# Lecture Notes on
# Tries

15-122: Principles of Imperative Computation
Thomas Cortina, Frank Pfenning, Rob Simmons, Penny Anderson

Lecture 23
April 9, 2015

## 1 Introduction

In the data structures implementing associative arrays so far, we have needed either an equality operation and a hash function, or a comparison operator with a total order on keys. Similarly, our sorting algorithms just used a total order on keys and worked by comparisons of keys. We obtain a different class of representations and algorithms if we analyze the structure of keys and decompose them. In this lecture we explore *tries*, an example from this class of data structures. The asymptotic complexity we obtain has a different nature from data structures based on comparisons, depending on the structure of the key rather than the number of elements stored in the data structure.

## 2 The Boggle Word Game

The Boggle word game is played on an $n \times n$ grid (usually $4 \times 4$ or $5 \times 5$). We have $n * n$ dice that have letters on all 6 sides and which are shaken so that they randomly settle into the grid. At that point we have an $n \times n$ grid filled with letters. Now the goal is to find as many words as possible in this grid within a specified time limit. To construct a word we can start at an arbitrary position and use any of the 8 adjacent letters as the second letter. From there we can again pick any adjacent letter as the third letter in the word, and so on. We may not reuse any particular place in the grid in the same word, but they may be in common for different words. For example,

in the grid

| E | F | R | A |
|---|---|---|---|
| H | G | D | R |
| P | S | N | A |
| E | E | B | E |

we have the words SEE, SEEP, and BEARDS, but not SEES. Scoring assigns points according to the lengths of the words found, where longer words score higher.
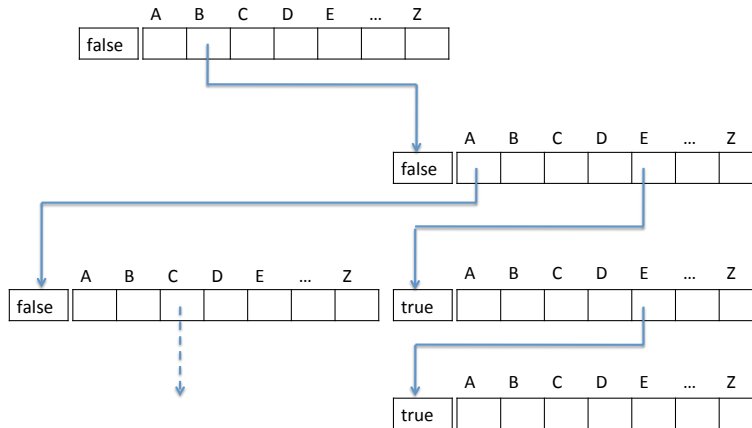
One simple possibility for implementing this game is to systematically search for potential words and then look them up in a dictionary, perhaps stored as a sorted word list, some kind of binary search tree, or a hash table. The problem is that there are too many potential words on the grid, so we want to consider prefixes and abort the search when a prefix does not start a word. For example, if we start in the upper right-hand corner and try horizontally first, then EF is a prefix for a number of words, but EFR, EFD, EFG, EFH are not and we can abandon our search quickly. A few more possibilities reveal that no word with 3 letters or more in the above grid starts in the upper left-hand corner.

Because a dictionary is sorted alphabetically, by prefix, we may be able to use a sorted array effectively in order for the computer to play Boggle and quickly determine all possible words on a grid. But we may still look for potentially more efficient data structures which take into account that we are searching for words that are constructed by incrementally extending the prefix.

## 3   Multi-Way Tries

One possibility is to use a *multi-way trie*, where each node has a potential child for each letter in the alphabet. Consider the word SEE. We start at the root and follow the link labeled S, which gets us to a node on the second level in the tree. This tree indexes all words with first character S. From here we follow the link labeled E, which gets us to a node indexing all words that start with SE. After one more step we are at SEE. At this point we cannot be sure if this is a complete word or just a prefix for words stored in it. In order to record this, we can either store a Boolean (`true` if the current prefix is a complete word) or terminate the word with a special character that cannot appear in the word itself.

Below is an example of a multi-way trie indexing the three words BE, BEE, and BACCALAUREATE.

A   B   C   D   E   ...   Z
false

A   B   C   D   E   ...   Z
false

A   B   C   D   E   ...   Z
false

A   B   C   D   E   ...   Z
true

A   B   C   D   E   ...   Z
true

While the paths to finding each word are quite short, including one more node than characters in the word, the data structure consumes a lot of space, because there are a lot of nearly empty arrays.

An interesting property is that the lookup time for a word is $O(k)$, where $k$ is the number of characters in the word. This is independent of how many words are stored in the data structure! Contrast this with, say, balanced binary search trees where the search time is $O(\log(n))$, where $n$ is the number of words stored. For the latter analysis we assumed that key comparisons were constant time, which is not really true because the keys (which are strings) have to be compared character by character. So each comparison, while searching through a binary search tree, might take up to $O(k)$ individual character comparisons, which would make it $O(k * \log(n))$ in the worst case. Compare that with $O(k)$ for a trie.

On the other hand, the wasted space of the multi-way trie with an array at each node costs time in practice. This is not only because this memory must be allocated, but because on modern architectures the so-called *memory hierarchy* means that accesses to memory cells close to each other will be much faster than accessing distant cells. You will learn more about this in 15-213 *Computer Systems*.

## 4  Binary Tries

The idea of the multi-way trie is quite robust, and there are useful special cases. One of these if we want to represent sets of numbers. In that case we can decompose the binary representation of numbers bit by bit in order to index data stored in the trie. We could start with the most significant or least significant bit, depending on the kind of numbers we expect. In this case every node would have at most two successors, one for $0$ and one for $1$. This does not waste nearly as much space and can be efficient for many purposes.

## 5  Linked Lists

For the particular application we have in mind, namely searching for words on a grid of letters, we could either use multiway tries directly (wasting space) or use binary tries (wasting time and space, because each character is decomposed into individual bits).

A compromise solution is replacing the array (which may end up mostly empty) with a linked list. This gives us two fundamentally different uses of pointers. Child pointers (drawn in blue) correspond to forward movement through the string. The next pointers of the linked list (drawn in red), on the other hand, connect what used to be parts of the same array list.

In this representation, it also becomes natural to have the Boolean "end of word" flag stored with the final character, rather than one step below the final character like we did above. (This means it's no longer possible to store the empty string, however.) The tree above, containing BACCALAU-REATE, BE, and BEE, now looks like this:
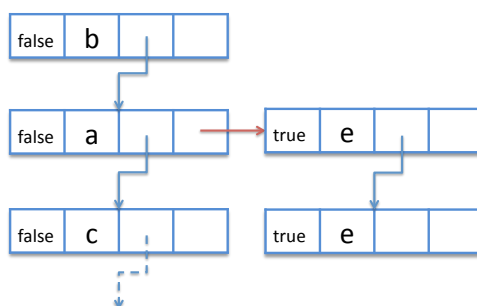
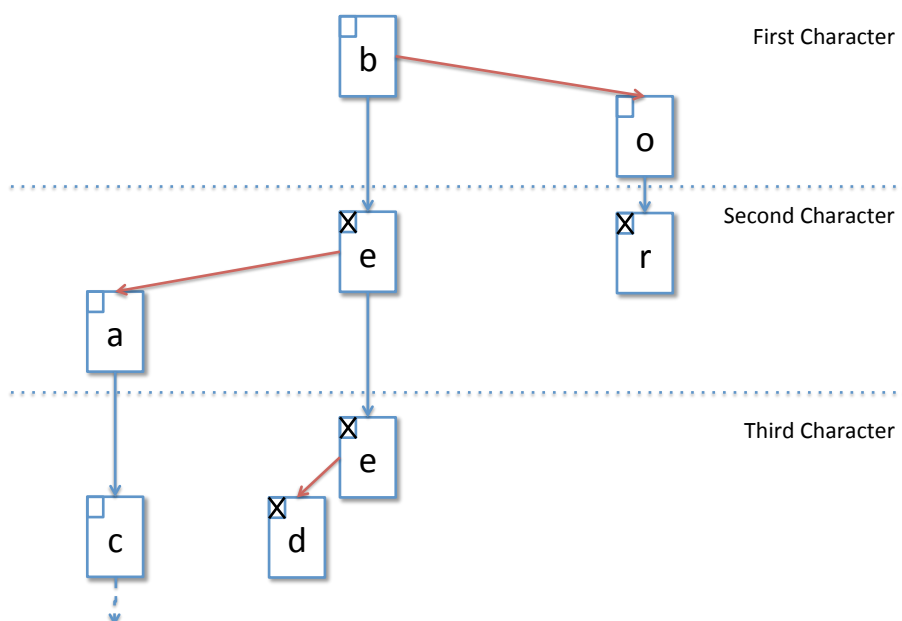If we add OR and BED, the result looks like this:
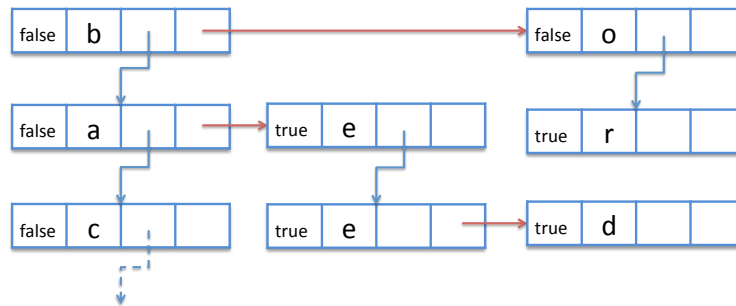


For lookup, we have to make at most 26 comparisons between each character in the input string and the characters stored in the tree. Therefore search time is $O(26*k) = O(k)$, where $k$ is the length of the string. Insertion has the same asymptotic complexity bound. Note that this still does not change with the number of strings, only with its length.

## 6 Ternary Search Tries

It should be apparent that we could get some performance gains over this linked list solution if we kept the linked lists sorted. This is an idea that allows us to motivate a more suitable data structure, a *ternary search trie* (TST) which combines ideas from binary search trees with tries. Roughly, at each node in a trie we store a binary search tree with characters as keys. The entries are pointers to the subtries.

More precisely, at each node we store a character $c$ and three pointers. The left subtree stores all words starting with characters alphabetically less than $c$. The right subtree stores all words starting with characters alphabetically greater than $c$ and the middle stores a subtrie with all words starting with $c$, from the second character on. The middle children work exactly like the child pointers from our linked list implementation. The left and right pointers work like the next pointers from the linked list version, and are correspondingly drawn in red in the diagram below, which contains the words BE, BED, BEE, BACCALAUREATE, and OR. In this diagram, we represent the flag for "end of word" using the presence or absence of an X.

We have not discussed any strategy for balancing TSTs. However, in the worst case (a completely unbalanced tree) we end up with something similar to the linked list implementation and we have to make at most 26 comparisons between each character in the input string and the characters stored in the tree. Therefore search time is $O(26 * k) = O(k)$, where $k$ is the length of the string. Even when the embedded trees are perfectly balanced, the constant factor decreases, but not the asymptotic complexity because $O(\log(26) * k) = O(k)$.

## Exercises

**Exercise 1** *Implement the game of Boggle as sketched in this lecture. Make sure to pick the letters according to the distribution of their occurrence in the English language. You might use the Scrabble dictionary itself, for example, to calculate the relative frequency of the letters.*

*If you are ambitious, try to design a simple textual interface to print a random grid and then input words from the human player and show the words missed by*

*the player.*

**Exercise 2** *Modify the implementation TSTs so it can store, for each word, the number of occurrences of that word in a book that is read word by word.*

**Exercise 3** *Modify the implementation of search in TSTs so it can process a star ('*') character in the search string. It can match any number of characters for words stored in the trie. This matching is done by adding all matching string to a queue that is an input argument to the generalized search function.*

*For example, after we insert BE, BED, and BACCALAUREATE, the string* "BE*" *matches the first two words, and* "*A*" *matches the only the third, in three different ways. The search string* "*" *should match the entire set of words stored in the trie and produce them in alphabetical order. You should decide if the different ways to match a search string should show up multiple times in the result queue or just one.*

**Exercise 4** *Modify TSTs to use a special non-alphabetic character (either period '.' or the nil character '\0'), which is shown as a small filled circle in the diagram, to represent the end of the word.*
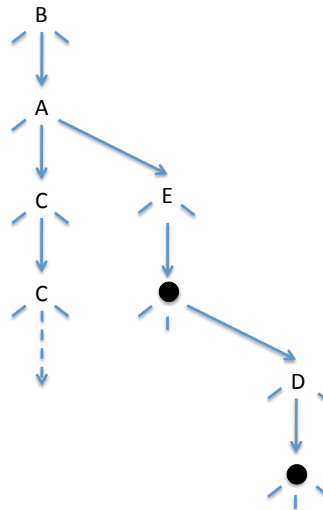


**Exercise 5** *Using this modified TST implementation from the question above, allow repeated searching of prefixes with the following interface:*

```
/* Prefix search interface */
typedef struct trie_node tnode;
```

```
tnode *trie_root(trie TR);
tnode *tnode_lookup_prefix(tnode *T, char *s); /* i < strlen(s) */
elem tnode_elem(tnode *T);
```

*The* `tnode` *pointer returned by* `trie_root` *or* `tnode_lookup_prefix` *should be a pointer to the root of a TST, and* `tnode_elem` *will look for the the special character* `'.'` *or* `'\0'` *by following* `left` *and* `right` *pointers and return any data stored at that special node.*

**Exercise 6** *Consider other implementations of the interface above that allow repeated searching of prefixes.*

**Exercise 7** *Implement sets of 32-bit integers as binary tries, including functions for testing set membership, intersection, union, and set difference.*

**Exercise 8** *Compare your implementation if integer sets with an implementation as binary search trees (assuming random insertion to avoid the complexities of rebalancing). Which operations on which kinds of sets would be more efficient in one representation than the other?*