CS 261-020
Data Structures

Lecture 14

Maps and Hash Tables (cont.)

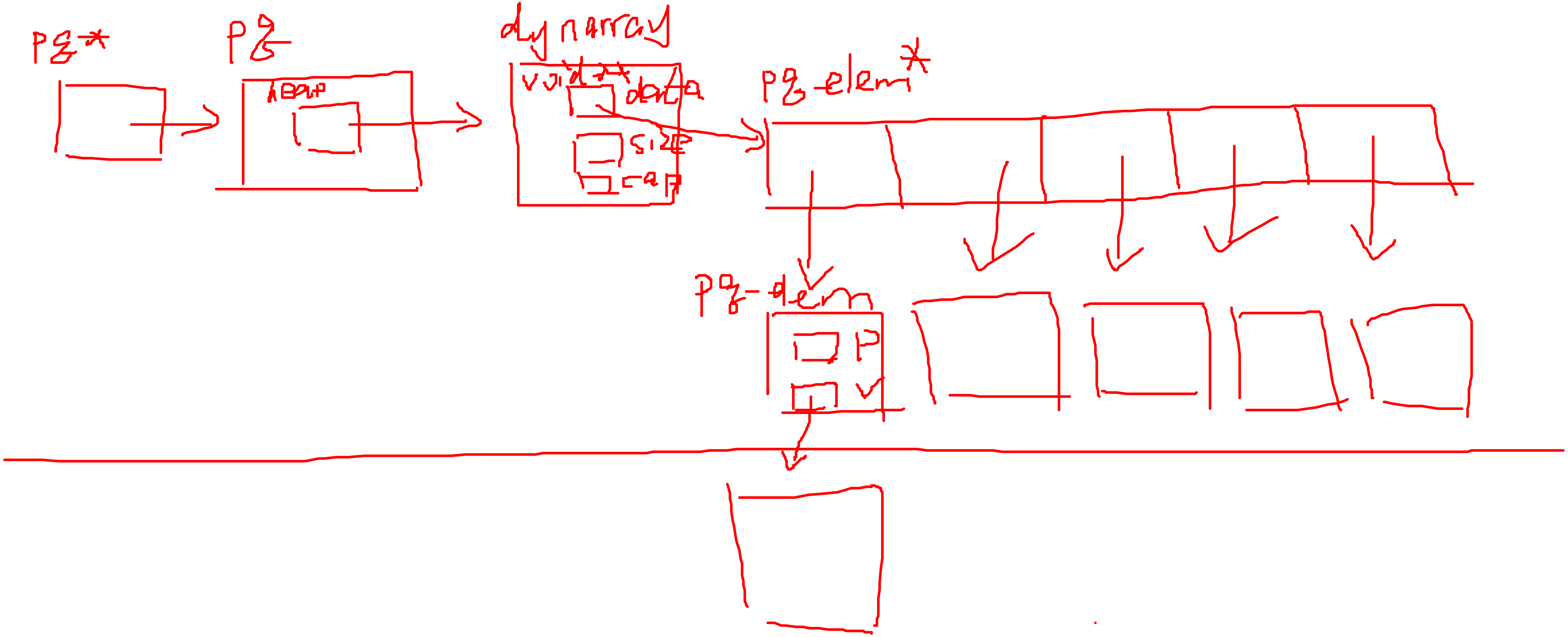
3/5/24, Tuesday

Oregon State University

Odds and Ends

- Assignment 5 will be posted tomorrow
- Recitation 9 posted

- Recitation 10: Mock Coding Interview (Proficiency Test)
  - Go to your registered section!!!

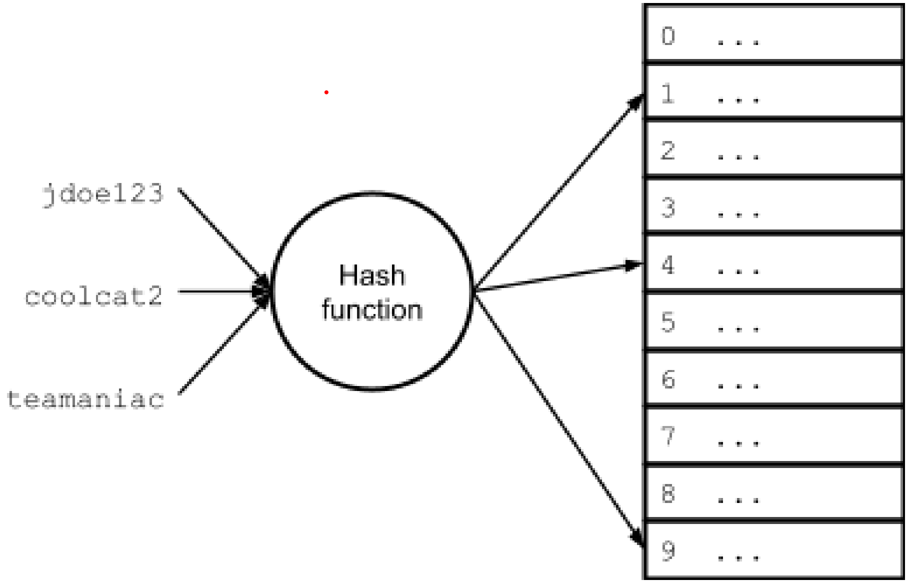

pq*
pq
heap
dynarray
void**
data
size
cap
pq_elem*
pq-elem
p
v

Review: Hash Tables

- A hash table is like an array (storing key/value structs), with a few important differences:
  - Elements can be indexed by values other than integers.
  - More than one element may share an index. (More later)

Key → Hash function → integer

```
hash = hash_function(key)
index = hash % array_size
```



Review: Hash Tables

- When choosing or designing a hash function, there are a few properties that are desirable:
  - **Determinism** – a given input should always map to the same hash value.
  - **Uniformity** – the inputs should be mapped as evenly as possible over the output range.
    - A non-uniform function can result in many collisions, where multiple elements are hashed to the same array index. (More later).
  - **Speed** – the function should have low computational burden

Review: Perfect and Minimally Perfect Hash Functions

- Collision: some keys map to the same index:
  - x != y, but hash(x) == hash(y)

- A perfect hash function is one that results in no collisions.

- A minimally perfect hash function is one that results in no collisions for a table size that exactly equals the number of elements.

Perfect and Minimally Perfect Hash Functions

- In practice, we don’t usually have such a nicely arranged situation, so it’s rare that our hash function will be minimally perfect.
  - For example, even with perfectly uniform random distribution of elements and a hash table with a capacity of 1 million elements, there is a 95% probability of a collision with only 2450 elements

- This means that, most likely, we’ll need to be able to deal with collisions

Collision Example:

- Hash function to store the words into a hash table.
- ```
  int string_hash(char* str) {
        return (int)(str[0] - 'a') % 6;
  }
  ```

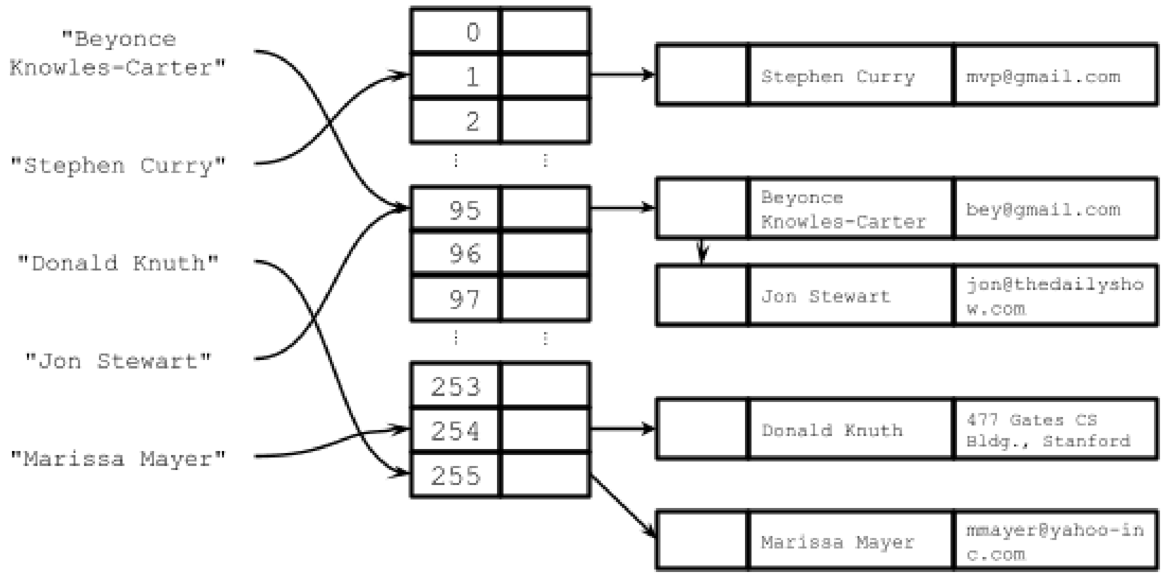
- "yummy"
  "delicious"
  "incredible"
  "fantastic"
  "exquisite"
  “date"

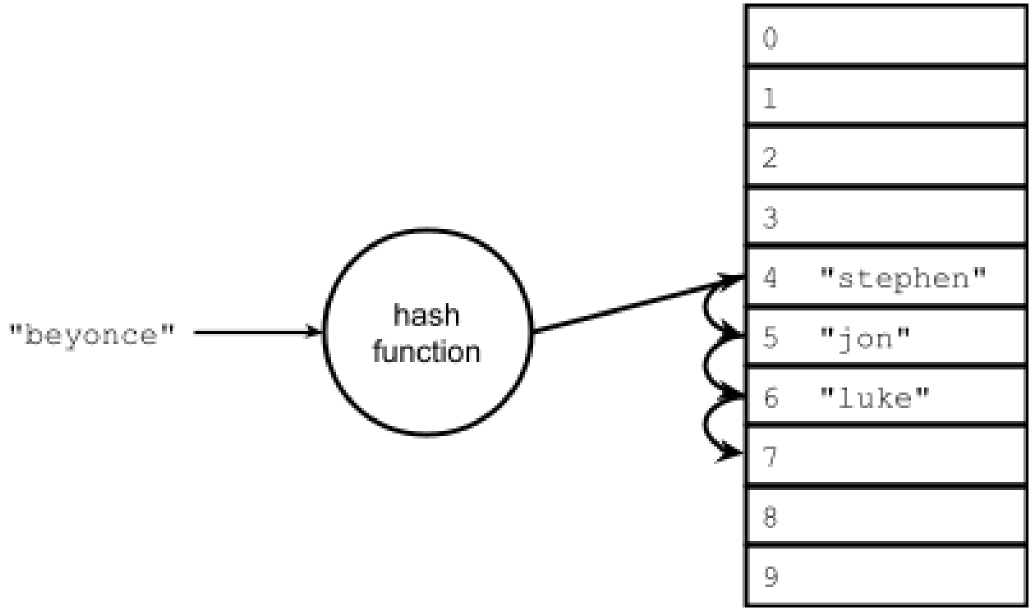
yummy	nonpareil	incredible	delicious		fantastic
  ```

Collision Resolution

Two mechanisms for resolving hash collisions

- Chaining
- Open addressing





1. Collision Resolution with Chaining

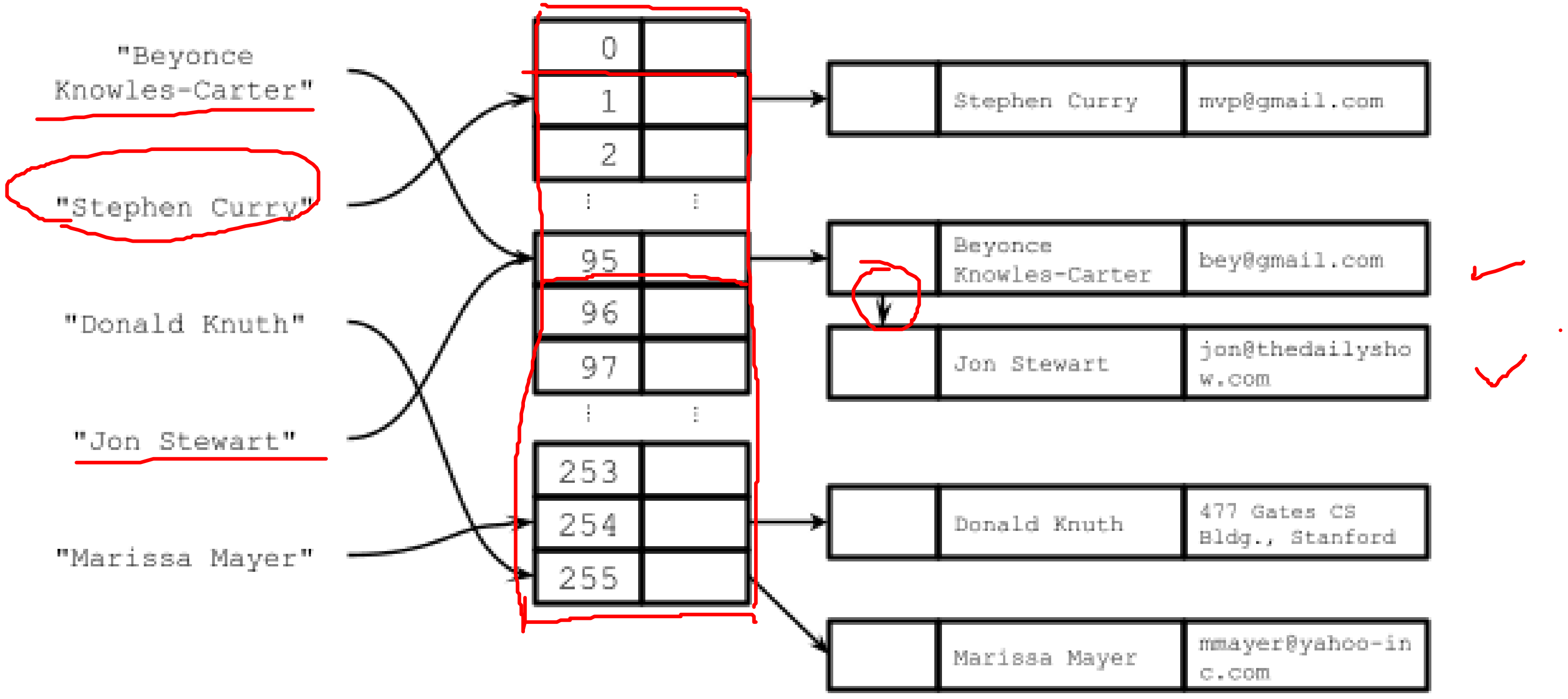
- The chaining method involves storing a collection of elements at each index in the hash table array.
  - Each collection is called a bucket or a chain.

- When a collision occurs, the new element is added to the collection at its corresponding hash index.

- Linked lists are a popular choice for maintaining the buckets themselves.
  - Other data structures could be used, e.g. a dynamic array or a balanced binary tree

1. Collision Resolution with Chaining

- Here's what a hash table with linked list-based chains might look like:



1. Collision Resolution with Chaining

- In a chained hash table,
- To loopup the value for a particular key:
  - Compute the element's bucket using the hash function
  - Search the data structure at that bucket for the element (using the key)
    - E.g. iterate through the items in the linked list.

- To add/remove an element:
  - Compute the element's bucket using the hash function
  - add or remove the element to/from the appropriate bucket's data structure
    - E.g. iterate through the items in the linked list.

1. Collision Resolution with Chaining

- **Load factor**: the average number of elements in each bucket:

$$\lambda = \frac{n}{m}$$

  - n is the total number of elements stored in the table
  - m is the number of buckets
  - **λ** Is the load factor
- In a chained hash table, the load factor can be greater than 1.
- As the load factor increases, operations on the table will slow down.
- For a linked list-based chained table,
  - For successful searches, the average number of links traversed is **λ** / 2.
  - For unsuccessful searches, the average number of links traversed is **λ**.

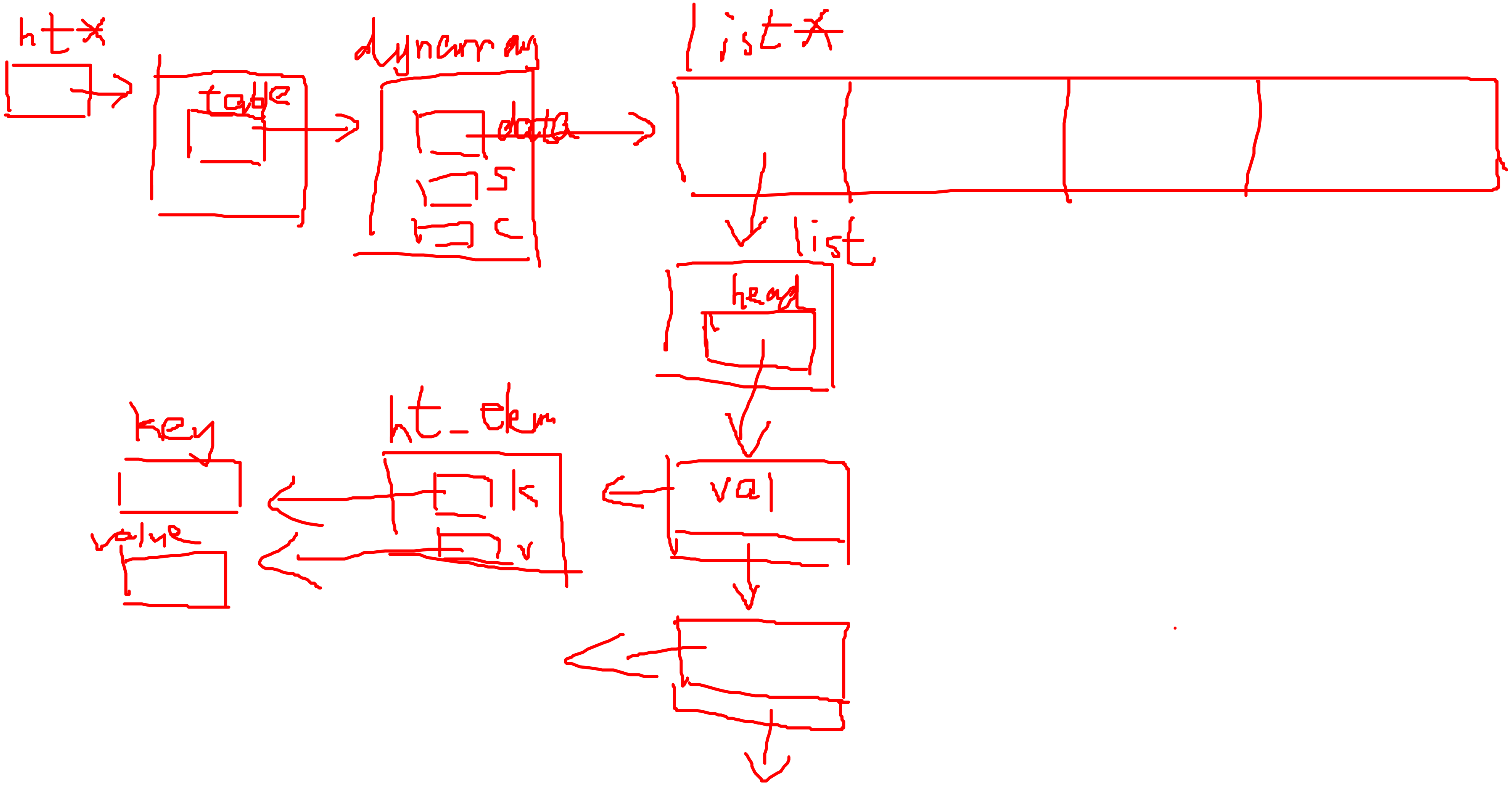
1. Collision Resolution with Chaining

- How to maintain the performance of the hash table?

- Double the number of buckets when the load factor reaches a certain limit (e.g. 8).
  - In other words, the hash table array could be implemented with a dynamic array whose resizing behavior is based on the load factor.

- How would we actually perform the resize?

- Re-compute the hash function for each element with the new number of buckets (i.e. using mod operator (%)).


ht*
table
dynarray
data
s
c
list*
list
head
val
ht_elem
k
v
key
value

1. Collision Resolution with Chaining

- What is the best-case complexity of a linked list-based chained hash table?
  - Assume that the hash function has a good distribution.
  - If the number of buckets is great than or equal to number of elements, i.e.: m >= n
  - Then, O(1)

- What is the worst-case complexity of a linked list-based chained hash table?
  - O(n), since all of the elements might end up in the same bucket.

1. Collision Resolution with Chaining

- What is the average-case complexity of a linked list-based chained hash table?
  - Assume that the hash function has a good distribution.
  - The average case for all operations is O(**λ**).
  - If the number of buckets is adjusted according to the load factor, then the number of elements is a constant factor of the number of buckets, i.e.:

$$\lambda = \frac{n}{m} = \frac{O(m)}{m} = O(1)$$

  - In other words, the average case performance of all operations can be kept to constant time.

2. Collision Resolution with Open Addressing

- The open addressing method: involves probing for an empty spot
  - Probing: the process of searching for an empty position.

- When using open addressing, all hashed elements are stored directly in the hash table array

- To insert an element:
  - Use the hash function to compute an initial index i for the element.
  - If the hash table array at index i is empty, insert the element there and stop.
  - Otherwise, increment i to the next index in the probing sequence (e.g. i + 1) and repeat

2. Collision Resolution with Open Addressing

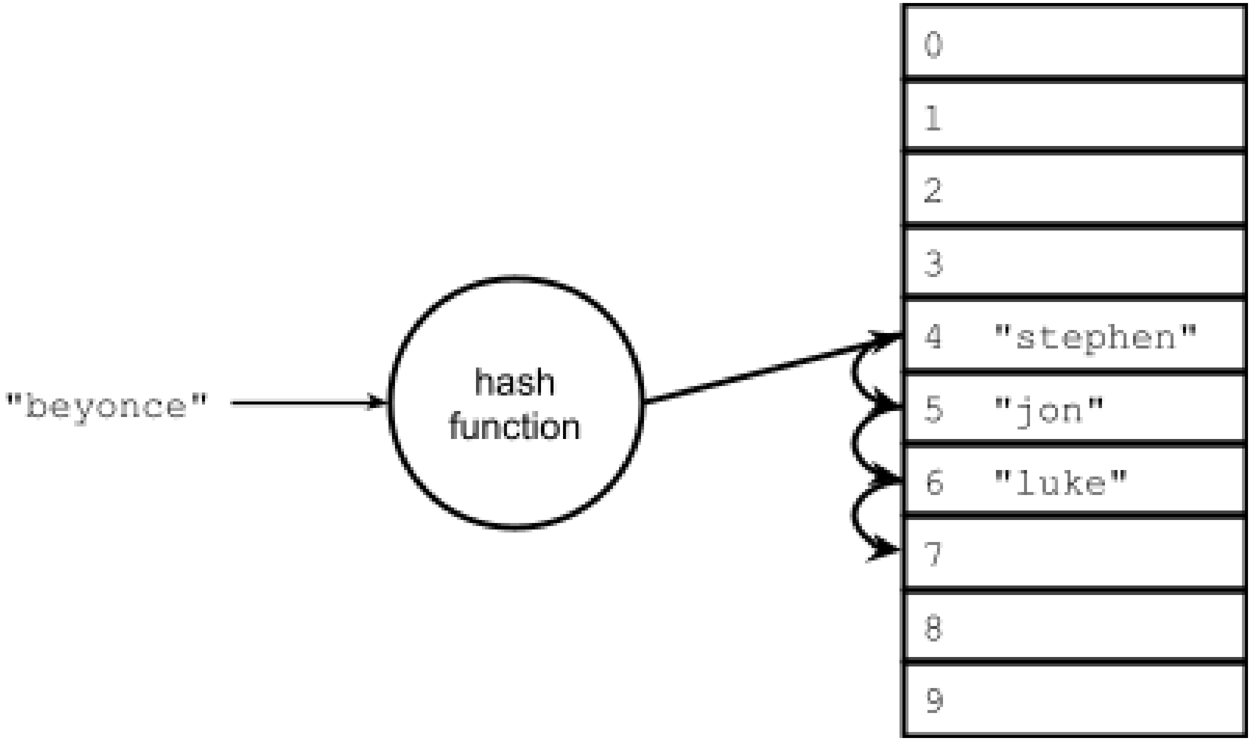
- Probing: the process of searching for an empty position.

- There are many different probing schemes:
  - Linear probing: $i = i + 1$
  - Quadratic probing: $i = i + j^2$ $(j = 1, 2, 3, \ldots)$
  - Double hashing: $i = i + j * h_2(key)$ $(j = 1, 2, 3, \ldots)$
    - Here, h_2 is a second, independent hash function.

2. Collision Resolution with Open Addressing

- For example, using linear probing, the key "beyonce" would be inserted at index 7, even though the hash function evaluates to 4 for that key:

beyonce



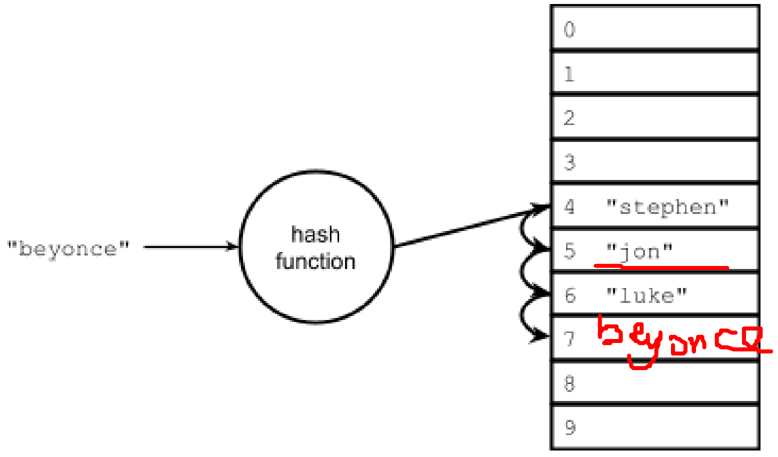
2. Collision Resolution with Open Addressing

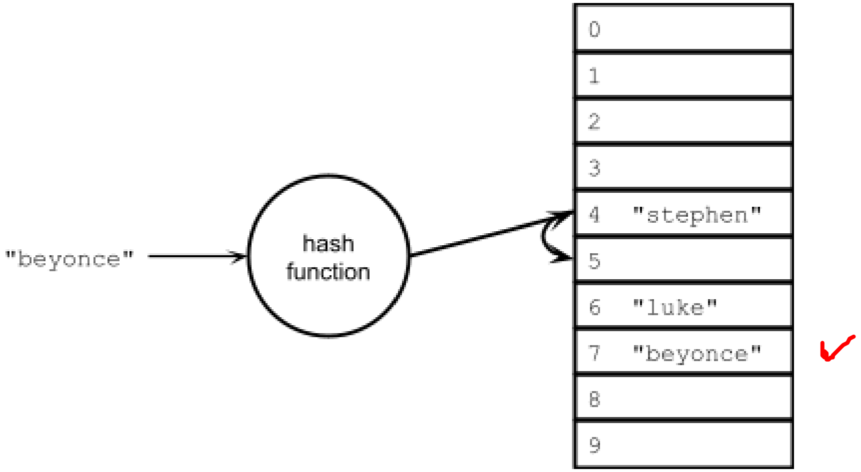
- To search for an element:
  - Use the hash function to compute an initial index i for the element
  - probe until we find either the element or an empty spot
    - If found an empty spot, then the element doesn’t exist

- What happens if we reach the end of the array while probing?
  - Simply wrap around to the beginning.

2. Collision Resolution with Open Addressing

- What happens when we remove an element?
  - Search for the element, then remove it
- What about searching after removing?
  - This could disrupt probing for elements after it.
- For example, what if we removed "jon" and then searched for "beyonce"?

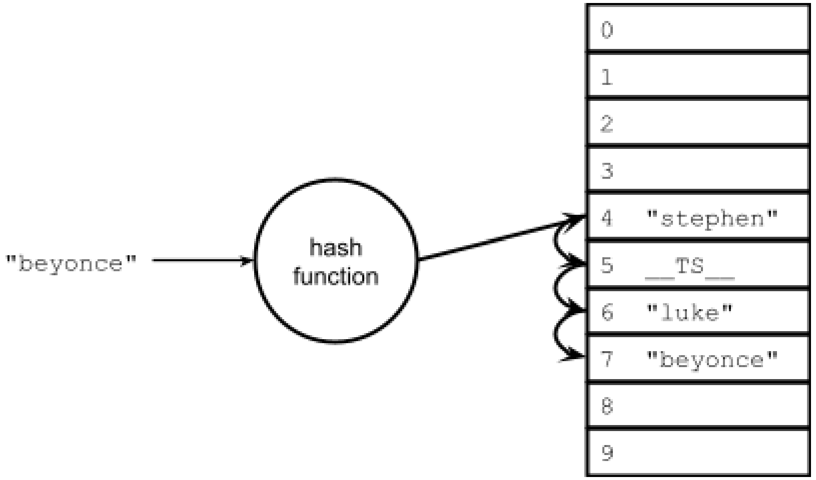




2. Collision Resolution with Open Addressing

- To get around this problem, we use a special value known as the tombstone
- Now, when an element is removed, we insert the tombstone value.
  - This value can be replaced when adding a new entry, but it doesn't halt search for an existing element.

- With a tombstone value __TS__ inserted for the removed "jon", the search above for "beyonce" could proceed as normal:



2. Collision Resolution with Open Addressing

- One problem: clustering, where elements are placed into the table into clusters of adjacent indices.
- For example, using linear probing, the probability of a new entry being added to an existing cluster increases as the size of the cluster increases
- ***larger cluster → more collision***

9, 0, 1, 2, 3

4, 5, 6, 7, 8

0	"elizabeth"
1	
2	
3	
4	"stephen"
5	"jon"
6	"luke"
7	"beyonce"
8	
9	"albert"

2. Collision Resolution with Open Addressing

- How to reduce clustering?
- By using quadratic probing and especially double hashing

- Using open addressing, a table's load factor cannot exceed 1.

- low load factor → avoid collisions
- low load factor → a lot of unused space

- In other words, there is a tradeoff between speed and space with open addressing.

2. Collision Resolution with Open Addressing

What is the complexity of open addressing? (Assuming truly uniform hashing)

- To insert a given item into the table (that's not already there):
- the probability (p) that the first probe is successful is

$$p = \frac{m-n}{m} = \frac{m}{m} - \frac{n}{m} = 1 - \lambda$$

  - There are m total slots and n filled slots, so m - n open spots.

2. Collision Resolution with Open Addressing

What is the complexity of open addressing? (Assuming truly uniform hashing)

- If the first probe fails, the probability that the second probe succeeds is

$$\frac{m-n}{m-1} \geq \frac{m-n}{m} = p$$

  - There are still m - n remaining open slots, but now we only have a total of m - 1 slots to look at, since we've examined one already.

2. Collision Resolution with Open Addressing

What is the complexity of open addressing? (Assuming truly uniform hashing)

- If the first two probes fail, the probability that the third probe succeeds is

$$\frac{m-n}{m-2} \geq \frac{m-n}{m} = p$$

  - There are still m - n remaining open slots, but now we only have a total of m - 2 slots to look at, since we've examined two already.

- And so forth. In other words, for each probe, the probability of success is at least p

2. Collision Resolution with Open Addressing

What is the complexity of open addressing? (Assuming truly uniform hashing)

- The expected number of probes until success is: (a geometric distribution)

$$\frac{1}{p} = \frac{1}{\frac{m-n}{m}} = \frac{1}{1-\frac{n}{m}} = \frac{1}{1-\lambda}$$

- In other words, the expected number of probes for any given operation is O($\frac{1}{1-\lambda}$).

Collision Resolution with Open Addressing

The expected number of probes for any given operation is $O(\frac{1}{1-\lambda})$.

- If we limit the load factor to a constant and reasonably small number, our operations will be O(1) on average.
- E.g. if we have $\boldsymbol{\lambda}$ = 0.75, then we would expect 4 probes, on average. For $\boldsymbol{\lambda}$ = 0.9, we would expect 10 probes.