601.220 Intermediate Programming

Spring 2023, Day 17 (March 1st)

Today's agenda

- Exercise 16 review
- More linked lists
- Exercise 17

Reminders/Announcements

- Midterm project:
  - Has been posted to the course website
  - We will go over the project in class on Friday
  - You should have a team repository (and a team) by now
- Midterm exam: in class on Friday, March 10th
  - Review materials are posted on course website
- HW4 due on Friday (Mar 3rd)
  - written assignment, no late submissions

Exercise 16 review

Node data type:

```
typedef struct node_ {
  char data;
  struct node_ *next;
} Node;
```

The `typedef` allows us to refer to the "`struct node_`" type as just "`Node`".

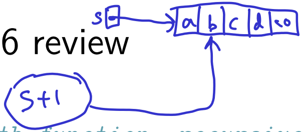
Exercise 16 review

```
// length function, while loop version
int length(const Node *n) {
  int count = 0;
  while (n != NULL) {
    count++;
    n = n->next;
  }
  return count;
}
```

advance

Note: `const Node *n` means "`n` is a pointer to `const Node`".
Function is saying that it won't modify the object that `n` points to.

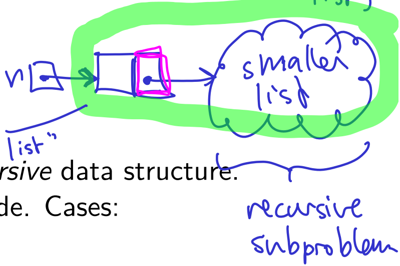
Exercise 16 review



base case (empty list)


```
// length function, recursive version
int length(const Node *n) {
  if (n == NULL) {
    return 0;
  }
  return 1 + length(n->next);
}
```

recursive case (nonempty list)



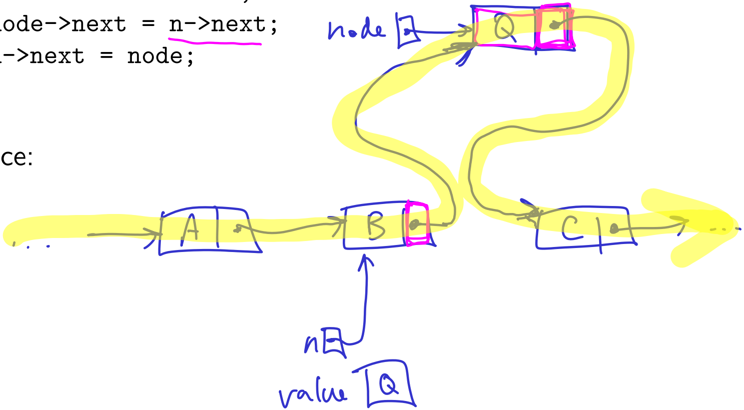
A linked list can be considered as a *recursive* data structure. Assume `n` is a pointer to a linked list node. Cases:

1. `n` is NULL: the list is empty
2. `n` points to a node: nonempty list, `n->next` points to a smaller list (with one fewer nodes than the overall list)

Exercise 16 review

```
void add_after(Node *n, char value) {
  const Node *node = malloc(sizeof(Node));
  node->data = value;
  node->next = n->next;
  n->next = node;
}
```

Trace:



Exercise 16 review

A $\boxed{B \quad C}$

$\Downarrow$ recursive call

C B A

```
void reverse_print(const Struct Node *n) {
  // Pseudo code:
  // if (n is the empty list)
  //   do nothing, return
  // else
  //   print the rest of the list in reverse order
  //   print the value of the first element
}
```

Day 17 recap questions

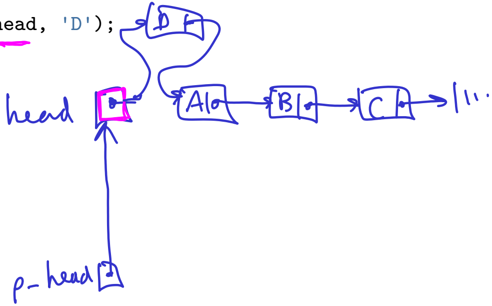
1. How do you implement *add_front* on a linked list?
2. How do you modify a singly linked list to create a doubly linked list?
3. How do you make a copy of a singly linked list?
4. Why does *add_after* takes a `struct Node *` as input, but *add_front* takes `struct Node **`?
5. What cases should be handled when implementing *remove_front*?

4. Why does *add_after* takes a `struct Node *` as input, but *add_front* takes `struct Node **`?

Because `add_after` needs to change which node the head pointer points to. For example:

```
struct Node *head = /* linked list containing 'A', 'B', 'C' */;
// ...
add_front(&head, 'D');
```

Before:



After:

1. How do you implement *add_front* on a linked list?

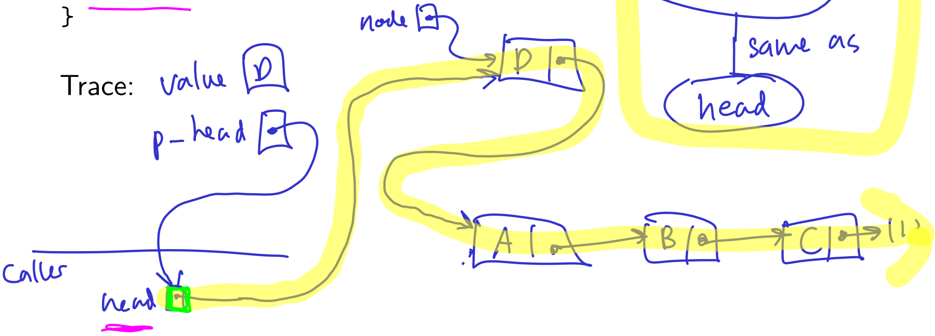
```
void add_front(struct Node **p_head, char value) {
  struct Node *node = malloc(sizeof(struct Node));
  node->data = value;
  node->next = *p_head;
  *p_head = node;
}
```

*p-head

same as

head

Trace:

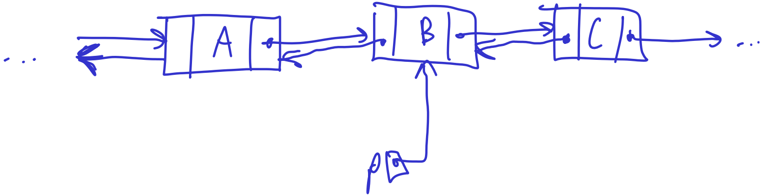


2. How do you modify a singly linked list to create a doubly linked list?

Have each node store a pointer to the *previous* node in the list, in addition to the next node in the list. I.e.:

```
struct Node {
  char payload;
  struct Node *prev, *next;
};
```

Example:

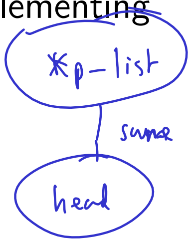


3. How do you make a copy of a singly linked list?

One way is to use recursion:

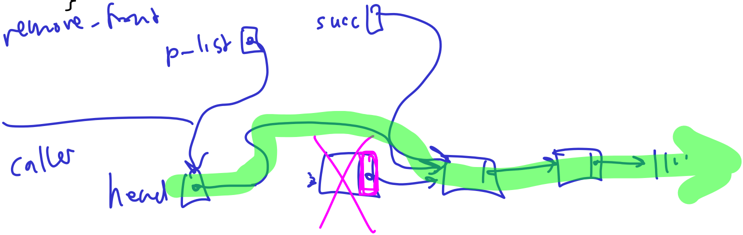
```
struct Node *copy_list(struct Node *n) {
  struct Node *result;
  if (n == NULL) {
    result = NULL;
  } else {
    result = malloc(sizeof(struct Node));
    result->payload = n->payload;
    result->next = copy_list(n->next);
  }
  return result;
}
```

5. What cases should be handled when implementing *remove_front*?



There should not be any special cases.

```
void remove_front(struct Node **p_list) {
  assert(*p_list != NULL);
  struct Node *succ = (*p_list)->next;
  free(*p_list);    // free original head node
  *p_list = succ;   // make head pointer point to second node
}
```



Exercise 18

- More linked list operations (including ones requiring pointer to head pointer)
- Again, drawing diagrams is very helpful for reasoning about linked list operations
- Talk to us if you have questions!

Notes

Notes

Notes

Notes

Notes