Intersection of two Sorted Linked Lists

Given two lists sorted in increasing order, create and return a new list representing the intersection of the two lists. The new list should be made with its own memory — the original lists should not be changed.

For example, let the first linked list be 1->2->3->4->6 and second linked list be 2->4->6->8, then your function should create and return a third list as 2->4->6.

Method 1 (Using Dummy Node)

The strategy here uses a temporary dummy node as the start of the result list. The pointer tail always points to the last node in the result list, so appending new nodes is easy. The dummy node gives tail something to point to initially when the result list is empty. This dummy node is efficient, since it is only temporary, and it is allocated in the stack. The loop proceeds, removing one node from either 'a' or 'b', and adding it to tail. When we are done, the result is in dummy.next.

```
#include<stdio.h>
#include<stdlib.h>

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

void push(struct node** head_ref, int new_data);

/*This solution uses the temporary dummy to build up the result list */
struct node* sortedIntersect(struct node* a, struct node* b)
{
  struct node dummy;
  struct node* tail = &dummy;
  dummy.next = NULL;

  /* Once one or the other list runs out -- we're done */
  while (a != NULL && b != NULL)
  {
    if (a->data == b->data)
    {
       push((&tail->next), a->data);
       tail = tail->next;
       a = a->next;
       b = b->next;
    }
    else if (a->data < b->data) /* advance the smaller list */
       a = a->next;
    else
       b = b->next;
  }
  return(dummy.next);
}

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

/* Function to print nodes in a given linked list */
void printList(struct node *node)
{
  while (node != NULL)
  {
   printf("%d ", node->data);
   node = node->next;
  }
}

/* Drier program to test above functions*/
int main()
{
  /* Start with the empty lists */
  struct node* a = NULL;
  struct node* b = NULL;
  struct node *intersect = NULL;

  /* Let us create the first sorted linked list to test the functions
   Created linked list will be 1->2->3->4->5->6 */
  push(&a, 6);
  push(&a, 5);
  push(&a, 4);
  push(&a, 3);
  push(&a, 2);
  push(&a, 1);

  /* Let us create the second sorted linked list
   Created linked list will be 2->4->6->8 */
  push(&b, 8);
  push(&b, 6);
  push(&b, 4);
  push(&b, 2);

  /* Find the intersection two linked lists */
  intersect = sortedIntersect(a, b);

  printf("\n Linked list containing common items of a & b \n ");
  printList(intersect);

  getchar();
}
```

Time Complexity: O(m+n) where m and n are number of nodes in first and second linked lists respectively.

Method 2 (Using Local References)

This solution is structurally very similar to the above, but it avoids using a dummy node Instead, it maintains a struct node** pointer, lastPtrRef, that always points to the last pointer of the result list. This solves the same case that the dummy node did — dealing with the result list when it is empty. If you are trying to build up a list at its tail, either the dummy node or the struct node** “reference” strategy can be used

```
#include<stdio.h>
#include<stdlib.h>

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

```
void push(struct node** head_ref, int new_data);

/* This solution uses the local reference */
struct node* sortedIntersect(struct node* a, struct node* b)
{
  struct node* result = NULL;
  struct node** lastPtrRef = &result;

  /* Advance comparing the first nodes in both lists.
    When one or the other list runs out, we're done. */
  while (a!=NULL && b!=NULL)
  {
    if (a->data == b->data)
    {
      /* found a node for the intersection */
      push(lastPtrRef, a->data);
      lastPtrRef = &((*lastPtrRef)->next);
      a = a->next;
      b = b->next;
    }
    else if (a->data < b->data)
      a=a->next;       /* advance the smaller list */
    else
      b=b->next;
  }
  return(result);
}

/* UTILITY FUNCTIONS */
/* 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);

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

/* Function to print nodes in a given linked list */
void printList(struct node *node)
{
  while(node != NULL)
  {
   printf("%d ", node->data);
   node = node->next;
  }
}

/* Drier program to test above functions*/
int main()
{
  /* Start with the empty lists */
  struct node* a = NULL;
  struct node* b = NULL;
  struct node *intersect = NULL;

  /* Let us create the first sorted linked list to test the functions
   Created linked list will be 1->2->3->4->5->6 */
  push(&a, 6);
  push(&a, 5);
  push(&a, 4);
  push(&a, 3);
  push(&a, 2);
  push(&a, 1);
  /* Let us create the second sorted linked list
   Created linked list will be 2->4->6->8 */
  push(&b, 8);
  push(&b, 6);
  push(&b, 4);
  push(&b, 2);

  /* Find the intersection two linked lists */
  intersect = sortedIntersect(a, b);

  printf("\n Linked list containing common items of a & b \n ");
  printList(intersect);

  getchar();
}
```

Time Complexity: O(m+n) where m and n are number of nodes in first and second linked lists respectively.

Method 3 (Recursive)

Below is the recursive implementation of sortedIntersect().

```
#include<stdio.h>
#include<stdlib.h>

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

struct node *sortedIntersect(struct node *a, struct node *b)
{
    /* base case */
    if (a == NULL || b == NULL)
        return NULL;

    /* If both lists are non-empty */

    /* advance the smaller list and call recursively */
    if (a->data < b->data)
        return sortedIntersect(a->next, b);

    if (a->data > b->data)
        return sortedIntersect(a, b->next);

    // Below lines are executed only when a->data == b->data
    struct node *temp = (struct node *)malloc(sizeof(struct node));
    temp->data = a->data;

    /* advance both lists and call recursively */
    temp->next = sortedIntersect(a->next, b->next);
    return temp;
}

/* UTILITY FUNCTIONS */
/* Function to insert a node at the begining 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);

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

/* Function to print nodes in a given linked list */
void printList(struct node *node)
{
    while (node != NULL)
    {
        printf("%d ", node->data);
        node = node->next;
    }
}

/* Drier program to test above functions*/
int main()
{
    /* Start with the empty lists */
    struct node* a = NULL;
    struct node* b = NULL;
    struct node *intersect = NULL;

    /* Let us create the first sorted linked list to test the functions
     Created linked list will be 1->2->3->4->5->6 */
    push(&a, 6);
    push(&a, 5);
    push(&a, 4);
    push(&a, 3);
    push(&a, 2);
    push(&a, 1);

    /* Let us create the second sorted linked list
     Created linked list will be 2->4->6->8 */
    push(&b, 8);
    push(&b, 6);
    push(&b, 4);
    push(&b, 2);

    /* Find the intersection two linked lists */
    intersect = sortedIntersect(a, b);

    printf("\n Linked list containing common items of a & b \n ");
    printList(intersect);

    return 0;
}
```

Time Complexity: O(m+n) where m and n are number of nodes in first and second linked lists respectively.

References:
cslibrary.stanford.edu/105/LinkedListProblems.pdf