Membership Structures

8

In Chap. 5 we covered data structures that support insertion, deletion, membership testing, and iteration. For some applications testing membership may be enough. Iteration and deletion may not be necessary. The classic example is that of a spell checker. Consider the job of a spell checker. A simple one may detect errors in spelling while a more advanced spell checker may suggest alternatives of *correctly* spelled words.

Clearly a spell checker is provided with a large dictionary of words. Using the list of words the spell checker determines whether a word you have is in the dictionary and therefore a correct word. If the word does not appear in the dictionary the word processor or editor may underline the word indicating it may be incorrectly spelled. In some cases the word processor may suggest an alternative, correctly spelled word. In some cases, the word processor may simply correct the misspelling. How do these spell checkers/correctors work? What kind of data structures do they use?

8.1 Chapter Goals

At first glance, a hash set (i.e. a Python dictionary) might seem an appropriate data structure for spell checking. Lookup time within the set could be done in O(1) time. However, the tradeoff is in the size of this hash map. A typical English dictionary might contain over 100,000 words. The amount of space required to store that many words would be quite large.

In this chapter we'll cover two data structures that are designed to test membership within a set. The first, a bloom filter, has significantly smaller space requirements and provides a very fast membership test. The other is a trie (pronounced try) data structure which has features that would not be readily available to a hash set implementation and may take up less space than a hash set.

K.D. Lee and S. Hubbard, *Data Structures and Algorithms with Python*,
Undergraduate Topics in Computer Science, DOI 10.1007/978-3-319-13072-9_8

8.2 Bloom Filters

Bloom filters are named for their creator, Burton Howard Bloom, who originally proposed this idea in 1970. Since then many authors have covered the implementations of bloom filters including Alan Tharp [7]. Wikipedia, while not always the authoritative source, has a very good discussion of bloom filters as well [8].

A bloom filter shares some ideas with hash sets while using considerably less space. A bloom filter is a data structure employing statistical probability to determine if an item is a member of a set of values. Bloom filters are not 100 % accurate. A bloom filter will never report a false negative for set membership, meaning that they will never report that an item doesn't belong to a set when it actually does. However, a bloom filter will sometimes report a false positive. It may report an item is in a set when it is actually not.

Consider the problem of spell checking. A spell checker needs to know if a typed word is correctly typed by looking it up in the dictionary. With a bloom filter, the typed word can be given to the bloom filter which will report that it is or is not a correctly typed word. In some cases it may report a word is correct when it is not.

A bloom filter is an array of bits along with a set of hashing functions. The number of bits in the filter and the number of hashing functions influences the accuracy of the bloom filter. The exact number of bits and hash functions will be discussed later. Consider a bloom filter with 20 bits and 3 independent hash functions. Initially all the bits in the filter are set to 0 as shown in Fig. 8.1.

Consider adding the word *cow* to the bloom filter. Assume that three independent hash functions hash the word cow, modulo 20, to 18, 9, and 3 respectively. The bits at indices 18, 9, and 3 are set to 1 to *remember* that *cow* has been added to the filter as shown in Fig. 8.2.

Now consider adding the word *cat* to the same filter. Assume the hash values from the three hash functions, modulo 20, are 0, 3, and 9. Inserting *cat* into the filter results in setting the bit at index 0 to a 1. The other two were already set by inserting *cow*. Finally, inserting *dog* into the filter results in the bloom filter shown in Fig. 8.3. The hash values for *dog* are 10, 9, and 8.

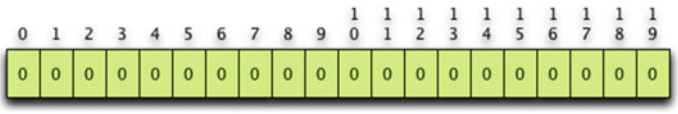

Fig. 8.1 An Empty Bloom Filter

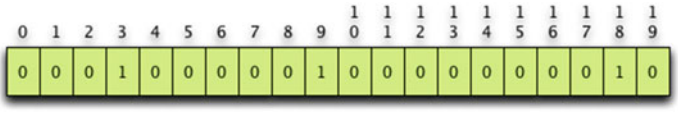

Fig. 8.2 After Inserting *cow* into the Bloom Filter

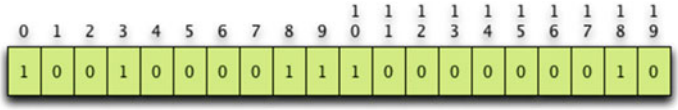

Fig. 8.3 After Inserting *cow*, *cat*, and *dog* into the Bloom Filter

Looking up an item in a bloom filter requires hashing the value again with the same hash functions generating the indices into the bit array. If the value at all indices in the bit array are one, then the lookup function reports success and otherwise failure.

Consider looking up a value that is not in the bloom filter of Fig. 8.3. If we look up *fox* the three hash function calls return 3, 12, and 18. The digit at indices 3 and 18 is a *1*. However, the digit at index 12 is a 0 and the lookup function reports that *fox* is not in the bloom filter.

Consider looking up the value *rabbit* in the same bloom filter. Hashing *rabbit* with the three hash functions results in values 8, 9, and 18. All three of the digits at these locations within the bloom filter contain a *1* and the bloom filter incorrectly reports that *rabbit* has been added to the filter. This is a false positive and while not desirable, must be acceptable if a bloom filter is to be used.

If a bloom filter is to be useful, it must never report a false negative. From these examples it should be clear that false negatives are impossible. False positives must be kept to a minimum. In fact, it is possible to determine on average how often a bloom filter will report a false positive. The probability calculation depends on three factors: the hashing functions, the number of items added to the bloom filter, and the number of bits used in the bloom filter. The analysis of these factors are covered in the next sections.

8.2.1 The Hashing Functions

Each item added to a bloom filter must be hashed by some number of hash functions which are completely independent of each other. Each hashing function must also be evenly distributed over the range of bit indices in the bit array. This second requirement is true of hashing functions for hash sets and hash tables as well. Uniform distribution is guaranteed by the built-in hash functions of Python and most other languages.

In the examples above, three hashing functions were required. Sometimes the required number of hashing functions can be much higher, depending on the number of items being inserted and the number of bits in the bit array. Creating the required number of independent, uniformly distributed hashing functions might seem like a daunting problem, but it can be solved in at least a couple of ways. Some hashing functions allow a seed value to be provided. In this case, different seed values could be used to create different hashing functions.

Another equally effective way of generating independent hashing functions is to append some known value to the end of each item before it is hashed. For instance, a *0* might be appended to the item before hashing it to get the first hash function. A *1* could be appended to the item before hashing to get the second hash function. Likewise, a *2* might be appended to get the third hash function value. So looking up *rabbit* in the bloom filter is accomplished by first hashing *rabbit0*, *rabbit1*, and *rabbit2* with the same hashing function. Since the hashing function is uniformly distributed, the values returned by the three hashed values will be independent of each other. And, all items with *0* appended will themselves be uniformly distributed. Likewise for items with *1* appended and with *2* appended.

8.2.2 The Bloom Filter Size

It is possible to find the required bloom filter size given a number of items to insert and a desired false positive probability. The probability of any one location within a bloom filter not being set by a hash function while inserting an item is given by the following formula where the filter consists of m bits.

$$1 - \frac{1}{m}$$

If the bloom filter uses k hash functions, then the probability that a bit in the bit array is not set by any of the hash functions required for inserting an item is given by this formula.

$$\left(1 - \frac{1}{m}\right)^k$$

If n items are inserted into the bloom filter then raising this formula to n will provide the probability that a bit within the bloom filter's bit array is still a zero after inserting all n items. So we have

$$\left(1 - \frac{1}{m}\right)^{nk}$$

So, the probability that a bit in the bloom filter is a *1* after inserting n items while using k hashing functions is given by this formula.

$$1 - \left(1 - \frac{1}{m}\right)^{nk}$$

Now consider looking up an item that was not added to the bloom filter. The probability that it will report a false positive can be found by computing the likelihood that each location within the bloom filter is a 1 for all k hashing functions. This is expressed as follows.

$$p = \left(1 - \left(1 - \frac{1}{m}\right)^{nk}\right)^k$$

This formula contains a sequence that can be approximated using the natural log [8] as

$$p = \left(1 - e^{kn/m}\right)^k$$

Using this formula it is possible to solve for m given an n and desired probability, p, of false positives. The formula is as follows.

$$m = -\frac{n\ ln\ p}{(ln\ 2)^2}$$

Finally, solving for k above results in the following formula.

$$k = \frac{m}{n} ln\ 2$$

These two formulas tell us how many bits are required in our filter to guarantee a maximum specified rate of false positives. We can also compute the required number of hash functions. For instance, for an English dictionary containing 109,583 words and a desired false postive percentage of no more than 1 % (expressed as 0.01 in the formula) requires a bit array of 1,050,360 bits and seven hashing functions.

The number of bits in this example may seem excessive. However, recall that they are bits. An efficient implementation requires roughly 128 KB of storage. The number of characters in the English dictionary used in these examples totals 935,171. Assuming 1 byte per character, storing all these words would require a minimum of 914 KB. The bloom filter represents quite a savings in space. In addition, during experiments the lookup time using the bloom filter never took longer than 160 µs. The lookup time is bounded by the number and efficiency of the hash functions used to compute the desired values. Assuming that the hash functions are dependent on the length of the string being hashed, then the lookup time is O(lk) where l is given by the length of the item being looked up and k is the number of hash functions.

8.2.3 Drawbacks of a Bloom Filter

Besides the obvious false positive potential, the bloom filter can only report *yes* or *no*. It can't suggest alternatives for items that might be close to being spelled correctly. A bloom filter has no memory of which bits were set by which items so a *yes* or *no* answer is the best we can get with even a *yes* answer not being correct in some circumstances. The next section presents a *Trie* data structure that will not report false positives and can be used to find alternatives for incorrectly spelled words.

8.3 The Trie Datatype

A trie is a data structure that is designed for re*TRIE*val. The data structure is pronounced like the word *try*. A trie is not meant to be used when deleting values from a data structure is required. It is meant only for retrieval of items based on a key value.

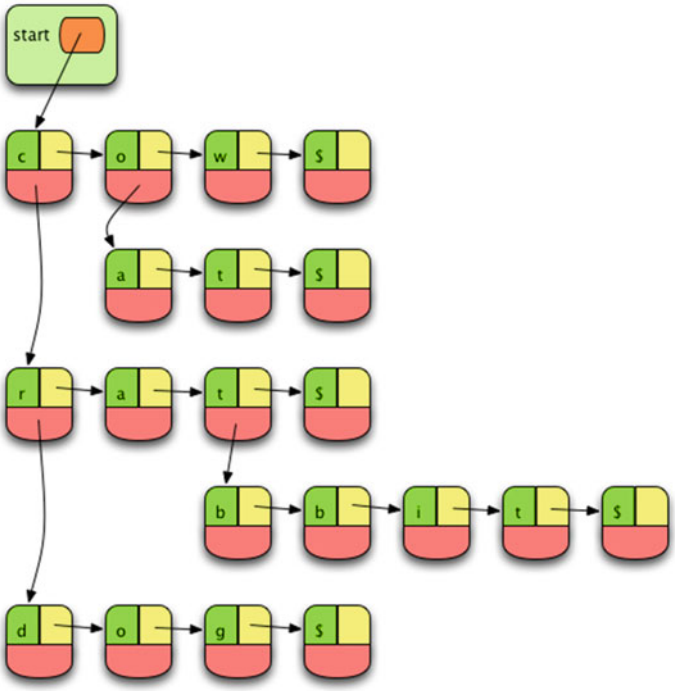

Fig. 8.4 After Inserting *cow*, *cat*, *rat*, *rabbit*, and *dog* into a Trie

Tries are appropriate when key values are made up of more than one unit and when the individual units of a key may overlap with other item keys. In fact, the more overlap the key units have, the more compact the trie data structure.

In the problem of spell checking, words are made up of characters. These characters are the individual units of the keys. Many words overlap in a dictionary like *a*, *an*, and *ant*. A trie may be implemented in several different ways. In this text we'll concentrate on the linked trie which is a series of link lists making up a matrix. Matrix implementations lead to sparsely populated arrays which take up much more room with empty locations. A linked trie has overhead for pointers, but is not sparsely populated.

The trie data structure begins with an empty linked list. Each node in the linked trie list contains three values: a unit of the key (in the spellchecker instance this is a character of the word), a next pointer that points to the next node in the list which would contain some other unit (i.e. character) appearing at the same position within a key (i.e. word), and a follows pointer which points at a node that contains the next unit within the same key. In Fig. 8.4 the follows pointer is in yellow while the next pointer field is in red.

When items are inserted into the trie a sentinel unit is added. In the case of the spell checker, a '$' character is appended to the end of every word. The sentinel is needed because words like *rat* are prefixes to words like *ratchet*. Without the sentinel character it would be unclear whether a word ended or was only a prefix of some other word.

In a trie keys with a common prefix share that prefix and are not repeated. The *next* pointer is used when more than one possible next character is possible. This saves space in the data structure. The trade-off is that the *next* and *follows* pointers take extra space in each node.

8.3.1 The Trie Class

```
class Trie:
    def __insert(node,item):
        # This is the recursive insert function.

    def __contains(node,item):
        # This is the recursive membership test.


    class TrieNode:
        def __init__(self,item,next = None, follows = None):
            self.item = item
            self.next = next
            self.follows = follows

    def __init__(self):
        self.start = None

    def insert(self,item):
        self.start = Trie.__insert(self.start,item)

    def __contains__(self,item):
        return Trie.__contains(self.start,item)
```

8.3.2 Inserting into a Trie

Inserting values into a trie can be done either iteratively, with a loop, or recursively. To recursively insert into a trie the *insert* method can call an *__insert* function. It is easier to write the recursive code as a function and not a method of the *Trie* class because the *node* value passed to the function may be *None*. To insert into the trie, the *__insert* function operates as follows.

1. If the key is empty (i.e. no units are left in the key), return *None* as the empty node.
2. If the node is *None* then a new node is created with the next unit of the key and the rest of the key is inserted and added to the *follows* link.

3. If the first unit of the key matches the unit of the current node, then the rest of the key is inserted into the *follows* link of the node.
4. Otherwise, the key is inserted into the *next* link of the node.

Building the trie recursively is simple. However, an iterative version would work just as well. The iterative version would require a loop and a pointer to the current node along with remaining key to insert. The iterative insert algorithm would behave in a similar fashion to the step outlined above but would need to keep track of the previous node as well as the current node so that links could be set correctly.

8.3.3 Membership in a Trie

Checking membership in a trie can also be accomplished recursively. The steps include a base case which might not be completely intuitive at first. The empty key is reported as a member of any trie because it works when checking membership. With the sentinel unit added to the trie, returning *True* for an empty key is completely safe because any real key will at least consist of the sentinel character. In the algorithm outlined here the sentinel is assumed to have already been added to the key. The steps for membership testing are as follows.

1. If the length of the key is 0, then report success by returning *True*.
2. If the node we are looking at is *None* then report failure by returning *False*.
3. If the first unit of the key matches the unit in the current node, then check membership of the rest of the key starting with the *follows* node.
4. Otherwise, check membership of the key starting with the *next* node in the trie.

Again, this code might be implemented iteratively with a while loop keeping track of the current node and the remainder of the key. Either a recursive or iterative implementation will work equally well.

8.3.4 Comparing Tries and Bloom Filters

Bloom filters are clearly faster for testing membership than a trie. However, the trie works acceptably well. While the longest bloom filter lookup time in a simple experiment was 160 μs, the longest trie lookup was 217 μs. Of course the trie takes more space, but common prefixes share nodes in a trie saving some space over storing each word distinctly in a data structure, as in a hash set.

For purposes of spell checking a trie has distinct advantages, since spelling alternatives can be easily found. Common typographical errors fall into one of four categories.

- Transposition of characters like *teh* instead of *the*
- Dropped characters like *thei* instead of *their*
- Extra characters like *thre* instead of *the*
- Incorrect characters like *thare* instead of *there*

If in searching in a trie a word is not found, these alternatives can also be searched for to find a selection of alternative spellings. What's more, these alternative spellings can be searched in parallel in a trie to quickly put together a list of alternatives. A bloom filter cannot be used to find alternative spellings since that information is lost once entered into the filter. Of course, a trie will never report a false positive either as is possible with a bloom filter.

8.4 Chapter Summary

Tries and bloom filters are two data structures for testing membership. Bloom filters are relatively small and will produce false positives some percentage of the time. Tries are larger, don't produce false positives, and can be used to find alternative key values that *are close* to the key being sought. While either data structure will work for spell checking, spelling correction would be aided by a trie while a bloom filter would not help.

As far as efficiency goes, bloom filters more efficiently test set membership, subject to the false positives that are sometimes produced. However, a trie also operates efficiently while also taking more space than a bloom filter. Both the bloom filter and the trie tested membership of words in the dictionary in microseconds. The bloom filter's worst time was 160 μs while the trie's worst time was 217 μs for the informal test performed on both.

Size requirements are also a concern of course. The example dictionary used in the development of both the bloom filter and the trie in this chapter contained 109,583 words. The bloom filter for this dictionary of words was approximately 128 KB in size. Assuming that the *next* and *follows* pointers take 4 bytes each and the key units (i.e. word characters) take 1 byte each, the size of the trie is roughly 3.1 MB in size. While the bloom filter is much smaller than the trie, both are well within the limits of what computers are capable of storing.

8.5 Review Questions

Answer these short answer, multiple choice, and true/false questions to test your mastery of the chapter.

1. Which datatype, the trie or the bloom filter, is susceptible to false positives?
2. What is a false positive in this context?

3. A bloom filter requires more or less storage than a trie?
4. When spell checking, which data type can be used for spelling correction?
5. How can you generate more than one hashing function for use in a bloom filter?
6. Add the words "a", "an", "ant", "bat", and "batter" to a trie. Draw the trie data structure showing its structure after inserting the words in the order given here.
7. Why is a sentinel needed in a trie?
8. Why is a sentinel not needed in a bloom filter?
9. What must be true of keys to be able to store them in a trie?
10. Which datatype, *trie* or *bloom filter*, is more efficient in terms of space? Which is more efficient in terms of speed?

8.6 Programming Problems

1. Go to the text website and download the dictionary of words. Build a bloom filter for this list of words and use it to spellcheck the declaration of independence, printing all the misspelled words to the screen.
2. Go to the text website and download the dictionary of words. Build a trie datatype for this list of words and use it to spellcheck the declaration of independence, printing all misspelled words to the screen.
3. Create a trie as in the previous exercise, but also print suggested replacements for all misspelled words. This is a tough assignment. Suggested replacements should not differ from the original in more than one of the ways suggested in the chapter.