UNIT-II

1. Sorting Techniques

1.1 Definition of Sorting

Sorting is the process of arranging data in a specific order, typically ascending or descending, based on a particular criterion. It is a fundamental operation in computer science that improves the efficiency of searching and organizing data.

1.2 Types of Sorting Techniques

Sorting techniques are broadly categorized into **Internal Sorting** and **External Sorting**.

1.2.1 Internal Sorting

Internal sorting occurs when the entire dataset to be sorted fits into the computer's main memory.
Features:

- Fast and efficient for small datasets.
- Relies entirely on RAM.
- Example: Sorting an array in a program.

1.2.2 External Sorting

External sorting is used for large datasets that do not fit into main memory and require disk access.
Features:

- Suitable for massive datasets.
- Involves techniques like divide-and-conquer.
- Example: Merging sorted chunks from multiple files.

1.3 Common Sorting Algorithms

Here are some widely used sorting algorithms:

1.3.1 Bubble Sort

Definition: A simple sorting algorithm that repeatedly compares adjacent elements and swaps them if they are in the wrong order.

Steps:

1. Compare adjacent elements.
2. Swap if they are in the wrong order.
3. Repeat for all elements until the list is sorted.

Example: Input: `[5, 2, 9, 1]`
1st pass: `[2, 5, 1, 9]` → `[2, 1, 5, 9]`
2nd pass: `[1, 2, 5, 9]`

Advantages:

- Easy to implement.
  Disadvantages:
- Inefficient for large datasets (O(n^2) complexity).

1.3.2 Insertion Sort

Definition: Builds a sorted list by taking one element at a time and inserting it into its correct position.
Steps:

1. Assume the first element is sorted.
2. Take the next element and compare it backward in the sorted part.
3. Insert it in its correct position.

Example: Input: `[4, 3, 2, 1]`
Sorted: `[3, 4, 2, 1]` → `[2, 3, 4, 1]` → `[1, 2, 3, 4]`

Advantages:

- Efficient for small datasets.
  Disadvantages:
- Poor performance for large datasets (O(n^2)).

1.3.3 Selection Sort

Definition: A sorting algorithm that repeatedly selects the smallest element from the unsorted part and places it in the sorted part.
Steps:

1. Find the smallest element in the unsorted array.
2. Swap it with the first unsorted element.
3. Repeat for the remaining elements.

Example: Input: `[64, 25, 12, 22, 11]`
1st step: `[11, 25, 12, 22, 64]`
2nd step: `[11, 12, 25, 22, 64]`

Advantages:

- Simple to implement.
  Disadvantages:
- Inefficient for large datasets (O(n^2)).

1.3.4 Merge Sort

Definition: A divide-and-conquer algorithm that divides the array into halves, sorts them, and then merges them.
Steps:

1. Divide the array into two halves.
2. Recursively sort each half.
3. Merge the sorted halves.

Example:
Input: `[38, 27, 43, 3]`
Divide: `[38, 27]`, `[43, 3]`
Sort: `[27, 38]`, `[3, 43]`
Merge: `[3, 27, 38, 43]`

Advantages:

- O(n log n) complexity.
  Disadvantages:
- Requires additional memory for merging.

1.3.5 Quick Sort

Definition: A divide-and-conquer algorithm that selects a pivot element, partitions the array, and recursively sorts the partitions.

Steps:

1. Choose a pivot element.
2. Partition the array such that elements smaller than the pivot go to the left and larger elements to the right.
3. Recursively sort the partitions.

Example:
Input: `[10, 80, 30, 90, 40, 50, 70]`
Pivot: `70` → Partition: `[10, 30, 40, 50, 70, 90, 80]`

Advantages:

- Fast for large datasets (O(n log n)).
  Disadvantages:
- Worst-case complexity O(n^2) for bad pivot selection.

1.3.6 Radix Sort

Definition: A non-comparative sorting algorithm that sorts integers by processing individual digits.
Steps:

1. Sort elements based on the least significant digit (LSD).
2. Sort repeatedly for each significant digit.

Example:
Input: `[170, 45, 75, 90]`
Sort by 1s: `[90, 170, 45, 75]`
Sort by 10s: `[170, 45, 75, 90]`

Advantages:

- Linear time complexity for small digit ranges.
  Disadvantages:
- Limited to specific types of data.

---

2. Linked Lists

2.1 Linked List vs. Arrays

Linked Lists: A data structure where elements are stored as nodes, and each node points to the next.
Arrays: A collection of elements stored at contiguous memory locations.

Advantages of Linked Lists:

- Dynamic size.
- Efficient insertions and deletions.

Disadvantages of Linked Lists:

- Random access not possible.
- Higher memory usage due to pointers.

2.2 Representation in Memory

Each node in a linked list consists of:

- **Data:** Holds the element.
- **Pointer:** Points to the next node in the sequence.

2.3 Types of Linked Lists

2.3.1 Singly Linked List

A linked list where each node points to the next node.
Operations:

- **Insertion:** Add a new node at a specific position.
- **Deletion:** Remove a node.
- **Traversal:** Iterate through the list.

2.3.2 Doubly Linked List

Each node has pointers to both its previous and next nodes.
Advantages:

- Easier bidirectional traversal.
  Disadvantages:

- Requires more memory.

2.3.3 Circular Linked List

The last node points back to the first node, forming a loop.
Advantages:

- Continuous traversal.

2.3.4 Doubly Circular Linked List

Combines features of doubly linked and circular linked lists.
Features:

- Both directions are traversable in a loop.

2.4 Operations on Singly Linked Lists

2.4.1 Insertion

Insert a node at the beginning, end, or middle of the list.
Example Code (C-like):

```
Node* insert(Node* head, int data) {
    Node* newNode = (Node*)malloc(sizeof(Node));
    newNode->data = data;
    newNode->next = head;
    return newNode;
}
```

2.4.2 Deletion

Remove a node by adjusting the pointers.

2.4.3 Traversal

Iterate through the list to access each element.

2.4.4 Searching

Find a node with a specific value by iterating through the list.

2.4.2 Deletion in Singly Linked List

Deletion involves removing a node from the linked list by updating pointers. There are three main cases:

1. **Delete from the Beginning:** Update the head pointer to the next node.
2. **Delete from the End:** Traverse the list to find the second last node and set its pointer to `NULL`.
3. **Delete from the Middle:** Adjust the pointer of the previous node to skip the node to be deleted.

Example Code (C-like):

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

2.4.3 Traversal in Singly Linked List

Traversal involves visiting each node in the linked list sequentially from the head to the last node.

Steps:

1. Start at the head node.
2. Follow the `next` pointer to access subsequent nodes.
3. Stop when the pointer is `NULL`.

Example Code (C-like):

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

2.4.4 Searching in Singly Linked List

Search for a specific element by comparing each node's data.

Steps:

1. Start at the head node.
2. Compare the node's data with the target value.
3. Continue to the next node if not found.
4. Stop when found or when the end of the list is reached.

Example Code (C-like):

```
bool search(Node* head, int key) {
    Node* temp = head;
    while (temp != NULL) {
        if (temp->data == key) return true;
        temp = temp->next;
    }
    return false;
}
```

2.5 Advantages and Disadvantages of Linked Lists

2.5.1 Advantages

- **Dynamic Size:** Can grow or shrink as needed.
- **Efficient Insertions/Deletions:** No need to shift elements.
- **Memory Utilization:** Allocates memory as needed.

2.5.2 Disadvantages

- **Sequential Access:** Cannot access elements randomly.
- **Memory Overhead:** Requires extra memory for pointers.
- **Complex Implementation:** More challenging to implement compared to arrays.

2.6 Use Cases of Linked Lists

- **Dynamic Data Structures:** Stacks, Queues, and Hash Tables.
- **Undo Mechanisms:** Used in applications like text editors.
- **Memory Management:** Used in operating systems for memory allocation.

This concludes the detailed notes for **UNIT-II** covering Sorting Techniques and Linked Lists. Let me know if you need further elaboration or additional examples!### **10. Multidimensional Arrays**

Definition:
A multidimensional array is an array of arrays, where data is stored in a tabular form (rows and columns) or in higher dimensions.

10.1 Representation:

1. **Two-Dimensional Arrays (Matrices):**
   Represented as rows and columns.
   Example:

   ```
   2D Array:
   [ [1, 2, 3],
    [4, 5, 6],
    [7, 8, 9] ]
   ```

   Memory is stored row by row (row-major) or column by column (column-major).

2. **Higher-Dimensional Arrays:**
   For 3D or more dimensions, data is organized in layers.
   Example (3D Array):

   ```
   Layer 1: [ [1, 2], [3, 4] ]
   Layer 2: [ [5, 6], [7, 8] ]
   ```

10.2 Advantages of Multidimensional Arrays:

1. Facilitates the representation of data in tabular or matrix form.
2. Useful for complex data modeling (e.g., images, grids).

10.3 Applications of Multidimensional Arrays:

1. Graphical representations (images, game boards).
2. Matrices in mathematical computations.
3. Storing and manipulating tabular data.