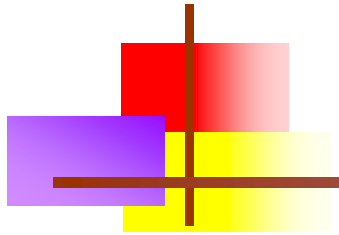
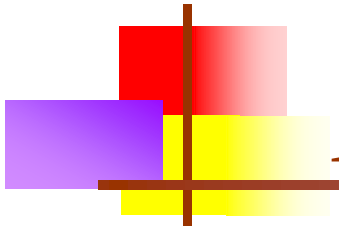


Linked Lists - II



Advantages of Linked lists

- Dynamic in nature. Memory allocated at run time.
- Insertion and Deletions are constant time operations. No need to shift nodes as was necessary with arrays.
- Other data structures like queues, stacks are easily implemented using linked lists

Polynomials

- Add
- $5 + 2x + 3x^2$
- $7x + 8$

- $13 + 9x + 3x^2$

Polynomials

- Add
- $5 + 2y + 3y^2$
- $7y + 8$

- $13 + 9y + 3y^2$

- Thus we simply need to store only the coefficients and the exponents

Storing polynomials using arrays

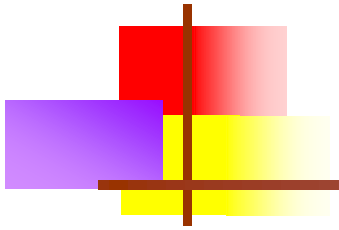
- $5 + 2x + 3x^2$
- [5 2 3]

- $7x + 8$
- [8 7 0]

- Store only the coefficients in proper place

Issues in Storing polynomials using arrays

- $5 + 2x + 3x^2 + 6x^5$
- [5 2 3 0 0 6]
- $5 + 2x + 3x^2 + 7\ x^{31}$
- [5 2 3 0 0 0 0 0 . . . . 7]
- Need to store so many zeroes in a very large sized array



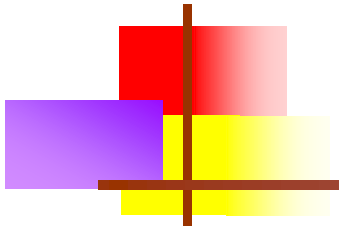
Storing Polynomials using linked lists

- Let us now see how two polynomials can be added.
- Let P1 and P2 be two polynomials.
  - stored as linked lists
  - Each node contains exponent and co-eff values
  - in sorted (decreasing) order of exponents
- The addition operation is defined as follows
  - Add terms of like-exponents.

Representing a polynomial using a linked list

- Store the coefficient and exponent of each term in nodes
  - int [] item1 = {5, 12}
  - int [] item2 = {2, 9}
  - int [] item3 = {-1, 3}

$$5x^{12} + 2x^{9} - x^{3}$$



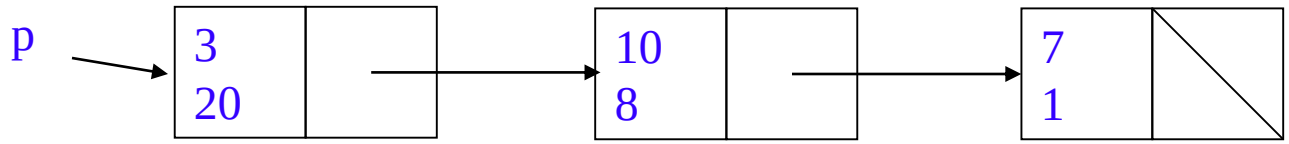
Operations on Polynomials

- We have P1 and P2 arranged in a linked list in decreasing order of exponents.
- We can scan these and add like terms.
  - Need to store the resulting term only if it has non-zero coefficient.
- The number of terms in the result polynomial P1+P2 need not be known in advance.
- We'll use as much space as there are terms in P1+P2.

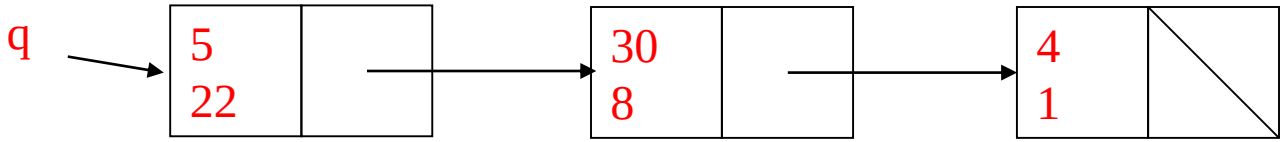
Addition of Two Polynomials

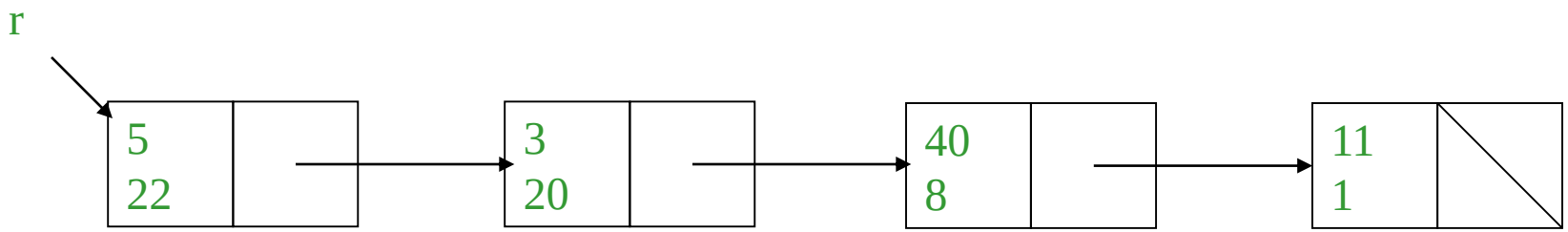
- One pass down each list: $\theta(n+m)$

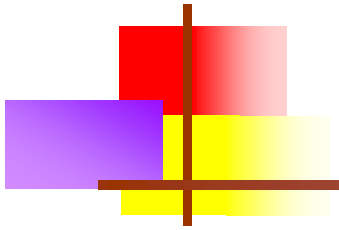
$7x+10x^8+3x^{20}$



$4x+30x^8+5x^{22}$

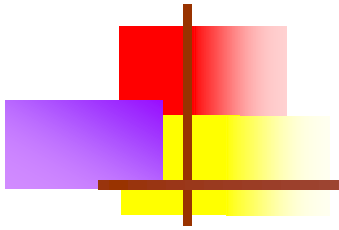




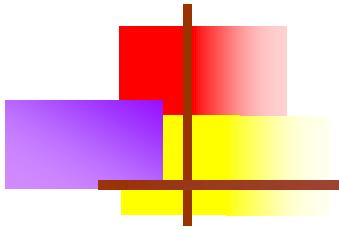


Further Operations

- Let us consider multiplication
- Can be done as repeated addition.
- So, multiply P1 with each term of P2.
- Add the resulting polynomials.

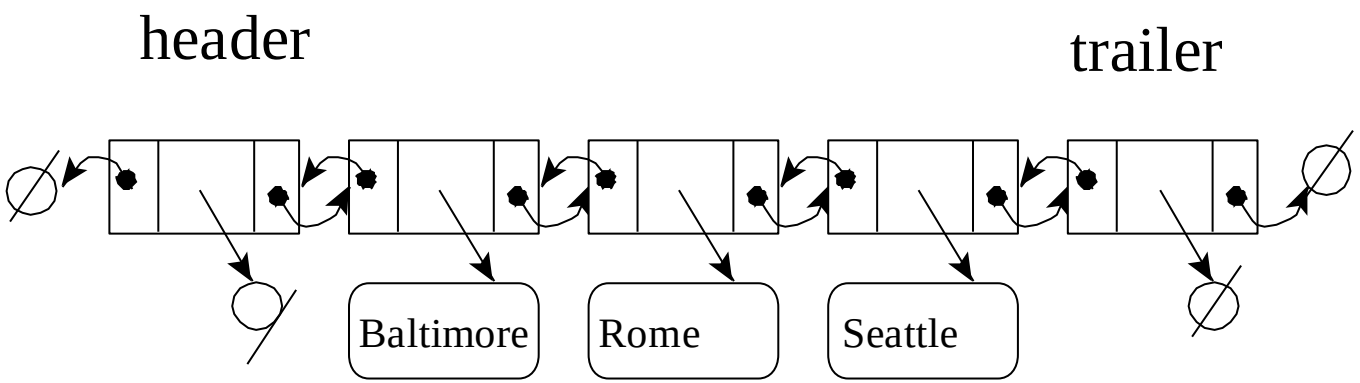


Doubly Linked Lists



- Permits traversal of list in both directions
- Useful where navigation in both directions needed
- Used by browsers to navigate forwards and backwards
- Various applications use this to redo and undo functionality (games)

For convenience, a doubly linked list has a **header** node and a **trailer** node. They are also called **sentinel** nodes, indicating both the ends of a list.

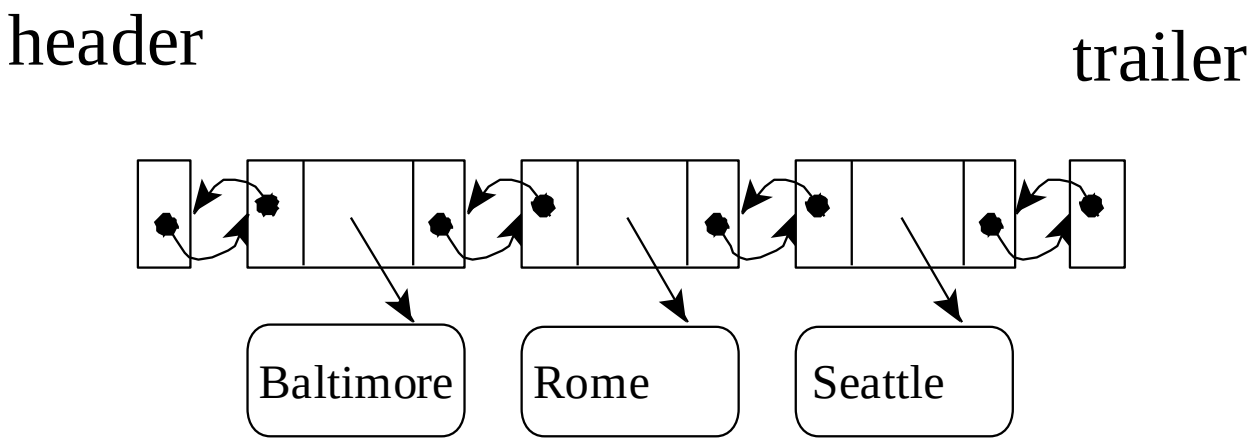


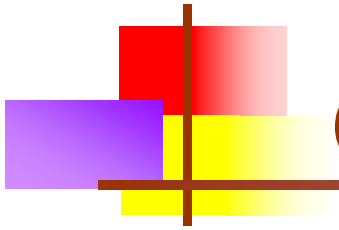
Difference from singly linked lists:

- each node contains two links.
- two extra nodes: header and trailer, which contain no elements.

A doubly linked list

Each node points to its prev node and its next node. Header node only points to next node.





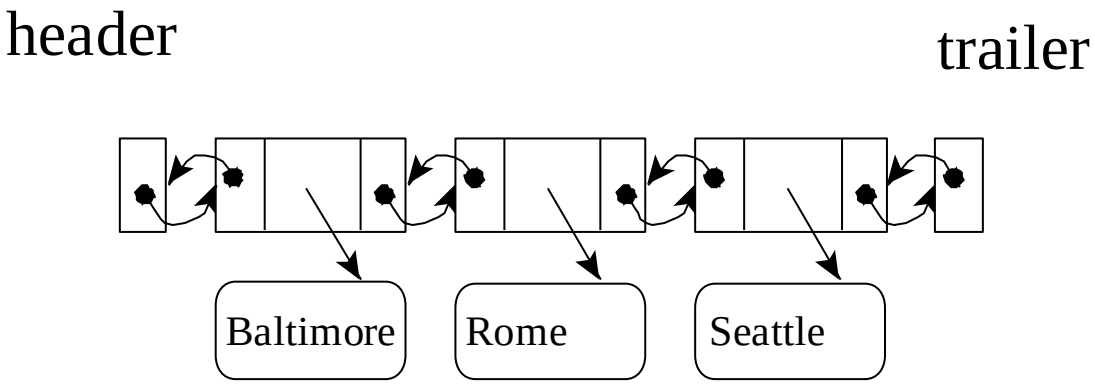
Class DLNode

Here is an implementation of nodes for doubly linked lists in Java:

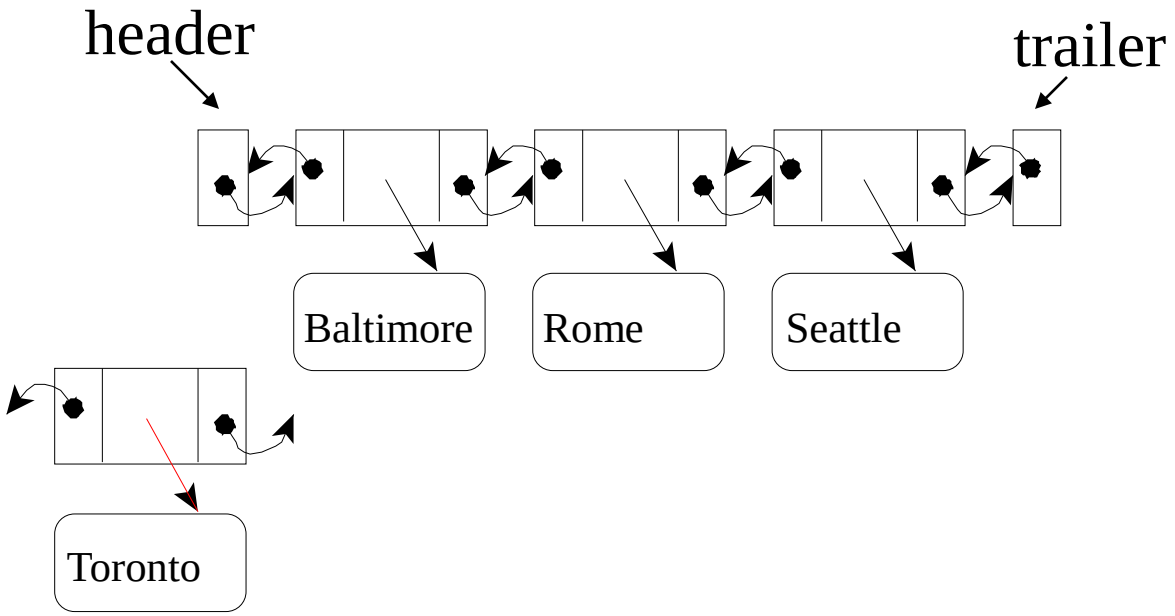
```
public class DLNode {
    private Object element;
    private DLNode next, prev;

    public DLNode() {
        this( null, null, null );
    }

    public DLNode( Object e, DLNode p, DLNode n
) {
        element = e;
        next = n;
        prev = p;
    }
```

Insertion of an Element at the Head

Before the insertion:

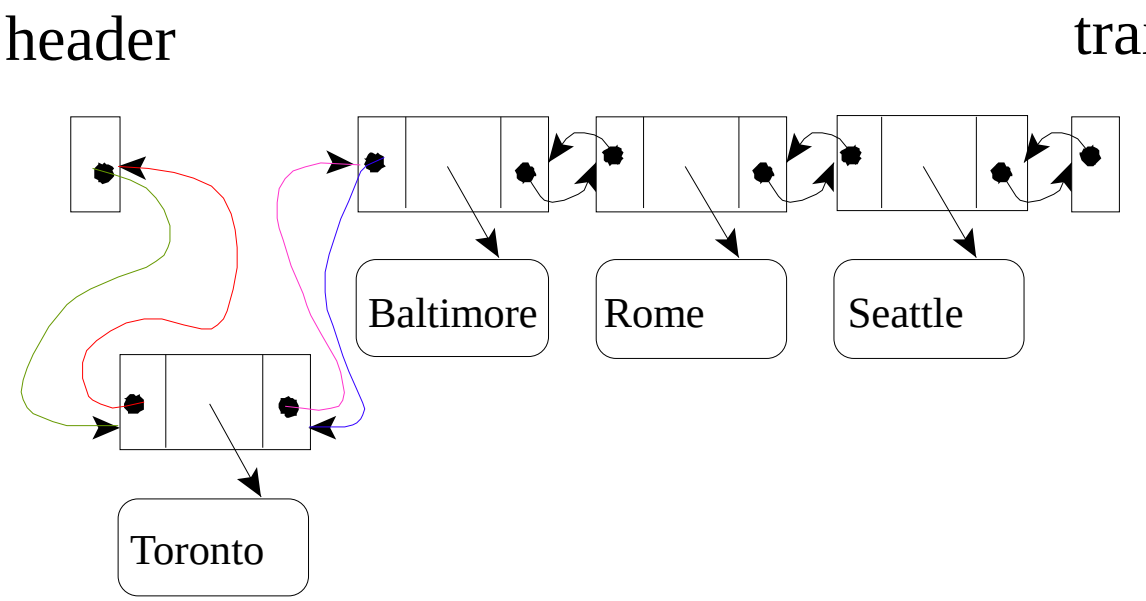


Have a new node:




```
DLNode x = new DLNode();
x.setElement(new String("Toronto"));
```

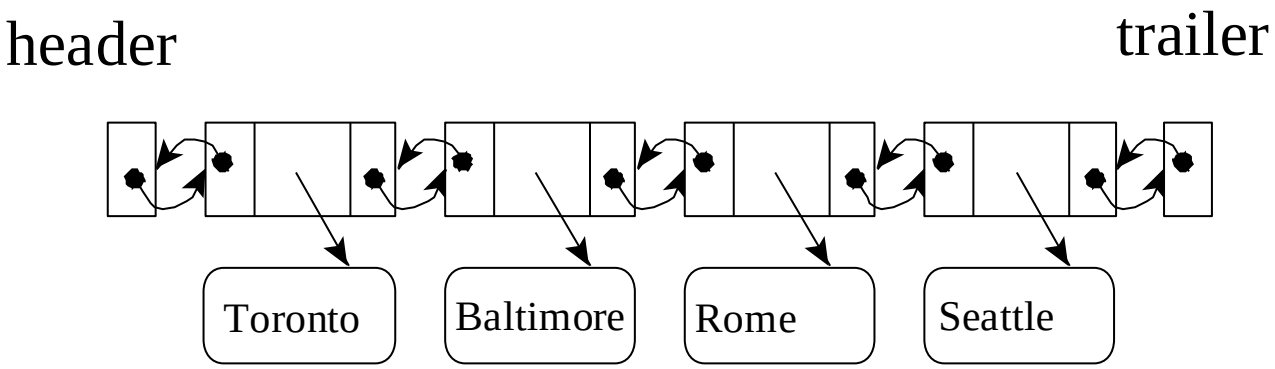
Update the links:




```
x.setPrev(header);
x.setNext(header.getNext());
(header.getNext()).setPrev(x);
header.setNext(x);
```

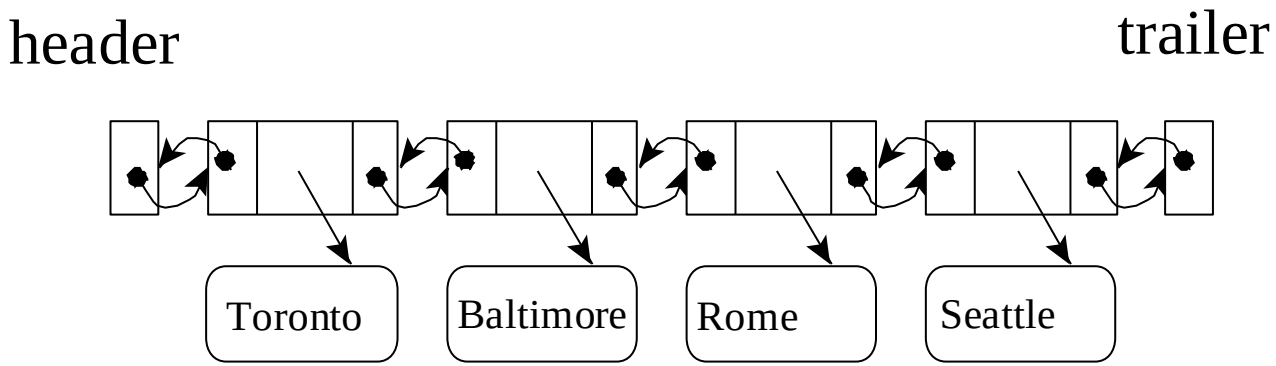
```
x.prev ← header;
x.next ← header.next;
header.next.prev ← x;
header.next ← x;
```

After the insertion:

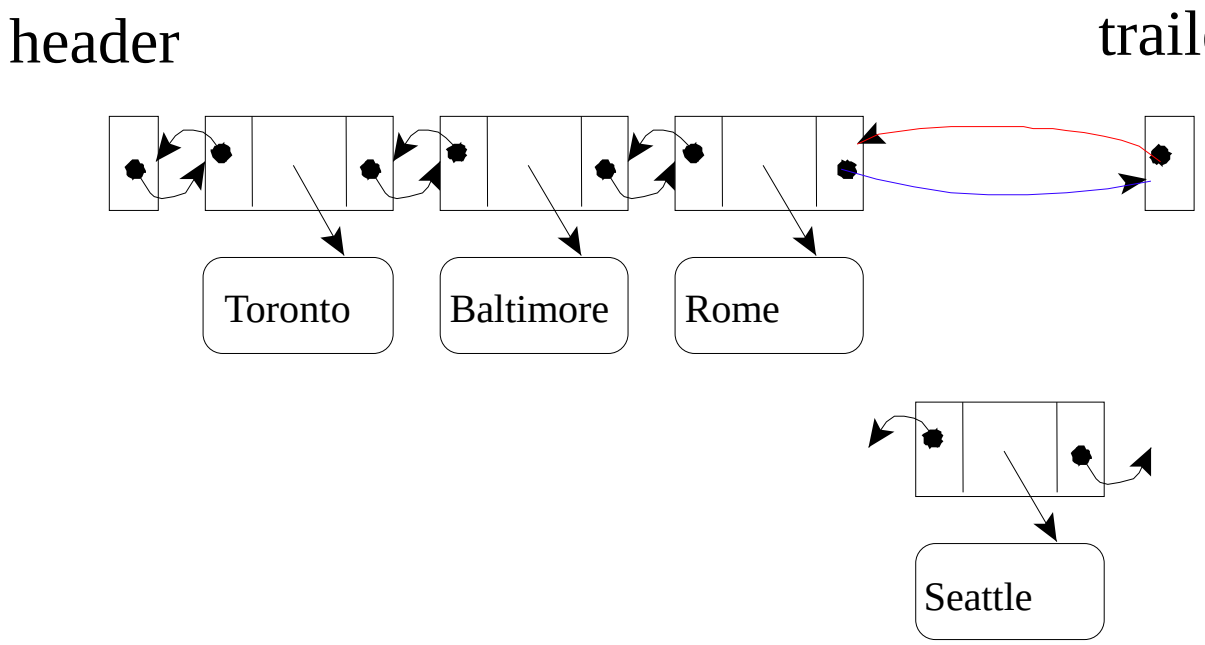


Deleting an Element at the Tail

Before the deletion:



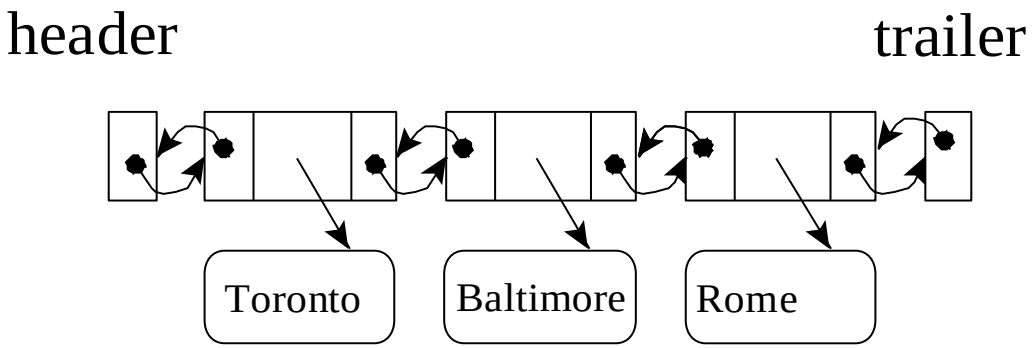
Update the links:

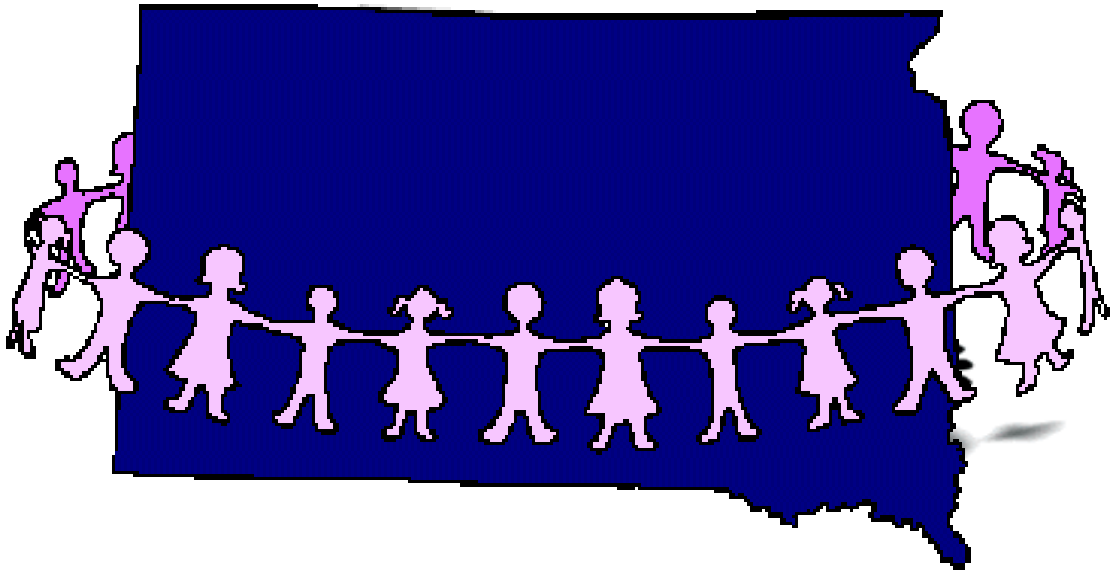



```
((trailer.getPrev()).getPrev()).setNext(trailer);
trailer.setPrev((trailer.getPrev()).getPrev());
```

```
trailer.prev.prev.next ← trailer;
trailer.prev ← trailer.prev.prev;
```

After the deletion:

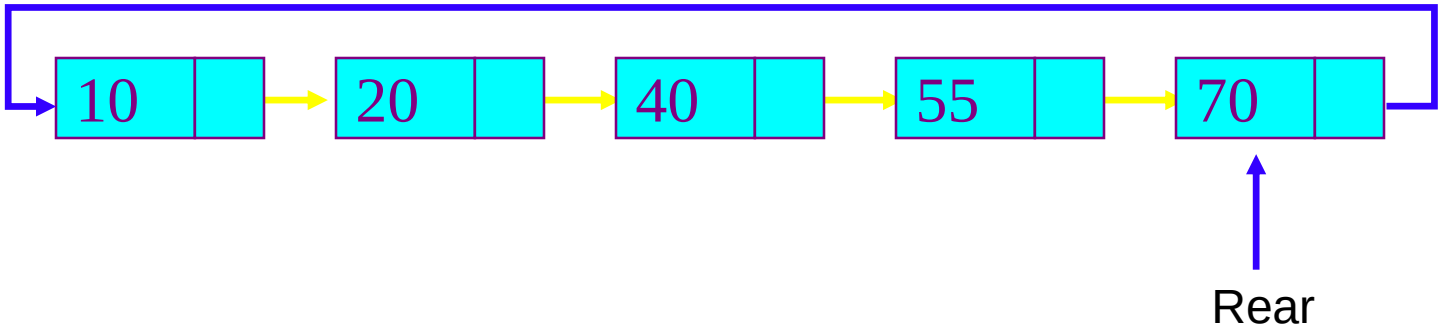


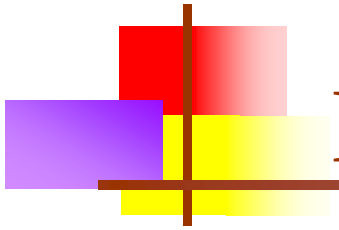


Circular Linked List

Circular Linked Lists

- A Circular Linked List is a special type of Linked List
- It supports traversing from the end of the list to the beginning by making the last node point back to the head of the list
- A Rear pointer is often used instead of a Head pointer



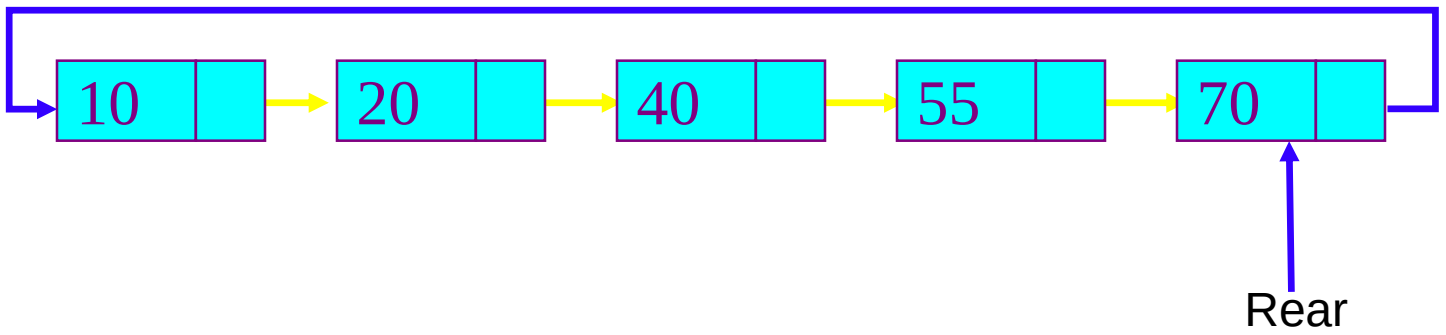


Motivation

- Circular linked lists are usually sorted
- Circular linked lists are useful for playing video and sound files in “looping” mode
- They are also a stepping stone to implementing graphs

Traverse the list

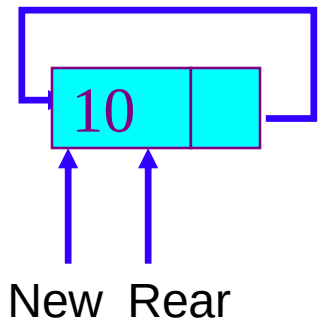
```
void print(NodePtr Rear){
   NodePtr Cur;
   if(Rear != NULL){
       Cur = Rear.getlink();
       do{
             print node data;
             Cur = Cur.getlink();
       }while(Cur != Rear.getlink();

   }
}
```



Insert Node

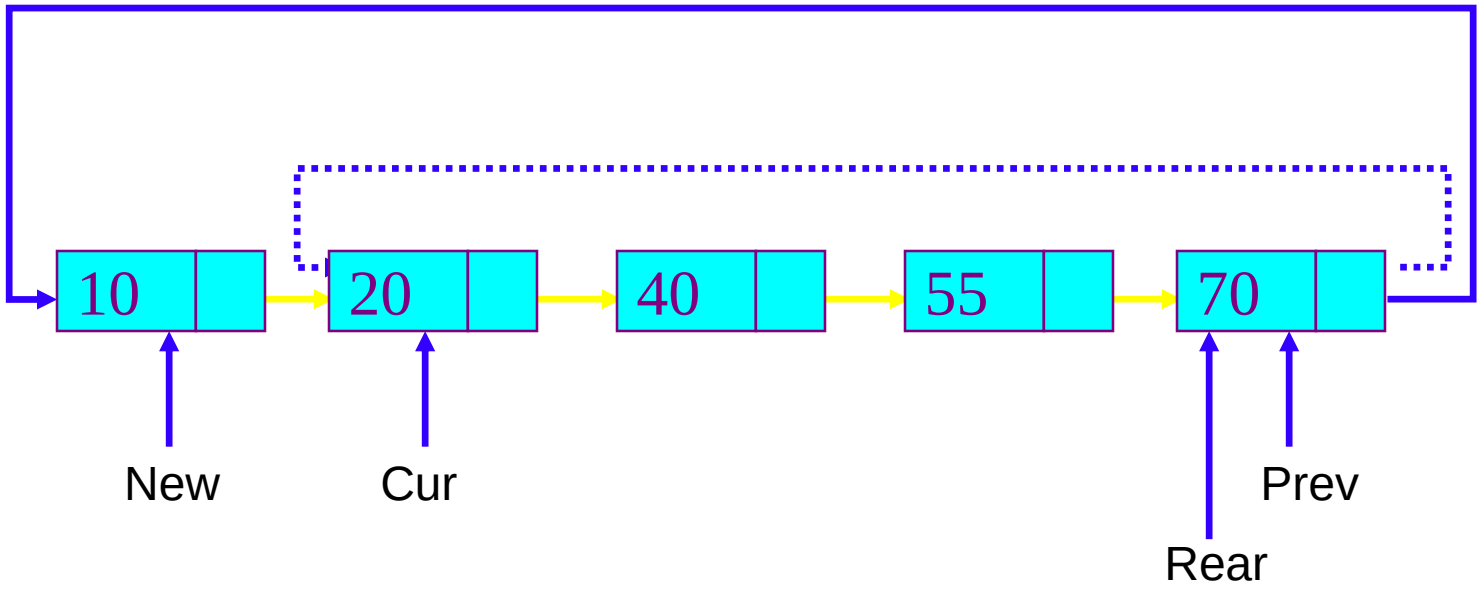
- Insert into an empty list

```
NotePtr New = new Node;
New.setData(10);

Rear = New;
Rear.setlink(Rear);
```



- Insert at head of a Circular Linked List

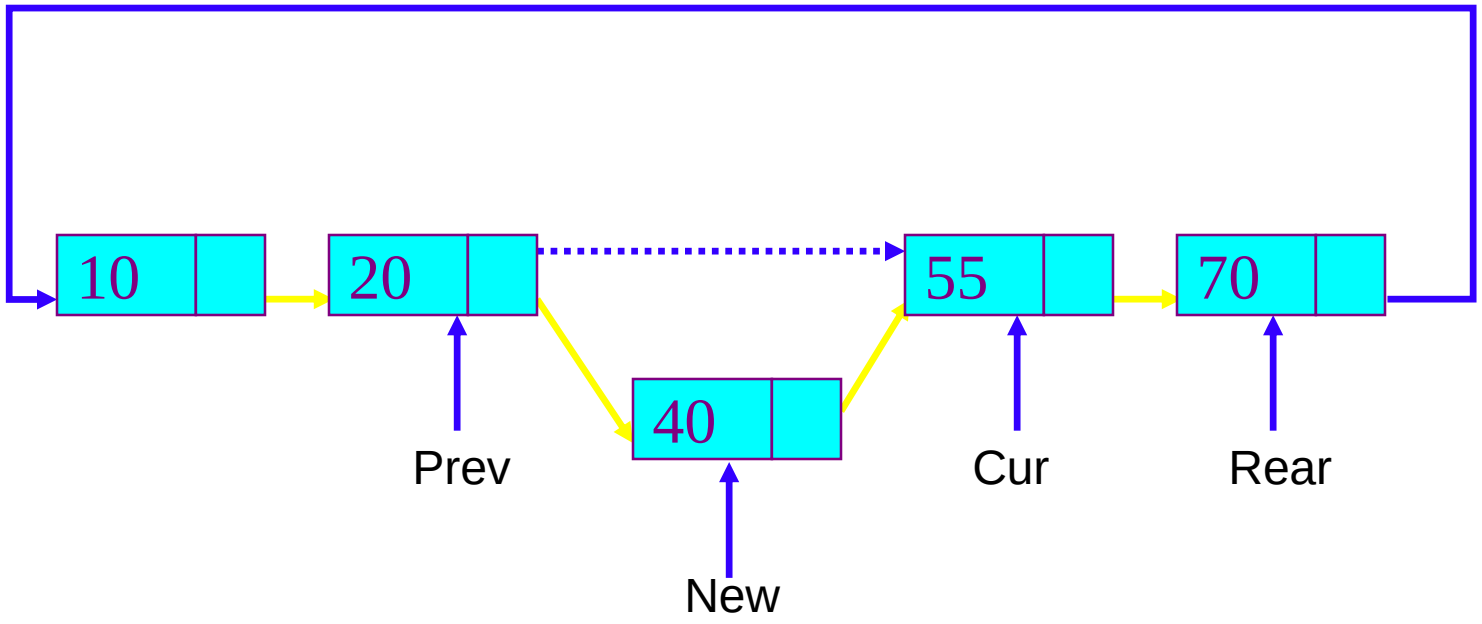
```
New.setlink(Cur);

Prev.setlink(New);
```



Insert in middle of a circular list

- Insert in middle of a Circular Linked List between `Pre` and `Cur`

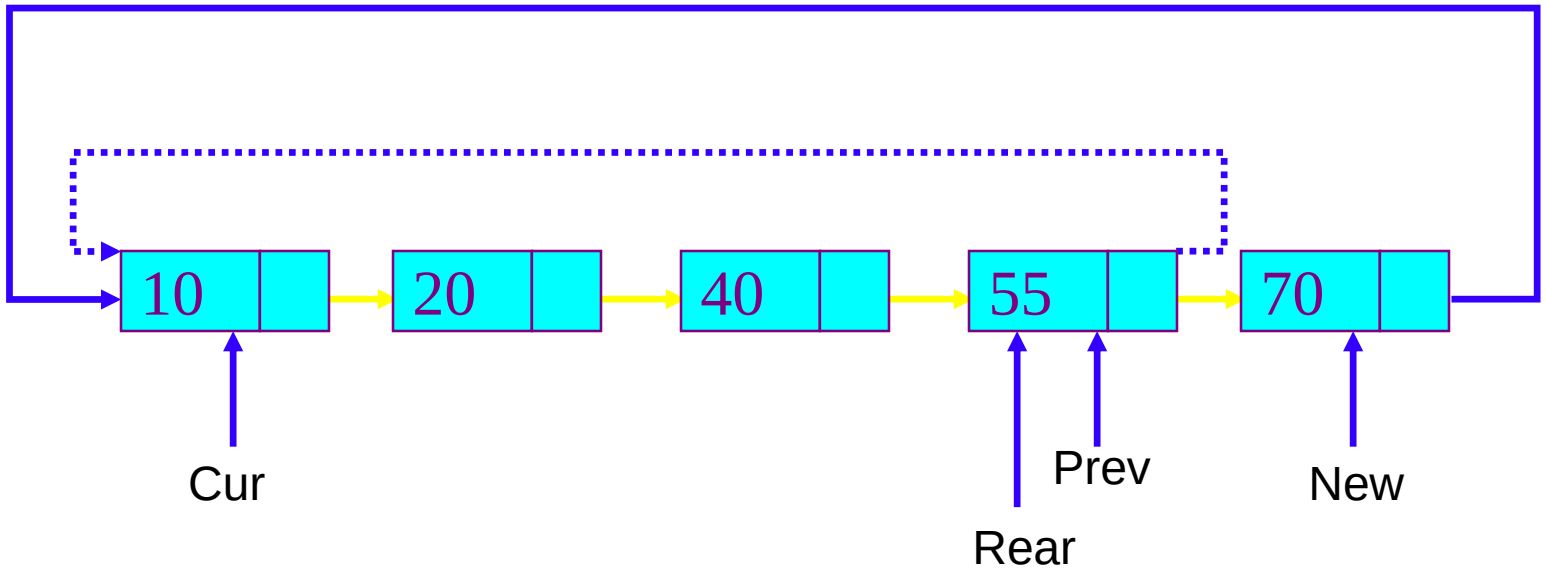
```
New.setlink(Cur);
Prev.setlink(New);
```



Insert at end

- Insert at end of a Circular Linked List

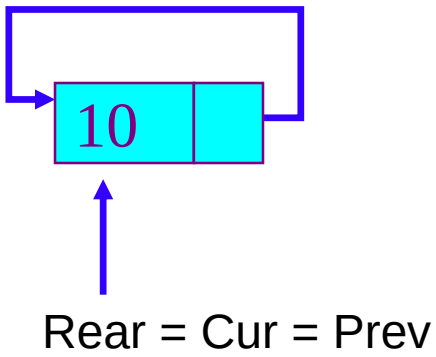
```
New.setlink(Cur);
Prev.setlink(New);
Rear = New;
```



Delete single remaining node

- Delete a node from a single-node Circular Linked List

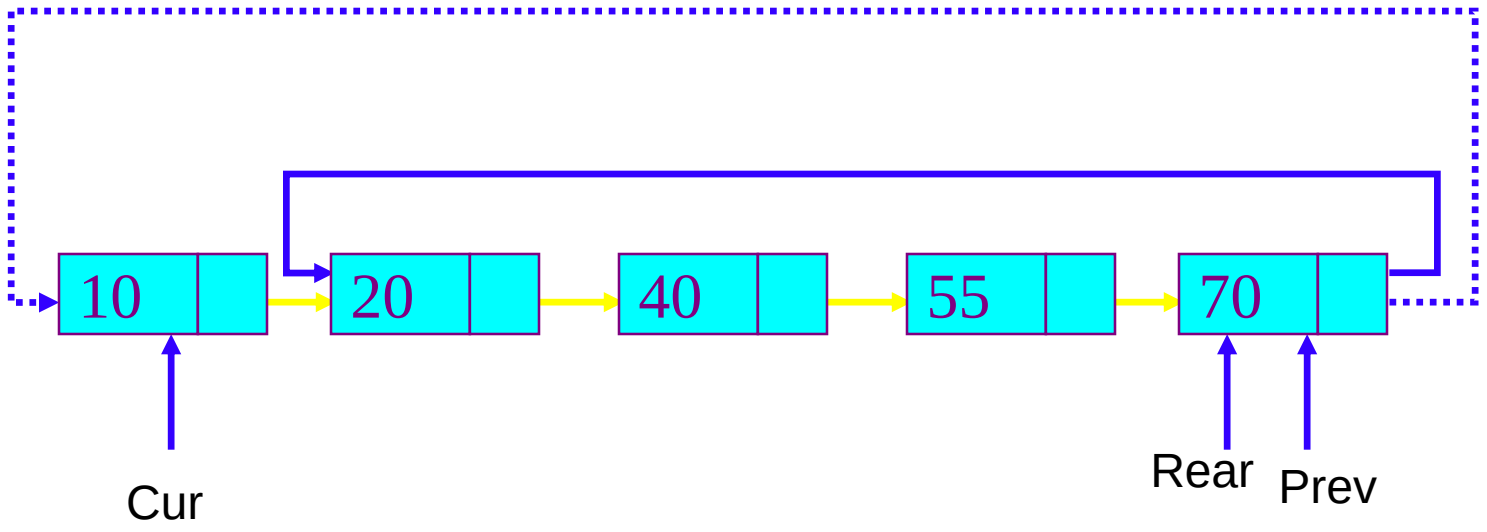
```
Rear = NULL;
delete Cur;
```



Delete Node

- Delete the head node from a Circular Linked List

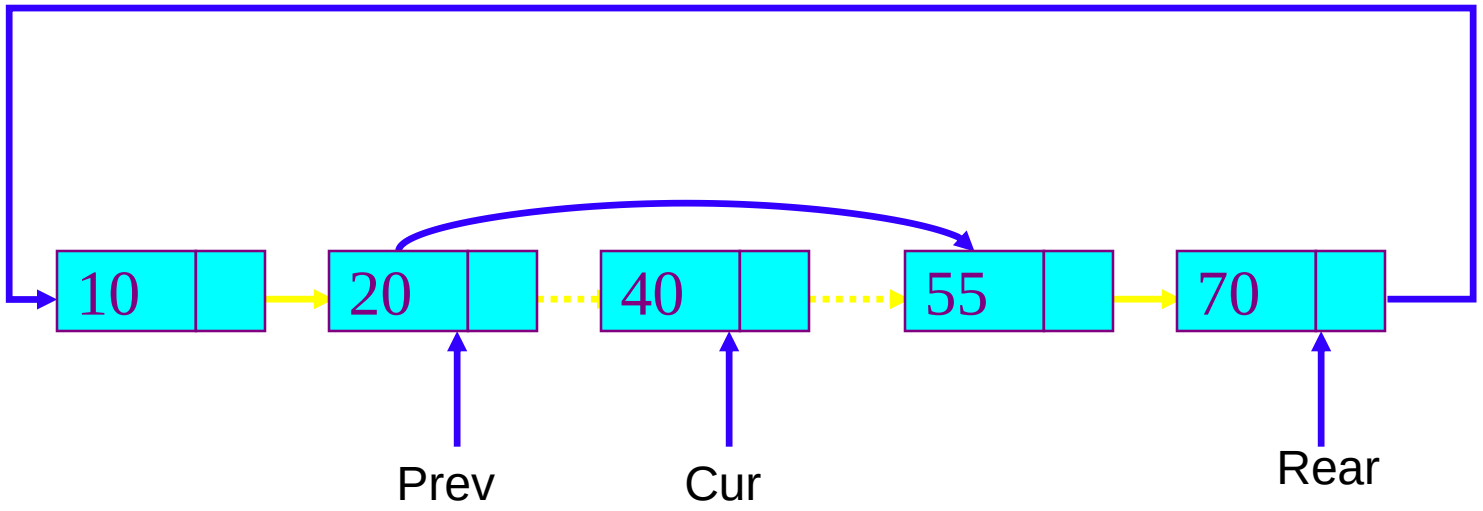
```
Prev.setlink( Cur.getlink());
```



Delete any node in middle of list

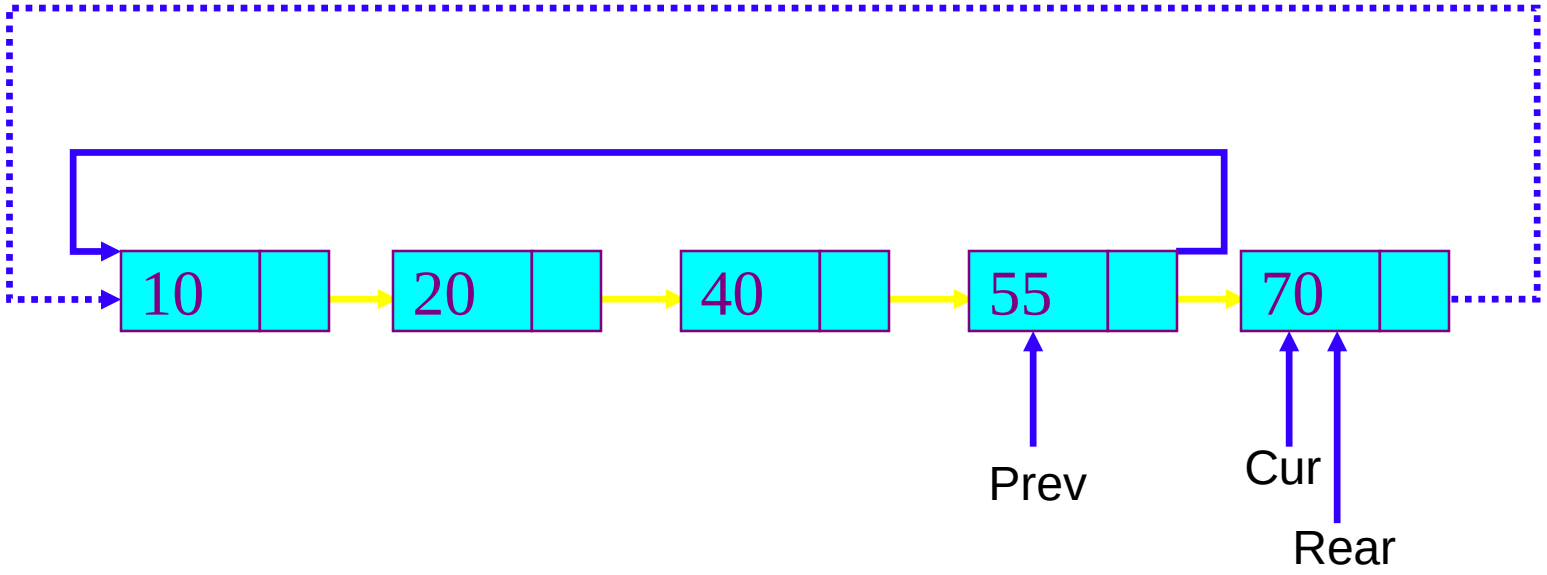
- Delete a middle node `Cur` from a Circular Linked List

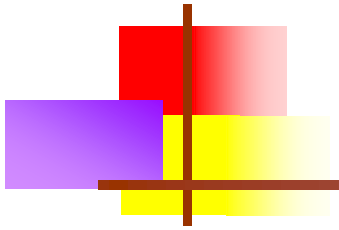
```
Prev.setlink(Cur.getlink());
```



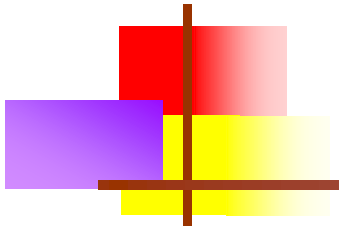
- Delete the end node from a Circular Linked List

```
Prev->next = Cur->next;    // same as: Rear->next;

delete Cur;

Rear = Prev;
```



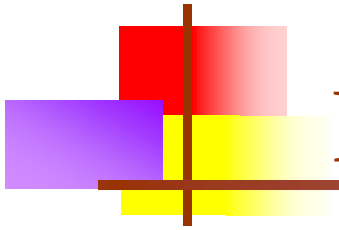


Radix sort



- [216 521 425 116 91 515 124 34 96 24]

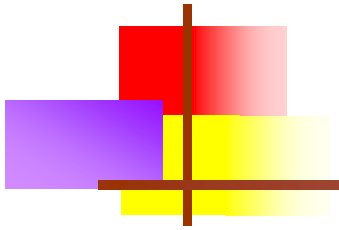
- Insertion sort or selection sort of n numbers has complexity of $O(n^2)$
- If the range of the numbers is known we can use a very fast method for sorting



RadixSort

- Radix = “The base of a number system” (Webster’s dictionary)
- History: used in 1890 U.S. census by Hollerith*
- Idea: BinSort on each digit, bottom up.

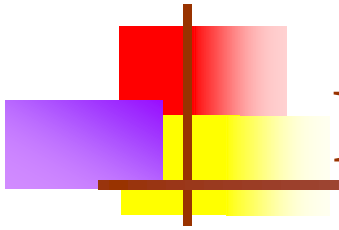
* Thanks to Richard Ladner for this fact, taken from Winter 1999 CSE 326 course web.



- The numbers are decomposed using some radix r.
- $928 = 9 * 10^2 + 2 * 10^1 + 8 * 10^0$

- Thus using the radix 10, the number 928 decomposes into digits 9, 2 and 8.
- The least significant digit is 8 and most significant digit is 9.

- In radix sort, the decomposed numbers are sorted by digits, starting with the least significant digit.



RadixSort – magic! It works.

- Input list:
  126, 328, 636, 341, 416, 131, 328

- BinSort on least significant digit:
  341, 131, 126, 636, 416, 328, 328

- BinSort result on next-higher digit:
  416, 126, 328, 328, 131, 636, 341

- BinSort that result on highest digit:
  126, 131, 328, 328, 341, 416, 636

Using radix sort on a linked list of integers

- Input list:
  126 → 328 → 636 → 341 → 416 → 131 → 328

- BinSort on least significant digit: 3 lists are formed
  341 → 131
- 126 → 636 → 416
- 328 → 328

- Link the 3 lists :
  341 → 131 → 126 → 636 → 416 → 328 → 328