DATA STRUCTURES

Hash tables: Coalesced chaining, Open adressing

Lect. PhD. Diana-Lucia Miholca

2022 - 2023

Babeş - Bolyai University
Faculty of Mathematics and Computer Science

In previous lecture...

- Hash tables
  - Direct-address table
  - Introduction to hash tables
  - Separate chaining

Today

- Coalesced chaining
- Open addressing

Coalesced chaining

D **Collision resolution by coalesced chaining**: each element is stored inside the table and has associated the index of the *next* element.

Coalesced chaining

Collision resolution by coalesced chaining: each element is stored inside the table and has associated the index of the *next* element.

When **adding a new element** and the index it hashes to is occupied:

- the element is added at any empty index and
- the *next* indexes are set so as to find it starting from its hashing index

Coalesced chaining

Collision resolution by coalesced chaining: each element is stored inside the table and has associated the index of the *next* element.

When **adding a new element** and the index it hashes to is occupied:

- the element is added at any empty index and
- the *next* indexes are set so as to find it starting from its hashing index

Since elements are in the table, α can be at most 1.

Coalesced chaining - example

Consider a hash table that uses coalesced chaining for collision resolution, with:

- m = 16
- the division method for hashing

Insert in the hash table, in the given order, the following elements: 76, 12, 109, 43, 22, 18, 55, 81, 91, 27, 13, 16, 39.

What is the hash value for 12? What is the hash value for 16?

Coalesced chaining - example

Consider a hash table that uses coalesced chaining for collision resolution, with:

- m = 16
- the division method for hashing

Insert in the hash table, in the given order, the following elements: 76, 12, 109, 43, 22, 18, 55, 81, 91, 27, 13, 16, 39.

What is the hash value for 12? What is the hash value for 16?

Let's compute the hash value for every element (key):

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

Example

- Initially, the hash table is empty
  - The first empty position is 0 and all *next* indexes are -1.

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1

firstEmpty = 0

- 76 is added at index 12
- 12 should also be added at index 12. Since it is already occupied, we add 12 at index *firstEmpty* (0) and set the *next* of 76 to 0. Then we reset *firstEmpty* to the next empty position.

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
12												76			
-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	0	-1	-1	-1

firstEmpty = 1

Example

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

- We continue similarly, having no collisions up to 81.
  - We need to update *firstEmpty* every time we occupy it.

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

firstEmpty = 3

- When adding 91, we add it at index *firstEmpty* and set the *next* link at index 11 to 3.

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

firstEmpty = 4

Example

- The final hash table:

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

firstEmpty = 10

Coalesced chaining - representation

- A hash table with coalesced chaining is represented in the following way:

Representation of a hash table with coalesced chaining:

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

For simplicity, in the following, we will consider only the keys.

Coalesced chaining - insert

Adding a key to a hash table with coalesced chaining:

subalgorithm insert (ht, k) **is:**

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

//post: k was added into ht

Coalesced chaining - insert

Adding a key to a hash table with coalesced chaining:

```
subalgorithm insert (ht, k) is:
//pre: ht is a HashTable, k is a TKey
//post: k was added into ht
    index ← ht.h(k)
    if ht.T[index] = NULL_TKey then //NULL_TKey means empty position
        ht.T[index] ← k
        if index = ht.firstEmpty then
            changeFirstEmpty()
        end-if
    else
        if ht.firstEmpty = ht.m then
            @resize and rehash
        end-if
        ht.T[ht.firstEmpty] ← k
//continued on the next slide...
```

Coalesced chaining - insert

Adding a key to a hash table with coalesced chaining:

```
        current ← index
        while ht.next[current] ≠ -1 execute
            current ← ht.next[current]
        end-while
        ht.next[current] ← ht.firstEmpty
        ht.next[ht.firstEmpty] ← - 1
        changeFirstEmpty(ht)
    end-if
end-subalgorithm
```

Complexity:

Coalesced chaining - insert

Adding a key to a hash table with coalesced chaining:

current $\leftarrow$ index
while ht.next[current] $\neq$ -1 **execute**
current $\leftarrow$ ht.next[current]
end-while
ht.next[current] $\leftarrow$ ht.firstEmpty
ht.next[ht.firstEmpty] $\leftarrow$ - 1
changeFirstEmpty(ht)
end-if
end-subalgorithm

Complexity: $\Theta(1)$ on average (under SUH assumption), $\Theta(n)$ - in the worst case

Coalesced chaining - ChangeFirstEmpty

Updating the first empty position in a hash table with coalesced chaining:

subalgorithm changeFirstEmpty(ht) **is:**
//pre: ht is a HashTable
//post: the value of ht.firstEmpty is set to the next free position
    ht.firstEmpty $\leftarrow$ ht.firstEmpty + 1
    while ht.firstEmpty $<$ ht.m **and** ht.T[ht.firstEmpty] $\neq$ $NULL_{TKey}$
execute
        ht.firstEmpty $\leftarrow$ ht.firstEmpty + 1
    end-while
end-subalgorithm

Complexity:

Coalesced chaining - ChangeFirstEmpty

Updating the first empty position in a hash table with coalesced chaining:

subalgorithm changeFirstEmpty(ht) **is:**
//pre: ht is a HashTable
//post: the value of ht.firstEmpty is set to the next free position
    ht.firstEmpty $\leftarrow$ ht.firstEmpty + 1
    while ht.firstEmpty $<$ ht.m **and** ht.T[ht.firstEmpty] $\neq$ $NULL_{TKey}$ **execute**
        ht.firstEmpty $\leftarrow$ ht.firstEmpty + 1
    end-while
end-subalgorithm

Complexity: $O(n)$

Coalesced chaining

🤝 *Remove* and *search* will be discussed in Seminar 5.

Open addressing

Collision resolution by open addressing: each element is stored inside table and there are no links.

Open addressing

Collision resolution by open addressing: each element is stored inside table and there are no links.

When adding a new element, we will:

- successively generate candidate positions
- check (*probe*) their availability and
- place the element in the first available one

Open addressing

In order to generate multiple positions, the hash function is extended with an additional parameter, i, which is the *probe number* and starts from 0.

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

For an element k, positions from the *probe sequence* $< h(k, 0), h(k, 1), h(k, 2), ..., h(k, m-1) >$ will be successively examined.

The *probe sequence* should be a permutation of $\{0, ..., m-1\}$, so that eventually every slot is probed.

Open addressing - Linear probing

A first scheme for defining the hash function is to use **linear probing**:

$$h(k, i) = (h'(k) + i) \ mod \ m \ \ \forall i = 0, ..., m - 1$$

- where $h'(k)$ is a *simple* hash function
  - For example: $h'(k) = k \ mod \ m$

The *probe sequence* for linear probing is:
$< h'(k), h'(k) + 1, h'(k) + 2, ..., m - 1, 0, 1, ..., h'(k) - 1 >$

Open addressing - Linear probing - example

Consider a hash table that uses open addressing for collision resolution, with:

- m = 16
- linear probing with $h'(k) = k \ mod \ m$

Insert into the table, in the given order, the following elements: 76, 12, 109, 43, 22, 18, 55, 81, 91, 27, 13, 16, 39.

Let's compute the value of the hash function for every element (key) when i = 0:

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

Open addressing - Linear probing - example

- The final hash table:

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

Open addressing - Linear probing - example

- The final hash table:

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

Disadvantages of linear probing:

- *Primary clustering* - long runs of occupied slots
- There are only *m* distinct probe sequences (once you have the starting position everything is fixed)

Advantages of linear probing:

- Probe sequence is always a permutation
- Can benefit from caching

Open addressing - Quadratic probing

In case of **quadratic probing** the hash function becomes:

$$h(k,i) = (h'(k) + c_1 * i + c_2 * i^2)\ mod\ m\ \ \forall i = 0, ..., m-1$$

- where $h'(k)$ is a *simple* hash function
  - for example: $h'(k) = k\ mod\ m$ and c_1 and c_2 are constants; c_2 should not be 0

Considering a simplified version of $h(k,i)$ with $c_1 = 0$ and $c_2 = 1$ the probe sequence would be:
$< h'(k), h'(k)+1, h'(k)+4, h'(k)+9, h'(k)+16, ... >$

Open addressing - Quadratic probing

The values of m, c_1 and c_2 should be chosen so that the probe sequence is a permutation.

If m is a prime number only the first half of the probe sequence is unique $\Rightarrow$ once the hash table is half full, there is no guarantee that an empty index will be found.

For example, for $m = 17$, $c_1 = 3$, $c_2 = 1$ and $k = 13$, the probe sequence is
$<$ **13, 0, 6, 14, 7, 2, 16, 15**, 16, 2, 7, 14, 6, 0, 13, 11, 11 $>$

Open addressing - Quadratic probing

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 $k = 3$:

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

Open addressing - Quadratic probing

If m is a prime number of the form $4 * j + 3$, $c_1 = 0$ and $c_2 = (-1)^i$ (so the probe sequence is +0, -1, +4, -9, etc.) the probe sequence is a permutation.

For example for $m = 7$ and $k = 3$:

- $h(3, 0) = (3 \% 7 + 0^2) \% 7 = 3$
- $h(3, 1) = (3 \% 7 - 1^2) \% 7 = 2$
- $h(3, 2) = (3 \% 7 + 2^2) \% 7 = 0$
- $h(3, 3) = (3 \% 7 - 3^2) \% 7 = 1$
- $h(3, 4) = (3 \% 7 + 4^2) \% 7 = 5$
- $h(3, 5) = (3 \% 7 - 5^2) \% 7 = 6$
- $h(3, 6) = (3 \% 7 + 6^2) \% 7 = 4$

Open addressing - Quadratic probing - Example

Consider a hash table that uses open addressing for collision resolution, with:

- $m = 16$
- quadratic probing with $h'(k) = k \ mod \ m$ and $c_1 = c_2 = 0.5$.

Insert into the table, in the given order, the following elements: 76, 12, 109, 43, 22, 18, 55, 81, 91, 27, 13, 16, 39.

Open addressing - Quadratic probing - example

- The final hash table:

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

Open addressing - Quadratic probing - example

- The final hash table:

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

! Disadvantages of quadratic probing:

- *Secondary clustering* - if two elements have the same initial probe positions, their whole probe sequence will be identical: $h(k_1, 0) = h(k_2, 0) \Rightarrow h(k_1, i) = h(k_2, i)$.
- There are only m distinct probe sequences
- The performance is sensitive to the values of m, c_1 and c_2.

Open addressing - Double hashing

Double hashing uses a hash function of form:

$$h(k, i) = (h'(k) + i * h''(k)) \ \% \ m \ \forall i = 0, ..., m - 1$$

- where $h'(k)$ and $h''(k)$ are *simple* hash functions; $h''(k)$ should never return 0.

For a key, k, the initial probe goes to position $h'(k)$ and the successive probe positions are offset from previous positions by the amount $h''(k)$, modulo m.

Open addressing - Double hashing

Similar to quadratic probing, not every combination of m and h'' will produce a complete permutation.

$h''(k)$ must be relatively prime to m. This can be ensured by:

- Choosing m as a power of 2 and designing h'' in such a way that it always returns an odd number.
- Choosing m as a prime number and designing h'' in such a way that it always returns a value from the {1, m-1}.

Open addressing - Double hashing

For example:

$$h'(k) = k\%m$$

$$h''(k) = 1 + k\%(m-1)$$

- For $m = 11$ and $k = 36$ we have:
  $h'(36) = 3$
  $h''(36) = 7$
- The probe sequence is: $< 3, 10, 6, 2, 9, 5, 1, 8, 4, 0, 7 >$

Open addressing - Double hashing - example

Consider a hash table that uses open addressing with double hashing for collision resolution, with:

- $m = 17$
- $h'(k) = k\%m$ and $h''(k) = 1 + (k\%16)$

Insert into the table, in the given order, the following elements: 75, 12, 109, 43, 22, 18, 55, 81, 92, 27, 13, 16, 39.

Values of the two hash functions for each element:

key	75	12	109	43	22	18	55	81	92	27	13	16	39
h' (key)	7	12	7	9	5	1	4	13	7	10	13	16	5
h''(key)	12	13	14	12	7	3	8	2	13	12	14	1	8

Open addressing - Double hashing - example

- The final hash table:

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
16	18		55	109	22		75		43	27	39	12	81		13	92

The main advantage of double hashing is that even if $h(k_1, 0) = h(k_2, 0)$ the probe sequences will be different if $k_1 \neq k_2$.

For example:

- 75: $< 7, 2, 14, 9, 4, 16, 11, 6, 1, 13, 8, 3, 15, 10, 5, 0, 12 >$
- 109: $< 7, 4, 1, 15, 12, 9, 6, 3, 0, 14, 11, 8, 5, 2, 16, 13, 10 >$

Since for every $(h'(k), h''(k))$ pair we have a separate probe sequence, double hashing generates $\approx m^2$ different permutations.

Open addressing - representation

- A hash table with open addressing for collision resolution is represented in the following way:

Representation of a hash table with open addressing:

HashTable:

    T: TKey[]
    m: Integer
    h: TFunction

For simplicity, in the following, we will consider only the keys.

Open addressing - insert

Adding an element

subalgorithm insert (ht, e) **is:**
//pre: ht is a HashTable, e is a TKey
//post: e was added in ht

Open addressing - insert

Adding an element

subalgorithm insert (ht, e) **is:**
//pre: ht is a HashTable, e is a TKey
//post: e was added in ht
    i ← 0
    pos ← ht.h(e, i)
    while $i < ht.m$ **and** ht.T[pos] $\neq NULL_{TKey}$ **execute**
    //$NULL_{TKey}$ means empty space
        i ← i + 1
        pos ← ht.h(e, i)
    end-while
    if $i = ht.m$ **then**
        @resize and rehash and compute the position for e again
    else
        ht.T[pos] ← e
    end-if
end-subalgorithm

Open addressing - searching

What should the *search* operation do?

Open addressing - remove

How can we *remove* an element from the hash table?

Open addressing - remove

How can we *remove* an element from the hash table?

We cannot just mark the position empty (by storing $NULL_{TKey}$ into it) - *search* might not find other elements

Open addressing - remove

How can we *remove* an element from the hash table?

We cannot just mark the position empty (by storing $NULL_{TKey}$ into it) - *search* might not find other elements

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

Open addressing - remove

How can we *remove* an element from the hash table?

We cannot just mark the position empty (by storing $NULL_{TKey}$ into it) - *search* might not find other elements

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

How does the usage of the value DELETED affect the implementation of the *insert* and *search* operation?

Open addressing - Performance

Theorem: In a hash table with open addressing with load factor $\alpha = n/m$ ($\alpha < 1$), the *average* number of probes is at most

- for *insert* and *unsuccessful search*

$$\frac{1}{1-\alpha}$$

- for *successful search*

$$\frac{1}{\alpha} * \ln \frac{1}{1-\alpha}$$

If α is constant, the average complexity is $\Theta(1)$

Worst case complexity is $\Theta(n)$

Containers represented using hash tables

Hash tables are used for representing the following containers:

- ADT Map (Sorted Map)
  - Python's dictionaries ({:}), Java HashMap, unordered_map in C++ STL
- ADT MultiMap (Sorted MultiMap)
  - HashMultimap in Guava (Google Core Libraries for Java) unordered_multimap in C++ STL
- ADT Set
  - HashSet in Java Collections API, Python's sets ({})
- ADT Bag
  - HashMultiset in Guava (for Java)

Hash table - Applications

Real-word applications of hash tables:

Programming languages

- Implementation of built-in data types (*dict* in Python, *HashMap* in Java)

Compilers

- For storing the programming language's keywords and for mapping the variables names with memory locations

File system

- For mapping file names to the the file path and to the physical location of that file on the disk

Password Verification:

- For storing hashed passwords

Data Integrity Checks

- To generate checksums on data files

Bibliography

- David M. Mount, *Lecture notes for the course Data Structures* (CMSC 420), at the Dept. of Computer Science, University of Maryland, College Park, 2001
- Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein, *Introduction to Algorithms*, Third Edition, The MIT Press, 2009
- Narasimha Karumanchi, *Data Structures and Algorithms Made Easy: Data Structures and Algorithmic Puzzles*, Fifth Edition, 2016
- Clifford A. Shaffer, *A Practical Introduction to Data Structures and Algorithm Analysis*, Third Edition, 2010

Thank you