COURSE TECHNOLOGY
CENGAGE Learning™

Data Structures Using C++ 2E

Chapter 9
Searching and Hashing Algorithms

Objectives

- Learn the various search algorithms
- Explore how to implement the sequential and binary search algorithms
- Discover how the sequential and binary search algorithms perform
- Become aware of the lower bound on comparison-based search algorithms
- Learn about hashing

Search Algorithms

- Item key
  - Unique member of the item
  - Used in searching, sorting, insertion, deletion
- Number of key comparisons
  - Comparing the key of the search item with the key of an item in the list
- Where/when to use?
  - Determine if a data item exist
  - Insert a data item
  - Delete a data item
- Performance of search algorithms

Sequential Search

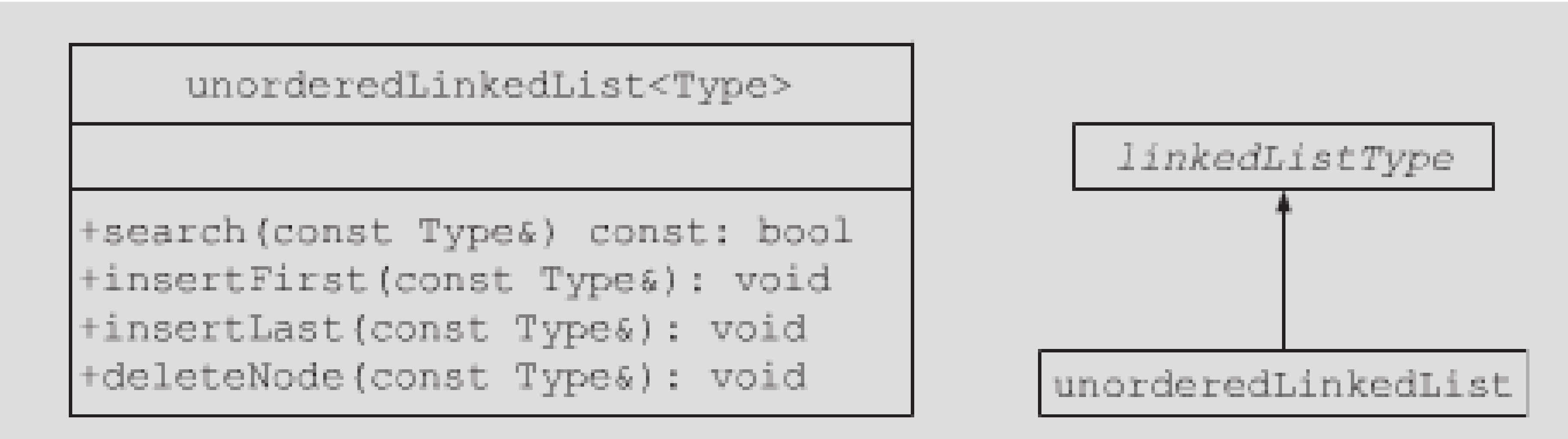
- Sequential search in
  - Array-based list (Chapter 3):

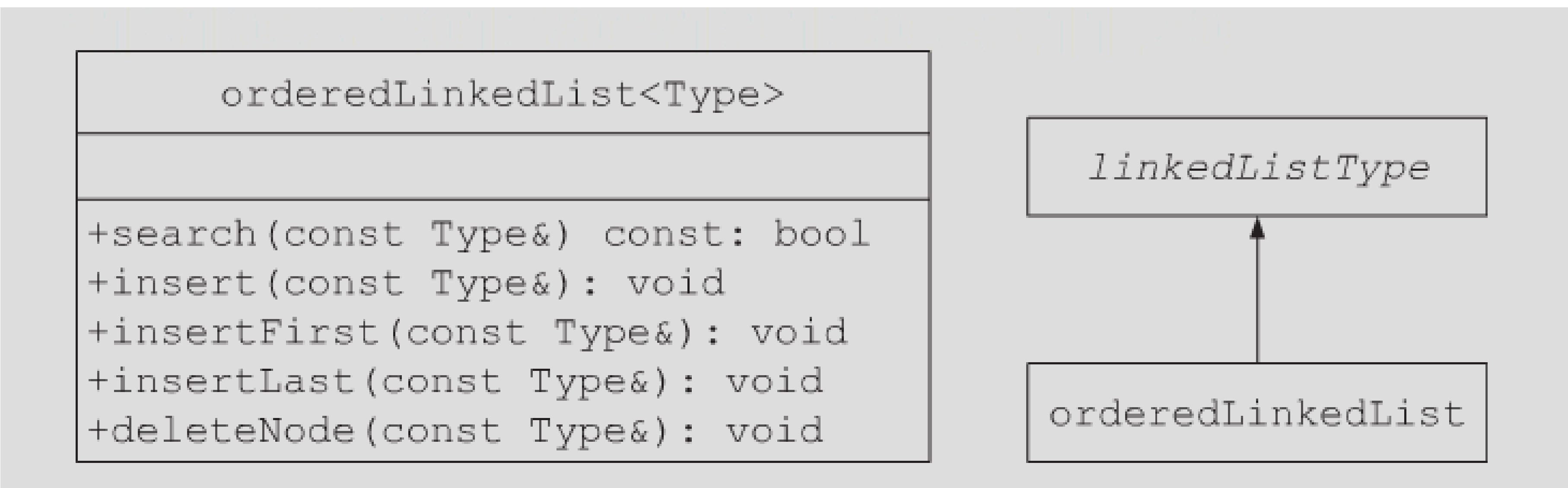
    `class arrayListType`
  - Linked lists (Chapter 5):

    `class unorderedLinkedList`

    `class orderedLinkedList`
- Works the same for array-based lists and linked lists

```
template <class elemType>
class arrayListType
{
public:
    bool isEmpty() const;
    bool isFull() const;
    int listSize() const;
    int maxListSize() const;
    void print() const;
    bool isItemAtEqual(int location, const elemType& item) const;
    void insertAt(int location, const elemType& insertItem);
    void insertEnd(const elemType& insertItem);
    void removeAt(int location);
    void retrieveAt(int location, elemType& retItem) const;
    void replaceAt(int location, const elemType& repItem);
    void clearList();
    int seqSearch(const elemType& item) const;
    void insert(const elemType& insertItem);
    void remove(const elemType& removeItem);

    arrayListType(int size = 100);

    arrayListType(const arrayListType<elemType>& otherList);

    ~arrayListType();

protected:
    elemType *list;   //array to hold the list elements
    int length;       //to store the length of the list
    int maxSize;      //to store the maximum size of the list
```


unorderedLinkedList<Type>
+search(const Type&) const: bool
+insertFirst(const Type&): void
+insertLast(const Type&): void
+deleteNode(const Type&): void
linkedListType
unorderedLinkedList


orderedLinkedList<Type>
+search(const Type&) const: bool
+insert(const Type&): void
+insertFirst(const Type&): void
+insertLast(const Type&): void
+deleteNode(const Type&): void
linkedListType
orderedLinkedList

```
template <class elemType>
int arrayListType<elemType>::seqSearch(const elemType& item) const
{
    int loc;
    bool found = false;

    for (loc = 0; loc < length; loc++)
        if (list[loc] == item)
        {
            found = true;
            break;
        }

    if (found)
        return loc;
    else
        return -1;
} //end seqSearch
```

Sequential Search Analysis

- Examine effect of `for` loop in `seqSearch` of Array-based lists (page 499)
- Different programmers might implement same algorithm differently
  - Number of key comparisons: typically the same
- Computer speed affects performance
  - Does not affect the number of key comparisons

- Exercise: recursive sequential search

Sequential Search Analysis (cont'd.)

- Sequential search algorithm performance
  - Examine worst case and average case
  - Count number of key comparisons
- Unsuccessful search
  - Search item not in list
  - Make *n* comparisons
- Successful search – depending on the location
  - Best case: make one key comparison
  - Worst case: algorithm makes *n* comparisons

Sequential Search Analysis (cont'd.)

- Determining the average number of comparisons
  - Consider all possible cases
  - Find number of comparisons for each case
  - Add number of comparisons, divide by number of cases

$$\frac{1+2+\ldots+n}{n}$$

It is known that

$$1+2+\cdots+n=\frac{n(n+1)}{2}$$

Therefore, the following expression gives the average number of comparisons made by the sequential search in the successful case:

$$\frac{1+2+\ldots+n}{n}=\frac{1}{n}\frac{n(n+1)}{2}=\frac{n+1}{2}$$

O(n)

Sequential Search Analysis (cont'd.)

TABLE 3-1 Time complexity of array-based list operations

Function	Time-complexity
`isEmpty`	$O(1)$
`isFull`	$O(1)$
`listSize`	$O(1)$
`maxListSize`	$O(1)$
`print`	$O(n)$
`isItemAtEqual`	$O(1)$
`insertAt`	$O(n)$
`insertEnd`	$O(1)$
`removeAt`	$O(n)$
`retrieveAt`	$O(1)$
`replaceAt`	$O(n)$
`clearList`	$O(1)$
`constructor`	$O(1)$
`destructor`	$O(1)$
`copy constructor`	$O(n)$
`overloading the assignment operator`	$O(n)$
`seqSearch`	$O(n)$
`insert`	$O(n)$
`remove`	$O(n)$

Sequential Search Analysis (cont'd.)

TABLE 5-7 Time-complexity of the operations of the `class unorderedLinkedList`

Function	Time-complexity
`search`	$O(n)$
`insertFirst`	$O(1)$
`insertLast`	$O(1)$
`deleteNode`	$O(n)$

Sequential Search Analysis (cont'd.)

TABLE 5-8 Time-complexity of the operations of the `class orderedLinkedList`

Function	Time-complexity
`search`	$O(n)$
`insert`	$O(n)$
`insertFirst`	$O(n)$
`insertLast`	$O(n)$
`deleteNode`	$O(n)$

Can we do better?

Ordered Lists

- Elements ordered according to some criteria
  - Usually ascending order
- Operations
  - Same as those on an unordered list
    - Determining if list is empty or full, determining list length, printing the list, clearing the list
- Defining ordered list as an abstract data type (ADT)
  - Use inheritance to derive the class to implement the ordered lists from `arrayListType` or `linkedListType`

Ordered Lists (cont'd.)

```
template <class elemType>
class orderedArrayListType: public arrayListType<elemType>
{
public:
    orderedArrayListType(int size = 100);
      //constructor

   ...
      //We will add the necessary members as needed.

private:
    //We will add the necessary members as needed.
}
```

```
template <class elemType>
class orderedLinkedListType: public linkedListType<elemType>
{
public:
    ...
}
```

Binary Search

- Performed only on ordered lists
- divide-and-conquer technique

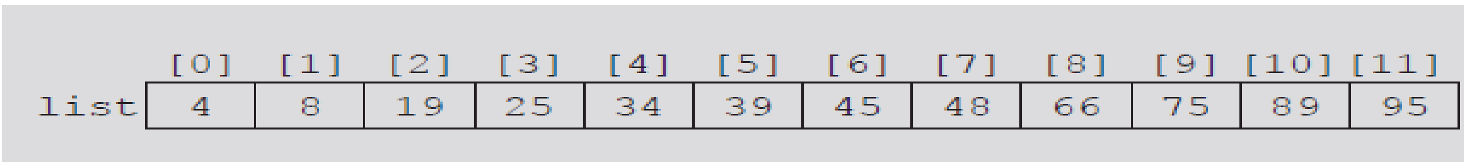


FIGURE 9-1 List of length 12. Searching for 75 in the list

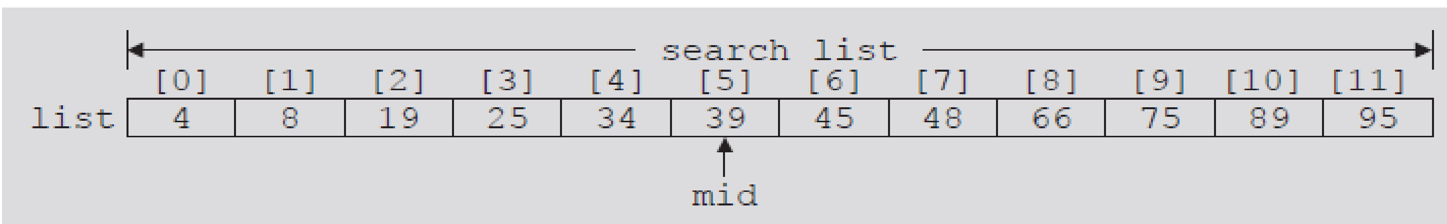


FIGURE 9-2 Search `list: list[0]...list[11]`

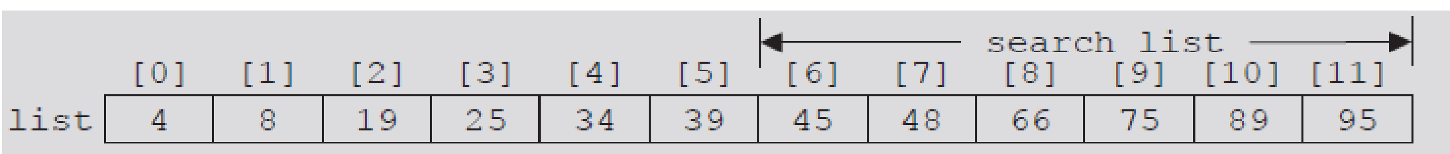


FIGURE 9-3 Search `list: list[6]...list[11]`

Binary Search (cont'd.)

- C++ function implementing binary search algorithm

```
template<class elemType>
int orderedArrayListType<elemType>::binarySearch
                                  (const elemType& item) const
{
   int first = 0;
   int last = length - 1;
   int mid;
   bool found = false;

   while (first <= last && !found)
   {
      mid = (first + last) / 2;

      if (list[mid] == item)
         found = true;
      else if (list[mid] > item)
         last = mid - 1;
      else
         first = mid + 1;
   }

   if (found)
      return mid;
   else
      return -1;
}//end binarySearch
```

Each iteration:
- Unsuccessful case: 2 key comparisons
- Successful case: 1 key comparison

Binary Search (cont'd.)

- Example 9-1. Searching for 89

	[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]	[11]
list	4	8	19	25	34	39	45	48	66	75	89	95

FIGURE 9-4 Sorted list for a binary search

Iteration	`first`	`last`	Mid	`list[mid]`	Number of comparisons
1	0	11	5	39	2
2	6	11	8	66	2
3	9	11	10	89	1 (`found` is `true`)

TABLE 9-1 Values of `first`, `last`, and `mid` and the number of comparisons for search item 89

Binary Search (cont'd.)

- Searching for 34

	[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]	[11]
list	4	8	19	25	34	39	45	48	66	75	89	95

FIGURE 9-4 Sorted list for a binary search

TABLE 9-2 Values of `first`, `last`, and `mid` and the number of comparisons for search item 34

Iteration	`first`	`last`	`mid`	`list[mid]`	Number of comparisons
1	0	11	5	39	2
2	0	4	2	19	2
3	3	4	3	25	2
4	4	4	4	34	1 (`found` is `true`)

Binary Search (cont'd.)

- Searching for 22

	[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]	[11]
list	4	8	19	25	34	39	45	48	66	75	89	95

FIGURE 9-4 Sorted list for a binary search

TABLE 9-3 Values of `first`, `last`, and `mid` and the number of comparisons for search item 22

Iteration	`first`	`last`	`mid`	`list[mid]`	Number of comparisons
1	0	11	5	39	2
2	0	4	2	19	2
3	3	4	3	25	2
4	3	2	The loop stops (because `first` > `last`)		

Insertion into an Ordered List

- After insertion: resulting list must be ordered
  - Find place in the list to insert item
    - Use algorithm similar to binary search algorithm
  - Slide list elements one array position down to make room for the item to be inserted
  - Insert the item
    - Use function `insertAt` of the class `arrayListType`

Insertion into an Ordered List (cont'd.)

- Algorithm to insert the item
- Function `insertOrd` implements algorithm

1. Use an algorithm similar to the binary search algorithm to find the place where the item is to be inserted.
2. `if` the item is already in this list
   `output` an appropriate message
   `else`
   use the function `insertAt` to insert the item in the list.

Insertion into an Ordered List (cont'd.)

- Add binary search algorithm and the `insertOrd` algorithm to the `class orderedArrayListType`

```
template <class elemType>
class orderedArrayListType: public arrayListType<elemType>
{
public:
    void insertOrd(const elemType&);
    int binarySearch(const elemType& item) const;
    orderedArrayListType(int size = 100);
};
```

```
template <class elemType>
void orderedArrayListType<elemType>::insertOrd(const elemType& item)
{
    int first = 0;
    int last = length - 1;
    int mid;

    bool found = false;

    if (length == 0)   //the list is empty
    {
        list[0] = item;
        length++;
    }
    else if (length == maxSize)
        cerr << "Cannot insert into a full list." << endl;
    else
    {
        while (first <= last && !found)
        {
            mid = (first + last) / 2;

            if (list[mid] == item)
                found = true;
            else if (list[mid] > item)
                last = mid - 1;
            else
                first = mid + 1;
        }//end while

        if (found)
            cerr << "The insert item is already in the list. "
                 << "Duplicates are not allowed." << endl;
        else
        {
            if (list[mid] < item)
                mid++;

            insertAt(mid, item);
        }
    }
}//end insertOrd
```

Why?

Insertion into an Ordered List (cont'd.)

- `class orderedArrayListType`
  - Derived from `class arrayListType`
  - List elements of `orderedArrayListType`
    - Ordered
- Must override functions `insertAt` and `insertEnd` of `class arrayListType` in `class orderedArrayListType`
  - If these functions are used by an object of type `orderedArrayListType`, list elements will remain in order

Insertion into an Ordered List (cont'd.)

- Can also override function `seqSearch`
  - Perform sequential search on an ordered list
    - Takes into account that elements are ordered

TABLE 9-4 Number of comparisons for a list of length n

Algorithm	Successful search	Unsuccessful search
Sequential search	$(n + 1) / 2 = O(n)$	$n = O(n)$
Binary search	$2\log_2 n - 3 = O(\log_2 n)$	$2\log_2(n+1) = O(\log_2 n)$

- Exercise: recursive binary search

Lower Bound on Comparison-Based Search Algorithms

- Comparison-based search algorithms
  - Sequential search: order *n*
  - Binary search: order $log_2 n$

- Theorem: # of comparison needed for any comparison-based search algorithm >= $log_2(n+1)$
- Corollary: binary search is the optimal worst-case algorithm for solving comparison-based search problem

- Devising a search algorithm with order less than $log_2 n$
  - Cannot be comparison based

Hashing

- Algorithm of order one (on average)
- Requires data to be specially organized
  - Hash table
    - Helps organize data
    - Stored in an array
    - Denoted by *HT*
  - Hash function
    - Arithmetic function denoted by h
    - Applied to key X
    - Compute $h(X)$: read as h of X
    - $h(X)$ gives address of the item

Hashing (cont'd.)

- Organizing data in the hash table
  - Store data within the hash table (array)
  - Store data in linked lists
- Hash table *HT* divided into *b* buckets
  - $HT[0], HT[1], . . ., HT[b – 1]$
  - Each bucket capable of holding *r* items
  - Follows that $br = m$, where *m* is the size of *HT*
  - Generally $r = 1$
    - Each bucket can hold one item
- The hash function *h* maps key *X* onto an integer *t*
  - $h(X) = t$, such that $0 <= h(X) <= b – 1$

Hashing (cont'd.)

- See Examples 9-2 and 9-3
- *Synonym*
  - Occurs if $h(X_1) = h(X_2)$
    - Given two keys X_1 and X_2, such that $X_1 \neq X_2$
- *Overflow*
  - Occurs if bucket t full
- *Collision*
  - Occurs if $h(X_1) = h(X_2)$
    - Given X_1 and X_2 non-identical keys

EXAMPLE 9-2

Suppose there are six students a_1, a_2, a_3, a_4, a_5, a_6 in the Data Structures class and their IDs are a_1: 197354863; a_2: 933185952; a_3: 132489973; a_4: 134152056; a_5: 216500306; and a_6: 106500306.

Let $k_1 = 197354863$, $k_2 = 933185952$, $k_3 = 132489973$, $k_4 = 134152056$, $k_5 = 216500306$, and $k_6 = 106500306$.

Suppose that HT denotes the hash table and HT is of size 13 indexed 0, 1, 2, ..., 12.

Define the function h: $\{k_1, k_2, k_3, k_4, k_5, k_6\} \rightarrow \{0, 1, 2, \ldots, 12\}$ by $h(k_i) = k_i \% 13$. (Note that % denotes the mod operator.)

Now

$h(k_1) = h(197354863) = 197354863 \% 13 = 4$	$h(k_4) = h(134152056) = 134152056 \% 13 = 12$
$h(k_2) = h(933185952) = 933185952 \% 13 = 10$	$h(k_5) = h(216500306) = 216500306 \% 13 = 9$
$h(k_3) = h(132489973) = 132489973 \% 13 = 5$	$h(k_6) = h(106500306) = 106500306 \% 13 = 3$

Suppose $HT[b] \leftarrow a$ means "store the data of the student with ID a into $HT[b]$." Then

$HT[4] \leftarrow 197354863$	$HT[5] \leftarrow 132489973$	$HT[9] \leftarrow 216500306$
$HT[10] \leftarrow 933185952$	$HT[12] \leftarrow 134152056$	$HT[3] \leftarrow 106500306$

EXAMPLE 9-3

Suppose there are eight students in the class in a college and their IDs are 197354864, 933185952, 132489973, 134152056, 216500306, 106500306, 216510306, and 197354865. We want to store each student's data into *HT* in this order.

Let $k_1 = 197354864$, $k_2 = 933185952$, $k_3 = 132489973$, $k_4 = 134152056$, $k_5 = 216500306$, $k_6 = 106500306$, $k_7 = 216510306$, and $k_8 = 197354865$.

Suppose that *HT* denotes the hash table and *HT* is of size 13 indexed 0, 1, 2, …, 12.

Define the function h: $\{k_1, k_2, k_3, k_4, k_5, k_6, k_7, k_8\} \rightarrow \{0, 1, 2, \ldots, 12\}$ by $h(k_i) = k_i \% 13$. Now

$h(k_1) = 197354864 \% 13 = 5$	$h(k_4) = 134152056 \% 13 = 12$	$h(k_7) = 216510306 \% 13 = 12$
$h(k_2) = 933185952 \% 13 = 10$	$h(k_5) = 216500306 \% 13 = 9$	$h(k_8) = 197354865 \% 13 = 6$
$h(k_3) = 132489973 \% 13 = 5$	$h(k_6) = 106500306 \% 13 = 3$	

As before, suppose $HT[b] \leftarrow a$ means "store the data of the student with ID a into $HT[b]$." Then

$HT[5] \leftarrow 197354864$	$HT[12] \leftarrow 134152056$	$HT[12] \leftarrow 216510306$
$HT[10] \leftarrow 933185952$	$HT[9] \leftarrow 216500306$	$HT[6] \leftarrow 197354865$
$HT[5] \leftarrow 132489973$	$HT[3] \leftarrow 106500306$	

Hashing (cont'd.)

- Overflow and collision occur at same time
  - If *r* = 1 (bucket size = one)
- Choosing a hash function
  - Main objectives
    - Choose an easy to compute hash function
    - Minimize number of collisions
- If `HTSize` denotes the size of hash table (array size holding the hash table)
  - Assume bucket size = one
    - Each bucket can hold one item
    - Overflow and collision occur simultaneously

Hash Functions: Some Examples

- Mid-square
  - Compute by squaring the key, then using the appropriate number of bits from the middle, which usually depend on all characters of the key
- Folding
  - Keys are divided into equal parts, except the last parts, then add all the parts
- Division (modular arithmetic)
  - In C++
    - $h(X) = i_X \% HTSize;$
  - C++ function

```
int hashFunction(char *insertKey, int keyLength)
{
    int sum = 0;

    for (int j = 0; j < keyLength; j++)
        sum = sum + static_cast<int>(insertKey[j]);

    return (sum % HTSize);
} // end hashFunction
```

Collision Resolution

- Desirable to minimize number of collisions
  - Collisions unavoidable in reality
    - Hash function always maps a larger domain onto a smaller range
- Resize HT: rehash existing items in HT
- Collision resolution technique categories
  - Open addressing (closed hashing)
    - Data stored within the hash table
  - Chaining (open hashing)
    - Data organized in linked lists
    - Hash table: array of pointers to the linked lists

HTSize > # of items

HTSize can be smaller than # of items

Collision Resolution: Open Addressing

- Data stored within the hash table
  - For each key *X*, *h*(*X*) gives index in the array
    - Where item with key *X* *likely* to be stored
- Linear probing
  - Constant 1
  - Constant c
- Random probing
- Rehashing
- Quadratic Probing
- Double hashing

Open Addressing: Linear Probing

- Starting at location *t*
  - Search array sequentially to find next available slot
- Assume circular array
  - If lower portion of array full
    - Can continue search in top portion of array using mod operator
  - Starting at *t*, check array locations using probe sequence
    - t, $(t + 1)$ % $HTSize$, $(t + 2)$ % $HTSize$, . . ., $(t + j)$ % $HTSize$

Open Addressing: Linear Probing (cont'd.)

- The next array slot is given by
  - $(h(X) + j)$ % *HTSize* where j is the j^{th} probe
- See Example 9-4
- C++ code implementing linear programming

```
hIndex = hashFunction(insertKey);
found = false;

while (HT[hIndex] != emptyKey && !found)
    if (HT[hIndex].key == key)
        found = true;
    else
        hIndex = (hIndex + 1) % HTSize;

if (found)
    cerr << "Duplicate items are not allowed." << endl;
else
    HT[hIndex] = newItem;
```

EXAMPLE 9-4

Consider the students' IDs and the hash function given in Example 9-3. Then we know that

$h(197354864) = 5 = h(132489973)$	$h(134152056) = 12 = h(216510306)$	$h(106500306) = 3$
$h(933185952) = 10$	$h(216500306) = 9$	$h(197354865) = 6$

Using the linear probing, the array position where each student's data is stored is:

ID	h(ID)	(h(ID) + 1) % 13	(h(ID) + 2) % 13
197354864	5		
933185952	10		
132489973	5	6	
134152056	12		
216500306	9		
106500306	3		
216510306	12	0	
197354865	6	7	

Open Addressing: Linear Probing (cont'd.)

- Causes **clustering (primary clustering)**
  - More and more new keys would likely be hashed to the array slots already occupied

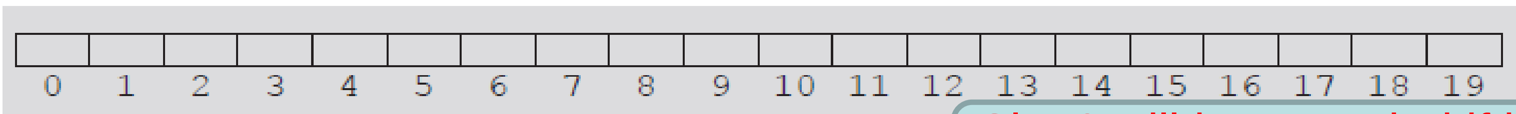


FIGURE 9-5 Hash table of size 20

Slot 9 will be occupied if h(X) = 6, 7, 8, or 9. Probability = 4/20

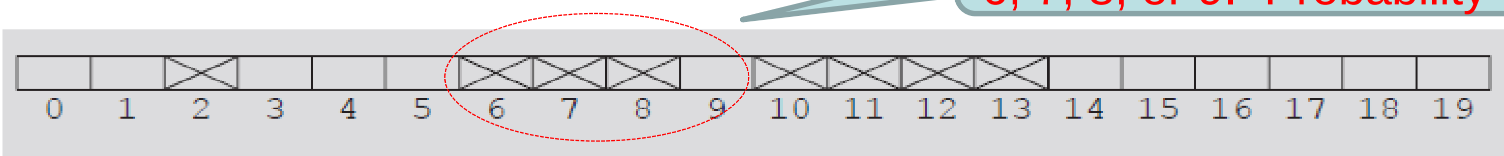


FIGURE 9-6 Hash table of size 20 with certain positions occupied

Probability that slot 14 will be occupied = 9/20

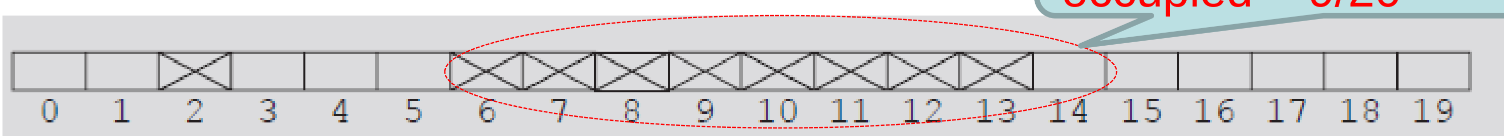


FIGURE 9-7 Hash table of size 20 with certain positions occupied

Open Addressing: Linear Probing (cont'd.)

- Improving linear probing
  - Skip array positions by fixed constant (c) instead of one
  - New hash address: $(h(X) + i * c) \% HTSize$
    - If $c = 2$ and $h(X) = 2k$, i.e., $h(X)$ even
      - Only even-numbered array positions visited
    - If $c = 2$ and $h(X) = 2k + 1$, i.e., $h(X)$ odd
      - Only odd-numbered array positions visited
    - To visit all the array positions
      - Constant c must be relatively prime to *HTSize*

Open Addressing: Random Probing

- Uses random number generator to find next available slot
  - i^{th} slot in probe sequence: $(h(X) + r_i)$ % *HTSize*
    - Where r_i is the i^{th} value in a random permutation of the numbers 1 to *HTSize* – 1
  - All insertions, searches use *same* random numbers sequence
  - Pros: Reduces primary clustering

Why?

EXAMPLE 9-5

Suppose that the size of the hash table is 101, and for the keys X_1 and X_2, $h(X_1) = 26$ and $h(X_2) = 35$. Also suppose that $r_1 = 2$, $r_2 = 5$, and $r_3 = 8$. Then the probe sequence of X_1 has the elements 26, 28, 31, and 34. Similarly, the probe sequence of X_2 has the elements 35, 37, 40, and 43.

Open Addressing: Rehashing

- If collision occurs with hash function h
  - Use a series of hash functions: $h_1, h_2, . . ., h_s$
  - If collision occurs at $h(X)$
    - Examine array slots $h_i(X)$, $1 <= i <= s$

Open Addressing: Quadratic Probing

- Suppose
  - Item with key X hashed at t , *i.e.*, $h(X) = t$ and $0 <= t <= HTSize - 1$
  - Position t already occupied
- Starting at position t
  - Linearly search array at locations
    - $(t + 1)\%\ HTSize$,
    - $(t + 2^2)\ \%\ HTSize = (t + 4)\ \%HTSize$,
    - $(t + 3^2)\ \%\ HTSize = (t + 9)\ \%\ HTSize$, . . .,
    - $(t + i^2)\ \%\ HTSize$
- Probe sequence: t, $(t + 1)\ \%\ HTSize$, $(t + 2^2)\ \%\ HTSize$, $(t + 3^2)\ \%\ HTSize$, . . ., $(t + i^2)\ \%\ HTSize$

EXAMPLE 9-6

Suppose that the size of the hash table is 101 and for the keys X_1, X_2, and X_3, $h(X_1) = 25$, $h(X_2) = 96$, and $h(X_3) = 34$. Then the probe sequence for X_1 is 25, 26, 29, 34, 41, and so on. The probe sequence for X_2 is 96, 97, 100, 4, 11, and so on. (Notice that $(96 + 3^2) \% 101 = 105 \% 101 = 4$.)

The probe sequence for X_3 is 34, 35, 38, 43, 50, 59, and so on. Even though element 34 of the probe sequence of X_3 is the same as the fourth element of the probe sequence of X_1, both probe sequences after 34 are different.

Open Addressing: Quadratic Probing (cont'd.)

- See Example 9-6
- Pros: reduces primary clustering
- Cons: does not probe all positions in the table
  - When *HTSize* is a prime, probes about half the table before repeating probe sequence
  - Collisions can be resolved if *HTSize* is a prime at least twice the number of items
  - Considerable number of probes
    - Assume full table
    - Stop insertion (and search)

Open Addressing: Quadratic Probing (cont'd.)

- Generating the probe sequence

$$\begin{aligned}
2^2 &= 1 + (2 \cdot 2 - 1) \\
3^2 &= 1 + 3 + (2 \cdot 3 - 1) \\
4^2 &= 1 + 3 + 5 + (2 \cdot 4 - 1) \\
&\vdots \\
i^2 &= 1 + 3 + 5 + 7 + \ldots + (2 \cdot i - 1), \quad i \geq 1.
\end{aligned}$$

Thus, it follows that

Why?

$$(t + i^2)\ \%\ HTSize = (t + 1 + 3 + 5 + 7 + \ldots + (2 \cdot i - 1))\ \%\ HTSize$$

Open Addressing: Quadratic Probing (cont'd.)

- Consider probe sequence
  - $t, t+1, t+2^2, t+3^2, \ldots, (t+i^2)$ % *HTSize*
  - C++ code computes i^{th} probe
    - $(t+i^2)$ % *HTSize*

```
int inc = 1;
int pCount = 0;
while (pCount < i) {
      t = (t + inc) % HTSize;
      inc = inc + 2;
      pCount++;
}
```

Open Addressing: Quadratic Probing (cont'd.)

- Pseudocode implementing quadratic probing

```
int pCount;
int inc;
int hIndex;

hIndex = hashFunction(insertKey);

pCount = 0;
inc = 1;

while (HT[hIndex] is not empty
       && HT[hIndex] is not the same as the insert item
       && pCount < HTSize / 2)
{
    pCount++;
    hIndex = (hIndex + inc ) % HTSize;
    inc = inc + 2;
}

if (HT[hIndex] is empty)
    HT[hIndex] = newItem;
else if (HT[hIndex] is the same as the insert item)
    cerr << "Error: No duplicates are allowed." << endl;
else
    cerr << "Error: The table is full. "
         << "Unable to resolve the collisions." << endl;
```

Open Addressing: Quadratic Probing (cont'd.)

- Random, quadratic probings eliminate **primary clustering**
- **Secondary clustering**
  - If two non-identical keys (X_1 and X_2) hashed to same home position ($h(X_1) = h(X_2)$)
    - Same probe sequence followed for both keys
  - If hash function causes a cluster at a particular home position
    - Cluster remains under these probings
    - Not original key

Open Addressing: Double hashing

- Solve secondary clustering with double hashing
  - Use linear probing
    - Increment value: function of key
  - If collision occurs at *h*(*X*)
    - Probe sequence generation

$(h(X) + i \star g\ (X))\ \%\ HTSize$

where g is the second hash function, and $i = 0, 1, 2, 3, \ldots.$

If the size of the hash table is a prime p, then we can define g as follows:

$g(k) = 1+(k\ \%\ (p-2))$

- See Examples 9-7 and 9-8

Deletion: Open Addressing

- Problem: two keys R and R’ hash to the same HT index. What happens when we delete R and then search for R’
  - Same probe sequence
  - R cannot be deleted by marking its position as empty in HT
- Solution: Use two arrays (of the same size)
  - Array one: stores the data
  - Array two: `indexStatusList`
    - Indicates whether a position in hash table free (0), occupied (1), used previously (-1)
- See code on pages 521 and 522
  - Class template implementing hashing as an ADT
  - Definition of function `insert`

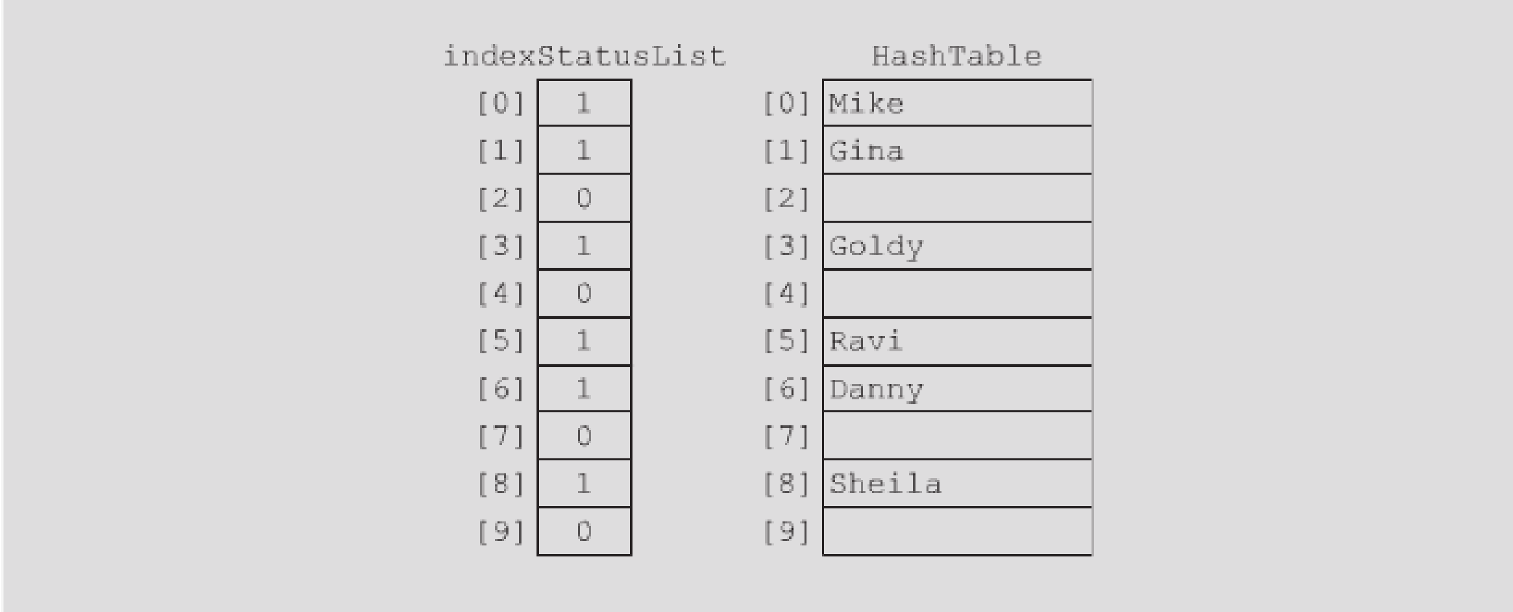


FIGURE 9-8 Hash table and `indexStatusList`

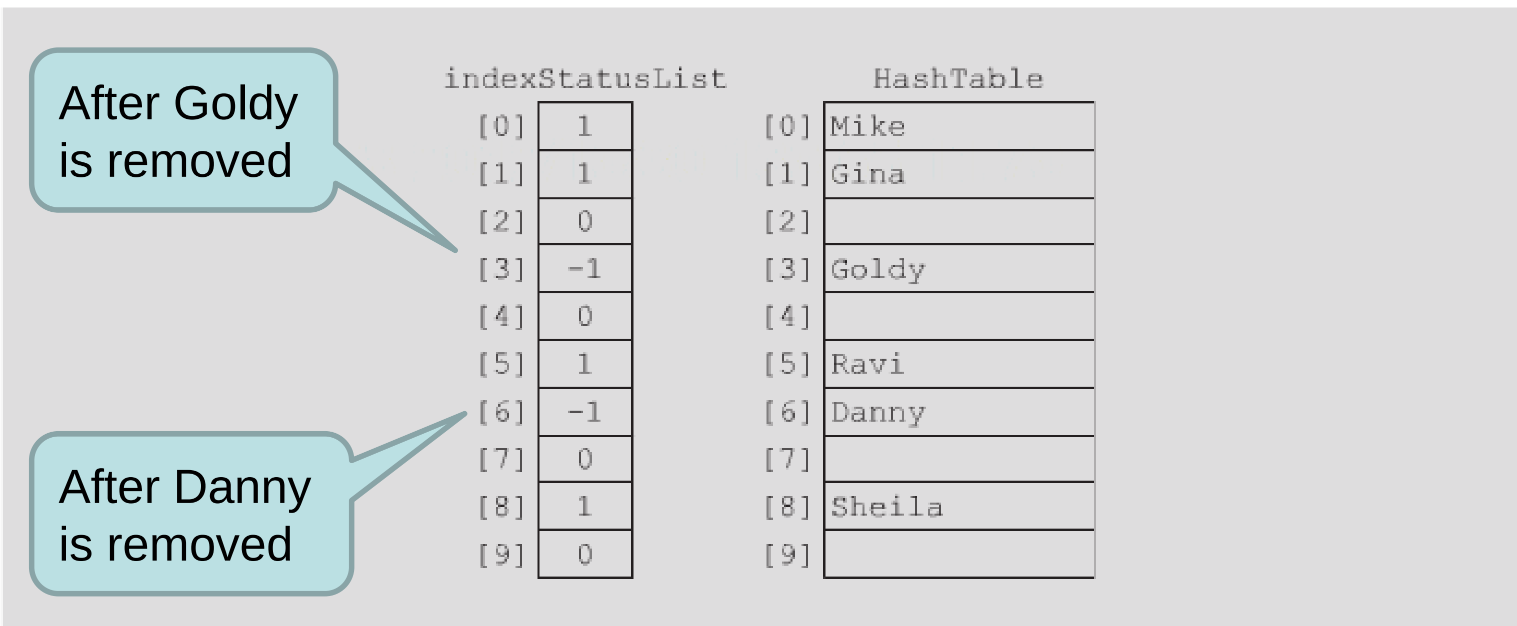
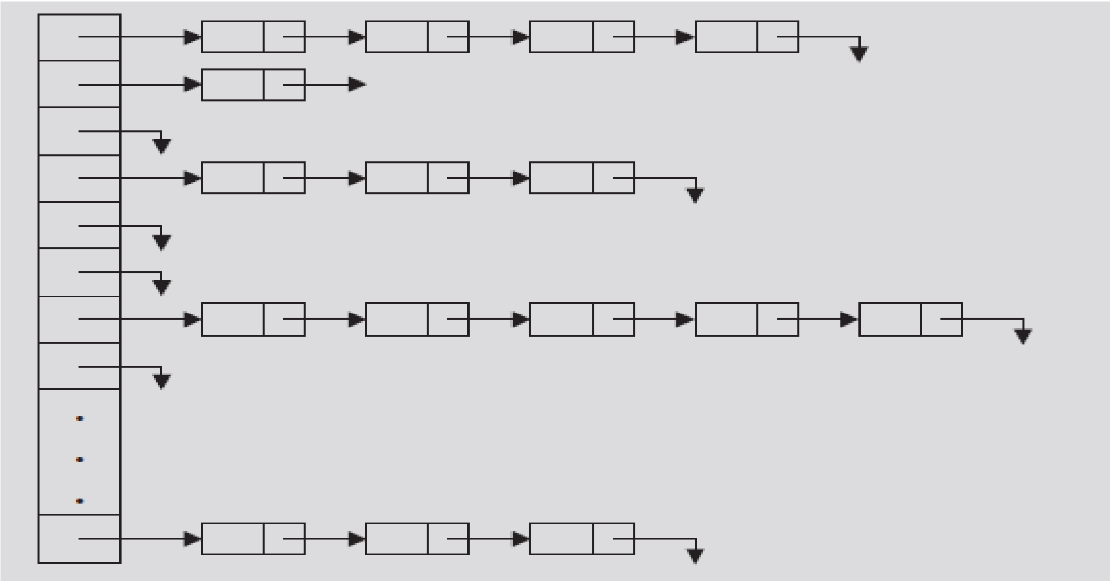


FIGURE 9-9 Hash table and `indexStatusList` after removing the entries at positions 3 and 6

Collision Resolution: Chaining (Open Hashing)

- Hash table *HT*: array of pointers
  - For each *j*, where 0 <= *j* <= *HTsize* -1
    - *HT*[*j*] is a pointer to a linked list
    - Hash table size (*HTSize*): less than or equal to the number of items

FIGURE 9-10 Linked hash table



Collision Resolution: Chaining (cont'd.)

- Item insertion and collision
  - For each key *X* (in the item)
    - First find $h(X) = t$, where $0 \leq t \leq HTSize - 1$
    - Item with this key inserted in linked list pointed to by $HT[t]$
  - For nonidentical keys X_1 and X_2
    - If $h(X_1) = h(X_2)$: Items with keys X_1 and X_2 inserted in same linked list
    - Collision handled quickly, effectively

Collision Resolution: Chaining (cont'd.)

- Search
  - Determine whether item *R* with key *X* is in the hash table
    - First calculate $h(X)$
  - Example: $h(X) = T$
    - Linked list pointed to by $HT[t]$ searched sequentially, or binary search
- Deletion
  - Delete item *R* from the hash table
    - Search hash table to find where in a linked list *R* exists
    - Adjust pointers at appropriate locations
    - Deallocate memory occupied by *R*

Collision Resolution: Chaining (cont'd.)

- Overflow: no longer a concern
  - Data stored in linked lists
  - Memory space to store data allocated dynamically
- Hash table size
  - No longer needs to be greater than number of items
- Hash table less than the number of items
  - Some linked lists contain more than one item
  - Good hash function has average linked list length still small (search is efficient)

Collision Resolution: Chaining (cont'd.)

- Advantages of chaining
  - Item insertion and deletion: straightforward
  - Efficient hash function
    - Few keys hashed to same home position
    - Short linked list (on average)
      - Shorter search length
    - If item size is large
      - Saves a considerable amount of space

Collision Resolution: Chaining (cont'd.)

- Disadvantage of chaining
  - Small item size wastes space
- Example: 1000 items each requires one word of storage
  - Chaining
    - Requires 3000 words of storage
  - Quadratic probing
    - If hash table size twice number of items: 2000 words
    - If table size three times number of items
      - Keys reasonably spread out
      - Results in fewer collisions

Hashing Analysis

- Load factor
  - Parameter α

$$\alpha = \frac{\text{Number of records in the table}}{HTSize}$$

TABLE 9-5 Average number of comparisons in hashing

	Successful search	Unsuccessful search
Linear probing	$\frac{1}{2}\left\{1+\frac{1}{1-\alpha}\right\}$	$\frac{1}{2}\left\{1+\frac{1}{(1-\alpha)^2}\right\}$
Quadratic probing	$\frac{-\log_2(1-\alpha)}{\alpha}$	$\frac{1}{1-\alpha}$
Chaining	$1+\frac{\alpha}{2}$	α

Summary

- Sequential search
  - Order *n*
- Ordered lists
  - Elements ordered according to some criteria
- Binary search
  - Order $log_2 n$
- Search analysis
  - Review number of key comparisons
  - Worst case, best case, average case

Summary (cont'd.)

- Hashing: Data organized using a hash table
- Hash functions
- Primary/secondary clustering
- Collision resolution technique categories
  - Open addressing (closed hashing)
    - Linear probing
    - Random probing
    - Rehashing
    - Quadratic Probing
    - Double hashing
  - Chaining (open hashing)

Self Exercises

- Programming Exercises: 1, 2, 3, 6, 8