DATA STRUCTURES

LECTURE 10

Lect. PhD. Oneț-Marian Zsuzsanna

Babeș - Bolyai University
Computer Science and Mathematics Faculty

2020 - 2021

In Lecture 9...

Binary Heap

Today

- Hash Tables

Example

- Assume that you were asked to write an application for Cluj-Napoca's public transportation service.
- In your application the user can select a bus line and the application should display the timetable for that bus-line (and maybe later the application can be extended with other functionalities).
- Your application should be able to return the info for a bus line and we also want to be able to add and remove bus lines (this is going to be done only by the administrators, obviously).
- And since your application is going to be used by several hundred thousand people, we need it to be very very fast.
    - The public transportation service is willing to maybe rename a few bus lines, if this helps you design a fast application.
- How/Where would you store the data?

Direct-address tables I

- If we want to formalize the 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

- 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)

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 an element
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 an element
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:
  - 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)

Think about it

- Assume that we have a direct address T of length m. How can we find the maximum element of the direct-address table? What is the complexity of the operation?

- How does the operation for finding the maximum change if we have a hash table, instead of a direct-address table (consider collision resolution by separate chaining, coalesced chaining and open addressing)?

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
- The two main points of discussion for hash tables are:
  - How to define the hash function
  - How to resolve 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

Examples of bad hash functions

- $h(k) =$ constant number
- $h(k) =$ random number

Examples of bad hash functions

- $h(k) =$ constant number
- $h(k) =$ random number
- assuming that the keys are CNP numbers:
  - a hash function considering just parts of it (first digit, birth year/date, county code, etc.)
  - assume m = 100 and you use the birth day from the CNP (as a number): h(CNP) = birthday % 100

Examples of bad hash functions

- $h(k)$ = constant number
- $h(k)$ = random number
- assuming that the keys are CNP numbers:
  - a hash function considering just parts of it (first digit, birth year/date, county code, etc.)
  - assume m = 100 and you use the birth day from the CNP (as a number): h(CNP) = birthday % 100
- m = 16 and h(k) % m can also be problematic

Examples of bad hash functions

- $h(k)$ = constant number
- $h(k)$ = random number
- assuming that the keys are CNP numbers:
    - a hash function considering just parts of it (first digit, birth year/date, county code, etc.)
    - assume m = 100 and you use the birth day from the CNP (as a number): h(CNP) = birthday % 100
- m = 16 and h(k) % m can also be problematic
- etc.

Hash function

- The simple uniform hashing theorem is hard to satisfy, especially when we do not know the distribution of data. Data does not always have a uniform distribution
  - dates
  - group numbers at our faculty
  - postal codes
  - first letter of an English word
- 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 natural number. In what follows, we assume that the keys are natural numbers.

The division method

The division method

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

For example:

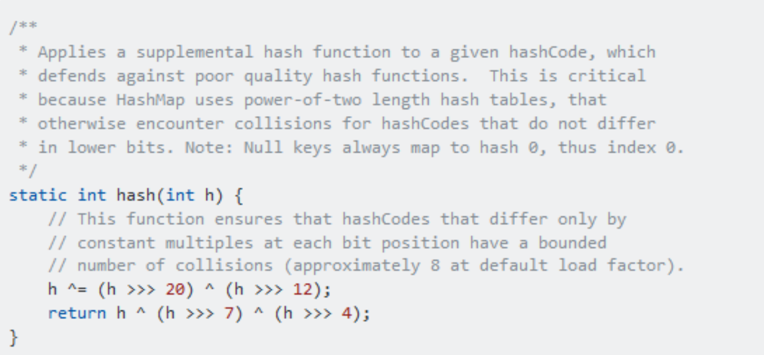
$$\begin{gathered}\mathrm{m} = 13 \\ \mathrm{k} = 63 => \mathrm{h(k)} = 11 \\ \mathrm{k} = 52 => \mathrm{h(k)} = 0 \\ \mathrm{k} = 131 => \mathrm{h(k)} = 1\end{gathered}$$

- 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 division method

- Interestingly, Java uses the division method with a table size which is power of 2 (initially 16).

- They avoid a problem by using a second function for hashing, before applying the mod:



```
/**
 * Applies a supplemental hash function to a given hashCode, which
 * defends against poor quality hash functions.  This is critical
 * because HashMap uses power-of-two length hash tables, that
 * otherwise encounter collisions for hashCodes that do not differ
 * in lower bits. Note: Null keys always map to hash 0, thus index 0.
 */
static int hash(int h) {
    // This function ensures that hashCodes that differ only by
    // constant multiples at each bit position have a bounded
    // number of collisions (approximately 8 at default load factor).
    h ^= (h >>> 20) ^ (h >>> 12);
    return h ^ (h >>> 7) ^ (h >>> 4);
}
```

Mid-square method

- Assume that the table size is 10^r, for example m = 100 (r = 2)
- For getting the hash of a number, multiply it by itself and take the middle *r* digits.
- For example, h(4567) = middle 2 digits of 4567 * 4567 = middle 2 digits of 20857489 = 57
- Same thing works for $m = 2^r$ and the binary representation of the numbers
- $m = 2^4$, h(1011) = middle 4 digits of 01111001 = 1110

The multiplication method I

The multiplication method

$h(k) = \mathit{floor}(m \ast \mathit{frac}(k \ast A))$ where

m - the hash table size

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

$\mathit{frac}(k \ast A)$ - fractional part of $k \ast A$

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 $a \in \{1, \dots, p-1\}$ and $b \in \{0, \dots, p-1\}$ 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 $< k_1, k_2, \dots, k_r >$ such that $k_i < m$ (or it can be transformed into such an array, by writing the k as a number in base m).
Let $< x_1, x_2, \dots, x_r >$ be a fixed sequence of random numbers, such that $x_i \in \{0, \dots, m-1\}$ (another number in base m with the same length).
$h(k) = \sum_{i=1}^{r} k_i * x_i \ \ mod \ m$

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) - *hashCode* in Java, *hash* in Python

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 word 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).

Cryptographic hashing

Cryptographic hashing

- Another use of hash functions besides as part of a hash table
- It is a hash function, which can be used to generate a code (the hash value) for any variable size data
- Used for checksums, storing passwords, etc.

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).

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 - 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)
    currentNode ← ht.T[position]
    while currentNode $\neq$ NIL and [currentNode].key $\neq$ k execute
        currentNode ← [currentNode].next
    end-while
    if currentNode $\neq$ NIL then
        search ← True
    else
        search ← False
    end-if
end-function
```

- Usually search returns the info associated with the key k

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, if the number of elements grows, the size of the table will grow as well)
  - $\alpha = n/m = O(m)/m = \Theta(1)$
  - 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.**

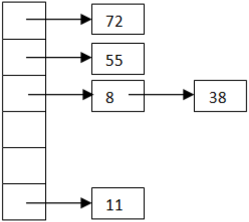
- In theory we can keep any number of elements in a hash table with separate chaining, but the complexity is proportional to α. If α is too large $\Rightarrow$ resize and rehash.

Example

- Assume we have a hash table with $m = 6$ that uses separate chaining for collision resolution, with the following policy: if the load factor of the table after an insertion is greater than or equal to 0.7, we double the size of the table
- Using the division method, insert the following elements, in the given order, in the hash table: 38, 11, 8, 72, 57, 29, 2.

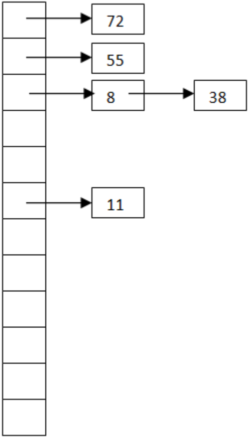
Example

- $h(38) = 2$ (load factor will be $1/6$)
- $h(11) = 5$ (load factor will be $2/6$)
- $h(8) = 2$ (load factor will be $3/6$)
- $h(72) = 0$ (load factor will be $4/6$)
- $h(55) = 1$ (load factor will be $5/6$ - greater than 0.7)
- The table after the first five elements were added:



Example

- Is it OK if after the resize this is our hash table?

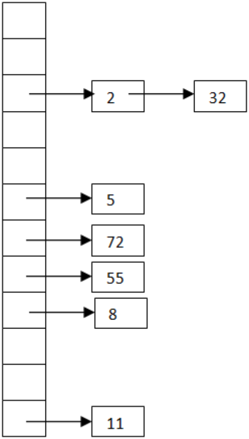


Example

- The result of the hash function (i.e. the position where an element is added) depends on the size of the hash table. If the size of the hash table changes, the value of the hash function changes as well, which means that search and remove operations might not find the element.
- After a resize operation, we have to add all elements again in the hash table, to make sure that they are at the correct position → rehash

Example

- After rehash and adding the other two elements:



Iterator

- What do you think, which containers cannot be represented on a hash table?

Iterator

- What do you think, which containers cannot be represented on a hash table?
- How can we define an iterator for a hash table with separate chaining?

Iterator

- What do you think, which containers cannot be represented on a hash table?
- How can we define an iterator for a hash table with separate chaining?
- Since hash tables are used to implement containers where the order of the elements is not important, our iterator can iterate through them in any order.
- For the hash table from the previous example, the easiest order in which the elements can be iterated is: 2, 32, 5, 72, 55, 8, 11

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:

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 $\leftarrow$ ht
    ith.currentPos $\leftarrow$ 0
    while ith.currentPos $<$ ht.m and ht.T[ith.currentPos] $=$ NIL execute
        ith.currentPos $\leftarrow$ ith.currentPos $+$ 1
    end-while
    if ith.currentPos $<$ ht.m then
        ith.currentNode $\leftarrow$ ht.T[ith.currentPos]
    else
        ith.currentNode $\leftarrow$ NIL
    end-if
end-subalgorithm
```

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

Iterator - other operations

- How can we implement the *getCurrent* operation?

Iterator - other operations

- How can we implement the *getCurrent* operation?
- How can we implement the *next* operation?

Iterator - other operations

- How can we implement the *getCurrent* operation?
- How can we implement the *next* operation?
- How can we implement the *valid* operation?

Sorted containers

- How can we define a sorted container on a hash table with separate chaining?

Sorted containers

- How can we define a sorted container on a hash table with separate chaining?
  - Hash tables are in general not very suitable for sorted containers.
  - However, if we have to implement a sorted container on a hash table with separate chaining, we can store the individual lists in a sorted order and for the iterator we can return them in a sorted order.