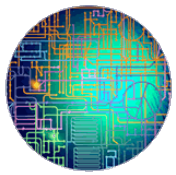


Indexed Search

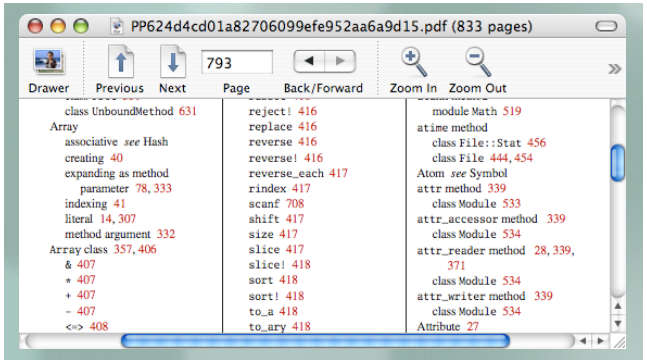
Data organization
  search tree
  hash table
Examples
Google

Overview

- In the last lecture we saw how ***preprocessing a pattern*** can make a search more efficient
  - build an FSA or a set of tables (Boyer-Moore)
  - can save FSA or tables, apply to any text
- Today's topic shows how to ***preprocess the text*** so a wide variety of patterns can be searched efficiently
  - preprocessing creates an ***index*** that can be used to look up patterns
- Reading
  - Search Trees (NTO Ch. 11)
    - skim this section
  - Storage by Hashing (NTO Ch. 43)
    - the main topic for today

Analogy

- An index data structure is like an index in a book
  - given a word find where it occurs in the text



Data Structure

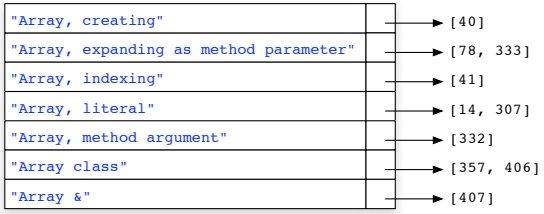
- A data structure that implements an index is an array of strings
  - the strings are called ***keys***
- Example
  - part of the index of the *Programming Ruby* book:

```
"Array, creating"
"Array, expanding as method parameter"
"Array, indexing"
"Array, literal"
"Array, method argument"
"Array class"
"Array &"
```

These keys are arbitrary strings, but some indices may restrict keys to single words

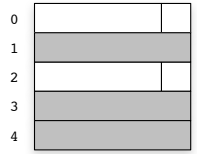
Data Structure (cont'd)

- The index usually associates some extra information with each key
  - in a book we expect to see a list of pages
  - in a search engine we want URLs of web sites
- An index contains ***pointers*** to this extra information
  - for today's talk we aren't concerned with these pointers, and will focus on the array contents



Terminology

- In this lecture, an array is a fixed-size collection of strings
  - we'll use the name M (for "memory") in algorithms
- An ***address*** identifies a location in the array
  - array addresses range from 0 to n-1
- In some cases an array may be only partially filled
  - in these cases we say it has ***empty*** entries
- Example
  - this index has room for 5 entries
  - M[0] and M[2] are used
  - M[1], M[3], and M[4] are empty



Linear Search

- The simplest way to use an index is to just scan from the beginning
  - this is known as ***linear search***
- Pseudo-code:
  - input: a string `s`
  - output: the address of the location of `s`, or `nil` if it's not found

```
i ← 0
while i < n
  if M[i] = S
    return i
  i ← i+1
return nil
```

- This algorithm is clearly $\mathcal{O}(n)$ -- when `s` is not in the index we have to scan all n entries

Binary Search

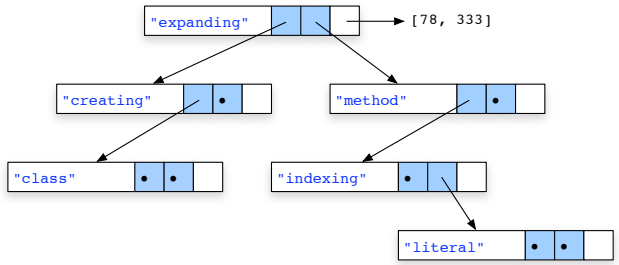
- If the items in the array are sorted, we can use a ***binary search***
- It's similar to how you would search a phone book or dictionary:
  - open the book to a page near the middle
  - if the item you are looking for is before the item at the top of the page, open a page half-way between the current page and the front of the book
  - if the item is after the last one on the page, open a page half way between the current page and the end of the book
- Basic idea: keep narrowing the boundaries of possible locations for the item
  - if the region is divided exactly in half each time it's a binary search

Efficiency of Binary Search

- When you're searching a book you don't always divide a region in equal sizes
  - if you're search for a word or name that starts with "C" you're likely to pick a page 1/5 or 1/4 of the way into the book
- So how efficient is a method that always divides by 2?
- If a list has n items, binary search will find S in at most $\log_2 n$ steps
  - example: with 1,000,000 entries binary search will do at most 20 comparisons
- To see why it is $\log_2 n$, consider the problem from the opposite direction:
  - start with a list of just one item
  - how many times do you have to double the number of items in the list to reach a size of n?
  - doubling x times leads to a size of 2^x
  - e.g. 2^{20} = 1,048,576

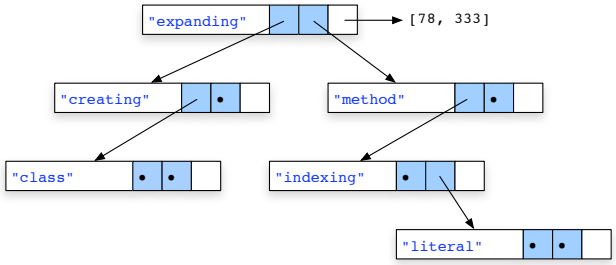
Binary Search Tree

- The binary search method works for items stored in an array
- Chapter 11 of *NTO* describes a related structure called a ***binary search tree***
- A tree is useful when new information is added over time



Binary Search Tree

- To insert a string S, compare it to the string R at the root of the tree
- If S < R move down the left subtree, otherwise move down the right subtree
- Insert S when you come to an empty subtree (marked by • in the diagram)



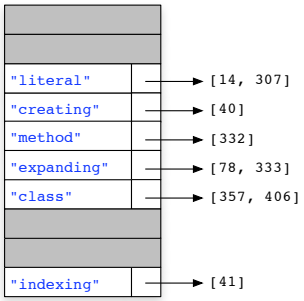
Hash Tables

- Binary search is a big improvement over linear search
  - $\mathcal{O}(\log_2 n)$ *vs.* $\mathcal{O}(n)$
  - 20 comparisons *vs.* 500,000 in an array of 1,000,000 items
- The binary search tree is also $\mathcal{O}(\log_2 n)$
- The next organization we'll look at is even more efficient
- A ***hash table*** takes $\mathcal{O}(1)$ steps to look up an item
  - $\mathcal{O}(1)$ means "constant time"
  - the algorithm takes the same amount of steps, no matter how big the array is
  - as we'll see the time actually depends on how full the array is

Hash Function

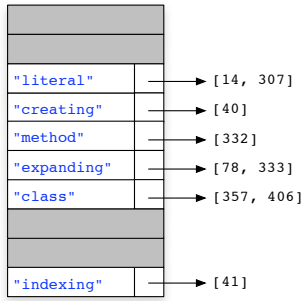
- The idea is to store S at an address computed by a ***hash function*** h
  - h is a function that maps strings to integers
  - the table has room for n items, so

    h: S → 0..n-1
- Example:
  - h(“literal”) = 2
  - h(“expanding”) = 5
- This table has room for 10 entries, but only 6 are filled



Simple Hash Function

- The function used to fill this table is to use the first letter of the string
  - “a” → 1
  - “b” → 2
  - ...
  - “i” → 9
  - “j” → 0
  - “k” → 1
  - ...
  - “s” → 9
  - “t” → 0
  - etc
- What happens when more than one string has the same address?
  - h(“creating”) = h(“class”)

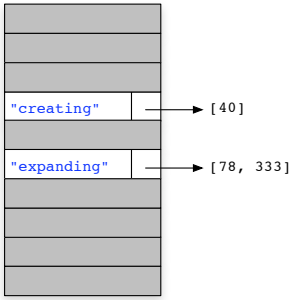


Collisions

- When two strings map to the same location it is called a ***collision***
- One simple method for dealing with collisions is to just start looking for the ***next open address***
- Let’s see how the example table was built
- First two strings:

  h(“creating”) = 3

  h(“expanding”) = 5
  - both go into open locations

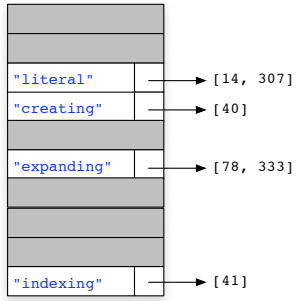


Collisions (cont’d)

- Second two strings:

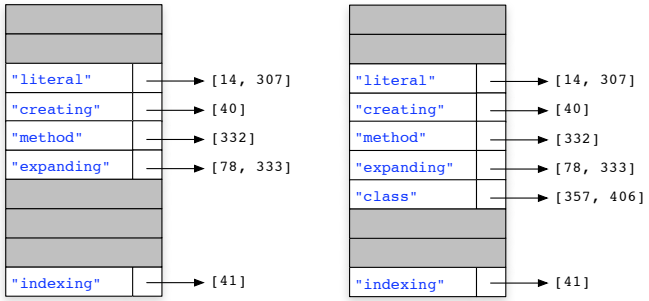
  h(“indexing”) = 9

  h(“literal”) = 2
  - these also go into open slots
- The next string (“methods”) wants to go in the place already occupied by “creating”
- The next open spot is below “creating”



Collisions (cont'd)

- The last string ("class") also wants to go where "creating" was placed
- Now we have to go all the way down to address 6 to find an open location



Statistics

- The example shows what happens when a table starts to fill up
  - more entries means fewer open spots
  - the probability of a collision grows
  - the number of steps to find a free location also grows
- When performance is important, one needs to make sure there will be sufficient empty locations
- If you know there will be m keys, choose a table size $n > m$
- But how much bigger?
- The "birthday paradox" suggests it might be a lot more than you think...

Birthday Paradox

- Suppose all birthdays are equally likely
- In a room with m people, what are the odds that two or more people have the same birthday?
  - intuition might tell you its around m / 365
  - that is the right percentage for the odds of matching a particular day
- But for *any two people* to have the same birthday:

n	odds of match
10	12%
20	41%
30	70%
50	97%
100	99.99996%

Search wikipedia for "birthday paradox"

Birthday Paradox (cont'd)

- The statistics behind the birthday paradox also apply to hash tables
  - birthdays: 365 choices
  - hash tables: n choices
- Using a calendar to find matching birthdays is the same as inserting new elements into a hash table
  - as you ask people their birthdays put a check mark next to a day on the calendar
  - that "cell" in the table is now full
  - the first match will correspond to a collision
- In practice one isn't concerned with avoiding all collisions
  - just make sure there aren't too many
  - rough guess: try to keep table half empty
  - i.e. choose $n > 2m$

Hash Function

- Having chosen a size for the table, the next question is, how should we define the hash function h?
- What we want is a function that will "scatter" the keys randomly throughout the table
- The simple example used before (check the first letter) is not a good choice for a large table
  - gives only 26 possible starting locations for English words
- Some other ideas:
  - look at the first k letters (e.g. first three, or first five)
  - not a good choice for keys like "Array.reverse", "Array.shift", "Array.sort", etc
  - choose k letters from "random" locations, e.g. 1st, 3rd, 6th, 11th

Aside: ASCII Characters

- Hash functions that use characters from strings rely on the fact that the characters are based on a numeric code
- The most common code: ASCII
  - pronounced "ass-key"
  - acronym for "American Standard Code for Information Interchange"
- Original version had 128 entries (upper and lower case, digits, punctuation)
- Extended ASCII has 256 entries (math symbols, accented letters, etc)

```
>> s = "hello"
>> for i in 0..s.length-1 do puts s[i] end
104
101
108
108
111
```


ASCII (cont'd)

- The first step in building a hash function in Ruby might be to sum up the codes for the letters to use
- Example: for a hash function that uses the first four letters in a word:

```
>> s = "reverse"
>> x = 0
>> for i in 0..3 do x += s[i] end
>> x
=> 434
```


Multiplicative Hash Function

- A neat mathematical technique (described in Ch. 43 of *NTO*) is the final step in defining a useful hash function
- Now that we have an initial integer value x based on the letters of the word, define the hash function as

$$h(x) = n((\phi x) \bmod 1)$$

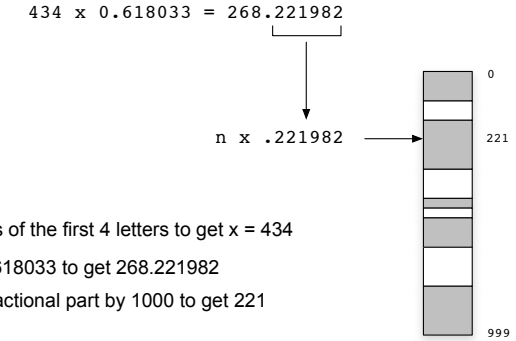
  where ϕ is the "golden ratio":

$$\phi = (\sqrt{5} - 1)/2 \approx 0.618033$$

- In other words, "multiply the fractional part of ϕx by n"
  - (diagram on the next slide)

Multiplicative Hash Function (cont'd)

- Example: finding a place for "reverse" in a table of size n = 1000



(1) sum the codes of the first 4 letters to get x = 434

(2) multiply by 0.618033 to get 268.221982

(3) multiply the fractional part by 1000 to get 221

Summary

- An index is an ***array of strings***
  - if the array is not sorted you need a ***linear search*** to look up a string
  - in a ***sorted*** array you can use a ***binary search***
  - a ***binary search tree*** is equivalent to a sorted array but may be easier to manage when new strings are added
- A ***hash table*** is an array with additional empty slots
  - use a ***hash function*** h(S) to find the location for string S
  - make sure you have lots of extra slots to minimize the number of ***collisions***

Google

- As an example of how these ideas are used in practice let's look at how Google does its searches
- Google uses a "web crawler" to go out on the internet and fetch pages
  - their goal is to get a copy of every page on the internet
- Issues:
  - don't get dynamic pages (e.g. from on-line games)
  - respect privacy: some sites don't want to be searched
    - example: course web site with solutions to problem sets
    - book publishers ask faculty members to protect the pages containing solutions
- Once the pages are collected organize them so searches are fast
  - main index or "forward index" links a page ID to its location in the system
  - a "reverse index" is an index for every word found on a page
  - connects words to pages that contain them

Google (cont'd)

- When processing a page to build the reverse index, consider the context
  - words that occur in headlines (e.g. page titles) and section headers are given a higher weight than words appearing in the page body
  - extract words appearing in links
- What made Google different: rank pages by "importance"
  - an important page is one that others refer to
- Example: search for "ducks"
  - Ducks Unlimited (a conservation organization)
  - Ducks of the World (a university environmental education site)
  - Ducks for Kids and Teachers (by ???)
  - Ducks at a Distance (USGS)
  - Anaheim Mighty Ducks (pro hockey team)
  - University of Oregon Official Athletic Site

Google (cont'd)

- From a 1998 paper:

  "After each document is parsed... every word is converted into a wordID by using an in-memory hash table"

  The Anatomy of a Large-Scale Hypertextual Web Search Engine, Sergey Brin and Lawrence Page, Computer Networks 30(1-7): 107-117 (1998) (search "google.pdf" to get a copy from a Stanford publication library)

- Some statistics from that era:
  - the search engine used a "lexicon" of 14,000,000 words
  - extracted from 24,000,000 pages
  - pages took up 147GB
  - word index required 37GB
  - word list (lexicon) was 293MB

Google (cont'd)

- Is a lexicon of 14,000,000 entries big enough?
- Depending on how you define "word" the English language has between 100,000 and 250,000 words
  - is "hot dog" a word?
  - how does punctuation work? is "pseudocode" the same as "pseudo-code"?
  - do you include place names? people? species names? ("*Yersinia pestis*" ?)

Review

- Know the basic search methods (linear search, binary search)
  - given a sorted list show which items are compared during a lookup
- Given the definition of a hash function be able to show how a hash table is filled
  - find the location based on h(S)
  - show the sequence of locations scanned when collisions occur