ECE 590.01
C++ Programming, Data structures, and Algorithms

Recursion
Admin

- Hwk 3
  - Out now
  - Start early

- Project Proposals
  - Due Friday

- Reading
  - Chapter 4

- OH tomorrow
Remind us where we left off last time?

- Who can remind us what we talked about?
Remind us where we left off last time?

- Who can remind us what we talked about?
  - Binary Search
    - On sorted arrays
  - Binary Search Trees
    - Pointer based data structure
    - Recursively defined
Today Recursion

- Today’s focus: recursion
  - Some people think it’s scary—but it’s not
  - Wonderful tool
    - Makes many tasks easier
    - Some languages **only** have recursion (no loops)

- Basic idea:
  - Use function you are writing to solve smaller instance of same problem
    - After all, the function you are writing solves that problem!
Structural Recursion

• We have seen two recursively defined data types:
  • Linked Lists
  • Binary Search Trees

• Can think of recursion here in terms of structural recursion
  • Recursively defined data structure
  • Leads to recursive algorithm
    • Examine structural cases
    • Use recursion to attack smaller instances of the same problem
Structural Recursion

```c
struct ll_node {
    int data;
    struct ll_node * next;
};
```

Structurally, an `ll_node *` is either
- `NULL`
orpointing at `(data + next)`
  - `next` (being an `ll_node *`) is a (smaller) list
    -> thus suitable for structural recursion
Structural Recursion

```c
struct ll_node {
    int data;
    struct ll_node *next;
}
```

Consider reversing a list...
Structural Recursion

Want to reverse this list...
Structural Recursion

Want to reverse this list...
What if we already had the rest of the list reversed?
Structural Recursion

Want to reverse this list...
What if we already had the rest of the list reversed?
Could just add 3 to the back
Reverse

ll_node * reverse(ll_node * lst) {
    if (lst == NULL) {
        
    } else {

    }
}

Two cases:
    NULL
Or
    pointing at a node
Reverse

ll_node * reverse(ll_node * lst) {
    if (lst == NULL) {

    }
    else {
        How to reverse empty list?
        Note: cannot recurse here (don’t have a recursive structure)

    }
}
Reverse

ll_node * reverse(ll_node * lst) {
    if (lst == NULL) {
        return NULL;
    }
    else {
        // How to reverse empty list?
        // Just the empty list
    }
}

ECE 590.01 (Hilton): Recursion
Reverse

ll_node * reverse(ll_node * lst) {
    if (lst == NULL) {
        return NULL;
    }
    else {
        ll_node * revList = reverse(lst->next);
    }
}

In the other case,
Said it would be nice to have
Reversal of rest of list
Reverse

ll_node * reverse(ll_node * lst) {
    if (lst == NULL) {
        return NULL;
    } else {
        ll_node * revList = reverse(lst->next);
        return add_to_back(lst->data, revList);
    }
}

Then add current data to back.
Reverse

ll_node * reverse(ll_node * lst) {
    if (lst == NULL) {
        return NULL;
    }
    else {
        ll_node * revList = reverse(lst->next);
        return add_to_back(lst->data, revList);
    }
}

Note: this does a functional update (builds a new list that is the reverse of the original, leaving the original unchanged).
Reverse

ll_node * reverse(ll_node * lst) {
    if (lst == NULL) {
        return NULL;
    }
    else {
        ll_node * revList = reverse(lst->next);
        return add_to_back(lst->data, revList);
    }
}

How efficient is this? (Big-O?)
We can do better

• That was $O(N^2)$
• We can reverse a list in $O(N)$ time.
  • We just have to build up our answer on the way down...
  • What we just saw is called “head recursion”
  • Another approach “tail recursion”
    • Return recursive result w/o doing any other work
Reverse: tail recursion

```c
ll_node * reverse(ll_node * lst,
                ll_node * ans) {

}
```

Now, build up answer on the way down
Reverse: tail recursion

```c
ll_node * reverse(ll_node * lst,
                   ll_node * ans) {
    if (lst == NULL) {
        return ans;
    } else {
        ll_node * temp = lst->next;
        lst->next = ans;
        ans = lst;
        lst = temp;
        return reverse(lst, ans);
    }
}
```

Still two cases
Reverse: tail recursion

```c
ll_node * reverse(ll_node * lst,
                  ll_node * ans) {
    if (lst == NULL) {

    }
    else {

    }
    
    But now, rev(NULL,x)
    is not just “reverse the empty list”
    Its “reverse the empty list where
    x represents the reversal of everything before”

    }
}
```
Reverse: tail recursion

```c
ll_node * reverse(ll_node * lst, ll_node * ans) {
    if (lst == NULL) {
        return ans;
    } else {
        So now there is nothing else to reverse,
        And we just return what we built up so far.
    }
}
```
Reverse: tail recursion

```c
ll_node * reverse(ll_node * lst,
                  ll_node * ans) {
    if (lst == NULL) {
        return ans;
    }
    else {
        return reverse(lst->next,
                        add_to_front(list->data,
                                     ans);
    }
}
```

**Recursive case:**
- reverse the rest of the list
- With a new answer of “what we have reversed so far” [which we build by adding to the front as we peel off the list]
Reverse: tail recursion

```c
ll_node * reverse(ll_node * lst,
                 ll_node * ans) {
    if (lst == NULL) {
        return ans;
    }
    else {
        return reverse(lst->next,
                       add_to_front(list->data,
                                     ans);
    }
}
```

This is now O(N)

Also, you can’t find this scary:
its just a loop
Reverse: tail recursion

ll_node * reverse(ll_node * lst, ll_node * ans) {
    if (lst == NULL) {
        return ans;
    }
    else {
        lst = lst ->next;
        ans = add_to_front(list->data, ans);
        b reverse;
    }
}

Compiler recognized tail recursion,
Turns recursive call into jump...
So it really looks exactly like a while loop
Tail recursion = while loops

• The above is not a coincidence:
  • Tail recursion is identical to a while loop (unless your compiler isn’t smart enough to do the right transform).

```
foo(a,b,c) {
    while (cond) {
        a = f(a,b,c);
        b = g(a,b,c);
        c = h(a,b,c);
    }
    return k(a,b,c);
}
```

```
foo(a,b,c) {
    if (!cond) {
        return k(a,b,c);
    }
    return foo(f(a,b,c), g(a,b,c), h(a,b,c));
}
```
Two sides of the same coin

- Tail recursion = while loop..
  - So why not just write while loop?....
    - Some languages don’t have loops
      - They don’t let you change something once you make it can only make new things
    - Different ways of thinking about our invariants
      - reverse(x,y) = reverse(x) appended to y
      - May help us understand correctness better
    - May make things much easier with mutually recursive functions
      - Very complex looping structures
- Head recursion: not just a loop
  - Could be done with loop + explicit stack
- But first: numeric recursion
Numeric recursion

• Can also do recursion on numbers

```c
int fact(int n) {
    if (n <= 0) {
        return 1;
    }
    else {
        return n * fact(n-1);
    }
}
```
Numeric = Structural

- Numeric Recursion = Structural Recursion
  - Just recursing on the (abstracted away) structure of numbers
  - Mathematically, natural numbers have a structure
    - Zero
    - Or Successor of Natural Number

- Could think of them as

```c
struct nat {
    struct nat * pred;
};
```

//would be inefficient to do normal math
Numeric Recursion

• Would not actually want to implement numbers this way
  • Quite inefficient (but doable)
  • Seems like a fun exercise though (hmmm...)

• Why talk about it?
  • Good mental model for recursion
  • Master structural recursion
  • Think of the “structure” of natural numbers
  • Need to deal with negative numbers?
    • Need a different structure
Mutual Recursion

• Let’s suppose we have a simple programming language:

• We have three types of l-values
  • An identifier (variable: x, y, z)
  • An l-value DOT an identifier (field access: x.y, z.q.a)
  • An l-value [ an expression ] (array access: a[3], a[b+c], a[y.z])

L-value = thing that can appear on left of assignment
Mutual Recursion

- Let us further suppose that we have a few types of expressions
  - Constants (4, 32, -7)
  - L-values (x, y.z, q[j.k][4].f)
  - Expression + Expression (42 + array[k])
  - Lvalue = Expression (x = y[32], z = q + 12)

- We now have a mutually recursive structure
  - L-values can reference expressions
  - Expressions can reference L-values
Mutual Recursion

• Mutually recursive structures lead to mutually recursive algorithms

```c
int evaluateExpr (expr * e, ...) {
    ...
    if (e->isLValueExpr()) {
        return evaluateLVal(e->lv);
    }
    ...
}

int evaluateLVal(lval * lv, ...) {
    ...
    if (lv->isArrayInd) {
        int ind = evaluateExpr(lv->indExp);
        ...
    }
    ...
```
Simple, but contrived example

• Even/odd: N is even if n-1 is odd and N is odd if N-1 is even

```cpp
bool isEven(int n) {
    if (n == 0) {
        return true;
    }
    return isOdd(n-1);
}

bool isOdd(int n) {
    if (n == 0) {
        return false;
    }
    return isEven(n-1);
}
```
Simple, but contrived example

- Only complexity: need prototype for forward reference

```cpp
bool isOdd(int n);
bool isEven(int n) {
    if (n == 0) {
        return true;
    }
    return isOdd(n-1);
}

bool isOdd(int n) {
    if (n == 0) {
        return false;
    }
    return isEven(n-1);
}
```
Recursion on Hwk3

- Directories are naturally recursive
  - Directories contain other directories
  - Twist: can have loops
    - Due to symlinks and hardlinks
- On hwk3: will need to process directories recursively
  - Look at file name
    - Regular file? Base case
    - Directory? Recursively call your function
Recursion on Hwk3

- Directories are naturally recursive
  - Directories contain other directories
  - Twist: can have loops
    - Due to symlinks and hardlinks

- On hwk3: will need to process directories recursively
  - Look at file name
    - Regular file? Base case
    - Directory? Recursively call your function

- Dealing with loops?
  - Keep a Set of what you have processed so far
    - Don’t re-process something if you already did it
Head Recursion

- Head recursion is not just a simple loop
  - Could replicate with explicit stack data structure
  - But why not just use call stack?

- Especially good for things like BSTs
  - Need to recurse left then recurse right
  - Stack keeps track of “all things to do”

- Example:
  - Sum elements of binary tree of ints
  - Let’s do in terminal