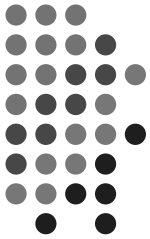
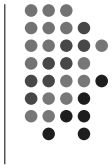
Sets & Maps 8B

Hash tables

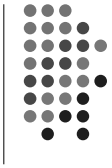




Hashing

- Data records are stored in a hash table.
- The position of a data record in the hash table is determined by its key.
- A hash function maps keys to positions in the hash table.
- If a hash function maps two keys to the same position in the hash table, then a collision occurs.

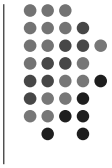
Example



- Let the hash table be an 11-element array.
- If k is the key of a data record, let H(k) represent the hash function, where
  H(k) = k mod 11.
- Insert the keys 83, 14, 29, 70, 10, 55, 72:

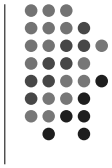
0	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*	*10*

Goals of Hashing



- An insert without a collision takes O(1) time.
- A search also takes O(1) time, if the record is stored in its proper location (without a collision).
- The hash function can take many forms:
  - If the key k is an integer:
    ```
    k % tablesize
    ```
  - If key k is a String (or any Object):
    ```
    k.hashCode() % tablesize
    ```
  - Any function that maps k to a table position!
- The table size should be a prime number.

Linear Probing



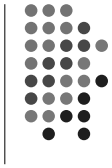
- During insert of key k to position p:

  If position p contains a different key, then examine positions p+1, p+2, etc.* until an empty position is found and insert k there.
- During a search for key k at position p:

  If position p contains a different key, then examine positions p+1, p+2, etc.* until either the key is found or an unused position is encountered.

**wrap around to beginning of array if p+i ≥ tablesize*

Linear Probing Example

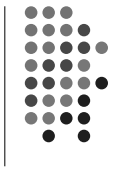


- Example: Insert additional keys 72, 36, 65, 48 using H(k) = k mod 11 and linear probing.

0	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*	*10*
55			**14**	**70**		**83**	**29**			**10**

Linear Probing can form clusters in the hash table.

Special consideration

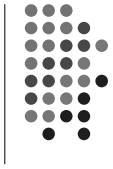


- If we remove a key from the hash table, can we get into problems?

0	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*	*10*
55			**14**	**70**	**36**		**29**	**72**	**48**	**10**

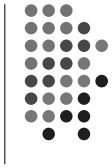
Remove 83. Now search for 48.
We can't find 48 due to the gap in position 6!
How can we solve this problem?

Efficiency
using Linear Probing



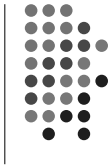
- Insert & Search for a hash table with n elements:
  - Expected (Average) Time: O(____)
  - Worst Case time O(____)

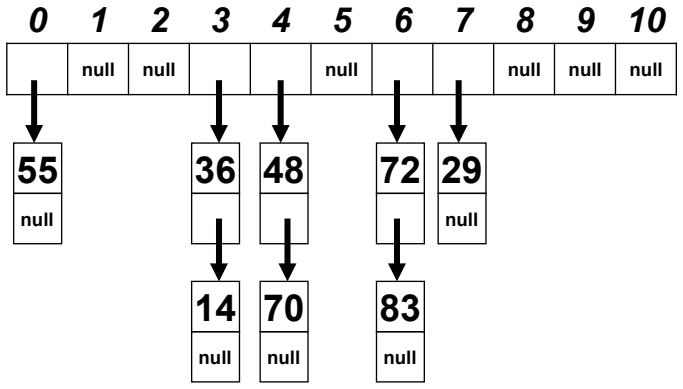
Chained Hashing



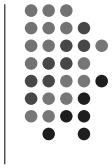
- The maximum number of elements that can be stored in a hash table implemented using an array is the table size.
- We can store more elements than the table size by using chained hashing.
  - Each array position in the hash table is a head reference to a linked list of keys (a "bucket").
  - All colliding keys that hash to an array position are inserted to that bucket.
- `HashMap` and `HashSet` use chained hashing.

Chaining



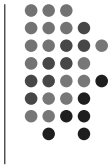


Hash codes and `Object`



- `Object` implements `equals` and `hashCode` methods, so each class we write inherits these.
  - `Object` calculates an object's hash code based on its address in memory.
  - Thus if two `Object`s are equal, their hash codes are equal also.
- If you override `equals` for a class, you should also override `hashCode` such that:
  - If `obj1.equals(obj2)` then `obj1.hashCode() == obj2.hashCode()`

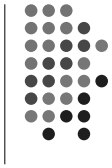
Hash Codes for Strings



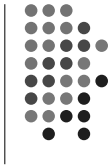
- Each character in a string has a unicode (int) value.
  'A'=65 'B'=66 'C'=67, ..., 'a'=97, 'b'=98, 'c'=99, ...
- Summing up the int values of the characters can lead to a lot of collisions:
  - "Act" "Cat" "Ads" sum of char codes = 280
- The `hashcode` method of the `String` class returns
  $s_0 \times 31^{n-1} + s_1 \times 31^{n-2} + ... + s_{n-1}$
  for the string $s_0 s_1 ... s_{n-1}$

String:	"Act"	"Cat"	"Ads"
hashcode:	65650	67510	67602

Birthday Paradox:
A Hashing Function



- Let k be a birthday.
  - Hash each birthday into a table of size 365 (one cell for each day of the year).
- Probability that n people don't have the same birthday: p = (364/365)*(363/365)*...*((365-n+1)/365)
  - When n $\geq$ 24, p < 0.5.
  - This means when n $\geq$ 24, chances are better that at least two people share the same birthday!
- For any hashing problem of reasonable size, we are almost certain to have collisions.

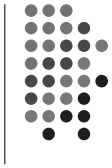


Load Factor

- The load factor α of a hash table with n elements is given by the following formula:

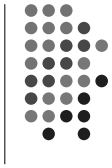
  α = n / table.length
- Thus, 0 $\leq \alpha \leq$ 1 for linear probing. (α can be greater than 1 for other collision resolution methods)
- For linear probing, as α approaches 1, the number of collisions increases

Reducing Collisions



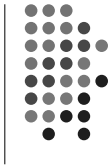
- The probability of a collision increases as the load factor increases.
- We cannot just double the size of the table and copy the elements from the original table to the new table.
  - Why?

Rehashing



- Algorithm:
  - Allocate a new hash table twice the size of the original table.
  - Reinsert each element of the old table into the new table (using the hash function).
  - Reference the new table as the hash table.
- `HashMap` and `HashSet` use a default load factor of 0.75 and an initial capacity of 16.

Average Search Time



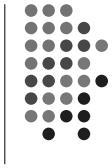
- In open-addressing, the average number of table elements examined in a successful search is approximately:

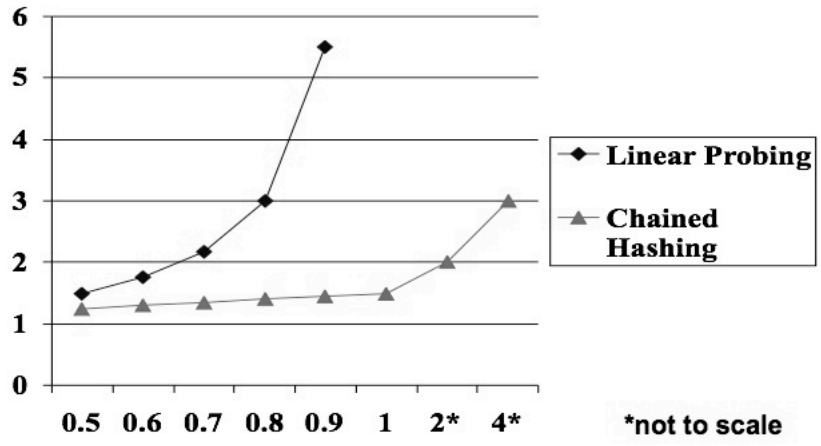
  $(1 + 1/(1-\alpha)) / 2$ using linear probing*

  $1 + \alpha/2$ using chained hashing

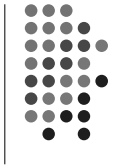
**assuming a non-full hash table with no removals*

Average number of searches during a successful search as a function of the load factor α



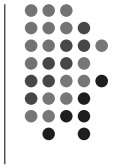


Example



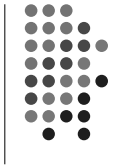
- If we have 60000 items to store in a hash table using open addressing (linear probing) and we desire a load factor of 0.75, how big should the hash table be?

Example (cont'd)



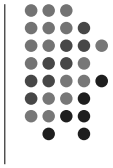
- If we have 60000 items to store in a hash table using open addressing (linear probing) and we have a load factor of 0.75, what is the expected number of comparisons to search for a key?

Example (cont'd)



- How large should the table size t be if we use chaining and desire the same expected number of comparisons as with the linear probing from the previous example?

Example (cont'd)



- How much memory is used in each case if each table holds 60000 keys with a load factor of 0.75?