Lectures: László Kozma < lkozma@inf.fu-berlin.de >
Tutorials: Katharina Klost

Contents:
- Selected topics about data structures and applications
- Theory course, M.Sc. level

Course website:
- KVV/Whiteboard
- page.mi.fu-berlin.de/lkozma/ds2020

Organization:
- Lectures: Tue/Thu 10-12
- Tutorials: Wed 14-16

Prerequisites:
- Algorithms/Mathematics (~ HA)
- O-notation, asymptotics
- computational models (TM, RAM model, pointer machine)
- basic data structures (array, list, stack, queue, balanced BST)
- amortised analysis (will recap)
- basic graph algorithms
- ...

Books / references:
- (see course websites)

Tutorials:
- one exercise sheet each week
- deadline ~ 10 days (Tue to next Fri)
- first sheet online, due 1st May (total 11/12)
- details to follow (how to submit, etc.)
- encouraged to work in pairs
- one programming exercise (details later)
- cite all sources, collaborators!

To pass:
- exam (oral/written): to be decided later
- 60% of exercise points, including programming exercise
- active participation in tutorials (details later)
- final grade: exam only.
1. Basic data structures

RAM model of computation

centralised memory

cell can contain integer word on $c \log n$ bits ($n^c$ different integers -- since cells address memory, we can have at most this many memory cells)

- RAM model is an idealised computer
- Each elementary operation/test takes O(1) time
- can implement familiar structures, pointers, arrays, etc.

Implementing an array

1. A = make_array(n,c)
2. read(A,i)
3. write(A,i,v)

solution sketch: use 3 arrays

- let’s try to implement an array on the RAM machine
  (we usually don’t worry about such low-level details)
- support three basic operations:
- n is size of the array, c is initial value (if an array entry has not been written, it has value c)
- difficulty: we cannot make assumption on initial memory contents
- all three operations should take O(1) time

verification:

when we allocate array, we don’t care about memory content

make_array(n,c)
allocate arrays Time, Values, Written of size n
size:=n, current:=0, init:=c

is_written(i)
if (Time[i]<=current) and (Written[Time[i]]==i) return TRUE
else return FALSE

read(i)
if (is_written(i))
return Values[i]
else
return init

write(i,v)
Values[i]:=v
if (is_written(i))
do nothing
else
current:=current+1
Written[current]:=i
Time[i]:=current

Verify correctness and that all three operations take O(1) time.
Implementing a stack with an array

1. S = make_stack()
2. pop(S)
3. push(S,v)

All three operations should take O(1) time

Idea: Allocate an array, keep track of top of the stack
update for each pop/push
Problem: how large should array be?

pop()
  n := n - 1
  output A[n+1]
  if (n == (capacity/4))
    capacity = [capacity/2]

(Note: we don’t reduce capacity even more, because we don’t want to trigger a doubling too soon)

push(v)
  n := n + 1
  A[n] := v
  if (n == capacity)
    allocate A of size 2n
    capacity := 2n
    copy over n items into new array
    free up old array

We maintain the following invariant at all times:

We maintain the following invariant at all times:

\[
\frac{\text{capacity}}{4} \leq n \leq \text{capacity}
\]

Analysis:
the only problem is that push may take time — n
we cannot give O(1) bound on actual cost of push
we do an amortized analysis.

Amortized costs:

- make_stack: 1 unit
- pop: 1 unit
- push: 3 unit

(all three operations have constant amortized cost)

Operations make_stack and pop are fine, since actual cost is also constant, 1 unit can pay for it.

A simple push actual cost is constant, 1 unit can pay for it, 2 units are deposited.

When it comes to the costly push, by Observation, we have already deposited n units.

This pays for the copying (actual cost O(n)).

Exercise: modify design such that actual cost is constant, not just the amortized.
(should still be array-based, not pointer-based).
Pointer-based data structures

A node contains a constant number of fields and pointers.

- Operations: O(1) time

- Modes with a constant number of fields and pointers
- A constant number of global variables
- A root node

Pointer-based DS

- More flexible
- No need for contiguous memory
  - Less efficient
  - Need more space

Pointer machine structures (restriction of RAM model)

- No arithmetic on addresses
- No hashing
Heaps (priority queues)

- store a collection of items
- item $x$ has field $x.key$, from some ordered set (typically integer)

- operations:
  - $H := \text{make\_heap}()$
  - $\text{insert}(H,x)$
  - $\text{extract\_min}(H)$
  - $\text{find\_min}(H)$
  - $\text{delete}(x)$
  - $\text{meld}(H_1,H_2)$
  - $\text{decreasekey}(x,k)$

Assume keys only accessed via comparisons.

Thm. At least one of $\text{extract\_min}$ and $\text{insert}$ must take time $\Omega(\log n)$

Insert $x_1,x_2,x_3,\ldots,x_n$ $\Rightarrow$ Sort $\Rightarrow \Omega(n \log n)$

Heap implementation:

- insert and $\text{extract\_min}$ in $O(\log(n))$ time.
- We need to traverse a path in the (balanced) tree
- Meld is difficult in array-based heaps: $O(n)$

Binomial heaps, Fibonacci heaps, Skew heaps, Hollow heaps, ...

Fibonacci heaps, and variants implement $\text{delete}$ and $\text{extract\_min}$ in $O(\log n)$, other operations in $O(1)$.

Check which bounds are amortized in which data structure.

Heap application: MEDIAN FILTER

Given a sequence $a_1,a_2,a_3,\ldots,a_n$

Replace $a_i$ by the median of $a_{i-k},a_{i-k+1},\ldots,a_i,a_{i+1},\ldots,a_{i+k}$ for all $i = k+1,\ldots,n-k$

window-size: $2k+1$

(we need not replace first and last $k$ items)

Naive algorithm: find median in each window

More efficient: use two heaps

Application: removing noise

- replace each point by median in surrounding window
- for some common types of noise, this is effective (corrupted point unlikely to be the median)
- depends on type/amount of noise and window-size, but often effective in practice
Algorithm idea:
as we go through the stream,
maintain two heaps that store current window of 2k+1 items:
- a max-heap H1 of size k
- a min-heap H2 of size k+1

Invariant:
all items in H1 are < all items in H2

(So minimum of H2 is the median of the 2k+1 items in H1 and H2.)

Running time: heap operations take $O(\log k)$ time in heaps of size $\sim k$.

Initial work (step 1): $O(k)$, as we partition 2k+1 items, build two heaps
Work for processing each item (step 2): $O(\log k)$

Total: $O(n\log k)$

Exercise: show that $O(n\log k)$ is best possible (Hint: use median filter to sort)

Algorithm:
1. put $a_1, \ldots, a_{2k+1}$ into the two heaps according to the invariant.
2. for $i=2k+2$ to $n$:
   x := find_min(H2) (this is the current median)
   output x
   a:=a_i (next element in stream)
   if a<x:
       insert(H1,a)
   else:
       insert(H2,a)
   delete $a_{i-(2k+1)}$ from heap that stores it (we have pointer to it)
   restore invariant
   (if |H1| > k, extract_max from H1 and insert into H2)
   (if |H1| < k, extract_min from H2 and insert into H1)