COMP20007 Design of Algorithms

Week 10 Workshop

Tutorial

1. Separate chaining Consider a hash table in which the elements inserted into each slot are stored in a linked list. The table has a fixed number of slots $L = 2$. The hash function to be used is $h(k) = k \mod L$.

Show the hash table after insertion of records with the keys

17 6 11 21 12 33 5 23 1 8 9

Can you think of a better data structure to use for storing the records in each slot?

2. Open addressing Consider a hash table in which each slot can hold one record and additional records are stored elsewhere in the table using linear probing with steps of size $i = 1$. The table has a fixed number of slots $L = 8$ The hash function to be used is $h(k) = k \mod L$.

Show the hash table after insertion of records with the keys

17 7 11 33 12 18 9

Repeat using linear probing with steps of size $i = 2$. What problem arises, and what constraints can we place on i and L to prevent it?

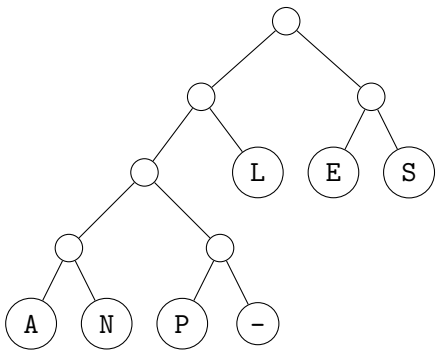
Can you think of a better way to find somewhere else in the table to store overflows?

3. Huffman code generation Huffman's Algorithm generates prefix-free code trees for a given set of symbol frequencies. Using these algorithms generate two code trees based on the frequencies in the following message:

`losslesscodes`

What is the total length of the compressed message using the Huffman code?

4. Canonical Huffman decoding The following code tree was generated using Huffman's algorithm, and converted into a Canonical Huffman code tree. Note: `_` denotes space.



Assign codewords to the symbols in the tree, such that left branches are denoted `0` and right branches are denoted `1`.

Use the resulting code to decompress the following message:

`00100110000011100011011011110011110110100010011011110001101111`

5. Asymptotic Complexity Classes (Revision) For each pair of the following functions, indicate whether $f(n) \in \Omega(g(n))$, $f(n) \in O(g(n))$ or both (in which case $f(n) \in \Theta(g(n))$).

(a) $f(n) = (n^3 + 1)^6$ and $g(n) = (n^6 + 1)^3$,

(b) $f(n) = 3^{3n}$ and $g(n) = 3^{2n}$,

(c) $f(n) = \sqrt{n}$ and $g(n) = 10n^{0.4}$,

(d) $f(n) = 2\log_2\{(n + 50)^5\}$ and $g(n) = (\log_e(n))^3$,

(e) $f(n) = (n^2 + 3)!$ and $g(n) = (2n + 3)!$,

(f) $f(n) = \sqrt{n^5}$ and $g(n) = n^3 + 20n^2$.

The following result may be useful,

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = \begin{cases} 0 & \text{implies that } f(n) \text{ has a smaller order of growth than } g(n), \\ c & \text{implies that } f(n) \text{ has the same rder of growth than } g(n), \\ \infty & \text{implies that } f(n) \text{ has a larger order of growth than } g(n). \end{cases}$$

Computer Lab

These exercises center around building hash functions for strings. Download `main.c`, `strhash.c`, `strhash.h`, and `words.txt` (2.5MB) from the LMS. Compile these files using the command `gcc -o hash strhash.c main.c` (or create a Makefile).

The resulting program `hash` reads strings from standard input and counts how many strings hash to each bucket in a hash table. The size of the table, and the hash function to use, are specified through the command line. The task for this lab is to use this program to explore the distributions produced by different hash functions.

1. Horrible hash functions At the moment, `strhash.c` provides only two hash functions: 'hash everything to bucket 0' and 'hash everything to a random bucket'. The first is terrible because it only uses one bucket. The second evenly distributes keys among all buckets, but it isn't even really a hash function (why?).

Run `./hash 14 0` and `./hash 14 r`. In each case, enter your favourite 10-20 words, one per line, then press control+D (MinGW: control+Z, enter). Run the programs again, this time taking input from `words.txt` (using `<`).

2. Bad hash functions Implement two new hash functions in `strhash.c`. For the first, hash a string to the ASCII value of its first character, mod table size `size`. For the second, hash a string to its length, mod table size `size`.

Modify the `hash()` and `name()` functions at the bottom of `strhash.c` to allow these functions to be used by `main.c`. Recompile the program and experiment with these hash functions using different table sizes.

Note: for large table sizes, you might want to pipe the output into `less` (e.g by running a command like `./hash 100 r < words.txt | less`) to make it easier to view (exit `less` by pressing `q`).

(Optional) implement a third bad hash function which treats the bits in the first few characters of the input string as as an unsigned integer, and returns that integer modulo `size`. Be careful of the case where the string does not have enough characters to form an integer.

3. Universal hash function Implement the following universal hash function for strings:

$$h(s, m) = \left(\sum_{i=0}^{len(s)-1} r[i] \times s[i] \right) \mod m$$

where r is a fixed array of random integers (hint: use a `static` flag to initialise this array only the first time you call the function). You may assume that 128 is the maximum possible value for $len(s)$.

Add your universal hash function to `hash()` and `name()`, and compare the bucket distributions it achieves with those of the other hash functions.