# Week 6

## Dictionaries, 2-3 Trees & Dynamic Programming

Thorben Klabunde

th-kl.ch

October 27, 2025

# Agenda

1. Mini-Quiz
2. Assignment
3. Recap
4. Additional Practice
5. Peer Grading

# Mini-Quiz

# Assignment

# Feedback Assignment 4

**Common points** from last week's assignment:

- **Ex. 4.3:** Remember that you are writing a correctness proof. Even though it appears "less formal", you should apply the same rigor, i.e., be precise in your wording, stick to your invariant, make your reasoning explicit and **explain why the code works**.
- **Ex. 4.4:** Good work on this ex. but be **careful with edge cases** (index out of bounds). Also, although not incorrect, **keeping your code simple** generally helps avoid bugs. You don't need to optimize it to the last iteration if the O-notation remains the same.
- **Ex. 4.5:** Well done! **Only point**, get acquainted with the **rounding** operators. **In the exam**, rounding errors give point **deductions**.

Remember to check the detailed feedback on Moodle! Reach out if you have any questions regarding the corrections.

# Assignment 5

- We will have a closer look at Ex. 5.2 and Ex. 5.3b).
- For Ex. 5.4, please refer to the Master Solution and the slides from Week 4.

# Ex. 5.2) - Walkthrough

**Exercise 5.2** *Guessing an interval.*

Alice and Bob play the following game:

- Alice selects two integers $1 \leq a < b \leq 200$, which she keeps secret.
- Then, Alice and Bob repeat the following:
  - Bob chooses two integers $0 \leq a' < b' \leq 201$.
  - If $a = a'$ and $b = b'$, Bob wins.
  - If $a' < a$ and $b < b'$, Alice tells Bob 'my numbers are strictly between your numbers!'.
  - Otherwise, Alice does not give any clue to Bob.

Bob claims that he has a strategy to win this game in 12 attempts at most. Prove that such a strategy cannot exist.

***Hint:*** *Represent Bob's strategy as a decision tree. Each edge of the decision tree corresponds to one of Alice's answers, while each leaf corresponds to a win for Bob.*

***Hint:*** *After defining the decision tree, you can show that there is at most one leaf for every non-leaf node and the number of non-leaf nodes is at most $2^n - 1$ for a tree of depth $n$ for $n \in \mathbb{N}_0 = \mathbb{N} \cup \{0\}$.*

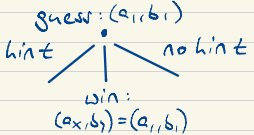
# Ex. 5.2) - Walkthrough

Main idea: Model as a decision tree & argue over the height.

(1) Notice that the game can be structured as a decision-tree:

Alice chooses pair $(a_x, b_y)$, $1 \le s_x < b_y \le 200$

Bob guesses:




(2)

(3)

(4)

# Ex. 5.3b) - Walkthrough

**Algorithm 2**

```
function h(n)
    i ← 1
    while i < n do
        f()
        i ← i + 1
    k(n)
    k(n)
function k(n)
    i ← 2
    while i < n do
        f()
        i ← i^2
    h(n/2)
```

# Ex. 5.3b) - Walkthrough

**Algorithm 2**

**function** $h(n)$
  $i \leftarrow 1$
  **while** $i < n$ **do**
    $f()$
    $i \leftarrow i + 1$
  $k(n)$
  $k(n)$

**function** $k(n)$
  $i \leftarrow 2$
  **while** $i < n$ **do**
    $f()$
    $i \leftarrow i^2$
  $h(n/2)$

```
def h(n):
    i ← 1
    while i<n:
        f()
        i ← i+1
    for j in 1..2:
        i ← 2
        while i<n:
            f()
            i ← i2
        h(n/2)
```

Rewritten:

```
h ←
while k <        :
    f()
    h ← k+1
```

We have to deal with two things here that make reading off the runtime a little hard.

(1) the number of calls to $f$ is split up btw. $h(n)$ & $k(n)$, which call each other.

(2) the loop-condition in $k(n)$ does not let us easily derive the number of iterations.

For (1) notice:

For (2), notice:

# Recap

# ADT Wörterbuch (Dictionary)

**What is it?**

A collection managing unique keys, like a real dictionary maps words to definitions. Supports: search($x$), insert($x$), delete($x$).

**Why not simple structures?** Arrays and Lists have drawbacks:

- Unsorted Array & Linked List: Slow search $O(n)$.
- Sorted Array: Slow updates (insert/delete) $O(n)$.

**Goal:** $\mathbf{O(\log n)}$ for all operations. How? Trees!

# Binary Search Trees (BSTs)

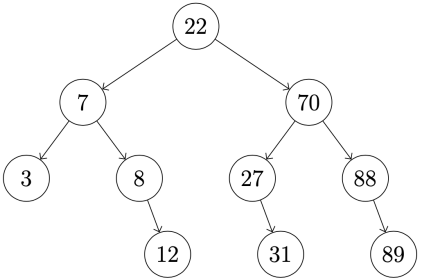
**Idea:** Organize keys in a binary tree for faster search.

## The Rule: Suchbaumbedingung

For every node $z$:

- All keys in left subtree $< z.key$.
- All keys in right subtree $> z.key$.

- **Operations:** Follow the rule to search for keys. Insert at leaves to maintain the search-tree condition.
- **Runtime:** $\mathbf{O(h)}$, where $h$ is the tree height.
- Problem: What if the tree becomes unbalanced?




A search tree with 10 nodes and height 3

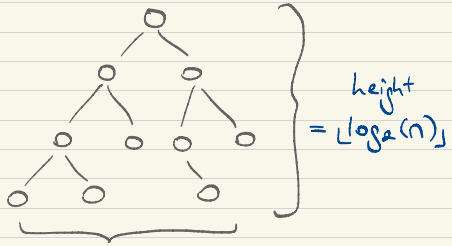
# Recap: Binary Tree Properties

Recall from last week (see slides week 5)

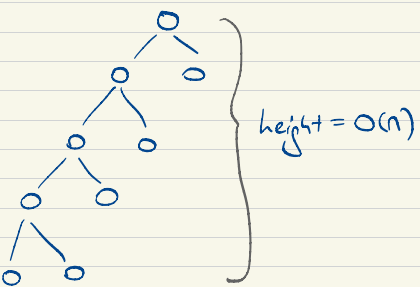
## Balanced

| level | # nodes |
|---|---|
| 0 | $2^0$ |
| 1 | $2^1$ |
| 2 | $2^2$ |
| $3 = h$ | $\leq 2^3$ |

total # nodes $\leq \sum_{i=0}^{h} 2^i = 2^{h+1} - 1$




## unbalanced




**Takeaway:** Ensure tree stays balanced (up to a degree) for efficient operations (get, insert, delete).

# Solution: 2-3 Trees (Our Version

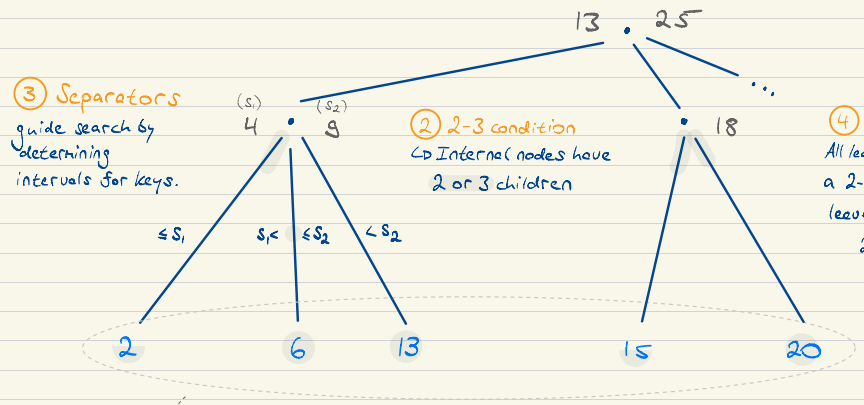
You saw in the lecture that the structure of a complete **binary tree is too rigid** for efficient insert and delete operations.

**2-3 Trees relax some constraints**, enabling efficient operations **while maintaining** the **important properties. Notably:**

1. **External Tree:** Keys are only in the leaves.
2. **2-3 Condition:** Internal nodes have 2 or 3 children
3. **Navigation Using Separators:** Internal nodes have separators (*not* keys) that determine the search intervals
4. **Guaranteed Logarithmic Height:** 2-3 Trees maintain a logarithmic height (derivation to follow)

**Disclaimer:** The notion of **2-3 Trees** as we introduce them in the lecture is **not standard**. **Be careful with material online** on this topic and stick to the script.

# Balanced Trees: 2-3 Trees (Our Version)




**3 Separators** guide search by determining intervals for keys.

**2 2-3 condition**
↳ Internal nodes have 2 or 3 children

**4 Logarithmic Height:**
All leaves are at the same height and since a 2-3 tree of height $h$ has at least $2^h$ leaves, it holds:

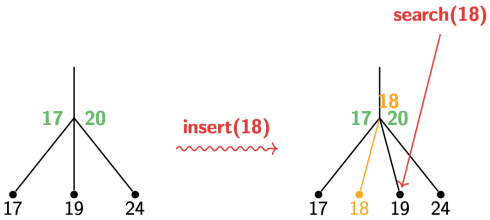
$$2^h \leq n \iff h \leq \log n \leq O(\log n)$$

**1 External tree:** Keys only in leaves

# 2-3 Trees: Insert Recap

**Steps:**

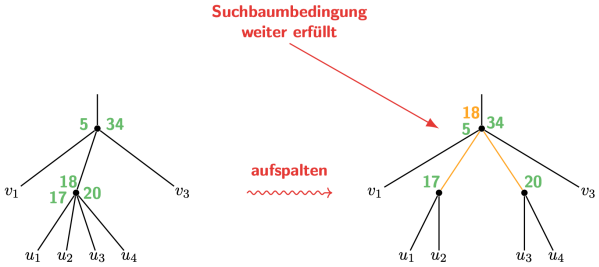
1. **Search & Insert:** Find position, add new leaf and parent separator.



# 2-3 Trees: Insert Recap

**Steps:**

1. **Search & Insert:** Find position, add new leaf and parent separator.
2. **Rebalance (if 4 children):**
   **Split** the node with 4 children into two nodes (2 children each).
   **Push** the middle separator up to the parent.



# 2-3 Trees: Insert Recap

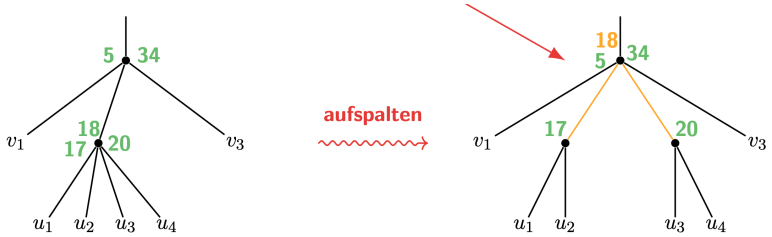
**Steps:**

1. **Search & Insert:** Find position, add new leaf and parent separator.
2. **Rebalance (if 4 children):**

   **Split** the node with 4 children into two nodes (2 children each).

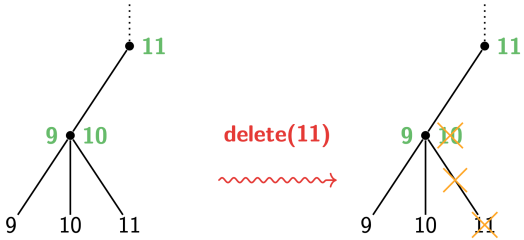
   **Push** the middle separator up to the parent.
3. **Propagate Up:** If pushing up causes the parent to have 4 children, repeat the split process recursively. A new root might be created if the original root splits.

**Runtime:** Search in $O(\log n)$ + at most $O(\log n)$ recursive splits $\implies$ $O(\log n)$ in total.



# 2-3 Trees: Delete Recap

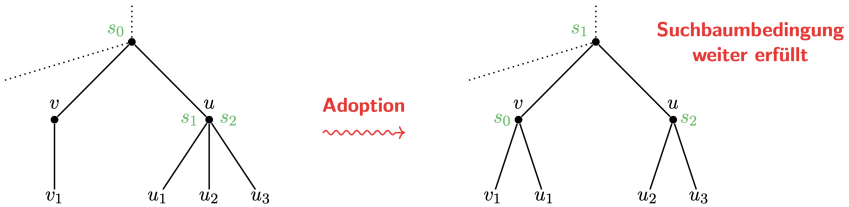
1. **Search & Remove:** Find leaf, remove it and parent separator.



# 2-3 Trees: Delete Recap

1. **Search & Remove:** Find leaf, remove it and parent separator.
2. **Rebalance (if 1 child):** Let node $v$ have 1 child. Check sibling $u$.

   **Case 1: Adoption (if $u$ has 3 children):** $v$ adopts a child from $u$. Redistribute separators.

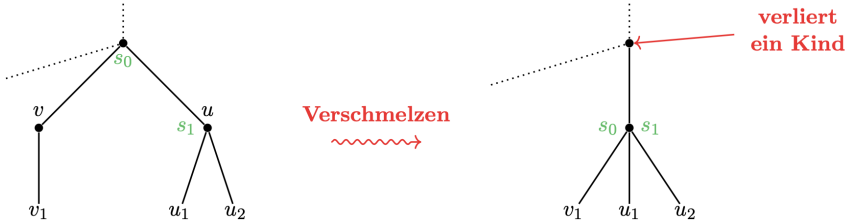


# 2-3 Trees: Delete Recap

1. **Search & Remove:** Find leaf, remove it and parent separator.
2. **Rebalance (if 1 child):** Let node $v$ have 1 child. Check sibling $u$.

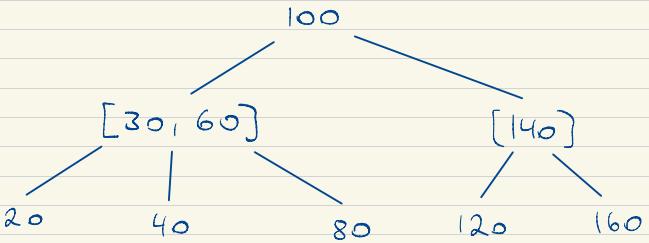
   **Case 1: Adoption (if $u$ has 3 children):** $v$ adopts a child from $u$. Redistribute separators.

   **Case 2: Merge (if $u$ has 2 children):** Combine $v$'s child and $u$'s children. Pull separator down from parent.
3. **Propagate Up:** Merging removes a child from the parent. May require recursive rebalancing upwards. Root might be removed if it ends up with 1 child.



## Worked Example:

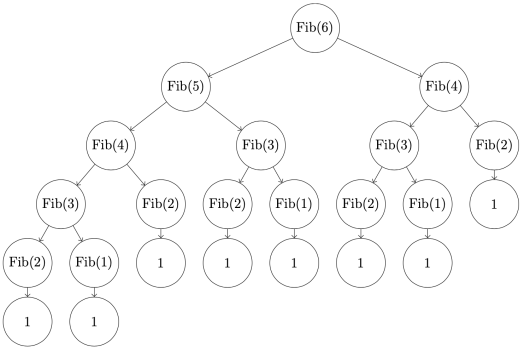
- Insert 50 into the following 2-3 tree:



# Dynamic Programming: The Idea

**What is it?** A technique to solve problems by breaking them down into simpler, *overlapping* subproblems. Solve each subproblem only once and store its result.

**Example: Calculating Fibonacci** $F_n = F_{n-1} + F_{n-2}$ Naive recursion is slow because it recalculates values like $F_3$ many times.




**DP avoids this re-computation.**

# DP Techniques: Memoization vs. Bottom-Up

**1. Memoization (Top-Down):** Start with the original problem and use recursion. Store results in a table ('memo') as you compute them. Check the table before computing.

```
memo = array filled with -1
FibM(n):
  if memo[n] != -1: return memo[n] // Already solved? Return stored value.
  if n <= 2: result = 1
  else: result = FibM(n-1) + FibM(n-2) // Compute recursively.
  memo[n] = result // Store result.
  return result
```

**2. Bottom-Up (Iterative):** Figure out the order needed. Start with base cases and compute solutions for progressively larger subproblems until you reach the final answer.

```
FibBU(n):
  F = new array[1..n]
  F[1] = 1; F[2] = 1 // Base cases.
  for i = 3 to n:
    F[i] = F[i-1] + F[i-2] // Compute using previous results.
  return F[n]
```

# Memoization vs. Bottom-Up: Comparison Table

| | Top-Down / Memoization | Bottom-Up / Iterative |
|---|---|---|
| **Pros** | • Often **mimics the recursive structure** directly, can be easier to think of initially.<br>• Only computes subproblems **actually needed** for the final answer.<br>• Computation order is handled **automatically** by recursion. | • Generally **faster** due to no recursion overhead.<br>• **Avoids** potential **stack overflow** errors from deep recursion.<br>• Often allows for space optimizations (e.g., using only the last row/few values). |
| **Cons** | • Suffers from function call **overhead**.<br>• Can hit **recursion depth limits** (stack overflow). | • Requires **careful thought** about the **correct order** to compute subproblems.<br>• Might compute subproblems not strictly needed for the specific final answer asked. |

**Note:** In this course, we will **always use Bottom-up**

# Your DP Strategy

Think of DP as exploring dependencies:

1. **Define Subproblem:** What piece of the final answer are you trying to calculate? Parameterize it (e.g., result for first $i$ items, result ending at index $j$).
2. **Find the Connection (Recurrence):** How can you get the answer for your current goal using answers you've *already* found for smaller/simpler goals? What choices do you have at each step?
   - Example: To get $F_n$, I need $F_{n-1}$ and $F_{n-2}$.
   - Example: To find the best path to $i$, consider the best paths to possible predecessors $j$.
3. **Identify the Starting Point (Base Cases):** What are the simplest goals you know the answer to without needing further connections? (e.g., $F_1, F_2$, empty string, first element)
4. **Extract the Solution:** Systematically calculate answers for your goals, starting from the base cases and using the connections, until you reach your main goal.

# Example 1: Jump Game

**Problem**

Min jumps from index 1 to $n$. From $i$, can jump to $j \in [i+1, i+A[i]]$.

**Intuition for Subproblem:** We want the minimum jumps to reach the **target** $n$. What information helps us get there? Knowing the minimum jumps to reach **intermediate positions** seems useful. If we know the minimum jumps to reach all positions $j < i$, maybe we can figure out the minimum jumps to reach $i$.

**Subproblem Attempt 1: $DP[i]$ = Min jumps to reach index $i$**

- **Recurrence:** To reach $i$, we must have jumped from some $j < i$ where $j + A[j] \geq i$. We want the best such $j$. $DP[i] = 1 + \min\{DP[j] \mid 1 \leq j < i, \quad j + A[j] \geq i\}$
- **Base Case:** $DP[1] = 0$.
- **Result:** $DP[n]$.
- **Runtime:** $O(n^2)$ - requires checking all previous $j$ for each $i$ ($\sum_{i=1}^{n-1} i = \Theta(n^2)$).

# Jump Game: Improving the Approach

**Alternative Angle:** Instead of focusing on the destination ($i$), let's focus on the number of jumps ($k$). What's the **farthest we can reach** in $k$ jumps?

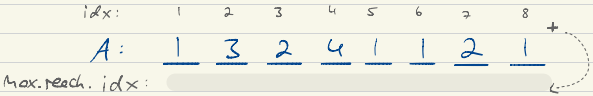
## Subproblem Attempt 2: $DP[i]$ = Max index reachable in $i$ jumps

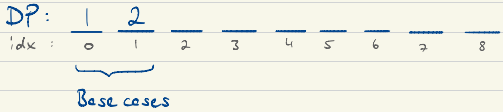
- **Recurrence (Improved):** The farthest reach with $k$ jumps ($DP[k]$) is found by considering the farthest jump possible from any position $j$ that was **newly** reachable in exactly $k-1$ jumps (i.e., $DP[k-2] < j \leq DP[k-1]$).

$$DP[k] = \max\{j + A[j] \mid DP[k-2] < j \leq DP[k-1]\}$$

- **Base Cases:** $DP[0] = 1,\ DP[1] = 1 + A[0]$.
- **Result:** Smallest $k$ where $M[k] \geq n$.
- **Runtime:** Each index $j$ is considered in the max only once over the entire algorithm. $O(n)$.

# Worked Example:




search range for DP[i] is:

$\square < i \leq \square$

Runtime = $O(n)$ since we consider every entry in A only once (compare coloring and search range).

# Example 2: Longest Common Subsequence (LGT)

**Problem**

Given $A[1..n], B[1..m]$. Find length of longest shared subsequence (not necessarily contiguous).

**Intuition for Subproblem:** We are comparing two sequences. The problem involves making decisions based on corresponding characters.

- Trying $L[i] =$ LGT of $A[1..i], B[1..i]$ fails because the optimal solution for prefixes doesn't necessarily extend to the optimal solution for longer strings. We lose crucial info about where the subsequence ends.
- We need to track progress in **both** strings independently. This points towards a 2D subproblem involving prefixes of both strings.

# Example 2: Longest Common Subsequence (LGT)

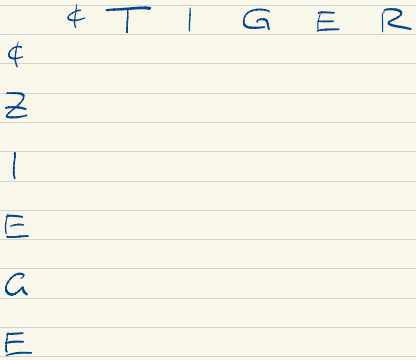
**Problem**

Given $A[1..n], B[1..m]$. Find length of longest shared subsequence. E.g., A=TIGER, B=ZIEGE $\implies$ LGT="IGE", Length=3.

- **Subproblem:** $L[i,j]$ = Length of LGT for $A[1..i]$ and $B[1..j]$
- **Recurrence:** Consider $a_i$ and $b_j$.
  - If $a_i = b_j$: They match! Include this character. $L[i,j] = 1 + L[i-1, j-1]$.
  - If $a_i \neq b_j$: We can't use both. The LGT is the best we can get by either ignoring $a_i$ ($L[i-1,j]$) or ignoring $b_j$ ($L[i,j-1]$). $L[i,j] = \max\{L[i-1,j], L[i,j-1]\}$.
- **Base Cases:** $L[i,0] = 0$, $L[0,j] = 0$.
- **Result:** $L[n,m]$.
- **Runtime:** Fill $n \times m$ table, $O(1)$ per cell $\implies$ $\mathbf{O(n \cdot m)}$.

# Worked Example:

**Find the Longest Common Subsequence of "TIGER", "ZIEGE"**



# Example 3: Editierdistanz (Edit Distance)

**Problem**

Min operations (insert, delete, replace) to transform $A \to B$. E.g., TIGER $\to$ ZIEGE needs 3 ops.

**Intuition for Subproblem:** Very similar structure to LGT – transforming one sequence into another using character operations. We're comparing prefixes again. The same logic applies: we need to track progress on both strings using two indices.

**Have a look at the script for the specifics or message me in case anything is unclear!**

Questions?

# Additional Practice

# DP - Min. Cost Stairs

**Description:**

You are given an integer array cost where cost[i] is the cost of ith step on a staircase. Once you pay the cost, you can either climb one or two steps.

You can either start from the step with index 0, or the step with index 1.

Return the minimum cost to reach the top of the floor.

**Example:**

- Input: cost = [10,15,20]
- Output: 15
- Explanation: You will start at index 1, Pay 15 and climb two steps to reach the top. The total cost is 15.

# DP - Min. Cost Stairs (1/2)

Compute the solution using bottom-up dynamic programming and state the run time of your algorithm.

Address the following aspects in your solution:

1. **Definition of the DP table:** What are the dimensions of the table DP [. . .]? What is the meaning of each entry?

2. **Computation of an entry:** How can an entry be computed from the values of other entries? Specify the base cases, i.e., the entries that do not depend on others.

*see next page for cont.*

# DP - Min. Cost Stairs (2/2)

3. **Calculation order:** In which order can entries be computed so that values needed for each entry have been determined in previous steps?

4. **Extracting the solution:** How can the final solution be extracted once the table has been filled?

5. **Runtime:** What is the run time of your solution?

**When you're done, try implementing your solution on CodeExpert!**

# Peer Grading

# Peer-Grading Exercise

This week's peer-grading exercise is **Exercise 5.4**

Please follow the usual process:

- Grade the assigned group's submission.
- Give constructive feedback and upload your feedback to Moodle by 23.59 tonight, at the latest.
- Contact me if you don't receive a submission to grade.