Dictionaries and Hash Tables

Nicolò Felicioni[1]

Dipartimento di Elettronica e Informazione
Politecnico di Milano

nicolo . felicioni @ polimi . it

May 25, 2021

[1]Based on Nicholas Mainardi's material.

Dictionaries

What is a dictionary?

- A dictionary is a collection of data which are accessible through a key in $O(1)$
- The key is usually an integer, which may be associated to arbitrary data, called satellite data
- Note that every raw data can be converted to an integer and used as a key (e.g. an ASCII string can be encoded as a base 128 integer)
- The key is generally unique

Naive implementation

- Suppose the keys are integers in the range $[0, \ldots, M-1]$
- Then, we can implement it with a simple array of size M
- This solution is known as **direct-address table**

Integer Sets and Direct-Address Tables

Dictionaries as Sets

- Sets are usually represented with a dictionary
- Indeed, elements of a set are unique, thus a unique key can be used to identify them
- In case the elements of a set are integers, the element is the key itself, no satellite data are necessary

Maximum on a Set of Integers

- We want to find the maximum of a set of integers ranging from $[0, \ldots, M-1]$
- Idea: with a direct-address table implementation, the index of the array is equal to the element of the set, thus the maximum is simply the last not empty element of the array
- We can easily look for it starting from $M-1$ position

Integer Sets and Direct-Address Tables

Can we get a more concise representation of a set of integers?

Reducing Spatial Complexity of Sets of Integers

- Consider a bit vector, which is an array where the single element is a bit
- Can we represent a set of integers with a bit vector?
- We can use the bit vector as a bit mask! Integer k belongs to the set if and only if the $k - th$ bit is 1!
- In this way, we represent a set with m bits instead of $m * w$, with w being the size of a word in the memory
- We can implement the bit vector as an array D with $\lceil \frac{m}{w} \rceil$ cells, each having w bits
- Then the $i - th$ element (e.g. $i - th$ bit) can be accessed by $(D[\lfloor \frac{i}{w} \rfloor] >> (i\%w))\&1$

Algorithms for Direct-Address Tables

Problem

Suppose to implement a dictionary with a direct-address table T, with the size of the table being huge. The table may contain garbage in its empty records, however we cannot initialize them to constant values since the table is too big. Is it possible to design a scheme which manages to perform basic operations $\mathrm{SEARCH}, \mathrm{INSERT}$ and DELETE in $O(1)$ on this huge table, with initialization cost being constant too?

- $\hookrightarrow$ It is possible to employ an additional array S, having the same size of the table T, to determine if a record is empty or not
- $\hookrightarrow$ The variable *size* stores the number of entries employed in S
- $\hookrightarrow$ Problem: How to efficiently ($O(1)$) search in this additional array if an element exists in the dictionary?

Algorithms for Direct-Address Tables

A Possible Solution

- Idea: We do not need to store the key in the record of the table, thus we can store the index of the additional array which contains information about this record
- How do we recognize an empty cell? It may be not sufficient to see that the index is a valid one (i.e. it is less than *size*) for the additional array S
  - $\hookrightarrow$ Indeed, a garbage value in a record may be a valid index in S
- Idea: the difference between a garbage access to S and a non-empty one is that in the latter case we know we have already accessed this cell from $T[k]$
- Thus, when we fill the additional array upon insertion of a key k, we can store k in the associated cell
- Following this strategy, we know that $\forall i \leq \mathit{size}(T[S[i]] = i)$

Algorithms for Direct-Address Table

Search, Insertion and Deletions

- Exploiting the previous property, an element with key k is in the dictionary, if and only if $T[k] \leq size \wedge S[T[k]] = k$. Indeed, for a key k' not in the dictionary:
  - Either $T[k'] > size$
  - Or $T[k'] \leq size$, but then $S[T[k']]$ has already been initialized with a key different from k'
- For insertion, it is sufficient to verify that the element k is not present and, if this the case, perform these operations:

  $$\begin{aligned} & size \leftarrow size + 1 \\ & S[size] \leftarrow k \\ & T[k] \leftarrow size \end{aligned}$$

- In the first insertion, we know for sure that k is empty, thus we do not need to search for it $\Rightarrow$ Initialization phase is $O(1)$

Deletions

- What about deletions? We can simply break the relations $S[T[k]] = k$ by writing a $k' \neq k$, but in this way we introduce a "hole" in the additional array!
  - $\hookrightarrow$ k' must be greater than any other key in the dictionary, otherwise we may insert k' by chance if $T[k'] = T[k]$
- Are holes a problem? Yes!
  - $\hookrightarrow$ Sooner or later, holes becomes the only free cells of S to insert new elements $\Rightarrow$ linear search time to find the "holes"!
- How to avoid "holes"? Shift all the cells of S has linear cost!
- Idea: we do not care about preserving the order of insertion of the keys in the additional array!
- Thus, we can replace the element to be erased with the last element being inserted, which is the one in $S[size]$:

```
    $T[S[size]] \leftarrow T[k]$
    $S[T[k]] \leftarrow S[size]$
    $size \leftarrow size - 1$
```

Hash Tables

- Direct-address tables are feasible only when the range of the keys to be stored is not too big
- We can use hash functions to map a wide range of keys to a fixed one, which is used as a direct-address table
- The new key is $h(k)$, with $H : U \mapsto [0, \ldots, M-1]$, with U being the universal set of keys, generally $\mathbb{N}$

Collisions

- Unless $|U| = M$, there necessarily exists $k_1, k_2(h(k_1) = h(k_2) \wedge k_1 \neq k_2)$
- In this case, we say that there is a collision between k_1 and k_2
- To solve such collisions, there are different methods, let's look at them through an example

Hash Tables: Example

- $M = 11$
- $K = [35, 83, 57, 26, 15, 63, 97, 46]$
- $h(k) = k \mod M$

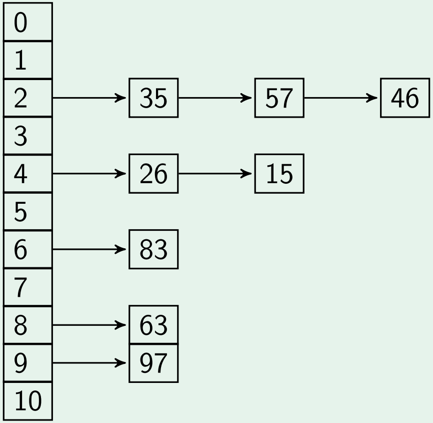
Hash Values

- $h(35) = 2$
- $h(83) = 6$
- $h(57) = 2$
- $h(26) = 4$
- $h(15) = 4$
- $h(63) = 8$
- $h(97) = 9$
- $h(46) = 2$

Hash Tables: Collisions

Chaining

- Each record of the table has a list of elements mapped to that record by the hash function *h*:



Hash Tables: Open Addressing

- With open addressing, if the record $h(k)$ is full, then an inspection sequence is used to find a free record to store k
- The hash function is replaced by $H : U \times [0, \dots, M-1] \mapsto [0, \dots, M-1]$, fulfilling $\forall i, j \in [0, \dots, M-1](H(k,i) = H(k,j) \Rightarrow i = j)$, that is the sequence must be a permutation of the records, to allow the key being stored in a free record eventually
- To store k, we compute $H(k,i)$, starting from $i = 0$, until we find a free record

Linear Probing

- In linear probing $H(k,i) = (h(k) + i) \mod 11$

0	1	2	3	4	5	6	7	8	9	10
		35	57	26	15	83	46	63	97	

Hash Tables: Open Addressing

- With linear probing, the keys are clustered around the records which have experienced collisions
- We should have a scheme placing elements not only in the closest records to $h(k)$

Quadratic Probing

- $H(k,i) = (h(k) + c_1 * i + c_2 * i^2) \mod m, c_2 \neq 0$
- For instance, $c_1 = 0, c_2 = 1 \Rightarrow H(K,i) = h(k) + i^2 \mod 11$

0	1	2	3	4	5	6	7	8	9	10
46		35	57	26	15	83		63	97	

However, with such a choice for c_1, c_2, the inspection sequence is not a permutation $\Rightarrow$ not all the records are visited! (e.g $H(k,4) = H(k,7)$, since $4^2 = 7^2 = 5 \mod 11$)

Hash Tables: Open Addressing

- How to choose feasible values for c_1, c_2?
- Unfortunately, if m is prime, then $c_2 \neq 0 \Rightarrow$ the inspection sequence is not a permutation
- Conversely, for $m = p^k$, where p is prime, choosing c_2 among multiples of p and c_1 among non-multiples of p makes the inspection sequence a permutation (e.g. $p = 7, m = 49$, $c_1 = 10$, $c_2 = 35$)

Quadratic Probing: Issues

- There is no longer clustering, but keys experiencing a collision always have the same inspection sequence, which may result in longer times to find a free record
- We need to get an inspection sequence which depends on the key itself!

Hash Tables: Open Addressing

Double Hashing

- Employ a second hash function, $h_2(k) = 1 + (k \mod (M-1)) = 1 + k \mod 10$
- Now, $H(k,i) = (h(k) + i \cdot h_2(k)) \mod 11$
- Hash Values:

k	35	83	57	26	15	63	97	46
$h_2(k)$	6	4	8	7	6	4	8	7

- Hash Table:

0	1	2	3	4	5	6	7	8	9	10
	46	35		26	15	83		63	97	57

Hash Tables: In-Place Chaining

Problem

We want to implement an hash table where collisions are solved by chaining. However, we do not want to allocate external memory for the elements, but we want to store them in the table itself in the free slots. Suppose we can use an hash table whose slots can store a flag and either an element followed by a pointer, or two pointers. We can use a list to store all the free slots. All the operations on the hash table and on the list must be $O(1)$

- Idea: we use a single slot to store the element and the pointer to the slot storing the next element in the chain
- The chain is finished when such a pointer is `NIL`
- The list storing free slots can be used as a stack $\Rightarrow$ PUSH and POP in $O(1)$
- How to perform $\mathrm{INSERT}, \mathrm{DELETE}$, SEARCH in $O(1)$?

Hash Tables: In-Place Chaining

SEARCH is trivial: search key k in the chain starting in slot $h(k)$

INSERT - 1

- The boolean flag is used to understand if the slot $l = h(k)$ is empty or not
- If l is empty, we can set the flag to true, store the element and set the pointer to `NIL`
  - NB: we don't erase l from free-slots list to save $O(n)$ look-up
  - Each time we pop a slot from the list, we must check that the slot is not full, popping the next slot in the list otherwise
- If l is full (i.e., it stores an element e), there are 2 cases:
  1. e is the head of a chain of elements $\Rightarrow$ e is in the "right" slot
  2. e is part of a chain of elements starting in another slot $\Rightarrow$ e is in the "wrong" slot
- How to distinguish between them? From the hash of the element being stored in l!

Hash Tables: In-Place Chaining

INSERT - 2

1. If $h(e.key) = l$, then it is case 1, thus we need to insert e at the head of this chain (as it is more efficient)
   - That is, we pop the first element of the free slots list, we copy the element being stored in l, then we overwrite l with e and set the pointer to the slot with the copied element
2. Conversely, if $h(e.key) \neq l$, then we need to start a new chain for slot l, moving the stored element somewhere else
   - That is, we pop the first element of the free slots list, with index *new*, we copy the element being stored in l, and then we insert e in l as if it was empty
   - Problem: the element initially stored in l was part of another chain, thus its predecessor in this chain still points to l, although there is no longer an element of this chain in l!
   - We search, in the chain starting at slot $h(e.key)$, the element which points to l, and then we replace this pointer with *new*

Hash Tables: In-Place Chaining

DELETE

- Given the key k of the element to be erased, we search this element in the hash table
- Denoting as l the slot where the element is found, there are 2 possible cases:
  1. $h(k) = l$, i.e., the element is the head of a chain $\Rightarrow$ Replace this slot with the content of the pointed one. Then, erase the old second element (whose index must be saved before overwriting slot l) and push its slot to the free slots stack
  2. $h(k) \neq l$, i.e., the element is in the middle of a chain $\Rightarrow$ Save the pointer to the next element, erase the content of this slot and push it to the free slots stack.
     - $\hookrightarrow$ As for INSERT, we move a slot being in the middle of a chain, thus the pointer of its predecessor must be updated with the address of its successor