Unit 5
Sorting, Searching and Search Trees

By : Nootan Padia

- **Milestone :**
  - **Sorting Techniques**
    - **Bubble sort**
    - **Selection sort**
    - **Insertion sort**
    - **Quick sort**
    - **Shell sort**
    - **Merge sort**
    - **Heap sort**
    - **Radix sort**

- **Milestone :**
  - **Searching Techniques**
    - **Linear Search**
    - **Binary Search (Simple and With Recursion)**
  - **Search Trees**
    - **Height – Balanced Trees**
    - **2 – 3 Trees**
    - **Weight – Balanced Trees**
    - **Trie Structure**
  - **Hash – Table Methods**
  - **Collision – Resolution Techniques**

Searching :

- **Searching means to find an element from a given sorted or unsorted list.**
- **There are various searching techniques :**
  - **Linear Search**
  - **Binary Search**
  - **Tree Searching Techniques**

Linear Search (Sequential Search)

- **A Linear Search(sequential search) of a list/array begins at the beginning of the list/array and continues until the item is found or the entire list/array has been searched.**
- **Linear search has complexity O(n) .**
- **Example : (Refer your class book.)**

Elements : 12,23,10,34,85 Key1 : 10 Key2 : 75

Function : LINEAR_SEARCH(K,N,X). Given an unordered vector K consisting of N+1 elements, this algorithm searches the vector for a particular element having the value X. (Assume Vector has value up to K[N]).

Step – 1 : [Initialize]

I ← 1

K[N+1] ← X

Step – 2 : [Search the Vector]

Repeat while K[I] =/= X

I = I + 1

Step – 3 : [Check Search is Successful or Not]

If I = N + 1

then Write('UNSUCCESSFUL SEARCH')

Return(0)

else Write('SUCCESSFUL SEARCH')

Return(1)

- The **average** length of search for **n** elements or records is given by :

  $E(n) = 1 * P_1 + 2 * P_2 + 3 * P_3 + ... + n * P_n$

  $= (1 + 2 + 3 + ... + n) P_n$

  Where $P = 1 = P1 + P2 + P3 + ... + P_n$

  If $P_a = P/n$ average probability then and putting this value of into the equation we get the final result.

  $= (1+2+3+...+n) P_n$

  $= P_a * [n(n+1)/2]$

  $=1/n * [n(n+1)/2]$

  $=(n+1)/2$, i.e., $O(n)$

Binary Search

- A **binary search** looks for an item in a list using a divide-and-conquer strategy.
- Binary search algorithm assumes that the items in the array being searched are **sorted.**
- The algorithm **begins at the middle** of the array in a binary search.
- Binary search has complexity $O(\log_2 n)$.
- Example : (Refer your class book)

Elements : 12,23,10,34,85 Key1 : 10 Key2 : 75

Function : BINARY_SEARCH(K,N,X). Given a vector K, consisting N elements in ascending order, this algorithm searches the structure for a given by X. The variable LOW, MIDDLE, and HIGH denote the lower, middle and upper limits of the search interval, respectively

Step – 1 : [Initialize]

LOW ← 1

HIGH ← N

Step – 2 : [Perform Search]

Repeat thru step 4 while LOW <= HIGH

Step – 3 : [Obtain index of midpoint of interval]

MIDDLE ← [(LOW + HIGH)/2]

Step – 4 : [Compare]

If X < K[MIDDLE]

then

HIGH ← MIDDLE – 1

else if X > K[MIDDLE]

then

LOW ← MIDDLE + 1

else

Write (‘SUCCESSFUL SEARCH’)

Return(MIDDLE)

Step – 5 : [Unsuccessful search]

Write (‘UNSUCCESSFUL SEARCH’)

Return(0)

Function : BINARY SEARCH R(P,Q,K,X). Given the bounds of a sub table P and Q and a vector K, this algorithm recursively searches for the given element whose value is stored in X. MIDDLE and LOC are integer variables

Step – 1 : [Search vector K between P and Q for value X]

If P > Q

then LOC ← 0

else

MIDDLE ← (P+Q)/2

If X < K[MIDDLE]

then LOC ← BINARY_SEARCH (P, MIDDLE – 1, K, X)

```
                else        if  X > K[MIDDLE]
                            then        LOC ←
        BINARY_SEARCH(MIDDLE + 1, Q, K, X)
                            else        LOC ← MIDDLE
```

Step – 2 : [Finished]

```
                Return(LOC)
```

Binary Search	Linear Search
Binary search requires the input data to be sorted.	Linear search doesn't require the sorted data.
Binary search requires an *ordering* comparison.	Linear search requires an *equality* comparison.
Binary search has complexity O(log n).	Linear search has complexity O(n).
Binary search requires random access to the data.	Binary search requires sequential access to the data.
Best case of binary search is when key value is middle value of list.	Best case of linear search is when key value is first value of list.

Sorting

- Sorting is the operation of arranging the records into some sequential order according to an ordering criterion.
- Sorting may be in ascending or descending order.
- There are many sorting techniques. Some of them are as follows :
  - Bubble sort
  - Selection sort
  - Insertion sort
  - Quick sort
  - Shell sort
  - Merge sort
  - Heap sort
  - Radix sort

Selection Sort

- One of the easiest ways to sort a table is by *selection.*
- Beginning with the first record in the table, a search is performed to locate the element which has the smallest key.
- When this element is found, it is interchanged with the first record in the table.
- This interchange places the record with the smallest key in the first position of the table.
- A search for the second smallest key is then carried out.
- This is accomplished by examining the keys of the records from the second element onward.
- The element which has the second smallest key is interchanged with the element located in the second position of the table.

The process of searching for the record with the next smallest key and placing it in its proper position continues until all records have been sorted in ascending order.

Example : (Refer class book)

Arrange following elements in ascending order using selection sort.

42, 23, 74, 11, 65, 58, 94, 36, 99, 87

Procedure : SELECTION_SORT(K,N). Given a vector K of N elements, this procedure rearranges the vector in ascending order. The variable PASS denotes the pass index and the position of the first element in the vector which is to be examined during a particular pass. The variable MIN_INDEX denotes the position of the smallest element encountered thus for in a particular pass. The variable I is used to index elements K[PASS] to K[N] to a given pass.

Step – 1 : [Loop on pass index]

Repeat thru step 4 for PASS = 1, 2, …, N – 1

Step – 2 : [Initialize minimum index]

MIN_INDEX ← PASS

Step – 3 : [Make a pass and obtain element with smallest value]

Repeat for I=PASS + 1, Pass + 2,…, N

If K[I] < K[MIN_INDEX]

then MIN_INDEX ← I

Step – 4 : [Exchange elements]

If MIN_INDEX =/= PASS

then K[PASS] ↔ K[MIN_INDEX]

Step – 5 : [Finished]

Return

Bubble Sort

- It differs form the selection sort in that, instead of finding the smallest record and then performing an interchange, two records are interchanged immediately upon discovering that they are out of order.
- It is very simple technique. However, this is not efficient in comparison to other sorting techniques. But for smaller lists this sorting technique works fine.
- Example : (Refer class book)

Arrange following elements in ascending order using selection sort.

42, 23, 74, 11, 65, 58, 94, 36, 99, 87

Procedure : BUBBLE_SORT(K,N). Given a vector K and N elements, this procedure sorts the elements into ascending order. The variable PASS and LAST denote the pass counter and position of the last unsorted element, respectively. The variable I is used to index the vector elements. The variable EXCHS is used to count the no. of exchanges made on any pass.

Step – 1 : [Initialize]

LAST ← N

Step – 2 : [Loop on pass index]

Repeat thru step 5 for PASS = 1,2,..,N – 1

Step – 3 : [Initialize exchanges counter for this pass]

EXCHS ← 0

Step – 4 : [Perform pair wise comparisons on unsorted elements]

Repeat for I=1,2,…,LAST – 1

If K[I] > K[I+1]

then K[I] ↔ K[I+1]

EXCHS ← EXCHS + 1

Step – 5 : [Were any exchanges made on this pass?]

If EXCHS = 0

then Return

else LAST ← LAST – 1

Step – 6 : [Finished]

Return

Insertion Sort

- The insertion sort works just like its name suggests – it inserts each item into its proper place in the final list.
- The simple implementation of this sorting technique requires two list structures – the source list and the list into which sorted items are inserted.
- To save memory, most implementations use an in – place sort that works by moving the current item past the already sorted items and repeatedly swapping it with the preceding item until it is in place.
- E. g. : 20,10,30,5

INSERTION_SORT(LIST , N)

Where LIST → represents the list of elements

N → represents no. of elements in the list.

Step 1 :[Generating Sorted List]

Repeat for I =1,2,3,… N

(i) TEMP ← LIST[I]

(ii) P ← I – 1

while(TEMP < LIST[P] && P > 0)

LIST[P + 1] ← LIST[P]

P ← P – 1

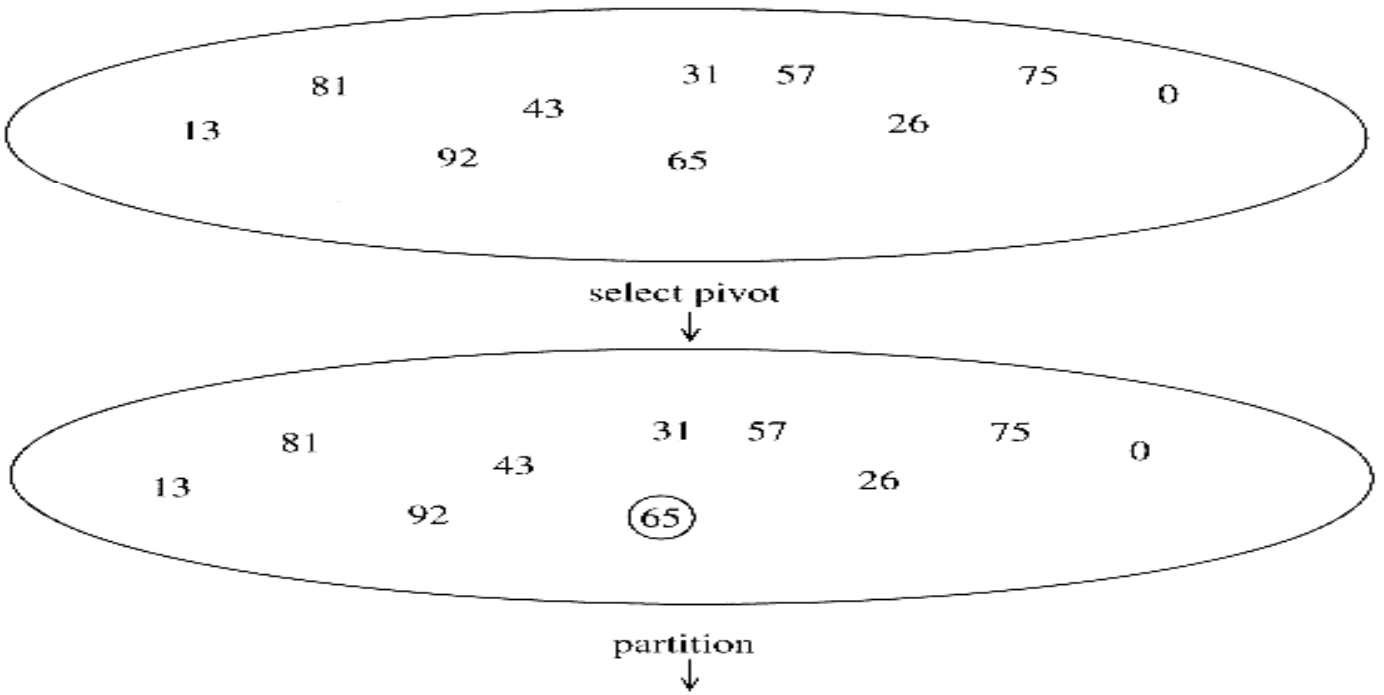
LIST[P + 1] ← TEMP

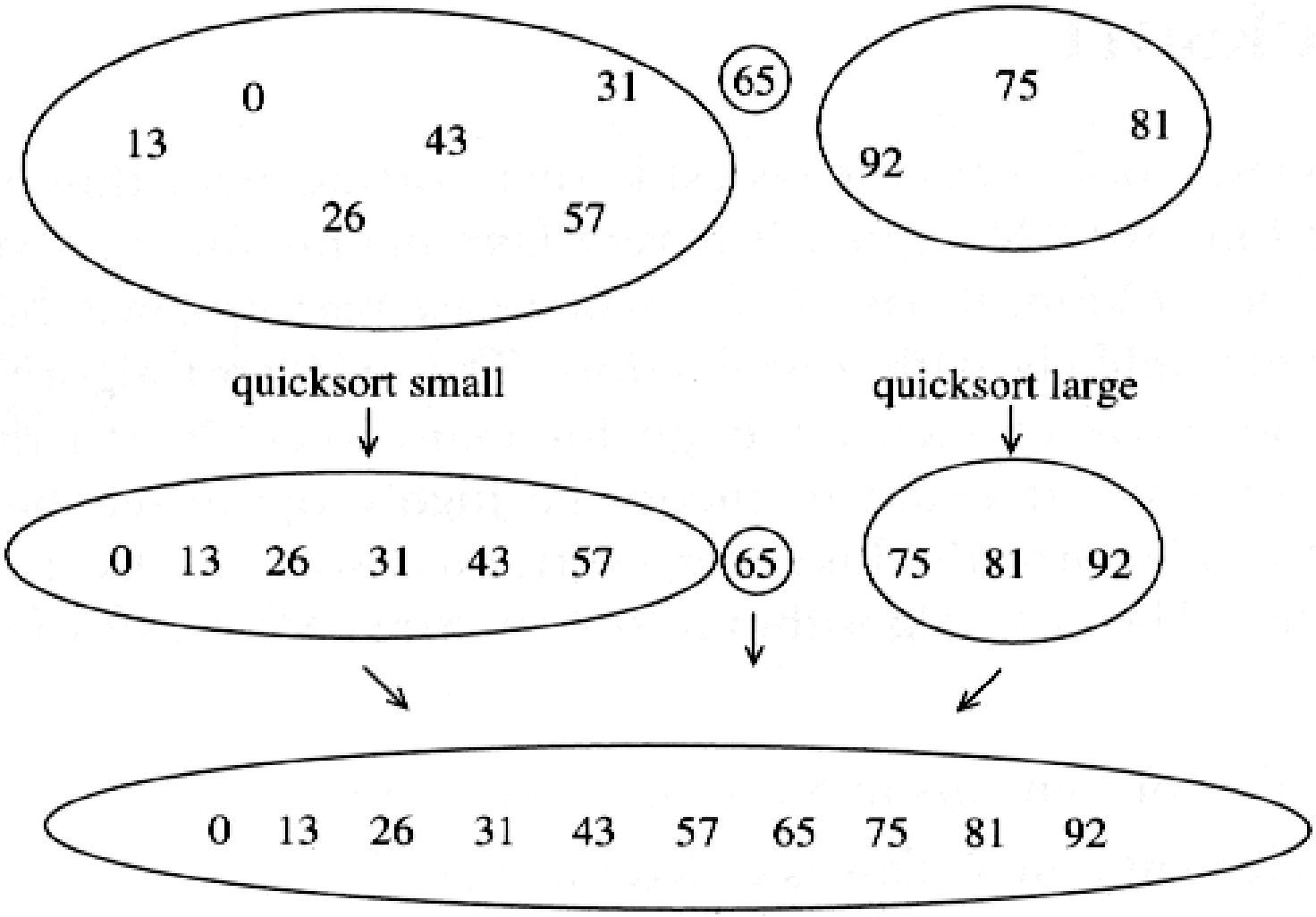
Step 2 : [Finished]

exit

Quick Sort

- Fastest known sorting algorithm in practice
- Average case: O(N log N)
- Worst case: $O(N^2)$ But, the worst case seldom happens.
- Divide-and-conquer recursive algorithm.
-




0
31
65
13
43
75
81
92
26
57
quicksort small
quicksort large
0 13 26 31 43 57
65
75 81 92
0 13 26 31 43 57 65 75 81 92

Procedure : QUICK_SORT (K, LB, UB). Given a table K of N records, this recursive procedure sorts the table in ascending order. A dummy record with K[N+1] is assumed where K[I] <= K[N+1] for all 1 <= I <= N. The integer parameters LB and UB denote the lower and upper bounds of the current subtable being processed. The indices I and J are used to select certain keys during the processing of each subtable. KEY contains the key value which is being placed in its final position within the sorted subtable. FLAG is a logical variable which indicates the end of the process that places a record in its final position. When FLAG becomes false, the input subtable has been partitioned into two disjointed parts.

```
Step – 1 : [Initializing]
                    FLAG ← true
Step – 2 : [Perform sort]
                    If LB < UB
                    Then    I ← LB
                            J ← UB + 1
                            KEY ← K[LB]
                            Repeat while FLAG
                                    I ← I + 1
                                    Repeat while K[I] < KEY
                                            I ← I + 1
                                    J ← J – 1
                                    Repeat while K[J] > KEY
                                            J ← J - 1
```

```
            If  I < J
            then     K[I] ←→ K[J]
            else     FLAG ← false
        K[LB] ←→ K[J]
        // SORT FIRST SUBTABLE
        Call QUICK_SORT(K, LB, J – 1)
        //SORT SECOND SUBTABLE
        Call QUICK_SORT(K, J + 1, UB)
```

Step – 3 : [Finished]

Return

Shell Sort

- The shell sort named after its developer Donald Shell.
- The sort technique works by comparing the list elements that are separated by a specific distance, i.e., a gap until the elements compared with the current gap are in order.
- Two then divides gap and the process continues.
- When the gap is reached to 0 and no changes are possible, the list is sorted.

SHELL_SORT(LIST , N)

Where LIST → represents the list of elements

N → represents the size of the list

step 1 : [initialize]

GAP ← N/2

step 2 : [performing sorting]

Do

Do

SWAP ← 0

Repeat for I=1,2,3,….,I<=(N - GAP)

if LIST[I] > LIST[I + GAP]

LIST[I] ↔ LIST[I+GAP]

SWAP = 1

Repeat While(SWAP)

Repeat While (GAP ← GAP/2)

step 3: [Finished]

Exit

Merge Sort

- **The operation of sorting is closely related to the process of merging.**
- **In the early days of data processing, merging was performed on cards with the aid of a machine called a *collator*.**
- **The collator had as input two decks of cards, each of which was sorted, and it proceeded to merge these two decks and to output a single sorted deck of cards.**
- **We will formulate a sorting algorithm based on successive merges.**

- There are two formulations of a merge sort.
  - The first approach, which is recursive, is simpler to write and analyze.
  - The second approach is iterative and more complex.
- By comparing both approaches it is obvious that the recursive formulation is superior to its iterative counterparts.
- Example : Refer class book.

Procedure : TWO_WAY_MERGE_SORT_R(K,START,FINISH). Given a vector K it is required to sort recursively it's elements between positions START and FINISH(inclusive). SIZE denotes the number of elements in the current subtable to be sorted. MIDDLE denotes the position of MIDDLE element of that subtable.

STEP 1 : [Compute the SIZE of the current subtable]

SIZE ← FINISH - START+1

STEP 2 : [Test base condition for subtable of size one]

If SIZE<=1

then Return

STEP 3: [Calculate midpoint position of current subtable]

MIDDLE ← START + [SIZE/2] - 1

STEP 4 : [Recursively sort first subtable]

Call TWO_WAY_MERGE_SORT_R(K,START,MIDDLE)

STEP 5 : [Recursively sort second subtable]

Call TWO_WAY_MERGE_SORT_R(K,MIDDLE+1,FINISH)

STEP 6 : [Merge two ordered subtable]

Call SIMPLE_MERGE (K, START, MIDDLE+1, FINISH)

STEP 7 : [Finished]

Return

SIMPLE_MERGE(K, FIRST, SECOND, THIRD). Given two ordered subtables stored in vector K with FIRST, SECOND and THIRD as just described, this procedure performs a simple merge. TEMP is a temporary vector used in the merging process. The variables I and J denote the cursor associated with the first and second subtables, respectively L is an index variable associated with the vector TEMP.

STEP 1 : [Initialize]

I ← FIRST

J ← SECOND

L ← 0

STEP – 2 : [Compare corresponding elements and output the smallest]

Repeat while I<SECOND and J<=THIRD

IF K [I] <=K [J]

THEN

L ← L+1

TEMP [L] ← K [I]

I ← I+1

ELSE

L ← L+1

TEMP [L] ← K [J]

J ← J+1

STEP – 3 : [Copy the remaining unprocessed elements in output area]

IF I>=SECOND

THEN

Repeat while J<=THIRD

L ← L+1

TEMP [L] ← K [J]

J ← J+1

ELSE

Repeat while I<SECOND

L ← L+1

TEMP [L] ← K [I]

I ← I+1

STEP – 4 : [Copy elements in temporary vector into original area]

Repeat for I=1, 2…..L

K [FIRST-1+I] ← TEMP [I]

STEP – 5 : [Finished]

Return

$$\text{Log}_2\ N = \text{Log}(N) / \text{Log}(2)$$

$\log_2(1) = 0$

$\log_2(2) = 1$

$\log_2(3) = 1.584962500721156$

$\log_2(4) = 2$

$\log_2(5) = 2.321928094887362$

$\log_2(6) = 2.584962500721156$

$\log_2(7) = 2.807354922057604$

$\log_2(8) = 3$

$\log_2(9) = 3.169925001442312$

$\log_2(10) = 3.321928094887362$

Algorithm TWO_WAY_MERGE_SORT. Given a vector R containing N records, this algorithm sorts the vector into ascending order by successfully invoking Procedure MERGE_PASS. An auxiliary vector C which has the same size as R is needed. L is a variable which specifies the number of elements in each subtable to be merged during a particular pass.

STEP – 1 : [Perform sort]

Repeat for L= 1, 2, 4,…, $2^{[\log_2 N]-1}$

If $\log_2 L$ is even

Then Call MERGE_PASS(R, N, C, L)

Else Call MERGE_PASS(C, N, R, L)

STEP – 2 : [Recopy if required]

If [$\log_2 N$] is odd

Then Repeat for I = 1, 2, …, N

R [I] $\leftarrow$ C [I]

STEP – 3 : [Finished]

Exit

Algorithm. MERGE_PASS(R,N,C,L). Given a table R of N records which is considered to be partitioned into ordered subtables, each of which contains L records (or fewer), this algorithm merges these subtables by pairs. The records of R are denoted by R[1], R[2], …, R[N]. An auxiliary table C is required in the merging process. The variables P and Q keep track of which pair of subtables are currently being merged. Since N is not necessarily an integral power of 2, the sizes of the subtables being merged are not always the same (that is, L). The sizes of the tables are given by variables N1 and N2. The variables I and J are indices to the corresponding records of the subtables that are to be merged.

STEP – 1 : [Initialize first pass]

$$P \leftarrow 1$$

$$N1 \leftarrow N2 \leftarrow L$$

$$Q \leftarrow P + L$$

STEP – 2 : [Perform one pass]

Repeat thru step 7 while Q <= N

STEP – 3 : [Initialize simple merge]

I ← P

J ← Q

S ← P

STEP – 4 : [Compare corresponding records and output smallest]

Repeat while I – P < N1 AND J – Q < N2

IF R[I] <= R[J]

THEN C[S] ← R[I]

I ← I + 1

S ← S +1

ELSE C[S] ← R[J]

J ← J + 1

S ← S +1

STEP – 5 : [Copy the remaining unprocessed records from a subtable into output area]

IF I – P >= N1

THEN Repeat For T = J, J+1, … , Q + N2 – 1

C[S] ← R[T]

S ← S + 1

ELSE Repeat For T = I , I+1, … , P + N1 – 1

C[S] ← R[T]

S ← S + 1

STEP – 6 : [Update and test direct subtable index]

P ← Q + N2

IF P > N

THEN Return

STEP – 7 : [Update Q and check bound of second subtable]

Q ← P + L

IF Q+L > N+1

THEN N2 ← N – Q + 1

STEP – 8 : [Copy unmatched subtable]

Repeat for T = P, P+1, … , N

C[T] ← R[T]

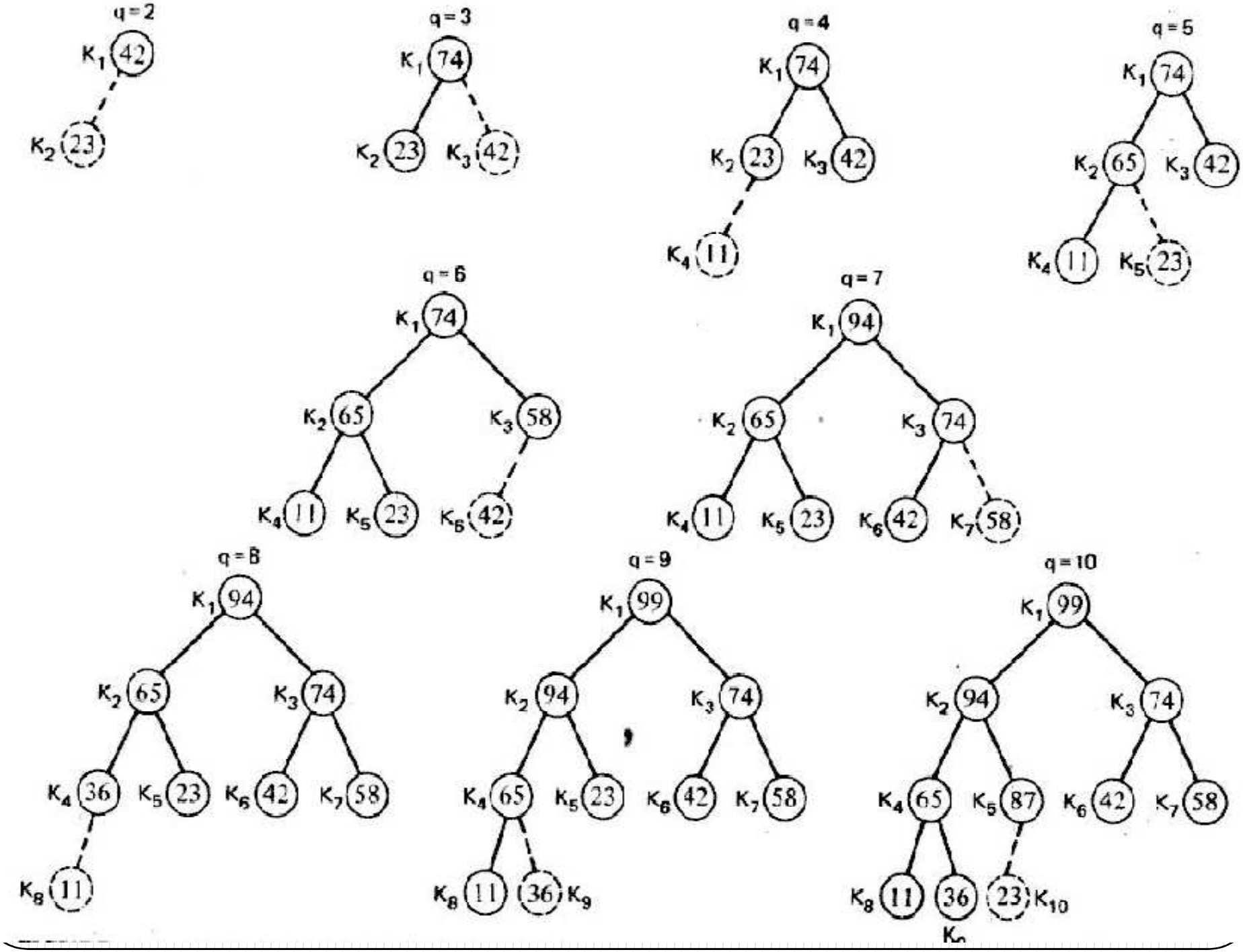
STEP – 9 : [Finished]

Return

Heap Sort

- A heap is a complete binary tree with the property that the value of each node is at least as large as its children (if they exist).
- This implies that the root of the heap has the largest element in the tree.
- Example of creating heap :

- For sorting, we are about to study two parts :
  - In the first part, we will form a heap, thus getting the largest element to the root.
  - In the second part, the output sequence is generated in the decreasing order by successively outputting the root and reconstructing the remaining tree into a heap.


q = 2
K1 42
K2 23
q = 3
K1 74
K2 23
K3 42
q = 4
K1 74
K2 23
K3 42
K4 11
q = 5
K1 74
K2 65
K3 42
K4 11
K5 23
q = 6
K1 74
K2 65
K3 58
K4 11
K5 23
K6 42
q = 7
K1 94
K2 65
K3 74
K4 11
K5 23
K6 42
K7 58
q = 8
K1 94
K2 65
K3 74
K4 36
K5 23
K6 42
K7 58
K8 11
q = 9
K1 99
K2 94
K3 74
K4 65
K5 23
K6 42
K7 58
K8 11
36
K9
q = 10
K1 99
K2 94
K3 74
K4 65
K5 87
K6 42
K7 58
K8 11
36
23
K10

- **Now Heap Sort tracing - Refer class book**

Procedure HEAP_CREATE (K, N) given a vector k (type integer) containing the keys of the N Record of table, this algorithm creates a heap as previously described .the index variable Q control the number of insertion which is to be performed. The integer variable J denotes the index of the parents of the K [I]. KEY (type integer) contains the key of record being inserted into an existing heap.

1. [Build heap]
   Repeat thru step 7 for Q =2, 3 ….N
2. [Initialize Construction phase]
   I←Q
   KEY←K [Q]
3. [obtain parent of new record]
   J←TRUNC (I/2)

4. [Place new record in exiting heap]

Repeat thru step 6 while I>1 and KEY>K[J]

5. [Interchange record]

K [I]←K[J]

6. [Obtain next parent]

I←J

J←TRUNC (I/2)

IF J<1

THEN J←1

7. [Copy new record into its proper place]

K [I]←KEY

8. [Finished]

Return

Procedure HEAP_SORT (K, N) Given a vector k(type integer)containing the keys of the N records of table and the procedure CREATE _HEAP which has been previously described ,this procedures sorts the table into ascending order. the variable Q represents the pass index .Index variable I and J are Used, Where the latter is the index of the son of the former. KEY is an integer variable which contains KEY of Records being swapped at each pass.

1. [Create the initial heap]
   Call CREATE_HEAP (K, N)
2. [Perform sort]
   Repeat thru step 10 for Q=N, N-1…2
3. [Exchange record]
   K [1]←→K[Q]

4. [Initialize pass]

I←1

KEY ←k[1]

J←2

5. [Obtain index of largest son of new record]

If J+1<Q

Then if K [J+1]>K [J]

Then J←J+1

6.[reconstruct the new record]

Repeat thru step 10 while<=Q-1 and K[J]>KEY

7. [interchange record]

K [I] <-K [J]

8. [obtain next left son]

I←J

J←2*I

9. [obtain index of next largest son]

If J+1 <Q

Then if K [J+1]>K [J]

Then J←J+1

Else if J>N

Then J←N

10. [copy record into is proper place]

K [I]←KEY

11. [finished]

Return

Radix Sort

- **The radix sort is a method of sorting which predates any digital computer.**
- **This was performed and is still performed on a mechanical card sorter.**
- **For a key of m digits, it requires m * n key accesses.**

Example : 12, 01,55,78,98,45,31

Step – 1 : Take last digit and store the particular value in particular packet.

0	1	2	3	4	5	6	7	8	9
	01	12			55			78	
	31				45			98	

Ans : 01,31,12,55,45,78,98

Step – 2 : Take first digit and store the particular value in particular packet.

0	1	2	3	4	5	6	7	8	9
01	12		31	45	55		78		98

Ans : 01,12,31,45,55,78,98

Algorithm : RADIX_SORT(FIRST, N). Gives a table of N racords arranged as a linked list, where each noad in the list consists of a key field(K) and a pointer field(LINK), this procedure performs a radix sort as previously described. The address of the first record in the linked table is given by the pointer variable FIRST. The vectors T and B are used to store the address or the rear and front records in each queue(pocket). In particular, the records T[i] and B[i] point to the top and bottom records in the ith pocket, respectively. The variable J is the pass index. The pointer variable R denotes the address of the current record being examined in the table and being directed to the appropriate pocket. NEXT is a pointer variable which is used during the combining of the pockets at the end of each pass. The integer variable D denotes the currunt digit in the key being examined.

Step – 1 : [Perform sort]

Repeat thru step 4 – for j = 1,2,..... , M

Step – 2 : [Initialize the pass]

Repeat for i=0,1,....9

T[i] ← B[i] ← NULL

R ← FIRST

Step – 3 : [Distribute each record in the appropriate pocket]

Repeat while R = NULL

D ← bj (obtain Jth digit of the key K(R))

```
            NEXT ← LINK(R)
            If T[D] = NULL
            then   T[D] ← B[D] ← R
            else   LINK(T[D]) ← R
                   T[D] ← R
            LINK(R) ← NULL
            R ← NEXT
4.  [Combine pockets]
        P ← 0
        Repeat while B[P] = NULL
            P ← P + 1
        FIRST ← B[P]
        Repeat for I = P + 1, P + 2, ..., 9
            PREV ← T[I - 1]
            If T[I] ≠ NULL
            then   LINK(PREV) ← B[I]
            else   T[I] ← PREV
5.  [Finished]
        Return
```

Comparison of all Sorting Techniques

Sorting Method	BUBBLE SORT
Introduction	Repeatedly stepping through the list to be sorted, comparing each pair of adjacent items and swapping them if they are in the wrong order.
Best case	$O(n^2)$
Worst case	$O(n^2)$
Memory requirement	No extra space needed.
Pros	1. A simple and easy method. 2. Efficient for small lists.
Cons	Highly inefficient for large data.

Sorting Method	SELECTION SORT
Introduction	Finds the minimum value in the list and then swaps it with the value in the first position, repeats these steps for the remainder of the list.
Best case	$O(n^2)$
Worst case	$O(n^2)$
Memory requirement	No extra space needed.
Pros	1. Recommended for small files. 2. Good for partially sorted data.
Cons	Inefficient for large lists.

Sorting Method	INSERTION SORT
Introduction	Every repetition of insertion sort removes an element from the input data, inserts it into the in the correct position in the already sorted list until no input elements remain.
Best case	O(n)
Worst case	$O(n^2)$
Memory requirement	No extra space needed.
Pros	1. Relatively simple and easy to implement. 2. Good for almost sorted data.
Cons	Inefficient for large lists.

Sorting Method	QUICK SORT
Introduction	Picks an element, called a pivot, from the list. Reorders the list so that all elements with values less than the pivot come before the pivot, whereas all elements with values greater than the pivot come after it. After this partitioning, the pivot is in its final position. This is called the partition operation. Recursively sorts the sub – list of the lesser elements and the sub – list of the greater elements.
Best case	$\mathbf{O(n\log_2 n)}$
Worst case	$O(n^2)$
Memory requirement	No extra space needed.
Pros	1. Extremely fast 2. Inherently recursive
Cons	Very complex algorithm

Sorting Method	SHELL SORT
Introduction	Shell sort is a multi-pass algorithm. Each pass is an insertion sort of the sequences consisting of every *h*-th element for a fixed gap *h* (also known as the increment). This is referred to as *h*-sorting.
Best case	$O(n^{1.5})$
Worst case	$O(n\log^2 n)$
Memory requirement	No extra space needed.
Pros	1. It is faster than a quick sort for small arrays. 2. Its speed and simplicity makes it a good choice in practice.
Cons	Slower for sufficiently big arrays.

Sorting Method	RADIX SORT
Introduction	Numbers are placed at proper locations by processing individual digits and by comparing individual digits that share the same significant position.
Best case	O(n)
Worst case	O(n)
Memory requirement	Extra space proportional to n is needed.
Pros	1. It is very simple and fast. 2. In – place, recursive, and one of the fastest sorting algorithms for numbers or strings of letters.
Cons	Radix sort can also take more space than other sorting algorithms since in addition to the array that will be sorted, there needs to be a sub – list for each of the possible digits or letters.

Sorting Method	MERGE SORT
Introduction	If the list is of length 0 or 1, then it is already sorted. Otherwise, the algorithm divides the unsorted list into two sub – lists of about half the size. Then, it sorts each sub – list recursively by reapplying the merge sort and then merges the two sub – lists back into one sorted list.
Best case	$O(n\log_2 n)$
Worst case	$O(n\log_2 n)$
Memory requirement	Extra space proportional to n is needed.
Pros	1. Good for external file sorting. 2. Can be applied to files of any size.
Cons	1. It requires twice the memory of the heap sort because of the second array used to store the sorted list. 2. It is recursive, which can make it a bad choice for applications that run on machines with limited memory.

Sorting Method	HEAP SORT
Introduction	It begins by building a heap out of the data set, and then removing a heap out of the data set, and then removing the largest item and placing it at the end of the partially sorted array. After removing the largest item, it reconstructs the heap, removes the largest remaining item, and places it in the next open position from the end of the partially sorted array. This is repeated until there are no items left in the heap and he sorted array is full.
Best case	$O(n\log_2 n)$
Worst case	$O(n\log_2 n)$
Memory requirement	No extra space needed.
Pros	1. It does not use recursion and that heap sort works just as fast for any data order. That is, there is basically no worst case scenario. 2. Heaps work well for small tables and the tables where changes are infrequent.
Cons	Do not work well for most large tables.

Search Trees (All in Tree and graph ppt)

Hash Table Methods (page no. 607)

- **Division Method**
- **Mid square method**
- **Folding Method**
- **Digit Analysis Method**
- **Length – Dependent Method**
- **Algebraic Coding**
- **Multiplicative Hashing**

Collision – Resolution Techniques (Short description)

- **Problem: many-to-one mapping**

  a potentially huge set of strings

  a small set of integers

- **Collision: Having a second key into a previously used slot.**

- **Collision resolution: deals with keys that are mapped to same slots.**

Detail : Collision – Resolution Techniques

- **A hashing function can map several keys into the same address. When this situation arises, the colliding records must be sorted and accessed as determined by a collision – resolution technique.**
- **There are two broad classes of such techniques :**
  - **Open addressing**
  - **Chaining**
- **The general objective of a collision – resolution technique is to attempt to place colliding records elsewhere in the table.**
- **This requires the investigation of a series of table positions until an empty one is found to accommodate a colliding records.**

❑ **With open addressing, if a record with key x is mapped to an address location d and this location is already occupied, then other locations in the table are examined until a free location is found for the new record.**

If a record with key K, is deleted, then K, is set to a special value called DELETE, which is not equal to the value of any key. The sequence in which the locations of a table are examined can be formulated in many ways.

One of the simplest techniques for resolving collisions is to use the following sequence of locations for a table of m entries.

d, d+1,….,m-1,m,1,2,….d-1

An unoccupied record location is always found if at least one is available; otherwise, the search halts unsuccessfully after scanning m locations. When retrieving a particular record, the same sequence of locations is examined until that record is found or until an unoccupied record position is encountered. In this latter case the desired record is not in the table, so the search fails. This collision – resolution technique is called linear probing.

OPENLP(X, INFO, INSERT). The record to be inserted or located is identified by the key argument X, any other information to be inserted is denoted by INFO, and the type of information to be performed is specified by the logical parameter INSERT. This function performs the table lookup or insertion operations. The function returns the position of the record in question, if successful. Otherwise, an error condition is signaled by a negated position. The hashing function HASH is used to calculate the initial position. If a record location never contained a record, then the corresponding key field has a special value called EMPTY.

STEP – 1 : [Calculate initial position]

d ← HASH(X)

STEP – 2 : [Perform indicated operation if location is found]

Repeat for i = d, d+1, …. , m-1, m, 1, 2, … , d-1

If X = K_i

Then If not INSERT

Then Return(i)

else Return(-i)

If K_i = EMPTY or K_i = DELETE

Then If INSERT

Then $K_i \leftarrow X$

$DATA_i \leftarrow$ INFO

Return(i)

else If K_i = EMPTY

Then

Return(-i)

STEP – 3 : [Table Overflow]

Write('OVERFLOW OR LOOK-UP ERROR')

Return(0)

-> PROBABILITY = 1/m

-> The ratio α = n / m is known as the load factor. Here, n = no. of records and m = size of the table.

-> Average Length = E[ALOS] =

(refer class book)

Table : E[ALOS] for Linear Probing

Load Factor α	No. of probes	
	Successful	Unsuccessful
.10	1.056	1.118
.20	1.125	1.281
.30	1.214	1.520
.40	1.333	1.889
.50	1.500	2.500
.60	1.750	3.625
.70	2.167	6.060
.80	3.000	13.000
.90	5.500	50.500
.95	10.500	200.500

O Drawbacks :-

1. Deletions from the table are difficult to perform. The strategy used here was to have a special table entry with a value of DELETE to denote the deletion of that entry. Such an approach enables the table to be searched in a proper way.

2. Another drawback of the linear probe method is caused by clustering effects, whose severity increases as the table becomes full. When the first insertion is made, the probability of a new element being inserted in a particular position is clearly 1/11. For the second insertion, however, the probability that position 2 will become occupied is twice as likely as any remaining available position: namely, the entry will be placed in position 2 if the key is mapped into either 1 or 2.

Continuing in this manner, on the fifth insertion the probability that the new entry will be place in position 5 is five times as likely as its being placed in any remaining unoccupied position. Thus, the trend is for long sequences of occupied positions to become longer. Such a phenomenon is called primary clustering.

- The detrimental effect of primary clustering can be reduced by selecting a different probing technique. Such a technique exists and is called random probing.

- This method generates a random sequence of positions rather than an ordered sequence as was the case in the linear probing method. The random sequence generated in this fashion must contain every position between 1 and m exactly once. A table is full when the first duplicate position is generated. (E.g. Refer class book)
- Although random probing has alleviated the problem of primary clustering, we have not eliminated all types of clustering. In particular, clustering occurs when two keys are hashed into the same value. In such an instance, the same sequence of positions is generated for both keys by the random probe method. The clustering phenomenon is called secondary clustering.

Function: OPENRP(X,INFO). Given a record identified by a key X, this function inserts corresponding INFO into a table represented by the structure R. The hashing function HASH is used to calculate an initial address. C and M are integers relatively prime to each other use for random probing. DELETE and EMPTY serve the same purpose as they did in Algorithm OPENLP.

STEP – 1 :[Calculate the initial position]

D ← HASH(X)

STEP – 2 :[Perform first probe]

If K[D]=EMPTY or K[D]=DELETE

Then K[D] ← X

DATA[D] ← INFO

Return (D)

STEP – 3 : [Initiate further search]

Y←D – 1

Y←(Y+C)mod M

J←Y+1

If J=D

Then Write('OVERFLOW')

Return(0)

STEP – 4 : [Find first open place]

Repeat step 5 while K[J] != EMPTY OR K[J]!=DELETE

STEP – 5 : [Scan next entry]

Y←(Y+C)mod M

J←Y+1

If J=D

Then Write('OVERFLOW')

Return(0)

STEP – 6 : [Finished]

K[J]←X

DATA[J]←INFO

Return(J)

Average Length of search for double hashing technique = E[ALOS] = (Refer class book)

Table : E[ALOS] for random probing with double hashing

Load Factor α	No. of probes	
	Successful	Unsuccessful
.10	1.054	1.111
.20	1.116	1.250
.30	1.189	1.429
.40	1.277	1.667
.50	1.386	2.000
.60	1.527	2.500
.70	1.720	3.333
.80	2.012	5.000
.90	2.558	10.000
.95	3.153	20.000

- One approach to alleviating this secondary clustering problem is to have a second hashing function, independent of the first, select the parameter c in the random probing method.
- For example, let us assume that H_1 is the first hashing function, with $H_1(x_1) = H_1(x_2) = i$ for $x_1 \neq x_2$ where i is the hash value. Now, if we have a second hashing function, H_2, such that $H_2(x_1) \neq H_2(x_2)$ when $x_1 \neq x_2$ and $H_1(x_1) = H_1(x_2)$, we can use $H_2(x_1)$ or $H_2(x_2)$ as the value of parameter c in the random probe method. The two random sequences generated by this scheme are different when H2 and H1 are independent.

- **The effects of secondary clustering can, therefore, be curtailed. This variation of open addressing is called double hashing (of rehashing).**
- **E.g. (refer class book)**

- In summary, there are three main difficulties with the open addressing method of collision resolution.
  - First, lists of colliding records for different hash values become intermixed. This phenomenon requires, on the average, more probes.
  - Second, we are unable to handle a table overflow situation in a satisfactory manner. On detecting an overflow, the entire table must be reorganized. The overflow problem cannot be ignored in many table applications where table space requirements can vary drastically.
  - Finally, the physical deletion of records is difficult.

- One of the most popular methods of handling overflow records is called separate chaining.
  - In this method, the colliding records are chained into a special overflow area which is distinct from the prime area.
  - This area contains that part of the table into which records are initially hashed.
  - A separate linked list is maintained for each set of colliding records.
  - Therefore, a pointer field is required for each record in the primary and overflow areas.
  - E.G. (Refer class book)

Average Length of search for separate chaining = E[ALOS] = (Refer class book)

Table : E[ALOS] with separate chaining

Load Factor α	No. of probes	
	Successful	Unsuccessful
.10	1.050	1.005
.20	1.100	1.019
.30	1.150	1.041
.40	1.200	1.070
.50	1.250	1.107
.60	1.300	1.149
.70	1.350	1.197
.80	1.400	1.249
.90	1.450	1.307
.95	1.475	1.337

- **Ex. 1 : perform linear probing on the following data : 12, 01, 04, 03, 07, 08, 10, 02, 05, 14. (hint : 1. (Hash(x) + i) mod MAX 2. use hashing function key % 10)**
- **Ex. 2 : consider the keys 7, 22, 17, 32, 5 and 24. let Max = 7. let us use quadratic probing. (hint : (Hash(x) + or - i^2) mod MAX**
- **Ex. 3 : let the hash function be key % 10, max = 10, and the keys be 12, 01, 18, 56, 79, 49. perform double hashing.**

Garbage Collection

- There are several methods of allocating and freeing storage.
- Several algorithms for freeing allocated storage that is no longer being used are introduced.
- Deciding when to allocate storage is simple.
- It is done when the programmer requests it by declaring a structure at program – block entry or by invoking a routine which creates a specific structure at run time.
- To processes the temporaries that were discovered at compile time, or when, during program execution, certain data – dependent temporaries are formed.
- But to free storage is not easy matter.
- Obviously, at block exit, the storage allocated at entry for local variables can all be freed.

- The difficult storage to handle is that storage which is dynamically allocated, such as list structure nodes of programmer – created blocks or strings.
- One method is to make the responsibility for freeing storage to the programmers.
- But this, if applied universally, places too much of a burden on the programmer – it is too easy to forget about some temporaries, structures, etc.
- Some languages do provide a dispose statement to allow programmers some responsibility for freeing at least some instances of their defined data structures.
- Most languages however, reserve for themselves the task of storage release, even if they do provide a dispose command by which the programmer can release certain blocks of storage.
- Therefore, the problem now becomes one of determining by what means the system decides to free storage.

- There are several methods.
  - One is to free no storage at all until there is almost none left. Then, all the allocated blocks are checked, and those that are no longer being used are freed. This method is called garbage collection. During the program execution, blocks of storage that once were needed but which at some later time became unnecessary and unused are called “garbage”. A garbage collection simply goes through and recovers these garbage blocks.
  - Another method is to free each block as soon as it becomes unused. This prevents the accumulation of garbage blocks but requires more run – time checking during processing. This method is generally implemented by means of reference counters – counters which record how many pointers to this block are still in existence in the program.

- Two problems arise in the context of storage release.
  - One is the accumulation of garbage as mentioned before; this has the effect of decreasing the amount of free storage available and consequently increasing the chances of having to refuse a request for storage.
  - The other problem is that of “dangling references”. A dangling reference is a pointer existing in a program which still accesses a block that has been freed. If ever the block is reallocated and then this dangling pointer is used, the program once again has access to that block which is now being used for completely different purposes. Results can be catastrophic.

- It is generally conceded that the dangling reference is potentially the more dangerous of the two problems, taken and so more effort is taken to minimize its likelihood of occurring. This, by the way, is another reason why not many systems allow the programmer much freedom to deallocate his own storage.

```
new(p);
-
-
-
q := p;
dispose(p);
```

- In the reference – counter method, a counter is kept that records how many different program elements have direct access to each block. When the block is first allocated, its reference counter is set to 1. Each time another link is made pointing to this block, the reference counter is incremented; each time a link to it is broken, the reference counter is decremented this point it is returned to the free list. Notice that this technique completely eliminates the dangling reference problem. The block is returned after there are no references to it in the program.

- There are certain drawbacks in using this method, however.
  - If the blocks that are allocated form a circular structure, then their reference counts will always remain set to at least 1 and none of the blocks will ever be freed, even if all pointers from outside the circular structure to blocks in the circular list are destroyed.
  - Another drawback to the reference – counter method is the overhead involved in maintaining the reference counts. This is a more serious objection because it can increase the execution time of the program significantly.

- The other method of determining when to free storage is garbage collection. This method makes use of a special routine which is invoked whenever the available storage is almost exhausted, or whenever a particular request cannot be met, or perhaps, whenever the amount of available storage has decreased beyond a certain predefined point. Normal program execution is interrupted while this routine frees garbage blocks and is resumed when the garbage collector has finished its work. The garbage collection algorithm generally has two phases.
  - The first phase consists of a tracing of all the access paths from all the program and system variables through the allocated blocks. Each block accessed in this way is marked.
  - Phase two consists of moving through the entire segment of returning to the free list every allocated block that has not been marked.

TRIE STRUCTURE

- A trie structure is a complete m-ary tree in which each node consists of m components.
- Typically, these components correspond to letters and digits.
- Trie structures occur frequently in the area of information organization and retrieval.
- In particular, branching at each node of level k depends on the kth character of a key.

Node Number

	1	2	3	4	5	6	7	8	9	10	11	12
b	-	-	-	-	-	-	-	-	-	-	-	GO
A	ALLOCATE	-	CALL	-	-	-	-	-	-	-	-	-
B	2	-	-	-	-	-	-	-	-	-	-	-
C	3	-	-	DCL	-	-	-	-	-	-	-	-
D	4	-	-	-	-	-	-	-	-	-	END	-
E	5	BEGIN	-	-	-	-	GET	-	-	-	-	-
F	6	-	-	-	-	-	-	-	-	-	-	-
G	7	-	-	-	-	-	-	-	-	-	-	-
H	-	-	CHECK	-	-	-	-	-	THEN	WHILE	-	-
I	IF	-	-	-	-	-	-	-	-	-	-	-
J	-	-	-	-	-	-	-	-	-	-	-	-
K	-	-	-	-	-	-	-	-	-	-	-	-
L	-	-	CLOSE	-	ELSE	FLOW	-	-	-	-	-	-
M	-	-	-	-	-	-	-	-	-	-	-	-
N	NO	-	-	-	11	-	-	-	-	-	-	-
O	OPEN	-	-	DO	-	FORMAT	12	-	TO	-	-	-
P	8	-	-	-	-	-	-	-	-	-	-	-
Q	-	-	-	-	-	-	-	-	-	-	-	-
R	RETURN	-	-	-	-	FREE	-	PROC	-	WRITE	-	-
S	STOP	-	-	-	-	-	-	-	-	-	-	-
T	9	-	-	-	-	-	-	-	-	-	ENTRY	GOTO
U	-	-	-	-	-	-	-	PUT	-	-	-	-
V	-	-	-	-	-	-	-	-	-	-	-	-
W	10	-	-	-	-	-	-	-	-	-	-	-
X	-	-	-	-	EXIT	-	-	-	-	-	-	-
Y	-	BY	-	-	-	-	-	-	-	-	-	-
Z	-	-	-	-	-	-	-	-	-	-	-	-

A FOREST REPRESENTATION OF THE TRIE GIVEN IN THE TABLE

(REFER CLASS BOOK FOR DETAIL)