Data Structures and Algorithms Notes

Soumik Seth

November 14, 2024

CONTENTS

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> v`$_2$`(v`$_1$`);` creates a vector containing all elements of v$_1$ 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.
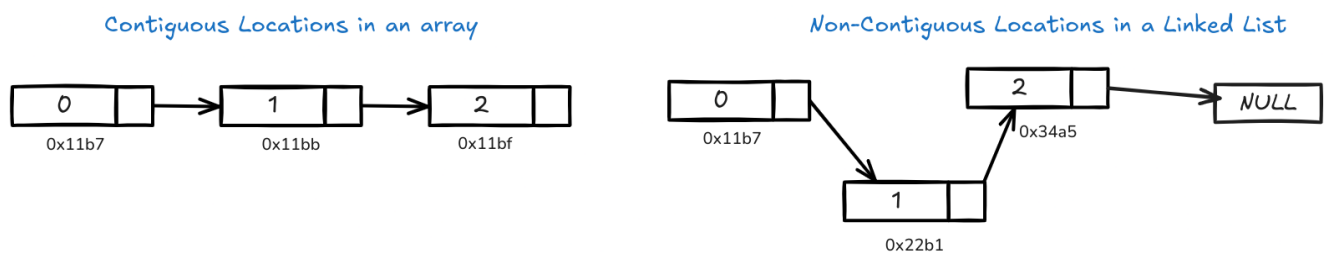
Figure 2.1: Array v/s Linked List

2.2.1   Creating a linked list

```cpp
#include <bits/stdc++.h>
using namespace std;

struct Node {
    int data;
    Node* next;
    Node(int val) : data(val), next(nullptr) {}
};

int main() {
    Node* head = new Node(1); // Create LL with head 1
    head->next = new Node(2);
    head->next->next = new Node(3);
    head->next->next->next = new Node(4);
    head->next->next->next->next = new Node(5);

    // create dummy node to travel the linked list
    Node* temp = head;
    while (temp != nullptr) {
        cout << temp->data << ' ';
        temp = temp->next; // go to the next node
    }

    return 0;
}
```

The above code produces a linked list that looks like this:
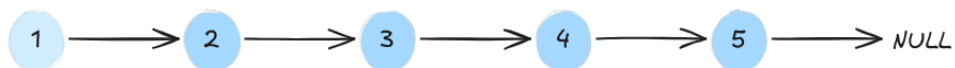
2.2.2   Types of linked lists

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.
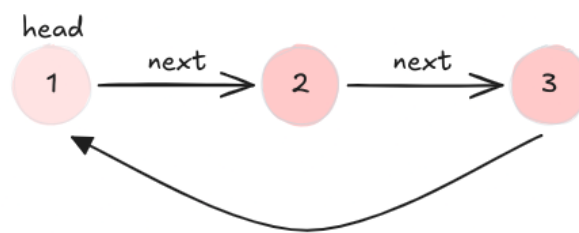
Doubly Linked Lists

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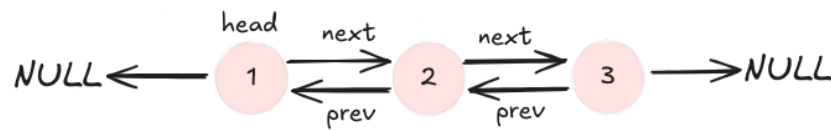
Circular Linked Lists

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

```cpp
Node* arrayToLinkedList(int arr[], int size) {
    if (size == 0) {
        return nullptr; // empty linked list
    }

    Node* head = new Node(arr[0]); // create head of linked list

    Node* curNode = head; // node to travel each node as it is added
    for (int i = 1; i < size; ++i) { // start from i = 1, as head = arr[0]
        curNode->next = new Node(arr[i]);
        curNode = curNode->next; // go to the new node just created
    }

    return head;
}
```

2.2.4   Find length of linked list

```cpp
int lengthOfLinkedList(Node* head) {
    int length = 0;
    Node* curNode = head;

    while (curNode != nullptr) {
        length++;
        curNode = curNode->next;
    }

    return length;
}
```

2.2.5   Delete head of linked list

To delete head, simply advance the head by one node

(and free the original head memory)

head

advance head

1 → 2 → 3 → 4

head

1 ✗→ 2 → 3 → 4
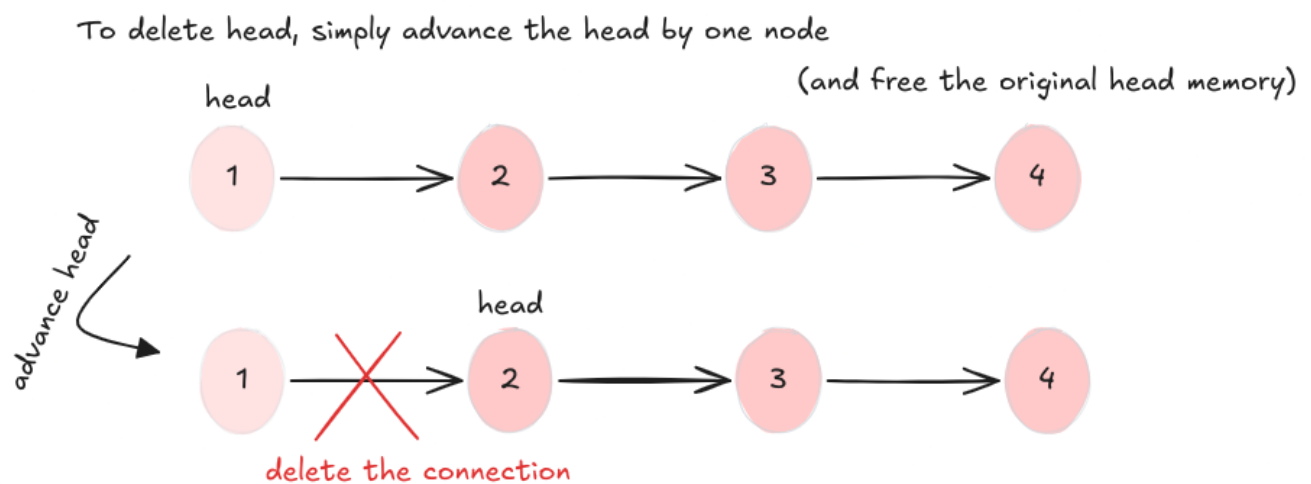
delete the connection

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 Kth 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

head

1 → 2 → 3 → 4 → 5 → NULL

k = 3

head

1 → 2 ✗ 3 ✗ 4 → 5 → NULL

Change links
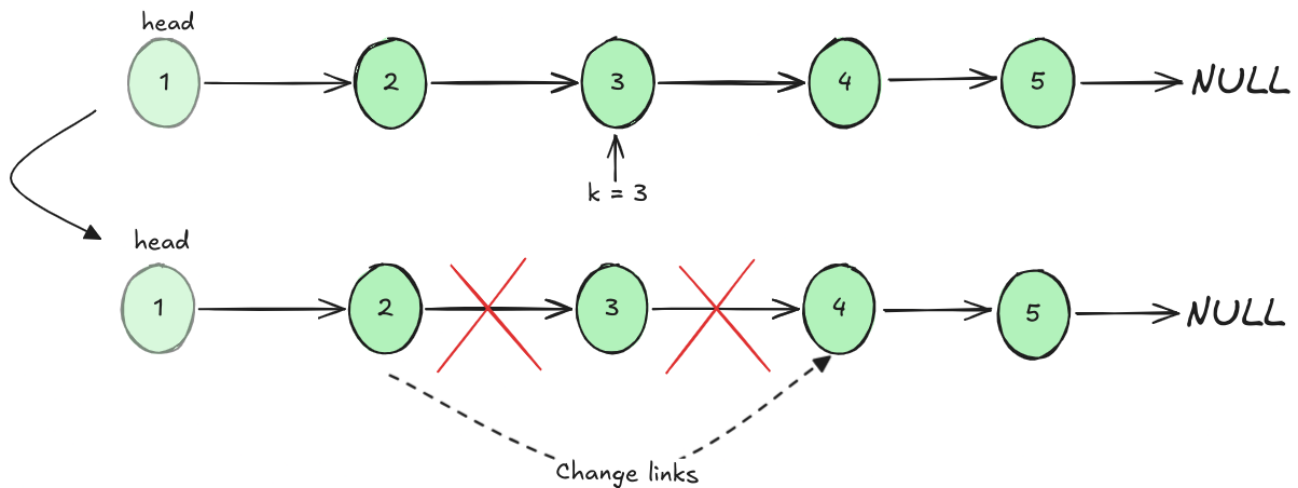
```cpp
Node* removeKthNode(Node* head, int k) {
    // if LL is empty or has only one element,
    // return null
    if (head == nullptr || head->next == nullptr)
            return nullptr;

    // if head is to be removed
    if (k == 1) {
        return deleteHead(head);
    }

    Node* curNode = head;
    for (int i = 2; i < k; ++i) {
        curNode = curNode->next;
    }

    Node* kthNode = curNode->next;
    curNode->next = kthNode->next;

    kthNode->next = nullptr;
    delete kthNode;

    return head;
}
```

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

```cpp
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 ow" << 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**: $\mathcal{O}\left(n^2\right)$
- **Space Complexity**: $\mathcal{O}\left(1\right)$

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 = $\mathcal{O}\left(n \log n\right)$
  - Average Case = $\mathcal{O}\left(n \log n\right)$
  - Worst Case = $\mathcal{O}\left(n^2\right)$

- **Space Complexity**: $\mathcal{O}\left(1\right)$