Data Structures Notes

by Zak
github: https://github.com/arrhenius975

January 2, 2025

Contents

Dedicated to my one and only lovely life-partner, best-friend and the love of my life @r0s3-V3lv3t (https://github.com/r0s3-V3lv3t)

1 Introduction

This document provides a structured overview of key data structures and algorithms (DSA) with their theoretical concepts, mathematical interludes, and C language implementations. The topics covered include linked lists, stacks, queues, and expression evaluation, among others.

2 Linked List

2.1 Introduction

A linked list is a linear data structure in which elements, called nodes, are connected using pointers. Each node contains two fields:

- **Data**: The value stored in the node.
- **Next**: A pointer to the next node in the list.

2.2 Types of Linked Lists

- Singly Linked List
- Doubly Linked List
- Circular Linked List
- Doubly Circular Linked List

2.3 Mathematical Interlude

Given a node n with value x, the complexity of insertion, deletion, and traversal depends on the pointer manipulation.

2.4 Implementation in C

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

// Define the Node structure
struct Node {
    int data;
    struct Node* next;
};
```

```c

// Function to create a new node
struct Node* createNode(int data) {
    struct Node* newNode = (struct Node*)malloc(sizeof(struct
     Node));
    newNode->data = data;
    newNode->next = NULL;
    return newNode;
}

// Function to insert at the beginning
void insertAtBeginning(struct Node** head, int data) {
    struct Node* newNode = createNode(data);
    newNode->next = *head;
    *head = newNode;
}

// Function to insert at the end
void insertAtEnd(struct Node** head, int data) {
    struct Node* newNode = createNode(data);
    if (*head == NULL) {
        *head = newNode;
        return;
    }
    struct Node* temp = *head;
    while (temp->next != NULL) {
        temp = temp->next;
    }
    temp->next = newNode;
}

// Function to insert after a specific node
void insertAfter(struct Node* prevNode, int data) {
    if (prevNode == NULL) {
        printf("The given previous node cannot be NULL\n");
        return;
    }
    struct Node* newNode = createNode(data);
    newNode->next = prevNode->next;
    prevNode->next = newNode;
}

// Function to delete a node by key
void deleteNode(struct Node** head, int key) {
    struct Node *temp = *head, *prev;

    // If head node itself holds the key
    if (temp != NULL && temp->data == key) {
        *head = temp->next; // Change head
        free(temp); // Free old head
        return;
    }

    // Search for the key to be deleted
    while (temp != NULL && temp->data != key) {
        prev = temp;
        temp = temp->next;
    }

    // If the key was not present
    if (temp == NULL) {
        printf("Key not found\n");
        return;
    }

    // Unlink the node and free memory
    prev->next = temp->next;
    free(temp);
}

// Function to print the linked list
void printList(struct Node* head) {
    while (head != NULL) {
        printf("%d -> ", head->data);
        head = head->next;
    }
    printf("NULL\n");
}

// Main function to test the Linked List
int main() {
    struct Node* head = NULL;

    // Insert nodes
    insertAtBeginning(&head, 10);
    insertAtBeginning(&head, 20);
    insertAtEnd(&head, 30);
    insertAfter(head->next, 25);

    // Print the list
    printf("Linked list: ");
    printList(head);

    // Delete a node
    deleteNode(&head, 20);
    printf("After deletion: ");
    printList(head);

    return 0;
}
```
Listing 1: Linked List Implementation in C

2.5 Floyd's Cycle Detection Algorithm

- To detect loops in a linked list:
- 1. Use two pointers (slow and fast).
- 2. Move slow by one step and fast by two steps.
- 3. If they meet, there's a loop.

2.6 Implementation in C

```c
int detectLoop(struct Node* head) {
    struct Node *slow = head, *fast = head;
    while (fast != NULL && fast->next != NULL) {
        slow = slow->next;
        fast = fast->next->next;
        if (slow == fast) return 1;  // Loop detected
    }
    return 0;  // No loop
}
```
Listing 2: Floyd's Cycle Detection for Linked List Implementation in C

3 Stack

3.1 Introduction

A stack is a linear data structure that follows the Last In, First Out (LIFO) principle. Common operations include:

- **Push**: Insert an element.
- **Pop**: Remove the top element.
- **Peek**: Retrieve the top element without removing it.

3.2 Mathematical Interlude

Stack operations can be mathematically modeled as transitions in a state machine with stack pointers.

3.3 Implementation in C

```c
#include <stdio.h>
#include <stdlib.h>
#define MAX 100  // Maximum size of the stack

// Global variables
int stack[MAX];
int top = -1;

// Function to push an element onto the stack
void push(int value) {
    if (top == MAX - 1) {
        printf("Stack Overflow\n");
        return;
    }
    stack[++top] = value;
    printf("Pushed %d onto the stack\n", value);
}

// Function to pop an element from the stack
int pop() {
    if (top == -1) {
        printf("Stack Underflow\n");
        return -1;
    }
    return stack[top--];
}

```

```c
// Function to peek (view the top element of the stack)
int peek() {
    if (top == -1) {
        printf("Stack is empty\n");
        return -1;
    }
    return stack[top];
}

// Function to display the stack elements
void display() {
    if (top == -1) {
        printf("Stack is empty\n");
        return;
    }
    printf("Stack elements: ");
    for (int i = top; i >= 0; i--) {
        printf("%d ", stack[i]);
    }
    printf("\n");
}

// Main function to test the stack
int main() {
    push(10);
    push(20);
    push(30);
    display();
    printf("Popped: %d\n", pop());
    printf("Top element: %d\n", peek());
    display();
    return 0;
}
```

Listing 3: Stack Implementation in C using arrays

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

// Define the Node structure
struct Node {
    int data;
    struct Node* next;
};

// Function to create a new node
struct Node* createNode(int data) {
    struct Node* newNode = (struct Node*)malloc(sizeof(
   struct Node));
```

```c
    newNode->data = data;
    newNode->next = NULL;
    return newNode;
}

// Function to push an element onto the stack
void push(struct Node** top, int data) {
    struct Node* newNode = createNode(data);
    newNode->next = *top;
    *top = newNode;
    printf("Pushed %d onto the stack\n", data);
}

// Function to pop an element from the stack
int pop(struct Node** top) {
    if (*top == NULL) {
        printf("Stack Underflow\n");
        return -1;
    }
    struct Node* temp = *top;
    int poppedValue = temp->data;
    *top = temp->next;
    free(temp);
    return poppedValue;
}

// Function to peek (view the top element of the stack)
int peek(struct Node* top) {
    if (top == NULL) {
        printf("Stack is empty\n");
        return -1;
    }
    return top->data;
}

// Function to display the stack elements
void display(struct Node* top) {
    if (top == NULL) {
        printf("Stack is empty\n");
        return;
    }
    printf("Stack elements: ");
    while (top != NULL) {
        printf("%d ", top->data);
        top = top->next;
    }
    printf("\n");
}

```

```c
// Main function to test the stack
int main() {
    struct Node* stack = NULL;

    push(&stack, 10);
    push(&stack, 20);
    push(&stack, 30);
    display(stack);
    printf("Popped: %d\n", pop(&stack));
    printf("Top element: %d\n", peek(stack));
    display(stack);
    return 0;
}
```

Listing 4: Stack Implementation in C using linked list

4 Queue

4.1 Introduction

A queue is a linear data structure that follows the First In, First Out (FIFO) principle. Types of queues include:

- Simple Queue
- Circular Queue
- Priority Queue

4.2 Mathematical Interlude

The complexity of enqueue and dequeue operations is $O(1)$ for simple and circular queues.

4.3 Implementation in C

```c
#include <stdio.h>
#include <stdlib.h>
#define MAX 100  // Maximum size of the queue

int queue[MAX];
int front = -1, rear = -1;

// Function to check if the queue is empty
int isEmpty() {
    return front == -1 || front > rear;
}

// Function to check if the queue is full
int isFull() {
    return rear == MAX - 1;
}

// Function to enqueue an element
void enqueue(int value) {
    if (isFull()) {
        printf("Queue Overflow\n");
        return;
    }
    if (front == -1) front = 0; // Initialize front
    queue[++rear] = value;
    printf("Enqueued: %d\n", value);
}
```

```c

// Function to dequeue an element
int dequeue() {
    if (isEmpty()) {
        printf("Queue Underflow\n");
        return -1;
    }
    int value = queue[front++];
    if (front > rear) front = rear = -1; // Reset the
   queue if empty
    return value;
}

// Function to display the queue
void display() {
    if (isEmpty()) {
        printf("Queue is empty\n");
        return;
    }
    printf("Queue elements: ");
    for (int i = front; i <= rear; i++) {
        printf("%d ", queue[i]);
    }
    printf("\n");
}

// Main function to test the Queue
int main() {
    enqueue(10);
    enqueue(20);
    enqueue(30);
    display();
    printf("Dequeued: %d\n", dequeue());
    display();
    return 0;
}
```

Listing 5: Queue Implementation in C using arrays

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

// Define the Node structure
struct Node {
    int data;
    struct Node* next;
};

// Define the front and rear of the queue
```

```c
struct Node* front = NULL;
struct Node* rear = NULL;

// Function to check if the queue is empty
int isEmpty() {
    return front == NULL;
}

// Function to enqueue an element
void enqueue(int value) {
    struct Node* newNode = (struct Node*)malloc(sizeof(
   struct Node));
    newNode->data = value;
    newNode->next = NULL;
    if (rear == NULL) {
        front = rear = newNode;
    } else {
        rear->next = newNode;
        rear = newNode;
    }
    printf("Enqueued: %d\n", value);
}

// Function to dequeue an element
int dequeue() {
    if (isEmpty()) {
        printf("Queue Underflow\n");
        return -1;
    }
    struct Node* temp = front;
    int value = temp->data;
    front = front->next;
    if (front == NULL) rear = NULL; // Reset rear if
   queue becomes empty
    free(temp);
    return value;
}

// Function to display the queue
void display() {
    if (isEmpty()) {
        printf("Queue is empty\n");
        return;
    }
    struct Node* temp = front;
    printf("Queue elements: ");
    while (temp != NULL) {
        printf("%d ", temp->data);
        temp = temp->next;
    }
    printf("\n");
}

// Main function to test the Queue
int main() {
    enqueue(10);
    enqueue(20);
    enqueue(30);
    display();
    printf("Dequeued: %d\n", dequeue());
    display();
    return 0;
}
```

Listing 6: Queue Implementation in C using linked list

5 Expression Evaluation

5.1 Introduction

Expression evaluation involves the conversion and computation of expressions in infix, postfix, and prefix notations using stacks.

5.2 Mathematical Interlude

Stack-based algorithms for expression evaluation are derived from operator precedence and associativity rules.

5.3 Implementation in C

5.3.1 Infix to Postfix Conversion

```c
#include <stdio.h>
#include <stdlib.h>
#include <ctype.h>
#include <string.h>

// Stack structure
#define MAX 100
char stack[MAX];
int top = -1;

// Stack operations
void push(char ch) {
    stack[++top] = ch;
}

char pop() {
    if (top == -1) return -1;
    return stack[top--];
}

char peek() {
    if (top == -1) return -1;
    return stack[top];
}

// Function to check precedence of operators
int precedence(char op) {
    if (op == '+' || op == '-') return 1;
    if (op == '*' || op == '/') return 2;
    if (op == '^') return 3;
    return 0;
```

```c
}

// Function to check if character is an operator
int isOperator(char ch) {
    return ch == '+' || ch == '-' || ch == '*' || ch == '/'
    || ch == '^';
}

// Function to convert infix to postfix
void infixToPostfix(char* infix, char* postfix) {
    int j = 0;
    for (int i = 0; infix[i] != '\0'; i++) {
        char ch = infix[i];

        if (isalnum(ch)) {  // If operand, add to postfix
            postfix[j++] = ch;
        } else if (ch == '(') {  // Push '(' onto stack
            push(ch);
        } else if (ch == ')') {  // Pop until '('
            while (peek() != '(') {
                postfix[j++] = pop();
            }
            pop();  // Remove '('
        } else if (isOperator(ch)) {  // If operator
            while (precedence(peek()) >= precedence(ch)) {
                postfix[j++] = pop();
            }
            push(ch);
        }
    }

    // Pop remaining operators
    while (top != -1) {
        postfix[j++] = pop();
    }
    postfix[j] = '\0';
}

// Main function
int main() {
    char infix[MAX], postfix[MAX];
    printf("Enter an infix expression: ");
    scanf("%s", infix);

    infixToPostfix(infix, postfix);
    printf("Postfix expression: %s\n", postfix);
    return 0;
}
```

Listing 7: Infix to Postfix Conversion in C

5.3.2 Prefix to Infix Conversion

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <ctype.h>

// Define the stack structure
typedef struct Stack {
    char* arr[100];
    int top;
} Stack;

// Function to initialize the stack
void initializeStack(Stack* stack) {
    stack->top = -1;
}

// Function to check if the stack is empty
int isEmpty(Stack* stack) {
    return stack->top == -1;
}

// Function to push an element onto the stack
void push(Stack* stack, char* str) {
    stack->arr[++(stack->top)] = str;
}

// Function to pop an element from the stack
char* pop(Stack* stack) {
    return stack->arr[(stack->top)--];
}

// Function to check if a character is an operator
int isOperator(char ch) {
    return ch == '+' || ch == '-' || ch == '*' || ch == '/';
}

// Function to convert a prefix expression to an infix
   expression
void prefixToInfix(char* prefix) {
    Stack stack;
    initializeStack(&stack);
    int length = strlen(prefix);

    // Traverse the prefix expression in reverse order
    for (int i = length - 1; i >= 0; i--) {
        if (isspace(prefix[i])) {
            continue;  // Skip whitespace
        }
        if (isOperator(prefix[i])) {
            // Pop two operands from the stack
            char* operand1 = pop(&stack);
            char* operand2 = pop(&stack);

            // Allocate memory for the resulting infix expression
            char* infix = (char*)malloc(strlen(operand1) + strlen(operand2) + 4);

            // Combine the operands and operator into a new infix expression
            sprintf(infix, "(%s %c %s)", operand1, prefix[i], operand2);

            // Push the resulting expression onto the stack
            push(&stack, infix);
        } else {
            // If it's an operand, push it onto the stack as a string
            char* operand = (char*)malloc(2);
            operand[0] = prefix[i];
            operand[1] = '\0';
            push(&stack, operand);
        }
    }

    // The final element on the stack is the resulting infix expression
    printf("Infix Expression: %s\n", pop(&stack));
}

// Main function to demonstrate the conversion
int main() {
    char prefix[] = "*+AB-CD";
    printf("Prefix Expression: %s\n", prefix);
    prefixToInfix(prefix);
    return 0;
}
```

Listing 8: Prefix to Infix Conversion in C

5.3.3 Postfix Evaluation

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

```c

// Stack for integer operands
int stack[MAX];
int top = -1;

void pushInt(int value) {
    stack[++top] = value;
}

int popInt() {
    return stack[top--];
}

// Function to evaluate postfix expression
int evaluatePostfix(char* postfix) {
    for (int i = 0; postfix[i] != '\0'; i++) {
        char ch = postfix[i];
        if (isdigit(ch)) {  // If operand, push onto stack
            pushInt(ch - '0');
        } else {  // Operator
            int b = popInt();
            int a = popInt();
            switch (ch) {
                case '+': pushInt(a + b); break;
                case '-': pushInt(a - b); break;
                case '*': pushInt(a * b); break;
                case '/': pushInt(a / b); break;
            }
        }
    }
    return popInt();
}

// Main function
int main() {
    char postfix[MAX];
    printf("Enter a postfix expression: ");
    scanf("%s", postfix);

    int result = evaluatePostfix(postfix);
    printf("Result of evaluation: %d\n", result);
    return 0;
}
```

Listing 9: Postfix Evaluation in C

6 Conclusion

In this document, we explored various fundamental data structures, including linked lists, stacks, queues, and their applications in expression evaluation (prefix, infix, and postfix conversions). These implementations serve as the backbone of many computational processes and are essential for solving complex problems efficiently.

Through detailed discussions, mathematical foundations, and structured C code implementations, this document provides a comprehensive study guide for understanding and applying these concepts. Each section has been carefully designed to not only introduce theoretical aspects but also emphasize hands-on coding practices.

The knowledge gained from these studies is pivotal for excelling in data structures and algorithms, which are integral parts of computer science curricula and essential skills for technical interviews and real-world software development.

References

The following books and chapters were used as references to understand and implement the concepts discussed:

- **Introduction to Algorithms**, Cormen et al.
  - Chapter 10: Elementary Data Structures
  - Chapter 22: Graph Algorithms (for advanced topics)
- **Data Structures Using C**, Reema Thareja
  - Chapter 3: Linked Lists
  - Chapter 4: Stacks and Queues
- **Fundamentals of Data Structures in C**, Ellis Horowitz, Sartaj Sahni, and Susan Anderson-Freed
  - Chapter 2: Arrays, Stacks, and Queues
  - Chapter 4: Linked Lists

These references, coupled with practical implementation in C, create a strong foundation for mastering data structures and algorithms.