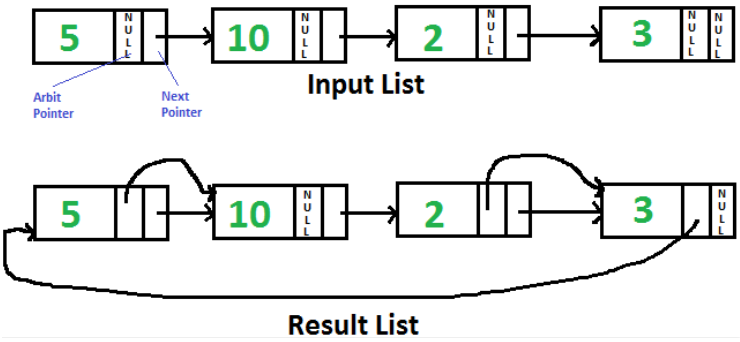
Point to next higher value node in a linked list with an arbitrary pointer

Given singly linked list with every node having an additional "arbitrary" pointer that currently points to NULL. Need to make the "arbitrary" pointer point to the next higher value node.



We strongly recommend to minimize your browser and try this yourself first

A **Simple Solution** is to traverse all nodes one by one, for every node, find the node which has next greater value of current node and change the next pointer. Time Complexity of this solution is $O(n^2)$.

An **Efficient Solution** works in O(nLogn) time. The idea is to use Merge Sort for linked list.
1) Traverse input list and copy next pointer to arbit pointer for every node.
2) Do Merge Sort for the linked list formed by arbit pointers.

Below is C implementation of above idea. All of the merger sort functions are taken from here. The taken functions are modified here so that they work on arbit pointers instead of next pointers.

```
// C program to populate arbit pointers to next higher value
// using merge sort
#include<stdio.h>
#include<stdlib.h>

/* Link list node */
struct node
{
    int data;
    struct node* next, *arbit;
};

/* function prototypes */
struct node* SortedMerge(struct node* a, struct node* b);
void FrontBackSplit(struct node* source,
                    struct node** frontRef, struct node** backRef);

/* sorts the linked list formed by arbit pointers
  (does not change next pointer or data) */
void MergeSort(struct node** headRef)
{
    struct node* head = *headRef;
    struct node* a, *b;

    /* Base case -- length 0 or 1 */
    if ((head == NULL) || (head->arbit == NULL))
        return;

    /* Split head into 'a' and 'b' sublists */
    FrontBackSplit(head, &a, &b);

    /* Recursively sort the sublists */
    MergeSort(&a);
    MergeSort(&b);
```

```
    /* answer = merge the two sorted lists together */
    *headRef = SortedMerge(a, b);
}

/* See http://geeksforgeeks.org/?p=3622 for details of this
   function */
struct node* SortedMerge(struct node* a, struct node* b)
{
    struct node* result = NULL;

    /* Base cases */
    if (a == NULL)
        return (b);
    else if (b==NULL)
        return (a);

    /* Pick either a or b, and recur */
    if (a->data <= b->data)
    {
        result = a;
        result->arbit = SortedMerge(a->arbit, b);
    }
    else
    {
        result = b;
        result->arbit = SortedMerge(a, b->arbit);
    }

    return (result);
}

/* Split the nodes of the given list into front and back halves,
   and return the two lists using the reference parameters.
   If the length is odd, the extra node should go in the front list.
   Uses the fast/slow pointer strategy.  */
void FrontBackSplit(struct node* source,
                    struct node** frontRef, struct node** backRef)
{
    struct node* fast, *slow;

    if (source==NULL || source->arbit==NULL)
    {
        /* length < 2 cases */
        *frontRef = source;
        *backRef = NULL;
        return;
    }

    slow = source,  fast = source->arbit;

    /* Advance 'fast' two nodes, and advance 'slow' one node */
    while (fast != NULL)
    {
        fast = fast->arbit;
        if (fast != NULL)
        {
            slow = slow->arbit;
            fast = fast->arbit;
        }
    }

    /* 'slow' is before the midpoint in the list, so split it in two
      at that point. */
    *frontRef = source;
    *backRef = slow->arbit;
    slow->arbit = NULL;
}

/* Function to insert a node at the beginging of the linked list */
void push(struct node** head_ref, int new_data)
{
    /* allocate node */
```

```
    struct node* new_node =
        (struct node*) malloc(sizeof(struct node));

    /* put in the data  */
    new_node->data  = new_data;

    /* link the old list off the new node */
    new_node->next = (*head_ref);

    new_node->arbit = NULL;

    /* move the head to point to the new node */
    (*head_ref)    = new_node;
}

// Utility function to print result linked list
void printListafter(struct node *node, struct node *anode)
{
    printf("Traversal using Next Pointer\n");
    while (node!=NULL)
    {
        printf("%d, ", node->data);
        node = node->next;
    }

    printf("\nTraversal using Arbit Pointer\n");
    while (anode!=NULL)
    {
        printf("%d, ", anode->data);
        anode = anode->arbit;
    }
}

// This function populates arbit pointer in every node to the
// next higher value. And returns pointer to the node with
// minimum value
struct node* populateArbit(struct node *head)
{
    // Copy next pointers to arbit pointers
    struct node *temp = head;
    while (temp != NULL)
    {
        temp->arbit = temp->next;
        temp = temp->next;
    }

    // Do merge sort for arbitrary pointers
    MergeSort(&head);

    // Return head of arbitrary pointer linked list
    return head;
}

/* Drier program to test above functions*/
int main()
{
    /* Start with the empty list */
    struct node* head = NULL;

    /* Let us create the list shown above */
    push(&head, 3);
    push(&head, 2);
    push(&head, 10);
    push(&head, 5);

    /* Sort the above created Linked List */
    struct node *ahead = populateArbit(head);

    printf("\nResult Linked List is: \n");
    printListafter(head, ahead);

    getchar();
    return 0;
```

```
    return 0;
}
```

Output:

```
Result Linked List is:
Traversal using Next Pointer
5, 10, 2, 3,
Traversal using Arbit Pointer
2, 3, 5, 10,
```