Eidgenössische
Technische Hochschule
Zürich

Ecole polytechnique fédérale de Zurich
Politecnico federale di Zurigo
Federal Institute of Technology at Zurich

Departement of Computer Science
Johannes Lengler, David Steurer
Lucas Slot, Manuel Wiedmer, Hongjie Chen, Ding Jingqiu

23 October 2023

Algorithms & Data Structures — Exercise sheet 5 — HS 23

The solutions for this sheet are submitted at the beginning of the exercise class on 30 October 2023.

Exercises that are marked by * are challenge exercises. They do not count towards bonus points.

You can use results from previous parts without solving those parts.

Sorting.

Exercise 5.1 *Sorting algorithms.*

Below you see four sequences of snapshots, each obtained in consecutive steps of the execution of one of the following algorithms: `InsertionSort`, `SelectionSort`, `QuickSort`, `MergeSort`, and `BubbleSort`. For each sequence, write down the corresponding algorithm.

3	6	5	1	2	4	8	7
3	6	5	1	2	4	8	7
3	5	6	1	2	4	8	7

3	6	5	1	2	4	8	7
3	5	1	2	4	6	7	8
3	1	2	4	5	6	7	8

3	6	5	1	2	4	8	7
3	6	1	5	2	4	7	8
1	3	5	6	2	4	7	8

3	6	5	1	2	4	8	7
3	6	5	1	2	4	7	8
3	6	5	1	2	4	7	8

Solution:

InsertionSort (top left) – BubbleSort (top right) – MergeSort (bottom left) – SelectionSort (bottom right).

Exercise 5.2 *Guessing an interval* **(1 point)**.

Alice and Bob play the following game:

- Alice selects two integers $1 \leq a < b \leq 200$, which she keeps secret.
- Then, Alice and Bob repeat the following:
  - Bob chooses two integers $0 \leq a' < b' \leq 201$.
  - If $a = a'$ and $b = b'$, Bob wins.

- If $a' < a$ and $b < b'$, Alice tells Bob ‘my numbers are strictly between your numbers!’. A previous version had the mistake that Alice gave information to Bob when $a < a'$ and $b' < b$, which has now been corrected to $a' < a$ and $b < b'$.

- Otherwise, Alice does not give any clue to Bob.

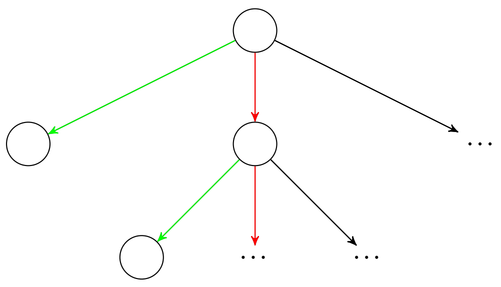
(a) Bob claims that he has a strategy to win this game in 12 attempts at most. Prove that such a strategy cannot exist.

Hint: *Represent Bob’s strategy as a decision tree. Each edge of the decision tree corresponds to one of Alice’s answers, while each leaf corresponds to a win for Bob.*

Hint: *After defining the decision tree, you can consider the sequence $k_0 = 1$ and $k_n = 2k_{n-1} + 2$ for $n \geq 1$, and prove that $k_n = 3 \cdot 2^n - 2$ for any $n \in \mathbb{N}_0 = \mathbb{N} \cup \{0\}$. The number of vertices in the decision tree should be related to k_n.*

Solution:

Bob’s strategy can be represented as follows, where green arrows correspond to a win, red arrows to ‘my numbers are strictly between your numbers!’, and black arrows to the absence of a clue. The vertices that are not leaves in this tree correspond to the guesses he makes.



Each vertex of the corresponding tree has at most three children, of which one (corresponding to Bob winning the game) has no child, while the two others can again have three children with the same structure as their parent. Note that not all vertices need to have exactly three children since it is possible that after a sequence of guesses Bob makes, not all answers of Alice are still possible for the next guess. For example, if he figured out the numbers of Alice with certainty after some number of steps, the next vertex has only one child (the one corresponding to Bob winning). The depth of the tree is the number of guesses Bob needs to make in the worst case.
Denoting by k_n the maximum number of vertices in a tree of depth $n \in \mathbb{N}_0$ of the above form, we see that

$$\begin{cases} k_0 = 1 \\ k_n = 2k_{n-1} + 2 \quad \forall n \geq 1. \end{cases}$$

The second equality is true since for a tree of depth n for $n \geq 1$, we have the root and a leaf (endpoint of the green edge) as well as two subtrees of depth $n-1$ of the same form (rooted at the endpoints of the red and black edges). We will now prove by induction that, for all $n \in \mathbb{N}_0$, we have $k_n = 3 \cdot 2^n - 2$.

- **Base Case.**
  For $n = 0$, we have $k_0 = 1 = 3 \cdot 2^0 - 2$, so the base case holds.

- **Induction Hypothesis.**
  Assume that the statement holds for $j \in \mathbb{N}$, i.e., $k_j = 3 \cdot 2^j - 2$.

- **Inductive Step.**
  We compute
  $$k_{j+1} = 2k_j + 2 = 2 \cdot (3 \cdot 2^j - 2) + 2 = 3 \cdot 2^{j+1} - 4 + 2 = 3 \cdot 2^{j+1} - 2.$$
  Thus, the statement also holds for $j+1$. By the principle of mathematical induction, we have $k_n = 3 \cdot 2^n - 2$ for any $n \in \mathbb{N}_0$.

Next, we want to count the number of pairs Alice can choose. Once she has chosen b, she has $b-1$ possibilities for a (the numbers in the set $\{1, 2, \ldots, b-1\}$). Thus, the total number of pairs Alice can choose is
$$\sum_{b=1}^{200}(b-1) = \left(\sum_{b=1}^{200} b\right) - 200 = \frac{200 \cdot 201}{2} - 200 = 19900,$$
where the second equality uses $\sum_{i=1}^{n} i = \frac{n(n+1)}{2}$ for any $n \in \mathbb{N}$, which was proven in exercise 0.1. In order for Bob's strategy to allow him to win for any pair of integers chosen by Alice, the tree representing his strategy must have at least 19900 leaves (one for each choice of Alice). If Bob's statement is true (i.e. he wins after at most 12 turns), this tree has depth at most 12 and therefore at most k_{12} vertices. Since $k_{12} = 12286 < 19900$, the decision tree corresponding to Bob's strategy cannot have 19900 leaves, hence Bob cannot certainly win in at most 12 attemps.

(b)* Can Bob have a strategy to win the game in 13 or 14 attempts?

Hint: *Follow the same strategy as for (a). After defining the decision tree, try to analyse the number of leaves in the decision tree corresponding to Bob's strategy. The sequence* $\ell_1 = 1$ *and* $\ell_n = 2\ell_{n-1} + 1$ for $n > 1$, for which you can prove $\ell_n = 2^n - 1$ for any $n \in \mathbb{N}$, might be helpful.

Solution:

We want to prove that Bob cannot have a strategy for 13 or 14 attemps. We follow the same proof strategy as for part (a) and represent Bob's strategy with the same decision tree. However, we now want to determine the number of leaves of this tree and not the number of vertices in total. Denoting by ℓ_n the maximum number of leaves in a tree of depth $n \in \mathbb{N}$ of the above form, we see that
$$\begin{cases} \ell_1 = 1 \\ \ell_{n+1} = 2\ell_n + 1 \quad \forall n > 1. \end{cases}$$
The second equality holds because for a tree of the above form with depth $n+1$ (for $n > 1$) we have one leaf with depth 1 (the endpoint of the green edge starting at the root) and all other leafs are in one of the subtrees rooted at the endpoints of the red and blue edges starting at the root. These subtrees are trees of the above form with depth n, so we get the formula $\ell_{n+1} = 2\ell_n + 1$ for $n > 1$. We will now prove by induction that, for all $n \in \mathbb{N}$, we have $\ell_n = 2^n - 1$.

- **Base Case.**
  For $n = 1$, we have $\ell_1 = 1 = 2^1 - 1$, so the base case holds.
- **Induction Hypothesis.**
  Assume that the statement holds for $j \in \mathbb{N}$, i.e., $\ell_j = 2^j - 1$.
- **Inductive Step.**
  We compute
  $$\ell_{j+1} = 2\ell_j + 1 = 2 \cdot (2^j - 1) + 1 = 2^{j+1} - 2 + 1 = 2^{j+1} - 1.$$
  Thus, the statement also holds for $j+1$. By the principle of mathematical induction, the statement $\ell_n = 2^n - 1$ is true for any $n \in \mathbb{N}$.

As for part (a), Alice has 19900 choices, so Bob's decision tree needs to have at least 19900 leaves. Thus, if Bob has a strategy to win in at most 14 attempts, we must have $\ell_{14} \geq 19900$. Now, $\ell_{14} = 2^{14} - 1 = 16383 < 19900$, hence Bob also cannot certainly win in at most 14 (or 13) attempts.

Exercise 5.3 *Building a Heap* **(1 point)**.

Recall that a binary tree is called *complete* if all of its layers are fully filled, except possibly the last layer, which should be filled from left to right. A *(max-)heap* is a complete binary tree with the extra property that for any node C with parent P,

$$\text{key}(P) \geq \text{key}(C). \qquad \text{(heap-condition)}$$

In this exercise, we formally prove the correctness of the following algorithm from the lecture, which adds a new node with key k to an existing, non-empty heap H. We will show that it performs at most $O(\log n)$ comparisons between keys, where n is the number of nodes in the heap H, and that it maintains the heap structure.

Algorithm 1 Heap insertion

```
function INSERT($H, k$)
    Add a new node $N$ with key $k$ to the bottom layer of $H$, in the left-most free position. If the
    bottom layer is full, instead create a new layer and add the node in the left-most position.
    $P \leftarrow$ the parent of $N$
    while $\text{key}(P) < \text{key}(N)$ do                    ▷ $N$ violates the heap-condition
        swap the keys of node $N$ and $P$.
        $N \leftarrow P$
        if $N$ is the root node then
            stop
        else
            $P \leftarrow$ the parent of $N$
```

Let H be a heap consisting of $n \geq 1$ nodes, and let $k \in \mathbb{N}$. Let H' be the data structure that results from executing $\text{Insert}(H, k)$.

(a) Prove that at most $O(\log n)$ comparisons between keys are performed in the execution of $\text{Insert}(H, k)$.

Hint: *After each iteration of the while-loop, what can you say about the* depth *of the node N?*

Solution:

As H is a complete binary tree on n nodes, it has at most $T \leq O(\log n)$ layers. At the start of the algorithm, the depth of the node N is at most $T + 1$. In every iteration of the while loop, the depth of N is decreased by one. As the algorithm terminates when N is the root node (i.e., has depth 0), at most $T+1$ iterations of the while-loop are executed, each one performing one comparison between keys.

(b) Let N_{stop} be the final node considered by the algorithm. Prove that all nodes in H' with depth less than or equal to $\text{depth}(N_{\text{stop}})$ satisfy the heap-condition. (A node N satisfies the heap-condition if it is the root node, or otherwise if $\text{key}(N) \leq \text{key}(\text{parent(N)})$.)

Hint: *Use the fact that H was a heap before we inserted the new node. Consider separately the two different reasons for the algorithm to terminate.*

Solution:

Suppose that the algorithm terminated because N_{stop} was the root node. Then there are no other nodes at depths less than or equal to $\text{depth}(N_{\text{stop}})$, and so there is nothing to check. So assume the algorithm terminated because $\text{key}(P) \geq \text{key}(N_{\text{stop}})$. Note that, by definition, N_{stop} then satisfies the heap-condition. Furthermore, none of the other nodes at depth less than or equal to $\text{depth}(N_{\text{stop}})$ had their keys changed. Since H was a heap before executing the algorithm, they thus satisfy the heap-condition.

(c) Let N_{stop} be the final node considered by the algorithm. Prove that all nodes in H' with depth strictly greater than $\text{depth}(N_{\text{stop}})$ satisfy the heap-condition. Using (b), conclude that H' is a heap.

Hint: *Let T be the depth of H'. Use induction to show that after t iterations of the while-loop, the heap-condition is satisfied by all nodes with depth strictly greater than $T - t$.*

Hint: *After swapping the keys of nodes N and P in an iteration of the while-loop, which nodes might potentially no longer satisfy the heap-condition?*

Solution:

We start by proving the statement of the first hint by induction. For $\ell \in \mathbb{N}$, we write $H_{>\ell}$ for the set of nodes in H at depth strictly greater than ℓ.
Base case: For $t = 0$, there are no nodes in $H_{>T-t} = H_{>T}$, and so there is nothing to show.
Induction hypothesis: After $t - 1$ iterations of the while-loop, the heap-condition is satisfied by all nodes in $H_{>T-(t-1)}$.
Induction step: Let N and P be the nodes whose keys are swapped in the t-th iteration of the while-loop. Using the induction hypothesis, all nodes in $H_{>T-(t-1)}$ satisfied the heap-condition before this swap was made. After the swap, any node in $H_{>T-(t-1)}$ whose parent's key is changed must be a child of N. The keys of these nodes are thus the keys of (grand)children of P (in the original heap H). We know by the transitive property that $\text{key}(P)$ is greater than or equal to the keys of its (grand)children. We conclude all nodes in $H_{>T-(t-1)}$ satisfy the heap-condition. It remains to consider nodes at depth exactly $T - (t - 1)$. Here, the only nodes whose parent had their key changed during the execution of the algorithm so far are the children of P; namely N and a potential second child C. Note that $\text{key}(N) \geq \text{key}(P) \geq \text{key}(C)$ before the swap was made. Therefore, after the swap, we must have $\text{key}(P) \geq \text{key}(N)$ and $\text{key}(P) \geq \text{key}(C)$. That is, N and C satisfy the heap-condition after the swap is made.

To conclude, it remains to note that $\text{depth}(N_{\text{stop}}) = T - t_{\text{stop}}$, where t_{stop} is the total number of iterations of the while-loop in the execution of $\text{Insert}(H, k)$.

Data structures.

Exercise 5.4 *Implementing abstract data types.*

In the lecture, you saw how we can implement the abstract data type list with operations insert, get, delete and insertAfter. In this exercise, the goal is to see how we can implement two other abstract data types, namely the stack (german "Stapel") and the queue (german "Schlange" or "Warteschlange"). The abstract data type stack is, as the name suggests, a stack of elements. For a stack S, we want to implement the two following operations; see also Figure 1.

- $\text{push}(x, S)$: Add x on top of the stack S.

- pop(S): Remove (and return) the top element of the stack S.

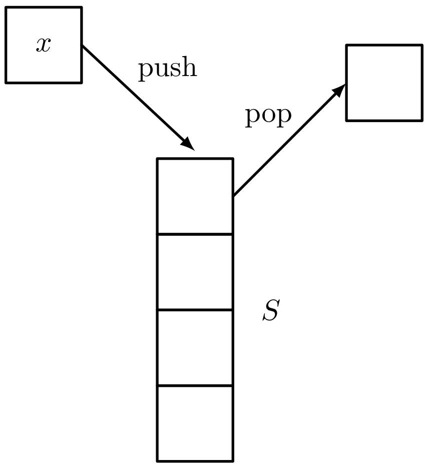


Figure 1: Abstract data type stack

The abstract data type queue is a queue of elements. For a queue Q, we want to implement the following two operations; see also Figure 2.

- enqueue(x, Q): Add x to the end of Q.
- dequeue(Q): Remove (and return) the first element of Q.

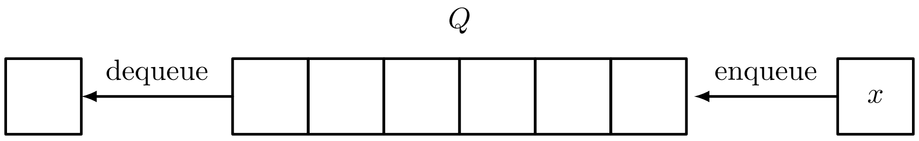


Figure 2: Abstract data type queue

(a) Which data structure from the lecture can be used to implement the abstract data type stack efficiently? Describe for the operations push and pop how they would be implemented with this data structure and what the run time would be.

Solution:

We can use a linked list to implement a stack. The elements of the stack are saved as the keys of the linked list in the same order as in the stack, where the first element of the list is the top element of the stack. The operation push(x, S) adds a new element at the start of the list with key x and pointer to the old start of the list. The pointer to the new start of the list is then a pointer to the newly created element. The operation pop(S) accesses the first element of the list (we have a pointer to this element) and returns it. We then set the pointer that saves the start of the list to the pointer that is stored in the current first element. After this, we can delete the first element. For both push(x, S) and pop(S), we only need to do a constant number of operations, so the run time is $O(1)$.

(b) Which data structure from the lecture can be used to implement the abstract data type queue efficiently? Describe for the operations enqueue and dequeue how they would be implemented with this data structure and what the run time would be.

Solution:

We can use a doubly linked list to implement a queue. The elements of the queue are saved as the keys of the doubly linked list in the same order as in the queue. We assume that the pointer to the start of the list points to the first element in the queue and the pointer to the end of the list to the last element in the queue. The operation $\text{enqueue}(x, Q)$ adds the new element at the end of the list using the pointer to the end of the list. The operation $\text{dequeue}(Q)$ accesses, returns and deletes the first element in the queue using the pointer to the beginning of the list. For both operations we then adjust the pointers accordingly, similar as we did in part (a) for the stack. The pointers we need to change are the pointers of the last and the newly added element (for $\text{enqueue}(x, Q)$) and of the second element (for $\text{dequeue}(Q)$) as well as the pointers to the start and end of the list. Since we have pointers in both directions, we can access and change these elements in constant time. Thus, both operations $\text{enqueue}(x, Q)$ and $\text{dequeue}(Q)$ have run time $O(1)$.

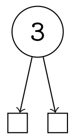
Remark: The following exercise 5.5 is related to the content of the lecture on Tuesday, October 24.

Exercise 5.5 *AVL trees* **(1 point)**.

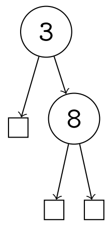
(a) Draw the tree obtained by inserting the keys 3, 8, 6, 5, 2, 9, 1 and 0 in this order into an initially empty AVL tree. Give also all the intermediate states after every insertion and before and after each rotation that is performed during the process.

Solution:

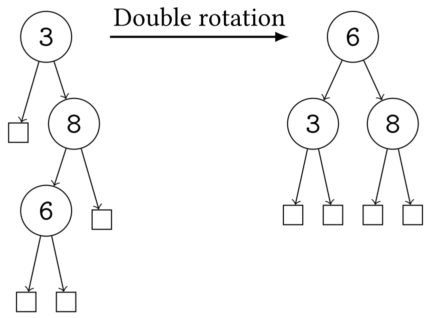
Insert 3:



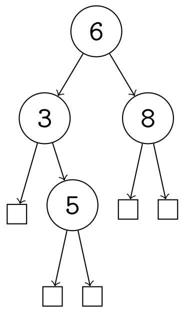
Insert 8:



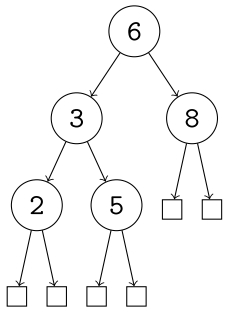
Insert 6:



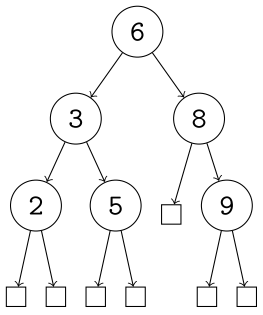
Insert 5:



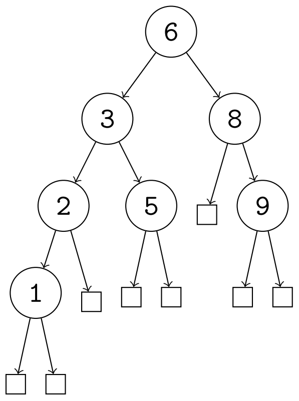
Insert 2:



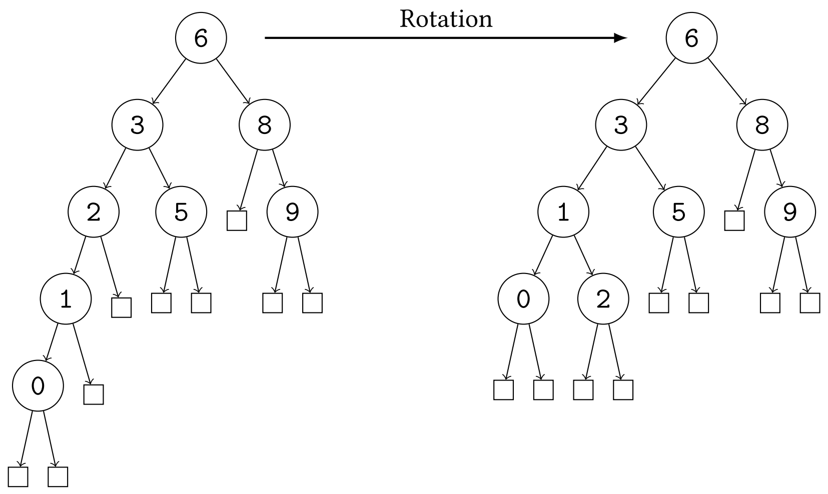
Insert 9:



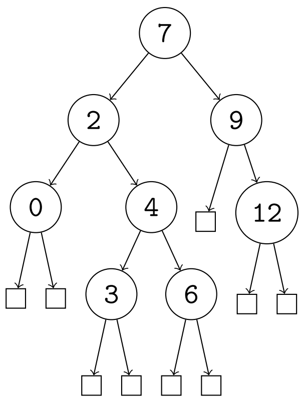
Insert 1:



Insert 0:



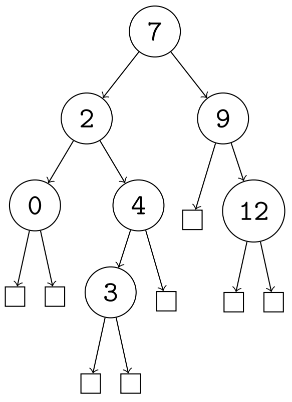
(b) Consider the following AVL tree.



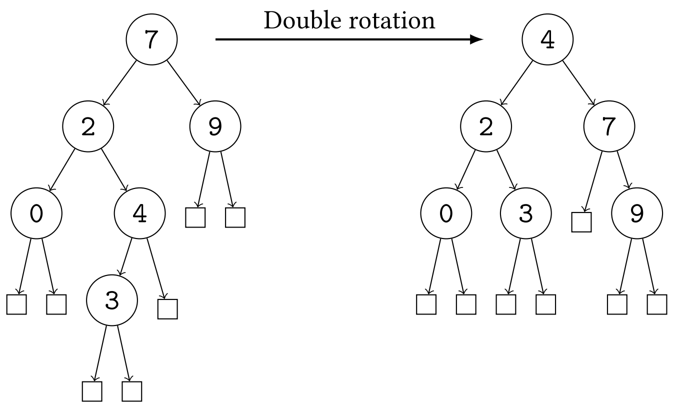
Draw the tree obtained by deleting 6, 12, 7 and 4 in this order from this tree. Give also all the intermediate states after every deletion and before and after each rotation that is performed during the process.

Solution:

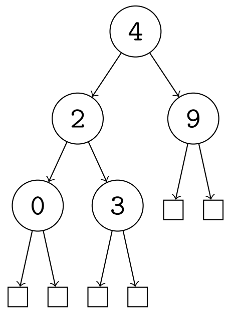
Delete 6:



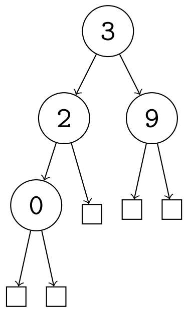
Delete 12:



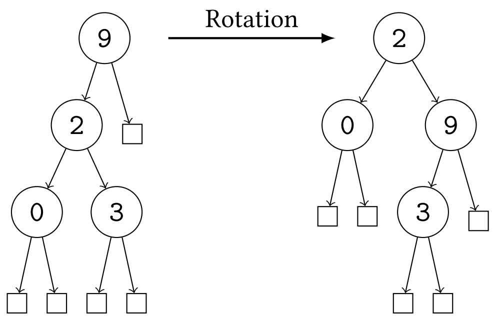
Delete 7:



Delete 4: Key 4 can either be replaced by its predecessor key, 3, or its successor key, 9. If key 4 is replaced by its predecessor:



If key 4 is replaced by its successor:


Rotation
9
2
0
3
2
0
9
3