

Unit 5

Heaps and Hashing

Objectives

Upon completion you will be able to

- **Explain Basic Heap concepts.**
- **Illustrate Heap Implementation**
- **Explain Basic Hashing concepts, methods and collision resolution**

Definition

A **heap**, as shown in Figure 9-1, is a binary tree structure with the following properties:

1. The tree is complete or nearly complete.
2. The key value of each node is greater than or equal to the key value in each of its descendants.

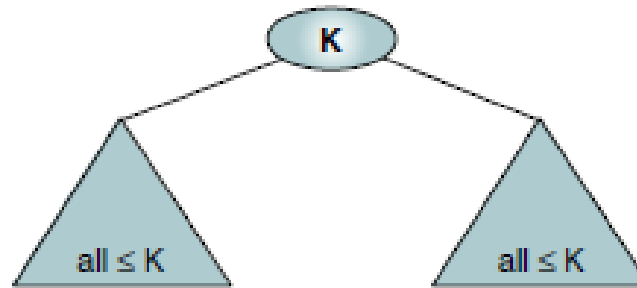
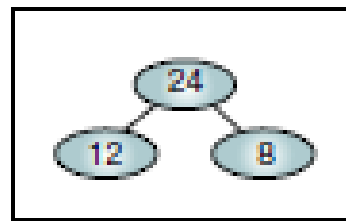


FIGURE 9-1 Heap

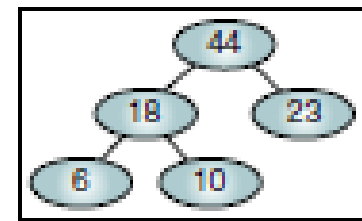
This structure is called a max-heap. The second property of a heap—the key value is greater than the keys of the subtrees—can be reversed to create a min-heap. That is, we can create a minimum heap in which the key value in a node is *less than* the key values in all of its subtrees. Generally speaking, whenever the term *heap* is used by itself, it refers to a max-heap.



(a) Root-only heap

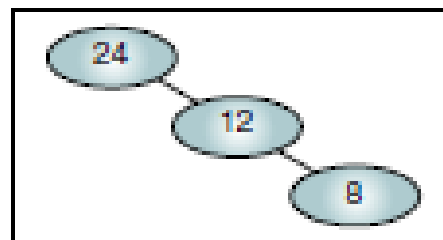


(b) Two-level heap

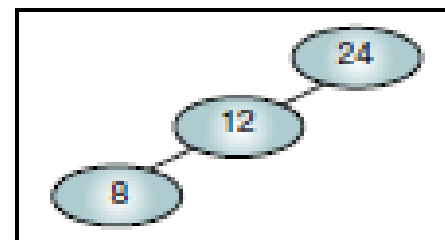


(c) Three-level heap

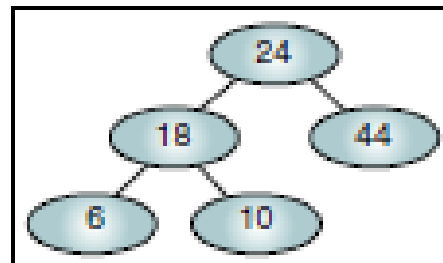
FIGURE 9-2 Heap Trees



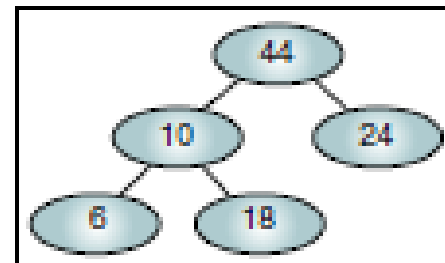
(a) Not nearly complete
(rule 1)



(b) Not nearly complete
(rule 1)



(c) Root not largest
(rule 2)



(d) Subtree 10 not a heap
(rule 2)

FIGURE 9-3 Invalid Heaps

Reheap Up

Imagine that we have a nearly complete binary tree with N elements whose first $N - 1$ elements satisfy the order property of heaps, but the last element does not. In other words, the structure would be a heap if the last element were not there. The reheap up operation repairs the structure so that it is a heap by floating the last element up the tree until that element is in its correct location in the tree.

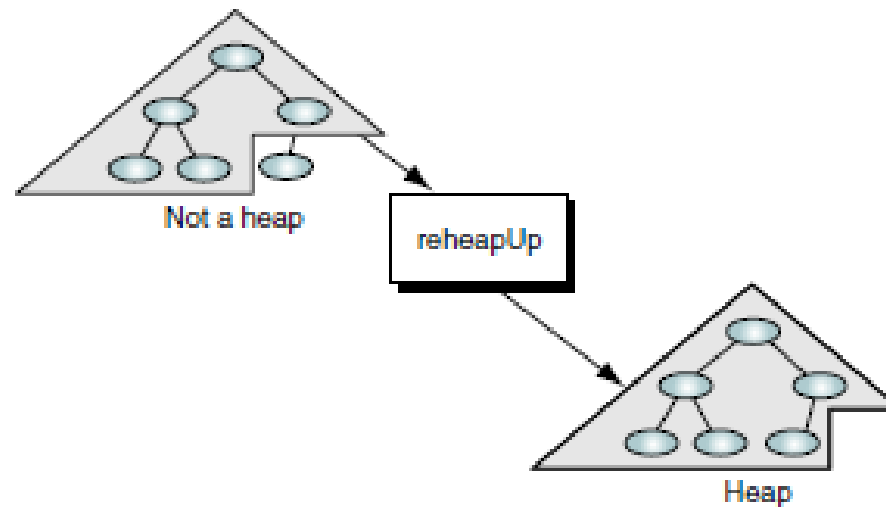
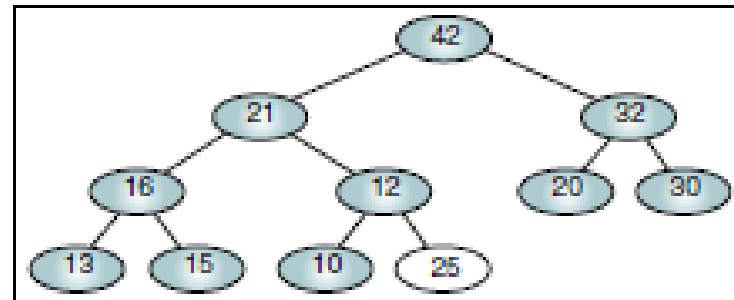
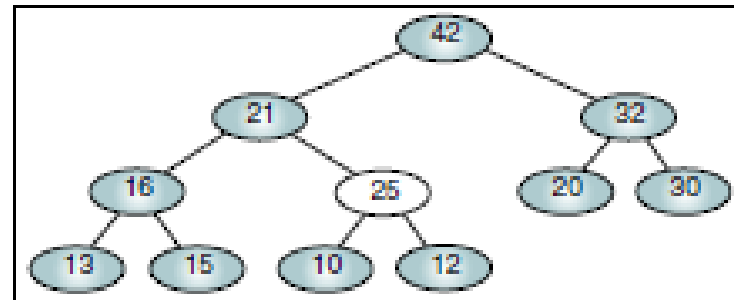


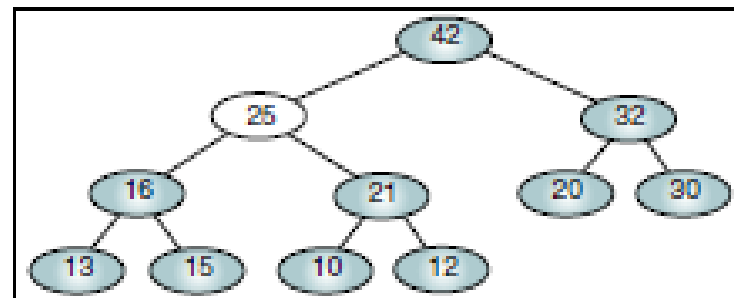
FIGURE 9-4 Reheap Up Operation



(a) Original tree: not a heap



(b) Last element (25) moved up



(c) Moved up again: tree is a heap

FIGURE 9-5 Reheap Up Example

ALGORITHM 9-1 Reheap Up

Algorithm reheapUp (heap, newNode)

Reestablishes heap by moving data in child up to its correct location in the heap array.

Pre heap is array containing an invalid heap

↪newNode is index location to new data in heap

Post heap has been reordered

```
1 if (newNode not the root)
  1 set parent to parent of newNode
  2 if (newNode key > parent key)
    1 exchange newNode and parent)
    2 reheapUp (heap, parent)
  3 end if
2 end if
end reheapUp
```

Reheap Down

Imagine we have a nearly complete binary tree that satisfies the heap order property except in the root position. This situation occurs when the root is deleted from the tree, leaving two disjoint heaps. To correct the situation, we move the data in the last tree node to the root. This action destroys the tree's heap properties. To restore the heap property, we need an operation that sinks the root down until it is in a position where the heap-ordering property is satisfied. We call this operation reheap down.

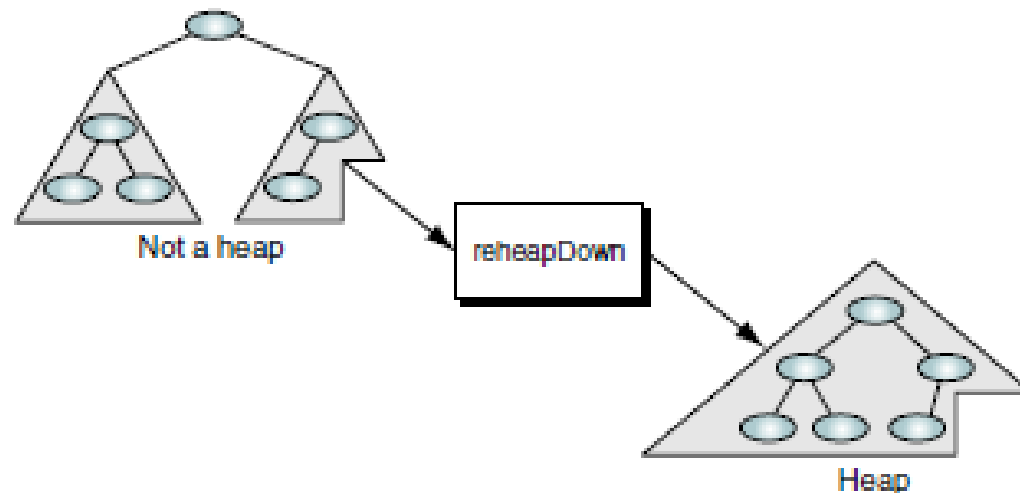
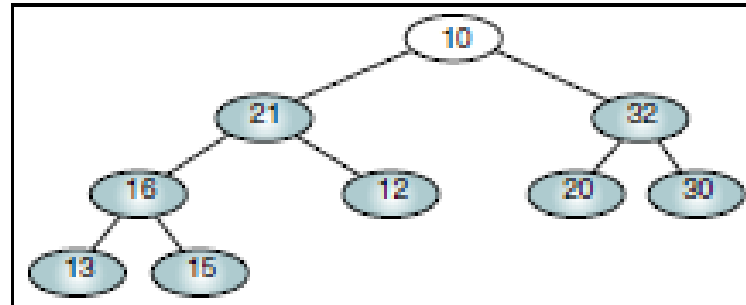
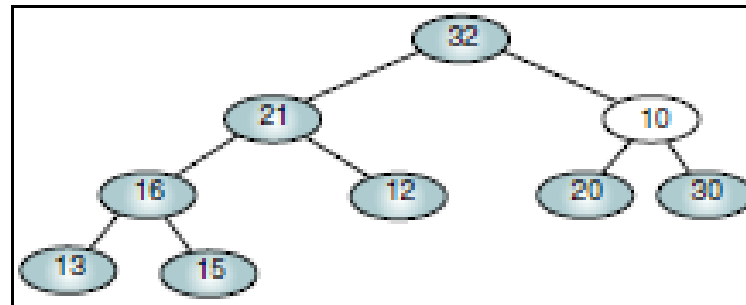


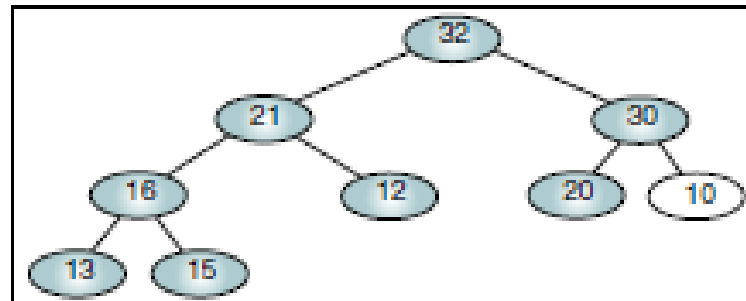
FIGURE 9-6 Reheap Down Operation



(a) Original tree: not a heap



(b) Root moved down (right)



(c) Moved down again: tree is a heap

FIGURE 9-7 Reheap Down Example

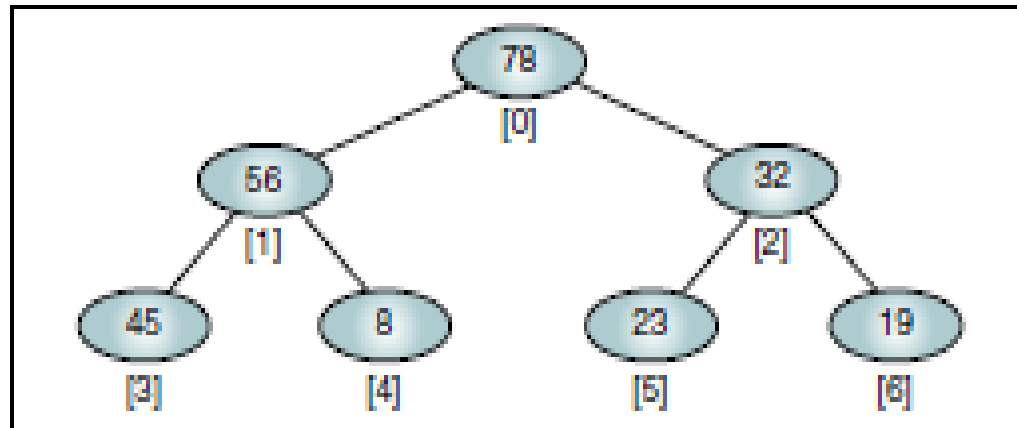
ALGORITHM 9-2 Reheap Down

```
Algorithm reheapDown (heap, root, last)
Reestablishes heap by moving data in root down to its
correct location in the heap.
    Pre    heap is an array of data
           root is root of heap or subheap
           last is an index to the last element in heap
    Post   heap has been restored
    Determine which child has larger key
1  if (there is a left subtree)
    1  set leftKey to left subtree key
    2  if (there is a right subtree)
        1  set rightKey to right subtree key
    3  else
        1  set rightKey to null key
    4  end if
    5  if (leftKey > rightKey)
        1  set largeSubtree to left subtree
    6  else
        1  set largeSubtree to right subtree
    7  end if
    Test if root > larger subtree
    8  if (root key < largeSubtree key)
        1  exchange root and largeSubtree
        2  reheapDown (heap, largeSubtree, last)
    9  end if
2  end if
end reheapDown
```

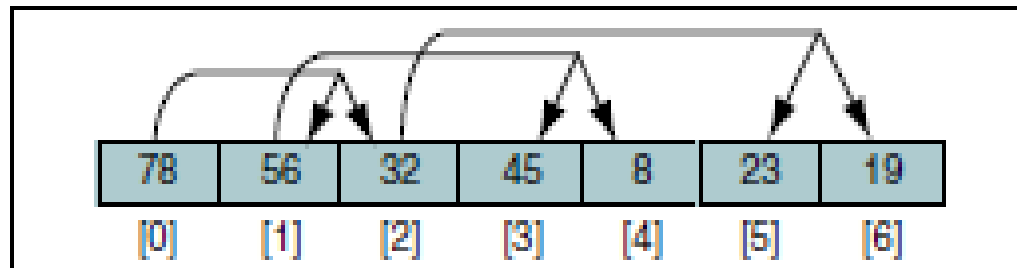
Heap Implementation

Although a heap can be built in a dynamic tree structure, it is most often implemented in an array. This implementation is possible because the heap is, by definition, complete or nearly complete. Therefore, the relationship between a node and its children is fixed and can be calculated as shown below.

1. For a node located at index i , its children are found at:
 - a. Left child: $2i + 1$
 - b. Right child: $2i + 2$
2. The parent of a node located at index i is located at
3. Given the index for a left child, j , its right sibling, if any, is $\lfloor (j - 1) / 2 \rfloor + 1$.
Conversely, given the index for a right child, k , its left sibling, which must exist, is found at $k - 1$.
4. Given the size, n , of a complete heap, the location of the first leaf is
5. $\lfloor n / 2 \rfloor$: location of the first leaf element, the location of the last nonleaf element is one less.



(a) Heap in its logical form



(b) Heap in an array

FIGURE 9-8 Heaps in Arrays

Build a Heap

Given a filled array of elements in random order, to build the heap we need to rearrange the data so that each node in the heap is greater than its children.

We begin by dividing the array into two parts, the left being a heap and the right being data to be inserted into the heap. At the beginning the root (the first node) is the only node in the heap and the rest of the array are data to be inserted.

To insert a node into the heap, we follow the parent path up the heap, swapping nodes that are out of order. If the nodes are in order, the insertion terminates and the next node is selected and inserted into the heap. This process is sometimes referred to as **heapify**.

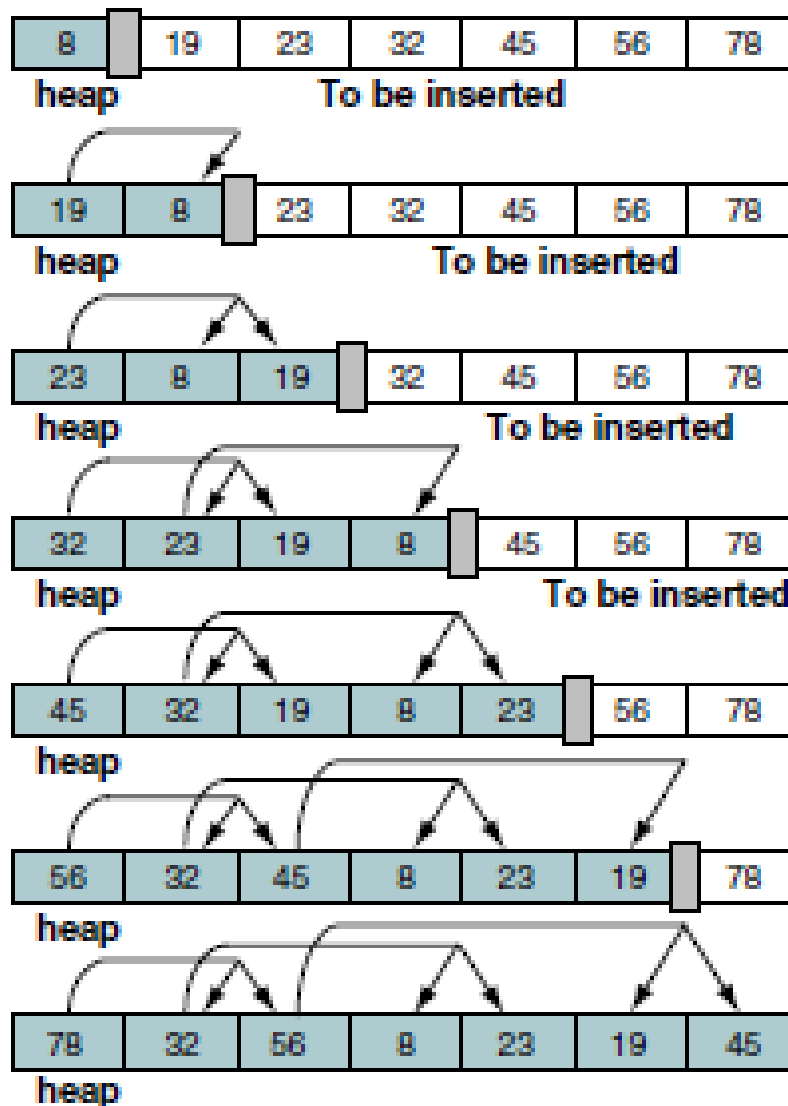
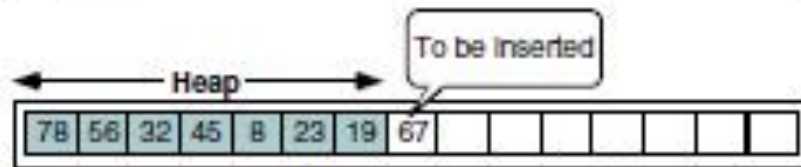
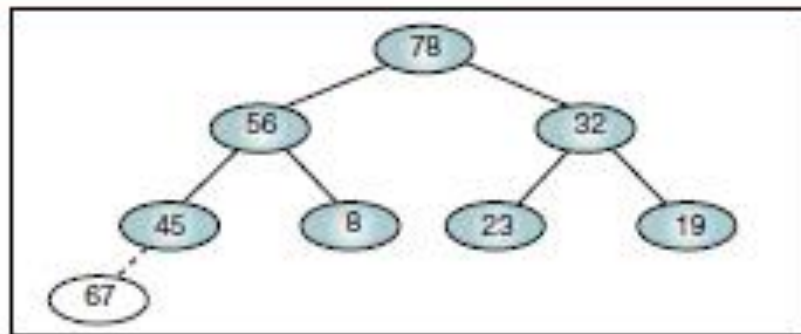


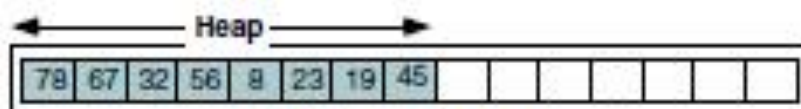
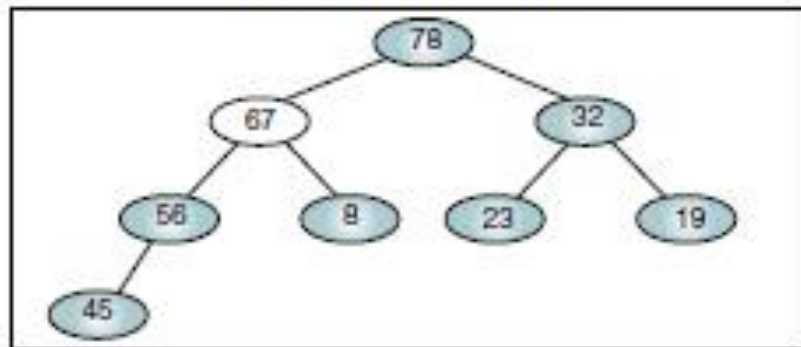
FIGURE 9-9 Building a Heap

ALGORITHM 9-3 Build Heap

```
Algorithm buildHeap (heap, size)
Given an array, rearrange data so that they form a heap.
    Pre    heap is array containing data in nonheap order
           size is number of elements in array
    Post   array is now a heap
1 set walker to 1
2 loop (walker < size)
    1 reheapUp(heap, walker)
    2 increment walker
3 end loop
end buildHeap
```



(a) Before reheap up



(b) After reheap up

FIGURE 9-10 Insert Node

Insert a Node into a Heap

To insert a node, we need to locate the first empty leaf in the array. We find it immediately after the last node in the tree, which is given as a parameter. To insert a node, we move the new data to the first empty leaf and reheap up.

ALGORITHM 9-4 Insert Heap

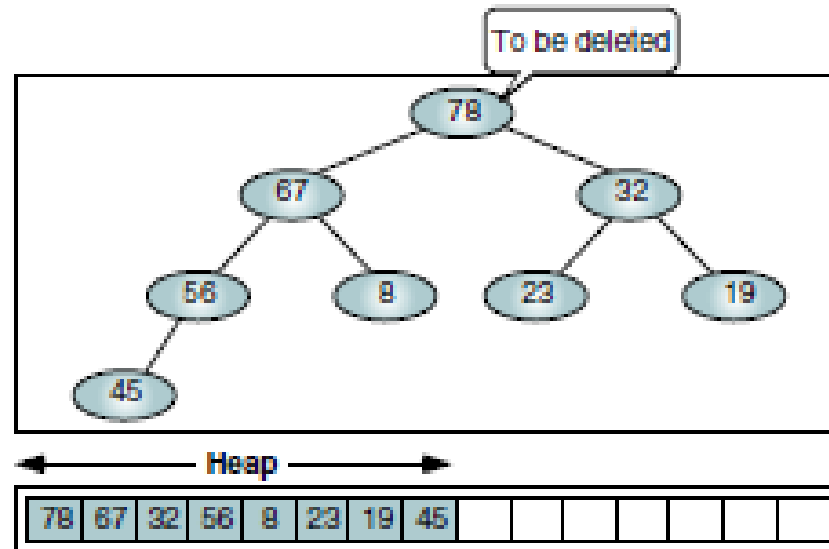
```
Algorithm insertHeap (heap, last, data)
Inserts data into heap.
    Pre    heap is a valid heap structure
           last is reference parameter to last node in heap
           data contains data to be inserted
    Post   data have been inserted into heap
    Return true if successful; false if array full
1  if (heap full)
    1  return false
2  end if
3  increment last
4  move data to last node
5  reheapUp (heap, last)
6  return true
end insertHeap
```


Delete a Node from a Heap

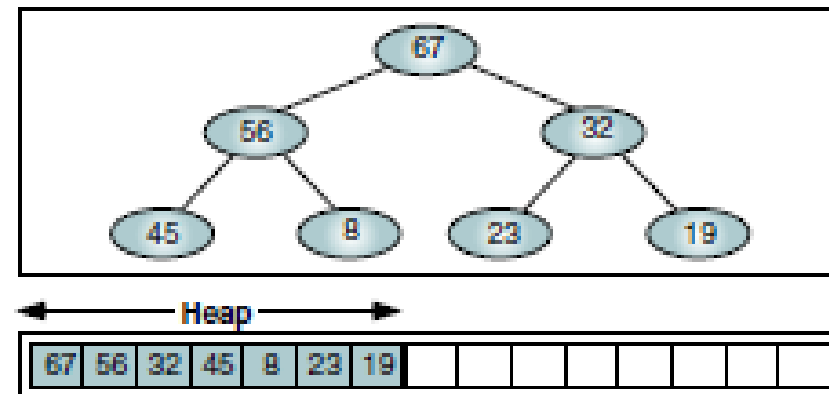
When deleting a node from a heap, the most common and meaningful logic is to delete the root. In fact, the rationale for a heap is to determine and extract the largest element, the root. After it has been deleted, the heap is thus left without a root. To reestablish the heap, we move the data in the last heap node to the root and reheap down.

ALGORITHM 9-5 Delete Heap Node

```
Algorithm deleteHeap (heap, last, dataOut)
Deletes root of heap and passes data back to caller.
    Pre    heap is a valid heap structure
           last is reference parameter to last node in heap
           dataOut is reference parameter for output area
    Post   root deleted and heap rebuilt
           root data placed in dataOut
    Return true if successful; false if array empty
1  if (heap empty)
    1  return false
2  end if
3  set dataOut to root data
4  move last data to root
5  decrement last
6  reheapDown (heap, 0, last)
7  return true
end deleteHeap
```

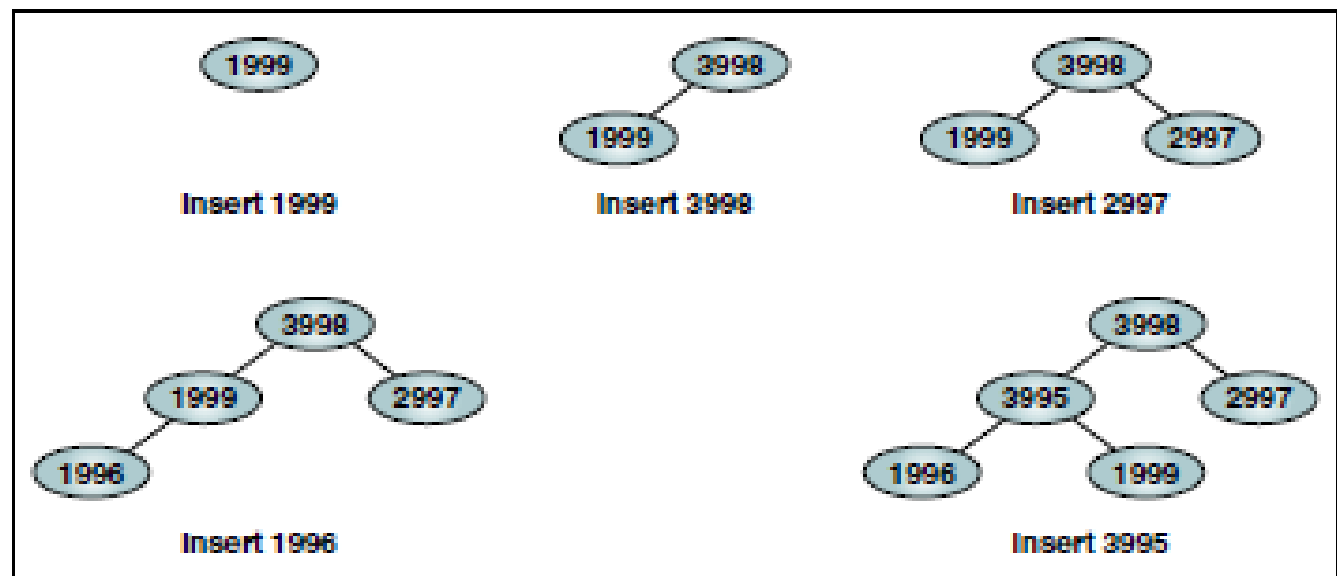


(a) Before delete

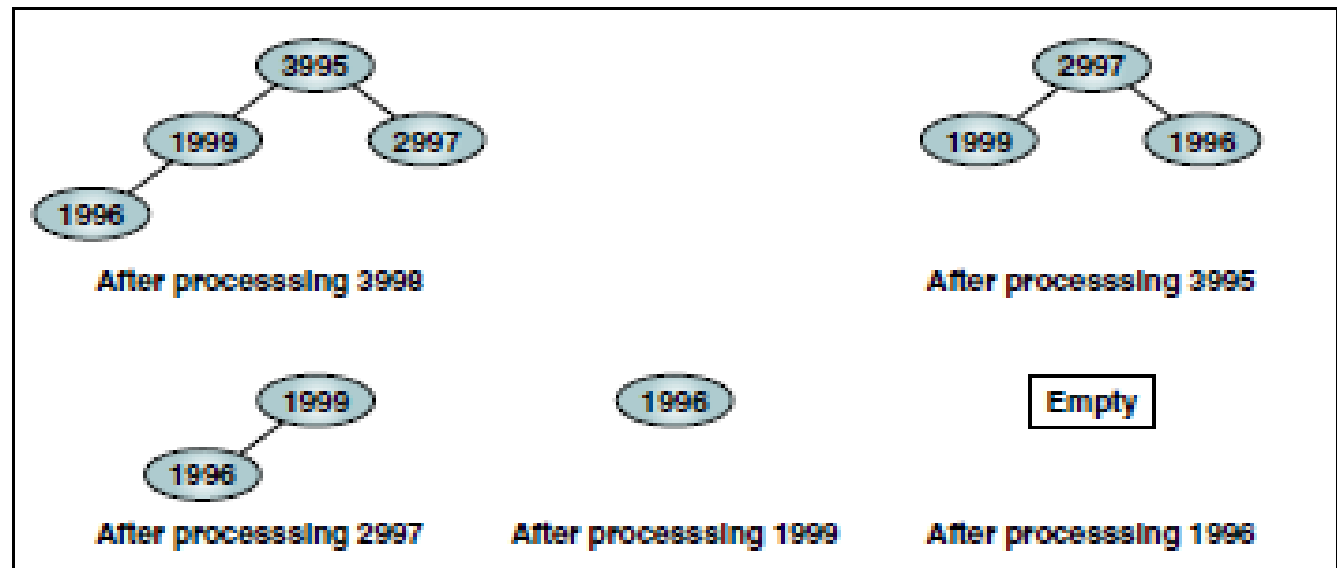


(b) After delete

FIGURE 9-11 deleteHeap Node



(a) Insert customers



(b) Process customers

FIGURE 9-16 Priority Queue Example

HASHING Basic Concepts

In a hashed search, the key, through an algorithmic function, determines the location of the data.

Because we are searching an array, we use a hashing algorithm to transform the key into the index that contains the data we need to locate.

Another way to describe hashing is as a key-to-address transformation in which the keys map to addresses in a list.

Hashing is a key-to-address mapping process.

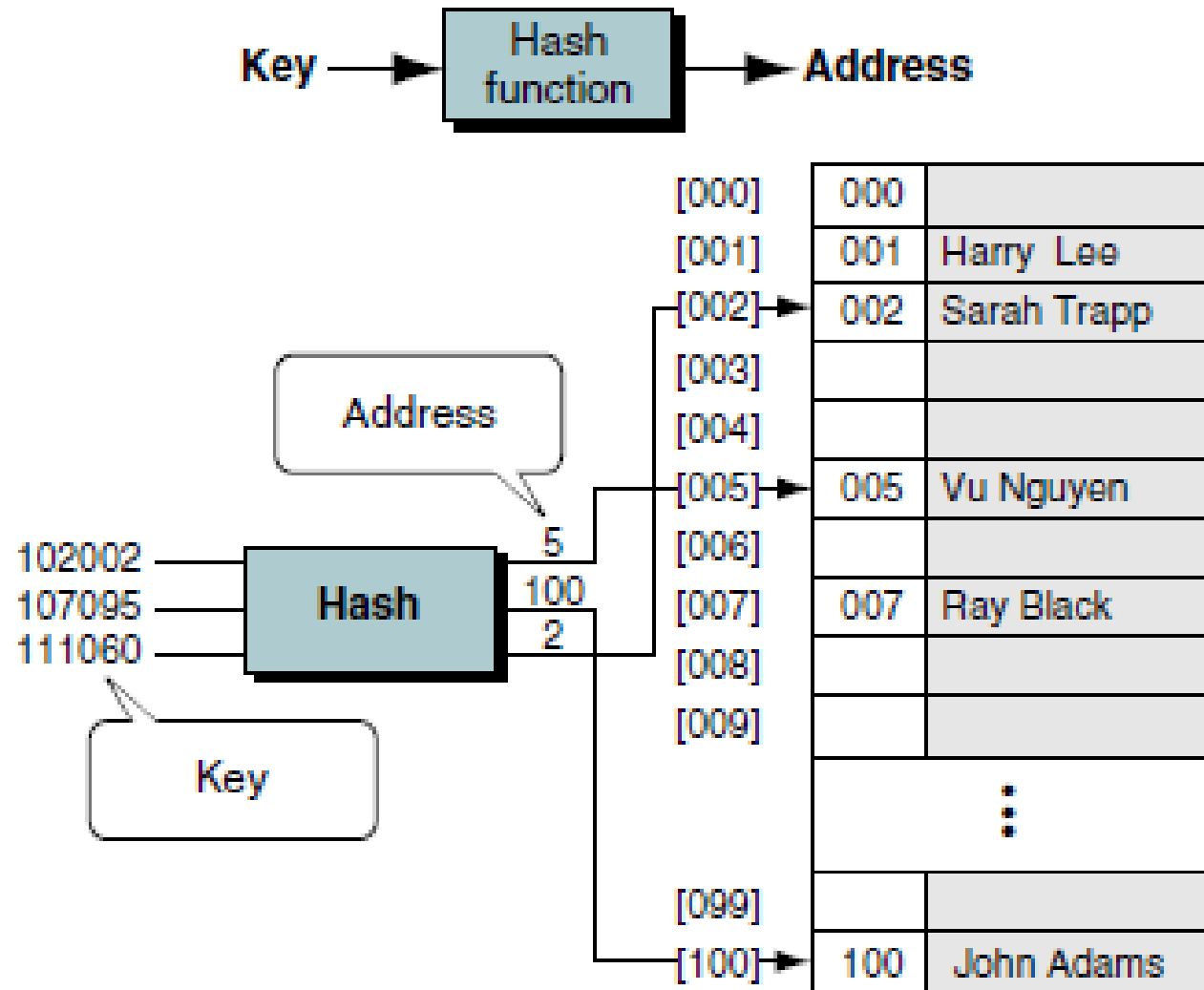


FIGURE 13-6 Hash Concept

Synonyms - The set of keys that hash to the same location

Collision - If the actual data that we insert into our list contain two or more synonyms, we can have collisions. A collision occurs when a hashing algorithm produces an address for an insertion key and that address is already occupied.

The address produced by the hashing algorithm is known as the **home address**.

The memory that contains all of the home addresses is known as the **prime area**.

When two keys collide at a home address, we must resolve the collision by placing one of the keys and its data in another location.

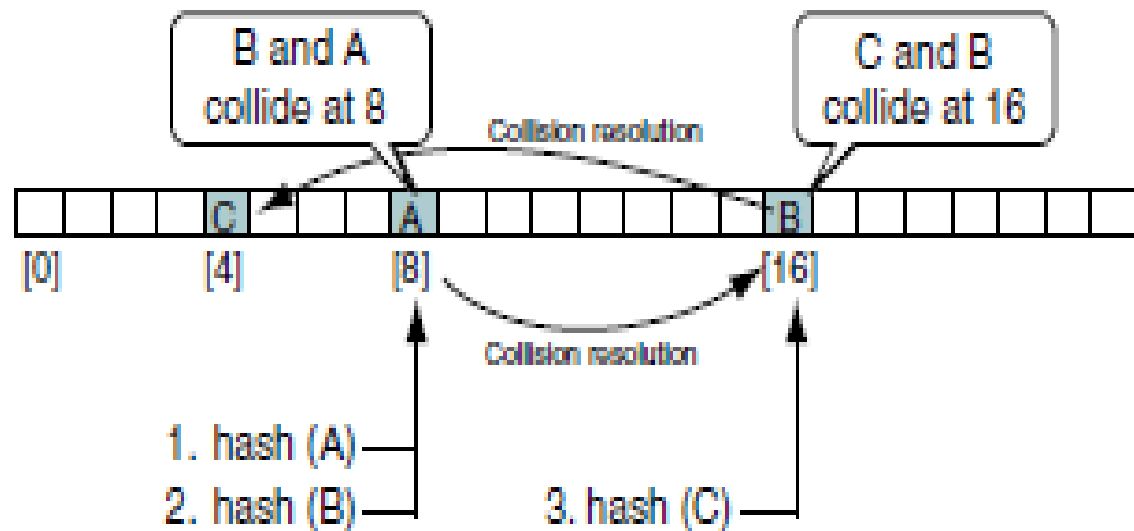


FIGURE 13-7 Collision Resolution Concept

When we need to locate an element in a hashed list, we must use the same algorithm that we used to insert it into the list. Consequently, we first hash the key and check the home address to determine whether it contains the desired element. If it does, the search is complete. If not, we must use the collision resolution algorithm to determine the next location and continue until we find the element or determine that it is not in the list. Each calculation of an address and test for success is known as a **probe**.

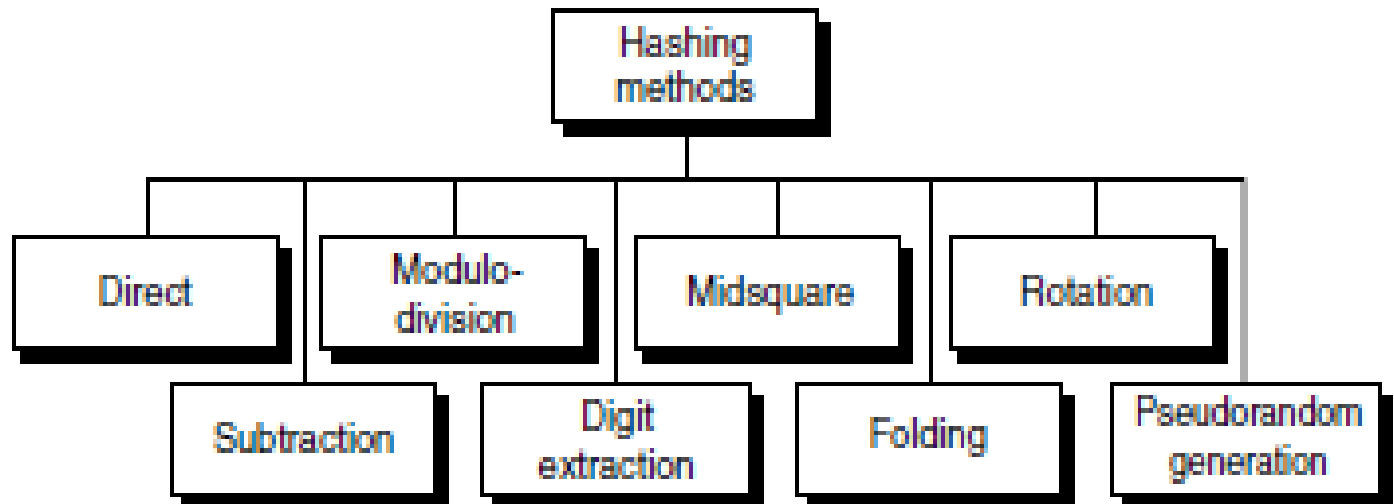


FIGURE 13-8 Basic Hashing Techniques

Direct Method

In direct hashing the key is the address without any algorithmic manipulation.

The data structure must therefore contain an element for every possible key.

The situations in which you can use direct hashing are limited, but it can be very powerful because it guarantees that there are no synonyms and therefore no collisions.

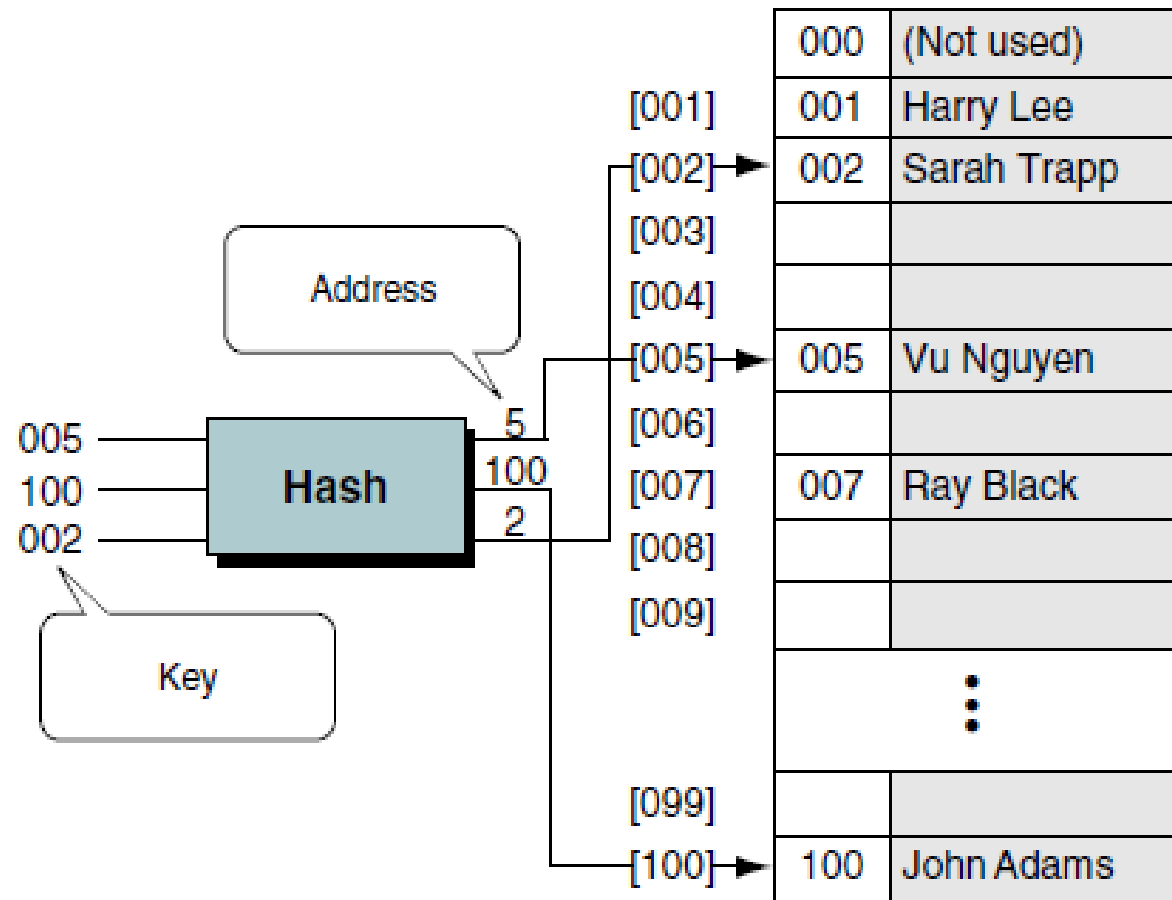


FIGURE 13-9 Direct Hashing of Employee Numbers

Subtraction Method

Sometimes keys are consecutive but do not start from 1. For example, a company may have only 100 employees, but the employee numbers start from 1001 and go to 1100. In this case we use subtraction hashing, a very simple hashing function that subtracts 1000 from the key to determine the address. The beauty of this example is that it is simple and guarantees that there will be no collisions. Its limitations are the same as direct hashing: it can be used only for small lists in which the keys map to a densely filled list.

The direct and subtraction hash functions both guarantee a search effort of one with no collisions. They are one-to-one hashing methods: only one key hashes to each address.

Modulo-division Method

Also known as division remainder, the modulo-division method divides the key by the array size and uses the remainder for the address. This method gives us the simple hashing algorithm shown below in which `listSize` is the number of elements in the array:

```
address = key MODULO listSize
```

This algorithm works with any list size, but a list size that is a **prime number** produces fewer collisions than other list sizes.

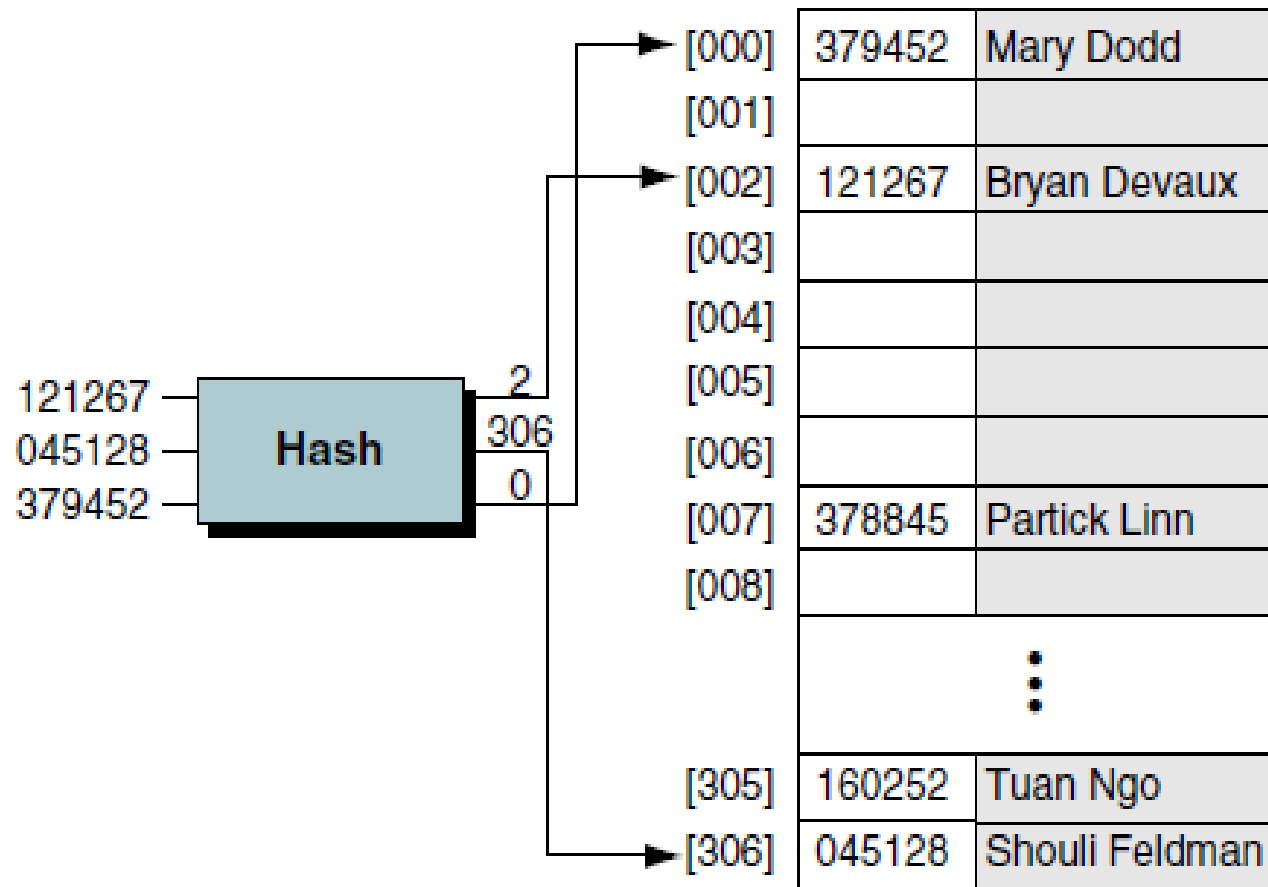


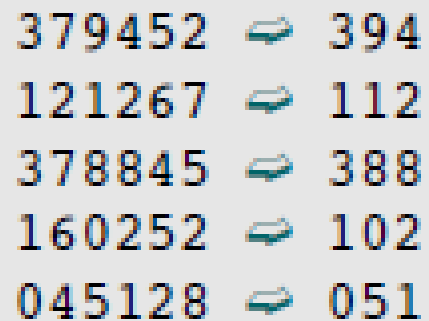
FIGURE 13-10 Modulo-division Hashing

$121267 / 307 = 395$ with remainder of 2

Therefore: $\text{hash}(121267) = 2$

Digit-extraction Method

Using digit extraction selected digits are extracted from the key and used as the address. For example, using our six-digit employee number to hash to a three digit address (000–999), we could select the first, third, and fourth digits (from the left) and use them as the address.



379452	⇔	394
121267	⇔	112
378845	⇔	388
160252	⇔	102
045128	⇔	051

Midsquare Method

In midsquare hashing the key is squared and the address is selected from the middle of the squared number. The most obvious limitation of this method is the size of the key. Given a key of six digits, the product will be 12 digits, which is beyond the maximum integer size of many computers. Because most personal computers can handle a nine-digit integer, let's demonstrate the concept with keys of four digits. Given a key of 9452, the midsquare address calculation is shown below using a four-digit address (0000–9999).

$9452^2 = 89340304$: address is 3403

As a variation on the midsquare method, we can select a portion of the key, such as the middle three digits, and then use them rather than the whole key. Doing so allows the method to be used when the key is too large to square. For example, we can select the first three digits and then use the midsquare method as shown below. (We select the third, fourth, and fifth digits as the address.)

379452:	379^2	=	143641	↔	364
121267:	121^2	=	014641	↔	464
378845:	378^2	=	142884	↔	288
160252:	160^2	=	025600	↔	560
045128:	045^2	=	002025	↔	202

Note that in the midsquare method the same digits must be selected from the key.

Folding Methods

Two folding methods are used: **fold shift** and **fold boundary**.

In fold shift the key value is divided into parts whose size matches the size of the required address. Then the left and right parts are shifted and added with the middle part. For example, imagine that we want to map Social Security numbers into three-digit addresses. We divide the nine-digit Social Security number into three three-digit numbers, which are then added. If the resulting sum is greater than 999, we discard the leading digit.

In fold boundary the left and right numbers are folded on a fixed boundary between them and the center number. The two outside values are thus reversed, where 123 is folded to 321 and 789 is folded to 987. It is interesting to note that the two folding methods give different hashed addresses.

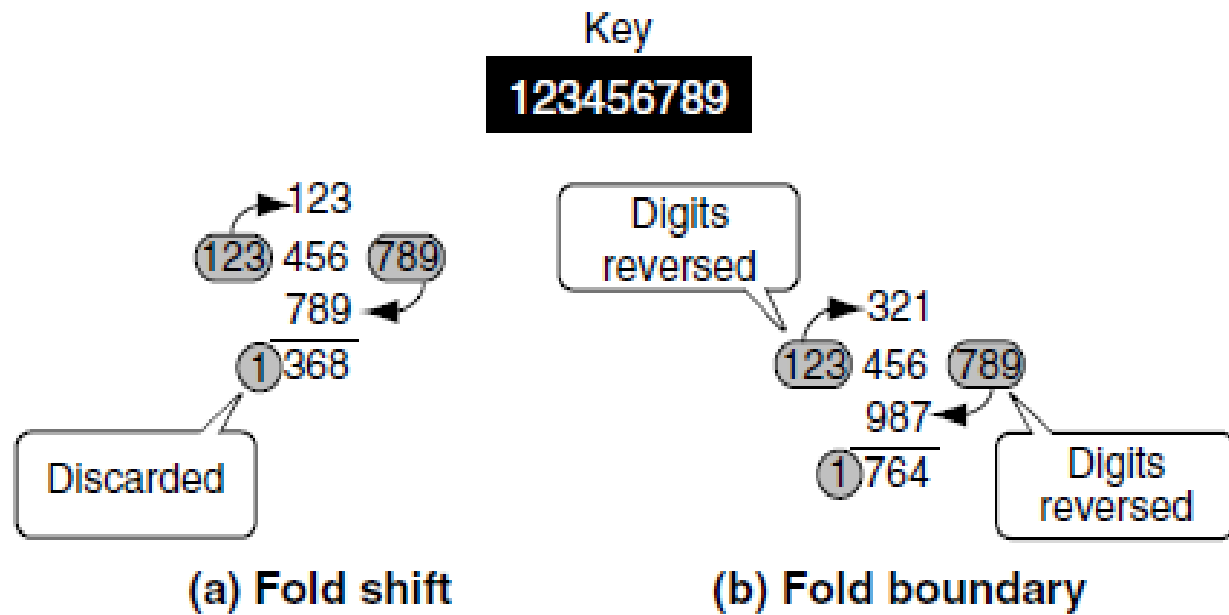


FIGURE 13-11 Hash Fold Examples

Rotation Method

Rotation hashing is generally not used by itself but rather is incorporated in combination with other hashing methods. It is most useful when keys are assigned serially, such as we often see in employee numbers and part numbers. A simple hashing algorithm tends to create synonyms when hashing keys are identical except for the last character. Rotating the last character to the front of the key minimizes this effect. For example, consider the case of a six-digit employee number that might be used in a large company.

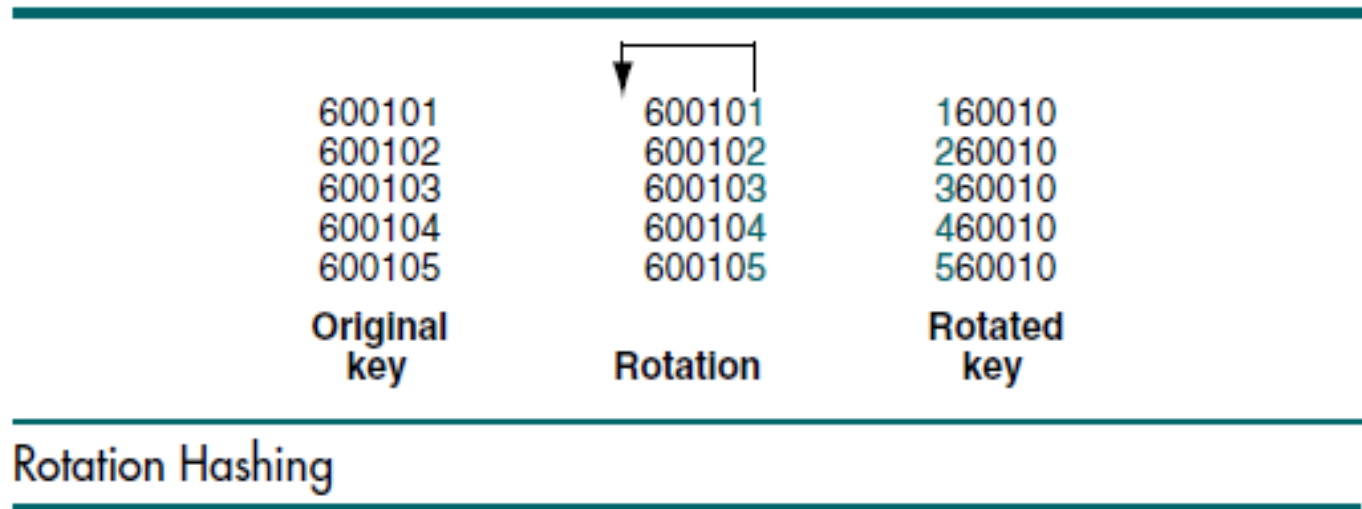


FIGURE 13-12 Rotation Hashing

Pseudorandom Hashing

In pseudorandom hashing the key is used as the seed in a pseudorandom-number generator, and the resulting random number is then scaled into the possible address range using modulo-division. Given a fixed seed, pseudorandom-number generators always generate the same series of numbers. That is what allows us to use them in hashing.

A common random-number generator is shown below.

$$y = ax + c$$

To use the pseudorandom-number generator as a hashing method, we set x to the key, multiply it by the coefficient a , and then add the constant c . The result is then divided by the list size, with the remainder being the hashed address. For maximum efficiency, the factors a and c should be prime numbers.

Let's demonstrate the concept with an example from Figure 13-10. To keep the calculation reasonable, we use 17 and 7 for factors a and c , respectively. Also, the list size in the example is the prime number 307.

```
y = ((17 * 121267) + 7) modulo 307  
y = (2061539 + 7) modulo 307  
y = 2061546 modulo 307  
y = 41
```

One Hashing Algorithm

Assume that we have an alphanumeric key consisting of up to 30 bytes that we need to hash into a 32-bit address.

The first step is to convert the alphanumeric key into a number by adding the American Standard Code for Information Interchange (ASCII) value for each character to an accumulator that will be the address.

As each character is added, we rotate the bits in the address to maximize the distribution of the values.

After the characters in the key have been completely hashed, we take the absolute value of the address and then map it into the address range for the file.

ALGORITHM 13-6 Hashing Algorithm

Algorithm hash (key, size, maxAddr, addr)

This algorithm converts an alphanumeric key of size characters into an integral address.

Pre key is a key to be hashed

size is the number of characters in the key

maxAddr is maximum possible address for the list

Post addr contains the hashed address

1 set looper to 0

2 set addr to 0

Hash key

3 for each character in key

1 if (character not space)

1 add character to address

2 rotate addr 12 bits right

2 end if

4 end loop

Test for negative address

5 if (addr < 0)

1 addr = absolute(addr)

6 end if

7 addr = addr modulo maxAddr

end hash

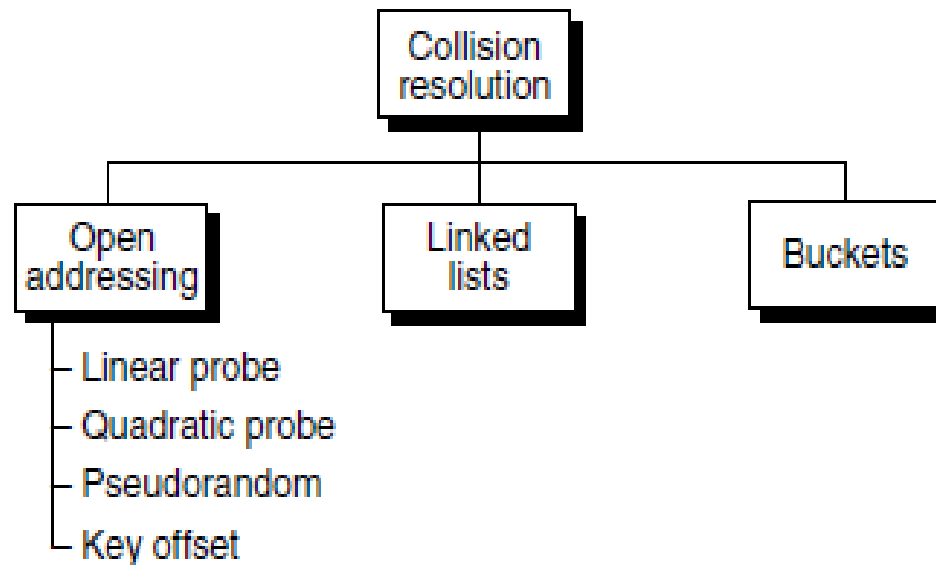


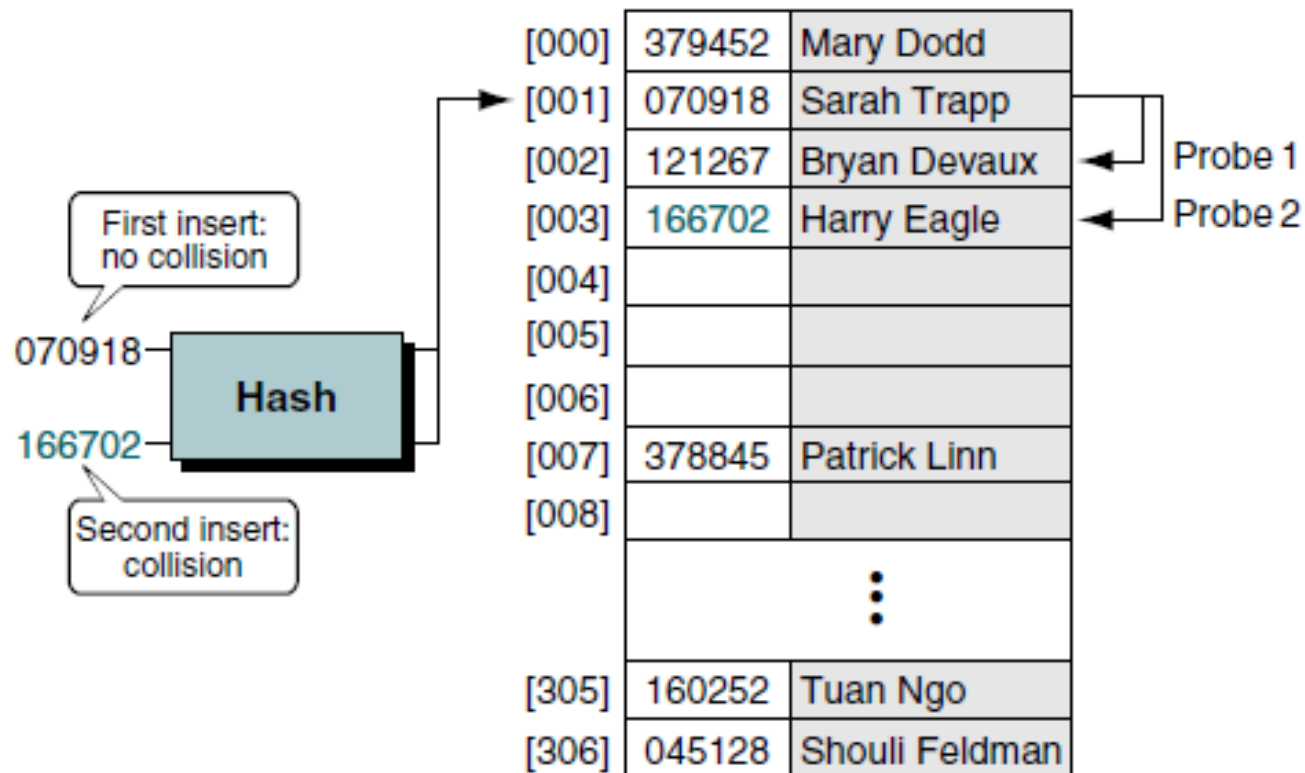
FIGURE 13-13 Collision Resolution Methods

Open addressing resolves collisions in the prime area—that is, the area that contains all of the home addresses.

In Linked list resolution, the collisions are resolved by placing the data in a separate overflow area.

Linear Probe

In a linear probe, which is the simplest, when data cannot be stored in the home address we resolve the collision by adding 1 to the current address.



Linear Probe Collision Resolution

Linear probes have two advantages:

- They are quite simple to implement.
- Data tend to remain near their home address.

Disadvantages:

- Linear probes tend to produce primary clustering(data clustered around home address)
- They tend to make the search algorithm more complex, especially after data have been deleted.

Quadratic Probe

Primary clustering can be eliminated by adding a value other than 1 to the current address. One easily implemented method is to use the quadratic probe. In the quadratic probe, the increment is the collision probe number squared.

To ensure that we don't run off the end of the address list, we use the modulo of the quadratic sum for the new address.

Probe number	Collision location	Probe ² and increment	New address
1	1	1 ² = 1	1 + 1 ⇨ 02
2	2	2 ² = 4	2 + 4 ⇨ 06
3	6	3 ² = 9	6 + 9 ⇨ 15
4	15	4 ² = 16	15 + 16 ⇨ 31
5	31	5 ² = 25	31 + 25 ⇨ 56
6	56	6 ² = 36	56 + 36 ⇨ 92
7	92	7 ² = 49	92 + 49 ⇨ 41
8	41	8 ² = 64	41 + 64 ⇨ 05
9	5	9 ² = 81	5 + 81 ⇨ 86
10	86	10 ² = 100	86 + 100 ⇨ 86

Quadratic Collision Resolution Increments

Disadvantages of the quadratic probe:

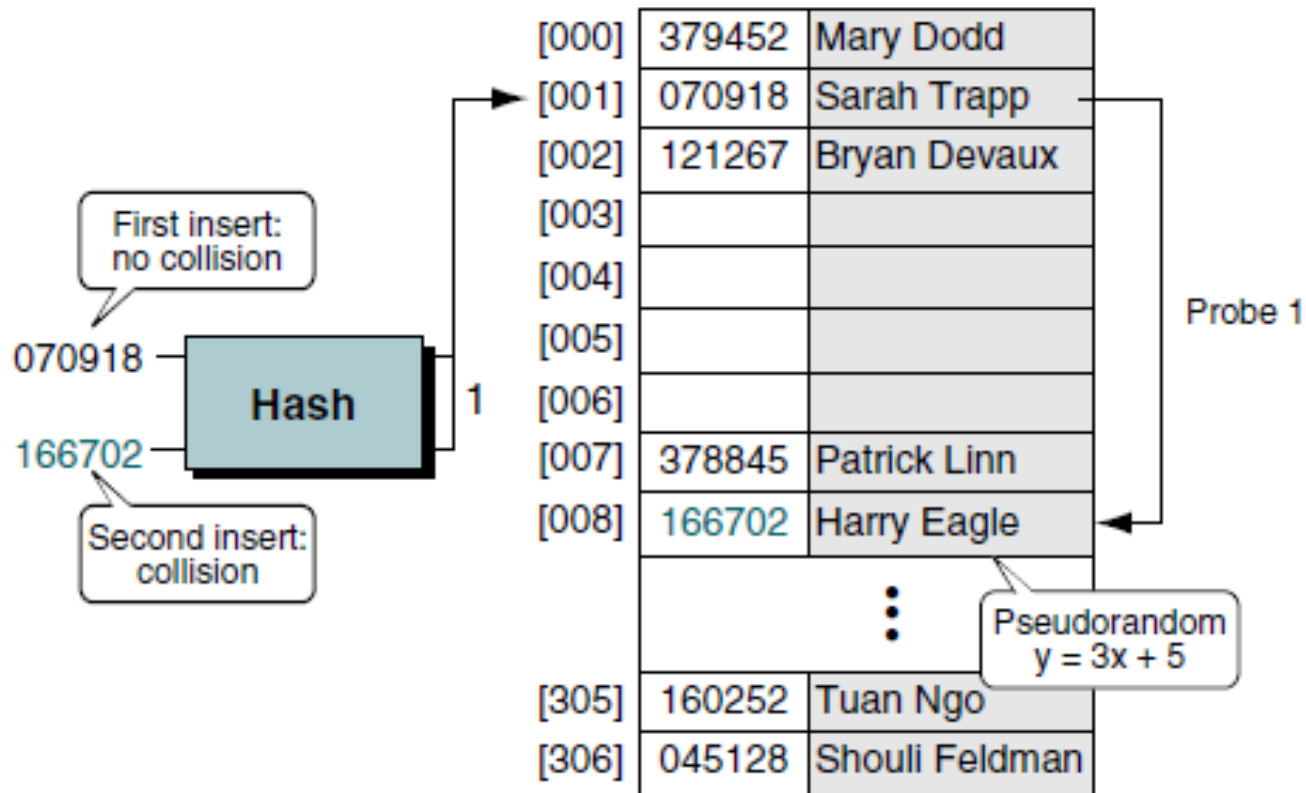
- Time required to square the probe number
- It is not possible to generate a new address for every element in the list.

Pseudorandom Collision Resolution

The last two open addressing methods are collectively known as double hashing. In each method, rather than use an arithmetic probe function, the address is rehashed.

Pseudorandom collision resolution uses a pseudorandom number to resolve the collision. We saw the pseudorandom-number generator as a hashing method in the “Pseudorandom Hashing”. We now use it as a collision resolution method. In this case, rather than use the key as a factor in the random-number calculation, we use the collision address.

Pseudorandom numbers are a relatively simple solution, but they have one significant limitation: all keys follow only one collision resolution path through the list.



Pseudorandom Collision Resolution

a is 3 and c is 5:

$$\begin{aligned}
 y &= (ax + c) \text{ modulo } \text{listSize} \\
 &= (3 \times 1 + 5) \text{ Modulo } 307 \\
 &= 8
 \end{aligned}$$

Key Offset

Key offset is a double hashing method that produces different collision paths for different keys. Whereas the pseudorandom-number generator produces a new address as a function of the previous address, key offset calculates the new address as a function of the old address and the key.

```
offset = ⌊key/listSize⌋  
address = ((offset + old address) modulo listSize)
```

Key	Home address	Key offset	Probe 1	Probe 2
166702	1	543	237	166
572556	1	1865	024	047
067234	1	219	220	132

Key-offset Examples

```
offset = ⌊166702/307⌋ = 543  
address = ((543 + 001) modulo 307) = 237
```

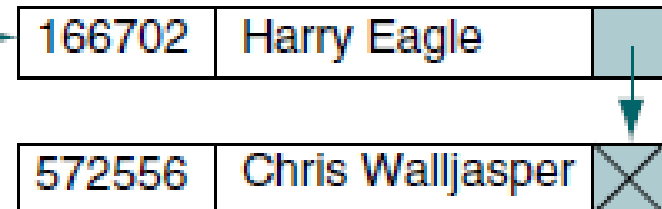
```
offset = ⌊166702/307⌋ = 543  
address = ((543 + 237) modulo 307) = 166
```

Linked List Collision Resolution

A major disadvantage to open addressing is that each collision resolution increases the probability of future collisions. This disadvantage is eliminated in the second approach to collision resolution: linked lists. A linked list is an ordered collection of data in which each element contains the location of the next element.

Linked list collision resolution uses a separate area to store collisions and chains all synonyms together in a linked list. It uses two storage areas: the prime area and the overflow area. Each element in the prime area contains an additional field—a link head pointer to a linked list of overflow data in the overflow area. When a collision occurs, one element is stored in the prime area and chained to its corresponding linked list in the overflow area. Although the overflow area can be any data structure, it is typically implemented as a linked list in dynamic memory.

[000]	379452	Mary Dodd	X
[001]	070918	Sarah Trapp	
[002]	121267	Bryan Devaux	X
[003]			X
[004]			X
[005]			X
[006]			X
[007]	378845	Patrick Linn	X
[008]			X
⋮			
[305]	160252	Tuan Ngo	X
[306]	045128	Shouli Feldman	X



Linked List Collision Resolution

Bucket Hashing

Another approach to handling the collision problems is bucket hashing, in which keys are hashed to buckets, nodes that accommodate multiple data occurrences. Because a bucket can hold multiple data, collisions are postponed until the bucket is full.

There are two problems with this concept.

First, it uses significantly more space because many of the buckets are empty or partially empty at any given time.

Second, it does not completely resolve the collision problem. At some point a collision occurs and needs to be resolved. When it does, a typical approach is to use a linear probe.

[000]	Bucket 0	379452	Mary Dodd
[001]	Bucket 1	070918	Sarah Trapp
		166702	Harry Eagle
		367173	Ann Giorgis
[002]	Bucket 2	121267	Bryan Devaux
		572556	Chris Walljasper
		⋮	Linear probe placed here
[307]	Bucket 307	045128	

Linear probe placed here

Bucket Hashing