# All 8 Programs - Plain Code

## Program 1: Array Operations

void main() {  
 int arr[SIZE], n, ch;  
 printf("Enter size of Array: ");  
 scanf("%d", &n);  
 printf("Enter Array Elements:\n");  
 for (int i = 0; i < n; i++)  
 scanf("%d", &arr[i]);  
  
 while (1) {  
 printf("\nMenu:\n1. Insert\n2. Delete\n3. Display\n4. Exit\n");  
 scanf("%d", &ch);  
 switch (ch) {  
 case 1: insert(arr, &n); break;  
 case 2: delete(arr, &n); break;  
 case 3: display(arr, n); break;  
 case 4: exit(0);  
 default: printf("Invalid choice\n");  
 }  
 }  
}  
  
void insert(int arr[], int \*n) {  
 int i, pos, ele;  
 printf("Enter Element to Insert: ");  
 scanf("%d", &ele);  
 printf("Enter Position: ");  
 scanf("%d", &pos);  
  
 for (i = (\*n) - 1; i >= pos - 1; i--) {  
 arr[i + 1] = arr[i];  
 }  
 arr[pos - 1] = ele;  
 \*n = \*n + 1;  
}  
  
void delete(int arr[], int \*n) {  
 int i, pos;  
 printf("Enter Position to Delete: ");  
 scanf("%d", &pos);  
  
 for (i = pos - 1; i < (\*n) - 1; i++) {  
 arr[i] = arr[i + 1];  
 }  
 \*n = \*n - 1;  
}  
  
void display(int arr[], int n) {  
 for (int i = 0; i < n; i++) {  
 printf("%d ", arr[i]);  
 }  
 printf("\n");  
}

## Program 2: Stack Operations

void main() {  
 int stack[SIZE], top = -1, ch;  
 while (1) {  
 printf("\nMenu:\n1. Push\n2. Pop\n3. Display\n4. Exit\n");  
 scanf("%d", &ch);  
 switch (ch) {  
 case 1: push(stack, &top, SIZE); break;  
 case 2: pop(stack, &top); break;  
 case 3: display(stack, top); break;  
 case 4: exit(0);  
 default: printf("Invalid choice\n");  
 }  
 }  
}  
  
void push(int stack[], int \*top, int max) {  
 if (\*top == max - 1) {  
 printf("Stack Overflow\n");  
 } else {  
 int ele;  
 printf("Enter Element to Push: ");  
 scanf("%d", &ele);  
 stack[++(\*top)] = ele;  
 }  
}  
  
void pop(int stack[], int \*top) {  
 if (\*top == -1) {  
 printf("Stack Underflow\n");  
 } else {  
 printf("Popped Element: %d\n", stack[(\*top)--]);  
 }  
}  
  
void display(int stack[], int top) {  
 if (top == -1) {  
 printf("Stack is Empty\n");  
 } else {  
 printf("Stack Elements:\n");  
 for (int i = top; i >= 0; i--) {  
 printf("%d\n", stack[i]);  
 }  
 }  
}

## Program 3: Infix to Postfix Conversion

void convert(char \*in, char \*post) {  
 char stack[SIZE];  
 int top = -1, i, j = 0;  
 stack[++top] = '#';  
 for (i = 0; in[i] != '\0'; i++) {  
 char sym = in[i];  
 while (stack\_prec(stack[top]) > in\_prec(sym)) {  
 post[j++] = stack[top--];  
 }  
 if (stack\_prec(stack[top]) != in\_prec(sym)) {  
 stack[++top] = sym;  
 } else {  
 top--;  
 }  
 }  
 while (stack[top] != '#') {  
 post[j++] = stack[top--];  
 }  
 post[j] = '\0';  
}  
  
int in\_prec(char sym) {  
 switch (sym) {  
 case '+': case '-': return 1;  
 case '\*': case '/': return 3;  
 case '(': return 9;  
 case ')': return 0;  
 default: return 7;  
 }  
}  
  
int stack\_prec(char sym) {  
 switch (sym) {  
 case '+': case '-': return 2;  
 case '\*': case '/': return 4;  
 case '#': return -1;  
 default: return 8;  
 }  
}

## Program 4: Postfix Evaluation

void evaluate(char \*postfix) {  
 int stack[SIZE], top = -1;  
 for (int i = 0; postfix[i] != '\0'; i++) {  
 if (isdigit(postfix[i])) {  
 stack[++top] = postfix[i] - '0';  
 } else {  
 int b = stack[top--];  
 int a = stack[top--];  
 switch (postfix[i]) {  
 case '+': stack[++top] = a + b; break;  
 case '-': stack[++top] = a - b; break;  
 case '\*': stack[++top] = a \* b; break;  
 case '/': stack[++top] = a / b; break;  
 }  
 }  
 }  
 printf("Result: %d\n", stack[top]);  
}

## Program 5: Circular Queue

void main() {  
 char queue[SIZE];  
 int front = 0, rear = -1, ch;  
 while (1) {  
 printf("\nMenu:\n1. Insert\n2. Delete\n3. Display\n4. Exit\n");  
 scanf("%d", &ch);  
 switch (ch) {  
 case 1: enqueue(queue, &rear, SIZE); break;  
 case 2: dequeue(queue, &front, rear, SIZE); break;  
 case 3: display(queue, front, rear, SIZE); break;  
 case 4: exit(0);  
 }  
 }  
}  
  
void enqueue(char queue[], int \*rear, int max) {  
 if ((\*rear + 1) % max == 0) {  
 printf("Queue Overflow\n");  
 } else {  
 char ele;  
 printf("Enter Element: ");  
 scanf(" %c", &ele);  
 \*rear = (\*rear + 1) % max;  
 queue[\*rear] = ele;  
 }  
}  
  
void dequeue(char queue[], int \*front, int rear, int max) {  
 if (\*front == rear) {  
 printf("Queue Underflow\n");  
 } else {  
 \*front = (\*front + 1) % max;  
 printf("Deleted: %c\n", queue[\*front]);  
 }  
}  
  
void display(char queue[], int front, int rear, int max) {  
 if (front == rear) {  
 printf("Queue is Empty\n");  
 } else {  
 int i = front + 1;  
 while (i != rear) {  
 printf("%c ", queue[i]);  
 i = (i + 1) % max;  
 }  
 printf("\n");  
 }  
}

## Program 6: Doubly Linked List

void main() {  
 struct node \*head = NULL;  
 int choice, data;  
 while (1) {  
 printf("\nMenu:\n1. Insert at End\n2. Insert at Beginning\n3. Delete at End\n4. Delete at Beginning\n5. Display\n6. Exit\n");  
 scanf("%d", &choice);  
 switch (choice) {  
 case 1:   
 printf("Enter Data: ");  
 scanf("%d", &data);  
 insert\_end(&head, data);  
 break;  
 case 2:   
 printf("Enter Data: ");  
 scanf("%d", &data);  
 insert\_beginning(&head, data);  
 break;  
 case 3: delete\_end(&head); break;  
 case 4: delete\_beginning(&head); break;  
 case 5: display(head); break;  
 case 6: exit(0);  
 default: printf("Invalid choice\n");  
 }  
 }  
}  
  
void insert\_end(struct node \*\*head, int data) {  
 struct node \*newNode = malloc(sizeof(struct node));  
 newNode->data = data;  
 newNode->next = NULL;  
 if (\*head == NULL) {  
 newNode->prev = NULL;  
 \*head = newNode;  
 } else {  
 struct node \*temp = \*head;  
 while (temp->next)  
 temp = temp->next;  
 temp->next = newNode;  
 newNode->prev = temp;  
 }  
}  
  
void insert\_beginning(struct node \*\*head, int data) {  
 struct node \*newNode = malloc(sizeof(struct node));  
 newNode->data = data;  
 newNode->prev = NULL;  
 newNode->next = \*head;  
 if (\*head)  
 (\*head)->prev = newNode;  
 \*head = newNode;  
}  
  
void delete\_end(struct node \*\*head) {  
 if (\*head == NULL) {  
 printf("List is Empty\n");  
 return;  
 }  
 struct node \*temp = \*head;  
 while (temp->next)  
 temp = temp->next;  
 if (temp->prev)  
 temp->prev->next = NULL;  
 else  
 \*head = NULL;  
 free(temp);  
}  
  
void delete\_beginning(struct node \*\*head) {  
 if (\*head == NULL) {  
 printf("List is Empty\n");  
 return;  
 }  
 struct node \*temp = \*head;  
 \*head = (\*head)->next;  
 if (\*head)  
 (\*head)->prev = NULL;  
 free(temp);  
}  
  
void display(struct node \*head) {  
 struct node \*temp = head;  
 while (temp) {  
 printf("%d ", temp->data);  
 temp = temp->next;  
 }  
 printf("\n");  
}

## Program 7: Binary Search Tree

void main() {  
 NODE root = NULL;  
 int choice, key;  
 while (1) {  
 printf("\nMenu:\n1. Insert\n2. Traverse\n3. Search\n4. Delete\n5. Exit\n");  
 scanf("%d", &choice);  
 switch (choice) {  
 case 1:   
 printf("Enter Key: ");  
 scanf("%d", &key);  
 root = insert(root, key);  
 break;  
 case 2:   
 printf("Inorder: ");  
 inorder(root);  
 printf("\n");  
 break;  
 case 3:   
 printf("Enter Key to Search: ");  
 scanf("%d", &key);  
 if (search(root, key))  
 printf("Key Found\n");  
 else  
 printf("Key Not Found\n");  
 break;  
 case 4:   
 printf("Enter Key to Delete: ");  
 scanf("%d", &key);  
 root = delete(root, key);  
 break;  
 case 5: exit(0);  
 default: printf("Invalid Choice\n");  
 }  
 }  
}  
  
NODE insert(NODE root, int key) {  
 if (root == NULL) {  
 NODE newNode = malloc(sizeof(struct node));  
 newNode->key = key;  
 newNode->left = newNode->right = NULL;  
 return newNode;  
 }  
 if (key < root->key)  
 root->left = insert(root->left, key);  
 else  
 root->right = insert(root->right, key);  
 return root;  
}  
  
void inorder(NODE root) {  
 if (root) {  
 inorder(root->left);  
 printf("%d ", root->key);  
 inorder(root->right);  
 }  
}  
  
NODE search(NODE root, int key) {  
 if (root == NULL || root->key == key)  
 return root;  
 if (key < root->key)  
 return search(root->left, key);  
 return search(root->right, key);  
}  
  
NODE delete(NODE root, int key) {  
 if (root == NULL)  
 return root;  
 if (key < root->key)  
 root->left = delete(root->left, key);  
 else if (key > root->key)  
 root->right = delete(root->right, key);  
 else {  
 if (root->left == NULL) {  
 NODE temp = root->right;  
 free(root);  
 return temp;  
 } else if (root->right == NULL) {  
 NODE temp = root->left;  
 free(root);  
 return temp;  
 }  
 NODE temp = minValueNode(root->right);  
 root->key = temp->key;  
 root->right = delete(root->right, temp->key);  
 }  
 return root;  
}  
  
NODE minValueNode(NODE node) {  
 NODE current = node;  
 while (current && current->left != NULL)  
 current = current->left;  
 return current;  
}

## Program 8: Graph Operations

void main() {  
 int graph[SIZE][SIZE], n, start, choice;  
 printf("Enter Number of Vertices: ");  
 scanf("%d", &n);  
 printf("Enter Adjacency Matrix:\n");  
 for (int i = 0; i < n; i++)  
 for (int j = 0; j < n; j++)  
 scanf("%d", &graph[i][j]);  
  
 while (1) {  
 printf("\nMenu:\n1. BFS\n2. DFS\n3. Exit\n");  
 scanf("%d", &choice);  
 switch (choice) {  
 case 1:   
 printf("Enter Starting Node: ");  
 scanf("%d", &start);  
 bfs(graph, n, start);  
 break;  
 case 2:   
 printf("Enter Starting Node: ");  
 scanf("%d", &start);  
 dfs(graph, n, start);  
 break;  
 case 3: exit(0);  
 default: printf("Invalid Choice\n");  
 }  
 }  
}  
  
void bfs(int graph[SIZE][SIZE], int n, int start) {  
 int visited[SIZE] = {0}, queue[SIZE], front = 0, rear = -1;  
 queue[++rear] = start;  
 visited[start] = 1;  
 while (front <= rear) {  
 int current = queue[front++];  
 printf("%d ", current);  
 for (int i = 0; i < n; i++) {  
 if (graph[current][i] && !visited[i]) {  
 queue[++rear] = i;  
 visited[i] = 1;  
 }  
 }  
 }  
}  
  
void dfs(int graph[SIZE][SIZE], int n, int start) {  
 static int visited[SIZE] = {0};  
 printf("%d ", start);  
 visited[start] = 1;  
 for (int i = 0; i < n; i++) {  
 if (graph[start][i] && !visited[i])  
 dfs(graph, n, i);  
 }  
}