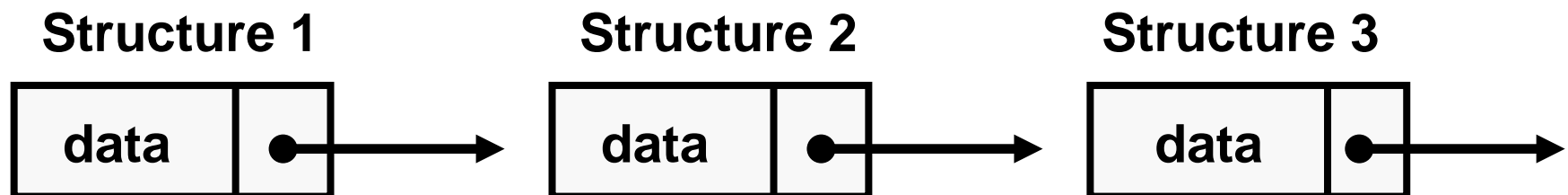# Linked List

# List

- A list refers to a sequence of data items
    - Example: An array
        - The array index is used for accessing and manipulation of array elements
    - Problems with arrays
        - The array size has to be specified at the beginning (at least during dynamic allocation)
            - realloc can be used to readjust size in middle, but contiguous chunk of memory may not be available
        - Deleting an element or inserting an element may require shifting of elements
        - Wasteful of space

# Linked List

- A completely different way to represent a list
  - □ Make each data in the list part of a structure
  - □ The structure also contains a pointer or link to the structure (of the same type) containing the next data
  - □ This type of list is called a linked list

**Structure 1**    **Structure 2**    **Structure 3**

| data | ● | → | data | ● | → | data | ● | → |

# Forming a linked list

- Let each structure of the list (lets call it node) have two fields:
  - □ One containing the data
  - □ The other containing the address of the structure holding the next data in the list
- The structures in the linked list need not be contiguous in memory
  - □ They are ordered by logical links that are stored as part of the data in the structure itself
  - □ The link is a pointer to another structure of the same type
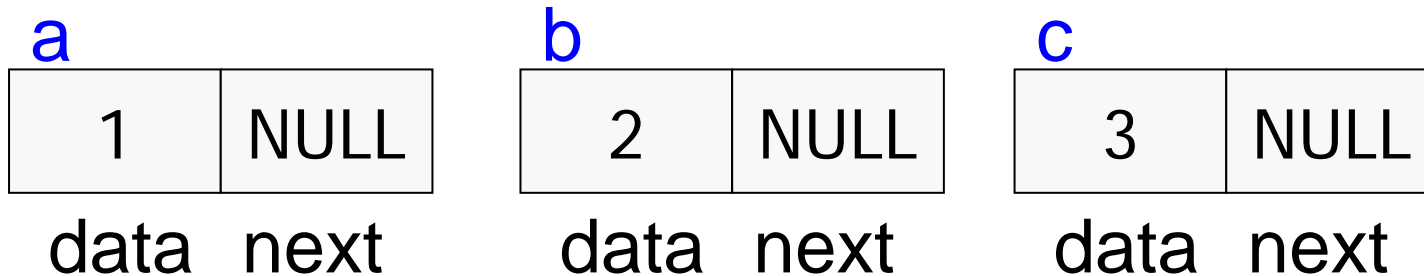
# Contd.

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

**node**

| data | next |
| --- | --- |
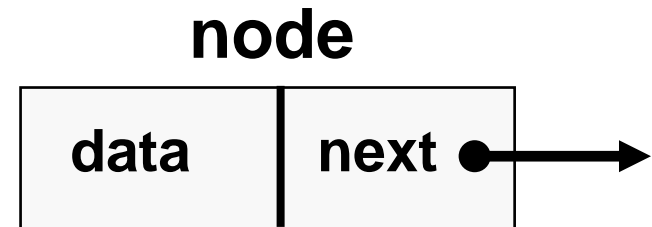
- The pointer variable next contains either the address of the location in memory of the successor list element or the special value NULL defined as 0

  - NULL is used to denote the end of the list (no successor element)

- Such structures which contain a member field pointing to the same structure type are called self-referential structures
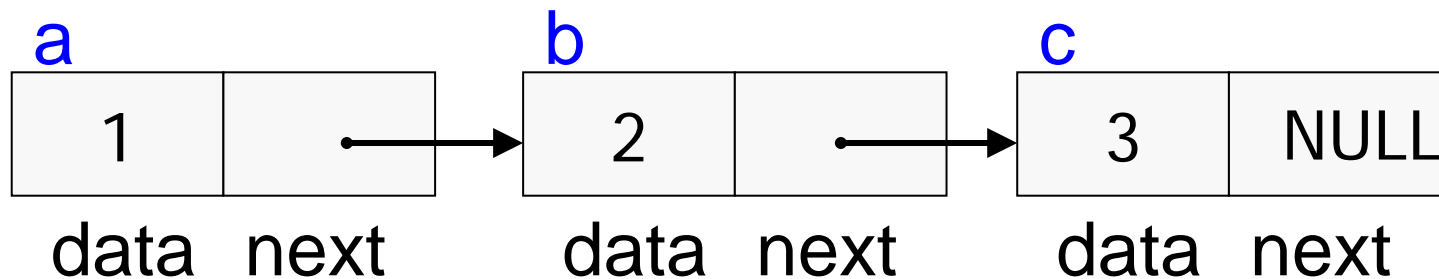
5

# Example: nodes of the list

struct node a, b, c;

a.data = 1;

b.data = 2;

c.data = 3;

a.next = b.next = c.next = NULL;

| a | | | b | | | c | | |
|---|---|---|---|---|---|---|---|---|
| 1 | NULL | | 2 | NULL | | 3 | NULL | |
| data | next | | data | next | | data | next | |

# Chaining these together

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



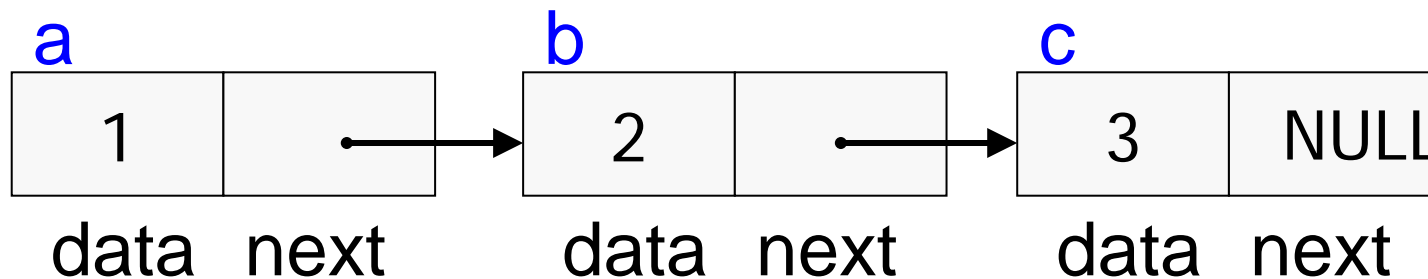| a | | | b | | | c | | |
|---|---|---|---|---|---|---|---|---|
| 1 | | → | 2 | | → | 3 | NULL | |
| data | next | | data | next | | data | next | |

What are the values of :
- a.next->data
- a.next->next->data

# Chaining these together

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

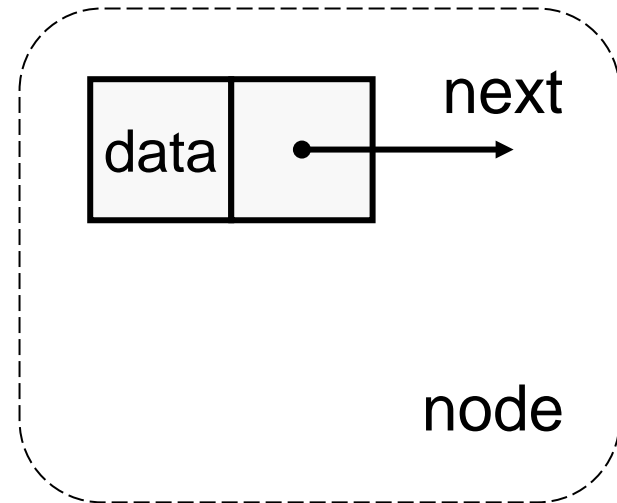| a | | b | | c | |
|---|---|---|---|---|---|
| 1 | → | 2 | → | 3 | NULL |
| data | next | data | next | data | next |

What are the values of :
- a.next->data    **2**
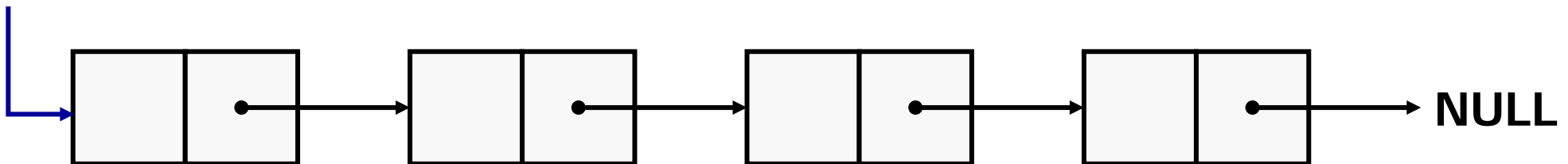- a.next->next->data    **3**

# Linked Lists

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

- Each node stores
  - data
  - link to the next node

# Contd.

- 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

**head**

# Contd.

■ In general, a node of the linked list may be represented as

**Name of the type of nodes**

```
struct node_name
  {
     type  member1;
     type member2;

     ..........
     struct node_name *next;
  };
```

**Data items in each element of the list**

**Link to the next element in the list**

# Example: list of student records

- Structure for each node

```
struct stud
  {
      int  roll;
      char name[30];
      int  age;
      struct stud *next;
  };
```

- Suppose the list has three students' records
- Declare three nodes n1, n2, and n3

```
struct stud n1, n2, n3;
```
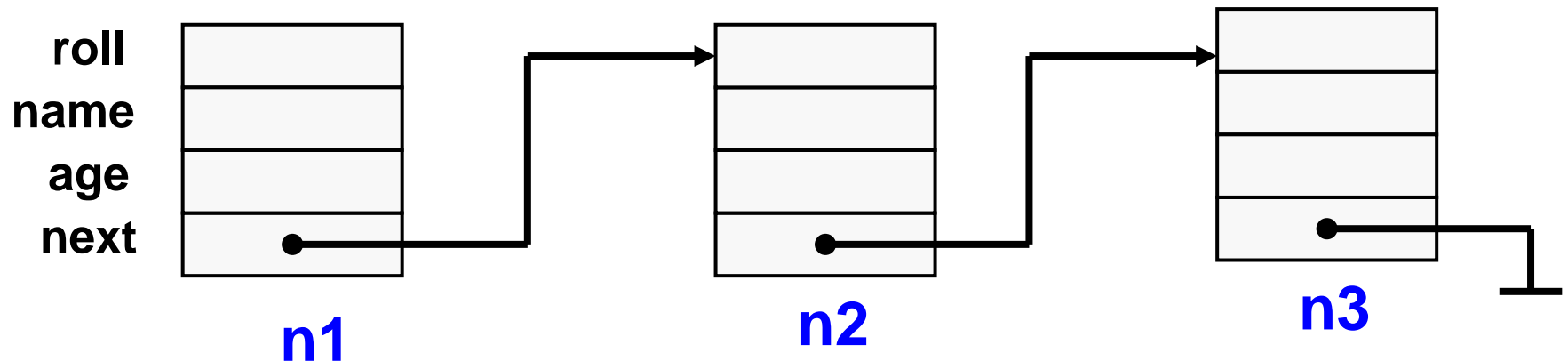
# Contd.

- Create the links between the nodes

  n1.next =  &n2 ;

  n2.next =  &n3 ;

  n3.next =  NULL ;   /* No more nodes follow */

- The final list looks like

# Code for the Example

```c
#include  <stdio.h>
struct  stud
  {
      int  roll;
      char  name[30];
      int  age;
      struct  stud  *next;
  };

int main()
{
    struct  stud  n1, n2, n3;
    struct  stud  *p;
    scanf ("%d %s %d", &n1.roll, n1.name, &n1.age);
    scanf ("%d %s %d", &n2.roll, n2.name, &n2.age);
    scanf ("%d %s %d", &n3.roll, n3.name, &n3.age);
```

```c
    n1.next   =   &n2 ;
    n2.next   =   &n3 ;
    n3.next   =   NULL ;


  /* Now traverse the list and print the
elements */


   p   =   &n1 ;    /* point to 1st element */
   while  (p != NULL)
   {
       printf ("\n %d %s %d",
       p->roll, p->name, p->age);
       p   =   p->next;
   }
   return 0;
}
```
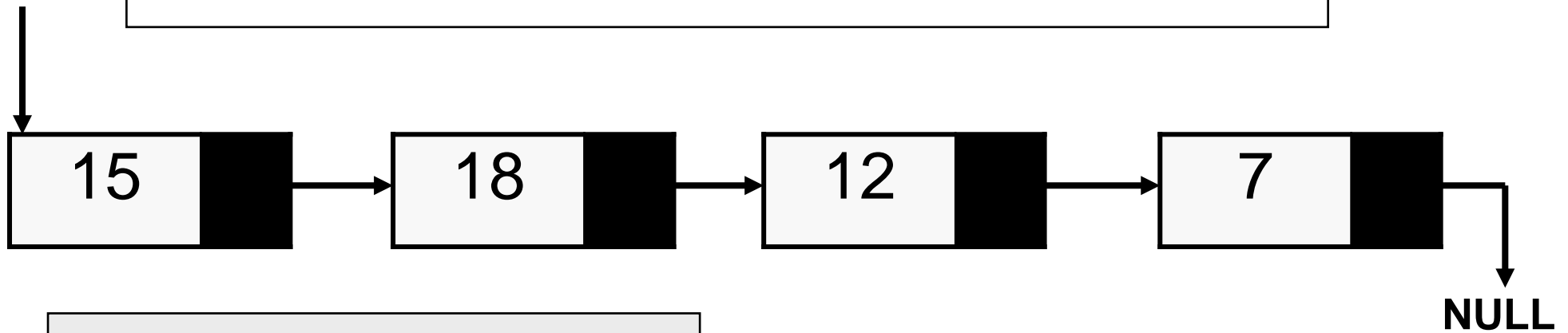
# Alternative Way

- Instead of statically declaring the structures n1, n2, n3,
    - ☐ Dynamically allocate space for the nodes
    - ☐ Use malloc individually for every node allocated
- This is the usual way to work with linked lists, as number of elements in the list is usually not known in advance (if known, we could have used arrays)
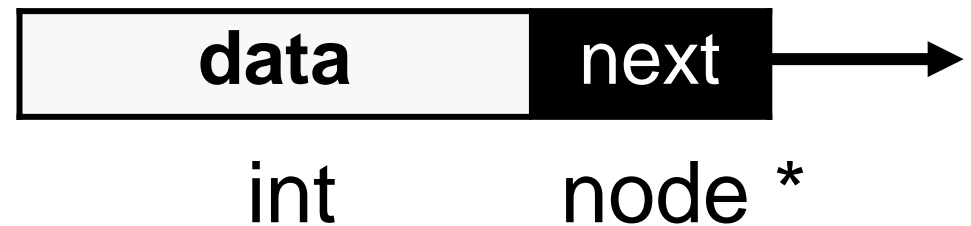- See examples next

# Example of dynamic node allocation

Storing a set of elements = {15,18,12,7}

```
struct node {
    int data ;
    struct node * next ;
} ;
struct node *p, *q;
```
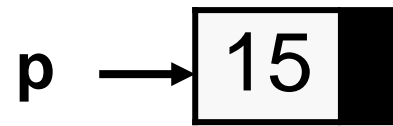
| data | next |
|------|------|
| int | node * |

# Adding 15 and 18 only

p = (struct node *) malloc(sizeof(struct node));

p->data=15;

p → [ 15 ■ ]

```
struct node {
    int data ;
    struct node * next ;
} ;
struct node *p, *q;
```

q = (struct node *) malloc(sizeof(struct node));

q->data=18;  q->next = NULL;
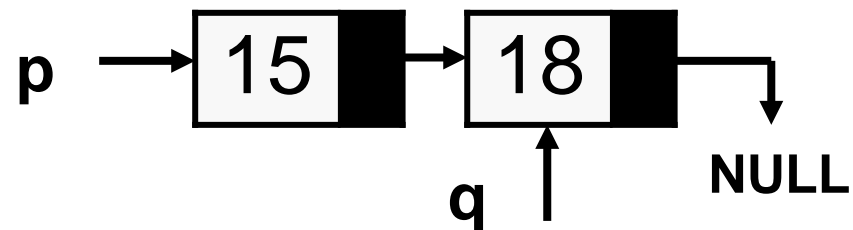
p → [ 15 ■ ]    [ 18 ■ ]→

NULL

q

p->next = q;

p → [ 15 ■ ]→[ 18 ■ ]→
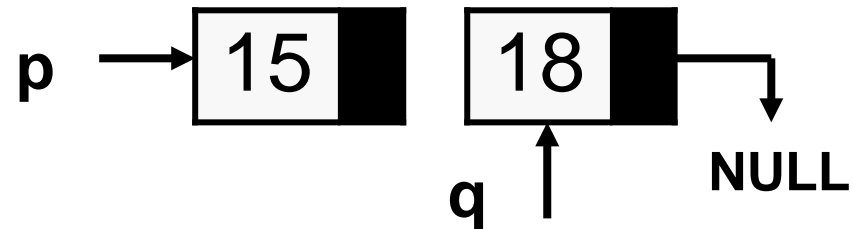
NULL

q

# Traversing the elements added

```
struct node {
      int data;
      struct node * next;
 };
int main()  {
    struct node *p,*q,*r;
    p = (struct node *) malloc(sizeof(struct node));
      :
    r=p;
    while(r!=NULL){
        printf("Data = %d \n",r->data);
        r=r->next;
    }
  return 0;
}
```

**Output**

Data = 15
Data = 18

**We could have done anything else other than printing as we traverse each element, like searching for example. Just like traversing an array.**

# Contd.

- We assumed two elements in the list, so took two pointers p and q

- What if the number of elements are not known?

  - ☐ Precisely the reason we use linked list

- Solution:

  - ☐ Remember the address of the first element in a special pointer (the head pointer), make sure to not overwrite it

  - ☐ Any other pointer can be reused

# Example: adding n elements read from keyboard

```c
int main()  {
   int n, i;
   struct node *head = NULL, *p, *prev;
   scanf("%d", &n);
   for (i = 0; i < n; ++i) {
       p = (struct node *) malloc(sizeof(struct node));
       scanf("%d", &p->data);
       p->next = NULL;
       if (head == NULL) head = p;
       else prev->next = p;
       prev = p;
    }
   return 0;
}
```

**head changes only once, when the first element is added**

**prev remembers the pointer to the last element added, p is linked to its next field**

**p and prev are reused as many times as needed**

# Example: printing an arbitrary sized list

```
int main()
{
    int n, i;
    struct node *head = NULL, *p;

        :
    p = head;
    while (p != NULL) {
        printf("%d   ", p->data);
        p = p->next;
    }
    return 0;
}
```

**Assumed that the list is already created and head points to the first element in the list**

**p is reused to point to the elements in the list (initially, to the first element)**

**When p points to the last element, p->next = NULL, so the loop terminates after this iteration**

# Important to remember

- Store the address of the first element added in a separate pointer (head in our examples), and make sure not to change it

  - If you lose the start pointer, you cannot access any element in the list, as elements are only accessible from the next pointers in the previous element

  - In the print example, we could have reused head, (head=head->next instead of p=p->next) as we do not need to remember the start pointer after printing the list, but this is considered bad practice, so we used a separate temporary pointer p

# Function to print a list

The pointer to the start of the list (head pointer) is passed as parameter

```c
void display (struct node *r)
{
    struct node *p = r;
    printf("List = {");
    while(p != NULL) {
        printf("%d, ", p->data);
        p = p->next;
    }
    printf("}\n");
}
```

# Common Operations on Linked Lists

- Creating a linked list (already seen)
- Printing a linked list (already seen)
- Search for an element in a linked list (can be easily done by traversing)
- Inserting an element in a linked list
  - Insert at front of list
  - Insert at end of list
  - Insert in sorted order
- Delete an element from a linked list

# Search for an element

Takes the head pointer as parameter

Traverses the list and compares value with each data

Returns the node with the value if found, NULL otherwise

```c
struct node *search (struct node *r, int value)
{
    struct node *p;
     p = r;
     while (p!=NULL){
          if (p->data == value) return p;
          p = p->next;
      }
     return p;
}
```

# Insertion in a list

- To insert a data item into a linked list involves
  - creating a **new** node containing the data
  - finding the correct place in the list, and
  - linking in the new node at this place
- Correct place may vary depending on what is needed
  - Front of list
  - End of list
  - Keep the list in sorted order
  - …

# Insert in front of list

Takes the head pointer and the value to be inserted (NULL if list is empty)
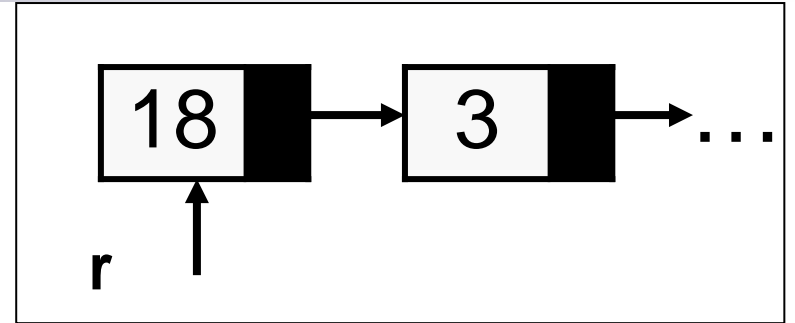
Inserts the value as the first element of the list

Returns the new head pointer value

```
struct node *insert(struct node *r, int value)
{
    struct node *p;
    p = (struct node *) malloc(sizeof(struct node));
    p->data = value;
    p ->next = r;
    return p;
}
```

# Contd.
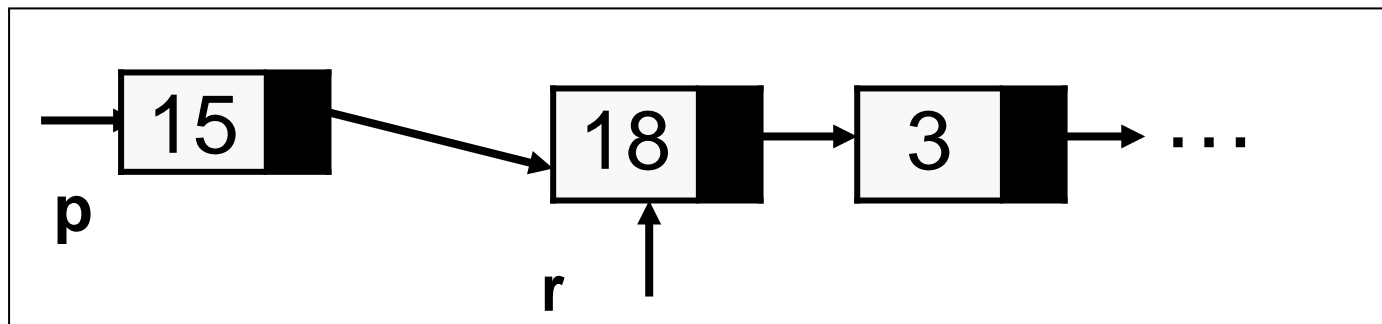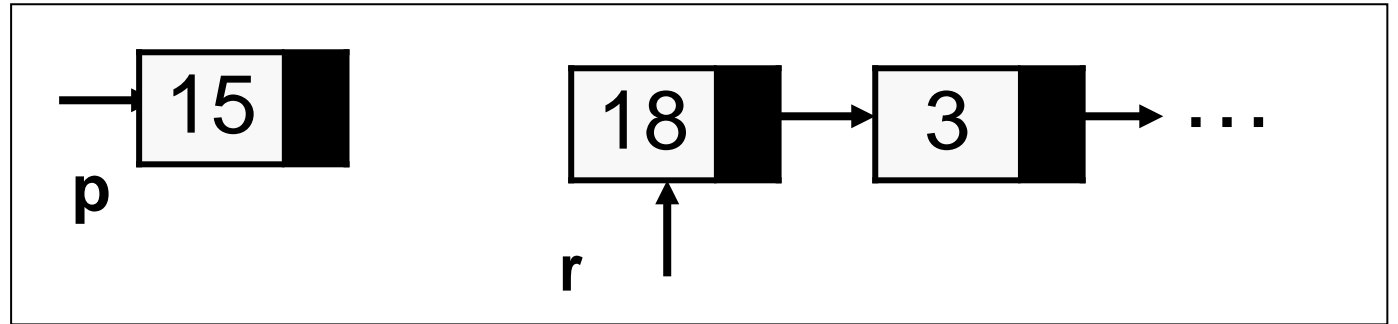
# Using the Insert Function

```
void display (struct node *);
struct node * insert(struct node * , int);
int main()
{    struct node *head;
     head = NULL;
     head = insert(head, 10);
     display(head);
     head = insert(head, 11);
    display(head);
    head = insert(head, 12);
    display(head);
    return 0;

}
```

**Output**

List = {10, }
List = {11, 10, }
List = {12, 11, 10, }

# Insert at end

Takes the head pointer and the value to be inserted (NULL if list is empty)

Inserts the value as the last element of the list

Returns the new head pointer value

```
struct node *insert_end(struct node *r,
                                int value)
{  struct node *p,*q;
   p = (struct node *) malloc(sizeof(struct node));
   p->data = value;
   p ->next = NULL;
   if (r==NULL) return p;  /* list passed is empty */
   q=r;
   while (q->next!=NULL)
       q=q->next;     /* find the last element */
   q->next =p;
   return r;
}
```

# Contd.
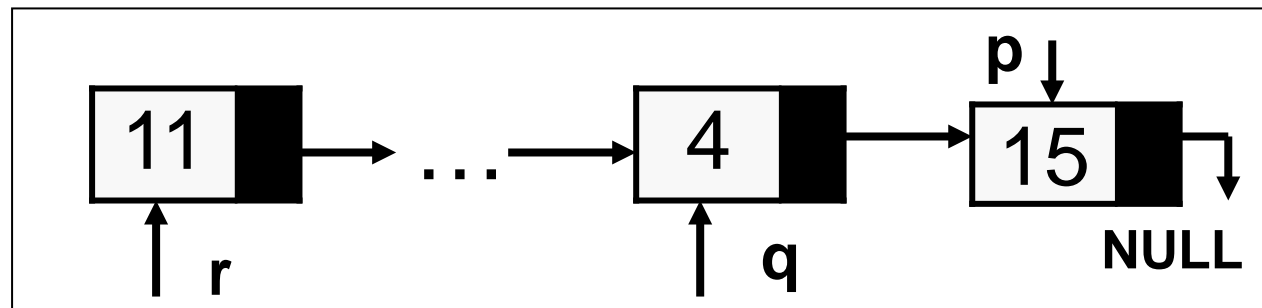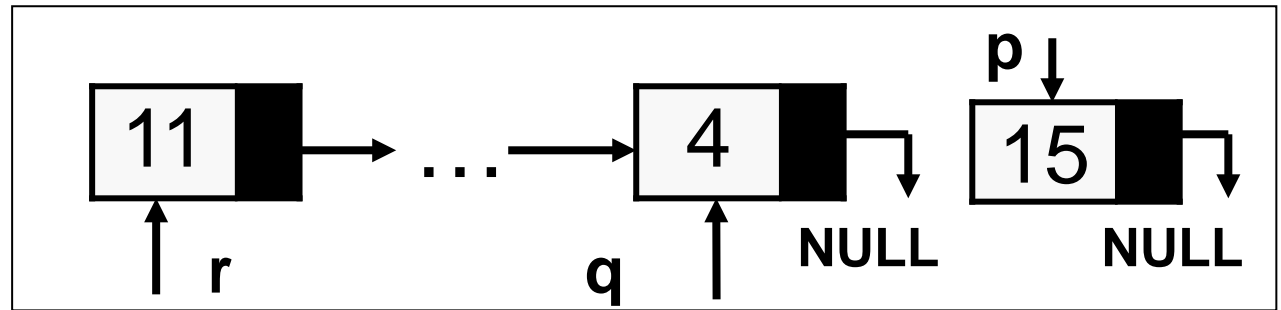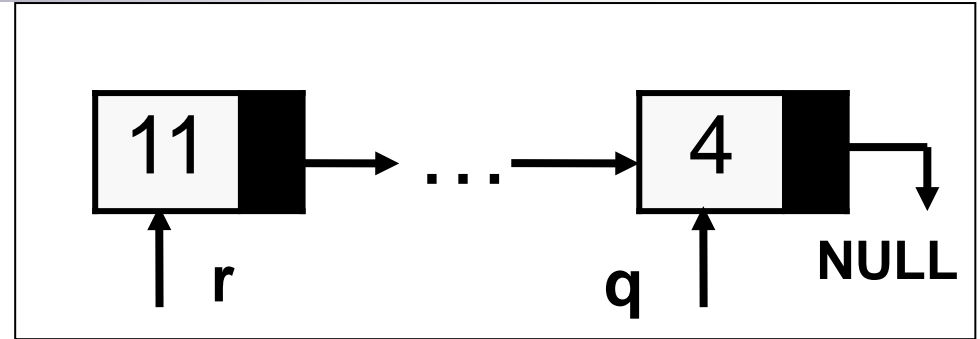
# Using the Insert at End Function

```
void display (struct node *);
struct node * insert(struct node * , int);
struct node * insert_end(struct node * , int);
int main()
{
    struct node *start;
    start = NULL;
    start = insert_end(start, 10);
    display(start);
    start = insert_end(start, 11);
    display(start);
    start = insert_end(start, 12);
    display(start);
    return 0;
}
```
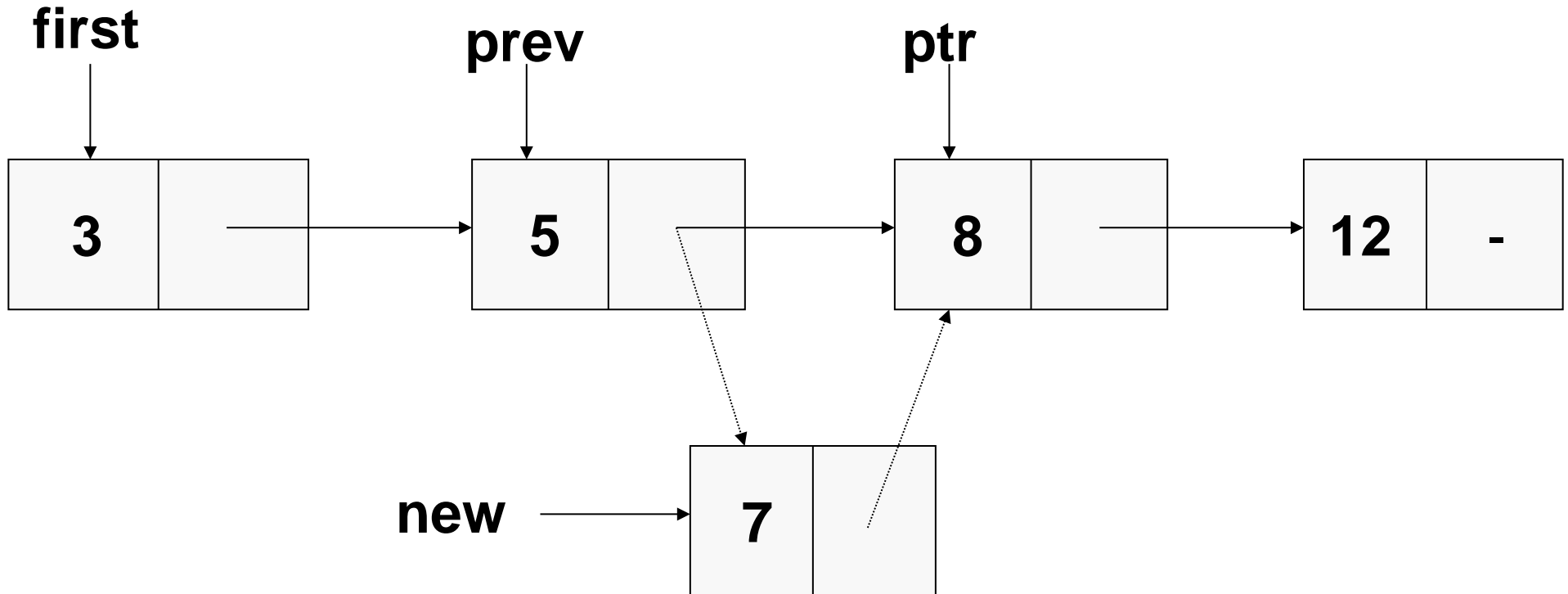
**Output**

List = {10, }
List = {10, 11, }
List = {10, 11, 12, }

# Insertion in Ascending Order

first

prev

ptr

| 3 | | → | 5 | | → | 8 | | → | 12 | - |

new → 7

- Create new node for the 7
- Find correct place – when ptr finds the 8 (7 < 8)
- Link in new node with previous (even if last) and ptr nodes
- Also check insertion before first node!

# Insert in ascending order & sort

```c
struct node * insert_asc(struct node * r, int value)
{ struct node *p, *q, *new;
  new = (struct node *) malloc(sizeof(struct node));
  new->data = value;  new ->next = NULL;
  p = r;  q = p;
  while(p!=NULL) {
    if (p->data >= value) { /* insert before */
      if (p==r) {  new->next =r; /* insert at start */
                    return new;  }
      new->next = p; /* insert before p */
      q->next = new;
      return r; }
    q = p;
    p = p->next; } /* exists loop if > largest */
  if (r==NULL) return new; /* first time */
  else q->next = new;  /* insert at end */
  return r; }
```

```c
int main()
{
    struct node *start;
    int i,n,value;
    start = NULL;
    scanf("%d",&n);
    for(i=0; i<n; i++) {
        printf("Give Data: " );
        scanf("%d",&value);
        start = insert_asc(start, value);
    }
    display(start);
    return 0;
}
```

35

# Deletion from a list

- **To delete a data item from a linked list**
  - ☐ Find the data item in the list, and if found
  - ☐ Delink this node from the list
  - ☐ Free up the malloc'ed node space

# Example of Deletion
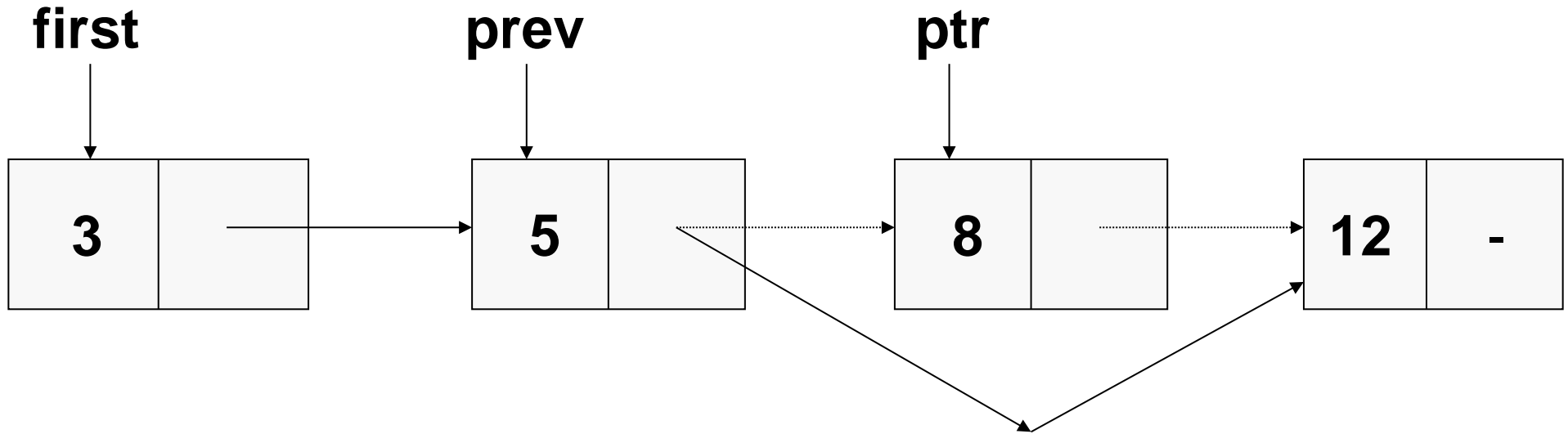


- When ptr finds the item to be deleted, e.g. 8, we need the previous node to make the link to the next one after ptr (i.e. ptr -> next)
- Also check whether first node is to be deleted

# Deleting an element

```
struct node * delete(struct node * r, int value)
{   struct node *p, *q;
    p = r;
    q = p;
    while(p!=NULL) {
        if (p->data == value){
            if (p==r) r = p->next;
            else  q->next = p->next;
            p->next = NULL;
            free(p);
            return r;
        }
        else { q = p;
            p = p->next;
        }
    }
    return r;
}
```
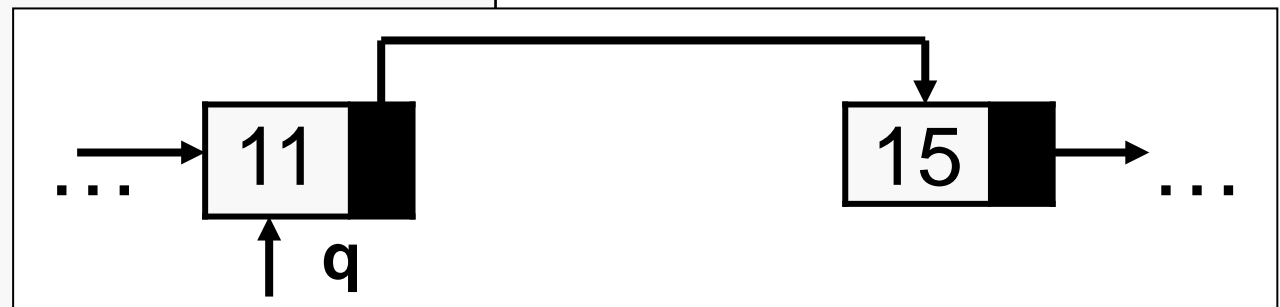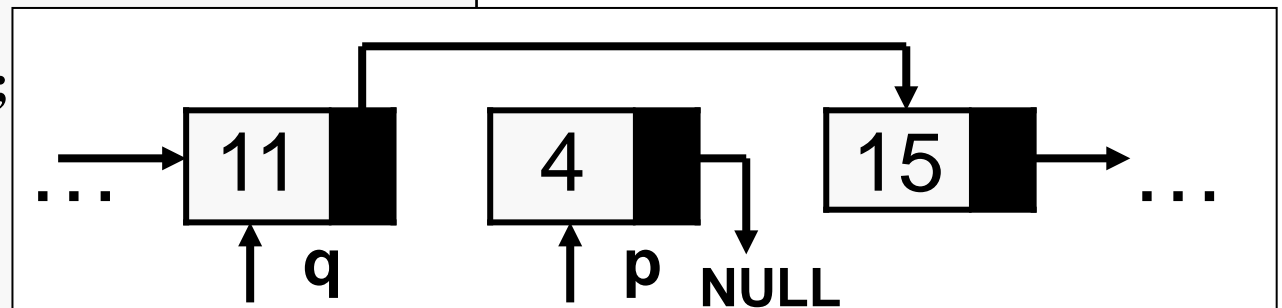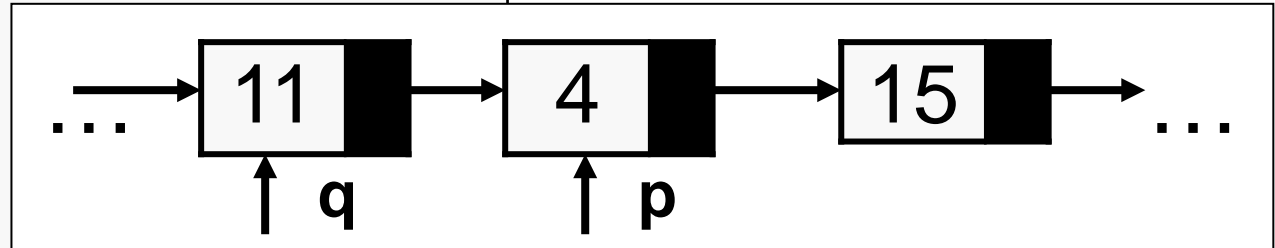






38

# Practice Problems

- Print a list backwards (try a recursive print)
- Count the number of elements in a list (both using and not using recursion)
- Concatenate two lists
- Reverse a list
- Delete the maximum element from a list

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