Contents

[List [Pointer-based] Recursive 2](#_Toc184742959)

[1. Add [specific index] - Has helper function 2](#_Toc184742960)

[2. Remove [specific index] – Has helper function 2](#_Toc184742961)

[3. Count - has helper function 3](#_Toc184742962)

[4. Reverse 4](#_Toc184742963)

[5. Search 4](#_Toc184742964)

[6. Sum 5](#_Toc184742965)

[7. Display 5](#_Toc184742966)

[8. findLargest 5](#_Toc184742967)

[9. SumOfEven 6](#_Toc184742968)

[List [Array] Recursive 7](#_Toc184742969)

[1. Add [Specific Index] 7](#_Toc184742970)

[2. Remove [Specific Index] 7](#_Toc184742971)

[3. Count (returns number of occurrence) 8](#_Toc184742972)

[4. Reverse √ 8](#_Toc184742973)

[5. Sum 9](#_Toc184742974)

[6. SumOfEven 9](#_Toc184742975)

[7. findMax 10](#_Toc184742976)

[8. sort (in ascending order) 10](#_Toc184742977)

[Stack Recursive Display 12](#_Toc184742978)

[1. Reverse 12](#_Toc184742979)

[2. Display √ 12](#_Toc184742980)

[3. Display Reversed √ 13](#_Toc184742981)

[Stack Recursive Functions [Pointer-based] 14](#_Toc184742982)

[1. Count √ 14](#_Toc184742983)

[Queue Recursive Functions [Pointer-based] 15](#_Toc184742984)

[1. Count √ 15](#_Toc184742985)

[2. Reverse √ 15](#_Toc184742986)

# 

# List [Pointer-based] Recursive

## Add [specific index] - Has helper function

bool List::addR(int index, ItemType item) {

    // Check if the index is within bounds

    if (index < size) {

        // Start the recursive addition process

        addRHelper(index, item, firstNode, 0);

        return true;

    }

    return false; // Return false if the index is out of bounds

}

Node\* List::addRHelper(int index, ItemType item, Node\* currentNode, int currentIndex) {

    // Base case: When the current index matches the target index

    if (currentIndex == index) {

        // Create a new node and insert it into the list

        Node\* newNode = new Node;

        newNode->item = currentNode->item; // Copy the item of the current node

        newNode->next = currentNode->next; // Set the new node's next pointer

        currentNode->next = newNode;       // Link the current node to the new node

        currentNode->item = item;          // Update the current node's item

        size++;                            // Increment the size of the list

        return nullptr;

    }

    // Recursive case: Traverse the list to the next node

    addRHelper(index, item, currentNode->next, currentIndex + 1);

    return nullptr;

}

## Remove [specific index] – Has helper function

bool List::removeR(int index) {

    // Check if the index is within bounds

    if (index < size) {

        removeRHelper(index, nullptr, firstNode, 0);

        return true;

    }

    return false; // Return false if the index is out of bounds

}

Node\* List::removeRHelper(int index, Node\* previousNode, Node\* currentNode, int currentIndex) {

    // Base case: When the target index is found

    if (currentIndex == index) {

        if (previousNode == nullptr) {

            // Special case: Removing the first node

            Node\* tempNode = firstNode;

            firstNode = firstNode->next; // Update the head of the list

            delete tempNode;             // Delete the first node

        } else {

            // General case: Removing a node in the middle or end

            previousNode->next = currentNode->next; // Bypass the current node

            delete currentNode;                     // Delete the current node

        }

        size--; // Decrement the size of the list

        return nullptr;

    }

    // Recursive case: Traverse to the next node

    removeRHelper(index, currentNode, currentNode->next, currentIndex + 1);

    return nullptr;

}

## Count - has helper function

// Count the number of occurrences of an item in the list, using recursion

int List::countR(ItemType item) {

    // Call the helper function to count occurrences of the item

    return countRHelper(front, item);

}

int List::countRHelper(Node\* currentNode, ItemType item) {

    // Base case: If the current node is null, return 0

    if (currentNode == nullptr) {

        return 0;

    }

    // Recursive case: Check the current node's item and proceed to the next node

    int count = (currentNode->item == item) ? 1 : 0;

    return count + countRHelper(currentNode->next, item);

}

## Reverse

void List::reverseR() {

    // Start the recursive reversal process

    reverseRHelper(front);

}

List::Node\* List::reverseRHelper(Node\* node) {

    // Base case: If the list is empty or only one node remains

    if (node == nullptr) {

        return nullptr; // Return for an empty list

    }

    if (node->next == nullptr) {

        front = node; // Update the head of the list

        return node;  // Return the last node, which will become the first

    }

    // Recursive step: Reverse the rest of the list

    List::Node\* lastNode = reverseRHelper(node->next);

    // Adjust the pointers for the current node

    lastNode->next = node; // Set the next of the last node in reversed sublist

    node->next = nullptr;  // Current node becomes the tail in reversed sublist

    return node; // Return the current node

}

## Search

bool search(struct Node\* front, int x) {

    // Base case: If the list is empty, return false

    if (front == nullptr) {

        return false;

    }

    // Check if the current node contains the target value

    if (front->item == x) {

        return true; // Key found

    }

    // Recur for the remaining nodes

    return search(front->next, x);

}

## Sum

int List::sumList(Node\* head) {

    // Base case: If the list is empty or we reach the end, return 0

    if (head == nullptr) {

        return 0;

    }

    // Recursive case: Sum the current node's item with the sum of the rest of the list

    return head->item + sumList(head->next);

}

## Display

void List::displayRecursively(Node\* current) {

    // Base case: If the current node is null, stop the recursion

    if (current == nullptr) {

        return;

    }

    // Print the current node's data

    cout << current->data << endl;

    // Recur for the next node

    displayRecursively(current->next);

}

## findLargest

int List::findLargest(Node\* current) {

    // Base case: If this is the last node, return its value

    if (current->next == nullptr) {

        return current->item;

    }

    // Recursive call to find the largest value in the remaining list

    int largestInRest = findLargest(current->next);

    // Return the larger value between the current node's item and the largest in the rest

    return (current->item > largestInRest) ? current->item : largestInRest;

}

Feasible Question: Recursion is not the best solution to solve this problem. Solution: Non-recursive code would be easier to read (recursion), faster (overhead of function call), and more memory efficient (obviously more stack frames).

## SumOfEven

int Sum(Node\* head) {

    // Base case: If the list is empty, return 0

    if (head == nullptr) {

        return 0;

    }

    // Recursive case: Check if the current node's data is even

    if (head->data % 2 == 0) {

        // Add the current node's value if it is even

        return head->data + Sum(head->next);

    } else {

        // Skip the current node's value if it is odd

        return Sum(head->next);

    }

}

## Sort

Sort in ascending order using recursion

// Sort the list in ascending order using recursion

// Split the list into two halves

List::Node\* List::split(Node\* head) {

    // Slow and fast pointers to find the middle of the list

    Node\* slow = head;

    Node\* fast = head;

    Node\* prev = nullptr;

    // Use slow-fast pointer technique to find the middle of the list

    while (fast != nullptr && fast->next != nullptr) {

        prev = slow;

        slow = slow->next;

        fast = fast->next->next;

    }

    // Split the list into two halves

    if (prev != nullptr) {

        prev->next = nullptr;

    }

    return slow;

}

// Merge two sorted lists into a single sorted list

List::Node\* List::merge(Node\* left, Node\* right) {

    // Base cases : If either list is empty, return the other list

    if (left == nullptr) {

        return right;

    }

    if (right == nullptr) {

        return left;

    }

    // Compare the data and recursively merge the lists

    if (left->item <= right->item) {

        left->next = merge(left->next, right);

        return left;

    } else {

        right->next = merge(left, right->next);

        return right;

    }

}

// Sort the linked list using Merge Sort

/\*

Time Complexity: O(n log n) due to the divide and conquer approach

Space Complexity: O(log n) due to recursion depth

\*/

List::Node\* List::sort(Node\* head) {

    // Base case : If the list is empty or has only one node, its already sorted

    if (head == nullptr || head->next == nullptr) {

        return head;

    }

    // Split the list into two halves

    Node\* mid = split(head);

    // Recursively sort the two halves

    Node\* left = sort(head);

    Node\* right = sort(mid);

    // Merge the sorted halves

    return merge(left, right);

}

# List [Array] Recursive

## Add [Specific Index]

bool List::addR(int index, ItemType item) {

    // Check if the index is within bounds

    if (index > size) {

        return false; // Index out of bounds

    }

    // Base case: If the index equals the size, add the item at the end

    if (index == size) {

        items[size] = item; // Add the new item

        size++;             // Increment the size

        return true;

    }

    // Recursive case: Shift items recursively to the right

    addR(index + 1, items[index]);

    items[index] = item; // Insert the new item at the correct position

    return true;

}

## Remove [Specific Index]

void List::removeR(int index) {

    // Ensure the index is within bounds

    if (index >= size) {

        return; // Index out of bounds, nothing to remove

    }

    // Base case: If this is the last element, simply reduce the size

    if (index == size - 1) {

        size--; // Decrement the size of the list

        return;

    }

    // Recursive case: Shift the current element to the left

    items[index] = items[index + 1];

    // Recur for the next index

    removeR(index + 1);

}

## Count (returns number of occurrence)

int List::count(ItemType item) {

    // Start the recursive count from index 0

    return countR(0, item);

}

int List::countR(int index, ItemType item) {

    // Base case: If the index is out of bounds, return 0

    if (index >= size) {

        return 0;

    }

    // Recursive case: Check if the current item matches the target

    return (items[index] == item ? 1 : 0) + countR(index + 1, item);

}

## Reverse √

void List::displayReverseOrder(string array[], int n) {

    // Base case: If the array has only one element, print it

    if (n == 1) {

        cout << array[n - 1] << endl;

        return; // End the recursion

    }

    // Recursive case: Print the current element and recur for the rest

    cout << array[n - 1] << endl;

    displayReverseOrder(array, n - 1);

}

## Sum

int findSum(int array[], int N) {

    // Base case: If there are no elements, the sum is 0

    if (N <= 0) {

        return 0;

    }

    // Recursive case: Add the last element to the sum of the remaining array

    return findSum(array, N - 1) + array[N - 1];

}

## SumOfEven

int List::addEven(int n) {

    // Base case: If we've reached the end of the list, stop recursion

    if (n >= size) {

        return 0;

    }

    // Recursive case: Add the current item if it's even, otherwise skip it

    if (items[n] % 2 == 0) {

        return items[n] + addEven(n + 1);

    } else {

        return addEven(n + 1);

    }

}

## findMax

int findMax(int array[], int n) {

    // Base case: If the array has only one element, return it

    if (n == 1) {

        return array[0];

    }

    // Recursive case: Find the maximum in the rest of the array

    int maxInRest = findMax(array, n - 1);

    // Compare the current element with the maximum of the rest

    return (array[n - 1] > maxInRest) ? array[n - 1] : maxInRest;

}

## sort (in ascending order)

void merge(int array[], int left, int mid, int right) {

    int n1 = mid - left + 1;  // Size of the left subarray

    int n2 = right - mid;     // Size of the right subarray

    // Create temporary subarrays

    int leftArray[n1], rightArray[n2];

    // Copy data to temporary subarrays

    for (int i = 0; i < n1; i++) {

        leftArray[i] = array[left + i];

    }

    for (int i = 0; i < n2; i++) {

        rightArray[i] = array[mid + 1 + i];

    }

    // Merge the temporary subarrays back into the original array

    int i = 0;      // Initial index of left subarray

    int j = 0;      // Initial index of right subarray

    int k = left;   // Initial index of merged subarray

    while (i < n1 && j < n2) {

        if (leftArray[i] <= rightArray[j]) {

            array[k] = leftArray[i];

            i++;

        } else {

            array[k] = rightArray[j];

            j++;

        }

        k++;

    }

    // Copy any remaining elements of the left subarray

    while (i < n1) {

        array[k] = leftArray[i];

        i++;

        k++;

    }

    // Copy any remaining elements of the right subarray

    while (j < n2) {

        array[k] = rightArray[j];

        j++;

        k++;

    }

}

void mergeSort(int array[], int left, int right) {

    // Base case: If the subarray has only one element, it's already sorted

    if (left >= right) {

        return;

    }

    // Find the middle point

    int mid = left + (right - left) / 2;

    // Recursively sort the left and right halves

    mergeSort(array, left, mid);

    mergeSort(array, mid + 1, right);

    // Merge the sorted halves

    merge(array, left, mid, right);

}

# Stack Recursive Display (Array Stuff)

## Reverse

void Stack::reverser() {

    if (isEmpty()) {

        return; // Base case: If the stack is empty, stop recursion

    }

    // Step 1: Remove the top element of the stack

    int topElement = top();

    pop();

    // Step 2: Recursively reverse the rest of the stack

    reverser();

    // Step 3: Insert the removed element at the bottom of the stack

    insertAtBottom(topElement);

}

void Stack::insertAtBottom(int item) {

    if (isEmpty()) {

        // Base case: If the stack is empty, push the item

        push(item);

        return;

    }

    // Step 1: Remove the top element

    int topElement = top();

    pop();

    // Step 2: Recursively insert the item at the bottom

    insertAtBottom(item);

    // Step 3: Push the removed element back on top

    push(topElement);

}

## Display

void Stack::PrintStack(Stack s) {

    // Base case: If the stack is empty, stop recursion

    if (s.empty()) {

        return;

    }

    // Step 1: Get the top element of the stack

    int x = s.top();

    // Step 2: Remove the top element from the stack

    s.pop();

    // Step 3: Recursively print the rest of the stack

    PrintStack(s);

    // Step 4: Print the current element

    cout << x << " ";

    // Note: Since `s` is passed by value, this does not modify the original stack

}

## Display Reversed

// Recursive function to display a stack in revrese order

void Stack::displayInOrderOfInsertionR() {

    // Create a temporary stack to assist in reversing the display

    Stack temp;

    // If the stack is not empty, display its contents in insertion order

    if (!isEmpty()) {

        cout << displayInOrderOfInsertionR2(temp) << endl;

        // Restore the original stack order

        displayInOrderOfInsertionR3(temp);

    }

}

string Stack::displayInOrderOfInsertionR2(Stack &tempStack) {

    // Base case: If the stack is empty, return an empty string

    if (isEmpty()) {

        return "";

    }

    // Recursive step: Pop the top element, save it in the temp stack, and process the rest

    ItemType temp;

    pop(temp);              // Remove the top element from the current stack

    tempStack.push(temp);   // Save it in the temporary stack

    // Recur for the remaining stack and append the current element to the result string

    return displayInOrderOfInsertionR2(tempStack) + " " + to\_string(temp);

}

void Stack::displayInOrderOfInsertionR3(Stack &tempStack) {

    // Base case: If the temporary stack is empty, stop recursion

    if (tempStack.isEmpty()) {

        return;

    }

    // Recursive step: Pop from the temporary stack and push it back into the original stack

    ItemType temp;

    tempStack.pop(temp);    // Remove the top element from the temporary stack

    push(temp);             // Restore it to the original stack

    // Recur for the rest of the temporary stack

    displayInOrderOfInsertionR3(tempStack);

}

## Sort stack

void Stack::sortStack() {

    // Base case: If the stack is empty, return

    if (isEmpty()) {

        return;

    }

    // Step 1: Remove the top element

    int topElement = top();

    pop();

    // Step 2: Recursively sort the remaining stack

    sortStack();

    // Step 3: Insert the removed element back into the sorted stack

    insertSorted(topElement);

}

void Stack::insertSorted(int item) {

    // Base case: If the stack is empty or the item is greater than the top element

    if (isEmpty() || item > top()) {

        push(item);

        return;

    }

    // Step 1: Remove the top element

    int topElement = top();

    pop();

    // Step 2: Recursively insert the item into the sorted stack

    insertSorted(item);

    // Step 3: Push the top element back onto the stack

    push(topElement);

# Stack Recursive Functions [Pointer-based]

## Count

int Stack::count() {

    Node\* tmp = topNode;

    return count2(0, tmp);

}

int Stack::count2(int index, Node\* tmp) {

    // Base case: If the current node is NULL, return the index

    if (tmp == nullptr) {

        return index;

    }

    // Recursive case: Increment the index and move to the next node

    return count2(index + 1, tmp->next);

}

# Queue Recursive Functions [Pointer-based]

## Count

    int Queue::count() {

        return countHelper(front);

    }

    int Queue::countHelper(Node\* node) {

        if (node == nullptr) {

            return 0;

        }

        return 1 + countHelper(node->next);

    }

## Reverse

    // Enqueue an element to the rear of the queue

    void enqueue(int value) {

        Node\* newNode = new Node(value);

        if (rear) {

            rear->next = newNode;

        }

        rear = newNode;

        if (!front) {

            front = rear;

        }

    }

    // Dequeue an element from the front of the queue

    int dequeue() {

        if (isEmpty()) {

            throw runtime\_error("Queue is empty");

        }

        Node\* temp = front;

        int value = temp->data;

        front = front->next;

        if (!front) {

            rear = nullptr;

        }

        delete temp;

        return value;

    }

    // Recursive function to reverse the queue

    void reverseQueue() {

        // Base case: If the queue is empty, return

        if (isEmpty()) {

            return;

        }

        // Step 1: Dequeue the front element

        int frontElement = dequeue();

        // Step 2: Recursively reverse the remaining queue

        reverseQueue();

        // Step 3: Enqueue the dequeued element to the rear

        enqueue(frontElement);

    }

## Display

    // Recursive function to display queue

    /\*

    Time Complexity: O(n) - Each node is visited exactly once.

    Space Complexity: O(n) - Recursive call stack for n nodes.

    \*/

    void Queue::display()

    {

        displayHelper(front);

    }

    void Queue::displayHelper(Node \*node)

    {

        if (node == nullptr)

        {

            return;

        }

        cout << node->data << " ";

        displayHelper(node->next);

    }

## Sum

    // Recursive function to calculate sum of elements in a queue

    /\*

Time Complexity: O(n) - Each node is visited exactly once.

Space Complexity: O(n) - Recursive call stack for n nodes.

\*/

    int Queue::sum()

    {

        return sumHelper(front);

    }

    int Queue::sumHelper(Node \*node)

    {

        if (node == nullptr)

        {

            return 0;

        }

        return node->data + sumHelper(node->next);

    }

## Find Max

    // Recursive function to find the maximum element in a queue

    /\*

    Time Complexity: O(n) - Each node is visited exactly once.

    Space Complexity: O(n) - Recursive call stack for n nodes.

    \*/

    int Queue::findMax()

    {

        return findMaxHelper(front, INT\_MIN);

    }

    int Queue::findMaxHelper(Node \*node, int maxSoFar)

    {

        if (node == nullptr)

        {

            return maxSoFar;

        }

        maxSoFar = max(maxSoFar, node->data);

        return findMaxHelper(node->next, maxSoFar);

    }

## Search element

    // Recursive function to search for an element in a queue

    /\*

    Time Complexity: O(n) - Each node is visited until the element is found or the end is reached.

    Space Complexity: O(n) - Recursive call stack for n nodes.

    \*/

    bool Queue::search(int value)

    {

        return searchHelper(front, value);

    }

    bool Queue::searchHelper(Node \*node, int value)

    {

        if (node == nullptr)

        {

            return false;

        }

        if (node->data == value)

        {

            return true;

        }

        return searchHelper(node->next, value);

    }

## Remove all occurrences

    void Queue::removeAll(int value)

    {

        front = removeAllHelper(front, value);

    }

    Node \*Queue::removeAllHelper(Node \*node, int value)

    {

        if (node == nullptr)

        {

            return nullptr;

        }

        if (node->data == value)

        {

            Node \*temp = node->next;

            delete node;

            return removeAllHelper(temp, value);

        }

        node->next = removeAllHelper(node->next, value);

        return node;

    }

## Count all even numbers

   // Recursive function to count the number of even elements in a queue

    /\*

    Time Complexity: O(n) - Each node is visited exactly once.

    Space Complexity: O(n) - Recursive call stack for n nodes.

    \*/

    int Queue::countEven()

    {

        return countEvenHelper(front);

    }

    int Queue::countEvenHelper(Node \*node)

    {

        if (node == nullptr)

        {

            return 0;

        }

        int count = (node->data % 2 == 0) ? 1 : 0;

        return count + countEvenHelper(node->next);

    }

## Check is sorted

    // Recursive function to check if the queue is sorted

    /\*

    Time Complexity: O(n) - Each node is visited exactly once.

    Space Complexity: O(n) - Recursive call stack for n nodes.

    \*/

    bool Queue::isSorted()

    {

        return isSortedHelper(front);

    }

    bool Queue::isSortedHelper(Node \*node)

    {

        if (node == nullptr || node->next == nullptr)

        {

            return true;

        }

        if (node->data > node->next->data)

        {

            return false;

        }

        return isSortedHelper(node->next);

    }

## Find middle of queue

 // Recursive function to find middle element of a queue

    /\*

    Time Complexity: O(n) - Count function takes O(n), and middle search takes O(n).

    Space Complexity: O(n) - Recursive call stack for middle search.

    \*/

    int Queue::findMiddle()

    {

        int size = count(); // Use the count function to get the size of the queue

        int middleIndex = size / 2;

        return findMiddleHelper(front, middleIndex, 0);

    }

    int Queue::findMiddleHelper(Node \*node, int middleIndex, int currentIndex)

    {

        if (node == nullptr)

        {

            return -1; // Should not happen if count is accurate

        }

        if (currentIndex == middleIndex)

        {

            return node->data;

        }

        return findMiddleHelper(node->next, middleIndex, currentIndex + 1);

    }

## Delete alternate nodes

    // Recursive function to delete alternate nodes

    /\*

    Time Complexity: O(n) - Each node is visited exactly once.

    Space Complexity: O(n) - Recursive call stack for n/2 recursive calls.

    \*/

    void Queue::deleteAlternate()

    {

        front = deleteAlternateHelper(front, true);

    }

    Node \*Queue::deleteAlternateHelper(Node \*node, bool keep)

    {

        if (node == nullptr)

        {

            return nullptr;

        }

        if (keep)

        {

            node->next = deleteAlternateHelper(node->next, false);

            return node;

        }

        else

        {

            Node \*temp = node->next;

            delete node;

            return deleteAlternateHelper(temp, true);

        }

    }