CSC230

Intro to C++ Lecture 14

Outline

- **Doubly Linked List**
- **Mid-term Exam revisit**

Singly Linked List

- **How to search one element in a singly linked list?**
  - **with head**
  - **with both head and tail**

Doubly-linked List

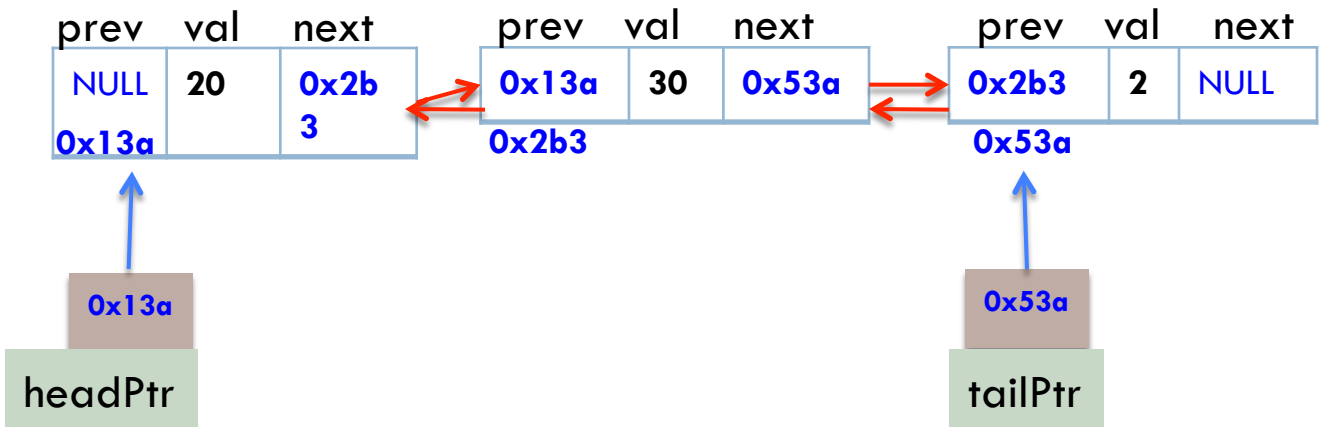
Each node has two pointers:

- One pointer (prev) points to the previous node in the list
- One pointer (next) points to the next node the in the list
- The first node's prev value is NULL
- The last node's next value is NULL

Q: Why doubly-linked list?

A: We can traverse/iterate the list backwards or forward.

```
#include<iostream>

using namespace std; struct DLNode {
int val; DLNode* prev; DLNode* next;
};

int main()
{
DLNode* headPtr, *tailPtr;
};
```

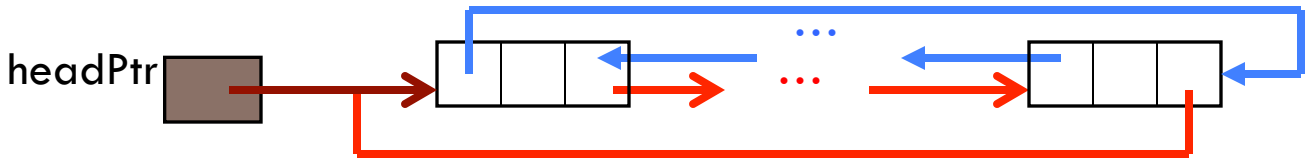


Circular Doubly Linked List

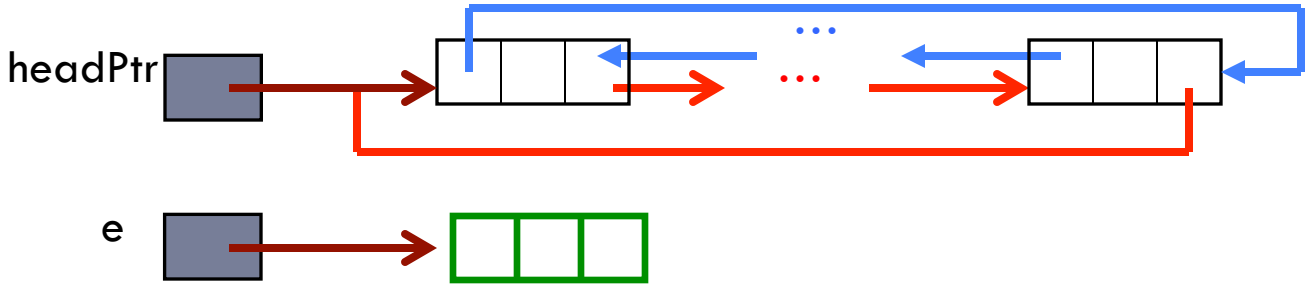
- In **singly linked** list, it takes O(n) time to access the last node.
- In **doubly linked** list, it takes O(1) time to access both the first node and the last node
  - But we maintain two pointers to the list. One is the head pointer, the other is the tail pointer.

In circular doubly linked list

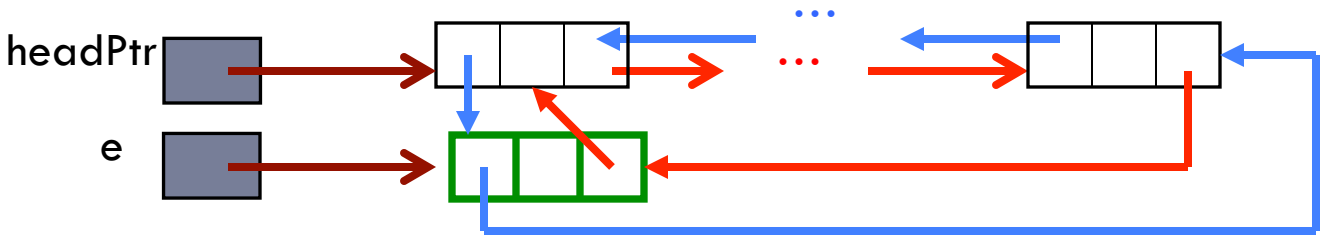
- It has head pointer, no tail pointer
- The access time of the first node and the last node is still O(1)
- The prev value of the first node points to the last node, the next value of the last node points to the first node



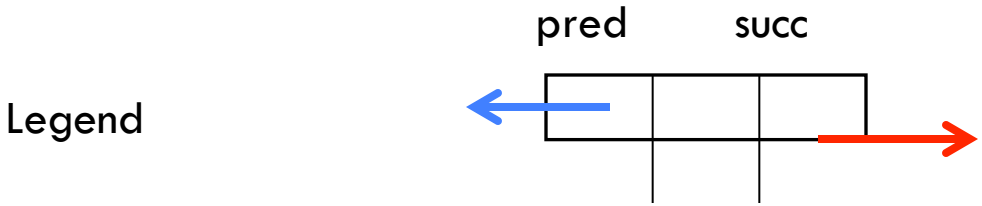
Suppose we want to append e to this list:



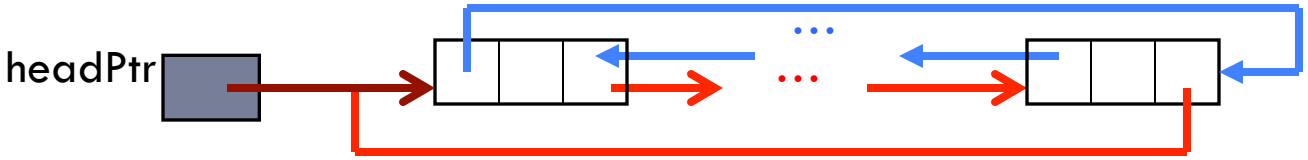
This is what it looks like after the append:



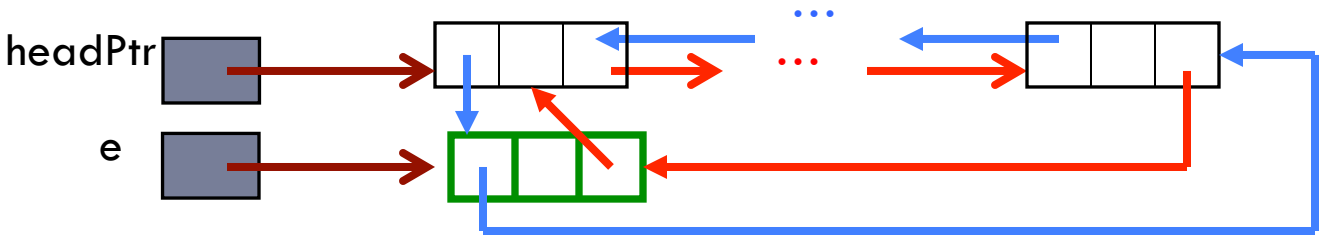
What if we prepended e instead of appending it?



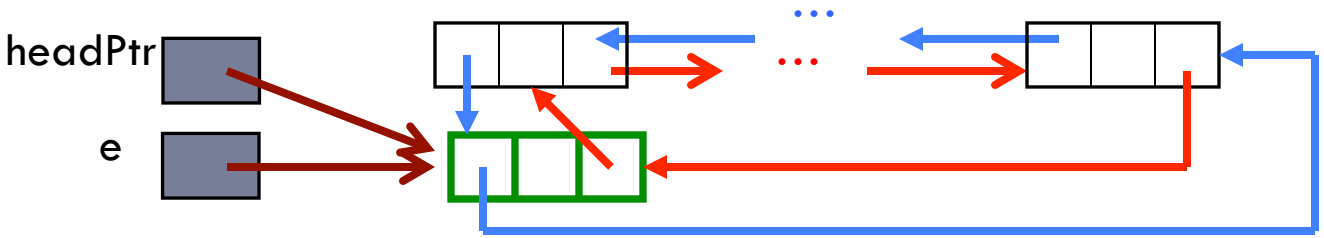
Suppose we want to append e to this list:



What append does:



What prepend does:

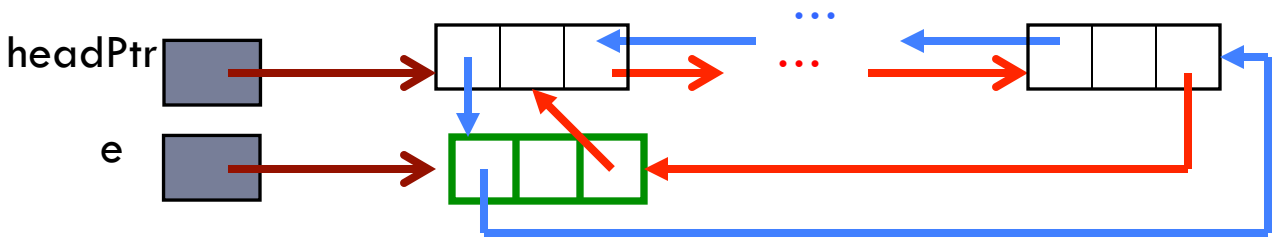


Therefore: prepend(e); can be done by

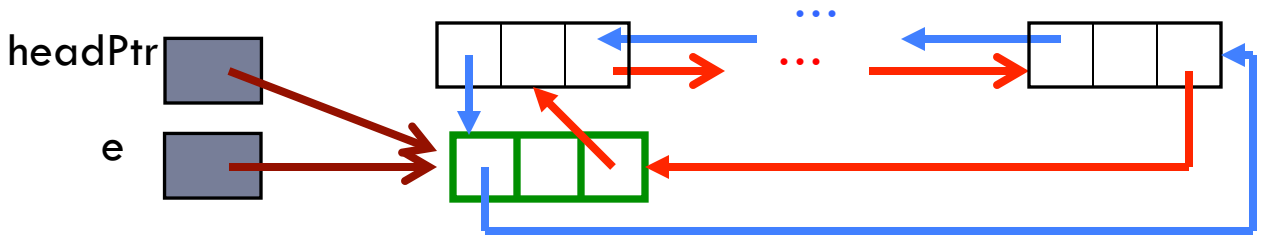
```
append(e); headPtr = headPtr->pred;
```

body of prepend

What append does



What prepend does



Prepend(e) is simply append(e); head= head->pred;

How much time did you spend writing and debug-ging prepend?

Did you try to write prepend in terms of append?

Morals of the story:

1. **Read carefully.**
2. **Visualize** what methods do; understand specs completely.
3. Avoid duplication of effort by using previously written methods

Access Example: Linear Search


```
#include<iostream>
using  namespace  std;
struct  Node{
  int  val;
  Node*  next;
};

Node* search(Node* head, int v){
  while(head!=NULL){
    if(head->val == v)
      return head;
    head = head->next;
  }
  return NULL;
}

int main(){
  Node* headPtr = new Node;
  headPtr->val = 20;
  headPtr->next = NULL;
  cout << search(headPtr, 20)->val << endl;
}
```

Recursion on Lists

- Recursion can be done on lists
  - Similar to recursion on integers
- Almost always
  - Base case: empty list
  - Recursive case: Assume you can solve problem on the tail, use that in the solution for the whole list
- Many list operations can be implemented very simply by using this idea
  - Although some are easier to implement using iteration

Recursive Search

- Base case: empty list
  - return false

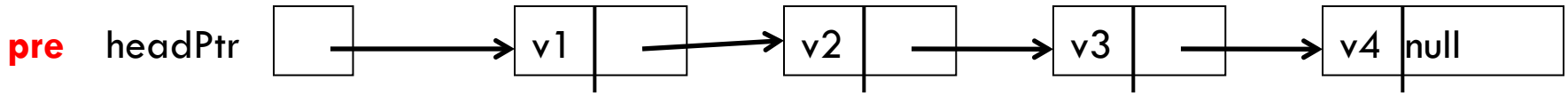
- Recursive case: non-empty list
  - if data in first cell equals object x, return true
  - else return the result of doing linear search on the tail

Recursive Search

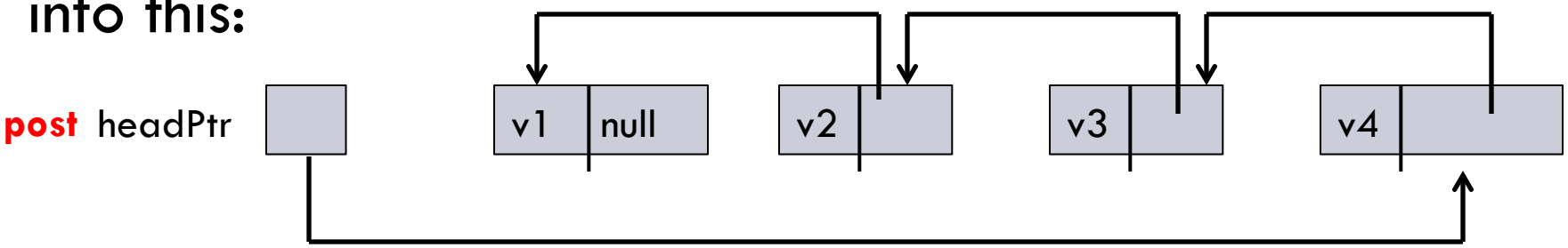

```
#include<iostream>
using namespace std;
struct Node{
  int val;
  Node* next;
};

Node* search(Node* head, int v){
  if(head==NULL) return NULL;
  if(head->val == v) return head;
  return search(head->next, v);
}

int main(){
  Node* headPtr = new Node;
  headPtr->val = 20;
  headPtr->next = NULL;
  cout << search(headPtr, 20)->val << endl;
}
```

Iterative linked list reversal

Change this:

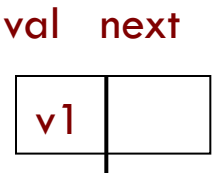


into this:



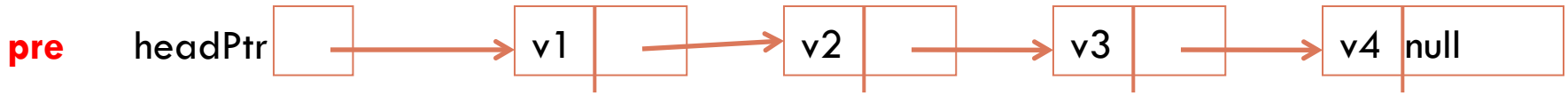
Reverse the list by changing **headPtr** and all the next fields

Legend:

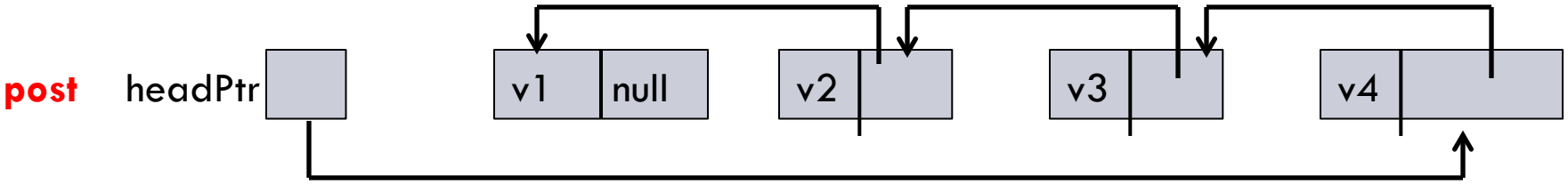


Iterative linked list reversal

Change this:



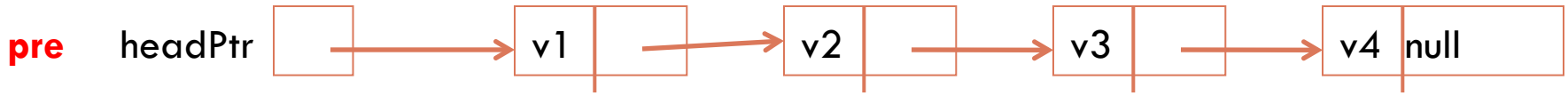
into this:



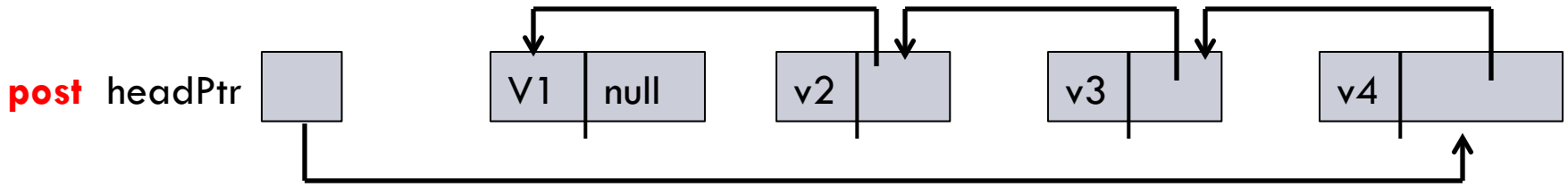
Use a loop, changing one **next** field at a time. Getting it right is best done by drawing a general picture that shows the state of affairs **before/after** each iteration of the loop. Do this by drawing a picture that combines the precondition and postcondition.

Iterative linked list reversal

Change this:



into this:

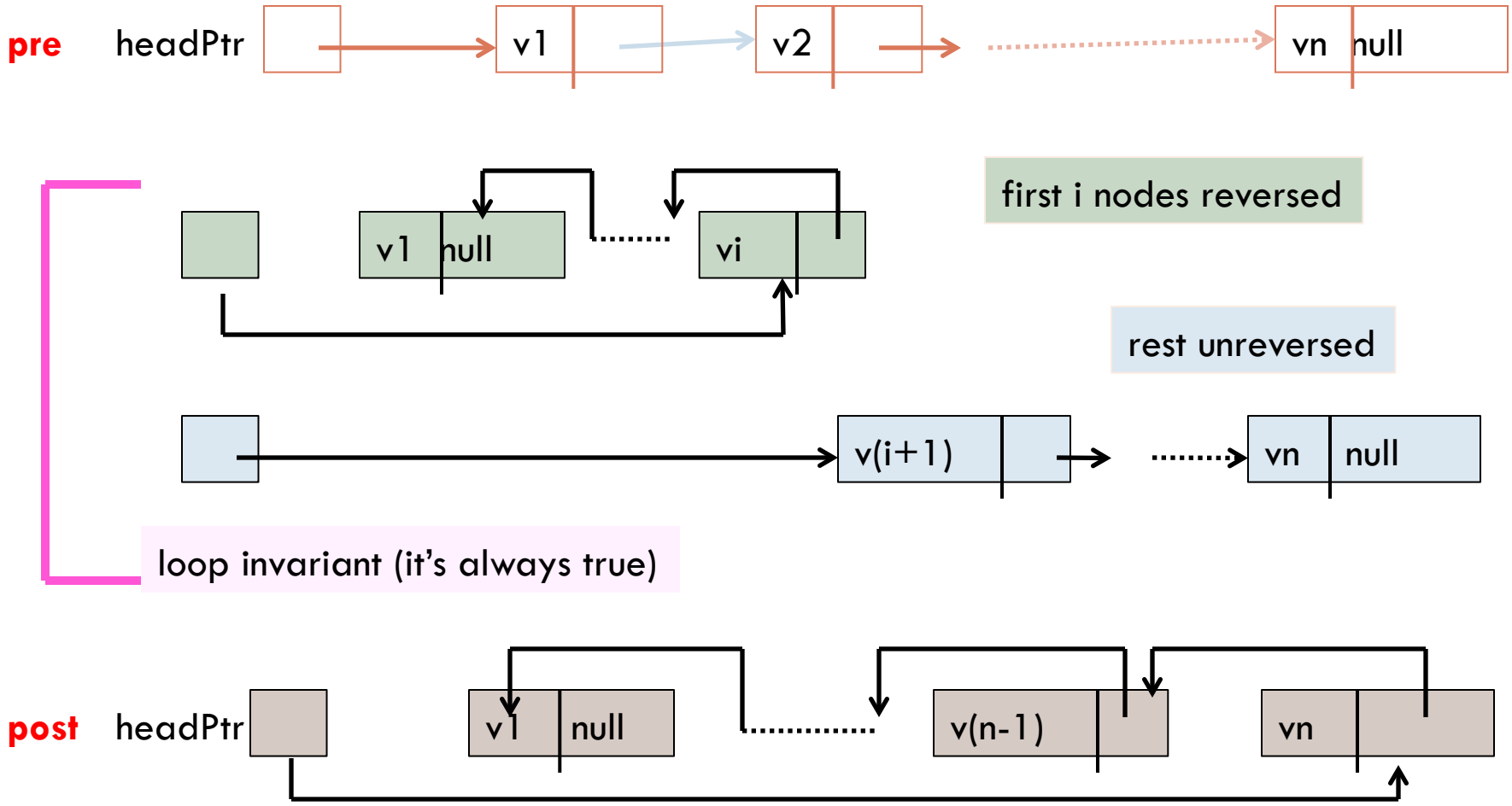


The loop will fix the next fields of nodes beginning with the first one, then the second, etc.

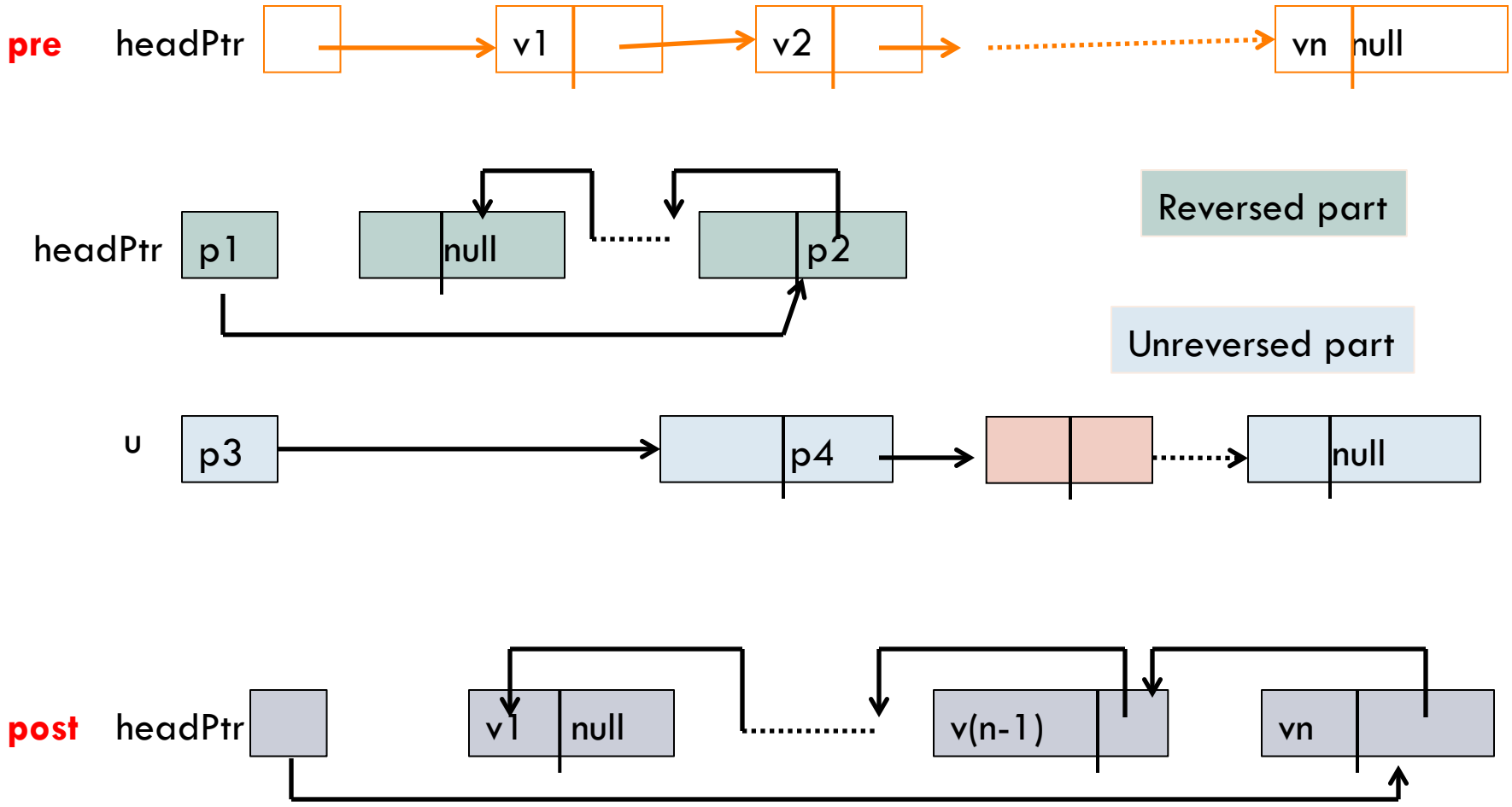
The first part of the list will be reversed —look like pre

The second part will not be reversed —look like post

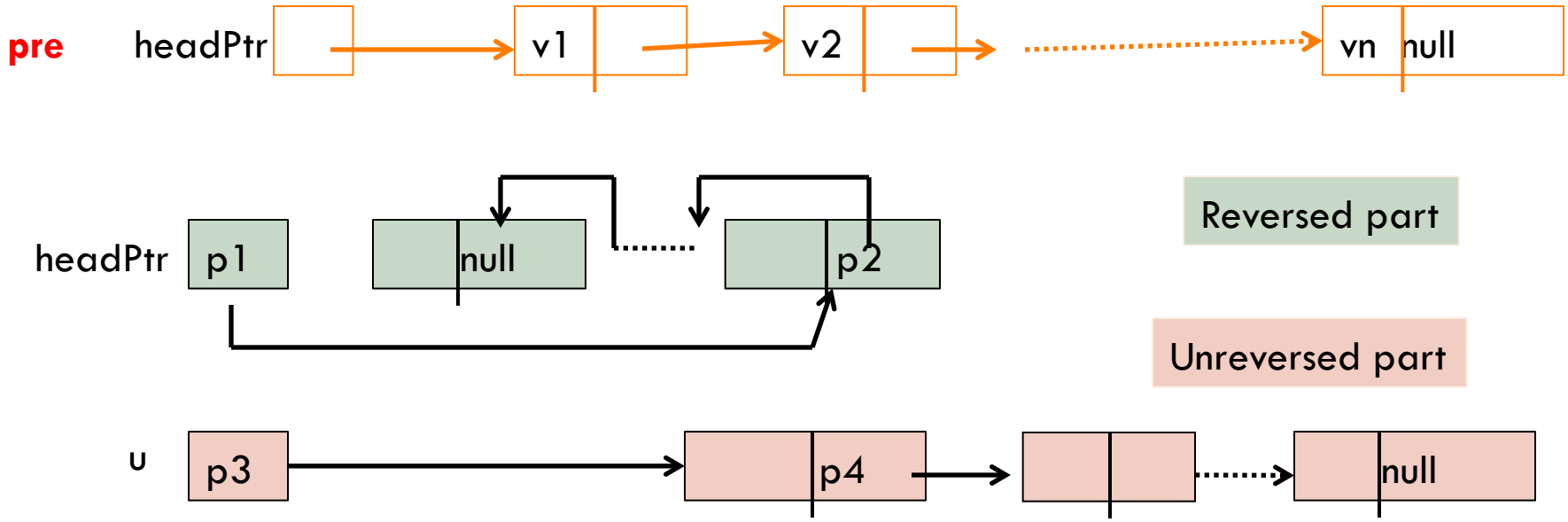
Iterative linked list reversal



Iterative linked list reversal



Make the invariant true initially



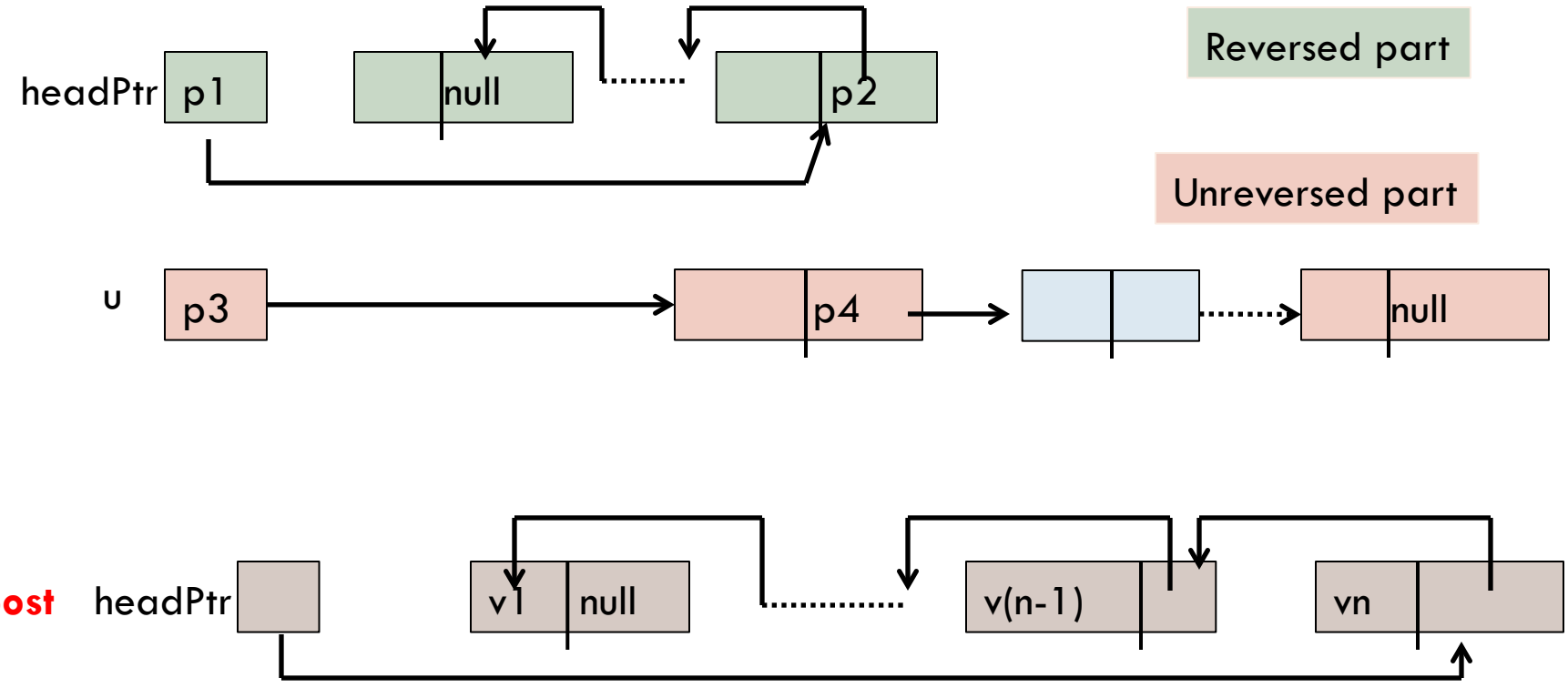
Initially, unreversed part is whole thing: u= headPtr;

Reversed part is empty: headPtr = null;

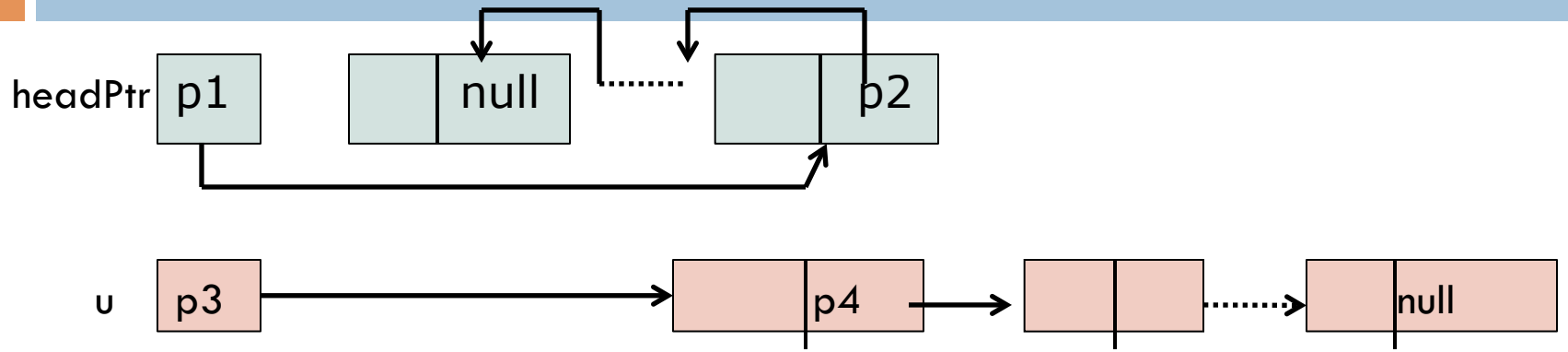
When to stop loop?

u= headPtr; headPtr= null;
while (u != null)

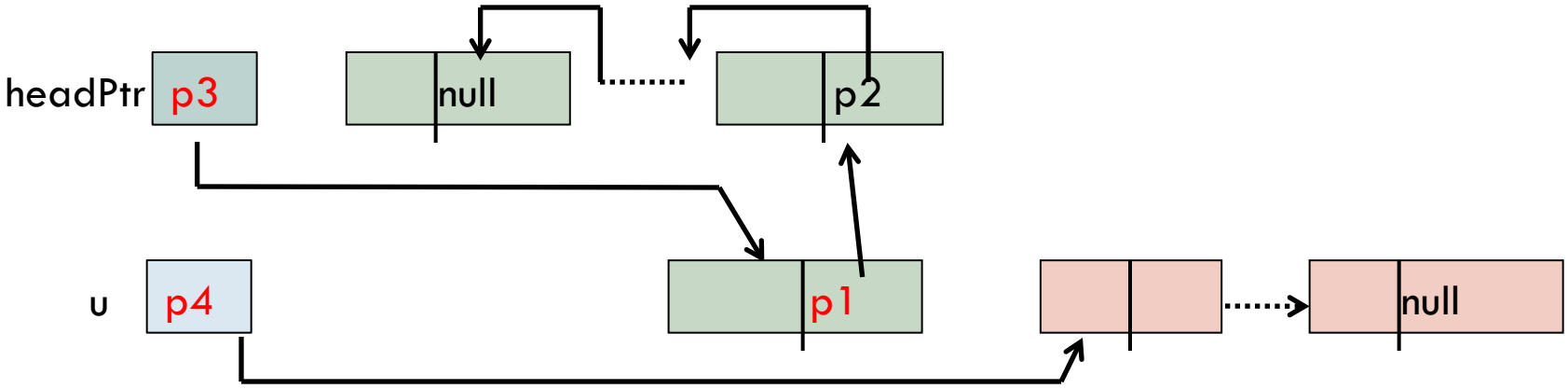
Upon termination, unreversed part is empty: u == null. Continue as long as u != null



Loop body: move one node from u list to head list. Draw the situation after the change




```
u= headPtr; headPtr= null;
while (u != null) {    Node* t= headPtr; headPtr= u; u= u->next; headPtr->next= t; }
```



Recursive Reverse

- Practice: Write a recursive function for Linked List Reversal!