

Lists

Lists

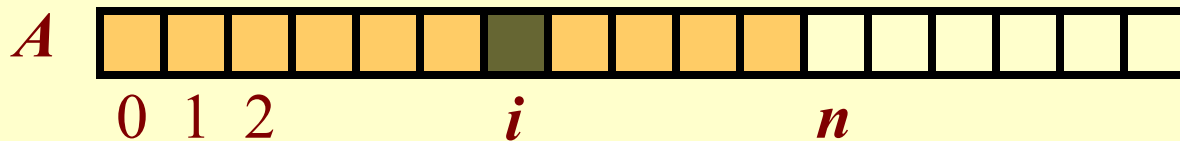
- Lists store arbitrary objects in linear structures.
- Arbitrary element access, insertion and removal are permitted.
- We will consider two distinct ways to access elements:

Index — an integer-valued location.

Link — a relative reference.

Array-based implementation

- Naturally, we use an array, say A .
- An element of index i is stored in $A[i]$.
- We also need an integer variable to keep track of the number of elements stored.



Array-based implementation

Pros.

- Easy to implement.
- Allow direct access to elements by indices.

Cons.

- Not suitable for frequent insertion and removal.
- Cannot naturally grow dynamically; to "grow," a new larger array is required.

Lists

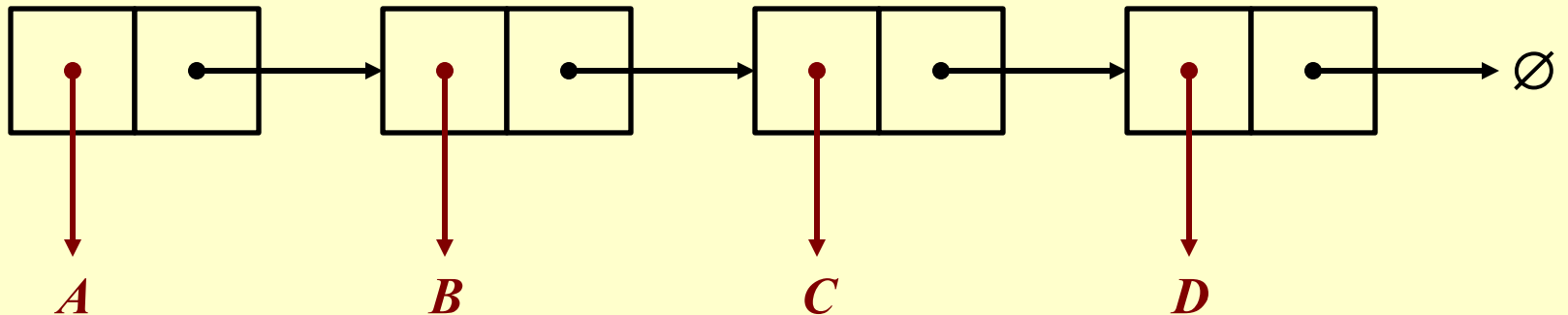
- Lists store arbitrary objects in linear structures.
- Arbitrary element access, insertion and removal are permitted.
- We will consider two distinct ways to access elements:

Index — an integer-valued location.

Link — a relative reference.

Linked lists

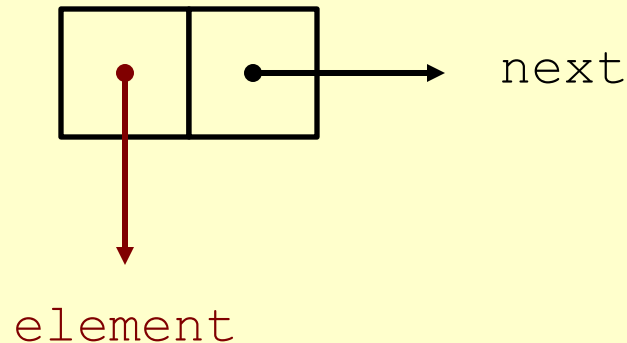
A *linked list* is a linear sequence of *nodes*.



Linked lists

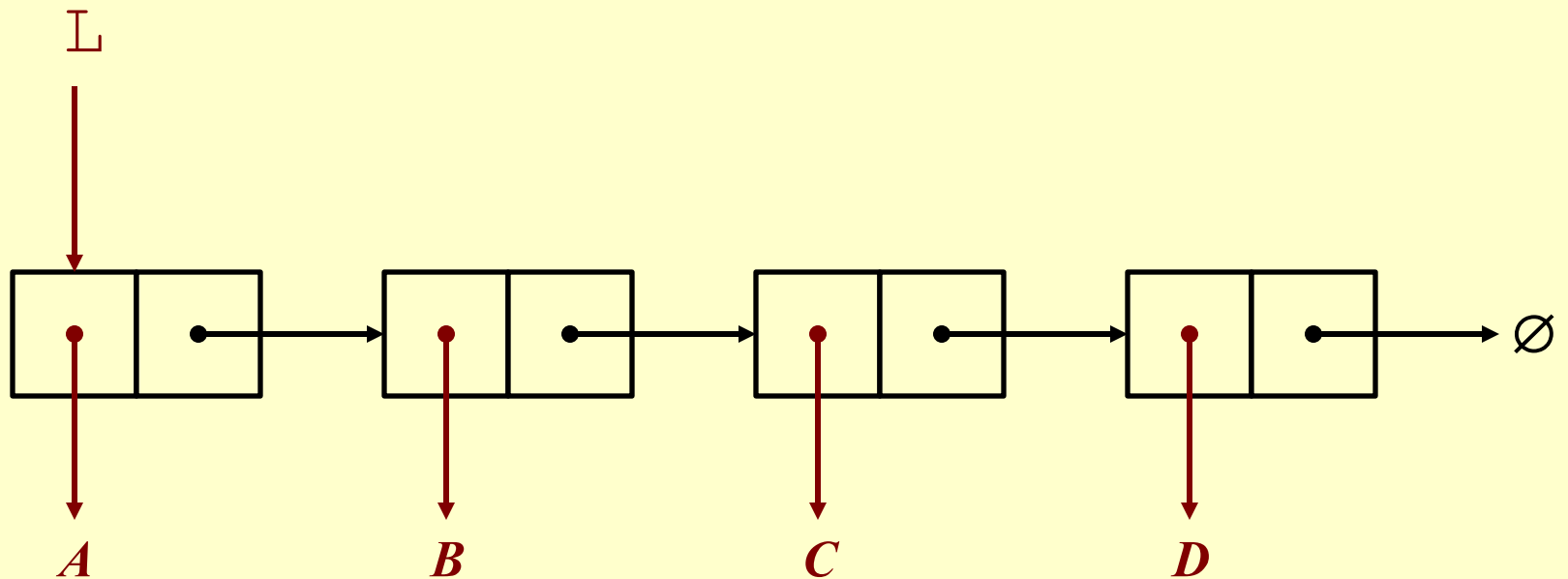
Each *node* has:

- *element*: a reference to an object.
- *next*: a reference to the next node.



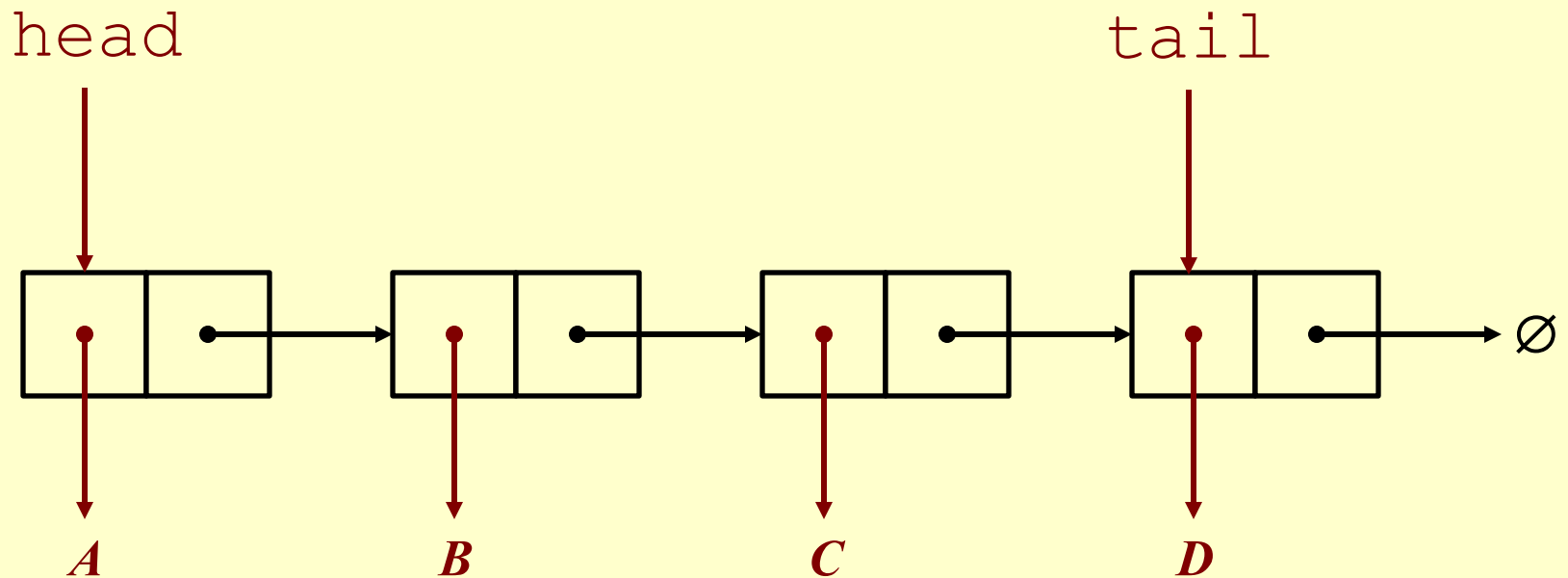
Linked lists

A *linked list* $\mathbb{L}$ is then identified by a reference to the first node.



Linked lists

It may also be identified by references to the first and last nodes.



Class Node for int

```
public class Node {  
  
    private int element;  
    private Node next;  
  
    public Node(int e, Node n) {  
        element = e;  
        next = n;  
    }  
  
    public int getElement() { return element; }  
    public Node getNext() { return next; }  
    public void setElement(int e) { element = e; }  
    public void setNext(Node n) { next = n; }  
  
}
```

Class Node for E

```
public class Node<E> {  
  
    private E element;  
    private Node<E> next;  
  
    public Node(E e, Node<E> n) {  
        element = e;  
        next = n;  
    }  
  
    public E getElement() { return element; }  
    public Node<E> getNext() { return next; }  
    public void setElement(E e) { element = e; }  
    public void setNext(Node<E> n) { next = n; }  
  
}
```

Class LinkedList

```
public class LinkedList<E> {  
  
    private Node<E> head;  
    private Node<E> tail;  
    private int size;  
  
    public LinkedList() {  
        head = null;  
        tail = null;  
        size = 0;  
    }  
  
    ...  
}
```

Class LinkedList

```
public class LinkedList<E> {  
  
    private Node<E> head;  
    private Node<E> tail;  
    private int size;  
  
    public Node<E> getHead() { return head; }  
    public Node<E> getTail() { return tail; }  
    public int getSize() { return size; }  
    public E get(Node<E> n) {  
        return n.getElement();  
    }  
    ...  
}
```

Class LinkedList

```
public class LinkedList<E> {  
    ...  
    public void insertFirst(E e) {  
        Node<E> x = new Node<E>(e, null);  
        if (head == null) head = tail = x;  
        else {  
            x.setNext(head);  
            head = x;  
        }  
        size++;  
    }  
    ...  
}
```

Class LinkedList

```
public class LinkedList<E> {  
  
    ...  
  
    public void insertAfter(Node<E> n, E e) {  
        Node<E> x = new Node<E>(e, null);  
        x.setNext(n.getNext());  
        n.setNext(x);  
        size++;  
    }  
  
    ...  
  
}
```

Class LinkedList

```
public class LinkedList<E> {  
  
    public E remove(Node<E> n) {  
        if (n == head) head = head.getNext();  
        else {  
            Node<E> x = head;  
            while (x.getNext() != n) x = x.getNext();  
            x.setNext(n.getNext());  
        }  
        size--;  
        return this.get(n);  
    }  
}
```

Linked lists

Example. To declare an empty linked list of integers named `numbers`, we put, using Java's class `Integer` for integers,

```
LinkedList<Integer> numbers = new LinkedList<Integer>();
```

This is an abbreviation of

```
LinkedList<Integer> numbers;  
numbers = new LinkedList<Integer>();
```

Linked lists

Example. To declare an empty linked list of dogs named `dogs`, we put, using Java's class `Dog` for dogs,

```
LinkedList<Dog> dogs = new LinkedList<Dog>();
```

This is an abbreviation of

```
LinkedList<Dog> dogs;  
dogs = new LinkedList<Dog>();
```

Linked lists

Example. A **while** loop to print all integers in the linked list `numbers`:

```
Node<Integer> n = numbers.getHead();  
while (n != null) {  
    System.out.println(numbers.get(n));  
    n = n.getNext();  
}
```

Linked lists

Example. A **while** loop to print all dogs' breeds in the linked list `dogs`:

```
Node<Dog> n = dogs.getHead();  
while (n != null) {  
    System.out.println(dogs.get(n).getBreed());  
    n = n.getNext();  
}
```

Searching

Example. A **while** loop to find the *maximum* integer in the linked list `numbers`:

```
int maxSoFar = MININT;
Node<Integer> n = numbers.getHead();
while (n != null) {
    if (numbers.get(n) > maxSoFar)
        maxSoFar = numbers.get(n);
    n = n.getNext();
}
System.out.println("Max: " + maxSoFar);
```

Searching

Example. A **while** loop to find the *oldest age* of a dog in the linked list `dogs`:

```
int oldestSoFar = 0;
Node<Dog> n = dogs.getHead();
while (n != null) {
    if (dogs.get(n).getAge() > oldestSoFar)
        oldestSoFar = dogs.get(n).getAge();
    n = n.getNext();
}
System.out.println("Oldest age: " + oldestSoFar);
```

Searching

Example. A **while** loop to find the *heaviest weight* of a dog in the linked list `dogs`:

```
int heaviestSoFar = 0;
Node<Dog> n = dogs.getHead();
while (n != null) {
    if (dogs.get(n).getWeight() > heaviestSoFar)
        heaviestSoFar = dogs.get(n).getWeight();
    n = n.getNext();
}
System.out.println("Heaviest weight: " + heaviestSoFar);
```