Module 4 Trees

38. Code to find element in the Binary Search Tree  
struct node { int info; struct node \*left; struct node \*right; }\*root;  
void BST::find(int item, node \*\*par, node \*\*loc) { node \*ptr, \*ptrsave; if (root == NULL) { \*loc = NULL; \*par = NULL; return; } if (item == root->info) { \*loc = root; \*par = NULL; return;  
} if (item < root->info) ptr = root->left; else ptr = root->right; ptrsave = root; while (ptr != NULL) { if (item == ptr->info) { \*loc = ptr; \*par = ptrsave; return; } ptrsave = ptr; if (item < ptr->info) ptr = ptr->left; else ptr = ptr->right; } \*loc = NULL; \*par = ptrsave; }

39. Code to insert element into Binary Search Tree  
struct node { int info; struct node \*left; struct node \*right; }\*root;  
void BST::insert(node \*tree, node \*newnode) { if (root == NULL) { root = new node; root->info = newnode->info; root->left = NULL; root->right = NULL; cout<<"Root Node is Added"<<endl; return; } if (tree->info == newnode->info) { cout<<"Element already in the tree"<<endl; return; } if (tree->info > newnode->info) { if (tree->left != NULL) {  
insert(tree->left, newnode); } else { tree->left = newnode; (tree->left)->left = NULL; (tree->left)->right = NULL; cout<<"Node Added To Left"<<endl; return; } } else { if (tree->right != NULL) { insert(tree->right, newnode); } else { tree->right = newnode; (tree->right)->left = NULL; (tree->right)->right = NULL; cout<<"Node Added To Right"<<endl; return; } } }

40. Code for Preorder Traversal of Binary Search Tree  
void BST::preorder(node \*ptr) { if (root == NULL) { cout<<"Tree is empty"<<endl; return; } if (ptr != NULL) { cout<<ptr->info<<" "; preorder(ptr->left); preorder(ptr->right); } }

41. Code for Inorder Traversal of Binary Search Tree  
void BST::inorder(node \*ptr) { if (root == NULL) { cout<<"Tree is empty"<<endl; return;  
} if (ptr != NULL) { inorder(ptr->left); cout<<ptr->info<<" "; inorder(ptr->right); } }

42. Code for Postorder Traversal of Binary Search Tree  
void BST::postorder(node \*ptr) { if (root == NULL) { cout<<"Tree is empty"<<endl; return; } if (ptr != NULL) { postorder(ptr->left); postorder(ptr->right); cout<<ptr->info<<" "; } }

43. Code for Displaying Binary Search Tree Structure  
void BST::display(node \*ptr, int level) { int i; if (ptr != NULL) { display(ptr->right, level+1); cout<<endl; if (ptr == root) cout<<"Root->: "; else { for (i = 0;i < level;i++) cout<<" "; } cout<<ptr->info; display(ptr->left, level+1); } }

44. Code to create Circular Linked List  
struct node { int info; struct node \*next; }\*last;  
void circular\_llist::create\_node(int value) { struct node \*temp; temp = new(struct node); temp->info = value; if (last == NULL) { last = temp; temp->next = last; } else { temp->next = last->next; last->next = temp; last = temp; } }

45. Code to insert element in a Circular Linked List  
struct node { int info; struct node \*next; }\*last;  
void circular\_llist::add\_begin(int value) { if (last == NULL) { cout<<"First Create the list."<<endl; return; } struct node \*temp; temp = new(struct node); temp->info = value; temp->next = last->next; last->next = temp; }

46. Code to insert element at a particular place in a Circular Linked List  
struct node { int info; struct node \*next; }\*last;  
void circular\_llist::add\_after(int value, int pos) { if (last == NULL) { cout<<"First Create the list."<<endl; return; }  
struct node \*temp, \*s; s = last->next; for (int i = 0;i < pos-1;i++) { s = s->next; if (s == last->next) { cout<<"There are less than "; cout<<pos<<" in the list"<<endl; return; } } temp = new(struct node); temp->next = s->next; temp->info = value; s->next = temp; /\*Element inserted at the end\*/ if (s == last) { last=temp; } }

47. Code to delete element from a Circular Linked List struct node { int info; struct node \*next; }\*last;  
void circular\_llist::delete\_element(int value) { struct node \*temp, \*s; s = last->next; /\* If List has only one element\*/ if (last->next == last && last->info == value) { temp = last; last = NULL; free(temp); return; } if (s->info == value) /\*First Element Deletion\*/ { temp = s; last->next = s->next; free(temp); return; } while (s->next != last) { /\*Deletion of Element in between\*/  
if (s->next->info == value) { temp = s->next; s->next = temp->next; free(temp); cout<<"Element "<<value; cout<<" deleted from the list"<<endl; return; } s = s->next; } /\*Deletion of last element\*/ if (s->next->info == value) { temp = s->next; s->next = last->next; free(temp); last = s; return; } cout<<"Element "<<value<<" not found in the list"<<endl; }

48. Code to search element in a Circular Linked List struct node { int info; struct node \*next; }\*last;  
void circular\_llist::search\_element(int value) { struct node \*s; int counter = 0; s = last->next; while (s != last) { counter++; if (s->info == value) { cout<<"Element "<<value; cout<<" found at position "<<counter<<endl; return; } s = s->next; } if (s->info == value) { counter++; cout<<"Element "<<value; cout<<" found at position "<<counter<<endl; return;  
} cout<<"Element "<<value<<" not found in the list"<<endl; }

49. Code to display Circular Linked List struct node { int info; struct node \*next; }\*last;  
void circular\_llist::display\_list() { struct node \*s; if (last == NULL) { cout<<"List is empty, nothing to display"<<endl; return; } s = last->next; cout<<"Circular Link List: "<<endl; while (s != last) { cout<<s->info<<"->"; s = s->next; } cout<<s->info<<endl; }

50. Code to update Circular Linked List struct node { int info; struct node \*next; }\*last;  
void circular\_llist::update() { int value, pos, i; if (last == NULL) { cout<<"List is empty, nothing to update"<<endl; return; } cout<<"Enter the node position to be updated: "; cin>>pos; cout<<"Enter the new value: "; cin>>value; struct node \*s; s = last->next; for (i = 0;i < pos - 1;i++) { if (s == last)  
{ cout<<"There are less than "<<pos<<" elements."; cout<<endl; return; } s = s->next; } s->info = value; cout<<"Node Updated"<<endl; }

51. Code to sort Circular Linked List struct node { int info; struct node \*next; }\*last;  
void circular\_llist::sort() { struct node \*s, \*ptr; int temp; if (last == NULL) { cout<<"List is empty, nothing to sort"<<endl; return; } s = last->next; while (s != last) { ptr = s->next; while (ptr != last->next) { if (ptr != last->next) { if (s->info > ptr->info) { temp = s->info; s->info = ptr->info; ptr->info = temp; } } else break; ptr = ptr->next; } s = s->next; } }

52. Code to create a node in a singly linked list  
node \*single\_llist::create\_node(int value) { struct node \*temp, \*s; temp = new(struct node); if (temp == NULL) { cout<<"Memory not allocated "<<endl; return 0; } else { temp->info = value; temp->next = NULL; return temp; } }

53. Code to insert element at the beginning of a singly linked list  
void single\_llist::insert\_begin() { int value; cout<<"Enter the value to be inserted: "; cin>>value; struct node \*temp, \*p; temp = create\_node(value); if (start == NULL) { start = temp; start->next = NULL; } else { p = start; start = temp; start->next = p; } cout<<"Element Inserted at beginning"<<endl; }

54. Code to insert element at the last position in a singly linked list  
void single\_llist::insert\_last() { int value; cout<<"Enter the value to be inserted: "; cin>>value; struct node \*temp, \*s; temp = create\_node(value); s = start;  
while (s->next != NULL) { s = s->next; } temp->next = NULL; s->next = temp; cout<<"Element Inserted at last"<<endl; }

55. Code to insert element at a given position in a singly linked list  
void single\_llist::insert\_pos() { int value, pos, counter = 0; cout<<"Enter the value to be inserted: "; cin>>value; struct node \*temp, \*s, \*ptr; temp = create\_node(value); cout<<"Enter the postion at which node to be inserted: "; cin>>pos; int i; s = start; while (s != NULL) { s = s->next; counter++; } if (pos == 1) { if (start == NULL) { start = temp; start->next = NULL; } else { ptr = start; start = temp; start->next = ptr; } } else if (pos > 1 && pos <= counter) { s = start; for (i = 1; i < pos; i++) { ptr = s; s = s->next; } ptr->next = temp; temp->next = s;  
} else { cout<<"Positon out of range"<<endl; } }

56. Code to sort a singly linked list  
void single\_llist::sort() { struct node \*ptr, \*s; int value; if (start == NULL) { cout<<"The List is empty"<<endl; return; } ptr = start; while (ptr != NULL) { for (s = ptr->next;s !=NULL;s = s->next) { if (ptr->info > s->info) { value = ptr->info; ptr->info = s->info; s->info = value; } } ptr = ptr->next; } }

57. Code to delete node at a given position in a singly linked list  
void single\_llist::delete\_pos() { int pos, i, counter = 0; if (start == NULL) { cout<<"List is empty"<<endl; return; } cout<<"Enter the position of value to be deleted: "; cin>>pos; struct node \*s, \*ptr; s = start; if (pos == 1) { start = s->next; }  
else { while (s != NULL) { s = s->next; counter++; } if (pos > 0 && pos <= counter) { s = start; for (i = 1;i < pos;i++) { ptr = s; s = s->next; } ptr->next = s->next; } else { cout<<"Position out of range"<<endl; } free(s); cout<<"Element Deleted"<<endl; } }

58. Code to update a given node in a singly linked list  
void single\_llist::update() { int value, pos, i; if (start == NULL) { cout<<"List is empty"<<endl; return; } cout<<"Enter the node position to be updated: "; cin>>pos; cout<<"Enter the new value: "; cin>>value; struct node \*s, \*ptr; s = start; if (pos == 1) start->info = value; else { for (i = 0;i < pos - 1;i++) { if (s == NULL) { cout<<"There are less than "<<pos<<" elements"; return;  
} s = s->next; } s->info = value; } cout<<"Node Updated"<<endl; }

59. Code to search an element in a singly linked list  
void single\_llist::search() { int value, pos = 0; bool flag = false; if (start == NULL) { cout<<"List is empty"<<endl; return; } cout<<"Enter the value to be searched: "; cin>>value; struct node \*s; s = start; while (s != NULL) { pos++; if (s->info == value) { flag = true; cout<<"Element "<<value<<" is found at position "<<pos<<endl; } s = s->next; } if (!flag) cout<<"Element "<<value<<" not found in the list"<<endl; }

60. Code to reverse a singly linked list  
void single\_llist::reverse() { struct node \*ptr1, \*ptr2, \*ptr3; if (start == NULL) { cout<<"List is empty"<<endl; return; } if (start->next == NULL) return; ptr1 = start; ptr2 = ptr1->next; ptr3 = ptr2->next; ptr1->next = NULL;  
ptr2->next = ptr1; while (ptr3 != NULL) { ptr1 = ptr2; ptr2 = ptr3; ptr3 = ptr3->next; ptr2->next = ptr1; } start = ptr2; }

68. Program to Check Binary Tree is Binary Search Tree  
struct node { int data; node\* left; node\* right; }; int isBSTUtil(node\* node, int min, int max); int isBST(node\* node) return(isBSTUtil(node, INT\_MIN, INT\_MAX)); int isBSTUtil(struct node\* node, int min, int max) { if (node==NULL) return 1; if (node->data < min || node->data > max) return 0; return isBSTUtil(node->left, min, node->data - 1) && isBSTUtil(node->right, node->data + 1, max); } node\* newNode(int data) { node\* nod = new node; nod->data = data; nod->left = NULL; nod->right = NULL; return nod; } int main() { node \*root = newNode(4); root->left = newNode(2); root->right = newNode(5); root->left->left = newNode(1); root->left->right = newNode(3); if (isBST(root)) cout<<"The Given Binary Tree is a BST"<<endl;  
else cout<<"The Given Binary Tree is not a BST"<<endl; return 0; }

69. Code for Counting the total nodes in a tree  
int countnode(treeptr root) { static int count=0; treeptr temp=root; if(temp!= NULL) { count ++; countnode(temp->left); countnode(temp->right); } return count; }

70. Code for Mirroring a given tree  
void mirror(treeptr root) { treeptr temp=root, temp1; if(temp) { if(temp->left) mirror(temp->left); if(temp->right) mirror(temp->right); /\* interchange \*/ temp1=temp->left; temp->left=temp->right; temp->right=temp1; } }

71. Code for Comparing two binary search trees  
int compare(treeptr root1, treeptr root2) { static int equal=0; if(root1==NULL && root2==NULL) return1; else if(root1!=NULL && root2!=NULL) if(root1 -> data == root2 -> data) if(compare(root1 -> left, root2 -> left)) equal = compare(root1 -> right,root2->right); else equal=0; return(equal); }

72. Code for Copying a tree  
treeptr treecopy(treeptr root) { treeptr newnode; if(root!= NULL) { newnode=nodealloc;  
newnode-> left = treecopy(root->left); newnode-> right = treecopy(root->right); newnode-> data = root->left; return(newnode); } else return NULL;  
}

73. Program for Breadth First Search #include<iostream.h> #include<conio.h> #include<stdlib.h> int cost[10][10], i, j, k, n, queue[10], front, rear, v, visit[10], visited[10]; void main() { int m; clrscr(); cout <<"enter no of vertices"; cin >> n; cout <<"enter no of edges"; cin >> m; cout <<"\n EDGES \n"; for(k=1;k<=m;k++) { cin >>i>>j; cost[i][j]=1; } cout <<"enter initial vertex"; cin >>v; cout <<"Visited vertices\n"; cout << v; visited[v]=1; k=1; while(k<n)  
{ for(j=1;j<=n;j++) if(cost[v][j]!=0 && visited[j]!=1 && visit[j]!=1) { visit[j]=1; queue[rear++]=j; } v=queue[front++]; cout<<v << " ";  
k++; visit[v]=0; visited[v]=1; }  
getch(); }

74. Program for Depth First Search #include<iostream.h> #include<conio.h> #include<stdlib.h> int cost[10][10],i,j,k,n,stack[10],top,v,visit[10],visited[10]; void main() { int m; cout <<"enter no of vertices"; cin >> n; cout <<"enter no of edges"; cin >> m; cout <<"\n EDGES \n";  
for(k=1;k<=m;k++) { cin>>i>>j; cost[i][j]=1; } cout <<"enter initial vertex"; cin >>v; cout <<"ORDER OF VISITED VERTICES"; cout << v <<" "; visited[v]=1; k=1; while(k<n) { for(j=n;j>=1;j--) if(cost[v][j]!=0 && visited[j]!=1 && visit[j]!=1) { visit[j]=1; stack [top]=j; top++; } v= stack[--top]; cout<<v << " "; k++; visit[v]=0; visited[v]=1; } getch(); }

75. Code for search an element in Binary Threaded Tree  
bool search(int key) { Node \*tmp = root->left;  
for (;;) { if (tmp->key < key) { if (tmp->rightThread) return false; tmp = tmp->right; } else if (tmp->key > key) { if (tmp->leftThread) return false; tmp = tmp->left; } else { return true; } } }

76. Code to Print Binary Threaded Tree  
void printTree() { Node \*tmp = root, \*p; for (;;) { p = tmp; tmp = tmp->right; if (!p->rightThread) { while (!tmp->leftThread) { tmp = tmp->left; } } if (tmp == root) break; cout<<tmp->key<<" "; } cout<<endl; }

77. Code to traverse B+ tree  
void traverse(B+TreeNode \*p) { cout<<endl; int i; for (i = 0; i < p->n; i++) { if (p->leaf == false)  
traverse(p->child\_ptr[i]); cout << " " << p->data[i]; } if (p->leaf == false) traverse(p->child\_ptr[i]); cout<<endl; }

78. Code to Sort B+ Tree  
void sort(int \*p, int n) { int i, j, temp; for (i = 0; i < n; i++) { for (j = i; j <= n; j++) { if (p[i] > p[j]) { temp = p[i]; p[i] = p[j]; p[j] = temp; } } } }

79. Code to find the Height of AVL Tree  
int avlTree::height(avl\_node \*temp) { int h = 0; if (temp != NULL) { int l\_height = height (temp->left); int r\_height = height (temp->right); int max\_height = max (l\_height, r\_height); h = max\_height + 1; } return h; }

80. Code to find Height Difference in AVL Tree  
int avlTree::diff(avl\_node \*temp) { int l\_height = height (temp->left); int r\_height = height (temp->right); int b\_factor= l\_height - r\_height; return b\_factor; }

81. Code for balancing AVL Tree  
avl\_node \*avlTree::balance(avl\_node \*temp) { int bal\_factor = diff (temp); if (bal\_factor > 1) { if (diff (temp->left) > 0) temp = ll\_rotation (temp); else temp = lr\_rotation (temp); } else if (bal\_factor < -1) { if (diff (temp->right) > 0) temp = rl\_rotation (temp); else temp = rr\_rotation (temp); } return temp; }

82. Code to Insert an element in AVL Tree  
avl\_node \*avlTree::insert(avl\_node \*root, int value) { if (root == NULL) { root = new avl\_node; root->data = value; root->left = NULL; root->right = NULL; return root; } else if (value < root->data) { root->left = insert(root->left, value); root = balance (root); } else if (value >= root->data) { root->right = insert(root->right, value); root = balance (root); } return root; }

83. Code to display AVL Tree  
void avlTree::display(avl\_node \*ptr, int level) { int i; if (ptr!=NULL) { display(ptr->right, level + 1);  
printf("\n"); if (ptr == root) cout<<"Root -> "; for (i = 0; i < level && ptr != root; i++) cout<<" "; cout<<ptr->data; display(ptr->left, level + 1); } }

84. Code for InOrder Traversal of AVL Tree  
void avlTree::inorder(avl\_node \*tree) { if (tree == NULL) return; inorder (tree->left); cout<<tree->data<<" "; inorder (tree->right); }

85. Code for PreOrder Traversal of AVL Tree  
void avlTree::preorder(avl\_node \*tree) { if (tree == NULL) return; cout<<tree->data<<" "; preorder (tree->left); preorder (tree->right); }

86. Code for PostOrder Traversal of AVL Tree  
void avlTree::postorder(avl\_node \*tree) { if (tree == NULL) return; postorder ( tree ->left ); postorder ( tree ->right ); cout<<tree->data<<" "; }

87. Code for Right-Right Rotation (RR) of AVL Tree  
avl\_node \*avlTree::rr\_rotation(avl\_node \*parent) { avl\_node \*temp; temp = parent->right; parent->right = temp->left; temp->left = parent; return temp; }

88. Code for Left-Left Rotation (LL) of AVL Tree  
avl\_node \*avlTree::ll\_rotation(avl\_node \*parent) { avl\_node \*temp; temp = parent->left; parent->left = temp->right; temp->right = parent; return temp; }

89. Code for Left-Right Rotation (LR) of AVL Tree  
avl\_node \*avlTree::lr\_rotation(avl\_node \*parent) { avl\_node \*temp; temp = parent->left; parent->left = rr\_rotation (temp); return ll\_rotation (parent); }

90. Code for Right-Left Rotation (RL) of AVL Tree  
avl\_node \*avlTree::rl\_rotation(avl\_node \*parent) { avl\_node \*temp; temp = parent->right; parent->right = ll\_rotation (temp); return rr\_rotation (parent); }