(Advanced) Data Structure Lectures: László Kozma < Ikozm Tutorials: Katharina Klost	₽ S a@inf.fu-berlin.de >	· · · · · · · · · · · · · · · · · · ·	· · · · · ·		· · · · · · · ·
Contents: - Selected topics about data st - Theory course, M.Sc. level	ructures and a	pplications	5		· · · · · · · · ·
Course website: - KVV/Whiteboard - page.mi.fu-berlin.de/lkozma/c	ds2020	· · · · · ·	· · · · · ·		· · · · · · · ·
Organization: Lectures: Tue/Thu 10-12 Tutorials: Wed 14-16 online	nline lectures, grou first: Tue 21st A e meetings Webex	ıp meeting(s) April 10:15Wel	: bex meeting	(fallback: Jitsi	meet)
Prerequisites: Algorithms/Mathematics (~ HA)	- O-notation, asymp - computational mo - basic data structu - amortised analysi - basic graph algori	ototics odels (TM, RAI ires (array, lis s (will recap) ithms	M model, poir t, stack, queu	nter machine) Je, balanced) BST)
Books / references:	· · · · · · ·	· · · · · ·	• • • • •	••••	· · · · · · ·
(see course websites)		0 0 0 0 0			
 one exercise sheet each week deadline ~ 10 days (Tue to next Fri) first sheet online, due 1st May (total 1 details to follow (how to submit, etc.) encouraged to work in pairs one programming exercise (details lat cite all sources, collaborators! 	1/12) ter)	· · · · · · · · · · · · · · · · · · ·			.
<u> </u>					• • • • • • •
lo pass:					
 exam (oral/written): to be decided late 60% of exercise points, including prog 	er Iramming exercise	• • • • •			
 active participation in tutorials (details final grade: exam only. 	s later)	0 0 0 0 0		0 0 0 0	
			• • • • •		
		· · · · ·			· · · · · ·
		• • • • •			
	• • • • • •		• • • • •		
	• • • • • •	• • • • •	• • • • •		• • • • • •
			• • • • •	• • • •	
		0 0 0 0 0			· · · · · ·
					• • • • • •
			• • • • •		
		0 0 0 0	0 0 0 0	0 0 0 0	
		• • • • •		0 0 0 0	

1 Basic data structures		
		Operations:
	• • • • • • • • • •	S[I] := k (constant)
		S[I] := S[J]
RAM model of computation		S[1] := S[J] < 0p > S[K]
Contralised memory		I'l K' can be constant or of the form
		S[constant] ("indirect addressing)
SI01 cell can contain inte	eger word on $c \log n$ bits	
S[1] (n ^c different inte	egers since cells address	e.g. S[S[4]] := S[6]
S[2] memory, we can he	ave at most this many	<op> can be +,-,*,bitwise operations, etc.</op>
		Tests:
		••••••••••••••••••••••••••••••••••••••
		<cmp $>$ can be =, <, <=, etc.
 RAM model is an idealised compute 	er	- results stored in memory cells
- Each elementary operation/test tal	kės O(I) timė	- input/output stored in memory cells
 - can implement familiar structures, 	pointers, arrays, etc.	
Implementing an array	- let's try to implement an ar	ray on the BAM machine
$1 \cdot \Lambda = m_2 k_0 \cdot 2rr_2 v (n_1 c)$	(we usually don't worry about	t such low-level details)
1. $A = IIIaKe_aIIay(II,C)$ 2. read(A i)	- support three basic operation	
2. redu(A,I)	- n is size of the array, c is in	itial value (if an array entry
	has not been written, it has v	value c)
	- difficulty: we cannot make a	assumption on initial memory contents
	- all three operations should	take O(1) time
solution sketch: use 3 arrays		
· · · · · · · · · · · · · · · · · · ·		
Time 2 4 6	n 3 c	
	size current init	
	size current init	when we allocate array we don't
(stores "time" of first writing or garbage	size current init	when we allocate array, we don't
(stores "time" of first writing, or garbage, if not yet written)	Size current mit.	when we allocate array, we don't care about memory content
(stores "time" of first writing, or garbage, if not yet written)	size current mit	when we allocate array, we don't care about memory content
(stores "time" of first writing, or garbage, if not yet written) Written 2 3	make_arr	when we allocate array, we don't care about memory content ay(n,c) rate arrays Time, Values, Written of size n
(stores "time" of first writing, or garbage, if not yet written) Written 2 3 4 6 2 //////////////////////////////////	make_arr. alloc	when we allocate array, we don't care about memory content ay(n,c) ate arrays Time, Values, Written of size n =n. current:=0. init:=c
(stores indices of cells in order of first writing (stores "time" of first writing, or garbage, if not yet written) Written (stores indices of cells in order of first writin correct for cells 1,,current)	make_arr alloc size:	when we allocate array, we don't care about memory content ay(n,c) ate arrays Time, Values, Written of size n =n, current:=0, init:=c
<pre>(stores "time" of first writing, or garbage, if not yet written) Written 2 3 4 6 2 (stores indices of cells in order of first writin correct for cells 1,,current) Values 2 4 6</pre>	make_arr alloc size:	when we allocate array, we don't care about memory content ay(n,c) ate arrays Time, Values, Written of size n =n, current:=0, init:=c
(stores "time" of first writing, or garbage, if not yet written) Written 2 (stores indices of cells in order of first writin correct for cells 1,,current) Values 2 4 5 7 9 7 9 7 9 7 7 9 7 7 7 7 7 7 7 7 7 7	make_arr alloc size:	when we allocate array, we don't care about memory content ay(n,c) ate arrays Time, Values, Written of size n =n, current:=0, init:=c
(stores "time" of first writing, or garbage, if not yet written) Written 2 3 4 6 2 //////////////////////////////////	written)	when we allocate array, we don't care about memory content ay(n,c) ate arrays Time, Values, Written of size n =n, current:=0, init:=c
(stores "time" of first writing, or garbage, if not yet written) Written 2 3 4 6 2 //////////////////////////////////	written)	when we allocate array, we don't care about memory content ay(n,c) rate arrays Time, Values, Written of size n =n, current:=0, init:=c
<pre>(stores "time" of first writing, or garbage, if not yet written) Written 2 3</pre>	written) written	when we allocate array, we don't care about memory content ay(n,c) ate arrays Time, Values, Written of size n =n, current:=0, init:=c n(i) me[i]<=current) and (Written[Time[i]]=i)
<pre>(x/A 3 / x/1 / x/2 / x/2</pre>	written) written) written	when we allocate array, we don't care about memory content ay(n,c) ate arrays Time, Values, Written of size n =n, current:=0, init:=c n(i) me[i]<=current) and (Written[Time[i]]=i) return TRUE return FALSE
<pre>(x/A 3 / x/1 / x/2 / x/2</pre>	written) written) written	when we allocate array, we don't care about memory content ay(n,c) ate arrays Time, Values, Written of size n =n, current:=0, init:=c n(i) me[i]<=current) and (Written[Time[i]]=i) return TRUE return FALSE
<pre>(xin 3) xin 1 xin 2 xin 2</pre>	written) written) written	when we allocate array, we don't care about memory content ay(n,c) ate arrays Time, Values, Written of size n =n, current:=0, init:=c n(i) me[i]<=current) and (Written[Time[i]]=i) return TRUE return FALSE
<pre>(x/A 3 / x/1 / x/2 / x/2</pre>	written) written) written) is_writter if (Tin else	when we allocate array, we don't care about memory content ay(n,c) ate arrays Time, Values, Written of size n =n, current:=0, init:=c n(i) me[i]<=current) and (Written[Time[i]]=i) return TRUE return FALSE
<pre>(x/A 3 / x/1 / x/2 / x/2</pre>	written) written) is_writter if (Tin else read(i)	when we allocate array, we don't care about memory content ay(n,c) ate arrays Time, Values, Written of size n =n, current:=0, init:=c n(i) me[i]<=current) and (Written[Time[i]]=i) return TRUE return TRUE return FALSE
<pre>(xin 3 x 1 x 2 x - x - x - x - x - x - x - x - x - x</pre>	written) written) is_writter if (Tin else read(i) if (is	when we allocate array, we don't care about memory content ay(n,c) ate arrays Time, Values, Written of size n =n, current:=0, init:=c n(i) me[i]<=current) and (Written[Time[i]]=i) return TRUE return TRUE return FALSE
<pre>(stores "time" of first writing, or garbage, if not yet written) Written 2 3 (stores indices of cells in order of first writin correct for cells 1,,current) Values 2 4 6 (stores array values, or garbage, if not yet illustration of state after three writes: A[4]:=2, A[6]:=5, A[2]:=9 write(i,v). Values[i]:=v if (is_written(i)) do nothing else</pre>	written) written) is_writter if (Tiu else read(i) if (is	when we allocate array, we don't care about memory content ay(n,c) ate arrays Time, Values, Written of size n =n, current:=0, init:=c n(i) me[i]<=current) and (Written[Time[i]]=i) return TRUE return TRUE return FALSE
<pre>(xin 3 x 1 x 2 x - x - x - x - x - x - x - x - x - x</pre>	written) written) is_writter if (Tin else read(i) if (is	when we allocate array, we don't care about memory content ay(n,c) ate arrays Time, Values, Written of size n =n, current:=0, init:=c n(i) me[i]<=current) and (Written[Time[i]]=i) return TRUE return TRUE return FALSE
<pre>(xi 3 y 1) (%) (2 y 2 y 2 y 2 y 4 y 4 y 4 y 4 y 4 y 4 y</pre>	written) written) is_writter if (Tin else read(i) if (is else else	when we allocate array, we don't care about memory content ay(n,c) ate arrays Time, Values, Written of size n =n, current:=0, init:=c n(i) me[i]<=current) and (Written[Time[i]]=i) return TRUE return TRUE return FALSE
<pre>(stores "time" of first writing, or garbage, if not yet written) Written 2 3 (stores indices of cells in order of first writin correct for cells 1,,current) Values 2 4 6 (stores array values, or garbage, if not yet illustration of state after three writes: A[4]:=2, A[6]:=5, A[2]:=9 write(i,v) Values[i]:=v if (is_written(i)) do nothing else current:=current+ Written[current]:= Time[i]:=current</pre>	written) written) is_writter if (Tin else read(i) if (is else else	when we allocate array, we don't care about memory content ay(n,c) ate arrays Time, Values, Written of size n =n, current:=0, init:=c n(i) me[i]<=current) and (Written[Time[i]]=i) return TRUE return TRUE return FALSE s_written(i)) return Values[i] return init
<pre>(stores "time" of first writing, or garbage, if not yet written) Written 2 3 (stores indices of cells in order of first writin correct for cells 1,,current) Values 2 4 6 (stores array values, or garbage, if not yet illustration of state after three writes: A[4]:=2, A[6]:=5, A[2]:=9 write(i,v). Values[i]:=v if (is_written(i)) do nothing else current:=current+ Written[current]:= Time[i]:=current</pre>	written) written) is_writter if (Tiu else read(i) if (is else	when we allocate array, we don't care about memory content ay(n,c) ate arrays Time, Values, Written of size n =n, current:=0, init:=c n(i) me[i]<=current) and (Written[Time[i]]=i) return TRUE return TRUE return FALSE
(stores "time" of first writing, or garbage, if not yet written) Written 2 3 (162) (stores indices of cells in order of first writin correct for cells 1,,current) Values 2 4 6 (stores array values, or garbage, if not yet illustration of state after three writes: A[4]:=2, A[6]:=5, A[2]:=9 write(i,v). Values[i]:=v if (is_written(i)) do nothing else current:=current+ Written[current]:= Time[i]:=current	written) written) is_writter if (Tin else read(i) if (is else	when we allocate array, we don't care about memory content ay(n,c) ate arrays Time, Values, Written of size n =n, current:=0, init:=c n(i) me[i]<=current) and (Written[Time[i]]=i) return TRUE return TRUE return FALSE s_written(i)) return Values[i] return init
<pre>(stores "time" of first writing, or garbage, if not yet written) Written 2 3 (4) 6 2) (stores indices of cells in order of first writin correct for cells 1,,current) Values 2 4 6 (199) 22 15 (stores array values, or garbage, if not yet illustration of state after three writes: A[4]:=2, A[6]:=5, A[2]:=9 write(i,v) Values[i]:=v if (is_written(i)) do nothing else current:=current4 Written[current]:= Time[i]:=current</pre>	vritten) written) is_writter if (Tin else read(i) if (is else verify correctness and that all three	when we allocate array, we don't care about memory content ay(n,c) rate arrays Time, Values, Written of size n =n, current:=0, init:=c n(i) me[i]<=current) and (Written[Time[i]]=i) return TRUE return TRUE return FALSE s_written(i)) return Values[i] return init
<pre>(stores "time" of first writing, or garbage, if not yet written) Written 2 3 (4) 6 2) (stores indices of cells in order of first writin correct for cells 1,,current) Values 2 4 6 (1) 9 2 10 5 (stores array values, or garbage, if not yet illustration of state after three writes: A[4]:=2, A[6]:=5, A[2]:=9 write(i,v) Values[i]:=v if (is_written(i)) do nothing else current:=current+ Written[current]:= Time[i]:=current</pre>	vritten) written) is_writter if (Tin else read(i) if (is else Verify correctness and that all three	when we allocate array, we don't care about memory content ay(n,c) ate arrays Time, Values, Written of size n =n, current:=0, init:=c n(i) me[i]<=current) and (Written[Time[i]]=i) return TRUE return TRUE return FALSE s_written(i)) return Values[i] return init
<pre>(stores "time" of first writing, or garbage, if not yet written) Written 2 3</pre>	size current init make_arr alloc size: written) is_writter if (Tin else read(i) if (is else Verify correctness and that all three	when we allocate array, we don't care about memory content ay(n,c) tate arrays Time, Values, Written of size n =n, current:=0, init:=c n(i) me[i]<=current) and (Written[Time[i]]=i) return TRUE return TRUE return FALSE s_written(i)) return Values[i] return init
<pre>(stores "time" of first writing, or garbage, if not yet written) Written 2 3 (4 6 2)///////////////////////////////////</pre>	written) written) is_writter if.(Tiu else read(i) if (is else verify correctness and that all three	when we allocate array, we don't care about memory content ay(n,c) sate arrays Time, Values, Written of size n =n, current:=0, init:=c n(i) me[i]<=current) and (Written[Time[i]]=i) return TRUE return TRUE return FALSE s_written(i)) return Values[i] return init
(stores "time" of first writing, or garbage, if not yet written) Written 2 3 (4 6 2) (stores indices of cells in order of first writin correct for cells 1,,current) Values 2 4 6 (19 2) (stores array values, or garbage, if not yet illustration of state after three writes: A[4]:=2, A[6]:=5, A[2]:=9 write(i,v). Values[i]:=v if (is_written(i)) do nothing else current:=current+ Written[current]:= Time[i]:=current	written) written) is_writter if (Tin else read(i) if (is +1 =i Verify correctness and that all three	<pre>when we allocate array, we don't care about memory content ay(n,c) ate arrays Time, Values, Written of size n =n, current:=0, init:=c n(i) me[i]<=current) and (Written[Time[i]]=i) return TRUE return TRUE return FALSE s_written(i)) return Values[i] return init operations take O(1) time.</pre>
(stores "time" of first writing, or garbage, if not yet written) Written 2 3 (1622) (stores indices of cells in order of first writin correct for cells 1,,current) Values 2 4 6 (stores array values, or garbage, if not yet illustration of state after three writes: A[4]:=2, A[6]:=5, A[2]:=9 write(i,v). Values[i]:=v if (is_written(i)) do nothing else current:=current4 Written[current]:= Time[i]:=current	written) written) is_writter if (Tin else read(i) if (is else Verify correctness and that all three	<pre>when we allocate array, we don't care about memory content ay(n,c) cate arrays Time, Values, Written of size n = n, current:=0, init:=c n(i) me[i]<=current) and (Written[Time[i]]=i) return TRUE return TRUE return FALSE s_written(i)) return Values[i] return init operations take O(1) time.</pre>
(stores "time" of first writing, or garbage, if not yet written) Written 2 3 (5 of 2) (stores indices of cells in order of first writin correct for cells 1,,current) Values 2 4 6 (1) 9 2 1 5 1 5 1 5 1 5 1 5 1 5 1 5 1 5 1 5 1	written) written) is_writter if (Tiu else read(i) if (is else Verify correctness and that all three	when we allocate array, we don't care about memory content ay(n,c) rate arrays Time, Values, Written of size n =n, current:=0, init:=c n(i) me[i]<=current) and (Written[Time[i]]=i) return TRUE return TRUE return FALSE s_written(i)) return Values[i] return init
<pre>(stores "time" of first writing, or garbage, if not yet written) Written (stores indices of cells in order of first writin correct for cells 1,,current) Values</pre>	size current init. make_arr alloc size: written) is_writter if.(Tiu else read(i) if (is else +1 =i	<pre>when we allocate array, we don't care about memory content ay(n,c) rate arrays Time, Values, Written of size n =n, current:=0, init:=c n(i) me[i]<=current) and (Written[Time[i]]=i) return TRUE return TRUE return FALSE s_written(i)) return Values[i] return init operations take O(1) time.</pre>
<pre>(stores "time" of first writing, or garbage, if not yet written) Written (stores indices of cells in order of first writin correct for cells 1,,current) Values ² 4 ⁶ (stores array values, or garbage, if not yet illustration of state after three writes: A[4]:=2, A[6]:=5, A[2]:=9 write(i,v) Values[i]:=v if (is_written(i)) do nothing else current:=current+ Written[current]:= Time[i]:=current</pre>	vritten) written) is_writter if.(Tiu else read(i) if (is else Verify correctness and that all three	<pre>when we allocate array, we don't care about memory content ay(n,c) ate arrays Time, Values, Written of size n =n, current:=0, init:=c n(i) me[i]<=current) and (Written[Time[i]]=i) return TRUE return TRUE return FALSE s_written(i)) return Values[i] return init operations take O(1) time.</pre>
<pre>(stores "time" of first writing, or garbage, if not yet written) Written 2 3 [4] 6 2]//////////////////////////////////</pre>	vritten) written) is_writter if (Tine else read(i) if (ise verify correctness and that all three	<pre>when we allocate array, we don't care about memory content ay(n,c) sate arrays Time, Values, Written of size n =n, current:=0, init:=c n(i) me[i]<=current) and (Written[Time[i]]=i) return TRUE return TRUE return FALSE s_written(i)) return Values[i] return init operations take O(1) time.</pre>

Implementing a stack with an array	
1 S = make stack/)	
$1. S = (\text{Ilake_stack}) \qquad Push$	
3. $push(S,v)$	Doubling/halving strategy
	make_stack creates an array of size 3
	capacity denotes size of the array, initially 3
All three operations should take O(1) time	n denotes size of stack (number of items), initially 0
Colar that the Miles	
<u></u>	• We maintain the following invariant at all times: • • • •
top of stack	
Idea: Allocate an array,	Copality & M & Capacity
keep track of top of the stack update for each pop/push	
Problem: how large should array be?	
· · · · · · · · · · · / · / · · · · · / · / · · · · · · / · / · · · · · · · / · / · · · · · · · / · / · · · · · · · / · / · · · · · · · · / · / · · · · · · · · / · / · · · · · · · · · / · / · · · · · · · · / · / · · · · · · · · / · / · · · · · · · · · / · / · · · · · · · · · / · / · · · · · · · · · / · / · · · · · · · · · · / · / · · · · · · · · · · · · / · / · · · · · · · · · · · · · · / · / · · · · · · · · · · · · · · · · / · / · · · · · · · · · · · · · · · / · / ·	
check if stock	
output A[n+1]	
$\frac{11}{(n==[capacity/4])} = [capacity/2] - (ree up half = 0)$	
arrm	
(Note: we den't reduce capacity even more bec. we den't	
want to trigger a doubling too soon)	
push(v)	
n:=n+1	• • • • • • • • • • • • • • • • • • • •
A[n]:=v if $(n==capacity)$	· · · · · · · · · · · · · · · · · · ·
allocate A of size 2n	 2•h
capacity:=2n	· · · · · · · · · · · · · · · · · · ·
free up old array	
• • • • • • • • • • • • • • • • • • • •	
· Analysis: · · · · · · · · · · · · · · · · · · ·	
Analysis:	· · · · · · · · · · · · · · · · · · ·
Analysis: the only problem is that push may take time ~n	
Analysis: the only problem is that push may take time ~n we cannot give O(1) bound on actual cost of push (i.e. wher	ion. When we allocate a new array of size 2n and copy n items
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. Observati (i.e. when simple pu	ion. When we allocate a new array of size 2n and copy n items n we do a costly push), then there have been at least n/2 ush operations since last allocation.
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. Observati (i.e. wher simple pu	ion. When we allocate a new array of size 2n and copy n items n we do a costly push), then there have been at least n/2 ush operations since last allocation.
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. Observati (i.e. wher simple pu	ion. When we allocate a new array of size 2n and copy n items n we do a costly push), then there have been at least n/2 ush operations since last allocation.
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. Observati (i.e. when simple pu	ion. When we allocate a new array of size 2n and copy n items n we do a costly push), then there have been at least n/2 ush operations since last allocation.
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: Operations make_stack and pop	ion. When we allocate a new array of size 2n and copy n items n we do a costly push), then there have been at least n/2 ush operations since last allocation. are fine, since actual cost is also constant, 1 unit can pay for it.
Analysis: the only problem is that push may take time ~n we cannot give O(1) bound on actual cost of push Observati (i.e. when simple push we do an amortized analysis. Amortized costs: Operations make_stack and pop make_stack: 1 unit.	ion. When we allocate a new array of size 2n and copy n items n we do a costly push), then there have been at least n/2 ush operations since last allocation.
Analysis: the only problem is that push may take time ~n we cannot give O(1) bound on actual cost of push Observati (i.e. when simple push we do an amortized analysis. Amortized costs: Operations make_stack and pop make_stack: 1 unit pop: 1 unit A simple push actual cost is cons	ion. When we allocate a new array of size 2n and copy n items n we do a costly push), then there have been at least n/2 ush operations since last allocation. are fine, since actual cost is also constant, 1 unit can pay for it. stant, 1 unit can pay for it, 2 units are deposited.
Analysis: the only problem is that push may take time ~n we cannot give O(1) bound on actual cost of push Observati (i.e. wher simple push we do an amortized analysis. Amortized costs: Operations make_stack and pop make_stack: 1 unit pop: 1 unit A simple push actual cost is cons when it comes to the costly push	ion. When we allocate a new array of size 2n and copy n items n we do a costly push), then there have been at least n/2 ush operations since last allocation. are fine, since actual cost is also constant, 1 unit can pay for it. stant, 1 unit can pay for it, 2 units are deposited. h, by Observation, we have already deposited n units.
Analysis: the only problem is that push may take time ~n we cannot give O(1) bound on actual cost of push Observati (i.e. wher simple push we do an amortized analysis. Amortized costs: Operations make_stack and pop make_stack: 1 unit pop: 1 unit push: 3 unit A simple push actual cost is cons (all three operations have constraint exerctions have This pays for the copying (actual	 ion. When we allocate a new array of size 2n and copy n items n we do a costly push), then there have been at least n/2 ush operations since last allocation. are fine, since actual cost is also constant, 1 unit can pay for it. stant, 1 unit can pay for it, 2 units are deposited. h, by Observation, we have already deposited n units. I cost O(n)).
Analysis: the only problem is that push may take time ~n we cannot give O(1) bound on actual cost of push Observati (i.e. when simple push we do an amortized analysis. Operations make_stack and pop Amortized costs: Operations make_stack and pop make_stack: 1 unit pop: 1 unit push: 3 unit A simple push actual cost is cons (all three operations have constant amortized cost) This pays for the copying (actual	ion. When we allocate a new array of size 2n and copy n items n we do a costly push), then there have been at least n/2 ush operations since last allocation. are fine, since actual cost is also constant, 1 unit can pay for it. stant, 1 unit can pay for it, 2 units are deposited. h, by Observation, we have already deposited n units. I cost O(n)).
Analysis: the only problem is that push may take time ~n we cannot give O(1) bound on actual cost of push Observati (i.e. wher simple push we do an amortized analysis. Amortized costs: Operations make_stack and pop make_stack: 1 unit pop: 1 unit A simple push actual cost is const When it comes to the costly push (all three operations have constant amortized cost) This pays for the copying (actual	 ion. When we allocate a new array of size 2n and copy n items n we do a costly push), then there have been at least n/2 ush operations since last allocation. are fine, since actual cost is also constant, 1 unit can pay for it. stant, 1 unit can pay for it, 2 units are deposited. h, by Observation, we have already deposited n units. I cost O(n)).
 Analysis: the only problem is that push may take time ~n we cannot give O(1) bound on actual cost of push Observati (i.e. wher simple push we do an amortized analysis. Amortized costs: Operations make_stack and pop make_stack; 1 unit pop: 1 unit push: 3 unit (all three operations have constant amortized cost) Exercise: modify design such that actual cost is constant, not just to the cost is constant. 	 ion. When we allocate a new array of size 2n and copy n items nive do a costly push), then there have been at least n/2 ush operations since last allocation. are fine, since actual cost is also constant, 1 unit can pay for it. stant, 1 unit can pay for it, 2 units are deposited. h, by Observation, we have already deposited n units. I cost O(n)).
 Analysis: the only problem is that push may take time ~n we cannot give O(1) bound on actual cost of push Observati (i.e. where simple push of the cost of push Observation of the cost of push Observation of the cost of push Observation of the cost of push Observation of the cost of push of the cost of the cost of push of the cost of the cost of the cost of the cost of push of the cost of the cost of push of the cost of the cost of the cost of the cost of push of the cost of the	 ion. When we allocate a new array of size 2n and copy n items n we do a costly push), then there have been at least n/2 ush operations since last allocation. are fine, since actual cost is also constant, 1 unit can pay for it. stant, 1 unit can pay for it, 2 units are deposited. h, by Observation, we have already deposited n units. I cost O(n)).
Analysis: the only problem is that push may take time ~n we cannot give O(1) bound on actual cost of push Observati (i.e. when simple push we do an amortized analysis. Operations make_stack and pop Amortized costs: Operations make_stack and pop make_stack: 1 unit pop: 1 unit push: 3 unit A simple push actual cost is const When it comes to the costly push (all three operations have constant amortized cost) This pays for the copying (actual the copying (actual cost is constant, not just the cost of the cost of the copying (actual cost of the copying (actual)	 ion. When we allocate a new array of size 2n and copy n items new do a costly push), then there have been at least n/2 ush operations since last allocation. are fine, since actual cost is also constant, 1 unit can pay for it. stant, 1 unit can pay for it, 2 units are deposited. h, by Observation, we have already deposited n units. I cost O(n)).
 Analysis: the only problem is that push may take time ~n we cannot give O(1) bound on actual cost of push (i.e. where simple push and the push actual cost is constant amortized cost) Amortized costs: Operations make_stack and pop make_stack: 1 unit pop: 1 unit push: 3 unit A simple push actual cost is constant amortized cost) A simple push actual cost is constant, not just t (should still be array-based, not pointer-based). 	 ion. When we allocate a new array of size 2n and copy n items nive do a costly push), then there have been at least n/2 ush operations since last allocation. are fine, since actual cost is also constant, 1 unit can pay for it. stant, 1 unit can pay for it, 2 units are deposited. h, by Observation, we have already deposited n units. I cost O(n)).
Analysis: the only problem is that push may take time ~n we cannot give O(1) bound on actual cost of push Observati (i.e. wher we do an amortized analysis. Amortized costs: Operations make_stack and pop make_stack: 1 unit pop: 1 unit push: 3 unit A simple push actual cost is come (all three operations have constant amortized cost) This pays for the copying (actual Exercise: modify design such that actual cost is constant, not just t (should still be array-based, not pointer-based).	ion. When we allocate a new array of size 2n and copy n items n we do a costly push), then there have been at least n/2 ish operations since last allocation. are fine, since actual cost is also constant, 1 unit can pay for it. stant, 1 unit can pay for it, 2 units are deposited. h, by Observation, we have already deposited n units. I cost O(n)).
Analysis: the only problem is that push may take time ~n we cannot give O(1) bound on actual cost of push Observati (i.e. wher we do an amortized analysis. Amortized costs: Operations make_stack and pop make_stack: 1 unit pop: 1 unit push: 3 unit A simple push actual cost is consecutive (all three operations have constant amortized cost) This pays for the copying (actual Exercise: modify design such that actual cost is constant, not just to (should still be array-based, not pointer-based).	ion. When we allocate a new array of size 2n and copy n items n we do a costly push), then there have been at least n/2 ush operations since last allocation. are fine, since actual cost is also constant, 1 unit can pay for it. stant, 1 unit can pay for it, 2 units are deposited. h, by Observation, we have already deposited n units. I cost_O(n)). the amortized.
Analysis: the only problem is that push may take time ~n we cannot give O(1) bound on actual cost of push Observati (i.e. wher we do an amortized analysis. Amortized costs: Operations make_stack and pop make_stack: 1 unit pop: 1 unit push: 3 unit A simple push actual cost is cons (all three operations have constant amortized cost) This pays for the copying (actual Exercise: modify design such that actual cost is constant, not just to (should still be array-based, not pointer-based).	ion. When we allocate a new array of size 2n and copy n items n we do a costly push), then there have been at least n/2 ish operations since last allocation. are fine, since actual cost is also constant, 1 unit can pay for it. stant, 1 unit can pay for it, 2 units are deposited. h, by Observation, we have already deposited n units. I cost O(n)).
Analysis: the only problem is that push may take time ~n we cannot give O(1) bound on actual cost of push Observati (i.e. wher we do an amortized analysis. Amortized costs: Operations make_stack and pop make_stack: 1 unit pop: 1 unit push: 3 unit A simple push actual cost is conse (all three operations have constant amortized cost) This pays for the copying (actual Exercise: modify design such that actual cost is constant, not just to (should still be array-based, not pointer-based).	 ion. When we allocate a new array of size 2n and copy n items new do a costly push), then there have been at least n/2 ush operations since last allocation. are fine, since actual cost is also constant, 1 unit can pay for it. stant, 1 unit can pay for it, 2 units are deposited. h, by Observation, we have already deposited n units. I cost_O(n)).
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: Amortized costs: Operations make_stack and pop make_stack: 1 unit pop: 1 unit push: 3 unit (all three operations have constant amortized cost) Exercise: modify design such that actual cost is constant, not just the (should still be array-based, not pointer-based).	ion. When we allocate a new array of size 2n and copy n items n we do a costly push), then there have been at least n/2 ish operations since last allocation. are fine, since actual cost is also constant, 1 unit can pay for it. stant, 1 unit can pay for it, 2 units are deposited. h, by Observation, we have already deposited n units. I cost O(n)).
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: Amortized costs: Operations make_stack and pop make_stack: 1 unit pop: 1 unit push: 3 unit (all three operations have constant amortized cost) Exercise: modify design such that actual cost is constant, not just t (should still be array-based, not pointer-based).	ion. When we allocate a new array of size 2n and copy n items n we do a costly push), then there have been at least n/2 ush operations since last allocation. are fine, since actual cost is also constant, 1 unit can pay for it. stant, 1 unit can pay for it, 2 units are deposited. h, by Observation, we have already deposited n units. I cost O(n)). the amortized.
Analysis: the only problem is that push may take time ~n we cannot give O(1) bound on actual cost of push Observati (i.e. wher we do an amortized analysis. Amortized costs: Operations make_stack and pop make_stack: 1 unit pop: 1 unit push: 3 unit A simple push actual cost is const (all three operations have constant amortized cost) This pays for the copying (actual Exercise: modify design such that actual cost is constant, not just t (should still be array-based, not pointer-based).	ion. When we allocate a new array of size 2n and copy n items n we do a costly push), then there have been at least n/2 ish operations since last allocation. are fine, since actual cost is also constant, 1 unit can pay for it. stant, 1 unit can pay for it, 2 units are deposited. h, by Observation, we have already deposited n units. I cost_O(n)). the amortized.
Analysis: the only problem is that push may take time ~n we cannot give O(1) bound on actual cost of push Observati (i.e. wher we do an amortized analysis. Amortized costs: Operations make_stack and pop make_stack: 1 unit pop: 1 unit push: 3 unit A simple push actual cost is conserved (all three operations have constant amortized cost) This pays for the copying (actual Exercise: modify design such that actual cost is constant, not just to (should still be array-based, not pointer-based).	<pre>ion. When we allocate a new array of size 2n and copy n items n we do a costly push), then there have been at least n/2 ush operations since last allocation. are fine, since actual cost is also constant, 1 unit can pay for it. stant, 1 unit can pay for it, 2 units are deposited. h, by Observation, we have already deposited n units. I cost_O(n)). the amortized.</pre>

Pointer-based data structures	
mode	a node contains a constant number of fields and pointers
operation (Stack implemention of D(1) time	
. modes wa countant # of fiel	des et portitées (pointer macune sometimes
· a constant it of glose united) (Hesmetrion of Mittel) hodel)
	-> to antimetic on address
nother-band De	· · · · · · · · · · · · · · · · · · ·
point post ()	· · · · · · · · · · · · · · · · · · ·
+ no need for contignoons 1	never
- less ellicient	
- weed 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() creat insert(H,x) extract_min(H)	E cupty hego x with min X. key	· · · · · · · · ·
find_min(H) delete(x) meld(H1,H2)-iten x decreasekey(x,k) ~ Merge	z, union Djíkstra	
Assume keys only accessed via comparisor	ns. # items in	heap
\rightarrow Thm. At least one of extract_min and insert	must take time $\Omega(\log n)$	
insert $x_1, x_2, x_3, \dots, x_n$ exctract_min n times	$> \mathcal{L}(n \log n)$	· · · · · · · · · ·
	min-hege	
Heap implementation:	(\times) X key \leq J. key	
insert and extract_min in O(log(n)) time.	Timplement: - array	
We need to traverse a path in the (balanced) tree	- pointers	Q
Meld is difficult in array-based heaps: O(n)		Å Ö
		5.6
Binomial heaps, Fibonacci heaps, Skew heaps, H	ollow heaps, Fibonacci heaps, and variants in extract_min in O(logn), other op Check which bounds are amorti	nplement delete and erations in O(1). zed 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 $a_{i-k}, a_{i-k+1},$	$a_i, a_{i+1}, \ldots, a_{i+k}$ for all $i=k+1, \ldots, n-k$	\dot{k}
window-size: 2k+1 (we need not replace	e first and last k items) $c_1 c_1 \lambda$	24.2
		21771 A
Application: removing noise	Naive algorithm: find median in each win	dow
	More efficient: use two heaps	$\rightarrow O(\mathbf{a} \mathbf{k})$ (use linear time selection)
 replace each point by median in surrounding window for some common types of noise, this is effective (corrup depends on type/amount of noise and window-size, but 	pted point unlikely to be the median) often effective in practice	

2 K+1 Kax	, Wih	
Q		Algorithm:
o o	Q Q	1. put a 1 a {2k+1} into the two heaps according to the invariant
		2. for $i=2k+2$ to n:
	H2 min boon size kul	x := find min(H2) (this is the current median)
пт пах-пеар, size к	HZ min-neap, size K+1	output x
Algorithm idea:		if a <x:< td=""></x:<>
as we go through the stream,		else:
maintain two heaps that store curr	ent window of 2k+1 items:	
- a max-heap H1 of size k	• • • • • • • • • • • •	delete $a_{i-(2k+1)}$ from heap that stores it (we have pointer to it)
- a min-heap H2 of size k+1		rectore invariant
Invariant:		$(if H1 > k, extract_max from H1 and insert into H2)$
all items in H1 are < all items in H2	2 • • • • • • • • • •	$(0 \Pi I < k, extract_nonin Hz and insert nite \Pi I)$
(So minimum of H2 is the median of H2 is the $2k+1$ items in H1 and H2)	of	
Running time: heap operations t	take O(logk) time in heaps of size \sim	~k. * * * * * * * * * * * * * * * * * * *
Initial work (step 1): O(k), as we Work for processing each item (partition $2k+1$ items, build two here $2k+1$ items, build two here $2k+1$	eaps
Total: O(nlogk)	• • • • • • • • • • •	
Exercise: show that O(nlogk	() is best possible (Hint: use media	n filter to sort)
· · · · · · · · · ·	• • • • • • • • • • •	
0 0 0 0 0 0 0 0 0 0	o o o o o o o o o o o	
· · · · · · · · · · ·	• • • • • • • • • • •	