Chapter 8

Searching

TIMOTHY K. SHIH
Tamkang University, Taiwan
tshih@cs.tku.edu.tw

8.1. Introduction

In our previous discussion of lists, stacks, and queues, the purposes of these structures are designed to store data, which should be retrieved for frequent usage. Information retrieval is one of the interesting issues in computer engineering and computer science. The challenges of information retrieval include the efficiency of storage usage, as well as the computation speed, which is important for fast retrieval. Searching means to retrieve a record of data from a structure or from a database, which may contain hundreds or even several millions of records. The storage structure of these records is the essential factor, which affects the speed of search. However, a good searching strategy also influences the speed, even on the same underlying storage structure. In this chapter, we will discuss the strategies, which can be used for information retrieval on different storage structures.

What is the purpose of searching? It is to find the information for a particular application. For instance, in a bank account, the information will be the balance of the account. But, based on what information does a search program perform. That is, what information does the program seek in order to find out the correct balance? Of course, it is the account number of the bank customer. In a search strategy, *records* and *keys* are the fundamental concepts. A record is an aggregation of information, which can be represented by an abstract type (discussed in Chapter 2). A key is an element of the record, which will identify the record. Usually, each key in a

storage structure or a database is unique. For instance, the account number of bank customers cannot be duplicated. But, the content of records may include similar or the same information. For example, a person and his father can have separate accounts in the same bank. But, the father and the son may have the same address. A record should contain sufficient fields in order to hold necessary information. The following is a simplified record of a bank account:

Account number: integer number
Password: string
Name: string
Address: long string
Telephone number: integer number or string
Additional telephone number: integer number of string
Balance: float point number

Note that, it is not necessary for the bank application program to fill information in each record. For instance, the additional telephone number of the above record is optional. However, the key field (i.e. the account number) should always contain correct information.

The storage structure discussed in previous chapters (e.g. linked lists, or array) can be used to store records. Each record of a linked list or an array of records may contain the detailed information of a bank account. Linked lists and arrays are some of the fundamental structures to hold records. However, in this book, other sophisticated structures, such as binary trees, B-trees and graphs, can be used as well. In this chapter, we will discuss search strategies based on linked lists and arrays. Other search mechanisms for the particular structures will be discussed in each individual chapter.

Almost all programming languages have an implementation of the array data type. An array is a sequential collection of atomic elements or compound elements. Each element has an index. The index can identify an element, which is a similar concept to keys of a storage structure. However, index usually represents the physical address of an element. But, keys represent a logical address. Indices are sequential since memory (i.e. RAM) locations are continuous. Keys may or may not be sequential. If an array is implemented in a computer memory, the search strategy can be sequential or via some dynamic mechanism. In addition to array, linked lists can be used to store a sequence of atomic or compounded elements. Usually, linked lists are implemented in computer memory. And, an access strategy of linked lists should be sequential.

Another aspect of searching relates to the storage devices used to store information records. If the records are only stored in computer memory

(i.e. RAM), the strategy is an *internal search*. If the records are stored completely or partially in hard disks, tapes or CD ROMs, the mechanism is an *external search*. In this chapter, we only discuss internal search. However, some of the methods can be used in external search as well.

8.2. Sequential Search

The simplest search strategy is the sequential search mechanism. Imagine that, there is a pile of photos from which you want to search for a picture. You can start from the first photo until you find the one, or search till the last photo (the photo is missing). The strategy is sequential. When you are looking for the picture, you are making a *comparison* between a photo and the picture in your mind. Comparison of sequential search can be on a single key, or on a compounded key (includes several search criteria). In a sequential storage structure, there are two indices, the first and the last indices, of which the boundary of maximal search is formed. In addition, a sequential search strategy needs to maintain a *current index*, which represents the position of the current comparison (i.e. to the photo you are looking at). In this chapter, we present the algorithms of searching. It is relatively easy to implement the algorithm in any programming language. The algorithm of sequential search follows:

Precondition: **first** is the first index of the storage structure
last is the last index of the storage structure
current is an index variable
key is the search key to the target record
Postcondition: **target** is the index points to the record found.
Or, **target** = -1 if the record is not found
Algorithm:
Initial step: set **current** = **first**
Search step: repeat {
If key == retrieve_key(**current**) then
Set **target** = **current**
Exit
Set **current** = next(**current**)
} until **current** == -1
If **current** · -1 then
Print "record found", retrieve_record(**target**)
Else
Print "record not found"

The sequential search algorithm firstly set the initial value of the current index (i.e. **current**). It is necessary to know that, in any iteration process, such as the for-loop statement, the repeat-until statement, etc., there are three important factors to watch out. They are the initial process, the incremental process, and the termination process. The search algorithm sets **current** to **first** for the initial process, uses a next function to increase the index value, and checks the termination process with the "**current** == **−1**" condition. Note that, in the algorithm, the "==" sign means to compare its left operand to its right operand, and to return a Boolean value. The next function returns a "**−1**" if the iteration is out of the storage boundary. The function call retrieve_key(**current**) takes an index and returns a key associated with the index. Similarly, the function call retrieve_record(**target**) takes an index and returns the record of the index. If there is record found in the storage structure, the record is printed; otherwise, the algorithm prints an error message.

8.2.1. *Search on Linked List*

The implementation of the sequential search depends on the underlying storage structure. It is possible to implement the algorithm using any type of structure, as long as the search process visits each record sequentially. But, we only present the implementation using linked list and array. Other structures are associated with their own searching strategy, which may not be sequential as well. The implementation using linked list uses the following mechanisms. Since linked lists are using pointers in a language such as C++, indices are pointers. The next function can be implemented by updating the return value to point to the next record. However, the next function should be able to access the "**last**" value in order to check with the storage boundary. The retrieve_key function takes the pointer of a record, and returns the key of the record. Usually, it will use the membership function of the abstract data type associated with the record. The retrieve_record function uses the same strategy. But, the function returns a compound value of the entire record. The Print function, as a method of the abstract type, should be able to output the compound value. An example of a linked list is illustrated in Fig. 8.1:

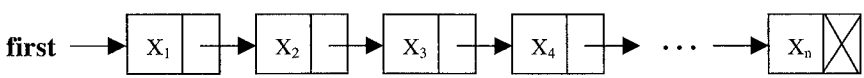

Fig. 8.1. A linked list.

Fig. 8.2. An array.

8.2.2. *Search on Array*

On the other hand, the sequential search implementation can be on an array. In this case, the next function can be an update of the **current** index. Since all of the indices are implemented using array indices, which are integers, it is relatively easier to trace or debug the program. The retrieve_key function can be replaced by an access to a record of array element. Similarly, the retrieve_record function gets values of the record. The algorithm used for array and for linked list should be the same. But, the implementation can be different.

The linked list implementation and the array implementation of the sequential search algorithm use the same strategy — sequential access of each record. However, it is not necessary for a search to be sequential, if array is used as a random access memory. However, if array is used as fast search storage, some preconditions should be established. In the next section, we discuss one of the faster search algorithms. An example of an array is illustrated in Fig. 8.2.

8.3. Binary Search

Suppose that, in your hand, you have a deck of student ID cards. And you want to search for a student. The worst case is you need to look at the cards one by one sequentially until the last student is found. This is sequential search. However, it is possible that you randomly look at some cards. If you are lucky, the student can be found in just a second. However, the random search mechanism does not guarantee you to find the student immediately. One of the methods that you can use to improve your luck but still perform reasonably is the *Binary Search* strategy. When you are checking a dictionary for a word, you assume it is in a location of the dictionary. Then, depending on where you tried, you will search forward or backward one or several times, until the word is found. The reason for searching forward and backward is that the words in a dictionary are sorted according to the alphabetical order. Conclusively, if we want such a fast search algorithm, two preconditions need to be established. First, the records should be sorted. Sorting algorithms will be discussed in Chapter 5.

Second, we need to "guess" for the starting point and decide which way to "guess" for the next search point, if necessary. In the following sections, we discuss the binary search algorithm. One condition to implement the binary search algorithm is the storage structure used should allow random access to every element. Therefore, linked list is not a suitable storage. Array is used for binary search.

8.3.1. *Binary Search Algorithm*

In Chapter 1, we discussed the concept of *recursion*. Recursion is a concept, which applies the same method to solve a problem, within a stepwise-decomposed domain, to find the solution. The process to find a student in a deck of ID cards can be recursively solved. Think about this. If the student number to be searched is smaller than the card picked, you do not need to care about the cards after the card has been currently selected. The deck of cards, before the card selected, may contain the target. The same procedure can be used to search the new deck of cards.

In order to realize such a search strategy, we need to have a boundary for search. Assuming that, a list of storage records is sorted. And we use **first** and **last** to bind the boundary, the recursive algorithm of binary search algorithm follows:

Precondition: **sorted_array** is an array of sorted records
first is the first index of the sorted array
last is the last index of the sorted array
current is an index variable
key is the search key to the target record
Postcondition: **target** is the index points to the record found.
Or, **target** = -1 if the record is not found
Algorithm: **recursive_bs(first, last)**

```
Search step:
    if first ≤ last {
        current = (first + last) / 2
        if key == retrieve_key(current) then
            Return current
        else if key > retrieve_key(current) then
            Return recursive_bs(current + 1, last)
        Else
            Return recursive_bs(first, current − 1)
        }
    else Return −1
```

```
Algorithm: binary_search
    Step: target = recursive_bs(first, last)
          If target ≠ −1 then
              Print "record found", retrieve_record(target)
          Else
              Print "record not found"
```

The recursive algorithm (i.e. **recursive_bs**) takes two parameters. Note that, in a programming language that supports recursion, each function call will maintain a local variable space, such as the two parameters passed to the function. Recursion in a run-time environment is implemented by using a stack. That is, each call to the function has its own space. Therefore, the **first** and the **last** variables of each function call have different values.

The position, which we "guess" is the middle of the sorted array. If the key is found, we return the position. Otherwise, the right or the left side of the array is searched depending on whether the key is greater than or less than the current value found. The recursive function is called by another function. If the return value indicates a useful target position, the record is printed. Otherwise, the error message is printed.

Recursive function is natural since it reflects the definition of functions. For instance, the factorial function is defined recursively. However, in a procedural programming language, which use for-loop or while-loop statements for iterative computing, it is easier to think and code iteratively. Moreover, recursive programming takes stack space in memory, as each call to the recursive function allocates a new local parameter space. In general, iterative programming is faster for commercial computers. In the following algorithm, we present an iterative version of the binary search algorithm:

```
Algorithm: iterative_bs
    Search step:
        While first ≤ last {
            current = (first + last) / 2
            if key == retrieve_key(current) then
                Return current
            else if key > retrieve_key(current) then
                first = current + 1
            Else
                last = current − 1
            }
    Return −1
```

```
Algorithm: binary_search
    Step: target = iterative_bs
        If target ≠ −1 then
            Print "record found", retrieve_record(target)
        Else
            Print "record not found"
```

The iterative version does not use parameters. However, the **first** and the **last** variables of the array indices will be changed if necessary. The two versions of algorithms can be implemented in any programming language, except that the recursive version needs to run on a language, which supports recursion. The two versions produce the same result for the same array. They also perform the same search sequence. In the following example, suppose we want to search for the record with a key of 9, the content of the record, "Jason" will be returned. And the search order is presented as shown in Table 8.1.

The values of indices of the iteration is shown in Table 8.2.

The search example takes three iterations to find "Jason". It is the worst cast of such a process. If the record we want to search is "Mary", we will find that in the first iteration. In general, binary search is faster than sequential search. If we use sequential search, it takes six steps to find "Jason" and five steps to find "Mary". In the following section, we should have an analysis of the two algorithms.

8.4. Analysis

When one analyzes an algorithm, in general, we talk about the average behavior of the algorithm. However, there are three cases for the analysis.

Table 8.1.

Array Index	1	2	3	4	5	6	7	8	9
Key	−26	−5	−2	0	4	9	28	59	99
Content	Tim	Nina	Tom	John	Mary	Jason	George	Alex	Jason
Search Order					1	3	2		

Table 8.2.

First	1	6	6
Last	9	9	6
Current	5	7	6

The *best case* is the lucky situation, of which the algorithm finds its solution in the shortest time. The *average case* is the one we are working on, which means on the average, how many steps the algorithm may take to complete the job. The *worst case* is, of course, the longest time for the algorithm to find the solution.

In a sequential search, if there are n elements in the storage structure, the worst case of search is that n steps are taken. Of course, the best case is that only 1 step is taken. But, what about the average case? Theoretically, the element could be in the first position, the second position, or, the nth position. Thus, the average case should take an average of all of these possible positions:

$$(1 + 2 + 3 + \cdots + (n - 1) + n)/n = (n + 1)/2$$

Consequently, the sequential search is an algorithm of time complexity of $O(n)$. The binary search algorithm is analyzed in the similar manner. If we are lucky, the middle of the search array contains the target record. Then, the best case is 1 step. The binary search algorithm uses a strategy to divide the array into two parts each time to guess the solution. But, how many times can we divide an array of n elements? The *halving function* is defined by $H(n) =$ *the number of times that* n *can be divided by two, stopping when the result is less than one.* And, $H(n) = \lfloor \log_2 n \rfloor + 1$. It turns out that $H(n)$ is nearly equal to the base 2 logarithm of n. The worst case of binary search is, therefore, equal to $O(\log_2 n)$. The analysis of average case is tedious. But, the average case of binary search has the time complexity same as the worst case, which is $O(\log_2 n)$. conclusively, we say that binary search is an algorithm of time complexity that equals $O(\log_2 n)$.

8.4.1. *The Comparison Trees*

Search algorithms use comparisons to identify the target as well as to decide the next position of search. The comparison process can be visualized, in order to have a better understanding of the search efficiency. According to the example presented in Sec. 8.3.1, and the sequential search algorithm that compares the equality of a target to the key, the comparison tree of sequential search can be drawn as shown in Fig. 8.3.

Note that, each comparison results in the equal (to the left tree) or the unequal (to the right tree) situation. A box representing the target record is found, and the circle representing the record is not found. For a sequential search, the worst case will result when one searches toward the end of the search tree. The binary search uses the equality comparison, as well as the inequality comparisons (i.e. the $\leq$ and the $\geq$ operators).

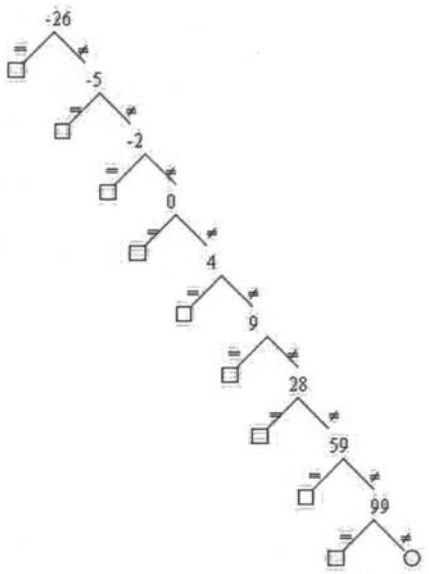

Fig. 8.3. The comparison tree of sequential search.

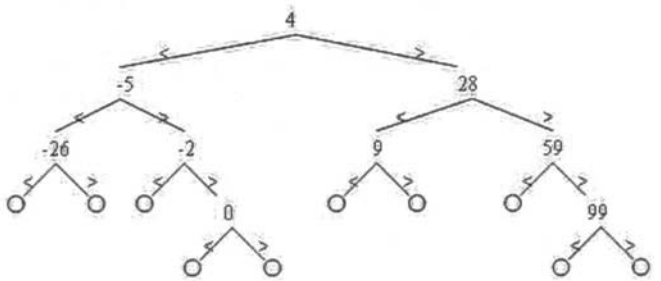

Fig. 8.4. The comparison tree of binary search.

According to the example presented in Sec. 8.3.1, the comparison tree of binary search is shown in Fig. 8.4.

Note that, the boxes are omitted as each number node represents the success retrieval of the number. From the visualized search trees, we can see that binary search algorithm performs better than sequential search in general. However, binary search requires to use array, and the data records should be sorted. On the other hand, sequential search does not have these restrictions.

8.5. Fibonacci Search

Binary search is faster than sequential search. But, is there a search algorithm faster than binary search? It is proven that, the best possible time complexity of search n objects will be $O(\log_2 n)$. However, within this restriction, we may improve the efficiency of the search algorithm. In the binary search algorithm, we use a division operator, which is time consuming. One way to improve the efficiency is to replace the division by a shift operator. Based on the integer representation used in many commercial computers, a right-shift operation will divide and truncate an integer, which is used as the index of the search array. On the other hand, it is possible to use the subtraction operator, if the programming language does not support the shift operator. Fibonacci search uses such a strategy. We have set this as an exercise problem. The Fibonacci search algorithm is very similar to binary search. However, it will perform better since subtraction is faster than division.

Exercises

Beginner's Exercises

1. Write a program, input a number, n, and read from a disk file of n records. A record can take a format similar to the one discussed in Sec. 8.1. Store these records in a linked list. Write a sequential search function, which accepts a key and prints the corresponding record or displays an error message if the record is not found.
2. Write a program using an array of n records, which is filled with sorted records. The sorting order is according to the key of each record. Write a binary search function, which accepts a key and prints the corresponding record or displays an error message if the record is not found.
3. A sorted array F with data $2, 6, 9, 11, 13, 18, 20, 25, 26, 28$. Please show how the following binary search algorithm works to find "9" from the input array F.

```
procedure BINSRCH(F , n , i , k)
    front ← 1 ; rear ← n
        while 1 <= n do
            m ← ⌊front + rear)/2 ⌋
            case
            : k > k_m : front ← m + 1        //look in upper half
            : k = k_m : i ← m ; return
```

```
                : k < k_m : rear ← m − 1          //look in lower half
                end
            end
        i ← 0                          //no record with key k
    end BINSRCH
```

4. Assume that there is an array with indices and keys shown below.

Array Index	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
Key	−26	−5	−2	0	7	11	13	24	35	39	44	46	52	55	59	67	72	79

Please show the sequence of compared keys when using binary search algorithm to search the key "−2."

Intermediate Exercises

5. Revise Exercise 2, by adding a counter before each of the comparison statement, to count the number of comparisons used. Write a program to invoke the binary search function, with all of the keys of the records as parameters on iterations. Summarize the number of comparisons used and find how many times in average, the comparison is made to retrieve all records.
6. The time complexity of binary search is _____.

Advanced Exercises

7. The Fibonacci number is defined as the following:

$$f(0) = 1$$
$$f(1) = 1$$
$$f(n) = f(n-1) + f(n-2), \text{ if } n > 1$$

Revise the binary search algorithm, using subtraction to replace the division operator. Use the concept of Fibonacci number.

8. On an ordered file with keys (1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16), use Fibonacci search to determine the number of key comparisons made while searching the keys "7" and "10." (Note: Fibonacci number $F_5 = 5$, $F_6 = 8$, $F_7 = 13$.)