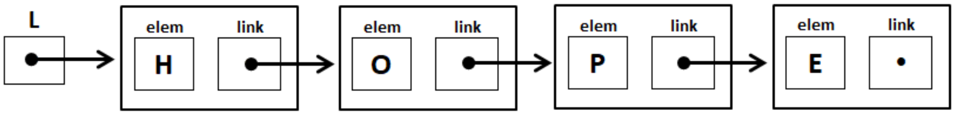
Linked List and Traversals

Linked list of characters

Definition and Declaration:

```
typedef struct node {
     char elem;
     struct node *link;
}ctype, *List;
List L;
```

Figure 1: Linked List with elements `'H'`, `'O'`, `'P'`, and `'E'`.



Assumption: List L is populated with 4 elements using `malloc()` or `calloc()` function.

Notes:

1) Variables (locations in memory) that are created via malloc() or calloc() are allocated in the heap, NOT in the stack.
2) Variables in the heap are dynamically allocated, i.e. during program execution when the malloc() function is executed.
3) Variables in the heap are DEALLOCATED dynamically using free() function.

Exercise:

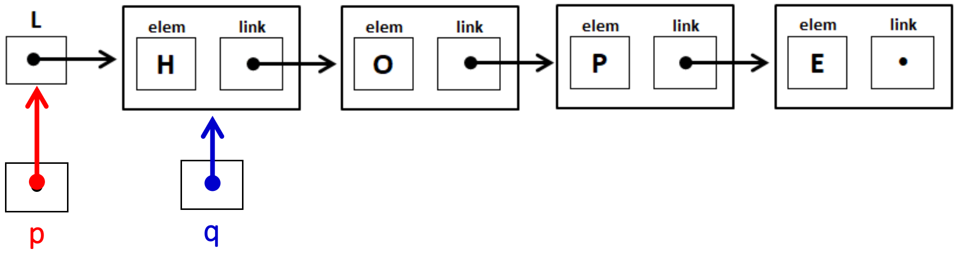
Based on the Figure 2:

1) Write the declaration of variables p and q.

2) Write the C statement that will let p point to L.

3) Write the C statement that will let q point to the node L is pointing to.

Figure 2: Linked List with pointer to a node q and pointer to a pointer to a node p.



Note: Observe the different variables that the pointers are pointing to.

There are two (2) types of linked list traversals:

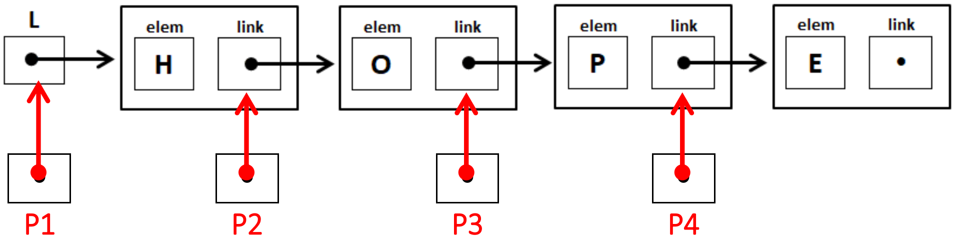
1) Pointer to node (PN) traversal using variable such as q.
   - This traversal is used when elements of the linked list are retrieved or modified.
   - Examples: displayList() and selectionSort()
   - Accessing of elements: `q->elem`
   - Moving the PN to point to the next node: `q = q->link;`

2) Pointer to Pointer to node (PPN) traversal using variable such as p.
   - This traversal is used when inserting a new element (node) or deleting an existing element(node) at position other the first.
   - Examples: insertLast(), insertPos(), and deleteID()
   - Accessing of elements: `(*p)->elem`
   - Moving the PPN to point to the next pointer: `p = &(*p)->link;`

Deleting an element (node) using PPN:

A linked list with 4 elements has 4 valid positions for deletion. PPNs P1, P2, P3, and P4 are used to delete elements at positions 1, 2, 3, and 4 respectively. Figure 3 shows the positions.

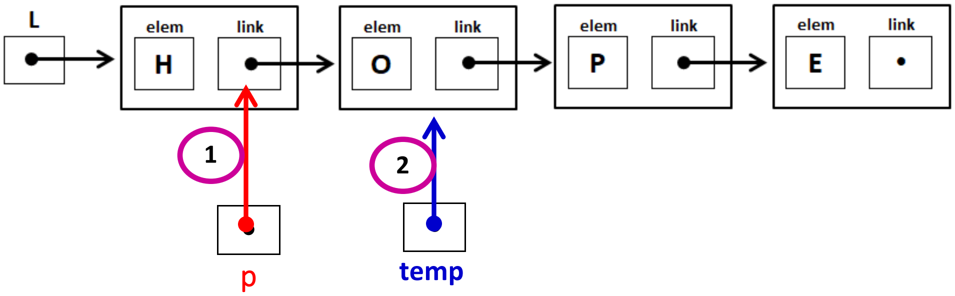
Figure 3: Linked list with pointer positions valid for deletion.



Steps in deleting a node:

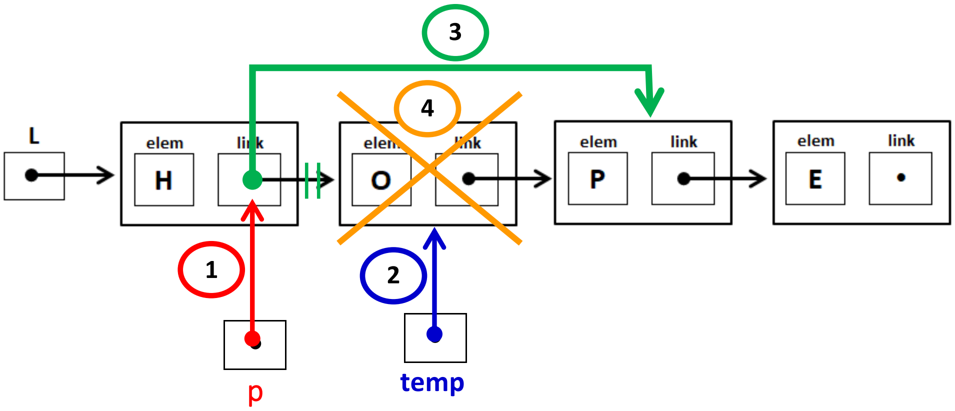
1) **Place PPN to the appropriate position. If element 'O' is to be deleted, PPN p will point to the link field of the node containing 'H'. Using deference or indirection, the alias of the link field pointed to by PPN p is *p.**
2) **Let a temporary PN variable, say temp, point to the node to be deleted. Hint: Use *p.**

Figure 4: Illustration of Steps 1 and 2 of deleting a node



3) **Let the link field pointed to by PPN p point to the node pointed to by temp.**
4) **Deallocate deleted node, i.e. node removed from the linked list, using `free()` function.**

Figure 5: Illustration of Steps 3 and 4 of deleting a node

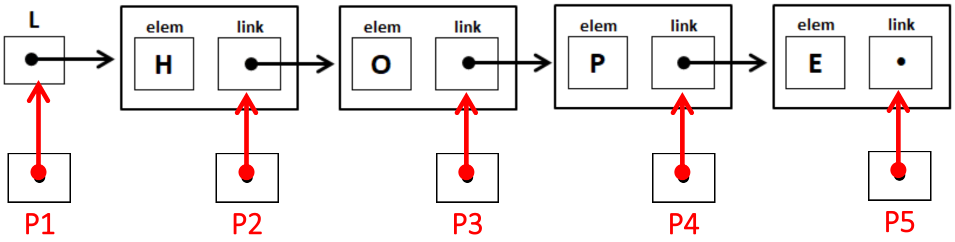


cfpeña052420

Inserting an element (node) using PPN:

A linked list with 4 elements has 5 valid positions for insertion. PPNs P1, P2, P3, P4, and P5 are used to insert elements at positions 1, 2, 3, 4, and 5 respectively. Figure 6 shows the positions.

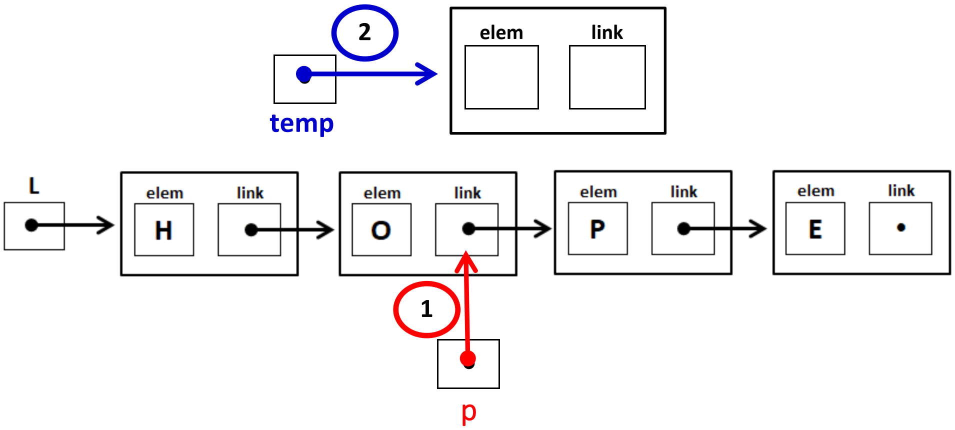
Figure 6: Linked list with pointer positions valid for insertion.



Steps in inserting a node:

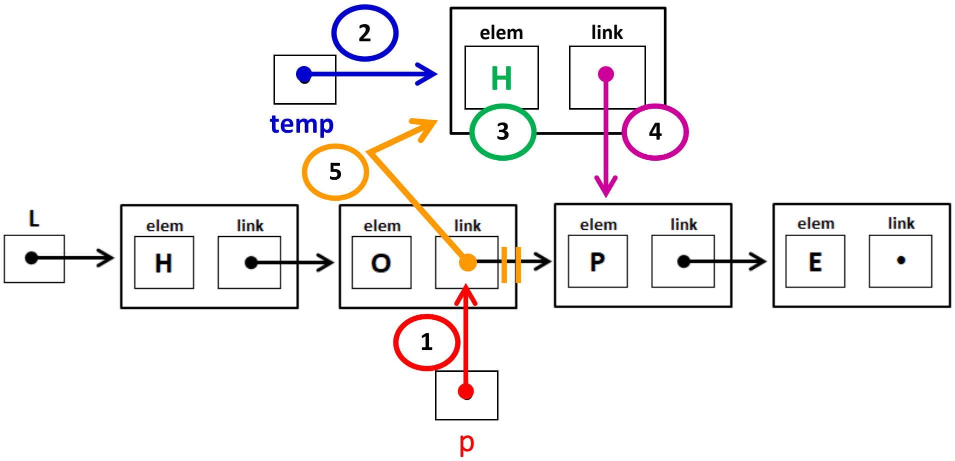
1) Place PPN to the appropriate position. Suppose element 'K' is to be inserted a position 3, then PPN p will point the link field of the node containing 'O', i.e. p is equal to P3 in the above illustration. See illustration below.
2) Create the dynamically allocate the new node using malloc(). Note: Always check if the memory allocation is successful.

Figure 7: Illustration of Steps 1 and 2 of inserting a node



3) Assign the new element 'K' in the elem field of the newly created node.
4) Let the link field of the newly created node point to the node containing element 'P', i.e. pointed to by *p.
5) Let the link field of the node containing element 'O', which is *p point to the newly created node.

Figure 8: Illustration of Steps 3, 4 and 5 of inserting a node



cfpeña052420