CS 225 – Data Structures

Midterm Exam – Model Answers

Conceptual Questions

Exercise 1. Design an algorithm to extract the k'th element of a FIFO queue and determine its worst case time complexity.

Note that while the k'th element is deleted from the queue, all other elements are to be kept and their order is to be preserved.

Solution. We can simply scan the queue from front to back. If the length of the queue is n we iterate n steps. Each time we execute *popfront* and assign the result gto a temporary variable *temp*. In the k'th step, we copy the value of *temp* to the output variable *result*. In all other steps we execute *pushback*(*temp*).

As all elements of the queue except the k'th one are removed from the front of the queue and re-inserted at the end, the resulting queue contains the same elements in the same order except for the k'th one, which has been eliminated from the queue and returned as output. So the algorithm extracts the k'th element from the queue as required.

As we iterate the *popfront* and *pushback* (or assignment to *result*) exactly n times, the worst case time complexity is in $O(n)$.

Exercise 2. The time complexity of insertion sort is in $O(n^2)$ in the worst case. Which data structure instead of lists can be used to turn insertion sort into an optimal sorting algorithm? Justify your answer.

Solution. We can use an AVL tree, which initially is empty. We scan the given list of length n from front to back and insert the elements one-by-one into the AVL-tree. The insertion takes time in $O(\log n)$, so the time complexity for building the AVL tree is in $O(n \log n)$.

Then we scan the AVL tree in a depth-first way, i.e. for every node first explore the left successor tree, then append the value of the node to an output list, which initially is empty, and explore the right successor tree. For an empty tree there is nothing to do. By the definition of an AVL tree the output list will contain all elements of the tree (and hence of the original list) in a sorted way.

As the scanning of the AVL tree examines each node exactly once, it requires time in $O(n)$. Altogether the algorithm requires time in $O(n \log n)$.

Exercise 3.

(i) Apply the divide-and-conquer algorithm for the rotation of the list

$$[13, 21, 7, 8, 5, 1, 3, 2, 6, 11, 10, 9, 4, 17, 18]$$

by four positions. Show explicitly the progression of the algorithm and the resulting output.

(ii) Apply the divide-and-conquer algorithm for selection to determine the 7'th smallest element of the list

$$[13, 21, 7, 8, 5, 1, 3, 2, 6, 11, 10, 9, 4, 17, 18] \ .$$

Show explicitly the progression of the algorithm and the resulting output.

SOLUTION.

(i) The algorithm proceeds as follows:

list	k	i	j	**Comment**
13, 21, 7, 8, 5, 1, 3, 2, 6, 11, 10, 9, 4, 17, 18	1	4	11	initialisation
9, 4, 17, 18, 5, 1, 3, 2, 6, 11, 10, 13, 21, 7, 8	1	4	7	swap sublists of length $i = 4$ at positions $k = 1$ and $k+j = 12$, update j to $j - i$
2, 6, 11, 10, 5, 1, 3, 9, 4, 17, 18, 13, 21, 7, 8	1	4	3	swap sublists of length $i = 4$ at positions $k = 1$ and $k + j = 8$, update j to $j - i$
5, 1, 3, 10, 2, 6, 11, 9, 4, 17, 18, 13, 21, 7, 8	4	1	3	swap sublists of length $j = 3$ at positions $k = 1$ and $k + i = 5$, update k to $k + j$ and i to $i - j$
5, 1, 3, 11, 2, 6, 10, 9, 4, 17, 18, 13, 21, 7, 8	4	1	2	swap sublists of length $i = 1$ at positions $k = 4$ and $k + j = 7$, update j to $j - i$
5, 1, 3, 6, 2, 11, 10, 9, 4, 17, 18, 13, 21, 7, 8	4	1	1	swap sublists of length $i = 1$ at positions $k = 4$ and $k + j = 6$, update j to $j - i$
5, 1, 3, 2, 6, 11, 10, 9, 4, 17, 18, 13, 21, 7, 8	4	1	1	swap sublists of length $i = 1$ at positions $k = 4$ and $k + i = 5$, terminate

The output list is $[5, 1, 3, 2, 6, 11, 10, 9, 4, 17, 18, 13, 21, 7, 8]$.

(ii) Let $k = 7$. Choose $p = 13$ as the pivot element. Then we get $U = [7, 8, 5, 1, 3, 2, 6, 11, 10, 9, 4]$, the sublist of elements $< p$ with $u = |U| = 11$ and $V = [21, 17, 18]$, the sublist of elements $> p$ with $v = 15 - |V| = 12$.

As $k < u$, we continue to select the 7'th smallest element in U. Choose $p = 7$ as pivot element. Then we get $U = [5, 1, 3, 2, 6, 4]$, $u = 6$, $V = [8, 11, 10, 9]$, and $v = 11 - |V| = 7$.

As $k = v$, the result in $p = 7$.

EXERCISE 4. Discuss what happens with Fibonacci heaps, if the marking of nodes is dropped.

(i) Describe which operations need to be changed. Which alternatives do you have?

(ii) Describe how the changes affect the amortised complexity of the affected operations. Is it possible to obtain the same complexity bounds?

(iii) Sketch an example of a sequence of operations on a Fibonacci heap, which would have worse complexity in case there are no marked nodes.

SOLUTION.

(i) Marked nodes in Fibonacci heaps are used to indicate, whether cascading cuts are triggered or not. If there are no marked nodes, we have two alternatives:

(a) Do not perform any cascading cuts, i.e. in the `decrease` operation simply cut out the identified subtree and insert its root into the root list.

(b) Always perform cascading cuts until a root is reached.

(ii) For `decrease` the complexity is in $O(c)$, where c is the number of cascading cuts. For alternative (a) above this becomes $O(1)$, which is equal to the amortised complexity of `decrease` with marked nodes. For alternative (b) in case of cascading cuts the change in the potential is bounded by $4 - c$, so the amortised complexity remains in $O(1)$.

However, in both cases it is not possible to maintain a logarithmic bound on the maximum degree of nodes in the Fibonacci heap. For alternative (a) the lack of cut operations does nor decrease degrees. For alternative (b) the abundance of cut operations leads to large root lists. The amortised complexity of `delete_min` including the `consolidate` remains in $O(D(n))$, but as $D(n)$ is not bounded by $\log_\phi n$, this may be as bad as $O(n)$. That is, the amortised complexity bound for `delete_min` becomes worse.

(iii) As the analysis in (ii) shows, we obtain worse complexity, if `delete_min` is executed when $D(n)$ is large (alternative (a)) or the root list is large (alternative (b)). In both cases sufficiently long sequences of `decrease` operations lead to such a state.

EXERCISE 5. Let p be a prime number and $m \leq p$. Consider a class $H = \{h_{a,b} \mid 0 \leq a, b \leq p-1\}$ of hash functions defined as

$$h_{a,b}(x) = ((ax + b) \mod p) \mod m \quad \text{for } x, y \in \mathbb{F}_p.$$

Show that this class H is c-universal for $c = (\lceil p/m \rceil / (p/m))^2$.

SOLUTION. For $x \neq y$ and arbitrary $a, b \leq p-1$ we have $h_{a,b}(x) = h_{a,b}(y)$ iff $(a(x-y) \bmod p) = 0 \mod m$. As we assume $x \not\equiv y \bmod p$, the inverse $(x-y)^{-1} \in \mathbb{F}_p$ exists. Hence $h_{a,b}(x) = h_{a,b}(y)$ holds iff $a \in \{0, \ldots, \lceil p/m \rceil - 1\}$. Therefore, we have $\lceil p/m \rceil \cdot m$ possible values for b and $\lceil p/m \rceil$ possible values for a with this property, i.e.

$$|\{h \in H \mid h(x) = h(y)\}| \leq \lceil p/m \rceil^2 \cdot m = \frac{c}{m} \cdot p^2 = \frac{c}{m} \cdot |H| \,,$$

as there are p^2 different choices for the pair (a, b). By definition this shows the claimed c-universality.

EXERCISE 6. Show that an arbitrary AVL tree with n nodes can be transformed into any other AVL search tree with the same elements by a sequence of $r(n)$ rotations with $r(n) \in O(n)$.

SOLUTION. Ignore the balances and simply consider an AVL tree as a BST. Then use the following way to represent a BST: $v_1, \ldots, v_k$ are vertices such that

- v_1 is the root, and v_{i+1} is the right successor of v_i for $1 \leq i \leq k-1$;
- t_i denotes the left successor tree of v_i;
- t_i contains n_i elements (n_i may be 0);
- the right successor tree of v_k is a right-going chain (it may be empty).

Assume that k is maximal with these properties. We show that we need at most $\sum_{i=1}^{k} n_i \leq n-1$ right rotations to transform this BST into a right-going chain.

Obviously, if we can show this for $k = 1$, then it also holds for arbitrary k: We can use $\sum_{i=1}^{k}$ right rotations to turn the subtree rooted at v_2 into a right-going chain, then use n_1 right rotations to turn the whole BST into a right-going chain. So assume now $k = 1$.

Now let v_0 denote the root, and v_1 be its left successor. Let the left and right successor trees of v_1 be t_1 and t_2 with n_1 and n_2 nodes, respectively. Assume that the right successor tree of v_0 is already a right-going chain. Then apply a single right rotation on v_0 and v_1. Using induction ($n_2 < n_1 + n_2 + 1$) we need at most n_2 right rotations to transform the subtree rooted at v_0 into a right-going chain. Again by induction ($n_1 < n_1 + n_2 + 1$) we then need at most n_1 right rotations to transform the the whole BST into a right-going chain.

It remains to have an induction base for $n_i \leq 1$. The four cases $n_1 = n_2 = 0$, $n_1 = 1/n_2 = 0$, $n_1 = 0/n_2 = 1$ and $n_1 = n_2 = 1$ require $1, 2, 2$ and 3 rotations, respectively, i.e. always $\leq n$ rotations.

Finally, if we have two arbitrary AVL trees B_1 and B_2 with the same n elements, then there are sequences of r_1 and r_2 right rotations transforming B_1 and B_2 into right-going chains. These chains must be the same BST B. Let these sequences be

$$B_1 \overset{\rho_1}{\to} B_{11} \overset{\rho_2}{\to} B_{12} \to \cdots \overset{\rho_{r_1}}{\to} B \overset{\rho'_{r_2}}{\leftarrow} \cdots \leftarrow B_{22} \overset{\rho'_2}{\leftarrow} B_{21} \overset{\rho'_1}{\leftarrow} B_2 \,.$$

Then for each ρ'_i take the inverse left rotation λ_i, which defines a sequence of rotations

$$B_1 \overset{\rho_1}{\to} B_{11} \overset{\rho_2}{\to} B_{12} \to \cdots \overset{\rho_{r_1}}{\to} B \overset{\lambda_{r_2}}{\to} \cdots \to B_{22} \overset{\lambda_2}{\to} B_{21} \overset{\lambda_1}{\to} B_2$$

transforming B_1 into B_2. The number of rotations is $r_1 + r_2 \leq 2(n-1) \in O(n)$.

Programming Questions

EXERCISE 7. Implement a function to *reverse* the order of elements in a FIFO queue. You may use unbounded arrays or linked lists for the implementation of queues.

Programming instructions: You may reuse the definition of the class FIFO and extend it (among others) by a function *reverse* or you may exploit the class DLIST for a different implementation of FIFO queues including a member function *reverse*.

Testing instructions: Then use the following queues for testing:

```
[0,-4,1,2,-8,3,4,5,-1, 6,9,-7,-6,-5,7,-3,8,-2]
[30,42,55,79,-5,264,187,56]
```

Furthermore, test your implementation after a sequence of *popfront* and *pushback* operations.

SOLUTION. See the C++ header and program files in `Ex7_FIFO.zip` or `Ex7_Dfifo.zip`.

Alternative 1. We use a circular array with an additional variable *revdir* indicating a direction. If *revdir* is **false**, the operations are executed in the same way as for FIFO queues. If *revdir* is **true**, *pushback* is replaced by a new operation *pushfront*, and *popfront* is replaced by a new operation *popback*. Furthermore, *back* and *front* swap their roles. Then *reverse* is simply implemented by a negation of *revdir*.

Alternative 2. We base the implementation of FIFO queues on doubly-linked lists, so *pushback* is implemented by *append*, *front* by *getitem*(1), and *popfront* combines *getitem*(1) and *remove*(1). Then *reverse* requires to swap the next- and previous pointers for all nodes including the dummy node.

EXERCISE 8. Implement a data structure to arrange two stacks in a single array of size n such that no stack overflow occurs, as long as the total number of elements in both stacks together does not exceed n. Your implementation must guarantee that *push* and *pop* can be realised in $O(1)$ time.

Programming instructions: Use the definition of the class ALIST and extend it by a functions pop_i, top_i, $push_i$ and is_empty_i $(i = 1, 2)$:

```
template<class T> class AList
{
public:
    ....
    T pop1;
    T top1;
    void push1(T item);
    bool isempty1(void);
    T pop2;
    T top2;
    void push2(T item);
    bool isempty2(void);
....
}
```

Testing instructions: Then use the following stacks for testing:

```
[0,-4,1,2,-8,3,4,5,-1, 6,9,-7,-6,-5,7,-3,8,-2]
[30,42,55,79,-5,264,187,56]
```

SOLUTION. See the C++ header and program files in `Ex8_Stacks.zip`. We use a representing array of length n. The first k entries are used for the elements of the first stack from bottom to top, and the last ℓ enties are used for the elements of the second stack from top to bottom. So allocation and deallocation become necessary, when $k+\ell = n$ or $k+\ell \leq n/4$ hold, respectively. The operations push, pop and top only need to use the correct indices.

EXERCISE 9. Implement a function *median* on AVL trees to retrieve the median of the stored elements. You may build your implementation on the class AVL with or without modification of the basic data structure, and you should exploit the divide-and-conquer algorithm to retrieve the median.

Programming instructions: Use the class AVL, modify it if necessary, and extend it by defining a member function *median* as follows:

```
template<class T> class AVL
{
public:
    ....
    T median(void);
....
}
```

Testing instructions: Then test your implementation on an AVL tree built by inserting the following integers:

```
0,30,42,55,-4,1,2,-8,79,3,4,5,-1,6,9,187,-7,-6,-5,7,-3,264,8,-2
```

SOLUTION. See the C++ header and program files in `Ex9_AVL.zip`. We use depth-first search through the AVL tree to collect a list of all elements of the tree. According to the definition of AVL trees the list must be ordered, so it suffices to select the k'th element for $k = \lceil n/2 \rceil$.