Contents

1 Amortized Analysis

There are a few ways we can reason about the complexity of a sequence of n operations:

1. We can bound the cost of the operation by $cost_{op} = O(f(n))$ and then say than n operations will take cost $n \cdot cost_{op} = n \cdot O(f(n))$. We used a similar analysis for BUILD-MIN-HEAP which consisted of n HEAPIFY operations, where each HEAPIFY operation took $O(\log n)$ time. Thus, the cost of BUILD-MIN-HEAP was $O(n \log n)$.

2. We can bound the actual cost of the *entire* sequence of n operations. This is *different* than bounding the cost of one operation and multiplying it by n to get the full cost. Now we are looking at the individual cost of each operation, adding it up to the total cost. We do this because some of the n operations might be cheap, while others might be more expensive. It would be naive to consider all of them as having the same cost. In our initial analysis of BUILD-MIN-HEAP, that naivity hurt us. More careful analysis showed the running time was $\Theta(n)$.

1.1 Amortized analysis of INSERT in a hash-table

Remember that the SEARCH time in a chaining hash-table of size m filled with n items was $\Theta(1 + \alpha) = \Theta(1 + n/m)$. As a result, if we INSERT too many elements so that $n \gg m$, our load factor α will become bigger and increase the SEARCH time.

Thus, it's important to adjust our table size so that $m = O(n)$ which ensures α is small and SEARCH is fast. A good way of increasing the size of our table is using *doubling*.

When $n = m$ and an INSERT operation is trying to add a new element to our hash table, we do the following:

1. Allocate a new hash table double in size.
2. Pick a new hash function h that will work with the new size.
3. Rehash everything into the new table, including the newly inserted item.
4. Delete the old hash table.

We start with a hash-table of size one so that by the 2nd INSERT we'll have to double our size, according to our above algorithm for table doubling. Let's now try and analyze a sequence of n INSERT operations. For simplicity (avoiding $\log$'s), we assume $n = 2^p$, for some p.

We can define the cost of the ith INSERT as:

$$c_i = \begin{cases} i, & \text{if } i-1 \text{ is a power of } 2 \\ 1, & \text{otherwise} \end{cases}$$

To see this, think about what happens as items are inserted into the hash table causing it to be resized every now and then. Our table will double in size from 1 to 2, to 4, to 8, to 16, and so on. We double the size after inserting the 2nd key, the 3rd key, the 5th key and the 9th key, because those are all instances when our table is full and we're trying to add another key to it. Note that all these numbers are off by one from powers of two: $1, 2, 4$ and 8. You should now see why the cost of the 9th INSERT is 8 (rounded down a little): because we had to double the size and rehash everything.

First, consider a small example, when $n = 12$:

Insert #	1	2	3	4	5	6	7	8	9	10	11	12
$size_i$	1	2	4	4	8	8	8	8	16	16	16	16
c_i	1	2	3	1	5	1	1	1	9	1	1	1
Insert time	1	1	1	1	1	1	1	1	1	1	1	1
Reallocation time	0	1	2	0	4	0	0	0	8	0	0	0

Now, if we want to compute the full cost for our 12 inserts, we get:

$$\begin{aligned} cost &= \left(\sum_{i=1}^{12} 1\right) + (1+2+4+8) \\ &= 12 \cdot 1 + (1+2+4+8) \\ &\leq 12 + 2 \cdot 12 \\ &= 12 \cdot O(1) \end{aligned}$$

We can generalize, for $n = 2^p$:

$$\begin{aligned} cost &= \left(\sum_{i=1}^{n} 1\right) + (1 + 2 + 4 + 8 + ... + 2^{p-1}) \\ &= n \cdot 1 + \sum_{i=0}^{p-1} 2^i \\ &\leq n + 2 \cdot n \\ &= O(n) \end{aligned}$$

Thus, if n inserts take $O(n)$ time, then we say that the *amortized cost* of one INSERT is $O(1)$.

Intuitively, you can also analyze this in the following way: Each time you double your hash table size after inserting element $2^i + 1$ you pay a cost proportional to 2^i because you have to allocate a new table and rehash everything. But up until that point, you have already paid cost 2^i to insert the previous 2^i items. This new *reallocation* cost is asymptotically equal to the insertion cost so far. Thus, if we add both of them together, we can say that the cost for *all* $2^i + 1$ inserts is proportional to 2^i. Thus, the *amortized cost* for a single INSERT is $O(1)$.

Exercise: Prove that in-order traversal in a BST using SUCCESSOR (a.k.a. NEXT-LARGEST) is $O(1)$ per node, amortized.

2 Rolling Hashes and Rabin Karp

2.1 Rabin Karp pattern string search

Suppose we are given a large text t and we want to find a word (or any text pattern) s in t.

For example, consider looking for the pattern $s = bbz$ in the text $t = bbbbbcbbbz$. The trivial algorithm works by matching every character in s with characters 0, 1 and 2 in t. If there's a match, we're done. If not, we try and match the characters 1, 2 and 3 in t. If there's a match, we're done. If not, we try and match the characters 2, 3 and 4. You get the idea. Here's a table that illustrates how this algorithms runs on t and s.

Indices #	0	1	2	3	4	5	6	7	8	9
s	b	b	b	b	b	c	b	b	b	z
t	b	b	z							
s	b	b	b	b	b	c	b	b	b	z
t		b	b	z						
s	b	b	b	b	b	c	b	b	b	z
t			b	b	z					
s	b	b	b	b	b	c	b	b	b	z
t					b	b	z			
s	b	b	b	b	b	c	b	b	b	z
t						b	b	z		
s	b	b	b	b	b	c	b	b	b	z
t							b	b	z	
s	b	b	b	b	b	c	b	b	b	z
t								b	b	z

We are done! We found s at the end of t

If $|t| = n$ and $|s| = k$, then this naive algorithm runs in $O(nk)$ time because for (almost) every position in t we do exactly k comparisons to check for a match. For a big enough n and for a $k = \Theta(n)$ this algorithm will run in $O(n^2)$, which is too slow.

Can we build an $O(n)$ search algorithm?

1. Might we employ hashing somehow?
2. How do you hash a string of characters?
3. If we hash a piece of text from t that starts at index i and ends at index j, is it any easier to compute the hash of the piece that starts at index $i + 1$ and ends at index $j + 1$?
4. If we could *roll* our hashes in $O(1)$ time as described above, what would that imply about the runtime of our algorithm?

We can hash pieces of t and compare their hash with the hash of s. If there's a hash match, we check for a real match by verifying character by character. Why? We could have collisions, where s and a piece of t that's different from s hash to the same value. This is unlikely when using good hash functions but it could happen!

How can we compute the hash of a string? For this problem, we consider only strings consisting of lower-case letters. Map each letter to its index in the English dictionary $a \to 0, b \to 1, \ldots, z \to 25$. Convert the string to a base 26 number modulo some big prime p.

Example: Let $t = bciz$, then $h(t) = (26^3 \cdot 1 + 26^2 \cdot 2 + 26^1 \cdot 8 + 26^0 \cdot 25) \mod p = 19161 \mod p$.

As you can see, for strings longer than say 32 characters, we could be dealing with big numbers: $26^{32} \cdot \text{`}a\text{`} + \ldots$. Thus, we need to work modulo a prime p.

This is great, but it does not give us an $O(n)$ algorithm. Every time we compare the hash of s with the hash h of the next piece in t, we have to compute that hash h. This takes $O(k)$ time, just like comparing s with this next piece character by character did.

Let $t_{i,k}$ denote the substring in t that starts at index i and has length k. Next, we notice that if we have $h(t_{i,k})$, we can actually compute $h(t_{i+1,k})$ in $O(1)$ time, not in $O(k)$ time.

$$\textbf{bcd}\text{e} \xrightarrow{h(\text{`}bcd\text{`})} h(\text{“}bcd\text{”}) = 26^2 \cdot 1 + 26^1 \cdot 2 + 3$$
$$\text{b}\textbf{cde} \xrightarrow{h(\text{`}cde\text{`})} h(\text{“}cde\text{”}) = 26^2 \cdot 2 + 26^1 \cdot 3 + 4$$

Note that to get $h(\text{“}cde\text{”})$ from $h(\text{“}bcd\text{”})$, all we have to do is apply the following operations to $h(\text{“}bcd\text{”})$:

1. Remove $b \Rightarrow$ obtain a hash for “cd”.
2. Multiply by $26 \Rightarrow$ obtain a hash for “cda”.
3. Add $e \Rightarrow$ obtain a hash for “cde”.

Mathematically, this translates to:

$$\begin{aligned} h(\text{“}cde\text{”}) &= (h(\text{“}bcd\text{”}) - 26^2 \cdot 1) \cdot 26 + 4 \\ &= (26^2 \cdot 1 + 26^1 \cdot 2 + 3 - 26^2 \cdot 1) \cdot 26 + 4 \\ &= (26^1 \cdot 2 + 3) \cdot 26 + 4 \\ &= 26^2 \cdot 2 + 26^1 \cdot 3 + 4 \end{aligned}$$

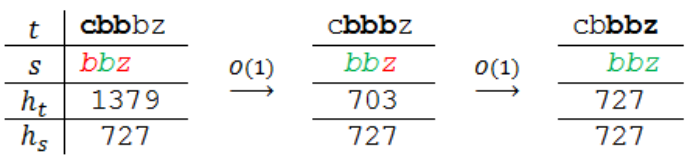
Note that all these operations are $O(1)$: we performed one subtraction, one multiplication and one addition. Also note that we ignored the mod p notation for convenience and that the calculations still hold mod p.

This new algorithm is starting to gain shape now. We can compare hashes in $O(1)$ and we can compute the hash of the next piece in t in $O(1)$ by using this *rolling hash* trick. We have an $O(n)$ time algorithm that works as follows:

RABIN-KARP-SEARCH(t, s)

```
1  n ← |t|
2  k ← |s|
3  exp ← 26^(k−1)
4
5  h_t ← hash(t_{0,k})
6  h_s ← hash(s)
7
8  if h_t == h_s and VERIFYMATCH(0, k)
9     then
10         PRINT("Found match at position 0!")
11         return 0
12
13 for i = 1 to n − k
14       do
15         ▷ Roll the hash: remove the first character
16         h_t ← h_t − exp · t[i − 1]
17         ▷ Roll the hash: multiply by 26
18         h_t ← h_t · 26
19         ▷ Roll the hash: add new character
20         h_t ← h_t + t[i − 1 + k]
21
22         ▷ Check for a match on the new hash
23         if h_t == h_s and VERIFYMATCH(i, k)
24           then
25                PRINT("Found match at position ", i, "!")
26                return i
27
28
29 PRINT("Found no match!")
30 return -1
```

The *Rabin-Karp* algorithm can verify if s occurs at a location i in t in $O(1)$ time (we exclude the extra character-by-character verification work which will not occur very often with good hashing). Since there are $O(n)$ such locations in t, the algorithm will run in $O(n)$ time.



2.2 ROLLINGHASH as an Abstract Data Structure (ADT)

We can think of the rolling hash as an *abstract data structure* that maintains the hash of a list. In Rabin Karp, the list was a sequence of characters (i.e. a piece of text), but we can generalize to any list. Such a ROLLINGHASH ADT supports the following operations, *all* in $O(1)$ time:

- `hash()` $\rightarrow$ computes the hash of the list.
- `append(val)` $\rightarrow$ adds `val` to the end of the list.
- `skip(val)` $\rightarrow$ removes the front element from the list, assuming it is `val`.
  - if the item in the front of the list is not `val`, then the behavior of the ADT from this point on will be undefined.

Key idea: Treat a list of enumerable items as a multidigit number u in base a This is basically *concatenating* the list items into a big number. What does *enumerable* mean? It means that every item in the universe of items can be assigned to a unique number in $[0, 1, 2, ...a - 1]$ (i.e. it can be assigned a digit in in base a).

For strings, characters can be interpreted as integers, with their exact values depending on what type of encoding is being used (e.g. ASCII, Unicode). In ASCII for instance, the codes for uppercase 'A' and 'B' are 65 and 66 respectively. Also, in ASCII, any character is represented as an 8-bit number so we can convert it to an integer from 0 to 255. As a result, our base would be $a = 256$ (i.e. the alphabet size for the ASCII code).

A character string $s = c_n c_{n-1} c_{n-2} \dots c_2 c_1 c_0$, where each characters c_i maps to integer d_i (as dictated by the ASCII code), would be converted to a number u in base 256 as follows:

$$u = d_n \cdot 256^n + d_{n-1} \cdot 256^{n-1} + \cdots + d_2 \cdot 256^2 + d_1 \cdot 256 + d_0$$

Let's try implementing the general HASH, APPEND and SKIP operations for our ROLLINGHASH ADT:

APPEND$(rh, item)$

```
▷ Convert our item to a number in base a
val ← GETNUMBER(rh, item)

▷ Fetch the rolling hash attributes like the current value,
▷ the base a, the prime p, the length of the list.
u ← rh.u
a ← rh.a
p ← rh.p
l ← rh.length

▷ Update our rolling hash u
u ← (u · a mod p) + val
u ← (u mod p)

▷ Store the result back in our rolling hash ADT
rh.u ← u
rh.length ← l + 1
```

SKIP$(item)$

```
▷ Convert our item to a number in base a
val ← GETNUMBER(rh, item)

▷ Fetch the rolling hash attributes like the current value, the base and the prime
u ← rh.u
a ← rh.a
p ← rh.p
l ← rh.l

▷ Compute the part that we have to remove
r ← ((a^(l−1) mod p) · val) mod p

▷ Subtract it from u and we're done
u ← (u − r) mod p

▷ Store the result back in our rolling hash ADT
rh.u ← u
rh.length ← l − 1
```

HASH(rh)

```
▷ Just return u, which is maintained modulo p
return u
```

Note the use of the GETNUMBER$(rh, item)$ function call, which is used to convert an item in the list to a number in base a. For letters, we would implement it as follows:

GETNUMBER$(rh, item)$

1 $\triangleright$ Just return the offset of the letter in the alphabet
2 **return** $alphabetOffset[item]$

Also note that we have to be *careful* when implementing SKIP so as to achieve an $O(1)$ running time. We have to compute a^{l-1} every time we SKIP and this could take $O(l)$ time, where l is the length of the list hashed by our ROLLINGHASH ADT. We note that this can be avoided by caching this power of a and storing it in our ROLLINGHASH ADT as a variable exp. But we have to keep it updated so that $exp = a^{l-1}$ across sequences of APPEND and SKIP operations. This means every time we APPEND we let $exp = (exp \cdot a) \mod p$ and every time we skip, we let $exp = (exp \cdot a^{-1}) \mod p$, where a^{-1} is the inverse modulo p of a. Initially, $exp = a^{-1}$.