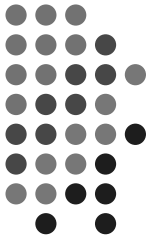
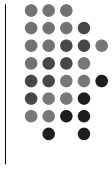
Sets & Maps 8B

Hash tables

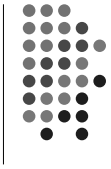




Hashing

- 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.

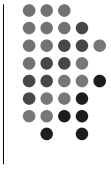
Example



- 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:

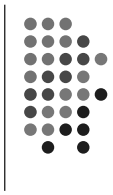
0	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*	*10*

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.

Hash Functions for String Values



```
int h(String x, int M) {
  char ch[];
  ch = x.toCharArray();
  int xlength = x.length();

  int i, sum;
  for (sum=0, i=0; i < x.length(); i++)
    sum += ch[i];
  return sum % M;
}
```

Some examples are as follows -

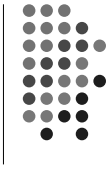
sum for the String “abc” = 97 + 98 + 99 = 294

Given M = 15, the result is 294 % 15 = 9

sum for the String “Hello” = 72 + 101 + 108 + 108 + 111= 500

Given M = 15, the result is 500 % 15 = 5

Linear Probing



- During insert of key k to position p:

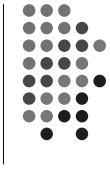
  If position p contains a different key, then examine positions p+1, p+2, etc.* until an empty position is found and insert k there.

- 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*

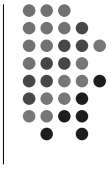
Linear Probing Example



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

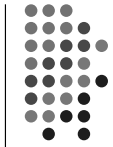
0	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*	*10*
55			**14**	**70**		**83**	**29**			**10**

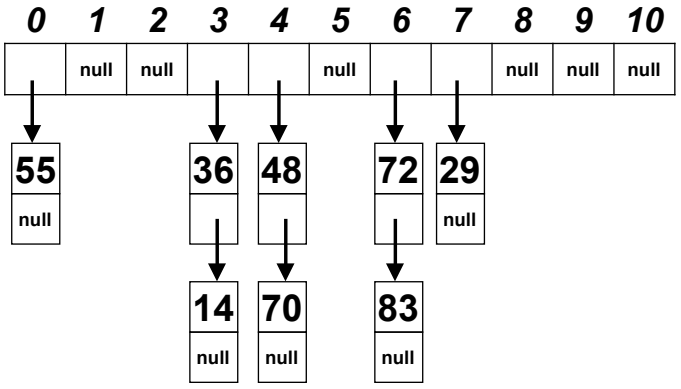
Chained Hashing



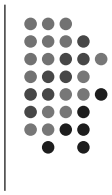
- The maximum number of elements that can be stored in a hash table implemented using an array is the table size.
- We can store more elements than the table size by using chained hashing.
  - Each array position in the hash table is a head reference to a linked list of keys (a "bucket").
  - All colliding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

Example: Insert additional keys 72, 36, 48, 29, 55, 14, 70, 83 using H(k) = k mod 11 and chained hashing.



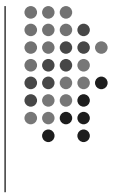


Use chained hashing and linear probing to insert the following keys:



103 8 108 208 308 10 9

Load Factor



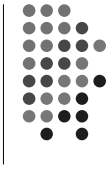
- The load factor α of a hash table with n elements is given by the following formula:

  α = n / table.length

- Thus, $0 < \alpha < 1$ for linear probing.
  (α can be greater than 1 for other collision resolution methods)

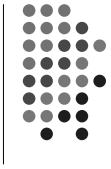
- For linear probing, as α approaches 1, the number of collisions increases

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.