Fundamental Data Structures – Part 2

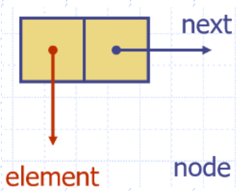
- **Circularly Linked Lists**
  - Round-Robin Scheduling
  - Designing and Implementing a Circularly Linked List
- **Equivalence Testing**
  - Equivalence Testing with Arrays
  - Equivalence Testing with Linked Lists
- **Cloning Data Structures**
  - Cloning Arrays
  - Cloning Linked Lists

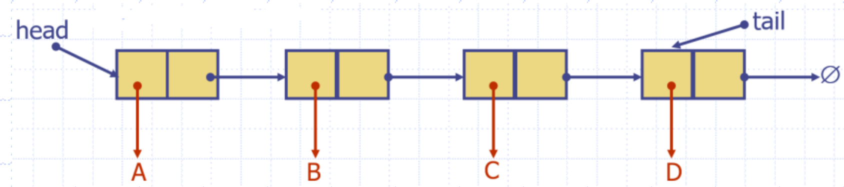
Lesson 2 Review

- **Data structures**
  - A way of organizing data
  - Leads to efficiency in insertions, deletions, searches, etc.
- **Arrays**
  - Fixed-size data structure
  - Use index to refer to array elements
  - length property
  - To add an entry into an array board at index i, we need to make room for it by shifting forward the $n - i$ entries $\boldsymbol{board}[i]$, …, $\boldsymbol{board}[n - 1]$
  - To remove the entry at index i, we need to fill the hole left by e by shifting backward the $n - i - 1$ elements $\boldsymbol{board}[i + 1]$, …, $\boldsymbol{board}[n - 1]$

Lesson 2 Review - Singly Linked List

- Data structure consisting of a sequence of nodes starting from a head node:





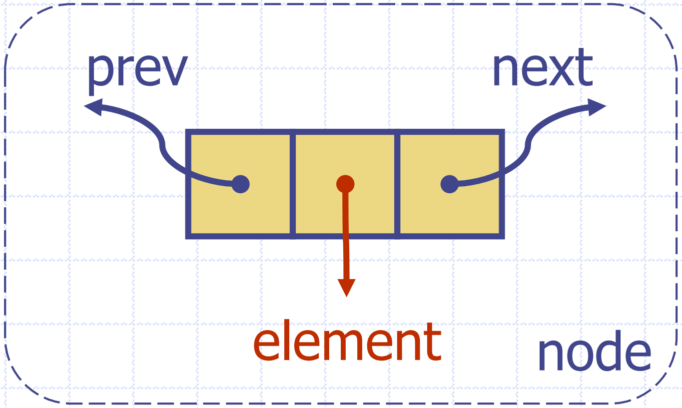
  - Each node contains the **element** (data) and a **link** to the next node
  - Self-referential **Node** class represents a node of singly linked list.
- **Insertion** of a new node **at the head - efficient**
  - Have new node point to old head node
  - Update head to point to new node
- **Inserting** a new node **at the tail - efficient**
  - Have new node point to null
  - Have old last node point to new node
  - Update tail to point to new node
- **Removing at the head - efficient**
  - Update head to point to next node in the list

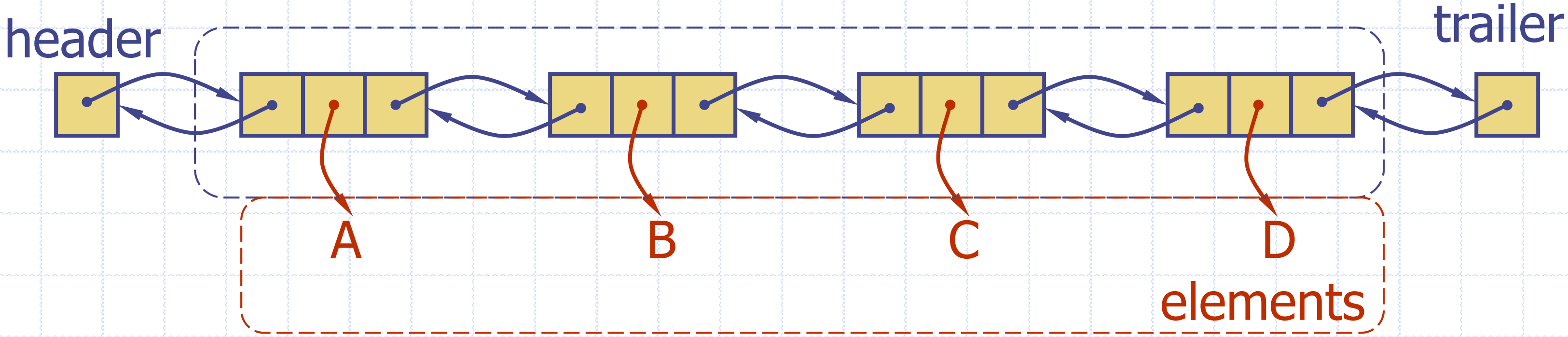
Lesson 2 Review - Singly Linked List

- **Removing at the tail** of a singly linked list
  - Need a loop to reach the last node
  - is **not efficient** - there is no constant-time way to update the tail to point to the previous node

Lesson 2 Review - Doubly Linked List

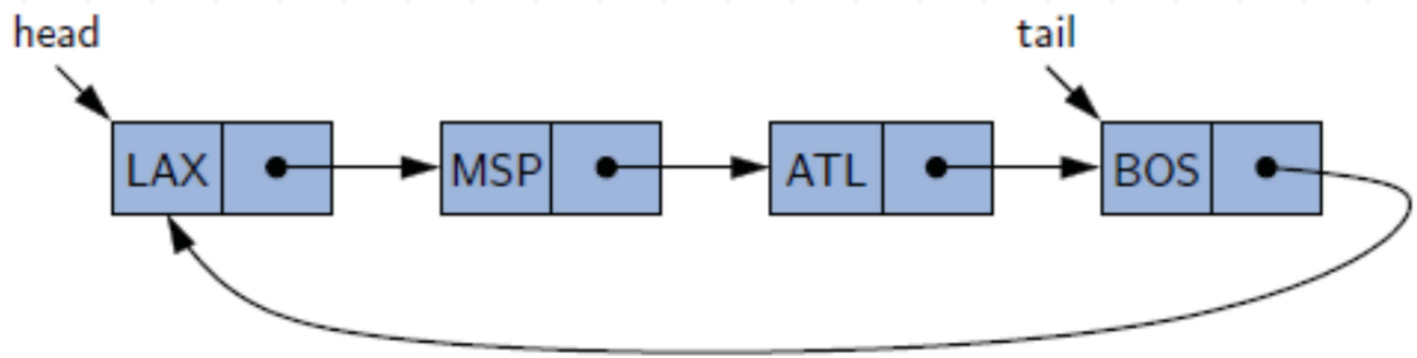
- A doubly linked list can be traversed forward and backward
- Nodes store:
  - element
  - link to the previous node
  - link to the next node
- Special **trailer** and **header** nodes
- Fast insertions and deletions between nodes





Presentation for use with the textbook Data Structures and Algorithms in Java, 6th edition, by M. T. Goodrich, R. Tamassia, and M. H. Goldwasser, Wiley, 2014

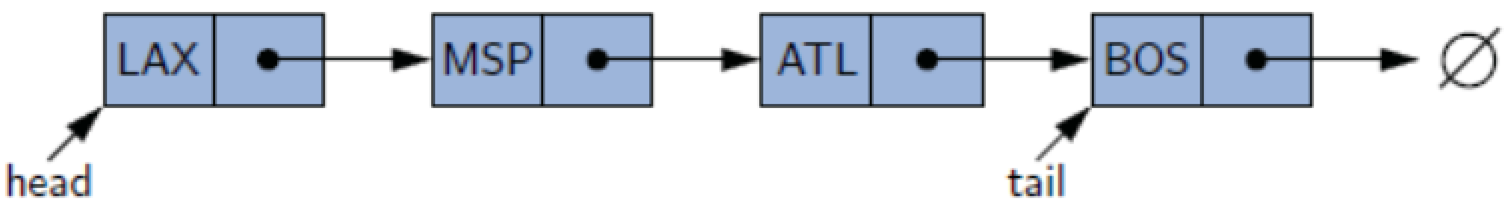
Circularly Linked Lists



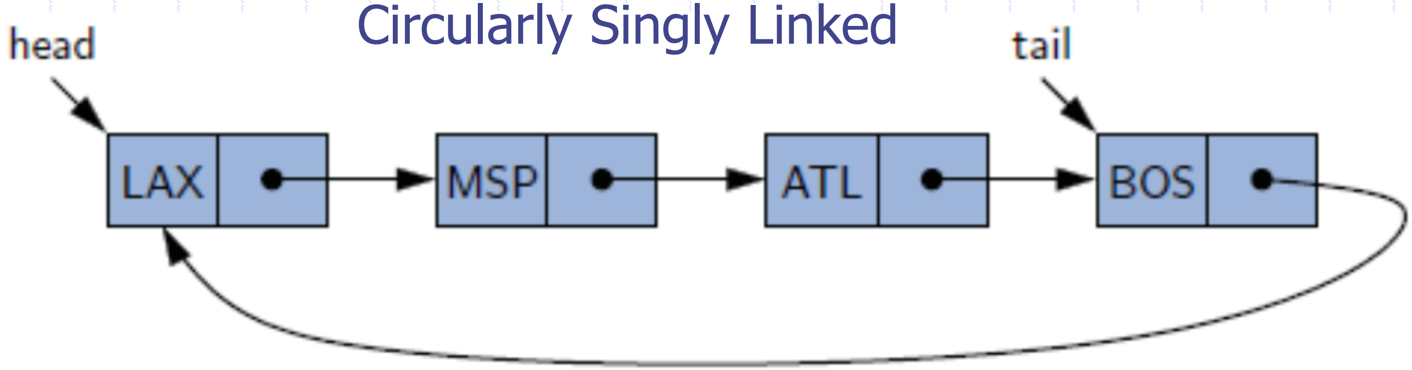
Circularly Linked List

- A circularly linked list, is essentially a singularly linked list in which the **next reference of the tail node is set to refer back to the head of the list** (rather than null)

Regular Singly Linked List



Circularly Singly Linked



Circularly Linked List

- In many applications data can be more naturally viewed as having a **cyclic order**, with well-defined neighboring relationships, but no fixed beginning or end.
- Examples:
  - many **multiplayer games are turn-based**, with player A taking a turn, then player B, then player C, and so on, but eventually **back to player A** again, and player B again, with the pattern repeating.
  - **city buses and subways often run on a continuous loop**, making stops in a scheduled order, but with no designated first or last stop per se.

Round-Robin Scheduling

- **Operating Systems** manage multiple processes and schedule for them CPU time.
- Most operating systems allow processes to effectively share use of the CPUs, using some form of an algorithm known as ***round-robin scheduling***.
  - A process is given a short turn to execute, known as a ***time slice***, but it is interrupted when the slice ends, even if its job is not yet complete.
  - Each active process is given its own time slice, **taking turns in a cyclic order**.
  - New processes can be added to the system, and processes that complete their work can be removed.

Round-Robin Scheduling

- A **round-robin scheduler** could be implemented with a traditional linked list, by repeatedly performing the following steps on **linked list *L*** (see Figure 3.15):
  1. process *p* = ***L*.removeFirst( )**
  2. Give a time slice to process *p*
  3. *L*.**addLast(*p*)**

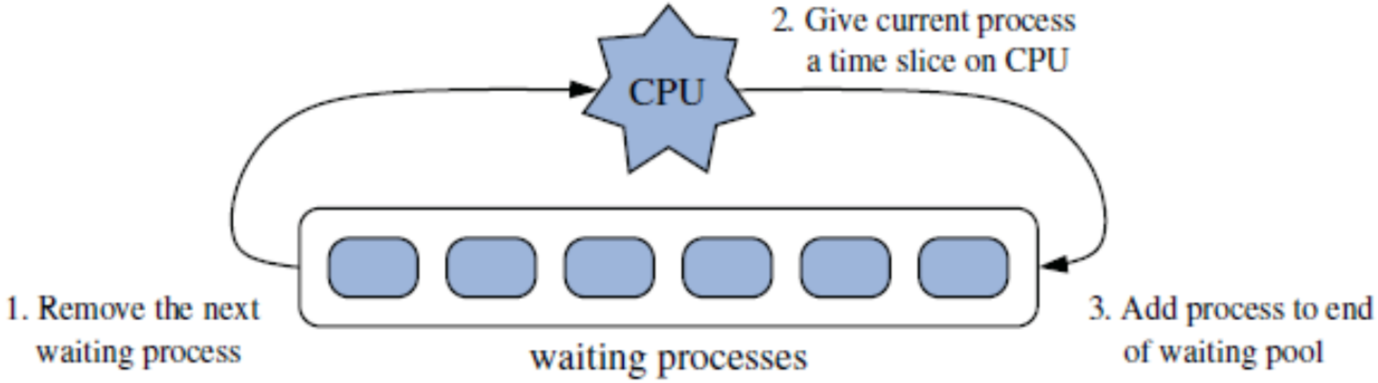


Figure 3.15: The three iterative steps for round-robin scheduling.

Designing a Circularly Linked List

- There are drawbacks to the use of a traditional linked list for implementing a round-robin scheduler.
  - inefficient to **repeatedly throw away a node** from one end of the list, only to create a new node for the same element when **reinserting it**.
  - inefficient to perform various updates to **decrement and increment the list's size** and to **unlink and relink nodes**.
- A **circularly linked list** is more efficient.
  - the **next** reference of the **tail** node is **set to refer back to the head** of the list (rather than null)

Designing a Circularly Linked List

- Add an additional update method **rotate( )**:
  - moves the first element to the end of the list (referenced by tail).
- With this new operation, round-robin scheduling can be efficiently implemented by repeatedly performing the following steps on a circularly linked list *C*:

  1. **p** = *C*.first( )

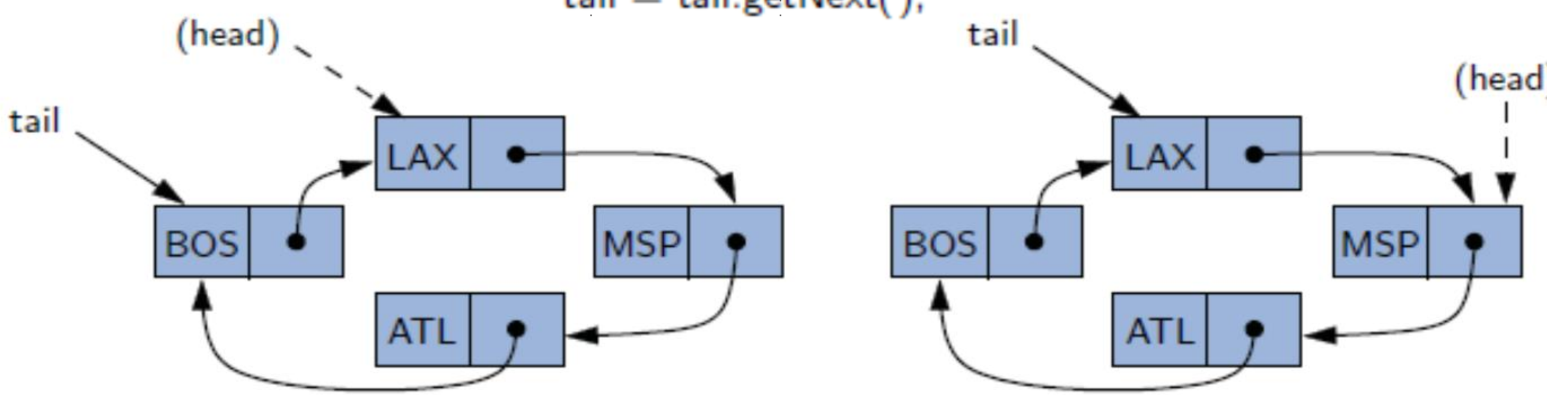
  2. Give a time slice to process **p**

  2. *C*.rotate( )
- one additional optimization - we **no longer explicitly maintain the head reference**.
- So long as we maintain **a reference to the tail**, we can **locate the head as tail.getNext( )**.

Operations on a Circularly Linked List

- Implementing the new rotate method is quite trivial.
- We do not move any nodes or elements, we simply **advance the tail reference to point to the node that follows it** (the implicit head of the list).

tail = tail.getNext( );



(a) Before the rotation (b) After the rotation

Inserting at the Head

- We can **add a new element at the front** of the list by creating a new node and linking it just *after* the tail of the list, as shown in Figure 3.18.

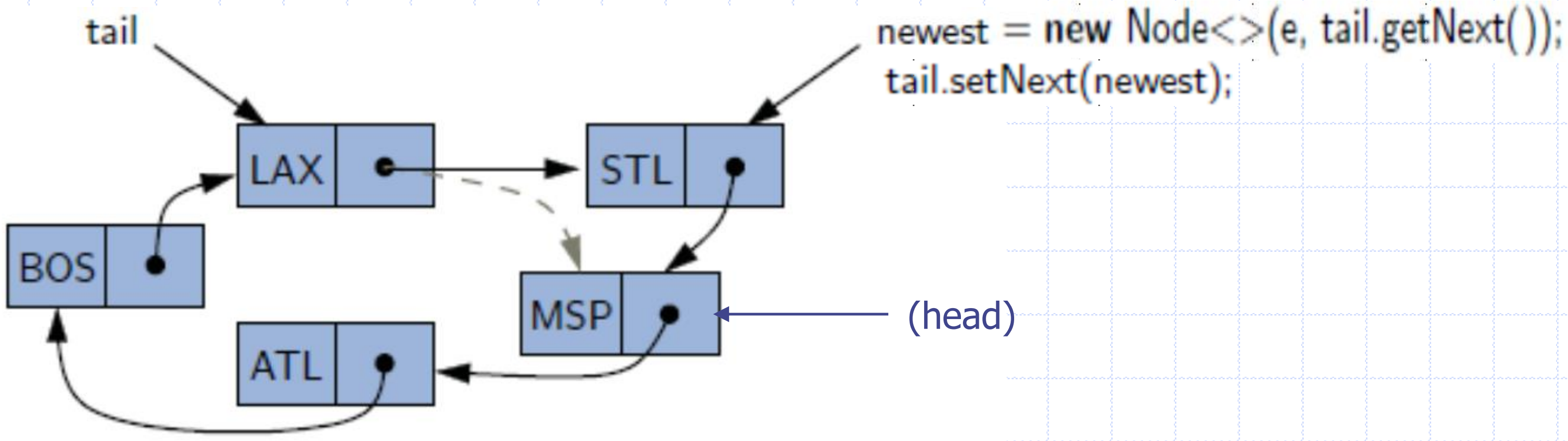
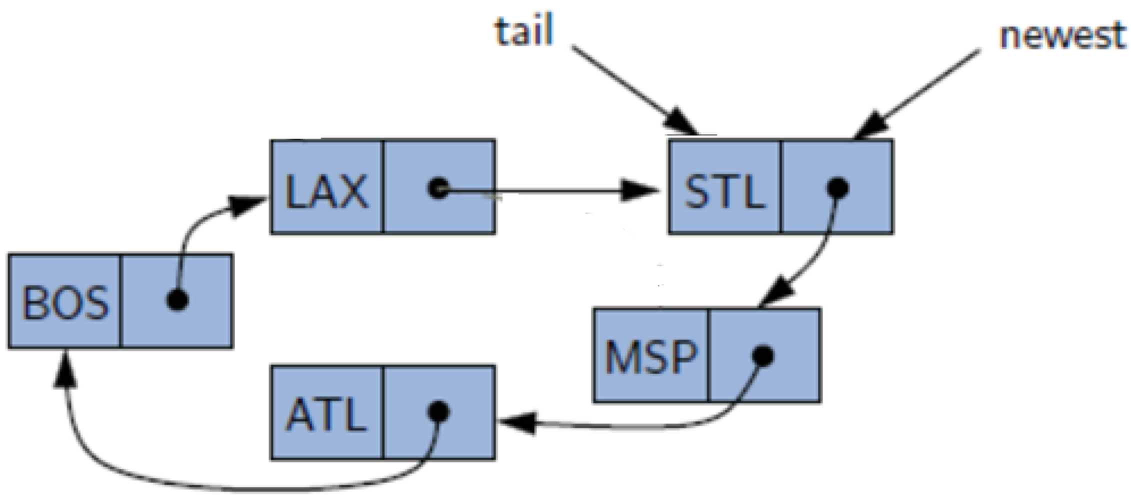


Fig. 3.18

Inserting at the Tail

- To implement the addLast method, we can rely on the use of a call to addFirst and then immediately advance the tail reference so that **the newest node becomes the last.**

```
addFirst(e);
tail = tail.getNext( );
```

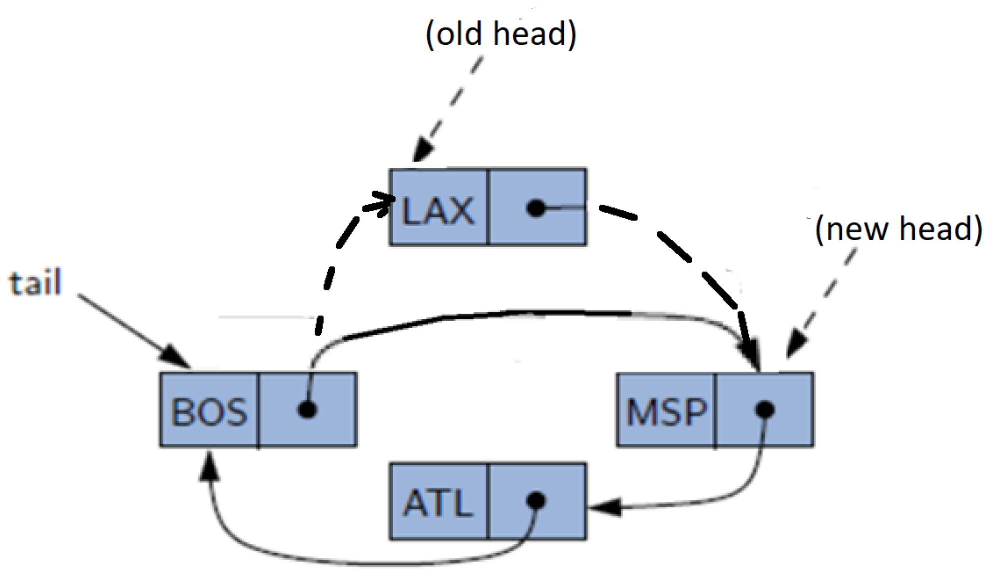


Java Methods

```
// update methods
public void rotate( ) {              // rotate the first element to the back of the list
  if (tail != null)                        // if empty, do nothing
    tail = tail.getNext( );                // the old head becomes the new tail
}
public void addFirst(E e) {                // adds element e to the front of the list
  if (size == 0) {
    tail = new Node<>(e, null);
    tail.setNext(tail);                    // link to itself circularly
  } else {
    Node<E> newest = new Node<>(e, tail.getNext( ));
    tail.setNext(newest);
  }
  size++;
}
public void addLast(E e) {                 // adds element e to the end of the list
  addFirst(e);                             // insert new element at front of list
  tail = tail.getNext( );                  // now new element becomes the tail
}
```

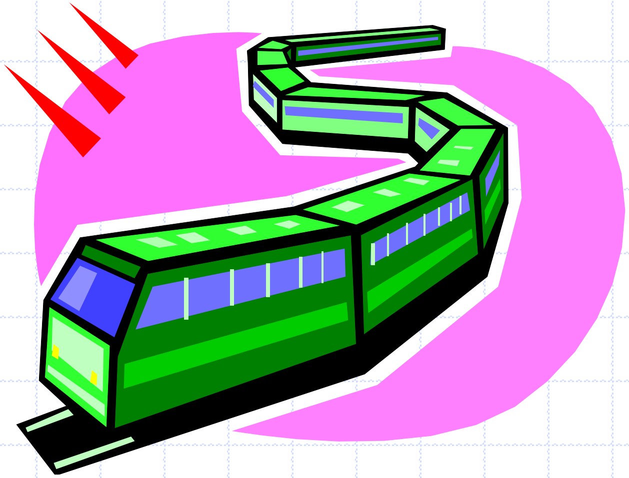
Java Methods – removal at the head

```
public E removeFirst( ) {                  // removes and returns the first element
  if (isEmpty( )) return null;             // nothing to remove
  Node<E> head = tail.getNext( );
  if (head == tail) tail = null;           // must be the only node left
  else tail.setNext(head.getNext( ));      // removes "head" from the list
  size−−;
  return head.getElement( );
}
```



Presentation for use with the textbook Data Structures and Algorithms in Java, 6th edition, by M. T. Goodrich, R. Tamassia, and M. H. Goldwasser, Wiley, 2014

Equivalence Testing



Equivalence Testing

- When working with reference types, there are many different notions of **what it means for one expression to be equal to another**.
- At the lowest level, if a and b are reference variables, then expression **a == b tests whether a and b refer to the same object** (or if both are set to the null value).
- However, for many types there is a higher-level notion of two variables being considered “equivalent” even if they do not actually refer to the same instance of the class.
- For example, we typically want to consider **two String instances to be equivalent to each other if they represent the identical sequence of characters**.

Equivalence Testing

- To support a broader notion of equivalence, all object types support a method named **equals**.
- Users of reference types should rely on the syntax **a.equals(b)**, unless they have a specific need to test the more narrow notion of identity.
- The **equals method is formally defined in the Object class**, which serves as a superclass for all reference types, but **that implementation reverts to returning the value of expression a == b**.
- Defining a more meaningful notion of equivalence requires knowledge about a class and its representation.

Equivalence Testing

- The consistency of Java's libraries depends upon the **equals** method defining what is known as an ***equivalence relation*** in mathematics, satisfying the following properties:
  - Treatment of null: For any nonnull reference variable x, the call **x.equals(null)** should return false (that is, nothing equals null except null).
  - Reflexivity: For any nonnull reference variable x, the call **x.equals(x)** should return true (that is, an object should equal itself).
  - Symmetry: For any nonnull reference variables x and y, the calls **x.equals(y)** and **y.equals(x)** should return the same value.
  - Transitivity: For any nonnull reference variables x, y, and z, if both calls **x.equals(y)** and **y.equals(z)** return true, then call **x.equals(z)** must return true as well.

Equivalence Testing with Arrays

- The following provides a summary of the treatment of equivalence for arrays, assuming that variables a and b refer to array objects:
  - **a == b**: Tests if a and b refer to the same underlying array instance.
  - **a.equals(b)**: Interestingly, this is identical to a == b.
    - Arrays are not a true class type and do not override the **Object.equals** method.

Equivalence Testing with Arrays

- **Arrays.equals(a,b)**: This provides a more intuitive notion of equivalence, returning true if the arrays have the same length and **all pairs of corresponding elements are "equal" to each other**.
  - More specifically, if the array elements are primitives, then it uses the standard **==** to compare values.
  - If elements of the arrays are a reference type, then it makes pairwise comparisons **a[k].equals(b[k])** in evaluating the equivalence.

Equivalence Testing with Arrays

- For most applications, the **Arrays.equals** behavior captures the appropriate notion of equivalence.
- However, there is an additional complication when using multidimensional arrays.
  - The fact that two-dimensional arrays in Java are really one-dimensional arrays nested inside a common one-dimensional array raises an interesting issue with respect to how we think about ***compound objects***, which are objects—like a two-dimensional array—that are made up of other objects.
  - In particular, it brings up the question of where a compound object begins and ends.

Multidimensional Arrays

Java Multidimensional Array (2d

programiz.com/java-programming/multidimensional-array

As mentioned, each component of array `a` is an array in itself, and length of each rows is also different.

	Column 1	Column 2	Column 3	Column 4
Row 1	1 a[0][0]	2 a[0][1]	3 a[0][2]	
Row 2	4 a[1][0]	5 a[1][1]	6 a[1][2]	9 a[1][3]
Row 3	7 a[2][0]			

Let's write a program to prove it.

```
class MultidimensionalArray {
    public static void main(String[] args) {

        int[][] a = {
                {1, 2, 3},
                {4, 5, 6, 9},
                {7},
        };

        System.out.println("Length of row 1: " + a[0].length);
        System.out.println("Length of row 2: " + a[1].length);
        System.out.println("Length of row 3: " + a[2].length);
    }
}
```


https://www.programiz.com/java-programming/multidimensional-array

Equivalence Testing with Arrays

- To support the more natural notion of multidimensional arrays being equal if they have equal contents, the class provides an additional method:
- **Arrays.deepEquals(a,b)**: Identical to Arrays.equals(a,b) except when the elements of a and b are themselves arrays, in which case it calls Arrays.deepEquals(a[k],b[k]) for corresponding entries, rather than a[k].equals(b[k]).

Equivalence Testing Linked Lists

- Using a definition very similar to the treatment of arrays by the java.util.Arrays.equals method, we consider **two lists to be equivalent if they have the same length and contents that are element-by-element equivalent**.
- We can evaluate such equivalence by **simultaneously traversing two lists, verifying that x.equals(y) for each pair of corresponding elements x and y**.

Equivalence Testing Linked Lists

- The implementation of the SinglyLinkedList.equals method is given in Code Fragment 3.19.

```
public boolean equals(Object o) {
  if (o == null) return false;
  if (getClass( ) != o.getClass( )) return false;
  SinglyLinkedList other = (SinglyLinkedList) o;        // use nonparameterized type
  if (size != other.size) return false;
  Node walkA = head;                                    // traverse the primary list
  Node walkB = other.head;                              // traverse the secondary list
  while (walkA != null) {
    if (!walkA.getElement( ).equals(walkB.getElement( ))) return false; //mismatch
    walkA = walkA.getNext( );
    walkB = walkB.getNext( );
  }
  return true;    // if we reach this, everything matched successfully
}
```

Code Fragment 3.19: Implementation of the SinglyLinkedList.equals method.

Cloning Data Structures

- The universal **Object** superclass defines a method named **clone**, which can be used to produce what is known as a ***shallow copy*** of an object.
- This uses the standard assignment semantics to **assign the value of each field of the new object equal to the corresponding field of the existing object that is being copied**.
- The reason this is known as a shallow copy is because if the field is a reference type, then an initialization of the form **duplicate.field = original.field** causes the field of the new object to refer to the same underlying instance as the field of the original object.

Cloning Data Structures

- A shallow copy is not always appropriate for all classes, and therefore, **Java intentionally disables use of the clone( )** method by declaring it as **protected**, and by having it throw a CloneNotSupportedException when called.
- The author of a class must explicitly **declare support for cloning by formally declaring that the class implements the Cloneable interface**, and by declaring a public version of the clone( ) method.
- That public method can simply call the protected one to do the field-by-field assignment that results in a shallow copy, if appropriate.
- However, for many classes, **the class may choose to implement a deeper version of cloning**, in which some of the referenced objects are themselves cloned.

Cloning Arrays

- Although arrays support some special syntaxes such as a[k] and a.length, it is important to remember that they are objects, and that array variables are reference variables.
- This has important consequences. As a first example, consider the following code:

```
int[ ] data = {2, 3, 5, 7, 11, 13, 17, 19};
int[ ] backup;
backup = data; // warning; not a copy
```

- The assignment of variable backup to data does not create any new array; it simply **creates a new alias for the same array**, as portrayed in Figure 3.23

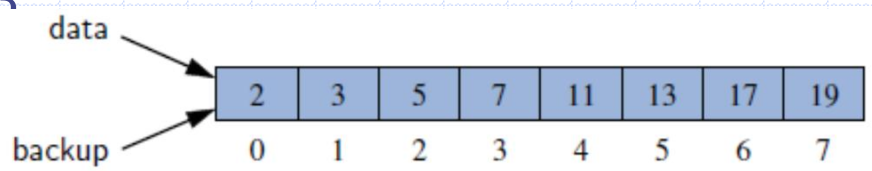


Figure 3.23: The result of the command backup = data for **int** arrays.

Cloning Arrays

- Instead, if we want to make a copy of the array, data, and assign a reference to the new array to variable, backup, we should write:

  backup = data.clone( );
- The clone method, when executed on an array, initializes each cell of the new array to the value that is stored in the corresponding cell of the original array.
- This results in an independent array, as shown in Figure 3.24.
- If we subsequently make an assignment such as data[4] = 23 in this configuration, the backup array is unaffected.

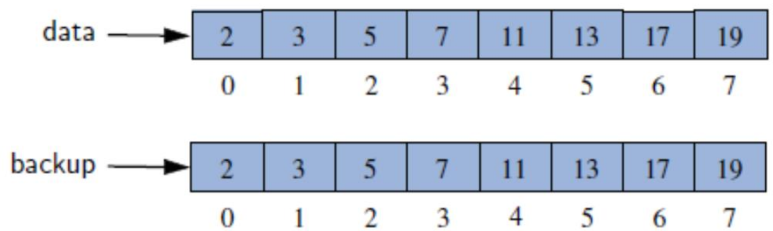


Figure 3.24: The result of the command backup = data.clone( ) for **int** arrays.

Cloning Arrays

- There are more considerations when copying an array that stores reference types rather than primitive types.
- The clone( ) method produces a ***shallow copy*** of the array, producing a new array whose **cells refer to the same objects referenced by the first array**.
- For example, if the variable *contacts* refers to an array of hypothetical *Person* instances, the result of the command:

guests = contacts.clone( ) produces a shallow copy, as portrayed in Figure 3.25.

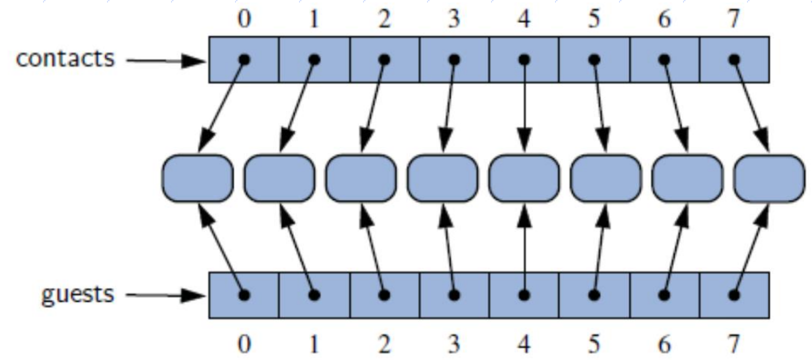


Figure 3.25: A shallow copy of an array of objects, resulting from the command guests = contacts.clone( ).

Cloning Arrays

- A ***deep copy*** of the contact list can be created by **iteratively cloning the individual elements**, as follows, but only if the Person class is declared as Cloneable.

```
Person[ ] guests = new Person[contacts.length];
for (int k=0; k < contacts.length; k++)
  guests[k] = (Person) contacts[k].clone( ); // returns Object type
```

- Because a two-dimensional array is really a one-dimensional array storing other one-dimensional arrays, the same distinction between a shallow and deep copy exists.
- Unfortunately, the java.util.Arrays class does not provide any "deepClone" method.

Cloning Arrays

- However, we can **implement our own method by cloning the individual rows of an array**, as shown in Code Fragment 3.20, for a two-dimensional array of integers.

```
public static int[ ][ ] deepClone(int[ ][ ] original)
{
    int[ ][ ] backup = new int[original.length][ ]; // create top-level array of arrays
    for (int k=0; k < original.length; k++)
        backup[k] = original[k].clone( ); // copy row k
    return backup;
}
```

Code Fragment 3.20: A method for creating a deep copy of a two-dimensional array of integers.

Cloning Linked Lists

- Let’s add support for cloning instances of the SinglyLinkedList class from previous lecture.
- The first step to making a class cloneable in Java is declaring that it implements the **Cloneable** interface.
- Therefore, we adjust the first line of the class definition to appear as follows:

```java
public class SinglyLinkedList<E> implements Cloneable {
```

- The remaining task is implementing a public version of the clone() method of the class, which we present in Code Fragment 3.21.

Cloning Linked Lists

```
public SinglyLinkedList<E> clone() throws CloneNotSupportedException {
  // always use inherited Object.clone() to create the initial copy
  SinglyLinkedList<E> other = (SinglyLinkedList<E>) super.clone(); // safe cast
  if (size > 0) {                          // we need independent chain of nodes
    other.head = new Node<>(head.getElement(), null);
    Node<E> walk = head.getNext();         // walk through remainder of original list
    Node<E> otherTail = other.head;        // remember most recently created node
    while (walk != null) {                 // make a new node storing same element
      Node<E> newest = new Node<>(walk.getElement(), null);
      otherTail.setNext(newest);           // link previous node to this one
      otherTail = newest;
      walk = walk.getNext();
    }
      other.tail = otherTail;
  }
  return other;
}
```

Code Fragment 3.21: Implementation of the SinglyLinkedList.clone method.

Cloning Linked Lists

- By convention, that method should begin by creating a new instance using a call to **super.clone( )**, which in our case invokes the method from the Object class (line 3).
- Because the inherited version returns an Object, we perform a narrowing cast to type **SinglyLinkedList<E>**.
- At this point in the execution, the other list has been created as a **shallow copy of the original**.
- Since our list class has two fields, *size* and *head*, the following
- assignments have been made:
  - other.size = this.size;
  - other.head = this.head;
- We create a new head node at line 5 of the code, and then perform a walk through the remainder of the original list (lines 8–13) while creating and linking new nodes for the new list.