DATA STRUCTURES AND ALGORITHMS
LECTURE 11

Lect. PhD. Marian Zsuzsanna

Babeș - Bolyai University
Computer Science and Mathematics Faculty

2017 - 2018

In Lecture 9-10...

- ADT Stack
- ADT Queue
- ADT Deque
- ADT Priority Queue
- Hash tables

Today

Hash table - reminder

- Hash tables are tables (arrays) - one consecutive memory zone - but it is not filled with elements from left to right as regular arrays are.
- For every element to be added a unique position is generated in the table using a *hash function*.
- This position is also generated for the element when it has to be removed or when we search for it.
- When the hash function generates the same position for two distinct elements we have a *collision* and we need a method to resolve it.

Hash table - reminder II

- In case of *separate chaining* at each position from the table we have a linked list.
- An important value for hash tables is the *load factor*, α, which is computed as: n/m.
- For separate chaining α can be larger than 1.

Example of separate chaining

- Consider a hash table of size $m = 11$ that uses separate chaining for collision resolution and a hash function with the division method
- Insert into the table the letters from *A SEARCHING EXAMPLE* (space is ignored)
- For each letter, the *hashCode* is the index of the letter in the alphabet.

Letter	A	S	E	R	C	H	I	N	G	X	M	P	L
HashCode	1	19	5	18	3	8	9	14	7	24	13	16	12

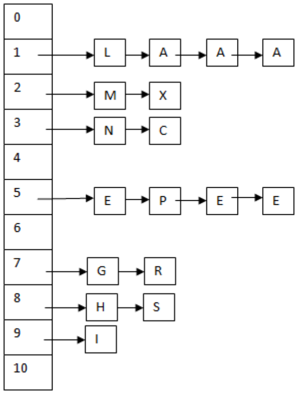
Example of separate chaining

- Consider a hash table of size $m = 11$ that uses separate chaining for collision resolution and a hash function with the division method
- Insert into the table the letters from *A SEARCHING EXAMPLE* (space is ignored)
- For each letter, the *hashCode* is the index of the letter in the alphabet.

Letter	A	S	E	R	C	H	I	N	G	X	M	P	L
HashCode	1	19	5	18	3	8	9	14	7	24	13	16	12
h(Letter)	1	8	5	7	3	8	9	3	7	2	2	5	1

Example of separate chaining

- After the letters were inserted in an empty hash table:



- Load factor α: $17/11 = 1.54$ - the average length of a list

Iterator

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

Iterator

- How can we define an iterator for a hash table with separate chaining?
- Since the order of the elements is not important, our iterator can iterate through them in any order.
- For the hash table from the previous slide, the easiest order in which the elements can be iterated is: LAAAMXNCEPEEGRHSI

Iterator

- Iterator for a hash table with separate chaining is a combination of an iterator on an array (table) and on a linked list.
- We need a current position to know the position from the table that we are at, but we also need a current node to know the exact node from the linked list from that position.

```
IteratorHT:
    ht: HashTable
    currentPos: Integer
    currentNode: ↑ Node
```

Iterator - init

- How can we implement the *init* operation?

Iterator - init

- How can we implement the *init* operation?

```
subalgorithm init(ith, ht) is:
//pre: ith is an IteratorHT, ht is a HashTable
    ith.ht ← ht
    ith.currentPos ← 0
    while ith.currentPos < ht.m and ht.T[ith.currentPos] = NIL execute
        ith.currentPos ← ith.currentPos + 1
    end-while
    if ith.currentPos < ht.m then
        ith.currentNode ← ht.T[ith.currentPos]
    else
        ith.currentNode ← NIL
    end-if
end-subalgorithm
```

- Complexity of the algorithm: $O(m)$

Iterator - getCurrent

- How can we implement the *getCurrent* operation?

Iterator - getCurrent

- How can we implement the *getCurrent* operation?

```
**subalgorithm** getCurrent(ith, elem) **is**:
    elem $\leftarrow$ [ith.currentNode].key
**end-subalgorithm**
```

- Complexity of the algorithm: $\Theta(1)$

Iterator - next

- How can we implement the *next* operation?

Iterator - next

- How can we implement the *next* operation?

```
subalgorithm next(ith) is:
    if [ith.currentNode].next $\neq$ NIL then
        ith.currentNode $\leftarrow$ [iht.currentNode].next
    else
        ith.currentPos $\leftarrow$ ith.currentPos + 1
        while ith.currentPos $<$ ith.ht.m and ith.ht.T[ith.currentPos]=NIL ex.
            ith.currentPos $\leftarrow$ ith.currentPos + 1
        end-while
        if ith.currentPos $<$ it.ht.m then
            ith.currentNode $\leftarrow$ ith.ht.T[ith.currentPos]
        else
            ith.currentNode $\leftarrow$ NIL
        end-if
    end-if
end-subalgorithm
```

- Complexity of the algorithm: $O(m)$

Iterator - valid

- How can we implement the *valid* operation?

Iterator - valid

- How can we implement the *valid* operation?

```
function valid(ith) is:
  if ith.currentNode = NIL then
    valid ← false
  else
    valid ← true
  end-if
end-function
```

- Complexity of the algorithm: $\Theta(1)$

Coalesced chaining

- Collision resolution by coalesced chaining: each element from the hash table is stored inside the table (no linked lists), but each element has a *next* field, similar to a linked list on array.
- 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.
- Since elements are in the table, α can be at most 1.

Coalesced chaining - example

- Consider a hash table of size $m = 19$ that uses coalesced chaining for collision resolution and a hash function with the division method
- Insert into the table the letters from *A SEARCHING EXAMPLE* (space is ignored)
- For each letter, the *hashCode* is the index of the letter in the alphabet.

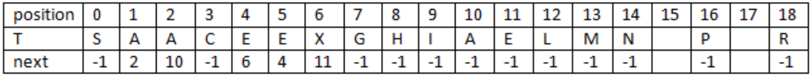
Letter	A	S	E	R	C	H	I	N	G	X	M	P	L
HashCode	1	19	5	18	3	8	9	14	7	24	13	16	12

Coalesced chaining - example

- Consider a hash table of size $m = 19$ that uses coalesced chaining for collision resolution and a hash function with the division method
- Insert into the table the letters from *A SEARCHING EXAMPLE* (space is ignored)
- For each letter, the *hashCode* is the index of the letter in the alphabet.

Letter	A	S	E	R	C	H	I	N	G	X	M	P	L
HashCode	1	19	5	18	3	8	9	14	7	24	13	16	12
h(Letter)	1	0	5	18	3	8	9	14	7	5	13	16	12

Coalesced chaining - example



position	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
T	S	A	A	C	E	E	X	G	H	I	A	E	L	M	N		P		R
next	-1	2	10	-1	6	4	11	-1	-1	-1	-1	-1	-1	-1	-1		-1		-1

- $m = 19$
- $\alpha = 0.89$
- $firstFree = 15$

Coalesced chaining - representation

- What fields do we need to represent a hash table where collision resolution is done with coalesced chaining?

Coalesced chaining - representation

- What fields do we need to represent a hash table where collision resolution is done with coalesced chaining?

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

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

Coalesced chaining - insert

```
subalgorithm insert (ht, k) is:
//pre: ht is a HashTable, k is a TKey
//post: k was added into ht
   pos ← ht.h(k)
   if ht.T[pos] = -1 then //-1 means empty position
      ht.T[pos] ← k
      ht.next[pos] ← -1
   else
      if ht.firstFree = ht.m then
         @resize and rehash
      end-if
      current ← pos
      while ht.next[current] ≠ -1 execute
         current ← ht.next[current]
      end-while
//continued on the next slide...
```

Coalesced chaining - insert

```
        ht.T[ht.firstFree] $\leftarrow$ k
        ht.next[ht.firstFree] $\leftarrow$ - 1
        ht.next[current] $\leftarrow$ ht.firstFree
        changeFirstFree(ht)
    **end-if**
**end-subalgorithm**
```

- Complexity: $\Theta(1)$ on average, $\Theta(n)$ - worst case

Coalesced chaining - ChangeFirstFree

```
subalgorithm changeFirstFree(ht) is:
//pre: ht is a HashTable
//post: the value of ht.firstFree is set to the next free position
   ht.firstFree ← ht.firstFree + 1
   while ht.firstFree < ht.m and ht.T[ht.firstFree] ≠ -1 execute
      ht.firstFree ← ht.firstFree + 1
   end-while
end-subalgorithm
```

- Complexity: $O(m)$
- *Think about it:* Should we keep the free spaces linked in a list as in case of a linked lists on array?

Coalesced chaining

- *Remove* and *search* operations for coalesced chaining will be discussed in Seminar 6.
- How can we define an iterator for a hash table with coalesced chaining? What should the following operations do?
  - init
  - getCurrent
  - next
  - valid

Open addressing

- In case of open addressing every element of the hash table is inside the table, we have no pointers, no next links.
- 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.

Open addressing

- 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.

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

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

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

Open addressing - Linear probing

- One version of 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 of size $m = 19$ that uses open addressing with linear probing for collision resolution (h'(k) is a hash function defined with the division method)
- Insert into the table the letters from *A SEARCHING EXAMPLE* (space is ignored)
- For each letter, the *hashCode* is the index of the letter in the alphabet.

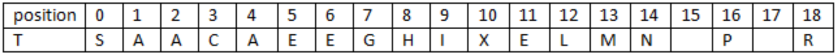
Letter	A	S	E	R	C	H	I	N	G	X	M	P	L
HashCode	1	19	5	18	3	8	9	14	7	24	13	16	12
h'(Letter)	1	0	5	18	3	8	9	14	7	5	13	16	12

Open addressing - Linear probing - example

position	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
T	S	A	A	C	A	E	E	G	H	I	X	E	L	M	N		P		R

- $\alpha = 0.89$

Open addressing - Linear probing - example



position	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
T	S	A	A	C	A	E	E	G	H	I	X	E	L	M	N		P		R

- $\alpha = 0.89$
- Disadvantages of linear probing:
  - There are only *m* distinct probe sequences (once you have the starting position everything is fixed)
  - *Primary clustering* - long runs of occupied slots

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 initialized when the hash function is initialized. 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: $< k, k + 1, k + 4, k + 9, k + 16, ... >$

Open addressing - Quadratic probing

- 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.
- If m is a prime number only the first half of the probe sequence is unique, so, once the hash table is half full, there is no guarantee that an empty space will be found.
    - For example, for $m = 17$, $c_1 = 3$, $c_2 = 1$ and $k = 13$, the probe sequence is $< 13, 0, 6, 14, 7, 2, 16, 15, 16, 2, 7, 14, 6, 0, 13, 11, 11 >$
    - For example, for $m = 11$, $c_1 = 1$, $c_2 = 1$ and $k = 27$, the probe sequence is $< 5, 7, 0, 6, 3, 2, 3, 6, 0, 7, 5 >$

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 $4 * k + 3$, $c_1 = 0$ 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 of size $m = 16$ that uses open addressing with quadratic probing for collision resolution (h'(k) is a hash function defined with the division method), $c_1 = c_2 = 0.5$.
- Insert into the table the letters from *HASHTABLE*
- For each letter, the *hashCode* is the index of the letter in the alphabet.

Letter	H	A	S	T	B	L	E
HashCode	8	1	19	20	2	12	5
h'(Letter)	8	1	3	4	2	12	5

Open addressing - Quadratic probing - example

position	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
T		A	A	S	T	B	E		H	H			L			

Open addressing - Quadratic probing - example

position	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
T		A	A	S	T	B	E		H	H			L			

- Disadvantages of quadratic probing:
  - The performance is sensitive to the values of m, c_1 and c_2.
  - *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 (once you have the starting position the whole sequence is fixed).

Open addressing - Double hashing

- In case of double hashing the hash function becomes:

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

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

- For a key, k, the first position examined will be $h'(k)$ and the other probed positions will be computed based on the second hash function, $h''(k)$.

Open addressing - Double hashing

- Similar to quadratic hashing, not every combination of m and $h''(k)$ will return a complete permutation as a probe sequence.
- In order to produce a permutation m and all the values of $h''(k)$ have to be relatively primes. This can be achieved in two ways:
    - Choose m as a power of 2 and design h'' in such a way that it always returns an odd number.
    - Choose m as a prime number and design h'' in such a way that it always return a value from the $\{0, \text{m-1}\}$ set.

Open addressing - Double hashing

- Choose m as a prime number and design h'' in such a way that it always return a value from the {0, m-1} set.
- 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 of size $m = 13$ that uses open addressing with double hashing for collision resolution, with $h'(k) = k\%m$ and $h''(k) = 1 + (k\%(m-1))$.
- Insert into the table the letters from *HASHTABLE*
- For each letter, the *hashCode* is the index of the letter in the alphabet.

Letter	H	A	S	T	B	L	E
HashCode	8	1	19	20	2	12	5
h'(Letter)	8	1	6	7	2	12	5
h''(Letter)	9	2	8	9	3	1	6

Open addressing - Double hashing - example

position	0	1	2	3	4	5	6	7	8	9	10	11	12
T		A	B	A	H	E	S	T	H				L

Open addressing - Double hashing - example

position	0	1	2	3	4	5	6	7	8	9	10	11	12
T		A	B	A	H	E	S	T	H				L

- 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:
  - Letter A, hashCode 1: $< 1, 3, 5, 7, 9, 11, 0, 2, 4, 6, 8, 10, 12 >$
  - Letter N, hashCode 14: $< 1, 4, 7, 10, 0, 3, 6, 9, 12, 2, 5, 8, 11 >$
- Since for every $(h'(k), h''(k))$ pair we have a separate probe sequence, double hashing generates $\approx m^2$ different permutations.

Open addressing - operations

- In the following we will discuss the implementation of the basic dictionary operations for collision resolution with open addressing.

- In the following, we will use the notation $h(k, i)$ for a hash function, without mentioning whether we have linear probing, quadratic probing or double hashing (code is the same for each of them, implementation of h is different only).

Open addressing - representation

- What fields do we need to represent a hash table with collision resolution with open addressing?

Open addressing - representation

- What fields do we need to represent a hash table with collision resolution with open addressing?

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

- For simplicity we will consider that we only have keys.

Open addressing - insert

- What should the *insert* operation do?

Open addressing - insert

- What should the *insert* operation do?

```
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] $\neq$ -1 execute
    //-1 means empty space
        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
```

Open addressing - search

- What should the *search* operation do?

Open addressing - search

- What should the *search* operation do?

```
function search (ht, e) is:
//pre: ht is a HashTable, e is a TKey
//post: returns true if e is in ht, false otherwise
    i ← 0
    pos ← ht.h(e, i)
    while $i < ht.m$ and ht.T[pos] $\neq$ -1 and ht.T.[pos] $\neq$ e execute
    //-1 means empty space
        i ← i + 1
        pos ← ht.h(e, i)
    end-while
    if $i < ht.m$ and ht.T[pos] = e execute
        search ← True
    else
        search ← False
    end-if
end-function
```

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?
- Removing an element from a hash table with open addressing is not simple:
  - we cannot just mark the position empty - *search* might not find other elements
  - you cannot move elements - *search* might not find other elements

Open addressing - remove

- How can we remove an element from the hash table?
- Removing an element from a hash table with open addressing is not simple:
  - we cannot just mark the position empty - *search* might not find other elements
  - you cannot move elements - *search* might not find other elements
- Remove is usually implemented to mark the deleted position with a special value, *DELETED*.
- How does this special value change the implementation of the *insert* and *search* operation?

Open addressing - Performance

- 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 complexity is $\Theta(1)$
- Worst case complexity is $O(n)$