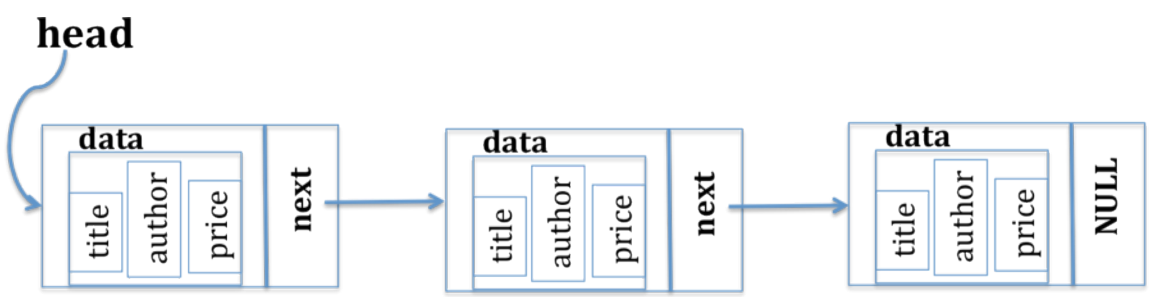
CSCI 2170 Linked List (1)

1. Define the basic structure to build a linked list:

```
struct   BookStruct
{
    string    title;
    string    author;
    float     price;
};

struct   Node
{
        BookStruct   data;
        Node*        next;
};
typedef Node* NodePtr;
```

2. Examine a linked list of N nodes:



- The 1st element in the list is special. Its name is "head". It is of type NodePtr, not Node
    - NodePtr head;
    - It is the only name by which the list nodes may be accessed
    - When the list is empty, i.e., when the list is first created and no node has been inserted into the list, designate head to be NULL
        - head=NULL;
    - (head==NULL) is a condition we can use to test whether the list is empty
    - (head != NULL) is a condition we can use to test that the end of the list has not been reached

- The ***next*** field of a node contains the memory address of the next node in the list
    - Important!! -- that is how the nodes are linked together
    - The next field of the last node in the list has value NULL
        - It provides a way of detecting the end of the list

- **Advantages** of using linked list, instead of array, to store data:
    - Memory efficiency → exact amount of memory is allocated for the data

- Time efficiency → insertion into and deletion from a list are more efficient

3. How to create a linked list of data items?

<u>For simplicity, the data will simply be an integer number in the following discussion:</u>

```
struct   Node
{
        int   data;
        Node* next;
};
typedef Node* NodePtr;
```

a. create a linked list with 3 nodes to store contact information of three person

```
NodePtr    cur = new  Node;      // create the first node
if (cur != NULL)
{
        cur→data=5;
        cur→next = NULL;
}
head = cur;   // linked list with a single node. Head pointer is pointing to the node

// create the second node for insertion
NodePtr    cur = new Node;
if ( cur != NULL)
{
        cur→data = 9;
        cur→next = NULL;
}

cur→next = head;  // linked the two nodes together by putting the new node
head = cur;                // at the beginning of the list, head is updated to point
cur = NULL;              // to the new "head" of the list
```

practice: create the 3rd node (with a value 100) and put it at the beginning of the list (how about at the end of the list? or in the middle of the list?)

4. **Traversing the list (starting from the head of the list, visit the nodes in the list one by one)**
   a. **print out the information in the list**

```
NodePtr   curr=head;
while (curr!=NULL)            // stops when the next field of the last
{                             // node in the list is reached.

     cout << curr→data << endl;

     curr= curr→next;   // important! This is how to get from one
}                       // node to the next node
```

   b. **Given a list of N nodes, print out the information of the node at position "*position*"**

```
NodePtr curr=head;
int  i=1;
while (curr !=NULL && i<position)  // detecting end of list should
{                            // always be the first condition because of
                                  // C++ "short circuit evaluation"
     curr = curr→next;
     i++;
};
if (curr!=NULL)
   cout << curr→data;
```

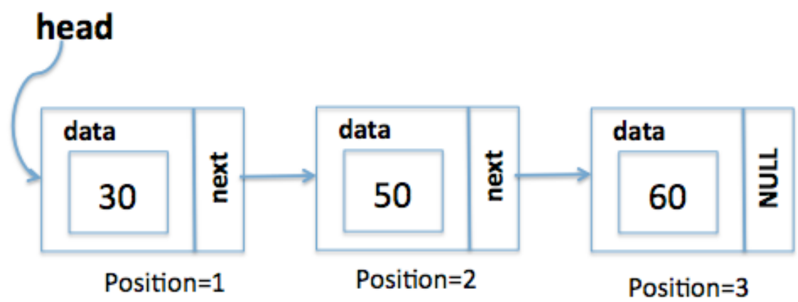
   c. **Given a list of n nodes, search for a specific number (linear search)**

```
bool found=false;
NodePtr  curr=head;
while (curr !=NULL)
{     if (curr→data == 17)
      {     cout << "17 is found"<< endl;'
            found=true;
      }
      curr=curr→next;
}
if (!found)
   cout << "the value is not in the list" << endl;
```

practice :

(1) how to print the position of the item in the list if the item is found?

(2) how to print out the content of the last node in the list?



d. insert a node at position "position" in the list in "unsorted list" (This function should be to handle ALL possible situations)

Two cases: Case 1: position == 1 → insert at the beginning of list
Case 2: position != 1 → insert in the middle or end of list

Step1: create a new node, assign proper values to the new node
newNode = new Node
newNode→data = newData
newNode→next = NULL

Step2: if the new node is to be added at the beginning:
Step 2a: newNode→next = head
Step 2b: head = newNode;
Question: Does it take care of empty list situation?

Step 3: if the new node is to be added in the middle or at the end:
Step 3a: traverse down the list and find the insertion point

```
curr = head;
prev = head;
count = 1;
while (curr!=NULL && count != position)
{
        prev= curr;
        curr = curr→next;
        count++;
}
```

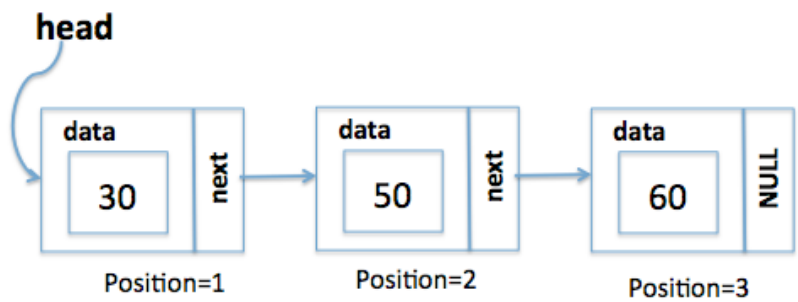
What are the points curr and prev point to at the end of the loop?

Step 3b: at this point, **curr** points at the position of insertion, **prev** points to the node right before the insertion location.

Insert the node by:

```
newNode→next = curr;
prev→next = newNode;
```

Step 4: update the size of the list.
Question: Does this work for end of list insertion?

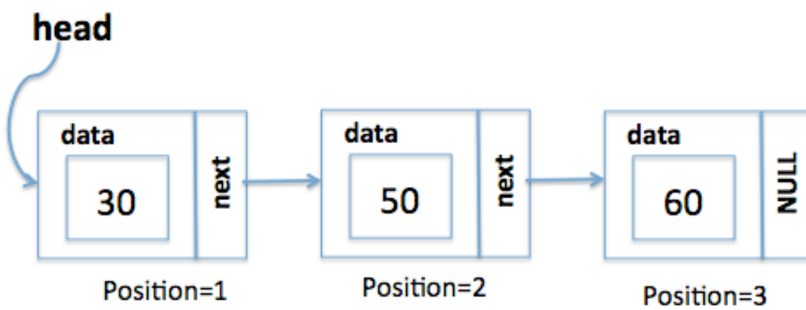


e. **What if the list is sorted? Assuming the list is sorted in ascending order, how to insert a node with *value 40* into the list at the appropriate spot in the list?**
(This time, we assume that we don't know ahead of time what is the correct position for this value, it is to be determined by the code itself)

```
Step 2:  decide if the list is empty
        if (head == NULL)
                head = newNode
        else if (40 < head→data) // add the newNode as the new head
        {
                … // same as (2.4) Step 2
        }

Step 3a:
        while (curr!=NULL && 40<curr→data)
        { …
        }
```

Does this code handle the situation where we want to insert a value 15? Or insert a value 75?



f. delete a node at position "position" in the list

two cases: (1) delete from the beginning → change the value of "head"
(2) delete from the middle or from the end of list → list traversal

Step 1: case 1 – position is 1

Detach first node from the list, update "head" value

```
cur = head;
head = head→next;
cur →next = NULL;
delete cur;
cur = NULL;
```

Step 2: case 2 – position is not 1, so:

Step 2a traverse down the list and find the insertion point

```
cur = head;
prev = head;     // why not: prev=head->next;   ??
count = 1;
while (cur !=NULL && count != position)
{
        prev= cur;
        cur = cur→next;
        count++;
 }
```

Step 2b: at this point, **cur** points at the position of deletion, **prev** points to the node right before the deletion location.

delete the node by: detach and relink

```
prev→next = cur→next;
cur→next = NULL;
delete cur;
cur= NULL;
```

Step 3: release the node

Step 4: Update the size of the list

g. **delete a node with *data* equal to 50.**

Step 2a : while (curr!=NULL && curr→data !="Mary")

Step 2b: add one more case: "Mary" not in list

```
if (curr !=NULL)
{
    …  // same as in (2.5) step 2b
}
else
      cout << "This person not in list";
```

h. Make a copy of an entire list – deep copy vs. shallow copy
(Copy constructor of a listclass with pointer implementation)

i. Delete an entire list
(Destructor of a listclass with pointer implementation)