Another hash application: fast pattern matching

Problem: Given strings $a = a_1 \cdots a_n$, and $b = b_1 \cdots b_t$, test if $b$ is a substring of $a$.

Naive algorithm: time $O(nt)$

Faster algorithms: time $O(n)$ (assume $t \leq n$)

- A simple, randomized algorithm (Karp, Rabin)
- A trickier deterministic algorithm (Knuth, Morris, Pratt)
The Karp/Rabin Algorithm (a variant)

Let \( \{h_k\}_{k \in \mathcal{K}} \) be an \( \varepsilon \)-universal family of hash functions on strings of length \( t \)

Algorithm:

choose a random key \( k \)
\( s \leftarrow h_k(b) \)
for \( i \leftarrow 1 \) to \( n - t + 1 \) do
\( s_i \leftarrow h_k(a_i \cdots a_{i+t-1}) \)
if \( s = s_i \) then
  if \( b = a_i \cdots a_{i+t-1} \) then
    return \text{match} \\
return \text{no match}
Running time analysis: two factors

- time to compute hash function
- expected time spent processing “false positives”: $O(\varepsilon \cdot n \cdot t)$

Use “polynomial evaluation” hash:

- view $a_i$’s, $b_j$’s, $k$ as elements of $\mathbb{Z}_p$, where $p$ is prime
- $h_k(a_1 \cdots a_t) = a_1 k^{t-1} + \cdots + a_t$
- $\varepsilon = t/p$
- time to evaluate each $h_k$: $O(t)$ naively, but we can do better
Computing a “Rolling Hash”

\[
\begin{align*}
& a_1 k^{t-1} + a_2 k^{t-1} + \cdots + a_t \\
- & a_1 k^{t-1} \\
\hline
& a_2 k^{t-1} + \cdots + a_t \\
& \times k \\
\hline
& a_2 k^t + \cdots + a_t k \\
& + a_{t+1} \\
\hline
& a_2 k^t + \cdots + a_t k + a_{t+1}
\end{align*}
\]
Karp/Rabin: conclusions

Assume $p$ is near machine word size (e.g., $2^{32}$)

Assume arithmetic in $\mathbb{Z}_p$ takes time $O(1)$

Time to compute hashes: $O(n)$

Expected time to process false positives: $O(nt^2/p)$, which is $O(n)$ for “reasonable” $t$ (e.g., $t < 2^{16}$)
The constant-time array initialization trick

Goals:

- an array $A$ indexed by $0, \ldots, m - 1$
- constant time access
- space $O(m)$
- 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 \(t\):

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

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

\(Valid(i)\): return

\[
0 \leq When[i] < t \text{ and } Which[When[i]] = i
\]

\(Access(i)\):

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:

```
00
01111
011110
0111111
11101
```
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
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, p₁ and p₂, 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

$p$

$w \ x \ y \ z$

$p$

$w \ y \ z$

$p_1$

$w \ x$

$p_2$

$y \ z$
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

\[
q \quad p \\
\text{u} \quad v \quad w \\
\text{u} \quad v \\
\text{w} \quad y
\]
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 =
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 ...