Contents

[Notes [Test Prep] 2](#_Toc58943631)

[List Recursive [Pointer-based] 2](#_Toc58943632)

[1. Add [specific index] - Has helper function √ 2](#_Toc58943633)

[2. Remove [specific index] √ 3](#_Toc58943634)

[3. Count - has helper function √ (number of occurrence) 4](#_Toc58943635)

[4. Reverse √ 4](#_Toc58943636)

[5. Search √ 4](#_Toc58943637)

[6. Sum √ 5](#_Toc58943638)

[7. Display √ 5](#_Toc58943639)

[8. findLargest √ 6](#_Toc58943640)

[9. SumOfEven 6](#_Toc58943641)

[List Recursive [Array] 6](#_Toc58943642)

[1. Add [Specific Index] √ 6](#_Toc58943643)

[2. Remove [Specific Index] √ 7](#_Toc58943644)

[3. Count (returns number of occurrence) √ 7](#_Toc58943645)

[4. Reverse √ 8](#_Toc58943646)

[5. Sum √ 8](#_Toc58943647)

[Stack Recursive Display 9](#_Toc58943648)

[1. Reverse √ 9](#_Toc58943649)

[2. Display √ 9](#_Toc58943650)

[3. Display Reversed √ 10](#_Toc58943651)

[Stack Recursive Functions [Pointer-based] 11](#_Toc58943652)

[1. Count √ 11](#_Toc58943653)

[Queue Recursive Functions [Pointer-based] 12](#_Toc58943654)

[1. Count √ 12](#_Toc58943655)

[2. Reverse √ 12](#_Toc58943656)

# Notes [Test Prep]

NOT TESTED – HASH TABLE, STACK

TESTED – LINKED LIST, QUEUE [POINTER], RECURSION, TIME COMPLEXITY

# List [Pointer-based] Recursive

## Add [specific index] - Has helper function √

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

if (index < size) {

addR2(index, item, firstNode,0);

return true;

}

}

Node\* List::addR2(int index, ItemType item, Node\* n,int count) {

if (count == index) {

Node\* newNode = new Node;

newNode->item = n->item;

newNode->next = n->next;

n->next = newNode;

n->item = item;

size++;

return NULL;

}

else {

addR2(index, n->item, n->next, count+1);

return NULL;

}

}

## Remove [specific index] √

bool List::removeR(int index) {

if (index < size) {

removeR2(index, NULL,firstNode ,0);

return true;

}

}

Node\* List::removeR2(int index, Node\* current, Node\* next, int count) {

if (index == count) {

current->next = next->next;

next->next = NULL;

delete(next);

size--;

return NULL;

}

else if (index == 1) {

Node\* tempNode = firstNode;

firstNode = firstNode->next;

tempNode->next = NULL;

delete(tempNode);

size--;

return NULL;

}

else {

removeR2(index, next, next->next, count + 1);

return NULL;

}

}

## Count - has helper function √ (number of occurrence)

int List::countR( ItemType item)

{

return countR2(front, item);

}

int List::countR2(Node\* temp, ItemType item)

{

if (temp == NULL)

return 0;

else

{

if (temp->item == item)

return 1 + countR2(temp->next, item);

else

return countR2(temp->next, item);

}

}

## Reverse √

void List::reverseR()

{

reverseR2(front);

}

List::Node\* List::reverseR2(Node\* node)

{

if (node == NULL) // list is empty

return NULL;

if (node->next == NULL) { // if there is only one node

front = node;

return node;

}

List::Node\* node1 = reverseR2(node->next); // reverse 2nd node onwards

node1->next = node; // adjust last node of reversed list

node->next = NULL; // node is now last node, so its next is NULL

return node;

## Search √

bool search(struct Node\* front, int x)   
{   
// Base case   
if (front == NULL)   
return false;   
  
// If key is present in current node, return true   
if (front->item == x)   
return true;   
  
// Recur for remaining list   
return search(front->next, x);   
}

## Sum √

int List::sumList(Node\* head)

{

//base case

if (size == 0 || head = NULL)

{

return 0;

}

else {

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

}

}

OR

int Sum(Node \*head)

{

if ( head != NULL )

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

else

return 0;

}

## Display √

void List::displayRecursively(Node\* current)                 
{  
 if (current != NULL){  
 cout << current->data << endl;  //print data  
 displayRecursively(current->next);  
 }  
}

## findLargest √

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

int current = p->item;

int next;

if (p->next == NULL) {

//The value at this node is obviously larger than a non-existent value

return current;

} else {

//Recur to find the highest value from the rest of the LinkedList

next = findLargest(p->next);

}

//Return the highest value between this node and the end of the list

if (current > next) {

return current;

} else {

return next;

}

}

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)

{

if ( head != NULL && head->data % 2 == 0)

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

elseif

return 0;

}

int Sum(Node\* head)

{

If ( head == NULL)

{

Return 0;

}

Else If (head->item % 2 == 0)

{

return head->item + Sum(head->item);

}

}

Else

{

Return sum(head->item);

}

}

# List [Array] Recursive

## Add [Specific Index] √

//ADDING ITEM AT A SPECIFIC INDEX.

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

if (index < size) {

addR(index + 1, items[index]);

items[index] = item;

return true;

}

else if (index == size) {

items[index] = item;

return true;

}

size++;

}

## Remove [Specific Index] √

void List::removeR(int index) {

if (index < size) {

if (index == size - 1) {

items[index] = items[index + 1];

size--;

}

else {

items[index] = items[index + 1];

removeR(index + 1);

}

}

}

## Count (returns number of occurrence) √

int List::count(ItemType item)

{

return countR(0, item);

}

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

{

if (index >= size)

{

return 0;

}

else

{

if (items[index] == item)

{

return 1 + countR(index + 1, item);

}

else

return countR(index + 1, item);

}

}

## Reverse √

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

{

    // base case

if (n==1)

cout << array[n-1];

    // recursive step

else

{

cout << array[n-1] << endl;

displayReverseOrder(array, n-1);

}

}

## Sum √

int findSum(int array[], int N)

{

if (N <= 0)

return 0;

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

}

## SumOfEven √

int List::addEven(int n) {

if (size == n + 1) {

if (items[n] % 2 == 0) {

return items[n];

}

else {

return 0;

}

}

else {

if (items[n] % 2 == 0) {

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

}

else {

return 0 + addEven(n+1);

}

}

}

# Stack Recursive Display

## Reverse √

void Stack::reverser()

{

Node\* tempNode = topNode;

Stack tempStack;

Stack tempStack2;

While(tempNode!= NULL){

tempStack.push(tempNode->item);

tempNode = tempNode->next;}

while (!isEmpty()){

pop();

}

tempNode = tempStack.topNode;

while (tempNode!=NULL) {

tempStack2.push(tempNode->item)

tempNode = tempNode->next;

}

tempNode = tempStack2.topNode;

while(tempNode != NULL) {

push(tempNode->item);

tempNode = tempNode->next;

` }

}

## Display √

void Stack::PrintStack(Stack s)

{

    // If stack is empty then return

    if (s.empty()) //remember to write

        return;

    int x = s.top();

    // Pop the top element of the stack

    s.pop();

    // Recursively call the function PrintStack

    PrintStack(s);

    // Print the stack element starting

    // from the bottom

    cout << x << " ";

    // Push the same element onto the stack

    // to preserve the order

    s.push(x);

}

## Display Reversed √

\*Must #include <sstream>\*

void Stack::displayInOrderOfInsertionR() {

Stack temp;

if (!isEmpty()) {

cout << displayInOrderOfInsertionR2(temp);

cout << endl;

}

displayInOrderOfInsertionR3(temp);

}

string Stack::displayInOrderOfInsertionR2(Stack &s) {

if (isEmpty()) {

return "";

}

else {

ItemType temp;

pop(temp);

s.push(temp);

return displayInOrderOfInsertionR2(s) + to\_string(temp);

}

}

void Stack::displayInOrderOfInsertionR3(Stack &s) {

if (s.isEmpty()) {

}

else {

ItemType temp;

s.pop(temp);

push(temp);

displayInOrderOfInsertionR3(s);

}

}

# Stack Recursive Functions [Pointer-based]

## Count √

int Stack::count()

{

Node\* tmp = topNode;

return count2(0, tmp);

}

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

{

if(tmp->next != NULL)

{

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

}

else

return index;

}

# Queue Recursive Functions [Pointer-based]

## Count √

int Queue::count()

{

Node\* tmp = frontNode;

return count2(0, tmp);

}

int Queue::count2(int index, Node\* tmp)

{

if(tmp->next != NULL)

{

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

}

else

return index;

}

## Reverse √

void reverseQueue(Queue &q) {

// Base case

// reverse of an empty queue is an empty queue

if (q.empty()) {

return;

}

// remove an element from queue and store it in a variable

int current = q.front();

q.pop();

// recursively call the reverseQueue method on remaining queue

reverseQueue(q);

// add the removed element to the end of the reversed queue

q.push(current);

}