Tirgul 8

- *Universal Hashing*
- *Remarks on Programming Exercise 1*
- *Solution to question 2 in theoretical homework 2*

Hash Tables

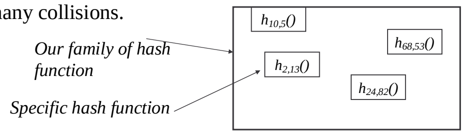
- We want to manage a set of n elements with the dictionary operations Insert, Search and Delete.
- Each element has a unique key from a universe U of possible keys, |U| >> n
- Hash table – an array of size m, m << |U|
- Hash function – a function that maps a key to a cell in the hash table.
- Required property – in order to work fast, the elements in the hash table should be equally distributed among the cells.
- Can you find a hash function that has this property on any input?
- No – since |U| >> m, there is always a bad input

Flashback

- Quick-sort
  - the pivot's position is fixed
  - there are good inputs and bad inputs.
- Randomized Quick-sort
  - uniform distribution on all the possible pivots
  - No more inputs discrimination – all the inputs have the same probability of working fast.

Universal Hashing

- Starting point: for every hash function, there is a "really bad" input.
- A possible solution: just as in quick sort, randomize the algorithm instead of looking at a random input.
- The logic behind it: There is no bad input. For every input there is a small chance of choosing a bad hash function for this input, i.e. a function that will cause many collisions.



Order of execution

- The order of execution:
  1. The input is fixed (everything that will be fed into the program is considered an input and is fixed now).
  2. The program is run
  3. The hash function is chosen (randomly) and remains fixed for the entire duration of the program run.

Ideal case - take 1

- What is our "ideal case"? (that we always use when trying to analyze good hash functions)
- A random choice of index.
- First try we will call a function good if:
  For every key *k*, and index *i* it holds that
  $P_h[\ h(k) = i] = 1/m$
- Is that good enough?
- Look at $\{ h_i(\cdot) \mid$ for every key $k : h_i(k) = i \}$
  This is obviously a bad family (everything collides).

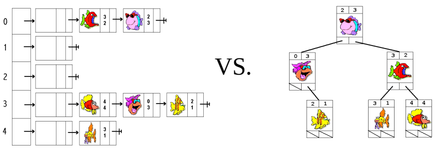
Ideal case - take 2

- We want that the probability of collision will be as in the random case.
- For every pair of keys $k_1 \neq k_2$ and pair of indices i_1, i_2 it holds that $P_h[\ h(k_1) = i_1 \ and\ \ h(k_2) = i_2\ ] = 1/m^2$
- What is the probability of collision?
- Sum the probabilities of collision in cell *i* for each i. This means $m*1/m^2 = 1/m$
- This is enough to ensure the expected number of collisions is as in the random case.

Ensuring good average performance

The chance that two keys will fall to the same slot is *1/m* - just like if the hash function was truly random!

Claim: When using a universal hash family H, the average number of look-ups of any hash operation is *n/m* (as in the random case)



- Hash table - If we have an estimation of *n*, the number of elements inserted to the table, we can choose the size of the table to be proportional to n. Then, we will have constant time performance - no matter how many elements we have: 10^6, 10^8, 10^{10}, or more...
- Balanced Tree - The performance of a balanced tree, *O(log n),* is affected by the number of elements we have! As we have more elements, we have slower operations. For very large numbers, like 10^{10}, this makes a difference.

Constructing a universal family

Choose p - a prime larger than all keys.

For any a,b in $Z_p=\{0,...,p-1\}$ denote fix a hash function:

$$h_{a,b}(k) = (a*k + b) \bmod p$$

<u>The universal family</u>: $H_p = \{ h_{a,b}(\cdot) \mid a,b \text{ in } Z_p \}$

<u>Theorem</u>: H_p is a universal family of hash functions.

Proof: for each $k_1 \neq k_2$ and each i_1,i_2 there is exactly one solution (a,b) for the equations :

$a*k_1 + b = i_1$ and $a*k_2 + b = i_2$.

Average over inputs – exact analsys

- In Universal Hashing - <u>no assumptions about the input</u> (I.e. for any input, most hash functions will handle it well).
- For example, we don't know a-priori how the grades are distributed. (surely they are not uniform over 0-100).
- If we <u>know</u> that the input has a specific distribution, we might want to use this.
- For example, if the input is uniformly distributed, then the simple division method will obtain simple uniform hashing.
- In general, we don't know the input's distribution, and so Universal Hashing is superior!

Programming exercise 1 Issues

- **Encapsulation** – hide internal structure from the user!
  - Easy maintenance using building blocks.
  - Protect the system from it's users - the user can't cause inconsistencies.
- **Reusable data structures** – DS implementation should fit all objects, not just books.
- **Sorting in a sorted map** – a sorted map is sorted only by <u>key</u> by definition. Sorting by the value as well is a violation of the 'contract'.
- **Documentation** –
  - **informative** documentation of methods and data members (//data members - not informative).
  - Different **naming conventions** for local variables vs. data members improves readability

Programming exercise 1 Issues

- **Debugging** – a major part of the exercise. Test different inputs as well as load tests.
- **Equals vs. ==**
  - == compares **addresses** of objects
  - **Equals** compares the **values** of the strings.
- The complication – compilers optimizations might put a constant string that is pointed at from different pointers in the same address. In this case, == and equals give the same results.
- We cannot count on this:
  - Using **new** allocates different addresses
  - Different compilers
- If your code failed on many tests because of this mistake, fix the problematic comparisons, and send only the fixed files (say exactly where the problem was).

Theoretical homework 2, Question 2

Finding an element

Let *A* be a data structure consisting of integers, and *x* an integer. The function *IsMember* gets as input *A* and *x*, and returns *true* when *x* appears in *A*, *false* otherwise.

(a) Assume *A* is an array. Write pseudo-code for *IsMember* function and analyze its worst-case complexity.

Solution:

The algorithm:

Go over the elements in *A* and compare them to *x*.

Stop when *x* is found or when <u>all</u> the elements in *A* are checked.

Worst-case time complexity: *O(n).*

Theoretical homework 2, Question 2

(b) Assume we can preprocess the contents of the array *A* in order to decrease the cost of *IsMember*. We can store the result of the preprocessing in a new data structure *B*.

(i) Indicate what data structure *B* and pre-processing function *PreProcess* can be used to speed-up *IsMember*. What is the worst-case complexity of *PreProcess*?

(ii) Write a function *IsMember I* that takes advantage of the data structure *B*. What is the worst-case complexity of *IsMember* ?

What is pre-processing and what is it good for?

Think about the following scenario: the site www.hevre.co.il keeps many lists of 'hevre' from different schools, movements, work places etc., some of them very popular.

Theoretical homework 2, Question 2

Once you are registered to a group, you can enter with your unique login as many times as you like and see what's up.

Identification is done by calling *IsMember*. As the lists become longer, identification becomes slower.

As the site's manager, you decide to organize the lists more efficiently so that *IsMember* works fast. What you need is a **pre-processing** function.

Pre-process – the idea of pre-processing is to organize the data in a certain way before starting to use it, so that common / online operations will work fast.

Theoretical homework 2, Question 2

Solution to 2-b-i:

- Pre-process – sort the array in *O(n log n)*
- *IsMember* = binary search, *O(log n)*

Do you have another solution?

Solution 2:

- Pre-process – build a hash table, *O(n)*
- *IsMember* – look for the key in the hash table, *O(1)* on the average.

Theoretical homework 2, Question 2

(c) Assume now that the array *A* is sorted but its contents are not known in advance. Write a new function *IsMember* whose worst-case complexity is *O(log i)*, when *x* appears in *A[i]*, and *O(log n)* when *x* does not appear in *A*.

Solution:

- First step – determine the range of *x* (lower and upper bounds) in *O(log i)*. Can we run a binary search?
- No, The required time complexity does not suit binary search
- So, what can we do?
- Search algorithm:
- Initialize: *u = 1*

```
While A[u] < x &  u < A.size
      u = 2*u
If ( u > A.size )
      u = A.size
Perform binary search of x in A[1…u]
```

DAST 2004

Theoretical homework 2, Question 2

- Complexity Analysis: If *x* is in index *i*,
- To get the upper bound u we do most $\lceil log(i) \rceil$ operations
- Next we perform binary search on *A[1…u]* complexity *O(log u)*
- Since $u < 2*i$ we get $O(\log u) = O(\log i)$