1 StringBuilder

Imagine you were concatenating a list of strings, as shown below. What would the running time of this code be? For simplicity, assume that the strings are all the same length (call this x) and that there are n strings.

```
1  String joinWords(String[] words) {
2     String sentence = "";
3     for (String w : words) {
4        sentence = sentence + w;
5     )
6     return sentence;
7  )
```

On each concatenation, a new copy of the string is created, and the two strings are copied over, character by character. The first iteration requires us to copy x characters. The second iteration requires copying 2x characters. The third iteration requires 3x, and so on. The total time therefore is $O(x + 2x + ... + nx)$. This reduces to $O(xn^2)$.

Why is it $O(xn^2)$? Because $1 + 2 + ... + n$ equals $n(n+1)/2$, or $O(n^2)$.

StringBuilder can help you avoid this problem. StringBuilder simply creates a resizable array of all the strings, copying them back to a string only when necessary.

```
1  String joinWords(String[] words) {
2     StringBuilder sentence = new StringBuilder();
3     for (String w : words) {
4        sentence.append(w);
5     )
6     return sentence.toString();
7  )
```

A good exercise to practice strings, arrays, and general data structures is to implement your own version of StringBuilder, HashTable and ArrayList.

Additional Reading: Hash Table Collision Resolution (pg 522), Rabin-Karp Substring Search (pg 636).

2 ArrayList & Resizable Arrays

In some languages, arrays (often called lists in this case) are automatically resizable. The array or list will grow as you append items. In other languages, like Java, arrays are fixed length. The size is defined when you create the array.

When you need an array-like data structure that offers dynamic resizing, you would usually use an ArrayList. An ArrayList is an array that resizes itself as needed while still providing O(1) access. A typical implementation is that when the array is full, the array doubles in size. Each doubling takes O(n) time, but happens so rarely that its amortized insertion time is still O(1).

```
1  ArrayList<String> merge(String[] words, String[] more) {
2    ArrayList<String> sentence = new ArrayList<String>();
3    for (String w : words) sentence.add(w);
4    for (String w : more) sentence.add(w);
5    return sentence;
6  )
```

This is an essential data structure for interviews. Be sure you are comfortable with dynamically resizable arrays/lists in whatever language you will be working with. Note that the name of the data structure as well as the "resizing factor" (which is 2 in Java) can vary.

Why is the amortized insertion runtime O(1)?

Suppose you have an array of size N. We can work backwards to compute how many elements we copied at each capacity increase. Observe that when we increase the array to K elements, the array was previously half that size. Therefore, we needed to copy $K/2$ elements.

```
final capacity increase    : n/2 elements to copy
previous capacity increase : n/4 elements to copy
previous capacity increase : n/8 elements to copy
previous capacity increase : n/16 elements to copy
...
second capacity increase   : 2 elements to copy
first capacity increase    : 1 element to copy
```

Therefore, the total number of copies to insert N elements is roughly $N/2 + N/4 + N/8 + \ldots + 2 + 1$, which is just less than N.

> If the sum of this series isn't obvious to you, imagine this: Suppose you have a kilometer—long walk to the store. You walk 0.5 kilometers, and then 0.25 kilometers, and then 0.125 kilometers, and so on. You will never exceed one kilometer (although you'll get very close to it).

Therefore, inserting N elements takes O(N) work total. Each insertion is O(1) on average, even though some insertions take O(N) time in the worst case.

Concatenation of Strings using String:

```
String joinWords(String[] words){
    String sentence = "";
    for(String w: words){
        sentence = sentence + w;
    }
    return sentence;
}
```

Complexity: $O(n^2)$

Concatenation of Strings using ~~String~~ StringBuilder:

```
String joinWords(String[] words){
    StringBuilder sentence = new
                             StringBuilder();
    for(String w: words){
        sentence.append(w);
    }
    return sentence.toString();
}
```

Merging Strings in Dynamic Arrays.

```
ArrayList<String> merge(String[] words,
                        String[] more) {

    ArrayList<String> sentence =
                        new ArrayList<String>();
    for(String w: words) sentence.add(w);
    for(String w: more) sentence.add(w);
    return sentence;

}
```

3 Hash Tables

A hash table is a data structure that maps keys to values for highly efficient lookup. There are a number of ways of implementing this. Here, we will describe a simple but common implementation.

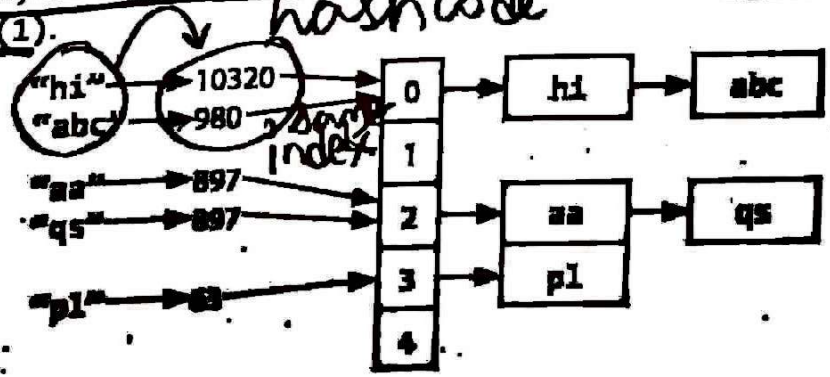
In this simple implementation, we use an array of linked lists and a hash code function. To insert a key (which might be a string or essentially any other data type) and value, we do the following:

1. First, compute the key's hash code, which will usually be an `int` or `long`. Note that two different keys could have the same hash code, as there may be an infinite number of keys and a finite number of ints.
2. Then, map the hash code to an index in the array. This could be done with something like `hash (key) % array_length`. Two different hash codes could, of course, map to the same index.
3. At this index, there is a linked list of keys and values. Store the key and value in this index. We must use a linked list because of collisions: you could have two different keys with the same hash code, or two different hash codes that map to the same index.

key → hashCode → index → LL

To retrieve the value pair by its key, you repeat this process. Compute the hash code from the key, and then compute the index from the hash code. Then, search through the linked list for the value with this key.

If the number of collisions is very high, the worst case runtime is O(N), where N is the number of keys. However, we generally assume a good implementation that keeps collisions to a minimum, in which case the lookup time is O(1).



Alternatively, we can implement the hash table with a balanced binary search tree. This gives us an O(log N) lookup time. The advantage of this is potentially using less space, since we no longer allocate a large array. We can also iterate through the keys in order, which can be useful sometimes.

Rabin-Karp Substring Search

The brute force way to search for a substring S in a larger string B takes O(s(b-s)) time, where s is the length of S and b is the length of B. We do this by searching through the first b - s + 1 characters in B and, for each, checking if the next s characters match S.

The Rabin-Karp algorithm optimizes this with a little trick: if two strings are the same, they must have the same hash value. (The converse, however, is not true. Two different strings can have the same hash value.)

Therefore, if we efficiently precompute a hash value for each sequence of s characters within B, we can find the locations of S in O(b) time. We then just need to validate that those locations really do match S.

For example, imagine our hash function was simply the sum of each character (where space = 0, a = 1, b = 2, and so on). If S is `ear` and `B = doe are hearing me`, we'd then just be looking for sequences where the sum is 24 (e + a + r). This happens three times. For each of those locations, we'd check if the string really is `ear`.

char	d	o	e		a	r	e		h	e	a	r	i	n	g		m	e
code:	4	15	5	0	1	18	5	0	8	5	1	18	9	14	7	0	13	5
sum of next 3:	24	20	6	19	24	23	13	13	14	24	28	41	30	21	20	18		

If we computed these sums by doing `hash('doe')`, then `hash('oe ')`, then `hash('e a')`, and so on, we would still be at $O(s(b-s))$ time. longer time

Instead, we compute the hash values by recognizing that `hash('oe ') = hash('doe') - code('d') + code(' ')`. This takes $O(b)$ time to compute all the hashes. optimised

You might argue that, still, in the worst case this will take $O(s(b-s))$ time since many of the hash values could match. That's absolutely true—for this hash function.

In practice, we would use a better *rolling* hash function, such as the Rabin fingerprint. This essentially treats a string like doe as a base 128 (or however many characters are in our alphabet) number.

$$\text{hash('doe')} = \text{code('d')} * 128^2 + \text{code('o')} * 128^1 + \text{code('e')} * 128^0$$

This hash function will allow us to remove the d, shift the o and e, and then add in the space.

$$\text{hash('oe ')} = (\text{hash('doe')} - \text{code('d')} * 128^2) * 128 + \text{code(' ')}$$

This will considerably cut down on the number of false matches. Using a good hash function like this will give us expected time complexity of $O(s + b)$, although the worst case is $O(sb)$.

Usage of this algorithm comes up fairly frequently in interviews, so it's useful to know that you can identify substrings in linear time.

Hash Table Collision Resolution

Essentially any hash table can have collisions. There are a number of ways of handling this.

Chaining with Linked Lists

With this approach (which is the most common), the hash table's array maps to a linked list of items. We just add items to this linked list. As long as the number of collisions is fairly small, this will be quite efficient.

In the worst case, lookup is O(n), where n is the number of elements in the hash table. This would only happen with either some very strange data or a very poor hash function (or both).

Chaining with Binary Search Trees

Rather than storing collisions in a linked list, we could store collisions in a binary search tree. This will bring the worst-case runtime to O(log n).

In practice, we would rarely take this approach unless we expected an extremely nonuniform distribution.

Terminology; Storage of Strings

3.1 Let W be the string ABCD. (*a*) Find the length of W. (*b*) List all substrings of W. (*c*) List all the initial substrings of W.

(a) The number of characters in W is its length, so 4 is the length of W.

Greek letter big lambda

(b) Any subsequence of characters of W is a substring of W. There are 11 such substrings:

Substrings:	ABCD,	ABC, BCD,	AB, BC, CD,	A, B, C, D,	Λ
Lengths:	4	3	2	1	0

(Here Λ denotes the empty string.)

(c) The initial substrings are ABCD, ABC, AB, A, Λ; that is, both the empty string and those substrings that begin with A.

3.7 PATTERN MATCHING ALGORITHMS

Pattern matching is the problem of deciding whether or not a given string pattern P appears in a string text T. We assume that the length of P does not exceed the length of T. This section discusses two pattern matching algorithms. We also discuss the complexity of the algorithms so we can compare their efficiencies.

Remark: During the discussion of pattern matching algorithms, characters are sometimes denoted by lowercase letters (a, b, c, ...) and exponents may be used to denote repetition; e.g.,

$$a^2b^3ab^2 \text{ for } aabbbabb \quad \text{and} \quad (cd)^3 \text{ for } cdcdcd$$

In addition, the empty string may be denoted by Λ, the Greek letter lambda, and the concatenation of strings X and Y may be denoted by $X \cdot Y$ or, simply, XY.

3.6 WORD PROCESSING

In earlier times, character data processed by the computer consisted mainly of data items, such as names and addresses. Today the computer also processes printed matter, such as letters, articles and reports. It is in this latter context that we use the term "word processing."

Given some printed text, the operations usually associated with word processing are the following:

(a) *Replacement.* Replacing one string in the text by another.
(b) *Insertion.* Inserting a string in the middle of the text.
(c) *Deletion.* Deleting a string from the text.

(1) Insertion

Suppose in a given text T we want to insert a string S so that S begins in position K. We denote this operation by

T K S

INSERT(text, position, string)

For example,

1 2 3

INSERT ('ABCDEFG', 3, 'XYZ') = 'ABXYZCDEFG'

1 2 3 4 5 6

INSERT ('ABCDEFG', 6, 'XYZ') = 'ABCDEXYZFG'

This INSERT

section as follows:

INSERT(T, K, S) = SUBSTRING(T, 1, K−1) //S// SUBSTRING(T, K, LENGTH(T)−K+1)

This is, the initial substring of T before the position K, which has length K − 1, is concatenated with the string S, and the result is concatenated with the remaining part of T, which begins in position K and has length LENGTH(T) − (K − 1) = Length (T) − K + 1. (We are assuming implicitly that T is a dyamic variable and that the size of T will not become too large.)

2 **Deletion**

2 Deletion

Suppose in a given text T we want to delete the substring which begins in position K and has length L. We denote this operation by

DELETE(text, position, length)

T K L

For example,

DELETE('ABCDEFG', 4, 2) = 'ABCFG'
DELETE('ABCDEFG', 2, 4) = 'AFG'

We assume that nothing is deleted if position K = 0. Thus

DELETE('ABCDEFG', 0, 2) = 'ABCDEFG'

The importance of this "zero case" is seen later.

The DELETE function can be implemented using the string operations given in the preceding section as follows:

DELETE(T, K, L) =
SUBSTRING(T, 1, K – 1)//SUBSTRING(T, K + L, LENGTH(T) – K – L + 1)

That is, the initial substring of T before position K is concatenated with the terminal substring of T beginning in position K + L. The length of the initial substring is K – 1, and the length of the terminal substring is:

LENGTH(T) – (K + L – 1) = LENGTH(T) – K – L + 1

We also assume that DELETE(T, K, L) = T when K = 0.

Now suppose text T and pattern P are given and we want to delete from T the first occurrence of the pattern P. This can be accomplished by using the above DELETE function as follows:

DELETE(T, INDEX(T, P), LENGTH(P))

That is, in the text T, we first compute INDEX(T, P), the position where P first occurs in T, and then we compute LENGTH(P), the number of characters in P. Recall that when INDEX(T, P) = 0 (i.e., when P does not occur in T) the text T is not changed.

Suppose after reading into the computer a text T and a pattern P, we want to delete every occurrence of the pattern P in the text T. This can be accomplished by repeatedly applying

DELETE(T, INDEX(T, P), LENGTH(P))

until INDEX(T, P) = 0 (i.e., until P does not appear in T). An algorithm which accomplishes this follows.

4. Exit.

We emphasize that after each deletion, the length of T decreases and hence the algorithm must stop. However, the number of times the loop is executed may exceed the number of times P appears in the original text T, as illustrated in the following example.

Example 3.7

The above example shows that when a text T is changed by a deletion, patterns may occur that did not appear originally.

QUICK REVIEW

1. A data type is simple if variables of that type can hold only one value at a time.
2. In a structured data type, each data item is a collection of other data items.
3. An array is a structured data type with a fixed number of components. Every component is of the same type, and components are accessed using their relative positions in the array.
4. Elements of a one-dimensional array are arranged in the form of a list.
5. There is no check on whether an array index is out of bounds.