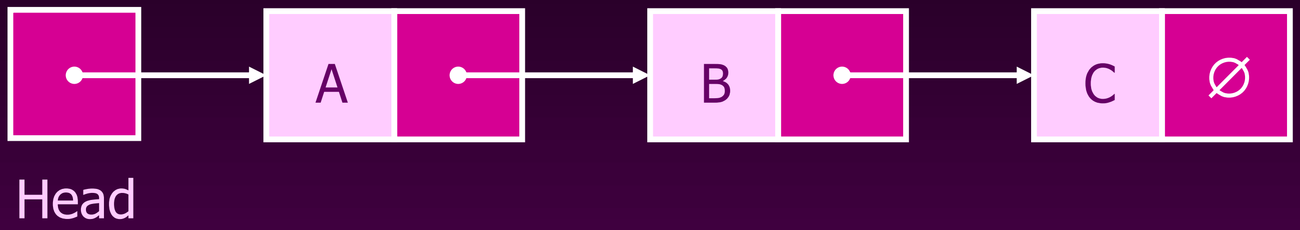
Linked Lists

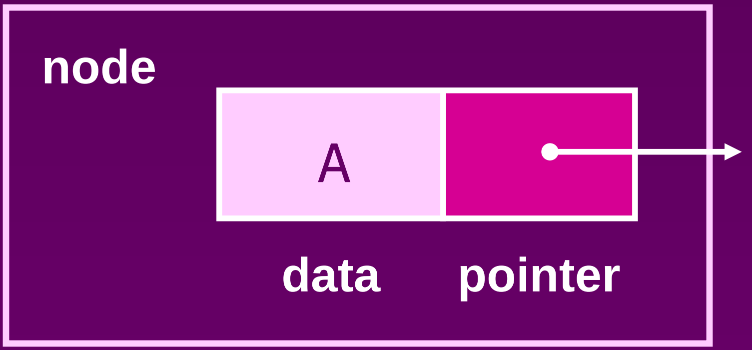
List Overview

- Linked lists
  - Abstract data type (ADT)
- Basic operations of linked lists
  - Insert, find, delete, print, etc.
- Variations of linked lists
  - Circular linked lists
  - Doubly linked lists

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
- *Head*: pointer to the first node
- The last node points to `NULL`



A Simple Linked List Class

- We use two classes: **`Node`** and **`List`**
- Declare `Node` class for the nodes
  - `data`: `double`-type data in this example
  - `next`: a pointer to the next node in the list

```
class Node {
public:
      double        data;        // data
      Node*         next;        // pointer to next
};
```

A Simple Linked List Class

- Declare `List`, which contains
  - `head`: a pointer to the first node in the list.
    Since the list is empty initially, `head` is set to `NULL`
  - Operations on `List`

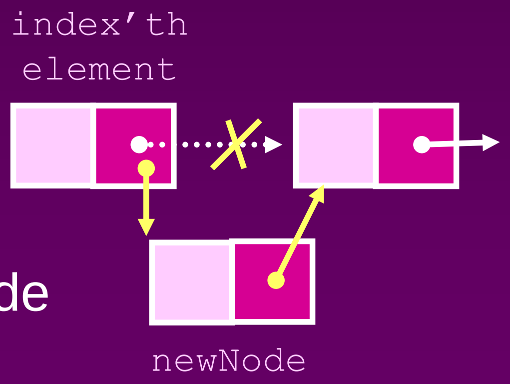
```
class List {
public:
      List(void) { head = NULL; }       // constructor
      ~List(void);                      // destructor

      bool IsEmpty() { return head == NULL; }
      Node* InsertNode(int index, double x);
      int FindNode(double x);
      int DeleteNode(double x);
      void DisplayList(void);
private:
      Node* head;
};
```

A Simple Linked List Class

- Operations of `List`
  - `IsEmpty`: determine whether or not the list is empty
  - `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
  - `DisplayList`: print all the nodes in the list

Inserting a new node

- `Node* InsertNode(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

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

      int currIndex  =      1;
      Node* currNode =      head;
      while (currNode && index > currIndex) {
            currNode        =      currNode->next;
            currIndex++;
      }
      if (index > 0 && currNode == NULL) return NULL;

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

Try to locate `index`'th node. If it doesn't exist, return `NULL`.

Inserting a new node

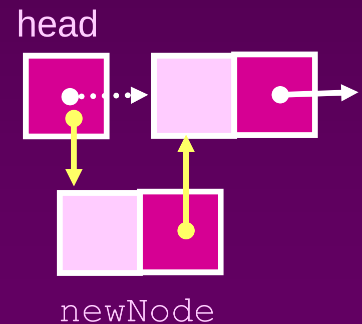
```
Node* List::InsertNode(int index, double x) {
        if (index < 0) return NULL;

        int currIndex  =      1;
        Node* currNode =      head;
        while (currNode && index > currIndex) {
                currNode        =       currNode->next;
                currIndex++;
        }
        if (index > 0 && currNode == NULL) return NULL;

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

Create a new node

Inserting a new node

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

      int currIndex  =      1;
      Node* currNode =      head;
      while (currNode && index > currIndex) {
            currNode        =      currNode->next;
            currIndex++;
      }
      if (index > 0 && currNode == NULL) return NULL;

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

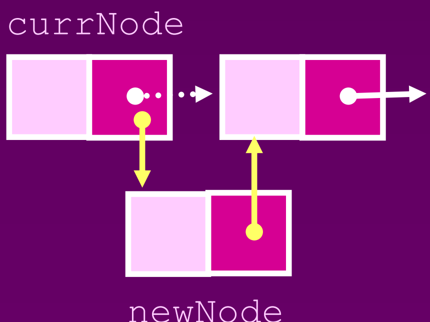
Insert as first element



Inserting a new node

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

      int currIndex  =       1;
      Node* currNode =       head;
      while (currNode && index > currIndex) {
            currNode        =       currNode->next;
            currIndex++;
      }
      if (index > 0 && currNode == NULL) return NULL;

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

Insert after `currNode`



Finding a node

- `int FindNode(double x)`
  - Search for a node with the value equal to `x` in the list.
  - If such a node is found, return its position. Otherwise, return 0.

```
int List::FindNode(double x) {
      Node* currNode     =     head;
      int currIndex      =     1;
      while (currNode && currNode->data != x) {
            currNode     =     currNode->next;
            currIndex++;
      }
      if (currNode) return currIndex;
      return 0;
}
```

Deleting a node

- `int DeleteNode(double x)`
  - Delete a node with the value equal to `x` from the list.
  - If such a node is found, return its position. Otherwise, return 0.
- 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
- Like `InsertNode`, there are two special cases
  - Delete first node
  - Delete the node in middle or at the end of the list

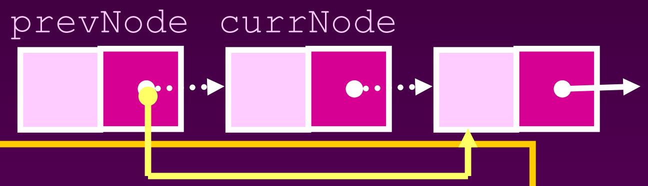
Deleting a node

```
int List::DeleteNode(double x) {
      Node* prevNode =      NULL;
      Node* currNode =      head;
      int currIndex  =      1;
      while (currNode && currNode->data != x) {
            prevNode      =       currNode;
            currNode      =       currNode->next;
            currIndex++;
      }
      if (currNode) {
            if (prevNode) {
                  prevNode->next =      currNode->next;
                  delete currNode;
            }
            else {
                  head           =      currNode->next;
                  delete currNode;
            }
            return currIndex;
      }
      return 0;
}
```

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

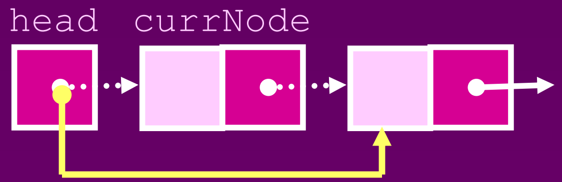
Deleting a node

```
int List::DeleteNode(double x) {
        Node* prevNode =       NULL;
        Node* currNode =       head;
        int currIndex  =       1;
        while (currNode && currNode->data != x) {
                prevNode        =       currNode;
                currNode        =       currNode->next;
                currIndex++;
        }
        if (currNode) {
                if (prevNode) {
                        prevNode->next =       currNode->next;
                        delete currNode;
                }
                else {
                        head            =       currNode->next;
                        delete currNode;
                }
                return currIndex;
        }
        return 0;
}
```



Deleting a node

```
int List::DeleteNode(double x) {
        Node* prevNode =        NULL;
        Node* currNode =        head;
        int currIndex  =        1;
        while (currNode && currNode->data != x) {
                prevNode        =       currNode;
                currNode        =       currNode->next;
                currIndex++;
        }
        if (currNode) {
                if (prevNode) {
                        prevNode->next =        currNode->next;
                        delete currNode;
                }
                else {
                        head            =       currNode->next;
                        delete currNode;
                }
                return currIndex;
        }
        return 0;
}
```



Printing all the elements

- `void DisplayList(void)`
  - Print the data of all the elements
  - Print the number of the nodes in the list

```
void List::DisplayList()
{
    int num          =      0;
    Node* currNode   =      head;
    while (currNode != NULL){
        cout << currNode->data << endl;
        currNode     =      currNode->next;
        num++;
    }
    cout << "Number of nodes in the list: " << num << endl;
}
```

Destroying the list

- ~List(void)
  - Use the destructor to release all the memory used by the list.
  - Step through the list and delete each node one by one.

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

Using `List`

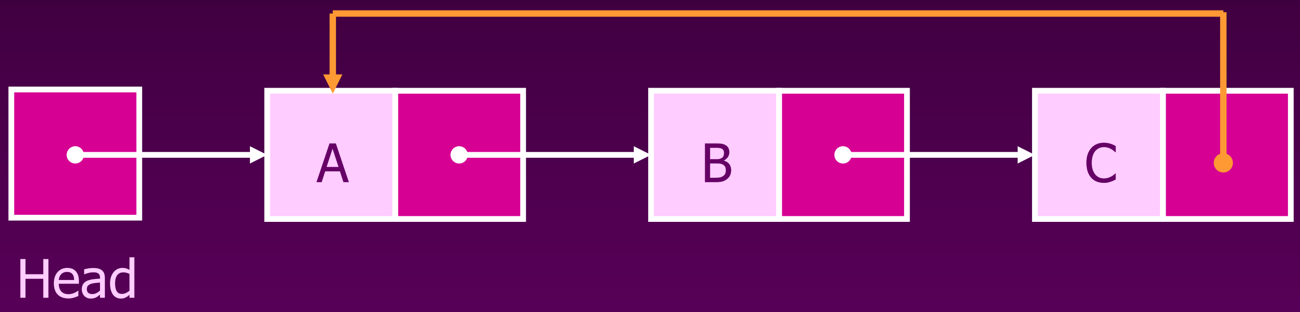
```
int main(void)
{
        List list;
        list.InsertNode(0, 7.0);    // successful
        list.InsertNode(1, 5.0);    // successful
        list.InsertNode(-1, 5.0);   // unsuccessful
        list.InsertNode(0, 6.0);    // successful
        list.InsertNode(8, 4.0);    // unsuccessful
        // print all the elements
        list.DisplayList();
        if(list.FindNode(5.0) > 0) cout << "5.0 found" << endl;
        else                       cout << "5.0 not found" << endl;
        if(list.FindNode(4.5) > 0) cout << "4.5 found" << endl;
        else                       cout << "4.5 not found" << endl;
        list.DeleteNode(7.0);
        list.DisplayList();
        return 0;
}
```

result

```
6
7
5
Number of nodes in the list: 3
5.0 found
4.5 not found
6
5
Number of nodes in the list: 2
```

Variations of Linked Lists

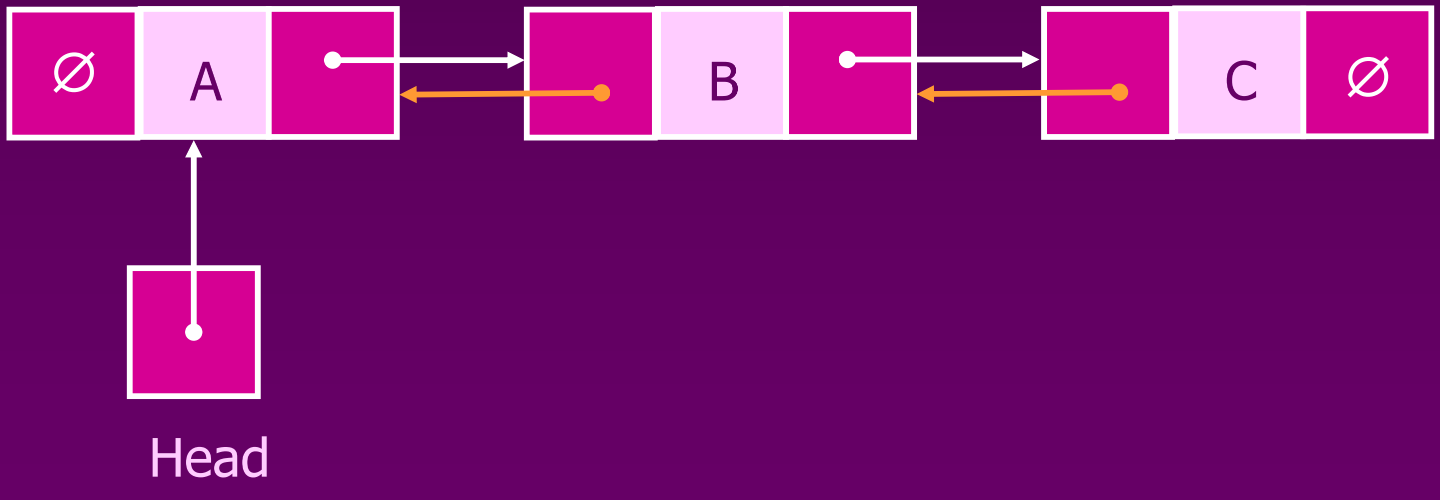
- *Circular linked lists*
  - The last node points to the first node of the list



  - How do we know when we have finished traversing the list? (Tip: 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



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.
    - In contrast, the size of a C++ array is fixed at compilation time.
  - **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.