Ternary Search Trees (TSTs)

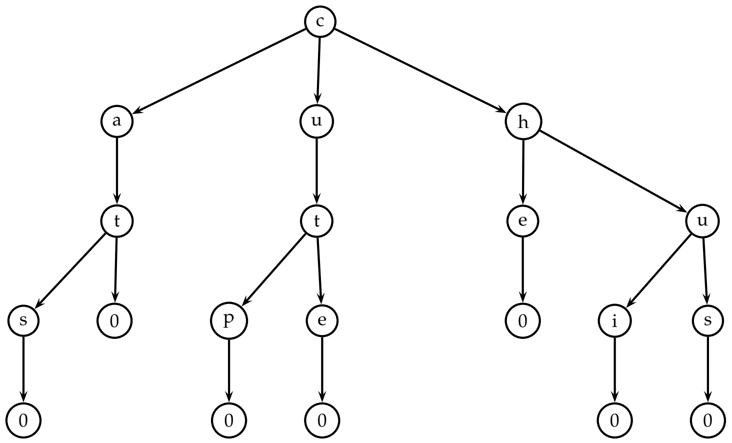
Salvador Roura

TSTs can be easily used to implement sets or maps when keys are so long that treating them as atomic elements would be too expensive. For instance, if we store strings that codify genetic information, then full key-comparison of those strings may have a prohibitive cost.

An explanation of TSTs, given by their inventors, can be found here:

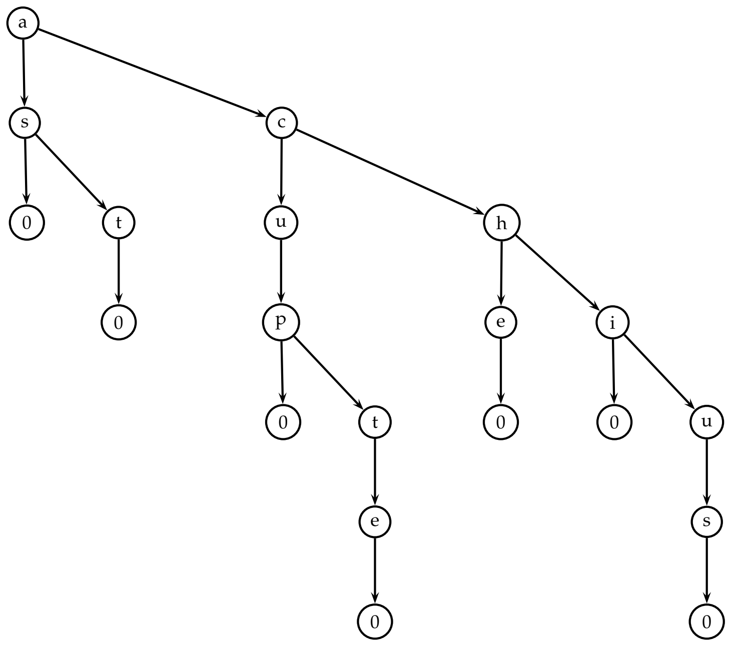
https://www.cs.upc.edu/ ps/downloads/tst/tst.html.

I think that the mentioned paper is clear enough. Therefore, I will only add a few things. To begin with, a couple of examples of TSTs, using the same keys, and inserted in the same order, as in the wikipedia: `"cute"`, `"cup"`, `"at"`, `"as"`, `"he"`, `"us"` and `"i"`.



Note that the 0's in the picture are not `'0'`, but the characters with code 0 (which is unprintable). Every path from the root to each 0 gives us a stored word, if we only collect the characters at the top of the pointers that we follow down vertically. For instance, to reach the third leftmost 0, corresponding to "cup", we pick 'c', we pick 'u', we discard 't', we pick 'p', we arrive at 0 and we stop.

The internal structure of a TST greatly depends on the insertion order of its keys. This is the result if we insert the words in alphabetical order: "as", "at", "cup", "cute", "he", "i" and "us":



To finish this brief summary of TSTs, below you have C++ code for their two most basic operations: searching for a word, and inserting a word. I have used a plain 0 for null pointers, and '!' (ASCII 33, smaller than any printable character except spaces) instead of 0 for terminating characters. This way you may debug this code more easily.

```
// a TST "is" the pointer to its root
typedef struct Node * TST;

struct Node {
  char c;        // current char
  TST l, m, r;  // left, middle and right pointers
};
```

```
// search for s[i..] in the (sub)TST t
bool search(TST t, const string& s, int i) {
  if (t == 0) return false;
  if (s[i] < t→c) return search(t→l, s, i);
  if (s[i] > t→c) return search(t→r, s, i);
  if (s[i] == '!') return true;
  return search(t→m, s, i + 1);
}
```

```
// insert s[i..] in the (sub)TST t
TST insert(TST t, const string& s, int i) {
  if (t == 0) return new Node { s[i], 0, (s[i] == '!' ? 0 : insert (0, s, i + 1)), 0 };
  if (s[i] < t→c) t→l = insert(t→l, s, i);
  else if (s[i] > t→c) t→r = insert(t→r, s, i);
  else t→m = insert(t→m, s, i + 1);
  return t;
}
```

As usual, those recursive procedures have iterative versions that may be a bit faster, but also slightly more complicated.