Lecture Notes on Tries

15-122: Principles of Imperative Computation
Thomas Cortina
Notes by Frank Pfenning

Lecture 24
April 19, 2011

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 4×4 or 5×5). 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.

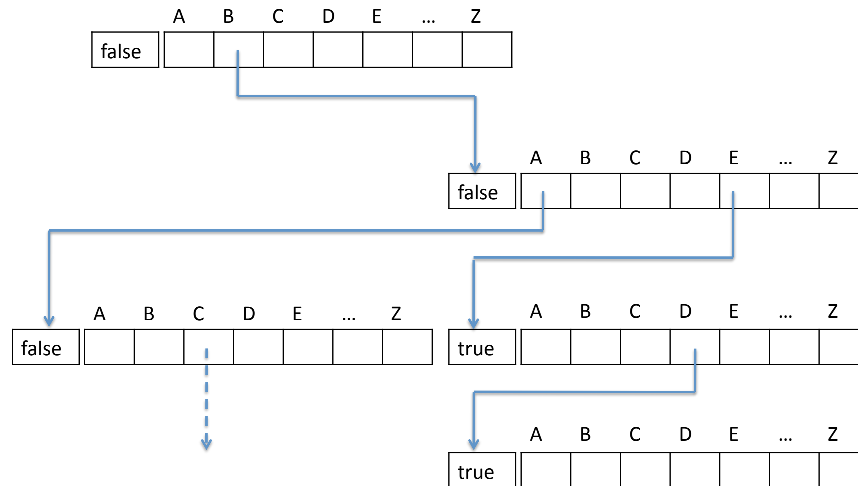
One simple possibility for implementing this game is to systematically search for potential words and then look them up in a dictionary, perhaps stored as a sorted word list, some kind of binary search tree, or a hash table. The problem is that there are too many potential words on the grid, so we want 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, BED, and BACCALAUREATE.



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 where 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 comparison, 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.

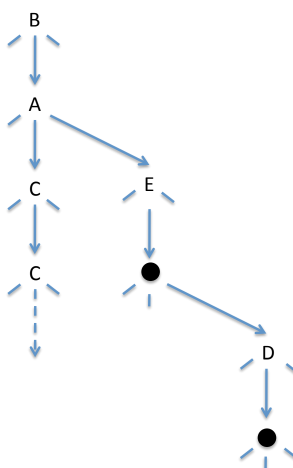
5 Ternary Search Tries

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 more suitable data structure is 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. Below is a diagram, again for the words BE, BED, and BACCALAUREATE. Instead of booleans, we use a special non-alphabetic character (period '.' in this case), which is shown as a small filled circle in the diagram. We have indicated null pointers by

short lines in the appropriate position in the tree.



In the ASCII table, the period is smaller than the letters from the alphabet, do D ends up to the right of '.'.

6 Asymptotic Complexity

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 does not change with the number of strings, only with its length. Even when the embedded trees are perfectly balanced, the constant factor decreases, but not the asymptotic complexity because $O(\log(26) * k) = O(k)$.

7 Specifying an Interface

In the example above and the code that we develop, we just represent a set of words rather than an associative array that stores arbitrary information for every key. Moreover, we commit to strings as keys. In that sense

our interface is not very abstract, but well-suited to our application. We also commit to words as consisting of only lowercase letters and provide a function `is_word` to verify whether a string is valid before entering it into the table or searching for it.

```
typedef struct tst* tst;
tst tst_new();
bool tst_search(tst TST, string s);
void tst_insert(tst TST, string s);
bool is_word(string s);
```

The string checking is simple.

```
bool is_wordchar(char c) {
  return 'a' <= c && c <= 'z';
}

bool is_word(string s) {
  int len = string_length(s);
  int i;
  for (i = 0; i < len; i++)
    if (!is_wordchar(string_charat(s,i)))
      return false;
  return true;
}
```

8 Checking Invariants

The declarations of the types is completely straightforward.

```
typedef struct trie* trie;
struct trie {
  char c;      /* discriminating character */
  trie left;
  trie middle;
  trie right;
};

struct tst {
  trie root;
};
```

To check that a trie is valid we use two mutually recursive functions. One checks the order invariant for the binary search trees embedded in the trie, the other checks the correctness of the subtries. For mutually recursive functions we need forward declare the function which comes textually second in the file so that type checking by the compiler can be done in order.

```
bool is_trie_root(trie T);
bool is_trie(trie T, char lower, char upper) {
  if (T == NULL) return true;
  //@assert T != NULL;
  if (!(is_wordchar(T->c) || T->c == '.')) return false;
  if (!(lower < T->c && T->c < upper)) return false;
  return
    is_trie(T->left, lower, T->c)
    && (   (T->c == '.' && T->middle == NULL)
        || (T->c != '.' && is_trie_root(T->middle)) )
    && is_trie(T->right, T->c, upper);
}

bool is_trie_root(trie T) {
  return is_trie(T, ' ', '~');
}
```

In the last function we use a space (`' '`) as a character whose ASCII code is small than all letters, and a tilde (`'~'`) as a character whose ASCII code is greater than all letters, essentially functioning as $-\infty$ and $+\infty$ for checking the intervals of a binary search tree with letters as keys.

One small refinement applied in the code above is the test that when we encounter the period `'.'` there cannot be any middle subtree. This is because a period cannot appear in the middle of a word. The presence of the period itself signals that the prefix that led us down to the node is in the tree.

9 Implementing Search on TSTs

Implementing search is just a direct combination of searching through a trie and searching through binary search tree. We pass a trie T, a string s and an index i which should either be a valid string index or be equal to the length of the string. If it is equal, we silently consider the character to be the word-ending period.

```
bool trie_search(trie T, string s, int i)
//@requires is_trie_root(T);
//@requires is_word(s);
//@requires 0 <= i && i <= string_length(s);
{ char si = (i == string_length(s)) ? '.' : string_charat(s, i);
  assert(is_wordchar(si) || si == '.', "illegal character in string");
  ...
}
```

If the tree is null, the word is not stored in the trie and we return `false`. On the other hand, if we are at the end of the string (`si = '.'`) and the character stored at the node is also `'.'`, then the word is in the trie and we return `true`. Otherwise, we continue to search in the left, middle, or right subtree as appropriate.

```
  if (T == NULL) return false;
  if (si == '.' && T->c == '.') return true;
  if (si < T->c) return trie_search(T->left, s, i);
  else if (si > T->c) return trie_search(T->right, s, i);
  else //@assert(si == T->c);
    return trie_search(T->middle, s, i+1);
```

Important for the last case: if the string character si is equal to the character stored at the node, then we look for the *remainder* of the word in the middle subtrie. This is implemented by passing $i + 1$ to the subtrie.

10 Implementing Insertion

Insertion follows the same structure as search, which is typical for the kind of data structure we have been considering in the last few weeks. If the tree to insert into is null, we create a new node with the character of the string we are currently considering (the ith) and null children and then continue with the insertion algorithm.

```
trie trie_insert(trie T, string s, int i)
//@requires is_trie_root(T);
//@requires is_word(s);
//@requires 0 <= i && i <= string_length(s);
{ char si = (i == string_length(s)) ? '.' : string_charat(s, i);
  assert(is_wordchar(si) || si == '.', "illegal character in string");
  if (T == NULL) {
```

```
    T = alloc(struct trie);
    T->c = si; T->left = NULL; T->right = NULL;
    T->middle = NULL;
  }
  ...
}
```

As usual with recursive algorithms, we return the the trie after insertion to handle the null case gracefully, but we operate imperatively on the subtries.

```
  if (si == '.' && T->c == '.') return T;
  if (si < T->c) T->left = trie_insert(T->left, s, i);
  else if (si > T->c) T->right = trie_insert(T->right, s, i);
  else T->middle = trie_insert(T->middle, s, i+1);
  return T;
```

At the top level we just insert into the root, with an initial index of 0. At this (non-recursive) level, insertion is done purely by modifying the data structure.

```
void tst_insert(tst TST, string s)
//@requires is_tst(TST);
//@requires is_word(s);
{
  TST->root = trie_insert(TST->root, s, 0);
  return;
}
```

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.*