

Lecture 9

- Hash Data Structure
 - Collision resolution by:
 - Coalesced chaining
 - Open addressing
 - Other types of hashing
 - Perfect hashing
 - Cuckoo hashig

Lect. PhD. Lupsa Dana
Babes - Bolyai University
Computer Science and Mathematics Faculty
2022 - 2023

Hash. Collisions

- Hash table : a table T of size m
- Hash function: map a key k to a slot in the table T
$$h : U \rightarrow \{0, 1, \dots, m-1\}$$
 - Since m is less than the total number of possible keys: two keys may hash to the same slot \Rightarrow a collision
 - we need techniques for resolving the conflict created by collisions

Collision resolution methods:

- Separate chaining
- Coalesced chaining
- Open addressing

Coalesced chaining

- Collision resolution by coalesced chaining: each element from the hash table is stored inside the table, but each element has a next field, similar to a linked list on array.

What are possible values for α ?

e.g.:

Key	76	12	109	43	22	18	55	81	91	27	13	16	39
Hash	12	12	13	11	6	2	7	1	11	11	13	0	7

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
12	81	18	91	27	13	22	55	16	39		43	76	109		
8	-1	-1	4	-1	-1	-1	9	-1	-1	-1	3	0	5	-1	-1

Representation ?

Coalesced chaining

Add operation

- When a new element has to be inserted and the position where it should be placed is occupied, we will put it to any empty position, and set the next link, so that the element can be found in a search.

e.g.:

- Consider a hash table of size $m = 16$ that uses coalesced chaining for collision resolution and a hash function with the division method
- Insert into the table: 76, 12, 109, 43, 22, 18, 55, 81, 91, 27, 13, 16, 39.

Key	76	12	109	43	22	18	55	81	91	27	13	16	39
Hash	12	12	13	11	6	2	7	1	11	11	13	0	7

Coalesced chaining

subalgorithm insert (ht, k) is:

pos \leftarrow ht.h(k)

if ht.T[pos] = \perp then *// it means empty position*

ht.T[pos] \leftarrow k

ht.next[pos] \leftarrow -1

else

if it exceeds α -threshold then @resize and rehash end-if

firstEmpty \leftarrow *getFirstEmpty*(ht)

current \leftarrow pos

while ht.next[current] \neq -1 execute

current \leftarrow ht.next[current]

end-while

ht.T[ht.firstEmpty] \leftarrow k

ht.next[ht.firstEmpty] \leftarrow - 1

ht.next[current] \leftarrow ht.firstEmpty

end-if

end-subalgorithm

//pre: ht is a HashTable, k is a TKey

//post: k was added into ht

// \perp - notation for the un-valid value

HashTable:

T: TKey[]

next: Integer[]

m: Integer

h: TFunction

Complexity: ?

BC, WC

Coalesced chaining

Think about:

How can we manage the free space (un-occupied positions) ?

- *By default*, in our classes: any element is put on the first unoccupied position, left to right, in the table

How can we implement the remove and search operations ?

- Remove operation for coalesced chaining will be discussed in Sem. 6

How can we define an iterator for a hash table with coalesced chaining?

- init, getCurrent, next, valid

Open addressing

elements are stored directly within the array

→ no next links

Collisions (solutions)

- linear probing
the interval between probes is fixed - often at 1.
- quadratic probing
the interval between probes increases proportional to the hash value
- double hashing
the interval between probes is computed by another hash function

probing: search through alternate locations in the table
(the probe sequence)

Open addressing: linear probing

Given a primary hash function $h': U \rightarrow \{0, 1, \dots, m - 1\}$

hash function for open addressing with linear probing is defined as:

$$h : U \times \{0, 1, \dots, m-1\} \rightarrow \{0, 1, \dots, m-1\}$$

$$h(k,i) = (h'(k) + i) \bmod m$$

Slot probed: $T[h'(k)]$, $T[h'(k) + 1]$, ... $T[m - 1]$,
 $T[0]$, $T[1]$, ... , $T[h'(k) - 1]$.

Problem : *primary clustering*

long runs of occupied slots build up, increasing the average search time.

Open addressing: linear probing

Think about :

- Consider a hash table of size $m = 16$ that uses open addressing and linear probing for collision resolution
- Insert into the table the following elements:

76, 12, 109, 43, 22, 18, 55, 81, 91, 27, 13, 16, 39...

Key	76	12	109	43	22	18	55	81	91	27	13	16	39
Hash	12	12	13	11	6	2	7	1	11	11	13	0	7

Assume m positions, n elements and $\alpha = 0.5$ ($n = m / 2$)

Arrangement 1: every second position is empty

(for example: even positions are occupied and odd ones are free)

Arrangement 2: all n elements are one after the other

(assume in the second half of the array)

What is number of probes (positions verified) that need to be checked to insert a new element?

Open addressing: linear probing

Disadvantages of linear probing:

- Once you have the starting position everything is fixed
- Primary clustering - long runs of occupied slots

Advantages of linear probing:

- Probe sequence is always a permutation

Open addressing: quadratic probing

Given a primary hash function $h': U \rightarrow \{0, 1, \dots, m - 1\}$

$$h(k,i) = (h'(k) + c1*i + c2*i^2) \bmod m$$

$c1$ and $c2 \neq 0$ are auxiliary constants,
and $i = 0, 1, \dots, m - 1$.

Problem: *secondary clustering*

if two keys have the same initial probe position,
then their probe sequences are the same:

$$h(k1, 0) = h(k2, 0) \Rightarrow h(k1, i) = h(k2, i).$$

Also, the performance is sensitive to the values of m , $c1$ and $c2$.

Quadratic probing: choosing h

- One important issue with quadratic probing is how we can choose the values of m , c_1 and c_2 so that the probe sequence is a permutation.
- For example, for $m = 11$, $c_1 = 1$, $c_2 = 1$ and $k = 27$, the probe sequence is
 $\langle 5, 7, 0, 6, 3, 2, 3, 6, 0, 7, 5 \rangle$
- If m is a power of 2 and $c_1 = c_2 = 0.5$, the probe sequence will always be a permutation.

For example for $m = 8$ and $k = 3$:

- $h(3, 0) = ((3 \bmod 8) + 0.5 * 0 + 0.5 * 0^2) \bmod 8 = 3$
- $h(3, 1) = ((3 \bmod 8) + 0.5 * 1 + 0.5 * 1^2) \bmod 8 = 4$
- $h(3, 2) = \dots = 6$
- $h(3, 3) = \dots = 1$
- $h(3, 4) = \dots = 5$
- $h(3, 5) = \dots = 2$
- $h(3, 6) = \dots = 0$
- $h(3, 7) = \dots = 7$

Open addressing: double hashing

Given a primary hash function $h1: U \rightarrow \{0, 1, \dots, m - 1\}$
and a secondary hash functions $h2: U \rightarrow \{0, 1, \dots, m - 1\}$

$$h(k,i) = (h1(k) + i * h2(k)) \bmod m$$

Remark:

one of the best methods for open addressing

Main advantage of double hashing is that

even if $h(k1, 0) = h(k2, 0)$,

the probe sequence will be different if $k1 \neq k2$

Double hashing: choosing h_1 and h_2

$$h(k, i) = (h_1(k) + i * h_2(k)) \bmod m,$$

- if m and $h_2(k)$ have greatest common divisor $d > 1$ for some key k , then a search for key k would examine only $(1/d)$ -th of the hash table.

$h_2(k)$ - relatively prime to the hash-table size m

Convenient ways to ensure this condition:

- m - a power of 2
design h_2 so that it always produces an odd number
- m - prime
design h_2 so that it always returns a positive integer less than m .

Example:

if m prime

$$h_1(k) = k \bmod m ,$$

$$h_2(k) = 1 + (k \bmod m'),$$

where m' slightly less than m (say, $m - 1$ or $m - 2$).

Open addressing - review

- In case of open addressing every element of the hash table is inside the table
- When we want to insert a new element, we will successively generate positions for the element, check (probe) the generated position, and place the element in the first available one.
- In order to generate multiple positions, we will extend the hash function and add to it another parameter, i , which is the probe number and starts from 0.

$$h : U \times \{ 0, 1, \dots, m-1 \} \rightarrow \{ 0, 1, \dots, m-1 \}$$

- For an element k , we will successively examine the positions
 $\langle h(k, 0), h(k, 1), h(k, 2), \dots, h(k, m-1) \rangle$ - called the probe sequence
- The probe sequence : is important to be a permutation of a hash table positions
 $\{ 0, 1, \dots, m-1 \}$, so that eventually every slot is considered.

Open addressing - insert

subalgorithm insert (ht, e) is:

$i \leftarrow 0$

$\text{pos} \leftarrow \text{ht.h}(e, i)$

while $i < \text{ht.m}$ and $\text{ht.T}[\text{pos}] \neq \perp$ execute

$i \leftarrow i + 1$

$\text{pos} \leftarrow \text{ht.h}(e, i)$

end-while

if $i = \text{ht.m}$ then

@resize and rehash

else

$\text{ht.T}[\text{pos}] \leftarrow e$

end-if

end-subalgorithm

What should the search operation do?

How can we remove an element from the hash table?

Open addressing

- How can we implement operation *search*?
- How can we *remove* an element from the hash table?

Removing an element from a hash table with open addressing is not simple. When removing an element, we should do it in such a way that it does not affect the search (otherwise it might not find other elements)

- Remove is usually implemented to mark the deleted position with a special value, DELETED.

Then:

- modify SEARCH
- modify INSERT

How ... ?

Performance : collision resolution by open addressing

Performance :

under the assumption of simple uniform hashing and
constant α , the average complexity for operations: $\Theta(1)$

Theorem:

under the assumption of uniform hashing

with load factor $\alpha = n/m < 1$

the expected number of probes

- in an unsuccessful search is at most $1/(1 - \alpha)$,
- for operation add is at most $1/(1 - \alpha)$
- in a successful search is at most $1/\alpha * \log(1/(1 - \alpha))$

(Cormen)

Other types of hashing

Perfect hashing

= No collisions

Assume: We have a static collection.

Idea: Use a two-level hashing scheme and universal hashing.

- Primary table: use $m=n$ positions
- Use a secondary hash table for each position i :
 - It has dimension n_i^2 , where n_i is the number of elements (collisions) associated to position i
 - Choose h_i to ensure that no collisions occurs

How do we find the hash functions ?

Perfect hashing

e.g.:

- 15 letters: I, N, S, X, E,....
- $m=15$
- *hashCode* : letter \rightarrow index in the alphabet

Use an universal set of hash function

Choose: $H_{a,b}(x) = ((a * x + b) \bmod p) \bmod m$
 $p=29, a=3, b=2$ (chosen randomly)

	I	N	S	X	E
<i>hashCode</i>	9	14	19	24	5
<i>h</i> (hashCode)	0	0	1	1	2

...

$a = 4$
 $b = 11$

$a = 5$
 $b = 2$

$a = 2$
 $b = 13$

...

Cuckoo hashing

In cuckoo hashing we have two hash tables of the same size and each hash table has its hash function.

- For each element there are exactly two positions where it can appear: one in the first hash table or the one in the second.
 - It is guaranteed that an element will be on one of these positions!
- **Search** is simple, because we only have to look at these two positions.
- **Delete** is simple, because we only have to look at these two positions and set to empty the one where we find the element.

e.g.:

$$h1(k) = k \bmod 11$$

m=11

Position	0	1	2	3	4	5	6	7	8	9	10
T		100		36			50			75	

Position	0	1	2	3	4	5	6	7	8	9	10
T	3	20		39	53		67			105	

$$h2(k) = (k \div 11) \bmod 11$$

Cuckoo hashing

Add an element:

- When we want to insert a new element we will compute its position in the first hash table.
- If the position is empty, we will place the element there.
- If the position in the first hash table is not empty, we will kick out the element that is currently there, and place the new element into the first hash table.
 - The element that was kicked off, will be placed at its position in the second hash table.
 - If that position is occupied, we will kick off the element from there and place it into its position in the first hash table.

We repeat the above process until we will get an empty position for an element.

If we get back to the same location with the same key we have a cycle and we cannot add this element

➔ resize, rehash

Cuckoo hashing

Think about:

Assume that we have two hash tables, with $m = 11$ positions and the following hash functions:

- $h_1(k) = k \bmod 11$
- $h_2(k) = (k \operatorname{div} 11) \bmod 11$

Insert: 20, 50, 53, 75, 100, 67*, 105, ...

Cuckoo hashing

Remark:

- Cuckoo hashing's performance is highly influenced by the load factor.
- While in some situation insert moves a lot of elements, it can be shown that if the load factor of the tables is below 0.5, the probability of a cycles is low and it is very unlikely that more than $O(\log_2 n)$ elements will be moved.
=> works well with a load factor below 0.5

Variations:

Cuckoo hashing with $k \geq 3$ tables

- It tends to perform much better than cuckoo hashing with $k = 2$ tables
- If we use three tables, the tables can have load factor of 0.91 and for 4 tables we have 0.97