COP 3502 – Computer Science 1

Lecture 04

Dr. Sahar Hooshmand
sahar.hooshmand@ucf.edu

Department of Computer Science

Slides modified from Dr. Ahmed, with permission

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
- **Second most-used** data structure after array
- Example representation of a Linked List:

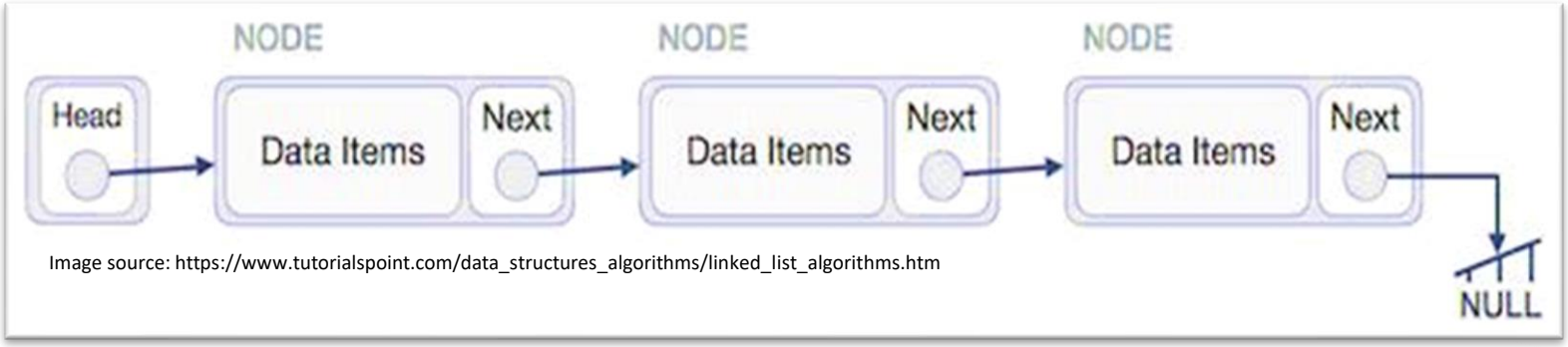


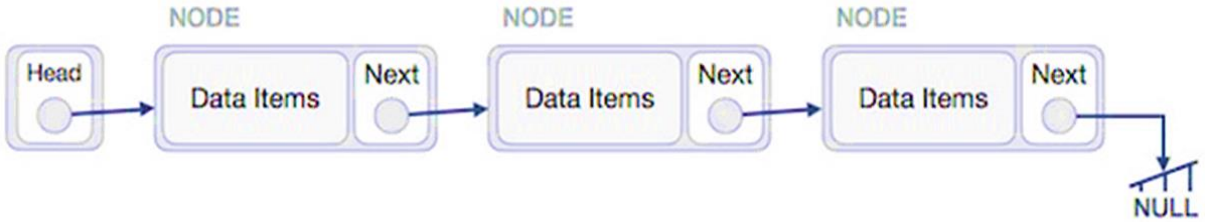
Image source: https://www.tutorialspoint.com/data_structures_algorithms/linked_list_algorithms.htm

Why linked list?

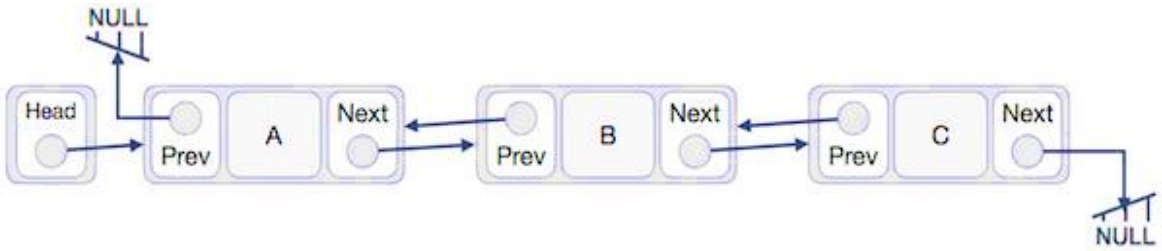
- Why not just use an array?
  - Each node in an array is stored, physically, in contiguous spaces in memory
  - Arrays are fixed size (not dynamic)
  - Inserting and deleting elements is difficult
  - If you have an array of size 1000 and if we want to insert an element after 5^{th} element, all the 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

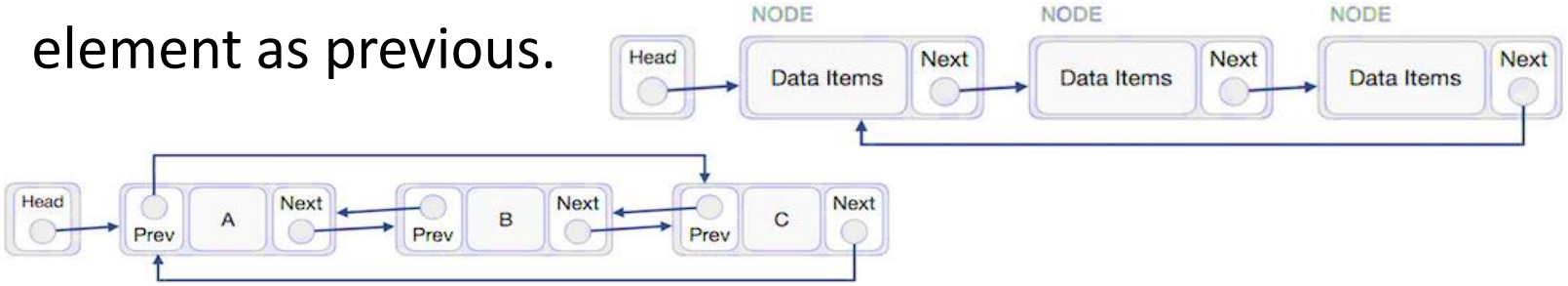
- **Simple/Singly Linked List** - Item navigation is forward only



- **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;
};
```

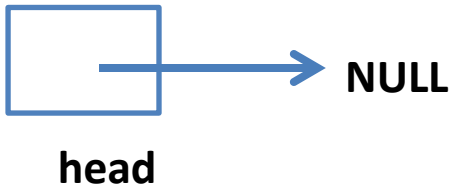
- The above node has 3 fields for data
- The right side example, has Book as data/info in the linked list's node

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

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

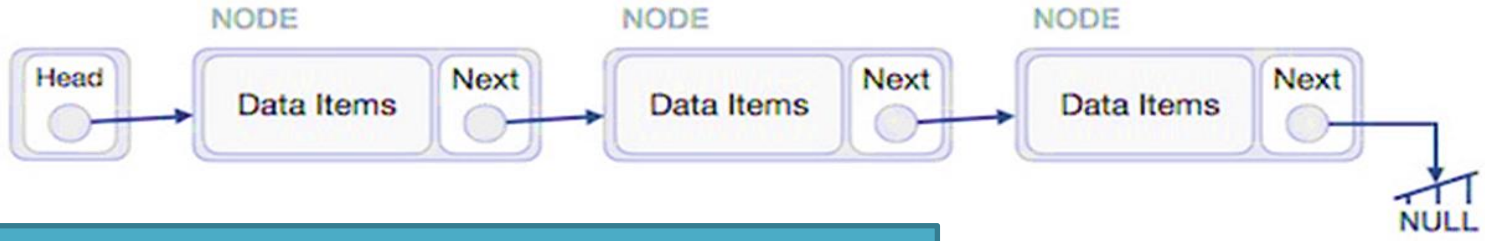
Head of the list

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

- Can you think how to display the info in the Linked List?
- Just add the following statement in the loop:

  printf ("%d ", t->info);

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!

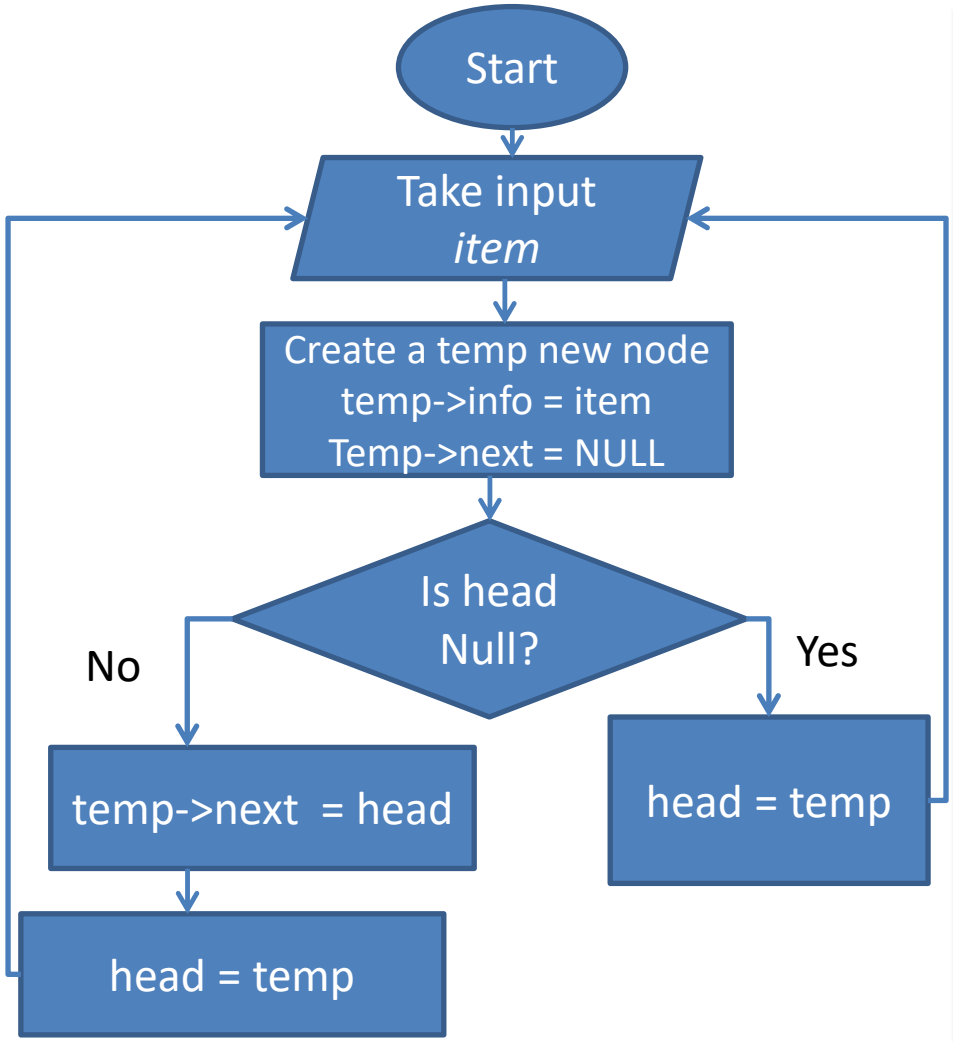
Coding linked list operations:

- 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

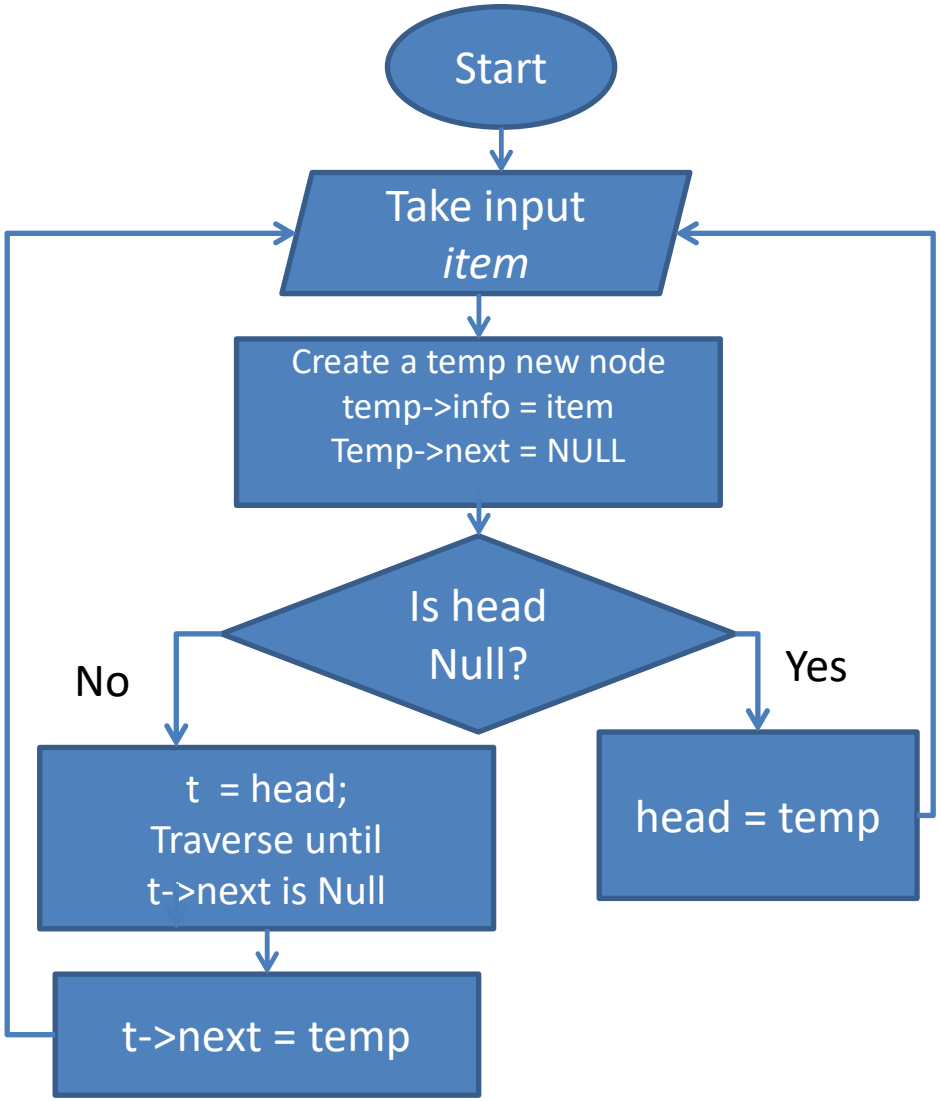



```
Node* insert_beginning(node *head, int
item)
{
      node *t;
       node *temp;
       temp=(node *) malloc( sizeof(node);
       temp->info=item;
       temp->next=NULL;
        if(head==NULL)
                head=temp;
         else
          {
                  temp->next = head;
           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