The constant-time array initialization trick

Goals:

- an array $A$ indexed by $0, \ldots, m - 1$
- constant time access
- space $O(m)$
- \textit{constant time initialization} (to some default value)
Idea: main array is $A$, two auxiliary arrays $When$ and $Which$, and a time counter $t$

If $A[i]$ is first assigned to at time $t$:

$$When[i] = t, \quad Which[t] = i$$

$Init()$: $t \leftarrow 0$

$Valid(i)$:

$$t' \leftarrow When[i]$$

return $0 \leq t' < t$ and $Which[t'] = i$

$Access(i)$:

if $Valid(i)$ then return $A[i]$

else return default value

$Assign(i, v)$:

if not $Valid(i)$ then

$$When[i] \leftarrow t, \quad Which[t] \leftarrow i, \quad t \leftarrow t + 1$$

$$A[i] \leftarrow v$$
Search tries: dictionary for strings

dictionary for strings over finite alphabet $\Delta$

assume $\Delta = \{0, \ldots, m - 1\}$

maintain an $m$-ary tree, where the items in the dictionary determine the paths in the tree
Example

Assume $m = 2$

Data:

\[
\begin{array}{c}
00 \\
01111 \\
011110 \\
0111111 \\
11101 \\
\end{array}
\]
Analysis

Assume

- each node is represented as an array of $m$ pointers
- $n$ items in the dictionary
- $N = \text{sum of lengths of all items}$

Time for lookup: $O(\text{length of item})$

Space: $O(Nm)$

- reduce to $O(N + nm)$ by compressing “chains”
- or even to $O(N + n)$ by using linked lists — drawback: slows down lookups
Example: chain compression

Data:

00
01111
011110
0111111
11101
Why chain compression works

**Fact:** if $T$ is a tree with $n$ leaves, and every internal node has degree $> 1$, then $T$ has at most $n - 1$ internal nodes

Time for insertion: $O(\text{length of item } + m)$

- $O(m)$ time needed to initialize a single array of $m$ pointers
- this assumes chain compression
- we can even get rid of the $O(m)$ term, using the “constant time array initialization trick”

Time for deletion: similar to insertion
2-3 trees: a dictionary for general data

Assume data items are totally ordered (<, >, =)

Assume $n$ items in the dictionary

Structure: a tree

- Data stored only at leaves (no duplicates)
- All leaves at the same level, in sorted order
- Each internal node:
  - has either 2 or 3 children
  - has a “guide”: the maximum data item in its subtree

Height of tree is $O(\log n)$
Example
Search(x): use guides

Insert(x): Search for x, and if it should belong under p:

add x as a child of p (if not already present)

if p now has 4 children:

- split p into two two nodes, \( p_1 \) and \( p_2 \), each with two children
- process p’s parent in the same way
- Special case: no parent — create new root, increasing height of tree by 1

Also need to update “guides” — easy

Time = \( O(\text{height}) = O(\log n) \)
Case when $p$ ends up with 4 children

\[ p \]
\[ w \quad x \quad y \quad z \]

\[ p \]
\[ w \quad y \quad z \]

\[ p_1 \]
\[ w \quad x \]

\[ p_2 \]
\[ y \quad z \]
Delete($x$): Search for $x$, and if found under $p$:

- remove $x$

  if $p$ now only has one child:

  - if $p$ is the root: delete $p$ (height decreases by 1)
  - if one of $p$’s siblings has 3 children: borrow one
  - if none of $p$’s siblings has 3 children:
    - one sibling $q$ must have 2 children
    - give $p$’s only child to $q$
    - delete $p$
    - process $p$’s parent
Easy case: borrow from sibling
Harder case: give away only child
2-3 trees: summary

Assume $n$ items in dictionary

Running time for lookup, insert, delete:

- $O(\log n)$ comparisons, plus $O(\log n)$ overhead

Space: $O(n)$ pointers
Dictionaries for strings: a comparison

hash tables, search tries, or balanced trees (e.g., 2-3 trees)?

Assume \( n \) strings of length \( t \) over an \( m \) letter alphabet

Time per lookup:

- tries and hash tables: \( O(t) \)

which is faster depends on the relative costs of memory access (tries will jump through \( t \) pointers) and hash function evaluation

tries may be faster for “misses”
Time per lookup (cont’d):

- balanced trees: $O(t \log n) — O(\log n)$ comparisons, each takes time $O(t)$

Space:

- hash tables and balanced trees very space efficient
- tries can be real space hogs

Support for other operations:

- tries support fast prefix matching
- balanced trees support fast in-order traversal (and other things)
- hash tables: nothing
Ternary Search Trees
the best of all possible worlds?

Reference: Sedgewick & Bentley
http://www.ddj.com/184410528

- Each internal node has 3 children and a “guide” that consists of a single letter in the alphabet
- To look for a string, compare current letter of string to guide of current node
  - update current node: branch left/down/right according to <, =, >
  - update current letter: advance 1 pos if =
3. The Algorithms

Just as Quicksort is isomorphic to binary search trees, so (most-significant-digit) radix sorting to the tree in Figure 1 was constructed by this criterion. Bentley and Saxe present the structure as a solution to a

This computing the value low Hoare's [9] solution finally as sorting

the significant partitioning Ternary each component gives

vector in-place, this pseudocode provides

ternary structures in-place, and references by

a partitioning to a median input

s,\ s, I=, II=, \ldots\) value for

111).<br /><br />

These elements are identical (most-significant-digit) to this algorithm. The binary search tree in Figure 1 was constructed by this criterion. Bentley and Saxe present the structure as a solution to a

as at be by he in is it of on or to

**Figure 2. A ternary search tree for 12 two-letter words**
Running time for a lookup: $O(t + \log n)$

- Assumes length of string is $t$, dictionary contains $n$ items, and that tree is well balanced

- Idea: each iteration of lookup step either
  - decreases length of string by 1, or
  - cuts number of items in half

Space: $O(n)$ (assuming path compression)

Support for other operations:

- prefix matching
- in-order traversal
- others ...