CS29003 ALGORITHMS LABORATORY
(PDS Brush Up)
Date: 18 – July – 2019

Doubly linked list with one pointer per node!

Recall that, in a singly linked list, every node of a linked list stores some data value and a pointer to the next node in the list; the value of the next pointer of the last node is set to NULL. The idea of singly linked list has been extended to doubly linked list as follows. In a doubly linked list, we store a pointer to the next node and another pointer to its previous node. Please refer to **??** for a pictorial overview.

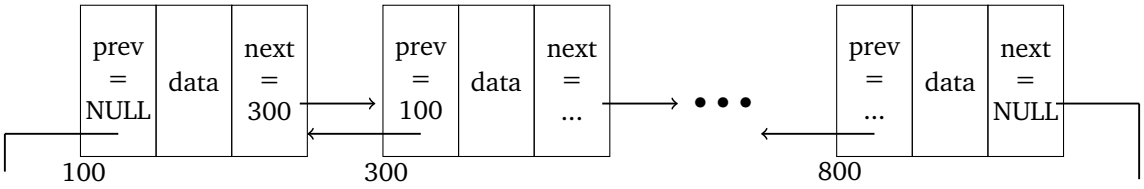

Figure 1: Usual structure of a doubly linked list with two pointers per node.

Let us tweak this basic structure of a doubly linked list as follows. Instead of storing two pointers per node, let us store only XOR of these two pointers in every node thereby saving (in every node) the space required to store one pointer. Please refer to **??** for a pictorial overview.

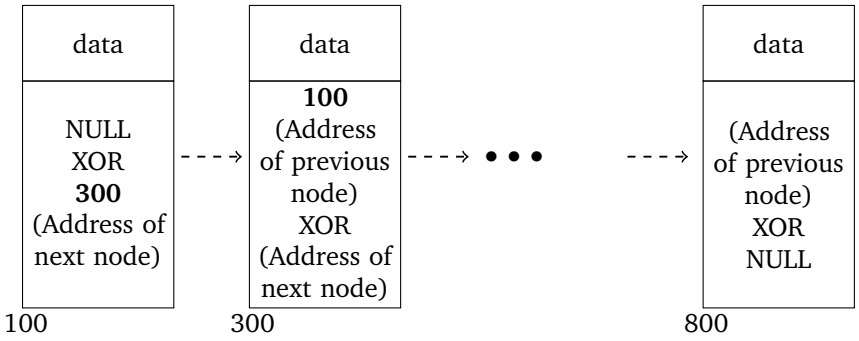

Figure 2: Modified structure of a doubly linked list with one pointer per node.

The XOR function is defined in the following manner. Let $X = (x_1, x_2, \ldots, x_n)$ and $Y = (y_1, y_2, \ldots, y_n)$ be two Boolean strings of length n bits each. Then X XOR $Y = (x_1$ XOR y_1, x_2 XOR $y_2, \ldots, x_n$ XOR $y_n)$ where 0 XOR $0 = 0, 1$ XOR $1 = 0, 0$ XOR $1 = 1, 1$ XOR $0 = 1$. One can easily verify the following properties of the XOR function.

- $\triangleright$ X XOR $Y = Y$ XOR X
- $\triangleright$ $(X$ XOR $Y)$ XOR $Z = X$ XOR $(Y$ XOR $Z)$
- $\triangleright$ X XOR $X = \bar{0}$

Convince yourself that all the usual operations on a doubly linked list can still be carried out correctly in our modified representation. In this exercise, you first take the length n of a doubly linked list from the

user. Then create a doubly linked list of length n with random data (integers). Needless to say that you should only store the XOR of the addresses of the previous and the next node in every node as discussed above. For every doubly linked list, you maintain a head and a tail pointer pointing respectively to the first and the last node of the list. In this doubly linked list, you perform the following operations:

1. Traverse the doubly linked list both from the front to the end and from the end to the front and print data values in that order. The prototype of your functions should be as follows.

   void traverse_from_front_to_end(struct node *head);

   void traverse_from_end_to_front(struct node *tail);

2. Reverse your doubly linked list. **Once the doubly linked list is created, you SHOULD NOT create any new node and change data field of any node.** The prototype of your function should be as follows.

   void reverse(struct node **head, struct node **tail);

   Observe that you need to pass double pointers here since the doubly linked list is going to change unlike traversal functions.

3. Sort the doubly linked list in non-decreasing order of data values. **Once the doubly linked list is created, you SHOULD NOT create any new node and change data field of any node.** The prototype of your function should be as follows.

   void sort(struct node **head, struct node **tail);

   Here too you need to pass double pointers for the same reason.

Sample Output

$n = 10$

Doubly linked list traversed from front to end: $3, 10, -12, 34, -10, -50, 25, 1, -18, -71$

Doubly linked list traversed from end to front: $-71, -18, 1, 25, -50, -10, 34, -12, 10, 3$

Reversed doubly linked list traversed from front to end: $-71, -18, 1, 25, -50, -10, 34, -12, 10, 3$

Sorted doubly linked list traversed from front to end: $-71, -50, -18, -12, -10, 1, 3, 10, 25, 34$