DSA Lab Manual: Experiments 1-12

Data Structures Laboratory

Contents

Experiment 1: Weekly Planner (Dynamic Memory Allocation)

The topic is Dynamic Memory Allocation (DMA). Imagine you are buying a diary. If you buy a fixed 100-page diary but only write for 2 days, you wasted 98 pages, right? That is **Static Memory**. But, if you can add pages *as you need them*, that is **Dynamic Memory**.

In this program, we are creating a Weekly Planner:

- **Struct Day:** We create a blueprint that holds the date, day name, and activity.
- **Malloc:** This is the hero function! `week = (struct Day *) malloc(7 * sizeof(struct Day));`. We are telling the computer: "Hey, at runtime, please reserve space exactly for 7 days."
- **Pointer:** The variable `week` is a pointer pointing to the first day of this allocated memory.
- **Free:** Finally, like a good programmer, we use `free(week)` to give the memory back to the system so we don't cause leaks!

0.1 Code

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

struct Day {
    int date;
    char dayname;
    char activity;
};

int main() {
    struct Day *week;
    int i;
    // Allocate memory for 7 days
    week = (struct Day *) malloc(7 * sizeof(struct Day));
    if (week == NULL) {
        printf("Memory allocation failed \n");
        return 1;
    }
    printf("\n -- Enter details for each of the 7 days -- \n");
    for (i = 0; i < 7; i++) {
        printf("\n Day %d : \n", i + 1);
        printf("Enter date : ");
        scanf("%d", &week[i].date);
        getchar(); // Clear new line left in input Buffer
        printf("Enter day name (e.g. Monday): ");
        scanf("%[^\n]", week[i].dayname); // Reads until newline
        printf("Enter activity of the day: ");
        scanf(" %[^\n]", week[i].activity);
    }
    printf("\n ----- Week Planner ----- \n");
    for (i = 0; i < 7; i++) {
        printf("\n Day %d \n", i + 1);
        printf("Date : %d \n", week[i].date);
        printf("Day Name : %s \n", week[i].dayname);
        printf("Activity : %s \n", week[i].activity);
    }
    free(week);
    return 0;
}
```

Experiment 2: String Pattern Matching

Let's understand Pattern Matching. Have you used "Find and Replace" in MS Word? That is exactly what we are coding here!

- **The Inputs:** We have a Main String (STR), a Pattern to find (PAT), and a Replacement (REP).
- **The Logic (Brute Force):** We iterate through the main string. At every position, we check: "Does the pattern match here?"
- **The Trick:** We use a function `patmatch()`. If it returns 1 (True), we insert the REP string into our result. If it returns 0 (False), we just copy the original character from STR.
- **Result:** We build a new string `res` character by character. Simple, right?

0.2 Code

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

int sl(char str[]) {
    int len = 0;
    while (str[len] != '\0')
        len++;
    return len;
}

int patmatch(char str[], char pat[], int pos) {
    int i = 0;
    while (pat[i] != '\0') {
        if (str[pos + i] != pat[i])
            return 0; // no match found
        i++;
    }
    return 1; // match found
}

void reppat(char str[], char pat[], char rep[], char res[]) {
    int i = 0, j = 0, k;
    int mlen = sl(str);
    int plen = sl(pat);
    int rlen = sl(rep);
    while (i < mlen) {
        if (patmatch(str, pat, i)) {
            for (k = 0; k < rlen; k++) {
                res[j++] = rep[k];
            }
            i = i + plen;
        } else {
            res[j++] = str[i++];
        }
    }
    res[j] = '\0';
}

int main() {
    char str, pat, rep, res;
    printf("Enter main string: ");
    scanf("%[^\n]", str);
```

```
43      printf("Enter the pattern to replace: ");
44      scanf(" %[^\n]", pat);
45      printf("Enter the replacement string: ");
46      scanf(" %[^\n]", rep);
47      reppat(str, pat, rep, res);
48      printf("\nResultant string: %s \n", res);
49      return 0;
50  }
```

Experiment 3: Stack Operations

Welcome to Stacks! The concept here is LIFO—Last In, First Out. Think of a stack of plates at a wedding buffet. You put the last plate on top, and that's the first one you pick up.

- **TOP Variable:** This is the boss. It tracks the index of the top element.
- **Push:** Increment top (`++top`), then insert.
- **Pop:** Take the value, then decrement top (`top--`).
- **Palindrome Check:** We push a string onto the stack. When we pop it, it comes out in reverse order! If `Original == Reverse`, it is a palindrome!

0.3 Code

```
#include <stdio.h>
#include <string.h>
#define max 50

int stack[max];
int top = -1;

void push(int x) {
    if (top == max - 1)
        printf("stack overflow \n");
    else {
        top++;
        stack[top] = x;
        printf("%d pushed to stack \n", x);
    }
}

void pop() {
    if (top == -1)
        printf("stack underflow \n");
    else {
        int x = stack[top];
        top--;
        printf("%d popped from stack \n", x);
    }
}

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

void pal() {
    char str, rev;
    int i, j = 0;
    printf("Enter a string : ");
    scanf("%s", str);
```

```
44      top = -1;
45      for (i = 0; str[i] != '\0'; i++) {
46          if (top == max - 1) {
47              printf("stack overflow \n");
48              return;
49          }
50          stack[++top] = str[i];
51      }
52      for (i = top; i >= 0; i--)
53          rev[j++] = stack[i];
54      rev[j] = '\0';
55      if (strcmp(str, rev) == 0)
56          printf("Palindrome \n");
57      else
58          printf("Not Palindrome \n");
59  }
60
61  void status() {
62      if (top == -1)
63          printf("stack empty \n");
64      else if (top == max - 1)
65          printf("stack full \n");
66      else
67          printf("Stack has %d elements \n", top + 1);
68  }
69
70  int main() {
71      int ch, val;
72      while (1) {
73          printf("\n 1. Push \n 2. Pop \n 3. Palindrome \n 4. Overflow \n
                5. Status \n 6. Exit \n Enter a choice : ");
74          scanf("%d", &ch);
75          switch (ch) {
76              case 1: printf("Enter value : "); scanf("%d", &val); push(
                    val); break;
77              case 2: pop(); break;
78              case 3: pal(); break;
79              case 4:
80                  if (top == -1) printf("Underflow \n");
81                  else if (top == max - 1) printf("Overflow \n");
82                  else printf("No under or over flow \n");
83                  break;
84              case 5: status(); display(); break;
85              case 6: return 0;
86              default: printf("Invalid choice \n");
87          }
88      }
89  }
```

Experiment 4: Infix to Postfix Conversion

Why do we need Postfix? Because computers hate brackets! Humans love `(A+B)`, but it's hard for computers. They prefer `AB+`.

- **The Stack's Role:** The stack acts as a "waiting room" for operators (+, -, *). Operands (A, B, 1, 2) go straight to the output.
- **Precedence:** If a "strong" operator comes (like *), and a "weak" one is in the stack (+), the weak one stays. But if a weak one comes and a strong one is sitting there, the strong one must perform its job first (POP).
- **Brackets:** Left bracket `(` is pushed. It waits until `)` arrives. When `)` comes, we pop everything until we find the matching `(`. This logic handles the BODMAS rules for us!

0.4 Code

```
#include <stdio.h>
#include <string.h>
#include <ctype.h>
#define max 50

int stack[max];
int top = -1;

void push(char ele) {
    stack[++top] = ele;
}

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

int precedence(char ele) {
    switch (ele) {
        case '#': return 0;
        case '(': return 1;
        case '+': case '-': return 2;
        case '*': case '/': case '%': return 3;
        case '$': case '^': return 4;
    }
    return 0;
}

int main() {
    char infix, postfix, ch;
    int i = 0, k = 0;
    printf("\n Enter the infix Expression : ");
    scanf("%s", infix);
    push('#');
    while ((ch = infix[i++]) != '\0') {
        if (ch == '(')
            push(ch);
        else if (isalnum(ch))
            postfix[k++] = ch;
        else if (ch == ')') {
            while (stack[top] != '(')
                postfix[k++] = pop();
            pop();
```

```
43          } else {
44              while (precedence(stack[top]) >= precedence(ch))
45                  postfix[k++] = pop();
46              push(ch);
47          }
48      }
49      while (stack[top] != '#')
50          postfix[k++] = pop();
51      postfix[k] = '\0';
52      printf("\n Given infix expression is : %s \n", infix);
53      printf("\n Postfix expression is : %s \n", postfix);
54      return 0;
55  }
```

Experiment 5: Postfix Eval & Tower of Hanoi

Part A: Postfix Evaluation Now that we have `AB+`, how do we calculate it? We scan left to right.

1. See a number? **PUSH** it.
2. See an operator? **POP** the top two numbers. Apply the math (`num1 + num2`). **PUSH** the result back.
3. At the end, the only thing left in the stack is your answer!

Part B: Tower of Hanoi This is pure Recursion magic! To move N disks from Source (A) to Destination (C):

1. Move `N-1` disks from A to B (Temp).
2. Move the biggest disk directly from A to C.
3. Move the `N-1` disks from B to C.

It is a function calling itself until `N=1` (Base condition).

0.5 Code

```c
// PART A: Postfix Evaluation
#include <stdio.h>
#include <ctype.h>
#include <math.h>
#include <string.h>

int func(char sym, int num1, int num2) {
    switch (sym) {
        case '+': return num1 + num2;
        case '-': return num1 - num2;
        case '*': return num1 * num2;
        case '/': return num1 / num2;
        case '%': return num1 % num2;
        case '^': return pow(num1, num2);
    }
    return 0;
}

int main() {
    int stack, top = -1, num1, num2, res;
    char suf, sym;
    printf("Enter the exp: ");
    scanf("%s", suf);
    int m = strlen(suf);
    for (int i = 0; i < m; i++) {
        sym = suf[i];
        if (isdigit(sym)) {
            stack[++top] = sym - '0';
        } else {
            num2 = stack[top--];
            num1 = stack[top--];
            res = func(sym, num1, num2);
            stack[++top] = res;
        }
    }
    printf("The answer is : %d \n", stack[top]);
}
```

```
38
39 // PART B: Tower of Hanoi
40 #include <stdio.h>
41 void tower(int n, char source, char temp, char dest) {
42     if (n == 1) {
43         printf("\n Move disk 1 from peg %c to peg %c", source, dest);
44         return;
45     }
46     tower(n - 1, source, dest, temp);
47     printf("\n Move the disk %d from peg %c to peg %c", n, source, dest
       );
48     tower(n - 1, temp, source, dest);
49 }
50 // Note: Normally main() would call tower()
```

Experiment 6: Queues (Simple & Circular)

Part A: Simple Queue FIFO—First In, First Out. Just like a line at the cinema.

- **Insertion (Enqueue):** Happens at the Rear.
- **Deletion (Dequeue):** Happens at the Front.
- **Problem:** Once Front moves forward, the space behind it is wasted!

Part B: Circular Queue Solution to the wastage! Imagine the queue is a circle or a round table.

- **The Logic:** Instead of just `rear++`, we use `rear = (rear + 1) % MAX`.
- If we reach index 4 and the max is 5, `(4+1)%5` becomes 0. We wrap around to the start! Now we never waste memory.

0.6 Code

```
// CIRCULAR QUEUE IMPLEMENTATION
#include <stdio.h>
#include <stdlib.h>
#define max 4

char q[max];
int front = 0, rear = -1, count = 0;

void insert(char item) {
    if (count == max) {
        printf("\n Queue Overflow !\n");
        return;
    }
    rear = (rear + 1) % max;
    q[rear] = item;
    count++;
    printf("Inserted '%c' \n", item);
}

void del() {
    if (count == 0) {
        printf("\n Queue Underflow !\n");
        return;
    }
    printf("Deleted '%c' \n", q[front]);
    front = (front + 1) % max;
    count--;
}

void display() {
    int i, idx;
    if (count == 0) {
        printf("\n Queue is empty. \n");
        return;
    }
    printf("\n Circular Queue contents : ");
    idx = front;
    for (i = 0; i < count; i++) {
        printf("%c ", q[idx]);
        idx = (idx + 1) % max;
    }
```

```
42      printf("\n");
43  }
44
45  int main() {
46      int choice;
47      char item;
48      while (1) {
49          printf("\n 1. Insert \n 2. Delete \n 3. Display \n 4. Exit \n")
                ;
50          scanf("%d", &choice);
51          switch (choice) {
52              case 1: printf("Enter char: "); scanf(" %c", &item); insert
                    (item); break;
53              case 2: del(); break;
54              case 3: display(); break;
55              case 4: exit(0);
56          }
57      }
58      return 0;
59  }
```

Experiment 7: Singly Linked List (SLL)

Well Arrays are contiguous (neighbors). Linked Lists are scattered.

- **The Node:** It has two parts: Data and a Pointer (`next`) to the next node.
- **Front Insertion (Stack Demo):** It's $O(1)$. New node comes, points to Head, and becomes the new Head. Very fast!
- **Rear Insertion:** You have to traverse the *whole* list to find the last guy, then attach the new node.
- **The Code:** We use `struct student` and dynamic memory (`malloc`) for every single student record. No memory wastage here!

0.7 Code

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

struct student {
    char usn; char name; char prog;
    int sem; long phno;
    struct student *next;
};
struct student *head = NULL;

void insert_f() {
    struct student *temp = (struct student *) malloc(sizeof(struct
        student));
    printf("Enter USN, Name, Prog, Sem, PhNo: ");
    scanf("%s %s %s %d %ld", temp->usn, temp->name, temp->prog, &temp->
        sem, &temp->phno);
    temp->next = head;
    head = temp;
}

void insert_r() {
    struct student *temp = (struct student *) malloc(sizeof(struct
        student));
    printf("Enter USN, Name, Prog, Sem, PhNo: ");
    scanf("%s %s %s %d %ld", temp->usn, temp->name, temp->prog, &temp->
        sem, &temp->phno);
    temp->next = NULL;
    if (head == NULL) { head = temp; return; }
    struct student *cur = head;
    while (cur->next != NULL) cur = cur->next;
    cur->next = temp;
}

void delete_f() {
    if (head == NULL) { printf("Empty\n"); return; }
    struct student *temp = head;
    head = head->next;
    free(temp);
}

void delete_r() {
    if (head == NULL) { printf("Empty\n"); return; }
```

```
39      if (head->next == NULL) { free(head); head = NULL; return; }
40      struct student *prev = NULL, *cur = head;
41      while (cur->next != NULL) { prev = cur; cur = cur->next; }
42      prev->next = NULL;
43      free(cur);
44  }
45
46  void display() {
47      struct student *cur = head;
48      int count = 0;
49      while (cur != NULL) {
50          printf("%s %s\n", cur->usn, cur->name);
51          count++; cur = cur->next;
52      }
53      printf("Count: %d\n", count);
54  }
55
56  int main() {
57      int choice, n, i;
58      while (1) {
59          printf("\n 1.Create 2.Display 3.Ins End 4.Del End 5.Ins Front
                6.Del Front 7.Exit\n");
60          scanf("%d", &choice);
61          switch(choice) {
62              case 1: printf("N: "); scanf("%d", &n); for(i=0;i<n;i++)
                    insert_f(); break;
63              case 2: display(); break;
64              case 3: insert_r(); break;
65              case 4: delete_r(); break;
66              case 5: insert_f(); break;
67              case 6: delete_f(); break;
68              case 7: exit(0);
69          }
70      }
71      return 0;
72  }
```

Experiment 8: Doubly Linked List (DLL)

Double Linked List = Double Power! In SLL, we could only go forward. If we missed something, we couldn't look back.

- **The Structure:** `struct node` now has `prev` (Previous), Data, and `next`.
- **Insertion/Deletion:** We must be careful! When we add a node, we have to update the `next` of the previous guy AND the `prev` of the next guy. If you forget one link, the chain breaks.
- **Queue/Dequeue Demo:** By inserting at Rear and deleting from Front (or vice versa), this list behaves exactly like a Double Ended Queue (Dequeue).

0.8 Code

```
#include <stdio.h>
#include <stdlib.h>
struct node {
    struct node *prev; int ssn; long int phno; float sal;
    char name, dept, desig;
    struct node *next;
};
struct node *head = NULL, *tail = NULL; int count = 0;

struct node* create() {
    struct node *temp = (struct node *) malloc(sizeof(struct node));
    temp->prev = NULL; temp->next = NULL;
    printf("Enter SSN, Name, Dept, Desig, Sal, Ph: ");
    scanf("%d %s %s %s %f %ld", &temp->ssn, temp->name, temp->dept,
       temp->desig, &temp->sal, &temp->phno);
    count++; return temp;
}

void insert_r() {
    struct node *newnode = create();
    if (head == NULL) { head = tail = newnode; return; }
    tail->next = newnode; newnode->prev = tail; tail = newnode;
}

void delete_f() {
    if (head == NULL) return;
    struct node *ptr = head;
    if (head == tail) { head = tail = NULL; }
    else { head = head->next; head->prev = NULL; }
    free(ptr); count--;
}

void display() {
    struct node *temp = head;
    while(temp != NULL) {
        printf("%d %s\n", temp->ssn, temp->name); temp = temp->next;
    }
    printf("Count: %d\n", count);
}
// Main function logic similar to SLL, calling these functions
```

Experiment 9: Polynomial Addition

Polynomials look like $3x^2 + 4x^1$. Adding them is just algebra!

- **The Logic:** We use two pointers, P (for poly 1) and Q (for poly 2).
- **Case 1 (Powers Equal):** If P has x^2 and Q has x^2, we add their coefficients ($3 + 5 = 8x^2$) and move both pointers.
- **Case 2 (P > Q):** If P has x^3 and Q has x^2, P is bigger. We copy P to the answer and move P.
- **Case 3 (Q > P):** Copy Q and move Q.
- **Leftovers:** If one list finishes, we just copy whatever is remaining in the other list.

0.9 Code

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

struct node { int coeff; int pow; struct node * next; };

void insert_f(struct node ** head, int c, int p) {
    struct node * temp = (struct node *) malloc(sizeof(struct node));
    temp->coeff = c; temp->pow = p; temp->next = *head; *head = temp;
}

struct node * addpoly(struct node * A, struct node * B) {
    struct node * res = NULL, *p = A, *q = B;
    while (p != NULL && q != NULL) {
        if (p->pow == q->pow) {
            insert_f(&res, p->coeff + q->coeff, p->pow);
            p = p->next; q = q->next;
        } else if (p->pow > q->pow) {
            insert_f(&res, p->coeff, p->pow); p = p->next;
        } else {
            insert_f(&res, q->coeff, q->pow); q = q->next;
        }
    }
    return res;
}
// Print and Main functions follow standard list traversal
```

Experiment 10: Binary Search Tree (BST)

BST is a disciplined tree.

- **Rule:** Everything SMALLER than the root goes to the LEFT. Everything LARGER goes to the RIGHT.
- **Recursion:** We use recursion for everything—Insertion, Traversal, Search.
- **Traversals:**
  - **Inorder (L-N-R):** Visits nodes in sorted order! (Ascending).
  - **Preorder (N-L-R):** Visit Node first.
  - **Postorder (L-R-N):** Visit Children first, then the Node.

0.10 Code

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

struct treeNode { int data; struct treeNode *left; struct treeNode *
    right; };

struct treeNode* insert(struct treeNode *root, int data) {
    if (root == NULL) {
        struct treeNode *temp = (struct treeNode *) malloc(sizeof(
            struct treeNode));
        temp->data = data; temp->left = temp->right = NULL;
        return temp;
    }
    if (data < root->data) root->left = insert(root->left, data);
    else if (data > root->data) root->right = insert(root->right, data)
        ;
    return root;
}

void inorder(struct treeNode *root) {
    if (root == NULL) return;
    inorder(root->left);
    printf("%d\t", root->data);
    inorder(root->right);
}
// Preorder and Postorder follow similar recursive structure
```

Experiment 11: Graph Operations (BFS & DFS)

Now we are diving into Graphs. Imagine a map of cities connected by highways. We need to figure out how to travel from one city to all others. We have two main strategies for this:

- **The Map (Adjacency Matrix):** We use a 2D array (Grid). If `matrix[i][j] == 1`, there is a direct road between City i and City j. If 0, no road.
- **DFS (Depth First Search):** Think of this as the "Stubborn Explorer". This algorithm picks a path and keeps walking forward until it hits a dead end. Only then does it backtrack. It uses a Stack (or Recursion) to remember where it came from.
- **BFS (Breadth First Search):** Think of this as the "Water Ripple". It visits all immediate neighbors first, then the neighbors' neighbors. It spreads out layer by layer. It uses a Queue because the first neighbor you see is the first one you explore.

0.11 Code

```
#include <stdio.h>
int a, q, visited, n, i, j, f = 0, r = -1;

void bfs(int v) {
    for (i = 1; i <= n; i++)
        if (a[v][i] && !visited[i]) q[++r] = i;
    if (f <= r) {
        visited[q[f]] = 1; bfs(q[f++]);
    }
}
void dfs(int v) {
    visited[v] = 1;
    for (i = 1; i <= n; i++)
        if (a[v][i] && !visited[i]) {
            printf("%d ", i); dfs(i);
        }
}
int main() {
    int v;
    printf("Enter n: "); scanf("%d", &n);
    printf("Enter matrix:\n");
    for(i=1;i<=n;i++) for(j=1;j<=n;j++) scanf("%d", &a[i][j]);
    printf("Start vertex: "); scanf("%d", &v);
    dfs(v); // Call BFS similarly
    return 0;
}
```

Experiment 12: Hashing (Linear Probing)

Finally the last topic! Hashing. Why do we need Hashing? Because searching is slow! If you have to find a student in a list of 1,000, checking them one by one (Linear Search) takes $O(N)$ time. We want **Magic Time**—$O(1)$!

- **The Hash Function ($K\%m$):** This is the magic spell. You give it a big Key (like Employee ID 24), and it gives you a small Index (address) to store it in. For example, $24\%10 = 4$.
- **Collision:** What if Key 14 comes? $14\%10 = 4$. Oh no! Slot 4 is full. This is a collision.
- **Linear Probing (The Solution):** Don't worry. If slot 4 is occupied, just look at the **very next spot** (Index 5). If that is full, look at 6. Just search linearly for the next empty spot. Simple!

0.12 Code

```c
#include <stdio.h>
#include <stdlib.h>
#define MAX 100

int main() {
    int NUM, key[MAX], m, i, *ht;
    printf("Enter count and table size: ");
    scanf("%d %d", &NUM, &m);
    ht = (int *)calloc(m, sizeof(int)); // Init to 0

    printf("Enter keys: ");
    for (i = 0; i < NUM; i++) {
        scanf("%d", &key[i]);
        int idx = key[i] % m;
        while(ht[idx] != 0) idx = (idx + 1) % m; // Linear Probing
        ht[idx] = key[i];
    }

    printf("Hash Table:\n");
    for (i = 0; i < m; i++) printf("%d: %d\n", i, ht[i]);
    return 0;
}
```