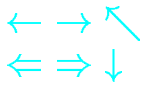
Algorithmen und Datenstrukturen
(Th. Ottmann und P. Widmayer)

Folien: Schnelle Sortierverfahren

Autor: Stefan Edelkamp

Institut für Informatik
Georges-Köhler-Allee
Albert-Ludwigs-Universität Freiburg

1 Überblick



Überblick

Kriterien für Sortierverfahren

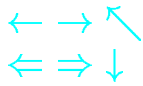
State-of-the-Art

Clever-Quicksort

Heapsort

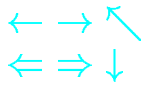
Quick-Heapsort

2 Kriterien für Sortierverfahren



- Allgemeinheit,
- Einfachheit,
- In Situ–Eigenschaft (wenig Platz),
- Schlüsselvergleiche schwer und
- größtmögliche Anzahl der Vergleiche $\leq n \log n + cn$,
- übrigen Operation $\leq c(n \log n)$.

3 State-of-the-Art



Untere Schranke: $C_{max}(n) > C_{av}(n) \geq \lceil \log(n!) \rceil - 1 \approx n \log n - 1.4427n,$

Ziel: $C_{max}(n)$ bzw. $C_{av}(n) \leq b \cdot n \log n + cn$ für kleine c, b

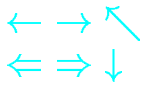
Quicksort-Varianten:

QUICKSORT (Hoare 1962)
$C_{av}(n) \approx 1.386n \log n - 2.846n + O(\log n)$
CLEVER-QUICKSORT (Hoare 1962)
$C_{av}(n) \approx 1.188n \log n - 2.255n + O(\log n)$
QUICK-HEAPSORT (Cantone & Cincotti 2000)
$C_{av}(n) = n \log n + 3n + o(n)$
QUICK-WEAK-HEAPSORT
$C_{av}(n) = n \log n + 0.2n + o(n)$

Heapsort-Varianten:

HEAPSORT (Williams 1964) bzw. (Floyd 1964)
$C_{max}(n) = 2n \log n + O(n)$
BOTTOM-UP-HEAPSORT (Wegener 1993)
$C_{max}(n) = 1.5n \log n + O(n)$ allerdings
$C_{av}(n) = n \log n + O(n)$
WEAK-HEAPSORT (Dutton 1993)
$C_{max}(n) = n \log n + 0.1n$
RELAXED-WEAK-HEAPSORT
$C_{max}(n) = n \log n - 0.9n$

4 Clever-Quicksort



Auswahl des Pivotelementes (Median-of-3 Strategie):

- a) $m = (l + r)/2$ oder b) $m = l + 1$
- Pivot: mittleres Element von $\{A[l], A[m], A[r]\}$
- vertausche $A[r]$ mit Pivot, weiter wie bisher

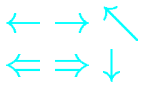
Worst-case:

- verschwindet in a) für auf- bzw. absteigende Sortierung
- existiert immer noch

Satz: Schlüsselvergleiche im Mittel

$$C_{av}(n) \approx 1.188\, n \log n - 2.255\, n + O(\log n)$$

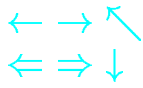
Implementation



Bemerkung: Siehe Sedgewick: The analysis of quicksort programs, Acta Informatica, Journal of Algorithms, 15(1):76-100, 1993

```
static void sort(Orderable A[], int left,int right) {
  if (right-left >= 3) {
    if(A[right].less(A[left+1])) swap(A,left+1,right);
    if(A[right].less(A[left])) swap(A,left,right);
    if(A[left].less(A[left+1])) swap(A,left+1,left);
    int i = left+1, j = right;
    Orderable v = A[left];
    do {
      do { i++; } while (A[i].less(v));
      do { j--; } while (v.less(A[j]));
      if (j >= i) swap(A,i,j);
    }
    while(j >= i);
    swap(A,left,j);
    if (j-left < right-i+1) {
      sort(A,left,j-1);
      sort(A,i,right);
    }
    else {
      sort(A,i,right);
      sort(A,left,j-1);
    }
  }
  else
    threesort(A,left,right);
  }
}
```

5 Heapsort



Effizientes Sortieren durch Auswählen

Prinzip von Heapsort: Sortieren durch wiederholtes Auswählen des Maximums (Auswahlsort). Verwende Struktur (Heap), die Bestimmung des Maximums effizient unterstützt.

```
Verwandle unsortierte Folge F
  in einen Heap;
while (F <> {})
  gebe maximales Element von F aus
    entferne maximales Element aus F und
    verwandle Restfolge wieder in
    einen Heap.
```

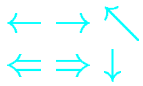
Definition:
Folge $F = (k_1, k_2, \ldots, k_n)$ von Schlüsseln heißt Heap, wenn für alle $i = 1, 2, \ldots, n/2$ gilt:

$$k_i \geq k_{2i} \quad \text{und} \quad k_i \geq k_{2i+1}$$

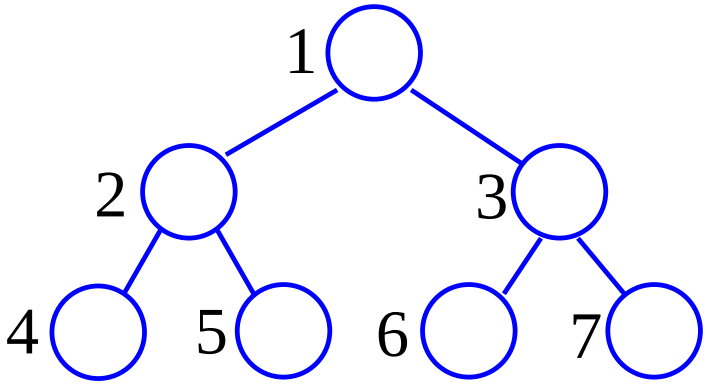
1	2	3	4	5	6	7
47	17	43	15	8	4	2

47	15	17	8	43	4	2

Veranschaulichung



1	2	3	4	5	6	7
47	17	43	15	8	4	2



Vollständiger Binärbaum mit Positionsnummern:

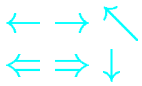
- Level i hat 2^i Knoten (außer dem letzten Level)
- Knoten von oben nach unten und von links nach rechts numeriert
- Knoten i hat Knoten $2i$ als linken und Knoten $2i + 1$ als rechten Sohn

Heapbedingung:

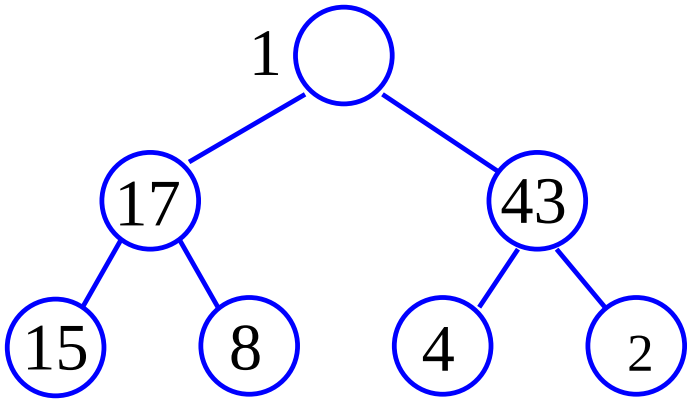
- Schlüssel(p) $\geq$ Schlüssel(p_l)
- Schlüssel(p) $\geq$ Schlüssel(p_r)

Maximum ist an der Wurzel (Position 1)

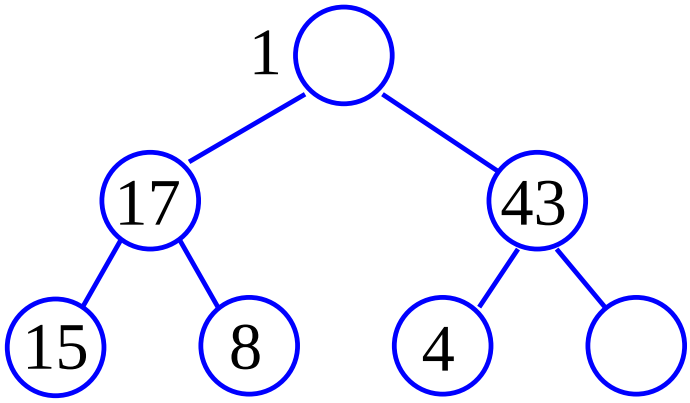
Entfernen des Maximums



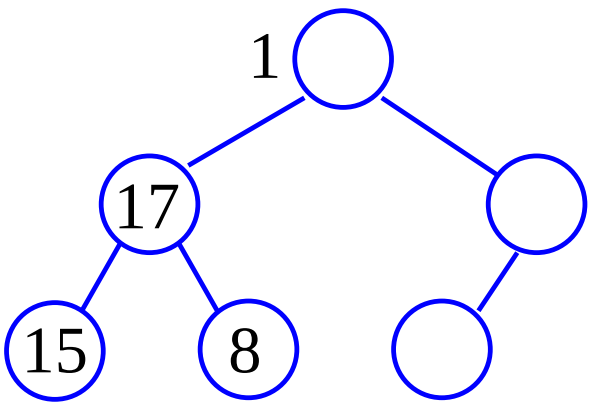
1. Entferne k_1



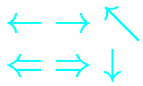
2. Übertrage k_n an die Wurzel



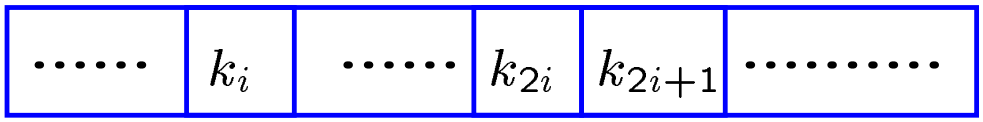
3. Versickere k_1

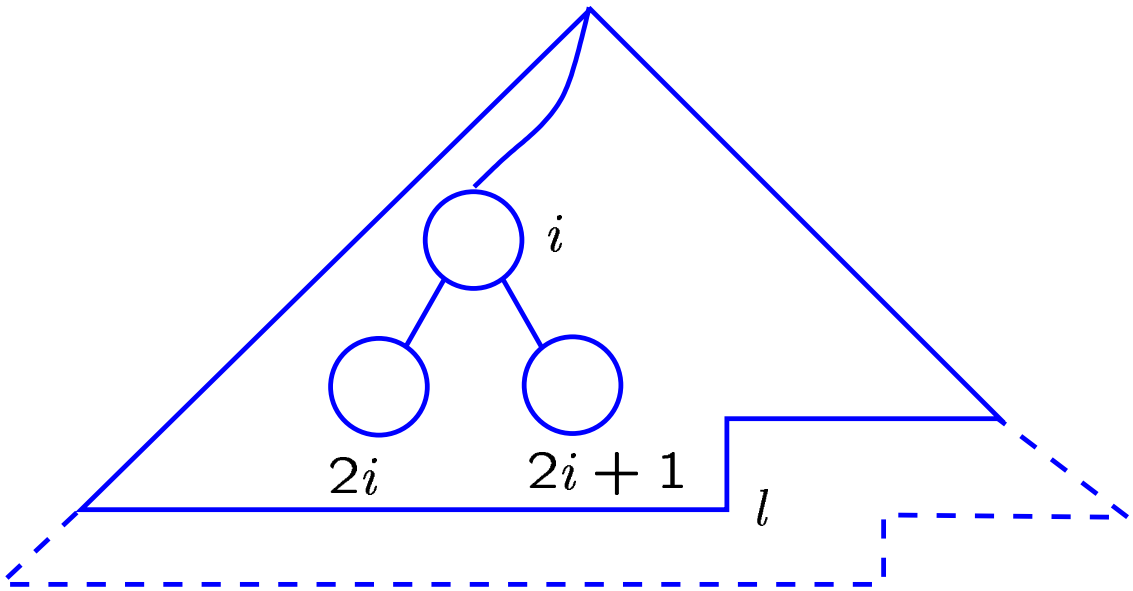


Versickern



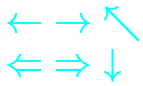
Allgemeiner: versickere k_i in $i, \ldots, l$



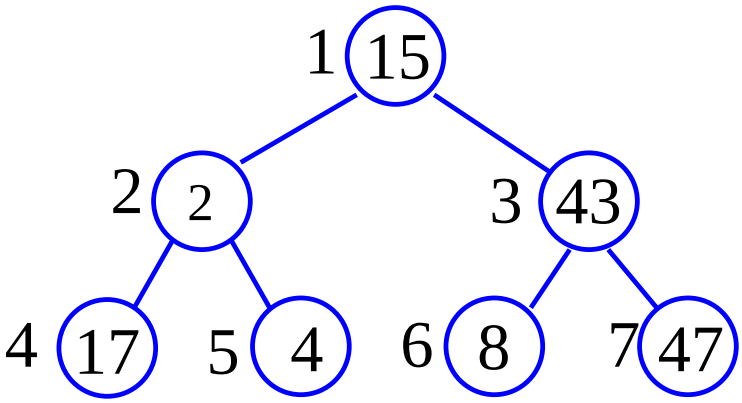



```
class HeapSort extends SortAlgorithm {
  static void pushdown(Orderable A[],int i,int n) {
    while (2*i <= n) { // i hat linken Sohn
      int j = 2*i;
      if (j < n && A[j].less(A[j+1]))
        j = j + 1; // 2i+1 ist groesserer Sohn
      if (A[i].less(A[j])) {
        swap(A,i,j);
        i = j;          // versickere weiter
      }
      else i = n;       // exit loop
    }
  }
  ...
}
```

Erstellen eines Heaps



Beobachtung:
Die Elemente $A[n/2], \ldots, A[n]$ erfüllen bereits die Heap-Bedingung

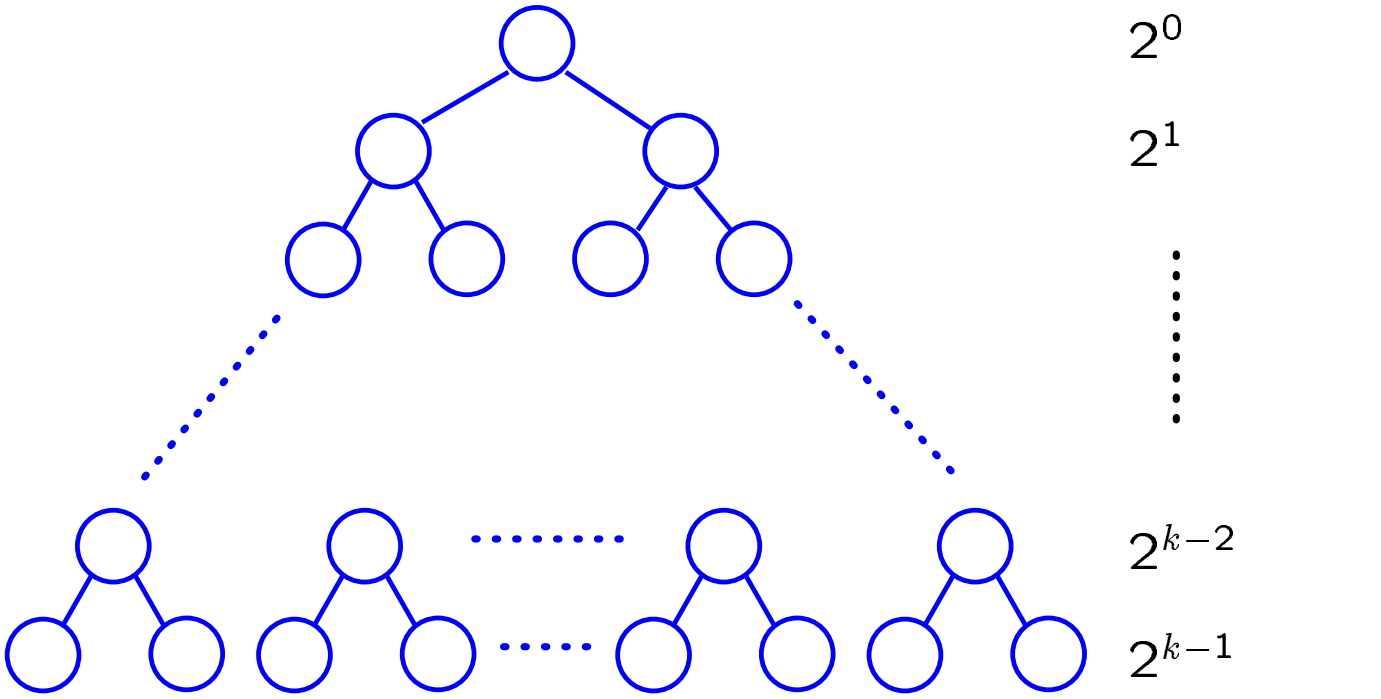



```
public static void heapify (Orderable A[]) {
  // verwandle A in einen Heap
  int n = A.length-1;
  for (int i = n/2; i >= 1; i--) {
    pushdown(A,i,n);
  }
}
```

Äußere Schleife

```
public static void sort (Orderable A[]){
  heapify(A);           // generiere Heap
  for (int i = A.length-1; i >= 2; i--) {
    swap(A,1,i);
    pushdown(A,1,i-1); // versickere Wurzel
  }
}
```

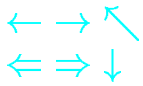
Iteriertes Versickern



$(k-1)-$ Tiefe	# Elemente	# Vergleiche
0	$2^{k-1}/2^0$	0
1	$2^{k-1}/2^1$	$2 \cdot 1$
2	$2^{k-1}/2^2$	$2 \cdot 2$
$\vdots$	$\vdots$	$\vdots$
i	$2^{k-1}/2^i$	$2i$
$\vdots$	$\vdots$	$\vdots$
$k-1$	$2^{k-1}/2^{k-1}$	$2(k-1)$

Gesamtaufwand = $2^{k-1} \sum_{i=0}^{k-1} 2\frac{i}{2^i}$

Analyse



Wissen:

Falls A Heap mit n Schlüsseln

max(A): 1, deletemax(A): $2 \log n$ (2 Vergl. pro Niveau)

heapify(A): $2n$

Insgesamt:

Laufzeit: $O(n \log n)$, genauer: $\leq 2n \log n + 2n$ Schlüsselvergleiche

Platzbedarf: n für das Eingabearray + $O(1)$

Schlimmster Fall in Heapsort:

Es wird immer das kleinste Element versickert

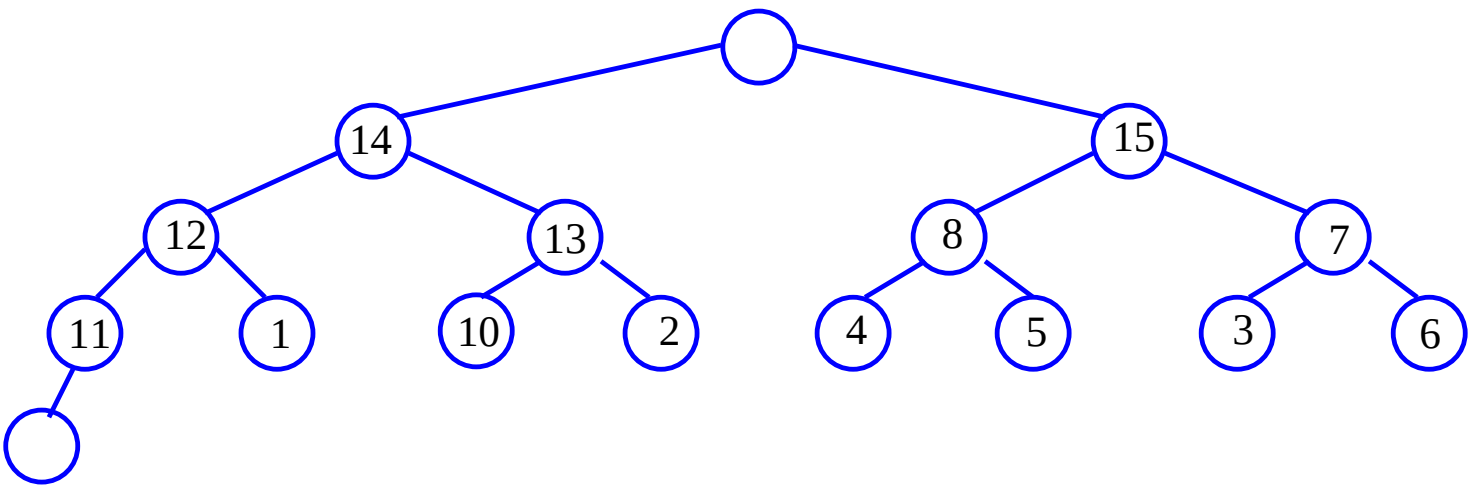
Beobachtung:

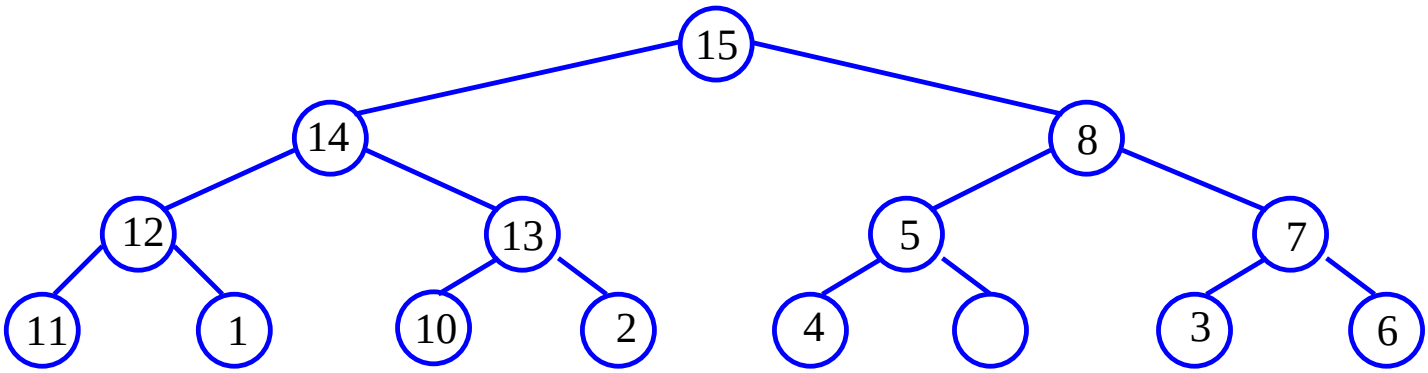
Die erwartete Tiefe eines versickerten Elementes im Heap ist groß.

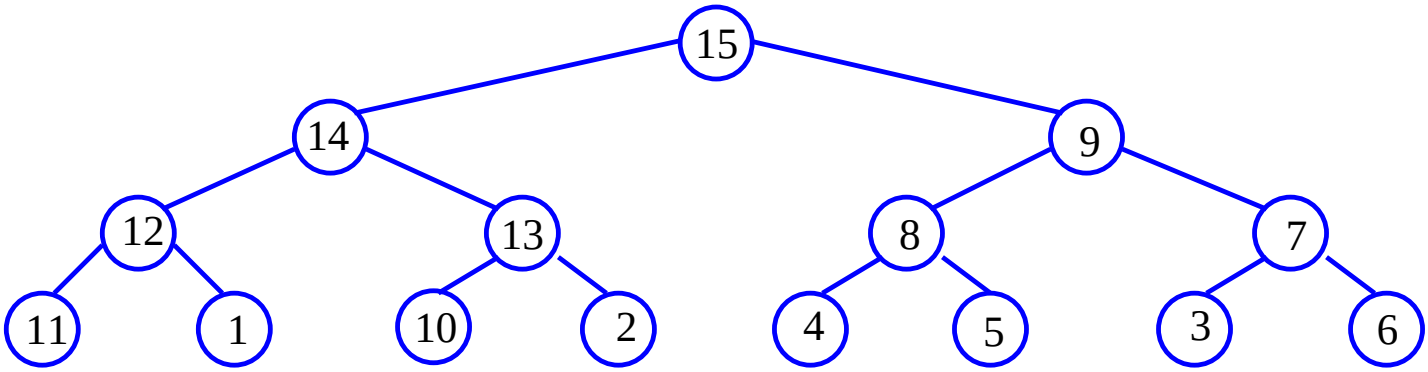
Idee Bottom-Up-Heapsort:

- Bestimme nur größeren der beiden Söhne mit einem Schlüsselvergleich pro Niveau
- sinke immer bis zu einem Blatt (search-leaf)
- steige dann wieder auf (bottom-up-search)

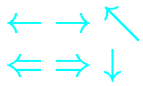
Bottom-Up-Heapsort







Bottom-Up-Heapsort: Implementation



Versickern:

```
void pushdown(int root,int m) {
  int j = LeafSearch(root,m);
  j = BottomUpSearch(root,j);
  Interchange(root,j);
}
```

Suche speziellen Weg:

```
static int LeafSearch(Orderable A[],int root,int m) {
  int i = 0, j = Path[i++] = root;
  while((2*j)<m) {
    if (A[2*j+1].less(A[2*j])) Path[i++] = j = 2*j;
    else                       Path[i++] = j = 2*j+1;
  }
  if(2*j==size) j = Path[i++] = m;
  return j;
}
```

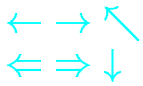
Suche Einsinkposition:

```
static int BottomUp-
Search(Orderable A[],int i,int j){
  while(j>i && A[j].less(A[i]))  j /= 2;
  return j;
}
```

Ringtausch effizienter als iteriertes Vertauschen:

```
static void Inter-
change(Orderable A[],int i,int j){
  int k=0; Orderable v = A[Path[0]];
  for(;Path[k]<j;k++) A[Path[k]] = A[Path[k+1]];
  A[Path[k]] = v;
}
```

Ergebnisse



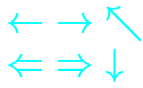
Satz: Bottom-up Heapsort führt zum Sortieren einer Folge von n Schlüsseln im schlechtesten Fall nur $1.5n \log n + (2 - c(n))n + O(\log^2 n)$ Schlüsselvergleiche aus (Wegener 1993).

Fleischer (1991) sowie Schaffer und Sedgewick (1993) haben worst-case Beispiele angegeben, bei denen die Anzahl der wesentlichen Vergleiche für BOTTOM-UP-HEAPSORT gleich $1.5n \log n - o(n \log n)$ ist.

Satz: Im Mittel benötigt Bottom-up Heapsort (nur) $n \log n + O(n)$ Schlüsselvergleiche (Li & Vitányi 1993).

Experimente:
Sei $d(n)$ so gewählt, daß $n \log n + d(n)n$ die erwartete Anzahl an Schlüsselvergleichen von BOTTOM-UP-HEAPSORT ist. Dann liegt $d(n)$ im Intervall von [0.34,0.39]. Diese Zahl ist groß für $n \approx 2^k$ und klein für $n \approx 1.4 \cdot 2^k$.

6 Quick-Heapsort



Idee: Hybrid-Algorithmus, verbindet den Divide-and-Conquer Ansatz von QUICKSORT mit HEAPSORT.

Trick: Aufteilung des Elementarrays in umgekehrter Richtung, d.h. $A[1..j-1]$ größer als Pivot $A[j]$ und $A[j+1..n]$ kleiner-gleich dem Pivot an j.

Einseitiges Heapsort: EXTERNAL-HEAPSORT wird für das kleinere Elementarray aufgerufen. Ist dies der erste Teil so konstruiert QUICK-HEAPSORT einen Max-, sonst einen Min-Heap.

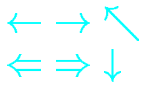
Versickerung in EXTERNAL-HEAPSORT: Das Wurzelelement wird jeweils durch das kleinere Kind ersetzt (1 Vergleich) und an die Endposition in den jeweils anderen Teilbereich geschrieben (Tausch).

Rekursion: $C^*_{av}(n)$ Mittel von EXTERNAL-HEAPSORT. $C_{av}(1) = 0$, $C_{av}(2) = 1$ und $C_{av}(2n) =$

$$2n + 1 + \frac{1}{2n}\left\{\sum_{j=1}^{n} C^*_{av}(j-1) + C_{av}(2n-j) + \sum_{j=n+1}^{2n} C^*_{av}(2n-j) + C_{av}(j-1)\right\}$$

Satz: Im Mittel sortiert QUICK-HEAPSORT n Elemente in $\leq n \log n + 3n + o(n)$ Vergleichen.

EXTERNAL-HEAPSORT



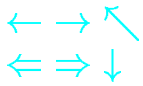
a) Fülle Lücke in Max-Heap mit jeweils kleinerem Kind und gebe resultierendes Blatt zurück

```
static int max_special_leaf(Orderable A[],
                            int left, int right) {
  int i = left+1;  // left son of root
  while( i<right ) {
    if (A[i].less(A[i+1])) i++;
    A[left + (i-left+1)/2 - 1] = A[i];
    i = left + 2*(i-left+1) - 1;
  }
  if (i == right) {
    A[left+(i-left+1)/2 - 1] = A[i];
    i = left + 2*(i-left+1) - 1;
  }
  return left + (i-left)/2;
}
```

b) Sortiere A in den Grenzen $I = [heapleft..heapright]$ hinein in den Bereich $[workright - |I| + 1..workright]$

```
static void external_maxheap_sort(Orderable A[],
     int heapleft, int heapright, int workright) {
 build_max_heap(A, heapleft, heapright);
 for(int j=heapright; j>=heapleft; j--) {
   Orderable tmp = A[workright];
   A[workright--] = A[heapleft];
   int l = max_special_leaf(A, heapleft, heapright);
   A[l] = tmp;
  }
}
```

Implementation: *QUICK-HEAPSORT*



c) Der Aufteilungsschritt:

```
static void quickheap(Orderable A[],
                      int left, int right) {
  while (right-left > 0) {
    int i = left, j = right+1;
    Orderable v = A[left];
    do {
      do j--; while (j>=i && A[j].less(v));
      do i++; while (i<=j && v.lessEqual(A[i]));
      if (j > i) swap(A,i,j);
    }
    while(j >= i);
    swap(A,left, right - (j-left));
    if( (j-left) >= (right-j) ) {
      external_minheap_sort( A, j+1, right, left );
      left = right - (j-left) + 1;
    }
    else {
      external_maxheap_sort( A, left+1, j, right );
      right = right - (j-left) - 1;
    }
  }
}
```

d) Sortierung im Bereich $A[1..n]$:

```
public static void sort(Orderable A[]) {
  quickheap(A,1,A.length-1);
}
```