**DOUBLY LINKED LIST IMPLEMENTATION:**

class Node: def __init__(self, value, next = None, prev = None): self.value = value self.next = next self.prev = prev **||||||** class DoublyLinkedList: def __init__(self): self.front = None self.back = None **|||** def printList(self): curr = self.front while curr: print(curr.value) curr = curr.next **|||** def insertFront(self, value): nn = Node(value) if self.front is None: self.front, self.back = nn, nn else: self.front.prev = nn nn.next = self.front self.front = nn **|||** def insertBack(self, value): nn = Node(value) if self.front is None: self.front, self.back = nn, nn else: self.back.next = nn nn.prev = self.back self.back = nn **|||** def popFront(self): if self.front is not None: toDelete = self.front if self.front == self.back: self.front, self.back = None, None else: self.front = self.front.next self.front.prev = None del toDelete **|||** def popBack(self): if self.front is not None: toDelete = self.back if self.front == self.back: self.front, self.back = None, None else: self.back = self.back.prev self.back.next = None **|||** def search(self, key): curr = self.front while curr: if curr.value == key: return True curr = curr.next return False

**DOUBLY LINKED LIST IMPLEMENTATION WITH SENTINAL NODES:**

class Node: def __init__(self, value, next = None, prev = None): self.value = value self.next = next self.prev = prev **||||||** class SentinalNodesLinkedList: **|||** def __init__(self): self.front = Node(None) self.back = Node(None, None, self.front) self.front.next = self.back **|||** def printList(self): curr = self.front.next while curr is not self.back: print(curr.value) curr = curr.next **|||** def push_front(self, value): nn = Node(value, self.front.next, self.front) self.front.next.prev = nn self.front.next = nn **|||** def pop_front(self): if self.front.next is not self.back: rm = self.front.next rm.next.prev = rm.prev rm.prev.next = rm.next del rm **|||** def push_back(self, value): nn = Node(value, self.back, self.back.prev) self.back.prev.next = nn self.back.prev = nn **|||** def pop_back(self): if self.front.next is not self.back: rm = self.back.prev rm.prev.next = rm.next rm.next.prev = rm.prev

**STACK IMPLEMENTATION:**

class Node: def __init__(self, value, next = None): self.value = value self.next = next **||||||** class Stack: def __init__(self): self.top = None self.size = 0 **|||** def printStack(self): curr = self.top while curr: print(curr.value) curr = curr.next **|||** def push(self, value): new_node = Node(value) if not self.isEmpty(): new_node.next = self.top self.top = new_node self.size += 1 **|||** def pop(self): if not self.isEmpty(): elem = self.top self.top = self.top.next self.size -= 1 del elem **|||** def isEmpty(self): return self.size == 0

**QUEUE IMPLEMENTATION:**

class Node: def __init__(self, value, next = None): self.value = value self.next = next **||||||** class Queue: def __init__(self): self.head = None self.tail = None self.size = 0 **|||** def enqueue(self, value): nn = Node(value) if self.tail is None and self.head is None: self.head = nn else: self.tail.next = nn self.tail = nn self.size += 1 **|||** def dequeue(self): if self.head is not None: elem = self.head if (self.size == 1): self.head, self.tail = None, None else: self.head = self.head.next self.size -= 1 del elem **|||** def isEmpty(self): return self.size == 0 **|||** def printQueue(self): curr = self.head while curr: print(curr.value) curr = curr.next

**SORTING TECHNIQUES:**

**Bubble Sort:**

def bubbleSort(arr): for i in range(0, len(arr)): for j in range(0, (len(arr) - i - 1)): if arr[j] > arr[j + 1]: arr[j], arr[j + 1] = arr[j + 1], arr[j] return arr

**Insertion Sort:**

def insertionSort(arr): for i in range(0, len(arr) - 1): j = i + 1 while j > 0 and arr[j] < arr[j - 1]: arr[j], arr[j - 1] = arr[j - 1], arr[j] j -= 1 return arr

**Selection Sort:**

def selectionSort(arr): for i in range(0, len(arr) - 1): min = i for j in range(i + 1, len(arr)): if arr[j] < arr[min]: min = j arr[i], arr[min] = arr[min], arr[i] return arr

**Merge Sort:**

```
def mergeSort(arr): if len(arr) == 1: return arr mid = len(arr) // 2 left = mergeSort(arr[0:mid]) right = mergeSort(arr[mid:len(arr)]) return merge(left, right) ||| def merge(arr1, arr2): combined = [] i, j, k = 0, 0, 0 while i < len(arr1) and j < len(arr2): if (arr1[i] <= arr2[j]): combined.append(arr1[i]) i += 1 else: combined.append(arr2[j]) j += 1 while i < len(arr1): combined.append(arr1[i]) i += 1 while j < len(arr2): combined.append(arr2[j]) j += 1 return combined
```

**Quick Sort:**

```
def quickSort(arr, low, high): if (low >= high): return arr pivot = arr[high] start, end = low, high while start <= end: while arr[start] < pivot: start += 1 while arr[end] > pivot: end -= 1 if (start <= end): arr[start], arr[end] = arr[end], arr[start] start += 1 end -= 1 quickSort(arr, low, end) quickSort(arr, start, high) return arr
```

**SEARCHING TECHNIQUES:**

**Linear Search:**

**Using Iteration:**

```
def linearSearch(arr, key): for i in range(0, len(arr)): if (arr[i] == key): return i return -1
```

**Using Recursion:**

```
def linearSearch(arr, key, index): if (index == len(arr)): return -1 if (arr[index] == key): return index return linearSearch(arr, key, index + 1)
```

**Binary Search:**

**Using Iteration:**

```
def binarySearch(arr, key): start = 0 end = len(arr) – 1 while start <= end: mid = start + (int) ((end - start) / 2) if (key < arr[mid]): end = mid – 1 elif (key > arr[mid]): start = mid + 1 elif (key == arr[mid]): return mid return -1
```

**Using Recursion:**

```
def binarySearch(arr, key, start, end): if start > end: return -1 mid = start + (int)((end - start) / 2) if (arr[mid] < key): return binarySearch(arr, key, mid + 1, end) elif (arr[mid] > key): return binarySearch(arr, key, start, mid - 1) elif (arr[mid] == key): return mid
```

**Self Adjusting List with search function implementation:**

**Without Sentinel Nodes:**

```
class SelfAdjustingList: ||| class Node: def __init__(self, dat, nx, pr): self.data = dat self.next = nx self.prev = pr ||| def __init__(self): self.front = None self.back = None ||| def search(self, v): curr = self.front while curr: if curr.data == v: if curr is not self.front: curr.prev.next = curr.next if curr is self.back: self.back = curr.prev else: curr.next.prev = curr.prev curr.next = self.front curr.prev = None self.front.prev = curr self.front = curr return True curr = curr.next return False ||| def append(self, data): # Create a new node new_node = self.Node(data, None, None) if self.front is None: self.front = new_node self.back = new_node else: new_node.prev = self.back self.back.next = new_node self.back = new_node ||| def display(self): # Display the list from front to back current = self.front while current: print(current.data) current = current.next
```

**With Sentinel Nodes:**

```
class SelfAdjustingList: class Node: def __init__(self, dat, nx = None, pr = None): self.data = dat self.next = nx self.prev = pr ||| def __init__(self): self.front = self.Node(None, None, None) self.back = self.Node(None, None, self.front) self.front.next = self.back ||| def search(self, v): curr = self.front.next while curr is not self.back: if (curr.data == v): curr.prev.next = curr.next curr.next.prev = curr.prev curr.next = self.front.next curr.prev = self.front self.front.next = curr return True curr = curr.next return False ||| def append(self, data): nn = self.Node(data) nn.prev = self.back.prev nn.next = self.back self.back.prev.next = nn self.back.prev = nn ||| def display(self): current = self.front.next while current is not self.back: print(current.data) current = current.next
```