**Lesson 5: Linked List, Stack, Queue, Tree**

1. **Linked List**:  
   A dynamic data structure consisting of nodes where each node contains data and a reference (or a link) to the next node in the sequence. Linked lists allow for efficient insertion and deletion at any position within the list, unlike arrays that require shifting elements.
2. **Stack**:  
   A linear data structure following the Last In First Out (LIFO) principle. Elements are pushed (added) to the top of the stack and popped (removed) from the top, allowing access to only the most recently added item. Stacks are commonly used in recursive algorithms and for tracking function calls.
3. **Queue**:  
   A linear data structure that operates on the First In First Out (FIFO) principle. Elements are enqueued (added) at the back and dequeued (removed) from the front. Queues are often used in scheduling algorithms, such as in operating systems and breadth-first search algorithms.
4. **Tree**:  
   A hierarchical data structure where each node has a value and references to its child nodes. The top node is called the root, and each node can have multiple children but only one parent (except for the root). Common types of trees include binary trees, binary search trees (BST), and balanced trees like AVL trees. Trees are widely used in databases, file systems, and AI algorithms.

**Data Structure**

A data structure is a collection of data elements that have one or more specific relationships among them. Data elements are not isolated; they are connected by certain relationships. These relationships between data elements form a structure, and there are four types:

1. **Set**:  
   In this structure, elements have no other relationship except that they belong to the same set. An example is an array in C.
2. **Linear Structure**:  
   Elements have one-to-one relationships, such as a linked list or a stack.
3. **Tree Structure**:  
   Elements have a one-to-many relationship, like in a tree data structure.
4. **Graph Structure**:  
   Elements have many-to-many relationships, as seen in graphs.

**Data Structures: Arrays and Structs**

An **array** is a structure that allows multiple elements to be stored continuously in memory, and its memory allocation is also continuous. Elements in an array are accessed via their index, starting from 0.

**Advantages:**

1. Fast element lookup by index.
2. Easy to traverse using the index.

**Disadvantages:**

1. Once an array's size is fixed, it cannot be expanded.
2. Arrays can only store one type of data.
3. Adding or deleting elements is slow because other elements need to be shifted.

**Suitable Scenarios:**

* Frequent lookups.
* Minimal storage space required.
* Rare additions or deletions of elements.

**Data Structures: Linked Lists**

A **linked list** is a non-continuous, non-sequential storage structure in physical memory. The logical order of data elements is achieved through pointers. Each element contains two parts: a data field (memory space for the element) and a pointer field that stores the address of the next node. Depending on how pointers are arranged, linked lists can form various structures such as singly linked lists, doubly linked lists, and circular linked lists.

**Advantages:**

* No need for pre-defined capacity; elements can be added or removed freely.
* Adding or deleting elements is fast, as it only requires updating the pointers of the adjacent nodes.

**Disadvantages:**

* Due to many pointer fields, it consumes more memory.
* Finding an element requires traversing the list, which is time-consuming.

**Suitable Scenarios:**

* Small data sets with frequent additions and deletions.

**Data Structures: Linked List**

In terms of logical structure, the data elements in a linked list are organized sequentially, but in actual memory storage, they are not stored contiguously like arrays. Instead, the data elements are randomly scattered across different memory locations. This storage structure is called the **linked storage of a linear list**.

Since the data is stored non-contiguously, each data element must have a pointer that links to its immediate successor to maintain the logical order. Thus, every data element points to the next one, with the last element pointing to NULL (empty).

**Key Components:**

1. **Data Field**: The actual information of the element.
2. **Pointer Field**: A pointer that stores the address of the next element in the sequence.

Here’s an example of how a linked list is defined in C:

*struct link {*

*int data; // Defines the data field*

*struct link\* next; // Defines the pointer field, storing the next node's information*

*};*

This structure allows flexibility in adding or removing elements, as each node is independent and connected only through pointers.

**Data Structures: Linked List Operations (Creation, Deletion, Traversal, Searching, Inserting Nodes)**

This code demonstrates key linked list operations like creating a list, deleting a node, inserting a node, and the logic behind traversing and manipulating the linked list.

**1. Creating a Linked List:**

*struct link\* create(int n) {*

*struct link \*headnode, \*node;*

*headnode = (struct link\*) malloc(sizeof(struct link)); // Allocate memory for the head node*

*headnode->next = NULL; // Set the head node’s pointer to NULL*

*for (int i = 0; i < n; i++) {*

*node = (struct link\*) malloc(sizeof(struct link)); // Allocate memory for new node*

*scanf("%d", &node->data); // Input data into the new node*

*node->next = headnode->next; // New node points to the current head’s next*

*headnode->next = node; // Head node points to the new node (new node becomes the last)*

*}*

*return headnode; // Return the head node*

*}*

This function creates a linked list by taking in a number n, which specifies the number of nodes. Each new node is added at the head of the list.

**2. Deleting a Node:**

*void deleteNode(struct link\* head, int n) {*

*struct link \*p = head, \*pr = head;*

*int i = 0;*

*// Traverse to the nth node*

*while (i < n && p != NULL) {*

*pr = p; // Save current node as previous*

*p = p->next; // Move to the next node*

*i++;*

*}*

*if (p != NULL) {*

*pr->next = p->next; // Skip over the node to be deleted*

*free(p); // Free memory for the deleted node*

*} else {*

*printf("Node does not exist!\n");*

*}*

*}*

This function deletes the node at position n in the linked list. It checks if the node exists, and then adjusts the pointers to remove the node.

**3. Inserting a Node:**

*void insertNode(struct link\* head, int n) {*

*struct link \*p = head, \*pr;*

*pr = (struct link\*) malloc(sizeof(struct link)); // Allocate memory for new node*

*printf("Input data:\n");*

*scanf("%d", &pr->data); // Input data for new node*

*int i = 0;*

*while (i < n && p != NULL) { // Traverse to the insert position*

*p = p->next;*

*i++;*

*}*

*if (p != NULL) {*

*pr->next = p->next; // Point the new node to the next node in the list*

*p->next = pr; // Insert the new node in the list*

*} else {*

*printf("Node does not exist!\n");*

*}*

*}*

This function inserts a new node at position n. It adjusts the pointers to ensure the new node is inserted in the right position.

**4. Important Points:**

* while(p->next != NULL): When used in loops, it stops just before the tail node. If used for outputting node data, the tail node’s data will not be printed.
* while(p != NULL): This condition includes the tail node but stops when p points to an unallocated or invalid memory area.

**Summary:**

* **Creating a linked list** involves allocating memory for nodes, inserting data, and linking nodes.
* **Deleting a node** requires traversing to the desired node, adjusting pointers, and freeing memory.
* **Inserting a node** requires finding the correct position, adjusting pointers, and linking the new node into the list.

**Data Structures: Linked List Operations (Create, Delete, Traverse, Search, Modify, Reverse)**

Here is a concise overview of various linked list operations, including creation, deletion, traversal, searching, modifying nodes, and reversing the list.

**1. Modify a Node in the Linked List:**

*void change(struct Stu\* head, int n) {*

*struct Stu\* p = head;*

*int i = 0;*

*// Traverse to the nth node*

*while (i < n && p != NULL) {*

*p = p->next;*

*i++;*

*}*

*if (p != NULL) {*

*printf("请输入修改之后的值:\n");*

*scanf("%d%s", &p->id, p->name); // Modify the node's data*

*} else {*

*printf("节点不存在！\n"); // If the node doesn't exist*

*}*

*}*

This function modifies the node at position n by allowing input for new values, provided the node exists.

**2. Reverse a Linked List:**

*STU\* link\_reversed\_order(STU\* head) {*

*STU \*pf = NULL, \*pb = NULL, \*tmp = NULL;*

*pf = head; // Initialize pf with the head*

*if (head == NULL) {*

*printf("链表为空，不需要逆序！\n"); // Empty list, no need to reverse*

*return head;*

*} else if (head->next == NULL) {*

*printf("链表只有一个节点，不需要逆序！\n"); // Single node, no need to reverse*

*return head;*

*} else {*

*pb = pf->next; // pb points to the next node*

*head->next = NULL; // Set head’s next to NULL (head becomes the tail)*

*while (pb != NULL) {*

*tmp = pb; // Temporarily store pb*

*pb = pb->next; // Move pb to the next node*

*tmp->next = pf; // Reverse the link*

*pf = tmp; // Move pf to tmp*

*}*

*head = pf; // New head of the reversed list*

*return head;*

*}*

*}*

This function reverses a linked list by adjusting the pointers to reverse the direction of the links between nodes. It handles special cases, such as empty lists or lists with only one node.

**Summary:**

* **Modifying a node** in the linked list is done by traversing to the required position and updating the node's data.
* **Reversing a linked list** involves changing the direction of the pointers so that the list elements are stored in the opposite order.

These operations are crucial for dynamic management of data in linked lists.

**Data Structures: Double Circular Linked List**

A **Double Circular Linked List** is a type of linked list where each node has pointers to both its previous and next nodes, and the last node connects back to the first node, forming a circular structure.

**1. Structure of Double Circular Linked List Node:**

*struct doubleCircularLinkedList {*

*struct doubleCircularLinkedList \*prior; // Pointer to the previous node*

*int data; // Data field*

*struct doubleCircularLinkedList \*next; // Pointer to the next node*

*};*

Each node contains three fields:

* **prior**: A pointer to the previous node.
* **data**: Stores the node's data.
* **next**: A pointer to the next node.

**2. Creating a Double Circular Linked List:**

*struct doubleCircularLinkedList\* createList() {*

*// Create a head node (often used as a placeholder or control node)*

*struct doubleCircularLinkedList\* headNode =*

*(struct doubleCircularLinkedList\*)malloc(sizeof(struct doubleCircularLinkedList));*

*// As it's a circular linked list, initialize head node's prior and next pointers to point to itself*

*headNode->prior = headNode;*

*headNode->next = headNode;*

*return headNode;*

*}*

* The **headNode** is initialized, and both its **prior** and **next** pointers point to itself, as it is the only node in the list initially. This structure ensures the circular nature of the list.

**Key Points:**

1. **Double Circular Linked List** allows bidirectional traversal due to pointers to both the previous and next nodes.
2. **Circular Structure**: The list loops back, connecting the last node to the first node.
3. **Efficient Insertions/Deletions**: In a double linked list, insertion and deletion of nodes can be done easily at both ends.

This type of list is useful in scenarios where you need to traverse in both directions and maintain a circular connection, such as in buffering or navigation systems.

**Data Structures: Double Circular Linked List - Node Creation and Insertion**

**1. Creating a New Node:**

*struct doubleCircularLinkedList\* createNode(int data) {*

*// Dynamically allocate memory for the new node*

*struct doubleCircularLinkedList\* newNode =*

*(struct doubleCircularLinkedList\*)malloc(sizeof(struct doubleCircularLinkedList));*

*// Initialize the node with given data and set both prior and next to NULL*

*newNode->data = data; // Set the data value*

*newNode->prior = NULL; // Initialize previous pointer to NULL*

*newNode->next = NULL; // Initialize next pointer to NULL*

*return newNode; // Return the newly created node*

*}*

* **createNode(int data)**: This function dynamically allocates memory for a new node, initializes the node with a data value, and sets both the prior and next pointers to NULL. It returns the new node.

**2. Inserting a Node at the Head:**

*void insertNodeByHead(struct doubleCircularLinkedList\* headNode, int data) {*

*// Create a new node*

*struct doubleCircularLinkedList\* newNode = createNode(data);*

*// Adjust four pointer variables for insertion at the head*

*newNode->prior = headNode; // The new node's prior points to head*

*newNode->next = headNode->next; // The new node's next points to the current first node*

*headNode->next->prior = newNode; // The current first node's prior points to the new node*

*headNode->next = newNode; // Head's next points to the new node*

*}*

* **insertNodeByHead**: This function inserts a new node at the head (right after the head node). The newly created node's pointers are adjusted to fit into the list, and the neighboring nodes' pointers are updated accordingly.

**3. Inserting a Node at the Tail:**

*void insertNodeByTail(struct doubleCircularLinkedList\* headNode, int data) {*

*// Create a new node*

*struct doubleCircularLinkedList\* newNode = createNode(data);*

*// Find the last node in the circular linked list*

*struct doubleCircularLinkedList\* lastNode = headNode;*

*while (lastNode->next != headNode) {*

*lastNode = lastNode->next; // Move to the next node until the last one is found*

*}*

*// Adjust the four pointers for insertion at the tail*

*headNode->prior = newNode; // Head's prior points to the new node*

*newNode->next = headNode; // The new node's next points to head*

*lastNode->next = newNode; // The last node's next points to the new node*

*newNode->prior = lastNode; // The new node's prior points to the last node*

*}*

* **insertNodeByTail**: This function inserts a new node at the tail (before the head node). It first finds the last node in the circular list by traversing from the head node, then adjusts the pointers of the new node, the last node, and the head node.

**Key Points:**

1. **Node Creation**: Memory is allocated dynamically using malloc, and the new node is initialized.
2. **Head Insertion**: A new node is inserted at the front of the list, right after the head node.
3. **Tail Insertion**: A new node is inserted at the end of the list, before the head node, to maintain the circular structure.

This structure ensures that the double circular linked list can efficiently insert nodes both at the head and the tail while maintaining the circular nature of the list.

**Data Structures: Double Circular Linked List - Deletion and Modification Functions**

**1. Deleting a Node at a Specified Location:**

*void SpecifyLocationToDelete(struct doubleCircularLinkedList\* headNode, int posData) {*

*struct doubleCircularLinkedList\* posNode = headNode->next; // Pointer to the specified node*

*struct doubleCircularLinkedList\* posNodePrior = headNode; // Pointer to the previous node*

*// Find the specified node*

*while (posNode->data != posData) {*

*posNodePrior = posNode; // Move to the next node*

*posNode = posNodePrior->next; // Update the posNode pointer*

*// Special handling if the node is not found*

*if (posNode->next == headNode) {*

*printf("指定位置不存在，无法删除！\n"); // "Specified position does not exist, cannot delete!"*

*return; // Exit if the node is not found*

*}*

*}*

*// Adjust pointers to remove the node*

*posNodePrior->next = posNode->next; // Previous node's next points to the node after the specified node*

*posNode->next->prior = posNodePrior; // The node after the specified node points back to the previous node*

*free(posNode); // Free the memory of the deleted node*

*}*

* **SpecifyLocationToDelete**: This function deletes a node with a specific value (posData). It traverses the list to find the node, adjusts the pointers of the surrounding nodes to bypass the node to be deleted, and then frees the memory allocated for that node. If the node is not found, it prints an error message.

**2. Modifying a Specified Element:**

*void modifySpecifiedElement(struct doubleCircularLinkedList\* headNode, int posData, int elem) {*

*struct doubleCircularLinkedList\* posNode = headNode->next; // Pointer to the specified node*

*struct doubleCircularLinkedList\* posNodePrior = headNode; // Pointer to the previous node*

*// Find the specified element*

*while (posNode->data != posData) {*

*posNodePrior = posNode; // Move to the next node*

*posNode = posNodePrior->next; // Update the posNode pointer*

*// Special handling if the node is not found*

*if (posNode->next == headNode) {*

*printf("不存在元素！\n"); // "Element does not exist!"*

*return; // Exit if the element is not found*

*}*

*}*

*// Modify the data of the found node*

*posNode->data = elem; // Set the new value*

*printf("修改成功！\n"); // "Modification successful!"*

*}*

* **modifySpecifiedElement**: This function modifies the data of a node with a specific value (posData). It traverses the list to find the node, and if found, updates its data with the new value (elem). If the node is not found, it prints an error message.

**Key Points:**

1. **Node Deletion**:
   * Traverses the list to find the node with the specified data.
   * Adjusts the pointers of the neighboring nodes to bypass the node being deleted.
   * Frees the memory for the deleted node.
2. **Node Modification**:
   * Similar traversal logic to find the node with the specified data.
   * Updates the node’s data if found, and provides feedback on the modification.

These functions enhance the management of a double circular linked list by allowing for the deletion and modification of nodes based on their values.

**Data Structures: Stack**

**Definition:**

* **Stack**: A stack is a linear data structure that follows the Last In, First Out (LIFO) principle, meaning that the last element added to the stack is the first one to be removed. It is one of the fundamental data structures used in various applications.

**Characteristics:**

* **Insertion and Deletion**: A stack restricts insertion (adding an element) and deletion (removing an element) to a single position, known as the **top** of the stack. The other end, where insertion and deletion are not allowed, is called the **bottom** of the stack.
* **Basic Operations**:
  + **PUSH**: This operation adds an element to the top of the stack.
  + **POP**: This operation removes the element from the top of the stack.
* **LIFO Principle**: In a stack, the most recently added element is the first one to be removed. This can be visualized as stacking boxes; when you put a smaller box into a larger box, you are performing a push operation. When you take a box out, you retrieve the last one added first.

**Visualization:**

The following diagram illustrates a stack:

mathematica

Copy code

Top

----

| | <- Element A (Most Recently Added)

| |

|----|

| | <- Element B

| |

|----|

| | <- Element C (Least Recently Added)

----

Bottom

In this example:

* **Element A** is at the top of the stack and will be the first to be removed when a pop operation is performed.
* **Element C** is at the bottom and will be the last to be removed.

**Summary:**

* A stack is a simple yet powerful data structure used for managing data in a specific order, where the last element added is the first one to be removed.
* It is widely used in programming for tasks such as function calls, expression evaluation, and backtracking algorithms.

**Stack Data Structure Implementation in C**

**Stack Initialization Function**

*void InitStack(PSTACK Stack) {*

*PNODE PNew = (PNODE)malloc(sizeof(NODE)); // Allocate memory for a new node*

*if (PNew == NULL) { // Check if memory allocation was successful*

*printf("Failed to allocate memory for new node!\n");*

*exit(-1);*

*}*

*Stack->PTOP = PNew; // Set the top pointer to the new node*

*Stack->PBOOTOM = PNew; // Set the bottom pointer to the new node*

*PNew->next = NULL; // Initialize the next pointer to NULL*

*printf("Stack created successfully!\n");*

*}*

**Node and Stack Structures**

*// Define a node structure*

*typedef struct Node {*

*ElementType Element; // Element stored in the node*

*struct Node\* next; // Pointer to the next node*

*} NODE, \*PNODE; // Define PNODE as a pointer to NODE*

*// Define the stack structure*

*typedef struct Stack {*

*PNODE PTOP; // Pointer to the top node of the stack*

*PNODE PBOOTOM; // Pointer to the bottom node of the stack*

*} STACK, \*PSTACK; // Define PSTACK as a pointer to STACK*

**Explanation**

* **Initialization Function**: The InitStack function initializes a new stack. It creates a new node and assigns it as both the top and bottom of the stack. The next pointer of the new node is set to NULL, indicating that the stack is empty initially.
* **Node Structure**:
  + The Node structure contains an Element to hold the data and a pointer next to link to the next node in the stack.
* **Stack Structure**:
  + The Stack structure contains pointers to the top and bottom nodes of the stack, allowing easy access to both ends of the stack.

This implementation sets the groundwork for creating and manipulating a stack data structure using linked nodes in C.

**Stack Operations: Push and Pop**

**Push Function**

*void PushStack(PSTACK Stack, int val) {*

*PNODE P = (PNODE)malloc(sizeof(NODE)); // Allocate memory for a new node*

*if (P == NULL) { // Check if memory allocation was successful*

*printf("Failed to allocate memory for new node!\n");*

*exit(-1);*

*}*

*P->Element = val; // Assign the value to the node's element*

*P->next = Stack->PTOP; // Point the new node to the current top node*

*Stack->PTOP = P; // Update the top pointer to the new node*

*}*

**Pop Function**

*void PopStack(PSTACK Stack, int \*val) {*

*if (Stack->PBOOTOM == Stack->PTOP) { // Check if the stack is empty*

*printf("Stack is empty!\n");*

*return;*

*}*

*PNODE P = Stack->PTOP; // Get the current top node*

*\*val = P->Element; // Retrieve the value from the top node*

*Stack->PTOP = P->next; // Update the top pointer to the next node*

*free(P); // Free the memory of the popped node*

*P = NULL; // Set the pointer to NULL to avoid dangling reference*

*printf("%d has been popped from the stack!\n", \*val);*

*}*

**Explanation**

* **Push Function (PushStack)**:
  + Allocates memory for a new node.
  + Checks if memory allocation is successful; if not, it prints an error message and exits the program.
  + Assigns the input value to the new node's Element.
  + Sets the next pointer of the new node to the current top of the stack.
  + Updates the top pointer of the stack to point to the new node.
* **Pop Function (PopStack)**:
  + Checks if the stack is empty by comparing the top pointer with the bottom pointer.
  + If the stack is empty, it prints a message and returns.
  + Retrieves the current top node, gets its value, and updates the top pointer to the next node.
  + Frees the memory allocated for the popped node.
  + Sets the pointer to NULL to avoid a dangling reference.
  + Prints the value that was popped from the stack.

These two functions implement the core stack operations: pushing an element onto the stack and popping an element from the stack.

**Data Structure: Queue**

A queue is a linear storage structure that, like a stack, has strict requirements for how data is "stored" and "retrieved." However, unlike a stack, which has one end that is closed, a queue has two open ends. The data can only enter from one end and exit from the other, as illustrated in the accompanying diagram.

**Key Characteristics:**

* **Queue Ends**:
  + The end where data enters is called the **"rear"** of the queue.
  + The end where data exits is called the **"front"** of the queue.
* **Queue Operations**:
  + The process of adding data to the queue is known as **"enqueue."**
  + The process of removing data from the queue is known as **"dequeue."**
* **FIFO Principle**:
  + The queue follows the **"First In, First Out" (FIFO)** principle, meaning that the first element added to the queue will be the first one to be removed.
  + For example, in a queue with elements 1, 2, and 3, if element 3 is dequeued, elements 1 and 2 must exit before element 3 can be removed.

**Comparison with Stack:**

* **Stack**:
  + Has one closed end (the top), allowing operations to follow the **"Last In, First Out" (LIFO)** principle.
  + Only allows data to be added or removed from the top.
* **Queue**:
  + Has two open ends, allowing data to be added at the rear and removed from the front, following the FIFO principle.

In summary, a queue is a linear storage structure that allows data to enter from one end and exit from the other, adhering to the FIFO principle, while a stack operates on the LIFO principle with one end closed.

**Data Structure: Queue Implementation Using Array**

Here’s an implementation of a queue using an array in C, along with an explanation of its key features.

*#define max 5 // Maximum size of the queue*

*int enQueue(int \*a, int front, int rear, int data) {*

*// Check if the queue is full*

*if ((rear + 1) % max == front) {*

*printf("Queue is full\n");*

*return rear; // Return the current rear if the queue is full*

*}*

*a[rear % max] = data; // Insert data into the queue*

*rear++; // Move rear pointer forward*

*return rear; // Return the updated rear*

*}*

*int deQueue(int \*a, int front, int rear) {*

*// Check if the queue is empty*

*if (front == rear % max) {*

*printf("Queue is empty\n");*

*return front; // Return the current front if the queue is empty*

*}*

*printf("%d\n", a[front]); // Print the data at the front of the queue*

*// Move front pointer forward and wrap around if needed*

*front = (front + 1) % max;*

*return front; // Return the updated front*

*}*

**Key Features:**

1. **Circular Array Implementation**:
   * The queue is implemented using a circular array to efficiently utilize space.
   * When the rear index reaches the end of the array, it wraps around to the beginning of the array using the modulo operator.
2. **Enqueue Operation (enQueue)**:
   * Adds an element to the rear of the queue.
   * Before adding, it checks if the queue is full (i.e., if incrementing rear causes it to equal front).
   * If the queue is not full, it inserts the data at the current rear position and increments rear.
3. **Dequeue Operation (deQueue)**:
   * Removes an element from the front of the queue.
   * Checks if the queue is empty (i.e., if front equals rear).
   * If not empty, it prints the element at the front and increments front to point to the next element.

**Thought Experiment: Linked List Implementation of Queue**

A queue can also be implemented using a linked list. This method avoids the fixed size limitation of arrays:

* **Node Structure**:
  + Each node contains data and a pointer to the next node.
* **Operations**:
  + **Enqueue**: Create a new node, link it to the current rear, and update the rear pointer.
  + **Dequeue**: Remove the front node, update the front pointer to the next node, and free the memory of the removed node.

Using a linked list allows for dynamic sizing, enabling the queue to grow and shrink as needed without wasting space.

This covers the implementation of a queue using an array, along with considerations for implementing a queue using a linked list.

**Data Structure: Array Sorting - Selection Sort**

**Selection Sort** is a straightforward and intuitive sorting algorithm. Its working principle involves repeatedly selecting the smallest (or largest) element from the unsorted portion of the array and moving it to the sorted portion.

**How Selection Sort Works:**

1. **Initialization**: Start with an unsorted array.
2. **Find the Minimum**: In each iteration, find the minimum element from the unsorted portion of the array.
3. **Swap**: Swap the found minimum element with the first element of the unsorted portion.
4. **Repeat**: Move the boundary between the sorted and unsorted portions one element forward and repeat the process until all elements are sorted.

**Selection Sort Implementation in C**

*void selectionSort(int arr[], int n) {*

*int len = n; // Length of the array*

*int minIndex, temp;*

*for (int i = 0; i < len - 1; i++) {*

*minIndex = i; // Assume the minimum is the first element of the unsorted array*

*for (int j = i + 1; j < len; j++) {*

*// Find the smallest element in the unsorted portion*

*if (arr[j] < arr[minIndex]) {*

*minIndex = j; // Update minIndex if a smaller element is found*

*}*

*}*

*// Swap the found minimum element with the first element of the unsorted array*

*temp = arr[i];*

*arr[i] = arr[minIndex];*

*arr[minIndex] = temp;*

*}*

*}*

**Characteristics of Selection Sort:**

* **Time Complexity**: The average and worst-case time complexity is O(n2)O(n^2)O(n2), making it inefficient for large datasets.
* **Space Complexity**: Selection Sort is an **in-place** sorting algorithm, which means it requires a constant amount of additional space O(1)O(1)O(1).
* **Stability**: It is not a stable sort, meaning that it may change the relative order of equal elements.
* **Use Cases**: Due to its simplicity and O(n2)O(n^2)O(n2) time complexity, Selection Sort is generally only efficient for small datasets.

**Conclusion**

Selection Sort is one of the most basic sorting algorithms. It is often one of the first sorting methods learned due to its simplicity. However, its inefficiency makes it less suitable for larger arrays, where more advanced algorithms like Quick Sort or Merge Sort are typically preferred. Its only significant advantage is that it does not require additional memory, making it useful in memory-constrained situations.

**Data Structure: Array Sorting - Bubble Sort**

**Bubble Sort** is a simple sorting algorithm that repeatedly traverses the array to be sorted. It compares adjacent elements and swaps them if they are in the wrong order. This process is repeated until no more swaps are needed, indicating that the array is sorted. The algorithm is named "Bubble Sort" because smaller elements "bubble" to the top of the array through successive swaps.

**How Bubble Sort Works:**

1. **Initialization**: Start with an unsorted array.
2. **Traversal**: Compare each pair of adjacent elements in the array.
3. **Swap**: If the first element is greater than the second, swap them.
4. **Repeat**: Continue this process for each element in the array, reducing the effective size of the unsorted portion with each pass.
5. **Completion**: The algorithm finishes when a complete pass is made without any swaps, indicating that the array is sorted.

**Bubble Sort Implementation in C**

*void bubbleSort(int arr[], int n) {*

*int len = n; // Length of the array*

*// Outer loop for each element*

*for (int i = 0; i < len - 1; i++) {*

*// Inner loop to compare adjacent elements*

*for (int j = 0; j < len - 1 - i; j++) {*

*// Compare adjacent elements and swap if necessary*

*if (arr[j] > arr[j + 1]) {*

*int temp = arr[j + 1]; // Store the next element*

*arr[j + 1] = arr[j]; // Swap elements*

*arr[j] = temp; // Update the current element*

*}*

*}*

*}*

*}*

**Characteristics of Bubble Sort:**

* **Time Complexity**: The average and worst-case time complexity is O(n2)O(n^2)O(n2), which makes it inefficient for large datasets.
* **Space Complexity**: Bubble Sort is an **in-place** sorting algorithm, requiring only a constant amount of additional space O(1)O(1)O(1).
* **Stability**: Bubble Sort is a stable sorting algorithm, meaning that it preserves the relative order of equal elements.
* **Best Case**: The best-case time complexity is O(n)O(n)O(n) when the array is already sorted.

**Conclusion**

Bubble Sort is one of the simplest sorting algorithms and is often used for educational purposes to illustrate sorting concepts. However, its inefficiency in handling larger datasets limits its practical applications. For larger arrays, more efficient algorithms such as Quick Sort, Merge Sort, or Heap Sort are generally preferred. Despite its limitations, Bubble Sort's stability and simplicity make it useful in certain situations, particularly when working with small or mostly sorted datasets.

**Data Structure: Array Sorting - Insertion Sort**

**Insertion Sort** is a straightforward and intuitive sorting algorithm. It builds a sorted sequence one element at a time by repeatedly taking an unsorted element and inserting it into its correct position within the already sorted part of the array.

**How Insertion Sort Works:**

1. **Initialization**: Start with the first element as a sorted array.
2. **Iteration**: For each subsequent element:
   * Compare it with the elements in the sorted array from right to left.
   * Shift larger elements one position to the right to create space for the new element.
   * Insert the current element into its correct position.
3. **Completion**: Repeat until all elements are sorted.

**Insertion Sort Implementation in C**

*void insertionSort(int arr[], int n) {*

*int len = n; // Length of the array*

*int preIndex, current;*

*// Start from the second element as the first is considered sorted*

*for (int i = 1; i < len; i++) {*

*preIndex = i - 1; // Index of the last sorted element*

*current = arr[i]; // Element to be inserted*

*// Shift elements of the sorted segment that are greater than current*

*while (preIndex >= 0 && arr[preIndex] > current) {*

*arr[preIndex + 1] = arr[preIndex]; // Move larger elements to the right*

*preIndex--;*

*}*

*// Insert the current element into its correct position*

*arr[preIndex + 1] = current;*

*}*

*}*

**Characteristics of Insertion Sort:**

* **Time Complexity**:
  + **Best Case**: O(n)O(n)O(n) when the array is already sorted.
  + **Average and Worst Case**: O(n2)O(n^2)O(n2), particularly when the array is sorted in reverse order.
* **Space Complexity**: It is an **in-place** sorting algorithm, requiring only O(1)O(1)O(1) additional space.
* **Stability**: Insertion Sort is stable, meaning that it preserves the relative order of equal elements.
* **Adaptive**: It performs better on partially sorted arrays, making it useful in practical applications.

**Algorithm Analysis**

Insertion Sort works well for small datasets or nearly sorted datasets because it efficiently reduces the number of swaps needed. The in-place characteristic allows it to sort with minimal memory overhead. Although its time complexity of O(n2)O(n^2)O(n2) makes it unsuitable for large datasets, its simplicity and effectiveness in specific scenarios contribute to its use in practice.

**Conclusion**

Insertion Sort is an excellent introductory algorithm for understanding sorting mechanics. It demonstrates fundamental concepts such as element comparison and movement. While not the most efficient for large datasets, it is particularly useful for smaller or partially sorted arrays. For larger datasets, consider using more efficient sorting algorithms like Quick Sort or Merge Sort.

**Data Structure: Array Sorting - Quick Sort**

**Quick Sort** is an efficient sorting algorithm that improves upon the basic sorting methods like Bubble Sort. It employs a divide-and-conquer strategy to sort elements quickly by partitioning the array.

**How Quick Sort Works:**

1. **Choosing a Pivot**: Select a pivot element from the array. This element will help divide the array into two parts.
2. **Partitioning**: Rearrange the elements in the array so that:
   * All elements less than the pivot are on its left.
   * All elements greater than the pivot are on its right.
3. **Recursive Sorting**: Recursively apply the same process to the sub-arrays formed by partitioning.
4. **Completion**: The process repeats until all sub-arrays are sorted.

**Quick Sort Implementation in C**

*// Function to partition the array*

*int partition(int arr[], int low, int high) {*

*int key = arr[low]; // Choose the first element as the pivot*

*while (low < high) {*

*// Move high pointer left until finding an element smaller than pivot*

*while (low < high && arr[high] >= key) high--;*

*if (low < high) arr[low++] = arr[high]; // Place smaller element on the left*

*// Move low pointer right until finding an element larger than pivot*

*while (low < high && arr[low] <= key) low++;*

*if (low < high) arr[high--] = arr[low]; // Place larger element on the right*

*}*

*arr[low] = key; // Place the pivot in its correct position*

*return low; // Return the pivot index*

*}*

*/\*\*\* Quick Sort Function*

*\* @param arr[] Array to be sorted*

*\* @param start Starting index of the array*

*\* @param end Last index of the array*

*\*\*\*/*

*void quick\_sort(int arr[], int start, int end) {*

*int pos;*

*if (start < end) {*

*pos = partition(arr, start, end); // Partitioning the array*

*quick\_sort(arr, start, pos - 1); // Recursively sort the left sub-array*

*quick\_sort(arr, pos + 1, end); // Recursively sort the right sub-array*

*}*

*return;*

*}*

**Characteristics of Quick Sort:**

* **Time Complexity**:
  + **Best and Average Case**: O(nlog⁡n)O(n \log n)O(nlogn), where nnn is the number of elements.
  + **Worst Case**: O(n2)O(n^2)O(n2), occurs when the smallest or largest element is always chosen as the pivot (e.g., already sorted array).
* **Space Complexity**: O(log⁡n)O(\log n)O(logn) for the recursive stack in the average case. It may go up to O(n)O(n)O(n) in the worst case.
* **In-Place**: Quick Sort does not require extra space for another array; it sorts the elements in place.
* **Not Stable**: The algorithm may change the relative order of equal elements, making it unstable.

**Algorithm Analysis**

Quick Sort is highly efficient for large datasets and is one of the fastest sorting algorithms in practice. Its average time complexity of O(nlog⁡n)O(n \log n)O(nlogn) makes it suitable for most sorting tasks. The choice of the pivot can significantly impact the algorithm's efficiency, which is why techniques like choosing a random pivot or the median-of-three method are often used to improve performance.

**Conclusion**

Quick Sort is an essential algorithm in computer science due to its efficiency and versatility. It serves as the basis for many advanced sorting techniques and is widely used in various applications, from database systems to sorting libraries. Understanding its implementation and behavior helps in making informed choices when dealing with sorting tasks in programming.

**Custom Memory Management Implementation**

In the process of implementing memory management, we can create custom malloc and free functions to simulate dynamic memory allocation and deallocation. This approach typically involves maintaining a free list to efficiently manage allocated and available memory blocks.

Below is an example of a simple memory management implementation using a linked list:

**Custom Memory Management Structure**

*#include <stdio.h>*

*#include <stdlib.h>*

*#define MEMORY\_POOL\_SIZE 1024 // Define the size of the memory pool*

*typedef struct Block {*

*size\_t size; // Size of the block*

*struct Block\* next; // Pointer to the next block*

*int free; // Flag indicating whether the block is free*

*} Block;*

*typedef struct MemoryPool {*

*char pool[MEMORY\_POOL\_SIZE]; // Memory pool*

*Block\* freeList; // Free block linked list*

*} MemoryPool;*

*MemoryPool memPool;*

*// Initialize the memory pool*

*void initMemoryPool() {*

*memPool.freeList = (Block\*)memPool.pool; // Set the head pointer of the free block list to point to the memory pool*

*memPool.freeList->size = MEMORY\_POOL\_SIZE - sizeof(Block); // Initialize the size of the free block*

*memPool.freeList->next = NULL; // No other free blocks currently*

*memPool.freeList->free = 1; // Mark as a free block*

*}*

*// Custom malloc function*

*void\* myMalloc(size\_t size) {*

*Block\* current = memPool.freeList; // Start from the head of the free block list*

*while (current != NULL) {*

*// Check if the block is large enough and is free*

*if (current->free && current->size >= size) {*

*// Split the block*

*if (current->size >= size + sizeof(Block) + 1) { // Check if there's enough space to split*

*Block\* newBlock = (Block\*)((char\*)current + sizeof(Block) + size);*

*newBlock->size = current->size - size - sizeof(Block);*

*newBlock->next = current->next;*

*newBlock->free = 1;*

*current->size = size; // Update the size of the current block*

*current->next = newBlock; // Update the linked list*

*}*

*current->free = 0; // Mark as allocated*

*return (char\*)current + sizeof(Block); // Return the pointer to the data area*

*}*

*current = current->next; // Move to the next block*

*}*

*return NULL; // No available blocks*

*}*

*// Custom free function*

*void myFree(void\* ptr) {*

*if (ptr == NULL) return; // Check for null pointer*

*Block\* block = (Block\*)((char\*)ptr - sizeof(Block)); // Get the block header*

*block->free = 1; // Mark as free block*

*// Merge free blocks*

*Block\* current = memPool.freeList;*

*while (current != NULL) {*

*if (current->free && current->next && current->next->free) {*

*current->size += sizeof(Block) + current->next->size; // Merge size*

*current->next = current->next->next; // Skip the merged block*

*} else {*

*current = current->next; // Move to the next block*

*}*

*}*

*}*

*// Test custom memory management*

*int main() {*

*initMemoryPool(); // Initialize the memory pool*

*// Test memory allocation*

*int\* arr = (int\*)myMalloc(10 \* sizeof(int));*

*if (arr == NULL) {*

*printf("Memory allocation failed!\n");*

*} else {*

*for (int i = 0; i < 10; i++) {*

*arr[i] = i; // Initialize the array*

*printf("%d ", arr[i]);*

*}*

*printf("\n");*

*}*

*// Free memory*

*myFree(arr);*

*return 0;*

*}*

**Code Explanation**

1. **Structure Definitions**:
   * Block: A structure used to manage memory blocks, including the block's size, a pointer to the next block, and a flag indicating whether it is free.
   * MemoryPool: Contains the memory pool and a pointer to the free block linked list.
2. **Memory Pool Initialization**:
   * initMemoryPool(): Initializes the memory pool by marking the first block as available and setting its size to the size of the entire memory pool.
3. **Custom Memory Allocation**:
   * myMalloc(size\_t size): Searches for available blocks in the free block list and allocates memory. If a suitable block is found, it is split for use.
4. **Custom Memory Deallocation**:
   * myFree(void\* ptr): Frees the allocated memory and merges adjacent free blocks.
5. **Testing Custom Memory Management**:
   * The main function initializes the memory pool, tests memory allocation, and then frees the allocated memory.

**Assignment**

**7.1** Implement the preorder traversal of a binary tree using a non-recursive approach.

**7.2** Use a stack to convert a decimal number to binary and display it using putchar().

**Decimal to Other Base Conversion Principle**

The conversion principle is given by: N=(Nd)×d+Nmod  dN = \left( \frac{N}{d} \right) \times d + N \mod dN=(dN​)×d+Nmodd For example: (1348)10=(2504)8(1348)\_{10} = (2504)\_{8}(1348)10​=(2504)8​

**7.1 Non-Recursive Preorder Traversal of a Binary Tree**

In this implementation, we will use a stack to perform a non-recursive preorder traversal of a binary tree.

*#include <stdio.h>*

*#include <stdlib.h>*

*typedef struct TreeNode {*

*int value;*

*struct TreeNode\* left;*

*struct TreeNode\* right;*

*} TreeNode;*

*typedef struct StackNode {*

*TreeNode\* treeNode;*

*struct StackNode\* next;*

*} StackNode;*

*void push(StackNode\*\* top, TreeNode\* node) {*

*StackNode\* newNode = (StackNode\*)malloc(sizeof(StackNode));*

*newNode->treeNode = node;*

*newNode->next = \*top;*

*\*top = newNode;*

*}*

*TreeNode\* pop(StackNode\*\* top) {*

*if (\*top == NULL) return NULL;*

*StackNode\* temp = \*top;*

*\*top = (\*top)->next;*

*TreeNode\* poppedNode = temp->treeNode;*

*free(temp);*

*return poppedNode;*

*}*

*int isEmpty(StackNode\* top) {*

*return top == NULL;*

*}*

*void preorderTraversal(TreeNode\* root) {*

*if (root == NULL) return;*

*StackNode\* stack = NULL; // Initialize stack*

*push(&stack, root); // Push the root node to stack*

*while (!isEmpty(stack)) {*

*TreeNode\* currentNode = pop(&stack); // Pop a node from the stack*

*printf("%d ", currentNode->value); // Process the current node*

*// Push right child first so that left child is processed first*

*if (currentNode->right != NULL) {*

*push(&stack, currentNode->right);*

*}*

*if (currentNode->left != NULL) {*

*push(&stack, currentNode->left);*

*}*

*}*

*}*

*// Function to create a new tree node*

*TreeNode\* createNode(int value) {*

*TreeNode\* newNode = (TreeNode\*)malloc(sizeof(TreeNode));*

*newNode->value = value;*

*newNode->left = newNode->right = NULL;*

*return newNode;*

*}*

*// Example usage*

*int main() {*

*TreeNode\* root = createNode(1);*

*root->left = createNode(2);*

*root->right = createNode(3);*

*root->left->left = createNode(4);*

*root->left->right = createNode(5);*

*printf("Preorder Traversal: ");*

*preorderTraversal(root); // Output: 1 2 4 5 3*

*return 0;*

*}*

**7.2 Decimal to Binary Conversion Using Stack**

In this implementation, we will convert a decimal number to binary using a stack and display the binary number using putchar().

*#include <stdio.h>*

*#include <stdlib.h>*

*typedef struct StackNode {*

*int data;*

*struct StackNode\* next;*

*} StackNode;*

*void push(StackNode\*\* top, int value) {*

*StackNode\* newNode = (StackNode\*)malloc(sizeof(StackNode));*

*newNode->data = value;*

*newNode->next = \*top;*

*\*top = newNode;*

*}*

*int pop(StackNode\*\* top) {*

*if (\*top == NULL) return -1; // Stack is empty*

*StackNode\* temp = \*top;*

*int poppedValue = temp->data;*

*\*top = (\*top)->next;*

*free(temp);*

*return poppedValue;*

*}*

*int isEmpty(StackNode\* top) {*

*return top == NULL;*

*}*

*void decimalToBinary(int n) {*

*StackNode\* stack = NULL; // Initialize stack*

*while (n > 0) {*

*int remainder = n % 2; // Get remainder*

*push(&stack, remainder); // Push remainder onto the stack*

*n = n / 2; // Divide n by 2*

*}*

*// Display the binary number using putchar*

*while (!isEmpty(stack)) {*

*int binaryDigit = pop(&stack);*

*putchar(binaryDigit + '0'); // Convert to character and print*

*}*

*putchar('\n'); // Print newline*

*}*

*// Example usage*

*int main() {*

*int decimalNumber = 1348;*

*printf("Decimal: %d, Binary: ", decimalNumber);*

*decimalToBinary(decimalNumber); // Output: 10101011100*

*return 0;*

*}*

**Explanation**

1. **Non-Recursive Preorder Traversal**:
   * A stack is used to traverse the binary tree without recursion.
   * The root node is pushed onto the stack, and then nodes are processed by popping from the stack.
   * The right child is pushed onto the stack first to ensure the left child is processed first.
2. **Decimal to Binary Conversion**:
   * The decimal number is repeatedly divided by 2, and the remainders are pushed onto a stack.
   * Once the division process is complete, the binary digits are popped from the stack and displayed using *putchar().*