Exercise sheet 5.

Advanced Algorithms

WiSe 2020/21

Instructor: László Kozma

Due 12:00, December 11th, 2020

Exercise 1 Trees and heaps

This exercise is optional. Its purpose is to recall some facts about basic data structures, and to think about their properties.

(a) (0 pts) Recall how a *binary heap* works, including the implementations of *insert*, *extract-min*, and building a heap in linear time. Recall how a *binary search tree* works, including the operations *search*, *insert*, and *delete*. Recall some dynamically balanced binary search tree, such as AVL- or red-black tree.

(b) (2 extra pts) Let us define the *depth* of a node x in a binary search tree as the number of edges on the path from x to the root of the tree. Define the *subtree-size* of a node x to be the number of nodes in the subtree rooted at x (not counting x itself). The subtree-size of a leaf is thus 0 and the subtree-size of the root is one less than the number of nodes in the tree. Show that in *any* binary search tree, the sum of node-depths equals the sum of subtree-sizes.
Hint: Double-counting.

(c) (2 extra pts) Recall the *rotation* operation in binary search trees. Show that given two arbitrary binary search trees T_1 and T_2 with nodes $\{1, \ldots, n\}$, there is a sequence of at most $2n$ rotations that transforms T_1 into T_2.
Hint: Rotate to some canonical state.

(d) (2 extra pts) A *treap* is a binary tree in which every node stores a *pair* of values. The nodes of the treap satisfy the binary search tree order with respect to the first value of each pair, and they satisfy the (min)heap-order with respect to the second value of each pair. Construct a treap with the following pairs of values: $(3, 5), (1, 4), (2, 8), (9, 1), (8, 3), (6, 2), (4, 7), (5, 9), (7, 6)$. Is the treap unique?

Exercise 2 Stacks and variants *8 Points*

Recall the array-based stack implementation from the lecture, with operations *create_stack*, *push*, *pop*, supported in constant amortized time.

(a) (4 pts) A *double-ended queue* ("deque") is a more powerful data structure that allows adding and removing elements at both ends of a list, i.e. it supports the operations *make_deque*, *push-right*, *push-left*, *pop-right*, *pop-left*.

Design a deque that uses memory efficiently and in which all operations take constant amortized time. Analyze the data structure rigorously with any method you prefer. You may re-use the stack construction from the lecture as a black box, or you may use arrays.

(b) (4 pts) Suppose you wanted a stack where all operations take *actual* constant time (as opposed to amortized). Describe how to modify the array-based design to achieve this. (A linked list would also work, but we insist on using arrays.)
Hint: The only "bad case" was when the array had to be doubled and elements had to be copied. Can you "spread" this work across multiple operations?

(c) (3 extra pts) The *waste* of a data structure is the difference between the number of memory cells in use and the number n of items stored in the data structure. The stack implementation discussed in the lecture has a waste of $O(n)$. Improve the design to reduce the waste to $O(\sqrt{n})$ at all times. (Make sure you account for all the extra pointers and other bookkeeping you may need in the design.)

Exercise 3 Heap variants *12 pts*

(a) (6 pts) Design an efficient data structure that allows inserting keys, and the operations *extract-small*, and *extract-large*, which extract (i.e. output and remove from the data structure) an arbitrary key that is among the smallest 25%, respectively largest 25% of the keys currently in the data structure. Show that all operations take constant amortized time.
Hint: using a few stacks and a few additional memory cells may be sufficient.

(b) (6 pts) Design an efficient data structure that supports the operations *insert* and *extract-median*, i.e. inserting a key, and returning and deleting the median of the current keys. You may use heaps as a black box. What are the running times of operations?

Exercise 4 Properties of heaps 6 *Points*

(a) (2 pts) Recall that (in the comparison model) if one of the operations *insert* and *extract-min* take time $o(\log n)$, the other type of operation must take time $\Omega(\log n)$.
Similarly argue that if *meld* takes time $O(n^{1-\varepsilon})$ for arbitrary constant $\varepsilon > 0$, then *extract-min* must take time $\Omega(\log n)$.

(b) (4 pts) Show the four properties of the binomial tree B_k observed in the lecture.
Hint: Induction.

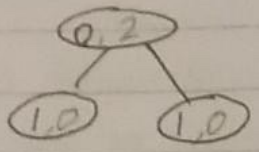
1) From: Yumeng Li and Thore Brehmer

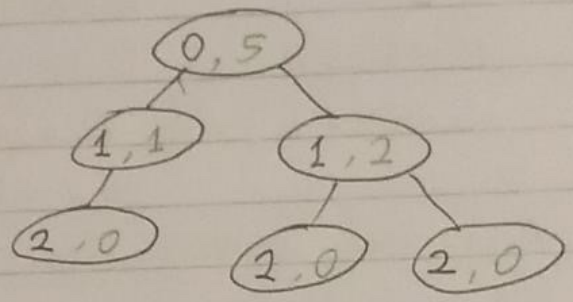
(b) (2 extra pts) Let us define the *depth* of a node x in a binary search tree as the number of edges on the path from x to the root of the tree. Define the *subtree-size* of a node x to be the number of nodes in the subtree rooted at x (not counting x itself). The subtree-size of a leaf is thus 0 and the subtree-size of the root is one less than the number of nodes in the tree. Show that in *any* binary search tree, the sum of node-depths equals the sum of subtree-sizes.
Hint: Double-counting.

(c) (2 extra pts) Recall the *rotation* operation in binary search trees. Show that given two arbitrary binary search trees T_1 and T_2 with nodes $\{1, \ldots, n\}$, there is a sequence of at most $2n$ rotations that transforms T_1 into T_2.
Hint: Rotate to some canonical state.

Ex1 b) Depth Subtree-size

do a small example





A node x with depth i contributes to the total sum of depths by i. x has i ancestors $\Rightarrow$ x contributes to the total sum of subtree-sizes by i (since it contributes by 1 to the subtree-size of each of its ancestors

total sum of depthes $= \sum_{x \text{ in tree}} \text{depth}(i) = \sum_{x \text{ in tree}}$ (x contribution to the total subtree-sizes sum)

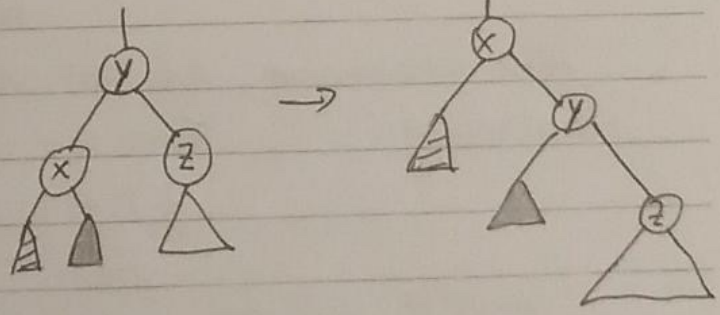
$=$ total subtree sizes sum.

c). Idea: Transforming any tree of size n to a right spine chain can be done in $\leq n$ rotations by followings:

Algo: [when there exists a left branch from y to x, perform a right rotation at y]

Note: After each iteration, we have one more right edge and one less left edge.

observe: a tree w/ n nodes has $n-1$ edges. so we are done within $(n-1)$ iterations.

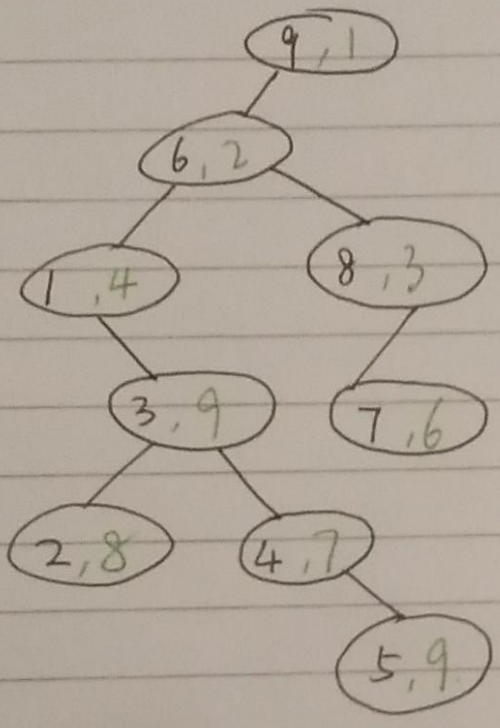


$T_1 \xrightarrow[(n-1) \text{ steps}]{\text{Algo.}}$ right spine $\xrightarrow[(n-1) \text{ step}]{(\text{Algo})^{-1}} T_2$.

$\Rightarrow$ $\leq 2(n-1)$ rotations to transform from T_1 to T_2.

(d) (2 extra pts) A *treap* is a binary tree in which every node stores a *pair* of values. The nodes of the treap satisfy the binary search tree order with respect to the first value of each pair, and they satisfy the (min)heap-order with respect to the second value of each pair. Construct a treap with the following pairs of values: $(3,5), (1,4), (2,8), (9,1), (8,3), (6,2), (4,7), (5,9), (7,6)$. Is the treap unique?

d).



The treap is unique

Exercise 2 Stacks and variants *8 Points*

Recall the array-based stack implementation from the lecture, with operations *create_stack*, *push*, *pop*, supported in constant amortized time.

(a) (4 pts) A *double-ended queue* ("deque") is a more powerful data structure that allows adding and removing elements at both ends of a list, i.e. it supports the operations *make_deque*, *push-right*, *push-left*, *pop-right*, *pop-left*.

Design a deque that uses memory efficiently and in which all operations take constant amortized time. Analyze the data structure rigorously with any method you prefer. You may re-use the stack construction from the lecture as a black box, or you may use arrays.

Operations:

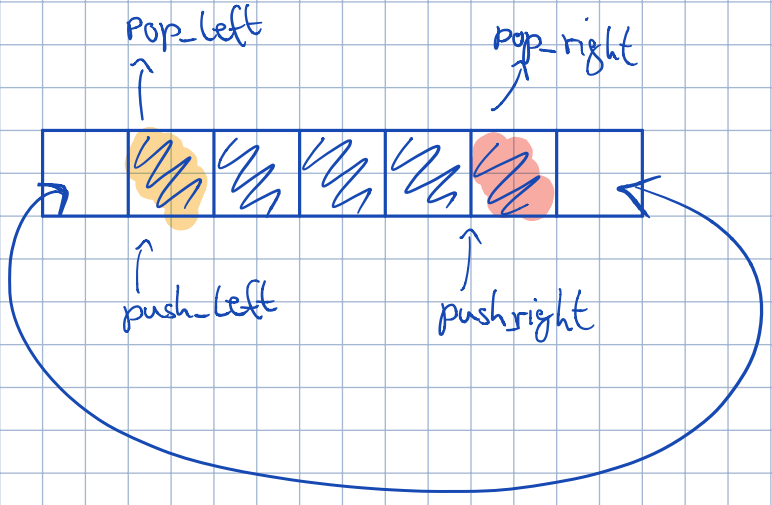
O(1) make_deque

push-right

push-left

pop-right

pop-left



Pointers:

left of Stack (as int → initialy 0)

right of Stack (as int → initialy 0)

capazity = size of array → initialy 3

n = number of elements → initialy 0

Invariant

$$\left\lfloor \frac{\text{capazity}}{4} \right\rfloor \leq n \leq \text{capazity}$$

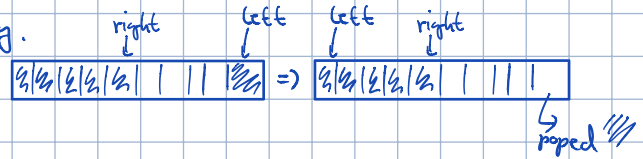
```
def pop_left(A)
    if n == 0:
        → nothing to pop.
          Stack is empty
    else:
        if left = capacity - 1:
            return A[left]
            left = 0
        else:
            return A[left]
            left += 1
```

O(1)

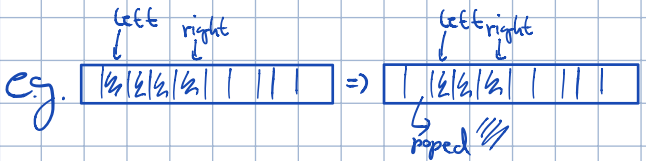
O(1)

O(1)

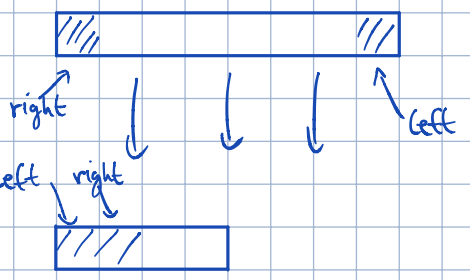
e.g.



e.g.




```
if ⌊capacity/4⌋ == n:
```

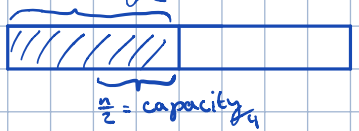


allocate array of size $\lfloor \frac{n}{2} \rfloor$

and copy all elements over in order starting from left (pointer) to right (pointer)

(pop_right is really similar to pop_left)

Obs: Then:



⇒ There have been $n/2$ pop (left or right) operations, to get to this case

To get to this case

Now:

Accounting Method

- pop will cost 3€
  - 1€ for simple operation (poping)
  - 2€ stored in item for later

⇒ $n/2$ push operations will store $2€ \cdot n/2 = n€$ total in the items

- to allocate a $\frac{\text{capacity}}{2}$ long list. There will be n copies ⇒ $n€$ cost

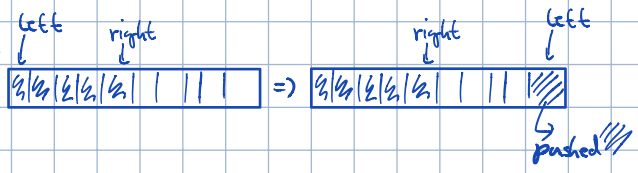
⇒ we have exactly as much € as we need ⇒ $O(n)$

```
def push_left(v, A)
    if (left == 0):
        left = capacity - 1
        A[left] = left
    else:
        left -= 1
        A[left] = left
```

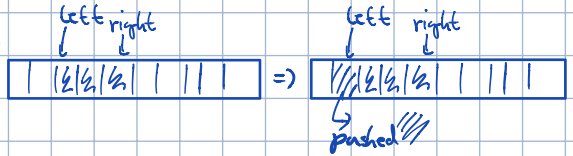
O(1)

O(1)

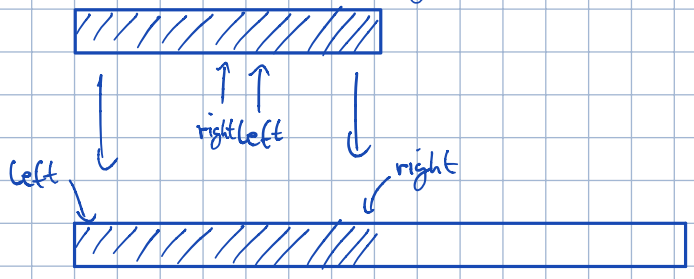
e.g.



e.g.




```
if (n == capacity):
```



allocate array of size 2n

and copy all elements over in order starting from left (pointer) to right (pointer)

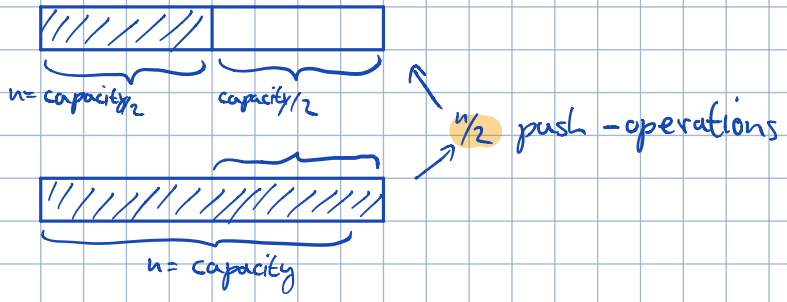
(push-right is just like push_left with minor differences)

Obs. There have been $\geq n/2$ push (left or right) operations, before the array needs to be allocated to 2n

Then:

To get to this case

Now:



Accounting Method

push will cost 3€

↳ 1€ for simple operation (pushing)

↳ 2€ stored in item for later

⇒ $n/2$ push operations will store $2€ \cdot n/2 = n€$ total in the items

- to allocate a 2-capacity long list. There will be n copies ⇒ $n€$ cost

⇒ we have exactly as much € as we need ⇒ $O(1)$

(b) (4 pts) Suppose you wanted a stack where all operations take *actual* constant time (as opposed to amortized). Describe how to modify the array-based design to achieve this. (A linked list would also work, but we insist on using arrays.)
Hint: The only "bad case" was when the array had to be doubled and elements had to be copied. Can you "spread" this work across multiple operations?

Operations

O(1) A = make_stack ( )

pop (A)

push (v, A)

Idea: when allocating a 2n long list, we dont copy all the elements from the full list to it at once.
We only copy one element from the full list to the 2n list, when we try to push a new element in it.
After we pushed n new elements to the 2n list, all n elements from the old (full) list have been copied aswell.

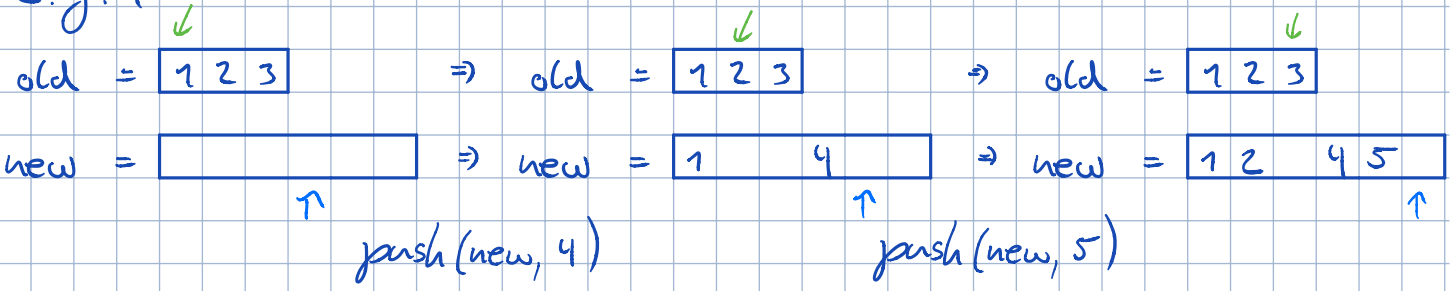
- For that we need a pointer on the full list.
→ Which starts at old_list[0]
After the pointer reached the end of the old_list → The full list can be deleted

- We also need a pointer on the new 2n list, which shows where to push new elements to.
→ Starts at new_list[n]
After the pointer reached the end of the new_list → old_list = new_list and new_list = allocate array with size 2n

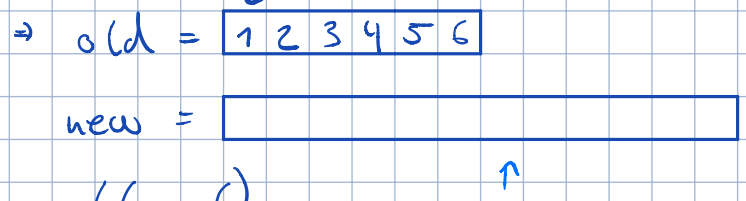
- We need to change the push operation, so that it will copy the element of the old_list pointer ⇒ O(1)
(remove the case in which it normaly copies all n elements)

- Need to add a case in pop, if we try to pop items which haven't been copied yet. Decrement the new_list pointer
⇒ still O(1)

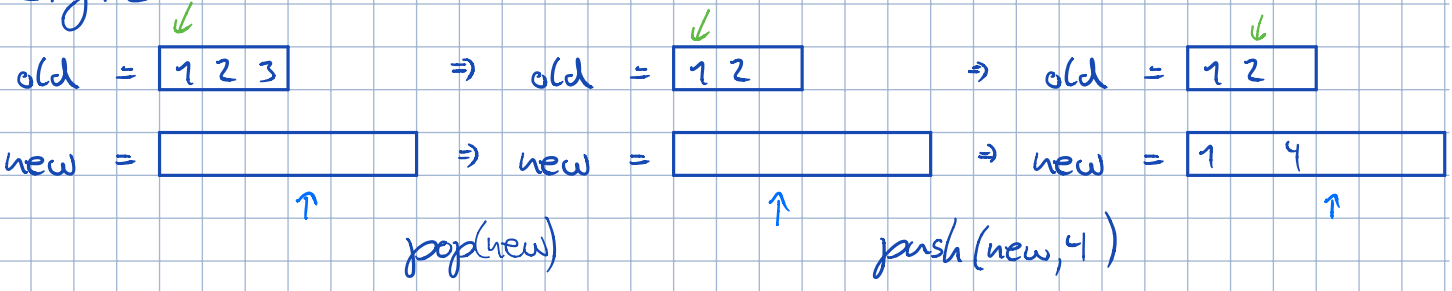
e.g. 1

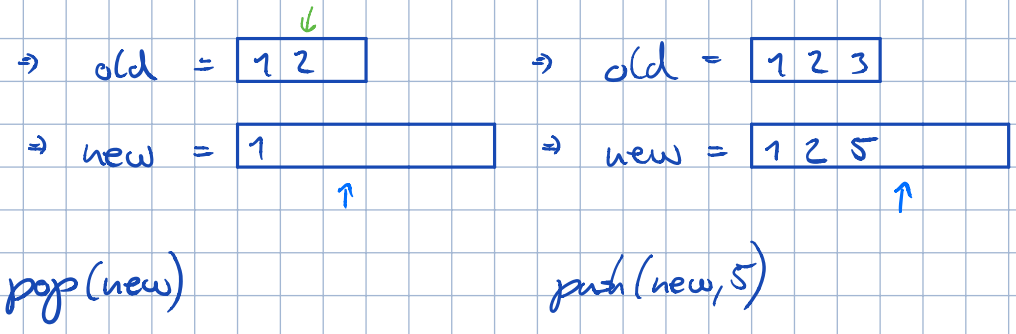


⇒ ~~1 2 3~~ deleted



e.g. 2





(c) (3 extra pts) The *waste* of a data structure is the difference between the number of memory cells in use and the number n of items stored in the data structure. The stack implementation discussed in the lecture has a waste of $O(n)$. Improve the design to reduce the waste to $O(\sqrt{n})$ at all times. (Make sure you account for all the extra pointers and other bookkeeping you may need in the design.)

Idea:

- Use Linked Lists instead of arrays.

(Increase List)

- instead of resizing the list to a list of size $2n$, we will add one more Linked List of size $\lfloor \sqrt{n} \rfloor - 1$.
  (-1 because the pointer of a Linked List takes 1 space)

(Decrease List)

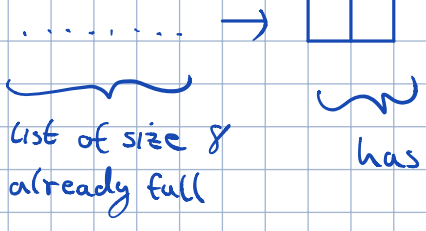
- Just delet the latest empty Linked List.

Works for $n > 8$

E.g.

for $n = 8 \ldots 10$

take 3 memory



Linked List of size 8 already full

has 0, 1 or 2 elements

=> waste will be 3, 1 or 0

0 if both cells are filled

1 if one cell is filled

3 if both cells are empty

↳ (at this point pointer will count as waste aswell)

$\sqrt{8} = 3$ waste is allowed.

Exercise 3 Heap variants *12 pts*

(a) (6 pts) Design an efficient data structure that allows inserting keys, and the operations *extract-small*, and *extract-large*, which extract (i.e. output and remove from the data structure) an arbitrary key that is among the smallest 25%, respectively largest 25% of the keys currently in the data structure. Show that all operations take constant amortized time.

3a: We use three stacks S, M, L with invariants:

$|S| \le 0.25n$, $|S| > 0$. $n :=$ current number of elements

$|L| \le 0.25n$, $|L| > 0$.

$S \le M \le L$, (i.e. elements in S are smaller than ~~that~~ those in M and elements in M are smaller than those in L).

And store "splitting" elements a, b s.t. $S \le a$, $L \ge b$ (i.e. all elements in S are smaller than a, and all elements in L are smaller than b).

And keep n up to date.

Implementation: ① insert(x): Compare x to a, if $x \le a$, push x to S; if $x \ge b$, push x to L; else, push x to M.

O(1) + Time(Rebalance).

$n \leftarrow n+1$

If $|S| > 0.25n$ or $|L| > 0.25n$, rebalance.

② extract-small: pop from S

$n \leftarrow n-1$

If $|S| = 0$ or $|L| > 0.25n$, rebalance.

③ extract-large: symmetric to ②.

④ rebalance: Pop all elements in one list

time O(n) Find elements a, b at rank $\frac{1}{8}n$ and $\frac{7}{8}n$.

Put elements $\le a$ into S

Put elements $\ge b$ into L.

rest into M.

After rebalancing, we can insert at least m elements until next rebalancing. Then we should have

$$\frac{\frac{1}{8}n+m}{n+m} \le \frac{1}{4}(n+m).$$

consider $m = \frac{1}{8}n$, this holds, so $m \ge \frac{1}{8}n$.

So Amortized time of rebalancing is $\frac{O(n)}{\frac{1}{8}n} = O(1)$.

(b) (6 pts) Design an efficient data structure that supports the operations *insert* and *extract-median*, i.e. inserting a key, and returning and deleting the median of the current keys. You may use heaps as a black box. What are the running times of operations?

Idea: use a Min-heap that stores the items ≥ median and use a Max-heap that stores the items ≤ median

- extract-median (Max_heap, Min_heap) ⇒ $O(\log n)$

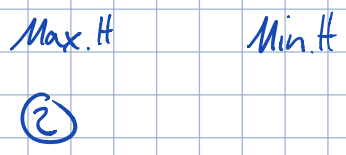
```
        if |Max-heap| > |Min-heap|
O(1)        return root of Max-heap
O(1)        remove root of Max-heap
O(log n)    balance Max-heap

        if |Max-heap| < |Min-heap|
O(1)        return root of Min-heap
O(1)        remove root of Min-heap
O(log n)    balance Min-heap

        if |Max-heap| == |Min-heap|   (⇒ elements are even, there are two
O(1)        return root of Max-heap    medians, doesent matter if we take
O(1)        remove root of Max-heap    it from Max- or Min-heap
O(log n)    balance Max-heap           we just take it from Max_heap)
```

- new_insert (x, Max_heap, Min_heap) ⇒ $O(\log n)$

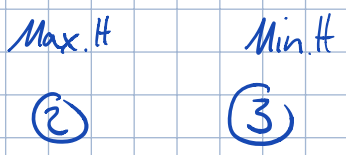
```
        if |Max-heap| == |Min-heap| == 0:   (if both are empty just insert it in
O(log n)    insert(x) in Max-heap            either one. We just take the Max-heap)

        if x ≤ root of Max_heap:
O(log n)    insert (x) in Max_heap

        if x > root of Max_heap:
O(log n)    insert(x) in Min_heap

        if ||Max_heap| - |Min_heap|| == 2:   (If the size of Max- and Min-heap
                                              differ by 2. Take the root of
            if |Max_heap| > |Min_heap|:       the bigger heap and intsert
O(log n)        insert (root of Max-heap) in Min-heap   it in the smaller heap)
O(log n)        balance Max_heap

            if |Max_heap| < |Min_heap|:
O(log n)        insert (root of Min_heap) in Max_heap
O(log n)        balance Min_heap
```

e.g. [2, 3, 7, 5, 4, 1, 6] 1 2 3 4 5 6 7

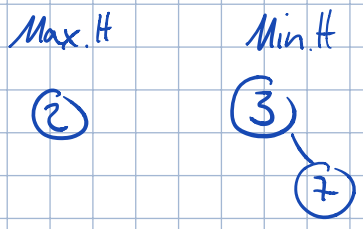
insert(2) =>



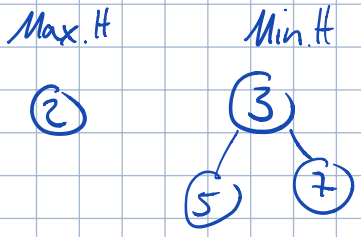
insert(3) =>



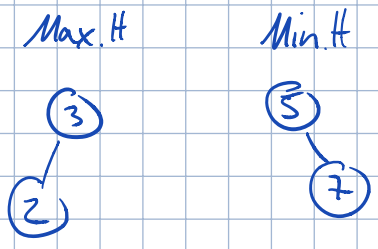
insert(7) =>



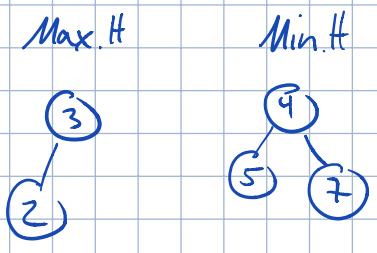
insert(5) =>



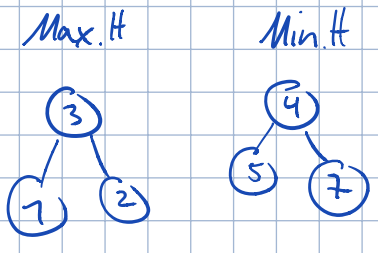
=>



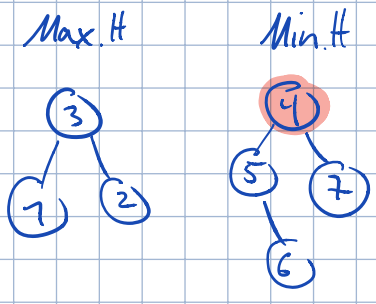
insert(4) =>



insert(1) =>



insert(6) =>



Exercise 4 Properties of heaps 6 *Points*

(a) (2 pts) Recall that (in the comparison model) if one of the operations *insert* and *extract-min* take time $o(\log n)$, the other type of operation must take time $\Omega(\log n)$.

Similarly argue that if *meld* takes time $O(n^{1-\varepsilon})$ for arbitrary constant $\varepsilon > 0$, then *extract-min* must take time $\Omega(\log n)$.

Ex4 a). Recall = insert + extract-min $\Rightarrow$ sorting.

Suppose we have n elements. We meld every two elements into a heap, each heap takes $O(1)$ time, and there are intotal $\frac{n}{2}$ heaps.

Then we meld these two-element heaps, each takes $O(2^{1-\varepsilon})$ time every two of and intotal $\frac{n}{4}$ heaps.

Do this iteration until we have an n-heap.

Time: $\frac{n}{2}O(1) + \frac{n}{4}O(2^{1-\varepsilon}) + \frac{n}{8}O((2^2)^{1-\varepsilon}) + \cdots$

$= o(\frac{n}{2}\lg n)$.

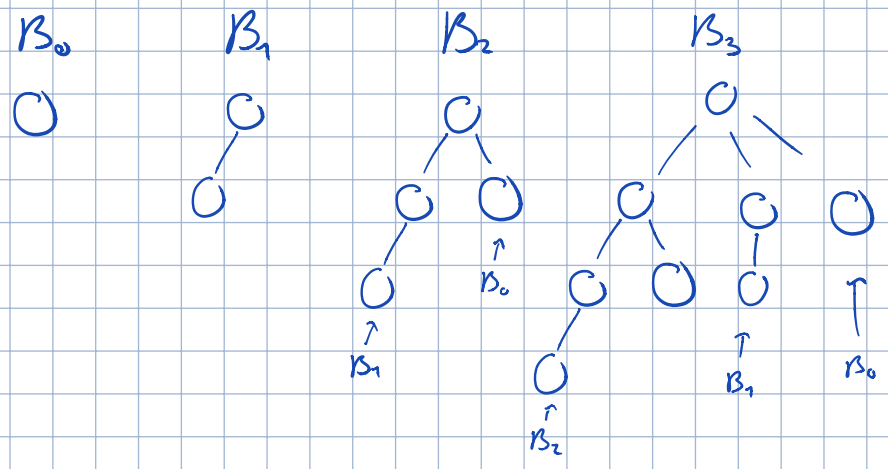
Since we have the heap, we can extract-min for n times

Then we ~~get~~ sort the n elements.

So $o(\frac{n}{2}\lg n) + T(\text{extract-min}) \cdot n \geq O(\lg n \cdot n)$.

So $T(\text{extract-min}) = \Omega(\lg n)$.

(b) (4 pts) Show the four properties of the binomial tree B_k observed in the lecture.
Hint: Induction.



Obs. B_k has 2^k nodes,
- root of degree k,
- height k
- $\binom{k}{\ell}$ nodes at depth ℓ

Proof (Exercise)

B_{k+1}: root node and children $B_{k-1}, \dots, B_0$ (in this order)

B_k has 2^k nodes

B_0 ○ ← 1 node

J.A. $k = 0$ $\quad B_0 = 2^0$ ✓

J.U. true for k

J.S. show for $k+1$

B_{k+1} has 2^{k+1} nodes

$= B_{k+1}$ has $1 + [B_k + B_{k-1} + \dots B_0]$ nodes

$= B_k + \underbrace{1 + [B_{k-1} + \dots B_0]}$

$= B_k + B_k$

J.U. $= 2^k + 2^k = 2 \cdot 2^k = 2^{k+1}$ □

B_k has root of degree k (don't really need induction for this)

J.A $k=0$ B_0 has root of degree 0 ✓

J.U. true for k

J.S. show for $k+1$

B_{k+1} has root of degree $k+1$

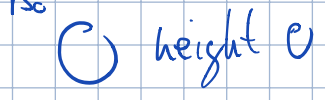
= B_{k+1} has root of degree sum of B_ks $[\underbrace{B_k, B_{k-1}, \ldots, B_0}_{k \text{ many}}]$

B_{k+1} has root of degree $1 + \sum_{i=0}^{k} 1$

B_{k+1} has root of degree $1+k$

□

B_k has height k



J.A $k=0$ B_0 has height 0 ✓

J.U. true for k

J.S. show for $k+1$

B_{k+1} has height $k+1$

= B_{k+1} has height $1 + \max_{\text{height}} [B_k\ B_{k-1} + \ldots B_0]$

J.U. = B_{k+1} has height $1 + \max_{\text{height}} [k + k-1 + \ldots 0]$

B_{k+1} has height $1+k$

□

B_k has $\binom{k}{l}$ nodes at depth k

I.A B_0 has $\binom{0}{0}$ nodes at depth 0 — B_0 ○ 1 node at depth 0

$= 1$

I.V. true for k

I.S. show for $k+1$

B_{k+1} has $\binom{k+1}{l}$ nodes at depth $k+1$

B_{k+1} = root '+' $[B_k, B_{k-1}, \dots, B_0]$ ('+' meaning as connection with the roots)

B_{k+1} = B_k (but one 'level' lower) '+' root '+' $[B_{k-1}, \dots, B_0]$

B_{k+1} = B_k (but one 'level' lower) '+' B_k

(-1 level as B_{k+1}) (same level as B_{k+1})

→ I.V. = $\binom{k}{l-1}$ + $\binom{k}{l}$

= $\binom{k+1}{l}$

(rule: $\binom{n+1}{k+1} = \binom{n}{k} + \binom{n}{k+1}$)

e.g. for 'level'