

# CSCE 2110

## Foundations of Data Structures

---

### Hashing

# Dictionary ADT

---

- Data structure with just three basic operations:
  - **findItem (i)**: find item with key (identifier) i
  - **insert (i)**: insert i into the dictionary
  - **remove (i)**: delete i
  - Just like words in a Dictionary
- Where do we use them:
  - Symbol tables for compiler
  - Customer records (access by name)
  - Games (positions, configurations)
  - Spell checkers, etc.

# How to Implement a Dictionary?

---

- Sequences
  - ordered
  - unordered
- Binary Search Trees
- Hashtables

# Hashing

---

- An important and widely useful technique for implementing data structures such as dictionaries
- Constant time per operation (on the average)
- Worst case time proportional to the size of the set for each operation (just like array and chain implementation)

# Basic Idea

---

- Use **hash function** to map keys into positions in a hash table Constant time per operation (on the average)

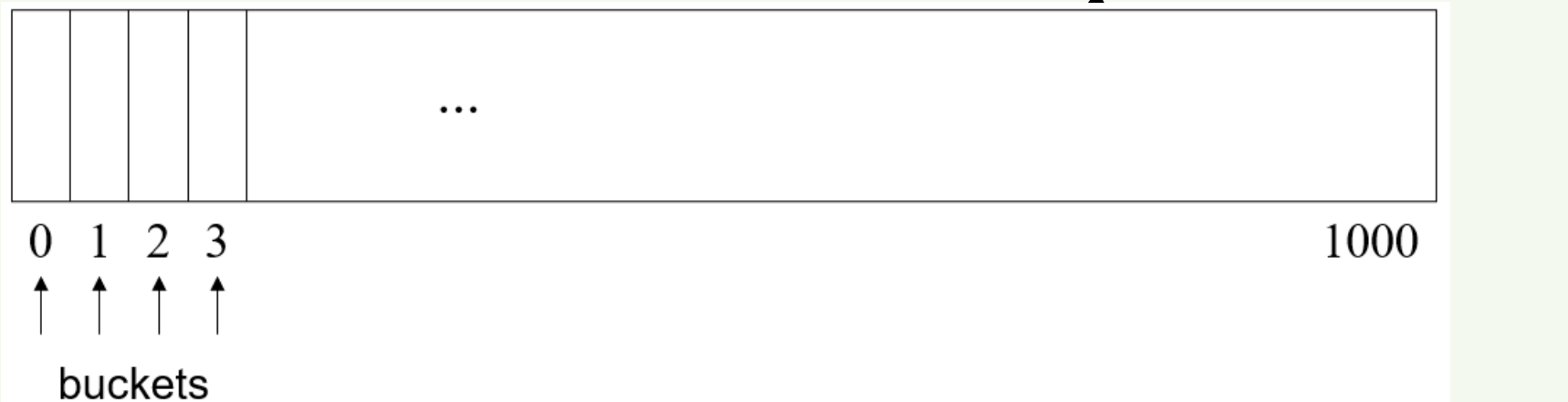
## Ideally

- If element  $e$  has key  $k$  and  $h$  is hash function, then  $e$  is stored in position  $h(k)$  of table
- To search for  $e$ , compute  $h(k)$  to locate position. If no element, dictionary does not contain  $e$ .

# Example

## Dictionary Student Records

- Keys are ID numbers (1000 - 2000), no more than 1001 students
- Hash function:  $h(k) = k - 1000$  maps ID into distinct table positions 0-1000
- array `table[1001]`



# Analysis (Ideal Case)

---

- $O(b)$  time to initialize hash table ( $b$  number of positions or **buckets** in hash table)
- $O(1)$  time to perform `insert`, `remove`, `search`

# Ideal Case is Unrealistic

---

- Works for implementing dictionaries, but many applications have key ranges that are too large to have 1-1 mapping between buckets and keys!

## Example

- Suppose key can take on values from 0 .. 65,535 (2 byte unsigned int)
- Expect  $\approx 1,000$  records at any given time
- Impractical to use hash table with 65,536 slots!



# Hash Functions

---

- If key range too large, use hash table with fewer buckets and a hash function which maps multiple keys to same bucket:

$h(k_1) = \beta = h(k_2)$ :  $k_1$  and  $k_2$  have **collision** at slot  $\beta$

- **Popular hash functions**: hashing by division

$h(k) = k \% D$ , where  $D$  is number of buckets in hash table

- Example: hash table with 11 buckets

$$h(k) = k \% 11$$

80 →

40 →

65 →

58 →

# Hash Functions

---

- If key range too large, use hash table with fewer buckets and a hash function which maps multiple keys to same bucket:

$h(k_1) = \beta = h(k_2)$ :  $k_1$  and  $k_2$  have **collision** at slot  $\beta$

- **Popular hash functions**: hashing by division

$h(k) = k \% D$ , where  $D$  is number of buckets in hash table

- Example: hash table with 11 buckets

$$h(k) = k \% 11$$

$$80 \rightarrow 3 \text{ (} 80 \% 11 = 3 \text{),}$$

$$40 \rightarrow 7,$$

$$65 \rightarrow 10$$

$$58 \rightarrow 3 \text{ **collision!**}$$

# Collision Resolution Policies

---

- Two classes:
  - (1) Open hashing, a.k.a. separate chaining
  - (2) Closed hashing, a.k.a. open addressing
- Difference has to do with whether collisions are stored outside the table (open hashing) or whether collisions result in storing one of the records at another slot in the table (closed hashing)

# Closed Hashing

---

- Associated with closed hashing is a **rehash strategy**:  
“If we try to place  $x$  in bucket  $h(x)$  and find it occupied, find alternative location  $h_1(x)$ ,  $h_2(x)$ , etc. Try each in order, if none empty table is full,”
- $h(x)$  is called **home bucket**
- Simplest rehash strategy is called **linear hashing**  
$$h_i(x) = (h(x) + i) \% D$$

(usually,  $i$  starts from 1 and is increased by 1 every time)
- In general, our collision resolution strategy is to generate a sequence of hash table slots (**probe sequence**) that can hold the record; test each slot until find empty one (**probing**)

# Example Linear (Closed) Hashing

---

- $D=8$ , keys  $a, b, c, d$  have hash values  $h(a)=3$ ,  $h(b)=0$ ,  $h(c)=4$ ,  $h(d)=3$

# Example Linear (Closed) Hashing

---

- $D=8$ , keys  $a, b, c, d$  have hash values  $h(a)=3$ ,  $h(b)=0$ ,  $h(c)=4$ ,  $h(d)=3$

0	b
1	
2	
3	a
4	c
5	
6	
7	

# Example Linear (Closed) Hashing

---

- $D=8$ , keys  $a, b, c, d$  have hash values  $h(a)=3$ ,  $h(b)=0$ ,  $h(c)=4$ ,  $h(d)=3$
- Where do we insert  $d$ ? 3 already filled

0	b
1	
2	
3	a
4	c
5	
6	
7	

# Example Linear (Closed) Hashing

- $D=8$ , keys  $a, b, c, d$  have hash values  $h(a)=3, h(b)=0, h(c)=4, h(d)=3$
- Where do we insert  $d$ ? 3 already filled
- Probe sequence using linear hashing:  
 $h_1(d) = (h(d)+1)\%8 = 4\%8 = 4$   
 $h_2(d) = (h(d)+2)\%8 = 5\%8 = 5^*$   
 $h_3(d) = (h(d)+3)\%8 = 6\%8 = 6$   
etc.  
7, 0, 1, 2
- Wraps around the beginning of the table!

0	b
1	
2	
3	a
4	c
5	
6	
7	



# Example Linear (Closed) Hashing

- $D=8$ , keys  $a, b, c, d$  have hash values  $h(a)=3, h(b)=0, h(c)=4, h(d)=3$
- Where do we insert  $d$ ? 3 already filled
- Probe sequence using linear hashing:  
 $h_1(d) = (h(d)+1)\%8 = 4\%8 = 4$   
 $h_2(d) = (h(d)+2)\%8 = 5\%8 = 5^*$   
 $h_3(d) = (h(d)+3)\%8 = 6\%8 = 6$   
etc.  
7, 0, 1, 2
- Wraps around the beginning of the table!

0	b
1	
2	
3	a
4	c
5	<b>d</b>
6	
7	

# Operations Using Linear Hashing

---

- `findItem`
  - Examine  $h(k)$ ,  $h_1(k)$ ,  $h_2(k)$ , ..., until we find  $k$  or home bucket
- `insert`
  - Examine  $h(k)$ ,  $h_1(k)$ ,  $h_2(k)$ , ..., until we find an empty bucket or home bucket
- `remove`
  - Perform `findItem` and then delete if it is in the hash table
  - May need to reorganize table after many deletions

# Performance Analysis - Worst Case

---

- Initialization:  $O(b)$ ,  $b$  # of buckets
- Insert and search:  $O(n)$ ,  $n$  number of elements in table; all  $n$  key values have same home bucket
- Not better than linear list for maintaining dictionary!

# Improved Collision Resolution

---

- **Linear probing:**  $h_i(x) = (h(x) + i) \% D$ 
  - all buckets in table will be candidates for inserting a new record before the probe sequence returns to home position
  - clustering of records, leads to long probing sequences

# Improved Collision Resolution

---

- **Linear probing:**  $h_i(x) = (h(x) + i) \% D$ 
  - all buckets in table will be candidates for inserting a new record before the probe sequence returns to home position
  - clustering of records, leads to long probing sequences
- **Linear probing with skipping:**  $h_i(x) = (h(x) + ic) \% D$ 
  - $c$  is a constant other than 1
  - records with adjacent home buckets will not follow same probe sequence

# Improved Collision Resolution

---

- **Linear probing:**  $h_i(x) = (h(x) + i) \% D$ 
  - all buckets in table will be candidates for inserting a new record before the probe sequence returns to home position
  - clustering of records, leads to long probing sequences
- **Linear probing with skipping:**  $h_i(x) = (h(x) + ic) \% D$ 
  - $c$  is a constant other than 1
  - records with adjacent home buckets will not follow same probe sequence
- **(Pseudo) Random probing:**  $h_i(x) = (h(x) + r_i) \% D$ 
  - $r_i$  is the  $i^{\text{th}}$  value in a random permutation of numbers from 1 to  $D-1$
  - insertions and searches use the same sequence of “random” numbers

# Example

Hash function:  $h(k) = k \% 11$

- What if the next element has home bucket 0?  
→ go to bucket 3

Same for elements with home bucket 1 or 2!

Only a record with home position 3 will stay.

⇒ probability =  $4/11$  that next record will  
go to bucket 3

- Similarly, records hashing to 7, 8, 9 will end up in  
10

Only records hashing to 4 will end up in position 4  
(probability = ? that next record will go to bucket 4);  
same for 5 and 6

0	1001
1	9537
2	3016
3	
4	
5	
6	
7	9874
8	2009
9	9875
10	

# Example

- insert 1052 (home bucket: 7)

0	1001
1	9537
2	3016
3	
4	
5	
6	
7	9874
8	2009
9	9875
10	



# Example

- insert 1052 (home bucket: 7)

0	1001
1	9537
2	3016
3	
4	
5	
6	
7	9874
8	2009
9	9875
10	1052

# Example

- insert 1052 (home bucket: 7)
- next element in bucket 3 with probability = ?

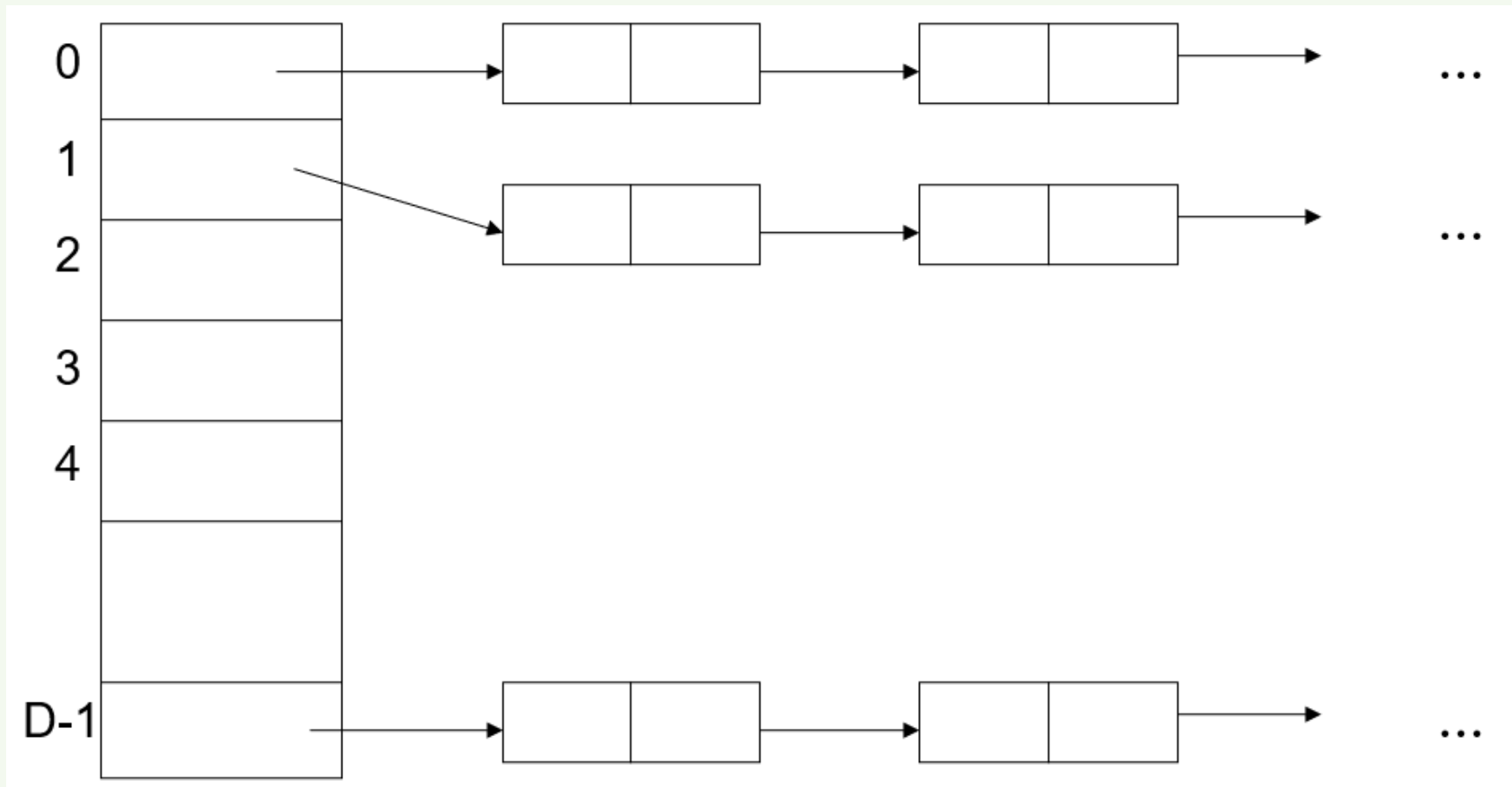
0	1001
1	9537
2	3016
3	
4	
5	
6	
7	9874
8	2009
9	9875
10	1052

# Open Hashing

---

- Each bucket in the hash table is the head of a linked list
- All elements that hash to a particular bucket are placed on that bucket's linked list
- Records within a bucket can be ordered in several ways
  - by order of insertion
  - by key value order
  - by frequency of access order

# Open Hashing Data Organization



# Analysis

---

- Open hashing is more appropriate when the hash table is kept in main memory, implemented with a standard in-memory linked list
- We hope that number of elements per bucket roughly equal in size, so that the lists will be short
- If there are  $n$  elements in set, then each bucket will have roughly  $n/D$
- If we can estimate  $n$  and choose  $D$  to be roughly as large, then the average bucket will have only one or two members

# Analysis

---

## Average time per dictionary operation:

- $D$  buckets,  $n$  elements in dictionary  $\Rightarrow$  average  $n/D$  elements per bucket
- insert, search, remove operation take  $O(1+n/D)$  time each
- If we can choose  $D$  to be about  $n$ , constant time

# Comparison with Closed Hashing

---

- Worst case performance is  $O(n)$  for both
- Number of operations for hashing (how many probes)

23 6 8 10 23 5 12 4 9 19

$D=9$

$h(x) = x \% D$

# Practice

---

- Draw the 11 entry hashtable for hashing the keys 12, 44, 13, 88, 23, 94, 11, 39, 20 using the function  $(2i+5) \bmod 11$  with
  - (1) closed hashing, linear probing
  - (2) open hashing, linked list