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

Abstract Data Types: Interface vs. Implementation

Ramp-up Reminder

- Ramp-up assignments: extra credit with a cap
  - Added straight to your final grade, as long as below the cap
  - Example: \(0.1 @ 80\) = 0.1 points, max of 80.
  - Final grade of \(X\) (before ramp-ups)?
  - Example:
    \[
    X = 82.5
    \]
    \[
    X' = X + \max(\min(R_{80}, 80 - X), 0)
    \]
    \[
    X'' = X' + \max(\min(R_{83}, 83 - X'), 0)
    \]
    \[
    X''' = X'' + \max(\min(R_{85}, 85 - X''), 0)
    \]
    \[
    X''' = X''' + \max(\min(R_{87}, 87 - X'''), 0)
    \]
    \[
    G = X''' + \max(\min(R_{90}, 90 - X'''), 0)
    \]
    \[
    \text{Note: 7 points of EC here is a LOT}
    \]

Remind us where we left off last time?

- Who can remind us what we talked about?
  - C
  - Man pages
  - Structures
  - Dynamic Allocation: malloc/free
    - Everyone get this?
  - Debugging: briefly

Now: Data Structures

- More complex problems require data structures
  - Need to store data and manipulate it in non-trivial ways
  - Two pieces to the above problem:
    - Interface: how does the rest of the algorithm access the DS?
      - What functions do we provide?
      - What arguments do they need to take?
      - What type of things get returned?
      - Are these functions pure? Or side-effecting?
    - Implementation:
      - How do we actually store the data?
      - Concerns: programmer effort, correctness, efficiency
ADT #1: Stack

- First ADT we will look at: the Stack
  - Last-in First-out Behavior
  - push: add a new item to the top of the stack (TOS)
  - peek: look at the item on TOS
  - pop: remove and return item on TOS
  - size: how many items in stack?

But what precisely is the interface?

- Previous slide: gist of the interface
  - "Gist of" is not good enough for programming...
  - What exactly does our interface look like?
    - Let's assume our stack holds void *'s right now
      - Stack of pointers to anything
    - Note: void * can be dangerous if abused
      - Nothing to stop us from pushing an int *, then treating as double * when popped
      - C++ gives us a slightly better option (next time)

Imperative Stack

```c
struct stack_t; //forward decl of type
stack_t * make_stack();
void push(stack_t * s, void * d);
void * peek(const stack_t * s);
void * pop(stack_t * s);
int size(const stack_t * s);
```

Functional Stack?

- Not only interface for a stack, could make a functional data structure
  - Functional data structure: never change old, make new
  - Interface written in SML: returning multiple values in C is ugly

```sml
type 'a stack
val empty: 'a stack
val push: 'a stack * 'a -> 'a stack
val peek: 'a stack -> 'a option
val pop: 'a stack -> 'a option * 'a stack
val size: 'a stack -> int
```

Functional Stack in C

- Could do it, either need pop() to just return new stack
  - Or, pass a pointer to where to write data:
```c
stack_t * pop(const stack_t * s, void ** dPtr);
```

Imperative Stack: Array impl

```c
struct stack_t {    void ** data;
    int tos;
    int max;
};
```
make_stack

```c
stack_t * make_stack() {
    stack_t * ans = malloc(sizeof(*ans));
    ans->max = 16; //some space to start?
    ans->data = malloc(ans->max * sizeof(ans->data));
    ans->tos = 0;
    return ans;
}
```

Allocate space for stack data structure

Allocate space for data

Initialize tos to indicate empty stack

Return answer

May need to realloc data if out of space

push

```c
void push (stack_t * s, void * d) {
    s->data[s->tos] = d;
    s->tos++;
}
```

Put data in tos slot, increment tos...

Unless...?
Stack: Array implementation (cont’d)

- Previous slide:
  \[ s\rightarrow_{\text{max}} = s\rightarrow_{\text{max}} \times 2; \]
  instead of
  \[ s\rightarrow_{\text{max}} = s\rightarrow_{\text{max}} + 1; \]
  - Efficiency: realloc may need to copy
  - Amortize cost of copying over many pushes
- Also, should be able to work through remaining functions
  (good practice)

Stack: Linked List Implementation

- Can implement same ADT in different ways
  - Code that uses it should be oblivious to implementation
  - Abstraction!
  - Could use a Linked List

\[
\text{struct } _{\text{ll}}\text{-node} \{
  \text{void } * \text{ data};
  \text{struct } _{\text{ll}}\text{-node} * \text{ next};
\};
\]

Linked Lists

- I’m going to assume most of you have seen LLs before
  - And know the basics of them in at least one language
  - Should be able to go implement LLs on your own in C
  - Can’t do that? Ramp-up
- Will take a moment to show you 3 ways to write “add to sorted list” and discuss them.

Way one: “look ahead one”

\[
\text{ll}_\text{-node} * \text{ add}_\text{-sorted}(\text{ll}_\text{-node} * \text{head}, \text{int \ data}) \{
  \text{ll}_\text{-node} * n = \text{malloc}()();
  n->\text{data} = \text{data};
  \text{if } (\text{head} == \text{NULL} || \text{data} < \text{head}->\text{data}) {\text{ Corner case if the new node goes first}}
  \text{return } n;
}\]

Way one: “look ahead one”

\[
\text{ll}_\text{-node} * \text{ add}_\text{-sorted}(\text{ll}_\text{-node} * \text{head}, \text{int \ data}) \{
  \text{ll}_\text{-node} * n = \text{malloc}()();
  n->\text{data} = \text{data};
  \text{if } (\text{head} == \text{NULL} || \text{data} < \text{head}->\text{data}) {\text{ Corner case if the new node goes first}}
  \text{return } n;
}\]

Way one: “look ahead one”

\[
\text{ll}_\text{-node} * \text{ add}_\text{-sorted}(\text{ll}_\text{-node} * \text{head}, \text{int \ data}) \{
  \text{ll}_\text{-node} * n = \text{malloc}()();
  n->\text{data} = \text{data};
  \text{if } (\text{head} == \text{NULL} || \text{data} < \text{head}->\text{data}) {\text{ Corner case if the new node goes first}}
  \text{return } n;
}\]

Way one: “look ahead one”

\[
\text{ll}_\text{-node} * \text{ add}_\text{-sorted}(\text{ll}_\text{-node} * \text{head}, \text{int \ data}) \{
  \text{ll}_\text{-node} * n = \text{malloc}()();
  n->\text{data} = \text{data};
  \text{if } (\text{head} == \text{NULL} || \text{data} < \text{head}->\text{data}) {\text{ Corner case if the new node goes first}}
  \text{return } n;
}\]

Way one: “look ahead one”

\[
\text{ll}_\text{-node} * \text{ add}_\text{-sorted}(\text{ll}_\text{-node} * \text{head}, \text{int \ data}) \{
  \text{ll}_\text{-node} * n = \text{malloc}()();
  n->\text{data} = \text{data};
  \text{if } (\text{head} == \text{NULL} || \text{data} < \text{head}->\text{data}) {\text{ Corner case if the new node goes first}}
  \text{return } n;
}\]

Way one: “look ahead one”

\[
\text{ll}_\text{-node} * \text{ add}_\text{-sorted}(\text{ll}_\text{-node} * \text{head}, \text{int \ data}) \{
  \text{ll}_\text{-node} * n = \text{malloc}()();
  n->\text{data} = \text{data};
  \text{if } (\text{head} == \text{NULL} || \text{data} < \text{head}->\text{data}) {\text{ Corner case if the new node goes first}}
  \text{return } n;
}\]

Way one: “look ahead one”

\[
\text{ll}_\text{-node} * \text{ add}_\text{-sorted}(\text{ll}_\text{-node} * \text{head}, \text{int \ data}) \{
  \text{ll}_\text{-node} * n = \text{malloc}()();
  n->\text{data} = \text{data};
  \text{if } (\text{head} == \text{NULL} || \text{data} < \text{head}->\text{data}) {\text{ Corner case if the new node goes first}}
  \text{return } n;
}\]

Way one: “look ahead one”

\[
\text{ll}_\text{-node} * \text{ add}_\text{-sorted}(\text{ll}_\text{-node} * \text{head}, \text{int \ data}) \{
  \text{ll}_\text{-node} * n = \text{malloc}()();
  n->\text{data} = \text{data};
  \text{if } (\text{head} == \text{NULL} || \text{data} < \text{head}->\text{data}) {\text{ Corner case if the new node goes first}}
  \text{return } n;
}\]

Way one: “look ahead one”

\[
\text{ll}_\text{-node} * \text{ add}_\text{-sorted}(\text{ll}_\text{-node} * \text{head}, \text{int \ data}) \{
  \text{ll}_\text{-node} * n = \text{malloc}()();
  n->\text{data} = \text{data};
  \text{if } (\text{head} == \text{NULL} || \text{data} < \text{head}->\text{data}) {\text{ Corner case if the new node goes first}}
  \text{return } n;
}\]

Way one: “look ahead one”

\[
\text{ll}_\text{-node} * \text{ add}_\text{-sorted}(\text{ll}_\text{-node} * \text{head}, \text{int \ data}) \{
  \text{ll}_\text{-node} * n = \text{malloc}()();
  n->\text{data} = \text{data};
  \text{if } (\text{head} == \text{NULL} || \text{data} < \text{head}->\text{data}) {\text{ Corner case if the new node goes first}}
  \text{return } n;
}\]

Way one: “look ahead one”

\[
\text{ll}_\text{-node} * \text{ add}_\text{-sorted}(\text{ll}_\text{-node} * \text{head}, \text{int \ data}) \{
  \text{ll}_\text{-node} * n = \text{malloc}()();
  n->\text{data} = \text{data};
  \text{if } (\text{head} == \text{NULL} || \text{data} < \text{head}->\text{data}) {\text{ Corner case if the new node goes first}}
  \text{return } n;
}\]
Way one: “look ahead one”

```c
ll_node * add_sorted(ll_node *head, int data) {
    ll_node * n = malloc(sizeof(*n));
    n->data = data;
    if (head == NULL || data < head->data) {
        n->next = head;
        return n;
    }
    ll_node * curr = head;
    while(curr->next != NULL && curr->next->data<data) {
        curr = curr->next;
    }
    n->next = curr->next;
    curr->next = n;
    return head;
}
```

Solution 1: examined

- Probably what you saw in prior courses if you saw this
- I hate it.
- Why?
  - Special cases, that aren’t really that special
  - Why should adding to the front be so different from anywhere else?
- What if we want to change things?
  - Sort the list some other way?
  - Changes in two places = bad

Pointer black-belt approach

- What is it we would really like here?
  - A way to point at “the thing we want to change”
  - Maybe its head
  - At the start

What type is this new arrow?

A pointer to (a pointer to a node)
Way two: “black belt in pointers”

```c
ll_node * add_sorted(ll_node *head, int data) {
    ll_node * n = malloc(sizeof(*n));
    n->data = data;
    llnode ** currp = &head;
    while (*currp != NULL && (*currp)->data < data) {
        currp = &(*currp)->next;
    }
    n->next = *currp;
    *currp = n;
    return head;
}
```

ECE 590.01 (Hilton): ADT
Starts the same make a new node and fill its data

The other arrow in the prev slide A pointer to a pointer to a node
It starts by pointing at the head variable
Note that *currp something changes head.

Way two: “black belt in pointers”

```
ll_node * add_sorted(ll_node *head, int data) {
    ll_node * n = malloc(sizeof(*n));
    n->data = data;
    llnode ** currp = &head;
    while (*currp != NULL && (*currp)->data < data) {
        currp = &(*currp)->next;
    }
    n->next = *currp;
    *currp = n;
    return head;
}
```

ECE 590.01 (Hilton): ADT
Put into list

Way two: “black belt in pointers”

```
ll_node * add_sorted(ll_node *head, int data) {
    ll_node * n = malloc(sizeof(*n));
    n->data = data;
    llnode ** currp = &head;
    while (*currp != NULL && (*currp)->data < data) {
        currp = &(*currp)->next;
    }
    n->next = *currp;  
    *currp = n;
    return head;
}
```

ECE 590.01 (Hilton): ADT
Return the list
Solution 2: Examined

• Solution 2 much better
  • No more corner cases
  • Want to change ordering? 1 place
  • Don't get pointers? Probably struggled to follow it
    • Could be good to trace through it

• Also, functionality problem -> level of indirection
  • "It's hard to add to the front and requires special code"
  • "No it doesn't, just add a level of indirection"

Solutions 3: Recursion

```c
ll_node * add_sorted(ll_node *head, int data) {
    if (head == NULL || head->data > data) {
        ll_node * n = malloc(sizeof(*n));
        n->data = data;
        n->next = head;
        return n;
    }
    head->next = add_sorted(head->next, data);
    return head;
}
```

Base case: want to insert here
  • `head` is `NULL`
  • - or `head` has bigger data

Recursive case
  • Add to the rest of the list
  • Update `next` to reflect whatever list came back
  • - Return this list

Solution 3: Examined

• No code duplication
• Simpler pointer manipulation
• Some people struggle with recursion for some reason
  • All you need to do to understand recursion is understand recursion!
• General algorithmic pattern for recursive data structure update
  • curr->x = recurse(curr->x, ...);
  • My (left/right/next/prev/...) pointer is going to be the result of recursively doing this operation on my (left/right/prev/...) pointer
• All, quite amenable to functional data structures
  • Never change an existing thing
  • Just make a new thing like the old thing, but slightly changed

Functional Data Structures

```c
s1 = push(4, push(3, make_stack()));
s2 = push(s1, 5);
s3 = push(s1, 42);
```

Functional Data Structures

• Functional Data Structures: wonderful things...
  • If you have garbage collection
    • Without it, need to explicitly reference count your nodes to know when to free
      • Know how many different lists something is in
  • Good implementations do minimal copying...
    • But give you the effect of having a full copy
    • Can be quite useful when you need it
Solution 3 (a): Functional Data Structure

```c
ll_node * add_sorted(ll_node *head, int data) {
    ll_node * n = malloc(sizeof(*n));
    if (head == NULL || head->data > data) {
        n->data = data;
        n->next = head;
        return n;
    }
    n->data = head->data;
    n->next = add_sorted(head->next, data);
    return n;
}
```

Make new nodes at every step:
- Old data structure unchanged
- Common part shared between structures

Interfaces and Implementations

<table>
<thead>
<tr>
<th>ADT (interface)</th>
<th>Implementation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Stack</td>
<td>Array</td>
</tr>
<tr>
<td>Stack</td>
<td>Linked List</td>
</tr>
<tr>
<td>Queue</td>
<td>Array</td>
</tr>
<tr>
<td>Queue</td>
<td>Linked List</td>
</tr>
</tbody>
</table>

Queues

- Stacks: LIFO
- Queues: FIFO
  - Generally call operations `enqueue` and `dequeue`
  - Add to end, remove from the other

Array implementation?
- Possible, but kind of a pain unless fixed size
- Two indices for "head" and "tail"

Linked List Implementation?
- Quite doable
- Probably want a pointer to the end of the LL
- Add at end
- Remove from front

Deque ("Deck")

- Double-ended queue: deque (pronounced "deck")
- Add to front or back
- Remove from front or back

Implementation Options?
- Array: super icky
- Linked List: doable
  - Add to front: easy
  - Remove from front: easy
  - Add to back: easy if we have "end" pointer
  - Remove from back:
    - Need to iterate through whole list to find next-to-last
    - And update end pointer

Doubly Linked Lists

- Doubly linked lists have next and prev in each node
  ```c
  struct _dbl_ll_node {
    void * data;
    struct _dbl_ll_node * next;
    struct _dbl_ll_node * prev;
  };
  typedef struct _dbl_ll_node dbl_ll_node;
  struct _dbl_ll {
    dbl_ll_node * head;
    dbl_ll_node * tail;
  };
  typedef struct _dbl_ll dbl_ll;
  ```
Doubly Linked Lists

- Should always maintain invariants
  \[ x->next == NULL || x->next->prev == x \]
  \[ x->prev == NULL || x->prev->next == x \]
- Can use assert to abort program if invariant violated
  \[ \text{assert}(x->next == NULL || x->next->prev == x); \]

Why assert?

- assert(x):
  if (!x) { print error message; abort(); }
- So why do it?
  - assert fails: program crashes...
  - Why not, cross fingers and hope that if its not right, nothing happens?
- Three reasons (who can think of them)
  1.
  2.
  3.

Why assert?

Three reasons
1. A wrong answer is far worse than no answer at all
   - Consider...
     - Missile trajectories
     - Stock trading software
     - ...
   2.
   3.

Why assert?

Three reasons
1. A wrong answer is far worse than no answer at all
   - Consider...
     - Missile trajectories
     - Stock trading software
     - ...
   2. Failing sooner is better than failing later
      - Want to detect problem as close to its source as possible
      - Rather than just strange behavior/crashing later
      - Plus, have specifics on the cause of the problem
   3.

My advice to you: Assert a lot

- Assert is your friend
  - Use it as much as you can
  - You should know your invariants, and thus should be able to write them down
    - If you don’t, you are already in dire straights
  - Will ultimately save you much time debugging
Summary

• Today:
  • Interface vs Implementation
  • ADTs:
    • Stacks
    • Queue
    • Deque
  • Data structures to implement them with (so far)
    • Arrays
    • Singly Linked Lists
    • Doubly Linked Lists
  • More on pointers
  • Recursion
  • Functional data structures
  • Assert