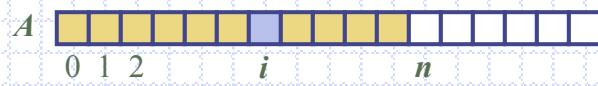


Arrays and Lists

1

Array Definition

- ❑ An **array** is a sequenced collection of variables all of the same type. Each variable, or **cell**, in an array has an **index**, which uniquely refers to the value stored in that cell. The cells of an array, A, are numbered 0, 1, 2, and so on.
 - ❑ Each value stored in an array is often called an **element** of that array.



2

Array Length and Capacity

- Since the length of an array determines the maximum number of things that can be stored in the array, we will sometimes refer to the length of an array as its **capacity**.
- In Java, the length of an array named `a` can be accessed using the syntax `a.length`. Thus, the cells of an array, `a`, are numbered `0, 1, 2, ..., n - 1`, and the cell with index `k` can be accessed with syntax `a[k]`.



3

Declaring Arrays (first way)

- The first way to create an array is to use an assignment to a literal form when initially declaring the array, using a syntax as:

```
elementType[] arrayName = { initialValue0, initialValue1, ..., initialValueN-1 };
```

- The `elementType` can be any Java base type or class name, and `arrayName` can be any valid Java identifier. The initial values must be of the same type as the array.

4

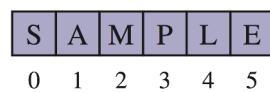
Declaring Arrays (second way)

- The second way to create an array is to use the **new** operator.
 - However, because an array is not an instance of a class, we do not use a typical constructor. Instead we use the syntax:
new elementType[length]
- length** is a positive integer denoting the length of the new array.
- The **new** operator returns a reference to the new array, and typically this would be assigned to an array variable.

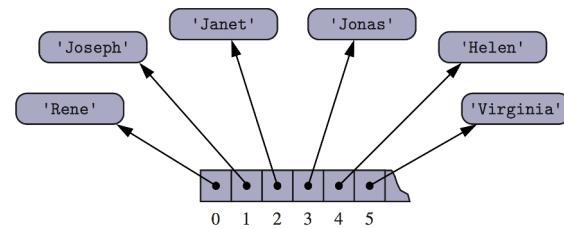
5

Arrays of Characters or Object References

- An array can store primitive elements, such as characters.



- An array can also store references to objects.



6

Arrays of Generic Types

- Java doesn't allow create new arrays of Generic types but:

```
Pair<String,Double>[] holdings;  
holdings=new Pair<String,Double>[25];  
holdings=new Pair[25];  
holdings[0]= new Pair <> ("Mia",35.24);  
  
// licit!  
// compiler error!  
// correct but warning for the cast  
// valid assignment
```

7

Java Example: Game Entries

- A game entry stores the name of a player and her best score so far in a game

```
1 public class GameEntry {  
2     private String name;           // name of the person earning this score  
3     private int score;            // the score value  
4     /** Constructs a game entry with given parameters.. */  
5     public GameEntry(String n, int s) {  
6         name = n;  
7         score = s;  
8     }  
9     /** Returns the name field. */  
10    public String getName() { return name; }  
11    /** Returns the score field. */  
12    public int getScore() { return score; }  
13    /** Returns a string representation of this entry. */  
14    public String toString() {  
15        return "(" + name + ", " + score + ")";  
16    }  
17 }
```

8

Java Example: Scoreboard

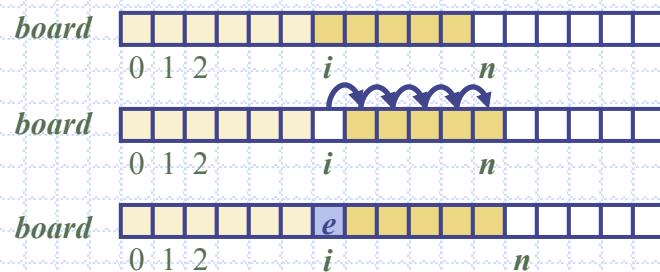
- Keep track of players and their best scores in an array, board
 - The elements of board are objects of class GameEntry
 - Array board is sorted by score

```
1  /** Class for storing high scores in an array in nondecreasing order. */
2  public class Scoreboard {
3      private int numEntries = 0;          // number of actual entries
4      private GameEntry[ ] board;         // array of game entries (names & scores)
5      /** Constructs an empty scoreboard with the given capacity for storing entries. */
6      public Scoreboard(int capacity) {
7          board = new GameEntry[capacity];
8      }
...   // more methods will go here
36 }
```

9

Adding an Entry

- To add an entry e into array $board$ at index i , we need to make room for it by shifting forward the $n - i$ entries $board[i], \dots, board[n - 1]$



10

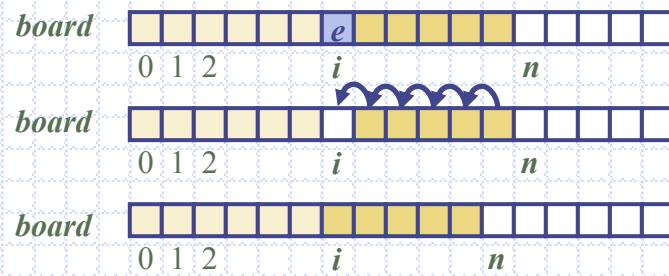
Java Example

```
9  /** Attempt to add a new score to the collection (if it is high enough) */
10 public void add(GameEntry e) {
11     int newScore = e.getScore();
12     // is the new entry e really a high score?
13     if (numEntries < board.length || newScore > board[numEntries-1].getScore()) {
14         if (numEntries < board.length)           // no score drops from the board
15             numEntries++;                      // so overall number increases
16         // shift any lower scores rightward to make room for the new entry
17         int j = numEntries - 1;
18         while (j > 0 && board[j-1].getScore() < newScore) {
19             board[j] = board[j-1];            // shift entry from j-1 to j
20             j--;
21         }
22         board[j] = e;                      // when done, add new entry
23     }
24 }
```

11

Removing an Entry

- To remove the entry e at index i , we need to fill the hole left by e by shifting backward the $n - i - 1$ elements $board[i + 1], \dots, board[n - 1]$



12

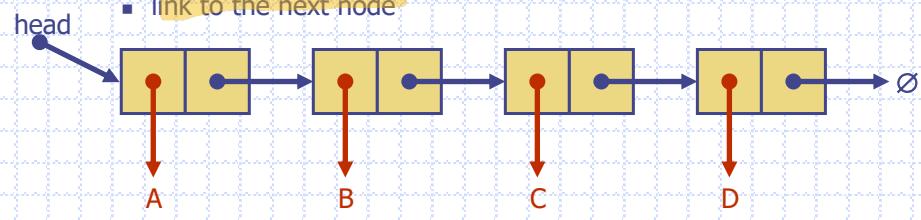
Java Example

```
25  /** Remove and return the high score at index i.*/
26  public GameEntry remove(int i) throws IndexOutOfBoundsException {
27      if (i < 0 || i >= numEntries)
28          throw new IndexOutOfBoundsException("Invalid index: " + i);
29      GameEntry temp = board[i];           // save the object to be removed
30      for (int j = i; j < numEntries - 1; j++) // count up from i (not down)
31          board[j] = board[j+1];           // move one cell to the left
32      board[numEntries - 1] = null;        // null out the old last score
33      numEntries--;
34      return temp;                      // return the removed object
35  }
```

13

Singly Linked List

- ❑ A singly linked list is a concrete data structure consisting of a sequence of nodes, starting from a head pointer
- ❑ Each node stores
 - element
 - link to the next node



14

Node object: (E, Node<E>)

<E>

ha un oggetto e un
altro
nodo

(Prox oggetto)

A Nested Node Class

```
1 public class SinglyLinkedList<E> {  
2     //----- nested Node class -----  
3     private static class Node<E> {  
4         private E element;           // reference to the element stored at this node  
5         private Node<E> next;       // reference to the subsequent node in the list  
6         public Node(E e, Node<E> n) {  
7             element = e;  
8             next = n;  
9         }  
10        public E getElement() { return element; }  
11        public Node<E> getNext() { return next; }  
12        public void setNext(Node<E> n) { next = n; }  
13    } //----- end of nested Node class -----  
... rest of SinglyLinkedList class will follow ...
```

15

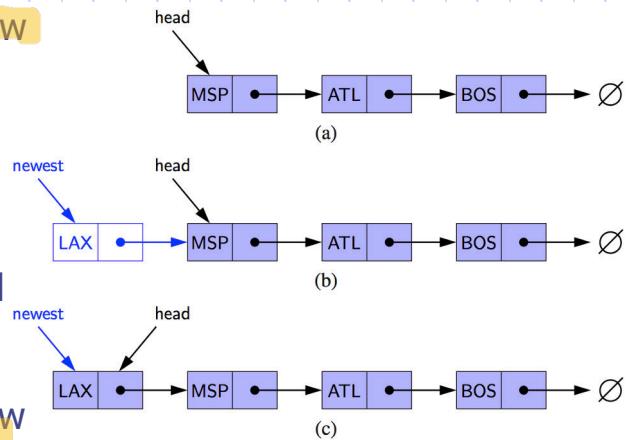
Accessor Methods

```
1 public class SinglyLinkedList<E> {  
...  
    (nested Node class goes here)  
14    // instance variables of the SinglyLinkedList  
15    private Node<E> head = null;          // head node of the list (or null if empty)  
16    private Node<E> tail = null;          // last node of the list (or null if empty)  
17    private int size = 0;                  // number of nodes in the list  
18    public SinglyLinkedList() { }          // constructs an initially empty list  
19    // access methods  
20    public int size() { return size; }  
21    public boolean isEmpty() { return size == 0; }  
22    public E first() {                     // returns (but does not remove) the first element  
23        if (isEmpty()) return null;  
24        return head.getElement();  
25    }  
26    public E last() {                      // returns (but does not remove) the last element  
27        if (isEmpty()) return null;  
28        return tail.getElement();  
29    }
```

16

Inserting at the Head

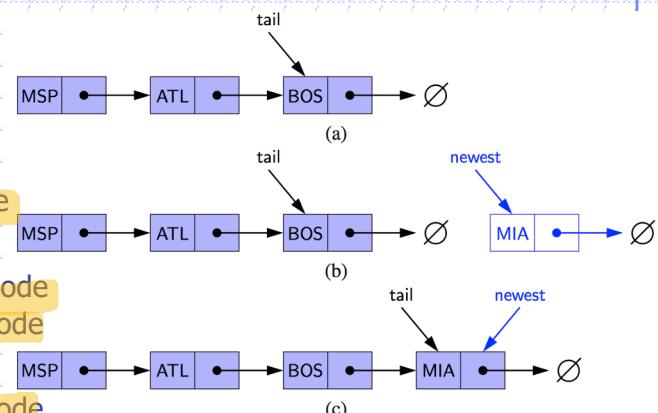
- Allocate new node
- Insert new element
- Have new node point to old head
- Update head to point to new node



17

Inserting at the Tail

- Allocate a new node
- Insert new element
- Have new node point to null
- Have old last node point to new node
- Update tail to point to new node



18

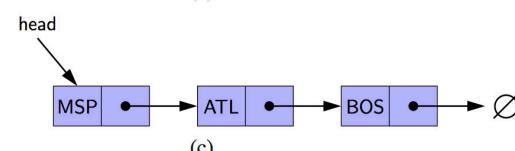
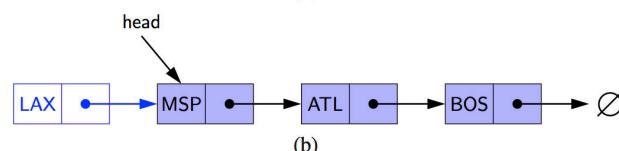
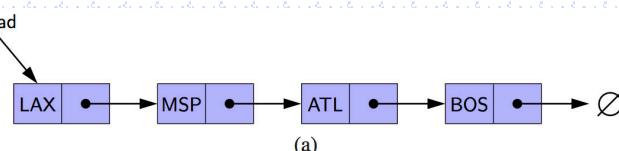
Java Methods

```
31 public void addFirst(E e) {           // adds element e to the front of the list
32     head = new Node<E>(e, head);    // create and link a new node
33     if (size == 0)                  // special case: new node becomes tail also
34         tail = head;
35     size++;
36 }
37 public void addLast(E e) {            // adds element e to the end of the list
38     Node<E> newest = new Node<E>(e, null); // node will eventually be the tail
39     if (isEmpty())
40         head = newest;             // special case: previously empty list
41     else
42         tail.setNext(newest);      // new node after existing tail
43     tail = newest;                // new node becomes the tail
44     size++;
45 }
```

19

Removing at the Head

- Update head to point to next node in the list
- Allow garbage collector to reclaim the former first node



20

Java Method

```
46 public E removeFirst() {  
47     if (isEmpty()) return null;  
48     E answer = head.getElement();  
49     head = head.getNext();  
50     size--;  
51     if (size == 0)  
52         tail = null;  
53     return answer;  
54 }  
55 }
```

// removes and returns the first element
// nothing to remove
// will become null if list had only one node
// special case as list is now empty

lamento de nome

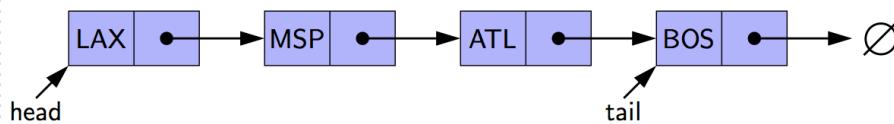
elements removed

Se avvia 1 elemento
next è NULL, quindi caso

21

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



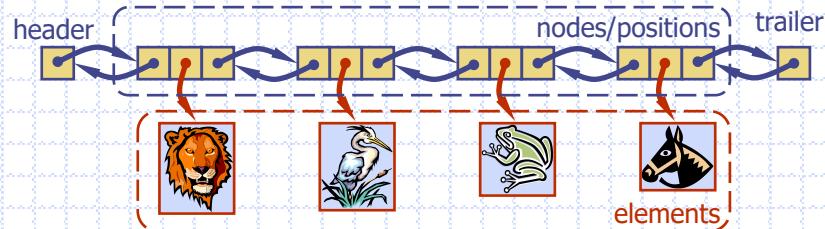
22

Esempio creazione e utilizzo SingleLinkedList

```
1 import java.io.*;
2 import java.util.Scanner;
3 import net.datastructures.*;
4
5 public class TestList {
6     public static void main(String argv[]) {
7         int i;
8         Scanner input =new Scanner(System.in);
9         SinglyLinkedList<Integer> lista= new SinglyLinkedList<>();
10        System.out.println("Avvio lista");
11
12        for (i=0; i<100;i++) {
13            Integer j=new Integer(i);
14            lista.addLast(j);
15        }
16        System.out.println("Elemento da inserire in coda alla lista");
17        Integer a=input.nextInt();
18        lista.addLast(a);
19        System.out.println("Elemento da inserire in testa alla lista");
20        Integer b=input.nextInt();
21        lista.addFirst(b);
22        System.out.println("Dimensione lista= "+ lista.size());
23        System.out.println("Primo lista= "+ lista.first());
24        System.out.println("Ultimo lista= "+ lista.last());
25        System.out.println("Stampa lista= "+ lista);
26    }
27 }
```

Doubly Linked List

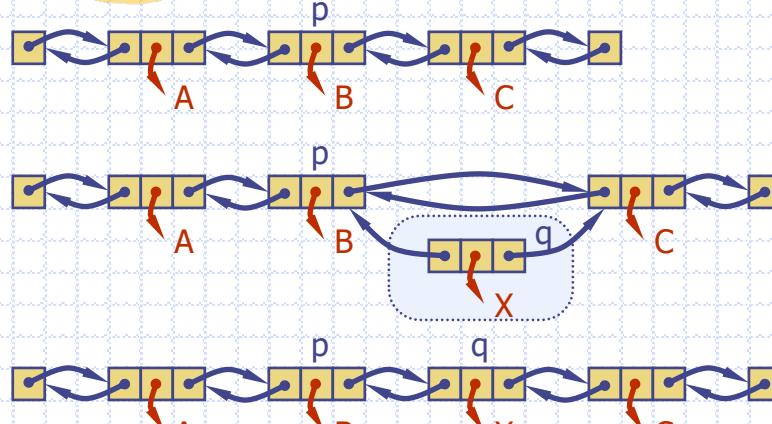
- ❑ A doubly linked list can be traversed forward and backward
- ❑ Nodes store:
 - element
 - link to the previous node
 - link to the next node
- ❑ Special trailer and header nodes



24

Insertion

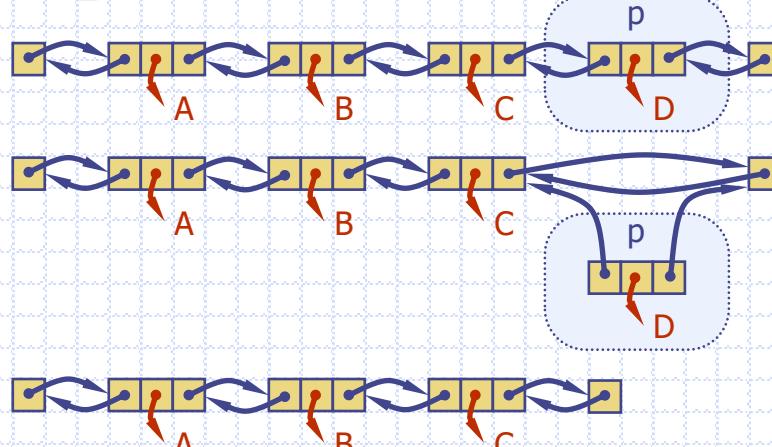
- Insert a new node, q , between p and its successor.



25

Deletion

- Remove a node, p , from a doubly linked list.



26

Doubly-Linked List in Java

```

1  /** A basic doubly linked list implementation. */
2  public class DoublyLinkedList<E> {
3      //----- nested Node class -----
4      private static class Node<E> {
5          private E element;           // reference to the element stored at this node
6          private Node<E> prev;        // reference to the previous node in the list
7          private Node<E> next;        // reference to the subsequent node in the list
8          public Node(E e, Node<E> p, Node<E> n) {
9              element = e;
10             prev = p;
11             next = n;
12         }
13         public E getElement() { return element; }
14         public Node<E> getPrev() { return prev; }
15         public Node<E> getNext() { return next; }
16         public void setPrev(Node<E> p) { prev = p; }
17         public void setNext(Node<E> n) { next = n; }
18     } //----- end of nested Node class -----
19 }
```

27

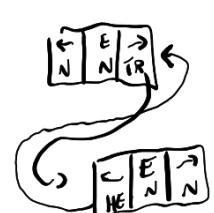
Non sono un pezzo
della lista.

Doubly-Linked List in Java, 2

```

21  private Node<E> header;           // header sentinel
22  private Node<E> trailer;          // trailer sentinel
23  private int size = 0;              // number of elements in the list
24  /** Constructs a new empty list. */
25  public DoublyLinkedList() {
26      header = new Node<E>(null, null, null);    // create header
27      trailer = new Node<E>(null, header, null);   // trailer is preceded by header
28      header.setNext(trailer);                     // header is followed by trailer
29  }
30  /** Returns the number of elements in the linked list. */
31  public int size() { return size; }
32  /** Tests whether the linked list is empty. */
33  public boolean isEmpty() { return size == 0; }
34  /** Returns (but does not remove) the first element of the list. */
35  public E first() {
36      if (isEmpty()) return null;                // first element is beyond header
37      return header.getNext().getElement();       // first element is before trailer
38  }
39  /** Returns (but does not remove) the last element of the list. */
40  public E last() {
41      if (isEmpty()) return null;                // last element is before trailer
42      return trailer.getPrev().getElement();      // last element is after header
43  }
```

// last element is before trailer



28

Doubly-Linked List in Java, 3

```

44 // public update methods
45 /** Adds element e to the front of the list. */
46 public void addFirst(E e) {
47     addBetween(e, header, header.getNext()); Riferimenti degli esterni
48 }
49 /** Adds element e to the end of the list. */
50 public void addLast(E e) {
51     addBetween(e, trailer.getPrev(), trailer); place just before the trailer
52 }
53 /** Removes and returns the first element of the list. */
54 public E removeFirst() {
55     if (isEmpty()) return null; nothing to remove
56     return remove(header.getNext()); first element is beyond header
57 }
58 /** Removes and returns the last element of the list. */
59 public E removeLast() {
60     if (isEmpty()) return null; nothing to remove
61     return remove(trailer.getPrev()); last element is before trailer
62 }

```

29

Così lo costante, non dipende da dimensione n.

Doubly-Linked List in Java, 4

```

64 // private update methods
65 /** Adds element e to the linked list in between the given nodes. */
66 private void addBetween(E e, Node<E> predecessor, Node<E> successor) {
67     // create and link a new node
68     Node<E> newest = new Node<E>(e, predecessor, successor); pushing
69     predecessor.setNext(newest); Operaz. Remove
70     successor.setPrev(newest); glancia man carentita.
71     size++;
72 }
73 /** Removes the given node from the list and returns its element. */
74 private E remove(Node<E> node) {
75     Node<E> predecessor = node.getPrev();
76     Node<E> successor = node.getNext();
77     predecessor.setNext(successor);
78     successor.setPrev(predecessor);
79     size--;
80     return node.getElement();
81 }
82 } //----- end of DoublyLinkedList class -----

```

30

Esempio creazione e utilizzo DoublyLinkedList

```
1 import java.io.*;
2 import java.util.Scanner;
3
4
5 public class TestDoubleList {
6     public static void main(String argv[])
7     {
8         int i;
9         Scanner input =new Scanner(System.in);
10        DoublyLinkedList<Integer> lista= new DoublyLinkedList<>();
11
12        System.out.println("Avvio lista");
13
14
15        for (i=0; i<100;i++) {
16            Integer j=new Integer(i);
17            lista.addLast(j);
18        }
19
20
21        System.out.println("Elemento da inserire in coda alla lista");
22        Integer a=input.nextInt();
23        lista.addLast(a);
24        System.out.println("Elemento da inserire in testa alla lista");
25        Integer b=input.nextInt();
26        lista.addFirst(b);
27        System.out.println("Dimensione lista= "+ lista.size());
28        System.out.println("Primo lista= "+ lista.first());
29        System.out.println("Ultimo lista= "+ lista.last());
30        System.out.println("Rimuovo Ultimo lista= "+ lista.remove(lista.trailer.getPrev()));
31        System.out.println("Nuovo Ultimo lista= "+ lista.last());
32        System.out.println("Stampa lista= "+ lista);
33    }
34 }
```

1

