Week 3

Announcements

- Project 2
  - Due next Tuesday, 4/20 🌲 @ 11PM
- Midterm
  - Next Thursday, **1/22 @ 6:30-8:30pm**
  - details will be emailed

DOUBLY LINKED LISTS BE LIKE

Questions?

- anything?
- Carey's Linked List notes: https://drive.google.com/drive/folders/1xq6EC2bjQiegT3GYs5f1jIhQAUty8gfH

- UPE CS32 MT 1 Review
  Week 4 Tuesday: 7-9PM PT
  https://ucla.zoom.us/j/94599416715?pwd=MGkzWEJIRFZMMWsyd1IxRlpleVM2QT09

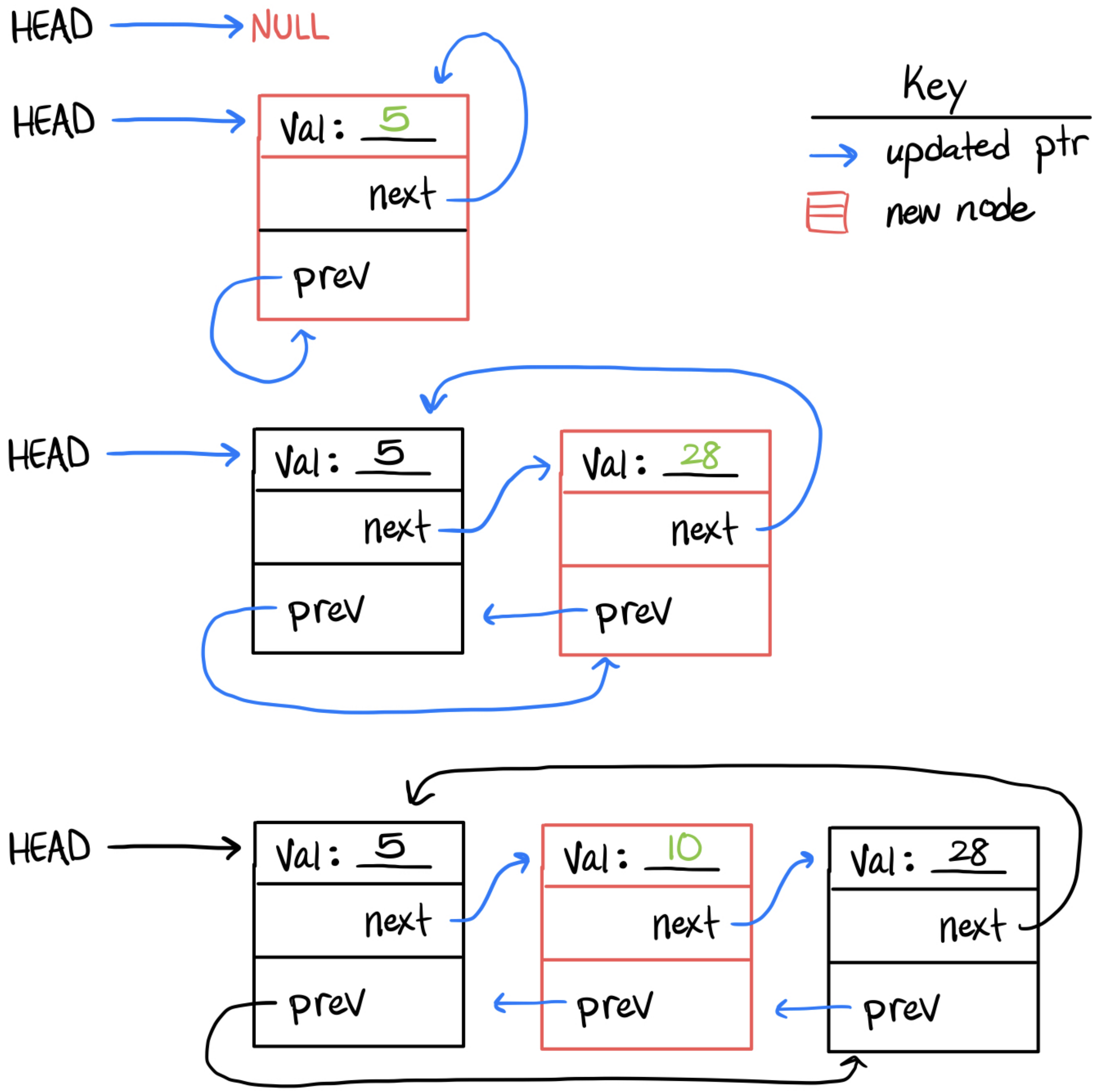
Arrays vs Linked Lists

- arrays
  - pros
    - memory is consecutive, so access is fast to any single element
  - cons
    - arrays are fixed size once they're allocated
      - inserting to the start means more shifting
        - costly to keep adding/deleting things if you want to maintain some order
- linked lists
  - pros
    - efficient insert/delete (don't have to do any shifting)
  - cons
    - have to iterate through entire list to find one element

Nodes

```
// by request we're doing a doubly linked, circular linked list

struct Node {
  int val;
  Node* next;
  Node* prev;
};

// todo: create a linked list with 3 nodes
// add values 5, then 28, then 10 in the middle
```

```
// first make a head pointer
Node* head = nullptr;

// first node, adding to an empty list **************************************
head = new Node();
head->val = 5;
// b/c list is circular, our one node points back to itself for next and prev
head->next = head;
head->prev = head;

// now adding a second node to our list with one node ***********************
Node* node2 = new Node();
node1->val = 28;

// again, b/c circular, next and prev of our second node point to the first node
node2 -> next = head;
node2 -> prev = head;

// update our first node's next pointer
// and the prev pointer of the node after node2, which is again head in this case
head->next = node2;
head->prev = node2;

// add a node to the middle *************************************************
Node* node3 = new Node();
node3->val = 10;

// node2 is now after us so we update the next pointer
// head is now before us so that is our prev
node3 -> next = node2;
node3 -> prev = head;

// head's next has changed
head->next = node3;

// and node2's previous has changed
node2->prev = node3;
```

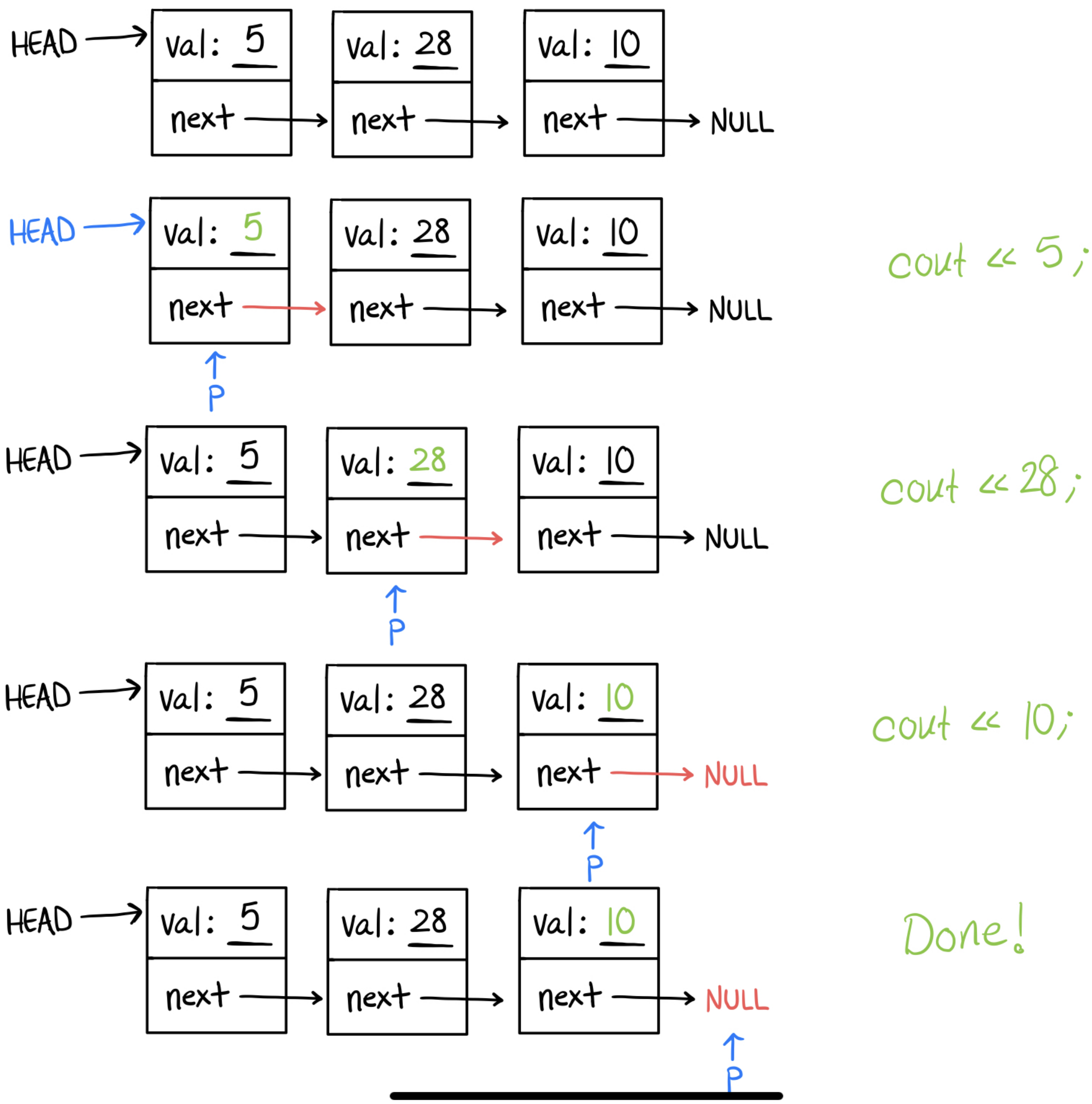


Linked Lists

```
// todo: create a class to handle our list
class LinkedList {
  public:
    void printList();
    void insert(int pos, int val);
    void remove(int pos);
  private:
    // todo: what do we want to keep track of?
    struct Node {
```

```
      int val;
      Node* next;
      Node* prev;
    };

    // for now, just a head pointer to have access to the start of our list
    // and size might be useful to use later on (make sure to increment/decrement)
    Node* head = nullptr;
    int size = 0;

    // b/c its circular we don't really need a tail (just use head->prev)
}

// todo: implement printList()
void LinkedList::printList() {

  // iterate starting from the head and just use our variable p to access values
  for( Node* p = head; p != head->prev; p = p->next ) {
    cout << p->val << endl;
  }
}
```

This drawing was created for a singly-linked LL (before we switched to doubly-linked), but still illustrates our logic correctly.



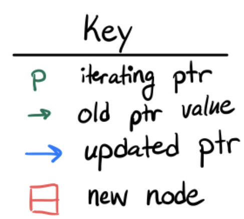
- 4 cases to check when adding and deleting nodes at certain positions
  - empty list
  - Beginning
  - Middle
  - End
- circular, doubly-linked LLs or LLs that use a dummy node help simplify our code by allowing us to combine certain edge cases

Add Nodes

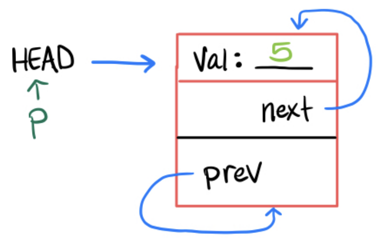
```
// todo: implement insert()
// give a position (assume in bounds), insert a new node and "shift" the old
// nodes to the right
void LinkedList::insert(int pos, int v) {

  // if pos is out of bounds, return without doing anything here
  if (pos < 0 || pos > size)
    return;

  // allocate new node, now we just have to make the proper pointer updates
  Node* n = new Node();
  n->val = v;

  // adding to an empty list
  if (head == nullptr) {
    head = n;
    head->next = head;
    head->prev = head;
  }
  else {

    // add to the beginning *********************************************
    if (pos == 0) {
      n->next = head;
      n->prev = head;

      head->prev = n
      head->next = n;

      // we've been using the old head up to now, but we have to update it
      // to point to our new node
      head = n;
    }

    // add our node somewhere in the middle or the end ******************

    // first iterate to the current node in our target position
    Node* p = head;

    // if we aren't adding to the end, increment p to where we want to insert
    if (pos != size) {
      for(int i = 0; i < pos; i++)
        p = p -> next;
    }

    n->next = p;
    n->prev = p->prev;

    (p->prev)->next = n;
    p->prev = n;
```

```
  }

  size++;
}
```

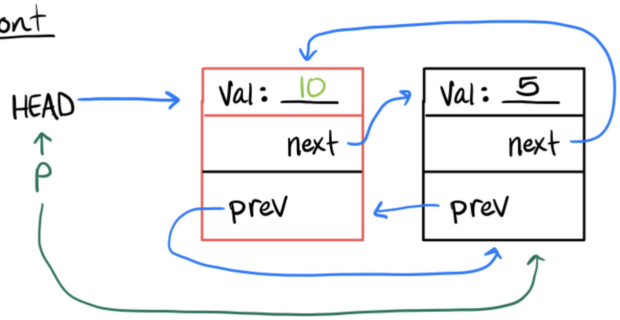
Init



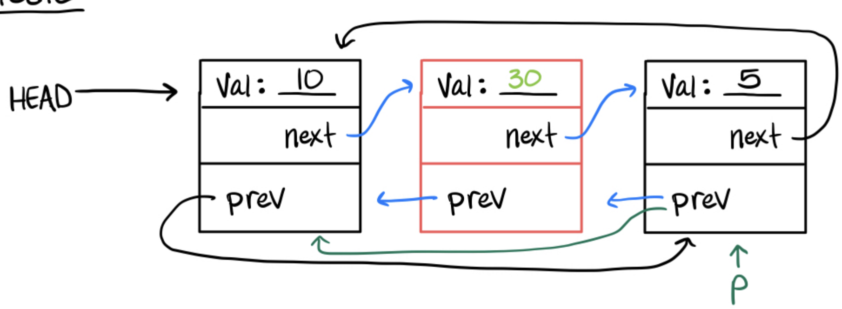
Empty List



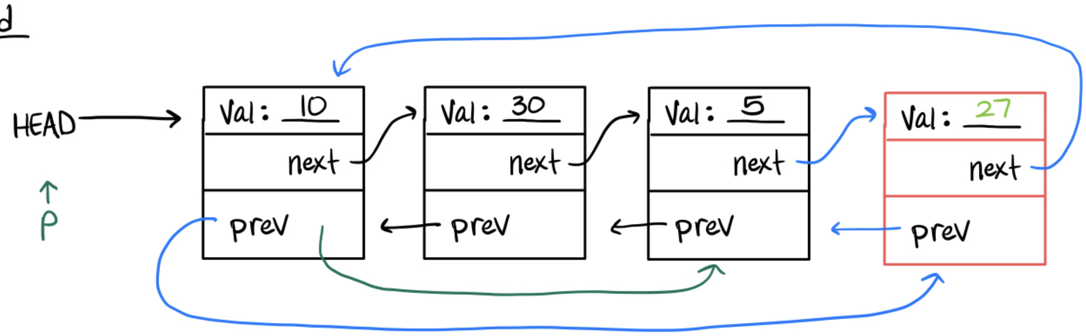
Front



Middle



End



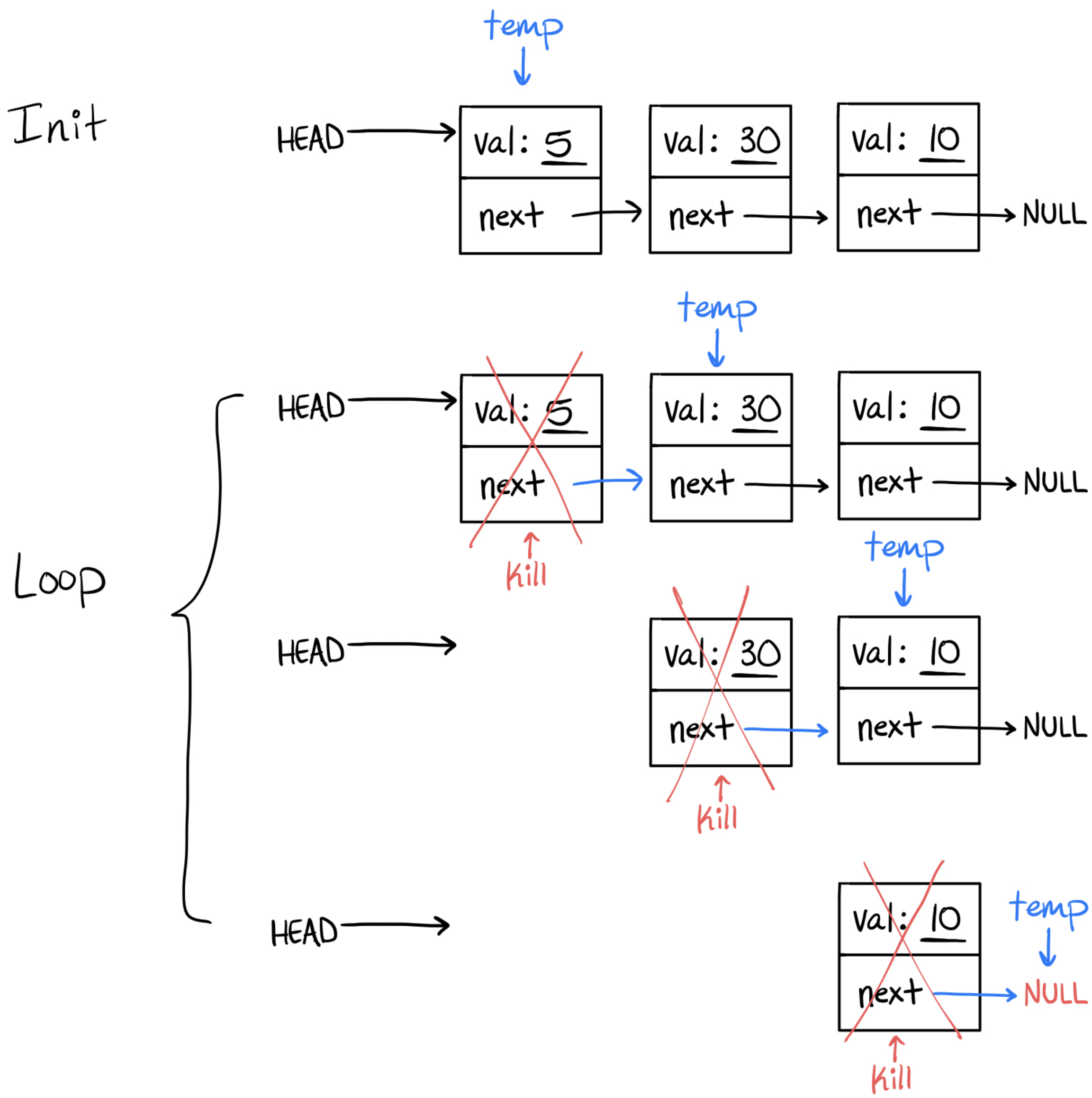
Delete Nodes

```
// didn't have time for this, but check the same cases as adding nodes

// todo: finish remove()
void LinkedList::remove(int pos) {

}
```

Destructor?

```
// needed? what about copy constructor and assignment operator?

~LinkedList() {
  Node* end = head->prev;
  for(Node* temp = head; temp != end; ) {

    Node* kill = temp; // make a pointer to access the node we want to destroy
    temp = temp->next; // increment temp so we can keep using it to iterate
    delete kill;
  }
}
```

Again, the drawing below is for a singly linked, non-circular list but it still illustrates how a destructor should work.



Stack (extra practice)

```
// we didn't get to this part and moved on to the worksheet
// what is a stack?

// can you implement one with a LL?
// no time left for this, let me know if you come up with any ideas :)

class LinkedListStack {
  public:
    void insert();
    Node pop();
```

```
  private:

}
```