DATA STRUCTURES AND ALGORITHMS

LECTURE 9

Marian Zsuzsanna

Babeș - Bolyai University
Computer Science and Mathematics Faculty

2017

In Lecture 8...

- ADT Deque
- ADT Priority Queue
- Binomial Heap

Today

1. Different problems
2. Hash tables

Different problems I

- Red-Black Card Game:
  - Statement: Two players each receive $\frac{n}{2}$ cards, where each card can be red or black. The two players take turns; at every turn the current player puts the card from the upper part of his/her deck on the table. If a player puts a red card on the table, the other player has to take all cards from the table and place them at the bottom of his/her deck. The winner is the player that has all the cards.
  - Requirement: Given the number n of cards, simulate the game and determine the winner.
  - Hint: use stack(s) and queue(s)

Different problems II

- Robot in a maze:
    - Statement: There is a rectangular maze, composed of occupied cells (X) and free cells (*). There is a robot (R) in this maze and it can move in 4 directions: N, S, E, V.
    - Requirements:
        - Check whether the robot can get out of the maze (get to the first or last line or the first or last column).
        - Find a path that will take the robot out of the maze (if exists).

X	*	*	X	X	X	*	*
X	*	X	*	*	*	*	*
X	*	*	*	*	*	X	*
X	X	X	*	*	*	X	*
*	X	*	*	R	X	X	*
*	*	*	X	X	X	X	*
*	*	*	*	*	*	*	X
X	X	X	X	X	X	X	X

Different problems III

- Hint - Version 1:
  - Let T be the set of positions where the robot can get from the starting position.
  - Let S be the set of positions to which the robot can get at a given moment and from which it could continue going to other positions.
  - A possible way of determining the sets T and S could be the following:

Different problems IV

T $\leftarrow$ {initial position}
S $\leftarrow$ {initial position}
while S $\neq \emptyset$ **execute**
  Let p be one element of S
  S $\leftarrow$ S $\setminus \{p\}$
  for each valid position q where we can get from p and which is not in T **do**
    T $\leftarrow$ T $\cup \{q\}$
    S $\leftarrow$ S $\cup \{q\}$
  end-for
end-while

- T can be a list, a vector or a matrix associated to the maze
- S can be a stack or a queue (or even a priority queue, depending on what we want)

Different problems V

- Hint - Version 2:
  - The solution is similar to the one presented on the previous slide.
  - If S is a queue, and T is a stack extended with the search operation, once we got out of the maze, T can be used to build the list of positions that got us to the margin of the maze. In this case we need both a stack and a queue.

Different problems VI

- How can we merge k sorted singly linked lists? How can we do it in $O(n * log_2 k)$ complexity (n is the total number of elements from the k lists)?

Direct-address tables I

- Consider the following problem:
  - We have data where every element has a key (a natural number).
  - The universe of keys (the possible values for the keys) is relatively small, $U = \{0, 1, 2, \ldots, m-1\}$
  - No two elements have the same key
  - We have to support the basic dictionary operations: INSERT, DELETE and SEARCH

Direct-address tables II

- Example 1: Store data about different bus lines for a city's public transportation service
  - We can consider the bus number as a key, and the data to be stored as a value (satellite data)
  - The bus numbers belong to a relatively small interval - in Cluj-Napoca it is around 100
  - Bus numbers are unique
- Example 2: Store data about employees of a company based on their employee numbers (a number from the 1 - 1000 interval, for example)

Direct-address tables III

- Solution:
  - Use an array T with m positions (remember, the keys belong to the $[0, m-1]$ interval)
  - Data about element with key k, will be stored in the $T[k]$ slot
  - Slots not corresponding to existing elements will contain the value NIL (or some other special value to show that they are empty)

Direct-address table - storing the elements

- In a direct-address table we have different ways of storing the elements:
  - We can store in the table pointers to elements.
  - We can store in the table the actual elements (both key and associated value).
  - We can store in the table only the value part (key is equal to the position) - we need a way of knowing if a position is occupied or not.

Operations for a direct-address table

function search(T, k) **is**:
//pre: T is an array (the direct-address table), k is a key
  search ← T[k]
end-function

Operations for a direct-address table

function search(T, k) **is**:
//pre: T is an array (the direct-address table), k is a key
    search ← T[k]
end-function

subalgorithm insert(T, x) **is**:
//pre: T is an array (the direct-address table), x is an element
    T[key(x)] ← x *//key(x) returns the key of element x*
end-subalgorithm

Operations for a direct-address table

function search(T, k) **is**:
//pre: T is an array (the direct-address table), k is a key
  search ← T[k]
end-function

subalgorithm insert(T, x) **is**:
//pre: T is an array (the direct-address table), x is an element
  T[key(x)] ← x *//key(x) returns the key of element x*
end-subalgorithm

subalgorithm delete(T, x) **is**:
//pre: T is an array (the direct-address table), x is an element
  T[key(x)] ← NIL
end-subalgorithm

Direct-address table - Advantages and disadvantages

- Advantages of direct address-tables:
    - They are simple
    - They are efficient - all operations run in $\Theta(1)$ time.
- Disadvantages of direct address-tables - restrictions on when they can be used:
    - The keys have to be natural numbers
    - The keys have to come from a small universe (interval)
    - The number of actual keys can be a lot less than the cardinal of the universe (storage space is wasted)

Hash tables

- Hash tables are generalizations of direct-address tables and they represent a *time-space trade-off*.
- Searching for an element still takes $\Theta(1)$ time, but as *average case complexity* (worst case complexity is higher)

Hash tables - main idea I

- We will still have a table T of size m (but now m is not the number of possible keys, $|U|$) - *hash table*
- Use a function h that will map a key k to a slot in the table T - *hash function*

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

- Remarks:
  - In case of direct-address tables, an element with key k is stored in $T[k]$.
  - In case of hash tables, an element with key k is stored in $T[h(k)]$.

Hash tables - main idea II

- The point of the hash function is to reduce the range of array indexes that need to be handled $=>$ instead of $|U|$ values, we only need to handle m values.
- Consequence:
  - two keys may hash to the same slot $=>$ **a collision**
  - we need techniques for resolving the conflict created by collisions

A good hash function I

- A good hash function:
  - can minimize the number of collisions (but cannot eliminate all collisions)
  - is deterministic
  - can be computed in $\Theta(1)$ time

A good hash function II

- satisfies (approximately) the assumption of simple uniform hashing: **each key is equally likely to hash to any of the m slots, independently of where any other key has hashed to**

$$P(h(k) = j) = \frac{1}{m} \; \forall j = 0, ..., m-1 \;\; \forall k \in U$$

- Examples of bad hash functions:
  - $h(k) =$ constant number
  - $h(k) =$ random number
  - $h(k) = k \; mod \; 10$ - when $m > 10$
  - etc.

A good hash function III

- In practice we use heuristic techniques to create hash functions that perform well.
- Most hash functions assume that the keys are natural numbers. If this is not true, they have to be interpreted as a natural number. In what follows, we assume that the keys are natural numbers.
- There are different methods for defining a hash function:
  - The division method
  - The multiplication method
  - Universal hashing

The division method

The division method

$h(k) = k \ \ mod \ \ m$

For example:

$$m = 13$$
$$k = 63 => h(k) = 11$$
$$k = 52 => h(k) = 0$$
$$k = 131 => h(k) = 1$$

- Requires only a division so it is quite fast
- Experiments show that good values for m are primes not too close to exact powers of 2

The multiplication method I

The multiplication method

$$h(k) = floor(m * frac(k * A)) \text{ where}$$

m - the hash table size

A - constant in the range $0 < A < 1$

$frac(k * A)$ - fractional part of $k * A$

For example

m = 13 A = 0.6180339887

k=63 => h(k) = floor(13 * frac(63 * A)) = floor(12.16984) = 12

k=52 => h(k) = floor(13 * frac(52 * A)) = floor(1.790976) = 1

k=129=> h(k)= floor(13 * frac(129 * A)) = floor(9.442999) = 9

The multiplication method II

- Advantage: the value of m is not critical, typically $m = 2^p$ for some integer p
- Some values for A work better than others. Knuth suggests $\frac{\sqrt{5}-1}{2} = 0.6180339887$

Universal hashing I

- If we know the exact hash function used by a hash table, we can always generate a set of keys that will hash to the same position (collision). This reduces the performance of the table.
- For example:

$m = 13$
$h(k) = k \ \ mod \ \ m$
k = 11, 24, 37, 50, 63, 76, etc.

Universal hashing II

- Instead of having one hash function, we have a collection $\mathcal{H}$ of hash functions that map a given universe U of keys into the range $\{0, 1, \ldots, m-1\}$
- Such a collection is said to be **universal** if for each pair of distinct keys $x, y \in U$ the number of hash functions from $\mathcal{H}$ for which $h(x) = h(y)$ is precisely $\frac{|\mathcal{H}|}{m}$
- In other words, with a hash function randomly chosen from $\mathcal{H}$ the chance of collision between x and y, where $x \neq y$, is exactly $\frac{1}{m}$

Universal hashing III

Example 1

Fix a prime number $p >$*the maximum possible value for a key from* U.
For every $a \in \{1, \ldots, p-1\}$ and $b \in \{0, \ldots, p-1\}$ we can define a hash function $h_{a,b}(k) = ((a * k + b) \ mod \ p) \ mod \ m$.

- For example:
  - $h_{3,7}(k) = ((3 * k + 7) \ mod \ p) \ mod \ m$
  - $h_{4,1}(k) = ((4 * k + 1) \ mod \ p) \ mod \ m$
  - $h_{8,0}(k) = ((8 * k) \ mod \ p) \ mod \ m$
- There are $p * (p - 1)$ possible hash functions that can be chosen.

Universal hashing IV

Example 2

If the key k is an array $< k_1, k_2, \ldots, k_r >$ such that $k_i < m$ (or it can be transformed into such an array).
Let $< x_1, x_2, \ldots, x_r >$ be a fixed sequence of random numbers, such that $x_i \in \{0, \ldots, m-1\}$.
$h(k) = \sum_{i=1}^{r} k_i * x_i \;\; mod \; m$

Universal hashing V

Example 3

Suppose the keys are $u - bits$ long and $m = 2^b$.
Pick a random $b - by - u$ matrix (called h) with 0 and 1 values only.
Pick $h(k) = h * k$ where in the multiplication we do addition mod 2.

$$\begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 \\ 1 & 1 & 1 & 0 \end{pmatrix} \begin{pmatrix} 1 \\ 0 \\ 1 \\ 0 \end{pmatrix} = \begin{pmatrix} 1 \\ 1 \\ 0 \end{pmatrix}$$

Using keys that are not natural numbers I

- The previously presented hash functions assume that keys are natural numbers.
- If this is not true there are two options:
  - Define special hash functions that work with your keys (for example, for real number from the [0,1) interval $h(k) = [k * m]$ can be used)
  - Use a function that transforms the key to a natural number (and use any of the above-mentioned hash functions on the result) - *hashCode*

Using keys that are not natural numbers II

- If the key is a string s:
  - we can consider the ASCII codes for every letter
  - we can use 1 for *a*, 2 for *b*, etc.
- Possible implementations for *hashCode*
  - $s[0] + s[1] + ... + s[n-1]$
    - Anagrams have the same sum (*SAUCE* and *CAUSE*)
    - *DATES* has the same sum (D = C + 1, T = U - 1)
    - Assuming maximum length of 10 for a word (and the second letter representation), *hashCode* values range from 1 (the word *a*) to 260 (*zzzzzzzzzz*). Considering a dictionary of about 50,000 words, we would have on average 192 words for a *hashCode* value.

Using keys that are not natural numbers III

- $s[0] * 26^{n-1} + s[1] * 26^{n-2} + ... + s[n-1]$ where
  - n - the length of the string
  - Generates a much larger interval of *hashCode* values.
  - Instead of 26 (which was chosen since we have 26 letters) we can use a prime number as well (Java uses 31, for example).

Collisions

- When two keys, x and y, have the same value for the hash function $h(x) = h(y)$ we have a *collision*.
- A good hash function can reduce the number of collisions, but it cannot eliminate them at all:
  - Try fitting $m + 1$ keys into a table of size m
- There are different collision resolution methods:
  - Separate chaining
  - Coalesced chaining
  - Open addressing

The birthday paradox

- *How many randomly chosen people are needed in a room, to have a good probability - about 50% - of having two people with the same birthday?*
- It is obvious that if we have 367 people, there will be at least two with the same birthday (there are only 366 possibilities).
- What might not be obvious, is that approximately 70 people are needed for a 99.9% probability
- 23 people are enough for a 50% probability

Separate chaining

- Collision resolution by separate chaining: each slot from the hash table T contains a linked list, with the elements that hash to that slot
- Dictionary operations become operations on the corresponding linked list:
  - $insert(T, x)$ - insert a new node to the beginning of the list $T[h(key[x])]$
  - $search(T, k)$ - search for an element with key k in the list $T[h(k)]$
  - $delete(T, x)$ - delete x from the list $T[h(key[x])]$

Hash table with separate chaining - representation

- A hash table with separate chaining would be represented in the following way (for simplicity, we will keep only the keys in the nodes).

Node:
  key: TKey
  next: ↑ Node

HashTable:
  T: ↑Node[] *//an array of pointers to nodes*
  m: Integer
  h: TFunction *//the hash function*

Hash table with separate chaining - insert

subalgorithm insert(ht, k) **is**:
//pre: ht is a HashTable, k is a TKey
//post: k was inserted into ht
  position ← ht.h(k)
  allocate(newNode)
  [newNode].next ← NIL
  [newNode].key ← k
  if ht.T[position] = NIL **then**
    ht.T[position] ← newNode
  else
    [newNode].next ← ht.T[position]
    ht.T[position] ← newNode
  end-if
end-subalgorithm

Hash table with separate chaining - search

```
function search(ht, k) is:
//pre: ht is a HashTable, k is a TKey
//post: function returns True if k is in ht, False otherwise
    position ← ht.h(k)
    currentN ← ht.T[position]
    while currentN ≠ NIL and [currentN].key ≠ k execute
        currentN ← [currentN].next
    end-while
    if currentN ≠ NIL then
        search ← True
    else
        search ← False
    end-if
end-function
```

- Usually search returns the info associated with the key k, not True or False (but now we work only with keys)

Analysis of hashing with chaining

- The average performance depends on how well the hash function h can distribute the keys to be stored among the m slots.
- **Simple Uniform Hashing** assumption: each element is equally likely to hash into any of the m slots, independently of where any other elements have hashed to.
- **load factor** α of the table T with m slots containing n elements
  - is n/m
  - represents the average number of elements stored in a chain
  - in case of separate chaining can be less than, equal to, or greater than 1.

Analysis of hashing with chaining - Insert

- The slot where the element is to be added can be:
  - empty - create a new node and add it to the slot
  - occupied - create a new node and add it to the beginning of the list
- In either case worst-case time complexity is: $\Theta(1)$
- If we have to check whether the element already exists in the table, the complexity of searching is added as well.

Analysis of hashing with chaining - Search I

- There are two cases
  - unsuccessful search
  - successful search
- We assume that
  - the hash value can be computed in constant time ($\Theta(1)$)
  - the time required to search an element with key k depends linearly on the length of the list $T[h(k)]$

Analysis of hashing with chaining - Search II

- **Theorem:** In a hash table in which collisions are resolved by separate chaining, an unsuccessful search takes time $\Theta(1 + \alpha)$, on the average, under the assumption of simple uniform hashing.
- **Theorem:** In a hash table in which collisions are resolved by chaining, a successful search takes time $\Theta(1 + \alpha)$, on the average, under the assumption of simple uniform hashing.
- Proof idea: $\Theta(1)$ is needed to compute the value of the hash function and α is the average time needed to search one of the *m* lists

Analysis of hashing with chaining - Search III

- If $n = O(m)$ (the number of hash table slots is proportional to the number of elements in the table)
  - $\alpha = n/m = O(m)/m = O(1)$
  - searching takes constant time on average
- Worst-case time complexity is $\Theta(n)$
  - When all the nodes are in a single linked-list and we are searching this list
  - In practice hash tables are pretty fast

Analysis of hashing with chaining - Delete

- If the lists are doubly-linked and we know the address of the node: $\Theta(1)$
- If the lists are singly-linked: proportional to the length of the list

- **All dictionary operations can be supported in $\Theta(1)$ time on average.**
- In theory we can keep any number of elements in a hash table with separate chaining, but the complexity is proportional to α. If α is too large $\Rightarrow$ resize and rehash.

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

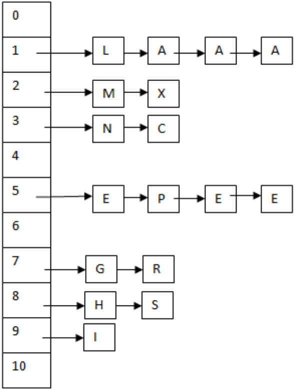
- Insert into the table the letters from *A SEARCHING EXAMPLE* (space is ignored)

- 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?
- 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.

<u>IteratorHT:</u>
  ht: HashTable
  currentPos: Integer
  currentNode: $\uparrow$ Node

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 *//the hash table is empty*
    ith.currentNode ← NIL
  end-if
end-subalgorithm

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

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?

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 $\neq$ NIL **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?

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

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