Lesson 11
Hash Tables:

Pure Consciousness is the Home of All Knowledge

Wholeness Statement

A hash table provides constant-cost find, insert, and delete operations on values by key; it can quickly decide where the data value should be located for fast insertion and retrieval.

The home of all the laws of nature is available and accessible in the field of pure consciousness.

Hash Table vs. Array

- Definition : An array into which data is inserted using a hash function is called a hash table.
- Array
  - An integer index is the key
- A Hash table is a generalization of an array
  - *keys* are used to look up corresponding *values*
  - Any object can be used as a key
- Example
  - A key field from a database can be used as a key in the hash table
  - the corresponding DB record could be the value in the hash table

⦿ Two basic operations: (usually also have a remove(Object key) operation)

› void put(Object key, Object value);

› Object get(Object key)

⦿ Simple Example (this is an elementary implementation of Hashtable ADT)

User's view

Char key	String value
'a'	"Adam"
'b'	"Bob"
'c'	"Charlie"
'w'	"William"

```
//insert into table
table.put('c', "Charlie");
//retrieve from table
table.get('c');   //returns "Charlie"
```

Implementation:

put(c, s):
obtain array index: c → (int)c → i = (int)c – 'a'
insert new Entry(c, s) into table[i]

get(c):
obtain array index: c → (int)c → i = (int)c – 'a'
Entry e = table[i]; return e.value

Pattern: KEY (non-number) → HASHCODE (number) → HASH VALUE (array index)

◉ Basic steps in creating a hashtable data structure:

a. Devise a way of converting keys to integers so that different keys are mapped to different integers. This is what Java's hashCode() function is for – must be overridden by every class (since hashCode() is a method of the Object class).

b. Devise a way of converting hashcodes to smaller integers (called *hash values*) that will be the indices of a smaller array, called the *table.* To do this, you usually need to decide on the tableSize, which is the length of the array. A typical way (note: Java does it differently – see below) of making hashvalue from hashcode is by the formula

hashvalue = hashcode % tableSize

key ⟶ hashcode ⟶ hashvalue
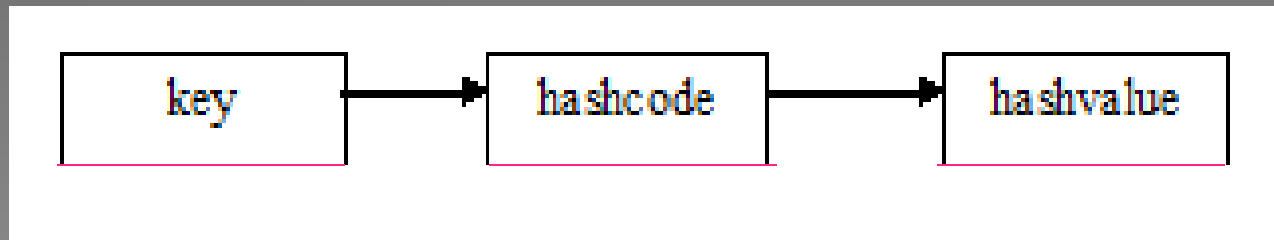
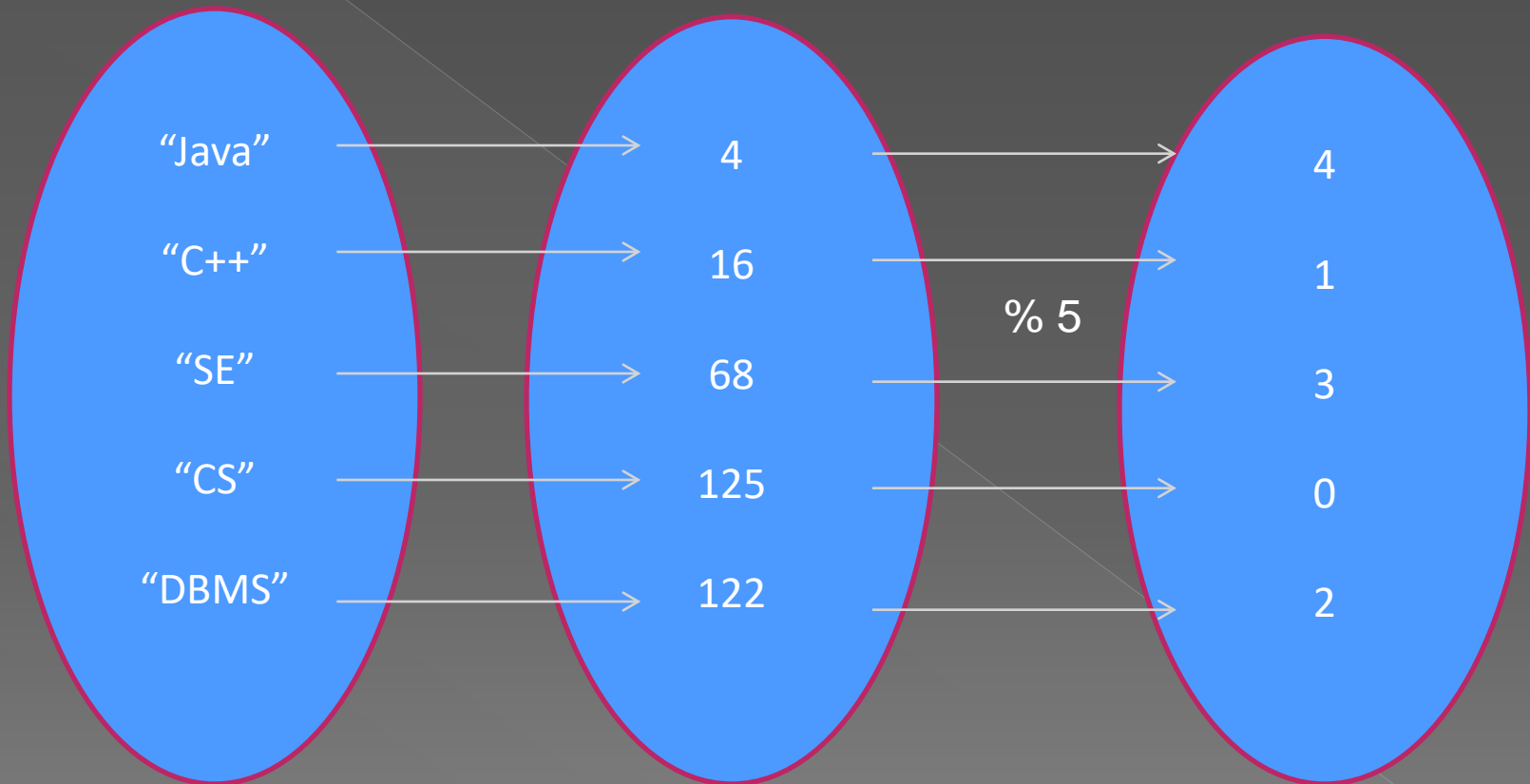# Example

Data item	Key Value	Table index = Key Value % table size

"Java" → 4 → 4

"C++" → 16 → 1

"SE" → 68 → % 5 → 3

"CS" → 125 → 0

"DBMS" → 122 → 2

CS	C++	DBMS	SE	JAVA
0	1	2	3	4

Computing a hashcode for a string in Java

The method hashCode has different implementation in different classes.

Devise a way of converting keys to integers so that different keys are mapped to different integers. This is what Java's hashCode() function is designed to do.

Example: Any Java String is converted to an integer via hashCode() by this formula: (Note that hashCode() is overridden in the String class.

In the String class, hashCode is computed by the following formula
s.charAt(0) * 31^{n-1} + s.charAt(1) * 31^{n-2} + ... + s.charAt(n-1)

where *s* is a string and *n* is its length. An example

String s = "ABC";   n = 3 (Size of the string)

"ABC" = 'A' * 31^2 + 'B' * 31 + 'C' = 65 * 31^2 + 66 * 31 + 67 = 64578

Collisions

- Two values can hash to the same array index, resulting in collision. It can be resolved using :

- **Open Addressing**: Search the array in some systematic way for an empty cell and insert the new item there if collision occurs.

  - Linear Probing (Search sequentially for vacant cells until find an empty)

  - Quadratic Probing(In quadratic probing, probes go to x+1, x+4, x+9, and so on)

  - Double Hashing(Double the sequence with constant factor)

- **Separate chaining**: Create an array of linked list, so that the item can be inserted into the linked list if collision occurs.

Creating Good Hash Codes

- User implementation you must specify the size of the hash table.

- In order to reduce the chance for collisions, you should make a hash table somewhat larger than the number of elements that you expect to insert.

- The table size should be a prime number, larger than the expected number of elements.

- In the standard library, you don't need to supply a table size. If the hash table gets too full, a new table of twice the size is created, and all elements are inserted into the new table.

The Hash Table ADT

- Create an empty hash table

    new object constructor call

- Add a key-value pair to hash table

    void put(Object key, Object value)

- Retrieve the value associated with a key

    Object get(Object key)

- Remove a key-value pair from hash table

    Object remove(Object key)

- Remove all key-value pairs from hash table

    void removeAll( )

Demo :MyHashTable.java, MyHashTableTest.java

Hash Tables

- Advantages
  - Simpler and faster than binary search tree implementations

- Disadvantages
  - Does not efficiently support traversing the table in sorted order
  - Requires estimating the maximum number of table items

Creating Your Own Hashtable

```java
public class MyHashtable {
        private static final int INITIAL_SIZE = 19;
        private int tableSize;
        private LinkedList[] table;
        public MyHashtable(){
                this(INITIAL_SIZE);
        }
        public MyHashtable(int tableSize) {
                this.tableSize = tableSize;
                table = new LinkedList[tableSize];
        }
```

```java
// FIRST TRY (needs to be fixed -- see SECOND_TRY BELOW)
public void put(Object key, Object value){
        //disallow null keys
        if(key==null) return;

        //get the "big" integer corresponding to the object
        //assumes key is not null
        int hashcode = key.hashCode();

        //compress down to a table slot
        int hash = hash(hashcode);

        //put the value and the key into an Entry object
        //which will be placed in the table in the
        //slot (namely, hash)
        //allows a null value
        Entry e = new Entry(key,value);

        // now place it in the table
        if(table[hash] == null){
                table[hash] = new LinkedList();
        }
        table[hash].add(e); // add the entry in the table
}
```

Big Problem: Suppose a client class attempts these put operations:

```
put(key, "Bob")
put(key, "Dave")
```

Suppose the hashvalue for key is 5. In the approach above, there will be two Entries placed in the list in slot 5 – [key, "Bob"] and [key, "Dave"]. Then there will be unpredictable results when a get(key) operation is performed.

```java
// SECOND_TRY
public void put(Object key, Object value){
    //disallow null keys
    if(key==null) return;

    //get the "big" integer corresponding to the object
    //assumes key is not null
    int hashcode = key.hashCode();

    //compress down to a table slot
    int hash = hash(hashcode);

    //create the entry
    Entry e = new Entry(key,value);

    boolean keyAlreadyInUse = false;
    if(table[hash] != null) {
      for(Object ob : table[hash]) {
            Entry ent = (Entry)ob;
            if (ent.key.equals(key)) {
                    keyAlreadyInUse = true;
                    ent.value = value; //update value for this Entry
            }
       }
    }
    //we handled case keyAlreadyInUse==true in loop
    if(!keyAlreadyInUse) {
            // now place it in the table
            if(table[hash] == null){
                    table[hash] = new LinkedList();
            }
            table[hash].add(e);
    }
}
```

```java
public Object get(Object key){
        //null key not allowed
        if(key==null) return null;
        //get the "big" integer corresponding to the object
        int hashcode = key.hashCode();

        //compress down to a table slot
        int hash = hash(hashcode);

        //if slot given by hash not yet in use, return null
        if(table[hash] == null) return null;

        //now look for the desired Entry
        Entry e = null;
        for(Iterator it = table[hash].iterator();
        it.hasNext();){
                e = (Entry)it.next();
                if(e.key.equals(key)) {
                        return e.value;
                }
        }
        return null;

}
```

```java
public String toString(){
        String n = System.getProperty("line.separator");
        StringBuilder sb = new StringBuilder();
        for(int i = 0; i < table.length;++i){
                if(table[i] != null){
                Entry next = null;
                        for(Iterator it = table[i].iterator(); it.hasNext();){
                                next = (Entry)it.next();
                                if(next.value!=null)
                                        sb.append(next + n);
                                else
                                        sb.append(next.key + "->null" + n);
                        }
                }
        }
        return sb.toString();
}
```

```java
 private int hash(int bigNum) {
     return bigNum % tableSize;
 }

private class Entry{
    private Object key;
    private Object value;
    Entry(Object key, Object value){
        this.key = key;
        this.value = value;
    }
    public String toString(){
        return key.toString()+"->" +
            value.toString();
    }

 }
}
```

Quiz

1. Efficient data structure for Searching from the index directly without doing linear or binary search

   a. Binary Search Tree b. Hash Table c. LinkedList

2. Hash Table is an Array based Data Structure.

   a. True   b. False

3. Which one is the easiest structure to maintain Sorted list?

   a. Hash Table   b. Binary Search Tree

Main Point 1

A hash table has three components: a hash function to convert keys to slots in a table, an array containing key-value pairs, and a collision resolution strategy. The field of Pure Creative Intelligence contains all knowledge, in seed form, that is necessary for maintaining order in the Universe.

Hash Table Applications

- The most common use of hashtables is as in-memory look-up tables. For example, Employee records from a database could be stored by using Employee ID as key and the entire Employee record as value.

- Another application of hashtables is for "bookkeeping" purposes. A simple example is an efficient procedure for removing duplicates from a list. The "naïve" way to remove duplicates is to use nested loops: For each element e in the list, use an inner loop to look at all elements preceding e in the list to see if e has occurred before; if so, remove this second occurrence of e.

- A more efficient approach is to do the following: Create an auxiliary hashtable H. For each e in the list, check to see if e is a key in H. If so, remove e from the list. If not, add the entry <e , e> to H.

- For a list having 1000 elements, the second procedure requires roughly 2000 steps of execution, whereas the first procedure requires on the order of 1,000,000 steps.
- Set ADT

The Set Abstraction

- A set is a collection that contains no duplicate elements and at most one `null` element
  - adding `"apples"` to the set
    `{"apples", "oranges", "pineapples"}` results in the same set (no change)
- Operations on sets include:
  - testing for membership [contains()]
  - adding elements [add()]
  - removing elements – [remove()]
  - union   A ∪ B [addAll()]
  - intersection   A ∩ B [retainAll()]
  - difference   A – B [removeAll()]
  - subset   A ⊂ B [containsAll()]
- containsAll tests the subset relationship
- addAll, retainAll, and removeAll perform union, intersection, and difference, respectively.

Predefined Library for Hash Concepts

HashSet : Hash table implementation of the Set interface.

HashMap : Hash table implementation of the Map interface.

> Like Hashtable it also accepts key value pair.

> It allows null for both key and value.

> Does not allow duplicate keys.

> It is unsynchronized. So come up with better performance in a Single threaded environment

> Iterator is used to Iterate.

Hashtable : Hash table implementation of the Map interface.

> Hashtable is basically a datastructure to retain values of key-value pair.

> Does not allow duplicate keys.

> It didn't allow null for both key and value. You will get NullPointerException if you add null value.

> It is synchronized. Useful in Multithreaded Environment.

> Enumeration is used to Iterate.

Java's Implementation of Hashtables

- Pre-j2se5.0: HashMap and Hashtable (HashMap is preferred; Hashtable is "legacy")
  Example:

```
HashMap map = new HashMap();
map.put("Bob", new Employee("Bob", 40000,
    1996, 10, 2));
Employee emp = (Employee)map.get("Bob");
```

Feature	java.util.HashMap	java.util.Hashtable	MyHashtable
Allows null key	Yes	No	No
Allows null values	Yes	No	Yes
Allows duplicate keys	No	No	No
Synchronized (for safe multithreading)	No	Yes	No

24

Overriding the hashCode() Method

1. Any implementation of the Hashtable ADT in Java will make use of the hashCode() function as the first step in producing a hash value (or table index) for an object that is being used as a key.

2. Default implementation of hashCode() provided in the Object class is not generally useful.

Example: We wish to use pairs (firstName, lastName) as keys for Person objects in a hashtable. (See Demo)

Demo illustrates that default hashCode method is not useful. By default, it simply gives a numeric representation of the memory location of an object. If two Pair objects, created at different times, are equal (using the equals method), we would expect them to have the same hashCodes, so that, after hashing, they are sent to the same table slot. But default hashCode method does not take into account the fields used by equals method, so equal Pair objects may be assigned different slots in the table.

3. **Conclusion:** *Whenever* equals *is overridden in a class,* hashCode() *must also be overridden.*

- **Example.** *Overriding hashCode in the Person-Pair example.* We must take in account the same fields in computing hashCode as those used in overriding equals. The fields in Pair are Strings, and Java already provides hashCodes for Strings. So we make use of these and combine them to produce a complex hashCode for Pair.

```java
public int hashCode() {
        int result = 17; //seed
        int hashFirst = first.hashCode();
        int hashSecond = second.hashCode();
        result += 31 * result + hashFirst;
        result += 31 * result + hashSecond;
        return result;
}
```

Creating a Hash Value from Object Data
(From Effective Java, 2nd Ed.)

- You are trying to define a hash value for each instance variable of a class. Suppose f is such an instance variable.

  - › If f is boolean, compute (f ? 1 : 0)
  - › If f is a byte, char, short, or int, compute (int) f.
  - › If f is a long, compute (int) (f ^ (f >>> 32))
  - › If f is a float, compute Float.floatToIntBits(f)
  - › If f is a double, compute Double.doubleToLongBits(f) which produces a long f1, then return (int) (f1 ^ (f1 >>> 32))
  - › If f is an object, compute f.hashCode()

Formula for creating your hashCode function

Step 1. Use the table above to produce a temporary hash of each variable in your class.

*Example***:** You have variables u, v, w. Produce (using the chart above) temporary hash vals hash_u, hash_v, hash_w.

Step 2. Combine these temporary hashes into a final hashCode that is to be returned

     Example:

```
int result = 17;
result += 31 * result + hash_u;
result += 31 * result + hash_v;
result += 31 * result + hash_w;
return result;
```

Quiz

1. Which one allow null keys and null values

   a. HashTable   b. HashMap

2. Which one is safe in mulltithread?

   a. HashTable   b. HashMap

3. HashSet allow duplicate values

   a. True   b. False

4. hashCode() method of an object return _____ value

   a. String   b. boolean   c. int

Demo Code

- HashSetDemo.java
- HashMapDemo.java
- HashtableDemo3.java
- needoverridehashcode package
- Employee.java
- Applets Folder

Use Of Data Structures

- ◉ Array List
  - › <u>Use</u>: when main need is for a list with random access reads, infrequent adds beyond initial capacity (or maximum number of list elements is known in advance), or the list only needs to occasionally be sorted
  - › <u>Avoid</u>: when many adds expected, but number of elements unpredictable, or inserts need to maintain some ordering by key.
- ◉ Linked List
  - › <u>Use</u>: when insertions and deletions are frequent, and/or many elements need to be added, but total number is unknown in advance
  - › <u>Avoid</u>: when there is a need for repeated access to the nth element (random access) (as in sorting)

Use Of Data Structures (cont.)

- ◉ Binary Search Tree
  - › <u>Use</u>: when data needs to be maintained in sorted order by key for frequent searches
  - › <u>Avoid</u>: when the extra benefit of keeping data in sorted order is not needed and rapid read access (e.g., iteration or random access) is needed
- ◉ Hash Table
  - › <u>Use</u>: when random access to objects is needed but array indexing is not practical (possible index range is too large)
  - › <u>Avoid</u>: when an ordering (possibly unrelated to keys(maybe by data input)) of data must be preserved, or iterating through values in the table is frequent

Use Of Data Structures (cont.)

- Set

  - Use: when elements do not need to be kept in a special order and searches for elements in the set are infrequent (do not need to be rapid)

  - Avoid: when an ordering of data must be preserved or searches for elements in the set are frequent (instead use TreeSet which uses a binary search tree, and is ordered)

Running Applets

- Applets are small Java programs designed to run as a window inside a web browser.

- Java was originally designed as an Applet-creating language.

- Misleading information has resulted in an unjustified worry about security of applets. At this time (2016), Firefox is the only browser that supports running of applets.

- An alternative is to run applets using the utility appletviewer.exe (located in the bin directory of the jdk distribution).

Running the Demo Applet

Use Hash.html in Demo directory of Lesson 11\ Applets from \\cs5

1. **<u>Run in Firefox</u>**

   › Program Files > Java > Configure Java > Security
   › Make sure "High" and "Enable Java content" are selected
   › Edit Site List > paste full path to the html file in format like this:
       file:///D:/Public\Courses\CS390-FPP\CS390-Renuka\Democode\lesson11\Applets
   › Ignore the warnings.
   › Click OK
   › Drag the html file into Firefox
   › Click "Allow and Remember" when asked
   › Click "Activate Java" if asked
   › If you get a message "Your version of Java is out of date", click Update – but do not let it remove previous versions of Java.
   › If necessary, drag the html file with the applet into Firefox once again
   › It will ask "Do you want to run this application"? Click Run
   › The applet should now run

Running the Demo Applet 2

Run the applet using appletviewer

- Place Hash.html, along with the folder which need .class, .java file, somewhere on your hard drive.

- Open a command window (click   and type cmd.exe and press Enter)

- Using CD commend to change into the specific directory which has the .html applet file

- Type in the window appletviewer Hash.html

- The applet should run.

Main Point 2

With the knowledge of data structures such as Lists, Stacks, Queues, Trees, and Hash Tables one can design programs that run most efficiently and simply. Knowledge and proper use of data structures illustrates the principle of "Do less and accomplish more".

Unity Chart

1. Arrays and ArrayLists provide highly efficient index-based access to a collection of elements.

2. The Hashtable ADT generalizes the behavior of an array by allowing non-integer keys (in fact, any object type can be used for a key), while retaining essentially random access efficiency for insertions, deletions, and lookups.

Transcendental Consciousness: TC is the home of all knowledge. The Upanishads declare "Know that by which all else is known" – this is the field of pure consciousness.

Impulses within the Transcendental field: The infinite diversity of these impulses within the transcendental field, is possible due to the infinite creativity and intelligence.

Wholeness moving within Itself: In Unity Consciousness, one sees that the "key" to accessing complete knowledge of any object is the infinite value of that object, pure consciousness, which is known in this state to be one's own Self.