Notes1(SinglyLL).txt
Document

Beautify these notes and add some useful points too if you want

Here's a beautified and expanded version of the notes on singly linked lists in C++. Additional explanations and tips have been added to enhance understanding.

Singly Linked Lists in C++

Introduction

In C++, a linked list is a data structure used to store a sequence of elements. Unlike arrays, which store elements in contiguous memory locations, a linked list consists of nodes where each node contains:

- A data element.
- A pointer (or reference) to the next node in the sequence.

This structure allows for efficient insertion and deletion of elements, as it doesn't require shifting other elements around, unlike arrays.

Basic Node Structure

Definition

Here's a basic definition of a node in a singly linked list:

cpp

```cpp
struct Node {
    int data;      // The data element
    Node* next;    // Pointer to the next node

    Node(int data1, Node* next1 = nullptr) {
        data = data1;
        next = next1;
    }
};
```

Creating and Using Nodes

Creating a Node

```cpp
Node x = Node(2, nullptr); // Created a variable of type Node
```

Getting Address of a Node

```cpp
Node* y = &x; // Getting the address of the node 'x'
```

Creating a Node Dynamically

```cpp
Node* y = new Node(2, nullptr); // Allocates memory and returns a pointer to the new Node
```

Accessing Node Data

For a dynamically allocated node (`` `Node*` ``):

```cpp
y->data; // Accessing the data of the node 'y'
y->next; // Accessing the next pointer of the node 'y'
```

For a statically allocated node (`` `Node` ``):

```cpp
x.data; // Accessing the data of the node 'x'
x.next; // Accessing the next pointer of the node 'x'
```

Example Code

Basic Node Usage

```cpp
#include <iostream>
using namespace std;

struct Node {
    int data;
    Node* next;

    Node(int data1, Node* next1 = nullptr) {
        data = data1;
        next = next1;
    }
};

int main() {
    Node x = Node(2, nullptr); // Initializing a Node
    Node* y = &x; // Getting the address of 'x'

    Node* m = new Node(3, nullptr); // Dynamically allocating a new Node
    cout << m->data << endl; // Output: 3

    // Cleaning up dynamically allocated memory
    delete m;

    return 0;
}
```

Converting an Array to a Linked List

To convert an array into a linked list, we can create a function that iterates through the array and creates linked list nodes for each element.

Conversion Code

```cpp
#include <iostream>
#include <vector>
using namespace std;

class Node {
public:
    int data;
    Node* next;

    Node(int data1, Node* next1 = nullptr) {
        data = data1;
        next = next1;
    }
};

Node* ConvertArrtoLL(vector<int>& arr) {
    if (arr.empty()) return nullptr;

    Node* head = new Node(arr[0]);
    Node* mover = head;
    for (int i = 1; i < arr.size(); i++) {
        Node* temp = new Node(arr[i]);
        mover->next = temp;
        mover = temp;
    }
    return head;
}

int main() {
    vector<int> arr = {2, 4, 5, 6, 7};
    Node* head = ConvertArrtoLL(arr);

    // Print the linked list to verify
    Node* temp = head;
    while (temp) {
        cout << temp->data << " ";
        temp = temp->next;
    }
    cout << endl;
    // Free the dynamically allocated memory
    while (head) {
        Node* toDelete = head;
        head = head->next;
        delete toDelete;
    }

    return 0;
}
```

Traversing a Linked List

To traverse a linked list, you start from the head and follow the `next` pointers until you reach a `nullptr`.

Traversal Code

```cpp
#include <iostream>
using namespace std;

class Node {
public:
    int data;
    Node* next;

    Node(int data1, Node* next1 = nullptr) {
        data = data1;
        next = next1;
    }
};

Node* ConvertArrtoLL(vector<int>& arr) {
    if (arr.empty()) return nullptr;

    Node* head = new Node(arr[0]);
    Node* mover = head;
    for (int i = 1; i < arr.size(); i++) {
        Node* temp = new Node(arr[i]);
        mover->next = temp;
        mover = temp;
    }
    return head;
}

int main() {
    vector<int> arr = {2, 3, 4, 6, 7};
    Node* head = ConvertArrtoLL(arr);
    Node* temp = head;

    while (temp) {
        cout << temp->data << " "; // Output: 2 3 4 6 7
        temp = temp->next;
    }
    cout << endl;

    // Free the dynamically allocated memory
    while (head) {
        Node* toDelete = head;
        head = head->next;
        delete toDelete;
    }
    return 0;
}
```

Calculating the Length of a Linked List

To find the length of a linked list, you can traverse the list and count the number of nodes.

Length Calculation Code

```cpp
int LengthofLL(Node* head) {
    int count = 0;
    Node* temp = head;
    while (temp) {
        count++;
        temp = temp->next;
    }
    return count;
}
```

Deleting Elements in a Linked List

Deleting the Head Node

```cpp
Node* deleteHead(Node* head) {
    if (head == nullptr) return nullptr;
    Node* temp = head;
    head = head->next;
    delete temp; // Free the old head node
    return head;
}
```

Deleting the Tail Node

```cpp
Node* DeleteTail(Node* head) {
    if (head == nullptr || head->next == nullptr) return nullptr;

    Node* temp = head;
    while (temp->next->next != nullptr) {
        temp = temp->next;
    }
    delete temp->next; // Delete the tail node
    temp->next = nullptr;
    return head;
}
```

Deleting the K-th Node

To delete the k-th node, you first need to traverse to the (k-1)-th node, then update its `next` pointer to skip the k-th node.

```cpp
Node* DeleteK(Node* head, int k) {
    if (head == nullptr) return nullptr;
    if (k == 1) { // Special case: delete head
        Node* temp = head;
        head = head->next;
        delete temp;
        return head;
    }

    Node* temp = head;
    for (int i = 1; temp != nullptr && i < k - 1; i++) {
        temp = temp->next;
    }
    if (temp == nullptr || temp->next == nullptr) return head;

    Node* nodeToDelete = temp->next;
    temp->next = nodeToDelete->next;
    delete nodeToDelete;

    return head;
}
```

Inserting Elements in a Linked List

Inserting at the Head

```cpp
Node* InsertHead(int val, Node* head) {
    return new Node(val, head);
}
```

Inserting at the K-th Position

To insert an element at the k-th position, you traverse to the (k-1)-th node and update the pointers accordingly.

```cpp
Node* InsertAtK(Node* head, int val, int k) {
    if (k == 1) {
        return new Node(val, head);
    }

    Node* temp = head;
    for (int i = 1; temp != nullptr && i < k - 1; i++) {
        temp = temp->next;
    }
    if (temp == nullptr) return head; // Position out of bounds

    Node* newNode = new Node(val, temp->next);
    temp->next = newNode;
```

```
        return head;
    }
```

Additional Tips:

1. **Memory Management**: Always ensure you free dynamically allocated memory to avoid memory leaks.
2. **Edge Cases**: Consider edge cases such as inserting or deleting nodes in an empty list.
3. **Code Modularity**: Break down your linked list operations into smaller, reusable functions.
4. **Error Handling**: Add error checks to handle operations on `nullptr` gracefully.

This beautified version of the notes should help in understanding the basics and some advanced operations of singly linked lists in C++.

ChatGPT can make mistakes. Check important info.