Linked Lists

List Overview

- Basic operations of linked lists

Insert, print, find, delete, etc.

- Variations of linked lists
  - Circular linked lists
  - Doubly linked lists

malloc and calloc

- The two primary memory allocation operations in c are malloc and calloc
  - For most situations, we will use malloc and calloc:
    - pointer = (type *) malloc(sizeof(type));
    - pointer = (type *) calloc(n, sizeof(type)); // n is the size of the array
    - free(pointer)
  - C++ has a simpler syntax
    - pointer = new Type;
    - pointer = new Type[n];
    - delete pointer;
    - delete[] pointer
  -

calloc example

```
#include <stdio.h>
#include <stdlib.h>                        // needed for calloc

void main()
{
        int i;
        int *x, *y;                        // two pointers to int arrays
        x = (int *) calloc(10, sizeof(int));      // x now points to an array of 10 ints
        for(i=0;i<10;i++) x[i] = i;               // fill the array with values
        …
         free(x);
}
```

calloc example in C++

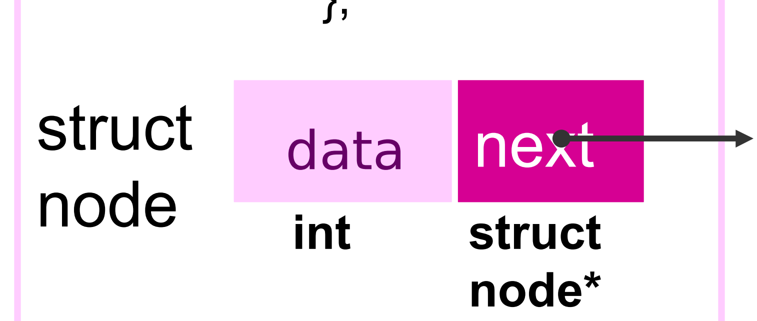
```
#include <stdio.h>
#include <stdlib.h>                               // needed for calloc

void main()
{
          int i;
          int *x, *y;                             // two pointers to int arrays
          x = new int[10];     // x now points to an array of 10 ints
          for(i=0;i<10;i++) x[i] = i;                    // fill the array with values
          …
           delete [] x;

}
```

Linked Lists

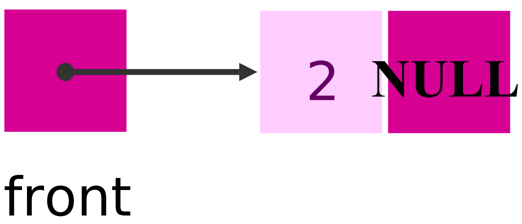
struct node* **front=NULL;**

- A *linked list* is a series of connected *nodes*
- Each node contains at least
  - A piece of data (any type)
  - Pointer to the next node in the list
- front: pointer to the first node
- The last node points to `NULL`

```
struct node {
   int data;
   struct node
*next;
};
```



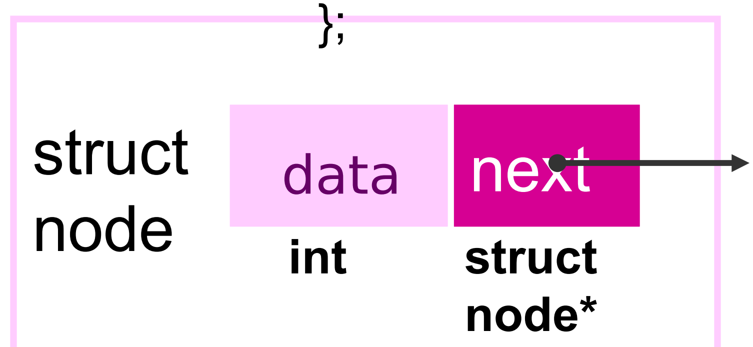
Linked Lists



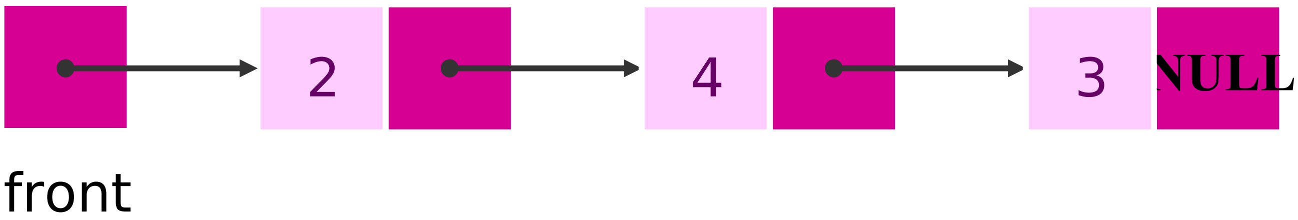

```
front=(struct node)
malloc(sizeof(struct node));
front->data=2;
front->next=NULL;
```

- A *linked list* is a series of connected *nodes*
- Each node contains at least
  - A piece of data (any type)
  - Pointer to the next node in the list
- front: pointer to the first node
- The last node points to `NULL`

```
struct node {
   int data;
   struct node
*next;
};
```

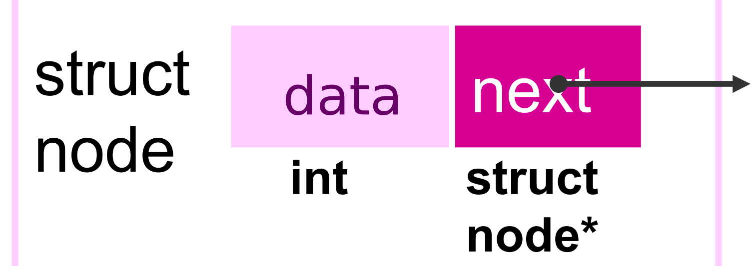


Linked Lists

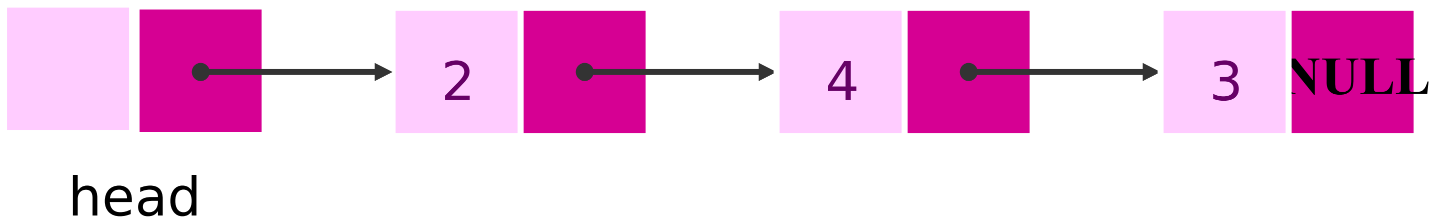


- A *linked list* is a series of connected *nodes*
- Each node contains at least
  - A piece of data (any type)
  - Pointer to the next node in the list
- front: pointer to the first node
- The last node points to `NULL`

```
struct node {
   int data;
   struct node
*next;
};
```

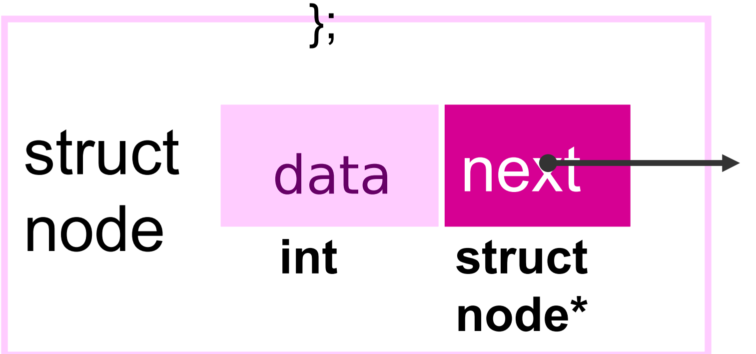


Linked Lists with a dummy head



- A *linked list* is a series of connected *nodes*
- Each node contains at least
  - A piece of data (any type)
  - Pointer to the next node in the list
- front: pointer to the first node
- The last node points to `NULL`

```
struct node {
   int data;
   struct node
*next;
};
```



A Simple Linked List Class

- Operations of `List`

  `CreateList`: create an empty list

  `IsEmpty`: determine whether or not the list is empty

  `DisplayList`: print all the nodes in the list

  `InsertNode`: insert a new node at a particular position

  `FindNode`: find a node with a given value

  `DeleteNode`: delete a node with a given value

Usage

```
struct Node {
      double data;             // data
      Node* next;              // pointer to next
};
Node* CreateList();
bool IsEmpty(Node* listHead);
Node* InsertNode(Node* listHead , int index, double x);
Node* FindNode(Node* listHead, double x);
Node* DeleteNode(Node* listHead, double x);
void DisplayList(Node* listHead);

int main()
{
      Node* head=CreateList();
      DisplayList(head);
      InsertNode(head,0, 10);
      InsertNode(head,0, 20);
      InsertNode(head,0, 30);
      DisplayList(head);
      InsertNode(head,3, 10);
      InsertNode(head,3, 20);
      InsertNode(head,3, 30);
      DisplayList(head);
}
```

Creating an empty list

- `Node * CreateList()`

```
struct Node {
        double data;            // data
        Node* next;             // pointer to next
};
Node* CreateList();

Node* CreateList()
{
      ???


}
```

Creating an empty list

- Node * CreateList()

```
struct Node {
        double data;            // data
        Node* next;             // pointer to next
};
Node* CreateList();
Node* CreateList()
{
        Node* head;
        head=new Node;
        head->next=NULL;
        return head;
}
```

C++

Creating an empty list

- `Node * CreateList()`

```
struct Node {
        double data;            // data
        Node* next;             // pointer to next
};
Node* CreateList();
```

```
Node* CreateList()
{
        Node* head;
        head=new Node;
        head->next=NULL;
        return head;
}
```

C++

```
struct Node* CreateList()
{
        struct Node* head;
        head=(struct Node*)
        malloc(sizeof(struct Node));
        head->next=NULL;
        return head;
}
```

C

Printing all the elements

- `void DisplayList(Node* listHead)`
  - Print the data of all the elements

```
void DisplayList(Node* head)
{
   printf("[ ");

   ???
   printf("]\n");
}
```

Printing all the elements

- `void DisplayList(Node* listHead)`
  - Print the data of all the elements

```
void DisplayList(Node* head)
{
   printf("[ ");
   for(Node* currNode = head->next;
       currNode!=NULL;
       currNode=currNode->next)
          printf("%f, ", currNode->data);
   printf("]\n");
}
```

Is empty?

- `bool IsEmpty(Node* listHead);`

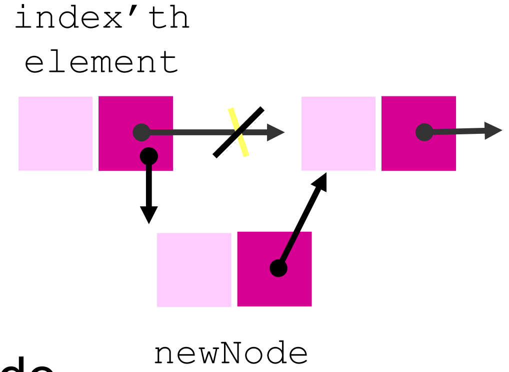
Is empty?

- `bool IsEmpty(Node* listHead);`

```
bool IsEmpty(Node* listHead)
{
        return listHead->next == NULL;
}
```

Inserting a new node

- `Node* InsertNode(Node* head, int index, double x)`
  - Insert a node with data equal to `x` after the `index'`th elements.
    (i.e., when `index = 0`, insert the node as the first element;
    when `index = 1`, insert the node after the first element, and so on)
  - If the insertion is successful, return the inserted node.
    Otherwise, return `NULL`.
    (If `index` is < 0 or > length of the list, the insertion will fail.)



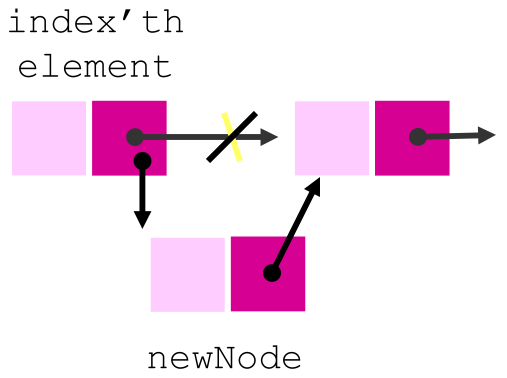
- Steps
  1. Locate `index`'th element
  2. Allocate memory for the new node
  3. Point the new node to its successor
  4. Point the new node's predecessor to the new node

Inserting a new node

- Possible cases of `InsertNode`
  1. Insert into an empty list
  2. Insert in front
  3. Insert at back
  4. Insert in middle
- But, in fact, only need to handle two cases
  - Insert as the first node (Case 1 and Case 2)
  - Insert in the middle or at the end of the list (Case 3 and Case 4)

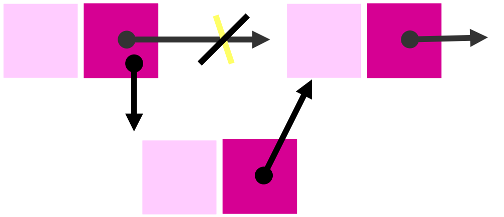
Inserting a new node (assume index = 0)

```
Node* InsertNode(Node* head, int index, double x) {

        Node* newNode  =      new    Node;
        newNode->data  =      x;
        if (index == 0) {
              ???
        }
        return newNode;
}
```



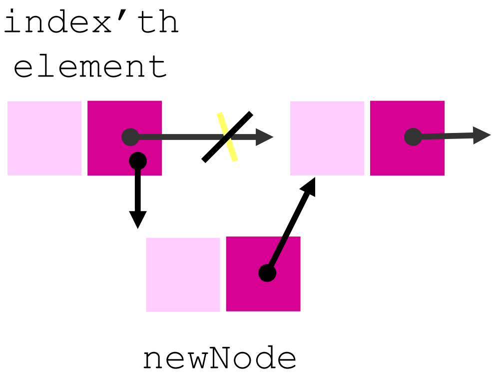
Inserting a new node (assume index = 0)

```
Node* InsertNode(Node* head, int index, double x) {

        Node* newNode  =       new     Node;
        newNode->data  =       x;
        if (index == 0) {
                newNode->next  =       head->next;
                head->next     =       newNode;
        }
        return newNode;
}
```



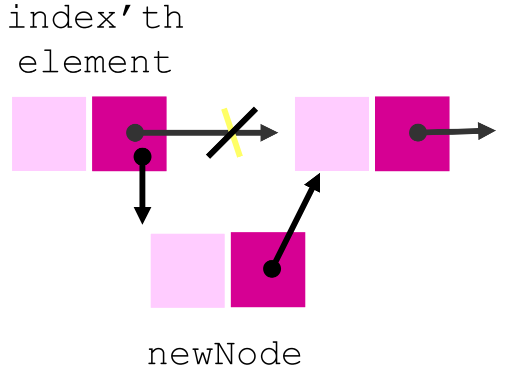
Inserting a new node

```
Node* InsertNode(Node* head, int index, double x) {
        if (index < 0) return NULL;

        Node* newNode  =      new    Node;
        newNode->data  =      x;
        int currIndex=0;
        Node* currNode=head;
         …?

        return newNode;
}
```



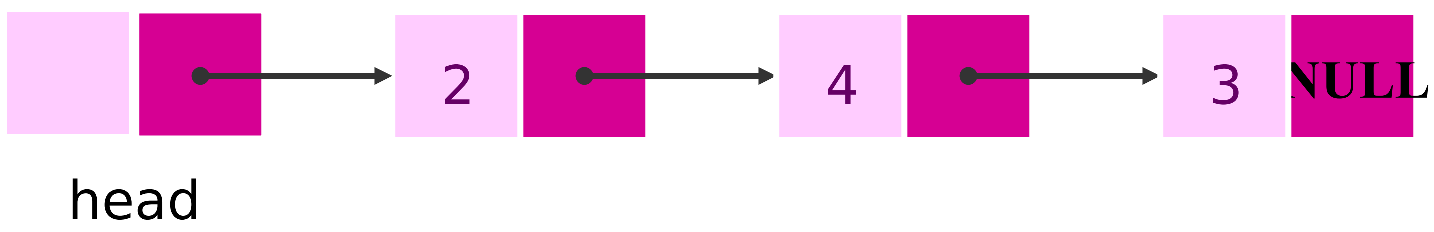
Inserting a new node

```
Node* InsertNode(Node* head, int index, double x) {
        if (index < 0) return NULL;

        Node* newNode  =       new     Node;
        newNode->data  =       x;
        int currIndex;
        Node* currNode;
        for( currNode = head, currIndex = 0;
                currNode && currIndex<index ;
                currNode=currNode->next)
                currIndex++;

        if (currNode == NULL) return NULL;
        newNode->next  =       currNode->next;
        currNode->next =       newNode;
        return newNode;
}
```



Finding a node

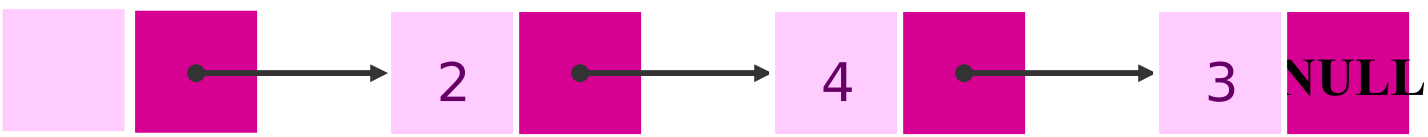
- `Node* FindNode(Node* head, double x)`
  - Search for a node with the value equal to `x` in the list.
  - If such a node is found, return the node. Otherwise, return NULL



printf("%f\n", FindNode(head, 3)->data);

Finding a node

- `Node* FindNode(Node* head, double x)`
  - Search for a node with the value equal to `x` in the list.
  - If such a node is found, return the node. Otherwise, return NULL



head

printf("%f\n", FindNode(head, 3)->data);

```
Node* FindNode(Node* head, double x) {
      Node* currNode    =     head->next;
      while (???)
            currNode    =     currNode->next;
      return currNode;
}
```

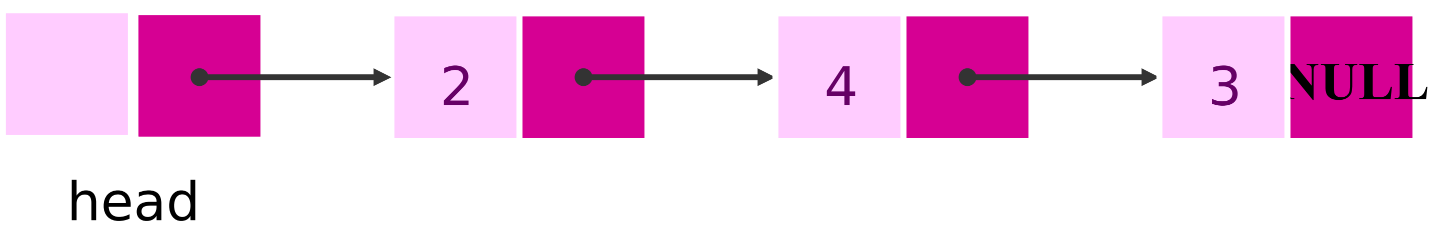
Finding a node

- `Node* FindNode(Node* head, double x)`
  - Search for a node with the value equal to `x` in the list.
  - If such a node is found, return the node. Otherwise, return NULL

```
Node* FindNode(Node* head, double x) {
      Node* currNode    =     head->next;
      while (currNode && currNode->data != x)
            currNode    =     currNode->next;
      return currNode;
}
```

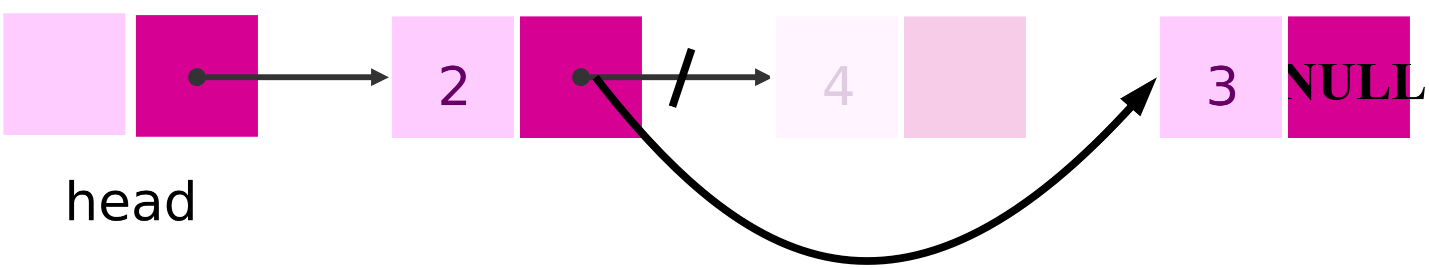
Deleting a node

- `bool DeleteNode(Node* head, double x)`
  - Delete a node with the value equal to `x` from the list.
  - If such a node is found, return true. Otherwise, return false.
- Steps
  - Find the desirable node (similar to `FindNode`)
  - Release the memory occupied by the found node
  - Set the pointer of the predecessor of the found node to the successor of the found node



Deleting a node

- `bool DeleteNode(Node* head, double x)`
  - Delete a node with the value equal to `x` from the list.
  - If such a node is found, return true. Otherwise, return false.
- Steps
  - Find the desirable node (similar to `FindNode`)
  - Release the memory occupied by the found node
  - Set the pointer of the predecessor of the found node to the successor of the found node

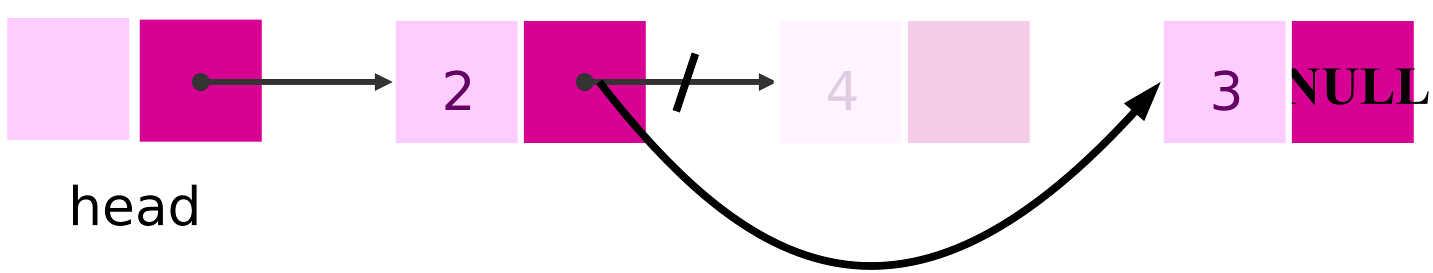


DeleteNode(head, 4);

Deleting a node

```
bool DeleteNode(Node* head, double x)
{
        Node* currNode = head->next;
        Node* prevNode  = head;
        while (currNode && currNode->data != x)
        {
               prevNode       =       currNode;
               currNode       =       currNode->next;
        }
        if (currNode) {
               ???
               return true;
        }
        return false;
}
```

Try to find the node with its value equal to `x`

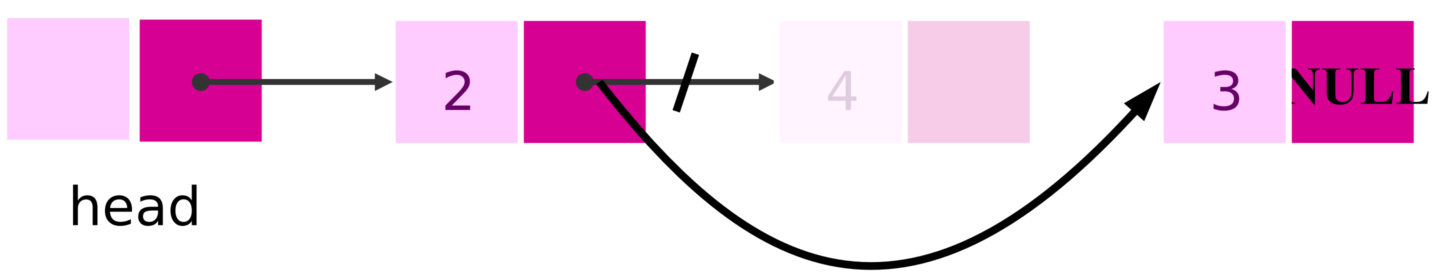


DeleteNode(head, 4);

Deleting a node

```
bool DeleteNode(Node* head, double x)
{
       Node* currNode = head->next;
       Node* prevNode  = head;
       while (currNode && currNode->data != x)
       {
              prevNode       =      currNode;
              currNode       =      currNode->next;
       }
       if (currNode) {
              prevNode ->next = currNode->next;
              delete currNode;
              return true;
       }
       return false;
}
```

Try to find the node with its value equal to `x`



DeleteNode(head, 4);

Destroying the list

- `void DestroyList(Node** pListHead)`
  - Use the destructor to release all the memory used by the list.
  - After destroying the list, set the head as NULL

```
void DestroyList(Node** pListHead)
{
      Node* head=*pListHead;
      Node* currNode = head, *nextNode = NULL;
      while (currNode != NULL)
      {
              ???
      }
      *pListHead=NULL;
}
```

Destroying the list

- `void DestroyList(Node** pListHead)`
  - Use the destructor to release all the memory used by the list.
  - After destroying the list, set the head as NULL

```
void DestroyList(Node** pListHead)
{
      Node* head=*pListHead;
      Node* currNode = head, *nextNode = NULL;
      while (currNode != NULL)
      {
            nextNode    =     currNode->next;
            // destroy the current node
            delete currNode;
            currNode    =     nextNode;
      }
      *pListHead=NULL;
}
```

Using List

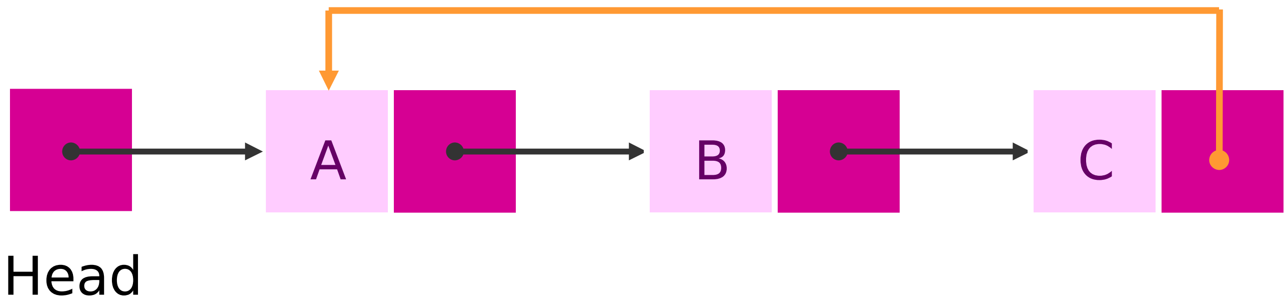
result

[6.0, 7.0, 5.0]
5.0 found
4.5 not found
[6.0, 5.0]

```
#include <iostream>
using namespace std;
int main(void)
{
        Node* list=CreateList();
        InsertNode(list, 0, 7.0);  // successful
        InsertNode(list, 1, 5.0);  // successful
        InsertNode(list, -1, 5.0); // unsuccessful
        InsertNode(list, 0, 6.0);  // successful
        InsertNode(list, 8, 4.0);  // unsuccessful
        // print all the elements
        DisplayList(list);
        if(FindNode(list, 5.0) )   cout << "5.0 found\n" ;
        else                       cout << "5.0 not found\n" ;
        if(FindNode(list, 4.5) ) cout << "4.5 found\n" ;
        else                       cout << "4.5 not found\n";
        DeleteNode(list, 7.0);
        DisplayList(list);
        DestroyList(&list);
        return 0;
}
```

Variations of Linked Lists

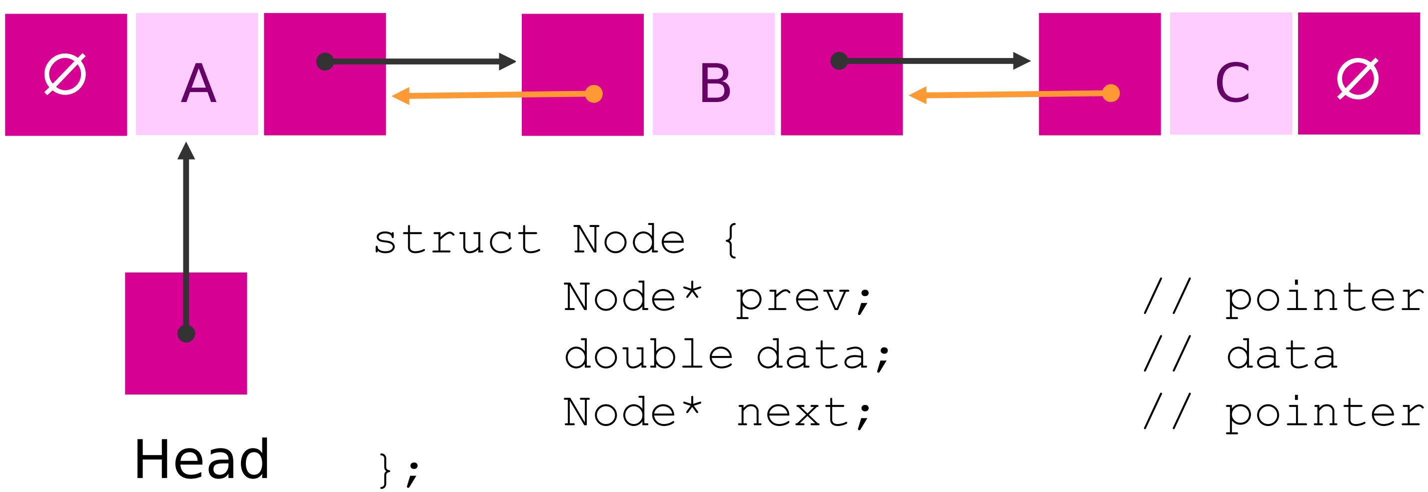
- *Circular linked lists*
  - The last node points to the first node of the list



  - How do we know when we have finished traversing the list?
    - check if the pointer of the current node is equal to the head

Variations of Linked Lists

- *Doubly linked lists*
  - Each node points to not only successor but the predecessor
  - There are two `NULL`: at the first and last nodes in the list
  - Advantage: given a node, it is easy to visit its predecessor. Convenient to traverse lists backwards




```
struct Node {
        Node* prev;             // pointer to prev
        double data;            // data
        Node* next;             // pointer to next
};
```

Array versus Linked Lists

- Linked lists are more complex to code and manage than arrays, but they have some distinct advantages.
  - **Dynamic**: a linked list can easily grow and shrink in size.
    - We don’t need to know how many nodes will be in the list. They are created in memory as needed.
  - **Easy and fast insertions and deletions**
    - To insert or delete an element in an array, we need to copy to temporary variables to make room for new elements or close the gap caused by deleted elements.
    - With a linked list, no need to move other nodes. Only need to reset some pointers.
  - **But...**
    - Slow to index an element