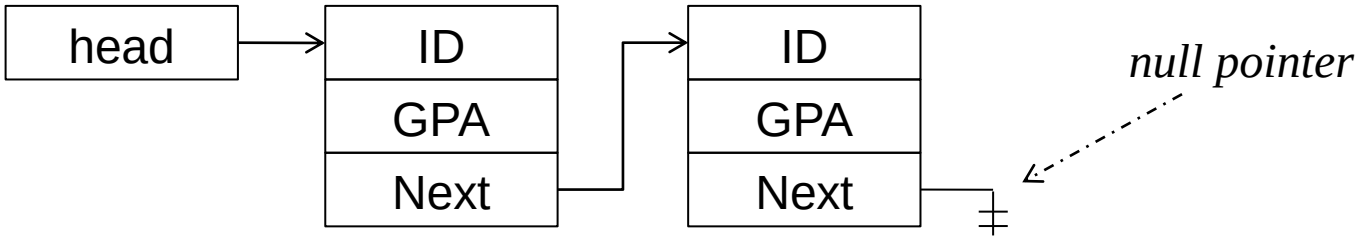
LINKED LISTS

OVERVIEW

OVERVIEW

- **What is a linked list?**
  - A collection of nodes objects that have been dynamically allocated and connected to each other
  - Each node object contains several pieces of information we want to store (student ID, GPA, etc.)
  - Each node object has a pointer to another node object
  - The pointer to the first node is stored in a head pointer
  - A linked list can grow dynamically by adding more nodes



OVERVIEW

- **Pro: Arrays are very fast to access**
  - We can read/write any location using array index
- **Con: Arrays are fixed size**
  - Must create a new array and copy data to grow in size

- **Pro: Linked lists are dynamic inn size**
  - We can insert or delete nodes anywhere without moving data
- **Con: Linked lists are slow to access**
  - Must start at head and loop over linked list to find data

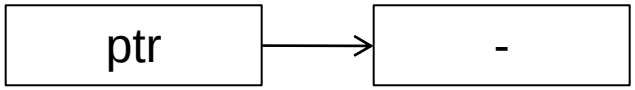
OVERVIEW

- **Review of pointer syntax:**
  - We declare a pointer variable using * operator
  - A pointer contains the address of another variable/object

```
int *ptr;
```

  - We allocate memory for a variable/object using new command
  - We draw this using one box pointing to another box

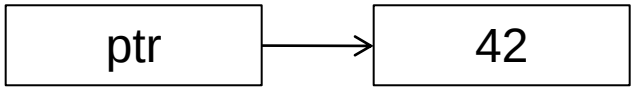
```
ptr = new int;
```



OVERVIEW

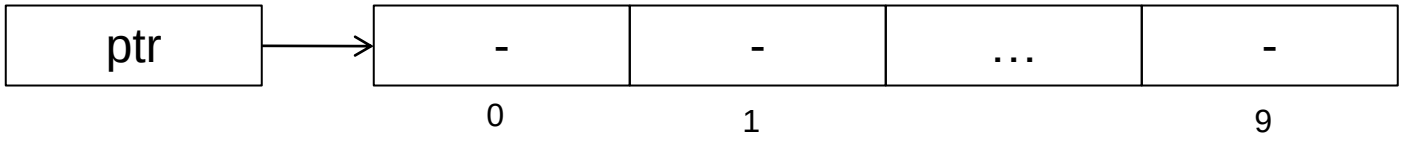
- **Review of pointer syntax:**
  - To access this dynamic memory we use * operator again
  - The * in front of a pointer dereferences the pointer

```
*ptr = 42;
```



  - We can allocate memory for an array with the new command
  - The array size is given inside square brackets

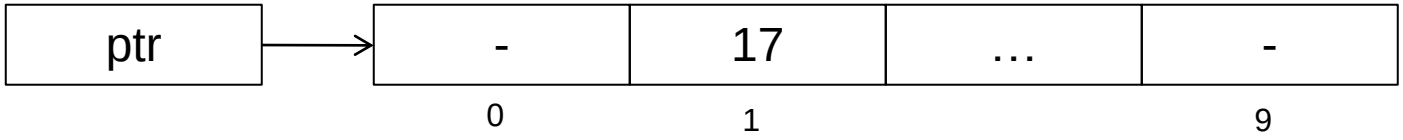
```
ptr = new int[10];
```



OVERVIEW

- **Review of pointer syntax:**
  - To access data in dynamic array we use index notation just like we use for a static array

```
ptr[1] = 17;
```



  - We can also access array using * operator by adding the index to the pointer address (ugly option)

```
cout << *(ptr+1);
```

OVERVIEW

- **Review of pointer syntax:**
  - To return memory to the O/S we use the delete command
  - We should also set pointer to NULL (pointer to nothing)

```
delete ptr;
ptr = NULL;
```

  - To return memory for an array we add [ ] before pointer
  - We should also set pointer to NULL (pointer to nothing)

```
delete [] ptr;
ptr = NULL;
```

OVERVIEW

- **Syntax for object pointers:**
  - To store data in a linked list we must declare a Node class

```
class Node
{
   public:
     int ID;
     float GPA;
     Node *next;
}
```

We simplify linked list code by using a class with public data and no methods

OVERVIEW

- **Syntax for object pointers:**
  - To store data in a linked list we must declare a Node class

```
class Node
{
   public:
     int ID;
     float GPA;
     Node *next;
}
```

We store data in a collection of Node attributes

OVERVIEW

- **Syntax for object pointers:**
  - To store data in a linked list we must declare a Node class

```
class Node
{
   public:
     int ID;
     float GPA;
     Node *next;
}
```

We use the next pointer to link this Node to another Node in list

OVERVIEW

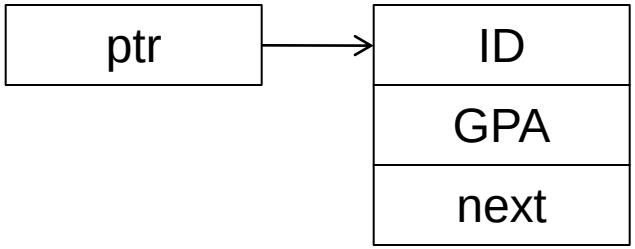
- **Syntax for object pointers:**
  - We declare a Node pointer variable using * operator
  - This pointer should be initialized to NULL

```
Node *ptr = NULL;
```

We draw NULL using ground symbol

  - To allocate space for a Node we use the new command

```
ptr = new Node();
```

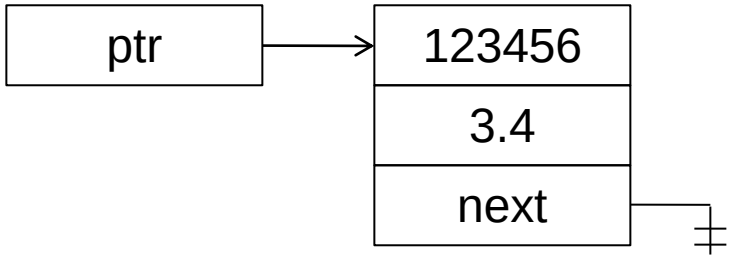


This allocates memory large enough for the Node object

OVERVIEW

- **Syntax for object pointers:**
  - To access fields inside a Node use use arrow -> notation

```
ptr->ID = 123456;
ptr->GPA = 3.4;
ptr->next = NULL;
```

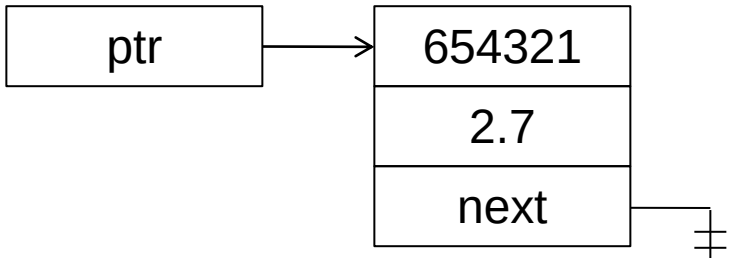


  - We start at pointer variable, follow the arrow, and go to field

OVERVIEW

- **Syntax for object pointers:**
  - If the Node fields are private, we must use get and set methods in the class to access the data fields
  - We use arrow -> notation before the method name

```
ptr->setID(654321);
ptr->setGPA(2.7);
ptr->setNext(NULL);
```



OVERVIEW

- **A linked list ADT has the following operations:**
  - Create – Initialize linked list data structure
  - Insert – Insert data into linked list (head, tail, sorted)
  - Print – Print all data in linked list
  - Search – Search for specific data in linked list
  - Delete – Remove specific value from the linked list
  - Copy – Copy all data from one linked list to another
  - Destroy – Delete linked list data structure

LINKED LISTS

OPERATION: CREATE LIST

CREATE LIST

- **To create a linked list we implement Node class with the data fields we want to store and declare a head pointer**
- **We initialize head to NULL to create an empty linked list**

```
class Node
{
   public:
     int ID;
     float GPA;
     Node *next;
}
Node *head = NULL;
```

The head pointer contains the address of the first Node in the linked list

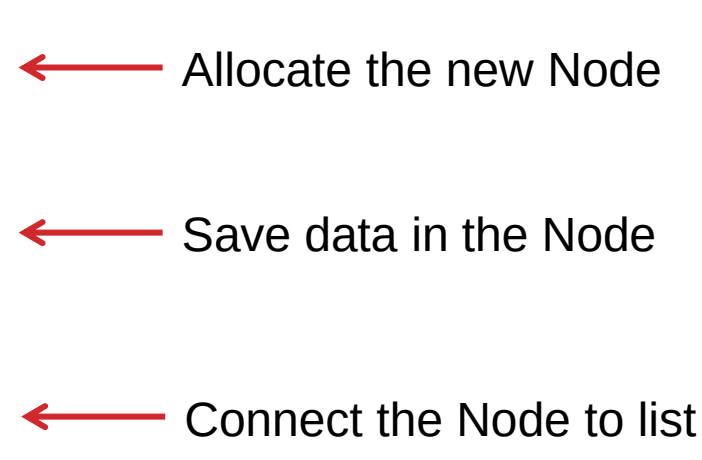
LINKED LISTS

OPERATION: INSERT HEAD

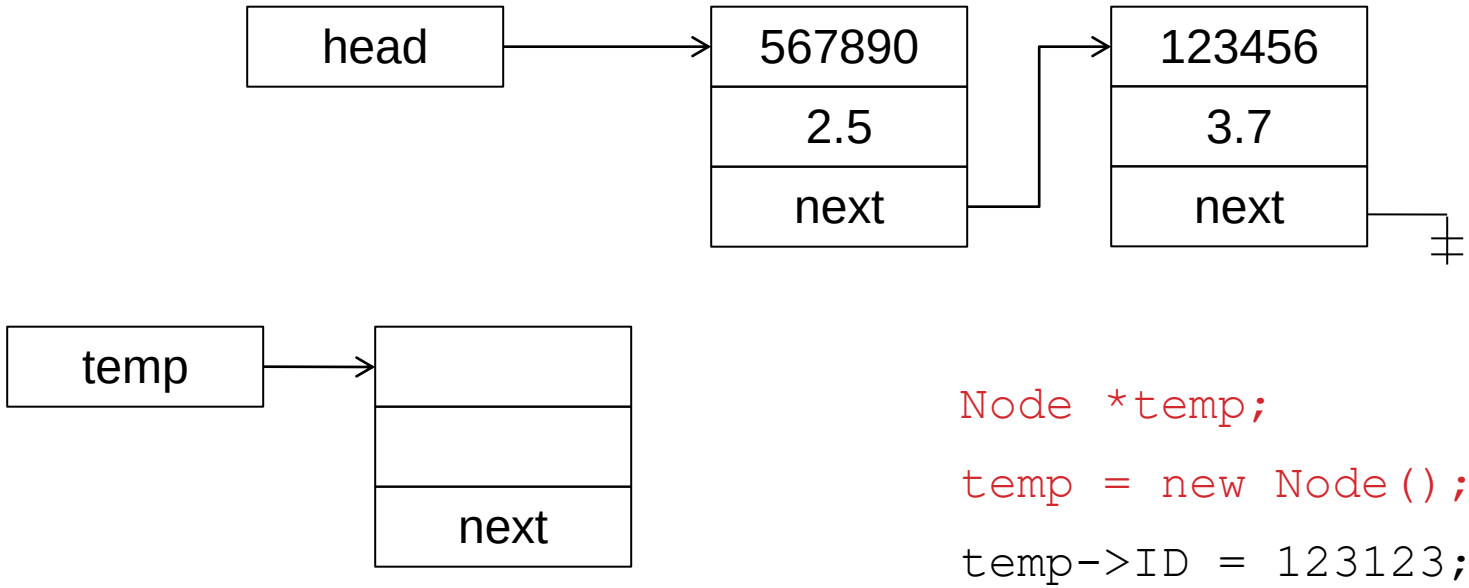
INSERT HEAD

- **The fastest way to insert data into a linked list is to connect a new node before the head of the current list**

```
Node *temp;
temp = new Node();
temp->ID = 123123;
temp->GPA = 3.1;
temp->next = head;
head = temp;
```

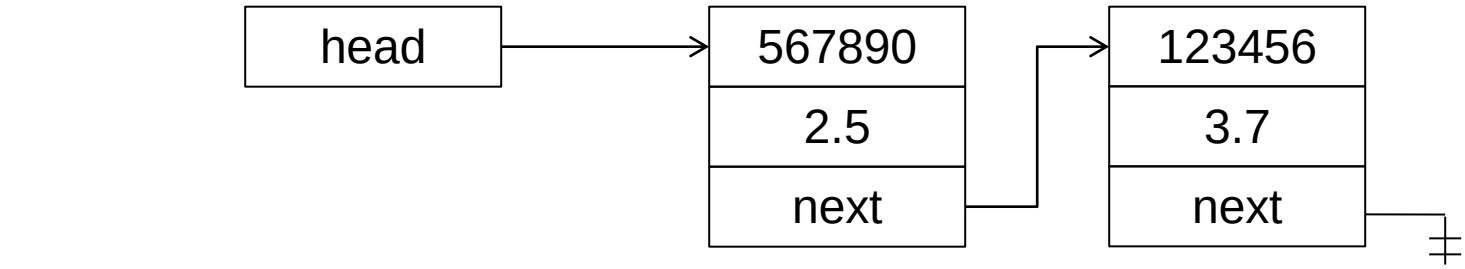


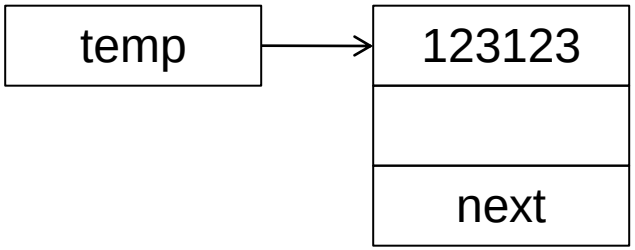
INSERT HEAD




```
Node *temp;
temp = new Node();
temp->ID = 123123;
temp->GPA = 3.1;
temp->next = head;
head = temp;
```

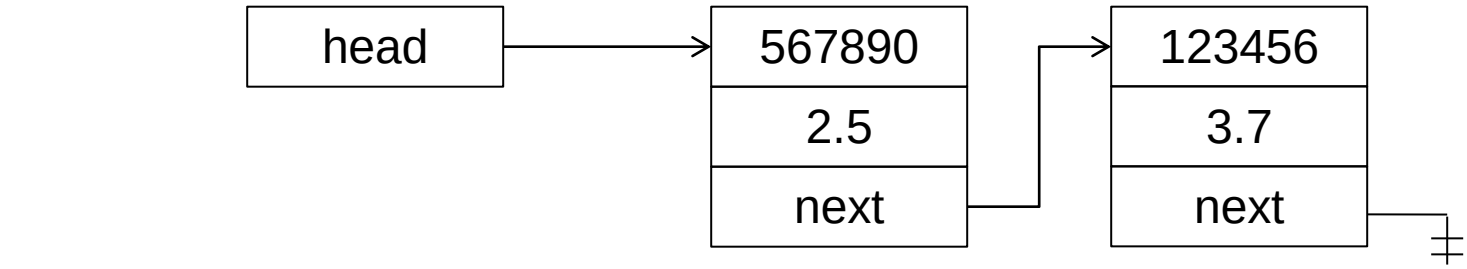
INSERT HEAD

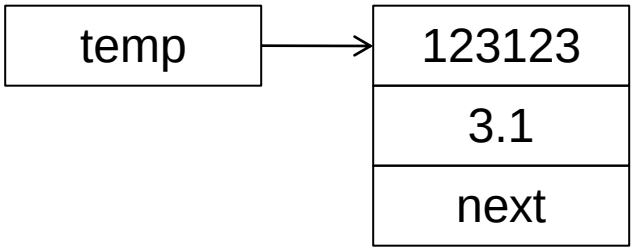





```
Node *temp;
temp = new Node();
temp->ID = 123123;
temp->GPA = 3.1;
temp->next = head;
head = temp;
```

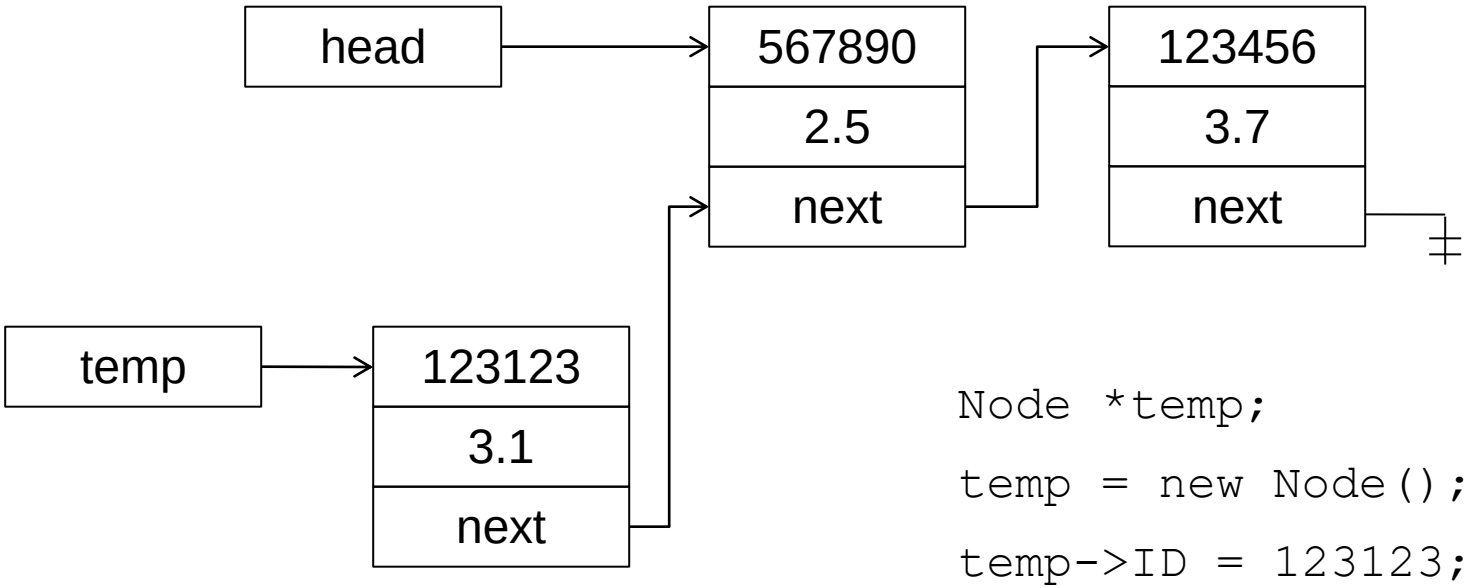
INSERT HEAD





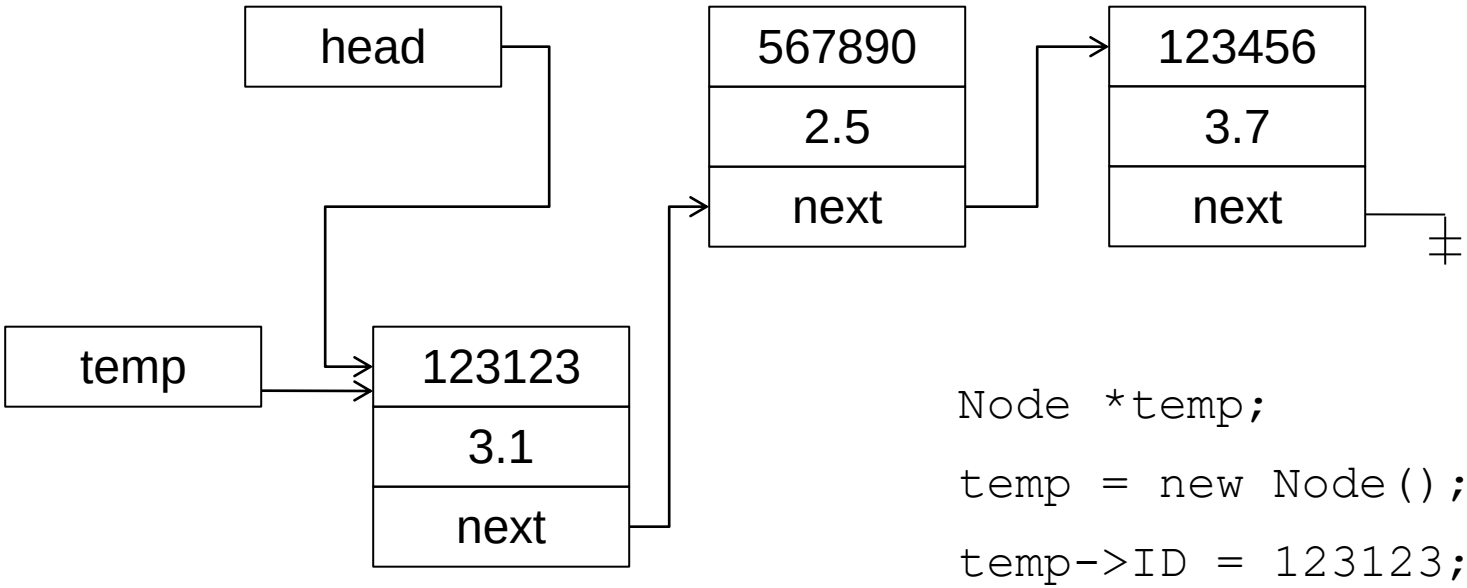

```
Node *temp;
temp = new Node();
temp->ID = 123123;
temp->GPA = 3.1;
temp->next = head;
head = temp;
```

INSERT HEAD




```
Node *temp;
temp = new Node();
temp->ID = 123123;
temp->GPA = 3.1;
temp->next = head;
head = temp;
```

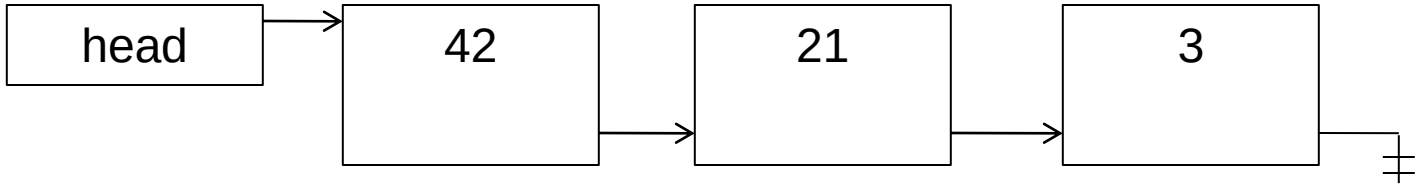
INSERT HEAD




```
Node *temp;
temp = new Node();
temp->ID = 123123;
temp->GPA = 3.1;
temp->next = head;
head = temp;
```

INSERT HEAD

- **When we do multiple insert head operations the data in the linked list is stored in reverse order**
  - The first value inserted will be at the end of the list
  - The last value inserted will be at the head of the list



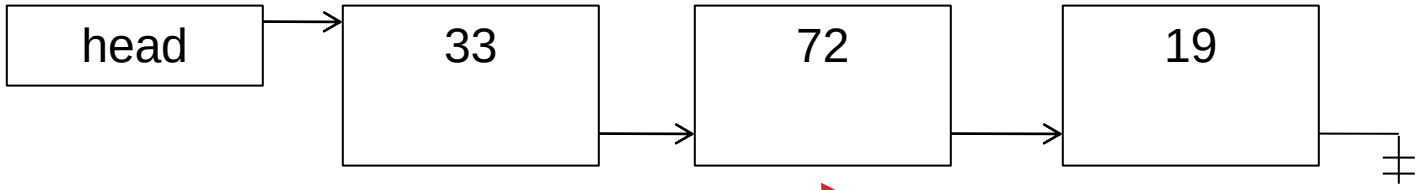

```
insert_head(3);
insert_head(21);
insert_head(42);
```

LINKED LISTS

OPERATION: INSERT TAIL

INSERT TAIL

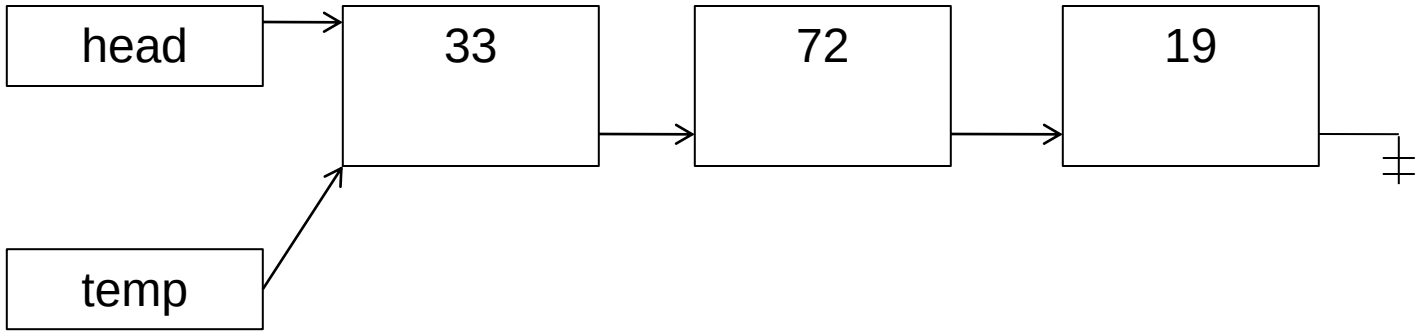
- **We can also insert data at the tail of the linked list**
  - Walk the linked list to find the last node
  - Create a new node and copy data into node
  - Assign pointer to connect the new node to linked list
  - We must handle the special case of an empty list




```
insert_tail(33);
insert_tail(72);
insert_tail(19);
```

Data is stored in same order it was inserted into the linked list

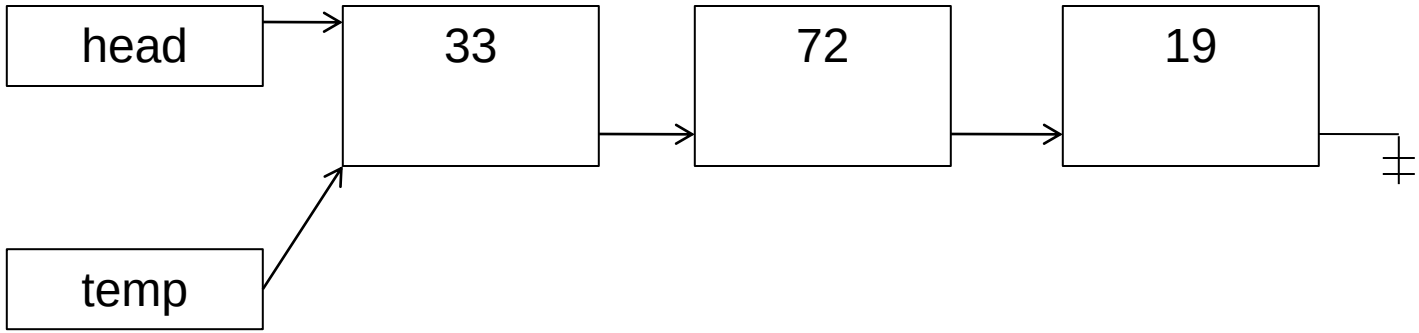
INSERT TAIL




```
// Walk the list to end
Node * temp = head;
while ((temp != NULL)&&
       (temp->next != NULL))
     temp = temp->next;
```

Set temp pointer to head of list

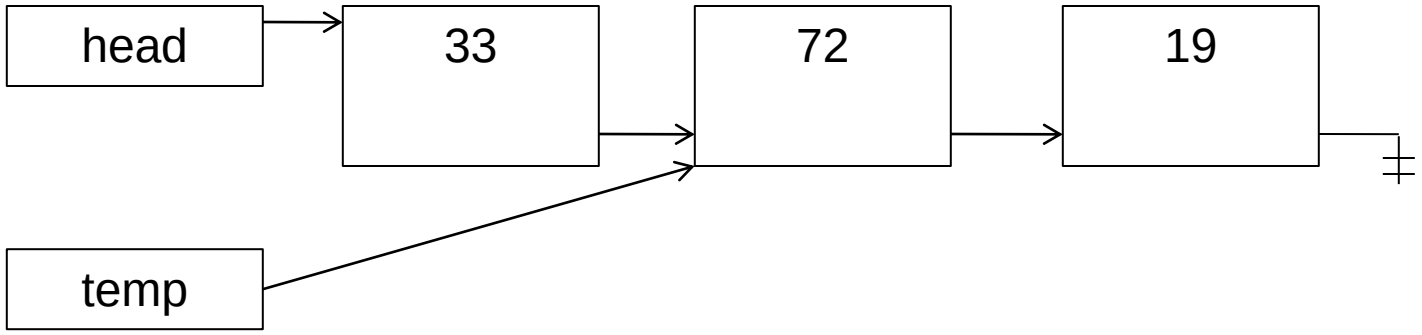
INSERT TAIL




```
// Walk the list to end
Node * temp = head;
while ((temp != NULL)&&
       (temp->next != NULL))
     temp = temp->next;
```

Check if temp pointer is not null and next pointer is not null

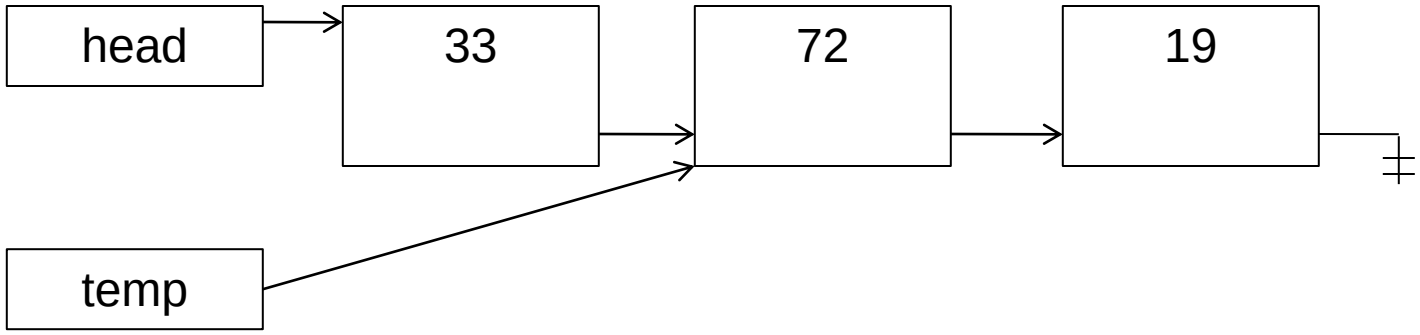
INSERT TAIL




```
// Walk the list to end
Node * temp = head;
while ((temp != NULL)&&
       (temp->next != NULL))
     temp = temp->next;
```

← Goes to 2nd node in the list

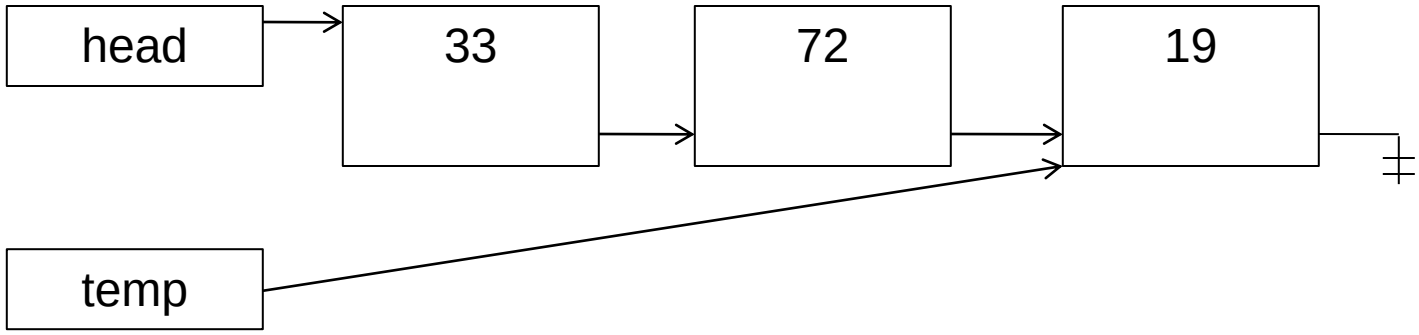
INSERT TAIL




```
// Walk the list to end
Node * temp = head;
while ((temp != NULL)&&
       (temp->next != NULL))
     temp = temp->next;
```

Check if temp pointer is not null and next pointer is not null

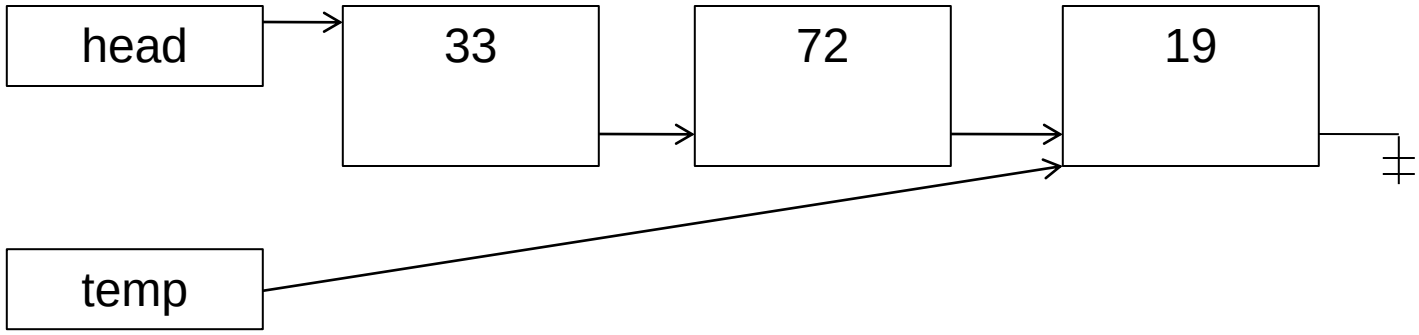
INSERT TAIL




```
// Walk the list to end
Node * temp = head;
while ((temp != NULL)&&
       (temp->next != NULL))
     temp = temp->next;
```

← Goes to 3rd node in the list

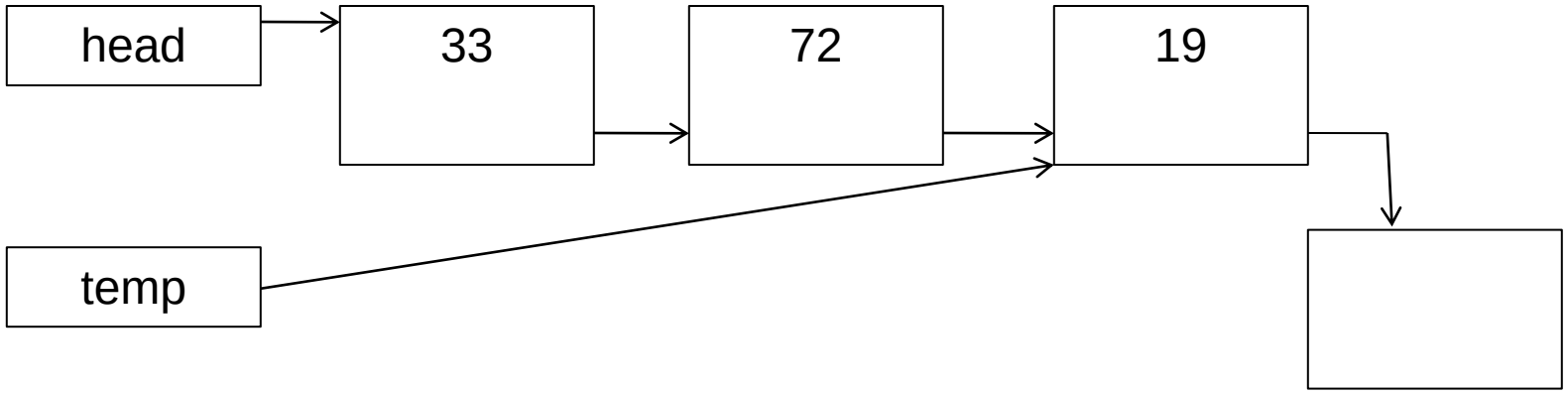
INSERT TAIL




```
// Walk the list to end
Node * temp = head;
while ((temp != NULL)&&
       (temp->next != NULL))
     temp = temp->next;
```

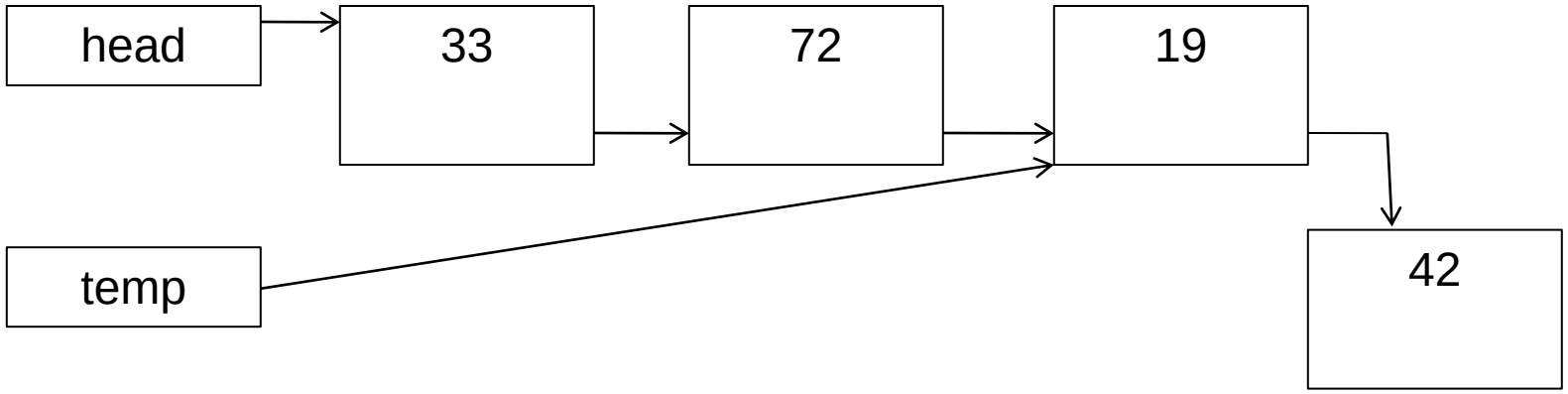
We stop when we reach last node because next is null

INSERT TAIL



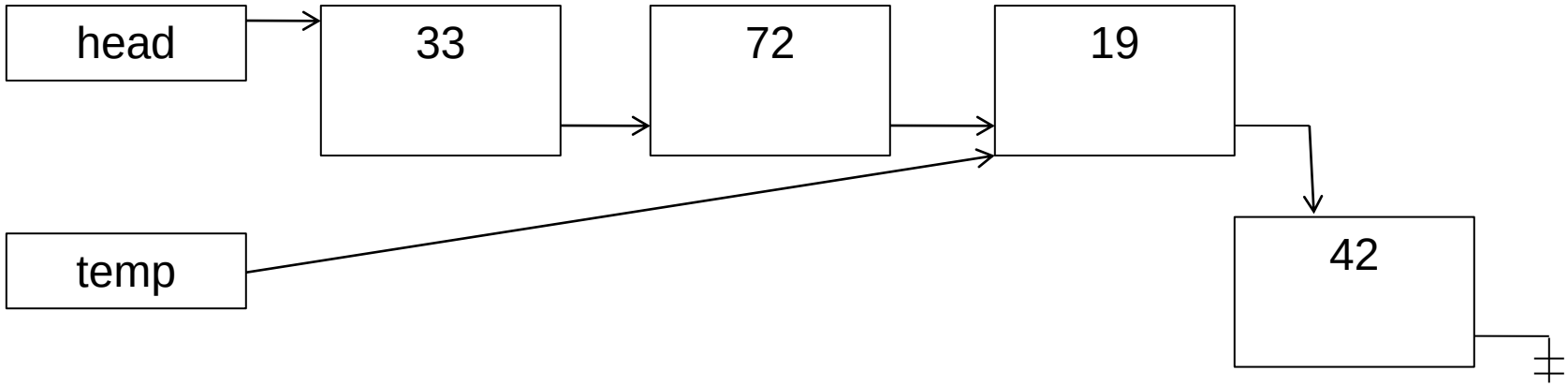

```
// Insert at end of list
if (temp != NULL)
{
      temp->next = new Node();
      temp->next->num = 42;
      temp->next->next = NULL;
}
```

INSERT TAIL



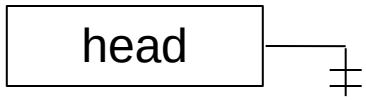

```
// Insert at end of list
if (temp != NULL)
{
      temp->next = new Node();
      temp->next->num = 42;
      temp->next->next = NULL;
}
```

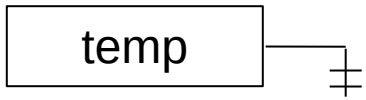
INSERT TAIL




```
// Insert at end of list
if (temp != NULL)
{
      temp->next = new Node();
      temp->next->num = 42;
      temp->next->next = NULL;
}
```

INSERT TAIL

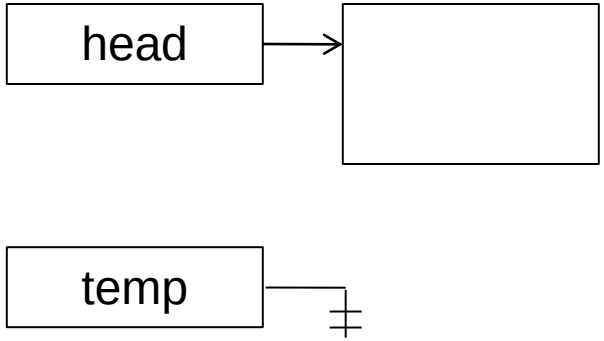





```
// Insert into empty list
if (temp == NULL)
{
      head = new Node();
      head->num = 42;
      head->next = NULL;
}
```

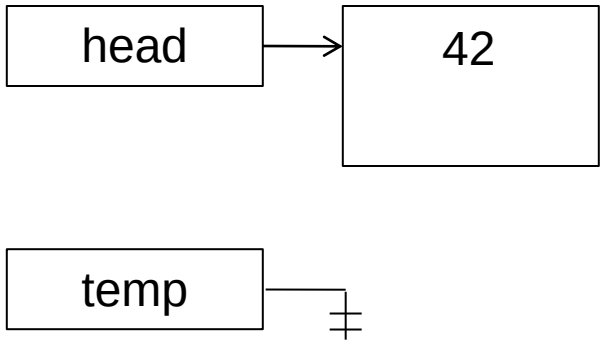
True when inserting into an empty linked list

INSERT TAIL



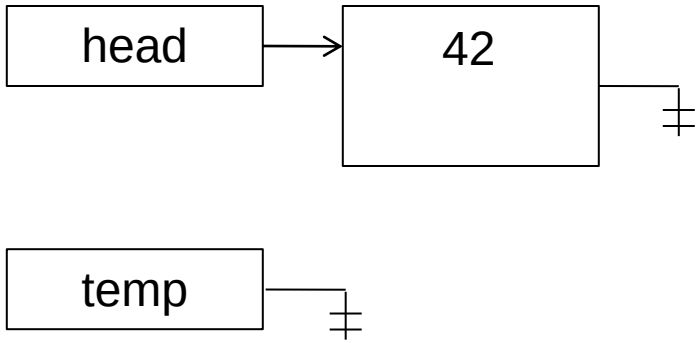

```
// Insert into empty list
if (temp == NULL)
{
      head = new Node();
      head->num = 42;
      head->next = NULL;
}
```

INSERT TAIL




```
// Insert into empty list
if (temp == NULL)
{
      head = new Node();
      head->num = 42;
      head->next = NULL;
}
```

INSERT TAIL




```
// Insert into empty list
if (temp == NULL)
{
      head = new Node();
      head->num = 42;
      head->next = NULL;
}
```

LINKED LISTS

OPERATION: PRINT LIST

PRINT LIST

- **To print all the data in a linked list we must start a the "head" and walk the list by following "next" pointers**

```
Node *ptr = head;
while (ptr != NULL)
{
  cout << ptr->ID << endl;
  cout << ptr->GPA << endl;
  ptr = ptr->next;
}
```


Create a pointer to head of list

PRINT LIST

- **To print all the data in a linked list we must start a the "head" and walk the list by following "next" pointers**

```
Node *ptr = head;
while (ptr != NULL)
{
  cout << ptr->ID << endl;
  cout << ptr->GPA << endl;
  ptr = ptr->next;
}
```

Loop until pointer is reaches the end of the linked list

PRINT LIST

- **To print all the data in a linked list we must start a the "head" and walk the list by following "next" pointers**

```
Node *ptr = head;
while (ptr != NULL)
{
  cout << ptr->ID << endl;
  cout << ptr->GPA << endl;
  ptr = ptr->next;
}
```

Print out Node data

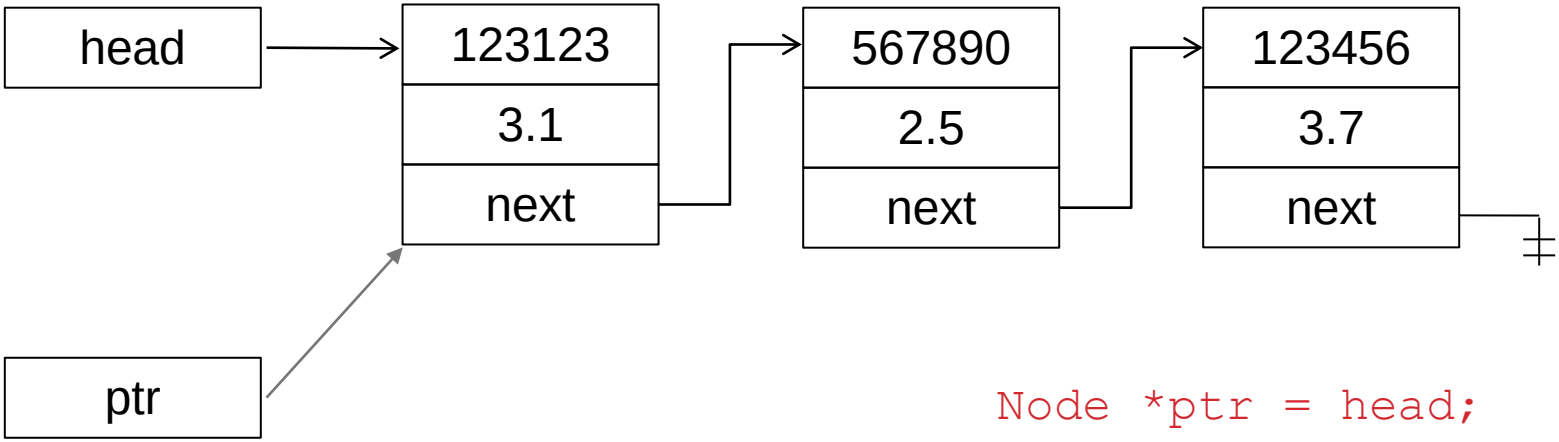
PRINT LIST

- **To print all the data in a linked list we must start a the "head" and walk the list by following "next" pointers**

```
Node *ptr = head;
while (ptr != NULL)
{
  cout << ptr->ID << endl;
  cout << ptr->GPA << endl;
  ptr = ptr->next;
}
```

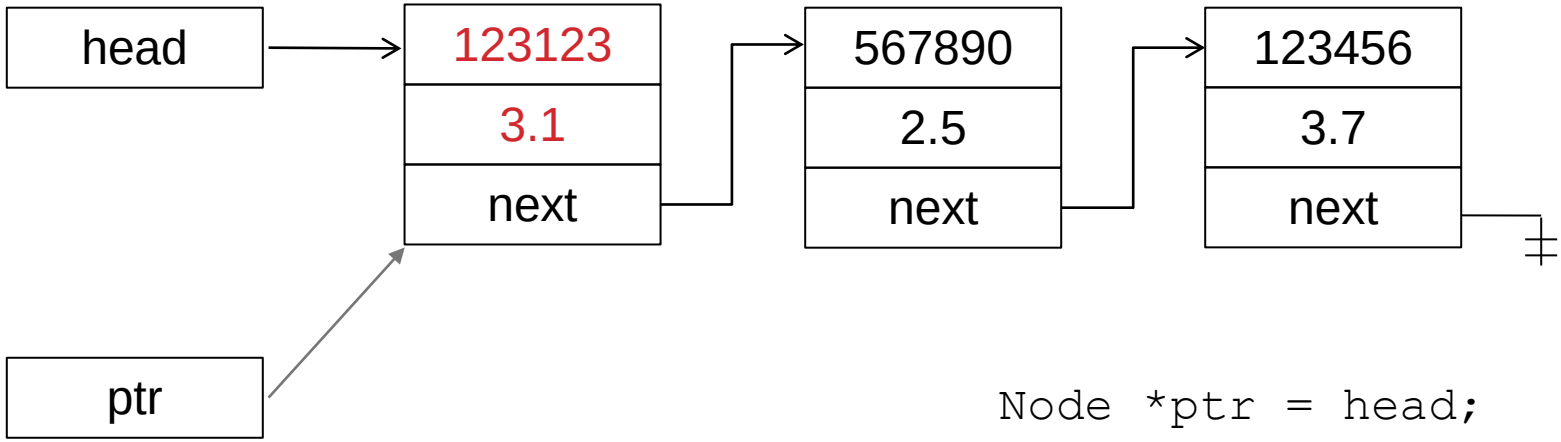
Go to the next Node in the list

PRINT LIST



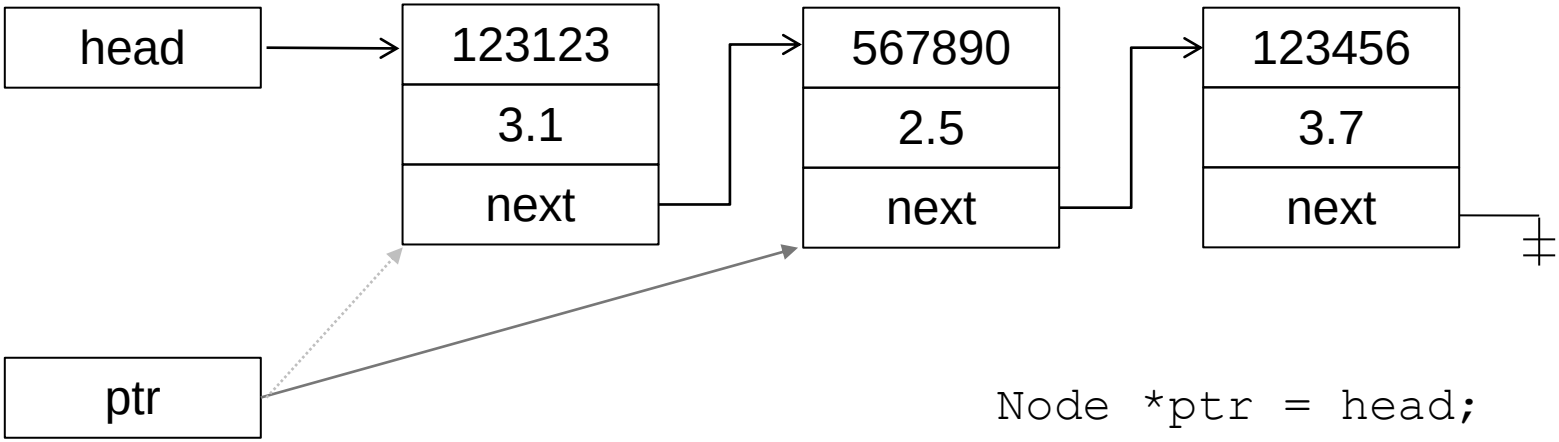

```
Node *ptr = head;
while (ptr != NULL)
{
   cout << ptr->ID << endl;
   cout << ptr->GPA << endl;
   ptr = ptr->next;
}
```

PRINT LIST



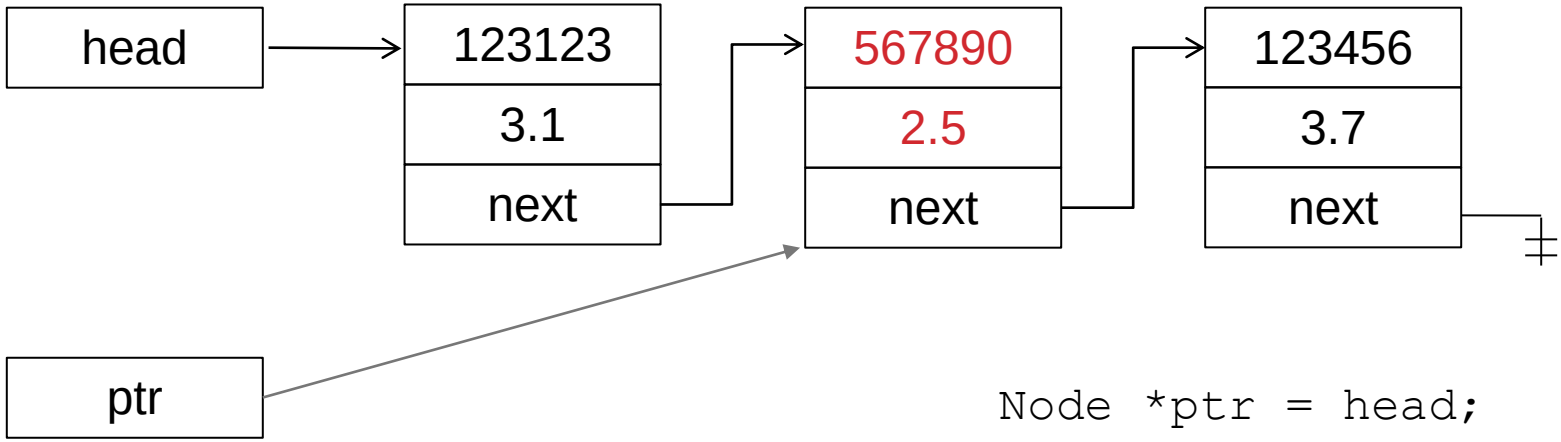

```
Node *ptr = head;
while (ptr != NULL)
{
   cout << ptr->ID << endl;
   cout << ptr->GPA << endl;
   ptr = ptr->next;
}
```

PRINT LIST



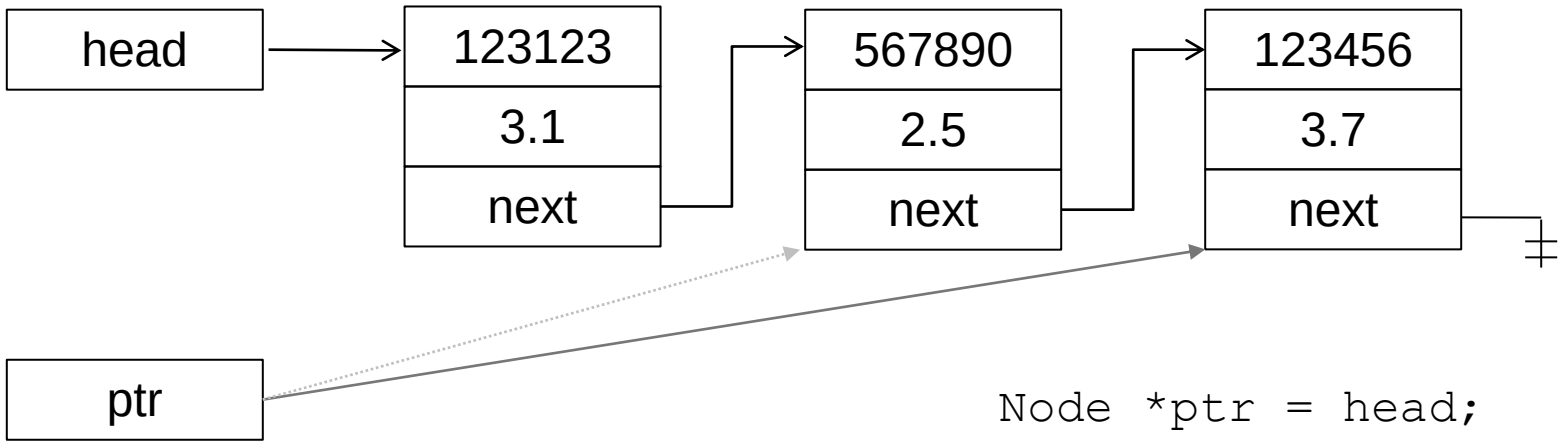

```
Node *ptr = head;
while (ptr != NULL)
{
   cout << ptr->ID << endl;
   cout << ptr->GPA << endl;
   ptr = ptr->next;
}
```

PRINT LIST



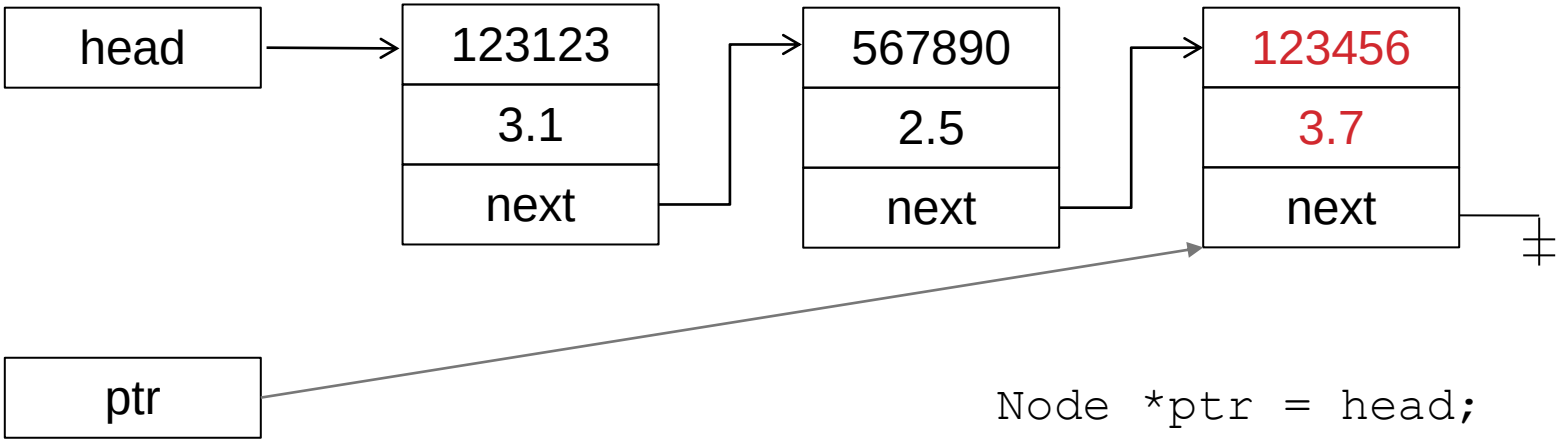

```
Node *ptr = head;
while (ptr != NULL)
{
   cout << ptr->ID << endl;
   cout << ptr->GPA << endl;
   ptr = ptr->next;
}
```

PRINT LIST



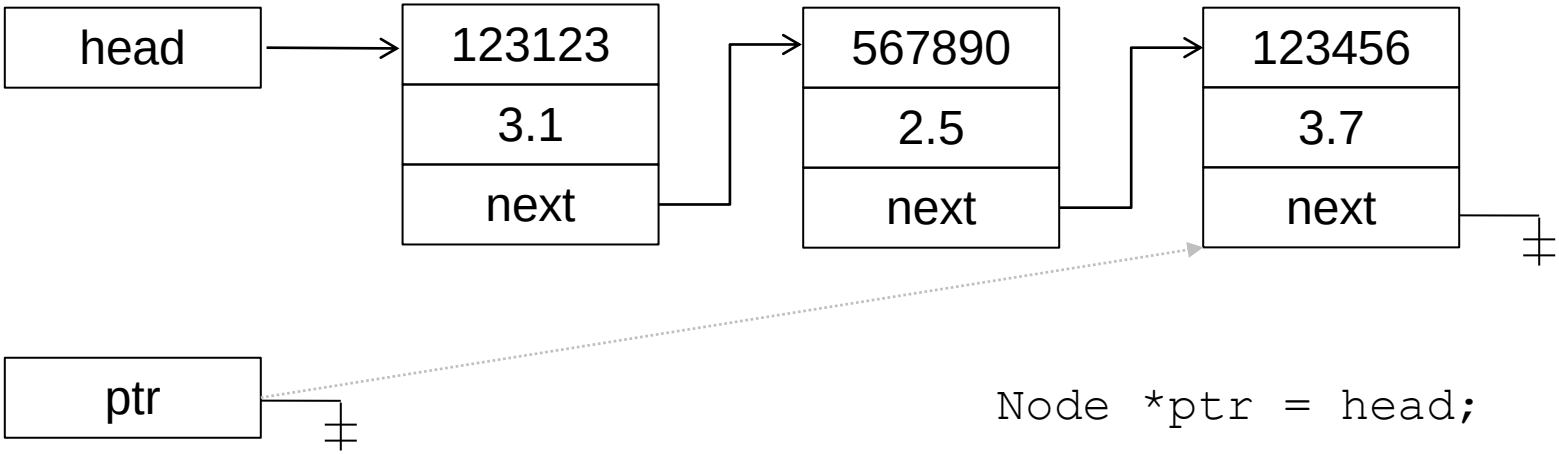

```
Node *ptr = head;
while (ptr != NULL)
{
   cout << ptr->ID << endl;
   cout << ptr->GPA << endl;
   ptr = ptr->next;
}
```

PRINT LIST



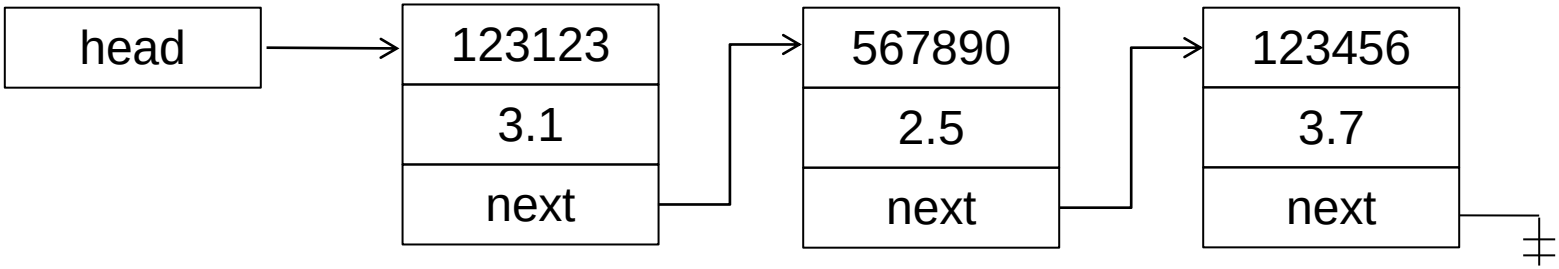

```
Node *ptr = head;
while (ptr != NULL)
{
   cout << ptr->ID << endl;
   cout << ptr->GPA << endl;
   ptr = ptr->next;
}
```

PRINT LIST




```
Node *ptr = head;
while (ptr != NULL)
{
   cout << ptr->ID << endl;
   cout << ptr->GPA << endl;
   ptr = ptr->next;
}
```

PRINT LIST



The loop stops
because ptr is NULL

```
Node *ptr = head;
while (ptr != NULL)
{
   cout << ptr->ID << endl;
   cout << ptr->GPA << endl;
   ptr = ptr->next;
}
```

PRINT LIST

- **A linked list can also be printed with a recursive function**

```
void print_list(Node *ptr)
{
   if (ptr != NULL)
   {
     cout << ptr->ID << endl;
     cout << ptr->GPA << endl;
     print_list(ptr->next);
   }
}
...
print_list(head)
```

We pass in pointer to first node

PRINT LIST

- **A linked list can also be printed with a recursive function**

```
void print_list(Node *ptr)
{
   if (ptr != NULL)
   {
     cout << ptr->ID << endl;
     cout << ptr->GPA << endl;
     print_list(ptr->next);
   }
}
...
print_list(head)
```

Terminating condition stops printing at end of the linked list

PRINT LIST

- **A linked list can also be printed with a recursive function**

```
void print_list(Node *ptr)
{
   if (ptr != NULL)
   {
     cout << ptr->ID << endl;
     cout << ptr->GPA << endl;
     print_list(ptr->next);
   }
}
...
print_list(head)
```

Print data in the first node of this linked list

PRINT LIST

- **A linked list can also be printed with a recursive function**

```
void print_list(Node *ptr)
{
   if (ptr != NULL)
   {
     cout << ptr->ID << endl;
     cout << ptr->GPA << endl;
     print_list(ptr->next);
   }
}
...
print_list(head)
```


Call function recursively with pointer to the next node to print the rest of linked list

PRINT LIST

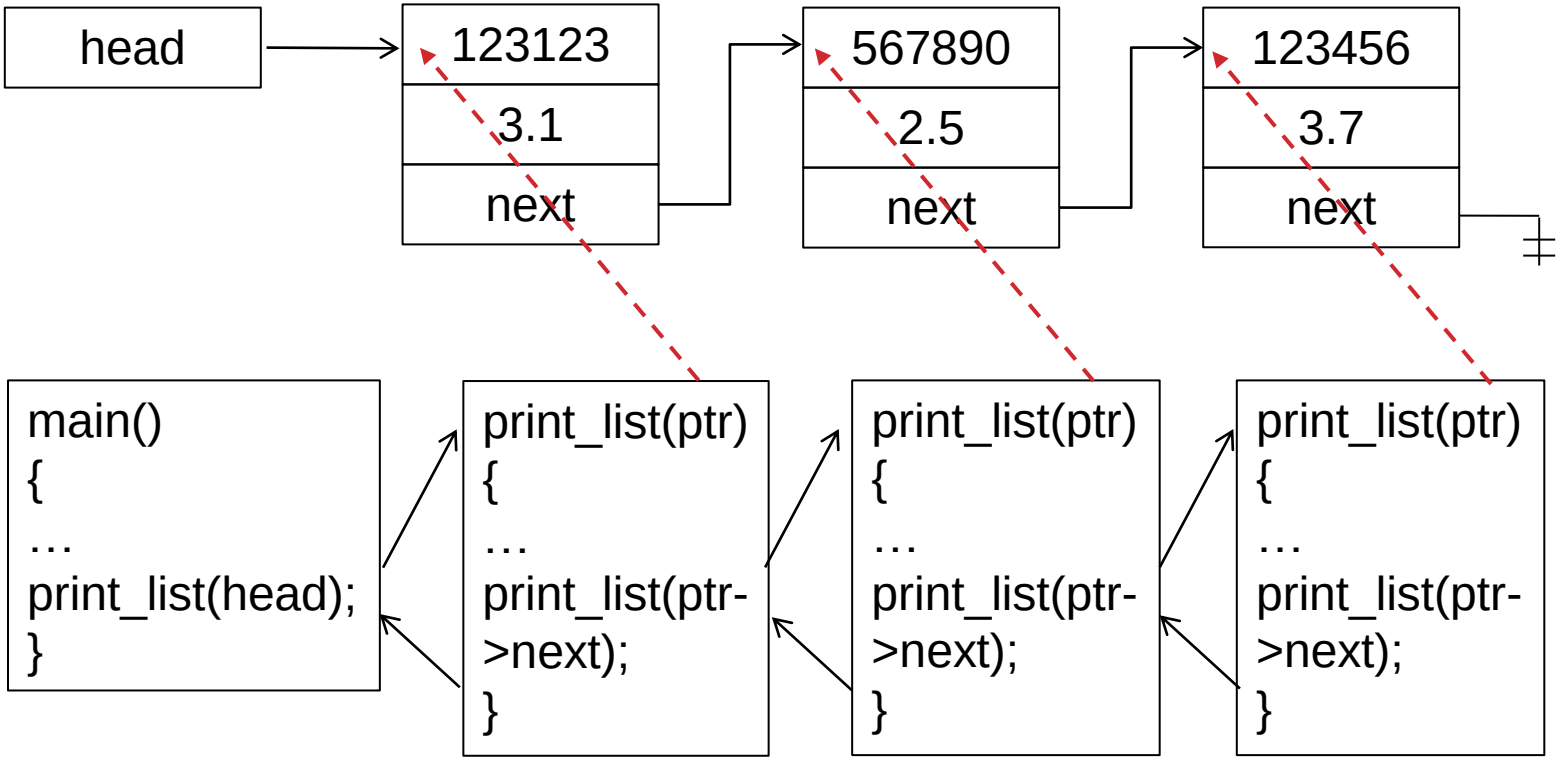
- **A linked list can also be printed with a recursive function**

```
void print_list(Node *ptr)
{
   if (ptr != NULL)
   {
     cout << ptr->ID << endl;
     cout << ptr->GPA << endl;
     print_list(ptr->next);
   }
}
...
print_list(head);
```

Call the function in main with head of the linked list

PRINT LIST

- **Box method trace of recursive print_list**



PRINT LIST

- **A linked list can also be printed in reverse order**

```
void print_list(Node *ptr)
{
   if (ptr != NULL)
   {
     print_list(ptr->next);
     cout << ptr->ID << endl;
     cout << ptr->GPA << endl;
   }
}
...
print_list(head)
```

Moving the recursive call here will print the linked list in reverse order because we print information after returning from the recursive function call

LINKED LISTS

OPERATION: SEARCH

SEARCH LIST

- **The goal of search is to "walk the linked list" starting at the head to see if a specific value is stored in the list**

```
// Walk the list to desired value
Node * temp = head;
while ((temp != NULL)&&
       (temp->num != value))
     temp = temp->next;
if (temp != NULL)
   cout << value << "was found\n";
```

Create a pointer to head of list

SEARCH LIST

- **The goal of search is to “walk the linked list” starting at the head to see if a specific value is stored in the list**

```
// Walk the list to desired value
Node * temp = head;
while ((temp != NULL)&&
       (temp->num != value))
     temp = temp->next;
if (temp != NULL)
   cout << value << “was found\n”;
```

Check if pointer is not null and desired value is not found

SEARCH LIST

- **The goal of search is to "walk the linked list" starting at the head to see if a specific value is stored in the list**

```
// Walk the list to desired value
Node * temp = head;
while ((temp != NULL)&&
       (temp->num != value))
     temp = temp->next;
if (temp != NULL)
   cout << value << "was found\n";
```

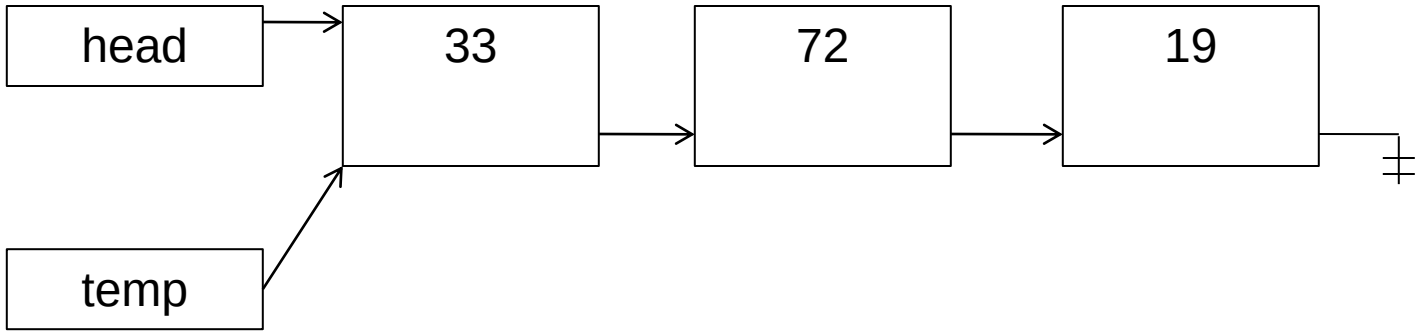
SEARCH LIST

- **The goal of search is to “walk the linked list” starting at the head to see if a specific value is stored in the list**

```
// Walk the list to desired value
Node * temp = head;
while ((temp != NULL)&&
       (temp->num != value))
     temp = temp->next;
if (temp != NULL)
   cout << value << “was found\n”;
```

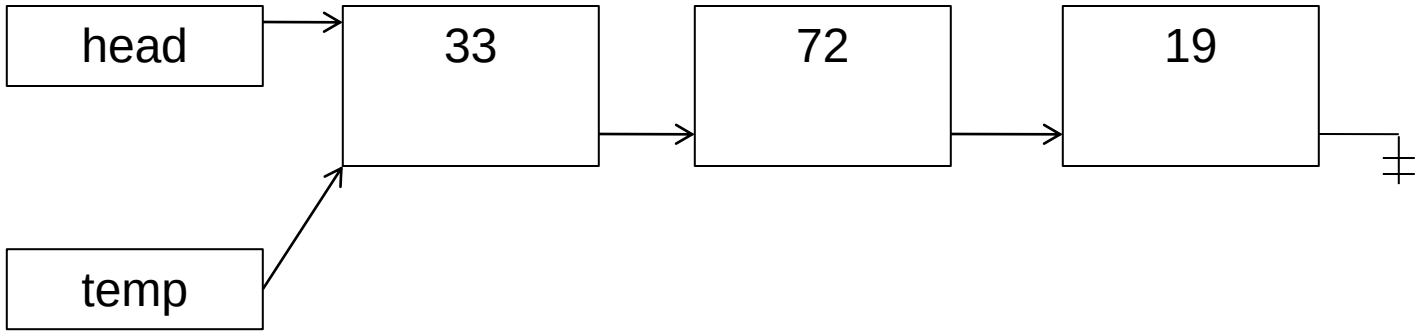
Print message if found

SEARCH LIST




```
// Walk the list to desired value
Node * temp = head;
while ((temp != NULL)&&
       (temp->value != 72))
     temp = temp->next;
```

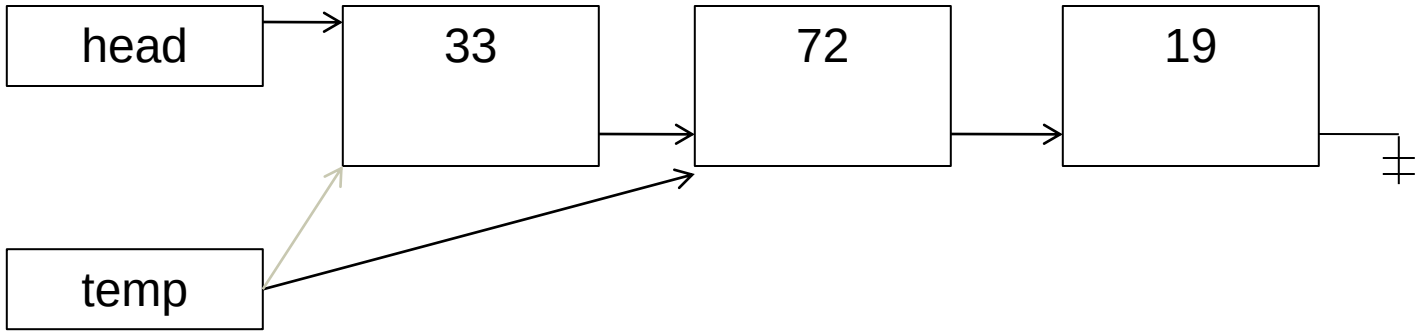
SEARCH LIST




```
// Walk the list to desired value
Node * temp = head;
while ((temp != NULL)&&
       (temp->value != 72))
     temp = temp->next;
```

← True so we continue

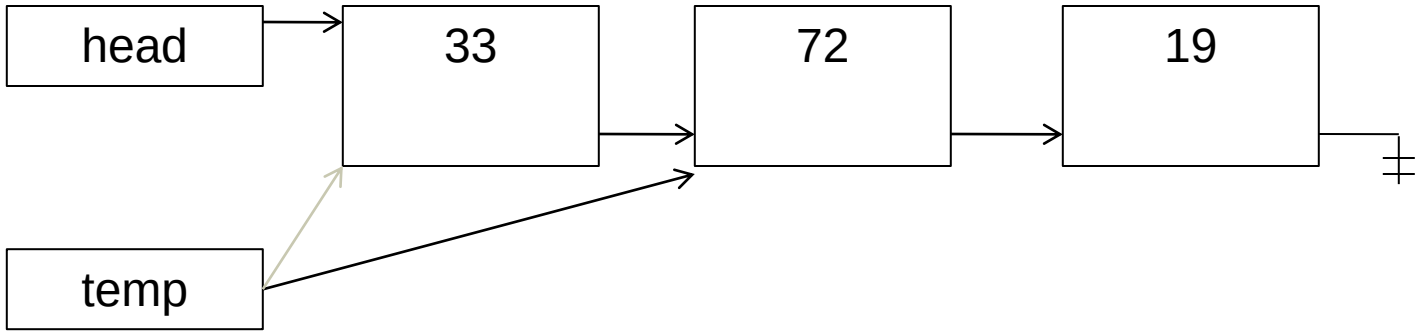
SEARCH LIST




```
// Walk the list to desired value
Node * temp = head;
while ((temp != NULL)&&
       (temp->value != 72))
     temp = temp->next;
```

Go to next node

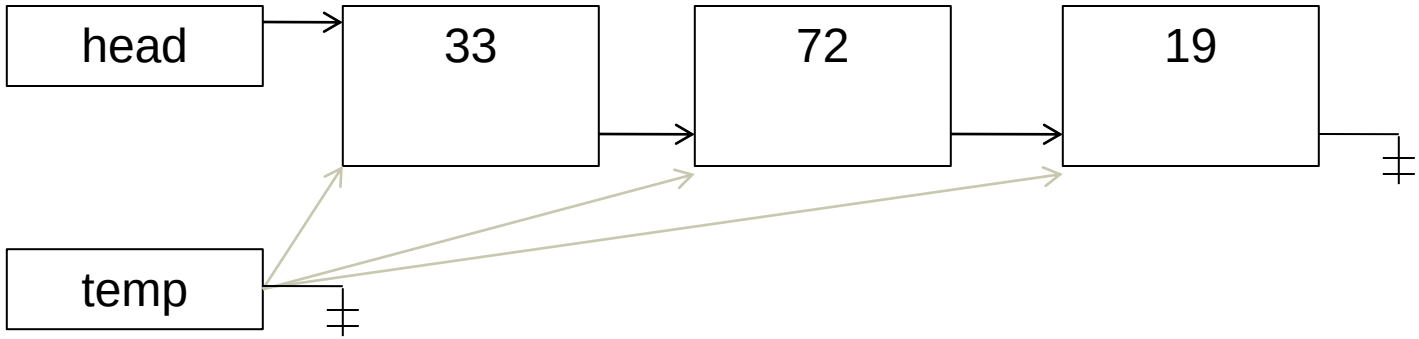
SEARCH LIST




```
// Walk the list to desired value
Node * temp = head;
while ((temp != NULL)&&
       (temp->value != 72))
     temp = temp->next;
```

Value equals 72 so this while loop stops

SEARCH LIST




```
// Walk the list to desired value
Node * temp = head;
while ((temp != NULL)&&
       (temp->value != 101))
     temp = temp->next;
```

What happens if data is not found in the linked list?

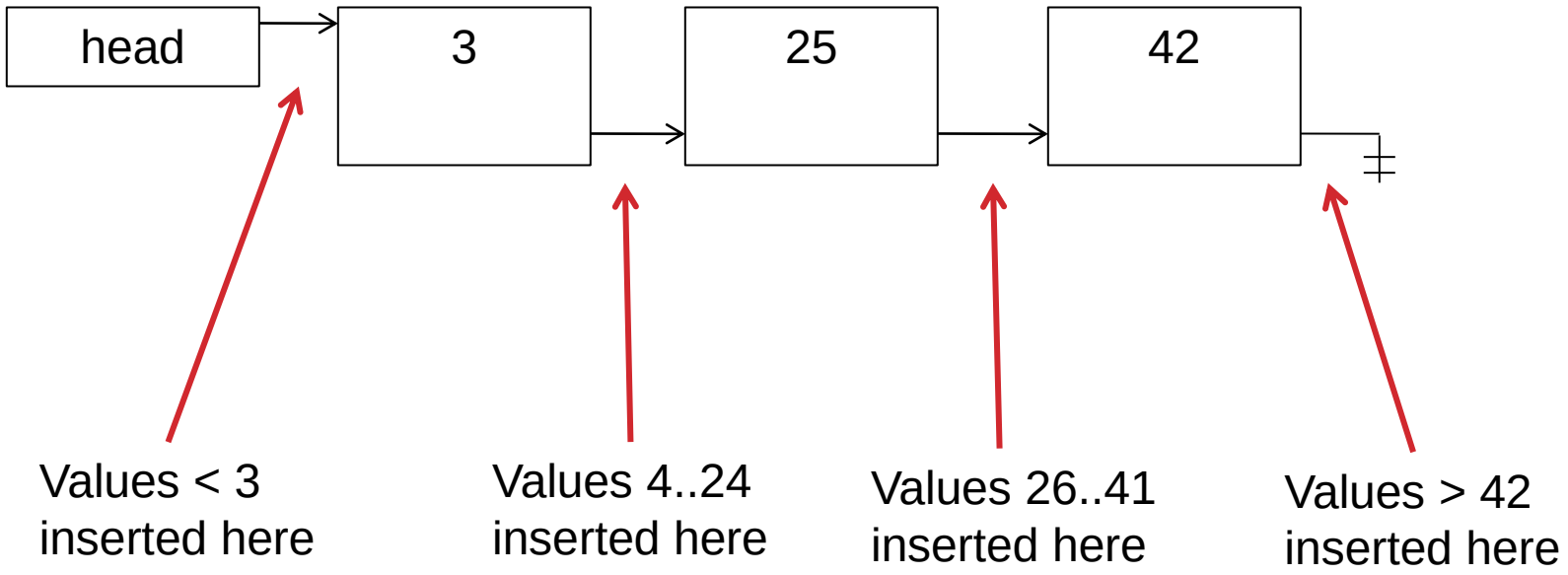
We will loop over whole list and stop when temp is NULL

LINKED LISTS

OPERATION: SORTED INSERT

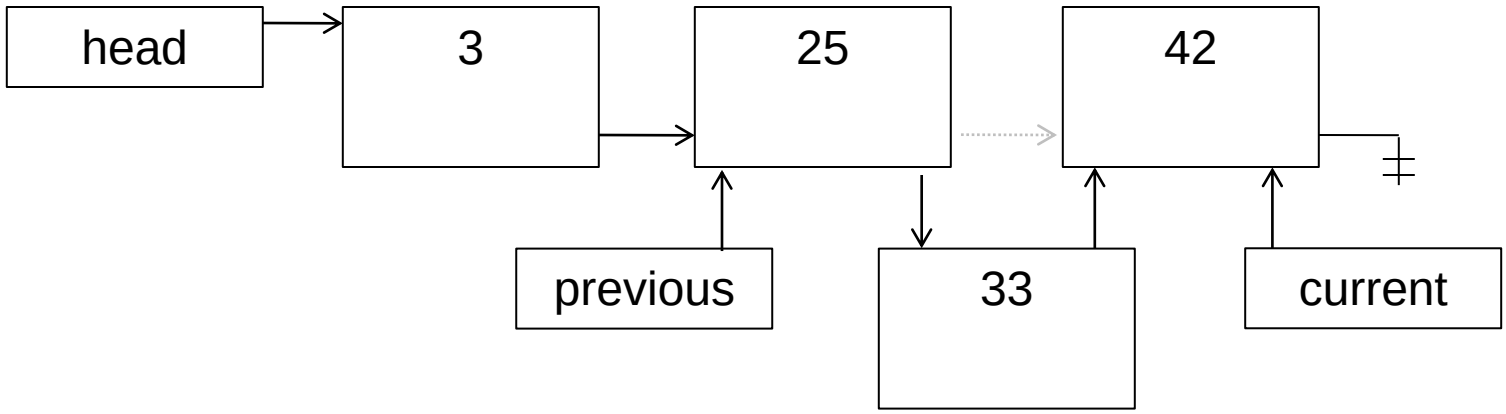
SORTED INSERT

- **The goal of sorted insert is to insert nodes into the list so data is always sorted in ascending (or descending) order**



SORTED INSERT

- **Pseudo code algorithm**
  - Start with two pointers previous and current to head node
  - Walk list until current value is greater than insertion value
  - Previous value will be is less than insertion value
  - Create a new node and store the insertion value
  - Connect new node to list between previous and current



SORTED INSERT

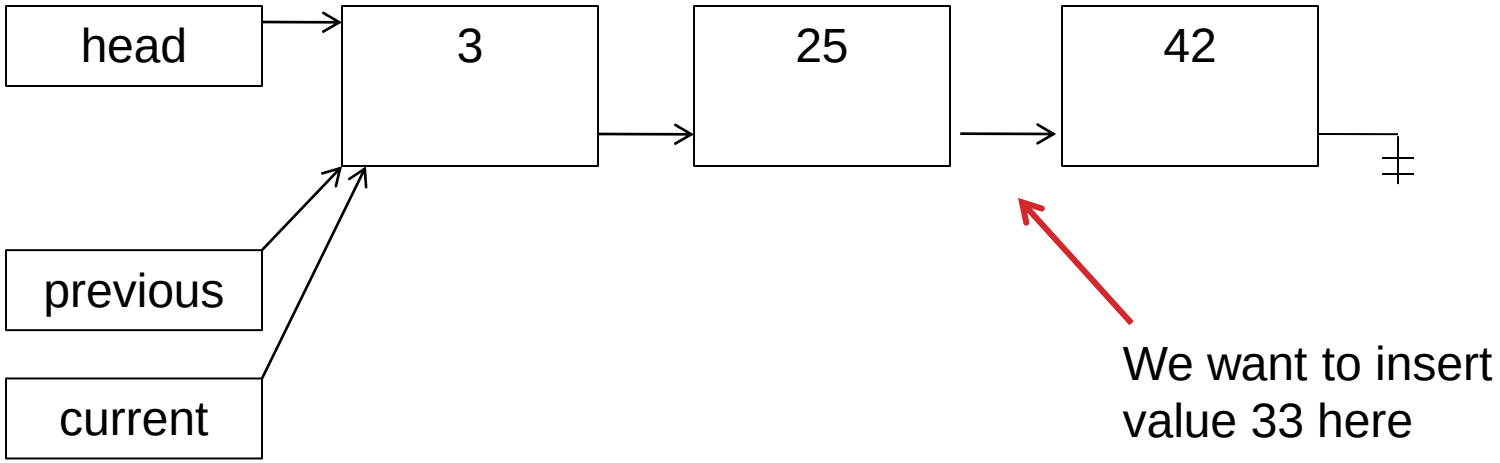
- **Implementation of sorted insert**

```
// Walk the list to insertion point
Node *curr = head;
node *prev = head;
while ((curr != NULL))&&(curr->num < value))
{
     prev = curr;
     curr = curr->next;
}
```

Stops loop when curr is NULL or num >= value

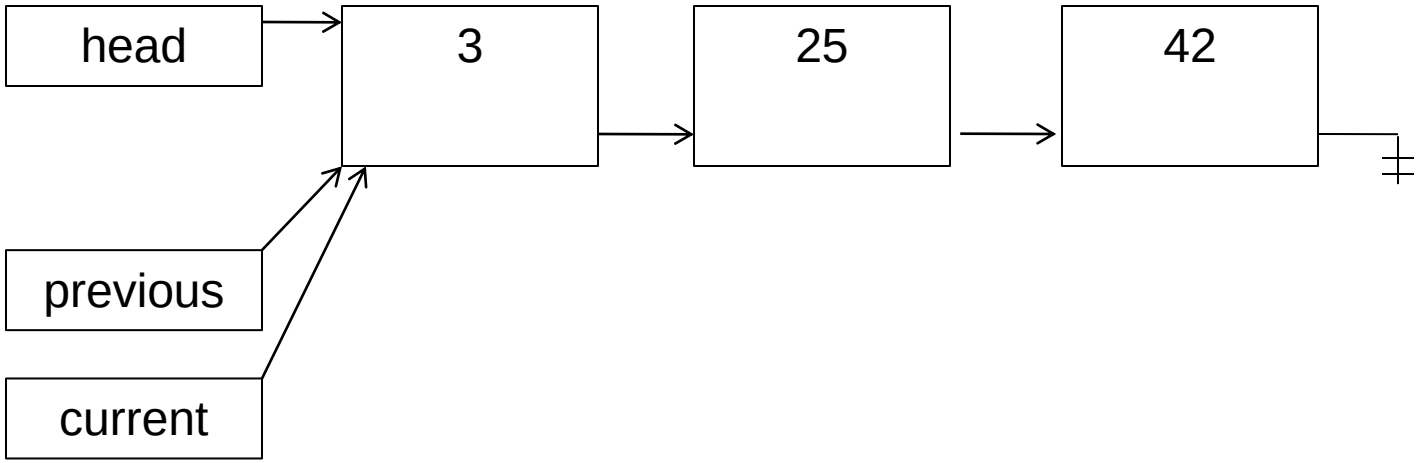
Moves both pointers one Node down the linked list with prev one node behind curr

SORTED INSERT



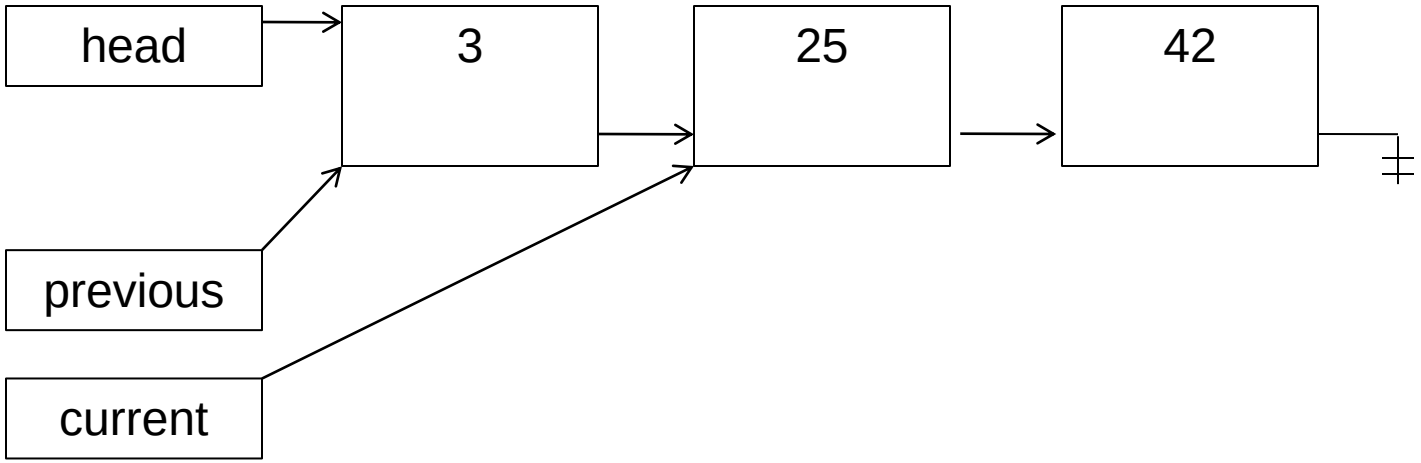

```
// Walk the list to insertion point
Node *curr = head;
node *prev = head;
while ((curr != NULL))&&(curr->num < value))
{
      prev = curr;
      curr = curr->next;
}
```

SORTED INSERT



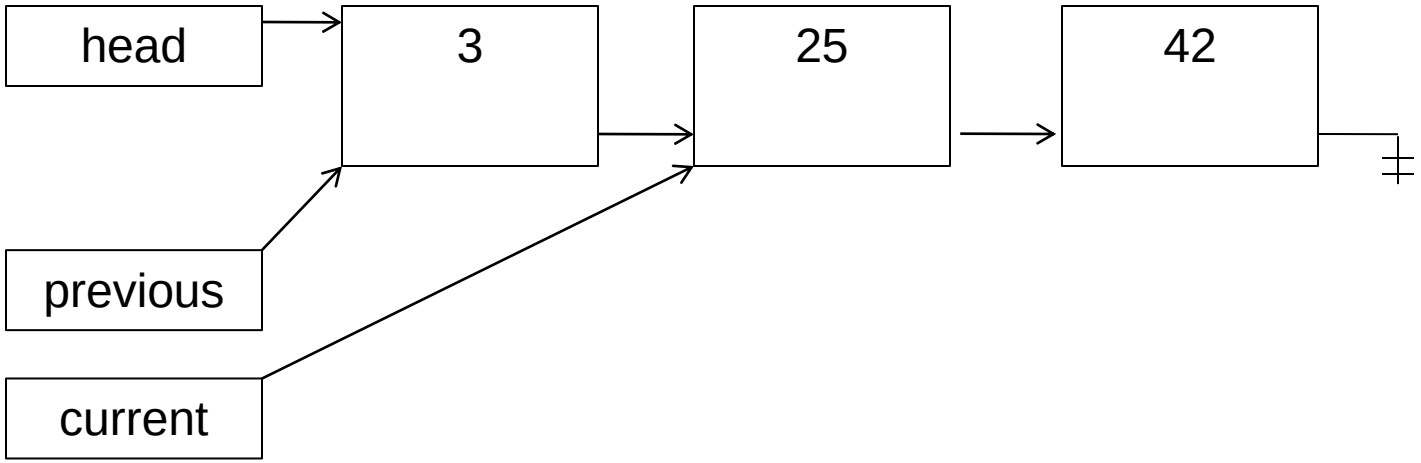

```
// Walk the list to insertion point
Node *curr = head;
node *prev = head;
while ((curr != NULL))&&(curr->num < value))   // TRUE
{
      prev = curr;
      curr = curr->next;
}
```

SORTED INSERT



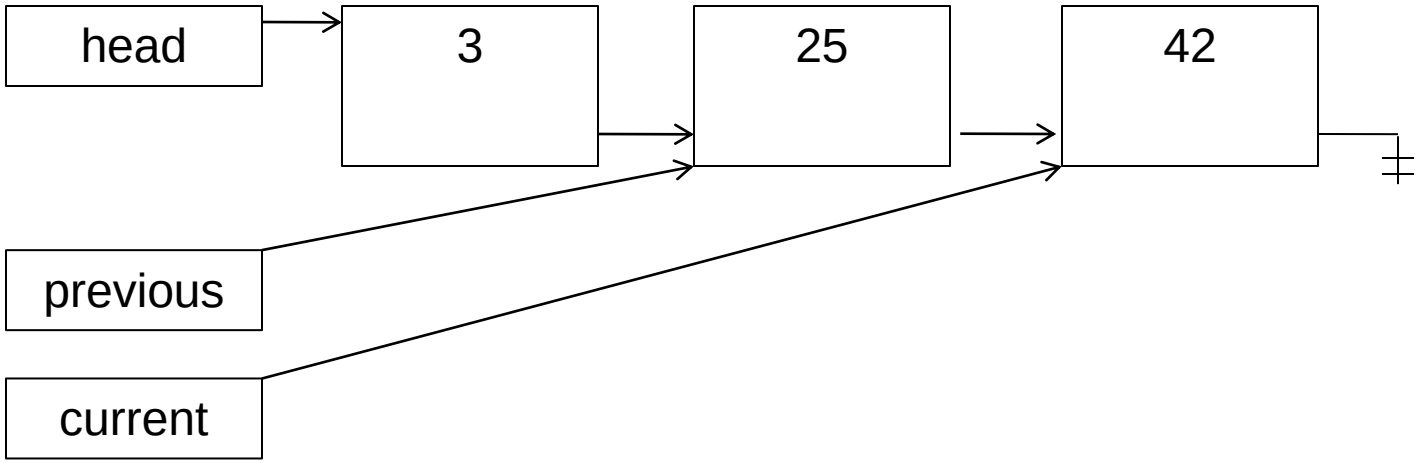

```
// Walk the list to insertion point
Node *curr = head;
node *prev = head;
while ((curr != NULL))&&(curr->num < value))
{
      prev = curr;
      curr = curr->next;
}
```

SORTED INSERT



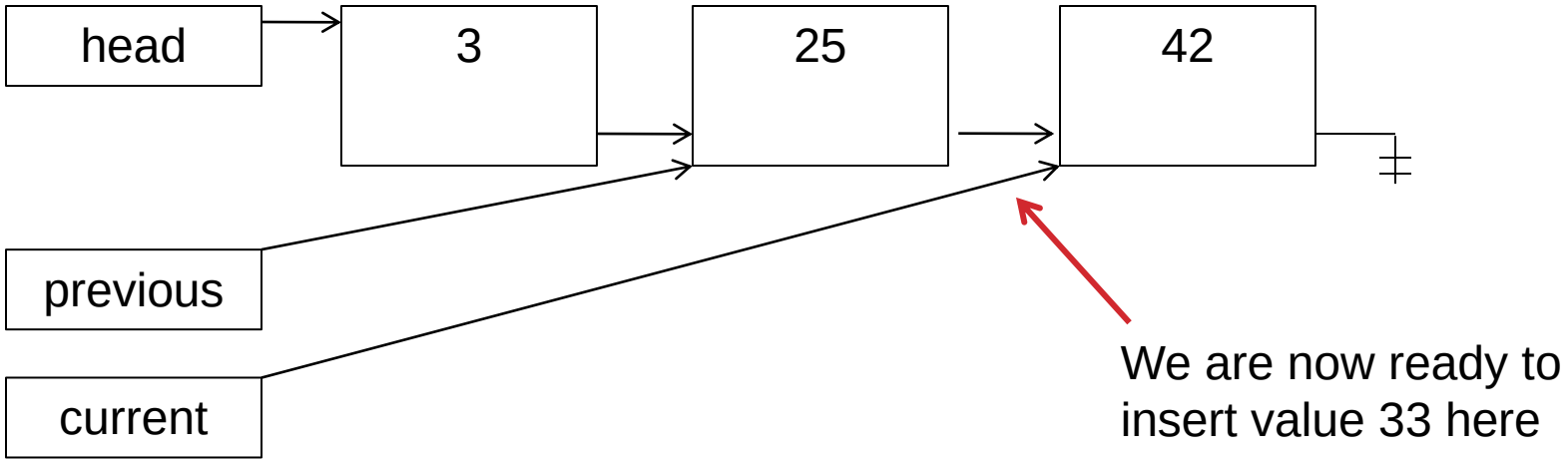

```
// Walk the list to insertion point
Node *curr = head;
node *prev = head;
while ((curr != NULL))&&(curr->num < value)) // TRUE
{
      prev = curr;
      curr = curr->next;
}
```

SORTED INSERT




```
// Walk the list to insertion point
Node *curr = head;
node *prev = head;
while ((curr != NULL))&&(curr->num < value))
{
      prev = curr;
      curr = curr->next;
}
```

SORTED INSERT

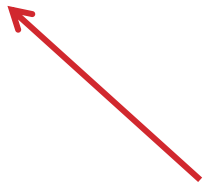



```
// Walk the list to insertion point
Node *curr = head;
node *prev = head;
while ((curr != NULL))&&(curr->num < value)) // FALSE
{
      prev = curr;
      curr = curr->next;
}
```

SORTED INSERT

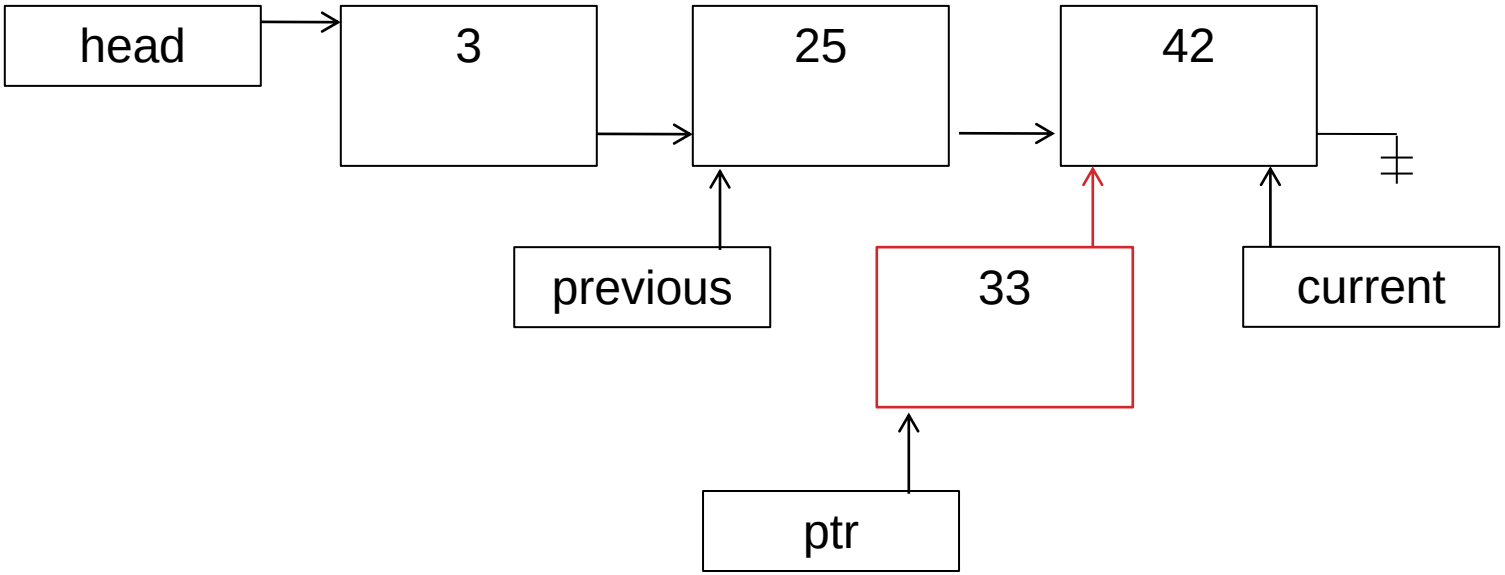
- **Implementation of sorted insert**

```
// Create new node and connect to current node
Node *ptr = new Node();
ptr->num = value;
ptr->next = curr;
```



We know the value in the new node is less than the value in the current node, so we set the next pointer to the current node

SORTED INSERT



SORTED INSERT

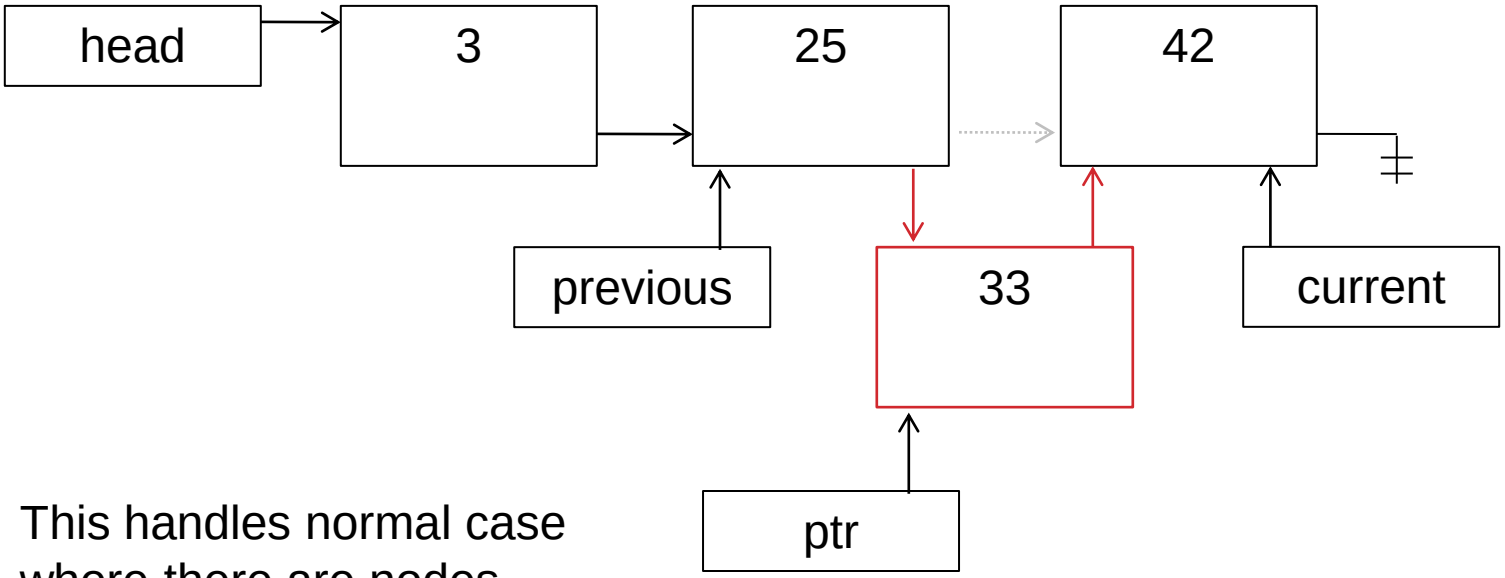
- **Implementation of sorted insert**

```
// Connect previous node to new node
if (prev == head)
   head = ptr;
else
   prev->next = ptr;
```

This handles special case where new node will be at the head of the list

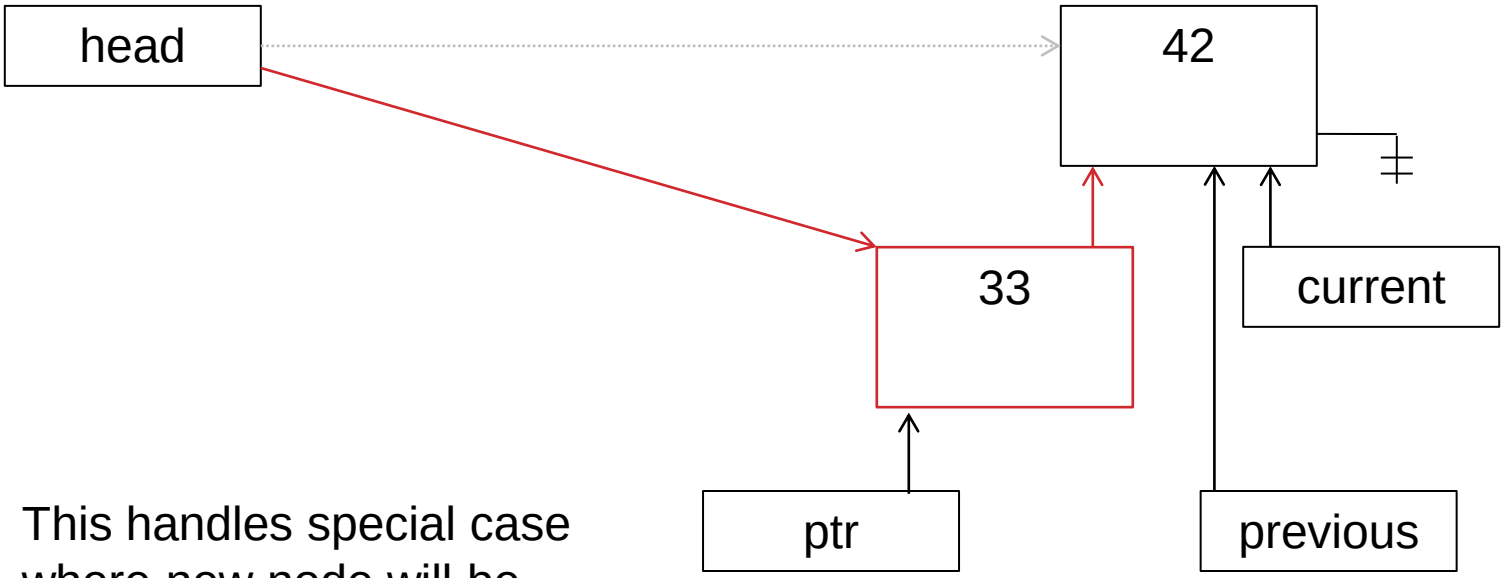
This handles normal case where there are nodes before inserted node

SORTED INSERT



This handles normal case where there are nodes before inserted node

SORTED INSERT



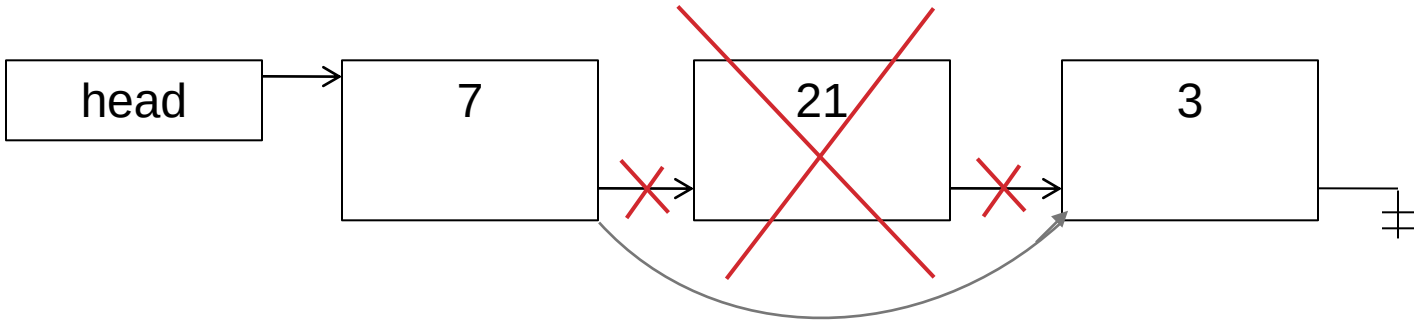
This handles special case where new node will be at the head of the list

LINKED LISTS

OPERATION: DELETE NODE

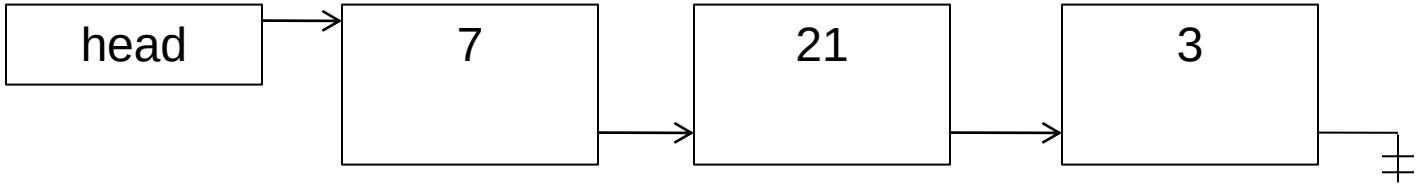
DELETE NODE

- **How can we remove data from a linked list?**
  - Search the linked list to find the node to delete
  - Change linked list pointers to “jump over” node
  - Return memory space to the operating system
  - Handle special cases (delete first node, last node, empty list)



DELETE NODE

- **What will this code do?**



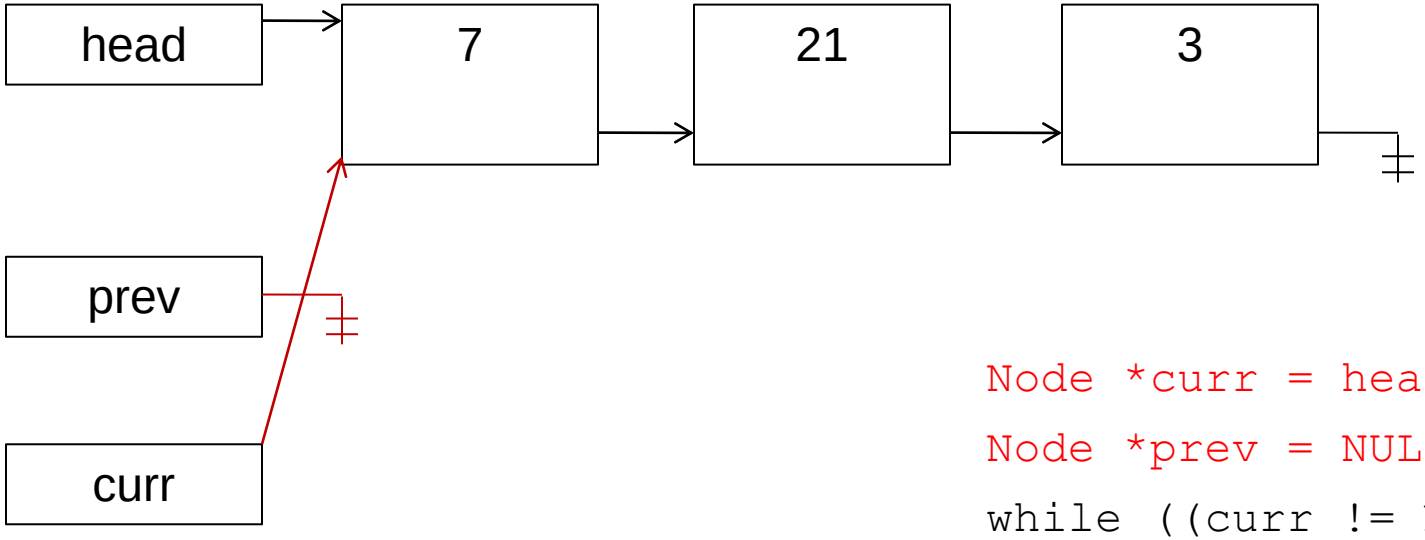
prev

curr

```
Node *curr = head;
Node *prev = NULL;
while ((curr != NULL)
&& (curr->num != 21))
{
    prev = curr;
    curr = curr->next;
}
```

DELETE NODE

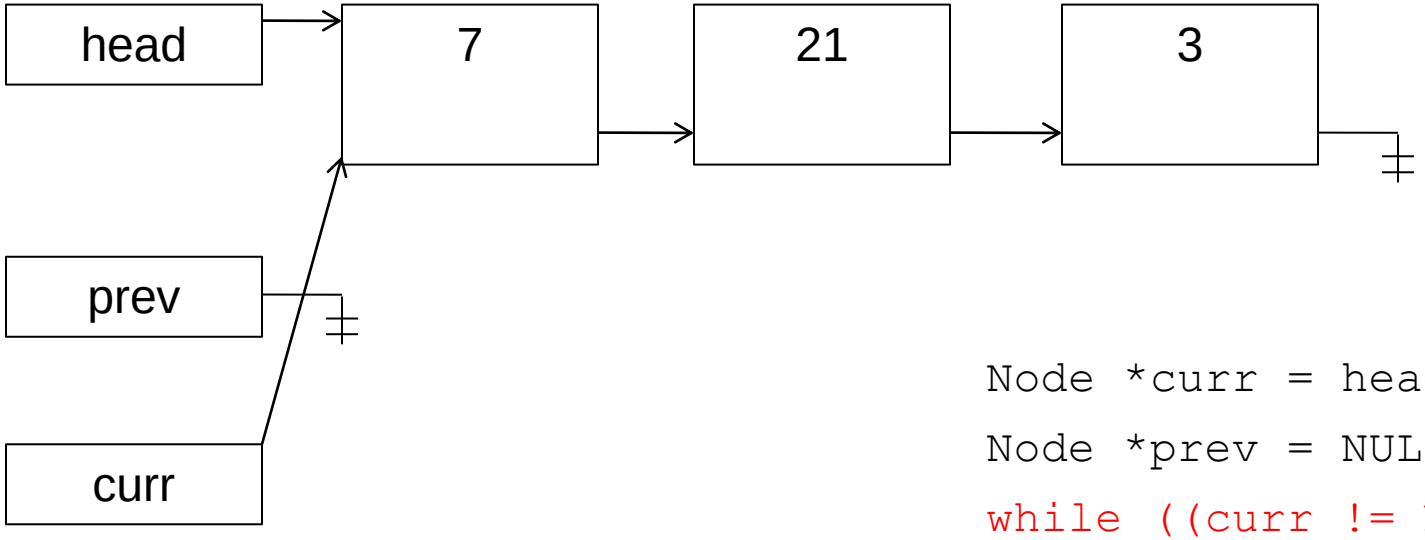
- **What will this code do?**




```
Node *curr = head;
Node *prev = NULL;
while ((curr != NULL)
&& (curr->num != 21))
{
    prev = curr;
    curr = curr->next;
}
```

DELETE NODE

- **What will this code do?**

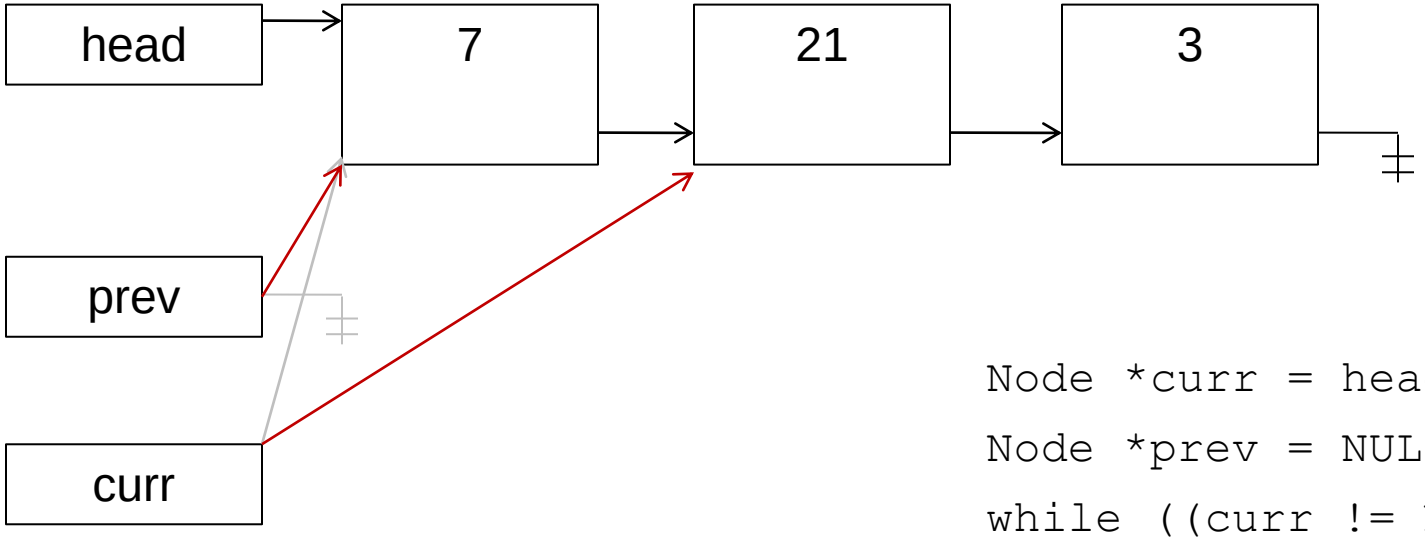


Condition is true

```
Node *curr = head;
Node *prev = NULL;
while ((curr != NULL)
&& (curr->num != 21))
{
    prev = curr;
    curr = curr->next;
}
```

DELETE NODE

- **What will this code do?**

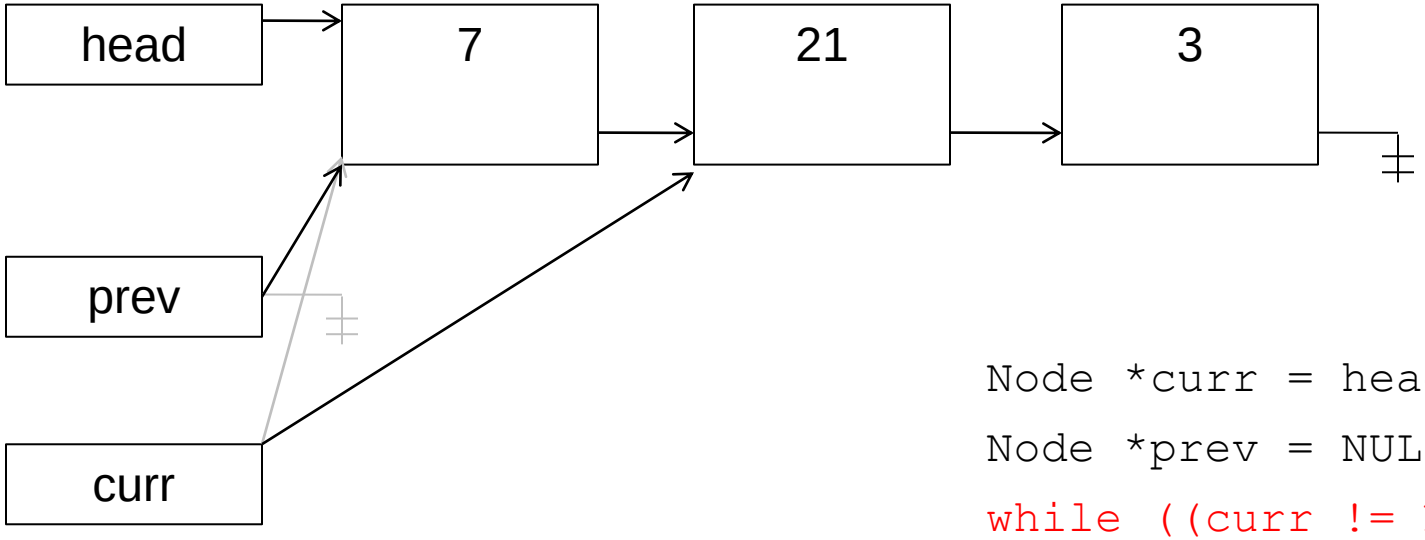



```
Node *curr = head;
Node *prev = NULL;
while ((curr != NULL)
&& (curr->num != 21))
{
    prev = curr;
    curr = curr->next;
}
```

Move both pointers to next nodes

DELETE NODE

- **What will this code do?**

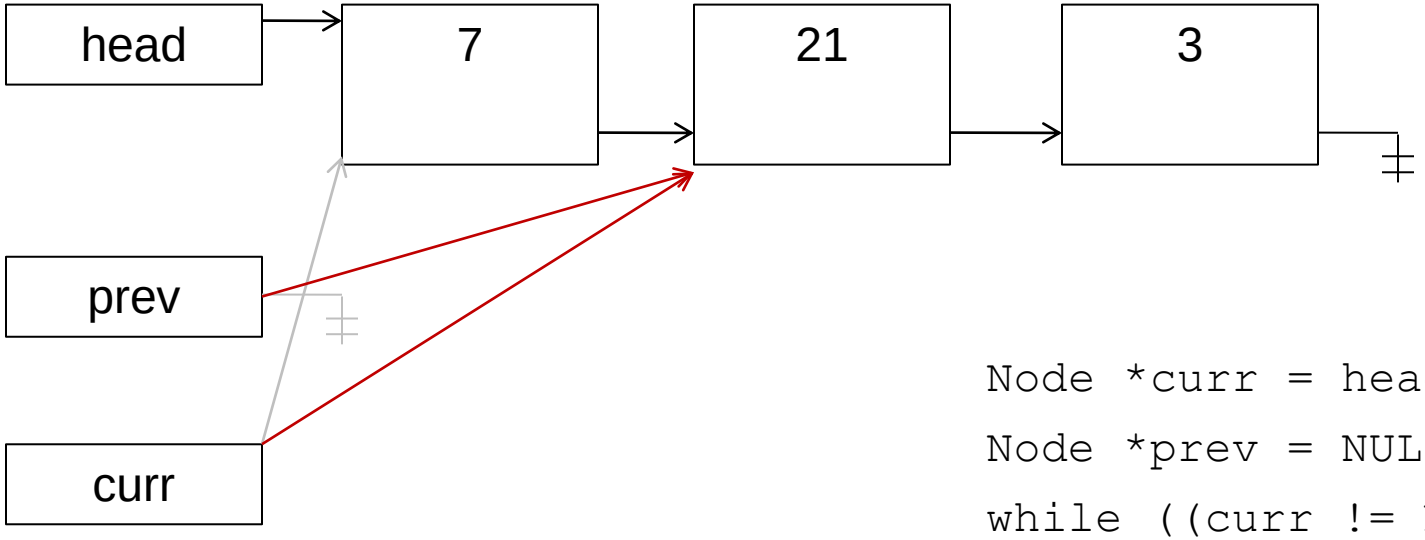



```
Node *curr = head;
Node *prev = NULL;
while ((curr != NULL)
&& (curr->num != 21))
{
    prev = curr;
    curr = curr->next;
}
```

Loop stops when we reach node with 21

DELETE NODE

- **What will this code do?**

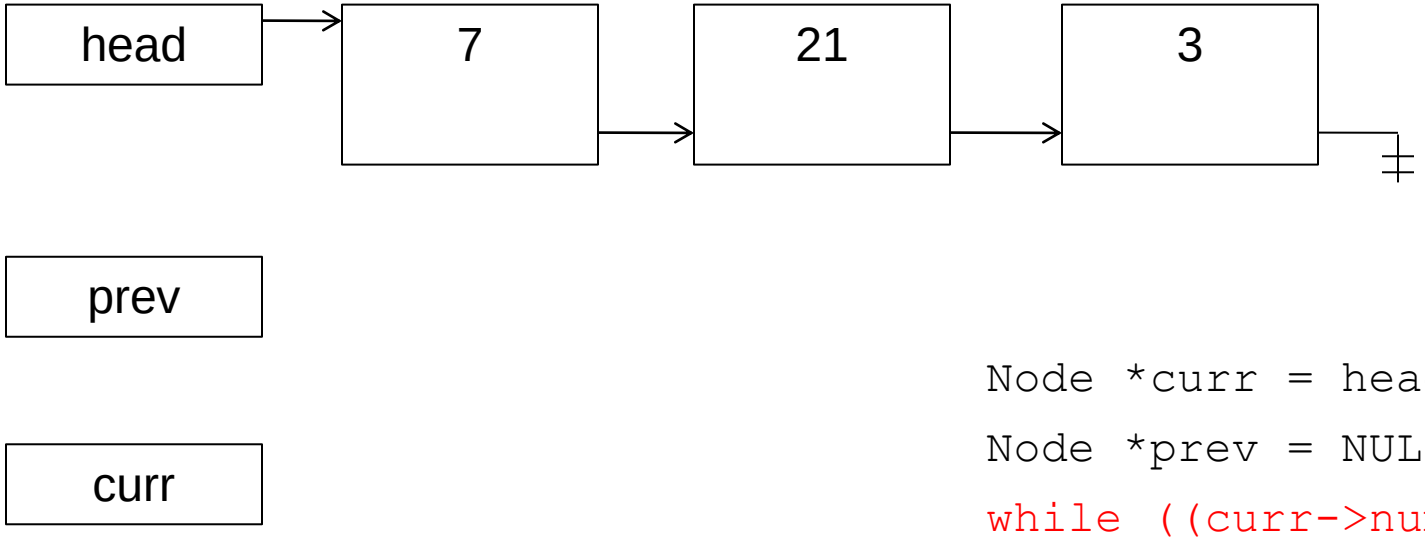



```
Node *curr = head;
Node *prev = NULL;
while ((curr != NULL)
&& (curr->num != 21))
{
    curr = curr->next;
    prev = curr;
}
```

Order of operations really matters here

DELETE NODE

- **What will this code do?**




```
Node *curr = head;
Node *prev = NULL;
while ((curr->num != 42)
&& (curr != NULL))
{
    prev = curr
    curr = curr->next;
}
```

DELETE NODE

- **Now we can change pointers to "jump over" deleted node**

```
// Adjust pointers and delete node
if (curr != NULL) && (curr == head)
     head = curr->next;
```

Special case where head points to deleted node

```
else if (curr != NULL)
   prev->next = curr->next;
```

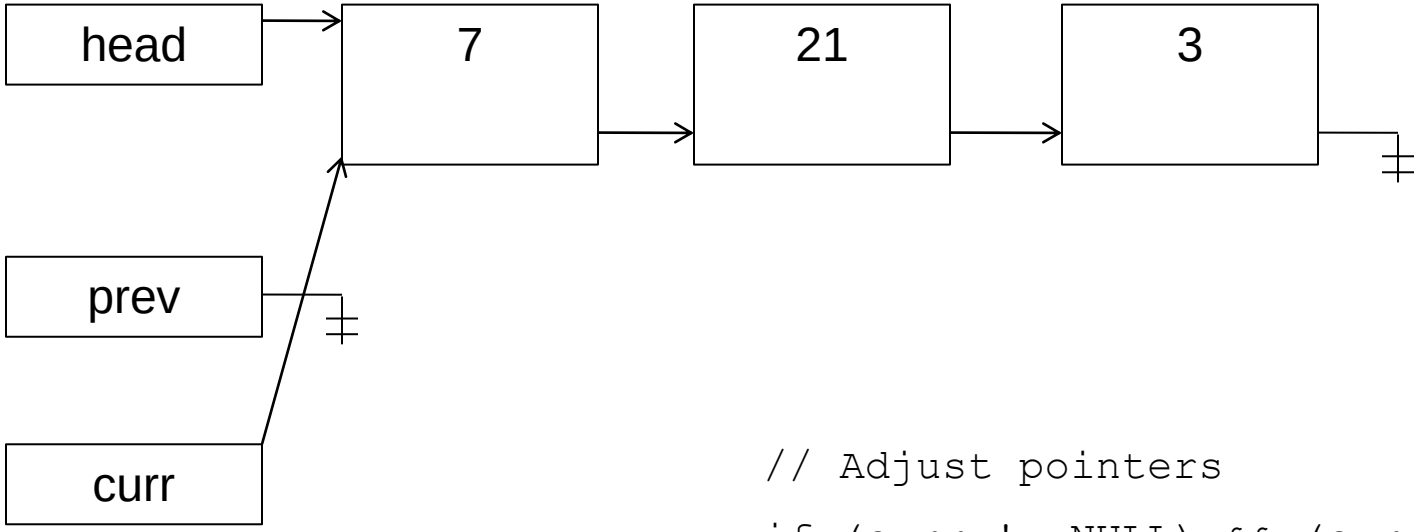
Normal case with node in middle or end of linked list

```
delete curr;
```

Return memory to O/S

DELETE NODE

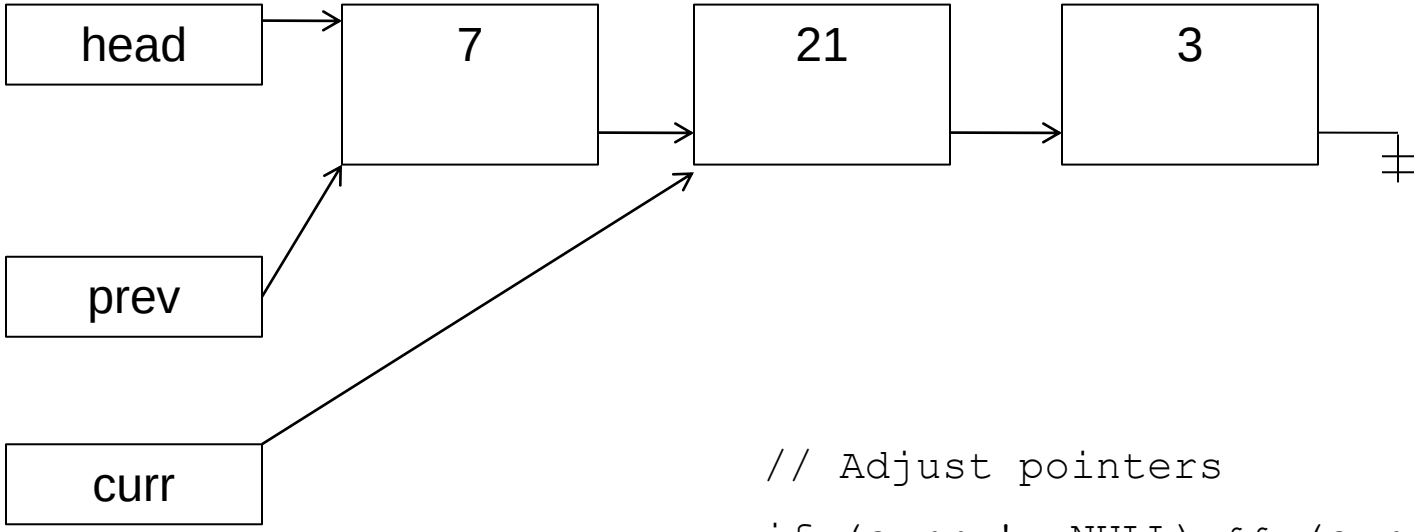
- **Verify code can delete node 7**




```
// Adjust pointers
if (curr != NULL) && (curr == head)
      head = curr->next;
else if (curr != NULL)
   prev->next = curr->next;
delete curr;
```

DELETE NODE

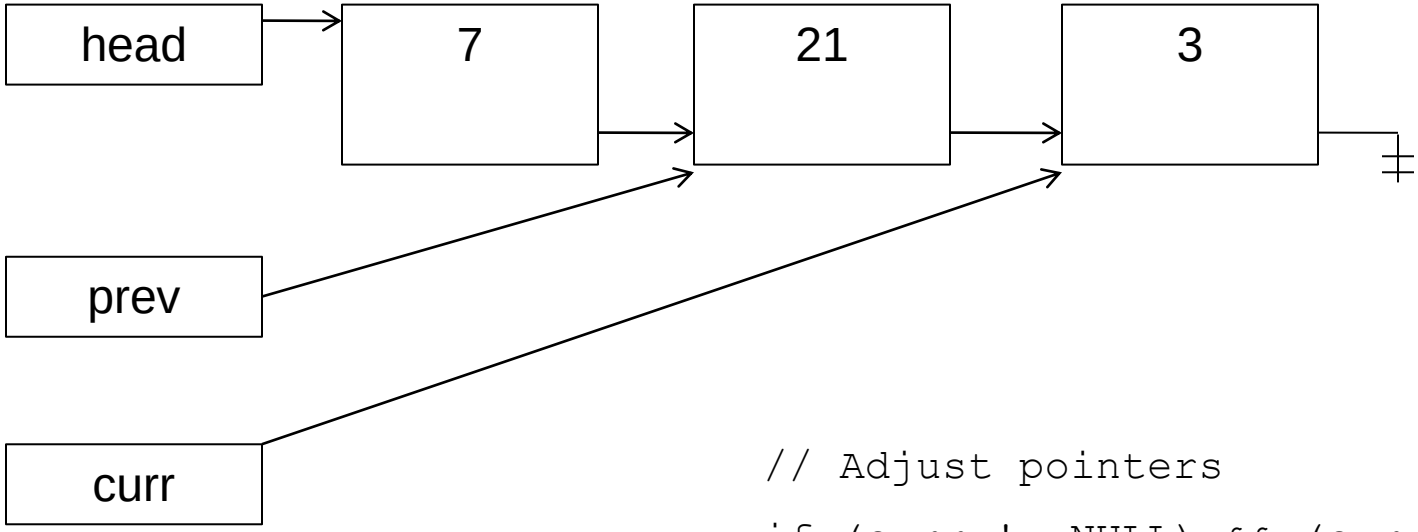
- **Verify code can delete node 21**




```
// Adjust pointers
if (curr != NULL) && (curr == head)
      head = curr->next;
else if (curr != NULL)
   prev->next = curr->next;
delete curr;
```

DELETE NODE

- **Verify code can delete node 3**




```
// Adjust pointers
if (curr != NULL) && (curr == head)
      head = curr->next;
else if (curr != NULL)
   prev->next = curr->next;
delete curr;
```

DELETE ALL NODES

- **In the destructor function we have to delete <u>all</u> nodes and give memory back to the operating system**

```
Node * ptr = head;
while (ptr != NULL)
{
    head = head->next;
    delete ptr;
    ptr = head;
}
```

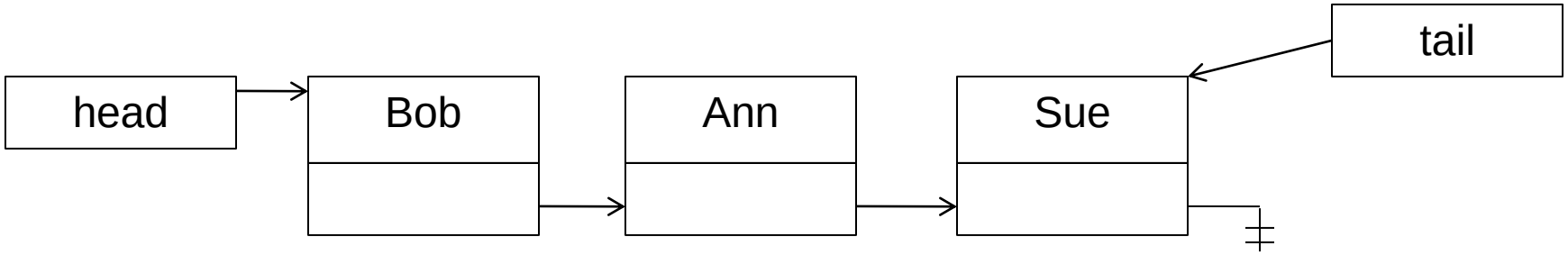
If we swap order we would be using a pointer after it is deleted (which could cause random bugs)

LINKED LISTS

HEAD AND TAIL POINTERS

HEAD AND TAIL POINTERS

- **We can insert nodes at the tail much faster if we maintain a pointer to the tail of the linked list.**




```
Node *ptr = new Node();
ptr->Value = "John";
ptr->Next = NULL;
Tail->Next = ptr;
Tail = ptr;
```

Clearly much faster than walking the whole linked list to find tail

HEAD AND TAIL POINTERS

- **We need to adapt insert tail code to handle the special case of inserting into empty linked list**

```
// Create new node
Node *ptr = new Node();
ptr->Value = value;
ptr->Next = NULL;

// Insert new node
if (Head == NULL)
   Head = ptr;
else
   Tail->Next = ptr;
Tail = ptr;
```

HEAD AND TAIL POINTERS

- **We also need to modify the delete code to make sure set tail pointer correctly when the tail node is deleted**

```
...
// Special case if we delete tail
if (curr == tail)
   tail = prev;

// Special case if we delete last node
if (head == NULL)
   tail = NULL;
```

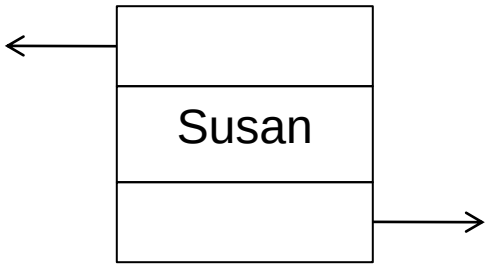
LINKED LISTS

DOUBLY LINKED LISTS

DOUBLY LINKED LISTS

- **In order to walk the linked list in either direction, we can add a second pointer called prev to the Node object**

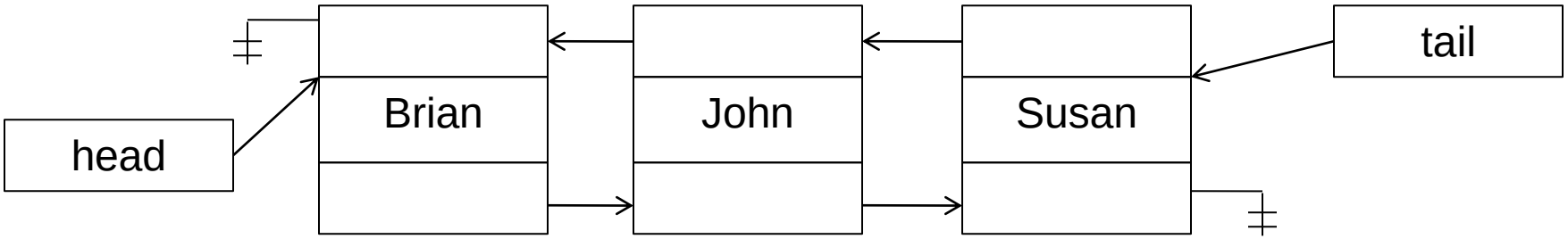
```
class Node
{
public:
    string name;
    Node *prev;
    Node *next;
};
```



These are not reserved words but almost everyone uses prev and next

DOUBLY LINKED LISTS

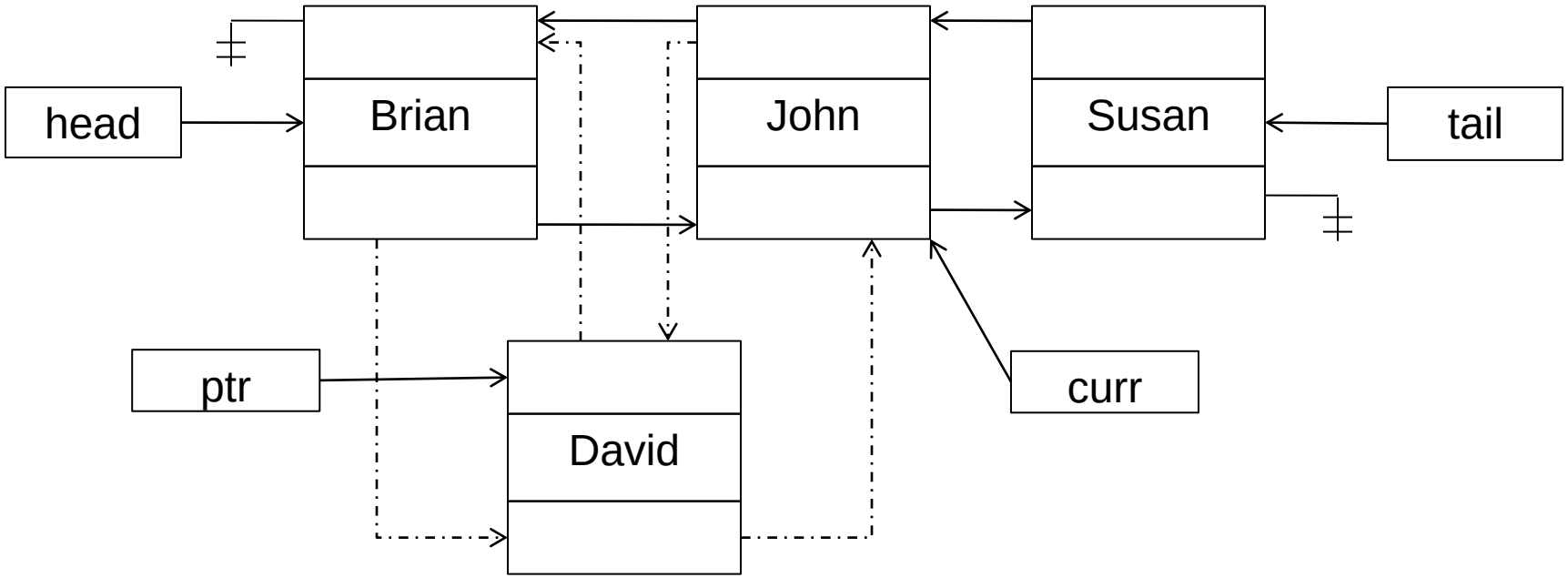
- **A doubly linked list has nice symmetry**
  - We can start at head and follow next from L to R
  - We can start at tail and follow prev from R to L



- **We must modify insert and delete to use both pointers**
  - In some cases, code is easier because we can use prev

DOUBLY LINKED LISTS

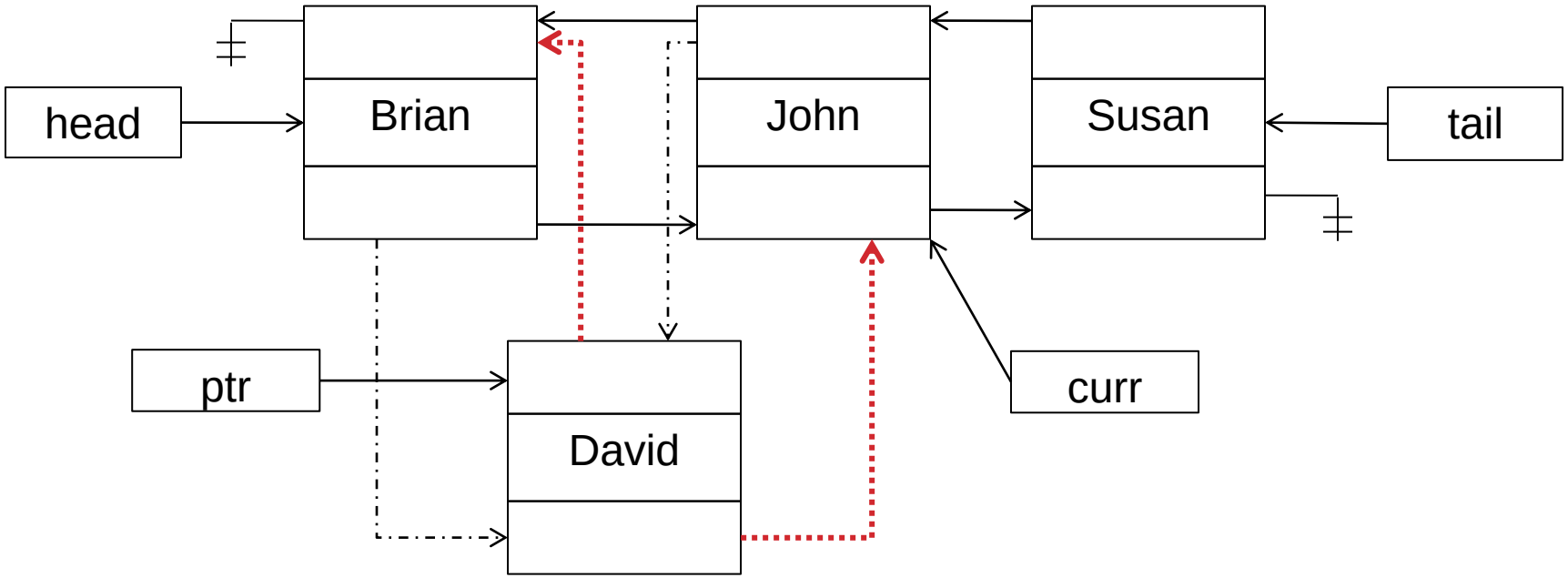
- **For sorted insertion, we must update four pointers**




```
Node *ptr = new Node();
ptr->name = "David";
```

DOUBLY LINKED LISTS

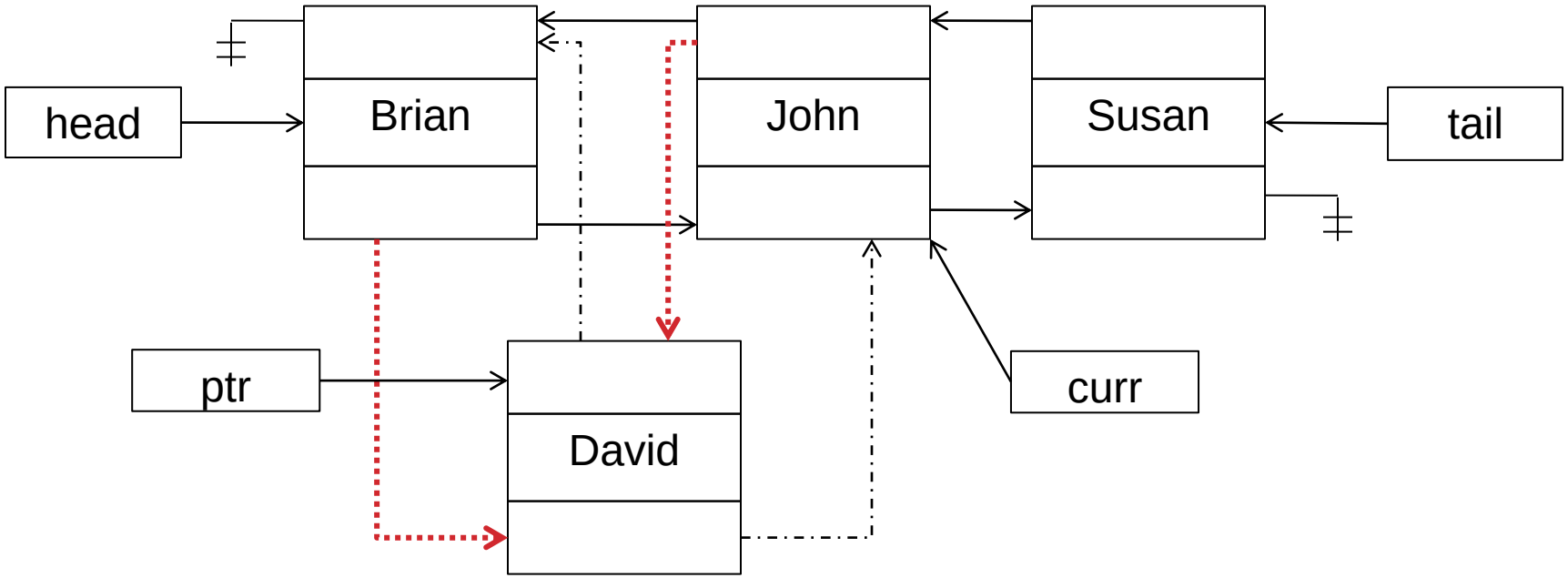
- **For sorted insertion, we must update four pointers**




```
ptr->next = curr;
ptr->prev = curr->prev;
```

DOUBLY LINKED LISTS

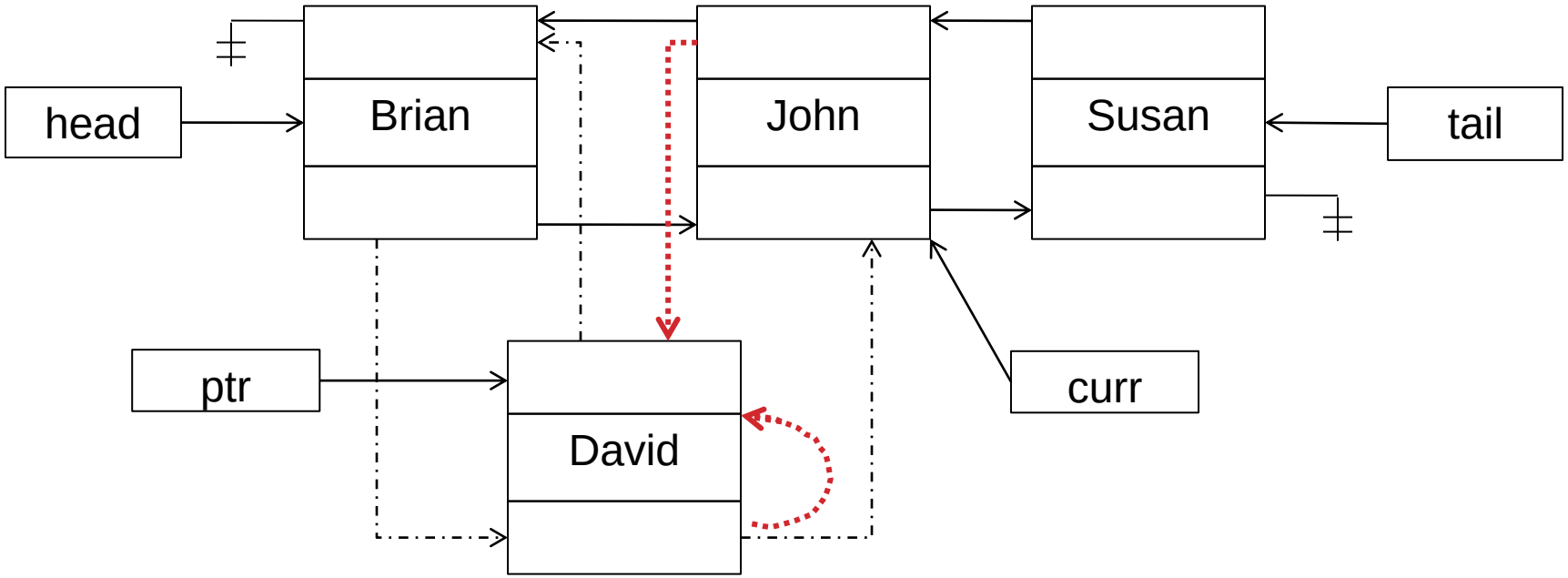
- **For sorted insertion, we must update four pointers**




```
curr->prev->next = ptr;
curr->prev = ptr;
```

DOUBLY LINKED LISTS

- **For sorted insertion, we must update four pointers**




```
curr->prev = ptr;          // Wrong order
curr->prev->next = ptr;    // Wrong order
```

DOUBLY LINKED LISTS

```
bool List::SortedInsert(int value)
{
   // Create new node
   LNode *ptr = new LNode();
   ptr->Value = value;
   ptr->Next = NULL;
   ptr->Prev = NULL;

   // Insert into empty list
   if (Head == NULL)
   {
      Head = ptr;
      Tail = ptr;
   }
```

DOUBLY LINKED LISTS

```
// Insert before head
else if (value <= Head->Value)
{
   ptr->Next = Head;
   Head->Prev = ptr;
   Head = ptr;
}

// Insert after tail
else if (value >= Tail->Value)
{
   ptr->Prev = Tail;
   Tail->Next = ptr;
   Tail = ptr;
}
```

DOUBLY LINKED LISTS

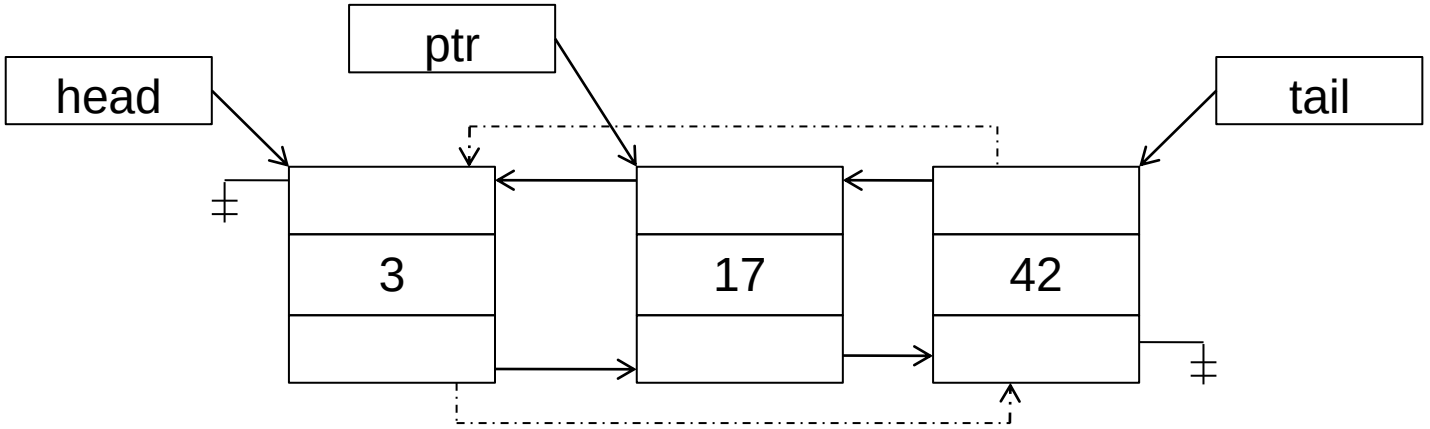
```
   // Insert in middle
   else
   {
      // Walk list to deletion point
      LNode *curr = Head;
      while ((curr != NULL) && (curr->Value < value))
         curr = curr->Next;

      // Connect node to list
      ptr->Next = curr;
      ptr->Prev = curr->Prev;
      curr->Prev->Next = ptr;
      curr->Prev = ptr;
   }
   return true;
}
```

Here is the code connecting nodes from the slide above

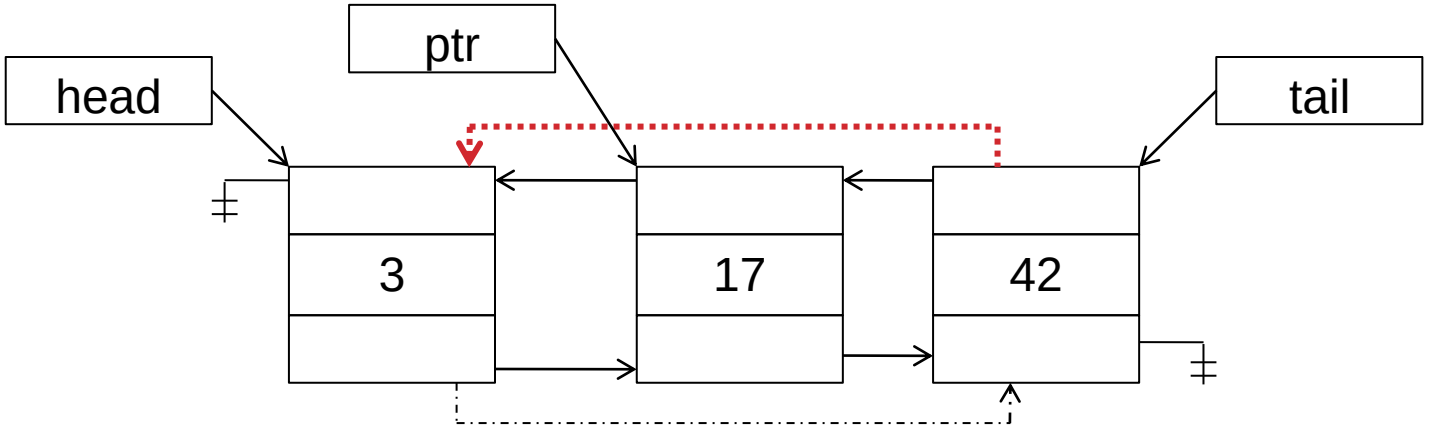
DOUBLY LINKED LISTS

- **For deletion from this list, we must update two pointers**
  - Pointer from previous node to the next node
  - Pointer from next node to the previous node



DOUBLY LINKED LISTS

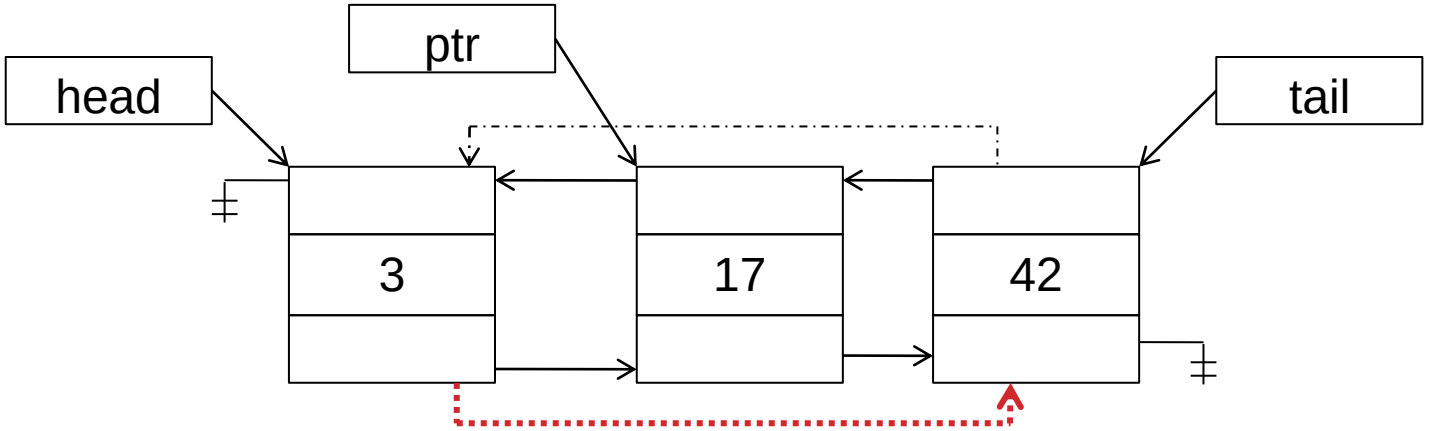
- **For deletion from this list, we must update two pointers**
  - Pointer from previous node to the next node
  - Pointer from next node to the previous node




```
ptr->next->prev = ptr->prev;
```

DOUBLY LINKED LISTS

- **For deletion from this list, we must update two pointers**
  - Pointer from previous node to the next node
  - Pointer from next node to the previous node




```
ptr->prev->next = ptr->next;
```

DOUBLY LINKED LISTS

```
bool List::Delete(int value)
{
   // Walk list to deletion point
   LNode *curr = Head;
   while ((curr != NULL) && (curr->Value != value))
      curr = curr->Next;

   // Check if value was found
   if (curr == NULL)
      return false;
```

DOUBLY LINKED LISTS

```
   // Connect previous node to next node
   if (curr == Head)
      Head = curr->Next;
   else
      curr->Prev->Next = curr->Next;

   // Connect next node to previous node
   if (curr == Tail)
      Tail = curr->Prev;
   else
      curr->Next->Prev = curr->Prev;

   // Delete node from list
   delete curr;
   return true;
};
```

LINKED LISTS

SUMMARY

SUMMARY

- **Linked lists can be used for applications where the amount of data varies at run time**
  - We can insert and delete data without moving other data
  - We can search for specific data items in the linked list
  - We can print all of the data in the linked list
  - We can create sorted or unsorted linked lists
  - We can use tail pointers to speed up insert tail
  - We can use doubly linked lists for symmetry

- **All linked list operations require that we "walk the list"**
  - We can not directly access specific locations in linked list
  - We can not implement binary search on linked list