Lecture 9

- Hash Data Structure
  - Collision resolution by:
    - Coalesced chaining
    - Open addressing
  - Hash in programming languages. Examples
  - Perfect hashing

Lect. PhD. Lupsa Dana
Babes - Bolyai University
Computer Science and Mathematics Faculty
2021 - 2022

Some slides borrowed from: Lect. PhD. Onet-Marian Zsuzsanna

Hash. Collisions

- Hash table : a table T of size m
- Hash function: map a key k to a slot in the table T

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

  - Since m is less than the total number of possible keys: two keys may hash to the same slot => 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

- 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 ← ht.h(k)
   if ht.T[pos] = ⊥ then // it means empty position
            ht.T[pos] ← k
            ht.next[pos] ← -1
   else
            if it exceeds α then    @resize and rehash    end-if
            firstEmpty ← getFirstEmpty(ht)
            current ← pos
            while ht.next[current] ≠ -1 execute
                     current ← ht.next[current]
            end-while
            ht.T[ht.firstEmpty] ← k
            ht.next[ht.firstEmpty] ← - 1
            ht.next[current] ← ht.firstEmpty
   end-if
end-subalgorithm
```

//pre: *ht is a HashTable, k is a TKey*
//post: *k was added into ht*

// ⊥ - *notation for an un-valid value*

HashTable:
T: TKey[]
next: Integer[]
m: Integer
h: TFunction

Complexity: **?**
BC, WC, AC

Coalesced chaining

Think about:

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

How can we implement the remove and search operations ?

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': $U \rightarrow \{0, 1, \ldots, m - 1\}$

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

h : U x { 0, 1, …, m-1 } → { 0, 1, …, 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 the average 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': $U \rightarrow \{0, 1, \ldots, m - 1\}$

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

$c1$ and $c2 <> 0$ are auxiliary constants,

and $i = 0, 1, \ldots, m - 1$.

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, c1 and c2 so that the probe sequence is a permutation.
- For example, for m = 11, c1 = 1, c2 = 1 and k = 27, the probe sequence is

  < 5, 7, 0, 6, 3, 2, 3, 6, 0, 7, 5 >
- If m is a power of 2 and c1 = c2 = 0.5, the probe sequence will always be a permutation.

  For example for m = 8 and k = 3:
  - h(3, 0) = ((3 mod 8) + 0.5 * 0 + 0.5 * 0^2) mod 8 = 3
  - h(3, 1) = ((3 mod 8) + 0.5 * 1 + 0.5 * 1^2) mod 8 = 4
  - h(3, 2) = ... = 6
  - h(3, 3) = ... = 1
  - h(3, 4) = ... = 5
  - h(3, 5) = ... = 2
  - h(3, 6) = ... = 0
  - h(3, 7) = ... = 7

Open addressing: double hashing

Given a primary hash function $h1: U \rightarrow \{0, 1, \ldots, m - 1\}$

and a secondary hash functions $h2: U \rightarrow \{0, 1, \ldots, m - 1\}$

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

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 h1 and h2

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

- if m and $h2(k)$ 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.

  $h2(k)$ - relatively prime to the hash-table size m

Convenient ways to ensure this condition:

- m - a power of 2
  design $h2$ so that it always produces an odd number
- m - prime
  design $h2$ so that it always returns a positive integer less than m.

Example:

if m prime

$h1(k) = k \bmod m$,

$h2(k) = 1 + (k \bmod m')$,

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.

  $h : U \times \{ 0, 1, \ldots, m-1 \} \rightarrow \{ 0, 1, \ldots, m-1 \}$
- For an element k, we will successively examine the positions

  $< h(k, 0), h(k, 1), h(k, 2) , \ldots , h(k, m - 1) >$ - called the probe sequence
- The probe sequence : is important to be a permutation of a hash table positions

  $\{ 0, 1, \ldots, m-1 \}$, so that eventually every slot is considered.

Open addressing - insert

```
subalgorithm insert (ht, e) is:
   i ← 0
   pos ← ht.h(e, i)
   while i < ht.m and ht.T[pos] ≠ ⊥ execute
         i ← i + 1
         pos ← ht.h(e, i)
   end-while
   if i = ht.m then
         @resize and rehash
   else
         ht.T[pos] ← 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*)

Hash table in programming languages

- Hash tables are an effective way to implement containers if we want to achieve good performance for **search**, **insert** and **delete**

JAVA Util

- HashSet
- HashMap

C++ STL

- unordered_set
- unordered_map

They use hash table with separate chaining.

Hash table in programming languages

Java.util

HashSet

- implements the Set interface; uses a hash table

HashMap

- implements the Map interface; uses a hash table

- HashMap()

  Constructs an empty HashMap with the default initial capacity (16)
  and the default load factor (0.75).

- HashMap(int initialCapacity)

  Constructs an empty HashMap with the specified initial capacity and the default load factor (0.75).

- HashMap(int initialCapacity, float loadFactor)

  Constructs an empty HashMap with the specified initial capacity and load factor.

https://docs.oracle.com/javase/8/docs/api/index.html

Hash in programming languages. Examples

Java Object

- public int hashCode()

  As much as is reasonably practical, the hashCode method defined by class Object does return distinct integers for distinct objects. (This is typically implemented by converting the internal address of the object into an integer, but this implementation technique is not required by the JavaTM programming language.)

- public boolean equals(Object obj)

  If two objects are equal then they must return same hash code

  (when they are compared by equals())

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 $\mathbf{h_i}$ 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 -> 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
hashCode	9	14	19	24	5
h *(hashCode)*	0	0	1	1	2

…

a = 4
b =11

a = 5
b = 2

a = 2
b = 13

…