# Data Structure: Linked Lists

Tanvir Ahmed, PhD
Department of Computer Science,
University of Central Florida

# Content

- Linked List
  - Types of Linked List
  - Linked Lists Operations
  - Singly Linked List
    - Traversing, Inserting, Deleting
    - Sorted Linked List
  - Doubly Linked List
    - Traversing, Inserting, Deleting
    - Sorted Linked List
  - Circular linked list

# Linked List

# Linked List

- Sequence of connected nodes containing data items.
- Each node contains a connection to another link
- **<u>Second most-used</u>** data structure after array
- Example representation of a Linked List:

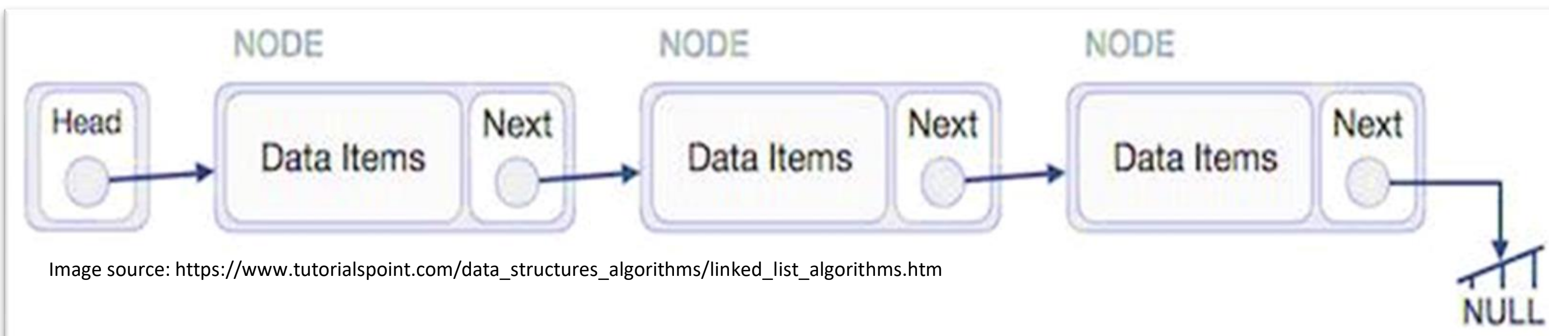


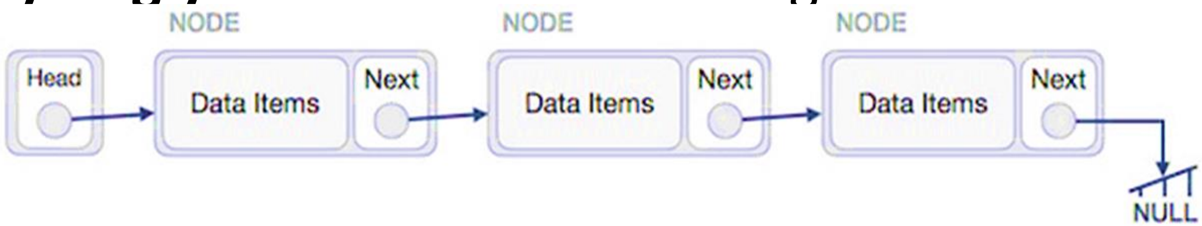

Image source: https://www.tutorialspoint.com/data_structures_algorithms/linked_list_algorithms.htm

# Why linked list?

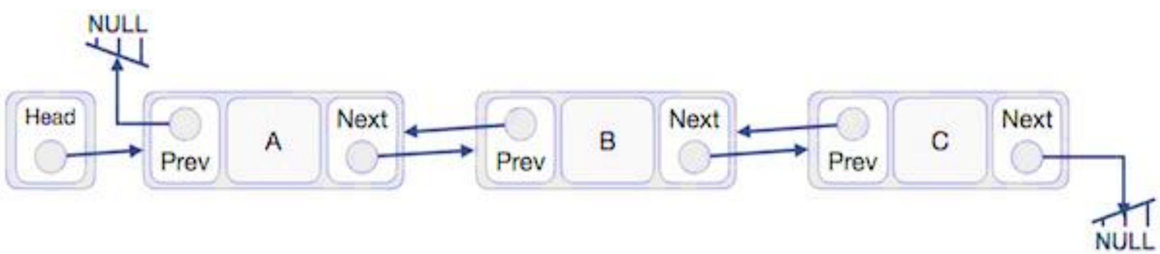
- Why not just use an array?
  - Each node in an array is stored, physically, in contiguous spaces in memory
  - Arrays are fixed size (not dynamic)
  - Inserting and deleting elements is difficult
  - If you have an array of size 1000 and if we want to insert an element after $5^{th}$ element, all the remaining 995 items must be shifted.
- Why linked list?
  - They are dynamic; length can increase and decrease as necessary.
  - Each node does not necessarily follow the previous one in memory
  - Insertion and deletion is cheap (only need to change few nodes at-most)
- What is negative side of linked list?
  - Getting a particular node may take a large number of operations, as we do not know the address of any individual node

# Types of Linked List

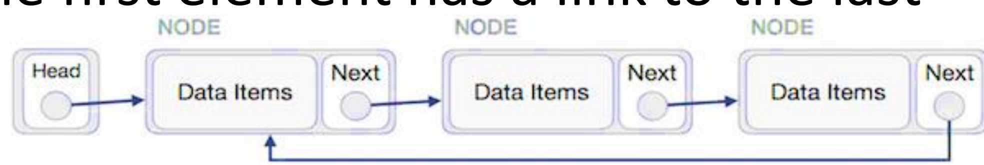
- **Simple/Singly Linked List** - Item navigation is forward only

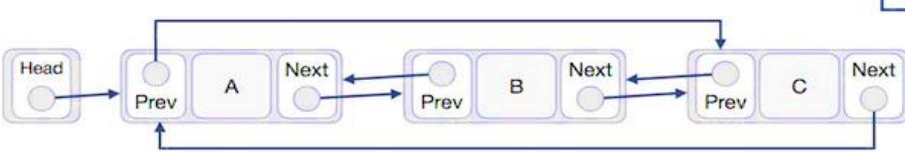



- **Doubly Linked List** – Items can be navigated forward and backward.




- **Circular Linked List** - Last item contains link of the first element as next and the first element has a link to the last element as previous.



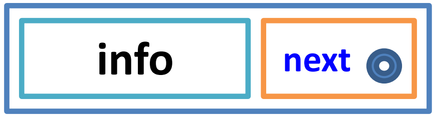



# Basic Operations of Linked Lists

- **Insertion** – Adds an element in the list.
- **Deletion** – Deletes a given item from the list.
- **Display** – Displays the complete list in a forward manner.
- **Search** – Search for a given item

# Simple/Singly Linked List

# Defining a Node




Node

```
typedef struct node
{
    int info;
    struct node  *next;
} node;
```

- A node has two parts:
  - **info** or known as data, that holds the data you want to store
    - You can store any type of data you want
      - It can be simply an integer or multiple integer, or it can be a string, or it can be even a structure
      - How about you create a playlist? In that case you might want to store song name, artist name, and more other information you want.
  - and **a link** which is a pointer. Known as next. It is a pointer that can point to a Node type variable
    - i.e., it can hold address of a Node

# A node with more than one fields

```
struct Book_node
 {
        char name[20];
        char author[8];
        int year;
        struct Book_node *next;
};
```

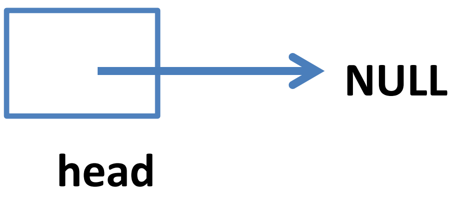
- The above node has 3 fields for data
- The right side example, has Book as data/info in the linked list's node

```
struct Book
 {
        char name[20];
        char author[8];
        int year;
};

struct Book_node
 {
        struct Book info;
        struct Book_node *next;
};
```

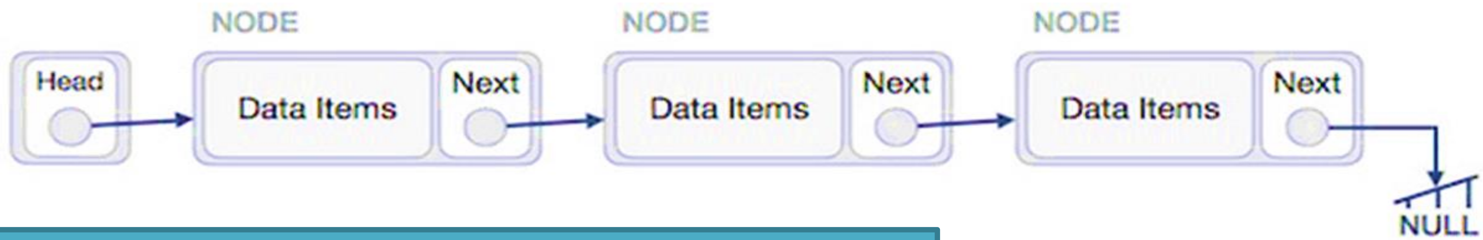
# Head of the list


- The first node in the linked list is considered as head.
- A node type pointer is used to keep track of the head
- It is the **most important** node in a linked list.
- If you loss head some how in your code, you will loss your linked list!
- **What is an emptily linked list?**
  - **If head is NULL!**



# Traversing a Linked List

- While dealing with linked list, you have to walk through the linked list a lot.
- Traversing means: Traversing/ walking from Head to other nodes and accessing them.
- Many operations require traversing
- Consider the following linked list. The following code snippet can give you an idea how to traverse the list.




```
node *t;

t = head; //assuming Head already initialized

while (t->Next != Null)
    t = t->Next;
```

- Can you think how to display the info in the Linked List?
- Just add the following statement in the loop:
  printf (“%d ”, t->info);

- Can you modify this code snippet to print only the even data?

# Operations in a Linked List

- Insert a new node
- Delete a node
- Search for a node
- Counting nodes
- Modifying nodes
- and more

# Insert into Linked List

You can insert into 3 different places:

1. Beginning of the list
2. End of the list
3. Between nodes in the list

**General Steps:**

1. Create a temporary node. Fill the “data” and “next”
2. Look for position where to insert
3. Link the temporary node appropriately in the list

**Special Caution:**

Always deal with head specially as if you loss or mistake with the head of the linked list, you will mess-up with your list!

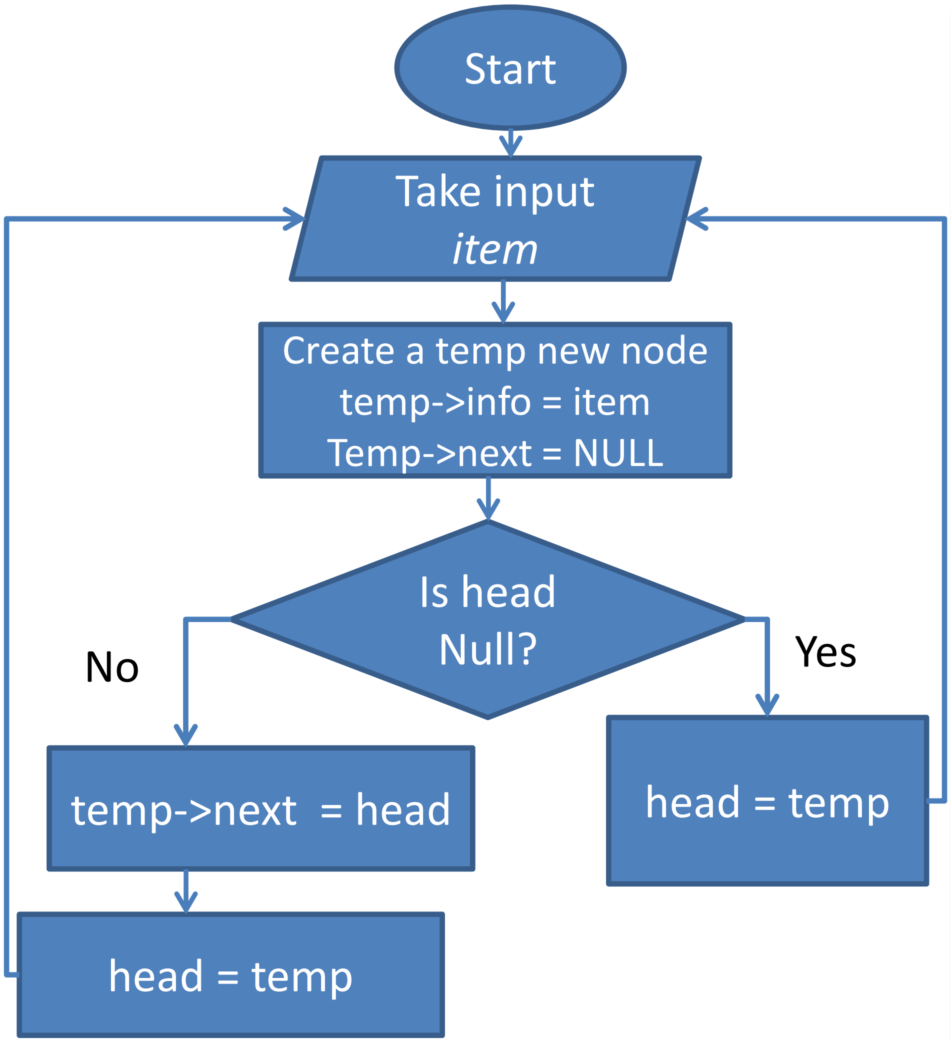
# Coding linked list operations:

- In order to understand/write codes of linked list:
  - it is best to draw example list and write code based on the drawing
  - During the lecture, we will draw and write line by line codes in white board, so that we can process it and relate them with the picture.
  - In the slide, I have just provided the basic codes without any comments due to space. However, they will be explained during the lecture as mentioned above.
  - However, most of the codes with detailed comments will be uploaded on webcourses.

# Inserting at the Beginning

- There can be many scenario when you might need to insert the node in the front of the list.
  - There can be two situations before insertion
    - The list might be empty. How would you know?
      - Who will be the head after insertion?
    - Or there might be existing node(s) in the list.
      - Who will be head now?
      - Who will be after the head?
      - Let's see in the next slides

# Inserting at the Beginning

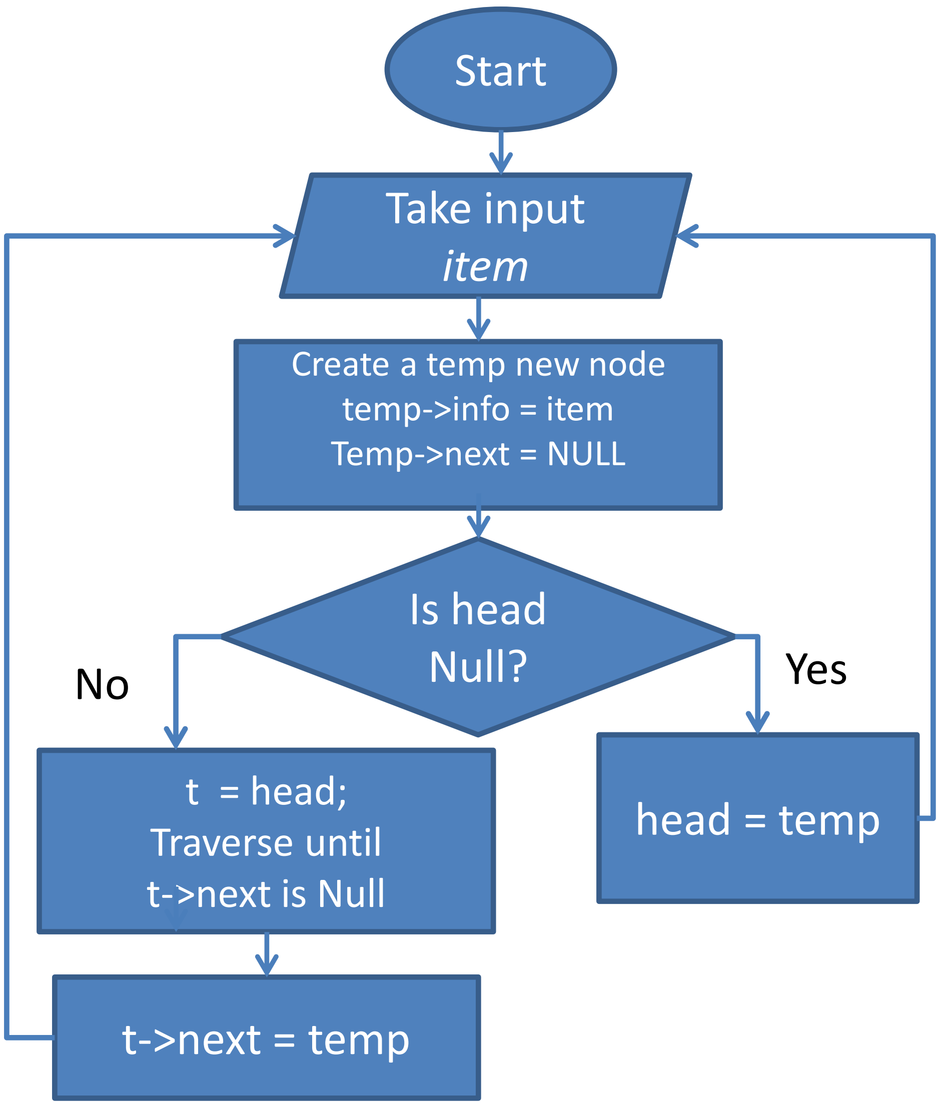



```
Node* insert_beginning(node *head, int
item)
{
      node *t;
       node *temp;
       temp=(node *) malloc( sizeof(node));
       temp->info=item;
       temp->next=NULL;
        if(head==NULL)
                 head=temp;
         else
          {
                    temp->next = head;
                     head = temp;
           }
          return head;
}
```

# Inserting at the End

- There can be many scenario when you might need to insert the node in the end of the list.
  - There can be two situations before insertion
    - The list might be empty. How would you know?
      - Who will be the head after insertion?
    - Or there might be existing node(s) in the list.
      - Who will be head now?
      - Who will be after the head?
      - Let's see in the next slides

# Inserting at the End




```
node* insert_end(node *head, int item)
{
        node *t;
        node *temp;
        temp=(node *) malloc( sizeof(node));
        temp->info=item;
        temp->next=NULL;
        if(head==NULL)
            head=temp;
        else
        {
            t=head;
            while(t->next!=NULL)
                t=t->next;
            t->next=temp;
        }
        return head;
}
```

#Now we will see a code example to see how they are implemented

#The code is available in the webcourses.

*“SinglyLinkedListInsert_Delete.c”*

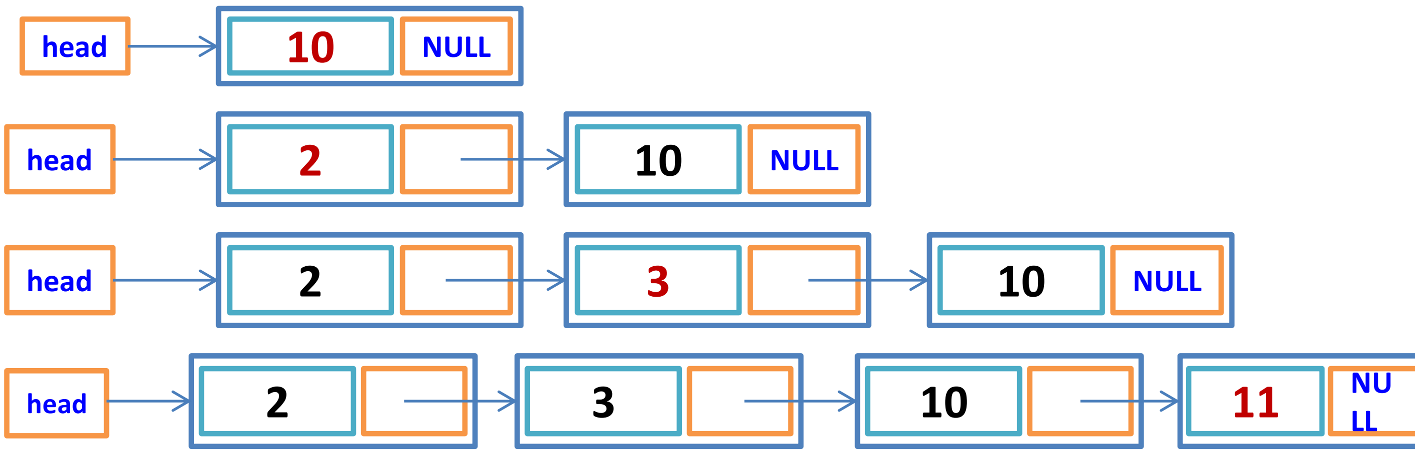
# Inserting Between Nodes

- There can be many scenarios where you might need to insert the node between nodes of the list.
- Example: *Sorted linked list*
- In this case, you might need to:
  - Insert in the beginning or front (if the list is empty or the item is smallest)
    - We have seen how to deal with this
  - Or at the end (the item is largest)
    - We have seen how to deal with this
  - Or between nodes
    - Who will get affected by this operation?
- Remember:
  - Still there can be case that your list might be empty:
    - Who will be the head after insertion?
    - Or there might be existing node(s) in the list.
  - And always take care of your head with special conditions

# Inserting Between Nodes

- **Use case: Sorted Linked List.**
- Example of sorted linked list insertion:

a. Insert: 10
b. Insert: 2
c. Insert: 3
d. Insert: 11




- See from the above examples, there can be situations where you might need to insert in the beginning, or to the end, or in between.
- But, all of these insertion is conditional!
  - It means where to insert, it depends the item you are inserting, and the items you already have in the linked list

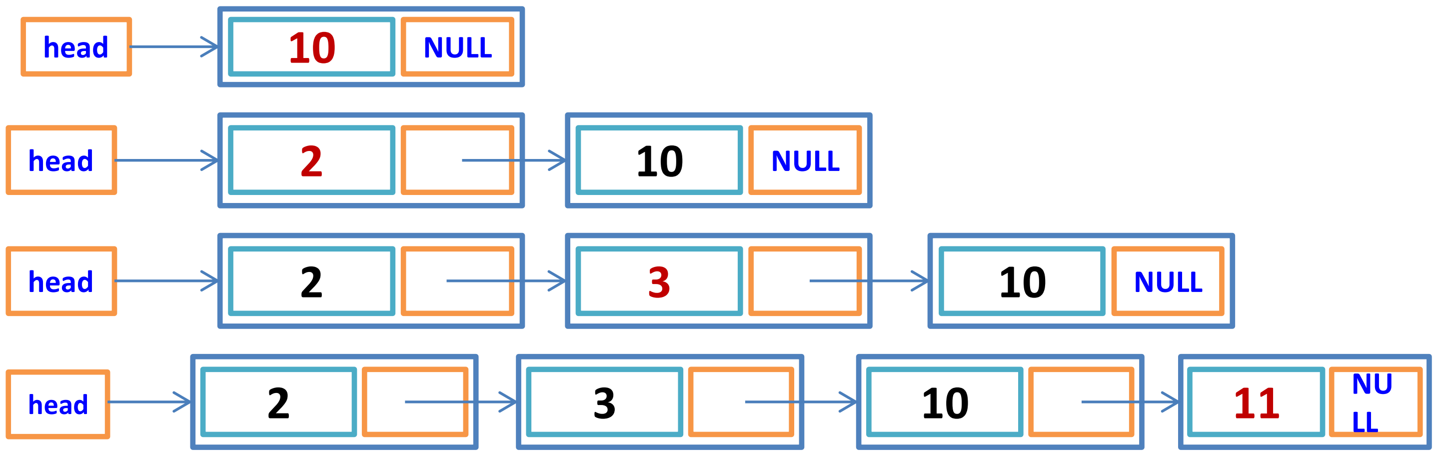
# Inserting Between Nodes

a. Insert 10

b. Insert 2

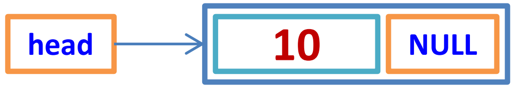
c. Insert 3

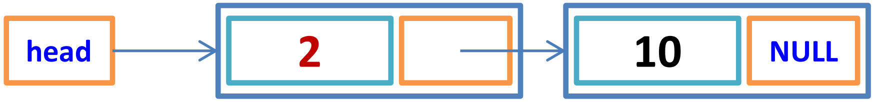
d. Insert 11



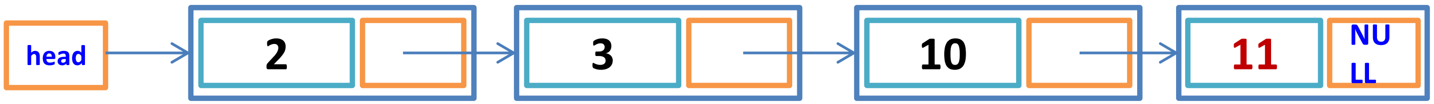

- So, while inserting in a sorted linked list, we have to find a position for the node we want to insert.
- As head is always special, can you guess at what scenario you will need to insert the node in the head in the sorted linked list?
- There can be two situations to insert a node in the head:
  - **Either head is null** (example a) or **head's item is greater than item** (example b)
  - See example a and b above.
  - So, just translate it to code:
- If head == NULL or head->info >= item
  - Insert in the beginning. (Example a and b above)
  - You should already know how to insert in the beginning

# Inserting Between Nodes

a. Insert 10 


b. Insert 2 


c. Insert 3 


d. Insert 11 


- Now, if we find out that the item should not be inserted in the head, what would be the next step?
- **We need to traverse the linked list to find the appropriate place.**
- Now, how long should we travers and how would you know that it is an appropriate place?
- There can be two reason to stop traversing:
    1. Either you find out that you have reached to the end of the linked list, because none of the items are bigger than the item you want to insert (example d above)
    2. Or you find out a node that has larger info than your item (Example c above).
- For case 1, we will stop at the node with 10 (in example d) and then just join our temp node after that. (linking 11 after 10)
- However, for case 2, we have to stop before the node with larger number as we cannot come back if we jump there. So, we look ahead.
    – For example, in example c, we have to stop at 2 so that we can join 3 after 2.

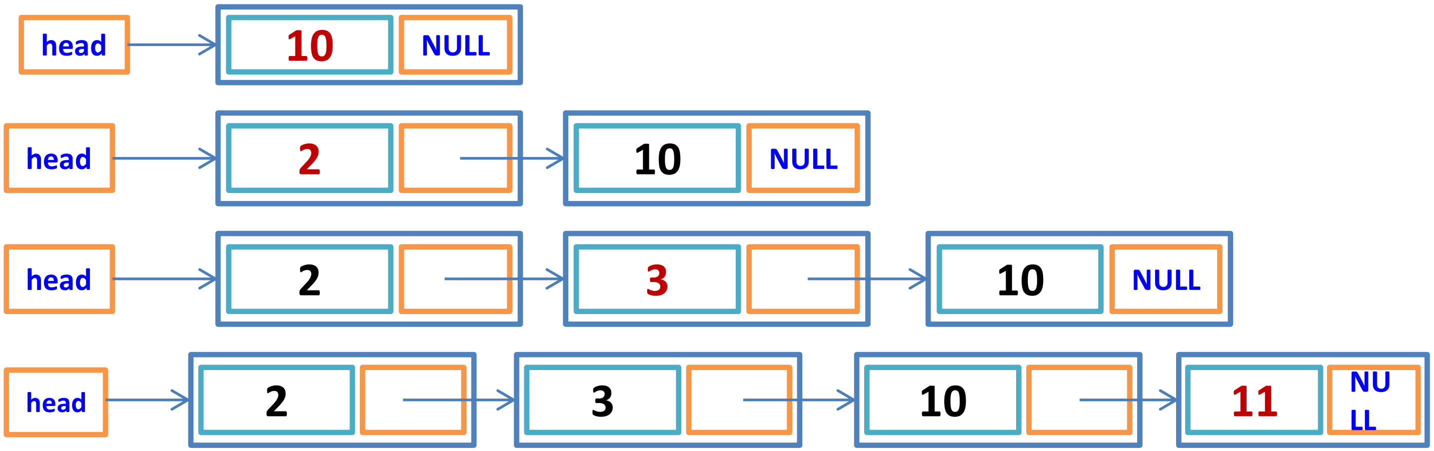
# Inserting Between Nodes

a. Insert 10

b. Insert 2

c. Insert 3

d. Insert 11




- So, for the scenario c and d above, the traversal will be like this:

```
t = head;
while (t->Next != NULL && t->next->info <item)
    t = t->Next;
```

- Now after this loop, t stops exactly where we wanted it to stop:
  - At 2 for inserting 3 ( example c)
  - And at 10 when inserting 11 (example d)
- Now, how can we join our temp node after them?
- Temp->next will be t->next
- And t-next will be temp

```
temp->next = t->next;
t->next = temp
```

# Activity

## Write *SortedInsert(node* head,int item)* function

Hints from previous slides

- If head is NULL or head->info >= item
  - Insert in the beginning.
- How long should we traverse? Example c and d

```
t = head;
while (t->Next != NULL && t->next->info <item)
    t = t->Next;
```

Insert *temp* after finding the position:

```
temp->next = t->next;   //for last node, temp->next will be NULL automatically as t->next was NULL
t->next = temp
```

# Activity

## Write *SortedInsert(int item)* function

Hints from previous slides

- If head is NULL or head->info >= item
  - Insert in the beginning.
- How long should we traverse? Example c and d

```
t = head;
while (t->Next != NULL && t->next->info <item)
    t = t->Next;
```

Insert *temp* after finding the position:

```
temp->next = t->next;
t->next = temp
```

```
node* Sort_insert(node* head, int item)
{
  Node *temp;
  Node *t;
   temp= (node *) malloc(sizeof(node));
   temp->info=item;
   temp->next=NULL;
   if (head==NULL || head->info >=item
   {
       temp->next = head;
       head = temp;
   }
    else
    {
       t = head;
      while (t->Next != NULL && t->next->info <item)
            t = t->Next;

      temp->Next = t->Next;
       t->next = temp
    }
    return head;
  }
```

# Delete operation

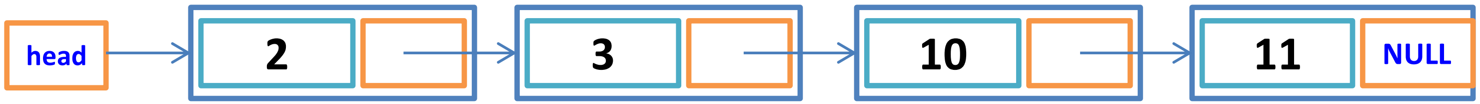
- You might need to delete a node :
  - from the front of the list
  - from the end of the list
  - between nodes of the list
- In case of stack and queues that we will be learning in next couple of lectures, they are by default deleted from the head or tail due to their nature
- However, most cases we would like to delete a particular item from the list.

# Delete Operation

- Many cases you will want to delete a specific node by searching.
  - So: for deleting, you have to search the node containing the item
  - The node can be in the beginning, *// how would you know that your item is in the beginning?*
    - *head->data == your item*
  - or in the end, // how would you know that?
    - If the node you want to look for has the next as NULL.
  - or in between nodes.
  - While searching for the node, do not jump to the node you want to delete while traversing.
    - Because you want to delete it and how would you join the previous node with the next one as you cannot go back?
      - So, have a look to the data of next node before going there.

# Delete operation

- Deletes an item from the linked list




- Delete 2:

```
temp = head;
head = head->next;
free(temp);
return head;
```

- Delete 12:
- Delete 3:

```
t = head;
while (t->next != NULL && t->next->info != item)
     t = t->next;
If(t->next == NULL ) return head; //item was not found
temp = t->next;
t->next = t->next->next;
free(temp)
return head;
```

We have to check the reason of exiting the loop.
If the loop exits for t->next == NULL , then it indicates the item was not found in the list

-In the above illustration, the colors used in the code are matched with the example in the left side

-Now write the function DelList(node* head, int item)

# Full Delete Function

```
node* DelList(node* head, int item)
{
        node *t;
        node *temp;
        if(head==NULL)
                return head;
        if(head->info==item)
        {
                temp=head;
                head=head->next;
                free(temp);
                return head;
        }

        t=head;
        while(t->next!=NULL && t->next->info != item)
                t=t->next;
        if(t->next==NULL)
                return head;
        temp=t->next;
        t->next=t->next->next;
        free(temp);
        return head;
}
```

# How would you implement Search operation?

- Didn’t you search the item while deleting?
- Searching is just like traversing the linked list until you find the item you are looking for (if item exists) or until you reach to the end of the linked list (not found).

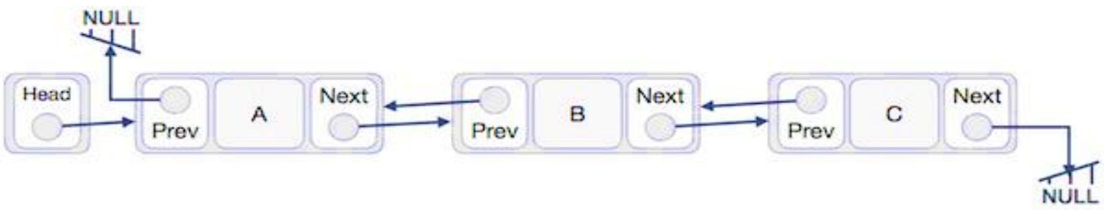
# Doubly Linked List

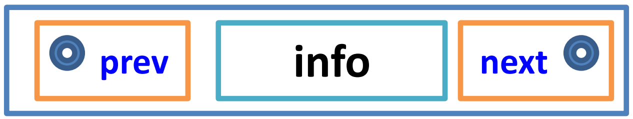
# Doubly Linked List

- In case of singly linked list:
  - Can you go back while traversing?
  - What will happen if you write head = head->next?
    - You can’t go back to the head
- In doubly linked list: you can go both forward and backward

- **Application Scenario of doubly linked list:**
- A music player which has next and prev buttons.
- The browser cache which allows you to hit the BACK-FORWARD pages.
- Applications that have a Most Recently Used list (a linked list of file names)
- Undo-Redo functionality

# Defining a Node



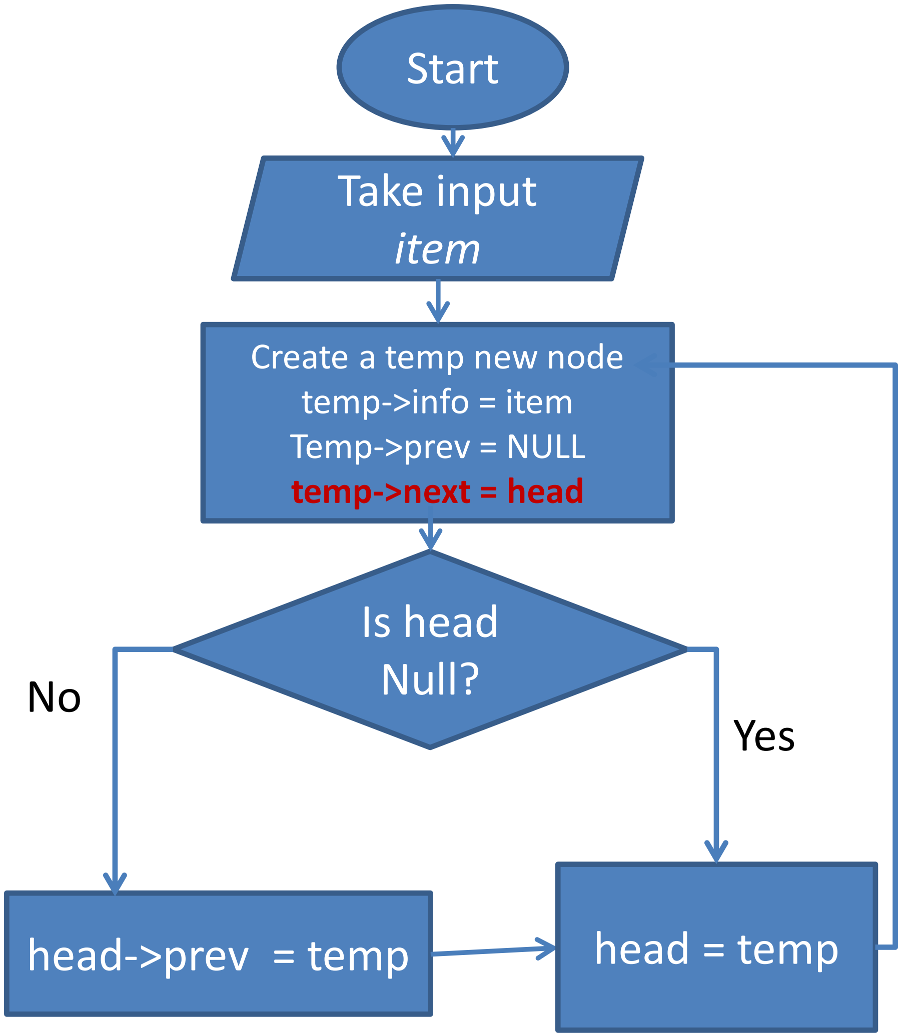




**Node**

```
typedef struct nod
{
    int info;
   struct nod *prev, *next;
} node;
```

- Info holds the data
- Prev pointer is used to point to previous node
- Next pointer is used to point to the next node

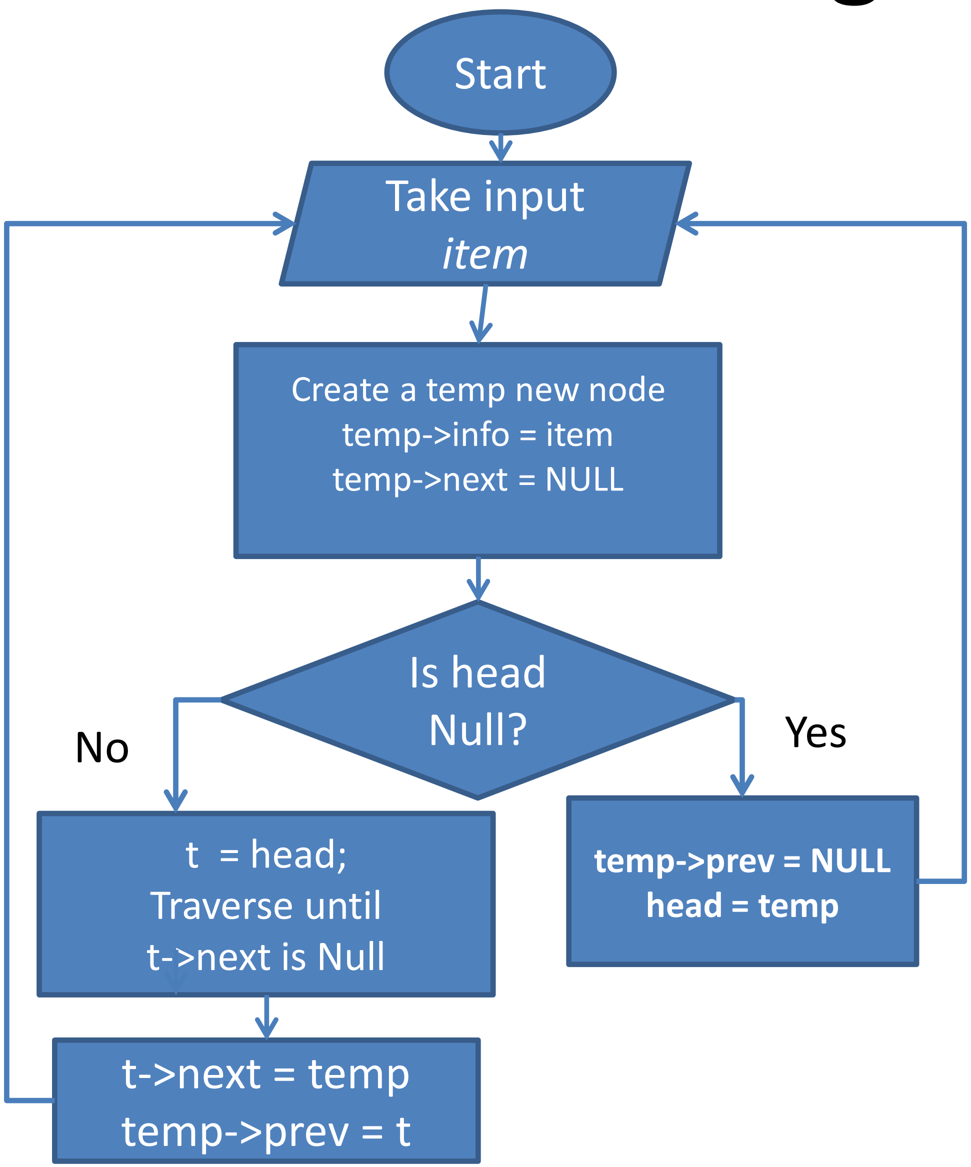
# Inserting at the Beginning




```
node* insert_beg(node* head, int item)
{
        node *t;
        node *temp;
        temp=(node *) malloc( sizeof(node);
        temp->info=item;
        temp->prev=NULL;
        temp->next=head;
        if(head != NULL)
            head->prev = temp;
        head = temp;

        return head;
}
```

# Inserting at the End




```
node *head;
node* insert_end(node *head, int item)
{
        node *t;
        node *temp;
        temp=(node *) malloc( sizeof(node));
        temp->info=item;
        temp->next=NULL;
        if(head==NULL)
        {
            temp->prev = NULL;
            head=temp;
        }
        else
        {
             t=head;
             while(t->next!=NULL)
                     t=t->next;
              t->next=temp;
            temp->prev = t;
        }
        return head;
}
```

# Inserting Between Nodes

- Like singly linked list, if you want to create a sorted doubly linked list, you might need to insert it in the beginning or end or between nodes.
  - It requires extra management for prev pointer.
- As always create a new temp node and fill-up the fields.
- Traverse where to insert
- Assign *prev* and *next* of the temp node based on t and t->next
- Adjust *next of t*
- Adjust *prev* of *t->next* (if *t->next is not NULL*)

# Exercise

- Write the SortedInsert operation for Doubly Linked List
  - The hints already provided in previous slide
  - Also take help from SortedInsert function of Singly Linked List.

# Delete operation

#similar to singly linked list, with some additional condition to adjust prev

Example of deleting 2


Example of deleting 15 (item does not exist)

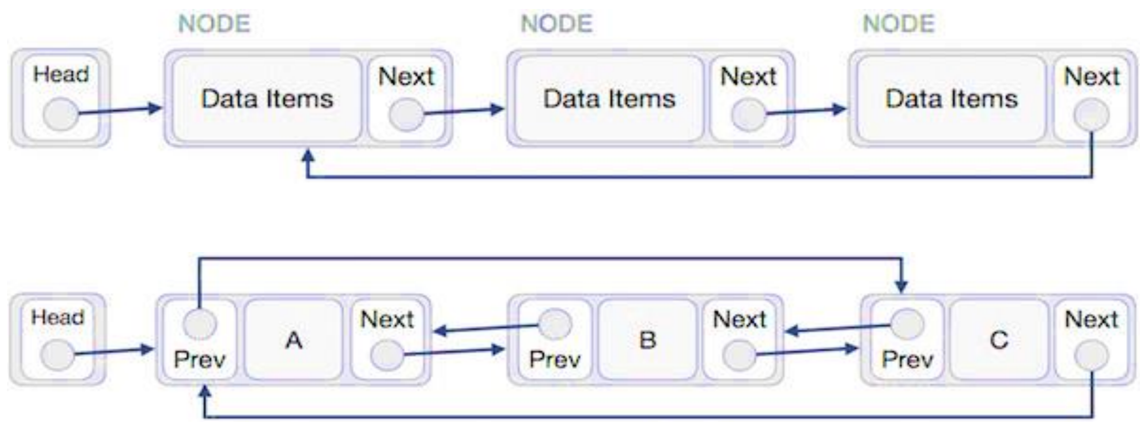
Deleting 3 or 10

If you delete 10, this will be false as t->next is null.

```
node* DelListDoubly(node* head, int item)
{
        node *t;
        node *temp;
        if(head==NULL)
                return head;//nothing to do
        if(head->info==item)  {
                temp=head;
                head=head->next;
                if (head != NULL) //new condition for doubly
                    head -> prev = NULL;
                free(temp);
                return head;
        }
        t=head;
//traverse until reach to the end or find the item in next node.
        while(t->next!=NULL && t->next->info != item)
                t=t->next;
        if(t->next==NULL)
                return head; //not found, skip
        temp=t->next;
        t->next=t->next->next;
        if (t->next)          //new condition for doubly
            t->next->prev = t;
        free(temp);
        return head;
}
```

# Circular Linked List

- **Circular Linked List** - Last item contains link of the first element as next and the first element has a link to the last element as previous.
- Applicable for both singly and doubly linked list




- For a circular linked list:
  - How would you know you are in the end of the list?
  - The last element should point to the head (instead of NULL)

# Summary

- Linked List
- Linked List Operations
- Sorted Linked List
- Doubly linked list
- Circular linked list
- During the lecture we explained most of the codes by scratching in papers.
  - Scratching various situations would really help to map them to code.

Question?

Thank you ☺