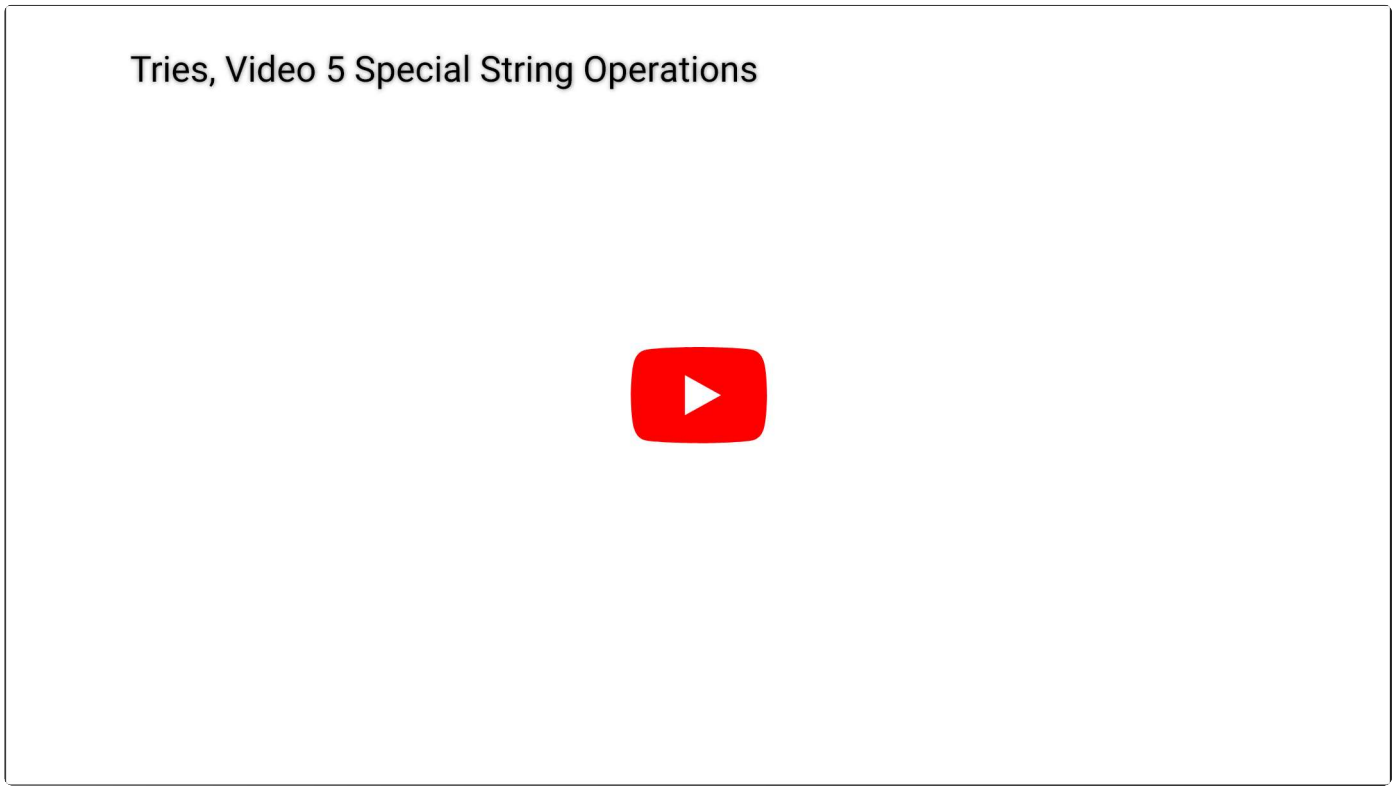
26.3 Trie String Operations

Trie, Trie again.

Tries give us the ability to have constant time lookup and insertion, but they do not always perform better than BSTs and Hash Tables. For any string, we have to traverse through every character, whereas in BSTs and Hash Tables we have access to the entire string immediately. However, Tries are *much* more useful in the specific application of String Operations.



The main appeal of tries is the ability to efficiently support specific string operations like *prefix matching*. You can imagine why tries make this extremely efficient! Say we were trying to find the longestPrefixOf. Just take the word you're looking for, compare each character with characters in your trie until you can go no longer. Similarly, if we wanted keyWithPrefix, we can traverse to the end of the prefix and return all remaining keys in the Trie.

Let's attempt to define a method `collect` which returns all of the keys in a Trie. The pseudocode will be as follows:

```
collect():
    Create an empty list of results x
    For character c in root.next.keys():
        Call colHelp(c, x, root.next.get(c))
    Return x

colHelp(String s, List<String> x, Node n):
    if n.isKey:
        x.add(s)
    For character c in n.next.keys():
        Call colHelp(s + c, x, n.next.get(c))
```

We first initialize our values inside of the parent function, and then create a recursive helper function to hold more parameters throughout the recursive calls. We only add the current string if it is a key, otherwise we concatenate the character to the string/path we are currently traversing and call the helper on the next child.

Now we can try writing the method `keysWithPrefix` which returns all keys that contain the prefix passed in as an argument. We will borrow heavily from the collect method above.

```
keysWithPrefix(String s):
    Find the end of the prefix, alpha
    Create an empty list x
    For character in alpha.next.keys():
        Call colHelp("sa" + c, x, alpha.next.get(c))
    Return x
```

Autocomplete

When you type into any search browser, for example Google, there are always suggestions of what you are about to type. This is extremely helpful and convenient. Say we were searching "How are you doing", if we just type in "how are" into google, we will see that it suggests this exact query.

One way to achieve this is using a Trie! We will build a map from strings to values.

- Values will represent how important Google thinks that string is (Probably frequency)
-

- Store billions of strings efficiently since they share nodes, less wasteful duplicates
- When a user types a query, we can call the method `keysWithPrefix(x)` and return the 10 strings with the highest value

One major flaw with this system is if the user types in short length strings. You can imagine that the number of keys with the prefix of the input is in the millions when in reality we only want 10. A possible solution to this issue is to store the best value of a substring in each node. We can then consider children in the order of the best value.

Another optimization is to merge nodes that are redundant. This would give us a "radix trie", which holds characters as well as strings in each node. We won't discuss this in depth.

Summary

Knowing the types of data that you are storing can give you great power in creating efficient data structures. Specifically for implementing Maps and Sets, if we know that all keys will be Strings, we can use a Trie:

- Tries theoretically have better performances for searching and insertion than hash tables or balanced search trees
- There are more implementations for how to store the children of every node of the trie, specifically three. These three are all fine, but hash table is the most natural
  - *DataIndexedCharMap* (Con: excessive use of space, Pro: speed efficient)
  - *Bushy BST* (Con: slower child search, Pro: space efficient)
  - *Hash Table* (Con: higher cost per link, Pro: space efficient)
- Tries may not actually be faster in practice, but they support special string operations that other implementations don't
  - `longestPrefixOf` and `keysWithPrefix` are easily implemented since the trie is stored character by character
  - `keysWithPrefix` allows for algorithms like autocomplete to exist, which can be optimized through use of a priority queue.

Name	key type	get(x)	
Balanced BST	comparable	$\Theta(logN)$	$\Theta(logN)$
RSC Hash Table	hashable	$\Theta(1)^{\dagger}$	$\Theta(1)^{*\dagger}$

Name	key type	get(x)	
Data Indexed Array	chars	$\Theta(1)$	$\Theta(1)$
Tries (BST, HT, DICM)	strings	$\Theta(1)$	$\Theta(1)$

*: on average, †: items are evenly spread

Previous
26.2 Trie Implementation

Next
26.4 Summary

Last updated 1 year ago