COMP20007 Design of Algorithms

Hashing

Daniel Beck

Lecture 19

Semester 1, 2023

Dictionaries - Recap

- Abstract Data Structure: collection of (key, value) pairs.
- Required operations: Search, Insert, Delete
- Last lecture: Binary Search Trees (and extensions)
- This lecture: Hash Tables.

Hash Tables

- A hash table is a continuous data structure with m preallocated entries.
- Average case performance for Search, Insert and Delete: $\Theta(1)$
- Requires a hash function: $h(K) \rightarrow i \in [0, m-1]$.
- A hash function should:
  - Be efficient ($\Theta(1)$).
  - Distribute keys evenly (uniformly) along the table.

Collisions

- Happens when the hash function give identical results to two different keys.
- We saw two solutions:
  - Separate Chaining
  - Linear Probing
- Practical efficiency will depend on the table load factor: $\alpha = n/m$. (n is the number of keys, m is the size of the table)

Separate Chaining

Assign multiple records per cell (usually through a linked list)

- Assuming even distribution of the n keys.
- A sucessful search requires $1 + \alpha/2$ operations on average.
- An unsucessful search requires α operations on average.
- Almost same numbers for Insert and Delete.
- Worst case $\Theta(n)$ only with a bad hash function (load factor is more of an issue).
- Requires extra memory.

Linear Probing

Populate successive empty cells.

- Much harder analysis, simplified results show:
- A sucessful search requires $(1/2) \times (1 + 1/(1-\alpha))$ operations on average.
- An unsucessful search requires $(1/2) \times (1 + 1/(1-\alpha)^2)$ operations on average.
- Similar numbers for Insert. Delete virtually impossible.
- Does not require extra memory.
- Worst case $\Theta(n)$ with a bad hash function and/or clusters.

Double Hashing

- A generalisation of Linear Probing.
- Apply a second hash function in case of collision.
  - First try: $h(K)$
  - Second try: $(h(K) + s(K)) \mod m$
  - Third try: $(h(K) + 2s(K)) \mod m$
  - ...
- Both Linear Probing and Double Hashing are sometimes referred as Open Addressing methods.

Rehashing

- High load factors deteriorate the performance of a hash table (for linear probing, ideally we should have $\alpha < 0.9$).
- Rehashing allocates a new table (usually around double the size) and move every item from the previous table to the new one.
- Very expensive operation, but happens infrequently.

Hashing Integers

- For large/unbounded integers, a standard hash function is $h(K) = K \mod m$
- Small m results in lots of collisions, large m takes excessive memory. Best m will vary.
- It is a good idea to use a prime number for m (especially for double hashing).

Hashing Strings

- Assume A $\mapsto 0$, B $\mapsto 1$, etc.
- Assume 26 characters and $m = 101$, as an example.
- Each character can be mapped to a binary string of length 5 ($2^5 = 32$).

We can think of a string as a long binary number:

M Y K E Y $\mapsto 0110011000010100010011000$ $(= 13379736)$

$$13379736 \bmod 101 = 64$$

So 64 is the position of string M Y K E Y in the hash table.

Hashing Strings

- Assume *chr* be the function that gives a character's number, so for example, $chr(\text{c}) = 2$
- We can represent this concatenation as an equation:

$$h(s) = \left(\sum_{i=0}^{|s|-1} chr(s_i) \times 32^{|s|-i-1}\right) \bmod m,$$

- In other words:

$$h(\text{M Y K E Y})$$
$$= 12 \times 32^4 + 24 \times 32^3 + 10 \times 32^2 + 4 \times 32^1 + 24 \times 32^0$$
$$= 13379736$$

Hashing Long Strings

- Long strings can quickly become difficult to calculate computationally.

$$h(\mathrm{V\ E\ R\ Y\ L\ O\ N\ G\ K\ E\ Y})$$
$$= (21 \times 32^{10} + 4 \times 32^{9} + \cdots)\ mod\ 101$$

- The term between parenthesis can become quite large and result in overflow.

Horner's Rule

- Instead of

$$21 \times 32^{10} + 4 \times 32^9 + 17 \times 32^8 + 24 \times 32^7 \cdots$$

factor out repeatedly:

$$(\cdots ((21 \times 32 + 4) \times 32 + 17) \times 32 + \cdots) + 24$$

- Now utilize these properties of modular arithmetic:

$$\begin{aligned} (x + y) \bmod m &= ((x \bmod m) + (y \bmod m)) \bmod m \\ (x \times y) \bmod m &= ((x \bmod m) \times (y \bmod m)) \bmod m \end{aligned}$$

So for each sub-expression it suffices to take values modulo m.

Horner's Rule

$$
\begin{aligned}
h(\mathrm{E\ Y}) &= (4 \times 32^1 + 24 \times 32^0) \mod m \\
&= (4 \times 32 + 24) \mod m \\
&= (((4 \times 32) \mod m) + (24 \mod m)) \mod m \\
&= (((4 \mod m) \times (32 \mod m) \mod m) \\
&+ 24 \mod m) \mod m
\end{aligned}
$$

Summary

Hash Tables:

- Implement dictionaries.
- Allow $\Theta(1)$ Search, Insert and Delete in the average case.
- Preallocates memory (size m).
- Requires good *hash functions*.
- Requires good collision handling.

Pros and Cons

If Hash Tables are so good, why bother with BSTs?

- Memory requirements are much higher.
- Hash Tables ignore *key ordering*, unlike BSTs.
- Queries like "give me all records with keys between 100 and 200" are easy within a BST but much less efficient in a hash table.

That being said, if hashing is applicable, a well-tuned hash table will typically outperform BSTs.

In Practice

Python dictionaries (*dict* type)

- Open addressing using *pseudo-random probing*
- Rehashing happens when $\alpha = 2/3$

C++ *unordered_maps*

- Uses chaining.
- Rehashing happens when $\alpha = 1$

Next lecture: Data Compression

COMP20007 Design of Algorithms

Data Compression

Daniel Beck

Lecture 20

Semester 1, 2023

Introduction

- So far, we talked about speed and space performance from an algorithm point of view.
- We assumed that records could fit in memory. (although we did mention secondary memory in Mergesort and B-trees)
- What to do when records are *too large?* (videos, for instance)

Fixed-length encoding

For text files, suppose each character has an fixed-size binary code.

B A G G E D

01000010 01000001 01000111 01000111 01000101 01000100

This is exactly what ASCII does.

Key insight: this coding has *redundant* information.

Run-length encoding

010000100100000101000111010001110100010101000100

01401201501013031013031013010101 30120

Character-level:

B A G G E D $\rightarrow$ B A 2 G E D

AAAABBBAABBBBBCCCCCCCCDABCBAAABBBBCCCD

4A3BAA5B8CDABCB3A4B3CD

Run-length encoding

- While not very useful for text data, it can work for some kinds of binary data.
- For text, the best algorithms move away from using fixed-length codes (ASCII).

Variable-length Encoding

- **Key idea:** some symbols appear more *frequently* than others.
- Instead of a fixed number of bits per symbol, use a *variable* number:
  - More frequent symbols use less bits.
  - Less frequent symbols use more bits.
- For this scheme to work, no symbol code can be a prefix of another symbol's code.

Variable-Length Encoding

Suppose we count symbols and find these numbers of occurrences:

Symbol	Weight
B	4
D	5
G	10
F	12
C	14
E	27
A	28

Here are some sensible codes that we may use for symbols:

Symbol	Code
A	11
B	0000
C	011
D	0001
E	10
F	010
G	001

Encoding a string

- Codes can be stored in a dictionary
- Once we have the codes, encoding is straightforward.
- For example, to encode 'BAGGED', simply concatenate the codes for B, A, G, G, E and D:

  000011001001100001

A	11
B	0000
C	011
D	0001
E	10
F	010
G	001

Decoding a string

- To decode we can use another dictionary where keys are codes and values are symbols.
- Starting from the first digit, look in the dictionary. If not present, concatenate the next digit and repeat until code is valid.

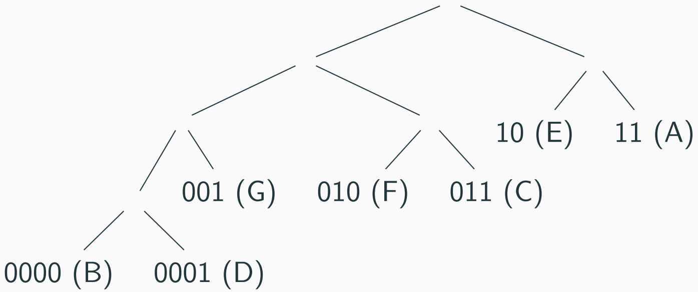
11	A
0000	B
011	C
0001	D
10	E
010	F
001	G

000011001001100001

Seems like it requires lots of misses, is there a better way?

Tries

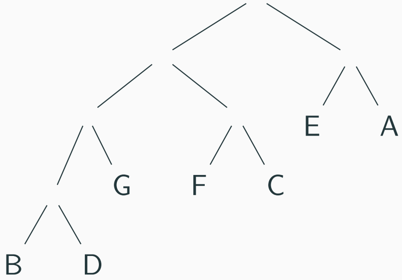
- Another implementation of a dictionary.
- Works when keys can be decomposed



This specific trie stores values only in the leaves $\rightarrow$ keeps prefix property.

Tries

To decode 000011001001100001, use the trie, repeatedly starting from the root, and printing each symbol found as a leaf.



How to choose the codes?

Huffman Encoding

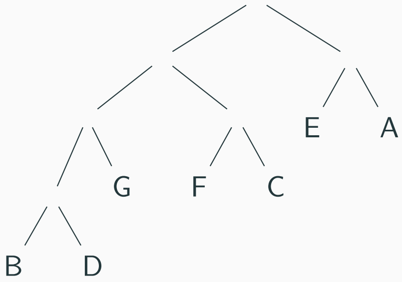
- Goal: obtain the optimal encoding given symbol frequencies. Optimal means shortest encoding.
- Treat each symbol as a leaf and build a binary tree bottom-up.
- Two nodes are fused if they have the smallest frequency.
    - Fusing means creating a parent node with the sum of frequencies of fused nodes.
- The resulting tree is a Huffman tree.

Huffman Trees

4	5	10	12	14	27	28
B	D	G	F	C	E	A

Tries

We end up with the Trie we showed before!



Summary

- Most data we store in our computer has *redundancy*.
- Compression uses redundancy to reduce space.
- Huffman is based on variable-length encoding.
- Tries to store codes.

Data Compression - In Practice

- Huffman encoding provides the basis for many advanced compression techniques.
- Lempel-Ziv compression assigns codes to *sequences of symbols*: used in GIF, PNG and ZIP.
- For sequential data (audio/video), an alternative is linear prediction: *predict* the next frame given the previous ones. Used in FLAC.
- **Lossy compression:** JPEG, MP3, MPEG and others. Also employ Huffman (among other techniques).

Next lecture: Complexity Theory