Hashes

lists

operation	array/vector	linked list
find (by value)	$\Theta(n)$	$\Theta(n)$
insert (end)	amortized $O(1)$	$\Theta(1)$
insert (beginning/middle)	$\Theta(n)$	$\Theta(1)$
remove (by value)	$\Theta(n)$	$\Theta(n)$
find (by index)	$\Theta(1)$	$\Theta(1)$

stacks

operation	array/vector	linked list
push	amortized $O(1)$	$\Theta(1)$
pop	$\Theta(1)$	$\Theta(1)$
top	$\Theta(1)$	$\Theta(1)$
isEmpty	$\Theta(1)$	$\Theta(1)$

queues

operation	array/vector	linked list
enqueue	amortized $O(1)$	$\Theta(1)$
dequeue	$\Theta(1)$	$\Theta(1)$

sets

abstract data type with subset of list operations:

- find (by value)
- insert (unspecified location)
- remove (by value)

omits:

- find (by index)
- insert at particular location

sets

operation	BST	AVL or red-black	vector	hash table
find (by value)	$\Theta(\text{height})$*	$\Theta(\log n)$	$\Theta(n)$	$O(1)$†
insert	$\Theta(\text{height})$*	$\Theta(\log n)$	amortized $O(1)$	$O(1)$†
remove	$\Theta(\text{height})$*	$\Theta(\log n)$	$\Theta(1)$	$O(1)$†
find max/min	$\Theta(\text{height})$*	$\Theta(\log n)$	$\Theta(n)$	$\Theta(n)$

sets

operation	BST	AVL or red-black	vector	hash table
find (by value)	$\Theta(\text{height})$*	$\Theta(\log n)$	$\Theta(n)$	$O(1)$†
insert	$\Theta(\text{height})$*	$\Theta(\log n)$	amortized $O(1)$	$O(1)$†
remove	$\Theta(\text{height})$*	$\Theta(\log n)$	$\Theta(1)$	$O(1)$†
find max/min	$\Theta(\text{height})$*	$\Theta(\log n)$	$\Theta(n)$	$\Theta(n)$

*BST: height is "often" $\Theta(\log n)$, but can be $\Theta(n)$

†hash table — $O(1)$ "usually", but $\Theta(n)$ worst case

sets

operation	BST	AVL or red-black	vector	hash table
find (by value)	$\Theta(\text{height})$*	$\Theta(\log n)$	$\Theta(n)$	$O(1)$†
insert	$\Theta(\text{height})$*	$\Theta(\log n)$	amortized $O(1)$	$O(1)$†
remove	$\Theta(\text{height})$*	$\Theta(\log n)$	$\Theta(1)$	$O(1)$†
find max/min	$\Theta(\text{height})$*	$\Theta(\log n)$	$\Theta(n)$	$\Theta(n)$

*BST: height is "often" $\Theta(\log n)$, but can be $\Theta(n)$

how to get worst case: insert in sorted order

†hash table — $O(1)$ "usually", but $\Theta(n)$ worst case

sets

operation	BST	AVL or red-black	vector	hash table
find (by value)	$\Theta(\text{height})$*	$\Theta(\log n)$	$\Theta(n)$	$O(1)$†
insert	$\Theta(\text{height})$*	$\Theta(\log n)$	amortized $O(1)$	$O(1)$†
remove	$\Theta(\text{height})$*	$\Theta(\log n)$	$\Theta(1)$	$O(1)$†
find max/min	$\Theta(\text{height})$*	$\Theta(\log n)$	$\Theta(n)$	$\Theta(n)$

*BST: height is “often” $\Theta(\log n)$, but can be $\Theta(n)$

how to get worst case: insert in sorted order

†hash table — $O(1)$ “usually”, but $\Theta(n)$ worst case

how to get worst case: insert specially chosen set of items

can design hash table to make this really rare

maps

abstract data type with key-value pairs

examples:

- key=computing ID, value=grade
- key=word, value=definition
- key=user ID, value=object with many fields

operations:

- find value by key
- insert(key, value)
- remove by key

map with vector

```
class KeyValuePair {
public:
    string key;
    int value;
};

class VectorMap {
public:
    void insert(const string& key, int value);
    int find(const string& key);  // XXX value if not found?
    void remove(const string& key);
private:
    vector<KeyValuePair> data;
};
```

maps

operation	BST	AVL or red-black	vector	hash table
find (by key)	$\Theta(\text{height})$*	$\Theta(\log n)$	$\Theta(n)$	$O(1)$†
insert	$\Theta(\text{height})$*	$\Theta(\log n)$	amortized $O(1)$	$O(1)$†
remove (by key)	$\Theta(\text{height})$*	$\Theta(\log n)$	$\Theta(1)$	$O(1)$†

*BST: height is "often" $\Theta(\log n)$, but can be $\Theta(n)$

†hash table — $O(1)$ "usually", but $\Theta(n)$ worst case

aside: standard library

std::map — balanced tree-based map

std::unordered_map — hashtable-based map

```
unordered_map<string, double> grades;
grades["cr4bd"] = 85.0;
...
if (grades.count("mst3k") > 0) {
    cout << "mst3k␣has␣a␣grade␣assigned\n";
}
for (unordered_map<string, double>::iterator it = grades.begin();
     it != grades.end(); ++it) {
    cout << it->first << "␣" << it->second << "\n";
 }
```

std::set — balanced tree-based set

std::unordered_set — hashtable-based set

key-value pairs

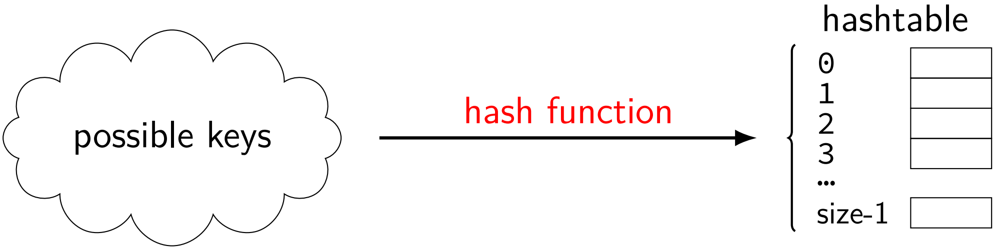
sets are special maps — map where values are ignored

hashtable

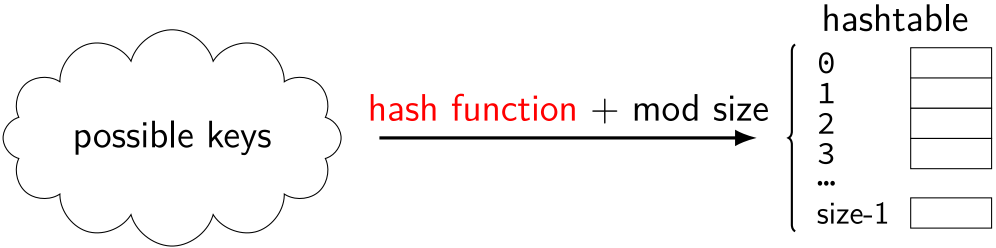
array of some size

- larger than # of total elements
- usually prime size

hash function: map keys to array indices



hash function properties (1)

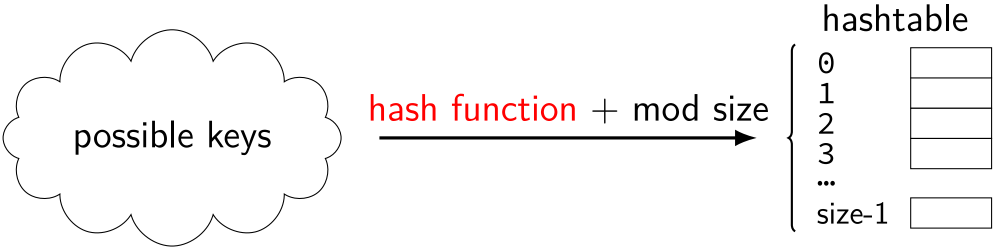


input: key type (e.g. string) → output: unsigned integer

then take typically — then take mod of the table size

result is the "bucket" used to store info for that key

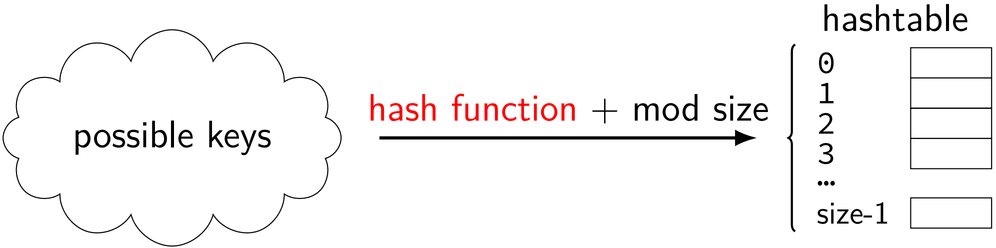
hash function properties (2)



must be deterministic

each key assigned to exactly one “bucket”

hash function properties (2)



must be deterministic

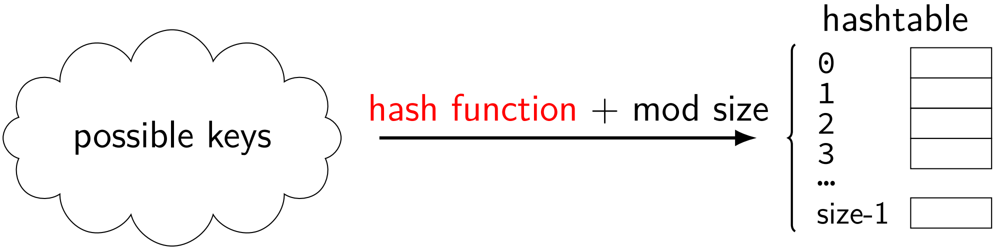
each key assigned to exactly one "bucket"

should be evenly distributed

two keys *unlikely* to share bucket

each bucket about as used as each other bucket

hash function properties (2)



must be deterministic

each key assigned to exactly one “bucket”

should be evenly distributed

two keys *unlikely* to share bucket

each bucket about as used as each other bucket

should be fast

activity

hash students here by birthday

- or choose arbitrary date — just be consistent

four options:

- decade of birth year (`(year/10)%10`)
- last digit of birth year (`year%10`)
- last digit of birth month (`month%10`)
- last digit of birth day (`day%10`)

exercise

hashtable: birthdate $\rightarrow$ info about person w/birthdate

which option is best?

A. birth year (`year`)
B. birth day (`day`)
C. days between now and birthdate (`(date - today()).days()`)
D. `year*128 + month*32 + day`
E. `year+month+day`
F. `year*(month-1)*(day-1)`

recall: deterministic, evenly distributed, fast

example (1)

key: integers

table size: 10

hash function: $h(k) = k$; hash+mod: $k \bmod 10$

insert 7, 18, 41, 34

index	keys
0	
1	
2	
3	
4	
5	
6	
7	
8	
9	

example (1)

key: integers

table size: 10

hash function: $h(k) = k$; hash+mod: k mod 10

insert 7, 18, 41, 34

7, h(7) mod 10 = 7 — use bucket 7

index	keys
0	
1	
2	
3	
4	
5	
6	
7	7
8	
9	

example (1)

key: integers

table size: 10

hash function: $h(k) = k$; hash+mod: $k \bmod 10$

insert 7, 18, 41, 34

  7, h(7) mod 10 = 7 — use bucket 7

  18, h(18) mod 10 = 8 — use bucket 8

  ...

index	keys
0	
1	
2	
3	
4	
5	
6	
7	7
8	18
9	

example (1)

key: integers

table size: 10

hash function: $h(k) = k$; hash+mod: $k \bmod 10$

insert 7, 18, 41, 34

7, h(7) mod 10 = 7 — use bucket 7

18, h(18) mod 10 = 8 — use bucket 8

...

index	keys
0	
1	41
2	
3	
4	
5	
6	
7	7
8	18
9	

example (1)

key: integers

table size: 10

hash function: $h(k) = k$; hash+mod: $k \bmod 10$

insert 7, 18, 41, 34

7, h(7) mod 10 = 7 — use bucket 7

18, h(18) mod 10 = 8 — use bucket 8

...

index	keys
0	
1	41
2	
3	
4	34
5	
6	
7	7
8	18
9	

example (1)

key: integers

table size: 10

hash function: $h(k) = k$; hash+mod: $k \bmod 10$

insert 7, 18, 41, 34

7, h(7) mod 10 = 7 — use bucket 7

18, h(18) mod 10 = 8 — use bucket 8

...

find 34, 28, 90

34, h(34) mod 10 = 4 — use bucket 4 — found

index	keys
0	
1	41
2	
3	
4	34
5	
6	
7	7
8	18
9	

example (1)

key: integers

table size: 10

hash function: $h(k) = k$; hash+mod: $k \bmod 10$

insert 7, 18, 41, 34

7, h(7) mod 10 $=$ 7 — use bucket 7

18, h(18) mod 10 $=$ 8 — use bucket 8

...

find 34, 28, 90

34, h(34) mod 10 $=$ 4 — use bucket 4 — found

28, h(28) mod 10 $=$ 8 — use bucket 8 — not a match

index	keys
0	
1	41
2	
3	
4	34
5	
6	
7	7
8	**18**
9	

example (1)

key: integers

table size: 10

hash function: $h(k) = k$; hash+mod: $k \bmod 10$

insert 7, 18, 41, 34

 7, h(7) mod 10 = 7 — use bucket 7

 18, h(18) mod 10 = 8 — use bucket 8

 ...

find 34, 28, 90

 34, h(34) mod 10 = 4 — use bucket 4 — found

 28, h(28) mod 10 = 8 — use bucket 8 — not a match

 90, h(90) mod 10 = 0 — use bucket 0 — nothing there

index	keys
0	
1	41
2	
3	
4	34
5	
6	
7	7
8	18
9	

hashtable algorithms

find (by key k): compute $i = h(k) \mod$ table size, check bucket at index i

need to check key — other keys may use same bucket

insert/remove (by key k): compute $i = h(k) \mod$ table size, use bucket at index i

but what if bucket is used by another key?

find max/min: check all buckets (linear time)

hashing strings

```
unsigned long hashTableIndex(const string &s, unsigned long ta
    return hash(s) % tableSize;
}

unsigned long hash(const string &s) {
    ???
}
```

some proposals (1)

```
unsigned long hash(const string &s) {
    return s[0];
}

unsigned long hash(const string &s) {
    unsigned long sum = 0;
    for (int i = 0; i < s.size(); ++i) {
        sum += s[i];
    }
    return sum;
}
```

some proposals (2)

```
unsigned long hash(const string &s) {
    unsigned long sum = 0;
    for (int i = 0; i < s.size(); ++i) {
        // deliberate use of wraparound on overflow
        sum *= 37;
        sum += s[i];
    }
    return sum;
}
```

example (2)

index	keys
0	
1	
2	
3	
4	
5	
6	
7	
8	
9	
10	

key: strings

table size: 11

hash function: $h(k) = \sum_i k_i$ (ASCII codes)

hash+mod: $h(k) \mod 11 = \sum_i k_i \mod 11$

insert "foo", "bar","baz"

find "baz", "quux"

example (2)

key: strings

table size: 11

hash function: $h(k) = \sum_i k_i$ (ASCII codes)

hash+mod: $h(k) \mod 11 = \sum_i k_i \mod 11$

insert "foo", "bar","baz"

h("foo") $=$ 324 — bucket $324 \mod 11 = 5$

find "baz", "quux"

index	keys
0	
1	
2	
3	
4	
5	"foo"
6	
7	
8	
9	
10	

example (2)

key: strings

table size: 11

hash function: $h(k) = \sum_i k_i$ (ASCII codes)

hash+mod: $h(k) \mod 11 = \sum_i k_i \mod 11$

insert “foo”, “bar”,“baz”

h(“foo”) $=$ 324 — bucket $324 \mod 11 = 5$

h(“bar”) $=$ 309 — bucket $309 \mod 11 = 1$

find “baz”, “quux”

index	keys
0	
1	“bar”
2	
3	
4	
5	“foo”
6	
7	
8	
9	
10	

example (2)

key: strings

table size: 11

hash function: $h(k) = \sum_i k_i$ (ASCII codes)

hash+mod: $h(k) \mod 11 = \sum_i k_i \mod 11$

insert "foo", "bar", "baz"

h("foo") = 324 — bucket $324 \mod 11 = 5$

h("bar") = 309 — bucket $309 \mod 11 = 1$

h("baz") = 317 — bucket $317 \mod 11 = 9$

find "baz", "quux"

index	keys
0	
1	"bar"
2	
3	
4	
5	"foo"
6	
7	
8	
9	"baz"
10	

example (2)

key: strings

table size: 11

hash function: $h(k) = \sum_i k_i$ (ASCII codes)

hash+mod: $h(k) \mod 11 = \sum_i k_i \mod 11$

insert “foo”, “bar”,“baz”

h(“foo”) $=$ 324 — bucket $324 \mod 11 = 5$
h(“bar”) $=$ 309 — bucket $309 \mod 11 = 1$
h(“baz”) $=$ 317 — bucket $317 \mod 11 = 9$

find “baz”, “quux”

index	keys
0	
1	“bar”
2	
3	
4	
5	“foo”
6	
7	
8	
9	“baz”
10	

example (2)

index	keys
0	
1	"bar"
2	
3	
4	
5	"foo"
6	
7	
8	
9	"baz"
10	

key: strings

table size: 11

hash function: $h(k) = \sum_i k_i$ (ASCII codes)

hash+mod: $h(k) \mod 11 = \sum_i k_i \mod 11$

insert "foo", "bar","baz"

h("foo") $=$ 324 — bucket $324 \mod 11 = 5$
h("bar") $=$ 309 — bucket $309 \mod 11 = 1$
h("baz") $=$ 317 — bucket $317 \mod 11 = 9$

find "baz", "quux"

h("quux") $=$ 317 — bucket $467 \mod 11 = 5$

example (1b)

key: integers

table size: 10

hash function: $h(k) = k$; hash+mod: $k \bmod 10$

insert 7, 18, 41, 34, 11

index	keys
0	
1	41
2	
3	
4	34
5	
6	
7	7
8	18
9	

example (1b)

key: integers

table size: 10

hash function: $h(k) = k$; hash+mod: $k \bmod 10$

insert 7, 18, 41, 34, 11

11, h(11) mod 10 = 1

index	keys
0	
1	41, 11
2	
3	
4	34
5	
6	
7	7
8	18
9	

hashtable algorithms

find (by key k): compute $i = h(k) \mod$ table size, check bucket at index i

need to check key — other keys may use same bucket

insert/remove (by key k): compute $i = h(k) \mod$ table size, use bucket at index i

but what if bucket is used by another key?

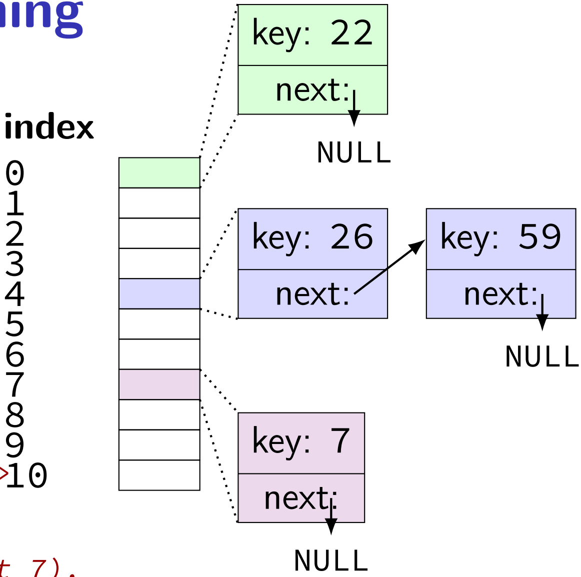
find max/min: check all buckets (linear time)

option 1: separate chaining

```
class HashTableBucket {
    int key;
    HashTableBucket *next;
    // ... + value?
};

class HashTable {
    ...;
private:
    vector<HashTableBucket> data;
        // could also use
        // vector<HashTableBucket*>
};

// insert {26 (bucket 4), 7 (bucket 7),
//         22 (bucket 0), 59 (bucket 4)}
```

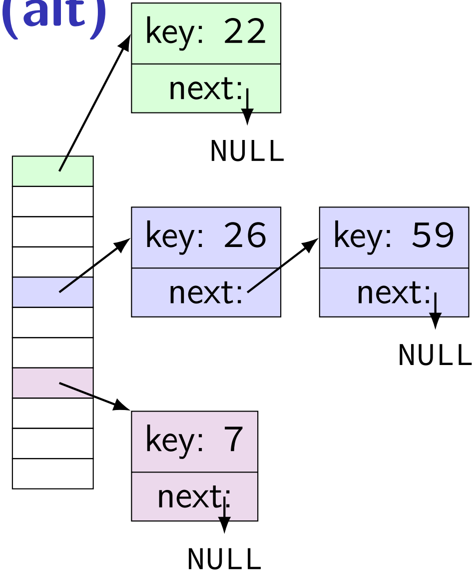


option 1: separate chaining (alt)

```
class HashTableBucket {
    int key;
    HashTableBucket *next;
    // ... + value?
};

class HashTable {
    ...;
private:
    vector<HashTableBucket*> data;
        // could also use
        // vector<HashTableBucket>
};

// insert {26 (bucket 4), 7 (bucket 7),
//         22 (bucket 0), 59 (bucket 4)}
```



load factors and chaining

load factor: $\lambda = \dfrac{\#\ \text{elements}}{\text{table size}}$

average number of elements per bucket: λ

find performance

average* time for find:

- unsuccessful: check λ items
- successful: check $1 + \lambda/2$ items (half of list)

*assuming we choose random keys?

find performance

average* time for find:

- unsuccessful: check λ items
- successful: check $1 + \lambda/2$ items (half of list)

*assuming we choose random keys?

maybe our keys aren't average

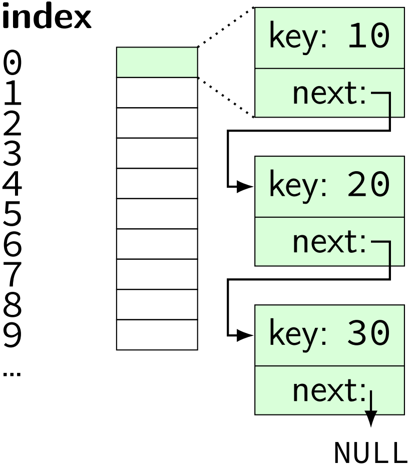
$h(k) = k$

size $= 10$

index $= h(k) \mod 10$

$\lambda = 0.3$

but if we usually lookup existing keys...



why use a linked list?

one item/bucket usually

- if not — we should use a balanced tree
- (or change hash functions?)

when not one, probably two or three

linked list — probably most efficient

- typical space overhead: one NULL pointer
- typical time overhead: check the one pointer

linked list alternatives

vector — way too much extra space

- size, capacity
- extra space reserved in array
- remember: typically just one element

balanced trees

- two pointers
- about same comparisons as linked list for size 2, 3

find performance revisited

with ideal hash function: $\Theta(\lambda)$ (load factor)

typically: adjust hashtable size so λ remains approximately constant

actual worst case: $\Theta(n)$ (I choose all the wrong keys)

insert performance

$\Theta(1)$

assuming we don't care about checking for a duplicate

don't care about sorting the linked list

insert at head

delete performance

need to do a find to get the bucket

then linked list removal $\Theta(1)$

(if singly linked list — track previous while finding)

rehashing

how big should the table be?

- $C \times$ number of items
- typical C — $\frac{1}{2}$ to 1

rehashing

how big should the table be?

$C \times$ number of items

typical C — $\frac{1}{2}$ to 1

as number of change: want to resize it!

called rehashing

...because we recompute every key's hash

when to rehash?

load factor $\lambda =$ elements/table size

typical policy: resize table when $\lambda >$ threshold

java.util policy: when $\lambda > 0.75$

alternatives:

- only when insert fails?

rehashing big-oh

worst case:

- everything hashes to same bucket
- $\Theta(n)$ time per insert
- $\Theta(n)$ inserts
- $\Theta(n^2)$ total time

if keys are well spread out between buckets

- "about" linear time

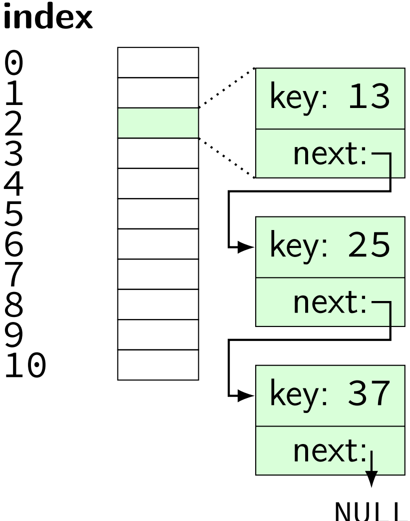
avoiding linked lists

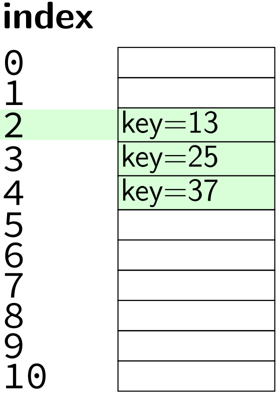
```
class HashTableBucket {
    ...
    int key;                    // 4 bytes
    int value;                  // 4 bytes
    HashTableBucket *next; // 8 bytes
};
```

gosh, that's a lot of overhead

...even though "usually" one item/bucket

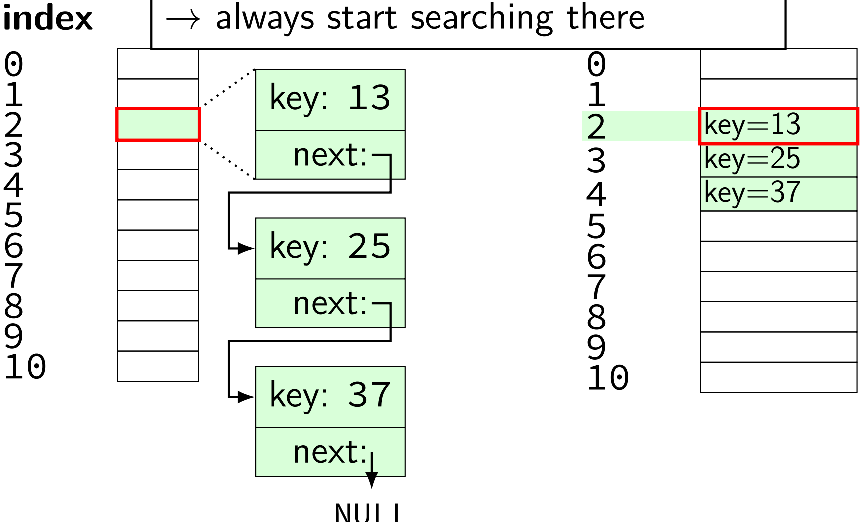
an alternative





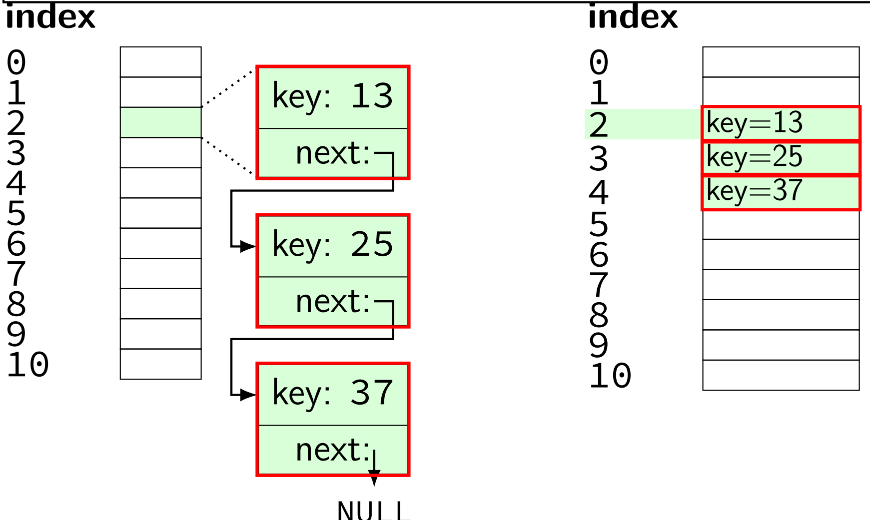
an alternative

this example: all keys hash to bucket 2
$(h(k) = k$, choose $h(k) \mod 11)$
$\rightarrow$ always start searching there



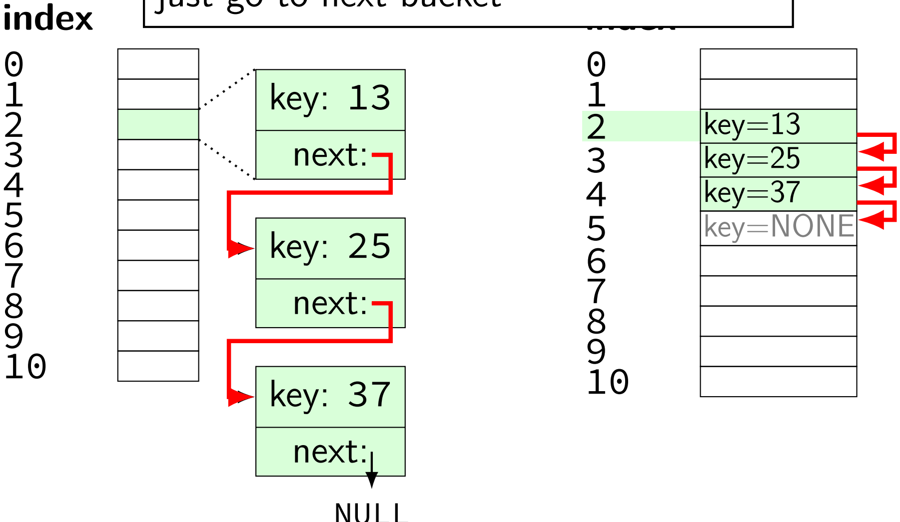
an alternative

both ways might search all keys with same hash mod size

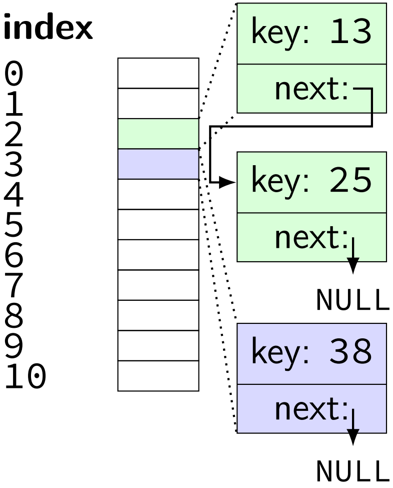


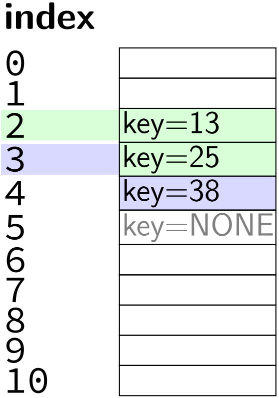
an alternative

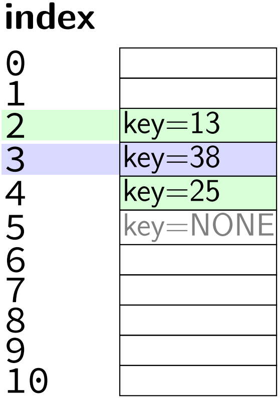
difference: new way — no next pointers
just go to next bucket



but what if...


index
0
1
2
3
4
5
6
7
8
9
10
key: 13
next:
key: 25
next:
NULL
key: 38
next:
NULL


index
0
1
2
3
4
5
6
7
8
9
10
key=13
key=25
key=38
key=NONE


index
0
1
2
3
4
5
6
7
8
9
10
key=13
key=38
key=25
key=NONE

open addressing generally

search $h(k) + f(0) \mod size$

then $h(k) + f(1) \mod size$

then $h(k) + f(2) \mod size$

…

linear probing: $f(i) = i$

probing possibilities

$h(k) + f(i) \mod size$

linear: $f(i) = i$ — previous diagram

quadratic: $f(i) = i^2$

double hashing $f(i) = i \times h_2(k)$ (second hash function)

probing possibilities

$h(k) + f(i) \mod size$

linear: $f(i) = i$ — previous diagram

quadratic: $f(i) = i^2$

double hashing $f(i) = i \times h_2(k)$ (second hash function)

linear probing example (2)

$h(k) = 3k + 7$

index $= h(k) \mod 10$

then check $h(k) + 1 \mod 10$, $h(k) + 2 \mod 10$, etc.

insert 4, 27, 37, 14, 21

$h(k) =$ 19, 88, 118, 49, 70

index	
0	
1	
2	
3	
4	
5	
6	
7	
8	
9	

linear probing example (2)

$h(k) = 3k + 7$

index $= h(k) \mod 10$

then check $h(k) + 1 \mod 10$, $h(k) + 2 \mod 10$, etc.

insert 4, 27, 37, 14, 21

$h(k) =$ 19, 88, 118, 49, 70

index	
0	
1	
2	
3	
4	
5	
6	
7	
8	
9	4

linear probing example (2)

$h(k) = 3k + 7$

index $= h(k) \mod 10$

then check $h(k)+1 \mod 10$, $h(k)+2 \mod 10$, etc.

insert 4, 27, 37, 14, 21

$h(k) =$ 19, 88, 118, 49, 70

index	
0	
1	
2	
3	
4	
5	
6	
7	
8	27
9	4

linear probing example (2)

$h(k) = 3k + 7$

index $= h(k) \mod 10$

then check $h(k) + 1 \mod 10$, $h(k) + 2 \mod 10$, etc.

insert 4, 27, 37, 14, 21

$h(k) =$ 19, 88, 118, 49, 70

index	
0	37
1	
2	
3	
4	
5	
6	
7	
8	27
9	4

linear probing example (2)

$h(k) = 3k + 7$

index $= h(k) \mod 10$

then check $h(k) + 1 \mod 10$, $h(k) + 2 \mod 10$, etc.

insert 4, 27, 37, 14, 21

$h(k) =$ 19, 88, 118, 49, 70

index	
0	37
1	14
2	
3	
4	
5	
6	
7	
8	27
9	4

linear probing example (2)

$h(k) = 3k + 7$

index $= h(k) \mod 10$

then check $h(k) + 1 \mod 10$, $h(k) + 2 \mod 10$, etc.

insert 4, 27, 37, 14, 21

$h(k) =$ 19, 88, 118, 49, 70

index	
0	37
1	14
2	21
3	
4	
5	
6	
7	
8	27
9	4

the clumping

we tend to get “clumps” of used buckets

reason why linear probing isn’t the only way

probing possibilities

$h(k) + f(i) \mod size$

linear: $f(i) = i$ — previous diagram

quadratic: $f(i) = i^2$

double hashing $f(i) = i \times h_2(k)$ (second hash function)

quadratic probing example

$h(k) = 3k + 7$

index $= h(k) \mod 10$

then check $h(k) + 1^2 \mod 10$, $h(k) + 2^2 \mod 10$, etc.

insert 4, 27, 14, 37, 22, 34

$h(k) =$ 19, 88, 49, 118, 73, 109

index	
0	
1	
2	
3	
4	
5	
6	
7	
8	
9	

quadratic probing example

$h(k) = 3k + 7$

index $= h(k) \mod 10$

then check $h(k) + 1^2 \mod 10$, $h(k) + 2^2 \mod 10$, etc.

insert 4, 27, 14, 37, 22, 34

$h(k) =$ 19, 88, 49, 118, 73, 109

index	
0	
1	
2	
3	
4	
5	
6	
7	
8	
9	4

quadratic probing example

$h(k) = 3k + 7$

index $= h(k) \mod 10$

then check $h(k) + 1^2 \mod 10$, $h(k) + 2^2 \mod 10$, etc.

insert 4, 27, 14, 37, 22, 34

$h(k) =$ 19, 88, 49, 118, 73, 109

index	
0	
1	
2	
3	
4	
5	
6	
7	
8	27
9	4

quadratic probing example

$h(k) = 3k + 7$

index $= h(k) \mod 10$

then check $h(k) + 1^2 \mod 10$, $h(k) + 2^2 \mod 10$, etc.

insert 4, 27, 14, 37, 22, 34

$h(k) =$ 19, 88, 49, 118, 73, 109

index	
0	14
1	
2	
3	
4	
5	
6	
7	
8	27
9	4

quadratic probing example

$h(k) = 3k + 7$

index $= h(k) \mod 10$

then check $h(k) + 1^2 \mod 10$, $h(k) + 2^2 \mod 10$, etc.

insert 4, 27, 14, 37, 22, 34

$h(k) =$ 19, 88, 49, 118, 73, 109

index	
0	14
1	
2	37
3	
4	
5	
6	
7	
8	27
9	4

quadratic probing example

$h(k) = 3k + 7$

index $= h(k) \mod 10$

then check $h(k) + 1^2 \mod 10$, $h(k) + 2^2 \mod 10$, etc.

insert 4, 27, 14, 37, 22, 34

$h(k) =$ 19, 88, 49, 118, 73, 109

index	
0	14
1	
2	37
3	22
4	
5	
6	
7	
8	27
9	4

quadratic probing example

$h(k) = 3k + 7$

index $= h(k) \mod 10$

then check $h(k) + 1^2 \mod 10$, $h(k) + 2^2 \mod 10$, etc.

insert 4, 27, 14, 37, 22, 34

$h(k) =$ 19, 88, 49, 118, 73, 109

index	
0	14
1	
2	37
3	22
4	
5	34
6	
7	
8	27
9	4

quadratic probing example

$h(k) = 3k + 7$

index $= h(k) \mod 10$

then check $h(k) + 1^2 \mod 10$, $h(k) + 2^2 \mod 10$, etc.

insert 4, 27, 14, 37, 22, 34

$h(k) =$ 19, 88, 49, 118, 73, 109

index	
0	14
1	
2	37
3	22
4	
5	34
6	
7	
8	27
9	4

probing possibilities

$h(k) + f(i) \mod size$

linear: $f(i) = i$ — previous diagram

quadratic: $f(i) = i^2$

double hashing $f(i) = i \times h_2(k)$ (second hash function)

double hashing example

$h(k) = k$

index $= h(k) \mod 10$

then check $h(k) + h_2(k) \mod 10$, $h(k) + 2h_2(k) \mod 10$, etc.

…where $h_2(k) = 7 - (k \mod 7)$

insert 89, 18, 58, 49, 69, 60

index	
0	
1	
2	
3	
4	
5	
6	
7	
8	
9	

double hashing example

$h(k) = k$

index $= h(k) \mod 10$

then check $h(k) + h_2(k) \mod 10$,
$h(k) + 2h_2(k) \mod 10$, etc.

...where $h_2(k) = 7 - (k \mod 7)$

insert 89, 18, 58, 49, 69, 60

index	
0	
1	
2	
3	
4	
5	
6	
7	
8	
9	89

double hashing example

$h(k) = k$

index $= h(k) \mod 10$

then check $h(k) + h_2(k) \mod 10$, $h(k) + 2h_2(k) \mod 10$, etc.

...where $h_2(k) = 7 - (k \mod 7)$

insert 89, 18, 58, 49, 69, 60

index	
0	
1	
2	
3	
4	
5	
6	
7	
8	18
9	89

double hashing example

$h(k) = k$

index $= h(k) \mod 10$

then check $h(k) + h_2(k) \mod 10$, $h(k) + 2h_2(k) \mod 10$, etc.

...where $h_2(k) = 7 - (k \mod 7)$

insert 89, 18, 58, 49, 69, 60

index	
0	
1	
2	
3	58
4	
5	
6	
7	
8	18
9	89

double hashing example

$h(k) = k$

index $= h(k) \mod 10$

then check $h(k) + h_2(k) \mod 10$, $h(k) + 2h_2(k) \mod 10$, etc.

...where $h_2(k) = 7 - (k \mod 7)$

insert 89, 18, 58, 49, 69, 60

index	
0	
1	
2	
3	58
4	
5	
6	49
7	
8	18
9	89

double hashing example

$h(k) = k$

index $= h(k) \mod 10$

then check $h(k) + h_2(k) \mod 10$, $h(k) + 2h_2(k) \mod 10$, etc.

...where $h_2(k) = 7 - (k \mod 7)$

insert 89, 18, 58, 49, 69, 60

index	
0	69
1	
2	
3	58
4	
5	
6	49
7	
8	18
9	89

double hashing example

$h(k) = k$

index $= h(k) \mod 10$

then check $h(k) + h_2(k) \mod 10$,
$h(k) + 2h_2(k) \mod 10$, etc.

...where $h_2(k) = 7 - (k \mod 7)$

insert 89, 18, 58, 49, 69, 60

index	
0	69
1	
2	60
3	58
4	
5	
6	49
7	
8	18
9	89

double hashing thrashing

$h(k) = k$; $h_2(k) = (k \mod 5) + 1$

index $= h(k) \mod 10$

then check $h(k) + h_2(k) \mod 10$,
$h(k) + 2h_2(k) \mod 10$, etc.

insert 10, 12, 14, 16, 18, 36

index	
0	
1	
2	
3	
4	
5	
6	
7	
8	
9	

double hashing thrashing

$h(k) = k$; $h_2(k) = (k \mod 5) + 1$

index $= h(k) \mod 10$

then check $h(k) + h_2(k) \mod 10$,
$h(k) + 2h_2(k) \mod 10$, etc.

insert 10, 12, 14, 16, 18, 36

index	
0	10
1	
2	12
3	
4	14
5	
6	16
7	
8	18
9	

double hashing thrashing

$h(k) = k$; $h_2(k) = (k \mod 5) + 1$

index $= h(k) \mod 10$

then check $h(k) + h_2(k) \mod 10$,
$h(k) + 2h_2(k) \mod 10$, etc.

insert 10, 12, 14, 16, 18, 36

$h(36) \mod 10 = 6$

$h(36) + h_2(36) \mod 10 = 36 + 2 \mod 10 = 8$

$h(36) + 2h_2(36) \mod 10 = 0$; $h(36) + 3h_2(36) \mod 10 = 2$

$h(36) + 4h_2(36) \mod 10 = 4$; $h(36) + 5h_2(36) \mod 10 = 6$

index	
0	10
1	
2	12
3	
4	14
5	
6	16
7	
8	18
9	

why prime sizes

prime sizes prevent this problem

$h_2(k)$ (2) was not relatively prime to table size (10)

result: didn't use all elements of table

similar issues with i^2, etc. (but not as bad)

why prime sizes (2)

often use prime sizes w/o open addressing

why — more forgiving for not great hash functions

example: $h(k) = k$

are most keys even? — oops if table size is even
are most keys $10k$? — oops if table size is multiple of 5

handling removal

with chaining: easy

- remove from linked list

with open addressing: hard

- need to not disrupt searches
- option 1: rehash every time (super-expensive)
- option 2: placeholder value + rehash eventually
- option 3: disallow deletion (lab 6)

cryptographic hashes

example: SHA-256

input: any string of bits

output: 256 bits

have security properties normal hashes don't:

- collision resistence
- preimage resistence

cryptographic hashes

example: SHA-256

input: any string of bits

output: 256 bits

have security properties normal hashes don't:

- **collision resistence**
- preimage resistence

collision resistence

security property of a cryptographic hash

it's very hard to find keys k_1 and k_2 so $h(k_1) = h(k_2)$

note: why SHA-256's output is so big (256 bits)

otherwise, just generate lots of hashes...

example application: verify download with hash of file contents

it's very hard to find two files with the same hash

even if you're trying

exercise: collision non-resistence

```
unsigned int hash(const string &s) {
    unsigned int sum = 0;
    for (int i = 0; i < s.size(); ++i) {
        // deliberate use of wraparound on overflow
        sum *= 37;
        sum += s[i];
    }
    return sum;
}
```

exercise: how to construct two strings with same hash?

exercise: collision non-resistence

```
unsigned int hash(const string &s) {
    unsigned int sum = 0;
    for (int i = 0; i < s.size(); ++i) {
        // deliberate use of wraparound on overflow
        sum *= 37;
        sum += s[i];
    }
    return sum;
}
```

exercise: how to construct two strings with same hash?

one idea: $\{60, x\}$ and $\{59, x + 37\}$ have the same hash

cryptographic hashes

example: SHA-256

input: any string of bits

output: 256 bits

have security properties normal hashes don't:

- collision resistence
- **preimage resistence**

preimage resistence

security property of a cryptographic hash

if given V, very hard to find k so $h(k) = V$

collision resistence usually implies preimage resistence

some cryptographic hash applications

verifying downloads

- get short hash from trusted source
- get big file from less trusted source
- use hash to make sure it's the right big file

message authentication

- did message 'X' really come from where I thought?
- usually: "magic" math operation that works on small amount of data
- hash turns big message into small amount of data