Linked Lists

Click to add text

Introduction

- Bricks, iron rods, cement bags, wooden planks:

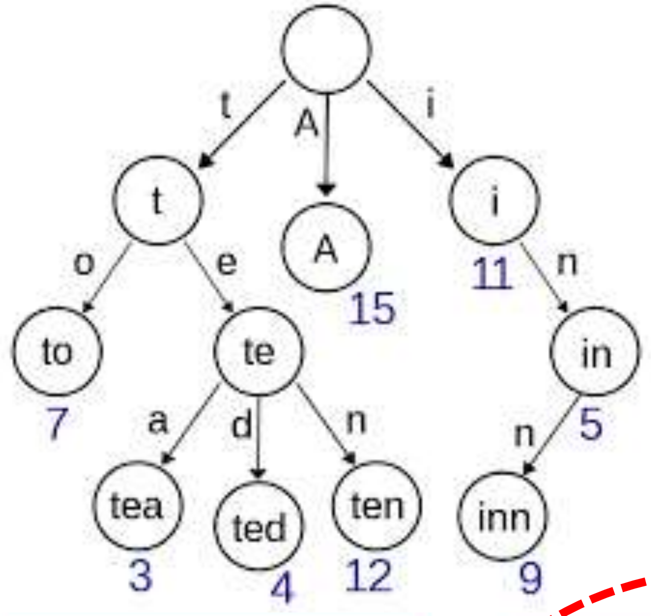
  - By themselves, are of little use in our daily lives.
  - But, become useful when civil engineers build structures using these basic building blocks.

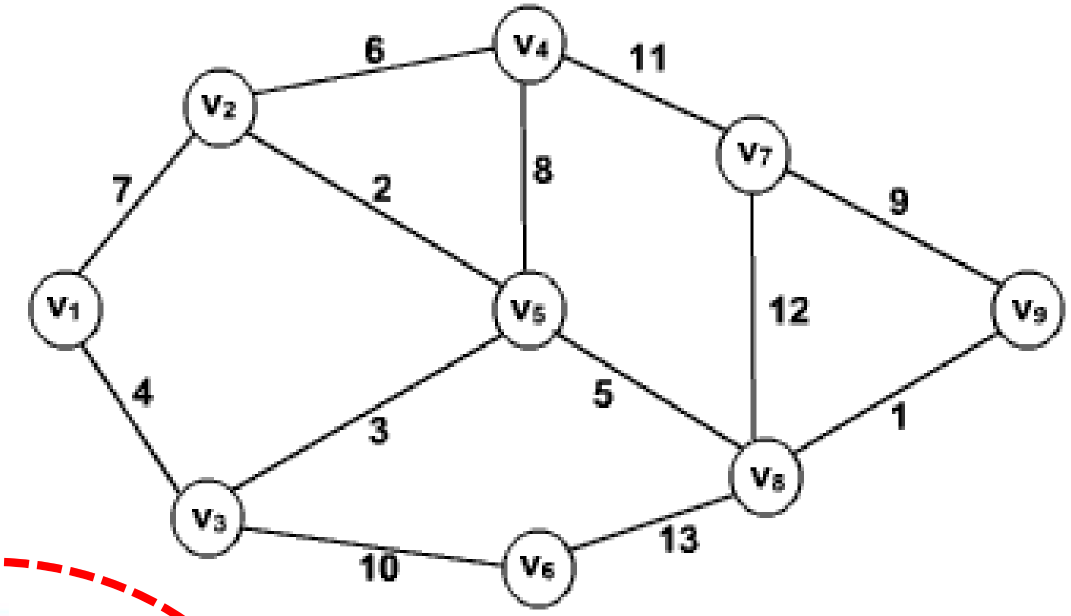
Civil Engineering Structures

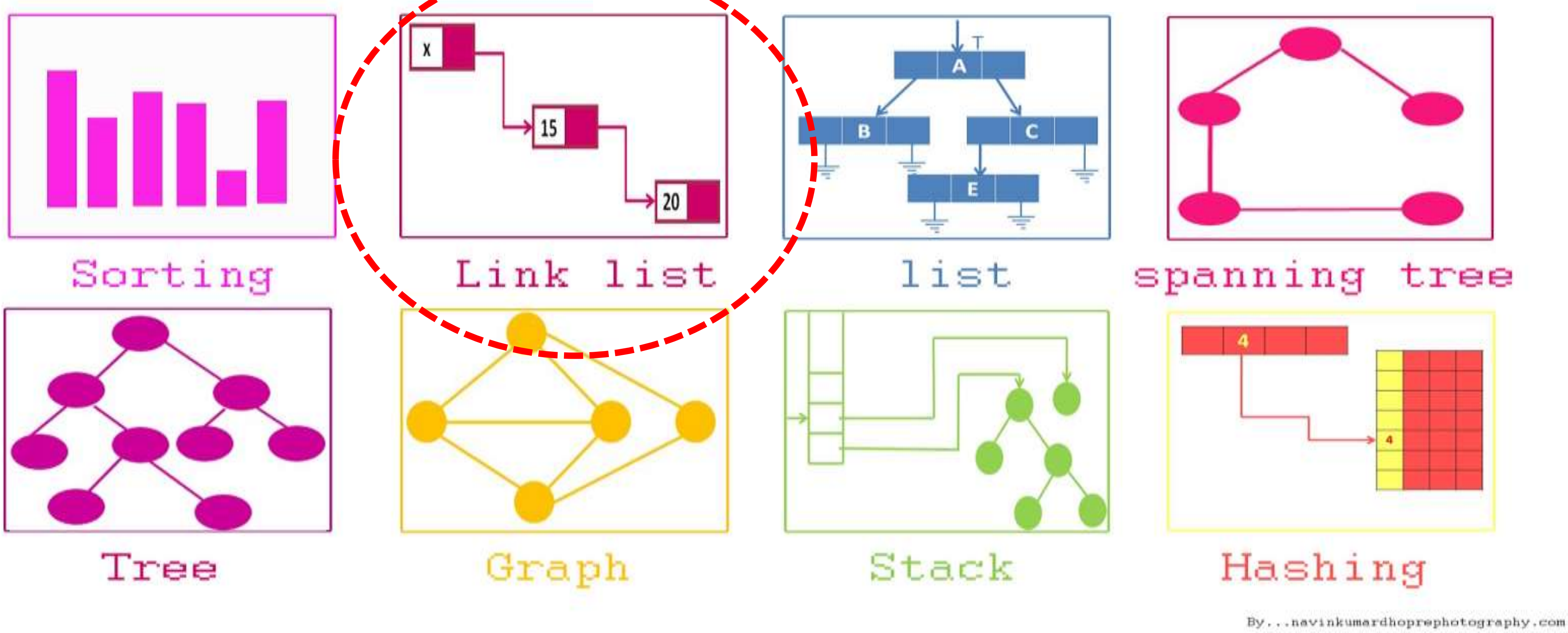
Basic Data Items

- Int
- Long int
- Float
- Char
- Double
- Unsigned
- Bool

Data Structures



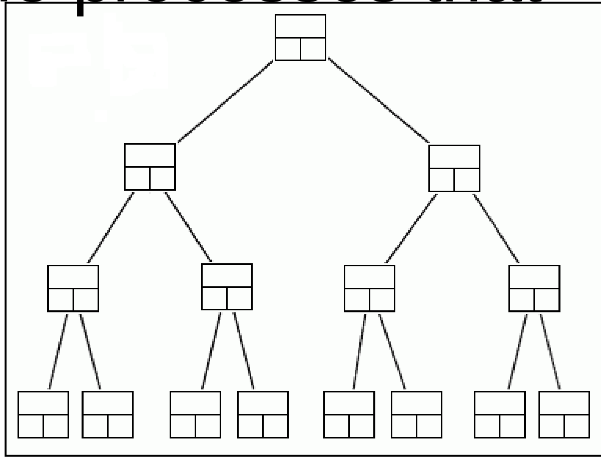




Data Structures

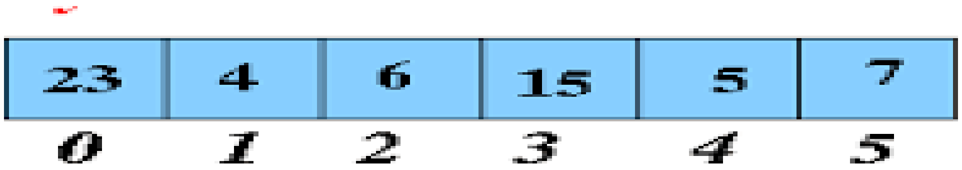
- Primitive data: int, float, char, etc.
- Data Structure:
  - Primitive data organized to build sophisticated collections of data.
  - Choice of Data structure affects the performance and structure of a program.
- Need to decide which data structure to use:
  - Depends on the nature of the data and the processes that need to be performed on that data.

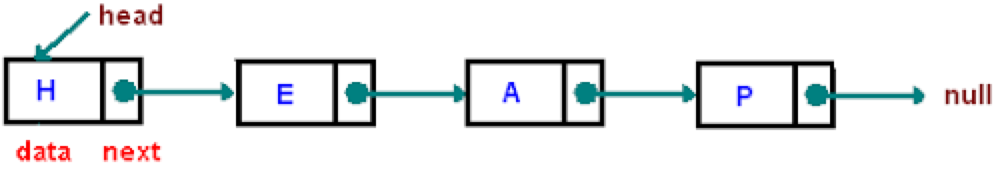
"Get your data structures correct, and the rest of the program will write itself." - *David Jones*

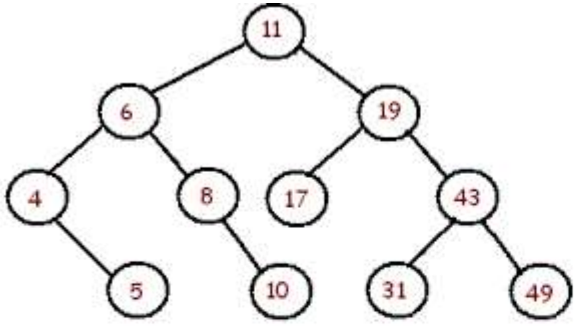


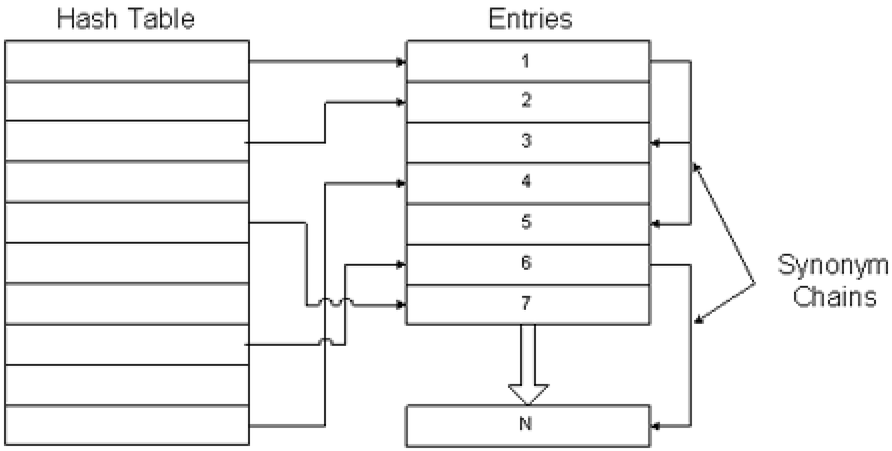
Common Data Structures

- Array
- Linked List
- Binary Tree
- Multi Way Tree
- BST
- Heap
- Hash Table` … …

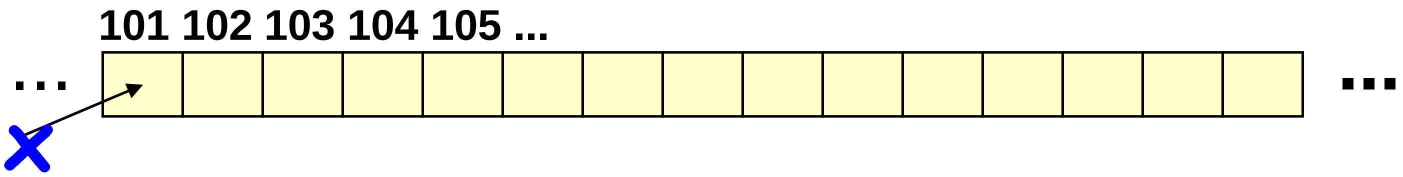








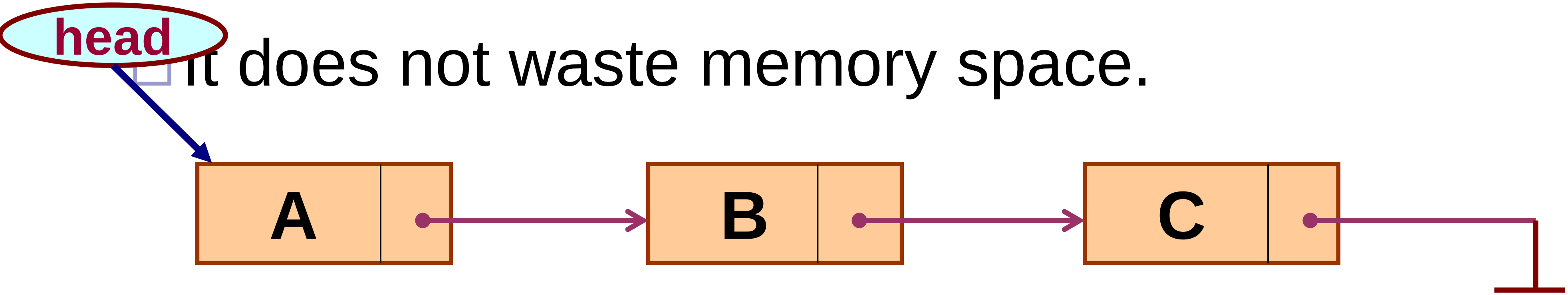
List



- **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, but adding in middle is not possible
      - □ Deleting an element or inserting an element may require shifting of elements
    - Wasteful of space

Linked List: An Introduction

- Size of a linked list can incrementally change during execution.
  - Successive elements are connected by pointers.
  - Last element points to NULL.
  - It can grow or shrink in size during execution of a program to make it just as long as required (arrays can't).
  - It does not waste memory space.



Linked Lists versus Arrays

- Advantage Linked lists:
  - □ Dynamic size
  - □ Ease of insertion/deletion
- ■ Disadvantage Linked lists:
  - – Random access is not allowed.
    - – Access elements sequentially starting from first node --- cannot do binary search with linked lists.
  - – Extra memory space for a pointer is required with each element of the list.
  - – Arrays have better cache locality

Inserting in Array Middle is Expensive…

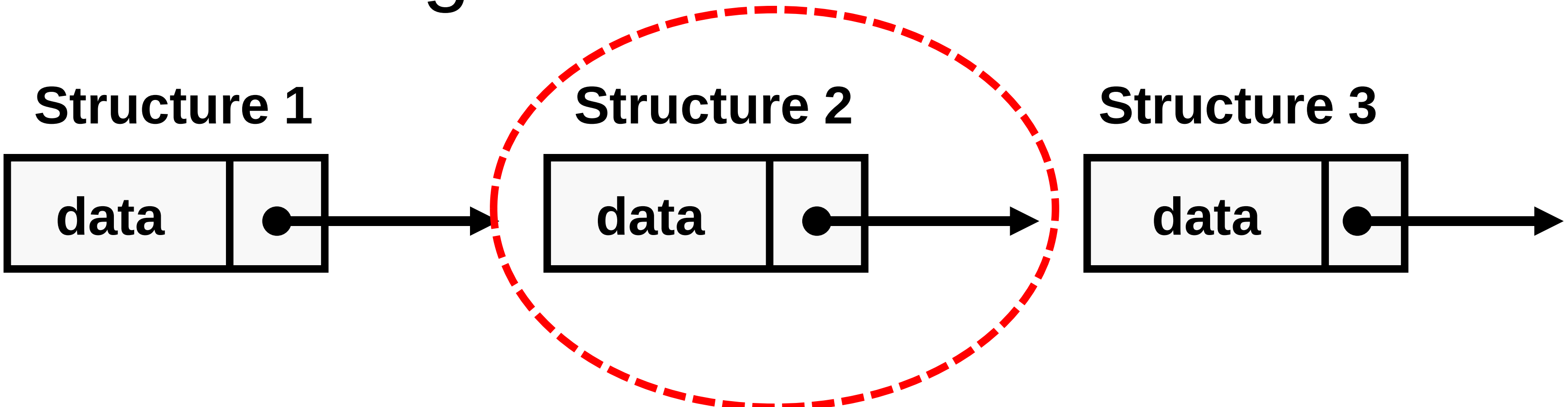
- Room has to be created for a new element:
  - To create room existing elements may have to shifted.
- For example:
  - We have a sorted list of IDs in an array id[].

id[] = [1000, 1010, 1050, 2000, 2040, …..].

  - Now we want to insert a new ID 1005
  - To maintain the sorted order, we have to move all the elements after 1000.
- Similarly deletion is expensive

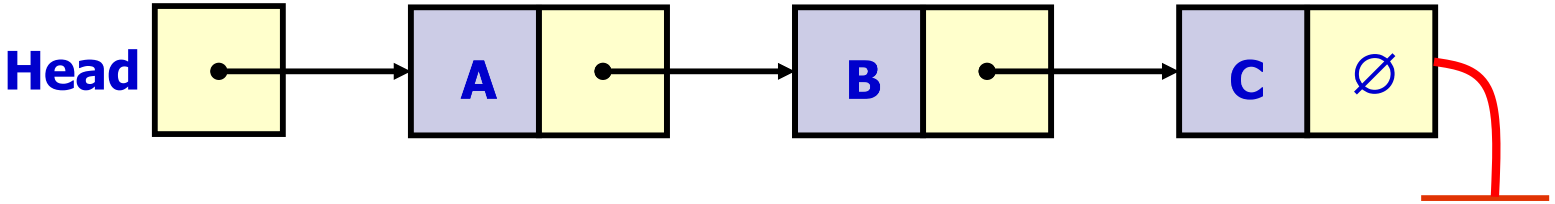
Linked List

- A very different way to represent a list
  - Make each data in the list member of a structure
  - The structure also contains a pointer or link to the structure (of the same type) containing the next data



Linked List: Basics

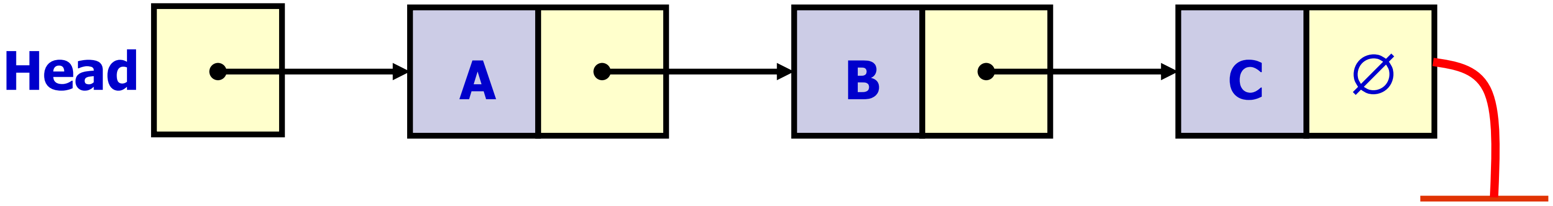
- Keeping track of a linked list:
  - □ Must know the pointer to the first element of the list (usually called *start*, *head*, …).



- Linked lists provide flexibility in allowing the items to be arranged efficiently.
  - □ Insert an element.
  - □ Delete an element.

Linked List: Basics

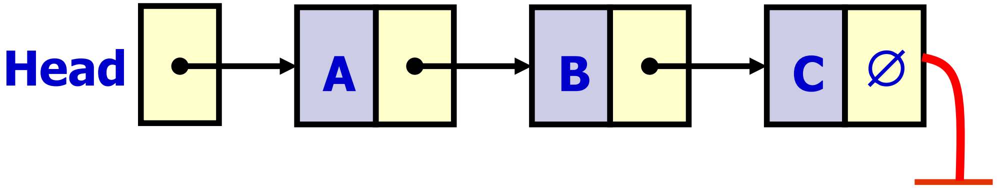
- A linked list is called "linked" because:
  - Each node in the list has a pointer that points (links) to the next node in the list.



Creating a List

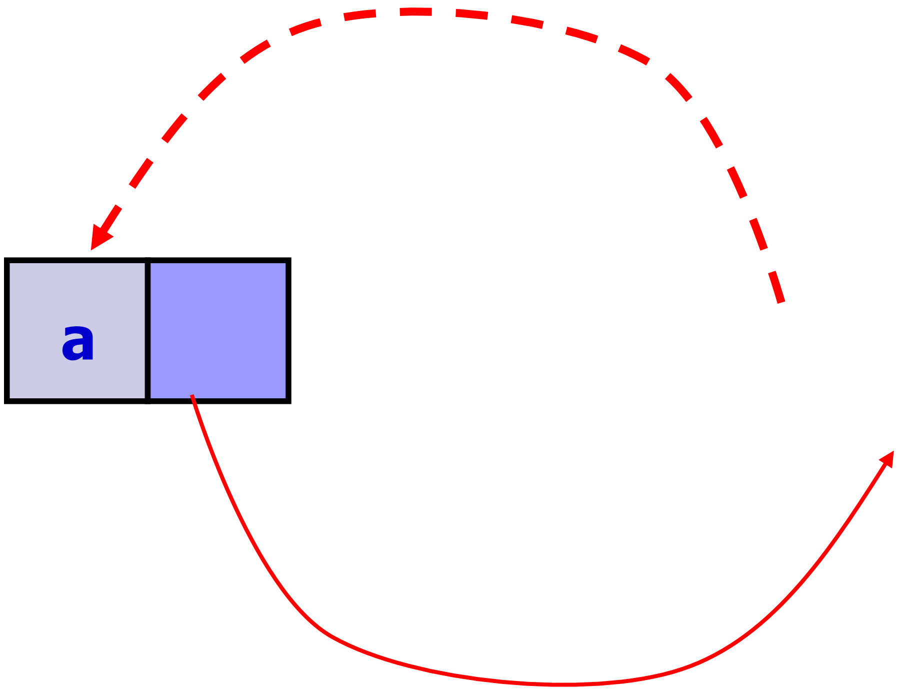
Creating a linked list

- Each structure of the list (lets call it node) has at least two fields:
  - Data
  - Address of the next structure (node) in the list



- The structures in the linked list need not be contiguous in memory
  - These 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

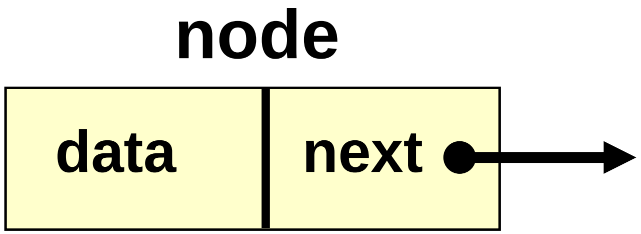
Creating list: First Define a Node



self-referential structure

Contd.

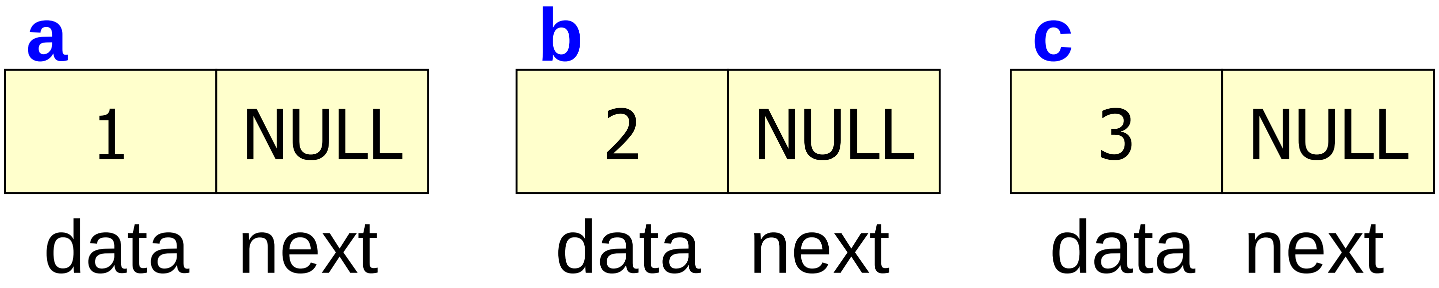
```
struct node {
        int data;
        struct node  *next;
    }
```



- The pointer variable next contains the address of the location in memory of the successor list element:
  - If no successor, pointer variable has the special value NULL (defined as 0)
  - NULL is used to denote the end of the list (no successor element)
- Structures which contain a member field pointing to the same structure type are called **self-referential structures**

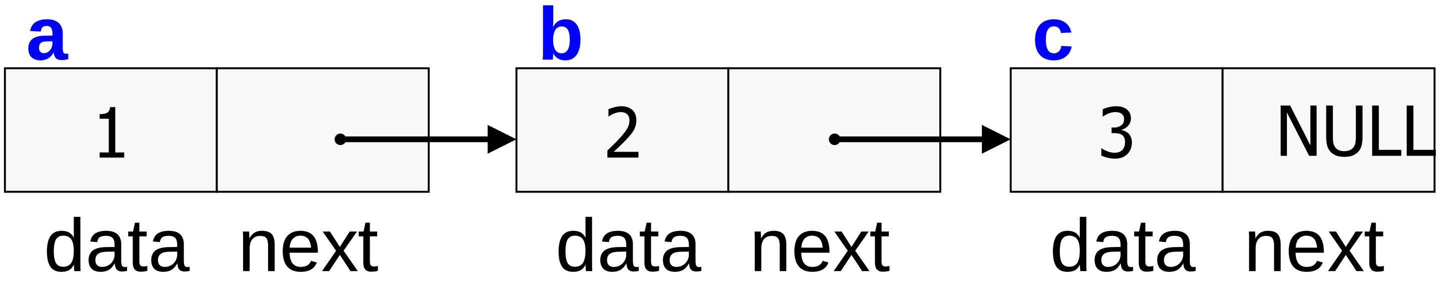
Example: nodes of the list

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



Chaining these together…

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

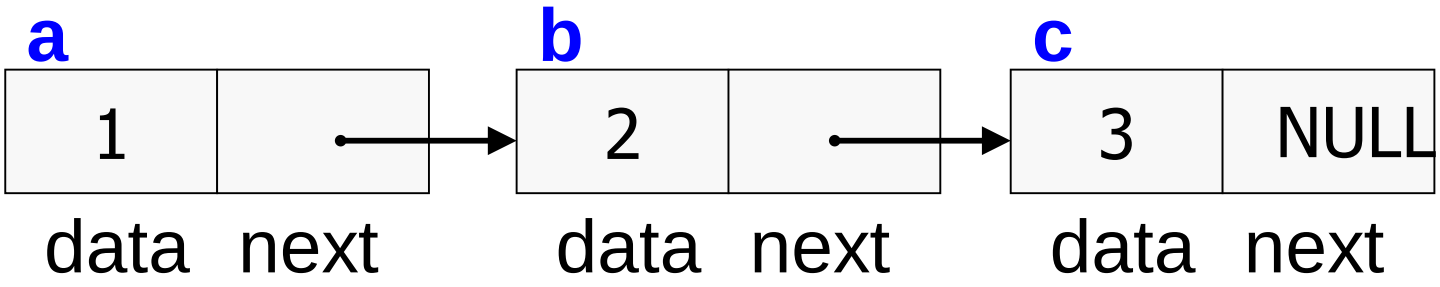


What are the values of :

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

Chaining these together…

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

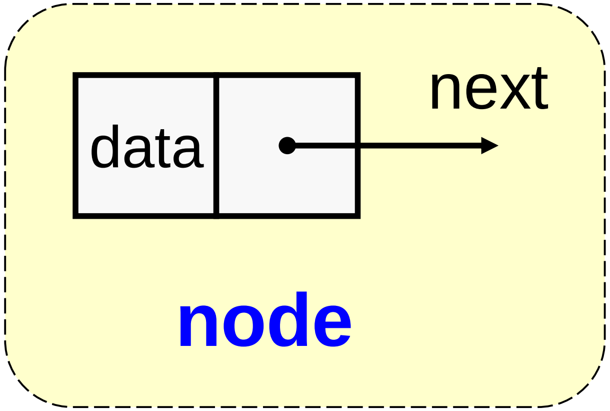


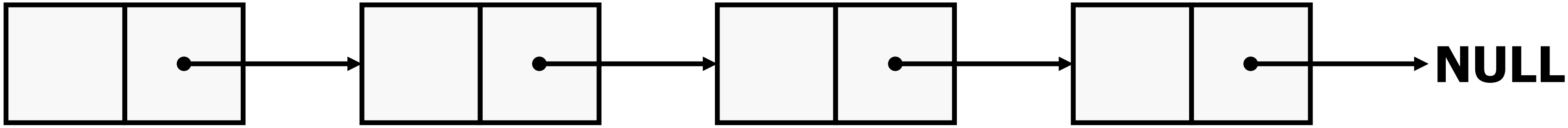
What are the values of :

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

Singly Linked Lists

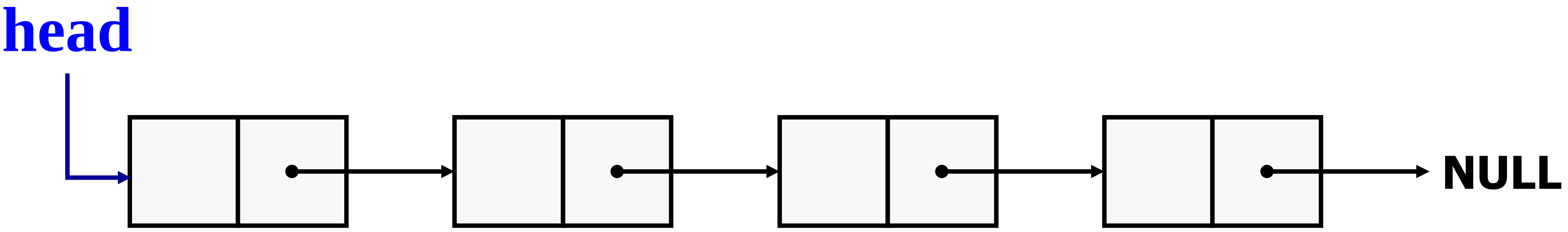
- A singly linked list is a data structure consisting of a sequence of nodes
- Each node stores
  - data
  - link to the next node





Contd.

- A head pointer stores the address of the first element of the list
- Each element points to its successor element
- The last element has a link value NULL



A general node type

- 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

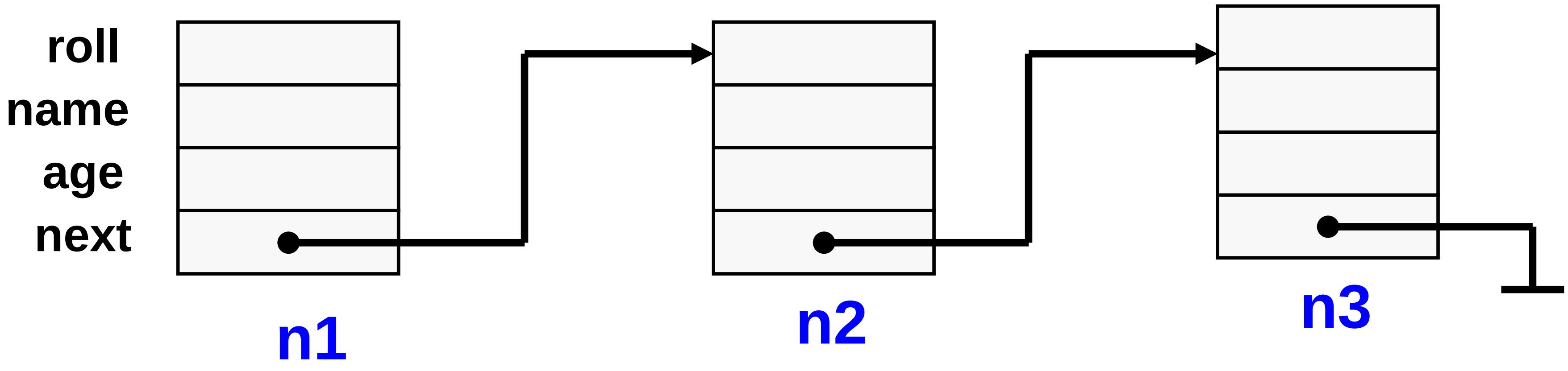
```
  struct stud n1, n2, n3;
```

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

```
#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);
```

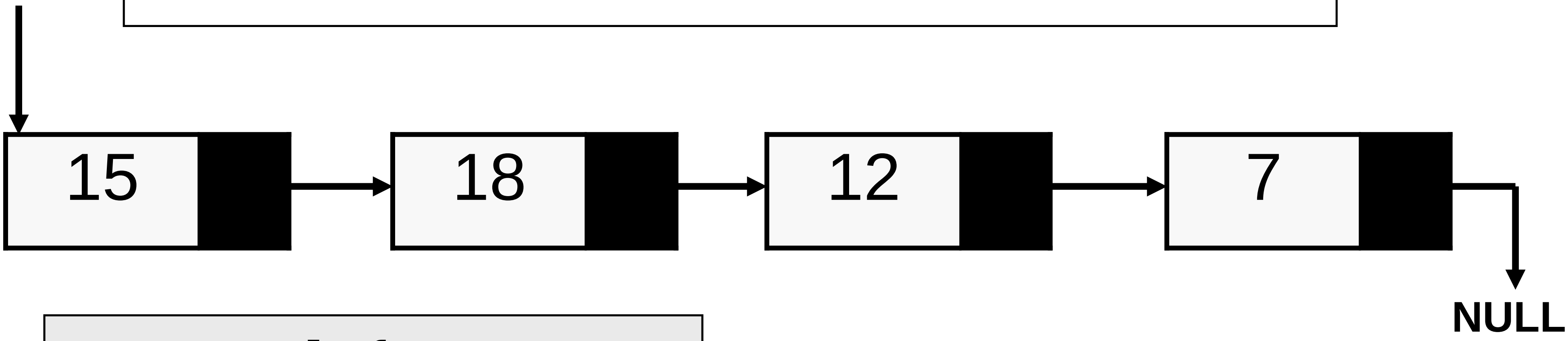
```
     n1.next  =  &n2 ;
     n2.next  =  &n3 ;
     n3.next  =  NULL ;

    /* 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: Dynamic

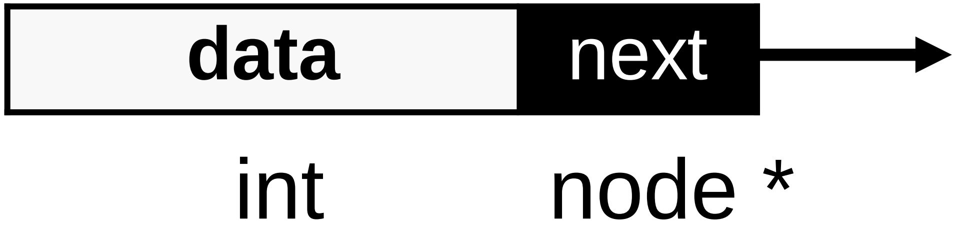
- Instead of statically declaring the structures n1, n2, n3,
  - Dynamically allocate space for the nodes
  - Use malloc individually for every node allocated
- This is the usual way to work with linked lists, as number of elements in the list is usually not known in advance (if known, we could have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

Example of dynamic node allocation

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



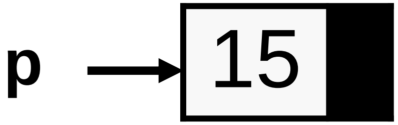
```
struct node {
    int data ;
    struct node * next ;
} ;
struct node *p, *q;
```



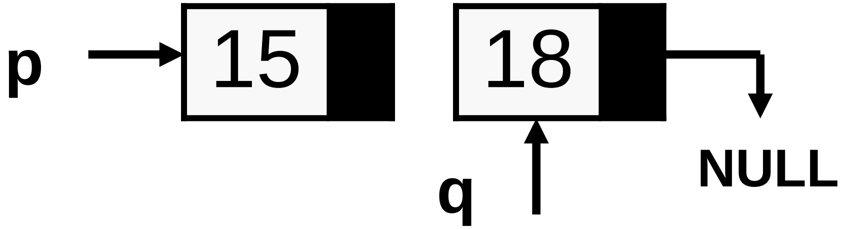
Adding 15 and 18 only

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

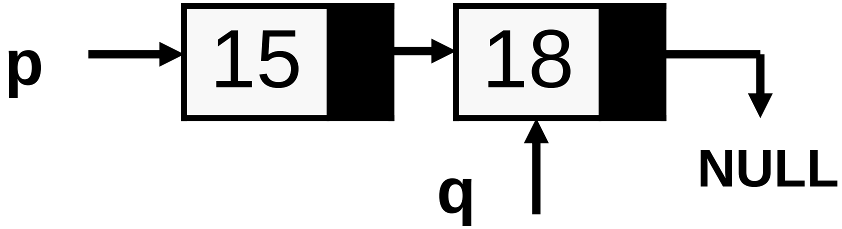
```
p = (struct node *) malloc(sizeof(struct node));
p->data=15;
```



```
q = (struct node *) malloc(sizeof(struct node));
q->data=18;  q->next = NULL;
```



```
p->next = q;
```



Traversing the elements

```
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
```

Using code similar to traversing, we could have also

- Searched for an element
- Updated an element etc.

Example: adding n elements read from keyboard

```
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

```
node *create_list(){
    int  k, n;
    node  *tmp, *head;

    printf  ("\n How many elements to enter?");
    scanf ("%d", &n);
    for  (k=0; k<n; k++){
        if (k == 0) {
          head = (node *) malloc(sizeof(node));
          tmp = head;
       }
       else{
             tmp->next= (node *) malloc(sizeof(node));
             tmp = tmp->next;
           }
  scanf("%d %s %d", &tmp->roll, tmp->name, &tmp->age);
    }

    p->next  =  NULL;
    return (head);
}
```

- From `main()` call the function as follows:

```
node *head;
………
head = create_list();
```

Traversing a List

What is to be done?

- Once the linked list has been constructed and *head* points to the first node of the list,
  - Follow the pointers.
  - Display the contents of the nodes as they are traversed.
  - Stop when the *next* pointer points to NULL.

Example: display 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 at any time points to the current element 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

Function to display a list

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

```
void display (struct node *r){
      struct node *p = r;
      printf(“List = {”);
                    ....
             Write code here
                    ....
                    ....

       printf(“}\n”);
 }
```

```
void display (node *head){ //Student
  int  count = 1;      //List Example
  node  *p;

  p = head;
  while (p != NULL){
    printf("\nNode %d:%d%s%d", count,
        p->roll, p->name, p->age);
    count++;
    p = p->next;
  }
  printf ("\n");
}
```

- From main() call the display function as:

```
node *head;
.........
display (head);
```

Important to remember

- Store the address of the first element in a separate pointer (head in our examples), and make sure you do not change it
  - If you lose the start pointer, you cannot access any element in the list
  - In the 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

Exercise

Assume that a linked list is made up of nodes that are defined by using the following structure:

```
struct listNode{
        int data;
        struct listNode *nextPtr;
} *cursor;
```

Suppose the variable **cursor** points to a node in a linked list. Which one of the following expressions will be true when **cursor** points to the tail node of the list?

a. (cursor == NULL)

b. (cursor->nextPtr == NULL)

c. (cursor->data == NULL)

d. (cursor.nextPtr == NULL)

Exercise

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

int f(struct node *head){
  struct node *p=head;
   return (
        (p == NULL) ||
        (p->next == NULL) ||
        (( P->data <= p->next->data) && f(p->next))
      );
 }
```

- What does the function f achieve, given that head points to the head of a singly linked list?

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

```
struct node *search (struct node *r, int value)
{
     struct node *p;
      p = r;
```

....

Write code here

....

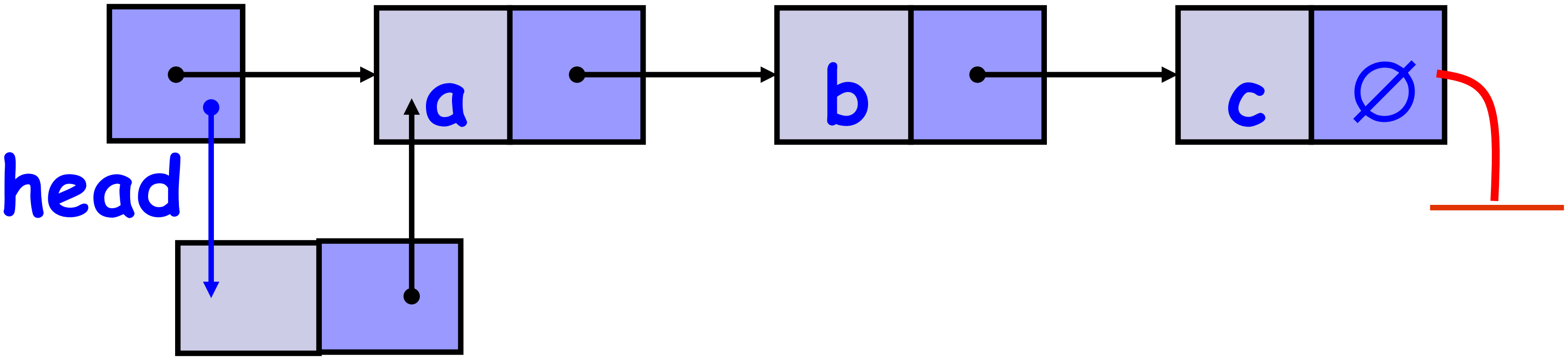
....

```
      }
      return p;
}
```

Insertion of data item 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
  - …

Inserting at Front of List



```
tmp=(node *) malloc(sizeof(node);
tmp->next=head;
head=tmp;
```

Insert in front of list

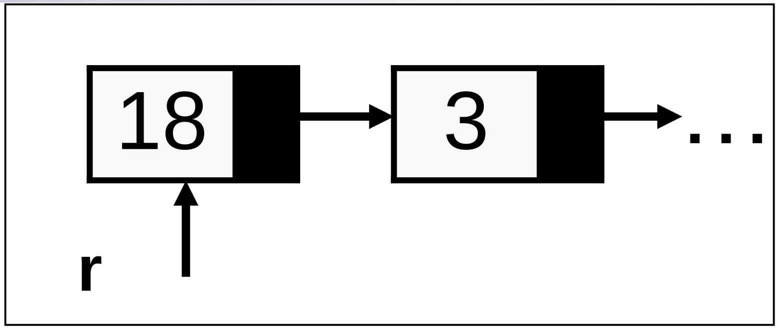
Takes the head pointer and the value to be inserted (NULL if list is empty)

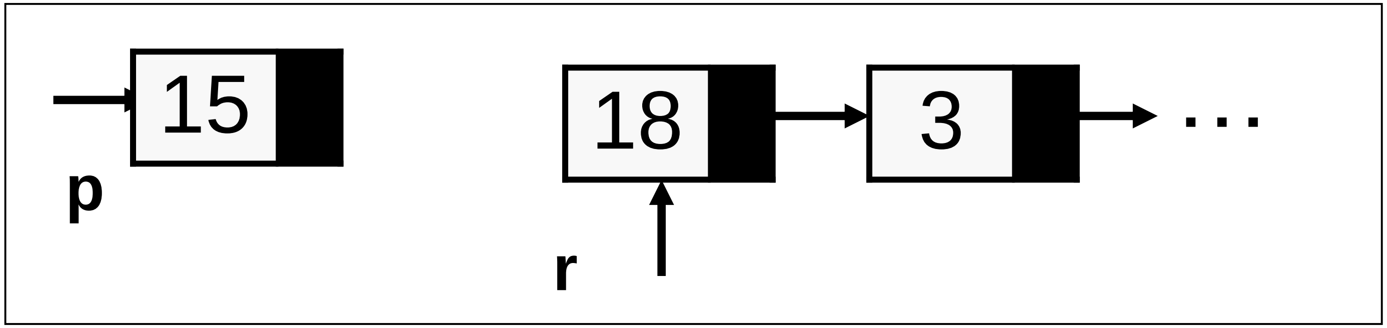
Inserts the value as the first element of the list

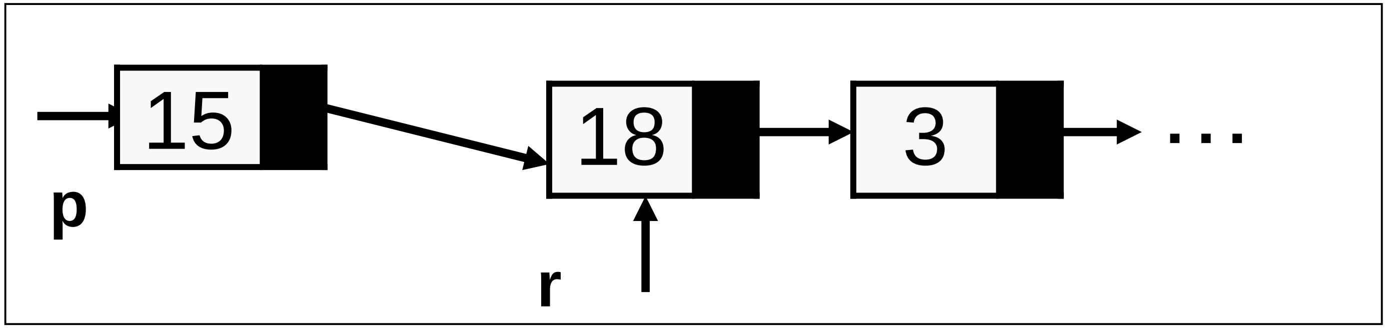
Returns the new head pointer value

```
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

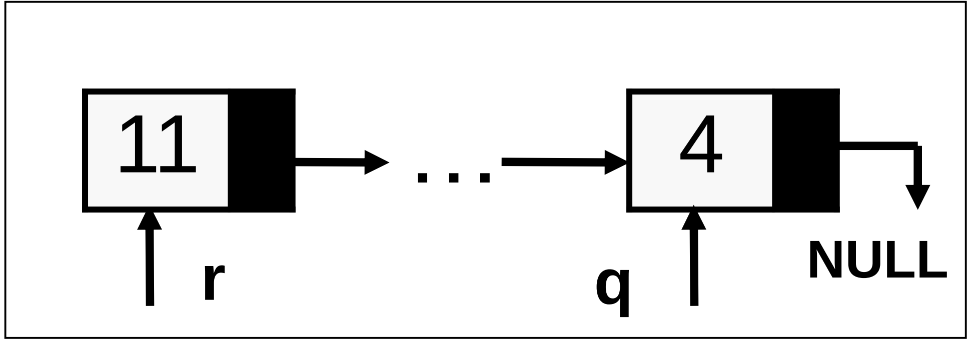
Takes the head pointer and the value to be inserted (NULL if list is empty)

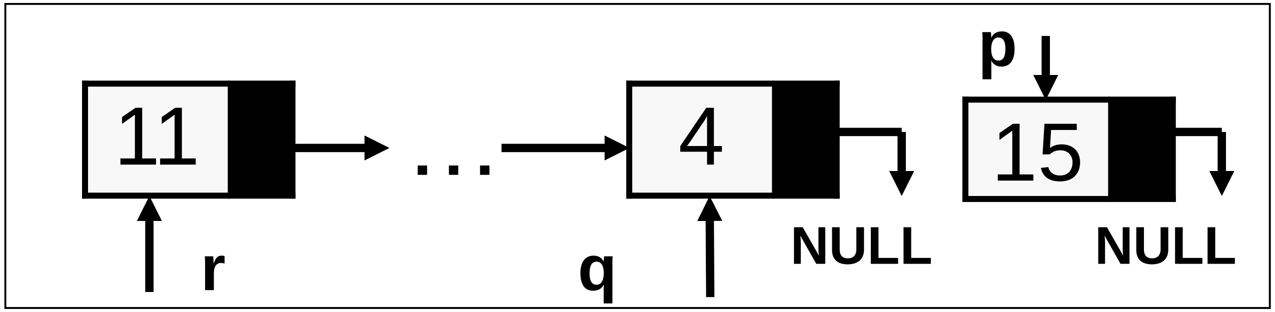
Inserts the value as the last element of the list

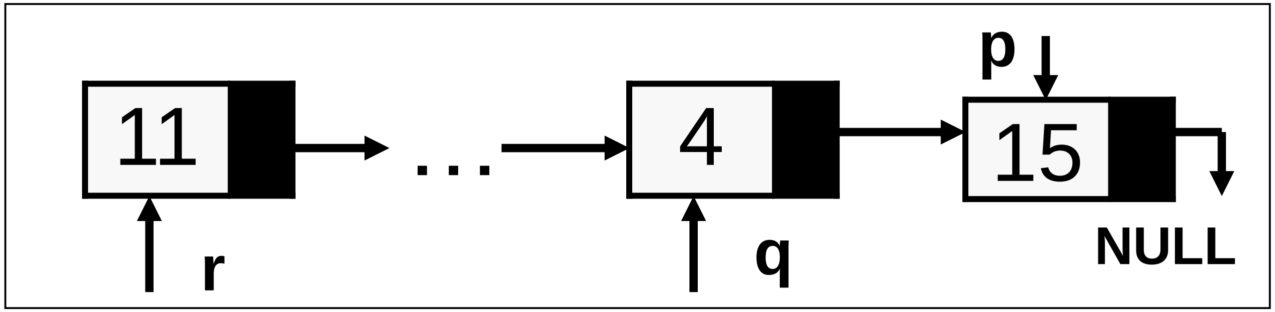
Returns the new head pointer value

```
 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.







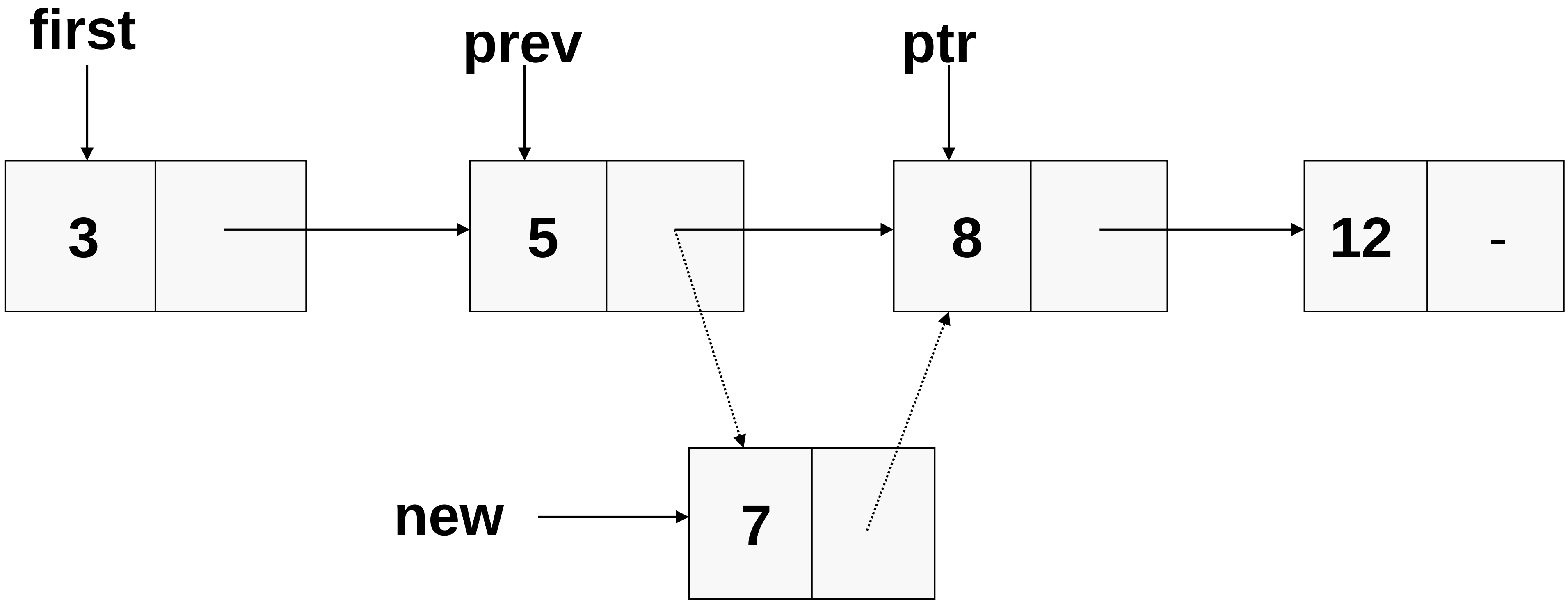
Using the Insert at End Function

```
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

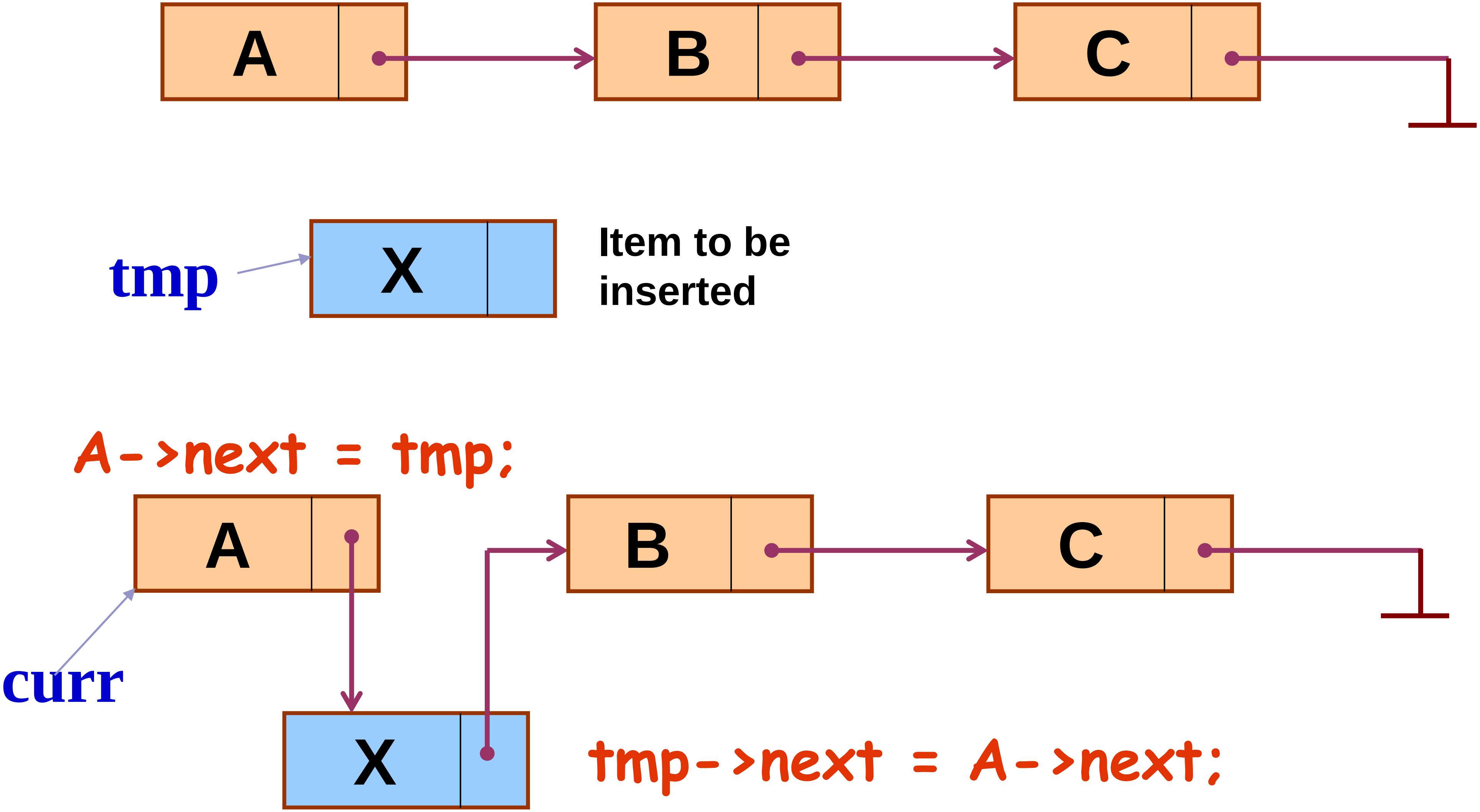
```
List = {10, }
List = {10, 11, }
List = {10, 11, 12, }
```

Insertion in Ascending Order



- 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!

Illustration: Insertion Middle of List

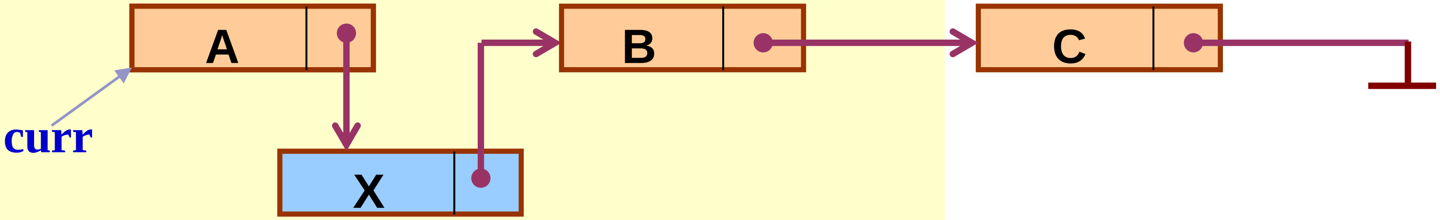


A->next = tmp;

tmp->next = A->next;

Pseudo-code for insertion

```
struct node {
   double data;
   struct node * next;
   };
```



```
void insert(node *curr){
node * tmp;


         tmp=(node *) malloc(sizeof(node));
         tmp->next=curr->next;
         curr->next=tmp;
}
```

Insert in ascending order & sort

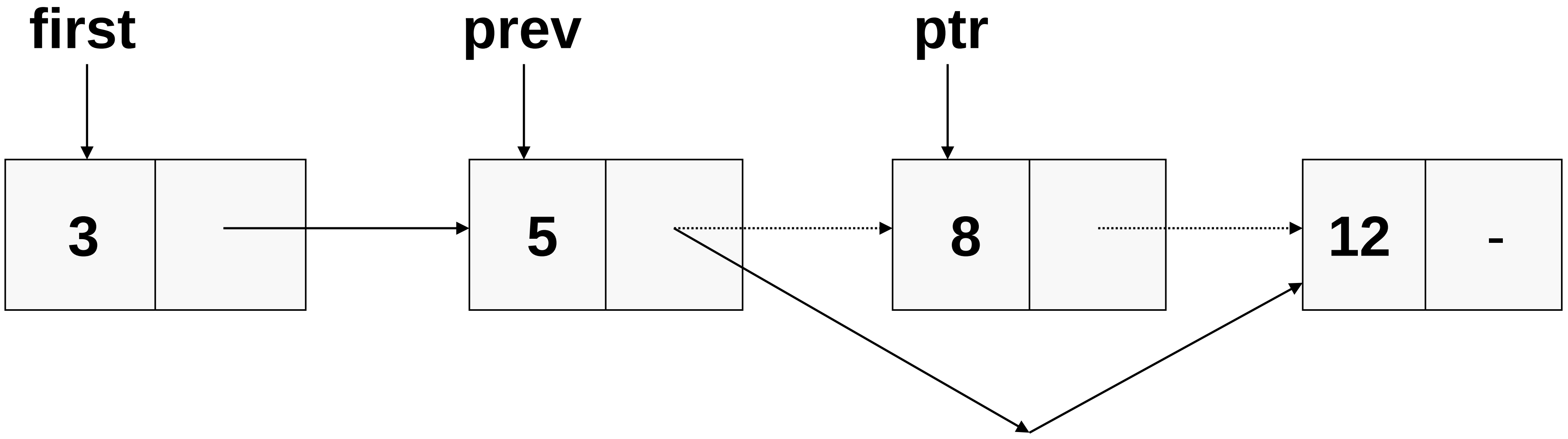
```
struct node * insert_asc(struct node * r, int value)
{ struct node *p, *q, *new;
  new = (struct node *) malloc(sizeof(struct node));
  new->data = value;  new ->next = NULL;
  p = r;  q = p;
  while(p!=NULL) {
    if (p->data >= value) { /* insert before */
      if (p==r) {  new->next =r; /* insert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

```
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;
}
```

Deletion from a list

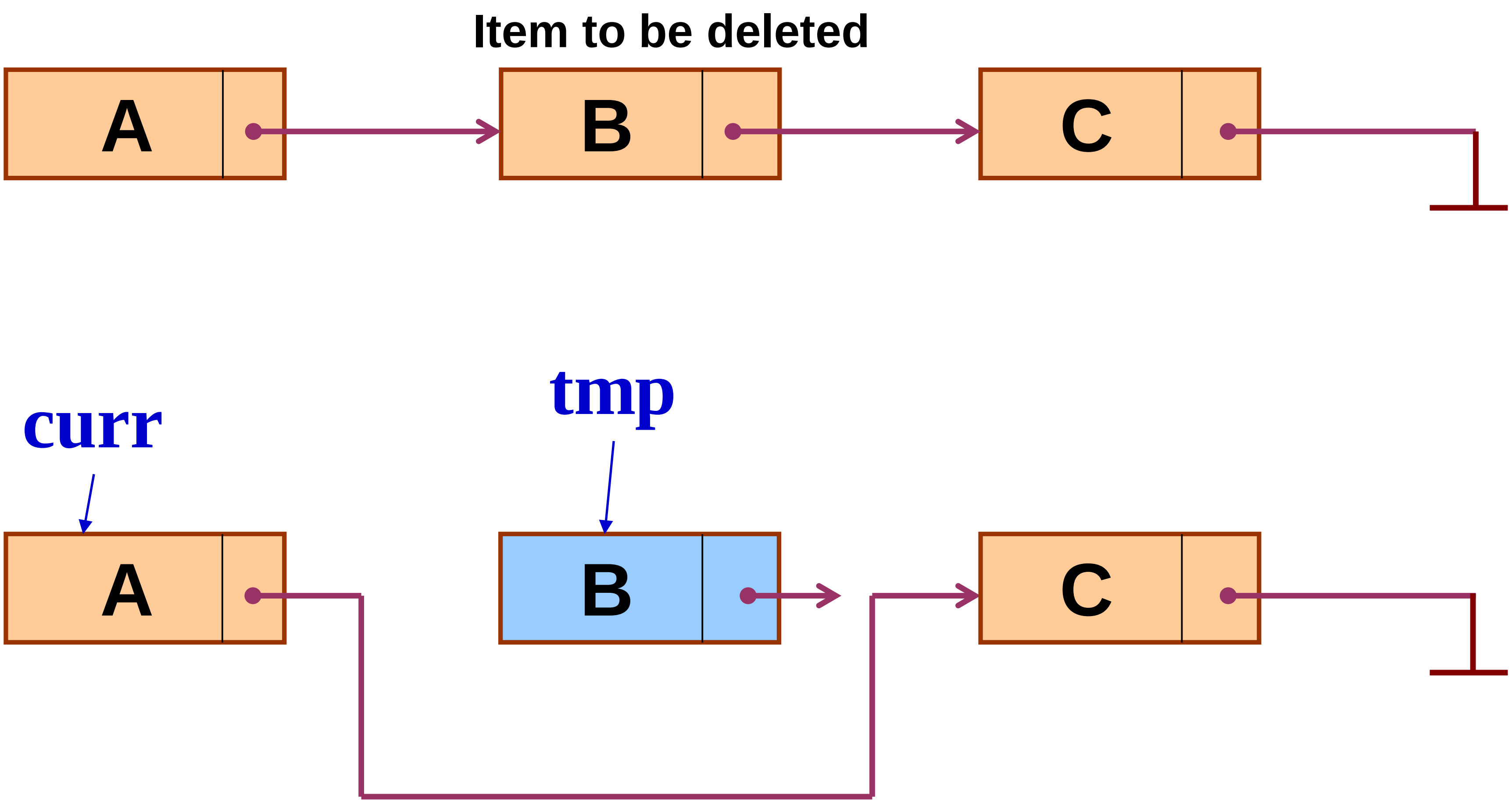
- To delete a data item from a linked list
  - Find the data item in the list, and if found
  - Delink this node from the list
  - Free up the malloc’ed node space

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

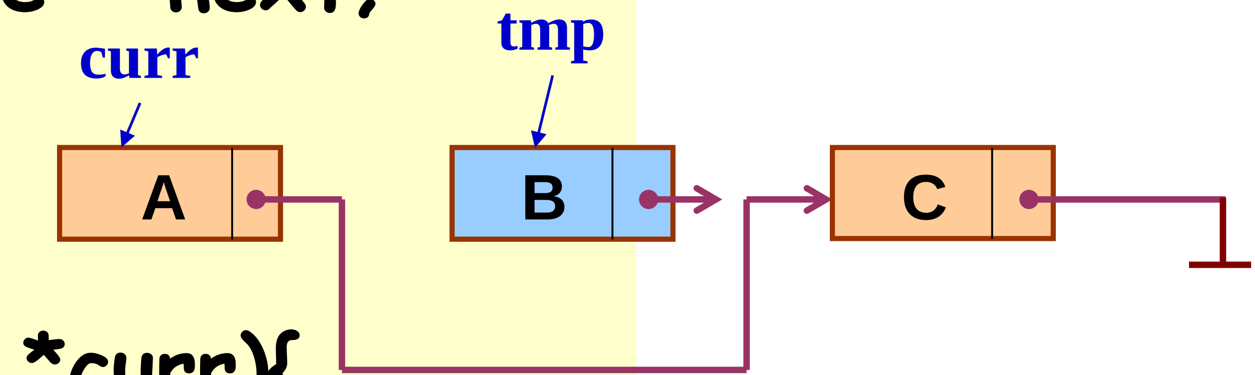
Illustration: Deletion



Quiz: How many links need to be changed to delete one node?

Code for deletion

```
struct  node {
        double data;
        struct node * next;
   } node;

void delete(node *curr){
      node * tmp;
      tmp=curr->next;
      curr->next=tmp->next;
      free(tmp);
}
```

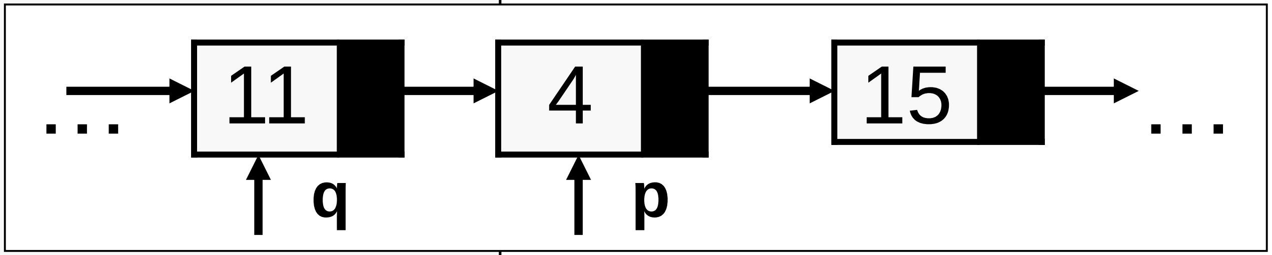


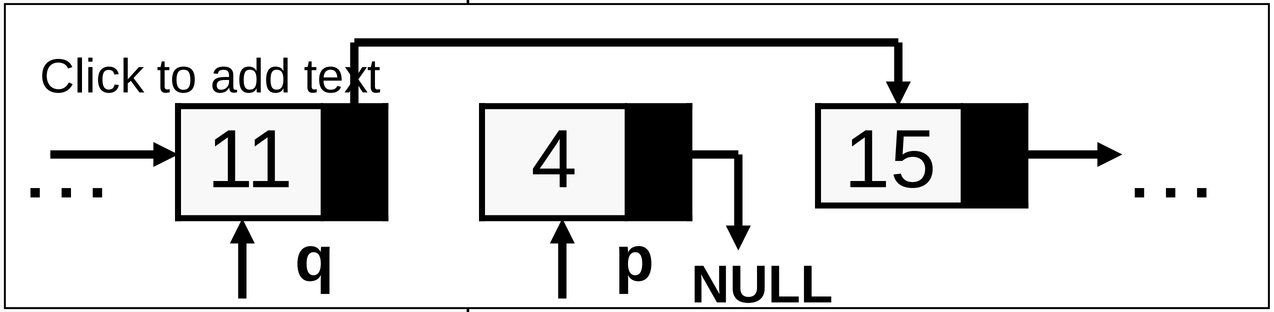
…
Write your code here…
…

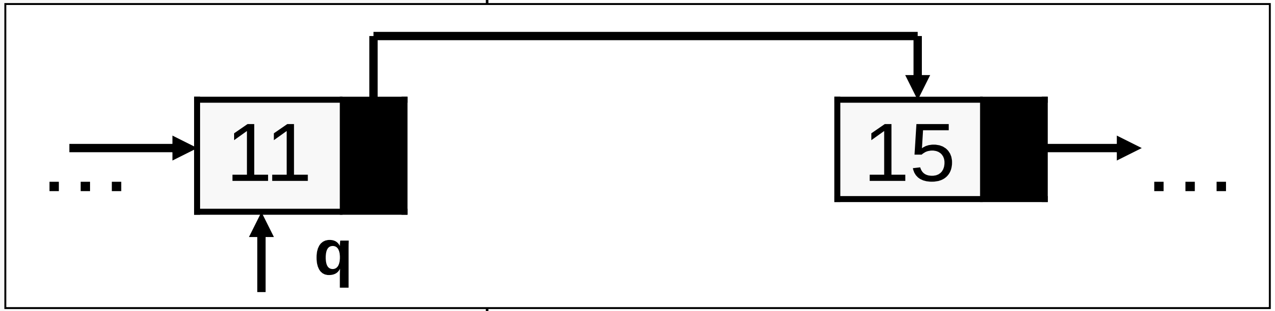
Deleting an element

delete(r,4)

```
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;
}
```







Exercise

Suppose you are given pointers to the first and the last nodes of a singly linked list, which of the following operations would require traversal of the linked list?

a. Delete the first node
b. Insert a new node as a first element
c. Delete the last node of the list
d. Add a new node at the end of the list

Exercise

- **delete-alternate-nodes(node *head)** Deletes every second node. Fill the missing parts

```
void delete_alternate_node(node *head){
        node *p = head, *oldNode;
        while( p->next != NULL ){
                oldNode = p->next;
                p->next = p->next->next; // Bypass deleted node
                delete oldNode;
                if( p->next != NULL ) p = p->next;
                                        // Advance to the next node
        }
}
```

…

Write your code here

Exercises

- 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

Reverse List

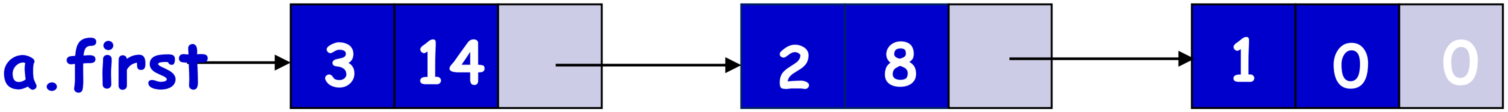
```
Next=head,; prev=NULL;
while (current != NULL)
   {
      next  = current->next;
      current->next = prev;
      prev = current;
      current = next;
   }
   head = prev;
}
```

Example Application – Polynomials

- A polynomial is a sum of terms.
- Each term consists of a coefficient and a (common) variable raised to an exponent.
- We consider only integer exponents, for now.
- Example: $\mathbf{9x^8 + 4x^3 + 5x - 10}$

Application – Polynomials

- How to represent a polynomial?
- Issues in representation
  - should not waste space
  - should be easy to use it for operating on polynomials.



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

Using Array for Polynomial

- Using an array, index k stores the coefficient of the term with exponent k.
- Advantages and disadvantages
  - □ Exponent stored implicitly (+)
  - □ May waste lot of space. When several coefficients are zero (—)
  - □ Exponents appear in sorted order (+)
  - □ Even if polynomials are not sparse, the result of applying an operation to two polynomials could be a sparse polynomial. (--)

Application – Polynomials

```
struct node{
    float coefficient;
    int exponent;
    struct node *next;
}
```

- How to use a linked list?
- Each node of the linked list stores the coefficient and the exponent.
- Should also store in the sorted order of exponents.
- The node structure is as follows:

Application -- Polynomials

- How can a linked list help?
  - Can only store terms with non-zero coefficients.
  - Does not waste space.
  - Need not know the terms in a result polynomial apriori. Can build as we go.

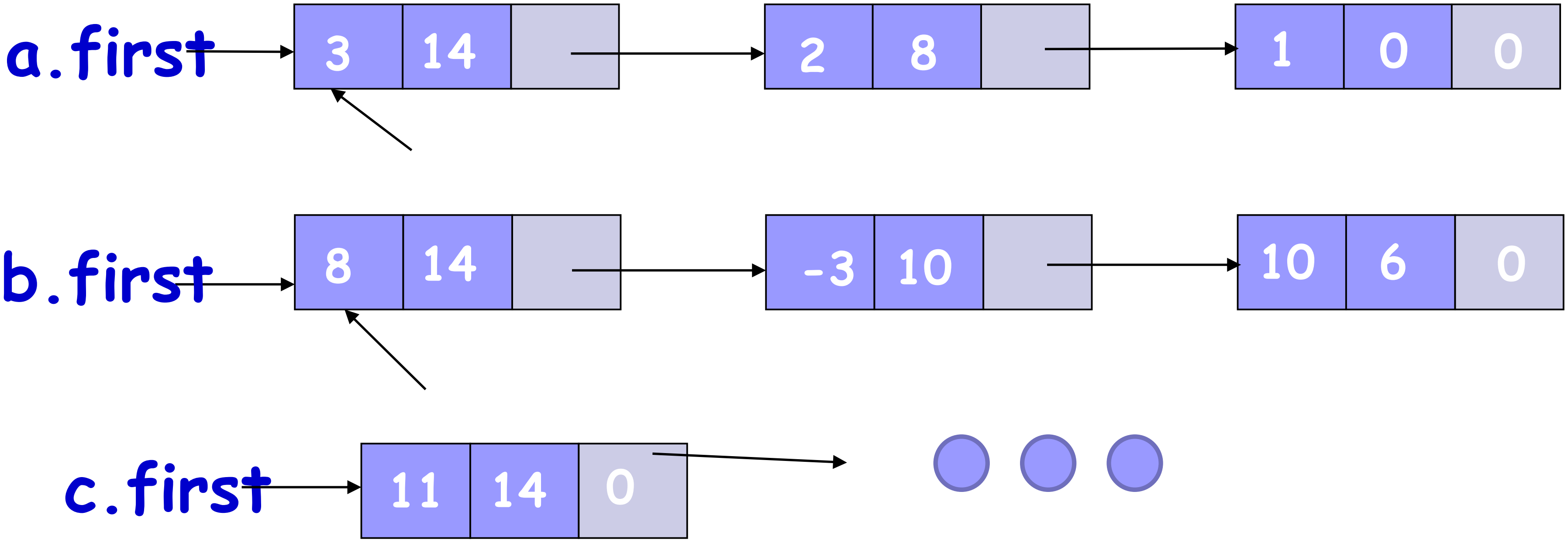
Operations on Polynomials

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

Operations on Polynomials

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

Addition of Two Polynomials



(i) `p->exp == q->exp`

Multiplication…

- Can be done as repeated addition.
- Multiply P1 with each term of P2.
- Add the resulting polynomials.