CSE 12:
Basic data structures and object-oriented design

Jacob Whitehill
jake@mplab.ucsd.edu

Lecture Thirteen
26 July 2012

Hash tables.

Recap of BSTs

- Binary search trees (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 tables** offer *O*(1) time costs for **`add`**/**`remove`**/**`find`** operations by investing more memory in the underlying storage.

Finite "universe" of objects to store

- Suppose you were developing a database of UCSD students.
- You may wish to build a function that allows you to retrieve a student's full record given their student ID:

```
class Student {
  int _studentID;  // 4 bytes long
  String _firstName, _lastName;
  String _address;
  String _favoriteColor;
}
```

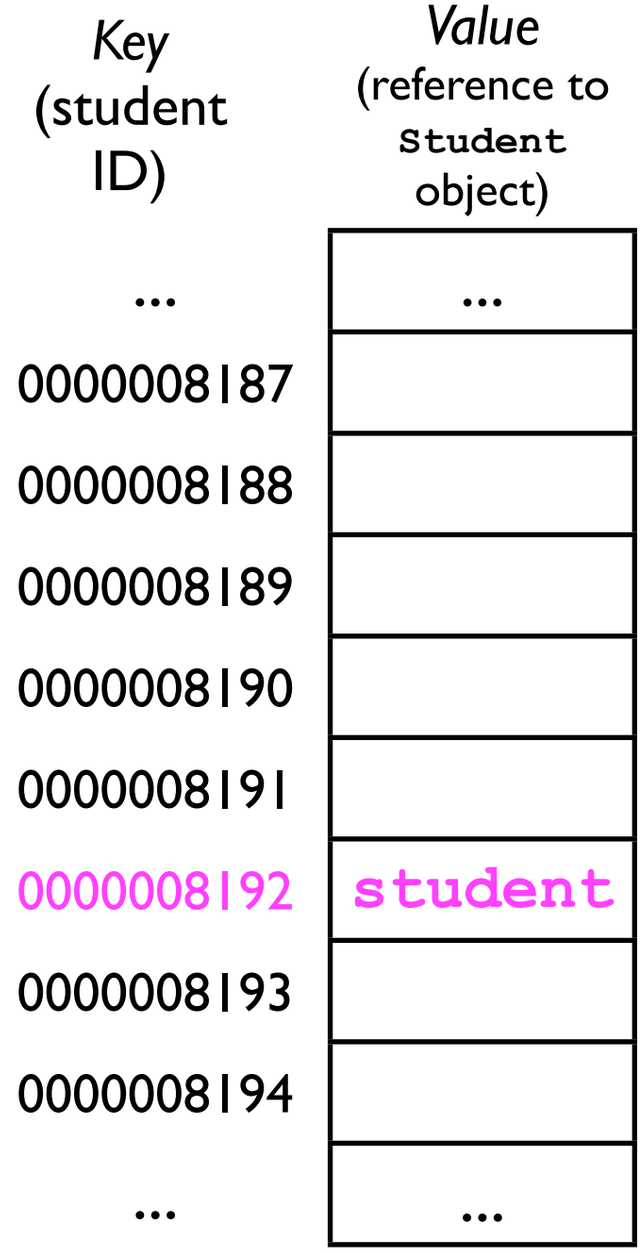
- In this application, the *key* would be the student ID, and the *value* associated with the key would be the whole **`Student`** record.

Finite "universe" of objects to store

- In this application (and others), there may exist only a finite **universe** of possible keys values.
  - For instance, a 4-byte integer can only store about 4 billion different values ==> 4 billion unique keys.
- Sometimes this finite set of keys is small enough that we can allocate an array big enough to give *every possible key its own slot.*
- In this case, we can make the "search" process for a particular key *trivial*:
  - We simply "jump" to the unique array index assigned to that key.
  - This takes only *O*(1) time in the *worst*-case.

Finite "universe" of objects to store

- For example, in a UCSD student database application, we could allocate an array 4 billion elements long.
- Whenever we wished to find a particular student, we simply *jump* to that student's correct slot using the student ID as the *index*.
  - E.g., student with ID# 000000008192 would be stored at slot 000000008192.
- Search is *trivial* -- *O*(1).

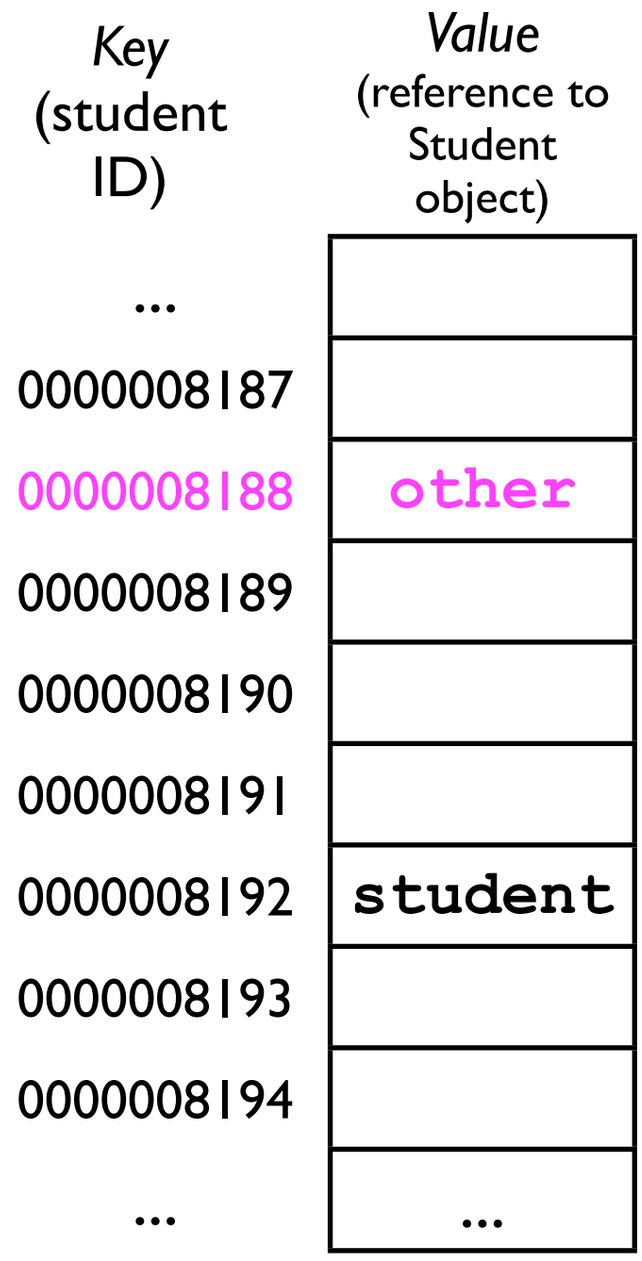


Finite "universe" of objects to store

- Similarly, when adding a student, we simply insert an entry at his/her *unique location* in the array:

```
Student other =
  new Student(8188, ...);
_container.add(other);
```

- Since student IDs are guaranteed to be unique, there is exactly *one* element that will ever reside at the index of the array.



Finite "universe" of objects to store

- Using the idea sketched above, we could implement a **`StudentDatabase`** with excellent time costs.

```
class StudentDatabase {
  Student[] _allPossibleStudents = new Student[4294967296];

  void add (Student s) {  // O(1)
    int index = s._studentID;
    _allPossibleStudents[index] = s;
  }

  // Returns null if s is not in the database
  Student get (Student s) {  // O(1)
    int index = s._studentID;
    return _allPossibleStudents[index];
  }

  void remove (Student s) {  // O(1)
    int index = s._studentID;
    _allPossibleStudents[index] = null;
  }
}
```

Finite "universe" of objects to store

- However, the space costs of this data structure are enormous.

```
class StudentDatabase {
  Student[] _allPossibleStudents = new Student[4294967296];

  void add (Student s) {  // O(1)
    int index = s._studentID;
    _allPossibleStudents[index] = s;
  }

  // Returns null if s is not in the database
  Student get (Student s) {  // O(1)
    int index = s._studentID;
    return _allPossibleStudents[index];
  }

  void remove (Student s) {  // O(1)
    int index = s._studentID;
    _allPossibleStudents[index] = null;
  }
}
```

Finite "universe" of objects to store

- 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 decreased 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.

Finite "universe" of objects to store

- 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).
  - 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*.

Hash tables

- This idea is called a "hash table."
- 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* must be bigger than *N*, the number of data the user will want to store.

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.

- Simple example:

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

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

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

Hash function

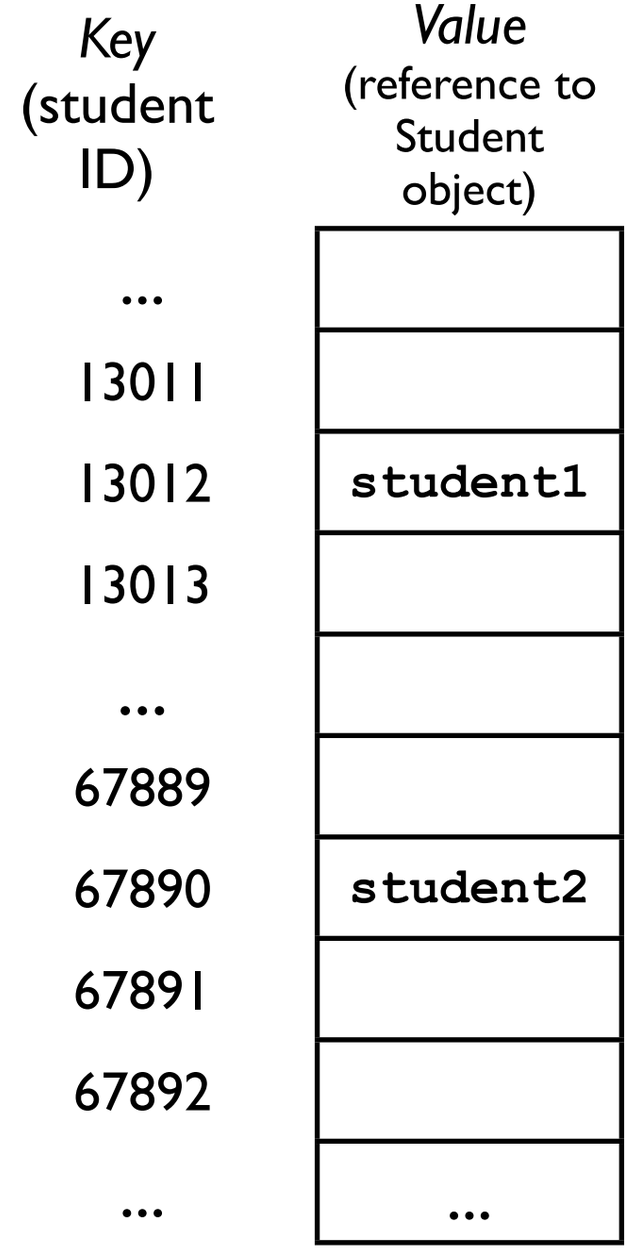
- To be useful, a hash function must be *fast*.
  - Its performance should not depend on the particular key.
- A hash function must also be *deterministic*:
  - Given the *same key*, it must *always* return the *same array index*.
    - (Otherwise, how would we find something we stored earlier?)

Hash function

- 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*.
  - We certainly don't want to map every possible key to the same array index!
  - Uniformity is important to ensure good performance.

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*.



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:

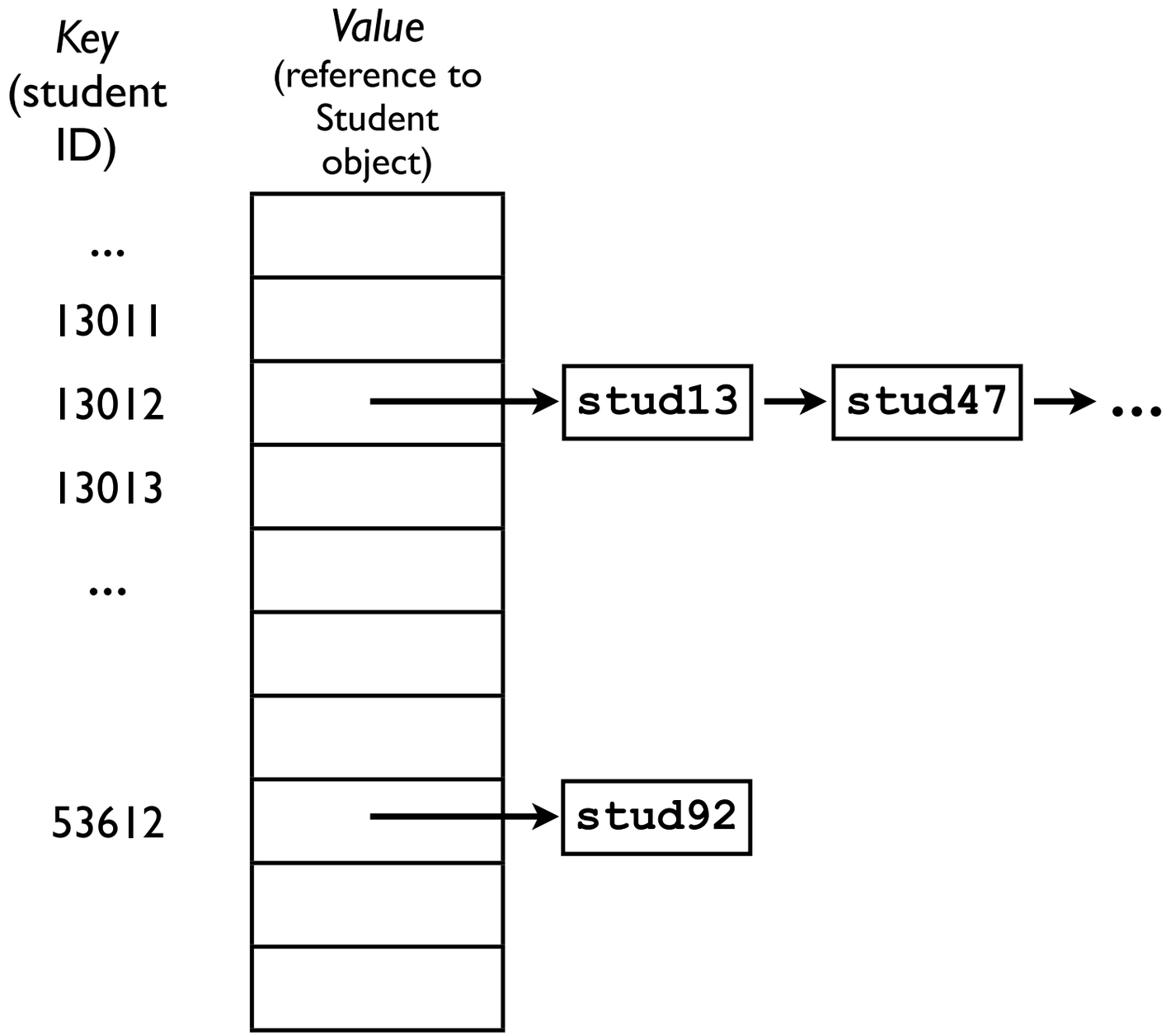
  1. **Chaining** (aka **separate chaining**) -- at each slot in the array, instead of storing only a single element, we store a linked list of elements.

  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.

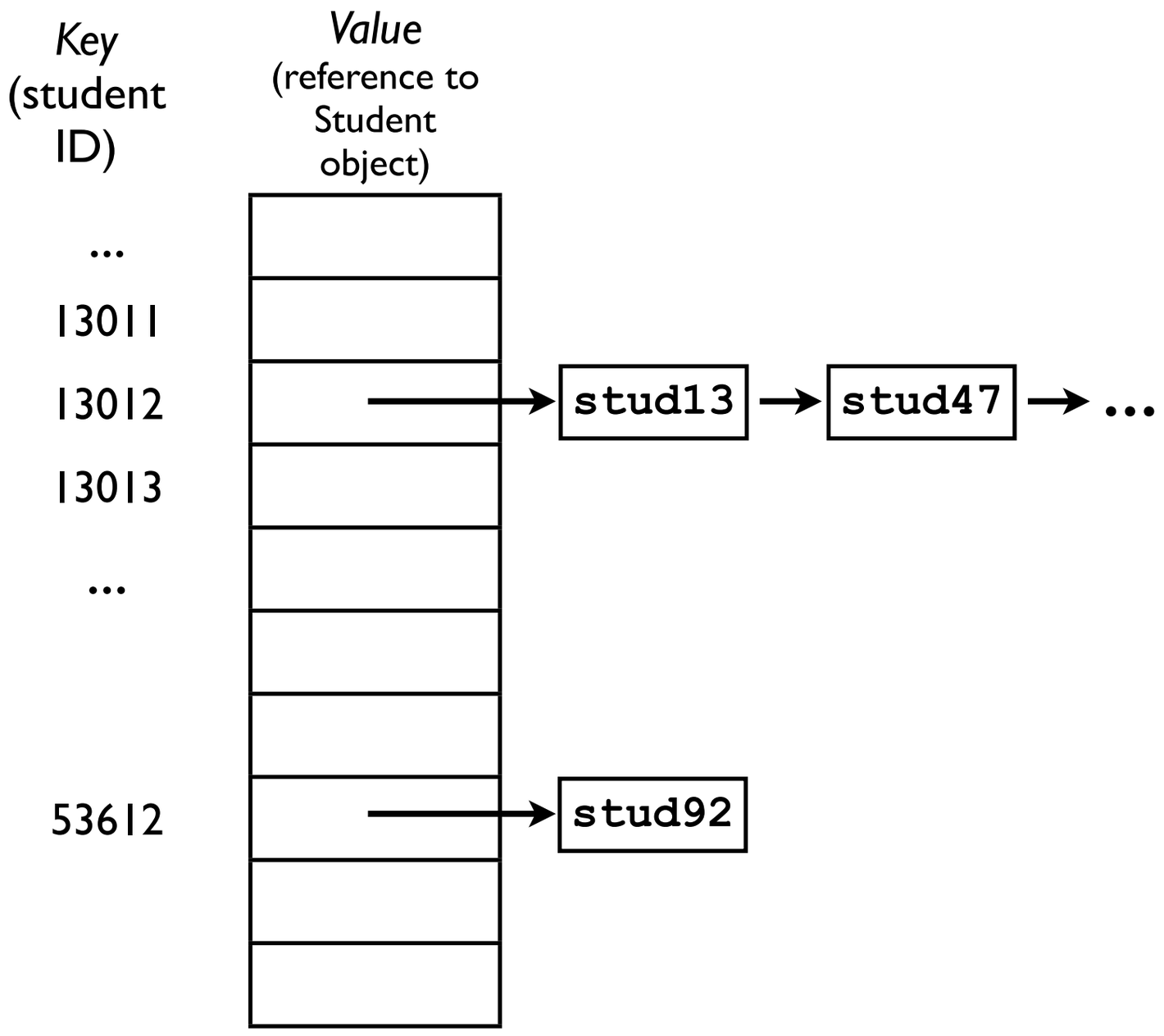


Chaining

- When looking for a particular object, we must:

  1. *Hash* the key to obtain the *index*.
  2. *Search* the list for the correct object.

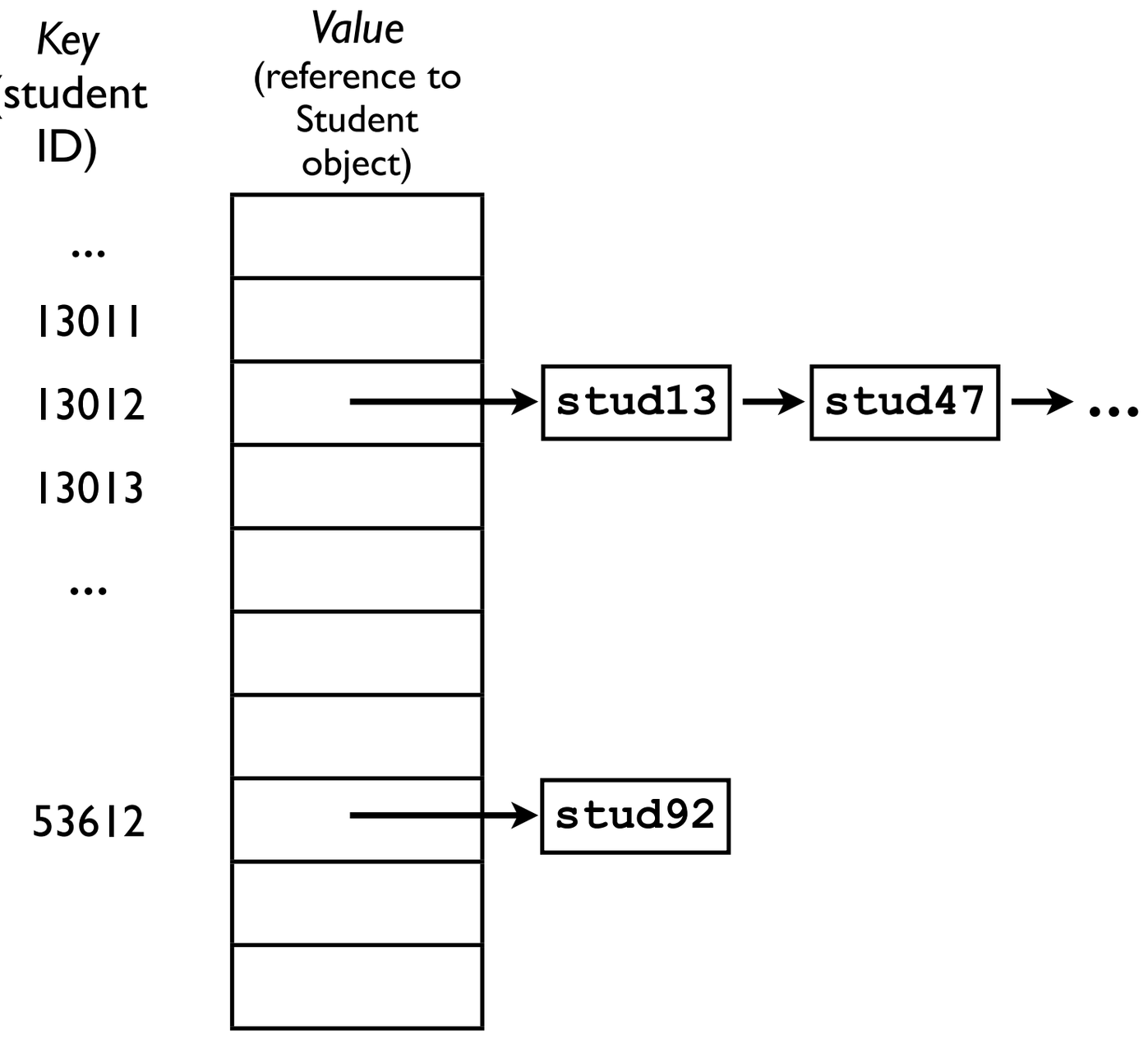
- This will still be *fast* as long as the linked lists are *short* (more later).



Chaining

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

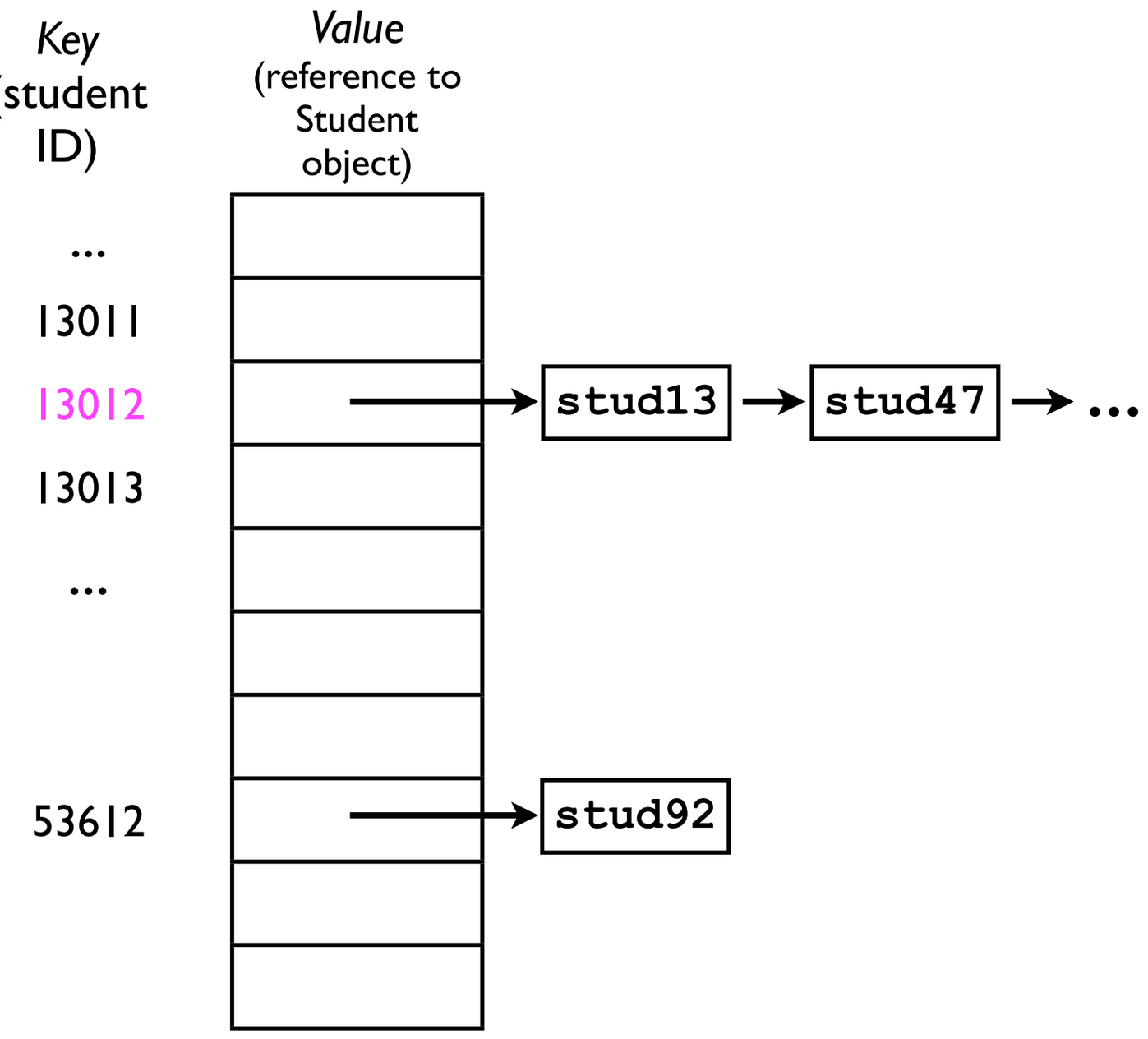
1. Hash `stud47`'s student ID to determine the *index*.
2. Jump to the head of the corresponding linked list.
3. Traverse the linked list until we find `stud47`.



Chaining

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

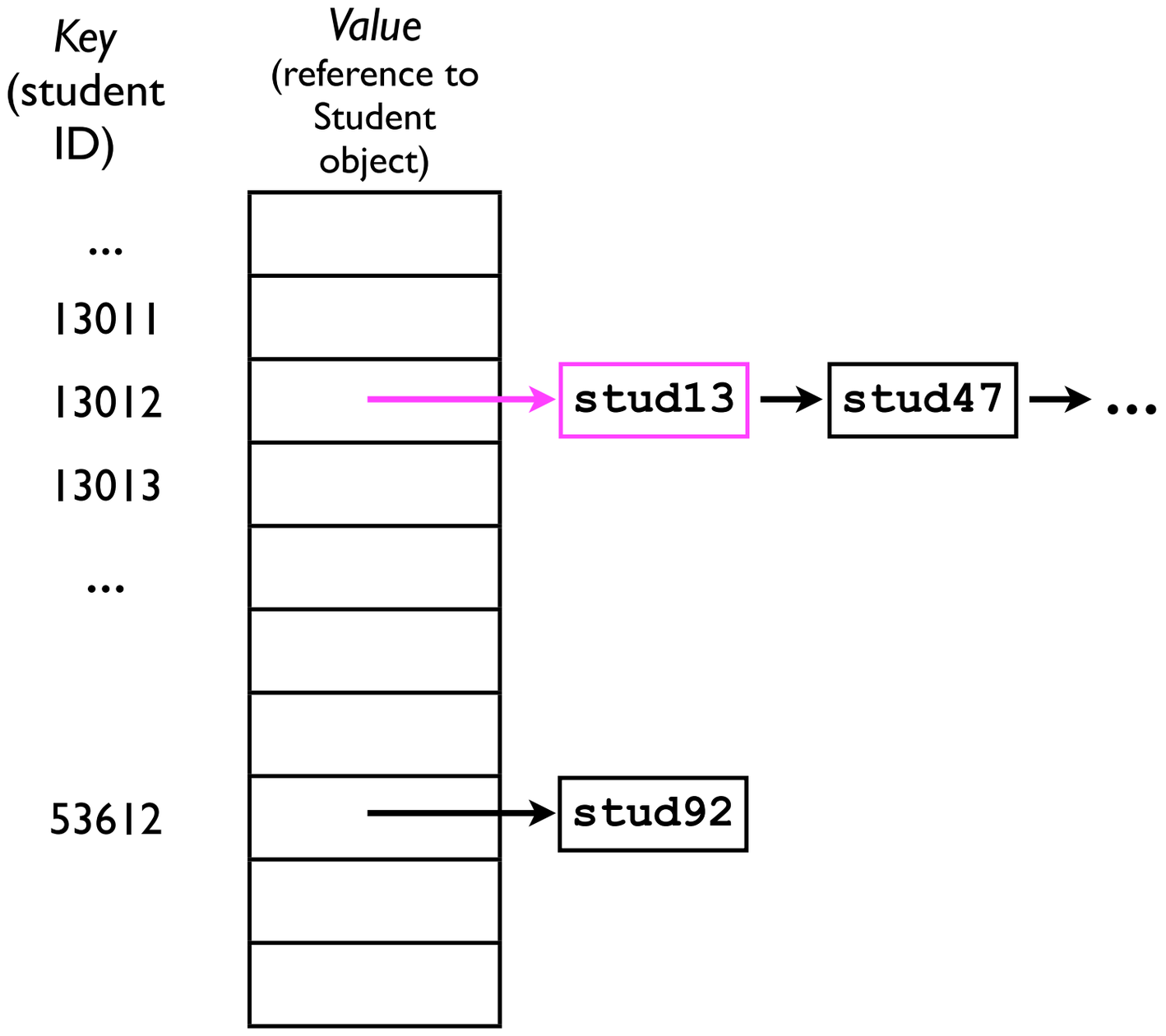
1. Hash `stud47`'s student ID to determine the *index*.
2. Jump to the head of the corresponding linked list.
3. Traverse the linked list until we find `stud47`.



Chaining

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

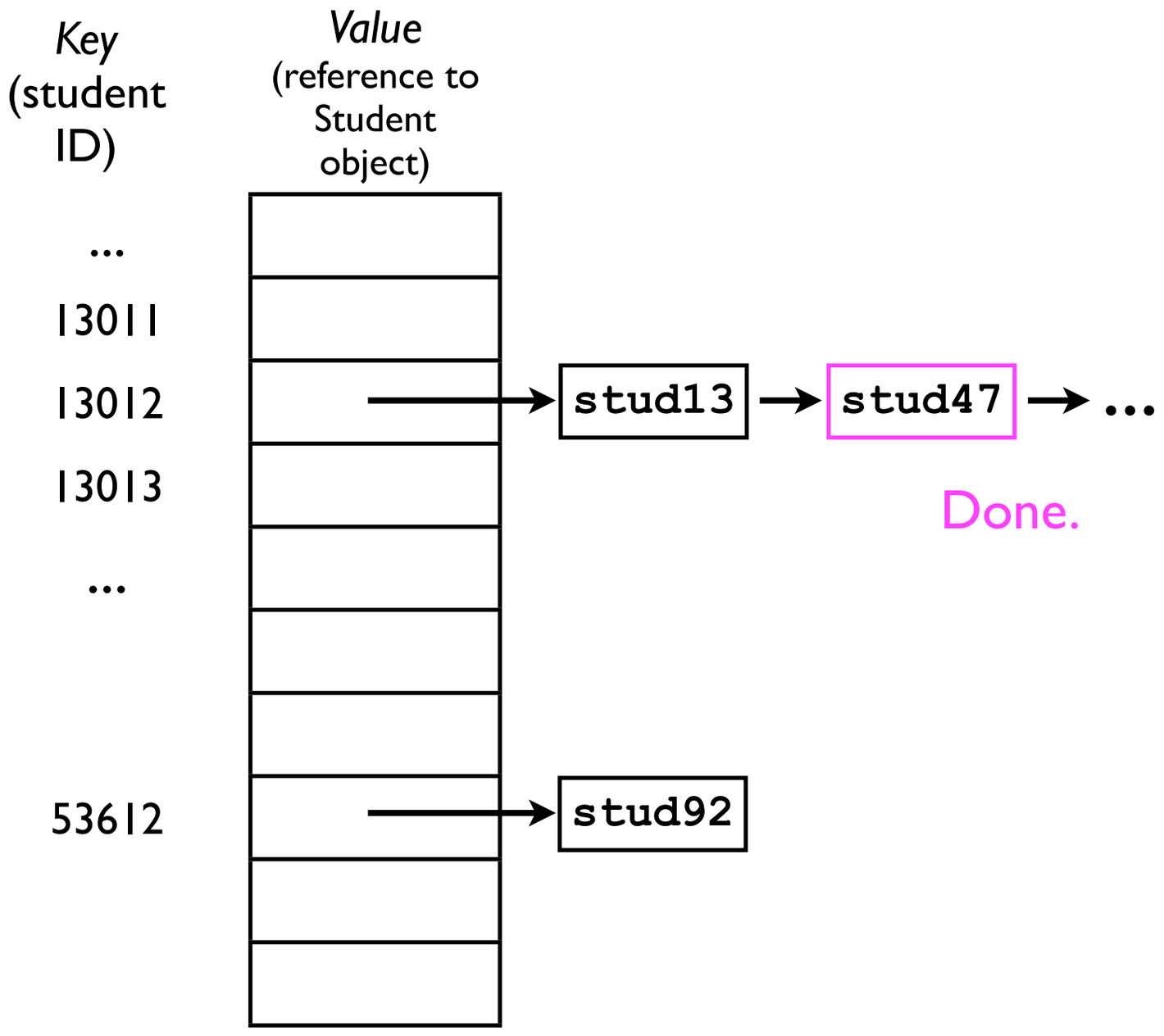
1. Hash `stud47`'s student ID to determine the *index*.
2. Jump to the head of the corresponding linked list.
3. Traverse the linked list until we find `stud47`.



Chaining

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

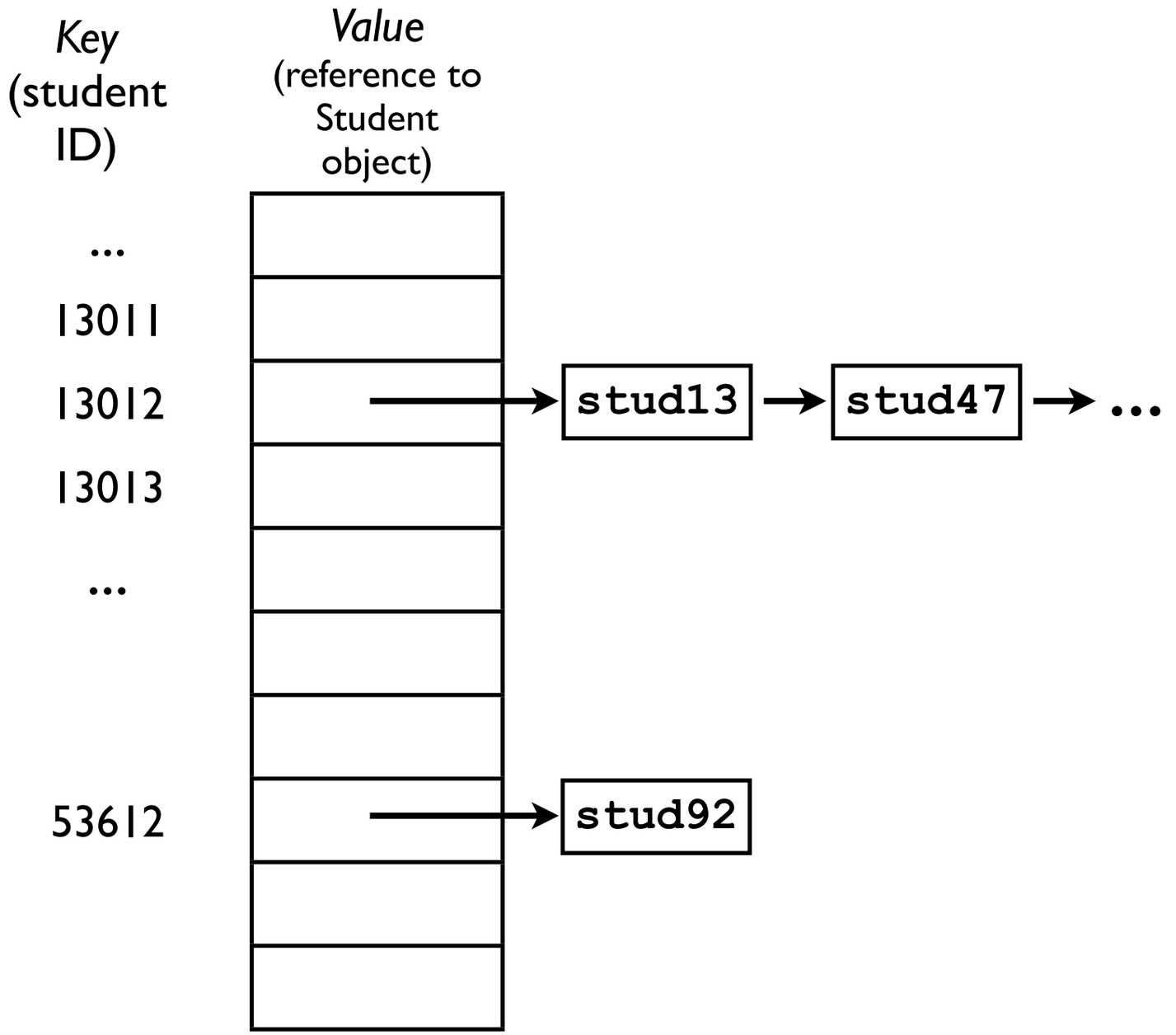
1. Hash `stud47`'s student ID to determine the *index*.
2. Jump to the head of the corresponding linked list.
3. Traverse the linked list until we find `stud47`.



Chaining

- If we reach the end of the linked list and don't find the desired student, then we know that student is *not contained in the hash table.*

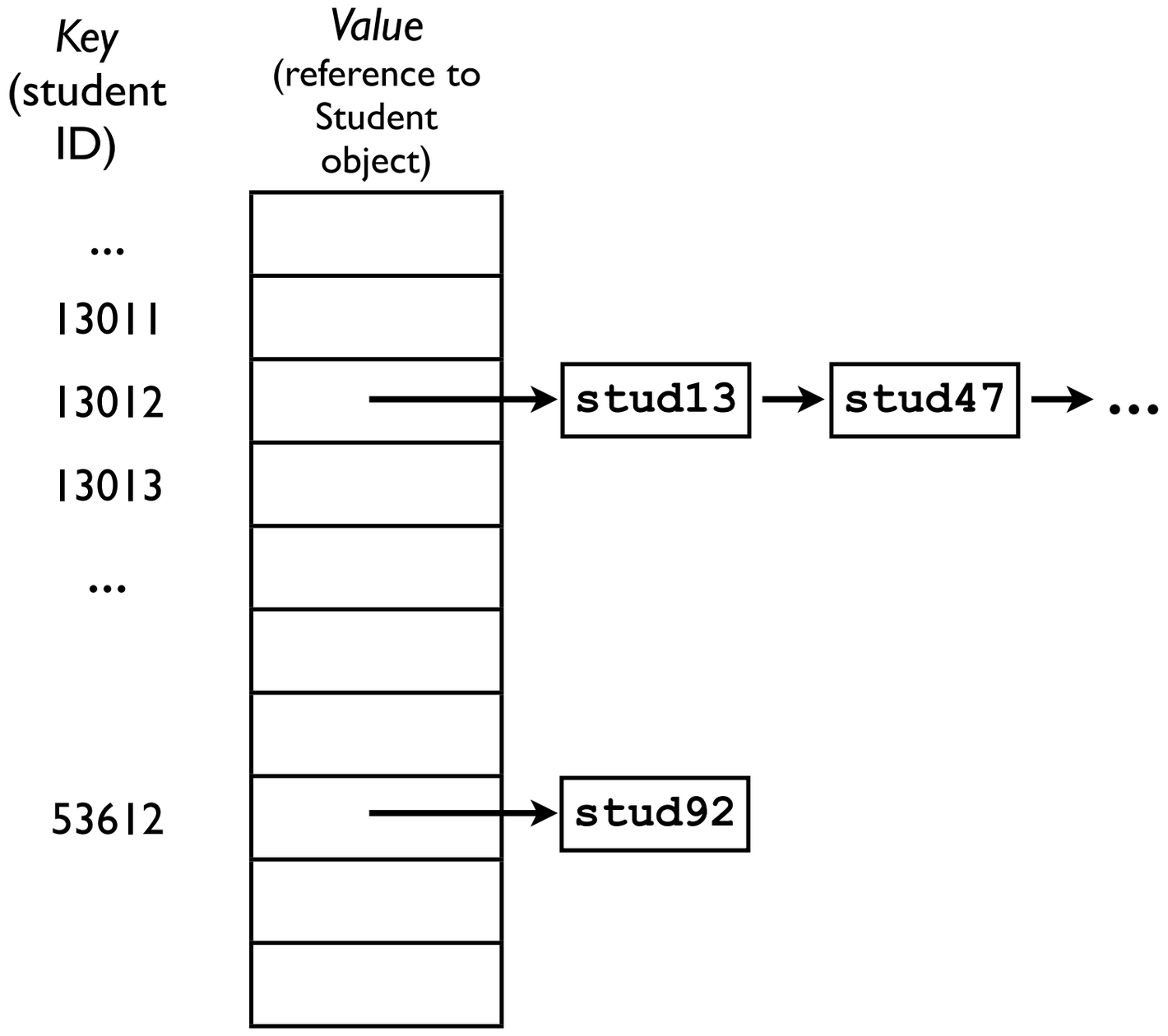
- There is *no other slot* at which the `Student` object would be stored.



Chaining

- If we wish to *remove* a student (e.g., `stud13`), we just:

1. Hash the student ID of the student we want to remove.
2. Find that student in the linked list.
3. "Splice it out".



Chaining: Performance analysis

- 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. *O*(1)
  - Jump to that array index. *O*(1)
  - Traverse through the linked list at that index. *O*(?)

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 and *N* actual objects stored, the average list length for any array slot is *N*/*M*.
  - Then, the average time to access any arbitrary object is *O*(1 + N/M).
  - Now, suppose that we always make sure that *M* > *N*.
    - Then *N*/*M* < 1.
    - Hence, average-case time cost is O(1+*N*/*M*)=O(1).

Open addressing

- An alternative to *chaining* is *open addressing*.
- With *open addressing*, there are no linked lists associated with array slots.
  - Instead, if a given slot is already "full", then the hash table "tries another one".
    - There are different strategies for "finding another one".
- Confusingly, open addressing is sometimes also known as *closed hashing*:
  - The "closed" refers to the fact that all data are stored within the array itself, not in a separate chain.

Open addressing

- There are different strategies for "finding another slot":
  - 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 *j*th probe (where *j* starts at 0) is given by the expression:

      (*i*+*j*) % M

Open addressing

- There are different strategies for "finding another slot":
  - 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 *j*th probe (where *j* starts at 0) is given by the expression:

      (*i*+*j*) % M

      Why not (*i* % M) + *j* ?

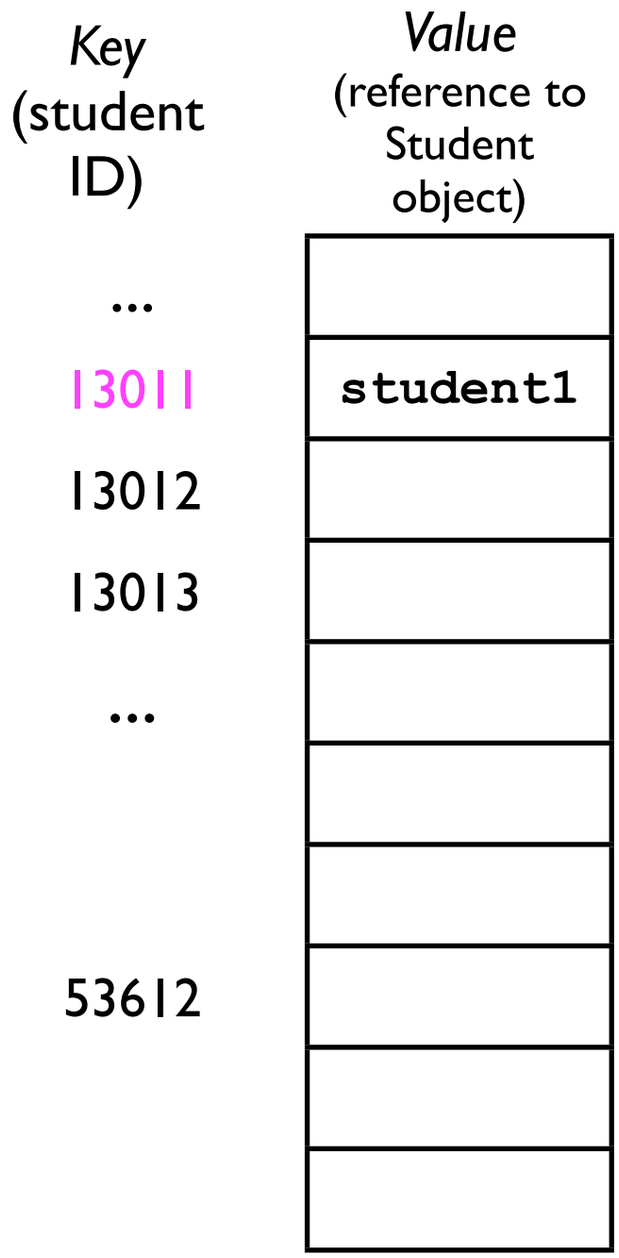
Open addressing

- Linear probing is simple to implement. However, it will tend to cause large amounts of *clustering* (more later).
- Better performance is achieved by **quadratic probing**:
  - If `hashFunction(key)` maps into an index *i* , then the index of the *j*th probe is given by the expression:

    $(i + c_0 * j + c_1 * j^2) \% M$

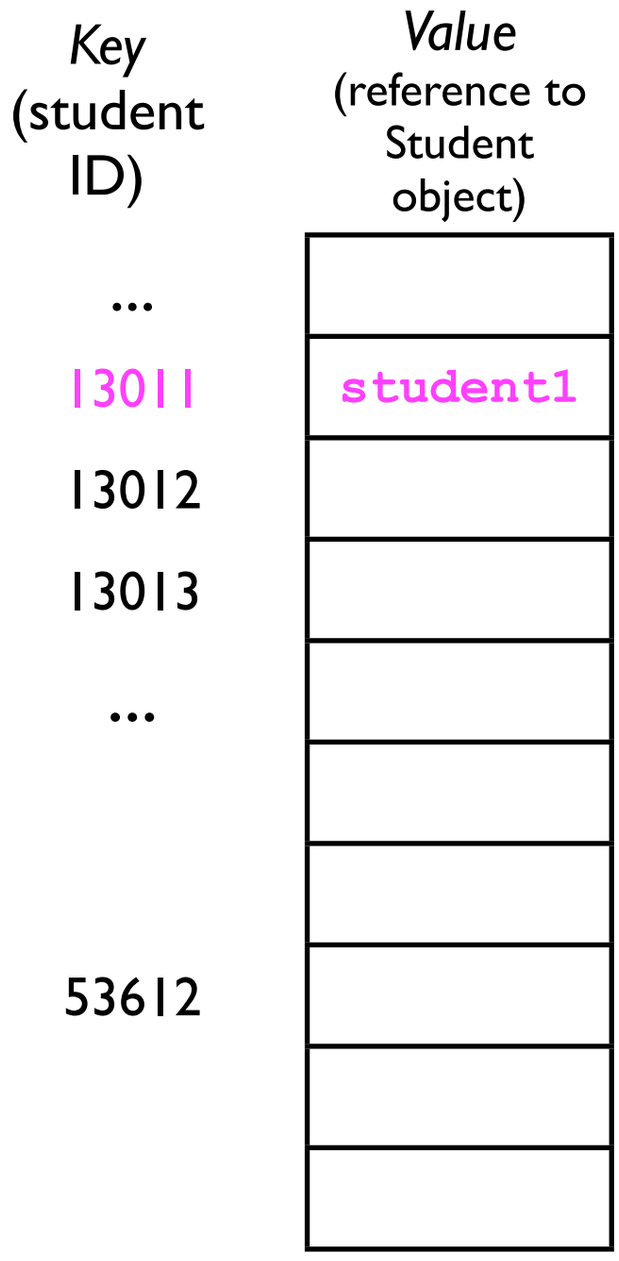
Linear probing

- Suppose our hash table contains M=100000 buckets, and we wish to add `student2` to the hash table, and `hashFunction(studentID2)` returns 13011.
- This slot is occupied by `student1`.
- The next slot we try is (13011+1) % 100000 = 13012.
- Since 13012 is available, we can insert `student2` at that slot.



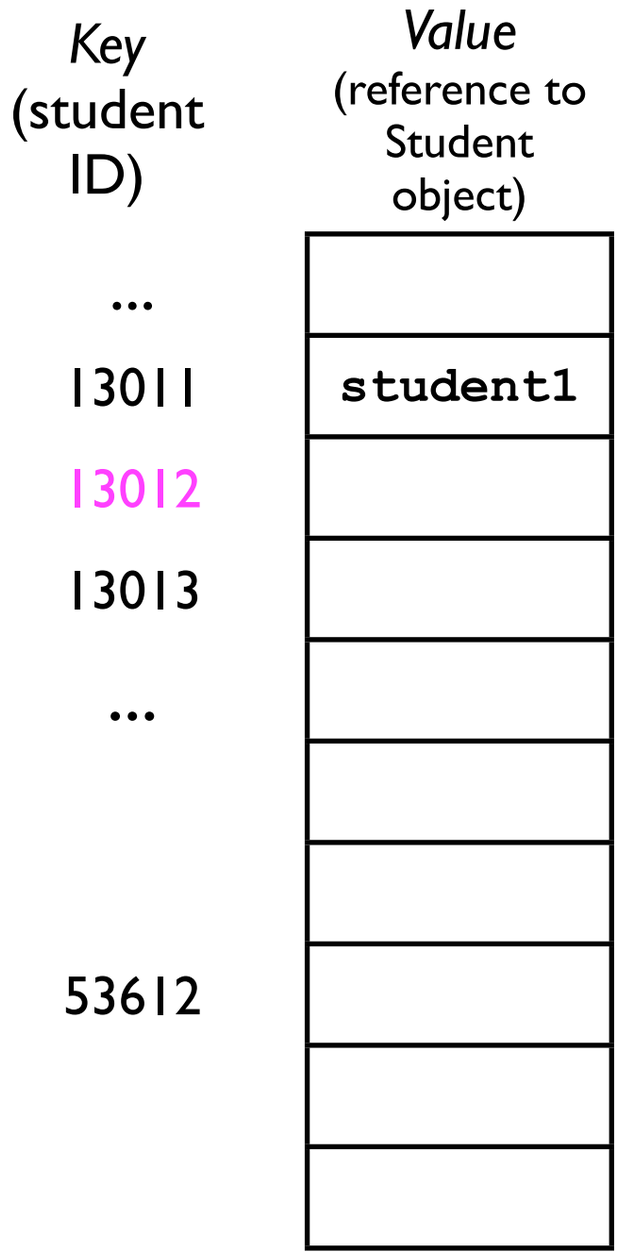
Linear probing

- Suppose our hash table contains M=100000 buckets, and we wish to add `student2` to the hash table, and **`hashFunction(studentID2)`** returns 13011.
- This slot is occupied by `student1`.
- The next slot we try is (13011+1) % 100000 = 13012.
- Since 13012 is available, we can insert `student2` at that slot.



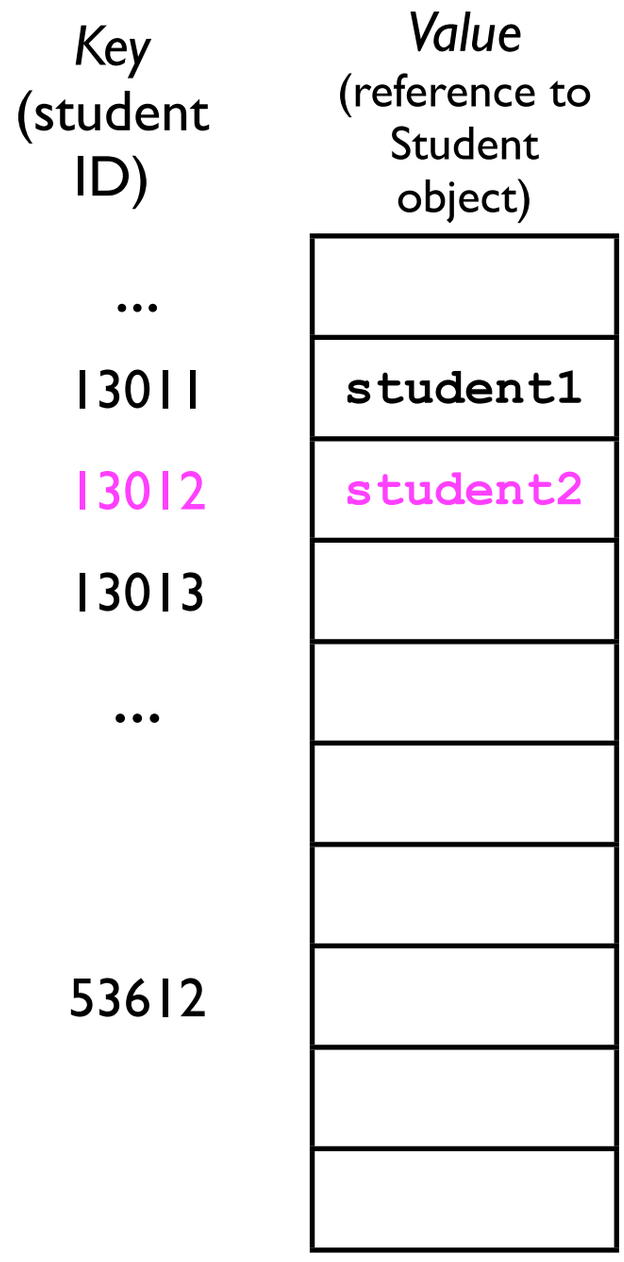
Linear probing

- Suppose our hash table contains M=100000 buckets, and we wish to add `student2` to the hash table, and **`hashFunction(studentID2)`** returns 13011.
- This slot is occupied by `student1`.
- The next slot we try is (13011+1) % 100000 = 13012.
- Since 13012 is available, we can insert `student2` at that slot.



Linear probing

- Suppose our hash table contains M=100000 buckets, and we wish to add `student2` to the hash table, and **`hashFunction(studentID2)`** returns 13011.
- This slot is occupied by `student1`.
- The next slot we try is (13011+1) % 100000 = 13012.
- Since 13012 is available, we can insert `student2` at that slot.



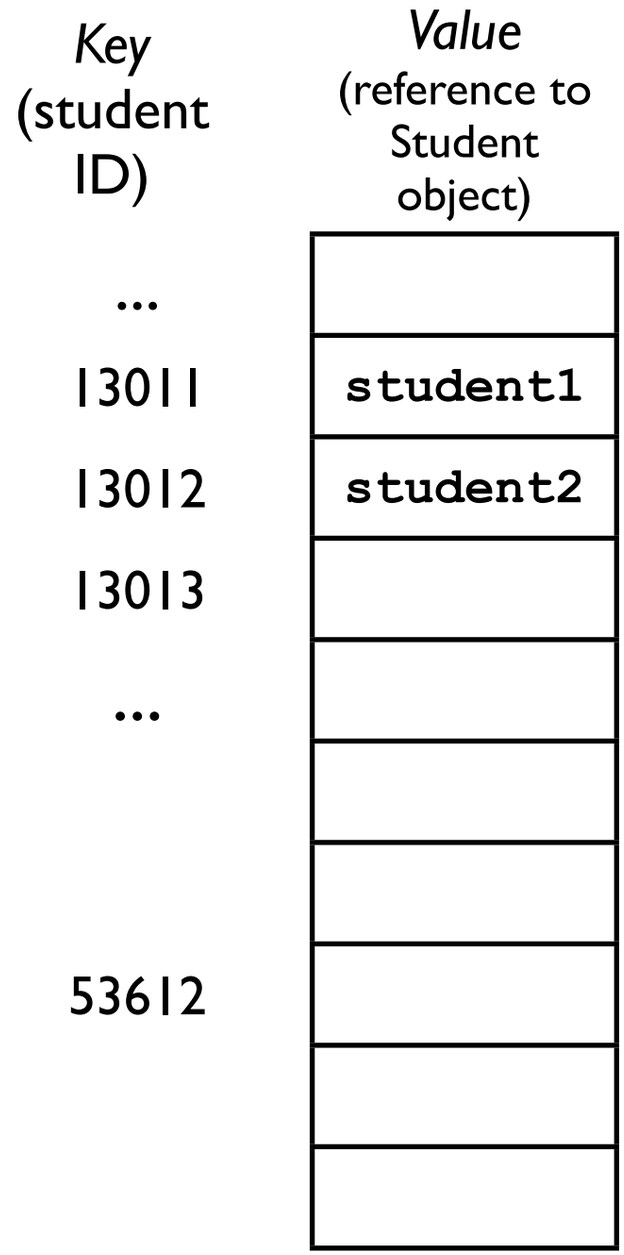
Linear probing

- When we later wish to find **`student2`**, we:

1. Compute its index using the hash function (*i*=13011).

2. Search down the array (using linear probing) until we find the correct object, *or* until we find an empty slot.

   - If we find an empty slot, then we know **`student2`** is *not* contained in the hash table.



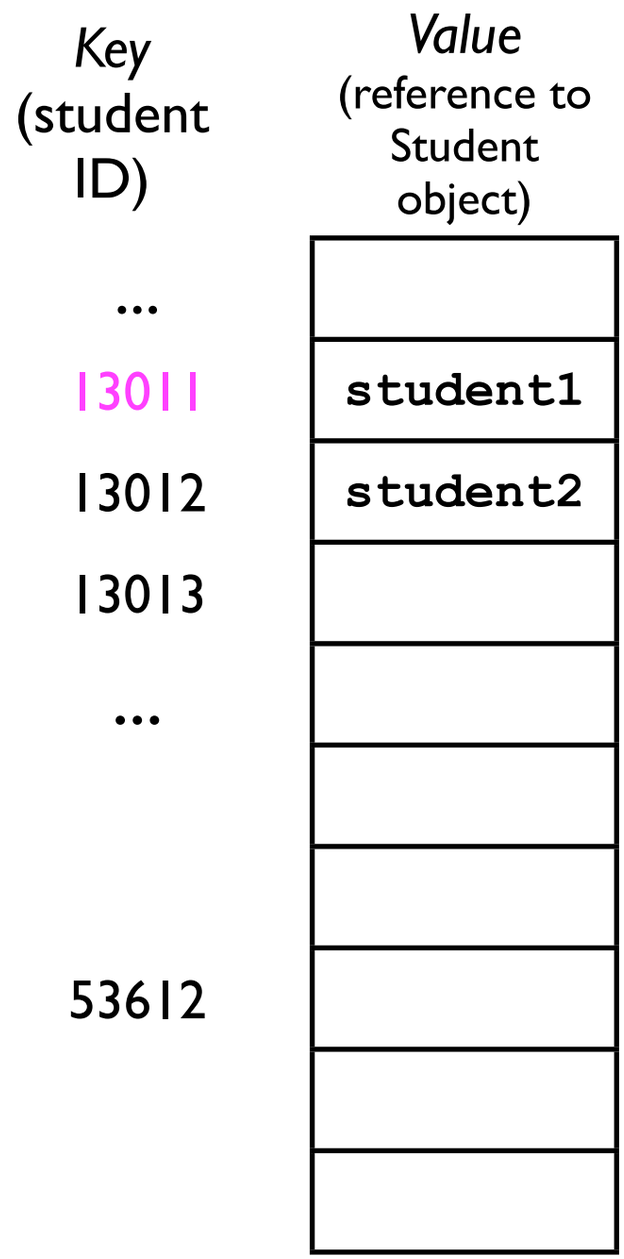
Linear probing

- When we later wish to find **`student2`**, we:

1. Compute its index using the hash function (*i*=13011).

2. Search down the array (using linear probing) until we find the correct object, *or* until we find an empty slot.

   - If we find an empty slot, then we know **`student2`** is *not* contained in the hash table.



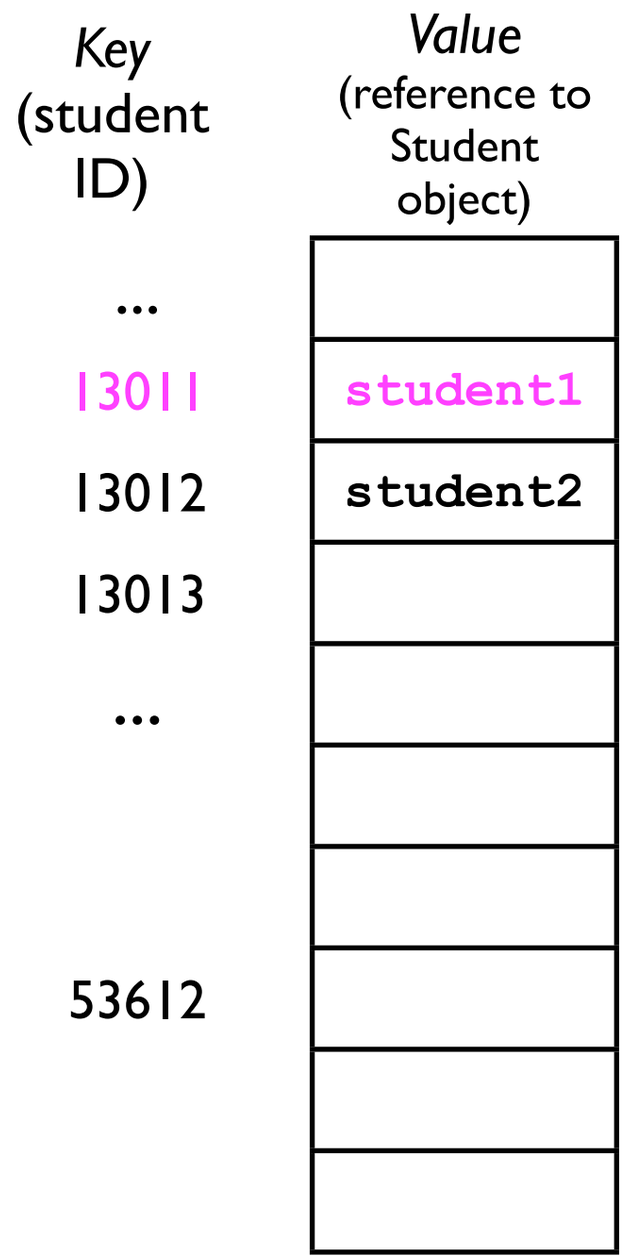
Linear probing

- When we later wish to find `student2`, we:

1. Compute its index using the hash function (*i*=13011).

2. Search down the array (using linear probing) until we find the correct object, *or* until we find an empty slot.

   - If we find an empty slot, then we know `student2` is *not* contained in the hash table.



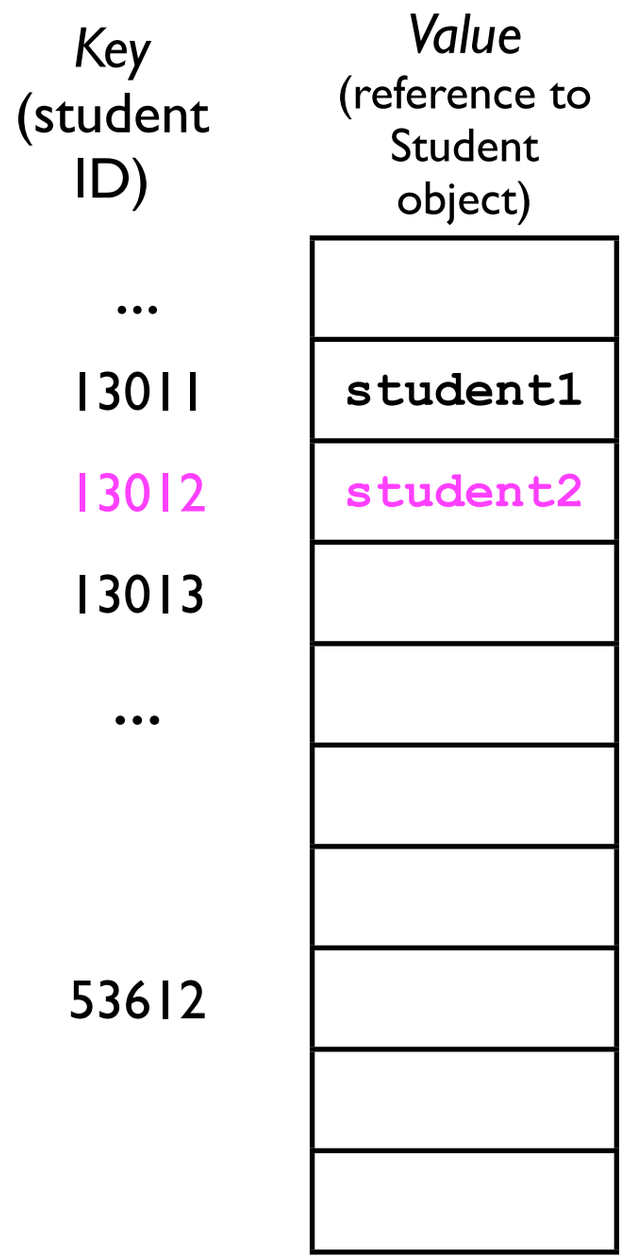
Linear probing

- When we later wish to find **`student2`**, we:

1. Compute its index using the hash function (*i*=13011).

2. Search down the array (using linear probing) until we find the correct object, *or* until we find an empty slot.

   - If we find an empty slot, then we know **`student2`** is *not* contained in the hash table.



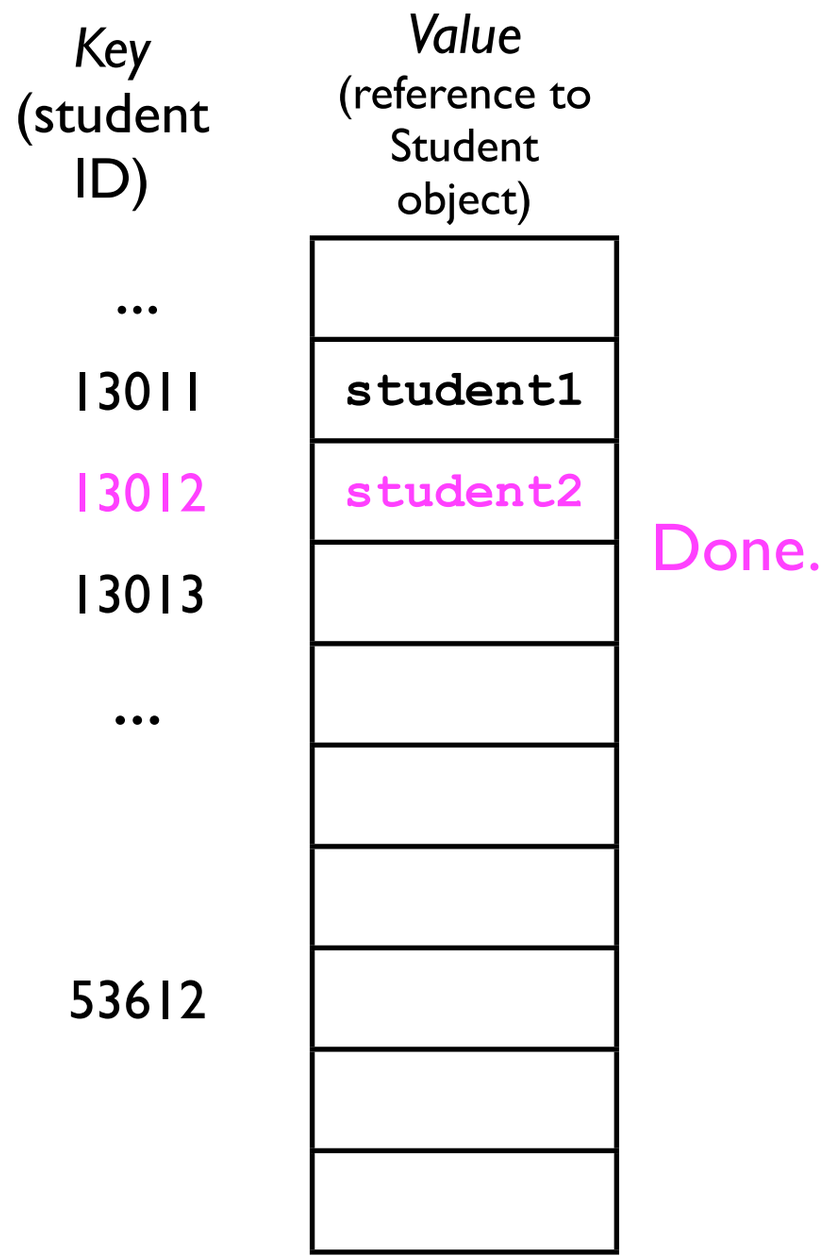
Linear probing

- When we later wish to find `student2`, we:

1. Compute its index using the hash function (*i*=13011).

2. Search down the array (using linear probing) until we find the correct object, *or* until we find an empty slot.

   - If we find an empty slot, then we know `student2` is *not* contained in the hash table.



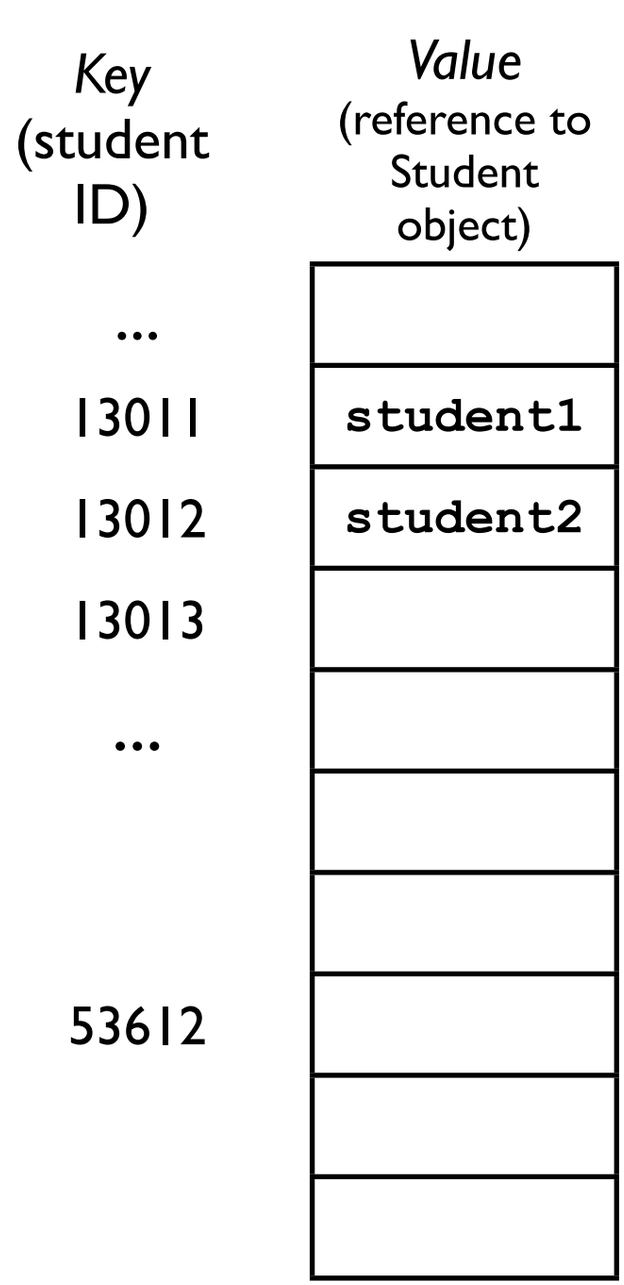
Linear probing

- Suppose we search for `student9`, whose hash code happens to also be 13011:

1. Compute its index using the hash function (*i*=13011).

2. Search down the array (using linear probing) until we find the correct object, *or* until we find an empty slot.

   - If we find an empty slot, then we know `student2` is *not* contained in the hash table.



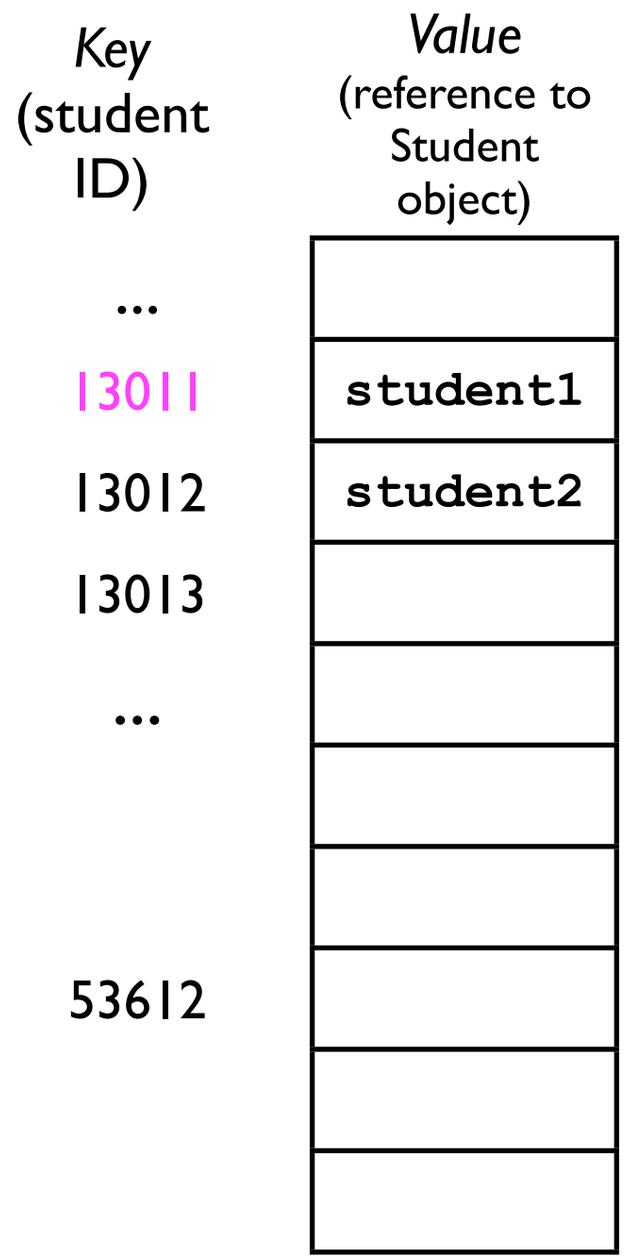
Linear probing

- Suppose we search for `student9`, whose hash code happens to also be 13011:

1. Compute its index using the hash function (*i*=13011).

2. Search down the array (using linear probing) until we find the correct object, *or* until we find an empty slot.

   - If we find an empty slot, then we know `student2` is *not* contained in the hash table.



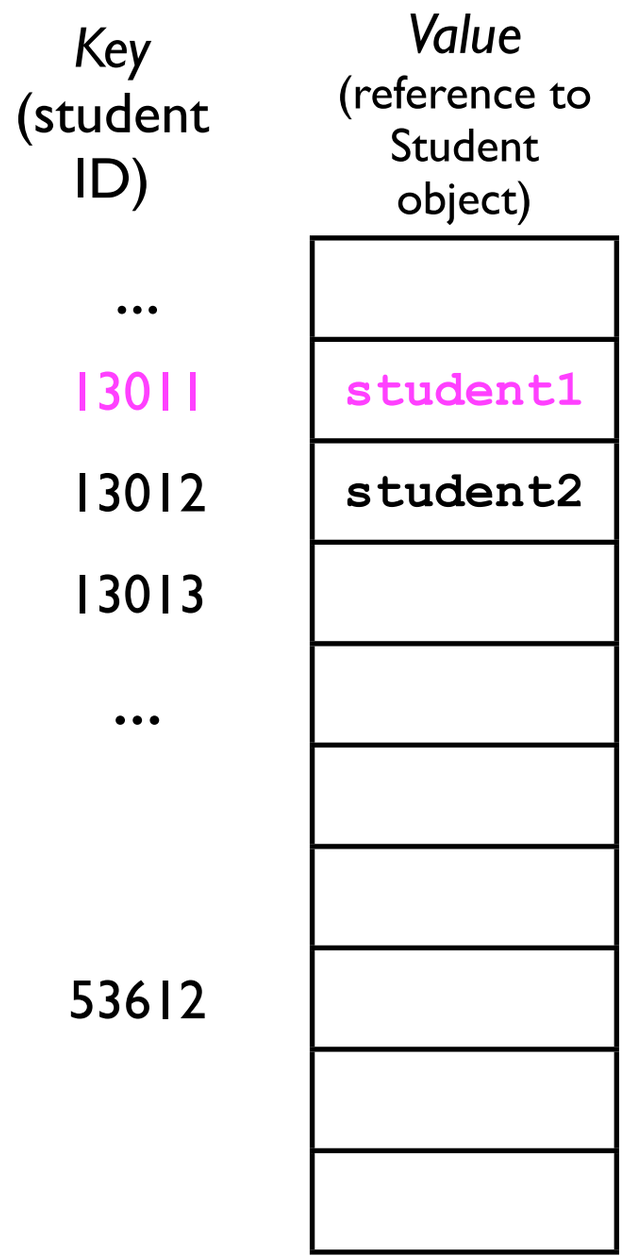
Linear probing

- Suppose we search for `student9`, whose hash code happens to also be 13011:

  1. Compute its index using the hash function (*i*=13011).

  2. Search down the array (using linear probing) until we find the correct object, *or* until we find an empty slot.

     - If we find an empty slot, then we know `student2` is *not* contained in the hash table.



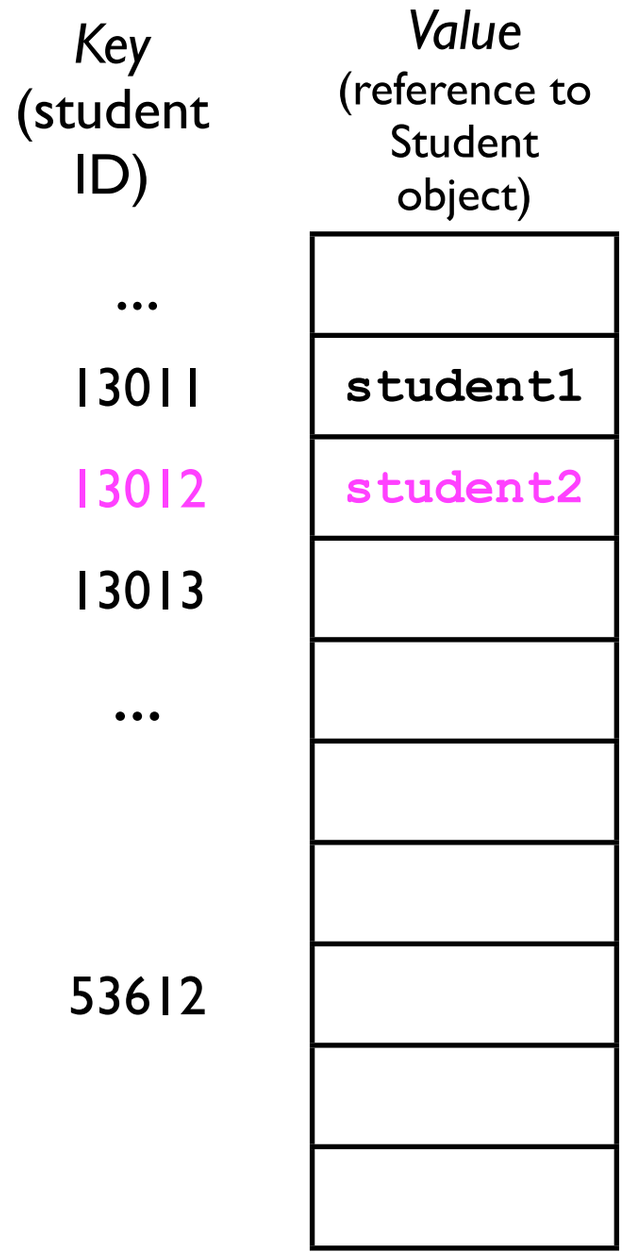
Linear probing

- Suppose we search for `student9`, whose hash code happens to also be 13011:

  1. Compute its index using the hash function (*i*=13011).

  2. Search down the array (using linear probing) until we find the correct object, *or* until we find an empty slot.

     - If we find an empty slot, then we know `student2` is *not* contained in the hash table.



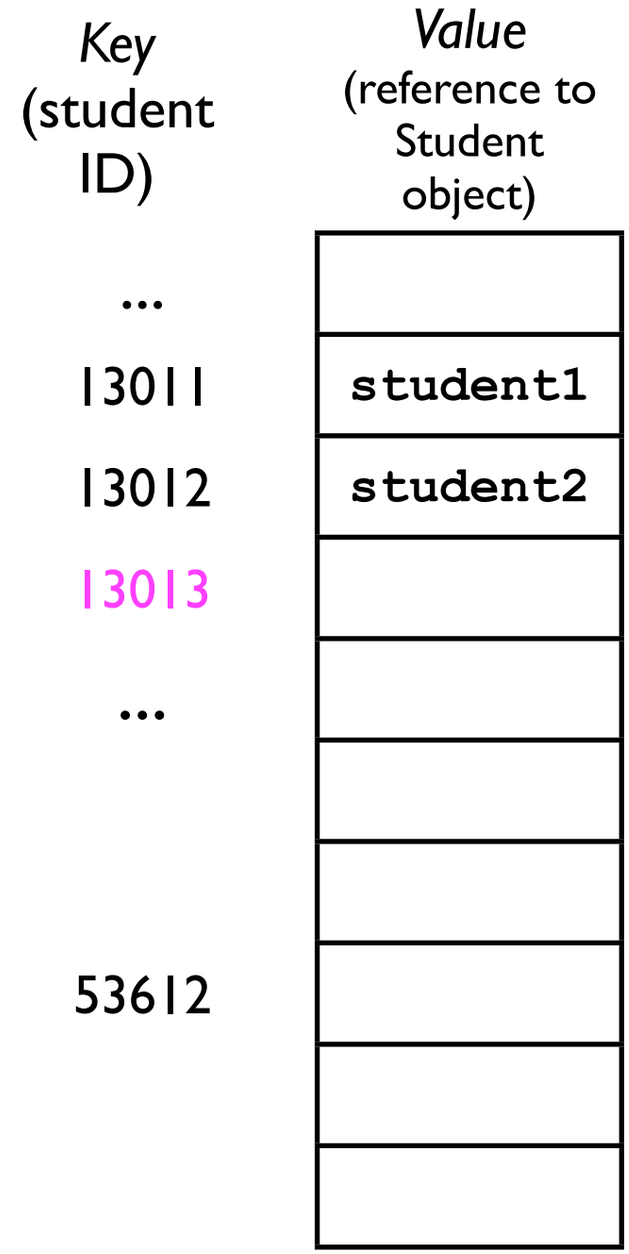
Linear probing

- Suppose we search for `student9`, whose hash code happens to also be 13011:

1. Compute its index using the hash function (*i*=13011).

2. Search down the array (using linear probing) until we find the correct object, *or* until we find an empty slot.

   - If we find an empty slot, then we know `student2` is *not* contained in the hash table.



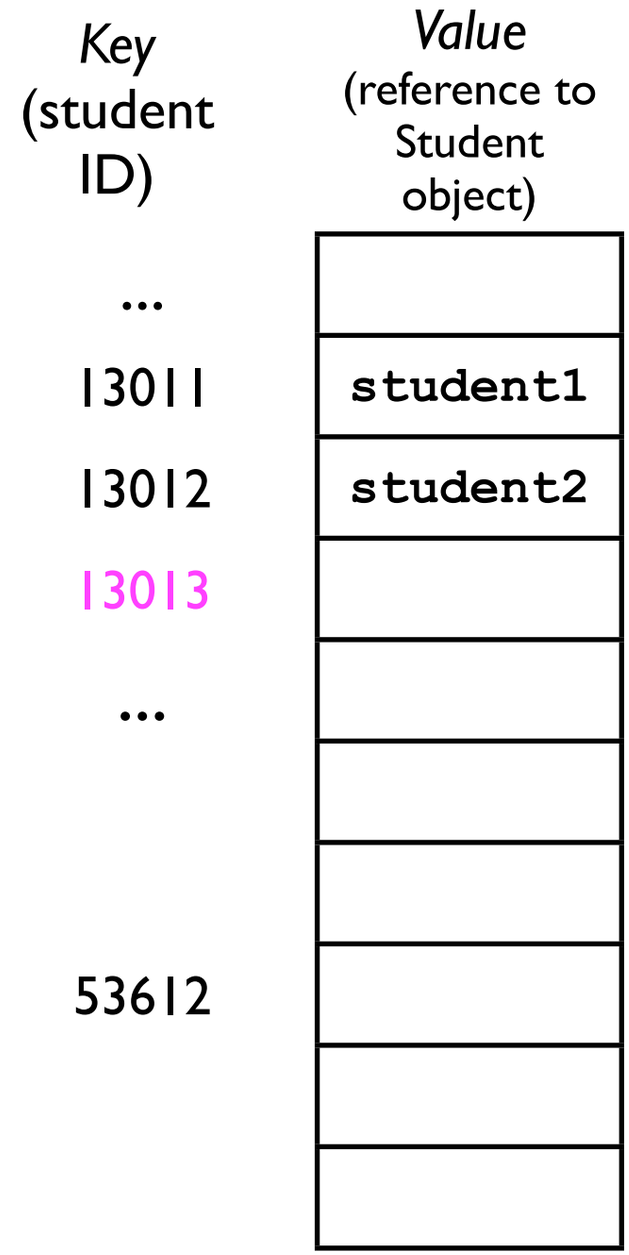
Linear probing

- Suppose we search for `student9`, whose hash code happens to also be 13011:

1. Compute its index using the hash function (*i*=13011).

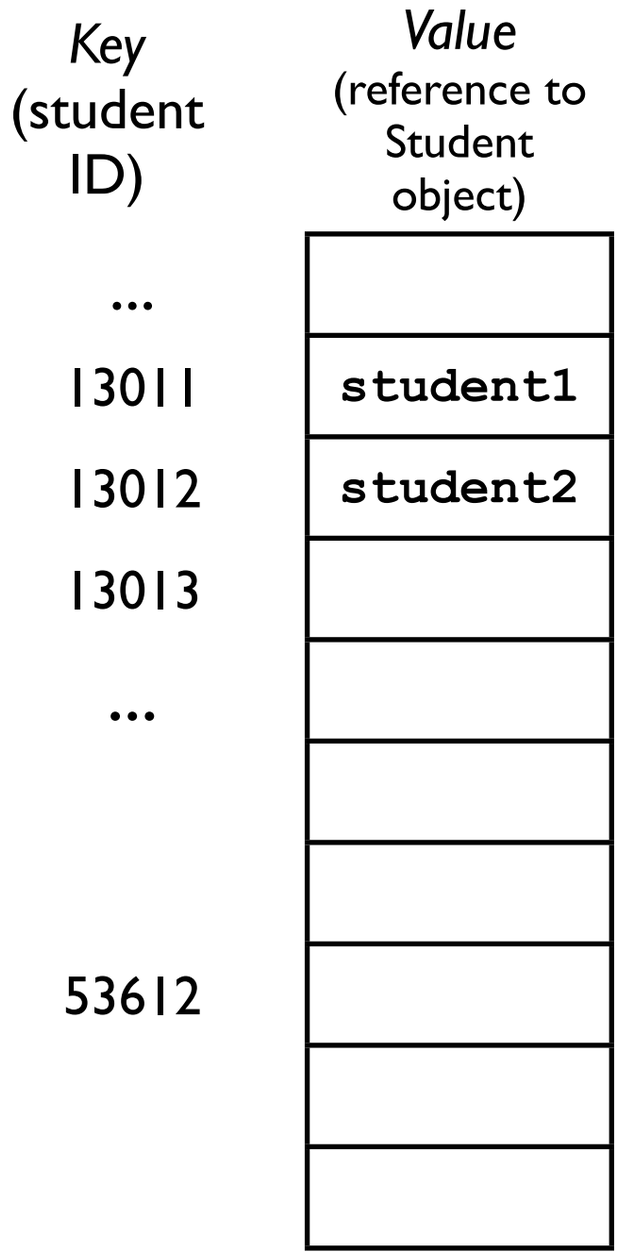
2. Search down the array (using linear probing) until we find the correct object, *or* until we find an empty slot.

   - If we find an empty slot, then we know `student2` is *not* contained in the hash table.



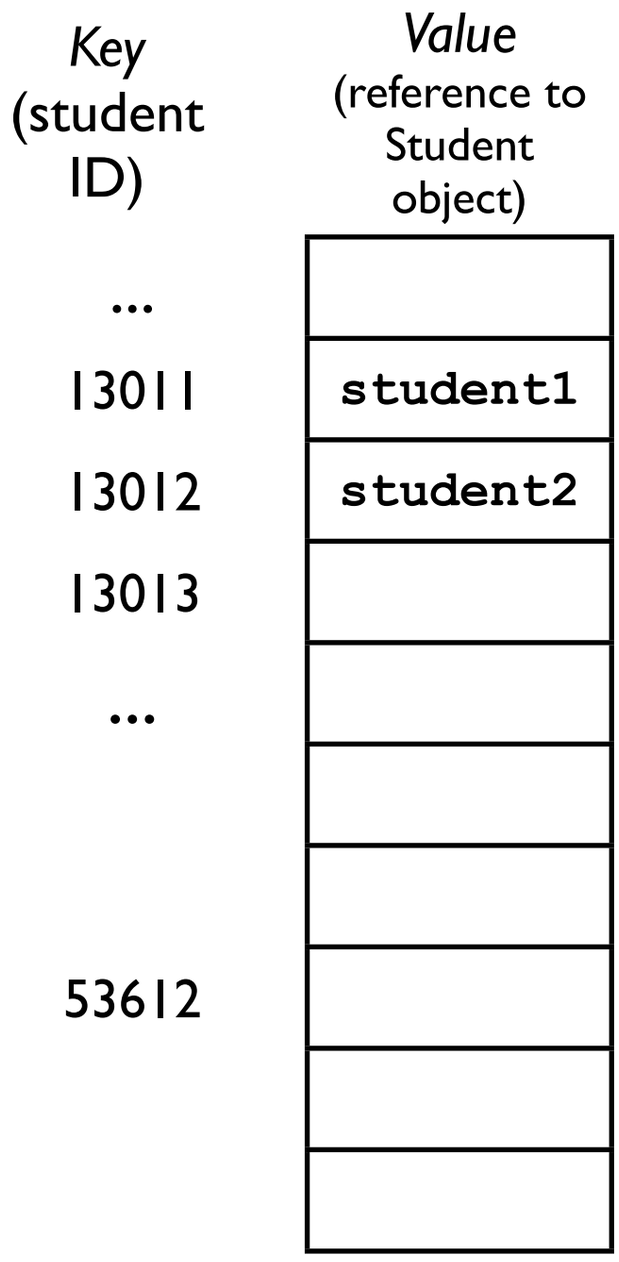
Open addressing

- Open addressing requires less memory than chaining because there are no linked lists.
- However, they suffer from a few complications:
  1. Removing an element.
  2. Clustering.



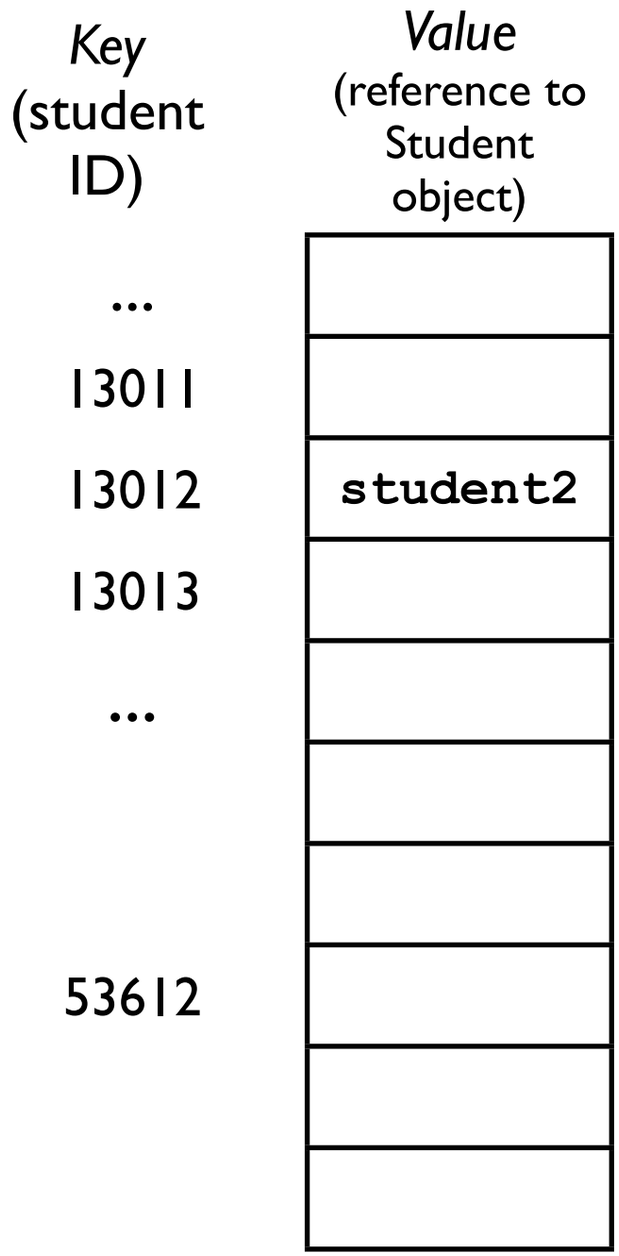
Removing an element

- Suppose we remove `student1` from the hash table.



Removing an element

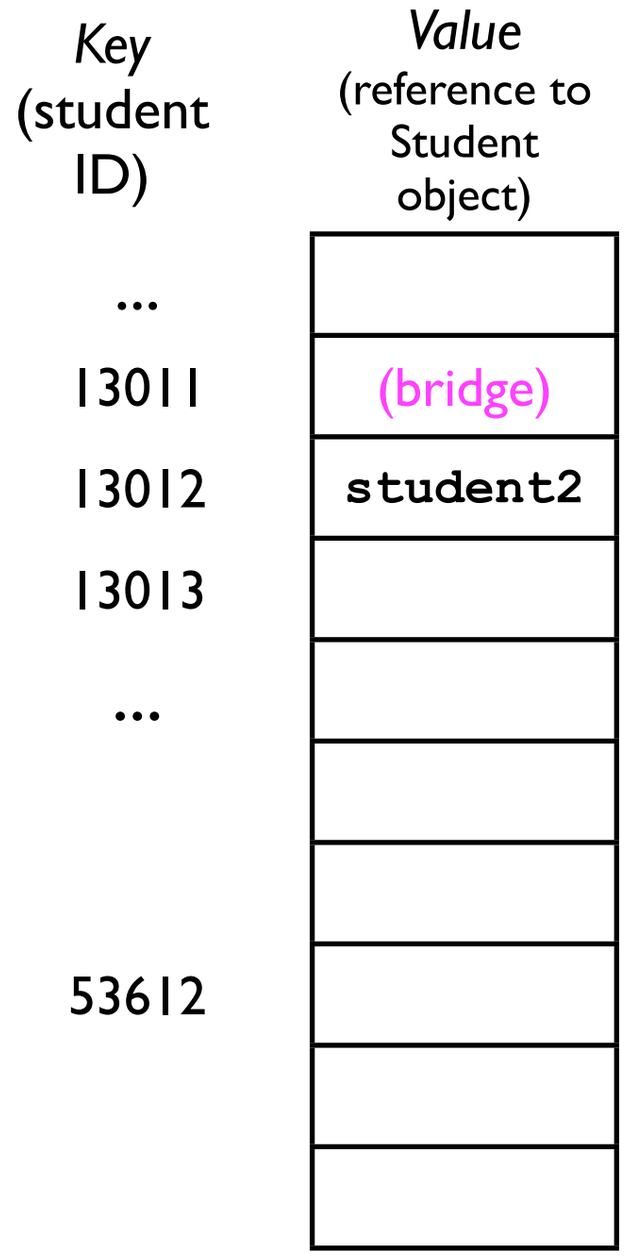
- Suppose we remove `student1` from the hash table.
- If we later search for `student2`, we will still hash to 13011, but find that it is *empty*.
- Does that mean `student2` is not contained in the hash table?
- No -- but we have to *record* that somehow.



Removing an element

- One method of recording that an element was deleted is a *bridge*, a special element that indicates "empty, but keep looking."

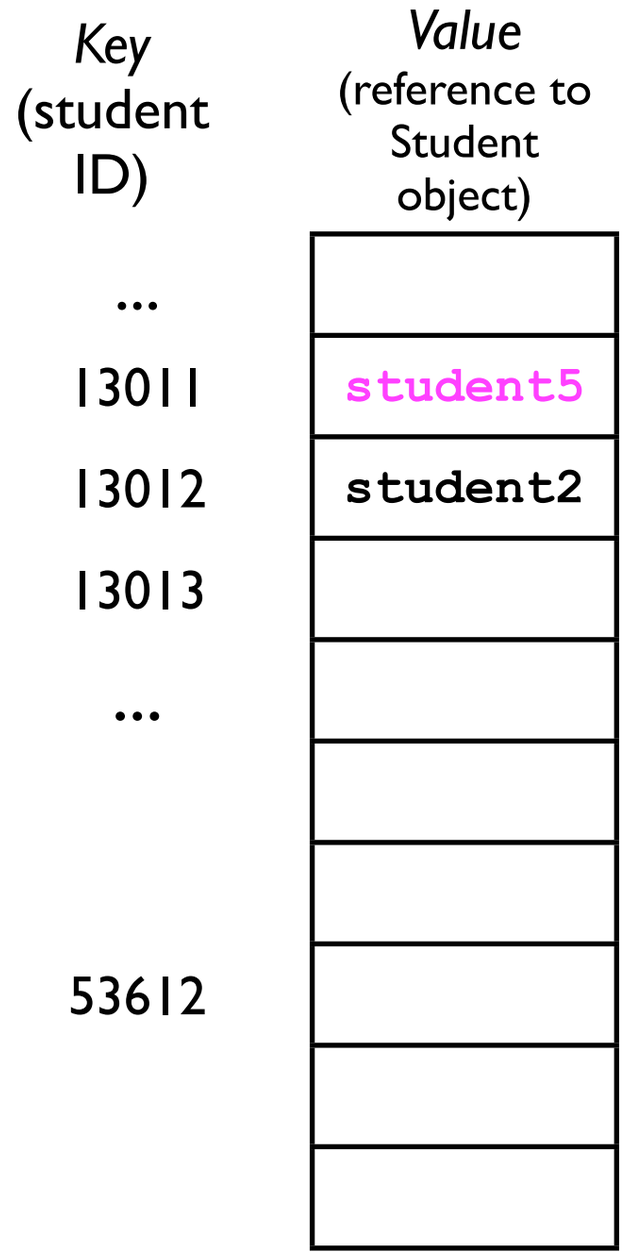
- If we later add another element, say `student5` that hashes to 13011, then we can *replace* the bridge with a real `Student` object.



Removing an element

- 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.



Clustering

- The other downside of open addressing is **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	
...	

Clustering

- The other downside of open addressing is **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	`student7`
...	

Open addressing

- Like chaining, *open addressing* guarantees O(1) time cost for the add/find/remove operations *so long as* $M > N$.
- Due to the bridging complication, open addressing is most useful when elements will *never be deleted*.
- The memory saved by not using linked lists can be used to allocate more slots ==> higher *M*.
  - The larger *M* is, the fewer collisions will occur, and the better the average-case performance will be.

Hash table ADTs

- So far we've focused more on how a hash table is implemented *internally* and less how a user would *use* it.
- There are two different *interfaces* that a hash table ADT might offer.
- The interface varies depending on whether:
  1. Key is a field *inside* the whole record.
  2. Key is *separate* and stored *outside* the record.

Key inside the record

- In some previous code examples we've conceptualized the *key* as a *field* within the whole object, e.g.:

```
class Student {
  int _studentID;
  String _firstName, _lastName;
  boolean _hasTeddyBear;
}
```

- This implementation of *keys* then lends itself to the following hash table interface:

```
interface HashTable<T extends HasKey> {
  void add (T o);
  T get (T o);
}
```

where the hypothetical `HasKey` interface guarantees that `T` offers a method called `int getKey()`.

Key inside the record

- The **`HashTable`** then might then be implemented as:

```
class HashTableImpl<T extends HasKey>
implements HashTable<T> {

  T[] _array;
  ...
  void add (T o) {
    _array[hashFunction(o.getKey())] = o;
  }

  T get (T o) {
    return _array[hashFunction(o.getKey())];
  }
}
```

Key inside the record

- The user could then use the hash table as follows:

```
class Student implements HasKey {
  int _studentID;
  String _firstName, _lastName;
  boolean _hasTeddyBear;
  ...
  int getKey () {
    return _studentID;
  }
}

hashTable<Student> students = new HashTable<Student>();

students.add(new Student(12345, “Jacky”, “O’Nassis”));
students.add(new Student(9231, “Bette”, “Midler”));

...
Student someStudent = students.get(new Student(9231));
```

Key inside the record

- It turns out this hypothetical **`HasKey`** interface is unnecessary because the Java **`Object`** class already offers a **`hashCode()`** method.
- **`hashCode()`** should return some "integer representation" of the object.
- The default implementation of **`Object.hashCode()`** is simply to return the *address* in memory (an **`int`**) of the object.
- Subclasses of **`Object`** can override **`hashCode()`** to do something more meaningful.

hashCode()

- For example, we could override **hashCode()** in **Student** to return the **Student**'s **_studentId** field:

HasKey interface no longer necessary

```
class Student implements HasKey {
  int _studentID;
  String _firstName, _lastName;
  boolean _hasTeddyBear;
  ...
  public int hashCode () {
    return _studentID;
  }
}
```

`hashCode()`

- Because **`hashCode()`** is available in *every Java* **`Object`**, we can simplify both the interface and implementation of the **`HashTable<T>`**:

`HasKey` interface no longer necessary

```
interface HashTable<T extends HasKey> {
  void add (T o);
  T get (T o);
}

class HashTableImpl<T> implements HashTable<T> {
  ...
  void add (T o) {
    _array[hashFunction(o.hashCode())] = o;
  }

  T get (T o) {
    return _array[hashFunction(o.hashCode())];
  }
}
```

Non-integer keys

- `hashCode()` also provides useful functionality for supporting *non-integer keys*.
  - E.g., we want to use a `Student`'s full name as the key.
    - A names is a `String`, which is not an `int`.
    - How do we convert from a `String` into an (`int`) index into the hash table?
      - Just delegate to `String.hashCode()`.

Non-integer keys

- Example:

```
class Student {
  int _studentID;
  String _firstName, _lastName;
  boolean _hasTeddyBear;
  ...
  public int hashCode () {
    String fullName = _firstName + " " + _lastName;
    return fullName.hashCode();
  }
}
```

Since `fullName` is a `String`, and `String` is an `Object`, then `fullName` is guaranteed to support the `hashCode()` method.

Hash code examples

- Suppose our key is:
  - A single character `c`:
    - We could convert `c` into its ASCII value, which is an integer (from 0-127).
  - A `String s` of characters:
    - We could convert each `c` in `s` to its ASCII value, and then add them together.
  - An image `im`:
    - We could add together the pixel values across all three (R,G,B) channels.

Note: these are just hypothetical examples, not necessarily how Java actually implements hash codes!

`hashCode()`

- In Java, the `hashCode()` method *must* uphold two properties:

  1. *Deterministic* -- multiple subsequent calls to `hashCode()` on the *same object* o must return the same value.

- Otherwise, `hashFunction(key.hashCode())` would map into a different array index -- and the hash table wouldn't be able to find o.

```
_array[hashFunction(o.hashCode())] = o;    // Add
…
return _array[hashFunction(o.hashCode()]; // Find
```

`hashCode()`

2. *Consistent across equal instances* -- if `o1.equals(o2)`, then `o1.hashCode()` *must* equal `o2.hashCode()`:

```
String s1 = "hello";
String s2 = new String("hello");  // Distinct copy
int hashCode1 = s1.hashCode();
int hashCode2 = s2.hashCode();  // Must equal hashCode1
```

- This means that if class `A` overrides the `equals()` method, then it must also override `hashCode()`.

`hashCode()`

- In addition, it is *desirable* for `hashCode()` to have:

  3. *Wide distribution across instances* -- `hashCode()` should return *different* values for *different* instances of the same class as much as possible.

     - If `A.hashCode()` returned the *same* hash value for *every* instance `o`, then *all* objects of type `A` would map into the same array index. `hashCode()` is always the same.

```
_array[hashFunction(key1.hashCode())] = o1;
_array[hashFunction(key2.hashCode())] = o2;  // Collision
_array[hashFunction(key3.hashCode())] = o3;  // Collision
_array[hashFunction(key4.hashCode())] = o4;  // Collision
```

     - This would yield terrible ($O(n)$) hash performance!

Key outside the record

- More commonly, however, hash tables *separate* the *key* from the *value*.
- A typical hash table interface might be:

```
interface HashTable<K,V> {
  void put (K key, V value);
  V get (K key);
}
```

Here, we are defining *two different* type parameters K (for keys) and V (for values).

Key outside the record

- This may be more convenient than the **`HashTable`** interface where the key is inside the record.
- Compare:
  - Key inside record:

```
hashTable.add(new Student(123, “Jimmy”, “Carter”));
...
Student student = hashTable.get(new Student(123));
```

  - Key separate from record:

```
hashTable.put(123, new Student(“Jimmy”, “Carter”));
...
Student student = hashTable.get(123);
```

No need to instantiate new **`Student`** object just to find an existing one.

Key outside the record

- Separating keys from values is especially useful when we use a hash table as a *dictionary*.
- A **dictionary** is a data structure for storing a set of associations between keys and values.

Key outside the record

- Example:
  - We can create a dictionary of English words to their meanings:

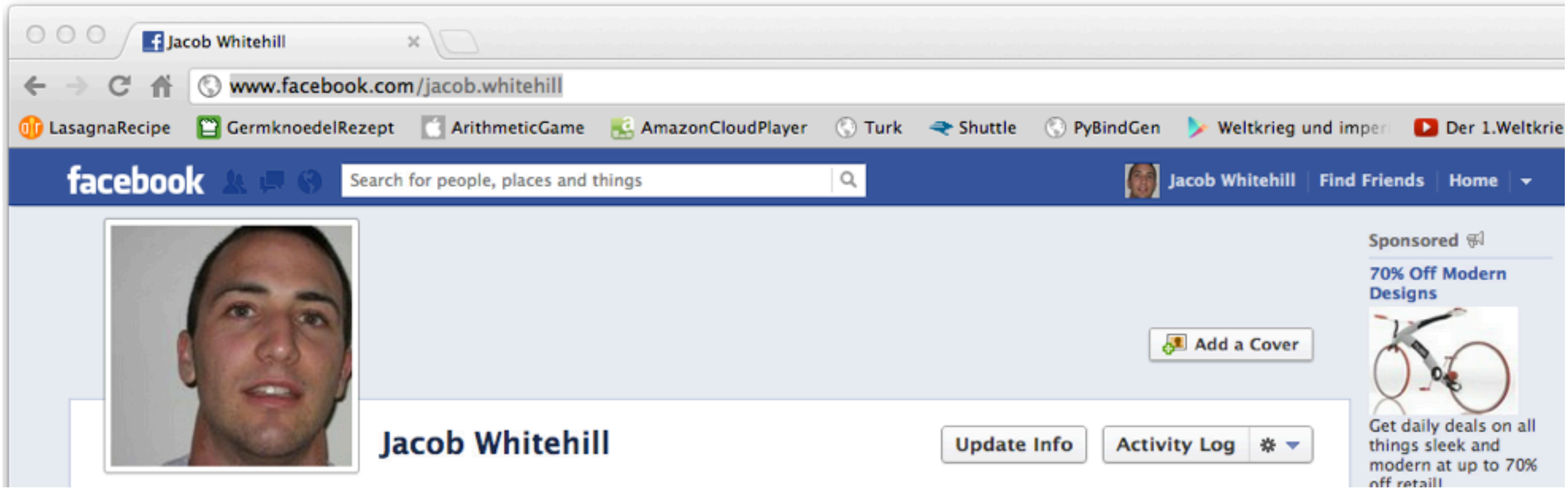
```
HashTable<String,String> englishDictionary =
  new HashTable<String,String>();
englishDictionary.put(
  “eggplant”,
  “The somewhat large egg-shaped fruit of a
   tropical Old World plant, eaten as a vegetable.”
);

...

String meaning = englishDictionary.get(“eggplant”);
```

Key outside the record

- To implement a **`HashTable`** in which keys are stored separately from values, we must "bind" the key and value together inside the table:

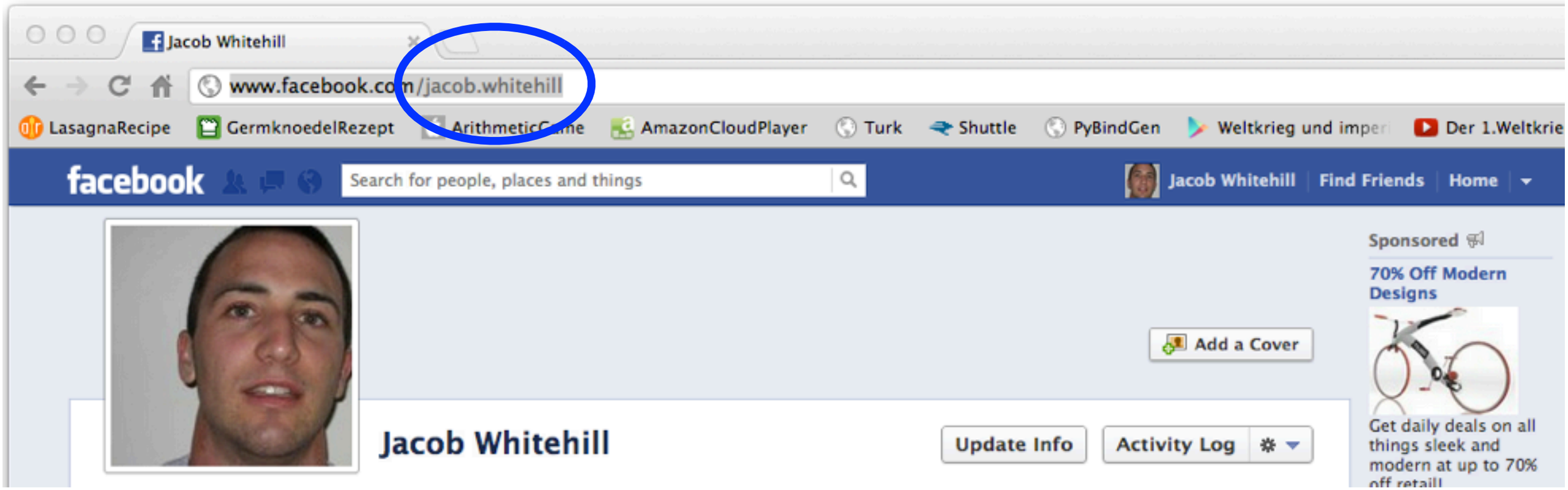
```
class HashTableImpl<K,V> implements HashTable<K,V> {
  static class Bucket {
    K _key;
    V _value;
    ...
  }
  Bucket[] _array;
  ...
  void put (K key, V value) {
    int bucketIdx = hashFunction(key.hashCode());
    _array[bucketIdx]._key = new Bucket(key, value);
  }

  V get (Key key) {
    return _array[hashFunction(key.hashCode())]._value;
  }
}
```

Hash table usage example



- Suppose we were implementing the Web server for Facebook.
  - Server must be robust to handle **billions** of requests per day!

Hash table usage example



- Facebook access is initiated when user enters the username of a particular user.
  - Given the username, the Facebook page, including pictures and text, must be found **fast**.

Hash table usage example

- Possible implementation:

```
Image findUserProfilePicture (String username) {
  String pathToImageFile = _usersToImagePaths.get(username);
  return new Image(pathToImageFile);
}
```

This needs to be super-fast to handle Facebook's typical load; hence, use a **hash table**.