Data Structures

Week 9 – Dynamic data structures: a model of memory management and the singly linked list

Dr. Ferenc Koczka · Eszterházy Károly University · 2026

Review

- So far our data structures were based on **arrays**: contiguous storage, direct access
- Stack, queue: we work at the **ends** of the array → O(1)
- **Pointer / reference** (week 1): a data item that stores a memory address

Questions for today

- What are the limits of static (array-based) storage?
- How does a program allocate and free memory?
- How is a **linked list** built, and how fast are its operations?

The limits of static data structures

- The size must be given **in advance** → too small or wasteful
- A **contiguous** free memory area is needed – for a large array this is not always available
- Insertion and deletion in the middle are **O(n)** – elements must be moved
- A dynamic array (week 3) helps, but on resizing **everything has to be copied**

> **Idea:** store the elements **scattered**, anywhere in memory, and let every element **know where the next one is**.

A model of dynamic memory management

A simple model

- **Memory** is a large array whose elements are records with a **Value** and a **Next** field
- A **pointer** is an **index** into this array – the special value **Nil** means “points nowhere”
- Notation: `Elem[P].Value`, `Elem[P].Next` – the fields of the element pointed to by P

index	Value	Next
1	C	5
2		4
3	A	7
4		6
5	D	Nil
6		8
7	B	1
8		Nil



The list (from Head):

`3 → 7 → 1 → 5 → Nil`

`A → B → C → D`

Free list:

`2 → 4 → 6 → 8 → Nil`

Here a 'pointer' is an array index – in real memory it is an address.

Reserving and freeing

The free slots are also kept in a **list** (FreeList).

```
Function Reserve(P)
  If FreeList = Nil then
    Return false        // memory is full
  else
    P ← FreeList
    FreeList ← Elem[P].Next
    Return true
  End if
End function
```

```
Procedure Free(P)
  Elem[P].Next ← FreeList
  FreeList ← P
End procedure
```

- Both are **O(1)**
- We take from the **front** of the free list and put back there – just like a stack!

In practice free memory is managed by the operating system and the runtime – in the old assignment its implementation could also be omitted.

In reality

	Allocation	Freeing
C	`p = malloc(sizeof(Node))`	`free(p)` – the programmer's responsibility
Java	`new Node(x)`	**automatic** – garbage collector (GC)
Go	`&Node{Value: x}` or `new(Node)`	**automatic** – garbage collector (GC)

Typical mistakes with manual memory management:

- **Memory leak:** allocated memory is never freed → the program uses more and more memory
- **Dangling pointer:** we point to freed memory and use it
- The garbage collector prevents these: whatever **nothing points to** any more is freed

Dynamic data structures: advantages and costs

Advantages

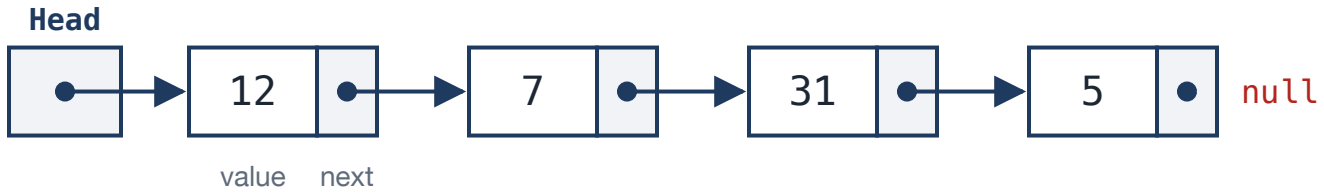
- They use as much memory as **currently needed** → memory can be **shared** between several data structures (resource sharing)
- No contiguous memory area is needed
- Insertion and deletion by **changing pointers**, without moving elements

Costs

- Every element also stores **one or more pointers** → more memory
- **No direct access** – to reach the **i**-th element we must walk through the previous ones
- Allocation/freeing and following pointers **take time**

The singly linked list

The structure of a linked list



- Every **element**: a value + a pointer to the **next** element
- **Head:** pointer to the first element · the `Next` field of the last element is **Nil**
- **Empty list:** Head = Nil

Property	Linked list
Elements	homogeneous
Size	**dynamic**
Storage	**scattered**
Access	**sequential**

A list element in Java and Go

Java

```java
class Node {
    int value;
    Node next;      // reference to the next one

    Node(int value, Node next) {
        this.value = value;
        this.next = next;
    }
}

Node head = null;   // empty list
```

Go

```go
type Node struct {
    Value int
    Next  *Node     // pointer to the next one
}

type List struct {
    Head *Node      // nil: empty list
}
```

Pseudocode	Java	Go
`Elem[P].Value` · `Elem[P].Next`	`p.value` · `p.next`	`p.Value` · `p.Next`
`Nil` · `Head`	`null` · `head`	`nil` · `l.Head`

Traversal and search

Traversal (processing)

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

Search

```
Function Search(Head, x)
  P ← Head
  While P ≠ Nil and Elem[P].Value ≠ x
    P ← Elem[P].Next
  End while
  Return P              // Nil if not found
End function
```

Java

```java
for (Node p = head; p != null; p = p.next)
    System.out.println(p.value);

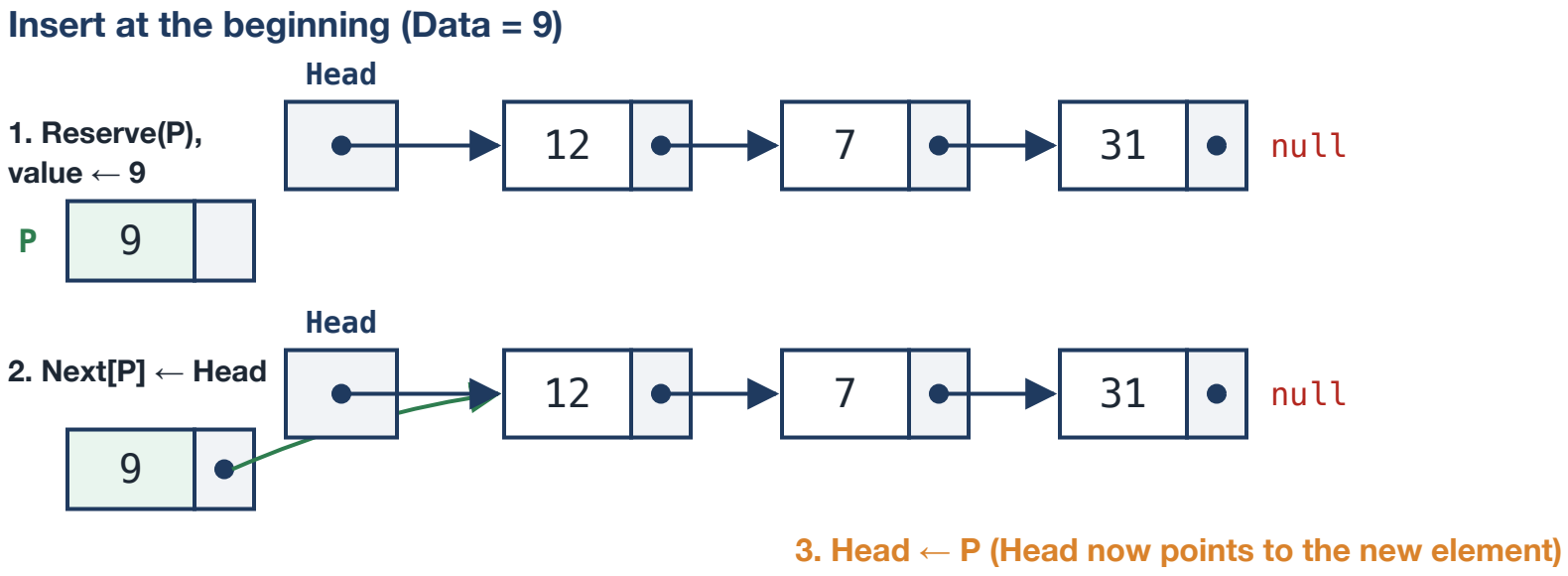
Node find(int x) {
    Node p = head;
    while (p != null && p.value != x)
        p = p.next;
    return p;
}
```

Go

```go
for p := l.Head; p != nil; p = p.Next {
    fmt.Println(p.Value)
}
```

Both are **O(n)**. No binary search even in a sorted list – there is no direct access!

Insertion at the beginning



Insertion at the beginning and deletion from the beginning

Insertion at the beginning – O(1)

```
Procedure InsertFirst(Head, Data)
  If Reserve(P) then
    Elem[P].Value ← Data
    Elem[P].Next ← Head
    Head ← P
  else
    ERROR("no free memory")
  End if
End procedure
```

Deletion from the beginning – O(1)

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

Java

```java
void insertFirst(int x) {
    head = new Node(x, head);
}

int removeFirst() {
    int x = head.value;
    head = head.next;   // the GC frees it
    return x;
}
```

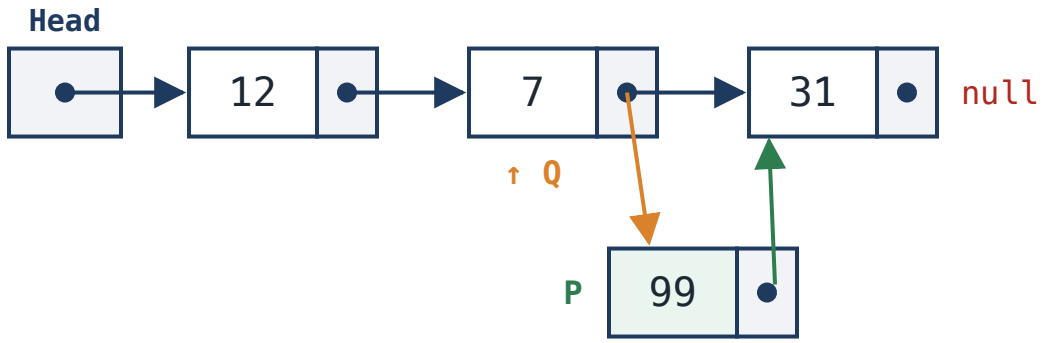
Go

```go
func (l *List) InsertFirst(x int) {
    l.Head = &Node{Value: x, Next: l.Head}
}

func (l *List) RemoveFirst() int {
    x := l.Head.Value
    l.Head = l.Head.Next
    return x
}
```

Insertion after a given element

Insert after the element pointed to by Q



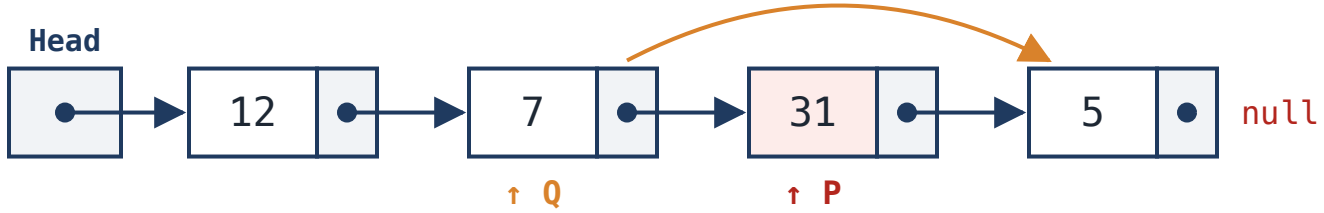
1. `Next[P] ← Next[Q]`
2. `Next[Q] ← P`

the order matters!

> In the reverse order (`Next[Q] ← P` first) we would **lose** the rest of the list – nothing would point to it any more!

Deletion after a given element

Delete after the element pointed to by Q



P ← Next[Q] ; Next[Q] ← Next[P] ; Free(P)

```
Procedure InsertAfter(Q, Data)
  If Reserve(P) then
    Elem[P].Value ← Data
    Elem[P].Next ← Elem[Q].Next
    Elem[Q].Next ← P
  End if
End procedure
```

```
Procedure DeleteAfter(Q)
  P ← Elem[Q].Next
  If P ≠ Nil then
    Elem[Q].Next ← Elem[P].Next
    Free(P)
  End if
End procedure
```

Both are **O(1)** – once we have Q. (Finding it is O(n).)

Operations at the end of the list

Insertion at the end – O(n)

```
Procedure InsertLast(Head, Data)
  If Reserve(P) then
    Elem[P].Value ← Data
    Elem[P].Next ← Nil
    If Head = Nil then
      Head ← P
    else
      Q ← Head
      While Elem[Q].Next ≠ Nil
        Q ← Elem[Q].Next
      End while
      Elem[Q].Next ← P
    End if
  End if
End procedure
```

Deletion from the end – O(n)

```
Procedure DeleteLast(Head)
  If Head = Nil then exit
  If Elem[Head].Next = Nil then
    Free(Head) ; Head ← Nil
  else
    Q ← Head      // we look for the last but one
    While Elem[Elem[Q].Next].Next ≠ Nil
      Q ← Elem[Q].Next
    End while
    Free(Elem[Q].Next)
    Elem[Q].Next ← Nil
  End if
End procedure
```

If we also keep an **End** pointer, insertion at the end becomes O(1) – deletion from the end does not (the last but one must still be found). Solution: doubly linked list (week 10).

Deleting an element by value

In a singly linked list we can only delete the element **after** a known one → while searching we must also remember the **previous** element.

```
boolean remove(int x) {
    Node prev = null, p = head;
    while (p != null && p.value != x) {
        prev = p;
        p = p.next;
    }
    if (p == null) return false;        // no such element
    if (prev == null) head = p.next;    // deleting the first element
    else prev.next = p.next;
    return true;
}
```

Always think of the **special cases**: empty list, the first element, the last element, a one-element list!

Linked list or array?

Operation	Array	Dynamic array	Linked list
accessing the i-th element	**O(1)**	**O(1)**	O(n)
search	O(n) / O(log n) sorted	O(n) / O(log n)	O(n)
insertion / deletion at the beginning	O(n)	O(n)	**O(1)**
insertion / deletion after a known element	O(n)	O(n)	**O(1)**
insertion at the end	O(1) if there is room	O(1) amortized	O(n), O(1) with an End pointer
space per element	value	value (+ reserve)	value + **pointer**

Measurement (inserting n = 160 000 elements at the beginning): `ArrayList` 870 ms · `LinkedList` 2 ms

Reading the middle element 20 000 times: `ArrayList` 0.1 ms · `LinkedList` 357 ms

Summary

- **Dynamic data structure:** elements scattered in memory, connected by pointers; its size changes at run time
- **Memory model:** array of records + free list; `Reserve(P)`, `Free(P)` – in Java/Go `new` and the garbage collector
- **Linked list:** Head → element → element → … → Nil; **sequential** access
- At the beginning and after a known element: insertion and deletion **O(1)** · search, at the end: **O(n)**
- The **order** of changing pointers matters · special cases: empty list, first element

Next week: sentinel list, doubly linked list, circular list.

Review questions

1. True or false: the use of dynamic data structures enables resource sharing.
2. True or false: the elements of a linked list can be accessed directly (by index).
3. True or false: improper use of dynamic data structures can result in memory leaks.
4. What happens if, when inserting after Q, first `Elem[Q].Next ← P` and then `Elem[P].Next ← Elem[Q].Next` is executed?
5. Write in pseudocode: insert the value `Data` at the **beginning** of the list given by the Head pointer! (`Reserve(P)` returns true if it succeeded.)
6. Why is deletion from the end of the list O(n), even if we know the End pointer?

Practice

✎ Demo programs

Program	Java	Go	What does it show?
Memory model	`MemoryModel.java`	`w09/memorymodel`	array of records, free list, Reserve/Free
Linked list	`SinglyLinkedList.java`	`w09/singlylist`	every basic operation
List vs. array	`ListVsArray.java`	–	insertion at the beginning, reading the middle element

The `MemoryModel` program implements exactly the pseudocode of the lecture – follow the printed tables!

✎ Exercises

1. **Sum and maximum:** compute them with a single traversal!
2. **i-th element:** return the **i**-th element of the list! How many steps does it take?
3. **Counting:** how many times does **x** occur in the list?
4. **Reversing in place:** reverse the list by changing the pointers, without creating new elements!
5. **Deleting every x:** delete every element with value **x**! (Pay attention to the ones at the beginning!)
6. **Middle element:** with a single traversal – one pointer moves one step, the other moves two!
7. **Sorted insertion:** insert so that the list stays in increasing order!

Solutions: `Exercises09.java`, `w09/exercises`

✎ Homework

1. Extend the list with an **End** pointer! Which operations become faster, and which must be modified?
2. Merge two **sorted** linked lists into one sorted list without creating new elements, in O(n + m) steps!
3. Decide whether a list contains a **cycle** (the `Next` field of some element points to an earlier element)! (Hint: hare and tortoise.)
4. Implement a stack with a linked list (push = insertion at the beginning, pop = deletion from the beginning)! Why is this better than working at the end?