Inhaltsverzeichnis

1 Sorting

Problem:

- n elements $\mathbf{x} = (x_1, x_2, ..., x_n)$
- Output: $\mathbf{x}^*$ ordered s.t. $x_i^* \leq x_{i+1}^*$

1.1 MinSort

Complexity:	$\mathcal{O}(n^2)$

Tabelle 1: Minsort attributes

1. Find the minimum and switch the value with the *first* position.
2. Find the minimum and switch the value with the *second* position.
3. ...

```python
def minsort ( elements ) :
    for i in range(0,len(elements)-1):
        minimum = i
        for j in range(i+1,len(elements)):
            if elements[j] < elements[minimum]:
                minimum = j
                if minimum != i :
                    elements[i],elements[minimum]=\
                        elements[minimum],elements[i]
    return elements
```

Code snippet 1: minsort()

1.2 Heapsort

1.2.1 Binary heap:

- Binary tree (preferably complete)
- **Heap property:** Each child is smaller(/larger) than the parent element.

Children of node i: $2i + 1$ and $2i + 2$
Parent of node i: floor($\frac{i-1}{2}$)

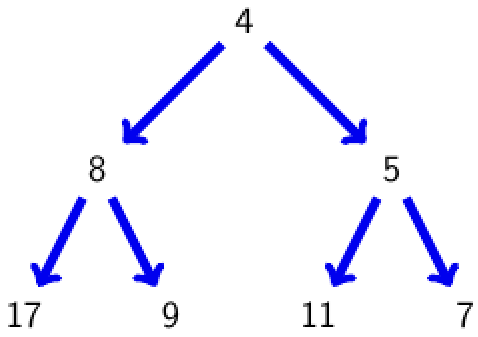


Abbildung 1: Valid min heap

1.2.2 Algorithm

Sifting down: Check whether current node violates the heap condition. If so: Switch with smaller child and repeat step with the new child until you reach the bottom.
Heapsort():

1. Heapify list by sifting down from the bottom up.
2. While elements are in the heap
   a) Remove root element and add it to the sorted list.
   b) Put the last element in the heap to the root position.
   c) Sift down from the root position

1.2.3 Attributes

- First: *heapify* array of n elements
  - Depends on depth of tree
  - In general: costs are linear with path length and number of nodes.
- Then: until all n elements are sorted:
  - constant stuff
  - sifting

Total runtime: $T(n) \leq 6 \cdot n \log_2 n \cdot C$

2 Runtime

Runtime is dependent on (other than efficiency of code):

- Specs of the computer
- Applications in the background
- Compiler effiency

2.1 $\mathcal{O} - Notation$

$$f \in \mathcal{O}(g) \Rightarrow f(n) \leq C \cdot g(n) \forall$$

Formal:

$$\mathcal{O}(g) = \{f : \mathbb{N} \to \mathbb{R} | \exists n_0 \in \mathbb{N}, \exists C > 0, \forall n > n_0 : f(n) \leq C \cdot g(n)\} \tag{1}$$

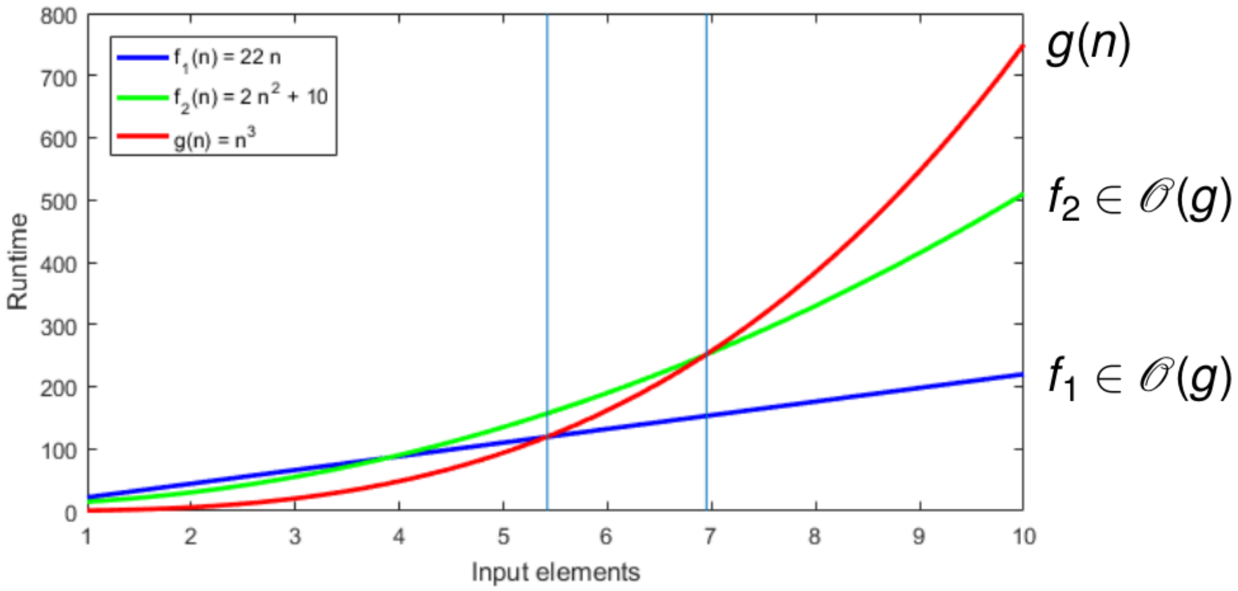


Abbildung 2: Illustration of $\mathcal{O}$

- We are only interested in the term with the highest order (i.e. the fastest growing summand), others are ignored.
- $f(n)$ is limited *from above* by $C \cdot g(n)$

2.2 Ω-Notation

$f \in \Omega(g) \Rightarrow f$ is growing at least as fast as g.

$$f \in \mathcal{O}(g) \Rightarrow f(n) \geq C \cdot g(n) \forall$$

Formal:

$$\Theta(g) = \{f : \mathbb{N} \to \mathbb{R} | \exists n_0 \in \mathbb{N}, \exists C > 0, \forall n > n_0 : f(n) \geq C \cdot g(n)\}$$

- We are only interested in the term with the highest order (i.e. the fastest growing summand), others are ignored.
- $f(n)$ is limited *from below* by $C \cdot g(n)$

2.3 Θ-Notation

$f \in \Theta(g) \Rightarrow f$ is growing at the same rate as g.

$$f \in \mathcal{O}(g) \Rightarrow f(n) \geq C \cdot g(n) \forall$$

Formal:

$$\Theta(g) = \mathcal{O}(g) \cap \Omega(g)$$

2.4 Summary

- With $\mathcal{O}$ notation we're interested in $n \to \infty$.
- $\mathcal{O}$ only applies for $n \geq n_0$.
- **Attention:** n_0 does **not** have to be a small number.

2.4.1 Limits

$$f \in \mathcal{O}(g) \Leftrightarrow \lim_{N\to\infty} \frac{f(n)}{g(n)} < \infty \tag{2}$$
$$f \in \Omega(g) \Leftrightarrow \lim_{N\to\infty} \frac{f(n)}{g(n)} > 0 \tag{3}$$
$$f \in \Theta(g) \Leftrightarrow 0 < \lim_{N\to\infty} \frac{f(n)}{g(n)} < \infty \tag{4}$$
$$\tag{5}$$

2.4.2 Algebraic rules

Transivity

$$f \in \mathcal{O}(g) \wedge g \in \mathcal{O}(h) \Rightarrow f \in \mathcal{O}(h) \tag{6}$$
$$f \in \Omega(g) \wedge g \in \Omega(h) \Rightarrow f \in \Omega(h) \tag{7}$$
$$f \in \Theta(g) \wedge g \in \Theta(h) \Rightarrow f \in \Theta(h) \tag{8}$$
$$\tag{9}$$

Symmetry

$$f \in \mathcal{O}(g) \Leftrightarrow g \in \Omega(f) \tag{10}$$
$$f \in \Theta(g) \Leftrightarrow g \in \Theta(f) \tag{11}$$

Reflexivity

$$f \in \Theta(f), f \in \Omega(f), f \in \mathcal{O}(f) \tag{12}$$

Trivial

$$f \in \mathcal{O}(f) \tag{13}$$
$$k \cdot \mathcal{O}(f) = \mathcal{O}(f) \tag{14}$$
$$\mathcal{O}(f + k) = \mathcal{O}(f) \tag{15}$$

Addition

$$\mathcal{O}(f) + \mathcal{O}(g) = \mathcal{O}(\max\{f, g\}) \tag{16}$$

Multiplication

$$\mathcal{O}(f) \cdot \mathcal{O}(g) = \mathcal{O}(f \cdot g) \tag{17}$$

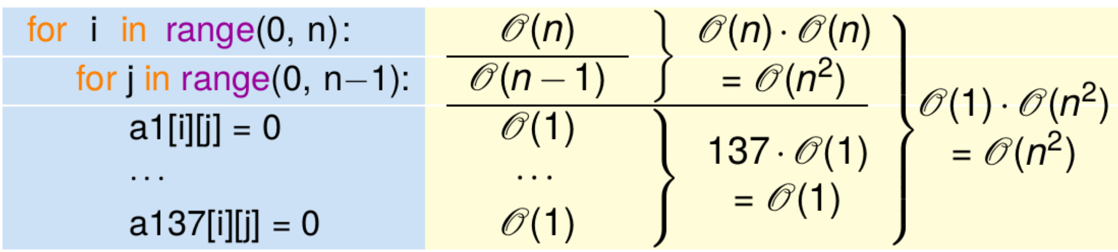


Abbildung 3: Behavior of $\mathcal{O}$ in loops

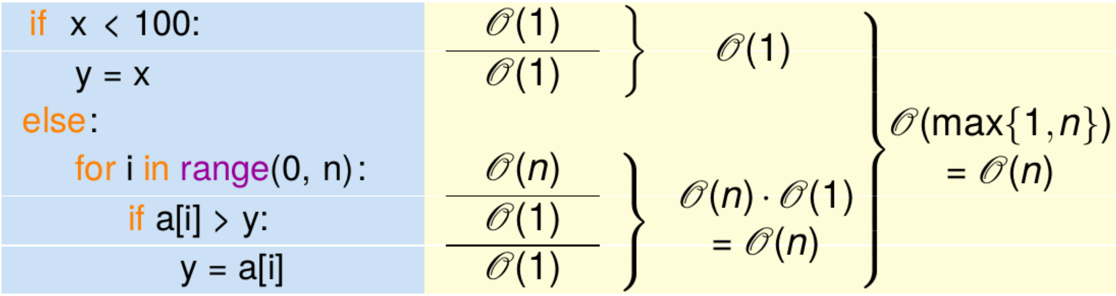


Abbildung 4: Behavior of $\mathcal{O}$ in conditions

Tabelle 2: Common runtime types

Runtime	Growth in time
$f \in \Theta(1)$	Constant
$f \in \Theta(\log_k n)$	Logarithmic
$f \in \Theta(n)$	Linear
$f \in \Theta(n \log n)$	n-log-n time (almost linear)
$f \in \Theta(n^2)$	Squared time
$f \in \Theta(n^3)$	Cubic time
$f \in \Theta(n^k)$	Polynomial time
$f \in \Theta(k^n),\ f \in \Theta(2^n)$	Exponential Time

3 Associative array

Associative arrays are arrays in which you access the elements not via index, but via a *key*.

$$A[\text{“Mueller”}] = \text{“0140-373830”}$$

Disadvantage: Lookup takes long ($\Theta(n)$)

4 Hashmap

Idea: Mapping the keys onto indices with a *hash function* h and store the data in a regular array.

- **Advantage:** Lookup takes $\Theta(1)$ (in the best case).
- **Problem:** If $h(x_i) = h(x_j), x_i \neq x_j \Rightarrow$ a *Collision* occurs. (Quite common, see the Birthday problem)

4.1 Buckets

Simple solution to collision: Lists (buckets) as entries to hashmaps

- Best case: n keys equally distributed over m buckets $\Rightarrow \approx \frac{m}{n}$
- Worst case: All n keys mapped onto the same bucket (*degenerated hash table*) $\Rightarrow$ Searching runtime $\Theta(n)$

4.2 Universal hashing

- Way of avoiding degenerated hash tables
- Define a set of hash functions.
- Choose a random hash function so that the expected result is an equal distribution over the buckets.
- Since a big universe is mapped onto a small set, no hash function is good/suitable for all key sets

Definition

- $\mathbb{U}$: Universe of possible keys
- $\mathbb{S} \subseteq \mathbb{U}$: Set of used keys
- m: Size of the hash table T
- $\mathbb{H} = \{h_1, h_2, ..., h_n\}$: Set of hash functions with $h_i : \mathbb{U} \to \{0, ..., m-1\}$

$\Rightarrow \frac{|\mathbb{S}|}{m} :=$ *table load*

- Runtime should be $\mathcal{O}(1 + \frac{|\mathbb{S}|}{m})$

$\mathbb{H}$ is *c-universal* $\Leftarrow \forall x, y \in \mathbb{U} | x \neq y$:

$$\frac{\overbrace{|h \in \mathbb{H} : h(x) = h(y)|}^{\text{No. of hash functions that create collisions}}}{\underbrace{|\mathbb{H}|}_{\text{No. of hash functions}}} \leq c \cdot \frac{1}{m}, \quad c \in \mathbb{R}$$

Which means:

$$p\underbrace{(h(x) = h(y))}_{\text{Collision}} \leq c \cdot \frac{1}{m}$$

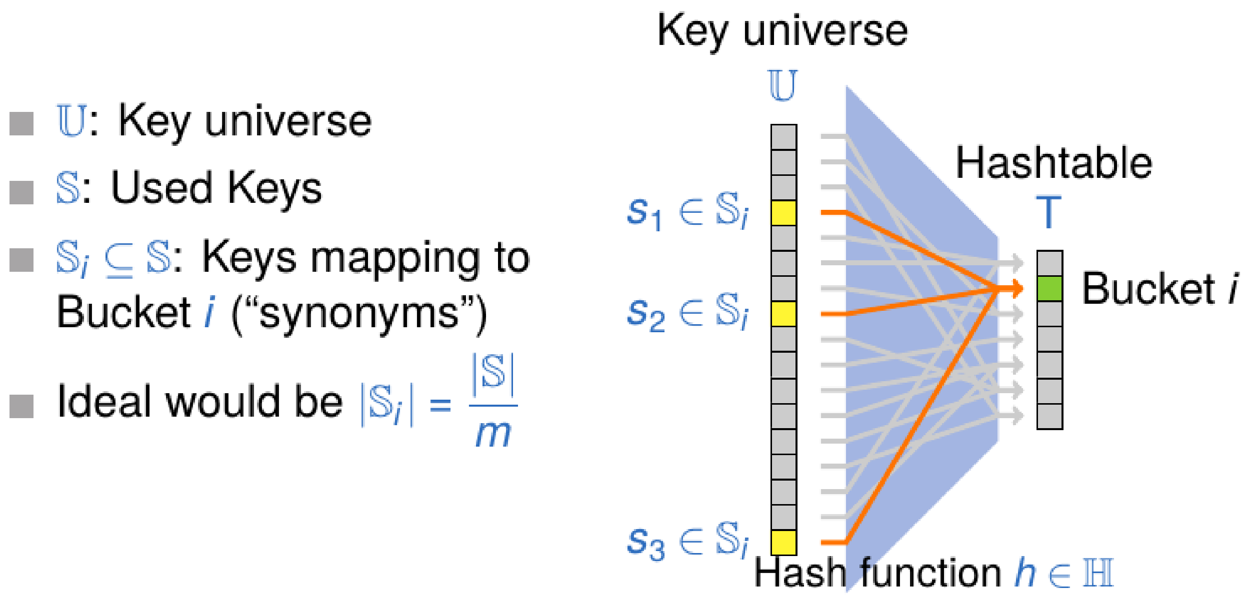


Abbildung 5: Schematic for universal hashing

Lookup time

- $\mathbb{H}$: c-universal class of hash functions
- $\mathbb{S}$: set of keys
- $h \in \mathbb{H}$: randomly selected hash functions
- $\mathbb{S}_i$:= the key x for which $h(x) = i$

Then, the average bucketsize is:

$$\mathbb{E}\{|\mathbb{S}_i|\} \leq 1 + c \cdot \frac{|\mathbb{S}|}{m} \tag{18}$$

Particularly:

$$m = \Omega(|\mathbb{S}|) \Rightarrow \mathbb{E}\{|\mathbb{S}_i|\} = \mathcal{O}(n) \tag{19}$$

Proof

Given:

- Pick two random keys $x, y \in \mathbb{S} | x \neq y$ and a random, c-universal hash function $h \in \mathbb{H}$
- Probability of a collision:
$$P(h(x) = h(y)) \leq \frac{c}{m}$$

To proof:

$$\mathbb{E}\{|\mathbb{S}_i|\} \leq 1 + c \cdot \frac{|\mathbb{S}|}{m}$$

Proof:

$$\mathbb{S}_i = \{x \in \mathbb{S} : h(x) = i\}$$

if $\mathbb{S}_i = \emptyset \Rightarrow |\mathbb{S}_i| = 0$; otherwise, let $x \in \mathbb{S}_i$ be any key:

$$I_y := \begin{cases} 1, & \text{if } h(y) = i \\ 0, & \text{else} \end{cases} \quad y \in \mathbb{S} \backslash \{x\}$$

$$\Rightarrow |\mathbb{S}_i| = 1 + \sum_{y \in \mathbb{S} \backslash x} I_y$$

$$\Rightarrow \mathbb{E}\{|\mathbb{S}_i|\} = \mathbb{E}\left\{1 + \sum_{y \in \mathbb{S} \backslash x} I_y\right\} = 1 + \sum_{y \in \mathbb{S} \backslash x} \underbrace{\mathbb{E}\{I_y\}}_{\leq c \cdot \frac{1}{m}}$$

$$\begin{aligned} \Rightarrow 1 + \sum_{y \in \mathbb{S} \backslash x} \mathbb{E}\{I_y\} \leq 1 + \sum_{y \in \mathbb{S} \backslash x} c \cdot \frac{1}{m} \\ = 1 + (|\mathbb{S}| - 1) \cdot c \cdot \frac{1}{m} \\ \leq 1 + c \cdot \frac{|\mathbb{S}|}{m} \end{aligned}$$

$$\mathbb{E}\{|\mathbb{S}_i|\} = 1 + \sum_{y \in \mathbb{S} \backslash x} \mathbb{E}\{I_y\} \leq 1 + c \cdot \frac{|\mathbb{S}|}{m} \quad \text{q.e.d.}$$

Examples for universal hashing

- p: big prime number, $p > m$, and $p \geq |\mathbb{U}|$
- $\mathbb{H}$: Set of all h for which:
$$h_{a,b}(x) = ((a \cdot x + b) \mod p) \mod m$$
where $1 \leq a < p, \quad 0 \leq b < p$
- This is $\approx$ 1-universal

4.3 Rehashing

- *Rehash*: New hash table with new random hash function
  $\rightarrow$ Expensive, but rarely done $\Rightarrow$ average cost is low

4.4 Linked lists for buckets

- Each bucket is a linked list.
- If a collision occurs the new keys are sorted into, or appended at the end of the list.
- Best case: Operations take $\mathcal{O}(1)$
- Worst case: $\mathcal{O}(n)$ e.g. for degenerated tables

4.5 Open Addressing

- For colliding keys we choose a new free entry.
- A *probe sequence* determines in which sequence the hash table is searched for a free bucket.
  - Entries are iteravly checked, until a free one is found where the element can be inserted.
  - If a lookup doesn't find the corresponding entry, probing has to be performed, until the element or a free entry is found.

Definitions

- $h(s)$: Hash function for key s
- $g(s,j)$: Probing function for key s with overflow positions $j \in \{0, ..., m-1\}$,
  e.g. $g(s,j) = j$
- The *probe sequence* is calculated by:

$$h(s,j) = (h(s) - g(s,j)) \mod m \in \{0, ..., m-1\} \tag{20}$$

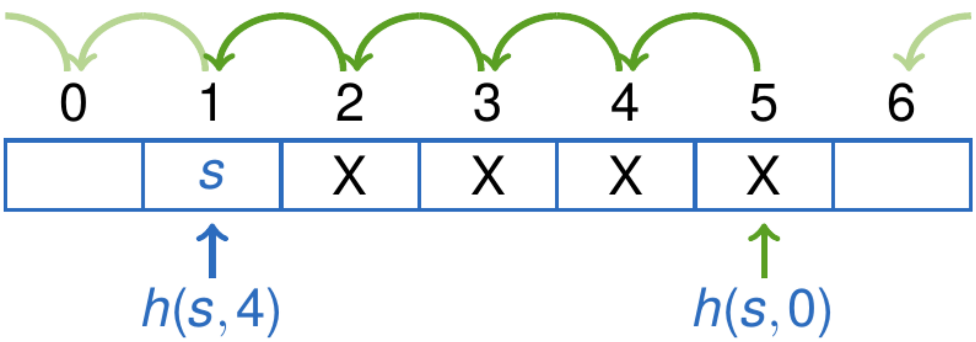


Abbildung 6: Linear sequence ($g(s,j) = j$)

Linear probing $g(s,j) = j$

- $g(s,j)$ clips from 0 to $m-1$.
- Can result in primary clustering

⇒ Hash collisions result in higher probability of hash collisions in close entries (hence, $\mathcal{O}(n)$ for lookup)

Squared probing

- Motivation: Avoid local clustering

$$g(s,j) := (-1)^j \lfloor \frac{j}{2} \rfloor^2 \tag{21}$$

- Resulting probe sequence:

$$h(s), h(s)+1, h(s)-1, h(s)+4, h(s)-4, ...$$

- If m is a prime number s.t. $m = 4 \cdot k + 3$, then the probe sequence is a permutation of the indices of the hash tabels
- Problem: Secondary clustering

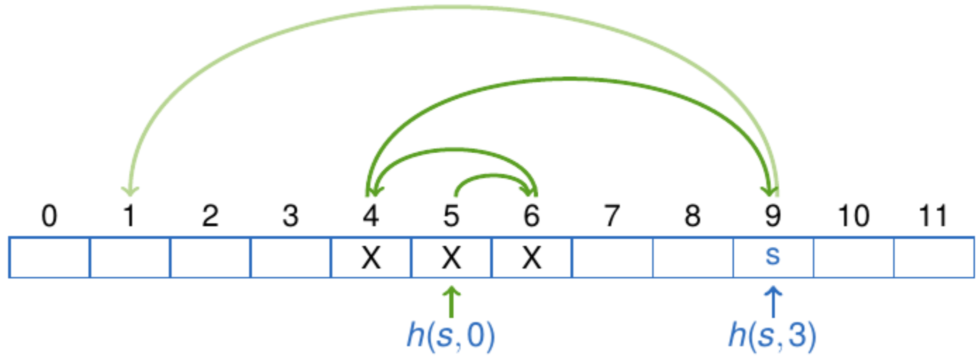


Abbildung 7: Squared probe sequence

Uniform probing

- So far: $g(s,j)$ independent of s
- *Uniform probing*: $g(s,j)$ also dependent on the key s
- Advantage: Prevents clustering, because different keys with the same hash value produce a different probe sequence
- Disadvantage: Hard to implement

Double hashing

- Use two independent hash functions $h_1(s), h_2(s)$

$$h(s,j) = (h_1(s) + j \cdot h_s(s)) \mod m \tag{22}$$

- Works well in practical use
- Approximation of uniform probing
- **Double hashing by** Brent
  - Test if $h(s_1, 1)$ is free
  - If yes, move s_1 from $h(s_1, 0)$ to $h(s_1, 1)$ and insert s_2 at $h(s_2, 0)$

Ordered hashing

- If a collission occurs for the keys s_0 and s_1, insert the smaller key and search a new position for the bigger according to the probe sequence.

⇒ Unsuccessful search can be aborted sooner

Robin-Hood Hashing

- If two keys s_1, s_2 collide, compare the length of the sequence j_1 and j_2.
- The key with the bigger search sequence is inserted at p_1, the other one gets reassigned according to the sequence.

Insert and Remove

- **Problem**:
  1. Key s_1 is inserted at p_1
  2. Key s_2 collides with s_2 at p_1 $\leftarrow$ gets inserted at p_2, due to probing order
  3. s_1 removed $\Rightarrow$ s_2 is virtually lost
- **Solution:**
  - Remove: Elements are marked as removed, but not deleted.
  - Insert: Elements marked as removed are overwritten.

5 Priority Queue

- Stores a set of elements
- Each element contains a key and a value.
- There is a total order (e.g. $\leq$) defined on the keys (heap).
- Operations
  - `insert(key, value)`:
    1. Append element at the end of the array
    2. Repair heap condition
  - `getMin()`: Return the first element or `None` if heap empty.
  - `deleteMin()`:
    1. Delete root of heap.
    2. Put last element at the root.
    3. Repair heap condition. (only up/down)
- Additional operations:
  - `changeKey(item, key)`:
    1. Change key value.
    2. Repair heap condition. (only up/down)
  - `remove(item)`:
    1. Replace element with the last element and shrink heap by one.
    2. Repair heap condition. (only up/down)

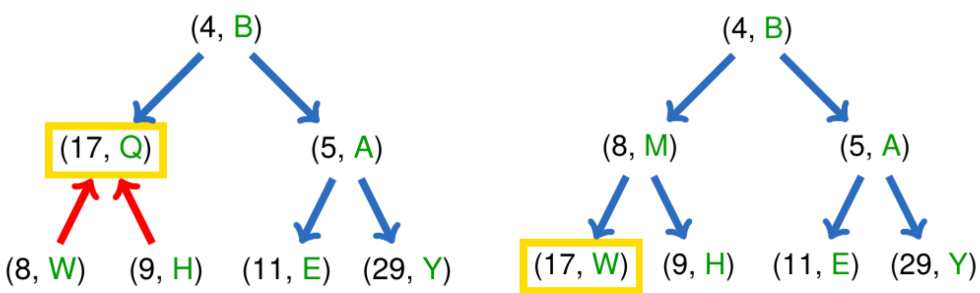


Abbildung 8: Sift up

- Multiple elements with the same key are allowed.
- **Each element has to store its' current position in the heap.**

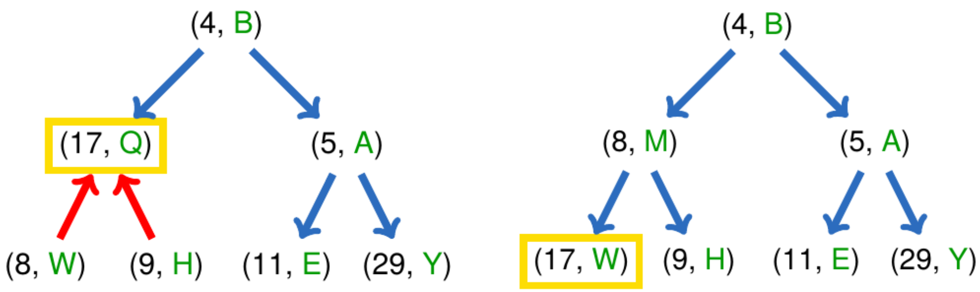


Abbildung 9: Sift down

6 Static and dynamic arrays

Static arrays have a fixed size (has to be known at compile time).

6.1 Dynamic arrays

Resizing an array:

1. Allocate array with new size
2. Copy entries from old array to new array

Naive implementation

- Resize array before each append to the exact needed size
- Runtime: $\mathcal{O}(n^2)$

Constantly generous allocation

- Allocate more space than needed.
- Amount of over-allocation C is constant.
- Runtime: still $\mathcal{O}(n^2)$

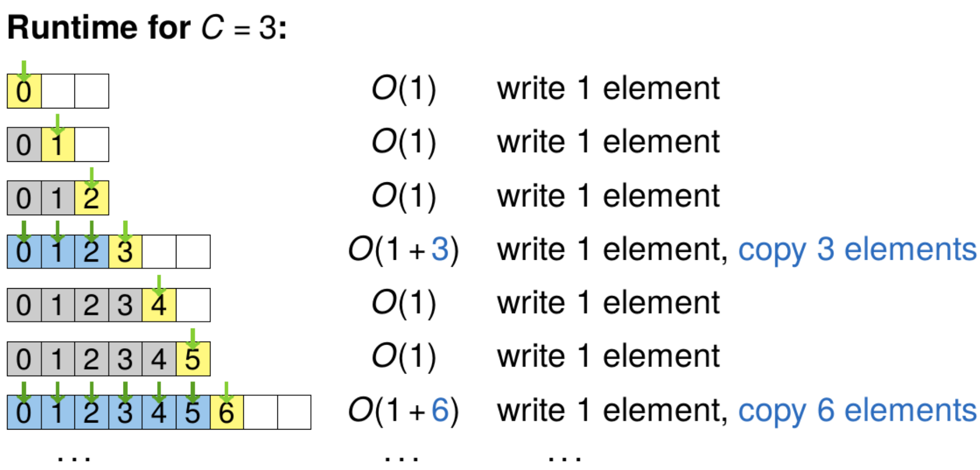


Abbildung 10: Runtime of constantly generous reallocation

- Most of the append operations now cost $\mathcal{O}(1)$, every C steps, cost of copying are added. $\Rightarrow$ We're getting faster

Variable overallocation

- Idea: Double size of the array for reallocation
- Runtime:
  - Now, all appends cost $\mathcal{O}(1)$
  - Every 2^i steps we have to add the cost $A \cdot 2^i$ ($i = 0, 1, 2, ..., k; k = \lfloor \log_2(n-1) \rfloor$)

$$\begin{aligned} T(n) &= n \cdot A + A \cdot \sum_{i=0}^{k} 2^i = n \cdot A + A(2^{k+1} - 1) \\ &\leq n \cdot A + A \cdot 2^{k+1} \\ &= n \cdot A + 2 \cdot A \cdot 2^k \\ &\leq n \cdot A + 2 \cdot A \cdot n \\ &= 3A \cdot n \\ &\in \mathcal{O}(n) \end{aligned}$$

- Further improvement:
  - Shrink array by half, if it is half-full.
  - Only shrink it to 75% to optimize appending afterwards.

6.2 Amortized analysis

- n instructions $O = \{O_1, ..., O_n\}$
- s_i: Size after operation i, $s_0 := 0$
- c_i: Capacity after operation i, $c_0 := 0$
- $T(O_i)$: Cost of operation i:

$$\text{Reallocation: } T(O_i) \leq A \cdot s_i$$
$$\text{Insert/Delete: } T(O_i) \leq A$$

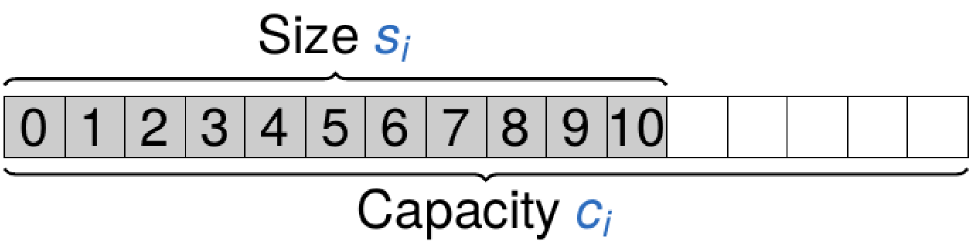


Abbildung 11: Static array with capacity c_i

- Implementation:
  - If $O_i =$ append and $s_{i-1} = c_{i-1}$:
    * Resize array to $c_i = \lfloor \frac{3}{2} s_i \rfloor$
    * $T(O_i) = A \cdot s_i$
  - If $O_i =$ remove and $s_{i-1} \leq \frac{1}{3} c_{i-1}$:
    * Resize array to $c_i = \lfloor \frac{3}{2} s_i \rfloor$
    * $T(O_i) = A \cdot s_i$
  - Amortized runtime:

$$\sum_{k=1}^{n} T(O_k) \leq 4A \cdot n$$

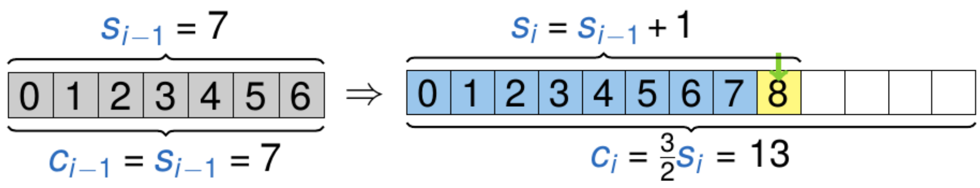


Abbildung 12: Append operation with reallocation

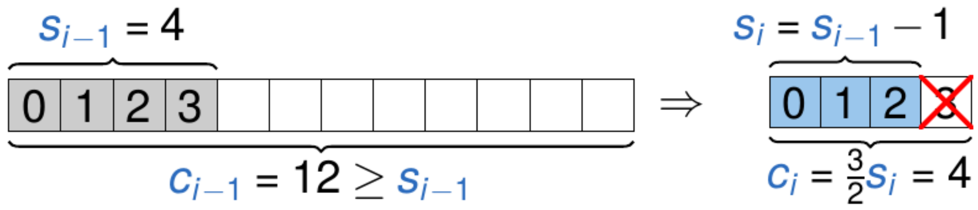


Abbildung 13: Remove operation with reallocation

7 Cache efficiency

- Even for the same number of operations, the runtime can differ substantially due to different memory access strategies.
  - Example: Adding up array entries in linear order vs. random order.
- Access times:
  - RAM→Cache: Slow ($\approx$ 100ns)
  - Cache→Register: Fast ($\approx$ 1ns)
- Cache organization:
  - The (L1-)cache can hold multiple memory blocks ($\approx$ 100kB)
  - Capacity is reached $\Rightarrow$ unused blocks are discarded. Different strategies:
    * Least Recently Used (LRU)
    * Least Frequently Used (LFU)
    * First in First Out (FIFO)
- Terminology:
  - Memory is divided in blocks of size B.
  - Cache has size M and can store ${}^{M}/_{B}$ blocks.
  - Data not in cache $\Rightarrow$ corresponding block is loaded from memory.
- Accessing the cache B times:
  - Best case: 1 block operation $\rightarrow$ good *locality*
  - Worst case: B block operations $\rightarrow$ bad *locality*
- Block loads on cache are called *cache misses*$\rightarrow$ *cache efficiency*
- Block operations on disk-cache are called *IOs* $\rightarrow$*IO efficiency*
- Example: Linear order
  - Sum up all elements in natural order:

$$\text{sum(a)} = a[1] + a[2] + ... + a[n]$$

  - Amount of block operations$=\lceil \frac{n}{B} \rceil$

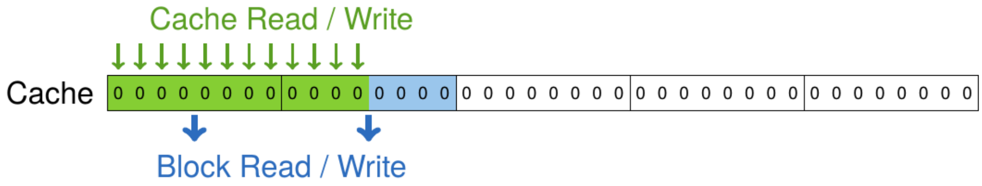


Abbildung 14: Good locality of sum operation

- Example: Random order
  - Sum up all elements in random order:

$$\text{sum(a)} = a[23] + a[42] + ... + a[3]$$

  - Amount of block operations:n in the worst case
  - Runtime factor difference: B

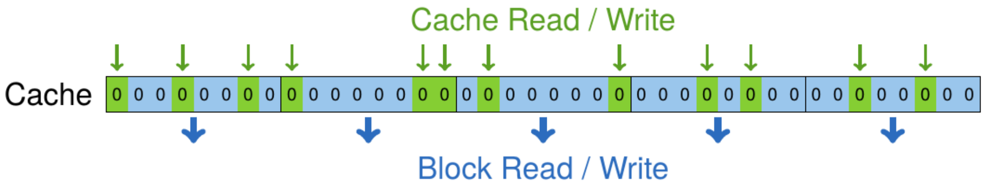


Abbildung 15: Bad locality of sum operation

  - Usually, the factor is substantially $< B$ (we might be lucky about the block position)

7.1 Quicksort

- Strategy: Divide and conquer
  - Task: Divide data into two parts where the left part contains all values $\leq$ those in right part
  - Chose one *pivot*-element
  - Both parts are sorted reucursively
- Approach:
  1. Pivot in (e.g.) first position, first rearrange list s.t. left part contains small, right part larger elements
     - s: Start-index of list
     - e: End-index of list
  2. Until $i > k$:
     - Increase i until it finds an element $> e_p$
     - Decrease k until it finds an element $< e_p$
     - If $i < k$: swap elements e_i and e_k
  3. Swap e_k with e_p
  4. Call quicksearck on $(s, k-1)$ and $(k+1, e)$
- Runtime:
  - Best case: $\mathcal{O}(n \log n)$
  - Worst case: $\mathcal{O}(n^2)$
  - Quicksort has quite good locality.

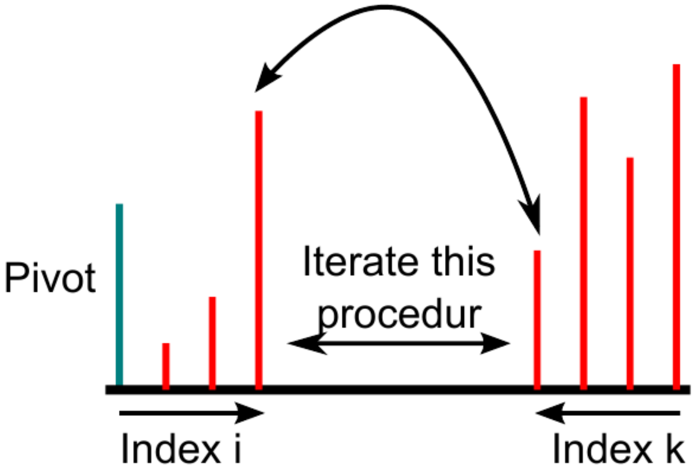


Abbildung 16: Quicksearch schematic

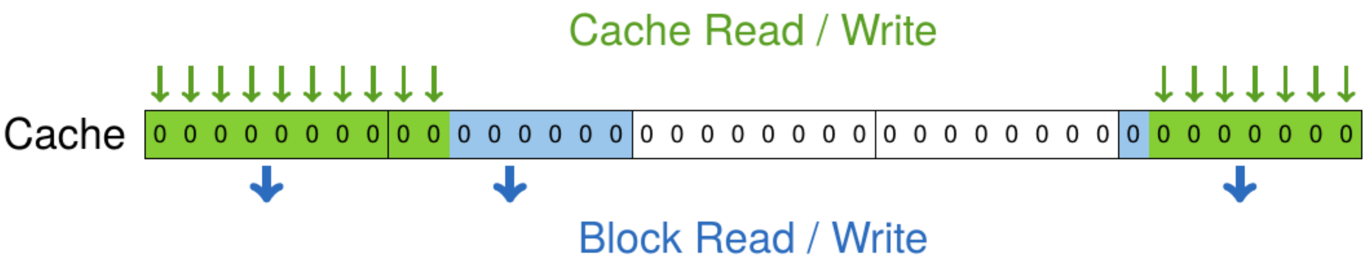


Abbildung 17: Locality of quicksort

  - Block operations: $IO(n) :=$ number of block operations for input size n

$$
\begin{aligned}
IO(n) &= \underbrace{A \cdot \frac{n}{B}}_{\text{splitting}} + \underbrace{2 \cdot IO(\frac{n}{2})}_{\text{recursive sort}} \\
&\leq 2A \cdot \frac{n}{B} + 4 \cdot IO(\frac{n}{4}) \\
&\leq 3A \cdot \frac{n}{B} + 8 \cdot IO(\frac{n}{4}) \\
&\leq \dots \\
&\leq kA \cdot \frac{n}{B} + 2^k \cdot IO\left(\frac{n}{2^k}\right) \\
&= \log_2\left(\frac{n}{B}\right) \cdot A \cdot \frac{n}{B} + \frac{n}{B} \cdot IO\left(B\right) \\
&\leq \log_2\left(\frac{n}{B}\right) \cdot A \cdot \frac{n}{B} + A \cdot \frac{n}{B} \\
&\in \mathcal{O}\left(\frac{n}{B} \cdot \log_2\left(\frac{n}{B}\right)\right)
\end{aligned}
$$

7.2 Divide and Conquer

Concept:

- *Divide* the problem into smaller subproblems
- *Conquer* subproblems through *recursive* solving. If subproblems are small enough, solve them *directly*.
- *Connect* all solutions of the subproblems to a solution of the full problem.

7.2.1 Features

- Requirements:

- Solution of trivial problems needs to be known.
- Dividing must be possible.
- Sub-Solutions have to be recombinable.

- Runtime:
  - If trivial solution $\in \mathcal{O}(1)$
  - And separation/combination of subproblems $\in \mathcal{O}(n)$
  - And the number of subproblems is finite

  $\Rightarrow$ **Runtime** $\in \mathcal{O}(n \cdot \log n)$

- Suitable for parallel processing, since subproblems are *independent* of each other

7.2.2 Implementation

- Smaller subproblems are elegant and simple, or it would be better to solve bigger subproblems directly.
- Recursion depth shouldn't get too big (stack/memory overhead).

7.2.3 Example: Maximum subtotal

1. Split sequence in the middle
2. Solve both halves
3. Combine both sub-solutions into a total solution
4. For the case of overlap split, we have to calculate rmax and lmax as well.
5. Solution: $\max(A, B, C)$

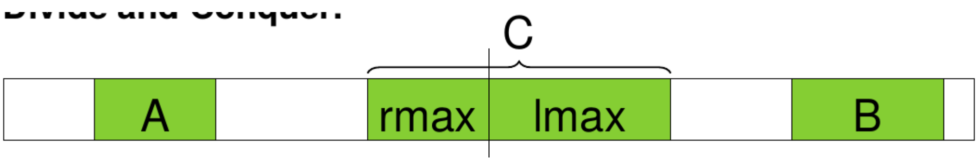


Abbildung 18: Approach to maximum subtotal

8 Recursion Equations

- Recursion equation:

$$T(n) = \begin{cases} f_0(n) & n = n_0 \\ a \cdot T\left(\frac{n}{b}\right) + f(n) & n > n_0 \end{cases} \tag{23}$$

  - $n = n_0$: Trivial case (usually $\in \mathcal{O}(1)$)
  - $a \cdot T\left(\frac{n}{b}\right)$: Solving of a subproblems with reduced input size ${}^{n}/_{b}$
  - $f(n)$: slicing and splicing of subsolution
  - Normally: $a > 1$ and $b > 1$

8.1 Substitution method

- Guess the solution and prove it with induction
- Example:

$$T(n) = \begin{cases} 1 & n = 1 \\ 2 \cdot T\left(\frac{n}{2}\right) + n & n > 1 \end{cases}$$

- Assumption: $T(n) = n + n \log_2 n$
- Proof: Induction (base:$n_0 = 1$, induction step: $n \to 2n$)
- Alternative Assumption: $T(n) \in \mathcal{O}(n \log n)$
- Solution: Find $c > 0$ with $T(n) \leq c \cdot n \log_2 n$ (again: induction)

8.2 Recursion tree method

- Can be used to make assumptions about the runtime
- Example:

$$T(n) = 3 \cdot T\left(\frac{n}{4}\right) + \Theta(n^2) \leq 3 \cdot T\left(\frac{n}{4}\right) + c \cdot n^2$$

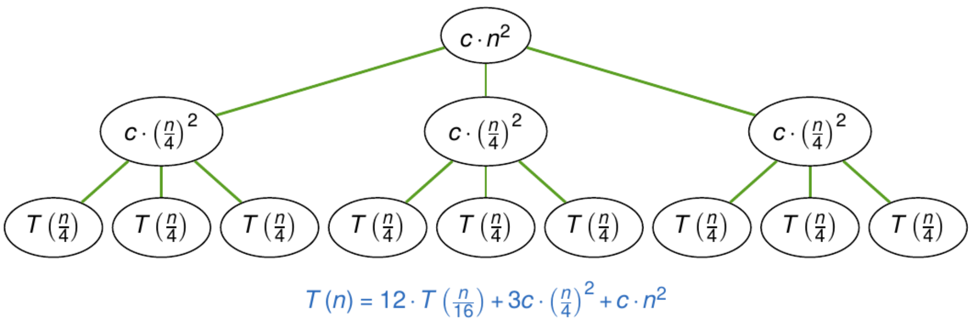


Abbildung 19: Recursion tree of example

- Costs of connecting the partial solutions (excludes the last layer):
  - Size of partial problems on level i: $s_i(n) = \left(\frac{1}{4}^i \cdot n\right)$
  - Costs of partial problem on level i:

$$T_i(n) = c \cdot \left(\left(\frac{1}{4}\right)^i \cdot n\right)^2$$

  - Number of partial problems on level i: $n_i = 3^i$

$\Rightarrow$ Costs on level i:

$$T_i(n) = 3^i \cdot c \cdot \left(\left(\frac{1}{4}\right)^i \cdot n\right)^2 = \left(\frac{3}{16}\right)^i \cdot c \cdot n^2$$

- Costs of solving the last layer:
  - Size of partial problems on the last level: $s_{i+1}(n) = 1$
  - Costs of partial problem on the last level: $T_{i+1}(n) = d$
  - With this the depth of the tree is:

$$\left(\frac{1}{4}\right)^i \cdot n = 1 \quad \Rightarrow n = 4^i \quad \Rightarrow i = \log_4 n$$

– Number of partial problems on the last level:

$$n_{i+1} = 3^{\log_4 n} = n^{\log_4 3}$$

⇒ Costs on the last level:

$$T_{i+1}(n) = d \cdot n^{\log_4 3}$$

- Total cost:

$$T(n) = \underbrace{\sum_{i=0}^{\log_4(n)-1} \left(\frac{3}{16}\right)^i}_{\text{geometric series, constant}} \cdot n^2 + \underbrace{d \cdot n^{\log_4 3}}_{\log_4 3<1 \Rightarrow \text{slow growth}} \in \mathcal{O}(n^2)$$

8.3 Master theorem

- Appoach to solve for a recursion equation of the form:

$$\boxed{T(n) = a \cdot T\left(\frac{n}{b}\right) + f(n), \quad a \leq 1, b < 1} \tag{24}$$

- $T(n)$ is the runtime of an algorithm
  - ... which divides a problem of size n in a partial problems.
  - ... which solves each partial problem recursively with a runtime of $T\left(\frac{n}{b}\right)$
  - ... which takes $f(n)$ steps to merge all partial solutions
- Three dominations possible:
  - Runtime of connecting the solution dominates
  - Runtime of solving the problems dominates
  - Both have equal influence

8.3.1 Simple form

- Special case with runtime of connecting the solutions: $f(n) \in \mathcal{O}(n)$
- **Runtime:**

$$T(n) = \begin{cases} \Theta\overbrace{(n^{\log_b a})}^{\text{No. of leaves}} & \text{if } a > b \quad \text{(Branching factor dominates)} \\ \Theta(n^{\log_b a}) & \text{if } a = b \quad \text{(Balanced case)} \\ \Theta(n) & \text{if } a < b \quad \text{(Shrinking factor dominates)} \end{cases}$$

8.3.2 General form

$$T(n) = a \cdot T\left(\frac{n}{b}\right) + f(n) \quad a \leq 1, b > 1 \tag{25}$$

- Case 1: $T(n) \in \Theta(n^{\log_b a})$ if $f(n) \in \mathcal{O}(n^{\log_b a - \varepsilon}), \varepsilon > 0$
  solving the partial problems dominates (last layer, leaves)
- Case 2: $T(n) \in \Theta(n^{\log_b a} \log n)$ if $f(n) \in \Theta(n^{\log_b a})$
  each layer has equal costs, $\log n$ layers
- Case 3: $T(n) \in \Theta(f(n))$ if $f(n) \in \Omega(n^{\log_b a + \varepsilon}), \varepsilon > 0$
  Merging the partial solutions dominates.
  Important: Regularity condition:

$$a \cdot f\left(\frac{n}{b}\right) \leq c \cdot f(n), \quad 0 \leq c \leq 1, n > n_0 \tag{26}$$

Case 1 - Example: $T(n) \in \Theta(n^{\log_b a})$ if $f(n) \in O(n^{\log_b a - \varepsilon}),\ \varepsilon > 0$

Solving the partial problems dominates (last layer, leaves)

- $T(n) = 8 \cdot T(\frac{n}{2}) + 1000 \cdot n^2$

$$a = 8,\ b = 2,\ f(n) = 1000 \cdot n^2,\ \underbrace{\log_b a = \log_2 8 = 3}_{n^3 \text{ leaves}}$$

$f(n) \in \mathcal{O}(n^{3-\varepsilon}) \Rightarrow T(n) \in \Theta(n^3)$

- $T(n) = 9 \cdot T(\frac{n}{3}) + 17 \cdot n$

$$a = 9,\ b = 3,\ f(n) = 17 \cdot n,\ \underbrace{\log_b a = \log_3 9 = 2}_{n^2 \text{ leaves}}$$

$f(n) \in \mathcal{O}(n^{2-\varepsilon}) \Rightarrow T(n) \in \Theta(n^2)$

Case 2: $T(n) \in \Theta(n^{\log_b a} \log n)$ if $f(n) \in \Theta(n^{\log_b a})$

Each layer has equal costs, $\log n$ layers

- $T(n) = 2 \cdot T(\frac{n}{2}) + 10 \cdot n$

$$a = 2,\ b = 2,\ f(n) = 10 \cdot n,\ \underbrace{\log_b a = \log_2 2 = 1}_{n^1 \text{ leaves}}$$

$f(n) \in \Theta(n^{\log_2 2}) \Rightarrow T(n) \in \Theta(n \log n)$

- $T(n) = T(\frac{2n}{3}) + 1$

$$a = 1,\ b = \frac{2}{3},\ f(n) = 1,\ \underbrace{\log_b a = \log_{3/2} 1 = 0}_{n^0 \text{ leaves} = 1 \text{ leaf}}$$

$f(n) \in \Theta(n^{\log_{3/2} 1}) \Rightarrow T(n) \in \Theta(n^0 \log n) = \Theta(\log n)$

Case 3: $T(n) \in \Theta(f(n))$ if $f(n) \in \Omega(n^{\log_b a + \varepsilon}),\ \varepsilon > 0$

Connecting all partial solutions dominates (first layer, root)

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

$$a = 2,\ b = 2,\ f(n) = n^2,\ \underbrace{\log_b a = \log_2 2 = 1}_{n^1 \text{ leaves}}$$

$f(n) \in \Omega(n^{1+\varepsilon})$

Check if regularity condition also holds:

$$2 \cdot \left(\frac{n}{2}\right)^2 \le c \cdot n^2 \quad \Rightarrow \frac{1}{2} \cdot n^2 \le c \cdot n^2 \quad \Rightarrow c \ge \frac{1}{2}$$

$\Rightarrow T(n) \in \Theta(n^2)$

- The master theorem is not always applicable. Example

$$T(n) = 2 \cdot T\left(\frac{n}{2}\right) + n \log n$$
$$a = 2, b = 2, f(n) = n \log n, \underbrace{\log_b a = \log_2 2 = 1}_{n^1 \text{ leaves}}$$

  - $f(n) \notin \mathcal{O}(n^{1-\varepsilon})$
  - $f(n) \notin \Theta(n^1)$
  - $f(n) \notin \Omega(n^{1+\varepsilon})$
  - $n \log n$ is *asymptotically* larger than n, but not *polynomially* larger.

9 Sorted collections

- Set of keys, mapped to values
- Elements are topologically sorted $\leq$ by their key
- The following operations are needed:
  - `insert(key, value)`
  - `remove(key)`
  - `lookup(key)`: Find the element with the given key, or if not present, return the next bigger one
  - `next()`: Returns the element with the next bigger key
  - `previous()`: Returns the element with the next smaller key

9.1 Static array

- Sorted, static array
- `lookup` time: $\mathcal{O}(logn)$
  with *binary search*
- `next/previous` time: $\Theta(1)$
- `insert/remove` time: up to $\Theta(n)$
  We have to copy up to n elements.

9.2 Hash map

- `lookup` time: $\Theta(1)$
  if element exists, otherwise result=`None`
- `next/previous` time: up to $\Theta(n)$
  Order of the elements is independent of the order of the keys.
- `insert/remove` time: $\Theta(1)$
  If m is big enough and the hash function is good

9.3 Doubly linked list

- `lookup` time: $\Theta(n)$
  Iterate over the elements in the list.
- `next/previous` time: $\Theta(1)$
  Elements are linked like a chain
- `insert/remove` time: $\Theta(1)$

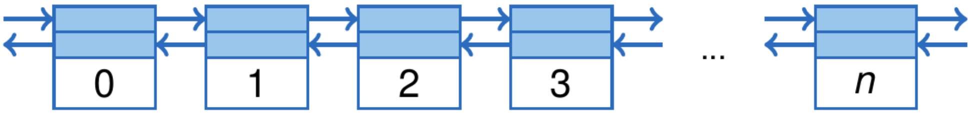


Abbildung 20: Doubly linked list

10 Linked lists

- Dynamic datastructure
- Amount of elements variable
- Data elements can be simple types upto complex datastructures
- Elements are linked through references/pointers to the predecessor/successor
- Singly or doubly linked possible

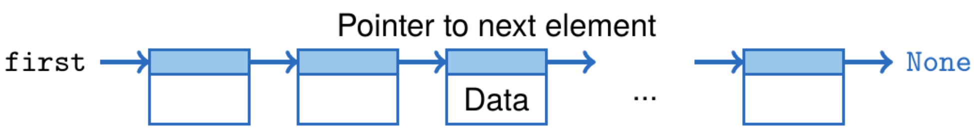


Abbildung 21: Singly Linked list

- Comparison to an array:
  - Needs extra space for storing the pointers
  - No need for copying elements on `insert` or `remove`
  - The number of elements can be modified without big computational overhead
  - No direct access of elements (Necessary to iterate over the list)
  - In general: worse locality than arrays

10.1 List with head/last element pointer

- Head element has pointer to first list element
- Pointer to last element
- May also hold additional information (e.g.: Number of elements)

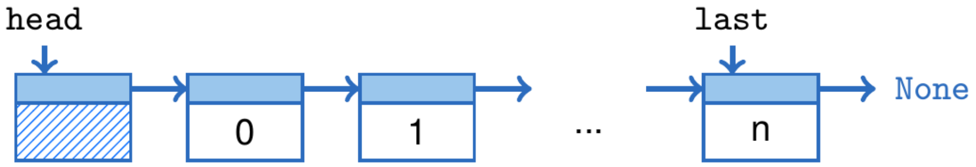


Abbildung 22: Linked list with header

10.2 Doubly linked list

- Pointer to successor element (last element successor: `None`)
- Pointer to predecessor element (first element predecessor: `None`)
- Iterate forward and backward

10.3 Usage

- Creating linked lists:
  - `first = Node(7)`
  - `first.nextNode = Node(3)`
- Inserting a node after node `cur`:
  1. `ins = Node(n)`
  2. `ins.nextNode = cur.nextNode`
  3. `cur.nextNode = ins`
- Removing a node `cur`:
  1. Find predecessor of `cur` (`while (pre.nextNode != cur) pre = pre.nextNode;`)
     - Runtime of $\mathcal{O}(n)$
     - **Doesn't work on first node!**
  2. `pre.nextNode = cur.nextNode`
  3. `delete cur`, or `cur=None` (automatic if you are a lazy hack who uses garbage collection!)
- Removing the first node:
  1. `first = first.nextNode`
  2. `delete cur`, if no garbage collection
- Using a `head` node:
  - Deleting the first node is no special case
  - Have to consider first node at other operations:
    * Iterating all nodes
    * Counting all nodes
    * ...
- `Head` and `last` node
  - Append elements to the end of the list: $\mathcal{O}(1)$
  - Pointer to `last` needs to be updated after all operations
- `get(key)`: Iterate the entries until at position ($\mathcal{O}(n)$)
- `find(value)`: Iterate the entries until value found ($\mathcal{O}(n)$)

```python
def append(self, value):
    last.nextNode = Node(value)
    last = last.NextNode
    itemCount += 1
```

Abbildung 23: Algorithm for appending to last element

10.4 Runtime

- Singly linked list:
  - `next`: $\mathcal{O}1$
  - `previous`: up to $\Theta(n)$
  - `insert`: $\mathcal{O}1$
  - `remove`: up to $\Theta(n)$
  - `lookup`: up to $\Theta(n)$
- Doubly linked list:
  - Useful to have a `head` node.
  - Only need one `head` node if we connect the list cyclic (Figure 20).

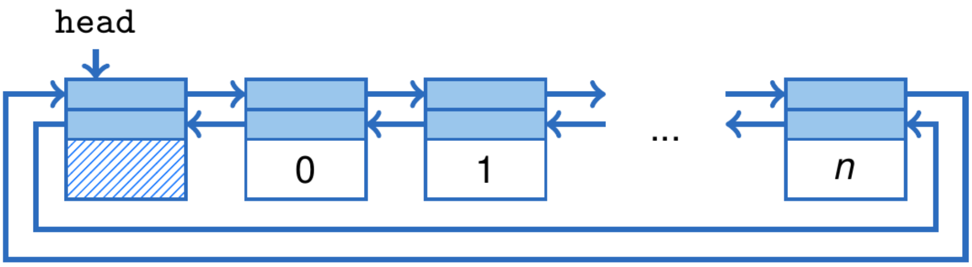
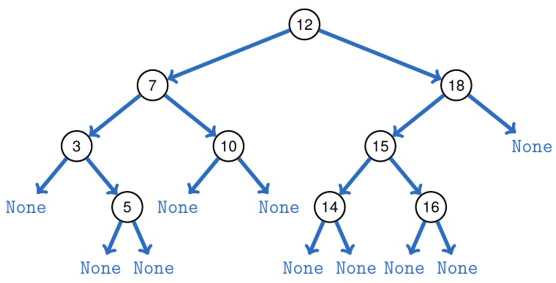


Abbildung 24: Cyclic doubly linked list

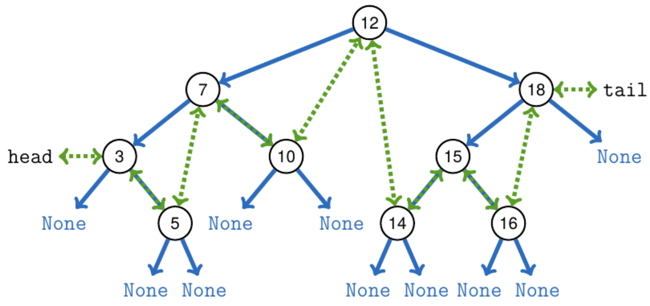
  - `next/previous`: $\mathcal{O}(1)$
  - `insert/remove`: $\mathcal{O}(1)$
  - `lookup`: up to $\Theta(n)$ (even if elements are sorted)

11 Binary search tree

- **Principle**:
  - Define a total order (e.g. $\leq, \geq$)
  - All nodes of the left subtree have *smaller keys* than the current node.
  - All nodes of the right subtree have *bigger keys* than the current node.
  - The next highest element of the current node is the leftmost element from the left subtree.
  - The next lowest element of the current node is the rightmost element from the right subtree.
- **Runtime:**
  - `next/previous`: $\mathcal{O}(1)$
  - `insert/remove`: $\mathcal{O}(\log n)$
  - `lookup`: up to $\Theta(n)$
- **Implementation**:



(a) Binary search tree with references



(b) Binary search tree with doubly linked list

  - We link all nodes through pointers/references
  - Each node has a pointer/reference to its' children (`leftChild/rightChild`)

- Implementation on steroids (with links):
  - Sorted doubly linked list of all elements

  $\Rightarrow$ Efficient implementation of `next/previous`

- `Lookup(key)`: "Search element. If not found, return element with next (bigger) key"
  - Start at root
  - Go down left/right recursively until found, or `None`
  - If `None`: return next biggest element

- `Insert(key, value)`:
  - Search for key in tree
  - If found: $\rightarrow$ replace value with the new one
  - Else: insert new node at the corresponding `None` entry

- `Remove(key)`: (quite tricky)
  1. Node has no `children`:
     - Find `parent` of the node.
     - Set the left/right `child` of the `parent` node to `None`.
  2. Node has one `child`:
     - Find the `child` of the node.
     - Find the `parent` of the node.
     - set the left/right `child` of the `parent` node to the node's `child`.
  3. Node has two children
     - Find the nodes' `successor`.
     - Replace the node with its' `successor`
     - Delete the `successor`

- Runtime of `insert()` and `lookup()`:
  - Up to $\Theta(d)$ ($d :=$ depth of the tree)
  - Best case $d = \log n$: $\Theta(\log n)$
  - Worst case $d = n$: $\Theta(n)$ (tree degenerated)
  - For consistent runtime of $\Theta(\log n)$, we have to *rebalance* the tree.

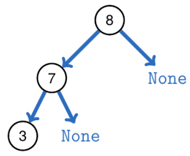


Abbildung 26: Degenerated search tree

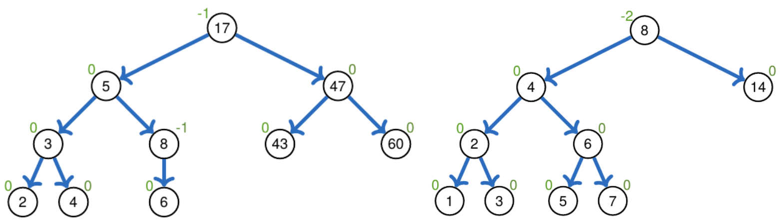
12 Balanced search trees

- How do we fix degenerated trees? → rebalancing!
- Rebalancing possibilities:
  - AVL-Tree:
    * Binary tree with 2 children per node
    * Balancing via "rotation"
  - (a,b)-Tree, or B-Tree:
    * Nodes have between a and b children
    * Balancing through *splitting* and *merging* nodes.
    * Used in data bases and file systems
  - Red-Black-Tree:
    * Binary tree with black and red nodes
    * Balancing through *rotation* and *recoloring*
    * Can be interpreted as (2,4)-tree

12.1 AVL-Tree

- **A**delson-**V**elskii, **L**andis (1963)
- Search tree with modified `insert()` and `remove` operations, while satisfying a *depth condition*.
- Prevents degeneration
- **Depth condition:** Highest possible height difference of left and right subtree = 1

⇒ Depth of tree is always $\mathcal{O}(\log n)$



(a) AVL tree (b) Not an AVL tree

- **Rotation:**

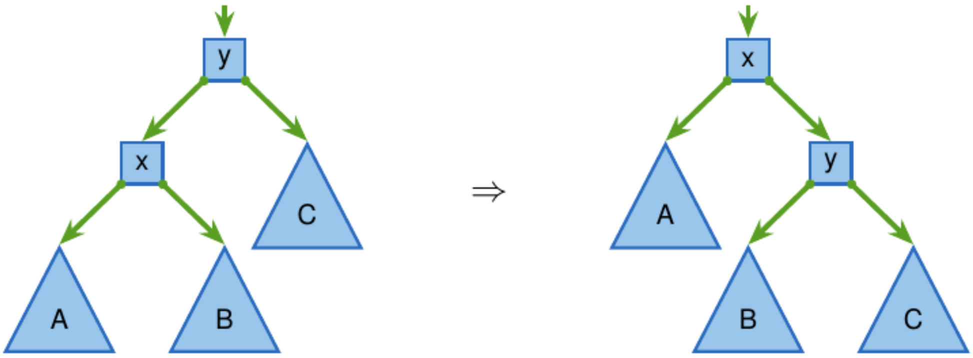


Abbildung 28: Rotation principle

  - `Parent-Child` relations are swapped for nodes which violate the depth-condition.
  - Attention: in this example, x's smaller subtree becomes y's larger subtree!

- If a height difference of ± 2 occurs after an `insert/remove`, the tree is rebalanced.
- **Disadvantages:**
  - Update cost is no longer an amortized $\mathcal{O}(1)$!
  - More memory consumption for depth values

⇒ Better option: (a,b)-Trees, a.k.a. b-trees (b for "balanced")

12.2 (a,b)-tree

- **Principle**:
  - Save a varying number of elements per node
  - All leaves have the same depth
  - Each inner node has $\geq a$ and $\leq b$ nodes (exeption: root node)
  - Subtrees are located "between" the elements.
  - $a \geq 2, b \geq 2a - 1$

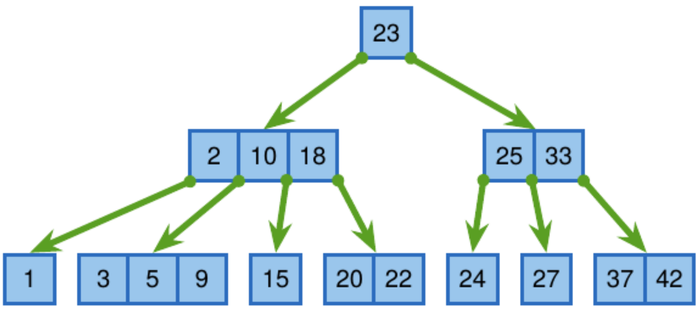


Abbildung 29: Example of an (2,4)-tree

- `lookup`: Same as in binary search tree
- `insert`:
  - Search the position to insert (always a leaf).
  - Insert node
  - Check if maximum number of nodes are exceeded.

  - If yes: Split the node!
  - ⇒ Two new nodes with $\lceil \frac{b-1}{2} \rceil$ and $\lfloor \frac{b-1}{2} \rfloor$ elements.
  - Checking the maximum number of nodes cascades up.
  - If we have to split the root node, we create a new one afterwards.

Abbildung 30: Splitting a node

- `remove`:
  - Search the element ($\mathcal{O}(\log n)$)
  - Case 1: Element is contained by a leaf ⇒ remove it!
  - Case 2: Contained by an inner node
    * Search the **successor** in the **right** subtree. (leftmost element of rightmost subtree, always contained by a leaf)
    * Replace the element with its' successor and delete the successor from the leaf
  - **Attention:** If size of leaf $< a$!
  - ⇒ **Rebalance** the tree:
    * Case 1: If the left or right neighbour node has leafs to spare, **get that one**

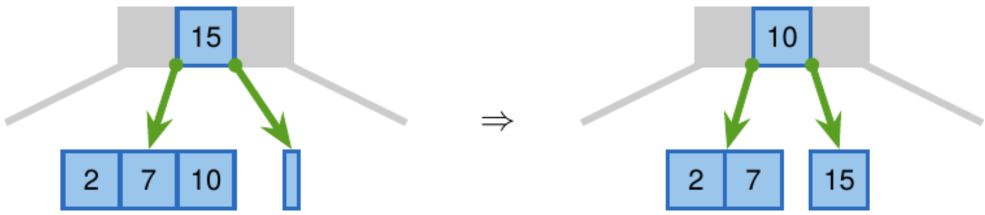


Abbildung 31: borrowing an element

    * Case 2: **Combine** the node with its' neighbour
      Check if we have to cascade upwards!

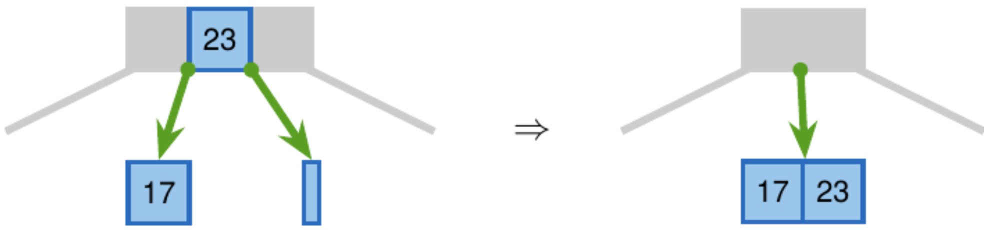


Abbildung 32: Combining with neighbour

    * If the root has only one child left, that child can become root.

Runtime of `lookup`, `insert` and `remove`

- All operations: $\mathcal{O}(d), d :=$depth of the tree
- Each node (except root) has more than a children
  $\Rightarrow n \geq a^{d-1}$ and $d \leq 1 + \log_a n \in \mathcal{O}(\log_a n)$
- `lookup`: $\in \Theta(d)$
- `insert/remove`: often only in $\mathcal{O}(1)$
- Only in worst case, we have to split or combine all nodes cascading up to root

12.2.1 Analysis of $b \geq 2a$

- nodes with **2, or 4 children** are expensive for delete and add respectively (borrowing, or splitting, possibly cascading up).

$\Rightarrow$ **3 children** are harmless:

- Φ_i:= Potential of the tree after the i-th operation.
- = the amount of harmless nodes (size 3)
- After expensive operation the tree is in a stable state.
- Takes some time until the next expensive operation occurs.
- Same principle of dynamic arrays: **Overallocate** clever, to get an amortized runtime of $\mathcal{O}(1)$