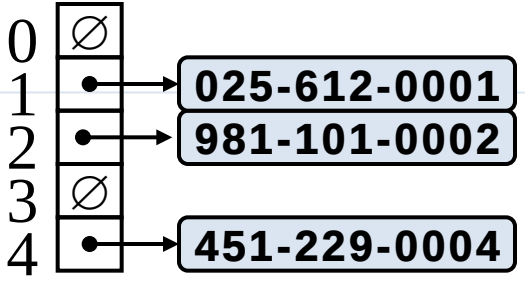
SUNY Korea

Stony Brook University

HASH TABLES: (CH10.2)



Presentation for use with the textbook

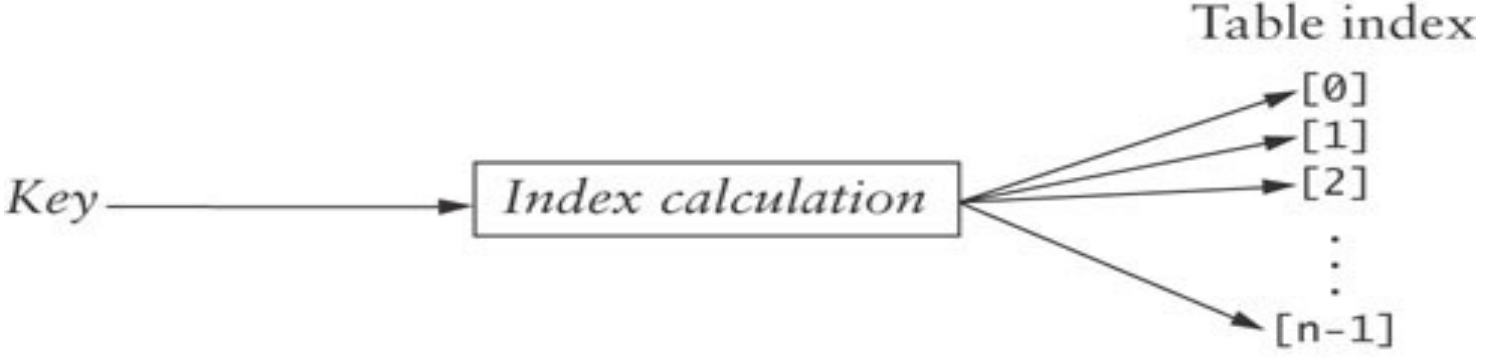
1. Data Structures and Algorithms in Java, 6th edition, by M. T. Goodrich, R. Tamassia, and M. H. Goldwasser, Wiley, 2014
2. Data Structures Abstraction and Design Using Java, 2nd Edition by Elliot B. Koffman & Paul A. T. Wolfgang, Wiley, 2010

HASH TABLES

- The goal of hash table is to be able to access an entry based on its key value, not its location
- We want to be able to access an entry directly through its key value, rather than by having to determine its location first by searching for the key value in an array
- Using a hash table enables us to retrieve an entry in constant time (on *average*, O(1))

HASH CODES AND INDEX CALCULATION

- The basis of hashing is to transform the item's key value into an integer value (its *hash code*) which is then transformed into a table index



HASH CODES AND INDEX CALCULATION (CONT.)

- However, what if all 65,536 Unicode characters were allowed?
- If you assume that on average 100 characters were used, you could use a table of 200 characters and compute the index by:

```
int index = unicode % 200
```

...	...
65	A, 8
66	B, 2
67	C, 3
68	D, 4
69	E, 12
70	F, 2
71	G, 2
72	H, 6
73	I, 7
74	J, 1
75	K, 2
...	...

HASH FUNCTIONS AND HASH TABLES

- **Hash tables** (implemented by a Map or Set) store objects at arbitrary locations and offer an average constant time for insertion, removal, and searching
- It is one of the most efficient data structures for implementing a map, and the one that is used most in practice.
- A hash function h maps keys of a given type to integers in a fixed interval $[0, N-1]$
  - A hash table for a given key type consists of
    - Hash function h
    - Array (called table) of size N
  - When implementing a map with a hash table, the goal is to store item (k, o) at index $i = h(k)$

BASE DATA STRUCTURE OF HASH TABLE

- *bucket array*:
  - each bucket may manage a collection of entries that are sent to a specific index by the hash function. (To save space, an empty bucket may be replaced by a null reference.)

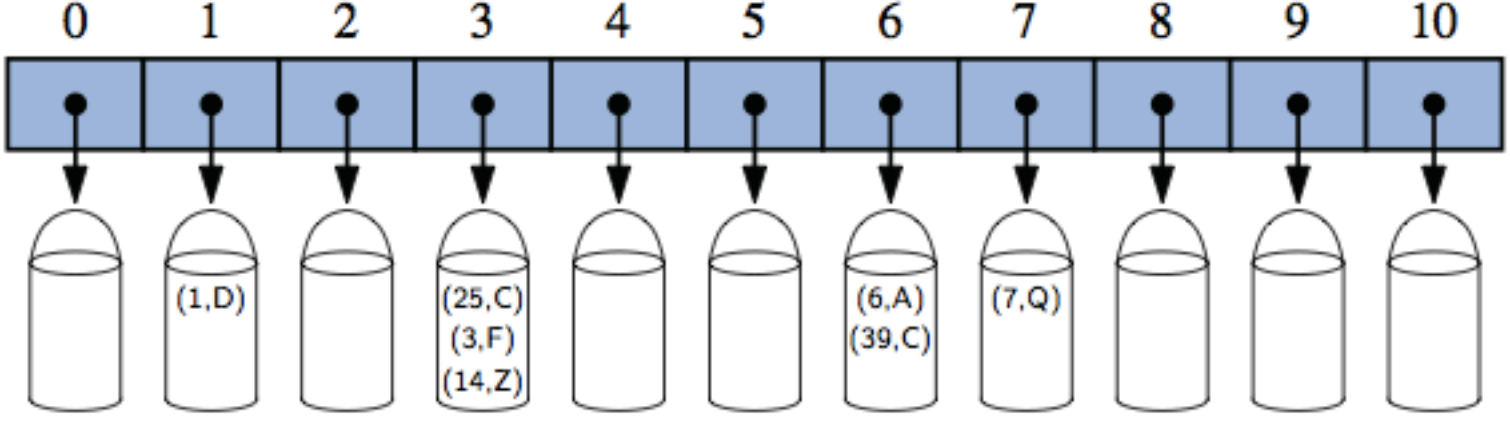
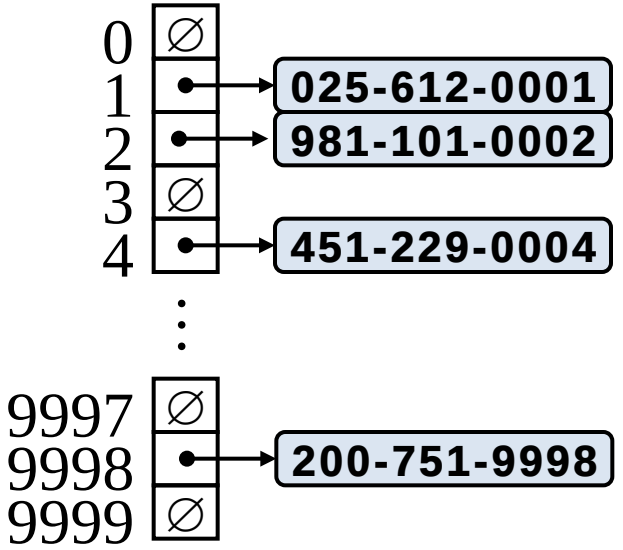


Figure 10.4: A bucket array of capacity 11 with entries (1,D), (25,C), (3,F), (14,Z), (6,A), (39,C), and (7,Q), using a simple hash function.

EXAMPLE

- design a hash table for a map storing entries as (SSN, Name), where SSN (social security number) is a nine-digit positive integer
- Our hash table uses an array of size $N = 10{,}000$ and the hash function
  $h(x) = \text{last four digits of } x$



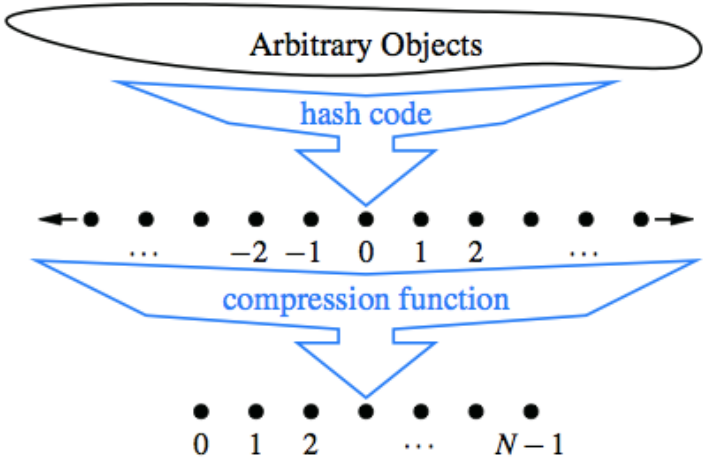
HASH FUNCTIONS

- The goal of a ***hash function***, h, is to map each key k to an integer in the range $[0, N - 1]$, where N is the capacity of the bucket array for a hash table.
- A hash function is usually specified as the composition of two functions:
  - **Hash code** (independent of hash table size – allow generic implementation):
    h_1: keys $\rightarrow$ integers
  - **Compression function** (dependent of hash table size):
    h_2: integers $\rightarrow [0, N - 1]$

HASH CODES

- The hash code is applied first, and the compression function is applied next on the result, i.e.,

$$h(x) = h_2(h_1(x))$$



- H1: Take an arbitrary key *k* in our map and compute an integer that is called the ***hash code*** for *k*
- The **hash code** need not be in the range [0, *N* − 1], and may even be negative.
- We desire that the set of hash codes assigned to our keys should avoid collisions as much as possible.

HASH CODES: BIT REPRESENTATION AS AN INTEGER

- Integer cast:
  - Java relies on 32-bit hash codes
  - We reinterpret the bits of the key as an integer
  - Suitable for keys of length less than or equal to the number of bits of the integer type (e.g., byte, short, int and float in Java)
- Component sum:
  - Partition the bits of the key into components of fixed length (e.g., 16 or 32 bits) and we sum (or exclusive-or) the components (ignoring overflows)
  - Suitable for numeric keys of fixed length greater than or equal to the number of bits of the integer type (e.g., long and & double in Java)

Both not good for character strings or other variable-length objects that can be viewed as tuples of the form $(x_0,x_1,\ldots,x_{n-1})$, where the order of the x_i's is significant. Ex> "stop", "tops", "pots", and "spot".

POLYNOMIAL HASH CODES

A **polynomial hash code** takes into consideration the positions of the x_i's by using multiplication by different powers as a way to spread out the influence of each component across the resulting hash code.

- Polynomial accumulation:
  - Partition the bits of the key into a sequence of components of fixed length (e.g., 8, 16 or 32 bits)
  - Evaluate the polynomial

$$x_0a^{n-1} + x_1a^{n-2} + \cdots + x_{n-2}a + x_{n-1}.$$

    where a!=1 is a nonzero constant, ignoring overflows
- Especially suitable for strings
  - (33, 37, 39, and 41 are particularly good choices for *a* when working with character strings that are English words.)
- Polynomial $p(a)$ can be evaluated in $O(n)$ time using Horner's rule:
  - The following polynomials are successively computed, each from the previous one in $O(1)$ time

    $p_0(a) = x_{n-1}$

    $p_i(a) = x_{n-i-1} + ax_{i-1}(a)$ $(i = 1, 2, \ldots, n-1)$
- We have $p(a) = p_{n-1}(a)$

$$x_{n-1} + a(x_{n-2} + a(x_{n-3} + \cdots + a(x_2 + a(x_1 + ax_0)) \cdots)).$$

CYCLIC-SHIFT HASH CODES

- A variant of the polynomial hash code replaces multiplication by *a* with a cyclic shift of a partial sum by a certain number of bits.
  - Ex.> 5-bit cyclic shift of the 32-bit

00111101100101101010100010101000 is achieved by taking the leftmost five bits and placing those on the rightmost side of the representation, resulting in 10110010110101010001010100000111.

```
static int hashCode(String s) {
  int h=0;
  for (int i=0; i<s.length(); i++) {
    h = (h << 5) | (h >>> 27);          // 5-bit cyclic shift of the running sum
    h += (int) s.charAt(i);             // add in next character
  }
  return h;
}
```

Shift	Collisions	
	Total	Max
0	234735	623
1	165076	43
2	38471	13
3	7174	5
4	1379	3
5	190	3
6	502	2
7	560	2
8	5546	4
9	393	3
10	5194	5
11	11559	5
12	822	2
13	900	4
14	2001	4
15	19251	8
16	211781	37

COMPRESSION FUNCTIONS

- **Compression function** maps integer hash code i into an integer in the range of [0, N-1]

- A good compression function: probability any two different keys collide is 1/*N*.
  - If a hash function is chosen well, it should ensure that the probability of two different keys getting hashed to the same bucket is 1/*N*.

- Methods:
  - Multiplication method
  - MAD method

COMPRESSION FUNCTION: DIVISION METHOD

- Function:

  $h_2(i) = i \bmod N$ (i: the hash code)
- The size N of the hash table is usually chosen to be a prime
  - Prime numbers are shown to helps “spread out” the distribution of hashed values.
  - Example: if we insert keys with hash codes {200,205,210,215,220,…,600} into a bucket array of size 100, then each hash code will collide with three others. But if we use a bucket array of size 101, then there will be no collisions.
  - The reason has to do with number theory and is beyond the scope of this course
- Choosing N to be a prime number is not always enough
  - If there is a repeated pattern of hash codes of the form $pN + q$ for several different *prime numbers, p*, then there will still be collisions.

COMPRESSION FUNCTION: MAD METHOD

- *Multiply-Add-and-Divide* (MAD) Method:

$$h_2(i) = [(ai + b) \bmod p] \bmod N$$

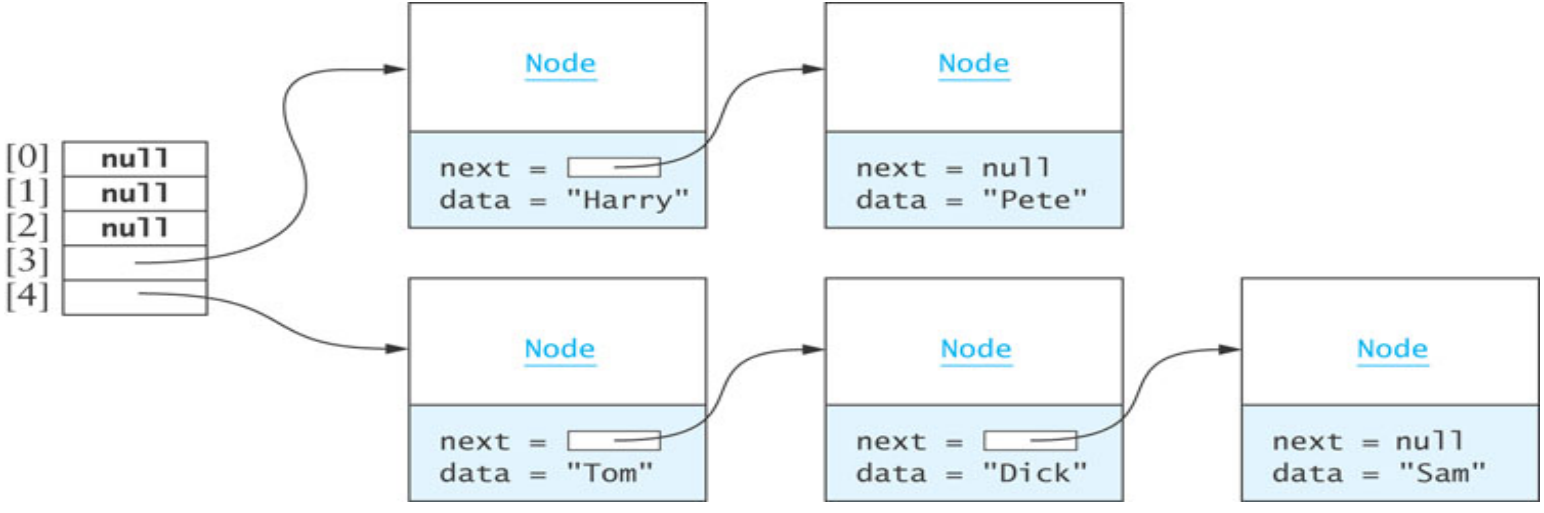
  - where N is the size of the hash table, p is a prime number larger than N, and a and b are integers chosen at random from the interval $[0, p - 1]$, with $a > 0$.
- MAD is chosen in order to eliminate repeated patterns in the set of hash codes and get us closer to having a "good" hash function

COLLISION-HANDLING SCHEMES

- The main idea of a hash table is to take a bucket array, A, and a hash function, h, and use them to implement a map by storing each entry (k,v) in the "bucket" $A[h(k)]$.
- Even with a good hash function, collisions happen, i.e., two distinct keys, k_1 and k_2, such that $h(k_1) = h(k_2)$.
- Collisions
  - Prevents us from simply inserting a new entry (k,v) directly into the bucket $A[h(k)]$
  - Complicates our procedure for insertion, search, and deletion operations.
- Collision handling schemes:
  - Separate Chaining
  - Open Addressing
    - Linear Probing and Variants of Linear Probing

CHAINING

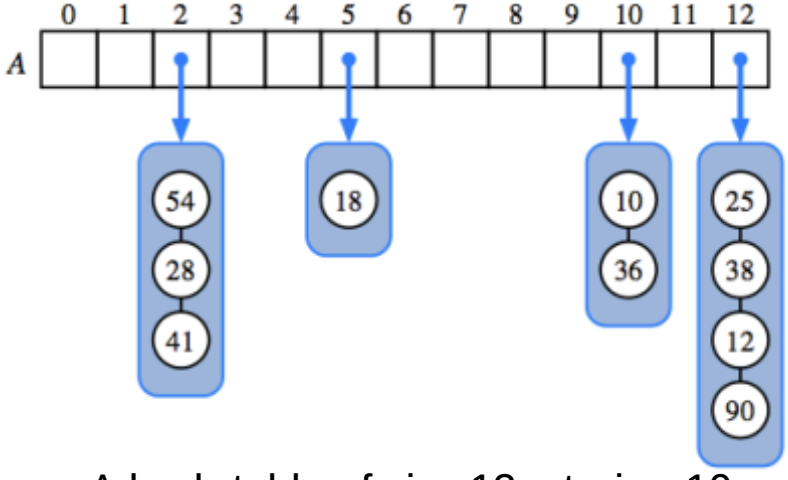
- Each table element references a linked list that contain s all of the items that hash to the same table index
  - The linked list often is called a *bucket*
  - The approach sometimes is called *bucket hashing*



COLLISION-HANDLING SCHEMES: SEPARATE CHAINING

- Separate Chaining Scheme: have each bucket $A[j]$ store its own secondary container, holding all entries (k,v) such that $h(k) = j$.

- Advantage: simple implementations of map operations
- Disadvantage: requires the use of an auxiliary data structure to hold entries with colliding keys



A hash table of size 13, storing 10 entries with integer keys, with collisions resolved by separate chaining. The compression function is $h(k) = k \bmod 13$. Values omitted.

COLLISION-HANDLING SCHEMES: OPEN ADDRESSING

- Open Addressing: store each entry directly in a table slot.
  - This approach saves space because no auxiliary structures are employed
  - Requires a bit more complexity to properly handle collisions.
- Open addressing requires
  - Load factor is always at most 1
  - Entries are stored directly in the cells of the bucket array itself.

- EX> Linear Probing and Its Variants

LOAD FACTOR

- Assuming we use a good hash function to index the n entries of our map in a bucket array of capacity N, the expected size of a bucket is n/N. Therefore, if given a good hash function, the core map operations run in $O(\lceil n/N \rceil)$. The ratio $\lambda = n/N$, called the ***load factor*** of the hash table, should be bounded by a small constant, preferably below 1. As long as λ is $O(1)$, the core operations on the hash table run in $O(1)$ expected time.

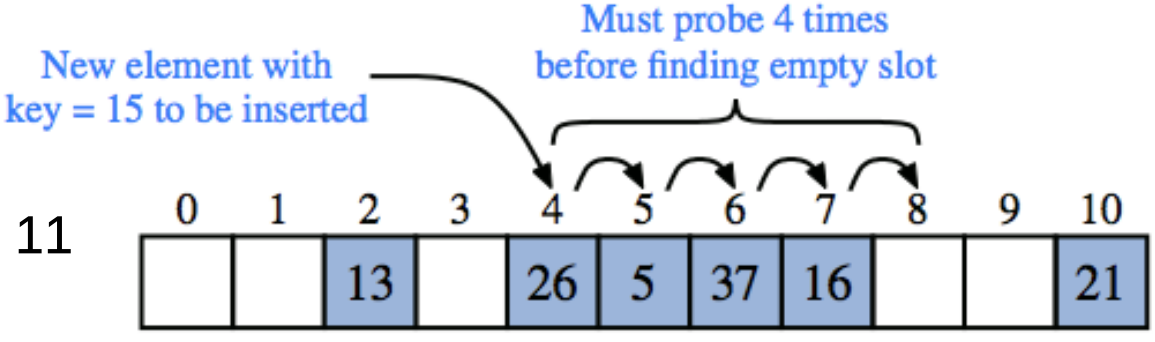
COLLISION-HANDLING SCHEMES: LINEAR PROBING

- Insertion scheme:
  - If we try to insert an entry (k, v) into a bucket $A[j]$ where $j = h(k)$
  - If $A[j]$ is already occupied, then we next try $A[(j+1) \bmod N]$.
  - If $A[(j+1) \bmod N]$ is also occupied, then we try $A[(j + 2) \bmod N]$,
  - and so on, until we find an empty bucket that can accept the new entry.

COLLISION-HANDLING SCHEMES: LINEAR PROBING (CONT.)

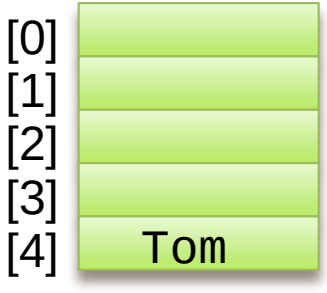
- Implementation when searching for an existing key, the first step of all get, put, or remove operations, need modification.
- Search Scheme: Starting from $A[h(k)]$, examine consecutive slots, until either
  - An entry with an equal key is found or
  - An empty bucket is found.

$H(k) = k \bmod 11$



HASH CODE INSERTION EXAMPLE

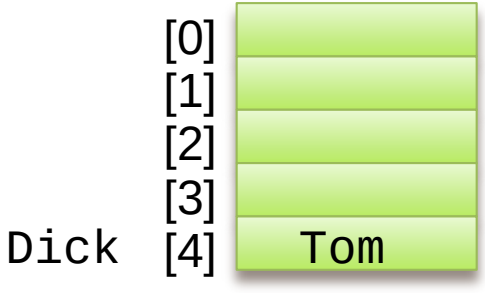
```
Tom    Dick    Harry    Sam    Pete
```



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

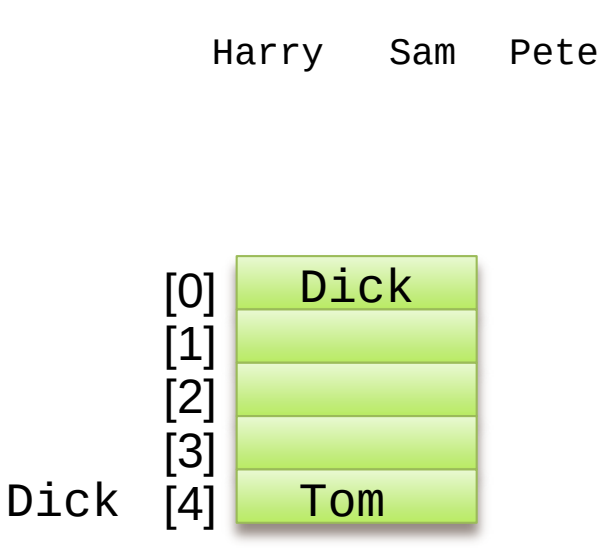
HASH CODE INSERTION EXAMPLE (CONT.)

Dick Harry Sam Pete



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

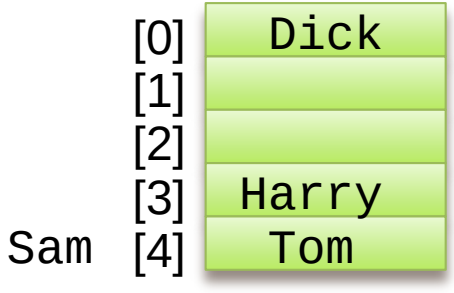
HASH CODE INSERTION EXAMPLE (CONT.)

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.)

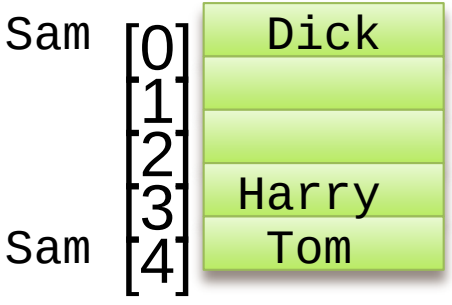
Sam Pete



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



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

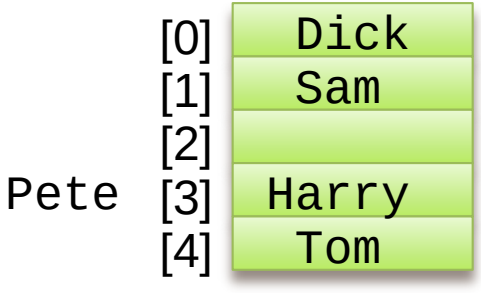
HASH CODE INSERTION EXAMPLE (CONT.)

Pete

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

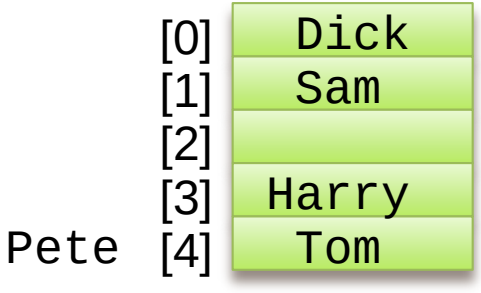
HASH CODE INSERTION EXAMPLE (CONT.)

Pete



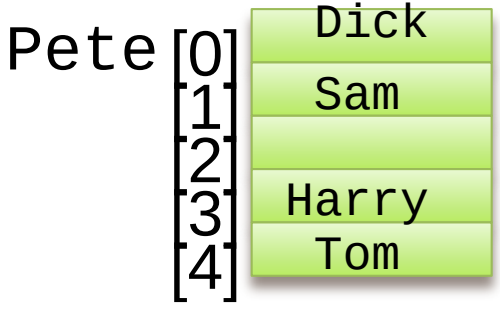
Name	hashCode()	hashCode()%5
"Tom"	84274	4
"Dick"	2129869	4
"Harry"	69496448	3
"Sam"	82879	4
"Pete"	2484038	3

HASH CODE INSERTION EXAMPLE (CONT.)



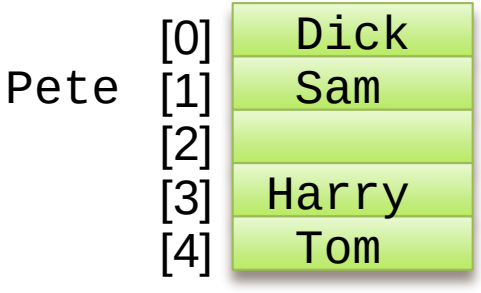
Name	hashCode()	hashCode()%5
"Tom"	84274	4
"Dick"	2129869	4
"Harry"	69496448	3
"Sam"	82879	4
"Pete"	2484038	3

HASH CODE INSERTION EXAMPLE (CONT.)



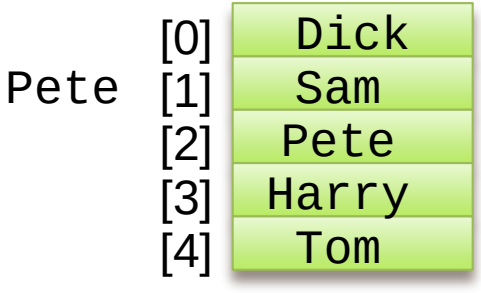
Name	hashCode()	hashCode()%5
"Tom"	84274	4
"Dick"	2129869	4
"Harry"	69496448	3
"Sam"	82879	4
"Pete"	2484038	3

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.)



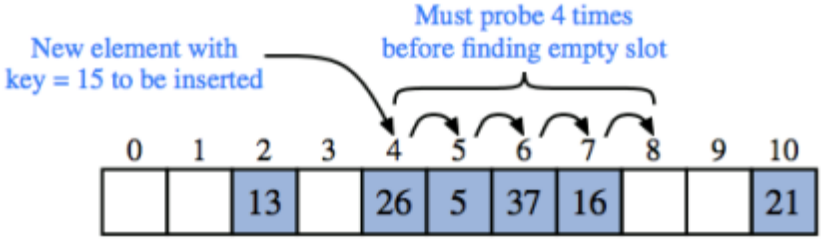
Name	hashCode()	hashCode()%5
"Tom"	84274	4
"Dick"	2129869	4
"Harry"	69496448	3
"Sam"	82879	4
"Pete"	2484038	3

Retrieval of "Tom" or "Harry" takes one step, O(1)

Because of collisions, retrieval of the others requires a linear search

COLLISION-HANDLING SCHEMES: LINEAR PROBING

- Deletion Scheme:
  - Cannot simply remove a found entry from its slot in the array
  - EX> after the insertion of key 15, if the entry with key 37 were trivially deleted, a subsequent search for 15 would fail because that search would start by probing at index 4, then index 5, and then index 6, at which an empty cell is found.



  - Resolve by replacing a deleted entry with a special “defunct” sentinel object.
    - Modify search algorithm so that the search for a key *k* will skip over cells containing the defunct
    - The put should remember a defunct locations during the search for *k*, and put the new entry (*k*,*v*), if no existing entry is found beyond it.

COLLISION-HANDLING SCHEMES: VARIANTS OF LINEAR PROBING

- Linear probing tends to cluster the entries of a map into contiguous runs, which may even overlap causing searches to slow down considerably.
- Avoiding Clustering with variant of Linear Probing:
  - **Quadratic Probing:** iteratively tries the buckets $A[(h(k)+ f(i)) \bmod N]$, for $i = 0,1,2,\ldots$, where $f(i) = i^2$, until finding an empty bucket.
  - **Double Hashing**: choose a secondary hash function, h', and if h maps some key k to a bucket $A[h(k)]$ that is already occupied (no clustering effect)

PROBLEMS WITH QUADRATIC PROBING

- Quadratic probing strategy complicates the removal operation.
- It does avoid the kinds of clustering patterns that occur with linear probing but still suffers from secondary clustering
  - **Secondary Clustering:** set of filled array cells still has a nonuniform pattern, even if we assume that the original hash codes are distributed uniformly.
- Calculation of next index $((h(k)+ f(i)) \bmod N)$ is time-consuming, involving multiplication, addition, and modulo division

PROBLEMS WITH QUADRATIC PROBING (CONT.)

- A more efficient way to calculate the next index (($h(k)+ f(i)$) mod N) is:

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

- Examples:
  - If the initial value of i is -1, successive values of i 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 methods is based on the mathematical series:

$$n^2 = 1 + 3 + 5 + \ldots + 2n - 1$$

PROBLEMS WITH QUADRATIC PROBING (CONT.)

- A more serious problem is that 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
- If the table size is a prime number and it is never more than half full, this won't happen
- However, requiring a half empty table wastes a lot of memory

DOUBLE HASHING

- Open addressing strategy that does not cause clustering of the kind produced by linear probing or the kind produced by quadratic probing
- Double hashing uses a secondary hash function $h'(k)$ and handles collisions by placing an item in the first available cell of the series

$$(h(k) + i*h'(k)) \bmod N$$

for $i = 0, 1, \ldots, N - 1$

- The table size N must be a prime to allow probing of all the cells
- The secondary hash function $h'(k)$ cannot have zero values
- Common choice of compression function for the secondary hash function:

$$h'(k) = q - (k \bmod q)$$

for some prime $q < N$.

- The possible values for $h'(k)$ are

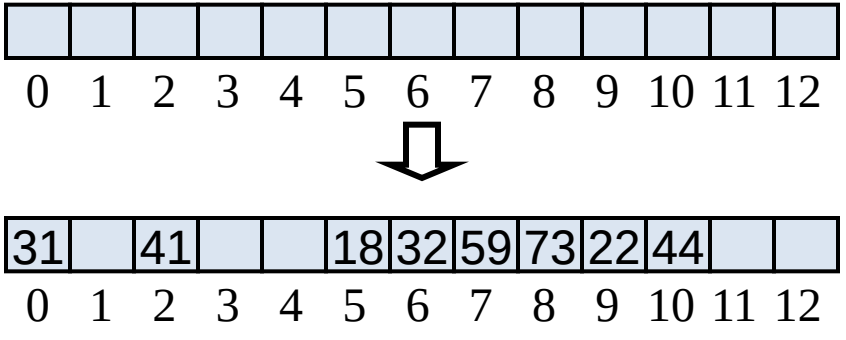
$$1, 2, \ldots, q$$

EXAMPLE OF DOUBLE HASHING

$(h(k) + i*h'(k)) \bmod N$

- Consider a hash table storing integer keys that handles collision with double hashing
  - $N = 13$
  - $h(k) = k \bmod 13$
  - $h'(k) = 7 - k \bmod 7$
- Insert keys 18, 41, 22, 44, 59, 32, 31, 73, in this order

k	$h(k)$	$h'(k)$	Probes		
18	5	3	5		
41	2	1	2		
22	9	6	9		
44	5	5	5	10	
59	7	4	7		
32	6	3	6		
31	5	4	5	9	0
73	8	4	8		



LOAD FACTOR AND EFFICIENCY

In the hash table schemes described thus far, it is important that the load factor, $\lambda = n/N$, be kept below 1.

- Ex> Separate chaining:
  - As λ gets very close to 1, the probability of a collision greatly increases
  - Collisions adds overhead to operations
    - Rely on linear-time list-based methods in buckets that have collision s
  - Recommended load factor is $\lambda < 0.9$ for hash tables with se parate chaining. (Java: $\lambda < 0.75$.)

LOAD FACTOR AND EFFICIENCY (CONT)

- Open addressing:
  - If the load factor λ gets higher than 0.5, clusters of entries in the bucket array start to grow as well.
  - These clusters cause the probing strategies to “bounce arou nd” the bucket array for a considerable amount of time befo re they find an empty slot.
  - Suggested load factor is $\lambda < 0.5$ for an open addressing sch eme with linear probing, and perhaps only a bit higher for ot her open addressing schemes.

REHASHING AND EFFICIENCY

- If an insertion causes the load factor of a hash table to go above the specified threshold
  - resize the table (to regain the specified load factor) and reinsert a ll objects into this new table (Rehashing).
- Rehashing:
  - Rehashing scatter the entries throughout the new bucket array.
  - Don’t need a new hash code.
  - Do need a new compression function
    - Takes into consideration the size of the new table.
    - It is a good requirement for the new array’s size to be a prime number a pproximately double the previous size (amortized analysis)

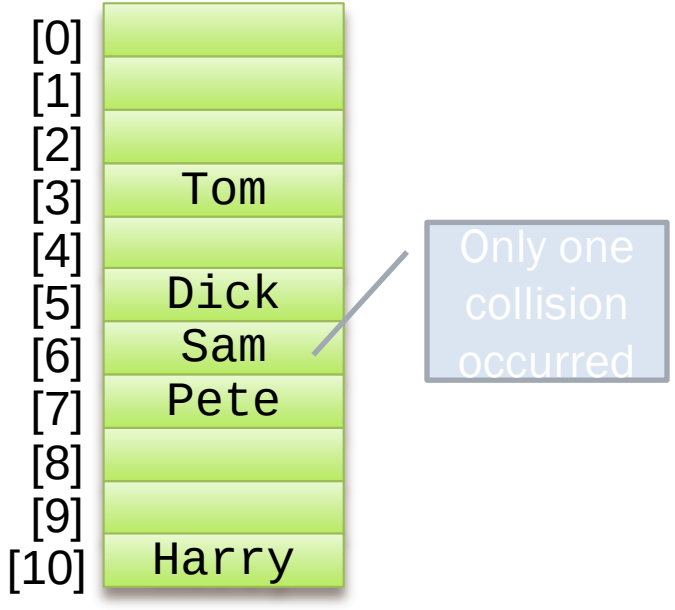
HASH CODE INSERTION EXAMPLE (CONT.)

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

The best way to reduce the possibility of collision (and reduce linear search retrieval time because of collisions) is to increase the table size



TRAVERSING A HASH TABLE

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

Dick, Sam, Pete, Harry, Tom

Tom, Dick, Sam, Pete, Harry

EFFICIENCY OF HASH TABLES

- Probabilistic basis of average-case analysis.
  - If our hash function is good, then we expect the entries to be unif ormly distributed in the N cells of the bucket array.
  - Thus, to store n entries, the expected number of keys in a bucket would be $\lceil n/N \rceil$, which is $O(1)$ if n is $O(N)$.
  - The costs associated with a periodic rehashing (needed after put or remove) can be accounted for separately, leading to an additio nal $O(1)$ amortized cost for put and remove.
- In the worst case, a poor hash function could map every e ntry to the same bucket.
  - Linear-time performance for the core operations

COMPARISON OF THE RUNNING TIMES

Method	Unsorted List	Hash Table	
		expected	worst case
get	$O(n)$	$O(1)$	$O(n)$
put	$O(n)$	$O(1)$	$O(n)$
remove	$O(n)$	$O(1)$	$O(n)$
size, isEmpty	$O(1)$	$O(1)$	$O(1)$
entrySet, keySet, values	$O(n)$	$O(n)$	$O(n)$

- Comparison of the running times of the methods of a map realized by means of an unsorted list or a hash table

JAVA HASH TABLE IMPLEMENTATION

RECALL THE MAP ADT

- size(), isEmpty()
- get(k): if the map M has an entry with key k, return its associated value; else, return null
- put(k, v): insert entry (k, v) into the map M; if key k is not already in M, then return null; else, return old value associated with k
- remove(k): if the map M has an entry with key k, remove it from M and return its associated value; else, return null
- entrySet(): return an iterable collection of the entries in M
- keySet(): return an iterable collection of the keys in M
- values(): return an iterator of the values in M

ABSTRACTHASHMAP

- It does not provide any concrete representation of a table of "buckets."
  - Separate chaining: each bucket will be a secondary map.
  - Open addressing: there is no tangible container for each bucket; the "buckets" are effectively interleaved due to the probing sequences.

ABSTRACTHASHMAP CLASS: ABSTRACT METHODS

Implementation depends on the collision handling schemes.

createTable( ): This method should create an initially empty table having size equal to a designated capacity instance variable.

bucketGet(h, k): This method should mimic the semantics of the public get method, but for a key k that is known to hash to bucket h.

bucketPut(h, k, v): This method should mimic the semantics of the public put method, but for a key k that is known to hash to bucket h.

bucketRemove(h, k): This method should mimic the semantics of the public remove method, but for a key k known to hash to bucket h.

entrySet( ): This standard map method iterates through *all* entries of the map. We do not delegate this on a per-bucket basis because "buckets" in open addressing are not inherently disjoint.

ABSTRACTHASHMAP CLASS: CONCRETE METHODS

- AbstractHashMap class provides:
  - hashValue(K key): Mathematical support in the form of a **hash compression function** using a randomized Multiply-Add-and- Divide (MAD) formula,
  - Resize(int newCap): Support for automatically resizing the underlying hash table when the load factor reaches a certain threshold.

ABSTRACT HASH MAP IN JAVA

```
1  public abstract class AbstractHashMap<K,V> extends AbstractMap<K,V> {
2    protected int n = 0;                    // number of entries in the dictionary
3    protected int capacity;                 // length of the table
4    private int prime;                      // prime factor
5    private long scale, shift;              // the shift and scaling factors
6    public AbstractHashMap(int cap, int p) {
7      prime = p;
8      capacity = cap;
9      Random rand = new Random();
10     scale = rand.nextInt(prime−1) + 1;
11     shift = rand.nextInt(prime);
12     createTable();
13   }
14   public AbstractHashMap(int cap) { this(cap, 109345121); }   // default prime
15   public AbstractHashMap() { this(17); }                      // default capacity
16   // public methods
17   public int size() { return n; }
18   public V get(K key) { return bucketGet(hashValue(key), key); }
19   public V remove(K key) { return bucketRemove(hashValue(key), key); }
20   public V put(K key, V value) {
21     V answer = bucketPut(hashValue(key), key, value);
22     if (n > capacity / 2)                 // keep load factor <= 0.5
23       resize(2 ∗ capacity − 1);           // (or find a nearby prime)
24     return answer;
25   }
```

$$MAD:\ h_2(i) = [(ai + b) \bmod p] \bmod N$$

a: scale b: shift p: prime N: size_of_hash

ABSTRACT HASH MAP IN JAVA, 2

$$MAD:\ h_2(i) = [(ai + b) \bmod p] \bmod N$$

a: scale b: shift p: prime N: size_of_hash

```
26    // private utilities
27    private int hashValue(K key) {
28      return (int) ((Math.abs(key.hashCode()*scale + shift) % prime) % capacity);
29    }
30    private void resize(int newCap) {
31      ArrayList<Entry<K,V>> buffer = new ArrayList<>(n);
32      for (Entry<K,V> e : entrySet())
33        buffer.add(e);
34      capacity = newCap;
35      createTable();                    // based on updated capacity
36      n = 0;                            // will be recomputed while reinserting entries
37      for (Entry<K,V> e : buffer)
38        put(e.getKey(), e.getValue());
39    }
40    // protected abstract methods to be implemented by subclasses
41    protected abstract void createTable();
42    protected abstract V bucketGet(int h, K k);
43    protected abstract V bucketPut(int h, K k, V v);
44    protected abstract V bucketRemove(int h, K k);
45  }
```

MAP WITH SEPARATE CHAINING

- To represent each bucket for separate chaining, we use an instance of the simpler UnsortedTableMap class.
- Entire hash table is then represented as a fixed-capacity array *A* of thevsecondary maps.
- Each cell, *A*[*h*], is initially a null reference;
- We only create a secondary map when an entry is first hashed to a particular bucket.

MAP WITH SEPARATE CHAINING

Delegate operations to a list-based map at each cell of unsor ted Map A[]:

```
Algorithm get(k):
return A[h(k)].get(k)

Algorithm put(k,v):
t = A[h(k)].put(k,v)
if t = null then                {k is a new key}
    n = n + 1
return t

Algorithm remove(k):
t = A[h(k)].remove(k)
if t ≠ null then                {k was found}
    n = n - 1
return t
```

MAP WITH SEPARATE CHAINING

- Because we choose to leave table cells as null until a secondary map is needed, each of these fundamental operations must begin by checking to see if $A[h]$ is null.
- In the case of bucketPut, a new entry must be inserted, so we instantiate a new UnsortedTableMap for $A[h]$ before continuing
- In our AbstractHashMap framework, the subclass has the responsibility to properly maintain the instance variable n when an entry is newly inserted or deleted.
  - In our implementation, we determine the change in the overall size of the map, by determining if there is any change in the size of the relevant secondary map before and after an operation.

HASH TABLE WITH CHAINING:CHAINHASHMAP

```
1  public class ChainHashMap<K,V> extends AbstractHashMap<K,V> {
2    // a fixed capacity array of UnsortedTableMap that serve as buckets
3    private UnsortedTableMap<K,V>[ ] table;   // initialized within createTable
4    public ChainHashMap( ) { super( ); }
5    public ChainHashMap(int cap) { super(cap); }
6    public ChainHashMap(int cap, int p) { super(cap, p); }
7    /** Creates an empty table having length equal to current capacity. */
8    protected void createTable( ) {
9      table = (UnsortedTableMap<K,V>[ ]) new UnsortedTableMap[capacity];
10   }
11   /** Returns value associated with key k in bucket with hash value h, or else null. */
12   protected V bucketGet(int h, K k) {
13     UnsortedTableMap<K,V> bucket = table[h];
14     if (bucket == null) return null;
15     return bucket.get(k);
16   }
17   /** Associates key k with value v in bucket with hash value h; returns old value. */
18   protected V bucketPut(int h, K k, V v) {
19     UnsortedTableMap<K,V> bucket = table[h];
20     if (bucket == null)
21       bucket = table[h] = new UnsortedTableMap<>( );
22     int oldSize = bucket.size( );
23     V answer = bucket.put(k,v);
24     n += (bucket.size( ) − oldSize);      // size may have increased
25     return answer;
26   }
```

HASH TABLE WITH CHAINING, 2: CHAINHASHMAP

```
27    /** Removes entry having key k from bucket with hash value h (if any). */
28    protected V bucketRemove(int h, K k) {
29      UnsortedTableMap<K,V> bucket = table[h];
30      if (bucket == null) return null;
31      int oldSize = bucket.size( );
32      V answer = bucket.remove(k);
33      n -= (oldSize - bucket.size( ));      // size may have decreased
34      return answer;
35    }
36    /** Returns an iterable collection of all key-value entries of the map. */
37    public Iterable<Entry<K,V>> entrySet( ) {
38      ArrayList<Entry<K,V>> buffer = new ArrayList<>( );
39      for (int h=0; h < capacity; h++)
40        if (table[h] != null)
41          for (Entry<K,V> entry : table[h].entrySet( ))
42            buffer.add(entry);
43      return buffer;
44    }
45  }
```

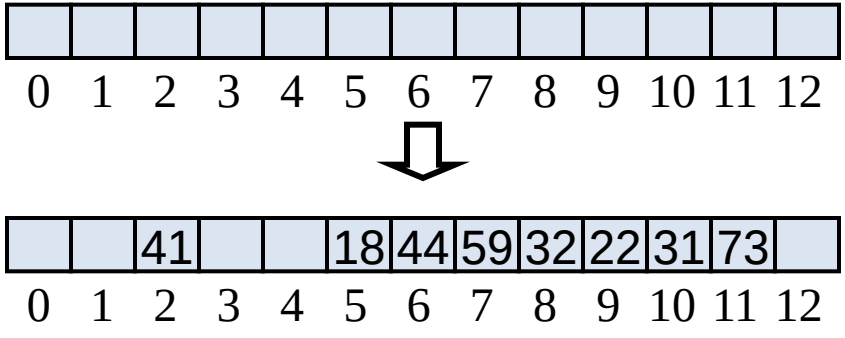
MAP WITH SEPARATE CHAINING: ANALYSIS

- Assuming load factor of n/N, each bucket has expected $O(1)$ size, provided that n is $O(N)$,
- The expected running time of operations get, put, and remove for this map is $O(1)$.
- The entrySet method (and thus the related keySet and values) runs in $O(n+N)$ time, as it loops through the length of the table (with length N) and through all buckets (which have cumulative lengths n).

LINEAR PROBING

- Open addressing: the colliding item is placed in a different cell of the table
- Linear probing: handles collisions by placing the colliding item in the next (circularly) available table cell
- Each table cell inspected is referred to as a "probe"
- Colliding items lump together, causing future collisions to cause a longer sequence of probes

- Example:
  - $h(x) = x \bmod 13$
  - Insert keys 18, 41, 22, 44, 59, 32, 31, 73, in this order



SEARCH WITH LINEAR PROBING

- Consider a hash table A that uses linear probing
- get(k)
  - We start at cell $h(k)$
  - We probe consecutive locations until one of the following occurs
    - An item with key k is found, or
    - An empty cell is found, or
    - N cells have been unsuccessfully probed

Algorithm *get*(k)
  $i \leftarrow h(k)$
  $p \leftarrow 0$
  repeat
    $c \leftarrow A[i]$
    if $c = \emptyset$
      return *null*
    else if *c.getKey* () = k
      return *c.getValue*()
    else
      $i \leftarrow (i + 1) \bmod N$
      $p \leftarrow p + 1$
  until $p = N$
  return *null*

UPDATES WITH LINEAR PROBING

- To handle insertions and deletions, we introduce a special object, called $DEFUNCT$, which replaces deleted elements
- remove(k)
  - We search for an entry with key k
  - If such an entry (k, o) is found, we replace it with the special item $DEFUNCT$ and we return element o
  - Else, we return $null$
- put(k, o)
  - We throw an exception if the table is full
  - We start at cell $h(k)$
  - We probe consecutive cells until one of the following occurs
    - A cell i is found that is either empty or stores $DEFUNCT$, or
    - N cells have been unsuccessfully probed
  - We store (k, o) in cell i

PROBE HASH MAP IN JAVA

```
1  public class ProbeHashMap<K,V> extends AbstractHashMap<K,V> {
2    private MapEntry<K,V>[ ] table;        // a fixed array of entries (all initially null)
3    private MapEntry<K,V> DEFUNCT = new MapEntry<>(null, null);   //sentinel
4    public ProbeHashMap( ) { super( ); }
5    public ProbeHashMap(int cap) { super(cap); }
6    public ProbeHashMap(int cap, int p) { super(cap, p); }
7    /** Creates an empty table having length equal to current capacity. */
8    protected void createTable( ) {
9      table = (MapEntry<K,V>[ ]) new MapEntry[capacity];   // safe cast
10   }
11   /** Returns true if location is either empty or the "defunct" sentinel. */
12   private boolean isAvailable(int j) {
13     return (table[j] == null || table[j] == DEFUNCT);
14   }
```

PROBE HASH MAP IN JAVA, 2

```
15    /** Returns index with key k, or −(a+1) such that k could be added at index a. */
16    private int findSlot(int h, K k) {
17      int avail = −1;                          // no slot available (thus far)
18      int j = h;                               // index while scanning table
19      do {
20        if (isAvailable(j)) {                  // may be either empty or defunct
21          if (avail == −1) avail = j;          // this is the first available slot!
22          if (table[j] == null) break;         // if empty, search fails immediately
23        } else if (table[j].getKey( ).equals(k))
24          return j;                            // successful match
25        j = (j+1) % capacity;                  // keep looking (cyclically)
26      } while (j != h);                        // stop if we return to the start
27      return −(avail + 1);                     // search has failed
28    }
```

PROBE HASH MAP IN JAVA, 2

```
35    /** Associates key k with value v in bucket with hash value h; returns old value. */
36    protected V bucketPut(int h, K k, V v) {
37      int j = findSlot(h, k);
38      if (j >= 0)                                  // this key has an existing entry
39        return table[j].setValue(v);
40      table[-(j+1)] = new MapEntry<>(k, v);        // convert to proper index
41      n++;
42      return null;
43    }
44    /** Removes entry having key k from bucket with hash value h (if any). */
45    protected V bucketRemove(int h, K k) {
46      int j = findSlot(h, k);
47      if (j < 0) return null;                      // nothing to remove
48      V answer = table[j].getValue( );
49      table[j] = DEFUNCT;                          // mark this slot as deactivated
50      n--;
51      return answer;
52    }
```

PROBE HASH MAP IN JAVA, 3

```
29    /** Returns value associated with key k in bucket with hash value h, or else null. */
30    protected V bucketGet(int h, K k) {
31      int j = findSlot(h, k);
32      if (j < 0) return null;                          // no match found
33      return table[j].getValue( );
34    }

53    /** Returns an iterable collection of all key-value entries of the map. */
54    public Iterable<Entry<K,V>> entrySet( ) {
55      ArrayList<Entry<K,V>> buffer = new ArrayList<>( );
56      for (int h=0; h < capacity; h++)
57        if (!isAvailable(h)) buffer.add(table[h]);
58      return buffer;
59    }
60  }
```

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 removal

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

STORAGE REQUIREMENTS FOR HASH TABLES, SORTED AR RAYS, AND TREES

- The performance of hashing is superior to that of binar y search of an array or a binary search tree, particularl y if the load factor is less than 0.75
- However, the lower the load factor, the more empty sto rage cells
  - there are no empty cells in a sorted array
- A binary search tree requires three references per nod e (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

    $2.5 = 1 + L/2$

    $5.0 = 2 + L$

    $3.0 = 60{,}000/n$

    $n = 20{,}000$