Stacks
Day6&7
15-03-2025 / 16-03-2025

Simple Array Implementation

```
class Stack {
private:
    int* arr;
    int capacity;
    int top;

public:
    Stack(int size) {
        capacity = size;
        arr = new int[capacity];
        top = -1;
    }

    bool isEmpty() {
        return top == -1;
    }

    bool isFull() {
        return top == capacity - 1;
    }

    int size() {
        return top + 1;
    }

    void push(int value) {
        if(isFull()) {
            cout << "Stack overflow!\n";
            return;
        }
        arr[++top] = value;
    }

    int pop() {
        if(isEmpty()) {
            cout << "Stack underflow!\n";
            return -1;
        }
        // return arr[top--];    // can be written more descriptively
as :
        int value = arr[top];
```

```
        top--;
        return value;
    }

    int peek() {
        if(isEmpty()) {
            cout << "Empty stack, no element to peek.\n";
        }
        return arr[top];
    }

    void deleteStack() {
        delete[] arr;
        arr = nullptr;
        capacity = 0;
        top = -1;
        cout << "Stack deleted successfully.\n";
    }

    ~Stack() {
        if(arr) {
            delete[] arr;
        }
    }
}
```

Dynamic Array Implementation

```
class DynamicStack {
private:
    int* arr;
    int capacity;
    int top;


    void resize() {
        int newCapacity = capacity * 2;
        int* newArr = new int[newCapacity];
        for(int i = 0; i < capacity; i++) {
            newArr[i] = arr[i];
        }
        delete[] arr;
        arr = newArr;
        capacity = newCapacity;
        cout << "Stack capacity doubled to " << capacity << ".\n";
    }
```

```
public:

    DynamicStack(int initialCapacity) {
        capacity = initialCapacity;
        arr = new int[capacity];
        top = -1;
    }

    bool isEmpty() {
        return top == -1;
    }

    bool isFull() {
        return top == capacity - 1;
    }

    int size() {
        return top + 1;
    }

    void push(int value) {
        if(isFull()) {
            resize();
        }
        arr[++top] = value;
    }

    int pop() {
        if(isEmpty()) {
            cout << "Stack underflow!\n";
            return -1;
        }
        // return arr[top--];    // can be written more descriptively
as :
        int value = arr[top];
        top--;
        return value;
    }

    int peek() {
        if(isEmpty()) {
            cout << "Empty stack, no element to peek.\n";
        }
        return arr[top];
```

```
    }

    void deleteStack() {
        delete[] arr;
        arr = nullptr;
        capacity = 0;
        top = -1;
        cout << "Stack deleted successfully.\n";
    }

    ~DynamicStack() {
        if(arr) {
            delete[] arr;
        }
    }
}
```

Linked List Implementation

```
struct Node {
    int data;
    Node* next;
    Node(int value) : data(value), next(nullptr) {}
};

class LinkedListStack {

private:
    Node* top;

public:
    LinkedListStack() : top(nullptr) {}

    bool isEmpty() {
        return top == nullptr;
    }

    int size() {
        int count = 0;
        Node* current = top;
        while(current) {
            count++;
            current = current->next;
        }
        return count;
    }
```

```
    void push(int value) {
        Node* newNode = new Node(value);
        newNode->next = top;
        top = newNode;
    }

    int pop() {
        if(isEmpty()) {
            cout << "Stack underflow!\n";
            return -1;
        }
        int poppedValue = top->data;
        Node* temp = top;
        top = top->next;
        delete temp;
        return poppedValue;
    }

    int peek() {
        if(isEmpty()) {
            cout << "Stack underflow!\n";
            return -1;
        }
        return top->data;
    }

    void deleteStack() {
        while(top) {
            Node* temp = top;
            top = top->next;
            delete temp;
        }
    }

    ~LinkedListStack() {
        deleteStack();
    }
}
```

Problems

Using stacks to check balancing of symbols/ parentheses

```
int matchSymbol(char a, char b) {
```

```
    if(a == '[' && b == ']') {
        return 1;
    }
    else if(a == '{' && b == '}') {
        return 1;
    }
    else if(a == '(' && b == ')') {
        return 1;
    }
    return 0;
}

int checkExpression(string expression) {
    int count;
    char temp;
    struct Stack *stk = createStack(5);
    for(count = 0; count < expression.length(); count++) {
        if(expression[count] == '(' || expression[count] == '{' ||
expression[count] == '[') {
            push(stk, expression[count]);
        }
        if(expression[count] == ')' || expression[count] == '}' ||
expression[count] == ']') {
            if(isEmpty(stk)) {
                cout << "The right symbols are more than the left
symbols.\n";
                return 0;
            } else {
                temp = pop(stk);
                if(!matchSymbol(temp, expression[count])) {
                    cout << "Mismatched symbols.\n";
                    return 0;
                }
            }
        }
    }
    if(isEmpty(stk)) {
        cout << "Expression is balanced.\n";
        return 1;
    } else {
        cout << "Expression is unbalanced.\n";
        return 0;
    }
}
```

Infix to postfix conversion

```
int precedence(char op) {
    if (op == '+' || op == '-') return 1;
    if (op == '*' || op == '/') return 2;
    if (op == '^') return 3; // Right associative
    return 0;
}

bool isOperator(char ch) {
    return (ch == '+' || ch == '-' || ch == '*' || ch == '/' || ch ==
'^');
}

string infixToPostfix(const string& infix) {
    stack<char> s;
    string postfix;

    for(char ch : infix) {
        if(isalnum(ch)) {
            postfix += ch;
        } else if(ch == '(') {
            s.push(ch);
        } else if(ch == ')') {
            while(!s.empty() && s.top() != '(') {
                postfix += s.top;
                s.pop();
            }
            s.pop();
        } else if(isOperator(ch)) {
            while(!s.empty() && precedence(s.top()) >=
precedence(ch)) {
                postfix += s.top();
                s.pop();
            }
            s.push(ch);
        }
    }

    while(!s.empty()) {
        postfix += s.top();
        s.pop();
    }
    return postfix;
}
```

Postfix Evaluation

```
int postfixEvaluation(string expression) {
    struct Stack* stk = createStack(expression.length());
    int i;
    for(i = 0; expression[i]; i++) {
        if(isdigit(expression[i])) {
            push(stk, expression[i]-'0');
        } else {
            int topElement = pop(stk);
            int secondTopElement = pop(stk);
            switch (expression[i]) {
                case '+': push(stk, secondTopElement + topElement);
break;
                case '-': push(stk, secondTopElement - topElement);
break;
                case '*': push(stk, secondTopElement * topElement);
break;
                case '/': push(stk, secondTopElement + topElement);
break;
            }
        }
    }
    return pop(stk);
}
```

Designing stack such that getMinimum() is O(1)

```
// first the basic stack implemetation

struct AdvancedStack {
    struct Stack *elementStack;
    struct Stack *minStack;
};

int isEmptyA(struct AdvancedStack *S) {
    return (S->elementStack->top == -1);
}

int sizeA(struct AdvancedStack *S) {
    return (S->elementStack->top+1);
}

int isFullA(struct AdvancedStack *S) {
    return (S->elementStack->top == S->elementStack->capacity-1);
}
```

```
void pushA(struct AdvancedStack *S, int data) {
    push(S->elementStack, data);
    if(isEmpty(S->minStack) || peek(S->minStack) >= data) {
        push(S->minStack, data);
    } else {
        push(S->elementStack, peek(S->minStack));
    }
}

int popA(struct AdvancedStack *S) {
    int temp;
    if(isEmpty(S->elementStack)) {
        return INT_MIN;
    }
    temp = peek(S->elementStack);
    if(peek(S->minStack) == pop(S->elementStack)) {
        pop(S->minStack);
    }
    return temp;
}

int peekA(struct AdvancedStack *S) {
    return peek(S->elementStack);
}

int getMinimum(struct AdvancedStack *S) {
    return peek(S->minStack);
}

struct AdvancedStack * createAdvancedStack(int capacity) {
    if(!S) {
        return NULL;
    }
    S->elementStack = createStack(capacity);
    S->minSTack = createStack(capacity);
    return S;
}

void deleteStackA(struct AdvancedStack *S) {
    if(S) {
        deleteStackA(S->elementStack);
        deleteStackA(S->minStack);
        free(S);
    }
}
```

Check if an array is a palindrome

```
bool isArrayPalindrome(int arr[], int n) {
    Stack stk(n);
    for(int i = 0; i < n/2; i++) {
        stk.push(arr[i]);
    }
    int start = (n % 2) ? n/2 : n/2 + 1;  //start checking from the
middle for odd sized arrays
    for(int i = start; i < n; i++) {
        if(stk.pop() != arr[i]) {
            return false;
        }
    }
    return true;
}
```

Check if a LL is a palindrome

```
struct Node {
    int data;
    Node* next;
    Node(int val) : data(val), next(nullptr) {}
};

bool isLLPalindrome(Node* head) {
    if(!head) {
        return true;
    }
    int n = 0;
    Node* temp = head;
    while(temp) {
        n++;
        temp = temp->head;
    }
    Stack stk(n);
    temp = head;
    for(int i = 0; i < n/2; i++) {
        stk.push(temp->data);
        temp = temp->next;
    }
    if(n % 2 != 0) {
        temp = temp->next;
    }
    while(temp) {
        if(stk.pop() != temp->data) {
```

```
            return false;
        }
        temp = temp->next;
    }
    return true;
}
```

Reversing elements of a stack using stack operations

```
void insertAtBottom(Stack &stk, int item) {
    if(stk.isEmpy()) {
        stk.push(item);
        return;
    }
    int topElement = stk.pop();
    insertAtBottom(stk, item);
    stk.push(topElement);
}

void reverseStack(Stack& stk) {
    if(stk.isEmpty()) {
        return;
    }
    int topElement = stk.pop();
    reverseStack(stk);
    insertAtBottom(stk, topElement);
}
```

Implementing two stacks using one array

```
// idea : divide array into two parts : stack 1 starts from beginning
and grows towards the right ; stack2 starts from the end and grows
towards the left

class TwoStacks {
    private :
        int* arr;
        int size;
        int top1;
        int top2;

    public:
        TwoStacks(int n) {
            size = n;
            arr = new int[n];
            top1 = -1;
            top2 = size;
```

```
}

~TwoStacks() {
    delete[] arr;
}

void push1(int data) {
    if (top1 + 1 < top2) {
        arr[++top1] = data;
    } else {
        cout << "Stack Overflow in Stack 1" << endl;
    }
}

void push2(int data) {
    if (top1 + 1 < top2) {
        arr[--top2] = data;
    } else {
        cout << "Stack Overflow in Stack 2" << endl;
    }
}

int pop1() {
    if (top1 >= 0) {
        return arr[top1--];
    } else {
        cout << "Stack Underflow in Stack 1" << endl;
        return INT_MIN;
    }
}

int pop2() {
    if (top2 < size) {
        return arr[top2++];
    } else {
        cout << "Stack Underflow in Stack 2" << endl;
        return INT_MIN;
    }
}

int peek1() {
    if (top1 >= 0) {
        return arr[top1];
    } else {
        cout << "Stack 1 is Empty" << endl;
```

```
                return INT_MIN;
            }
        }

        int peek2() {
            if (top2 < size) {
                return arr[top2];
            } else {
                cout << "Stack 2 is Empty" << endl;
                return INT_MIN;
            }
        }

};
```

Given an array A, find maximum of j-i subjected to the constraint A[i] < A[j] ?

```
int maxIndexDifference(vector<int>& A) {
    int n = A.size();
    stack<int> minStack;
    for(int i = 0; i < n; i++) {
        if(minStack.empty() || A[minStack.top()] > A[i]) {
            minStack.push(i);
        }
    }
    int maxDiff = 0;
    for(int j = n - 1; j >= 0; j--) {
        while(!minStack.empty() && A[minStack.top()] < A[j]) {
            maxDiff = max(maxDiff, j - minStack.top());
            minStack.pop();
        }
    }
    return maxDiff;
}
```

Recursively remove all adjacent duplicates

```
string removeAdjacentDuplicates(string str) {
    stack<char> charStack;
    for(char ch : str) {
        if(charStack.empty() || charStack.top() != ch) {
            charStack.push(ch);
        } else {
            charStack.pop();
        }
    }
    string result ="";
```

```
    while(!charStack.empty()) {
        result = charStack.top() + result;
        charStack.pop();
    }
    return result;
}

string recursivelyRemoveAdjDup(string str) {
    string reducedStr = removeAdjacentDuplicates(str);
    if(reducedStr != str) {
        return recursivelyRemoveAdjDup(reducedStr);
    }
    return reducedStr;
}
```

Given an array of elements, replace every element with nearest greater element on the right of that element

```
vector<int> nearestGreaterOnRight(vector<int>& arr) {
    int n = arr.size();
    vector<int> result(n, -1);
    stack<int> s;

    for (int i = n - 1; i >= 0; i--) {
        while (!s.empty() && s.top() <= arr[i]) {
            s.pop();
        }
        if (!s.empty()) {
            result[i] = s.top();
        }
        s.push(arr[i]);
    }

    return result;
}
```

Min no of swaps to make the string balanced

```
int minSwapsToBalance(string str) {
    stack<char> s;

    for (char ch : str) {
        if (ch == '[') {
            s.push(ch);
        } else if (!s.empty() && s.top() == '[') {
            s.pop();
```

```
        }
    }

    int unbalanced = s.size();
    return (unbalanced + 1) / 2;
}
```

Check if each successive pair of integers is consecutive or not

```
bool arePairsConsecutive(stack<int>& s) {
    stack<int> tempStack;
    bool isConsecutive = true;

    while (!s.empty()) {
        tempStack.push(s.top());
        s.pop();
    }

    while (tempStack.size() > 1) {
        int first = tempStack.top();
        tempStack.pop();
        int second = tempStack.top();
        tempStack.pop();

        if (abs(first - second) != 1) {
            isConsecutive = false;
        }

        s.push(first);
        s.push(second);
    }

    while (!tempStack.empty()) {
        s.push(tempStack.top());
        tempStack.pop();
    }

    return isConsecutive;
}
```