

Student Name \_\_\_\_\_

### Searching Techniques

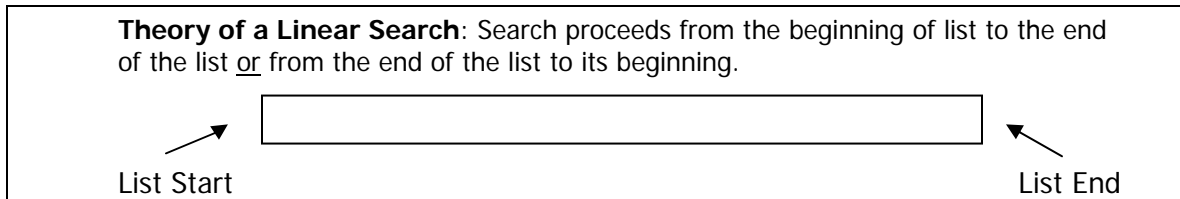
Searching techniques are utilized to locate items within an array. The item to be searched is known as the **key**.

Techniques used to search arrays include both the linear search, also known as a sequential search, and the binary search. For the linear model approach, a search is performed through an element by element comparison basis, from either the beginning to the end or from the end to the beginning. For this type of model the list may be in any order. For the binary model approach, the list must be in sort order first.

### Linear ( Sequential ) Search Logic

Linear Model Approach - search is by an element by element comparison. List may not be in any order vs. the Binary Model Approach - where list must be first in sort order.

**Diagram 1 - Linear Search Routine**



### Linear Search - Pseudocode

The pseudocode for the linear search model is:

```

For all of the items in a list
  Compare the list item with a desired item ( the key )
  If the item was found,
    Return index value of the current item
  End If
End for
Return -1 if the item is not found

```

### Linear Search - Sample Source Code

The program code given below consists of a function that returns the location of key in the list. The number - 1 is returned if the value is not found.

```

int linearSearch(const int array[], int key, int size)
{
    int n;
    for (n = 0; n <= size - 1; ++n)
        if ( array[n] == key )
            return n;
    return -1;
}

```

Student Name \_\_\_\_\_

### Linear Search - Advantages and Disadvantages

Of the advantages of the linear model for searching is that the list need not be sorted. If an item is close to the beginning then not many comparisons are needed. One disadvantage of this model is that a worst case scenario occurs when the item to be found is at the end or near end of the list.

**Advantages:** List does not have to be sorted. If item is close to the beginning, not many comparisons are needed.

**Disadvantages:** Worst case scenario occurs when item at the end or near end of list.

### Linear Search - Number of Comparisons

For a linear search, the number of comparisons needed, on the average ( assuming that the desired item is equally likely to be anywhere within the list ), is summarized in the chart below.

Average number of required comparisons is  $N / 2$  where  $N$  = the size of the list.

**Example:** For a 10 element list, the average number of comparisons needed =  $10 / 2 = 5$ .  
For a 10,000 element list, the average number of comparisons needed =  $10,000 / 2 = 5,000$ .

### Linear Search - Example Program

The following program will search for a department name to match a code number.

```
#include <iostream>
using namespace std;

void main() {
    int code[5] = {10, 20, 30, 40, 50};
    char dept[5][10] = { "Accounts", "Benefits", "Mail Room",
                        "Payables", "Payroll" };

    char found = 'n';
    int number;

    cout << "Enter a Department Code Number ---> ";
    cin >> number;

    for(int i = 0; i < 5 && found == 'n'; i++)
    {
        if(number == code[i])
        {
            found = 'y';
            cout << "\n Department: " << dept[i] << "\n" << endl;
        }
    }
    if(found == 'n')
    {
        cout << "\nDepartment not found - check number\n" << endl;
    }
}
```

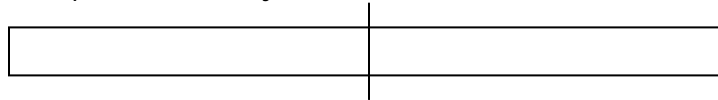
Student Name \_\_\_\_\_

### Binary Search Logic

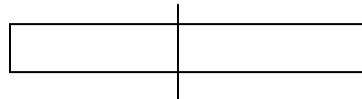
The binary search model, another type of searching algorithm, is very fast compared to a linear search in that fewer comparisons are needed. One disadvantage of this model is that the list must be sorted.

The basic theory behind a binary search is that a binary search cuts the number of elements needed to search in half each pass through a while loop. As illustrated below,

Search 1: List is split in half--If key is found...endWhile



Search 2: Split list again--If key is found...endWhile



::

Search N: Keep splitting list until search item is found -- If key is found...endWhile

Else Item was not found

### Binary Search - Pseudocode / Flowchart

The pseudocode for the binary search algorithm is given below.

Set the lower index to 0.

Set the upper index to 1 less than the size of the list.

Begin with the first term in the list.

While the lower index is less than or equal to the upper index,

    Set the midpoint index to the integer average of the lower and upper index values.

    Compare the desired item to the midpoint element.

        If the desired element equals the midpoint element,

            Return the index value of the current item.

        Else if the desired element is **greater** than the midpoint element,

            Set the lower index value to the midpoint value plus 1.

        Else if the desired element is **less** than the midpoint element,

            Set the lower index value to the midpoint value less 1.

    EndIf

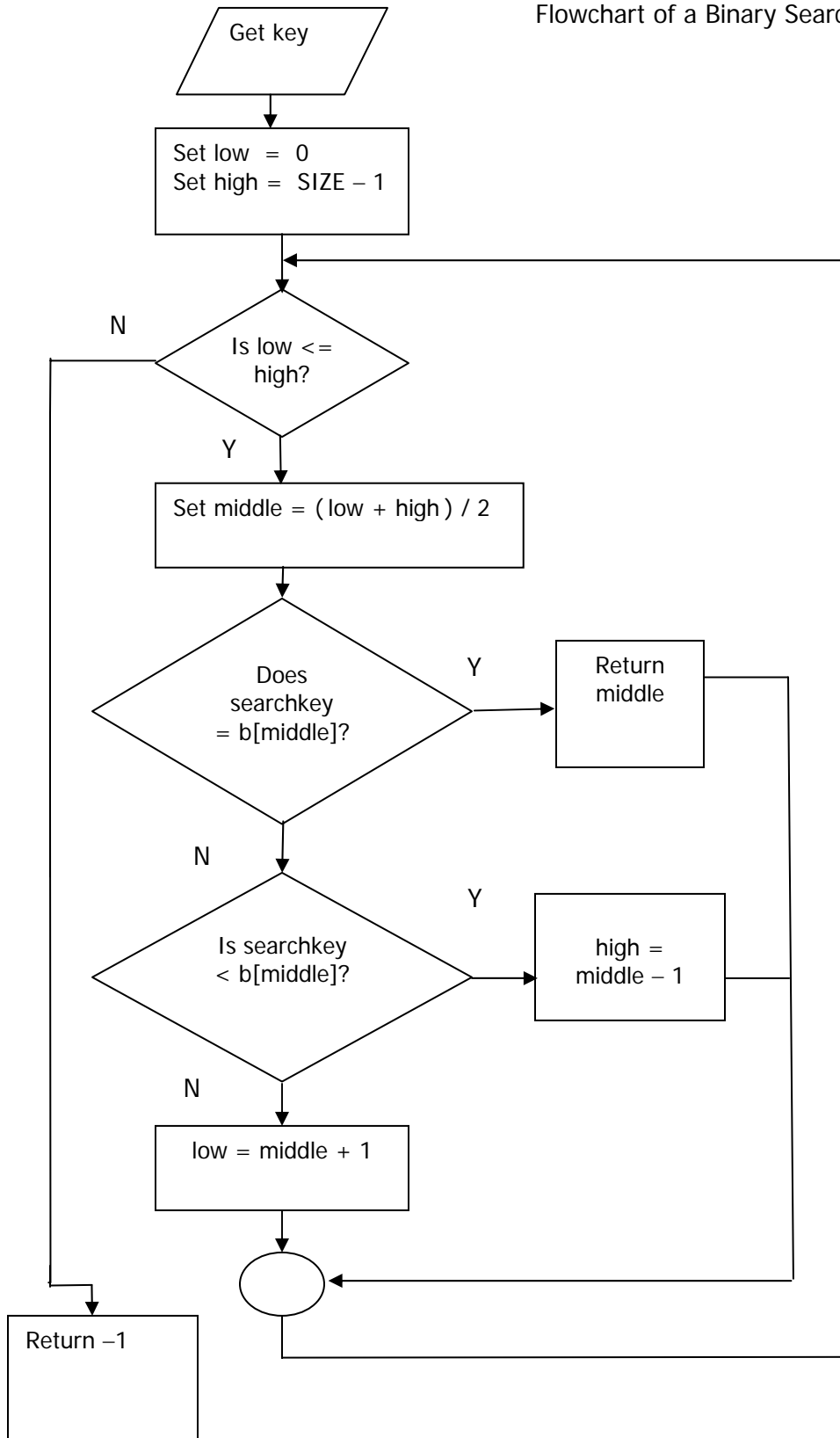
EndWhile.

Return -1 because the item was not found.

A flowchart version of the pseudocode follows on the next page.

Student Name \_\_\_\_\_

Flowchart of a Binary Search



Student Name \_\_\_\_\_

### Binary Search - Sample Source Code

The program code given below consists of a function that returns the location of key in the list. The number - 1 is returned if the value is not found.

```
int binarySearch(const int b[], int searchKey, int low, int high)
{
    int middle;
    while(low <= high) {
        middle = (low + high) / 2;

        printRow(b, low, middle, high);

        if (searchKey == b[middle])
            return middle;
        else if (searchKey < b[middle])
            high = middle - 1;
        else
            low = middle + 1;
    }

    return -1;    /* searchKey not found */
}
```

### Binary Search - Example

Given the following array declaration:

```
int b[] = {1, 3, 4, 10, 12, 45};
```

apply the above binary search function method to this sorted array to locate the number 10.

Based on this array example, the initial values are: low = 0, high = SIZE - 1 = 5, middle = (low + high) / 2 = 2, and b[middle] gets value (4) as middle points to index 2. The search routine yields the following table.

<b>low</b>	<b>high</b>	<b>middle</b>	<b>b[middle]</b>
<b>0</b>	<b>5</b>	<b>2</b>	<b>4</b>
<b>low</b>	<b>high</b>	<b>middle</b>	<b>b[middle]</b>
<b>3</b>	<b>5</b>	<b>4</b>	<b>12</b>
<b>low</b>	<b>high</b>	<b>middle</b>	<b>b[middle]</b>
<b>3</b>	<b>3</b>	<b>3</b>	<b>10</b>

Note – variables **low**, **high**, and **middle** always have index values

**Item found at index# 3**

Student Name \_\_\_\_\_

### Binary Search - Advantages and Disadvantages

The binary search model, another type of searching algorithm, is very fast compared to a linear search in that fewer comparisons are needed. One disadvantage of this model is that the list must be sorted.

### Binary Search - Number of Comparisons

Binary Search (BS) logic:

1<sup>st</sup> pass – N elements are considered

2<sup>nd</sup> pass – N/2 elements are eliminated leaving N/2 elements to consider

::

Nth pass- another N/2 elements are eliminated leaving N/2 to consider

In general, after  $p$  passes through the loop, the number of values remaining to be searched is  $N/(2^p)$ . For the *worst* case, the search can continue until there is less than or equal to 1 element remaining. This expressed mathematically as  $N/(2^p) \leq 1$  or if rephrased  $p$  can be expressed as the smallest integer such that  $2^p \geq N$ .

Example:

For a 1000 element array, N is 1000 and the maximum number of passes,  $p$ , for the search is 10.

Student Name \_\_\_\_\_

## Sorting and Searching

### Bubble Sort

One method used to sort an array is known as a bubble sort. During such a sort, the larger elements "bubble up" to the top (the high end) of the array similar to the bubbles in a carbonated beverage.

An example bubble sort program is shown below.

#### Bubble Sort Program

```
#include <iostream>
using namespace std;

void main()
{
    const int SIZE = 6;
    int a[SIZE] = {3,2,1,6,5,4};

    cout << " Unsorted Array\n" << endl;
    for ( int i = 0; i <= SIZE - 1; i++)

    cout << " a[" << i << "] = " << a[i] << endl;

    int temp_hold;

    for (int pass = 1; pass <= SIZE - 1; pass++) // multiple passes

    for (int i = 0; i <= SIZE - 2; i++) //one pass

    if (a[i] > a[i + 1]) // one comparison

    {
        temp_hold = a[i]; // one swap
        a[i] = a[i + 1];
        a[i + 1] = temp_hold;
    }

    cout << "\n Sorted Array\n" << endl;
    for ( i = 0; i <= SIZE - 1; i++)
    cout << " a[" << i << "] = " << a[i] << endl;
    cout << "\n\n";
}
```

The function of the above sort is to sequence through the array a and check, element by element, if elements need to be swapped such that the array will be placed in ascending order.

The output of this sample program follows immediately.

Student Name \_\_\_\_\_

Unsorted Array

```
a[0] = 3
a[1] = 2
a[2] = 1
a[3] = 6
a[4] = 5
a[5] = 4
```

Sorted Array

```
a[0] = 1
a[1] = 2
a[2] = 3
a[3] = 4
a[4] = 5
a[5] = 6
```

A more efficient version of the prior program follows below. This new version uses minimal time to sort the array.

#### Bubble Sort Program (Efficient Version)

```
#include <iostream>
using namespace std;

void main() {
    const int SIZE = 6;
    int a[SIZE] = {3,2,1,6,5,4};

    cout << " Unsorted Array\n" << endl;
    for ( int i = 0; i <= SIZE - 1; i++)

    cout << " a[" << i << "] = " << a[i] << endl;

    int temp_hold, flag = 1;
    while(flag)          // set flag to reduce comparisons
    {
        flag = 0;
        for (int i = 0; i <= SIZE - 2; i++)          // one pass
            if (a[i] > a[i + 1] )
                {
                    temp_hold = a[i];          // one comparison
                    a[i] = a[i + 1];          // one swap
                    a[i + 1] = temp_hold;
                }
        flag = 1;
    }

    cout << "\n Sorted Array\n" << endl;
    for ( i = 0; i <= SIZE - 1; i++)
    cout << " a[" << i << "] = " << a[i] << endl;
    cout << "\n\n";
}
}
```

Student Name \_\_\_\_\_

### Searching Algorithms - Built in Binary Search Function

```
#include <iostream.h>
#include <stdlib.h>
#include <string.h>

typedef int (*fptr)(const void*, const void*);

int CompareThis(char Item1[], char Item2[]);

struct Dept
{
    char stName[21];
    int iNumber;
}Department[] = {"Accounts",30}, {"Benefits",40}, {"Mail Room",50},
{"Payables",60}, {"Payroll",70}};

void main()
{
    //char stName[21];
    //int iNumber;
    //dept[] =
    char stKey[21];
    cout << "enter dept name";
    cin >> stKey;

    Dept *Result = (Dept *) bsearch(stKey, Department,
    sizeof(Department),sizeof(Dept),(fptr) CompareThis);

    if(Result == NULL)
    cout << "dept not found\n":
    else
    {
        int iSubscript = Result - Department;
        cout << "\nDepartment number is "
            << Department[iSubscript].iNumber;
    }
}

int CompareThis(char Item1[], char Item2[])
{
    return(stricmp(Item1, Item2));
}
```

Student Name \_\_\_\_\_

Special section-Clock functions to time your sorts

System Clock-What it does. The system clock ticks at a rate approximating 18.2 ticks per second. By using the **clock()** function you can get a value returned that approximates the amount of time the calling program has been running.

To transform the value into seconds, simply divide it by the constant **CLOCKS\_PER\_SEC**.

Example: using **clock()**

```
#include<time.h> //clock()
#include<stdio.h> //printf()

void elapsed_time(void)
{
    printf("Time elapsed = %u secs\n", clock()/CLOCKS_PER_SEC);
}
```

Example: using **difftime()** a function which measures a starting and stopping point.

The example below gives the time in seconds it takes to loop from 0 to 100000.

Note **difftime()** prototype-**double difftime(time\_t time2, time\_t time1);**

```
#include<time.h> //difftime()
#include<stdio.h> //printf()

void main() {

    time_t begin, end; //create objects of type time_t
    long unsigned i;
    begin= time(NULL);
    for (i=0;i<100000; i++ ) ;
    end = time(NULL);
    printf("Loop used %f secs\n", difftime(end,begin));

}
```