Departement of Computer Science
Johannes Lengler, David Steurer
Kasper Lindberg, Lucas Slot, Hongjie Chen, Manuel Wiedmer

21 October 2024

Algorithms & Data Structures Exercise sheet 5 HS 24

The solutions for this sheet are submitted on Moodle until 27 October 2024, 23:59.

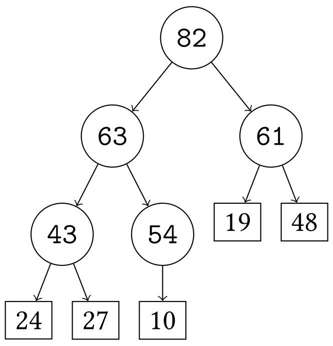
Exercises that are marked by * are challenge exercises. They do not count towards bonus points.

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

The solutions are intended to help you understand how to solve the exercises and are thus more detailed than what would be expected at the exam. All parts that contain explanation that you would not need to include in an exam are in grey.

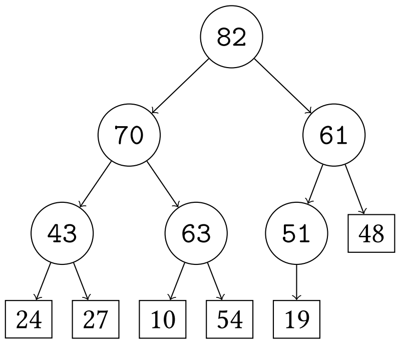
Exercise 5.1 *Max-Heap operations* **(1 point)**.

(a) Consider the following max-heap:

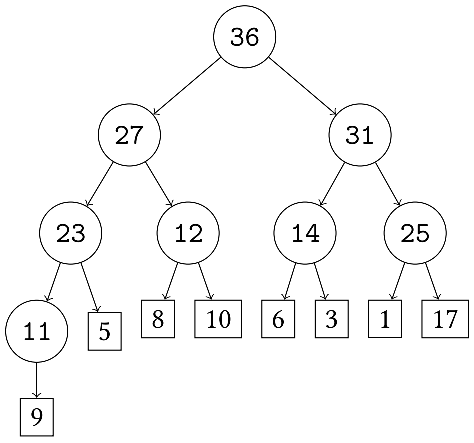


Draw the max-heap after inserting the elements 70 and 51 in that order.

Solution:

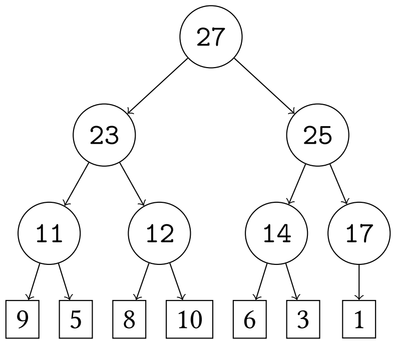


(b) Consider the following max-heap:



Draw the max-heap after two ExtractMax operations.

Solution:



Guidelines for correction:

Award 1/2 point per correctly solved part.

Exercise 5.2 *Guessing an interval.*

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

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

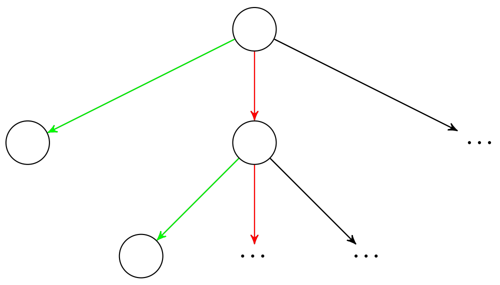
Hint: *Represent Bob's strategy as a decision tree. Each edge of the decision tree corresponds to one of Alice's answers, while each leaf corresponds to a win for Bob.*

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:

The solution is divided into three parts: the construction of the decision tree, proving an upper bound for the number of vertices in the tree and an argument about the non-existence of the required strategy. We will show that Bob cannot cover all possible choices for a and b in twelve steps by comparing the possible scenarios covered in the tree and the choices.

Construction of decision tree: 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. 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).

Upper bound for the number of vertices: The tree contains all possible scenarios of the game. The depth 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 $j \in \mathbb{N}_0$, 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 $n \in \mathbb{N}_0$.

Non-existence of the required strategy: We want to compare the number of pairs Alice can choose and Bob can determine in 12 steps. Once Alice 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

$$\sum_{b=1}^{200}(b-1) = \left(\sum_{b=1}^{200} b\right) - 200 = \frac{200 \cdot 201}{2} - 200 = 19900.$$

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 attempts.

Exercise 5.3 *Quick(?) sort* **(1 point)**.

Recall the pseudocode for the *quick sort* algorithm from the lecture:

Algorithm 1 quick sort

```
1: function QuickSort($A$, $\ell$, $r$)
2:     if $\ell < r$ then
3:         $k$ = Partition($A, \ell, r$)
4:         QuickSort($A, \ell, k-1$)
5:         QuickSort($A, k+1, r$)
6: function Partition($A, \ell, r$)
7:     $i \leftarrow \ell$
8:     $j \leftarrow r - 1$
9:     $p \leftarrow A[r]$                          ▷ Choose the rightmost entry as pivot
10:    repeat
11:        while $i < r$ and $A[i] \leq p$ do
12:            $i \leftarrow i + 1$
13:        while $j > \ell$ and $A[j] > p$ do
14:            $j \leftarrow j - 1$
15:        if $i < j$ then
16:            Swap $A[i]$ and $A[j]$
17:    until $i > j$
18:    Swap $A[i]$ and $A[r]$                ▷ At the end, the correct place for the pivot is $i$
19:    Return $i$
```

We want to study the number of comparisons between array entries the quick sort algorithm performs when we apply it to an array $A[1 \ldots n]$ consisting of n unique integers which is already sorted in ascending order (so $A[1] < A[2] < \ldots < A[n]$).

(a) Show that the number of comparisons $T(n)$ between array entries that QuickSort$(A, 1, n)$ performs when applied to a sorted array A as above, and with the above rule to select the pivot satisfies the recursive relation

$$T(1) = 0, \quad T(n) = T(n-1) + (n-1) \quad \forall n \geq 2.$$

You may assume for simplicity that PARTITION(A, ℓ, r) always performs exactly $r - \ell$ comparisons between entries. In your argument, refer to the pseudocode above.

Solution:

We note that, if A is a sorted array with unique entries, then for any ℓ, r with $\ell < r$, the function PARTITION(A, ℓ, r) does not perform any SWAP operations. Indeed, the pivot $A[r]$ is strictly larger than all other entries of $A[\ell \dots r]$. Therefore, at the end of the two WHILE-loops in PARTITION, the index i will equal r, and the index j will equal $r-1$. So, the condition of the IF-statement will never be met. After the REPEAT $-$ UNTIL, the index i will be equal to r, and so 'swapping' $A[i]$ and $A[r]$ at the end does not change anything.

Now, consider the call QUICKSORT$(A, 1, n)$, with $n \geq 2$. First, PARTITION$(A, 1, n)$ is called. As we have seen, this will return $k = n$, and make no changes to A. By assumption, it performs exactly $n-1$ comparisons between entries. Then, QUICKSORT$(A, 1, n-1)$ and QUICKSORT$(A, n+1, n)$ are called. The latter call immediately terminates. As A has remained sorted, the former corresponds exactly to calling QUICKSORT$(\widetilde{A}, 1, n-1)$ on a sorted array $\widetilde{A}$ of length $n-1$. We conclude that $T(n) = T(n-1) + (n-1)$.

Finally, we note that the call QUICKSORT$(A, 1, 1)$ immediately terminates, showing that $T(1) = 0$.

(b) Assume $n \geq 3$. Show that $T(n) = \Theta(n^2)$. To do so, first give an exact expression for $T(n)$ based on the recursive formula of part (a) (your exact expression does not need to be maximally simplified, e.g., it is allowed to contain summation-symbols).

Hint: *Based on the recursive formula from part (a), how could you write $T(n)$ in terms of $T(n-2)$? How could you write it in terms of $T(n-3)$? Repeat this process.*

Solution:

By 'telescoping' the recursive relation from part (a), and using $T(1) = 0$, we find that

$$T(n) = T(n-1) + (n-1) = T(n-2) + (n-2) + (n-1) = \ldots = \sum_{i=1}^{n} (n-i) = \sum_{j=0}^{n-1} j.$$

Now, we see that $\sum_{j=0}^{n-1} j \leq \sum_{j=0}^{n-1} n \leq O(n^2)$. On the other hand,

$$\sum_{j=0}^{n-1} j \geq \sum_{j=\lfloor n/2 \rfloor}^{n-1} j \geq \sum_{j=\lfloor n/2 \rfloor}^{n-1} \lfloor n/2 \rfloor \geq \left(\lfloor n/2 \rfloor - 1\right)^2 \geq \Omega(n^2).$$

See also Exercise 1.2. Alternatively, you could find an exact expression for $\sum_{j=0}^{n-1} j$ using induction.

Guidelines for correction:

Award 1/2 point per correctly solved part.

(a) It is important that the argument refers to the pseudocode. It is not enough to just claim for instance that PARTITION makes no changes to A. If the only mistake is that the case $T(1) = 0$ is not mentioned explicitly, award 1/2 point.

(b) The telescoping argument is the most important part. From the exact expression, a very formal argument that $T(n) = \Theta(n^2)$ is not needed. For instance, a reference to an earlier exercise would suffice.

Exercise 5.4 *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,

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

Also recall that for a tree T, the root is at level 0 and the leaves are at level $\text{height}(T)$; for a node at level ℓ, its children are at level $\ell + 1$.

In this exercise, we formally prove the correctness of the following algorithm from lecture, which converts any complete binary tree into a heap.

Algorithm 2 Heap Construction

```
function HEAPIFY(T)
    for $t = \text{height}(T) - 1, \dots, 0$ do
        for nodes $N$ at level $t$ do
            for $\ell = t, \dots, \text{height}(T) - 1$ do
                $C_1 \leftarrow$ the left child of $N$, if no such child exists assign it key $-\infty$.
                $C_2 \leftarrow$ the right child of $N$, if no such child exists assign it key $-\infty$.
                if $\text{key}(C_1) \geq \text{key}(C_2)$ and $\text{key}(C_1) > \text{key}(N)$ then
                    Swap the keys of nodes $N$ and $C_1$.
                    $N \leftarrow C_1$
                else if $\text{key}(C_1) < \text{key}(C_2)$ and $\text{key}(C_2) > \text{key}(N)$ then
                    Swap the keys of nodes $N$ and $C_2$.
                    $N \leftarrow C_2$
                else
                    Exit inner for loop
```

Let T be a complete binary tree consisting of n nodes with $n \geq 2$. Let H be the data structure that results from executing $\text{Heapify}(T)$.

(a) Prove that the executing $\text{Heapify}(T)$ returns a valid heap.

__Hint:__ Use the invariant $I(t)$ for $0 \leq t \leq \text{height}(T)$: all nodes from levels $\text{height}(T), \dots, t$ satisfy the heap condition, namely $\text{key}(P) \geq \text{key}(C)$ where P is the parent node of level at least t, and C is a child of P.

Solution:

We prove the invariant in the hint by mathematical induction on t (going in the opposite direction of standard induction).

- **Base Case.**
  We prove the statement $I(t)$ is true for $t = \text{height}(T)$. We have that all nodes from level $\text{height}(T)$ are leaves and hence are not parent nodes. Thus the invariant holds vacuously.

- **Inductive Hypothesis.**
  We assume the statement $I(t)$ is true for some $t \in \mathbb{N}$, $\text{height}(T) \geq t > 0$, i.e. after $\text{height}(T) - t$ iterations of the outermost loop.

- **Inductive Step.**
  We must show the statement $I(t-1)$ also holds.

By the inductive hypothesis all nodes from levels $\text{height}(T), \ldots, t$ satisfy the heap condition.

Now consider a node N at level $t-1$. If it has no children then the node satisfies the heap condition and we are done.

Otherwise, let C_1 be its left child and C_2 its right child if it exists, otherwise we assume $\text{key}(C_2) = -\infty$. Our algorithm swaps they key of N with the key of the larger child if we satisfy $\text{key}(C_1) > \text{key}(N)$ or $\text{key}(C_2) > \text{key}(N)$. Thus N now satisfies the heap condition but the swapped child may not. The innermost for loop however follows the swapped child and repeats this process until either we have processed a node at level $\text{height}(T) - 1$ or the node satisfies the heap condition. Since nodes at level $\text{height}(T)$ are leaf nodes and not parents, we do not need to consider these nodes. Thus any modified node along these sequence of swaps still satisfy the heap condition and this includes our original node N.

Since we do this process on all nodes at level $t-1$, this shows that the algorithm after this loop iteration has all nodes from levels $\text{height}(T), \ldots, t-1$ satisfying the heap condition. Thus $I(t-1)$ holds.

By the principle of mathematical induction, $I(t)$ is true for all $t \in \mathbb{N}$, $\text{height}(T) \geq t \geq 0$. In particular, $I(0)$ holds, which means that after the first $\text{height}(T)$ iterations of the outer loop, the nodes from levels $\text{height}(T), \ldots, 0$ satisfy the heap condition. In particular, all nodes satisfy the heap condition. This shows that after $\text{height}(T)$ steps the binary tree is now a heap, which shows correctness of the Heapify algorithm.

Guidelines for correction:

Award 1/2 point if base case and inductive hypothesis is correct, while inductive step is attempted and on the right track but not fully correct. Award 1 point if inductive step is also correct.

(b)* Prove that the runtime of executing $\text{Heapify}(T)$ takes time $O(n)$.

You may use the fact that for any $k \in \mathbb{N}$

$$\sum_{i=1}^{k} \frac{i}{2^i} \leq 2\,.$$

Hint: *Write the runtime as an outer sum over the various levels and an inner sum over all the nodes of that level.*

Solution:

Define $f(t) = \text{height}(T) - t$. For each node N, the second for loop says that our algorithm processes the subtree rooted at N exactly once. Lets analyze the runtime of the inner most loop on this subtree.

Say our node N is at level t, then for its two children, we check if either are bigger than N and if so, swap with the bigger one. Then we make this swapped child our new N and repeat downwards until we either hit the bottom of the tree or our node satisfies the heap condition. In the worst case, we hit the bottom of the tree, and since each intermediate height used only a constant number of operations, we executed $O(f(t))$ number of comparisons and swaps. Thus for each node N at level t, we use at most $O(f(t))$ steps.

Thus the total runtime is

$$\sum_{t=0}^{\text{height}(T)-1} \sum_{N\,:\,\text{node of level } t} O(f(t)) \leq \sum_{t=0}^{\text{height}(T)} \sum_{N\,:\,\text{node of level } t} c \cdot f(t) = c \sum_{t=0}^{\text{height}(T)} \sum_{N\,:\,\text{node of level } t} f(t)$$

for some $c \in \mathbb{R}_+$.

To further simplify this formula, we can bound the number of nodes at each level. For a binary tree, we can see that there is at most one node at level 0, namely the root and each node has at most 2 children, so there are at most 2 nodes of level 1. Continuing this reasoning inductively, we can see that there are at most 2^t nodes of level t.

Thus for a fixed t we can bound the inner sum by

$$\sum_{N\,:\,\text{node of level } t} f(t) \leq 2^t f(t) = 2^{t+f(t)} \frac{f(t)}{2^{f(t)}} = 2^{\text{height}(T)} \frac{f(t)}{2^{f(t)}}$$

Then our original sum becomes

$$c \sum_{t=0}^{\text{height}(T)} \sum_{N\,:\,\text{node of level } t} h(t) \leq c \sum_{t=0}^{\text{height}(T)} 2^{\text{height}(T)} \frac{f(t)}{2^{f(t)}} = c \cdot 2^{\text{height}(T)} \sum_{t=0}^{\text{height}(T)} \frac{f(t)}{2^{f(t)}}$$

Note that our inner sum is just the hint but with reverse indexing, thus we can bound the sum by 2. This gets us

$$c \cdot 2^{\text{height}(T)} \cdot 2 = 2c \cdot 2^{\text{height}(T)}$$

As $\text{height}(T) \leq \lfloor \log_2(n) \rfloor$, we can see that $2^{\text{height}(T)} \leq 2^{\lfloor \log_2(n) \rfloor} \leq 2^{\log_2(n)+1} = 2n$. Thus the total runtime can be bounded by $2c \cdot 2n \leq O(n)$.

Data structures.

Exercise 5.5 *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.
- $\text{pop}(S)$: Remove (and return) the top element of the stack S.

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.

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

(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.

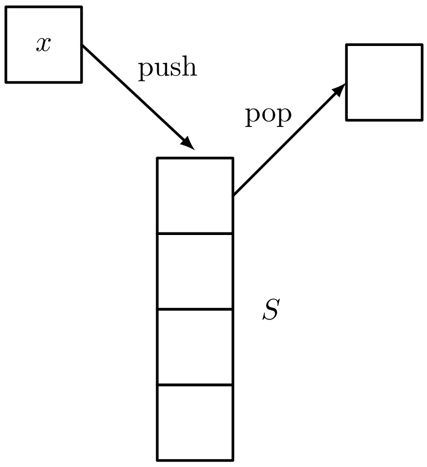


Figure 1: Abstract data type stack

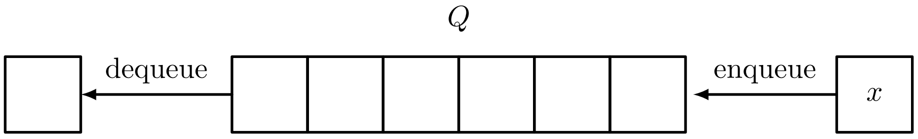


Figure 2: Abstract data type queue

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 $\mathrm{push}(x, S)$ 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 $\mathrm{pop}(S)$ 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 $\mathrm{push}(x, S)$ and $\mathrm{pop}(S)$, 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 $\mathrm{enqueue}(x, Q)$ adds the new element at the end of the list using the pointer to the end of the list. The operation $\mathrm{dequeue}(Q)$ 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 $\mathrm{enqueue}(x, Q)$) and of the second element (for $\mathrm{dequeue}(Q)$) 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 $\mathrm{enqueue}(x, Q)$ and $\mathrm{dequeue}(Q)$ have run time $O(1)$.