Week 5

Quicksort, Heapsort & ADTs

Thorben Klabunde

www.th-kl.ch

20.10.2025

Agenda

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

Mini-Quiz

Mini-Quiz 4

1. **CI.:** $5e^{2\ln(n)} = \Theta(e^{10\ln(n)})$.

   Answer: False. We have $5e^{2\ln(n)} = 5(e^{\ln(n)})^2 = 5n^2$ and $e^{10\ln(n)} = (e^{\ln(n)})^{10} = n^{10}$. We check the limit:

   $$\lim_{n\to\infty} \frac{5e^{2\ln(n)}}{e^{10\ln(n)}} = \lim_{n\to\infty} \frac{5n^2}{n^{10}} = \lim_{n\to\infty} \frac{5}{n^8} = 0.$$

2. **CI.:** Suppose you have some algorithm A and let an increasing function $T(n)$ be its runtime on an input of size n. Assume you know that $T(n) \leq T(n-1) + 100n$ and that $T(1) = 10$. Then $T(n) \leq O(n^2)$.

   Answer: True. Expanding out $T(n) \leq T(n-1) + 100n \leq$
   $T(n-2) + 100(n-1) + 100n \leq \cdots \leq T(1) + 100(\sum_{i=1}^{n} i) = 10 + 100\frac{n(n+1)}{2} = O(n^2)$.

3. **CI.:** Every comparison-based sorting algorithm needs to perform at least $\Omega(n\log(n))$ swaps.

   Answer: False. Selection sort only uses $O(n)$ swaps.

Mini-Quiz 4

4. Suppose we apply insertion sort to the array $A_n = [n, n-1, \ldots, 2, 1]$. (E.g., $A_5 = [5, 4, 3, 2, 1]$). Let $s(n)$ be the number of **swap** operations that are performed before the array is fully sorted (in ascending order).

   Select one:

   ✓ **a.** $s(n) \geq \Omega(n^2)$

   At step j, we need to move $A[j] = n - j + 1$ all the way to the first index via $j - 1$ many swaps. So a total of $\sum_{j=2}^{n}(j-1) = \Omega(n^2)$ swaps.

   b. $s(n) \leq O(n)$

5. Which of the following sorting algorithms have the invariant that: after j steps, the first j elements are sorted? (A step refers to one iteration of the outer loop for each sorting algorithm).

   Select one or more:

   a. Merge sort

   b. Bubble sort

   ✓ **c.** Insertion sort

   From lecture, insertion sort has the invariant that keeps the first j indices are sorted after j

Mini-Quiz 4

6. **Cl.:** Asymptotically (in Θ-notation), *merge sort*, *heap sort*, and *deterministic quick sort* all have the same worst-case runtime.

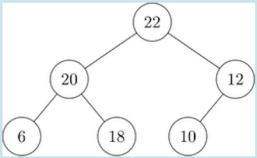
   Answer: False. Deterministic *quick sort* has a worst-case runtime of $\Theta(n^2)$.

7. **Cl.:** Suppose all keys in a max-heap are unique. Then the node with the minimum key is always the left-most leaf.

   Answer: False. Consider the valid max-heap with root 3, left child 2, and right child 1. The minimum key (1) is the right leaf, not the left-most.

Mini-Quiz 4

Consider the following max-heap,



After executing one **`ExtractMax`** operation, what is the correct max-heap?

Assignment

Ex. 4.4a)

Exercise 4.4 *Searching for the summit* **(1 point)**.

Suppose we are given an array $A[1 \dots n]$ with n **unique** integers that satisfies the following property. There exists an integer $k \in [1, n]$, called the *summit index*, such that $A[1 \dots k]$ is a strictly increasing array and $A[k \dots n]$ is a strictly decreasing array. We say an array is **valid** is if satisfies the above properties.

(a) Provide an algorithm that find this k with worst-case running time $O(\log n)$. Give the pseudocode and give an argument why its worst-case running time is $O(\log n)$.

Note: Be careful about edge-cases! It could happen that $k = 1$ or $k = n$, and you don't want to peek outside of array bounds without taking due care.

Ex. 4.4a)

Algorithm 2 Find the summit

function FINDSUMMITINDEX(T, i, j)
- $m \leftarrow \lfloor (i+j)/2 \rfloor$
- **if** $j = i$ **then**
  - **return** i
- **if** $T[m+1] < T[m]$ **then** ▷ m is right of the summit (or is the summit)
  - **return** FINDSUMMITINDEX(T, i, m) ▷ keep searching in the left half
- **else** ▷ m is strictly left of the summit
  - **return** FINDSUMMITINDEX($T, m+1, j$) ▷ keep searching in the right half

Input: Valid array T of length n with unique elements
Output: FINDSUMMITINDEX($T, 1, n$)

Let $A(n)$ be the worst-case runtime on an array of length n.

Notice that by the above pseudocode it holds: $A(n) \leq A(\frac{n}{2}) + C$, for some $C \in \mathbb{N}$.

To apply the Master Theorem, we rewrite the expression as: $A(n) \leq 1 \cdot A(\frac{n}{2}) + C \cdot n^0$

We apply the M.T. with $\log a = \log 1 = 0 = b$ and obtain $A(n) \leq O(\log n)$.

Ex. 4.4b)

(b) Given an integer x, provide an algorithm with running time $O(\log n)$ that checks if x appears in the (valid) array or not. Describe the algorithm either in words or pseudocode and argue about its worst-case running time.

Ex. 4.4b)

Algorithm 3 Search in a valid array

Input: Valid integer array T of length n with unique elements, integer x

$k \leftarrow$ FINDSUMMITINDEX$(T, 1, n)$

$k_1 \leftarrow \text{BS}^{\uparrow}(T[1..k], x)$ ▷ search in array $T[1..k]$, sorted in ascending order

$k_2 \leftarrow \text{BS}^{\downarrow}(T[k+1..n], x)$ ▷ search in array $T[k+1..n]$, sorted in descending order

Output: k_1 or k_2

Ex. 4.5a) - Counting Function Calls

Algorithm 4

```
i ← 1
while i ≤ n do
    j ← i
    while 2^j ≤ n do
        f()
        j ← j + 1
    i ← i + 1
```

$2^j \leq n \iff j \leq \lfloor \log(n) \rfloor$

$$\sum_{i=1}^{n} \sum_{j=i}^{\lfloor \log(n) \rfloor} 1 = \sum_{i=1}^{n} \max(0, \lfloor \log(n) \rfloor - i + 1)$$

(use max operator to figure out index range where the inner sum is empty)

$$= \sum_{i=1}^{\lfloor \log(n) \rfloor} (\lfloor \log(n) \rfloor - i + 1)$$

(leave out iterations where no function call occurs (empty inner sum))

$$= \sum_{i=1}^{\lfloor \log(n) \rfloor} i$$

(reverse sum & adjust indices)

$$= \frac{\lfloor \log(n) \rfloor \lfloor \log(n) \rfloor + 1}{2} = \Theta(\log(n)^2)$$

Ex. 4.5b) - Counting Function Calls

Algorithm 5

```
function A(n)
    i ← 0
    while i < n^2 do
        j ← n
        while j > 0 do
            f()
            f()
            j ← j − 1
        i ← i + 1
    k ← ⌊n/2⌋
    for l = 0 . . . 3 do
        if k > 0 then
            A(k)
            A(k)
```

You may assume that the function $T : \mathbb{N} \to \mathbb{R}^+$ denoting the number of calls of the algorithm to f is increasing.

Ex. 4.5b) - Counting Function Calls

Let $T(n)$ be the total number of calls to f. Notice from the pseudocode that:

$T(1) = 2$ and for $n \geq 2$ we get the recurrence:

$$T(n) = \underbrace{\sum_{i=0}^{n^2-1} \sum_{j=1}^{n} 2}_{} + 8 \cdot T\left(\frac{n}{2}\right)$$

$$= n^2(2n) + 8 \cdot T\left(\frac{n}{2}\right)$$

$$= 8 \cdot T\left(\frac{n}{2}\right) + 2n^3$$

Applying the Master Theorem with $a = 8$, $b = 3$ and $c = 2$, we find: $T(n) = \Theta(n^3 \log(n))$

Theory Recap

This Week's Big Picture

This week, we finished sorting and introduced the concept of Abstract Data-Types (ADTs) and first data structures. Key Takeaways:

- **Efficient Sorting and Trade-Offs**
  Quicksort uses a clever partitioning scheme, while Heapsort introduces a new data structure, the heap, to sort efficiently in-place.

- **Familiarization with Tree-Based Data-Structures**
  You should become gain intuition for tree-based data structures. In particular you should feel comfortable describing and reasoning about the properties of tree-based data-structures.

- **Abstract Data Types (ADTs)**
  You should understand the concept of separating an interface (an ADT) from its implementation (a data-structure) and understand the trade-offs involved for specific ADTs (this week: lists).

Sorting Algorithms: A Summary

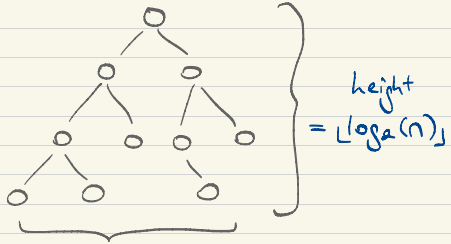
Algorithm	Comparisons	Swaps	Extra Space	Locality
BubbleSort	$O(n^2)$	$O(n^2)$	$O(1)$	good
SelectionSort	$O(n^2)$	$O(n)$	$O(1)$	good
InsertionSort	$O(n \log n)$	$O(n^2)$	$O(1)$	good
Mergesort	$O(n \log n)$	$O(n \log n)$	$O(n)$	good

Recap: Binary Tree Properties

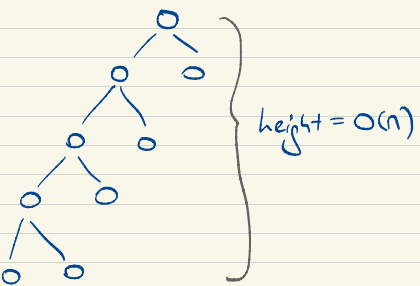
Balanced

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



We generally want to ensure our tree-based data structures are balanced (up to a degree) to ensure better properties.

Theoretical Lower Bound for Comparison-Based Sorting

How can we prove that $O(n \log n)$ is the best possible runtime for sorting?

Key Idea: The Decision Tree for Comparison-Based Sorting

Any comparison-based sorting algorithm can be modeled as a **decision tree**.

- Each **internal node** represents a comparison (e.g., $A[i] < A[j]$?).
- Each **leaf node** represents one of the possible sorted outcomes.

The worst-case number of comparisons is the height h of the tree.

For an input array A of size n with distinct elements, how many possible sorted outcomes (permutations) are there?

- There are **n**! possible permutations of the input array. The algorithm must be able to distinguish between all of them.
- Each of these $n!$ outcomes must correspond to at least one leaf in the decision tree.

Deriving the Lower Bound for Sorting

Theorem: Lower Bound for Comparison-Based Sorting

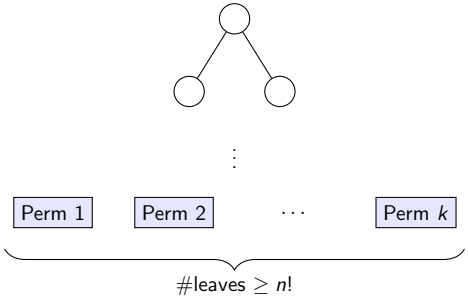
Any comparison-based sorting algorithm requires at least $\mathbf{\Omega(n \log n)}$ comparisons in the worst case.

Proof Sketch:

1. The algorithm must distinguish between $\mathbf{n}!$ possible permutations, so the decision tree must have at least $n!$ leaves.
2. A binary tree of height h has at most $\mathbf{2^h}$ leaves.
3. Therefore, the number of leaves must be at least the number of outcomes: $n! \leq 2^h \implies h \geq \log_2(n!)$
4. Using the property that $\log(n!) = \Theta(n \log n)$, we get:

$$h \geq \Omega(n \log n)$$

The worst-case runtime is at least logarithmic in the number of outcomes.



Sorting in $O(n)$ - A Contradiction?

No! The $\Omega(n \log n)$ bound is not violated.

That theorem **only** applies to **comparison-based sorting** algorithms. We can achieve $O(n)$ if we **make assumptions** about the data and use its *value* directly.

Example: Counting Sort

- **Key Assumption:** The n input elements are **integers** in a known, finite range, e.g., $[\mathbf{0}, \mathbf{k}]$.
- **Core Idea:** We don't sort the elements at all. We just *count* the occurrences of each element and then rebuild the array.
- **How it works:**
  1. Create a "count" array C of size $k + 1$, initialized to zeros.
  2. **Count ($O(n)$):** Iterate the input A. For each element x, increment its counter: $\mathbf{C[x] += 1}$.
  3. **Rebuild ($O(n + k)$):** Iterate C from $i = 0$ to k. For each i, add the value i to the output array $C[i]$ times.
- **Runtime: $\mathbf{O(n + k)}$**

Quicksort

Idea: A "divide and conquer" algorithm where the main work happens in the dividing step, not the combining step.

1. **Partition:** Pick an element, the **pivot**. Rearrange the array such that all elements smaller than the pivot come before it, while all elements greater come after it.

   $\Longrightarrow$ **Invariant:** After the partitioning, the pivot is in its final sorted position.
2. **Conquer:** Recursively apply Quicksort to the sub-array of smaller elements and the sub-array of larger elements.
3. **Combine:** No work needed! The array is sorted once the recursive calls return.

Quicksort

Idea: A "divide and conquer" algorithm where the **main work happens in the dividing step**, not the combining step.

1. **Partition:** Pick an element, the **pivot**. Rearrange the array such that all elements smaller than the pivot come before it, while all elements greater come after it. After this partitioning, the pivot is in its final sorted position.
2. **Conquer:** Recursively apply Quicksort to the sub-array of smaller elements and the sub-array of larger elements.
3. **Combine:** No work needed! The array is sorted once the recursive calls return.

```
void quicksort(int l, int r, int[] arr) {
    if (l>=r) return;
    int p = arr[r];

    int p_idx = partition(l, r, p, arr);
    quicksort(l, p_idx-1, arr);
    quicksort(p_idx+1, r, arr);
}
```

```
int partition(int l, int r, int p, int[] arr) {
    int i = l; int j = r-1;
    while(true){
        while(i < r && arr[i] <= p) i++;
        while(j > i && arr[j] > p) j--;
        if (i>=j) break;
        swap(i, j, arr);
    }
    swap(i, r, arr);
    return i;
}
```

Quicksort: Partitioning Analysis

Runtime Analysis

Note: The **runtime depends** on the **pivot selection**!

- **Best/Average Case:** The pivot splits the array into two roughly equal halves.
  - Recurrence: $T(n) = 2T(n/2) + \Theta(n)$
  - Solution: $T(n) = \mathbf{O(n \log n)}$
- **Worst Case:** The pivot is always the smallest or largest element (e.g., when the array is already sorted and you pick the last element as pivot).
  - Recurrence: $T(n) = T(n-1) + \Theta(n)$
  - Solution: $T(n) = \mathbf{O(n^2)}$

Note: The worst case is rare, especially if the pivot is chosen randomly. You will prove next term in Algorithms and Probability that for a randomly chosen index, QuickSort has an expected runtime of $O(n \log n)$

The Max-Heap Data Structure

New Approach: Use a **special data structure** that **keeps elements in order** to find / extract the maximum element efficiently.

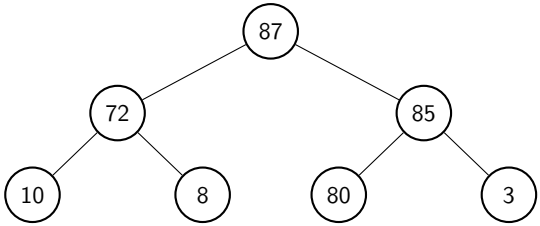
A Max-Heap is a binary tree that is:

1. **Complete:** All levels are full, except possibly the last, which is filled from left to right.
2. **Satisfies the Max-Heap Property:** The key of any node is greater than or equal to the keys of its children.

Key Consequence: The largest element is always at the root of the tree!

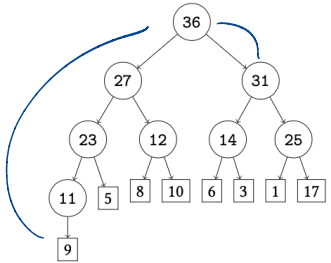
Key Properties - Since a Heap is a complete binary tree:

- Depth $\leq \lfloor \log(n) \rfloor$
- # Leaves $\leq \lceil n/2 \rceil$



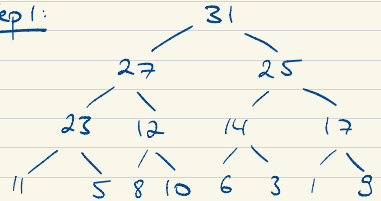
Your-Turn: Max-Heap Operations

(b) Consider the following max-heap:

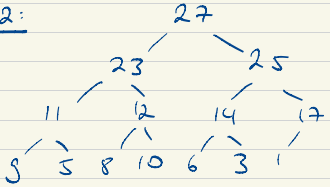


Draw the max-heap after two ExtractMax operations.

Step 1:



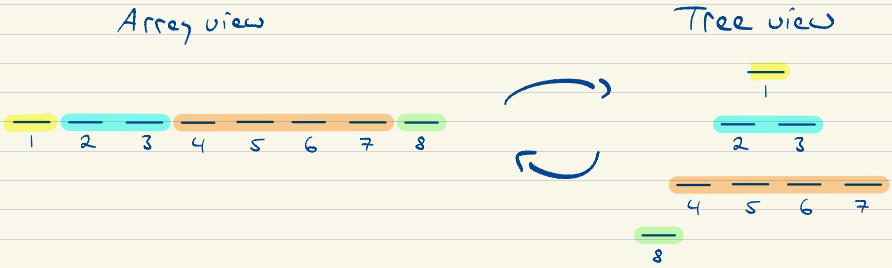
Step 2:



Implementing a Max-Heap using Arrays

Array Representation: A heap is typically stored in an array. For a node at index k (using 1-based indexing):

- Its parent is at index $\lfloor k/2 \rfloor$.
- Its left child is at $2k$.
- Its right child is at $2k + 1$.



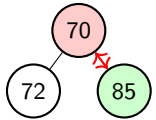
Heapsort

HeapSort proceeds in two phases:

1. **Build-Max-Heap:** First, convert the unsorted input array into a max-heap (or start with an empty heap and add elements)
2. **Sort:** Repeatedly perform the following steps:
   1. **Swap** the maximum element $A[1]$ with the last element of the current heap, $A[i]$.
   2. **Reduce heap size** by one. The swapped element is now in its final sorted position.
   3. **Restore Heap Property:** The new root may violate the heap property. Fix by letting the element "sink" to its correct position.

```
Heapsort(A)
  n = A.length
  for i = floor(n/2)..1 do: // 1. Build the initial max-heap
    RestoreHeapCondition(A, i, n)
  for i = n..2 do: // 2. Extract elements one by one
    swap(A[1], A[i])
    RestoreHeapCondition(A, 1, i-1)
```

RestoreHeapCondition (Sift-Down):



Swap with **largest child** until restored.

Heapsort: Analysis Properties

Runtime Analysis

- **Build-Heap:** The first loop runs $\approx n/2$ times. Each 'RestoreHeapCondition' call takes at most $O(\log n)$ time

  $\implies$ $O(n \log n)$ in total (a tighter analysis shows it's actually $O(n)$).

- **Sorting Phase:** The second loop runs $n-1$ times. The i-th iteration takes $O(\log i)$ for the 'RestoreHeapCondition' call. The total time is $\sum_{i=2}^{n} O(\log i) = \mathbf{O(n \log n)}$.

Additionally:

- **Space Complexity:** $O(1)$ extra space for in-place sorting.
- **Locality:** Not very good. Accessing parents and children can involve large jumps in memory, which is not cache-friendly.

Abstract Data Types (ADT)

A fundamental concept in computer science is the **separation of interface and implementation**.

Abstract Data Type (ADT)

An ADT is a model for a data type. It specifies:

- A set of **objects** (i.e., the type of data being stored).
- A set of **operations** that can be performed on these objects.

An ADT defines **what** can be done, but **not how** it is done.

A **Data Structure** is a concrete implementation of an ADT.

Example: The List ADT

- **Objects:** A collection of items in a fixed sequence.
- **Operations (a selection):**
  - 'insert(k, L)': Add an item with key 'k' to the end of list 'L'.
  - 'get(i, L)': Return the i-th item in 'L'.
  - 'delete(o, L)': Remove object 'o' from list 'L'.

Data Structures for the List ADT

The choice of data structure affects the performance of the ADT's operations. Let's compare three common implementations for the List ADT.

Operation	**Array**	**Singly Linked List**	**Doubly Linked List**
get(i)	$O(1)$	$O(i)$	$O(i)$
insert (at end)	$O(1)$	$O(n)$*	$O(1)$
insertAfter(o, k)	$O(n)$	$O(1)$	$O(1)$
delete(o)	$O(n)$	$O(n)$	$O(1)$

Assumes a pointer/index to object 'o' is given for 'insertAfter' and 'delete'.
*$O(1)$ if we keep a pointer to the tail.

Key Trade-off: Arrays provide fast random access (get(i)), while linked lists provide fast insertions and deletions in the middle.

Questions?

Additional Practice

/ 4 P d) *Induction:* Prove by mathematical induction that for any positive integer n,

$$\sum_{k=1}^{n} \frac{1}{k^2} \leq 2 - \frac{1}{n}.$$

cl. $\forall n \in \mathbb{N}^+ \left(\sum_{k=1}^{n} \frac{1}{k^2} \leq 2 - \frac{1}{n} \right)$

pr. We proceed by induction on n.

B.C. Let $n = 1$ and notice: $\sum_{k=1}^{1} \frac{1}{k^2} = 1 \leq 2 - \frac{1}{1}$,

wherefore the B.C. holds.

I.H. Assume the given inequality holds for some $n \in \mathbb{N}^+$.

I.S. Then for $n+1$, it holds:

$$\sum_{k=1}^{n+1} \frac{1}{k^2} = \left(\sum_{k=1}^{n} \frac{1}{k^2} \right) + \frac{1}{(n+1)^2}$$

$$\overset{\text{I.H.}}{\leq} \left(2 - \frac{1}{n}\right) + \frac{1}{(n+1)^2}$$

$$= 2 - \frac{(n+1)^2}{n(n+1)^2} + \frac{n}{n(n+1)^2} \quad \left| \text{(bring fractions to common denominator)} \right.$$

$$= 2 - \frac{n^2 + 2n + 1 - n}{n(n+1)^2} \leq 2 - \frac{n^2 + n}{n(n+1)^2} \quad \left| \begin{array}{l} \text{note, } \forall n \in \mathbb{N} \\ -(n^2+n+1) \leq -(n^2+n) \end{array} \right.$$

$$= 2 - \frac{n(n+1)}{n(n+1)^2} = 2 - \frac{1}{(n+1)}$$

Peer Grading

Peer-Grading Exercise

This week's peer-grading exercise is **Exercise 4.4**

Each group grades the group below in the table I sent you (resp. the last one grades the first one). Please send the other group your solution. If you don't get their solution, please contact me so I can send it to you.