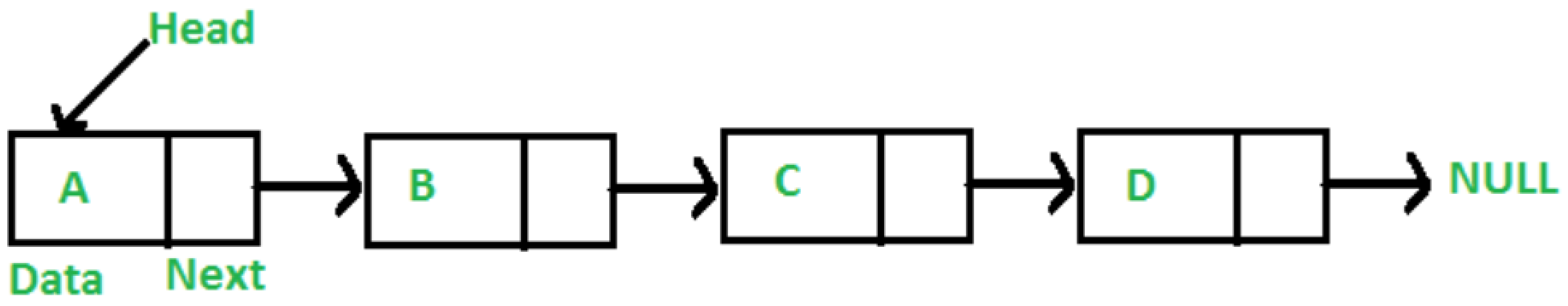
Introduction to Linked List: Review

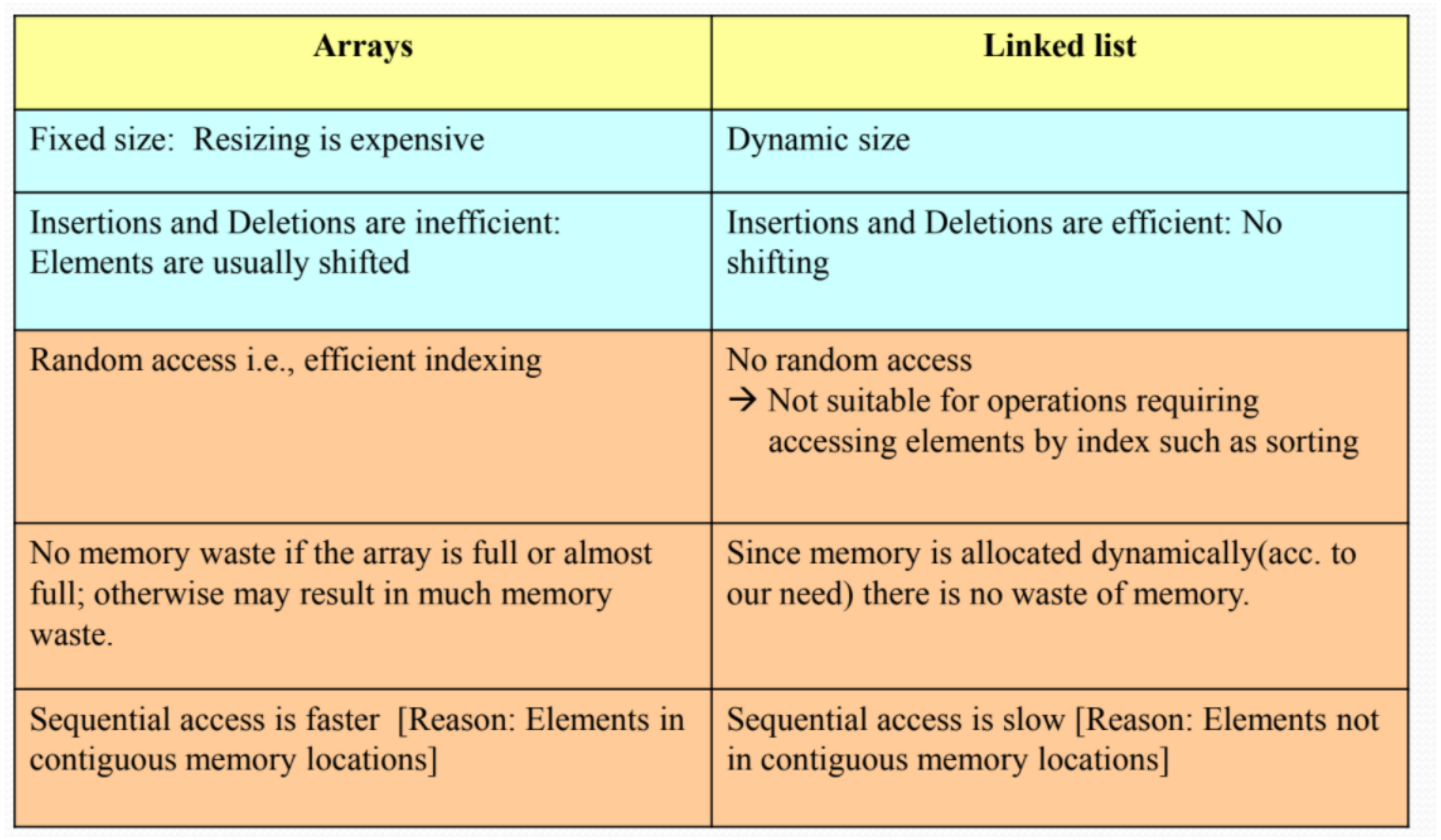
Source: http://www.geeksforgeeks.org/data-structures/linked-list/

Linked List

- Fundamental data structures in C
- Like arrays, linked list is a linear data structure
- Unlike arrays, linked list elements are not stored at contiguous location, the elements are linked using pointers



Array vs Linked List



Arrays	**Linked list**
Fixed size: Resizing is expensive	Dynamic size
Insertions and Deletions are inefficient: Elements are usually shifted	Insertions and Deletions are efficient: No shifting
Random access i.e., efficient indexing	No random access → Not suitable for operations requiring accessing elements by index such as sorting
No memory waste if the array is full or almost full; otherwise may result in much memory waste.	Since memory is allocated dynamically(acc. to our need) there is no waste of memory.
Sequential access is faster [Reason: Elements in contiguous memory locations]	Sequential access is slow [Reason: Elements not in contiguous memory locations]

Why Linked List-1

- Advantages over arrays
  - Dynamic size
  - Ease of insertion or deletion
    - Inserting a new element in an array of elements is expensive, because room has to be created for the new elements and to create room existing elements have to be shifted

      For example, in a system if we maintain a sorted list of IDs in an array id[].

      id[] = [1000, 1010, 1050, 2000, 2040]

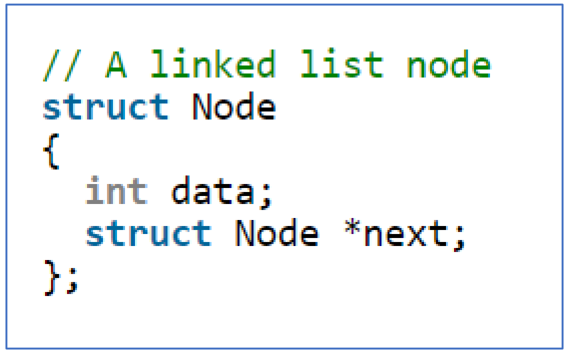
      If we want to insert a new ID 1005, then to maintain the sorted order, we have to move all the elements after 1000
    - Deletion is also expensive with arrays until unless some special techniques are used. For example, to delete 1010 in id[], everything after 1010 has to be moved

Why Linked List-2

- Drawbacks of Linked List
  - Random access is not allowed.
    - Need to access elements sequentially starting from the first node. So we cannot do binary search with linked lists
    - Extra memory space for a pointer is required with each element of the list

Representation in C

- A linked list is represented by a pointer to the first node of the linked list
  - The first node is called head
  - If the linked list is empty, then the value of head is null
- Each node in a list consists of at least two parts
  1. Data
  2. Pointer to the next node
- In C, we can represent a node using structures



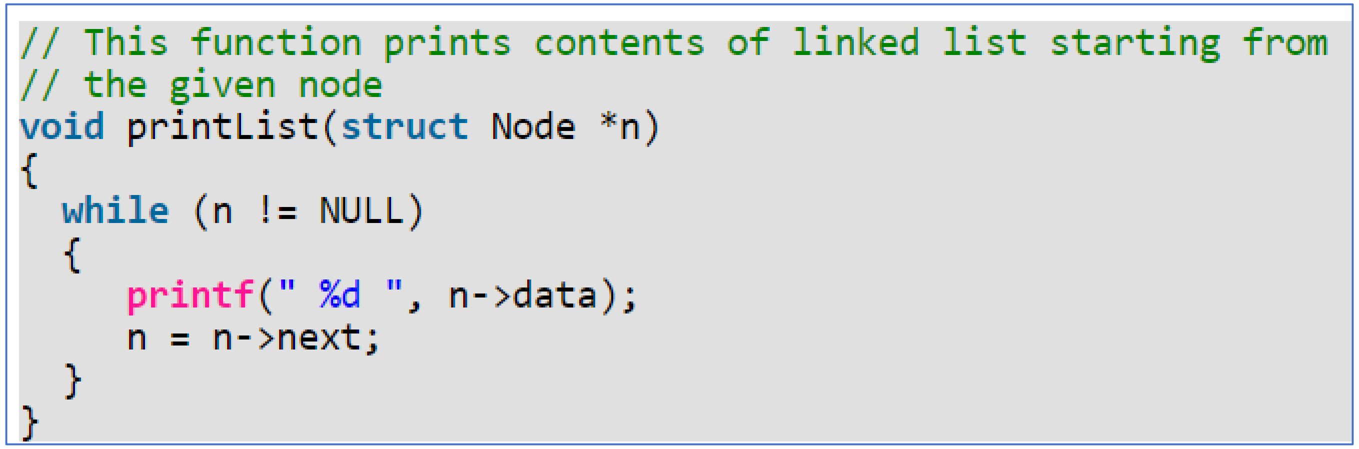
```
// A linked list node
struct Node
{
  int data;
  struct Node *next;
};
```

First Simple Linked List in C

```c
// A simple C program to introduce
// a linked list
#include<stdio.h>
#include<stdlib.h>

struct Node
{
  int data;
  struct Node *next;
};

// Program to create a simple linked
// list with 3 nodes
int main()
{
  struct Node* head = NULL;
  struct Node* second = NULL;
  struct Node* third = NULL;

  // allocate 3 nodes in the heap
  head = (struct Node*)malloc(sizeof(struct Node));
  second = (struct Node*)malloc(sizeof(struct Node));
  third = (struct Node*)malloc(sizeof(struct Node));

  /* Three blocks have been allocated  dynamically.
     We have pointers to these three blocks as first, second and third
       head           second          third
        |               |               |
        |               |               |
    +---+-----+     +----+----+     +----+----+
    | #   | # |     | #  | #  |     |  # |  # |
    +---+-----+     +----+----+     +----+----+

   # represents any random value.
   Data is random because we haven't assigned anything yet  */

  head->data = 1; //assign data in first node
  head->next = second; // Link first node with the second node

  /* data has been assigned to data part of first block (block
    pointed by head).  And next pointer of first block points to
    second.  So they both are linked.

       head          second         third
        |              |              |
        |              |              |
    +---+---+     +----+----+     +-----+----+
    | 1  | o----->| #  | #  |     |  #  | #  |
    +---+---+     +----+----+     +-----+----+
  */

  second->data = 2; //assign data to second node
  second->next = third; // Link second node with the third node

   /* data has been assigned to data part of second block (block pointed by
      second). And next pointer of the second block points to third block.
     So all three blocks are linked.

       head         second         third
        |             |             |
        |             |             |
    +---+---+     +---+---+     +----+----+
    | 1  | o----->| 2 | o-----> |  # |  # |      */
    +---+---+     +---+---+     +----+----+

   third->data = 3; //assign data to third node
   third->next = NULL;

   /* data has been assigned to data part of third block (block pointed
     by third). And next pointer of the third block is made NULL to indicate
     that the linked list is terminated here.

      We have the linked list ready.

           head
             |
             |
        +---+---+     +---+---+       +----+------+
        | 1  | o----->|  2  | o-----> |  3 | NULL |
        +---+---+     +---+---+       +----+------+


     Note that only head is sufficient to represent the whole list.  We can
     traverse the complete list by following next pointers.    */

   return 0;
}
```

Linked List Traversal

- In the previous program, we created a simple linked list with three nodes
- Let us traverse the created list and print the data of each node



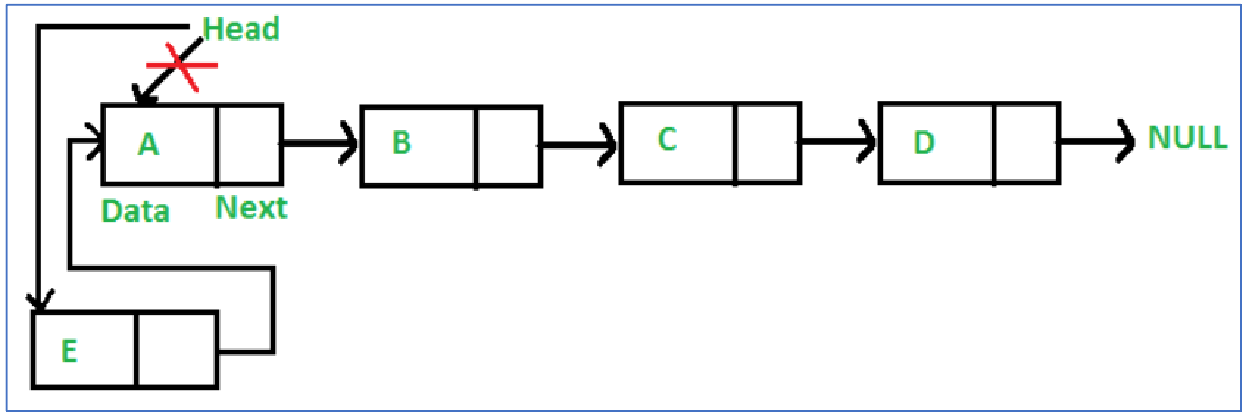
```c
// This function prints contents of linked list starting from
// the given node
void printList(struct Node *n)
{
  while (n != NULL)
  {
     printf(" %d ", n->data);
     n = n->next;
  }
}
```

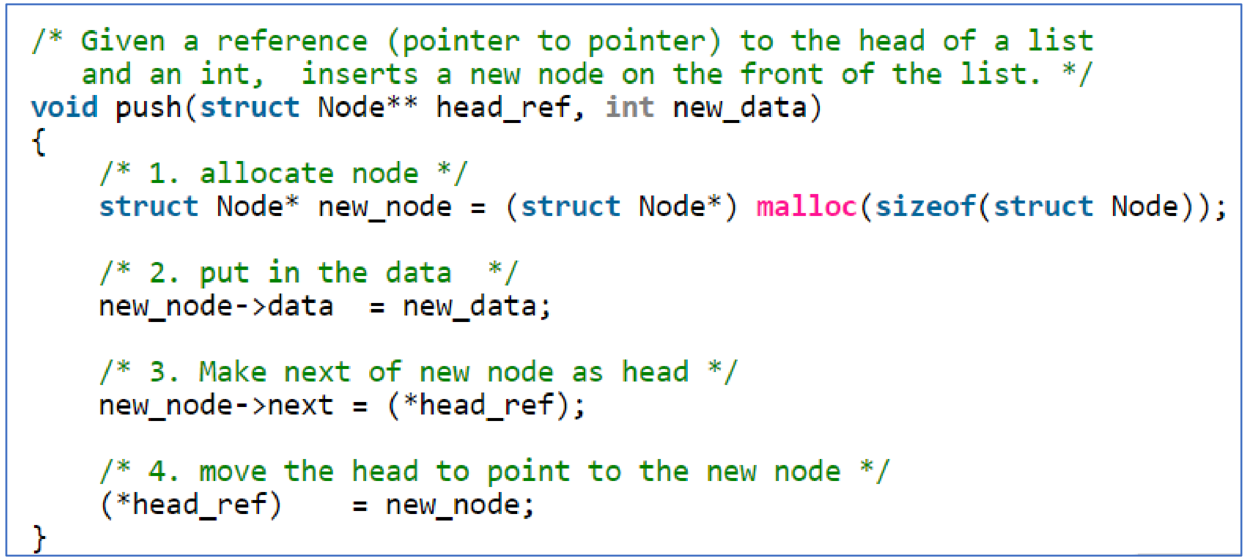
Inserting A Node

- Methods to insert a new node in linked list
    1. At the front of the linked list
    2. After a given node
    3. At the end of the linked list

Add a node at the front

- A 4 step process
- New node can be added before the head of the given linked list
- Newly added node becomes the new head of the linked list
- Let us call the function that adds at the front of the list is push()
- The push() must receive a pointer to the head pointer, because push must change the head pointer to point to the new node

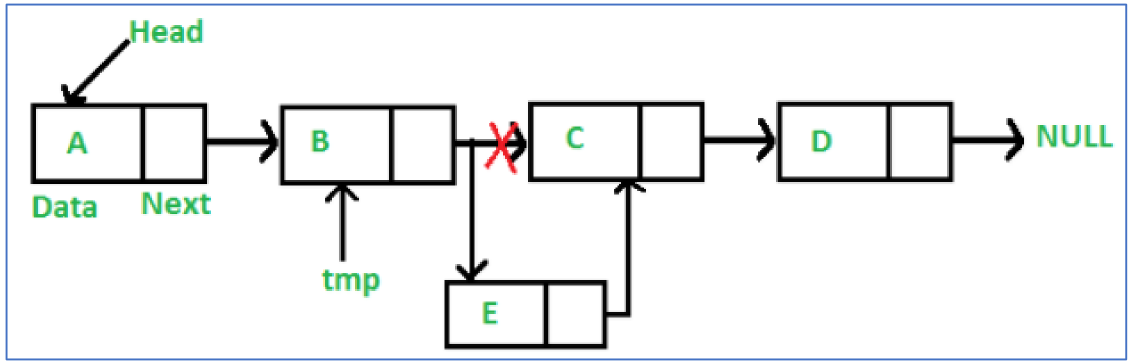


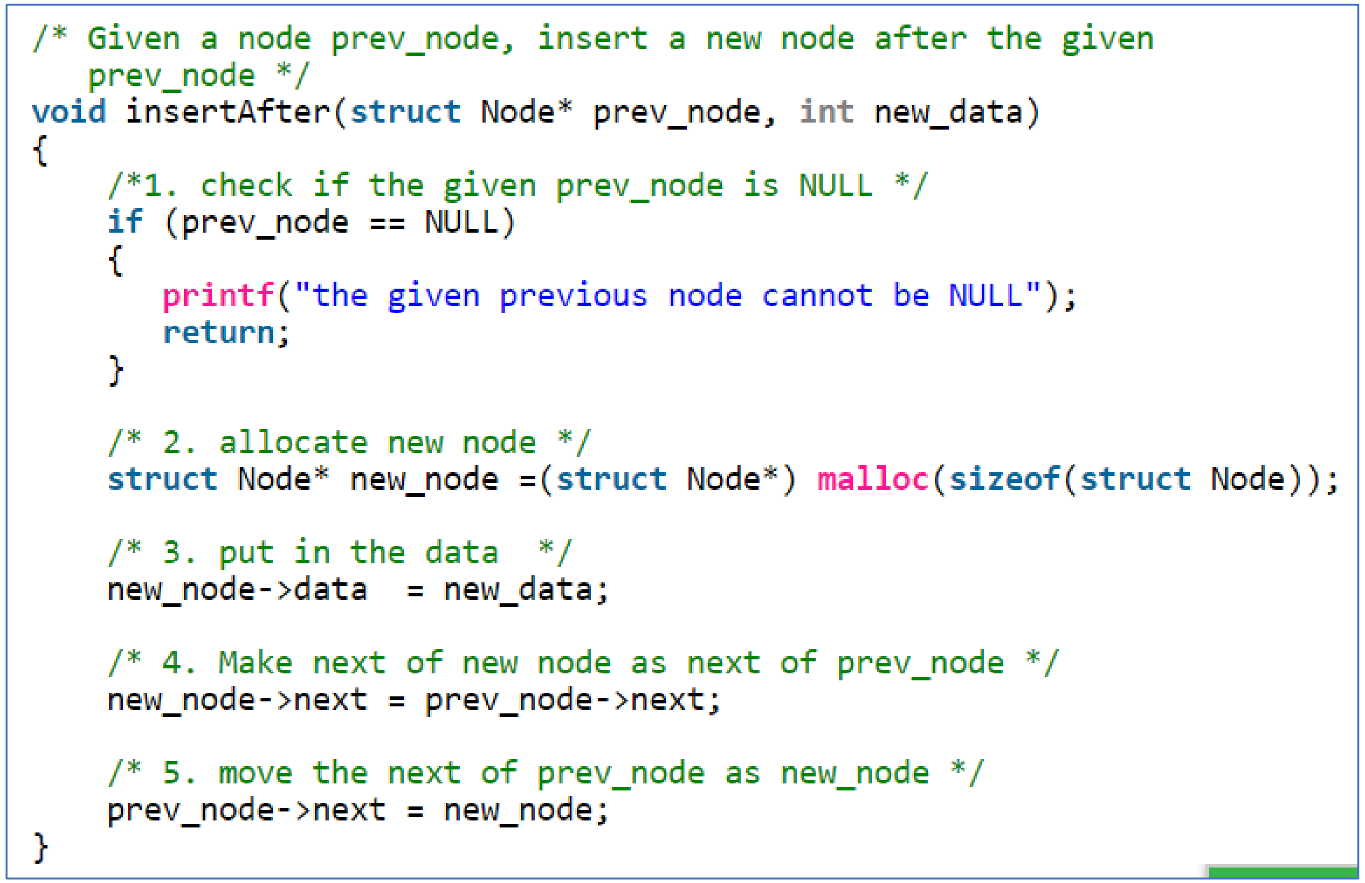


```
/* Given a reference (pointer to pointer) to the head of a list
   and an int,  inserts a new node on the front of the list. */
void push(struct Node** head_ref, int new_data)
{
    /* 1. allocate node */
    struct Node* new_node = (struct Node*) malloc(sizeof(struct Node));

    /* 2. put in the data  */
    new_node->data  = new_data;

    /* 3. Make next of new node as head */
    new_node->next = (*head_ref);

    /* 4. move the head to point to the new node */
    (*head_ref)    = new_node;
}
```

Add a node after a given node

- A 5 step process
- We are a given pointer to a node
- New node is inserted after the given node

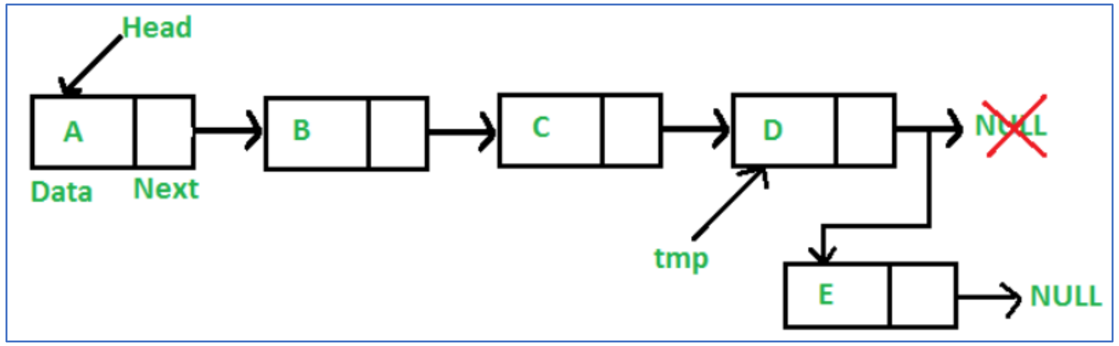


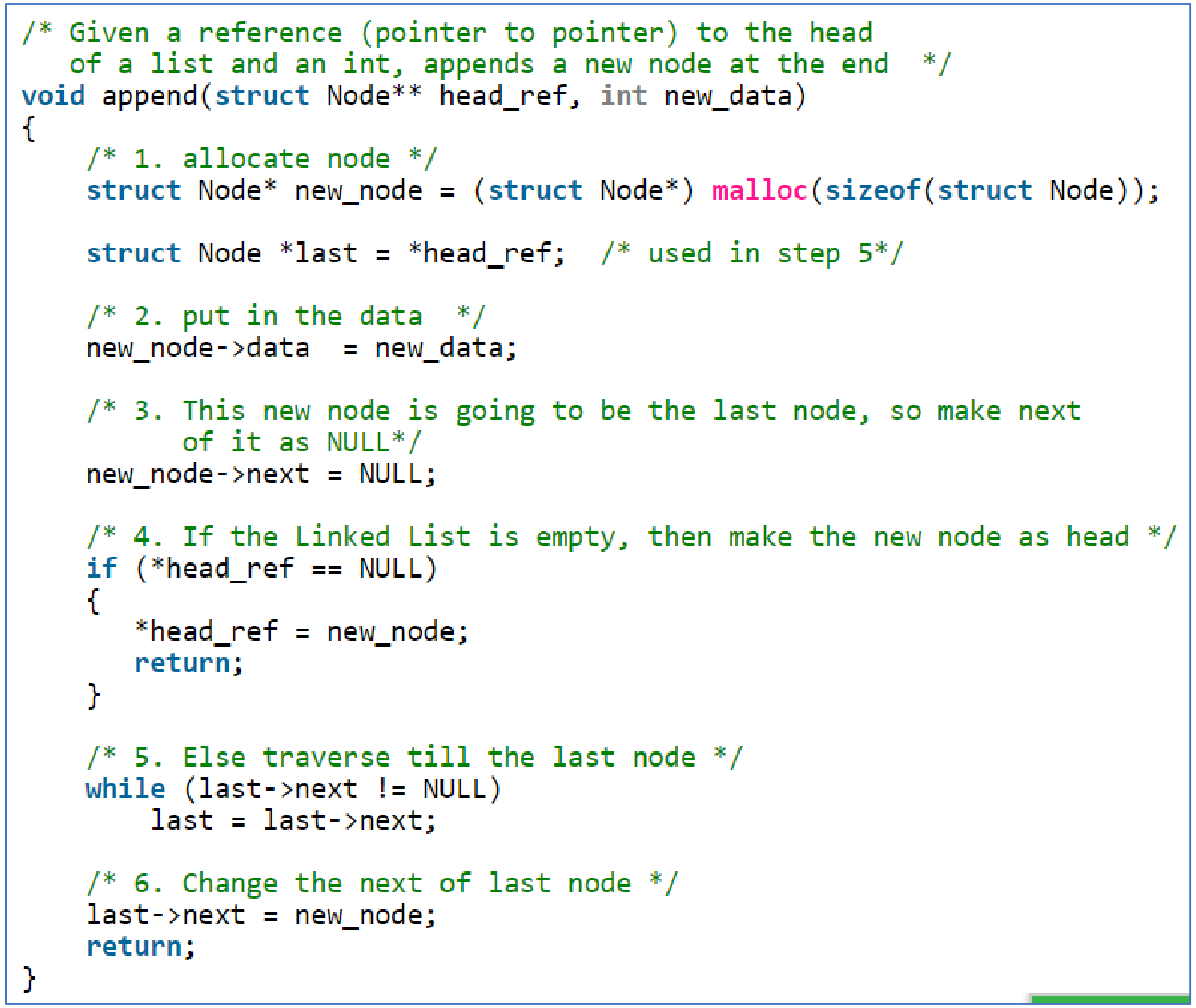


```
/* Given a node prev_node, insert a new node after the given
   prev_node */
void insertAfter(struct Node* prev_node, int new_data)
{
    /*1. check if the given prev_node is NULL */
    if (prev_node == NULL)
    {
       printf("the given previous node cannot be NULL");
       return;
    }

    /* 2. allocate new node */
    struct Node* new_node =(struct Node*) malloc(sizeof(struct Node));

    /* 3. put in the data  */
    new_node->data  = new_data;

    /* 4. Make next of new node as next of prev_node */
    new_node->next = prev_node->next;

    /* 5. move the next of prev_node as new_node */
    prev_node->next = new_node;
}
```

Add a node at the end

- A 6 step process
- New node can be added after the last node of the given linked list
- Since a linked list is typically represented by the head of it, we have traverse the list till end and then change the next of last node to new node

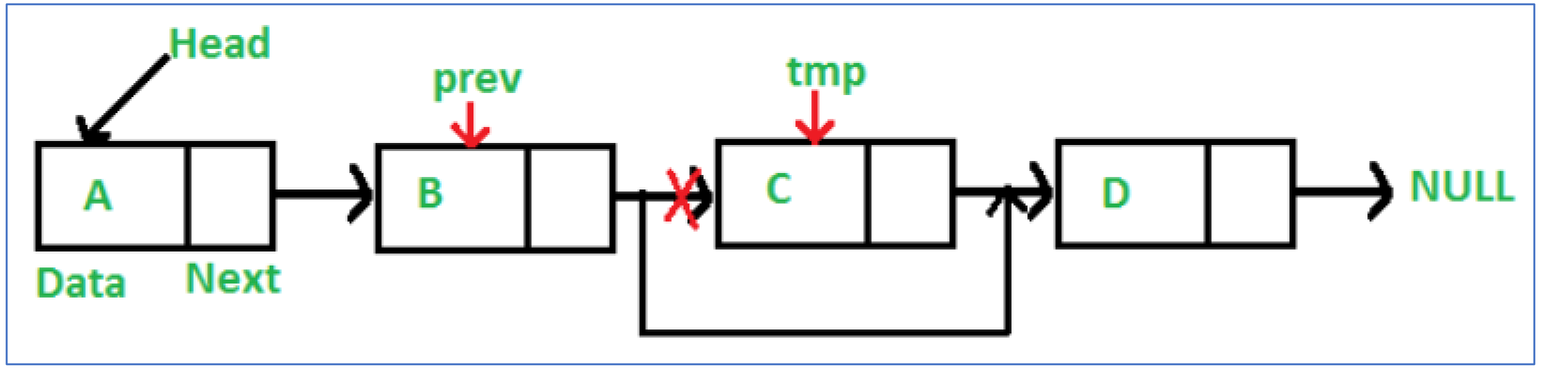




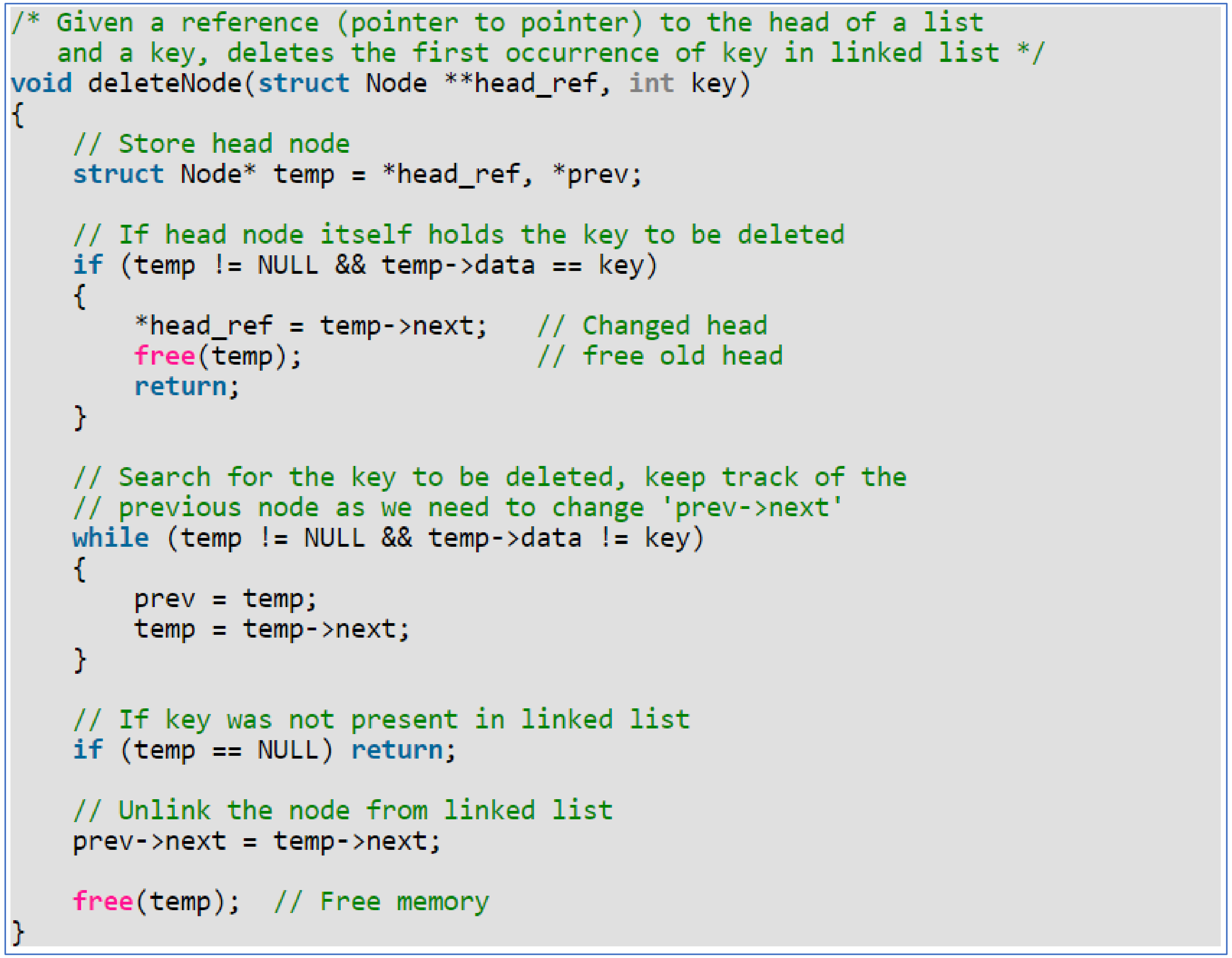
```
/* Given a reference (pointer to pointer) to the head
   of a list and an int, appends a new node at the end  */
void append(struct Node** head_ref, int new_data)
{
    /* 1. allocate node */
    struct Node* new_node = (struct Node*) malloc(sizeof(struct Node));

    struct Node *last = *head_ref;  /* used in step 5*/

    /* 2. put in the data  */
    new_node->data  = new_data;

    /* 3. This new node is going to be the last node, so make next
          of it as NULL*/
    new_node->next = NULL;

    /* 4. If the Linked List is empty, then make the new node as head */
    if (*head_ref == NULL)
    {
       *head_ref = new_node;
       return;
    }

    /* 5. Else traverse till the last node */
    while (last->next != NULL)
        last = last->next;

    /* 6. Change the next of last node */
    last->next = new_node;
    return;
}
```

Deleting A Node

- Let us formulate the problem statement to understand the deletion process.
  - Given a “key”, delete the first occurrence of this key in linked list
- To delete a node from linked list, we need to do following steps:
  - Find previous node of the node to be deleted
  - Change next of previous node
  - Free memory for the node to be deleted
  - Since every node of linked list is dynamically allocated using malloc() in C, we need to call free() for freeing memory allocated for the node to be deleted



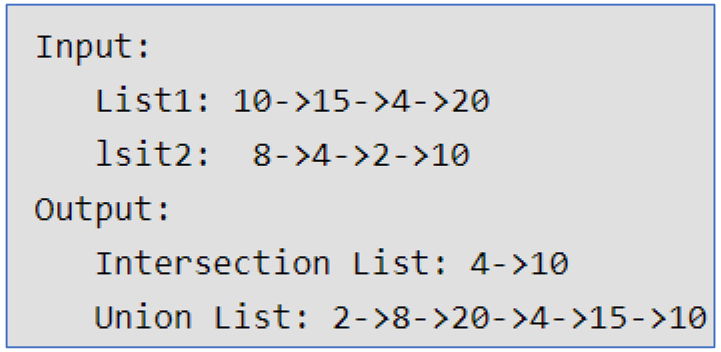
Deleting A Node



```c
/* Given a reference (pointer to pointer) to the head of a list
   and a key, deletes the first occurrence of key in linked list */
void deleteNode(struct Node **head_ref, int key)
{
    // Store head node
    struct Node* temp = *head_ref, *prev;

    // If head node itself holds the key to be deleted
    if (temp != NULL && temp->data == key)
    {
        *head_ref = temp->next;   // Changed head
        free(temp);               // free old head
        return;
    }

    // Search for the key to be deleted, keep track of the
    // previous node as we need to change 'prev->next'
    while (temp != NULL && temp->data != key)
    {
        prev = temp;
        temp = temp->next;
    }

    // If key was not present in linked list
    if (temp == NULL) return;

    // Unlink the node from linked list
    prev->next = temp->next;

    free(temp);  // Free memory
}
```

Union & Intersection of Two Linked Lists

- Problem Statement: Given two linked lists, create union & intersection lists that contain union and intersection of the elements present in the given lists
- Order of elements in output lists doesn’t matter
- Intersection (list1, list2)
  - Initialize result list as NULL
  - Traverse list1 and look for its each element in list2
  - If the element is present in list2, then add the element to result
- Union (list1,list2)
  - Initialize result list as NULL
  - Traverse list1 and add all of its elements to result
  - Traverse list2: if an element of list2 is already present in result then do not insert it to the result, otherwise insert



```
Input:
   List1: 10->15->4->20
   lsit2:  8->4->2->10
Output:
   Intersection List: 4->10
   Union List: 2->8->20->4->15->10
```

Union & Intersection of Two Linked Lists

```c
/* Function to get union of two linked lists head1 and head2 */
struct Node *getUnion(struct Node *head1, struct Node *head2)
{
    struct Node *result = NULL;
    struct Node *t1 = head1, *t2 = head2;

    // Insert all elements of list1 to the result list
    while (t1 != NULL)
    {
        push(&result, t1->data);
        t1 = t1->next;
    }

    // Insert those elements of list2 which are not
    // present in result list
    while (t2 != NULL)
    {
        if (!isPresent(result, t2->data))
            push(&result, t2->data);
        t2 = t2->next;
    }

    return result;
}
```

```c
/* Function to get intersection of two linked lists
  head1 and head2 */
struct Node *getIntersection(struct Node *head1,
                              struct Node *head2)
{
    struct Node *result = NULL;
    struct Node *t1 = head1;

    // Traverse list1 and search each element of it in
    // list2. If the element is present in list 2, then
    // insert the element to result
    while (t1 != NULL)
    {
        if (isPresent(head2, t1->data))
            push (&result, t1->data);
        t1 = t1->next;
    }

    return result;
}
```

```c
/* A utility function that returns true if data is
   present in linked list else return false */
bool isPresent (struct Node *head, int data)
{
    struct Node *t = head;
    while (t != NULL)
    {
        if (t->data == data)
            return 1;
        t = t->next;
    }
    return 0;
}
```

```c
/* A utility function to insert a node at the begining of a 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;
}
```

Link of the Codes

- Linked List Traversal:

  http://www.geeksforgeeks.org/linked-list-set-1-introduction/

- Linked List Insertion:

  http://www.geeksforgeeks.org/linked-list-set-2-inserting-a-node/

- Linked List Deletion:

  http://www.geeksforgeeks.org/linked-list-set-3-deleting-node/

- Union & Intersection of Two Linked Lists:

  http://www.geeksforgeeks.org/union-and-intersection-of-two-linked-lists/