## Program 1
# // AVL tree implementation in C

```
#include <stdio.h>
#include <stdlib.h>

// Create Node
struct Node {
  int key;
  struct Node *left;
  struct Node *right;
  int height;
};

int max(int a, int b);

// Calculate height
int height(struct Node *N) {
  if (N == NULL)
    return 0;
  return N->height;
}

int max(int a, int b) {
  return (a > b) ? a : b;
}

// Create a 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;
  return (node);
}

// Right rotate
struct Node *rightRotate(struct Node *y) {
  struct Node *x = y->left;
  struct Node *T2 = x->right;
```

```
  x->right = y;
  y->left = T2;

  y->height = max(height(y->left), height(y->right)) + 1;
  x->height = max(height(x->left), height(x->right)) + 1;

  return x;
}

// Left rotate
struct Node *leftRotate(struct Node *x) {
  struct Node *y = x->right;
  struct Node *T2 = y->left;

  y->left = x;
  x->right = T2;

  x->height = max(height(x->left), height(x->right)) + 1;
  y->height = max(height(y->left), height(y->right)) + 1;

  return y;
}

// Get the balance factor
int getBalance(struct Node *N) {
  if (N == NULL)
    return 0;
  return height(N->left) - height(N->right);
}

// Insert node
struct Node *insertNode(struct Node *node, int key) {
  // Find the correct position to insertNode the node and insertNode it
  if (node == NULL)
    return (newNode(key));

  if (key < node->key)
    node->left = insertNode(node->left, key);
  else if (key > node->key)
    node->right = insertNode(node->right, key);
  else
    return node;

  // Update the balance factor of each node and
```

```
  // Balance the tree
  node->height = 1 + max(height(node->left),
              height(node->right));

  int balance = getBalance(node);
  if (balance > 1 && key < node->left->key)
    return rightRotate(node);

  if (balance < -1 && key > node->right->key)
    return leftRotate(node);

  if (balance > 1 && key > node->left->key) {
    node->left = leftRotate(node->left);
    return rightRotate(node);
  }

  if (balance < -1 && key < node->right->key) {
    node->right = rightRotate(node->right);
    return leftRotate(node);
  }

  return node;
}

struct Node *minValueNode(struct Node *node) {
  struct Node *current = node;

  while (current->left != NULL)
    current = current->left;

  return current;
}

// Delete a nodes
struct Node *deleteNode(struct Node *root, int key) {
  // Find the node and delete it
  if (root == NULL)
    return root;

  if (key < root->key)
    root->left = deleteNode(root->left, key);

  else if (key > root->key)
    root->right = deleteNode(root->right, key);
```

```
else {
  if ((root->left == NULL) || (root->right == NULL)) {
    struct Node *temp = root->left ? root->left : root->right;

    if (temp == NULL) {
      temp = root;
      root = NULL;
    } else
      *root = *temp;
    free(temp);
  } else {
    struct Node *temp = minValueNode(root->right);

    root->key = temp->key;

    root->right = deleteNode(root->right, temp->key);
  }
}

if (root == NULL)
  return root;

// Update the balance factor of each node and
// balance the tree
root->height = 1 + max(height(root->left),
             height(root->right));

int balance = getBalance(root);
if (balance > 1 && getBalance(root->left) >= 0)
  return rightRotate(root);

if (balance > 1 && getBalance(root->left) < 0) {
  root->left = leftRotate(root->left);
  return rightRotate(root);
}

if (balance < -1 && getBalance(root->right) <= 0)
  return leftRotate(root);

if (balance < -1 && getBalance(root->right) > 0) {
  root->right = rightRotate(root->right);
  return leftRotate(root);
}
```

```
  return root;
}

// Print the tree
void printPreOrder(struct Node *root) {
  if (root != NULL) {
    printf("%d ", root->key);
    printPreOrder(root->left);
    printPreOrder(root->right);
  }
}

int main() {
  struct Node *root = NULL;

  root = insertNode(root, 2);
  root = insertNode(root, 1);
  root = insertNode(root, 7);
  root = insertNode(root, 4);
  root = insertNode(root, 5);
  root = insertNode(root, 3);
  root = insertNode(root, 8);

  printPreOrder(root);

  root = deleteNode(root, 3);

  printf("\nAfter deletion: ");
  printPreOrder(root);

  return 0;
}
```

Other links: https://www.w3resource.com/c-programming-exercises/tree/c-tree-exercises-10.php

### Program 2

## // Searching a key on a B-tree in C

```
#include <stdio.h>
#include <stdlib.h>

#define MAX 3
```

```c
#define MIN 2

struct BTreeNode {
  int val[MAX + 1], count;
  struct BTreeNode *link[MAX + 1];
};

struct BTreeNode *root;

// Create a node
struct BTreeNode *createNode(int val, struct BTreeNode *child) {
  struct BTreeNode *newNode;
  newNode = (struct BTreeNode *)malloc(sizeof(struct BTreeNode));
  newNode->val[1] = val;
  newNode->count = 1;
  newNode->link[0] = root;
  newNode->link[1] = child;
  return newNode;
}

// Insert node
void insertNode(int val, int pos, struct BTreeNode *node,
        struct BTreeNode *child) {
  int j = node->count;
  while (j > pos) {
    node->val[j + 1] = node->val[j];
    node->link[j + 1] = node->link[j];
    j--;
  }
  node->val[j + 1] = val;
  node->link[j + 1] = child;
  node->count++;
}

// Split node
void splitNode(int val, 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)->val[j - median] = node->val[j];
    (*newNode)->link[j - median] = node->link[j];
    j++;
  }
  node->count = median;
  (*newNode)->count = MAX - median;

  if (pos <= MIN) {
    insertNode(val, pos, node, child);
  } else {
    insertNode(val, pos - median, *newNode, child);
  }
  *pval = node->val[node->count];
  (*newNode)->link[0] = node->link[node->count];
  node->count--;
}

// Set the value
int setValue(int val, int *pval,
           struct BTreeNode *node, struct BTreeNode **child) {
  int pos;
  if (!node) {
    *pval = val;
    *child = NULL;
    return 1;
  }

  if (val < node->val[1]) {
    pos = 0;
  } else {
    for (pos = node->count;
       (val < node->val[pos] && pos > 1); pos--)
      ;
    if (val == node->val[pos]) {
      printf("Duplicates are not permitted\n");
      return 0;
    }
  }
  if (setValue(val, pval, node->link[pos], child)) {
    if (node->count < MAX) {
      insertNode(*pval, pos, node, *child);
```

```
    } else {
      splitNode(*pval, pval, pos, node, *child, child);
      return 1;
    }
  }
  return 0;
}

// Insert the value
void insert(int val) {
  int flag, i;
  struct BTreeNode *child;

  flag = setValue(val, &i, root, &child);
  if (flag)
    root = createNode(i, child);
}

// Search node
void search(int val, int *pos, struct BTreeNode *myNode) {
  if (!myNode) {
    return;
  }

  if (val < myNode->val[1]) {
    *pos = 0;
  } else {
    for (*pos = myNode->count;
       (val < myNode->val[*pos] && *pos > 1); (*pos)--)
      ;
    if (val == myNode->val[*pos]) {
      printf("%d is found", val);
      return;
    }
  }
  search(val, pos, myNode->link[*pos]);

  return;
}

// Traverse then nodes
void traversal(struct BTreeNode *myNode) {
  int i;
  if (myNode) {
```

```
    for (i = 0; i < myNode->count; i++) {
      traversal(myNode->link[i]);
      printf("%d ", myNode->val[i + 1]);
    }
    traversal(myNode->link[i]);
  }
}

int main() {
  int val, ch;

  insert(8);
  insert(9);
  insert(10);
  insert(11);
  insert(15);
  insert(16);
  insert(17);
  insert(18);
  insert(20);
  insert(23);

  traversal(root);

  printf("\n");
  search(11, &ch, root);
}
```

## Program 3
## // 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;
}

// Insert
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);
}
```

## // 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);
}
```

## Program 3

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

```
// Max-Heap data structure in C

#include <stdio.h>
int size = 0;
void swap(int *a, int *b)
{
  int temp = *b;
  *b = *a;
  *a = temp;
}
void heapify(int array[], int size, int i)
{
  if (size == 1)
  {
    printf("Single element in the heap");
  }
  else
  {
    int largest = i;
    int l = 2 * i + 1;
    int r = 2 * i + 2;
    if (l < size && array[l] > array[largest])
      largest = l;
    if (r < size && array[r] > array[largest])
      largest = r;
```

```
    if (largest != i)
    {
      swap(&array[i], &array[largest]);
      heapify(array, size, largest);
    }
  }
}
void insert(int array[], int newNum)
{
  if (size == 0)
  {
    array[0] = newNum;
    size += 1;
  }
  else
  {
    array[size] = newNum;
    size += 1;
    for (int i = size / 2 - 1; i >= 0; i--)
    {
      heapify(array, size, i);
    }
  }
}
void deleteRoot(int array[], int num)
{
  int i;
  for (i = 0; i < size; i++)
  {
    if (num == array[i])
      break;
  }

  swap(&array[i], &array[size - 1]);
  size -= 1;
  for (int i = size / 2 - 1; i >= 0; i--)
  {
    heapify(array, size, i);
  }
}
```

```
void printArray(int array[], int size)
{
  for (int i = 0; i < size; ++i)
    printf("%d ", array[i]);
  printf("\n");
}
int main()
{
  int array[10];

  insert(array, 3);
  insert(array, 4);
  insert(array, 9);
  insert(array, 5);
  insert(array, 2);

  printf("Max-Heap array: ");
  printArray(array, size);

  deleteRoot(array, 4);

  printf("After deleting an element: ");

  printArray(array, size);
}
```

## Construct a min heap tree program using c

```
#include <stdio.h>
#include <stdlib.h>

typedef struct MinHeap MinHeap;
struct MinHeap {
    int* arr;
    // Current Size of the Heap
    int size;
    // Maximum capacity of the heap
    int capacity;
};
```

```
int parent(int i) {
    // Get the index of the parent
    return (i - 1) / 2;
}

int left_child(int i) {
    return (2*i + 1);
}

int right_child(int i) {
    return (2*i + 2);
}

int get_min(MinHeap* heap) {
    // Return the root node element,
    // since that's the minimum
    return heap->arr[0];
}

MinHeap* init_minheap(int capacity) {
    MinHeap* minheap = (MinHeap*) calloc (1, sizeof(MinHeap));
    minheap->arr = (int*) calloc (capacity, sizeof(int));
    minheap->capacity = capacity;
    minheap->size = 0;
    return minheap;
}

MinHeap* insert_minheap(MinHeap* heap, int element) {
    // Inserts an element to the min heap
    // We first add it to the bottom (last level)
    // of the tree, and keep swapping with it's parent
    // if it is lesser than it. We keep doing that until
    // we reach the root node. So, we will have inserted the
    // element in it's proper position to preserve the min heap property
    if (heap->size == heap->capacity) {
        fprintf(stderr, "Cannot insert %d. Heap is already full!\n", element);
        return heap;
    }
    // We can add it. Increase the size and add it to the end
    heap->size++;
```

```
    heap->arr[heap->size - 1] = element;

    // Keep swapping until we reach the root
    int curr = heap->size - 1;
    // As long as you aren't in the root node, and while the
    // parent of the last element is greater than it
    while (curr > 0 && heap->arr[parent(curr)] > heap->arr[curr]) {
        // Swap
        int temp = heap->arr[parent(curr)];
        heap->arr[parent(curr)] = heap->arr[curr];
        heap->arr[curr] = temp;
        // Update the current index of element
        curr = parent(curr);
    }
    return heap;
}

MinHeap* heapify(MinHeap* heap, int index) {
    // Rearranges the heap as to maintain
    // the min-heap property
    if (heap->size <= 1)
        return heap;

    int left = left_child(index);
    int right = right_child(index);

    // Variable to get the smallest element of the subtree
    // of an element an index
    int smallest = index;

    // If the left child is smaller than this element, it is
    // the smallest
    if (left < heap->size && heap->arr[left] < heap->arr[index])
        smallest = left;

    // Similarly for the right, but we are updating the smallest element
    // so that it will definitely give the least element of the subtree
    if (right < heap->size && heap->arr[right] < heap->arr[smallest])
        smallest = right;
```

```
    // Now if the current element is not the smallest,
    // swap with the current element. The min heap property
    // is now satisfied for this subtree. We now need to
    // recursively keep doing this until we reach the root node,
    // the point at which there will be no change!
    if (smallest != index)
    {
        int temp = heap->arr[index];
        heap->arr[index] = heap->arr[smallest];
        heap->arr[smallest] = temp;
        heap = heapify(heap, smallest);
    }

    return heap;
}

MinHeap* delete_minimum(MinHeap* heap) {
    // Deletes the minimum element, at the root
    if (!heap || heap->size == 0)
        return heap;

    int size = heap->size;
    int last_element = heap->arr[size-1];

    // Update root value with the last element
    heap->arr[0] = last_element;

    // Now remove the last element, by decreasing the size
    heap->size--;
    size--;

    // We need to call heapify(), to maintain the min-heap
    // property
    heap = heapify(heap, 0);
    return heap;
}

MinHeap* delete_element(MinHeap* heap, int index) {
    // Deletes an element, indexed by index
    // Ensure that it's lesser than the current root
```

```
    heap->arr[index] = get_min(heap) - 1;

    // Now keep swapping, until we update the tree
    int curr = index;
    while (curr > 0 && heap->arr[parent(curr)] > heap->arr[curr]) {
        int temp = heap->arr[parent(curr)];
        heap->arr[parent(curr)] = heap->arr[curr];
        heap->arr[curr] = temp;
        curr = parent(curr);
    }

    // Now simply delete the minimum element
    heap = delete_minimum(heap);
    return heap;
}

void print_heap(MinHeap* heap) {
    // Simply print the array. This is an
    // inorder traversal of the tree
    printf("Min Heap:\n");
    for (int i=0; i<heap->size; i++) {
        printf("%d -> ", heap->arr[i]);
    }
    printf("\n");
}

void free_minheap(MinHeap* heap) {
    if (!heap)
        return;
    free(heap->arr);
    free(heap);
}

int main() {
    // Capacity of 10 elements
    MinHeap* heap = init_minheap(10);

    insert_minheap(heap, 40);
    insert_minheap(heap, 50);
    insert_minheap(heap, 5);
```

```
    print_heap(heap);

    // Delete the heap->arr[1] (50)
    delete_element(heap, 1);

    print_heap(heap);
    free_minheap(heap);
    return 0;
}
```

## 4. Implement BFT and DFT for given graph, when graph is represented by

## a) Adjacency Matrix b) Adjacency Lists

```
/*

 * C program to implement bfs using adjacency matrix

 */


#include <stdio.h>


int n, i, j, visited[10], queue[10], front = -1, rear = -1;

int adj[10][10];


void bfs(int v)

{

    for (i = 1; i <= n; i++)

        if (adj[v][i] && !visited[i])

            queue[++rear] = i;
```

```
    if (front <= rear)
    {
        visited[queue[front]] = 1;
        bfs(queue[front++]);
    }
}

void main()
{
    int v;
    printf("Enter the number of vertices: ");
    scanf("%d", &n);
    for (i = 1; i <= n; i++)
    {
        queue[i] = 0;
        visited[i] = 0;
    }
    printf("Enter graph data in matrix form:    \n");
    for (i = 1; i <= n; i++)
        for (j = 1; j <= n; j++)
            scanf("%d", &adj[i][j]);
    printf("Enter the starting vertex: ");
    scanf("%d", &v);
    bfs(v);
```

```
    printf("The node which are reachable are:    \n");

    for (i = 1; i <= n; i++)

        if (visited[i])

            printf("%d\t", i);

        else

            printf("BFS is not possible. Not all nodes are reachable");

    return 0;
}
```

Output

```
Enter the number of vertices: 4
Enter graph data in matrix form:
0 1 1 0
1 0 0 1
1 0 0 1
0 1 1 0
Enter the starting vertex: 2
The node which are reachable are:
1       2       3       4
```

## BFS Program in C using Adjacency List

```
*
 * C program to implement bfs using adjacency list
 */

#include <stdio.h>
#include <stdlib.h>

struct node
{
    int vertex;
    struct node *next;
};

struct node *createNode(int);
```

```
struct Graph
{
    int numVertices;
    struct node **adjLists;
    int *visited;
};

struct Graph *createGraph(int vertices)
{
    struct Graph *graph = malloc(sizeof(struct Graph));
    graph->numVertices = vertices;

    graph->adjLists = malloc(vertices * sizeof(struct node *));
    graph->visited = malloc(vertices * sizeof(int));

    int i;
    for (i = 0; i < vertices; i++)
    {
        graph->adjLists[i] = NULL;
        graph->visited[i] = 0;
    }

    return graph;
}

void addEdge(struct Graph *graph, int src, int dest)
{
    struct node *newNode = createNode(dest);
    newNode->next = graph->adjLists[src];
    graph->adjLists[src] = newNode;

    newNode = createNode(src);
    newNode->next = graph->adjLists[dest];
    graph->adjLists[dest] = newNode;
}

struct node *createNode(int v)
{
    struct node *newNode = malloc(sizeof(struct node));
```

```
    newNode->vertex = v;
    newNode->next = NULL;
    return newNode;
}

void printGraph(struct Graph *graph)
{
    int v;
    for (v = 0; v < graph->numVertices; v++)
    {
        struct node *temp = graph->adjLists[v];
        printf("\n Adjacency list of vertex %d\n ", v);
        while (temp)
        {
            printf("%d -> ", temp->vertex);
            temp = temp->next;
        }
        printf("\n");
    }
}

void bfs(struct Graph *graph, int startVertex)
{
    struct node *queue = NULL;
    graph->visited[startVertex] = 1;
    enqueue(&queue, startVertex);

    while (!isEmpty(queue))
    {
        printQueue(queue);
        int currentVertex = dequeue(&queue);
        printf("Visited %d ", currentVertex);

        struct node *temp = graph->adjLists[currentVertex];

        while (temp)
        {
            int adjVertex = temp->vertex;

            if (graph->visited[adjVertex] == 0)
```

```
            {
                graph->visited[adjVertex] = 1;
                enqueue(&queue, adjVertex);
            }
            temp = temp->next;
        }
    }
}

int isEmpty(struct node *queue)
{
    return queue == NULL;
}

void enqueue(struct node **queue, int value)
{
    struct node *newNode = createNode(value);
    if (isEmpty(*queue))
    {
        *queue = newNode;
    }
    else
    {
        struct node *temp = *queue;
        while (temp->next)
        {
            temp = temp->next;
        }
        temp->next = newNode;
    }
}

int dequeue(struct node **queue)
{
    int nodeData = (*queue)->vertex;
    struct node *temp = *queue;
    *queue = (*queue)->next;
    free(temp);
    return nodeData;
}
```

```
void printQueue(struct node *queue)
{
    while (queue)
    {
        printf("%d ", queue->vertex);
        queue = queue->next;
    }
    printf("\n");
}

int main(void)
{
    struct Graph *graph = createGraph(6);
    printf("\nWhat do you want to do?\n");
    printf("1. Add edge\n");
    printf("2. Print graph\n");
    printf("3. BFS\n");
    printf("4. Exit\n");
    int choice;
    scanf("%d", &choice);
    while (choice != 4)
    {
        if (choice == 1)
        {
            int src, dest;
            printf("Enter source and destination: ");
            scanf("%d %d", &src, &dest);
            addEdge(graph, src, dest);
        }
        else if (choice == 2)
        {
            printGraph(graph);
        }
        else if (choice == 3)
        {
            int startVertex;
            printf("Enter starting vertex: ");
            scanf("%d", &startVertex);
            bfs(graph, startVertex);
```

```
        }
        else
        {
            printf("Invalid choice\n");
        }
        printf("What do you want to do?\n");
        printf("1. Add edge\n");
        printf("2. Print graph\n");
        printf("3. BFS\n");
        printf("4. Exit\n");
        scanf("%d", &choice);
    }
    return 0;
}
```

**Output:**

```
What do you want to do?
1. Add edge
2. Print graph
3. BFS
4. Exit
1
Enter source and destination: 0 1
What do you want to do?
1. Add edge
2. Print graph
3. BFS
4. Exit
1
Enter source and destination: 0 2
What do you want to do?
1. Add edge
2. Print graph
3. BFS
4. Exit
1
Enter source and destination: 1 2
What do you want to do?
1. Add edge
2. Print graph
```

```
3. BFS
4. Exit
1
Enter source and destination: 2 3
What do you want to do?
1. Add edge
2. Print graph
3. BFS
4. Exit
2

 Adjacency list of vertex 0
 2 -> 1 ->

 Adjacency list of vertex 1
 2 -> 0 ->

 Adjacency list of vertex 2
 3 -> 1 -> 0 ->

 Adjacency list of vertex 3
 2 ->

 Adjacency list of vertex 4


 Adjacency list of vertex 5

What do you want to do?
1. Add edge
2. Print graph
3. BFS
4. Exit
3
Enter starting vertex: 0
0
Visited 0 2 1
Visited 2 1 3
Visited 1 3
Visited 3
```

```
What do you want to do?
1. Add edge
2. Print graph
3. BFS
4. Exit
4
```

## DFS Program in C

```
// dfs program in C
#include <stdio.h>
#include <stdlib.h>

struct node {
  int vertex;
  struct node* next;
};

struct node* createNode(int v);

struct Graph {
  int totalVertices;
  int* visited;
  struct node** adjLists;
};

void DFS(struct Graph* graph, int vertex) {
  struct node* adjList = graph->adjLists[vertex];
  struct node* temp = adjList;
```

```
  graph->visited[vertex] = 1;
  printf("%d -> ", vertex);

  while (temp != NULL) {
    int connectedVertex = temp->vertex;

    if (graph->visited[connectedVertex] == 0) {
      DFS(graph, connectedVertex);
    }
    temp = temp->next;
  }
}


struct node* createNode(int v) {
  struct node* newNode = malloc(sizeof(struct node));
  newNode->vertex = v;
  newNode->next = NULL;
  return newNode;
}

struct Graph* createGraph(int vertices) {
  struct Graph* graph = malloc(sizeof(struct Graph));
  graph->totalVertices = vertices;

  graph->adjLists = malloc(vertices * sizeof(struct node*));
```

```
  graph->visited = malloc(vertices * sizeof(int));

  int i;
  for (i = 0; i < vertices; i++) {
    graph->adjLists[i] = NULL;
    graph->visited[i] = 0;
  }
  return graph;
}

void addEdge(struct Graph* graph, int src, int dest) {
  struct node* newNode = createNode(dest);
  newNode->next = graph->adjLists[src];
  graph->adjLists[src] = newNode;

  newNode = createNode(src);
  newNode->next = graph->adjLists[dest];
  graph->adjLists[dest] = newNode;
}

void displayGraph(struct Graph* graph) {
  int v;
  for (v = 1; v < graph->totalVertices; v++) {
    struct node* temp = graph->adjLists[v];
    printf("\n%d =>  ", v);
    while (temp) {
      printf("%d, ", temp->vertex);
      temp = temp->next;
```

```
    }
    printf("\n");
  }
  printf("\n");
}

int main() {
  struct Graph* graph = createGraph(8);
  addEdge(graph, 1, 5);
  addEdge(graph, 1, 2);
  addEdge(graph, 1, 3);
  addEdge(graph, 3, 6);
  addEdge(graph, 2, 7);
  addEdge(graph, 2, 4);

  printf("\nThe Adjacency List of the Graph is:");
  displayGraph(graph);

  printf("\nDFS traversal of the graph: \n");
  DFS(graph, 1);

  return 0;
}
```

# output

```
The Adjacency List of the Graph is:
1 =>  3, 2, 5,
2 =>  4, 7, 1,
```

3 =>  6, 1,

4 =>  2,

5 =>  1,

6 =>  3,

7 =>  2,

DFS traversal of the graph:

1 -> 3 -> 6 -> 2 -> 4 -> 7 -> 5