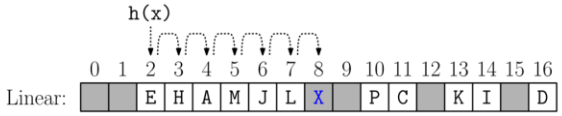
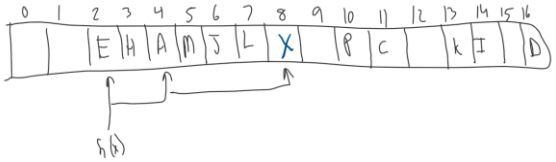
HW 4

1) O(1) amortized search, O(1) amortized insertion

2)





3)

4) TLDR; We need to define a load factor that will determine when we need to rehash all elements (either we go below load factor or we go above it) and create a new hash table. Amortized analysis works. See below if interested. Components of a full answer. (Use load factor, load factor based on elements, upper and lower bound and proof of solution for O(1)

Controlling the Load Factor and Rehashing: Recall that the load factor of a hashing scheme is $\lambda = n/m$, and the expected running time of hashing operations using separate chaining is $O(1 + \lambda)$. We will see below that other popular collision-resolution methods have running times that grow as $O(\lambda/(1 - \lambda))$. Clearly, we would like λ to be small and in fact strictly smaller than 1. Making λ too small is wasteful, however, since it means that our table size is significantly larger than the number of keys. This suggests that we define two constants $0 < \lambda_{\min} < \lambda_{\max} < 1$, and maintain the invariant that $\lambda_{\min} \leq \lambda \leq \lambda_{\max}$. This is equivalent to saying that $n \leq \lambda_{\max} m$ (that is, the table is never too close to being full) and $m \leq n/\lambda_{\min}$ (that is, the table size is not significantly larger than the number of entries). Define the *ideal load factor* to be the mean of these two, $\lambda_0 = (\lambda_{\min} + \lambda_{\max})/2$.

Now, as we insert new entries, if the load factor ever exceeds $\lambda_{\max}$ (that is, $n > \lambda_{\max} m$), we replace the hash table with a larger one, devise a new hash function (suited to the larger size), and then insert the elements from the old table into the new one, using the new hash function. This is called *rehashing* (see Fig. 2). More formally:

- Allocate a new hash table of size $m' = \lceil n/\lambda_0 \rceil$
- Generate a new hash function h' based on the new table size
- For each entry (x, v) in the old hash table, insert it into the new table using h'
- Remove the old table

Observe that after rehashing the new load factor is roughly $n/m' \approx \lambda_0$, thus we have restored the table to the ideal load factor. (The ceiling is a bit of an algebraic inconvenience. Throughout, we will assume that n is sufficiently large that floors and ceilings are not significant.)

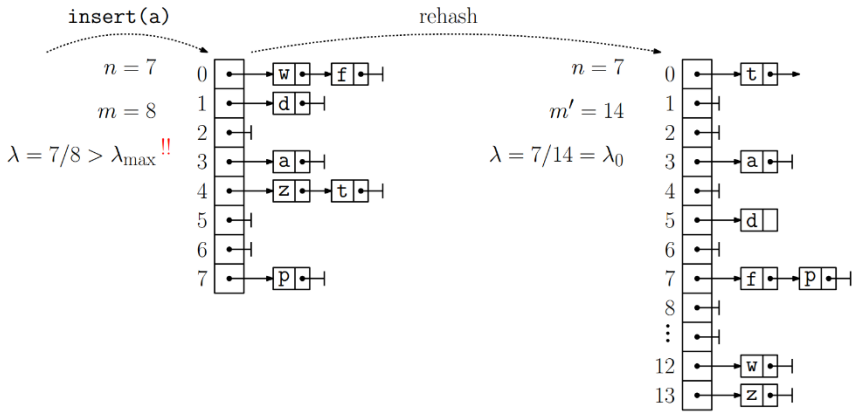


Fig. 2: Controlling the load factor by rehashing, where $\lambda_{\min} = 0.25$, $\lambda_{\max} = 0.75$, and $\lambda_0 = 0.5$.

Symmetrically, as we delete entries, if the load factor ever falls below $\lambda_{\min}$ (that is, $n < \lambda_{\min} m$), we replace the hash table with a smaller one of size $\lceil n/\lambda_0 \rceil$, generate a new hash function for this table, and we rehash entries into this new table. Note that in both cases (expanding and contracting) the hash table changes by a constant fraction of its current size. This is significant in the analysis.

Amortized Analysis of Rehashing: Observe that whenever we rehash, the running time is proportional to the number of keys n. If n is large, rehashing clearly takes a lot of time. But observe that once we have rehashed, we will need to do a significant number of insertions or deletions before we need to rehash again.

To make this concrete, let's consider a specific example. Suppose that $\lambda_{\min} = 1/4$ and $\lambda_{\max} = 3/4$, and hence $\lambda_0 = 1/2$. Also suppose that the current table size is $m = 1000$. Suppose the most recent insertion caused the load factor to exceed our upper bound, that is $n > \lambda_{\max} m = 750$. We allocate a new table of size $m' = n/\lambda_0 = 2n = 1500$, and rehash all the old elements into this new table. In order to overflow this new table, we will need for n to increase to some higher value n' such that $n'/m' > \lambda_{\max}$, that is $n' > (3/4)1500 = 1125$. In order to grow from the current 750 keys to 1125 keys, we needed to have at least 375 more insertions (and perhaps many more operations if finds and deletions were included as well). This means that we can *amortize* the (expensive) cost of rehashing 750 keys against the 375 (cheap) insertions.

Hopefully, this idea will sound familiar to you. In an earlier lecture, we discussed the idea of doubling an array to store a stack. We showed there that by doubling the storage each time the stack overflowed, the amortized cost of each operation is just $O(1)$. There was no magic to doubling. Increasing the storage by any constant factor works, and the same analysis applies here as well. Each time we rehash, we are either increasing or decreasing the hash-table size by a constant factor. Assuming that the hash operations themselves take constant time, we can "charge" the expensive rehashing time to the inexpensive insertions or deletions that led up to the present state of affairs.

Recall that the *amortized cost* of a series of operations is the total cost divided by the number of operations.

Theorem: Assuming that individual hashing operations take $O(1)$ time each, if we start with an empty hash table, the amortized complexity of hashing using the above rehashing method with minimum and maximum load factors of $\lambda_{\min}$ and $\lambda_{\max}$, respectively, is at most $1 + 2\lambda_{\max}/(\lambda_{\max} - \lambda_{\min})$.

Proof: Our proof is based on the same *token-based argument* that we used in the earlier lecture. Let us assume that each standard hashing operation takes exactly 1 unit of time, and rehashing takes time n, where n is the number of entries currently in the table. Whenever we perform a hashing operation, we assess 1 unit to the actual operation, and save $2\lambda_{\max}/(\lambda_{\max} - \lambda_{\min})$ *work tokens* to pay for future rehashings.

There are two ways to trigger rehashing: expansion due to insertion, and contraction due to deletion. Let us consider insertion first. Suppose that our most recent insertion has triggered rehashing. This implies that the current table contains roughly $n \approx \lambda_{\max} m$ entries. (Again, to avoid worrying about floors and ceilings, let's assume that n is quite large.) The last time the table was rehashed, the table contained $n' = \lambda_0 m$ entries immediately after the rehashing finished. This implies that we inserted at least $n - n' = (\lambda_{\max} - \lambda_0)m$ entries. Therefore, the number of work tokens we have accumulated since then is at least

$$\begin{aligned}
(\lambda_{\max} - \lambda_0)m\frac{2\lambda_{\max}}{\lambda_{\max} - \lambda_{\min}} &= \left(\lambda_{\max} - \frac{\lambda_{\max} + \lambda_{\min}}{2}\right) m\frac{2\lambda_{\max}}{\lambda_{\max} - \lambda_{\min}} \\
&= \left(\frac{\lambda_{\max} - \lambda_{\min}}{2}\right) m\frac{2\lambda_{\max}}{\lambda_{\max} - \lambda_{\min}} \\
&= \lambda_{\max} m \approx n,
\end{aligned}$$

which implies that we have accumulated enough work tokens to pay the cost of n to rehash.

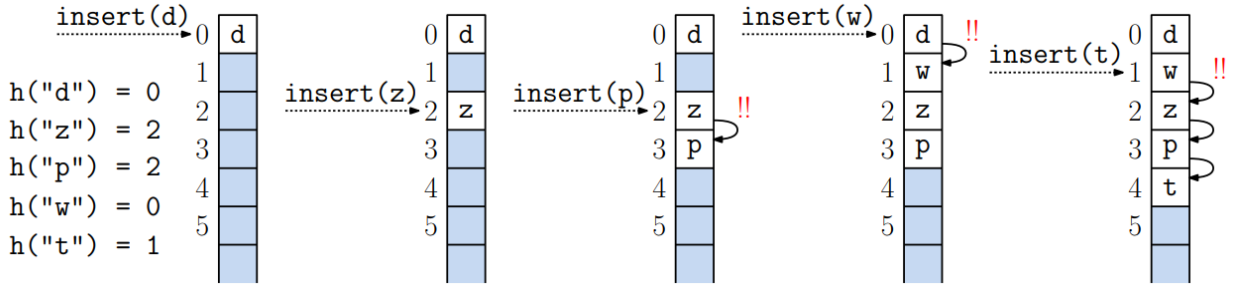
Next, suppose that our most recent deletion has triggered rehashing. This implies that the current table contains roughly $n \approx \lambda_{\min} m$ entries. (Again, to avoid worrying about floors and ceilings, let's assume that n is quite large.) The last time the table was rehashed, the table contained $n' = \lambda_0 m$ entries immediately after the rehashing finished. This implies that we deleted at least $n' - n = (\lambda_0 - \lambda_{\min})m$ entries. Therefore, the number of work tokens we have accumulated since then is at least

$$\begin{aligned}
(\lambda_0 - \lambda_{\min})m\frac{2\lambda_{\max}}{\lambda_{\max} - \lambda_{\min}} &= \left(\frac{\lambda_{\max} + \lambda_{\min}}{2} - \lambda_{\min}\right) m\frac{2\lambda_{\max}}{\lambda_{\max} - \lambda_{\min}} \\
&= \left(\frac{\lambda_{\max} - \lambda_{\min}}{2}\right) m\frac{2\lambda_{\max}}{\lambda_{\max} - \lambda_{\min}} \\
&= \lambda_{\max} m \geq \lambda_{\min} m \approx n,
\end{aligned}$$

again implying that we have accumulated enough work tokens to pay the cost of n to rehash.

To make this a bit more concrete, suppose that we set $\lambda_{\min} = 1/4$ and $\lambda_{\max} = 3/4$, so that $\lambda_0 = 1/2$ (see Fig. 2). Then the amortized cost of each hashing operation is at most $1 + 2\lambda_{\max}/(\lambda_{\max} - \lambda_{\min}) = 1 + 2(3/4)/(1/2) = 4$. Thus, we pay just additional factor of four due to rehashing. Of course, this is a worst case bound. When the number of insertions and deletions is relatively well balanced, we do not need rehash very often, and the amortized cost is even smaller.

5)



6)

0	
1	
2	2
3	
4	15
5	
6	6
7	7
8	28
9	
10	
11	
12	19

For part b we were looking to see where 19 lands after reinsertion. All other elements land in the same spot. If 19 lands in spot 8 that is full credit. If it lands in spot 12 it is wrong. If it lands elsewhere that is dependent on part a and partial credit was given as appropriate.