DATA STRUCTURES (AND ALGORITHMS)

Hast Tables: Introduction, Separate Chaining.

Lect. PhD. Diana-Lucia Miholca

2023 - 2024

Babeş - Bolyai University
Faculty of Mathematics and Computer Science

In Lecture 6...

- ADT List
- ADT Stack
- ADT Queue

Today

- Hash Tables
  - Direct-address table
  - Introduction to hash tables
  - Hash tables with separate chaining

Direct-address table

Consider the following problem:

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

Direct-address table

Solution:

Use an array T with m positions (since the the keys belong to $\{0, 1, 2, ..., m - 1\}$)

- The element with key k, will be stored in the $T[k]$ slot
- Slots not corresponding to existing elements will contain the value NIL

Operations for a direct-address table - search

Searching in a direct-address table:

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

Operations for a direct-address table - insert

Inserting in a direct-address table:

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

Operations for a direct-address table - delete

Deleting from a direct-address table:

subalgorithm delete(T, x) **is**:

//pre: T is an array (the direct-address table), x is an element

Operations for a direct-address table - delete

Deleting from a direct-address table:

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

Direct-address table - Advantages and disadvantages

- Advantages of direct-address tables:

Direct-address table - Advantages and disadvantages

Advantages of direct-address tables:

- They are simple

Direct-address table - Advantages and disadvantages

Advantages of direct-address tables:

- They are simple
- They are time-efficient - all operations run in $\Theta(1)$ time.

Direct-address table - Advantages and disadvantages

Advantages of direct-address tables:

- They are simple
- They are time-efficient - all operations run in $\Theta(1)$ time.

Disadvantages of direct-address tables:

Direct-address table - Advantages and disadvantages

Advantages of direct-address tables:

- They are simple
- They are time-efficient - all operations run in $\Theta(1)$ time.

Disadvantages of direct-address tables:

- The keys have to be natural numbers
- The keys have to come from a small universe (interval)

Direct-address table - Advantages and disadvantages

Advantages of direct-address tables:

- They are simple
- They are time-efficient - all operations run in $\Theta(1)$ time.

Disadvantages of direct-address tables:

- The keys have to be natural numbers
- The keys have to come from a small universe (interval)
- The number of actual keys $<<$ the cardinal of the universe $\Rightarrow$ storage space is wasted

Hash tables

A **hash table** is a generalization of a direct-address table that represents 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

There is still a table T of size m (but m is not the number of possible keys, $|U|$).

There is also a function h, called *hash function*, that maps a key k to an index in the table T:

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

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

The aim of the hash function is to reduce the range of array indexes that are needed $=>$ instead of $|U|$ indexes, we only need m indexes.

Hash tables - main idea

- The aim of the hash function is to reduce the range of array indexes that are needed $=>$ instead of $|U|$ indexes, we only need m indexes.

- Two keys may hash to the same index $=>$ **a collision** $=>$ we need techniques for resolving the conflict created by collisions.

Hash tables - main idea

The aim of the hash function is to reduce the range of array indexes that are needed $=>$ instead of $|U|$ indexes, we only need m indexes.

Two keys may hash to the same index $=>$ **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

A good hash function:

- ✓ is deterministic
- ✓ can be computed in $\Theta(1)$ time
- ✓ can minimize the number of collisions
- ✓ 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

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

Examples of bad hash functions

- $h(k) =$ constant number

Examples of bad hash functions

- $h(k) =$ constant number
  - does not reduce the number of collisions

Examples of bad hash functions

- $h(k) = $ constant number
  - does not reduce the number of collisions
- $h(k) = $ random number

Examples of bad hash functions

- $h(k) =$ constant number
  - does not reduce the number of collisions
- $h(k) =$ random number
  - it is not deterministic

Examples of bad hash functions

- $h(k) =$ constant number
  - does not reduce the number of collisions
- $h(k) =$ random number
  - it is not deterministic
- assuming that a key is a Personal Numeric Code, a hash function considering just parts of it (first digit, birth year/date, county code, etc.)
  - favors the collisions

Examples of bad hash functions

- $h(k) =$ constant number
  - does not reduce the number of collisions
- $h(k) =$ random number
  - it is not deterministic
- assuming that a key is a Personal Numeric Code, a hash function considering just parts of it (first digit, birth year/date, county code, etc.)
  - favors the collisions
- h(k) = k % m, when m = 16
  - favors the collisions, by considering only the last four bites

Hash function

The simple uniform hashing assumption is hard to satisfy.

In practice, we use heuristic techniques to create hash functions that perform well.

The division method

The division method

$h(k) = k \mod m$

For example:

.

m = 13

k = 24 => h(k) = 11

k = 26 => h(k) = 0

k = 131 => h(k) = 1

.

- ✓ Requires only one 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).

To avoid a problem, a second function is used, 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);
}
```

The multiplication method

The multiplication method:

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

m - the hash table size

A - constant, $0 < A < 1$

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

Some values for A work better than others. Knuth suggests $\frac{\sqrt{5}-1}{2} = 0.6180339887$ (golden ratio)

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 value of m is not critical, typically $m = 2^p$

Universal hashing

If we know the exact hash function used by a hash table, we can always generate a set of keys that will collide. This reduces the performance of the table.

Example:

$m = 13$

$h(k) = k \mod m$

k = 1, 14, 27, 40, 53, 66, etc.

Universal hashing

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 **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 $\frac{|\mathcal{H}|}{m}$.

With a hash function randomly chosen from $\mathcal{H}$ the chance of collision between x and y, where $x \neq y$, is $\frac{1}{m}$.

Universal hashing

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 for any p.

Universal hashing

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, by writing k in base m), let $< x_1, x_2, \ldots, x_r >$ be a fixed sequence of random numbers, such that $x_i \in \{0, \ldots, m-1\}$ (another number in base m with the same length).
$h(k) = \sum_{i=1}^{r} k_i * x_i \;\; mod \; m$

Universal hashing

Example 3 (the matrix method):

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.
Opt for $h(k) = h * k \ (mod\ 2)$ (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

The majority of the previously presented hash functions assume that the keys are natural numbers.

If this is not true, there are two options:

- Define special hash functions that work with the given keys
  - For example, for real number from the [0,1) interval $h(k) = [k * m]$ can be used
- Use a function that converts the key to a natural number
  - *hashCode* in Java, *hash* in Python

Using keys that are not natural numbers

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*)

  ! 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

$s[0] * 26^{n-1} + s[1] * 26^{n-2} + ... + s[n-1]$, where n is the length of the string

- Generates a much larger interval of *hashCode* values.

- Instead of 26 (chosen because 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:

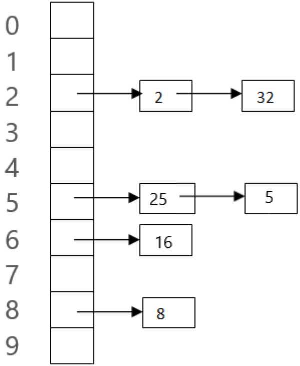
- Separate chaining
- Coalesced chaining
- Open addressing

Separate chaining

Collision resolution by separate chaining: each slot from the hash table T contains a linked list with all the elements that hash to that slot.

Separate chaining - Example

- m = 10
- h(k) = k % m



Separate chaining - Operations

The operations are performed on the corresponding linked list:

- $insert(T, x)$

Separate chaining - Operations

The operations are performed on the corresponding linked list:

- $insert(T, x)$ - insert a new node to the beginning of the list $T[h(key(x))]$
- $search(T, k)$

Separate chaining - Operations

The operations are performed 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 $T[h(k)]$
- $delete(T, x)$

Separate chaining - Operations

The operations are performed 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 is represented in the following way:

Representation of a node:

Node:
  key: TKey
  next: ↑ Node

Representation of a hash table with separate chaining:

HashTable:
  T: ↑Node[] *//an array of pointers to nodes*
  m: Integer
  h: TFunction: TKey → {0, 1, ..., m-1} *//the hash function*

For simplicity, we keep only the keys in nodes

Hash table with separate chaining - search

Searching in a hash table with separate chaining:

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

Hash table with separate chaining - search

Searching in a hash table with separate chaining:

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)

Hash table with separate chaining - search

Searching in a hash table with separate chaining:

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]

Hash table with separate chaining - search

Searching in a hash table with separate chaining:

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 $\leftarrow$ ht.h(k)
    currentNode $\leftarrow$ ht.T[position]
    while currentNode $\neq$ NIL **and** [currentNode].key $\neq$ k **execute**
        currentNode $\leftarrow$ [currentNode].next
    end-while

Hash table with separate chaining - search

Searching in a hash table with separate chaining:

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

Usually *search* returns the value associated with the key *k*

Analysis of hashing with chaining

The average time-performance depends on the quality of the hash function.

Simple Uniform Hashing (SUH) assumption: each element is equally likely to hash to any of the m slots, independently of where any other elements have hashed to.

The **load factor**, α, of the table T with m slots containing n elements is n/m and represents the average number of elements stored in a chain.

In case of separate chaining, it can be less than, equal to, or greater than 1.

Analysis of hashing with chaining - Search

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

Analysis of hashing with chaining - Search

Theorem: In a hash table with separate chaining, an *unsuccessful* search takes $\Theta(1+\alpha)$, on the average, under the assumption of SUH.

Theorem: In a hash table with separate chaining, a *successful* search takes time $\Theta(1+\alpha)$, on the average, under the assumption of SUH.

Proof idea: $\Theta(1)$ is needed to compute the hash value, while $\Theta(\alpha)$ is the average time needed to search in one of the m lists.

Analysis of hashing with chaining - Search

If $n = O(m)$:

$\alpha = n/m = O(m)/m = \Theta(1)$

searching takes constant time on average

Worst-case time complexity is $\Theta(n)$

when all the elements collide $\Rightarrow$ they are in the same list and we are searching this list

In practice, hash tables are pretty fast.

Analysis of hashing with chaining - Insert

- We create a new node and add it to the beginning of the list at index $h(key(x))$

Analysis of hashing with chaining - Insert

We create a new node and add it to the beginning of the list at index $h(key(x))$

The worst-case time complexity is $\Theta(1)$

If we have to check whether the key already exists in the table, then the complexity of searching should be taken into account, too.

Analysis of hashing with chaining - Delete

We have to search for the node containing the element to be deleted and remove the node (if found)

- We can also find the node previous to the one to be deleted, while searching, in order to facilitate deletion

The time-complexity is given by the searching part.

Conclusions

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 that uses separate chaining for collision resolution, with:

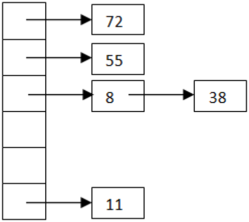
- m = 6
- the following resizing policy: if $\alpha \geq 0.7 \Rightarrow$ we double the size of the table

Using the division method, insert the following elements, in the given order, in the hash table: 38, 11, 8, 72, 55, 29, 2.

Example

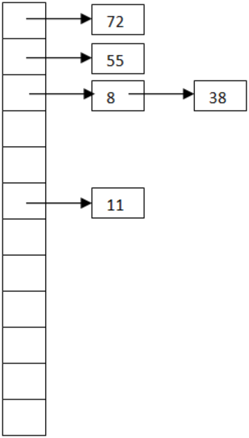
- $h(38) = 2$ (load factor will be 1/6)
- $h(11) = 5$ (load factor will be 2/6)
- $h(8) = 2$ (load factor will be 3/6)
- $h(72) = 0$ (load factor will be 4/6)
- $h(55) = 1$ (load factor will be 5/6 - greater than 0.7)

- The table after the first five elements were added:



Example

Is it OK if after the resize the hash table, with $m = 12$ is the following?



Example

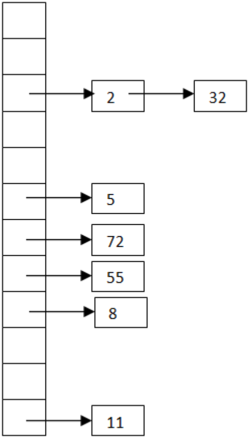
The hash value depends on the size of the hash table. If the size of the hash table changes, the value of the hash function changes as well.

- *search* and *remove* operations might not find the element.

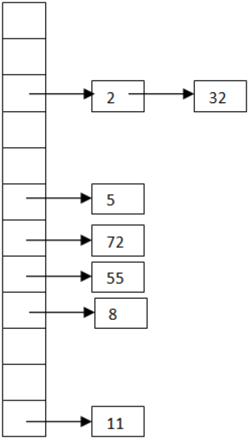
After resizing, we have to **rehash** the elements by adding them again in the resized hash table.

Example

- After rehashing and adding the other two elements:



Iterator



For the exemplified hash table, the easiest order in which the elements can be iterated is: 2, 32, 5, 72, 55, 8, 11.

Iterator

The iterator for a hash table with separate chaining is a combination of:

- an iterator on an array (table)
- an iterator on linked lists

We need a compound cursor consisting of:

- the current index in the table
- the pointer to the current node in the current linked list

Representation of the Iterator over a Hash Table with separate chaining:

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

Iterator - init

How can we implement the *init* operation?

The constructor of an iterator over a hash tables with separate chaining:

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

What is the time complexity?

Iterator - init

How can we implement the *init* operation?

The constructor of an iterator over a hash tables with separate chaining:

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

What is the time complexity?

✓ $O(m)$

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?

We can store the individual lists in a sorted order and for the iterator we can merge them.

Sorted containers

How can we define 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 merge them.

Hash tables are in general not very suitable for sorted containers.

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-world 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

SEPARATE

CHAINING

TABLE

HASH

THANK YOU