|  |  |
| --- | --- |
| **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 Node removeDuplicates(Node head) {  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;  return head;  } |

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **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 hanoi(int n) {  if (n <= 0) return;  solve(n, 'A', 'B', 'C');  }  private void hanoi(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(int value) {  Root = Insert(Root, value);  }  private Node Insert(Node node, int value) {  if (node == null) {  return new Node(value);  }  if (value < node.value) {  node.Left = Insert(node.Left, value);  } else {  node.Right = Insert(node.Right, value);  }  return node;  } | | |
| 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<List<Edge>> vertexList;  public AdjList(int count) {  vertexList = new List<List<Edge>>();  for (int i = 0; i < count; i++) {  vertexList.Add(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() {  var hs = new HashSet<int>();  var q = new Queue<int>();  q.Enqueue(0);  hs.Add(0);  while (q.Count > 0) {  var vertex = q.Dequeue();  var edges = map[vertex];  printf(vertex + " - ");  foreach (var edge in edges) {  if (hs.Contains(edge.vertex) == true) continue;  q.Enqueue(edge.vertex);  hs.Add(edge.vertex);  }  }  } | | | | | | | | |
| **Given a (decimal – e.g. 3.72) number that is passed in as a string, print the binary representation. If the number cannot be represented accurately in binary, print “ERROR”.** | | | | | | **Implement the following: a) Bubble Sort b) Selection Sort c) Merge Sort d) Quick Sort e) Bucket Sort** | | | | | | | | |
| private string PrintBinary(string n) {  var split = n.Split('.');  int num = Convert.ToInt32(split[0]);  var dec = Convert.ToDouble("0." + split[1]);  var decString = "";  var stack = new Stack<char>();  while (num > 0) {  stack.Push((num % 2 == 0) ? '0' : '1');  num >>= 1;  }  var sbNum = new StringBuilder();  var sbDec = new StringBuilder();  while (stack.Count > 0) sbNum.Append(stack.Pop());  int cnt = 0;  while (dec != 0.0) {  if (cnt > 32) return "ERROR";  dec \*= 2.0;  if (dec >= 1.0) {  sbDec.Append("1");  dec -= 1.0;  } else {  sbDec.Append("0");  }  }  cnt++;  return sbNum.ToString();+ "." + sbDec.ToString();  } | | | | | | private void bubbleSort(int[] n) {  for (int i = n.Length - 1; i >= 0; i--) {  var swapped = false;  for (int j = 0; j < i; j++) {  if (n[j] > n[j + 1]) {  var tmp = n[j + 1];  n[j + 1] = n[j];  n[j] = tmp;  swapped = true;  }  }  if (swapped == false) break;  }  }  private void selectionSort(int[] n) {  for (int i = 0; i < n.Length; i++) {  int index = i;  for (int j = i + 1; j < n.Length; j++) {  if (n[j] < n[index]) {  index = j;  }  }  int tmp = n[i];  n[i] = n[index];  n[index] = tmp;  }  } | | | | | | | | |
| private int[] mergeSort(int[] n) {  return mergeSort(n, 0, n.Length);  }  private int[] mergeSort(int[] n, int s, int e) {  if (e - s <= 1) return new int[] { n[s] };  int mid = (s + e) / 2;  int count = e - s;  var left = mergeSort(n, s, mid);  var right = mergeSort(n, mid, e);  var tmp = new int[count];  int li = 0;  int ri = 0;  for (int index = 0; index < count; index++) {  if (li < left.Length && (ri >= right.Length || left[li] < right[ri])) {  tmp[index] = left[li++];  } else {  tmp[index] = right[ri++];  }  }  return tmp;  } | | | | | | | | | | public void quickSort(int[] n) {  quickSort(n, 0, n.Length - 1);  }  private void quickSort(int[] n, int s, int e) {  if (s >= e) return;  int p = partition(n, s, e);  quickSort(n, s, p - 1);  quickSort(n, p + 1, e);  }  private int partition(int[] n, int s, int e) {  int piv = n[e], p = e;  for (int i = 0; i < p; i++) {  if (n[i] > piv) {  swap(n, i, p - 1);  swap(n, p, p - 1);  i--;  p--;  }  }  return p;  } | | | | |
| public void bucketSort(int[] n) {  int max = Int32.MinValue;  for (int i = 0; i < n.Length; i++) max = (n[i] > max) ? n[i] : max;  var buckets = new List<int>[max + 1];  for (int i = 0; i < buckets.Length; i++) buckets[i] = new List<int>();  for (int i = 0; i < n.Length; i++) {  var b = n[i] % (max + 1);  buckets[b].Add(n[i]);  }  int index = 0;  for (int i = 0; i < buckets.Length; i++) {  var bucket = buckets[i];  for (int j = 0; j < bucket.Count; j++) {  n[index++] = bucket[j];  }  }  } | | | | | | | | | | **Implement binary search.** | | | | |
| public int BinarySearch (int[] arr, int key) {  int min = 0;  int max = arr.Length - 1;      while (min <= max) {          int mid = (min + max) / 2;          if (key == arr[mid]) {              return mid;          } else if (key < arr[mid]) {             max = mid - 1;          } else {              min = mid + 1;          }     }     return -1;  } | | | | |
| **Given an integer, write a function to determine if it is a power of three.** | | | | **Write a function to find a) gcd b) lcm of two numbers.** | | | | | | | | | | |
| public bool IsPowerOfThree(int n) {  return (Math.Log10(n) / Math.Log10(3)) % 1 == 0;  } | | | | public int gcd(int a, int b) {  while (b != 0) {  int t = b;  b = a % b;  a = t;  }  return a; } | | | | | | | | | //gcd(a,b) \* lcm(a,b) = ab  public int lcm(int a, int b) {  if (a == 0 || b == 0) return 0;  var t = gcd(a, b);  if (t == 0) return 0;  return (a \* b) / t;  } | |
| **Write an efficient algorithm that searches for a value in an *m x n* matrix. This matrix has the following property: it is sorted from left to right, top to bottom.** | | | | | **Implement a function to check if a tree is balanced. For the purposes of this question, a balanced tree is defined to be a tree such that no two leaf nodes differ in distance from the root by more than one.** | | | | | | | | | |
| public bool SearchMatrix(int[,] matrix, int target) {  if (matrix == null) return false;  var m = matrix.GetLength(1);  var n = matrix.GetLength(0);  if (m == 0 || n == 0) return false;  int start = 0, end = m \* n - 1;  while (start <= end) {  int mid = (start + end) >> 1;  int midX = mid / m;  int midY = mid % m;  if (matrix[midX, midY] == target) return true;  else if (target < matrix[midX, midY]) end = mid - 1;  else start = mid + 1;  }  return false;  } | | | | | public bool isBalanced() {  if (root == null) return false;  return getMaxDepth(root) - getMinDepth(root) <= 1;  }  public int getMinDepth(Node node) {  if (node == null) return 0;  return 1 + Math.Min(getMinDepth(node.left), getMinDepth(node.right));  }  public int getMaxDepth(Node node) {  if (node == null) return depth;  return 1 + Math.Max(getMaxDepth(node.left), getMaxDepth(node.right));  } | | | | | | | | | |
| **Implement a full tree with *n* nodes.** | **Given a sorted (increasing order) array, write an algorithm to create a binary tree with minimal height.** | | | | | | | | | | | | | |
| public Node createFullTree(int n) {  if (n <= 0) return null;  int cnt = 1;  var q = new Queue<int>();  var root = new Node(0);  q.Enqueue(root);  while (cnt < n && q.Count > 0) {  var node = q.Dequeue();  if (cnt++ < n) {  node.left = new Node(0);  q.Enqueue(node.left);  }  if (cnt++ < n) {  node.right = new Node(0);  q.Enqueue(node.right);  }  }  return root;  } | public Node createBinaryTree(int[] arr) {  if (arr == null || arr.Length == 0) return null;  return createBinaryTree(arr, 0, arr.Length - 1);  }  private Node createBinaryTree(int[] arr, int min, int max) {  if (min > max) return null;  int mid = (min + max) >> 1;  var node = new Node(arr[mid]);  node.left = createBinaryTree(arr, min, mid - 1);  node.right = createBinaryTree(arr, mid + 1, max);  return node;  } | | | | | | | | | | | | | |
| **Write an algorithm to find the ‘next’ node of a given node in a binary search tree where each node has a link to its parent for a) In-Order Successor b) Pre-Order Successor c) Post-Order Successor** | | | | | | | | | | | | | |
| public Node getInOrderSuccessor(Node node) {  if (node == null) return null;  if (node.parent == null || node.right != null) {  return getLeftMost(node.right);  }  var cur = node;  while (node.parent != null) {  node = node.parent;  if (node.left == cur) return node;  cur = node;  }  return null;  } | | | | | | | | | | private Node getLeftMost(Node node) {  if (node == null) return null;  while (node.left != null) node = node.left;  return node;  } | | | |
| public Node getPreOrderSuccessor(Node node) {  if (node == null) return null;  if (node.left != null) return node.left;  if (node.right != null) return node.right;  var cur = node;  while (node.parent != null) {  node = node.parent;  if (node.left == cur && node.right != null) {  return node.right;  }  cur = node;  }  return null;  } | | | | | | | public Node getPostOrderSuccessor(Node node) {  if (node == null) return null;  if (node.parent == null) return null;  if (node.parent.left == node) {  if (node.parent.right == null) return node.parent;  node = node.parent.right;  while (node.left != null) node = node.left;  while (node.right != null) node = node.right;  return node;  }  return node.parent;  } | | | | | | | |
| **Design an algorithm and write code to find the first common ancestor of two nodes in a binary tree. Avoid storing additional nodes in a data structure. NOTE: This is not necessarily a binary search tree.** | | | | | | | | **Given a string, find the first non-repeating character in it and return its index. If it doesn't exist, return -1.** | | | | | | |
| public Node getCommonAncestor(Node n1, Node n2) {  if (n1 == null || n2 == null || root == null) return null;  if (n1 == root || n2 == root) return null;  return getCommonAncestor(root, n1, n2);  }  private Node getCommonAncestor(Node parent, Node n1, Node n2) {  if (parent == null) return null;  if ((contains(parent.left, n1) && contains(parent.right, n2)) ||  (contains(parent.left, n2) && contains(parent.right, n1))) {  return parent;  }  else if (contains(parent.left, n1) && contains(parent.left, n2)) {  if (parent.left == n1 || parent.left == n2) return parent;  else return getCommonAncestor(parent.left, n1, n2);  }  else if (contains(parent.right, n1) && contains(parent.right, n2)) {  if (parent.right == n1 || parent.right == n2) return parent;  else return getCommonAncestor(parent.right, n1, n2);  }  return null;  }  private bool contains(Node node, Node target) {  if (node == null) return false;  if (node == target) return true;  return contains(node.left, target) || contains(node.right, target);  } | | | | | | | | public int FirstUniqChar(string s) {  var map = new int[26];  for (int i = 0; i < s.Length; i++) {  var c = s.ElementAt(i);  map[c - 'a']++;  }  for (int i = 0; i < s.Length; i++) {  if (map[s.ElementAt(i) - 'a'] == 1) return i;  }  return -1;  } | | | | | | |
| **Write a method to sort an array of strings so that all the anagrams are next to each other.** {"silent","abcdef","kfc","listen","tenlis"} 🡪 {,"abcdef","kfc","silent","listen","tenlis"} | | | | | | |
| public void sortChars(string[] s) {  Array.Sort(s, new AnagramComparer());  }  private class AnagramComparer : IComparer<String> {  public int Compare(String s1, String s2) {  var c1 = s1.ToArray();  var c2 = s2.ToArray();  Array.Sort(c1);  Array.Sort(c2);  var a1 = new string(c1);  var a2 = new string(c2);  return a1.CompareTo(a2);  }  } | | | | | | |

|  |  |  |  |
| --- | --- | --- | --- |
| **Given an integer *n*, return all distinct solutions to the *n*-queens puzzle. ‘Q’ and ‘.’ both indicate a queen and an empty space respectively.** | | | **Given a collection of intervals, merge all overlapping intervals.**  **For example, Given**[1,3],[2,6],[8,10],[15,18]**, return**[1,6],[8,10],[15,18]**.** |
| public List<List<string>> SolveNQueens(int n) {  if (n <= 0) return null;  var rows = new int[n];  var result = new List<List<string>>();  for (int i = 0; i < n; i++) rows[i] = -1;  placeQueens(rows, 0, result);  return result;  }  private void placeQueens(int[] rows, int i, List<List<string>> result) {  if (i == rows.Length) {  var str = new List<string>();  for (int x = 0; x < rows.Length; x++) {  var sb = new StringBuilder();  for (int y = 0; y < rows.Length; y++) {  sb.Append((rows[x] == y) ? "Q" : ".");  }  str.Add(sb.ToString());  }  result.Add(str);  return;  }  for (int j = 0; j < rows.Length; j++) {  if (canPlaceQueen(rows, i, j) == true) {  var copy = new int[rows.Length];  Array.Copy(rows, copy, rows.Length);  copy[i] = j;  placeQueens(copy, i + 1, result);  if (i + 1 == rows.Length) break;  }  }  }  private bool canPlaceQueen(int[] rows, int i, int j) {  for (int x = 0; x < rows.Length; x++) if (rows[x] == j) return false;  for (int x = 0; x < rows.Length; x++)  if (rows[x] != -1 && rows[x] - x == j - i)  return false;  for (int x = 0; x < rows.Length; x++)  if (rows[x] != -1 && rows[x] + x == i + j)  return false;  return true;  } | | | public List<Interval> Merge(IList<Interval> intervals) {  if (intervals == null || intervals.Count == 0) return null;  if (intervals.Count == 1) return intervals.ToList();  var arr = intervals.ToArray();  Array.Sort(arr, new QComparer());  var result = new List<Interval>();  var last = arr[0];  //[1,3],[2,6],[8,10],[15,18] -> [1,6],[8,10],[15,18].  //[1,4],[2,3] //[1,4],[1,3] //[1,4],[4,5]  for (int i = 1; i < arr.Length; i++) {  var cur = arr[i];  if (cur.start <= last.end) {  last.end = Math.Max(cur.end, last.end);  } else {  result.Add(last);  last = cur;  }  }  result.Add(last);  return result;  }  private class QComparer : IComparer<Interval> {  public int Compare(Interval n1, Interval n2) {  return n1.start - n2.start;  }  }  public class Interval {  public int start;  public int end;  public Interval() { start = 0; end = 0; }  public Interval(int s, int e) { start = s; end = e; }  } |
| **In how many ways can you climb stairs to the top if you either climb 1 or 2 steps?** |
| public int ClimbStairs(int n) {  return ClimbStairs(n, new int[n + 1]);  }  private int ClimbStairs(int n, int[] map) {  if (n <= 1) return 1;  if (n == 2) return 2;  if (map[n] != 0) return map[n];  map[n] = ClimbStairs(n - 1) + ClimbStairs(n - 2);  return map[n];  } |
| public int ClimbStairs(int n) {  if (n <= 0) return 0;  if (n <= 2) return n;  int a = 1; int b = 2;  int sum = 0;  for (int i = 2; i < n; i++) {  sum = a + b;  a = b;  b = sum;  }  return sum;  } | **One way to serialize a binary tree is to use pre-order traversal. When we encounter a non-null node, we record the node's value. If it is a null node, we record using a sentinel value such as #. For example, the above binary tree can be serialized to the string "9,3,4,#,#,1,#,#,2,#,6,#,#", where # represents a null node. Given a string of comma separated values, verify whether it is a correct preorder traversal serialization of a binary tree. Find an algorithm without reconstructing the tree.** | | |
| public bool IsValidSerialization(string preorder) {  if (string.IsNullOrEmpty(preorder)) return false;  int numCnt = 0; int nullCnt = 0;  var nums = preorder.Split(',');  if (nums[0] == "#") {  if (nums.Length == 1) return true;  else return false;  }  foreach (var num in nums) {  if (num == "#") {  nullCnt++;  } else {  if (nullCnt == numCnt + 1) return false;  numCnt++;  }  if (nullCnt > numCnt + 1) return false;  }  return nullCnt == numCnt + 1;  } | | |
| **Given a sorted linked list, delete all nodes that have duplicate numbers, leaving only *distinct* numbers from the original list.** | | **You have two very large binary trees: T1, with millions of nodes, and T2, with hun­dreds of nodes. Create an algorithm to decide if T2 is a subtree of T1.** | |
| public Node DeleteDuplicates(Node head) {  if (head == null || head.next == null) return head;  Node dummy = new Node(0);  dummy.next = head;  head = dummy;  while (head.next != null && head.next.next != null) {  if (head.next.val == head.next.next.val) {  int val = head.next.val;  while (head.next != null && head.next.val == val) {  head.next = head.next.next;  }  } else {  head = head.next;  }  }  return dummy.next;  } | | public bool isSub(Node t1, Node t2) {  if (t2 == null) return true;//empty tree is always a subtree  return isSubTree(t1, t2);  }  private bool isSubTree(Node n1, Node n2) {  if (n1 == null) return false;  if (n1.val == n2.val) {  if (isMatch(n1, n2)) return true;  }  return isSubTree(n1.left, n2) || isSubTree(n1.right, n2);  }  private bool isMatch(Node n1, Node n2) {  if (n2 == null && n1 == null) return true;  if (n2 != null && n1 == null) return false;  if (n2 == null && n1 != null) return true;  if (n1.val != n2.val) return false;  return isMatch(n1.left, n2.left) && isMatch(n1.right, n2.right);  } | |

|  |  |
| --- | --- |
| **Given a 2D board and a word, find if the word exists in the grid. The word can be constructed from letters of sequentially adjacent cell, where "adjacent" cells are those horizontally or vertically neighboring. The same letter cell may not be used more than once. For example, given board = [ ["ABCE"], ["SFCS"], ["ADEE"] ]**  **"ABCCED" returns true, "SEE" returns true, "ABCB" returns false.** |  |
| public bool Exist(char[,] board, string word) {  if (board == null || board.GetLength(0) == 0 || board.GetLength(1) == 0) return false;  if (string.IsNullOrEmpty(word) == true) return true;  int imax = board.GetLength(0), jmax = board.GetLength(1);  for (int i = 0; i < imax; i++) {  for (int j = 0; j < jmax; j++) {  if (board[i, j] == word.ElementAt(0)) {  if (Exist(board, word, i, j, 0, new bool[imax, jmax]) == true) return true;  }  }  }  return false;  }  private bool Exist(char[,] board, string word, int i, int j, int index, bool[,] visited) {  if (index == word.Length) return true;  if (i < 0 || i >= board.GetLength(0) || j < 0 || j >= board.GetLength(1)) return false;  if (visited[i, j] == true) return false;  var imax = board.GetLength(0);  var jmax = board.GetLength(1);  visited[i, j] = true;  if (board[i, j] == word.ElementAt(index)) {  if (Exist(board, word, i, j + 1, index + 1, visited) == true) return true;  if (Exist(board, word, i + 1, j, index + 1, visited) == true) return true;  if (Exist(board, word, i, j - 1, index + 1, visited) == true) return true;  if (Exist(board, word, i - 1, j, index + 1, visited) == true) return true;  }  visited[i, j] = false;  return false;  } |  |