CS 225 – Data Structures

Midterm Exam 1 – Selected Model Answers (Conceptual Questions)

Exercise 1. Explain how 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. Show that *push* and *pop* can be realised in $O(1)$ time.

Solution. Let $A[0,\ldots,n-1]$ be an array of size n. When placing two stacks st_1 and st_2 into A, we use $A[0,\ldots,i-1]$ for st_1 and $A[j,\ldots,n-1]$ for st_2. In doing so, we make sure that $A[i-1]$ is the top element of st_1 and $A[j]$ is the top element of st_2.

Then $n_1 = i$ is the number of elements in st_1 and $n_2 = n - j$ is the number of elements in st_2. So we will not have a stack overflow as long as $n_1 + n_2 \leq n$.

We can realise the operation pop_1 on st_1 simply by decreasing n_1 and returning $A[n_1-1]$. The operation $push_1$ on st_1 with input *newitem* requires to set $A[n_1] = newitem$ and to increment n_1. Clearly, both operations are in $O(1)$.

Analogously, we can realise the operation pop_2 on st_2 simply by decreasing n_2 and returning $A[n_1 - n_2]$. The operation $push_2$ on st_2 with input *newitem* requires to set $A[n - n_2 - 1] = newitem$ and to increment n_2. Clearly, both operations are in $O(1)$.

The operations $push_1$ or $push_2$ can only be executed, if $n_1+n_2 \leq n$ is satisfied. If not, we first have to use an *allocate* operation, which creates a new array B of size $2n$, copies st_1, i.e. we will have $B[i] = A[i]$ for $0 \leq i \leq n_1 - 1$, and also copies st_2, i.e. we will have $B[j+n] = A[j]$ for $n - n_2 \leq j \leq n - 1$.

In case $n_1+n_2 \leq \lfloor n/4 \rfloor$ and $n > min$ hold (where min is a minimum size for the representing array), we perform a *deallocate* operation. This creates a new array B of size $n/2$, copies st_1, i.e. we will have $B[i] = A[i]$ for $0 \leq i \leq n_1 - 1$, and also copies st_2, i.e. we will have $B[j] = A[j+n]$ for $n - n_2 \leq j \leq n - 1$. □

Exercise 2. Assume that n elements are stored in a hash table with m entries and a random hash function h is used. What is the expected number of collisions, i.e.

$$|\{\{x,y\} \mid x \neq y \wedge h(x) = h(y)\}| \; ?$$

Solution. We have $h(x), h(y) \in \{0,\ldots,m-1\}$. As h is assumed to be a random hash function, the probability for equation is $P(h(x) = h(y)) = 1/m$, as for fixed $h(x)$ there are m independent choices for $h(y)$ and only one of them yields equality.

Furthermore, we have $|\{\{x,y\} \mid x \neq y\}| = \frac{n(n-1)}{2}$, which implies that

$$|\{\{x,y\} \mid x \neq y \wedge h(x) = h(y)\}| \;=\; \frac{1}{m} \cdot \frac{n(n-1)}{2} \; .$$ □

EXERCISE 3. Use an array to represent doubly linked lists, i.e. every entry in the array is a triple containing a stored item, the index of the next list element and the index of the previous list element. Use a hash function into this array with linear probing. Describe the list operations on this data structure. Does this data structure help to improve the performance of operations on doubly linked lists?

SOLUTION. If we represent the list $[a_0, \ldots, a_k]$ with $a_i \in T$ in an array $A[0, \ldots, n-1]$ of size n in the required way, we use positions $i_0, \ldots, i_k$ such that for $0 \leq j \leq k$ we have $A[i_j] = (a_j, i_{j+1}, i_{j-1})$, where the additional setting $i_{-1} = i_{k+1} = None$ is used. We can assume that i_0 is always known. In addition, let $h : T \to \{0, \ldots, n-1\}$ be a hash function. List operations are the following:

get(p) to retrieve the p'th element a_p of the list. Then we need to follow the next links, i.e. we follow the sequence $i_0, i_1, \ldots, i_p$, where i_{j+1} is the second component of $A[i_j]$, and finally return the first component of $A[i_p]$.

set(p, x) to update the p'th element of the list to $a_p = x$. Same as for set(p) we follow the sequence $i_0, i_1, \ldots, i_p$ and finally assign x to the first component of $A[i_p]$.

append(x) to add a new last element to the list. In this case determine $i = h(x)$ and let i_{k+1} be the smallest index $\geq i$ such that $A[i_{k+1}]$ is undefined—continue with $0, 1, 2, \ldots$ if there is no such value $\leq n-1$. Furthermore, determine the current last element $A[i_k] = (y, None, i_{k-1})$ by traversing through the list in the same way as for get and set. Then update $A[i_k] = (y, i_{k+1}, i_{k-1})$ and insert $A[i_{k+1}] = (x, None, i_k)$.

insert(p, x) to insert a new p'th element to the list. Same as for append determine $i = h(x)$ and let i'_p be the smallest index $\geq i$ such that $A[i'_p]$ is undefined. Furthermore, proceed as for the get operation to determine $A[i_{p-1}] = (y_1, i_p, i_{p-2})$ and $A[i_p] = (y_2, i_{p+1}, i_{p-1})$. Then set $A[i'_p] = (x, i_p, i_{p-1})$ and update $A[i_{p-1}] = (y_1, i'_p, i_{p-2})$ and $A[i_p] = (y_2, i_{p+1}, i'_p)$.

delete(p) to delete the p'th element from the list. Analogous to insert determine $A[i_{p-1}] = (y_1, i_p, i_{p-2})$, $A[i_p] = (x, i_{p+1}, i_{p-1})$ and $A[i_{p+1}] = (y_2, i_{p+2}, i_p)$. Then update $A[i_{p-1}] = (y_1, i_{p+1}, i_{p-2})$ and $A[i_{p+1}] = (y_2, i_{p+2}, i_{p-1})$, and make $A[i_p]$ undefined.

The availability of the hash function h has no impact on the complexity of these operations. It will only allow us to implement fast search for an element x in the list, which would be a new list operation. □

EXERCISE 4.

(i) Using a representation by arrays for lists with elements in a totally ordered set T implement an operation *select*, which for a given argument $x \in T$ returns the sublist of all elements that are $\leq x$.

(ii) Extend (i), determine the *median* m of the list, and implement an operation *select_low* returning the sublist of all elements $\leq m$.

EXERCISE 5.

(i) Using doubly linked lists with elements in a set T implement an operation *reverse*, which reverses the order of the elements in a given list.

(ii) Show that your operation works in-place and requires time in $O(n)$, where n is the length of the given list.

SOLUTION. We only look a the conceptual part (ii).

A node in a doubly-linked list contains a value $v \in T$, a forward pointer to the next node in the list, and a backward pointer to the previous element in the list, i.e. each node can be represented as a triple (v, f, p). The forward pointer of the last list element as well as the backward pointer of the first node point to the address d of a dummy node. The dummy node itself does contains the value *undef*, the forward pointer points to the first list element, and the backward pointer points to the last list element. In case of an empty list these pointers are both d.

We obtain a reversed list, if for each node (v, f, p) including the dummy node we simply swap f with p. A swap operation can be done in constant time. If there are n elements in the list, we need to execute $n + 1$ such swap operations, so the time complexity is in $O(n)$.

For these operations we need only three additional variables, one to store temporarily one of the values in the swap, the second one to store temporarily the forward pointer to be available for navigating to the next list element, and the third one to store d in order to know when all nodes of the list have been dealt with. This satisfies the requirement to operate in-place.

EXERCISE 6.

(i) Consider queues with elements in $T \times \mathbb{N}$, where the number n in a pair (t, n) is considered as the *priority* of t, and implement an operation *pop_vip* that removes the first pair (t_0, n_0) in the queue, for which n_0 is minimal, and returns the element $t_0 \in T$.

(ii) Determine the amortised complexity of your operation.

SOLUTION. We only look a the conceptual part (ii).

The operations *front*, *pop_front*, *back* and *push_back* remain unchanged. Assume that these operations (up to some multiplicative constant c) change the potential by 0, 1, 0 and 2, which amounts to an amortised complexity in $O(1)$ in all cases, taking into account that *allocate* and *deallocate* change the potential by $-n$, when n is the number of elements in the queue. This has been shown in the lectures.

The new operation *pop_vip* has to scan the whole queue to find the minimum value n_0. Each time it may be necessary to update the candidate position, indicating where the so far smallest priority value was found. Once the minimum n_0 is found, all following list entries must be shifted. In the worst case we need $3n$ operations, where n is the number of elements in the queue, so *pop_vip* requires time in $O(n)$.

If we consider an additional add-on from the *push_back* operations to the potential, this must account for up to n possible *pop_vip* operations each requiring a multiple of n. However, this is

not possible. A additional constant contribution of *push_back* to the potential does not change the complexity of *pop_vip*. Hence, also the amortised time complexity of *pop_vip* remains in $O(n)$.

EXERCISE 7. 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

$$h_{a,b}(x) = ((ax + b) \mod p) \mod m \,.$$

Note. This class H is actually $(\lceil p/m \rceil/(p/m)^2)$-universal.

(i) Implement a version of hashing with linear probing, where each time rehashing becomes necessary the new hash function is randomly selected from the class H.

(ii) Discuss advantages and disadvantages of this approach.

SOLUTION. We only look a the conceptual part (ii).

As the class of hash functions in $(\lceil p/m \rceil/(p/m)^2)$-universal, the expected execution time of *find* and *delete* in a hash table with chaining is in $O(1 + (\lceil p/m \rceil/(p/m)^2)n/m)$. The same holds for linear probing with the difference that we search in the hash table rather than in the lists stored in the entries of the table.

As $\lceil p/m \rceil/(p/m)^2 \approx m/p$, we have time complexity in $O(1 + n/p)$. If p is a large prime number, this further reduces to $O(n/p)$. Thus, if p is large, the time complexity tends to be lower than using a fixed hash function.

On the other hand, for a very large p the remainder $\mod p$ may degerate to the identity, which implies that the advantage disappear. Furthermore, in a hash table with linear probing we may have placeholders or extend the search by looking also at entries with different hash values, so the expected length of the search is larger than for hashing with chaining.

In summary, the choice of p is decisive for the approach having advantages, and the presence of placeholders is a disadvantage.