601.220 Intermediate Programming

Summer 2024, Meeting 10 (June 24th)

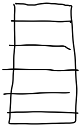
Today's agenda

- Exercise 15 review
- "Day 17" material
  - Linked lists
  - Exercise 17
- "Day 18" material
  - More linked lists
  - Exercise 18

Reminders/Announcements

- Update on Midterm project deadline: ~~due Friday, June 28th~~ due Saturday, June 29th , 11pm
  - An additional day was added to compensate the fact the starter code was release on Sunday.
  - **No late days allowed** for the midterm project (Please refer to the homework policy at the syllabus tab on the course website)
- Midterm exam: in class on Wednesday, July 3th

Exercise 15 review - part 2



- Identify the endianness of ugrad machines
- Endianness is the order or sequence in which multi byte words are stored in memory.
    - Little-endian (LE)
    - Big-endian (BE)
- Why is it important to know which system is our computer using?
- Example with base 10 number: 950238851 can be represented in hexadecimal base as 38 A3 7E 83.

Exercise 15 review - part 2

FF ~> 11 11 11 11

Example with base 10 number: 950238851 can be represented in hexadecimal base as 38 A3 7E 83.

most sig byte ⟷ least sig byte

Big endian	little endian
38	83
A3	7E
7E	A3
83	38

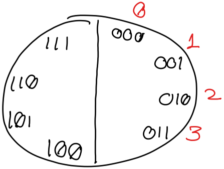
Exercise 15 review - part 2

Identify the endianness of ugrad machines . . . .

```
(gdb) break endian.c:21
Breakpoint 1 at 0x1243: file endian.c, line 21.
(gdb) run
[...output omitted...]
Breakpoint 1, main () at endian.c:21
21    printf("%u\n", *p);
(gdb) print/x ((unsigned char *)p)[0]
$1 = 0x83
(gdb) print/x ((unsigned char *)p)[1]
$2 = 0x7e
(gdb) print/x ((unsigned char *)p)[2]
$3 = 0xa3
(gdb) print/x ((unsigned char *)p)[3]
$4 = 0x38
```

In base-16, 950238851 is 38A37E83. Since we're seeing the bytes in order from least to most significant, the ugrad machines are *little endian*.

Exercise 15 review - part 3



To negate a two's complement value:

- Invert all of the bits (the ~ operator is useful for this)
- Add 1

Exercise 15 review - part 3

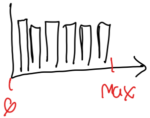
1 0000 ...

Note that `0x80000000U` is the `unsigned int` value with only the most significant bit set to 1. This is the sign bit, and values with this bit set are negative.

```
unsigned int magnitude(unsigned int value) {
  if ((value & 0x80000000U) == 0U) {
    return value; // value is non-negative
  }

  // value is negative, so invert bits and add 1
  value = ~value;  // invert bits
  value += 1U;     // add 1
  return value;
}
```

1 0 0 0 ... → 0x80...
& 1 1 0 1 ... → value
1 0 0 0 ...

Exercise 15 review - part 4



Generating a uniformly distributed pseudo-random integer in the range 0 (inclusive) to `max_num` (exclusive):

```
int gen_uniform(int max_num) {
  return rand() % max_num;
}
```

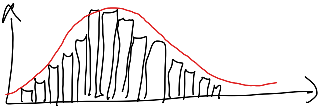
`rand()` generates integers between 0 and RAND_MAX

Exercise 15 review - part 4

Generating 500 random values in range 0 (inclusive) to `max_range` (exclusive) and tallying them in the `hist` array:

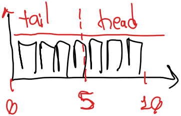
```
for (int i = 0; i < 500; i++) {
  hist[gen_uniform(max_range)]++;
}
```

Exercise 15 review



Generating normally-distributed integer values in the range 0 (inclusive) to `max_range` (exclusive):

- Base idea: throw a coin a 1000 times. How many heads would you expect to obtain?
- Now lets repeat this experiment a thousand times. What is the distribution of the total number of heads of all thousand experiments?



Exercise 15 review

Generating normally-distributed integer values in the range 0 (inclusive) to `max_range` (exclusive):

```
int normal_rand(int max_num) {
  int result = 0;
  for (int i = 1; i < max_num; i++) {
    if ((rand() & 1) == 1) {
      result++;
    }
  }
  return result;
}
```

This is basically flipping a coin `max_num-1` times and counting how many times it's heads.

Exercise 15 review

Generating 500 normally-distributed values in the range 0 (inclusive) to `max_range` (exclusive) and tallying them in the `hist` array:

```
for (int i = 0; i < 500; i++) {
  hist[normal_rand(max_range)]++;
}
```

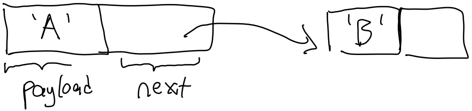
Day 17 recap questions

1. Describe the linked list structure by a diagram.
2. Compare arrays and linked lists. Write down their pros and cons.
3. What is a linked list's head? How is it different from a node? Explain.
4. How do you calculate `length` of a linked list?
5. How do you implement `add_after` on a singly linked list?

1. Describe the linked list structure by a diagram.

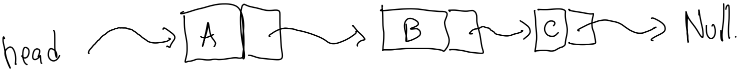
`struct Node` type:

```
struct Node {
  char payload; // payload could be any data type
  struct Node *next;
};
```



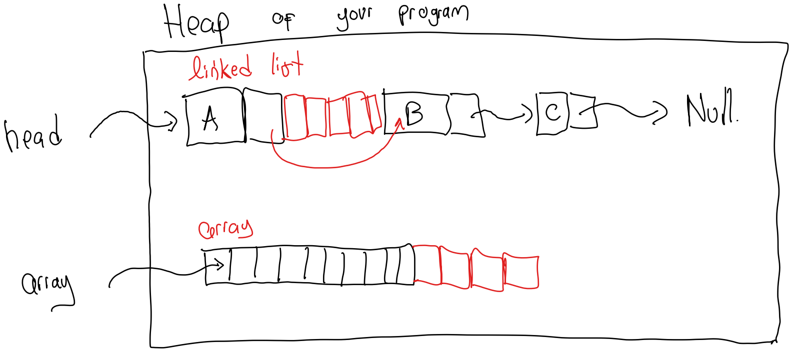
Example linked list

```c
// code creating a linked list
struct Node *head = malloc(sizeof(struct Node));
head->payload = 'A';
head->next = malloc(sizeof(struct Node));
head->next->payload = 'B';
head->next->next = malloc(sizeof(struct Node));
head->next->next->payload = 'C';
head->next->next->next = NULL;
```



A more concise representation

Arrays vs linked lists

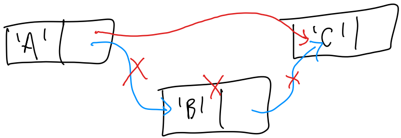


2. Compare arrays and linked lists. Write down their pros and cons.

Arrays:

- Pro: O(1) access to arbitrary element
- Con: O(N) to insert or remove element at arbitrary position
- Pro: better locality (fewer cache misses when iterating)
- Pro: more compact
- Con: fixed size, to reallocate must allocate new array and copy existing data

Linked list pros and cons



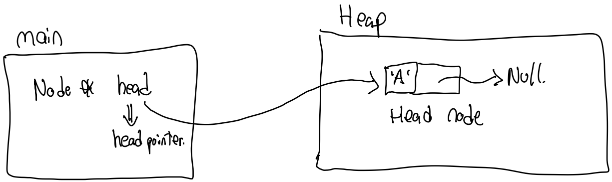
Linked list:

- Con: O(N) access to arbitrary element
- Pro: O(1) to remove element at arbitrary position
- Con: worse locality (more cache misses when iterating)
- Con: less compact (next pointers require space)
- Pro: can grow incrementally, nodes are allocated one at a time

3. What is a linked list's head? How is it different from a node? Explain.

Contrast: *head pointer* vs. *head node*. The head pointer is a pointer variable storing a pointer to the first node. The head node *is* the first node in the linked list.

Picture:

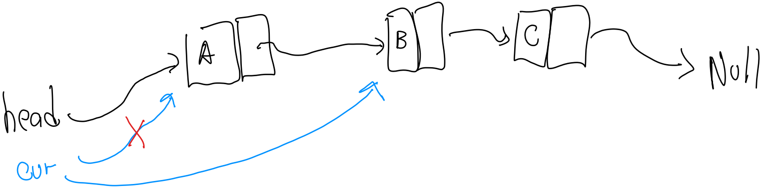


4. How do you calculate `length` of a linked list?

A loop is required:

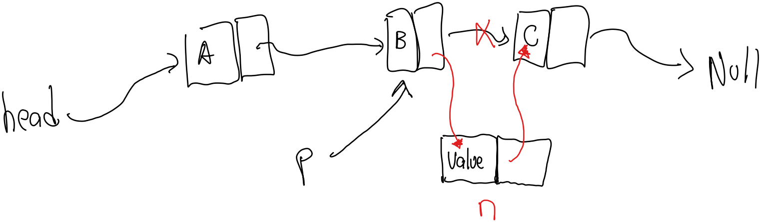
```
struct Node *head = /* points to first node */;
int count = 0;

for (struct Node *cur = head; cur != NULL; cur = cur->next) {
  count++;
}
```

(*cur).next



5. How do you implement add_after on a singly linked list?

```
void add_after(struct Node *p, char value) {
  struct Node *n = malloc(sizeof(struct Node));
  n->payload = value;
  n->next = p->next;
  p->next = n;
}
```



Exercise 17

- Basic linked list functions
- Drawing pictures to reason about how linked lists operations should work is very helpful!
- Note that `reverse_print` is most easily implemented using recursion
- Breakout rooms 1–10 are "social"
- Use Slack to let us know if you have questions

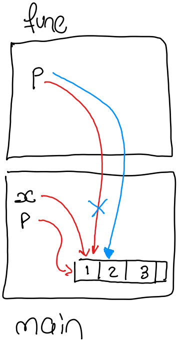
Day 18 recap questions

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*?

Changing the address of a pointer inside a function

The wrong way of doing it . . . . c is a passed by value language!

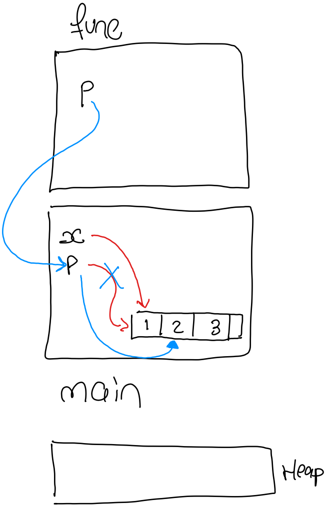
```
void func(int *p)
{
    p+=1;
    printf("Inside func %d\n", *p);
}
int main()
{
    int x[5] = {1,2,3,4,5};
    int *p = x;
    func(p);
    printf("outside func %d\n", *p);

    return 0;
}
```



Changing the address of a pointer inside a function

The the right way of doing it ....

```
void func(int **p)
{
    *p+=1;
    printf("Inside func %d\n", **p);
}
int main()
{
    int x[5] = {1,2,3,4,5};
    int *p = x;
    func(&p);
    printf("outside func %d\n", *p);
}
```

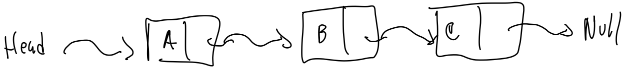


4. Why does *add_after* takes a `struct Node *` as input, but *add_front* takes `struct Node **`?

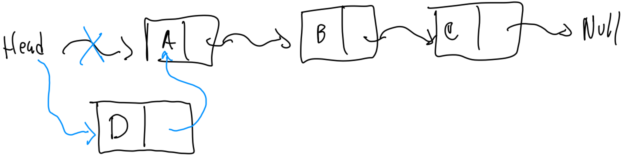
Because `add_after` needs to change which node the head pointer points to. For example:

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

Before:



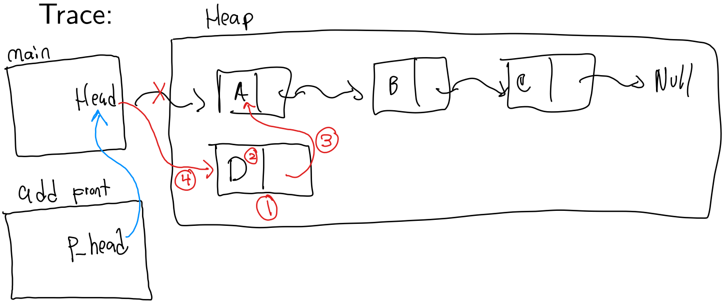
After:



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;
}
```

Trace:

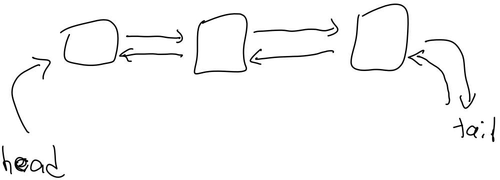


2. How do you modify a singly linked list to create a doubly linked list?

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;
};
```

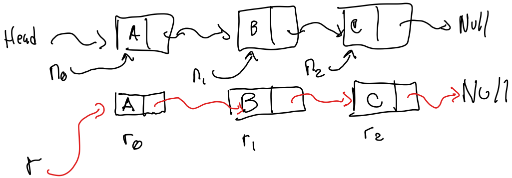
Example:



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
- Breakout rooms 1–10 are “social”
- Use Slack to let us know if you have any questions!

Notes

Notes

Notes

Notes

Notes