Chapter 9
Search Algorithms

Chapter 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

Sequential Search

```cpp
template<class elemType>
int arraylistType<elemType>::seqSearch(const elemType& item)
{
    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;
}
```

What is the time complexity?

Search Algorithms

Suppose that there are \( n \) elements in the list. The following expression gives the average number of comparisons, assuming that each element is equally likely to be sought:

\[
\frac{1 + 2 + \ldots + n}{n}
\]

It is known that

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

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

\[
\frac{1 + 2 + \ldots + n}{n} = \frac{n(n + 1)}{2} - \frac{n+1}{2}
\]

Search Algorithms

• Search item: target
• To determine the average number of comparisons in the successful case of the sequential search algorithm:
  – Consider all possible cases
  – Find the number of comparisons for each case
  – Add the number of comparisons and divide by the number of cases

Binary Search

(assumes list is sorted)

```
// Fig. 9-1: List of Length 12

// Fig. 9-2: Search LSD, LSD(1) . . . LSD(11)
```


Binary Search: middle element

\[
\text{mid} = \frac{\text{first} + \text{last}}{2}
\]

Binary Search: Example

```
template<class elemType>
int orderedArrayListType<elemType>::binarySearch(const elemType& item)
{
    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;
}
```

Performance of Binary Search
Performance of Binary Search

- **Unsuccessful search**
  - for a list of length \( n \), a binary search makes approximately \( 2 \log_2 (n + 1) \) key comparisons

- **Successful search**
  - for a list of length \( n \), on average, a binary search makes \( 2 \log_2 n - 4 \) key comparisons
  - Worst case upper bound: \( 2 + 2 \log_2 n \)

Search Algorithm Analysis Summary

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Successful Search</th>
<th>Unsuccessful Search</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sequential search</td>
<td>( \frac{n+1}{2} = O(n) )</td>
<td>( n = O(n) )</td>
</tr>
<tr>
<td>Binary search</td>
<td>( 2 \log_2 n - 4 = O(\log n) )</td>
<td>( 2 \log_2 (n+1) = O(\log n) )</td>
</tr>
</tbody>
</table>

Lower Bound on Comparison-Based Search

- **Definition**: A comparison-based search algorithm performs its search by repeatedly comparing the target element to the list elements.

- **Theorem**: Let \( L \) be a list of size \( n > 1 \). Suppose that the elements of \( L \) are sorted. If \( \text{SRH}(n) \) 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 \( \text{SRH}(n) = \log_2 (n + 1) \).
  - If list not sorted, worst case is \( n \) comparisons

- **Corollary**: The binary search algorithm is the optimal worst-case algorithm for solving search problems by the comparison method (when the list is sorted).
  - For unsorted lists, sequential search is optimal

Hashing

- An alternative to comparison-based search
- Requires storing data in a special data structure, called a **hash table**
- Main objectives to choosing hash functions:
  - Choose a hash function that is easy to compute
  - Minimize the number of collisions

Commonly Used Hash Functions

- **Mid-Square**
  - Hash function, \( h \), computed by squaring the identifier
  - Using appropriate number of bits from the middle of the square to obtain the bucket address
  - Middle bits of a square usually depend on all the characters, it is expected that different keys will yield different hash addresses with high probability, even if some of the characters are the same
Commonly Used Hash Functions

Suppose that each key is a string. The following C++ function uses the division method to compute the address of the key:

```cpp
int hashFunction(char *key, int keyLength)
{
    int sum = 0;
    for(int j = 0; j <= keyLength; j++)
        sum = sum + static_cast<int>(key[j]);
    return (sum % HTSize);
} // end hashFunction
```

Collision Resolution

- Algorithms to handle collisions
- Two categories of collision resolution techniques
  - Open addressing (closed hashing)
  - Chaining (open hashing)

Collision Resolution: Open Addressing

Pseudocode implementing linear probing:

```cpp
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;
```

Random Probing

- Uses a random number generator to find the next available slot
- \(i\)th slot in the probe sequence is: \((h(X) + r) \mod HTSize\) where \(r\) is the \(i\)th value in a random permutation of the numbers 1 to \(HTSize - 1\)
- All insertions and searches use the same sequence of random numbers

Quadratic Probing

- \(i\)th slot in the probe sequence is: \((h(X) + i^2) \mod HTSize\) (start \(i\) at 0)
- Reduces primary clustering of linear probing
- We do not know if it probes all the positions in the table
- When \(HTSize\) is prime, quadratic probing probes about half the table before repeating the probe sequence
Deletion: Open Addressing

- When deleting, need to remove the item from its spot, but cannot reset it to empty (Why?)

Collision Resolution: Chaining (Open Hashing)

- No probing needed; instead put linked list at each hash position

Hashing Analysis

Let

\[ \alpha = \frac{\text{Number of records in the table}}{\text{Table size}} \]

Then \( \alpha \) is called the **load factor**

Average Number of Comparisons

- Linear probing:
  - Successful search: \( \frac{1}{2} \left( 1 + \frac{1}{1 - \alpha} \right) \)
  - Unsuccessful search: \( \frac{1}{2} \left( 1 + \frac{1}{1 - \alpha} \right) \)

- Quadratic probing:
  - Successful search: \( \frac{-\log(1 - \alpha)}{\alpha} \)
  - Unsuccessful search: \( \frac{1}{1 - \alpha} \)
### Searching Lists

- There are many instances when one is interested in storing and searching a list:
  - A phone company wants to provide caller ID: Given a phone number a name is returned.
  - Somebody who plays the Lotto thinks they can increase their chance to win if they keep track of the winning number from the last 3 years.
- Both of these have simple solutions using arrays, linked lists, binary trees, etc.
- Unfortunately, none of these solutions is adequate, as we will see.

### Hashing

#### Hash Table Example

- I have 5 friends whose phone numbers I want to store in a hash table of size 8. Their names and numbers are:
  - Susy Olson: 555-1212
  - Sarah Skillet: 555-4321
  - Ryan Hamster: 545-3241
  - Justin Case: 555-6798
  - Chris Lindmeyer: 535-7869
- I use as a hash function \( h(k) = k \mod 8 \), where \( k \) is the phone number viewed as a 7-digit decimal number.
- Notice that \( 5554321 \mod 8 = 5453241 \mod 8 = 1 \).
- But I can’t put Sarah Skillet and Ryan Hamster both in the position 1.
- Can we fix this problem?

#### Hash Table Problems

- The problem with hash tables is that two keys can have the same hash value. This is called **collision**.
- There are several ways to deal with this problem:
  - Pick the hash function \( h \) to minimize the number of collisions.
  - Implement the hash table in a way that allows keys with the same hash value to all be stored.
- The second method is almost always needed, even for very good hash functions. Why?
- We will talk about 2 collision resolution techniques:
  - Chaining
  - Open Addressing
- But first, a bit more on hash functions

#### Hash Functions

- Most hash functions assume the keys come from the set \( \{0, 1, \ldots \} \).
- If the keys are not natural numbers, some method must be used to convert them:
  - Phone number and ID numbers can be converted by removing the hyphens.
  - Characters can be converted using ASCII.
  - Strings can be converted by converting each character using ASCII and then interpreting the string of natural numbers as if it were stored base 128.
- We need to choose a good hash function.
- A hash function is good if:
  - it can be computed quickly, and
  - the keys are distributed uniformly throughout the table.

#### Problematic List Searching

Here are some solutions to the problems:

- **For caller ID:**
  - Use an array indexed by phone number.
  - Requires one operation to return name.
  - An array of size 1,000,000,000 is needed.
  - Use a linked list.
  - This requires \( O(n) \) time and space, where \( n \) is the number of actual phone numbers.
  - This is the best we can do space-wise, but the time required is horrendous.
  - A balanced binary tree: \( O(n) \) space and \( O(\log n) \) time. Better, but not good enough.
- **For the Lotto** we could use the same solutions.
  - The number of possible lotto numbers is 15,625,000 for a 6/50 lotto.
  - The number of entries stored is on the order of 1000, assuming a daily lotto.

#### The Hash Table Solution

- A **hash table** is similar to an array:
  - Has fixed size \( m \).
  - An item with key \( k \) goes into index \( h(k) \), where \( h \) is a function from the keyspace to \( \{0, 1, \ldots, m-1\} \).
- The function \( h \) is called a **hash function**.
- We call \( h(k) \) the **hash value** of \( k \).
- A **hash table** allows us to store values from a large set in a small array in such a way that searching is fast.
- The space required to store \( n \) numbers in a hash table of size \( m \) is \( O(n + m) \). Notice that this does not depend on the size of the keyspace.
- The average time required to insert, delete, and search for an element in a hash table is \( O(1) \).
- Sounds perfect. Then what's the problem? Let's look at an example.

\( \begin{array}{ll}
S_1 & 555-4321 \\
S_2 & 555-1212 \\
S_3 & 555-6798 \\
S_4 & 545-3241 \\
S_5 & 535-7869 \\
\end{array} \)
**Some Good Hash Functions**

- **The division method:**
  
  \[ h(k) = k \mod m \]
  
  - The modulus \( m \) must be chosen carefully.
  - Powers of 2 and 10 can be bad. Why?
  - Prime numbers not too close to powers of 2 are a good choice.
  - We can pick \( m \) by choosing an appropriate prime number that is close to the table size we want.

- **The multiplication method:** Let \( A \) be a constant with \( 0 < A < 1 \). Then we use
  
  \[ h(k) = \lfloor m(kA \mod 1) \rfloor, \]
  
  where \( \lfloor kA \mod 1 \rfloor \) means the fractional part of \( kA \).
  - The choice of \( m \) is not as critical here.
  - We can choose \( m \) to make the implementation easy and/or fast.

**Universal Hashing**

- Let \( x \) and \( y \) be distinct keys.

- Let \( \mathcal{H} \) be a set of hash functions.

- \( \mathcal{H} \) is called **universal** if the number of functions \( h \in \mathcal{H} \) for which \( h(x) = h(y) \) is precisely \( |\mathcal{H}|/m \).

- In other words, if we pick a random \( h \in \mathcal{H} \), the probability that \( x \) and \( y \) collide under \( h \) is \( 1/m \).

- Universal hashing can be useful in many situations.
  - You are asked to come up with a hashing technique for your boss.
  - Your boss tells you that after you are done, he will pick some keys to hash.
  - If you get too many collisions, he will fire you.
  - If you use universal hashing, the only way he can succeed is by getting lucky. Why?

**Collision Resolution: Chaining**

- With **chaining**, we set up an array of links, indexed by the hash values, to lists of items with the same hash value.

- Let \( n \) be the number of keys, and \( m \) the size of the hash table. Define the **load factor** \( \alpha = n/m \).

- Successful and unsuccessful searching both take time \( \Theta(1 + \alpha) \) on average, assuming simple uniform hashing.

- By **simple uniform hashing** we mean that a key has equal probability to hash into any of the \( m \) slots, independent of the other elements of the hash table.

**Collision Resolution: Open Addressing**

- With Open Addressing, all elements are stored in the hash table.

- This means several things
  - Less memory is used than with chaining since we don’t have to store pointers.
  - The hash table has an absolute size limit of \( m \).
  - Thus, planning ahead is important when using open addressing.
  - We must have a way to store multiple elements with the same hash value.

- We define our hash functions with an extra parameter, the **probe number** \( i \). Thus our hash functions look like \( h(k, i) \).

- We compute \( h(k, 0), h(k, 1), \ldots, h(k, i) \) until \( h(k, i) \) is empty.

- The hash function \( h \) must be such that the **probe sequence** \( (h(k, 0), \ldots, h(k, m - 1)) \) is a permutation of \( (0, 1, \ldots, m - 1) \). Why?

- Insertion and searching are fairly quick, assuming a good implementation.

- Deletion can be a problem because the probe sequence can be complex.

- We will discuss 2 ways of defining probe sequences:
  - Linear Probing
  - Double Hashing

**Universal Hashing**

- Let \( x \) and \( y \) be distinct keys.

- Let \( \mathcal{H} \) be a set of hash functions.

- \( \mathcal{H} \) is called **universal** if the number of functions \( h \in \mathcal{H} \) for which \( h(x) = h(y) \) is precisely \( |\mathcal{H}|/m \).

- In other words, if we pick a random \( h \in \mathcal{H} \), the probability that \( x \) and \( y \) collide under \( h \) is \( 1/m \).

- Universal hashing can be useful in many situations.
  - You are asked to come up with a hashing technique for your boss.
  - Your boss tells you that after you are done, he will pick some keys to hash.
  - If you get too many collisions, he will fire you.
  - If you use universal hashing, the only way he can succeed is by getting lucky. Why?

**Linear Probing**

- Let \( g \) be a hash function.

- When we use linear probing, the probe sequence is computed by
  
  \[ h(k, i) = (g(k) + i) \mod m. \]

- When we insert an element, we start with the hash value, and proceed element by element until we find an empty slot.

- For searching, we start with the hash value and proceed element by element until we find the key we are looking for.

- **Example:** Let \( g(k) = k \mod 13 \). We will insert the following keys into the hash table:

  18, 41, 22, 44, 59, 32, 31, 73

- **Problem:** The values in the table tend to cluster.
**Double Hashing**
- With **double hashing** we use two hash functions.
- Let $h_1$ and $h_2$ be hash functions.
- We define our probe sequence by
  $$h(k, i) = (h_1(k) + i \times h_2(k)) \mod m.$$  
- Often pick $m = 2^n$ for some $n$, and ensure that $h_2(k)$ is always odd.
- Double hashing tends to distribute keys more uniformly than linear probing.

**Double Hashing Example**
- Let $h_1 = k \mod 13$
- Let $h_2 = 1 + (k \mod 8)$, and
- Let the hash table have size 13.
- Then our probe sequence is defined by
  $$h(k, i) = (h_1(k) + i \times h_2(k)) \mod 13$$
  $$= (k \mod 13 + i(1 + (k \mod 8))) \mod 13$$
- Insert the following keys into the table:
  18, 41, 22, 44, 59, 32, 31, 73

**Open Addressing: Performance**
- Again, we set $\alpha = n/m$. This is the average number of keys per array index.
- Note that $\alpha \leq 1$ for open addressing.
- We assume a good probe sequence has been used. That is, for any key, each permutation of $(0, 1, \ldots, m - 1)$ is equally likely as a probe sequence.
- The average number of probes for insertion or unsuccessful search is at most
  $$1/(1 - \alpha).$$
- The average number of probes for a successful search is at most
  $$\frac{1}{\alpha} \ln\frac{1}{1 - \alpha} + \frac{1}{\alpha}.$$