sorting

used in many different settings
also artistic impressions, even dance

puzzle: can we sort 5 elements in 7 comparisons?

overview

• heap sort: more
• heapsort
• priority queues
• quicksort
• material
crucial procedures for heapsort

maxHeapify (or downMaxHeap)

buildMaxHeap

heapSort

maxHeapify or downMaxHeap: pseudo code

**Algorithm** maxHeapify(A, i):

\[ l := \text{left}(i) \]
\[ r := \text{right}(i) \]

\[ \text{if } l \leq A.\text{heap-size and } A[l] > A[i] \text{ then} \]
\[ \quad \text{largest} := l \]
\[ \text{else} \]
\[ \quad \text{largest} := i \]
\[ \text{if } r \leq A.\text{heap-size and } A[r] > A[\text{largest}] \text{ then} \]
\[ \quad \text{largest} := r \]
\[ \text{if } \text{largest} \neq i \text{ then} \]
\[ \quad \text{swap}(A[i], A[\text{largest}]) \]
\[ \quad \text{maxHeapify}(A, \text{largest}) \]

maxHeapify: correctness

induction on the height of node \( i \)
if the height is 0, then immediate
if the height is > 0, then two cases

- case \( \text{largest} = i \) immediate
- case \( \text{largest} = l \) (and equivalent for \( \text{largest} = r \)) use induction

maxHeapify: time complexity

with \( h \) the height of the heap:

\[ T(h) = T(h - 1) + 1 \text{ if } h > 0 \text{ gives } T(h) \in \mathcal{O}(h) \]

then use \( h \in \Theta(\log n) \)

with \( n \) the number of nodes of the heap:

\[ T(n) = T\left(\frac{2}{3}n\right) + 1 \text{ if } n > 1 \text{ gives } T(n) \in \mathcal{O}(\log n) \]

because in the worst case the bottom level is exactly half full
Algorithm buildMaxHeap($H$):

$H$.size := $H$.length

for $i = \lceil H\.length/2 \rceil$ downto 1 do

downMaxHeap($H$, $i$)

buildMaxHeap: correctness

use loop invariant:

at the start of the for-loop each node $i + 1, \ldots, n$ is the root of a max-heap

init:

for $i = \lceil n/2 \rceil$ the nodes $i + 1, \ldots, n$ are leaves

loop:

children are max-heaps

use correctness of maxHeapify

end:

for $i = 0$ the invariant gives correctness of the output

buildMaxHeap: complexity

rough estimation:

for each of the $\lceil n/2 \rceil$ internal nodes of the heap
we do maxHeapify which is in $O(h)$
so buildMaxHeap in $O(n \cdot \log n)$

more precise estimation:

for every height $j$ in $0, \ldots, \lceil \log n \rceil$
for each of the at most $\lceil n/2^j \rceil$ nodes of height $j$
we do downMaxHeap which is in $O(j)$
so (!) buildMaxHeap is in $O(n)$

buildMaxHeap: alternative algorithm?

from 1 to $\lceil n/2 \rceil$ ?

try $[1, 3, 2, 4, 5, 6, 7]$ with $i = 1, 2, 3$
heapSort

**Algorithm heapSort(H):**

buildMaxHeap(H)

for \( i = H.length \) downto 2 do

swap \( H[1] \) and \( H[i] \)

\( H.\text{heap-size} := H.\text{heap-size} - 1 \)

downMaxHeap(H, 1)

heapSort: analysis

running time in \( O(n \log n) \)

what happens to a sorted input?

what is the best-case?

inspired by heapSort: smooth sort

SmoothSort, an alternative for sorting in situ

by Edsger W. Dijkstra,

Burroughs Corporation

Abstract

Like heapSort --- which inspired it --- smoothsort is an algorithm for sorting in situ. It is of order \( N \cdot \log N \) in the worst case, but of order \( N \) in the best case, with a smooth transition between the two. (Hence its name.)

Key Words and Phrases: sorting in situ, heapSort, sorting trees, sift, computational complexity.

CR Categories: 5.25, 5.31

smooth sort: inventor

Edsger W. Dijkstra 1930–2002

Turing Award 1972
overview

- heap sort: more
- heapsort
- priority queues
- quicksort
- material

priority queue

data type for maintaining a set
access via keys
queue where most important element is served first
most important: minimum key or maximum key

priority queue: abstract data type

**insert**\((Q, x)\) inserts element \(x\) in the (max- or min-)priority queue \(Q\)

for max-priority queue:

**extractMax**\((Q)\) removes element with maximum key

**maximum**\((Q)\) returns (but does not remove) element with max key

priority queue: implementation

use a heap

max-heap for max-priority queue

min-heap for min-priority queue
maximum: pseudo-code

\[ H \text{ a max-heap; return the max} \]

\[ \textbf{Algorithm} \text{ maximum}(H): \]
\[ \text{return } H[1] \]

remove: pseudo-code

\[ H \text{ max-heap; remove and return maximum; error omitted} \]

\[ \textbf{Algorithm} \text{ extractMax}(H): \]
\[ \text{max := } H[1] \]
\[ H[1] := H[H.\text{heap-size}] \]
\[ H.\text{heap-size} := H.\text{heap-size} - 1 \]
\[ \text{maxHeapify}(H, 1) \]
\[ \text{return max} \]

running time? in \( O(\log n) \)

insert for max-priority queue: pseudo-code

\[ H \text{ a max-heap; insert key } k; \text{ bubble upwards} \]

\[ \textbf{Algorithm} \text{ insert}(H, k): \]
\[ H.\text{heap-size} := H.\text{heap-size} + 1 \]
\[ H[H.\text{heap-size}] := -\infty \]
\[ \text{HeapIncreaseKey}(H, H.\text{heap-size}, k) \]

example: insert

add key 19 to the max-heap \[ 15, 13, 9, 5, 12, 8, 7, 4, 1, 6 \]
priority queue operations: running time

- **insert** in $O(\log n)$
- **remove** in $O(\log n)$
- **maximum or minimum** in $O(1)$

questions about algorithms

- give the intuition of the algorithm
- give the pseudo-code for the algorithm
- apply the algorithm
- adapt the algorithm
- analyse correctness of the algorithm
- analyse (worst-case) time complexity of the algorithm

overview

- heap sort: more
- heapsort
- priority queues
- quicksort
- material

quicksort

- merge sort is in $O(n \log n)$ but it is not in-place
- quicksort is also a divide-and-conquer sorting algorithm
- it can partly solve the drawback of merge sort
quicksort: idea

divide: partition the input-array $A$ into two parts:
up to index $q$ containing keys less than or equal to pivot
starting index $q + 1$ containing keys larger than pivot
recur: sort recursively the two sub-arrays
conquer: combine the two partial solutions into one sorted array
here there is nothing to do

quicksort: pseudo-code

we sort the (sub-)array $A[p \ldots r]

Algorithm quickSort($A, p, r$):
    if $p < r$ then
        $q :=$ partition($A, p, r$)
        quickSort($A, p, q - 1$)
        quickSort($A, q + 1, r$)

what happens if $p = r$?

partition: idea

while the list contains more than one element:
take a number $k$ from the list, called the pivot
index $i$ indicates the last key of the small-ones-so-far
index $j$ indicates the first key to be compared with the pivot
if $j$ finds a key smaller than pivot
swap that key with the one at $i + 1$

partition: pseudo-code

Algorithm partition($A, p, r$):
    $x := A[r]$
    $i := p - 1$
    for $j = p$ to $r - 1$ do
        if $A[j] \leq x$ then
            $i := i + 1$
            exchange $A[i]$ with $A[j]$
        exchange $A[i + 1]$ with $A[r]$
    return $i + 1$

quickSort calls partition only if $p < r$
running time? $r - p$ steps so in $\mathcal{O}(r - p + i) = \mathcal{O}(n)$
question: apply partition

3 6 1 7 8 2 5 4

question: give recursion tree for quicksort

3 6 1 7 8 2 5 4

sorting: xkcd

property: at the beginning of the for-loop

- if \( p \leq k \leq i \) then \( A[k] \leq \text{pivot} \)
  the part up to (included) \( i \) contains the small-ones-so-far

- if \( i + 1 \leq k \leq j - 1 \) then \( A[k] > \text{pivot} \)
  the ones not selected for swapping by \( j \) are large-ones-so-far

- if \( k = r \) then \( A[k] = \text{pivot} \)
  pivot does not move during iterations of for-loop

partition: correctness
partition: correctness

does the property hold initially?

is the property maintained during the loop?

if so, then it is an invariant! does it yield correctness?

quicksort: correctness

follows from correctness of partition and induction

overview

extra material

- heap sort: more
- heapsort
- priority queues
- quicksort
- material

- smooth sort
- sorting wikipedia