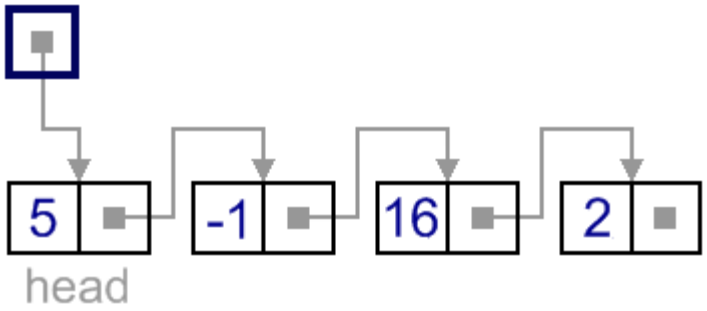
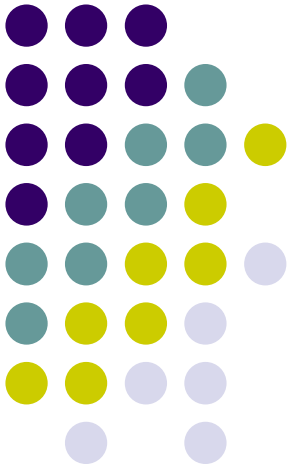


Linked Lists

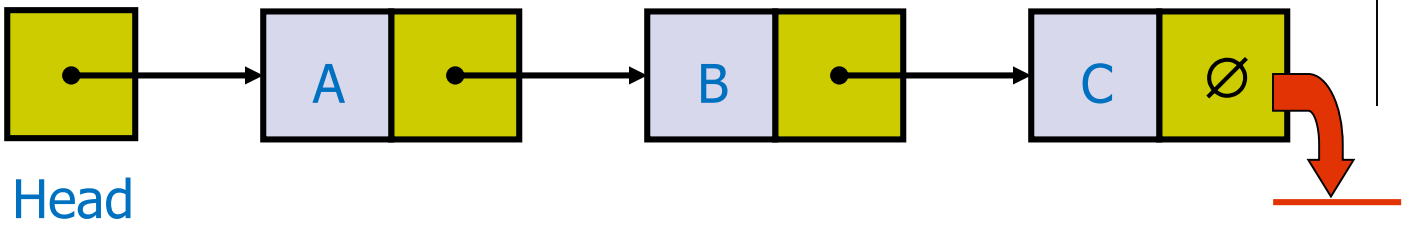


Dr. Gurpreet Singh Lehal,
Department of Computer Science,
Punjabi University

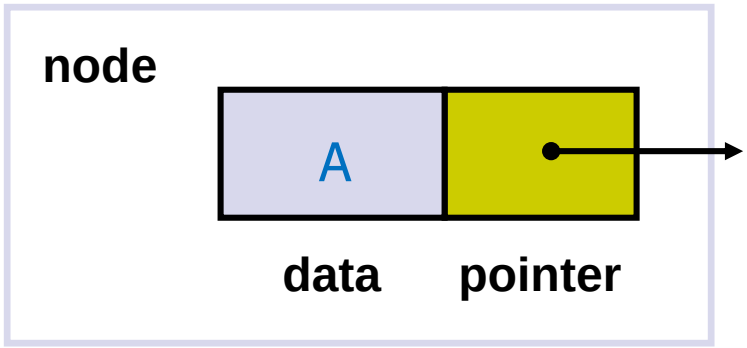
Linked Lists

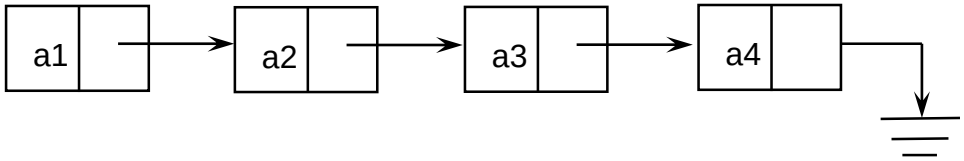
- A linear collection of self-referential objects, called nodes, connected by links
  - linear: for every node in the list, there is one and only one node that precedes it (except for possibly the first node, which may have no predecessor,) and there is one and only one node that succeeds it, (except for possibly the last node, which may have no successor)
  - self-referential: a node that has the ability to refer to another node of the same type, or even to refer to itself
  - node: contains data of any type, including a reference to another node of the same data type, or to nodes of different data types
  - Usually a list will have a beginning and an end; the first element in the list is accessed by a reference to that class, and the last node in the list will have a reference that is set to `null`

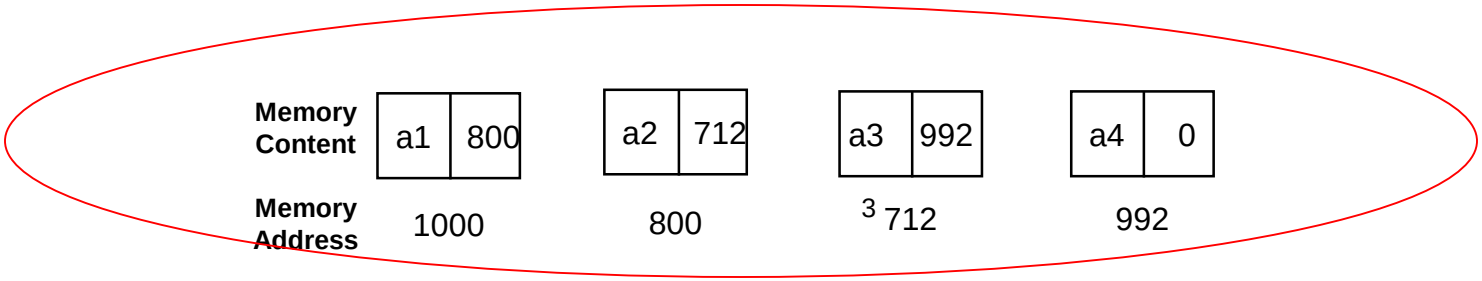
Linked Lists



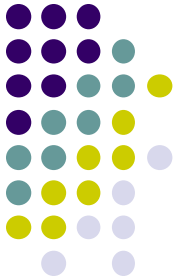
- A ***linked list*** is a linear collection of connected ***nodes***
- Each node contains at least
  - **A piece of data (any type)**
  - **Pointer to the next node in the list**
- *Head*: pointer to the first node
- The last node points to **`NULL`**





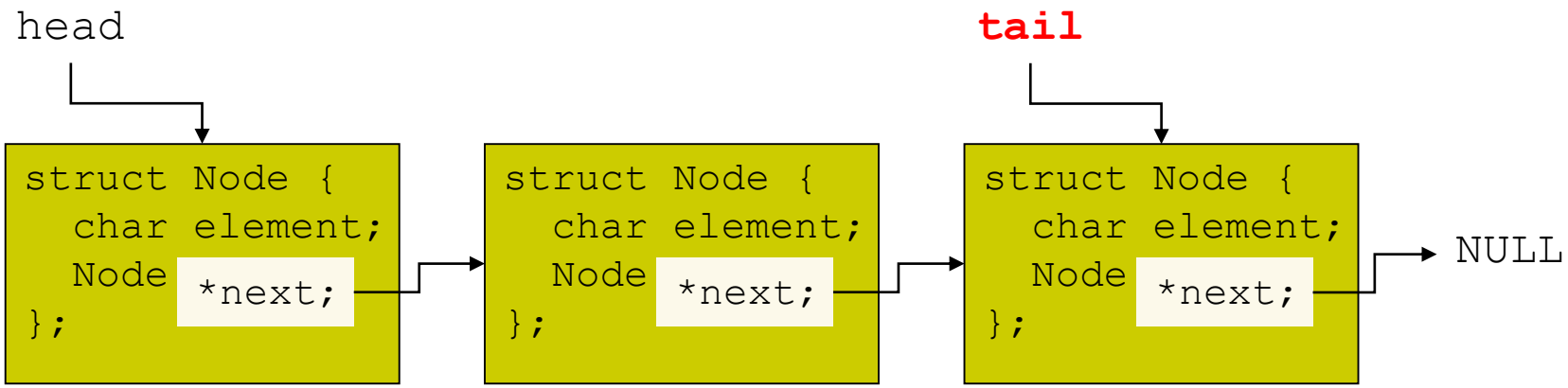


Composition of a Linked List

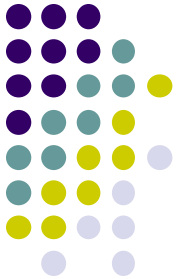


A linked list is called "**linked**" because each node in the series (i.e. the chain) has a **pointer** to the **next node** in the list, e.g.

a) The list head is a pointer to the *first node* in the list.
b) Each node in the list points to the next node in the list.
c) The last node points to NULL (the usual way to signify the end).

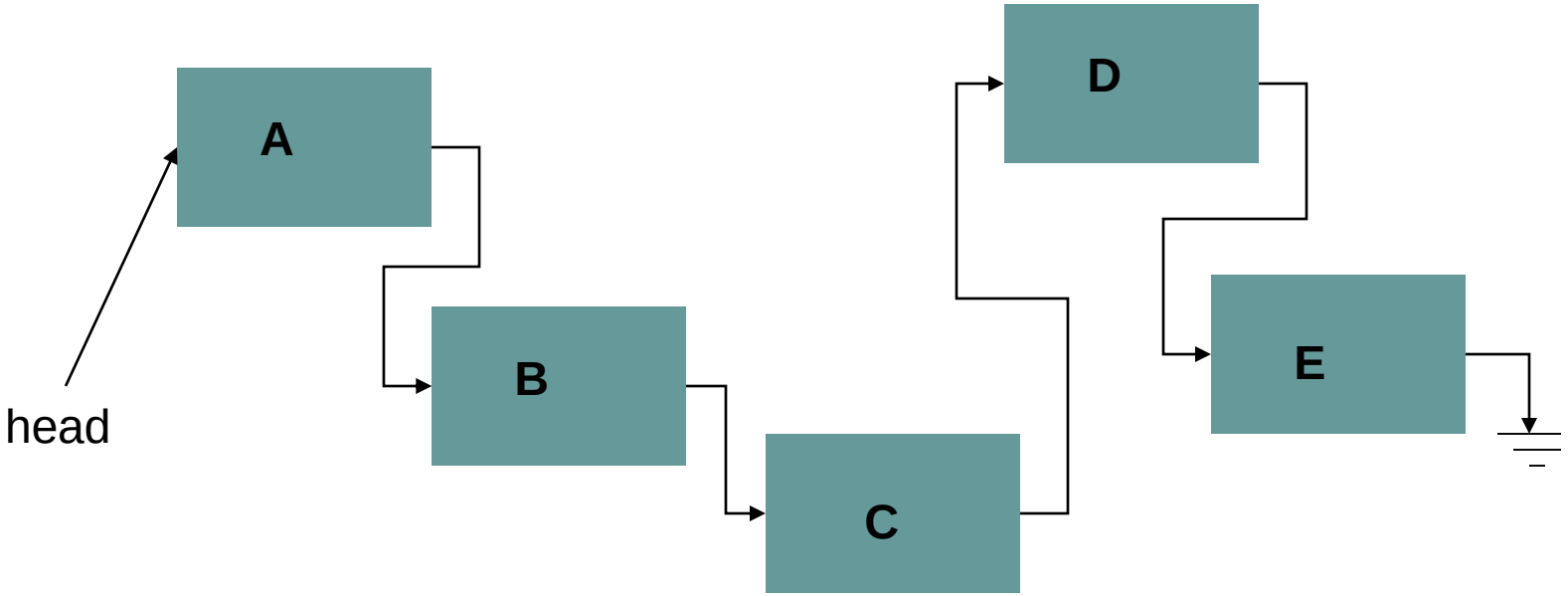


Composition of a Linked List



The pointers play a big role in maintaining the linked list. The nodes in a linked list can be **spread out** over memory. Therefore, it is not possible to **calculate** the address of the next node, like it is possible to calculate the next element of an array. If a pointer from one node to another is lost, the list from that point is lost forever. Therefore, extreme care should be taken when assigning or re-assigning a pointer in a linked list.

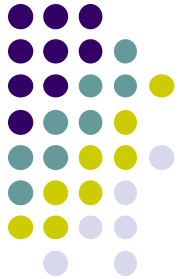
Composition of a Linked List



If the pointer between any nodes (say B and C) is re-assigned erroneously to null or anything else, the access to the rest of the nodes (C, D and E) is then lost.

If you loose the head pointer, you loose the entire list.

Advantages of Linked Lists over Arrays

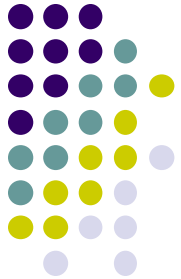


Linked lists are more complex to code and manage than arrays, but they have some distinct advantages.

a) **A linked list can easily grow and shrink in size -** The programmer doesn't need to know how many nodes will be in the list. They are created in memory as needed.

b) **Speed of insertion or deletion from the list -** Inserting and deleting elements into and out of arrays requires moving elements of the array. When a node is inserted, or deleted from a linked list, none of the other nodes have to be moved.

Creating a Linked List

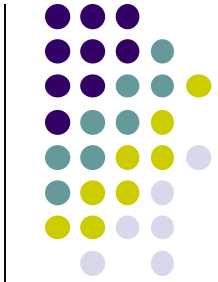


Just like any other data type, the information about the node has to be first be declared.

Step 1) Declare a data structure for the nodes.

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

    Typedef struct Node *NODEPTR;
```

Creating a Linked List



a) In this example, the first member of the Node holds the node's **data**. This could just as well be just a string, or a structure of student records.

b) The second member is a **pointer** called next. It holds the **address** of any object that is a structure of type Node. Hence each Node struct can point to the next node in the list.

The Node struct contains a pointer to an object of the **same type** as that being declared. It is called a ***self-referential data structure***. This makes it possible to create nodes that point to other nodes of the same type.

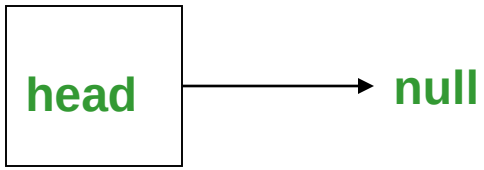
Creating a Linked List

Next, since there is no physical relationship between nodes, a pointer needs to be created to point to the first logical node in the list.

Step 2) Declare a *pointer* to serve as the **head** of the list:

```
NODEPTR head = NULL;
```

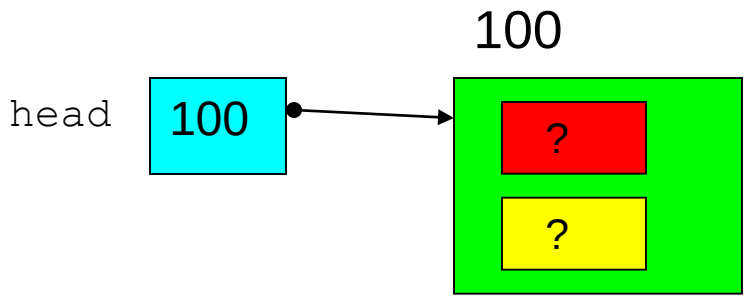
Once we have done these 2 steps (i.e. declared a node data structure, and created a NULL head pointer, we have an empty linked list.



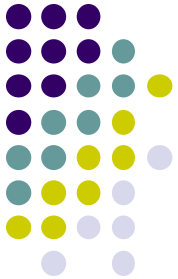
Creating a Linked List

- **Step 3)** Allocate memory for the node

```
head = (NODEPTR) /*type casting */
       malloc(sizeof(struct node));
```



Creating a Linked List

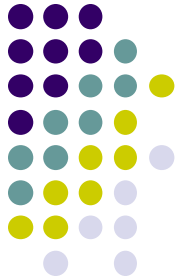


- Note : head is not a node. It is a simple pointer to a node. head used to "maintain" start of list
- To assign values to data items in first node we have following statement:

  (*head).data = 12;

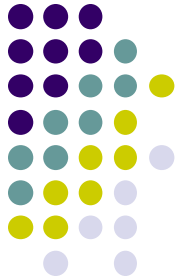
  - head is a pointer variable so *head is the node that head points to
  - The parentheses are necessary because the dot operator **.** has higher precedence than the dereference operator ***

Creating a Linked List

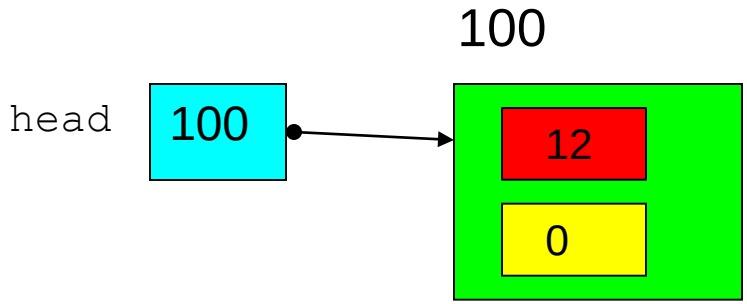


- The arrow operator -> combines the actions of the dereferencing operator * and the dot operator to specify a member of a struct or object pointed to by a pointer
  - **(*head).data= 12;**
    can be written as
    head->data= 12;
  - The arrow operator is more commonly used
- **head->next = NULL;**

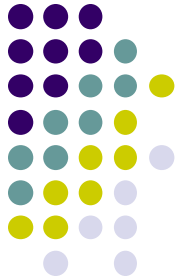
Creating a Linked List



- The Linked list containing a single node can be visualised as



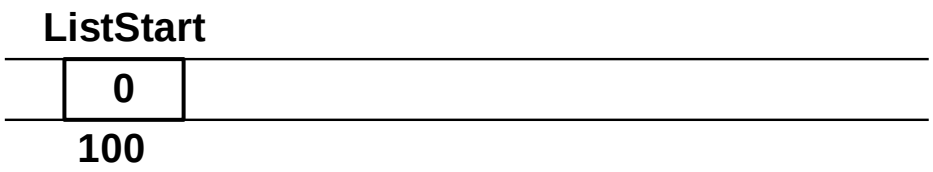
Sample Linked List Operations



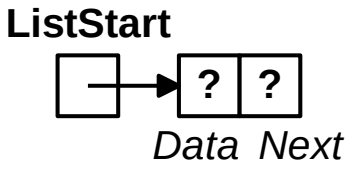
```
void main() {
  NODEPTR ListStart = NULL;
  /* safest to give ListStart an initial legal
     value -- NULL indicates empty list */
```

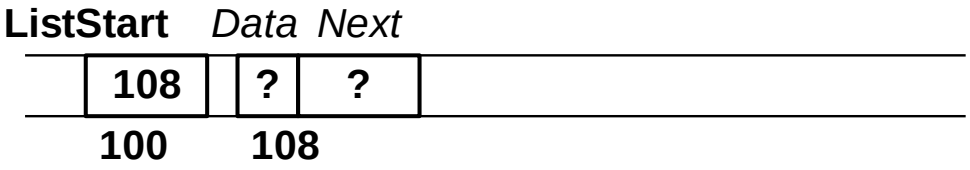
Pictorially

In Memory



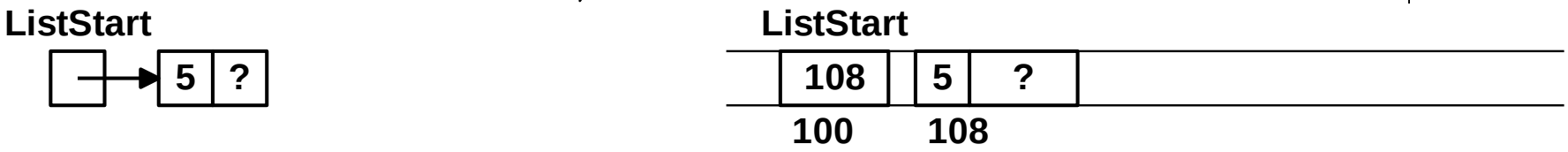

```
  ListStart = (NODEPTR) malloc(sizeof(struct node));
    /* ListStart points to memory allocated at
       location 108 */
```



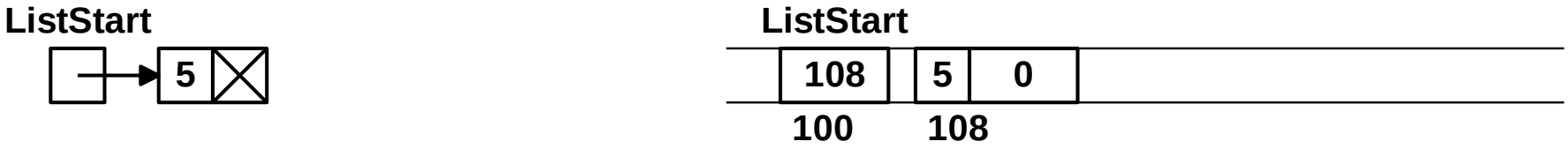


Sample Linked List Ops (cont)

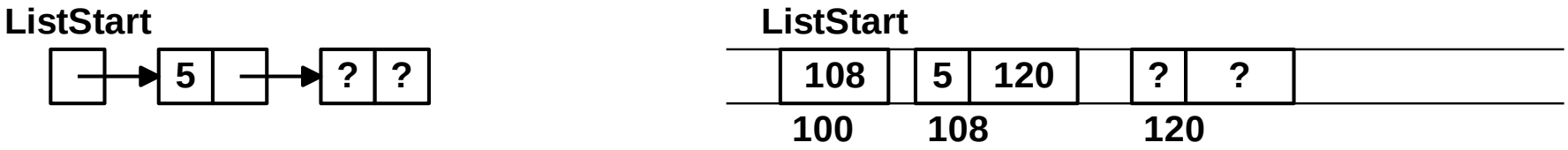
```
ListStart->data = 5;
```



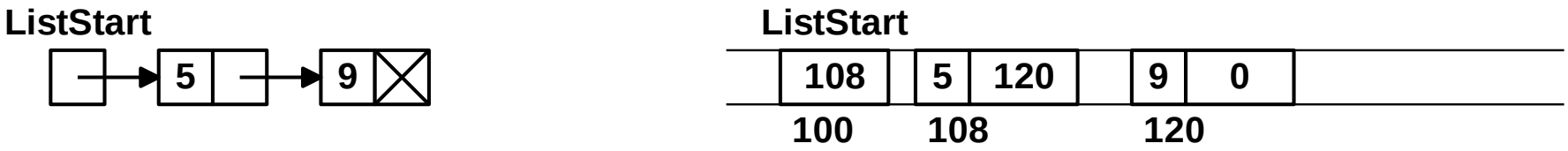

```
ListStart->next = NULL;
```



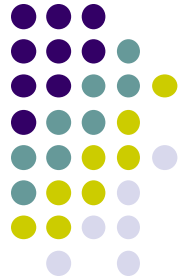

```
ListStart->next = (NODEPTR) malloc(sizeof(struct node));
```



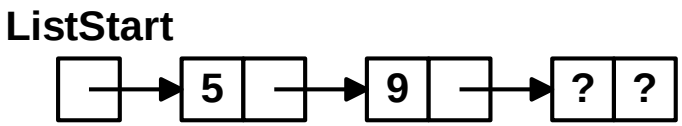

```
ListStart->next->data = 9;
ListStart->next->next = NULL;
```

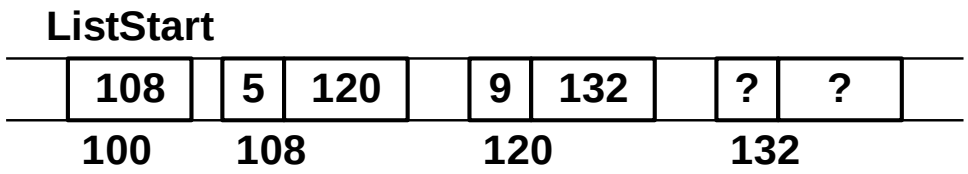


Sample Linked List Ops (cont)

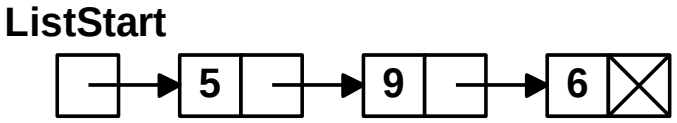


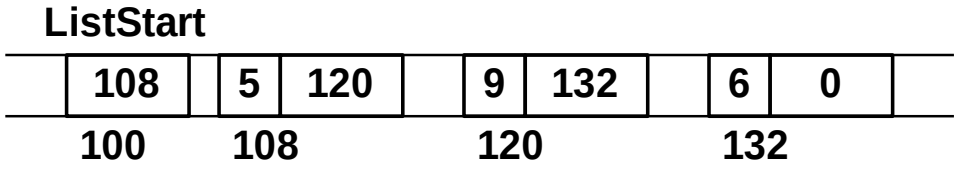
```
ListStart->next->next = (NODEPTR) malloc(sizeof(struct node));
```





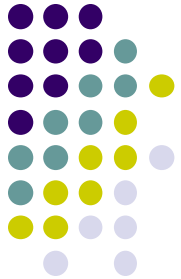

```
ListStart->next->next->data = 6;
ListStart->next->next->next = NULL;
```






```
/* Linked list of 3 elements (count data values):
   ListStart points to first element
   ListStart->next points to second element
   ListStart->next->next points to third element
   and ListStart->next->next->next is NULL to
     indicate there is no fourth element */
```

Basic Operations on a Linked List

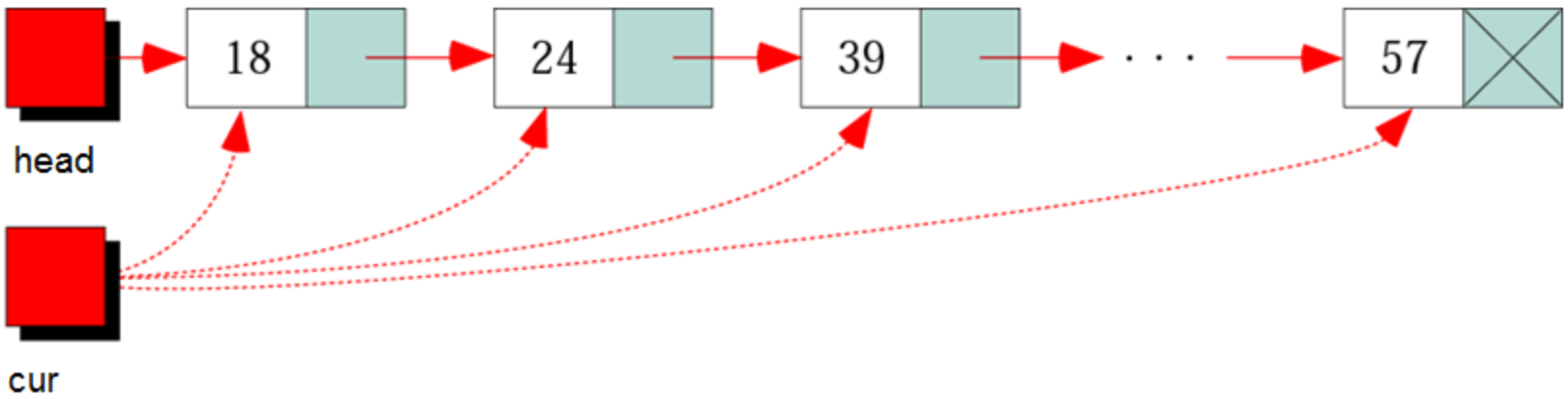


1. Traverse (walk) the list.
2. Add a node.
3. Delete a node.
4. Search for a node.

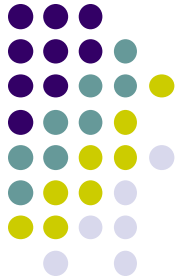
Displaying the contents of a linked list

- A traverse operation visits each node in the linked list
  - A pointer variable cur keeps track of the current node

```
for (Struct Node *cur = head;
      cur != NULL; cur = cur->next)
   printf("%d", cur->data);
```

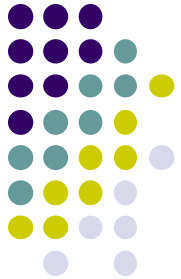


A few questions on linked lists



- Write a function to count the nodes of a linked list.
- `int get_count(NODEPTR head)`

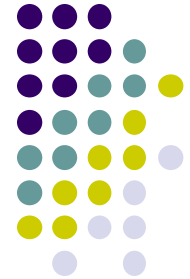
A few questions on linked lists



- Write a function to count the nodes of a linked list.
- `int get_count(NODEPTR head)`

```
int count_list(NODEPTR head)
{
    int count;
    for(count=0; head != NULL; head=head->next, count++);
    return(count);
}
```

A few questions on linked lists



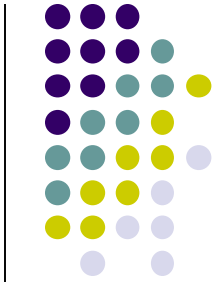
- Write a function to find the value stored in nth node of a linked list.
- `int get_nth(NODEPTR head, int n)`
- Where n=1 means first node

A few questions on linked lists

- Write a function to find the value stored in nth node of a linked list.
- `int get_nth(NODEPTR head, int n)`
- Where n=1 means first node

```
int get_nth(NODEPTR head, int n)
{
   int count;
   for(count=1; head != NULL && count!=n ; head=head->next, count++);
   if(head)
      return(head->data);
   else return(0);
}
```

A few questions on linked lists

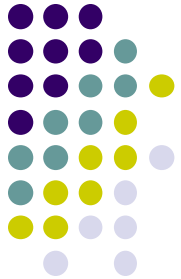


- Write a function to display the median value of an ordered linked list.
- `int get_median(NODEPTR head)`

A few questions on linked lists

- Write a function to find the value stored in nth node **from end** of a linked list.
- `int get_end_nth(NODEPTR head, int n)`
- Where n=0 means last node

A few questions on linked lists



- Write a function to display the median value of an ordered linked list.
- `int get_median(NODEPTR head)`

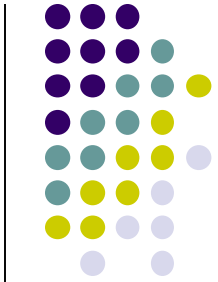
```
int get_median(NODEPTR head)
{
        int count = count_list(head);
        int m1, m2;
        if(count%2)
                return(get_nth(head, 1+count/2));
        m1 = get_nth(head, count/2);
        m2 = get_nth(head, 1+(count/2));
        return ((m1+m2)/2);
}
```

A few questions on linked lists

- Write a function to find the value stored in nth node **from end** of a linked list.

```
int get_end_nth(NODEPTR head, int n)
{
   int count;
   NODEPTR p1=head, p2=head;
   for(count=0; p1 != NULL && count!=n ; p1=p1->next, count++);
      while(p1 && p1->next)
      {
            p1=p1->next;
            p2=p2->next;
      }
      if(p2)
            return(p2->data);
      else return(0);
}
```

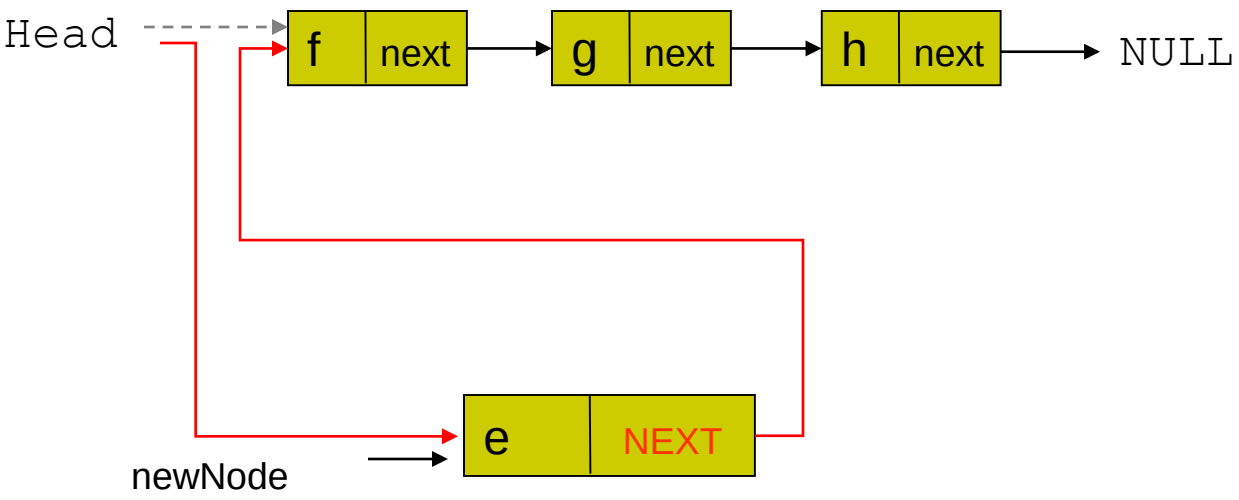
Adding Nodes to a Linked List



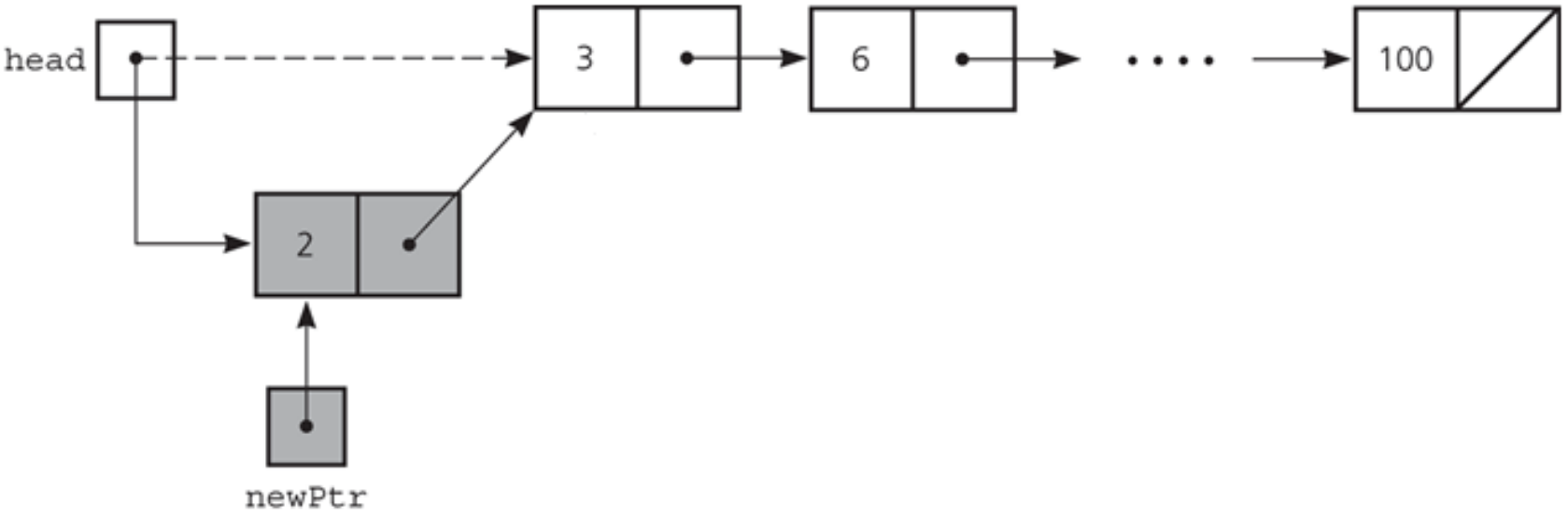
There are four steps to add a node to a linked list:

- Allocate memory for the new node.
- Determine the insertion point you need to know only the new node's predecessor (`pPre`)
- Point the new node to successor of pPre.
- Point pPre to the new node.

Inserting a Node at the Front



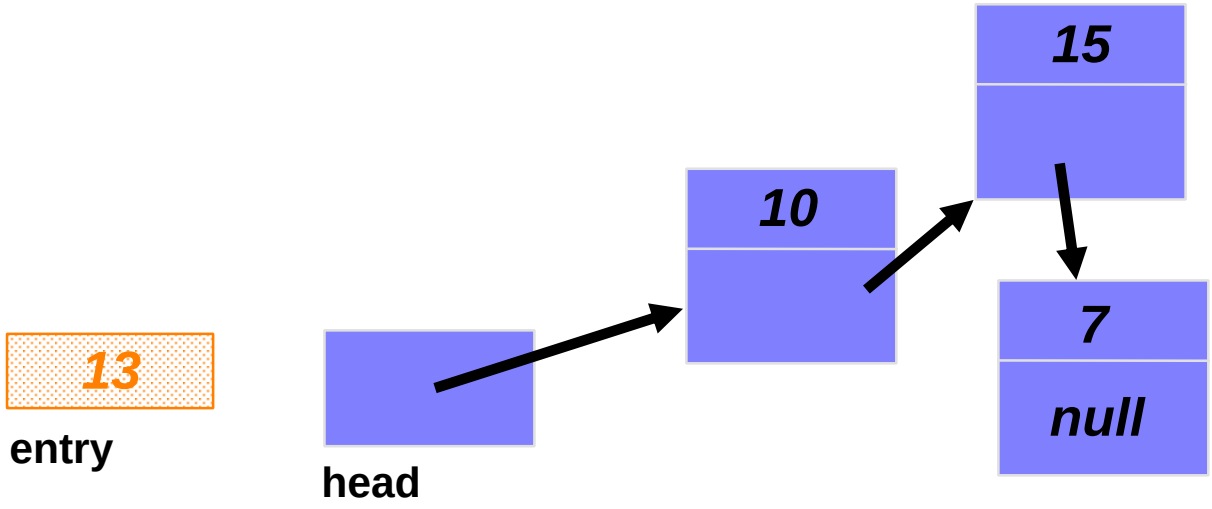
Inserting a Node at the Front



Inserting a Node at the Front

```
void insert_start(NODEPTR *head, int x);
```

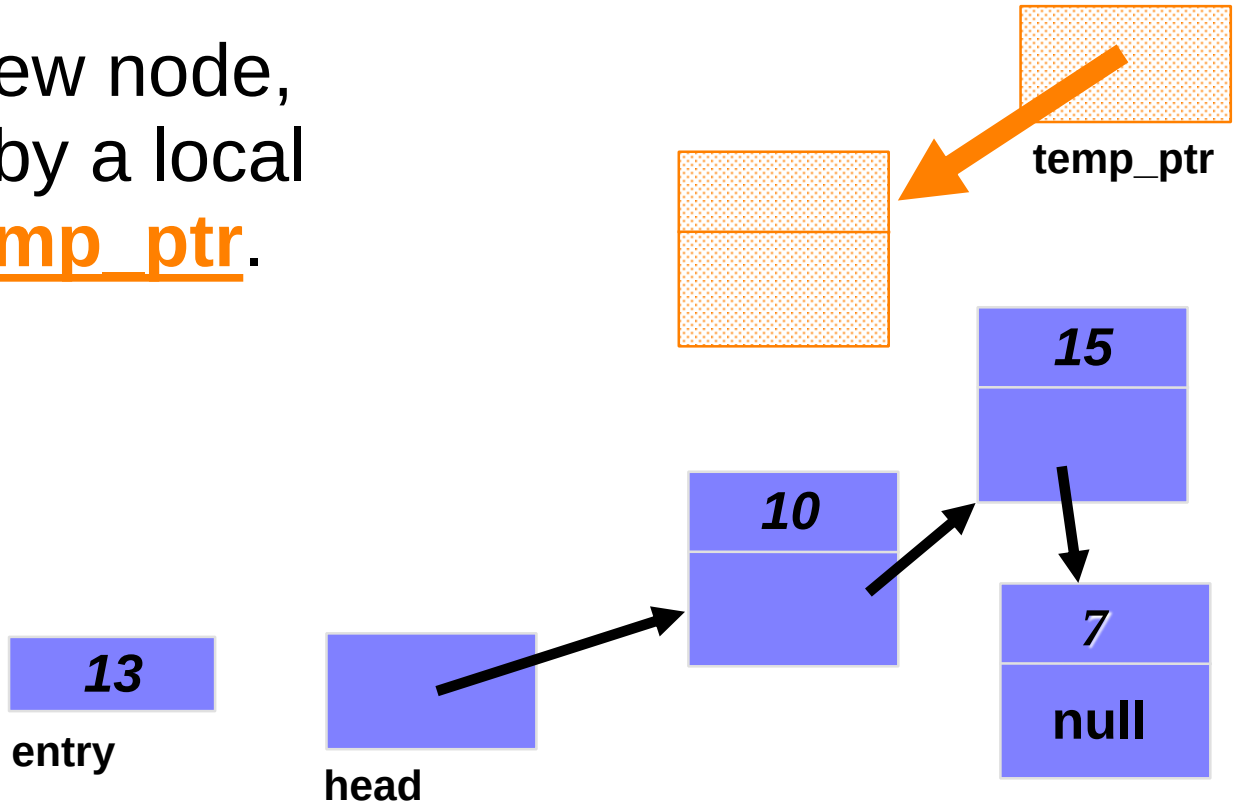
We want to add a new entry, 13, to the **front** of the linked list shown here.



Inserting a Node at the Front

```
void insert_start(NODEPTR *head, int x);
```

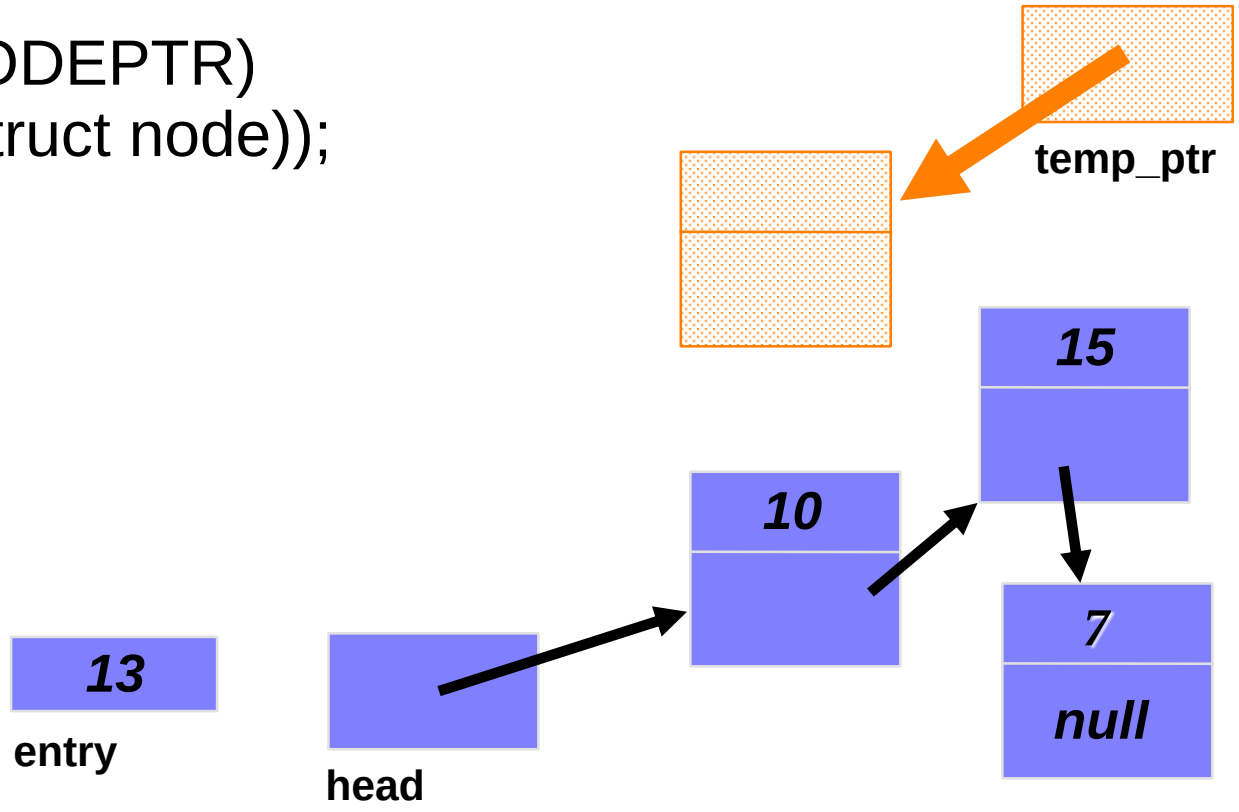
- Create a new node, pointed to by a local variable **temp_ptr**.



Inserting a Node at the Front

```
void insert_start(NODEPTR *head, int x);
```

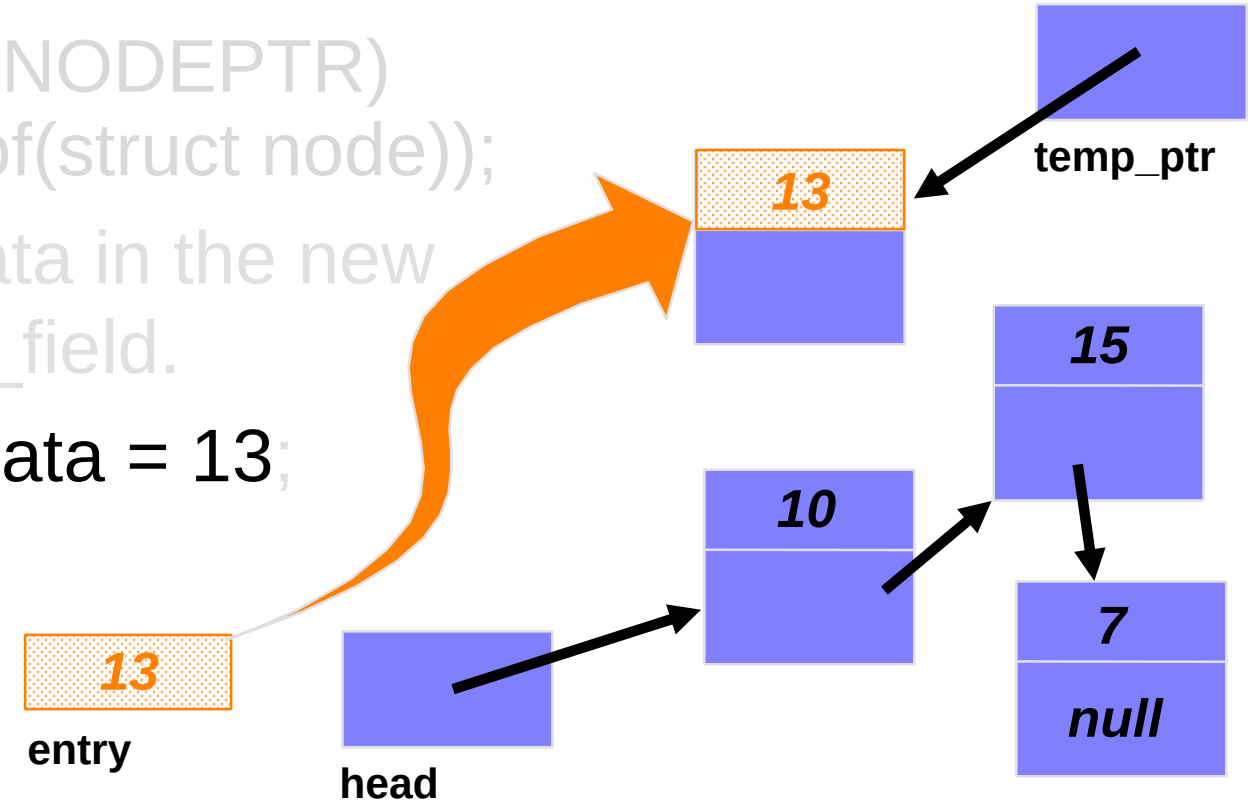
- temp_ptr = (NODEPTR) malloc(sizeof(struct node));



Inserting a Node at the Front

```
void insert_start(NODEPTR *head, int x);
```

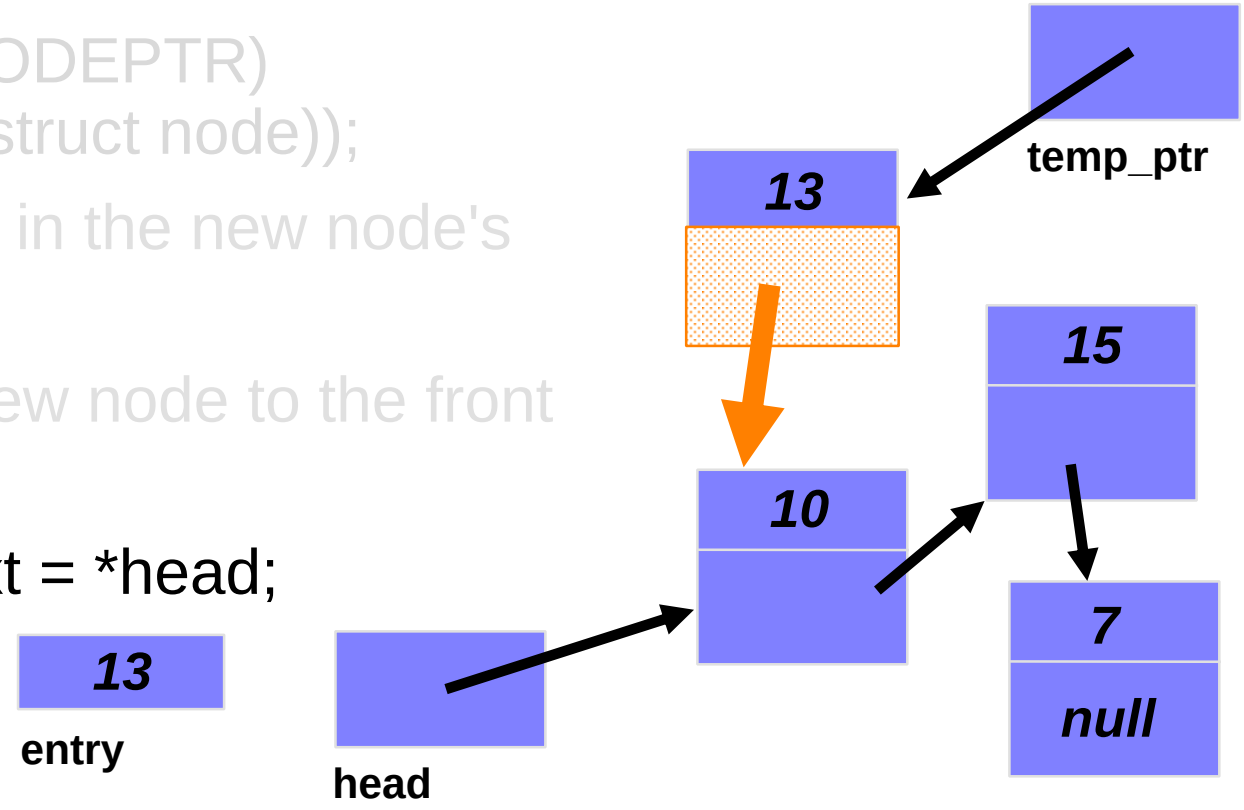
- temp_ptr = (NODEPTR) malloc(sizeof(struct node));
- Place the data in the new node's data_field.
- temp_ptr->data = 13;



Inserting a Node at the Front

```
void insert_start(NODEPTR *head, int x);
```

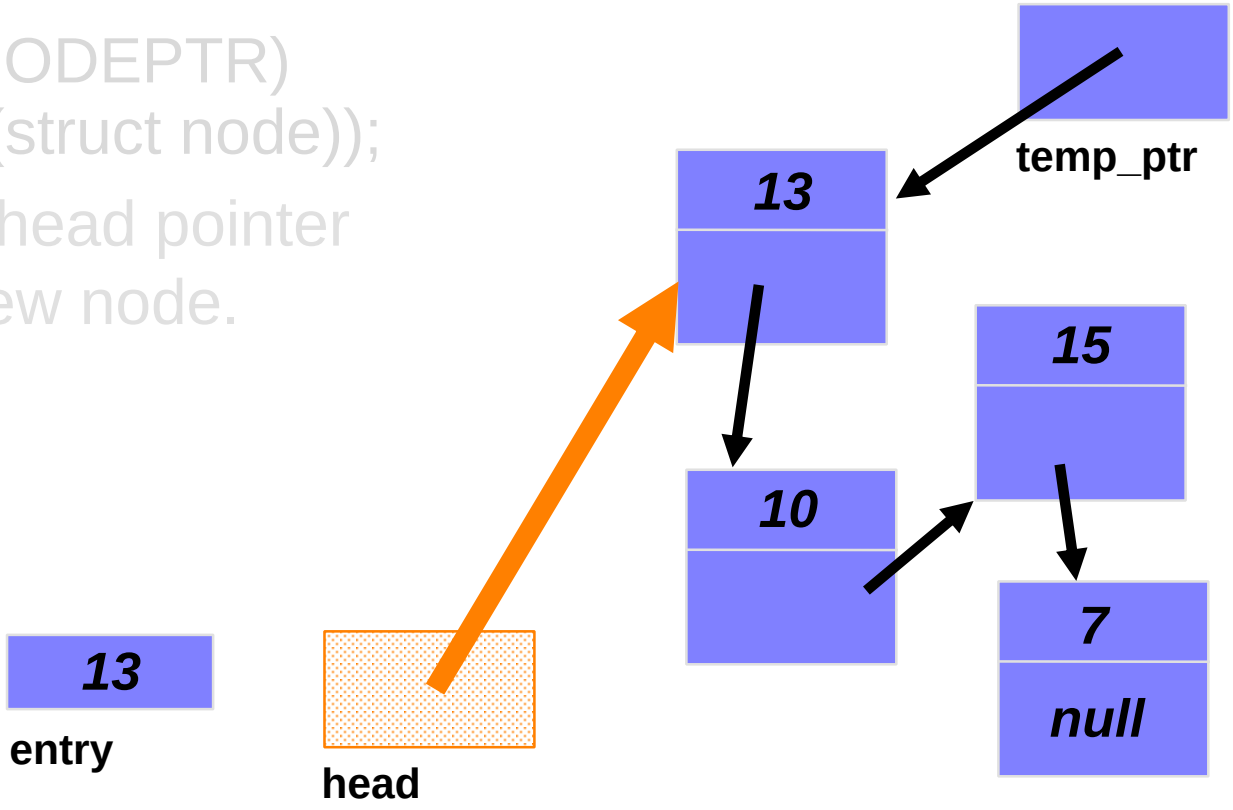
- temp_ptr = (NODEPTR) malloc(sizeof(struct node));
- Place the data in the new node's data_field.
- Connect the new node to the front of the list.
- temp_ptr->next = *head;



Inserting a Node at the Front

```
void insert_start(NODEPTR *head, int x);
```

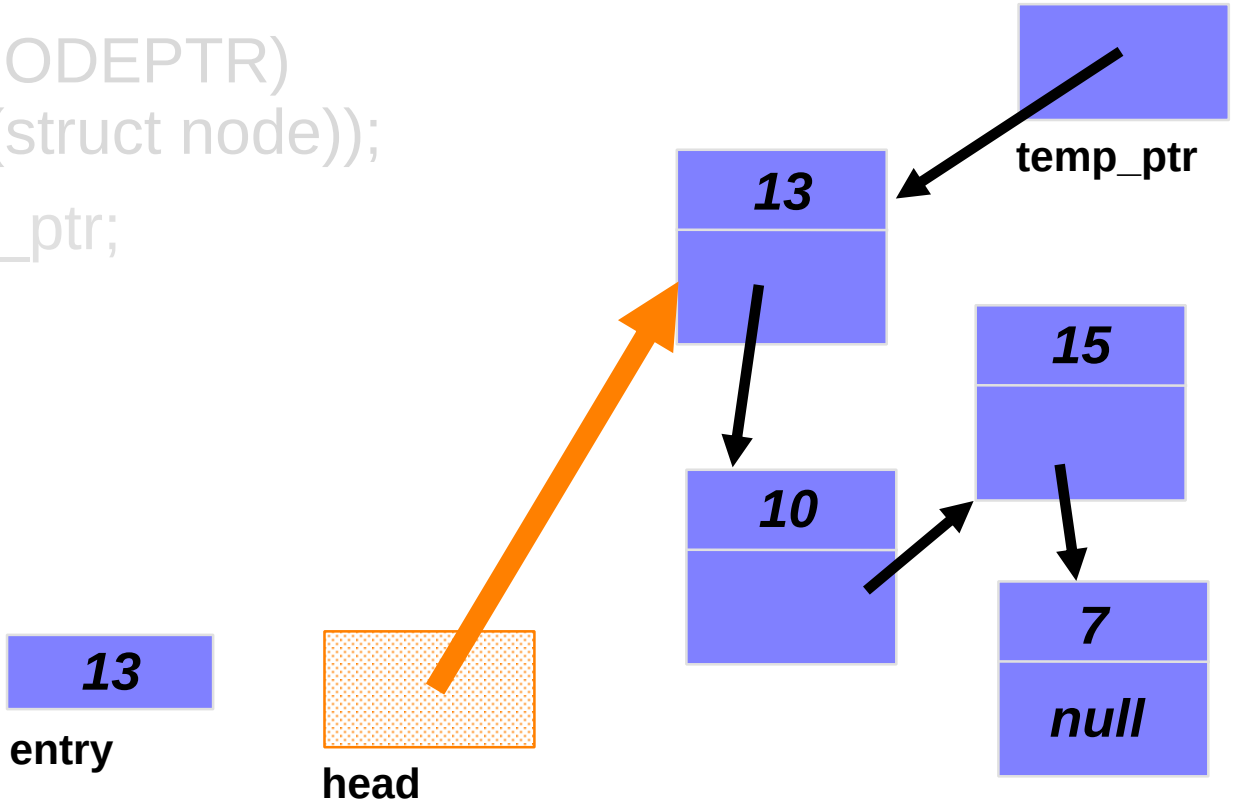
- temp_ptr = (NODEPTR) malloc(sizeof(struct node));
- Make the old head pointer point to the new node.



Inserting a Node at the Front

```
void insert_start(NODEPTR *head, int x);
```

- temp_ptr = (NODEPTR) malloc(sizeof(struct node));
- *head = temp_ptr;

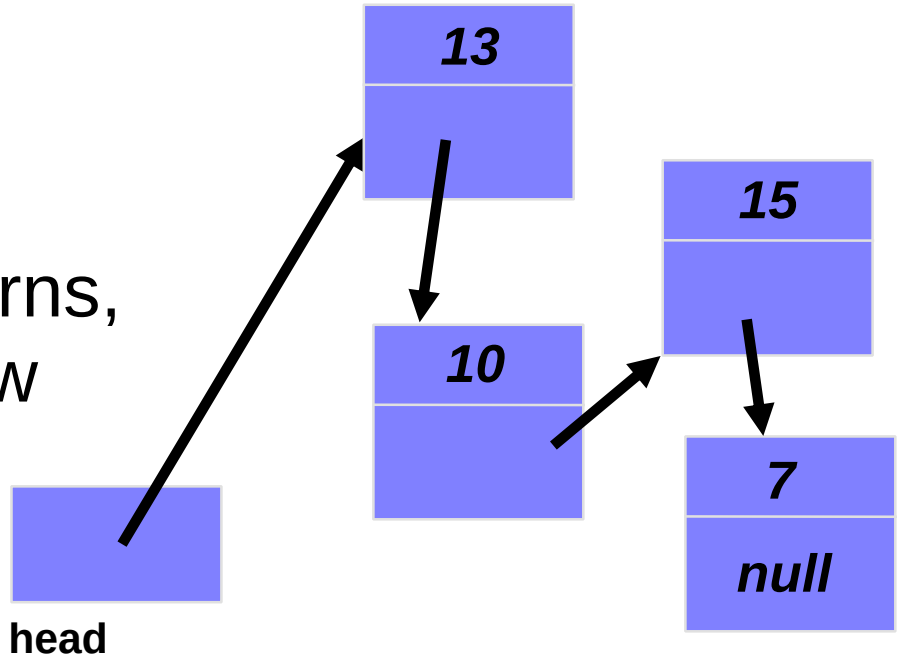


Inserting a Node at the Front

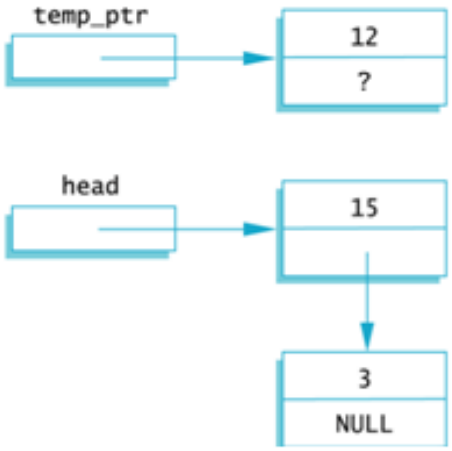
```
void insert_start(NODEPTR *head, int x);
```

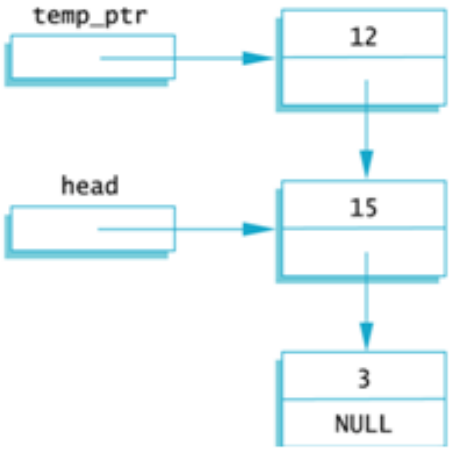
- temp_ptr = (NODEPTR) malloc(sizeof(struct node));
- *head = temp_ptr;

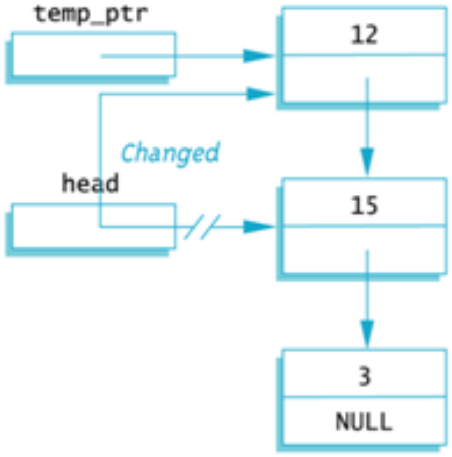
When the function returns, the linked list has a new node at the front.

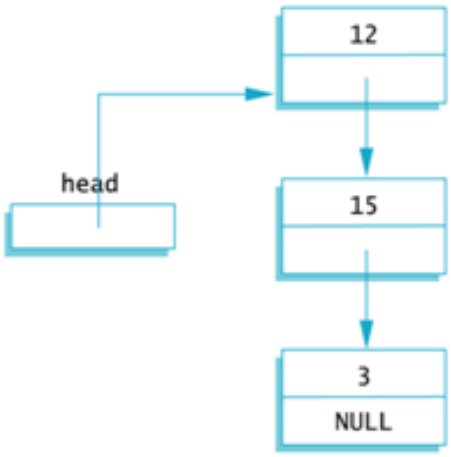


```
void insert_start(NODEPTR *head, int x)
{
  NODEPTR temp_ptr  = (NODEPTR) malloc(sizeof(struct node));
  temp_ptr->data = x;
  temp_ptr->next = *head;
  *head = temp_ptr;
}
```



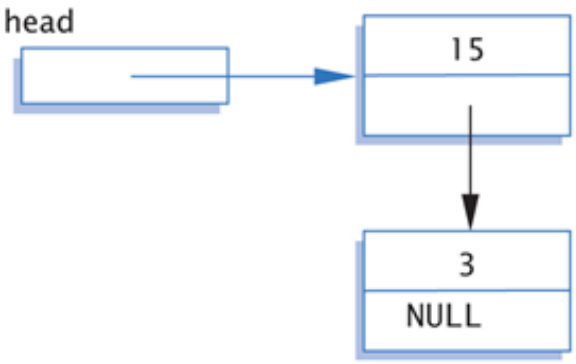




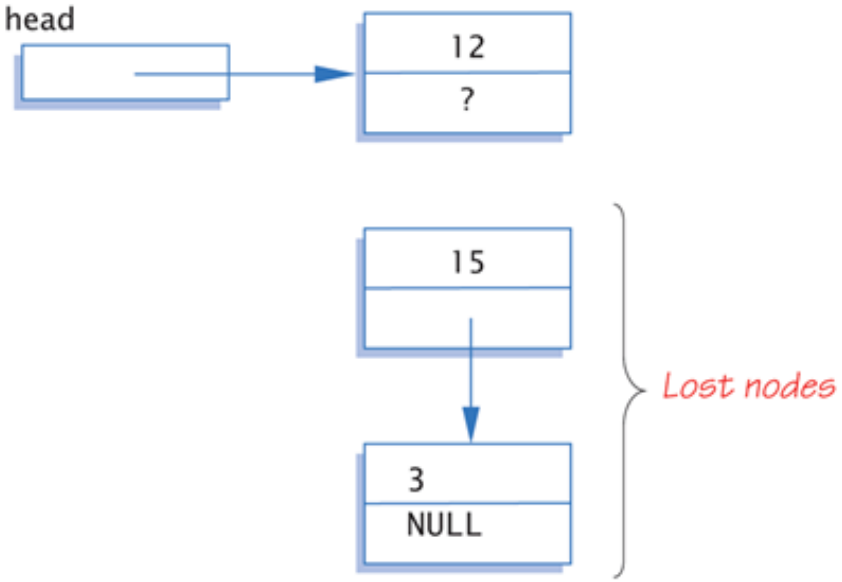


Lost Nodes Pitfall

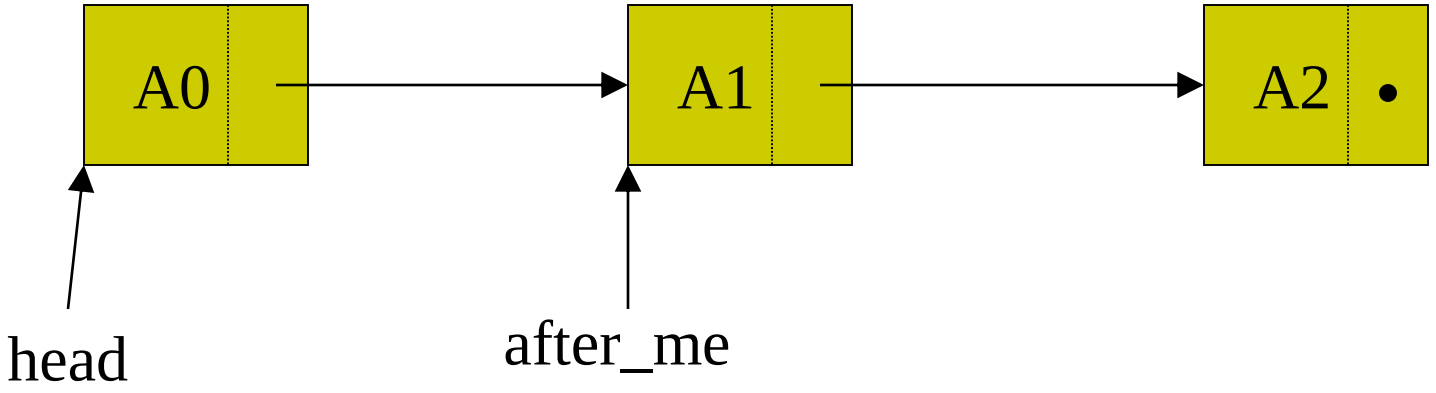
Linked list before insertion



Situation after executing
head = (NODEPTR) malloc(sizeof(struct node));
head->data=12;



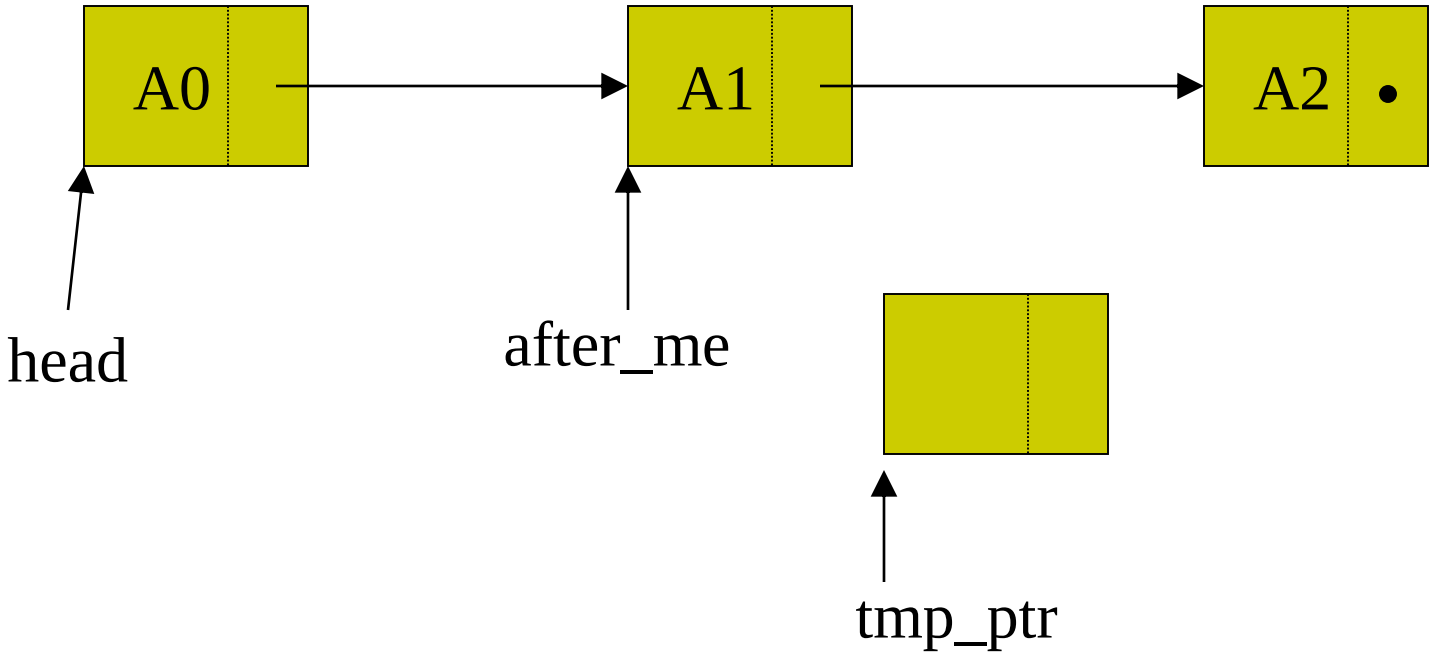
Inserting element in middle



At any point, we can add a new item **x** by doing this:

```
   temp_ptr = (NODEPTR) malloc(sizeof(struct
node));
   temp_ptr->data = x;
   temp_ptr->next = after_me->next;
   after_me->next = temp_ptr;
```

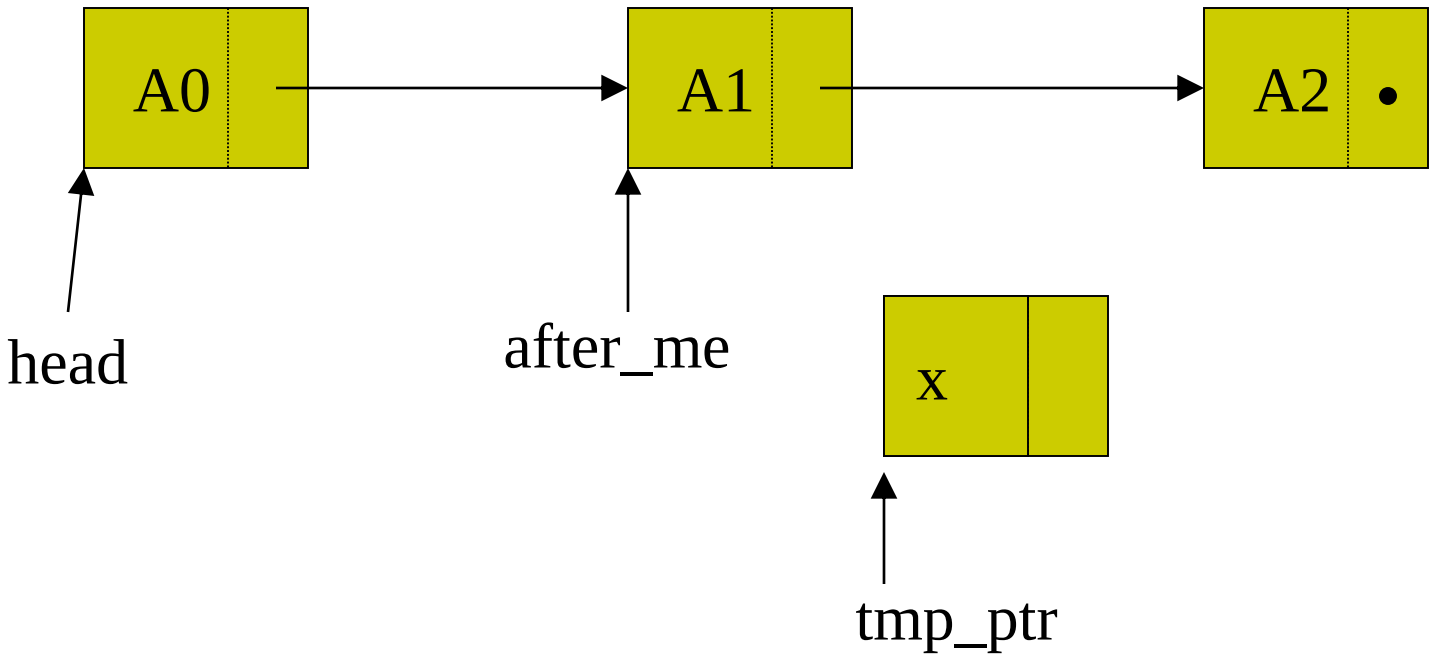
Inserting element in middle



At any point, we can add a new item **x** by doing this:

```
  temp_ptr = (NODEPTR) malloc(sizeof(struct
node));
  temp_ptr->data = x;
  temp_ptr->next = after_me->next;
  after_me->next = temp_ptr;
```

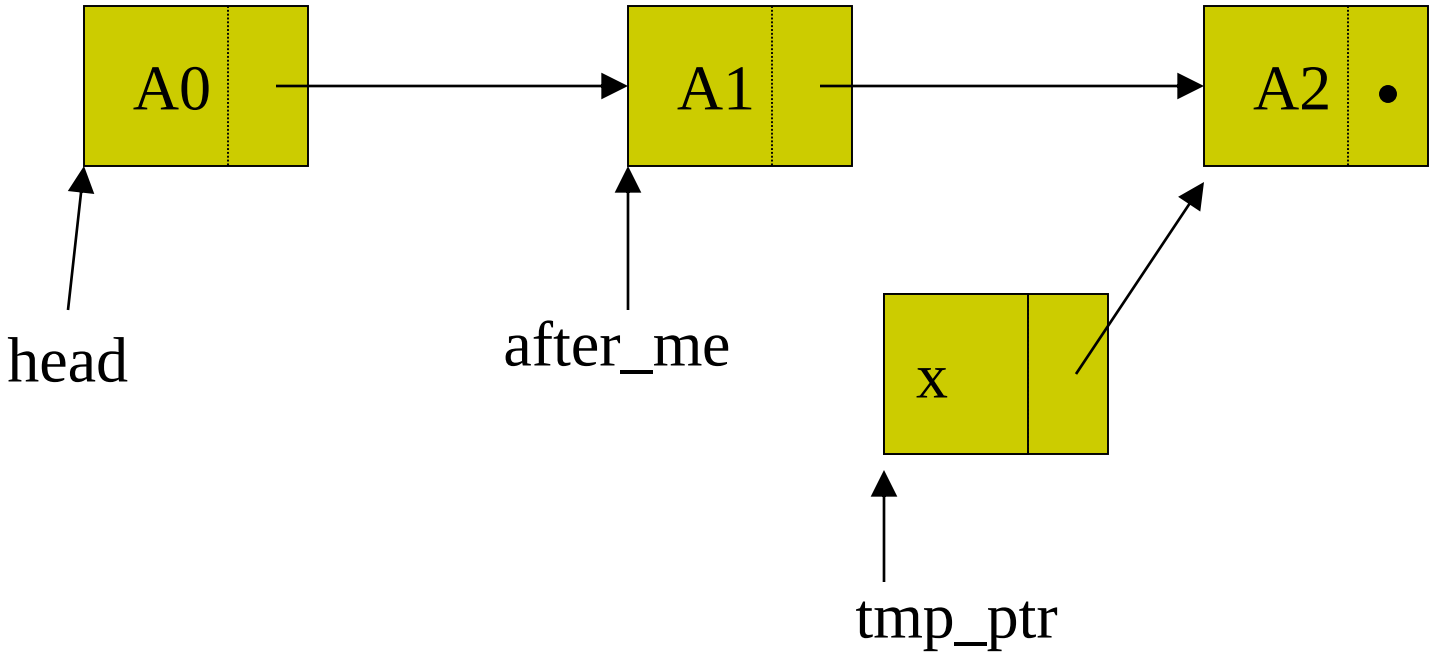
Inserting element in middle



At any point, we can add a new item **x** by doing this:

```
temp_ptr = (NODEPTR) malloc(sizeof(struct node));
temp_ptr->data = x;
temp_ptr->next = after_me->next;
after_me->next = temp_ptr;
```

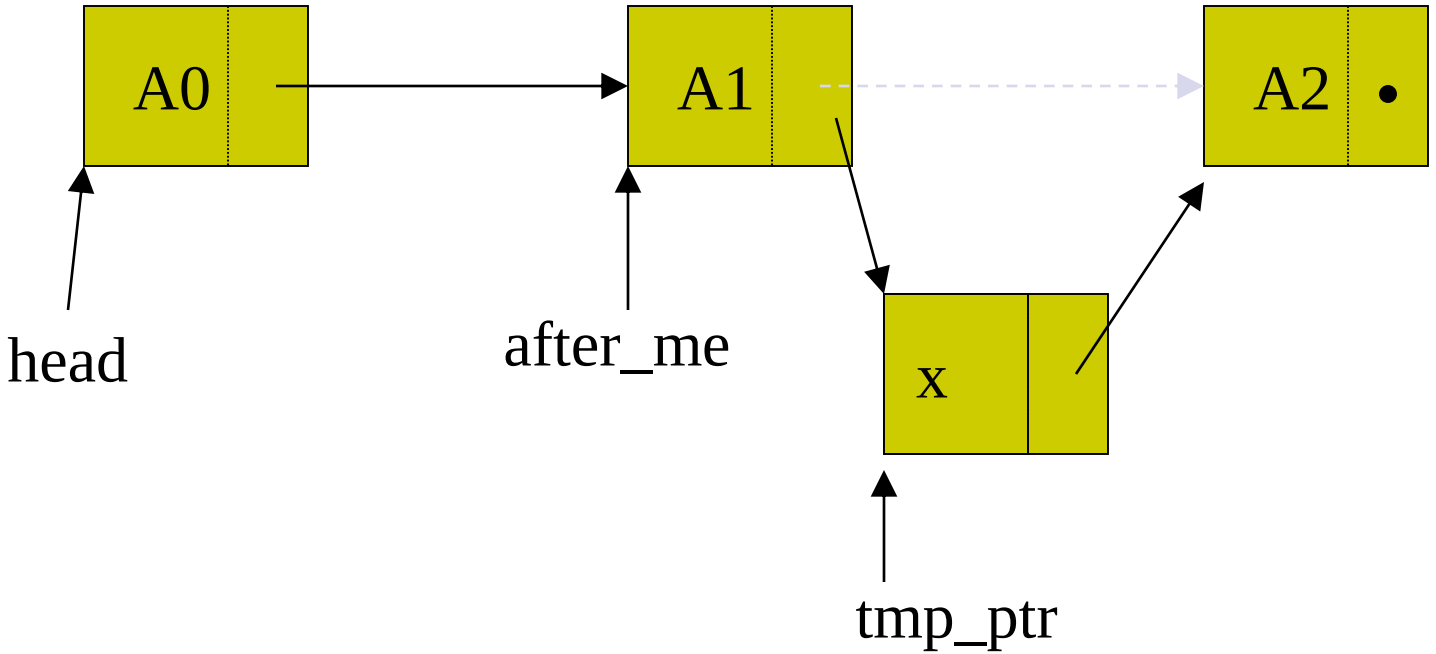
Inserting element in middle



At any point, we can add a new item **x** by doing this:

```
  temp_ptr = (NODEPTR) malloc(sizeof(struct
node));
  temp_ptr->data = x;
  temp_ptr->next = after_me->next;
  after_me->next = temp_ptr;
```

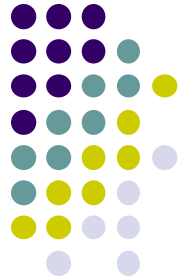
Inserting element in middle



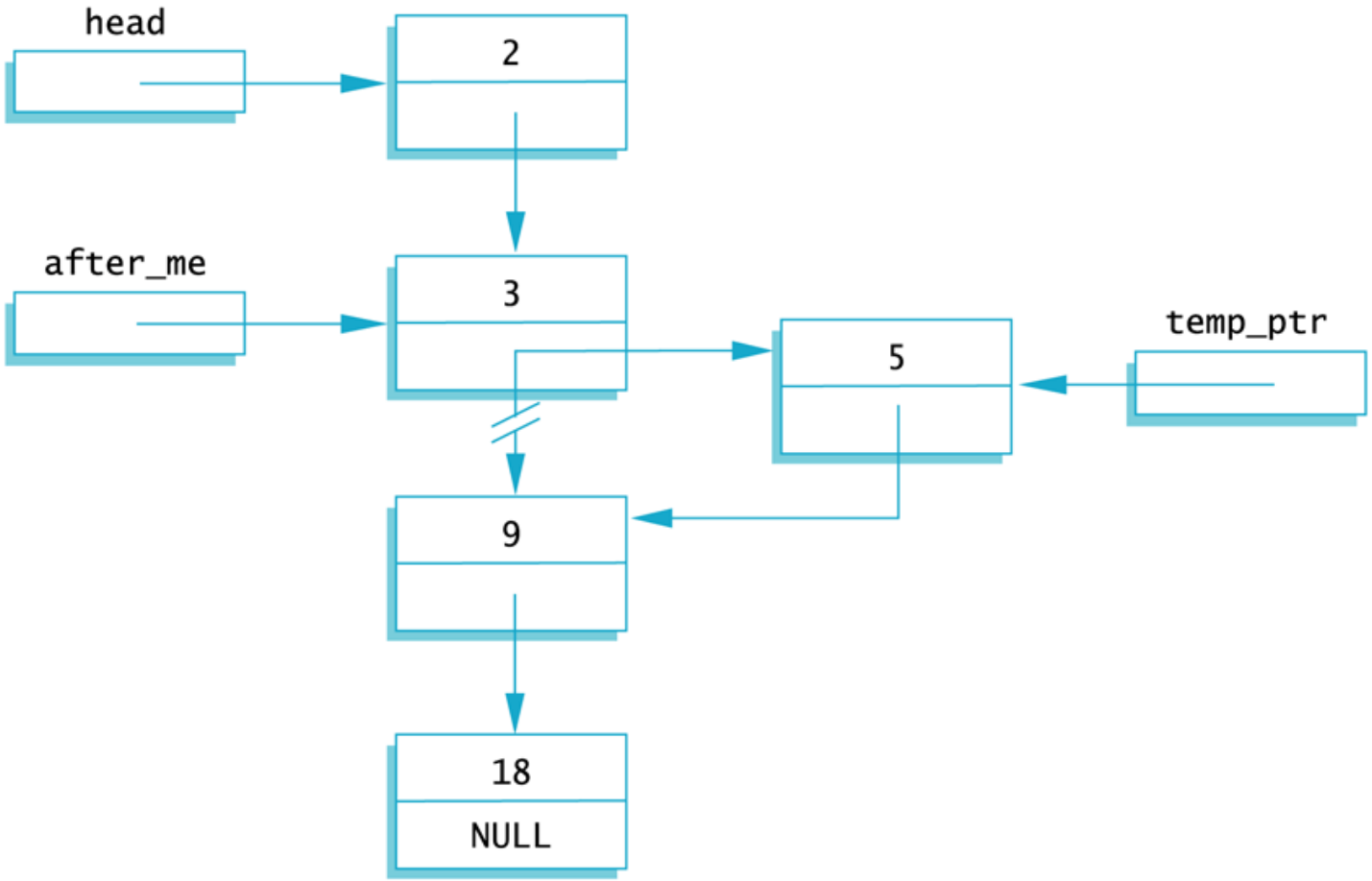
At any point, we can add a new item **x** by doing this:

```
temp_ptr = (NODEPTR) malloc(sizeof(struct node));
temp_ptr->data = x;
temp_ptr->next = after_me->next;
after_me->next = temp_ptr;
```

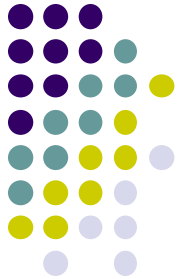
```
void insert_after(NODEPTR after_me, int x)
{
  NODEPTR temp_ptr = (NODEPTR) malloc(sizeof(struct node));
  temp_ptr ->data = x;
  temp_ptr ->next = after_me->next;
  after_me->next = temp_ptr;
}
```



Inserting in the Middle of a Linked List



Adding a node to a sorted list

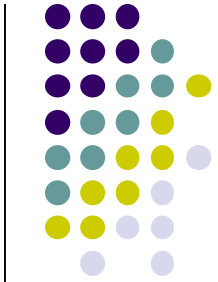


You are given a linked list in which the elements are ordered. You want to add a node with a new element, but you want to keep the list still ordered.

To do so, you have to spot the place to insert the new node by traversing the list. Here, there are some cases to consider:

- What if the list is empty?
- What if I need to add a node before the beginning of the list?
- What if I need to add a node somewhere inside the list?
  - Caution you cannot go back!
- What if I need to add a node at the end of the list?

Adding a node to a sorted list

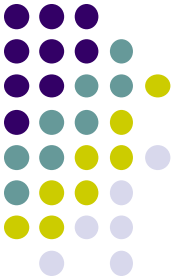


Create a new node.
Store data in the new node.
if there are no nodes in the list or new value smaller than value in first node
    Make the new node the first node.
else
    Find the first node whose value is greater than or equal the new value, or the end of the list (whichever is first).
    Insert the new node before the found node, or at end of the list if no node was found.
endif

Adding a node to a sorted list (Program)

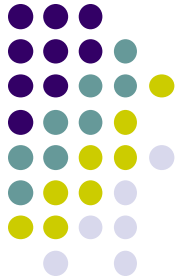
```
main()
{
  NODEPTR head = NULL, p,  prev;
  int i;

  scanf("%d" ,&i);
  insert_start(&head, i);
  scanf("%d", &i);
  while (i)
  {
    if(head->data>i) insert_start(&head, i);
    else
    {
    for(p=head; p!=NULL; prev=p, p=p->next)
      if(p->data > i) {insert_after(prev, i); break;}
    if(p==NULL) insert_after(prev, i);
    }
    scanf("%d", &i);
  }
 }
```

Adding a node to a sorted list (Program)



```
main()
{
  NODEPTR head = NULL, p,  prev;
  int i;

  scanf("%d", &i);
  while (i)
  {
     for(prev = NULL, p= head; (p != NULL)&& (i > p->data);
prev = p, p = p->next);
     if(prev==NULL)
        insert_start(&head, i);
     else
        insert_after(prev, i);
    scanf("%d", &i);
  }
}
```

A few questions on linked lists



- Print a singly-linked list backwards with and without recursion.
- Write a C program to copy a linked list.
- Reorder A->B->C->D->E as B->A->D->C->E in a singly linked list
- Write a function to return the Nth-to-Last element in a singly linked list of unknown length. If N = 0, then your function must return the last element.
- Write a function to reverse a Linked-list
- How would you find a loop in a singly-linked list?
- Find the middle element in a singly linked list without counting the number of elements.

A few questions on linked lists

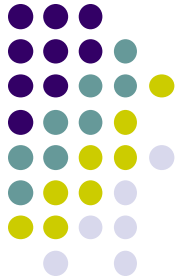
- Given an integer linked list of which both first half and second half are sorted independently. Write a function to merge the two parts to create one single sorted linked list in place [do not use any extra space].
- Sample test case:
  - Input 1 :1->2->3->4->5->1->2; Output: 1->1->2->2->3->4->5
  - Input 2: 1->5->7->9->11->2->4->6; Output 2: 1->2->4->5->6->7->9->11
- Given two linked lists, return the intersection of the two lists: i.e. return a list containing only the elements that occur in both of the input lists.

Another interview question

- How would you find a loop in a singly-linked list?

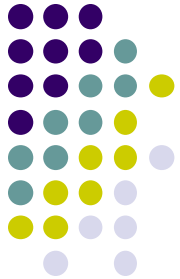
```
int detectloop(NODEPTR home)
{
  NODEPTR slow_p = home, fast_p = home;
  while(fast_p &&  fast_p->next )
  {
    slow_p = slow_p->next;
    fast_p  = fast_p->next->next;
    if (slow_p == fast_p)
       return 1;
   }
   return 0;
}
```

A more challenging question



- How would remove the loop in a singly-linked list, if it exists?

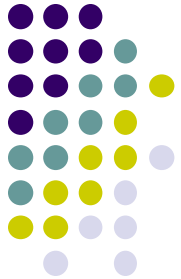
A more challenging question



- How would remove the loop in a singly-linked list, if it exists?

1. Detect Loop using Floyd's Cycle detection algo and get the pointer to a loop node.
2. Count the number of nodes in loop. Let the count be k.
3. Fix one pointer to the head and another to kth node from head.
4. Move both pointers at the same pace, they will meet at loop starting node.
5. Get pointer to the last node of loop and make next of it as NULL.

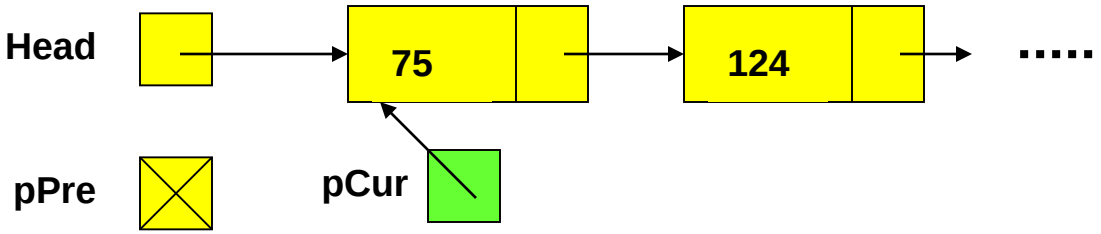
Deleting a Node from a Linked List



- Deleting a node requires that we logically remove the node from the list by changing various links and then physically deleting the node from the list (i.e., return it to the heap).
- Any node in the list can be deleted. Note that if the only node in the list is to be deleted, an empty list will result. In this case the head pointer will be set to NULL.
- To logically delete a node:
  - First locate the node itself (pCur) and its logical predecessor (pPre).
  - Change the predecessor's link field to point to the deleted node's successor (located at pCur -> next).
  - Recycle the node using the free() function.

Deleting the First Node from a Linked List

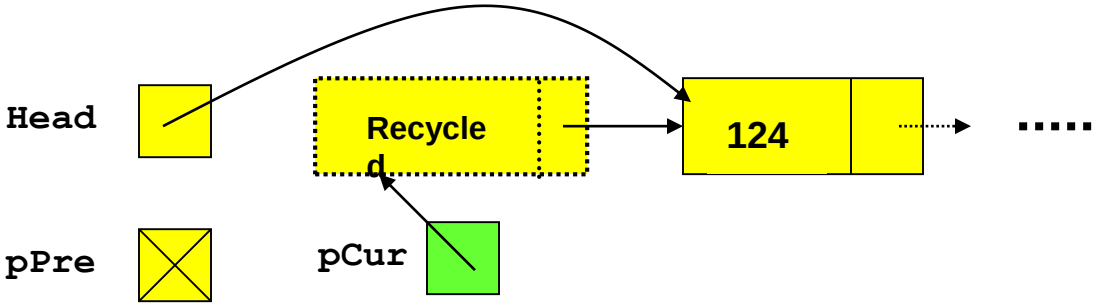
Before:



Code:

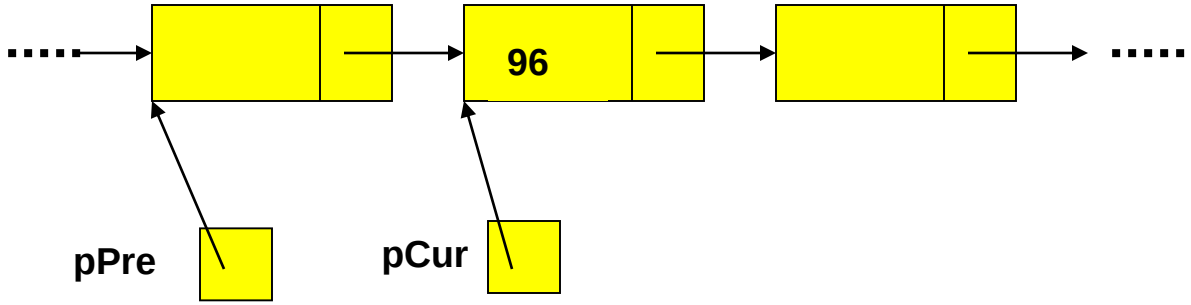
```
Head = pCur -> next;
free(pCur);
```

After:



Deleting a Node from a Linked List – General Case

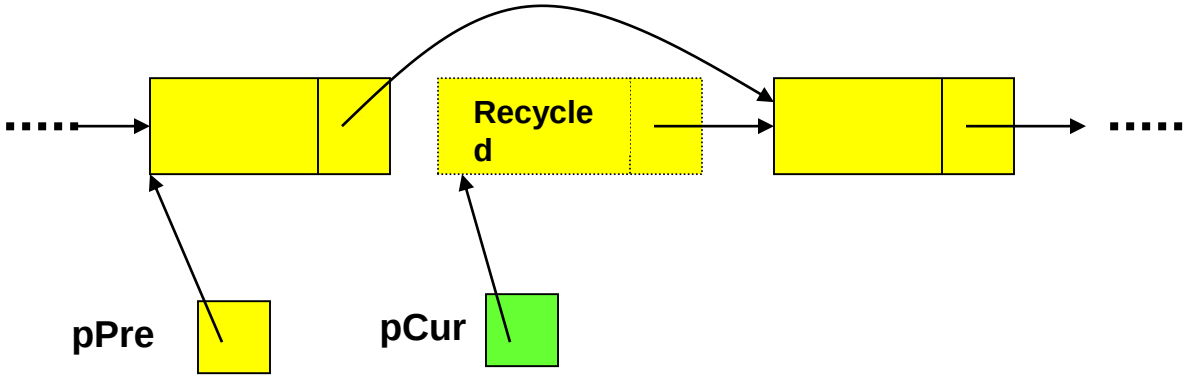
Before:



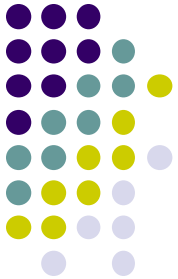
Code:

```
pPre -> next = pCur -> next;
free(pCur);
```

After:



Deleting a Node From Start of Linked List

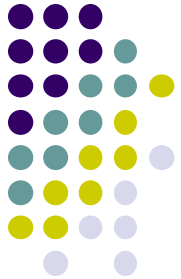


```
int delete_start(NODEPTR *p)
{
   int i;
   NODEPTR q;
   if(*p==NULL)return 0;
   i = (*p)->data;
   q = (*p)->next;
   free(*p);
   *p = q;
   printf("\nValue deleted =%d\n", i);
   return i;
}
```

Deleting a Node From a Linked List

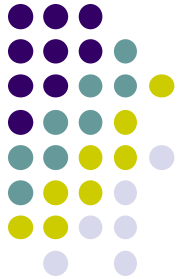
```
int delete_after(NODEPTR p)
{
   int i;
   NODEPTR q;
   if(p==NULL)return 0;
   if(p->next==NULL) return 0;
   q = p->next;
   p->next = p->next->next;
   i = q->data;
   free(q);
   printf("\nValue deleted = %d\n", i);
   return i;
}
```

A few questions on linked lists



- Write a function to delete all nodes of a linked list

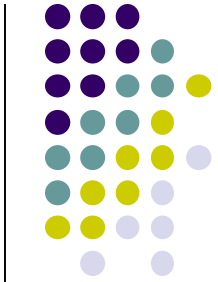
A few questions on linked lists



- Write a function to delete all nodes of a linked list

```
void delete_list(NODEPTR *list)
{
     while(*list)
          delete_start(list);
}
```

A few questions on linked lists



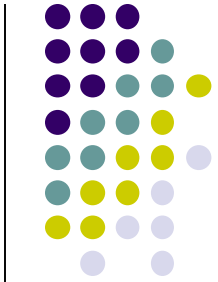
- Write a function to remove alternate nodes from a given linked list

A few questions on linked lists

- Write a function to remove alternate nodes from a given linked list

```
void delete_alternate(NODEPTR list)
{
      NODEPTR p=list;

      while(p!=NULL && p->next != NULL)
      {
            delete_after(p);
            p=p->next;
      }
}
```

A few questions on linked lists



- Write a function to delete all nodes containing given value

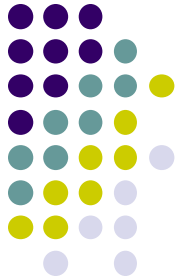
```
void delete_all_val(NODEPTR *list, int val)
```

A few questions on linked lists

- Write a function to delete all nodes containing given value

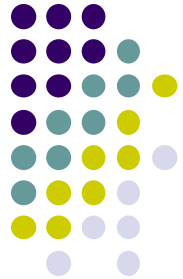
```
void delete_all_val(NODEPTR *list, int val)
{
        NODEPTR p=*list;
        while(*list != NULL && (*list)->data == val)
                delete_start(list);
        p=*list;
        while(p!=NULL && p->next != NULL)
        {
                if(p->next->data == val)
                        delete_after(p);
                else p=p->next;
        }
}
```

A few questions on linked lists



- Write a function to delete all duplicates in a linked list

A few questions on linked lists

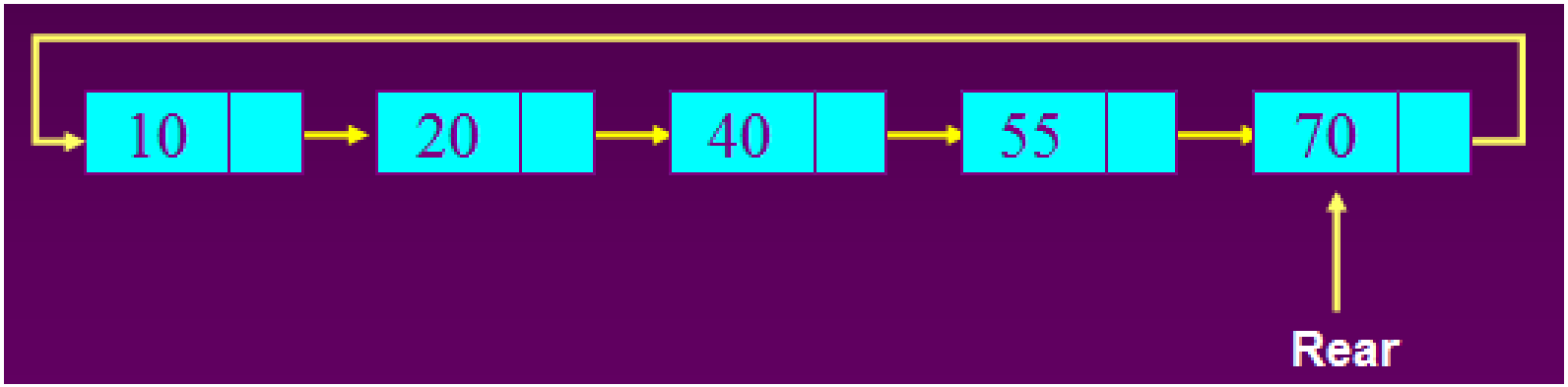


- Write a function to delete all duplicates in a linked list

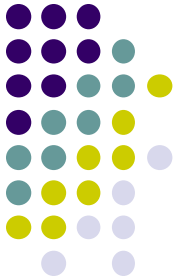
```
void remove_duplicate(NODEPTR *list)
{
  NODEPTR p=*list;
  while(p!=NULL)
  {
    delete_all_val(&(p->next), p->data);
     p=p->next;
  }
}
```

Circular Linked Lists

- A Circular Linked List is a special type of Linked List. No node contains *NULL*
- 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 (Why?)

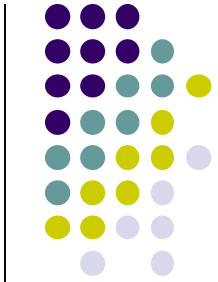


Circular Linked List Definition



```
struct Node{
  int data;
   struct Node *next;
};
typedef struct Node *NODEPTR;
```

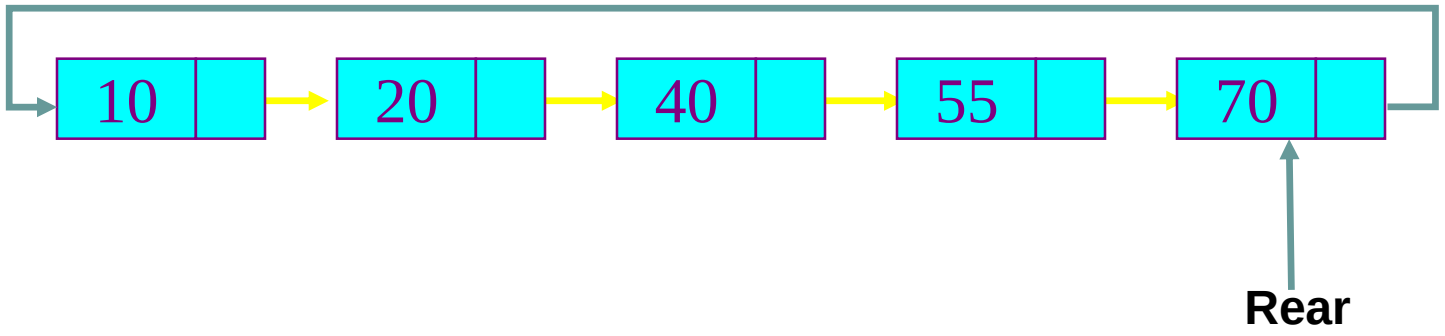
Circular Linked List Operations



- `print(NODEPTR Rear)`
  `//print the Circular Linked List once`
- `insertNode(NODEPTR Rear, int item)`
  `//add new node to ordered circular linked list`
- `deleteNode(NODEPTR Rear, int item)`
  `//remove a node from circular linked list`

Traverse the list

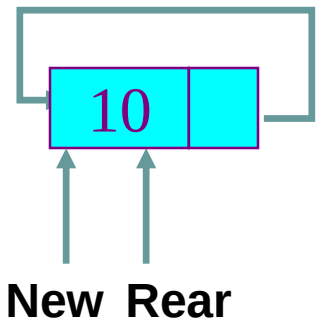
```
void print(NODEPTR Rear){
   NODEPTR Cur;
   if(Rear != NULL){
       Cur = Rear->next;
       do{
             printf("%d", Cur->data);
             Cur = Cur->next;
       }while(Cur != Rear->next);
   }
}
```

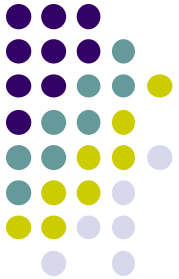


Insert Node

- Insert into an empty list

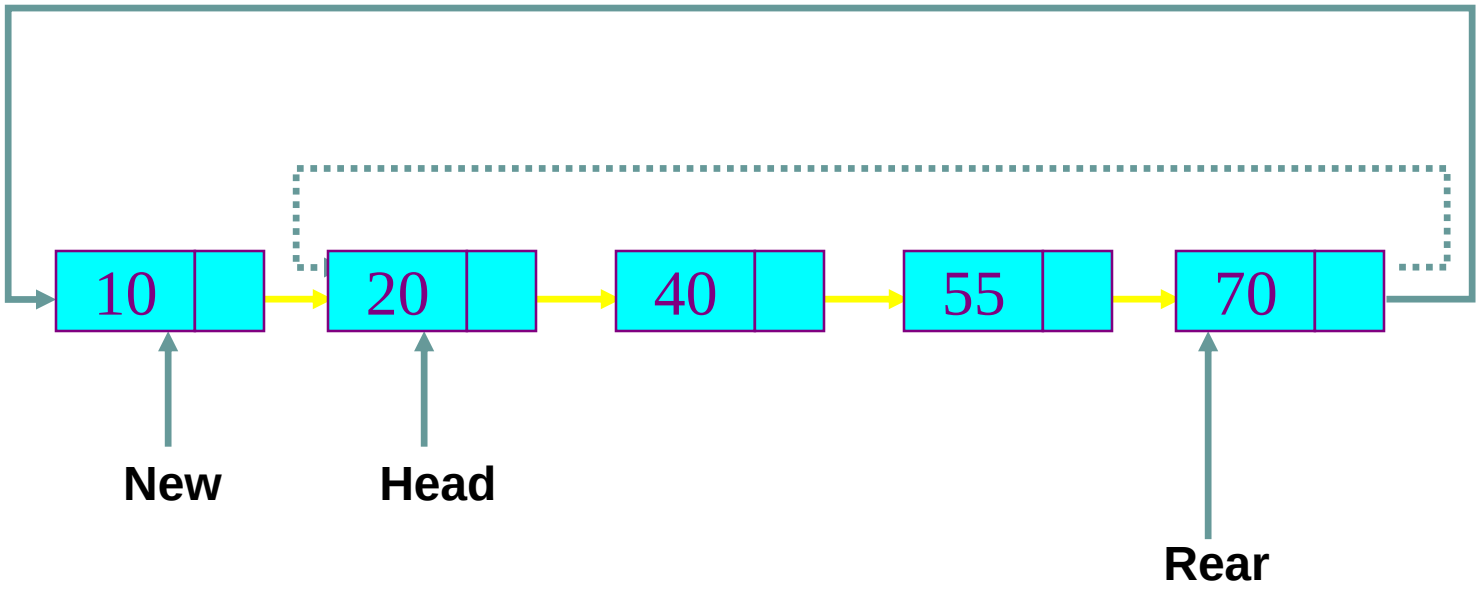
```
NODEPTR New =  malloc(sizeof(struct node));

New->data = 10;

Rear = New;
Rear->next = Rear;
```





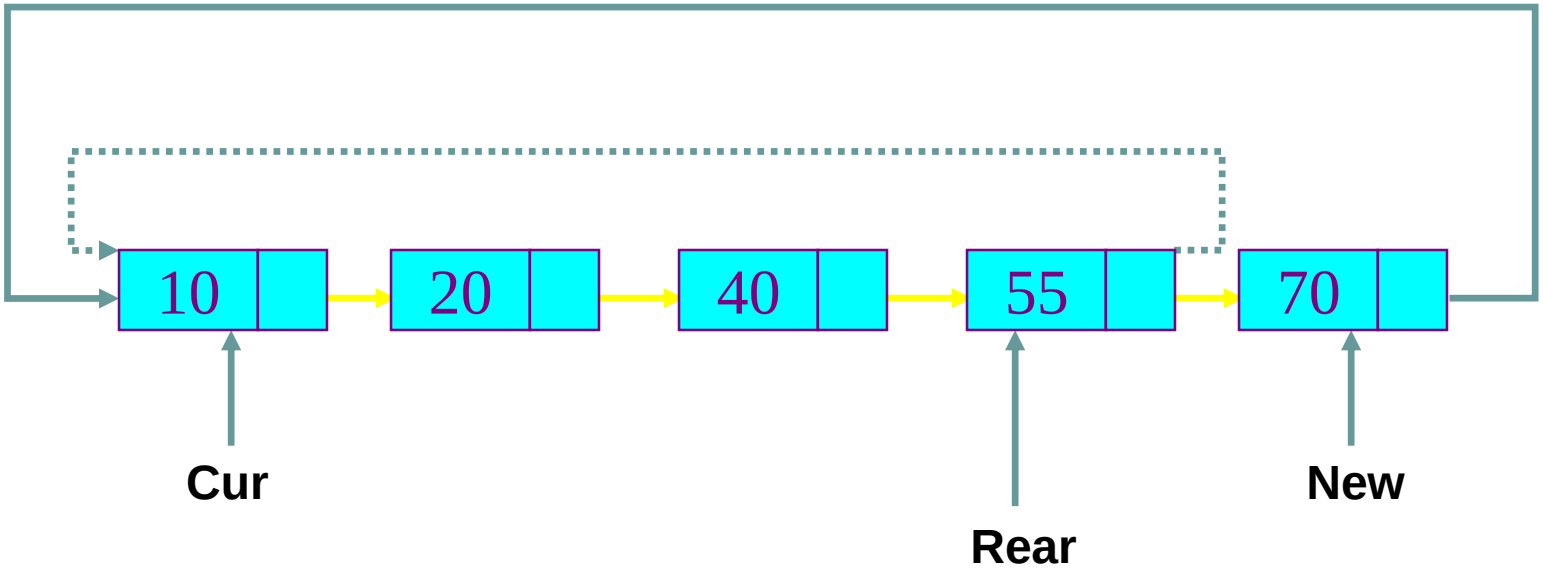
- Insert to head of a Circular Linked List

```
New->next = Rear->next;

Rear->next = New;
```



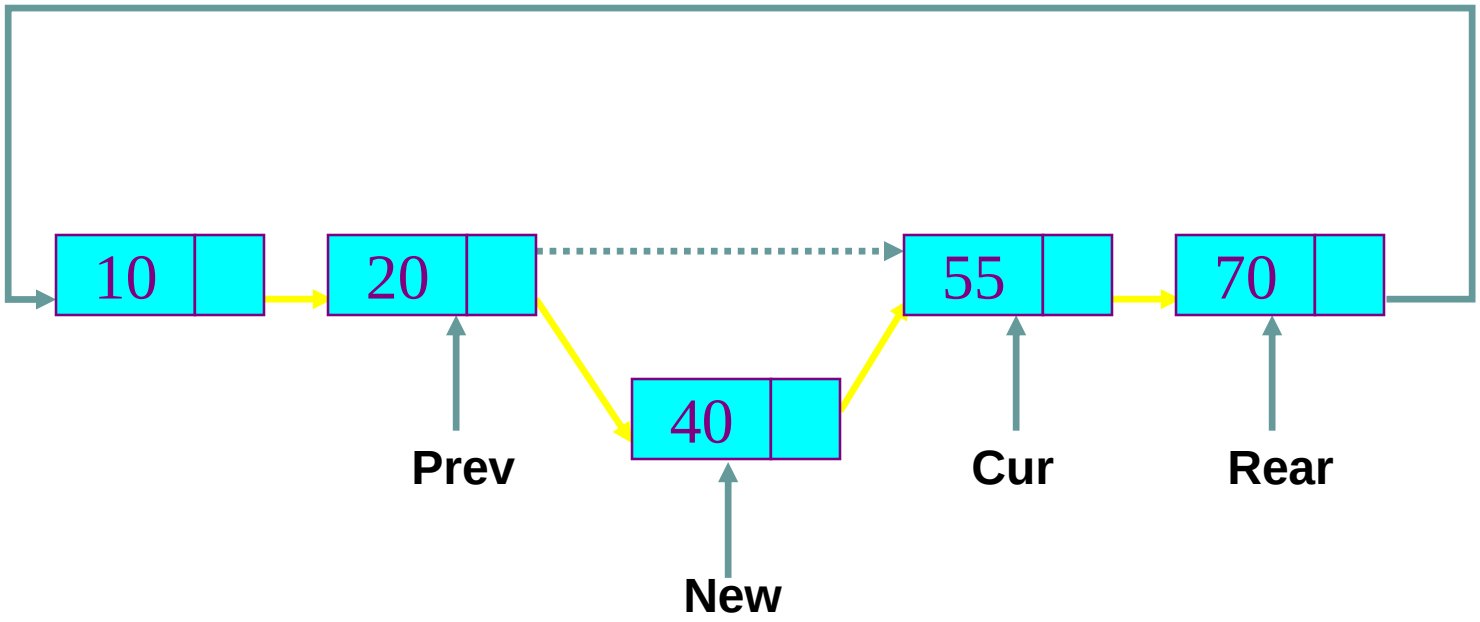
- Insert to end of a Circular Linked List

```
New->next = Rear->next;

Rear->next = New;

Rear = New;
```



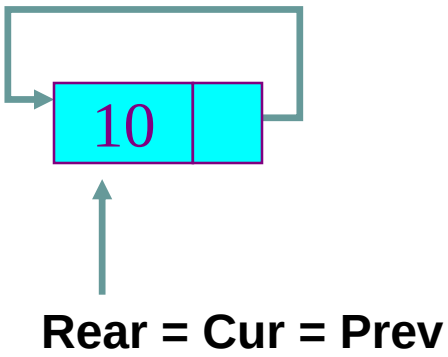
- Insert to middle of a Circular Linked List between `Prev` and `Cur`

```
New->next = Cur;

Prev->next = New;
```



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

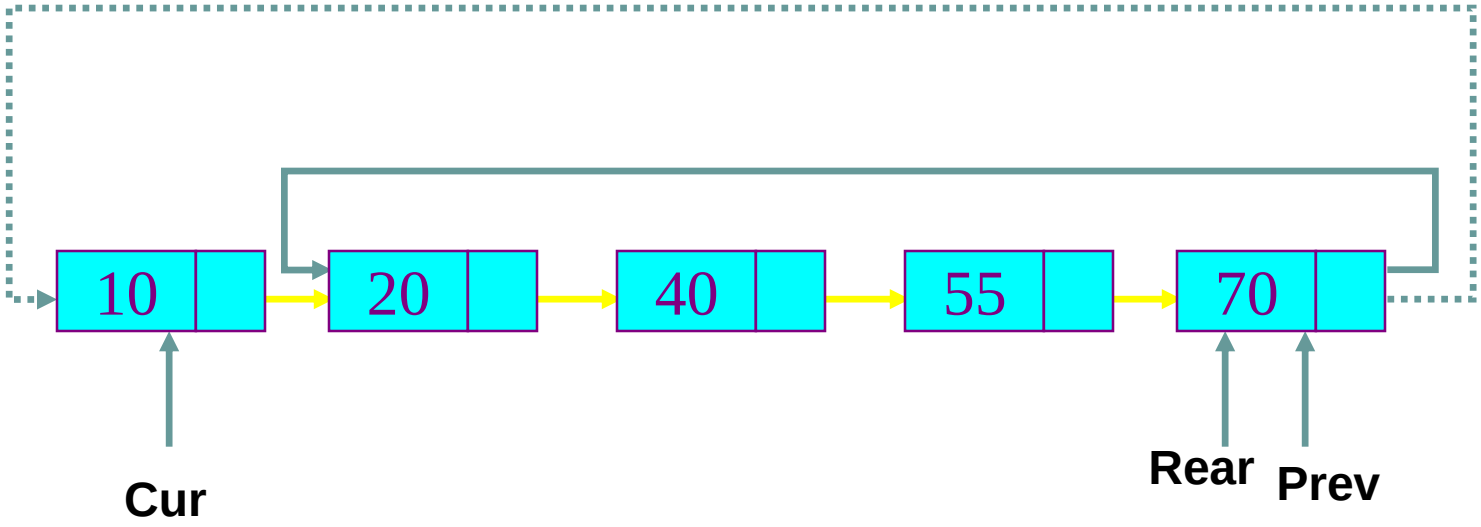
```
Rear = NULL;
Free(Cur);
```



Delete Node

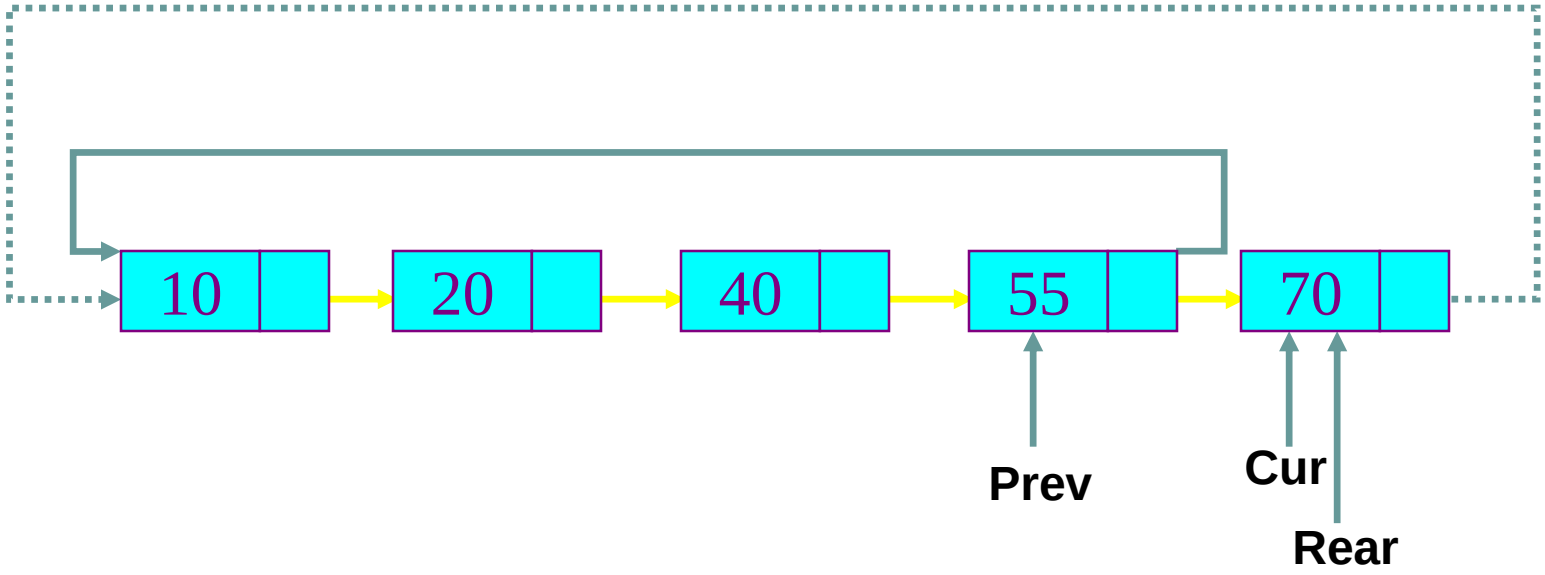
- Delete the head node from a Circular Linked List

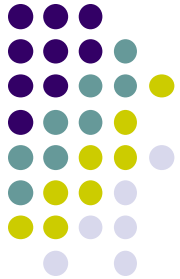
```
Cur = Rear->next;

Rear->next = Cur->next;

free(Cur);
```



- Delete the end node from a Circular Linked List

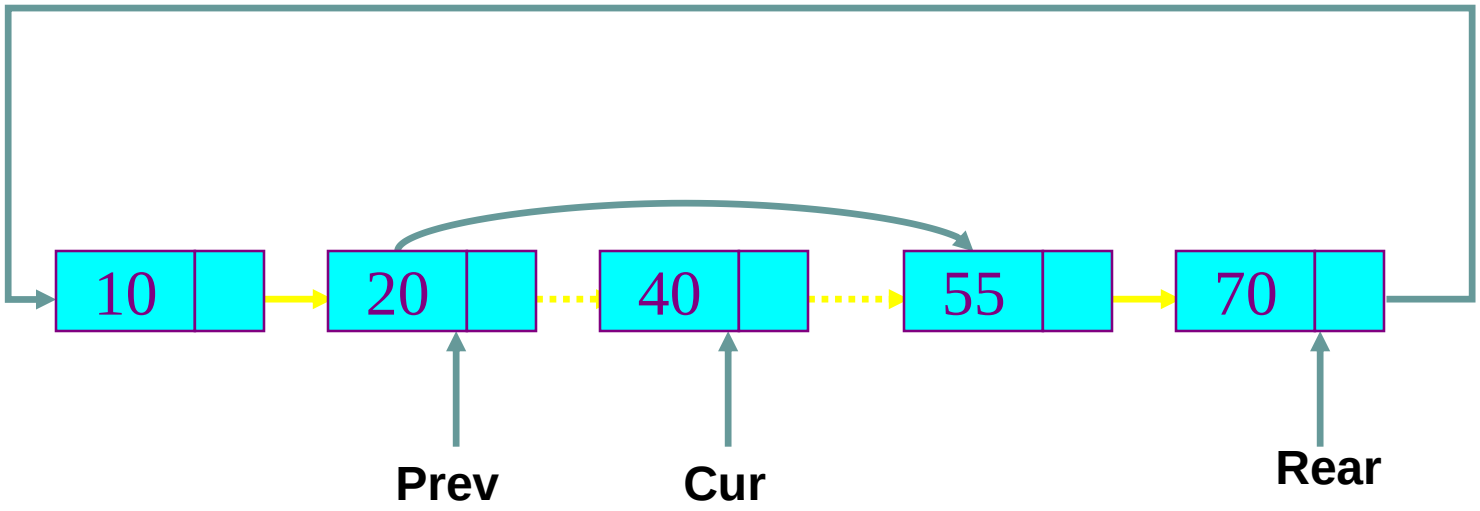
```
Prev->next = Rear->next;

free(Cur);

Rear = Prev;
```





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

```
Prev->next = Cur->next;

Free(Cur);
```



A few questions on linked lists

- Write a function to convert singly linked list to circular linked list.

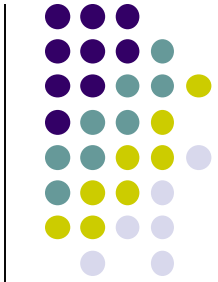
```
single_to_circular(NODEPTR *list)
```

A few questions on linked lists

- Write a function to convert singly linked list to circular linked list.

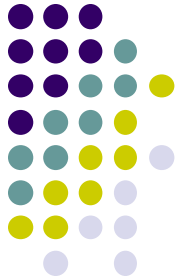
```
void single_to_circular(NODEPTR *list)
{
      NODEPTR p=*list;
      if(p==NULL) return;
      while(p->next)
            p = p->next;
      p->next = *list;
      *list = p;
}
```

Applications of singly linked lists



- Polynomials
- Sparse arrays
- File Allocation Table
- Big numbers

Polynomial Application



- Many high level polynomials have many terms with a 0 coefficient.
- Would need to store them if using an array.
- Use a linked list to not take memory for items with a 0 coefficient.

Polynomials

$$A(x) = a_{m-1}x^{e_{m-1}} + a_{m-2}x^{e_{m-2}} + \ldots + a_0 x^{e_0}$$

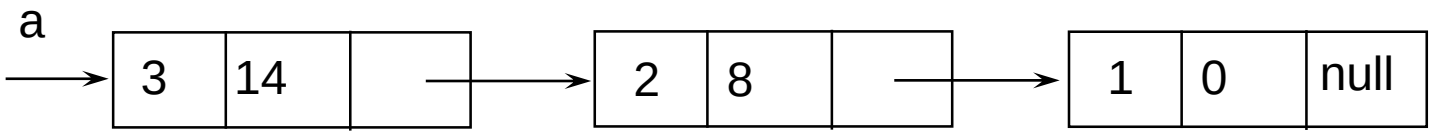
Representation

```
typedef struct poly_node *poly_pointer;
typedef struct poly_node {
    int coef;
    int expon;
    poly_pointer next;
};
poly_pointer a, b, c;
```

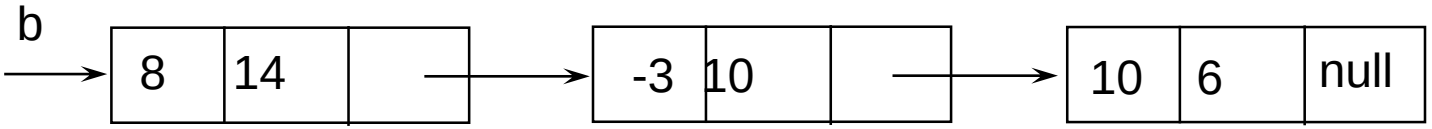
coef	expon	link

Example

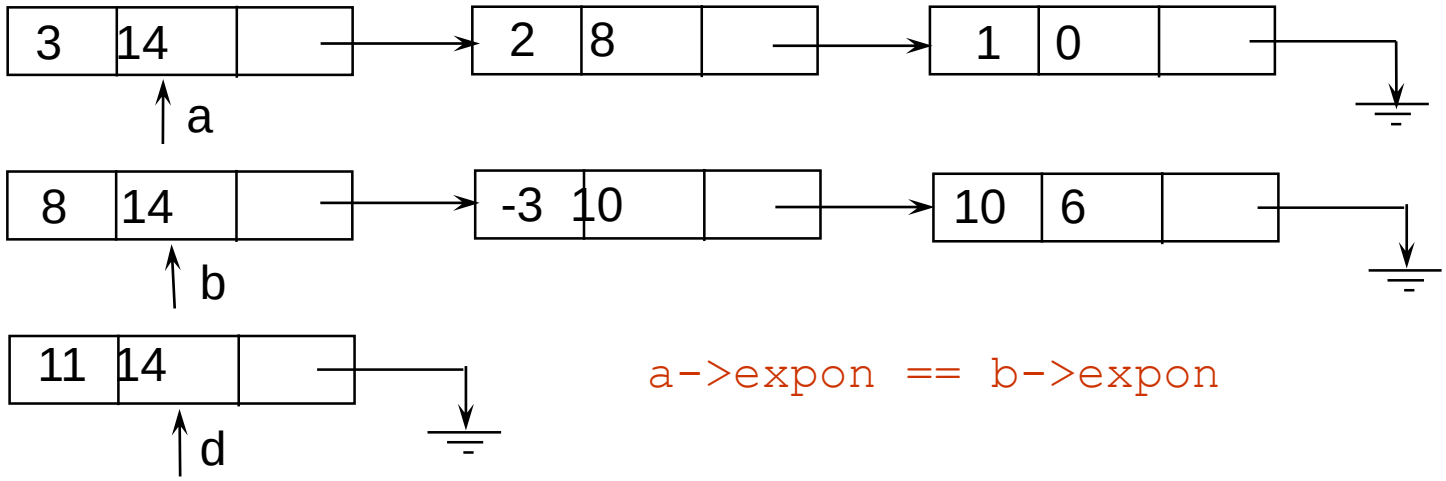
$$a = 3x^{14} + 2x^{8} + 1$$

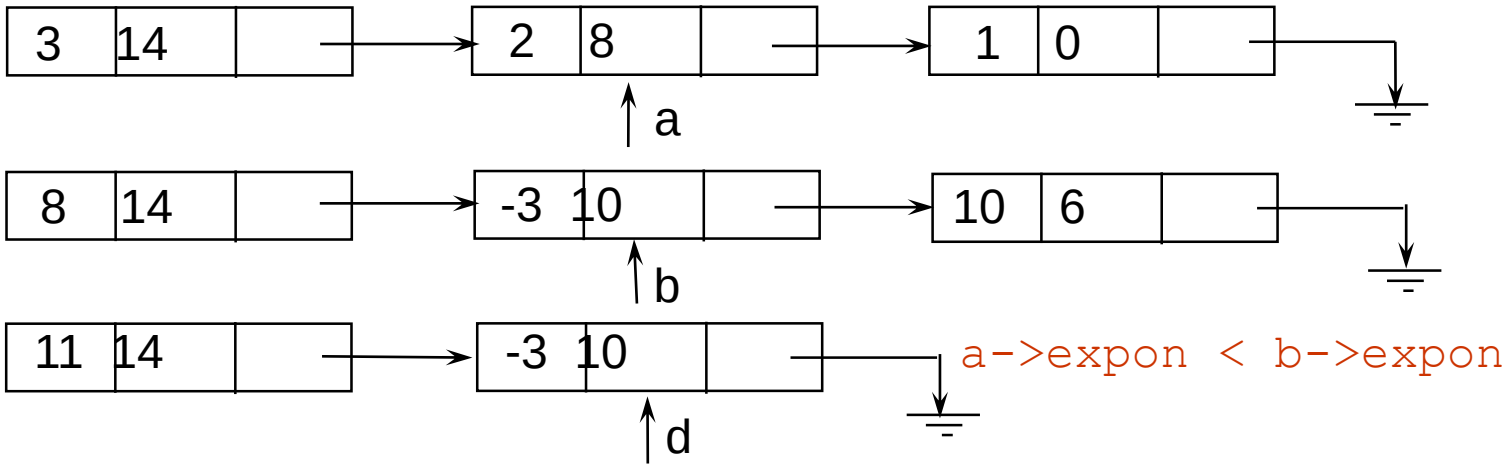


$$b = 8x^{14} - 3x^{10} + 10x^{6}$$

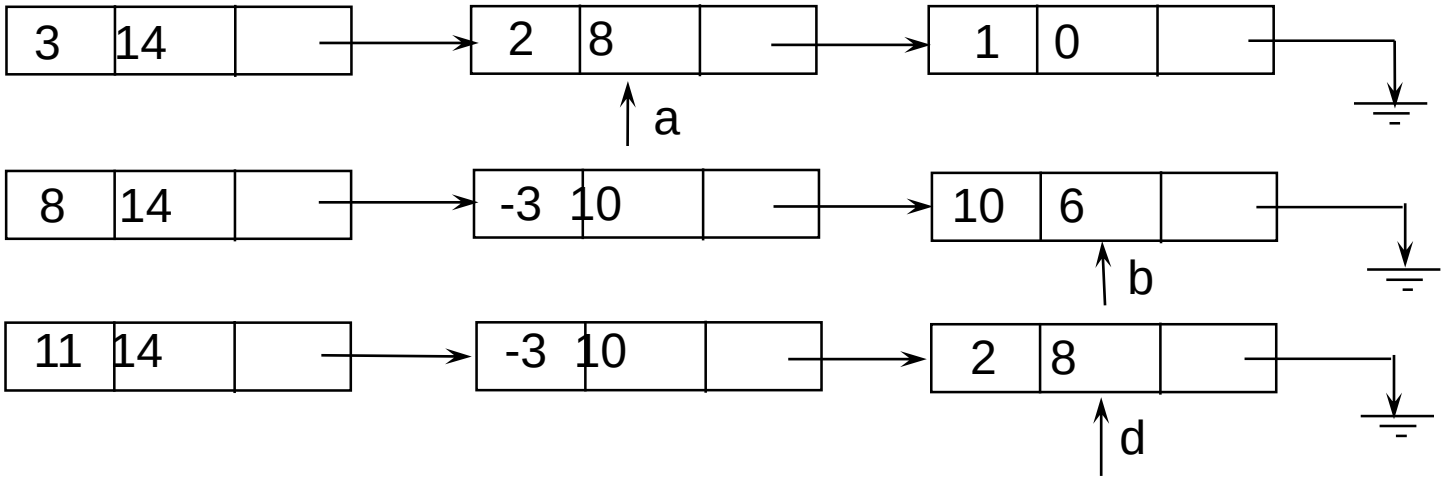


Adding Polynomials



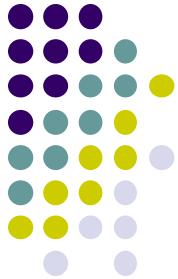


Adding Polynomials (cont'd)



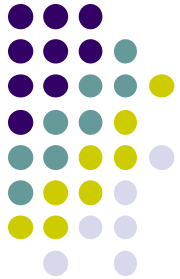

```
a->expon > b->expon
```

About sparse arrays



- A sparse array is simply an array most of whose entries are zero (or null, or some other default value)
- For example: Suppose you wanted a 2-dimensional array of course grades, whose rows are students and whose columns are courses
  - There are about 22,000 students
  - There are about 5000 courses
  - This array would have about 11,00,00,000 entries
  - Since most students take fewer than 5000 courses, there will be a lot of empty spaces in this array
  - This is a big array, even by modern standards
- There *are* ways to represent sparse arrays efficiently

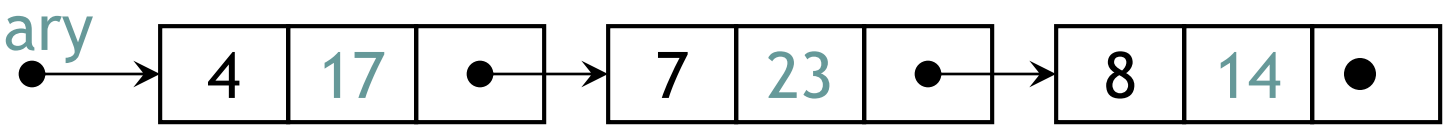
Sparse arrays as linked lists



- We will start with sparse one-dimensional arrays, which are simpler
  - We'll do sparse two-dimensional arrays later
- Here is an example of a sparse one-dimensional array:

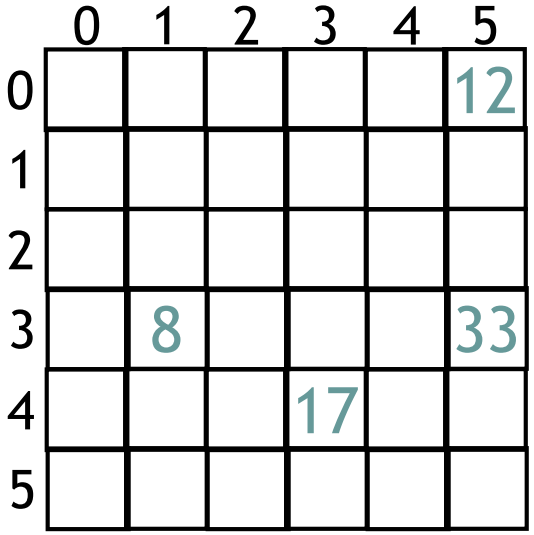
	0	1	2	3	4	5	6	7	8	9	10	11
ary	0	0	0	0	17	0	0	23	14	0	0	0

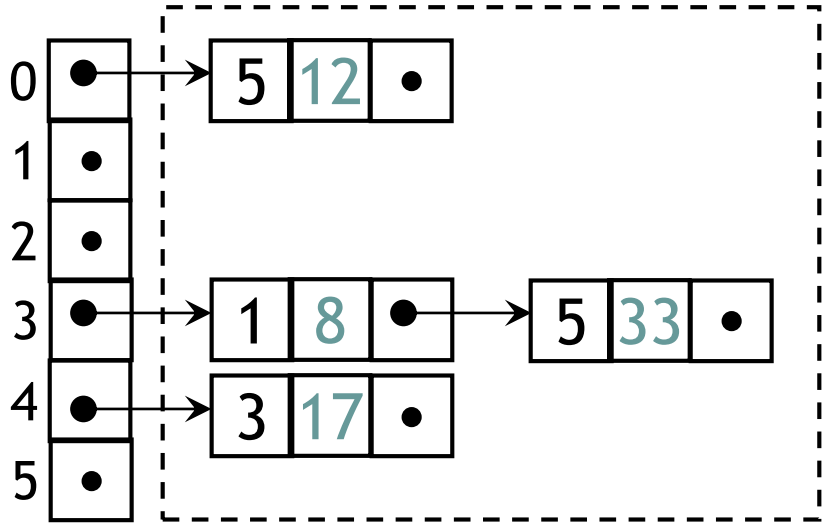
- Here is how it could be represented as a linked list:



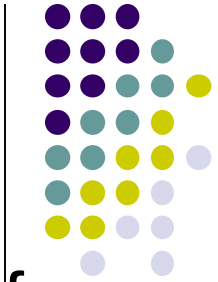
Sparse two-dimensional arrays

- Here is an example of a sparse two-dimensional array, and how it can be represented as an *array* of linked lists:





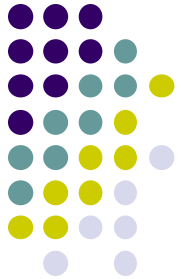
  - With this representation,
    - It is efficient to step through all the elements of a *row*
    - It is expensive to step through all the elements of a *column*
    - Clearly, we could link columns instead of rows
    - Why not both?



Big Numbers

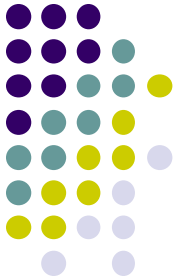
- The unsigned int type in C requires 4 bytes of memory storage. With 4 bytes we can store integers as large as 2^{32}-1; but what if we need bigger integers, for example ones having hundreds of digits?
- One way of dealing with this is to use a different storage structure for integers, such as an array of digits. If we represent an integer as an array of digits, where each digit is stored in a different array index.
- But if we want the integers to be as large as we like, the best data structure will be linked list, where each node holds one digit.

A few examples of big number



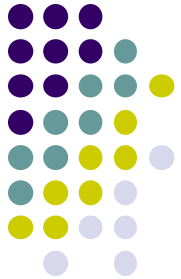
- Factorial of 10,000.

A few examples of big number



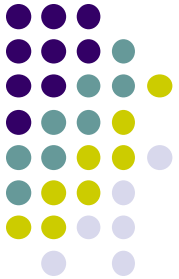
- Factorial of 10,000 is 35,659 digits long

A few examples of big number



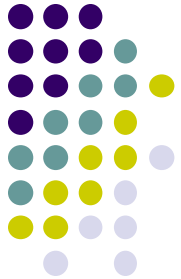
- Find the largest prime number

A few examples of big number

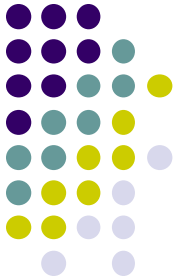


- Find the largest prime number
- As of January 2016, the **largest known prime number** is $2^{74,207,281} - 1$, a number with 2,23,38,618 digits.

Implementation of Stacks and Queues using linked lists

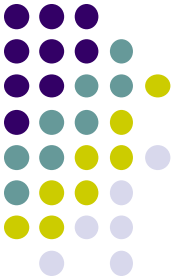


Implementation of Stacks and Queues using linked lists



```
push = insert_start(&list);
Pop = delete_start(&list);
Enqueue = insert_end(&rear);
Dequeue = delete_start(&rear);
```

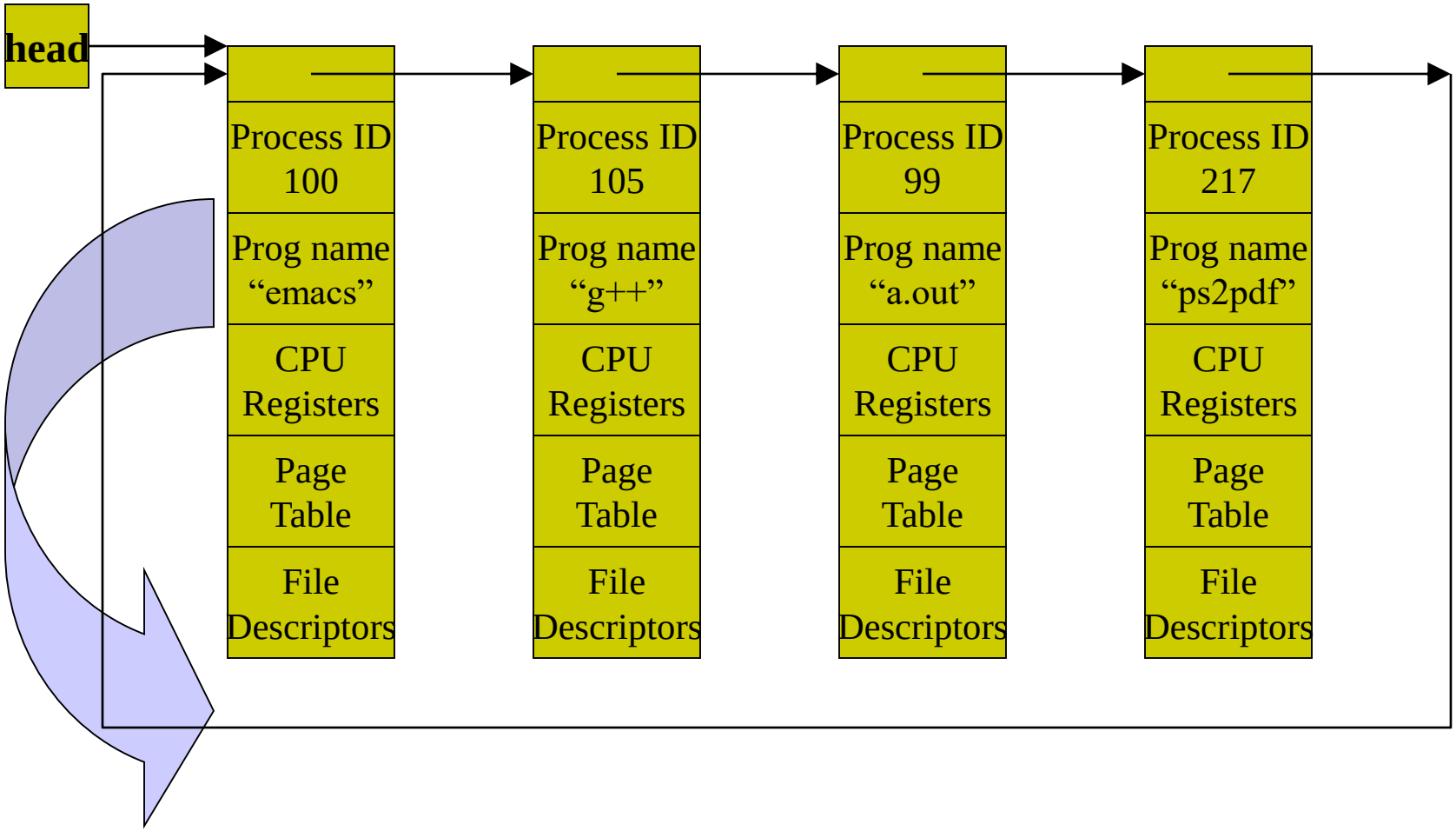
Programming Assignment



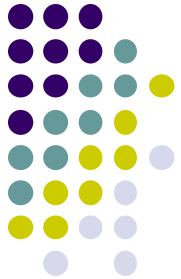
Write a program to implement add two sparse matrices using linked lists.

Circular List Application

- Round-Robin Based Job Scheduling



Josephus Problem

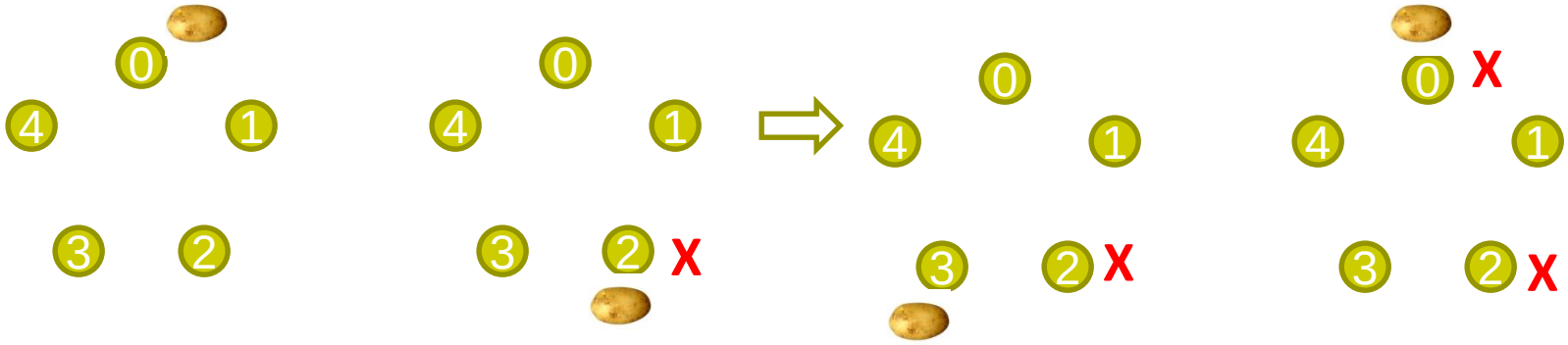


- Given a group of n men arranged in a circle under the edict that every mth man will be executed going around the circle until only one remains, find the position in which you should stand in order to be the last survivor.

Josephus Problem - Example

M=3, N=5

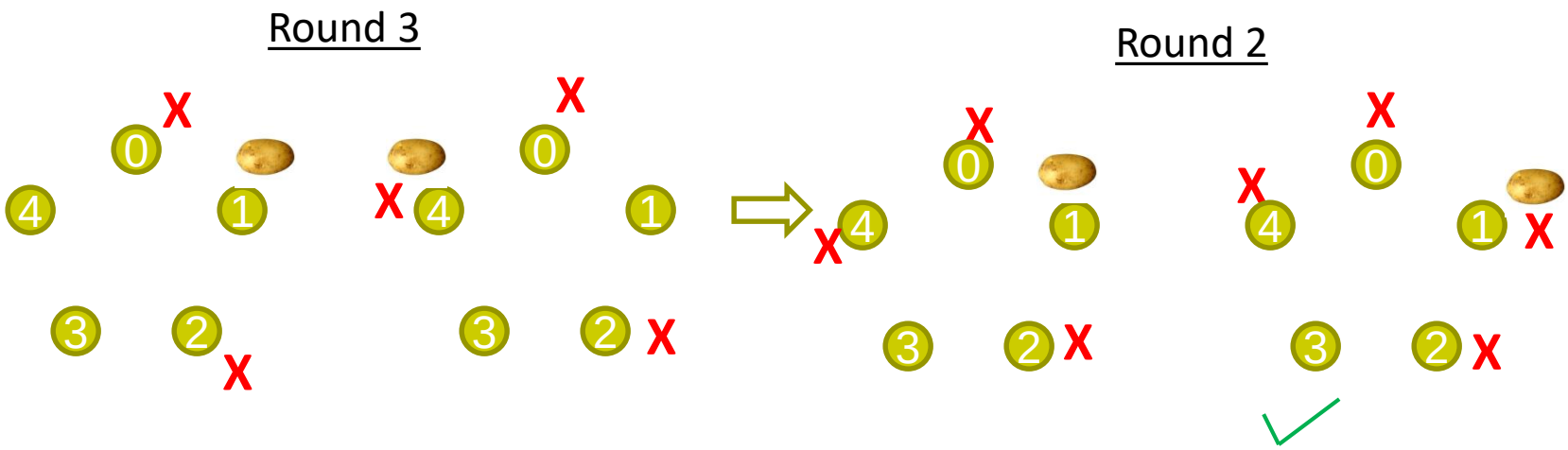
Initial state: Round 1 Round 2



Person removed so far: {2, 0,

Josephus Problem - Example

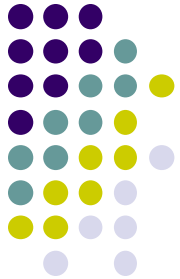
M=3, N=5



Person removed so far: {2, 0, 4, 1 }

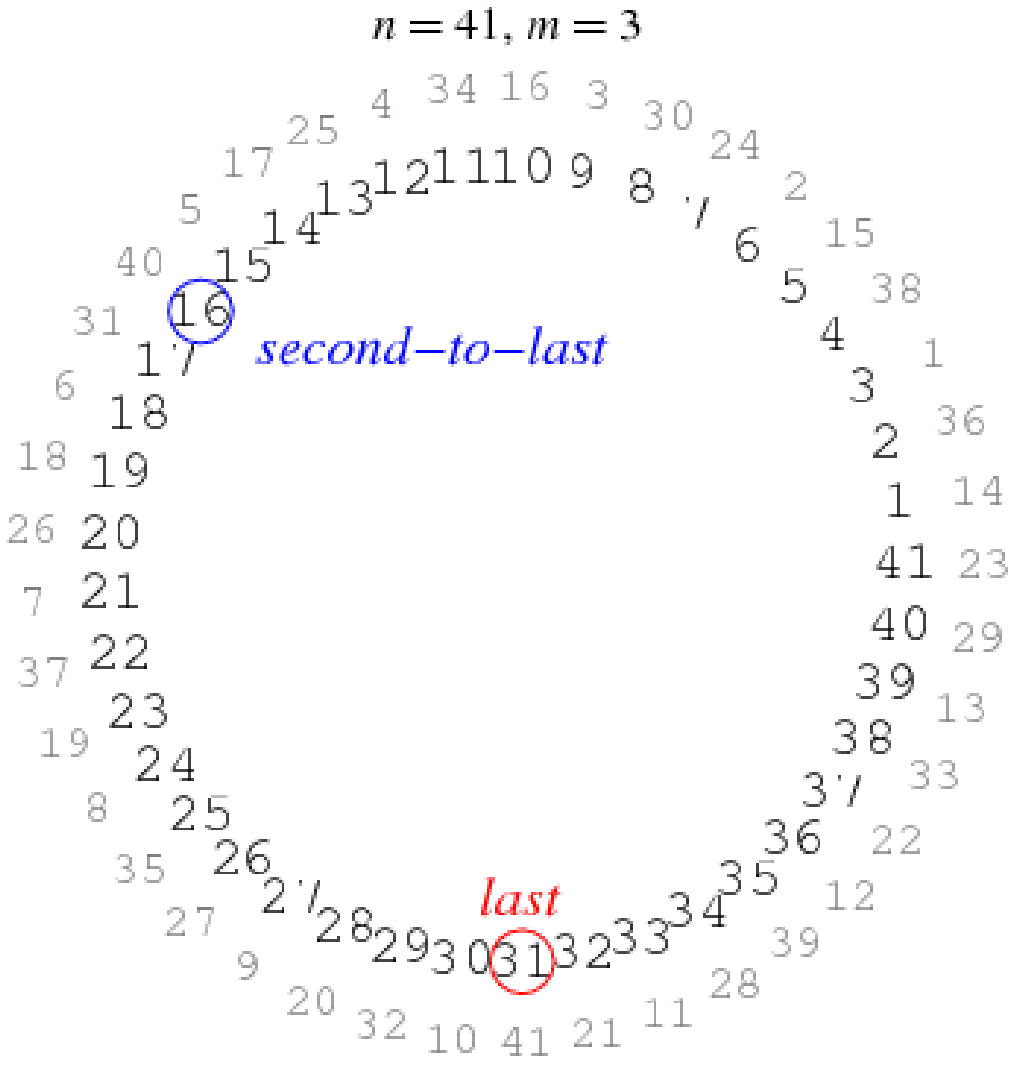
Survivor is 3

OriginalJosephus Problem

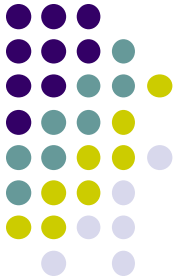


- The original Josephus problem consisted of a circle of 41 men with every third man killed.In order for the lives of the last two men to be spared, they must be placed at positions 31 (last) and 16 (second-to-last). The complete list in order of execution is 3, 6, 9, 12, 15, 18, 21, 24, 27, 30, 33, 36, 39, 1, 5, 10, 14, 19, 23, 28, 32, 37, 41, 7, 13, 20, 26, 34, 40, 8, 17, 29, 38, 11, 25, 2, 22, 4, 35, 16, 31.

Original Josephus Problem

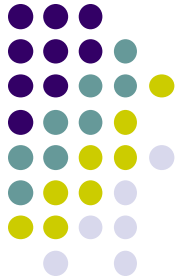


Josephus Problem - Solution



```
while(list->next!=list)
{
   for(i=0; i<count-1; i++)
      list = list->next;
   delete_after(list);
}
```

Array of linked lists

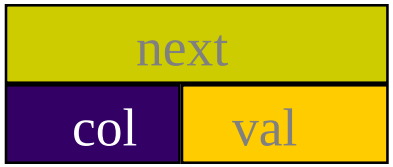


- An array of linked list combines a static structure (an array) and a dynamic structure (linked lists) to form a useful data structure. This type of a structure is appropriate for applications, where say for example, number of categories is known in advance, but how many nodes in each category is not known. For example, we can use an array (of size 26) of linked lists, where each list contains words starting with a specific letter in the alphabet.

Implement sparse matrix using array of linked lists

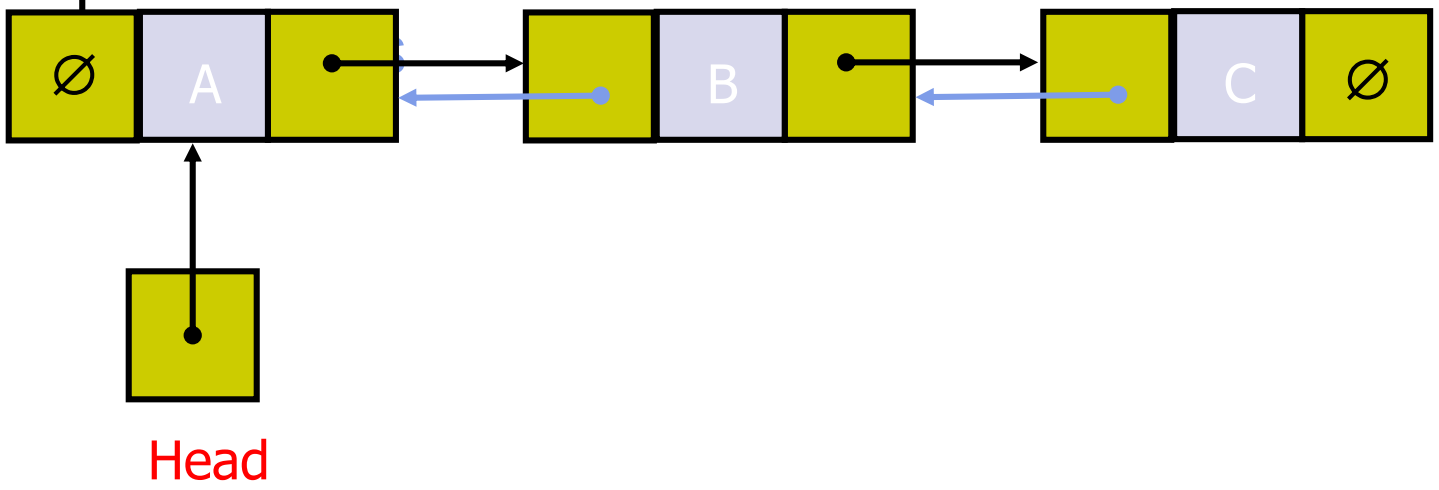
Node structure.

```
struct node
{
  int col, val;
  struct node *next;
};
typedef struct node *NODEPTR;
```

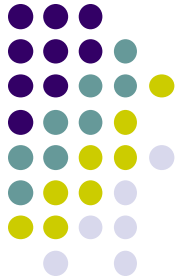


Variations of Linked Lists

- *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

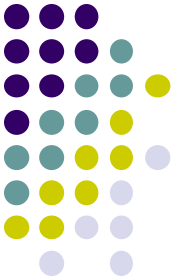


Advantages of Doubly Linked Lists



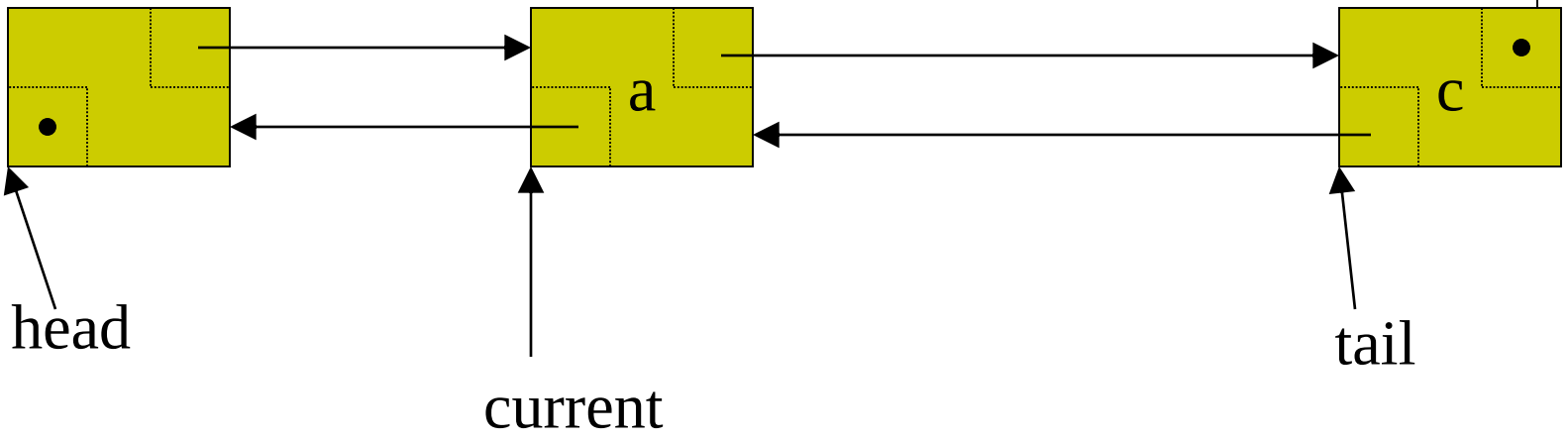
- Solves the problem of traversing backwards in an ordinary linked list.(Implementing big numbers)
- A link to the previous item as well as to the next item is maintained.
- The only disadvantage is that every time an item is inserted or deleted, two links have to be changed instead of one.

Doubly Linked List Definition



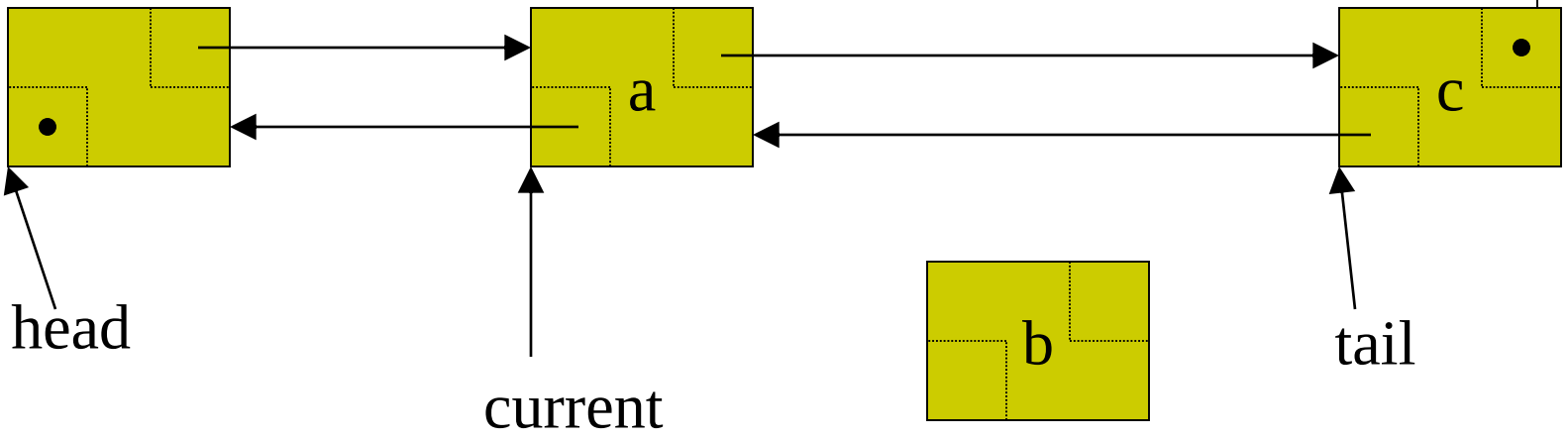
```
struct Node{
  int data;
   struct Node *next, *prev;
};
typedef struct Node *NODEPTR;
```

Inserting into a Doubly Linked List



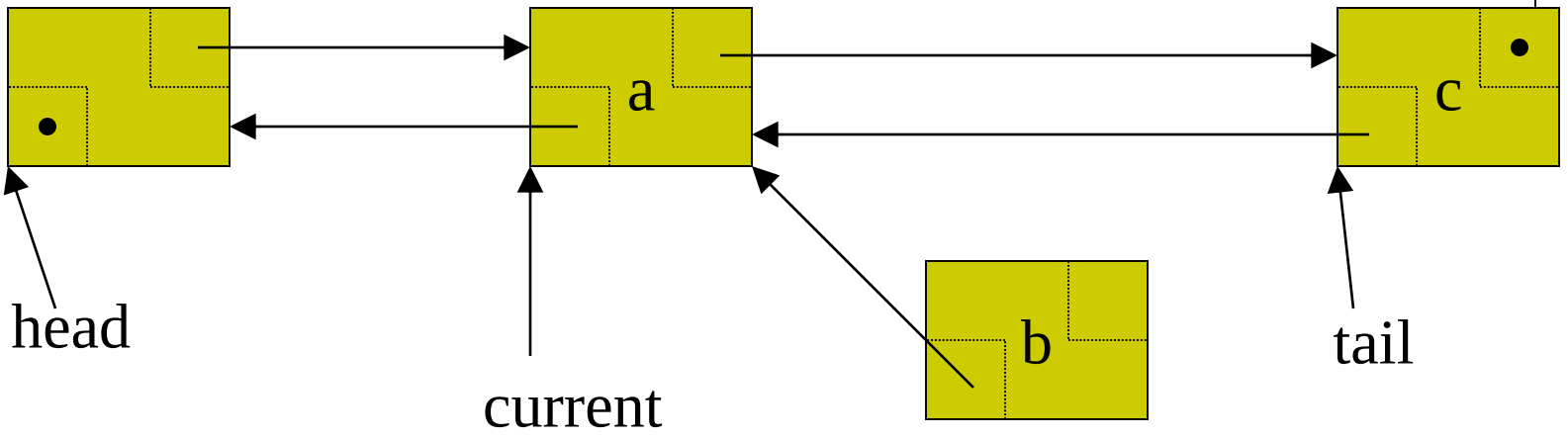

```
newNode = (NODEPTR) malloc(sizeof(struct node));
newNode->prev = current;
newNode->next = current->next;
newNode->prev->next = newNode;
newNode->next->prev = newNode;
current = newNode;
```

Inserting into a Doubly Linked List



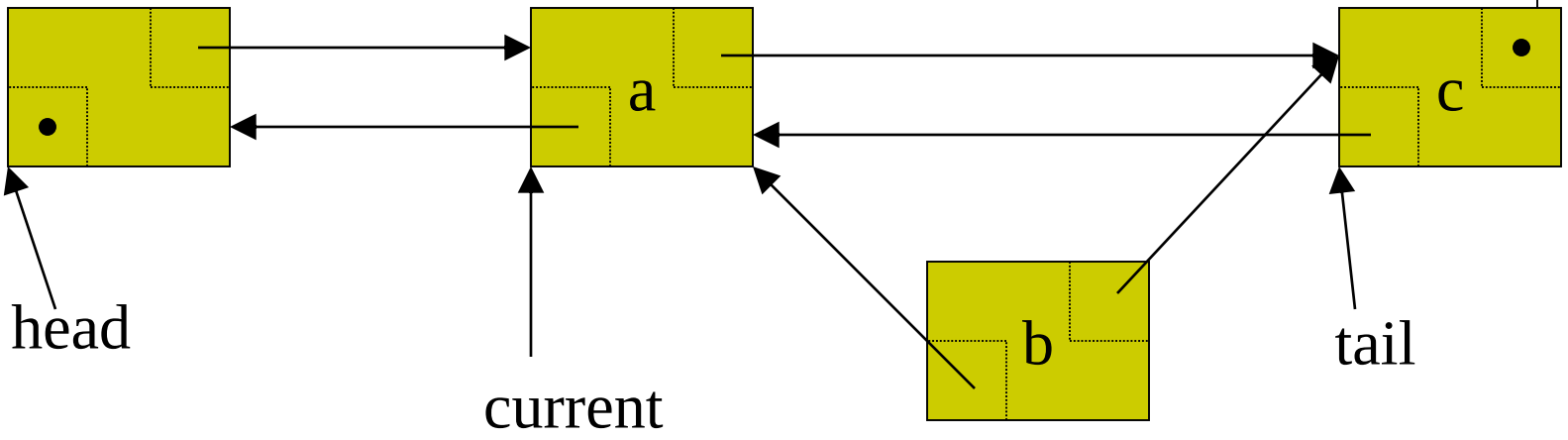

```
newNode = (NODEPTR) malloc(sizeof(struct node));
newNode->prev = current;
newNode->next = current->next;
newNode->prev->next = newNode;
newNode->next->prev = newNode;
current = newNode;
```

Inserting into a Doubly Linked List



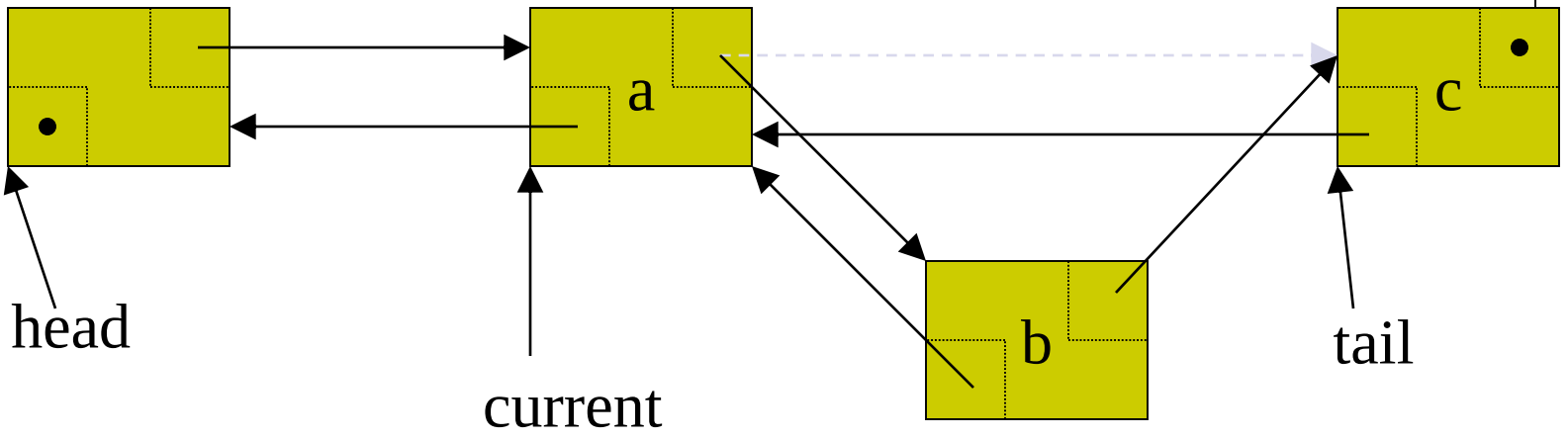

```
newNode = (NODEPTR) malloc(sizeof(struct node));
newNode->prev = current;
newNode->next = current->next;
newNode->prev->next = newNode;
newNode->next->prev = newNode;
current = newNode;
```

Inserting into a Doubly Linked List



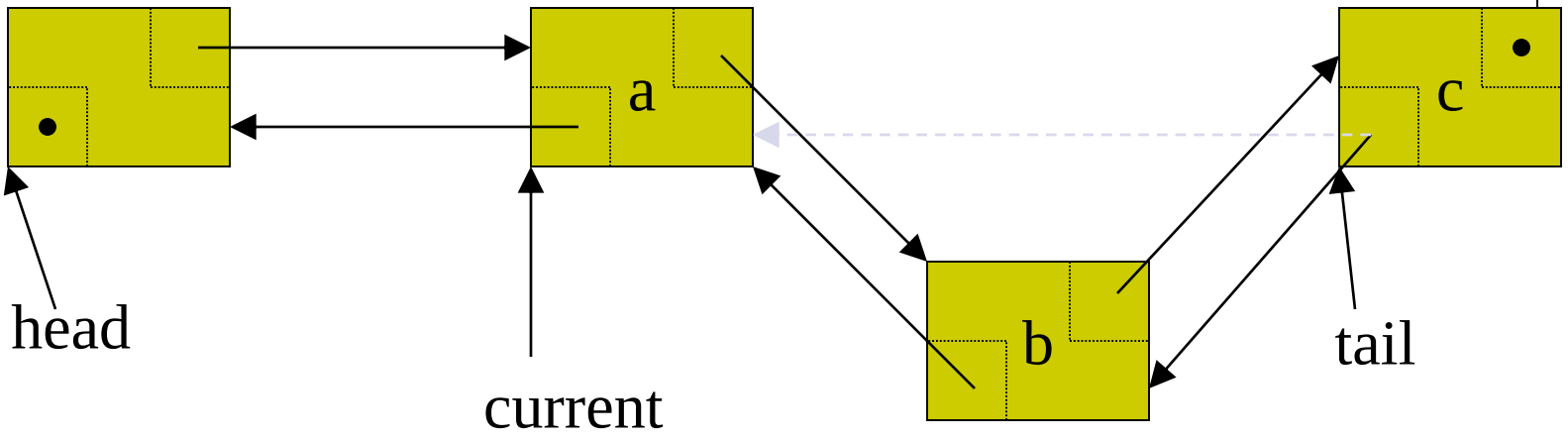

```
newNode = (NODEPTR) malloc(sizeof(struct node));
newNode->prev = current;
newNode->next = current->next;
newNode->prev->next = newNode;
newNode->next->prev = newNode;
current = newNode;
```

Inserting into a Doubly Linked List



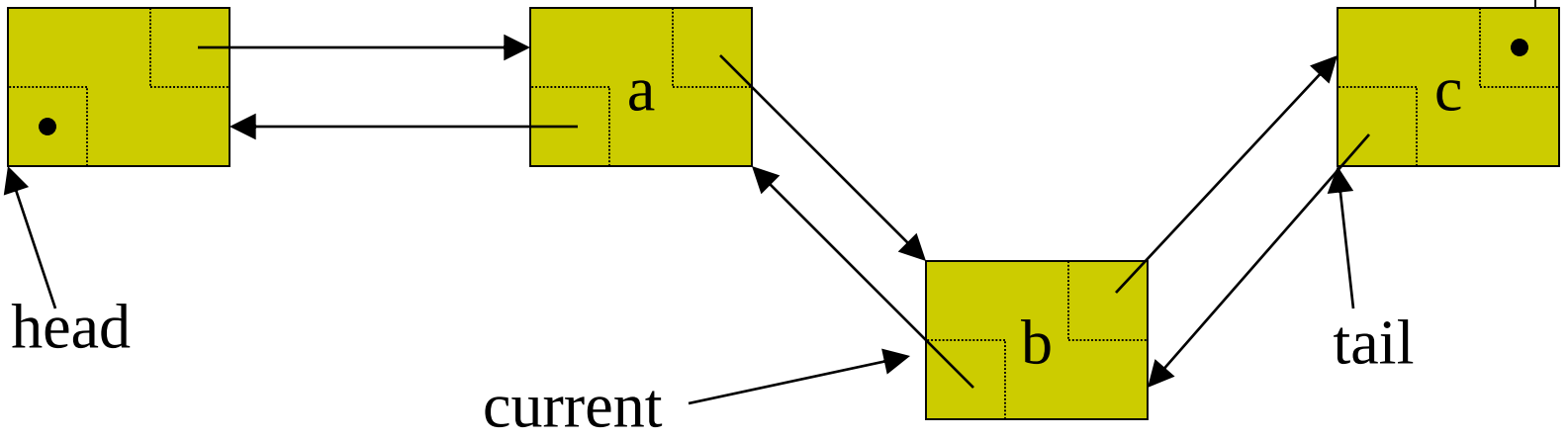

```
newNode = (NODEPTR) malloc(sizeof(struct node));
newNode->prev = current;
newNode->next = current->next;
newNode->prev->next = newNode;
newNode->next->prev = newNode;
current = newNode;
```

Inserting into a Doubly Linked List



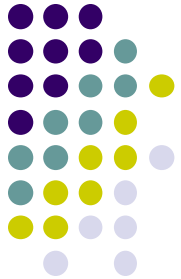

```
newNode = (NODEPTR) malloc(sizeof(struct node));
newNode->prev = current;
newNode->next = current->next;
newNode->prev->next = newNode;
newNode->next->prev = newNode;
current = newNode;
```

Inserting into a Doubly Linked List




```
newNode = (NODEPTR) malloc(sizeof(struct node));
newNode->prev = current;
newNode->next = current->next;
newNode->prev->next = newNode;
newNode->next->prev = newNode;
current = newNode;
```

Inserting into a Doubly Linked List

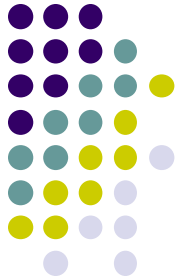


Take care of all cases

Write code for

- Insert_start(NODEPTR *head, int val)
- Insert_end(NODEPTR head, int val);

Inserting at start of doubly linked list

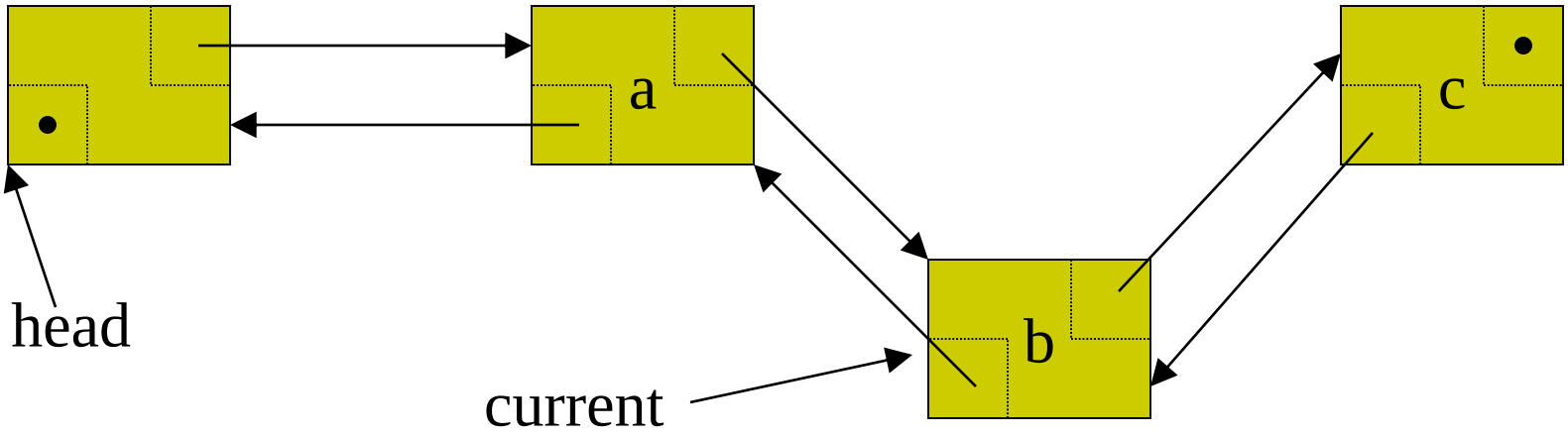


```
Insert_start(NODEPTR *head, int val)
{
  NODEPTR a = (NODEPTR) malloc(sizeof(struct node));
  a->data = val;
  if(*head==NULL)
  {
    a->prev=a->next=NULL;
    *head = a;
    return;
  }
  a->prev=NULL;
  a->next=*head;
  (*head)->prev = a;
  *head = a;
}
```

Inserting at end of doubly linked list

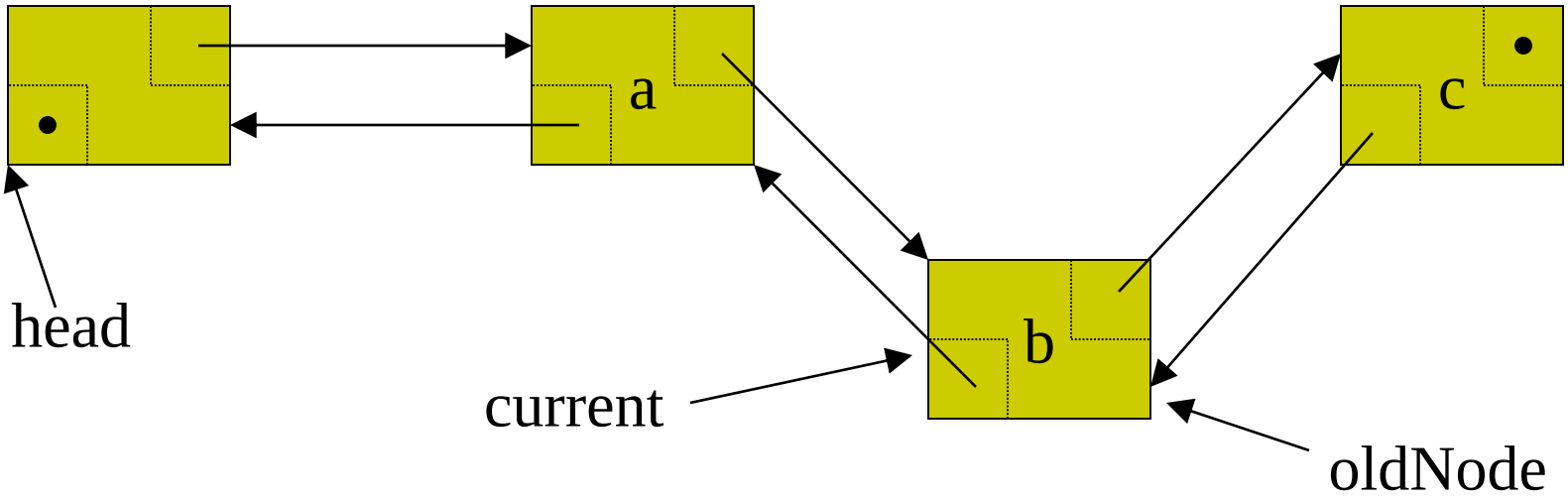
```
Insert_end(NODEPTR *head, int val)
{
  NODEPTR p=*head, a = (NODEPTR) malloc(sizeof(struct node));
  a->data = val;
  if(*head==NULL)
  {
    a->prev=a->next=NULL;
    *head = a;
    return;
  }
  while(p->next) p = p->next;
  p->next = a;
  a->prev=p;
  a->next=NULL;
}
```

Deleting an element from a double linked list



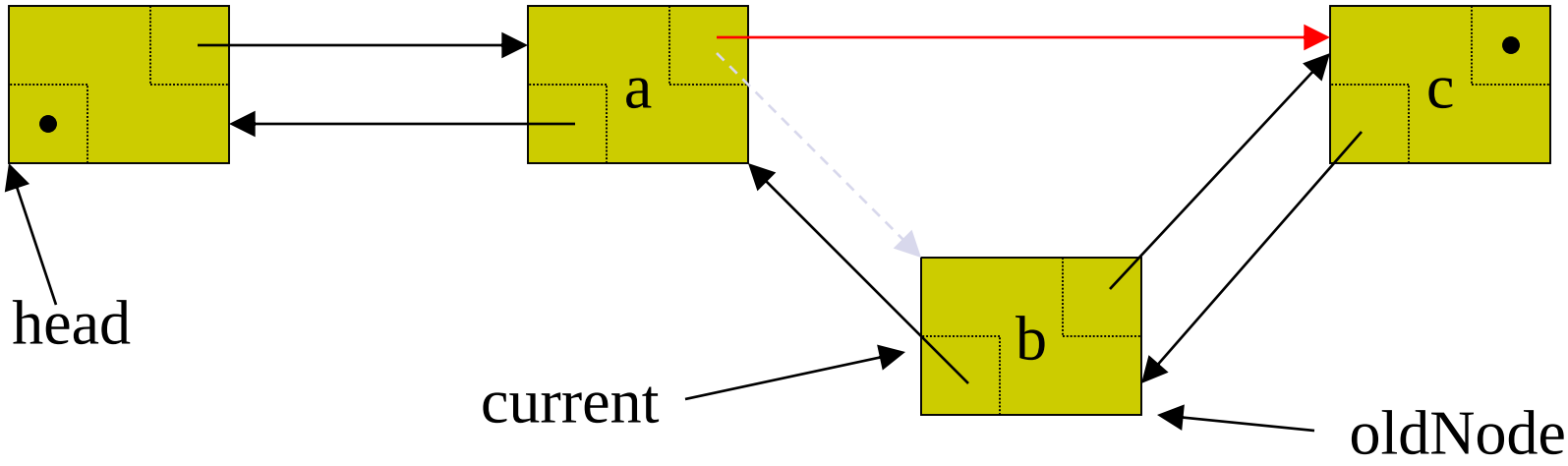

```
oldNode=current;
oldNode->prev->next = oldNode->next;
oldNode->next->prev = oldNode->prev;
current = oldNode->prev;
free(oldNode);
```

Deleting an element from a double linked list



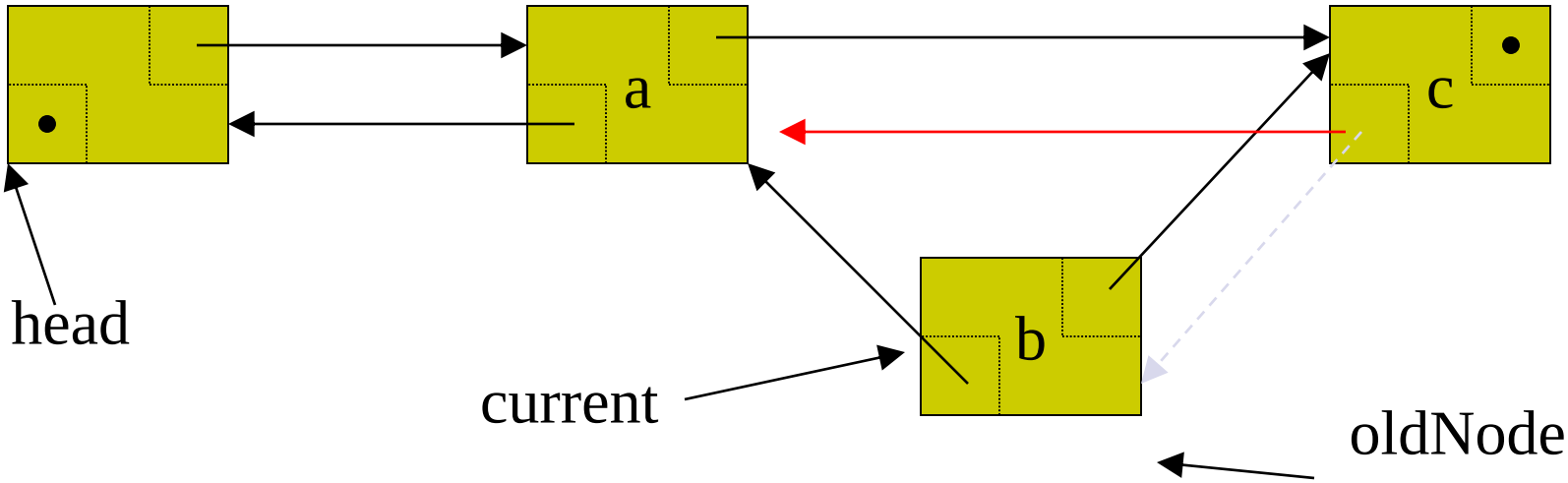

```
oldNode=current;
oldNode->prev->next = oldNode->next;
oldNode->next->prev = oldNode->prev;
current = oldNode->prev;
free(oldNode);
```

Deleting an element from a double linked list



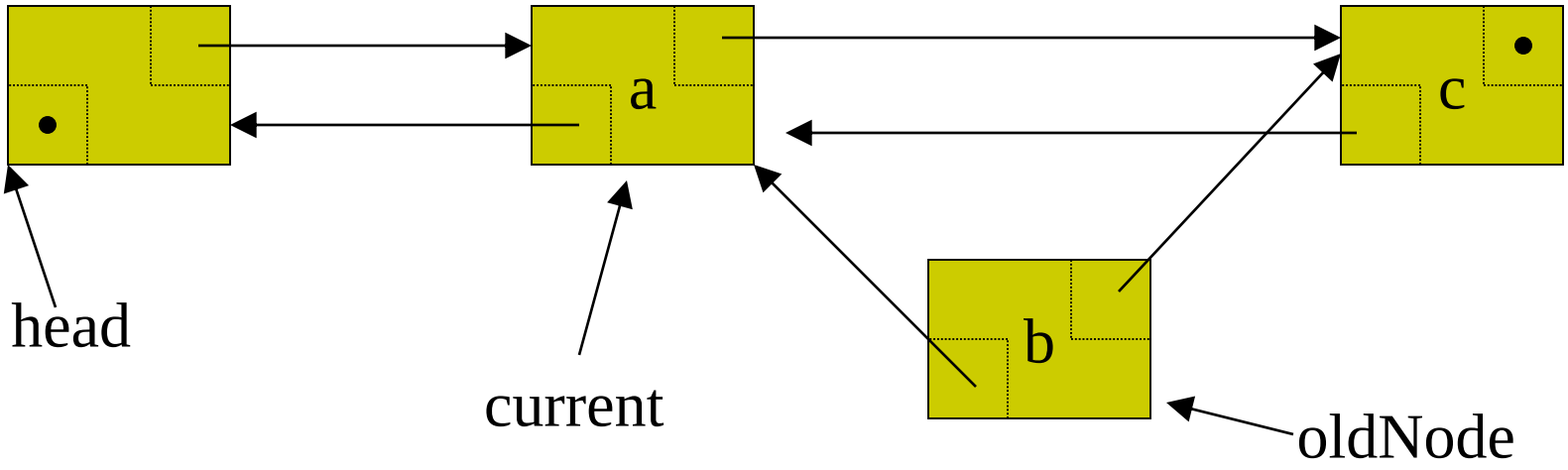

```
oldNode=current;
oldNode->prev->next = oldNode->next;
oldNode->next->prev = oldNode->prev;
current = oldNode->prev;
free(oldNode);
```

Deleting an element from a double linked list



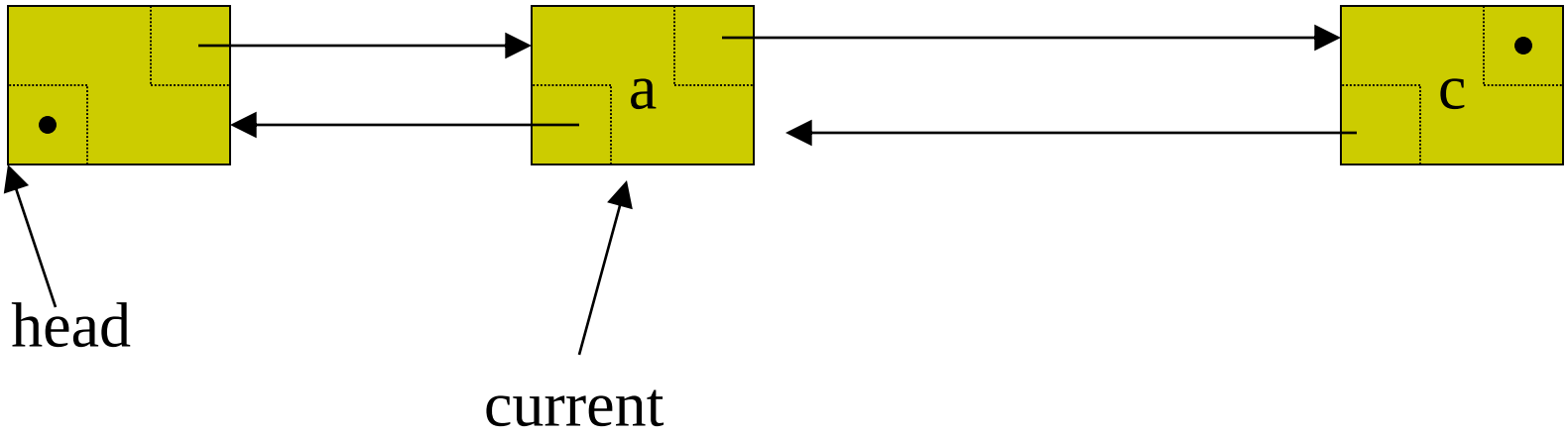

```
oldNode=current;
oldNode->prev->next = oldNode->next;
oldNode->next->prev = oldNode->prev;
current = oldNode->prev;
free(oldNode);
```

Deleting an element from a double linked list



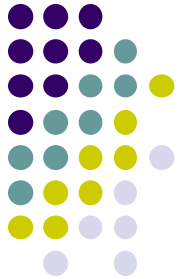

```
oldNode=current;
oldNode->prev->next = oldNode->next;
oldNode->next->prev = oldNode->prev;
current = oldNode->prev;
free(oldNode);
```

Deleting an element from a double linked list




```
oldNode=current;
oldNode->prev->next = oldNode->next;
oldNode->next->prev = oldNode->prev;
current = oldNode->prev;
free(oldNode);
```

Deleting an element from a double linked list

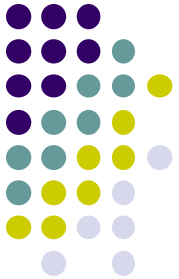


- Write down functions to delete first and last elements in doubly linked list
  - delete_first(NODEPTR *head)
  - delete_last(NODEPTR *head)

Deleting an element from a double linked list

```
int delete_start(NODEPTR *p)
{
  int i;
  NODEPTR q;
  if(*p==NULL)return 0;
  i = (*p)->data;
  q = (*p)->next;
  if(q)
    q->prev = NULL;
  free(*p);
  *p = q;
  printf("\nValue deleted =%d\n", i);
  return i;
}
```

Doubly Linked List Application



- Line-based text editors like Unix ed