Let’s take a Look Back...

• The first ADT we looked at -- the List ADT -- was very general. It could be applied to any type that had a test for equality.

• The performance of lists was \(O(N)\) for \(\text{FIND, INSERT}\) (in order) and \(\text{DELETE}\).

• We saw an improvement in performance with Binary Search Trees (\(\text{FIND, INSERT and DELETE}\) were \(O(\log N)\)), but we added the condition that the type had to support not just a test for equality, but also a test for less than and greater than.

And a Look Forward...

• Next, we’re going to see a new ADT that has even better performance, but we add more restrictions to the type. The restrictions are reasonable, but it does mean that this ADT doesn’t work for every type.

• We will see this idea again throughout the class. The less generic an ADT, the better the performance (in general).
Tries

- A very different data structure - looks like a tree, but behaves quite differently!

- Supports
  - \( v = \text{FIND} \) (key)
  - \( \text{INSERT} \) (key, value)
  - \( \text{DELETE} \) (key)

- All three operations can be performed in \( O(L) \) time where \( L \) is the length of the key.

- The time required by these operations \( O(L) \), regardless of the number of keys in the trie! (if \( L \) is bounded, they’re all \( O(1) \)!)
How is this accomplished?

• Keys must be strings.

A string is a *finite* sequence of **characters** drawn from a *finite alphabet*.

An alphabet is a set, along with an operator that can test whether two elements in the set are identical. (there is often an ordering placed on the elements of the set, i.e. alphabetical order, but we will not assume one)

• Each character in a key is represented by an edge in a tree.

• Nodes that represent the end of a key are marked with an **END-OF-KEY** marker.

• To search for a key, start at the root and follow the edge corresponding to each character.

  • If no such edge, fail.
  • If search ends at **END-OF-KEY** node, succeed.
What words (keys) are represented by this trie?

- Note - it is very tempting to "collapse" all of the "at"s together, but this won’t work. A more sophisticated representation can allow this, but not a trie. We will study such representations later...
Representing Tries:

de la Briandais Trees and Patricias

• Tries can be represented by M-ary trees (where M is the size of the alphabet). This can potentially waste a gigantic amount of memory, however, as we have seen:

\[ M + (M - 1)(N - 1) \] unused edges

• How can we do better?

• Tries implemented as lists of lists - like the linked list representation we saw earlier. These are called de la Briandais trees.

• If many of the nodes in a particular path have only one child, we can “compress” the path to a single edge by allowing each edge to represent a string of characters, instead of a single character. This is the approach used by the patricia data structure, which is described in the course book.
Algorithms

• Print all of the keys in a trie, in order.
  • What is the big-O of such an algorithm? What is the big-Omega? (how can we even tackle such a problem?)
• Find all the keys in a trie that differ by one character from a given string.