Tables and Hashing

bitCode technologies

learn.innovate.share

Tables: rows & columns of information

- A *table* has several *fields* (types of information)
  - A telephone book may have fields **name**, **address**, **phone number**
  - A user account table may have fields **user id**, **password**, **home folder**

- To find an *entry* in the table, you only need know the contents of <u>one</u> of the fields (not <u>all</u> of them). This field is the *key*
  - In a telephone book, the key is usually **name**
  - In a user account table, the key is usually **user id**

- Ideally, a key *uniquely identifies* an entry
  - If the key is **name** and no two entries in the telephone book have the same name, the key uniquely identifies the entries

bitCode technologies

The Table ADT: operations

- **insert**: given a key and an entry, inserts the entry into the table

- **find**: given a key, finds the entry associated with the key

- **remove**: given a key, finds the entry associated with the key, *and* removes it

Also:

- **getIterator**: returns an iterator, which visits each of the entries one by one (the order may or may not be defined)

etc.

bitCode technologies

+ How should we implement a table?

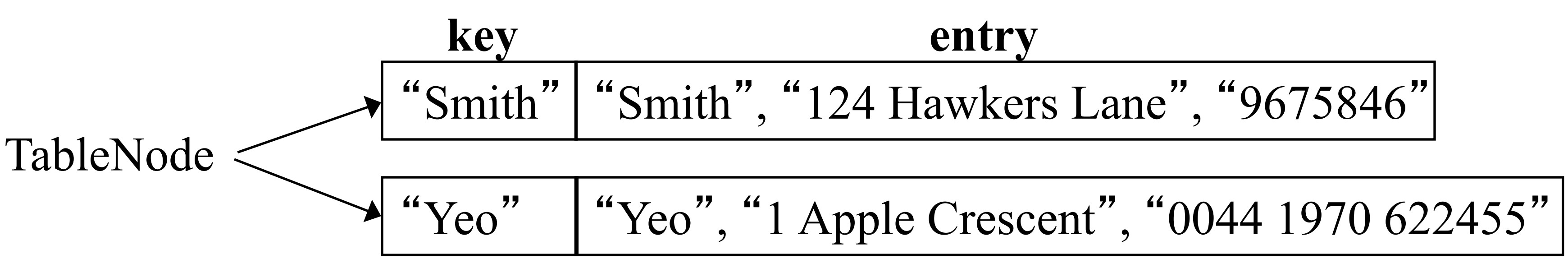
Our choice of representation for the Table ADT depends on the answers to the following

- How often are entries inserted and removed?
- How many of the possible key values are likely to be used?
- Is the table small enough to fit into memory?
- How long will the table exist?

bitCode technologies

TableNode: a key and its entry

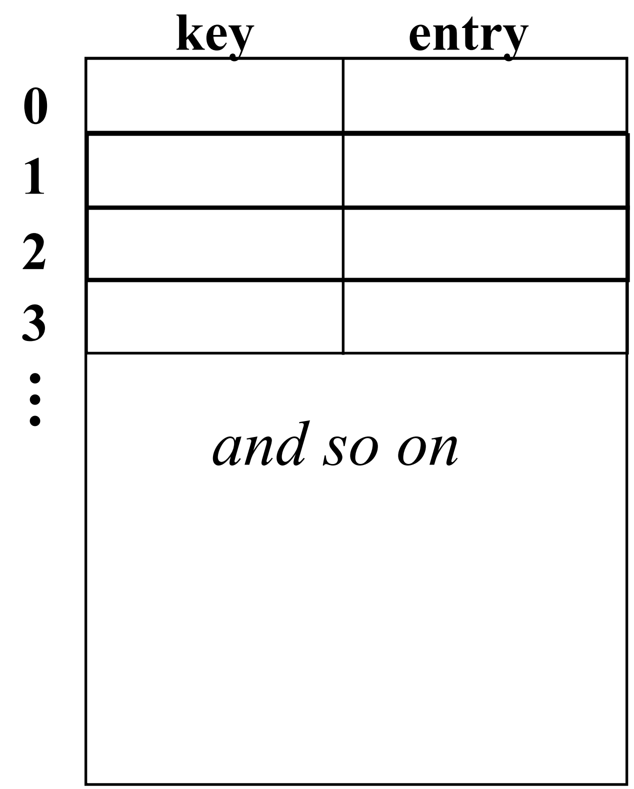
- For searching purposes, it is best to store the key and the entry separately (even though the key's value may be inside the entry)



bitCode technologies

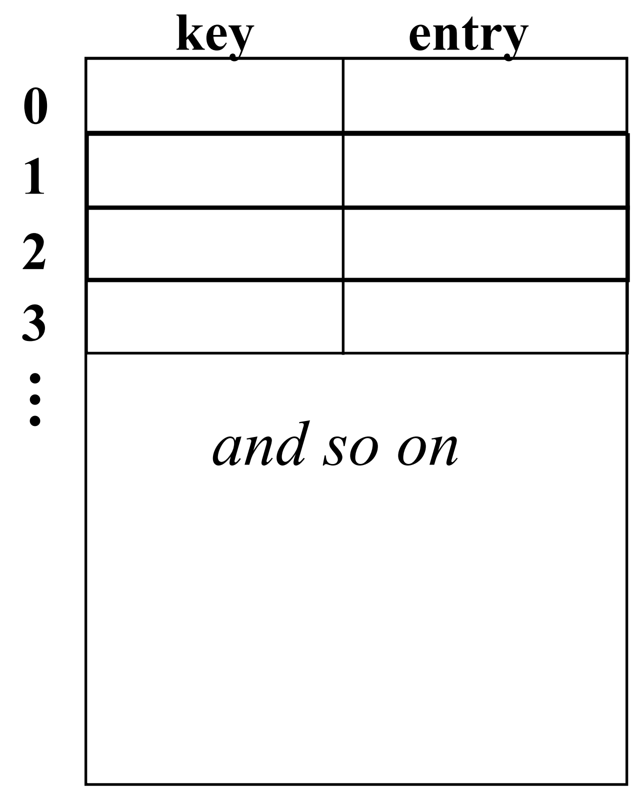
Implementation 1: unsorted sequential array

- An array in which TableNodes are stored consecutively in *any* order
- **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)



Implementation 2:sorted sequential array

- An array in which TableNodes are stored consecutively, *sorted* by key
- **insert**: add in sorted order; O(*n*)
- **find**: binary chop; O(log *n*)
- **remove**: find, remove node and shuffle down; O(*n*)

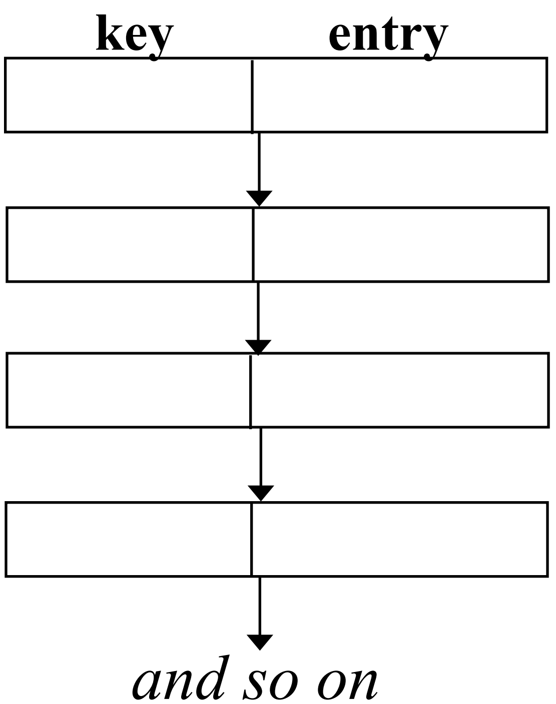


We can use binary chop because the array elements are sorted

bitCode technologies

Implementation 3:linked list (unsorted or sorted)

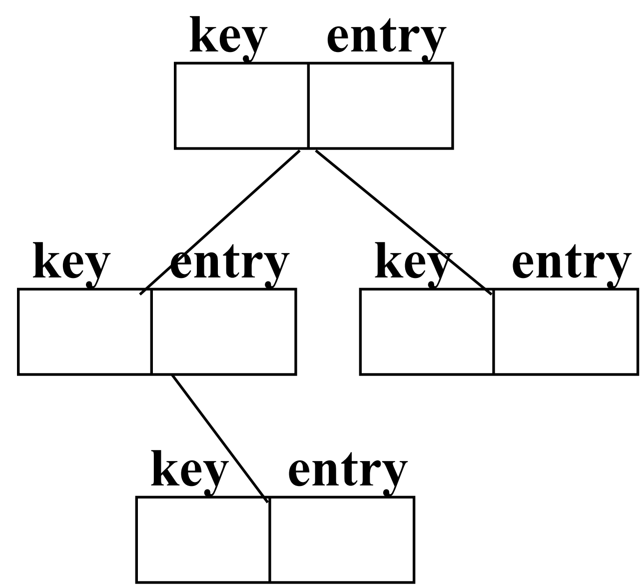
- TableNodes are again stored consecutively
- **insert**: add to front; O(1)
  or O(n) for a sorted list
- **find**: search through potentially all the keys, one at a time; O(*n*)
  still O(n) for a sorted list
- **remove**: find, remove using pointer alterations; O(*n*)



bitCode technologies

Implementation 4: AVL tree

- An AVL tree, ordered by key
- **insert**: a standard insert; O(log *n*)
- **find**: a standard find (without removing, of course); O(log *n*)
- **remove**: a standard remove; O(log *n*)



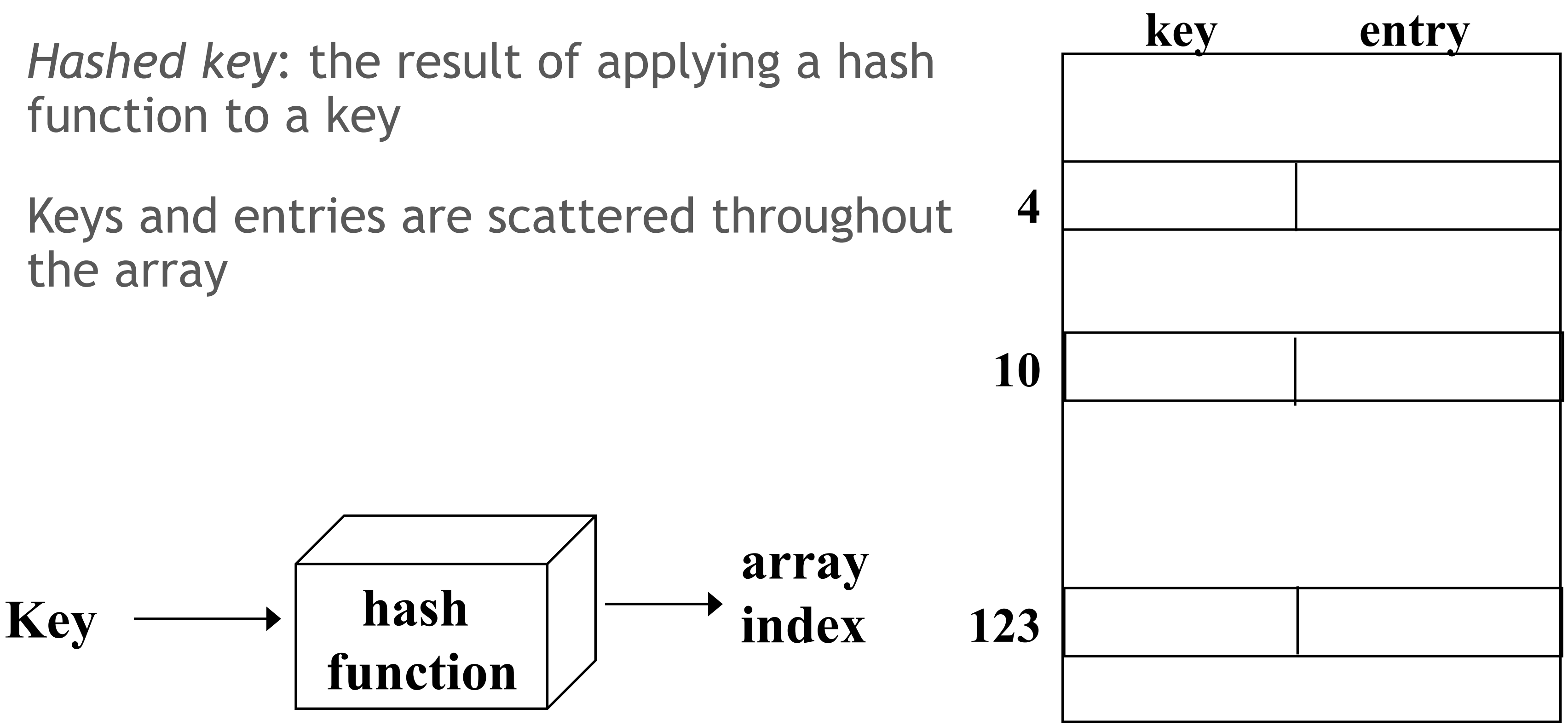
and so on

O(log *n*) is very good…

…but O(1) would be even better!

Implementation 5: *hashing*

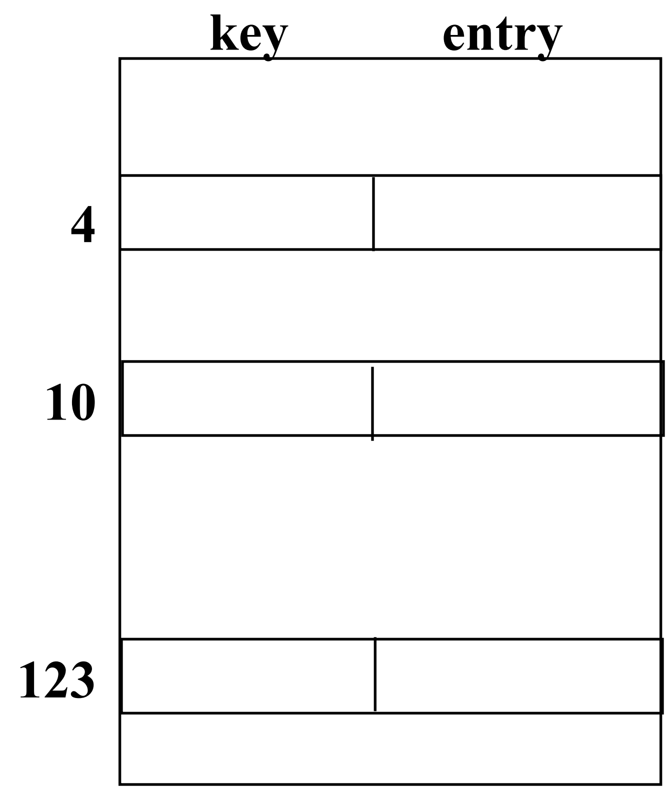
- An array in which TableNodes are ***not*** stored consecutively - their place of storage is calculated using the key and a *hash function*
- *Hashed key*: the result of applying a hash function to a key
- Keys and entries are scattered throughout the array



bitCode technologies

+ Implementation 5: *hashing*

- An array in which TableNodes are ***not*** stored consecutively - their place of storage is calculated using the key and a *hash function*
- **insert**: calculate place of storage, insert TableNode; 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) !

bitCode technologies

Hashing example: a fruit shop

- 10 stock details, 10 table 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

bitCode technologies

+ Three factors affecting performance of hashing

- The hash function
  - Ideally, it should distribute keys and entries evenly throughout the table
  - It should minimise *collisions*, where the position given by the hash function is already occupied
- The collision resolution strategy
  - *Separate chaining*: chain together several keys/entries in each 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

bitCode technologies

+ Examples of hash functions (1)

- Truncation: If students have an 9-digit identification number, take the last 3 digits as the table position
  - e.g. 925371622 becomes 622
- 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

bitCode technologies

Examples of hash functions (2)

- Using a telephone number as a key
  - The area code is not random, so will not spread the keys/entries evenly through the table (many collisions)
  - The last 3-digits are more random

- Using a name as a key
  - Use full name rather than surname (surname not particularly random)
  - Assign numbers to the characters (e.g. a = 1, b = 2; or use Unicode values)
  - Strategy 1: Add the resulting numbers. Bad for large table size.
  - Strategy 2: Call the number of possible characters c (e.g. c = 54 for alphabet in upper and lower case, plus space and hyphen). Then multiply each character in the name by increasing powers of c, and add together.

bitCode technologies

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**

- If the table size is 100, and all the hashed keys are divisible by 10, there will be many collisions!
  - Particularly bad if table size is a power of a small integer such as 2 or 10

- More generally, 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*

Collision resolution: open addressing (1)

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

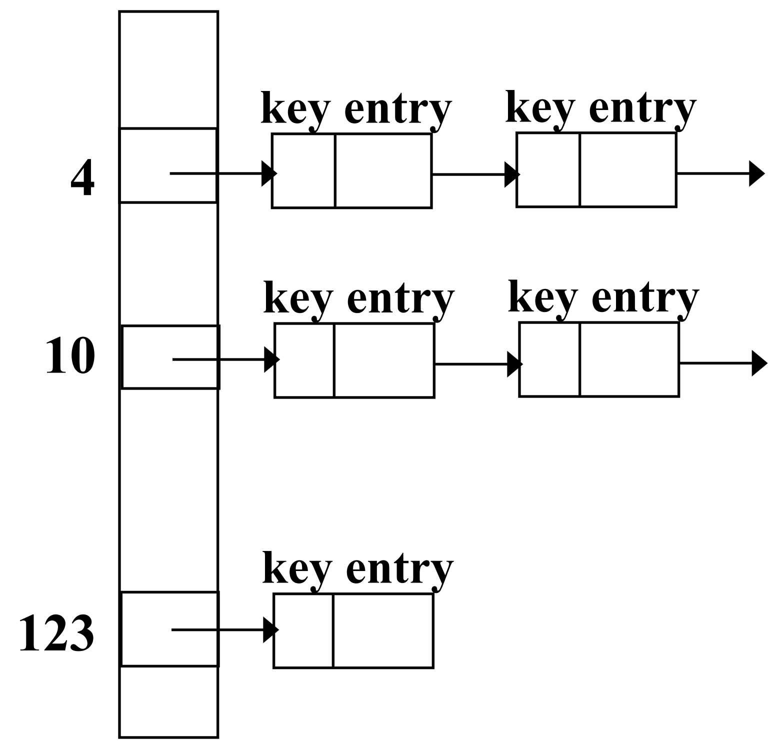
- **Linear probing:** increase by 1 each time
- **Quadratic probing:** to the original position, add 1, 4, 9, 16,...

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

Collision resolution: chaining

- Each table position is a linked list
- Add the keys and entries anywhere in the list (front easiest)
- Advantages:
  - 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!



bitCode technologies

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

bitCode technologies

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