Hashing

Sets and Dictionaries

What do we use arrays for?

1 To keep a *collection* of elements of the same type in one place

- *E.g., all the words in the Collected Works of William Shakespeare*

"a"	"rose"	"by"	"any"	"name"	…	"Hamlet"

- The array is used as a **set**
  - the index where an element occurs doesn't matter much
- Main operations:
  - add an element
    - like uba_add for unbounded arrays
  - check if an element is in there
    - this is what search does (linear if unsorted, binary if sorted)
  - go through all elements
    - using a for-loop for example

What do we use arrays for?

2 As a *mapping* from indices to values

- *E.g., the monthly average high temperatures in Pittsburgh*

	0	1	2	3	4	5	6	7	8	9	10	11	12
High:	X	35	38	50	62	72	80	83	82	75			

0 = *unused*
1 = Jan
…
12 = Dec

- The array is used as a **dictionary**
  - value is associated to a specific index
  - the indices are critical
- Main operations:
  - **insert**/update a value for a given index
    - *E.g., High[10] = 63 -- the average high for October is 63°F*
  - **lookup** the value associated to an index
    - *E.g., High[3] -- looks up the average temperature for March*

Dictionaries, beyond Arrays

- Generalize index-to-value mapping of arrays so that
  - index does not need to be a contiguous number starting at 0
  - in fact, index doesn't have to be a number at all
- A **dictionary** is a mapping from keys to values

key

$$k \mapsto e$$

entry if *e* contains *k*
value otherwise

  - e.g.: mapping from month to high temperature (*value*)

    key "march" $\mapsto$ 50 value

  - e.g.: mapping from student id to student record (*entry*)

    key "bovik" $\mapsto$ ("Harry", "Bovik", "bovik", "1989") entry

  - arrays: index *3* is the key, contents *A[3]* is the value

    key 3 $\mapsto$ A[3] value

Dictionaries

key **entry**

$$k \longmapsto e$$

- Contains at most one entry associated to each key

- main operations:
  - create a **new** dictionary
  - **lookup** the entry associated with a key
    - or report that there is no entry for that key
  - **insert** (or update) an entry

  (we will consider only these)

- many other operations of interest
  - delete an entry given its key
  - number of entries in the dictionary
  - print all entries, …

Dictionaries in the Wild

- Dictionaries are a primitive data structure in many languages
  - Like arrays in C0
  - E.g.,
    - Python
    - Javascript
    - PHP, …

Sample PHP session

Linux Terminal

```
# php -a
php > $A[0] = 3;
php > echo $A[0];
3
php > $A[15122] = 11;
php > echo $A[15122];
11
php > echo $A[3];
PHP Notice:  Undefined offset: 3 in php shell code on line 1
php > $A["hello world"] = 13;
```

- They are not primitive in low level languages like C and C0
  - We need to implement them and provide them as a library
  - This is also what we would do to write a Python interpreter

Implementing Dictionaries

- based on what we know so far …
  - worst-case complexity assuming the dictionary contains *n* entries

Binary search

Move other elements out of the way

Linear search

Linear search on list

	unsorted array with (key, value) data	***(key, value) array sorted by key***	***linked list with (key, value) data***
lookup	*O(n)*	*O(log n)*	*O(n)*
insert	*O(1) amortized*	*O(n)*	*O(1)*

adding to an unbounded array

Add to the front of the list

  - **Observation**: operations are fast when we know where to look
- **Goal**: efficient lookup and insert for large dictionaries
  - about O(1)

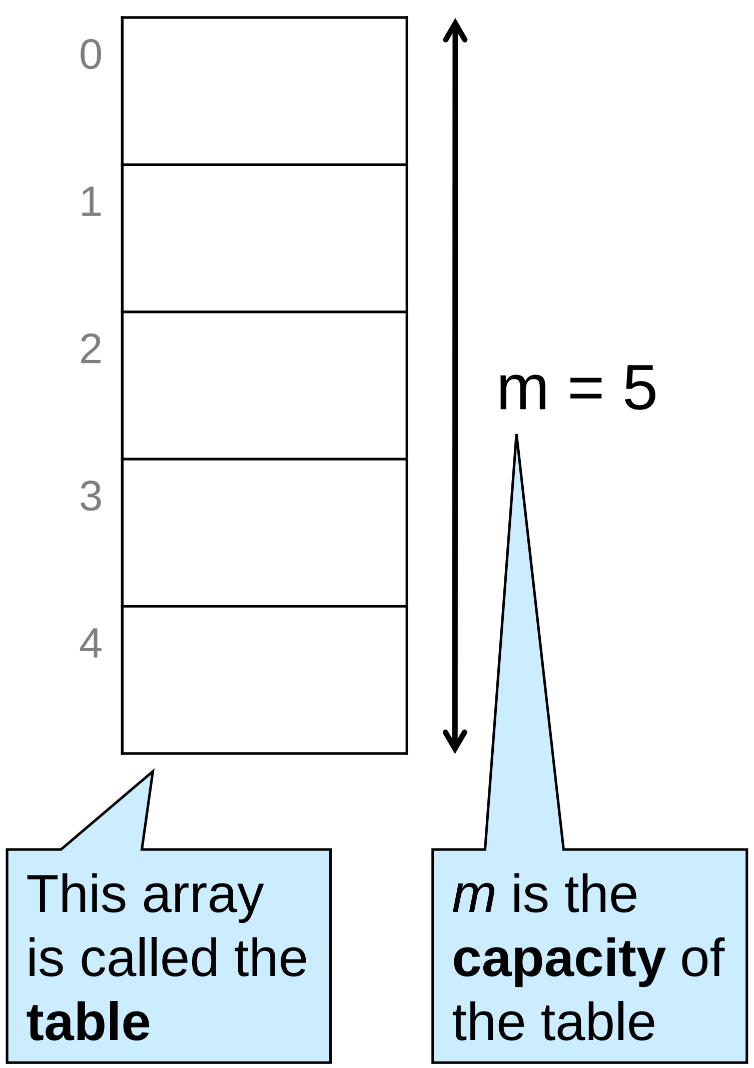
Dictionaries with Sparse Numerical Keys

Example

A dictionary that maps zip codes (keys) to neighborhood names (values) for the students in this room

- zip codes are 5-digit numbers -- e.g., 15213
  - use a 100,000-element array with indices as keys?
  - possibly, but most of the space will be wasted:
    - only about 200 students in the room
    - only some 43,000 zip codes are currently in use
- Use a much smaller *m*-element array
    - here m=5
  - reduce key to an index in the range [0,m)
    - here reduce a zip code to an index between 0 to 4
    - do zipcode % 5

- This is the first step towards a **hash table**



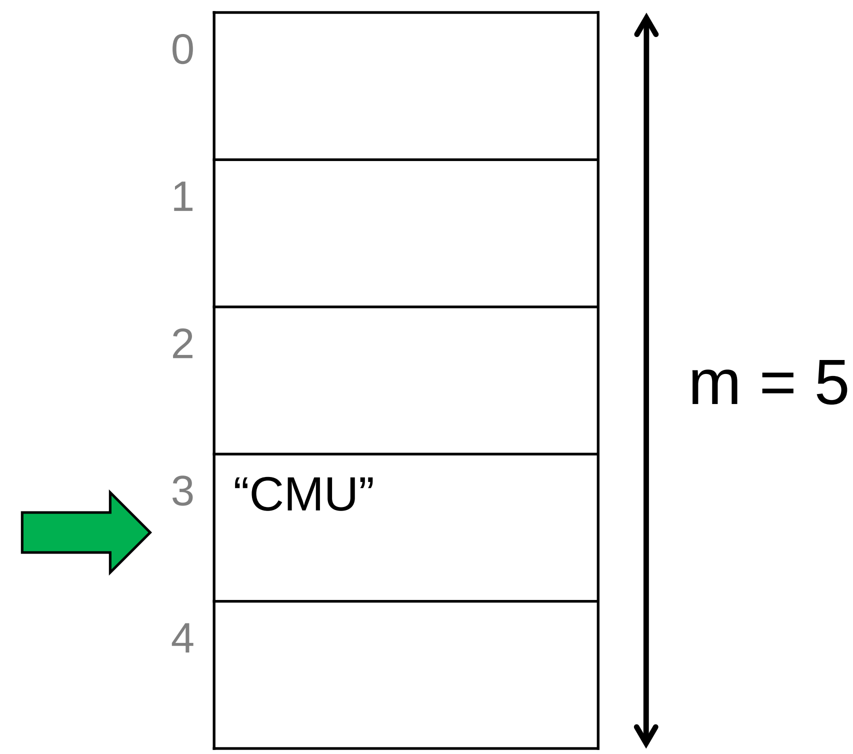
insert (15213, “CMU”)
insert (15122, “Kennywood”)
lookup 15213
lookup 15219
lookup 15217
insert (15217, “Squirrel Hill”)
lookup 15217
lookup 15219

Example

- We now perform a sequence of insertions and lookups

key value

  - o insert (15213, “CMU”)
    - ➢ compute table index as 15213 % 5 = 3
      - ❑ insert “CMU” at index 3



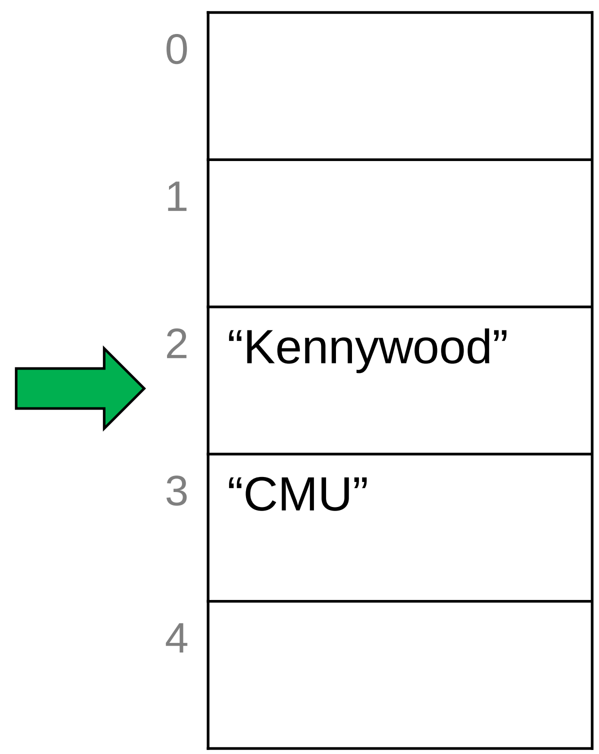
Example

insert (15213, “CMU”)
insert (15122, “Kennywood”)
lookup 15213
lookup 15219
lookup 15217
insert (15217, “Squirrel Hill”)
lookup 15217
lookup 15219

key

value

- insert (15122, “Kennywood”)
  - compute table index as 15122 % 5 = 2
    - insert “Kennywood” at index 2



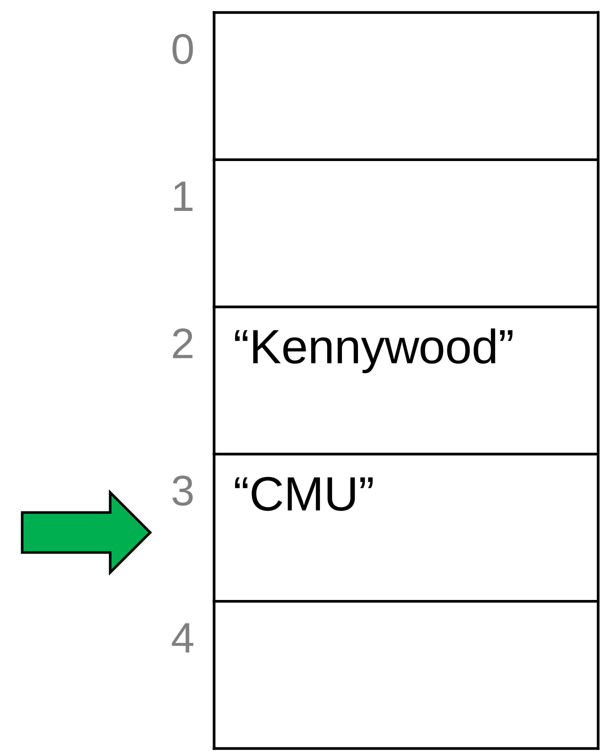
Example

insert (15213, “CMU”)
insert (15122, “Kennywood”)
lookup 15213
lookup 15219
lookup 15217
insert (15217, “Squirrel Hill”)
lookup 15217
lookup 15219

key

- lookup 15213
  - compute table index as 15213 % 5 = 3
    - return contents of index 3
      - “CMU”

value



Example

insert (15213, “CMU”)
insert (15122, “Kennywood”)
lookup 15213
lookup 15219
lookup 15217
insert (15217, “Squirrel Hill”)
lookup 15217
lookup 15219

key

- lookup 15219
  - compute table index as 15219 % 5 = 4
    - nothing at index 4
    - report there is no value for 15219

no value

0	
1	
2	“Kennywood”
3	“CMU”
4	

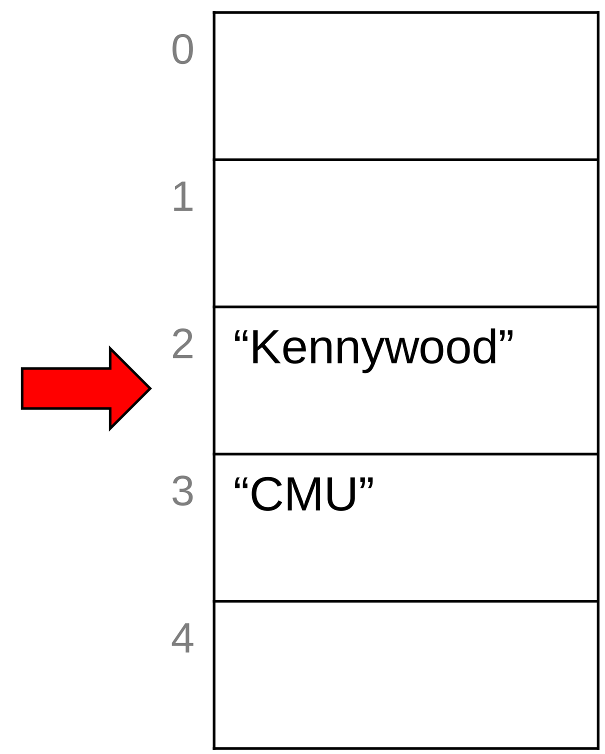
Example

insert (15213, “CMU”)
insert (15122, “Kennywood”)
lookup 15213
lookup 15219
lookup 15217
insert (15217, “Squirrel Hill”)
lookup 15217
lookup 15219

key

- lookup 15217
  - compute table index as 15217 % 5 = 2
    - return contents of index 2
      - “Kennywood”

value



- This is **incorrect**!
  - we never inserted an entry with key 15217
  - it should signal there is no value

We need to store **both** the **key** and the **value** -- the whole **entry**

Example

insert (15213, "CMU")
insert (15122, "Kennywood")
lookup 15213
lookup 15219
lookup 15217
insert (15217, "Squirrel Hill")
lookup 15217
lookup 15219

key

- lookup 15217
  - compute table index as 15217 % 5 = 2
    - check the key at index 2
      15122 ≠ 15217
    - entry at index 2 is not about this key ✘

no value for 15217

0	
1	
2	(15122, "Kennywood")
3	(15213, "CMU")
4	

- lookup now returns a whole entry

Example

insert (15213, “CMU”)
insert (15122, “Kennywood”)
lookup 15213
lookup 15219
lookup 15217
insert (15217, “Squirrel Hill”)
lookup 15217
lookup 15219

key

- insert (15217, “Squirrel Hill”)
  - compute table index as 15217 % 5 = 2
    - there is an entry in there
    - check its key

      15122 ≠ 15217 ✗
    - entry at index 2 is not about this key

0	
1	
2	(15122, “Kennywood”)
3	(15213, “CMU”)
4	

- We have a **collision**
  - different entries map to the same index

Dealing with Collisions

Two common approaches

- **Open addressing**
  - if table index is taken, store new entry at a predictable index nearby
    - **linear probing**: use next free index (modulo m)
    - **quadratic probing**: try table index + 1, then +4, then +9, etc.
- **Separate chaining**
  - do not store the entries in the table itself but in **buckets**
    - bucket for a table index contain all the entries that map to that index
    - buckets are commonly implemented as **chains**
      - a chain is a NULL-terminated linked list

Collisions are Unvoidable

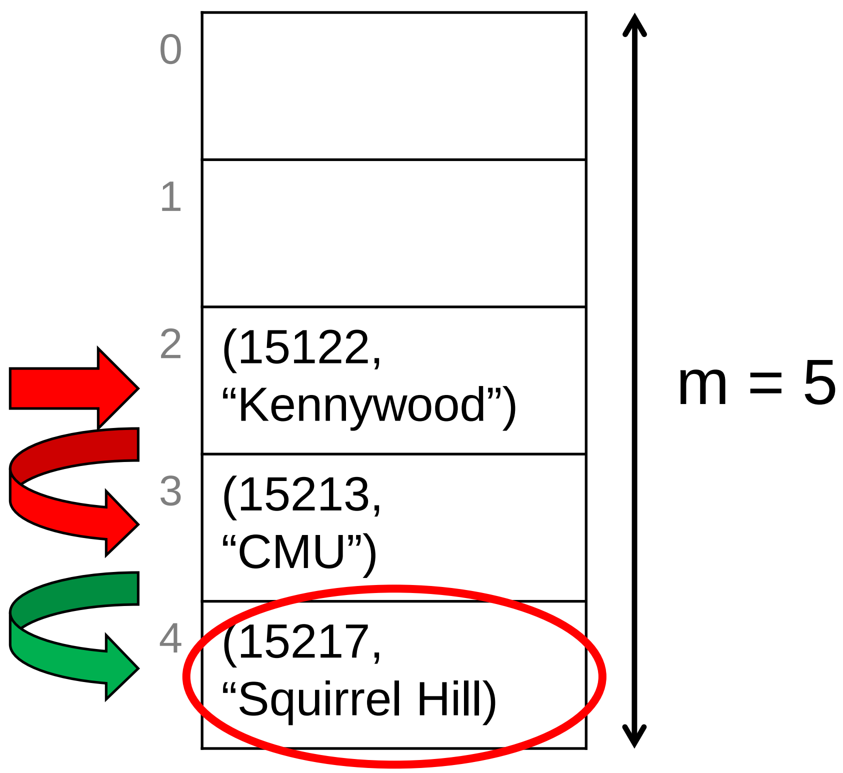
- If n > m
  - **pigeonhole principle**
    - *“If we have n pigeons and m holes and n > m, one hole will have more than one pigeon”*
  - This is a certainty
- If n > 1
  - **birthday paradox**
    - *“Given 25 people picked at random, the probability that 2 of them share the same birthday is > 50%”*
  - This is a probabilistic result

Example, continued with linear probing

insert (15213, “CMU”)
insert (15122, “Kennywood”)
lookup 15213
lookup 15219
lookup 15217
insert (15217, “Squirrel Hill”)
lookup 15217
lookup 15219

key

- insert (15217, “Squirrel Hill”)
  - compute table index as 15217 % 5 = 2
    - there is an entry in there
    - check its key: 15122 ≠ 15217 ✗
  - try next index, 3
    - there is an entry in there
    - check its key: 15213 ≠ 15217 ✗
  - try next index, 4
    - there is no entry in there ✓
    - insert (15217, “Squirrel Hill”) at index 4



Example, continued with linear probing

insert (15213, “CMU”)
insert (15122, “Kennywood”)
lookup 15213
lookup 15219
lookup 15217
insert (15217, “Squirrel Hill”)
lookup 15217
lookup 15219

key

- Lookup 15217
  - compute table index as 15217 % 5 = 2
    - there is an entry in there
    - check its key: 15122 ≠ 15217 ✗
  - try next index, 3
    - there is an entry in there
    - check its key: 15213 ≠ 15217 ✗
  - try next index, 4
    - there is an entry in there
    - check its key: 15217 = 15217 ✓
    - return (15217, “Squirrel Hill”)

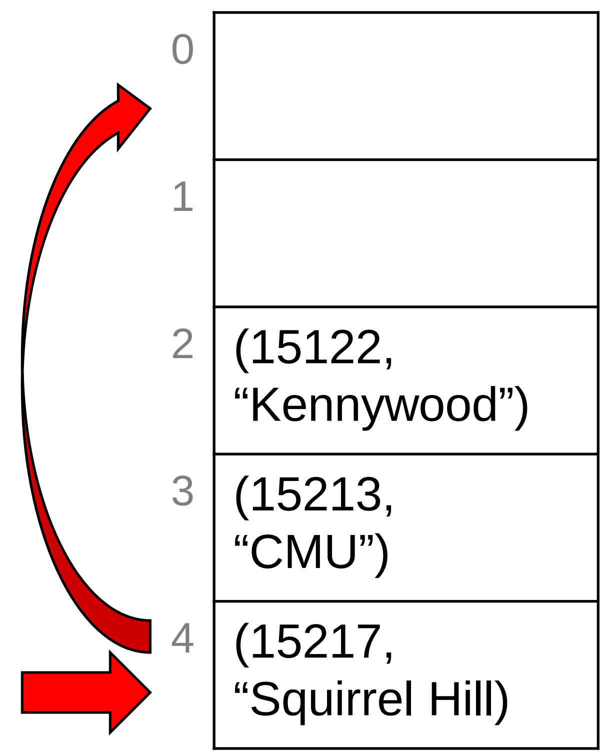
0	
1	
2	(15122, “Kennywood”)
3	(15213, “CMU”)
4	(15217, “Squirrel Hill)

Example, continued with linear probing

insert (15213, “CMU”)
insert (15122, “Kennywood”)
lookup 15213
lookup 15219
lookup 15217
insert (15217, “Squirrel Hill”)
lookup 15217
lookup 15219

key

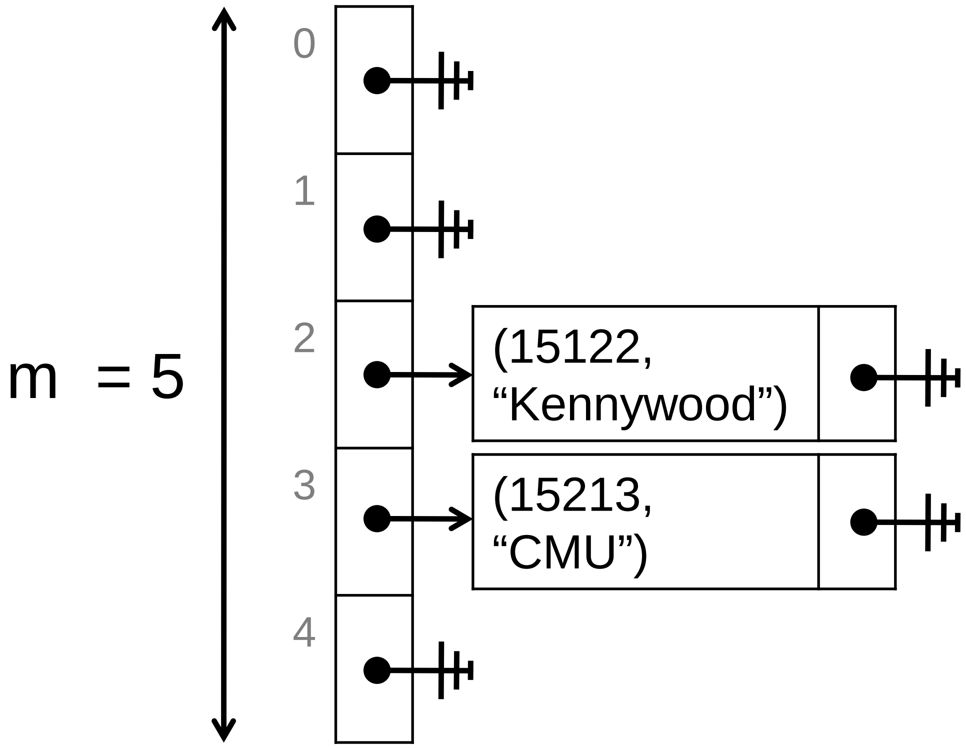
- Lookup 15219
  - compute table index as 15219 % 5 = 4
    - there is an entry in there
    - check its key: 15217 ≠ 15219 ✘
  - try next index, 5 % 5 = 0
    - there is no entry in there
    - report there is no entry for 15219 ✘



Example, continued with separate chaining

insert (15213, “CMU”)
insert (15122, “Kennywood”)
lookup 15213
lookup 15219
lookup 15217
insert (15217, “Squirrel Hill”)
lookup 15217
lookup 15219

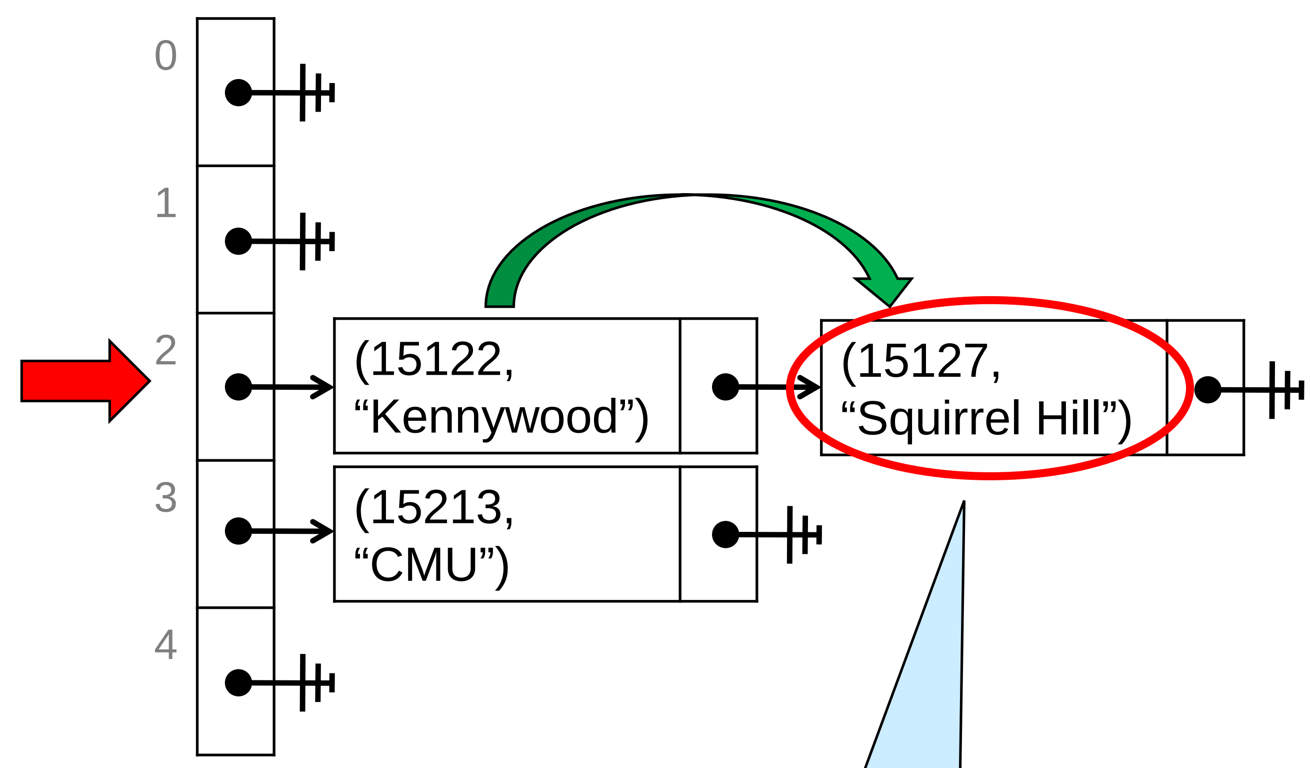
- Each table position contains a chain
  - a NULL-terminated linked list of entries
  - chain at index *i* contains all entries that map to *i*



Example, continued with separate chaining

insert (15213, “CMU”)
insert (15122, “Kennywood”)
lookup 15213
lookup 15219
lookup 15217
insert (15217, “Squirrel Hill”)
lookup 15217
lookup 15219

- insert (15217, “Squirrel Hill”)
  - compute table index as 15217 % 5 = 2
    - points to a chain node
    - check its key: 15122 ≠ 15217
  - try next node
    - there is no next node
    - create new node and insert (15217, “Squirrel Hill”) in it

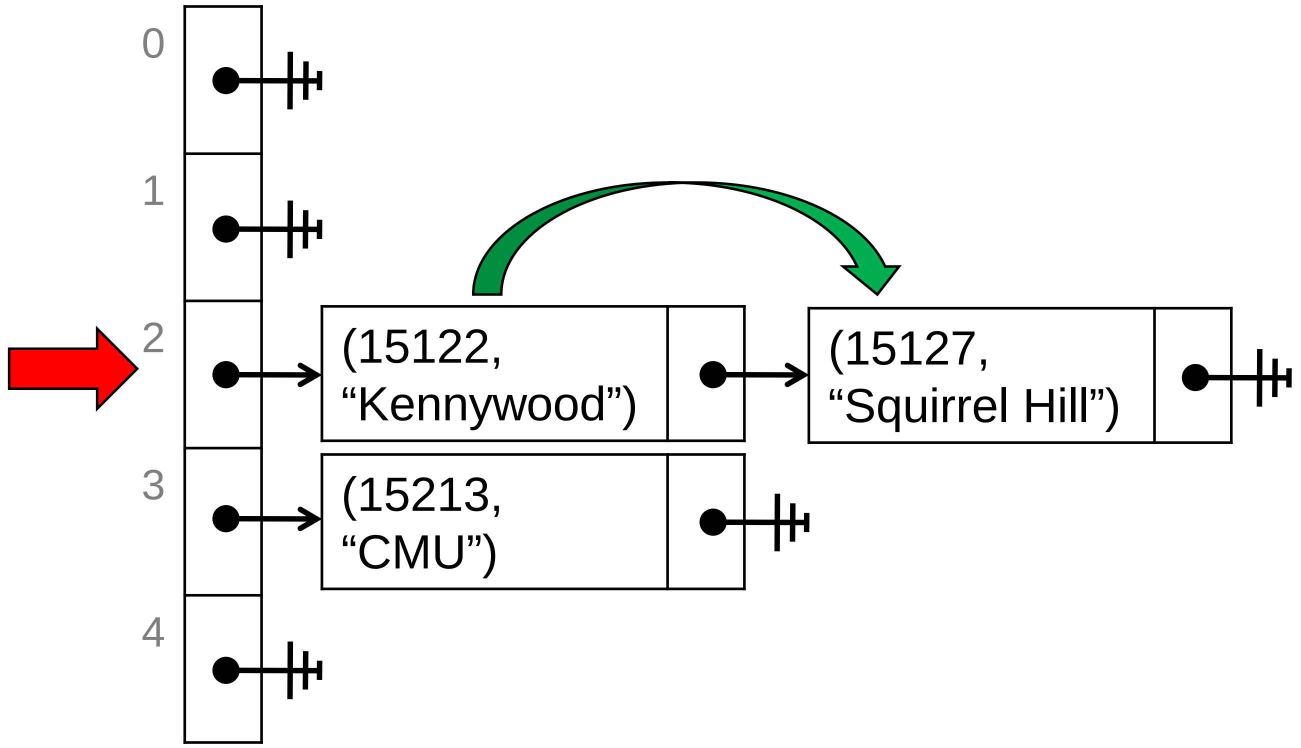


In practice, it is easier to insert new nodes at the beginning of a chain

Example, continued with separate chaining

insert (15213, “CMU”)
insert (15122, “Kennywood”)
lookup 15213
lookup 15219
lookup 15217
insert (15217, “Squirrel Hill”)
lookup 15217
lookup 15219

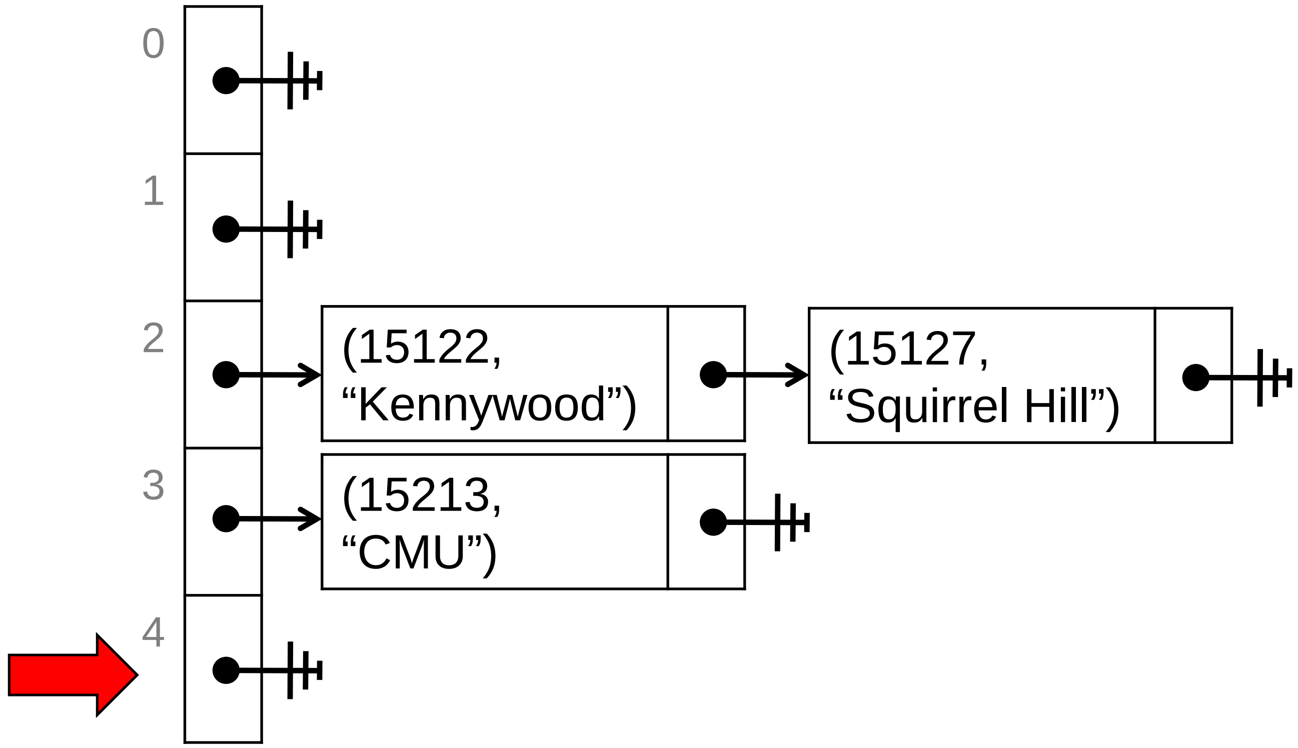
- lookup 15217
  - compute table index as 15217 % 5 = 2
    - points to a chain node
    - check its key: 15122 ≠ 15217
  - try next node
    - check its key: 15217 = 15217
    - return (15217, “Squirrel Hill”)



Example, continued with separate chaining

insert (15213, “CMU”)
insert (15122, “Kennywood”)
lookup 15213
lookup 15219
lookup 15217
insert (15217, “Squirrel Hill”)
lookup 15217
lookup 15219

- lookup 15219
  - compute table index as 15219 % 5 = 4
    - there is no chain node
    - report there is no entry for 15219



Cost Analysis

Setup

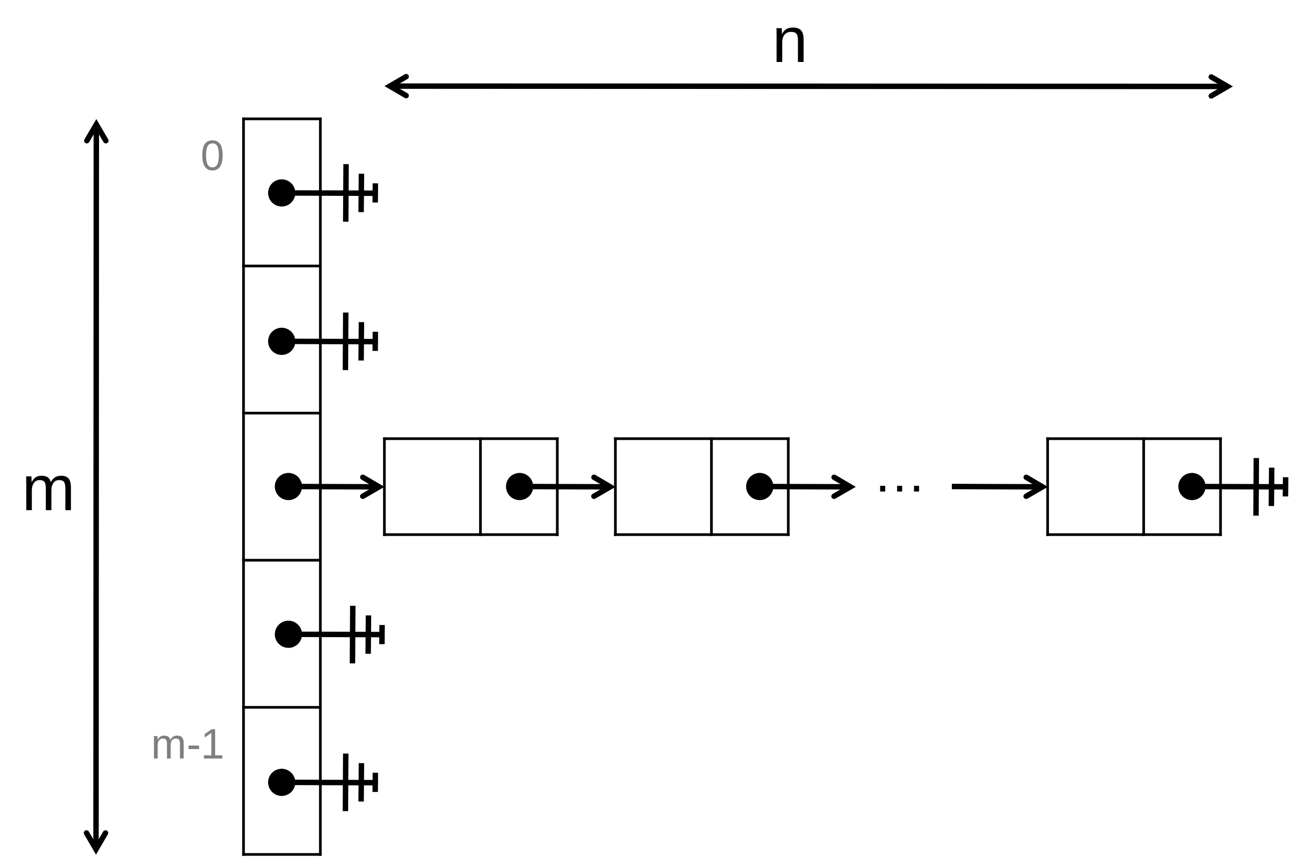
- Assume
  - the dictionary contains *n* entries
  - the table has capacity *m*
  - collisions are resolved using separate chaining
    - the analysis is similar for open addressing with linear probing
      - but not as visually intuitive

- What is the cost of **lookup**?

  - Observe that **insert** has the same cost
    - we need to check if an entry with that key is already in the dictionary
      - if so, replace that entry (update)
      - if not, add a new node to the chain (proper insert)

Worst Possible Layout

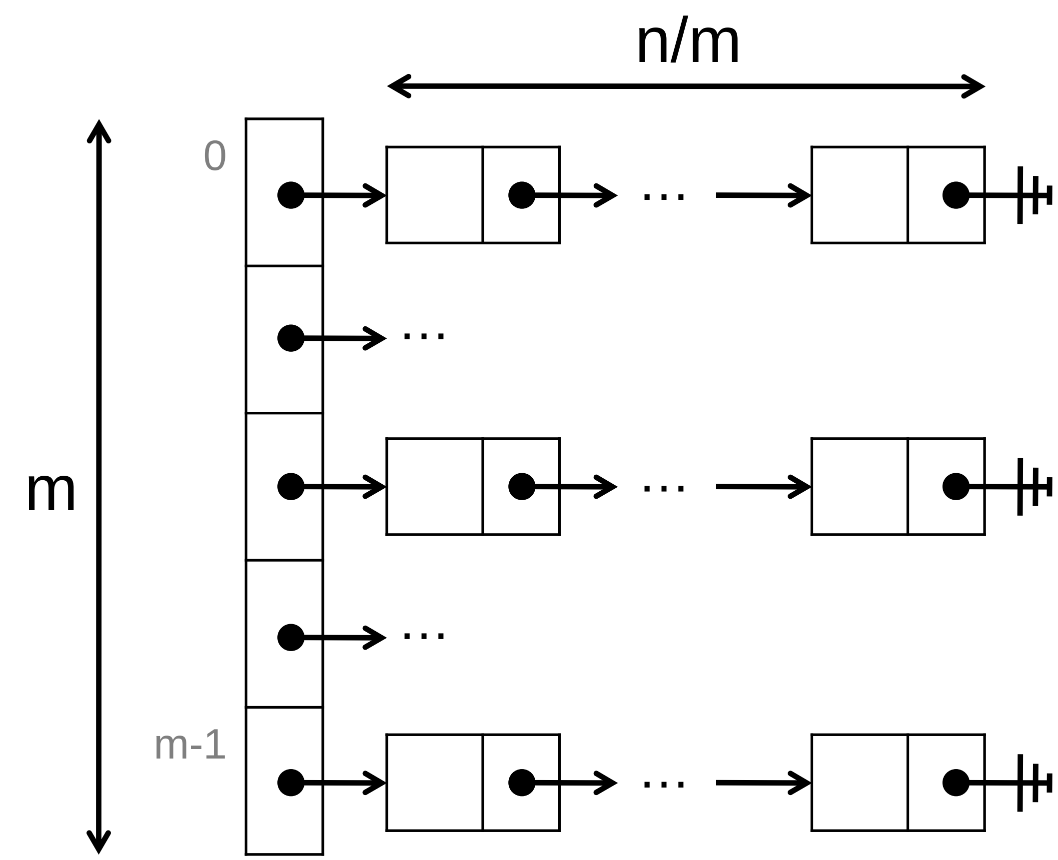
- All entries are in the same bucket
  - look for a key that belongs to this bucket but that is not in the dictionary



- Looking up a key has cost O(n)
  - find the bucket -- O(1)
  - going through all n nodes in the chain

Best Possible Layout

- All buckets have the same number of entries
  - o all chains have the same length
    - ➢ n/m
  - o n/m is called the **load factor** of the table
    - ➢ in general, the load factor is a fractional number, e.g., 1.2347



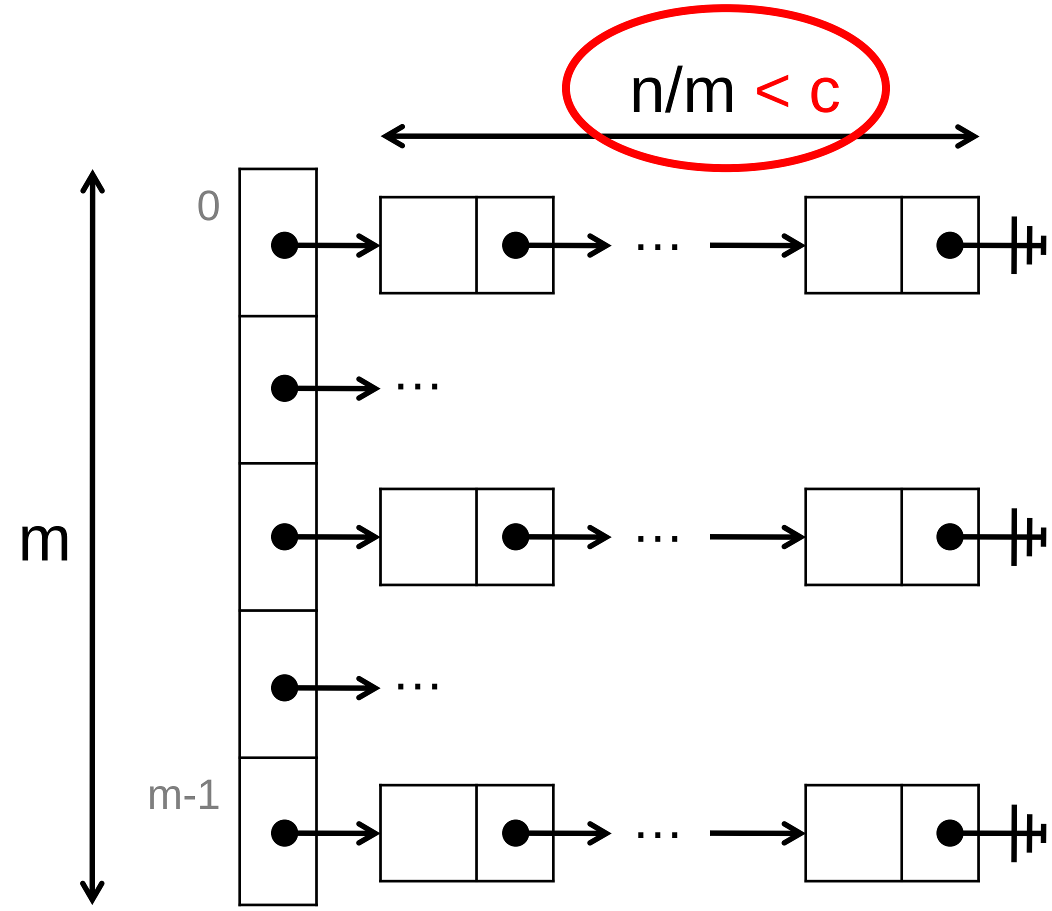
- Looking up a key has **worst-case** cost O(n/m)
  - o find the bucket -- O(1)
  - o go through all n/m nodes in the chain
- O(n/m) is also the **average-case** complexity of lookup
  - o the sum of the cost of all layouts divided the number of layouts

Best Possible Layout

Worst-case cost is O(n/m)

- Can we arrange so that n/m is **about constant**?
  - Yes! Resize the table when n/m reaches a fixed threshold c
    - often, we choose c = 1.0

c is a **constant**



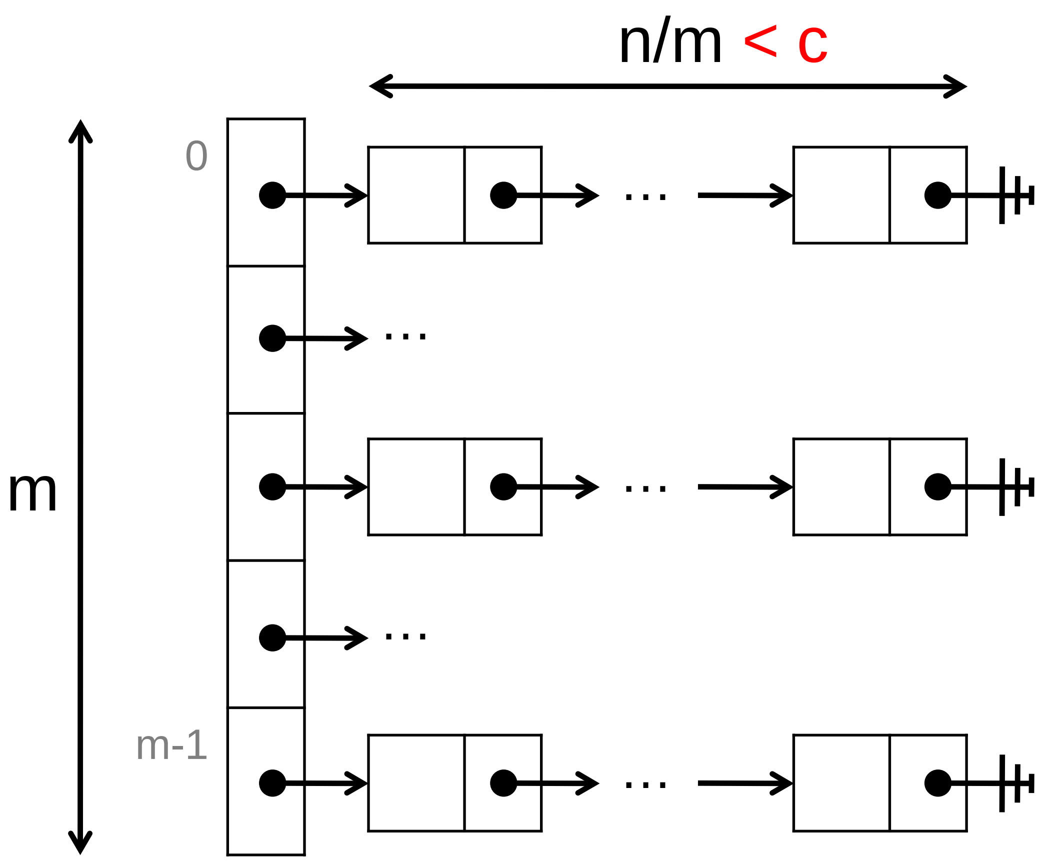
- When inserting, **double** the size of the table when n/m reaches c
- The worst-case cost becomes **O(1) amortized**
  - like with unbounded arrays

Best Possible Layout

Why O(1) amortized?

- Setup
  - dictionary contains n entries
  - table has capacity m
  - n/m < c

c is a constant



- After inserting a new entry,
  - either (n+1)/m < c
  - or (n+1)/m ≥ c

Resize the table

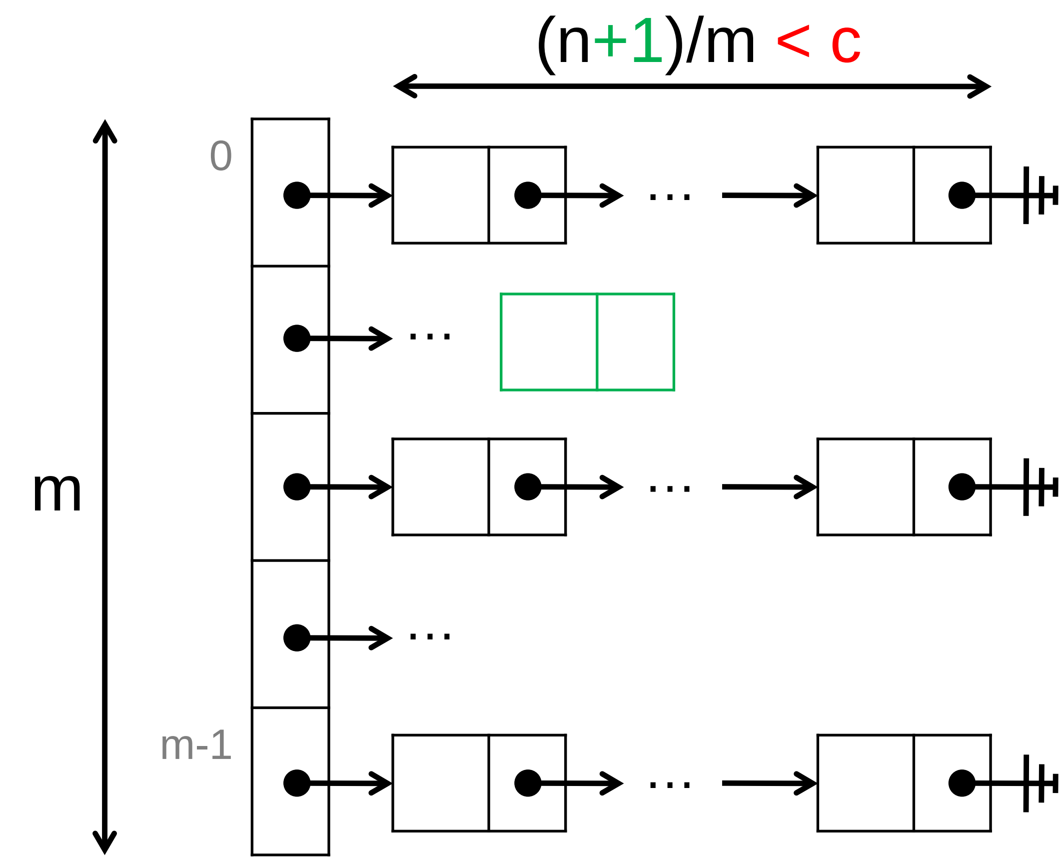
Best Possible Layout

Why O(1) amortized?

- Case (n+1)/m < c
  - o go to the right bucket
  - o check if it contains an entry with this key
    - ➢ examine about n/m nodes
    - ➢ that's at most c nodes

c is a constant

  - o insert or update the entry



This insert costs O(1)

Since (n+1)/m < c, the next lookup also costs O(1)

Best Possible Layout

Why O(1) amortized?

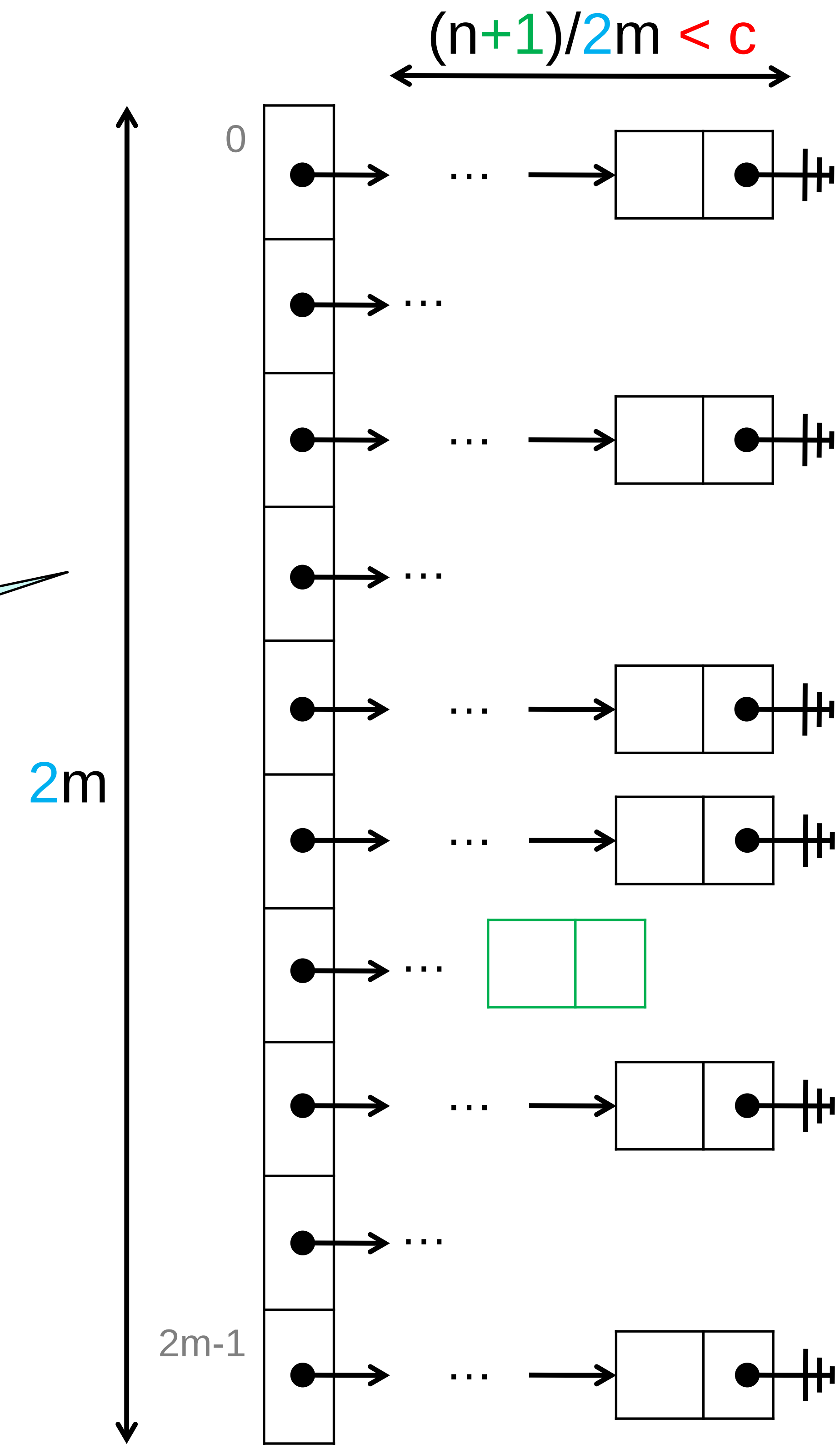
- Case (n+1)/m ≥ c
  - o double the table capacity to 2m
  - o insert **all entries** into the new table
    - ➢ n times O(1)
    - ➢ that's O(n)

__If__ we keep on being in the best possible layout

This insert costs O(n)

  - o The new load factor is (n+1)/2m < c
    - ➢ because (n+1)/2m < 2n/2m = n/m < c

Thus, the next lookup costs O(1)



Best Possible Layout

Why O(1) amortized?

- After inserting a new entry,
  - o either $(n+1)/m < c$
    - ➢ costs O(1)
  - o or $(n+1)/m \geq c$
    - ➢ costs O(n)
    - ➢ but the next n inserts will cost O(1)

This is cheap!

This is expensive!

Assuming we still have the best possible layout …

- Just like with unbounded array
  - o many cheap operations can pay for the rare expensive ones
- Thus, insert has **O(1) amortized cost**
  - o because lookup depends on what was inserted in the table, we view it too as having O(1) amortized cost

Best Possible Layout

- When will we be in this ideal case?
  - when the indices associated with the keys in the table are **uniformly distributed** over [0,m)
  - this happens when the keys are chosen at **random** over the integers

- Is this typical?
  - Keys are rarely random
    - e.g., if we take first digit of zip code (instead of last)
      - many students from Pennsylvania: lots of 1
      - many students from the West Coast: lots of 9 (mapped to 4, modulo 5)
  - We shouldn’t count on it

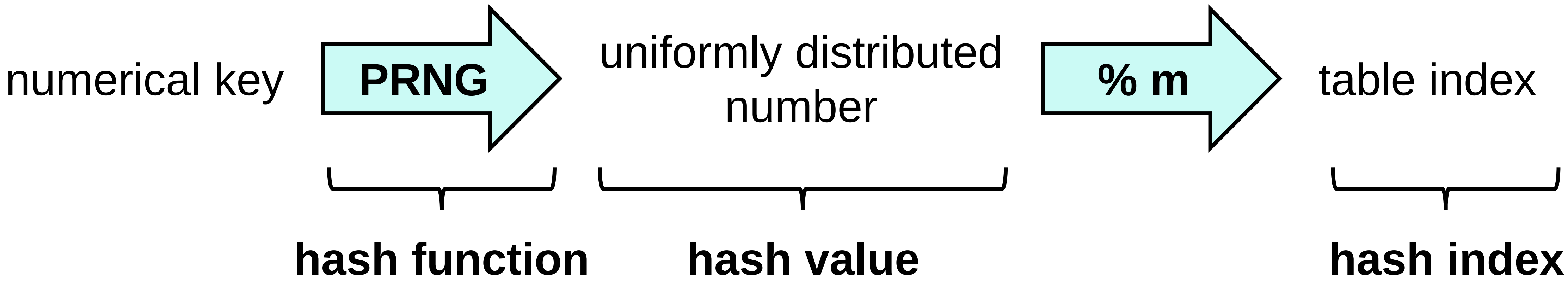
Best Possible Layout

- Can we *arrange* so that we **always** end up in this ideal case?
  - ➢ unless we are really, really unlucky
  - We want the indices associated to keys to be scattered
    - ➢ be **uniformly distributed** over the table indices
    - ➢ bear little relation to the key itself

- Run the key through a **pseudo-random number generator**
  - “*random number generator*”: result *appears* random
    - ❑ uniformly distributed
    - ❑ (apparently) unrelated to input
  - “*pseudo*”: always returns the same result for a given key
    - ❑ deterministic

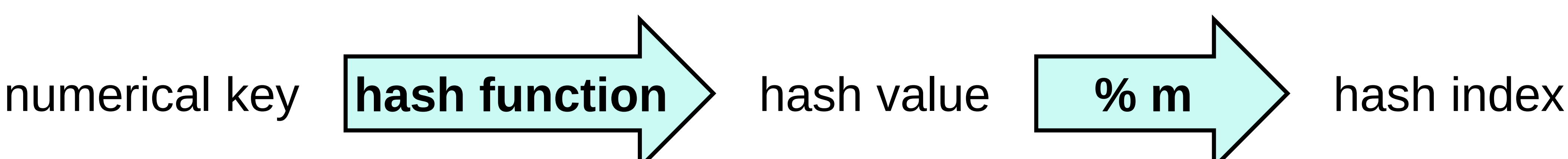
numerical key $\xrightarrow{\textbf{PRNG}}$ uniformly distributed number $\xrightarrow{\textbf{\% m}}$ table index

Hash Tables



This is a **hash table**

- a PRNG is an example of a **hash function**
  - a function that turns a key into a number on which to base the table index
- its result is a **hash value**
- it is then turned into a **hash index** in the range [0, m)



Hash Table Complexity

- Complexity of **lookup**, assuming
  - the dictionary contains *n* entries
  - the table has capacity *m*
  - and …

Output is **uniformly distributed** and **unrelated to input**

	Bad hash function	***Good hash function***
No resizing	*O(n)*	*(Left as exercise)*
UBA-style resizing	*(Left as exercise)*	*O(1)* *<u>average</u>* *and* *<u>amortized</u>*

Double the size of the table when load factor exceeds target

From good hash function

From UBA-style resizing

Pseudo-Random Number Generators

Linear Congruential Generators

- A common form of PRNG is

$$f(x) = a * x + c \bmod d$$

  - for appropriate constants *a*, *c* an *d*

- With 32-bit ints and handling overflow via modular arithmetic, we choose $d = 2^{32}$
  - *mod d* is automatic
- To get uniform distribution, we pick
  - a ≠ 0
  - c and d to be relative primes

- This is called a **linear congruential generator** (LCG)
  - Cost is O(1)

Linear Congruential Generators

$$f(x) = a * x + c \bmod d$$

- a ≠ 0, and *c* and *d* relatively prime
- $d = 2^{32}$

- Implemented in the C0 **rand library**

  #use <rand>

  - a = 1664525
  - c = 1013904223

- Do it yourself?

```
int lgc(int x) {
  return 1664525 * x + 1013904223 ;
}
```

The rand library is a bit more general. It’s interface is:

```
// typedef ___ rand_t;
rand_t init_rand (int seed);
int rand(rand_t gen):
```

Look it up!

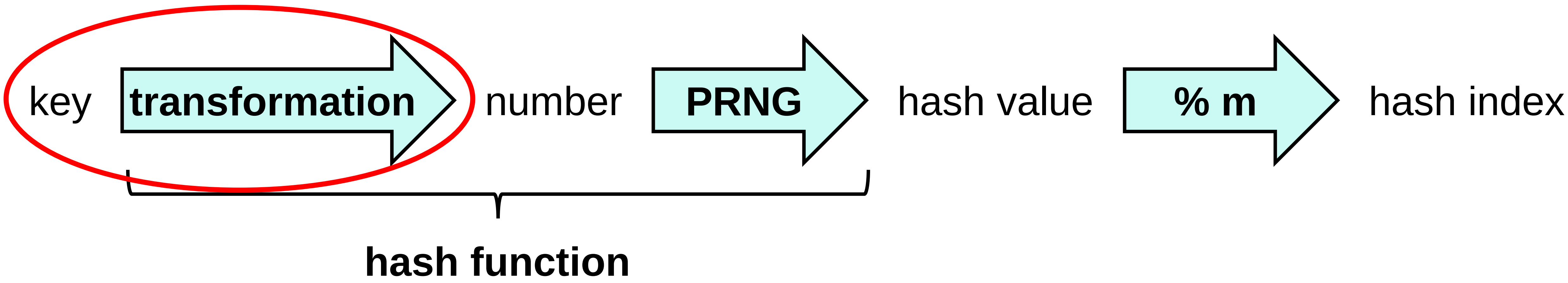
Cryptographic Hash Functions

- Hash functions are used pervasively in cryptography
- Cryptographic hash functions have additional requirements
  - practically impossible to find x given h(x)
  - practically impossible to find x and y such that h(x) = h(y)
- Cryptographic hash functions are overkill for use in hash tables

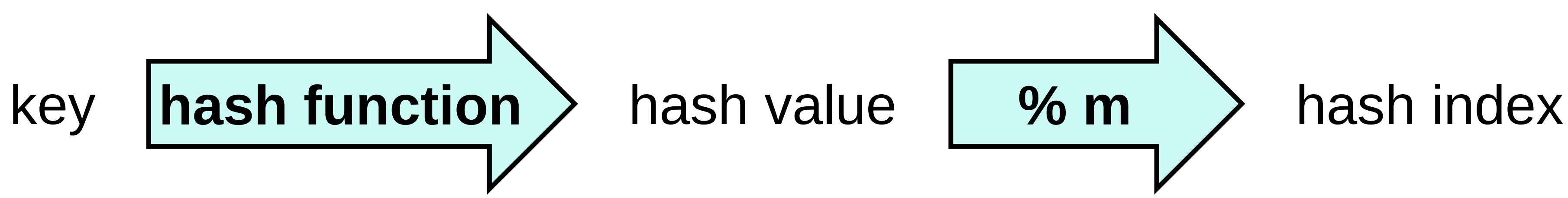
Non-numerical Keys

Hashing Non-numerical Keys

- Simply transform the key into a number first (*cheaply*)



- The whole transformation from key to hash value is called the hash function
  - often implemented as a single function



Dictionaries Summary

- We can use hash tables to implement efficient dictionaries
  - type of keys can be anything we want
  - O(1) average and amortized cost for **lookup** and **insert**

key $\xrightarrow{\textbf{hash function}}$ hash value $\xrightarrow{\textbf{\% m}}$ hash index

  - Collision resolved via separate chaining or open addressing
    - Open addressing is more common in practice
      - uses less space

- They are called **hash dictionaries**

Dictionaries Summary

- Complexity assuming
  - o the dictionary contains *n* entries
  - o the table has capacity *m*

	unsorted array with (key, value) data	***(key, value) array sorted by key***	***linked list with (key, value) data***	***Hash Tables***
lookup	*O(n)*	*O(log n)*	*O(n)*	*O(n)* *O(n/m) average* *O(1) average and amortized*
insert	*O(1) amortized*	*O(n)*	*O(1)*	*O(n)* *O(n/m) average* *O(1) average and amortized*

** The same analysis applies for open addressing hash tables*

What about Sets?

- A **set** can be understood as a special case of a dictionary
  - keys = entries
    - These are the elements of the set
  - **lookup** can simply return true or false
    - this now checks set membership

- A set implemented as a hash dictionary is called a **hash set**