Slides by Andrew DeOrio
and James Juett

EECS 280

Programming and Introductory Data Structures

Recursion, Function Pointers and Testing

Review: recursion

- Tail recursion and iteration are the same
  - We proved this last time with a *constructive* proof
  - Should you ever use tail recursion?

    Pro: Compiler handles update dependencies

    Pro: Sometimes tail recursion (base/rec. cases) is more intuitive

    Con: Sometimes iteration is more intuitive

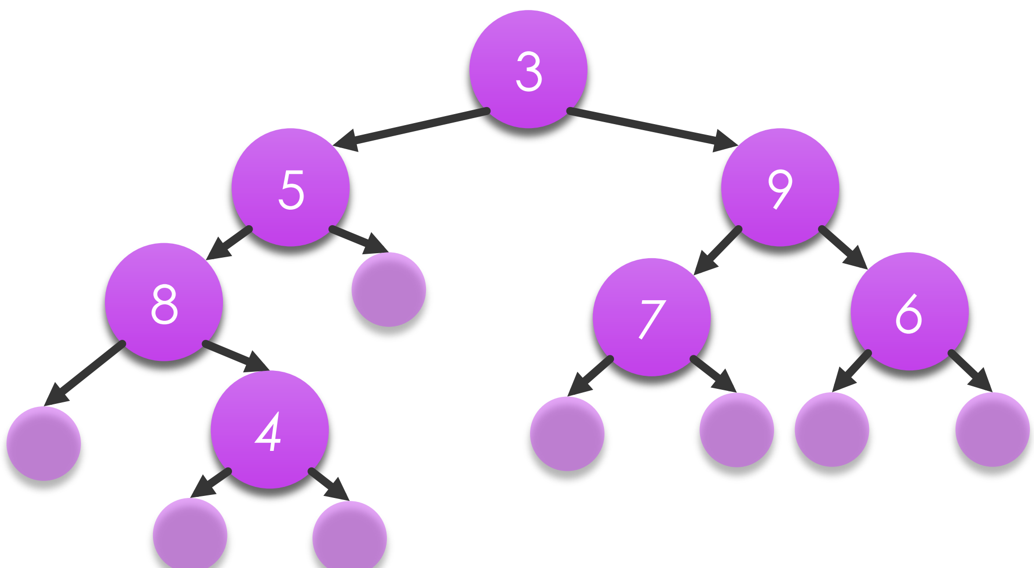
    Con: Can be tricky to ensure tail call optimization

- "General" recursion is expensive
  - Stack frames sit and store pending work
  - Why would you ever use this??
  - Sometimes you need it

Review: recursive data structures

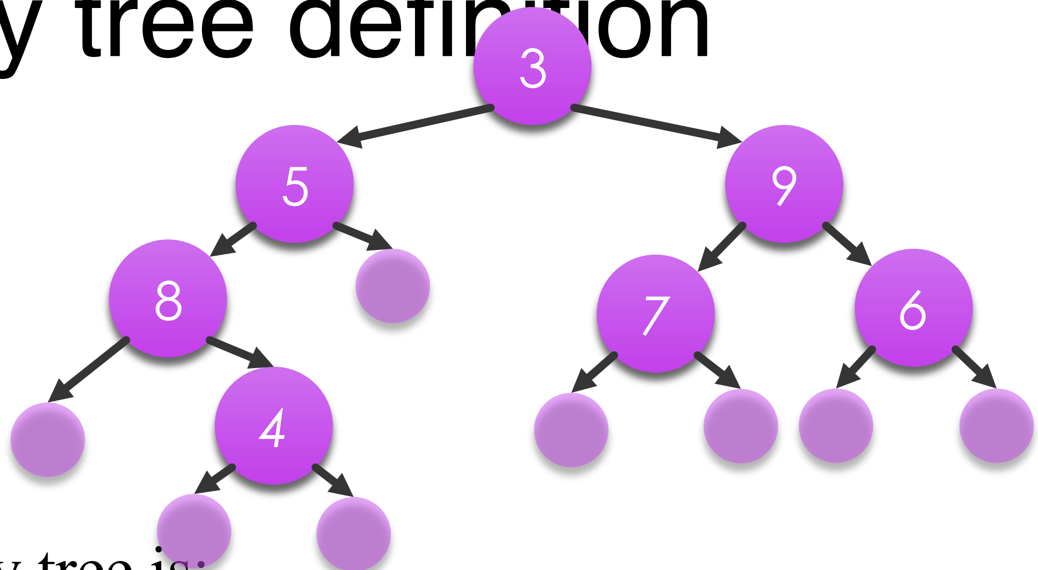
- List (last time)

- Tree (today)



Binary tree definition

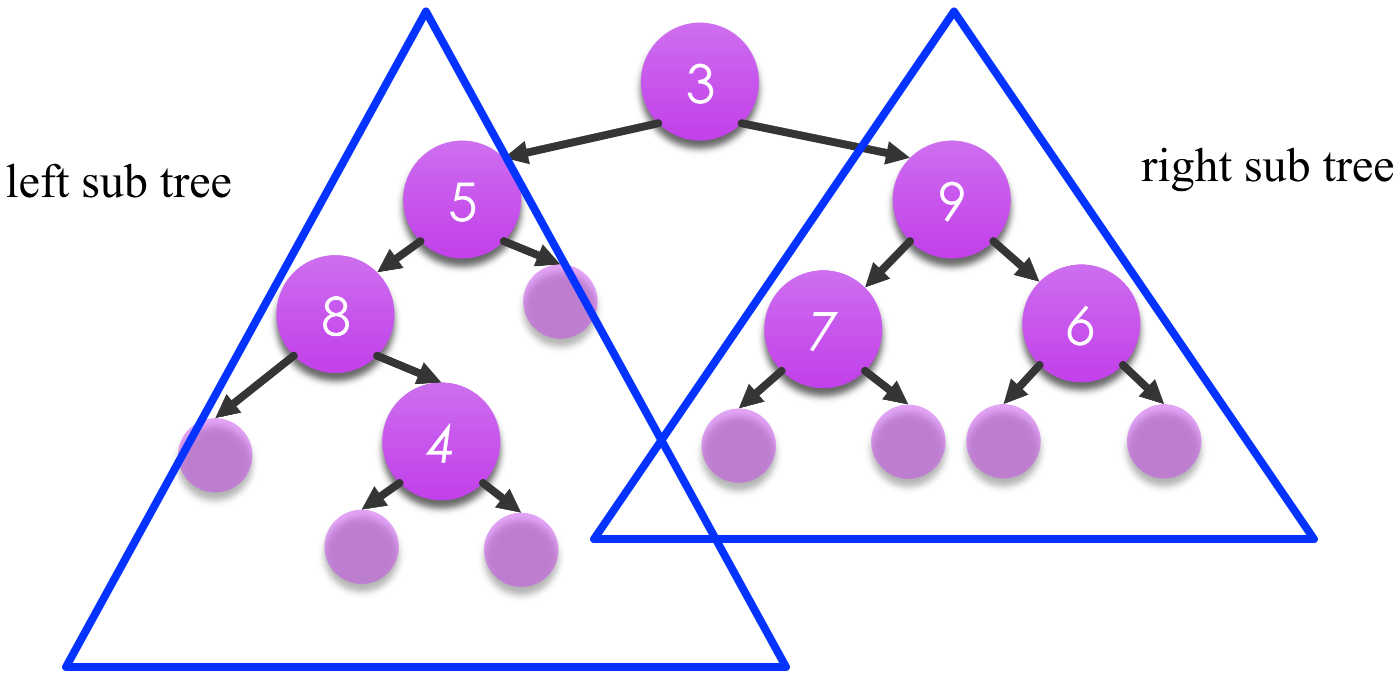
- Tree



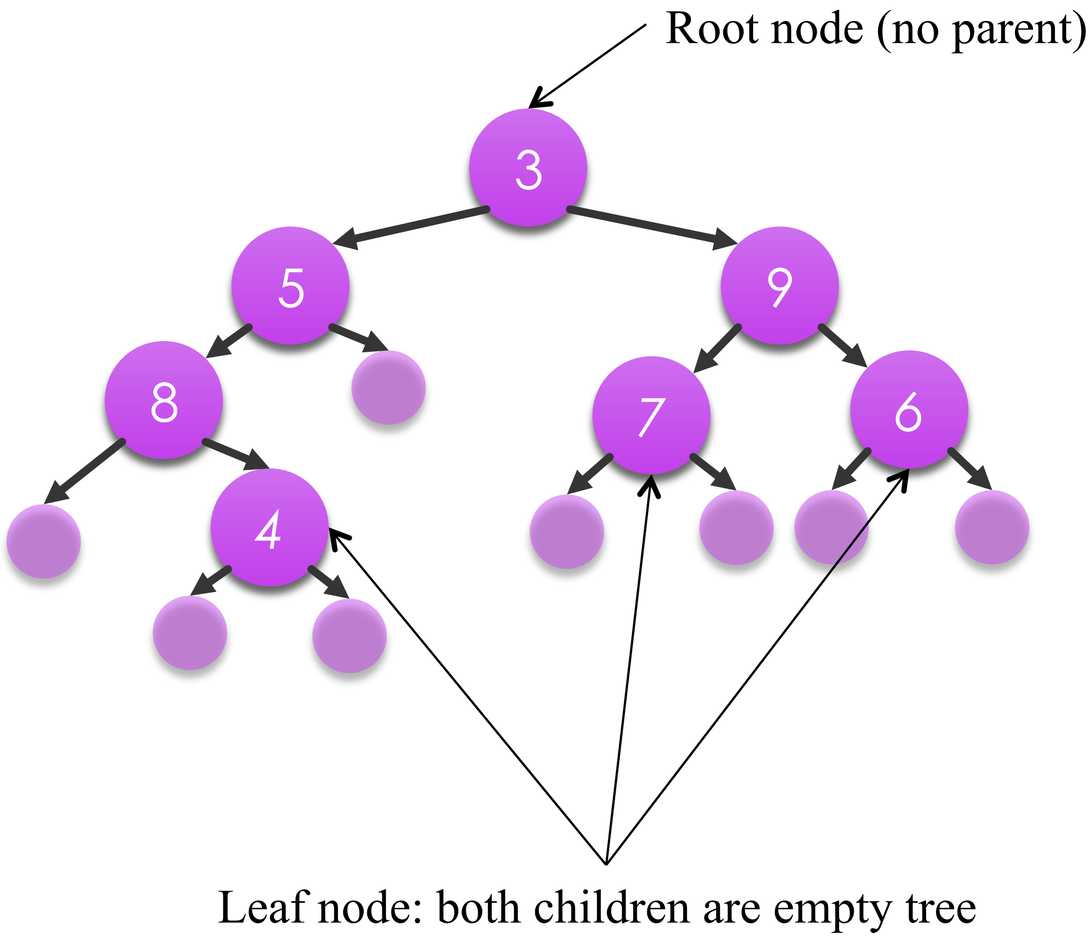
- A binary tree is:
  - The empty tree, or
  - An integer element, plus two children, called the left subtree and the right subtree, both binary trees

- Recursively defined data structure

Tree terminology

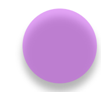


Tree terminology

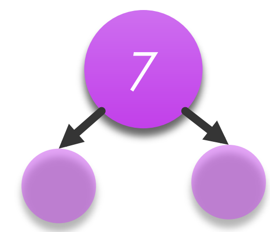


Tree type provided in project 2

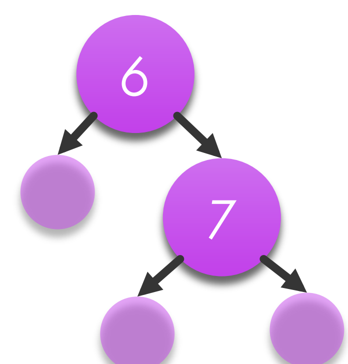
- In addition to `list_t`, the type `tree_t` is provided in project 2
- `tree_t empty = tree_make();`



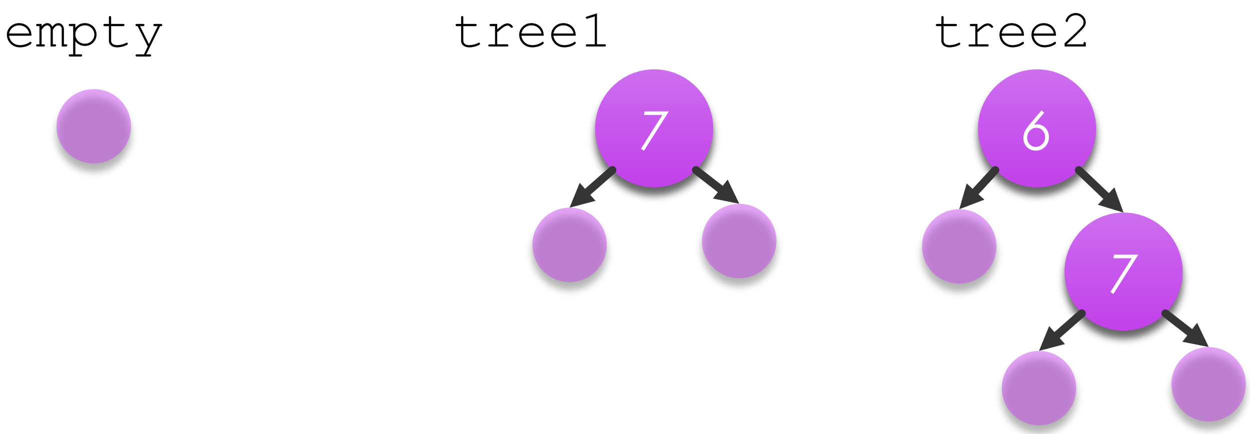
```
tree_t tree1 = tree_make(7,
                         empty,
                         empty);
```




```
tree_t tree2 = tree_make(6,
                         emphty,
                         tree1);
```



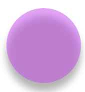
Tree type provided in project 2



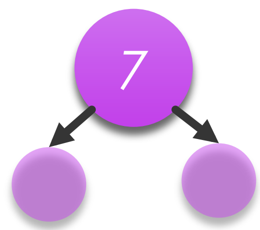

```
tree_isEmpty(empty) //true
tree_isEmpty(tree1) //false
tree_isEmpty(tree2) //false
```

Tree type provided in project 2

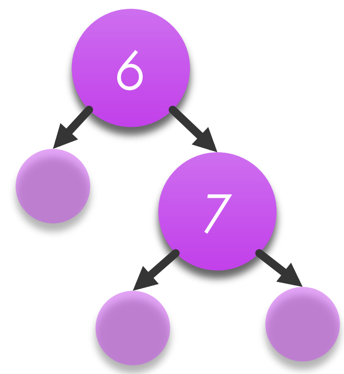
`empty`



`tree1`



`tree2`

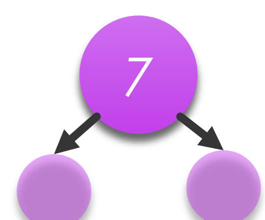



```
tree_elt(tree1); // 7

tree_elt(empty); // BAD!
```

`tree_left(tree2);`

`tree_right(tree2);`



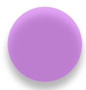
Tree properties

- Trees are *immutable*
  - There is no `tree_setElement`

- Think of lists as values
  - To “modify” a tree, you just have to “compute” a new one with desired changes
  - This means putting the three parts of the tree back together using `tree_make()...`
    - element, left subtree, right subtree

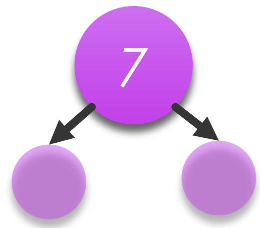
Exercise: tree_depth()

empty



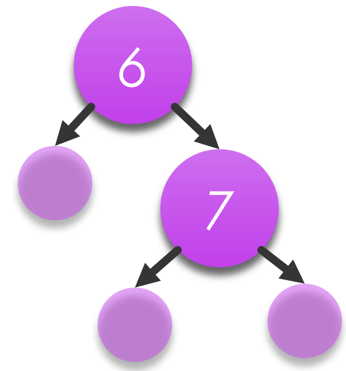
tree_depth(empty) //0

tree1



tree_depth(tree1) //1

tree2



tree_depth(tree2) //2

```
//EFFECTS:  Returns the depth of tree
int tree_depth (tree_t tree){

    //use general recursion
    //you can use the max(a, b) function from
    //last time
}
```

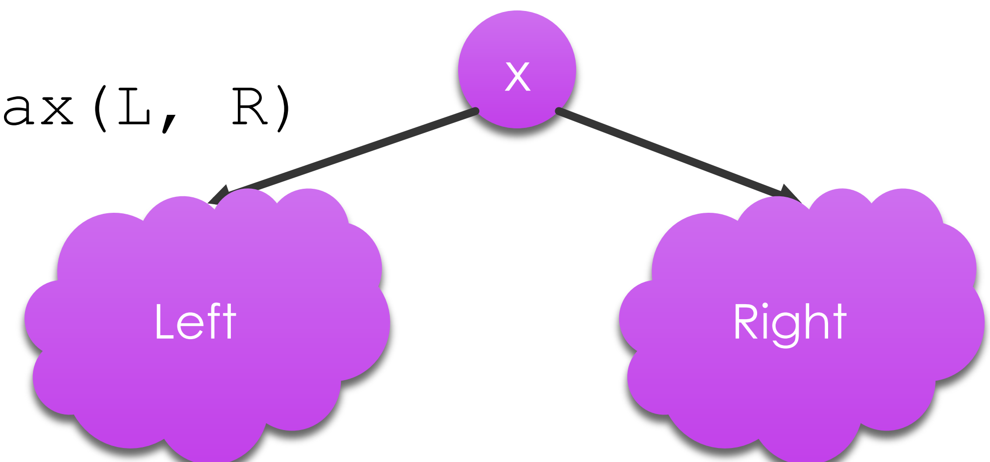
Hint

- Base case

```
if (tree_isEmpty(tree)) {
  return 0;
}
```

- Recursive case

```
depth = 1 + max(L, R)
```




```
L = depth(tree_left(tree))    R = depth(tree_right(tree))
```

Solution: tree_depth()

```
int max(int x, int y){ return x > y ? x : y; }

//EFFECTS:  Returns the depth of tree.
int tree_depth (tree_t tree){

  if (tree_isEmpty(tree)) return 0; // BASE CASE

  // RECURSIVE CASE
  int L = tree_depth(tree_left(tree));
  int R = tree_depth(tree_right(tree));
  return 1 + max(L, R);
}
```

Exercise: tree_depth() tail

```
//EFFECTS:  Returns the depth of tree
int tree_depth (tree_t tree){


  // Use tail recursion OR iteration


}
```

Solution: tree_depth() tail

```
//EFFECTS:  Returns the depth of tree
int tree_depth (tree_t tree){

  // Use tail recursion OR iteration


}
```

- You can't do it!!!!

Function pointers

- On to function pointers

Recall tail recursive list_max()

```
int max(int x, int y){ return x > y ? x : y; }

int list_max_h(list_t list, int so_far){
  if (list_isEmpty(list)) return so_far;
  return list_max_h(
    list_rest(list),
    max(list_first(list), so_far)
  );
}

int list_max (list_t list){
  assert(!list_isEmpty(list);
  return list_max_h(list, list_first(list));
}
```

Exercise: how would you change this code to implement list_min()?

Solution: list_min()

```
int max(int x, int y){ return x > y ? x : y; }
int min(int x, int y){ return x < y ? x : y; }
int list_min_h(list_t list, int so_far){
  if (list_isEmpty(list)) return so_far;
  return list_min_h(
    list_rest(list),
max min(list_first(list), so_far)
  );
}

int list_min (list_t list){
  assert(!list_isEmpty(list);
  return list_min_h(list, list_first(list));
}
```

Rewriting list_min() and list_max()

- We've copy pasted our code for `list_max()` to write `list_min()`
- Let's make one new function that can do the work of both

```
int list_extreme(list_t list, int so_far);
```

Rewriting list_min() and list_max()

- Now we need to pass the behavior of min and max as an input to `list_extreme`

```
int max(int x, int y){ return x > y ? x : y; }
int min(int x, int y){ return x < y ? x : y; }
int list_extreme(list_t list, int so_far, /* min or max */) {
  //...
}
```

- How can we make a function behave like a variable so we can make it an input to `list_extreme()`?

Function Pointers

- Function pointers are variables
- Function pointers are like a second name for a function
- Function pointers let us decide between two (or more) functions at run time

```
int max(int x, int y){ return x > y ? x : y; }
int min(int x, int y){ return x < y ? x : y; }
int main() {
  int (*comp)(int,int);
  comp = max;
  cout << comp(-42, 42); //42
}
```

Function Pointers

```
int max(int x, int y){ return x > y ? x : y; }
int min(int x, int y){ return x < y ? x : y; }
int main() {
  int (*comp)(int,int);
  comp = max;
  cout << comp(-42, 42); //42
}
```

- What does this mean?

```
    comp            //comp is a variable
   (*comp)(       )//the type is function pointer
int (*comp)(       )//... to fxn that returns an int
int (*comp)(int,int)//... and takes 2 ints as input
```

Function Pointers

```
int max(int x, int y){ return x > y ? x : y; }
int min(int x, int y){ return x < y ? x : y; }
int main() {
  int (*comp)(int,int);
  comp = max;
  cout << comp(-42, 42); //42
}
```

- We can assign a value to our new variable
- The value is the name of a function
- The instructions for executing a function are stored somewhere…a function pointer actually stores the **address** where the function is stored

Function Pointers

```
int max(int x, int y){ return x > y ? x : y; }
int min(int x, int y){ return x < y ? x : y; }
int main() {
  int (*comp)(int,int);
  comp = max;
  cout << comp(-42, 42); //42
}
```

- Then, we can use our `comp` variable just like a function
- It will do the same thing as `max` in this example

Rewriting list_min() and list_max()

```
int max(int x, int y){ return x > y ? x : y; }
int min(int x, int y){ return x < y ? x : y; }
int list_extreme(list_t list, int so_far, int (*comp)(int,int)) {
  //...
}
```

- Now we'll add an input to list_extreme using our new type

Rewriting list_min() and list_max()

```
int max(int x, int y){ return x > y ? x : y; }
int min(int x, int y){ return x < y ? x : y; }
int list_extreme(list_t list, int so_far, int (*comp)(int,int)) {
  if (list_isEmpty(list)) return so_far;
  return list_extreme(
    list_rest(list),
    comp(list_first(list), so_far)
    );}
```

- And use `comp` instead of `min` or `max`

Rewriting list_min() and list_max()

```
int max(int x, int y){ return x > y ? x : y; }
int min(int x, int y){ return x < y ? x : y; }
int list_extreme(list_t list, int so_far, int (*comp)(int,int)) {
  if (list_isEmpty(list)) return so_far;
  return list_extreme(
    list_rest(list),
    comp(list_first(list), so_far)
    );
}
```

- Finally, we can write `list_min()` and `list_max()`

```
int list_max(list_t list){ return list_extreme(list, 1, max); }
int list_min(list_t list){ return list_extreme(list, 1, min); }
```

Why function pointers?

- Avoid code duplication
- We can use a simple function, like `min` or `max` to change the behavior of a more complicated function, like `list_extreme`

Exercise

```
bool all_of(list_t list, bool(*predicate)(int)) {
  //EFFECTS: returns true if predicate returns true
  //  for all elements in list

  if (list_isEmpty(list)) //base case
    return true;
  if (!predicate(list_first(list))) //check first
    return false;
  return all_of(list_rest(list), fn); //rec. case
}
```

- Write these two functions. Use `all_of()` and helper functions

```
bool all_even(list_t list);
bool all_odd(list_t list);
```

Solution

```
bool is_even(int i) {
  //EFFECTS: returns true if i is even
  return (i % 2) == 0;
}

bool all_even(list_t list) {
  //EFFECTS: returns true if all elements are even
  return all_of(list, is_even);
}
```

Solution

```
bool is_odd(int i) {
  //EFFECTS: returns true if i is odd
  return (i % 2) == 1;
}

bool all_odd(list_t list) {
  //EFFECTS: returns true if all elements are odd
  return all_of(list, is_odd);
}
```

Testing

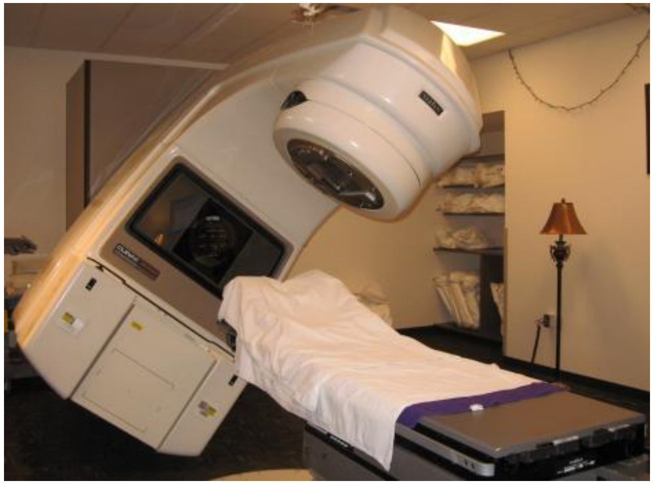
- The goals of testing:
- Ensure that code works as expected
- Meet the requirements of the spec

When Testing Goes Wrong

- Patriot missile bug
  - Persian Gulf war
  - Missile defense system to detect an incoming missile, then deploy a counter-missile to destroy it.
    - Needs to happen fast
  - Bug in floating point time conversion code
- System failed to intercept a missile
  - 28 killed, 98 injured

When Testing Goes Wrong

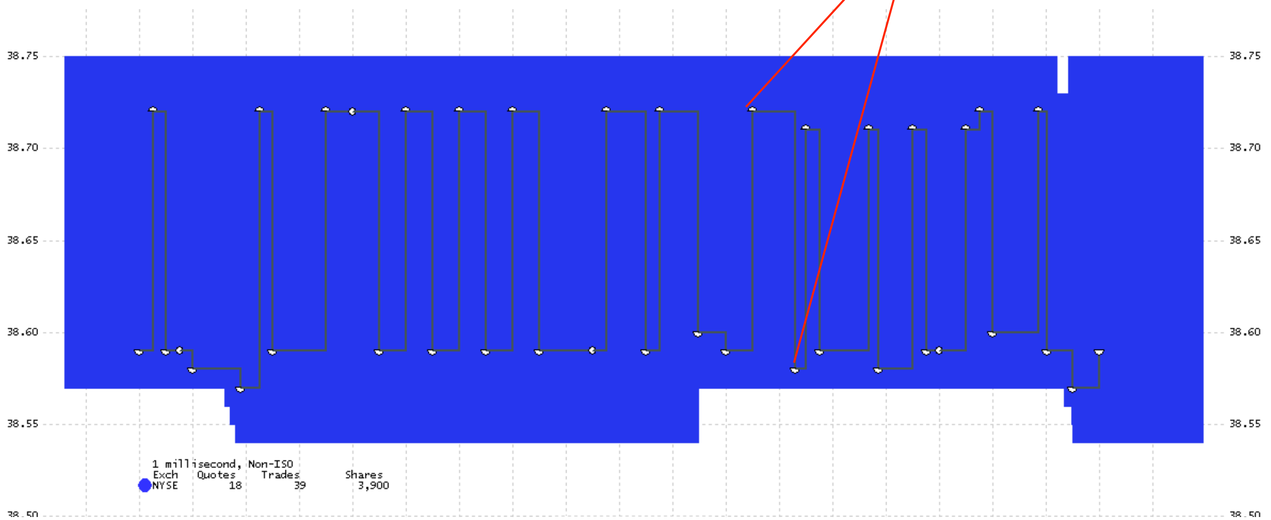
- Therac-25 Radiation Therapy Machine (1980’s)
- Radiation therapy
  - Low-power direct electron beam or megavolt X-ray
- Bug in software caused megavolt X-ray instead of low-power
  - Race condition
- Patients died of radiation poisoning



When Testing Goes Wrong

- Knight Capital Group: high-speed stock trading
  - Goal: buy low, sell high
- August 1, 2012: upgraded software algorithm
  - Bug: buy high, sell low!
- Cost $440M

trades: buy high, then sell low



1 ms

Motivation for testing

- Good testing yields correctly working software products
- Good testing yields good scores on projects
  - Typically, the difference between a good and bad score on a project doesn't have very much to do with your talent as a programmer. **It has much more to do with your talents as a tester!**

- Testing is not the same as debugging
  - Testing: Discovering that something is broken
  - Debugging: Fixing something once you know it's broken

Psychology of a Good Tester

- Adversarial frame of mind.
- You must be convinced that the code you are testing is broken and your task is to find out where.
- You can NEVER REST and must ALWAYS BE DILIGENT, because the code is NEVER FINISHED!
- Everyone makes mistakes, and one essential nature of a mistake is that the person who made it (i.e. YOU) didn't realize it was wrong in the first place – you thought it was perfect!

Types of testing

- Unit testing
  - One piece at a time (e.g., a function)
  - Find and fix bugs early! **Less work.**
    - Test smaller, less complex, easier to understand units.
    - You just wrote the code: easier to debug
- System testing
  - Entire project (code base)
  - Do this *after* unit testing
- Regression testing
  - Automatically run all unit and system tests after a code change

How to test

1. Understand the specification
   - Identify the required behaviors

2. Write tests
   - Code to run your other code
   - Try to test only one behavior per test

3. Check the results
   - Know the answers in advance

Debugging Hint for Recursion

- Add a `cout` statement to the top of your recursive function
  - Watch the recursion in action
  - Very helpful for trees!

```
int countdown4(int i, int n){
  cout << "countdown4("
       << i << ","
       << n << ")" << endl;
  if (i > n) return 0;
  countdown4(i+1, n);
  cout << i << endl;
  return i;
}
```

```
$ ./a.out
countdown4(1,3)
countdown4(2,3)
countdown4(3,3)
countdown4(4,3)
3
2
1
```

Kinds of test cases

Imagine we are writing test cases for the chop function from project 2.[1]

```
// REQUIRES n >= 0 and list has at least n elements
// EFFECTS: returns the list equal to list without its last n elements
list_t chop(list_t list, int n);
```

Don't write these

Type Prohibited	`("cat", 2)`	`(1, 5)`	`(3, [1,2])`	`(true, 0)`
REQUIRES Prohibited	`([], 1)`	`([42], -1)`	`([1,2,3,4,5], 4)`	
Simple	`([1,2,3],2)`		`([1,2,3,4,5],1)`	
(Edge) Special	`([], 0)`	`([42], 0)`	`([42], 1)`	`([1,2], 2)`
Nonsense	Sorry, no good examples for `chop`.			
Stress	`([1,2,3,4,5,6,7,8,9,…,100000], 50000)`			

“Guard against Murphy, not Machiavelli! 1

- Do: Try to write test cases to catch bugs that people would realistically make
  - ([1,2,3,4,5],5)
  - “Tricky because it chops everything. Could expose a bug.”

- Don’t: Try to write test cases to catch bugs introduced by a devious coder
  - ([4,4,4,4,4],3)
  - “Tricky because maybe it crashes when the list has all fours.”

- Think about what makes test cases meaningfully different.
  - chop doesn’t use element values, so these aren’t meaningfully different.
- ([V,W,X,Y,Z],5) vs. ([A,B,C,D,E],5)

1 Attributed to Damian Conway

> "Thorough testing with "small" test cases is sufficient to find all bugs within a system."

The Small Scope Hypothesis

- Think about what makes two test cases meaningfully different for the function's behavior
- Beyond a small size, just making test cases bigger doesn't make them meaningfully different
  - ([1,2,3,4,5],3) vs. ([1,2,3,4,5,6],3)