|  |  |
| --- | --- |
| **Design an algorithm and write code to remove the duplicate characters in a string without using any additional buffer. Write the test cases for this method.** | **Implement a Doubly Linked List that has methods a) add b) remove.** |
| public string removeDuplicates(string str) {      //test cases : aaabbb, ababab, aaaa, abcd, null, "", aabc  if (string.IsNullOrEmpty(str) == true) return "";  var arr = str.ToArray();  int tail = 1;  for (int i = 1; i < arr.Length; i++) {  int j = 0;  for (; j < tail; j++) {  if (arr[i] == arr[j]) break;  }  if (j == tail) {  arr[j] = arr[i];  tail++;  }  }  var result = new char[tail];  for (int i = 0; i < result.Length; i++) {  result[i] = arr[i];  }  return new string(result);  } | public class DoublyLinkedList {  public Node head;  public Node tail;  public void add(int v) {  if (head == null) {  head = new Node(v);  tail = head;  return;  }  tail.next = new Node(v);  tail.next.prev = tail;  tail = tail.next;  }  public Node remove(int v) {  if (head == null) return null;  if (head.value == v) {  if (head == tail) {  var ret = head;  head = null;  tail = null;  return ret;  }  var ret2 = head;  head.next.prev = null;  head = head.next;  return ret2;  }  for (var node = head.next; node.next != null; node = node.next) {  if (node.value == v) {  node.prev.next = node.next;  node.next.prev = node.prev;  return node;  }  }  if (tail.value == v) {  var ret = tail;  tail.prev.next = null;  tail = tail.prev;  return ret;  }  return null;  }  } |
| **Write code to remove duplicates from an unsorted linked list without using a temporary buffer.** |
| public string removeDuplicates(DoublyLinkedList list) {  if (list == null || list.head == null) return;  var tail = list.head.next;  var prev = list.head;  for (var i = tail; i != null; i = i.next) {  Node j = list.head;  for (; j != tail; j = j.next) {  if (i.value == j.value) break;  }  if (j == tail) {  j.value = i.value;  prev = tail;  tail = tail.next;  }  }  prev.next = null;  } |

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| **Implement a) Stack b) Queue.** | **Write code for Tower of Hanoi.** | | | | | | |
| public class Stack {  private Node top;  public void push(int v) {  var node = new Node(v);  node.next = top;  top = node;  }  public int pop() { {  if (top == null) return -1;  var ret = top.value;  top = top.next;  return ret;  }  }  public class Queue {  public Node head;  public Node tail;  public void enq(int v) {  if (head == null) {  head = new Node(v);  tail = head;  return;  }  var node = new Node(v);  tail.next = node;  tail = node;  }  public int deq() {  if (head == null) return -1;  var ret = head.value;  head = head.next;  tail = (head == null) ? null : tail;  return ret;  }  } | public void solve(int n) {  if (n <= 0) return;  solve(n, 'A', 'B', 'C');  }  private void solve(int n, char from, char tmp, char to) {  if (n == 1) {  PrintLn("Moving " + n + " from " + from + " to " + to);  return;  }  solve(n - 1, from, to, tmp);  PrintLn("Moving " + n + " from " + from + " to " + to);  solve(n - 1, tmp, from, to);  } | | | | | | |
| **Implement the following algorithms: a) In-Order b) Pre-Order c) Post-Order d) Level-Order e) Insert Node Method** | | | | | | |
| public void PrintInOrder() {  PrintInOrder(Root);  } | | | public void PrintPreOrder() {  PrintPreOrder(Root);  } | | | public void PrintPostOrder() {  PrintPostOrder(Root);  } |
| private void PrintInOrder(Node node) {  if (node == null) return;  PrintInOrder(node.Left);  Print(node.ToString() + ", ");  PrintInOrder(node.Right);  }  private void PrintPreOrder(Node node) {  if (node == null) return;  Print(node.ToString() + ", ");  PrintPreOrder(node.Left);  PrintPreOrder(node.Right);  }  private void PrintPostOrder(Node node) {  if (node == null) return;  PrintPostOrder(node.Left);  PrintPostOrder(node.Right);  Print(node.ToString() + ", ");  } | | | | public void Insert(Node node) {  Root = Insert(Root, node);  }  private Node Insert(Node cur, Node node) {  if (cur == null) {  return node;  }  if (node.left < cur.left) {  cur.Left = Insert(cur.Left, node);  } else {  cur.Right = Insert(cur.Right, node);  }  return cur;  } | | |
| public void PrintLevelOrder() {  if (Root == null) return;  var q = new Queue<Node>();  q.Enqueue(Root);  while (q.Count > 0) {  var node = q.Dequeue(); Print(node.ToString() + ", ");  if (node.Left != null) q.Enqueue(node.Left);  if (node.Right != null) q.Enqueue(node.Right);  }  } | | **Implement a) Depth First Search b) Breadth First Search** | | | | public class AdjList {  private List<Edge>[] vertexList;  public AdjList(int count) {  vertexList = new List<Edge>[count];  for (int i = 0; i < count; i++) {  vertexList[i]= new List<Edge>();  }  }  } | |
| public class Edge {  public int vertex { get; set; }  public int cost { get; set; }  public Edge(int v, int c) {  vertex = v;  cost = c;  }  } | | | |
| public void DepthFirstSearch() {  if (vertexList == null || vertexList.Count() <= 0) return;  var hashSet = new HashSet<int>();  DepthFirstSearch(0, hashSet);  }  public void DepthFirstSearch(int vertex, HashSet<int> hashSet) {  hashSet.Add(vertex);  var edgeList = vertexList[vertex];  foreach (var edge in edgeList) {  if (hashSet.Contains(edge.vertex) == true) continue;  DepthFirstSearch(edge.vertex, hashSet);  }  } | | | public void BreadthFirstSearch() {  if (IsEmptyList() == true) return;  var q = new Queue<int>();  q.Enqueue(0);  while (q.Count > 0) {  var vertex = q.Dequeue();  var edgeList = vertexList[vertex];  Print(vertex + " - ");  foreach (var edge in edgeList) {  q.Enqueue(edge.Vertex);  }  }  } | | | | |
|  | | |  | | | | |