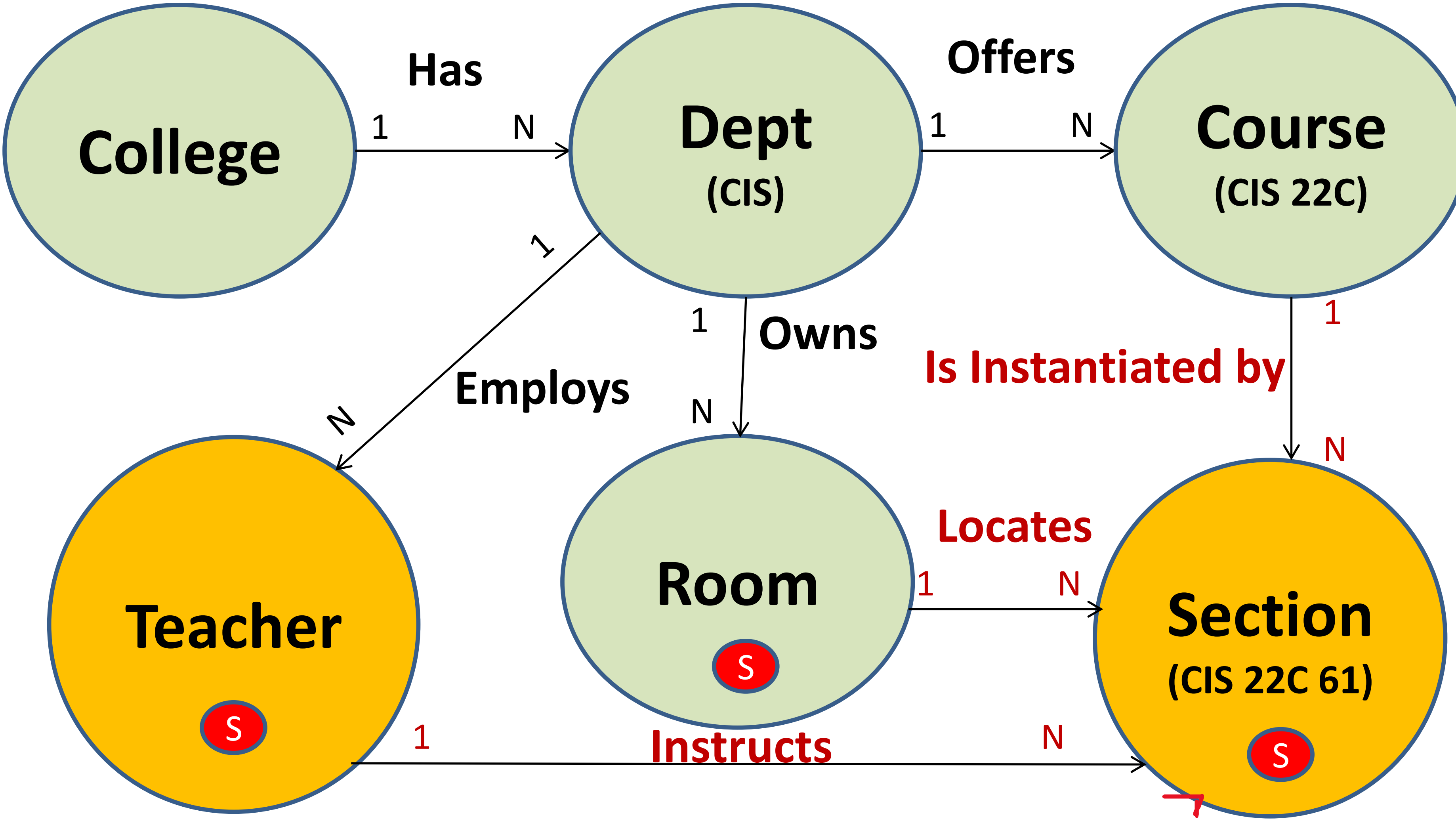
Associative Arrays

Key

Value

Conceptual Class Diagram:

Schedule of Classes

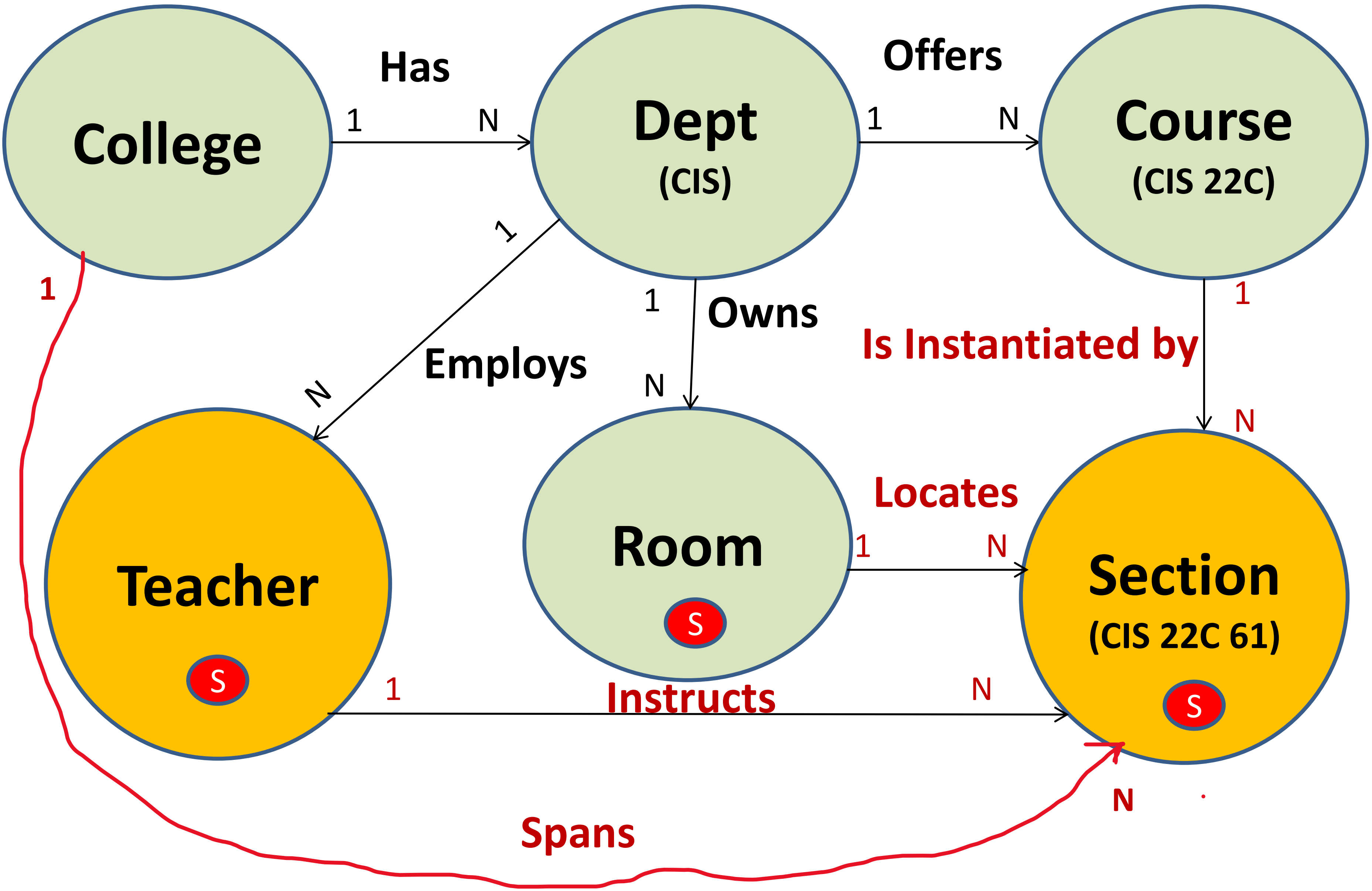


Converting Section ID to Section Ptr

- **College gets unique Section ID “CIS 23C 01Y”**
  – College selects Dept. “CIS” from its Dept. collection
  – College asks CIS Dept object for Course “23C 01Y”
    - CIS Dept selects Course 23C from its Course collection
    - CIS Dept asks Course 23C for Section 01Y
      – CIS 23C Course selects Section 01Y from its Sect. collection
      – **CIS 23 Course returns Ptr to Section “CIS 23C 01Y”**
- **College gets unique Section ID “22317” (CRN #)**
  – College selects “22317” from its Section collection
  – **Section Collection returns Ptr to Section “CIS 23C 01Y”**

Conceptual Class Diagram:

Schedule of Classes



Ordered Array vs. Associative Array

Ordered Array

Index	Value
0	Object Ptr
1	Object Ptr
2	Object Ptr
3	Object Ptr
	Object Ptr
N	Object Ptr

Element Key is Integer index.

Associative Array

Index	Value
Key	Object Ptr
Key	Object Ptr
Key	Object Ptr
Key	Object Ptr
Key	Object Ptr
Key	Object Ptr

Element Key is the ASCII String ID of the object.

Ex: “Section CRN” of “22327” → CIS22C 01Y, (Section Object assigned that CRN)

Associative Array: Key/Value Table

Index	Value
Key	Value
Key	Value
Key	Value
Key	Value
Key	Value
Key	Value

class ValueTable // Ex: Key = Student ID or Section CRN
{
  public:
    bool **add** (key, value); // Fails if matches another key
    value& **get** (key); // Returns matching object (or Null)
    bool **remove** (key);   // Deletes Key/Object entry if exists
    unsigned int **getSize**(); // Returns # key/value pairs in Table

Which of our Data Structures (Orderable Array, Linked List) might effectively be wrapped to support the Table Data Abstraction, **and why?**

Key / Value Table

Index	Value
Key	Value
Key	Value
Key	Value
Key	Value
Key	Value
Key	Value

Table value retrieval is based solely upon the supplied Key. There are no “*getNext()*” or “*getPrevious()*” public methods available.

Problems mapping to Ordered Array / Link List Data Structures because Table doesn’t require “ordering”, and ordering is what those collections do.

So why not Ordered Array?

UID To Thing Array

	Key	Value
0	UID	Thing Ptr
1	UID	Thing Ptr
2	UID	Thing Ptr
3	UID	Thing Ptr
N	UID	Thing Ptr

```
class ThingTable  // Ex: Key = Student ID or Section CRN
{
    public:
      bool add (uid, Thing*);  // Fails if matches another UID
    Thing& get (uid);        // Returns matching object (or Null)
     bool remove (uid);     //  Deletes  Key/Object entry if exists
    unsigned int getSize(); // Returns # key/value pairs in Table
```

[Each element is structure of **Key** (UID String) / **Value** (Thing Ptr)]

UID entered → binary search for entry with UID (log 2 of N opps). If found, Thing Ptr is retrieved.

Problem: As # of Table elements ↑ (ex: >20,000 Students attending a College) <u>in an environment with lots of inserts / removes</u>, maintaining Key ordering becomes increasingly untenable. **Why?**

Collection Fitting

The critical OA problem: **size of the collection.**

If: Large OA ➔ multiple smaller OAs

Then: Insert / Remove Update times would be manageable.

But that involves knowing which OA to select when trying to retrieve an element based on its Key.

How could that be done?

	Key	Value
0	UID	Thing Ptr
1	UID	Thing Ptr
2	UID	Thing Ptr
3	UID	Thing Ptr
4	UID	Thing Ptr

	Key	Value
0	UID	Thing Ptr
1	UID	Thing Ptr
2	UID	Thing Ptr
3	UID	Thing Ptr
4	UID	Thing Ptr
5	UID	Thing Ptr
6	UID	Thing Ptr

	Key	Value
0	UID	Thing Ptr
1	UID	Thing Ptr
2	UID	Thing Ptr

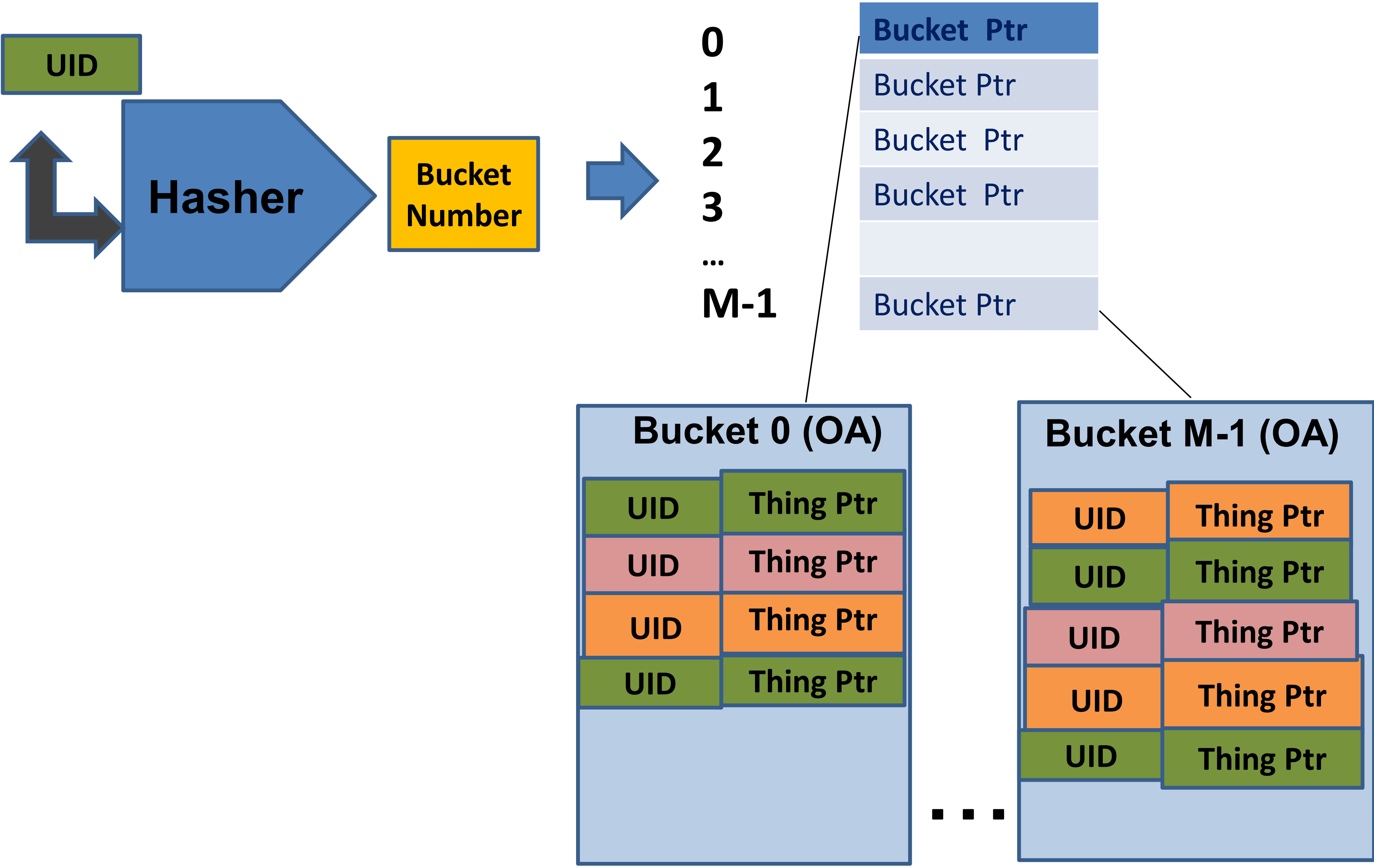
…

Hash Table: External Interface

	Key	Value
0	UID	Thing Ptr
1	UID	Thing Ptr
2	UID	Thing Ptr
3	UID	Thing Ptr
…		
N	UID	Thing Ptr

```
class ThingTable  // Ex: Key = Student ID or Section CRN
{
   public:
     bool insert (uid, Thing*);  // Fails if matches another key
    Thing& get (uid);        // Returns matching object (or Null)
    bool remove (uid);    //  Deletes  Key/Object entry if exists
   unsigned int getSize(); // Returns # key/value pairs in Table
```

Hash Table: Internal Implementation



How might "Buckets" be Implemented?

- **Each Bucket is an unordered collection** of all Elements with Keys that "hashed" to its number.
- **A Bucket Element for a "Thing" might look like:**

struct ThingElement { // Key / value pair (could be templated)

    String tid;   // **Key** (UID of Thing)

    Thing* pthing;}; // **Value** (Thing Ptr)

- The number of Buckets (M) is assigned at Hash Table creation and **normally NEVER changes**.
- The number of Elements within each Bucket **can change dramatically** as the collection evolves.

➔ *A Hash Table has a preset array of unordered OA Collection pointers. Each OA collects the set of Key/Value Elements whose keys hashed to the EA's index in the preset array.*

The Table "Hasher"
("grind up" every UID to a Bucket #)

The Object UIDs

The Bucket #'s

Requirements of any Hasher Algorithm

- **UID → Bucket # conversion:**
  - Must be fast
  - Same UID must ALWAYS produce same Bucket # **Why?**
  - Must work on strings (Ex: Student Name) as well as integers and floats. **How can this best be done?**
  - Element distribution across Buckets must be "somewhat" uniform **Why? What is "somewhat"?**

Requirements of any Hasher Algorithm

- Same UID must ALWAYS produce same Bucket # **Why?**
  - Object Ptr stored in Bucket its UID hashes to
  - Object Ptr retrieved based on Bucket its UID hashes to
- Must work on strings (Ex: Student Name) as well as integers and floats. **How can this best be done?**
  - Convert all UIDs to a String, and have the Hasher operate on that.
- Element distribution across Buckets must be “somewhat” uniform **Why? What is “somewhat”?**
  - If all UIDs map to only a few buckets, not much has been achieved

➔ Hasher algorithm must generate a <u>flat</u> element distribution among buckets

Tuning: Customizing the Hasher Algorithm

Assuming a max of N students, there is a tradeoff between # Buckets (**M**) and the expected # Elements in each Bucket (**P**).

– **M** up ➔ **P** down (trade storage for speed) … and vis-versa

In our example, assume 20,000 Students. If we chose to have 2000 Buckets, we can expect each Bucket to contain between 10 (great) and 30 (marginally acceptable) Student entries.

➔ **Note: It is a LOT quicker to search 30 entries than 20,000**

Tuning: Getting the "Perfect" Hash

- **"Perfect" Hash**
  - **P** = **N/M** for each of **M** buckets (no Bucket clustering)

  ➔ **"tailoring" Hasher to data being hashed**

In our example, assume 20,000 Students. If we chose to have 2000 Buckets, with a perfect hash, each Bucket will contain precisely 10 Student entries.

- **"Perfect" Hash when M == N**

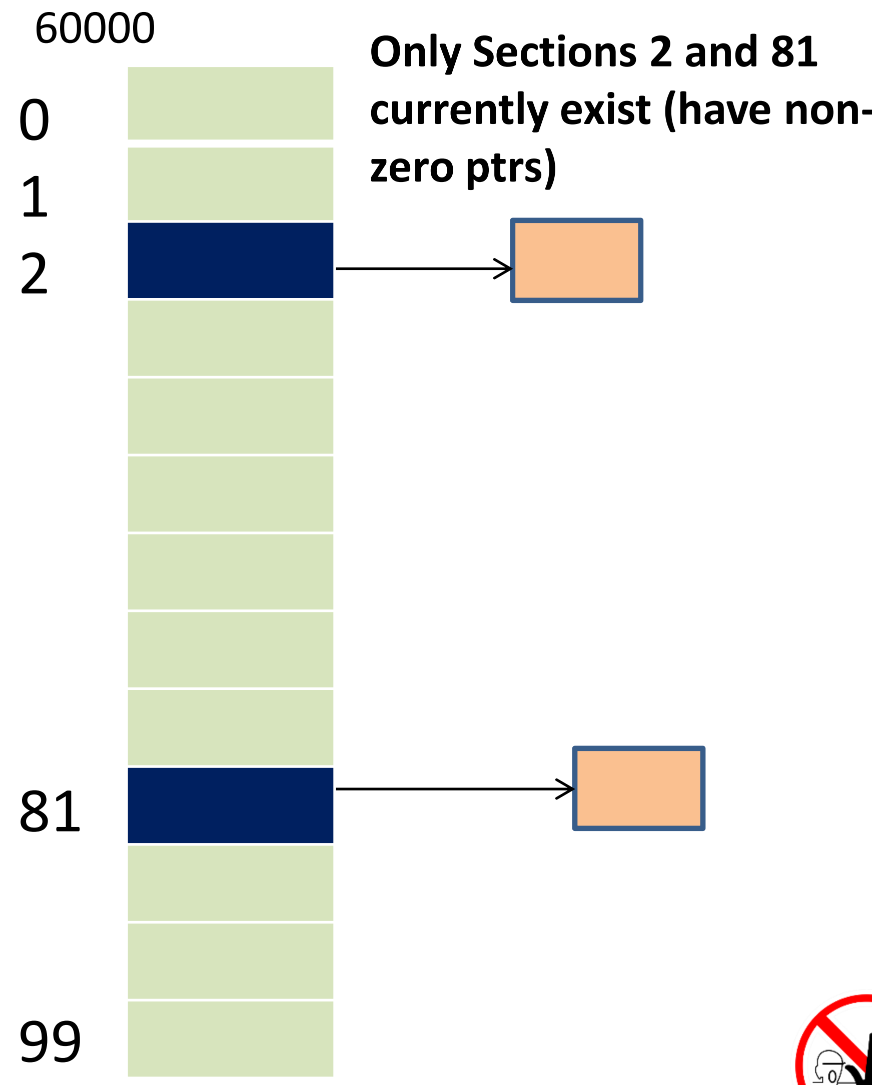
  ➔**P = 1** (one entry per bucket)

  - Instantaneous key to entry conversion (hash)

“Perfect” Hash of Section Collection: “Sparse Array” of Course’s Section Pointers?

Advantages:

1. **Section #** IS index
2. No push back on insertion
3. No pull forward on remove
4. Instantaneous retrieval



“Sparse Array” of Course’s Section Pointers?

No.

1. Consider:

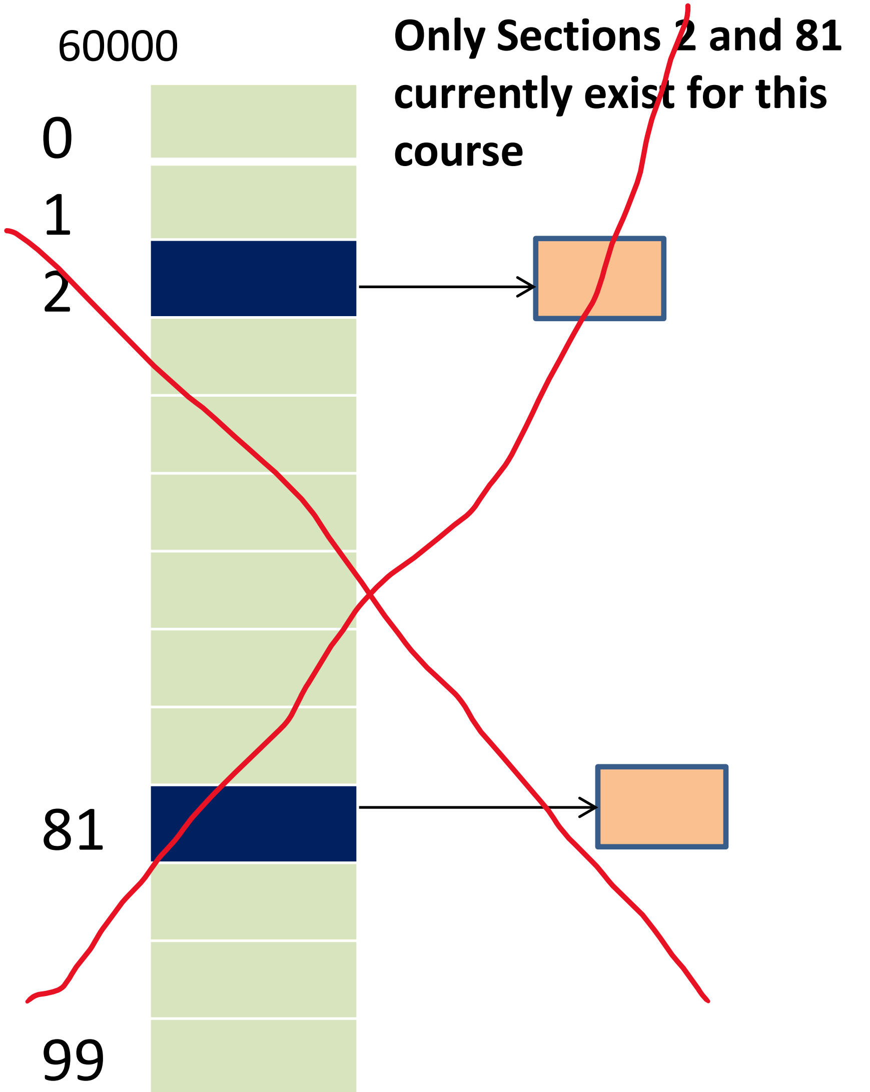
CIS 22B 04YY

CIS 22B 04 ZZ

Qualifiers like YY and ZZ are not handled

2. At < 20 sections / course, and infrequent inserts / removes,

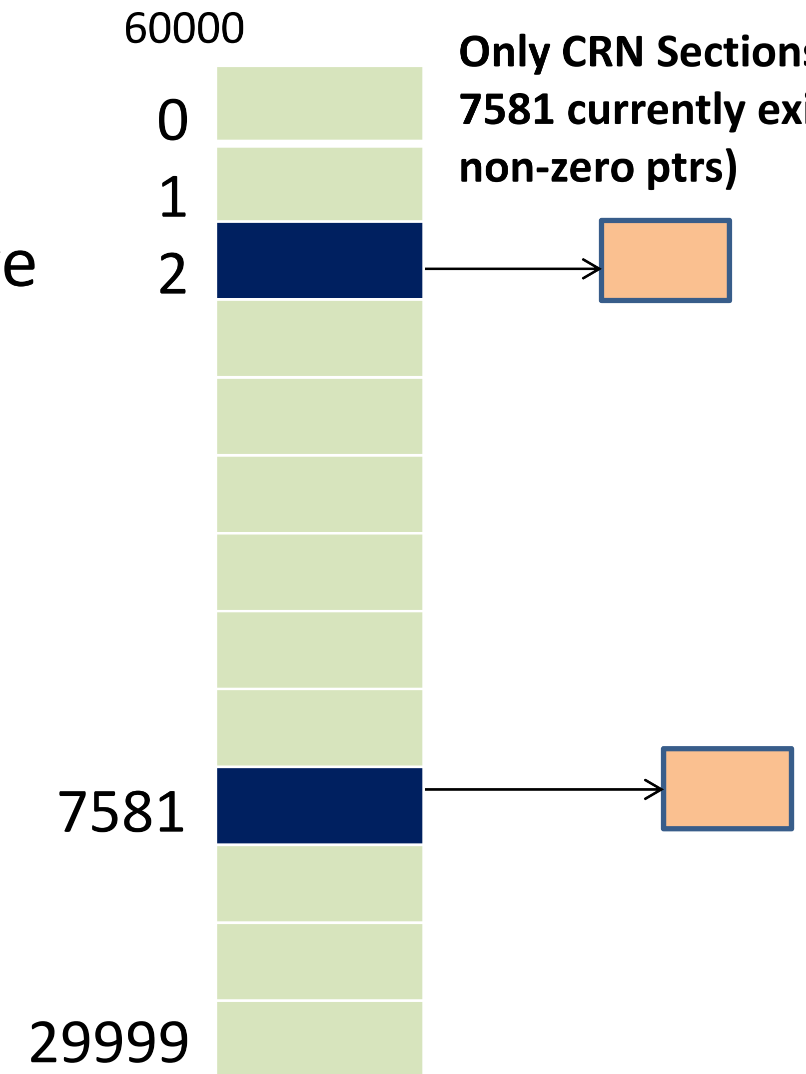
➔use basic Array of Section Ptrs (unordered)



"Perfect" Hash of Section Collection = "Sparse Array" of College's Section Pointers?

ID is CRN #

1. No insertion push back
2. No pull forward on remove
3. Instantaneous retrieval



Only CRN Sections 2 and 7581 currently exist (have non-zero ptrs)

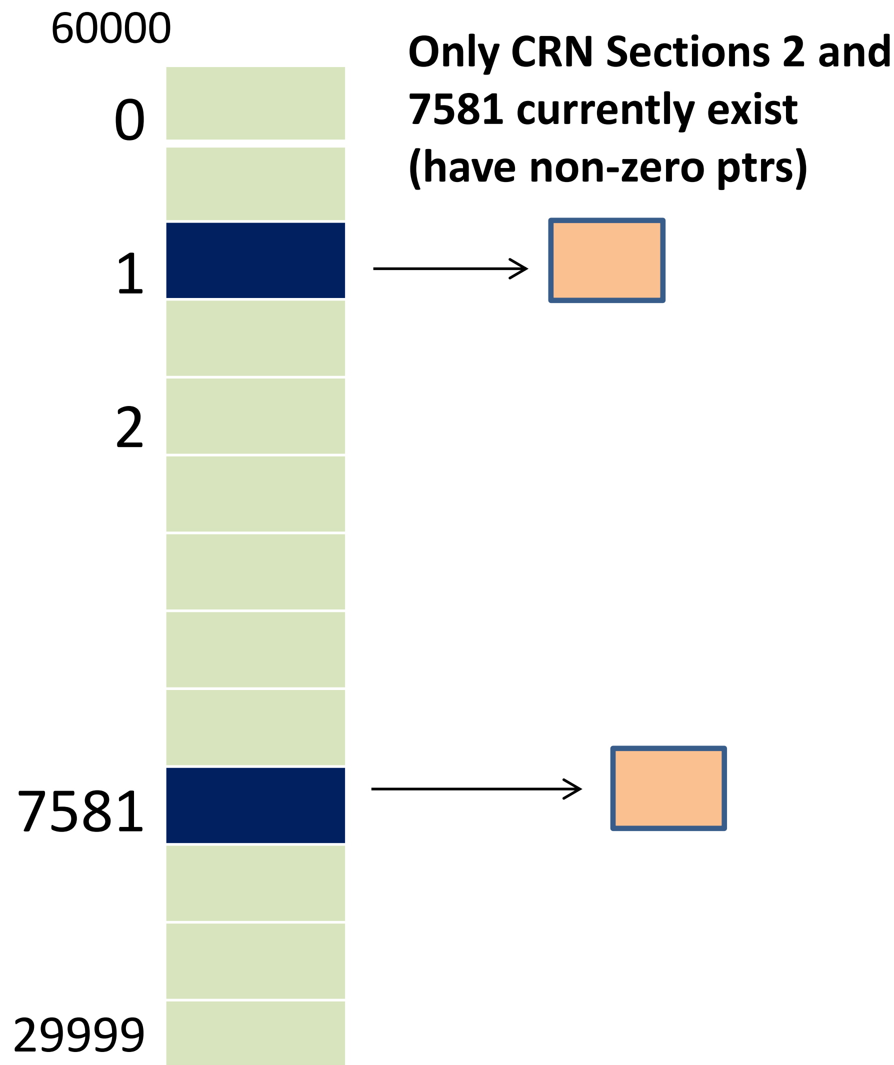
“Sparse Array” of College’s Section Pointers?

Yes.

- This solution requires an array of size 30,000 to access < 3000 sections.

+ But each array element is only a pointer

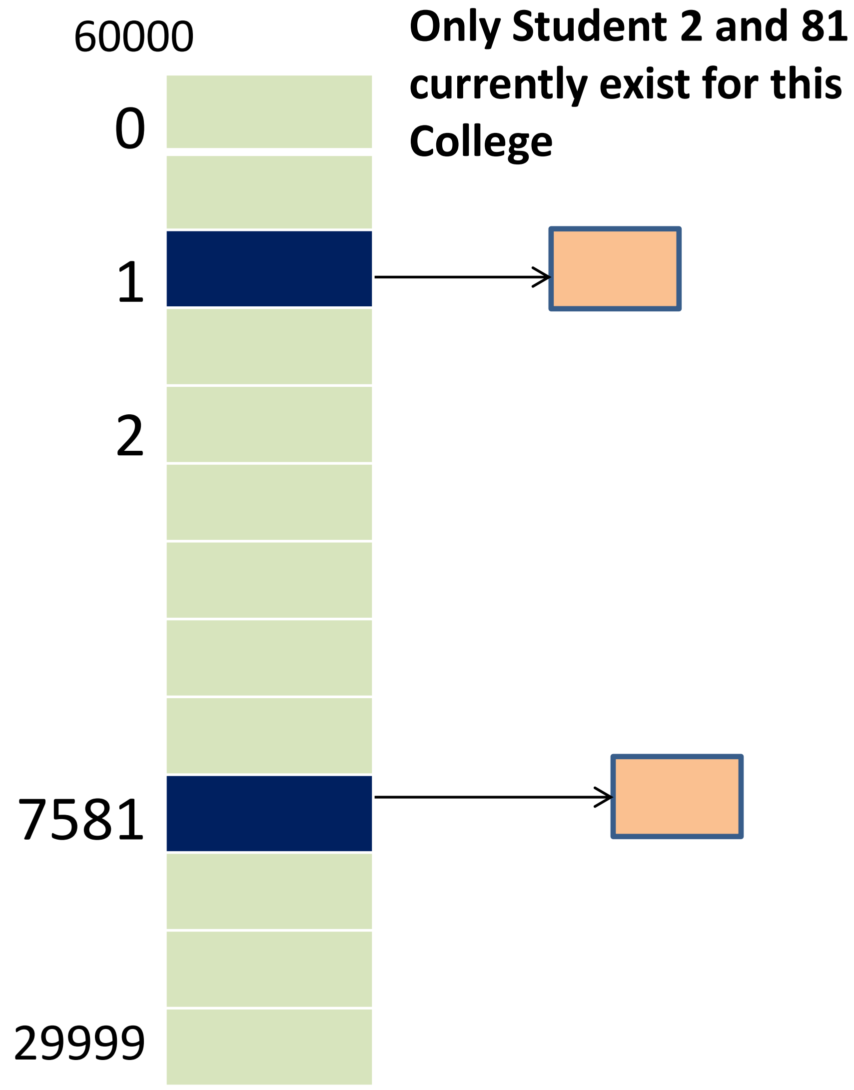
➔ “Perfect Hash” is no Hash at all.



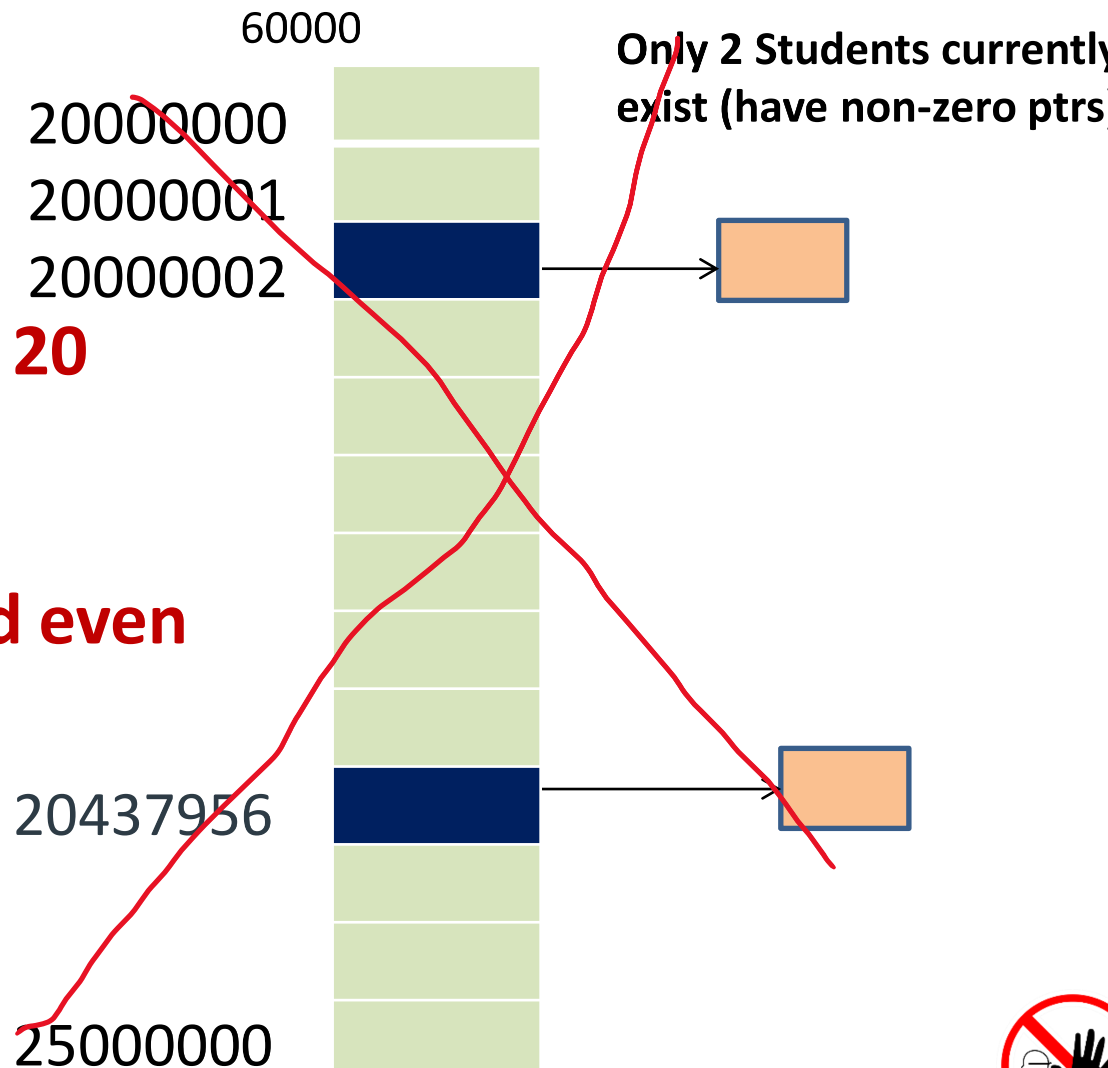
“Sparse Array” of College’s Student Pointers?

Advantages:

1. **Student ID** IS index
2. No push back on insertion
3. No pull forward on remove
4. Instantaneous retrieval



"Perfect" Hash of Student Collection: "Sparse Array" of College's Student Pointers?



Student ID's range from 20 million to 25 million

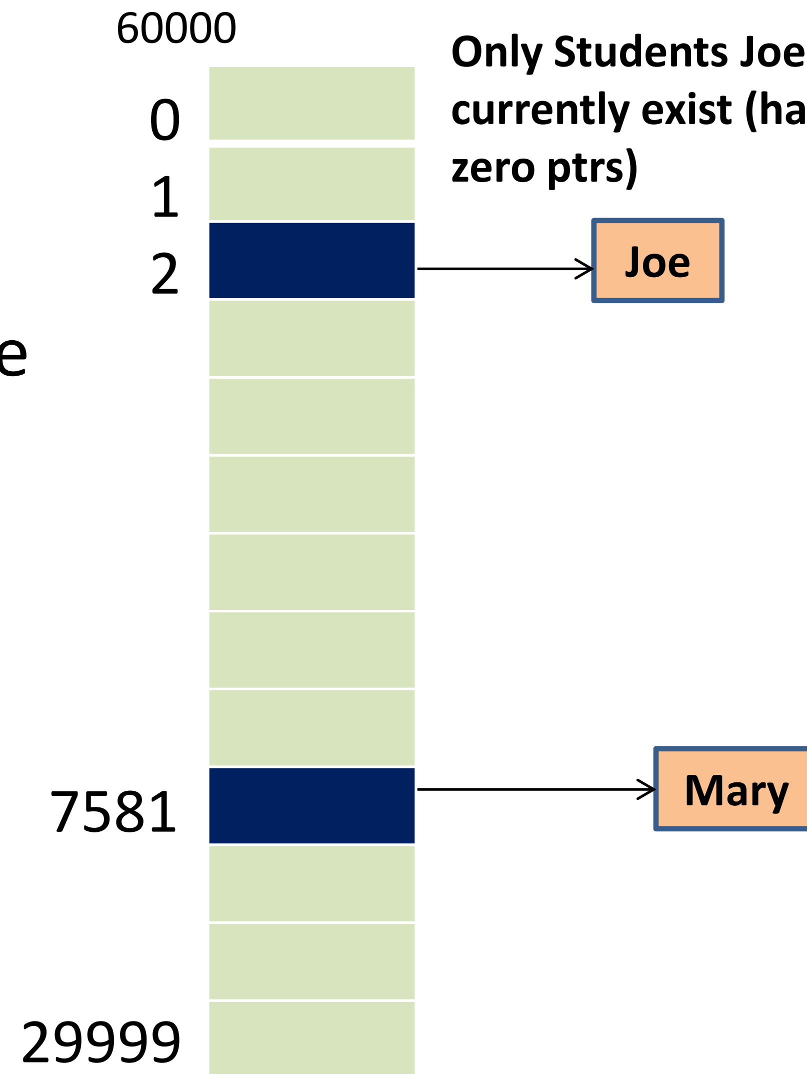
That's too wide a spread even if subtract 20 million

➔ use "Hash Table"

“Perfect” Hash of Student Collection: “Sparse Array” of College’s Student Names?

Advantages:

1. **Student Name** IS index
2. No push back on insertion
3. No pull forward on remove
4. Instantaneous retrieval



Only Students Joe and Mary currently exist (have non-zero ptrs)

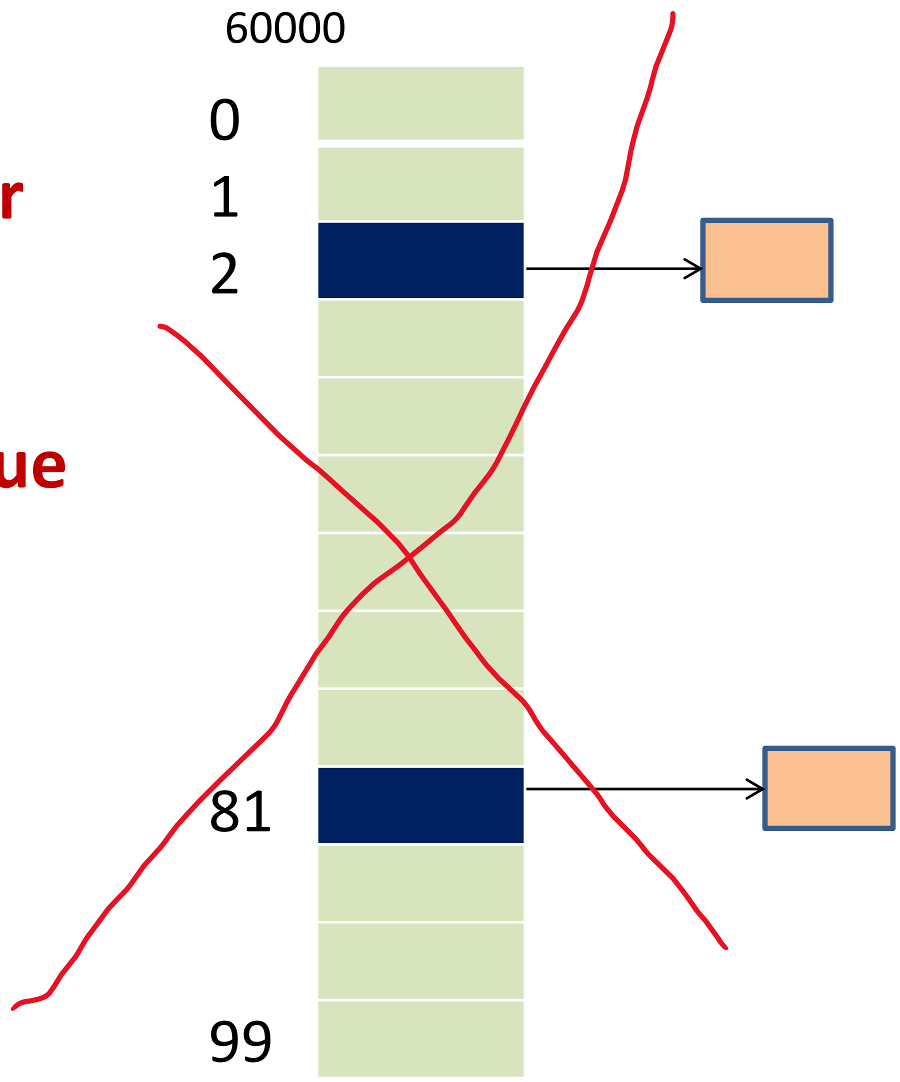
“Sparse Array” of College’s Student Names?

Really?

1. **What does the integer index represent?**

2. **There could be >1 value for a given key**

➔ **Use hash table**



The Simplest Hasher Algorithm
(Use on Student Name)

1. Add up the ASCII value of each digit in the Key.
2. Take modulo M of the sum S **(S%M).** (M = # buckets)
3. For 20+ char strings the modulos of the names will **hopefully** evenly distribute across the numbers 0 to M-1

```
unsigned int hasher (String name, unsigned int bucketLimit)
 { // Convert a Student's Name into a bucket # in the specified range
   unsigned int sum = 0; // Sum of all the ASCII characters in the String
   int k = name.length (); // Length of the Student's Name
   for (int i =0; i < k; i++)
        sum += (int) name[i]; // Sum the ASCII value of each character
return (sum % bucketLimit); // Total / # buckets → return remainder
}
```

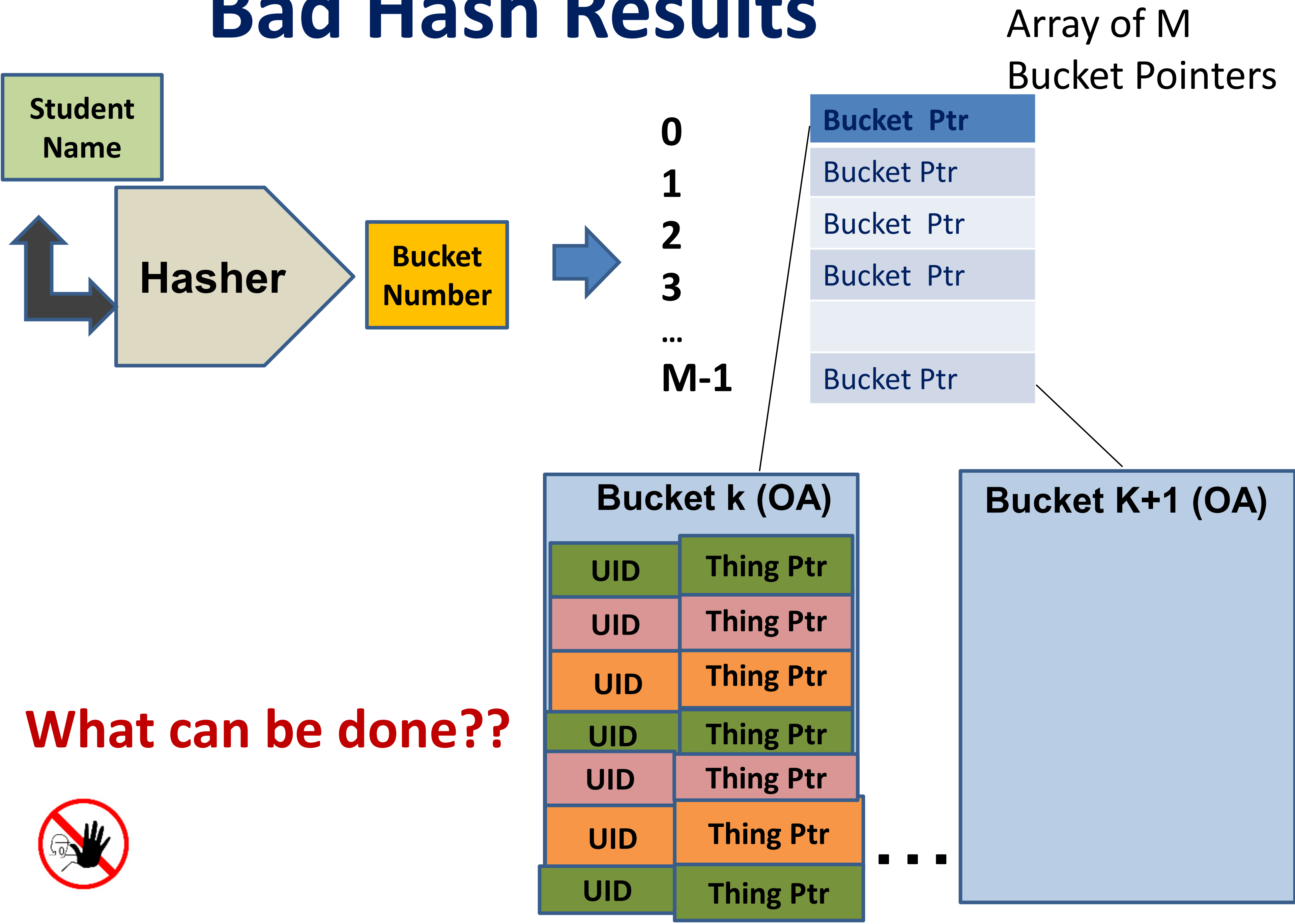
Dangers of the Simplest Hasher

- **Internals**
  - All names with same letters map to same bucket
  - All characters have equal weight
- **Warning: Beware short strings and large # buckets!**
  - Ex: 20,000 students and 10,000 buckets (modulo 10,000)
    - Ideally each bucket has exactly 2 Entries
  - Assume student names average 15 characters:
    - Minimum: 15 * 65(A) = 975
    - Maximum: 15 * 90(Z) = 1350

➔20,000 Entries packed into 375 Buckets (53+ / Bucket)!!

➔9,625 Buckets totally empty

Bad Hash Results



What can be done??

Tuning the Simplest Hasher to the Data #1

(→ Randomly spread the range of possible hashed values)

1. **Break up the Key** into multiple segments (ex: 4-character groups)
2. **Vary the "Weight"** of different characters in the segment to spread out the range of the segment "contribution" (which modulo "**folds over**" the group of M Buckets)
3. **Total up the "Contribution"** of each segment and use as the hash value "modulo'd" by the total # Buckets.

Tuning the Simplest Hasher #1

(Randomly spread range of possible hashed values)

Break name string into a set of 4-character chunks

Zero "*TOTAL*" (Hash value we will be computing)

For each "chunk"

  Reset "*WEIGHT*" to 1; "*CONTRIBUTION*" to 0

  For each character in chunk:

    Multiply ASCII value by WEIGHT

    Add value to CONTRIBUTION

    WEIGHT *= 256 // Do later characters in chunk have more impact?

// Contribution is now the value of this segment to the Hash

  Add CONTRIBUTION to TOTAL ("aaaa" generated 1,633,771,873)

When complete TOTAL is a large relatively random Hash.

<u>NOW</u> set Bucket# = TOTAL % 9997

Impacts of Hashing on Student ID

- Data Description
  - Sample Range: 10,000,000 to 21,000,000
  - Spread of 11 million!
  - Data clustered around:
    - 10,000,000 – 11,500,000 // Spread of 1,500,000
    - 20,000,000 – 21,500,000 // Spread of 1,500,000 [majority]

Should we hash on Student ID directly?

Impacts of Hashing on Student ID

Should we hash on Student ID directly? Yes

- **Create 10,000 buckets**
  - Assume 20,000 – 30,000 students in 11 million range
    - 2-5 Entries / Bucket if random
    - > 1000+ Student Entries / Bucket if not

    ➔ **Examine Student ID concentrations**
  - The empty spaces just "refold" over the # Buckets.
  - "%" speed does not depend on the size of the numerator

➔ **Why not!!**

Tuning #2: Expanding the # Buckets

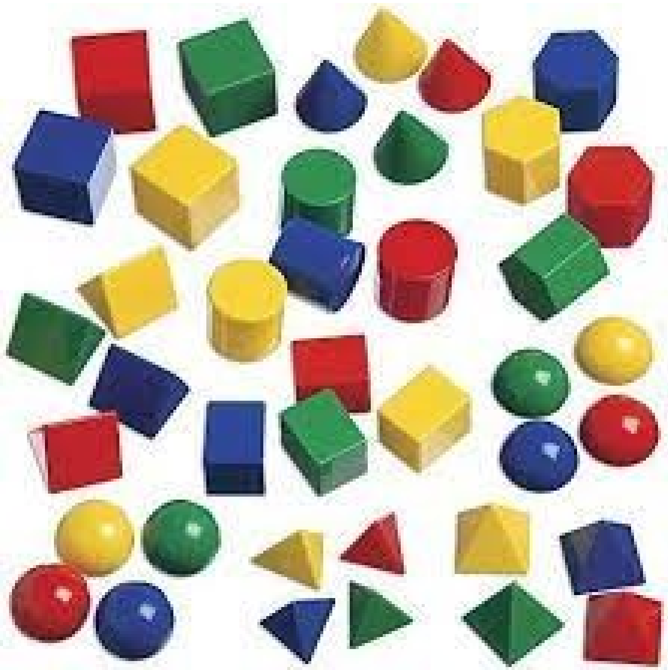
As N (# students) supported by the Hash Table rises, the expected # of Elements in each of M Buckets (P) will also rise.

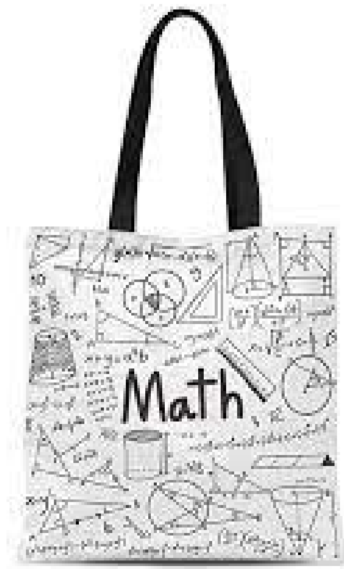
In the same way an Extended Array "resizes itself", so can a Hash Bucket Array. This involves:

- Allocating twice the number of new Buckets (2M)
- Doubling the "modulo %" constant from M to 2M (or a close prime) to distribute elements to one of 2M Buckets.
- Serially walking through all original Buckets and for each key / Thing ptr element pair found, rehash the Key, and store a duplicate element in the resulting Bucket.
- Deleting the old M Buckets

➔ **This should cut the number of Elements per Bucket ~ by 2**

Sets and Bags





Two kinds of Hash Tables: Set / Bag

- **We have assumed hash key is a Unique Object ID**
  - Section CRN # and Student ID: **TRUE**
  - Student Name: **False**

- **A SET hash table has one value for every key**
- **A BAG hash table has possibly multiple values for a given key**

➔ Does anything change for the client of a Bag?

Does anything change for the client of a Bag?

```
class StudentSet  // Key = Student ID (stringified)
{
  public:
    bool add (String, Student*);  // Fails if String == existing Entry string
   Student& get (String);  // Returns matching object Ptr  (or Null)
   bool  remove (String);      //  Removes entry from Bucket (or False)
                               // Corresponding Student Object deleted
  unsigned int getSize(); // Returns # key/value pairs in Table
```

➔Yes. The API changes! A Set is <u>not</u> a Bag

What specific API changes need to be made?

Does anything change for the client of a Bag?

1. **“*Add*”** now allows additions of entries which duplicate existing keys
2. ***“Get”*** returns unordered Array container (usually size 1) of Values (easy, since all entries with the same key map to the same Bucket)
3. **“Remove”** requires specification of key AND value (inelegant, but client must know which object should be deleted)

Does anything change for the client of a Bag?

```
class StudentBag  {   // Ex: Key = Student Name
  public:
     bool add (String, Student*);  // Does NOT fail
                                   // if String == existing Entry string
    StudentEA& get (String);  // Returns ptr to OA if >1 matches, or Null
    bool remove (String, Student*);   //  Deletes  only THIS entry
    unsigned int getSize(); // Returns # key/value pairs in Table
};
```

Relationship of Set to Bag

1. Set inherits from Bag
2. Bag inherits from Set
3. Set wraps Bag
4. Bag wraps Set
5. Totally separate collection classes

```
Class ThingSet  // Ex: Key = Student ID (unique)
{
  private:
     ThingBag tb;  // Wrap Bag – it does more than Set needs

  public:
    bool add(key, object)
   {  // Must fail if key is duplicate of an existing key in Table
         if (tb.get (key) != Null)  // Object with this key already exists!!
              return (FALSE);
         tb.add (key, object);  // Add to collection … it's a unique object
              return (TRUE);
    }
   Thing *get (key);  // This is a Set, so there is only 1 possible.
                      // Returns matching object from EA[0] (or Null)

   remove (key) { // Must get object to give to Bag Delete
                  // Call tb.get (key)  If success, object returned
                  // Then Call tb.remove (key, object)
```

Optimizing Collections: Enrollment

Collector	Object Collected	Collection Type	Object Identifier / Qualifier	Order	Example Of Use
Department	Teacher			By Tenure	Section Assignment
College	Student		Student ID	None	Find Student given ID
College	Student		Student Name	None	Find Student given Name
Department	Course		Course #	Course #	Print catalog

Optimizing Collections: Enrollment

Collector	Object Collected	Collection Type	Object Identifier / Qualifier	Order	Example Of Use
Department	Teacher Ptrs	FIFO Q		By Tenure	Section Assignment
College	Student Ptrs	Hash Table Set	Student ID	None	Find Student given ID
College	Student Pts	Hash Table Bag	Student Name	None	Find Student given Name
Department	Course Ptrs	Ordered Array	Course #	Course # (use as Ordered Array Index)	Print catalog

Design Patterns

- **Data Structure**
  - Memory arrangement of multiple objects
  - Ex: Unordered Array, Bidirectional List, Hash Table
- **Data Abstraction**
  - Provides external vision of that arrangement (public API)
  - Ex: Ordered Array, Stack, Queue, Set, Bag, …
- **Algorithm (<u>How</u> to Use)**
  - Function that leverages one or more Data Abstractions
  - Ex: Hasher, Sort, Recursion, …
- **Design Pattern (<u>When</u> to use)**
  - Leverages abstractions and Algorithms to solve Application or Developer Need
  - Ex: Wrapper, Factory, …

Factory Design Pattern: Construction

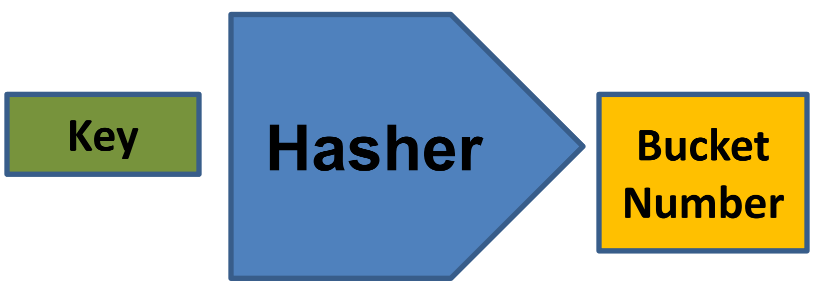
- **Encapsulate "Constructor" Arguments**
  - Constructor "initializes" hidden variables
- **Object attributes obtained from**:
  - External User Interface (ex: "Schedule" boundaries)
  - Other applications (ex: Student Transcript from SIS)
  - Enterprise Data Base (SQL requests) – Give Obj ID
- **Get (lazy load)!!!** Thing &Table::get (key);
  - If already in a memory bucket, return reference
  - If not in memory, gather data, instantiate object and return reference
  - Objects load in memory <u>as needed</u>!

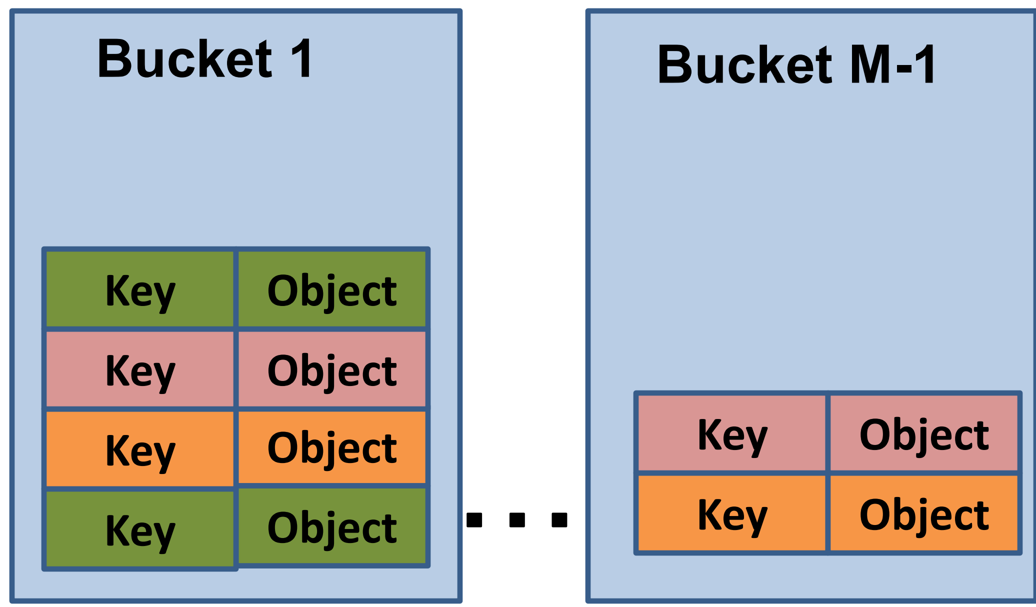
Factory: Supply Object – Construct as Needed

```
class StudentFactory  // Ex: Key = Student ID {
  public:
    insert (uid, Student*);  // Fails if matches another key
    remove (uid);       //  Deletes  uid/Student* entry if exists
   Student *get (uid);  // Returns Student* with that UID if exists
         // Otherwise create Student (gather data via uid)
        // Hash created Student UID to Bucket, return Student*.
```

Students constructed dynamically, only as needed!!

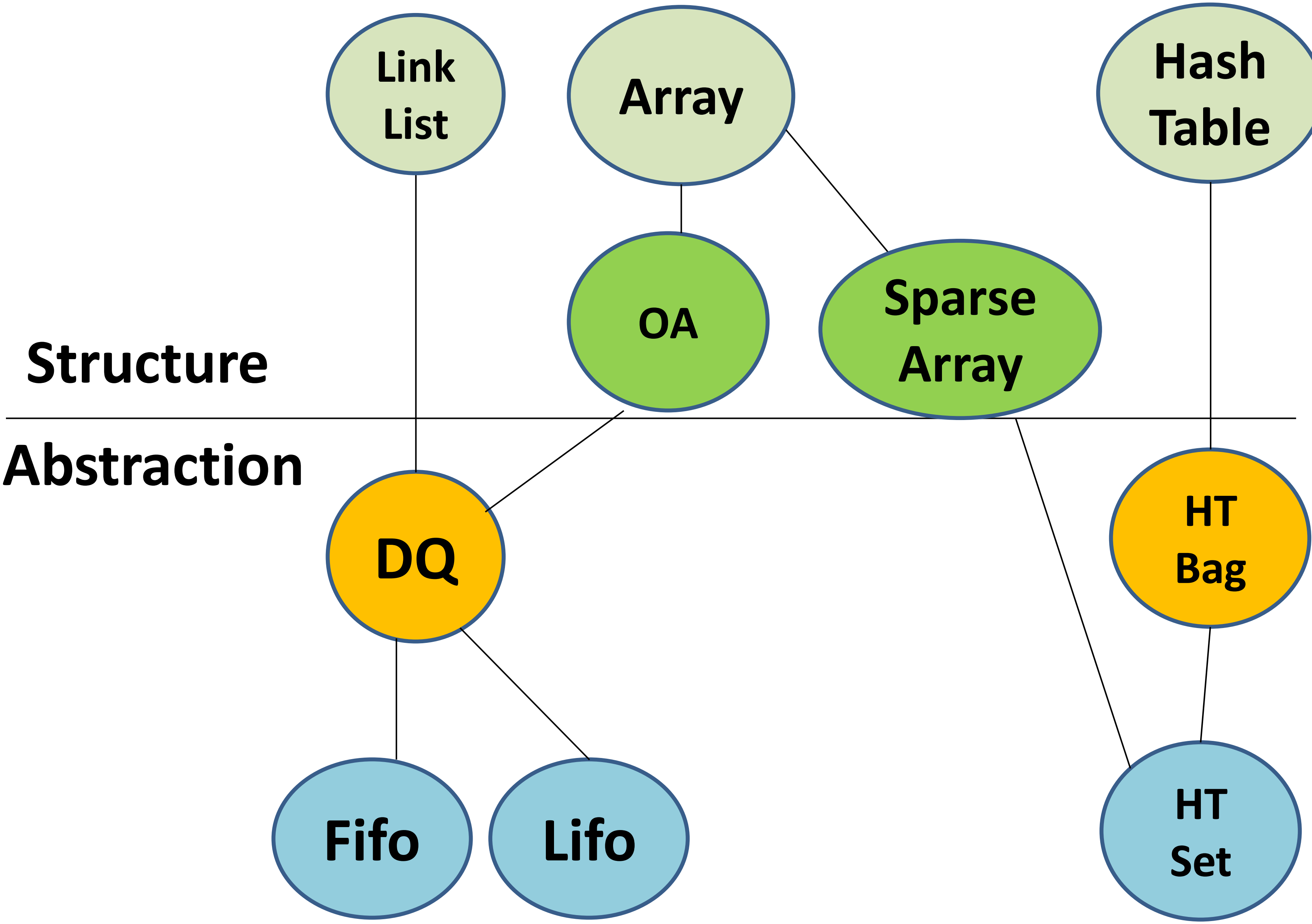
Factory: Design Pattern





Hash Table: Data Structure
Assoc. Array Data Abstraction
Hasher: Algorithm

Relationship of Collections



Data Structure Comparisons

Operation / Data Structure	**Add**	**Remove**	**Get**	**Report: GetNext GetPrev**
Sparse Array	**Instantaneous.** Assign to index slot	**Instantaneous.** Remove from index slot.	**Instantaneous.** Assign to index slot.	**Slow:** Walk thru all index spaces
Ordered Array	**Slow**. Insert → pushback	**Slow**: Remove → move down	**Fast**. Binary search to item	**Fast.** Binary Search to current item
Linked List	**Slow:** "walk" ½ entries	**Slow:** "walk" ½ entries	**Slow:** "walk" ½ entries	**Instantaneous** Move to adjacent entry
Hash Table	**???**	**???**	**???**	**???**

Data Structure Comparisons

Operation / Data Structure	Add	Remove	Get	Report: GetNext GetPrev
Sparse Array	**Instantaneous.** Assign to index slot	**Instantaneous.** Remove from index slot.	**Instantaneous.** Assign to index slot.	**Slow:** Walk thru all index spaces
Ordered Array	**Slow.** Insert → pushback	**Slow**: Remove → move down	**Fast.** Binary search to item	**Fast.** Binary Search to current item
Deque	**Slow:** "walk" ½ entries	**Slow:** "walk" ½ entries	**Slow:** "walk" ½ entries	**Instantaneous** Move to adjacent entry
Hash Table	**Very Fast.** Hash key, assign to Bucket. Append	**Very Fast.** Hash Key, find Bucket, Delete	**Very Fast.** Hash Key, find Bucket. Get.	**Unsupported.** No "ordering"

Hash Table vs. Ordered Array

Comparisons (N elements)	Hash Table	Ordered
Element "Find"	Hash to Bucket. Assume very few Elements **O(1)**	Binary search for specific element **O(log N)**
New Element Addition	Hash to Bucket. Insert **O(1)**	Find O(log N) + push every element past insertion point back one = **~O(N/2)**
Element Deletion (once there)	Hash to Bucket. Remove **O(1)**	Find O(Log N) + push every element past remove point forward one = **~O(N/2)**
Element "Ordering"	**N/A**	Ordered at Element insertion. **O(1)**
Reporting	**N/A** Elements are not ordered and cannot be easily sorted	All elements are present in order **O(N)**