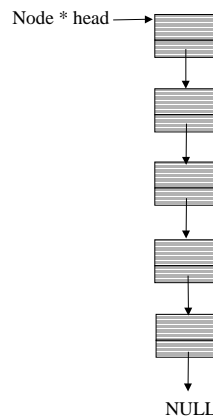


## Linked Lists

- A container class
- Completely dynamic storage
- A linked list is made up of nodes
  - ✦ Each node contains a datum
    - e.g. an int or
    - a project
  - ✦ Each node also contains a pointer
- Each node's pointer points to the *next* node.
  - ✦ The node's form a chain of nodes, hence a linked list.
  - ✦ The final pointer points to NULL.
- The list requires that we have a pointer variable that points to the first node, or the "head" of the list.

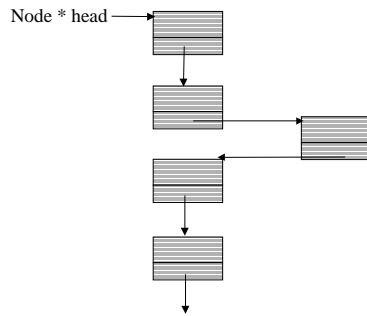
## Linked List Structure

Node: 



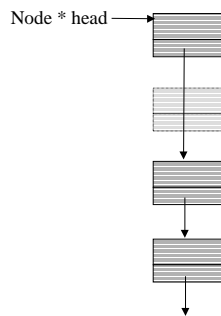
## Adding a Node

- ✦ Find the node ( $N_x$  say) before the one you want to add.
- ✦ Assign the value of  $N_x$ 's *next* to a temp pointer.
- ✦ Create a new node ( $N_{x+1}$  say) using  $N_x$ 's *next* and the item that you wish to add.
- ✦ Assign the value of temp to  $N_{x+1}$ 's *next*.



## Deleting a Node

- ✦ Find the node ( $N_x$  say) you want to delete.
- ✦ Assign the value of  $N_x$ 's *next* to a temp pointer.
- ✦ Delete the memory that  $N_{x-1}$ 's *next* points at.
- ✦ Assign the value of temp to  $N_x$ 's *next*.
- ✦ How do you keep track of  $N_{x-1}$ ?



## Adding and Deleting Nodes

- Easier to add or delete a node from the head or tail.
  - ✦ It is not necessary to find the node.
  - ✦ Must still remember to store the appropriate pointer before rearranging.
- If the list is sorted the node must be inserted in the correct position so it is necessary to find the node.
- To avoid any global sorting we can ensure that a linked list remains sorted by always adding nodes in the sort order.
- Searching a linked list (finding a node with a particular value) is linear. What about an array?

## Interface

```
class ItemList
{
public:
    ItemList(); //default constructor
    ItemList(const ItemList & iL); //copy
    ~ItemList(); //destructor
    ItemList & operator=(const ItemList & iL);
    void addItem(const & Item it);
    void insertItem(const & Item it);
    bool deleteItem(const & Item it);
    int getNumItems();
private:
    struct Node
    {
        Item nodeData;
        Node *next;
    }
    int numItems;
    Node *head;
    void emptyList();
}
```

## Implementation 1

```
#include "ItemList.h"

ItemList::ItemList() //default
{
    numItems = 0;
    head = NULL;
}

ItemList::ItemList(const ItemList & iL)
{
    Node *temp;
    temp = iL.head;
    while (temp != NULL)
    {
        addItem(temp->nodeData);
        temp = temp->next;
    }
}

~ItemList::ItemList()
{
    emptyList();
}
```

## Implementation 2

```
void ItemList::addItem(const Item & it)
{
    Node * temp = head;
    if (head == NULL)
    {
        head = new node;
        head ->nodeData = it;
        head ->next = NULL;
        ++numItems;
    }
    else
    {
        while (temp->next != NULL)
            temp = temp-> next;
        temp->next = new Node;
        temp->next->nodeData = it;
        temp->next->next = NULL;
        ++numItems;
    }
}

int ItemList::getNumItems();
{
    return numItems;
}
```

## Implementation 3

```
ItemList & ItemList::operator=
    (const ItemList & iL)
{
    if (this != & iL)
    {
        Node *temp;
        emptyList();
        temp = iL.head;
        while (temp != NULL)
        {
            addItem(temp->nodeData);
            temp = temp->next;
        }
    }
    return *this;
}

void ItemList::emptyList()
{
    Node *temp = head;
    while (head->next != NULL)
    {
        temp = head->next;
        head->next = temp->next;
        delete temp;
    }
    delete head;
    numItems = 0;
}
```

## Implementation 4

```
void ItemList::insertItem(const & Item it);
{
    Node *temp;
    Node *temp2;
    if (head == NULL || head->nodeData > it)
    {
        if (head == NULL)
            temp = NULL;
        else
            temp = head->next;
        head = new node;
        head->nodeData = it;
        head->next = temp;
    }
    else
    {
        temp = head;
        do
        {
            temp2 = temp;
            temp = temp->next;
        } while(temp->nodeData < it && temp != NULL);
        temp2->next = new node;
        temp2 = temp2->next;
        temp2->nodeData = it;
        temp2->next = temp;
    }
    ++numItems;
}
```

## Implementation 5

```
bool ItemList::deleteItem(const &Item it);
{
    Node *temp = head;
    Node *temp2;
    if (head->nodeData == it)
    {
        temp = head;
        head = head->next;
        delete temp;
        --numItems;
        return true;
    }
    while (temp != NULL)
    {
        temp2 = temp;
        temp = temp->next;
        if (temp->nodeData == it)
        {
            temp2->next = temp->next;
            delete temp;
            --numItems;
            return true;
        }
    }
    return false;
}
```