Deleting a Node From a List

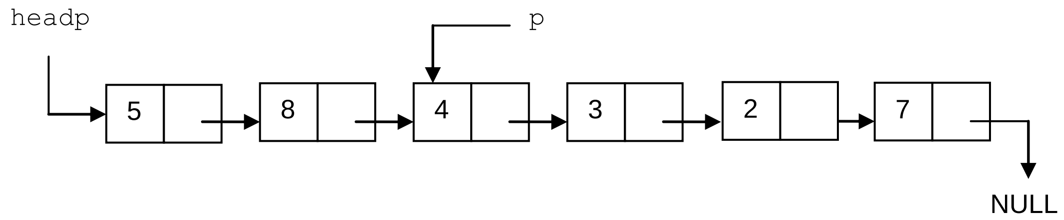
- The steps of deleting a node from a list are as follows:

    1. Save the address of the node you want to delete
    2. Save the data in the node you want to delete
    3. Form the new links
    4. Delete the node from the memory (using `free` function)

- Similar to add operation, we will examine the delete operation in four cases:

    1. Deleting a node between two nodes
    2. Deleting the last node of a list
    3. Deleting the first node of a list
    4. Deleting the only node of a list with a single node

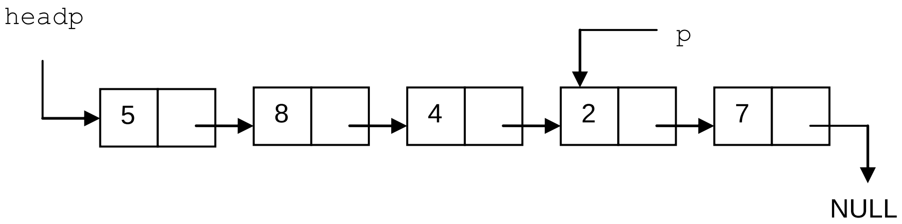
- In order to delete a node between two nodes, we need to know the address of the node which comes before the node we want to delete. For example, let's delete the node after the node pointed by `p` in the list below. Thus, we want to delete 3.



- We need to remember the address of the node we want to delete, so that we can free its memory location after the delete operation. Therefore, we need an additional variable to store that address.

```
void delete_after(node_t *p, int *item)
{
     node_t *del;
     /* save the address of the node you want to delete,
        thus the node after p */
     del = p->next;
     /* store the data in the node you want to delete in
        *item */
     *item = del->data;
     /* form the new links. link the node before the one to
        be deleted to the node that comes after it. */
     p->next = del->next;
     /* delete the node from the memory */
     free(del);
}
```

- Can we use the same function to delete the last node of a list, for instance, if we want to delete 7 from the list below?

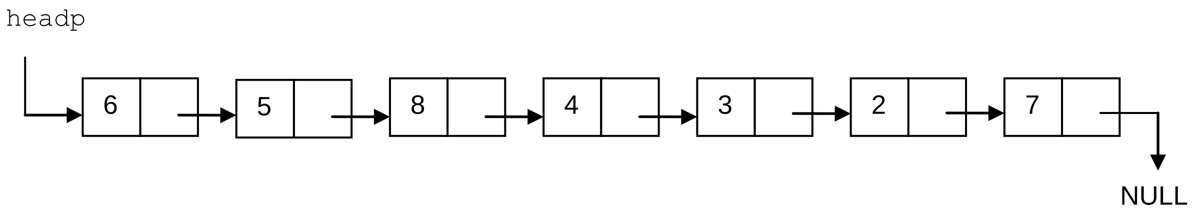


- Of course, we should first of all find the address of the node before the last node, so that we can delete the node after it. Assume that it is found and put in `p`, and the function is called as

```
delete_after(p, &num);
```

- First of all, the address of the last node, thus `p->next` will be stored in `del`. Then, data of that node, thus 7, will be put into `num`. Later, new links will be arranged, so that `p` will be linked to `del->next`, thus to NULL. That is correct, because after deleting the last node, the previous one, thus `p`, will be the last node of the list. Finally, the node will be deleted from the memory.

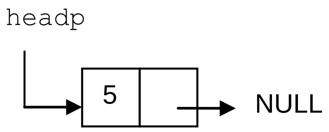
- Can we use the `delete_after` function to delete the first node of a list, for example to delete 6 from the the list below?



- No, because the function needs `p`, the address of the node before the node to be deleted, but there is no node before the first one. If we send `headp` to the function, it will delete 5, not 6. Therefore, similar to add operation, we need to define a separate function to delete the first node of a list. This function needs the address of the first node, thus `headp`, as a parameter.

```
node_t *delete_first(node_t *headp, int *item) {
     node_t *del;
     /* Remember the address of the first node */
     del = headp;
     /* Store the data in the first node in *item */
     *item = del->data;
     /* Form the new links. HEAD must point to the second
        node. */
     headp = del->next;
     /* Delete the first node from the memory */
     free(del);
     return (headp);
}
```

- Can we use the delete_first function to delete the only node of a list with a single node, so that the list will become empty? For example, can we use it to delete 5 from the the list below?



- If we call the function as

```
headp = delete_first(headp, &num);
```

first of all, the address of the only node, thus headp will be stored in del. Then, data of that node, thus 5, will be put into num. Later, new links will be arranged, so that headp will be linked to headp->next, thus to NULL. That is correct, because after deleting the only node of a list with a single node, the list becomes empty, thus headp points to NULL. Finally, the node will be deleted from the memory.

- Therefore, when we are dealing with forward linked lists, we need to define two different delete functions: delete_first, which deletes the only node of a list with a single node, or the first node of any list, and delete_after, which deletes a node between two nodes, or the last node of any list.

Example: Define a function that adds a new item to the end of a list.

- You know that, to add a new node to the end of a list, we must first of all find the address of the last node, so that we can call add_after function to make the operation. The address of the last node is only known by the previous node. The address of that node is only known by the node before it, and so on.

- The only address that we know is the address of the first node, which is stored in headp. Therefore, we have to start from the first node, follow the path shown by the pointers of each node, and get to the address of the last node. As you know, this operation is called as *traversing the list*.

- How can we determine which node is the last one? If a node is pointing to NULL, it is the last node.

- So, we will use a while loop, which checks if a node does not point to NULL, so that, when it terminates, it will find the address of the last node. We will use a pointer p which will start from the headp and move to the next node in each repetition of the loop. Then, we will use that pointer as the parameter of add_after function.

- What about if the initial list is empty? To add a new node to an empty list, we should call the function add_beginning instead of add_after. Therefore, to consider that case, we should add an if check at the beginning.

- Hence, the solution is:

```
node_t *add_end(node_t *headp, int item)
{
      node_t *p;
      /* If the list is empty, add the item at the beginning
         of the list */
      if (headp == NULL)
          headp = add_beginning(headp, item);
      else
      {   /* find the address of the last node */
          /* start from the beginning */
          p = headp;
          /* repeat until a node that points to NULL
             (i.e., the last node) is found */
          while (p->next != NULL)
              /* pass to the next node */
              p = p->next;
          /* add the item after that node */
          add_after(p, item);
      }
      return (headp);
}
```

- Notice that, if we did not make the if check, it would also cause a problem in the condition of the while loop, because in an empty list `headp->next` is undefined.

Example: Define a function that deletes the last node of a list.

- Remember that, we must first of all find the address of the node before the last node, so that we can call `delete_after` function to make the operation.

- How can we determine which node is the node before the last one? If the node after a node is pointing to NULL, it is the node before the last one.

- So, we need to use a similar loop, which checks if the node after a node does not point to NULL, so that, when it terminates, it will find the address of the node before the last node.

- What about if the initial list is empty? If the list is empty, no node should be deleted. So, we should do all those operations only if the list is not empty. Thus we need to check if `headp != NULL` before starting the operations.

- What about if there is only one node in the list? Deleting the last node of a list with one node means deleting its first node. Thus, we should call the function `delete_first` instead of `delete_after`. Therefore, to consider that case, we should add another if check.

- Hence, the solution is:

```
node_t *delete_last(node_t *headp, int *item)
{
     node_t *p;
     /* Check if the list is not empty */
     if (headp != NULL)
         /* If there is only one node in the list, the last
            node is also the first node */
         if (headp->next == NULL)
             headp = delete_first(headp, item);
         else
         {   /* find the address of the node before the last
                node */
             p = headp;
             while (p->next->next != NULL)
                 p = p->next;
             /* delete the node after it, thus last node */
             delete_after(p, item);
         }
     else // if the list is empty, put a dummy value in item
         *item = -987654321;
     return (headp);
}
```

- Notice that, if we did not make the second if check, it would also cause a problem in the condition of the while loop, because in a list with a single node `headp->next->next` is undefined.

Example: Define a function that deletes the n^{th} node of a list.

- In order to delete the n^{th} node we must know the address of the $(n-1)^{st}$ node, so that we can use `delete_after` function. But, if we are required to delete the first node, thus if n is 1, then we need to use `delete_first` function. If the list is empty, or if there are not that many nodes in the list, nothing can be deleted, you should return a dummy value.

```
node_t *delete_nth(node_t *headp, int n, int *item)
{
     node_t  *p;
     int k;
     if (headp != NULL)
         if (n == 1)
             headp = delete_first(headp, item);
         else
         {   /* find the address of the (n-1)st node */
             p = headp;
             k = 1;
             /* Repeat until the last node or the (n - 1)st
                node is reached */
             while (p->next != NULL && k < n - 1)
             {
               p = p->next;
               k++;
             }
             /* If the last node is reached before the
                (n - 1)st node, there are less than n nodes
                in the list; return a dummy value */
             if (p->next == NULL)
               *item = -987654321;
             else
               /* delete the nth node */
               delete_after(p, item);
         }
     else
         *item = -987654321;
     return (headp);
}
```

Example: Define a function to destroy a list.

```
node_t *destroy_list(node_t *headp)
{
     node_t  *p;
     while (headp != NULL)
     {
         p = headp;
         headp = headp->next;
         free(p);
     }
     return (headp);
}
```