Indian Institute of Information Technology Allahabad

Data Structures and Algorithms

Linked List

.. and other linked structures

Dr. Shiv Ram Dubey
Assistant Professor
Department of Information Technology
Indian Institute of Information Technology, Allahabad

Email: srdubey@iiita.ac.in Web: https://profile.iiita.ac.in/srdubey/

DISCLAIMER

The content (text, image, and graphics) used in this slide are adopted from many sources for academic purposes. Broadly, the sources have been given due credit appropriately. However, there is a chance of missing out some original primary sources. The authors of this material do not claim any copyright of such material.

Lists

- List is a sequence of data items of same type.
- Array – one way to represent a list.
  - Constant time access given index of an element

Lists

- List is a sequence of data items of same type.
- Array – one way to represent a list.
  - Constant time access given index of an element
- Problems with arrays
  - Size of an array should be specified beforehand (at least while dynamically allocating memory).
  - Deleting/Inserting an element requires shifting of elements.
  - Wasted space.

Lists

- List is a sequence of data items of same type.
- Array – one way to represent a list.
  - Constant time access given index of an element
- Problems with arrays
  - Size of an array should be specified beforehand (at least while dynamically allocating memory).
  - Deleting/Inserting an element requires shifting of elements.
  - Wasted space.

Polynomial: x^25 + 3x^7 – 4

Store in an array poly[26]
poly[i] contains coefficient of x^i

poly[0] = -4, poly[7] = 3, poly[25] = 1
poly[i] = 0 for all i != 0,7,25

Can we avoid storing so many 0's?
Store (0,-4), (7,3), (25,1) instead.
How do we 'link' these pairs?

Dynamic Memory Allocation: Review

```
typedef struct {
    int hiTemp;
    int loTemp;
    double precip;
} WeatherData;
```

```
int main ( ) {
        int numdays;
        WeatherData *days;
        scanf (“%d”, &numdays) ;
        days=(WeatherData *)malloc
        (sizeof(WeatherData)*numdays);
        if (days == NULL)
            printf (“Insufficient memory\n”);
        ...
        free (days) ;
}
```

Self-Referential Structures

A structure referencing itself – how?

Self-Referential Structures

A structure referencing itself – how?

So, we need a pointer inside a structure that points to a structure of the same type.

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

Self-Referential Structures

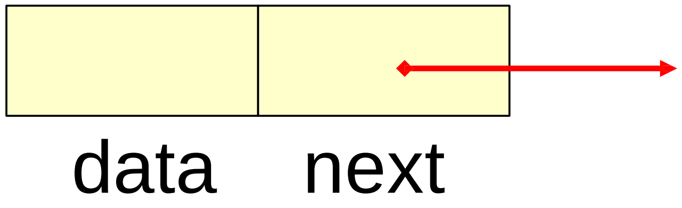
```
struct list {
        int data ;
        struct list *next ;
} ;
```

The pointer variable next is called a link.

Each structure is linked to a succeeding structure by next.

Pictorial Representation

A structure of type struct list



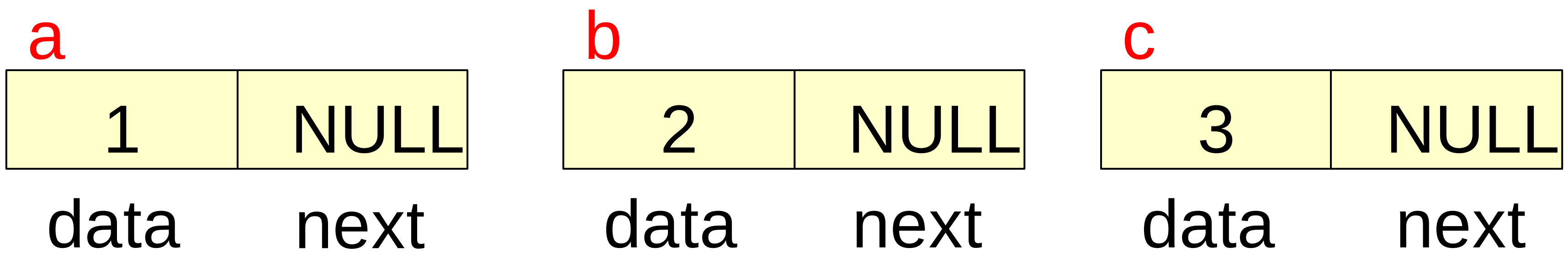
The pointer variable next contains either

- an address of the location in memory of the successor list element
- or the special value NULL defined as 0.

NULL is used to denote the end of the list.

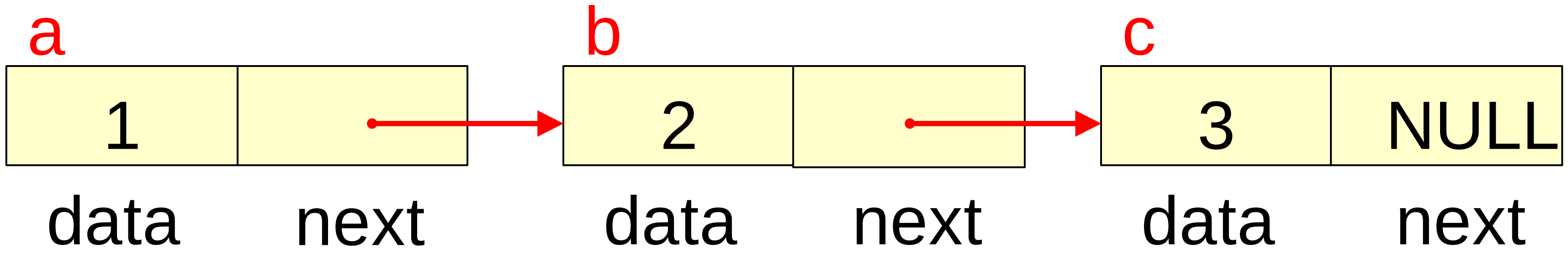
Pictorial Representation

```
struct list a, b, c;
a.data = 1; b.data = 2; c.data = 3;
a.next = b.next = c.next = NULL;
```



Pictorial Representation

```
a.next = &b;
b.next = &c;
```

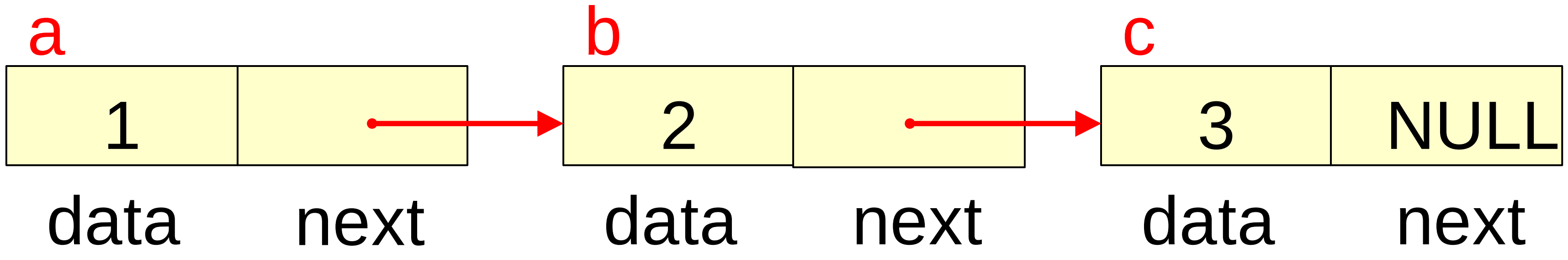


What are the values of :

- a.data
- a.next->data
- a.next->next->data

Pictorial Representation

a.next = &b;
b.next = &c;



What are the values of :

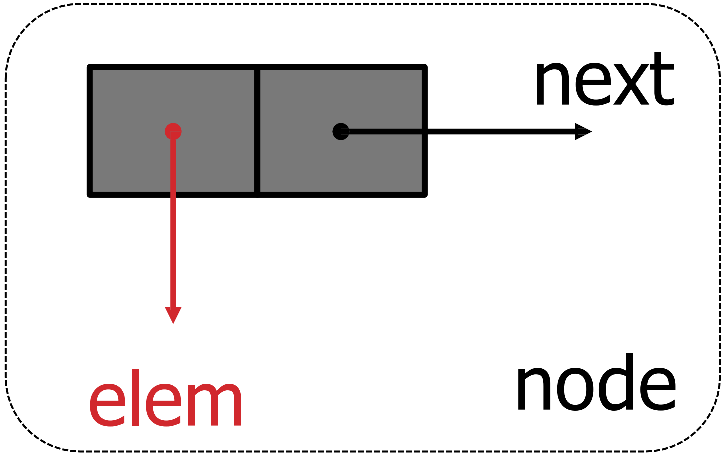
- a.data 1
- a.next->data 2
- a.next->next->data 3

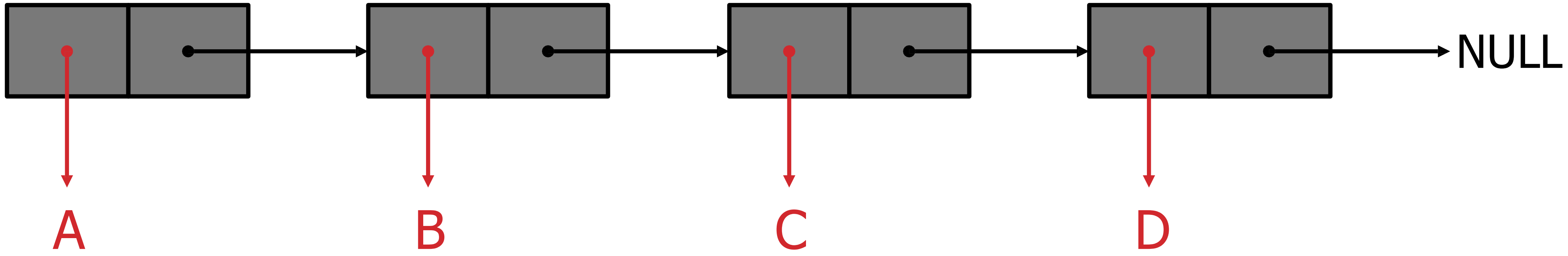
Linked Lists

A singly linked list is a concrete data structure consisting of a sequence of nodes

Each node stores

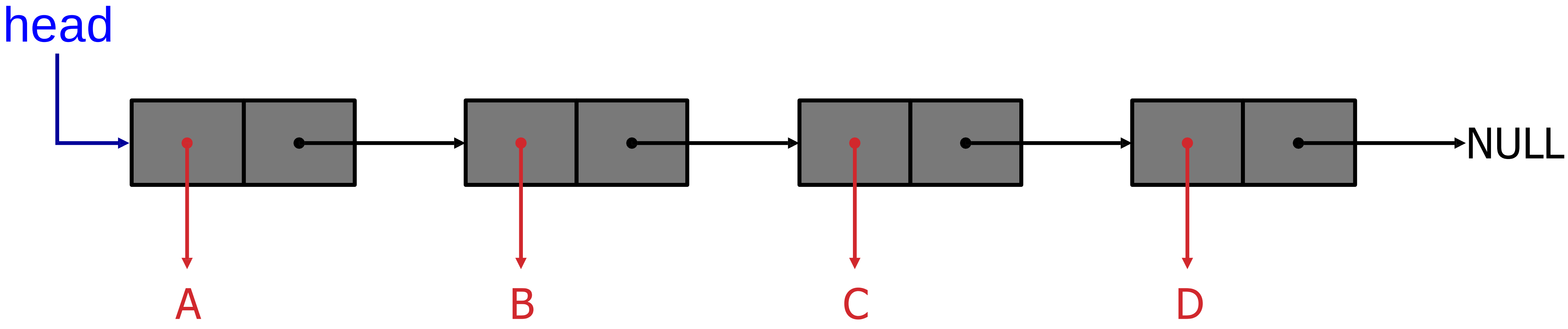
- element
- link to the next node





Linear Linked Lists

- A head pointer addresses the first element of the list.
- Each element points at a successor element.
- The last element has a link value NULL.



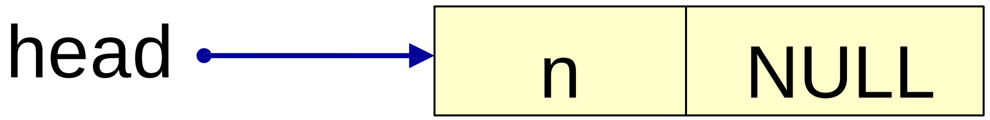
Header File: list.h

```
#include <stdio.h>

#include <stdlib.h>

typedef char DATA;

struct list {

        DATA d;

        struct list *next;

};
typedef struct list ELEMENT;
typedef ELEMENT *LINK;
```

Storage Allocation

```
LINK head ;
head = (LINK) malloc (sizeof(ELEMENT));
head->d = ‘n’;
head->next = NULL;
```

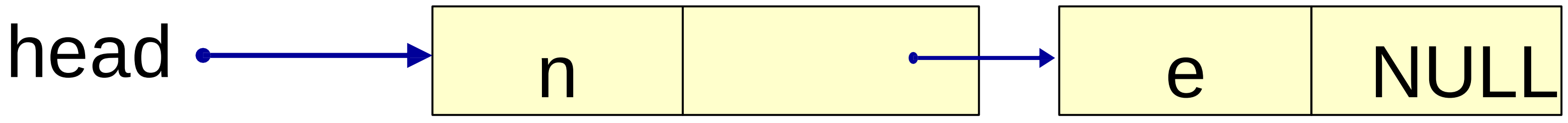
creates a single element list.



Storage Allocation

```
head->next = (LINK) malloc (sizeof(ELEMENT));
head->next->d = ‘e’;
head->next->next = NULL;
```

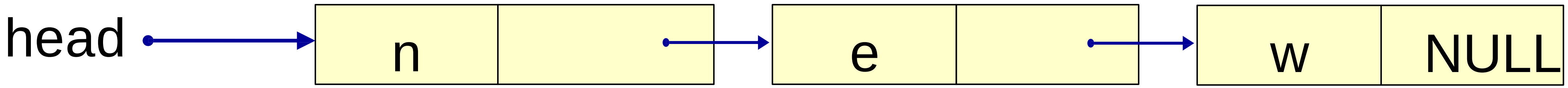
A second element is added.



Storage Allocation

```
head->next->next = (LINK) malloc (sizeof(ELEMENT));
head->next->next->d = 'w';
head->next->next->next = NULL;
```

We have a 3-element list pointed to by head.
The list ends when next has the sentinel value NULL.



List Operations

- How to initialize such a self referential structure (LIST),
- How to insert such a structure into the LIST,
- How to delete elements from it,
- How to search for an element in it,
- How to print it,
- How to free the space occupied by the LIST?

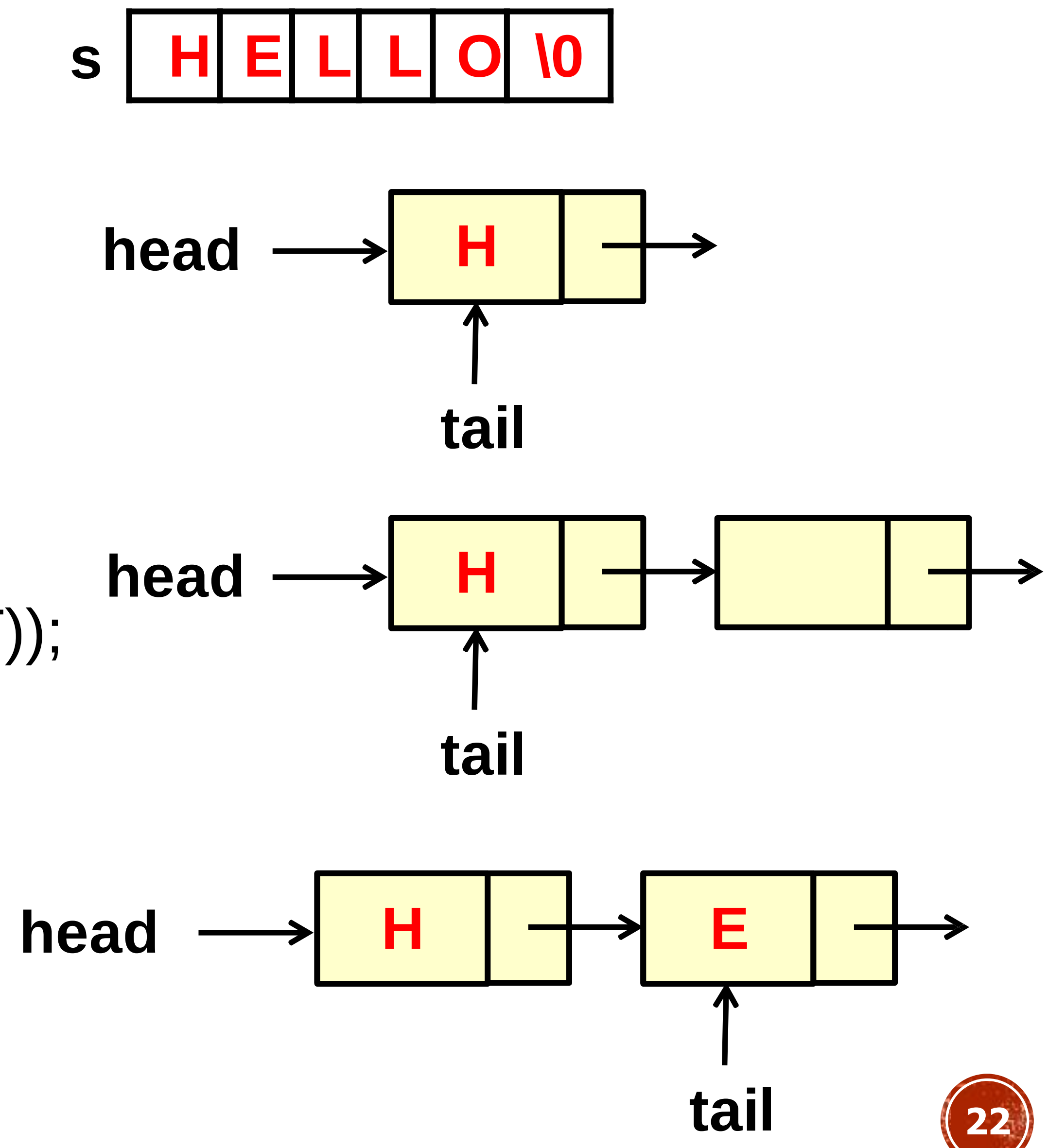
Produce a list from a string (Recursive Version)

```
#include <stdio.h>
#include <stdlib.h>
typedef char DATA;
struct list {
    DATA d;
    struct list *next;
};
typedef struct list
ELEMENT;
typedef ELEMENT
*LINK;
```

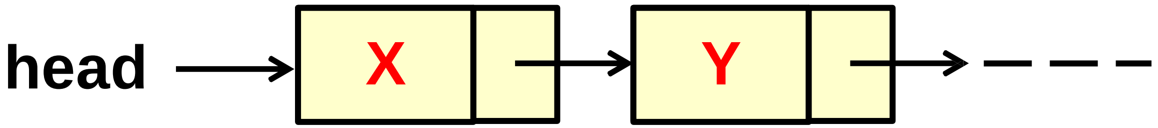
```
LINK StrToList (char s[ ]) {
LINK head ;
if (s[0] == '\0') return NULL ;
else{
        head = (LINK) malloc (sizeof(ELEMENT));
        head->d = s[0];
        head->next = StrToList (s+1);
        return head;
}
```

Produce a list from a string (Iterative Version)

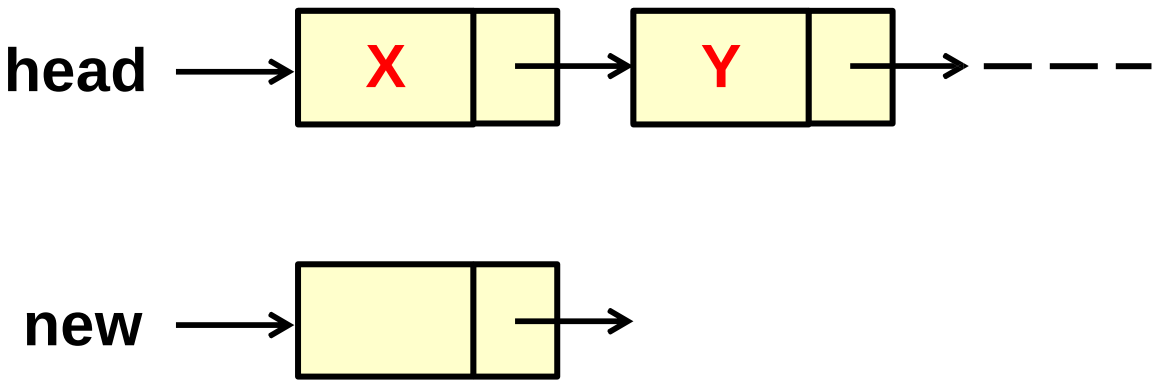
```
LINK SToL (char s[ ]) {
  LINK head = NULL, tail; int i;
  if (s[0] != '\0'){
     head = (LINK) malloc (sizeof(ELEMENT));
     head->d = s[0];
     tail = head;
     for (i=1; s[i] != '\0'; i++){
       tail->next=(LINK)malloc(sizeof(ELEMENT));
       tail = tail->next;  tail->d = s[i];
     }
  tail->next = NULL;
  }
return head;
}
```



Inserting at the Head



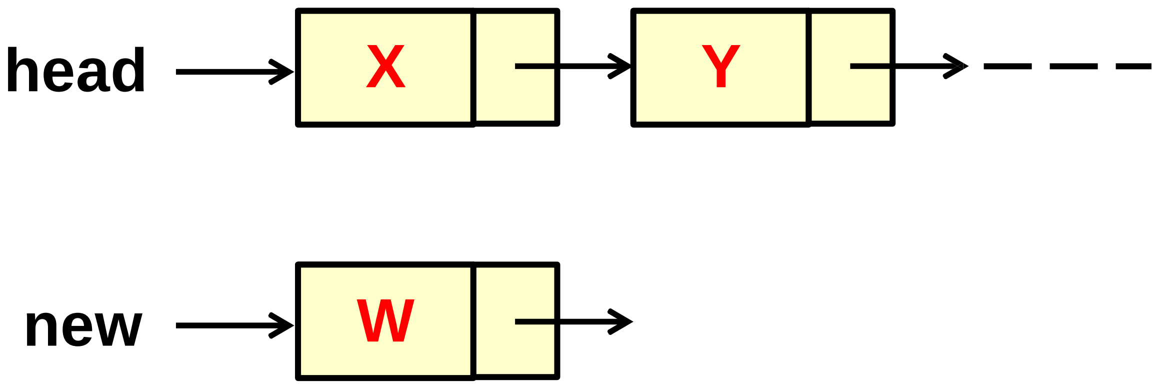
Inserting at the Head



1. Allocate a new node

new = malloc(sizeof(ELEMENT));

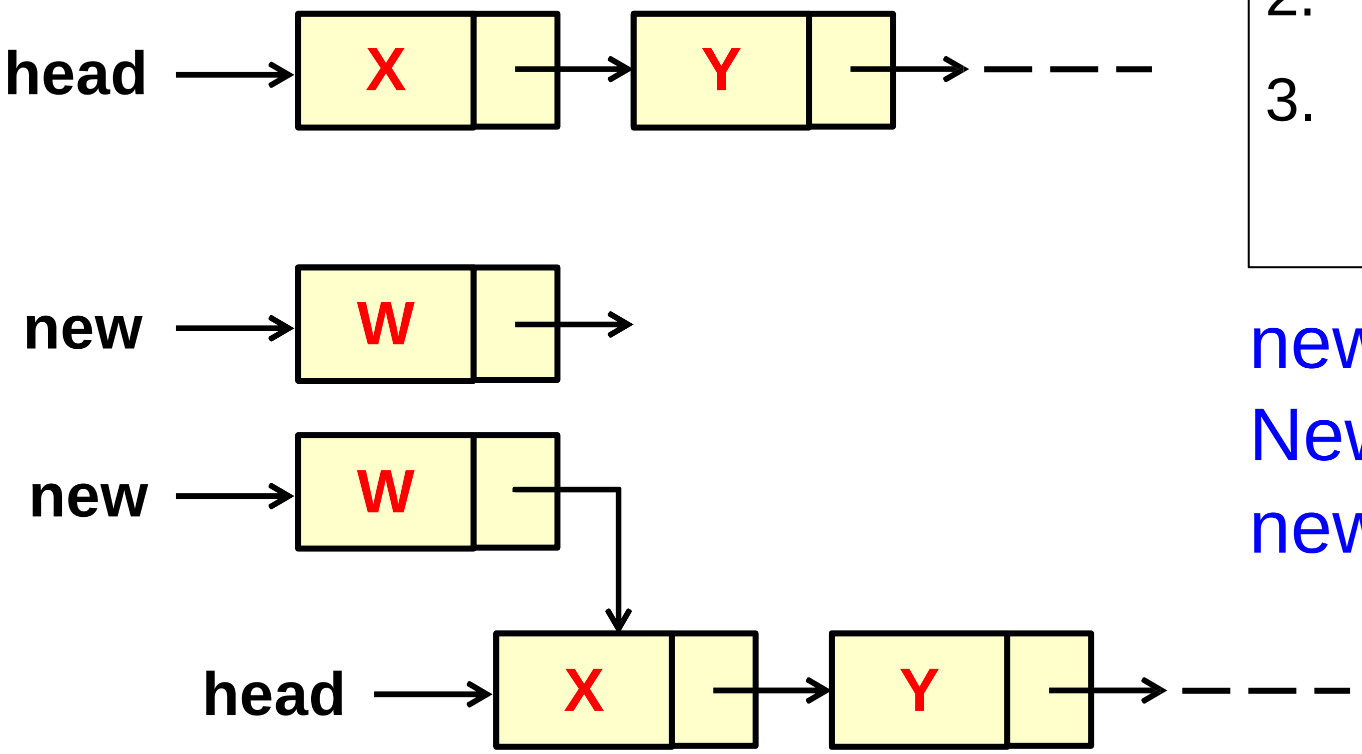
Inserting at the Head



1. Allocate a new node
2. Insert new element

new = malloc(sizeof(ELEMENT));
New->d = ‘W’;

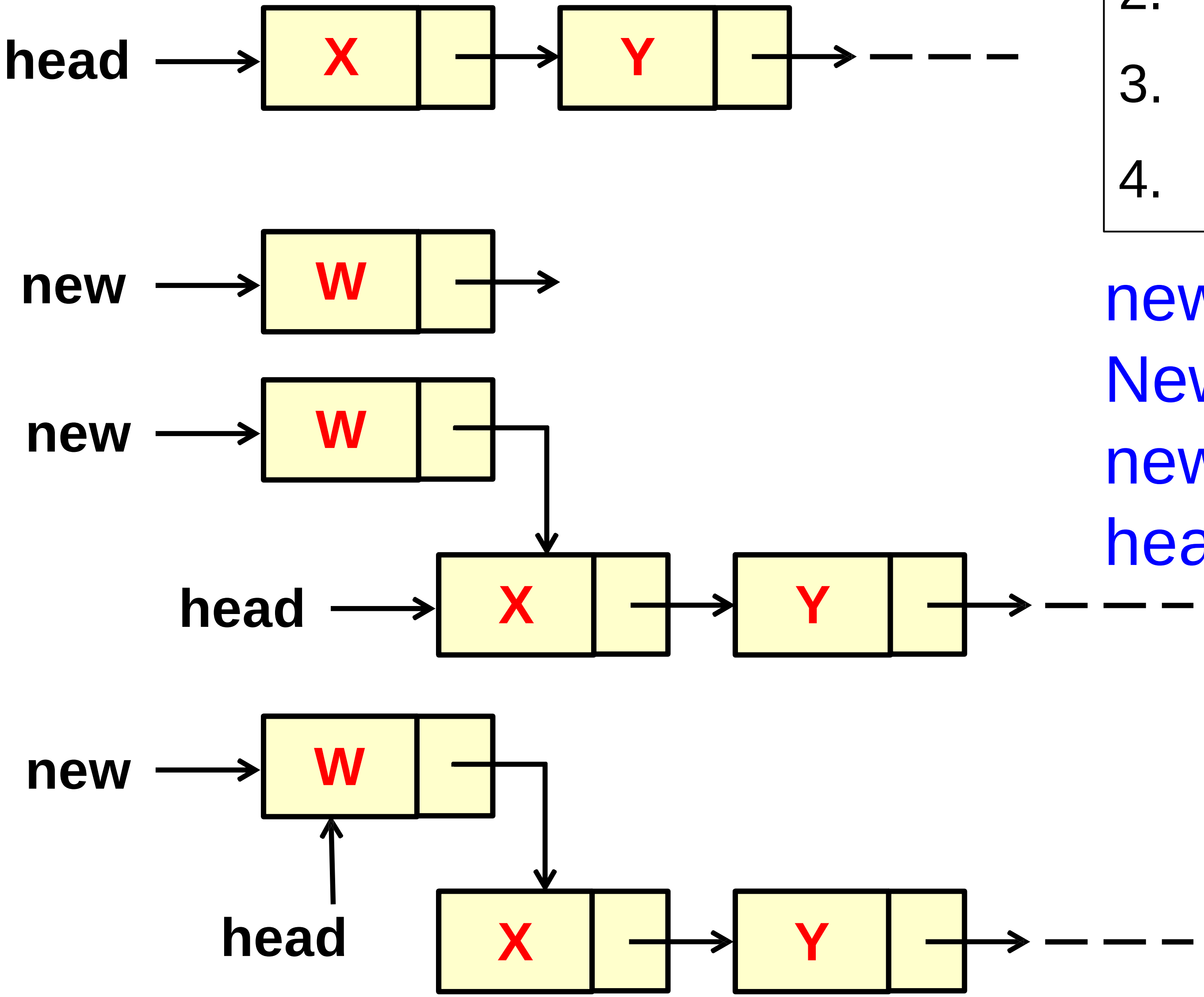
Inserting at the Head



1. Allocate a new node
2. Insert new element
3. Make new node point to old head

```
new = malloc(sizeof(ELEMENT));
New->d = 'W';
new-> next = head;
```

Inserting at the Head

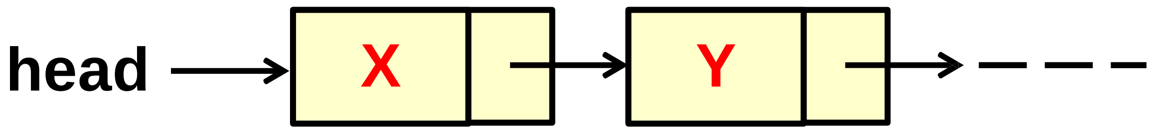


1. Allocate a new node
2. Insert new element
3. Make new node point to old head
4. Update head to point to new node

```
new = malloc(sizeof(ELEMENT));
New->d = 'W';
new-> next = head;
head = new;
```

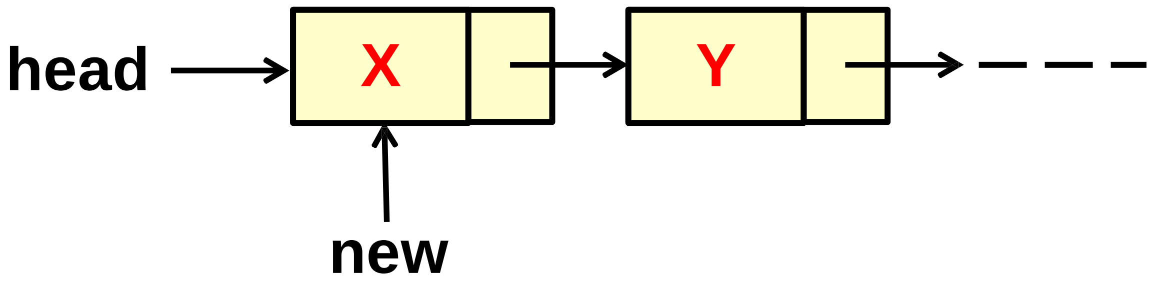
Removing the Head

1. Update head to point to next node in the list
2. Allow garbage collector to reclaim the former first node



Removing the Head

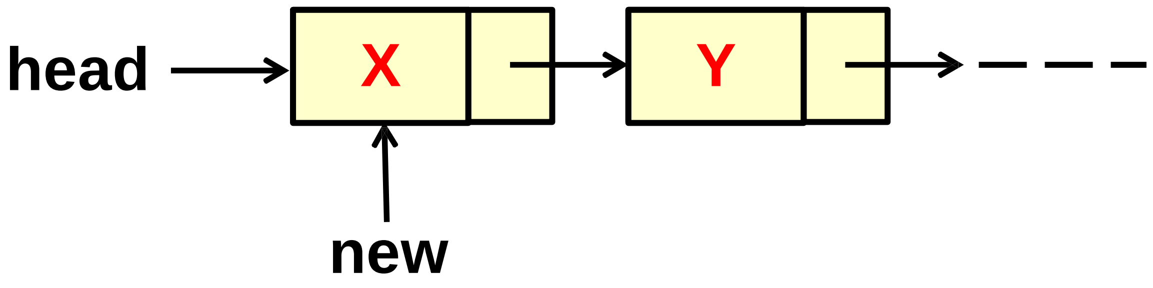
1. Update head to point to next node in the list
2. Allow garbage collector to reclaim the former first node



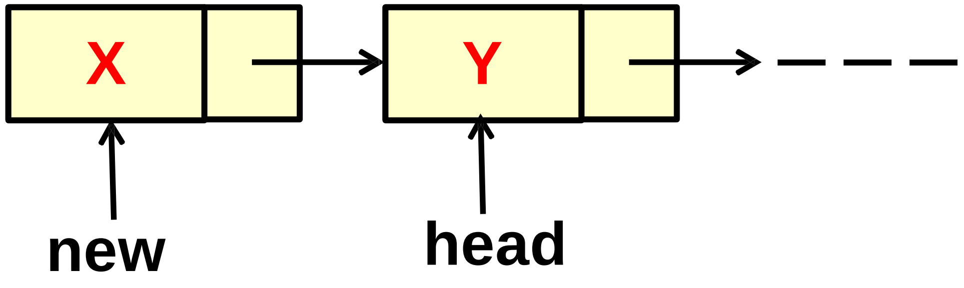
new = head;

Removing the Head

1. Update head to point to next node in the list
2. Allow garbage collector to reclaim the former first node



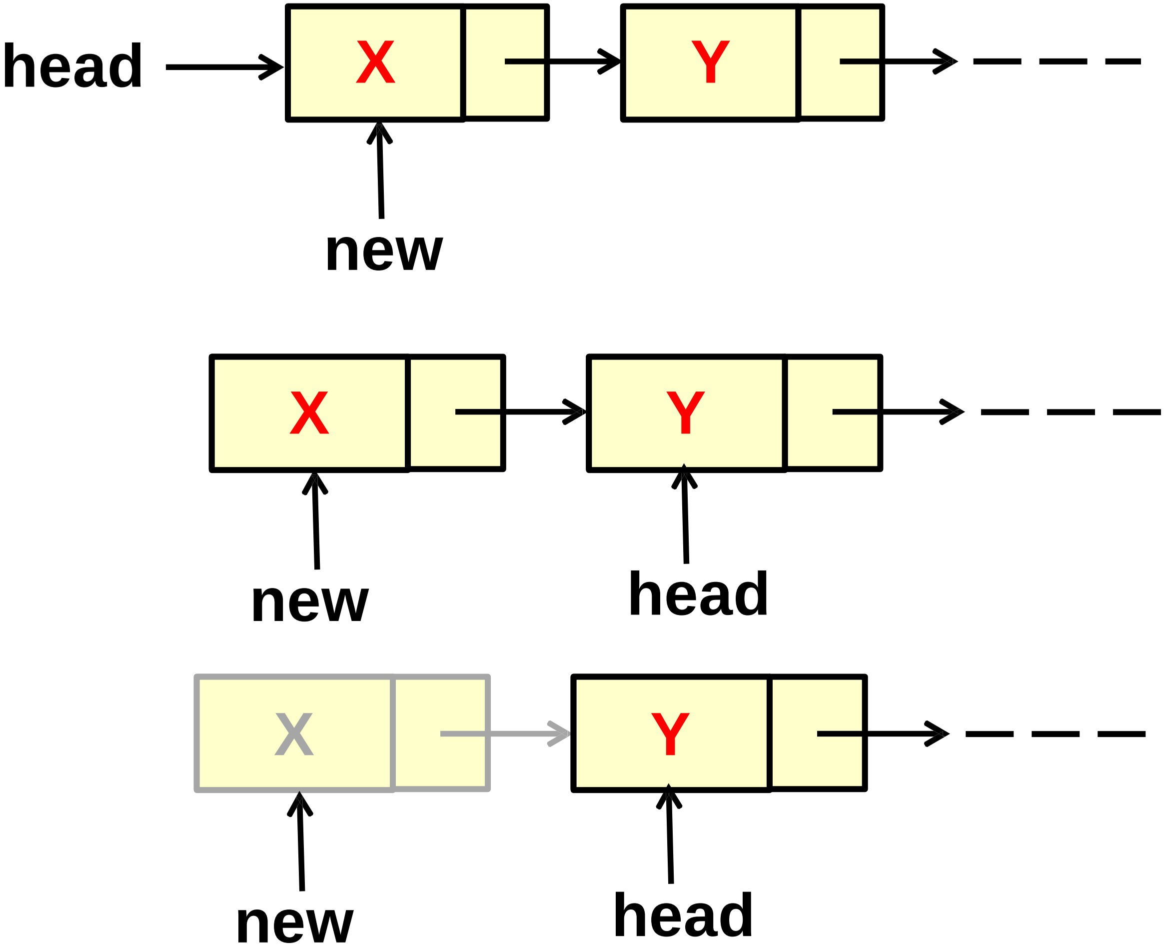
new = head;



head = new->next;

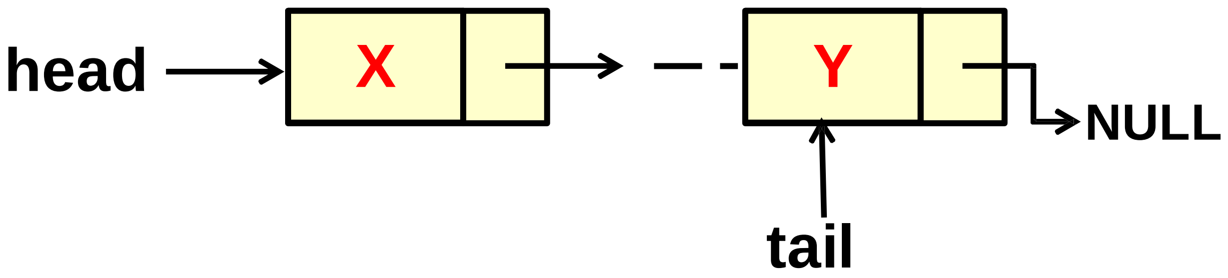
Removing the Head

1. Update head to point to next node in the list
2. Allow garbage collector to reclaim the former first node

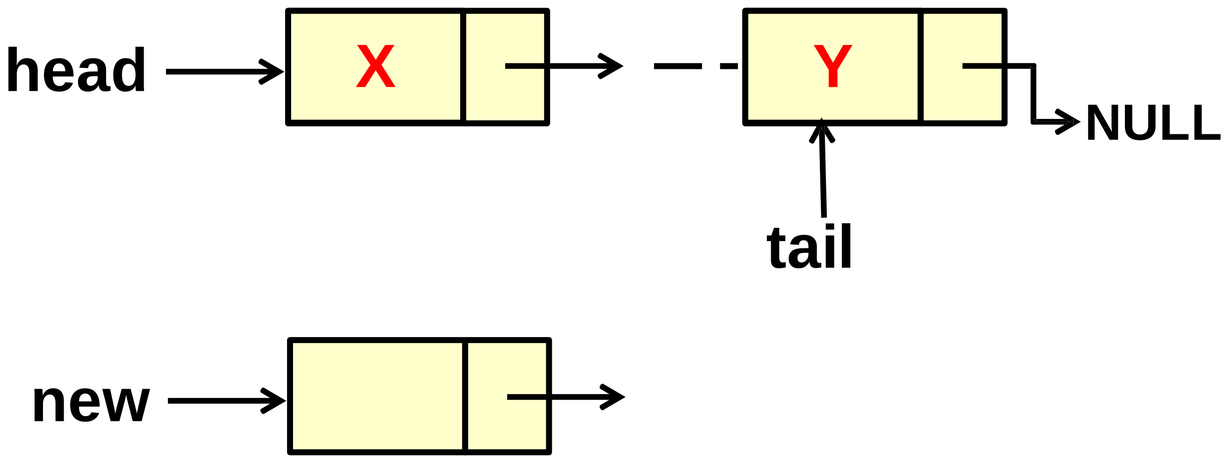



```
new = head;
head = new->next;
free(new);
```

Inserting at the Tail



Inserting at the Tail

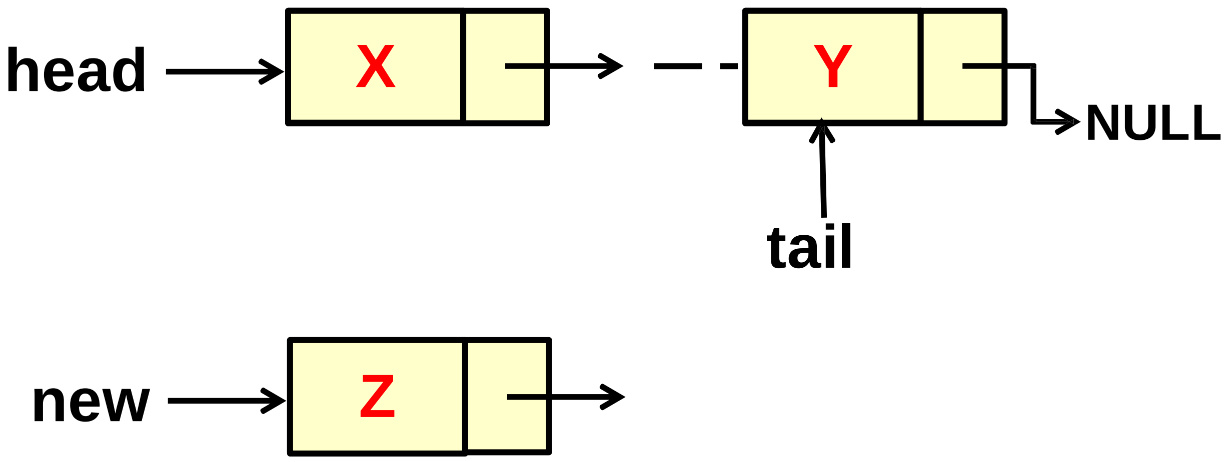


1. Allocate a new node

new = malloc(sizeof(ELEMENT));

Inserting at the Tail

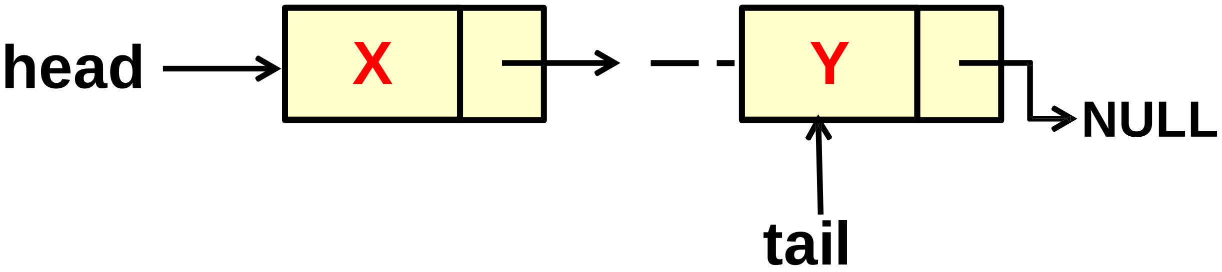
1. Allocate a new node
2. Insert new element



new = malloc(sizeof(ELEMENT));
new->d = ‘Z’;

Inserting at the Tail

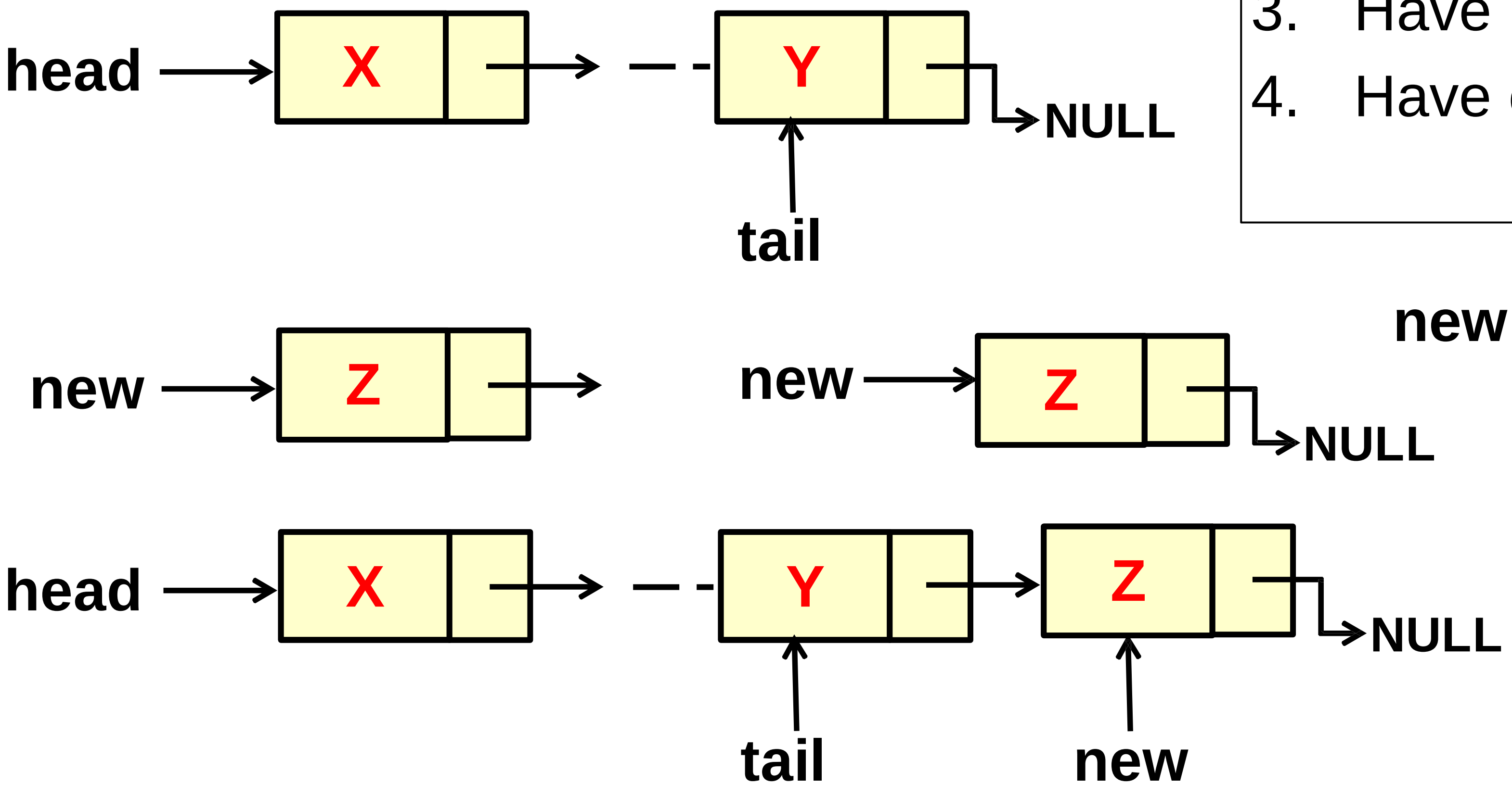
1. Allocate a new node
2. Insert new element
3. Have new node point to null




```
new = malloc(sizeof(ELEMENT));
new->d = 'Z';
new->next = NULL;
```

Inserting at the Tail

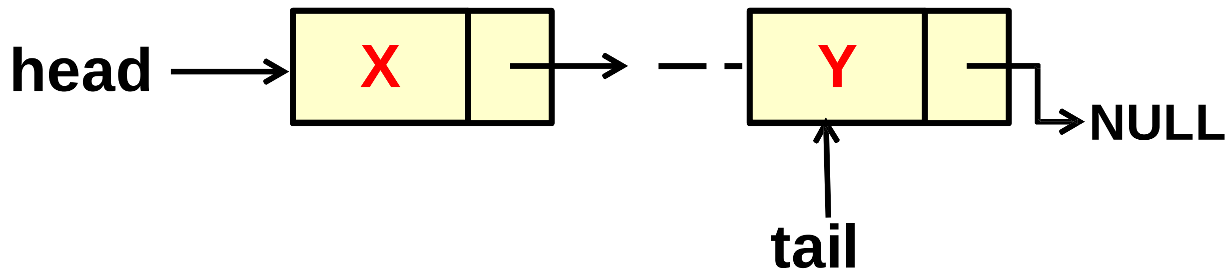
1. Allocate a new node
2. Insert new element
3. Have new node point to null
4. Have old last node point to new node



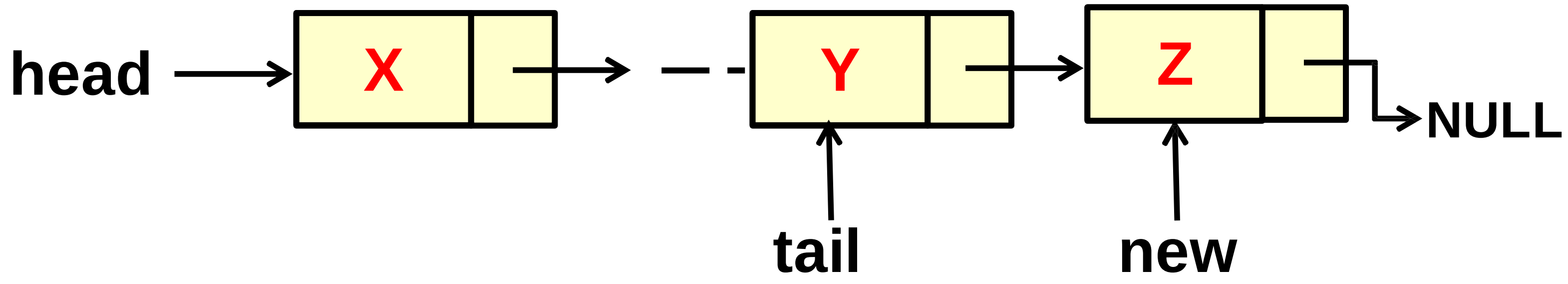

```
new = malloc(sizeof(ELEMENT));
new->d = ‘Z’;
new->next = NULL;

tail->next = new;
```

Inserting at the Tail

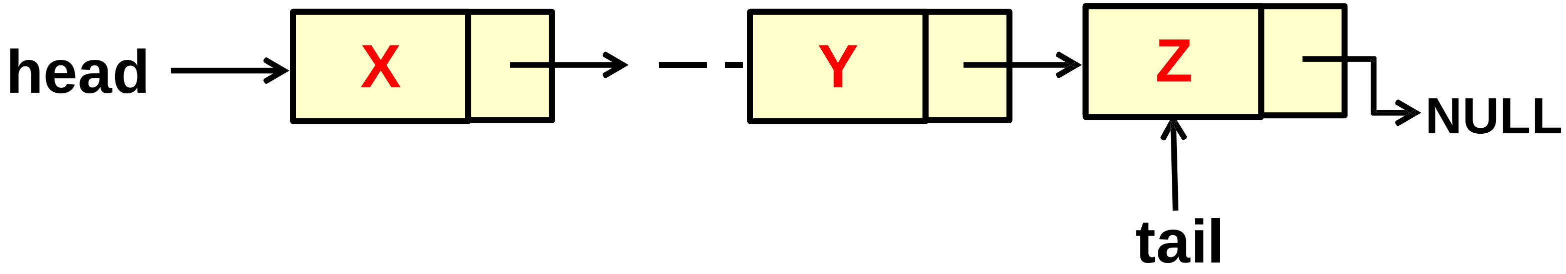
1. Allocate a new node
2. Insert new element
3. Have new node point to null
4. Have old last node point to new node
5. Update tail to point to new node




```
new = malloc(sizeof(ELEMENT));
new->d = 'Z';
new->next = NULL;
```



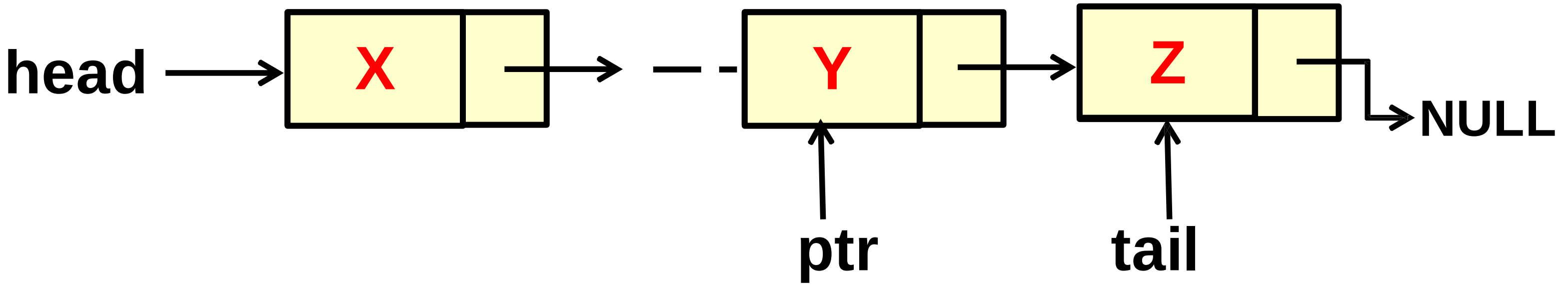

```
tail->next = new;
```




```
tail = new;
```

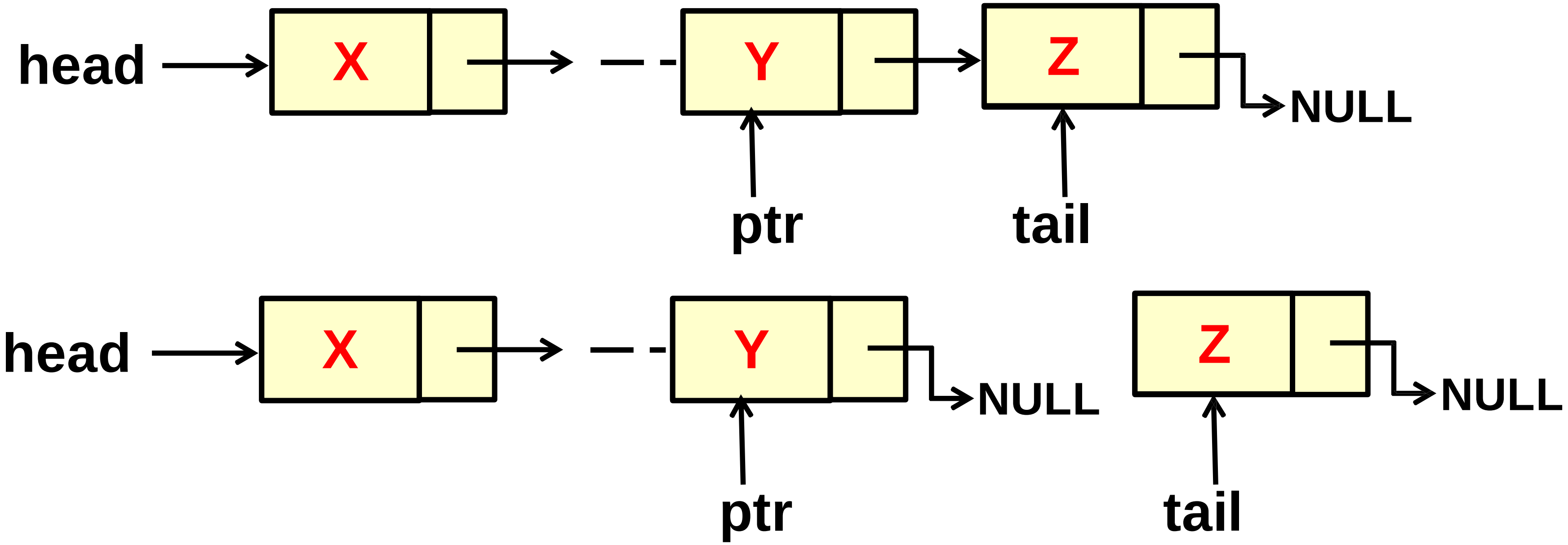
Removing the Tail

1. Bring ptr to the second last node



Removing the Tail

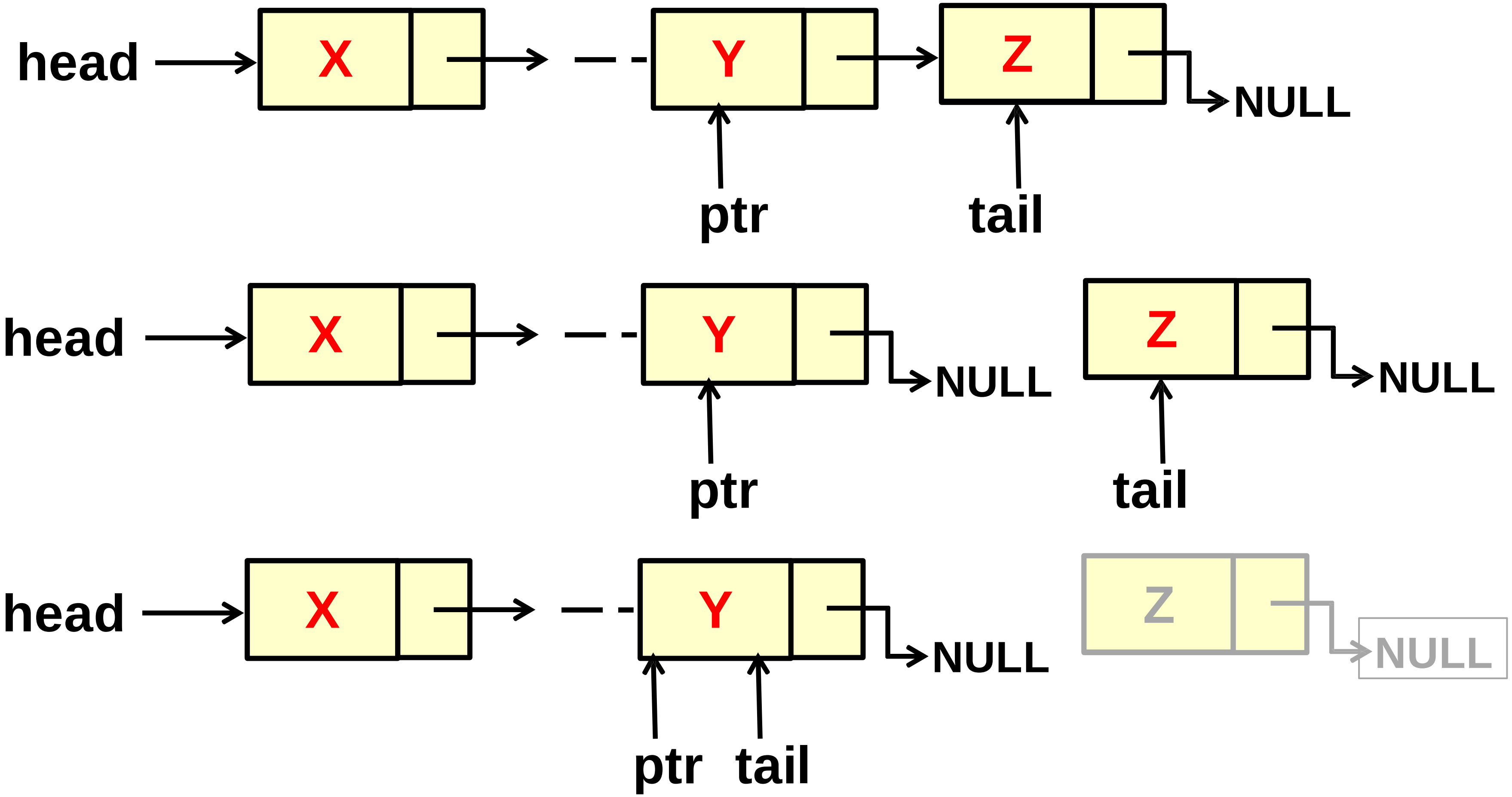
1. Bring ptr to the second last node
2. Make ptr->next equal to NULL



ptr->next = NULL;

Removing the Tail

1. Bring ptr to the second last node
2. Make ptr->next equal to NULL
3. Free tail
4. Make ptr the new tail

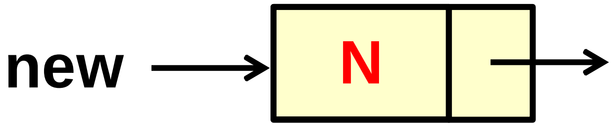


ptr->next = NULL;

free(tail);
tail = ptr;

Insertion into an ordered list

Create a *new* node containing the data

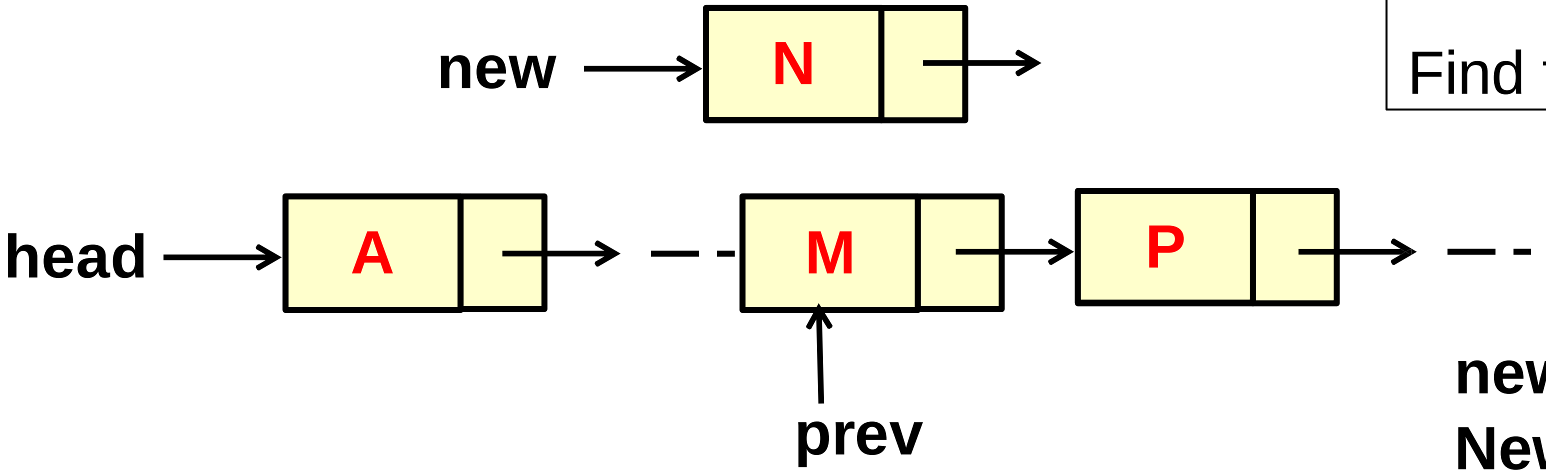


new = malloc(sizeof(ELEMENT));
New->d = 'N';

Insertion into an ordered list

Create a *new* node containing the data

Find the correct place in the list



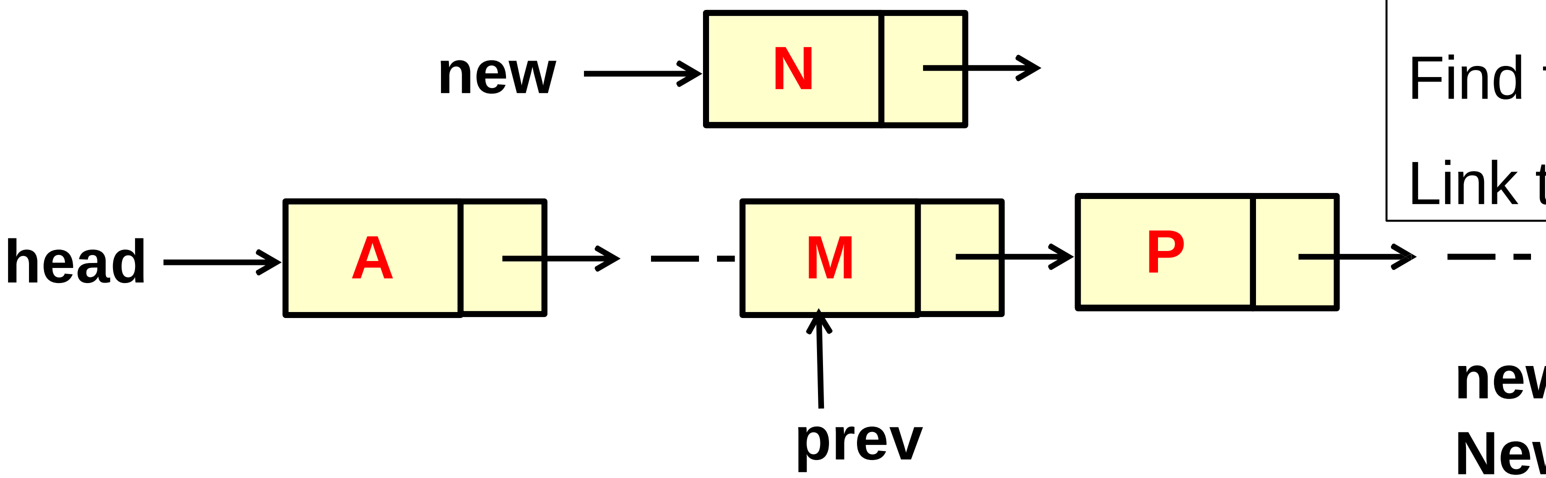
new = malloc(sizeof(ELEMENT));
New->d = ‘N’;

Insertion into an ordered list

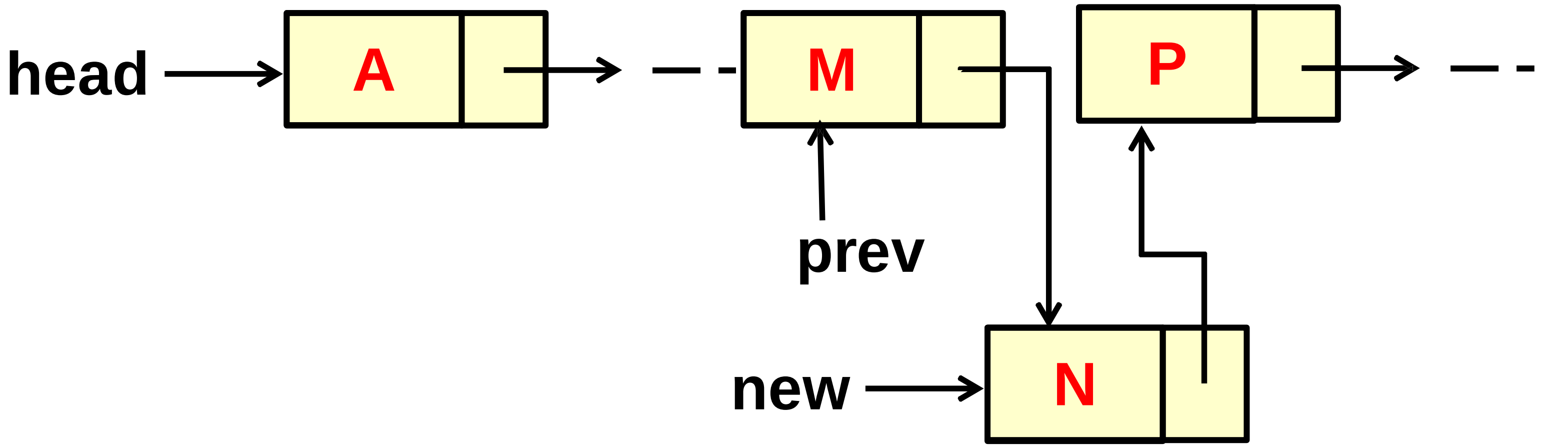
Create a *new* node containing the data

Find the correct place in the list, and

Link the *new* node at this place.




```
new = malloc(sizeof(ELEMENT));
New->d = 'N';
```



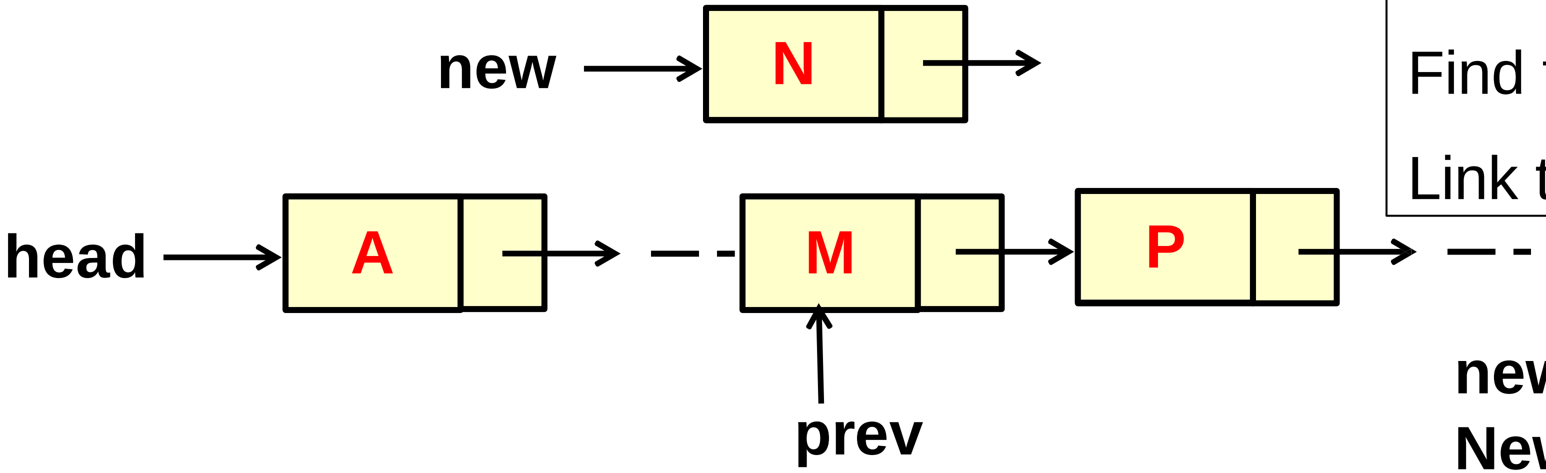

```
new->next = prev->next;
prev->next = new;
```

Insertion into an ordered list

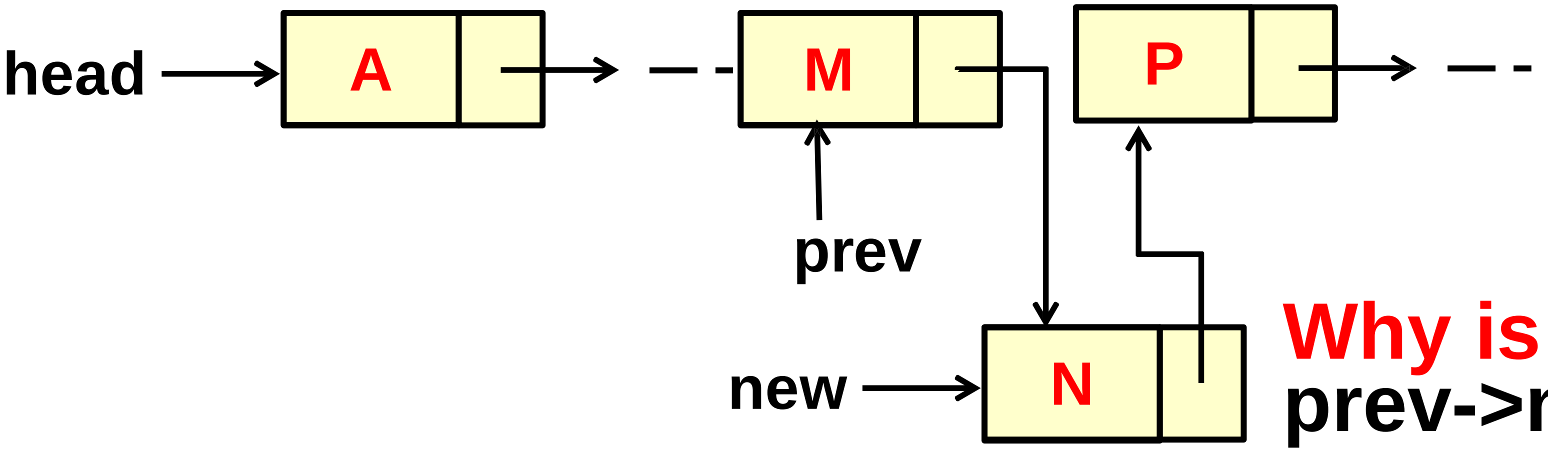
Create a *new* node containing the data

Find the correct place in the list, and

Link the *new* node at this place.



new = malloc(sizeof(ELEMENT));
New->d = 'N';



new->next = prev->next;
prev->next = new;

Why is the following not okay?
prev->next = new;
new->next = prev->next;

Insertion Function

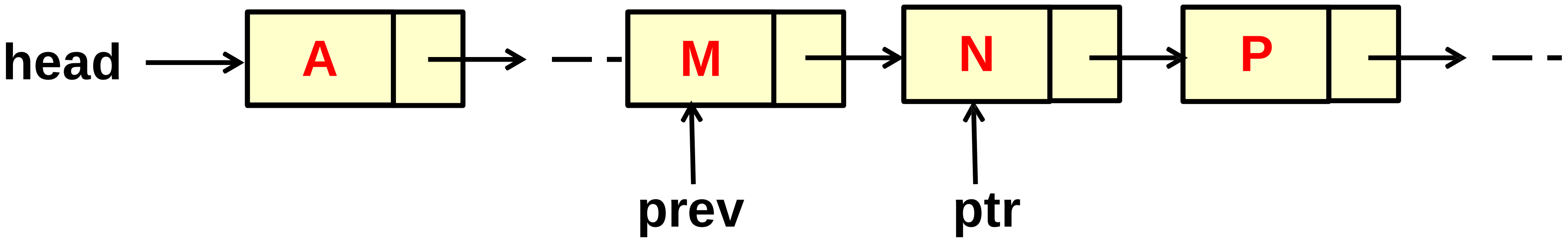
```
#include <stdio.h>
#include <stdlib.h>
struct list {
     int data;
     struct list *next;
};
typedef struct list ELEMENT;
typedef ELEMENT * LINK;

LINK create_node(int val) {
     LINK newp;
     newp = (LINK) malloc (sizeof
                 (ELEMENT));
     newp -> data = val;
     return (newp);
}
```

```
LINK insert (int value, LINK ptr)
{   LINK newp, prev, first;
    newp = create_node(value);
    if (ptr == NULL || value <= ptr -> data)
    {    // insert as new first node
         newp -> next = ptr;
         return newp;  // return pointer to first node
     }
     else {  // not first one
         first = ptr; // remember start
         prev = ptr;
         ptr = ptr-> next; // second
         while (ptr != NULL && value > ptr -> data)
         { prev = ptr; ptr = ptr -> next; }
         prev -> next = newp;     // link in
         newp -> next = ptr;   //new node
         return first;
     }
}
```

Deletion

Steps:

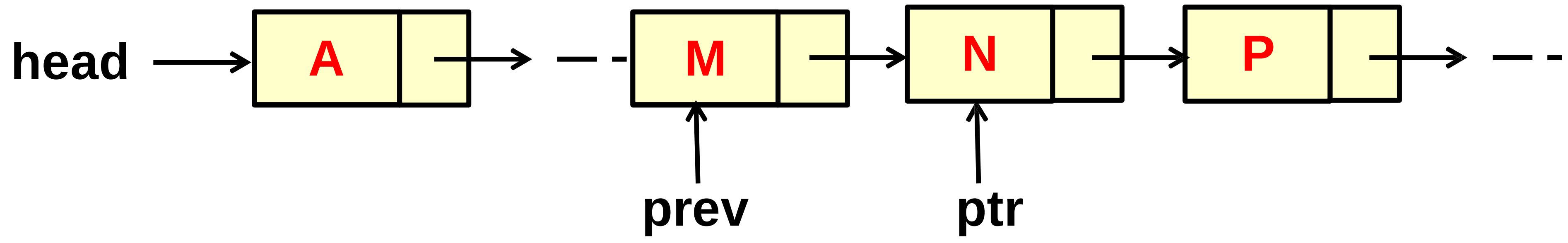
- Finding the data item in the list



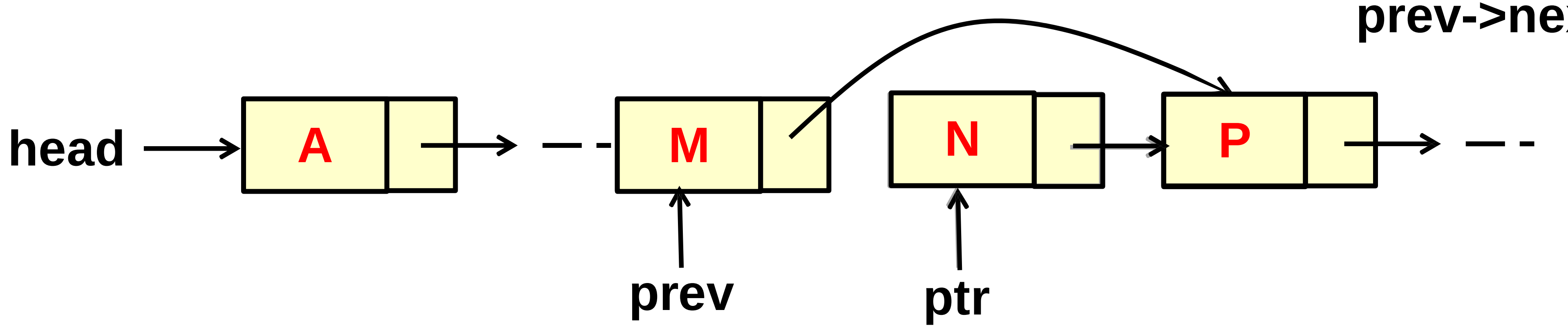
Deletion

Steps:

- Finding the data item in the list, and
- Linking out this node

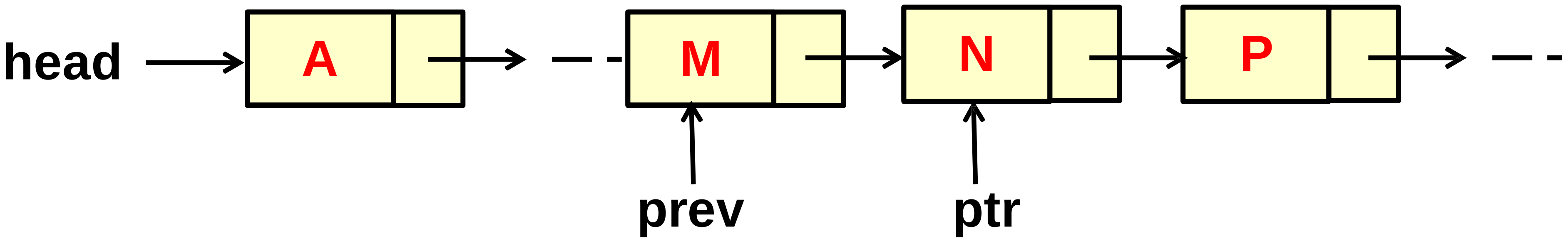


prev->next = ptr->next;

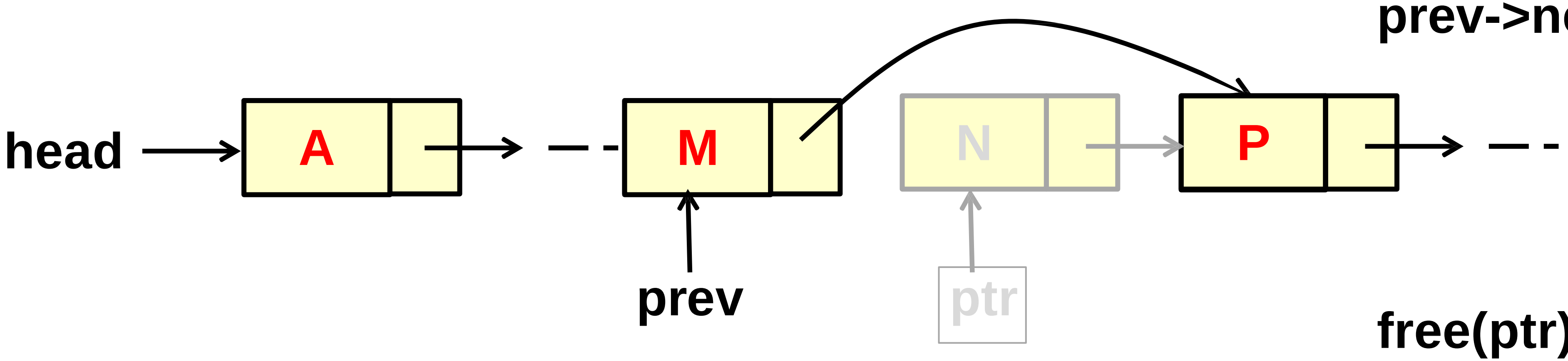


Deletion

Steps:
- Finding the data item in the list, and
- Linking out this node, and
- Freeing up this node as free space.



prev->next = ptr->next;

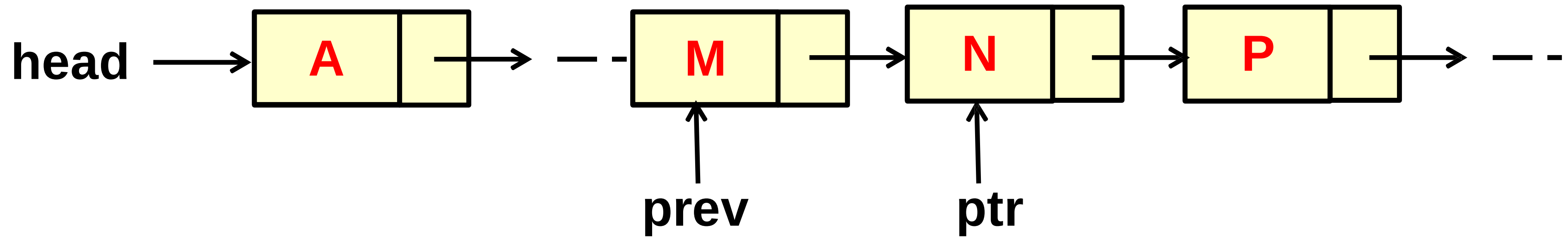


free(ptr);

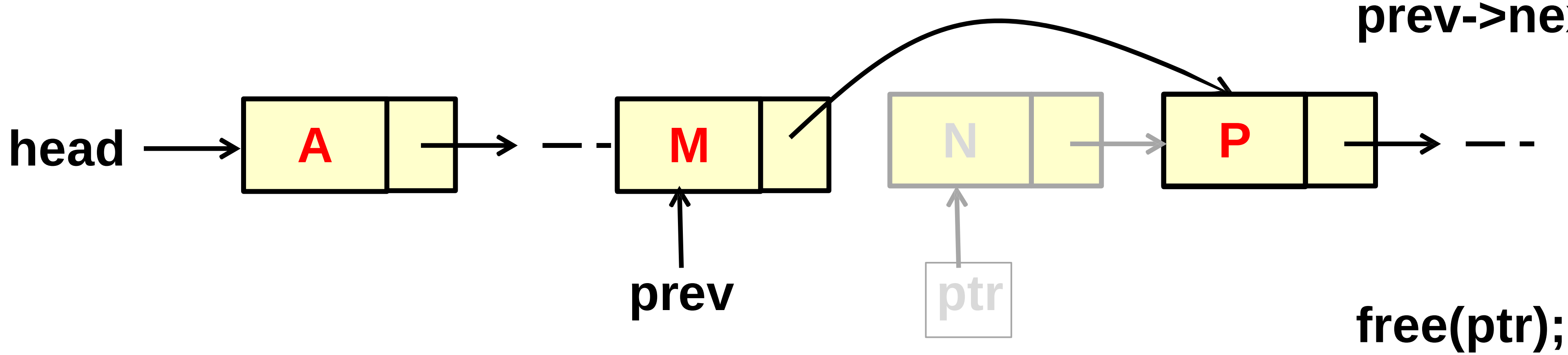
Deletion

Steps:

- Finding the data item in the list, and
- Linking out this node, and
- Freeing up this node as free space.



prev->next = ptr->next;



free(ptr);

What will happen if we did the following?

```
free(ptr);
prev->next = ptr->next;
```

Deletion Function

```
// delete the item from ascending list
LINK delete_item(int val, LINK ptr) {
    LINK prev, first;

    first = ptr; //remember start
    if (ptr == NULL) return NULL;
   else if (val == ptr -> data) //first node
   {
         ptr = ptr -> next; //second node
         first->next = NULL;
         free(first); //free up node
         return ptr; //second
   }
else    //check rest of list
{   prev = ptr;
     ptr = ptr -> next;
     // find node to delete
     while (ptr != NULL && val > ptr->data)
     { prev = ptr; ptr = ptr -> next; }
     if (ptr == NULL || val != ptr->data){
         // NOT found in ascending list
         return first;   //original
     }
     else {   //found, delete ptr node
         prev -> next = ptr -> next;
         ptr->next = NULL;
         free(ptr); //free node
         return first; //original
}}}
```

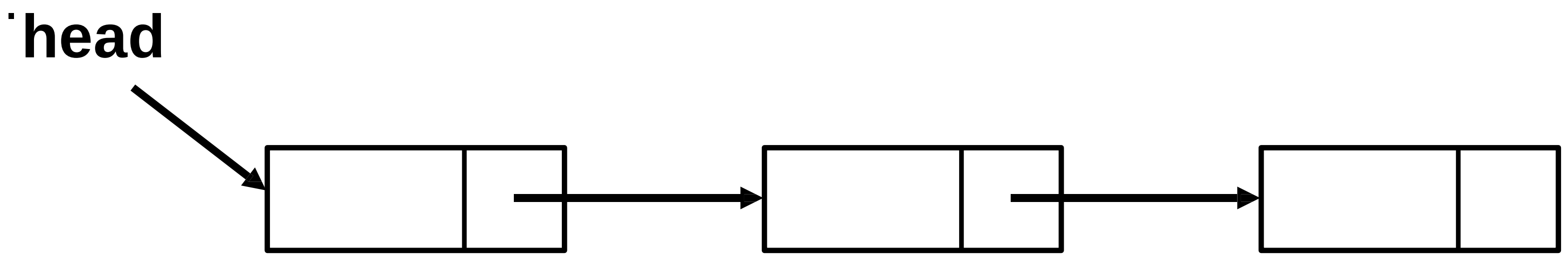
Linear Searching for a data element in a list

```
int Search( LINK head, int element) {
   LINK temp;

   temp = head;
   while (temp != NULL) {
         if (temp -> data == element) return 1;
         temp = temp -> next;
   }
   return 0;
}
```

Printing a list

```
void print_list (LINK head) {
    LINK temp;
    temp = head;
    while(temp!=NULL) {
        if(temp->next ==NULL)
            printf("%d. END OF LIST \n", temp->data);
        else printf("%d -> ", temp->data);
        temp = temp->next;
    }
}
```

Printing a list backwards



- How can you print backwards when the links are in forward direction?
- Can you apply recursion?

Printing a list backwards – *recursively*

```
void PrintArray(LINK head) {
    if(head -> next == NULL) {    //boundary condition to stop recursion
        printf(" %d, ",head -> data);
        return;
    }
    PrintArray(head -> next);         //calling function recursively
    printf(" %d,",head -> data);      //Printing current element
    return;
}
```

Freeing a list

- What will happen if we free the first node of the list without placing a pointer on the second?

Freeing a list

- What will happen if we free the first node of the list without placing a pointer on the second?
- In each iteration temp1 points at the head of the list and temp2 points at the second node.

```
void Free(ELEMENT *head) {
        ELEMENT *temp1, *temp2;
        temp1 = head;
        while(temp1 != NULL) {
                temp2 = temp1 -> next;
                temp1->next = NULL;
                free(temp1);
                temp1 = temp2;
        }
}
```

Counting the number of nodes in a list

RECURSIVE APPROACH

```
int count (LINK head) {
    if (head == NULL) return 0;
    return 1+count(head->next);
}
```

ITERATIVE APPROACH

```
int count (LINK head) {
    int cnt = 0;
    for ( ; head != NULL; head=head->next)
        ++cnt;
        return cnt;
}
```

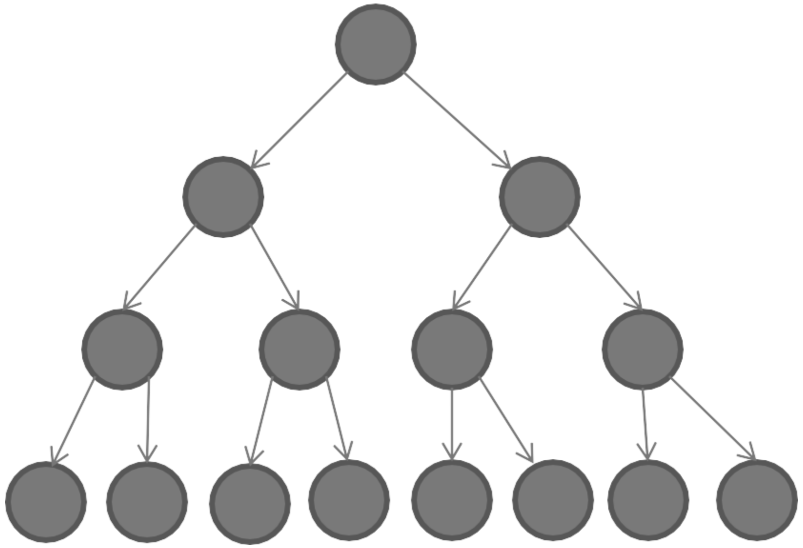
Concatenate two Lists

```
void concatenate  (LINK ahead, LINK bhead) {
        if (ahead->next == NULL)
              ahead->next = bhead ;
        else
              concatenate (ahead->next, bhead);
}
```

… And "Other" linked structures

- Like Trees, Sparse Matrices and Graphs

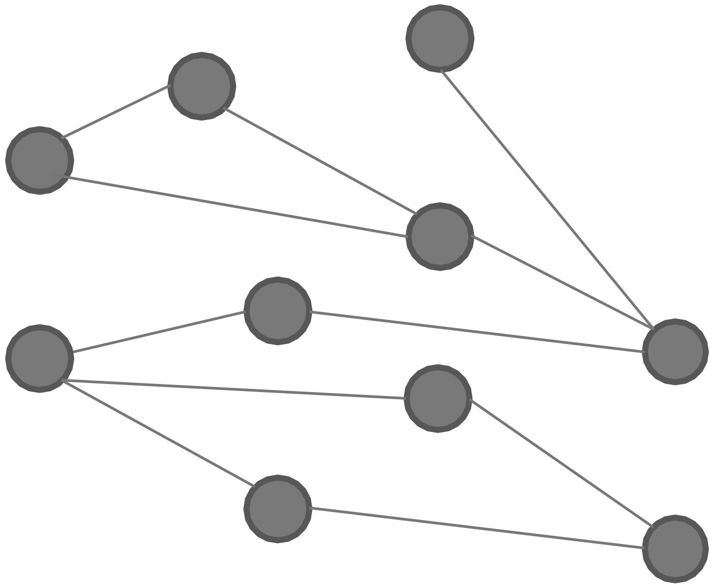
Binary Tree
(Height = 3)



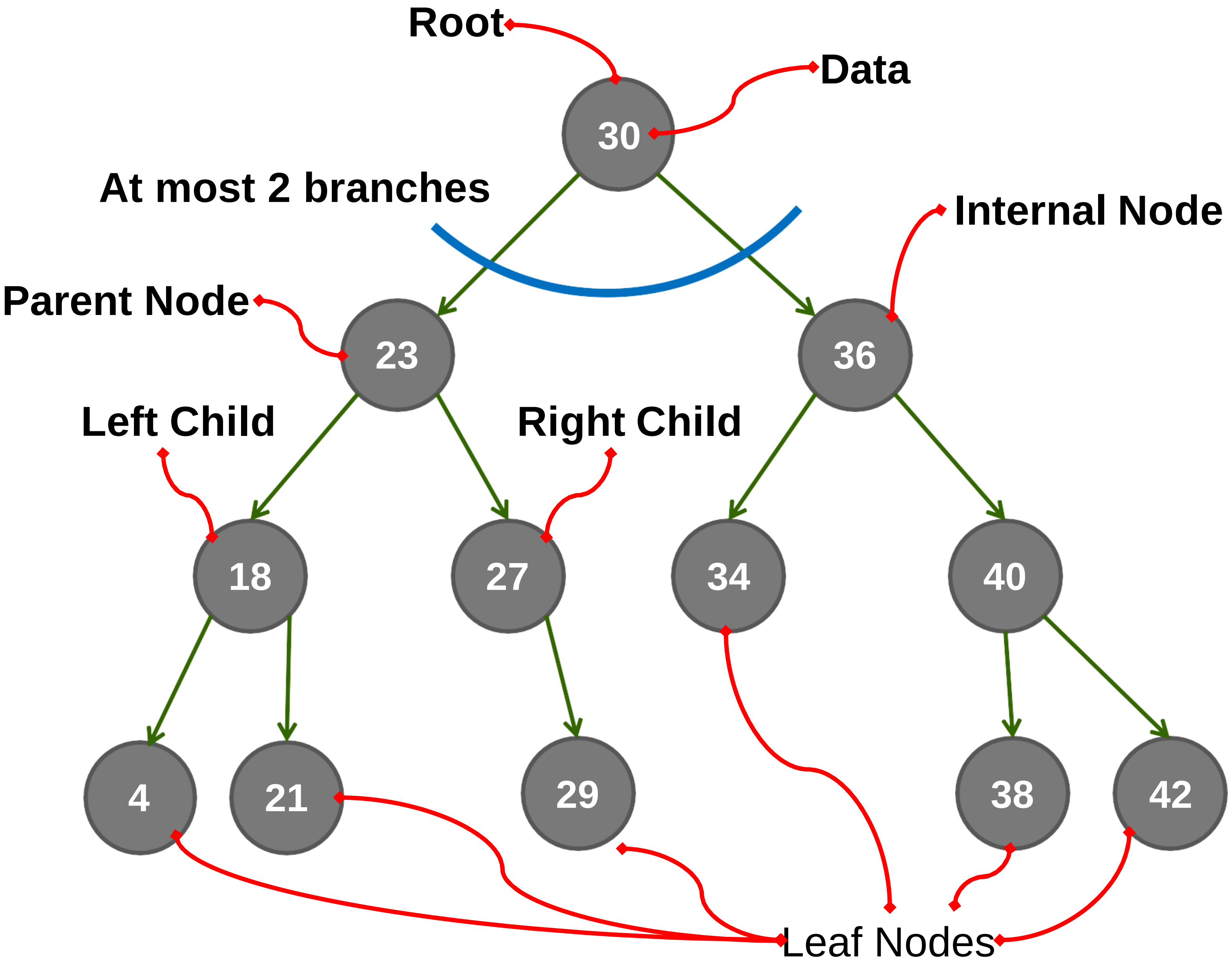
Sparse Matrix

$$\begin{bmatrix} 0 & 9 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 5 \\ 6 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 3 & 0 \end{bmatrix}$$

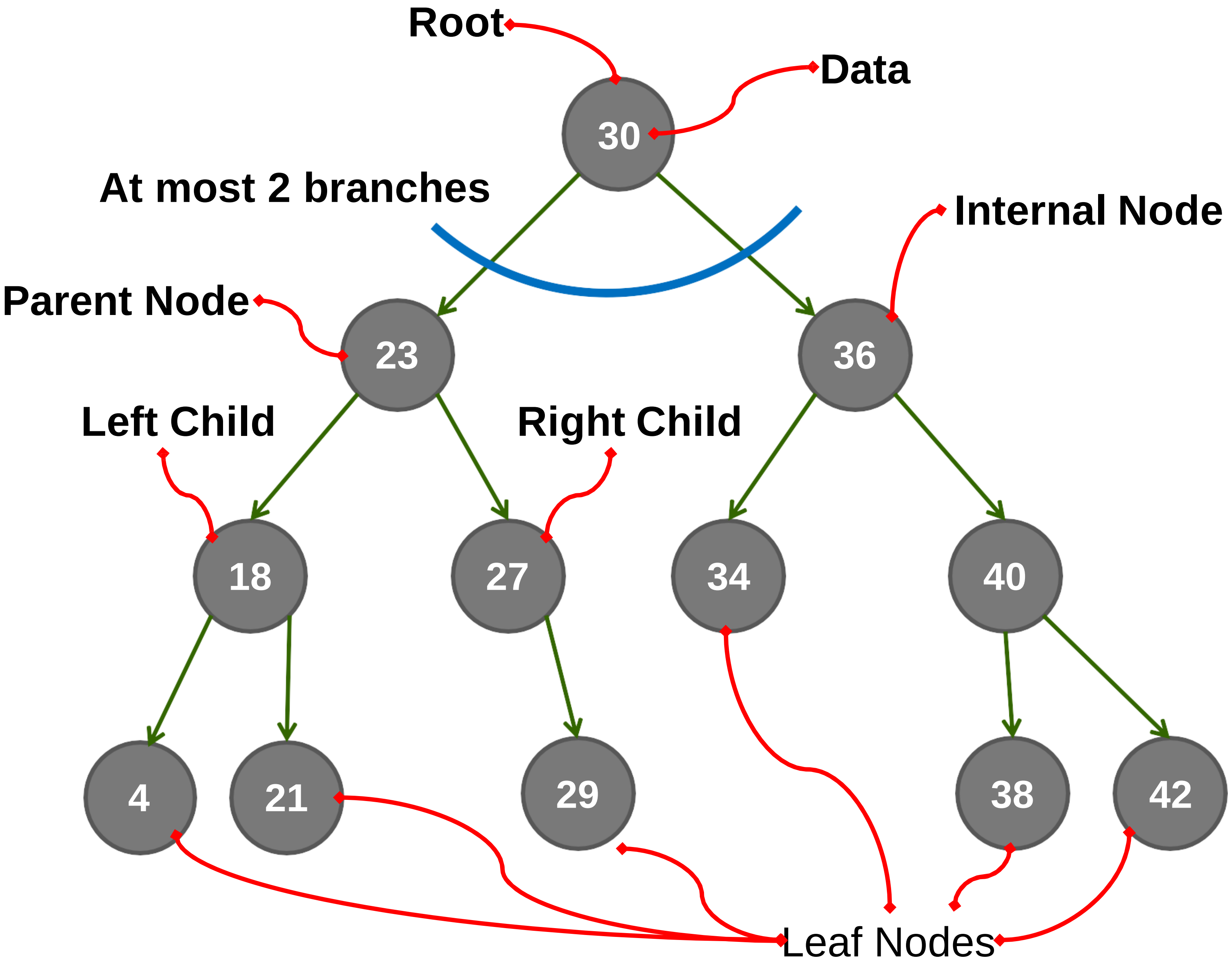
Graph
(Number of Vertices = 10)
(Number of Edges = 11)



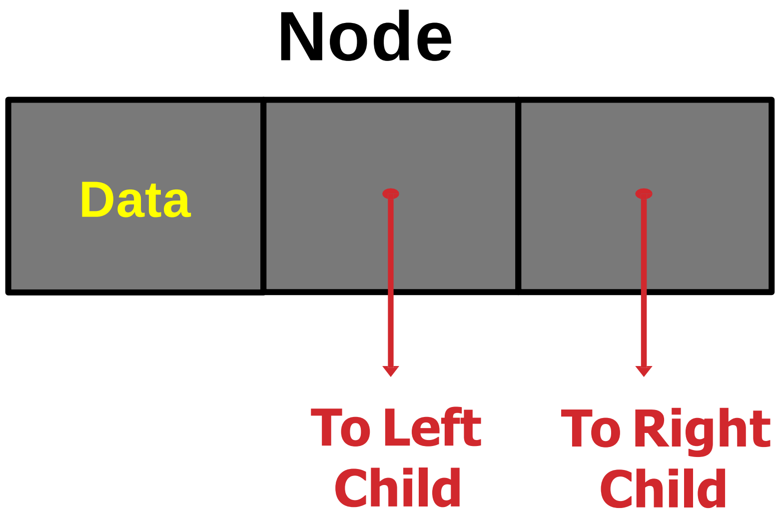
… And "Other" linked structures: Binary Trees



… And "Other" linked structures: Binary Trees




```
struct node {
    int data;
    struct node* left;
    struct node* right;
}
```



… And "Other" linked structures: Sparse matrices

For the sparse matrix below:

$$\begin{bmatrix} 0 & 9 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 5 \\ 6 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 3 & 0 \end{bmatrix}$$

Storage as a 2-D Array:

int M[5][6];

Storage required for 30 elements (with only 4 non-zero entries)

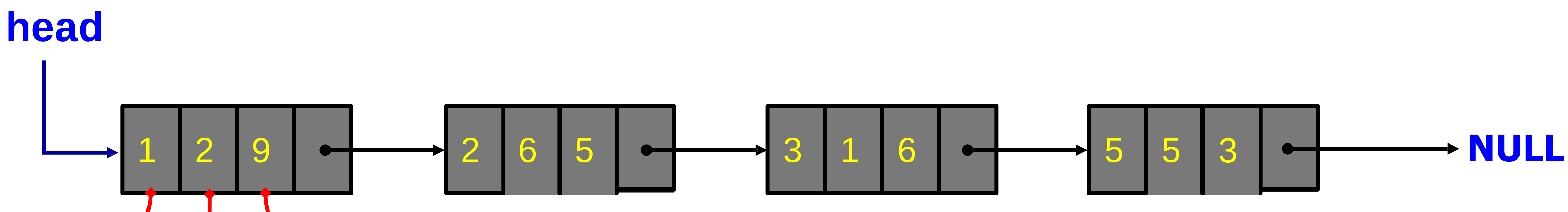
= 30 * sizeof(int) = 120Bytes (For integers of size 4 Bytes)

… And "Other" linked structures: Sparse matrices

Storage as a list of Tripples: (row, column, data)

$$\begin{bmatrix} 0 & 9 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 5 \\ 6 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 3 & 0 \end{bmatrix}$$

```
struct tripple {
   int row, column, data;
   struct tripple *next;
}
```



Row Column Data

Storage required for 4 entries = (3 × sizeof(int) + sizeof(struct tripple*) × 4)
= (3 × 4 + 8) × 4
= 80 Bytes < 120 Bytes

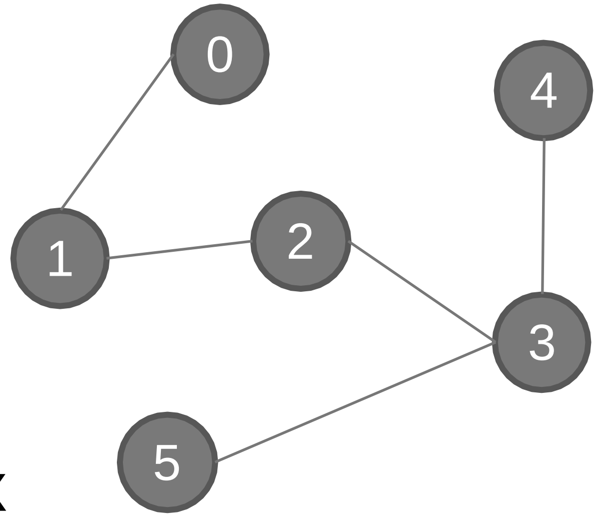
… And "Other" linked structures: Graphs

<u>Adjacency Matrix Representation:</u>

- Matrix location (i , j) indicates an edge between vertices "i" and "j"

$$\begin{bmatrix} 0 & 1 & 0 & 0 & 0 & 0 \\ 1 & 0 & 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 & 1 & 1 \\ 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 \end{bmatrix}$$

Adjacency Matrix for an undirected graph



Storage as a 2-D Array:

int G[6][6];

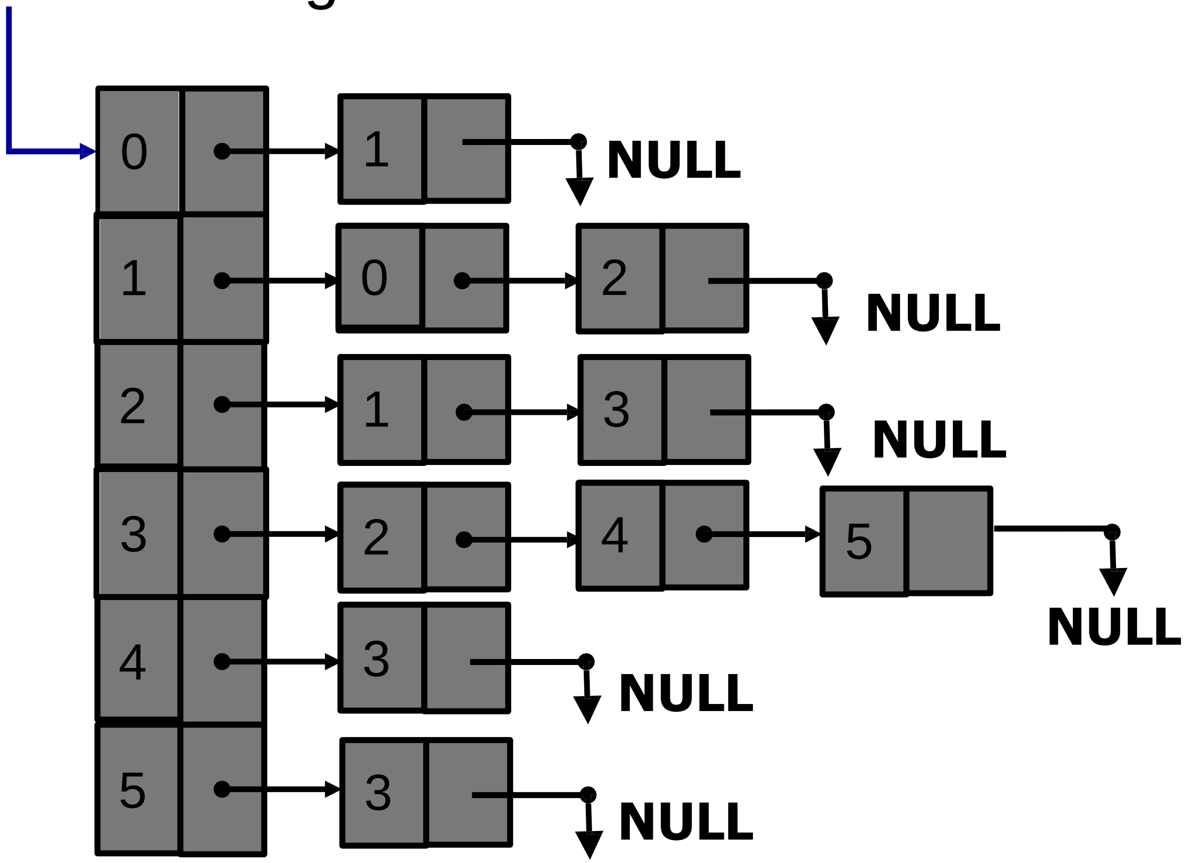
Storage required for 36 elements (with 6 vertices 5 edges)

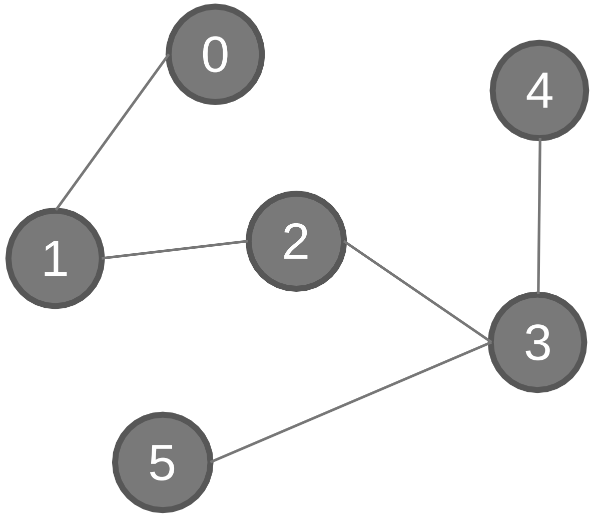
= 36 * sizeof(int) = 144Bytes (For integers of size 4 Bytes)

… And "Other" linked structures: Graphs

Adjacency List Representation:

- Each vertex's neighbours are maintained in a linked list






```
struct vertex{
    int id;
    struct vertex *next_adj;
}
```

Storage required = (| V | + sum of degree) * sizeof(structure)

Practice Problems

1. Concatenate two lists (iteratively)
2. Reverse a list
3. Delete the maximum element from a list
4. Rotate the list by k positions counter-clockwise
5. Write functions to create, insert, delete, display, search a sparse matrix

For each of the above, first create the linked list by reading in integers from the keyboard and inserting one by one to an empty list

Acknowledgement

- IIT Kharagpur

Thank You