Practical 1- Postfix Expression Using Stack

```
#include <stdio.h>
#include <ctype.h>
#define MAX 100
int stack[MAX], top = -1;
void push(int item) { stack[++top] = item; }
int pop() { return stack[top--]; }
void EvalPostfix(char postfix[]) {
    for (int i = 0; postfix[i] != ')'; i++) {
        char ch = postfix[i];
        if (isdigit(ch))
            push(ch - '0');
        else {
            int A = pop(), B = pop();
            switch (ch) {
                case '+': push(B + A); break;
                case '-': push(B - A); break;
                case '*': push(B * A); break;
                case '/': push(B / A); break;
            }
        }
    }
    printf("Result: %d\n", pop());
}
int main() {
    char postfix[MAX];
    printf("Enter postfix expression ending with ')':\n");
    scanf("%s", postfix);
    EvalPostfix(postfix);
    return 0;
```

Practical 2- Infix To Postfix

```
#include <stdio.h>
#include <stdlib.h>
#include <ctype.h>
#include <string.h>
#define SIZE 100
char stack[SIZE];
int top = -1;
void push(char item) {
    if (top >= SIZE - 1) {
        printf("\nStack Overflow.\n");
        exit(1);
    }
    stack[++top] = item;
}
char pop() {
    if (top < 0) {
        printf("\nStack Underflow: Invalid Infix Expression.\n");
        exit(1);
    }
    return stack[top--];
}
int is_operator(char symbol) {
    return (symbol == '^' || symbol == '*' || symbol == '/' || symbol == '+' || symbol == '-');
}
int precedence(char symbol) {
    if (symbol == '^') return 3;
    if (symbol == '*' || symbol == '/') return 2;
    if (symbol == '+' || symbol == '-') return 1;
    return 0;
}
```

```
void InfixToPostfix(char infix[], char postfix[]) {

  int i = 0, j = 0;

  char item, x;

  push('(');

  strcat(infix, ")");

  while ((item = infix[i++]) != '\0') {

    if (item == '(') push(item);

    else if (isalnum(item)) postfix[j++] = item;

    else if (is_operator(item)) {

      while (is_operator((x = pop())) &&
precedence(x) >= precedence(item))

        postfix[j++] = x;

      push(x);

      push(item);

    } else if (item == ')') {

      while ((x = pop()) != '(') postfix[j++] = x;

    } else {

      printf("\nInvalid Infix Expression.\n");

      exit(1);

    }

  }

  postfix[j] = '\0';

}

int main() {

  char infix[SIZE], postfix[SIZE];

  printf("Enter an Infix expression:\n");

  scanf("%s", infix);

  InfixToPostfix(infix, postfix);

  printf("Postfix Expression: %s\n", postfix);

  return 0;

}
```

Practical 4- Stack using two queues

```
#include <stdio.h>
#define MAX 3
int q1[MAX], q2[MAX], f1 = -1, r1 = -1, f2 = -1, r2 = -
1;

int isEmpty(int front) { return front == -1; }
void enqueue(int q[], int *front, int *rear, int val) {
  if (*rear == MAX - 1) {
    printf("Stack Overflow\n");
    return;
  }
  if (*front == -1) *front = 0;
  q[++(*rear)] = val;
}
int dequeue(int q[], int *front, int *rear) {
  if (isEmpty(*front)) {
    printf("Stack Underflow\n");
    return -1;
  }
  int val = q[*front];
  if (*front == *rear) *front = *rear = -1;
  else (*front)++;
  return val;
}
void push(int x) { enqueue(q1, &f1, &r1, x); }
void pop() {
  if (isEmpty(f1)) {
    printf("Stack Underflow\n");
    return;
  }
  while (f1 != r1) enqueue(q2, &f2, &r2,
dequeue(q1, &f1, &r1));
  printf("Popped: %d\n", dequeue(q1, &f1, &r1));
  while (!isEmpty(f2)) enqueue(q1, &f1, &r1,
dequeue(q2, &f2, &r2));
}
void display() {
  if (isEmpty(f1)) {
    printf("Stack is empty\n");
    return;
  }
  for (int i = f1; i <= r1; i++) printf("%d ", q1[i]);
  printf("\n");
}
int main() {
  int choice, value;
  while (1) {
    printf("1. Push\n2. Pop\n3. Display\n4.
Exit\nEnter choice: ");
    scanf("%d", &choice);
    switch (choice) {
      case 1: printf("Enter value: "); scanf("%d",
&value); push(value); break;
```

```
        case 2: pop(); break;
        case 3: display(); break;
        case 4: return 0;
        default: printf("Invalid choice\n");
      }
   }
}
```

Practical 5- Queue using two Stacks

```
#include<stdio.h>

#define N 5

int stack1[N], stack2[N], top1 = -1, top2 = -1, count = 0;

void push1(int data) { if(top1 < N-1) stack1[++top1] = data; else printf("\nStack Overflow"); }

int pop1() { return (top1 == -1) ? -1 : stack1[top1--]; }

void push2(int x) { if(top2 < N-1) stack2[++top2] = x; else printf("\nStack Overflow"); }

int pop2() { return (top2 == -1) ? -1 : stack2[top2--]; }


void enqueue(int x) { push1(x); count++; }

void dequeue() {

   if(top1 == -1 && top2 == -1) { printf("\nQueue is empty\n"); return; }

   while(top1 != -1) push2(pop1());

   printf("\nThe dequeued element is %d\n", pop2());

   count--;

   while(top2 != -1) push1(pop2());

}


void display() {

   if(top1 == -1) printf("\nQueue is empty\n");

   else {

      printf("\nQueue elements: ");

      for(int i = 0; i <= top1; i++) printf("%d ", stack1[i]);

      printf("\n");

   }

}

void main() {

   int choice, value;

   while(1) {

      printf("\n1. Enqueue\n2. Dequeue\n3. Display\n4. Exit\nEnter choice: ");

      scanf("%d", &choice);

      switch(choice) {

         case 1:

            if(top1 == N-1) printf("\nQueue is full!");

            else { printf("\nEnter value: "); scanf("%d", &value); enqueue(value); }

            break;

         case 2: dequeue(); break;

         case 3: display(); break;

         case 4: return;

         default: printf("\nInvalid choice");

      }

   }

}
```

Practical 6-Single Linked List

```
#include <stdio.h>

#include <stdlib.h>

struct Node {

   int data;

   struct Node* next;

};

void insertAtBeginning(struct Node** head, int data) {

   struct Node* new_node = (struct Node*)malloc(sizeof(struct Node));

   new_node->data = data;

   new_node->next = *head;

   *head = new_node;

}

void insertAtEnd(struct Node** head, int data) {

   struct Node* new_node = (struct Node*)malloc(sizeof(struct Node)), *last = *head;

   new_node->data = data; new_node->next = NULL;

   if (!*head) { *head = new_node; return; }

   while (last->next) last = last->next;

   last->next = new_node;

}

void deleteNode(struct Node** head, int key) {

   struct Node *temp = *head, *prev;

   if (temp && temp->data == key) { *head = temp->next; free(temp); return; }

   while (temp && temp->data != key) { prev = temp; temp = temp->next; }

   if (!temp) return;

   prev->next = temp->next; free(temp);

}

int searchNode(struct Node* head, int key) {

   while (head) { if (head->data == key) return 1; head = head->next; }

   return 0;

}

void sortList(struct Node** head) {

   struct Node *current, *index;

   int temp;

   for (current = *head; current; current = current->next)

      for (index = current->next; index; index = index->next)

         if (current->data > index->data) { temp = current->data; current->data = index->data; index->data = temp; }

}

void printList(struct Node* head) {

   while (head) { printf("%d ", head->data); head = head->next; }

}

int main() {

   struct Node* head = NULL;

   insertAtEnd(&head, 1);

   insertAtBeginning(&head, 2);

   insertAtBeginning(&head, 3);

   insertAtEnd(&head, 4);

   insertAtBeginning(&head, 5);

   printf("List: "); printList(head);

   deleteNode(&head, 3); printf("\nAfter deletion: "); printList(head);

   int item = 4;

   printf("\n%d is %sfound\n", item, searchNode(head, item) ? "" : "not ");

   sortList(&head); printf("Sorted List: "); printList(head);

   return 0;}
```

Practical 7- Stack using Linked List

```
#include <stdio.h>
#include <stdlib.h>
struct Node {
   int data;
   struct Node* next;};
void push(struct Node** top, int value) {
   struct Node* newNode = (struct Node*)malloc(sizeof(struct Node));
   newNode->data = value;
   newNode->next = *top;
   *top = newNode;}
void pop(struct Node** top) {
   if (!*top) { printf("Underflow!\n"); return; }
   struct Node* temp = *top;
   printf("Popped %d\n", temp->data);
   *top = temp->next;
   free(temp);}
void display(struct Node* top) {
   printf("Stack: ");
   while (top) { printf("%d ", top->data); top = top->next; }
   printf("\n");}
int main() {
   struct Node* top = NULL;
   push(&top, 10); push(&top, 20); push(&top, 30);
   display(top);
   pop(&top);
   display(top);
   return 0;
}
```

Practical 8-BST

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


struct Node {
   int key;
   struct Node *left, *right;
};


struct Node* createNode(int key) {
   struct Node* node = (struct Node*)malloc(sizeof(struct Node));
   node->key = key;
   node->left = node->right = NULL;
   return node;
}


struct Node* insert(struct Node* root, int key) {
   if (!root) return createNode(key);
   if (key < root->key) root->left = insert(root->left, key);
   else if (key > root->key) root->right = insert(root->right, key);
   return root;}
int findMin(struct Node* root) { while (root->left) root = root->left; return root->key; }
int findMax(struct Node* root) { while (root->right) root = root->right; return root->key; }
int search(struct Node* root, int key) {
   if (!root) return 0;
   return root->key == key || search(key < root->key ? root->left : root->right, key);}
int main() {
   struct Node* root = NULL;
```

```
  int keys[] = {50, 30, 70, 20, 40, 60, 80};
  for (int i = 0; i < 7; i++) root = insert(root, keys[i]);
  printf("Min: %d\nMax: %d\n", findMin(root), findMax(root));
  printf("Key 40 is %sfound.\n", search(root, 40) ? "" : "not ");
  return 0;}
```

Peactical 9- HASHING

```
#include <stdio.h>
#include <stdlib.h>
#define SIZE 10
struct Node {
  int key;
  struct Node* next;
} *hashTable[SIZE] = {NULL};
int hash(int key) { return key % SIZE; }
void insert(int key) {
  int idx = hash(key);
  struct Node* node = (struct Node*)malloc(sizeof(struct Node));
  node->key = key;
  node->next = hashTable[idx];
  hashTable[idx] = node;
}
int search(int key) {
  struct Node* temp = hashTable[hash(key)];
  while (temp && temp->key != key) temp = temp->next;
  return temp != NULL;
}
void display() {
  for (int i = 0; i < SIZE; i++) {
    printf("%d: ", i);
    for (struct Node* t = hashTable[i]; t; t = t->next) printf("%d -> ", t->key);
    printf("NULL\n");
  }
}
int main() {
  insert(10); insert(20); insert(15); insert(25);
  display();
  printf("Search 15: %s\n", search(15) ? "Found" : "Not Found");
  return 0;
}
```

Practical 10-SORTING

Insertion Sort

```
#include <stdio.h>
void insertionSort(int arr[], int n) {
  for (int i = 1; i < n; i++) {
    int key = arr[i], j = i - 1;
    while (j >= 0 && arr[j] > key) arr[j + 1] = arr[j--];
    arr[j + 1] = key;
  }
}
int main() {
  int arr[] = {5, 2, 9, 1, 5, 6}, n = sizeof(arr) / sizeof(arr[0]);
  insertionSort(arr, n);
  for (int i = 0; i < n; i++) printf("%d ", arr[i]);
  return 0;
}
```

Merge Sort:

```
#include <stdio.h>

void merge(int arr[], int l, int m, int r) {
    int i = 0, j = 0, k = l, n1 = m - l + 1, n2 = r -
m, L[n1], R[n2];
    for (i = 0; i < n1; i++) L[i] = arr[l + i];
    for (j = 0; j < n2; j++) R[j] = arr[m + 1 + j];
    for (i = 0, j = 0; i < n1 && j < n2; k++)
arr[k] = (L[i] <= R[j]) ? L[i++] : R[j++];
    while (i < n1) arr[k++] = L[i++];
    while (j < n2) arr[k++] = R[j++];
}
void mergeSort(int arr[], int l, int r) {
    if (l < r) {
        int m = l + (r - l) / 2;
        mergeSort(arr, l, m);
        mergeSort(arr, m + 1, r);
        merge(arr, l, m, r);
    }
}
int main() {
    int arr[] = {12, 11, 13, 5, 6, 7}, n =
sizeof(arr) / sizeof(arr[0]);
    mergeSort(arr, 0, n - 1);
    for (int i = 0; i < n; i++) printf("%d ", arr[i]);
    return 0;
}
```

Quick Sort:

```
#include <stdio.h>
int partition(int arr[], int low, int high) {
    int pivot = arr[high], i = low - 1, temp;
    for (int j = low; j < high; j++)
        if (arr[j] < pivot) temp = arr[++i], arr[i] =
arr[j], arr[j] = temp;
    temp = arr[i + 1], arr[i + 1] = arr[high],
arr[high] = temp;
    return i + 1;
}
void quickSort(int arr[], int low, int high) {
    if (low < high) {
        int pi = partition(arr, low, high);
        quickSort(arr, low, pi - 1);
        quickSort(arr, pi + 1, high);
    }
}
int main() {
    int arr[] = {10, 7, 8, 9, 1, 5}, n = sizeof(arr) /
sizeof(arr[0]);
    quickSort(arr, 0, n - 1);
    for (int i = 0; i < n; i++) printf("%d ", arr[i]);
    return 0;
}
```