23CSE235 ADVANCED DATA STRUCTURES & ALGORITHM ANALYSIS LAB

1.Construct an AVL tree for a given set of elements which are stored in a file. And implement insert and delete operation on the constructed tree. Write contents of tree into a new file using in-order

PROGRAM

```
#include <stdio.h>
#include <stdlib.h>
// AVL tree node structure
struct Node {
int key;
struct Node* left;
struct Node* right;
int height;
};
// Function to get the height of the tree
int height(struct Node* N) {
if (N == NULL)
return 0;
return N->height;
}
// Function to get the maximum of two integers
int max(int a, int b) {
return (a > b) ? a : b;
}
// Function to create a new AVL node
struct Node* newNode(int key) {
struct Node* node = (struct Node*)malloc(sizeof(struct Node));
node->key = key;
node->left = NULL;
node->right = NULL;
node->height = 1;  // New node is initially added at leaf
return(node);
}
// Right rotate subtree rooted with y
struct Node* rightRotate(struct Node* y) {
struct Node* x = y->left;
struct Node* T2 = x->right;
   // Perform rotation
x->right = y;
y->left = T2;
   // Update heights
y->height = max(height(y->left), height(y->right)) + 1;
x->height = max(height(x->left), height(x->right)) + 1;
   // Return new root
return x;
}
// Left rotate subtree rooted with x
struct Node* leftRotate(struct Node* x) {
struct Node* y = x->right;
```

```
struct Node* T2 = y->left;
   // Perform rotation
y->left = x;
x->right = T2;
   // Update heights
x->height = max(height(x->left), height(x->right)) + 1;
y->height = max(height(y->left), height(y->right)) + 1;
   // Return new root
return y;
}
// Get the balance factor of a node
intgetBalance(struct Node* N) {
if (N == NULL)
return 0;
return height(N->left) - height(N->right);
}
// Recursive function to insert a key in the subtree rooted with node and returns the new root
of the subtree
struct Node* insert(struct Node* node, int key) {
   // 1. Perform the normal BST insertion
if (node == NULL)
return(newNode(key));
if (key < node->key)
node->left = insert(node->left, key);
else if (key > node->key)
node->right = insert(node->right, key);
else  // Equal keys are not allowed in AVL trees
return node;
   // 2. Update height of this ancestor node
node->height = 1 + max(height(node->left), height(node->right));
   // 3. Get the balance factor of this ancestor node to check whether this node became
unbalanced
int balance = getBalance(node);
   // If this node becomes unbalanced, then there are 4 cases
   // Left Left Case
if (balance > 1 && key < node->left->key)
returnrightRotate(node);
   // Right Right Case
if (balance < -1 && key > node->right->key)
returnleftRotate(node);
   // Left Right Case
if (balance > 1 && key > node->left->key) {
node->left = leftRotate(node->left);
returnrightRotate(node);
   }
   // Right Left Case
if (balance < -1 && key < node->right->key) {
node->right = rightRotate(node->right);
returnleftRotate(node);
   }
```

```
    // Return the (unchanged) node pointer
return node;
}
// Function to find the node with the minimum key value
struct Node* minValueNode(struct Node* node) {
struct Node* current = node;
    // Loop down to find the leftmost leaf
while (current->left != NULL)
current = current->left;
return current;
}
// Recursive function to delete a node with given key
struct Node* deleteNode(struct Node* root, int key) {
    // STEP 1: PERFORM STANDARD BST DELETE
if (root == NULL)
return root;
    // If the key to be deleted is smaller than the root's key, then it lies in left subtree
if (key < root->key)
root->left = deleteNode(root->left, key);
    // If the key to be deleted is greater than the root's key, then it lies in right subtree
else if (key > root->key)
root->right = deleteNode(root->right, key);
    // If key is same as root's key, then this is the node to be deleted
else {
        // Node with only one child or no child
if ((root->left == NULL) || (root->right == NULL)) {
struct Node* temp = root->left ? root->left : root->right;
            // No child case
if (temp == NULL) {
temp = root;
root = NULL;
            }
else  // One child case
                *root = *temp;  // Copy the contents of the non-empty child
free(temp);
        }
else {
            // Node with two children: Get the inorder successor (smallest in the right subtree)
struct Node* temp = minValueNode(root->right);
            // Copy the inorder successor's data to this node
root->key = temp->key;
            // Delete the inorder successor
root->right = deleteNode(root->right, temp->key);
        }
    }
    // If the tree had only one node then return
if (root == NULL)
return root;
    // STEP 2: UPDATE HEIGHT OF THE CURRENT NODE
root->height = max(height(root->left), height(root->right)) + 1;
```

```
   // STEP 3: GET THE BALANCE FACTOR OF THIS NODE (to check whether this node
became unbalanced)
int balance = getBalance(root);
   // If this node becomes unbalanced, then there are 4 cases
   // Left Left Case
if (balance > 1 &&getBalance(root->left) >= 0)
returnrightRotate(root);
   // Left Right Case
if (balance > 1 &&getBalance(root->left) < 0) {
root->left = leftRotate(root->left);
returnrightRotate(root);
   }
   // Right Right Case
if (balance < -1 &&getBalance(root->right) <= 0)
returnleftRotate(root);

   // Right Left Case
if (balance < -1 &&getBalance(root->right) > 0) {
root->right = rightRotate(root->right);
returnleftRotate(root);
   }
return root;
}
// Inorder traversal to write the nodes to a file
voidinOrderToFile(struct Node* root, FILE* file) {
if (root != NULL) {
inOrderToFile(root->left, file);
fprintf(file, "%d ", root->key);
inOrderToFile(root->right, file);
   }
}
int main() {
struct Node* root = NULL;
   FILE* file;
   // Inserting nodes
root = insert(root, 10);
root = insert(root, 20);
root = insert(root, 30);
root = insert(root, 40);
root = insert(root, 50);
root = insert(root, 25);
   // Deleting a node
root = deleteNode(root, 40);

   // Writing tree contents to a file in in-order traversal
file = fopen("avl_tree_output.txt", "w");
if (file == NULL) {
printf("Error opening file!\n");
return 1;
   }
```

```
inOrderToFile(root, file);
fclose(file);
printf("AVL Tree written to file (In-order traversal)\n");
return 0;
}
```

OUTPUT

Insert the following keys in order: 10, 20, 30, 40, 50, and 25.

1. **Insert 10**: Tree becomes:

```
10
```

2. **Insert 20**: Tree becomes:

```
  10
    \
     20
```

3. **Insert 30**: Tree becomes unbalanced (Right Right case), so perform left rotation:

```
   20
  /  \
 10   30
```

4. **Insert 40**: Tree becomes:

```
   20
  /  \
 10   30
        \
         40
```

5. **Insert 50**: Tree becomes unbalanced (Right Right case), so perform left rotation:

```
    30
   /  \
  20   40
 /       \
10        50
```

6. Insert 25: Tree becomes unbalanced (Right Left case), so perform right-left rotation:

```
    30
   /  \
  20   40
 /  \    \
10  25   50
```

Deletion Example

Delete 40: Node 40 has one child (50), so replace it with 50:

Copy code

```
    30
   /  \
  20   50
 /  \
10  25
```

In-order Traversal:

- After insertion and deletion, the AVL tree in in-order traversal (left-root-right) is: `10 20 25 30 50`.

Output File (`avl_tree_output.txt`):
The contents of the file after the in-order traversal will be:
10 20 25 30 50
This is a sorted sequence of the tree's nodes as per in-order traversal.

2. Construct B-Tree an order of 5 with a set of 100 random elements stored in array. Implement searching, insertion and deletion operations.

B-TREE INSERTION OF ORDER 5

```
// Insertioning a key on a B-tree in C
#include <stdio.h>
#include <stdlib.h>
#define MAX 3
#define MIN 2
struct btreeNode {
int item[MAX + 1], count;
struct btreeNode *link[MAX + 1];
};
struct btreeNode *root;
// Node creation
struct btreeNode *createNode(int item, struct btreeNode *child) {
struct btreeNode *newNode;
newNode = (struct btreeNode *)malloc(sizeof(struct btreeNode));
newNode->item[1] = item;
newNode->count = 1;
newNode->link[0] = root;
newNode->link[1] = child;
return newNode;
}
void insertValue(int item, int pos, struct btreeNode *node,
struct btreeNode *child) {
int j = node->count;
while (j > pos) {
node->item[j + 1] = node->item[j];
node->link[j + 1] = node->link[j];
   j--;
  }
  node->item[j + 1] = item;
  node->link[j + 1] = child;
  node->count++;
}
// Split node
void splitNode(int item, int *pval, int pos, struct btreeNode *node,
        struct btreeNode *child, struct btreeNode **newNode) {
  int median, j;
 if (pos > MIN)
   median = MIN + 1;
```

```
  else
    median = MIN;
  *newNode = (struct btreeNode *)malloc(sizeof(struct btreeNode));
  j = median + 1;
  while (j <= MAX) {
    (*newNode)->item[j - median] = node->item[j];
    (*newNode)->link[j - median] = node->link[j];
    j++;
  }
  node->count = median;
  (*newNode)->count = MAX - median;
if (pos <= MIN) {
    insertValue(item, pos, node, child);
  } else {
    insertValue(item, pos - median, *newNode, child);
  }
  *pval = node->item[node->count];
  (*newNode)->link[0] = node->link[node->count];
  node->count--;
}
// Set the value of node
int setNodeValue(int item, int *pval,
          struct btreeNode *node, struct btreeNode **child) {
  int pos;
  if (!node) {
    *pval = item;
    *child = NULL;
    return 1;
  }
if (item < node->item[1]) {
    pos = 0;
  } else {
    for (pos = node->count;
       (item < node->item[pos] && pos > 1); pos--);
    if (item == node->item[pos]) {
      printf("Duplicates not allowed\n");
      return 0;
    }
  }
  if (setNodeValue(item, pval, node->link[pos], child)) {
    if (node->count < MAX) {
      insertValue(*pval, pos, node, *child);
} else {
splitNode(*pval, pval, pos, node, *child, child);
return 1;
}
}
return 0;
}
// Insert the value
```

```
void insertion(int item) {
int flag, i;
struct btreeNode *child;
flag = setNodeValue(item, &i, root, &child);
if (flag)
root = createNode(i, child);
}
// Copy the successor
void copySuccessor(struct btreeNode *myNode, int pos) {
struct btreeNode *dummy;
dummy = myNode->link[pos];
for (; dummy->link[0] != NULL;)
dummy = dummy->link[0];
myNode->item[pos] = dummy->item[1];
}
// Do rightshift
void rightShift(struct btreeNode *myNode, int pos) {
struct btreeNode *x = myNode->link[pos];
int j = x->count;
while (j > 0) {
x->item[j + 1] = x->item[j];
x->link[j + 1] = x->link[j];
}
x->item[1] = myNode->item[pos];
x->link[1] = x->link[0];
x->count++;
x = myNode->link[pos - 1];
myNode->item[pos] = x->item[x->count];
myNode->link[pos] = x->link[x->count];
x->count--;
return;
}
// Do leftshift
void leftShift(struct btreeNode *myNode, int pos) {
int j = 1;
struct btreeNode *x = myNode->link[pos - 1];
x->count++;
x->item[x->count] = myNode->item[pos];
x->link[x->count] = myNode->link[pos]->link[0];
x = myNode->link[pos];
myNode->item[pos] = x->item[1];
x->link[0] = x->link[1];
x->count--;
while (j <= x->count) {
x->item[j] = x->item[j + 1];
    x->link[j] = x->link[j + 1];
    j++;
  }
  return;
}
```

```
// Merge the nodes
void mergeNodes(struct btreeNode *myNode, int pos) {
  int j = 1;
struct btreeNode *x1 = myNode->link[pos], *x2 = myNode->link[pos - 1];
 x2->count++;
  x2->item[x2->count] = myNode->item[pos];
  x2->link[x2->count] = myNode->link[0];
while (j <= x1->count) {
    x2->count++;
    x2->item[x2->count] = x1->item[j];
    x2->link[x2->count] = x1->link[j];
    j++;
  }
 j = pos;
  while (j < myNode->count) {
    myNode->item[j] = myNode->item[j + 1];
    myNode->link[j] = myNode->link[j + 1];
    j++;
  }
  myNode->count--;
  free(x1);
}
// Adjust the node
void adjustNode(struct btreeNode *myNode, int pos) {
  if (!pos) {
    if (myNode->link[1]->count > MIN) {
      leftShift(myNode, 1);
    } else {
      mergeNodes(myNode, 1);
    }
  } else {
    if (myNode->count != pos) {
      if (myNode->link[pos - 1]->count > MIN) {
        rightShift(myNode, pos);
      } else {
        if (myNode->link[pos + 1]->count > MIN) {
          leftShift(myNode, pos + 1);
        } else {
          mergeNodes(myNode, pos);
        }
      }
    } else {
      if (myNode->link[pos - 1]->count > MIN)
        rightShift(myNode, pos);
      else
        mergeNodes(myNode, pos);
    }
  }
}
// Traverse the tree
```

```
void traversal(struct btreeNode *myNode) {
  int i;
  if (myNode) {
    for (i = 0; i < myNode->count; i++) {
      traversal(myNode->link[i]);
      printf("%d ", myNode->item[i + 1]);
    }
    traversal(myNode->link[i]);
  }
}
int main() {
  int item, ch;
 insertion(8);
  insertion(9);
  insertion(10);
  insertion(11);
  insertion(15);
  insertion(16);
  insertion(17);
  insertion(18);
  insertion(20);
  insertion(23);
traversal(root);
}
```

OUTPUT

```
8 9 10 11 15 16 17 18 20 23
Process returned 0 (0x0)   execution time : 0.906 s
Press any key to continue.
```

BTREE DELETION OF ORDER 5

```
// Deleting a key from a B-tree in C
#include <stdio.h>
#include <stdlib.h>
#define MAX 3
#define MIN 2
struct BTreeNode {
int item[MAX + 1], count;
struct BTreeNode *linker[MAX + 1];
};
struct BTreeNode *root;
// Node creation
struct BTreeNode *createNode(int item, struct BTreeNode *child) {
struct BTreeNode *newNode;
newNode = (struct BTreeNode *)malloc(sizeof(struct BTreeNode));
newNode->item[1] = item;
```

```
newNode->count = 1;
newNode->linker[0] = root;
newNode->linker[1] = child;
return newNode;
}
// Add value to the node
void addValToNode(int item, int pos, struct BTreeNode *node,
struct BTreeNode *child) {
int j = node->count;
while (j > pos) {
node->item[j + 1] = node->item[j];
   node->linker[j + 1] = node->linker[j];
   j--;
  }
  node->item[j + 1] = item;
  node->linker[j + 1] = child;
  node->count++;
}
// Split the node
void splitNode(int item, int *pval, int pos, struct BTreeNode *node,
        struct BTreeNode *child, struct BTreeNode **newNode) {
  int median, j;
if (pos > MIN)
   median = MIN + 1;
  else
   median = MIN;
  *newNode = (struct BTreeNode *)malloc(sizeof(struct BTreeNode));
  j = median + 1;
  while (j <= MAX) {
   (*newNode)->item[j - median] = node->item[j];
   (*newNode)->linker[j - median] = node->linker[j];
   j++;
  }
  node->count = median;
  (*newNode)->count = MAX - median;
  if (pos <= MIN) {
   addValToNode(item, pos, node, child);
  } else {
   addValToNode(item, pos - median, *newNode, child);
  }
  *pval = node->item[node->count];
  (*newNode)->linker[0] = node->linker[node->count];
  node->count--;
}
// Set the value in the node
int setValueInNode(int item, int *pval,
          struct BTreeNode *node, struct BTreeNode **child) {
  int pos;
  if (!node) {
   *pval = item;
```

```
    *child = NULL;
    return 1;
  }
if (item < node->item[1]) {
    pos = 0;
  } else {
    for (pos = node->count;
       (item < node->item[pos] && pos > 1); pos--);
    if (item == node->item[pos]) {
      printf("Duplicates not allowed\n");
      return 0;
    }
  }
  if (setValueInNode(item, pval, node->linker[pos], child)) {
    if (node->count < MAX) {
      addValToNode(*pval, pos, node, *child);
    } else {
      splitNode(*pval, pval, pos, node, *child, child);
      return 1;
    }
  }
  return 0;
}
// Insertion operation
void insertion(int item) {
  int flag, i;
  struct BTreeNode *child;
 flag = setValueInNode(item, &i, root, &child);
  if (flag)
    root = createNode(i, child);
}
// Copy the successor
void copySuccessor(struct BTreeNode *myNode, int pos) {
  struct BTreeNode *dummy;
  dummy = myNode->linker[pos];
for (; dummy->linker[0] != NULL;)
    dummy = dummy->linker[0];
  myNode->item[pos] = dummy->item[1];
}
// Remove the value
void removeVal(struct BTreeNode *myNode, int pos) {
  int i = pos + 1;
while (i <= myNode->count) {
myNode->item[i - 1] = myNode->item[i];
myNode->linker[i - 1] = myNode->linker[i];
i++;
}
myNode->count--;
}
// Do right shift
```

```
void rightShift(struct BTreeNode *myNode, int pos) {
struct BTreeNode *x = myNode->linker[pos];
int j = x->count;
while (j > 0) {
x->item[j + 1] = x->item[j];
x->linker[j + 1] = x->linker[j];
}
x->item[1] = myNode->item[pos];
x->linker[1] = x->linker[0];
x->count++;
x = myNode->linker[pos - 1];
myNode->item[pos] = x->item[x->count];
myNode->linker[pos] = x->linker[x->count];
x->count--;
return;
}
// Do left shift
void leftShift(struct BTreeNode *myNode, int pos) {
int j = 1;
  struct BTreeNode *x = myNode->linker[pos - 1];
 x->count++;
  x->item[x->count] = myNode->item[pos];
  x->linker[x->count] = myNode->linker[pos]->linker[0];
x = myNode->linker[pos];
  myNode->item[pos] = x->item[1];
  x->linker[0] = x->linker[1];
  x->count--;
while (j <= x->count) {
    x->item[j] = x->item[j + 1];
    x->linker[j] = x->linker[j + 1];
    j++;
  }
  return;
}
// Merge the nodes
void mergeNodes(struct BTreeNode *myNode, int pos) {
  int j = 1;
  struct BTreeNode *x1 = myNode->linker[pos], *x2 = myNode->linker[pos - 1];
x2->count++;
  x2->item[x2->count] = myNode->item[pos];
  x2->linker[x2->count] = myNode->linker[0];
while (j <= x1->count) {
    x2->count++;
    x2->item[x2->count] = x1->item[j];
    x2->linker[x2->count] = x1->linker[j];
    j++;
  }
 j = pos;
  while (j < myNode->count) {
    myNode->item[j] = myNode->item[j + 1];
```

```
    myNode->linker[j] = myNode->linker[j + 1];
    j++;
  }
  myNode->count--;
  free(x1);
}
// Adjust the node
void adjustNode(struct BTreeNode *myNode, int pos) {
  if (!pos) {
    if (myNode->linker[1]->count > MIN) {
      leftShift(myNode, 1);
    } else {
      mergeNodes(myNode, 1);
    }
  } else {
    if (myNode->count != pos) {
      if (myNode->linker[pos - 1]->count > MIN) {
        rightShift(myNode, pos);
      } else {
        if (myNode->linker[pos + 1]->count > MIN) {
          leftShift(myNode, pos + 1);
        } else {
          mergeNodes(myNode, pos);
        }
      }
    } else {
      if (myNode->linker[pos - 1]->count > MIN)
        rightShift(myNode, pos);
      else
        mergeNodes(myNode, pos);
    }
  }
}
// Delete a value from the node
int delValFromNode(int item, struct BTreeNode *myNode) {
  int pos, flag = 0;
  if (myNode) {
    if (item < myNode->item[1]) {
      pos = 0;
      flag = 0;
    } else {
      for (pos = myNode->count; (item < myNode->item[pos] && pos > 1); pos--);
      if (item == myNode->item[pos]) {
        flag = 1;
      } else {
        flag = 0;
      }
    }
    if (flag) {
      if (myNode->linker[pos - 1]) {
```

```
      copySuccessor(myNode, pos);
      flag = delValFromNode(myNode->item[pos], myNode->linker[pos]);
      if (flag == 0) {
        printf("Given data is not present in B-Tree\n");
      }
    } else {
      removeVal(myNode, pos);
    }
  } else {
    flag = delValFromNode(item, myNode->linker[pos]);
  }
  if (myNode->linker[pos]) {
    if (myNode->linker[pos]->count < MIN)
      adjustNode(myNode, pos);
  }
 }
 return flag;
}
// Delete operaiton
void delete (int item, struct BTreeNode *myNode) {
 struct BTreeNode *tmp;
 if (!delValFromNode(item, myNode)) {
  printf("Not present\n");
  return;
 } else {
  if (myNode->count == 0) {
    tmp = myNode;
    myNode = myNode->linker[0];
    free(tmp);
  }
 }
 root = myNode;
 return;
}
void searching(int item, int *pos, struct BTreeNode *myNode) {
 if (!myNode) {
  return;
 }
if (item < myNode->item[1]) {
  *pos = 0;
 } else {
  for (*pos = myNode->count;
    (item < myNode->item[*pos] && *pos > 1); (*pos)--)
    ;
  if (item == myNode->item[*pos]) {
    printf("%d present in B-tree", item);
    return;
  }
 }
 searching(item, pos, myNode->linker[*pos]);
```

```
  return;
}
void traversal(struct BTreeNode *myNode) {
  int i;
  if (myNode) {
    for (i = 0; i < myNode->count; i++) {
      traversal(myNode->linker[i]);
      printf("%d ", myNode->item[i + 1]);
    }
    traversal(myNode->linker[i]);
  }
}
int main() {
  int item, ch;
 insertion(8);
  insertion(9);
  insertion(10);
  insertion(11);
  insertion(15);
  insertion(16);
  insertion(17);
  insertion(18);
  insertion(20);
  insertion(23);
 traversal(root);
delete (20, root);
  printf("\n");
  traversal(root);
}
```

OUTPUT

```
8 9 10 11 15 16 17 18 20 23
8 9 10 11 15 16 17 18 23 8 9
Process returned 0 (0x0)   execution time : 0.078 s
Press any key to continue.
```

3.Construct Min and Max Heap using arrays, delete any element and display the content of the Heap

MIN HEAP INSERTION AND DELETION

```
#include <stdio.h>
#include <stdlib.h>
// Declare a heap structure
struct Heap {
    int* arr;
    int size;
    int capacity;
};
// define the struct Heap name
typedef struct Heap heap;
// forward declarations
heap* createHeap(int capacity, int* nums);
void insertHelper(heap* h, int index);
void heapify(heap* h, int index);
int extractMin(heap* h);
void insert(heap* h, int data);
// Define a createHeap function
heap* createHeap(int capacity, int* nums)
{
    // Allocating memory to heap h
    heap* h = (heap*)malloc(sizeof(heap));
    // Checking if memory is allocated to h or not
    if (h == NULL) {
        printf("Memory error");
        return NULL;
```

```
    }
    // set the values to size and capacity
    h->size = 0;
    h->capacity = capacity;
    // Allocating memory to array
    h->arr = (int*)malloc(capacity * sizeof(int));
    // Checking if memory is allocated to h or not
    if (h->arr == NULL) {
        printf("Memory error");
        return NULL;
    }
    int i;
    for (i = 0; i < capacity; i++) {
        h->arr[i] = nums[i];
    }
    h->size = i;
    i = (h->size - 2) / 2;
    while (i >= 0) {
        heapify(h, i);
        i--;
    }
    return h;
}
// Defining insertHelper function
void insertHelper(heap* h, int index)
{
    // Store parent of element at index
    // in parent variable
    int parent = (index - 1) / 2;
    if (h->arr[parent] > h->arr[index]) {
```

```
        // Swapping when child is smaller
        // than parent element
        int temp = h->arr[parent];
        h->arr[parent] = h->arr[index];
        h->arr[index] = temp;
        // Recursively calling insertHelper
        insertHelper(h, parent);
    }
}
void heapify(heap* h, int index)
{
    int left = index * 2 + 1;
    int right = index * 2 + 2;
    int min = index;
    // Checking whether our left or child element
    // is at right index or not to avoid index error
    if (left >= h->size || left < 0)
        left = -1;
    if (right >= h->size || right < 0)
        right = -1;
    // store left or right element in min if
    // any of these is smaller that its parent
    if (left != -1 && h->arr[left] < h->arr[index])
        min = left;
    if (right != -1 && h->arr[right] < h->arr[min])
        min = right;
    // Swapping the nodes
    if (min != index) {
        int temp = h->arr[min];
        h->arr[min] = h->arr[index];
```

```
        h->arr[index] = temp;
        // recursively calling for their child elements
        // to maintain min heap
        heapify(h, min);
    }
}
int extractMin(heap* h)
{
    int deleteItem;
    // Checking if the heap is empty or not
    if (h->size == 0) {
        printf("\nHeap id empty.");
        return -999;
    }
    // Store the node in deleteItem that
    // is to be deleted.
    deleteItem = h->arr[0];
    // Replace the deleted node with the last node
    h->arr[0] = h->arr[h->size - 1];
    // Decrement the size of heap
    h->size--;
    // Call minheapify_top_down for 0th index
    // to maintain the heap property
    heapify(h, 0);
    return deleteItem;
}
// Define a insert function
void insert(heap* h, int data)
{
    // Checking if heap is full or not
```

```
    if (h->size < h->capacity) {
        // Inserting data into an array
        h->arr[h->size] = data;
        // Calling insertHelper function
        insertHelper(h, h->size);
        // Incrementing size of array
        h->size++;
    }
}
void printHeap(heap* h)
{
    int i;
    for (i = 0; i < h->size; i++) {
        printf("%d ", h->arr[i]);
    }
    printf("\n");
}
int main()
{
    int arr[9] = { 9, 8, 7, 6, 5, 4, 3, 2, 1 };
    heap* hp = createHeap(9, arr);
    printHeap(hp);
    extractMin(hp);
    printHeap(hp);
    return 0;
}
```

OUTPUT

```
1 2 3 6 5 4 7 8 9
2 5 3 6 9 4 7 8
Process returned 0 (0x0)   execution time : 0.737 s
```

Press any key to continue

MAX HEAP INSERTION

```
#include <malloc.h>
#include <stdio.h>
// Declare a heap structure
struct Heap {
    int* arr;
    int size;
    int capacity;
};
// define the struct Heap name
typedef struct Heap heap;
// forward declarations
heap* createHeap(int capacity, int* nums);
void insertHelper(heap* h, int index);
void maxHeapify(heap* h, int index);
int extractMax(heap* h);
void insert(heap* h, int data);
// Define a createHeap function
heap* createHeap(int capacity, int* nums)
{
    // Allocating memory to heap h
    heap* h = (heap*)malloc(sizeof(heap));
    // Checking if memory is allocated to h or not
    if (h == NULL) {
        printf("Memory error");
        return NULL;
    }
    // set the values to size and capacity
    h->size = 0;
    h->capacity = capacity;
    // Allocating memory to array
    h->arr = (int*)malloc(capacity * sizeof(int));
```

```
    // Checking if memory is allocated to h or not
    if (h->arr == NULL) {
        printf("Memory error");
        return NULL;
    }
    int i;
    for (i = 0; i < capacity; i++) {
        h->arr[i] = nums[i];
    }
    h->size = i;
    i = (h->size - 2) / 2;
    while (i >= 0) {
        maxHeapify(h, i);
        i--;
    }
    return h;
}
// Defining maxHeapify_bottom_up function
void insertHelper(heap* h, int index)
{
    // Store parent of element at index
    // in parent variable
    int parent = (index - 1) / 2;
    if (h->arr[parent] < h->arr[index]) {
        // Swapping when child is smaller
        // than parent element
        int temp = h->arr[parent];
        h->arr[parent] = h->arr[index];
        h->arr[index] = temp;
        // Recursively calling maxHeapify_bottom_up
        insertHelper(h, parent);
    }
}
void maxHeapify(heap* h, int index)
```

```
{
    int left = index * 2 + 1;
    int right = index * 2 + 2;
    int max = index;
    // Checking whether our left or child element
    // is at right index of not to avoid index error
    if (left >= h->size || left < 0)
        left = -1;
    if (right >= h->size || right < 0)
        right = -1;
    // store left or right element in max if
    // any of these is smaller that its parent
    if (left != -1 && h->arr[left] > h->arr[max])
        max = left;
    if (right != -1 && h->arr[right] > h->arr[max])
        max = right;
    // Swapping the nodes
    if (max != index) {
        int temp = h->arr[max];
        h->arr[max] = h->arr[index];
        h->arr[index] = temp;
        // recursively calling for their child elements
        // to maintain max heap
        maxHeapify(h, max);
    }
}
int extractMax(heap* h)
{
    int deleteItem;
    // Checking if the heap is empty or not
    if (h->size == 0) {
        printf("\nHeap id empty.");
        return -999;
    }
```

```
    // Store the node in deleteItem that
    // is to be deleted.
    deleteItem = h->arr[0];
    // Replace the deleted node with the last node
    h->arr[0] = h->arr[h->size - 1];
    // Decrement the size of heap
    h->size--;
    // Call maxheapify_top_down for 0th index
    // to maintain the heap property
    maxHeapify(h, 0);
    return deleteItem;
}
// Define a insert function
void insert(heap* h, int data)
{
    // Checking if heap is full or not
    if (h->size < h->capacity) {
        // Inserting data into an array
        h->arr[h->size] = data;
        // Calling maxHeapify_bottom_up function
        insertHelper(h, h->size);
        // Incrementing size of array
        h->size++;
    }
}
void printHeap(heap* h)
{
int i;
    for ( i = 0; i < h->size; i++) {
        printf("%d ", h->arr[i]);
    }
    printf("\n");
}
void main()
```

```
{
   int arr[9] = {1,2,3,4,5,6,7,8,9};
   heap* hp = createHeap(9, arr);
   printHeap(hp);
   extractMax(hp);
   printHeap(hp);
}
```

OUTPUT

```
9 8 7 4 5 6 3 2 1
8 5 7 4 1 6 3 2
Process returned 10 (0xA)   execution time : 0.640 s
Press any key to continue.
```

MAX HEAP DELETION

```
#include <malloc.h>
#include <stdio.h>
// Declare a heap structure
struct Heap {
    int* arr;
    int size;
    int capacity;
};
// define the struct Heap name
typedef struct Heap heap;
// forward declarations
heap* createHeap(int capacity, int* nums);
void insertHelper(heap* h, int index);
void maxHeapify(heap* h, int index);
int extractMax(heap* h);
void insert(heap* h, int data);
// Define a createHeap function
heap* createHeap(int capacity, int* nums)
{
    // Allocating memory to heap h
    heap* h = (heap*)malloc(sizeof(heap));
    // Checking if memory is allocated to h or not
    if (h == NULL) {
        printf("Memory error");
        return NULL;
    }
    // set the values to size and capacity
    h->size = 0;
    h->capacity = capacity;
    // Allocating memory to array
```

```
    h->arr = (int*)malloc(capacity * sizeof(int));
    // Checking if memory is allocated to h or not
    if (h->arr == NULL) {
        printf("Memory error");
        return NULL;
    }
    int i;
    for (i = 0; i < capacity; i++) {
        h->arr[i] = nums[i];
    }
    h->size = i;
    i = (h->size - 2) / 2;
    while (i >= 0) {
        maxHeapify(h, i);
        i--;
    }
    return h;
}
// Defining maxHeapify_bottom_up function
void insertHelper(heap* h, int index)
{
    // Store parent of element at index
    // in parent variable
    int parent = (index - 1) / 2;
    if (h->arr[parent] < h->arr[index]) {
        // Swapping when child is smaller
        // than parent element
        int temp = h->arr[parent];
        h->arr[parent] = h->arr[index];
        h->arr[index] = temp;
```

```
        // Recursively calling maxHeapify_bottom_up
        insertHelper(h, parent);
    }
}
void maxHeapify(heap* h, int index)
{
    int left = index * 2 + 1;
    int right = index * 2 + 2;
    int max = index;
    // Checking whether our left or child element
    // is at right index of not to avoid index error
    if (left >= h->size || left < 0)
        left = -1;
    if (right >= h->size || right < 0)
        right = -1;
    // store left or right element in max if
    // any of these is smaller that its parent
    if (left != -1 && h->arr[left] > h->arr[max])
        max = left;
    if (right != -1 && h->arr[right] > h->arr[max])
        max = right;
    // Swapping the nodes
    if (max != index) {
        int temp = h->arr[max];
        h->arr[max] = h->arr[index];
        h->arr[index] = temp;
        // recursively calling for their child elements
        // to maintain max heap
        maxHeapify(h, max);
    }
```

```
}
int extractMax(heap* h)
{
    int deleteItem;
    // Checking if the heap is empty or not
    if (h->size == 0) {
        printf("\nHeap id empty.");
        return -999;
    }
    // Store the node in deleteItem that
    // is to be deleted.
    deleteItem = h->arr[0];
    // Replace the deleted node with the last node
    h->arr[0] = h->arr[h->size - 1];
    // Decrement the size of heap
    h->size--;
    // Call maxheapify_top_down for 0th index
    // to maintain the heap property
    maxHeapify(h, 0);
    return deleteItem;
}
// Define a insert function
void insert(heap* h, int data)
{
    // Checking if heap is full or not
    if (h->size < h->capacity) {
        // Inserting data into an array
        h->arr[h->size] = data;
        // Calling maxHeapify_bottom_up function
        insertHelper(h, h->size);
```

```
        // Incrementing size of array
        h->size++;
    }
}
void printHeap(heap* h)
{
int i;
    for ( i = 0; i < h->size; i++) {
        printf("%d ", h->arr[i]);
    }
    printf("\n");
}
void main()
{
    int arr[9] = {1,2,3,4,5,6,7,8,9};
    heap* hp = createHeap(9, arr);
    printHeap(hp);
    extractMax(hp);
    printHeap(hp);
}
```

OUTPUT

```
9 8 7 4 5 6 3 2 1

8 5 7 4 1 6 3 2

Process returned 10 (0xA)   execution time : 0.531 s

Press any key to conti
```

4.Implement BFT and DFT for given graph, when graph is represented by
a) Adjacency Matrix
b) Adjacency Lists

A) ADJACENCY MATRIX

```
#include <stdio.h>
// N vertices and M Edges
int N, M;
// Function to create Adjacency Matrix
void createAdjMatrix(int Adj[][N + 1],int arr[][2])
{
   // Initialise all value to this
   // Adjacency list to zero
   for (int i = 0; i < N + 1; i++) {

      for (int j = 0; j < N + 1; j++) {
         Adj[i][j] = 0;
      }
   }
   // Traverse the array of Edges
   for (int i = 0; i < M; i++) {

      // Find X and Y of Edges
      int x = arr[i][0];
      int y = arr[i][1];

      // Update value to 1
      Adj[x][y] = 1;
      Adj[y][x] = 1;
   }
}
// Function to print the created
// Adjacency Matrix
void printAdjMatrix( int Adj[][N + 1])
{
   // Traverse the Adj[][]
   for (int i = 1; i < N + 1; i++) {
      for (int j = 1; j < N + 1; j++) {
         // Print the value at Adj[i][j]
         printf("%d ", Adj[i][j]);
      }
      printf("\n");
   }
}
// Driver Code
int main()
{
```

```
    // Number of vertices
    N = 5;
    // Given Edges
    int arr[][2]
        = { { 1, 2 }, { 2, 3 },
            { 4, 5 }, { 1, 5 } };
    // Number of Edges
    M = sizeof(arr) / sizeof(arr[0]);
    // For Adjacency Matrix
    int Adj[N + 1][N + 1];
    // Function call to create
    // Adjacency Matrix
    createAdjMatrix(Adj, arr);
    // Print Adjacency Matrix
    printAdjMatrix(Adj);
    return 0;
}
```

OUTPUT

```
0 1 0 0 1
1 0 1 0 0
0 1 0 0 0
0 0 0 0 1
1 0 0 1 0

Process returned 0 (0x0)   execution time : 0.832 s
Press any key to continue.
```

B) ADJACENCY LIST

```
#include <stdio.h>
#include <stdlib.h>
// Structure to represent a node in the adjacency list
struct Node {
    int vertex;
    struct Node* next;
};
// Structure to represent the graph
struct Graph {
    int numVertices;
    struct Node** adjLists;
    int isDirected;
};
// Function to create a new node
struct Node* createNode(int v) {
    struct Node* newNode = malloc(sizeof(struct Node));
    newNode->vertex = v;
    newNode->next = NULL;
    return newNode;
}
// Function to create a graph
struct Graph* createGraph(int vertices, int isDirected) {
    struct Graph* graph = malloc(sizeof(struct Graph));
    graph->numVertices = vertices;
    graph->isDirected = isDirected;
    // Create an array of adjacency lists
    graph->adjLists = malloc(vertices * sizeof(struct Node*));
    // Initialize each adjacency list as empty
    for (int i = 0; i < vertices; i++) {
        graph->adjLists[i] = NULL;
    }
    return graph;
}
// Function to add an edge to the graph
void addEdge(struct Graph* graph, int src, int dest) {
    // Add edge from src to dest
    struct Node* newNode = createNode(dest);
    newNode->next = graph->adjLists[src];
    graph->adjLists[src] = newNode;
    // If the graph is undirected, add an edge from dest to src as well
    if (!graph->isDirected) {
        newNode = createNode(src);
        newNode->next = graph->adjLists[dest];

        graph->adjLists[dest] = newNode;
```

```
    }
}
// Function to print the adjacency list representation of the graph
void printGraph(struct Graph* graph) {
    printf("Vertex:  Adjacency List\n");
    for (int v = 0; v < graph->numVertices; v++) {
        struct Node* temp = graph->adjLists[v];
        printf("%d --->", v);
        while (temp) {
            printf(" %d ->", temp->vertex);
            temp = temp->next;
        }
        printf(" NULL\n");
    }
}
int main() {
    // Create an undirected graph with 3 vertices
    struct Graph* undirectedGraph = createGraph(3, 0);
    // Add edges to the undirected graph
    addEdge(undirectedGraph, 0, 1);
    addEdge(undirectedGraph, 0, 2);
    addEdge(undirectedGraph, 1, 2);
    printf("Adjacecncy List for Undirected Graph:\n");
    printGraph(undirectedGraph);
    // Create a directed graph with 3 vertices
    struct Graph* directedGraph = createGraph(3, 1);
    // Add edges to the directed graph
    addEdge(directedGraph, 1, 0);
    addEdge(directedGraph, 1, 2);
    addEdge(directedGraph, 2, 0);
```

```
    printf("\nAdjacecncy List for Directed Graph:\n");
    printGraph(directedGraph);
    return 0;
}
```

OUTPUT

```
Adjacency List for Undirected Graph:
Vertex:  Adjacency List
0 ---> 2 -> 1 -> NULL
1 ---> 2 -> 0 -> NULL
2 ---> 1 -> 0 -> NULL
Adjacency List for Directed Graph:
Vertex:  Adjacency List
0 ---> NULL
1 ---> 2 -> 0 -> NULL
2 ---> 0 -> NULL
Process returned 0 (0x0)   execution time : 0.677 s
Press any key to continue.
```

5.Write a program for finding the biconnected components in a given graph

```
#include <stdio.h>
#include <stdlib.h>
#include <stdbool.h>
#define MAX 100
// Structure for storing an edge
typedef struct {
   int u, v;
} Edge;
int V, E;
int timeCounter;
int disc[MAX], low[MAX], parent[MAX];
bool visited[MAX];
Edge stack[MAX * MAX];
int top = -1;
// Graph using adjacency list
int adj[MAX][MAX];
int degree[MAX];
// Store all Biconnected Components
void printComponentUntil(Edge target) {
   Edge e;
   printf("Biconnected Component:\n");
   do {
      e = stack[top--];
      printf("(%d -- %d)\n", e.u, e.v);
   } while (!(e.u == target.u && e.v == target.v));
   printf("\n");
}
void DFS(int u) {
   visited[u] = true;
   disc[u] = low[u] = ++timeCounter;
   int children = 0;
  for (int i = 0; i < degree[u]; i++) {
      int v = adj[u][i];
      if (!visited[v]) {
         children++;
         parent[v] = u;
         stack[++top] = (Edge){u, v};
         DFS(v);

         low[u] = (low[u] < low[v]) ? low[u] : low[v];
         if ((parent[u] == -1 && children > 1) ||
            (parent[u] != -1 && low[v] >= disc[u])) {
            printComponentUntil((Edge){u, v});
         }
      }
```

```
        else if (v != parent[u] && disc[v] < disc[u]) {
            low[u] = (low[u] < disc[v]) ? low[u] : disc[v];
            stack[++top] = (Edge){u, v};
        }
    }
}
void findBCC( ) {
    for (int i = 0; i < V; i++) {
        parent[i] = -1;
        visited[i] = false;
        disc[i] = low[i] = -1;
    }
    timeCounter = 0;
    for (int i = 0; i < V; i++) {
        if (!visited[i]) {
            DFS(i);

            // Print remaining edges
            if (top != -1) {
                printf("Biconnected Component:\n");
                while (top != -1) {
                    Edge e = stack[top--];
                    printf("(%d -- %d)\n", e.u, e.v);
                }
                printf("\n");
            }
        }
    }
}

void addEdge(int u, int v) {
    adj[u][degree[u]++] = v;
    adj[v][degree[v]++] = u;
}

int main() {
    printf("Enter number of vertices and edges: ");
    scanf("%d %d", &V, &E);
    printf("Enter %d edges (u v):\n", E);
    for (int i = 0; i < E; i++) {
        int u, v;
        scanf("%d %d", &u, &v);
        addEdge(u, v);
    }
    printf("\nFinding Biconnected Components...\n\n");
    findBCC();
```

```
    return 0;
}
```

OUTPUT

```
Enter number of vertices and edges: 5 5
Enter 5 edges (u v):
0 1
1 2
2 3
3 4
4 5
Finding Biconnected Components...

Biconnected Component:
(4 -- 5)

Biconnected Component:
(3 -- 4)

Biconnected Component:
(2 -- 3)

Biconnected Component:
(1 -- 2)

Biconnected Component:
(0 -- 1)

Process returned 0 (0x0)   execution time : 16.283 s
Press any key to continue.
```

6.Implement Quick sort and Merge sort and observe the execution time for various input sizes (Average, Worst and Best cases).

QUICK SORT

```
#include<stdio.h>
#include<conio.h>
void quicksort(int number[25],int first,int last){
   int i, j, pivot, temp;
 if(first<last)
{
 pivot=first;
  i=first;
j=last;
while(i<j)
{
while(number[i]<=number[pivot]&&i<last)
i++;
while(number[j]>number[pivot])
 j--;
if(i<j)
{
 temp=number[i];
number[i]=number[j];
number[j]=temp;
 }
}
 temp=number[pivot];
  number[pivot]=number[j];
   number[j]=temp;
  quicksort(number,first,j-1);
 quicksort(number,j+1,last);
}
```

```
}
int main()
{
  int i, count, number[25];
 printf("How many elements are u going to enter?: ");
  scanf("%d",&count);
printf("Enter %d elements: ", count);
  for(i=0;i<count;i++)
 scanf("%d",&number[i]);
quicksort(number,0,count-1);
printf("Order of Sorted elements: ");
   for(i=0;i<count;i++)
      printf(" %d",number[i]);
getch();
return 0;
}
```

OUTPUT

How many elements are you going to enter: 12

Enter 12 elements: 2 3 1 22 33 35 34 45 53 39 47 52

Order of Sorted elements: 1 2 3 22 33 34 35 39 45 47 52 53

MERGE SORT

```
#include <stdio.h>
#include <stdlib.h>
#include <conio.h>
// merge function
void Merge(int arr[], int left, int mid, int right)
{
   int i, j, k;
   int size1 = mid - left + 1;
   int size2 = right - mid;
 int Left[size1], Right[size2];
 for (i = 0; i < size1; i++)
       Left[i] = arr[left + i];
for (j = 0; j < size2; j++)
       Right[j] = arr[mid + 1 + j];
// merging of the array
   i = 0;
   j = 0;
   k = left;
   while (i < size1 && j < size2)
   {
      if (Left[i] <= Right[j])
      {
         arr[k] = Left[i];
         i++;
      }
      else
      {
         arr[k] = Right[j];
         j++;
```

```
        }
        k++;
    }
// copying the elements from Left[], if any
    while (i < size1)
    {
        arr[k] = Left[i];
        i++;
        k++;
    }
// copying the elements from Right[], if any
    while (j < size2)
    {
        arr[k] = Right[j];
        j++;
        k++;
    }
}
//merge sort function
void Merge_Sort(int arr[], int left, int right)
{
    if (left < right)
    {
int mid = left + (right - left) / 2;
 // recursive calling of merge_sort
        Merge_Sort(arr, left, mid);
        Merge_Sort(arr, mid + 1, right);
Merge(arr, left, mid, right);
    }
}
```

```
int main()
{
 int size,i;
 printf("Enter the size: ");
 scanf("%d", &size);
 int arr[size];
 printf("Enter the elements of array: ");
    for (i = 0; i < size; i++)
    {
        scanf("%d", &arr[i]);
    }
Merge_Sort(arr, 0, size - 1);
printf("The sorted array is: ");
 for (i = 0; i < size; i++)
    {
        printf("%d ", arr[i]);
    }
    printf("\n");
    getch();
    return 0;
}
```

OUTPUT

Enter the size: 11

Enter the elements of array: 23 24 25 2 34 42 56 87 29 11 5

The sorted array is: 2 5 11 23 24 25 29 34 42 56 87

7.Compare the performance of Single Source Shortest Paths using Greedy method when the graph is represented by adjacency matrix and adjacency lists

```
#include <stdio.h>
#include <limits.h>
#define MAX 100
#define INF INT_MAX
// Function to find the vertex with minimum distance
int min_distance(int dist[], int visited[], int V) {
   int min = INF, min_index = -1;
   for (int v = 0; v < V; v++)
      if (!visited[v] && dist[v] <= min) {
         min = dist[v];
         min_index = v;
      }
   return min_index;
}
// Dijkstra's Algorithm
void dijkstra(int graph[MAX][MAX], int V, int src) {
   int dist[MAX];
   int visited[MAX] = {0};
   for (int i = 0; i < V; i++)
      dist[i] = INF;
   dist[src] = 0;
   for (int count = 0; count < V - 1; count++) {
      int u = min_distance(dist, visited, V);
      visited[u] = 1;

      for (int v = 0; v < V; v++) {
         if (!visited[v] && graph[u][v] && dist[u] != INF
            && dist[u] + graph[u][v] < dist[v]) {
            dist[v] = dist[u] + graph[u][v];
         }
      }
   }

   // Print the result
   printf("Vertex\tDistance from Source %d\n", src);
   for (int i = 0; i < V; i++)
      printf("%d\t\t%d\n", i, dist[i]);
}
int main() {
   int V = 5;
   int graph[MAX][MAX] = {
      {0, 10, 0, 30, 100},
      {10, 0, 50, 0, 0},
      {0, 50, 0, 20, 10},
```

```
        {30, 0, 20, 0, 60},
        {100, 0, 10, 60, 0}
    };
    int source = 0;
    dijkstra(graph, V, source);
    return 0;
}
```

OUTPUT

```
Vertex     Distance from Source 0
0           0
1           10
2           50
3           30
4           60

Process returned 0 (0x0)   execution time : 0.765 s
Press any key to continue.
```

```
Graph (represented as a matrix) is:
    (10)       (50)
 0 ------- 1 -------- 2
 |        |         |
(30)    (100)      (10)
 |        |         |
 3 ------- 4 ---------
    (60)
```

8.Implement Job Sequencing with deadlines using Greedy strategy.

```
#include <stdbool.h>
#include <stdio.h>
#include <stdlib.h>
// A structure to represent a Jobs
typedef struct Jobs
{
  char id; // Jobs Id
  int dead; // Deadline of Jobs
  int profit; // Profit if Jobs is over before or on deadline
} Jobs;
// This function is used for sorting all Jobs's according to profit
int compare(const void* a, const void* b){
  Jobs* temp1 = (Jobs*)a;
  Jobs* temp2 = (Jobs*)b;
  return (temp2->profit - temp1->profit);
}
// Find minimum between two numbers.
int min(int num1, int num2){
  return (num1 > num2) ? num2 : num1;
}
int main(){
  Jobs arr[] = {
    { 'a', 2, 100 },
    { 'b', 2, 20 },
    { 'c', 1, 40 },
    { 'd', 3, 35 },
    { 'e', 1, 25 }
  };
  int n = sizeof(arr) / sizeof(arr[0]);
  printf("Following is maximum profit sequence of Jobs: \n");
  qsort(arr, n, sizeof(Jobs), compare);
  int result[n]; // To store result sequence of Jobs
  bool slot[n]; // To keep track of free time slots
  // Initialize all slots to be free
  for (int i = 0; i < n; i++)
    slot[i] = false;
  // Iterate through all given Jobs
  for (int i = 0; i < n; i++) {
    // Find a free slot for this Job
    for (int j = min(n, arr[i].dead) - 1; j >= 0; j--) {
      // Free slot found
      if (slot[j] == false) {
        result[j] = i;
        slot[j] = true;
        break;
      }
 }
}
```

```
  // Print the result
  for (int i = 0; i < n; i++)
    if (slot[i])
      printf("%c ", arr[result[i]].id);
  return 0;
}
```

OUTPUT

Following is maximum profit sequence of Jobs:

c a d

Process returned 0 (0x0) execution time : 0.840 s

Press any key to continue.

9.Write a program to solve 0/1 Knapsack problem Using Dynamic Programming.

```
#include <stdio.h>
#include <string.h>
int findMax(int n1, int n2){
  if(n1>n2) {
    return n1;
  } else {
    return n2;
  }
}
int knapsack(int W, int wt[], int val[], int n){
  int K[n+1][W+1];
  for(int i = 0; i<=n; i++) {
    for(int w = 0; w<=W; w++) {
      if(i == 0 || w == 0) {
        K[i][w] = 0;
      } else if(wt[i-1] <= w) {
        K[i][w] = findMax(val[i-1] + K[i-1][w-wt[i-1]], K[i-1][w]);
      } else {
        K[i][w] = K[i-1][w];
      }
    }
  }
  return K[n][W];
}
int main(){
  int val[5] = {70, 20, 50};
  int wt[5] = {11, 12, 13};
  int W = 30;
  int len = sizeof val / sizeof val[0];
  printf("Maximum Profit achieved with this knapsack: %d", knapsack(W, wt, val, len));
}
```

OUTPUT

Maximum Profit achieved with this knapsack: 120

Process returned 0 (0x0) execution time : 0.012 s

Press any key to continue

10. Implement N-Queens Problem Using Backtracking.

```
#include<stdio.h>
#define BOARD_SIZE 5
void displayChess(int chBoard[BOARD_SIZE][BOARD_SIZE])
{
  for (int row = 0; row < BOARD_SIZE; row++)
{
    for (int col = 0; col < BOARD_SIZE; col++)
      printf("%d ", chBoard[row][col]);
    printf("\n");
  }
}
int isQueenPlaceValid(int chBoard[BOARD_SIZE][BOARD_SIZE], int crntRow, int
crntCol)
{
  // checking if queen is in the left or not
  for (int i = 0; i < crntCol; i++)
    if (chBoard[crntRow][i])
      return 0;
  for (int i = crntRow, j = crntCol; i >= 0 && j >= 0; i--, j--)
    //checking if queen is in the left upper diagonal or not
    if (chBoard[i][j])
      return 0;
  for (int i = crntRow, j = crntCol; j >= 0 && i < BOARD_SIZE; i++, j--)
    //checking if queen is in the left lower diagonal or not
    if (chBoard[i][j])
      return 0;
  return 1;
}
int solveProblem(int chBoard[BOARD_SIZE][BOARD_SIZE], int crntCol)
{
  //when N queens are placed successfully
  if (crntCol >= BOARD_SIZE)
    return 1;
  // checking placement of queen is possible or not
  for (int i = 0; i < BOARD_SIZE; i++) {
    if (isQueenPlaceValid(chBoard, i, crntCol))
{
      //if validate, place the queen at place (i, col)
      chBoard[i][crntCol] = 1;
      //Go for the other columns recursively
      if (solveProblem(chBoard, crntCol + 1))
        return 1;
      //When no place is vacant remove that queen
      chBoard[i][crntCol] = 0;
    }
  }
  return 0;
}
```

```
int displaySolution() {
  int chBoard[BOARD_SIZE][BOARD_SIZE];
  for(int i = 0; i < BOARD_SIZE; i++)
    for(int j = 0; j < BOARD_SIZE; j++)
      //set all elements to 0
      chBoard[i][j] = 0;
  //starting from 0th column
  if (solveProblem(chBoard, 0) == 0)
{
    printf("Solution does not exist");
    return 0;
  }
  displayChess(chBoard);
  return 1;
}
int main()
 {
  displaySolution();
  return 0;
}
```

OUTPUT

```
1 0 0 0 0

0 0 0 1 0

0 1 0 0 0

0 0 0 0 1

0 0 1 0 0

Process returned 0 (0x0)   execution time : 0.846 s

Press any key to continue
```

11. Use Backtracking strategy to solve 0/1Knapsack problem.

```
#include <stdio.h>
// Function to find maximum between two numbers
int max(int a, int b)
{
   if (a > b)
      return a;
   return b;
}
// Returns the maximum value that can be put in a knapsack
// of capacity W
int knapsackRecursive(int W, int wt[], int val[], int n)
{
   // Base Case
   if (n == 0 || W == 0)
      return 0;
   if (wt[n - 1] > W)
      return knapsackRecursive(W, wt, val, n - 1);
   else
      return max(val[n - 1]  + knapsackRecursive(W - wt[n - 1], wt, val, n - 1),
               knapsackRecursive(W, wt, val, n - 1));
}
// Driver Code
int main()
{
   int values[] = { 3, 4, 5, 6 };
   int weight[] = { 2, 3, 4, 5 };
   int W = 8;
   // Find the number of items
   int n = sizeof(values) / sizeof(values[0]);
   // Output the maximum profit for the knapSack
   printf(  "Maximum value that can be put in knapsack: %d\n", knapsackRecursive(W,
weight,    values, n));
   return 0;
}
```

OUTPUT

```
Maximum value that can be put in knapsack: 10
Process returned 0 (0x0)   execution time : 0.000 s
Press any key to continue.
```

12.Implement Travelling Sales Person problem using Branch and Bound approach.

```
#include <stdio.h>
#include <stdlib.h>
#include <limits.h>
#define N 4 // Number of cities
int final_path[N + 1];
int visited[N];
int final_res = INT_MAX;
// Copy current path to final_path
void copyToFinal(int curr_path[]) {
    for (int i = 0; i < N; i++)
        final_path[i] = curr_path[i];
    final_path[N] = curr_path[0];
}
// Find the minimum edge cost from city i
int firstMin(int cost[N][N], int i) {
    int min = INT_MAX;
    for (int k = 0; k < N; k++)
        if (cost[i][k] < min && i != k)
            min = cost[i][k];
    return min;
}
// Find the second minimum edge cost from city i
int secondMin(int cost[N][N], int i) {
    int first = INT_MAX, second = INT_MAX;
    for (int j = 0; j < N; j++) {
        if (i == j)
            continue;
        if (cost[i][j] <= first) {
            second = first;
            first = cost[i][j];
        } else if (cost[i][j] < second && cost[i][j] != first)
            second = cost[i][j];
    }
    return second;
}
// Branch and Bound recursive function
void TSPRec(int cost[N][N], int curr_bound, int curr_weight,
            int level, int curr_path[]) {
    if (level == N) {
        if (cost[curr_path[level - 1]][curr_path[0]] != 0) {
            int curr_res = curr_weight + cost[curr_path[level - 1]][curr_path[0]];
            if (curr_res < final_res) {
                copyToFinal(curr_path);
                final_res = curr_res;
            }
        }
```

```
        return;
    }
    for (int i = 0; i < N; i++) {
        if (cost[curr_path[level - 1]][i] != 0 && visited[i] == 0) {
            int temp = curr_bound;
            curr_weight += cost[curr_path[level - 1]][i];
            if (level == 1)
                curr_bound -= ((firstMin(cost, curr_path[level - 1]) + firstMin(cost, i)) / 2);
            else
                curr_bound -= ((secondMin(cost, curr_path[level - 1]) + firstMin(cost, i)) / 2);

            if (curr_bound + curr_weight < final_res) {
                curr_path[level] = i;
                visited[i] = 1;
                TSPRec(cost, curr_bound, curr_weight, level + 1, curr_path);
            }
            curr_weight -= cost[curr_path[level - 1]][i];
            curr_bound = temp;

            for (int j = 0; j < N; j++)
                visited[j] = 0;
            for (int j = 0; j <= level - 1; j++)
                visited[curr_path[j]] = 1;
        }
    }
}
// Main function to set up and call recursive TSP solver
void TSP(int cost[N][N]) {
    int curr_path[N + 1];
    int curr_bound = 0;

    for (int i = 0; i < N; i++) {
        visited[i] = 0;
        curr_path[i] = -1;
    }
    for (int i = 0; i < N; i++)
        curr_bound += (firstMin(cost, i) + secondMin(cost, i));
        curr_bound = (curr_bound & 1) ? curr_bound / 2 + 1 : curr_bound / 2;
    visited[0] = 1;
    curr_path[0] = 0;
    TSPRec(cost, curr_bound, 0, 1, curr_path);
}
int main() {
    int cost[N][N] = {
        {0, 10, 15, 20},
        {10, 0, 35, 25},
```

```
        {15, 35, 0, 30},
        {20, 25, 30, 0}
    };
    TSP(cost);
    printf("Minimum cost: %d\n", final_res);
    printf("Path Taken: ");
    for (int i = 0; i <= N; i++)
        printf("%d ", final_path[i]);
    printf("\n");
    return 0;
}
```

OUTPUT

```
Minimum cost: 80
Path Taken: 0 1 3 2 0

Process returned 0 (0x0)   execution time : 1.156 s
Press any key to continue.
```