- First of all, using a fix data type (e.g., `int` or `String`) is stupid since it limits the re-usability. Much better is to use `Object` or a generic type `<T>`.
- Common implementation techniques
  1. Array based (data is stored in arrays)
  2. Linked data structures (coming soon)
  3. Using other, more general, data structures.
     (For example, using a list to implement a stack or a queue.)
- The technique used might have a large impact on how efficient (fast/memory) a data structure is in a given application.

Our example – The IntList

```
public interface IntList extends Iterable<Integer> {
    /* Add integer n to the end of the list. */
    public void add(int n);

    /* Inserts integer n at position index. Shifts the element currently at that
     * position (if any) and any subsequent elements to the right. */
    public void addAt(int n, int index) throws IndexOutOfBoundsException;

    /* Remove integer at position index. */
    public void remove(int index) throws IndexOutOfBoundsException;

    /* Get integer at position index. */
    public int get(int index) throws IndexOutOfBoundsException;

    /* Find position of integer n, otherwise return -1 */
    public int indexOf(int n);

    /* Number of integers currently stored. */
```

An array based implementation

```
public class ArrayIntList implements IntList {
    private int size = 0;    // Current size
    private int[] values;    // data storage

    public ArrayIntList() {values = new int[8];}

    public void add(int n) {
        values[size++] = n;
        if (size == values.length)  // increase size
            resize();
    }

    private void resize() { // Double the array size
        int[] tmp = new int[2*values.length];
        /* Copy from values to tmp */
        System.arraycopy(values,0,tmp,0,values.length);
        values = tmp;
    }
```

The `remove` Method

```
/* Remove integer at position index. */
public void remove(int index) throws IndexOutOfBoundsException {
    checkIndex(index, size);
    for (int i=index;i<size;i++)
        values[i] = values[i+1];    // Move one step forward
    size--;
}

/* Used by remove(), get(), and addAt() */
private void checkIndex(int index, int upper) throws IndexOutOfBoundsExcepti
    if (index < 0 || index >= upper) { // If not within range ....
        String msg = "Index = "+index+", Upper boundary = "+upper;
        throw new IndexOutOfBoundsException(msg);
    }
}
```

Using `Iterable` Classes

Implements `Iterable` ⇒ easy to iterate over all elements.

Two options:

1. Traverse content using iterators

```
IntList list = new ArrayIntList();
...    // adding integers

Iterator<Integer> it = list.iterator();
while (it.hasNext())
    System.out.println(" "+it.next());
```

2. Apply the extended `for`-statement

```
IntList list = new ArrayIntList();
...    // adding integers

for (int j : list)  // Extended for-statement
    System.out.print(" "+j);
```

Iterators and `Iterable`

- `IntList extends Iterable<Integer>`
  ⇒ All implementations must also implement the `Iterable` interface
- The `java.lang.Iterable` interface

```
public interface Iterable<T> {
    /** Returns an iterator over a set of elements of type T. */
    public Iterator<T> iterator();
}
```

- The `java.util.Iterator` interface

```
public interface Iterator<T> {

    boolean hasNext() // true if the iteration has more elements.
    T next()          // Returns the next element in the iteration .
    void remove()     // Removes the last element returned (optional !!!)
}
```

All methods are public. I have simply dropped `public` to save space.

Implementing Iterators

Inside the class ArrayIntList:

```
/* Implement the Iterable<Integer> interface */
public Iterator<Integer> iterator() { return new ListIterator(); }

/* Inner class implementation of an Iterator */
private class ListIterator implements Iterator<Integer> {
    private int count = 0;
    public Integer next() {return values[count++];}

    public boolean hasNext() {return count<size;}

    public void remove() {
        throw new RuntimeException("remove() is not implemented");
    }
}
```

Why iterators?

- Easy to iterate over all elements ...
- ... without breaking encapsulation
  $\Rightarrow$ give access to data without chance to modify content.

A 10 Minute Break!

ZZZZZZZZZZZZZZZZZZZZZ

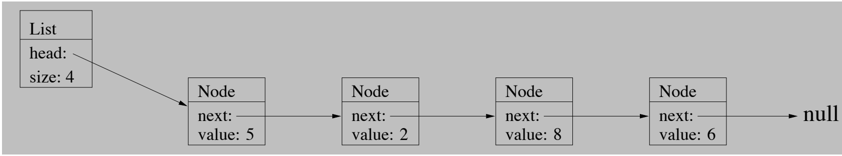
Linked Lists

Problems with array based implementations

- `remove()` $\Rightarrow$ all elements to the right must be moved one step forwards
- `addAt()` $\Rightarrow$ all elements to the right must be moved one step backwards
- A large portion of the array may not be in use $\Rightarrow$ waste of memory

Linked Lists

- Usually made up of two classes (list + node)
- The list holds a reference to the first node (the list field `head`)
- Each element is stored in a node (the node field `value`)
- Each node knows its predecessor node (the node field `next`)
- A user interacts with the list, nodes are encapsulated within the list class



A Linked Implementation

```
public class LinkedIntList implements IntList {
    private int size = 0;          // Current size
    private Node head = null;      // First node/element

    public void add(int n) {
        if (head == null)          // Add first element
            head = new Node(n);
        else {
            Node node = head;
            while (node.next != null)  // Find last node
                node = node.next;
            node.next = new Node(n);  // Attach new node
        }
        size++;
    }

    private class Node { // Private inner Node class
        int value;
        Node next = null;

        Node(int v) { value = v;}
```

get(int index) and indexOf(int n)

```
/* Find node at position index and return it's value. */
public int get(int index) throws IndexOutOfBoundsException {
    checkIndex(index, size);        // Exception if index outside [0, size-1]
    Node node = head;
    for (int i=0;i<index;i++)       // Move to index node
        node = node.next;
    return node.value;
}

/* Return index of first node with value == n */
public int indexOf(int n) {
    Node node = head;
    int index = 0;
    while (node != null) {
        if (node.value == n)
            return index;
        index++;
        node = node.next;
    }
    return -1;          // Or raise an exception?
}
```

Implementing `Iterable<Integer>`

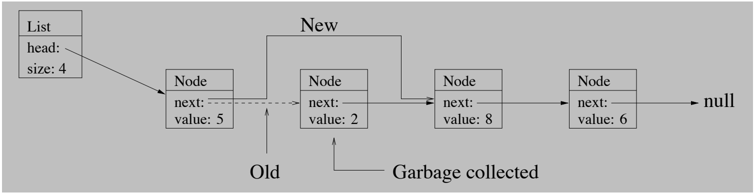
```
public Iterator<Integer> iterator() {   return new ListIterator(); }

class ListIterator implements Iterator<Integer> { // Inner iterator class
    private Node node = head; // First node is head
    public Integer next() {
        int val = node.value;          // Read current value
        node = node.next;              // Move one step ahead
        return val;
    }

    public boolean hasNext() {return node != null;}
    public void remove() { throw new RuntimeException("remove() is not implemented"); }
}
```

Note: Iteration using a for-statement and `get()` is very slow

```
    for (int i=0;i<list.size(); i++) {
        int n = list.get(i);       // starts from the head each time
        ....
    }
```

That is: always use iterators to traverse a linked list.

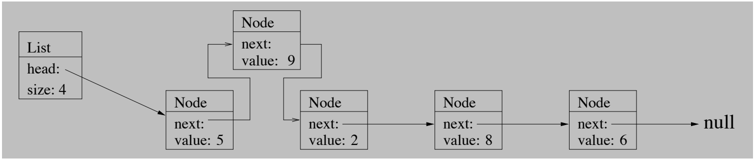
The `remove(int index)` Method



Note: We can't move backwards ⇒ we must operate from the node *before* the node we want to remove.

```
public void remove(int index) throws IndexOutOfBoundsException {
    ...   // handle case index = 0;

    Node before = head;           /* Find node before index */
    for (int i=0;i<index-1;i++)
        before = before.next;

    Node delete = before.next;
    before.next = delete.next;   // Bypass deleted node
    size--;
```

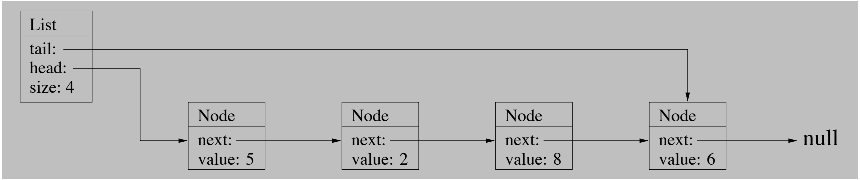
The Method `addAt(int n, int index)`




```
Node nw = new Node(n);
if (index == 0) { // Add first
    nw.next = head;
    head = nw;
}
else {
    Node before = head;      // Find node before index
    for (int i=0;i<index-1;i++)
        before = before.next;

    nw.next = before.next;   // Insert new after "before"
    before.next = nw;
}
size++;
```

Variant 1: Head and tail

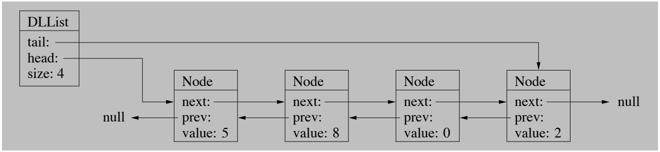
Problem: `add()` ⇒ step through the whole list ⇒ very slow
(Serious since `add()` is a frequently used operation.)
Solution: Equip the list with an additional field `Node tail` that always references the last node.




```
public void add(int n) {
    if (head == null)      // Add first element
        head = new Node(n);
        tail = head;
    else {
        tail.next = new Node(n);  // Attach new node
        tail = tail.next   // Update tail
    }
```

Variant 2: Double-linked List

Problem: We can only traverse list in one direction
⇒ (for example) printing the list content backwards is very slow
Solution: Each node has a field `Node prev` that references the previous node.




```
public class DoubleLinkedIntList implements IntList {
    private int size = 0;
    private Node head = null;
    private Node tail = null;
    ...                                   // Method defintions

    private class Node { // Private inner Node class
        int value;
        Node next = null;
        Node prev = null;
        Node(int v) { value = v;}
```

Linked vs Array-based Implementations

- We don't have to shuffle elements in `remove()`, `addAt()`
  (Repeated `remove(0)`, `addAt( ,0)` ⇒ array based is very slow.)
- We don't need the `resize` mechanism to increase the storage capacity
- No half-empty arrays that waste memory
- But ... the `get()` method is slow in the linked version since you always start from the head node.
- Linked approaches are suitable for stacks and queues where the `head/tail` approach gives direct access to the entry points
- The Java Class Library contains
  1. `java.util.ArrayList` (array based)
  2. `java.util.LinkedList` (double linked)
  3. `java.util.Vector` (array based, thread safe)
- Both version implement the `java.util.List` interface
- If in doubt, use `ArrayList`, it is faster in most scenarios.
  Never use `Vector` unless you have a threaded application.

Selecting Data Structure

Before deciding upon which data structure to use, consider:

1. What operations are required?
   $\Rightarrow$ decides what type of data structure to use (e.g., list, queue, or set)
2. What operations are most frequently used?
   $\Rightarrow$ decides what implementation to use (e.g. array based or linked)

Notice:

- The Java Class Library contains a number of data structures
  $\Rightarrow$ In most cases, we don't have to implement our own data structures.
- To select the best one from the Java Class Library
  $\Rightarrow$ knowledge about their strength and weakness is required.

Exercise 1: A Linked Queue

Provide a linked implementation of the following Queue interface:

```
public interface Queue {
    int size();                      // current queue size
    boolean isEmpty();               // true if queue is empty
    void enqueue(Object element);    // add element at end of queue
    Object dequeue();                // return and remove first element.
    Object first();                  // return (without removing) first e
    Object last();                   // return (without removing) last el
    String toString();            // string representation of list conten
    Iterator iterator();             // element iterator
}
```

The iterator traverses all elements currently in the queue. Illegal operations on an empty queue (e.g., `last()`) should generate an exception. You are not allowed to use any of the classes in the Java Class Library in this assignment.
Hint: Use the approach with `head` and `tail`.

Different Kinds of Documentation

```
public class LinkedIntList implements IntList {
    private int size = 0;        // End-of-line Comment
    private Node head = null;

    /**
     * Javadoc Comment
     * Appends integer <code>n</code> at the end of the list.
     *
     * @param n integer to be added.
     */
    public void add(int n) {

        ...
    }

    /*
        Multi-line Comment
     */
    private class Node { //

        ...
    }
```

Javadoc Comments

- Javadoc comments are used to document program code
- Javadoc comment: `\** ... *\`,
  common block comments: `\* ... *\`
- Javadoc has a number of *tags* with special meaning.

```
/**
* Get integer at position <code>index</code>.
*
* @param     <code>index</code>, position of element to be returned.
* @return    element at position <code>index</code>
* @throws    IndexOutOfBoundsException if <code>index</code>
*            outside current range <code>[0, size]</code>.
*/
public int get(int index) throws IndexOutOfBoundsException;
```

- HTML tags can be used on text. E.g. `<b>important</b>` that it is...
- **Why Javadoc?**
  - Javadoc ⇒ can generate HTML pages with information.
  - The Java API documentation is a fine example.

Generating HTML Documentation

- Generate HTML with Eclipse: `Project` → `Generate Javadoc`
- Eclipse uses the program `javadoc.exe` that comes with all Java JDK installations.
- Eclipse may need configurations to find `javadoc.exe`.
- A large number of pages are generated: Start page, package structure, ...
- ...but also one page per class: `IntList.java` → `IntList.html`
- **Notice:** One page will be generated for every chosen class, even if they don't contain Javadoc comments.

Method Comments

- Position is important. Must be right above the method.
- The tags `@param`, `@return` and `@throws` are used for important information.

```
/**
 * Inserts integer <code>n</code> at position index. Shifts the element currently at that
 * position (if any) and any subsequent elements to the right.
 *
 * @param     n integer to be added.
 * @param     index position where <code>n</code> should be added.
 * @throws    IndexOutOfBoundsException if <code>index</code>
 *            outside current range <code>[0, size]</code>.
 */
public void addAt(int n, int index) throws IndexOutOfBoundsException;
```

- **Notice:**
  - Text can contain ordinary HTML tags.
  - The first sentence (`Inserts integer ...`) is used in `Method Summary`.
  - All sentences are used in the more detailed `Method Details`.

Class Comments

- Usually between `import` and class declaration.

```
...
package linked;
/**
 * An interface representing a simple integer list. It provides
 * support for accessing (add, remove, get) at an arbitrary position
 * in the list.
 * <p/>
 * Currently available <code>IntList</code> implementations in the
 * <code>linked</code> package are:
 * <ul>
 * <li> {@link linked.LinkedIntList} </li>
 * <li> {@link linked.ArrayIntList} </li>
 * </ul>
 *
 * @author Jonas Lundberg
 * @see java.util.List
 * @since 2006-11-06
 */
public interface IntList extends Iterable<Integer> {
  ...
```

Getting started with Javadoc

We recommend:

1. Download a few classes that use Javadoc comments.
   (For example IntList.java and ArrayIntList.java in my Java Examples)
2. Use Eclipse to generate Javadoc comments.
3. Learn more:
   - Study downloaded examples.
   - Google: `Javadoc + Tutorial`

You will be asked to add Javadoc comments to the Queue in Exercise 1.