SASTRA DEEMED TO BE UNIVERSITY
SCHOOL OF COMPUTING

I B.TECH. CSBS

CSE210 – DATA STRUCTURES & ALGORITHMS LABORATORY

Course Objective

This course will help the learner to select appropriate data structure, design technique and algorithm for a given application.

LIST OF EXERCISES

1. Create a Stack and perform the stack operations using arrays.
2. Towers of Hanoi using user defined stacks.
3. Create queue and perform the queue operation using arrays
4. Reading, writing, and addition of polynomials.
5. Line editors with line count, word count showing on the screen.
6. Trees with all operations- Binary search tree, AVL tree
7. All graph algorithms- BFS and DFS
8. Sorting Techniques-Insertion sort, Merge sort
9. Searching techniques - Binary search and Hash search
10. Saving / retrieving non-linear data structure in/from a file

Ex. No. 1: Operations on Stack implemented using array

Declare a stack S having top containing the index of the current top most element and an array of integers. Define the operations ISFULL, ISEMPTY, PUSH, POP and PEEK. Provide the options

1. Multi Push → inserting multiple elements into stack by reading from a text file and pushing them into stack
2. Multi Pop → deleting all the elements from the stack and writing them into a text file,
3. Push → inserting single element into stack by reading from the console and pushing it into the stack.
4. Pop → deleting topmost element from the stack and display it on the console
5. Peek → retrieve the topmost element from the stack and display it on the console.
6. Exit → Terminating the execution

$\boldsymbol{Algorithm\ ISFULL(S)}$

$//\ To\ check\ whether\ the\ stack\ is\ full\ or\ not$

1. $If\ S.top\ >=\ S.length$
2. $\quad Return\ TRUE$
3. $Else$
4. $\quad Return\ FALSE$

$\boldsymbol{Algorithm\ ISEMPTY(S)}$

$//\ To\ check\ whether\ the\ stack\ S\ is\ empty\ or\ not$

1. $If\ S.top\ =\ 0$
2. $\quad Return\ TRUE$
3. $Else$
4. $\quad Return\ FALSE$

$\boldsymbol{Algorithm\ PUSH(S, x)}$

$//\ To\ insert\ an\ element\ x\ into\ stack\ S$

1. $If\ ISFULL(S)$
2. $\quad Print\ \text{“}Stack\ Overflows\text{”}$
3. $\quad Exit$
4. $Else$
5. $\quad S.top\ =\ S.top\ +\ 1$
6. $\quad S[S.top]\ =\ x$

Algorithm POP(S)

//To delete an element from the stack S

1. *If ISEMPTY(S)*
2. *Print "Stack Underflows"*
3. *Exit*
4. *Else*
5. $x = S[S.top]$
6. $S.top = S.top - 1$
7. *Return x*

Algorithm PEEK(S)

//To retrieve an element from the stack S

1. *If ISEMPTY(S)*
2. *Print "Stack Underflows"*
3. *Exit*
4. *Else*
5. *Return* $S[S.top]$

Ex. No. 2: Implementing Towers of Hanoi using Stacks

Declare a stack containing top, an array of structure containing the following fields to hold the value of N, source needle SN, intermediate needle IN, destination needle DN and return address RA.

$\boldsymbol{Algoritnm\ HANOI}(\boldsymbol{N})$

$//\, To\ move\ N\ discs\ \ from\ source\ needle\ `A'\ to\ destination\ needle\ `C'\ using$
$//\, intermediate\ needle\ `B'$.

1. $t\ =\ (N, `A', `B', `C', 18)$
2. $PUSH(S, t)$
3. $t\ =\ PEEK(S)$
4. $if\ t.N\ =\ 0$
5. $\quad goto\ step\ t.RA$
6. $Else$
7. $\quad t\ =\ (t.N - 1, t.SN, t.IN, t.DN, 9)$
8. $\quad goto\ step\ 2$
9. $t\ =\ POP(S)$
10. $t\ =\ PEEK(S)$
11. $Print\ `Move\ Disc\ ', t.N, `\ from\ ', t.SN, `\ to\ ', t.DN$
12. $t\ =\ PEEK(S)$
13. $t\ =\ (t.N - 1, t.IN, t.SN, t.DN, 15)$
14. $goto\ step\ 2$
15. $t\ =\ POP(S)$
16. $t\ =\ PEEK(S)$
17. $goto\ step\ t.RA$
18. $Return$

Ex. No. 3: Operations on Queue implemented using array

Declare a queue Q having front and rear containing the index of the current first & last elements and an array of integers. Define the operations ISFULL, ISEMPTY, ENQUEUE and DEQUEUE.

Provide the options

1. Multi Enqueue → inserting multiple elements into queue by reading from a text file and appending them into end of the queue
2. Multi Dequeue → deleting all the elements from the queue and writing them into a text file,
3. Enqueue → inserting single element into end of the queue by reading from the console and enqueuing it into the queue.
4. Dequeue → deleting first element from the queue and display it on the console
5. Exit → Terminating the execution

$\boldsymbol{Algorithm\ ISFULL(Q)}$

$//\ To\ check\ whether\ the\ queue\ is\ full\ or\ not$

1. $If\ Q.rear\ >=\ Q.length$
2. $\quad Return\ TRUE$
3. $Else$
4. $\quad Return\ FALSE$

$\boldsymbol{Algorithm\ ISEMPTY(Q)}$

$//\ To\ check\ whether\ the\ queue\ Q\ is\ empty\ or\ not$

1. $If\ Q.front\ =\ 0$
2. $\quad Return\ TRUE$
3. $Else$
4. $\quad Return\ FALSE$

$\boldsymbol{Algorithm\ ENQUEUE(Q, x)}$

$//\ To\ insert\ an\ element\ x\ into\ queue\ Q$

1. $If\ ISFULL(Q)$
2. $\quad Print\ \text{“}Queue\ Overflows\text{”}$
3. $\quad Exit$
4. $Else$
5. $\quad Q.rear\ =\ Q.rear\ +\ 1$
6. $\quad Q[Q.rear]\ =\ x$
7. $\quad If\ Q.front\ =\ 0$
8. $\qquad Q.front\ =\ 1$

$\boldsymbol{Algorithm\ DEQUEUE(Q)}$

$//To\ delete\ an\ element\ from\ the\ Queue\ Q$

1. $If\ ISEMPTY(Q)$
2. $\quad Print\ \text{“}Queue\ Underflows\text{”}$

```
    Exit
Else
    x = Q[Q.front]
    If Q.front = Q.rear
        Q.front = Q.rear = 0
    Else
        Q.front = Q.front + 1
    Return x
```

Ex. No. 4: Reading, writing, and addition of polynomials.

Represent the polynomial as a singly linked list with the following node structure:

Declare a Singly Linked List containing a pointer called **'First'** to hold the address of the first element of the list. Each node in the singly linked list should contain three fields Coef, Exp and Link to hold the coefficient of the term, exponent of the term and address of the next term respectively. Provide functions for inserting a node into the list in the decreasing order of its Exp, deleting a node from the list, searching a node having a given Exp, traversing the list and displaying all the terms of a polynomial.

Algorithm INSERT_ORD(P, coef, exp)

```
// To insert a new term of the polynomial in the decreasing order its exponent
n = Allocate_Node()
n→coef = coef
n→exp = exp
n→link = NULL
if P.First = NULL
    P.First = P.Last = n
else
    prev = NULL
    cur = First
    while cur != NULL and cur→exp > n→exp
        prev = cur
        cur = cur→link
    If prev = NULL
        P.First = n
    Else
        n→link = cur
        prev→link = n
        if cur = NULL
            P.Last = n
Return P
```

Algorithm INSERT_AT_LAST(P, coef, exp)

// *To insert a new term of the polynomial at end*

1. $n = Allocate\_Node()$
2. $n \rightarrow coef = coef$
3. $n \rightarrow exp = exp$
4. $n \rightarrow link = NULL$
5. $if\ P.First = NULL$
6. $\quad n \rightarrow link = P.First$
7. $\quad P.First = P.Last = n$
8. $Else$
9. $\quad n \rightarrow link = P.Last \rightarrow link$
10. $\quad P.Last \rightarrow link = n$
11. $\quad P.Last = n$
12. $Return\ P$

Algorithm DisplayPoly(P)

// *To display the polynomial*

1. $If\ P.First = NULL$
2. $\quad Print$ *‘Empty Polynomial’*
3. $Else$
4. $Temp = P.First$
5. $While\ Temp \rightarrow Link\ != NULL$
6. $\quad Print\ Temp \rightarrow coef,$ “ x^ “ $, Temp \rightarrow Exp,$ “ + ”
7. $\quad Temp = Temp \rightarrow link$
8. $Print\ Temp \rightarrow coef,$ “ x^ “ $, Temp \rightarrow exp$
9. $Return$

Algorithm CreatePoly()

// *Create a polynomial P*

1. $P.First = P.Last = NULL$
2. $While\ there\ is\ input$
3. $\quad Read\ coef, exp$
4. $\quad P = INSERT\_ORD(P, coef, exp)$
5. $Return\ P$

Algorithm DeletePoly()

// *Release the nodes of polynomial P*

1. $t = P.First$
2. $While\ t\ != NULL$
3. $\quad x = t$
4. $\quad t = t \rightarrow link$
5. $\quad RETNODE(x)$

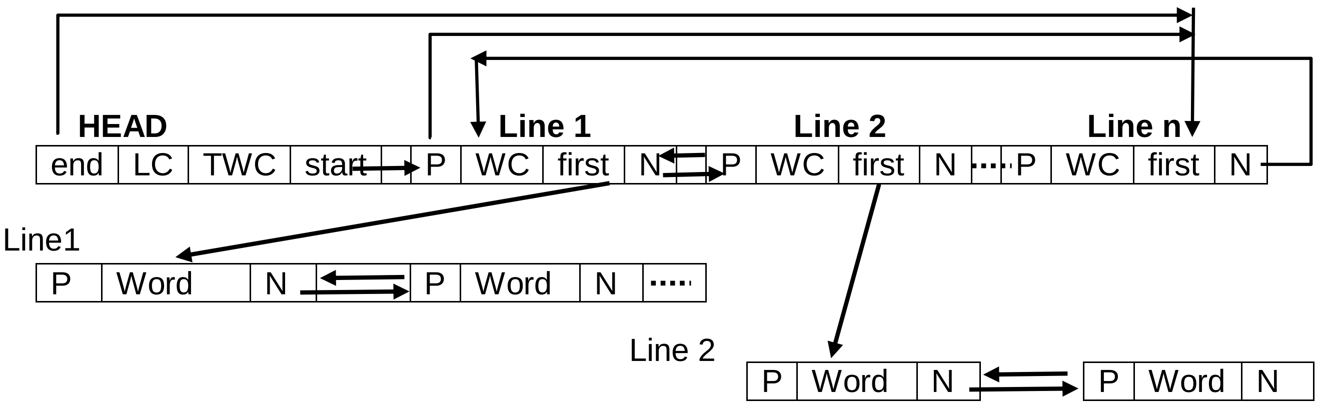
Algorithm ADD_POLY(P, Q)

// Adding two polynomials P and Q.

```
R.First = R.Last = NULL
t1 = P
t2 = Q
While t1 != NULL and t2 != NULL
    If t1→exp > t2→exp
        R = INSERT_AT_LAST(R, t1→coef, t1→exp)
        t1 = t1→link
    Else if t1→exp < t2→exp
        R = INSERT_AT_LAST(R, t2→coef, t2→exp)
        t2 = t2→link
    Else
        coef = t1→coef + t2→coef
        exp = t1→exp
        If coef != 0
            R = INSERT_AT_LAST(R, coef, t2→exp)
        t1 = t1→link
        t2 = t2→link
While t1 != NULL
    R = INSERT_AT_LAST(R, t1→coef, t1→exp)
    t1 = t1→link
While t2 != NULL
    R = INSERT_AT_LAST(R, t2→coef, t2→exp)
    t2 = t2→link
Return R
```

Ex. No. 5: Line editors with line count, word count showing on the screen

Create a text editor that allows you to create a text file and read and write contents to the file line by line. Also provide options to insert a new line, delete an existing line, to find and replace all occurrences of a given word in the file with another word. Use circular doubly linked list to store the line as CDLL of words. P indicates link to previous node, N indicates link to next node, 'first' indicates pointer to first word of the line i, WC indicates the number of word line i, start indicates pointer to first line and end indicates pointer to last line, LC indicates the total number of lines and TWC indicates the total number of words in entire list.



$\boldsymbol{Algorithm\ createHead}()$

$//\ To\ create\ headnode$

1. $head\ =\ GETNODE()$
2. $head \rightarrow start\ =\ head \rightarrow end\ =\ NULL$
3. $head \rightarrow LC\ =\ head \rightarrow TWC\ =\ 0$
4. $return\ head$

$\boldsymbol{Algorithm\ Append}(\boldsymbol{head, line})$

$//\ To\ append\ a\ line\ to\ the\ end\ of\ CDLL$

1. $lh\ =\ GETNODE()$
2. $lh \rightarrow P\ =\ lh \rightarrow N\ =\ NULL$
3. $lh \rightarrow first\ =\ NULL$
4. $lh \rightarrow WC\ =\ 0$
5. $if\ (head \rightarrow end\ ==\ NULL)$
6. $\quad head \rightarrow start\ =\ head \rightarrow end\ =\ lh$
7. $\quad lh \rightarrow P\ =\ lh$
8. $\quad lh \rightarrow N\ =\ lh$
9. $else$
10. $\quad lh \rightarrow P\ =\ head \rightarrow end$
11. $\quad lh \rightarrow N\ =\ head \rightarrow end \rightarrow N$
12. $\quad head \rightarrow end \rightarrow N\ =\ lh$

```
  head→end = lh
  head→start→P = lh
i = 0
while line[i] ≠ '\0'
    k = 0;
    while line[i] is not a space and not end of line
        word[k] = line[i]
        k = k + 1
        i = i + 1
    lh→WC = lh→WC + 1
    nn = GETNODE()
    nn→next = nn→prev = NULL
    nn→word = word)
    if lh→first == NULL
lh→first = nn
nn→prev = nn→next = NULL
last = nn
    else
nn→prev = last
last→next = nn
last = nn
    while line[i] is a space
        i = i + 1
head→TWC = head→TWC + lh→WC
head→LC = head→LC + 1
```

***Algorithm Insert_At_Loc*(*head*, *loc*, *line*)**

// To insert a line at the specified position

```
  WC = 0
  last = NULL
  lh = GETNODE()
  lh→P = lh→N = NULL
  lh→first = NULL
  lh→WC = 0
  i = 0
while line[i] ≠ '\0'
    k = 0;
    while line[i] is not a space and not end of line
        word[k] = line[i]
        k = k + 1
        i = i + 1
    lh→WC = lh→WC + 1
    nn = GETNODE()
    nn→next = nn→prev = NULL
```

```
    nn→word = word)
    if lh→first = NULL
lh→first = nn
nn→prev = nn→next = NULL
last = nn
    else
nn→prev = last
last→next = nn
last = nn
    while line[i] is a space
        i = i + 1
  if (loc = 1)
      lh→P = head→end
      lh→N = head→start
      head→start→P = lh
      head→end→N = lh
      head→start = lh
  else
       cur_line = head→start
      k = 1
      while k < loc − 1 and cur_line→N ≠ head→start
      cur_line = cur_line→N;
      k = k + 1
      lh→P = cur_line
      lh→N = cur_line→N
      cur_line→N→P = lh
      cur_line→N = lh
      if  head→end = cur_line
        head→end = lh
head→TWC = head→TWC + lh→WC
head→LC = head→LC + 1
```

***Algorithm Delete_At_Loc*(*head*, *loc*)**

// To delete a line at the specified position

```
i = 1
if head→start == NULL
  print "Empty Text."
    return;
if head→LC < loc
    print "No such line exists in the text."
    return
cur_line = head→start
while cur_line ≠ head→end and i < loc
     cur_line = cur_line→N
```

```
        i = I + 1
    if cur_line→P ≠ head→end
        cur_line→P→N = cur_line→N
    else
        head→start = cur_line→N          //delete first node
    if cur_line→N ≠ head→start
        cur_line→N→P = cur_line→P
    else
        head→end = cur_line→P//delete last node
    head→LC = head→LC − 1
    head→TWC = head→TWC − cur_line→WC
    cur_word = cur_line→first
    while cur_word ≠ NULL
        t = cur_word
        cur_word = cur_word→next
        RETNODE(t)
    RETNODE(cur_line)
    return
```

Algorithm Find(***head***, ***word***)

// To find the position of all occurrences of a word in the text

```
flag = FALSE
i = 1
if head→start = NULL
    print "Empty text"
    return
cur_line = head→start
while cur_line ≠ head→end
    pos = 0
    cur_word = cur_line→first
    while cur_word ≠ NULL
        pos = pos + 1
        if cur_word→word = word
            flag = TRUE
                print "Occurrence of", word, "is in Line – ", i, "at position ", pos
        cur_word = cur_word→next
    cur_line = cur_line→N
    i = i + 1
cur_word = head→end→first
pos = 0
while cur_word ≠ NULL
    pos = pos + 1
    if cur_word→word = word
        flag = TRUE
```

```
            print "Occurrence of", word, "is in Line – ", i, "at position ", pos
        cur_word = cur_word→next
    if flag = FALSE
        print word, " is not present in the text."
```

***Algorithm Display*(*head*)**

// To display all the lines of the text

```
lh = head→start
i = 1
while lh ≠ head→end
    cur = lh→first
    print "Line – ", i
    while cur ≠ NULL
        print cur→word
        cur = cur→next
    i = i + 1
    lh = lh→N
cur = lh→first
print "Line − ", i
while cur != NULL
    print cur→word
    cur = cur→next
print "No. of lines = ", head→LC, No. of words = ", head→TWC
return
```

Ex. No. 6a: Operations on Binary Search Tree

Declare a BST T having root containing the address of the root element. Define the operations Insertion, Deletion, Traversal in preorder, inorder and postorder, search for a given element, finding the minimum element, and finding the maximum element.

Algorithm InsertBST(root, x)

// To insert a new element x into a BST

```
1.  T = GETNODE()
2.    T→data = x
3.    T→lchild = T→rchild = NULL
4.    if root = NULL
5.        root = T
6.        return
7.    parent = NULL
8.    cur = root
9.    while cur ≠ NULL
10.       parent = cur
11.       if x < cur→data
12.           cur = cur→lchild
13.       else if x > cur→data
14.           cur = cur→rchild
15.       else
16.           Print "Duplicate value. Cannot insert"
17.           return
18.   if x < parent→data
19.       parent→lchild = T
20.   else
21.       parent→rchild = T
22.   return
```

Algorithm Inorder(T)

// To traverse the BST in inorder: Left, Data, Right

```
1.  if T ≠ NULL
2.      Inorder(T→lchild)
3.      Print T→data
4.      Inorder(T→rchild)
```

Algorithm Preorder(T)

// To traverse the BST in preorder: Data, Left, Right

```
1.  if T ≠ NULL
2.      Print T→data
```

```
3.        Preorder(T→lchild)
4.        Preorder(T→rchild)
```

***Algorithm Postorder*(*T*)**

// To traverse the BST in postorder: Left, Right, Data

```
1.  if T ≠ NULL
2.        Postorder(T→lchild)
3.        Postorder(T→rchild)
4.        Print T→data
```

***Algorithm Minimum*(*root*)**

// To find the minimum element in the BST

```
1.  T = root
2.  while T→lchild ≠ NULL
3.  T = T→lchild
4.  return T→data
```

***Algorithm Maximum*(*root*)**

// To find the maximum element in the BST

```
1.  T = root
2.  while T→rchild ≠ NULL
3.  T = T→rchild
4.  return T→data
```

***Algorithm SearchBST*(*root*, *x*)**

// To find an element x in the BST

```
1.  if root = NULL
2.          print "Empty Binary Search Tree."
3.          return
4.     cur = root
5.     while cur ≠ NULL
6.            if x < cur→data
7.                 cur = cur→lchild
8.           else if x > cur→data
9.                 cur = cur→rchild
10.          else
11.                Print "Element Found"
12.                return
13.    Print "Element Not Found"
14.    return root
```

Algorithm DeleteBST(T, x)

// To delete an element x from the BST

```
if T = NULL
     print "Element not found!! Deletion cannot be performed."
     return T
if x < T→data
    T→lchild = DeleteBST(T→lchild, x)
else if x > T→data
    T→rchild = DeleteBST(T→rchild, x)
else
     if T→lchild = NULL
          dltptr = T
          T = T→rchild
          RETNODE( dltptr)
     else if T→rchild = NULL
          dltptr = T
          T = T→lchild
          RETNODE(dltptr)
     else
          dltptr = T→lchild
          while dltptr−> rchild ≠ NULL
                dltptr = dltptr→rchild
          T→data = dltptr→data
          T→lchild = DeleteBST(T→lchild, dltptr→data)
return T
```

Ex. No. 6b: Operations on AVL Tree

Create a height balanced BST (AVLTree) T having root containing the address of the root element. Define the operations Insertion, Traversal in preorder, inorder and postorder, search for a given element.

$\boldsymbol{Algorithm\ InsertAVL(T, x)}$

$//\ To\ insert\ a\ new\ element\ x\ into\ a\ BST$

1. $if\ T\ =\ NULL$
2. $T\ =\ GETNODE()$
3. $T \rightarrow data\ =\ x$
4. $T \rightarrow lchild\ =\ T \rightarrow rchild\ =\ NULL$
5. $Return\ T$
6. $If\ x\ <\ T \rightarrow data$
7. $T -> lchild\ =\ InserAVL(T \rightarrow lchild, x)$
8. $else\ if\ x\ >\ T \rightarrow data$
9. $T -> rchild\ =\ InserAVL(T \rightarrow rchild, x)$
10. $else$
11. $Print\ "Duplicate\ value. Cannot\ insert"$
12. $return$
13. $bf\ =\ getBalanceFactor(T)$
14. $if\ bf\ ==\ 2\ and\ x\ <\ T \rightarrow lchild \rightarrow data$
15. $return\ RightRotate(T)$
16. $if\ bf\ ==\ -2\ and\ x\ >\ T \rightarrow rchild \rightarrow data$
17. $return\ LeftRotate(T)$
18. $if\ bf\ ==\ 2\ and\ x\ >\ T \rightarrow lchild \rightarrow data$
19. $T \rightarrow lchild\ =\ LeftRotate(T \rightarrow lchild)$
20. $return\ RightRotate(T)$
21. $if\ bf\ ==\ -2\ and\ x\ <\ T \rightarrow rchild \rightarrow data$
22. $T \rightarrow rchild\ =\ RightRotate(T \rightarrow rchild)$
23. $return\ LeftRotate(T)$
24. $return\ T$

$\boldsymbol{Algorithm\ getHeight(T)}$

$//\ To\ find\ the\ height\ of\ the\ node$

1. $if\ T\ =\ NULL$
2. $return\ 0$
3. $else$
4. $hl\ =\ getHeight(T \rightarrow lchild)$
5. $hr\ =\ getHeight(T \rightarrow rchild)$
6. $If\ hl\ >=\ hr$
7. $return\ hl + 1$
8. $else$

9. $return\ hr + 1$

$\boldsymbol{Algorithm\ getBalanceFactor(T)}$

$//\ To\ find\ the\ balance\ factor\ of\ the\ node\ T$

1. $If\ T\ =\ NULL$
2. $\quad return\ 0$
3. $else$
4. $\quad hl\ =\ getHeight(T \rightarrow lchild)$
5. $\quad hr\ =\ getHeight(T \rightarrow rchild)$
6. $\quad return\ hl\ -\ hr$

$\boldsymbol{Algorithm\ Inorder(T)}$

$//\ To\ traverse\ the\ BST\ in\ inorder: Left, Data, Right$

1. $if\ T\ \neq\ NULL$
2. $\quad Inorder(T \rightarrow lchild)$
3. $\quad Print\ T \rightarrow data$
4. $\quad Inorder(T \rightarrow rchild)$

$\boldsymbol{Algorithm\ Preorder(T)}$

$//\ To\ traverse\ the\ BST\ in\ preorder: Data, Left, Right$

1. $if\ T\ \neq\ NULL$
2. $\quad Print\ T \rightarrow data$
3. $\quad Preorder(T \rightarrow lchild)$
4. $\quad Preorder(T \rightarrow rchild)$

$\boldsymbol{Algorithm\ Postorder(T)}$

$//\ To\ traverse\ the\ BST\ in\ postorder: Left, Right, Data$

1. $if\ T\ \neq\ NULL$
2. $\quad Postorder(T \rightarrow lchild)$
3. $\quad Postorder(T \rightarrow rchild)$
4. $\quad Print\ T \rightarrow data$

$\boldsymbol{Algorithm\ SearchAVL(root, x)}$

$//\ To\ find\ an\ element\ x\ in\ the\ BST$

1. $if\ root\ =\ NULL$
2. $\quad print\ \text{“}Empty\ Binary\ Search\ Tree.\text{”}$
3. $\quad return$
4. $cur\ =\ root$
5. $while\ cur\ \neq\ NULL$
6. $\quad if\ x\ <\ cur \rightarrow data$

7. $\quad cur = cur \rightarrow lchild$
8. $\quad else\ if\ x > cur \rightarrow data$
9. $\quad\quad cur = cur \rightarrow rchild$
10. $\quad else$
11. $\quad\quad Print$ "$Element\ Found$"
12. $\quad\quad return$
13. $Print$ "$Element\ Not\ Found$"
14. $return\ root$

$\boldsymbol{Algorithm\ LeftRotate(T)}$

1. $Y = T \rightarrow rchild$
2. $T \rightarrow rchild = Y \rightarrow lchild$
3. $Y \rightarrow lchild = T$
4. $return\ Y$

$\boldsymbol{Algorithm\ RightRotate(T)}$

1. $Y = T \rightarrow lchild$
2. $T \rightarrow lchild = Y \rightarrow rchild$
3. $Y \rightarrow rchild = T$
4. $return\ Y$

Exercise 8a: Insertion Sort

Given an unsorted array A of n elements, arrange the elements in ascending order using Insertion Sort and Merge Sort.

$\boldsymbol{Algorithm\ InsertionSort(A, n)}$

//To arrange the elements of the array A in ascending order

1. $for\ j = 2\ to\ n$
2. $\quad key = A[j]$
3. $\quad i = j - 1$
4. $\quad while\ i > 0\ and\ A[i] \geq key$
5. $\qquad\qquad A[i+1] = A[i]$
6. $\qquad\qquad i = i - 1$
7. $\quad A[i+1] = key$

Exercise 8b: Merge Sort

$\boldsymbol{Algorithm\ MergeSort(A, low, high)}$

//To arrange the elements of the array $A[low..high]$ in ascending //order using Divide & Conquer Strategy

1. $if\ low < high$
2. $\quad mid = \left\lfloor \frac{(low+high)}{2} \right\rfloor$
3. $\quad MERGE\_SORT(A,\ low,\ mid)$
4. $\quad MERGE\_SORT(A,\ mid+1,\ high)$
5. $\quad MERGE(A,\ low,\ mid,\ high)$

$\boldsymbol{Algorithm\ MERGE(A, low, mid, high)}$

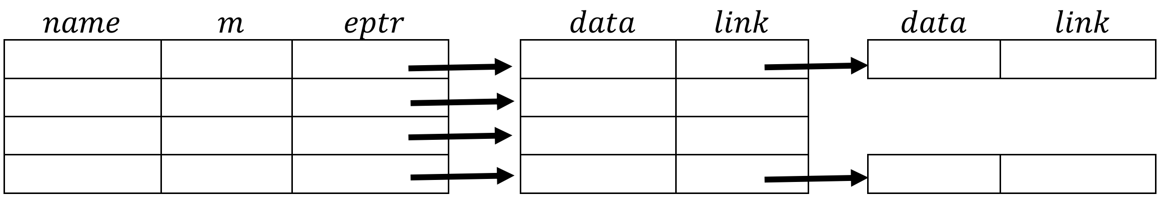
//Merging the elements of two sorted subarrays $A[low..mid]$ and //$A[mid+1..high]$ into $A[low..high]$. $T[low..high]$ is a //temporary array used for merging

1. $i = low,\ j = mid + 1,\ k = low$
2. $while\ i \leq mid\ and\ j \leq high$
3. $\quad if\ A[i] \leq A[j]$
4. $\qquad T[k] = A[i]$
5. $\qquad i = i + 1$
6. $\quad else$
7. $\qquad T[k] = A[j]$
8. $\qquad j = j + 1$
9. $\quad k = k + 1$
10. $if\ i > mid$
11. $\quad while\ j \leq high$
12. $\qquad T[k] = A[j]$
13. $\qquad j = j + 1$

```
        k = k + 1
else
    while i ≤ mid
        T[k] = A[i]
        i = i + 1
        k = k + 1
for i = low to high
    A[i] = T[i]
```

Exercise 9: Traversing the graph in BFS & DFS

The graph G=(V,E) is represented as adjacency list. It is stored as a vertex table consisting of 'n' vertices with the following structure where $name$ is vertex name, m is number of adjacent vertices, $eptr$ holds the address of the first node in the singly linked list of adjacent vertices. In the singly linked list the index of the adjacent vertex is stored in $data$ and the $link$ points to the next node in the list, if any.



Traverse the vertices in breadth first and depth first manner.

Algorithm BFS(G, s)

```
CreateQueue(Q)
for i = 1 to n
      v[i].dist = 0
      v[i].visited = false
v[s].visited = true
v[s].dist = 0
ENQUEUE(Q, s)
while not isEmpty(Q)
      u = DEQUEUE(Q)
      t = v[u].eptr
      while t ≠ NULL
            w = t → data
            if v[w].visited = false
                  v[w].visited = true
                  v[w].dist = v[u].dist + 1
                  ENQUEUE(Q, w)
            t = t → link
```

Algorithm DFS(G)

```
for i = 1 to n
      v[i].dfn = 0
      v[i].visited = false
count = 0
for i = 1 to n
```

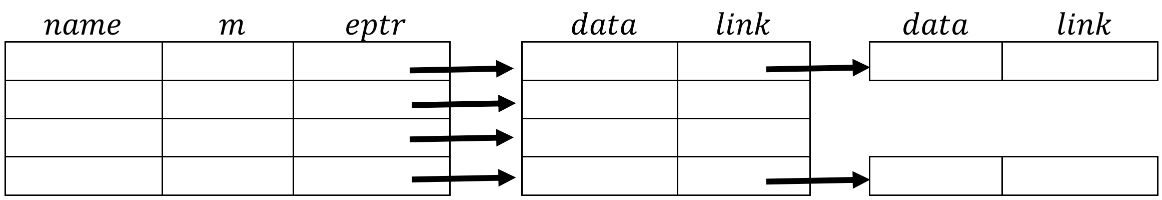
```
        if v[i].visited = false
                DFS_VISIT(G, u)
```

Algorithm DFS_VISIT(G, u)

```
count = count + 1
v[u].dfn = count
v[u].visited = true
t = v[u].eptr
while t ≠ NULL
        w = t → data
        if v[w].visited = false
                print v[u].name, "→", v[w].vname
                DFS_VISIT(G, w)
        t = t → link
```

Exercise 10: Store and Load Non-Linear Data Structure (Graph)

The graph G=(V,E) is represented as adjacency list. It is stored as a vertex table consisting of 'n' vertices with the following structure where $name$ is vertex name, m is number of adjacent vertices, $eptr$ holds the address of the first node in the singly linked list of adjacent vertices. In the singly linked list the index of the adjacent vertex is stored in $data$ and the $link$ points to the next node in the list, if any.



***Algorithm StoreGraph*($g, filename$)**
// To save the details of graph g in the given file.

1. $Open\ file\ for\ \ writing$
2. $WriteFile\ \ \ g.n$
3. $for\ i = 1\ to\ g.n$
4. $\quad WriteFile\ \ \ g.v[i].name$
5. $\quad WriteFile\ g.v[i].m$
6. $\quad p\ =\ g.v[i].eptr$
7. $\quad while\ (p! =\ NULL)$
8. $\qquad WriteFile\ \ p \to data$
9. $\qquad p\ =\ p \to link$
10. $Close\ the\ file$

***Algorithm LoadGraph*($filename$)**
//To load the details of graph from the given file to graph data structure g

1. $Open\ file\ for\ \ Reading$
2. $ReadFile\ \ \ g.n$
3. $for\ i = 1\ to\ g.n$
4. $\quad ReadFile\ \ \ g.v[i].name$
5. $\quad ReadFile\ \ g.v[i].m$
6. $\quad g.v[i].eptr = NULL$
7. $\quad for\ j = 1\ to\ m$
8. $\qquad p = GETNODE()$
9. $\qquad ReadFile\ \ p \to data$
10. $\qquad p \to link = NULL$
11. $\qquad if\ g.v[i].eptr =\ NULL)$
12. $\quad g.v[i].eptr = p$

13. $else$
14. $p \rightarrow link \ = \ g.v[i].eptr$
15. $g.v[i].eptr \ = \ p$
16. $Close \ the \ file$

$\boldsymbol{Algorithm \ DisplayGraph(g)}$

//To display the details of graph g

1. $print$ "$No. \quad Name \quad Adjacent \ Node \ List$ "
2. $for \ i = 1 \ to \ g.n$
3. $print \ i,$" " $, g.v[i].name$
4. $t = g.v[i].eptr$
5. $while \ t \neq NULL$
6. $print$ "$\rightarrow$" $, \ t \rightarrow data$
7. $t = t \rightarrow link$