(Advanced) Data Structures

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~~ online lectures, group meeting(s):
first: Tue 21st April 10:15 Webex meeting (fallback: Jitsi meet)

Tutorials: Wed 14-16 online meetings Webex

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!

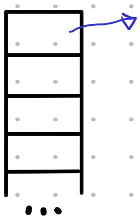
To pass:
- exam (oral/written): to be decided later
- 60% of exercise points, including programming exercise
- active participation in tutorials (details later)
- final grade: exam only.

1. Basic data structures

RAM model of computation

centralised memory

S[0]
S[1]
S[2]



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)

Operations:
S[I] := k (constant)
S[I] := S[J]
S[I] := S[J] <op> S[K]

I,J,K can be constant or of the form S[constant] ("indirect addressing)

e.g. S[S[4]] := S[6]
<op> can be +,-,*,bitwise operations, etc.

Tests:
S[I] <cmp> S[J]
<cmp> can be =, <, <=, etc.

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

- results stored in memory cells
- input/output stored in memory cells

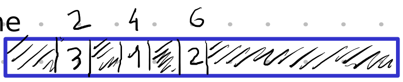
Implementing an array

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

- 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

solution sketch: use 3 arrays

Time



(stores "time" of first writing, or garbage, if not yet written)

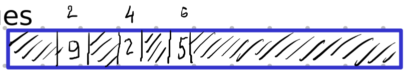
$\boxed{n}$ size $\boxed{3}$ current $\boxed{c}$ init

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

Written

(stores indices of cells in order of first writing, correct for cells 1,...,current)

Values



(stores array values, or garbage, if not yet written)

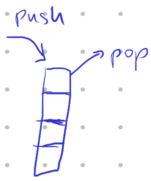
illustration of state after three writes:
A[4]:=2, A[6]:=5, A[2]:=9

```
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

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

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

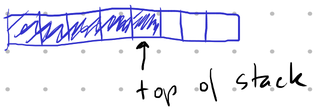
Verify correctness and that all three operations take O(1) time.

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?

Doubling/halving strategy

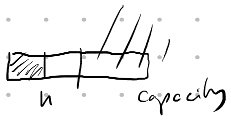
make_stack creates an array of size 3

capacity denotes size of the array, initially 3
n denotes size of stack (number of items), initially 0

We maintain the following invariant at all times:

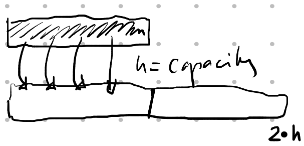
$$\left\lfloor \frac{capacity}{4} \right\rfloor \le n \le capacity$$

pop() — check if stack nonempty
    n:=n-1
    output A[n+1]
    if (n==[capacity/4])
        capacity = [capacity/2] — free up half of array



(Note: we don't reduce capacity even more, bec. 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



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.

Observation. When we allocate a new array of size 2n and copy n items (i.e. when we do a costly push), then there have been at least n/2 simple push operations since last allocation.

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.

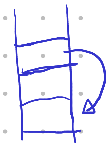
A simple push actual cost is constant, 1 unit can pay for it, 2 units are deposited.

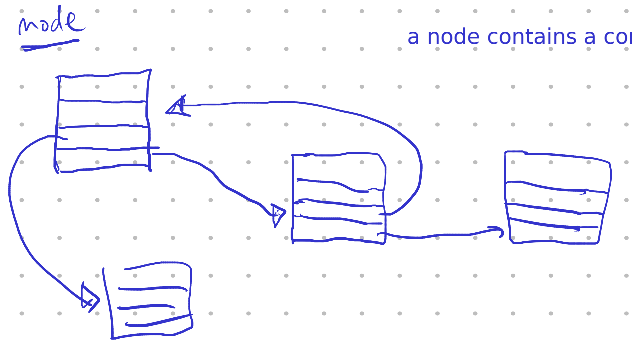
When it comes to the costly push, by Observation, we have already deposited n units.

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

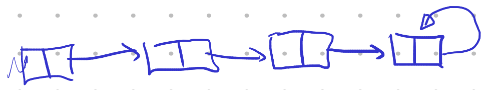
Exercise: modify design such that actual cost is constant, not just the amortized.
(should still be array-based, not pointer-based).

Pointer-based data structures





a node contains a constant number of fields and pointers



operations O(1) time | Stack implementation

beginning

- nodes w. a constant # of fields & pointers
- a constant # of global variables
- a **root** node

} pointer machine structures (restriction of RAM model)

→ no arithmetic on addresses

→ no hashing

pointer-based DS

+ more flexible
+ no need for contiguous memory
− less efficient
− need more space

Heaps (priority queues)

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

- operations:
    - H := make_heap() → create empty heap
    - insert(H,x)
    - extract_min(H) → extract x with min x.key
    - find_min(H)
    - delete(x) → item x
    - meld(H1,H2) → merge, union
    - decreasekey(x,k) → e.g. Dijkstra

Assume keys only accessed via comparisons.

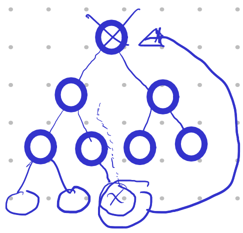
items in heap

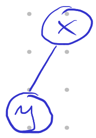
→ Thm. At least one of extract_min and insert must take time $\Omega(\log n)$

insert $x_1, x_2, x_3, \ldots, x_n$
exctract_min n times

→ sort → $\Omega(n \log n)$

Heap implementation:

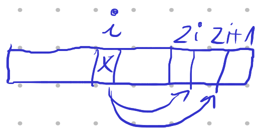




min-heap

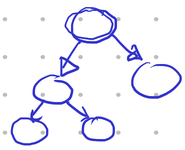
x.key ≤ y.key

Implement:
- array



also d-ary

- pointers



insert and 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 delete and
extract_min in O(logn), other operations in O(1).
Check which bounds are amortized in which data structure.

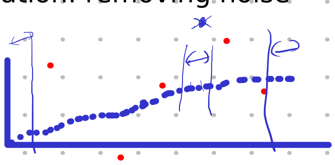
Heap application: MEDIAN FILTER

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)

Application: removing noise



O(k)   2k+1

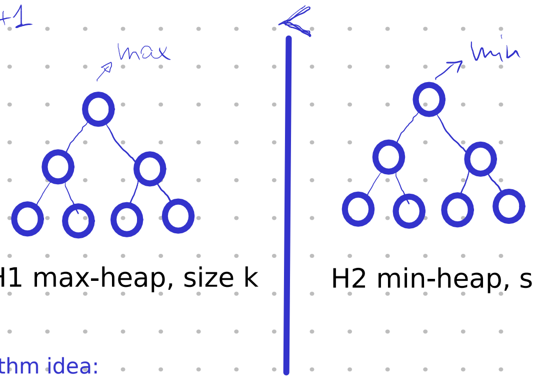
Naive algorithm: find median in each window → O(nk)

(use linear time selection)

More efficient: use two heaps

- 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

2k+1



H1 max-heap, size k

H2 min-heap, size k+1

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.)

Algorithm:

1. put a_1, ..., 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)

Running time: heap operations take O(logk) time in heaps of size ~k.

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

Exercise: show that O(nlogk) is best possible (Hint: use median filter to sort)