# Data Structures

Week 10 – Sentinel list, doubly linked list, circular list

Dr. Ferenc Koczka · Eszterházy Károly University · 2026

# Review

- **Singly linked list:** Head → … → Nil; sequential access
- At the beginning and **after** a known element: O(1); at the end and search: O(n)
- **Sentinel search in an array** (week 2): the value we look for is put at the end → one condition is enough

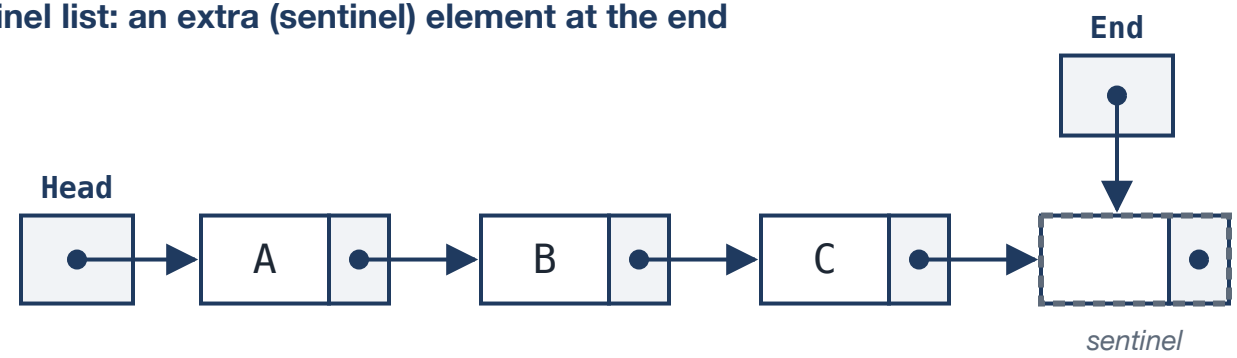
## Problems of the singly linked list

- Many **special cases**: empty list, first element, last element
- The search loop has **two conditions**: `P ≠ Nil and Elem[P].Value ≠ x`
- We cannot insert **before** an element, and we cannot delete **the element itself** (only the one after it)

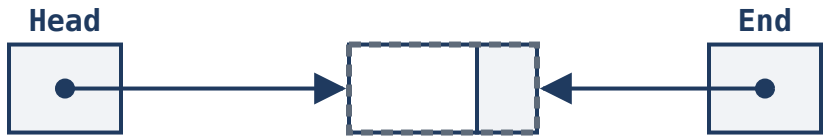
# Sentinel list

# The structure of a sentinel list

**Sentinel list: an extra (sentinel) element at the end**




**Empty sentinel list:**




**Head = End (storage is NOT zero!)**

- At the end of the list there is a **sentinel** (guard) element that stores no real data
- **Head:** points to the first element, **End:** points to the sentinel
- Empty list: **Head = End** – but even then **one element** (the sentinel) takes up space!

# Operations of the sentinel list

| Operation | Idea | Steps |
|---|---|---|
| **Create** | reserve a sentinel, Head ← End ← sentinel | O(1) |
| **Process** | from Head to End (not to Nil!) | O(n) |
| **Search** | the value is written into the sentinel | O(n) |
| **Insert before** | copying trick | O(1) |
| **Delete an element** | copying trick | O(1) |
| **Make empty** | Head ← End (the elements go to the free list) | O(1) – O(n) |
| **Properties** | **homogeneous · dynamic · scattered · sequential access** | |

# Creation and processing

**Creation**

```
Procedure Create(Head, End)
  If Reserve(P) then
    Head ← P
    End ← P
  else
    ERROR("no free memory")
  End if
End procedure
```

**Processing**

```
Procedure Process(Head, End)
  P ← Head
  While P ≠ End
    Operation(Elem[P].Value)
    P ← Elem[P].Next
  End while
End procedure
```

**Java**

```java
class SentinelList {
    Node head, end;

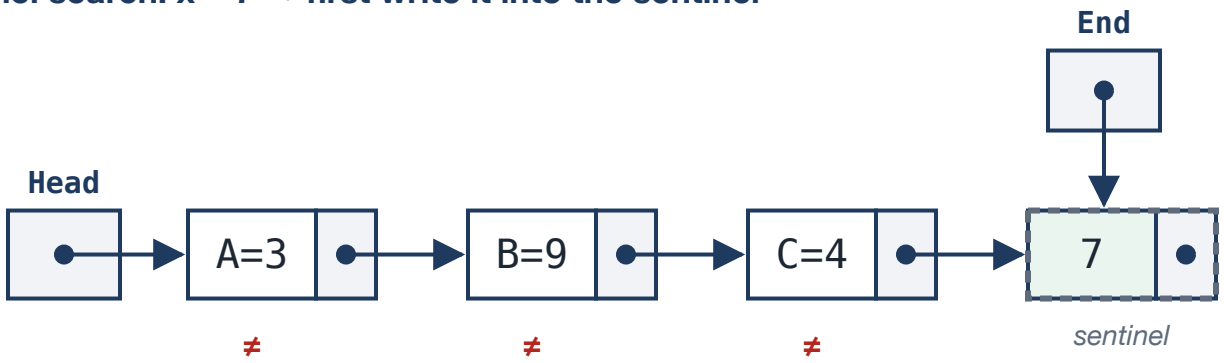
    SentinelList() {
        end = new Node();   // the sentinel
        head = end;
    }

    boolean isEmpty() { return head == end; }

    void print() {
        for (Node p = head; p != end; p = p.next)
            System.out.print(p.value + " ");
    }
}
```

# Sentinel search

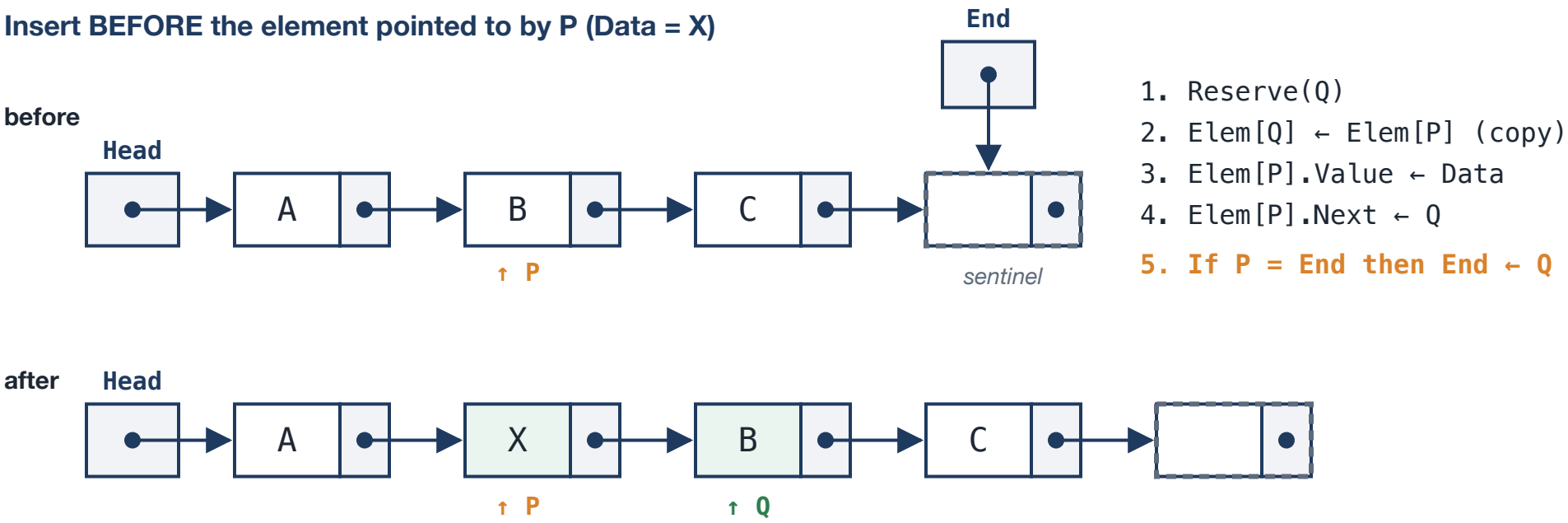
**Sentinel search: x = 7 → first write it into the sentinel**




**= → P = End: not found**

```
Function SentinelSearch(Head, End, x)
  Elem[End].Value ← x                  // written into the sentinel
  P ← Head
  While Elem[P].Value ≠ x              // a single condition!
    P ← Elem[P].Next
  End while
  If P = End then Return Nil  else Return P
End function
```

# Insertion before a given element




**Trick:** we cannot insert **before** P (we do not know the previous element), but we can insert **after** it. So we copy P's **content** into the new element, and write the new data into P.

# Insertion before – pseudocode and code

**Pseudocode**

```
Procedure InsertBefore(Head, End, P, Data)
  If Reserve(Q) then
    Elem[Q] ← Elem[P]       // Value and Next
    Elem[P].Value ← Data
    Elem[P].Next ← Q
    If P = End then
      End ← Q               // Q is the new sentinel
    End if
  else
    ERROR("no free memory")
  End if
End procedure
```

**Insertion at the end** = insertion before the sentinel:

`InsertBefore(Head, End, End, Data)` → **O(1)**!

**Insertion at the beginning** = insertion before Head

**Java**

```java
void insertBefore(Node p, int x) {
    Node q = new Node();
    q.value = p.value;
    q.next = p.next;
    p.value = x;
    p.next = q;
    if (p == end) end = q;
}

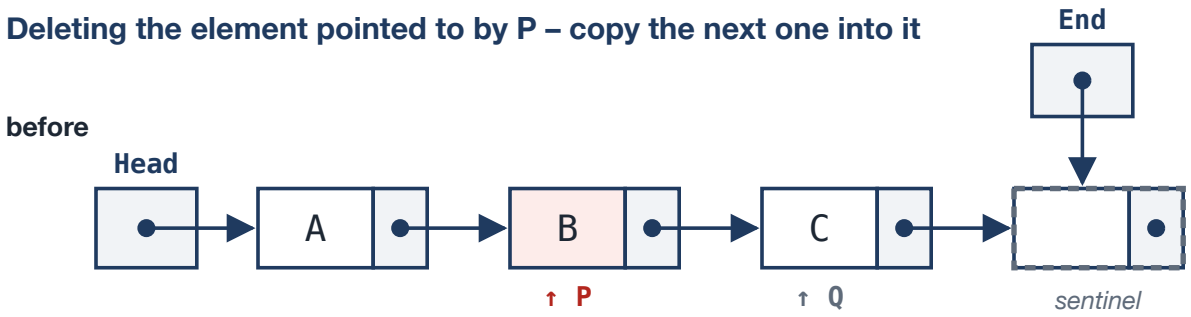
void insertLast(int x)  { insertBefore(end, x); }
void insertFirst(int x) { insertBefore(head, x); }
```

**Go**

```go
func (l *SentinelList) InsertBefore(p *Node, x int) {
    q := &Node{Value: p.Value, Next: p.Next}
    p.Value = x
    p.Next = q
    if p == l.End {
        l.End = q
    }
}
```

# Deleting a given element

**Deleting the element pointed to by P – copy the next one into it**




1. `Q ← Elem[P].Next`
2. `Elem[P] ← Elem[Q]` (C into B's place)
3. **`If Q = End then End ← P`**
4. `Free(Q)`

```
Procedure Delete(Head, End, P)        // P ≠ End
  Q ← Elem[P].Next
  Elem[P] ← Elem[Q]                   // the content of the next element into P
  If Q = End then End ← P             // P is the new sentinel
  Free(Q)
End procedure
```

Deleting the **first** element is not a special case either – Head does not change!

# What did we gain with the sentinel?

| | Singly linked list | Sentinel list |
|---|---|---|
| Empty list | Head = Nil | Head = End (space: 1 element) |
| Loop condition of the search | 2 conditions | **1 condition** |
| Insertion before | impossible (only after) | **O(1)** |
| Deleting a given element | impossible (only the one after) | **O(1)** |
| Insertion at the end | O(n) | **O(1)** |
| Special cases (first element) | many | **none** |

The copying trick **changes which element stores which value** – if outside pointers point to the elements, they may “slip”. For large records copying is expensive too.

# Sorted sentinel list

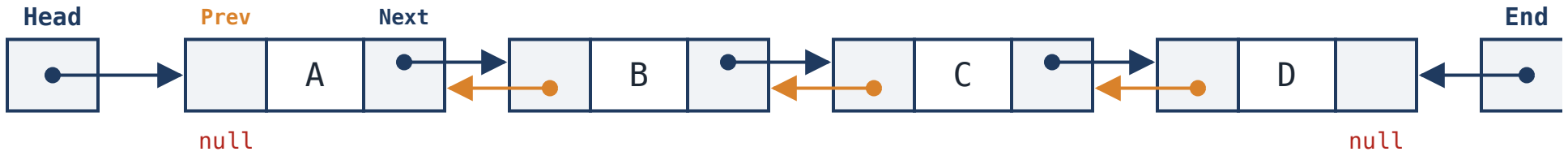
The sentinel also simplifies sorted insertion: we write an **infinitely large** value into the sentinel, so the search certainly stops.

```
Procedure SortedInsert(Head, End, Data)
  Elem[End].Value ← +∞
  P ← Head
  While Elem[P].Value < Data
    P ← Elem[P].Next
  End while
  InsertBefore(Head, End, P, Data)
End procedure
```

After inserting `5, 2, 8, 1, 9, 3`: `[1 2 3 5 8 9]` – the same code for an empty list, the beginning and the end!

# Doubly linked list

# The structure of a doubly linked list




*each element stores two pointers: previous and next*

- Every element stores the address of the **previous** and of the **next** element
- **Head** and **End** pointers: the list can be traversed in both directions
- Space: **two** pointers per element

# Doubly linked list – operations

**Deleting element P – O(1)**

```
Procedure Delete(Head, End, P)
  If Elem[P].Prev ≠ Nil then
    Elem[Elem[P].Prev].Next ← Elem[P].Next
  else
    Head ← Elem[P].Next
  End if
  If Elem[P].Next ≠ Nil then
    Elem[Elem[P].Next].Prev ← Elem[P].Prev
  else
    End ← Elem[P].Prev
  End if
  Free(P)
End procedure
```

No copying needed – **we know both neighbours**.

Insertion before/after, deletion from the end: all **O(1)**.

**Java**

```java
void remove(Node p) {
    if (p.prev != null) p.prev.next = p.next;
    else head = p.next;
    if (p.next != null) p.next.prev = p.prev;
    else tail = p.prev;
}

void insertLast(int x) {          // O(1)
    Node n = new Node(x);
    n.prev = tail;
    if (tail != null) tail.next = n;
    else head = n;
    tail = n;
}
```

# The doubly linked list in practice

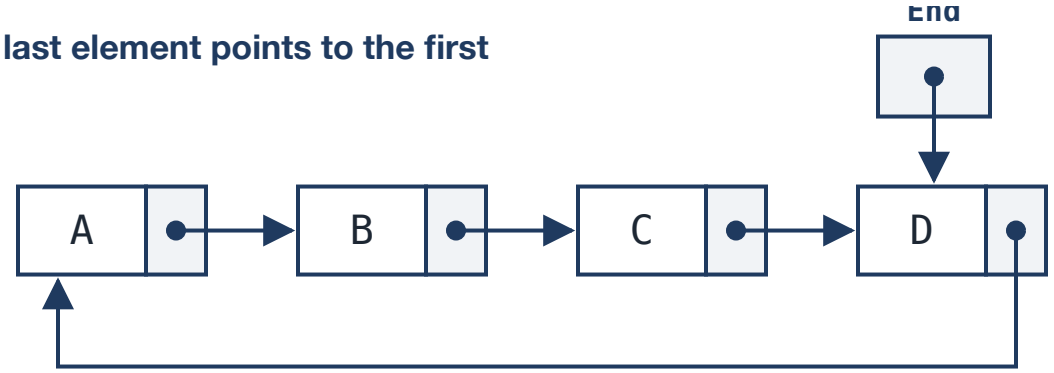
- Java `LinkedList`: a doubly linked list – `addFirst`, `addLast`, `removeFirst`, `removeLast` are all O(1)
- Go `container/list`: doubly linked, sentinel-based, circular list
- **Applications:**
  - browser history (back **and** forward)
  - music player: previous / next song
  - text editor: moving the cursor between lines
  - **LRU cache:** fast removal of the least recently used element (combined with a hash table)

> **Doubly linked sentinel list:** a single sentinel that is both the element before the first and after the last – so **every** operation can be written without special cases (Go's `container/list` works this way).

# Circular list

# Circular list

**Circular list: the last element points to the first**




```
first = Elem[End].Next
```

**no Nil in the chain!**

- The last element points not to **Nil** but to the **first** element
- A single pointer to the **last** element is enough: first = `Elem[End].Next` → insertion at the beginning **and** at the end are both O(1)
- Traversal: from the first element until we **get back** (post-test loop!)

**Applications:** round-robin scheduling, a looping playlist, the Josephus problem, turn-based multiplayer games

# Comparing the lists

| | Singly | Sentinel | Doubly | Circular |
|---|---|---|---|---|
| Pointers per element | 1 | 1 | **2** | 1 |
| Space of an empty list | 0 elements | **1 element** | 0 elements | 0 elements |
| Insertion at the beginning | O(1) | O(1) | O(1) | O(1) |
| Insertion at the end | O(n) | **O(1)** | **O(1)** | **O(1)** |
| Deletion from the end | O(n) | O(n) | **O(1)** | O(n) |
| Deleting a given element | O(n)* | **O(1)** | **O(1)** | O(n)* |
| Backward traversal | ✗ | ✗ | ✓ | ✗ |

* the previous element must be found

# Summary

- **Sentinel list:** Head + End, End points to an empty sentinel element; empty list: Head = End
  - sentinel search with **one** condition · insert before and delete with the **copying trick** in O(1) · no special cases
- **Doubly linked list:** Prev + Next pointers; traversable in both directions; deletion O(1)
- **Circular list:** the last element points to the first; one End pointer is enough
- The choice depends on the **most frequent operations** and on the **space needed**

**Next week:** stack and queue based on lists, and the hash table.

# Review questions

1. True or false: an empty sentinel list needs no space, since no element is stored in it.
2. True or false: the space needed by a sentinel list and by a doubly linked list storing the same sequence is always the same, since both are given by two pointers (Head and End).
3. Write the pseudocode of the sentinel search! The list is given by the Head and End pointers.
4. Draw the sentinel list `[3, 7]`, and its state after inserting before `7` (Data = 5)!
5. Why do we need a post-test loop to traverse a circular list?
6. Which list would you choose if you often have to delete from the **end** of the list?

# Practice

# ✎ Demo programs

| Program | Java | Go | What does it show? |
|---|---|---|---|
| Sentinel list | `SentinelList.java` | `w10/sentinellist` | creation, sentinel search, insert before, delete, sorted insertion |
| Doubly linked list | `DoublyLinkedList.java` | `w10/doublylist` | forward/backward traversal, O(1) deletion, built-in list |
| Circular list | `CircularList.java` | `w10/circularlist` | traversal, the Josephus problem |

`SentinelList` is a complete solution of the sentinel list part of the old assignment (initialization, processing, search, insertion, deletion).

# ✎ Exercises

1. **Maximum:** find the largest element of a sentinel list!
2. **Counting with a sentinel:** how many times does **x** occur? The inner loop should have only one condition!
3. **Deleting the even elements:** delete every even element of a sentinel list with the copying trick!
4. **Palindrome:** on a doubly linked list, with two pointers from the two ends!
5. **Size of a circular list:** count the elements of a circular list!

Solutions: `Exercises10.java`, `w10/exercises`

# ✎ Homework

1. Implement a **doubly linked sentinel** list with a single sentinel (the element before the first and after the last is the same)! Write insert before, insert after and delete – without a single `if`!
2. Merge two sorted sentinel lists into a third sorted sentinel list!
3. Implement an **LRU cache** with a doubly linked list: the most recently used element goes to the front; when full, delete from the end!
4. Measure: from how many elements is the sentinel search noticeably faster than the simple search?