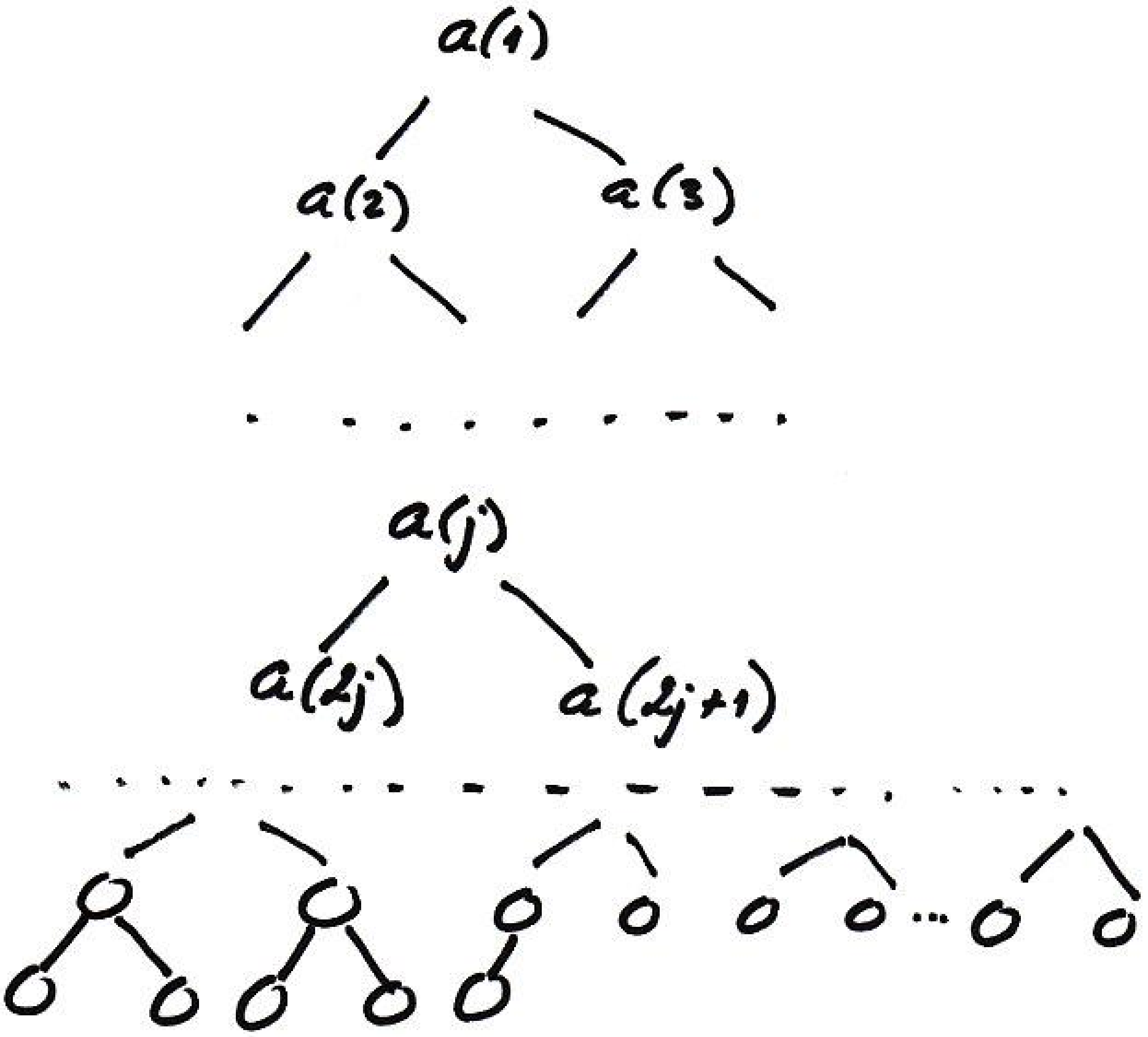
HEAPSORT

Williams 1964

binární min-halda reprezentovaná polem



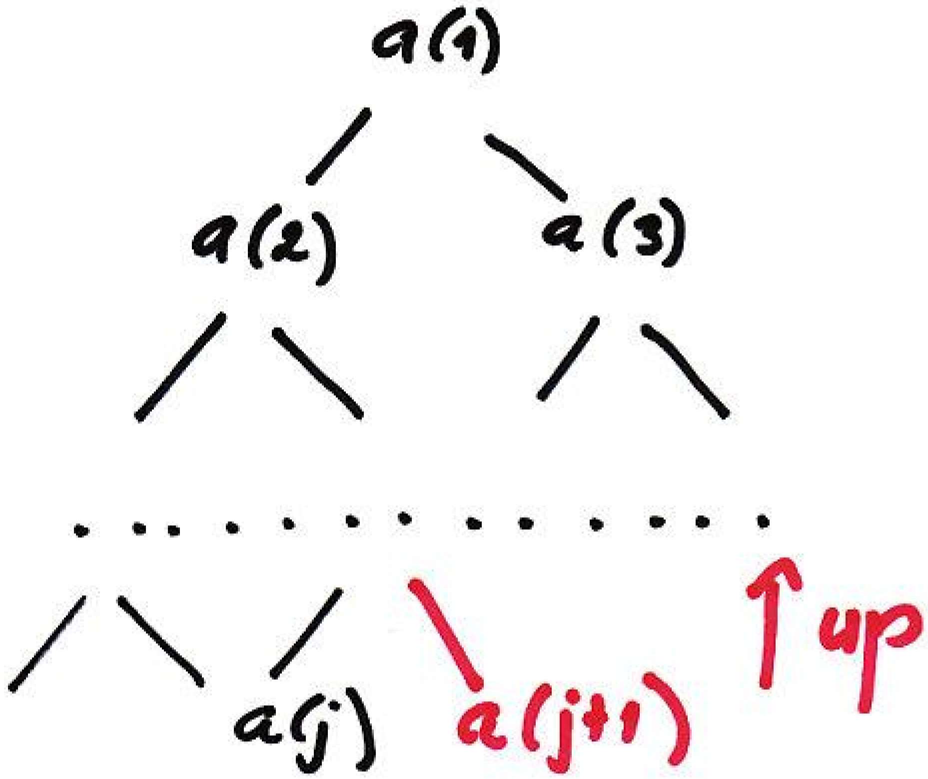
$a(j) < a(2j)$

$a(j) < a(2j+1)$

1) <u>výstavba haldy</u> – postupnými Inserty zdola

<u>for</u> j=1 <u>to</u> n-1 <u>do</u> Insert (A, j, j+1) <u>enddo</u>

Insert:



složitost: $n \log n$ porovnání i výměn

2) <u>třídění</u> – $n \times$ Deletemin

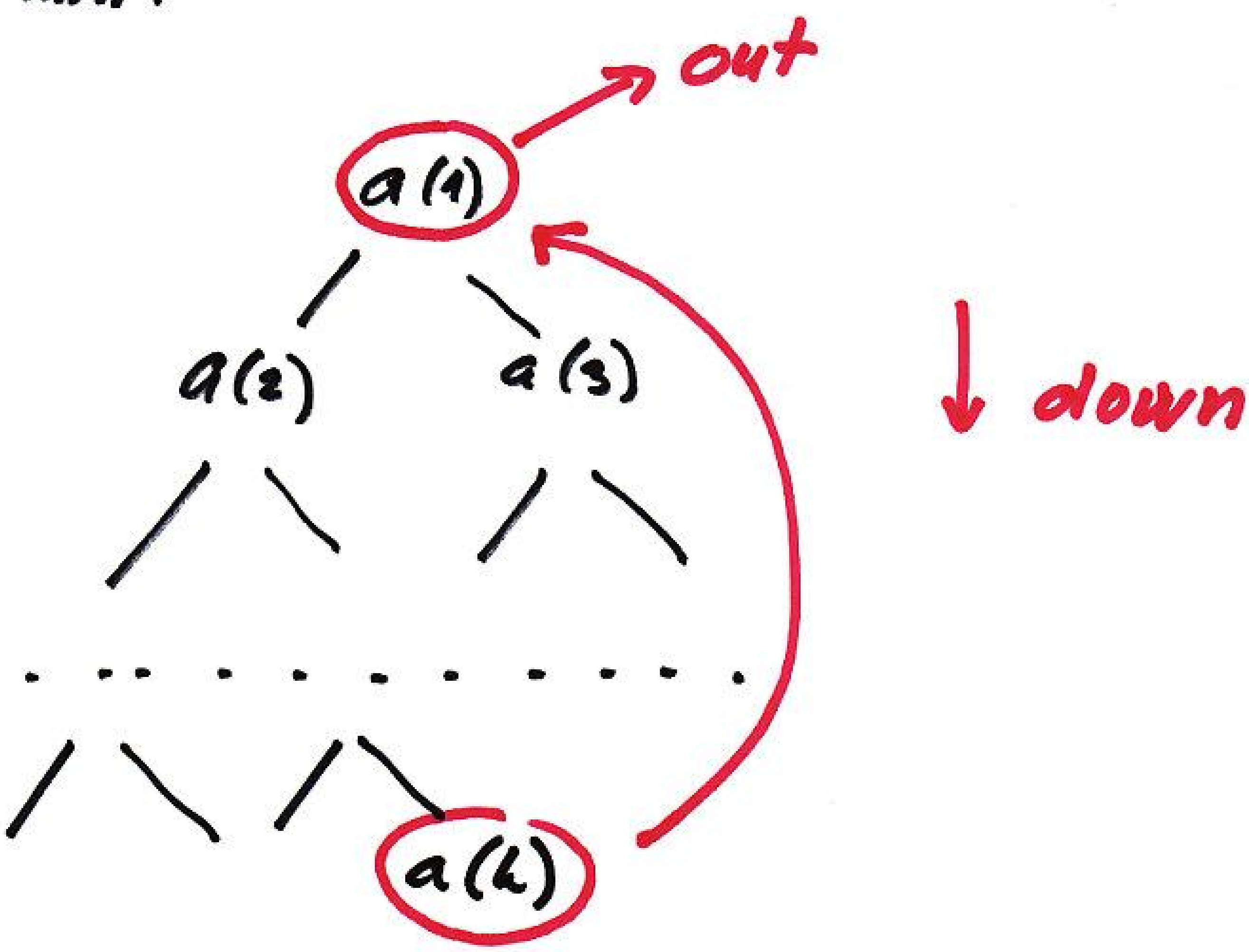
<u>for</u> k:= n <u>downto</u> 1 <u>do</u>

out := a(1), a(1):=a(k)

uprav haldu

<u>enddo</u>

Deletemin:



složitost: $2n \log n$ porovnání

$n \log n$ výměn

celkem Heapsort: $3n \log$ porovnání

$2n \log n$ výměn

(v nejhorším případě)

průměrný případ - výstavba haldy: $O(n)$

netřídí na místě

Floyd (1964) - Treesort 3

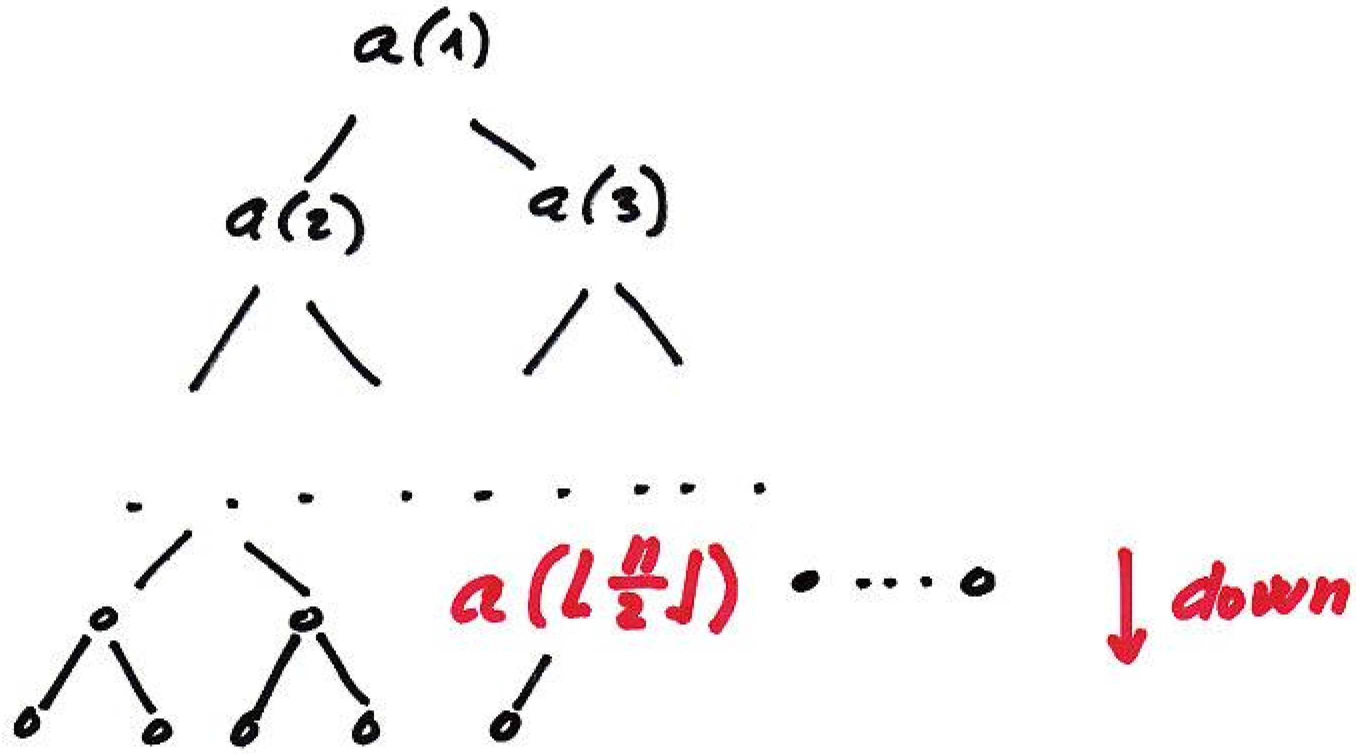
binární max-halda reprezentovaná polem

1) výstavba haldy - "propadáním"

for $i := \lfloor \frac{n}{2} \rfloor$ **downto** 1 **do**

  down $a(i)$

enddo



složitost: $O(n)$

Dk:

v hloubce i je 2^i prvků

padají maximálně o $k-1-i$

$k = \lfloor \log n \rfloor + 1$ je výška haldy

počet porovnání:

$$\sum_{i=0}^{k-2} 2(k-1-i)2^i = 2(k-1)\sum_{i=0}^{k-2} 2^i - 2\sum_{i=0}^{k-2} i2^i =$$

$$= 2^{k+1} - 2(k+1) \sim 2n - 2\log n$$

počet výměn - poloviční

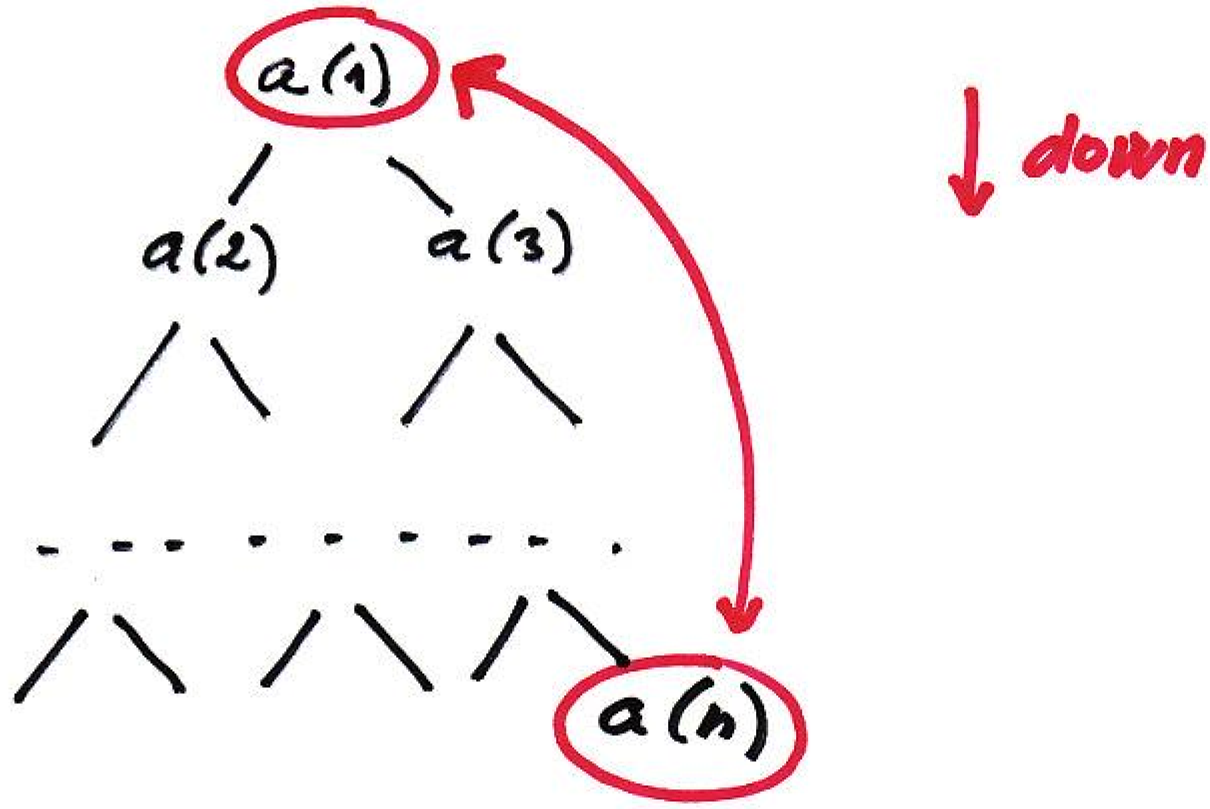
2) třídění – n x Deletemax

for k:=n downto 2 do

vyměň a(1) s a(k)

uprav haldu

enddo



Složitost: $2n \log n$ porovnání

$n \log n$ výměn

celkem: $2n \log n + O(n)$ porovnání

$n \log n + O(n)$ výměn

třídí na místě

46

porovnání, když je v heapsortu použita t-nární halda. Nejprve předpokládejte přímé zobecnění programu 9.7 a pak předpokládejme, že Floydova metoda může ve smyčce ušetřit jednu operaci.

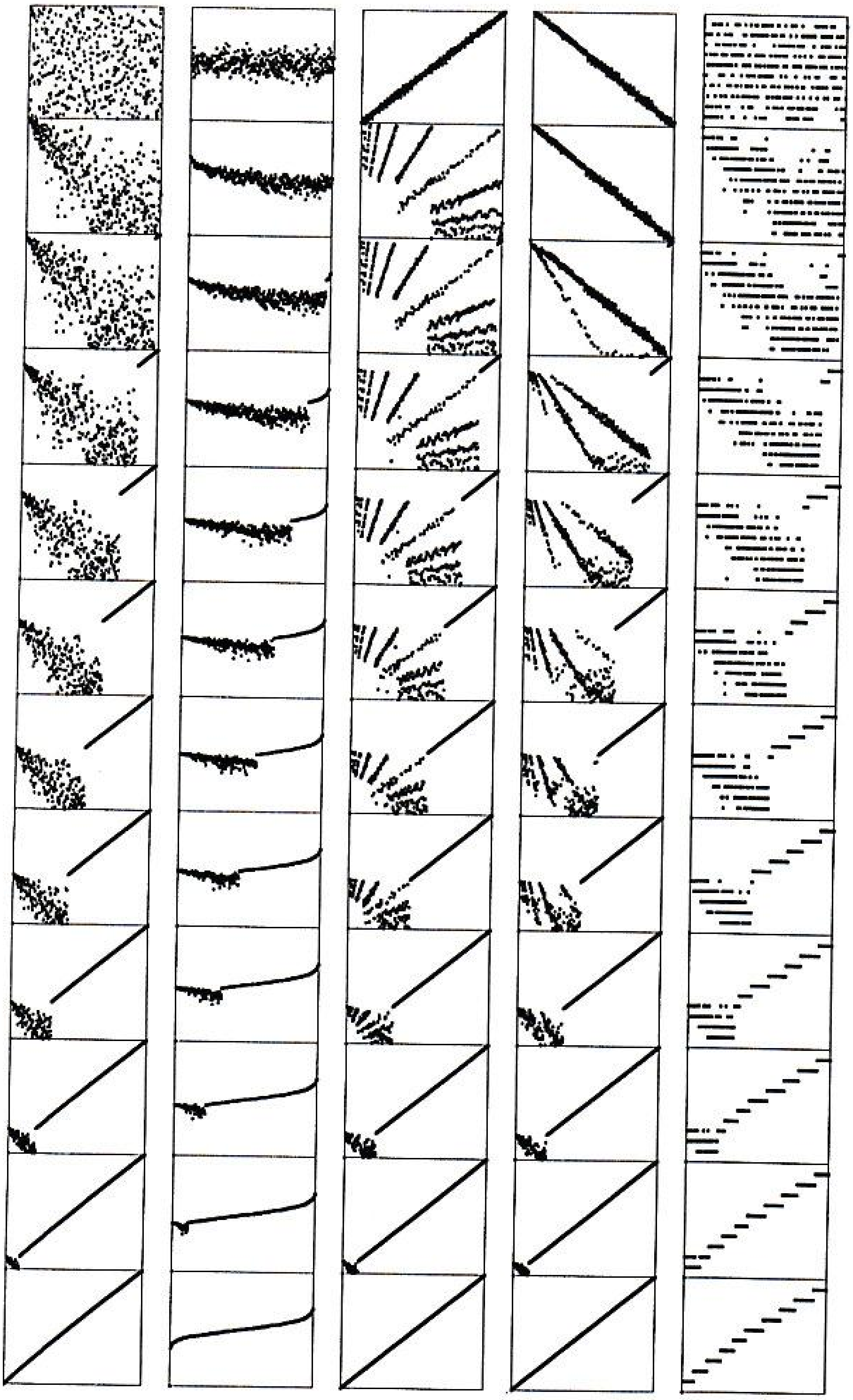


Diagram 9.12 Dynamické charakteristiky heapsortu pro různé typy souborů

Doba běhu heapsortu není nijak mimořádně citlivá na vstup. Bez ohledu na to, jaké jsou vstupní hodnoty, je největší prvek nalezen během méně než lg N *kroků. Horní diagramy ukazují soubory, jež jsou náhodné, gaussiánské, téměř uspořádané, téměř reverzně uspořádané a náhodně uspořádané (nahoře, zleva doprava). Druhý diagram seshora ukazuje vytvoření haldy algoritmem zdola nahoru a zbývající diagramy ukazují pro každý soubor proces setřiďování. Haldy tak či onak ještě odrážejí původní soubor, ale jak proces pokračuje, stávají spíše haldami pro náhodný soubor*

o 9.33 Pro $N = 32$ udejte uspořádání klíčů, jež přiměje heapsort použít tolik porovnání, kolik jen je možné.

Carlsson (1987) - bottom-up heapsort

Deletemax:

odstraň kořen, místo nech prázdné
<u>while</u> prázdné místo není v poslední hladině <u>do</u>
zaplň prázdné místo větším synem
<u>enddo</u>
prázdné místo := poslední prvek
<u>up</u> prázdné místo
poslední prvek := odstraněný kořen

složitost: log n porovnání a výměn směrem dolů
log n — " — vzhůru

celkem 2 log n porovnání i výměn

<u>třídění</u> - n× Deletemax

složitost: 2n log n porovnání i výměn

bottom-up binary insertion heapsort

speciální cesta – cesta z kořene do listu po větších synech

speciální list – list, kde končí spec. cesta

Deletemax:

odstraň kořen

najdi speciální list

najdi správnou pozici pro poslední prvek na spec. cestě ze spec. listu do kořene binárním vyhledáváním

odsuň prvky od správné pozice nahoru o 1 místo směrem ke kořeni

správná pozice := poslední prvek

poslední prvek := odstraněný kořen

složitost : $\log n$ porovnání směrem dolů

$\log\log n$ — " — vzhůru

$\log n$ výměn

celkem na heapsort :

$n(\log n + \log\log n)$ porovnání

$n \log n$ výměn

Tento způsob úpravy haldy se použije i při výstavbě haldy "propadáním".
Složitost : $O(n)$

<u>Wegener (1990)</u> - další verze této metody

bez binárního vyhledávání

$1{,}5\, n \log n + O(n)$ porovnání

Xunrang, Yuzhang (1990) - new heapsort

Deletemax:

odstraň kořen

prázdné místo nech propadnout do $2/3$ výšky haldy

prázdné místo := poslední prvek

buď up prázdné místo

nebo down prázdné místo

poslední prvek := odstraněný kořen

počet porovnání:

$$\frac{2}{3}\log n + \begin{cases} \frac{2}{3}\log n & \text{když up} \\ 2\cdot\frac{1}{3}\log n & \text{když down} \end{cases} = \frac{4}{3}\log n$$

počet výměn (dělají se zároveň s porovnáními):

$$\frac{2}{3}\log n + \begin{cases} \frac{2}{3}\log n = \frac{4}{3}\log n & \text{když up} \\ \frac{1}{3}\log n = \log n & \text{když down} \end{cases}$$

volba $2/3$ je optimální

Gonnet, Munro (1986) – heapsort s rekurzí

Deletemax:

odstraň kořen

$r = \lceil \log n - \log\log n \rceil$

prázdné místo nech propadnout do hloubky r

prázdné místo := poslední prvek

buď up prázdné místo s binárním vyhled.

nebo rekurzivně Deletemax na haldu, jejímž kořenem je prázdné místo

poslední prvek := odstraněný kořen

složitost (porovnání): $\log n + \log^* n$

iterovaný logaritmus : $\log^* n = 0$ pro $n \leq 1$

$\log^* n = 1 + \log^*(\log n)$ pro $n > 1$

celkem na heapsort : $n \log n + n \log^* n$ $(+ O(n))$

Dk: h = výška haldy

k = hloubka, do které se prázdné místo necha propadnout

platí:

$$T(h) = \begin{cases} 2 & \text{pro } h = 1 \\ \max(k + \log k,\ k + T(h-k) + 1) & \text{pro } h > 1 \end{cases}$$

$$k = h - \log h$$

$$T(h) = \max(h,\ h - \log h + T(\log h) + 1) =$$

$$= h - \log h + (\log h - \log\log h + T(\log\log h) + 1) + 1 =$$

$$= h - \log\log h + T(\log\log h) + 1 + 1 =$$

$$= h - \log\log h + (\log\log h - \log\log\log h + T(\log\log\log h) + 1) + 1 + 1 =$$

$$= h - \log\log\log h + T(\log\log\log h) + 1 + 1 + 1 =$$

$$\ldots = h + \log^* h$$

Jiná volba: $k = \frac{h}{2}$, lineární vyhledávání

$$T(h) = \begin{cases} 2 & \text{pro } h = 1 \\ \max(2k,\ k + T(h-k) + 1) & \text{pro } h > 1 \end{cases}$$

$$T(h) = \max\left(h,\ \frac{h}{2} + T\left(\frac{h}{2}\right) + 1\right) =$$

$$= \frac{h}{2} + \left(\frac{h}{4} + T\left(\frac{h}{4}\right) + 1\right) + 1 =$$

$$= \frac{h}{2} + \frac{h}{4} + \left(\frac{h}{8} + T\left(\frac{h}{8}\right) + 1\right) + 1 + 1 =$$

$$\ldots = h \sum_{k=1}^{\log h} \left(\frac{1}{2}\right)^k + \log h \cdot 1 =$$

$$= h\left(1 - \frac{1}{h}\right) + \log h = h - 1 + \log h =$$

$$= \log n + \log\log n$$

stejně rychlé jako bottom-up heapsort s binárním vyhledáváním

McDiarmid, Reed (1989) - MDR - heapsort

používá pomocné bitové pole s hodnotami

$b(j)$ = UNKNOWN (vztah mezi syny vrcholu $a(j)$ je neznámý)

= LEFT (větší je levý syn)

= RIGHT (větší je pravý syn)

drasticky snižuje počet porovnání

složitost: $(n+1)\log n + O(n)$ porovnání

Cantone, Cincotti (2000) - Quick Heapsort

a) vybere se pivot M

b) pole se rozdělí na prvky > M (levá část)

< M (pravá část)

c) vezme se menší z obou částí

je-li to levá část, provede se na ní

Heapsort s max-haldou

odebírané prvky se umisťují (vyměňují)

na konec pravé části

(pravá část analogicky)

d) pivot se umístí na správné místo

e) nesetříděný zbytek pole se třídí rekurzivně

$$\underbrace{\cdots\cdots}_{>M}\ \underset{M}{|}\ \underbrace{\cdots\cdots\cdots\cdots}_{<M}$$ rozdělení

$$\underbrace{\cdots\cdots\cdots\cdots}_{\substack{<M\\ \text{nesetříděné}}}\ \underset{M}{|}\ \underbrace{\cdots\cdots}_{>M\ \text{setříděné}}$$

Dutton (1993) – weak heapsort

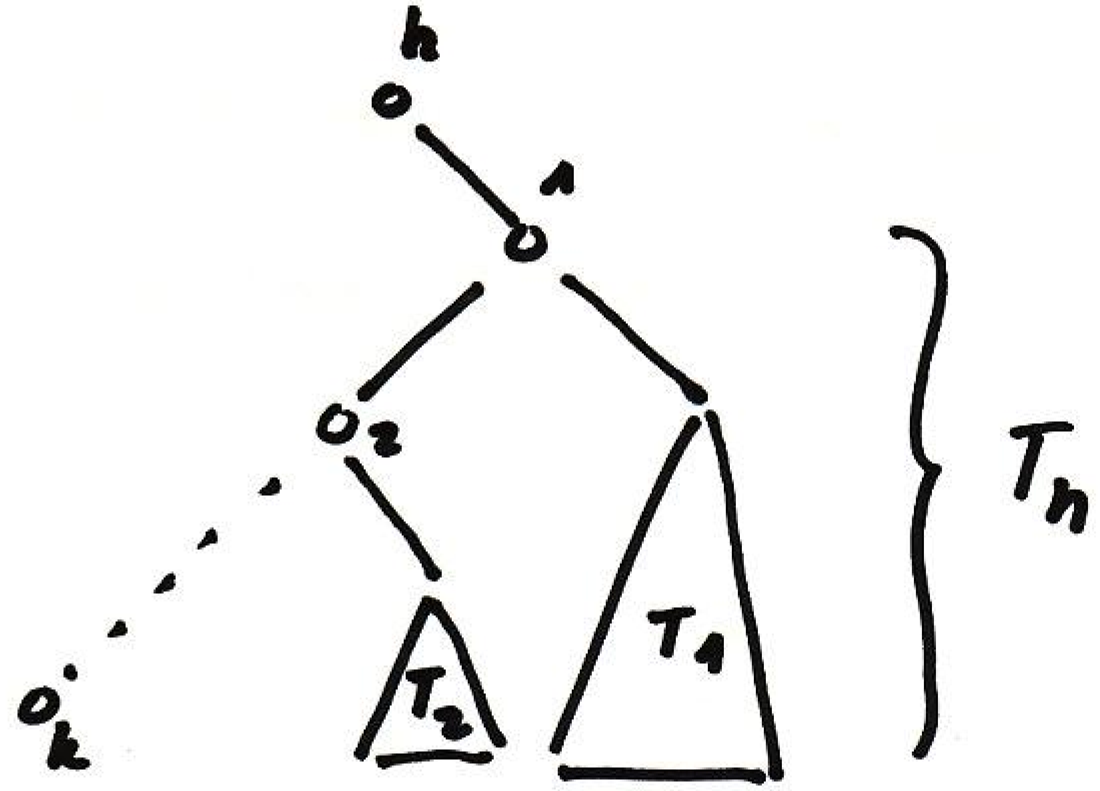
weak – halda:

1) hodnota v jakémkoli uzlu je větší než hodnota v jakémkoli uzlu jeho pravého podstromu

2) kořen nemá levý podstrom

3) uzly s méně než 2 potomky jsou v posledních dvou hladinách

rekurzivní struktura:



princip algoritmu:

1) postavit haldu

2) odstranit kořen

T_n se rozpadne na les podhald $T_1, \dots, T_k$

zmergují se do nové weak-haldy tak, že

max. z vrcholů $1, \dots, k$ bude kořen

formálně je to hodně komplikované

teoreticky velmi rychlé

Další experimentální výsledky

P. D. Dutton: Weak-heapsort, BIT 33 (1993), 372-381

valued approximation is similar to that for the upper bound given in the proof of Theorem 4.1.

Simulation results are presented in the Table. For each n, identical sets of 20 randomly generated distinct values were given to each sorting routine. The second column, WHS, is the average number of comparisons required by WeakHeapSort. The third column, QSORT, reflects those of an implementation of Quicksort [5] which partitions with the middle of three randomly selected values. The fourth column, BUHS, is the data reported for Bottom-Up heapsort from [1]: the last column, MDRS, is the result of an implementation of Wegener's version of Bottom-Up heapsort [10]. The figures in parenthesis are the expected number of compares for each instance. For example, with WeakHeapSort, this is approximated by an upper bound of the average of the best and worst case number of compares as given above, i.e., $(n - 0.5)\log n - 0.413n$.

Table. *Simulated numbers of compares.*

n	WHS	QSORT	BUHS	MDRS
10	26(27)	28(23)	32(38)	31(33)
50	254(258)	260(231)	293(300)	283(284)
100	614(619)	624(574)	696(701)	669(669)
500	4247(4271)	4543(4211)	4650(4669)	4507(4506)
1000	9497(9547)	10142(9598)	10312(10338)	10032(10013)
5000	59222(59367)	64207(61731)	63309(63303)	61862(61678)
10000	128465(128740)	140570(135326)	136648(136607)	133719(133352)
50000	758835(759824)	850962(814483)	799741(799132)	785982(782857)

In [9], Wegener presents a formula (attributed to Kemp) resulting from an analysis of the expected number of compares for the version of Quicksort used here, that is approximately $1.188n\log(n-1) - 2.255n + 1.188\log(n-1) + 2.507$. Our simulation, column QSORT, exceeds the formula values (given in parenthesis) consistently by 4–5%. Even then, when $n > 500$, the formula values are still greater than those observed for WeakHeapSort. The expected values for columns BUHS and MDRS were derived with the formulas $n\log n + 0.373n$ and $n\log n + 0.0475n$, respectively, and arise from using the midpoints of the expected ranges for $f(n)$ given in Section 1.

WeakHeapSort seems always to achieve the best case number of compares, $n\lceil\log n\rceil - 2^{\lceil\log n\rceil} + 1 < \log n - 0.913986n$, when the data are initially in ascending order. The reason for this is not yet understood, but one should not conclude that preprocessing the data will be beneficial since the difference between the best and worst case number of comparisons is at most $n - \lceil\log n\rceil - 1$. It is difficult to imagine preprocessing that would use fewer compares than this difference.

Edelkamp, Stiegeler: Pushing the Limits in Sequential Sorting 59

WAE 2000, LNCS, Springer 2000

by using one bit to encode on which branch the smaller element can be found and another one to mark if this information is unknown.

Table 1 presents data of our c++-implementation of various performant sorting algorithms on some random sets of floting-point data. We have referenced the number of data comparisons, transpositions and assignments.

Comparisons	10	100	1000	10000	100000	1000000
QUICKSORT	30.56	729.10	11835.10	161168.63	2078845.53	25632231.16
CLEVER-QUICKSORT	28.86	649.36	10335.83	143148.86	1826008.56	22290868.50
BOTTOM-UP-HEAPSORT	32.43	694.76	10305.10	136513.60	1698272.06	20281364.60
MDR-HEAPSORT	30.76	656.43	9886.83	132316.06	1656365.03	19863064.60
WEAK-HEAPSORT	27.60	618.63	9513.40	128568.30	1618689.90	19487763.03
RELAXED-WEAK-HEAPSORT	25.00	573.00	8977.00	123617.00	1568929.00	18951425.00
GREEDY-WEAK-HEAPSORT	25.00	573.00	8977.00	123617.00	1568929.00	18951425.00
QUICK-HEAPSORT	29.66	781.83	11630.00	150083.96	1827962.60	22000593.93
QUICK-WEAK-HEAPSORT	28.43	660.90	10045.76	133404.13	1659777.23	20266957.30
CLEVER-HEAPSORT	29.70	727.73	11322.86	148107.30	1819264.86	21416210.80
CLEVER-WEAK-HEAPSORT	27.86	609.23	9418.96	128164.83	1614333.26	19602152.26

Exchanges	10	100	1000	10000	100000	1000000
QUICKSORT	9.36	161.00	2369.70	31501.43	391003.10	4668411.06
CLEVER-QUICKSORT	8.53	165.33	2434.03	32294.80	401730.13	4803056.93
BOTTOM-UP-HEAPSORT	9.00	99.00	999.00	9999.00	99999.00	999999.00
MDR-HEAPSORT	9.00	99.00	999.00	9999.00	99999.00	999999.00
WEAK-HEAPSORT	15.50	336.66	4969.70	65037.53	803705.33	9578990.20
RELAXED-WEAK-HEAPSORT	22.76	412.06	5800.13	75001.40	915744.06	10790844.36
GREEDY-WEAK-HEAPSORT	20.00	357.30	5187.33	69255.83	856861.66	10177597.60
QUICK-HEAPSORT	7.63	72.20	670.80	6727.60	65321.76	696398.30
QUICK-WEAK-HEAPSORT	13.46	313.00	4990.66	66929.90	838173.16	9985011.30
CLEVER-HEAPSORT	7.46	70.70	661.00	6478.16	64919.86	652938.26
CLEVER-WEAK-HEAPSORT	14.53	326.66	5071.36	67578.40	842446.46	10066407.13

Assignments	10	100	1000	10000	100000	1000000
QUICKSORT	6.33	66.00	666.70	6670.86	66651.43	666649.50
CLEVER-QUICKSORT	4.33	46.26	462.20	4571.30	45745.93	457071.86
BOTTOM-UP-HEAPSORT	43.86	781.66	11116.30	144606.73	1779038.73	21088698.53
MDR-HEAPSORT	43.50	769.00	10965.56	142759.63	1758266.06	20856586.16
WEAK-HEAPSORT	11.00	101.00	1001.00	10001.00	100001.00	1000001.00
RELAXED-WEAK-HEAPSORT	0.00	0.00	0.00	0.00	0.00	0.00
GREEDY-WEAK-HEAPSORT	5.20	126.80	1288.40	11649.73	118028.26	1226882.80
QUICK-HEAPSORT	32.93	693.70	10752.93	141534.50	1757454.76	20697840.96
QUICK-WEAK-HEAPSORT	4.46	11.46	18.90	25.30	33.26	42.76
CLEVER-HEAPSORT	30.16	716.70	10827.53	142134.60	1759835.10	20892687.56
CLEVER-WEAK-HEAPSORT	2.96	8.50	14.36	20.70	26.16	32.93

Table 1. Number of comparisons, exchanges, assignments of the different algorithms on random data averaged over 30 runs.

60

Cantone, Cincotti:

	Integer			Double			cmp_1 (Double)		
.	$t_c = 0.05\mu sec$, $t_m = 0.06\mu sec$			$t_c = 0.05\mu sec$, $t_m = 0.07\mu sec$			$t_c = 0.3\mu sec$, $t_m = 0.07\mu sec$		
n=	10^4	10^5	10^6	10^4	10^5	10^6	10^4	10^5	10^6
H	0.05	0.75	12.37	0.10	1.40	20.35	0.14	1.96	27.28
BU	0.09	1.25	18.80	0.13	1.83	25.88	0.15	2.00	27.62
i-Q	0.04	0.40	4.81	0.07	0.80	9.74	0.10	1.25	15.10
Q	0.03	0.37	4.54	0.06	0.74	8.92	0.09	1.12	13.49
c-Q	$\boxed{0.03}$	$\boxed{0.33}$	$\boxed{4.08}$	$\boxed{0.05}$	$\boxed{0.69}$	$\boxed{8.34}$	$\boxed{0.08}$	$\boxed{0.99}$	$\boxed{11.99}$
QH	0.05	0.64	10.43	0.08	1.10	16.85	0.10	1.42	19.96
c-QH	0.04	0.65	10.48	0.08	1.11	16.92	0.10	1.38	20.05

	cmp_1 (Integer)			cmp_2 (Integer)			cmp_3 (Integer)		
	$t_c = 0.19\mu sec$, $t_m = 0.06\mu sec$			$t_c = 2.9\mu sec$, $t_m = 0.06\mu sec$			$t_c = 4.7\mu sec$, $t_m = 0.06\mu sec$		
n=	10^4	10^5	10^6	10^4	10^5	10^6	10^4	10^5	10^6
H	0.10	1.35	19.16	0.54	7.04	88.29	1.00	12.92	159.66
BU	0.11	1.52	21.23	0.37	4.72	58.94	0.64	8.09	98.42
i-Q	0.07	0.85	10.12	0.42	5.44	64.96	0.79	10.24	120.69
Q	0.07	0.80	9.60	0.40	4.93	59.53	0.74	9.20	110.29
c-Q	$\boxed{0.05}$	$\boxed{0.68}$	$\boxed{8.30}$	0.35	4.42	53.82	0.64	8.22	99.24
QH	0.08	0.99	13.99	0.37	4.57	54.57	0.66	8.17	95.11
c-QH	0.07	0.98	13.98	$\boxed{0.34}$	$\boxed{4.22}$	$\boxed{52.15}$	$\boxed{0.61}$	$\boxed{7.63}$	$\boxed{91.58}$

Table 2. Average running times in seconds (sample size = 10).

Cache performance has considerably less influence on the behaviour of sorting algorithms than does paging performance (cf. [14], Chap. 8); for such reason, we believe that we can ignore completely possible negative effects due to caching.

Concerning virtual memory problems, i.e. demand paging, all Quicksort algorithms show good locality of reference, whereas Heapsort algorithms, and also QuickHeapsort algorithms, tend to use pages that contain the top of the heap heavily, and to use in a random manner pages that contain the bottom of the heap (cf. [14]). Such observation allows us to conclude that an execution of c-Q cannot be more penalized than an execution of c-QH by delays due to paging problems. Hence, we can reasonably conclude that the success of c-QH is not due to paging performance.

5. Conclusions

We presented QuickHeapsort, a new practical "in-place" sorting algorithm obtained by merging some characteristics of Bottom-Up-Heapsort and Quicksort. Both theoretical analysis and experimental tests confirm the merits of QuickHeapsort.

The experimental results obtained show that it is convenient to use clever-QuickHeapsort when the input size n is large enough and each key comparison operation is computationally expensive.

t_c ... čas na 1 porovnání

t_m ... čas na 1 výměnu (move)

5.2 Srovnání výsledků hlavní implementace

Řazení algoritmů v grafech odpovídá pořadí, v jakém jsou algoritmy uvedeny v kap. 3.

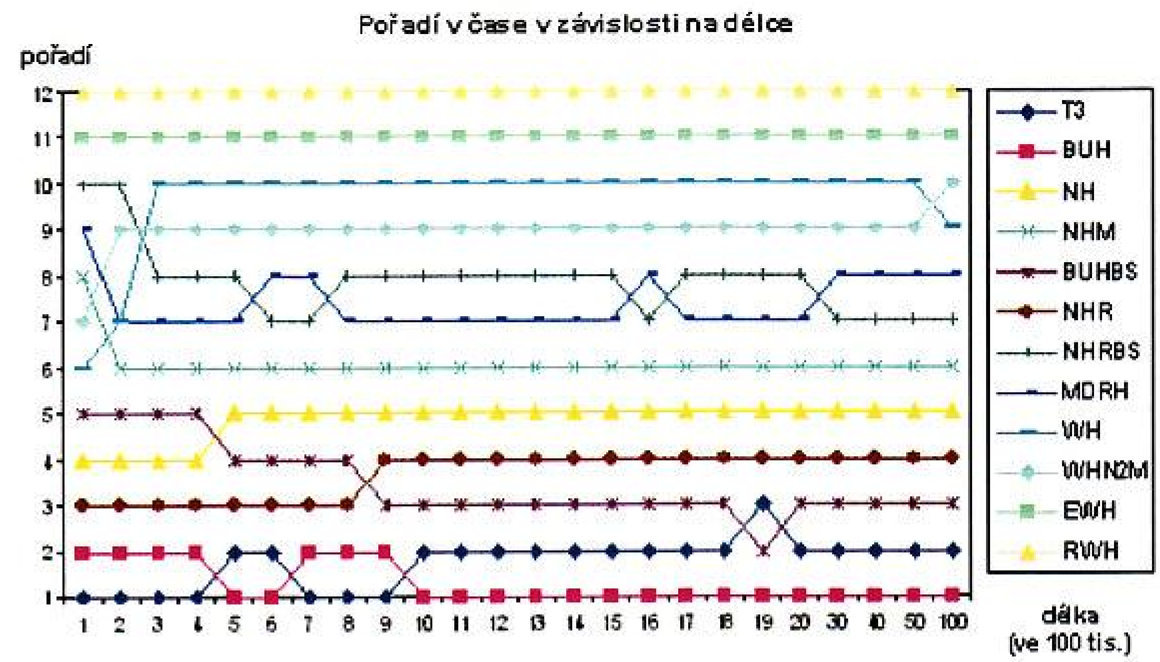
5.2.1 Porovnání algoritmů

Tabulka 5.2 zobrazuje pořadí jednotlivých algoritmů. Pořadí v čase je výsledkem porovnání pořadí podle průměrných časů algoritmů dosažených v každé z testovaných délek.

algoritmus	čas	porovnání
T3	2.	12.
BUH	1.	6.
NH	5.	11.
NHM	6.	7.
BUHBS	3.	8.
NHR	4.	9.
NHRBS	8.	10.
MDRH	7.	5.
WH	10.	4.
WHN2M	9.	3.
EWH	11.	1.
RWH	12.	1.

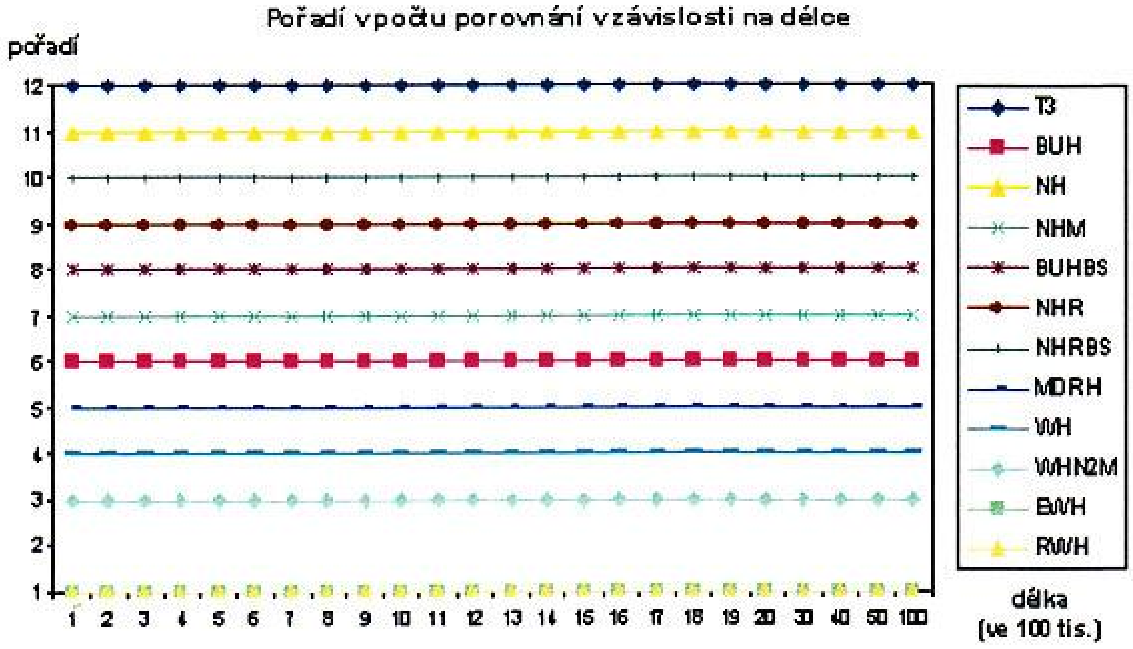
Tabulka 5.2: Pořadí v dosažených časech a počtech porovnání

Z grafu 5.5 vidíme, že pořadí není neměnné, ale mění se v závislosti na délce testovaných posloupností.



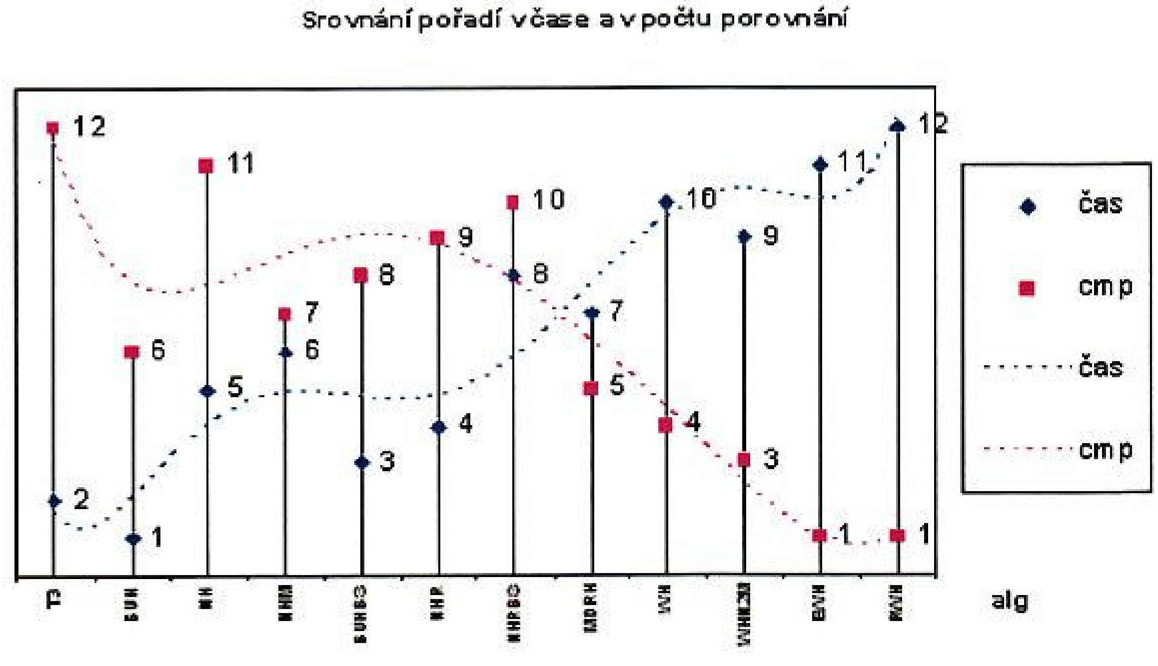
Obrázek 5.5: Pořadí v průměrném čase v závislosti na délce

Jiná je situace u pořadí daného počtem porovnání, jak ho ukazuje graf 5.6. Pořadí je neměnné, nejnižšího počtu porovnání dosahují algoritmy *EWH* a *RWH* (mají stejný počet porovnání), nejvyšší počet porovnání má *T3*.



Obrázek 5.6: Pořadí v průměrném počtu porovnání v závislosti na délce

Zajímavé je srovnání obou grafů, tedy porovnání pořadí každého z algoritmů z grafu 5.5 s pořadím z grafu 5.6. Srovnání vidíme na grafu 5.7. Pro názornost jsou hodnotami obou veličin proloženy křivky (polynomy šestého stupně). Pořadí v dosaženém čase je (pro každý algoritmus zvlášť) průměrem pořadí přes všechny testované délky posloupností. Z grafu můžeme vyvodit několik pozorování.

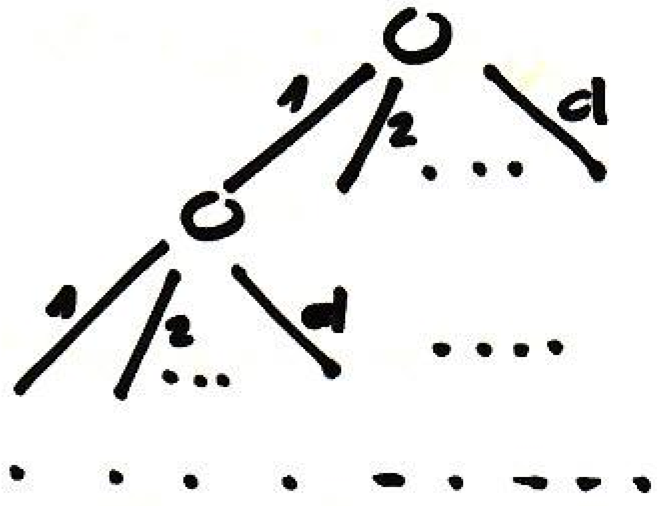


Obrázek 5.7: Srovnání pořadí v čase a v počtu porovnání

Za prvé vyniká kontrast u algoritmu *T3* – ačkoliv tento algoritmus potřebuje při třídění zdaleka nejvíce porovnání, patří zároveň k nejrychlejším. Tento algo-

Heapsort na víceregulárních haldách

d = stupeň regularity



Johnson (1975): 3 a 4-regulární haldy jsou ve všech haldových operacích rychlejší než binární

Luk (1999):

složitost Heapsortu $T(n) \le \dfrac{(c_1 d + c_2)\, n \ln n}{\ln d}$

c_1 = cena porovnání, c_2 = cena výměny

$d = 3$ je optimální, když $c_2 = 0$

$d = 4$ — " — $c_1 = c_2$

použití: externí třídění

třídění v hierarchické paměti

Očekávaná složitost

očekávaná složitost Insertu je $O(1)$

2,607 porovnání

posun o 1,607 hladin

Dobekort (1982):

očekávaná složitost Williams-Floydova heapsortu je stejná jako nejhorší případ (včetně konstant)

Heapsorty v průměru rychlejší než Quicksort

bottom-up heapsort

MDR-heapsort

Quick-heapsort

Weak-heapsort

týká se počtu porovnání

vyplatí se, když cena porovnání je vysoká

Složitost v nejlepším případě

Larin (1975):

hypotéza, že nejlepší případ může být $O(n)$

Wirth (1976):

nejlepší jsou posloupnosti setříděné v opačném pořadí?

skutečnost:

$O(n)$ pouze pro posloupnost, která má všechny prvky stejné

jinak $O(n \log n)$ – asi 2x rychlejší než nejhorší případ

Ding, Weiss (1992)

Bollobás, Fenner, Frieze (1996)

Heapsort není adaptivní na předtříděné posloupnosti