

8. Sorting II

Heapsort, Quicksort, Mergesort

8.1 Heapsort

[Ottman/Widmayer, Kap. 2.3, Cormen et al, Kap. 6]

209

210

Heapsort

Inspiration from selectsort: fast insertion

Inspiration from insertion sort: fast determination of position

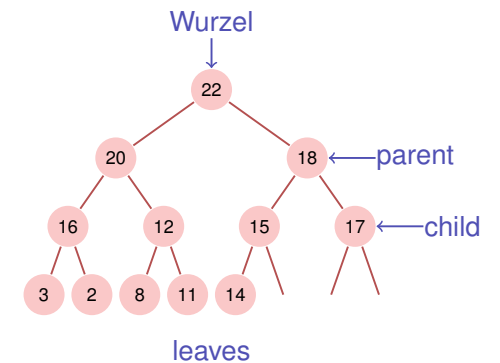
❓ Can we have the best of two worlds?

⚠️ Yes, but it requires some more thinking...

[Max-]Heap⁶

Binary tree with the following properties

- 1 complete up to the lowest level
- 2 Gaps (if any) of the tree in the last level to the right
- 3 **Heap-Condition:**
Max-(Min-)Heap: key of a child smaller (greater) than that of the parent node



211

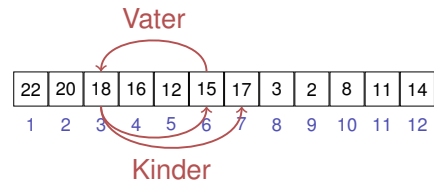
⁶Heap(data structure), not: as in "heap and stack" (memory allocation)

212

Heap and Array

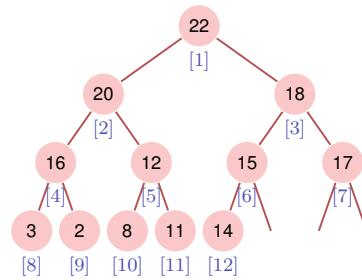
Tree \rightarrow Array:

- $\text{children}(i) = \{2i, 2i + 1\}$
- $\text{parent}(i) = \lfloor i/2 \rfloor$



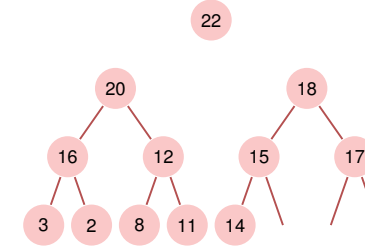
Depends on the starting index⁷

⁷For array that start at 0: $\{2i, 2i + 1\} \rightarrow \{2i + 1, 2i + 2\}$, $\lfloor i/2 \rfloor \rightarrow \lfloor (i - 1)/2 \rfloor$



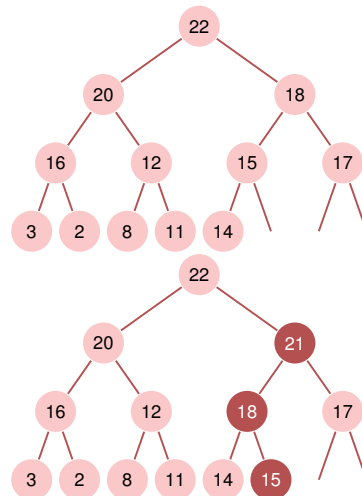
Recursive heap structure

A heap consists of two heaps:



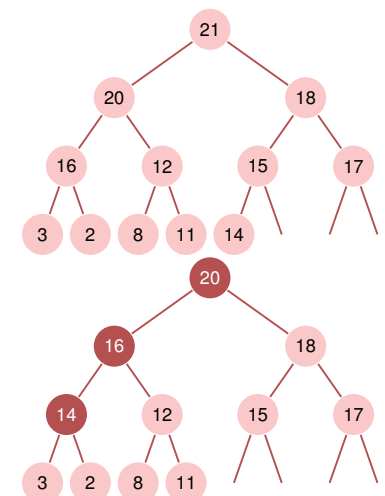
Insert

- Insert new element at the first free position. Potentially violates the heap property.
- Reestablish heap property: climb successively
- Worst case number of operations: $\mathcal{O}(\log n)$



Remove the maximum

- Replace the maximum by the lower right element
- Reestablish heap property: sink successively (in the direction of the greater child)
- Worst case number of operations: $\mathcal{O}(\log n)$



Algorithm Sink(A, i, m)

Input : Array A with heap structure for the children of i . Last element m .

Output : Array A with heap structure for i with last element m .

```

while  $2i \leq m$  do
   $j \leftarrow 2i$ ; //  $j$  left child
  if  $j < m$  and  $A[j] < A[j + 1]$  then
     $j \leftarrow j + 1$ ; //  $j$  right child with greater key
  if  $A[i] < A[j]$  then
    swap( $A[i], A[j]$ )
     $i \leftarrow j$ ; // keep sinking
  else
     $i \leftarrow m$ ; // sinking finished
  
```

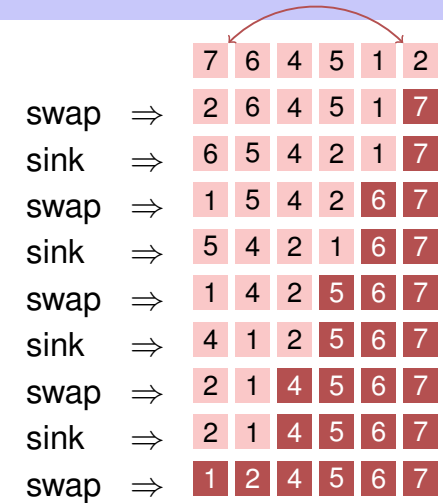
217

Sort heap

$A[1, \dots, n]$ is a Heap.

While $n > 1$

- swap($A[1], A[n]$)
- Sink($A, 1, n - 1$);
- $n \leftarrow n - 1$



218

Heap creation

Observation: Every leaf of a heap is trivially a correct heap.

Consequence: Induction from below!

219

Algorithm HeapSort(A, n)

Input : Array A with length n .

Output : A sorted.

```

for  $i \leftarrow n/2$  downto 1 do
  Sink( $A, i, n$ );
// Now  $A$  is a heap.
for  $i \leftarrow n$  downto 2 do
  swap( $A[1], A[i]$ )
  Sink( $A, 1, i - 1$ )
// Now  $A$  is sorted.
  
```

220

Analysis: sorting a heap

Sink traverses at most $\log n$ nodes. For each node 2 key comparisons. \Rightarrow sorting a heap costs in the worst case $2 \log n$ comparisons.

Number of memory movements of sorting a heap also $\mathcal{O}(n \log n)$.

8.2 Mergesort

[Ottman/Widmayer, Kap. 2.4, Cormen et al, Kap. 2.3],

Analysis: creating a heap

Calls to sink: $n/2$. Thus number of comparisons and movements:
 $v(n) \in \mathcal{O}(n \log n)$.

But mean length of sinking paths is much smaller:

$$v(n) = \sum_{h=0}^{\lfloor \log n \rfloor} \left\lceil \frac{n}{2^{h+1}} \right\rceil \cdot c \cdot h \in \mathcal{O}\left(n \sum_{h=0}^{\lfloor \log n \rfloor} \frac{h}{2^h}\right)$$

$s(x) := \sum_{k=0}^{\infty} kx^k = \frac{x}{(1-x)^2}$ ($0 < x < 1$). With $s(\frac{1}{2}) = 2$:

$$v(n) \in \mathcal{O}(n).$$

221

222

Intermediate result

Heapsort: $\mathcal{O}(n \log n)$ Comparisons and movements.

? Disadvantages of heapsort?

- ! Missing locality: heapsort jumps around in the sorted array (negative cache effect).
- ! Two comparisons before each required memory movement.

223

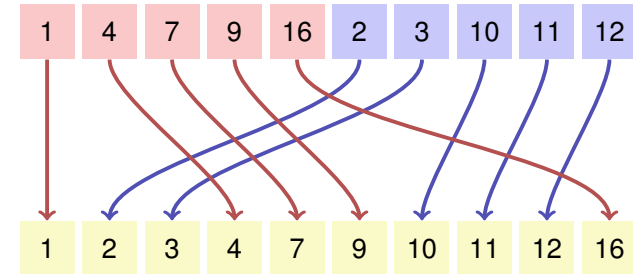
224

Mergesort

Divide and Conquer!

- Assumption: two halves of the array A are already sorted.
- Minimum of A can be evaluated with two comparisons.
- Iteratively: sort the presorted array A in $\mathcal{O}(n)$.

Merge



225

226

Algorithm Merge(A, l, m, r)

Input : Array A with length n , indexes $1 \leq l \leq m \leq r \leq n$. $A[l, \dots, m]$, $A[m + 1, \dots, r]$ sorted

Output : $A[l, \dots, r]$ sortiert

```
1  $B \leftarrow$  new Array( $r - l + 1$ )
2  $i \leftarrow l$ ;  $j \leftarrow m + 1$ ;  $k \leftarrow 1$ 
3 while  $i \leq m$  and  $j \leq r$  do
4   if  $A[i] \leq A[j]$  then  $B[k] \leftarrow A[i]$ ;  $i \leftarrow i + 1$ 
5   else  $B[k] \leftarrow A[j]$ ;  $j \leftarrow j + 1$ 
6    $k \leftarrow k + 1$ ;
7 while  $i \leq m$  do  $B[k] \leftarrow A[i]$ ;  $i \leftarrow i + 1$ ;  $k \leftarrow k + 1$ 
8 while  $j \leq r$  do  $B[k] \leftarrow A[j]$ ;  $j \leftarrow j + 1$ ;  $k \leftarrow k + 1$ 
9 for  $k \leftarrow l$  to  $r$  do  $A[k] \leftarrow B[k - l + 1]$ 
```

Correctness

Hypothesis: after k iterations of the loop in line 3 $B[1, \dots, k]$ is sorted and $B[k] \leq A[i]$, if $i \leq m$ and $B[k] \leq A[j]$ falls $j \leq r$.

Proof by induction:

Base clause: the empty array $B[1, \dots, 0]$ is trivially sorted.

Induction step ($k \rightarrow k + 1$):

- $wlog A[i] \leq A[j]$, $i \leq m, j \leq r$.
- $B[1, \dots, k]$ is sorted by hypothesis and $B[k] \leq A[i]$.
- After $B[k + 1] \leftarrow A[i]$ $B[1, \dots, k + 1]$ is sorted.
- $B[k + 1] = A[i] \leq A[i + 1]$ (if $i + 1 \leq m$) and $B[k + 1] \leq A[j]$ if $j \leq r$.
- $k \leftarrow k + 1, i \leftarrow i + 1$: Statement holds again.

227

228

Analysis (Merge)

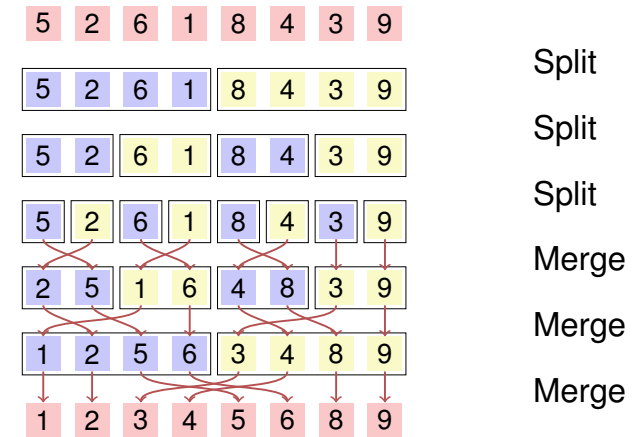
Lemma

If: array A with length n , indexes $1 \leq l < r \leq n$. $m = \lfloor (l+r)/2 \rfloor$ and $A[l, \dots, m]$, $A[m+1, \dots, r]$ sorted.

Then: in the call of $\text{Merge}(A, l, m, r)$ a number of $\Theta(r-l)$ key movements and comparison are executed.

Proof: straightforward (Inspect the algorithm and count the operations.)

Mergesort



229

230

Algorithm recursive 2-way Mergesort(A, l, r)

Input : Array A with length n . $1 \leq l \leq r \leq n$

Output : Array $A[l, \dots, r]$ sorted.

if $l < r$ **then**

```

     $m \leftarrow \lfloor (l+r)/2 \rfloor$            // middle position
    Mergesort( $A, l, m$ )              // sort lower half
    Mergesort( $A, m+1, r$ )            // sort higher half
    Merge( $A, l, m, r$ )               // Merge subsequences
  
```

Analysis

Recursion equation for the number of comparisons and key movements:

$$C(n) = C\left(\left\lfloor \frac{n}{2} \right\rfloor\right) + C\left(\left\lfloor \frac{n}{2} \right\rfloor\right) + \Theta(n) \in \Theta(n \log n)$$

231

232

Algorithm StraightMergesort(A)

Avoid recursion: merge sequences of length 1, 2, 4, ... directly

Input : Array A with length n

Output : Array A sorted

$length \leftarrow 1$

while $length < n$ **do** // Iteriere über die Längen n

$right \leftarrow 0$

while $right + length < n$ **do** // Iteriere über die Teilfolgen

$left \leftarrow right + 1$

$middle \leftarrow left + length - 1$

$right \leftarrow \min(middle + length, n)$

 Merge($A, left, middle, right$)

$length \leftarrow length \cdot 2$

Analysis

Like the recursive variant, the straight 2-way mergesort always executed a number of $\Theta(n \log n)$ key comparisons and key movements.

233

234

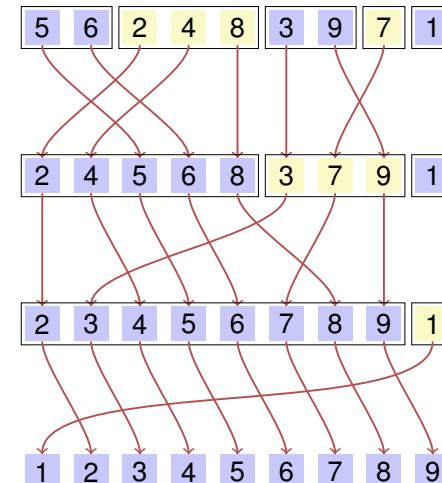
Natural 2-way mergesort

Observation: the variants above do not make use of any presorting and always execute $\Theta(n \log n)$ memory movements.

❓ How can partially presorted arrays be sorted better?

⚠ Recursive merging of previously sorted parts (*runs*) of A .

Natural 2-way mergesort



235

236

Algorithm NaturalMergesort(A)

Input : Array A with length $n > 0$

Output : Array A sorted

repeat

$r \leftarrow 0$

while $r < n$ **do**

$l \leftarrow r + 1$

$m \leftarrow l$; **while** $m < n$ **and** $A[m + 1] \geq A[m]$ **do** $m \leftarrow m + 1$

if $m < n$ **then**

$r \leftarrow m + 1$; **while** $r < n$ **and** $A[r + 1] \geq A[r]$ **do** $r \leftarrow r + 1$

 Merge(A, l, m, r);

else

$r \leftarrow n$

until $l = 1$

8.3 Quicksort

[Ottman/Widmayer, Kap. 2.2, Cormen et al, Kap. 7]

Analysis

In the best case, natural merge sort requires only $n - 1$ comparisons.

❓ Is it also asymptotically better than StraightMergesort on average?

⚠️ No. Given the assumption of pairwise distinct keys, on average there are $n/2$ positions i with $k_i > k_{i+1}$, i.e. $n/2$ runs. Only one iteration is saved on average.

Natural mergesort executes in the worst case and on average a number of $\Theta(n \log n)$ comparisons and memory movements.

237

238

Quicksort

❓ What is the disadvantage of Mergesort?

⚠️ Requires $\Theta(n)$ storage for merging.

❓ How could we reduce the merge costs?

⚠️ Make sure that the left part contains only smaller elements than the right part.

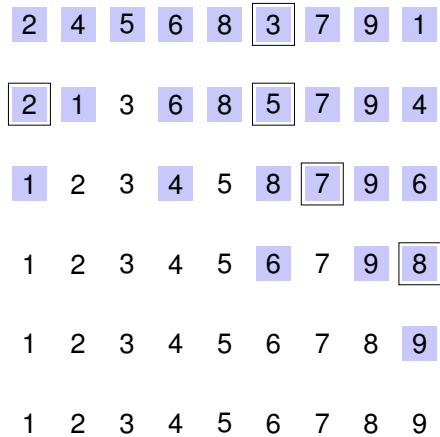
❓ How?

⚠️ Pivot and Partition!

239

240

Quicksort (arbitrary pivot)



241

Algorithm Quicksort($A[l, \dots, r]$)

Input : Array A with length n . $1 \leq l \leq r \leq n$.

Output : Array A , sorted between l and r .

if $l < r$ **then**

 Choose pivot $p \in A[l, \dots, r]$

$k \leftarrow \text{Partition}(A[l, \dots, r], p)$

 Quicksort($A[l, \dots, k - 1]$)

 Quicksort($A[k + 1, \dots, r]$)

242

Reminder: algorithm Partition($A[l, \dots, r], p$)

Input : Array A , that contains the sentinel p in $[l, r]$ at least once.

Output : Array A partitioned around p . Returns the position of p .

while $l < r$ **do**

while $A[l] < p$ **do**

$l \leftarrow l + 1$

while $A[r] > p$ **do**

$r \leftarrow r - 1$

 swap($A[l], A[r]$)

if $A[l] = A[r]$ **then**

$l \leftarrow l + 1$

// Only for keys that are not pairwise different

return $l - 1$

243

Analysis: number comparisons

Best case. Pivot = median; number comparisons:

$$T(n) = 2T(n/2) + c \cdot n, T(1) = 0 \Rightarrow T(n) \in \mathcal{O}(n \log n)$$

Worst case. Pivot = min or max; number comparisons:

$$T(n) = T(n - 1) + c \cdot n, T(1) = 0 \Rightarrow T(n) \in \Theta(n^2)$$

244

Analysis: number swaps

Result of a call to partition (pivot 3):

2 1 3 6 8 5 7 9 4

② How many swaps have taken place?

① 2. The maximum number of swaps is given by the number of keys in the smaller part.

245

Analysis: number swaps

Intellectual game

- Each key from the smaller part pay a coin when swapped.
- When a key has paid a coin then the domain containing the key is less or equal than half the previous size.
- Every key needs to pay at most $\log n$ coins. But there are only n keys.

Consequence: there are $\mathcal{O}(n \log n)$ key swaps in the worst case.

246

Randomized Quicksort

Despite the worst case running time of $\Theta(n^2)$, quicksort is used practically very often.

Reason: quadratic running time unlikely if the choice of the pivot and the presorting is not very disadvantageous.

Avoidance: randomly choose pivot. Draw uniformly from $[l, r]$.

247

Analysis (randomized quicksort)

Expected number of compared keys with input length n :

$$T(n) = (n - 1) + \frac{1}{n} \sum_{k=1}^n (T(k - 1) + T(n - k)), \quad T(0) = T(1) = 0$$

Claim $T(n) \leq 4n \log n$.

Proof by induction:

Base clause straightforward for $n = 0$ (with $0 \log 0 := 0$) and for $n = 1$.

Hypothesis: $T(n) \leq 4n \log n$ für ein n .

Induction step: $(n - 1 \rightarrow n)$

248

Analysis (randomized quicksort)

$$\begin{aligned}
 T(n) &= n - 1 + \frac{2}{n} \sum_{k=0}^{n-1} T(k) \stackrel{H}{\leq} n - 1 + \frac{2}{n} \sum_{k=0}^{n-1} 4k \log k \\
 &= n - 1 + \sum_{k=1}^{n/2} 4k \underbrace{\log k}_{\leq \log n-1} + \sum_{k=n/2+1}^{n-1} 4k \underbrace{\log k}_{\leq \log n} \\
 &\leq n - 1 + \frac{8}{n} \left((\log n - 1) \sum_{k=1}^{n/2} k + \log n \sum_{k=n/2+1}^{n-1} k \right) \\
 &= n - 1 + \frac{8}{n} \left((\log n) \cdot \frac{n(n-1)}{2} - \frac{n}{4} \left(\frac{n}{2} + 1 \right) \right) \\
 &= 4n \log n - 4 \log n - 3 \leq 4n \log n
 \end{aligned}$$

249

Analysis (randomized quicksort)

Theorem

On average randomized quicksort requires $\mathcal{O}(n \cdot \log n)$ comparisons.

250

Practical considerations

Worst case recursion depth $n - 1^8$. The also memory consumption of $\mathcal{O}(n)$.

Can be avoided: recursion only on the smaller part. Then guaranteed $\mathcal{O}(\log n)$ worst case recursion depth and memory consumption.

⁸stack overflow possible!

251

Quicksort with logarithmic memory consumption

Input : Array A with length n . $1 \leq l \leq r \leq n$.

Output : Array A , sorted between l and r .

while $l < r$ **do**

 Choose pivot $p \in A[l, \dots, r]$

$k \leftarrow \text{Partition}(A[l, \dots, r], p)$

if $k - l < r - k$ **then**

 Quicksort($A[l, \dots, k - 1]$)

$l \leftarrow k + 1$

else

 Quicksort($A[k + 1, \dots, r]$)

$r \leftarrow k - 1$

The call of Quicksort($A[l, \dots, r]$) in the original algorithm has moved to iteration (tail recursion!): the if-statement became a while-statement.

252

Practical considerations.

Practically the pivot is often the median of three elements. For example: $\text{Median3}(A[l], A[r], A[\lfloor l + r/2 \rfloor])$.

There is a variant of quicksort that requires only constant storage. Idea: store the old pivot at the position of the new pivot.