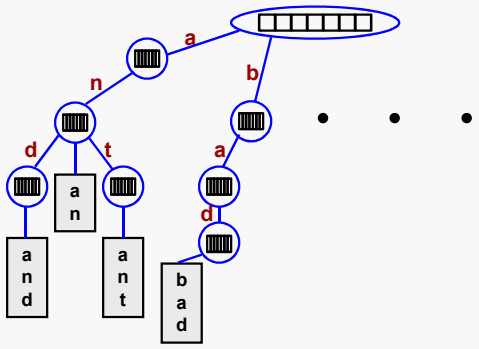
Definition and Example

trie a tree whose structure is determined by an equal subdivision of the range of key values; e.g., PR quadtree

a tree for storing strings in which there is one node for every common prefix; the strings are (perhaps) stored in extra leaf nodes; sometimes called an *alphabet trie*



Logical Properties

Internal nodes exist only as "guides".

An internal node does not store data, but its position in the tree identifies the key valuethat is associated with it.

The branching factor is determined by the alphabet from which the strings are formed.

The depth of the trie depends on the lengths of the strings that it stores, not the number of strings that it stores.

If the strings are uniformly distributed then the trie will be well-balanced.

Performance Properties

Looking up a key is generally faster than it would be in a BST storing full strings.

Finding a key of length m is clearly O(m), regardless of how many strings are in the trie.

Tries can require less space than a BST of strings, since duplicate prefixes are not stored multiple times.

Implementation

The primary issue would seem to be the design of the internal nodes in the trie, and the primary issue there would seem to be how to organize the pointers. For example:

- Array of pointers whose dimension equals the size of the alphabet
- Linked list of pointers

Either choice involves compromises…

Array of Pointers

Using an array of pointers is straightforward if the alphabet can be viewed as a contiguous sequence of characters represented by ASCII codes.

For example, if the alphabet is the characters 'a' through 'z' then we may rely on:

```
Node* Array[26] = {0};
char ch = 'g';          // pick a character
Array[ch - 'a'] = new Node(. . .);
       // 'g' - 'a' == 103 - 97 == 6
```

However, if the alphabet is very large, but the number of actual character pairings is relatively small, the first choice will store many NULL pointers, wasting space in many nodes.

List of Pointers

Using a linked list of pointers may save space, since the list will only contain entries for actual children.

On the other hand, there's no automatic way to know what character corresponds to a child pointer, so we must explicitly store the corresponding characters.

And, of course, the list pointers themselves are pure storage overhead.

Issues and Optimizations

One issue is "how do we recognize the end of a string that is a prefix of another string"?

The answer is usually that we will store a special flag in each node indicating whether it is the final character in a valid string.

Storing the full strings in special leaf nodes is unnecessary, since the string can be assembled, character by character, as the tree is searched.

On the other hand, we could also use leaf nodes to store unshared suffixes of words.

For example, consider `or` and `orange`. We might store these in the tree as

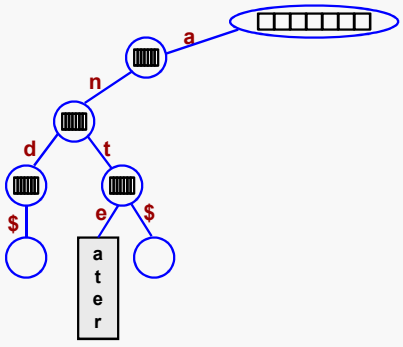
`o-r` and `o-r-ange`

(at least until we add a word like `orangutang`.

Since this eliminates several nodes and the associated pointers, the space savings may be considerable.

Variants

Do not decompose unshared suffixes, but store them in leaf nodes:



Store the strings in reverse order… if we condense unshared data as described above, this would take advantage of unshared prefixes in the set of strings.