Algorithms and Data Structures (CSci 115)

California State University Fresno
College of Science and Mathematics
Department of Computer Science
H. Cecotti

Learning outcomes

- From Arrays to Lists
  - Arrays vs. Lists
- To understand:
  - Addition/Deletion of Elements in Arrays (impact on memory allocation)
    - Unordered array
    - Ordered array
  - Linked Lists
    - Creating a node
    - Adding/Removing a node

Arrays

- Arrays are really useful for storing data
  - You know how many elements you have (fixed size)
- Especially true when we know **in advance** how many pieces of data (elements) we need
- Crucially, we can go straight to an element as long as we know its location
  - We say we "access an element"
  - **Direct access**
    - based on the location in memory
    - you can compute its position based on the size of an element in the array
- Arrays can be problematic when we have to deal with Insertions and/or Deletions

Array of Unordered Elements

- Let's assume we
  - maintain an array of **unordered elements**
  - keep a record of the number of elements (**`noOfElements`**)
  - keep some "unoccupied elements" as free space towards the end of the array to allow us to increase the number of values held

INITIAL ARRAY

myArray	0	1	2	3	4	5	6	7
	56	39	45	5	28	63		

How do we add/remove elements?

- If we are asked to **insert** a new value we must:
  - Insert the value in the next available location
  - Increment the number of elements (`noofElements++`)
- To delete an element we must:
  - Locate the element
  - Close up the space occupied by that element
  - Decrement the number of elements (`noofElements--`)

Inserting an Element

- INITIAL ARRAY (after a few insertions)

myArray	0	1	2	3	4	5	6	7
	56	39	45	5	28	63		

`noOfElements` 6

To insert the value 50 - if there is space

```
myArray[noOfElements] = 50;
noOfElements++;
```

myArray	0	1	2	3	4	5	6	7
	56	39	45	5	28	63	50	

`noOfElements` 7

Deleting an Element

- **ORIGINAL ARRAY**

myArray	0	1	2	3	4	5	6	7
	56	39	45	5	28	63	50	

To delete the value 28

STEP 1: Find location of value to be deleted

myArray	0	1	2	3	4	5	6	7
	56	39	45	5	~~28~~	63	50	

STEP 2: If found - close up the gap

myArray	0	1	2	3	4	5	6	7
	56	39	45	5	28	63	50	

myArray	0	1	2	3	4	5	6	7
	56	39	45	5	63	63	50	

myArray	0	1	2	3	4	5	6	7
	56	39	45	5	63	50	50	

STEP 3: Decrement the number of elements

`noOfElements` | 6 |

Problems?

- Note we still have an extra value (50) at `myArray[6]`
- **In general:**
  - We are restricted to having no more values than the size of the array
  - Insertions are trivial
  - Deletions cause problems

myArray	0	1	2	3	4	5	6	7
	56	39	45	5	63	50	50	

Array of Ordered Elements

INITIAL ARRAY

myArray	0	1	2	3	4	5	6	7
	27	36	45	54	58	63		

- Assume we
  - maintain an array of **ordered elements**
    - keep a record of the number of elements (`noOfElements`)
  - keep some “unoccupied elements” as free space towards the end of the array to allow us to increase the number of values held

How do we insert into this array?

- If we are asked to insert a new value, we must:
  - Find out exactly where we want to place the value (so as to maintain the ordering)
  - Free up the space at the appropriate position
  - Actually insert the value
- To free up space
  - we normally have to move some of the elements **along the array**
- Only when we have freed up the space can we insert the value

INITIAL ARRAY (after a few insertions)

myArray	0	1	2	3	4	5	6	7
	27	36	45	54	58	63		

noOfElements 6

To insert the value 50 - if there is space

STEP 1:
Identify WHERE the value is to placed (i.e. location 3)

myArray	0	1	2	3	4	5	6	7
	27	36	45	54	58	63		

STEP 2: Creating Space

myArray	0	1	2	3	4	5	6	7
	27	36	45	54	58	63		

myArray	0	1	2	3	4	5	6	7
	27	36	45	54	58	63	63	

myArray	0	1	2	3	4	5	6	7
	27	36	45	54	58	58	63	

myArray	0	1	2	3	4	5	6	7
	27	36	45	54	54	58	63	

Inserting a Value

- STEP 3: Insert the value 50

myArray	0	1	2	3	4	5	6	7
	27	36	45	50	54	58	63	

Note how we can easily do this by DIRECTLY ACCESSING the array element:

```
myArray [3]  =  50;
```

STEP 4: Increment the number of elements

```
noOfElements++;
```

noOfElements	7

Deleting Values

- **Deleting** values can be just as tricky
- We have to remove the value
- We then have to **close up** the space previously occupied
- Decrement the number of elements

ORIGINAL ARRAY

myArray	0	1	2	3	4	5	6	7
	27	36	45	50	54	58	63	

To delete the value 50

STEP 1: Find location of value to be deleted

myArray	0	1	2	3	4	5	6	7
	27	36	45	~~50~~	54	58	63	

STEP 2: If found - close up the gap

myArray	0	1	2	3	4	5	6	7
	27	36	45	50	54	58	63	

myArray	0	1	2	3	4	5	6	7
	27	36	45	54	54	58	63	

myArray	0	1	2	3	4	5	6	7
	27	36	45	54	58	58	63	

Note extra value

STEP 3: Decrement the number of elements

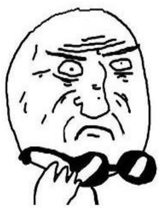
`noOfElements` | 6

Arrays

- INSERTIONS
  - Insertions at the **END** of the array are trivial
  - Insertions at the **START** or somewhere in the **MIDDLE** cause problems
    - Shift of the elements
- DELETIONS
  - Deletions from the **END** of the array are trivial
  - Deletions at the **START** or somewhere in the **MIDDLE** cause problems
    - Shift of the elements
- Length of arrays is fixed

Dynamic arrays

- Properties: Size + Capacity
- Double the capacity when it is full (size==capacity)
- We consider when capacity=n
  - ➢ We add n elements → O(1)
  - ➢ We add element n+1 → O(n) ☹
  - ➢ **But on average**
    - o (n*O(1)+O(n))/(n+1) = O(1) constant ☺
    - o n+1 operations

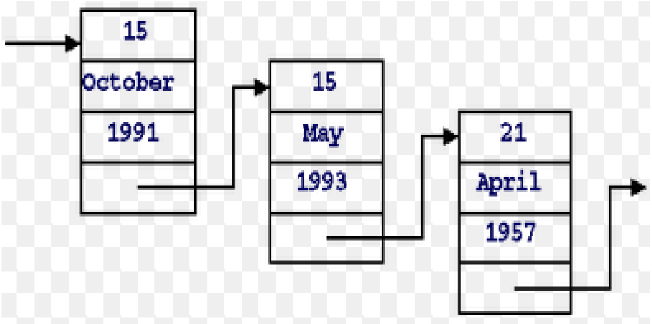


  - ➢ When n is very large, you will do a lot of insert in O(1) before you do the action that costs O(n)

Linked Lists

- An array is a **static** data structure
  - length of the array cannot be altered at run time
  - all the elements are kept at consecutive memory locations

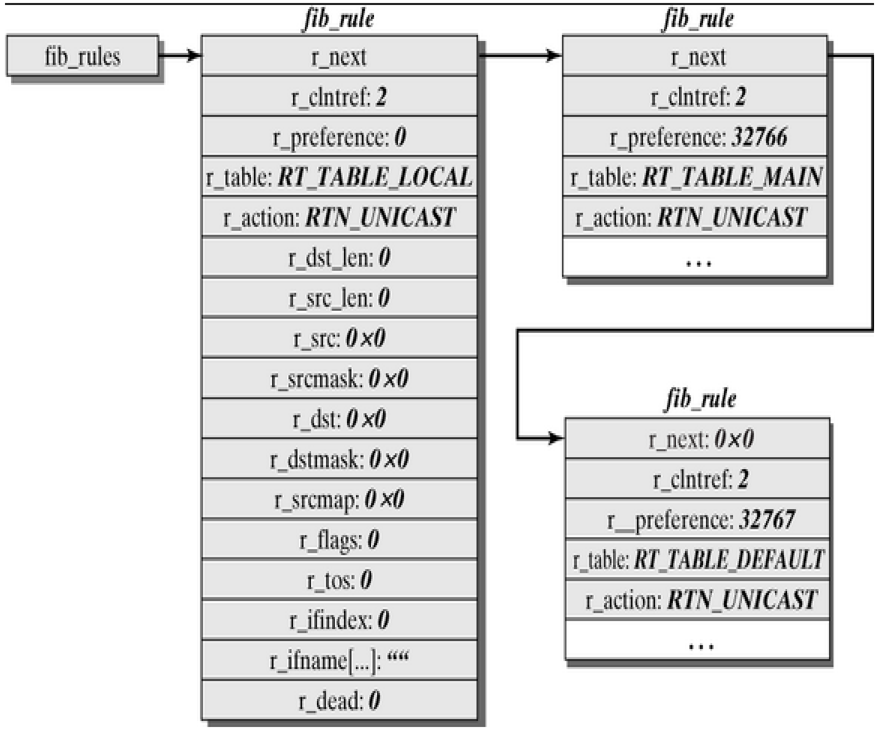
- A linked list is a **dynamic** data structure
  - length can be increased and decreased at run time
  - the elements may be kept at any location but still be connected to each other
  - create objects
  - use references to link objects



Lists - Real-world Examples

- Examples:

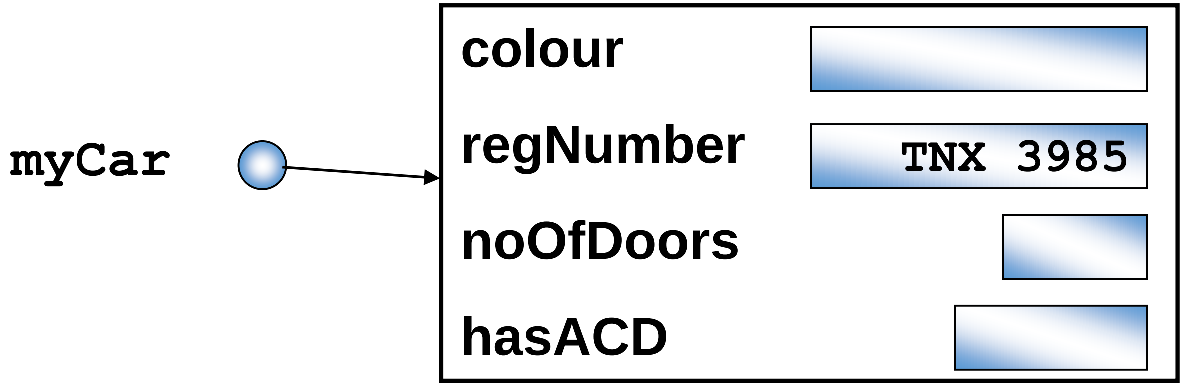
Routing Rules in Linked Lists



Using References to Link Objects

- Objects are created dynamically using the **`new`** operator
- A variable used to keep track of an object is actually a **reference** to the object

```
Car myCar = new Car("TNX 3985");
```



➢ Accomplishes two things:
  - declares **`myCar`** to be a reference to a **`Car`**
  - instantiates an object of class **`Car`**

Linked Lists

- Consider an object that contains a reference to another object of the same type:

```
class Node {
   private String data;
   private Node next;
}//Node
```

- This kind of class definition is called **self-referential** because it contains a field – called **next** – of the same type as itself

Linked Lists

- 2 objects of this class can be instantiated and chained together by having the **`next`** reference of one **`Node`** object refer to the other
- The second object's **`next`** reference can refer to a third **`Node`** object, and so on, creating a **linked list**
- The last node in the list would have a **`next`** reference that is **`null`**, indicating the end of the list
- **Head**
  - first node
- **Tail**
  - refer either to the rest of the list after the head,
  - or to the last node in the list.

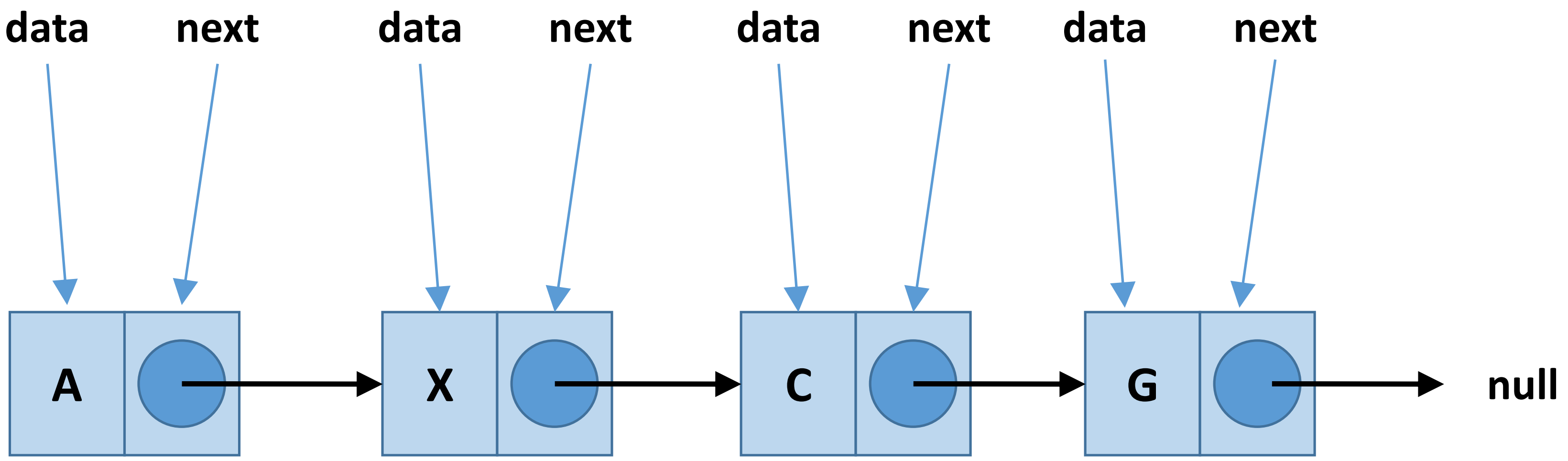
Linked Lists

- Interface List.h
  - Example:

```
#pragma once

#include "DataStructure.h"

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

class MyList : public DataStructure
{
public:
    MyList();
    ~MyList();

    DataStructure* clone() { return new MyList(); }

    void Createnode(int value);

    void Insert(int value);

    void Insert_first(int value);
    void Insert_last(int value);
    void Insert_position(int pos,int value);
    void Delete_first();
    void Delete_last();
    void Delete_position(int pos);
    void Display();
    void DisplayFile();

private:
    node *head, *tail;
};
```

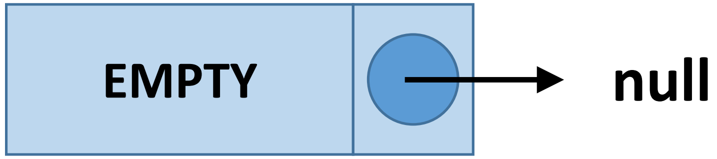
Diagram of a Linked List

▪



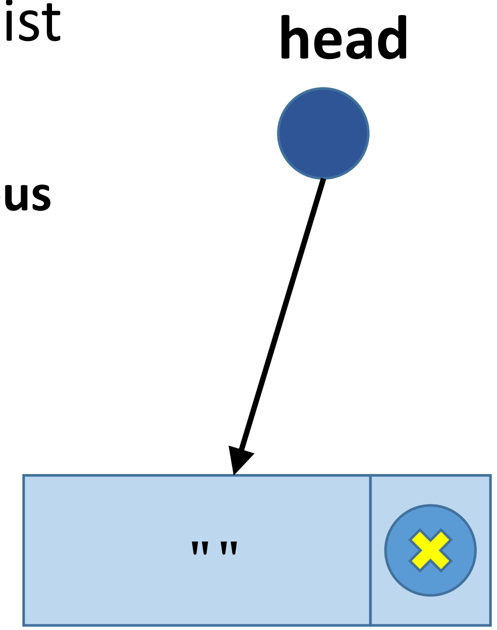
Node Class

```
class Node {
   private String data;
   private Node next;
   protected Node() {
      data = "EMPTY";
      next = null;
   }//Default Constructor
   protected Node(String newData, Node newNext) {
      data = newData;
      next = newNext;
   }//Alternative Constructor
   //get() and set(…) methods
}//Node
```



Header Node

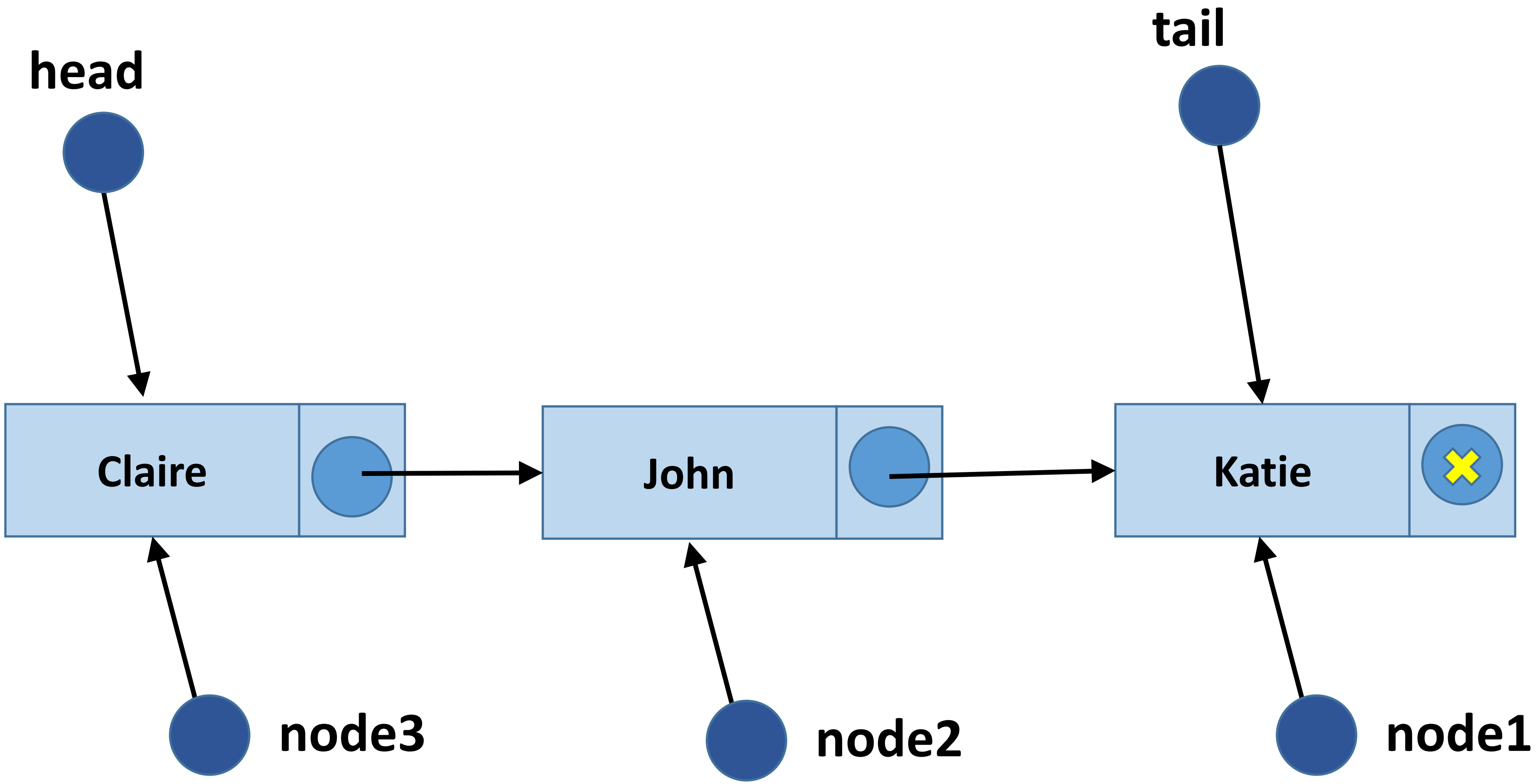
- A problem with linked lists
  - operations at the start and the end of the lists can be difficult
- Possible to avoid the special cases that occur at the start of the list by adding an extra node (the **header node**) at the start of each list
- This **header node** does not contain meaningful data
  - but ensures that all the nodes containing meaningful data have a **previous** node
- A similar trick can be used at the end of a list



SingleLinkedList Class

- Can be defined to include
  - Node **`head`** – the first element in the list
  - A count of the number of Nodes in the list (**`noOfElements`**)
    - For loop
  - A series of permissible methods
  - Optionally, a node **`tail`** – to access the last element in the list

SingleLinkedList Class - Visually

- 

SingleLinkedList Class

- Methods required:
  - Initialise a SingleLinkedList object
  - Add an element to the start of the list
  - Remove an element from the start of the list
  - Return the number of elements in the list
  - Print out the list
- Other possible methods:
  - Add an element to the end of a list
  - Remove an element from the end of the list
  - Add an element to the middle of a list
  - Remove an element from the middle of the list

Printing the List Contents

- Printing the values in the list

```
IF list is NOT empty
    Output number of elements
    Create a temp Node to point to the head          Node temp;
    WHILE (temp != null)
        Output value at temp node
        Move temp to next node in the list
ELSE
    Output “List is Empty”
```

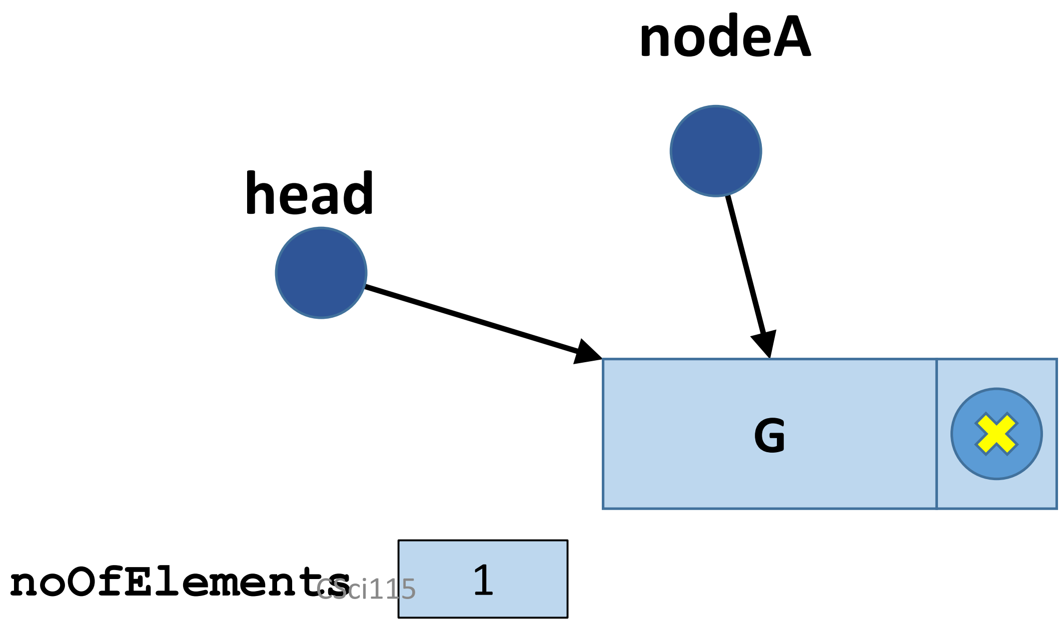
SingleLinkedList Class

```
public class SingleLinkedList {

   private Node head;
   private int noOfElements;

   protected SingleLinkedList () {
      head = null;
      noOfElements = 0;
   }//Constructor
   …
}//SingleLinkedList

SingleLinkedList myList = new SingleLinkedList();
```

Building a Node

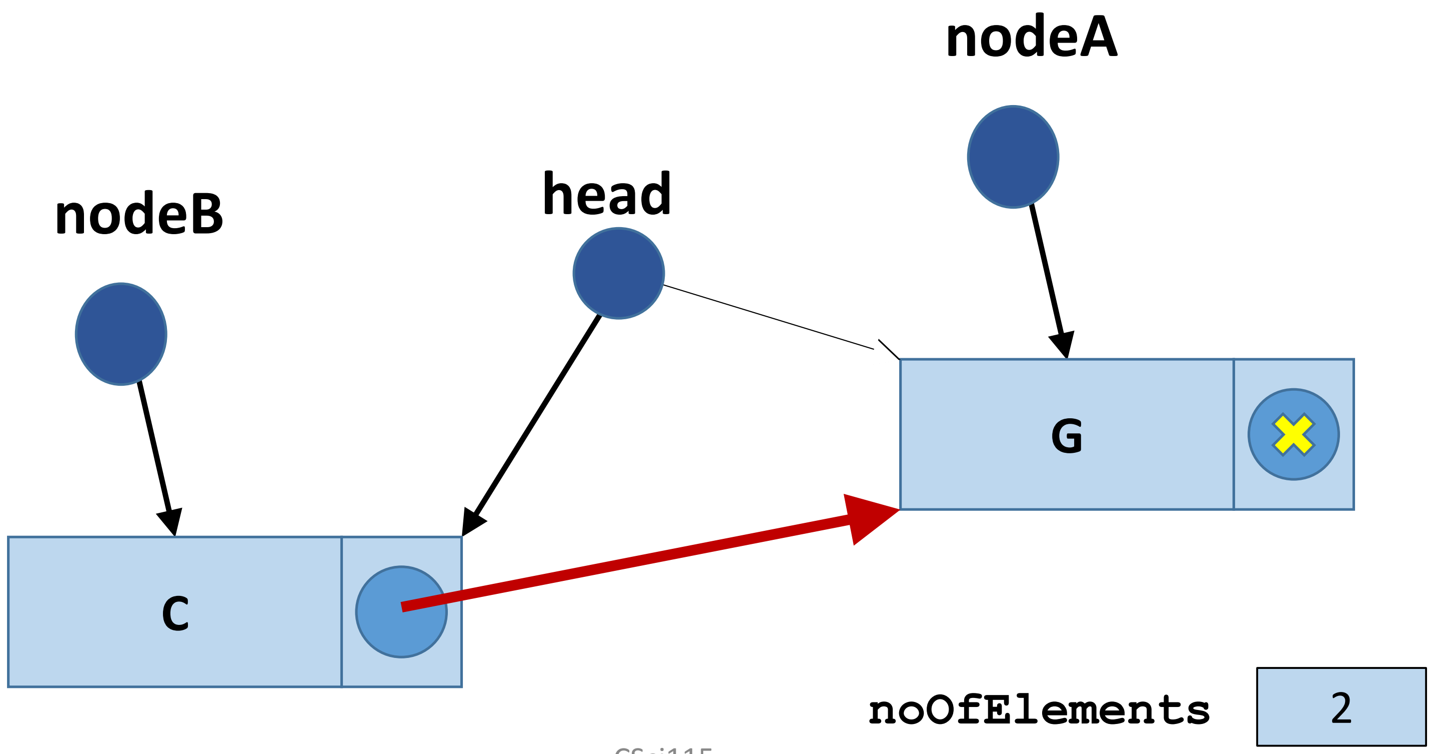
- All Nodes will be built as follows:

```
Node nodeA  =  new Node ("G", null);
myList.addStart(nodeA);
```



Adding Another Node

```
Node nodeB  =  new Node ("C", null);
myList.addStart(nodeB);
```



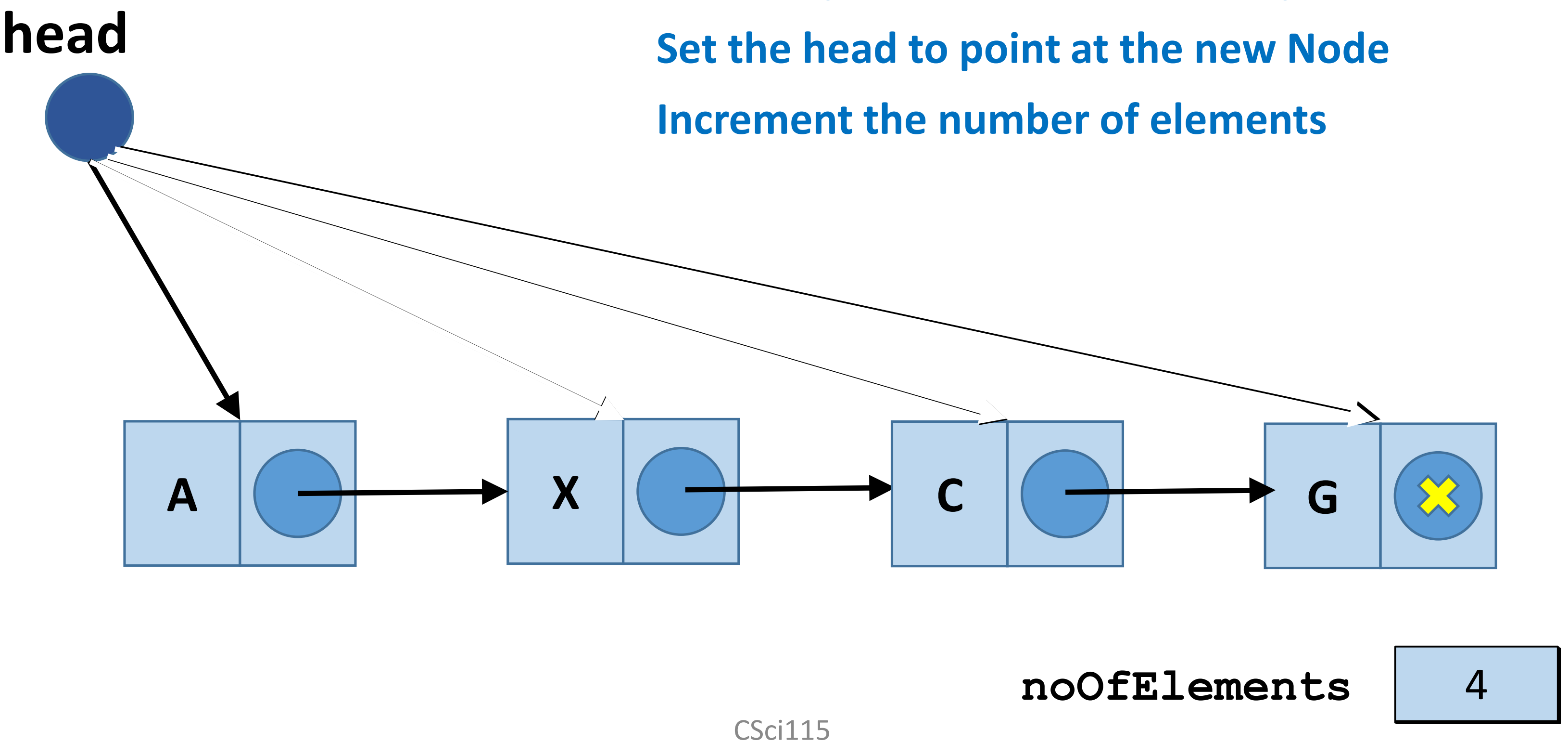
Adding More Nodes

- We can add more elements to the start of the list in a similar manner

Set next pointer of new Node to point to head

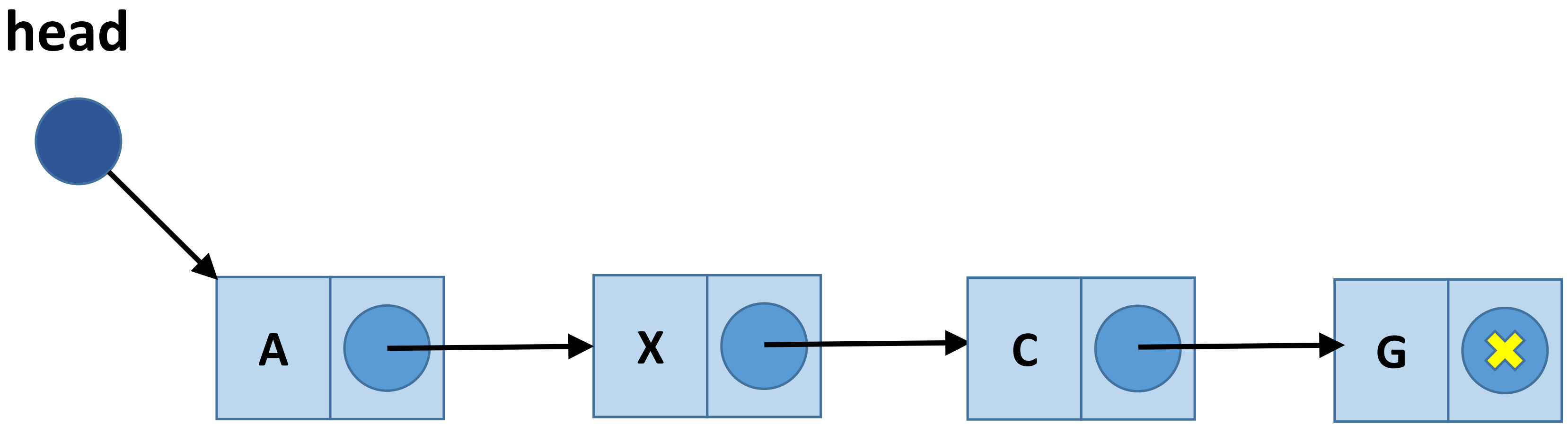
Set the head to point at the new Node

Increment the number of elements



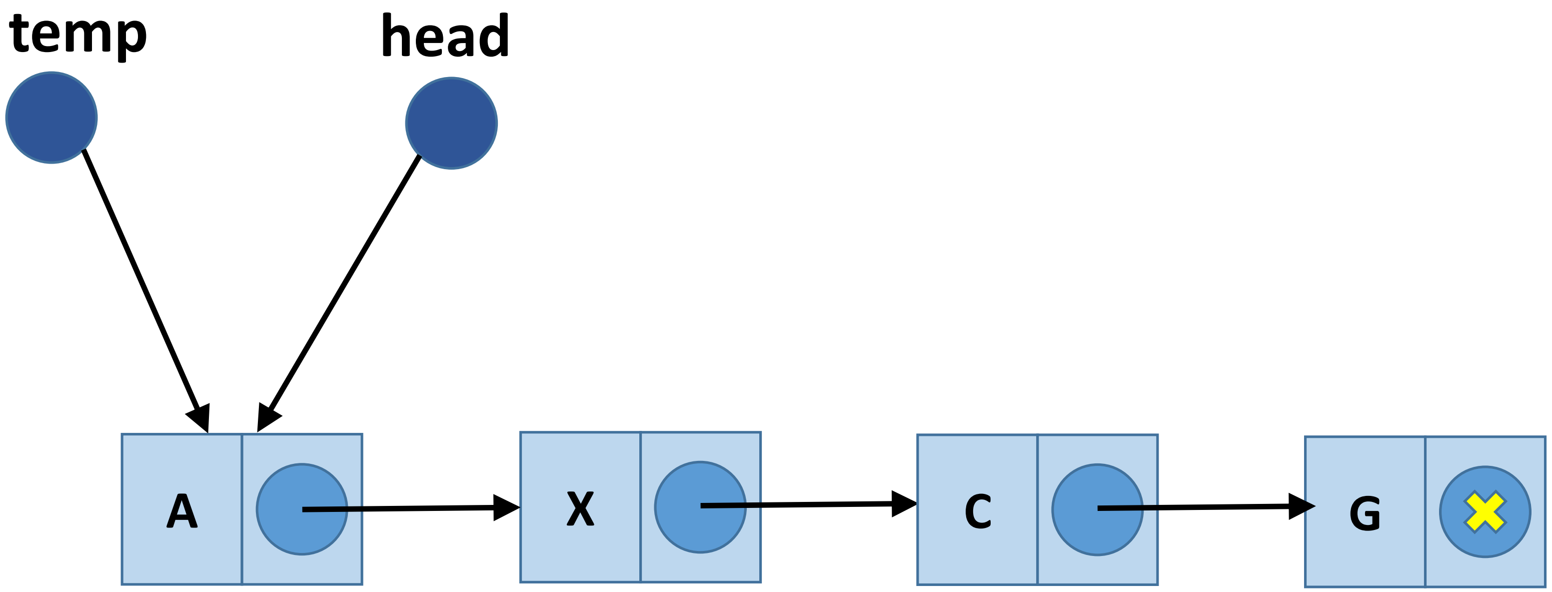
Removing a Node

- We can remove elements from the start of the list
- First of all we must check to see whether there is a **head** element
- If there is not – then we cannot delete it!
- If there is we can proceed as follows:



Removing a node

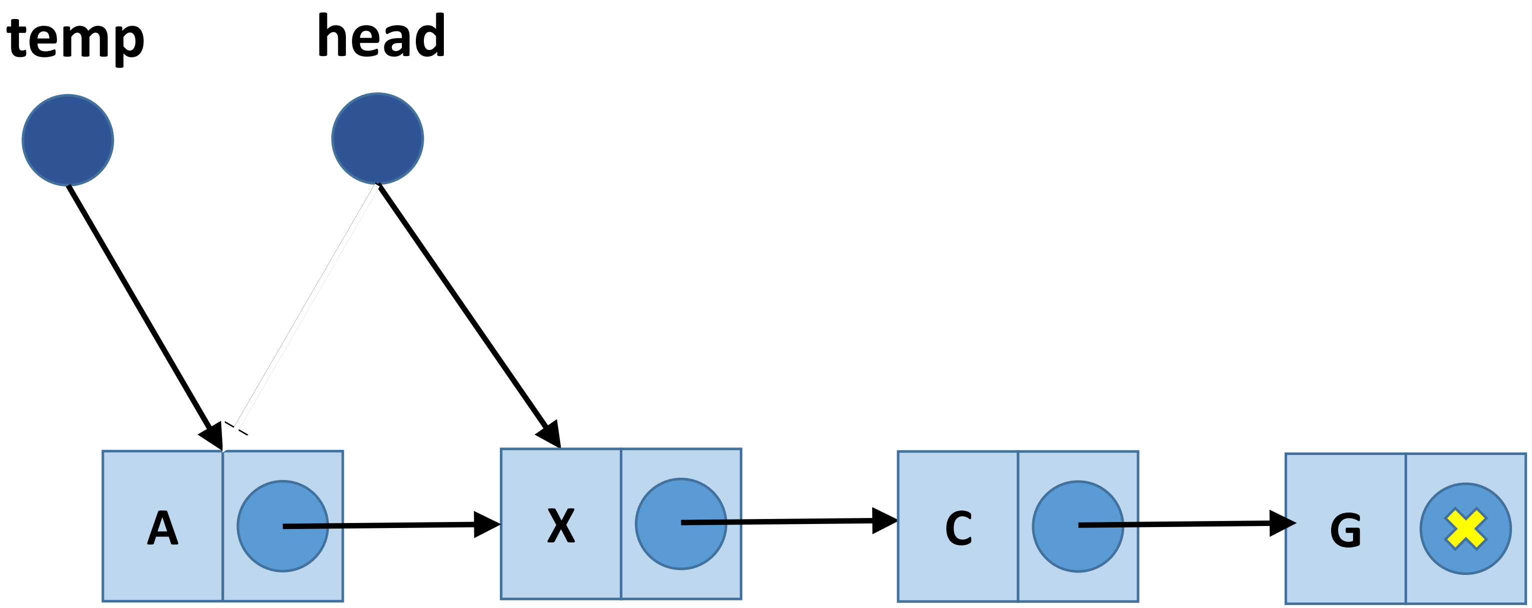
- **STEP 1:**
  - ➢Create a **temp** Node
  - ➢Make it point to the **head** element



Removing a node

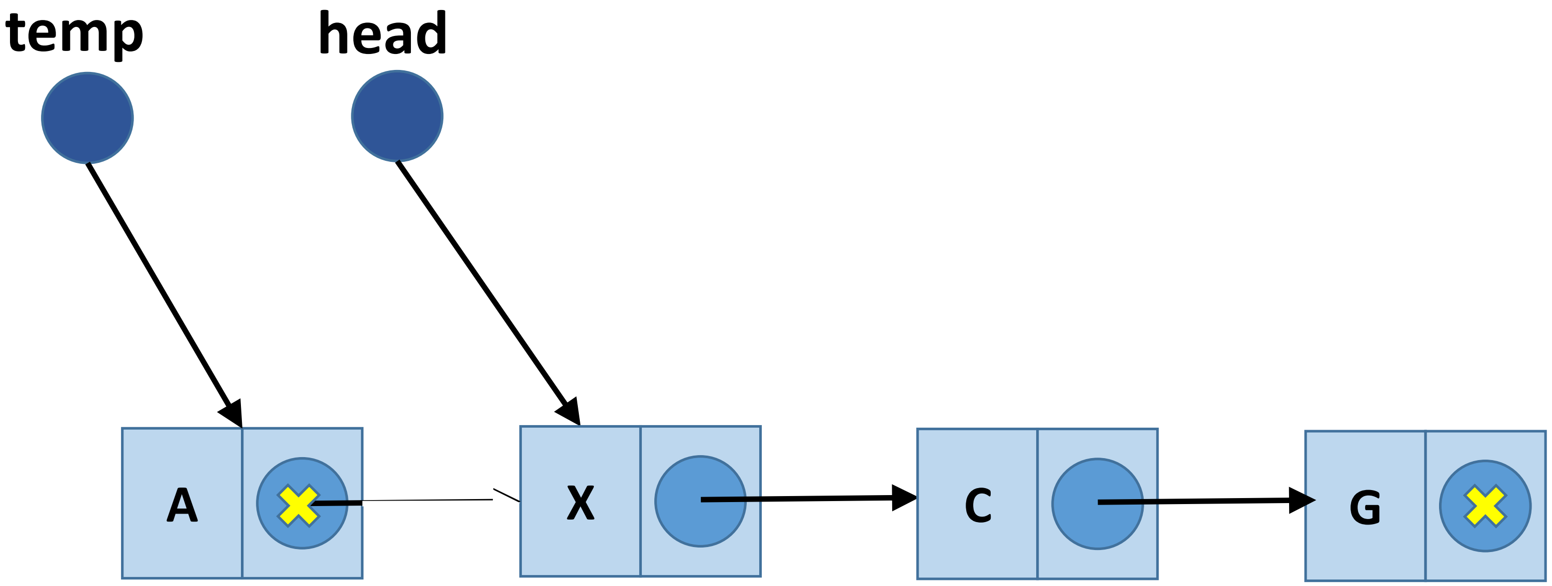
- **STEP 2:**
  - ➢Advance **head** to point to next node –

```
head.getNext()
```



Removing a node

- **STEP 3:**
  - Set the 'connecting' pointer to **null**



- **STEP 4:**
  - Decrement the number of elements

SingleLinkedList Class

- Add an element to the end of a list

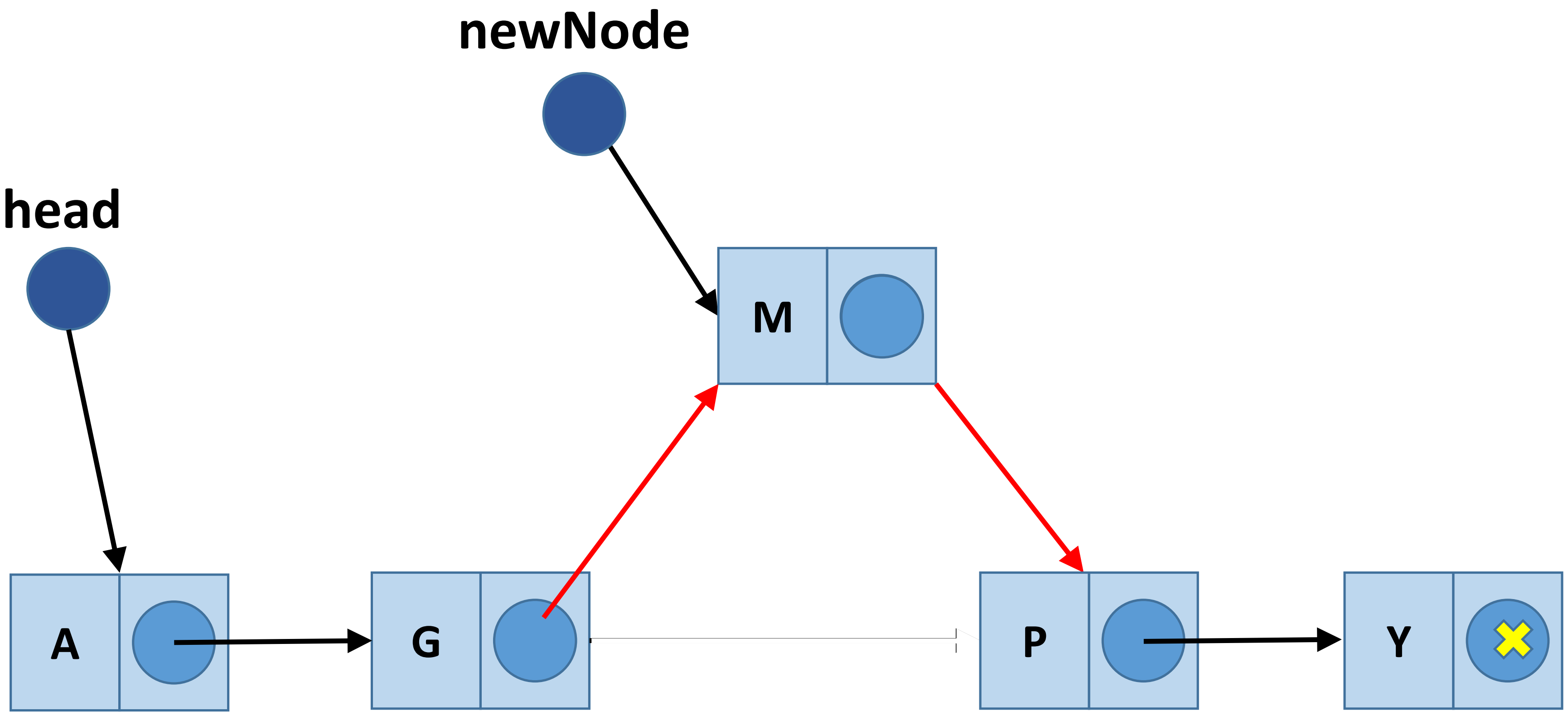
```
Set next pointer to null
IF list is empty
    Set head to point to the new node
ELSE
    Create a temp node to point to the head
    WHILE (temp.next != null)
        Set temp to point to temp.next
    Set temp.next to point to the new node
Increment the number of elements
```

SingleLinkedList Class

- Remove an element from the end of the list

```
IF list is empty
    Output message
ELSE
    If there is only one element
        Set the head to null
    ELSE
        Create a temp node to point to the head
        WHILE (temp.next.next != null)
            Set temp to point to temp.next
            Set temp.next to point to null
            Decrement the number of elements
```

Inserting into an Ordered List



Inserting into an Ordered List

- Check:
  - If list is empty
  - If newNode is to be inserted at the start
  - If newNode is to be inserted at the end
  - If newNode is to be inserted in the middle
- Increment number of elements

General Points

- Generally
  - developing software that works in MOST cases is fairly straightforward
- Developing software that works in **ALL** cases is much more difficult
  - Many additional checks required
- Software Developers
  - → **eye for detail**
- Efficiency
  - Insertion and deletion at the beginning of a linked list
    - very fast
    - involve changing only **one** or **two** references
  - Finding, deleting, or inserting **next** to a specific item
    - requires searching through, on average, **half the items in the list** !

Linked Lists over Arrays?

- Linked lists
  - preferred mostly when you don't know the volume of data to be stored
    - the number of elements can change
- Example
  - in an employee management system, one cannot use arrays as they are of fixed length while any number of new employees can join
  - In scenarios like these
    - linked lists are used as their capacity can be increased or decreased at run time, as and when required

Questions ?

- Midterm1/Final
  - ➢Be confident with insert, delete functions
    - o Iterative/recursive versions
    - o Pointers/References
- Reading:
  - ➢Csci 115 book: section 5.1
  - ➢Chapter 10: Elementary Data Structures
    - o Introduction to Algorithms 3rd Ed.