Open Addressing & Linear Probing

Implementing a hash map

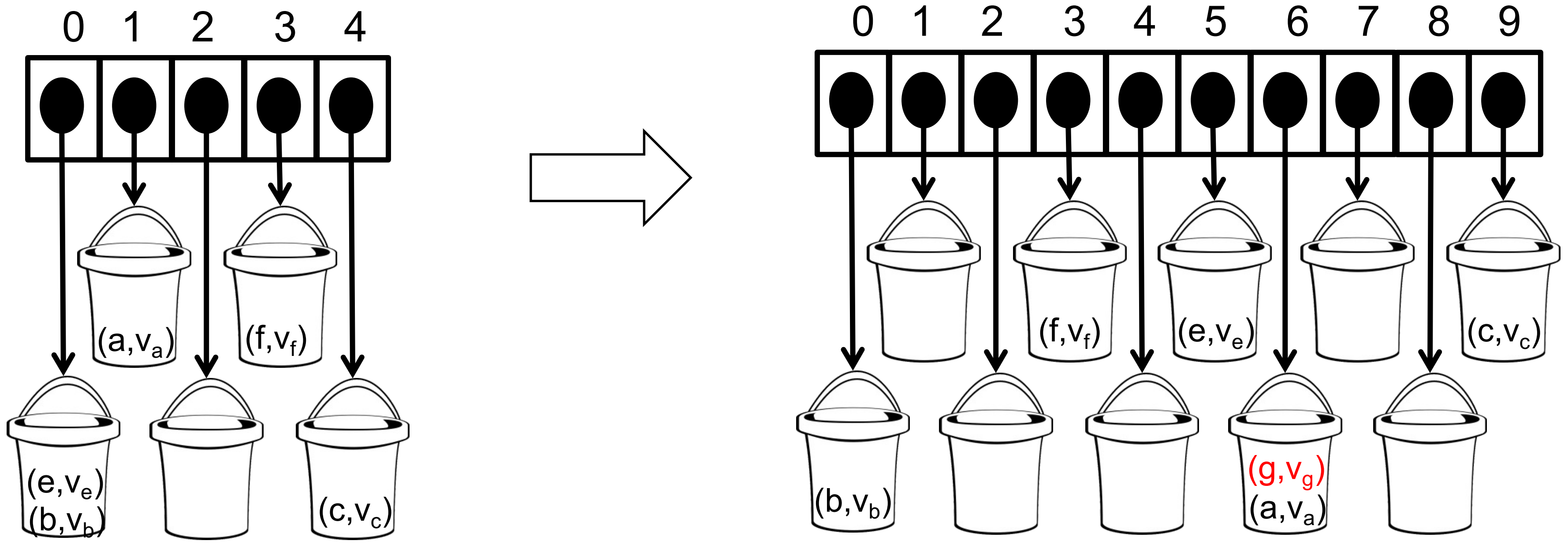
Open Addressing and Linear Probing

(last lecture of new material)

Recap

Items are stored with a *key,* which is used for lookup.
Each key must be unique.
Store (key,value) pairs in an array of known size N.
A *hash function* generates an integer for each key.
A *compression function* generates an index in the array for each hashed key.
Items receiving same index are stored in a collection at that cell – needs search.
If too many items, ‘search’ may be too expensive

- get bigger array, re-hash/compress, and move into new array
- if ‘too many’ threshold chosen wisely, gives expected lookup in time O(1)



map.setitem('g’, v_g)

Hashing and compression should distribute the keys evenly across the cells

Implementations (I)

Version 1: fixed size list, bucket array

```
# comphash(key) = hash(key) % len(list)

getitem(key):
    if there is a bucket at list[comphash(key)]
        search the bucket for the key and return value or None
    return None

setitem(key, value):
    if there is a bucket at list[comphash(key)]
        search the bucket for the key
        if key found, update with new value
        else append new Element to bucket and increment size
    else start a new bucket with new Element and increment size

delitem(key):
    if there is a bucket at list[comphash(key)]
        search the bucket for the key
        if key found,
            decrement size
            remove the element and return its value
    return None
```

Implementations (II)

Version 2: dynamic size list, bucket array

```
# comphash(key) = hash(key) % len(list)

setitem(key, value):
    if there is a bucket at list[comphash(key)]
        search the bucket for the key
        if key found, update with new value
        else append new Element and increment size
    else start a new bucket with new Element and increment size
    if size > f(len(list))
        resize by factor c

resize(c):
    oldlist = list
    list = new list with c None items
    size = 0
    for each bucket in oldlist
        for each element in bucket
            add the element into the (list) hash table
            size +=1
```

Flat arrays and open addressing

The separate chaining method and the bucket array requires the use of additional data structures (e.g. lists)for the buckets, which take up extra space.

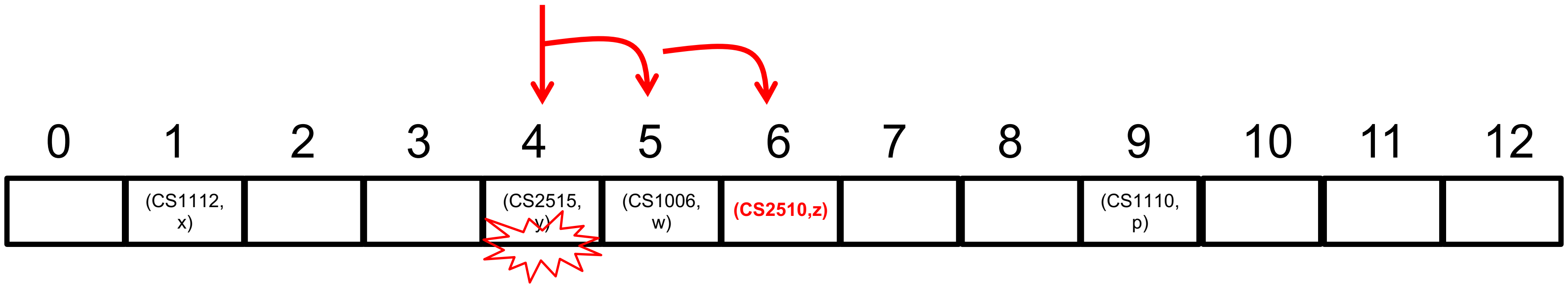
In *open addressing*, we store each element in a separate cell in the list. The cell cannot be uniquely determined by the compressed hash because of collisions (so the address is *open*). So how do we determine the address?

We need to find a policy that will:

- determine where we put a new (k,v) pair in all cases
- allow us to find a (k,v) pair if it is already in the structure
- do this efficiently – i.e. maintain expected O(1) complexity

We need a *policy* that can be applied in all cases, and that allows us to find items on lookup, while avoiding efficiency problems ...

Where do we add (CS2510, ...), which hashes & compresses to 4?



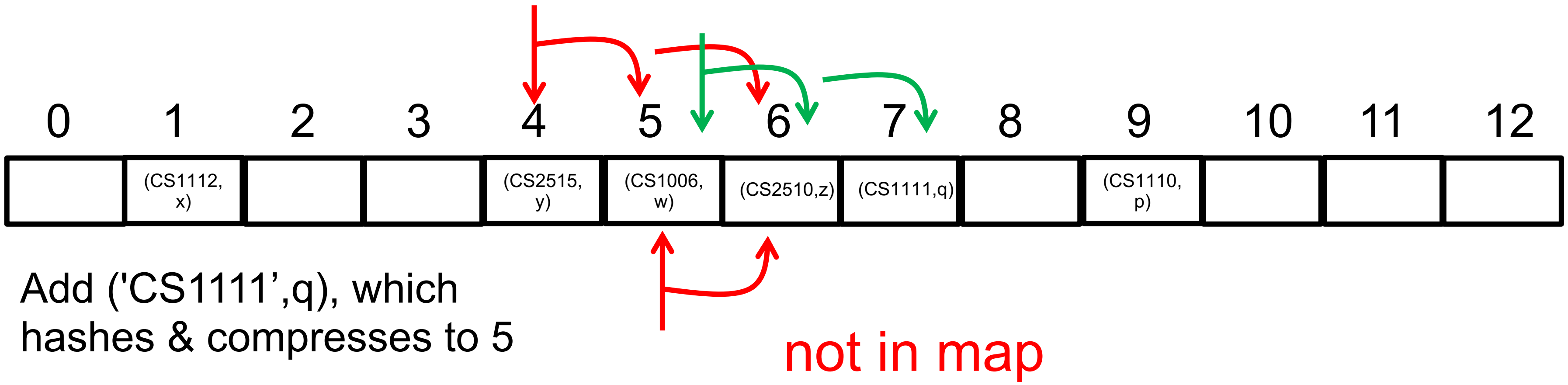
Linear probing (?)

To set an element, we search right from the compressed hash value until:

- we find the key, and we update the value, or
- we find an empty cell, and we add the new element

To delete an element, we search right from the compressed hash until:

- we reach an empty cell, and do nothing (element not in map), or
- we find the key, and we delete the element



Add ('CS1111',q), which hashes & compresses to 5

Delete CS2510.

But what happens now on a search for ('CS1111')?

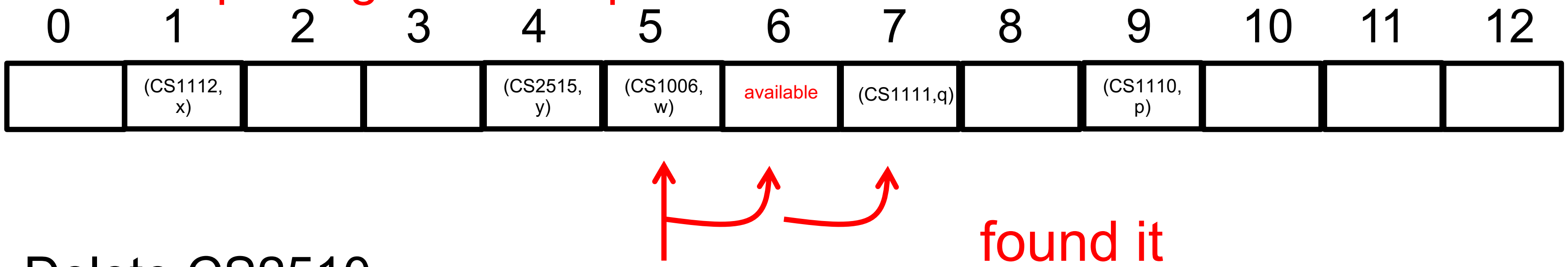
Linear probing

To set an element, we search right from the compressed hash value, remembering the first available cell, until:

- we find the key, and we update the value, or
- we find an empty cell, and we add the new element in what was the first available cell, or in the empty cell if no availables found

To delete an element, we search right from the compressed hash until:

- we reach an empty cell, and do nothing (element not in map), or
- we find the key, and we delete the element,
  - replacing it with a special '*available*' marker



found it

Delete CS2510.

What happens on a search for ('CS1111', ...)?

Linear Probing (2)

All values must get the same opportunity to be added

- 'search right' must wrap round the end of the list and continue from the beginning, for up to *n* steps in total
- use modular arithmetic
  - comphash(k) % N, then (comphash(k) + 1) % N, (comphash(k) + 2) %N, ...

The load factor (i.e. n/N) must never get above 1

- Why not?
- standard practice is to grow the list when n/N = 0.5
- rather than double the size, grow to 2N-1, or 2N+1
  - more chance of new size being a prime number ...
- should also shrink when more space than needed
- as with bucket array, all items must be re-hashed & compressed

Implementations (III)

Version 3: open addressing, linear probing, dynamic size

```
# comphash(key) = hash(key) % len(list)
getitem(key):
    if there is something in list[comphash(key)]
        search & wrap from there for the key and return value or None
    return None

setitem(key, value):
    if there is something in list[comphash(key)]
        search & wrap from there for the key, and remember “available”
        if key found, update with new value
        else insert in first available or none; increment size
    else insert and increment size
    if size > f(len(list))
        resize by factor c

resize(c):
    oldlist = list
    list = new list with c None items
    for each cell in oldlist
        if cell has (k,v) insert (k,v) into the (list) hash table
            size +=1
```

Implementations (III cont.)

```
delitem(key):
    if there is something in list[comphash(key)]
        search & wrap from there, using probing scheme, for the key
        if key found
            replace with “available”; decrement size
    if size < g(len(list))
        resize by factor c

#Note – not particularly efficient – list can get filled up with
#”available” markers – should also maintain a count of them, and
# when too many, re-compress all entries, removing ‘avaialble’
```

Other probing schemes

Linear probing tends to produce long chains of occupied cells, and so searching tends to O(n)

Quadratic probing tries the cell

(comphash(k) + i^2) % N for each search step i

Double hashing tries the cell

(comphash(k) + i*h'(k)) % N for each search step i, where h'() is another hash function.

Python's strategy

Pseudo-random probing tries the cell

(comphash(k) + f(i)) % N, for each search step i, where f(i) is determined by a pseudo-random number generator

Complexity

	getitem(k)	contains()	setitem(k,v)	delitem()	build full map
unsorted list	O(n)	O(n)	O(n)	O(n)	$O(n^2)$
sorted list	O(log n)	O(log n)	O(n)	O(n)	$O(n^2)$
AVL Tree	O(log n)	O(log n)	O(log n)	O(log n)	O(n log n)

Hash Table
expected:
	O(n)
	O(1)
		O(n)
		O(1)
			O(n)
			O(1)
				O(n)
				O(1)
					$O(n^2)$
					O(n)

Note: for most uses, choose Python's dict data structure (i.e. a hash table) – it is almost always faster.

Exercise: how would you implement a *sorted dictionary* ?

Next lecture

Revision