Module (1)

Introduction to Data Structure:

- **Definition:**
  - Data structure is the organization of data along with a set of operations that can be applied to them, implemented using well-defined algorithms.
  - It encompasses how data elements are arranged, the operations applicable to the arrangement, and the algorithms for executing these operations.

Understanding Data Structures and Algorithms with Toy Examples:

- **Scenario I - Random Access:**
  - Books are arranged linearly.
  - Insertion and removal can happen at arbitrary locations without affecting others.
  - Operations are characterized by their randomness.
  - It is a way of arranging elements in memory such that an element can be accessed/visited directly without visiting other elements.
- **Scenario II - Last-In First-Out (LIFO) or First-In Last-Out (FILO):**
  - Books inserted and removed from one side only.
  - Removal occurs from the same side as the last insertion.
  - Illustrates the concept of a stack.
- **Scenario III - First-In First-Out (FIFO) or Last-In Last-Out (LILO):**
  - Insertion from one end, removal from the other.
  - Emphasizes the order of insertion and removal.
  - Demonstrates the concept of a queue.
- **Variations and Complexity:**
  - Other arrangements include hierarchical or network structures.
  - Each structure has associated operations like insertion and removal, executed through well-defined algorithms.

Data Structure Components:

- **Organization:**
  - How data elements are arranged, such as linearly or hierarchically.
- **Operations:**
  - Actions that can be performed on the arrangement, like insertion and removal.
- **Algorithms:**

- Well-defined sequences of sub-operations to execute operations efficiently.

Key Takeaways:

- Data structures provide a framework for organizing data.
- Understanding operations and algorithms is crucial for efficient data manipulation.
- Various arrangements cater to different needs, from linear to hierarchical or networked structures.

Conclusion:

- Data structures are fundamental in organizing and manipulating data efficiently, essential for various computational tasks.

Understanding Programs, Data Structures, and Algorithms:

Introduction:

- Recap of previous lesson: Definition of data structure and algorithm with examples.
- Focus of current lesson: Understanding programs and their relation to data structures and algorithms.

Example 1: Finding Factorial of an Integer:

- Problem: Calculate factorial of a given integer 'n'.
- Data Needed: Single value 'n'.
- Data Size: One.
- Storage: Primary memory (RAM).
- Data Structure: Primitive data structure (single variable).
- Algorithm: Iterative multiplication loop.
- Implementation: Example program in C language.

A primitive data type (or data structure) is one of the default data types (or data structures) of the underlying programming language, which are designed to store only one value.

Example 2: Searching in a Collection of Integers:

- Problem: Find a specific number in a collection.
- Data Needed: Collection of integer numbers.

- Data Size: Multiple values.
- Storage: Memory locations.
- Data Structure: Non-primitive data structures.
- Algorithm: Depends on the chosen data structure.
- Implementation: Example program using primitive data structures and alternative using a non-primitive data structure (array).

Understanding Programs:

- Definition: Programs consist of algorithms applied to data structures. A program is a set of instructions that tells a computer how to perform a specific task or tasks.
- Components:
  - Algorithm: Defined steps to solve a problem.
  - Data Structure: Organizes and stores data for efficient manipulation. It is a defined by data organization + set of operations + respective algorithms.
- Relation: Data structure and algorithm together define a program's functionality.
  Conclusion:
- Data structures organize data for efficient manipulation.
- Algorithms provide step-by-step instructions to solve problems.
- Programs combine algorithms with data structures to achieve desired functionality.
- Course Focus: Learning various data structures, their operations, algorithms, implementations, and real-world applications.

Basic Terminologies I

Introduction:

- Recap of previous session covering data structures, algorithms, and programs.
- Current focus: Defining basic terminologies used in the course.

Data, Data Element, and Data Type:

- **Data:** Collection of raw facts, examples include integers, cities, or records.
- **Data Element:** Logical unit constituting data, e.g., each integer in a collection of integers.
- **Data Type:** Specifies the type of data each element can hold, such as integer, string, or user-defined structures.

Storage Structure of Data Collection in Memory:

- **Primary Memory (RAM):** Volatile memory for temporary storage during program execution. Instances of data structures are defined in this memory.
- **Secondary Memory (Hard Disk):** Non-volatile memory for permanent storage.
- **Memory Allocation Patterns:**
  - Contiguous Memory Allocation: Elements stored at contiguous memory locations.
  - Non-contiguous Memory Allocation: Elements stored at arbitrary memory locations.

Contiguous Memory Allocation:

- Elements stored in linear fashion.
- Supports random access memory, allowing direct access to any element.
- No additional memory required for storing addresses of other elements.
- Disadvantages:
  - Static allocation, unable to change size during runtime.
  - OS defragmentation if contiguous memory locations are not available
  - Unused memory space may occur.
  - Requires large RAM for storing large data collections.

Non-contiguous Memory Allocation:

- Enables dynamic memory allocation, allowing growth and shrinkage during runtime.
- Does not require large contiguous memory blocks.
- Disadvantages:
  - Requires additional memory for storing addresses of other elements.
  - Higher storage requirement compared to contiguous allocation.
  - Accessing elements may require traversing through other elements to obtain addresses.

Conclusion:

- Understanding basic terminologies such as data, data element, and data type is crucial.
- Memory allocation patterns, including contiguous and non-contiguous, impact program efficiency and flexibility.
- Each allocation method has advantages and disadvantages, influencing decision-making in program design and implementation.

Basic Terminologies - II

Week 1 Lecture: Basic Terminologies II

Operations on Data Structures:

- **Creation:** Generates an empty instance of a data structure.
- **Insertion:** Adds an element to an instance of the data structure.
- **Deletion:** Removes an element from the data structure.
- **Updation:** Modifies the value of a data element.
- **Searching:** Determines the existence or non-existence of a data element.
- **Traversal:** Visits every data element in the data structure.
- **Sorting:** Arranges data elements in ascending or descending order.
- **Merging:** Combines data elements from two or more data structures.

Properties of an Algorithm:

- **Input Specified:** Clearly defined input(s); can have zero or more inputs.
- **Output Specified:** Clearly defined output(s); can have one or more outputs.
- **Definiteness:** Every step of the algorithm must be unambiguously defined.
- **Finiteness:** Algorithm should terminate after a finite number of steps.
- **Effectiveness:** Steps should be achievable within a specified time with pen and pencil.

Complexity of Algorithms:

- Defined by time complexity and space complexity.
- Time complexity: Time taken by the algorithm to execute.
- Space complexity: Space required by the algorithm to solve the problem.
- Represented using asymptotic notation.

Types of Data Structures:

- **Primitive or Non-Primitive**: based on number of values that can be stored in a structure.
- **Linear or Non-linear:** Based on arrangement of elements (sequential or hierarchical).
- **Homogeneous or Heterogeneous:**
  - Homogeneous: All elements have single-value data and same data type
  - Heterogeneous: Elements can have multiple values (e.g., value and address). A data structure that allows you to store different data type values.
- **Static or Dynamic:**
  - Static: Size of storage cannot be changed during runtime.
  - Dynamic: Size of storage can be changed during runtime.

Data Structures Covered in the Course:

- Array
- Linked List
- Stack and Queue
- Tree or Binary Tree
- Hash
- Graph

Conclusion:

- Understanding operations and properties of algorithms is essential for problem-solving.
- Data structures vary in characteristics such as arrangement, homogeneity, and dynamicity.
- The course will focus on non-primitive data structures, including arrays, linked lists, stacks, queues, trees, hash tables, and graphs.

Week 1 Lecture: Array

Definition of Array:

- An array is a collection of finite, homogeneous data elements stored at contiguous memory locations.
- Example: Collection of numbers (5, 7, 2, 8, 1, 3) stored in contiguous memory.

Elements and Index:

- **Elements:** Data values within the array (e.g., 5, 7, 2).
- **Index:** Identifies the location of elements in the array.

Homogeneity and Linearity:

- **Homogeneous:** All elements are of the same data type.
- **Linear:** Elements are arranged sequentially (e.g., 1st, 2nd, 3rd).

Accessing Elements:

- Elements are accessed using indices.
- Indexing starts from a lower bound and ends at an upper bound.
- Example: A(1) represents the first element, A(2) the second, and so on.

Random Access:

- Arrays support random access, allowing direct access to any element using its index (e.g., A(i)).

Types of Arrays:

- **One-dimensional Array:** Collection of zero-dimensional arrays (single values).
- **Two-dimensional Array:** Array of one-dimensional arrays (rows and columns).
- **Three-dimensional Array:** Array of two-dimensional arrays (plates, rows, and columns).

Accessing Elements in Multi-dimensional Arrays:

- Accessed using row, column, and plate indices.
- Example: A(k)(j)(i) represents the element in the kth plate, jth row, and ith column.

Generalization to n-dimensional Array:

- Array of n-1 dimensional arrays with the same size.
- Accessed using indices up to the nth dimension.

Conclusion:

- Arrays provide efficient storage and retrieval of data.
- Multi-dimensional arrays extend the concept to higher dimensions.
- Understanding array dimensions and indexing is crucial for efficient data manipulation.

Week 1 Lecture: Representation of Array in Memory

Linear Representation in Memory:

- Arrays, represented pictorially in multi-dimensional forms, are stored linearly in computer memory.
- RAM (Random Access Memory) is a linear arrangement of memory cells, each storing one byte of information.

One-dimensional Array Representation:

- Memory is a collection of memory locations.
- An array of size n and data type 'type' requires n*k bytes of memory (k bytes per element).
- Elements are stored consecutively, starting from the base address.
- The address of an arbitrary element A(i) is calculated as Base + (i - lower_bound) * k.

Two-dimensional Array Representation (Row Major Order):

- For an n*m array, total memory required is n*m*k.
- Elements are stored row-wise: first row, second row, and so on.
- Address of element A(i)(j) = Base + ((i - lower_bound_row) *columns + (j - lower_bound_column))* k.

Two-dimensional Array Representation (Column Major Order):

- Elements are stored column-wise: first column, second column, and so on.
- Address of element A(i)(j) = Base + ((j - lower_bound_column) *rows + (i - lower_bound_row))* k.

Three-dimensional Array Representation:

- Array of two-dimensional arrays (plates).
- Address of element A(i)(j)(k) is calculated similarly, considering complete plates, rows, and columns.

Generalization to n-dimensional Array:

- For an n-dimensional array, address calculation involves traversing through each dimension.
- Address of an element in an n-dimensional array is calculated by considering complete (k-1)-dimensional arrays in each dimension.

Conclusion:

- Arrays, though visually represented in multi-dimensional forms, are stored linearly in computer memory.
- Understanding the linear representation and address calculation of array elements is essential for efficient data manipulation.

Operations on Array - Create Array

Introduction:

- In this session, we'll discuss operations on arrays, specifically focusing on how to create an array and allocate memory for it in RAM.

Creating an Array:

- The create operation initializes an empty array and allocates the required memory.
- Arrays in C can be created at compile-time or runtime.

Compile-time Array Creation:

1. **Static Memory Allocation:**
   - Memory is allocated during compilation.
   - To create a one-dimensional array:

     ```
     int array[100];
     ```

   - For a two-dimensional array:

     ```
     int array[20][30];
     ```

     - Memory required = number of elements *size of each element (e.g., 20* 30 * 2 bytes for integers).
   - For a three-dimensional array:

     ```
     int array[10][20][30];
     ```

     - Memory required = number of elements *size of each element (e.g., 10* 20 *30* 2 bytes for integers).

Runtime Array Creation:

1. **Dynamic Memory Allocation:**
   - Memory is allocated during program execution using functions like `malloc`.
   - To create a one-dimensional array:

     ```
     int* array = (int*)malloc(100 * sizeof(int));
     ```

     - `malloc` reserves memory and returns the address of the block, assigned to a pointer.
   - To create a two-dimensional array:

     ```
     int** array = (int**)malloc(5 * sizeof(int*));
     for (int i = 0; i < 5; i++) {
         array[i] = (int*)malloc(10 * sizeof(int));
     }
     ```

     - `malloc` is used multiple times to allocate memory for each row separately.
   - For a three-dimensional array, the process involves multiple levels of memory allocation, each for different dimensions.

General Syntax for Array Declaration:

- **Compile-time (Static)**

```
data_type array_name[size1][size2]...[sizeN];
```

- **Runtime (Dynamic)**

```
data_type** array = (data_type**)malloc(size1 * sizeof(data_type*));
for (int i = 0; i < size1; i++) {
    array[i] = (data_type*)malloc(size2 * sizeof(data_type));
}
```

Summary:

- Creating an array involves either compile-time static allocation or runtime dynamic allocation.
- Compile-time allocation is straightforward with direct array declarations.
- Runtime allocation involves using `malloc` to reserve memory dynamically, often requiring multiple steps for multi-dimensional arrays.

Week 1 Lecture: Operations on Array - Traversal and Search Operations

Traversal and Search Operation

Objective:

- Visit all the elements in an array and display them.

Implementation:

- **C Function for Traversal:**

```
void display(int arr[], int n) {
    for (int i = 0; i < n; i++) {
        printf("%d ", arr[i]);
    }
}
```

  - The function takes an array and its size as parameters.
  - Uses a `for` loop to iterate from the first index (0) to the last index (n-1).
  - Displays each element during the traversal.

Complexity:

- **Time Complexity:**
  - Defined by the number of elements in the array (n).
  - Each element is visited once, hence the time complexity is O(n) or Θ(n).
- **Space Complexity:**
  - The space needed is for storing the array elements, which is O(n) or Θ(n).

Search Operation

Objective:

- Check if a given element is present in an array.

Implementation:

- **C Function for Linear Search:**

```c
int search(int arr[], int x, int n) {
    for (int i = 0; i < n; i++) {
        if (arr[i] == x) {
            return i; // Element found, return index
        }
    }
    return -1; // Element not found
}
```

  - The function takes an array, the element to search for, and the array size as parameters.
  - Scans the array from the first index (0) to the last index (n-1).
  - Compares each element with the search element `x`.
  - Returns the index if the element is found, otherwise returns -1.

Complexity:

- **Time Complexity:**
  - **Best Case:** Θ(1) - The element is found at the first index.
  - **Worst Case:** Θ(n) - The element is found at the last index or not found at all.
  - **Average Case:** Θ(n) - The element can be at any position with equal probability.
    - For an array of size n, the average number of comparisons is (1 + 2 + ... + n) / n = (n + 1) / 2 ≈ n/2.
- **Space Complexity:**

- The memory required for storing the elements in the array is O(n) or Θ(n).

Example of the Search Function:

```c
#include <stdio.h>

int search(int arr[], int x, int n) {
    for (int i = 0; i < n; i++) {
        if (arr[i] == x) {
            return i; // Element found
        }
    }
    return -1; // Element not found
}

int main() {
    int arr[] = {3, 5, 7, 10, 15};
    int n = sizeof(arr) / sizeof(arr[0]);
    int x = 10;
    int result = search(arr, x, n);
    if (result != -1) {
        printf("Element found at index %d\n", result);
    } else {
        printf("Element not found\n");
    }
    return 0;
}
```

Summary:

- **Traversal:** Visit and display each element of an array from start to end.
  - **Time Complexity:** O(n)
  - **Space Complexity:** O(n)
- **Search:** Find if an element exists in the array.
  - **Time Complexity:**
    - Best Case: Θ(1)
    - Worst Case: Θ(n)
    - Average Case: Θ(n)
  - **Space Complexity:** O(n)

Operation on Array - Insert, Delete and Update

Insertion Operation

Task:

- Insert a new element at any arbitrary index `i` in an array.

Assumptions:

1. **Memory Reservation:** The array has enough reserved memory to accommodate the new element.
2. **Contiguous Memory:** Elements are stored at contiguous memory locations.
3. **Update After Insertion:** After inserting, the first `n+1` locations should be occupied by `n+1` elements.

Process:

1. **Identify Insertion Index:** Assume insertion at index 3 (4th location).
2. **Create Free Space:** Move all elements from index `i` to the last element one position backward starting from the last element to free space at index `i`.
   - Move the element at the upper bound index to the upper bound + 1 index.
   - Repeat this process backward until index `i` is reached.
3. **Insert New Element:** Assign the new element to the free space at index `i`.
4. **Update Array Size:** Increment the array size by 1.

Algorithm:

- For each index `j` from `upper_bound` to `i`:

```
arr[j+1] = arr[j];
```

- Insert the new element at index `i`:

```
arr[i] = new_element;
```

- Update the array size.

Time Complexity:

- **Best Case:** Inserting after the last element – O(1).
- **Worst Case:** Inserting at the first location – O(n).
- **Average Case:** Insert at any position with equal probability – O(n).

Delete Operation

Task:

- Delete an element from an arbitrary index `i`.

Process:

1. **Identify Deletion Index:** Assume deletion at index 3.
2. **Move Elements Forward:** Move all elements after index `i` one position forward.
    - Start from index `i` and move each element to the previous index.
3. **Update Array Size:** Decrement the array size by 1.

Algorithm:

- For each index `j` from `i` to `upper_bound-1`:

```
arr[j] = arr[j+1];
```

- Update the array size.

Time Complexity:

- **Best Case:** Deleting the last element – O(1).
- **Worst Case:** Deleting the first element – O(n).
- **Average Case:** Delete from any location with equal probability – O(n).

Update Operation

Task:

- Change the value of an element at index `i`.

Process:

- Directly visit the index and update the value.

Algorithm:

- Update the value at index `i`:

```
arr[i] = new_value;
```

Time Complexity:

- **Update:** O(1) (due to random access in arrays).

Summary and Program

- **Creation:** Define an array using compile-time memory allocation.

- **Search:** Use a function to check for a value.
- **Insert:** Insert a new element using the defined insertion function.
- **Delete:** Remove an element using the defined delete function.
- **Display:** Show the elements after insertions and deletions.

Complete Program Example

```
#include <iostream>

using namespace std;

void insertElement(int arr[], int& n, int index, int element) {
    for (int i = n; i > index; --i) {
        arr[i] = arr[i - 1];
    }
    arr[index] = element;
    ++n;
}

void deleteElement(int arr[], int& n, int index) {
    for (int i = index; i < n - 1; ++i) {
        arr[i] = arr[i + 1];
    }
    --n;
}

void displayArray(int arr[], int n) {
    for (int i = 0; i < n; ++i) {
        cout << arr[i] << " ";
    }
    cout << endl;
}

int main() {
    int arr[100] = {10, 20, 30, 40, 50};
    int n = 5;

    // Insert element 15 at index 3
    insertElement(arr, n, 3, 15);
    displayArray(arr, n);

    // Delete element at index 2
    deleteElement(arr, n, 2);
    displayArray(arr, n);

    return 0;
}
```

Key Points

- **Insertion:** Shift elements to make space, then insert and update size.
- **Deletion:** Shift elements to fill space, then update size.
- **Update:** Directly access and change the value at the specified index.

Sparse Matrix - Part I

Introduction to Sparse Matrix

Definition:

- A sparse matrix is a type of matrix with a high proportion of zero elements.
- Different definitions in literature:
  - More than half of the elements are zero.
  - More than two-thirds of the elements are zero.
  - Generalized: A matrix with a significant number of zero entries.

Motivation:

- Storing all elements, including zeros, can waste memory.
- For large matrices, storing only non-zero elements can save significant space.

Storage Strategies

Two Cases:

1. Store all elements, including zeros.
2. Store only the non-zero elements to save space.

When to Store Non-Zero Elements:

- Effective for large matrices to reduce memory usage.

Types of Sparse Matrices

1. **Triangular Matrices:**
   - **Left Lower Triangular Matrix:** Non-zero elements are below the diagonal.
   - **Right Lower Triangular Matrix:** Non-zero elements are above the diagonal.
2. **Diagonal Matrix:** Only diagonal elements are non-zero.
3. **Tri-Diagonal Matrix:** Non-zero elements are on the diagonal, and the first diagonals above and below it.

Sparsity is number of zeroes/ number of total elements

Storing Sparse Matrices

Example: Left Lower Triangular Matrix:

- Non-zero elements are stored in a linear arrangement using row-major order.

Row-Major Order Storage:

- Store elements row by row.
- Example:
  - First row: 1 element
  - Second row: 2 elements
  - Third row: 3 elements, etc.

Memory Arrangement:

- Transform the 2D matrix into a 1D array, storing only non-zero elements.

Algorithm for Storing Non-Zero Elements

1. **Create a 1D Array:**
   - Size of the array should ideally match the number of non-zero elements.
   - If the exact number is unknown, reserve a larger space.
2. **Traverse the Matrix:**
   - Use two loops: outer loop for rows, inner loop for columns.
   - Check the relationship between row index `i` and column index `j`.
   - If `i >= j` (for lower triangular), store the element.
3. **Code Example:**

```
int sparse_matrix_to_array(int matrix[][N], int n, int result[]) {
    int k = 0;
    for (int i = 0; i < n; ++i) {
        for (int j = 0; j <= i; ++j) {  // Only consider lower triangular part
            if (matrix[i][j] != 0) {
                result[k++] = matrix[i][j];
            }
        }
    }
    return k;  // Return number of non-zero elements
}
```

Accessing Elements in Sparse Matrix

Effective Address Calculation:

- **Concept:** Calculate the address of any element `A[i][j]` in the 1D array.
- **Formula:**

```
effective_address = base_address + sum_of_non_zero_elements_before_A[i][j]
```

- **Steps:**
  1. Count elements in complete rows before the target row.
  2. Add elements in the target row up to column `j`.

Example Calculation:

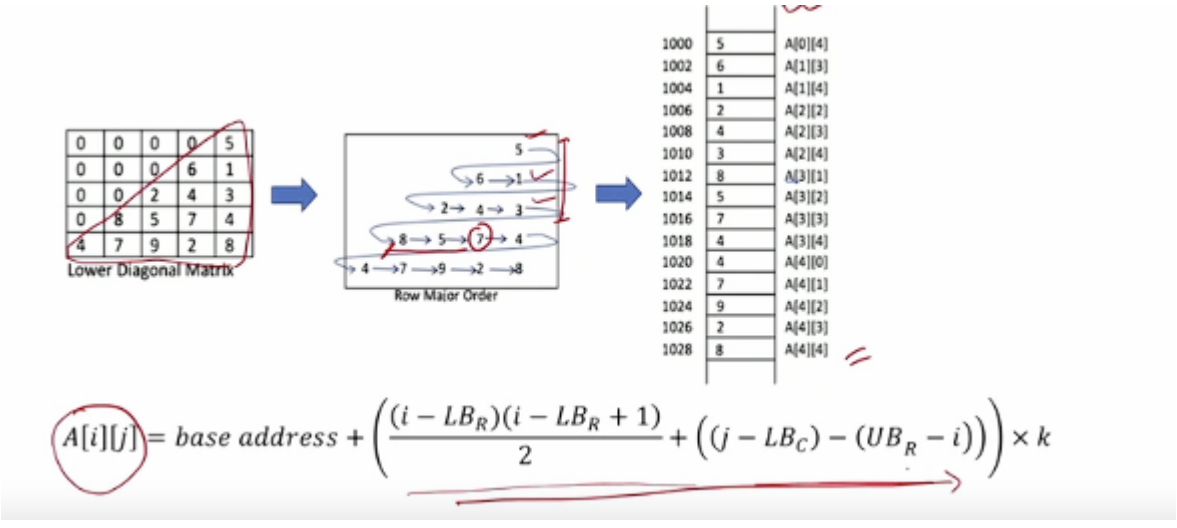
- **Target Element:** `A[4][3]`
- **Count Elements:**
  - First row: 1 element
  - Second row: 2 elements
  - Third row: 3 elements
  - Fourth row up to column 3: 4 elements
- **Total:** (1 + 2 + 3 + 3 = 9) elements before `A[4][3]`.

Effective Address Formula:

- For left lower triangular matrix:

```
effective_address = base_address + (i * (i + 1) / 2) + j
```

- For right lower triangular matrix



Generalization

- The same principles apply to other types of triangular matrices and can be adapted for column-major order.

Column-Major Order:

- Store elements column by column.

Effective Address Formula for Right Lower Triangular Matrix:

- Similar steps but consider elements in column-major order.

Summary

- **Sparse Matrices:** Efficient storage for matrices with many zero elements.
- **Types:** Triangular, diagonal, and tri-diagonal matrices.
- **Storage:** Transform 2D matrices into 1D arrays, storing only non-zero elements.
- **Access:** Calculate effective addresses for elements using predefined formulas.

Key Points:

- Understand when and why to use sparse matrices.
- Efficiently store and access non-zero elements to save memory.

Sparse Matrix - Part II

Transforming a Diagonal Matrix into a Linear Array

Objective:

- Convert a diagonal matrix, which is a type of sparse matrix, into a linear array format.

Process:

1. **Store Only Diagonal Elements:**
    - In a diagonal matrix, only the diagonal elements are non-zero.
    - Store these diagonal elements linearly.

Effective Address Calculation:

- Use a specific expression to estimate the effective address for each diagonal element.

Tri-diagonal Matrix Transformation

Definition:

- A tri-diagonal matrix has non-zero elements on the main diagonal, one above the main diagonal, and one below the main diagonal.

Storage:

- Store non-zero elements row-wise: 1st row, 2nd row, 3rd row, etc.
- Use an expression to estimate the effective address of these elements.

Example:

- For element (A(3,2)), calculate its effective address, such as 1016.

Generalized Sparse Matrix

Characteristics:

- No specific pattern for non-zero elements.
- Any index in the matrix can have a zero or non-zero value.
- Cannot use simple linear transformations for effective address calculation.

Approaches for Storing Sparse Matrices:

Using an Array

1. **Structure:**
   - Use a 2D array with dimensions based on the number of non-zero elements and three columns.
   - Columns represent:
     - **Column 1:** Row index
     - **Column 2:** Column index
     - **Column 3:** Non-zero element
2. **Example:**
   - Define a matrix `b` of dimensions 5x3 (5 non-zero elements, 3 columns).
   - Store each non-zero element along with its row and column index.

```
b = [
  [0, 1, 5],  // Element 5 at row 0, column 1
  [1, 0, 7],  // Element 7 at row 1, column 0
  ...
]
```

Search Operation:

- To search for an element, scan the entire array since any index can be zero or non-zero.

Using Dynamic Memory Allocation (Linked List)

Concept:

- Store non-zero elements dynamically using linked lists.
- Each node in the linked list will store:
  - The row index
  - The column index
  - The non-zero value
  - A pointer to the next node

Note:

- Detailed discussion on implementation using linked lists will be covered later.

Summary

- **Diagonal Matrix Transformation:** Store only diagonal elements linearly.
- **Tri-diagonal Matrix:** Store non-zero elements row-wise and calculate effective addresses.
- **Generalized Sparse Matrix:** Use arrays or dynamic structures like linked lists to store non-zero elements along with their indices.

Key Points

1. **Storage Methods:**
   - Linear arrays for matrices with specific patterns (diagonal, tri-diagonal).
   - Arrays with row and column indices for generalized sparse matrices.
   - Dynamic structures like linked lists for more flexible storage.
2. **Effective Address Calculation:**
   - Use expressions to estimate addresses for elements in structured sparse matrices.
3. **Search and Other Operations:**
   - For generalized sparse matrices, operations like search require scanning the entire storage structure.

These notes cover the key concepts and processes for transforming and managing sparse matrices as discussed in the lecture.

Abstract Data Type (ADT)

Introduction to ADT

Scenario:

- John, a novice programmer, needs to work with arrays but lacks knowledge on implementing array operations.
- Bob, an expert programmer, creates an Abstract Data Type (ADT) for arrays to help John.

Abstract Data Type (ADT):

- ADT consists of:
  - Storage mechanism
  - Data organization
  - Set of functions for accessing data

Implementation:

- Bob provides the ADT to John, who can then use it to solve problems without knowing its internal implementation.
- John uses the functions provided by Bob to manipulate data within the ADT.

Definition of ADT

Programming Terminology:

- ADTs are defined using the `class` keyword in programming languages.

Components:

- **Data Members:** Variables for storing data.
- **Function Members:** Functions to operate on the data.

Access Specifiers:

- **Private:** Members restricted to ADT itself.
- **Public:** Members accessible to client or application programmers.

Core Concepts of ADT

Implementer vs. Client:

- **Implementer:** Creates the ADT.

- **Client:** Uses the ADT without knowing its implementation details.

User's Perspective:

- ADT defines data from the client's perspective.
- Clients interact with the ADT using its provided functions (API).

Properties of ADT

1. **Encapsulation:**
    - Storage data and functions combined as a single unit.
2. **Data Hiding:**
    - Private members are hidden from application programmers.
3. **Inheritance:**
    - One ADT can inherit from another ADT, enabling hierarchical organization.

Advantages of ADT

1. **Client Convenience:**
    - Clients use ADTs without knowing internal implementation details.
2. **Ease of Updating:**
    - Implementers can update ADTs without affecting client programs.
    - Changes in internal structure can be made seamlessly.

Example:

```
class Array {
private:
    // Private data members
    int* elements;
    int size;

    // Private member functions
    void resize(int new_size);
public:
    // Public member functions
    Array();
    ~Array();
    void insert(int element, int index);
    void remove(int index);
    int get(int index);
};
```

Which of the following are not access specifiers? answer: static

public
static
(static is used to define elements that are common to all instances of a class or do not require an instance to be accessed)
private
protected

Conclusion:

- ADTs provide a way to separate data structure implementation from its usage.
- Clients interact with ADTs using a set of functions, without knowing implementation details.
- Implementers can update ADTs independently, ensuring seamless integration with client programs.
- Public members are the API connecting the data structure and the application program.

These notes cover the fundamentals of Abstract Data Types (ADTs) as discussed in the lecture.

Assignments

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

int main() {
    int *array = NULL;  // Pointer to dynamically allocated array
    int i = 0, value, size = 0;

    // Input elements into the array
    printf("Enter elements (enter -1 to stop):\n");
    while(1) {
        printf("Element %d: ", i+1);
        scanf("%d", &value);

        if (value == -1) {
            break;
        }

        // Reallocate memory for the array
        size++;
        array = (int*) realloc(array, size * sizeof(int));
        if (array == NULL) {
            printf("Memory allocation failed!\n");
            return 1; // Exit the program with an error code
        }
```

```
        array[i] = value;
        i++;
    }

    // Print the elements of the array
    printf("The elements in the array are:\n");
    for(int j = 0; j < i; j++) {
        printf("%d ", array[j]);
    }

    // Free the dynamically allocated memory
    free(array);

    return 0;
}
```

```
#include<stdio.h>
int main(){

    int size;
    printf("Enter the size of array: ");
    scanf("%d",&size);
    int arr[size];
    for (int i =0; i < size;i++)
    {
        printf("element %d: ",i);
        scanf("%d",&arr[i]);
    }
    for(int i =size-1;i>=0;i--){
        printf("%d\n",arr[i]);
    }
    return 0;

}
```

```
#include<stdio.h>
int main(){

    int size;
    int sum =0;
    printf("Enter the size of array: ");
    scanf("%d",&size);
    int arr[size];
    for (int i =0; i < size;i++)
    {
```

```
        printf("element %d: ",i);
        scanf("%d",&arr[i]);
    }
    for (int i =0; i < size;i++)
    {
        sum+=arr[i];
    }
    printf("sum: %d",sum);
    return 0;

}
```

```
#include<stdio.h>
int main(){

    int size;
    printf("Enter the size of array: ");
    scanf("%d",&size);
    int arr[size];
    int brr[size];
    for (int i =0; i < size;i++)
    {
        printf("element %d: ",i);
        scanf("%d",&arr[i]);
    }
    printf("another array: \n");
    for (int i =0; i < size;i++)
    {
        brr[i]=arr[i]; //copying operation
        printf("%d\n",brr[i]);
    }
    return 0;

}
```