

# Data Structures and Algorithms Notes

Soumik Seth

December 2, 2024



---

# CONTENTS

<b>1</b>	<b>C++ STL (Standard Template Libraries)</b>	<b>5</b>
1.1	Vectors . . . . .	5
1.1.1	Declaring and initializing a vector . . . . .	5
<b>2</b>	<b>Data Structures</b>	<b>7</b>
2.1	Types of Data Structures . . . . .	7
2.2	Linked Lists . . . . .	7
2.2.1	Creating a linked list . . . . .	8
2.2.2	Types of linked lists . . . . .	8
2.2.3	Creating a linked list from an array . . . . .	9
2.2.4	Find length of linked list . . . . .	10
2.2.5	Delete head of linked list . . . . .	10
2.2.6	Delete tail of a linked list . . . . .	11
2.2.7	Delete $K^{th}$ element of a linked list . . . . .	12
2.3	Stacks . . . . .	13
2.3.1	Implementing a stack using arrays . . . . .	13
2.3.2	Procedure . . . . .	13
2.3.3	FAQs . . . . .	15
<b>3</b>	<b>Sorting Algorithms</b>	<b>17</b>
3.1	Bubble Sort . . . . .	17
3.2	Quick Sort . . . . .	17



---

---

# CHAPTER 1

---

## C++ STL (STANDARD TEMPLATE LIBRARIES)

### 1.1 Vectors

Vectors are sequence containers representing arrays that can change in size. Just like arrays, vectors use contiguous storage locations for their elements, which means that their elements can also be accessed using offsets on regular pointers to its elements, and just as efficiently as in arrays. But unlike arrays, their size can change dynamically, with their storage being handled automatically by the container. Internally, vectors use a dynamically allocated array to store their elements.

#### 1.1.1 Declaring and initializing a vector

- `vector<int> v;` creates an empty container having size = 0
- `vector<int> v(5, 0);` creates a vector containing *five* zeroes
- `vector<int> v2(v1);` creates a vector containing all elements of v<sub>1</sub> in the same order
- `vector<pair<int, int>> v;` creates a vector of pairs
  1. `v = { {1, 2}, {3, 4}, {5, 6} };`
  2. `v[0] == {1, 2}`
  3. `v[0].first == 1; v[0].second == 2;`
- `vector<tuple<int, int, int>> v;` creates a vector of tuples
  1. `v = { {1, 2, 3}, {4, 5, 6} };`
  2. `v[0] == {1, 2, 3}`
  3. `get<0>(v[0]) == 1`
  4. `get<1>(v[0]) == 2`
  5. `get<2>(v[0]) == 3`



---

# CHAPTER 2

---

## DATA STRUCTURES

### 2.1 Types of Data Structures

- Linked Lists
- Stacks
- Queues
- Trees
- Graphs

### 2.2 Linked Lists

A linked list is a linear data structure resembling a chain, where each node is connected to the next, and each node represents an individual element. Unlike arrays, the elements in a linked list are not stored in contiguous memory locations. Every node in a linked list contains two pieces of information, the **data** of node and **pointer** to the next node.

Unlike arrays, the size of linked lists can be increased or decreased easily.

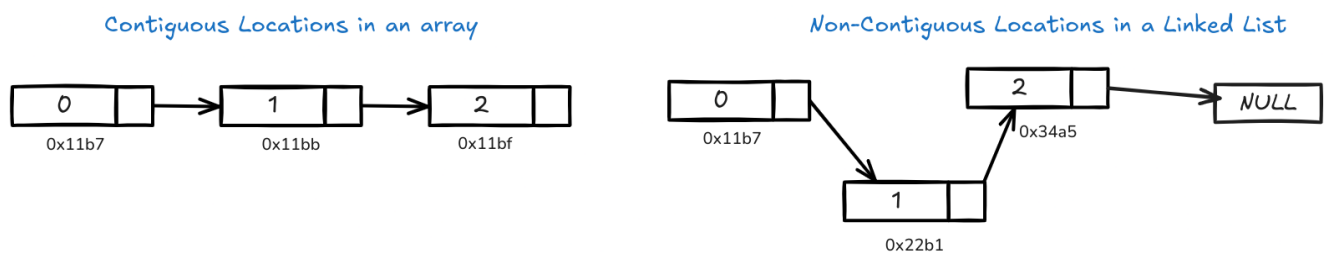
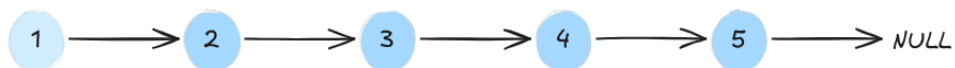


Figure 2.1: Array v/s Linked List

## 2.2.1 Creating a linked list

```
1  #include <bits/stdc++.h>
2  using namespace std;
3
4  struct Node {
5      int data;
6      Node* next;
7      Node(int val) : data(val), next(nullptr) {}
8  };
9
10 int main() {
11     Node* head = new Node(1); // Create LL with head 1
12     head->next = new Node(2);
13     head->next->next = new Node(3);
14     head->next->next->next = new Node(4);
15     head->next->next->next->next = new Node(5);
16
17     // create dummy node to travel the linked list
18     Node* temp = head;
19     while (temp != nullptr) {
20         cout << temp->data << ' ';
21         temp = temp->next; // go to the next node
22     }
23
24     return 0;
25 }
```

The above code produces a linked list that looks like this:



## 2.2.2 Types of linked lists

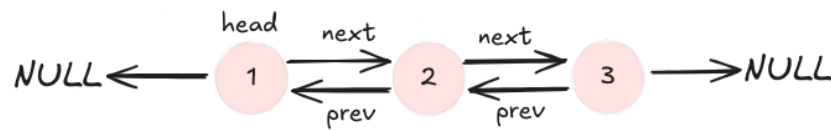
### Singly Linked Lists

In a singly linked list, each node points to the next node in the sequence. Traversal is straightforward but limited to moving in one direction, from the head to the tail.



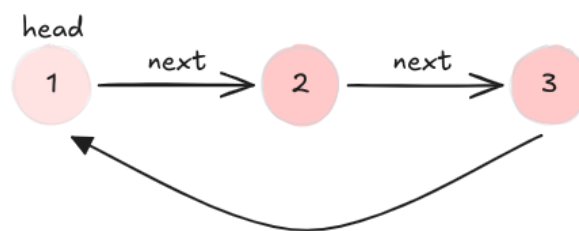
## Doubly Linked Lists

In this type, each node points to both the next node and the previous node, allowing for bidirectional connectivity.



## Circular Linked Lists

In a circular linked list, the last node points back to the head node, forming a closed loop.



### 2.2.3 Creating a linked list from an array

```
1 Node* arrayToLinkedList(int arr[], int size) {
2     if (size == 0) {
3         return nullptr; // empty linked list
4     }
5
6     Node* head = new Node(arr[0]); // create head of linked list
7
8     Node* curNode = head; // node to travel each node as it is added
9     for (int i = 1; i < size; ++i) { // start from i = 1, as head = arr[0]
10        curNode->next = new Node(arr[i]);
11        curNode = curNode->next; // go to the new node just created
12    }
13
14    return head;
15 }
```

### 2.2.4 Find length of linked list

```
1  int lengthOfLinkedList(Node* head) {  
2      int length = 0;  
3      Node* curNode = head;  
4  
5      while (curNode != nullptr) {  
6          length++;  
7          curNode = curNode->next;  
8      }  
9  
10     return length;  
11 }
```

### 2.2.5 Delete head of linked list

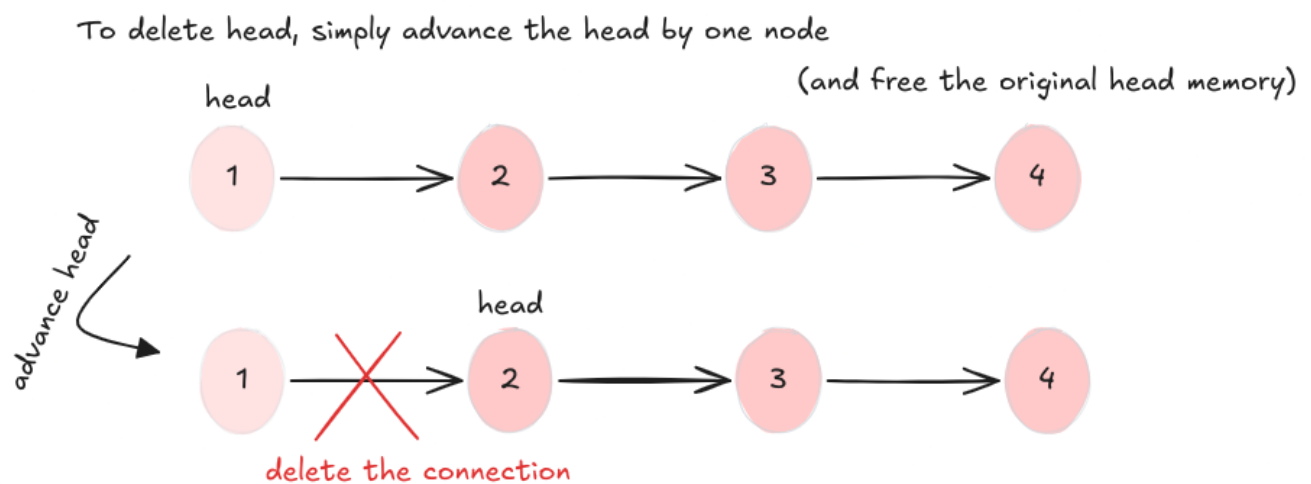




Figure 2.2: Delete head of a linked list



```
1  Node* deleteHead(Node* head) {
2      // if LL is empty or has only one element
3      // return null
4      if (head == nullptr || head->next == nullptr) {
5          return nullptr;
6      }
7      Node* prevHead = head;
8      head = head->next; // advance the current head
9      // delete connection of prevHead
10     prevHead->next = nullptr;
11     delete prevHead; // free memory
12
13     return head;
14 }
```

### 2.2.6 Delete tail of a linked list

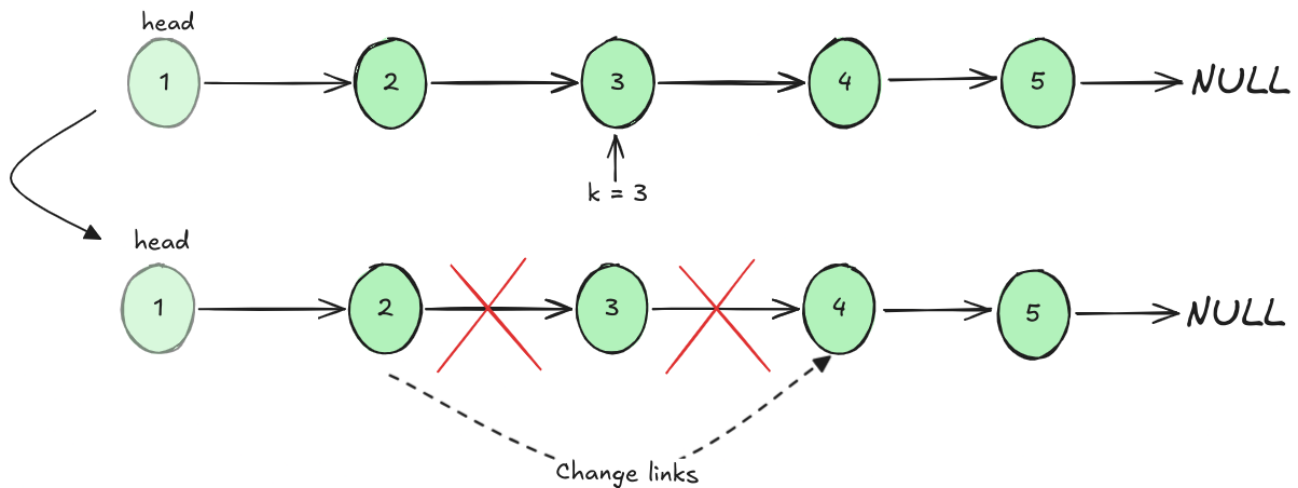


```
1  Node* deleteTail(Node* head) {
2      if (head == nullptr || head->next == nullptr) {
3          return nullptr;
4      }
5
6      // travel to node just before the tail node
7      Node* curNode = head;
8      while (curNode->next->next != nullptr) {
9          curNode = curNode->next;
10     }
11
12     Node* tail = curNode->next;
13     curNode->next = nullptr;
14
15     delete tail;
16     return head;
17 }
```

## 2.2.7 Delete $K^{th}$ element of a linked list

First travel to  $(k - 1)^{th}$  node.  
Then join  $(k - 1)^{th}$  and  $(k + 1)^{th}$  nodes.

If  $k == 1$ ,  
it denotes the head of the  
LL. Simply call the `deleteHead()` function



```

1  Node* removeKthNode(Node* head, int k) {
2      // if LL is empty or has only one element,
3      // return null
4      if (head == nullptr || head->next == nullptr)
5          return nullptr;
6
7      // if head is to be removed
8      if (k == 1) {
9          return deleteHead(head);
10     }
11
12     Node* curNode = head;
13     for (int i = 2; i < k; ++i) {
14         curNode = curNode->next;
15     }
16
17     Node* kthNode = curNode->next;
18     curNode->next = kthNode->next;
19
20     kthNode->next = nullptr;
21     delete kthNode;
22
23     return head;
24 }

```

## 2.3 Stacks

A stack is a linear data structure that follows the Last In, First Out (LIFO) principle. This means that the last element added to the stack will be the first one to be removed.

### 2.3.1 Implementing a stack using arrays

- `push()` : Pushes an element onto the stack
- `pop()` : Removes the topmost element from the stack *and* returns it
- `top()` : Only returns the topmost element from the stack
- `isEmpty()` : Returns *true* if stack is empty else *false*

### 2.3.2 Procedure

- 1.

## ArrayStack class

```
class ArrayStack {
private:
    int topIndex;
    int maxSize;
    int* arr;

public:
    ArrayStack(int size = 1000) {
        maxSize = size;
        arr = new int[maxSize];
        topIndex = -1; // denotes empty stack
    }

    // Destructor to free memory if object is not referenced anymore
    ~ArrayStack() {
        delete[] arr;
    }

    void push(int x) {
        if (topIndex + 1 >= maxSize) {
            std::cout << "Stack overflow" << std::endl;
            return;
        }
        arr[++topIndex] = x;
    }

    int pop() {
        if (isEmpty()) {
            std::cout << "Stack is empty" << std::endl;
            return -1;
        }
        return arr[topIndex--];
    }

    int top() {
        if (isEmpty()) {
            std::cout << "Stack is empty" << std::endl;
            return -1;
        }
        return arr[topIndex];
    }

    bool isEmpty() {
        return topIndex == -1;
    }
};
```

### 2.3.3 FAQs

#### 1. Next Greater Element

Given an array `arr` of size `n` containing elements, find the next greater element for each element in the array in the order of their appearance. The next greater element of an element in the array is the nearest element on the right that is greater than the current element. If there does not exist a next greater element for the current element, then the next greater element for that element is `-1`.

Example Input:





---

# CHAPTER 3

---

## SORTING ALGORITHMS

### 3.1 Bubble Sort

**Description:** Repeatedly swaps adjacent elements if they are in the wrong order.

```
void bubbleSort(int arr[], int n) {
    for (int i = 0; i < n - 1; i++) {
        for (int j = 0; j < n - i - 1; j++) {
            if (arr[j] > arr[j + 1])
                swap(arr[j], arr[j + 1]);
        }
    }
}
```

- Time Complexity:  $\mathcal{O}(n^2)$
- Space Complexity:  $\mathcal{O}(1)$

### 3.2 Quick Sort

**Description:** A pivot is chosen which is then placed in its correct place (the pivot's position in the sorted array, called the *partition index*). This step is then repeated for left and right portion of the pivot recursively.

1. **Recursion:** Involves recursively calling the helper function

```
void quickSortHelper(int arr[], int low, int high) {
    if (low < high) {
        int pIndex = partition(arr, low, high);
        quickSortHelper(arr, low, pIndex - 1);
        quickSortHelper(arr, pIndex + 1, high);
    }
}
```

2. **Partition:** Function that partitions the array and places the pivot in its correct place

```
int partition(int arr[], int low, int high) {
    int pivot = arr[high];
    int i = low - 1;
    for (int j = low; j < high; ++j) {
        if (arr[j] <= pivot) {
            swap(arr[i + 1], arr[j]);
            i = i + 1;
        }
    }
    swap(arr[i + 1], arr[high]);
    return (i + 1);
}
```

- **Time Complexity:**
  - Best Case =  $\mathcal{O}(n \log n)$
  - Average Case =  $\mathcal{O}(n \log n)$
  - Worst Case =  $\mathcal{O}(n^2)$
- **Space Complexity:**  $\mathcal{O}(1)$