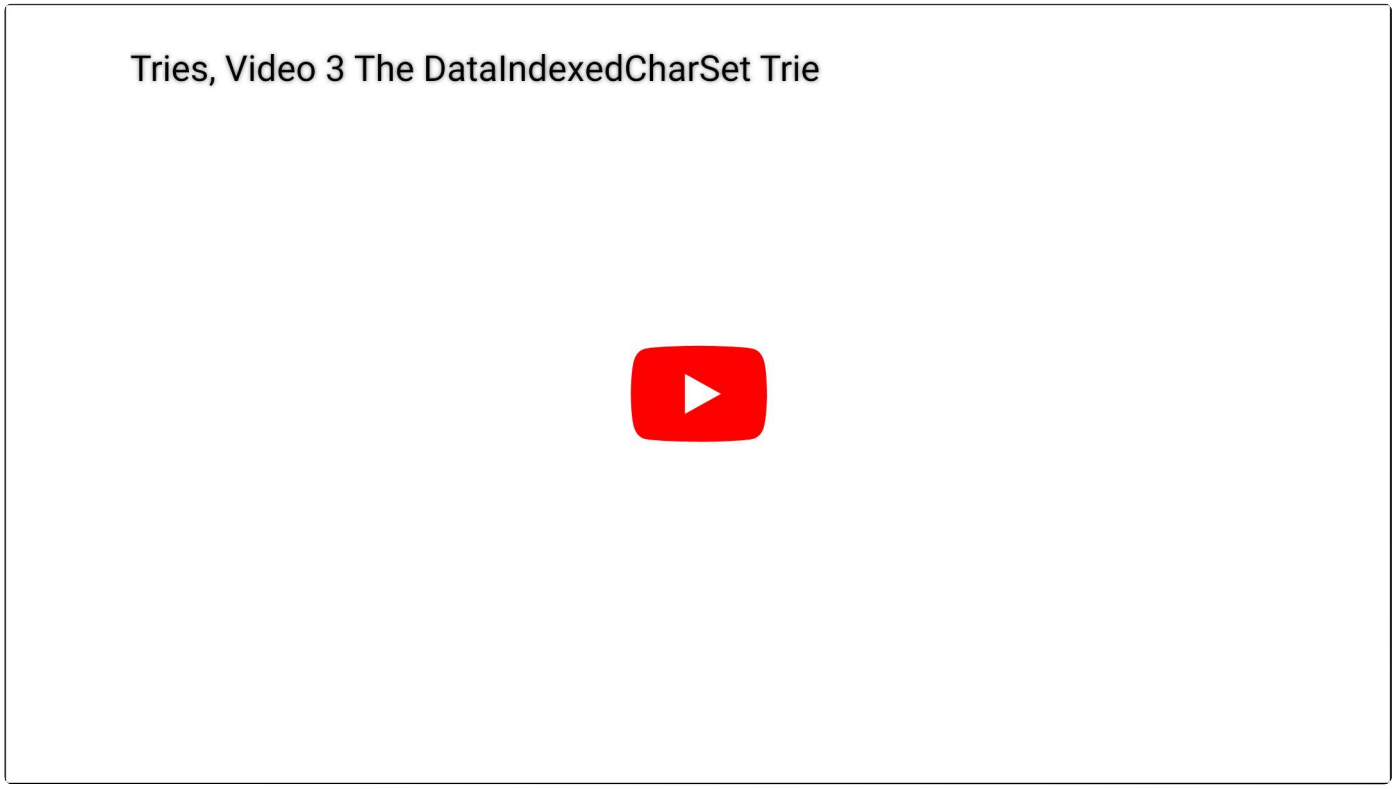
26.2 Trie Implementation

Giving it the old college Trie.



Now, we will walk through an implementation of a Trie. We'll take a first approach with the idea that each node stores a letter, its children, and a color. Since we know each node key is a character, we can use our `DataIndexedCharMap` class we defined earlier to map to all of a nodes' children. Remember that each node can have at most the number of possible characters as its number of children.

```
public class TrieSet {
   private static final int R = 128; // ASCII
   private Node root;    // root of trie

   private static class Node {
      private char ch;
      private boolean isKey;
      private DataIndexedCharMap next;

      private Node(char c, boolean blue, int R) {
         ch = c;
         isKey = blue;
         next = new DataIndexedCharMap<Node>(R);
      }
   }
}
```

Note that for any given node, the DataIndexedCharMap object for that node will have mostly null values if nodes in our tree have relatively few children. For a node with only one child, we will have 128 links with 127 equal to null and 1 being used. This means that we are wasting a lot of excess space! We will explore alternative representations further on.

From this, we can make another important observation: each link corresponds to a character if and only if that character **exists**. Therefore, we can remove the Node's character variable and instead base the value of the character from its position in the parent `DataIndexedCharMap`.

```
public class TrieSet {
   private static final int R = 128; // ASCII
   private Node root;    // root of trie

   private static class Node {
      // no more 'ch' instance variable
      private boolean isKey;
      private DataIndexedCharMap next;

      private Node(boolean blue, int R) {
         isKey = blue;
         next = new DataIndexedCharMap<Node>(R);
      }
   }
}
```

Performance

Given a Trie with N keys the runtime for our Map/Set operations are as follows:

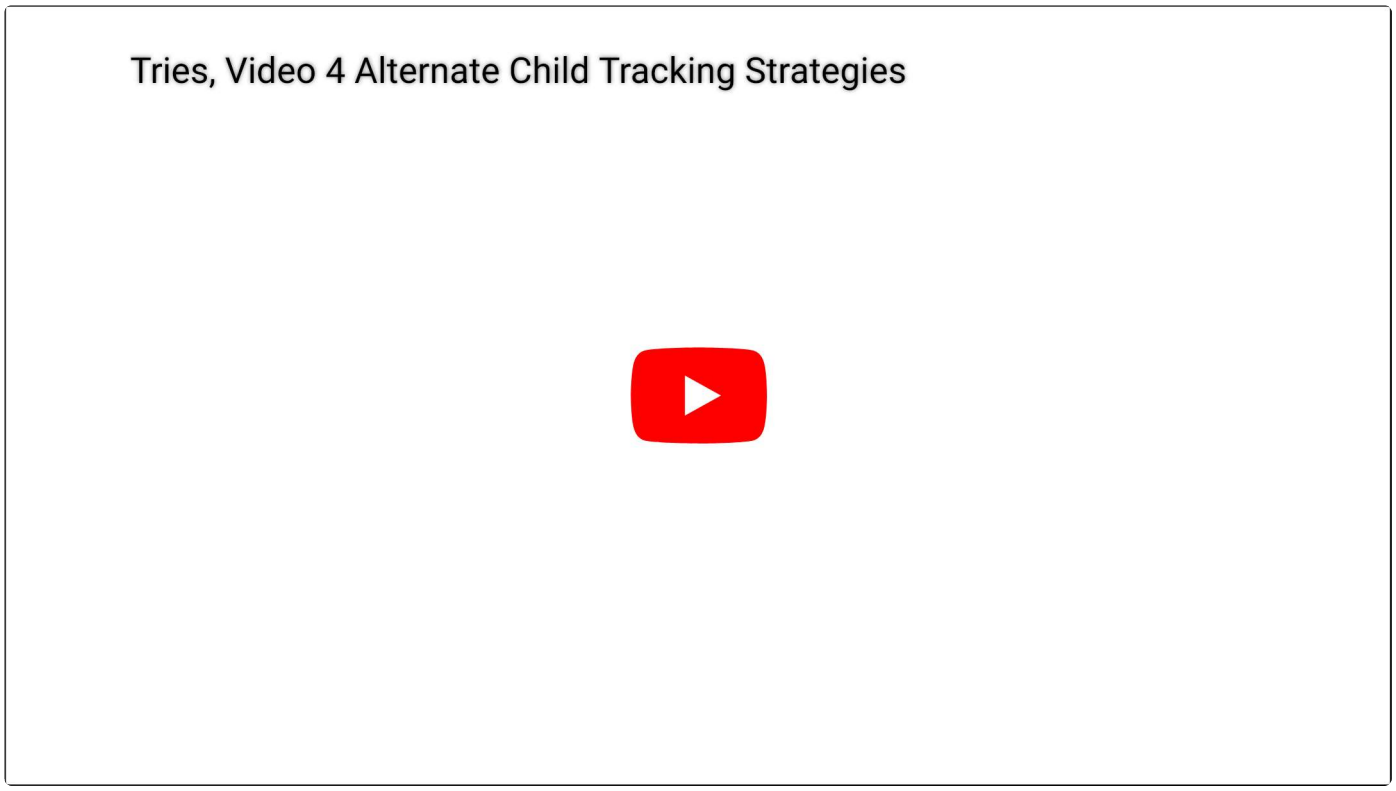
- `add` : $\Theta(1)$
- `contains` : $\Theta(1)$

Why is this the case? It doesn't matter how many items we have in our Trie, the runtime will always be *independent* of the number of keys. We only traverse the length of one key in the worst case ever, which is unrelated to the number of keys in the Trie. Therefore, let's analyze the runtime with a more appropriate measurement: L, the length of the key we are searching for:

- `add` : $\Theta(L)$
- `contains` : $\Theta(L)$

We have achieved constant runtime without having to worry about amortized resizing times or having an even spreading of keys! Our only issue is that as we mentioned above, our current design is extremely wasteful memory-wise since each node contains an array for every single character even if that character doesn't exist.

Improvement: Child Tracking



To address the issue of wasted space, let us explore two possible solutions:

- *Alternate Idea #1*: Hash-Table based Trie. This won't create an array of 128 spots but instead initialize the default value and resize the array only when necessary with the load factor.
- *Alternate Idea #2*: BST based Trie. Again this will only create children pointers when necessary, and we will store the children in the BST. We will have to worry about the runtime for searching in this BST, but this is not a bad approach.

When we implement a Trie, we have to pick a map to our children. A Map is an ADT, so we must also choose the underlying implementation for the map. What does this reiterate to us? There is an **abstraction** barrier between the implementations and the ADT that we are trying to create. This abstraction barrier allows us to take advantage of what each implementation has to offer when we try to meet the ADT behavior. Let's consider each advantage:

- DataIndexedCharMap
  - Space: 128 links per node
  - Runtime: $\Theta(1)$
- BST
  - Space: C links per node, where C is the number of children

    - Runtime: O(log$R$), where R is the size of the alphabet
- Hash Table
    - Space: C links per node, where C is the number of children
    - Runtime: O(R), where R is the size of the alphabet

Note: Cost per link is higher in BST and Hash Tables; R is a fixed number (this means we can think of the runtimes as constant)

We can takeaway a couple of things. There is a slight memory and efficiency trade off (with BST/Hash Tables vs. DataIndexedCharMap). The runtimes for Trie operations are still constant without any caveats. Tries will especially thrive with some special operations.

Previous
26.1 Introduction to Tries

Next
26.3 Trie String Operations

Last updated 1 year ago