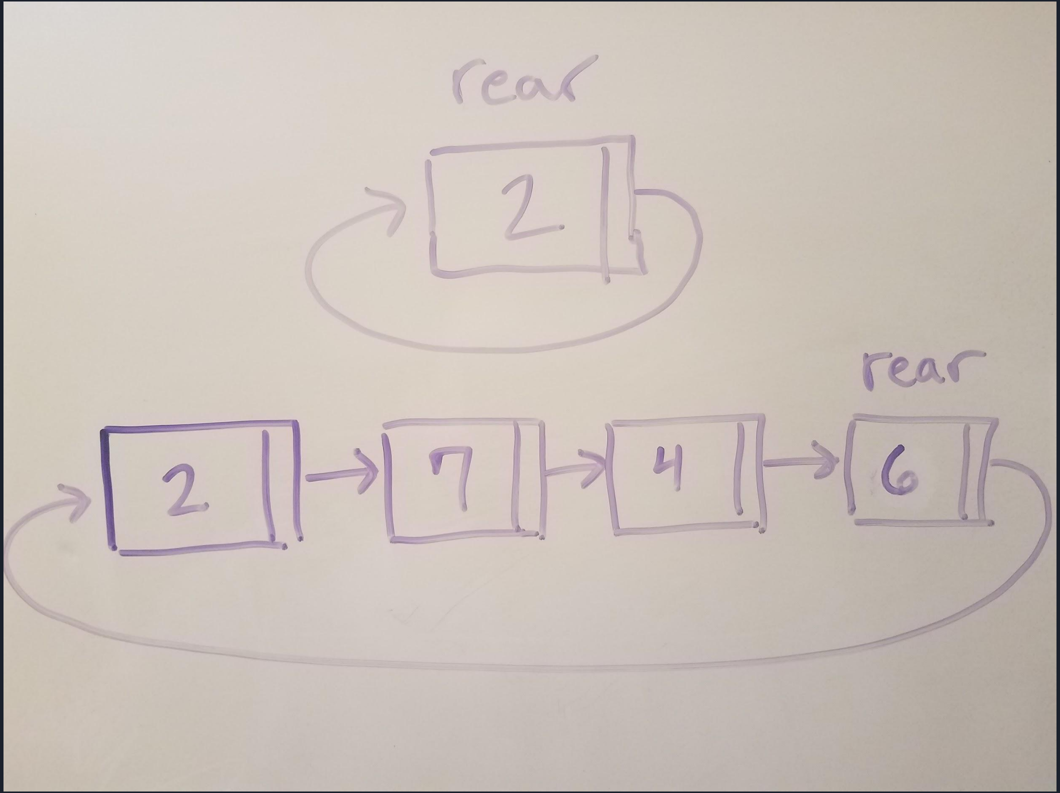
CS163 Lab 4

CLL, DLL,
Midterm Demo Practice

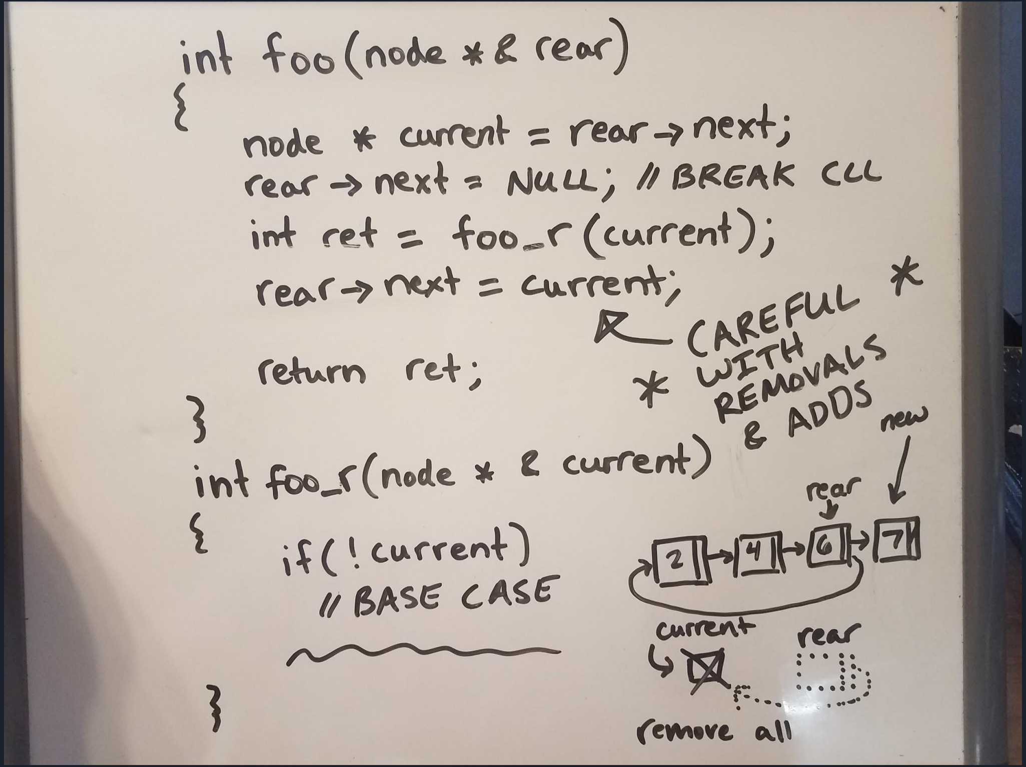
Matthew R. Hubbard

Circular Linked List(CLL)



CLL Techniques

Break the CLL into LLL



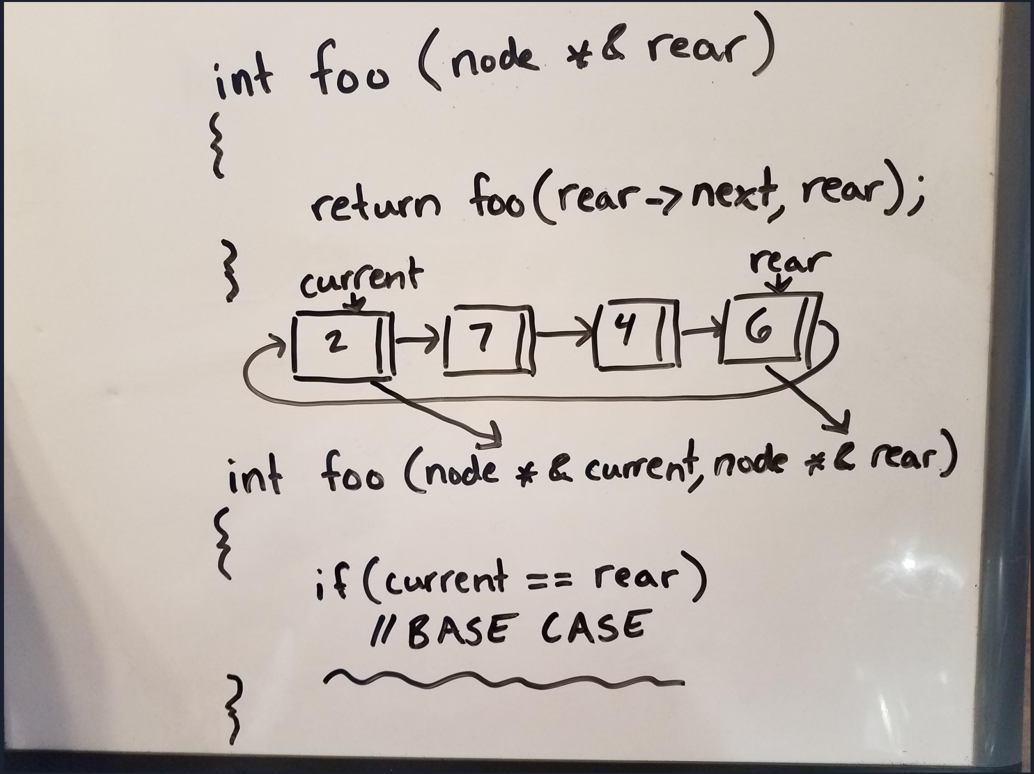
```
int foo(node *& rear)
{
    node * current = rear->next;
    rear->next = NULL; // BREAK CLL
    int ret = foo_r(current);
    rear->next = current;

    return ret;
}
int foo_r(node * & current)
{
      if(!current)
        // BASE CASE

      ~~~~~~~~~~~~~~~~~~~~
  }
```

* CAREFUL WITH REMOVALS & ADDS *

new
rear
2
4
6
7
current
rear
remove all

CLL Techniques

Traverse with current, hold rear



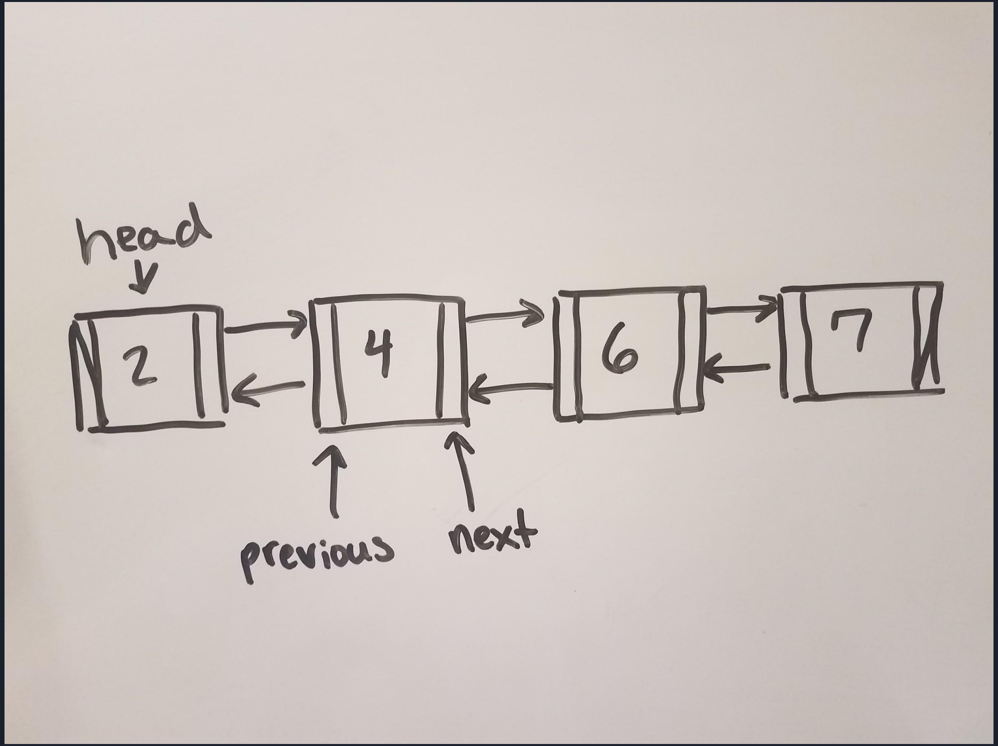
```
int foo (node *& rear)
{
    return foo(rear->next, rear);
}
```

```
int foo (node *& current, node *& rear)
{
    if (current == rear)
      // BASE CASE
    ~~~~~~~~~~~~~~~
}
```

Circular Linked List(CLL) Pitfalls

- Check your special cases!!!
  - Empty list
  - One node (rear->next = rear) *Rear->next could be itself. Does this break your code?
  - Add at rear (update rear)
  - Remove at front (make sure rear->next isn't pointing at the old deleted front)
    - *you might not have to make a special case for this with recursion by reference but keep it in mind
- Relink list if you break it! *(Be careful with removals and insertions)
- Do you need to do work in the base case?

Doubly Linked List(DLL)



DLL Pitfalls

- Cases
  - Empty list
  - Head changes (ex. Remove or add at head, update head, set head->previous to NULL)
  - Last node changes (ex. Remove or add at last node, make sure next pointer is NULL)
  - One node
  - Many nodes
  - Have to do work on all nodes (2, 2, 2, 2)
- Make sure to update your previous pointers!
  - If the list isn't displaying backwards correctly, something is wrong with your previous pointers!
- Head->previous = NULL
  - Terminate the list on both ends

General Pitfalls

- Watch your control flow!
  - Do you call the recursive call correctly in each call?
- Watch your returns! No code will execute after a return!
- Use your return values! Are you catching each call and using it?
- Be careful with pass by value vs. pass by reference.
  - In general, pass by value if you're not modifying the list (or if head will never be changed), pass by reference if you are modifying the list (head could change), or pass by reference if you need to update the value outside of the function call (or subsequent calls)
- Be careful with re-assigning head when pass by reference! Remember we ARE the pointer we passed in. (head = head->next will actually change the previous head passed in)
- Don't dereference a NULL pointer or deleted memory!
- Remember to use the new keyword only when you need to make a new node!

Recursion Rules

1. Base case
2. Move towards base case
3. Call yourself recursively

Recursion

Thinking about the problem (before coding)

- What part is repetitive in nature? (Recursion)
- What’s not repetitive? (Can we do this in wrapper?)
- What’s the base case? (When should we stop?)
- What are the special cases? (Empty, 1 node..etc)
- What should I be returning to the client?
- Do I understand what the question is asking?
- Try diagrams and pseudocode!

Recursion

Thinking about the problem

- Walk through code line by line while updating a pointer diagram
- How am I handling what’s being returned on each function invocation?
- Are there paths through my algorithm that don’t return?
- Remember, no code will execute after a return statement

Pitfalls

- No code executes after a return statement
- If you change arguments, change them for all calls including the recursive ones
- Pass by value but using variable as if it was passed by reference
- Only use the new keyword if you actually want to make a new node.
- if(head = NULL)
- Not catching return values
- Not calling the correct function in main

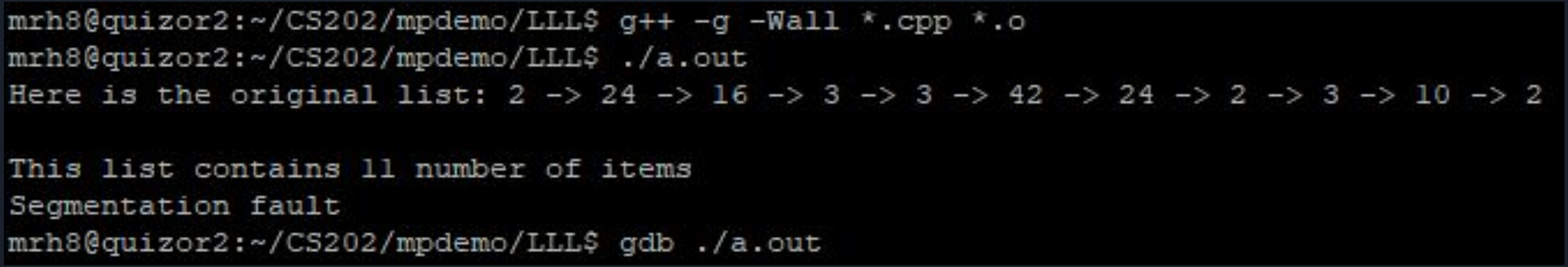
Demo Steps to Success

- Choose your problem wisely (take the first one that you think you can do (that doesn't seem ridiculously hard)).
- Take time to understand what the problem is asking and ask questions if it's unclear.
- Take a deep breath and try to relax.
- Think about the algorithm you would use to solve it:
- What's recursive? What's iterative? What should happen in the wrapper?
- What should be returned?
- Can you figure out how to solve it with different arguments?
- Would doing work after the recursive call be helpful?
- What could go wrong?
- What are the special cases?
- Draw diagrams and write pseudocode
- Start coding (make a file and #include .h file)
- **Compile often!** Solve syntax issues as they come up.

Demo Steps to Success

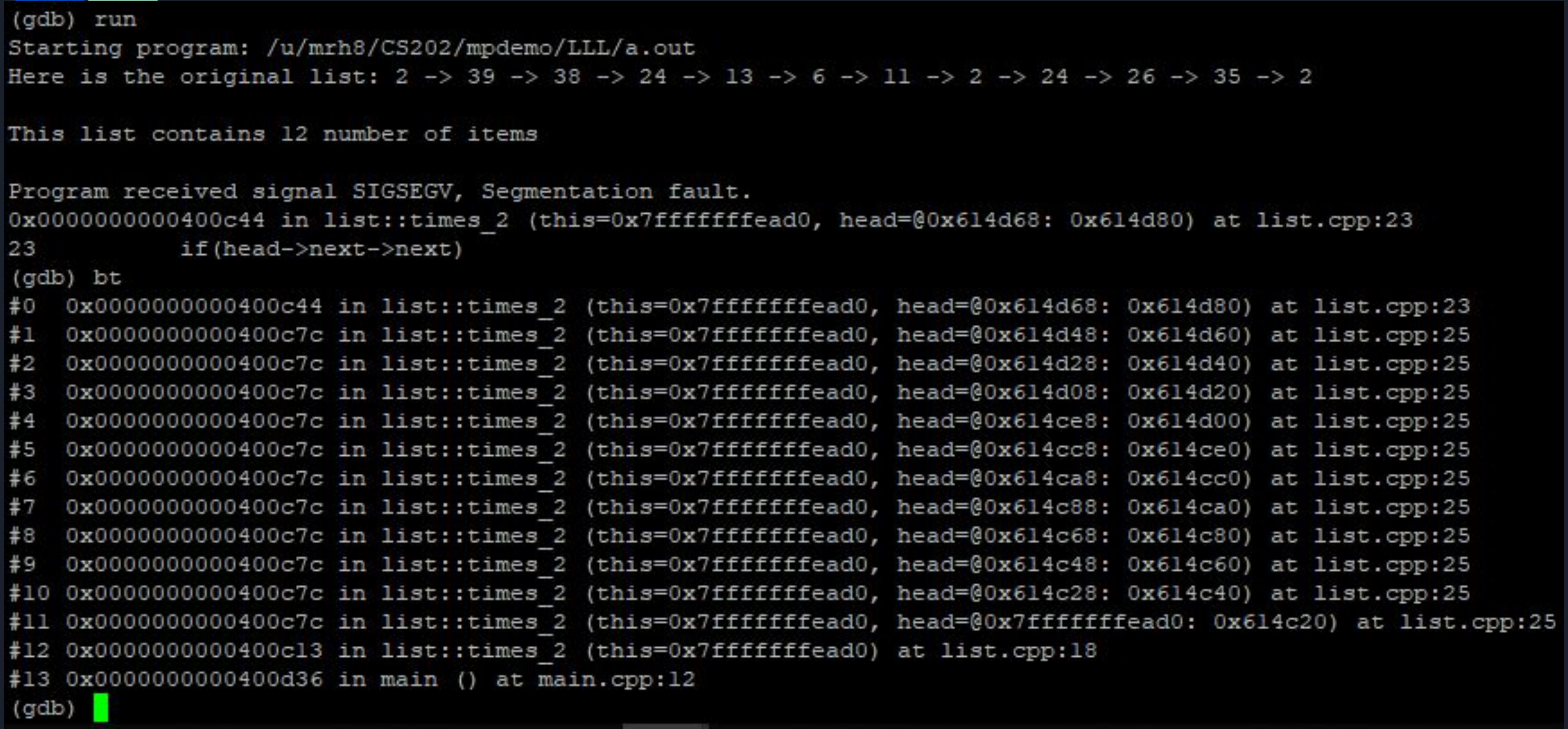
- What happens if it seg faults?:
  - Is there anywhere where you could be deleting or accessing a NULL node or a node that's not NULL but is pointing to deleted memory?
  - Try running in gdb and running a backtrace. What line is it seg faulting at and why?
- What happens if it's not doing what I expect?:
  - Go line by line updating a pointer diagram
  - Don't just draw what you *think* is happening but what your code is *actually* doing!
  - Check your special cases (What happens at the end of the list or what happens if there's a run of 2's, etc..)
- Okay, I think it's working… now what?:
  - Test, test, test! Make sure it works with a couple different lists.
  - The list won't ever be empty or only have one item or no 2's. Does your code handle special cases that aren't covered?

gdb backtracking



```
mrh8@quizor2:~/CS202/mpdemo/LLL$ g++ -g -Wall *.cpp *.o
mrh8@quizor2:~/CS202/mpdemo/LLL$ ./a.out
Here is the original list: 2 -> 24 -> 16 -> 3 -> 3 -> 42 -> 24 -> 2 -> 3 -> 10 -> 2

This list contains 11 number of items
Segmentation fault
mrh8@quizor2:~/CS202/mpdemo/LLL$ gdb ./a.out
```

gdb backtracking



```
(gdb) run
Starting program: /u/mrh8/CS202/mpdemo/LLL/a.out
Here is the original list: 2 -> 39 -> 38 -> 24 -> 13 -> 6 -> 11 -> 2 -> 24 -> 26 -> 35 -> 2

This list contains 12 number of items

Program received signal SIGSEGV, Segmentation fault.
0x0000000000400c44 in list::times_2 (this=0x7fffffffead0, head=@0x614d68: 0x614d80) at list.cpp:23
23            if(head->next->next)
(gdb) bt
#0  0x0000000000400c44 in list::times_2 (this=0x7fffffffead0, head=@0x614d68: 0x614d80) at list.cpp:23
#1  0x0000000000400c7c in list::times_2 (this=0x7fffffffead0, head=@0x614d48: 0x614d60) at list.cpp:25
#2  0x0000000000400c7c in list::times_2 (this=0x7fffffffead0, head=@0x614d28: 0x614d40) at list.cpp:25
#3  0x0000000000400c7c in list::times_2 (this=0x7fffffffead0, head=@0x614d08: 0x614d20) at list.cpp:25
#4  0x0000000000400c7c in list::times_2 (this=0x7fffffffead0, head=@0x614ce8: 0x614d00) at list.cpp:25
#5  0x0000000000400c7c in list::times_2 (this=0x7fffffffead0, head=@0x614cc8: 0x614ce0) at list.cpp:25
#6  0x0000000000400c7c in list::times_2 (this=0x7fffffffead0, head=@0x614ca8: 0x614cc0) at list.cpp:25
#7  0x0000000000400c7c in list::times_2 (this=0x7fffffffead0, head=@0x614c88: 0x614ca0) at list.cpp:25
#8  0x0000000000400c7c in list::times_2 (this=0x7fffffffead0, head=@0x614c68: 0x614c80) at list.cpp:25
#9  0x0000000000400c7c in list::times_2 (this=0x7fffffffead0, head=@0x614c48: 0x614c60) at list.cpp:25
#10 0x0000000000400c7c in list::times_2 (this=0x7fffffffead0, head=@0x614c28: 0x614c40) at list.cpp:25
#11 0x0000000000400c7c in list::times_2 (this=0x7fffffffead0, head=@0x7fffffffead0: 0x614c20) at list.cpp:25
#12 0x0000000000400c13 in list::times_2 (this=0x7fffffffead0) at list.cpp:18
#13 0x0000000000400d36 in main () at main.cpp:12
(gdb)
```