Unit-1 Basic concepts of Data structure

➥ **Algorithm: ARRY_TRAVERSE(A, LB, UB)**

This algorithm traverses the array.

- A is an array
- LB is lower limit of an array
- UB is upper limit of an array

Step1. *[initialize]*
Count ← LB
Step2. *[perform traversal and increment counter]*
While count <= LB
write a[count]
count = count + 1
Step3. [finished]
return()

➥ **Algorithm: ARRY_DEL(A, POS, N, ELE)**

This algorithm deletes an element from the array.

- A is an array
- POS indicate position where we want to insert an element
- N represent number of elements in array
- ELE represent element to be inserted
- temp is temporary variable

Step1. *[Delete element]*
ELE ← A[POS]
Step2. *[initialize]*
temp ← POS
Step3. *[Move the elements one position up]*
repeat while temp ≤ N-1
A[temp] ←A[temp + 1]
temp ← temp + 1
Step4. *[Decrease size of array]*
N ← N – 1
Step5. *[Finished]*
return()

➥ **Algorithm: ARRY_INS(A, POS, N, ELE)**

This algorithm inserts an element into the array at position POS.

- A is an array
- POS indicate position where we want to insert an element
- N represent number of elements in array
- ELE represent element to be inserted
- temp is temporary variable

Step1. *[initialize]*
temp ← N
Step2. *[move the elements one position down]*
repeat while temp ≥ POS
A[temp+1] ←A[temp]
temp ← temp + 1
Step3. *[insert element]*
A[POS] ← ELE
Step4. *[Increase array size]*
N ← N + 1
Step5. *[Finished]*
return()

➥ **Algorithm: SEQ_SEARCH(K, N, X)**

This algorithm finds an element X from the given vector (array) K.

- N is the total number of elements in the vector.

Step1. *[initialize search]*
i ← 1
K[N+1] ← X
Step2. *[Search the vector]*
repeat while K[X] ≠ X
i ← i + 1
Step3. *[Successful search?]*
if (i = N + 1)
then
write("Unsuccessful search")
return 0
else
write("Successful search")
return 1

➥ Algorithm: ARRY_SEARCH(A, N, X)

This algorithm finds an element X from the given array.

- A is an array
- N represent number of elements in array
- X is the element to be searched

Step1. *[initialize search]*
  I ← 1
  A[N+1] ← X
Step2. *[search the array]*
  repeat while A[I] ≠ X
    I ← I + 1
Step3. *[Successful search?]*
  if I = N + 1
  then write("Unsuccessful Search")
    return 0
  else
    write("Successful Search")
    return 1

➥ Algorithm: ARRY_SORT(A, N)

This algorithm sorts the given array in ascending order.

- A is an array
- N represent number of elements in array
- PASS denotes pass index
- LAST denotes last unsorted element
- EXCH counts the total number of exchanges made during pass

Step1. *[initialize]*
    LAST ← N
Step2. *[Loop on pass index]*
    Repeat thru step 5 for PASS = 1,2,…, N-1
Step3. *[initialize exchange counter]*
    EXCH ← 0
Step4. *[perform comparison]*
  repeat for I = 1,2,.., LAST - 1
    if A[I] > A[I+1] then
      A[I]←A[I+1]
      EXCH ← EXCH+1
Step5. *[Exchange made for this pass?]*
  if EXCH = 0 then
    write("Sorting completed")
  else
    LAST ← LAST – 1
Step6. *[finished]*
  return()

➥ Algorithm: BINARY_SEARCH(K, N, X)

This algorithm returns the index of element if search is successful, 0 otherwise.

- K is a vector consisting of N elements in the ascending order
- Variables LOW, MID and HIGH denote the lower, middle and upper limit of search interval
- If search is successful, this function returns the index of the searched element, 0 otherwise
- X is the element is to be searched

Step1. *[initialize]*
  LOW ←1
  HIGH ← N
Step2. *[perform search]*
  repeat thru step 4
  while (LOW ≤ HIGH)
Step3. *[obtain index of middle point value]*
  MID ← |(LOW + HIGH) / 2|
Step4. *[compare]*
  If (X < k[MID])
  then HIGH ← MID – 1
  else if (X > k[MID])
  then LOW ← MID + 1
  else write ("SUCCESSFUL SEARCH")
  return (MID)
step5. *[unsuccessful search]*
  write ("UNSUCCESSFUL SEARCH")
  return (0)

Unit-2 Strings

☆ Finding length of string
Algorithm: STR_LEN (str)

- This algorithm counts the length of the given string.
- str is the given string.

Step1. *[initialize]*
        count ← 0
Step2. *[Process until end of the string]*
        Repeat while (str[count] ≠ NULL)
        count = count + 1
Step3. *[Finish]*
        return (count)

☆ Algorithm: TOLOWER (S1, S2)

- This algorithm converts the entered string into lower case.
- S1 is the string that we have to convert.

Step1. *[initialize]*
        i ← 0
Step2. *[convert upper to lower case]*
        while(s1[i] ≠ NULL)
                if(s1[i] ≥ 'a' AND s1[i] ≤ 'z')
                then s2[i] = s1[i] + 32
        i ← i + 1
Step3. *[Finish]*
        S2[i] ← NULL
        return (s2)

☆ Algorithm: STR_CAT (S1, S2, S3)

This algorithm concatenate two strings S1 and S2 into the third string S3.

Step1. *[initialize]*
        count1 ← 0
        count2 ← 0
Step2. *[Process until end of the first string]*
        Repeat while (S1[count1] ≠ NULL)
                S3 [count1] ← S1 [count1]
                count1 ← count1 + 1
Step3. *[Process until end of second string]*
        Repeat while (S2[count2] ≠ NULL)
                S3[count1] ← S2[count2]
                count1 ← count1 + 1
                count2 ← count2 + 1
Step4. *[finish]*
        S3[count1] ← NULL
        return (S3)

☆ Algorithm: STR_APPND (S2, S1)

- This algorithm appends the string S1 with the content of string S2. Means the content of string S2 will be added at the end of the string S1.

Step1. *[initialize]*
        count1 ← 0
        count2 ← 0
Step2. *[Process until end of the string]*
        Repeat while (S1[count1] ≠ NULL)
                count1 ← count1 + 1
Step3. *[Process until end of second string and append it to string one]*
        Repeat while (S2[count2] ≠ NULL)
                S1[count1] ← S2[count2]
                count1 ← count1 + 1

                count2 ← count2 + 1
Step4. *[terminate string one and finish]*
        S1[count1] ← NULL
        return (S1)

☆ Algorithm: STR_REVERSE (org)

- This algorithm reverses the given string (org) and store in into new string (rev).

Step1. *[find length of given string]*
        len ← strlen (org)
Step2. *[initialize]*
        count ← 0
        len ← len – 1
Step3. *[process string and copy character by character]*
        While (len >= 0)
                rev [count] ← org [len]
                count ← count + 1
                len ← len -1
Step4. *[Finish]*
        rev [count] ← NULL
        return (rev)

☆ **Algorithm: COPY(S1, S2)**

- This algorithm copying the content of the given string S1 into other one S2.

```
Step1. [initialize]
        count1 ← 0
Step2. [Process until end of the string and
        do copying]
        Repeat while (S1[count1] ≠ NULL)
        S2 [count1] ← S1 [count1]
        count1 ← count1 + 1
Step3. [terminate copied string]
        S2[count1] ← NULL
Step4. [finished]
        return (S2)
```

☆ **Algorithm: SCOMP (S1,S2)**

This algorithm compares two strings S1 and S2 and finds whether they are equal or not.

```
Step 1. [initialize]
        i ← 0
Step 2. [find length of two strings]
        L1←strlen (S1)
        L2 ← strlen (S2)
Step 3. [compare length of two strings]
        if (L1 ≠ L2)
                then write ("STRINGS ARE NOT
                EQUAL")
                return ()
Step 4. [process strings and compare character
by character]
        Repeat while (i < L1)
                if S1[i] ≠ S2 [i]
                        then write ("SRTINGS ARE
                        NOT EQUAL")
                        return()
                else
                        i ←i + 1
Step 5. [return equal sting]
        write ("STRINGS ARE EQUAL")
        return()
```

☆ **Algorithm: SUBSTR(S, POS, NUM)**

- This algorithm finds the sub string SUB in the given string S, from the position POS followed by NUM characters.

```
Step 1. [initialize]
        i ← 0
        POS ←POS -1
Step 2. [process the string]
        Repeat while (NUM > 0)
                SUB[i] ← S[POS]
                i ←i + 1
                POS ← POS + 1
                NUM ← NUM – 1
Step 3. [finish]
        SUB [i] ← NULL
        return (SUB)
```

☆ **Algorithm: STR_INSERT(S1, S2,POS)** - This algorithm insert the string S2 in given string S1 at the position POS. - S1 is the given string in which you want to insert the string S2 at particular position POS. - TEMP is temporary array. Step1. *[initialize]* i ← 0 j ← 0 step2. *[reach up to the position of insertion and copy string]* repeat while(i < POS) S3[j] ← S1[i] i ← i + 1 j ← j + 1 Step3. *[copy remaining content of S1 into TEMP string]* t ← 0 repeat while(i <=strlen(S1)) TEMP[t] ← S1[i] Step4. *[reset the value of i]* i ← 0 Step5. *[copy the content of string S2 into S3]* repeat while(i <=strlen(S2)) S3[j] ←S2[i] Step6. *[combining string]* strcat(S3, TEMP) Step7. *[Finish]* return (S3)	☆ **Algorithm: STR_DEL(Str1, POS,C)** - This algorithm deletes a sub string C from the given string Str1 after the particular position POS. - Str2 is used to store the partial part of the strings. Step1. *[initialize]* i ← 0 Step2. *[copy the string in str3 up to the position where we want to delete substring]* repeat while (i < POS) str3[i] ← str1[i] i ← i + 1 Step3. *[temporary terminate Str3]* str3[i] ← NULL Step4. *[set the position of i and j]* i ← POS + C j ← 0 Step5. *[copy the string from str1into str2 that we want after deletion]* repeat while (i < strlen(str1)) str2[j] ← str1[i]; i ←I + 1 j ← j + 1 Step6. *[terminate str2]* str2[j] ← NULL Step7. *[concatenate str3 and str2 in str3 to get desired string]* strcat(str3, str2) Step8. *[Finish]* return (str3)

Unit-3 Stack and Queues

✹ Insert an element in Stack

Algorithm: PUSH (S,TOP,X)

- This algorithm insert an element on the top of the stack
- S representing stack
- TOP is a pointer which points to the top of the stack
- X is the element to be inserted

Step 1. *[check for stack overflow]*
    If TOP >= N
    Then write ("STACK OVERFLOW")
        return()
Step 2. *[increment TOP]*
    TOP ← TOP + 1
Step 3. *[insert element]*
    S[TOP] ← X
Step 4. *[finished]*
    return()

✹ Delete an element from Stack

Algorithm: POP (S,TOP)

- This algorithm removes an element from the top of the stack
- S representing stack
- TOP is a pointer which points to the top of the stack
- Y is a variable which holds deleted element

Step 1. *[check for stack underflow]*
    If TOP = 0
    Then write ("STACK UNDERFLOW")
    return(0)
Step 2. *[delete an element]*
    Y ← S[TOP]
Step 3. *[decrement pointer]*
    TOP ← TOP -1
Step 4. *[return deleted element]*
    return (Y)

✹ Insert an element in Queue.

Algorithm: QINSERT (Q,F,R,N,Y)

- This algorithm insert an element into the queue
- Q represents Queue vector containing N elements
- F and R are pointers pointing to the FRONT and REAR end
- Initially F and R set to 0
- Y is the element to be inserted

Step1. *[check for queue overflow]*
    If R >= N Then write ("QUEUE OVERFLOW")
    Return
Step2. *[increment REAR pointer]*
    R ← R + 1
Step3. *[insert element]*
    Q[R] ← Y
Step4. *[is FRONT pointer properly set?]*
    If F = 0 then F ← 1
    return()

✹ Delete an element from Queue.

Algorithm: QDELETE(Q,F,R)

- This algorithm deletes an element from the queue
- Q represents Queue vector containing N elements
- F and R are pointers pointing to the FRONT and REAR end

Step1. *[check for queue underflow]*
    If F = 0
    Then write ("QUEUE UNDERFLOW")
    return (0)
Step2. *[delete element]*
    Y ← Q[F]
Step3. *[check for queue empty]*
    If F = R
    Then F ← R ← 0
    Else F ←F+1
Step4. *[return element]*
    return (Y)

✹ Insert an element in Circular Queue
Algorithm: CQ_INSERT (Q,F,R,N,Y)

- This procedure inserts an element in the circular queue
- Q represents vector, F is front pointer and R is rear pointer, N is no of elements and Y is an element that is to be inserted

```
Step1. [reset rear pointer]
        if R = N
        then   R ← 1
        else   R ← R + 1
step2. [Overflow?]
        if F =R
        then   write ("QUEUE OVERFLOW")
        return()
step3. [insert element]
        Q[R] ← Y
step4. [is front pointer properly set?]
        if F = 0
        then   F ← 1
        return()
```

✹ Delete an element from Circular Queue.
Algorithm: CQ_DELETE (Q,F,R,N)

- This procedure deletes an element and returns it from circular queue
- Q represents vector, F is front pointer and R is rear pointer, N is no of elements
- Y is temporary variable

```
Step1. [underflow?]
        if F = 0
        then   write ("QUEUE UNDERFLOW")
        return (0)
step2. [delete element]
        Y ← Q[F]
step3. [queue empty?]
        if F =R
        then   F ← R ← 0
        return (Y)
step4. [increment front pointer]
        if F = N
        then   F ← 1
        else   F ← F + 1
        return (Y)
```

Unit – 4 Linked List

◈ Insert a node at the start (beginning) of the linked list.

Algorithm: INSERT_START (X, FIRST)

- X is a new element to be inserted, FIRST is a pointer to the first element of linked list which contain INFO and LINK fields.
- AVAIL is a pointer to the first node in availability list.
- NEW is a temporary pointer variable

Step1. *[underflow?]*
    if AVAIL = NULL
    then write (“AVAILABILITY LIST UNDERFLOW”)
    return (FIRST)
Step2. *[obtain address of next free node]*
    NEW ← AVAIL
Step3. *[remove free node from availability stack]*
    AVAIL ← LINK (AVAIL)
Step4. *[initialize fields of new node new node]*
    INFO(NEW) ← X
    LINK(NEW) ← FIRST
    FIRST ← NEW
Step5. *[return address of new node]*
    return (FIRST)

◈ Insert a node at the end of the linked list.

Algorithm: INSERT_END (X, FIRST)

- X is a new element to be inserted, FIRST is a pointer to the first element of linked list which contain INFO and LINK fields.
- AVAIL is a pointer to the node in availability list
- NEW and SAVE are temporary pointer variables

Step1. *[underflow?]*
    if AVAIL = NULL
    then write (“AVAILABILITY STACK UNDERFLOW”)
        Return (FIRST)
Step2. *[obtain address of next free node]*
    NEW ← AVAIL
Step3. *[remove free node from availability stack]*
    AVAIL ← LINK (AVAIL)
Step4. *[initialize fields of new node]*
    INFO (NEW) ← X
    LINK (NEW) ← NULL
Step5. *[is the list empty?]*
    if FIRST = NULL
    then Return (NEW)
Step6. *[initiate search for last node]*
    SAVE ← FIRST
Step7. *[search for end of the list]*
    repeat while LINK (SAVE) ≠ NULL
        SAVE ← LINK (SAVE)
Step8. *[set LINK field of last node to NEW]*
    LINK (SAVE) ← NEW
Step9. *[return first node pointer]*
    Return (FIRST)

◈ Insert a node at specific position.

Algorithm: INSERT_POS (X, FIRST)

- This procedure inserts a new node N into linked list at specified by address X.
- FIRST is a pointer to the first element of linked list which contain INFO and LINK fields.
- AVAIL is a pointer to the node in availability list.
- NEW and SAVE are temporary pointer variables.

Step1. *[availability list underflow?]*
  If AVAIL = NULL
  Then write (“AVAILABILITY LIST UNDERFLOW”)
  return (FIRST)

Step2. *[obtain address of next free node]*
  NEW ← AVAIL

Step3. *[remove free node from availability stack]*
  AVAIL ← LINK (AVAIL)

Step4. *[initialize INFO content of NEW node]*
  INFO (NEW) ← N

Step5. *[is the list empty?]*
  if FIRST = NULL
  then LINK(NEW) ← NULL
    return (NEW)

Step6. *[only first node?]*
  if LINK (FIRST) = NULL OR X = FIRST
  then LINK (NEW) ← FIRST
    FIRST ← NEW
    Return (FIRST)

Step7. *[search the list until desired address found]*
  SAVE ← FIRST
  repeat while LINK (SAVE) ≠ NULL and SAVE ≠ X
    PRED ← SAVE
    SAVE ← LINK (SAVE)

Step8. *[node found?]*
  LINK (PRED) ← NEW
  LINK(NEW) ← SAVE

Step9. *[finished]*
  return (FIRST)

◈ Insert anode in ordered Linked list.

Algorithm: INSERT_ORD (X, FIRST)

- X is a new element to be inserted, FIRST is a pointer to the first element of linked list which contain INFO and LINK fields.
- AVAIL is a pointer to the node in availability list
- NEW andc SAVE are temporary pointer variables

Step1. *[availability list underflow?]*
  if AVAIL = NULL
  then write (“AVAILABILITY STACK UNDERFLOW”)
    return (FIRST)

Step2. *[obtain address of next free node]*
  NEW ← AVAIL

Step3. *[remove free node from availability stack]*
  AVAIL ← LINK (AVAIL)

Step4. *[initialize INFO content of NEW node]*
  INFO (NEW) ← X

Step5. *[is the list empty?]*
  if FIRST = NULL
  then LINK(NEW) ← NULL
    return (NEW)

Step6. *[does the new node precede all other nodes?]*
  if INFO (NEW) ≤ INFO (FIRST)
  then LINK (NEW) ← FIRST
    return (NEW)

Step7. *[initialize temporary pointer]*
  SAVE ← FIRST

Step8. *[search for predecessor of new node]*
  Repeat while LINK(SAVE) ≠ NULL and INFO (LINK (SAVE)) ≤INFO (NEW)
    SAVE ← LINK (SAVE)

Step9. *[set LINK fields of NEW node and its predecessor]*
  LINK(NEW) ← LINK(SAVE)
  LINK(SAVE) ← NEW

Step10. *[return first node]*
  return (FIRST)

◈ Delete the first node in singly liked list
Algorithm: DELETE_FIRST (FIRST)

- This procedure deletes a node from the beginning of a singly linked list.
- FIRST is a pointer to the first element of linked list which contain INFO and LINK fields.
- AVAIL is a pointer to the first node in availability list.
- TEMP is a temporary pointer variable.

```
Step1. [list is empty?]
        if FIRST = NULL
        then   write (" LIST UNDERFLOW")
               return()
Step2. [initialize temporary variable]
        TEMP ← FIRST
Step3. [delete node]
        if LINK (FIRST) = NULL
        then   FIRST ← NULL
        else   FIRST ← LINK (FIRST)
Step4. [return node to the availability list]
        LINK (TEMP) ← AVAIL
        AVAIL ← TEMP
        return()
```

◈ Delete he last node in singly linked list
Algorithm: DELETE_END (FIRST)

- This procedure deletes a node from the end of a singly linked list.
- FIRST is a pointer to the first element of linked list which contain INFO and LINK fields.
- AVAIL is a pointer to the first node in availability list.
- TEMP and PRED are temporary pointer variables. PRED keeping track of predecessor node.

```
Step1. [List is empty?]
        if FIRST = NULL
        then   write ("UNDERFLOW")
               return()
Step2. [Initialize temporary variable]
        TEMP ← FIRST
Step3. [Search target node]
        if(LINK(FIRST = NULL)
        then   FIRST ← NULL
        else   repeat while(LINK(TEMP) ≠
        NULL)
               PRED ← TEMP
               TEMP ← LINK(TEMP)
Step4. [Delete a node]
        LINK(PRED) ← LINK(TEMP)
Step5. [Return node to the availability list]
        LINK(TEMP) ← AVAIL
        AVAIL ← TEMP
        return()
```

◈ Search a node in singly linked list.
Algorithm: SEARCH (FIRST, X)

- This algorithm searches the given element in the list.
- FIRST is a pointer to the first element of linked list.
- X is the element which we want to search.
- SAVE is a temporary variable. POS is the variable showing position of the searched element.

```
Step1. [initialize]
        POS ←1
        SAVE ← FIRST
Step2. [Linked list empty?]
        if (FIRST = NULL)
        then write ("List is empty")
        return()
Step3. [Search element]
        repeat while (INFO(SAVE) ≠ X OR
        LINK(SAVE) ≠ NULL)
               SAVE ←LINK(SAVE)
               POS ← POS + 1
        if (INFO(SAVE) ≠ X)
        then   write("Node is not in the list)
        else   write("Node is found at")
               write(POS)
Step4. [Finished]
        return()
```

◈ **Count the number of nodes in the linked list.**

Algorithm: COUNT (FIRST)

- This procedure counts the number of nodes in the list
- FIRST is a pointer points to the first node of linked list which contain INFO and LINK fields.
- SAVE is a temporary variable
- count is a variable counting number of nodes.

Step1. *[empty list?]*
if FIRST = NULL
then write (“LIST IS EMPTY”)

Step2. *[initialize]*
count ← 1
SAVE ← FIRST

Step3. *[process the list until end of the list]*
repeat while LINK (SAVE) ≠ NULL
count ← count + 1

step4. *[finished]*
return (count)

◈ **Insert a new node at the beginning of the doubly linked list.**

Algorithm: INSERT_START_DBL(L, R, X)

- This procedure inserts the node at start position into the list.
- L and R are left most and right most address of doubly linked list.
- X is a new element to be inserted.
- The node to be inserted is denoted by NEW.
- The left and right links of a node are denoted by LPTR and RPTR, information field is denoted by INFO.
- The name of the element of the list in NODE.

Step1. *[obtain new node from availability list]*
NEW ← NODE

Step2. *[initialize new node]*
INFO (NEW) ← X
LPTR(NEW) ← NULL
RPTR(NEW) ← NULL

Step3. *[insertion into the list]*
if L = NULL AND R = NULL
then L ← R ← NEW
else RPTR(NEW) ← L
LPTR(L) ← NEW
L ← NEW

Step4. *[Finished]*
return()

◈ **Insert a new node at the end of the doubly linked list.**

Algorithm: INSERT_END_DBL(L, R, X)

- This procedure inserts the node at last position into the list.
- L and R are left most and right most address of doubly linked list.
- X is a new element to be inserted.
- The node to be inserted is denoted by NEW.
- The left and right links of a node are denoted by LPTR and RPTR, information field is denoted by INFO.
- The name of the element of the list in NODE.

Step1.*[obtain new node from availability list]*
NEW ← NODE

Step2. *[initialize new node]*
INFO (NEW) ← X
LPTR(NEW) ← NULL
RPTR(NEW) ← NULL

Step3. *[insertion into the list]*
if L = NULL AND R = NULL
then L ← R ← NEW
else LPTR(NEW) ← R
RPTR(R) ← NEW
R ← NEW

Step4. *[Finished]*
return()

◈ Insert a node in doubly linked list. (at any position).

Algorithm: INSERT_ANY_DBL(L, R, M, X)

- This procedure inserts the node at any position into the list.
- L and R are left most and right most address of doubly linked list.
- X is a new element to be inserted.
- The node to be inserted is denoted by NEW.
- The left and right links of a node are denoted by LPTR aknd RPTR, information field is denoted by INFO.
- The name of the element of the list in NODE.
- The insertion to be performed to the left of a specified node with its address given by M.

Step1. *[obtain new node from availability list]*
        NEW ← NODE
Step2. *[copy information field]*
        INFO (NEW) ← X
Step3. *[insertion into the empty list]*
        if R = NULL
        then LPTR (NEW) ← RPTR (NEW) ← NULL
                L ← R ← NEW
                return()
Step4. *[left most insertion]*
        if M = L
        then LPTR (NEW) ← NULL
                RPTR (NEW) ← M
                LPTR (M) ← NEW
                L ← NEW
                return()
Step5. *[insert in middle]*
        LPTR (NEW) ← LPTR (M)
        RPTR (NEW) ← M
        LPTR (M) ← NEW
        RPTR (LPTR (NEW)) ← NEW
        return()

◈ Delete a node from the first position (beginning) in doubly linked list.

Algorithm: DEL_DBL_START(L,R)

- This procedure deletes a node from the beginning of doubly linked list.
- L and R are left most and right most address of doubly linked list.
- The left and right links of a node are denoted by LPTR and RPTR.
- TEMP is a temporary pointer points to the node which is to be deleted.

Step1. *[Check for underflow.]*
        if R = NULL
        then write(“List Underflow”)
                return()
Step2. *[Delete node]*
        TEMP ← L
        if L = R   (single node)
        then L ← R ← NULL
        else L ← RPTR(L)
                LPTR(L) ← NULL
Step3. *[Return deleted node]*
        return(TEMP)

◈ Delete a node from the end (last position) in doubly linked list.

Algorithm: DEL_DBL_LAST(L,R)

Step1. *[Check for underflow.]*
        if R = NULL
        then write(“Underflow”)
                return()
Step2. *[Delete node]*
        TEMP ← R
        if L = R   (single node)
        then L ← R ← NULL
        else R ← LPTR(R)
                RPTR(R) ← NULL
Step 3. *[Return deleted node]*
        return(TEMP)

Unit-5 Sorting and Hashing

✡ Bubble Sort

Algorithm: BUBBLE_SORT (K, N)

- K is a vector of N elements
- PASS denotes the pass counter and LAST denotes the last unsorted element
- I is a index variable
- EXCH is a variable counting number of exchanges made on any pass

Step1. *[initialize]*
    LAST ← N
Step2. *[loop on pass index]*
    Repeat thru step 5 for PASS = 1, 2, 3,.., N-1
Step3. *[initialize exchange counter]*
    EXCH ← 0
Step4. *[perform pair wise comparison on unsorted elements]*
    Repeat for I = 1, 2, .., LAST-1
    if (K[I] > K[I+1])
    then K[I] ←→ K[I+1]
        EXCH ← EXCH +1
Step5. *[where exchanges made?]*
    if EXCH = 0
    then return()
    else LAST = LAST -1
Step6. *[finished]*
    return()

✡ Selection Sort

Algorithm: SELECTION_SORT (K, N)

- This procedure rearranges the vector (elements) in ascending order.
- K is a vector containing N elements
- PASS denotes the pass index.
- MIN_INDEX denotes the position of the smallest element.

Step1. *[loop on pass index]*
    Repeat thru step 4 for PASS = 1,2,3,.., N-1
Step2. *[initialize minimum index]*
    MIN_INDEX ← PASS
    (initially first element set as the minimum index.)
Step3. *[make a pass and obtain element with the smallest value]*
    repeat for I = PASS+1, PASS+2,…., N
        if K[I] < K[MIN_INDEX]
        then MIN_INDEX ←I
Step4. *[exchange elements]*
    if MIN_INDEX ≠ PASS
    then K[PASS] ←→K[MIN_INDEX]
Step5. *[finished]*
    return()

✡ Quick Sort

Algorithm: QUICK_SORT (K, LB, UB)

- K is a vector containing N elements
- LB and UB denotes the lower and upper bound of the table
- FLAG is a logical variable
- KEY is a key value which is being placed at its final position after each pass

Step1. *[initialize]*
    FLAG ← true
Step2. *[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]
    Call QUICK_SORT (K, LB, J-1)
    (sort first sub table)
    Call QUICK_SORT (K, J+1, UB)
    (sort second sub table)
Step3. *[finished]*
    return ()

✡ Insertion Sort

Algorithm: INS_SORT(a,n)

- a in array contains data elements.
- N is the number of elements.
- KEY points to key element to be compared.

Step1. *[initialize i]*
    i ← 1 (pointing to second element of the array)
Step2. *[traverse through the list and compare elements]*
    repeat while (i < n)
        KEY ← a[i]
        j ← i – 1
        repeat while (j >= 0 AND KEY < a[j])
            a[j+1] ← a[j]
            j <- j – 1
            a[j+1] ← KEY
Step3. *[Finish]*
    return()

✡ Radix Sort

Algorithm: RADIX_SORT()

For each digit in the key
{
    Initialize the queues.
    repeat for i=0 to n
        y ← a[i]
        j ← digit at the particular position of y
        insert y in the appropriate queue[j] (pocket).
    Combine the queue to form a new linked list
}

✡ Merge Sort

Algorithm: MERGE_SORT (list1, n, list2, n1, n2)

- This algorithm sorts two tables into third one (both tables must be sorted)
- List1 is the first list (table) of n elements.
- List2 is the second list (table) of m elements.
- List3 is the third list (table) where elements are to be sorted.
- n1 is the number of elements of list1 and n2 is number of elements of list2.

Step1. *[initialize]*
    i ← 0
    j ← 0
    k ← 0
step2. *[initialize loop]*
    repeat thru step 3 while ((i<n) and (j<m))
step3. *[comparing corresponding elements]*
    if (list1[i] < list2[j])
    then list3[k] ← list1[i]
        i ← i+1
        k ← k+1
    else if (list1[i] > list2[j])
    then list3[k] ← list2[j]
        j ← j+1
        k ← k+1
    else //if elements of both lists are same
        list3[k] ← list1[i]
        i ← i+1
        k ← k+1
step4. *[size of list1 is greater than list2]*
    if (i < n)
    then repeat for x = i, i+1,….., n-1
        list3[k] ← list1[i]
        k ← k+1
        i ← i+1
step5. *[size of list2 is greater than list1]*
    if (j<m)
    then repeat for y =j, j+1,….., m-1
        list3[k] ← list2[j]
        k ← k+1
        j ← j+1
step6. *[finished]*
    return()

Unit – 6 Tress

❋ **Algorithm: PREORDER (T)**

- This function traverses the tree in pre-order
- T is a pointer which points to the root node of the tree
- LPTR and RPTR denotes to the left pointer and right pointer of the particular node.

Step1. *[process the root node]*
    If T != NULL
    then write (DATA (T))
    else write ('EMPTY TREE')
        return
step2. *[process the left sub tree]*
    if LPTR (T) != NULL
    then call PREORDER (LPTR(T))
step3. *[process the right sub tree]*
    if RPTR (T) != NULL
    then call PREORDER (RPTR(T))
step4. *[finished]*
    return()

❋ **Algorithm: INORDER (T)**

- This function traverses the tree in in-order
- T is a pointer which points to the root node of the tree
- LPTR and RPTR denotes to the left pointer and right pointer of the particular node.

Step1. *[check for empty tree]*
    if T = NULL
    Then write ('EMPTY TREE')
        return
step2. *[process the left sub tree]*
    if LPTR (T) != NULL
    then call INORDER (LPTR(T))
step3. *[process the root node]*
    write (DATA (T))
step4. *[process the right sub tree]*
    if RPTR (T) != NULL
    then call INORDER (RPTR(T))
step5. *[finished]*
    return()

❋ **Algorithm: POSTORDER (T)**

- This function traverses the tree in post-order
- T is a pointer which points to the root node of the tree
- LPTR and RPTR denotes to the left pointer and right pointer of the particular node.

Step1. *[check for empty tree]*
    If T = NULL
    Then write ("EMPTY TREE")
        Return
Step2. *[process the left sub tree]*
    if LPTR (T) != NULL
    then call POSTORDER (LPTR(T))
step3. *[process the right sub tree]*
    if RPTR (T) != NULL
    then call POSTORDER (RPTR(T))
step4. *[process the root node]*
    write (DATA (T))
step5. *[finished]*
    return()

❋ **Algorithm for copy operation on binary tree.**

Algorithm: COPY_BT(T)

Step1. *[check for empty tree]*
    if T = NULL
    then write "Empty Tree"
        return(0)
Step2. *[create a new node]*
    NEW ← NODE
Step3. *[Copy information field]*
    DATA(NEW) ← DATA(T)
Step4. *[Set the structural links]*
    LPTR(NEW) ← COPY(LPTR(T))
    RPTR(NEW) ← COPY(RPTR(T))
Step5. *[Return address of new node]*
    return (NEW)