



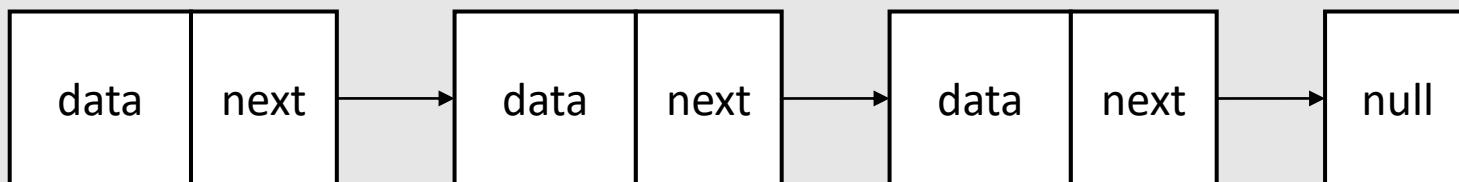
# Linked Lists

Prof. Darrell Long

CSE 13S

# Linked Lists

- The diagram below depicts a *singly linked list*.
- A *linked* data structure.
  - In a *singly* linked list, each node contains a data field and a pointer to the *next* node in the list.
  - In a *doubly* linked list, each node contains a data field and pointers to the *next and previous* nodes in the list.
- The last node in the list points to a terminator, usually a *null* pointer.



# Linked Structures

- Linked lists are members of the class of *linked structures*.
  - Linked lists
  - Trees
  - Tries
  - Graphs
  - Sparse matrices
  - ... and more.

# Advantages

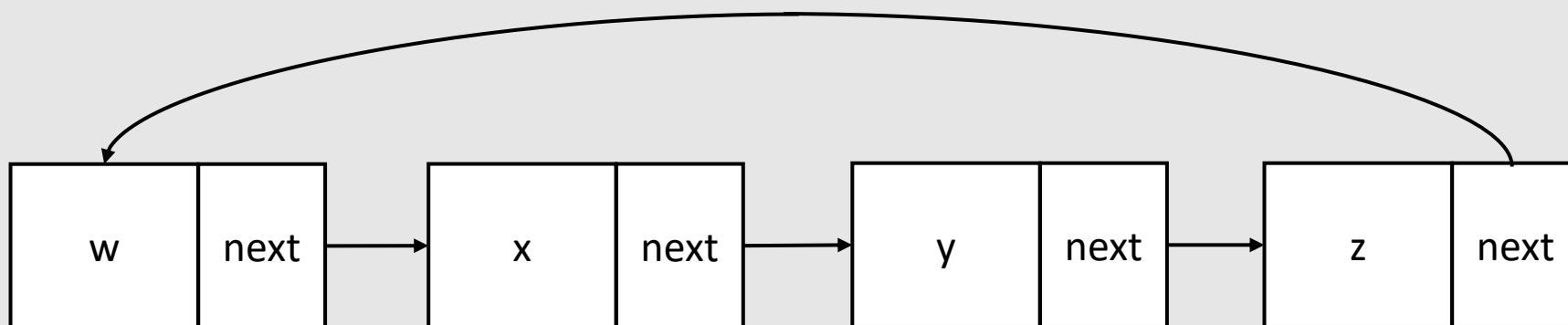
- No fixed memory allocation:
  - Grow and shrink at run-time without pre-allocating memory.
  - No need to know the initial size of the list.
- Insertion and Deletions:
  - No need to shift elements after insertion or deletion.
  - Only update the address to the next pointer of a node.
- Usage:
  - Easily implement linear data structures like stacks and queues.

# Disadvantages

- Memory usage:
  - Storing pointer to next node requires extra memory.
  - Arrays are friendlier to processor caches.
  - Slightly less memory efficient than arrays.
- Traversal:
  - Cannot randomly access elements, must traverse all elements up to the element we want to access.
  - Reverse traversing is difficult in singly linked lists.
    - Easy in doubly linked list but uses extra memory to store an additional pointer.

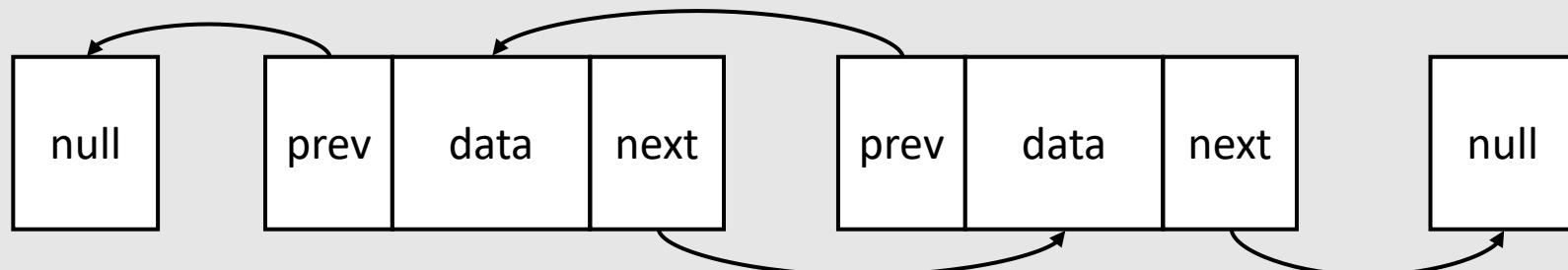
# Circular Singly Linked List

- The last node of the linked list points back to the tail.



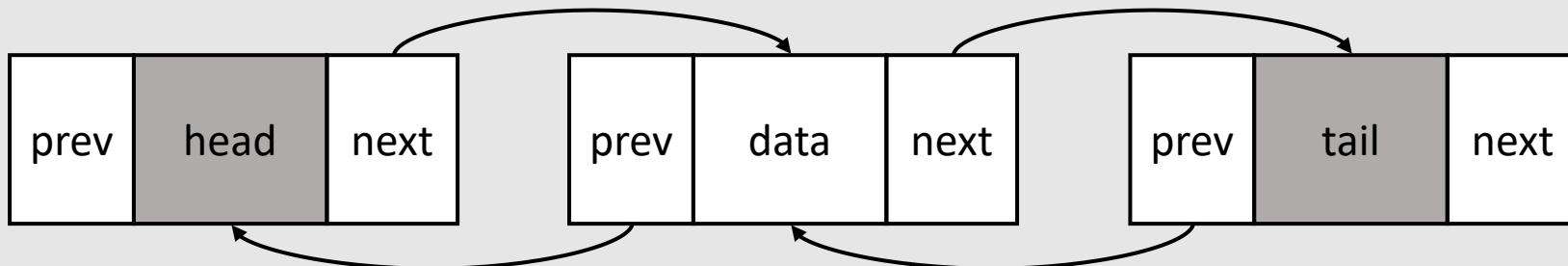
# Doubly Linked Lists

- Each node has a pointer to both the previous and next nodes.
- Allows traversal in two directions.
- Less memory efficient than a normal linked list.
- Typically implemented with *sentinel nodes*.



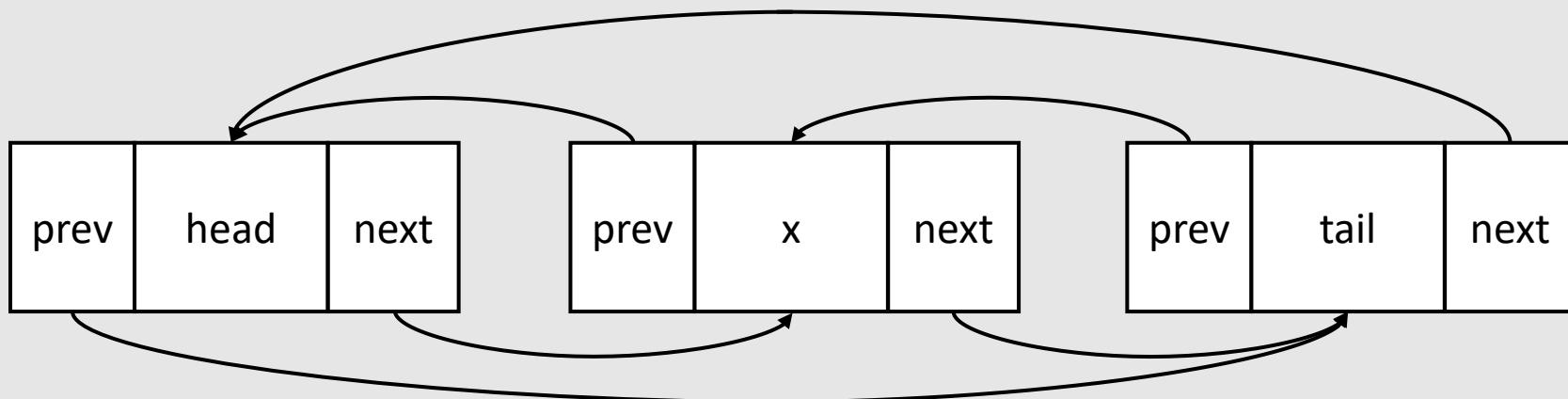
# Sentinel Nodes

- Designated “dummy nodes” used to mark the ends of a linked list.
- In a doubly linked list, sentinel nodes are placed at the head and tail.
  - When performing an insertion, nodes will always go between two nodes.
  - The grayed boxes below indicate the sentinels.



# Circular Doubly Linked List

- The head of the linked list points back to the tail.
- The tail of the linked list points to the head.



# A singly linked list ADT

14 May 2021

```
typedef struct ListNode ListNode;  
  
struct Listnode {  
    char *key;  
    ListNode *next;  
};
```

```
ListNode *node_create(const char *key) {  
    ListNode *t = (ListNode *) malloc(sizeof(ListNode));  
    if (t) {  
        t->key = strdup(key);  
        t->next = NULL;  
    }  
    return t;  
}
```

© 2020 Darrell Long

# Constructor

- Allocate memory needed for a single node.
- A node's key is the duplicated key.
- A node initially points to NULL.

```
void node_delete(ListNode **n) {  
    if (*n) {  
        free((*n)->key);  
        free(*n);  
        *n = NULL;  
    }  
    return;  
}
```

© 2020 Darrell Long

## Node destructor

- Free memory allocated for a node.
- A double pointer is passed so we can NULL the original pointer.

```
void ll_delete(ListNode **head) {
    while (*head != NULL) {
        ListNode *next = NULL; // Save pointer to next node.
        next = (*head)->next;
        node_delete(head);
        *head = next;
    }
    return;
}
```

# Linked list destructor

- Walks the linked list and deletes each node.

© 2020 Darrell Long

```
ListNode *ll_lookup(ListNode *head, const char *key) {
    for (ListNode *curr = head; curr != NULL; curr = curr->next) {
        if (strcmp(curr->key, key) == 0) {
            return curr;
        }
    }
    return NULL;
}
```

© 2020 Darrell Long

# Lookup

- Walks the linked list to look for a specified key.
  - If the key matches, then return the node, move on otherwise.
- Linear search complexity for singly and doubly linked lists is  $O(n)$ .
  - For keys that are strings with a maximum of  $m$  characters, the search complexity is  $O(mn)$ .
- Worst case: the key is *absent*.

```
ListNode *ll_insert(ListNode **head, const char *key) {
    if (ll_lookup(*head, key) != NULL) {
        return *head;
    }
    ListNode *n = node_create(key);
    n->next = *head;
    *head = n;
    return *head;
}
```

© 2020 Darrell Long

# Insertion

- Check if the key is already in the list.
  - We don't want duplicates.
- Create a node with the key.
- Point the created node at the head.
- The new head is the created node.

```
void ll_print(ListNode *head) {  
    for (ListNode *curr = head; curr != NULL; curr = curr->next) {  
        printf("%s\n", curr->key);  
    }  
    return;  
}
```

© 2020 Darrell Long

# Printing

- Walk through the linked list and print out each node's key.

```
ListNode *ll_remove(ListNode **head, const char *key) {
    if (*head) {
        ListNode *curr = *head;
        ListNode *prev = NULL;
        while (curr != NULL) {
            if (strcmp(curr->key, key) == 0) {
                if (prev != NULL) {
                    prev->next = curr->next;
                } else {
                    // If prev is NULL, we're on the head.
                    *head = curr->next;
                }
                node_delete(&curr);
                return *head;
            }
            prev = curr;
            curr = curr->next;
        }
    }
    return *head;
}
```

## Removing

- Track the current and previous nodes.
  - Initially, the previous node is NULL.
- Walk the linked list.
  - If the current node contains a matching key:
    - If there was a previous node, point it at the node after the current node.
    - Else, we were on the head, so make the head point to the node the current node is pointing at.
    - Delete the found node, then we're done.
  - The previous node is now the current node.
  - The current node is its next node.

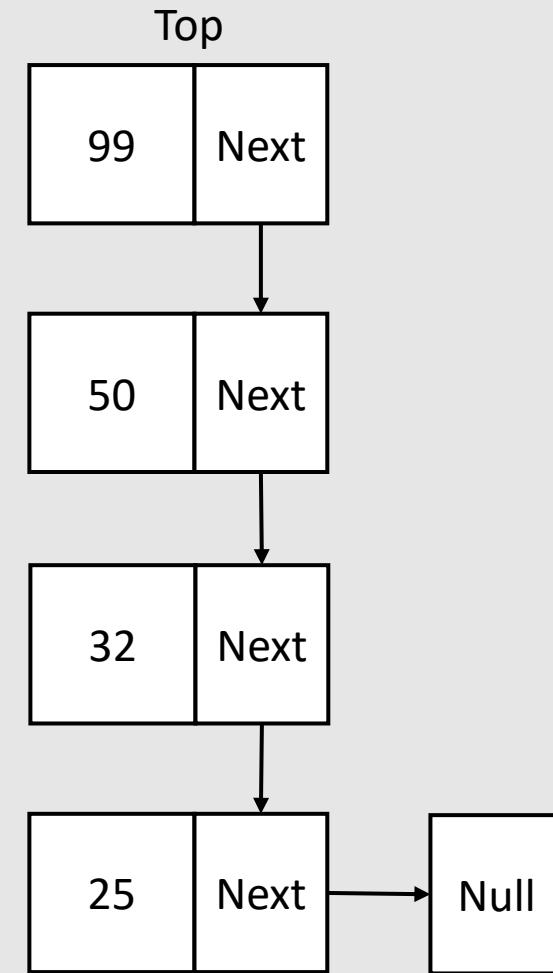
```
ListNode *ll_mtf(ListNode **head, const char *key) {
    if (*head) {
        ListNode *curr = *head;
        ListNode *prev = NULL;
        while (curr != NULL) {
            if (strcmp(curr->key, key) == 0) {
                if (prev != NULL) {
                    prev->next = curr->next;
                    curr->next = *head;
                    *head = curr;
                    return *head;
                }
            }
            prev = curr;
            curr = curr->next;
        }
    }
    return *head;
}
```

## Move-to-front

- Track the current and previous nodes.
  - Initially, the previous node is `NULL`.
- Walk the linked list.
  - If the current node contains a matching key:
    - If there was a previous node, point it at the node after the current node.
    - The current node should now point at the head.
    - The new head is the current node, so we're done.
  - Else, we were on the head, so no need to move to the front.
  - The previous node is now the current node.
  - The current node is its next node.

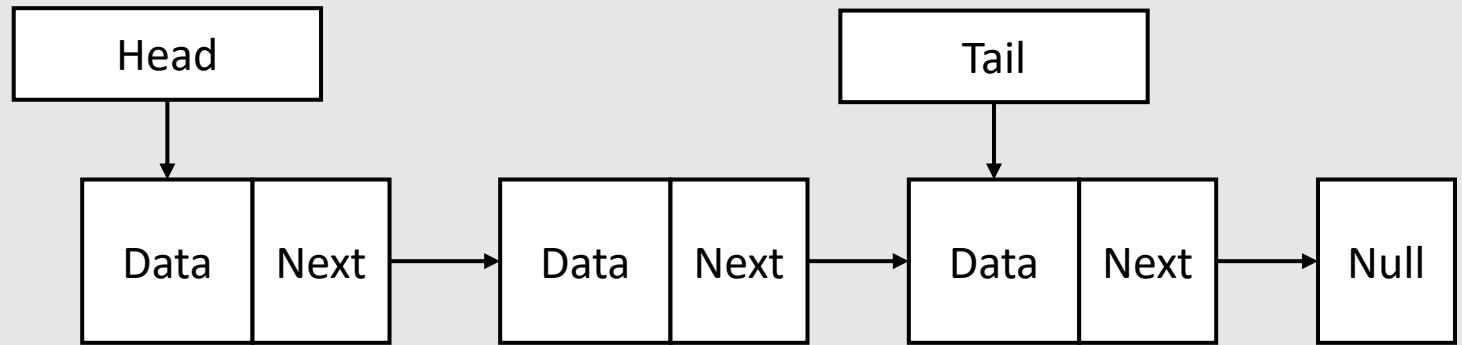
# Linked List Stacks

- Stack size is limited only by available memory.
- Pushing an element is inserting it at the head.
- Popping an element is removing it from the head.



# Linked List Queues

- Add at the tail.
- Remove at the head.



# A doubly linked list ADT

14 May 2021

```
typedef struct listNode listNode;

struct listNode {
    char *key;
    listNode *fwd, *rev;
};

typedef struct {
    listNode *head, *tail;
} listHead;
```

```
listNode *newNode(char *key) {
    listNode *n = (listNode *) malloc(sizeof(listNode));
    if (n) {
        n->key = strdup(key);
        n->fwd = n->rev = NIL;
    }
    return n;
}

listHead *newList(void) {
    listHead *h = (listHead *) malloc(sizeof(listHead));
    if (h) {
        h->head = h->tail = NIL;
    }
    return h;
}
```

## Constructors

- To create a node:
  - Allocate memory for the node.
  - Duplicate the key for the node.
  - The forward and reverse pointers are NIL (NULL) to start with.
- To create a list:
  - Allocate memory for the list.
  - The head and tail of the list are NIL to start with.

```
bool prependList(listHead *h, listNode *n) {
    if (h && n) {
        if (h->head == NIL && h->tail == NIL) {
            h->head = h->tail = n;
        } else {
            n->fwd = h->head;
            h->head->rev = n;
            h->head = n;
            n->rev = NIL;
        }
        return true;
    } else {
        return false;
    }
}
```

## Prepending

- Prepends a node  $n$  to the list (inserts at the head).
- If both the head and tail are NIL:
  - The only node in the list is the node to prepend.
- Else:
  - The node after  $n$  is the head.
  - The node before the head is now  $n$ .
  - The new head is  $n$ .
  - There is no node behind  $n$ .

```
bool appendList(listHead *h, listNode *n) {
    if (h && n) {
        if (h->head == NIL && h->tail == NIL) {
            h->head = h->tail = n;
        } else {
            n->rev = h->tail;
            h->tail->fwd = n;
            h->tail = n;
            n->fwd = NIL;
        }
        return true;
    } else {
        return false;
    }
}
```

## Appending

- Appends a node  $n$  to the list (inserts at the tail).
- If both the head and tail are NIL:
  - The only node in the list is the node to append.
- Else:
  - The node before  $n$  is the tail.
  - The node after the tail is now  $n$ .
  - The new tail is  $n$ .
  - There is no node after  $n$ .

```
bool insertList(listHead *h, listNode *n) {
    if (h && n) {
        if (h->head == NIL && h->tail == NIL) {
            h->head = h->tail = n;
        } else {
            listNode *p = h->head;
            while (p != NIL && strcmp(n->key, p->key) > 0) {
                p = p->fwd;
            }
            if (p == NIL || p == h->tail) {
                appendList(h, n);
            } else if (p == h->head) {
                prependList(h, n);
            } else {
                n->fwd = p->fwd;
                n->rev = p;
                p->fwd->rev = n;
                p->fwd = n;
            }
        }
        return true;
    } else {
        return false;
    }
}
```

## Inserting

- Inserts a node  $n$  lexicographically.
  - Specifically, in reverse alphabetic order.
- If both the head and tail are NIL:
  - The only node in the list is the node to insert.
- Else:
  - Traverse to where the node should go.
  - If we're at the end of the linked list, we append the node.
  - If we're at the head of the linked list, we prepend the node.
  - If we're in the middle:
    - The current node is  $p$ .
    - The node after  $n$  is the node  $p$  is point to.
    - The node before  $n$  is now  $p$ .
    - The node after  $p$  should point back to  $n$ .
    - The node after  $p$  is now  $n$ .

```
listNode *popList(listHead *h) {  
    if (h && h->head) {  
        listNode *p = h->head;  
        h->head = p->fwd;  
        p->fwd = p->rev = NIL;  
        return p;  
    }  
    return NIL;  
}
```

## Popping

- Disconnects and returns the head of the linked list.
- If the head exists:
  - Save a pointer  $p$  to the head.
  - The new head is the node after  $p$ .
  - Make sure  $p$  doesn't point anywhere and return it.
- Else:
  - Return NIL.

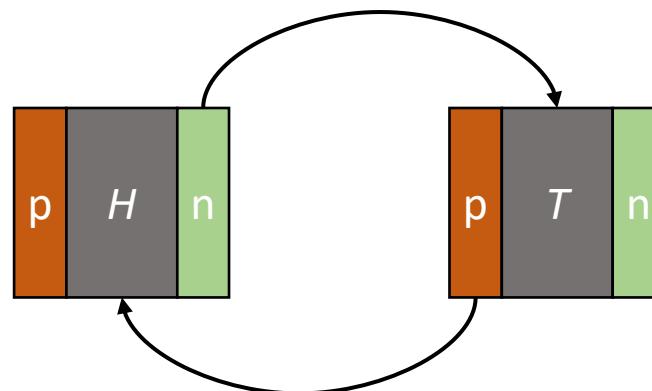
```
listNode *dropList(listHead *h) {  
    if (h && h->tail) {  
        listNode *p = h->tail;  
        h->tail = p->rev;  
        p->fwd = p->rev = NIL;  
        return p;  
    }  
    return NIL;  
}
```

## Dropping

- Disconnects and returns the tail of the linked list.
- If the tail exists:
  - Save a pointer *p* to the tail.
  - The new tail is the node before *p*.
  - Make sure *p* doesn't point anywhere and return it.
- Else:
  - Return NIL.

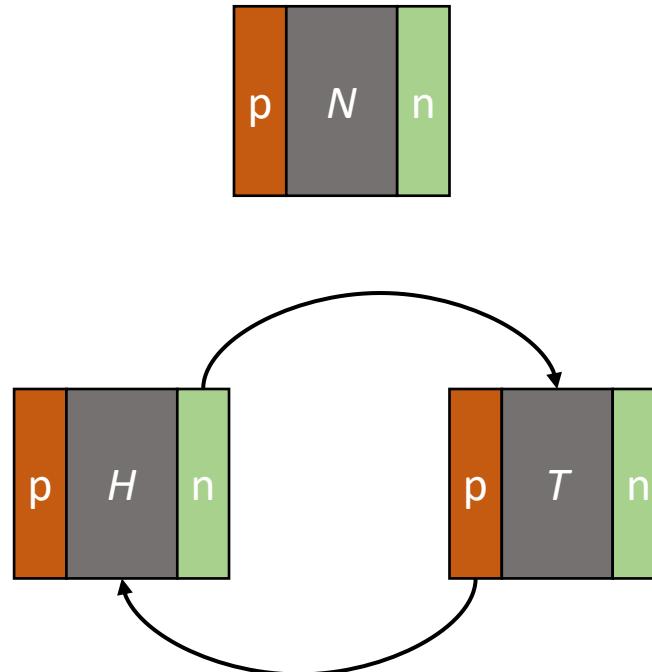
# Inserting Into a Doubly Linked List

- Assume there are two dummy nodes to serve as the head and tail.
  - These are referred to as *sentinel nodes*.
  - We'll label them as  $H$  and  $T$ , respectively.
- The presence of the sentinel nodes means there are always two nodes to insert between.
  - **Con:** Overhead of needing two extra nodes.
  - **Pro:** Cleans up the logic needed to insert a node.
- Each node has its own  $p$  and  $n$ .
  - These are the pointers to the previous and next nodes.



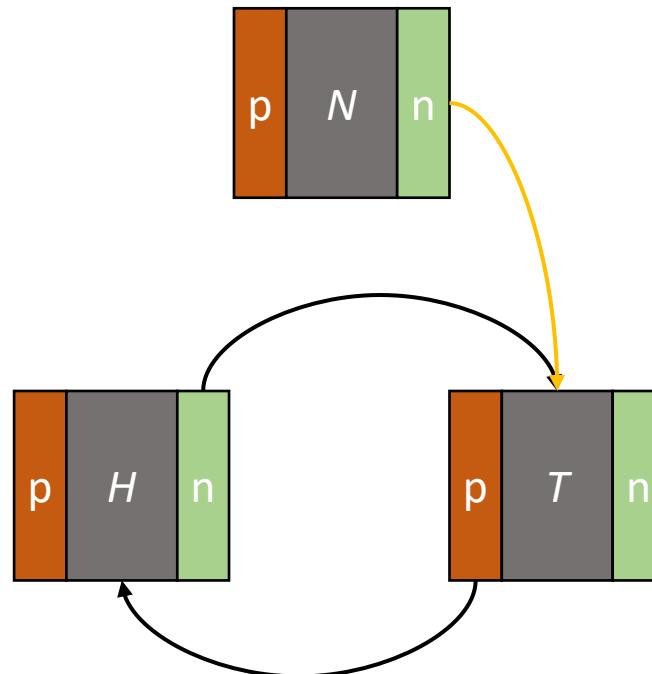
# Inserting Into a Doubly Linked List

- We have a new node  $N$  that we will insert.



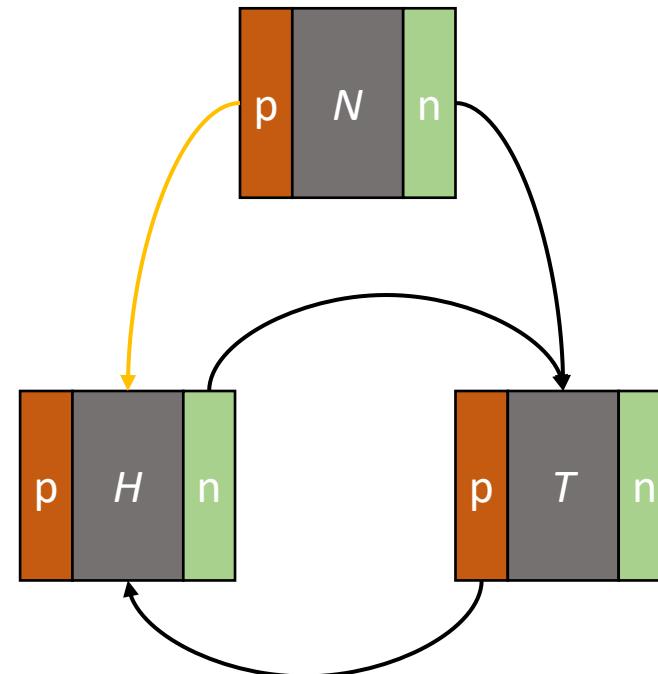
# Inserting Into a Doubly Linked List

- We have a new node  $N$  that we will insert into the doubly linked list.
  1. The node after  $N$  should be the node that  $H$  was pointing to.



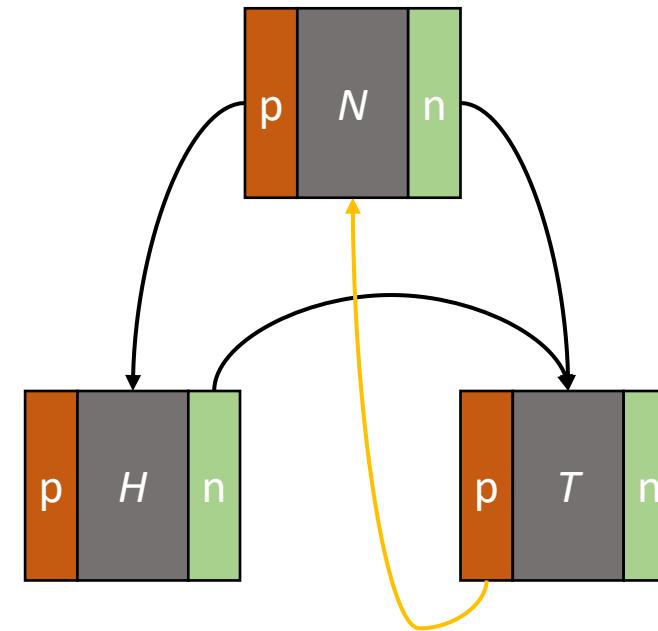
# Inserting Into a Doubly Linked List

- We have a new node  $N$  that we will insert.
  1. The node after  $N$  should be the node that  $H$  was pointing to.
  2. The node before  $N$  should be the head sentinel node,  $H$ .



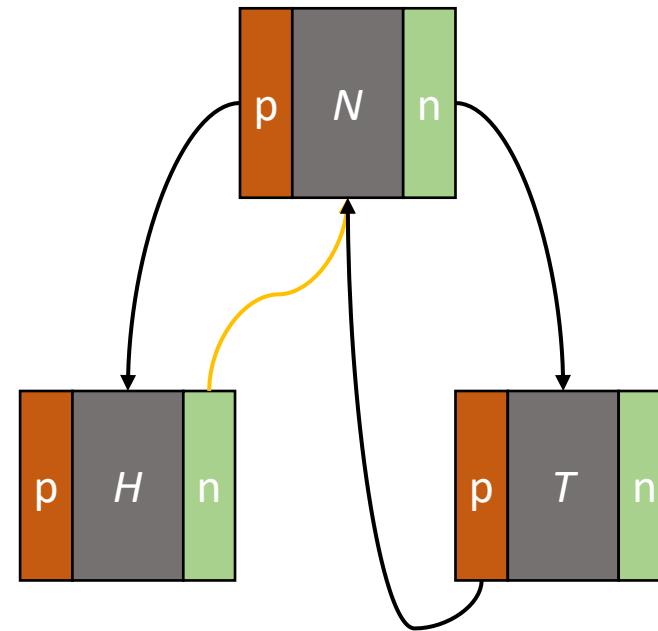
# Inserting Into a Doubly Linked List

- We have a new node  $N$  that we will insert.
  1. The node after  $N$  should be the node that  $H$  was pointing to.
  2. The node before  $N$  should be the head sentinel node,  $H$ .
  3. The node  $H$  is pointing to should now point back to  $N$ .



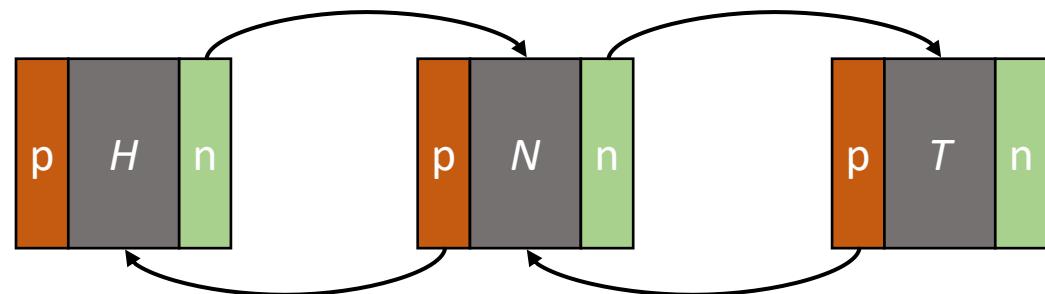
# Inserting Into a Doubly Linked List

- We have a new node  $N$  that we will insert.
  1. The node after  $N$  should be the node that  $H$  was pointing to.
  2. The node before  $N$  should be the head sentinel node,  $H$ .
  3. The node  $H$  is pointing to should now point back to  $N$ .
  4. The node after  $H$  should now be  $N$ .



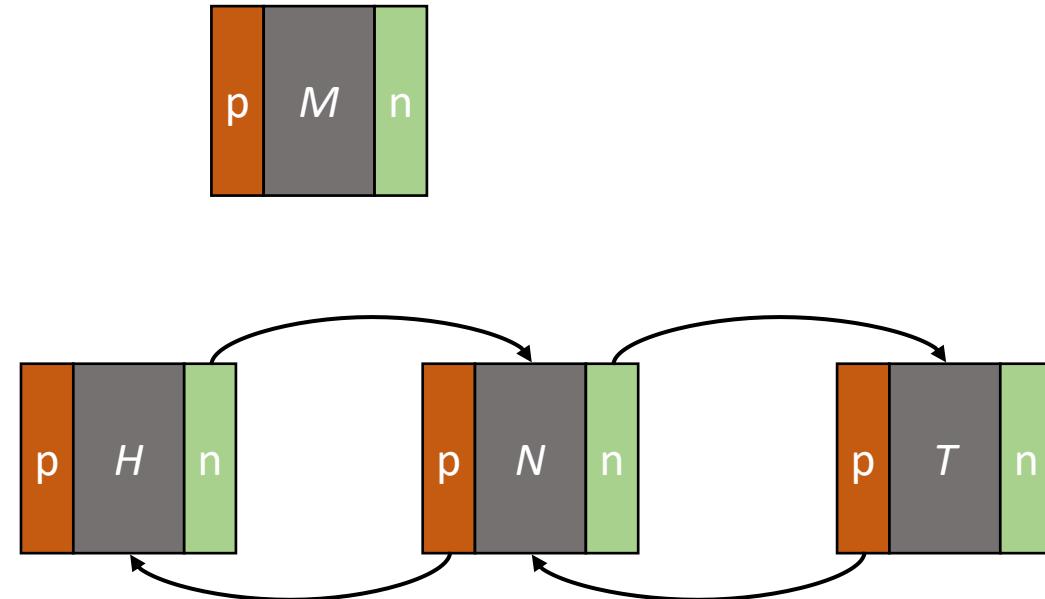
# Inserting Into a Doubly Linked List

- We have a new node  $N$  that we will insert.
  1. The node after  $N$  should be the node that  $H$  was pointing to.
  2. The node before  $N$  should be the head sentinel node,  $H$ .
  3. The node  $H$  is pointing to should now point back to  $N$ .
  4. The node after  $H$  should now be  $N$ .
- $N$  is now at the front of the doubly linked list.



# Inserting Into a Doubly Linked List

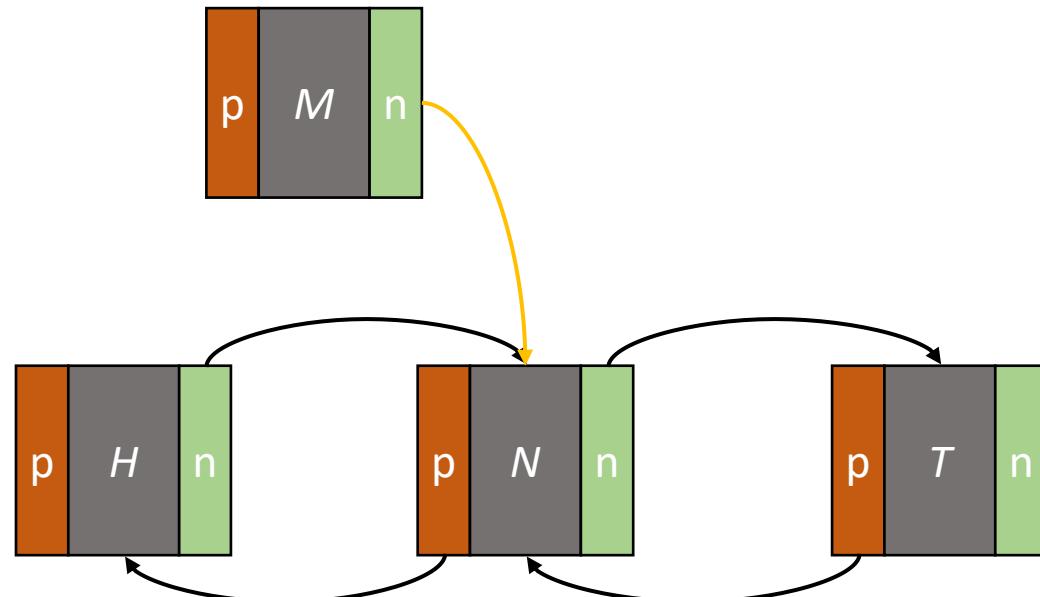
- Let's try inserting another node  $M$  following the same steps as done with  $N$ .
  - The node after  $M$  should be the node that  $H$  was pointing to.
  - The node before  $M$  should be the head sentinel node,  $H$ .
  - The node  $H$  is pointing to should now point back to  $M$ .
  - The node after  $H$  should now be  $M$ .



# Inserting Into a Doubly Linked List

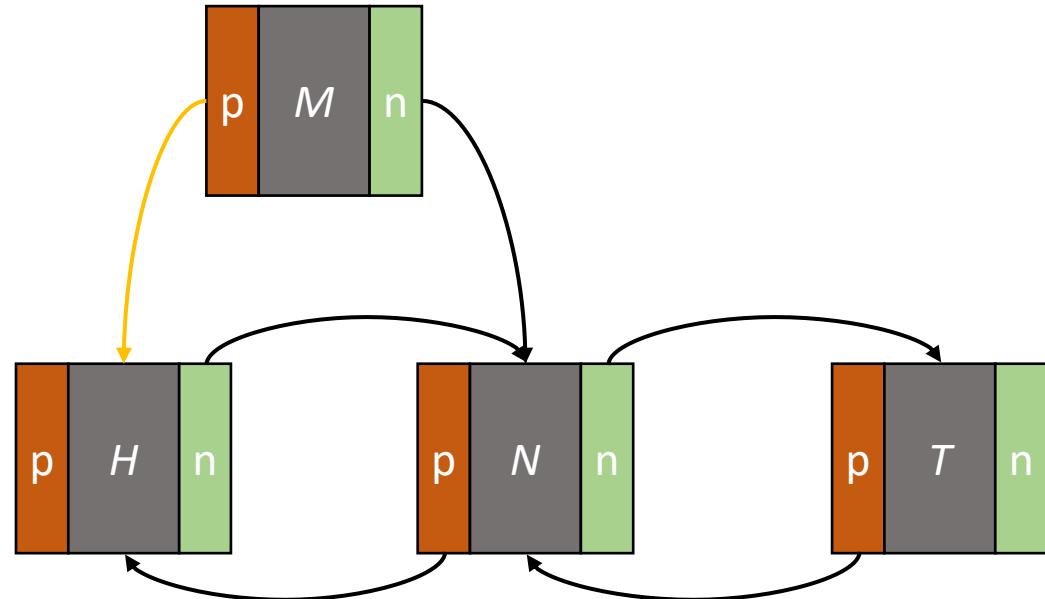
- Let's try inserting another node  $M$  following the same steps as done with  $N$ .

1. The node after  $M$  should be the node that  $H$  was pointing to.
2. The node before  $M$  should be the head sentinel node,  $H$ .
3. The node  $H$  is pointing to should now point back to  $M$ .
4. The node after  $H$  should now be  $M$ .



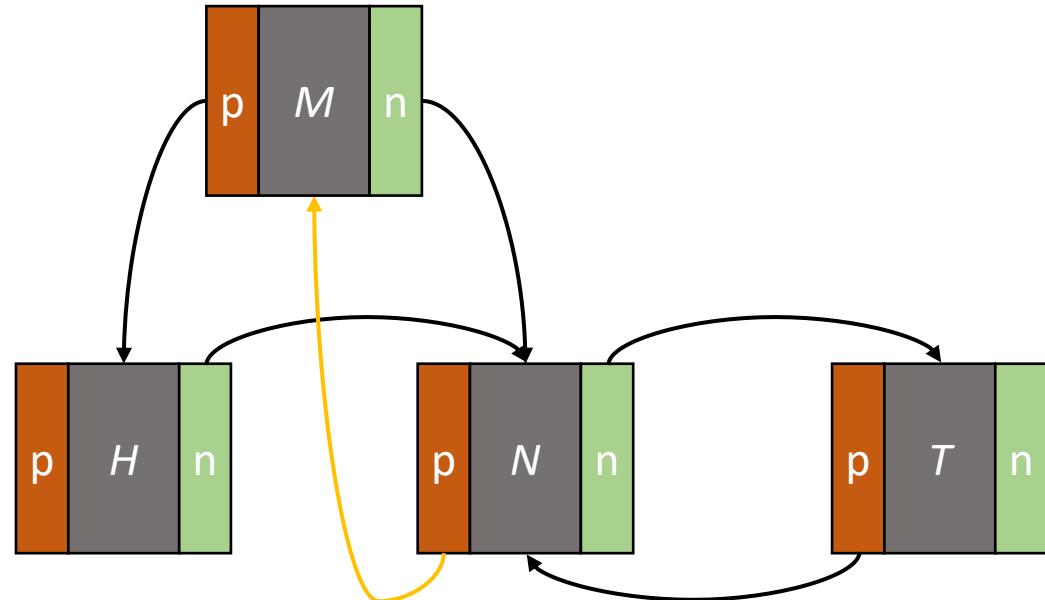
# Inserting Into a Doubly Linked List

- Let's try inserting another node  $M$  following the same steps as done with  $N$ .
  - The node after  $M$  should be the node that  $H$  was pointing to.
  - The node before  $M$  should be the head sentinel node,  $H$ .**
  - The node  $H$  is pointing to should now point back to  $M$ .
  - The node after  $H$  should now be  $M$ .



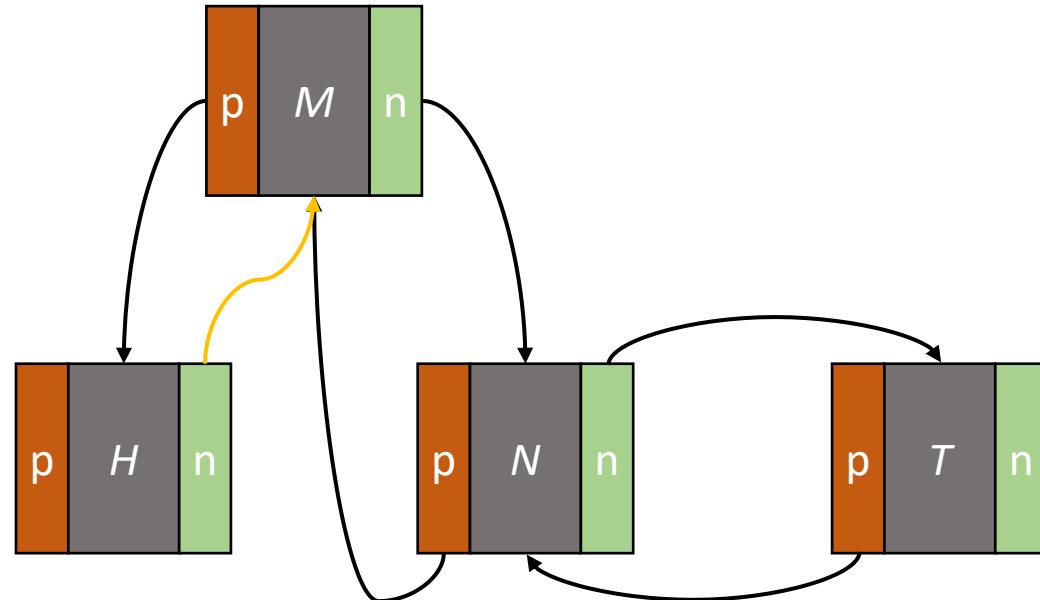
# Inserting Into a Doubly Linked List

- Let's try inserting another node  $M$  following the same steps as done with  $N$ .
  - The node after  $M$  should be the node that  $H$  was pointing to.
  - The node before  $M$  should be the head sentinel node,  $H$ .
  - The node  $H$  is pointing to should now point back to  $M$ .**
  - The node after  $H$  should now be  $M$ .



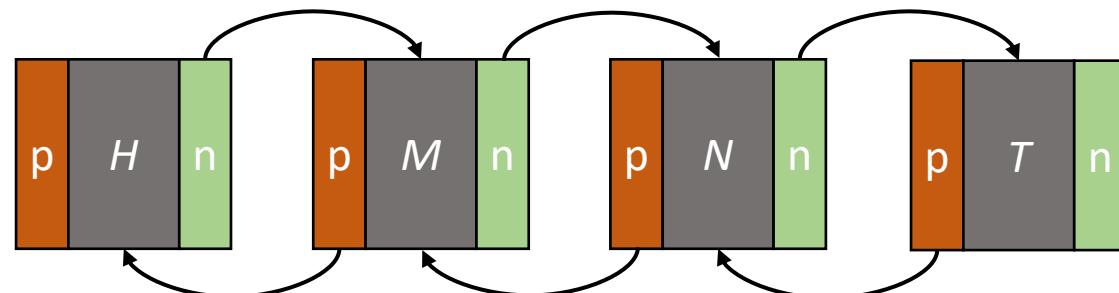
# Inserting Into a Doubly Linked List

- Let's try inserting another node  $M$  following the same steps as done with  $N$ .
  - The node after  $M$  should be the node that  $H$  was pointing to.
  - The node before  $M$  should be the head sentinel node,  $H$ .
  - The node  $H$  is pointing to should now point back to  $M$ .
  - The node after  $H$  should now be  $M$ .**



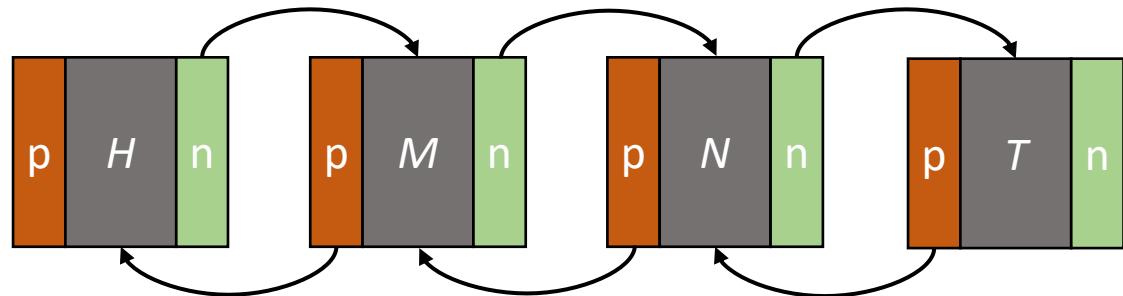
# Inserting Into a Doubly Linked List

- Let's try inserting another node  $M$  following the same steps as done with  $N$ .
  - The node after  $M$  should be the node that  $H$  was pointing to.
  - The node before  $M$  should be the head sentinel node,  $H$ .
  - The node  $H$  is pointing to should now point back to  $M$ .
  - The node after  $H$  should now be  $M$ .
- $M$  is now at the front of the doubly linked list.



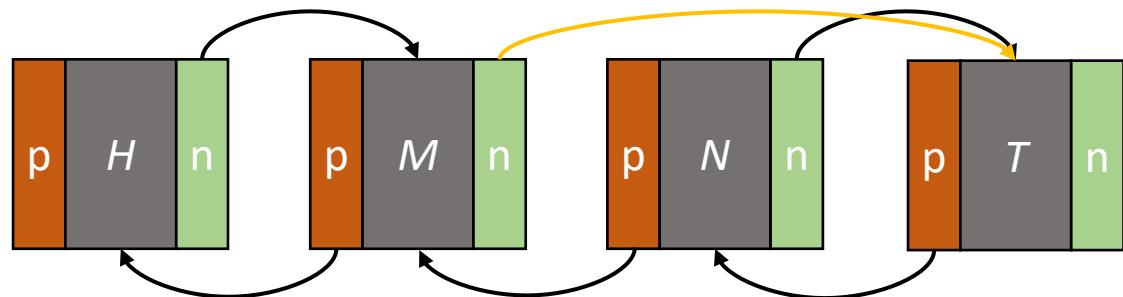
# Move-to-front

- Now that we've inserted  $M$ , we decide we don't like the current order of the linked list.
  - We want  $N$  to be at the front.



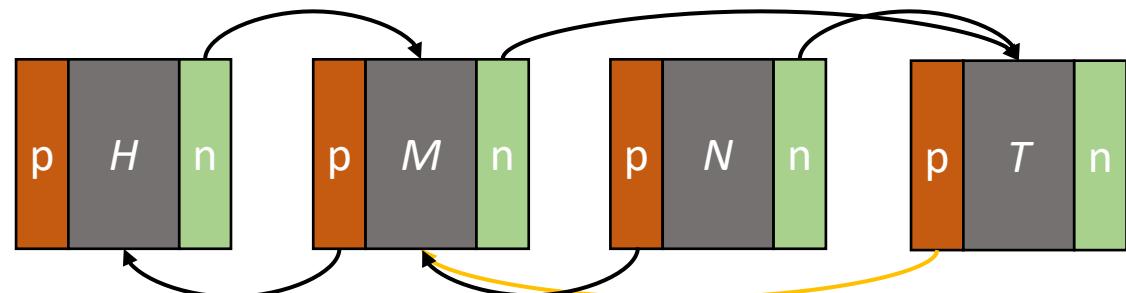
# Move-to-front

1. The node before  $N$  should point to the node after  $N$ .



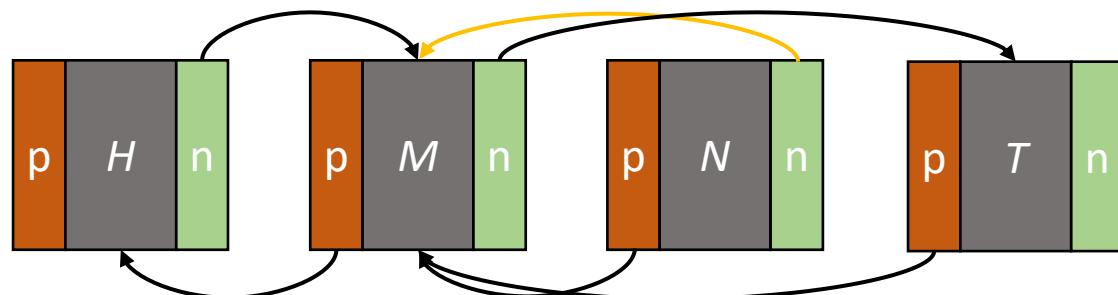
# Move-to-front

1. The node before  $N$  should point to the node after  $N$ .
2. The node after  $N$  should point to the node before  $N$ .



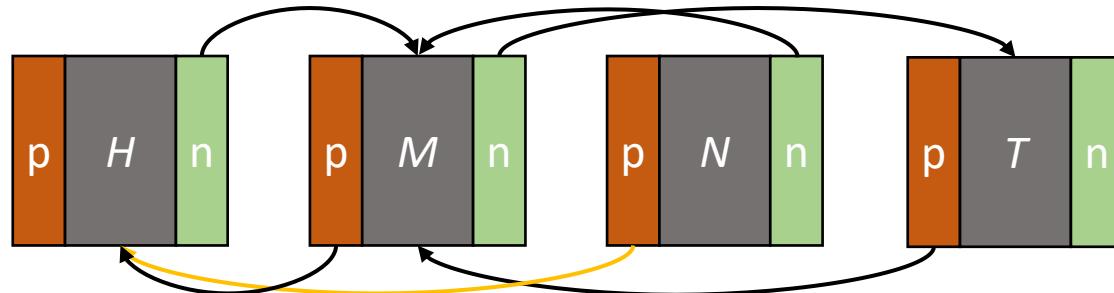
# Move-to-front

1. The node before  $N$  should point to the node after  $N$ .
2. The node after  $N$  should point to the node before  $N$ .
3. The node after  $N$  should be the node that the head sentinel node  $H$  was pointing to (this will look a bit messy).

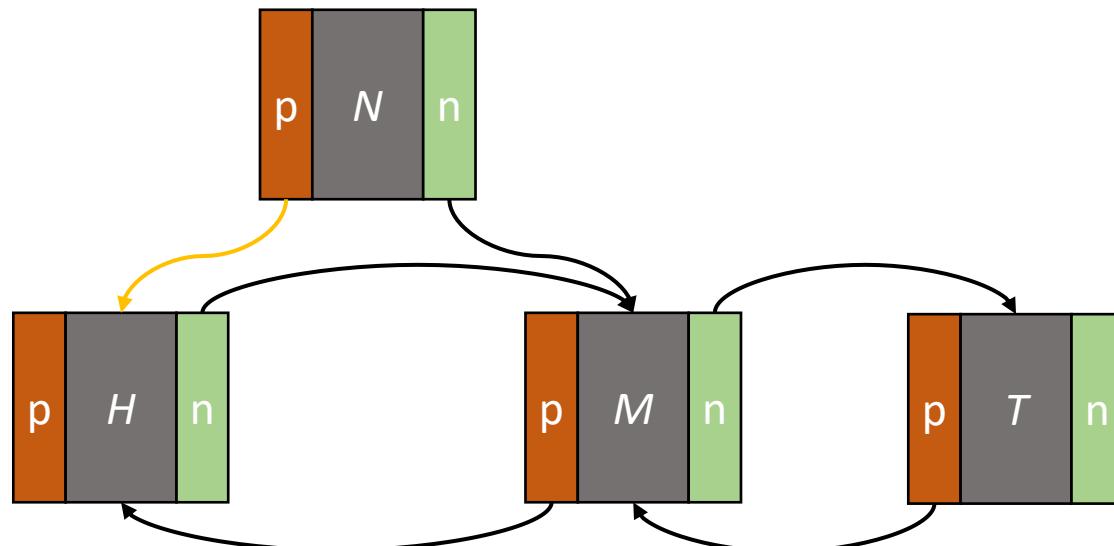


## Move-to-front

1. The node before  $N$  should point to the node after  $N$ .
2. The node after  $N$  should point to the node before  $N$ .
3. The node after  $N$  should be the node that the head sentinel node  $H$  was pointing to.
4. The node before  $N$  should now be  $H$ .

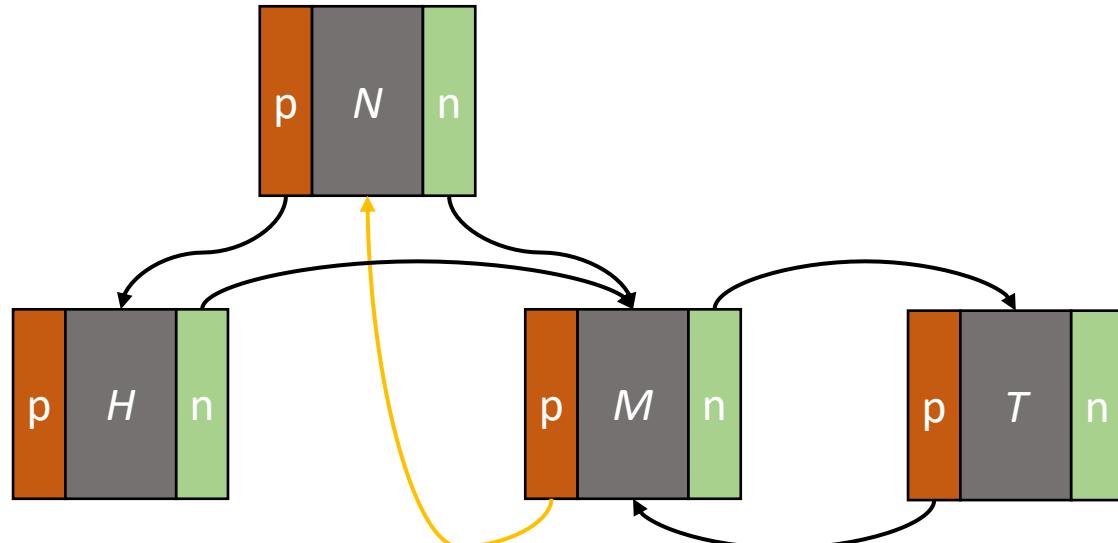


Just to clean things up a bit.



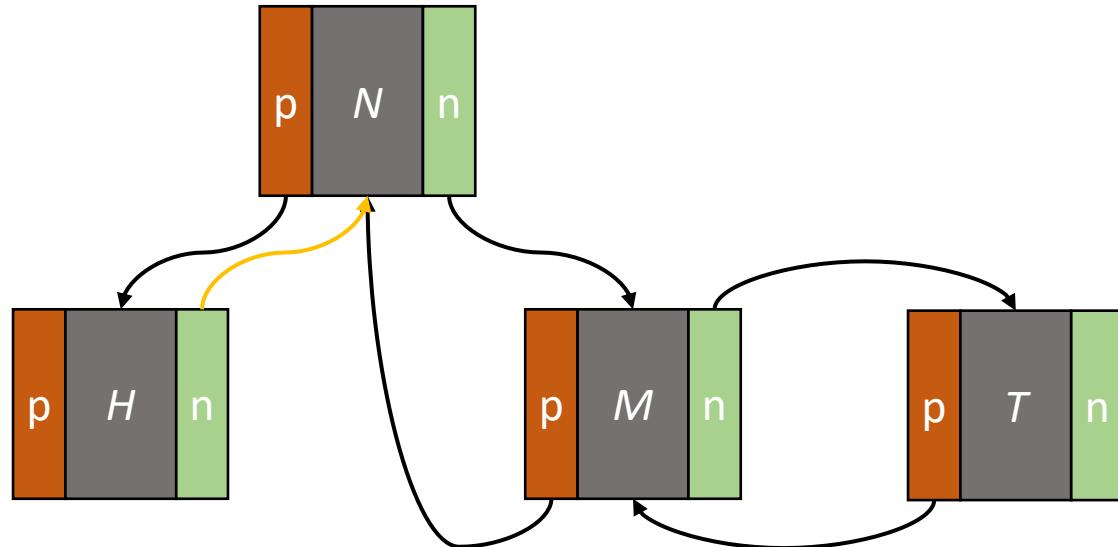
# Move-to-front

1. The node before  $N$  should point to the node after  $N$ .
2. The node after  $N$  should point to the node before  $N$ .
3. The node after  $N$  should be the node that the head sentinel node  $H$  was pointing to.
4. The node before  $N$  should now be  $H$ .
5. The node after  $H$  should point back to  $N$ .



# Move-to-front

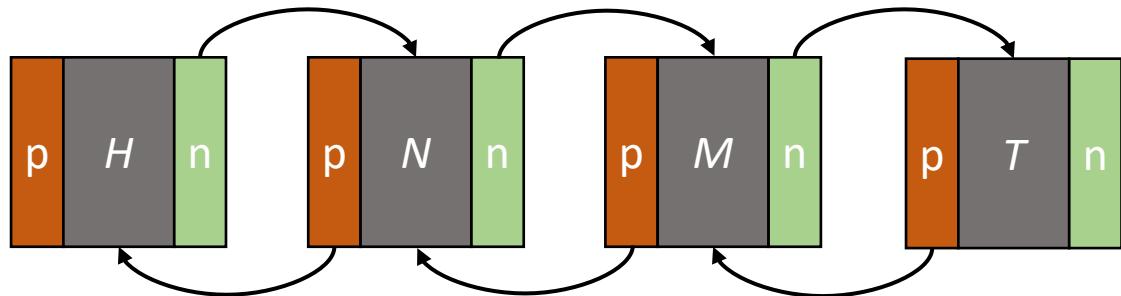
1. The node before  $N$  should point to the node after  $N$ .
2. The node after  $N$  should point to the node before  $N$ .
3. The node after  $N$  should be the node that the head sentinel node  $H$  was pointing to.
4. The node before  $N$  should now be  $H$ .
5. The node after  $H$  should point back to  $N$ .
6. The node after  $H$  should now be  $N$ .



# Move-to-front

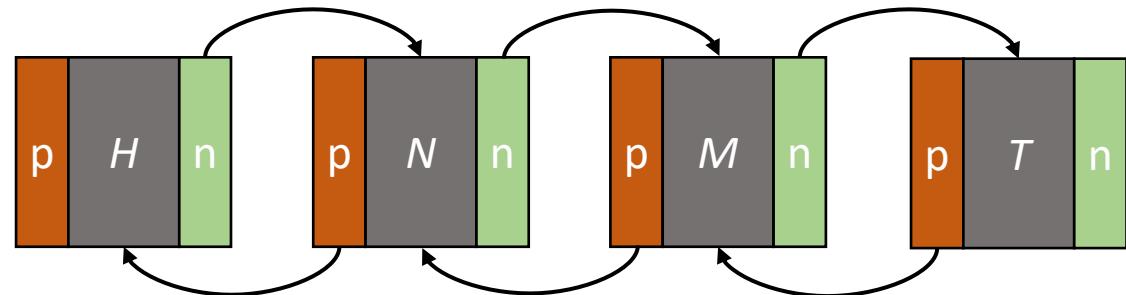
1. The node before  $N$  should point to the node after  $N$ .
2. The node after  $N$  should point to the node before  $N$ .
3. The node after  $N$  should be the node that the head sentinel node  $H$  was pointing to.
4. The node before  $N$  should now be  $H$ .
5. The node after  $H$  should point back to  $N$ .
6. The node after  $H$  should now be  $N$ .

$N$  is now at the front.



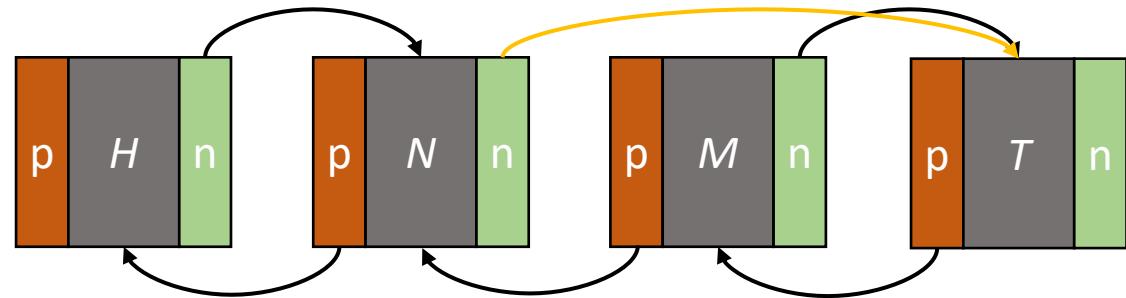
# Removing a node

- We decide that we don't like node  $M$  very much and want to remove it.



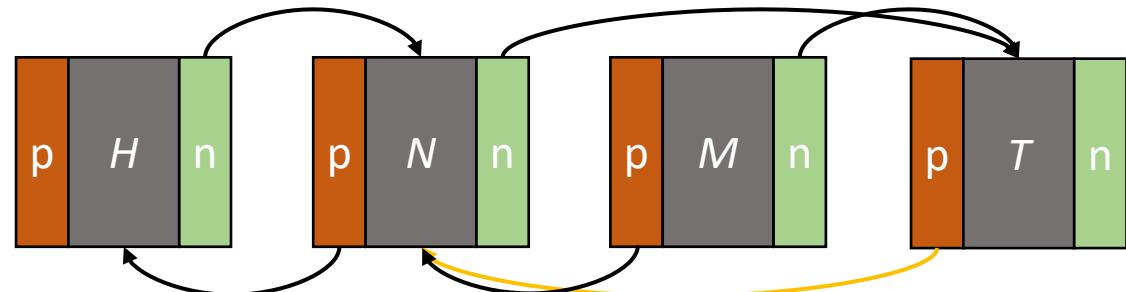
# Removing a node

- We decide that we don't like node  $M$  very much and want to remove it.
  1. The node before  $M$  should point to the node after  $M$ .



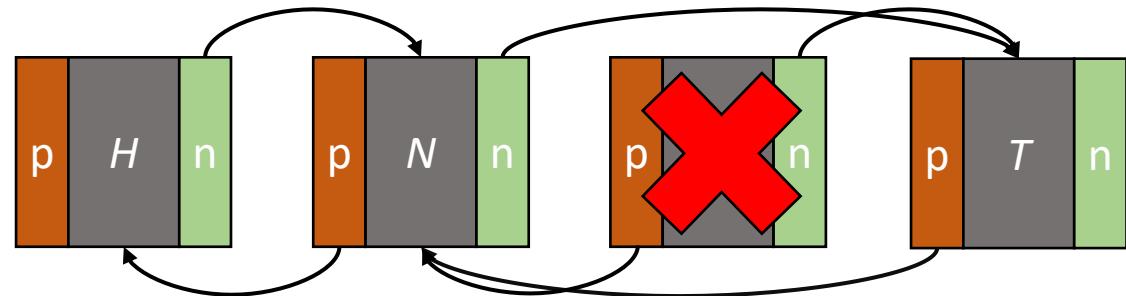
# Removing a node

- We decide that we don't like node  $M$  very much and want to remove it.
  1. The node before  $M$  should point to the node after  $M$ .
  2. The node after  $M$  should point to the node before  $M$ .



# Removing a node

- We decide that we don't like node  $M$  very much and want to remove it.
  1. The node before  $M$  should point to the node after  $M$ .
  2. The node after  $M$  should point to the node before  $M$ .
  3. Goodbye  $M$ .



# Removing a node

- We decide that we don't like node  $M$  very much and want to remove it.
  1. The node before  $M$  should point to the node after  $M$ .
  2. The node after  $M$  should point to the node before  $M$ .
  3. Goodbye  $M$ .
- $M$  is removed now.

