MODULE 6

SEARCHING

DATA STRUCTURE AND ALGORITHMS

FACULTY OF COMPUTING
UNIVERSITI TEKNOLOGI MALAYSIA

OBJECTIVES FOR STUDENTS

1. Able to describe the searching technique concept and the purpose of searching operation.
2. Develop C++ code to implement the basic searching algorithms.
3. Able to analyze the efficiency of the searching technique.
4. Able to develop C++ code to implement searching technique in problem solving.

KEY CONCEPT

1.0 INTRODUCTION TO SEARCHING

1.1. **Searching definition -** A process to determine whether an element is a **member** of a certain data set or a collection of elements. The process also aims to find the location of the element with a specific value (key) within the collection of elements.

1.2. The process can also be seen as an attempt to **search** for a certain record in a file
   - i. Each record contains **data field** and **key field**
   - ii. **Key field** is a group of characters or numbers used as an **identifier** for each record
   - iii. Searching can done based on the key field.

1.3. Consider the following data set of employee record. Searching can be done based on certain field: **empID**, or **empl_IC**, or **empName.**
   - To search **empID** = 122, give us the record value at index 1.

index	empID	Empl_IC	EmpName	Post
[0]	1111	701111-11-1234	Ahmad Faiz Azhar	Programmer
[1]	**122**	**800202-02-2323**	**Mohd. Azim Bin Mohd. Razi**	**Clerk**
[2]	211	811003-03-3134	Nurina Raidah Bt Abdul Aziz	System Analyst

1.4. Among the popular searching techniques are as follows:

i. Sequential search
ii. Binary Search
iii. Binary Tree Search
iv. Indexing

1.5. Similar with sorting, Searching can also be implemented in two cases, **internal** and **external** search.

- **External search** – only implemented if searching is done on a **very large size** of data. Half of the data need to be processed in **RAM** while half of the data is in the **secondary storage**.
- **Internal search** – searching technique that is implemented on a **small size** of data. All data can be load into **RAM** while the searching process is conducted.

2.0 BASIC SEQUENTIAL SEARCH

2.1. Used for searching that involves records stored in the main memory (RAM).

2.2. Basic sequential search also used to search an element from unsorted list.

2.3. Basic sequential search is the **simplest search algorithm**, but is also the slowest and can only be used to search from a small list. The efficiency of sequential search is low compared to other searching techniques.

2.4. In a sequential search, (1) every element in the array will be examine sequentially, starting from the first element; (2) The process will be repeated until the last element of the array or until the searched data is found.

2.5. Searching strategy :

- **Examines** each element in the array one by one (sequentially) and compares its value with the one being looked for – the search key.
- Search is successful if the search key **matches** with the value being compared in the array. Searching process is **terminated**.
- else, if no matches is found, the search process is **continued to the last** element of the array. Search is **failed** array if there is no match found from the array.

2.6. Basic sequential implementation are given in Program 6.1:

```
// Program 6.1
// Search items in an array of ascending order.
int SequenceSearch( int   search_key,
                     const int array [ ],
                     int   array_size )
{  int p;
   int index =-1;
   //-1 means record is not found
   for ( p = 0; p < array_size; p++ ){
           if ( search_key == array[p] ){
              indeks = p;//assign current array index
                 break;
           }//end if
   } //end for
          return index;
} //end function
```

Every element in the array will be examined until the search key is found

Or until the search process has reached the last element of the array

2.7. Figure 6.1 shows an example of basic sequential search implementation with search key 22 and an array of integer [11 33 22 55 44].

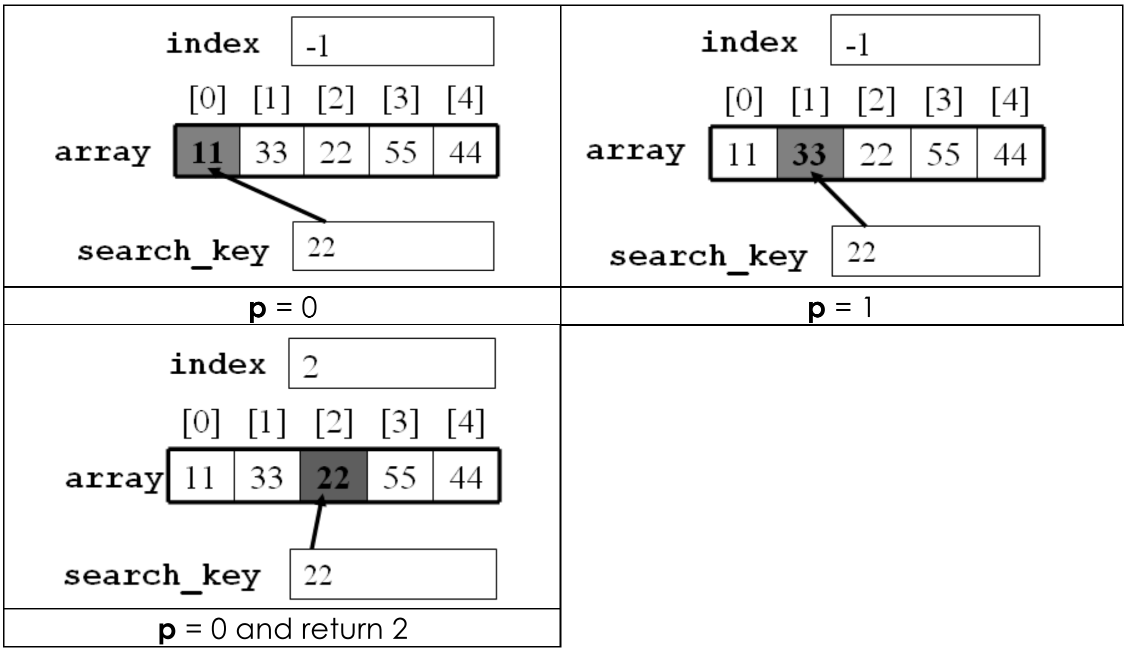


Figure 6.1 The basic sequential search implementation with search key 22

2.8. Example of basic sequential search implementation with search key 25 and an array of integer [11 33 22 55 44]. Every element in the array will be examined using the for loop and during the value of variable **p** will be increment from 0 to 4. No match found and the value of variable found and index will remind false and -1 respectively.

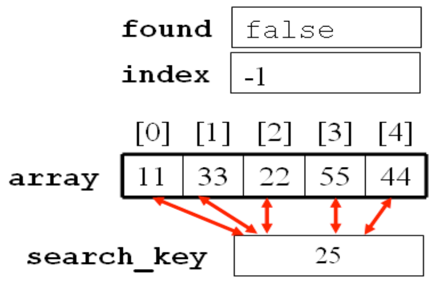


Figure 6.2 The basic sequential search implementation with search key 25

2.9. Basic sequential search analysis:

- Searching time for sequential search is O(n).
- If the searched key is located at the end of the list or the key is not found, then the loop will be repeated based on the number of element in the list, O(n).
- If the list can be found at index 0, then searching time is, O(1).

2.10. Problem and improvement of basic sequential search technique.

- **Problem:**
  - Search key is compared with all elements in the list, **O(n)** time consuming for large datasets.
- **Solution** to minimize the searching process.
  - The efficiency of basic search technique can be improved by searching on a **sorted list**.
  - For searching on ascending list, the search key will be compared one by one until :
    - i. The searched key is **found**.
    - ii. Or until the searched **key value is smaller than the item compared** in the list.

```
// Program 6.2
// Improved Basic Search
// For searching on ascending list

int SortedSeqSearch ( int search_key, const int array[],int
array_size)
{  int p;
   int index = -1;
   //-1 means record not found
   for ( p = 0; p < array_size; p++ )
   {  if (search_key < array [p] )
          break;
          // loop repetition terminated
          // when the value of search key is
          // smaller than the current array element
        else if (search_key == array[p])
        {
          index = p;
```

the searched key value is smaller than the item compared in the list.

```
            // assign current array index
            break;
          } // end else-if
    }//end for
    return index; // return the value of index
} //end function
```

the searched key is **found**

2.11. Example of the improved basic sequential search with search key 25 and an array of integer [11 22 33 44 55]:

Step 1	**index** -1 [0] [1] [2] [3] [4] **array** 11 22 33 44 55 **search_key** 25	Initial value for variable **index** and **array** elements
Step 2	**index** -1 [0] [1] [2] [3] [4] **array** **11** 22 33 44 55 **search_key** 25	p = 1 **search_key** is compared with the first element in the **array**
Step 3	**index** -1 [0] [1] [2] [3] [4] **array** 11 **22** 33 44 55 **search_key** 25	p = 2 **search_key** is compared with the second element in the **array**
Step 4	**index** -1 [0] [1] [2] [3] [4] **array** 11 22 **33** 44 55 **search_key** 25	p = 3 **search_key** is compared with the third element in the **array** • the value of **search_key** is smaller than the current element of array • loop repetition is terminated using "**break**" statement

Figure 6.3 The improved basic sequential search implementation with search key 25

2.12. Conclusion on steps to execute sequential search function on a sorted list:

- If the elements in the list are not in a sorted (asc/desc) order, loop will be repeated based on the number of elements in the list.
- When the list is not sorted the loop is repeated 5 times, compared to 3 times if the list is in sorted order as shown in the previous example.
- If the list is sorted in descending order, change operator **"<"** to operator **">"** in the loop **for.**

3.0 BINARY SEARCH

3.1. The drawback of sequential search algorithm is having to **traverse the entire list**, *O(n)*.

3.2. Sorting the list does minimize the cost of traversing the whole data set, but we can improve the searching efficiency by using the **Binary Search** algorithm.

3.3. Consider a list in ascending sorted order. For a sequential search, searching is from the beginning until an item is found or the end is reached.

3.4. Binary search improve the algorithm by **removing as much of the data set as possible** so that the item is found more quickly.

3.5. Search process is started at the **middle** of the list, then the algorithm determine which half the item is in (because the list is sorted).

- It divides the working range in half with a single test. By repeating the procedure, the result is an efficient search algorithm-*O(log2 n)*.

3.6. Implementation of Binary Search - starts by comparing the search key with the element at the **middle**.

i. If the value matches, it will be return to the calling function (index = MIDDLE)
ii. If the search key < the middle element, search will be focused on the elements between the first element to the element before the middle element (MIDDLE -1)
iii. If the search key is not found, the element at the middle of the first element and the MIDDLE -1 element will be compared with the search key.
iv. If the search key > the middle element, search will only be focused on the elements between the second MIDDLE element to the first MIDDLE element.
v. Search is repeated until the searched key is found or the last element in the subset is traversed (**LEFT > RIGHT**).

3.7. Figure 6.4 shows an example of the implementation of Binary Search with search key 35 in an array of integer [11 22 33 44 55 66 77] .

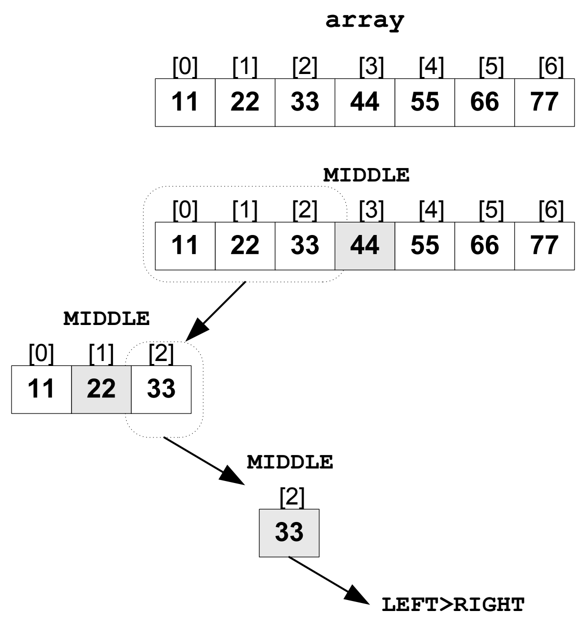


Figure 6.4 The implementation of Binary Search with search key 35

3.8. Program 6.3 is the Binary search function. The numbers i-v in the comment are referring to the implementation described at point 3.6.

```
// Program 6.3
// Search items in an array of ascending order.
int binary_search( int  search_key, int array_size,
                            const int array [] )
{  bool found = false;
   int index = -1 //-1 means record not found
   int MIDDLE,
   LEFT = 0,
   RIGHT = array_size-1;
   while (( LEFT<= RIGHT ) && (!found)) // v.
   {      MIDDLE = (LEFT + RIGHT )/2;// Get middle index
          if ( array[MIDDLE] == search_key)
          {   index = MIDDLE;
              found = true;
          }
       else if (array[MIDDLE] > search_key) //ii
          RIGHT= MIDDLE– 1; //iii search is focused
                   // on the left side of list
       else
          LEFT= MIDDLE+ 1 //iv. search is focused
                   // on the right side of the list
     } //end while
    return index; //i
```

```
}//end function
```

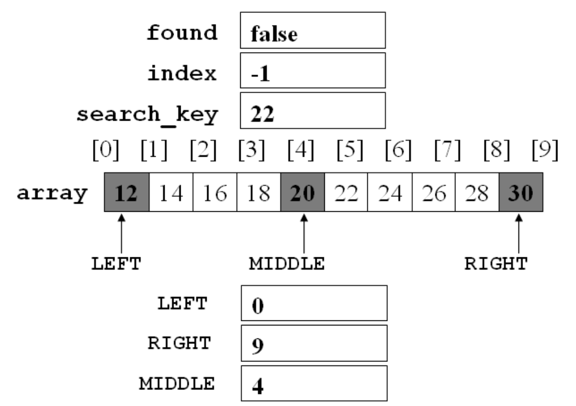
3.9. Consider the implementation of the Binary Search on a sorted list [11 22 33 44 55 66 77] with search key 35.

a. Search starts by obtaining the **MIDDLE** index of the array:
   MIDDLE= (0 + 6) / 2= 3 { First **MIDDLE** index}
b. search_key **35** is compared with the element at the fourth index in the array, which is array[3] with the value **44**.
c. **search_key < MIDDLE** value, therefore search will be focused on the elements between the first index and the third index only (index 1 to **MIDDLE-1**)
d. Process to obtain **MIDDLE** index is repeated:
   MIDDLE = (0 + 2) / 2= 1 { second **MIDDLE** index}
e. **search_key 35** is compared with the element at the second index, which is array[1] with the value **22**
f. **search_key > MIDDLE** value, therefore search will be focused only on the elements between the second **MIDDLE** index to the first **MIDDLE** index.
   MIDDLE = (2 + 2) / 2= 2 { third **MIDDLE** index}
g. Element at the third index, array[2] with the value 33 is not equal to the value of the search key.
h. Search process has reached the last element of the traversed subset, therefore search is terminated and assumed fail.
i. To search from the list sorted descending, change operator " **>** " to operator " **<** " to the following statement :

   else if (array [**MIDDLE**] **>** search_key)
   RIGHT = MIDDLE – 1;

3.10. Consider another implementation of the Binary Search on a sorted list [12 14 16 18 20 2 24 26 28 30] with search key 22 and the array size is 10 in Figure 6.5.

Step 1



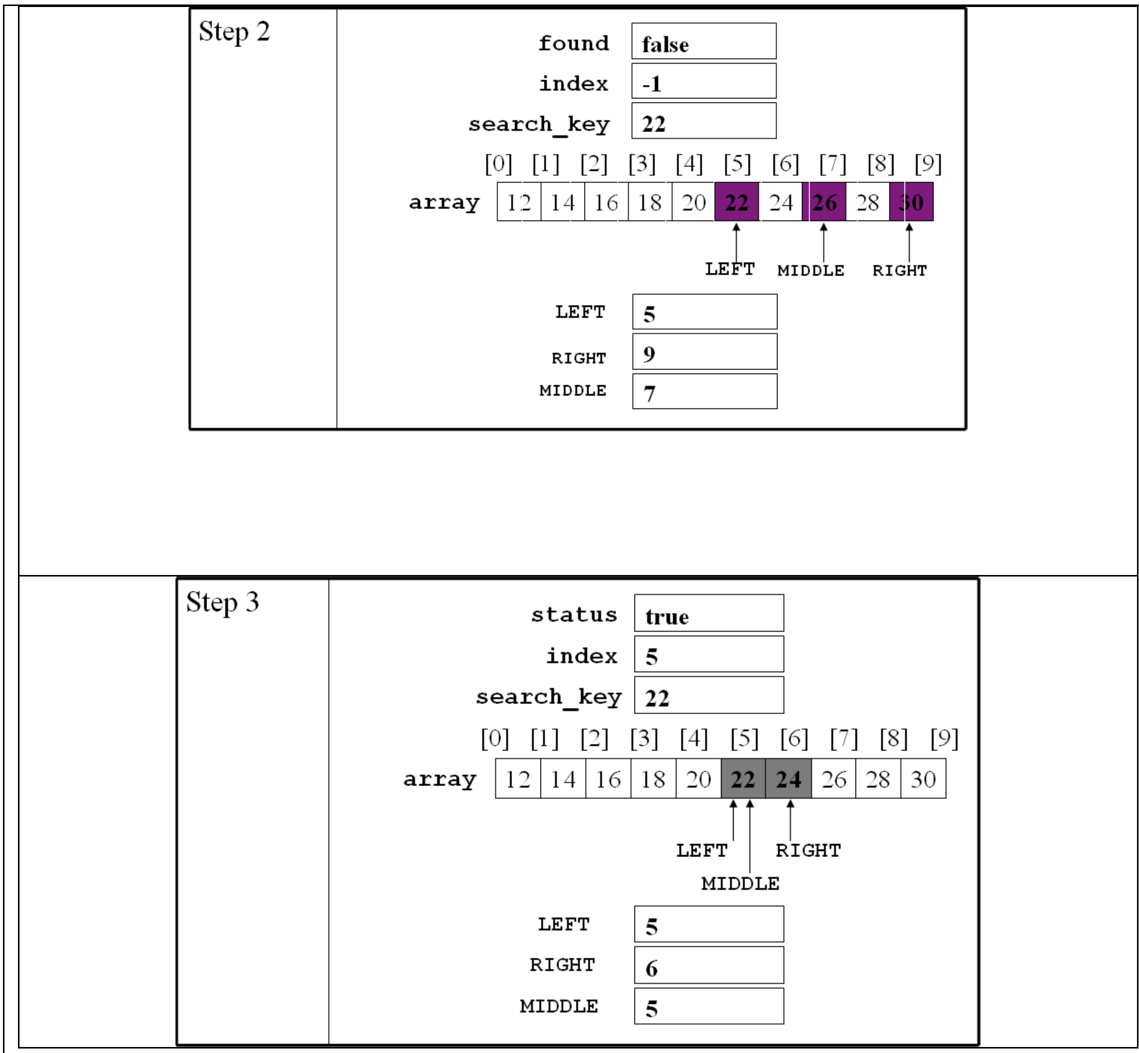


Figure 6.5 The implementation of Binary Search with search key 22

3.11. Binary Search analysis:

- Binary Search starts searching by comparing element in the middle. Thus the searching process start at *n/2* for a list size = n.
- If the middle value does not matches with the search key, then the searching area will be reduced to the left or right sublist only. This will reduce the searching area to ½ *n*.
- From half of the list, the second middle value will be identified. Again, if the middle value does not matches with the search key, the searching area will be reduced to the left or right sub list only. The searching area will reduce ½ (½ *n*).
- The process of looking for middle point and reducing the searching area to the left or right sublist will be repeated until the middle value is equal to the middle value (search key is found) or the last value in sublist has been traverse.
- If the repetition occur *k* times, then at iteration *k*, the searching area is reduced to $(½)^k n$.
- Figure 6.6 shows the reducing size for binary search area.

- At iteration k for array size = *n* , searching area will be reduced from *n* to $(\frac{1}{2})^k n$.

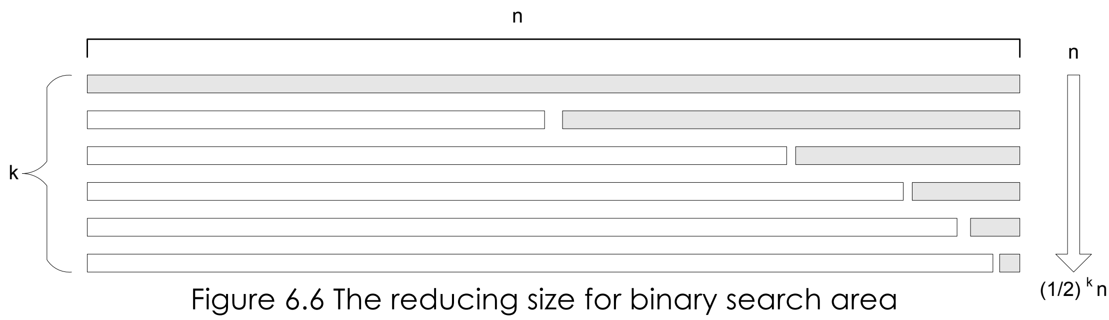


Figure 6.6 The reducing size for binary search area

- Figure 6.7 shows an example of the reducing size for binary search area with 1 billion data.
- It can be concluded that to search item in the middle of the list, the complexity is O(1).
- Searching item at the back or front of the list is faster with only O(29).
- However, searching item at front of the list is the worst case of Binary Search with O(29).

1	500,000,000
2	250,000,000
3	125,000,000
4	62,500,000
5	31,250,000
6	15,625,000
7	7,812,500
8	3,906,250
9	1,953,125
10	976,562
11	488,281
12	244,140
13	122,070
14	61, 035
15	30,517

16	15,258
17	7,629
18	3,814
19	1,907
20	953
21	476
22	238
23	119
24	59
25	29
26	14
27	7
28	3
29	1

Figure 6.7 The reducing size for binary search area with 1 billion data

PROGRAMMING EXERCISES

LAB 1: BASIC SEQUENTIAL SEARCH

Given the following Program 6.4, type and run the program to perform the tasks given below.

```
// Program 6.4
#include<iostream.h>
int SequentialSearch(int [], int, int);

void main()
{       int num[100];
        int k, target, j;
        char ans = 'y';

        cout << "Please enter size of the array:";
        cin >> k;
        for(int i = 0; i < k; i++)
        {       cout << "num[" << i << "]=";
                cin >> num[i];
        }
        do {
          cout << "\nEnter the search key :";
            cin >> target;
            j = SequentialSearch(num, k, target);
            if (j == -1)
                  cout << "Failed" << endl;
            else
                  cout << "Found at num[" << j << "]\n";
            cout << "Find another number?:";
            cin >> ans;
        } while (ans == 'y');

}
int SequentialSearch(int a[], int n, int target)
{       int i;
        for (i = 0; i < n; i++)
                if (a[i] == target)
                        return i;
                return -1;
}
```

i. Read the input for **num** array which has the following 10 ascending numbers:

	[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]
num	8	4	10	5	20	4	15	23	12	11

Figure 6.8 Unsorted data

ii. Perform searching with the following key values: 5, 4, and 25.

iii. What are the output be when performing a search for the values?

LAB 2: BINARY SEARCH

Refer to the given Program 6.4 in Lab 1, replace the **SequentialSearch()** function with **BinarySearch()**function given below.

```
// Program 6.5
int BinarySearch(int a[], int n, int target)
{       int first = 0;
        int last = n - 1;
        int mid;
        while (first <= last)
        {       mid = (first + last) / 2;
                if(target == a[mid])
                        return mid;
                else if(target < a[mid])
                {
                  last = mid - 1;
                  cout << "Middle value:" << mid <<
                "\tfirst:" << first << "\tlast:"
        << last << endl;
                }
                else
                {
                  first = mid + 1;
                  cout << "Middle value:" << mid <<
                "\tfirst:" << first << "\tlast:"
        << last << endl;
                }
        }
        return -1;
}
```

Based on Program 6.4 and the function in Program 6.5, perform the following tasks:

a. Read **num** array input with the following 10 ascending numbers:

	[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]
num	2	5	6	8	10	12	15	16	20	21

Figure 6.9 Sorted data for **num** array

b. Perform search with the value of search key 5, 20 and 25. What will be the output?

iv.

LAB 3: BASIC SEQUENTIAL SEARCH AND BINARY SEARCH ALGORITHMS

Carefully study the following program template and the sequential searching function on a sorted list called **SeqSearch**().

```
//Program 6.6
#include <iostream.h>
#include <conio.h>
#include <stdio.h>

class Person
{ public:
    int key;
    char name[30];
    char uta[30];
  public:
     Person()
     { Person(0,"","");
     }
     Person( int key, char name[], char uta[])
     { this->key=key;
       strcpy(this->name, name);
       strcpy(this->uta, uta);
     }
};

/**************Function prototypes **********************/
void display_array(Person list[], int size, char title[]);
void QuickSort(Person list[], int first, int last);
int Divider(Person T[], int awal, int last);
int SeqSearch(int key, Person list[], int size);
int Binary_Search(int key, Person list[], int size);
void pause();
/*******************************************************/
const int COUNT = 14;  // number of array elements.
void main(void)
{  // array T, object Person
  Person T[COUNT] = {          Person(21, "Utada",  "ichi"),
                               Person(61, "Hikaru", "ni"),
                               Person(11, "Ito",       "san"),
                               Person(31, "Yuna",    "shi"),
                  Person(79, "Hamasaki", "yon" ),
                  Person(83, "Ayumi",  "go" ),
                  Person(68, "Koda",    "roku" ),
                  Person(78, "Kumi",    "shichi" ),
                  Person(96, "Namie", "nana" ),
                  Person(87, "Amuro", "hachi" ),
                  Person(57, "Otsuka", "ku" ),
                  Person(88, "Ai",         "kyu" ),
                  Person(69, "Kaoru",  "jyuu" ),
```

```
                        Person(49, "Amane",           "jyuichi" )

                    };
   display_array(T, COUNT, "Original Array:");
   pause();

   /********** SORT *****************/
   cout << "****** SORTING*********\n\n";
   QuickSort(T, 0, COUNT-1);
   display_array(T, COUNT, "Sorted Array:");
   pause();

   /********** Search *****************/
   cout << "****** SEARCHING *********\n\n";
   int key;    // search key
     int index;  //the array index for the element found
                 //from searching process

   cout <<"Enter search key: ";
   cin >> key;

   // call the search function here
                        index = SeqSearch(key,T,COUNT);
   //  index = Binary_Search(key,T,COUNT);

   cout << "\n\nSearch result:\n";
   cout << "\tIndex element: " << index << endl;
   if (index>-1)
   { cout << "\tKey: "  << T[index].key << endl;
     cout << "\tName: " << T[index].name << endl;
     cout << "\tTitle: " << T[index].uta << endl;
   }
    else
       { cout << "\tRecord not found!!!" << endl;
       }
   pause();
 }

 /************************************************************
   Function:     display_array
   Description:  Print object Person from array

 ************************************************************/
 void display_array(Person list[], int size, char title[])
 {int i;
  cout << title <<endl <<endl;
  cout <<" Key \t  Name   \t Title\n";
  cout <<"-----\t -------\t------- \n";

  for (i=0; i<size; i++)
```

```
   cout << list[i].key <<"\t"<< list[i].name
       <<"\t\t"<< list[i].uta <<"\n";
 cout << endl;
}

/**********************************************************
  Function:  QuickSort
  Description: Execute Quick Sort Algorithm
**********************************************************/
void QuickSort(Person list[], int first, int last)
{ int cut;

  if (first<last)
  {  cut = Divider(list, first, last);
     QuickSort(list, first, cut);
     QuickSort(list, cut+1, last);
  }
}

int Divider(Person T[], int first, int last)
{          int pivot;
           int loop, divide, frombottom, fromtop;
           Person temp;

           pivot = T[first].key;
           frombottom = first; fromtop = last;
           loop = 1;

  while (loop)
  {    while (T[fromtop].key>pivot)
      {  // search for values less than pivot
         // from the top of array
              fromtop--;
      }

      while (T[frombottom].key<pivot)
      {   // search for values bigger than pivot
                      // from bottom of array
               frombottom++;
      }

      if (frombottom < fromtop)
      {  // swap location of pivot
              temp = T[frombottom];
              T[frombottom] = T[fromtop];
              T[fromtop] = temp;
      } else
      {   loop = 0;
          divide = fromtop;
      }
```

```
  } //end while (loop)
  return divide;
} //end function Divider

/*********************************************************
  Function:  SeqSearch
  Description: Execute Sequential Search Process
  Parameters:
    key,  which is the search key
    list, which is the array, assume the array list is
          sorted in ascending order
    size, the number of records in array
  Returns:
     If record is found, the record index will be returned.
     Otherwise, if not found, value -1 will be returned.
*********************************************************/


int SeqSearch(int key, Person list[], int size)
{
        int i;
        for (i = 0; i < size; i++)
        {       if (key == list[i].key) return i;
        // if record is found
                        if (key < list[i].key) return -1;
                // if search key > current record key,
                // no need to continue search on
        // the remaining records
        }
        return -1;
}
/*********************************************************
  Function:  Binary_Search
  Description: Execute Binary Search Process
  Parameters:
     key,  which is the search key
     list, which is the array, assume the array list
           is sorted in ascending order
     size, the number of records in array
  Returns:
     If record is found, the record index will be returned.
     Otherwise, if not found, value -1 will be returned.
*********************************************************/
int Binary_Search(int key, Person list[], int size)
{       // local variable
        bool found = false;
        int MIDDLE, LEFT = 0, RIGHT = size - 1;
        int i = -1;
}
```

```
/*********************************************************
 Function:    pause
 Description:  pausing the screen for awhile to view output
*********************************************************/

void pause()
{
 cout << "\n\nPress any key....\n";
 getch();
 }
```

Based on the given code modify the program according to the following specification:

a. Write a new search function **Binary_Search**() that executes binary search algorithm to replace function **SeqSearch**().

b. Change the order of the array from ascending to descending by modifying the sort function **QuickSort**().Write again both search function **SeqSearch**() and **Binary_Search**() based on descending order.

c. Execute the program and see if your searching functions give out the correct output.

EXERCISES

EXERCISE 1: SEQUENTIAL SEARCH ALGORITHM

Program 6.7 is a sequential search function.

```
// Program 6.7
int SequenceSearch(int search_key, int array[],int array_size )
{         int p;
          int index = -1;
          // -1 means record is not found

          for (p = 0; p < array_size; p++)
          {
                    if (search_key == array[p])
                    {         indeks = p;
                      // assign current array index
                              break; // terminate searching
                    } // end if
          } // end for
          return index;
          // return location of value
} // end function
```

In the following figure, **DATA** is an array with the size of 10, which stores integer values.

index	**[0]**	**[1]**	**[2]**	**[3]**	**[4]**	**[5]**	**[6]**	**[7]**	**[8]**	**[9]**
DATA	49	21	34	35	40	7	15	26	12	4

Figure 6.10 Unsorted data

a. Based on the given function, show each steps of the search process on the **DATA** array based on the following search key values.
   i. 49
   ii. 4
   iii. 37

b. Compare the searching time for both search keys. Based on the searching time, discuss the efficiency of the searching technique.

v.

c. Explain the use of **break** statement to the **for** loop in the function in Program 6.7.

In the following figure, the array **DATA1** has been sorted in ascending order.

index	**[0]**	**[1]**	**[2]**	**[3]**	**[4]**	**[5]**	**[6]**	**[7]**	**[8]**	**[9]**
DATA1	4	7	12	15	21	26	34	35	40	49

Figure 6.11 Sorted data

d. Rewrite the function in Program 6.7 to execute searching on a sorted list.

e. Based on your answer in question (d), show each steps of the search process on the **DATA1** array based on the following search key values.

   i. 4
   ii. 37

f. Based on your answer in question (e), explain how and why is the sequential search on a sorted list better than on an unsorted list if the value being searched is not in the list.

EXERCISE 2: SEQUENTIAL SEARCH ALGORITHM AND THE ANALYSIS

Assume that **r** is an array consisting *n* number of records. Each record has a key field k, and the key for the **i**-th record is referred to as **r[i].k**. The records in the array **r** are in an unsorted order.

a. Write a sequential search function in C++, for the array **r**, with the purpose of searching a given record key. Use the most efficient algorithm that you have learned.

b. If the array **r** are in a sorted order, write a sequential search function in C++ that will execute a search on a sorted array based on a given search key.

c. Discuss the difference in the efficiency of the algorithm for the answers gave in questions (a) and (b).

EXERCISE 3: BINARY SEARCH ALGORITHM

Program 6.8 is a binary search function.

```
// Program 6.8
int binary_search( int search_key, int array_size,
const int array [] )
{
        bool found = false;
        int index = -1 //-1 means record not found
        int MIDDLE,
        LEFT = 0,
        RIGHT = array_size - 1;

        while (( LEFT <= RIGHT ) && (!found))
        {       MIDDLE = (LEFT + RIGHT )/ 2;
                // Get middle index
                if (array[MIDDLE] == search_key)
                {       index = MIDDLE;
                        found = true;
                }
                else if (array[MIDDLE] > search_key)
                        RIGHT = MIDDLE – 1;
                        // search is focused
                // on the left side of list
                else
                        LEFT = MIDDLE + 1
                    // search is focused
                    // on the right side of the list
        } // end while
        return  index;
} // end function
```

a. Discuss the differences between sequential search and binary search algorithms.

b. Based on the function in Program 6.8 show each steps of the search process on the **DATA** array in Figure 6.11 on the following search key values.

i. 4
ii. 37

c. State the values for variables **LEFT**, **RIGHT** and **MIDDLE** found at each step.

EXERCISE 4: BINARY SEARCH ALGORITHM

Give 2 classes of search. State which search is suitable for huge records and which is suitable for small number of records.

a. Given an ascending order array as follows:

index	[0]	[1]	[2]	[3]	[4]	[5]	[6]
Array	2	5	8	10	19	21	35

Figure 6.12 Sorted data

b. Give right and left values in a box that are involved in the process of binary search. Assume that the search key is 15. Fill in the values in Figure 6.13.

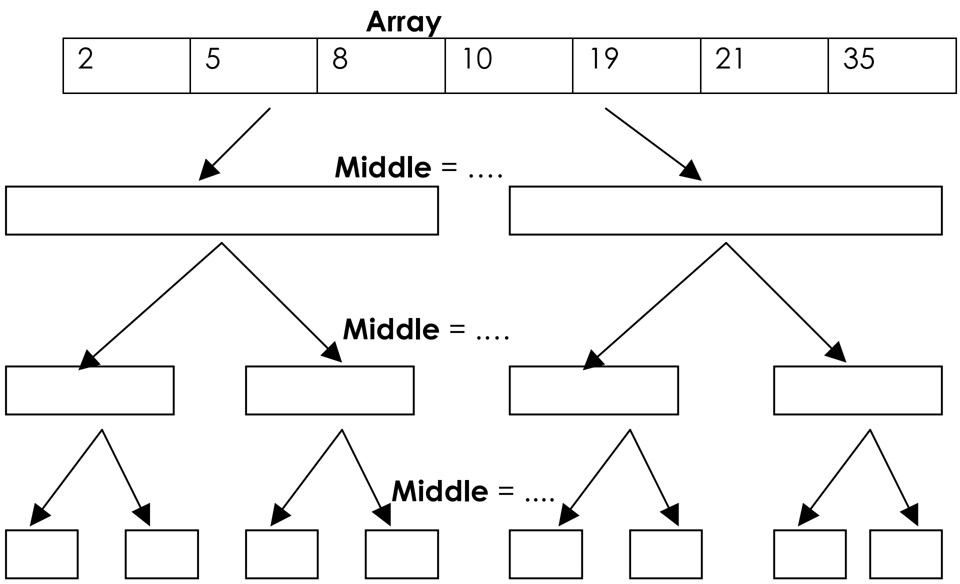


Figure 6.13 Binary search

EXERCISE 5: BASIC SEQUENTIAL SEARCH AND BINARY SEARCH ALGORITHMS

The following class diagram named **month** has 2 attributes, **key** – an integer value for **month** and **monthName** – the **char**[] value for month. The following array figure shows an array of class month, named **arrayA** with 7 elements. The array is sorted in descending order.

Month
key : int monthName : char[]

Figure 6.14 Class diagram **month**

	[0]	[1]	[2]	[3]	[4]	[5]	[6]
Key	**12**	**10**	**8**	**7**	**5**	**4**	**2**
monthName	**December**	**October**	**August**	**July**	**May**	**April**	**February**

Figure 6.15 **arrayA**

Answer the following questions based on **arrayA.**

a. Complete the sequeantial search function **SeqSearch**() below :

```
// Program 6.9
int SeqSearch(int search_key, month arrayA[], int array_size)
{
        int p;
        int index = -1;
        for (p = 0; p < array_size; p++)
        {
                if ______________________________
                {       index = p;
        // if record is found, assign
                // current array index
                        break;
                }
                else if ____________________________
                        break; // else, no need to                    //
                    continue search on the                    //
                    remaining records
        } //end for
        return index;
}//end function
```

b. Complete the binary search function of **Binary_Search()** below.

```
// Program 6.10
int Binary_Search(int search_key, month arrayA[], int array_size)
{
        bool found = false;
        int MIDDLE, LEFT = 0, RIGHT = array_size-1;
        int i = -1;
        while ((LEFT<=RIGHT) && (!found))
        {       MIDDLE = ____________________
                if (arrayA[MIDDLE].key == search_key)
                {       i = ____________;
                found = true;
                }
        else if ______________________
                RIGHT = _________________
    // search is focused on the left
        //  side of the list
     else
        LEFT = __________________
    //  search is focused on the right
        // side of the list
        } // end while
        return i;
} // end function
```

c. Complete the table below to show the values of the variables **i**, **left**, **right**, **middle** and **found** in order to search *12* as the search key during the searching process of **Binary_Search()** function. The searching process is implemented in the **arrayA**, which has been sorted in descending order.

i	LEFT	RIGHT	MIDDLE	FOUND

EXERCISE 6: BASIC SEQUENTIAL SEARCH AND BINARY SEARCH ALGORITHMS

Program 6.11 is a sequential search function. Based on the array **DATA** figure below, answer all the following questions.

```
// Program 6.11
int SequenceSearch(int search_key, int array[ ], int array_size )
{       int pass;
        int index = -1;
        //-1 means record is not found

        for (pass = 0; pass < array_size; pass++)
        {
                if (search_key == array[pass])
```

```
                    {       index = pass;
                     // assign current array index
                            break; // terminate searching
                    } // end if
            } // end for
            return index;
            // return location of value
} // end function

```

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

Figure 6.16 **DATA**

a. The above figure shows an array **DATA** with the size of 12, which stores integer values. Determine the value of iteration, **pass**, **array[pass]**, **index** and number of comparisons in the following table format, in each steps of searching process on the array **DATA** by using the sequential search function in Program 6.11 for the following search keys.

i. 45

ii. 22

Iteration	**pass**	**array**[**pass**]	**index**	Number of comparisons
..	..		..	..

vi. Program 6.12 is a binary search function. Based on the same array figure above, answer all the following questions.

```
// Program 6.12
int binary_search( int search_key, int array_size,
const int array [] ){
        bool found = false;
        int index = -1 //-1 means record not found
        int MIDDLE,
        LEFT = 0,
        RIGHT = array_size-1;

        while (( LEFT<= RIGHT ) && (!found))
        {       MIDDLE = (LEFT + RIGHT ) / 2;
                // Get middle index
                if (array[MIDDLE] == search_key)
                {       index = MIDDLE;
                        found = true;
                }
                else if (array[MIDDLE] > search_key)
                        RIGHT= MIDDLE– 1;
                        // search is focused
                        // on the left side of list
```

```
                else
                        LEFT = MIDDLE + 1
                        // search is focused
            // on the right side of the list
        } //end while
        return  index;
} //end function
```

b. Based on the above array, determine the value of iteration, **LEFT**, **RIGHT**, **MIDDLE**, **array**[**MIDDLE**], **index** and number of comparisons like in the table below, in each steps of searching process on the array **DATA** by using the binary search function in Program 6.12 for the following search keys.
   i. 45
   ii. 22

Iteration	**LEFT**	**RIGHT**	**MIDDLE**	**array[MIDDLE]**	**index**	Number of comparisons
..	..	..	..	..	..	..

c. Based on your answers in questions (a) and (b), what can you conclude on the efficiency of both search algorithms? Explain in term of efficiency and number of comparisons.

EXERCISE 7: SEARCHING TECHNIQUES AND THE ANALYSIS

Answer the following questions based on the sorted array named **marks** shown in Figure 6.17.

index	**[0]**	**[1]**	**[2]**	**[3]**	---	**[19]**	**[20]**	---	**[29]**	**[30]**	**---**	**[34]**	**---**	**[38]**
marks	**51**	**55**	**59**	**60**	---	**75**	**76**	---	**85**	**86**	**---**	**90**	**--**	**95**

Figure 6.17 : Sorted Array, **marks**

a. Based on the given array in Figure 6.17, assume that your mark in this course is **60** and need to be searched using **SortedSeqSearch()** function. Show the tracing of the search using variables **index, p**, **search_Key** and **found** as shown in the table format below.

index	**search_Key**	**p**	**found**

b. Assume your mark in this course is **90** and need to be searched using Binary Search function. Show the tracing of your search using variables **left**, **right**, **middle**, **marks[middle]** and **found** as shown in the table format below.

left	right	middle	marks[middle]	found

c. Fill in the following table with the number of steps and the complexity required in searching the **minimum mark**, **the average mark** (assume 75 is the average) and the **maximum mark**. Based on the results, compare and discuss the efficiency of Binary Search technique and Sequential Search (on sorted data) algorithms in the three searching cases.

	Search Comparisons			
Search Key	Linear Search		Binary Search	
	Number of steps	Complexity	Number of steps	Complexity
51				
75				
95				
Efficiency Analysis for the three cases				