Lecture 21-22

Hash Tables

Hash Tables

Recap of BSTs

- Balanced BSTs offer O(log n) time costs for add/remove/find operations by exploiting order relationships among data.

- They are also memory efficient, in that only as many nodes are allocated as elements are contained in the BST

- However...

...by utilizing more memory (greater space complexity), we can achieve even higher performance (lower time complexity)

Hash sets, hash tables and **hash maps** offer **O(1+alpha*)** time costs for add/remove/find operations by investing more memory in the underlying storage.

* load factor

DSC30 gang

- (Marina, 1) ← The Boss
- (Kalkin, 2) ← Tutor 1
- (Cassidy, 3) ← Tutor 2
- ……
- (Elif, 25) ←Student 1
- (Grace, 26) ← Student 2
- …..

How to store them?

Direct hashing

- (Marina, 1) ← The Boss
- (Kalkin, 2) ← Tutor 1
- (Cassidy, 3) ← Tutor 2
- ……
- (Sam, 25) ←Student 1
- (Ethan, 26) ← Student 2
- …..

0	1	2	3	…	25	26	….
X	Marina	Kalkin	Cassidy		Elif	Grace	

Running time for Direct Hashing

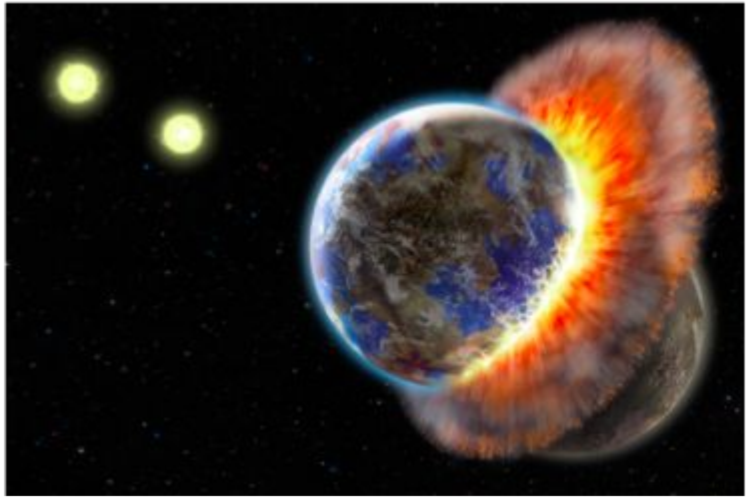
```
Class StudentDataBase {
  Student[] allStudents = new Student [4294967268];
  void add (Student s) {
  int index = s.studentId;
  allStudents[index]=s;
}
 Student get (Student s) {
  int index = s.studentId;
  return allStudent[index];
}
void remove(Student s) {
  int index = s.studentId;
  allStudents[index] = null;
}
```

Space complexity is horrible :(

- Allocating 4 billion entries for a UCSD student database is extremely wasteful.
    - The universe of keys is far larger than the number we expect to ever want to store.
- What if we decrease the size of the table from 4 billion entries to storing, say, just 100,000?
- 100,000 is still far higher than the actual number of UCSD students, so there should be plenty of space.

Collision

- With an array of only 100,000 entries, each student ID would no longer have its own unique array index -- multiple student IDs would have to "share" an index.
    - We call the "sharing" of an array index by 2 (or more) student IDs a collision.
- Whenever a collision occurs, we have to store the Student object "somewhere else" (more later).



Collision

- However, if we're **clever** about how we assign array indices to student IDs, then collisions will rarely occur.

- We can still achieve *O(1+..)* add/find/remove time in the *average case.*

Idea: Hash Tables

- A hash table consists of a large array of M “slots” (or “buckets”) to store the user’s data.

- A hash table also requires:

  1. Some way of converting from an object’s ***key*** into an ***index*** that specifies where that object should be stored. This is called a hash function.
  2. A method of handling collisions.

In order to ensure good performance, M (number of cells) must be bigger than N, the number of data the user will want to store.

Do we have a collision?

M=5; (size of the hash table)

Student1 id is 1;

Student2 id is 4;

Student3 id is 8;

A: Yes

B: No

C: May be

Hash function

- A *hash function* maps an object's *key* into an array index, i.e., a number from 0...M-1, where *M* is the number of entries in the hash table.
- **Example**:

```
int hashFucntion (int studentID){
    return studentID % M; }
```

The modulus operator % divides studentID by M, and then returns the remainder. Examples:

- 3 % 10 = ?
- 107 % 10 = 7
- 7 % 4=3
- 16 % 5 = 1

Do we have a collision?

M=5; (size of the hash table)

Student1 id is 1;

Student2 id is 4;

Student3 id is 8;

A: Yes

B: No

C: May be

Hash function

- To be useful, a hash function must be **fast**
  - Its performance should not depend on the particular key.
  - O(1+..) (otherwise how can we achieve desired O(1+..) performance?)

Hash function

- To be useful, a hash function must be **fast**
  - Its performance should not depend on the particular key.
  - O(1+..) (otherwise how can we achieve desired O(1+..) performance?)

- A hash function must also be **deterministic**:
  - Given the same key, it must always return the same array index. (Otherwise, how would we find something we stored earlier?)

Hash function

- To be useful, a hash function must be **fast**
  - Its performance should not depend on the particular key.
  - O(1+..) (otherwise how can we achieve desired O(1+..) performance?)

- A hash function must also be **deterministic**:
  - Given the same key, it must always return the same array index. (Otherwise, how would we find something we stored earlier?)

- A “good” hash function should also be **uniform**:
  - Each “slot” i in the array should be equally likely to be chosen as any other slot j.

Is it a good hash function?

```
int hashFucntion (int studentID){
 return M/2; } //M is a size of a Hash Table
```

A: Yes, I like it

B: No, it is not fast

C: No, it is not deterministic

D: No, it is not uniform

Hash function

- For instance, if `M` is 100000, then `studentID % 100000` is simply the *last 5 digits* of the student ID, e.g.:
  - `student1` with Student ID `0000013012` would map to index `13012`.
  - `student2` with Student ID `1234567890` would map to index `67890`.
  - These *indices* specify *where* in the array the students are *stored.*

Key (student ID)	*Value* (reference to Student object)
...	
13011	
13012	`student1`
13013	
...	
67889	
67890	`student2`
67891	
67892	
...	...

Handling collisions

- Unfortunately, on occasion, there would be two (or more) `Student` objects who are "hashed" (mapped) into the same array slot.

```
studentID1 = 2200012345;
studentID2 = 1926112345;
hashTable.add(studentID1, student1);
hashTable.add(studentID2, student2);
```

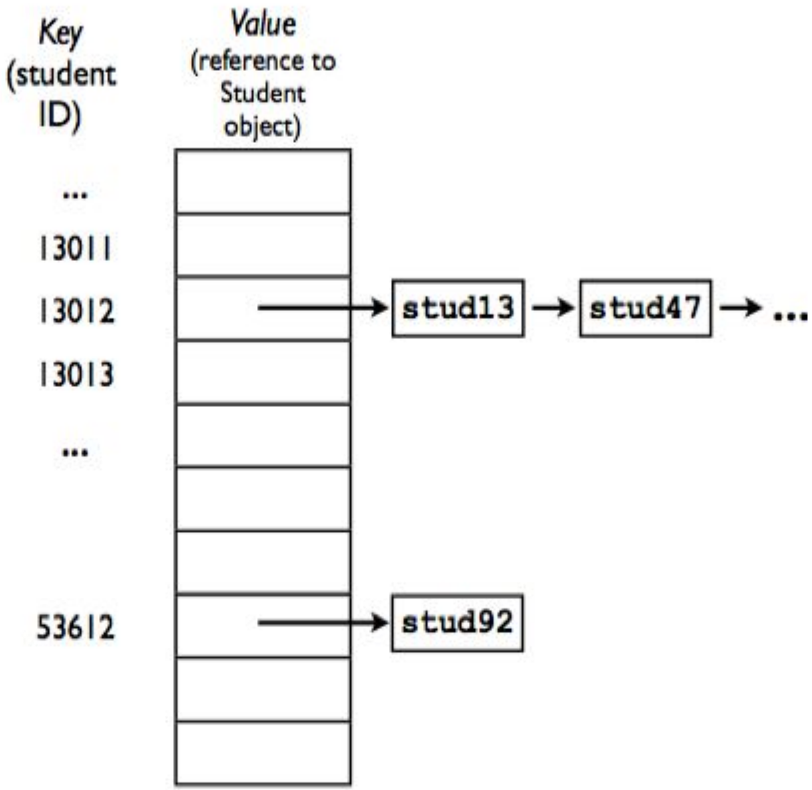
- This is called a *collision* -- two different `Student` objects map into the *same array index*.
- How do we handle these collisions?

Handling collisions

- There are two principal ways of handling collisions:
  1. **Chaining** (aka **separate chaining**) -- at each slot in the array, instead of storing only a single element, we store a linked list of elements.
  2. **Open addressing** -- if `student5` "hashes" to array index 123, and array index 123 is already occupied, then we look for "another" index at which to store `student5`, e.g., 124.
     - Different schemes for determining "another index".

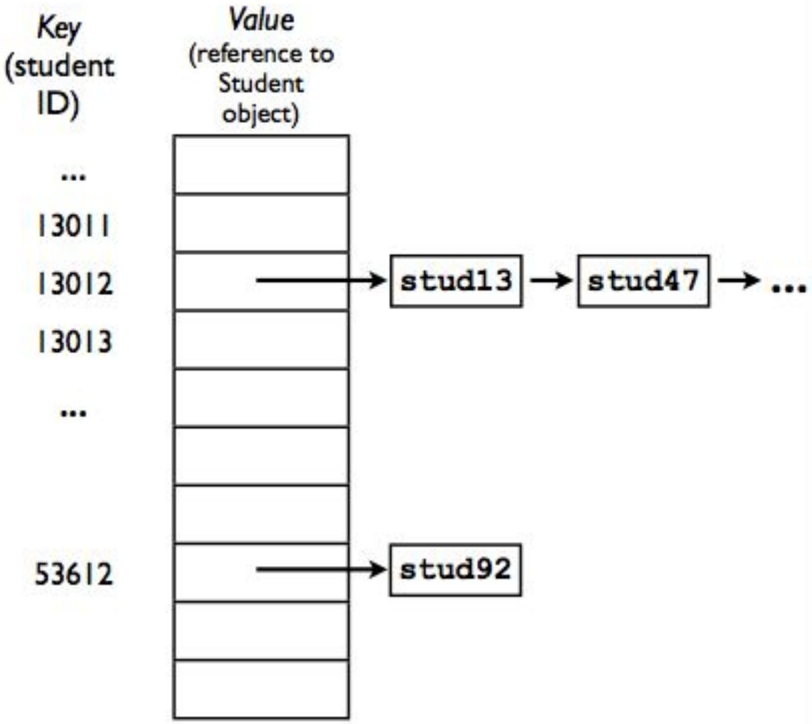
Chaining

- Each slot in the array contains not an object itself, but rather a pointer to the *head* of a *linked list* of objects which all map to the same index.



Chaining

- When looking for a particular object, we must:
  1. *Hash* the key to obtain the *index*.
  2. *Search* the list for the correct object.
- This will still be *fast* as long as the linked lists are *short* (more later).

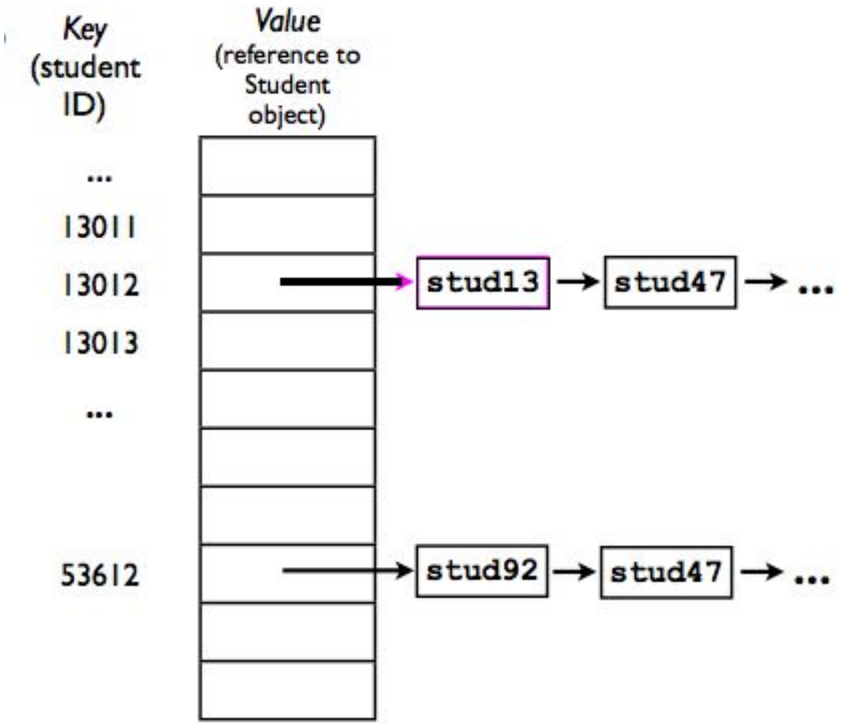


Assume a good hash function

Is this possible?

A: Yes

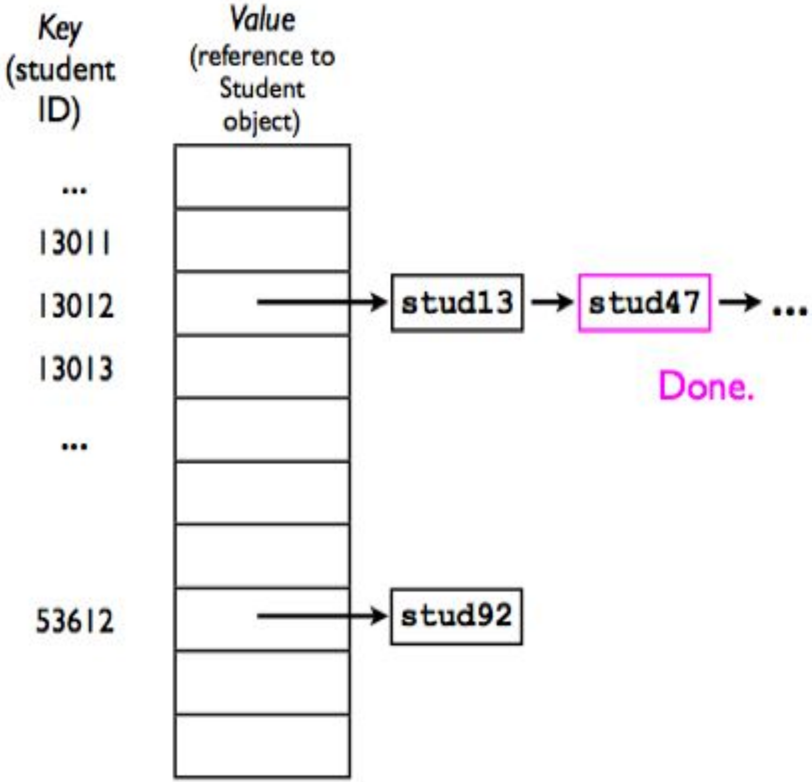
B: No



Chaining

- For example, if we wish to find `stud47` with student ID `0925113012`.

1. Hash `stud47`'s student ID to determine the *index*.
2. Jump to the head of the corresponding linked list.
3. Traverse the linked list until we find `stud47`.



Chaining: complexity

How fast are the **add/remove/find** operations for a hash table with chaining?

To find an object in a hash table, we must:

- **Hash the key**. A: *O(1), B: O(n),* C: unknown
- **Jump to that array index**. A: *O(1), B: O(n), C: unknown*
- **Traverse** through the linked list at that index (worst).

  A: O(1)

  B: O(n)

  C: Something else

Chaining: performance analysis

In the *worst case, all N* objects stored in the hash table will hash to the *same array index*

- This means that the linked list at that index will be *N* elements long.
- To find an arbitrary element in a linked list we need *O(N)* time.
- This is no better than just using a linked list by itself!

Chaining: performance analysis

- However, in the average case, a hash table performs much better:
  - Given M slots in the array and N objects to store, the average list length for any array slot is N/M. (Goes without proof)
  - Then, the average time to access any arbitrary object is
  - O(1+N/M). (Goes without proof).

- Now, suppose that we always make sure that M>N:
  - Then N/M < 1.
  - Therefor, average-case time cost is O(1+N/M) = O(1)

Load factor

Load factor $\alpha = N/M$ of the hash table.

The proper way to report a running time for hash tables is:

$O(1+\alpha)$, where α is a load factor. If α becomes too large, then the running time is not a constant anymore.

For separate chaining load factor of 1 is OK. If it becomes larger:

double the size of the table and rehash.

reminders

*mic

PA8, 9

Open addressing

Idea

- In a case of a collision, we look for another available cell.

- Must be deterministic
  - If we want to find an element, we should visit the same cells.

Simplest: linear probing

- If **hashFunction(key)** maps into an index *i* that is already occupied, then try *i+1.*
- If that doesn't work, try *i+2, i+3,* ..., etc.
- If we get to *M-1,* we want to "wrap around" back to 0.

- The index of the *jth* probe (where *j* starts at 0) is given by the expression (i+j) % M
- You can think of the *probe* as one step in your search

Exercise

- The index of the *jth* probe (where *j* starts at 0) is given by the expression (i+j) % M

- Insert the given numbers in the hash table (M=7), resolving collisions by linear probing (use %M for hash function)

1, 15, 2, 7, 14, 22

$\%7$

1, 15, 2, 7, 14, 22

Click to add text

$1\%7 \rightarrow 1$

$15\%7 \rightarrow 1$

$2\%7 \rightarrow 2$

$7\%7 \rightarrow 0$

$14\%7 \rightarrow 0$

$22\%7 \rightarrow 1$

Primary clustering

- If too many keys hash to the same index -- *or to nearby indices* -- then the linear probing may become expensive.
- Consider the hash table to the right:
    - 13011-13016 are already occupied.
    - If we want to add another student `student7` who also hashes to 13011, then we have to step through 7 elements.
    - The longer the cluster, the higher the time cost for add/find/remove.

Key (student ID)	*Value* (reference to Student object)
...	
13011	`student5`
13012	`student1`
13013	`student9`
13014	`student8`
13015	`student3`
13016	`student4`
13017	
...	

Removing an element

Suppose we want to remove student1 from the hash table:

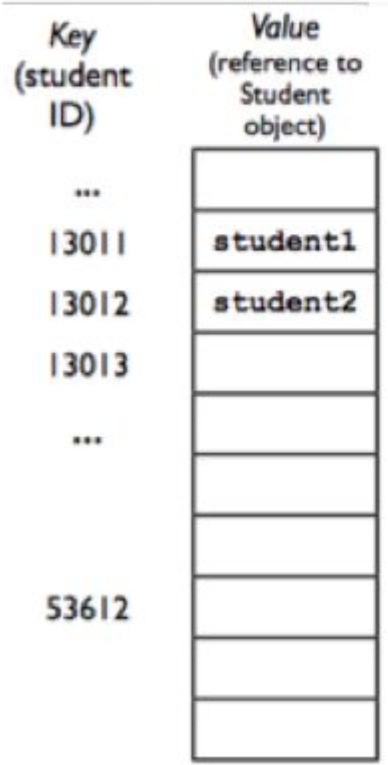
```
Table[13011] = null;
```

Good idea?

A: Yes.

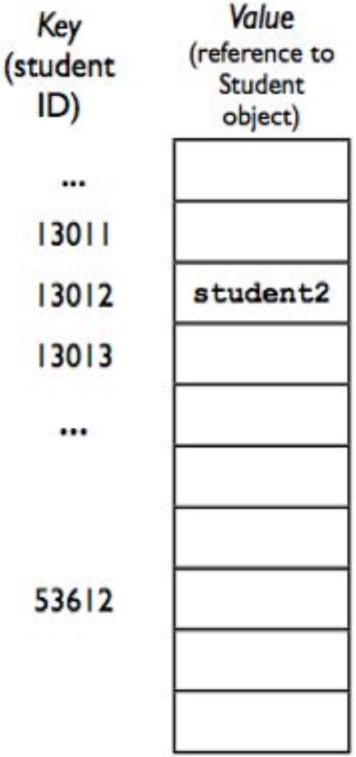
B: No

C: Seems like a bad idea but I do not know why.



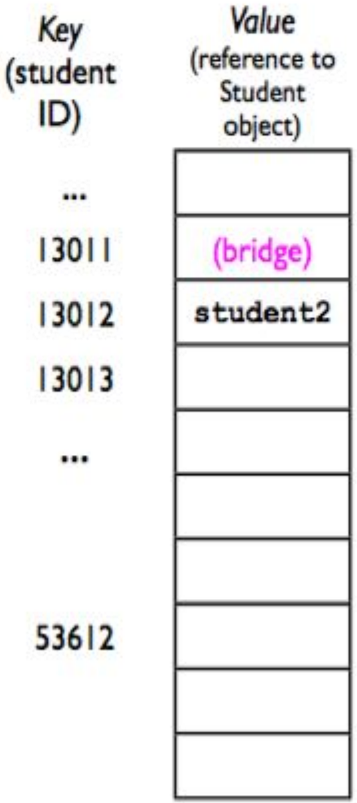
Removing an element

- Suppose we remove `student1` from the hash table.
- If we later search for `student2`, we will still hash to 13011, but find that it is *empty*.
  - Does that mean `student2` is not contained in the hash table?
  - No -- but we have to *record* that somehow.



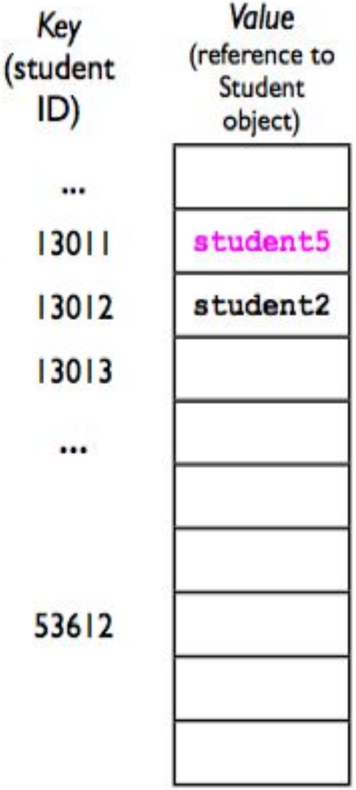
Removing an element (lazy deletion)

- One method of recording that an element was deleted is a *bridge*, a special element that indicates "empty, but keep looking."
- If we later add another element, say `student5` that hashes to 13011, then we can *replace* the bridge with a real `Student` object.



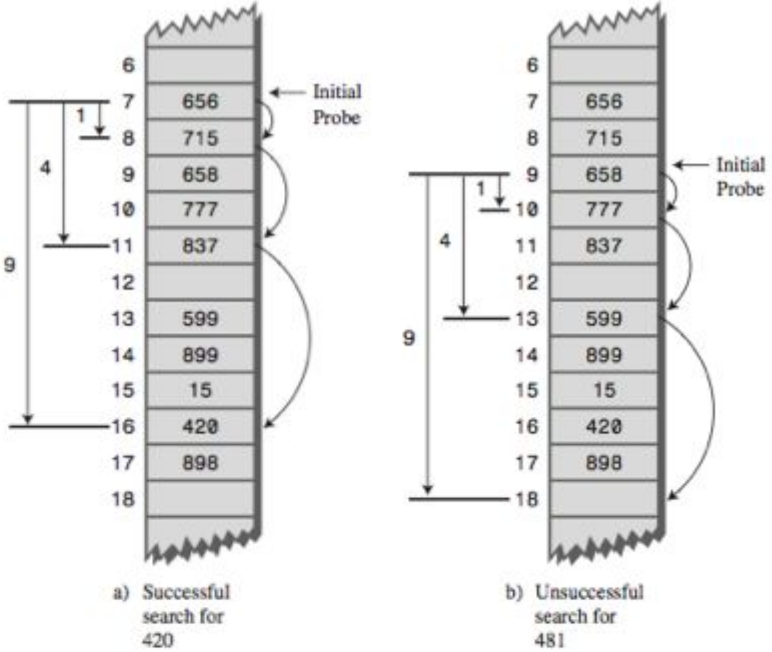
Removing an element (lazy deletion)

- One method of recording that an element was deleted is a *bridge*, a special element that indicates "empty, but keep looking."
- If we later add another element, say `student5` that hashes to 13011, then we can *replace* the bridge with a real `Student` object.



Quadratic probing

- Quadratic probing is an attempt to keep clusters from forming. The idea is to probe more widely separated cells, instead of those adjacent to the primary hash site.



- In quadratic probing, probes go to $x+1$, $x+4$, $x+9$, $x+16$, $x+25$, and so on. The distance from the initial probe is the square of the step number: $x+1^2$, $x+2^2$, $x+3^2$, $x+4^2$, $x+5^2$, and so on

The Problem with Quadratic Probes

- Quadratic probes suffer from a different and more subtle clustering problem.
- Let’s say 184, 302, 420, and 544 all hash to 1 and are inserted in this order.
- What is the hash table after we insert these numbers? (assume the table is large enough)

- 184, 302, 420, and 544

Secondary clustering

- Quadratic probes suffer from a different and more subtle clustering problem.
  - This occurs because all the keys that hash to a particular cell follow the same sequence in trying to find a vacant space.

- Let's say 184, 302, 420, and 544 all hash to 1 and are inserted in this order. Each additional item with a key that hashes to 1 will require a longer probe.
  - This phenomenon is called secondary clustering.
- Secondary clustering is not a serious problem, but quadratic probing is not often used because there's a slightly better solution.

Double Hashing

Double Hashing

Idea: generate probe sequences that depend on the **key** instead of being the same for every key.

Then numbers with *different* keys that hash to the same index will use *different* probe sequences.

The solution is to hash the key a **second** time, using a different hash function, and use the result as the step size.

- $h(k, i) = (h_1(k) + i\, h_2(k)) \bmod m$.
  - h_1 and h_2 are auxiliary hash functions.
  - i is the probe step

Example: $h(k, i) = (h_1(k) + i\, h_2(k)) \bmod m$.

- Assume M = 13 (size of the table)
- $h_1(k) = k \bmod 13$
- $h_2(k) = 1 + (k \bmod 11)$.

Let’s insert 79, 69, 98, 72, 14, 50 using double hashing.

$h(k, i) = (h_1(k) + i\, h_2(k)) \bmod m$; 79, 69, 98, 72, 14, 50; h1(k) = k mod 13, h2(k) = 1 + (k mod 11

Restrictions on h_2

- **Experience** has shown that this secondary hash function must have certain characteristics:

1. It must not be the same as the primary hash function.
2. It must never output a 0 (otherwise, there would be no step;

   every probe would land on the same cell, and the algorithm would go into an endless loop).

Analysis of open-address hashing

- Load factor $\alpha = N/M$ of the hash table. Since at most one element occupies each slot and M=>N, α<=1.
- Facts (without proof).
  - For linear probing the load factor can't be more than 2/3 (under ½ is the best)
  - For quadratic probing and double hashing load factor of 2/3 is OK.
  - If the load factor increases, you must re-hash: create a larger array and **re-hash** every element into a new hash table.