Lec 18 Problem solving with linked lists

链表数据结构 (Linked List Data Structure)

1. 概念解读:

- **定义:** 链表是一种有序的节点集合，每个节点包含一些数据，并通过指针连接起来 。它是一种动态数据结构，大小可以在运行时改变。
- 节点 (Node):

  链表的基本单元，通常包含两部分：
  - **数据域 (Data):** 存储实际信息 (例如学生信息、数字等)。
  - **指针域 (Pointer/Next):** 存储指向链表中下一个节点的内存地址 。
- **头指针 (Head Pointer):** 指向链表中第一个节点的指针 。它是访问整个链表的入口。如果链表为空，头指针通常为 `NULL`。
- **尾节点 (Tail Node):** 链表中的最后一个节点 。其指针域通常指向 `NULL`，表示链表的结束 。

图示:

```
Head Pointer ---> [Node 0 | ptr] ---> [Node 1 | ptr] ---> [Node 2 | NULL]
                    ^                 ^                 ^
                  Data域            Data域            Data域
```

链表遍历 (Traversal)

1. 概念解读:

- 遍历是指按照节点的连接顺序访问链表中的每一个节点。通常从头节点开始，沿着 `next` 指针逐个访问，直到遇到 `NULL` 指针。

2. 代码分析 (`print_data` 函数):

```
// 假设 student 结构体已定义，包含 name, UIN, GPA, next 等字段
void print_data(student *head)
{
    printf("Name     UIN      GPA\n");
    // 使用 head 作为临时指针进行遍历
    while(head) // 等价于 while(head != NULL)
    {
        // 打印当前节点的数据
        printf("%s     %d     %f\n", head->name, head->UIN, head->GPA);
        // 移动到下一个节点
        head = head->next; // 将 head 指向下一个节点的地址
    }
}
```

- **参数:** `student *head` - 接收链表的头指针。注意，这里接收的是头指针的 *值*(第一个节点的地址)，而不是头指针本身的地址。因此，函数内部对 `head` 的修改 (`head = head->next;`) 不会影响 `main` 函数中的原始 `headptr`。
- **循环条件:** `while(head)` - 只要当前节点指针 `head` 不是 `NULL`，就继续循环。
- **访问数据:** `head->name`, `head->UIN`, `head->GPA` - 使用 `->` 操作符访问当前节点的数据成员。
- **移动指针:** `head = head->next;` - 这是遍历的核心。将 `head` 更新为其指向节点的 `next` 成员的值，即下一个节点的地址。

3. 流程理解:

1. 函数接收头节点地址。
2. 检查地址是否为 `NULL`。
3. 如果不为 `NULL`，打印当前节点信息。
4. 将当前指针更新为当前节点的 `next` 指针的值。
5. 重复步骤 2-4，直到当前指针为 `NULL`。

在链表尾部插入节点 (Insertion at Tail)

1. 概念解读:

- 在链表末尾添加一个新节点。需要先遍历到链表的最后一个节点，然后修改其 `next` 指针指向新创建的节点。
- **特殊情况:** 如果链表为空 (头指针为 `NULL`)，则新节点直接成为头节点。
- **指针修改:** 这个操作 *可能* 需要修改头指针（当链表为空时），或者修改最后一个节点的 `next` 指针。为了能在函数内部修改调用者（如 `main` 函数）的头指针变量，或者修改链表中某个节点的 `next` 指针成员，函数需要接收指针的地址，即使用 **双重指针 (Double Pointer)**。

2. 代码分析 (`insert_tail` 函数):

```
// 假设 student 结构体已定义
void insert_tail (student **head, student *data)
// **head: 指向头指针变量地址的指针 (e.g., &headptr in main)
// *data:  指向包含新学生数据的临时变量地址 (e.g. &temp in main)
{
    // 遍历阶段：找到指向最后一个节点 next 指针的地址
    while(*head) // 检查 *head (即 headptr 的值) 是否为 NULL
    {
        // 关键步骤：更新 head (局部变量) 使其指向当前节点 next 指针的地址
        head = &((*head)->next);
        // *head: 当前节点地址
        // (*head)->next: 当前节点 next 指针的值 (下一节点地址或 NULL)
        // &((*head)->next): 当前节点 next 指针成员自身的地址
    }

    // 插入阶段：创建新节点并链接
    // 此时 *head 是 NULL，而 head 指向最后一个节点 next 指针的地址 (或原始 headptr 的地址，如果链表为空)
    student *tmp = (student*)malloc(sizeof(student)); // 分配新节点内存
    *tmp = *data;  // 复制数据 (浅拷贝，指针成员 name 只复制地址)
                   // 注意：在 GDB 示例中，temp.name 是独立 malloc 的，所以这里复制的是指向 "LMNO" 的指针
                   // 如果 data->name 没有单独分配内存，这里需要深拷贝

    // 关键链接步骤：修改指针，将新节点链接到链表末尾
    *head = tmp; // 将 head 指向的地址 (最后一个节点的 next 或 headptr) 的内容设置为新节点 tmp 的地址
}
```

- 双重指针 `student **head`:
  - `head`: (函数内的局部变量) 存储的是 *指向链表节点的指针* 的地址。例如，在 `main` 中调用 `insert_tail(&headptr, ...)` 时，`head` 的值是 `&headptr`。
  - `*head`: 解引用一次，得到 `head` 所指向地址的内容，即 *链表节点的地址*。例如，`*head` 的值相当于 `headptr` 的值（第一个节点的地址，或者 `NULL`）。
  - `**head`: 解引用两次，得到 *链表节点的内容*(结构体本身)。例如，`**head` 相当于 `*headptr`。
- `while(*head)` 循环:
  - 条件 `*head` 检查当前指向的节点是否存在。
  - `head = &((*head)->next);` 是精髓所在。它不移动节点指针 `*head`，而是更新 `head` 这个双重指针，使其指向 *当前节点的 `next` 指针成员的内存地址*。当循环结束时，`*head` 为 `NULL`，而 `head` 正好指向需要被修改以链接新节点的那个指针（即最后一个节点的 `next` 成员，或者 `main` 中的 `headptr` 变量本身）。
- `malloc` 和数据复制:

- `malloc(sizeof(student))` 分配足够存储一个 `student` 结构体的内存。
- `*tmp = *data;` 是结构体赋值。它将 `data` 指向的结构体内容按成员逐一复制到 `tmp` 指向的内存中。对于指针成员（如 `name`），它只复制指针的值（地址），而不是指针指向的内容。这称为浅拷贝。如果 `data->name` 指向的内存在 `insert_tail` 返回后被释放或修改，`tmp->name` 也会受影响。在 GDB 示例中，`temp.name` 在 `main` 中分配了独立内存，所以没问题。

- `head = tmp;` 这是最终的链接步骤。`head` 此时指向最后一个节点的 `next` 成员的地址（或 `headptr` 的地址）。`*head = tmp` 将这个地址的内容修改为新分配节点 `tmp` 的地址。

3. GDB 调试流程详解:

- **初始状态 (main):** `headptr` 指向第一个节点 ("ghik")。`temp` 包含新数据 ("LMNO")。调用 `insert_tail(&headptr, &temp)`。
- **进入 `insert_tail`:** `head` 的值为 `&headptr`。`*head` 的值为 `headptr` (节点 "ghik" 的地址)。`**head` 是节点 "ghik" 的内容。`data` 是 `&temp`。
- **第一次循环:** `*head` 非空。`head` 更新为 `&((*head)->next)`，即节点 "ghik" 的 `next` 成员的地址。此时 `*head` 是下一个节点 ("defg") 的地址。`**head` 是节点 "defg" 的内容。
- **后续循环:** 重复上述过程，`head` 不断指向当前节点 `next` 成员的地址，`*head` 指向下一个节点地址，直到 `*head` 为 `NULL`。
- **循环结束:** `*head` 为 `NULL`。`head` 指向最后一个节点 ("abcd"，假设) 的 `next` 成员的地址 (`0x5555555592d0`)。
- **分配新节点:** `malloc` 返回新内存地址给 `tmp` (`0x555555559810`)。
- **复制数据:** `*tmp = *data;` 将 `temp` 的内容复制到 `tmp`。`tmp->name` 指向 "LMNO"。
- **链接:** `*head = tmp;` 将地址 `0x5555555592d0` (最后一个节点 "abcd" 的 `next` 成员) 的内容从 `NULL` 修改为 `tmp` 的地址 (`0x555555559810`)。
- **返回 `main`:** `headptr` 仍然指向第一个节点 ("ghik")，但链表结构已被修改，最后一个节点的 `next` 现在指向新添加的 "LMNO" 节点。

4. 讨论:

- **为何不用临时指针遍历?** 如果像 `print_data` 那样用 `student *current = *head;` 然后 `while(current->next)` 遍历，最后 `current->next = tmp;`，在链表为空时无法处理，且代码更复杂。使用 `**head` 的方法更简洁统一。
- **直接赋值 `*head = data`?** 不可以。`data` 指向的是 `main` 函数栈上的临时变量 `temp`。函数返回后 `temp` 的内存会被回收，导致链表指向无效内存。必须 `malloc` 分配堆内存，并将数据复制过去。
- **`malloc` 的内存泄漏?** `insert_tail` 函数内部的 `tmp` 指针在函数返回后丢失。但由于 `*head = tmp;` 将 `tmp` 的地址连接到了链表中，这块内存可以通过链表访问，不会立即泄漏。但是，复制数据时 `*tmp = *data;` 中的 `tmp->name` 如果之前有 `malloc` (虽然示例中没有)，可能会导致 `tmp->name` 原来的内存泄漏。并且 `tmp->next` 被 `data->next` (NULL) 覆盖。正确的做法是先 `malloc`，然后逐个成员赋值，特别是 `name` 需要深拷贝（如果需要的话），并确保 `tmp->next = NULL;`。

- **函数返回值:** 可以让 `insert_tail` 返回新节点的地址 (`return tmp;`)，但这对于尾部插入不是必需的，因为调用者通常关心的是头指针。

其他链表操作

- 在已知节点前/后插入:
  - **之后插入 (Easier):** 找到已知节点 `known_node`。 `new_node->next = known_node->next; known_node->next = new_node;`
  - **之前插入 (Harder):** 需要找到已知节点的前一个节点 `prev_node`。`new_node->next = prev_node->next; prev_node->next = new_node;` 如果在头节点前插入，需要修改头指针，因此函数参数最好是 `**head`。
- 删除节点:
  1. 找到要删除节点 (`to_delete`) 的 *前一个* 节点 (`prev_node`)。
  2. 修改指针: `prev_node->next = to_delete->next;`
  3. 释放内存: `free(to_delete);`
  - **特殊情况:** 删除头节点时，需要修改头指针 (`*head = (*head)->next;`)。删除操作通常也需要 `**head` 参数。
- **释放整个链表 (`delete_record`):** 需要遍历链表，逐个释放节点内存。常用方法是使用两个指针，一个指向当前节点，一个指向下一个节点，防止释放当前节点后丢失后续链表的连接。

```
void delete_record(student **head) {
    student *current = *head;
    student *next_node;
    while (current != NULL) {
        next_node = current->next; // 保存下一个节点的地址
        // 如果节点内有动态分配的内存（如 name），先释放内部内存
        // free(current->name); // 假设 name 是 malloc 的
        free(current);         // 释放当前节点
        current = next_node;   // 移动到下一个节点
    }
    *head = NULL; // 将原始头指针设为 NULL
}
```

- **创建排序链表:** 每次插入新节点时，不再是插入到尾部，而是遍历链表找到合适的位置（按 GPA 降序）进行插入，维持链表的有序性。这类似于合并排序中的合并步骤，或者插入排序。

队列 (Queue)

1. 概念解读:

- **定义:** 队列是一种遵循 **先进先出 (First-In, First-Out, FIFO)** 原则的数据结构 。最早添加到队列中的元素是第一个被移除的。
- 操作:
  - **Enqueue:** 将元素添加到队尾 (Rear/End)。
  - **Dequeue:** 从队头 (Front/Head) 移除元素。
- **链表实现:** 可以用链表实现队列。通常在 **链表尾部** 添加元素 (Enqueue)，在 **链表头部** 移除元素 (Dequeue) 。为了高效地在尾部添加，最好维护一个指向队尾节点的指针 (Tail Pointer)，但这会增加实现的复杂度。如果只用头指针，Enqueue 需要遍历到队尾 (O(n))，Dequeue 在队头操作 (O(1))。

2. 代码分析 (基于头指针，Enqueue 遍历):

- Enqueue: 与在链表 尾部 插入节点的操作 (`insert_tail`) 非常相似，需要 `**head`参数。

```
// 和 insert_tail 代码几乎一样
void enqueue(node **queue_head, node *data) {
    node **current = queue_head; // 用双指针遍历
    while (*current) {
        current = &((*current)->next);
    }
    node *new_node = (node*)malloc(sizeof(node));
    *new_node = *data; // 可能需要深拷贝
    new_node->next = NULL; // 队列尾部节点 next 为 NULL
    *current = new_node;  // 链接新节点
}
```

- Dequeue:这与从链表 头部删除节点的操作相同。需要 `**head`参数。

```
node* dequeue(node **queue_head) {
    if (*queue_head == NULL) { // 队列为空
        return NULL;
    }
    node *temp = *queue_head;       // 临时保存队头节点
    *queue_head = (*queue_head)->next; // 更新队头指针
    // temp->next = NULL; // 可选
    return temp; // 返回队头节点指针
}
```

双向链表 (Doubly Linked List)

1. 概念解读:

- **定义:** 每个节点除了包含数据域和指向下一个节点的 `next` 指针外，还包含一个指向 **前一个节点** 的 `prev` 指针 。

- **优点:** 可以双向遍历，从某个节点向前或向后查找都更方便。某些操作（如删除已知节点）比单向链表更高效（如果已知节点指针，删除是 O(1)，单向链表需要 O(n) 查找前驱）。
- **缺点:** 每个节点需要额外的空间存储 `prev` 指针。插入和删除操作需要维护两个指针 (`next` 和 `prev`)，稍微复杂一些。
- **`prev` 指针的实现:** 课件中的 `prev` 指针被定义为 `struct dll_node_t **prev;`。这是一种不寻常但有效的设计。`prev` 不直接存储前一个节点的地址，而是存储 *指向前一个节点的 `next` 指针成员* 或 *头指针变量* 的地址。这种设计的优点在于，无论是修改头指针还是修改中间节点的 `next` 指针，都可以通过 `*(node->prev) = ...` 统一处理。

2. 代码分析 (`dll_insert_sorted`):

```
typedef struct dll_node_t {
    int val;
    struct dll_node_t *next; // 指向下一个节点
    struct dll_node_t **prev; // 指向前一个节点的 next 指针成员的地址，或头指针变量的地址
} dll_node;

/* add to the doubly linked list */
void dll_insert_sorted(dll_node **head, int v) { // head 是指向 main 中 head 指针变量的地址
    dll_node *tmp = malloc(sizeof(*tmp)); // 分配新节点
    tmp->val = v;                         // 设置新节点的值

    // 1. 找到插入位置
    // head（局部变量）在循环中会被修改，指向需要被新节点链接的那个指针的地址
    while (*head && ((*head)->val < v)) { // 当前节点存在 且 当前节点值小于新值 v
        head = &((*head)->next); // 移动 head，使其指向当前节点 next 指针的地址
    }
    // 循环结束时，*head 是第一个 >= v 的节点地址，或者 NULL（如果 v 最大或链表为空）
    // head 指向需要被修改的指针的地址（可能是 main 的 headptr，或某个节点的 next）

    // 2. 链接新节点 tmp
    tmp->next = *head; // 新节点的 next 指向找到的位置的节点（*head）

    // 3. 更新后继节点的 prev 指针（如果存在）
    if (*head) { // 如果 *head 不是 NULL（即插入位置不是链表末尾）
        // 让 *head 指向的那个节点的 prev 指针指向新节点 tmp 的 next 成员的地址
        (*head)->prev = &(tmp->next);
    }

    // 4. 设置新节点的 prev 指针
    // 新节点的 prev 指向 head（它存储了前一个节点的 next 指针的地址，或头指针变量的地址）
    tmp->prev = head;

    // 5. 更新前驱节点的 next 指针（或头指针）
```

```
    // 修改 head 指向的那个指针变量（前驱的 next 或 main 的 headptr），使其指向新节点 tmp
    *head = tmp;
}
```

3. 流程理解 (main 调用 dll_insert_sorted(&head, ...)):

- **dll_insert_sorted(&head, 3):** 链表为空。while 循环不执行。tmp->next = NULL。if(*head) 不执行。tmp->prev = &head (main中的)。*(&head) = tmp (即 head = tmp in main)。链表: 3 -> NULL。3->prev 指向 main 中的 head 变量。
- **dll_insert_sorted(&head, 1):** *head 是节点 3。(*head)->val (3) 不小于 v (1)。while 循环不执行。tmp->next = node(3)。if(*head) 执行，node(3)->prev = &(tmp->next)。tmp->prev = &head (main中的)。*(&head) = tmp (即 head = tmp in main)。链表: 1 -> 3 -> NULL。1->prev 指向 main 中的 head。3->prev 指向 node(1) 的 next 成员。
- **dll_insert_sorted(&head, 2):** *head 是节点 1。(*head)->val (1) 小于 v (2)。head 更新为 &((*head)->next)，即指向 node(1) 的 next 成员的地址。此时 *head 是节点 3 的地址。再次检查 while，(*head)->val (3) 不小于 v (2)。循环结束。tmp->next = node(3)。if(*head) 执行，node(3)->prev = &(tmp->next)。tmp->prev = head (指向 node(1) 的 next 成员地址)。*head = tmp (修改 node(1) 的 next 指针，使其指向 node(2) (即tmp))。链表: 1 -> 2 -> 3 -> NULL。1->prev 指向 main 中的 head。2->prev 指向 node(1) 的 next 成员。3->prev 指向 node(2) 的 next 成员。