# COMP90038

# Algorithms and Complexity


Lecture 17: Hashing

(with thanks to Harald Søndergaard & Michael Kirley)

Andres Munoz-Acosta

munoz.m@unimelb.edu.au

Peter Hall Building G.83

# Recap

- We talked about using some memory space (in the form of extra tables, arrays, etc.) to speed up our computation.
  - Memory is cheap, time is not.



- Sorting by counting

- Horspool's Algorithm

# Sorting by counting

- Lets go through this example carefully:
  - The keys are: [1 2 3 4 5]
  - The data is: [5 5 1 5 4 1 2 3 5 5 1 5 5 3 5 1 3 5 4 5]



- Lets count the appearances of each key:



| Key | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|
| Occurrences | | | | | |



- Lets add up the occurrences

| | | | | | |
|---|---|---|---|---|---|
| Occurrences | | | | | |
| | | | | | |
| Cumulation | | | | | |

# Sorting by counting

- Lets sort the data:

| Key | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|
| Cumulation | 4 | 5 | 8 | 10 | 20 |
| P[20] | | | | | 19 |
| P[10] | | | | 9 | |
| P[19] | | | | | 18 |
| P[8] | | | 7 | | |



| Index | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 | 19 | 20 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Unsorted | 5 | 5 | 1 | 5 | 4 | 1 | 2 | 3 | 5 | 5 | 1 | 5 | 5 | 3 | 5 | 1 | 3 | 5 | 4 | 5 |
| Sorted | | | | | | | | 3 | | 4 | | | | | | | | | 5 | 5 |

# Horspool's algorithm

- Lets go through this example carefully:
  - The pattern is TAACG (A=1, T=2, G=3, C=4 → P[.] = [2 1 1 4 3], m =5)
  - The string is GACCGCGTGAGATAACGTCA
- This algorithm creates the table of shifts:



**function** FINDSHIFTS($P[\,]$, $m$)
  **for** $i \leftarrow 0$ to $alphasize - 1$ **do**
    $Shift[i] \leftarrow m$
  **for** $j \leftarrow 0$ to $m - 2$ **do**
    $Shift[P[j]] \leftarrow m - (j + 1)$

| | A | T | G | C |
|---|---|---|---|---|
| After first loop | 5 | 5 | 5 | 5 |
| j=0 | 5 | 4 | 5 | 5 |
| j=1 | 3 | 4 | 5 | 5 |
| j=2 | 2 | 4 | 5 | 5 |
| j=3 | 2 | 4 | 5 | 1 |

# Horspool’s algorithm

- We append a **sentinel** at the end of the data to guarantee completion
  - The string is now GACCGCGTGAGATAACGTCATAACG



| | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 | 19 | 20 | 21 | 22 | 23 | 24 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| STRING | G | A | C | C | G | C | G | T | G | A | G | A | T | A | A | C | G | T | C | A | T | A | A | C | G |
| T[.] | 3 | 1 | 4 | 4 | 3 | 4 | 3 | 2 | 3 | 1 | 3 | 1 | 2 | 1 | 1 | 4 | 3 | 2 | 4 | 1 | 2 | 1 | 1 | 4 | 3 |
| FAILED (C!=A) | T | A | A | C | G | | | | | | | | | | | | | | | | | | | | |
| IS 'CG' SOMEWHERE ELSE? | T | A | A | C | G | | | | | | | | | | | | | | | | | | | | |
| (NO, SHIFT BY G) | | | | | | | | | | | | | | | | | | | | | | | | | |
| FAILED (A!=G, SHIFT BY A) | | | | | | T | A | A | C | G | | | | | | | | | | | | | | | |
| FAILED (A!=G, SHIFT BY A) | | | | | | | | T | A | A | C | G | | | | | | | | | | | | | |
| FAILED (A!=G, SHIFT BY A) | | | | | | | | | | T | A | A | C | G | | | | | | | | | | | |
| FAILED (C!=G, SHIFT BY C) | | | | | | | | | | | | T | A | A | C | G | | | | | | | | | |
| FOUND AT 16 | | | | | | | | | | | | | T | A | A | C | G | | | | | | | | |

# Horspool's algorithm

- For this algorithm, at the end of **while** True **do** iteration, [*i* *k*] are:

```
function HORSPOOL(P[·], m, T[·], n)
    FINDSHIFTS(P, m)
    i ← m − 1
    while True do
        k ← 0
        while k < m and P[m − 1 − k] = T[i − k] do
            k ← k + 1
        if k = m then
            if i ≥ n then
                return −1
            else
                return i − m + 1
        i ← i + Shift[T[i]]
```

| i | k |
|---|---|
| 9 | 2 |
| 11 | 0 |
| 13 | 0 |
| 15 | 0 |
| 16 | 0 |

# Hashing

- **Hashing** is a standard way of implementing the abstract data type “dictionary”, a collection of <attribute name, value> pairs. For example an student record:



  - Attributes: Student ID, Name, data of birth, address, major, etc…



- Implemented well, it makes data retrieval very fast.



- A **key** identifies each record. It can be anything: integers, alphabetical characters, even strings
  - It should map efficiently to a positive integer.
  - The set *K* of keys need not be bounded.

# Hashing

- We will store our records in a **hash table** of size *m*.
  - *m* should be large enough to allow efficient operation, without taking up excessive memory.



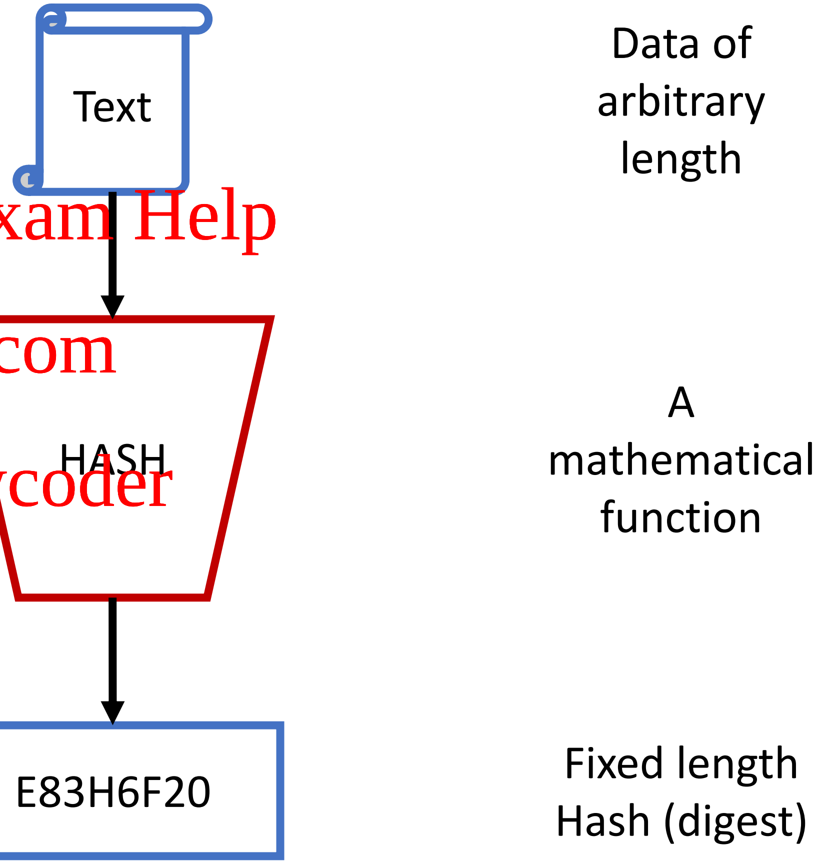
- The idea is **to have a function *h* that takes the key *k***, and determines an index in the hash table. This is the **hash function**.
  - A record with key *k* should be stored in location *h*(*k*).



- The **hash address** is the value of *h*(*k*).
  - Two different keys could have the same hash address (a collision).

# Hashing

- Few example application are:
  - The MD5 algorithm used for data integrity verification.
  - The blockchain structure used in crypto currencies



# The Hash Table

- We can think of the **hash table** as an abstract data structure supporting operations:
  - Find
  - Insert
  - Lookup (search and insert if not found)
  - Initialize
  - Delete
  - Rehash



- The challenges in implementing a table are:
  - Design a robust hash function
  - Handling of same addresses (collisions) for different key values

# The Hash Function

- The hash function:
  - Must be easy (cheap) to compute.
  - Ideally distribute keys evenly across the hash table.



- Examples:
  - If the keys are integers, we could define $h(n) = n \bmod m$. If m=23:

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| n | 19 | 392 | 179 | 359 | 262 | 321 | 97 | 468 |
| h(n) | 19 | 1 | 18 | 14 | 9 | 22 | 5 | 8 |

  - If the keys are strings, we could define a more complex function.

# Hashing of strings

- Assume:
  - this table of 26 characters.
  - a hash table of size 13
  - the hash function:

$$h(s) = \left( \sum_{i=0}^{|s|-1} \mathrm{a}_i \right) \mod 13$$

  - and the following list of keys:

    [A, FOOL, AND, HIS, MONEY, ARE, SOON, PARTED]



| char | a |
|---|---|
| A | 0 |
| B | 1 |
| C | 2 |
| D | 3 |
| E | 4 |
| F | 5 |
| G | 6 |
| H | 7 |
| I | 8 |

| char | a |
|---|---|
| J | 9 |
| K | 10 |
| L | 11 |
| M | 12 |
| N | 13 |
| O | 14 |
| P | 15 |
| Q | 16 |
| R | 17 |

| char | a |
|---|---|
| S | 18 |
| T | 19 |
| U | 20 |
| V | 21 |
| W | 22 |
| X | 23 |
| Y | 24 |
| Z | 25 |
| | |

# Calculating the addresses

| | | | | | | SUM | *h*(*s*) |
|---|---|---|---|---|---|---|---|
| A | | | | | | | |
| 0 | | | | | | 0 | 0 |
| F | O | O | L | | | | |
| 5 | 14 | 14 | 11 | | | 44 | 5 |
| A | N | D | | | | | |
| 0 | 13 | 3 | | | | 16 | 3 |
| H | I | S | | | | | |
| 7 | 8 | 18 | | | | 33 | **7** |
| M | O | N | E | Y | | | |
| 12 | 14 | 13 | 4 | 24 | | 67 | 2 |
| A | R | E | | | | | |
| 0 | 17 | 4 | | | | 21 | 8 |
| S | O | O | N | | | | |
| 18 | 14 | 14 | 13 | | | 59 | **7** |
| P | A | R | T | E | D | | |
| 15 | 0 | 17 | 19 | 4 | 3 | 58 | 6 |

The same address!!!

# A more complex hash function

- Assume a binary representation of the 26 characters
  - We need **5 bits** per character (0 to 31)
- Instead of adding, we **concatenate** the binary strings
- Our hash table is of size 101 (***m* is prime**)
- Our key will be 'MYKEY'



| char | a | bin(a) |
|---|---|---|
| A | 0 | 00000 |
| B | 1 | 00001 |
| C | 2 | 00010 |
| D | 3 | 00011 |
| E | 4 | 00100 |
| F | 5 | 00101 |
| G | 6 | 00110 |
| H | 7 | 00111 |
| I | 8 | 01000 |

| char | a | bin(a) |
|---|---|---|
| J | 9 | 01001 |
| K | 10 | 01010 |
| L | 11 | 01011 |
| M | 12 | 01100 |
| N | 13 | 01101 |
| O | 14 | 01110 |
| P | 15 | 01111 |
| Q | 16 | 10000 |
| R | 17 | 10001 |

| char | a | bin(a) |
|---|---|---|
| S | 18 | 10010 |
| T | 19 | 10011 |
| U | 20 | 10100 |
| V | 21 | 10101 |
| W | 22 | 10110 |
| X | 23 | 10111 |
| Y | 24 | 11000 |
| Z | 25 | 11001 |

# A more complex hash function

| | STRING | | | | | KEY | KEY mod 101 |
|---|---|---|---|---|---|---|---|
| | M | Y | K | E | Y | | |
| int | 12 | 24 | 10 | 4 | 24 | | |
| bin(int) | 01100 | 11000 | 01010 | 00100 | 11000 | | |
| Index | 4 | 3 | 2 | 1 | 0 | | |
| | | | | | | | |
| 32^(index) | 1048576 | 32768 | 1024 | 32 | 1 | | |
| a*(32^index) | 12582912 | 786432 | 10240 | 128 | 24 | 13379736 | 64 |



- By concatenating the strings, we are basically multiplying by 32
- Note that the hash function is a polynomial:

$$h(s) = a_{|s|-1}32^{|s|-1} + a_{|s|-2}32^{|s|-2} + \cdots + a_1 32 + a_0$$

# Handling Long Strings as Keys

- What would happen if our key is the longer string ‘VERYLONGKEY’



$$h(VERYLONGKEY) = (21 \times 32^{10} + 4 \times 32^{9} + \cdots + 4 \times 32 + 24) \mod 101$$



- The stuff between parentheses quickly becomes a very large number quickly
  - DEC: 23804165628760600
  - BIN: 1010100100100011100001100110100011110100001001000011000



- Calculating this polynomial by brute force is very expensive

# Horner's rule

- Fortunately there is a trick, **Horner's rule**, that simplifies polynomial calculation.



$$p(x) = a_3 \times x^3 + a_2 \times x^2 + a_1 \times x + a_0$$



- By factorizing *x* we have that:



$$p(x) = ((a_3 \times x) + a_2) \times x + a_1) \times x + a_0$$

- If we apply the modulus we have:

$$p(x) = ((((a_3 \times x) + a_2) \times x + a_1) \times x + a_0) \mod m$$

# Horner's rule

- We then can use the following properties of modular arithmetic:



$$x \boxplus y = (x + y) \mod m = ((x \mod m) + (y \mod m)) \mod m$$



$$x \boxtimes y = (x \times y) \mod m = ((x \mod m) \times (y \mod m)) \mod m$$



- Given that modulus **distributes across all operations**, then we have:

$$p(x) = (((((a_3 \boxtimes x) \boxplus a_2) \boxtimes x) \boxplus a_1) \boxtimes x) \boxplus a_0$$

- The results of each **operation will not exceed *m*.**

# Handling collisions

- The hash function should be as random as possible.



- However, in some cases different keys will be mapped to the same hash table address. For example $h(n) = n \bmod 23$



| KEY | 19 | 392 | 179 | 359 | 663 | 262 | 639 | 321 | 97 | 468 | 814 |
|---|---|---|---|---|---|---|---|---|---|---|---|
| ADDRESS | 19 | 1 | 18 | 14 | 19 | 9 | 18 | 22 | 5 | 8 | 9 |



- When this happens we have a **collision**.

- Different hashing methods resolve collisions differently.

# Separate Chaining

- Each element *k* of the hash table is a **linked list**, which makes collision handling very easy



| ADDRESS | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 | 19 | 20 | 21 | 22 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| LIST | | 392 | | | | 97 | | | 468 | 262 | | | | | 359 | | | | 179 | 19 | | | |
| | | | | | | | | | | 814 | | | | | | | | | 639 | 663 | | | |





- **Exercise**: add to this table [83 110 14]

- The **load factor** $\alpha = n/m$, where $n$ is the number of items stored.
  - Number of probes in **successful** search $\sim(1 + \alpha)/2$.
  - Number of probes in **unsuccessful** search $\sim\alpha$.

# Separate chaining: advantages and disadvantages

- Compared with **sequential search**, reduces the number of comparisons by the size of the table (a factor of *m*).



- Good in a dynamic environment, when (number of) keys are hard to predict.



- The chains can be ordered, or records may be “pulled up front” when accessed.



- Deletion is easy.
- However, separate chaining uses extra storage for links.

# Open-Addressing Methods

- With **open-addressing** methods (also called **closed hashing**) all records are stored in the hash table itself (not in linked lists hanging off the table).



- There are many methods of this type. We focus on two:
  - **linear probing**
  - **double hashing**



- For these methods, the load factor $\alpha \leq 1$.

# Linear probing

- In case of collision, try the next cell, then the next, and so on.



- Assume the following data (and its **keys**) arriving one at the time:



[19(**19**) 392(**1**) 179(**18**) 663(**19**→**20**) 639(**18**→**21**) 321(**22**) ...]



- Search proceeds in similar fashion
- If we get to the end of the table, we wrap around.
- For example, if key 20 arrives, it will be placed in cell 0.

# Linear probing

- **Exercise**: Add [83(**14**) 110(**18**) 497(**14**)] to the table



| ADDRESS | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 | 19 | 20 | 21 | 22 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| LIST | | 392 | | | | 97 | | | 468 | 262 | 814 | | | | 359 | | | | 179 | 19 | 663 | 639 | 321 |



- Again let *m* be the table size and *n* be the number of records stored.



- As before, $\alpha = n/m$ is the load factor. Then, the average number of probes:
  - Successful search: $0.5 + 1/(2(1-\alpha))$
  - Unsuccessful: $0.5 + 1/(2(1-\alpha)^2)$

# Linear probing: advantages and disadvantages

- Space-efficient.
- Worst-case performance miserable; must be careful not to let the load factor grow beyond 0.9.



- Comparative behavior, $m$ = 11113, $n$ = 10000, $\alpha$ = 0.9:
  - Linear probing: 5.5 probes on average (success)
  - Binary search: 12.3 probes on average (success)
  - Linear search: 5000 probes on average (success)
- **Clustering** (large groups of contiguous keys) is a major problem:
  - The collision handling strategy leads to clusters of contiguous cells being occupied.
- Deletion is almost impossible.

# Double Hashing

- To alleviate the clustering problem in linear probing, there are better ways of resolving collisions.



- One is **double hashing** which uses a second hash function *s* to determine an **offset** to be used in probing for a free cell.



- For example, we may choose $s(k) = 1 + k \bmod 97$.



- By this we mean, if $h(k)$ is occupied, next try $h(k) + s(k)$, then $h(k) + 2s(k)$, and so on.

- This is another reason why **it is good to have *m* being a prime number**. That way, using $h(k)$ as the offset, we will eventually find a free cell if there is one.

# Rehashing

- The standard approach to avoiding performance deterioration in hashing is to keep track of the load factor and to **rehash** when it reaches, say, 0.9.



- Rehashing means allocating a larger hash table (typically about twice the current size), revisiting each item, calculating its hash address in the new table, and inserting it.

- This **"stop-the-world"** operation will introduce long delays at unpredictable times, but it will happen relatively infrequently.

# An exam question type

- With the hash function $h(k) = k \bmod 7$. Draw the hash table that results after inserting in the given order, the following values



- When collisions are handled by:
  - separate chaining
  - linear probing
  - double hashing using $h'(k) = 5 - (k \bmod 5)$

# Solution



| Index | 0 | 1 | 2 | 3 | 4 | 5 | 6 |
|---|---|---|---|---|---|---|---|
| | | | | | | | |
| Separate Chaining | | | | | | | |
| | | | | | | | |
| | | | | | | | |
| Linear Probing | | | | | | | |
| | | | | | | | |
| Double Hashing | | | | | | | |

# Rabin-Karp String Search

- The Rabin-Karp string search algorithm is based on string hashing.

- To search for a string $p$ (of length $m$) in a larger string $s$, we can calculate $hash(p)$ and then check every substring $s_i \ldots s_{i+m-1}$ to see if it has the same hash value. Of course, if it has, the strings may still be different; so we need to compare them in the usual way.



- If $p = s_i \ldots s_{i+m-1}$ then the hash values are the same; otherwise the values are **almost certainly** going to be different.

- Since false positives will be so rare, the $O(m)$ time it takes to actually compare the strings can be ignored.

# Rabin-Karp String Search

- Repeatedly hashing strings of length *m* seems like a bad idea. However, the hash values can be calculated **incrementally**. The hash value of the length-*m* substring of *s* that starts at position *j* is:



$$\mathrm{hash}(s, j) = \sum_{i=0}^{m-1} \mathrm{chr}(s_{j+i}) \times a^{m-i-1}$$





- where *a* is the alphabet size. From that we can get the next hash value, for the substring that starts at position *j*+1, **quite cheaply**:

$$\mathrm{hash}(s, j+1) = (\mathrm{hash}(s, j) - a^{m-1}\mathrm{chr}(s_j)) \times a + \mathrm{chr}(s_{j+m})$$

- modulo *m*. Effectively we just subtract the contribution of $s_j$ and add the contribution of $s_{j+m}$, for the cost of two multiplications, one addition and one subtraction.

# An example

- The data '31415926535'
- The hash function $h(k) = k \bmod 11$
- The pattern '26'



| STRING | 3 | 1 | 4 | 1 | 5 | 9 | 2 | 6 | 5 | 3 | 5 |
|---|---|---|---|---|---|---|---|---|---|---|---|
| 31 MOD 11 | | 9 | | | | | | | | | |
| 14 MOD 11 | | | 3 | | | | | | | | |
| 41 MOD 11 | | | | 8 | | | | | | | |
| 15 MOD 11 | | | | | 4 | | | | | | |
| 59 MOD 11 | | | | | | 4 | | | | | |
| 92 MOD 11 | | | | | | | 4 | | | | |
| 26 MOD 11 | | | | | | | | 4 | | | |

# Why Not Always Use Hashing?

- Some drawbacks:



  - If an application calls for traversal of all items in sorted order, a hash table is no good.





  - Also, unless we use separate chaining, deletion is virtually impossible.

  - It may be hard to predict the volume of data, and rehashing is an expensive "stop-the-world" operation.

# When to Use Hashing?

- All sorts of information retrieval applications involving thousands to millions of keys.



- Typical example: Symbol tables used by compilers. The compiler hashes all (variable, function, etc.) names and stores information related to each – no deletion in this case.





- When hashing is applicable, it is usually superior; a well-tuned hash table will outperform its competitors.

- **Unless** you let the load factor get too high, or you botch up the hash function. It is a good idea to print statistics to check that the function really does spread keys uniformly across the hash table.

# Next lecture



- Dynamic programming and optimization.