

Rajalakshmi Engineering College

Rajalakshmi Nagar, Thandalam, Chennai - 602 105

Department of Computer Science and Engineering



CS19241 - Data Structures

Unit - V

Lecture Notes

(Regulations - 2019)



Prepared by:

B.BHUVANESWARAN

Assistant Professor (SG) / CSE / REC

bhuvaneswaran@rajalakshmi.edu.in

1.1 INTRODUCTION

Searching refers to determining whether an element is present in a given list of elements or not. If the element is found to be present in the list then the search is considered as successful, otherwise it is considered as an unsuccessful search. The search operation returns the location or address of the element found. There are various searching methods that can be employed to perform search on a data set. The choice of a particular searching method in a given situation depends on a number of factors, such as:

1. Order of elements in the list, i.e., random or sorted.
2. Size of the list Let us explore the various searching methods one by one.

1.2 TYPES OF SEARCHING

- Linear search
- Binary search

REVIEW QUESTIONS

1. What is searching?
2. What are the types of searching?

2.1 INTRODUCTION

It is one of the conventional searching techniques that sequentially searches for an element in the list. It typically starts with the first element in the list and moves towards the end in a step-by-step fashion. In each iteration, it compares the element to be searched with the list element, and if there is a match, the location of the list element is returned.

Consider an array of integers A containing n elements. Let k be the value that needs to be searched. The linear search technique will first compare A[0] with k to find out if they are same. If the two values are found to be same then the index value, i.e., 0 will be returned as the location containing k. However, if the two values are not same then k will be compared with A[1]. This process will be repeated until the element is not found. If the last comparison between k and A[n- 1] is also negative then the search will be considered as unsuccessful. Figure depicts the linear search technique performed on an array of integers.

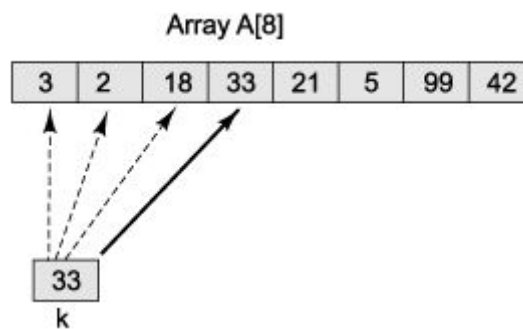


Fig. Linear search

As shown in Fig., the value k is repeatedly compared with each element of the array A. As soon as the element is found, the corresponding index location is returned and the search operation is terminated.

2.2 ALGORITHM

LinearSearch(A[], N, KEY)

A[] : Integer array
N : Integer
KEY : Integer

Step 1 : Start.
Step 2 : Repeat For I = 0 to N-1.
Step 3 : If A[I] = KEY then Goto Step 4 else Goto Step 5.
Step 4 : Return I and Stop.
Step 5 : Increment I by 1.
Step 6 : [End of Step 3 For loop].
Step 7 : Return -1.
Step 8 : Stop.

2.3. ROUTINE

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

2.4 PROGRAM

```
#include <stdio.h>

int LinearSearch(int a[], int n, int key);

int main()
{
    int n, a[10], i, key, pos;
    printf("Enter the limit : ");
    scanf("%d", &n);
    printf("Enter the elements : ");
    for (i = 0; i < n; i++)
        scanf("%d", &a[i]);
    printf("Enter the element to search : ");
    scanf("%d", &key);
    pos = LinearSearch(a, n, key);
    if (pos != -1)
        printf("Element found at location : %d", pos);
    else
        printf("Element not found.");
    return 0;
}

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

Output

```
Enter the limit : 5
Enter the elements : 30 40 50 20 10
Enter the element to search : 20
Element found at location : 3
```

Enter the limit : 5
Enter the elements : 30 40 50 20 10
Enter the element to search : 55
Element not found.

2.5 EFFICIENCY OF LINEAR SEARCH

Assume that an array containing n elements is to be searched for the value k . In the best case, k would be the first element in the list, thus requiring only one comparison. In the worst case, it would be last element in the list, thus requiring n comparisons.

To compute the efficiency of linear search we can add all the possible number of comparisons and divide it by n .

$$\begin{aligned}\text{Thus, efficiency of linear search} &= (1 + 2 + \dots + n) / n \\ &= n(n + 1) / 2n \\ &= O(n)\end{aligned}$$

2.6 ADVANTAGES OF LINEAR SEARCH

Some of the key advantages of linear search technique are:

- It is a simple searching technique that is easy to implement.
- It does not require the list to be sorted in a particular order.

2.7 LIMITATIONS OF LINEAR SEARCH

The disadvantages associated with it are as follows:

- It is quite inefficient for large sized lists.
- It does not leverage the presence of any pre-existing sort order in a list.

REVIEW QUESTIONS

1. What is a linear search?
2. What is the efficiency of linear search?
3. What are the advantages and disadvantages of linear search?

3.1 INTRODUCTION

Binary search technique has a prerequisite – it requires the elements of a data structure (list) to be already arranged in a sorted manner before search can be performed in it. It begins by comparing the element that is present at the middle of the list. If there is a match, then the search ends immediately and the location of the middle element is returned. However, if there is a mismatch then it focuses the search either in the left or the right sub list depending on whether the target element is lesser than or greater than middle element. The same methodology is repeatedly followed until the target element is found.

Consider an array of integers A containing eight elements, as shown in Fig. Let $k = 21$ be the value that needs to be searched.

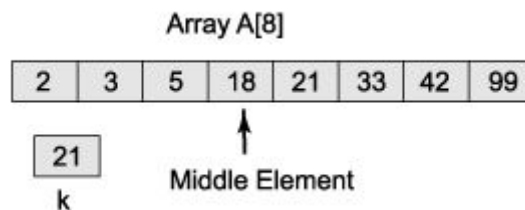


Fig. Binary search

As we can see in Fig., the array A on which binary search is to be performed is already sorted. The following steps describe how binary search is performed on array A to search for value k:

1. First of all, the middle element in the array A is identified, which is 18.
2. Now, k is compared with 18. Since k is greater than 18, the search is focused on the right sub list.
3. The middle element in the right sub list is 33. Since k is less than 33, the search is focused on the left sub list, which is {21, 33}.
4. Now, again k is compared with the middle element of {21, 33}, which is 21. Thus, it matches with k.
5. The index value of 21, i.e., 4 is returned and the search is considered as successful.

3.2 ALGORITHM

BinarySearch (A[], N, KEY)

A[] : Integer array
 N : Integer
 KEY : Integer

- Step 1 : Start.
- Step 2 : Set FIRST = 0.
- Step 3 : Set LAST = N – 1.
- Step 4 : Repeat While FIRST <= LAST.
- Step 5 : Set MID = (FIRST + LAST) / 2.
- Step 6 : If A[MID] = KEY then Goto Step 7 else Goto Step 8.
- Step 7 : Return MID and Stop.
- Step 8 : If A[MID] < KEY then Goto Step 9 else Goto Step 10.

| | | |
|---------|---|---------------------------------------|
| Step 9 | : | Set FIRST = MID + 1 and Goto Step 11. |
| Step 10 | : | Set LAST = MID - 1. |
| Step 11 | : | [End of Step 4 While loop]. |
| Step 12 | : | Return -1. |
| Step 14 | : | Stop. |

3.3 ROUTINE

```

int BinarySearch(int a[], int n, int key)
{
    int first, last, mid;
    first = 0;
    last = n - 1;
    while (first <= last)
    {
        mid = (first + last) / 2;
        if (a[mid] == key)
            return mid;
        else if (a[mid] < key)
            first = mid + 1;
        else
            last = mid - 1;
    }
    return -1;
}

```

3.4 PROGRAM

```

#include <stdio.h>

int BinarySearch(int a[], int n, int key);

int main()
{
    int n, a[10], i, key, pos;
    printf("Enter the limit : ");
    scanf("%d", &n);
    printf("Enter the elements in sorted order : ");
    for (i = 0; i < n; i++)
        scanf("%d", &a[i]);
    printf("Enter the element to search : ");
    scanf("%d", &key);
    pos = BinarySearch(a, n, key);
    if (pos != -1)
        printf("Element found at location : %d", pos);
    else
        printf("Element not found.");
    return 0;
}

```

```

int BinarySearch(int a[], int n, int key)
{
    int first, last, mid;
    first = 0;
    last = n - 1;
    while (first <= last)
    {
        mid = (first + last) / 2;
        if (a[mid] == key)
            return mid;
        else if (a[mid] < key)
            first = mid + 1;
        else
            last = mid - 1;
    }
    return -1;
}

```

Output

Enter the limit : 5
 Enter the elements in sorted order : 10 20 30 40 50
 Enter the element to search : 30
 Element found at location : 2

Enter the limit : 5
 Enter the elements in sorted order : 10 20 30 40 50
 Enter the element to search : 55
 Element not found.

3.5 EFFICIENCY OF BINARY SEARCH

Best Case

The best case for a binary search algorithm occurs when the element to be searched is present at the middle of the list. In this case, only one comparison is made to perform the search operation.

Thus, efficiency = $O(1)$

Worst Case

The worst case for a binary search algorithm occurs when the element to be searched is not present in the list. In this case, the list is continuously divided until only one element is left for comparison.

Let n be the number of list elements and c be the total number of comparisons made in the worst case.

Now, after every single comparison, the number of list elements left to be searched is reduced by 2.

Thus, $c = \log_2 n$

Hence, efficiency = $O(\log_2 n)$

3.6 ADVANTAGES OF BINARY SEARCH

Some of the key advantages of binary search technique are:

- It requires lesser number of iterations.
- It is a lot faster than linear search.

3.7 LIMITATIONS OF BINARY SEARCH

The disadvantages associated with it are as follows:

- Unlike linear search, it requires the list to be sorted before search can be performed.
- In comparison to linear search, the binary search technique may seem to be a little difficult to implement.

REVIEW QUESTIONS

1. What is a binary search?
2. What is the worst case efficiency of binary search?
3. What is the time complexity of binary search?
4. What are the advantages and disadvantages of binary search?

11.1 INTRODUCTION

Sorting and searching are two of the most common operations performed by computers all around the world. The sorting operation arranges the numerical and alphabetical data present in a list, in a specific order or sequence. Searching, on the other hand, locates a specific element across a given list of elements. At times, a list may require sorting before the search operation can be performed on it.

Sorting Techniques

The sorting techniques are categorized into:

- Internal Sorting
- External Sorting

11.2 INTERNAL SORTING

All sorting techniques which require the data set to be present in the main memory are referred as internal sorting techniques. Examples:

- Insertion sort
- Selection sort
- Shell sort
- Bubble sort
- Quick sort
- Heap sort

11.3 EXTERNAL SORTING

External Sorting, takes place in the secondary memory of a computer. Since the number of objects to be sorted is too large to fit in main memory. Examples:

- Merge Sort
- Multiway Merge
- Polyphase merge

REVIEW QUESTIONS

1. What is sorting?
2. What are the types of sorting?
3. Differentiate between internal sorting and external sorting. (or) Differentiate internal and external sorting.
4. List the four types of sorting techniques.
5. What is the need for external sorting?
6. List sorting algorithm which uses logarithmic time complexity.

12.1. INTRODUCTION

Insertion sort works by taking elements from the list one by one and inserting them in their current position into a new sorted list. Insertion sort consists of $N - 1$ passes, where N is the number of elements to be sorted. The i^{th} pass of insertion sort will insert the i^{th} element $A[i]$ into its rightful place among $A[1], A[2], \dots, A[i-1]$. After doing this insertion the records occupying $A[1] \dots A[i]$ are in sorted order.

To understand the insertion sorting method, consider a scenario where an array A containing n elements needs to be sorted. Now, each pass of the insertion sorting method will insert the element $A[i]$ into its appropriate position in the previously sorted subarray, i.e., $A[1], A[2], \dots, A[i-1]$.

The following list describes the tasks performed in each of the passes:

Pass 1: $A[2]$ is compared with $A[1]$ and inserted either before or after $A[1]$. This makes $A[1], A[2]$ a sorted sub array.

Pass 2: $A[3]$ is compared with both $A[1]$ and $A[2]$ and inserted at an appropriate place. This makes $A[1], A[2], A[3]$ a sorted sub array.

Pass $n-1$: $A[n]$ is compared with each element in the sub array $A[1], A[2], A[3], \dots, A[n-1]$ and inserted at an appropriate position. This eventually makes the entire array A sorted.

12.2. ALGORITHM

InsertionSort($A[], N$)

$A[]$: Integer array
 N : Integer

Step 1 : Start.
Step 2 : Repeat For $I = 1$ to $N-1$.
Step 3 : Set $TEMP = A[I]$.
Step 4 : Set $J = I$.
Step 5 : Repeat While $J > 0$ and $A[J - 1] > TEMP$.
Step 6 : Set $A[J] = A[J - 1]$.
Step 7 : Decrement J by 1.
Step 8 : [End of Step 5 While loop].
Step 9 : Set $A[J] = TEMP$.
Step 10 : Increment I by 1.
Step 11 : [End of Step 2 For loop].
Step 12 : Stop.

12.3 ROUTINE

```
void InsertionSort(int a[], int n)
{
    int i, j, temp;
    for (i = 1; i < n; i++)
    {
        temp = a[i];
        j = i;
        while (j > 0 && a[j - 1] > temp)
        {
            a[j] = a[j - 1];
            j = j - 1;
        }
        a[j] = temp;
    }
}
```

12.4. EXAMPLE

Consider an unsorted array as follows:

20 10 60 40 30 15

Passes of Insertion Sort

| Original | 20 | 10 | 60 | 40 | 30 | 15 | Positions Moved |
|--------------|----|----|----|----|----|----|-----------------|
| After i = 1 | 10 | 20 | 60 | 40 | 30 | 15 | 1 |
| After i = 2 | 10 | 20 | 60 | 40 | 30 | 15 | 0 |
| After i = 3 | 10 | 20 | 40 | 60 | 30 | 15 | 1 |
| After i = 4 | 10 | 20 | 30 | 40 | 60 | 15 | 2 |
| After i = 5 | 10 | 15 | 20 | 30 | 40 | 60 | 4 |
| Sorted Array | 10 | 15 | 20 | 30 | 40 | 60 | |

12.5 PROGRAM

```
#include <stdio.h>

void InsertionSort(int a[], int n);

int main()
{
    int n, i, a[10];
    printf("Enter the limit : ");
    scanf("%d", &n);
    printf("Enter the elements : ");
    for (i = 0; i < n; i++)
        scanf("%d", &a[i]);
    InsertionSort(a, n);
}
```

```

        printf("The sorted elements are : ");
        for (i = 0; i < n; i++)
            printf("%d\t", a[i]);
        return 0;
    }

void InsertionSort(int a[], int n)
{
    int i, j, temp;
    for (i = 1; i < n; i++)
    {
        temp = a[i];
        j = i;
        while (j > 0 && a[j - 1] > temp)
        {
            a[j] = a[j - 1];
            j = j - 1;
        }
        a[j] = temp;
    }
}

```

Output

Enter the limit : 5
Enter the elements : 30 40 50 20 10
The sorted elements are : 10 20 30 40 50

12.6 EFFICIENCY OF INSERTION SORT

Assume that an array containing n elements is sorted using insertion sort technique.

- The minimum number of elements that must be scanned = $n - 1$.
- For each of the elements the maximum number of shifts possible = $n - 1$.
- Thus, efficiency of insertion sort = $O(n^2)$

12.7 ANALYSIS OF INSERTION SORT

| | | |
|-----------------------|---|----------|
| Best case analysis | : | $O(n)$ |
| Average case analysis | : | $O(n^2)$ |
| Worst case analysis | : | $O(n^2)$ |

12.8 ADVANTAGES OF INSERTION SORT

Some of the key advantages of insertion sorting technique are:

- It is one of the simplest sorting techniques that is easy to implement.
- It performs well in case of smaller lists.
- It leverages the presence of any existing sort pattern in the list, thus resulting in better efficiency.

12.9 LIMITATIONS OF INSERTION SORT

The disadvantages associated with insertion sorting technique are as follows:

- The efficiency of $O(n^2)$ is not well suited for large sized lists.
- It is expensive because of shifting all following elements by one.

REVIEW QUESTIONS

1. Give the working principle of insertion sort.
2. What is the time complexity of Insertion sort algorithm? (or) What is the time complexity of the insertion sort?
3. What is the running time of insertion sort if all keys are equal?
4. Sort the sequence 3, 1, 4, 1, 5, 9, 2, 6, 5 using insertion sort.

13.1 INTRODUCTION

Selection sort is one of the most basic sorting techniques. It works on the principle of identifying the smallest element in the list and moving it to the beginning of the list. This process is repeated until all the elements in the list are sorted.

13.2 ALGORITHM

SelectionSort(A[], N)

A[] : Integer array
N : Integer

Step 1 : Start.
Step 2 : Repeat For I = 0 to N - 1.
Step 3 : Set MIN = I.
Step 4 : Repeat For J = I + 1 to N.
Step 5 : If A[J] < A[MIN] then Goto Step 6 else Goto Step 7.
Step 6 : Set MIN = J.
Step 7 : Increment J by 1.
Step 8 : [End of Step 4 For loop].
Step 9 : Set TEMP = A[I].
Step 10 : Set A[I] = A[MIN].
Step 11 : Set A[MIN] = TEMP.
Step 12 : Increment I by 1.
Step 13 : [End of Step 2 For loop].
Step 14 : Stop.

13.3 ROUTINE

```
void SelectionSort(int a[], int n)
{
    int i, j, min, temp;
    for (i = 0; i < n - 1; i++)
    {
        min = i;
        for (j = i + 1; j < n; j++)
        {
            if (a[j] < a[min])
                min = j;
        }
        temp = a[i];
        a[i] = a[min];
        a[min] = temp;
    }
}
```

13.4. EXAMPLE

Let us consider an example where a list L contains five integers stored in a random fashion, as shown:

| | | | | |
|----|---|---|----|----|
| 18 | 3 | 2 | 33 | 21 |
|----|---|---|----|----|



List L

Now, if the list L is sorted using selection sort technique then first of all the first element in the list, i.e., 18 will be selected and compared with all the remaining elements in the list. The element which is found to be the lowest amongst the remaining set of elements will be swapped with the first element. Then, the second element will be selected and compared with the remaining elements in the list. This process is repeated until all the elements are rearranged in a sorted manner.

Passes of Selection Sort

Table illustrates the sorting of list L in ascending order using selection sort.

A single iteration of the selection sorting technique that brings the smallest element at the beginning of the list is called a pass. As we can see in the above table, four passes were required to sort a list of five elements. Hence, we can say that selection sort requires $n-1$ passes to sort an array of n elements.

| Pass | Comparison | Resultant Array | | | | | | | | | | |
|--|---|-----------------|----|----|----|----|---|---|---|----|----|----|
| 1 | <table><tr><td>18</td><td>3</td><td>2</td><td>33</td><td>21</td></tr></table> | 18 | 3 | 2 | 33 | 21 | <table><tr><td>2</td><td>3</td><td>18</td><td>33</td><td>21</td></tr></table> | 2 | 3 | 18 | 33 | 21 |
| 18 | 3 | 2 | 33 | 21 | | | | | | | | |
| 2 | 3 | 18 | 33 | 21 | | | | | | | | |
| 2 | <table><tr><td>2</td><td>3</td><td>18</td><td>33</td><td>21</td></tr></table> | 2 | 3 | 18 | 33 | 21 | <table><tr><td>2</td><td>3</td><td>18</td><td>33</td><td>21</td></tr></table> | 2 | 3 | 18 | 33 | 21 |
| 2 | 3 | 18 | 33 | 21 | | | | | | | | |
| 2 | 3 | 18 | 33 | 21 | | | | | | | | |
| 3 | <table><tr><td>2</td><td>3</td><td>18</td><td>33</td><td>21</td></tr></table> | 2 | 3 | 18 | 33 | 21 | <table><tr><td>2</td><td>3</td><td>18</td><td>33</td><td>21</td></tr></table> | 2 | 3 | 18 | 33 | 21 |
| 2 | 3 | 18 | 33 | 21 | | | | | | | | |
| 2 | 3 | 18 | 33 | 21 | | | | | | | | |
| 4 | <table><tr><td>2</td><td>3</td><td>18</td><td>33</td><td>21</td></tr></table> | 2 | 3 | 18 | 33 | 21 | <table><tr><td>2</td><td>3</td><td>18</td><td>21</td><td>33</td></tr></table> | 2 | 3 | 18 | 21 | 33 |
| 2 | 3 | 18 | 33 | 21 | | | | | | | | |
| 2 | 3 | 18 | 21 | 33 | | | | | | | | |
| <p> denotes the currently selected element</p> <p> denotes the smallest element identified in the current pass</p> | | | | | | | | | | | | |

13.5. PROGRAM

```
#include <stdio.h>

void SelectionSort(int a[], int n);

int main()
{
    int i, n, a[10];
    printf("Enter the limit : ");
    scanf("%d", &n);
    printf("Enter the elements : ");
    for (i = 0; i < n; i++)
        scanf("%d", &a[i]);
    SelectionSort(a, n);
}
```



```

    printf("The sorted elements are : ");
    for (i = 0; i < n; i++)
        printf("%d\t", a[i]);
    return 0;
}

void SelectionSort(int a[], int n)
{
    int i, j, min, temp;
    for (i = 0; i < n - 1; i++)
    {
        min = i;
        for (j = i + 1; j < n; j++)
        {
            if (a[j] < a[min])
                min = j;
        }
        temp = a[i];
        a[i] = a[min];
        a[min] = temp;
    }
}

```

Output

Enter the limit : 5

Enter the elements : 30 40 50 20 10

The sorted elements are : 10 20 30 40 50

13.6 EFFICIENCY OF SELECTION SORT

Assume that an array containing n elements is sorted using selection sort technique.

Now, the number of comparisons made during first pass = $n - 1$.

Number of comparisons made during second pass = $n - 2$.

Number of comparisons made during last pass = 1.

$$\begin{aligned}
 \text{So, total number of comparisons} &= (n - 1) + (n - 2) + \dots + 1 \\
 &= n * (n - 1) / 2 \\
 &= O(n^2)
 \end{aligned}$$

$$\text{Thus, efficiency of selection sort} = O(n^2)$$

13.7 ANALYSIS OF SELECTION SORT

| | | |
|-----------------------|---|----------|
| Best case analysis | : | $O(n^2)$ |
| Average case analysis | : | $O(n^2)$ |
| Worst case analysis | : | $O(n^2)$ |

13.8 ADVANTAGES OF SELECTION SORT

Some of the key advantages of selection sorting technique are:

- It is one of the simplest of sorting techniques.
- It is easy to understand and implement.
- It performs well in case of smaller lists.
- It does not require additional memory space to perform sorting.

13.9 LIMITATIONS OF SELECTION SORT

The disadvantages associated with selection sort that prevent the programmers from using it often are as follows:

- The efficiency of $O(n^2)$ is not well suited for large sized lists.
- It does not leverage the presence of any existing sort pattern in the list.

REVIEW QUESTIONS

1. Give the working principle of selection sort.
2. What is the time complexity of selection sort algorithm?

14.1 INTRODUCTION

Shell sort is an algorithm that first sorts the elements far apart from each other and successively reduces the interval between the elements to be sorted. It is a generalized version of insertion sort.

In shell sort, elements at a specific interval are sorted. The interval between the elements is gradually decreased based on the sequence used. The performance of the shell sort depends on the type of sequence used for a given input array.

Some of the optimal sequences used are:

- Shell's original sequence: $N/2, N/4, \dots, 1$
- Knuth's increments: $1, 4, 13, \dots, (3k - 1) / 2$
- Sedgewick's increments: $1, 8, 23, 77, 281, 1073, 4193, 16577 \dots 4j+1 + 3 \cdot 2j+1$
- Hibbard's increments: $1, 3, 7, 15, 31, 63, 127, 255, 511 \dots$
- Papernov & Stasevich increment: $1, 3, 5, 9, 17, 33, 65, \dots$
- Pratt: $1, 2, 3, 4, 6, 9, 8, 12, 18, 27, 16, 24, 36, 54, 81 \dots$

14.2. ALGORITHM

ShellSort(A[], N)

A[] : Integer array
N : Integer

Step 1 : Start.
Step 2 : Repeat For GAP = N / 2 to 0.
Step 3 : Repeat For I = GAP to N.
Step 4 : Set TEMP = A[I].
Step 5 : Repeat For J = I to J >= GAP and A[J - GAP] > TEMP.
Step 6 : Set A[J] = A[J - GAP].
Step 7 : [End of Step 5 For loop].
Step 8 : Set A[J] = TEMP.
Step 9 : [End of Step 3 For loop].
Step 10 : [End of Step 2 For loop].
Step 11 : Stop.

14.3 ROUTINE

```
void ShellSort(int a[], int n)
{
    int gap, i, j, temp;
    for (gap = n / 2; gap > 0; gap /= 2)
    {
        for (i = gap; i < n; i += 1)
        {
            temp = a[i];
```

```

        for (j = i; j >= gap && a[j - gap] > temp; j -= gap)
        {
            a[j] = a[j - gap];
        }
        a[j] = temp;
    }
}

```

14.4 EXAMPLE

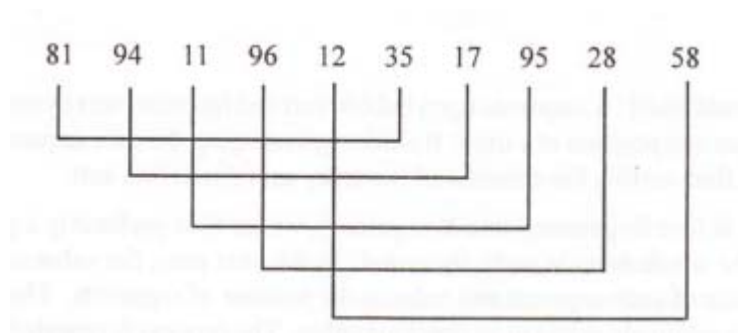
Consider an unsorted array as follows:

81 94 11 96 12 35 17 95 28 58

Here $N = 10$, the first pass as $K = 5$ ($10/2$)

81 94 11 96 12 35 17 95 28 58

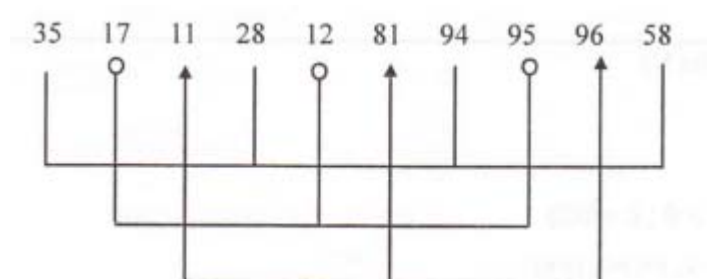
Passes of Shell Sort



After first pass:

35 17 11 28 12 81 94 95 96 58

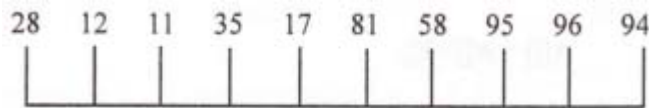
In second pass, K is reduced to 3.



After second pass:

28 12 11 35 17 81 58 95 96 94

In third pass, K is reduced to 1.



The final sorted array is:

11 12 17 28 35 58 81 94 95 96

14.5 PROGRAM

```
#include <stdio.h>
void ShellSort(int a[], int n);

int main()
{
    int i, n, a[10];
    printf("Enter the limit : ");
    scanf("%d", &n);
    printf("Enter the elements : ");
    for (i = 0; i < n; i++)
        scanf("%d", &a[i]);
    ShellSort(a, n);
    printf("The sorted elements are : ");
    for (i = 0; i < n; i++)
        printf("%d\t", a[i]);
    return 0;
}

void ShellSort(int a[], int n)
{
    int gap, i, j, temp;
    for (gap = n / 2; gap > 0; gap /= 2)
    {
        for (i = gap; i < n; i += 1)
        {
            temp = a[i];
            for (j = i; j >= gap && a[j - gap] > temp; j -= gap)
            {
                a[j] = a[j - gap];
            }
            a[j] = temp;
        }
    }
}
```

Output

Enter the limit : 5

Enter the elements : 30 40 50 20 10

The sorted elements are : 10 20 30 40 50

14.6 ANALYSIS OF SHELL SORT

| | | |
|-----------------------|---|---------------|
| Best case analysis | : | $O(n \log n)$ |
| Average case analysis | : | $O(n^{1.5})$ |
| Worst case analysis | : | $O(n^2)$ |

14.7 ADVANTAGES OF SHELL SORT

Some of the key advantages of shell sorting technique are:

- It is one of the fastest sorting techniques for sorting small number of elements.
- It requires relatively small amount of memory.

REVIEW QUESTIONS

1. Give the working principle of shell sort.
2. What is the time complexity of shell sort algorithm?

15.1 INTRODUCTION

Bubble sort is one of the simplest internal sorting algorithms. Bubble sort works by comparing two consecutive elements and the largest element among these two bubbles towards right. At the end of the first pass the largest element gets sorted and placed at the end of the sorted list. This process is repeated for all pairs of the elements until it moves the largest element to the end of the list in that iteration.

Bubble sort consists of $n-1$ passes, where 'n' is the number of elements to be sorted. In the 1st pass, the largest element will be placed in the n^{th} position. In the 2nd pass, the second largest element will be placed in the $(n-1)^{\text{th}}$ position. In $(n-1)^{\text{th}}$ pass only the first two elements are compared.

15.2 ALGORITHM

BubbleSort(A[], N)

| | | |
|---------|---|---|
| A[] | : | Integer array |
| N | : | Integer |
| Step 1 | : | Start. |
| Step 2 | : | Repeat For I = 0 to N-1. |
| Step 3 | : | Repeat For J = I+1 to N-1-I. |
| Step 4 | : | If A[J] > A[J+1] then Goto Step 5 else Goto Step 8. |
| Step 5 | : | Set T = A[J]. |
| Step 6 | : | Set A[J] = A[J+1]. |
| Step 7 | : | Set A[J+1] = T. |
| Step 8 | : | Increment J by 1. |
| Step 9 | : | [End of Step 3 For loop]. |
| Step 10 | : | Increment I by 1. |
| Step 11 | : | [End of Step 2 For loop]. |
| Step 12 | : | Stop. |

15.3 ROUTINE

```
void BubbleSort(int a[], int n)
{
    int i, j, t;
    for (i = 0; i < n - 1; i++)
    {
        for (j = 0; j < n - 1 - i; j++)
        {
            if (a[j] > a[j + 1])
            {
                t = a[j];
                a[j] = a[j + 1];
                a[j + 1] = t;
            }
        }
    }
}
```

15.4 EXAMPLE

Let us consider an example where a list L contains five integers stored in a random fashion, as shown:

| | | | | |
|----|---|---|----|----|
| 18 | 3 | 2 | 33 | 21 |
|----|---|---|----|----|

List L

Passes of Bubble Sort

Table illustrates the sorting of list L in ascending order using bubble sort:

| Pass | Comparison | Resultant Array |
|---|---|---------------------|
| 1 | <div><div>18323321</div><div>31823321</div><div>32183321</div><div>32183321</div></div> | <div>32182133</div> |
| 2 | <div><div>32182133</div><div>23182133</div><div>23182133</div></div> | <div>23182133</div> |
| 3 | <div><div>23182133</div><div>23182133</div></div> | <div>23182133</div> |
| 4 | <div><div>23182133</div></div> | <div>23182133</div> |
| ⏟ → denotes the pair of consecutive elements being compared | | |

As we can see in the above illustration, four passes were required to sort a list of five elements. Hence, we can say that bubble sort requires $n-1$ passes to sort an array of n elements.

15.5 PROGRAM

```
#include <stdio.h>

void BubbleSort(int a[], int n);

int main()
{
    int a[10], i, n;
    printf("Enter the limit : ");
    scanf("%d", &n);
    printf("Enter the numbers : ");
    for (i = 0; i < n; i++)
        scanf("%d", &a[i]);
    BubbleSort(a, n);
    printf("The sorted elements are : ");
    for (i = 0; i < n; i++)
        printf("%d\t", a[i]);
    return 0;
}
```

```

void BubbleSort(int a[], int n)
{
    int i, j, t;
    for (i = 0; i < n - 1; i++)
    {
        for (j = 0; j < n - 1 - i; j++)
        {
            if (a[j] > a[j + 1])
            {
                t = a[j];
                a[j] = a[j + 1];
                a[j + 1] = t;
            }
        }
    }
}

```

Output

Enter the limit : 5

Enter the numbers : 30 40 50 20 10

The sorted elements are : 10 20 30 40 50

15.6. EFFICIENCY OF BUBBLE SORT

Assume that an array containing n elements is sorted using bubble sort technique.

| | |
|---|--|
| Number of comparisons made in first pass | $= n-1.$ |
| Number of comparisons made in second pass | $= n-2.$ |
| Number of comparisons made in last pass | $= 1.$ |
| Total number of comparisons made | $= (n-1) + (n-2) + \dots + 1$ $= n * (n-1) / 2$ $= O(n^2)$ |

Thus, efficiency of bubble sort $= O(n^2)$.

15.7. ANALYSIS OF BUBBLE SORT

| | | |
|-----------------------|---|----------|
| Best case analysis | : | $O(n^2)$ |
| Average case analysis | : | $O(n^2)$ |
| Worst case analysis | : | $O(n^2)$ |

15.8. ADVANTAGES OF BUBBLE SORT

Some of the key advantages of bubble sorting technique are:

- It is easy to understand and implement.
- It leverages the presence of any existing sort pattern in the list, thus resulting in better efficiency.

15.9. LIMITATIONS OF BUBBLE SORT

The disadvantages associated with bubble sorting technique are given below:

- The efficiency of $O(n^2)$ is not well suited for large sized lists.
- It requires large number of elements to be shifted.
- It is slow in execution as large elements are moved towards the end of the list in a step-by-step fashion.

REVIEW QUESTIONS

1. Give the working principle of bubble sort.
2. What is the time complexity of bubble sort algorithm?
3. Sort the following numbers using bubble sort 10, 5, 7, 11, 4, 1.

16.1. INTRODUCTION

Quick Sort is the most efficient internal sorting technique. It possesses a very good average case behaviour among all the sorting techniques. It is also called partitioning sort which uses divide and conquer techniques.

The quick sort works by partitioning the array $A[1], A[2] \dots A[n]$ by picking some key value in the array as a pivot element. Pivot element is used to rearrange the elements in the array. Pivot can be the first element of an array and the rest of the elements are moved so that the elements on left side of the pivot are lesser than the pivot, whereas those on the right side are greater than the pivot. Now, the pivot element is placed in its correct position. Now the quicksort procedure is applied for left array and right array in a recursive manner.

16.2. ALGORITHM

QuickSort($A[]$, LEFT, RIGHT)

$A[]$: Integer array
LEFT, RIGHT : Integer

Step 1 : Start.
Step 2 : If LEFT < RIGHT then Goto Step 3 else Goto Step 23.
Step 3 : Set PIVOT = LEFT.
Step 4 : Set I = LEFT + 1.
Step 5 : Set J = RIGHT.
Step 6 : Repeat While I < J.
Step 7 : Repeat While $A[I] < A[PIVOT]$.
Step 8 : Increment I by 1.
Step 9 : [End of Step 7 While loop].
Step 10 : Repeat While $A[J] > A[PIVOT]$.
Step 11 : Decrement J by 1.
Step 12 : [End of Step 10 While loop].
Step 13 : If I < J then Goto Step 14 else Goto Step 17.
Step 14 : Set TEMP = $A[I]$.
Step 15 : Set $A[I] = A[J]$.
Step 16 : Set $A[J] = TEMP$.
Step 17 : [End of Step 6 While loop].
Step 18 : Set TEMP = $A[PIVOT]$.
Step 19 : Set $A[PIVOT] = A[J]$.
Step 20 : Set $A[J] = TEMP$.
Step 21 : QUICKSORT(LEFT, J - 1),
Step 22 : QUICKSORT(J + 1, RIGHT).
Step 23 : Stop.

16.3 ROUTINE

```
void QuickSort(int a[], int left, int right)
{
    int i, j, temp, pivot;
    if (left < right) {
        pivot = left;
        i = left + 1;
        j = right;
        while (i < j)
        {
            while (a[i] < a[pivot])
                i++;
            while (a[j] > a[pivot])
                j--;
            if (i < j) {
                temp = a[i];
                a[i] = a[j];
                a[j] = temp;
            }
        }
        temp = a[pivot];
        a[pivot] = a[j];
        a[j] = temp;
        QuickSort(a, left, j - 1);
        QuickSort(a, j + 1, right);
    }
}
```

16.4 EXAMPLE

Consider an unsorted array as follows:

40 20 70 14 60 61 97 30

Here PIVOT = 40, i = 20, j = 30.

- The value of i is incremented till $a[i] \leq \text{Pivot}$ and the value of j is decremented till $a[j] \geq \text{pivot}$, this process is repeated until $i < j$.
- If $a[i] > \text{pivot}$ and $a[j] < \text{pivot}$ and also if $i < j$ then swap $a[i]$ and $a[j]$.
- If $i > j$ then swap $a[j]$ and $a[\text{pivot}]$.

Once the correct location for PIVOT is found, then partition array into left sub array and right subarray, where left sub array contains all the elements less than the PIVOT and right sub array contains all the elements greater than the PIVOT.

Passes of Quick Sort

| | | | | | | | |
|--------------|----------|----|----|----|----|----|----------|
| 40 | 20 | 70 | 14 | 60 | 61 | 97 | 30 |
| Pivot | i | | | | | | j |

| | | | | | | | |
|--------------|----|----------|----|----|----|----|----------|
| 40 | 20 | 70 | 14 | 60 | 61 | 97 | 30 |
| Pivot | | i | | | | | j |

As $i < j$, swap($a[i]$, $a[j]$), swap(70, 30).

| | | | | | | | |
|--------------|----|----------|----|----|----|----|----------|
| 40 | 20 | 30 | 14 | 60 | 61 | 97 | 70 |
| Pivot | | i | | | | | j |

| | | | | | | | |
|--------------|----|----|----------|----|----|----|----------|
| 40 | 20 | 30 | 14 | 60 | 61 | 97 | 70 |
| Pivot | | | i | | | | j |

| | | | | | | | |
|--------------|----|----|----|----------|----|----|----------|
| 40 | 20 | 30 | 14 | 60 | 61 | 97 | 70 |
| Pivot | | | | i | | | j |

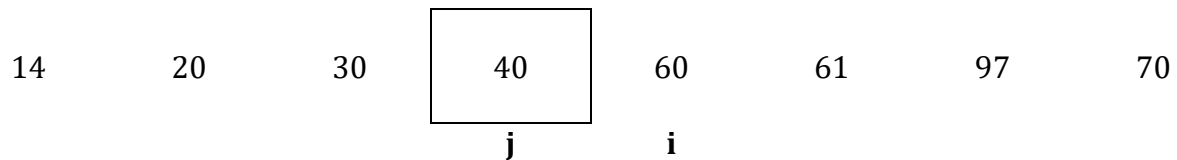
| | | | | | | | |
|--------------|----|----|----|----------|----|----------|----|
| 40 | 20 | 30 | 14 | 60 | 61 | 97 | 70 |
| Pivot | | | | i | | j | |

| | | | | | | | |
|--------------|----|----|----|----------|----------|----|----|
| 40 | 20 | 30 | 14 | 60 | 61 | 97 | 70 |
| Pivot | | | | i | j | | |

| | | | | | | | |
|--------------|----|----|----|-----------|----|----|----|
| 40 | 20 | 30 | 14 | 60 | 61 | 97 | 70 |
| Pivot | | | | ij | | | |

| | | | | | | | |
|--------------|----|----|----------|----------|----|----|----|
| 40 | 20 | 30 | 14 | 60 | 61 | 97 | 70 |
| Pivot | | | j | i | | | |

As $i > j$, swap($a[j]$, Pivot), swap(14, 40).



Now, the pivot element has reached its correct position. The elements lesser than the Pivot {14, 20, 30} is considered as left sub array. The elements greater than the pivot {60, 61, 97, 70} is considered as right sub array. Then the QuickSort procedure is applied recursively for both these arrays.

16.5 PROGRAM

```
#include <stdio.h>

void QuickSort(int a[], int left, int right);

int main()
{
    int i, n, a[10];
    printf("Enter the limit : ");
    scanf("%d", &n);
    printf("Enter the elements : ");
    for (i = 0; i < n; i++)
        scanf("%d", &a[i]);
    QuickSort(a, 0, n - 1);
    printf("The sorted elements are : ");
    for (i = 0; i < n; i++)
        printf("%d\t", a[i]);
    return 0;
}

void QuickSort(int a[], int left, int right)
{
    int i, j, temp, pivot;
    if (left < right) {
        pivot = left;
        i = left + 1;
        j = right;
        while (i < j)
        {
            while (a[i] < a[pivot])
                i++;
            while (a[j] > a[pivot])
                j--;
            if (i < j) {
                temp = a[i];
                a[i] = a[j];
                a[j] = temp;
            }
        }
    }
}
```

```

        temp = a[pivot];
        a[pivot] = a[j];
        a[j] = temp;
        QuickSort(a, left, j - 1);
        QuickSort(a, j + 1, right);
    }
}

```

Output

Enter the limit : 5

Enter the numbers : 30 40 50 20 10

The sorted elements are : 10 20 30 40 50

16.6 ANALYSIS OF QUICK SORT

| | | |
|-----------------------|---|---------------|
| Best-case analysis | : | $O(n \log n)$ |
| Average-case analysis | : | $O(n \log n)$ |
| Worst-case analysis | : | $O(n^2)$ |

16.7 ADVANTAGES OF QUICK SORT

Some of the key advantages of quick sorting technique are:

- It is one of the fastest sorting algorithms.
- Its implementation does not require any additional memory.
- It has better cache performance and high speed.

16.8 LIMITATIONS OF QUICK SORT

The disadvantages associated with quick sorting technique are as follows.

- The worst case efficiency of $O(n^2)$ is not well suited for large sized lists.
- Its algorithm is considered as a little more complex in comparison to some other sorting techniques.

REVIEW QUESTIONS

1. Give the working principle of quick sort.
2. What is the best case time complexity of the Quick sort algorithm? (or) Write the time complexities of quick sorting method. (or) Determine the average running time of quick sort. (or) What is the worst case complexity of quick sort?
3. What is the best way of choosing of choosing the pivot element in quick sort?

17.1 INTRODUCTION

The most common algorithm used in external sorting is the merge sort. This algorithm follows divide and conquer strategy.

- In dividing phase, the problem is divided into smaller problem and solved recursively.
- In conquering phase, the partitioned array is merged together recursively.

Merge sort is applied to the first half and second half of the array. This gives two sorted halves, which can then be recursively merged together using the merging algorithm.

The basic merging algorithm takes two input arrays A and B and an output array C. The first element of A array and B array are compared, then the smaller element is stored in the output array C and the corresponding pointer is incremented. When either input array is exhausted the remainder of the other array is copied to an output array C.

17.2 ALGORITHM

MERGE(ARR[], LEFT, CENTER, RIGHT)

A[] : Integer array
LEFT : Integer
CENTER : Integer
RIGHT : Integer

Step 1 : Start.
Step 2 : Set $N1 = CENTER - LEFT + 1$.
Step 3 : Set $N2 = RIGHT - CENTER$.
Step 4 : Repeat For $I = 0$ to $N1 - 1$.
Step 5 : Set $A[I] = ARR[LEFT + I]$.
Step 6 : Increment I by 1.
Step 7 : [End of Step 4 For loop].
Step 8 : Repeat For $J = 0$ to $N2 - 1$.
Step 9 : Set $B[J] = ARR[CENTER + 1 + J]$.
Step 10 : Increment J by 1.
Step 11 : [End of Step 8 For loop].
Step 12 : Repeat While $APTR < N1$ AND $BPTR < N2$.
Step 13 : If $A[APTR] \leq B[BPTR]$ then Goto Step 14 else Goto Step 18.
Step 14 : Set $ARR[CPTR] = A[APTR]$.
Step 15 : Increment $APTR$ by 1 and Goto Step 19.
Step 16 : Set $ARR[CPTR] = B[BPTR]$.
Step 17 : Increment $BPTR$ by 1.
Step 18 : Increment $CPTR$ by 1.
Step 19 : [End of Step 12 While loop].
Step 20 : Repeat While $APTR < N1$.
Step 21 : Set $ARR[CPTR] = A[APTR]$.
Step 22 : Increment $APTR$ by 1.
Step 23 : Increment $CPTR$ by 1.

| | | |
|---------|---|------------------------------|
| Step 24 | : | [End of Step 20 While loop]. |
| Step 25 | : | Repeat While BPTR < N2. |
| Step 26 | : | Set ARR[CPTR] = B[BPTR]. |
| Step 27 | : | Increment BPTR by 1. |
| Step 28 | : | Increment CPTR by 1. |
| Step 29 | : | [End of Step 25 While loop]. |
| Step 30 | : | Stop. |

17.3 MERGE SORT ROUTINE

```

void MergeSort(int arr[], int left, int right)
{
    int center;
    if (left < right)
    {
        center = (left + right) / 2;
        MergeSort(arr, left, center);
        MergeSort(arr, center + 1, right);
        Merge(arr, left, center, right);
    }
}

void Merge(int arr[], int left, int center, int right)
{
    int a[20], b[20], n1, n2, aptr, bptr, cptr, i, j;
    n1 = center - left + 1;
    n2 = right - center;
    for (i = 0; i < n1; i++)
        a[i] = arr[left + i];
    for (j = 0; j < n2; j++)
        b[j] = arr[center + 1 + j];
    aptr = 0;
    bptr = 0;
    cptr = left;
    while (aptr < n1 && bptr < n2)
    {
        if (a[aptr] <= b[bptr])
        {
            arr[cptr] = a[aptr];
            aptr++;
        }
        else
        {
            arr[cptr] = b[bptr];
            bptr++;
        }
        cptr++;
    }
}

```

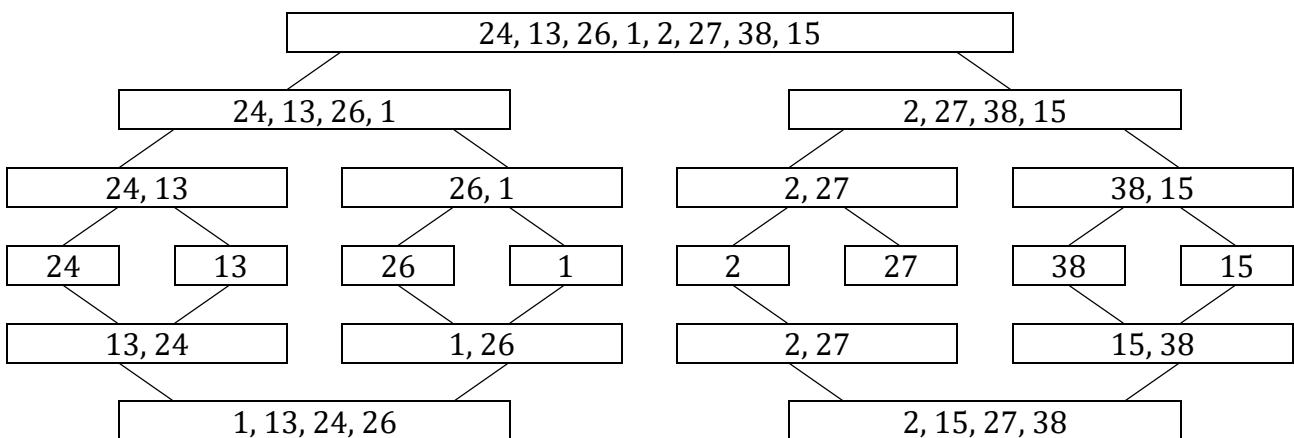
```

while (aptr < n1)
{
    arr[cptr] = a[aptr];
    aptr++;
    cptr++;
}
while (bptr < n2)
{
    arr[cptr] = b[bptr];
    bptr++;
    cptr++;
}
}

```

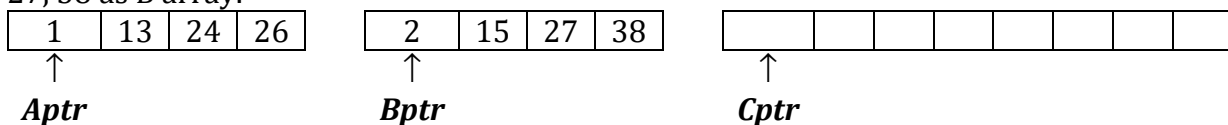
17.4 EXAMPLE

For instance, to sort the eight element array 24, 13, 26, 1, 2, 27, 38, 15, we recursively sort the first four and last four elements, obtaining 1, 13, 24, 26, 2, 15, 27, 38 then these array is divided into two halves and the merging algorithm is applied to get the final sorted array.

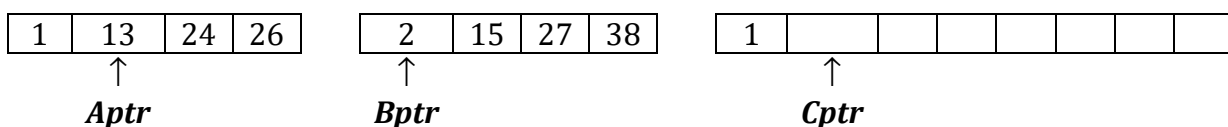


Now, the merging algorithm is applied as follows:

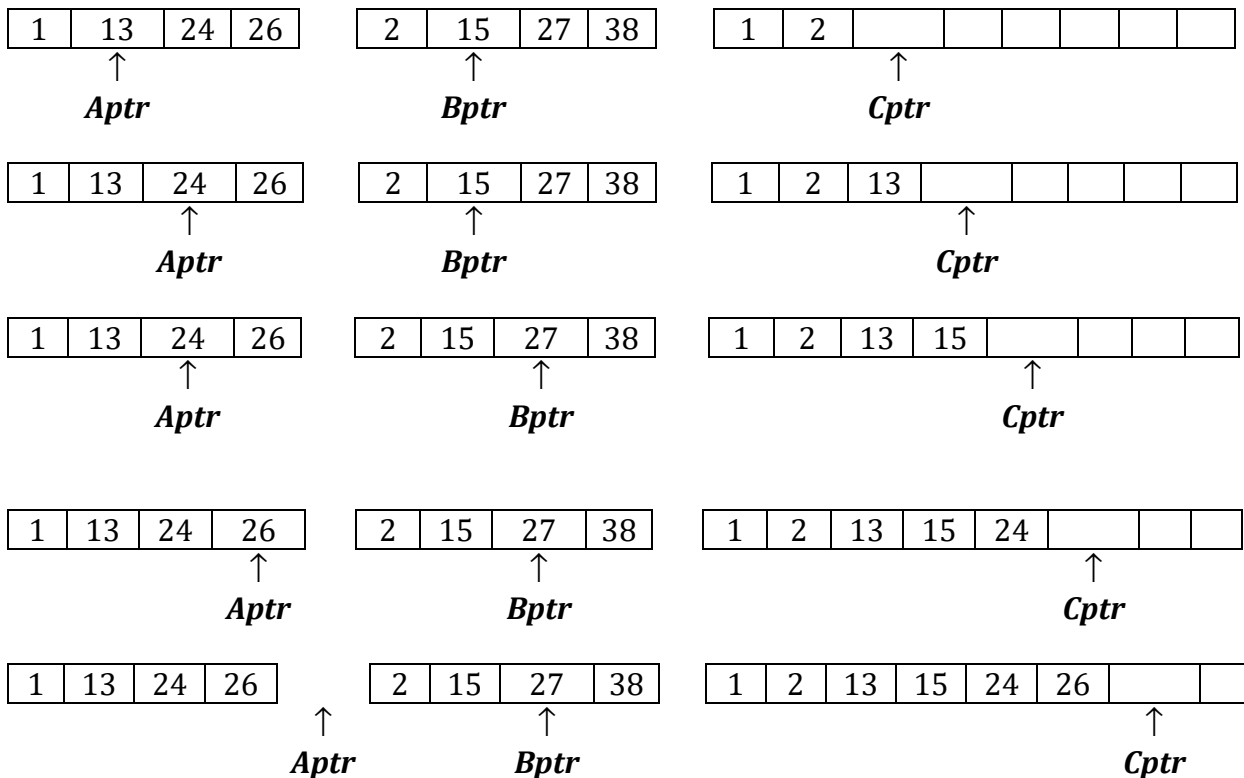
Let us consider first 4 elements 1, 13, 24, 26 as A array and the next four elements 2, 15, 27, 38 as B array.



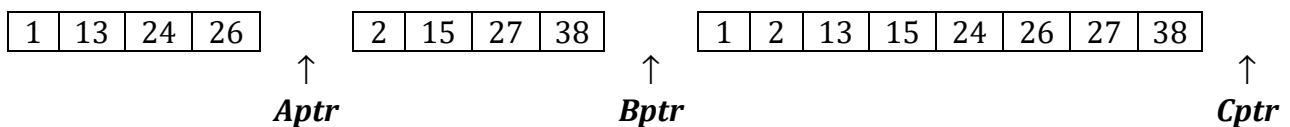
First, the element 1 from A array and element 2 from B array is compared, then the smallest element 1 from A array is copied to an output array C. Then the pointers Aptr and Cptr is incremented by one.



Next, 13 and 2 are compared and the smallest element 2 from B array is copied to C array and the pointers Bptr and Cptr gets incremented by one. This proceeds until A array and B array are exhausted, and all the elements are copied to an output array C.



Since A array is exhausted, the remaining elements of B array is then copied to C array.



Final sorted array:

| | | | | | | | |
|---|---|----|----|----|----|----|----|
| 1 | 2 | 13 | 15 | 24 | 26 | 27 | 38 |
|---|---|----|----|----|----|----|----|

17.5 PROGRAM

```
#include <stdio.h>
```

```
void MergeSort(int arr[], int left, int right);
```

```
void Merge(int arr[], int left, int center, int right);
```

```
int main()
```

```
{
```

```
    int i, n, arr[20];
```

```
    printf("Enter the limit : ");
```

```
    scanf("%d", &n);
```

```
    printf("Enter the elements : ");
```

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

```
        scanf("%d", &arr[i]);
```

```
    MergeSort(arr, 0, n - 1);
```

```
    printf("The sorted elements are : ");
```

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

```
        printf("%d\t", arr[i]);
```

```
    return 0;
```

```
}
```



```

void MergeSort(int arr[], int left, int right)
{
    int center;
    if (left < right)
    {
        center = (left + right) / 2;
        MergeSort(arr, left, center);
        MergeSort(arr, center + 1, right);
        Merge(arr, left, center, right);
    }
}

void Merge(int arr[], int left, int center, int right)
{
    int a[20], b[20], n1, n2, aptr, bptr, cptr, i, j;
    n1 = center - left + 1;
    n2 = right - center;
    for (i = 0; i < n1; i++)
        a[i] = arr[left + i];
    for (j = 0; j < n2; j++)
        b[j] = arr[center + 1 + j];
    aptr = 0;
    bptr = 0;
    cptr = left;
    while (aptr < n1 && bptr < n2)
    {
        if (a[aptr] <= b[bptr])
        {
            arr[cptr] = a[aptr];
            aptr++;
        }
        else
        {
            arr[cptr] = b[bptr];
            bptr++;
        }
        cptr++;
    }
    while (aptr < n1)
    {
        arr[cptr] = a[aptr];
        aptr++;
        cptr++;
    }
    while (bptr < n2)
    {
        arr[cptr] = b[bptr];
        bptr++;
        cptr++;
    }
}

```

Output

Enter the limit : 5

Enter the numbers : 30 40 50 20 10

The sorted elements are : 10 20 30 40 50

17.6 ANALYSIS OF MERGE SORT

| | | |
|-----------------------|---|---------------|
| Worst case analysis | : | $O(N \log N)$ |
| Best case analysis | : | $O(N \log N)$ |
| Average case analysis | : | $O(N \log N)$ |

17.7 ADVANTAGES

Some of the key advantages of merge sorting technique are:

- It is a fast and stable sorting method.
- It always ensures an efficiency of $O(n \log n)$.
- It has better cache performance.
- Merge Sort is a Stable Sort.

17.8 LIMITATIONS

The disadvantages associated with merge sorting technique are as follows

- It requires additional memory space to perform sorting. The size of the additional space is in direct proportion to the size of the input list.
- Even though the number of comparisons made by merge sort are nearly optimal, its performance is slightly lesser than that of quick sort.
- Merge sort sorts the larger amount of data making use of external storage device.
- It requires extra memory space.

REVIEW QUESTIONS

1. Give the working principle of merge sort.
2. What is the time complexity of merge sort algorithm?
3. State the algorithmic technique used in merge sort. (or) Which technique is suitable for merge sort? Why?
4. Merge sort is better than insertion sort. Why?
5. Sort 3, 1, 4, 1, 5, 9, 2, 6 using merge sort.

4.1 INTRODUCTION

The implementation of hash tables is frequently called hashing. Hashing is a technique used for performing insertions, deletions, and finds in constant average time.

4.1.1 Types of Hashing

There are two types of hashing namely:

- Static hashing
- Dynamic hashing

4.1.1.1 Static Hashing

In static hashing, the hash function maps search key values to a fixed set of locations.

4.1.1.2 Dynamic Hashing

In dynamic hashing, the hash table can grow to handle more items. The associated hash function must change as the table grows.

4.2 HASH TABLE

A hash table is a data structure, which is implemented by a hash function and used for searching elements in quick time. In a hash table, hash keys act as the addresses of the elements.

4.2.1 Applications of Hash Tables

The applications of hash table are:

- Compilers
- Graph theory problem
- Online spelling checkers etc.
- Database systems
- Symbol tables
- Data dictionaries
- Network processing algorithms
- Browse caches

REVIEW QUESTIONS

1. What is hashing? (or) Define hashing.
2. What is a hash table?
3. What are the applications of hash tables?

5.1 INTRODUCTION

Hashing finds the location of an element in a data structure without making any comparisons. In contrast to the other comparison-based searching techniques, like linear and binary search, hashing uses a mathematical function to determine the location of an element. This mathematical function called hash function accepts a value, known as key, as input and generates an output known as hash key. The hash function generates hash keys and stores elements corresponding to each hash key in the hash table. The keys that hash function accepts as input could be a digit associated with the input elements.

Let us consider a simple example of a file containing information for five employees of a company. Each record in that file contains the name and a three-digit numeric Employee ID of the employee. In this case, the hash function will implement a hash table of five slots using Employee IDs as the keys. That means, the hash function will take Employee IDs as input and generate the hash keys, as shown in Fig.

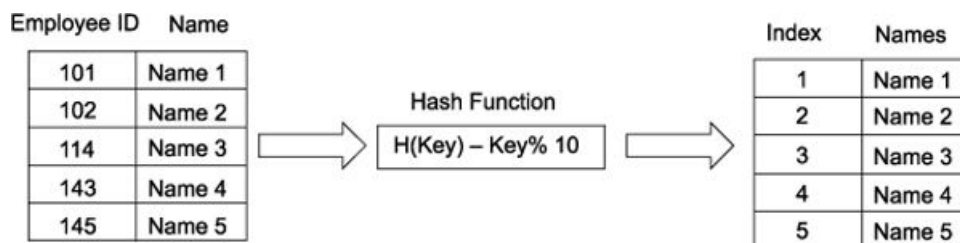


Fig. Generating hash keys

In the hash table generated in the above example, the hash function is $\text{Employee ID} \% 10$. Thus, for Employee ID 101, hash key will be calculated as 1. Therefore, Name1 will be stored at position 1 in the hash table. For Employee ID 102, hash key will be 2, hence Name2 will be stored at position 2 in the hash table. Similarly, Name3, Name4, and Name5 will be stored at position 4, 3, and 5 respectively, as shown in Fig. Later, whenever an employee record is searched using the Employee ID, the hash function will indicate the exact position of that record in the hash table.

A good hash functions should:

- Minimize collisions
- Be easy and quick to compute
- Distribute key values evenly in the hash table
- Use all the information provided in the key

5.3 METHODS OF HASHING FUNCTIONS

- Mid square method
- Modulo division or division remainder method
- Folding method
- Digit or character extraction method
- Radix transformation

5.3.1 Mid Square Method

In this method, the key is squared and the middle part of the result based on the number of digits required for addressing is taken as the hash value. This method works well if the keys do not contain a lot of leading and trailing zeros.

$H(X)$ = return middle digits of X^2

For example:

Map the key 2453 into a hash table of size 1000, there, $X = 2453$.

$$X^2 = 6017209$$

Extract middle value 172 as the hash value.

5.3.2 Modulo Division or Division Remainder Method

This method computes the hash value from the key using the modulo (%) operator. Here, the table size that is power of 2 like 32, 64 and 1024 should be avoided as it leads to more collisions. It is always better to select table size not close to the power of 2.

$H(\text{Key})$ = return $\text{Key} \% \text{TableSize}$

For example:

Map the key 4 into a hash table of size 4.

$$H(4) = 4 \% 4 = 0$$

5.3.3 Folding Method

This method involves splitting keys into two or more parts each of which has the same length as the required address with the possible exception of the last part and then adding the parts to form the hash address. There are two types of folding method. They are:

- Fold shifting method
- Fold boundary method

5.3.4 Fold Shifting Method

Key is broken into several parts of same length of the required address and then added to get the hash value. Final carry is ignored.

For example:

Map the key 123203241 to a range between 0 to 999.

Let $X = 1\ 2\ 3\ 2\ 0\ 3\ 2\ 4\ 1$

Position X into 123, 203, 241, then add these three values.

$123 + 203 + 241 = 567$ to get the hash value.

5.3.5 Fold Boundary Method

Key is broken into several parts and reverse the digits in the outermost partitions and then add the partition to form the hash value.

For example:

$X = 123203241$

Partition = 123, 203, 241

Reverse the boundary partition = 321, 203, 142

Add the partition = $321 + 203 + 142$

Hash value = 666.

5.3.6 Pseudo Random Number Generator Method

This method generates random number given a seed as parameter and the resulting random number then scaled into the possible address range using modulo division. It must ensure that it always generates the same random value for a given key. The random number produced can be transformed to produce a valid hash value.

$$X_{i+1} = A X_i \% \text{TableSize}$$

5.3.7 Digit or Character Extraction Method

This method extracts the selected digits from the key and used it as the address or it can be reversed to give the hash value.

For example:

Map the key 123203241 to a range between 0 to 9999.

Select the digits from the positions: 2, 4, 5, and 8.

Therefore the hash value = 2204.

Similarly, the hash value can also be obtained by considering the above digit positions and reversing it, which yields the hash value as 4022.

5.3.8 Radix Transformation Method

In this method a key is transformed into another number base to obtain the hash value.

For example:

Map the key $(8465)_{10}$ in the range 0 to 999 using base 15.

$$(8465)_{10} = (2795)_{15}$$

The key 8465 is placed in the hash address 2795.

REVIEW QUESTIONS

1. What is a hash function?
2. List out the different methods of hashing function.
3. What is a mid square method?
4. What is a modulo division or division remainder method?
5. What is a folding method?
6. What is a fold shifting method?
7. What is a fold boundary method?
8. What is a pseudo random number generator method?
9. What is a digit or character extraction method?
10. What is a radix transformation method?

6.1 INTRODUCTION

The hash function takes some key values as input, performs some mathematical calculation, and generates hash key to ascertain the position in the hash table where the record corresponding to the key will be stored. However, it is quite possible that the hash function generates same hash keys for two different key values. That means, two different records are indicated to be stored at the same position in the hash table. This situation is termed as collision.

As a result, a hash function must be designed in such a way that the possibility of a collision is negligible. Various techniques such as, linear probing, chaining without replacement, and chaining with replacement are used to evade the chances of a collision.

6.2 VARIOUS TECHNIQUES OF HASHING

The various techniques of hashing are:

- Separate chaining
- Open addressing
 - Linear probing
 - Quadratic probing
 - Double hashing
- Rehashing
- Extendible hashing

REVIEW QUESTIONS

1. What is a collision? How it can be prevented?
2. List out the various techniques of hashing.

7.1 INTRODUCTION

The separate chaining technique uses linked lists to overcome the problem of collisions. In this technique, all the keys that resolve to the same hash values are maintained in a linked list.

Whenever a collision occurs, the key value is added at the end of the corresponding list. Thus, each position in the hash table acts as a header node for a linked list. To perform a search operation, the list indicated by the hash function is searched sequentially.

Insert 0, 1, 81, 4, 64, 25, 16, 36, 9, 49.

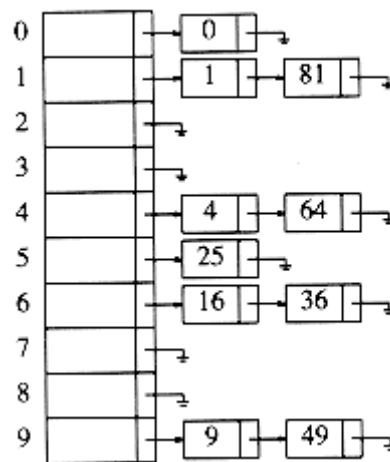


Fig. A separate chaining hash table

7.2 INSERTION

To perform an insert, we traverse down the appropriate list to check whether the element is already in place (if duplicates are expected, an extra field is usually kept, and this field would be incremented in the event of a match). If the element turns out to be new, it is inserted either at the front of the list or at the end of the list, whichever is easiest.

$$H(\text{Key}) = \text{Key} \% \text{TableSize}$$

$$\text{Insert 0} \quad H(0) = 0 \% 10 = 0$$

$$\text{Insert 1} \quad H(1) = 1 \% 10 = 1$$

$$\text{Insert 81} \quad H(81) = 81 \% 10 = 1$$

The element 81 collides to the same hash value 1. To place the value at this position, perform the following:

- Traverse the list to check whether it is already present.
- Since it is not already present, insert at the end of the list.

Similarly, the rest of the elements are inserted.

Insert 4 $H(4) = 4 \% 10 = 4$

Insert 64 $H(64) = 64 \% 10 = 4$

Insert 25 $H(25) = 25 \% 10 = 5$

Insert 16 $H(16) = 16 \% 10 = 6$

Insert 36 $H(36) = 36 \% 10 = 6$

Insert 9 $H(9) = 9 \% 10 = 9$

Insert 49 $H(49) = 49 \% 10 = 9$

ROUTINE TO PERFORM INSERTION

```
void Insert(int Key, HashTable H)
{
    Position Pos, NewCell;
    List L;
    Pos = Find(Key, H);
    if(Pos == NULL)
    {
        NewCell = malloc(sizeof(struct ListNode));
        if(NewCell == NULL)
            printf("Out of space!!!");
        else
        {
            L = H->TheLists[Hash(Key, H->TableSize)];
            NewCell->Next = L->Next;
            NewCell->Element = Key;
            L->Next = NewCell;
        }
    }
}
```

7.3 FIND

To perform a find, we use the hash function to determine which list to traverse. We then traverse this list in the normal manner, returning the position where the item is found.

7.4 ADVANTAGE

- More number of elements can be inserted as it uses array of linked lists.
- Collision resolution is simple and efficient.

7.5 DISADVANTAGE

- It requires pointers, which occupies more memory space.
- It takes more effort to perform a search, since it takes time to evaluate the hash function and also to traverse the list.

REVIEW QUESTIONS

1. Define separate chaining.

8.1 INTRODUCTION

Separate chaining hashing has the disadvantage of requiring pointers. This tends to slow the algorithm down a bit because of the time required to allocate new cells, and also essentially requires the implementation of a second data structure.

Open addressing hashing is an alternative to resolving collisions with linked lists. In an open addressing hashing system, if a collision occurs, alternate cells are tried until an empty cell is found.

More formally, cells $h_0(X)$, $h_1(X)$, $h_2(X)$, ... are tried in succession where

$$h_i(X) = (\text{Hash}(X) + F(i)) \bmod \text{TableSize}$$

with $F(0) = 0$.

8.2 OPEN ADDRESSING TECHNIQUES FOR COLLISION RESOLUTION

Three common collision resolution strategies are:

- Linear Probing
- Quadratic Probing
- Double Hashing

8.3 LINEAR PROBING

In linear probing, F is a linear function of i , typically $F(i) = i$. This amounts to trying cells sequentially (with wraparound) in search of an empty cell.

$$h_i(X) = (\text{Hash}(X) + i) \% N$$

$$h_0(X) = \text{Hash}(X) \% N$$

$$h_1(X) = (\text{Hash}(X) + 1) \% N$$

$$h_2(X) = (\text{Hash}(X) + 2) \% N$$

Figure shows the result of inserting keys {89, 18, 49, 58, 69} into a hash table using the same hash function as before and the collision resolution strategy, $F(i) = i$.

The first collision occurs when 49 is inserted; it is put in the next available spot, namely spot 0, which is open.

58 collides with 18, 89, and then 49 before an empty cell is found three away.

The collision for 69 is handled in a similar manner.

As long as the table is big enough, a free cell can always be found, but the time to do so can get quite large. Worse, even if the table is relatively empty, blocks of occupied cells start forming. This effect, known as primary clustering, means that any key that hashes into the cluster will require several attempts to resolve the collision, and then it will add to the cluster.

| | Empty Table | After 89 | After 18 | After 49 | After 58 | After 69 |
|---|-------------|----------|----------|----------|----------|----------|
| 0 | | | | 49 | 49 | 49 |
| 1 | | | | | 58 | 58 |
| 2 | | | | | | 69 |
| 3 | | | | | | |
| 4 | | | | | | |
| 5 | | | | | | |
| 6 | | | | | | |
| 7 | | | | | | |
| 8 | | | 18 | 18 | 18 | 18 |
| 9 | | 89 | 89 | 89 | 89 | 89 |

Figure: Open addressing hash table with linear probing, after each insertion

Advantage

- It doesn't require pointers.

Disadvantage

- It forms clusters, which degrades the performance of the hash table for storing and retrieving.

8.4 QUADRATIC PROBING

Quadratic probing is a collision resolution method that eliminates the primary clustering problem of linear probing. Quadratic probing is what you would expect-the collision function is quadratic. The popular choice is $F(i) = i^2$.

$$h_i(K) = (\text{Hash}(K) + i^2) \% N$$

$$h_0(X) = \text{Hash}(X) \% N$$

$$h_1(X) = (\text{Hash}(X) + 1) \% N$$

$$h_2(X) = (\text{Hash}(X) + 4) \% N$$

Figure shows the resulting open addressing hash table with this collision function on the same input used in the linear probing example.

When 49 collides with 89, the next position attempted is one cell away. This cell is empty, so 49 is placed there.

Next 58 collides at position 8. Then the cell one away is tried but another collision occurs. A vacant cell is found at the next cell tried, which is $2^2 = 4$ away. 58 is thus placed in cell 2.

The same thing happens for 69.

| | Empty Table | After 89 | After 18 | After 49 | After 58 | After 69 |
|---|-------------|----------|----------|----------|----------|----------|
| 0 | | | | 49 | 49 | 49 |
| 1 | | | | | | |
| 2 | | | | | 58 | 58 |
| 3 | | | | | | 69 |
| 4 | | | | | | |
| 5 | | | | | | |
| 6 | | | | | | |
| 7 | | | | | | |
| 8 | | | 18 | 18 | 18 | 18 |
| 9 | | 89 | 89 | 89 | 89 | 89 |

Figure: Open addressing hash table with quadratic probing, after each insertion

ROUTINE TO PERFORM INSERTION

```

void Insert(int Key, HashTable H)
{
    Position Pos;
    Pos = Find(Key, H);
    if(H->TheCells[Pos].Info != Legitimate )
    {
        H->TheCells[Pos].Info = Legitimate;
        H->TheCells[Pos].Element = Key;
    }
}

```

ROUTINE TO PERFORM FIND

```

Position Find(int Key, HashTable H)
{
    Position CurrentPos;
    int CollisionNum;
    CollisionNum = 0;
    CurrentPos = Hash(Key, H->TableSize);
    while((H->TheCells[CurrentPos].Info != Empty) &&
        (H->TheCells[CurrentPos].Element != Key))
    {
        CurrentPos += 2 * ++CollisionNum - 1;
        if(CurrentPos >= H->TableSize)
            CurrentPos -= H->TableSize;
    }
    return CurrentPos;
}

```

Limitations

Although quadratic probing eliminates primary clustering, elements that hash to the same position will probe the same alternate cells. This is known as secondary clustering. Secondary clustering is a slight theoretical blemish. Simulation results suggest that it generally causes less than an extra half probe per search.

8.5 DOUBLE HASHING

Double hashing uses the idea of applying a second hash function to the key when a collision occurs. The result of the second hash function will be the number of positions from the point of collision to insert. It must never evaluate to zero.

$$h_i(X) = (\text{Hash}(X) + i * \text{Hash}_2(X)) \text{ Mod TableSize}$$

A popular second hash function is:

$$\text{Hash}_2(\text{Key}) = R - (\text{Key} \% R)$$

where R is a prime number that is smaller than the size of the table.

If we choose R = 7, then Figure shows the results of inserting the same keys as before.

| | Empty Table | After 89 | After 18 | After 49 | After 58 | After 69 |
|---|-------------|----------|----------|----------|----------|----------|
| 0 | | | | | | 69 |
| 1 | | | | | | |
| 2 | | | | | | |
| 3 | | | | | 58 | 58 |
| 4 | | | | | | |
| 5 | | | | | | |
| 6 | | | | 49 | 49 | 49 |
| 7 | | | | | | |
| 8 | | | 18 | 18 | 18 | 18 |
| 9 | | 89 | 89 | 89 | 89 | 89 |

Figure: Open addressing hash table with double hashing, after each insertion

The first collision occurs when 49 is inserted. $h_2(49) = 7 - 0 = 7$, so 49 is inserted in position 6.

$$\text{Hash}(49) = 49 \% 10 = 9$$

$$\text{Hash}_2(49) = 7 - (49 \% 7) = 7 - 0 = 7$$

$$h_1(49) = (9 + 1 * 7) \% 10 = (9 + 7) \% 10 = 16 \% 10 = 6.$$

$h_2(58) = 7 - 2 = 5$, so 58 is inserted at location 3.

$$\text{Hash}(58) = 58 \% 10 = 8$$

$$\text{Hash}_2(58) = 7 - (58 \% 7) = 7 - 2 = 5$$

$$h_1(58) = (8 + 1 * 5) \% 10 = (8 + 5) \% 10 = 13 \% 10 = 3.$$

Finally, 69 collides and is inserted at a distance $h_2(69) = 7 - 6 = 1$ away.

$$\text{Hash}(69) = 69 \% 10 = 9$$

$$\text{Hash}_2(69) = 7 - (69 \% 7) = 7 - 6 = 1$$

$$h_1(69) = (9 + 1 * 1) \% 10 = (9 + 1) \% 10 = 10 \% 10 = 0.$$

If we tried to insert 60 in position 0, we would have a collision. Since $h_2(60) = 7 - 4 = 3$, we would then try positions 3, 6, 9, and then 2 until an empty spot is found. It is generally possible to find some bad case, but there are not too many here.

REVIEW QUESTIONS

1. Define open addressing hashing.
2. List out the open addressing techniques for collision resolution.
3. What is meant by primary clustering?
4. What is meant by secondary clustering?
5. What is linear probing?
6. What is quadratic probing?
7. What is double hashing?

9.1 INTRODUCTION

If the table gets too full, the running time for the operations will start taking too long and inserts might fail for closed hashing with quadratic resolution. This can happen if there are too many deletions intermixed with insertions. A solution, then, is to build another table that is about twice as big (with associated new hash function) and scan down the entire original hash table, computing the new hash value for each (non-deleted) element and inserting it in the new table. This entire operation is called rehashing.

Rehashing can be implemented in several ways with quadratic probing.

- Rehash as soon as the table is half full.
- Rehash only when an insertion fails.
- Rehash when the table reaches a certain load factor.

As an example, suppose the elements 13, 15, 24, and 6 are inserted into a closed hash table of size 7. The hash function is $h(X) = X \bmod 7$. Suppose linear probing is used to resolve collisions. The resulting hash table appears in Figure.

| | |
|---|----|
| 0 | 6 |
| 1 | 15 |
| 2 | |
| 3 | 24 |
| 4 | |
| 5 | |
| 6 | 13 |

Figure: Open addressing hash table with linear probing with input 13,15, 6, 24

If 23 is inserted into the table, the resulting table in Figure will be over 70 percent full.

| | |
|---|----|
| 0 | 6 |
| 1 | 15 |
| 2 | 23 |
| 3 | 24 |
| 4 | |
| 5 | |
| 6 | 13 |

Figure: Open addressing hash table with linear probing after 23 is inserted

Because the table is so full, a new table is created. The size of this table is 17, because this is the first prime which is twice as large as the old table size. The new hash function is then $h(X) = X \bmod 17$. The old table is scanned, and elements 6, 15, 23, 24, and 13 are inserted into the new table. The resulting table appears in Figure.

| | |
|----|----|
| 0 | |
| 1 | |
| 2 | |
| 3 | |
| 4 | |
| 5 | |
| 6 | 6 |
| 7 | 23 |
| 8 | 24 |
| 9 | |
| 10 | |
| 11 | |
| 12 | |
| 13 | 13 |
| 14 | |
| 15 | 15 |
| 16 | |

Figure: Open addressing hash table after rehashing

This entire operation is called rehashing. This is obviously a very expensive operation; the running time is $O(N)$, since there are N elements to rehash and the table size is roughly $2N$, but it is actually not all that bad, because it happens very infrequently.

ROUTINE TO PERFORM REHASHING

```
HashTable Rehash(HashTable H)
{
    int i, OldSize;
    Cell *OldCells;
    OldCells = H->TheCells;
    OldSize = H->TableSize;
    H = InitializeTable(2 * OldSize);
    for(i = 0; i < OldSize; i++)
        if(OldCells[i].Info == Legitimate)
            Insert(OldCells[i].Element, H);
    free(OldCells);
    return H;
}
```

REVIEW QUESTIONS

1. What is rehashing?
2. When the rehashing be implemented?

10.1 INTRODUCTION

If either open hashing or closed hashing is used, the major problem is that collisions could cause several blocks to be examined during a find, even for a well-distributed hash table. Furthermore, when the table gets too full, an extremely expensive rehashing step must be performed, which requires $O(n)$ disk accesses.

A clever alternative, known as extendible hashing, allows a find to be performed in two disk accesses. Insertions also require few disk accesses.

Let us suppose, for the moment, that our data consists of several six-bit integers. Figure shows an extendible hashing scheme for this data. The root of the "tree" contains four pointers determined by the leading two bits of the data. Each leaf has up to $m = 4$ elements. It happens that in each leaf the first two bits are identical; this is indicated by the number in parentheses. To be more formal, D will represent the number of bits used by the root, which is sometimes known as the directory. The number of entries in the directory is thus 2^D . d_L is the number of leading bits that all the elements of some leaf L have in common. d_L will depend on the particular leaf, and $d_L \leq D$.

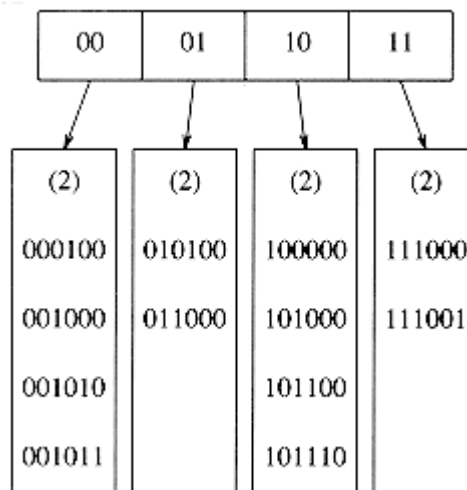


Figure: Extendible hashing: original data

Suppose that we want to insert the key 100100. This would go into the third leaf, but as the third leaf is already full, there is no room. We thus split this leaf into two leaves, which are now determined by the first three bits. This requires increasing the directory size to 3. These changes are reflected in Figure.

Notice that all of the leaves not involved in the split are now pointed to by two adjacent directory entries. Thus, although an entire directory is rewritten, none of the other leaves are actually accessed.

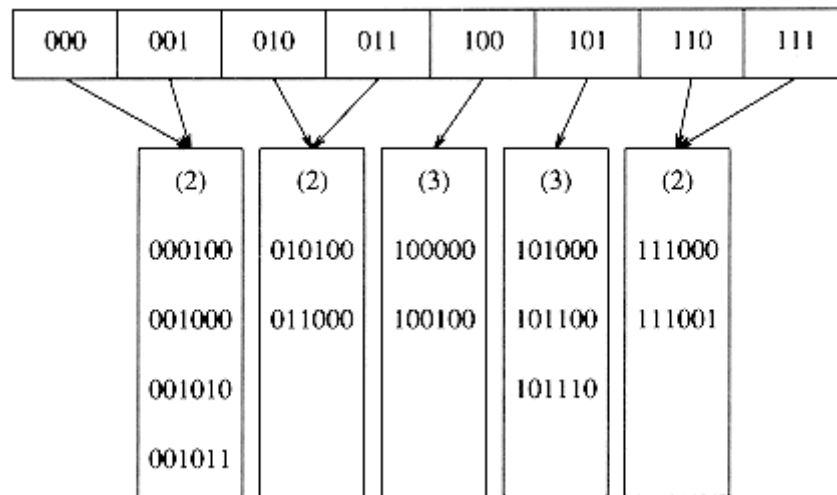


Figure: Extendible hashing: after insertion of 100100 and directory split

If the key 000000 is now inserted, then the first leaf is split, generating two leaves with $d_L = 3$. Since $D = 3$, the only change required in the directory is the updating of the 000 and 001 pointers. See Figure.

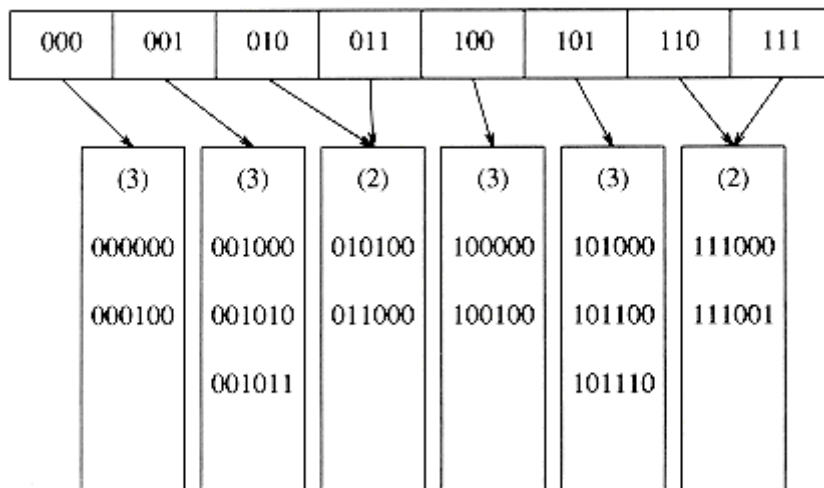


Figure: Extendible hashing: after insertion of 000000 and leaf split

This very simple strategy provides quick access times for Insert and Find operations on large databases.

REVIEW QUESTIONS

1. What is an extendible hashing? (or) Define extendible hashing.
2. Compare the various hashing techniques.



RAJALAKSHMI

ENGINEERING COLLEGE

Website : www.rajalakshmi.org

Elearning : www.rajalakshmicolleges.net/moodle

***Give a man a fish and you feed him for a day.
Teach him how to fish and you feed him for a lifetime.***