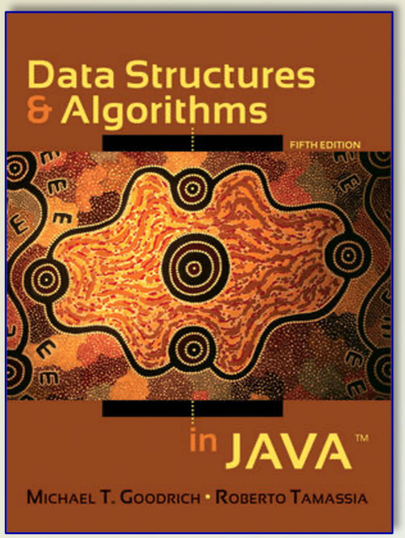
Data Structure & Algorithms in JAVA

5th edition

Michael T. Goodrich

Roberto Tamassia



Chapter 3: Indices, Nodes, and Recursion

CPSC 3200

Algorithm Analysis and Advanced Data Structure

Chapter Topics

- Using arrays.
- Singly Linked Lists.
- Doubly Linked Lists.
- Circularly Linked Lists and Linked-List Sorting.
- Recursion.

Data types vs. Data Structures

- A ***data type*** is a well-defined collection of data with a well-defined set of operations on it. (**Abstract**)

- A ***data structure*** is an actual implementation of a particular abstract data type.

- Abstract data types add clarity by separating the definitions from the implementations.

- **Example:** The abstract data type Set has operations such as:

  EmptySet(S), Insert(x,S), Delete(x,S), etc.

Data Types in Java

- Java is ***strongly-typed***, which means that all variables must first be declared before they can be used.

- A collection of values along with a set of operations that can be performed on those values. (the definition of a class).

- Java has a large library of classes that have been written for us to use.
- This includes many ***data structures***.

- The classes we write when we program can be considered user defined types.

Classes as data types:

- It can be said that an **interface** for a class is actually the data type.
- If we are to make this distinction, then the class can be thought of as the implementation of the data type.

Number Types (Primitives)

- A ***primitive type*** is predefined by the language and is named by a reserved keyword. Primitive values do not share state with other primitive values.
- These are the exception to the Object/Class rule we discussed; they are not objects created from a class.
- Primitives are stored on the Stack where they can be referenced by the Stack Pointer in the CPU's register. This is very efficient.
- Objects are created on the Heap in the computer's memory with the **new** keyword. Then, we use a reference to access the Object.
  - For a small, simple variable this is not very efficient.

Primitive Types Cont…

- The eight primitive data types supported by the Java programming language are:

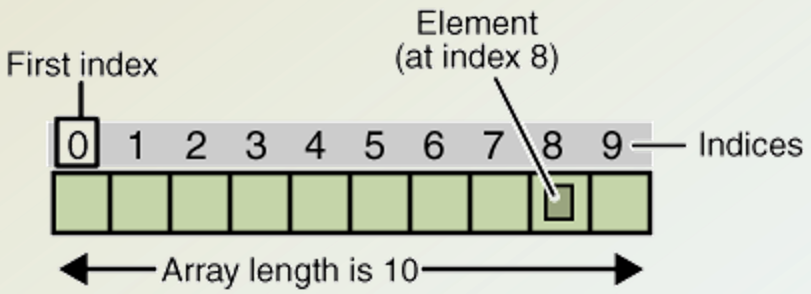
  - **boolean** (True or False)
  - **char** (16 bit, Unicode 0 to Unicode 2^{16} - 1)
  - **byte** (8 bit, -128 to 128)
  - **short** (16 bit, -2^{15} to 2^{15} - 1)
  - **int** (32 bit, -2^{31} to 2^{31} - 1)
  - **long** (64 bit, -2^{63} to 2^{63} - 1)
  - **float** (IEEE-754)
  - **double** (IEEE-754)

  - void

Data Structures

- Data Structures are a ‘composite’ data type.
- This means they can be decomposed.
    - Into other composite types.
    - Finally, into a data type.

- Data structures are collections of elements.
    - The most well known is the array.



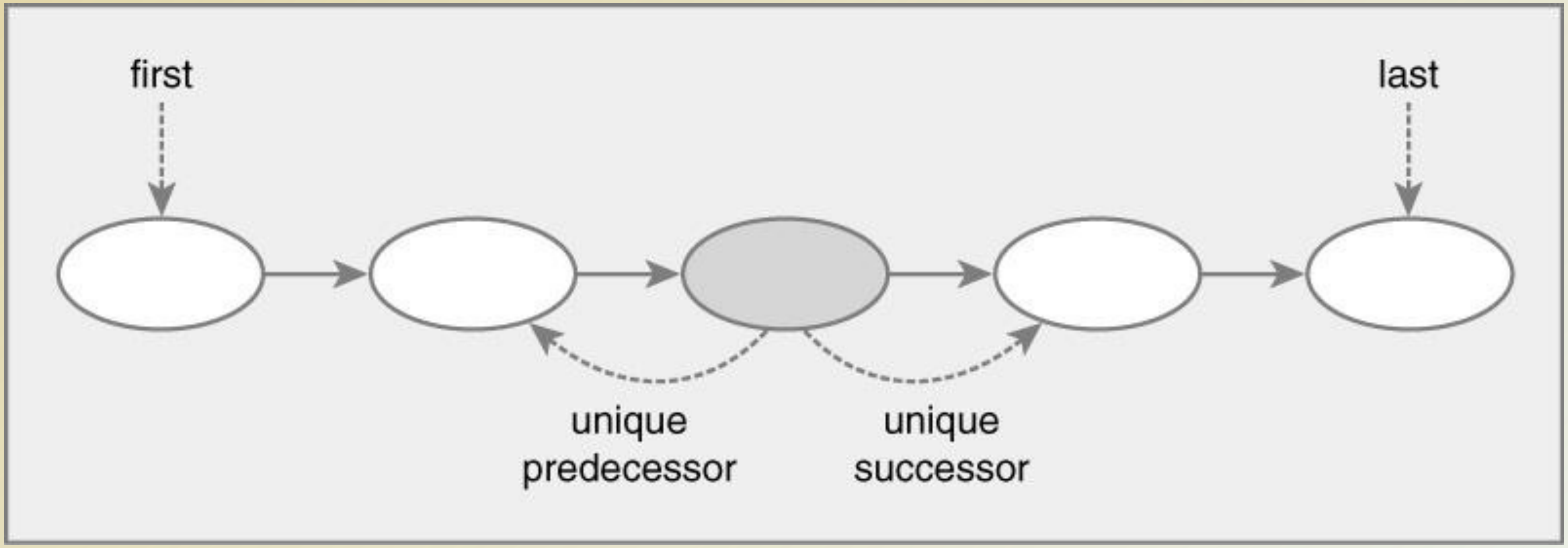
Classes of composite Data Structures

- Linear Data Structures.
- Hierarchical Data Structures.
- Graph Data Structures.
- Sets.

Linear Data Structures

- A data structure is said to be ***linear*** if its elements form a sequence or a linear list.
  - Arrays
  - Linked Lists
  - Stacks, Queues

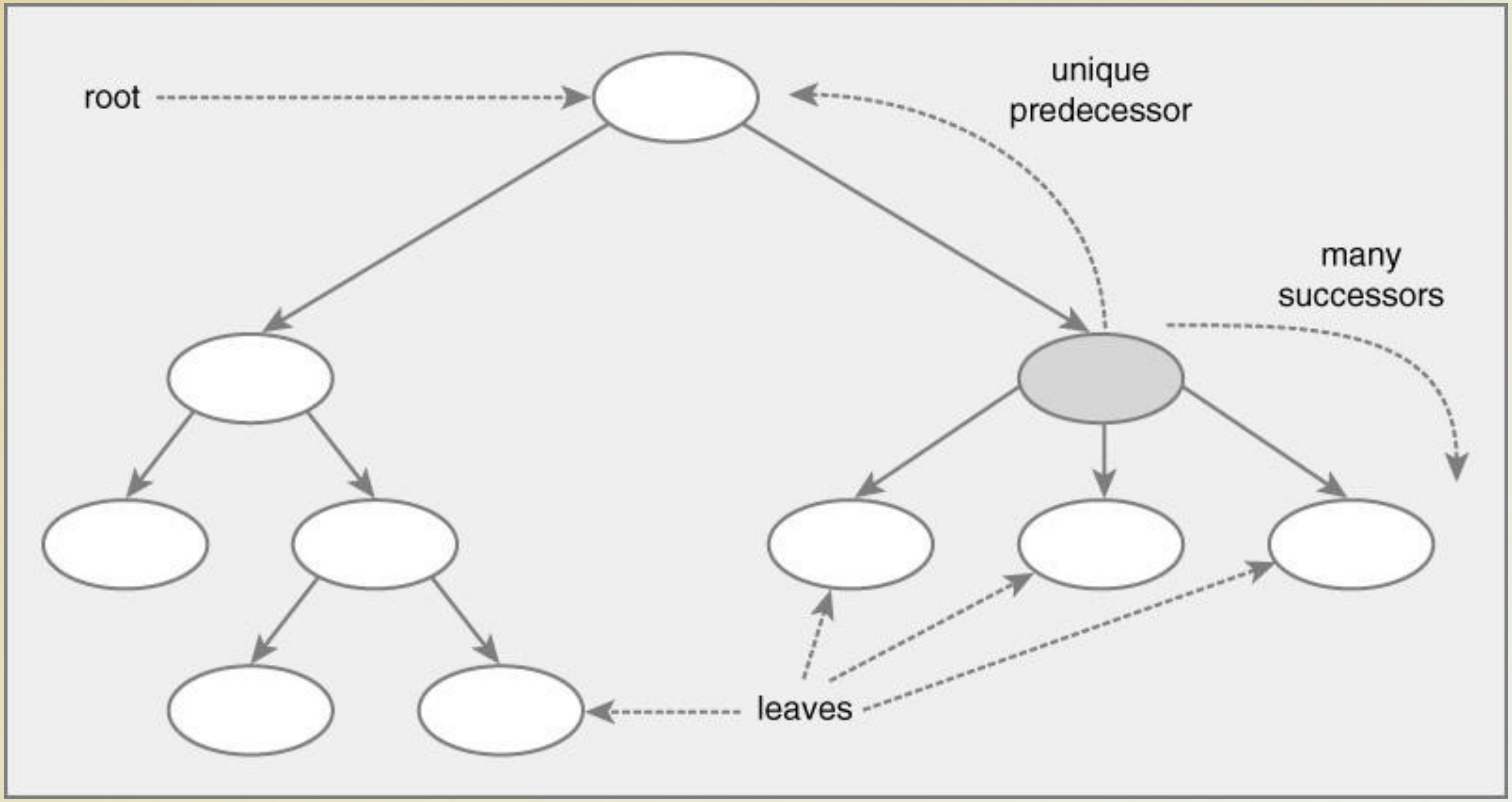
- A *one:one* relationship between elements in the collection.
  - Assuming the structure is not empty, there is a first and a last element.
  - Every element except the first has a unique *predecessor*.
  - Every element except the last has a unique *successor*.



General model of a linear data structure

Hierarchical Data Structures

- Hierarchical Data Structures
  - A *one:many* relationship between elements in the collection.
    - Assuming the structure is not empty, there is a unique element called the *root*.
    - There may be zero to many terminating nodes called *leaves*.
    - Nodes that are neither roots nor leaves are called *internal*.



General model of a hierarchical data structure

Hierarchical Data Structures cont …

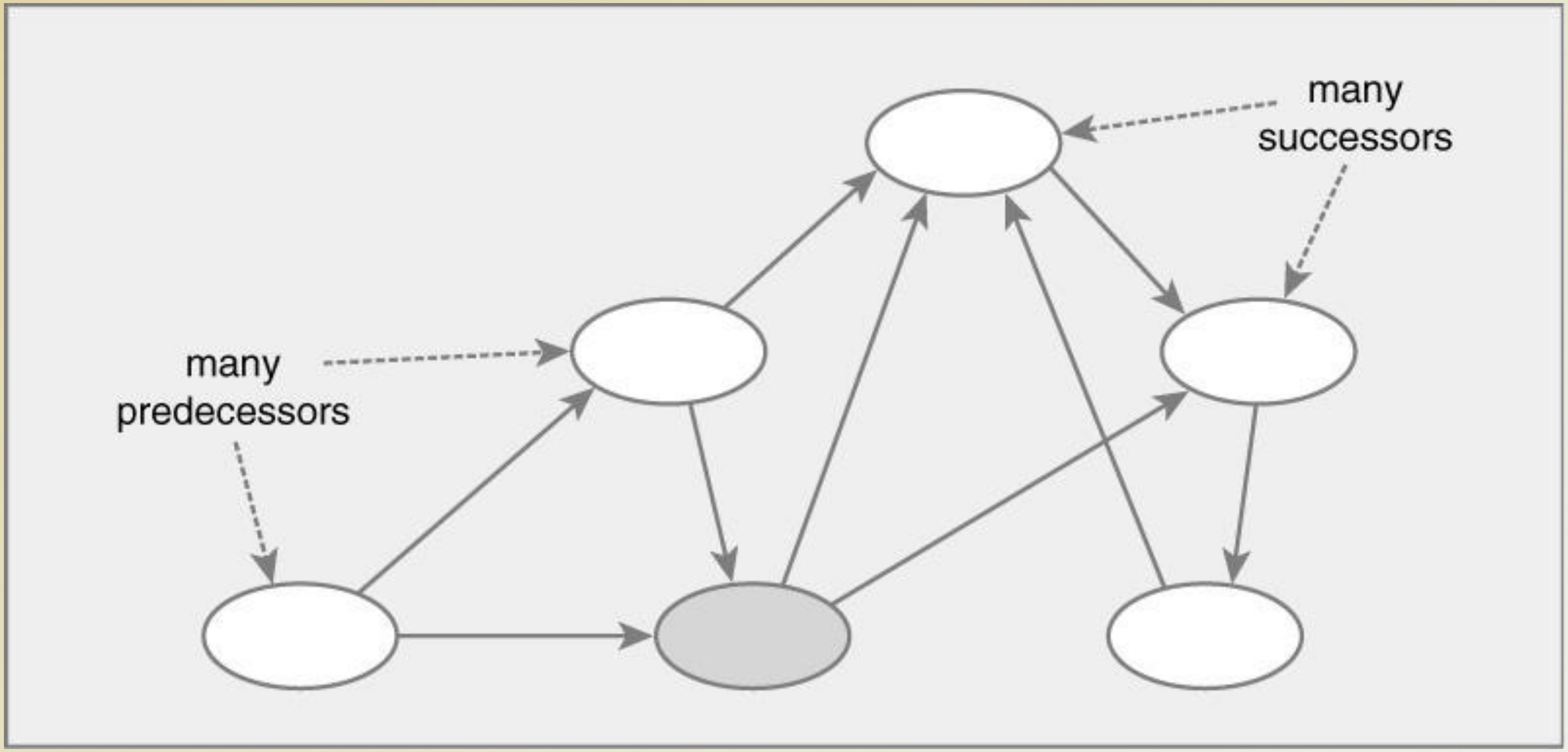
- Every element except the root has a unique *predecessor*.
- Every element except a leaf has a one or more *successors*.
- An internal node has exactly one predecessor and one or more successors.
- There is more than one way to traverse a hierarchical data structure.

Hierarchical Data Structures cont…

- Generally called *trees*, they are very important and widely used.
- A few types are; Generalized Trees, Binary Trees, Binary Search Trees, AVL Trees (balanced binary search trees), Splay Trees, B Trees, & P Trees.
- Similar to linear data types, the basic structure is the same. Each version has different rules and operations.

Graph Data Structures

- A *many:many* relationship between elements in the collection.
- An element (***E***) in graph can be connected arbitrarily to any other element in the graph, (including itself).
- Conversely, any number of elements can be connected to ***E***.



General model of a graph data structure

Linear Data Structures

- Traversal through a liner data structure is called ***iteration***.
- The basic structures are the same.
- The operations and restrictions are different.

Operations on Linear Data Structure

- **Traversal**: Travel through the data structure.
- **Search**: Traversal through the data structure for a given element.
- **Insertion**: Adding new elements to the data structure.
- **Deletion**: Removing an element from the data structure.
- **Sorting**: Arranging the elements in some type of order.
- **Merging**: Combining two similar data structures into one.

What is an array?

- **Fixed-size** data collection.
- All of the elements of an array are of the same **same data-type** (either primitives or objects).
- Its elements have **indexes** ranging from **0** to **n-1**

  n is `<Array name>.length`
- In Java, Arrays are objects.

What is an array?

- All Java arrays are technically **one-dimensional**.
- Two-dimensional arrays are arrays of arrays.
- Declaring an array **does not** create an array object or allocate space in memory; it creates a variable with a reference to an array.
- Array variable declarations must indicate a dimension by using **[ ]**

Array access and memory allocation

- One of the principal reasons why arrays are used so widely is that their elements can be accessed in constant time O(1).
- It takes the same time to access **a[1]** or **a[10]**.
- The address of **a[x]** can be determined arithmetically by adding a suitable offset to the machine **address of the head of the array**.
- The elements of the array are stored in a **contiguous** block of memory. This has advantages and disadvantages !?

Array Declaration/ one-dimension

- Array declaration has two components: **array's type** and the **array's name**.

```
Syntax: ArrayType[ ] arrayName;

int[ ] anArray;//declares an array of integers
```

- **type** is the data type of the contained elements.

- The size of the array is not part of its type .

```
    String[5] s; // illegal declaration
    String[ ] s; // one-dimensional array
    String[ ][ ] s; // two-dimensional array
```

Array Construction

- To construct an array we use the **`new`** keyword

```
class-name[ ] array-name =
                          new class-name[capacity];
```

or

```
primitive-type[ ] array-name =
                          new primitive-type[capacity];
```

- *primitive-type* and *class-name* specify the type of element stored in the array.
- *array-name* is any valid Java identifier that refers to all elements in the array.
- *Capacity (size)* is an integer expression representing the maximum number of elements that can be stored into the array.

Declaring and Constructing an Array

```
int[] anArray; // declares an array of integers

myIntArray = new int[10]; //construct the array
myHamburgerArray = new Hamburger[13];
```

- Since arrays are allocated at runtime, we can use a variable to set their dimension

```
int arrSize = 100;
String[] myArray = new String[arrSize];
```

or

- both declare and construct

```
String[] myStringArray = new String[5];
```

Initializing an Array

- Several ways – here's one that declares, creates and initializes all in one:

```
String[] fears = {“spiders”, “heights”,
                        “bridges”, “flying”};

char[] grades = { 'A', 'B', 'C', 'D', 'E' };
```

Note: (1) curly brackets
(2) comma-separated list

Array Examples

```
// declare and initialize
   int[] courseMarks = new int[8];
// populate
   courseMarks[0] = 89;
   courseMarks[1] = 56;
   courseMarks[2] = 24;
       …
```

- values for rest is **0**
- for objects[ ] the value will be **null**

Accessing Individual Elements in the Array

- Individual array elements are referenced through subscripts of this form:

```
array-name [ int-expression ]
```

- **int-expression** is an integer that should be in the range of **0..Size-1**.

MyArray[n] refers to the element of the array n (starting from 0)

```
<array name>.length
```

length is a built-in property to determine the size of any array

- First element is **myArray[0]**
- Last element is **myArray[length.myArray-1]**

Out of Bounds/Range

- If you attempt to access an array element beyond the largest index, a run time error will occur and the program will terminate is the error is not handled.

Subscripts can get "out of range"

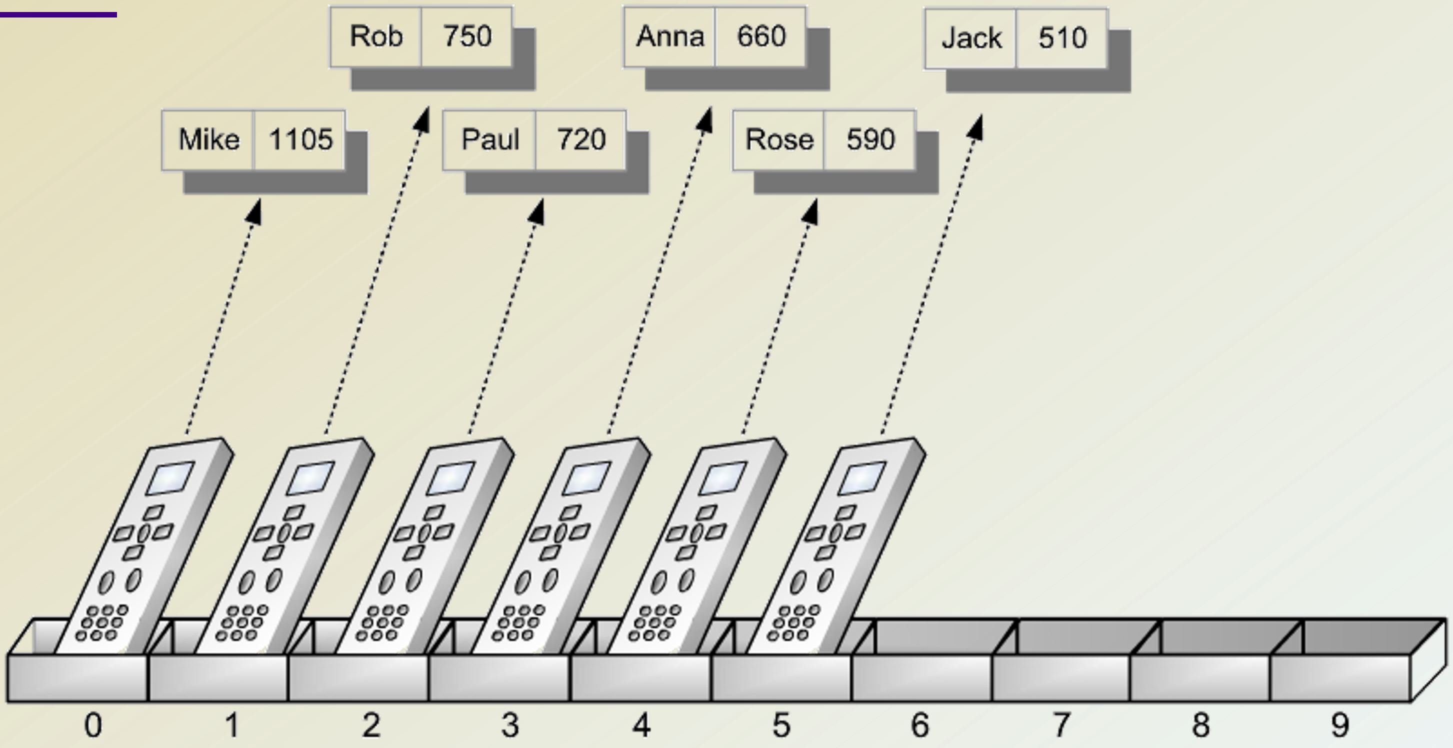
```
String[] name = new String[1000];
name[-1] = "Subscript too low";
name[ 0] = "This should be the first name";
name[999] = "This is the last good subscript";
name[1000] = "Subscript too high";
```

Two of the above references will cause **ArrayIndexOutOfBounds** exceptions.

Storing Game Entries in an Array

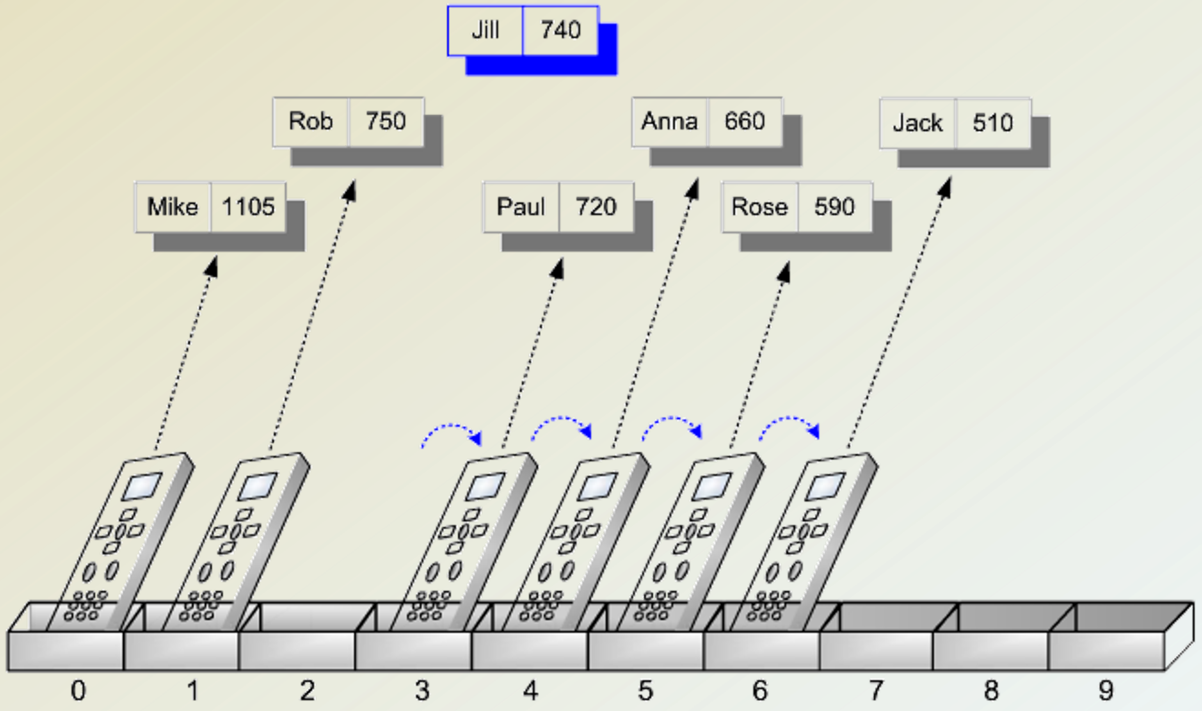
GameEntry.java

Scores.java



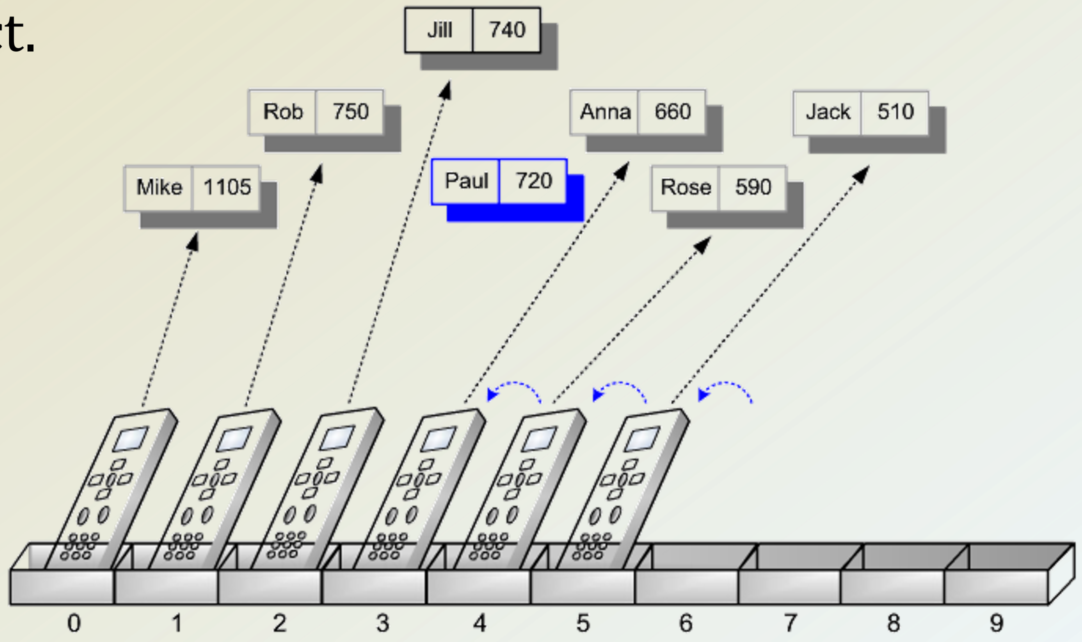
Storing Game - Insertion

add(e): Insert game entry ***e*** into the collection of high scores. If the collection is full, then ***e*** is added only if its score is higher than the lowest score in the set, and in this case, ***e*** replaces the entry with the lowest score.



Storing Game – Object Removal

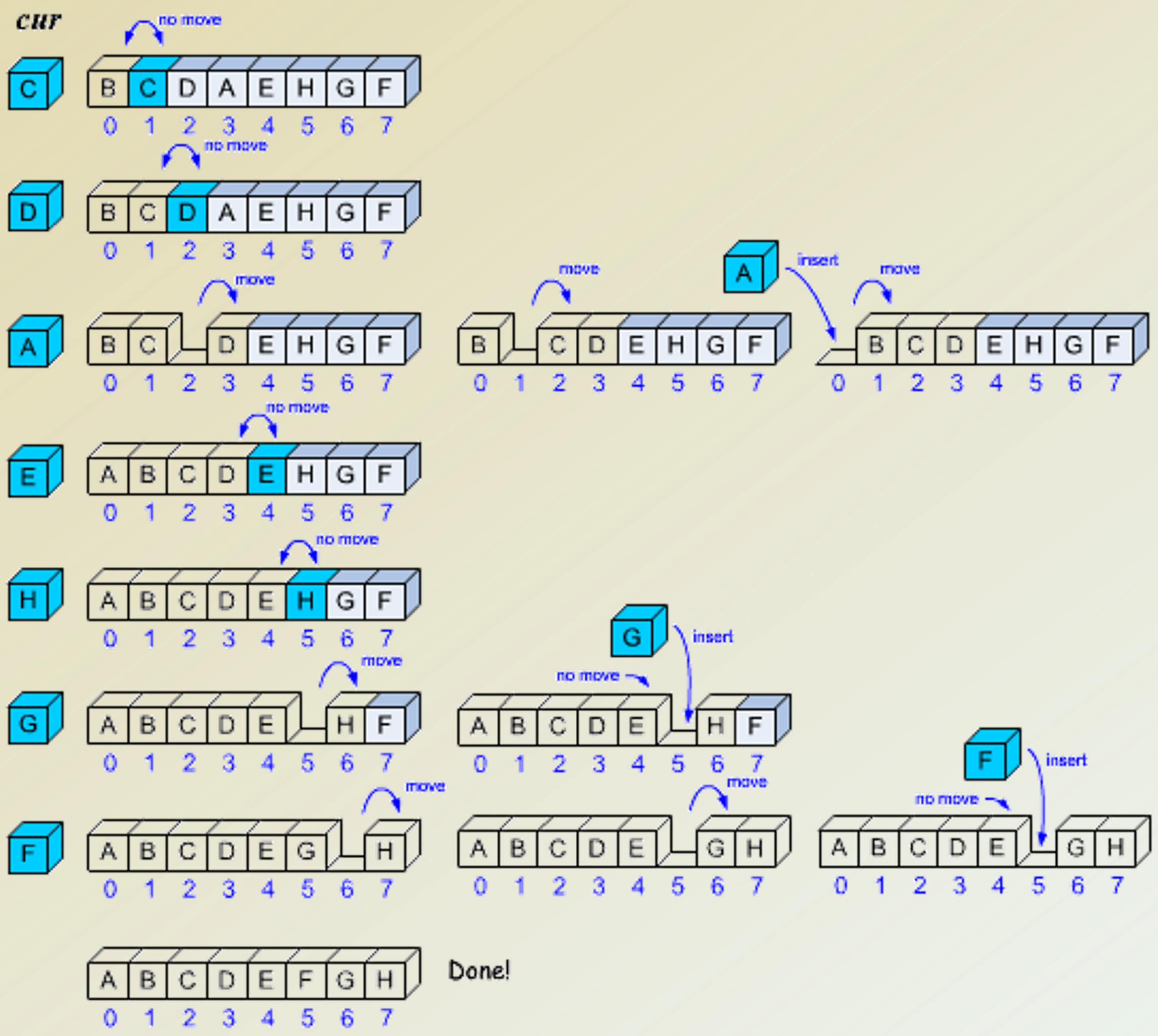
remove(i): Remove and return the game entry ***e*** at index ***i*** in the entries array. If index ***i*** is outside the bounds of the entries array, then this method throws an exception; otherwise, the entries array will be updated to remove the object at index ***i*** and all objects previously stored at indices higher than ***i*** are "moved over" to fill in for the removed object.



Sorting an Array – Insertion Sort (1/4)

```
Algorithm InsertionSort(A):
        Input:   An array A of n comparable elements
        Output: The array A with elements rearranged in non-
                decreasing order
        for i ← 1 to n−1 do
            Insert A[i] at its proper location in A[0],A[1],...,A[i−1].
```

Sorting an Array – Insertion Sort (2/4)



Sorting an Array – Insertion Sort (3/4)

```
Algorithm InsertionSort(A):
    Input:   An array A of n comparable elements
    Output: The array A with elements rearranged in
             nondecreasing order
    for i ← 1 to n−1 do
        {Insert A[i] at its proper location in A[0],A[1],...,A[i−1]}
        cur ← A[i]
        j ← i−1
        while j ≥ 0 and a[j] > cur do
            A[j+1] ← A[j]
            j ← j−1
        A[j+1] ← cur {cur is now in the right place}
```

Sorting an Array – Insertion Sort (4/4)

```
/** Insertion sort of an array of characters into non-decreasing
order */
public static  void insertionSort(char[] a)
{
     int n = a.length;
     for (int i = 1; i < n; i++)
     {
         // index from the second character in a
         char cur = a[i]; // the current character to be inserted
         int j = i - 1;

         // start comparing with cell left of I
         // while a[j] is out of order with cur
         while ((j >= 0) && (a[j] > cur))
             a[j + 1] = a[j--]; // move a[j] right and decrement j

         a[j + 1]=cur;       // this is the proper place for cur
     }
}
```

java.util.Arrays

equals(A,B): Returns true if and only if the array A and the array B are equal. Two arrays are considered equal if they have the same number of elements and every corresponding pair of elements in the two arrays are equal. That is, A and B have the same elements in the same order.

fill(A,x): Stores element x into every cell of array A, provided the type of array A is defined so that it is allowed to store the value x.

http://docs.oracle.com/javase/7/docs/api/java/util/Arrays html

java.util.Arrays

copyOf(A,n): Returns an array of size ***n*** such that the first ***k*** elements of this array are copied from **A**, where ***k*** =min{n, A.length}. If ***n*** > A.length, then the last ***n***− A.length elements in this array will be padded with default values, e.g., ***0*** for an array of int and ***null*** for an array of objects.

copyOfRange(A,s,t): Returns an array of size ***t*** − ***s*** such that the elements of this array are copied in order from A[***s***] to A[***t***−1], where ***s*** < ***t***, with padding as with copyOf( ) if ***t*** > A.length.

java.util.Arrays

sort(A): Sorts the array **A** based on a natural ordering of its elements, which must be comparable. This methods uses the quick-sort algorithm discussed in Section 11.2.

toString(A): Returns a String representation of the array **A**, which is a comma-separated list of the elements of A, ordered as they appear in A, beginning with [and ending with].

The string representation of an element **A[i]** is obtained using String.valueOf(**A[i]**), which returns the string "***null***" for a null object and otherwise calls **A[i]**.toString().

java.util.Random

nextBoolean( ): Returns the next pseudo-random boolean value.

nextFloat( ): Returns the next pseudo-random float value, between 0.0 and 1.0.

nextInt( ): Returns the next pseudo-random int value.

nextInt(n): Returns the next pseudo-random int value in the range [0,n).

setSeed(s): Sets the seed of this pseudo-random number generator to the long s.

http://docs.oracle.com/javase/7/docs/api/java/util/Random.html

Example

`ArrayTest.java`

Cryptography with Character Array

- Java makes it easy for us to create string objects from character arrays and vice versa.
- To create an object of class String from a character array **A**, we simply use the expression, new String(**A**)
- For example, the string we would construct from the array **A** = [a,c,a,t] is acat.

- Given a string S, we can create a character array representation of S by using the expression, S.toCharArray( )
- For example, if we call toCharArray on the string adog, we would get the array **B** = [a, d, o, g].

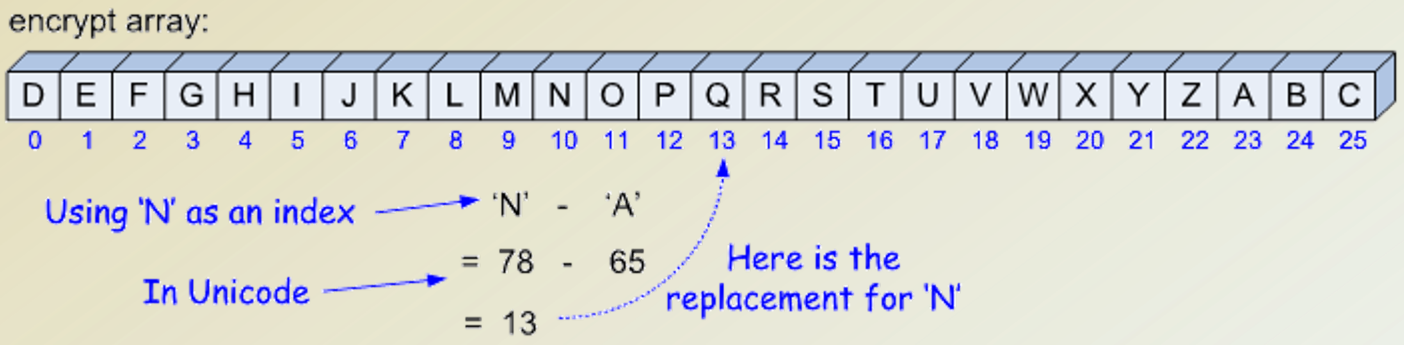
The Caesar Cipher

- The Caesar cipher involves replacing each letter in a message with the letter that is **three** letters after it in the alphabet for that language. So, in an English message, we would replace each A with D, each B with E, each C with F, and so on. We continue this approach all the way up to W, which is replaced with Z. Then, we let the substitution pattern wrap around, so that we replace X with A, Y with B, and Z with C.

The Caesar Cipher

- If we were to number our letters like array indices, so that A is 0, B is 1, C is 2, and so on, then we can write the Caesar cipher as a simple formula:

Replace each letter i with the letter (i+3) mod 26



Caesar.java

Two Dimensional Arrays

- Many computer games, be they strategy games, simulation games, or first-person conflict games, use a two-dimensional "board."
- Programs that deal with such **positional** games need a way of representing objects in a two-dimensional space.
- A natural way to do this is with a **two-dimensional array**, where we use two indices, say i and j, to refer to the cells in the array.
- The first index usually refers to a **row** number and the second to a **column** number.

Two Dimensional Arrays

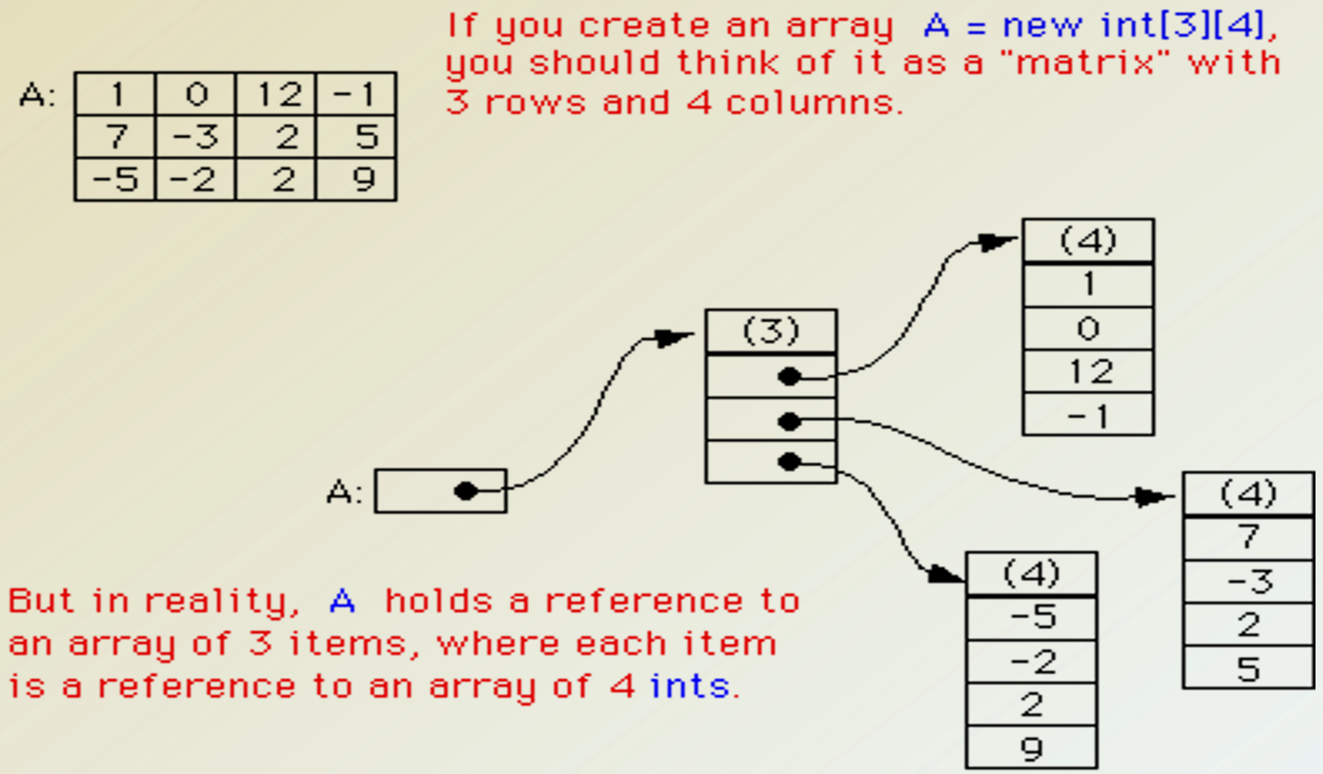
- we can create a **two-dimensional array** as an array of arrays.
- Such a two-dimensional array is sometimes also called a **matrix**.
- In Java, we declare a two-dimensional array as follows:

```
int[][] Y = new  int[8][10];


Y[i][i+1] = Y[i][i] + 3;
i = Y.length;   // i is 8
j = Y[4].length;  // j is 10
```

Declaring & Initializing Multidimensional Arrays

```
int[][] A = { { 1, 0, 12, -1 }, { 7, -3, 2, 5 }, { -5, -2, 2, 9 } };
```



If you create an array A = new int[3][4], you should think of it as a "matrix" with 3 rows and 4 columns.

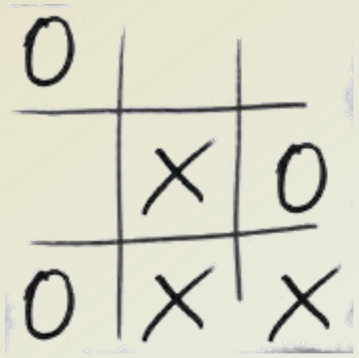
A:

1	0	12	-1
7	-3	2	5
-5	-2	2	9

But in reality, A holds a reference to an array of 3 items, where each item is a reference to an array of 4 ints.

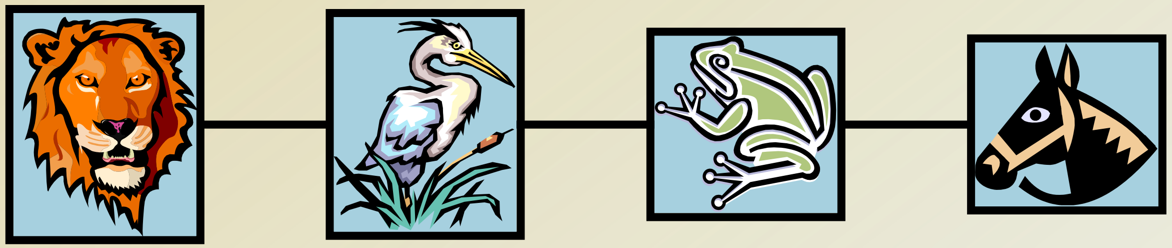
Tic-Tac-Toe

- with a **0** indicating an empty cell, a **1** indicating an X, and a **–1** indicating O



`TicTacToe.java`

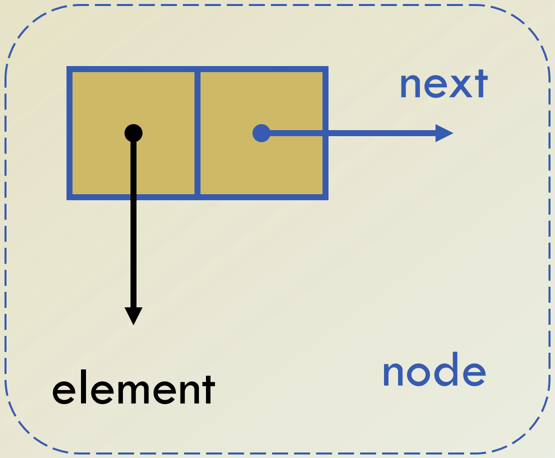
Singly Linked List



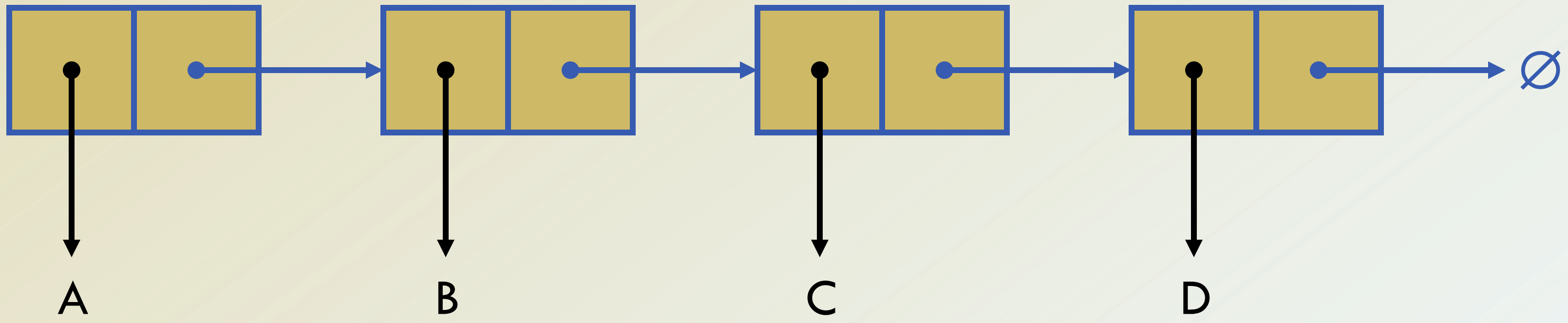
Singly Linked List

- A singly linked list is a concrete data structure consisting of a sequence of nodes.

- Each node stores
  - element.
  - link to the next node.



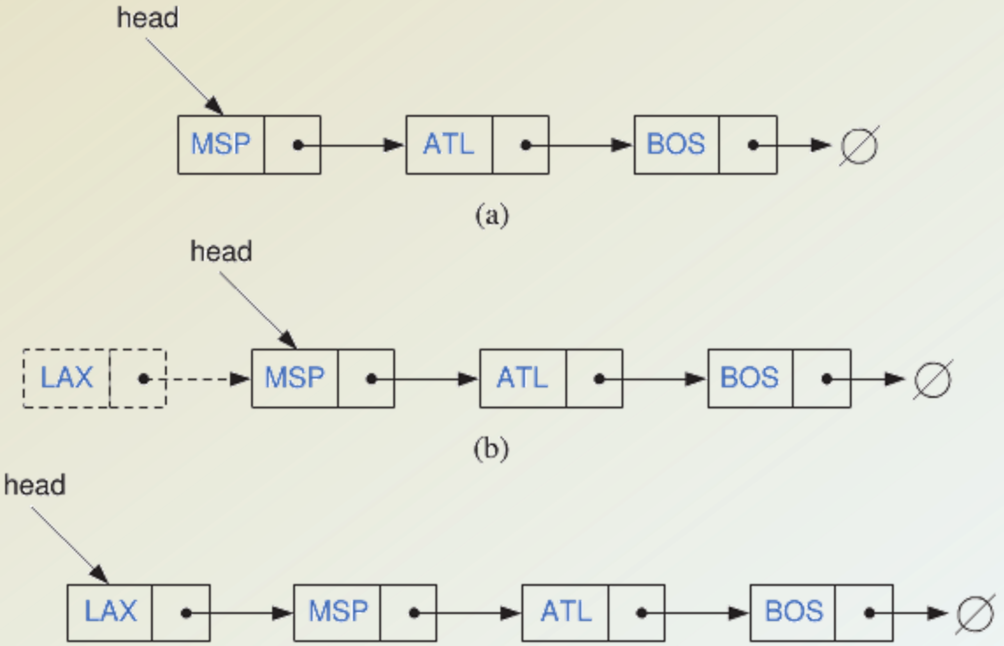
Node.java



Inserting at the Head

```
Algorithm addFirst(v):
    v.setNext(head)  {make v point to the old head node}
    head ← v  {make variable head point to new node}
    size ← size+1  {increment the node count}
```

1. Allocate a new node.
2. Insert new element.
3. Have new node point to old head.
4. Update head to point to new node.



(a)

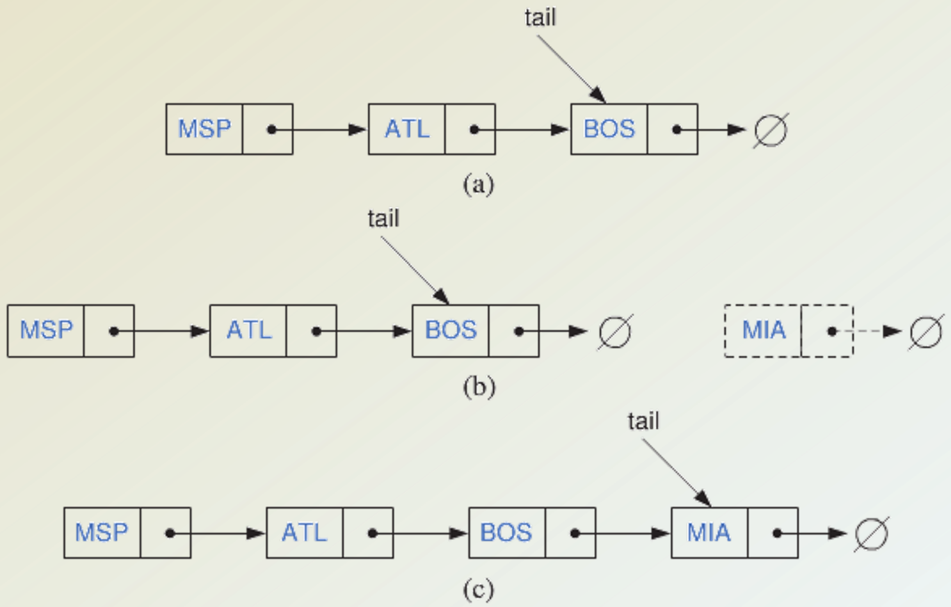
(b)

(c)

Inserting at the Tail

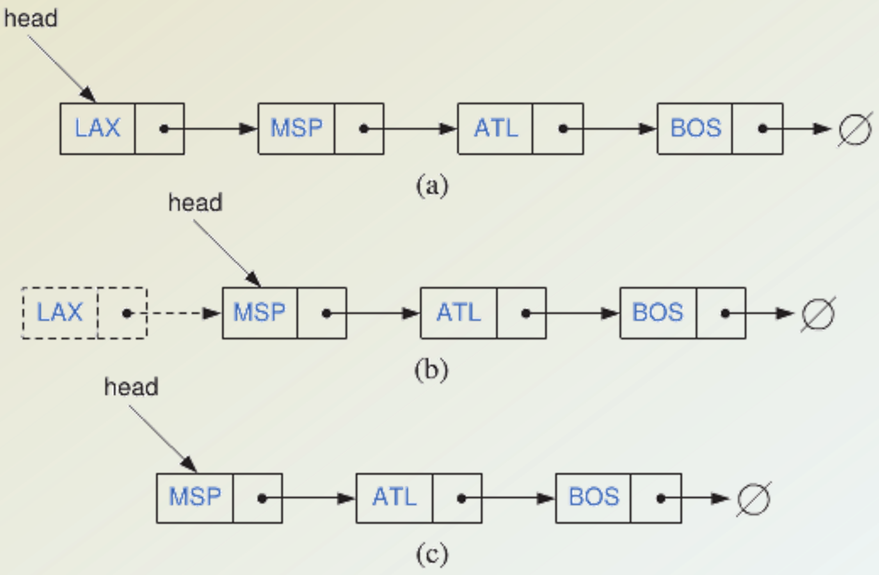
```
Algorithm addLast(v):
    v.setNext(null)  {make new node v point to null object}
    tail.setNext(v)  {make old tail node point to new node}
    tail ← v  {make variable tail point to new node.}
    size ← size+1  {increment the node count}
```

1. Allocate a new node
2. Insert new element
3. Have new node point to null
4. Have old last node point to new node
5. Update tail to point to new node



Removing at the Head

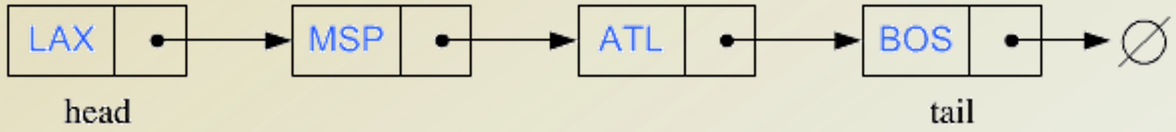
```
Algorithm removeFirst():
    if head = null then
        Indicate an error: the list is empty.
    t ← head
    head ← head.getNext()  {make head point to next node (or null)}
    t.setNext(null)  {null out the next pointer of the removed node}
    size ← size−1  {decrement the node count}
```



1. Update head to point to next node in the list
2. Allow garbage collector to reclaim the former first node

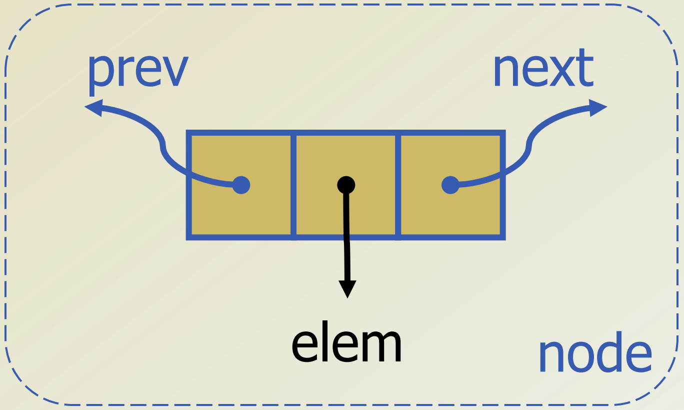
Removing at the Tail

- Removing at the tail of a singly linked list is not efficient!
- There is no constant-time way to update the tail to point to the previous node.

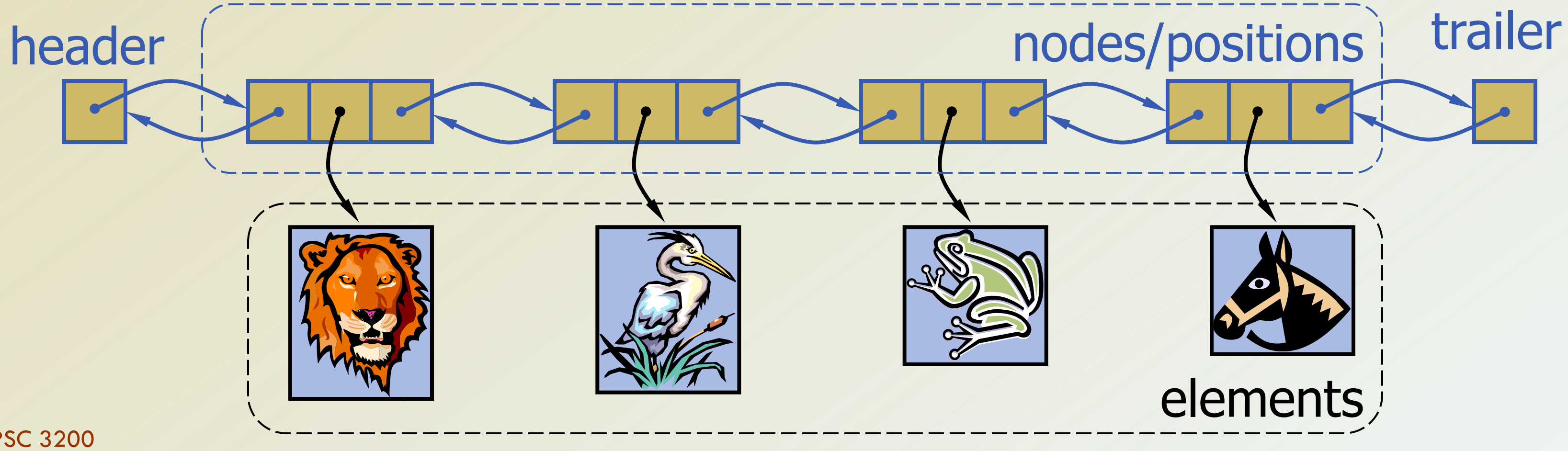


Doubly Linked List

- 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 (sentinel)

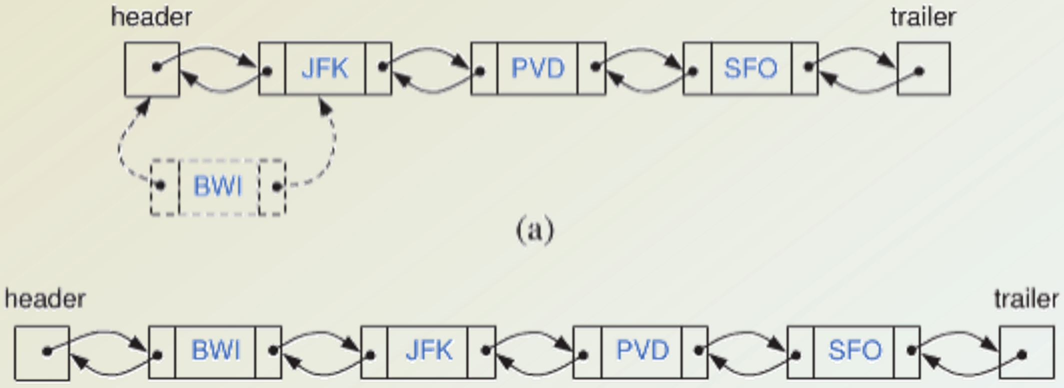


DNode.java



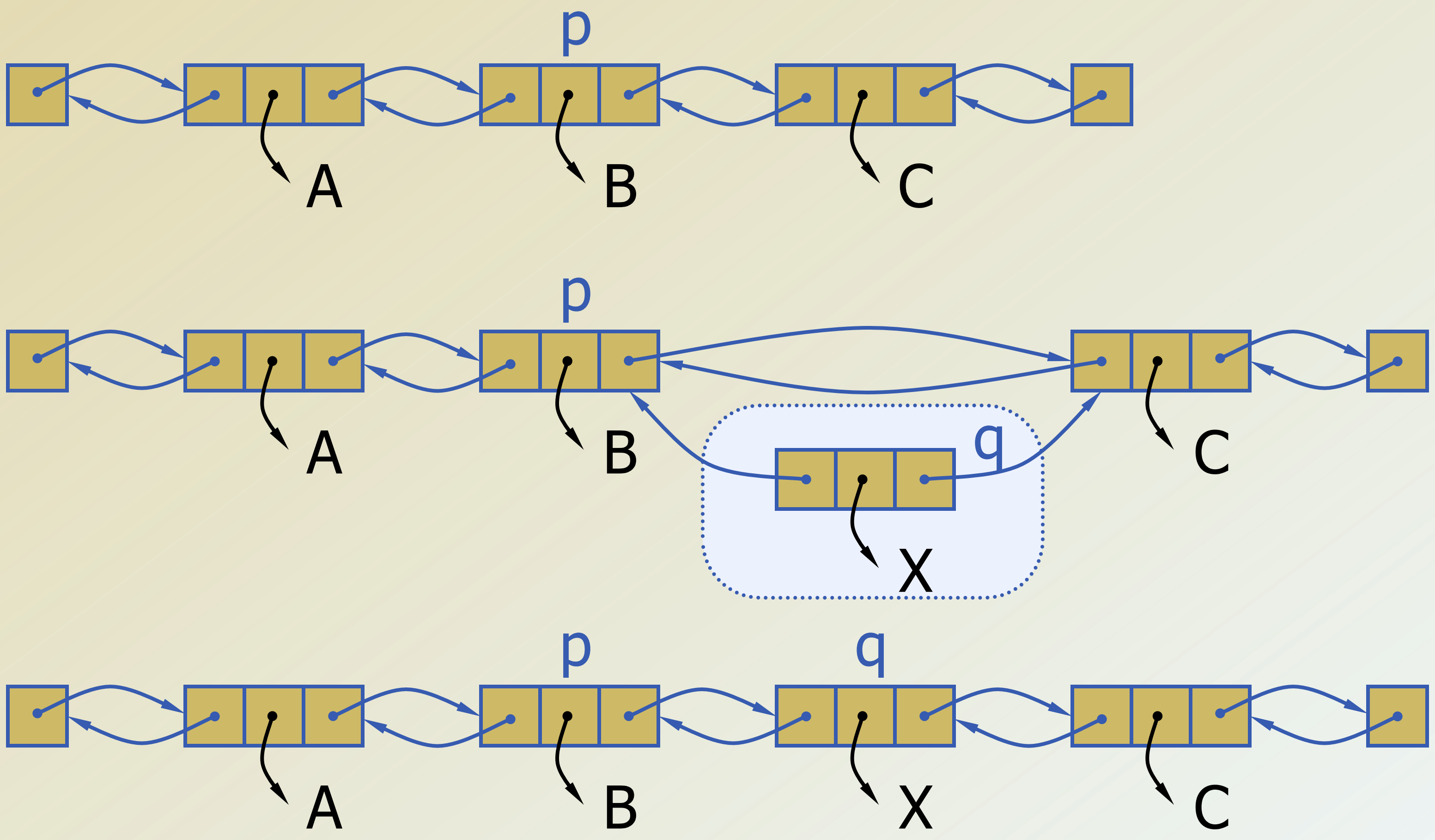
Add First – Algorithm

```
Algorithm addFirst(v):
        w ← header.getNext( )  {first node}
        v.setNext(w)
        v.setPrev(header)
        w.setPrev(v)
        header.setNext(v)
        size = size+1
```



Insertion in the Middle

- We visualize operation insertAfter(p, X), which returns position q

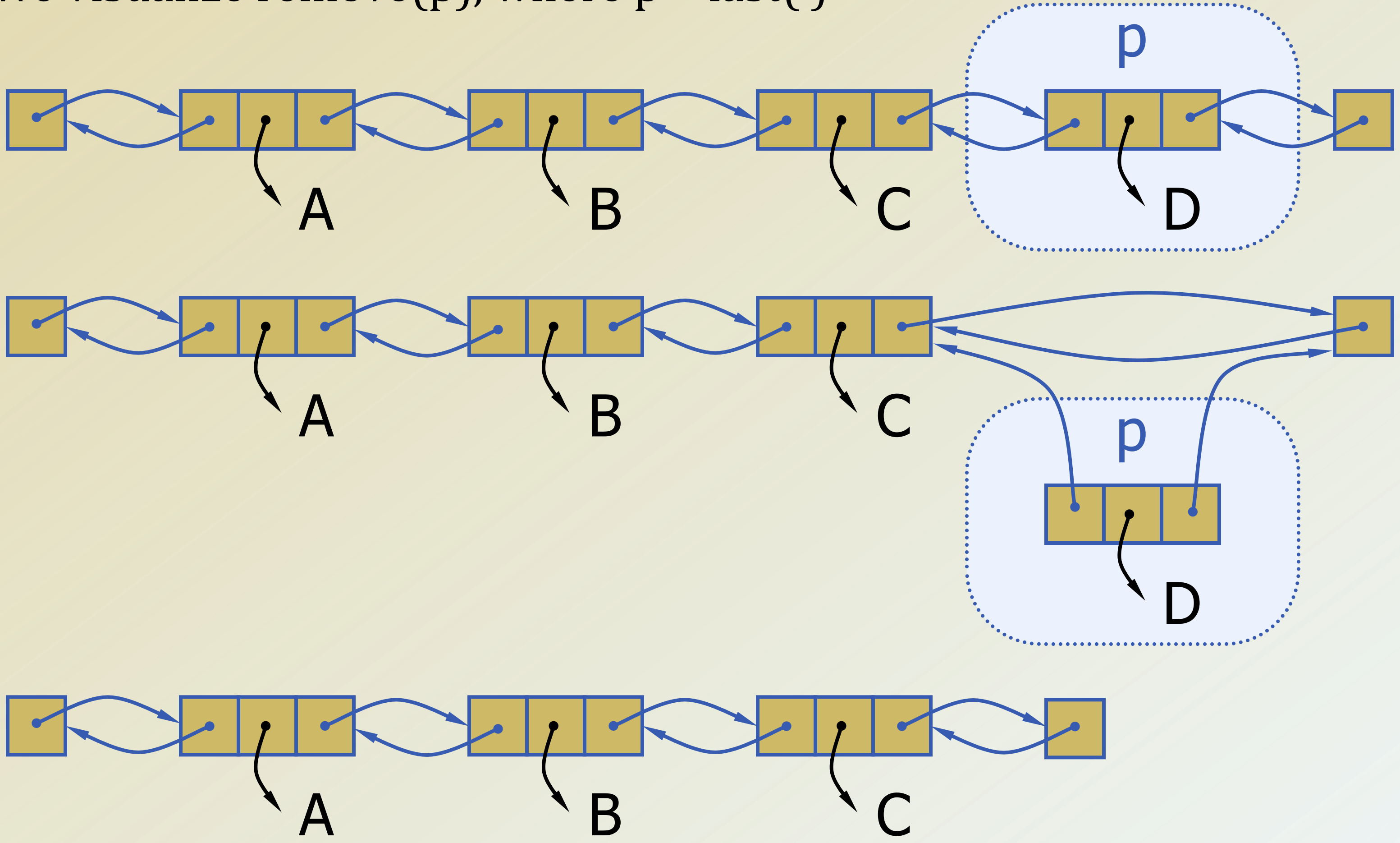


Insertion in the Middle - Algorithm

```
Algorithm addAfter(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}
```

Remove Last

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

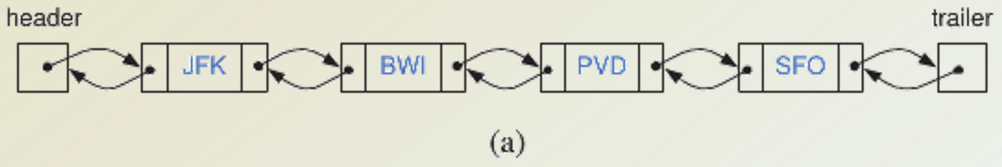


Remove Last - Algorithm

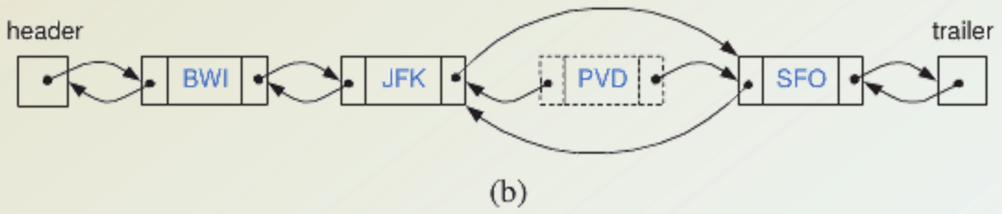
```
Algorithm removeLast( ):
    if size = 0 then
        Indicate an error: the list is empty
    v ← trailer.getPrev( )  {last node}
    u ← v.getPrev( )  {node before the last node}
    trailer.setPrev(u)
    u.setNext(trailer)
    v.setPrev(null)
    v.setNext(null)
    size = size−1
```

Remove in the Middle

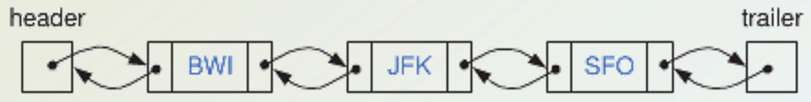
```
Algorithm remove(v):
        u ← v.getPrev( )  {node before v}
        w ← v.getNext( )  {node after v}
        w.setPrev(u)  {link out v}
        u.setNext(w)
        v.setPrev(null)  {null out the fields of v}
        v.setNext(null)
        size ← size−1  {decrement the node count}
```



(a)



(b)



(c)

Sorting a Doubly Linked List

```
Algorithm InsertionSort(L):
    Input: A doubly linked list L of comparable elements
    Output: The list L with elements rearranged in nondecreasing order
    if L.size( ) <= 1 then
        return
    end ← L.getFirst( )
    while end is not the last node in L do
        pivot ← end.getNext( )
        Remove pivot from L
        ins ← end
        while ins is not the header and ins's element is greater than pivot's do
            ins ← ins.getPrev( )
        Add pivot just after ins in L
    if  ins = end then {We just added pivot after end in this case}
        end ← end.getNext( )
```

DList.java

Circular Linked List

- A circularly linked list has the same kind of nodes as a singly linked list.
- There is no head or tail in a circularly linked list.
- For instead of having the last node's next pointer be null, in a circularly linked list, it points back to the first node.
- A special node, which we call the **cursor**.
  - Where to start from.
  - When we are done.

Circular Linked List Methods

- **add(v):** Insert a new node v immediately after the cursor; if the list is empty, then v becomes the cursor and its next pointer points to itself.
- **remove( ):** Remove and return the node v immediately after the cursor (not the cursor itself, unless it is the only node); if the list becomes empty, the cursor is set to null.
- **advance( ):** Advance the cursor to the next node in the list.

Recursion

- repetition can be achieved by writing loops, such as **for** loops and **while** loops.
- Another way to achieve repetition is through **recursion**, which occurs when a function refers to itself in its own definition.
- A function is said to be recursive if it calls itself either directly or indirectly through another function.
- A recursive function knows when to stop calling itself once a ***base case*** is reached.

Using Recursion To Solve Problems

- Recursion is a common form of the general-purpose problem-solving technique called ***"divide and conquer"***.

- The principle of divide-and-conquer is that, you solve a given problem **P** in 3 steps:
  1. Divide **P** into several sub-problems, **P1,P2,..., Pn**.
  2. Solve, however you can, each of the sub-problems to get solutions **S1, S2, ..., Sn.**
  3. Use **S1, S2, ..., Sn** to construct a solution to the original problem **P**.

- This is often recursive, because in order to solve one or more of the sub-problems, you might use this very same strategy.

- For instance, you might solve **P1** by sub-dividing it into P1a,P1b..., solving each one of them, and putting together their solutions **S1a, S1b, ,,,** to get the solution **S1** to problem **P1**.

The Factorial Function

- The factorial of a positive integer n, denoted **n!**, is defined as the product of the integers from **1** to **n**.

$$n! = \begin{cases} 1 & \text{if } n = 0 \\ n \cdot (n-1) \cdot (n-2) \cdots 3 \cdot 2 \cdot 1 & \text{if } n \geq 1. \end{cases}$$

```
factorial(5) = 5·(4·3·2·1) = 5·factorial(4)
```

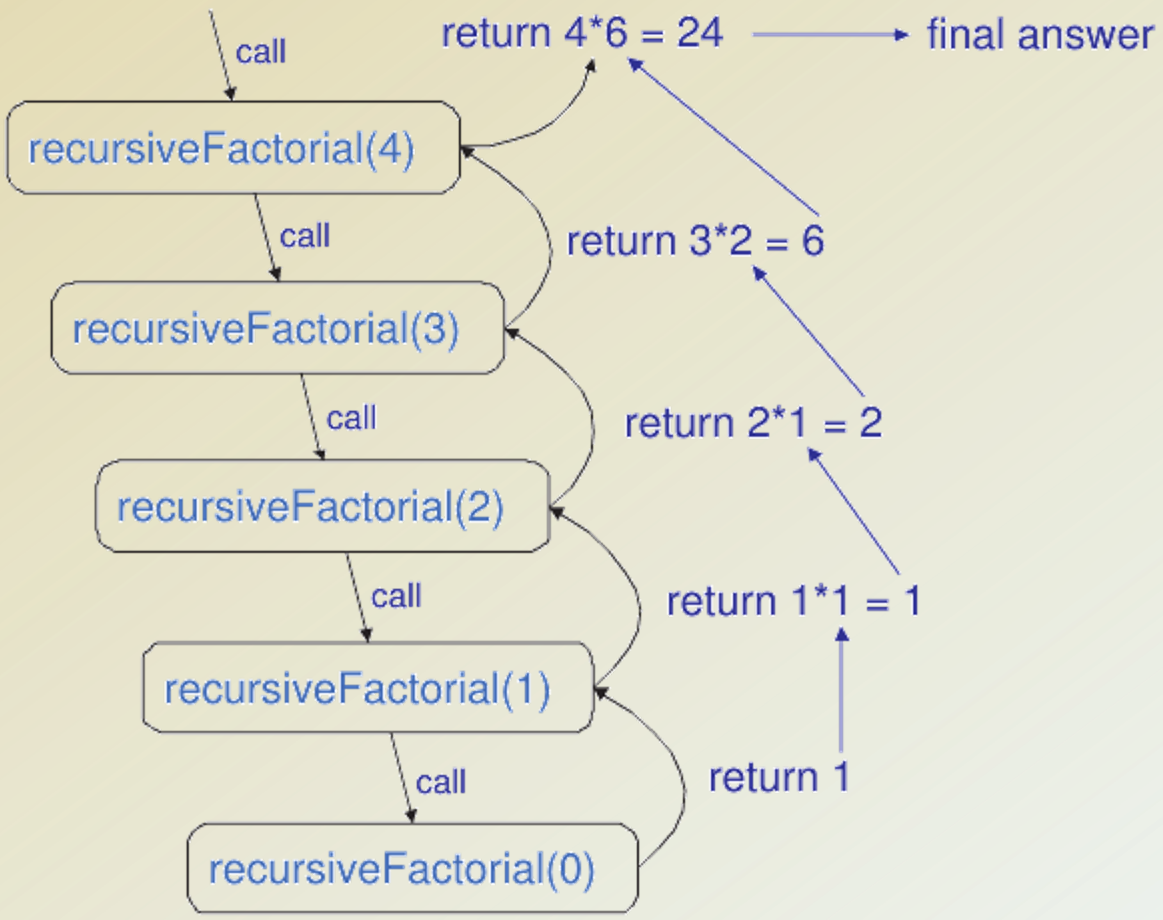
- **recursive definition:**

$$\text{factorial}(n) = \begin{cases} 1 & \text{if } n = 0 \\ n \cdot \text{factorial}(n-1) & \text{if } n \geq 1. \end{cases}$$

Recursive Implementation of the Factorial Function

```
public  static int recursiveFactorial(int n)
{
   // recursive factorial function
   if (n == 0)
      return 1;       // basis case
   else
      return n * recursiveFactorial(n-1); // recursive case
}
```

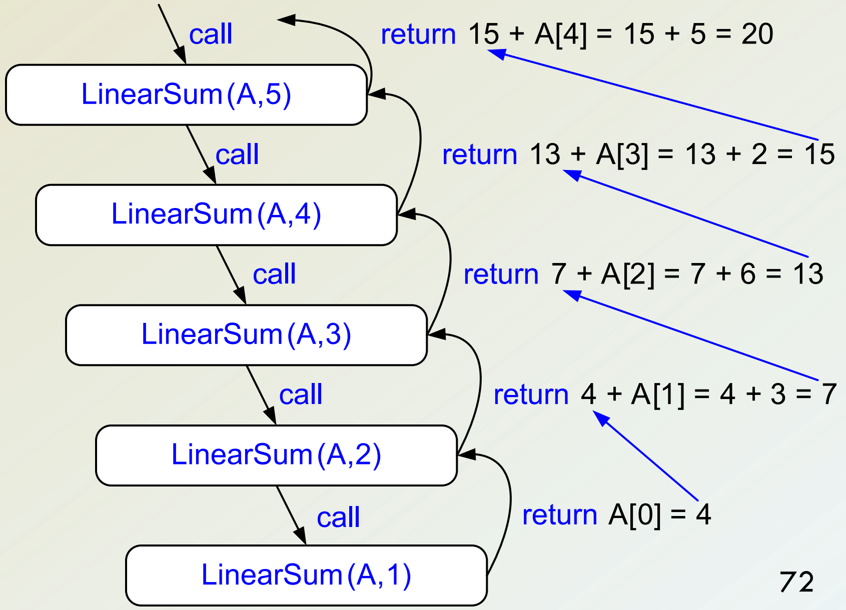
Recursive Trace



Linear Recursion
Summing the Elements of an Array Recursively

```
Algorithm LinearSum(A, n):
    Input: A integer array A and an integer n = 1, such that A has at least n
      elements
    Output: The sum of the first n integers in A
    if n = 1 then
      return A[0]
    else
      return LinearSum(A, n - 1) + A[n - 1]
```

recursion trace:


call
return 15 + A[4] = 15 + 5 = 20
LinearSum(A,5)
call
return 13 + A[3] = 13 + 2 = 15
LinearSum(A,4)
call
return 7 + A[2] = 7 + 6 = 13
LinearSum(A,3)
call
return 4 + A[1] = 4 + 3 = 7
LinearSum(A,2)
call
return A[0] = 4
LinearSum(A,1)

Linear Recursion
Summing the Elements of an Array Recursively

- For an input array of size ***n***, Algorithm LinearSum makes ***n*** calls.

- Hence, it will take an ***amount of time*** that is roughly proportional to ***n***, since it spends a constant amount of time performing the non-recursive part of each call.

- Moreover, we can also see that the ***memory space*** used by the algorithm (in addition to the array A) is also roughly proportional to ***n***, since we need a constant amount of memory space for each of the ***n*** boxes in the trace at the time we make the final recursive call (for n = 1)

Linear Recursive

- **Test for base cases**
  - Begin by testing for a set of ***base cases*** (there should be at least one).
  - Every possible chain of recursive calls must eventually reach a base case, and the handling of each base case should not use recursion.
- **Recur once**
  - Perform a single recursive call.
  - This step may have a test that decides which of several possible recursive calls to make, but it should ultimately make just one of these calls.
  - Define each possible recursive call so that it makes progress towards a base case.

Reversing an Array

```
Algorithm ReverseArray(A, i, j):
   Input: An array A and nonnegative integer indices i and j
   Output: The reversal of the elements in A starting at index
           i and ending at j
  if i < j then
      Swap A[i] and A[ j]
      ReverseArray(A, i + 1, j - 1)
  return
```

Defining Arguments for Recursion

- In creating recursive methods, it is important to define the methods in ways that facilitate recursion.
- This sometimes requires we define additional paramaters that are passed to the method.
- For example, we defined the array reversal method as ReverseArray(A, i, j), not ReverseArray(A).

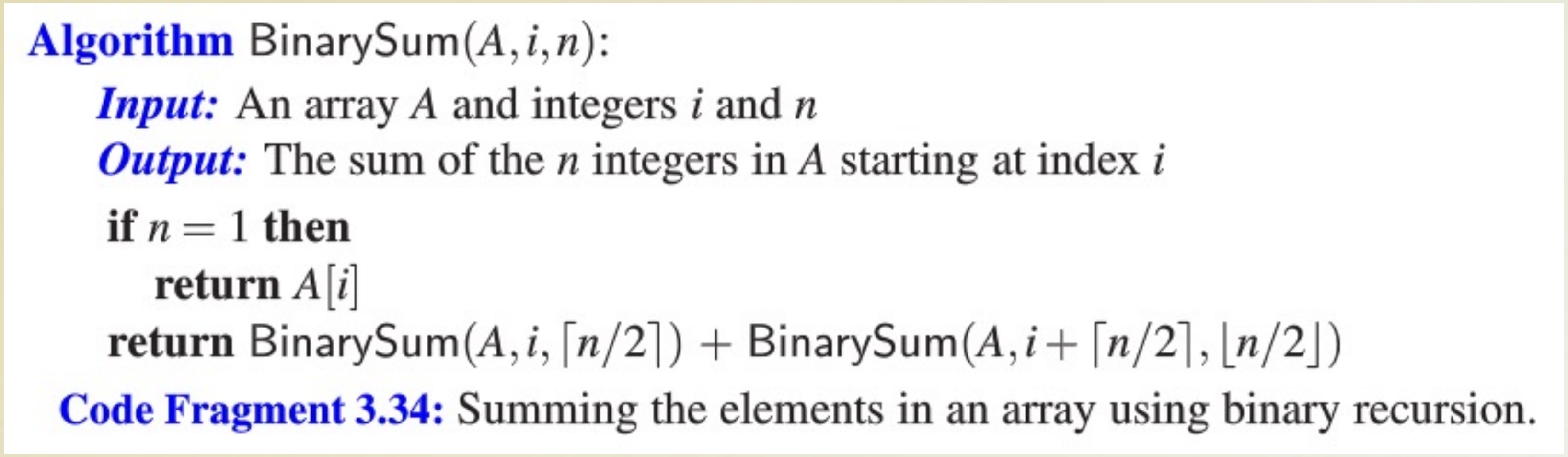
Tail Recursion

- Tail recursion occurs when a linearly recursive method makes its recursive call as its last step.
- The array reversal method is an example.

- Methods can be easily converted to non-recursive methods (which saves on some resources).
- Example:

```
Algorithm IterativeReverseArray(A, i, j ):
    Input: An array A and nonnegative integer indices i and j
    Output: The reversal of the elements in A starting at index i and
            ending at j
  while i <  j do
      Swap A[i ] and A[ j ]
      i  = i + 1
      j  = j - 1
  return
```

Binary Recursion



```
Algorithm BinarySum($A, i, n$):
    Input: An array $A$ and integers $i$ and $n$
    Output: The sum of the $n$ integers in $A$ starting at index $i$
    if $n = 1$ then
        return $A[i]$
    return BinarySum($A, i, \lceil n/2 \rceil$) + BinarySum($A, i + \lceil n/2 \rceil, \lfloor n/2 \rfloor$)
```

Code Fragment 3.34: Summing the elements in an array using binary recursion.

Recursion Vs. Iterations

- So which way is better? – iteration or recursion?
- Answer is that it depends on what you are trying to do.
- Usually, a recursive approach more naturally mirrors the problem at hand. So, a recursive approach makes it simpler to tackle a problem which may not have the most obvious of answers.
- However, recursion carries an overhead that for each recursive call needs space on the stack frame.
- This extra memory need can be quite processor intensive and consume a lot of memory if recursion is nested deeply.
- Iteration does not have this overhead as it occurs within the method so the overhead of repeated method calls and extra memory is omitted.

Recursion Vs. Iterations

Recursion

- Terminates when a base case is reached.
- Each recursive call requires extra space on the stack frame (i.e. memory).
- If we get infinite recursion, we will eventually run out of memory, resulting in a stack overflow.
- Solutions to some problems are easier to formulate recursively.

Iteration

- Terminates when a condition is proven to be false.
- Each iteration does not require any extra space as it resides in the same method.
- An infinite loop could potentially loop forever since there is no extra memory being created.
- Iterative solutions to a problem may not always be as obvious as a recursive solution.