Lec 17 Linked List

1. 什么是链表?

概念： 链表是一种**线性**数据结构，但与数组不同，它在内存中的存储**不必是连续的**。它由一系列**节点 (Node)** 组成。每个节点包含两部分：

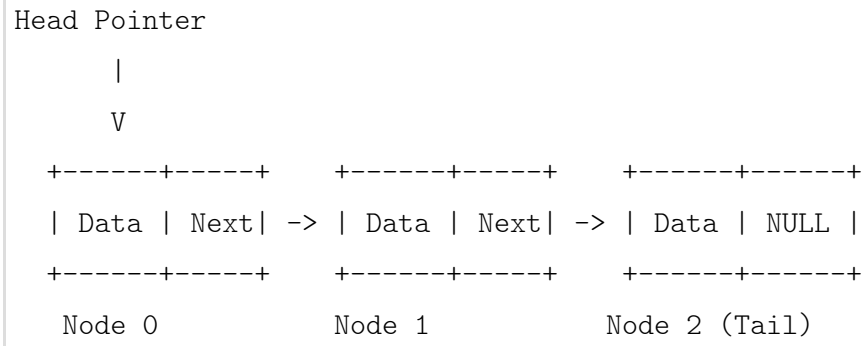
1. **数据域 (Data)**：存储实际的数据信息，可以是任何类型（整数、浮点数、字符、结构体等）。
2. **指针域 (Pointer/Next)**：存储指向链表中**下一个节点**的内存地址。

关键术语：

- **节点 (Node)**：链表的基本组成单元，包含数据和指向下一个节点的指针。
- **头指针 (Head Pointer)**：一个指向链表**第一个节点**的指针。它是访问整个链表的入口。如果链表为空，头指针为 `NULL`。
- **尾节点 (Tail Node)**：链表的**最后一个节点**。它的指针域通常指向 `NULL`，表示链表的结束。
- **NULL**：一个特殊的指针值，表示指针不指向任何有效的内存地址。用于标记链表末尾或空链表。

可视化： 可以想象链表像一列火车：

- 每个**车厢**就是一个**节点**。
- 车厢里装载的**货物**就是**数据域**。
- 连接两节车厢的**挂钩**就是**指针域**，指向下一节车厢。
- **火车头**（或者说指向火车头的指示牌）就是**头指针**。
- 最后一节车厢的**尾部**（没有挂钩指向后面）就是**尾节点**，其指针域为 `NULL`。



2. 链表 vs. 数组

这是数据结构中一个经典的对比：

特性	数组(Array)	链表(Linked List)
内存分配	静态（编译时）或动态（运行时）	通常动态（运行时 `malloc/new`）
内存结构	连续	不一定连续
大小	固定（静态）或可调整但可能耗时	**动态**，可轻松增长或缩减
访问元素	**随机访问** (O(1)) - 通过索引直接访问	**顺序访问** (O(n)) - 必须从头遍历
插入/删除元素	慢 (O(n)) - 可能需要移动大量元素	**快** (O(1)) - 如果知道位置，只需改指针
查找元素	取决于是否有序（无序O(n)，有序O(log n)）	慢 (O(n)) - 必须顺序遍历
额外开销	无（或只有少量元数据）	每个节点都需要额外空间存指针
缓存局部性	好（连续内存）	差（节点分散）

总结：

- 需要**频繁插入/删除**，且**不关心随机访问**？ -> **链表**更优。
- 需要**快速随机访问**元素？ -> **数组**更优。
- 不确定需要多少空间，希望**动态调整**？ -> **链表**更灵活。

3. C 语言实现链表

在 C 语言中，“定义”一个链表主要包含两个步骤：

1. **定义链表节点的数据结构 (Node Structure)**：你需要指定每个节点包含哪些信息（数据域）以及如何指向下一个节点（指针域）。
2. **定义指向链表开头的指针 (Head Pointer)**：这个指针将作为访问整个链表的入口。

3.1 定义链表

步骤 1: 定义节点结构 (Node Structure)

使用 `struct` 关键字来定义一个结构体，代表链表中的一个节点。这个结构体内部必须包含：

- 至少一个成员来存储数据（数据域）。

- 一个指向**同类型结构体**的指针（指针域），通常命名为 `next`。

```
#include <stdio.h>
#include <stdlib.h> // 为了使用 malloc 和 free
#include <string.h> // 为了使用 strcpy（如果数据域包含字符串）

// 定义节点结构体
typedef struct Node {
    int data;             // 数据域（这里用 int 举例）
    struct Node* next;    // 指针域，指向下一个同类型的节点
} Node;

// 也可以定义包含更复杂数据的节点，比如讲座中的学生记录：
typedef struct StudentNode {
    char *name;           // 指向动态分配的姓名字符串
    int uin;
    float gpa;
    struct StudentNode *next;
} StudentNode;
```

解释:

- `struct Node { ... };` 定义了一个名为 `Node` 的结构体蓝图。
- `int data;` 是这个节点存储的数据。
- `struct Node *next;` 是这个结构体的关键部分，它是一个指针，可以指向内存中的**另一个 `struct Node` 类型的实例**。正是这个 `next` 指针将节点链接起来。

步骤 2: 定义头指针 (Head Pointer)

链表本身通常由一个指向其第一个节点的指针来表示，即头指针。

- 声明一个**指向节点结构类型**的指针变量。
- 对于一个**空链表**，这个头指针必须初始化为 `NULL`。

```
int main() {
    // 2. 定义头指针，并初始化为空链表
    Node *head = NULL; // 使用 typedef 后的简洁写法
    // 或者: struct Node *head = NULL; // 不使用 typedef 的写法

    // 现在 'head' 就代表（或指向）一个链表了。
    // 目前它是一个空链表，因为 head 指向 NULL。
```

```
    // 后续的操作（如添加节点）会修改 head 指针
    // 或者修改 head 指向的节点的 next 指针。

    // ...（后续添加、删除、遍历链表的代码）

    return 0;
}
```

3.2 动态内存分配 (malloc / free)

链表的节点通常在**堆 (Heap)** 上动态创建，因为我们事先不知道需要多少节点。

- **malloc(size_t size)**：分配指定字节大小的内存块，返回一个指向该内存块起始位置的 void* 指针。如果分配失败，返回 NULL。
- **free(void *ptr)**：释放之前由 malloc（或 calloc, realloc）分配的内存。

创建新节点的过程：

```
// 函数：创建一个包含指定数据的新节点
Node* createNode(int dataValue) {
    // 1. 分配内存：为新节点请求足够的空间
    Node* newNode = (Node*)malloc(sizeof(Node)); // sizeof(Node) 计算结构体大小

    // 2. 检查分配是否成功：！！！非常重要！！！
    if (newNode == NULL) {
        perror("Memory allocation failed for new node"); // perror 打印错误信息
        return NULL; // 返回 NULL 表示失败
    }

    // 3. 初始化节点：设置数据域和指针域
    newNode->data = dataValue; // 存储数据
    newNode->next = NULL;      // 新节点暂时不指向任何节点

    // 4. 返回新创建节点的指针
    return newNode;
}
```

释放节点内存：

```
// 释放单个节点的内存
```

```
void freeNode(Node* node) {
    if (node != NULL) {
        free(node);
    }
}

// 如果节点内有动态分配的数据（如 StudentNode 的 name），需要先释放内部数据
void freeStudentNode(StudentNode* node) {
    if (node != NULL) {
        free(node->name); // 先释放动态分配的 name 字符串
        free(node);       // 再释放节点本身
    }
}
```

3.3 指针的角色（特别是双重指针 **）

- **单指针 (Node*)**：
  - 通常用于指向链表的**头节点** (`Node* head = NULL;`)。
  - 用于在链表中**遍历** (`Node* current = head;`)。
  - 作为函数参数传递时，如果函数**只需要读取**链表内容或**不修改头指针本身**，使用单指针即可。
- **双重指针 (Node**)**：
  - 当你需要在**函数内部修改调用者作用域中的头指针**时，必须使用双重指针。
  - 场景：
    - 在链表**头部插入**新节点（头指针需要指向新节点）。
    - **删除头节点**（头指针需要指向原头节点的下一个节点）。
    - **完全清空**链表（最终头指针需要变回 `NULL`）。
    - 某些**尾部插入**的实现方式（如果链表可能为空，需要修改头指针）。
  - **原理**：传递 `Node** head_ptr` 意味着传递了**头指针变量自身的内存地址**。在函数内部通过 `*head_ptr` 可以解引用得到头指针，并对其进行修改，这个修改会影响到原始的头指针变量。
  - **示例**：`int AddPerson(Person **ourList, ...)`

4. 核心链表操作

4.1 遍历链表 (Traversal)

访问链表中的每一个节点，通常用于打印、查找或计算。

迭代法 (Iterative):

```
// 函数：打印链表中的所有数据
void printList(Node* head) {
    Node* current = head; // 从头节点开始

    printf("List: ");
    while (current != NULL) { // 只要当前节点不是 NULL
        printf("%d -> ", current->data); // 打印当前节点数据
        current = current->next;        // 移动到下一个节点
    }
    printf("NULL\n"); // 打印链表结束标记
}
```

递归法 (Recursive):

```
// 函数：递归打印链表
void printListRecursive(Node* head) {
    // 基本情况 (Base Case)：如果链表为空或到达末尾，停止
    if (head == NULL) {
        printf("NULL\n");
        return;
    }
    // 递归步骤 (Recursive Step)：
    printf("%d -> ", head->data);  // 打印当前节点
    printListRecursive(head->next); // 递归调用处理剩余部分
}
```

4.2 插入节点 (Insertion)

a) 头插法 (Insert at Head): 在链表的最前面插入新节点。新节点将成为新的头节点。

```
// 函数：在链表头部插入一个新节点
// 使用双重指针 **headRef 因为需要修改原始的 head 指针
void insertAtHead(Node** headRef, int dataValue) {
    // 1. 创建新节点
    Node* newNode = createNode(dataValue);
```

```
    if (newNode == NULL) {
        return; // 创建失败，直接返回
    }

    // 2. 将新节点的 next 指向当前的头节点
    newNode->next = *headRef; // *headRef 是当前的 head 指针

    // 3. 更新头指针，使其指向新节点
    *headRef = newNode; // 修改原始的 head 指针
}
```

b) 尾插法 (Insert at Tail): 在链表的末尾添加新节点。

```
// 函数：在链表尾部插入一个新节点
void insertAtTail(Node** headRef, int dataValue) {
    // 1. 创建新节点
    Node* newNode = createNode(dataValue);
    if (newNode == NULL) {
        return;
    }
    // newNode->next 已经是 NULL 了 (createNode 中设置的)

    // 2. 如果链表为空，新节点就是头节点
    if (*headRef == NULL) {
        *headRef = newNode;
        return;
    }

    // 3. 如果链表不为空，找到尾节点
    Node* last = *headRef;
    while (last->next != NULL) { // 遍历直到找到最后一个节点 (next 为 NULL)
        last = last->next;
    }

    // 4. 将原尾节点的 next 指向新节点
    last->next = newNode;
    [source: 66] // 对应迭代实现
}
```

c) 在指定位置插入 (Insert at Position / Sorted Insert):

```c
// 函数：在指定位置（索引）之后插入节点（假设索引从0开始）
// （为简化，未完全实现错误检查）
void insertAfter(Node* prevNode, int dataValue) {
    if (prevNode == NULL) {
        printf("Previous node cannot be NULL\n");
        return;
    }

    Node* newNode = createNode(dataValue);
    if (newNode == NULL) return;

    newNode->next = prevNode->next; // 新节点指向 prevNode 原来的下一个节点
    prevNode->next = newNode;       // prevNode 指向新节点
}

// 函数：在有序链表（升序）中插入节点，保持有序性
void insertSorted(Node** headRef, int dataValue) {
    Node* newNode = createNode(dataValue);
    if (newNode == NULL) return;

    Node* current = *headRef;
    Node* prev = NULL;

    // 找到插入位置（第一个大于等于 dataValue 的节点之前）
    while (current != NULL && current->data < dataValue) {
        prev = current;
        current = current->next;
    }

    // 情况1：在头部插入（链表为空或新值最小）
    if (prev == NULL) {
        newNode->next = *headRef;
        *headRef = newNode;
    }
    // 情况2：在中间或尾部插入
    else {
        newNode->next = current; // 或者 newNode->next = prev->next;
        prev->next = newNode;
    }
     [source: 85, 86] // 对应有序插入逻辑
```

```
}
```

4.3 删除节点 (Deletion)

a) 删除头节点 (Delete from Head):

```
// 函数：删除头节点
// 返回被删除节点的数据（可选），如果链表为空返回特定值（如 -1）
int deleteFromHead(Node** headRef) {
    // 1. 检查链表是否为空
    if (*headRef == NULL) {
        printf("List is empty, cannot delete from head.\n");
        return -1; // 或其他错误指示
    }

    // 2. 保存当前头节点到临时指针
    Node* tempNode = *headRef;
    int deletedData = tempNode->data; // 保存数据（可选）

    // 3. 将头指针移动到下一个节点
    *headRef = (*headRef)->next; // *headRef = tempNode->next;

    // 4. 释放原头节点的内存
    free(tempNode);

    return deletedData;
}
```

b) 删除具有特定值的节点 (Delete by Value):

```
// 函数：删除链表中第一个具有特定值的节点
void deleteNodeByValue(Node** headRef, int valueToDelete) {
    Node* current = *headRef;
    Node* prev = NULL;

    // 1. 查找要删除的节点，同时跟踪前一个节点
    while (current != NULL && current->data != valueToDelete) {
        prev = current;
        current = current->next;
    }
```

```
    // 2. 如果没找到
    if (current == NULL) {
        printf("Value %d not found in the list.\n", valueToDelete);
        return;
    }

    // 3. 如果要删除的是头节点
    if (prev == NULL) { // or current == *headRef
        *headRef = current->next; // 移动头指针
    }
    // 4. 如果要删除的是中间或尾部节点
    else {
        prev->next = current->next; // 让前一个节点指向当前节点的下一个节点，跳过当前节点
    }

    // 5. 释放找到的节点的内存
    free(current);
}
```

4.4 查找节点 (Searching)

```
// 函数：查找链表中是否包含特定值
// 返回指向找到的第一个节点的指针，如果没找到返回 NULL
Node* findNode(Node* head, int valueToFind) {
    Node* current = head;
    while (current != NULL) {
        if (current->data == valueToFind) {
            return current; // 找到了，返回节点指针
        }
        current = current->next;
    }
    return NULL; // 遍历完没找到，返回 NULL
}
[source: 83] // 对应查找逻辑
```

4.5 释放整个链表 (Freeing the Entire List)

非常重要，防止内存泄漏！

```
// 函数：释放链表所有节点的内存
```

```
void freeList(Node** headRef) {
    Node* current = *headRef;
    Node* nextNode;

    while (current != NULL) {
        nextNode = current->next; // **先保存下一个节点的指针**
        free(current);            // 释放当前节点
        current = nextNode;       // 移动到下一个节点
    }

    // 最后，将原始头指针设为 NULL，表示链表已空
    *headRef = NULL;
}

// 对于 StudentNode，需要先释放内部 name
void freeStudentList(StudentNode** headRef) {
    StudentNode* current = *headRef;
    StudentNode* nextNode;

    while (current != NULL) {
        nextNode = current->next;
        free(current->name);  // 先释放 name
        free(current);        // 再释放节点
        current = nextNode;
    }
    *headRef = NULL;
     [source: 68, 81] // 对应释放逻辑
}
```

5. 关键注意事项

1. **内存管理**：始终记得 `free` 掉 `malloc` 分配的内存。对于每个 `malloc` 都应该有一个对应的 `free`。忘记 `free` 会导致**内存泄漏**。
2. **NULL 指针检查**：在访问节点的 `data` 或 `next` 之前，务必检查该节点指针是否为 `NULL`。访问 `NULL` 指针的成员会导致**段错误 (Segmentation Fault)**。特别是 `malloc` 的返回值检查！
3. **指针操作**：修改 `next` 指针时要格外小心，错误的修改会**破坏链表结构**（丢失后续节点）。使用临时指针 (`temp`) 存储节点地址通常是个好习惯，尤其是在删除操作中。
4. **双重指针**：理解何时以及为何需要使用双重指针 (`Node**`) 来修改函数外部的头指针。

5. 边界条件：测试代码时要考虑各种边界情况：
    - 空链表 (`head == NULL`)。
    - 只有一个节点的链表。
    - 操作发生在链表头部。
    - 操作发生在链表尾部。