Algorithms and Data Structures 1 Summary

VUB Computer Science

Academic Year 2025-2026

Contents

0 Introduction and Terminology (Hoofdstuk 0 & 0.5)

Note for readers

About 1/3 of this summary consists of code snippets which you do not have to know by heart, but you definitely do need to understand what's going on, for example on the written exam you'll be asked to write a sorting algorithm not implemented in lectures/WPOs, so solid knowledge on the workings of these procedures is essential. Good luck vro

0.1 Data and Data Constructors

Primitive vs. Compound Data

Primitive data values are data values that cannot be decomposed (in the same programming language). Examples: numbers, booleans, characters, symbols.
Compound data (also called *data structures*) is data constructed using a data constructor. The components are either primitive data values or other compound data.

Data Type

Each data value belongs to a set with a certain name: the **data type** of the value. Each data type determines which operations are applicable to its values.
Every data type has:

- **Constructors**: reserve memory and initialize components
- **Accessors** (getters): read components
- **Mutators** (setters): modify components

0.2 Procedure Types

Procedure Type Notation

Every Scheme procedure operates on inputs of certain data types and produces output of a certain data type:

$$\texttt{sin} : (\text{number} \to \text{number})$$
$$\texttt{car} : (\text{pair} \to \text{any})$$
$$\texttt{append} : (\text{pair} \times \text{pair} \to \text{pair})$$

0.3 Algorithms

Algorithm

An **algorithm** is a general and computable procedure for solving a problem. Named after Abu Abdullah Muhammad ibn Musa al-Khwarizmi.

0.4 Abstract Data Types (ADTs)

What is an ADT?

An **Abstract Data Type** is like a *contract* or *specification*. It tells you:

1. **What** operations you can do (the interface)
2. **Not how** they work internally (the implementation is hidden)

Analogy: Think of a TV remote control. You know you can press "Volume Up" and the volume increases. You don't need to know *how* the electronics inside make this happen. The buttons are the *interface*, the circuits are the *implementation*.

Users vs. Implementors

ADTs create a separation between two types of programmers:

- **Users:** Call the operations, don't care about internal details
- **Implementors:** Write the code that makes the operations work

Why is this useful?

- **Readability:** Code is easier to understand
- **Adaptability:** Can change implementation without breaking user code
- **Reusability:** Same ADT can be used in many programs

Genericity

A **generic data structure** can store any type of data. You don't write one stack for numbers and another for strings, you write one generic stack that works with anything.
In Scheme: We achieve this by passing an equality predicate as a parameter. Written as ADT$\langle V \rangle$ meaning "ADT that stores values of type V".

0.5 The Dictionary ADT, A Central Example

What is a Dictionary?

A **dictionary** (also called a "map" or "associative array") stores **key-value pairs**. Think of it like a real dictionary:

- **Key:** The word you look up (e.g., "algorithm")
- **Value:** The definition you get back

Operations:

- `insert!(dict, key, value)`, Add or update a key-value pair
- `find(dict, key)`, Look up the value for a key
- `delete!(dict, key)`, Remove a key-value pair

The Big Question: How do we implement these operations *efficiently*? This is one of the central questions of this course! We'll see many different implementations: lists, trees, hash tables...

0.6 Implementing ADTs in Scheme

0.6.1 Procedural Style

The simplest approach: represent data as pairs/lists and write separate functions.

```
; PROCEDURAL STYLE: Simple but representation is exposed
; Example: Complex numbers as (real . imaginary) pairs

; Constructor: creates a new complex number
(define (new-complex re im)
  (cons re im))              ; Just a pair (real . imag)

; Accessors: extract parts from the representation
(define (real-part c) (car c))  ; First element = real
(define (imag-part c) (cdr c))  ; Second element = imaginary

; Operations: work with the data through accessors
(define (complex-add c1 c2)
  (new-complex (+ (real-part c1) (real-part c2))
               (+ (imag-part c1) (imag-part c2))))
```

0.6.2 Record Style (define-record-type)

Cleaner syntax with auto-generated constructor, predicate, and accessors/mutators.

```
; RECORD STYLE: Scheme auto-generates accessors and mutators
; Syntax: (define-record-type NAME
;           (CONSTRUCTOR field1 field2 ...)
```

```
;           PREDICATE?
;           (field GETTER SETTER!) ...)

(define-record-type person
  (make-person name age salary)   ; Constructor
  person?                         ; Predicate: is this a person?
  (name person-name person-name!) ; Getter and setter for name
  (age person-age person-age!)    ; Getter and setter for age
  (salary person-salary person-salary!))

; Usage example:
(define p (make-person "Alice" 25 50000))
(person-name p)        ; => "Alice" (get name)
(person-age! p 26)     ; Mutate: set age to 26
```

0.6.3 Object-Oriented Style (Dispatcher)

Full encapsulation: data is hidden inside a closure, accessed only via messages.

```
; OO STYLE: Data hidden in closure, accessed via messages
; The constructor returns a PROCEDURE that handles messages

(define (new-counter)
  (let ((count 0))              ; Private state (hidden!)
    (lambda (msg)               ; Return a dispatcher function
      (cond
        ((eq? msg 'increment!)  ; Message: increment counter
         (set! count (+ count 1)))
        ((eq? msg 'get) count)  ; Message: get current value
        ((eq? msg 'reset!)      ; Message: reset to zero
         (set! count 0))
        (else (error "Unknown message"))))))

; Usage: send messages to the object
(define c (new-counter))  ; Create a counter "object"
(c 'increment!)           ; Send increment message
(c 'get)                  ; => 1 (get the value)
```

ADT Implementation Trade-offs

- **Procedural**: Simple, fast, but representation exposed (users can break it)
- **Records**: Good balance, Scheme generates code for you
- **Object-oriented**: Full encapsulation, but slower (dispatcher overhead)

0.7 Scheme Fundamentals for AD1

0.7.1 Named Let (Loops)

```
; Factorial using named let
(define (factorial n)
  (let loop ((n n) (acc 1))
    (if (= n 0)
        acc
        (loop (- n 1) (* n acc)))))

; Reverse list using named let
(define (my-reverse lst)
  (let loop ((lst lst) (acc '()))
    (if (null? lst)
        acc
        (loop (cdr lst) (cons (car lst) acc)))))
```

0.7.2 Vectors

```
; Vector creation
(define v1 (make-vector 5 0))      ; #(0 0 0 0 0)
(define v2 (vector 1 2 3 4 5))     ; #(1 2 3 4 5)

; Access and mutation - O(1)
(vector-ref v2 2)                  ; => 3
(vector-set! v2 2 99)              ; v2 = #(1 2 99 4 5)

; Vector length
(vector-length v2)                 ; => 5
```

0.7.3 Matrix Operations (Nested Vectors)

```
(define (make-matrix rows cols init)
  (let ((m (make-vector rows)))
    (let loop ((i 0))
      (if (< i rows)
          (begin
            (vector-set! m i (make-vector cols init))
            (loop (+ i 1)))))
    m))

(define (matrix-ref m i j)
  (vector-ref (vector-ref m i) j))

(define (matrix-set! m i j val)
  (vector-set! (vector-ref m i) j val))
```

1 Performance Analysis (Hoofdstuk 1)

1.1 Why Do We Care About Performance?

The Problem with Stopwatches

You could measure how fast your program runs with a stopwatch, but this has problems:

- Results depend on your computer's speed
- Results depend on what else is running
- Doesn't tell you how it scales to larger inputs

Better approach: Count the number of "steps" the algorithm takes as a function of input size n.

1.2 Asymptotic Notation, The Language of Efficiency

Big-O Notation: "At Most This Slow"

$O(g(n))$ means: "The algorithm takes **at most** $c \cdot g(n)$ steps for some constant c."
In normal words: "It won't be slower than this (ignoring constant factors)."
Examples:

- $O(1)$, Constant time (doesn't depend on input size)
- $O(\log n)$, Logarithmic (very fast, like binary search)
- $O(n)$, Linear (check each element once)
- $O(n^2)$, Quadratic (nested loops, gets slow fast!)
- $O(2^n)$, Exponential (impractical for $n > 30$)

Why Do We Ignore Constants?

We write $O(n)$ not $O(5n + 3)$ because:
- Constants depend on the computer/language
- For large n, the *shape* of growth matters more than the exact number
- An $O(n^2)$ algorithm with a tiny constant is still slower than $O(n)$ for big n

Key insight: We care about *scalability*, how does performance change as input grows?

The Three Notations

- $O(g(n))$, **Upper bound** (worst case, "at most")
- $\Omega(g(n))$, **Lower bound** (best case, "at least")
- $\Theta(g(n))$, **Tight bound** (both upper and lower, "exactly")

Analogy: If your commute is $O(1$ hour), it takes at most 1 hour. If it's $\Theta(30$ minutes), it always takes about 30 minutes.

Formal Definitions ***(niet te kennen voor het examen)***

Here are the formal set-theoretic definitions:
Big-O: $O(g(n)) = \{f \mid \exists c > 0, n_0 \geq 0 : \forall n \geq n_0 : 0 \leq f(n) \leq c \cdot g(n)\}$
Big-Ω: $\Omega(g(n)) = \{f \mid \exists c > 0, n_0 \geq 0 : \forall n \geq n_0 : 0 \leq c \cdot g(n) \leq f(n)\}$
Big-Θ: $\Theta(g(n)) = \{f \mid \exists c_1, c_2 > 0, n_0 \geq 0 : \forall n \geq n_0 : c_1 \cdot g(n) \leq f(n) \leq c_2 \cdot g(n)\}$
Translation: These say that from some point n_0 onwards, f is bounded above/below by a constant multiple of g.

Hierarchy of Complexity Classes

From fastest to slowest:

$$O(1) < O(\log n) < O(\sqrt{n}) < O(n) < O(n \log n) < O(n^2) < O(n^3) < O(2^n) < O(n!)$$

Practical guideline:
- $O(n)$ or better: Fast for any realistic input
- $O(n \log n)$: Standard for good sorting algorithms
- $O(n^2)$: Okay for small n, problematic for $n > 10000$
- $O(2^n)$: Only works for $n < 30$ or so

1.3 Analysis of Scheme Expressions

Complexity Rules

- Primitive operations (`+`, `car`, `vector-ref`): $O(1)$
- List/string conversions (`list->X`): $O(n)$
- `(if c t e)`: $O(1 + f_c(n) + f_t(n) + f_e(n))$
- `(begin e1 ... en)`: $O(1 + f_{e_1}(n) + \ldots + f_{e_n}(n))$

For recursive procedures: $O(b(n) \cdot r(n))$ where $b(n)$ = body complexity, $r(n)$ = recursion calls.

2 Strings and Pattern Matching (Hoofdstuk 2)

2.1 The Pattern Matching Problem

What is Pattern Matching?

The Problem: You have a long piece of text (like a book) and you want to find where a specific word or phrase appears.

Real-world examples:

- Finding "Ctrl+F" in a text editor
- Detecting a keyword in DNA sequences
- Searching for a song title in a playlist

In programming terms:

- **Text** t = the "haystack" (what we search *in*)
- **Pattern** p = the "needle" (what we search *for*)
- **Goal:** Return the position (index) where p starts in t, or `#f` if not found

Important Notation:

- $|s|$ or n_s = length of string s (number of characters)
- s_k = the character at position k (0-indexed)
- **Prefix:** The beginning part of a string. "hello" has prefixes: "", "h", "he", "hel", "hell", "hello"
- **Suffix:** The ending part of a string. "hello" has suffixes: "", "o", "lo", "llo", "ello", "hello"

2.2 Brute Force Algorithm

Idea: Try Every Position

The simplest approach: slide the pattern across the text one position at a time, and at each position, check if all characters match. **Analogy:** Like placing a ruler on a page and checking letter by letter if it matches, then sliding the ruler one position to the right and repeating. Glossary:

- $n_{t/p}$ = length of text or pattern
- $i_{t/p}$ = current index of text/pattern while we're searching
- $|t/p|$ = cardinality (= how many letters are in a string)
- $\leftarrow$ = value assignment similar to scheme definitions

Algorithm 1: Brute Force Pattern Matching (pseudocode, see scheme implementation below)

Input: Text t, Pattern p
Output: Position of first match or `#f`
$n_t \leftarrow |t|$, $n_p \leftarrow |p|$;
$i_t \leftarrow 0$, $i_p \leftarrow 0$;
while $i_p \leq n_p - 1$ *and* $i_t \leq n_t - n_p$ **do**
  if $t[i_t + i_p] = p[i_p]$ **then**
    $i_p \leftarrow i_p + 1$;
  else
    $i_t \leftarrow i_t + 1$;
    $i_p \leftarrow 0$;
  end
end
return i_t *if match found, else* `#f`;

Why is Brute Force Slow?

Worst case: $O(n_t \cdot n_p)$, At every position in the text, we might need to check almost all characters of the pattern before finding a mismatch.
Example of bad case: Text = "aaaaaaaab", Pattern = "aaab". At each position, we match "aaa" before failing on "b".

2.3 QuickSearch Algorithm

Idea: Skip Smarter!

Key insight: When we find a mismatch, instead of moving just one position to the right, we can sometimes skip multiple positions.
How? Look at the character in the text that is *just after* where the pattern would end. Then check: does this character even appear in our pattern?

- If **NO**: We can skip the entire pattern length! That character can't be part of any match.
- If **YES**: Align the pattern so that character matches its rightmost occurrence in the pattern.

The Shift Table

A precomputed table that tells us: for each possible character, how many positions can we safely shift?
Building it: For each character c in the pattern at position i, the shift is $n_p - i$ (distance from end). For characters not in the pattern, the shift is $n_p + 1$.

Algorithm 2: QuickSearch

```
Precompute shift table σ for each character;
while not matched and within bounds do
    if all characters match at current position then
        return current position;
    else
        c ← character just after pattern position in text;
        shift by σ(c) positions;
    end
end
```

Complexity: Worst-case still $O(n_t \cdot n_p)$, but **in practice often sublinear** (faster than linear) because we skip many positions.

2.4 Knuth-Morris-Pratt Algorithm

KMP: The Clever Algorithm

Key insight: When we find a mismatch after matching some characters, we've already learned something about the text! Don't throw away that information.
Analogy: If you're reading "ABCAB..." and looking for "ABCABD", when you fail at position 5 (expected D, got something else), you already know the text has "AB" at positions 3-4. So you don't need to restart completely, you can continue from there!

The Failure Function σ (Also called "Partial Match Table")

For each position i in the pattern, $\sigma(i)$ tells us: **what is the length of the longest proper prefix that is also a suffix of the substring** $p[0..i-1]$**?**
In simple terms: If we've matched i characters and then fail, $\sigma(i)$ tells us how many characters we can "reuse" without re-checking them.
Example: Pattern = "ABCABD"

- $\sigma(0) = -1$ (special case: nothing matched yet)
- $\sigma(1) = 0$ ("A" has no proper prefix that equals a suffix)
- $\sigma(4) = 1$ ("ABCA" $\rightarrow$ prefix "A" = suffix "A")
- $\sigma(5) = 2$ ("ABCAB" $\rightarrow$ prefix "AB" = suffix "AB")

KMP Complexity (Optimal)

- Building the failure table: $O(n_p)$
- Searching: $O(n_t)$
- **Total:** $O(n_t + n_p)$, Linear time!

This is optimal because we have to at least read every character once.

Algorithm	Worst Case	Best Case	Average
Brute Force	$O(n_t \cdot n_p)$	$O(n_t)$	$O(n_t \cdot n_p)$
QuickSearch	$O(n_t \cdot n_p)$	Sublinear	Fast in practice
KMP	$O(n_t + n_p)$	$O(n_t + n_p)$	$O(n_t + n_p)$

Table 1: Pattern Matching Algorithm Comparison

2.5 Scheme Implementations

The following implementations show how to translate the algorithms into working Scheme code. Study the comments to understand each step.

2.5.1 Brute Force Implementation

```
(define (brute-force text pattern)
  (define nt (string-length text))    ; Length of text
  (define np (string-length pattern)) ; Length of pattern

  ; Outer loop: try each possible starting position in text
  (let outer ((it 0))                 ; it = position in text
    (if (> it (- nt np))              ; Past the last valid position?
        #f                            ; Pattern not found
        ; Inner loop: compare characters at current position
        (let inner ((ip 0))           ; ip = position in pattern
          (cond
            ; All pattern chars matched? Success!
            ((= ip np) it)
            ; Current chars match? Check next char
            ((char=? (string-ref text (+ it ip))
                     (string-ref pattern ip))
             (inner (+ ip 1)))
            ; Mismatch: try next position in text
            (else (outer (+ it 1))))))))
```

2.5.2 QuickSearch Implementation

```
; Build shift table: for each character, how far can we skip?
(define (make-shift-table pattern)
  (define np (string-length pattern))
  ; Default shift: pattern length + 1 (char not in pattern)
  (define table (make-vector 256 (+ np 1)))
  ; For each char in pattern, compute its shift distance
  (let loop ((i 0))
    (if (< i np)
        (begin
          ; Shift = distance from this position to end of pattern
          (vector-set! table
                       (char->integer (string-ref pattern i))
                       (- np i))      ; Further right = smaller shift
          (loop (+ i 1)))))
  table)

(define (quicksearch text pattern)
  (define nt (string-length text))
  (define np (string-length pattern))
  (define shift-table (make-shift-table pattern))

  (let outer ((it 0))                 ; Current text position
    (if (> (+ it np) nt)              ; Would pattern go past text end?
        #f                            ; Not found
        (let inner ((ip 0))           ; Check each pattern char
          (cond
            ((= ip np) it)            ; Full match!
            ((char=? (string-ref text (+ it ip))
                     (string-ref pattern ip))
             (inner (+ ip 1)))        ; Match: check next
            ; Mismatch: look at char AFTER where pattern ends
            (else
             (let ((shift (vector-ref shift-table
                            (char->integer
                              (string-ref text (+ it np))))))
               (outer (+ it shift)))))))))  ; Skip ahead!
```

2.5.3 KMP Failure Function

```
; Build the "failure function" (partial match table)
; sigma[i] = length of longest proper prefix of pattern[0..i-1]
;            that is also a suffix
(define (compute-failure pattern)
  (define np (string-length pattern))
  (define sigma (make-vector np 0))   ; Will store failure values
  (vector-set! sigma 0 -1)            ; Special case: nothing matched

  ; k = length of current prefix-suffix match we're tracking
  (let loop ((ip 1) (k 0))
    (if (< ip np)
        (cond
          ; Current char extends the match
          ((char=? (string-ref pattern ip)
                   (string-ref pattern k))
           (vector-set! sigma ip (+ k 1))
           (loop (+ ip 1) (+ k 1)))
          ; No match, but k > 0: try shorter prefix
          ((> k 0)
           (loop ip (vector-ref sigma (- k 1))))
          ; No match and k = 0: no prefix-suffix here
          (else
           (vector-set! sigma ip 0)
           (loop (+ ip 1) 0)))))
  sigma)
```

2.5.4 KMP Match

```
(define (kmp-match text pattern)
  (define nt (string-length text))
  (define np (string-length pattern))
  (define sigma (compute-failure pattern))

  ; it = text position, ip = pattern position
  (let loop ((it 0) (ip 0))
    (cond
      ; Matched entire pattern!
      ((= ip np) (- it np))           ; Return start position
      ; Reached end of text without match
      ((>= it nt) #f)
      ; Characters match: advance both pointers
      ((char=? (string-ref text it)
               (string-ref pattern ip))
       (loop (+ it 1) (+ ip 1)))
      ; Mismatch after some matches: use failure function
      ((> ip 0)
       ; Don't restart! Use what we learned about the pattern
       (loop it (vector-ref sigma (- ip 1))))
      ; Mismatch at start: just move text pointer
      (else
       (loop (+ it 1) 0)))))
```

3 Linear Data Structures (Hoofdstuk 3)

3.1 Headed Lists and Vectors

Headed Data Structure

A data structure with a **header** containing:

- Reference to the actual data (list/vector)
- Equality predicate (for genericity)
- Optional: size, tail pointer

This solves problems with "naked" lists/vectors (fixed size, call-by-value issues).

3.2 Positional Lists ADT

A **positional list** allows you to access elements by their *position* (a reference to a specific node), not just by index.

Key operations: `first`, `last`, `next`, `previous`, `peek`, `update!`, `add-before!`, `add-after!`, `delete!`, `find`.

Operation	Vector	Singly Linked	Doubly Linked	DL + size/tail
`length`	$O(1)$	$O(n)$	$O(n)$	$O(1)$
`first/last`	$O(1)$	$O(1)/O(n)$	$O(1)/O(n)$	$O(1)$
`next/previous`	$O(1)$	$O(1)/O(n)$	$O(1)$	$O(1)$
`peek/update!`	$O(1)$	$O(1)$	$O(1)$	$O(1)$
`add-after!`	$O(n)$	$O(1)$	$O(1)$	$O(1)$
`add-before!`	$O(n)$	$O(n)$	$O(1)$	$O(1)$
`delete!`	$O(n)$	$O(n)$	$O(1)$	$O(1)$
`find`	$O(n)$	$O(n)$	$O(n)$	$O(n)$

Table 2: Positional List Implementation Comparison

Why These Complexities?

Vector (Array):
- `length`: Stored in header → $O(1)$
- `first/last/next/previous`: Direct indexing → $O(1)$
- `add/delete`: Must shift all subsequent elements → $O(n)$

Singly Linked List:
- `first`: Just follow head pointer → $O(1)$
- `last`: Must walk through entire list → $O(n)$
- `next`: Follow pointer → $O(1)$
- `previous`: No backward pointer! Must restart from head → $O(n)$
- `add-after`: Just rewire one pointer → $O(1)$
- `add-before/delete`: Need to find previous node first → $O(n)$

Doubly Linked List:
- `previous`: Now we have a backward pointer → $O(1)$
- `add-before/delete`: Can find previous directly → $O(1)$
- `last`: Still $O(n)$ unless we store a tail pointer

DL + size/tail: Store extra metadata in header → all navigation is $O(1)$

Memory Trade-Off

More pointers = faster operations, but uses more memory:
- Vector: $\Theta(n)$ cells (just data)
- Singly Linked: $\Theta(2n)$ cells (data + next pointer)
- Doubly Linked: $\Theta(3n)$ cells (data + next + previous pointers)

3.3 Searching in Linear Structures

Binary Search

For **sorted** data with $O(1)$ indexing (vectors):
- Compare with middle element
- Recursively search left or right half
- **Complexity:** $O(\log n)$

3.3.1 Headed List Implementation

```
; HEADED LIST: Wrapper around a plain Scheme list
; Stores the list data plus an equality predicate for genericity

(define-record-type headed-list
  (make-headed-list data eq)
  headed-list?
  (data data data!)     ; The actual list (mutable)
  (eq equality))        ; ==? predicate for comparing elements

; Create new empty headed list with given equality
(define (new ==?)
  (make-headed-list '() ==?))

; Check if list is empty
(define (empty? hlst)
  (null? (data hlst)))

; Add element to front - O(1)!
; Just cons to the front, no need to traverse
(define (add-to-front! hlst val)
  (data! hlst (cons val (data hlst)))
  hlst)
```

3.3.2 Binary Search Implementation

```
; BINARY SEARCH: Find element in SORTED vector in O(log n)
; Key insight: each comparison eliminates half the remaining elements

(define (binary-search vec key <<?)
  ; low/high define the current search range
  (let loop ((low 0) (high (- (vector-length vec) 1)))
    (if (> low high)
        #f                                ; Range empty = not found
        (let* ((mid (quotient (+ low high) 2))  ; Middle index
               (mid-val (vector-ref vec mid)))  ; Value at middle
          (cond
            ; key < mid-val? Search left half
            ((<<? key mid-val) (loop low (- mid 1)))
            ; key > mid-val? Search right half
            ((<<? mid-val key) (loop (+ mid 1) high))
            ; key == mid-val? Found it!
            (else mid))))))
```

3.3.3 Sorted List Implementation

```
; SORTED LIST: Elements always kept in order
; Insert is O(n) but find can use early termination

(define-record-type sorted-list
  (make current <<? ==?)
  sorted-list?
  (current current current!)  ; Head of the sorted list
  (<<? lesser)                ; < predicate for ordering
  (==? equality))             ; = predicate for equality

; ADD: Insert value in correct sorted position - O(n)
; Walk through list until we find where val belongs
(define (add! slst val)
  (define <<? (lesser slst))
  ; Track previous node so we can insert between prev and next
  (let loop ((prev '())
             (next (current slst)))
    (cond
      ; Found the right spot? (end of list OR val < next element)
      ((or (null? next) (<<? val (car next)))
       (if (null? prev)
           (current! slst (cons val next))  ; Insert at front
           (set-cdr! prev (cons val next)))) ; Insert between prev and next
      ; Not yet, keep searching
      (else (loop next (cdr next)))))
  slst)
```

4 Linear ADTs (Hoofdstuk 4)

4.1 Stacks (LIFO)

What is a Stack?

A **stack** is like a stack of plates: you can only add or remove from the **top**.
LIFO = Last In, First Out: The last thing you put on is the first thing you take off.
Real-world examples:

- **Undo button:** Each action is "pushed" onto a stack; undo "pops" the most recent one
- **Browser back button:** Previous pages are stacked; going back pops the stack
- **Function calls:** When a function calls another, the return address is pushed onto the "call stack"

Operations:

- `push!(stack, val)`, Add element to top
- `pop!(stack)`, Remove and return top element
- `top(stack)`, Look at top element (without removing)
- `empty?(stack)`, Is the stack empty?

All operations are $O(1)$, constant time!

4.1.1 Vector Implementation

Uses a vector with an index pointing to the top element.

```
; STACK (Vector Implementation)
; Uses a fixed-size vector with a "top pointer" (index)
; top-idx = -1 means empty, top-idx = n-1 means full

(define-record-type stack
  (make storage top-idx)
  stack?
  (storage storage)          ; The vector holding elements
  (top-idx top-idx top-idx!)) ; Index of top element (-1 if empty)

(define (new size)
  (make (make-vector size) -1))  ; Start with empty stack

(define (empty? s) (= (top-idx s) -1))
(define (full? s) (= (top-idx s) (- (vector-length (storage s)) 1)))

(define (push! s val)
  (if (full? s) (error "stack full"))
  (top-idx! s (+ (top-idx s) 1))  ; Move pointer up
  (vector-set! (storage s) (top-idx s) val)  ; Store value
  s)

(define (pop! s)
  (if (empty? s) (error "stack empty"))
  (let ((val (vector-ref (storage s) (top-idx s))))
    (top-idx! s (- (top-idx s) 1))  ; Move pointer down
    val))                           ; Return the value

(define (top s)
  (if (empty? s) (error "stack empty"))
  (vector-ref (storage s) (top-idx s)))  ; Just peek, don't remove
```

4.1.2 Linked Implementation

Uses a linked list; the "top" is just the head of the list.

```
; STACK (Linked Implementation)
; Top of stack = head of list
; Simpler: no size limit, no wasted space

(define-record-type stack
  (make top-node)
```

```
  stack?
  (top-node top-node top-node!))  ; Points to first node (or '())

(define (new) (make '()))  ; Empty stack = empty list

(define (empty? s) (null? (top-node s)))

(define (push! s val)
  ; New node becomes the new head, old head is now second
  (top-node! s (cons val (top-node s)))
  s)

(define (pop! s)
  (if (empty? s) (error "stack empty"))
  (let ((val (car (top-node s))))  ; Get value from head
    (top-node! s (cdr (top-node s)))  ; Move head to next node
    val))

(define (top s)
  (if (empty? s) (error "stack empty"))
  (car (top-node s)))  ; Just look at head
```

4.2 Queues (FIFO)

What is a Queue?

A **queue** is like a line at a shop: people join at the back and leave from the front.
FIFO = First In, First Out: The first thing you add is the first thing you get back.
Real-world examples:

- **Print queue:** Documents print in the order they were sent
- **Waiting line:** First person in line gets served first
- **Message queue:** Process messages in the order they arrive

Operations:

- `enqueue!(queue, val)`, Add element at the back (rear)
- `serve!(queue)`, Remove and return element from the front
- `peek(queue)`, Look at front element (without removing)

All operations are $O(1)$ with proper implementation!

4.2.1 Linked Queue Implementation

Need two pointers: one to the front (for serving) and one to the back (for adding).

```
; QUEUE (Linked Implementation)
; head = front of queue (where we serve)
; rear = back of queue (where we add)

(define-record-type queue
  (make head rear)
  queue?
  (head head head!)    ; Front of queue
  (rear rear rear!))   ; Back of queue

(define (new) (make '() '()))  ; Both pointers start empty
(define (empty? q) (null? (head q)))

(define (enqueue! q val)
  (let ((new-node (cons val '())))  ; Create new node
    (if (empty? q)
        ; Empty queue: new node is both head AND rear
        (begin (head! q new-node) (rear! q new-node))
        ; Non-empty: attach to current rear, update rear
        (begin (set-cdr! (rear q) new-node)
               (rear! q new-node))))
  q)

(define (serve! q)
  (if (empty? q) (error "queue empty"))
```

```
  (let ((val (car (head q))))   ; Get front value
    (head! q (cdr (head q)))    ; Move head to next
    (if (null? (head q))        ; Queue now empty?
        (rear! q '()))          ; Clear rear too!
    val))
```

4.2.2 Circular Vector Queue

Use a vector as a "circular buffer" where indices wrap around with modulo.

```
; QUEUE (Circular Vector Implementation)
; Trick: treat vector as circular using modulo
; head-idx and rear-idx chase each other around the circle

(define-record-type queue
  (make storage head-idx rear-idx)
  queue?
  (storage storage)
  (head-idx head-idx head-idx!)   ; Index of front element
  (rear-idx rear-idx rear-idx!))  ; Index of NEXT free slot

(define (new size)
  (make (make-vector size) 0 0))  ; Both indices start at 0

; Empty when head catches up to rear
(define (empty? q) (= (head-idx q) (rear-idx q)))

; Full when rear is ONE behind head (circular)
(define (full? q)
  (= (modulo (+ (rear-idx q) 1) (vector-length (storage q)))
     (head-idx q)))

(define (enqueue! q val)
  (if (full? q) (error "queue full"))
  (vector-set! (storage q) (rear-idx q) val)
  ; Wrap around using modulo!
  (rear-idx! q (modulo (+ (rear-idx q) 1)
                       (vector-length (storage q))))
  q)

(define (serve! q)
  (if (empty? q) (error "queue empty"))
  (let ((val (vector-ref (storage q) (head-idx q))))
    ; Wrap around using modulo!
    (head-idx! q (modulo (+ (head-idx q) 1)
                         (vector-length (storage q))))
    val))
```

4.3 Priority Queues (HPFO)

What is a Priority Queue?

Like a regular queue, but elements have **priorities**. The element with the *highest priority* comes out first, regardless of when it was added.

HPFO = Highest Priority First Out

Real-world examples:

- **Emergency room:** Patients are seen by urgency, not arrival time
- **Task scheduler:** High-priority processes run before low-priority ones
- **To-do list:** Most important tasks come first

Operations:

- `enqueue!(pq, val, priority)`, Add element with priority
- `serve!(pq)`, Remove and return highest-priority element
- `peek(pq)`, Look at highest-priority element

Operation	Sorted List	Unsorted List	Heap
`enqueue!`	$O(n)$	$O(1)$	$O(\log n)$
`serve!`	$O(1)$	$O(n)$	$O(\log n)$
`peek`	$O(1)$	$O(n)$	$O(1)$

Table 3: Priority Queue: The heap gives the best balance of fast insert AND fast serve!

4.4 Heaps

What is a Heap?

A **heap** is a clever way to implement a priority queue. It's a **complete binary tree** where every parent is "better" than its children (smaller for min-heap, larger for max-heap).
Key insight: We store this tree in a regular **array** using index math:

- Parent of node i: $\lfloor (i-1)/2 \rfloor$
- Left child of node i: $2i+1$
- Right child of node i: $2i+2$

The heap property: Every parent $\leq$ its children (for a min-heap).
Why is this fast? The tree has height $\log n$, so insert and delete only need to fix $\log n$ nodes!

Heap Operations

Insert (sift-up):

1. Put new element at the end of the array
2. "Bubble it up": while smaller than parent, swap with parent

Delete-min (sift-down):

1. Save the root (smallest element)
2. Move last element to root position
3. "Push it down": while larger than a child, swap with smallest child

Heap Properties

- Height of heap with n elements: $h = \lfloor \log_2 n \rfloor$
- Number of leaves: $\lceil n/2 \rceil$
- `insert!`: $O(\log n)$, `delete!`: $O(\log n)$, `peek`: $O(1)$

Heap Properties

1. Height of heap with n elements: $h = \lfloor \log_2 n \rfloor$
2. Number of leaves: $\lceil n/2 \rceil$
3. Nodes at height h: at most $\lceil n/2^{h+1} \rceil$

Algorithm 3: Heap Operations

Insert: Place new element at end, `sift-up`;
Delete: Replace root with last element, `sift-down`;
Sift-up: Compare with parent, swap if smaller, repeat;
Sift-down: Compare with children, swap with smallest, repeat;

4.4.1 Heap Implementation

```
; HEAP - Priority queue implemented as array-based tree
; <<? is the comparison function (e.g., < for min-heap)

(define-record-type heap
  (make storage size <<?))
  heap?
```

```
  (storage storage)   ; Vector holding heap elements
  (size size size!)   ; Current number of elements
  (<<? lesser))       ; Comparison function

(define (new capacity <<?)
  (make (make-vector capacity) 0 <<?))

; Index calculations for tree navigation
(define (parent i) (quotient (- i 1) 2))  ; Parent index
(define (left-child i) (+ (* 2 i) 1))     ; Left child index
(define (right-child i) (+ (* 2 i) 2))    ; Right child index

; Helper: swap two elements in a vector
(define (swap! vec i j)
  (let ((tmp (vector-ref vec i)))
    (vector-set! vec i (vector-ref vec j))
    (vector-set! vec j tmp)))

; SIFT-UP: "Bubble" element up to restore heap property
; Used after insert: new element might be too small for its position
(define (sift-up heap idx)
  (define <<? (lesser heap))
  (define vec (storage heap))
  (let loop ((i idx))
    (if (> i 0)                           ; Not at root yet?
        (let ((p (parent i)))
          (when (<<? (vector-ref vec i)   ; Child < parent?
                     (vector-ref vec p))
            (swap! vec i p)               ; Swap them
            (loop p))))))                 ; Continue up

; SIFT-DOWN: "Push" element down to restore heap property
; Used after delete: root might be too large for its position
(define (sift-down heap idx)
  (define <<? (lesser heap))
  (define vec (storage heap))
  (define n (size heap))
  (let loop ((i idx))
    (let* ((l (left-child i))
           (r (right-child i))
           (smallest i))                  ; Start assuming parent is smallest
      ; Check if left child is smaller
      (when (and (< l n)
                 (<<? (vector-ref vec l) (vector-ref vec smallest)))
        (set! smallest l))
      ; Check if right child is even smaller
      (when (and (< r n)
                 (<<? (vector-ref vec r) (vector-ref vec smallest)))
        (set! smallest r))
      ; If a child was smaller, swap and continue
      (when (not (= smallest i))
        (swap! vec i smallest)
        (loop smallest)))))

; INSERT: Add element, then bubble it up
(define (insert! heap val)
  (define n (size heap))
  (vector-set! (storage heap) n val)  ; Put at end
  (size! heap (+ n 1))                ; Increase size
  (sift-up heap n)                    ; Fix heap property
  heap)

; DELETE: Remove root, replace with last, push down
(define (delete! heap)
  (if (= (size heap) 0) (error "heap empty"))
  (let ((val (vector-ref (storage heap) 0)))  ; Save root
    (size! heap (- (size heap) 1))
    (vector-set! (storage heap) 0             ; Move last to root
                 (vector-ref (storage heap) (size heap)))
    (sift-down heap 0)                        ; Fix heap property
    val))

(define (peek heap)
```

```
  (if (= (size heap) 0) (error "heap empty"))
  (vector-ref (storage heap) 0))  ; Root is always min/max
```

4.4.2 Heapify, Building a Heap in $O(n)$

Clever Insight

You might think building a heap from n elements takes $O(n \log n)$ (insert each one). But there's a trick: start from the bottom of the tree and sift-down each non-leaf node. This takes only $O(n)$!

```
; Build heap from existing vector in O(n) time!
; Trick: only non-leaf nodes need sift-down
(define (from-scheme-vector vec <<?)
  (define n (vector-length vec))
  (define heap (make vec n <<?))
  ; Start from last non-leaf (parent of last element)
  (let loop ((i (quotient n 2)))
    (when (>= i 0)
      (sift-down heap i)    ; Fix this subtree
      (loop (- i 1))))      ; Move to previous node
  heap)
```

Operation	Complexity
`insert!`	$O(\log n)$
`delete!`	$O(\log n)$
`peek`	$O(1)$
`from-scheme-vector` (heapify)	$O(n)$

Table 4: Heap Complexity

5 Sorting (Hoofdstuk 5)

5.1 Why Study Sorting?

The Sorting Problem

Goal: Rearrange elements so they're in order (ascending or descending).
Why is this important?

- Sorted data enables **binary search** ($O(\log n)$ instead of $O(n)$)
- Many algorithms *require* sorted input
- Finding duplicates, medians, etc. becomes much easier

Sorting Properties

- **Stable**: Equal elements keep their original relative order. Important when sorting by multiple criteria!
- **In-place**: Uses only $O(1)$ extra memory (sorts "within" the original array)
- **Comparative**: Only uses comparisons (has theoretical lower bound $\Omega(n \log n)$)

5.2 Simple Sorting Algorithms, Quadratic

These are easy to understand but slow for large inputs. Useful for small arrays or nearly-sorted data.

5.2.1 Bubble Sort, The Beginner's Algorithm

Idea: Repeatedly compare adjacent elements and swap if out of order. Large elements "bubble up" to the end.

Algorithm 4: Bubble Sort

```
for i ← n − 2 down to 0 do
    for j ← 0 to i do
        if a[j + 1] < a[j] then
            swap a[j] and a[j + 1];
        end
    end
end
```

Avoid Bubble Sort!

Bubble Sort is $O(n^2)$ in the worst AND average case. It's mainly taught for educational purposes. Use something better in practice!

5.2.2 Insertion Sort, Good for Small/Nearly-Sorted Data

Idea: Like sorting playing cards in your hand. Take each card and insert it into its correct position among the already-sorted cards.

- **Best case:** $O(n)$, already sorted! Just check each element once.
- **Worst case:** $O(n^2)$, reverse sorted, every element must move all the way.
- Stable, In-place, Good for nearly sorted data or small n

5.2.3 Selection Sort, Minimal Swaps

Idea: Find the minimum element, swap it to position 0. Find the next minimum, swap to position 1. Repeat.

- Always exactly $\Theta(n^2)$ comparisons (no best/worst case difference)
- Only $O(n)$ swaps (good if swapping is expensive)
- **Not stable**, In-place

5.3 Advanced Sorting Algorithms, O(n log n)

These algorithms achieve the *optimal* complexity for comparison-based sorting.

5.3.1 QuickSort, The Fast One (Usually)

Idea: Pick a "pivot" element. Partition the array so all smaller elements go left, all larger go right. Recursively sort each side.

Why is it fast? Each partition splits the problem in half (ideally), giving $\log n$ levels of recursion, each doing $O(n)$ work.

Algorithm 5: QuickSort

```
Input: Array a, indices l, r
if l < r then
    Choose pivot p (e.g., first element);
    Partition: smaller elements go left, larger go right;
    m ← final position of pivot;
    QuickSort(a, l, m − 1);
    QuickSort(a, m + 1, r);
end
```

QuickSort's Achilles' Heel

If the pivot is always the smallest or largest element (e.g., first element on sorted input), we get $O(n^2)$!
Solutions: Random pivot, median-of-three, or switch to insertion sort for small subarrays.

- Best/Average case: $O(n \log n)$, Worst case: $O(n^2)$
- **Not stable**, In-place (but uses $O(\log n)$ stack space for recursion)

5.3.2 MergeSort, The Reliable One

Idea: Divide the array in half. Sort each half recursively. Merge the two sorted halves back together.

Why is it fast? We always split exactly in half ($\log n$ levels), and merging takes $O(n)$ work per level.

Algorithm 6: MergeSort

if $p < r$ **then**
  $q \leftarrow \lfloor (p + r)/2 \rfloor$;
  MergeSort(a, p, q); // Sort left half
  MergeSort$(a, q + 1, r)$; // Sort right half
  Merge(a, p, q, r); // Combine sorted halves
end

- **Always** $O(n \log n)$, no bad cases!
- **Stable** (if we're careful during merge)
- **Not in-place**: requires $O(n)$ extra space for merging
- Excellent for linked lists and external sorting (sorting huge files)

5.3.3 HeapSort, The Elegant One

Idea: Build a max-heap from the array. Repeatedly extract the maximum (root) and put it at the end.

Algorithm 7: HeapSort

Build max-heap from array: $O(n)$;
for $i \leftarrow n - 1$ *down to* 1 **do**
  Swap $a[0]$ (max) with $a[i]$;
  Reduce heap size by 1;
  Sift-down on position 0: $O(\log n)$;
end

- Always $O(n \log n)$, guaranteed!
- **Not stable**, but **truly in-place** (only $O(1)$ extra space)

5.4 Sorting Implementations

5.4.1 Bubble Sort

```
; BUBBLE SORT: Compare adjacent elements, swap if out of order
; "Bubbles" large elements to the end of the array

(define (bubble-sort vector <<?)
  ; Helper: swap two elements at given indices
  (define (swap idx1 idx2)
    (let ((temp (vector-ref vector idx1)))
      (vector-set! vector idx1 (vector-ref vector idx2))
      (vector-set! vector idx2 temp)
      #t))  ; Return true to indicate a swap happened

  ; Outer loop: after each pass, one more element is sorted
  (let outer-loop ((unsorted-idx (- (vector-length vector) 2)))
    (if (>= unsorted-idx 0)
        ; Inner loop: bubble through unsorted portion
        (if (let inner-loop ((inner-idx 0) (changed? #f))
              (if (> inner-idx unsorted-idx)
```

```
                    changed?  ; Return whether any swaps happened
                    (inner-loop (+ inner-idx 1)
                      ; Compare adjacent elements
                      (if (<<? (vector-ref vector (+ inner-idx 1))
                               (vector-ref vector inner-idx))
                          (swap inner-idx (+ inner-idx 1))  ; Swap!
                          changed?))))
              (outer-loop (- unsorted-idx 1))))))  ; Continue if swaps happened
```

5.4.2 Insertion Sort

```
; INSERTION SORT: Like sorting cards in your hand
; Pick each element and insert it in the right place

(define (insertion-sort vector <<?)
  ; Start from second-to-last, work backwards
  (let outer-loop ((outer-idx (- (vector-length vector) 2)))
    (let ((current (vector-ref vector outer-idx)))  ; Card to insert
      ; Find where current belongs by shifting larger elements right
      (vector-set! vector
        (let inner-loop ((inner-idx (+ 1 outer-idx)))
          (cond
            ; Found the right spot? (or hit end)
            ((or (>= inner-idx (vector-length vector))
                 (not (<<? (vector-ref vector inner-idx) current)))
             (- inner-idx 1))  ; Return insertion position
            (else
             ; Shift this element left to make room
             (vector-set! vector (- inner-idx 1)
                          (vector-ref vector inner-idx))
             (inner-loop (+ inner-idx 1)))))
        current)  ; Place current in the gap
      ; Continue with remaining unsorted elements
      (if (> outer-idx 0)
          (outer-loop (- outer-idx 1))))))
```

5.4.3 Selection Sort

```
; SELECTION SORT: Find minimum, swap to front, repeat
; Always O(n^2) comparisons, but only O(n) swaps

(define (selection-sort vector <<?)
  ; Helper: swap two elements
  (define (swap i j)
    (let ((keep (vector-ref vector i)))
      (vector-set! vector i (vector-ref vector j))
      (vector-set! vector j keep)))

  ; For each position, find the smallest remaining element
  (let outer-loop ((outer-idx 0))
    (swap outer-idx
          ; Find index of smallest element from outer-idx onwards
          (let inner-loop ((inner-idx (+ outer-idx 1))
                           (smallest-idx outer-idx))
            (cond
              ; Done scanning? Return smallest index
              ((>= inner-idx (vector-length vector))
               smallest-idx)
              ; Found smaller? Update smallest-idx
              ((<<? (vector-ref vector inner-idx)
                    (vector-ref vector smallest-idx))
               (inner-loop (+ inner-idx 1) inner-idx))
              ; Keep current smallest
              (else
               (inner-loop (+ inner-idx 1) smallest-idx)))))
    ; Continue to next position
    (if (< outer-idx (- (vector-length vector) 1))
        (outer-loop (+ outer-idx 1)))))
```

5.4.4 QuickSort

```
; QUICKSORT: Pick pivot, partition around it, recurse
; Fast on average O(n log n), but O(n^2) on sorted input!

(define (quicksort vector <<?)
  ; Helper: swap two elements
  (define (swap i j)
    (let ((keep (vector-ref vector i)))
      (vector-set! vector i (vector-ref vector j))
      (vector-set! vector j keep)))

  ; Partition: rearrange so elements < pivot are left,
  ;            elements > pivot are right
  (define (partition pivot i j)
    ; Find element >= pivot from left
    (define (shift-right i)
      (if (<<? (vector-ref vector i) pivot)
          (shift-right (+ i 1)) i))
    ; Find element <= pivot from right
    (define (shift-left j)
      (if (<<? pivot (vector-ref vector j))
          (shift-left (- j 1)) j))
    ; Swap if pointers haven't crossed
    (let ((si (shift-right i)) (sj (shift-left j)))
      (cond ((< si sj) (swap si sj) (partition pivot si (- sj 1)))
            (else sj))))  ; Return partition boundary

  ; Recursive sort
  (define (quicksort-rec l r)
    (if (< l r)
        (begin
          ; Ensure first element <= last (optimization)
          (if (<<? (vector-ref vector r) (vector-ref vector l))
              (swap l r))
          ; Partition using first element as pivot
          (let ((m (partition (vector-ref vector l) (+ l 1) r)))
            (swap l m)                  ; Move pivot to final position
            (quicksort-rec l (- m 1))    ; Sort left of pivot
            (quicksort-rec (+ m 1) r))))) ; Sort right of pivot

  (quicksort-rec 0 (- (vector-length vector) 1)))
```

5.4.5 MergeSort

```
; MERGESORT: Divide in half, sort each, merge back
; Always O(n log n), but needs O(n) extra space

(define (merge-sort vector <<?)
  ; Merge two sorted halves: [p..q] and [q+1..r]
  (define (merge p q r)
    (let ((work (make-vector (+ (- r p) 1))))  ; Temp storage
      ; Copy work array back to original
      (define (copy-back a b)
        (vector-set! vector b (vector-ref work a))
        (if (< a (- (vector-length work) 1))
            (copy-back (+ a 1) (+ b 1))))
      ; Copy remaining elements from one half
      (define (flush k i until)
        (vector-set! work k (vector-ref vector i))
        (if (< i until)
            (flush (+ k 1) (+ i 1) until)
            (copy-back 0 p)))
      ; Main merge: pick smaller element from each half
      (let merge-iter ((k 0) (i p) (j (+ q 1)))
        (cond
          ; Both halves have elements? Compare and take smaller
          ((and (<= i q) (<= j r))
           (let ((v1 (vector-ref vector i))
                 (v2 (vector-ref vector j)))
```

```
            (if (<<? v1 v2)
                (begin (vector-set! work k v1)
                       (merge-iter (+ k 1) (+ i 1) j))
                (begin (vector-set! work k v2)
                       (merge-iter (+ k 1) i (+ j 1))))))
          ; Left half has remaining? Flush it
          ((<= i q) (flush k i q))
          ; Right half has remaining? Flush it
          (else (flush k j r))))))

  ; Recursive divide-and-conquer
  (define (merge-sort-rec p r)
    (if (< p r)
        (let ((q (quotient (+ r p) 2)))  ; Middle point
          (merge-sort-rec p q)           ; Sort left half
          (merge-sort-rec (+ q 1) r)     ; Sort right half
          (merge vector p q r))))        ; Merge sorted halves

  (merge-sort-rec 0 (- (vector-length vector) 1))
  vector)
```

5.4.6 HeapSort

```
; HEAPSORT: Build max-heap, repeatedly extract max
; Always O(n log n), truly in-place (only O(1) extra space)

(define (heapsort vector <<?)
  ; Use >> for max-heap (we want largest at root)
  (define >>? (lambda (x y) (not (<<? x y))))
  ; Build max-heap from the vector in O(n)
  (define heap (from-scheme-vector vector >>?))
  ; Extract elements from largest to smallest
  (let extract ((idx (- (vector-length vector) 1)))
    ; Put max at end, shrink heap, repeat
    (vector-set! vector idx (delete! heap))
    (if (> idx 0)
        (extract (- idx 1)))))
```

Lower Bound for Comparison-Based Sorting

Any comparison-based sorting algorithm requires $\Omega(n \log n)$ comparisons in the worst case.
Proof: Decision tree has $n!$ leaves (permutations). Height $h \geq \log(n!) \in \Omega(n \log n)$.

5.5 Linear Sorting Algorithms, Linear Time

These are **non-comparative** and require additional knowledge about keys.

5.5.1 Counting Sort

1. Count occurrences of each key value
2. Compute cumulative sums
3. Place elements at correct positions

Complexity: $O(n + k)$ where k = range of key values. Stable.

5.5.2 Radix Sort

Sort digit by digit (least significant first), using a stable sort (e.g., counting sort) for each digit.
Complexity: $O(d \cdot n)$ where d = number of digits. Stable.

5.5.3 Bucket Sort

Distribute elements into buckets, sort each bucket, concatenate.
Complexity: $O(n)$ average case with uniform distribution.

Algorithm	Best	Average	Worst	Stable	In-place
Bubble Sort	$\Omega(n)$	$O(n^2)$	$O(n^2)$	Yes	Yes
Insertion Sort	$\Omega(n)$	$O(n^2)$	$O(n^2)$	Yes	Yes
Selection Sort	$\Theta(n^2)$	$\Theta(n^2)$	$\Theta(n^2)$	No	Yes
QuickSort	$\Omega(n \log n)$	$O(n \log n)$	$O(n^2)$	No	No
MergeSort	$\Theta(n \log n)$	$\Theta(n \log n)$	$\Theta(n \log n)$	Yes	No
HeapSort	$\Theta(n \log n)$	$\Theta(n \log n)$	$\Theta(n \log n)$	No	Yes
Counting Sort		$O(n + k)$		Yes	No
Radix Sort		$O(d \cdot n)$		Yes	No

Table 5: Comprehensive Sorting Algorithm Comparison

6 Trees (Hoofdstuk 6)

6.1 Why Trees?

Trees vs. Linear Structures

Linear structures (lists, arrays) give us $O(n)$ search. Can we do better?
Yes! With trees, we can achieve $O(\log n)$ search, much faster for large datasets.
Intuition: Instead of checking each element one by one, we eliminate half the possibilities at each step (like binary search, but dynamic).

6.2 Tree Terminology

Binary Tree

A tree where each node has at most 2 children (left and right).
Key terms:

- **Root:** The topmost node (no parent)
- **Leaf:** A node with no children
- **Height:** Longest path from root to any leaf
- **Complete binary tree:** All levels filled except possibly the last, which is filled left-to-right

Height of complete binary tree with n nodes: $h = \lfloor \log_2 n \rfloor$

6.3 Tree Traversals

How do we visit every node in a tree? There are different orders:

Depth-First Traversals

- **Preorder** (Root, Left, Right): Visit root first, then children
- **Inorder** (Left, Root, Right): For BSTs, gives sorted order!
- **Postorder** (Left, Right, Root): Visit children before root (useful for deletion)

Breadth-First Traversal

Visit nodes *level by level*, using a queue. Good for finding shortest paths.

6.4 Binary Search Trees (BST)

What is a BST?

A binary tree with a special **ordering property**: for every node n:

- All elements in the **left** subtree are $< n$
- All elements in the **right** subtree are $> n$

Why is this useful? To find an element, compare with the root:

- If equal: found!
- If smaller: look in left subtree
- If larger: look in right subtree

Each comparison eliminates half the remaining tree (ideally).

BST Operations

Find: Navigate left/right based on comparison, $O(h)$
Insert: Find the right spot, add new node, $O(h)$
Delete: Three cases:

1. **Leaf:** Simply remove it
2. **One child:** Replace with that child
3. **Two children:** Replace with inorder successor (leftmost in right subtree)

The Problem with BSTs

If we insert elements in sorted order, the tree becomes a "stick" (linked list) with height n instead of $\log n$. Then all operations become $O(n)$!
Solution: Keep the tree **balanced** using AVL trees.

6.5 AVL Trees, Self-Balancing BSTs

What is an AVL Tree?

An AVL tree is a BST that automatically keeps itself balanced. Named after inventors Adelson-Velskii and Landis (1962).
The AVL property: For *every* node, the height difference between left and right subtrees is **at most 1**.
Result: Height is always $O(\log n)$, so all operations are $O(\log n)$, guaranteed!

Balance Factor

For each node, we track its "balance":

- `balanced`: left and right have same height
- `Lhigh`: left subtree is 1 taller
- `Rhigh`: right subtree is 1 taller

If the imbalance becomes 2, we need to fix it with a **rotation**.

6.5.1 Rotations, Fixing Imbalance

When Do We Need Rotations?

After insert or delete, if a node becomes "unbalanced" (height difference = 2), we fix it with 1-2 rotations.

Four cases:

1. **Left-Left:** Inserted in left child's left subtree → single right rotation
2. **Right-Right:** Inserted in right child's right subtree → single left rotation
3. **Left-Right:** Inserted in left child's right subtree → double rotation (left then right)
4. **Right-Left:** Inserted in right child's left subtree → double rotation (right then left)

All rotations are $O(1)$, just reassigning a few pointers!

	BST		AVL	
Operation	Worst	Average	Worst	Average
`find`	$O(n)$	$O(\log n)$	$O(\log n)$	$O(\log n)$
`insert!`	$O(n)$	$O(\log n)$	$O(\log n)$	$O(\log n)$
`delete!`	$O(n)$	$O(\log n)$	$O(\log n)$	$O(\log n)$

Table 6: BST vs AVL: AVL eliminates the worst case!

6.6 Tree and BST Implementations

6.6.1 Binary Tree Node

```
; BINARY TREE NODE: Each node has a value and two children
; null-tree represents empty tree (using '())

(define-record-type tree
  (new v l r)
  tree?
  (v value value!)    ; The data stored in this node
  (l left left!)      ; Left child (or null-tree)
  (r right right!))   ; Right child (or null-tree)

(define null-tree '())                    ; Empty tree sentinel
(define (null-tree? node) (eq? node null-tree))
```

6.6.2 Tree Traversals

```
; PREORDER: Visit Root BEFORE children (Root-Left-Right)
; Use case: copy a tree, prefix expression
(define (pre-order tree proc)
  (when (not (null-tree? tree))
    (proc (value tree))           ; Visit root first
    (pre-order (left tree) proc)  ; Then left subtree
    (pre-order (right tree) proc))) ; Then right subtree

; INORDER: Visit Root BETWEEN children (Left-Root-Right)
; Use case: BST gives sorted order!
(define (in-order tree proc)
  (when (not (null-tree? tree))
    (in-order (left tree) proc)   ; Visit left first
    (proc (value tree))           ; Then root
    (in-order (right tree) proc))) ; Then right

; POSTORDER: Visit Root AFTER children (Left-Right-Root)
; Use case: delete tree, postfix expression
(define (post-order tree proc)
  (when (not (null-tree? tree))
    (post-order (left tree) proc)  ; Visit left first
    (post-order (right tree) proc) ; Then right
```

```
    (proc (value tree))))          ; Root last

; BREADTH-FIRST: Visit level by level using a queue
; Use case: find shortest path, level-order print
(define (breadth-first tree proc)
  (define q (queue:new))           ; Create queue for pending nodes
  (queue:enqueue! q tree)
  (let loop ()
    (when (not (queue:empty? q))
      (let ((node (queue:serve! q)))  ; Get next node
        (proc (value node))           ; Visit it
        ; Enqueue children for later
        (when (not (null-tree? (left node)))
          (queue:enqueue! q (left node)))
        (when (not (null-tree? (right node)))
          (queue:enqueue! q (right node)))
        (loop)))))
```

6.6.3 BST Implementation

```
; BINARY SEARCH TREE: Ordered binary tree
; All elements in left < root < all elements in right

(define-record-type bst
  (make r e l)
  bst?
  (r root root!)   ; The root node of the tree
  (e equality)     ; ==? predicate for comparing keys
  (l lesser))      ; <<? predicate for ordering

(define (new ==? <<?)
  (make null-tree ==? <<?))  ; Start with empty tree

; FIND: Search for a key, returning its value or #f
(define (find bst key)
  (define <<? (lesser bst))
  (define ==? (equality bst))
  (let find-key ((node (root bst)))
    (if (null-tree? node)
        #f                           ; Not found
        (let ((v (value node)))
          (cond
            ((==? v key) v)          ; Found it!
            ((<<? v key)             ; Key is larger?
             (find-key (right node)))  ; Search right
            (else                    ; Key is smaller?
             (find-key (left node)))))))) ; Search left)

; INSERT: Add a new value to the BST
(define (insert! bst val)
  (define <<? (lesser bst))
  ; Walk down tree, tracking parent and which child slot to use
  (let insert-iter ((parent null-tree)
                    (child! (lambda (i c) (root! bst c)))  ; Setter for child
                    (child (root bst)))
    (cond
      ; Empty spot? Create new node here
      ((null-tree? child)
       (child! parent (new val null-tree null-tree)))
      ; val > current? Go right
      ((<<? (value child) val)
       (insert-iter child right! (right child)))
      ; val < current? Go left
      ((<<? val (value child))
       (insert-iter child left! (left child)))
      ; val == current? Update value (no duplicates)
      (else (value! child val)))))
```

6.6.4 BST Delete

```
; DELETE: Remove a value from the BST
; Three cases: leaf, one child, two children
(define (delete! bst val)
  (define <<? (lesser bst))
  (define ==? (equality bst))

  ; Helper: find leftmost node (inorder successor)
  ; Used when deleting node with two children
  (define (find-leftmost deleted parent child! child)
    (if (null-tree? (left child))
        (begin
          (value! deleted (value child))  ; Copy value to deleted node
          (child! parent (right child)))  ; Remove this node
        (find-leftmost deleted child left! (left child))))

  ; Delete the node after finding it
  (define (delete-node parent child! child)
    (cond
      ; Case 1: No left child? Replace with right child
      ((null-tree? (left child))
       (child! parent (right child)))
      ; Case 2: No right child? Replace with left child
      ((null-tree? (right child))
       (child! parent (left child)))
      ; Case 3: Two children? Find inorder successor
      (else
       (find-leftmost child child right! (right child)))))

  ; Find the node to delete
  (let find-node ((parent null-tree)
                  (child! (lambda (i c) (root! bst c)))
                  (child (root bst)))
    (cond
      ((null-tree? child) #f)                ; Not found
      ((==? (value child) val)               ; Found it!
       (delete-node parent child! child))
      ((<<? (value child) val)               ; val > current
       (find-node child right! (right child)))
      (else                                  ; val < current
       (find-node child left! (left child))))))
```

6.6.5 AVL Rotations

```
; AVL ROTATIONS: Fix imbalanced trees with O(1) pointer changes
; "black" = unbalanced node, "white" = its higher child

; SINGLE LEFT ROTATION: For Right-Right imbalance
;       black             white
;         \               /
;        white   =>    black
;         /               \
;       tree             tree
(define (single-rotate-left! black)
  (define white (right black))    ; white becomes new root
  (define tree (left white))      ; tree moves from white to black
  (right! black tree)             ; tree becomes black's right child
  (left! white black)             ; black becomes white's left child
  white)                          ; return new root

; SINGLE RIGHT ROTATION: For Left-Left imbalance
(define (single-rotate-right! black)
  (define white (left black))
  (define tree (right white))
  (left! black tree)
  (right! white black)
  white)

; DOUBLE ROTATION (Left-then-Right): For Left-Right imbalance
```

```
; First rotate child left, then rotate this node right
(define (double-rotate-left-then-right! black)
  (left! black (single-rotate-left! (left black)))
  (single-rotate-right! black))

; DOUBLE ROTATION (Right-then-Left): For Right-Left imbalance
(define (double-rotate-right-then-left! black)
  (right! black (single-rotate-right! (right black)))
  (single-rotate-left! black))
```

	BST		AVL	
Operation	Worst	Average	Worst	Average
`find`	$O(n)$	$O(\log n)$	$O(\log n)$	$O(\log n)$
`insert!`	$O(n)$	$O(\log n)$	$O(\log n)$	$O(\log n)$
`delete!`	$O(n)$	$O(\log n)$	$O(\log n)$	$O(\log n)$

Table 7: BST vs AVL Tree Complexity

7 Hash Tables (Hoofdstuk 7)

7.1 Basic Concepts

Hash Function

A function $h : K \to \{0, 1, \ldots, M-1\}$ that maps keys to array indices in $O(1)$.
The **home address** of key k is $h(k) \mod M$.

Collision

When $h(k_1) = h(k_2)$ for $k_1 \neq k_2$.
A **perfect hash function** produces no collisions (rare in practice).

Load Factor

$$\alpha = \frac{n}{M}$$

where n = number of elements, M = table size. Keep $\alpha < 0.75$ for good performance.

Funneling

When keys hash to only a portion of the table. Caused by common factors between $h(k)$ and M.
Solution: Choose M prime, or M as power of 2 with $h(k)$ always odd.

7.2 Collision Resolution Strategies

7.2.1 External Chaining

Each table slot contains a linked list (bucket) of all elements that hash to that slot.

- α can exceed 1
- Average case: $O(\alpha)$ for all operations
- Worst case: $O(n)$ (all keys hash to same slot)

7.2.2 Open Addressing

All elements stored directly in the table. On collision, **probe** for next free slot.

Linear Probing

$$h'(k,i) = (h(k) + c \cdot i) \mod M$$

where $i = 1, 2, 3, \ldots$
Requirement: c and M must be coprime.
Problem: Primary clustering (long runs of occupied slots).

Quadratic Probing

$$h'(k,i) = (h(k) + c_1 \cdot i + c_2 \cdot i^2) \mod M$$

Eliminates primary clustering, but still has secondary clustering.

Double Hashing

$$h'(k,i) = (h_1(k) + h_2(k) \cdot i) \mod M$$

Uses two hash functions. Eliminates both primary and secondary clustering.
Requirement: $h_2(k)$ and M must be coprime.

Tombstone

When deleting with open addressing, mark slot as `'deleted` (tombstone) rather than `'empty`, to preserve probe sequences.

Clustering

Primary clustering: If probe sequences of different keys intersect, they become identical thereafter.
Secondary clustering: If two keys have same initial hash, they follow identical probe sequences.

7.3 Expected Number of Probes

Method	Successful	Unsuccessful
Linear Probing	$\frac{1}{2}\left(1 + \frac{1}{(1-\alpha)^2}\right)$	$\frac{1}{2}\left(1 + \frac{1}{1-\alpha}\right)$
Double/Quadratic	$\frac{1}{\alpha}\ln\left(\frac{1}{1-\alpha}\right)$	$\frac{1}{1-\alpha}$
External Chaining	$1 + \frac{\alpha}{2}$	$1 + \alpha$

Table 8: Expected Probes by Load Factor α

7.4 Hash Functions

Good Hash Function Properties

- **Fast**: Should not dominate algorithm complexity
- **Strong**: Each location has equal probability
- **Simple**: Easy to analyze for clustering/funneling

Common Methods:
- **Folding**: Sum of digits (weak)
- **Digit Selection**: Choose specific digits
- **Division**: $h(k) = k \mod M$ (use prime M)
- **Multiplication**: $h(k) = M \cdot (kC \mod 1)$ with $C \approx 0.618034$

7.5 Hash Table Implementations

7.5.1 External Chaining

```
; EXTERNAL CHAINING: Each slot contains a linked list (bucket)
; Multiple keys can hash to the same slot without problem

(define-record-type external-chaining
  (make s h e)
  dictionary?
  (s storage)        ; Vector of buckets (linked lists)
  (h hash-function)  ; Maps key -> index
  (e equality))      ; ==? for comparing keys

; Key-value pair stored as simple cons cell
(define make-assoc cons)
(define assoc-key car)
(define assoc-value cdr)

; Create new hash table with M buckets
(define (new ==? M h)
  (make (make-vector M '())          ; M empty buckets
        (lambda (k) (modulo (h k) M))  ; Wrap hash to table size
        ==?))

; FIND: Hash to bucket, then linear search in bucket
(define (find table key)
  (define vec (storage table))
  (define h (hash-function table))
  (define ==? (equality table))
  (define home (h key))              ; Get bucket index
  (let loop ((bucket (vector-ref vec home)))
    (cond
      ((null? bucket) #f)            ; Not in bucket
      ((==? (assoc-key (car bucket)) key)
       (assoc-value (car bucket)))   ; Found it!
      (else (loop (cdr bucket))))))  ; Check next in bucket

; INSERT: Add to front of bucket (O(1))
(define (insert! table key val)
  (define vec (storage table))
  (define h (hash-function table))
  (define home (h key))
  ; Prepend new association to existing bucket
  (vector-set! vec home
               (cons (make-assoc key val)
                     (vector-ref vec home)))
  table)
```

7.5.2 Linear Probing with Tombstones

```
; LINEAR PROBING: All elements stored directly in table
; On collision, check next slot (and next, and next...)

(define c 1)  ; Probe step size (must be coprime with M)
(define (rehash address M) (modulo (+ address c) M))

(define-record-type linear-probing
  (make s h e)
  dictionary?
  (s storage)        ; Vector of slots (not buckets!)
  (h hash-function)
  (e equality))

; Create table with M slots, all initially 'empty
(define (new ==? M h)
  (make (make-vector M 'empty)
        (lambda (x) (modulo (h x) M)) ==?))

; FIND: Probe until we find key or an empty slot
```

```scheme
(define (find table key)
  (define vec (storage table))
  (define M (vector-length vec))
  (define h (hash-function table))
  (define ==? (equality table))
  (let loop ((addr (h key)))            ; Start at home address
    (let ((slot (vector-ref vec addr)))
      (cond
        ((eq? slot 'empty) #f)          ; Empty = not found
        ((eq? slot 'deleted)            ; Tombstone? Keep looking
         (loop (rehash addr M)))
        ((==? (assoc-key slot) key)     ; Found it!
         (assoc-value slot))
        (else                           ; Wrong key, keep probing
         (loop (rehash addr M)))))))

; INSERT: Find empty or deleted slot, store there
(define (insert! table key val)
  (define vec (storage table))
  (define M (vector-length vec))
  (define h (hash-function table))
  (let loop ((addr (h key)))
    (let ((slot (vector-ref vec addr)))
      (cond
        ; Found empty slot or tombstone? Use it
        ((or (eq? slot 'empty) (eq? slot 'deleted))
         (vector-set! vec addr (make-assoc key val)))
        ; Slot occupied? Keep probing
        (else (loop (rehash addr M))))))
  table)

; DELETE: Mark slot as 'deleted (tombstone)
; IMPORTANT: Can't actually clear the slot, or we'd break
; probe sequences for other keys!
(define (delete! table key)
  (define vec (storage table))
  (define M (vector-length vec))
  (define h (hash-function table))
  (define ==? (equality table))
  (let loop ((addr (h key)))
    (let ((slot (vector-ref vec addr)))
      (cond
        ((eq? slot 'empty) #f)          ; Not found
        ((eq? slot 'deleted)            ; Tombstone, keep looking
         (loop (rehash addr M)))
        ((==? (assoc-key slot) key)     ; Found it!
         (vector-set! vec addr 'deleted) table)
        (else                           ; Wrong key, keep probing
         (loop (rehash addr M)))))))
```

8 Dictionary Implementation Comparison

	insert!		**delete!**		**find**	
Implementation	Worst	Avg	Worst	Avg	Worst	Avg
Plain List	$O(1)$	$O(1)$	$O(n)$	$O(n)$	$O(n)$	$O(n)$
Sorted List (linked)	$O(n)$	$O(n)$	$O(n)$	$O(n)$	$O(n)$	$O(n)$
Sorted List (vector)	$O(n)$	$O(n)$	$O(n)$	$O(n)$	$O(\log n)$	$O(\log n)$
BST	$O(n)$	$O(\log n)$	$O(n)$	$O(\log n)$	$O(n)$	$O(\log n)$
AVL Tree	$O(\log n)$	$O(\log n)$	$O(\log n)$	$O(\log n)$	$O(\log n)$	$O(\log n)$
Hash Table	$O(n)$	$O(1)$	$O(n)$	$O(1)$	$O(n)$	$O(1)$

Table 9: Dictionary Implementation Comparison

Ordered vs Unordered Dictionaries

Tree-based implementations maintain ordering and support range queries. Hash tables do not maintain ordering but offer $O(1)$ average case.