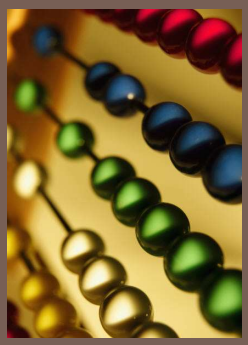
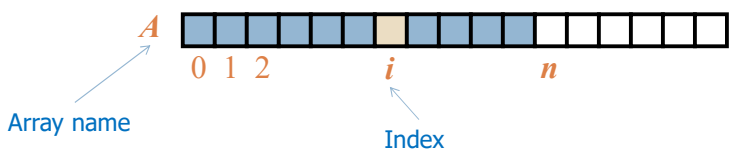
ARRAYS



1

Array Definition

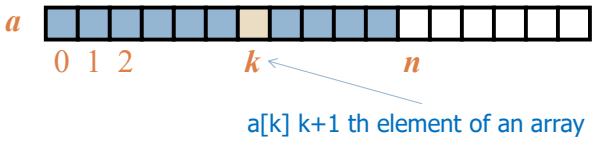
- An ***array*** is a sequenced collection of variables all of the same type. Each variable, or ***cell***, in an array has an ***index***, which uniquely refers to the value stored in that cell. The cells of an array, A, are numbered 0, 1, 2, and so on.
- Each value stored in an array is often called 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, and so on, up through a.length−1, and the cell with index *k* can be accessed with syntax a[*k*].



a[k] k+1 th element of an array

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:

$$elementType[]\ arrayName = \{initialValue_0, initialValue_1, \ldots, initialValue_{N-1}\};$$

- 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

Ex. Arrays

- Dot product.

```
double[] x = { 0.3, 0.6, 0.1 };   // declare an array x
double[] y = { 0.5, 0.1, 0.4 };   // declare second array y
int N = x.length;       // length of array x
double sum = 0.0;
for (int i = 0; i < N; i++) {
   sum = sum + x[i]*y[i];
}
```

5

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.

6

Ex. Arrays

- An array of size 1000000.

```
// scales to handle large arrays
double[] a = new double[1000000];
…
a[123456] = 3.0;
…
a[987654] = 8.0;
…
double x = a[123456] + a[987654];
```

declares, creates, and initializes

7

Ex. Arrays

```
int N = 10; // size of array
double[] a; // declare the array
a = new double[N]; // create the array
for (int i = 0; i < N; i++) // initialize the array
a[i] = 0.0; // all to 0.0
```

Compact alternative:

```
int N = 10; // size of array
double[] a = new double[N];  // declare create init
```

8

Exercise?

- Find maximum of the array values
- Reverse the elements within an array

9

Ex. Arrays

Create an array with random values	`double [] a = new double [N];`
	`for (int i = 0; i < N; i++ )`
	`  a[i] = Math.random();`

Print the array values one per line	`for (int i = 0; i < N; i++ )` `  System.out.println(a[i]);`

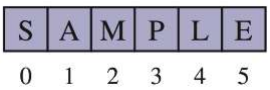
Find maximum of the array values	`double max = Double.NEGATIVE_INFINITY;`
	`for (int i = 0; i < N; i++ )`
	`  if (a[i]>max) max = a[i];`

Reverse the elements within an array	`for (int i = 0; i < N / 2; i++ )`
	`{`
	`  double temp = b[i];`
	`  b[i]=b[N-1-i];`
	`  b[N-i-1] = temp;`
	`}`

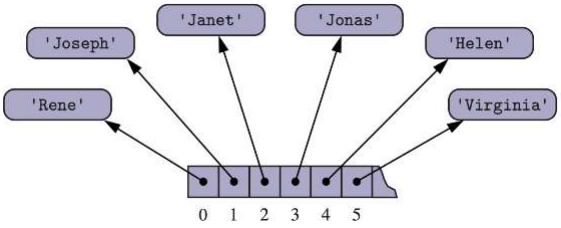
10

Arrays of Characters or Object References

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



- An array can also store references to objects.



11

Ex. Arrays

- Print a random card.

```
String[] rank = { "2", "3", "4", "5", "6", "7", "8", "9",
"10", "Jack", "Queen", "King", "Ace" };

String[] suit = { "Clubs", "Diamonds", "Hearts", "Spades" };

int i = (int) (Math.random() * 13); // between 0 and 12
int j = (int) (Math.random() * 4); // between 0 and 3

System.out.println(rank[i] + " of " + suit[j]);
```

12

Ex. Arrays

Matrix Addition	double[][] c = new double[N][N];
	for (int i = 0; i < N; i++)
	  for (int j = 0; j < N; j++)
	  c[i][j] = a[i][j] + b[i][j];

Matrix Multiplication	double[][] c = new double[N][N];
	for (int i = 0; i < N; i++)
	  for (int j = 0; j < N; j++)
	  for (int k = 0; k < N; k++)
	  c[i][j] += a[i][k] * b[k][j];

13

Java Example: Game Entries

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

```
public class GameEntry {
private String name; // name of the person earning this score
private int score; // the score value
/** Constructs a game entry with given parameters.. */
public GameEntry(String n, int s) {
    name = n;
    score = s;
}
/** Returns the name field. */
public String getName( ) { return name; }
/** Returns the score field. */
public int getScore( ) { return score; }
/** Returns a string representation of this entry. */
public String toString( ) {
    return "(" + name + ", " + score + ")";
}
}
```

14

Java Example: Scoreboard

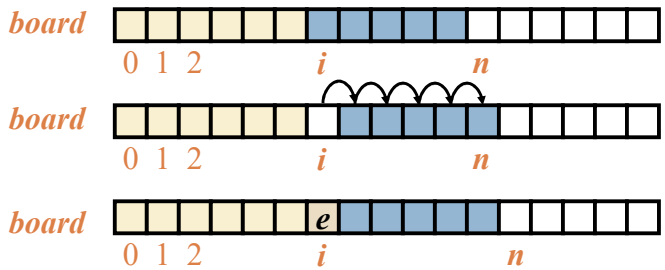
- Keep track of players and their best scores in an array, board
  - The elements of board are objects of class GameEntry
  - Array board is sorted by score

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

15

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 $board[i], \ldots, board[n-1]$



16

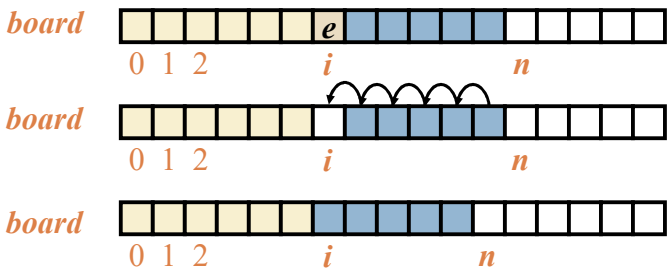
Java Example

```
/** Attempt to add a new score to the collection (if it is high enough) */
public void add(GameEntry e) {
int newScore = e.getScore( );
// is the new entry e really a high score?
if (numEntries < board.length || newScore > board[numEntries-1].getScore( ))
{
    if (numEntries < board.length) // no score drops from the board
        numEntries++; // so overall number increases
    // shift any lower scores rightward to make room for the new entry
    int j = numEntries - 1;
    while (j > 0 && board[j-1].getScore( ) < newScore) {
        board[j] = board[j-1]; // shift entry from j-1 to j
        j--; // and decrement j
    }
    board[j] = e; // when done, add new entry
}
}
```

17

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 $board[i + 1], \ldots, board[n - 1]$



18

Java Example

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

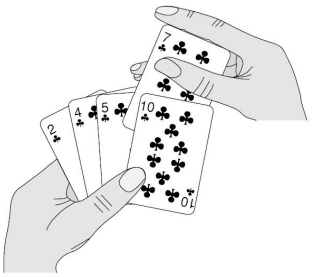
19

Sorting an Array

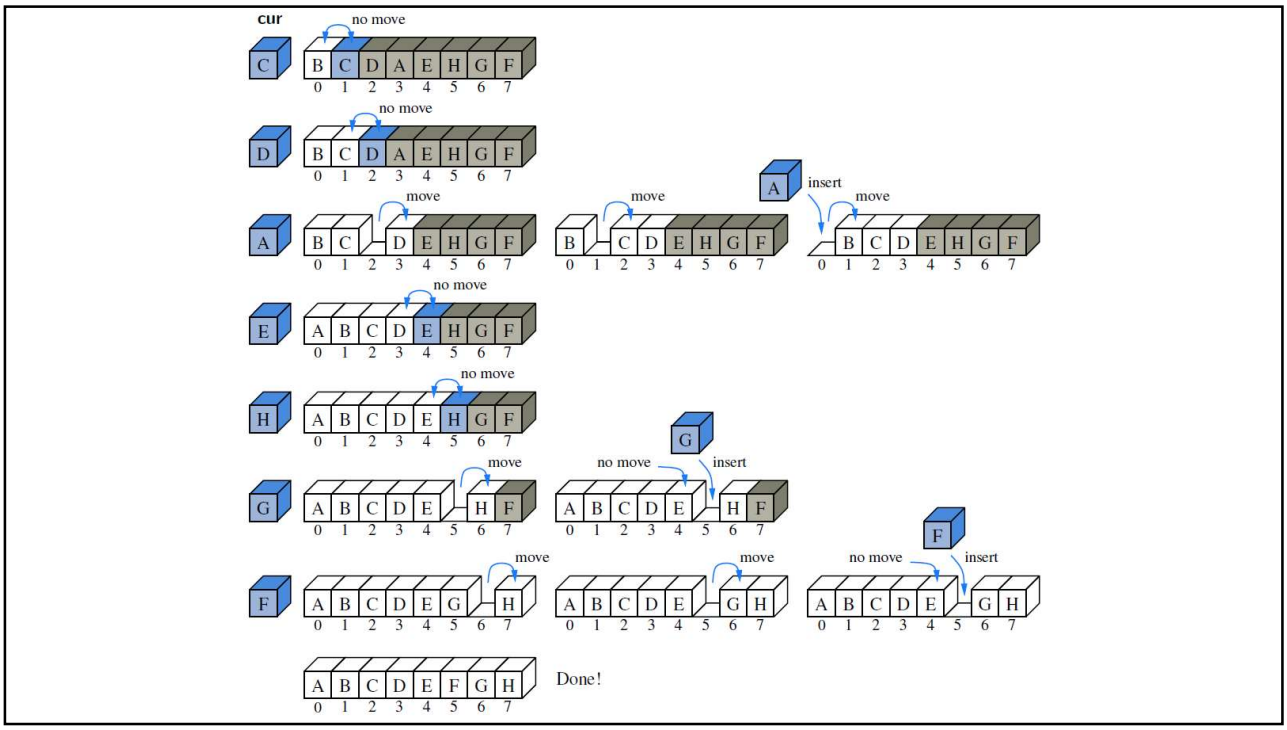
- Insertion-Sort Algorithm

Algorithm InsertionSort(*A*):
Input: *An array A of n comparable elements*
Output: *The array A with elements rearranged in nondecreasing order*

for *k from 1 to n−1 do*
Insert *A[k] at its proper location within A[0], A[1], . . ., A[k].*



20



21

Insertion-Sort Algorithm

```
/** Insertion-sort of an array of characters into nondecreasing order */
public static void insertionSort(char[ ] data) {
int n = data.length;
for (int k = 1; k < n; k++) { // begin with second character

    char cur = data[k]; // time to insert cur=data[k]
    int j = k; // find correct index j for cur
    while (j > 0 && data[j-1] > cur) {
    // thus, data[j-1] must go after cur
        data[j] = data[j-1]; // slide data[j-1] rightward
        j--; // and consider previous j for cur
    }
    data[j] = cur; // this is the proper place for cur

}
}
```

22

java.util Methods for 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 values in the same order.
fill(*A, x*)	Stores value *x* in every cell of array *A*, provided the type of array *A* is defined so that it is allowed to store the value *x*.
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*, padded as with copyOf( ) if *t* > *A.length*.

23

java.util Methods for Arrays

toString(*A*)	Returns a String representation of the array *A*, beginning with [, ending with], and with elements of *A* displayed separated by string ", ". The string representation of an element *A[i]* is obtained using *String.valueOf(A[i])*, which returns the string "null" for a null reference and otherwise calls *A[i].toString( )*.
sort(*A*)	Sorts the array *A* based on a natural ordering of its elements, which must be comparable.
binarySearch(*A, x*)	Searches the sorted array *A* for value *x*, returning the index where it is found, or else the index of where it could be inserted while maintaining the sorted order.

As static methods, these are invoked directly on the java.util.Arrays class, not on a particular instance of the class. For example, if data were an array, we could sort it with syntax, java.util.Arrays.sort(data), or with the shorter syntax Arrays.sort(data) if we first import the Arrays class.

24

Random Numbers

- PseudoRandom Number Generation
  - Another feature built into Java, which is often useful when testing programs dealing with arrays, is the ability to generate pseudorandom numbers, that is, numbers that appear to be random (but are not necessarily truly random). In particular, Java has a built-in class, java.util.Random, whose instances are ***pseudorandom number generators, that is, objects that compute a sequence of numbers that are statistically*** random.

next = (a * cur + b) % n
where a, b, and n are appropriately chosen integers, and % is the modulus operator. Something along these lines is, in fact, the method used by java.util.Random objects, with n = 2^{48}

25

java.util.Random Methods

nextBoolean( )	Returns the next pseudorandom boolean value.
nextDouble( )	Returns the next pseudorandom double value, between 0.0 and 1.0.
nextInt( )	Returns the next pseudorandom int value.
nextInt(*n*)	Returns the next pseudorandom int value in the range from 0 up to but not including *n*.
setSeed(*s*)	Sets the seed of this pseudorandom number generator to the long *s*.

26

Multidimensional Array

- 2+ dimensional arrays are similar to the matrix representation.
- Each element can be accessed as a[i] [j]

```
int M = 10;
int N = 3;

double[][] a = new double[M][N];

for (int i = 0; i < M; i++) {
    for (int j = 0; j < N; j++) {

        a[i][j] = 0.0;

    }
}
```

2d array decleration

assign value to i, j th element

27

Declaring 2D Arrays

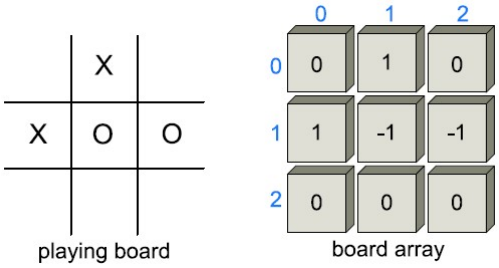
- Initializing 2D arrays by listing values.

```
double[][] p = {
    { .02, .92, .02, .02, .02 },
    { .02, .02, .32, .32, .32 },
    { .02, .02, .02, .92, .02 },
    { .92, .02, .02, .02, .02 },
    { .47, .02, .47, .02, .02 },
};
```

28

Two-Dimensional Arrays and Positional Games (Tic-Tac-Toe)

- Two players—X and O—alternate in placing their respective marks in the cells of this board, starting with player X. If either player succeeds in getting three of his or her marks in a row, column, or diagonal, then that player wins.



0 indicating an empty cell, a 1 indicating an X, and a −1 indicating an O. if the values of a row, column, or diagonal add up to 3 or −3, respectively, there is a win.

29

```
/** Simulation of a Tic-Tac-Toe game (does not do strategy). */
public class TicTacToe {
	public static final int X = 1, O = -1; // players
	public static final int EMPTY = 0; // empty cell
	private int board[ ][ ] = new int[3][3]; // game board
	private int player; // current player
	/** Constructor */
	public TicTacToe( ) { clearBoard( ); }
	/** Clears the board */
	public void clearBoard( ) {
		for (int i = 0; i < 3; i++)
			for (int j = 0; j < 3; j++)
				board[i][j] = EMPTY; // every cell should be empty
				player = X; // the first player is 'X'
	}
	/** Puts an X or O mark at position i,j. */
	public void putMark(int i, int j) throws IllegalArgumentException {
		if ((i < 0) || (i > 2) || (j < 0) || (j > 2))
			throw new IllegalArgumentException("Invalid board position");
		if (board[i][j] != EMPTY)
			throw new IllegalArgumentException("Board position occupied");
		board[i][j] = player; // place the mark for the current player
		player = - player; // switch players (uses fact that O = - X)
	} // Code continues on the next page
```

30

```
/** Checks whether the board configuration is a win for the given player. */
public boolean isWin(int mark) {
    return ((board[0][0] + board[0][1] + board[0][2] == mark*3) // row 0
    || (board[1][0] + board[1][1] + board[1][2] == mark*3) // row 1
    || (board[2][0] + board[2][1] + board[2][2] == mark*3) // row 2
    || (board[0][0] + board[1][0] + board[2][0] == mark*3) // column 0
    || (board[0][1] + board[1][1] + board[2][1] == mark*3) // column 1
    || (board[0][2] + board[1][2] + board[2][2] == mark*3) // column 2
    || (board[0][0] + board[1][1] + board[2][2] == mark*3) // diagonal
    || (board[2][0] + board[1][1] + board[0][2] == mark*3)); // rev diag
}
/** Returns the winning player's code, or 0 to indicate a tie (or unfinished game).*/
public int winner( ) {
    if (isWin(X))
        return(X);
    else if (isWin(O))
        return(O);
    else
        return(0);
}
```

31

```
/** Returns a simple character string showing the current board. */
public String toString( ) {
    StringBuilder sb = new StringBuilder( );
    for (int i=0; i<3; i++) {
        for (int j=0; j<3; j++) {
            switch (board[i][j]) {
                case X: sb.append("X"); break;
                case O: sb.append("O"); break;
                case EMPTY: sb.append(" "); break;
            }
            if (j < 2) sb.append("|"); // column boundary
        }
        if (i < 2) sb.append("\n-----\n"); // row boundary
    }
    return sb.toString( );
}
/** Test run of a simple game */
```

32

```
    /** Test run of a simple game */
    public static void main(String[ ] args) {
        TicTacToe game = new TicTacToe( );
        /* X moves: */ /* O moves: */
        game.putMark(1,1); game.putMark(0,2);
        game.putMark(2,2); game.putMark(0,0);
        game.putMark(0,1); game.putMark(2,1);
        game.putMark(1,2); game.putMark(1,0);
        game.putMark(2,0);
        System.out.println(game);
        int winningPlayer = game.winner( );
        String[ ] outcome = {"O wins", "Tie", "X wins"}; // rely on ordering
        System.out.println(outcome[1 + winningPlayer]);
    }
} // end of class
```

```
O|X|O
-----
O|X|X
-----
X|O|X
Tie
```

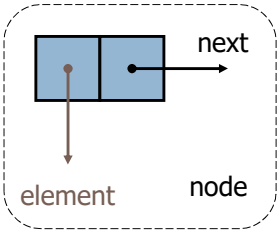
33

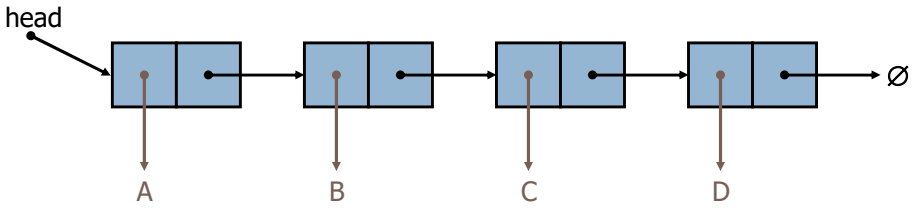
SINGLY LINKED LISTS

34

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





35

A Nested Node Class

```
public class SinglyLinkedList<E> {
  //---------------- nested Node class ----------------
  private static class Node<E> {
    private E element;                  // reference to the element stored at this node
    private Node<E> next;               // reference to the subsequent node in the list
    public Node(E e, Node<E> n) {
      element = e;
      next = n;
    }
    public E getElement( ) { return element; }
    public Node<E> getNext( ) { return next; }
    public void setNext(Node<E> n) { next = n; }
  } //---------- end of nested Node class ----------
  ... rest of SinglyLinkedList class will follow ...
```

36

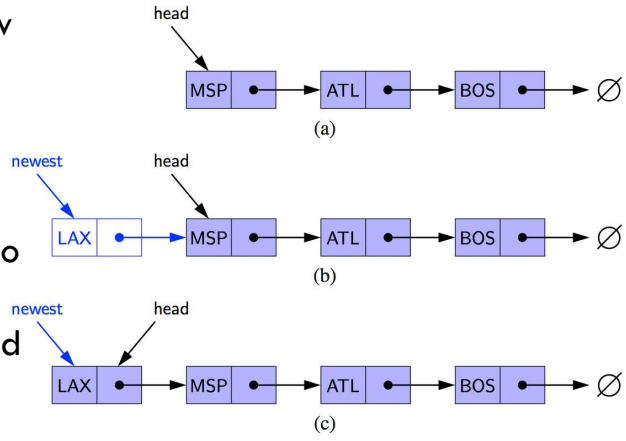
Accessor Methods

```
public class SinglyLinkedList<E> {
  (nested Node class goes here)
  // instance variables of the SinglyLinkedList
  private Node<E> head = null;          // head node of the list (or null if empty)
  private Node<E> tail = null;          // last node of the list (or null if empty)
  private int size = 0;                 // number of nodes in the list
  public SinglyLinkedList( ) { }        // constructs an initially empty list
  // access methods
  public int size( ) { return size; }
  public boolean isEmpty( ) { return size == 0; }
  public E first( ) {                   // returns (but does not remove) the first element
    if (isEmpty( )) return null;
    return head.getElement( );
  }
  public E last( ) {                    // returns (but does not remove) the last element
    if (isEmpty( )) return null;
    return tail.getElement( );
  }
```

37

Inserting at the Head

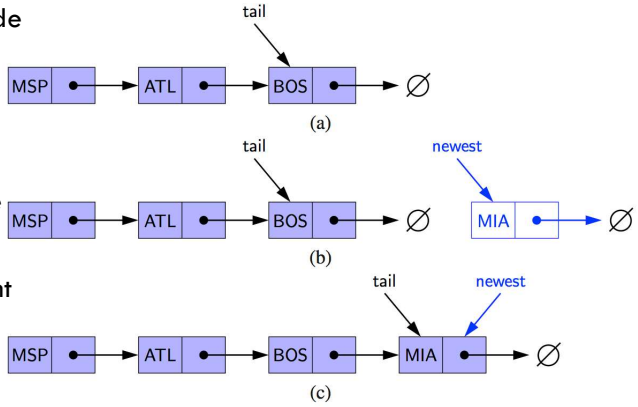
- Allocate new node
- Insert new element
- Have new node point to old head
- Update head to point to new node



38

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



39

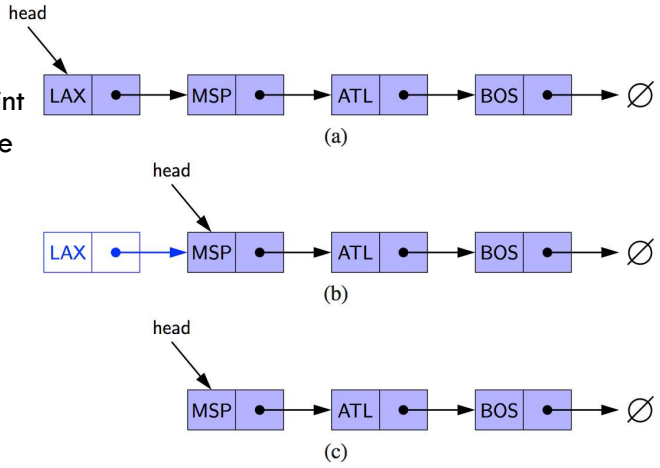
Java Methods

```
public void addFirst(E e) {              // adds element e to the front of the list
  head = new Node<>(e, head);            // create and link a new node
  if (size == 0)
    tail = head;                         // special case: new node becomes tail also
  size++;
}
public void addLast(E e) {               // adds element e to the end of the list
  Node<E> newest = new Node<>(e, null);  // node will eventually be the tail
  if (isEmpty())
    head = newest;                       // special case: previously empty list
  else
    tail.setNext(newest);                // new node after existing tail
  tail = newest;                         // new node becomes the tail
  size++;
}
```

40

Removing at the Head

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



41

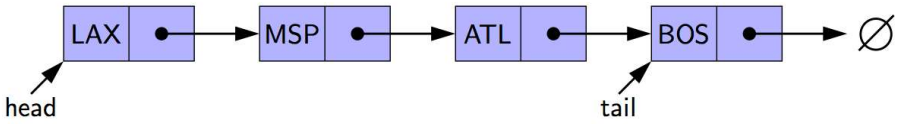
Java Method

```
public E removeFirst( ) {                  // removes and returns the first element
  if (isEmpty( )) return null;             // nothing to remove
  E answer = head.getElement( );
  head = head.getNext( );                  // will become null if list had only one node
  size−−;
  if (size == 0)
    tail = null;                           // special case as list is now empty
  return answer;
 }
}
```

42

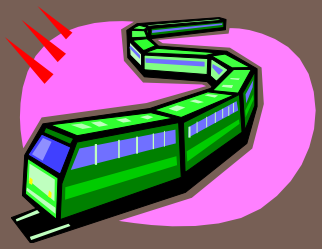
Removing at the Tail

- 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



43

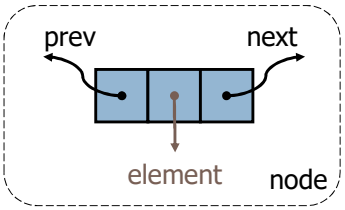
DOUBLY LINKED LISTS

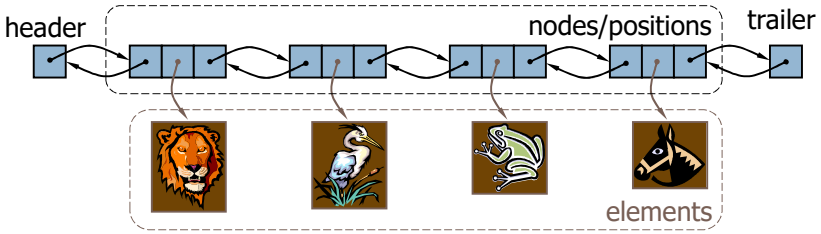


44

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

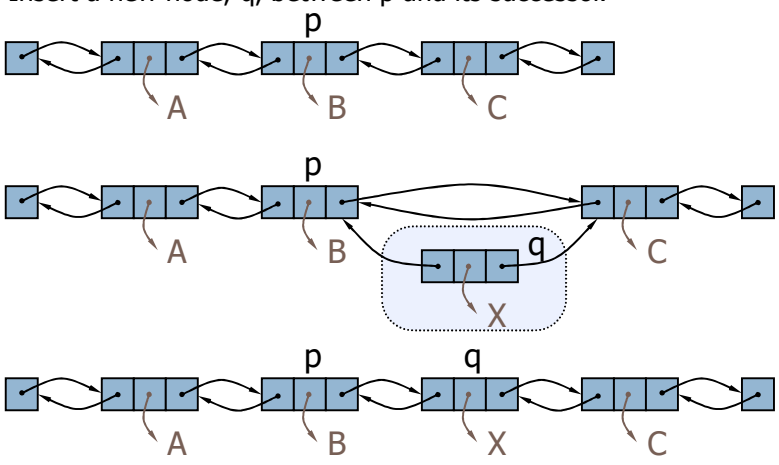




45

Insertion

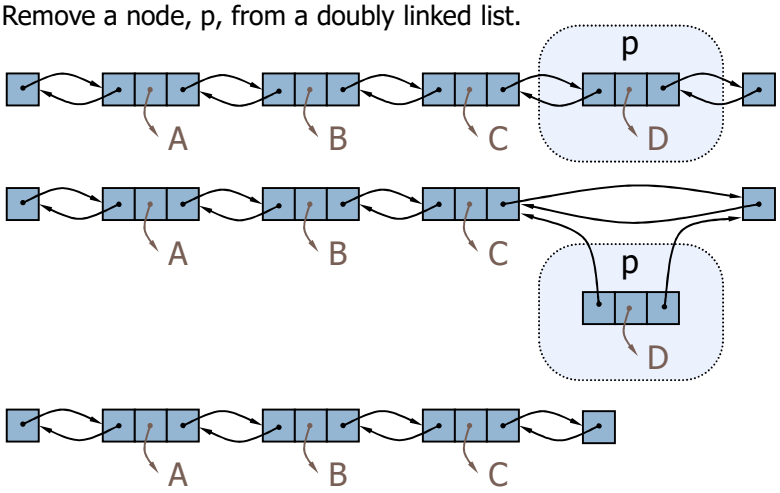
- Insert a new node, q, between p and its successor.



46

Deletion

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



47

Doubly-Linked List in Java

```
/** A basic doubly linked list implementation. */
public class DoublyLinkedList<E> {
  //---------------- nested Node class ----------------
  private static class Node<E> {
    private E element;                // reference to the element stored at this node
    private Node<E> prev;             // reference to the previous node in the list
    private Node<E> next;             // reference to the subsequent node in the list
    public Node(E e, Node<E> p, Node<E> n) {
      element = e;
      prev = p;
      next = n;
    }
    public E getElement( ) { return element; }
    public Node<E> getPrev( ) { return prev; }
    public Node<E> getNext( ) { return next; }
    public void setPrev(Node<E> p) { prev = p; }
    public void setNext(Node<E> n) { next = n; }
  } //---------- end of nested Node class ----------
```

48

Doubly-Linked List in Java, 2

```
private Node<E> header;                          // header sentinel
private Node<E> trailer;                         // trailer sentinel
private int size = 0;                            // number of elements in the list
/** Constructs a new empty list. */
public DoublyLinkedList( ) {
  header = new Node<>(null, null, null);         // create header
  trailer = new Node<>(null, header, null);      // trailer is preceded by header
  header.setNext(trailer);                       // header is followed by trailer
}
/** Returns the number of elements in the linked list. */
public int size( ) { return size; }
/** Tests whether the linked list is empty. */
public boolean isEmpty( ) { return size == 0; }
/** Returns (but does not remove) the first element of the list. */
public E first( ) {
  if (isEmpty( )) return null;
  return header.getNext( ).getElement( );        // first element is beyond header
}
/** Returns (but does not remove) the last element of the list. */
public E last( ) {
  if (isEmpty( )) return null;
  return trailer.getPrev( ).getElement( );       // last element is before trailer
}
```

49

Doubly-Linked List in Java, 3

```
// public update methods
/** Adds element e to the front of the list. */
public void addFirst(E e) {
  addBetween(e, header, header.getNext( ));       // place just after the header
}
/** Adds element e to the end of the list. */
public void addLast(E e) {
  addBetween(e, trailer.getPrev( ), trailer);     // place just before the trailer
}
/** Removes and returns the first element of the list. */
public E removeFirst( ) {
  if (isEmpty( )) return null;                    // nothing to remove
  return remove(header.getNext( ));               // first element is beyond header
}
/** Removes and returns the last element of the list. */
public E removeLast( ) {
  if (isEmpty( )) return null;                    // nothing to remove
  return remove(trailer.getPrev( ));              // last element is before trailer
}
```

50

Doubly-Linked List in Java, 4

```
  // private update methods
  /** Adds element e to the linked list in between the given nodes. */
  private void addBetween(E e, Node<E> predecessor, Node<E> successor) {
    // create and link a new node
    Node<E> newest = new Node<>(e, predecessor, successor);
    predecessor.setNext(newest);
    successor.setPrev(newest);
    size++;
  }
  /** Removes the given node from the list and returns its element. */
  private E remove(Node<E> node) {
    Node<E> predecessor = node.getPrev( );
    Node<E> successor = node.getNext( );
    predecessor.setNext(successor);
    successor.setPrev(predecessor);
    size--;
    return node.getElement( );
  }
} //---------- end of DoublyLinkedList class ----------
```

51

HW.

- P-2.31 Write a Java program to simulate an ecosystem containing two types of creatures, ***bears and fish. The ecosystem consists of a river, which is modeled as a relatively*** large array. Each cell of the array should contain an Animal object, which can be a Bear object, a Fish object, or null. In each time step, based on a random process, each animal either attempts to move into an adjacent array cell or stay where it is. If two animals of the same type are about to collide in the same cell, then they stay where they are, but they create a new instance of that type of animal, which is placed in a random empty (i.e., previously null) cell in the array. If a bear and a fish collide, however, then the fish dies (i.e., it disappears). Use actual object creation, via the new operator, to model the creation of new objects, and provide a visualization of the array after each time step.

52