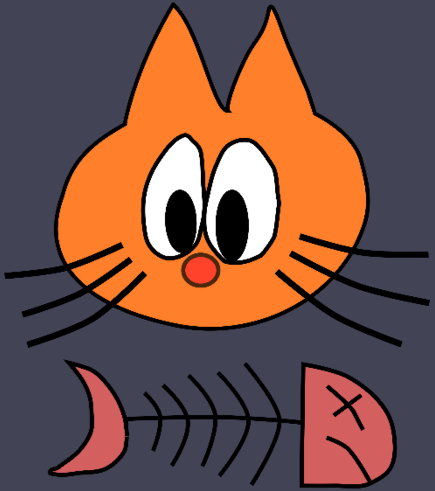


LECTURE 27

Hashing

Assignment

- Read section 7.3
  - Do the following self-check exercises in section 7.3: 7.4, 7.5, 7.6 and 7.7.
  - Do the programming exercise 1.

The requirements for the hash code

- we want to reflect **all mutable data values** in the hash code;
- we want the **bits of the data that vary most randomly** to affect the **lower bits** of the hash code;
- given a set of typical random keys, we want hash codes where every bit has a 0.5 probability of being a 0 (or 1);
- we want the hash function to execute quickly

Don't like hand-waving?
Read D. Knuth's "The Art of Computer Programming"!

Java `HashCode` Method

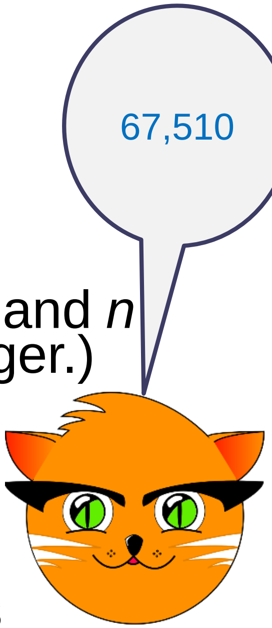
- *Problem:* For strings, hash = sum of the `int` values of all characters returns the same hash code for `"sign"` and `"sing"`
- To solve this, Java API algorithm accounts for position of the characters as well
- `String.hashCode()` returns the integer calculated by the formula:

$$s_0 \times 31^{(n-1)} + s_1 \times 31^{(n-2)} + \ldots + s_{n-1}$$

  where s_i is the integer value of the *i*th character of the string, and *n* is the length of the string. (This is performed on a 32-bit integer.)
- *Example:* String “Cat” has a hash code of:

$$\text{‘C’} \times 31^2 + \text{‘a’} \times 31 + \text{‘t’} = 67{,}510$$

- “31 is a prime number [a true statement], and prime numbers generate relatively few collisions [a meaningless statement]”



Java `HashCode` Method

- But this may be too large for our table, and so we compute:

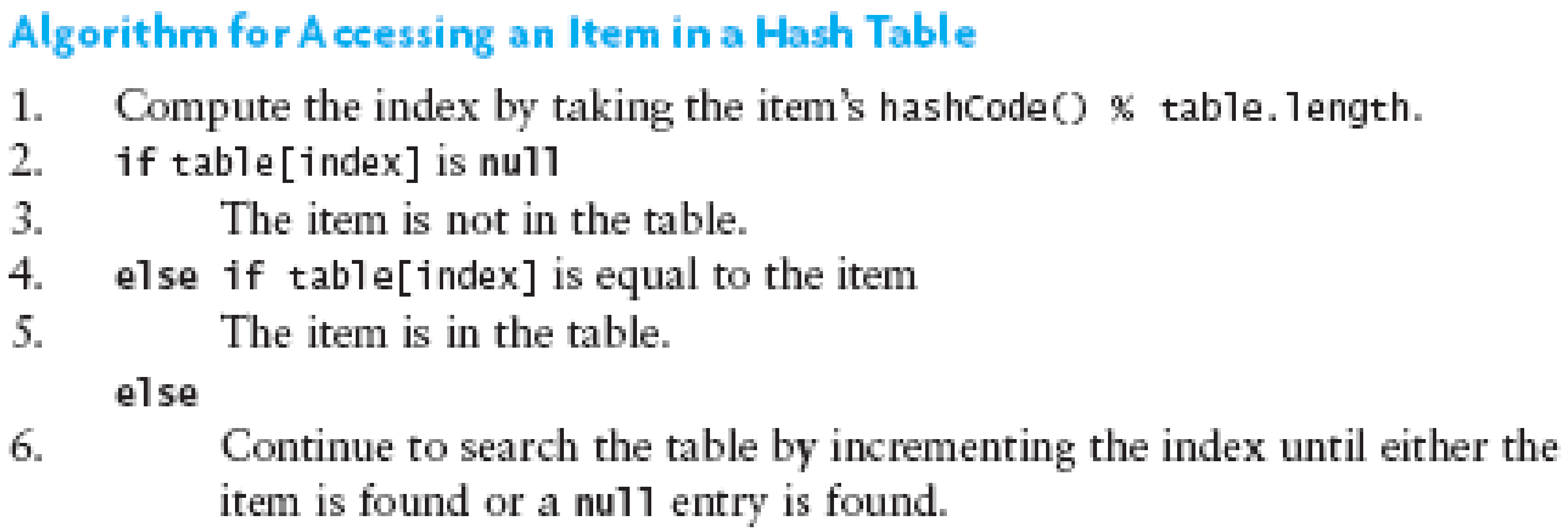
```
HashValue = s.hashCode() % table.length
```

- The probability of a collision, of course, depends on the number of entries in the table (although I don’t see why the books says that it is *proportional* to how full the table is).

Open Addressing

- We now consider two ways to organize hash tables:
  - open addressing
  - chaining
- In open addressing *probing* is used to access an item in a hash table
  - If the index calculated for an item's key is occupied by an item with that key, we have found the item
  - If that element contains an item with a different key, increment the index by one
  - Keep incrementing until you find the key or a `null` entry (assuming the table is not full)

Open Addressing (cont.)



Algorithm for Accessing an Item in a Hash Table

```
1.  Compute the index by taking the item's hashCode() % table.length.
2.  if table[index] is null
3.          The item is not in the table.
4.  else if table[index] is equal to the item
5.          The item is in the table.
    else
6.          Continue to search the table by incrementing the index until either the
            item is found or a null entry is found.
```

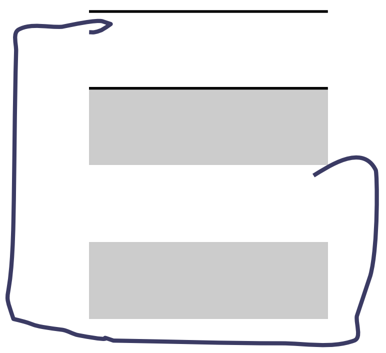
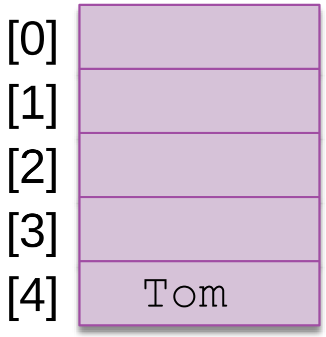


Table Wraparound and Search Termination

- As we increment the table index, it should wrap around as in a circular array
- This enables us to search the part of the table before the hash code value in addition to the part of the table after the hash code value
- But this process can be infinite!
- How do we know when to stop searching if the table is full and we have not found the correct value?
  - **Stop** when the index value for the next probe is the same as the hash code value for the object (this is where we started!)
- How to reduce the number of collisions?
  - Ensure that the table is never full by **increasing its size** after an insertion **when its load factor exceeds a specified threshold**

Hash Code Insertion Example

```
Tom  Dick  Harry  Sam Pete
```



Name	hashCode()	hashCode()%5
"Tom"	84274	4
"Dick"	2129869	4
"Harry"	69496448	3
"Sam"	82879	4
"Pete"	2484038	3

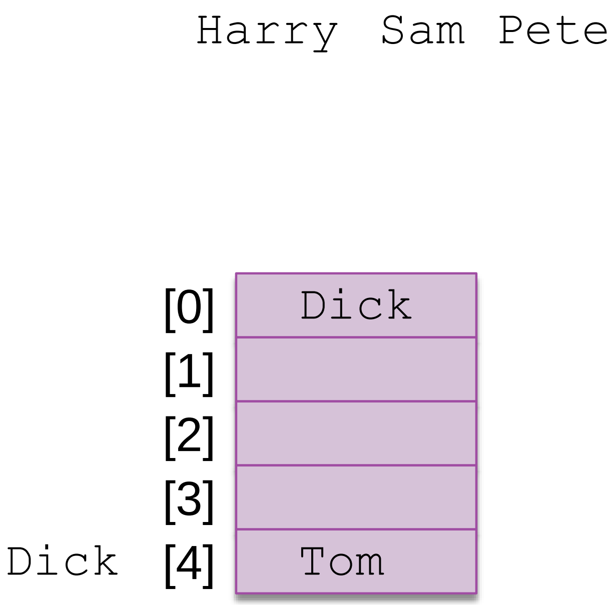
Hash Code Insertion Example (cont.)

Dick Harry Sam Pete

[0]	
[1]	
[2]	
[3]	
Dick [4]	Tom

Name	hashCode()	hashCode()%5
"Tom"	84274	4
"Dick"	2129869	4
"Harry"	69496448	3
"Sam"	82879	4
"Pete"	2484038	3

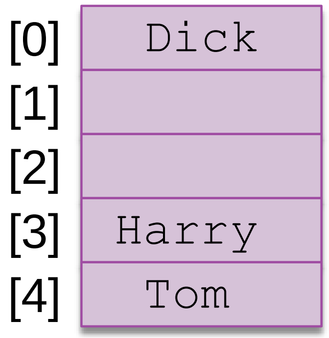
Hash Code Insertion Example (cont.)



Name	hashCode()	hashCode()%5
"Tom"	84274	4
"Dick"	2129869	4
"Harry"	69496448	3
"Sam"	82879	4
"Pete"	2484038	3

Hash Code Insertion Example (cont.)

`Harry Sam Pete`



Name	hashCode()	hashCode()%5
"Tom"	84274	4
"Dick"	2129869	4
"Harry"	69496448	3
"Sam"	82879	4
"Pete"	2484038	3

Hash Code Insertion Example (cont.)

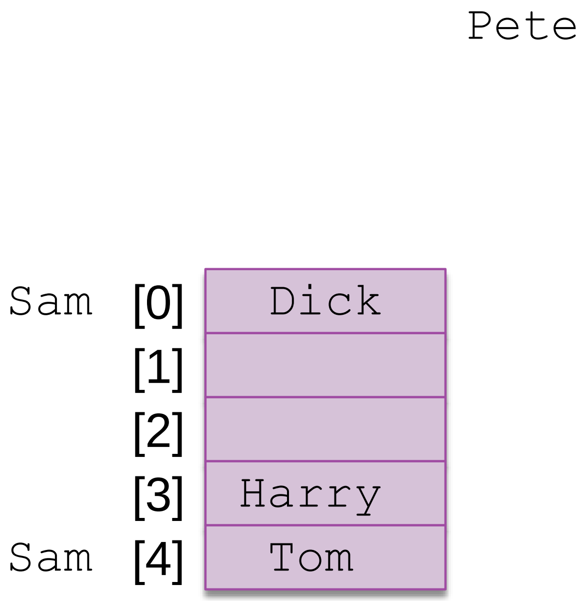
Sam Pete

[0]	Dick
[1]	
[2]	
[3]	Harry
[4]	Tom

Sam

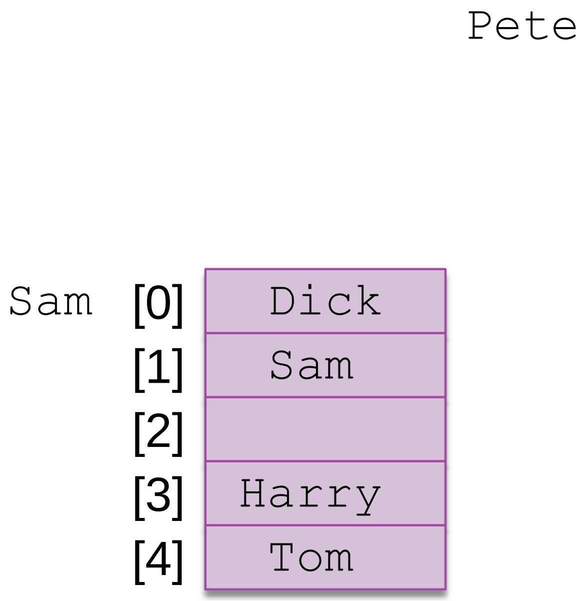
Name	hashCode()	hashCode()%5
"Tom"	84274	4
"Dick"	2129869	4
"Harry"	69496448	3
"Sam"	82879	4
"Pete"	2484038	3

Hash Code Insertion Example (cont.)



Name	hashCode()	hashCode()%5
"Tom"	84274	4
"Dick"	2129869	4
"Harry"	69496448	3
"Sam"	82879	4
"Pete"	2484038	3

Hash Code Insertion Example (cont.)



Name	hashCode()	hashCode()%5
"Tom"	84274	4
"Dick"	2129869	4
"Harry"	69496448	3
"Sam"	82879	4
"Pete"	2484038	3

Hash Code Insertion Example (cont.)

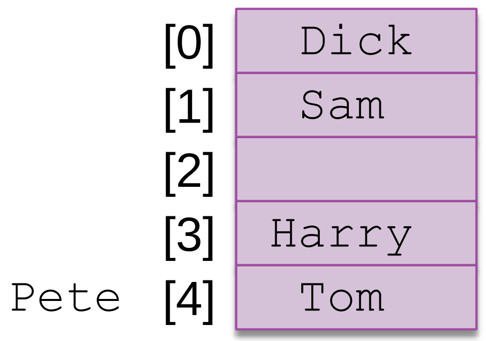
Pete

[0]	Dick
[1]	Sam
[2]	
[3]	Harry
[4]	Tom

Pete

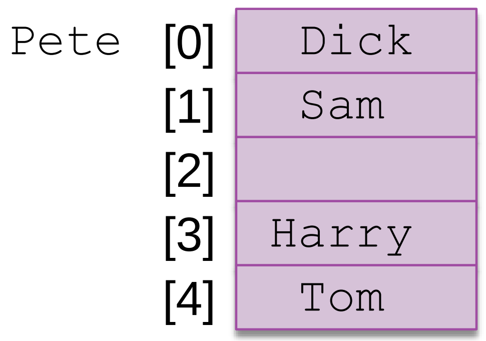
Name	hashCode()	hashCode()%5
"Tom"	84274	4
"Dick"	2129869	4
"Harry"	69496448	3
"Sam"	82879	4
"Pete"	2484038	3

Hash Code Insertion Example (cont.)



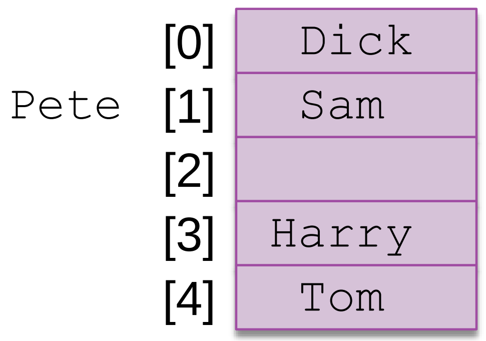
Name	hashCode()	hashCode()%5
"Tom"	84274	4
"Dick"	2129869	4
"Harry"	69496448	3
"Sam"	82879	4
"Pete"	2484038	3

Hash Code Insertion Example (cont.)



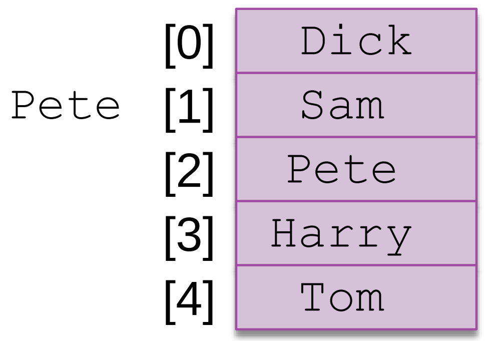
Name	hashCode()	hashCode()%5
"Tom"	84274	4
"Dick"	2129869	4
"Harry"	69496448	3
"Sam"	82879	4
"Pete"	2484038	3

Hash Code Insertion Example (cont.)



Name	hashCode()	hashCode()%5
"Tom"	84274	4
"Dick"	2129869	4
"Harry"	69496448	3
"Sam"	82879	4
"Pete"	2484038	3

Hash Code Insertion Example (cont.)



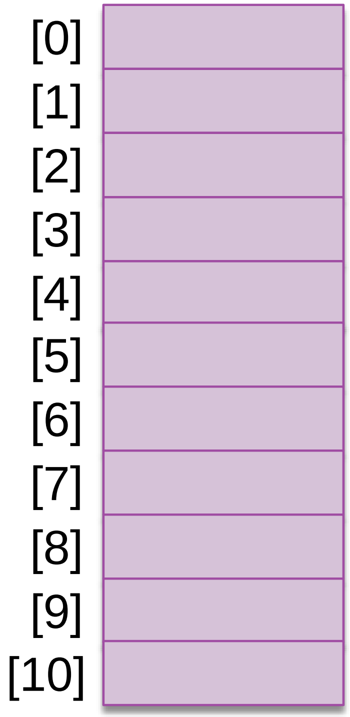
Name	hashCode()	hashCode()%5
"Tom"	84274	4
"Dick"	2129869	4
"Harry"	69496448	3
"Sam"	82879	4
"Pete"	2484038	3

Retrieval of "Tom" and "Harry" takes one step

Because of collisions, retrieval of the others ends up in a linear search

Hash Code Insertion Example (cont.)

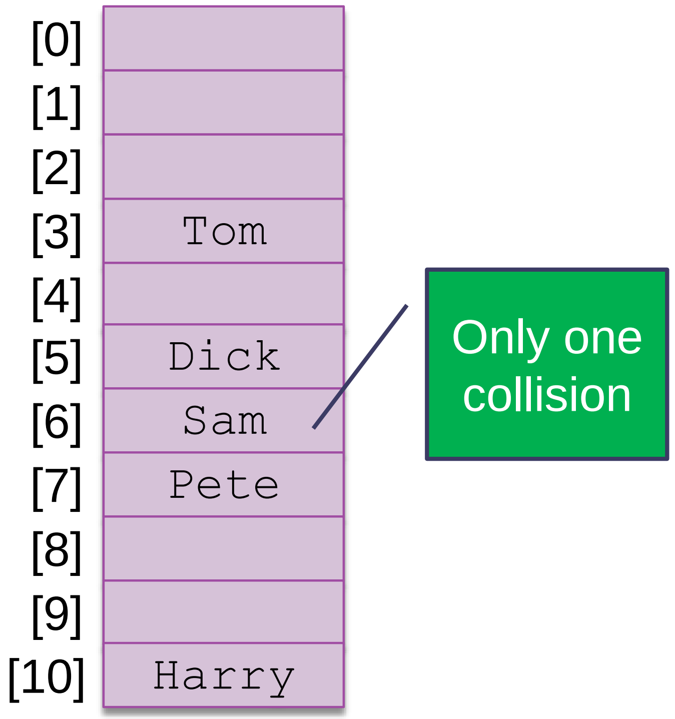
Name	hashCode()	hashCode()%11
`"Tom"`	84274	3
`"Dick"`	2129869	5
`"Harry"`	69496448	10
`"Sam"`	82879	5
`"Pete"`	2484038	7



Hash Code Insertion Example (cont.)

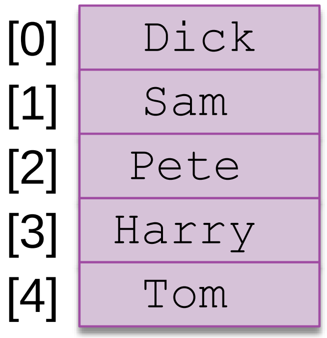
Name	hashCode()	hashCode()%11
"Tom"	84274	3
"Dick"	2129869	5
"Harry"	69496448	10
"Sam"	82879	5
"Pete"	2484038	7

A way to reduce the number of collisions (and thus reduce the retrieval time) is to increase the table size

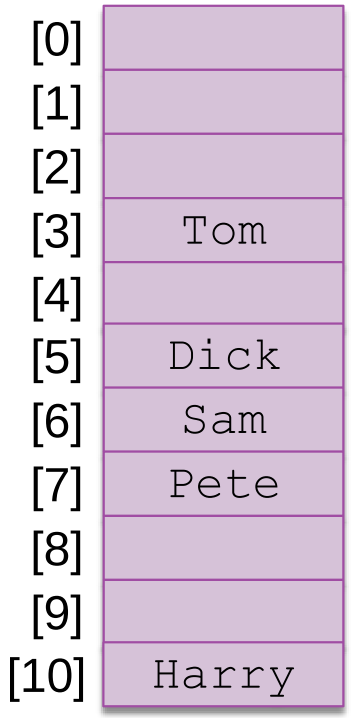


Traversing a Hash Table

- One cannot traverse a hash table in a meaningful way since the sequence of stored values is unknown



```
Dick, Sam, Pete, Harry, Tom
```



```
Tom, Dick, Sam,
Pete, Harry
```

Deleting an Item Using Open Addressing

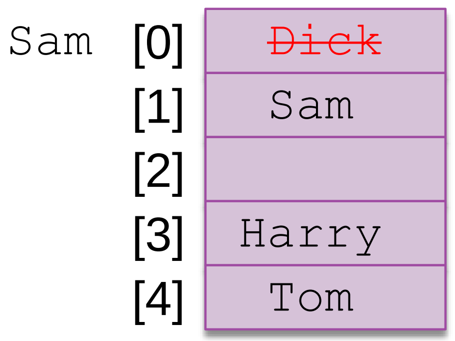
- When an item is deleted, we cannot simply set its table entry to null
- If we search for an item that may have collided with the deleted item, we may conclude incorrectly that it is not in the table.
- Instead, we store a dummy value or mark the location as available, but previously occupied
- Deleted items waste storage space and reduce search efficiency unless they are marked as available

But re-using the “deleted” space is not easy!

- We may *not* re-use the space occupied by a deleted item with a new item (when implementing a *set*) because…
- …we must go to the end of the search chain to ensure that the item is not in the table already!
- See the example!

An example

`Insert: “Sam”`



Name	hashCode()	hashCode()%5
"Tom"	84274	4
"Dick"	2129869	4
"Harry"	69496448	3
"Sam"	82879	4
"Pete"	2484038	3

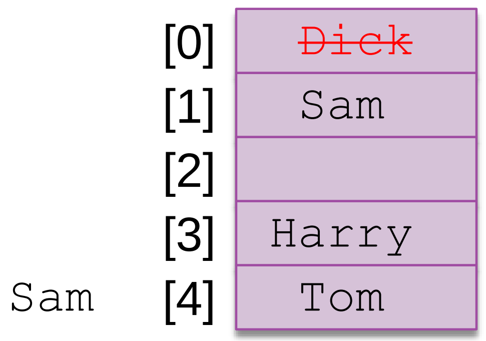
Insert “Sam” again

Sam

	[0]	~~Dick~~
	[1]	Sam
	[2]	
	[3]	Harry
Sam	[4]	Tom

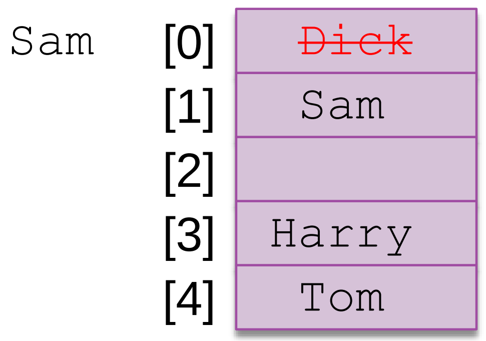
Name	hashCode()	hashCode()%5
"Tom"	84274	4
"Dick"	2129869	4
"Harry"	69496448	3
"Sam"	82879	4
"Pete"	2484038	3

So far so good



Name	hashCode()	hashCode()%5
"Tom"	84274	4
"Dick"	2129869	4
"Harry"	69496448	3
"Sam"	82879	4
"Pete"	2484038	3

Free space!



Name	hashCode()	hashCode()%5
"Tom"	84274	4
"Dick"	2129869	4
"Harry"	69496448	3
"Sam"	82879	4
"Pete"	2484038	3

Scream…

Name	hashCode()	hashCode()%5
"Tom"	84274	4
"Dick"	2129869	4
"Harry"	69496448	3
"Sam"	82879	4
"Pete"	2484038	3

[0]	Sam
[1]	Sam
[2]	
[3]	Harry
[4]	Tom

Reducing Collisions by Expanding the Table Size

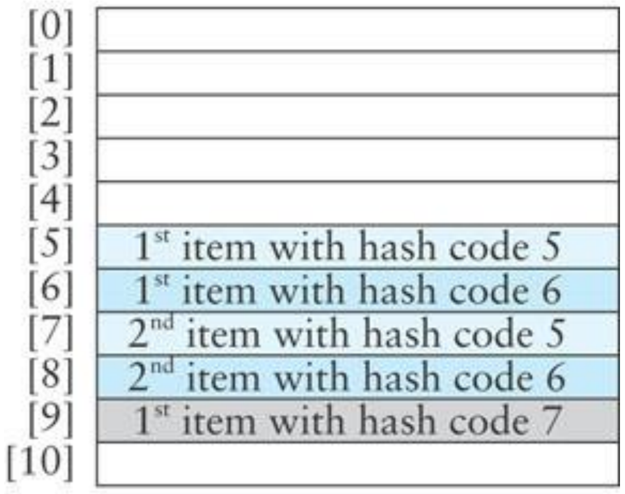
- Use a prime number for the size of the table [trust me!]
- A fuller table results in more collisions, so, when a hash table becomes sufficiently full, a larger table should be allocated and the entries reinserted
- We **must** reinsert (*rehash*) values into the new table; do not copy values as some search chains which were wrapped may break
- Deleted items are **not** reinserted, which saves space and reduces the length of some search chains

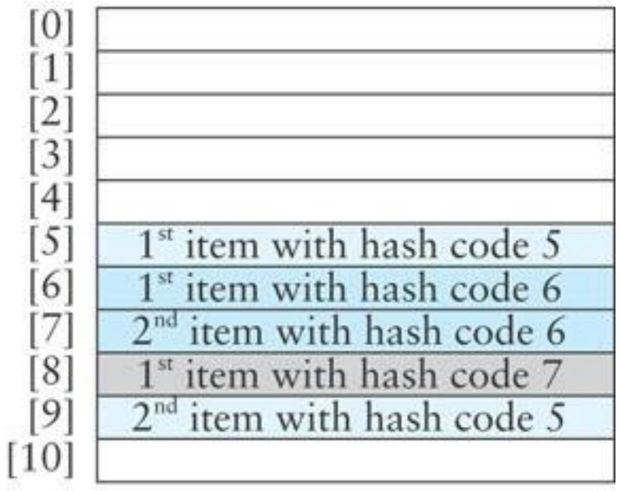
Reducing Collisions Using Quadratic Probing

- Linear probing tends to form clusters of keys in the hash table, causing longer search chains
- *Quadratic probing* can reduce the effect of clustering
  - Increments form a quadratic series ($1 + 2^2 + 3^2 + \ldots$)

```
probeNum++;
index = (startIndex + probeNum * probeNum) % table.length
```

- If an item has a hash code of 5, successive values of index are 6 (5+1), 9 (5+4), 14 (5+9), . . .


[0]
[1]
[2]
[3]
[4]
[5] 1st item with hash code 5
[6] 1st item with hash code 6
[7] 2nd item with hash code 5
[8] 2nd item with hash code 6
[9] 1st item with hash code 7
[10]


[0]
[1]
[2]
[3]
[4]
[5] 1st item with hash code 5
[6] 1st item with hash code 6
[7] 2nd item with hash code 6
[8] 1st item with hash code 7
[9] 2nd item with hash code 5
[10]

Problems with Quadratic Probing

- N.B.: A relatively easy number-theoretic result ensures that if the table is prime and the load is under 50%, then the probe sequence will succeed
- The disadvantage of quadratic probing is that the next index calculation is time-consuming, involving multiplication, addition, and modulo division
- A more efficient way to calculate the next index is:

```
k += 2;
index = (index + k) % table.length;
```

Problems with Quadratic Probing (cont.)

- Examples:
  - If the initial value of k is -1, successive values of k will be 1, 3, 5, …
  - If the initial value of index is 5, successive value of index will be 6 (= 5 + 1), 9 (= 5 + 1 + 3), 14 (= 5 + 1 + 3 + 5), …
- The proof of the equality of these two calculation:

$$n^2 = 1 + 3 + 5 + ... + 2n - 1$$

Problems with Quadratic Probing (cont.)

- A more serious problem: not all table elements are examined when looking for an insertion index; this may mean that
  - an item can't be inserted even when the table is not full
  - the program will get stuck in an infinite loop searching for an empty slot
- Again: If the table size is a prime number and it is never more than half full, this won't happen.
- …But keeping a half-empty table is a waste of memory!

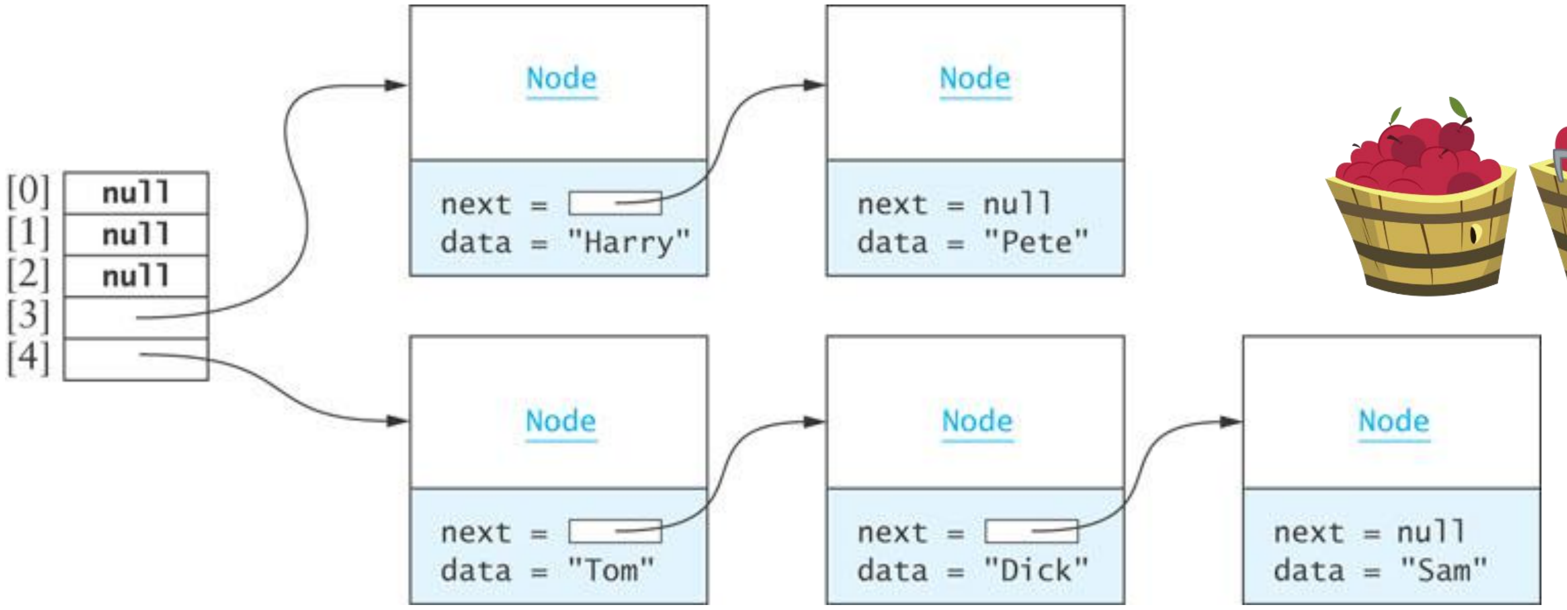
An alternative

- Remember, we talked about
  - open addressing
  - …and Chaining?

Completed

Chaining

- *Chaining* is an alternative to open addressing
- Each table element references a linked list that contains all of the items that hash to the same table index
  - The linked list often is called a *bucket*
  - The approach sometimes is called *bucket hashing*



Chaining: the advantages

- Only items that have the same value for their hash codes are examined when looking for an object
- We may be able to store more elements in the table than the the table size (!)
- Once we determine that an item is not present, we can insert it at the beginning or end of the list
- To remove an item, we simply delete it; no dummy items or marking is needed

Performance of Hash Tables

- *Load factor* is the number of filled cells divided by the table size
- The lower the load factor, the better the performance as there is a smaller chance of collision when a table is sparsely populated
- If there are no collisions, performance for search and retrieval is O(1) regardless of the table size

Performance of Open Addressing versus Chaining

- Donald E. Knuth derived the following formula for the expected number of comparisons, *c*, required for finding an item that is in a hash table using open addressing with linear probing and a load factor *L*

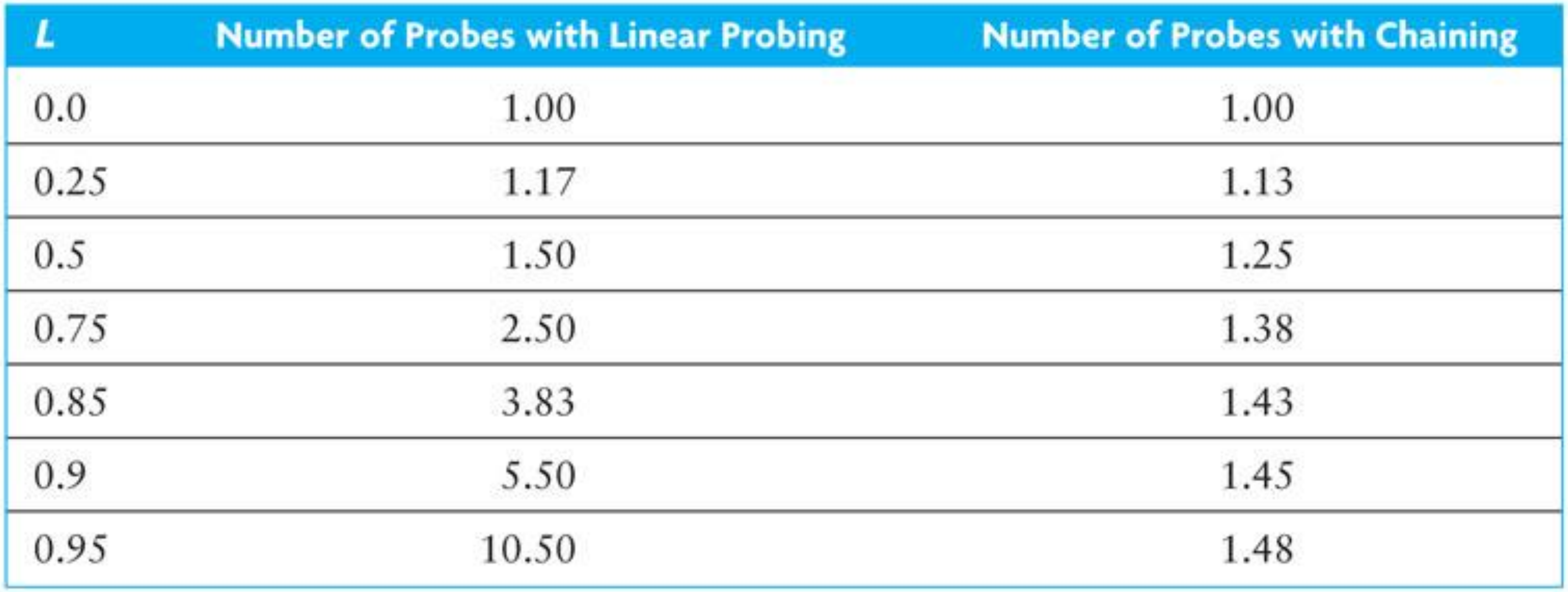
$$c = \frac{1}{2}\left(1 + \frac{1}{1 - L}\right)$$

Performance of Open Addressing versus Chaining (cont.)

- Using chaining, if an item is in the table, on average we must examine the table element corresponding to the item's hash code and then half of the items in each list
- The average number of items in a list is *L*, the number of items divided by the table size

$$c = 1 + \frac{L}{2}$$

Performance of Open Addressing versus Chaining (cont.)



L	Number of Probes with Linear Probing	Number of Probes with Chaining
0.0	1.00	1.00
0.25	1.17	1.13
0.5	1.50	1.25
0.75	2.50	1.38
0.85	3.83	1.43
0.9	5.50	1.45
0.95	10.50	1.48

Performance of Hash Tables versus Sorted Array and Binary Search Tree

- The number of comparisons required for a binary search of a sorted array is O(log *n*)
    - A sorted array of size 128 requires up to 7 probes (2^7 is 128) which is more than for a hash table of any size that is 90% full
    - A binary search tree performs similarly
- Insertion or retrieval

hash table	O(1) expected; worst case O(*n*)
sorted array	O(log *n*)
binary search tree	O(log *n*); worst case O(*n*)

Storage Requirements for Hash Tables, Sorted Arrays, and Trees

- The performance of hashing is superior to that of binary search of an array or a binary search tree, particularly if the load factor is less than 0.75
- But the lower the load factor, the more there are empty storage cells
  - there are no empty cells in a sorted array
- A binary search tree requires three references per node (item, left subtree, right subtree), so more storage is required for a binary search tree than for a hash table with load factor 0.75

Storage Requirements for Open Addressing and Chaining

- For open addressing, the number of references to items (key-value pairs) is *n* (the size of the table)
- For chaining , the average number of nodes in a list is *L* (the load factor) and *n* is the number of table elements
  - Using the Java API `LinkedList`, there will be three references in each node (item, next, previous)
  - Using our own single linked list, we can reduce the references to two by eliminating the previous-element reference
  - Therefore, storage for $n + 2L$ references is needed

Storage Requirements for Open Addressing and Chaining (cont.)

- Example:
  - Assume open addressing, 60,000 items in the hash table, and a load factor of 0.75
  - This requires a table of size 80,000 and results in an expected number of comparisons of 2.5
  - Calculating the table size *n* to get similar performance using chaining

    $2.5 = 1 + L/2$

    $5.0 = 2 + L$

    $3.0 = 60{,}000/n$

    $n = 20{,}000$

Storage Requirements for Open Addressing and Chaining (cont.)

- A hash table of size 20,000 provides storage space for 20,000 references to lists
- There are 60,000 nodes in the table (one for each item)
- This requires storage for 140,000 references (2 x 60,000 + 20,000), which is 175% of the storage needed for open addressing