DS lab internal

Data Structures Internal Lab-1 Answers

Viva Questions

1. What is a collision in hashing? What are the various methods for collision resolution in hashing? Explain them.

A collision in hashing occurs when two or more distinct keys are mapped to the same hash table index by the hash function. This happens because hash functions compress a large key space into a smaller array of slots, leading to potential overlaps.

Various methods for collision resolution include:

- **Chaining (Separate Chaining)**: Each hash table slot contains a linked list (or other data structure) of elements that hash to that index. When a collision occurs, the new element is appended to the list. Search, insert, and delete operations traverse the list. This method is simple and handles an arbitrary number of collisions but may degrade to O(n) time in the worst case if all keys hash to one slot.
- **Open Addressing (Probing)**: All elements are stored directly in the hash table array. When a collision occurs, the algorithm probes for the next available slot using a probe sequence. Common probing methods:
  - **Linear Probing**: Probe sequentially (e.g., h(k) + i mod m, where i = 0,1,2,…). Simple but prone to primary clustering (long runs of occupied slots).
  - **Quadratic Probing**: Probe using a quadratic function (e.g., h(k) + i^2 mod m). Reduces clustering but can lead to secondary clustering and may not probe all slots if the table size isn't prime.
  - **Double Hashing**: Use a second hash function for probing (e.g., h(k) + i * h2(k) mod m). Offers better distribution and probes more slots.
- **Rehashing**: When the load factor exceeds a threshold, resize the hash table (usually double the size) and reinsert all elements using a new hash function. This is often combined with other methods to maintain efficiency.

Other less common methods include cuckoo hashing (using multiple hash functions and displacing elements) and perfect hashing (for static sets with no collisions).

2. Write the algorithm for infix to postfix and convert the following infix expression into postfix expression: A + B - C * D * E / F - G.

Algorithm for Infix to Postfix Conversion (using a Stack):

1. Create an empty stack for operators and an empty list (or string) for the postfix output.
2. Scan the infix expression from left to right.
3. For each token:
   - If it's an operand (variable or number), append it to the postfix output.
   - If it's an opening parenthesis '(', push it onto the stack.
   - If it's a closing parenthesis ')', pop operators from the stack and append to postfix until an opening parenthesis is encountered (discard the '(').
   - If it's an operator:
     - While the stack is not empty and the top operator has equal or higher precedence than the current operator, pop from stack and append to postfix.
     - Push the current operator onto the stack.
1. After scanning, pop any remaining operators from the stack and append to postfix.
2. Return the postfix expression.

Precedence rules: Typically, ^ (exponent) > * / > + - (left-associative).

Conversion of A + B - C * D * E / F - G:

- Start with empty stack and postfix.
- A: Operand → Postfix: A
- +: Operator, stack empty → Push +
- B: Operand → Postfix: A B
- -: Operator, + has lower precedence → Pop + → Postfix: A B +; Push -
- C: Operand → Postfix: A B + C
- *: Operator, - has lower precedence → Push *
- D: Operand → Postfix: A B + C D
- *: Operator, * has equal precedence → Pop * → Postfix: A B + C D *; Push *
- E: Operand → Postfix: A B + C D * E
- /: Operator, * has equal precedence (assuming left-associative) → Pop * → Postfix: A B + C D * E *; Push /
- F: Operand → Postfix: A B + C D * E * F
- -: Operator, / has higher precedence → Pop / → Postfix: A B + C D * E * F /; - has equal precedence to previous - → Pop - → Postfix: A B + C D * E * F / -; Push -
- G: Operand → Postfix: A B + C D * E * F / - G
- End: Pop - → Postfix: A B + C D * E * F / - G -

Final Postfix: AB+CD*E*F/-G-
(Note: The expression is ambiguous without parentheses, but assuming standard precedence: A + B - ((C * D * E) / F) - G, the postfix is AB+CD*E*F/-G-)

3. What are skip lists? Write algorithms for insertion, deletion and search operations on skip lists.

Skip lists are probabilistic data structures that allow fast search, insertion, and deletion in sorted linked lists by adding multiple layers of express lanes (pointers) to skip over elements.
A skip list consists of multiple levels of linked lists, where the bottom level is a standard sorted linked list, and higher levels are sparser subsets acting as indexes. Each node has a random height (number of levels it participates in), typically chosen with probability p=0.5.

Search Algorithm:

1. Start at the highest level of the header node.
2. Traverse forward while the next node's key < target.
3. If next node's key >= target, drop down one level.
4. Repeat until level 0.
5. If key == target, return node; else, not found.

Time: O(log n) expected.

Insertion Algorithm:

1. Perform search to find the position where the new key should be inserted, recording the update nodes (last node at each level before dropping).
2. Randomly choose the height h for the new node (e.g., geometric distribution with p=0.5).
3. Create a new node with h levels.
4. For each level i from 0 to h-1:
   - Set new node's forward[i] = update[i]->forward[i]
   - Set update[i]->forward[i] = new node
1. If h > current max level, update the header and increase max level.

Time: O(log n) expected.

Deletion Algorithm:

1. Perform search to find the node to delete, recording update nodes as in insertion.
2. If not found, return.
3. For each level i from 0 to node->height-1:
   - Set update[i]->forward[i] = node->forward[i]
1. Free the node.

2. Decrease max level if necessary (if header's forward[max] == NULL).
Time: O(log n) expected.

4. Write short notes on

i) Applications of queue and stack.

- **Queue (FIFO)**:
  - BFS in graphs/trees (level-order traversal).
  - Task scheduling (e.g., print queues, CPU scheduling like FCFS).
  - Buffers in OS/networks (e.g., message queues, keyboard input).
  - Simulation of real-world queues (e.g., bank lines).
- **Stack (LIFO)**:
  - Function call recursion (call stack for activation records).
  - Expression evaluation/parsing (e.g., infix to postfix, postfix evaluation).
  - Undo/redo in editors.
  - Backtracking algorithms (e.g., maze solving, DFS).

ii) Different Hash Functions

- **Division Method**: h(k) = k mod m (m prime, not power of 2/10). Simple, but poor if keys have patterns.
- **Multiplication Method**: h(k) = floor(m * (k * c mod 1)) where c is irrational (e.g., golden ratio). Good for arbitrary m.
- **Folding Method**: Split key into parts, add them, then mod m. Useful for strings/numbers.
- **Mid-Square Method**: Square key, take middle digits, mod m.
- **Universal Hashing**: Family of functions to minimize collisions probabilistically.
- For strings: Polynomial rolling hash (e.g., djb2, sdbm).

Good hash functions should be fast, uniform, and minimize collisions.

iii) Abstract data type and data structures

- **Abstract Data Type (ADT)**: A mathematical model defining data and operations without implementation details (e.g., Stack ADT with push, pop, top). Focuses on "what" (interface).
- **Data Structure**: Concrete implementation of an ADT (e.g., stack using array or linked list). Focuses on "how" (memory, algorithms).

ADTs provide abstraction for modularity; data structures affect efficiency (time/space).

Programs

All programs are in C. I've provided complete, compilable code with comments. Assume standard input/output.

1. a) Implement Stack using Arrays.

```
#include
#include
#define MAX 100

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

void push(int item) {
    if (top >= MAX - 1) {
        printf("Stack Overflow\n");
        return;
    }
    stack[++top] = item;
}
```

```
int pop() {
    if (top < 0) {
        printf("Stack Underflow\n");
        exit(1);
    }
    return stack[top--];
}

int peek() {
    if (top < 0) {
        printf("Stack Empty\n");
        exit(1);
    }
    return stack[top];
}

int isEmpty() {
    return top < 0;
}

int main() {
    push(10);
    push(20);
    printf("Top: %d\n", peek());
    printf("Pop: %d\n", pop());
    return 0;
}
```

1. b) Implement single linked list deletion operation.

```
#include
#include

struct Node {
    int data;
    struct Node* next;
};

struct Node* head = NULL;

void insert(int data) {  // Helper: Insert at end
    struct Node* newNode = (struct Node*)malloc(sizeof(struct Node));
```

```
    newNode->data = data;
    newNode->next = NULL;
    if (head == NULL) {
        head = newNode;
    } else {
        struct Node* temp = head;
        while (temp->next != NULL) temp = temp->next;
        temp->next = newNode;
    }
}

void deleteNode(int key) {
    struct Node* temp = head, *prev = NULL;
    if (temp != NULL && temp->data == key) {
        head = temp->next;
        free(temp);
        return;
    }
    while (temp != NULL && temp->data != key) {
        prev = temp;
        temp = temp->next;
    }
    if (temp == NULL) {
        printf("Key not found\n");
        return;
    }
    prev->next = temp->next;
    free(temp);
}

void printList() {
    struct Node* temp = head;
    while (temp != NULL) {
        printf("%d ", temp->data);
        temp = temp->next;
    }
    printf("\n");
}

int main() {
    insert(10);
    insert(20);
```

```
    insert(30);
    printList();  // 10 20 30
    deleteNode(20);
    printList();  // 10 30
    return 0;
}
```

2. Implement stack using Single Linked Lists.

```
#include
#include

struct Node {
    int data;
    struct Node* next;
};

struct Node* top = NULL;

void push(int data) {
    struct Node* newNode = (struct Node*)malloc(sizeof(struct Node));
    newNode->data = data;
    newNode->next = top;
    top = newNode;
}

int pop() {
    if (top == NULL) {
        printf("Stack Underflow\n");
        exit(1);
    }
    int data = top->data;
    struct Node* temp = top;
    top = top->next;
    free(temp);
    return data;
}

int peek() {
    if (top == NULL) {
        printf("Stack Empty\n");
        exit(1);
```

```
    }
    return top->data;
}

int main() {
    push(10);
    push(20);
    printf("Top: %d\n", peek());
    printf("Pop: %d\n", pop());
    return 0;
}
```

3. a) Implement Queue using Arrays.

```
#include
#include
#define MAX 100

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

void enqueue(int item) {
    if (rear >= MAX - 1) {
        printf("Queue Overflow\n");
        return;
    }
    if (front == -1) front = 0;
    queue[++rear] = item;
}

int dequeue() {
    if (front < 0 || front > rear) {
        printf("Queue Underflow\n");
        exit(1);
    }
    return queue[front++];
}

int isEmpty() {
    return front < 0 || front > rear;
}
```

```
int main() {
    enqueue(10);
    enqueue(20);
    printf("Dequeue: %d\n", dequeue());
    return 0;
}
```

3. b) Implement single linked list search operation.

```
#include
#include

struct Node {
    int data;
    struct Node* next;
};

struct Node* head = NULL;

void insert(int data) {  // Same as before
    struct Node* newNode = (struct Node*)malloc(sizeof(struct Node));
    newNode->data = data;
    newNode->next = NULL;
    if (head == NULL) {
        head = newNode;
    } else {
        struct Node* temp = head;
        while (temp->next != NULL) temp = temp->next;
        temp->next = newNode;
    }
}

int search(int key) {
    struct Node* temp = head;
    int pos = 0;
    while (temp != NULL) {
        if (temp->data == key) return pos;
        temp = temp->next;
        pos++;
    }
    return -1;  // Not found
}
```

```
int main() {
    insert(10);
    insert(20);
    insert(30);
    int pos = search(20);
    printf("Found at position: %d\n", pos);  // 1
    return 0;
}
```

4. Implement queue using single linked lists.

```
#include
#include

struct Node {
    int data;
    struct Node* next;
};

struct Node* front = NULL, *rear = NULL;

void enqueue(int data) {
    struct Node* newNode = (struct Node*)malloc(sizeof(struct Node));
    newNode->data = data;
    newNode->next = NULL;
    if (rear == NULL) {
        front = rear = newNode;
    } else {
        rear->next = newNode;
        rear = newNode;
    }
}

int dequeue() {
    if (front == NULL) {
        printf("Queue Underflow\n");
        exit(1);
    }
    int data = front->data;
    struct Node* temp = front;
    front = front->next;
```

```
    if (front == NULL) rear = NULL;
    free(temp);
    return data;
}

int main() {
    enqueue(10);
    enqueue(20);
    printf("Dequeue: %d\n", dequeue());
    return 0;
}
```

5. Write a C program that converts infix expression to postfix expression.

```
#include
#include
#include
#include

#define MAX 100

char stack[MAX];
int top = -1;

void push(char item) {
    if (top >= MAX - 1) exit(1);
    stack[++top] = item;
}

char pop() {
    if (top < 0) exit(1);
    return stack[top--];
}

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

```
void infixToPostfix(char* infix, char* postfix) {
    int i = 0, j = 0;
    while (infix[i] != '\0') {
        if (isalnum(infix[i])) {
            postfix[j++] = infix[i++];
        } else if (infix[i] == '(') {
            push(infix[i++]);
        } else if (infix[i] == ')') {
            while (stack[top] != '(') postfix[j++] = pop();
            pop();  // Discard '('
            i++;
        } else {
            while (top >= 0 && precedence(stack[top]) >=
precedence(infix[i])) {
                postfix[j++] = pop();
            }
            push(infix[i++]);
        }
    }
    while (top >= 0) postfix[j++] = pop();
    postfix[j] = '\0';
}

int main() {
    char infix[] = "A+B-C*D*E/F-G";
    char postfix[MAX];
    infixToPostfix(infix, postfix);
    printf("Postfix: %s\n", postfix);  // AB+CD*E*F/-G-
    return 0;
}
```

6. a) Write a C program to evaluate postfix expressions.

```
#include
#include
#include
#include

#define MAX 100

int stack[MAX];
int top = -1;
```

```
void push(int item) {
    if (top >= MAX - 1) exit(1);
    stack[++top] = item;
}

int pop() {
    if (top < 0) exit(1);
    return stack[top--];
}

int evaluatePostfix(char* postfix) {
    int i = 0;
    while (postfix[i] != '\0') {
        if (isdigit(postfix[i])) {
            push(postfix[i] - '0');  // Assuming single-digit operands
        } else {
            int op2 = pop();
            int op1 = pop();
            switch (postfix[i]) {
                case '+': push(op1 + op2); break;
                case '-': push(op1 - op2); break;
                case '*': push(op1 * op2); break;
                case '/': push(op1 / op2); break;
                // Add more operators if needed
            }
        }
        i++;
    }
    return pop();
}

int main() {
    char postfix[] = "53+82-*";  // Example: (5+3)*(8-2) = 48
    int result = evaluatePostfix(postfix);
    printf("Result: %d\n", result);
    return 0;
}
```

(Note: This assumes single-digit operands and basic operators. For variables/multi-digit, extend with tokenization.)

6. b) Implement single linked list deletion operation.

(Same as 1.b above.)

7. Implement Single linked list insertion and deletion operations.

```
#include
#include

struct Node {
    int data;
    struct Node* next;
};

struct Node* head = NULL;

void insert(int data) {  // Insert at end
    struct Node* newNode = (struct Node*)malloc(sizeof(struct Node));
    newNode->data = data;
    newNode->next = NULL;
    if (head == NULL) {
        head = newNode;
    } else {
        struct Node* temp = head;
        while (temp->next != NULL) temp = temp->next;
        temp->next = newNode;
    }
}

void deleteNode(int key) {  // Same as before
    struct Node* temp = head, *prev = NULL;
    if (temp != NULL && temp->data == key) {
        head = temp->next;
        free(temp);
        return;
    }
    while (temp != NULL && temp->data != key) {
        prev = temp;
        temp = temp->next;
    }
    if (temp == NULL) return;
    prev->next = temp->next;
    free(temp);
}
```

```
void printList() {
    struct Node* temp = head;
    while (temp != NULL) {
        printf("%d ", temp->data);
        temp = temp->next;
    }
    printf("\n");
}

int main() {
    insert(10);
    insert(20);
    printList();  // 10 20
    deleteNode(10);
    printList();  // 20
    return 0;
}
```

Data Structures Internal Lab-1: Algorithms for Programs

Below, I provide the algorithms for each of the programs listed in the lab, in the order they appear. Each algorithm is described in step-by-step pseudocode or structured English, focusing on the key operations implemented in the corresponding C code. These are high-level descriptions without the implementation details (e.g., no specific C syntax like malloc or printf), but they capture the logic.

1. a) Algorithm for Implementing Stack using Arrays

A stack is a LIFO structure with operations: push, pop, peek, isEmpty.

- **Push(item)**:
  1. If the stack is full (top >= max size - 1), report overflow.
  2. Increment top.
  3. Set stack[top] = item.
- **Pop()**:
  1. If the stack is empty (top < 0), report underflow.
  2. Retrieve stack[top].
  3. Decrement top.
  4. Return the retrieved item.
- **Peek()**:
  1. If the stack is empty, report error.
  2. Return stack[top].
- **isEmpty()**:
  1. Return true if top < 0, else false.

1. b) Algorithm for Implementing Single Linked List Deletion Operation

Assumes a singly linked list with nodes containing data and next pointer. Head points to the first node.

- **DeleteNode(key)**:
    1. Set temp = head, prev = null.
    2. If temp is not null and temp.data == key:
    - Set head = temp.next.
    - Free temp.
    - Return.
    1. While temp is not null and temp.data != key:
    - Set prev = temp.
    - Set temp = temp.next.
    1. If temp is null, report key not found.
    2. Else:
    - Set prev.next = temp.next.
    - Free temp.

2. Algorithm for Implementing Stack using Single Linked Lists

Stack operations using linked nodes (each with data and next). Top points to the top node.

- **Push(data)**:
    1. Create a new node with data.
    2. Set newNode.next = top.
    3. Set top = newNode.
- **Pop()**:
    1. If top is null, report underflow.
    2. Set data = top.data.
    3. Set temp = top.
    4. Set top = top.next.
    5. Free temp.
    6. Return data.
- **Peek()**:
    1. If top is null, report empty.
    2. Return top.data.

3. a) Algorithm for Implementing Queue using Arrays

A queue is a FIFO structure with operations: enqueue, dequeue, isEmpty. Uses front and rear indices.

- **Enqueue(item)**:
    1. If rear >= max size - 1, report overflow.
    2. If front == -1, set front = 0.
    3. Increment rear.
    4. Set queue[rear] = item.
- **Dequeue()**:
    1. If front < 0 or front > rear, report underflow.
    2. Retrieve queue[front].
    3. Increment front.
    4. Return the retrieved item.
- **isEmpty()**:
    1. Return true if front < 0 or front > rear, else false.

(Note: This is a simple linear queue; for circular, adjust with modulo.)

3. b) Algorithm for Implementing Single Linked List Search Operation

Assumes a singly linked list with head.

- **Search(key)**:
    1. Set temp = head.
    2. Set position = 0.
    3. While temp is not null:
    - If temp.data == key, return position.
    - Set temp = temp.next.
    - Increment position.
    1. Return -1 (not found).

4. Algorithm for Implementing Queue using Single Linked Lists

Queue operations using linked nodes. Front points to first, rear to last.

- **Enqueue(data)**:
    1. Create a new node with data and next = null.
    2. If rear is null:
    - Set front = rear = newNode.
    1. Else:
    - Set rear.next = newNode.
    - Set rear = newNode.
- **Dequeue()**:
    1. If front is null, report underflow.
    2. Set data = front.data.
    3. Set temp = front.
    4. Set front = front.next.
    5. If front is null, set rear = null.
    6. Free temp.
    7. Return data.

5. Algorithm for Converting Infix Expression to Postfix Expression

Uses a stack for operators. Assumes valid infix input.

- **InfixToPostfix(infix)**:
    1. Initialize empty stack and empty postfix output.
    2. For each token in infix (scanned left to right):
    - If token is alphanumeric (operand), append to postfix.
    - If token is '(', push to stack.
    - If token is ')':
        - While stack top != '(', pop and append to postfix.
        - Pop '(' (discard).
    - If token is operator:
        - While stack not empty and top has >= precedence than token, pop and append to postfix.
        - Push token.
    1. While stack not empty, pop and append to postfix.
    2. Return postfix.

(Precedence: ^ > * / > + -; left-associative.)

6. a) Algorithm for Evaluating Postfix Expressions

Uses a stack for operands. Assumes single-digit numbers and basic operators for simplicity.

- **EvaluatePostfix(postfix)**:
    1. Initialize empty stack.
    2. For each token in postfix (left to right):
    - If token is digit, push (token - '0') to stack.
    - If token is operator:
        - Pop op2 (right operand).
        - Pop op1 (left operand).
        - Compute result based on operator (+, -, *, /).
        - Push result.
    1. Return stack top (final result).

(Extend for multi-digit/variables by tokenizing properly.)

6. b) Algorithm for Implementing Single Linked List Deletion Operation

(This is a duplicate of 1.b; refer to the algorithm there.)

7. Algorithm for Implementing Single Linked List Insertion and Deletion Operations

Assumes insertion at end for simplicity.

- **Insert(data)**: (At end)
    1. Create new node with data and next = null.
    2. If head is null:
    - Set head = newNode.
    1. Else:
    - Set temp = head.
    - While temp.next != null, set temp = temp.next.
    - Set temp.next = newNode.
- **DeleteNode(key)**: (Same as 1.b)
    1. Set temp = head, prev = null.
    2. If temp not null and temp.data == key:
    - head = temp.next.
    - Free temp.
    - Return.
    1. While temp not null and temp.data != key:
    - prev = temp.
    - temp = temp.next.
    1. If temp null, return (not found).
    2. prev.next = temp.next.
    3. Free temp.