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
- Can use `class arrayListType` (Chapter 3)
  - Implements a list and basic operations in an array

Sequential Search

- Array-based lists
  - Covered in Chapter 3
- Linked lists
  - Covered in Chapter 5
- Works the same for array-based lists and linked lists
- See code on page 499

Sequential Search Analysis

- Examine effect of `for` loop in code on page 499
- Different programmers might implement same algorithm differently
- Computer speed affects performance

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
- Conducting algorithm performance analysis
  - 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

Sequential Search Analysis (cont'd.)

- Determining the average number of comparisons (cont'd.)

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

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:

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

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 `class arrayListType`
  - Define two classes

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
- Uses divide-and-conquer technique

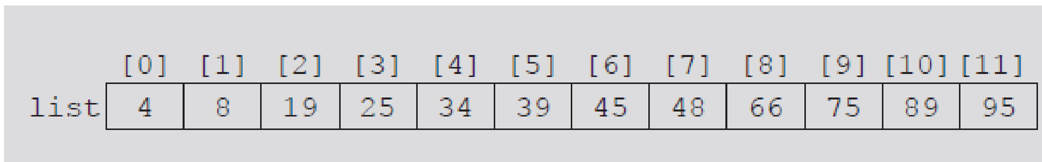


FIGURE 9-1 List of length 12

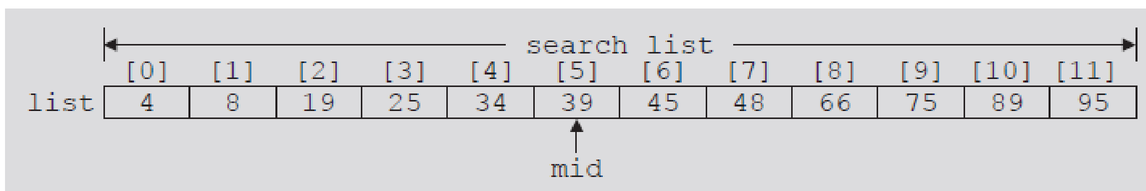


FIGURE 9-2 Search `list, list[0]...list[11]`

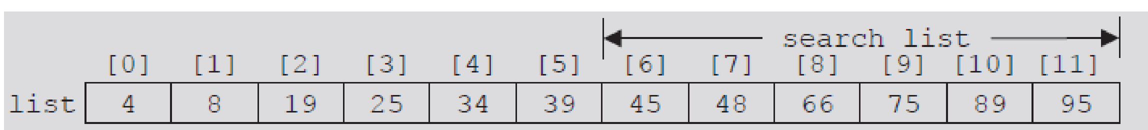


FIGURE 9-3 Search `list, list[6]...list[11]`

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
```

Binary Search (cont'd.)

- Example 9-1

	[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.)

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`)

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` (`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.

```
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
```

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);
};
```

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	$2\log_2 n - 3 = O(\log_2 n)$	$2\log_2(n+1) = O(\log_2 n)$

Lower Bound on Comparison-Based Search Algorithms

- Comparison-based search algorithms
  - Search list by comparing target element with list elements
- Sequential search: order *n*
- Binary search: order $log_2 n$

Lower Bound on Comparison-Based Search Algorithms (cont'd.)

- Devising a search algorithm with order less than $log_2 n$
  - Obtain lower bound on number of comparisons
- Cannot be comparison based

Theorem: Let L be a list of size $n > 1$. Suppose that the elements of L are sorted. If $\mathrm{SRH}(n)$ denotes the minimum number of comparisons needed, in the worst case, by using a comparison-based algorithm to recognize whether an element x is in L, then $\mathrm{SRH}(n) \geq \log_2(n + 1)$.

Corollary: The binary search algorithm is the optimal worst-case algorithm for solving search problems by the comparison method.

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 nonidentical keys

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
- Folding
- 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
- 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

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

- 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*

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;
```

Linear Probing (cont'd.)

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

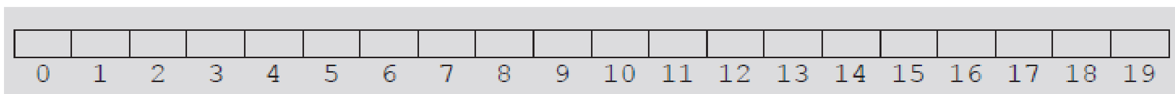


FIGURE 9-5 Hash table of size 20

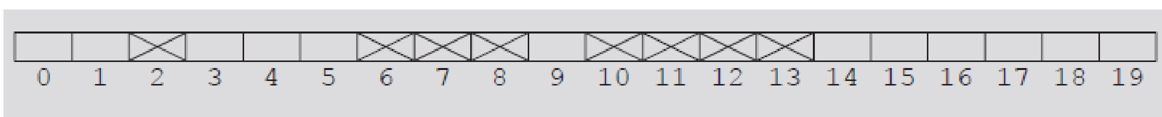


FIGURE 9-6 Hash table of size 20 with certain positions occupied

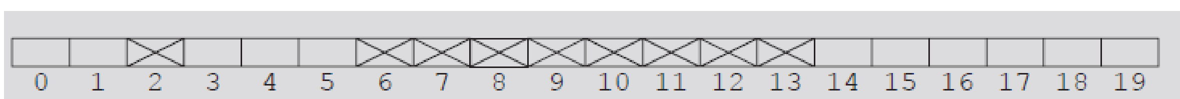


FIGURE 9-7 Hash table of size 20 with certain positions occupied

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)* = 2*k* (*h(X)* even)
      - Only even-numbered array positions visited
    - If *c* = 2 and *h(X)* = 2*k* + 1, (*h(X)* odd)
      - Only odd-numbered array positions visited
    - To visit all the array positions
      - Constant *c* must be relatively prime to *HTSize*

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
- See Example 9-5

Rehashing

- If collision occurs with hash function h
  - Use a series of hash functions: $h_1, h_2, \ldots, h_s$
  - If collision occurs at $h(X)$
    - Array slots $h_i(X)$, $1 <= h_i(X) <= s$ examined

Quadratic Probing

- Suppose
  - Item with key X hashed at t $(h(X) = t$ 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, $(t + 1)\ \%\ HTSize$ $(t + 2^2)\ \%\ HTSize$, $(t + 3^2)\ \%\ HTSize$, . . ., $(t + i^2)\ \%\ HTSize$

Quadratic Probing (cont'd.)

- See Example 9-6
- Reduces primary clustering
- Does not probe all positions in the table
  - Probes about half the table before repeating probe sequence
    - When *HTSize* is a prime
  - Considerable number of probes
    - Assume full table
    - Stop insertion (and search)

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

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

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 (p < i)
{
    t = (t + inc) % HTSize;
    inc = inc + 2;
    pCount++;
}
```

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;
```

Quadratic Probing (cont'd.)

- Random, quadratic probings eliminate primary clustering
- Secondary clustering
  - Random, quadratic probing functions of home positions
    - Not original key

Quadratic Probing (cont'd.)

- Secondary clustering (cont'd.)
  - If two nonidentical 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

Quadratic Probing (cont'd.)

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

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

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

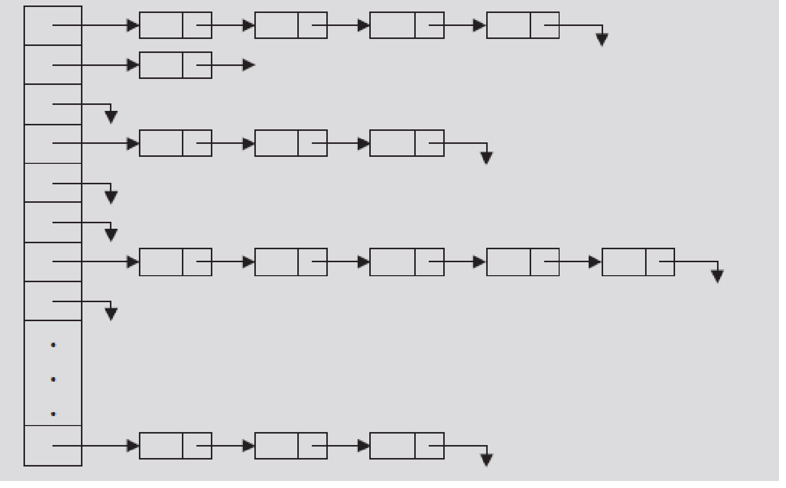
Deletion: Open Addressing

- Designing a class as an ADT
  - Implement hashing using quadratic probing
- Use two arrays
  - One stores the data
  - One uses `indexStatusList` as described in the previous section
    - Indicates whether a position in hash table free, occupied, used previously
- See code on pages 521 and 522
  - Class template implementing hashing as an ADT
  - Definition of function `insert`

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 <= *t* <= *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
- 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 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 $log_2 n$
- Hashing
  - Data organized using a hash table
  - Apply hash function to determine if item with a key is in the table
  - Two ways to organize data

Summary (cont'd.)

- Hash functions
  - Mid-square
  - Folding
  - Division (modular arithmetic)
- Collision resolution technique categories
  - Open addressing (closed hashing)
  - Chaining (open hashing)
- Search analysis
  - Review number of key comparisons
  - Worst case, best case, average case