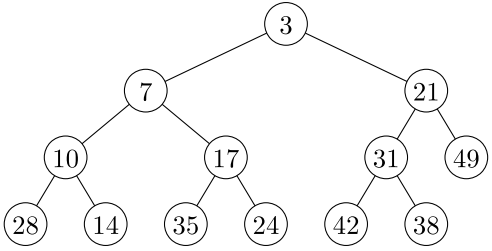
Heaps

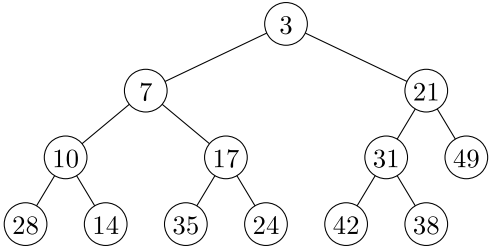


Definition

Ein Heap ist ein Binärbaum, der

- die Heapeigenschaft hat (Kinder sind größer als der Vater),
- bis auf die letzte Ebene vollständig besetzt ist,
- höchstens eine Lücke in der letzten Ebene hat, die ganz rechts liegen muß.

Heaps

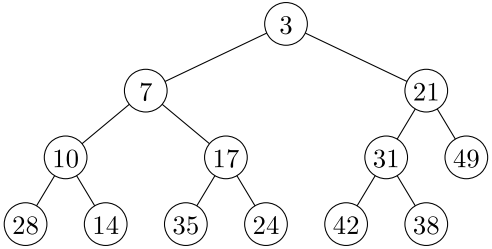


Definition

Ein Heap ist ein Binärbaum, der

- die Heapeigenschaft hat (Kinder sind größer als der Vater),
- bis auf die letzte Ebene vollständig besetzt ist,
- höchstens eine Lücke in der letzten Ebene hat, die ganz rechts liegen muß.

Heaps



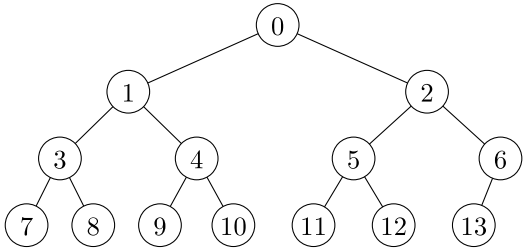
Definition

Ein Heap ist ein Binärbaum, der

- die Heapeigenschaft hat (Kinder sind größer als der Vater),
- bis auf die letzte Ebene vollständig besetzt ist,
- höchstens eine Lücke in der letzten Ebene hat, die ganz rechts liegen muß.

Heaps

Ein Heap kann sehr gut in einem Array gespeichert werden:

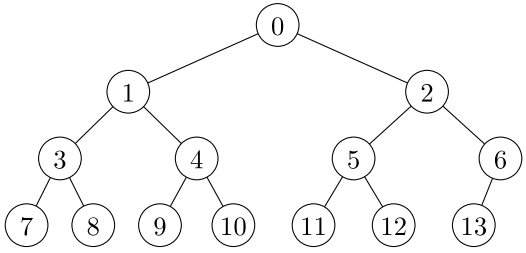


0	1	2	3	4	5	6	7	8	9	10	11	12	13

- Linkes Kind von i ist $2i + 1$
- Rechtes Kind von i ist $2i + 2$
- Vater von i ist $\lfloor (i - 1)/2 \rfloor$

Heaps

Ein Heap kann sehr gut in einem Array gespeichert werden:

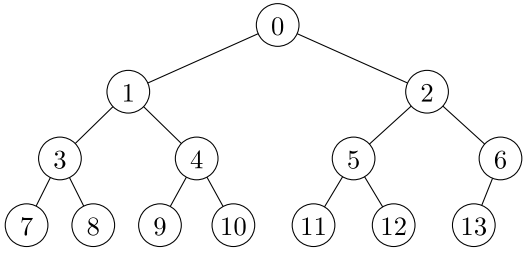


0	1	2	3	4	5	6	7	8	9	10	11	12	13

- Linkes Kind von i ist $2i + 1$
- Rechtes Kind von i ist $2i + 2$
- Vater von i ist $\lfloor (i - 1)/2 \rfloor$

Heaps

Ein Heap kann sehr gut in einem Array gespeichert werden:

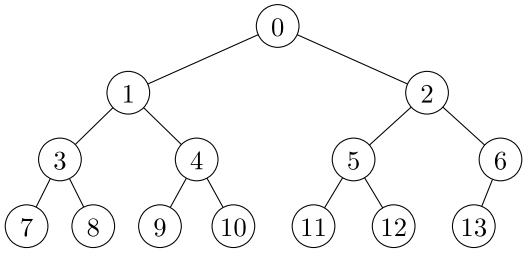


0	1	2	3	4	5	6	7	8	9	10	11	12	13

- Linkes Kind von i ist $2i + 1$
- Rechtes Kind von i ist $2i + 2$
- Vater von i ist $\lfloor (i - 1)/2 \rfloor$

Heaps

Ein Heap kann sehr gut in einem Array gespeichert werden:



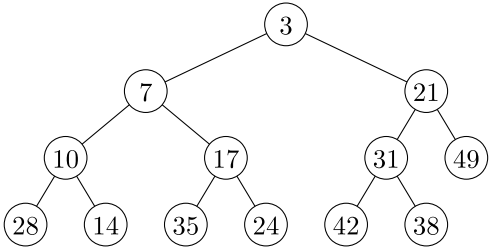
0	1	2	3	4	5	6	7	8	9	10	11	12	13

- Linkes Kind von i ist $2i + 1$
- Rechtes Kind von i ist $2i + 2$
- Vater von i ist $\lfloor (i - 1)/2 \rfloor$

Heaps

Gegeben sind n Schlüssel $a_1, \ldots, a_n$.

Frage: Wie können wir einen Heap konstruieren, der diese Schlüssel enthält?

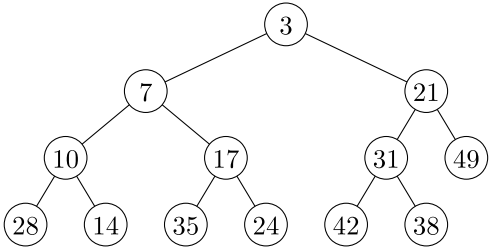


Antwort: Nacheinander in einen leeren Heap einfügen.

Heaps

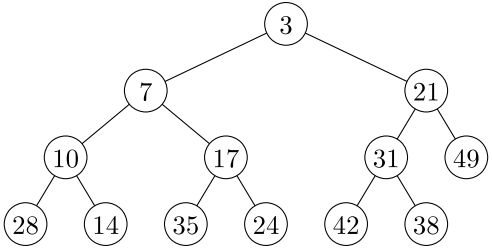
Gegeben sind n Schlüssel $a_1, \ldots, a_n$.

Frage: Wie können wir einen Heap konstruieren, der diese Schlüssel enthält?



Antwort: Nacheinander in einen leeren Heap einfügen.

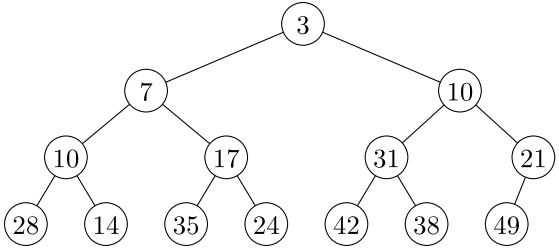
Einfügen in einen Heap



Es soll der Schlüssel 10 eingefügt werden.

1. Hänge den Schlüssel an das Ende
2. Die Heap-Eigenschaft ist jetzt verletzt
3. Lasse den neuen Schlüssel zur richtigen Stelle aufsteigen.

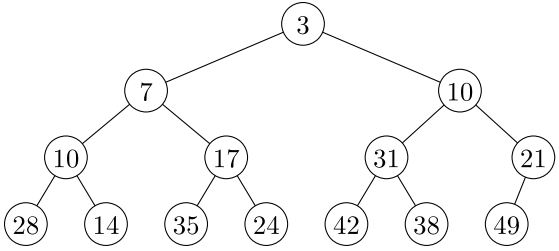
Einfügen in einen Heap



Es soll der Schlüssel 10 eingefügt werden.

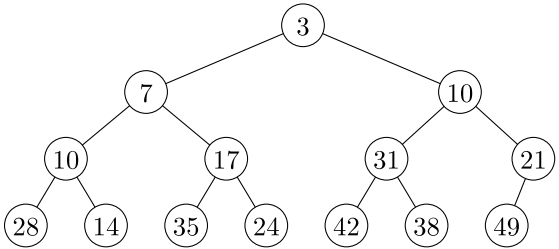
1. Hänge den Schlüssel an das Ende
2. Die Heap-Eigenschaft ist jetzt verletzt
3. Lasse den neuen Schlüssel zur richtigen Stelle aufsteigen.

Einfügen in einen Heap



Es soll der Schlüssel 10 eingefügt werden.

1. Hänge den Schlüssel an das Ende
2. Die Heap-Eigenschaft ist jetzt verletzt
3. Lasse den neuen Schlüssel zur richtigen Stelle aufsteigen.



Java

Code missing!

Ursprüngliches Problem:

Gegeben sind n Schlüssel $a_1, \ldots, a_n$.

Frage: Wie können wir einen Heap konstruieren, der diese Schlüssel enthält?

Java

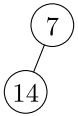
```
public Heap(Collection⟨D⟩ data) {
  super();
  addAll(data);
  for(int i = 1; i < size(); i++) bubble_up(i);
}
```

So wird der Heap schrittweise aufgebaut.

Wir bilden einen Heap aus den Schlüsseln 7, 14, 21, 28, 35, 42, 49, 3, 10, 17, 24, 31, und 38:

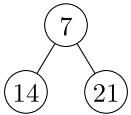
So wird der Heap schrittweise aufgebaut.

Wir bilden einen Heap aus den Schlüsseln 7, 14, 21, 28, 35, 42, 49, 3, 10, 17, 24, 31, und 38:



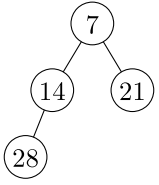
So wird der Heap schrittweise aufgebaut.

Wir bilden einen Heap aus den Schlüsseln 7, 14, 21, 28, 35, 42, 49, 3, 10, 17, 24, 31, und 38:



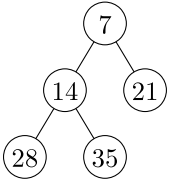
So wird der Heap schrittweise aufgebaut.

Wir bilden einen Heap aus den Schlüsseln 7, 14, 21, 28, 35, 42, 49, 3, 10, 17, 24, 31, und 38:



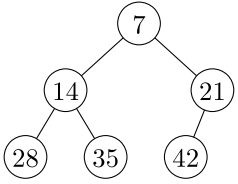
So wird der Heap schrittweise aufgebaut.

Wir bilden einen Heap aus den Schlüsseln 7, 14, 21, 28, 35, 42, 49, 3, 10, 17, 24, 31, und 38:



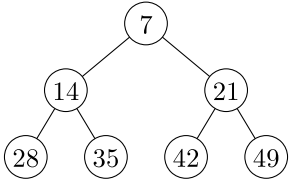
So wird der Heap schrittweise aufgebaut.

Wir bilden einen Heap aus den Schlüsseln 7, 14, 21, 28, 35, 42, 49, 3, 10, 17, 24, 31, und 38:



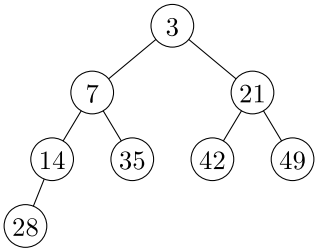
So wird der Heap schrittweise aufgebaut.

Wir bilden einen Heap aus den Schlüsseln 7, 14, 21, 28, 35, 42, 49, 3, 10, 17, 24, 31, und 38:



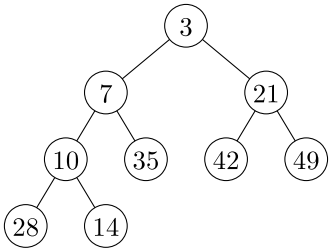
So wird der Heap schrittweise aufgebaut.

Wir bilden einen Heap aus den Schlüsseln 7, 14, 21, 28, 35, 42, 49, 3, 10, 17, 24, 31, und 38:



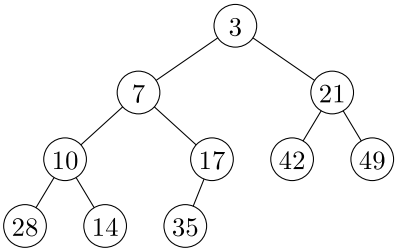
So wird der Heap schrittweise aufgebaut.

Wir bilden einen Heap aus den Schlüsseln 7, 14, 21, 28, 35, 42, 49, 3, 10, 17, 24, 31, und 38:



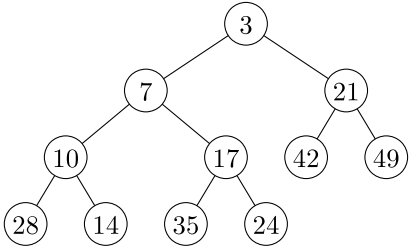
So wird der Heap schrittweise aufgebaut.

Wir bilden einen Heap aus den Schlüsseln 7, 14, 21, 28, 35, 42, 49, 3, 10, 17, 24, 31, und 38:



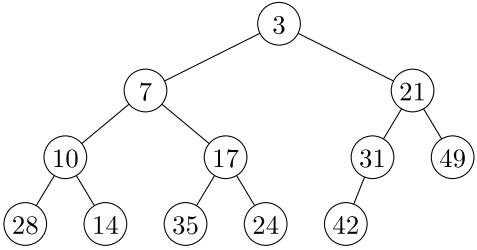
So wird der Heap schrittweise aufgebaut.

Wir bilden einen Heap aus den Schlüsseln 7, 14, 21, 28, 35, 42, 49, 3, 10, 17, 24, 31, und 38:



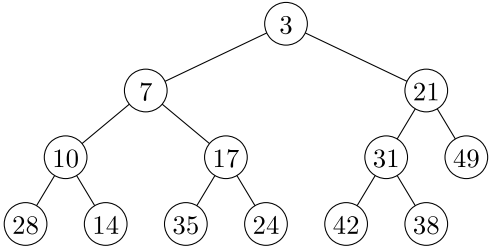
So wird der Heap schrittweise aufgebaut.

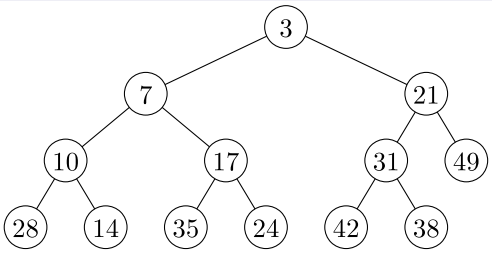
Wir bilden einen Heap aus den Schlüsseln 7, 14, 21, 28, 35, 42, 49, 3, 10, 17, 24, 31, und 38:



So wird der Heap schrittweise aufgebaut.

Wir bilden einen Heap aus den Schlüsseln 7, 14, 21, 28, 35, 42, 49, 3, 10, 17, 24, 31, und 38:



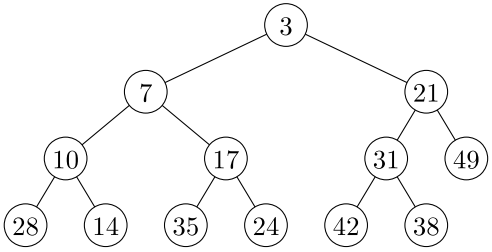


1. Welches ist der größte Schlüssel?
2. Welches ist der kleinste Schlüssel?

Der kleinste Schlüssel ist in der Wurzel.

→ Wiederholtes Entfernen des kleinsten Schlüssels liefert die Schlüssel in aufsteigender Reihenfolge.

Können wir die Wurzel aus einem Heap effizient entfernen?

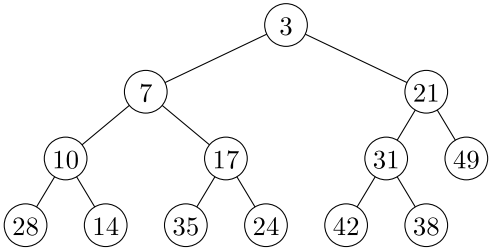


1. Welches ist der größte Schlüssel?
2. Welches ist der kleinste Schlüssel?

Der kleinste Schlüssel ist in der Wurzel.

→ Wiederholtes Entfernen des kleinsten Schlüssels liefert die Schlüssel in aufsteigender Reihenfolge.

Können wir die Wurzel aus einem Heap effizient entfernen?

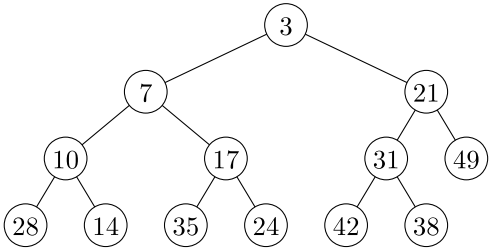


1. Welches ist der größte Schlüssel?
2. Welches ist der kleinste Schlüssel?

Der kleinste Schlüssel ist in der Wurzel.

→ Wiederholtes Entfernen des kleinsten Schlüssels liefert die Schlüssel in aufsteigender Reihenfolge.

Können wir die Wurzel aus einem Heap effizient entfernen?



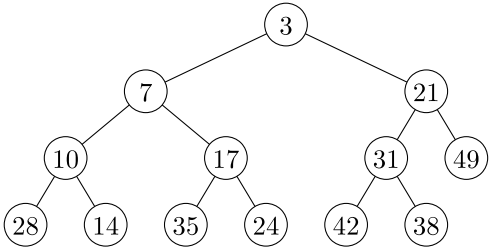
1. Welches ist der größte Schlüssel?
2. Welches ist der kleinste Schlüssel?

Der kleinste Schlüssel ist in der Wurzel.

→ Wiederholtes Entfernen des kleinsten Schlüssels liefert die Schlüssel in aufsteigender Reihenfolge.

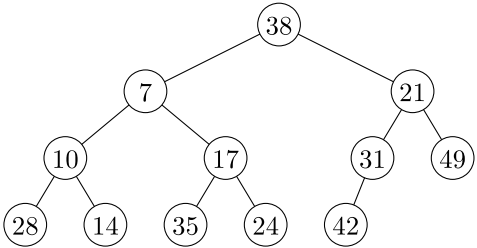
Können wir die Wurzel aus einem Heap effizient entfernen?

Entfernen der Wurzel – extract-min



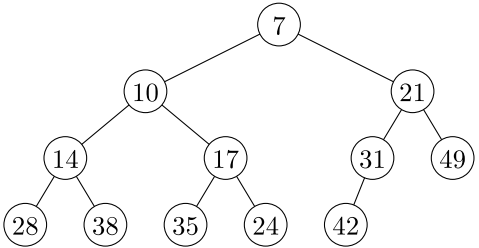
1. Ersetze den Schlüssel der Wurzel durch den Schlüssel des letzten Knoten
2. Lösche den letzten Knoten
3. Jetzt ist die Heap-Eigenschaft verletzt
4. Lasse den Schlüssel in der Wurzel hinuntersinken.

Entfernen der Wurzel – extract-min



1. Ersetze den Schlüssel der Wurzel durch den Schlüssel des letzten Knoten
2. Lösche den letzten Knoten
3. Jetzt ist die Heap-Eigenschaft verletzt
4. Lasse den Schlüssel in der Wurzel hinuntersinken.

Entfernen der Wurzel – extract-min

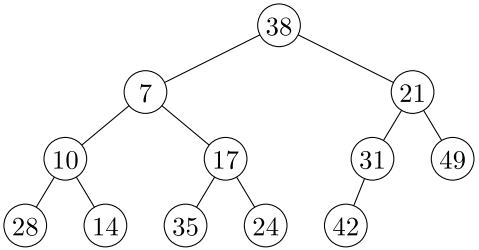


1. Ersetze den Schlüssel der Wurzel durch den Schlüssel des letzten Knoten
2. Lösche den letzten Knoten
3. Jetzt ist die Heap-Eigenschaft verletzt
4. Lasse den Schlüssel in der Wurzel hinuntersinken.

Java

Code missing!

extract-min

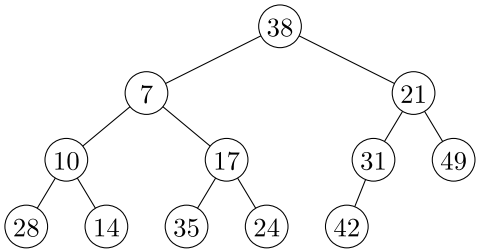


Hinuntersinken: Zwei Vergleiche pro Schritt.

Alternative:

1. Hinuntersinken bis zum Blatt.
2. Dann von dort aufsteigen.

extract-min



Hinuntersinken: Zwei Vergleiche pro Schritt.

Alternative:

1. Hinuntersinken bis zum Blatt.
2. Dann von dort aufsteigen.

Heapsort

Der Algorithmus Heapsort ist jetzt einfach:

1. Konstruiere einen Max-Heap (größer = oben)
2. Entferne wiederholt das größte Element
3. Speichere es an der frei werdenden Position

Laufzeit: $O(n \log n)$

Einfügen und extract_max in $O(\log n)$ Zeit

Heapsort ist ein in-place–Verfahren.

Heapsort

Der Algorithmus Heapsort ist jetzt einfach:

1. Konstruiere einen Max-Heap (größer = oben)
2. Entferne wiederholt das größte Element
3. Speichere es an der frei werdenden Position

Laufzeit: $O(n \log n)$

Einfügen und extract_max in $O(\log n)$ Zeit

Heapsort ist ein in-place–Verfahren.

Heapsort

Der Algorithmus Heapsort ist jetzt einfach:

1. Konstruiere einen Max-Heap (größer = oben)
2. Entferne wiederholt das größte Element
3. Speichere es an der frei werdenden Position

Laufzeit: $O(n \log n)$

Einfügen und extract_max in $O(\log n)$ Zeit

Heapsort ist ein in-place–Verfahren.

Heapsort

Der Algorithmus Heapsort ist jetzt einfach:

1. Konstruiere einen Max-Heap (größer = oben)
2. Entferne wiederholt das größte Element
3. Speichere es an der frei werdenden Position

Laufzeit: $O(n \log n)$

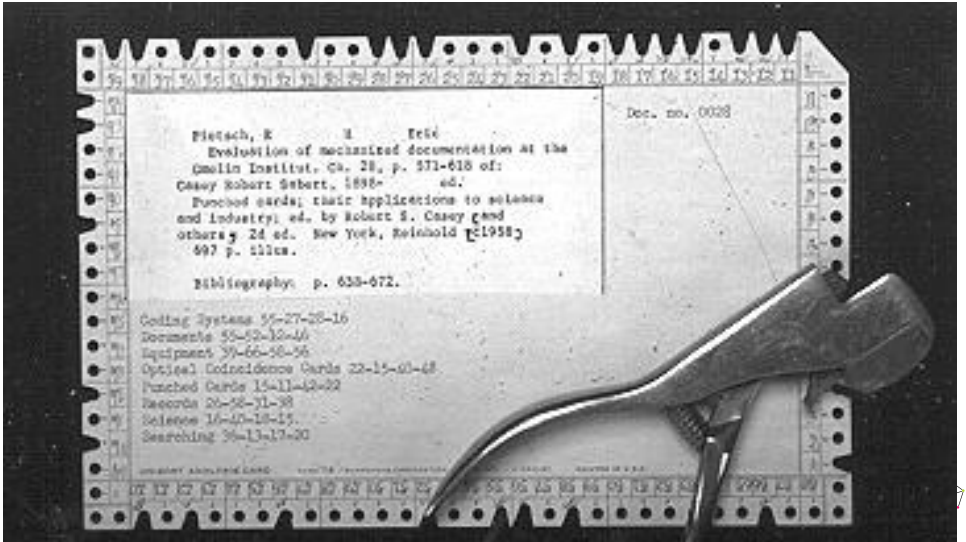
Einfügen und extract_max in $O(\log n)$ Zeit

Heapsort ist ein in-place–Verfahren.

Heapsort – Beispiel

Heapsort – Schnellere Variante

Radixsort


Doc. no. 0028
Pietsch, R H Erle
Evaluation of mechanized documentation at the
Camelin Institut. Ch. 28, p. 571-618 of:
Casey Robert Sebert, 1898- ed.
Punched cards; their applications to science
and industry; ed. by Robert S. Casey and
others. 2d ed. New York, Reinhold [c1958]
697 p. illus.
Bibliography: p. 633-672.
Coding Systems 55-27-28-16
Documents 55-62-12-40
Equipment 39-66-58-56
Optical Coincidence Cards 22-15-60-48
Punched Cards 15-11-42-32
Records 20-58-31-38
Science 16-40-18-15.
Searching 36-13-17-20

Gegeben seien Binärzahlen einer gewissen Länge:

1110101000001000100
1011000110011111100 1
0111110011111010010 0
0000101111010001000 1
0101101100011000000 0
1101011100010000011 0
0001011101110100001 1
0110101111100000100 1
1000110110000000110 0
1110000001000000101 0
1010101000001111000 0
1010010000101100101 0
0000000001000000001 1
0100000011001101000 1
0100000011001101000 1
0000000001000000001 1
1011000110011111100 1
1110101000001000100 0

Sortiere nach dem ersten Bit.

Bitstrings in Matrix $A[1 \ldots n, 1 \ldots w]$.
Vor dem Bitarray stehe $00 \ldots 0$ und danach $11 \ldots 1$.

Algorithmus

```
procedure radix_exchange_sort(i, a, b) :
  if i > w then return fi;
  s := a; t := b;
  while s < t do
    while A[s, i] = 0 do s := s + 1;
    while A[t, i] = 1 do t := t − 1;
    vertausche A[s] und A[t]
  od;
  vertausche A[s] und A[t];
  radix_exchange_sort(i + 1, a, t);
  radix_exchange_sort(i + 1, s, b)
```

11101010000010001000	11101010000010001000
10110001100111111001	01111100111110100100
01111100111110100100	01011011000110000000
00001011110100010001	11010111000100000110
01011011000110000000	10001101100000001100
11010111000100000110	11100000010000001010
00010111011101000011	10101010000011110000
01101011111000001001	10100100001011001010
10001101100000001100	10110001100111111001
11100000010000001010	00001011110100010001
10101010000011110000	00010111011101000011
10100100001011001010	01101011111000001001
00000000010000000011	00000000010000000011
01000000110011010001	01000000110011010001

Sortiere nach dem letzten Bit.

Dann nach dem vorletzten usw. ($\rightarrow$ stabil!)

1110101000001000100**0**	1110101000001000100**0**
1011000110011111110**1**	0111110011111010010**0**
0111110011111010010**0**	0101101100011000000**0**
0000101111010001000**1**	1101011100010000011**0**
0101101100011000000**0**	1000110110000000110**0**
1101011100010000011**0**	1110000001000000101**0**
0001011101110100001**1**	1010101000001111000**0**
0110101111100000100**1**	1010010000101100101**0**
1000110110000000110**0**	1011000110011111110**1**
1110000001000000101**0**	0000101111010001000**1**
1010101000001111000**0**	0001011101110100001**1**
1010010000101100101**0**	0110101111100000100**1**
0000000001000000001**1**	0000000001000000001**1**
0100000011001101000**1**	0100000011001101000**1**

Sortiere nach dem letzten Bit.

Dann nach dem vorletzten usw. ($\rightarrow$ stabil!)

Straight-Radix-Sort

Algorithmus

```
procedure straight_radix_sort(i) :
  pos0 := 1;  pos1 := n + 1;
  for k = 1, ..., n do pos1 := pos1 − A[k, i] od;
  for k = 1, ..., n do
    if A[k, i] = 0 then B[pos0] := A[k];  pos0 := pos0 + 1
                   else B[pos1] := A[k];  pos1 := pos1 + 1 fi
  od
```

Sortiere stabil nach dem *i*-ten Bit.

Ergebnis ist in B.

Straight-Radix-Sort

Vollständiger Algorithmus:

Algorithmus

```
procedure straight_radix_sort :
  for i = w, ..., 1 do
    pos0 := 1;  pos1 := n + 1;
    for k = 1, ..., n do pos1 := pos1 − A[k, i] od;
    for k = 1, ..., n do
      if A[k, i] = 0 then B[pos0] := A[k];  pos0 := pos0 + 1
                     else B[pos1] := A[k];  pos1 := pos1 + 1 fi
    od;
    A := B;
  od;
```