COMP20007 Design of Algorithms

Week 12 Workshop Solutions

Tutorial

1. Separate chaining Here's the hash table after inserting the keys according to the hash function $h(k) = k \mod L$ with $L = 2$:

$$\begin{array}{r|l} 0 & 6 \to 12 \to 8 \\ 1 & 17 \to 11 \to 21 \to 33 \to 5 \to 23 \to 1 \to 9 \end{array}$$

In terms of better data structures over a standard linked list, there are plenty of options to try.

- A move-to-front (MTF) list could adapt to access patterns to provide better average performance
- An array (possibly also with MTF) could save on space for all those `next` pointers, and could and also yield better cache performance*.
- A balanced search tree could ensure $O(\log n)$ lookups in the worse case even if the hash function distributes keys unevenly.
  However, before reaching for more complicated data structures as a cure for poor hashing performance, it might be better to try increasing the table size and/or improving the hash function.

*Cache performance is related to how nicely your algorithm behaves in its memory access patterns. It's a matter of practical concern as it has a real impact on algorithm performance. With a MTF linked list, memory accesses might be distant from one another as we follow pointers to wildly different parts of memory. In contrast, the elements of an array are *contiguous*, so subsequent accesses are likely to be in nearby areas of memory.
Cache performance is not really a concern in this subject, but it's definitely something to be aware of.

2. Open addressing Here's the hash table after inserting the keys according to the hash function $h(k) = k \mod L$ with $L = 8$:

0	1	2	3	4	5	6	7
	17	33	11	12	18	9	7

If we repeat using $i = 2$, we can insert the first 6 keys without much trouble, but then we can't find a place for 9:

0	1	2	3	4	5	6	7
	17	18	11	12	33		7
	9?		9?		9?		9?

The problem is that $i = 2$ and $L = 8$ have a common factor other than 1 (it's 2), in maths terms, they are not *coprime*. So, it's possible for the key 9 to fall into a loop of steps that doesn't actually encounter every cell. As a result, we can't insert 9, even though there are still empty buckets in the hash table.

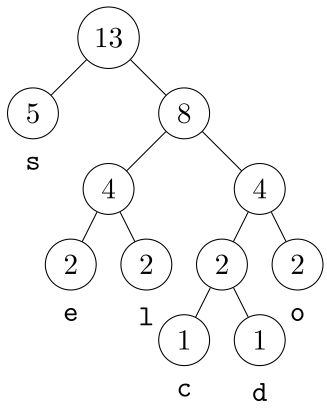
We can avoid this problem if we make sure i and L have no common factors other than 1. Appropriate choices include: L a power of 2 and i odd, or L prime and $i < L$.

In terms of better open addressing strategies, one major improvement comes from the idea of double hashing (choosing a different i for each element). Double hashing can help to eliminate clustering. However, based on the discussion above, we must place certain constraints on the output of the second hash function to ensure that it's coprime with L. Also, it should always be greater than 0 (why?).

3. Huffman code generation Frequency counts:

`s`	`l`	`o`	`e`	`c`	`d`
5	2	2	2	1	1

A possible Huffman tree:



Using 0 for left branches and 1 for right branches we get the following codewords:

`s`	0
`l`	101
`o`	111
`e`	100
`c`	1100
`d`	1101

Hence, the encoded version of `losslesscodes` is:

101 111 0 0 101 100 0 0 1100 111 1101 100 0

Other trees are also possible depending on how you break ties while constructing the tree. All trees give a total message length of 31 bits (sum of codeword lengths multiplied by frequencies).

4. Canonical Huffman decoding The code is:

symbol	codeword	length
`A`	`0000`	4
`N`	`0001`	4
`P`	`0010`	4
`_`	`0011`	4
`E`	`01`	2
`L`	`10`	2
`S`	`11`	2

The message is:

`PLEASE LESS SLEEPLESSNESS`

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) Using the binomial theorem or otherwise we get,

$$f(n) = n^{3\times 6} + \binom{6}{1} n^{3\times 5} + \binom{6}{2} n^{3\times 4} + \binom{6}{3} n^{3\times 3} + \binom{6}{4} n^{3\times 2} + \binom{6}{5} n^{3\times 1} + 1,$$

and

$$g(n) = n^{6\times 3} + 3n^{6\times 2} + 3n^{6\times 1} + 1$$

so $f(n) \in \Theta(g(n))$ as the heighest growing terms are n^{18}.

(b)

$$f(n) = 3^{3n} = 27^n \quad \text{and} \quad g(n) = 3^{2n} = 9^n$$

So,

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = \lim_{n\to\infty} \frac{27^n}{9^n} = \lim_{n\to\infty} \left(\frac{27}{9}\right)^n = \infty$$

As such $f(n) = \Omega(g(n))$.

(c) $f(n) \in \Theta(n^{0.5})$ and $g(n) = 10n^{0.4} \in \Theta(n^{0.4})$. $f(n)$ grows faster. So $f(n) \in \Omega(g(n))$.

(d)

$$f(n) = 2\log_2\{(n+50)^5\} = 10\log_2(n+50)$$

and

$$g(n) = (\log_e(n))^3 = \frac{(\log_2(n))^3}{\text{const}}$$

Now,

$$\begin{aligned}
\lim_{n\to\infty} \frac{f(n)}{g(n)} &= \lim_{n\to\infty} \frac{10\log_2(n+50)}{\frac{(\log_2(n))^3}{\text{const}}} \\
&= \lim_{n\to\infty} \left(\text{const} \times \frac{\log_2(n+50)}{\log_2(n)} \times \frac{1}{(\log_2(n))^2}\right) \\
&= \text{const} \times \left(\lim_{n\to\infty} \frac{\log_2(n+50)}{\log_2(n)}\right) \times \left(\lim_{n\to\infty} \frac{1}{(\log_2(n))^2}\right) \\
&= \text{const} \times 1 \times 0 = 0
\end{aligned}$$

So $f(n) \in O(g(n))$.

To see why $\frac{\log_2(n+50)}{\log_2(n)} \to 1$ we apply l'Hopital's rule. Since $\log_2(n+50) \to \infty$ and $\log_2(n) \to \infty$ as $n \to \infty$ we can claim that,

$$\begin{aligned}
\lim_{n\to\infty} \frac{\log_2(n+50)}{\log_2(n)} &= \lim_{n\to\infty} \frac{\frac{1}{\ln 2}}{\frac{1}{\ln 2}} \times \frac{\ln(n+50)}{\ln(n)} = \lim_{n\to\infty} \frac{\ln(n+50)}{\ln(n)} = \lim_{n\to\infty} \frac{\frac{d}{dn}(\ln(n+50))}{\frac{d}{dn}(\ln(n))} \\
&= \lim_{n\to\infty} \frac{\frac{1}{n+50}}{\frac{1}{n}} = \lim_{n\to\infty} \frac{n}{n+50} = \lim_{n\to\infty} \frac{1}{1+\frac{50}{n}} = 1.
\end{aligned}$$

(e)

$$\begin{aligned}
\lim_{n\to\infty} \frac{f(n)}{g(n)} &= \lim_{n\to\infty} \frac{(n^2+3)!}{(2n+3)!} \\
&= \lim_{n\to\infty} \frac{(n^2+3)(n^2+2)\cdots(2n+4)(2n+3)(2n+2)\cdots 1}{(2n+3)(2n+2)\cdots 1} \\
&= \lim_{n\to\infty} (n^2+3)(n^3+2)\cdots(2n+4) \\
&= \infty
\end{aligned}$$

So $f(n) \in \Omega(g(n))$.

(f) $f(n) = n^{2.5}$ and $g(n) = n^3 + 20n^2$. Since n^3 grows faster than $n^{2.5}$ we have $f(n) \in O(g(n))$.

5. (Revision) Quicksort & Mergesort

(a)

```
[3, 8, 5, 2, 1, 3, 5, 4, 8]
 ^
 p

[3, 8, 5, 2, 1, 3, 5, 4, 8]
 i                           j

[3, 8, 5, 2, 1, 3, 5, 4, 8]
    i           j

Swap:
[3, 3, 5, 2, 1, 8, 5, 4, 8]
    i           j

[3, 3, 5, 2, 1, 8, 5, 4, 8]
       i     j

Swap:
[3, 3, 1, 2, 5, 8, 5, 4, 8]
       i     j

[3, 3, 1, 2, 5, 8, 5, 4, 8]
          j  i

Crossed Over => Stop
Swap A[l] with A[j]:
[2, 3, 1, 3, 5, 8, 5, 4, 8]

Done!
```

(b)

```
      [3, 8, 5, 2, 1, 3, 5, 4, 8]
       p
      [2, 1, 3, 3, 8, 5, 5, 4, 8]
                p
   [2, 1, 3]               [8, 5, 5, 4, 8]
    p                       p
   [1, 2, 3]               [5, 5, 4, 8, 8]
       p                             p

  [1]     [3]            [5, 5, 4]       [8]
                          p
                         [5, 4, 5]
                                p

                        [5, 4]
                         p
                        [4, 5]
                            p
                        [4]
=> [1, 2, 3, 3, 4, 5, 5, 8, 8]
```

(c)

```
[3,   8,   5,     2,   1,    3,   5,    4,   8]
[3,   8,   5,     2,   1]   [3,   5,    4,   8]
[3,   8,   5]    [2,   1]   [3,   5]   [4,   8]
[3,   8]  [5]    [2,   1]   [3,   5]   [4,   8]
[3]  [8]  [5] |  [2]  [1] | [3]  [5] | [4]  [8]
[3,   8]  [5] |  [2]  [1] | [3]  [5] | [4]  [8]
[3,   5,   8] |  [1,   2] | [3,   5] | [4,   8]
[1,   2,   3,     5,   8] | [3,   4,    5,   8]
[1,   2,   3,     3,   4,    5,   5,    8,   8]
```

7. (Optional) Karp-Rabin Hashing First we must compute the hash of the pattern $P =$"CAB":

$$\begin{aligned} h(\text{"CAB"}) &= a^2 \cdot \text{chr(C)} + a \cdot \text{chr(A)} + \text{chr(B)} \mod m \\ &= 4^2 \cdot 2 + 4 \cdot 0 + 1 \mod 11 \\ &= 33 \mod 11 \\ &= 0 \end{aligned}$$

As discussed in the question we can compute $h(\text{"CAD"})$ like so:

$$\begin{aligned} h(\text{"CAD"}) &= a^2 \cdot \text{chr(C)} + a \cdot \text{chr(A)} + \text{chr(D)} \mod m \\ &= 4^2 \cdot 2 + 4 \cdot 0 + 3 \mod 11 \\ &= 35 \mod 11 \\ &= 2 \end{aligned}$$

Then we can compute each successive substring of 3 characters like so:

$$\begin{aligned} h(\text{"ADA"}) &= 4\left(h(\text{"CAD"}) - 4^2 \cdot 2\right) + 0 \mod m \\ &= 4(2 - 32) + 0 \mod 11 \\ &= 4(-30) + 0 \mod 11 \\ &= 4(-30 \mod 11) \mod 11 \\ &= 4(3) \mod 11 \\ &= 12 \mod 11 \\ &= 1 \end{aligned}$$

From $h(\text{"ADA"})$ we can compute $h(\text{"DAC"})$ like so:

$$\begin{aligned} h(\text{"DAC"}) &= 4\left(h(\text{"ADA"}) - 4^2 \cdot 0\right) + 2 \mod m \\ &= 4(1) + 2 \mod 11 \\ &= 6 \mod 11 \\ &= 6 \end{aligned}$$

Completing this for the rest of the substrings with 3 characters in T we get:

$$\begin{aligned} h(T[0 \dots 2]) &= h(\text{"CAD"}) = 2 \\ h(T[1 \dots 3]) &= h(\text{"ADA"}) = 1 \\ h(T[2 \dots 4]) &= h(\text{"DAC"}) = 6 \\ h(T[3 \dots 5]) &= h(\text{"ACA"}) = 8 \\ h(T[4 \dots 6]) &= h(\text{"CAB"}) = 0 \end{aligned}$$

Notice that $h(T[4 \dots 6]) = 0$ and $h(P) = 0$. Therefore, $T[0 \dots 2]$ *may* be equal to P.

Why can we not be sure that $T[0 \dots 2] = P$, as we may get a **collision** with our hash function. The hash function may give the same result for two different substrings, for instance h("DBCD") also equals 0. As a result we must check manually that the strings match. If we choose a large m then these collisions become increasingly less likely, and thus the additional computation complexity required by checking "false positives" (*i.e.*, when the hash function values are the same but the strings differ) becomes negligible.

How about the time complexity of this algorithm? Initially computing $h(S)$ for a string S of n characters takes $O(n)$ time. So computing $h(P)$ takes $O(|P|)$ time.

Note that there are $|T| + 1 - |P|$ substrings of length $|P|$ in T. The first of which takes $O(|P|)$ time to hash. However, due to the ability to compute hashes incrementally in $O(1)$ time the remaining $|T| - |P|$ substrings only take $O(1)$ time each. Thus the total time complexity of computing the hashes for all substrings becomes:

$$|P| + (|T| - |P|) = O(|T|).$$

Taking into account the time complexity of computing the hash of P we get a total time complexity of the Karp-Rabin string search algorithm of $O(|T| + |P|)$.