Hashing

Tables: rows & columns of information

- A table has several fields (types of information). For **example** a user account table may have fields user id, username, password
- To find an entry in the table, you only need know the contents of one of the fields (not all of them). This field is the key.
  - In a user account table, the key is usually user id
- Ideally, a key uniquely identifies an entry
- For searching purposes, it is best to store the key and the entry separately (even though the key's value may be inside the entry)

key	entry
112	112, "ABC", "abc112"
231	231, "XYZ", "xyz123"

2

Implementation 1: unsorted sequential array

- **insert**: add to back of array; O(1)
- **find**: search through the keys one at a time, potentially all of the keys; O(*n*)
- **remove**: find + replace removed node with last node; O(*n*)

	key	entry
0		
1		
2		
3		
⋮	*and so on*	

3

Implementation 2: sorted sequential array

- **insert**: add in sorted order; O(*n*)
- **find**: binary search; O($\log_2 n$)
- **remove**: find, remove node and shuffle down; O(*n*)

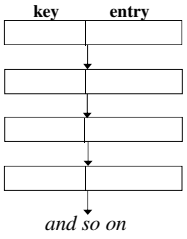
We can use binary search because the array elements are sorted

	key	entry
0		
1		
2		
3		
⋮	*and so on*	

4

Implementation 3: linked list (unsorted or sorted)

- **insert**: add to front; O(1)
  or O(n) for a sorted list
- **find**: search through potentially all the keys, one at a time; O(*n*)
  still O(n) for a sorted list
- **remove**: find, remove using pointer alterations; O(*n*)



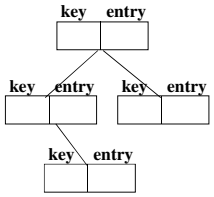
5

Implementation 4: AVL tree

- An AVL tree, ordered by key
- **insert**: a standard insert; O(log *n*)
- **find**: a standard find, O(log *n*)
- **remove**: a standard remove; O(log *n*)

O(log *n*) is very good…

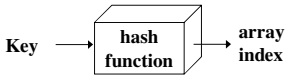
…but O(1) would be even better!



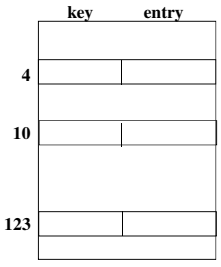
6

Implementation 5: *hashing*

- An array in which records are ***not*** stored consecutively - their place of storage is calculated using the key and a *hash function*



- *Hashed key*: the result of applying a hash function to a key
- Keys and entries are scattered throughout the array

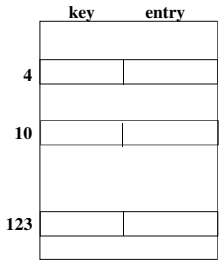


7

Implementation 5: *hashing*

- **insert**: calculate place of storage, insert record; O(1)
- **find**: calculate place of storage, retrieve entry; O(1)
- **remove**: calculate place of storage, set it to null; O(1)

All are O(1) !



8

Hashing example: a fruit shop

- 10 stock details, 10 array positions
- Stock numbers are between 0 and 1000
- Use *hash function*: stock no. / 100
- What if we now insert stock no. 350?
  - Position 3 is occupied: there is a *collision*
- *Collision resolution strategy*: insert in the next free position (*linear probing*)
- Given a stock number, we find stock by using the hash function again, and use the collision resolution strategy if necessary

	key	entry
0	85	85, apples
1		
2		
3	323	323, guava
4	462	462, pears
5	350	350, oranges
6		
7		
8		
9	912	912, papaya

9

Three factors affecting the performance of hashing

- The hash function
  - Ideally, it should distribute keys and entries evenly throughout the table.
  - It should minimise *collisions.*
- The collision resolution strategy
  - *chaining*: chain together several keys/entries in each hash key position
  - *Open addressing*: store the key/entry in a different position
- The size of the table
  - Too big will waste memory; too small will increase collisions and may eventually force *rehashing* (copying into a larger table)
  - Should be appropriate for the hash function used – and a prime number is best

10

Choosing a hash function: turning a key into a table position

- Truncation
  - Ignore part of the key and use the rest as the array index
  - A fast technique, but check for an even distribution throughout the table
- Mid Square Method
  - Key k is squared and hash function H(k) =l where l is obtained by deleting digits from both ends of k*k.
- Folding
  - Partition the key into several parts and then combine them in any convenient way
  - Unlike truncation, uses information from the whole key
- Modular arithmetic *(used by truncation & folding, and on its own)*
  - To keep the calculated table position within the table, divide the key by the size of the table, and take the remainder as the new position

11

Examples of hash functions

- Truncation: If students have an 9-digit identification number, take the last 3 digits as the table position
  - e.g. 925371622 becomes 622
- Mid square:- k = 3205 k*k = 10272025 and H(k) =72
- Folding: Split a 9-digit number into three 3-digit numbers, and add them
  - e.g. 925371622 becomes 925 + 376 + 622 = 1923
- Modular arithmetic: If the table size is 1000, the first example always keeps within the table range, but the second example does not (it should be mod 1000)
  - e.g. 1923 mod 1000 = 923

12

Choosing the table size to minimise collisions

- As the number of elements in the table increases, the likelihood of a *collision* increases - so make the table **as large as practical**
- Collisions may be more frequent if:
  - greatest common divisor (hashed keys, table size) > 1
- Therefore, make the table size **a prime number** (gcd = 1)

Collisions may still happen, so we need a *collision resolution strategy*

13

Collision resolution: open addressing

Probing: If the table position given by the hashed key is already occupied, increase the position by some amount, until an empty position is found

- **Linear probing**: increase by 1 each time
- **Quadratic probing**: to the original position, add 1, 4, 9, 16,…

Use the collision resolution strategy when inserting *and* when finding (ensure that the search key and the found keys match)

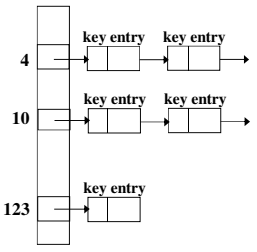
May also *double hash*: result of linear probing × result of another hash function

14

Collision resolution: chaining

- Each table position is a linked list
- Add the keys and entries anywhere in the list (front easiest)
- Advantages over open addressing:
  - Simpler insertion and removal
  - Array size is not a limitation (but should still minimise collisions: make table size roughly equal to expected number of keys and entries)
- Disadvantage
  - Memory overhead is large if entries are small

No need to change position!



15

Applications of Hashing

- Compilers use hash tables to keep track of declared variables
- A hash table can be used for on-line spelling checkers — if misspelling detection (rather than correction) is important, an entire dictionary can be hashed and words checked in constant time
- Game playing programs use hash tables to store seen positions, thereby saving computation time if the position is encountered again
- Hash functions can be used to quickly check for inequality — if two elements hash to different values they must be different

16

When are other representations more suitable than hashing?

- Hash tables are very good if there is a need for many searches in a reasonably stable table
- Hash tables are not so good if there are many insertions and deletions, or if table traversals are needed — in this case, AVL trees are better
- Also, hashing is very slow for any operations which require the entries to be sorted
  - e.g. Find the minimum key

17

Performance of Hashing

- The number of probes depends on the load factor (usually denoted by λ) which represents the ratio of entries present in the table to the number of positions in the array
- We also need to consider successful and unsuccessful searches separately
- For a chained hash table, the average number of probes for an unsuccessful search is λ and for a successful search is 1 + λ/2

18

Performance of Hashing (2)

- For open addressing, the formulae are more complicated but typical values are:

Load Factor	0.1	0.5	0.8	0.9	0.99
Successful search					
Linear Probes	1.05	1.6	3.4	6.2	21.3
Quadratic Probes	1.04	1.5	2.1	2.7	5.2
Unsuccessful search					
Linear Probes	1.13	2.7	15.4	59.8	430
Quadratic probes	1.13	2.2	5.2	11.9	126

- Note that these do not depend on the size of the array or the number of entries present but only on the ratio (the load factor)

19