CS 24000 L04

Midterm Exam II Study/Review Session

Basic Pointer Operations 1

```
int x = 0;

int y = 0;

int *p = NULL;


p = &x;

*p = 5;

p = &y;

*p = 7;

printf("%d %d\n", x, y);
```

Basic Pointer Operations 1 - Solution

```
int x = 0;

int y = 0;

int *p = NULL;


p = &x; // p points to x

*p = 5; // setting x's value to 5

p = &y; // p now points to y

*p = 7; // setting y's value to 7

printf("%d %d\n", x, y); // prints "5 7"
```

Basic Pointer Operations 2

```
int x = 0;

int y = 0;

int *p = NULL;

int *q = NULL;


p = &x;

q = p;

*q = 7;

q = 3;

printf("%d %d\n", x, y);
```

Basic Pointer Operations 2 - Solution

```
int x = 0;
int y = 0;
int *p = NULL;
int *q = NULL;

p = &x; // p points to x
q = p;  // q points to x
*q = 7; // setting x's value to 7
q = 3; // q points to 0x03 (a pointer is just a number, but this one doesn't point anywhere meaningful)
printf("%d %d\n", x, y); // prints "7 0"
```

Basic Pointer Operations 3

```
int x = 0;

int y = 0;

int *p = NULL;

int *q = NULL;


p = &y;

q = &x;

p = 2;

printf("%d %d\n", x, y);
```

Basic Pointer Operations 3 - Solution

```
int x = 0;

int y = 0;

int *p = NULL;

int *q = NULL;


p = &y; // p points to y

q = &x; // q points to x

p = 2; // p points to 0x02, again not meaningful

printf("%d %d\n", x, y); // prints “0 0”
```

Swapping Function - Why we use pointers

- Write a function called 'swap' that will accept two pointers to integers
  and will exchange the contents of those integer locations.
  . Show a call to this subroutine to exchange two variables.
  . Why is it necessary to pass pointers to the integers instead of just
  passing the integers to the Swap subroutine?
  . What would happen if you called swap like this:

```
int x = 5;
swap(&x, &x);
```

  . Can you do this: (why or why not?)

```
swap(&123, &456);
```

Swapping Function - Why we use pointers - Solution

- Write a function called 'swap' that will accept two pointers to integers and will exchange the contents of those integer locations.

```
void swap(int *a, int *b) {
        int tmp = *a;
        *a = *b;
        *b = tmp;
}
```

. Show a call to this subroutine to exchange two variables.

. Why is it necessary to pass pointers to the integers instead of just passing the integers to the Swap subroutine?

We want to modify the original values instead of just what's on the stack. This is a scope problem; a function's arguments are limited to the scope of the function.

. What would happen if you called swap like this:

```
int x = 5;
swap(&x, &x); // The swap would simply swap 5 and 5; nothing interesting happens
```

. Can you do this: (why or why not?)

```
swap(&123, &456);    //No! You can't dereference an integer literal in C
```

Dynamic Memory

write statements to do the following memory operations:

. reserve space for 100 integers and assign a pointer to that space to pi

. reserve space for 5 floats and assign a pointer to that space to pf

. unreserve the space that pi points to

. reserve space for enough characters to hold the string in my_string and assign a pointer to that space to pc. Copy my_string into that space.

. free everything that hasn't been unreserved yet.

Dynamic Memory - Solution

write statements to do the following memory operations:

. reserve space for 100 integers and assign a pointer to that space to pi

```
pi = malloc(100 * sizeof(int));
```

. reserve space for 5 floats and assign a pointer to that space to pf

```
pf = malloc(5 * sizeof(float));
```

. unreserve the space that pi points to

```
free(pi); pi = NULL;
```

. reserve space for enough characters to hold the string in my_string and assign a pointer to that space to pc. Copy my_string into that space.

```
pc = malloc(strlen(my_string) + 1);
// remember +1 for the null terminator
strcpy(my_string, pc);
```

. free everything that hasn't been unreserved yet.

```
free(pc); pc = NULL;
free(pf); pf = NULL;
```

Dangling Pointers

- What happens if you reserve memory and assign it to a pointer named p

  and then reserve more memory and assign the new pointer to p? How can

  you refer to the first memory reservation?

This is a classic example of a dangling pointer, meaning p's memory address becomes inaccessible after calling malloc again.

The way around this is to store the old value of p somewhere else before calling malloc again.

Double Free

- Does it make sense to free() something twice? What's a good way to prevent this from happening?

No! We ask you to set pointers to NULL after calling free as a way to prevent this; if a pointer is NULL, don’t free it!

Recursion Part 1 - Factorial

```
int factorial(int n) {
    if (n == 1) return 1; // The base case - a crucial step to solving a recursive problem
    return n * factorial(n - 1); // The second step to a recursive problem is the recursion
}
```

Recursion Part 2 - Fibonacci

```
int fib(int n) {
    if (n == 0) return 0;
    if (n == 1) return 1;
    return fib(n-1) + fib(n-2);
}
```

Bonus: Why is this inefficient, and how can we make it better?

Recursion - Fibonacci - Bonus Question

Why was our implementation inefficient? Every number in the sequence is counted twice, and each time takes O(n) time to compute (fib(1) + fib(2) + …).

*We can speed this up by **caching** our results.*

```
int fib_array[MAX_LEN] = {0, 1}; // cache for fibonacci numbers we've already found, seeded with 0 and 1

for (int i = 2; i < MAX_LEN; i++) fib_array[i] = -1; // initializing unseen entries to -1

int fib(int n) {

        if (fib_array[n] != -1) return fib_array[n];

        int result fib(n-1) + fib(n-2);

        fib_array[n] = result;

        return result;

}
```

Linked Lists Review

A linked list is simply a set of nodes with values and a pointer to the next element in the list.

Other types of linked lists:

- Doubly linked lists, where each node has a forward and back pointer
- Circular linked lists, where the last node in the list points back to the first

You should know how to traverse forward and backward, and how to insert and delete

(A review for that can be found in my HW9 notes)

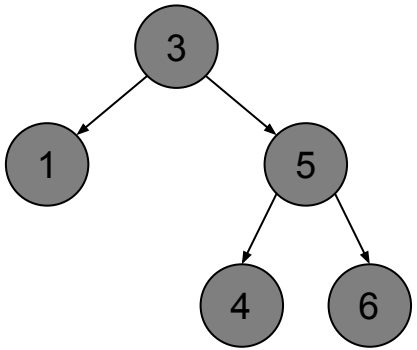
Trees - Basically more advanced lists

In a binary tree, nodes that you insert can go:

Left, if its value is less than the root node, or

Right, if its value is greater than or equal to the root node

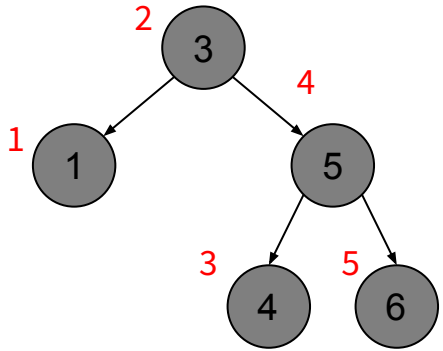
(note, equal to can change between implementations, but it doesn't change the traversal order)



Trees - Forward Traversal Order (Left, current, right)

When traversing a list in forward order, the nodes will inherently be sorted. Why?

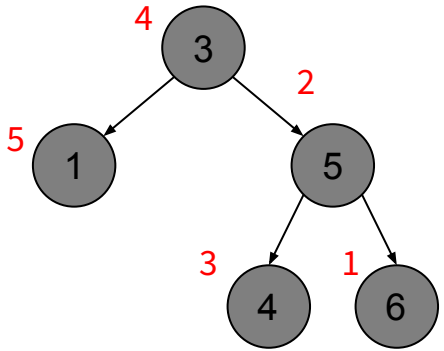
- Forward always traverses the left child first, which is always less than the right child
- The left child and its children will be traversed completely before the right child



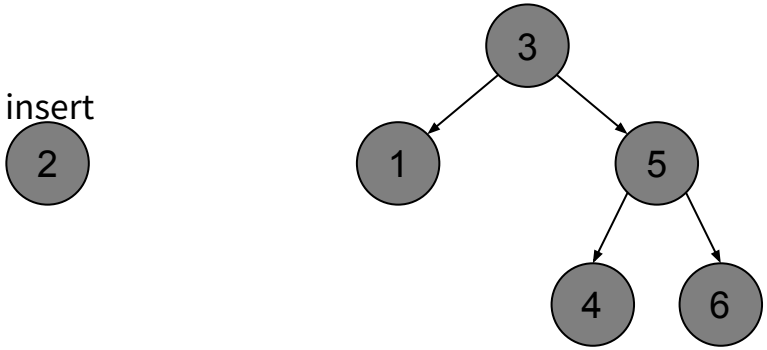
Trees - Backward Traversal Order (Right, current, left)

When traversing a list in backward order, the nodes will inherently be reverse sorted. Why?

- Backward always traverses the right child first, which is always greater than the left child
- The right child and its children will be traversed completely before the left child



Trees - Insertion



When inserting into a tree:

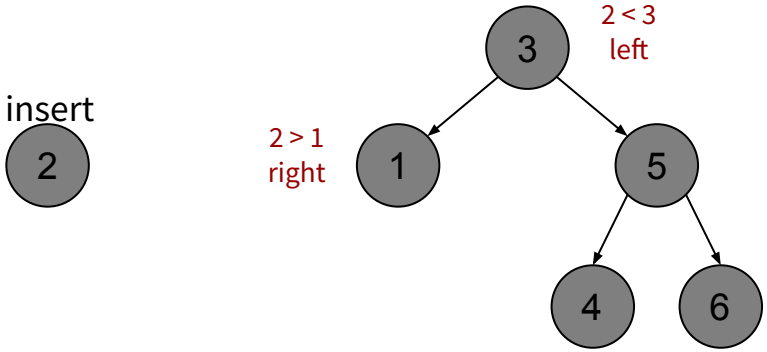
Start at the root node

If the new node is less than the root, take the left path

If greater than or equal, take the right

When the current node has no children, append to either the left or the right, respectively

Trees - Insertion



When inserting into a tree:

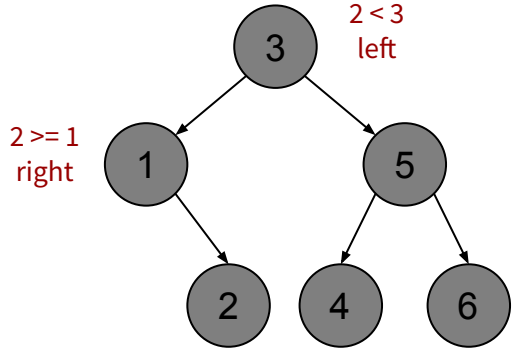
Start at the root node

If the new node is less than the root, take the left path

If greater than or equal, take the right

When the current node has no children, append to either the left or the right, respectively

Trees - Insertion



When inserting into a tree:

Start at the root node

If the new node is less than the root, take the left path

If greater than or equal, take the right

When the current node has no children, append to either the left or the right, respectively