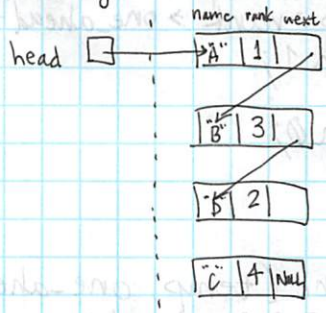
2017|08|15 TUESDAY • first: home-brew bubble sort || then: qsort! • ①

• sorting linked list. Let me quickly imagine what what might look like:



I want 1, 2, 3, 4 ranking

```
temp = head;
int* counter = 0;
while (temp) {
    ~~counter = temp->rank~~
    if (counter < temp->rank) {   (if 0 < 1)
        counter = temp->rank;
        ~~temp = temp->next;~~
    } else {
        Sort_operation(temp);
        ~~temp = temp->next;~~
    }
    temp = temp->next;
}
```

• perhaps I should take a step back and remind myself how I'd sort simpler objects? I already did @ w/ qsort and arrays: pretty straightforward. Maybe I should look at my bubblesort self-made program to remind myself how some of the gets might work? That might make it easier to manipulate a linked list?

```
int rank_sort (const void *va, const void *vb) {
    float * a = (float *)va;
    float * b = (float *)vb;
    if ( (*a) < (*b) ) {
        return -1;
    } else if ( (*a) > (*b) ) {
        return 1;
    } else {          // if ((*a) == (*b))
        return 0;
    }
}
```

Sandbox/08
08b
09

• ooh boy, so I've found my old sorting work and it's very useful to find how I've created my own bubble sort and also integrated qsort to do more sophisticated (homemade) brains. But how to move forward from here? how to harness my discovery of my old work and internalize it and then manipulate it to work for a linked list (and then, how to understand what ~~I~~ I even did?)? I suppose I could compare and contrast my two old files to see if any gems can be gleaned from @ old work. And I could diagram the memory of my linked list against the logic of the sorting functions I created. And, of course, I could always talk w/ Jun!

• so, evenoddBrain() is the same btwn sandbox/08 and 11, minus one minor tweek in 11: ~~assign~~ typecast const void * variables as (int*), ~~and so, overall, the~~ →@'s w/c were interfacing w/ qsort instead of my own homebrewed sort func.

(think about how I'd do switchfunc() and complogicFunc() w/ linked list)

```
~~while (linked_list) {~~
temp = linked_list;
~~next~~ one_ahead = temp→next;
while (temp) {
    compLogicFunc (temp ~~→rank~~, one_ahead);
    temp = temp→next;
    one_ahead = temp→next;
}
```

```
int compLogicFunc (temp, one_ahead) {
    if (temp→rank > one_ahead→rank) {
        return 1;
    } else {
        return 0;
    }
}
```

whew! this is exhausting!

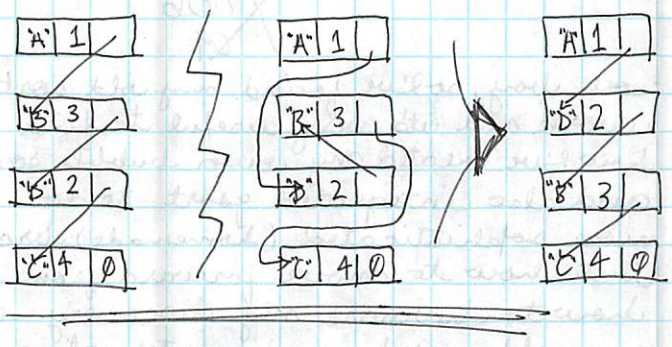
not a pointer :→

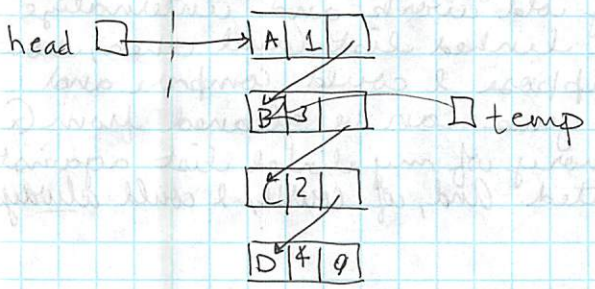
```
void switchFunc (temp, one_ahead) {
    ~~int~~ crane = one_ahead;
    one_ahead = temp;
    temp = crane;
}
```

```
~~node~~ node * temp = ~~linked_list;~~ new node;
temp = linked_list;
node * one_ahead = new node;
one_ahead = temp→next;
std::string name = "";
float rank = 0.0;

compLogicFunc (temp, one_ahead);
```

```
void switchFunc (temp, one_ahead) {
    std::string = one_ahead→name;
    float rank = one_ahead→rank;
    one_ahead = temp;
```





~~a→~~
~~a→next = c~~
~~b→~~
a→next = c;
c→next = b;
b→next = d;

Jim binary search

- array of memory addresses, then swap them around (on paper)
- now I've got list of memory addresses in correct order.
- now update marker boards.
  - set x → next to literal memory address as it appears on array.

qsort does this sorting.

I heap array = memory addresses of nodes
return array (ll to a converter)

II impose order ~~a~~ of memory addresses in array onto existing linked list. (a to ll converter)

- assumptions

I&II node is struct ~~void* data;~~ ~~int*~~ int data;
node * next;

(I) API for ll to a func.

~~return~~ no return (void func)

2017/08/17: my function looks different: also pass array variable as argument

node ** ~~void~~ array_ify (node * head_pointer, int * size_buffer)

8/17 no return variable/value: array directly manipulated...

as defined before func

//(return value is pointer to array of node * 's)
// array size = size_buffer

size_buffer
↳ doesn't contain useful info ~~before~~ before func runs
(blank notepaper...)
↳ this is API for (I)

8/17 my 'size buffer'/dynamic allocation is also different from Jim's vision

(II) node * list_ify (node ** node_array, int size)

same as size_buffer, but it's not an int * so doesn't need/can't be manipulated, which is why it's int and also just size

//return value can be assigned to head_pointer in calling context "it is the new linked ~~list~~ list"

= = =

I guess tomorrow I'll just keep on keeping on! Whew! Exhausting!!!

commentary 8/17: it appears ① main differences between my code and Jim's API are ① array pointer and dynamic array length (I) both defined before function (rather than within -- sort of -- and ② my function ~~is~~ is of type void and Jim's function returns type node** → which, I must admit, is a bit confusing to my current level of understanding/scope of vision...

2017/08/16 Wednesday

Jim: user enters numbers to add to stack → add to linked list . . . .

~~void add_new_node (node * * head) {~~

- how to transition from array to node? I'm sure I'd incorporate brains of, say, insert_new_at_front() in std::cin brains, but I'm not exactly sure how . . .

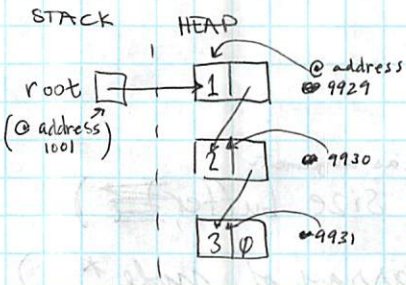
```
node* temp=root; root;
while ( std::cin >> user_var ) {
    std::cout << "enter integer:";
    std::cin >> temp -> data;
    temp -> next = root;
    if ( temp -> data == 0) {
        break;
    }
}

print_node(root);
```

```
int user_input;
std::cout << "enter non-zero integer: " << std::endl;
while (std::cin >> user_input) { //or something
    std::cin >> user_input;
    if (user_input != 0) {
        temp -> data = user_input;
        temp -> next = root;
        root = temp;
        temp = NULL;
    } else if (user_input == 0) {
        std::cout << " all done!" << std::endl;
        break;
    } else {
        std::cout << "input error" << std::endl;
        break;
    }
}
```

print_heap(node); ← passes value node points at (address of node is sacred, unaffected.)

print_stack(& node); ← passes address of node



- if I pass (& root) to function, I pass address 1001
- if I pass (root) to function, I pass address 9929.

print_heap(root) ⇒ print_heap(node * head)

print_stack(&root) ⇒ print_stack(node * * head)

1 ✓ place cin>> while() loop in black box

2. instead of just printing addresses, I want to push them into an array.
   - ✓ → first manually set length of array
   - → then use buffer or something
   - ✓ → also: array on stack or heap (try heap, but start on stack if ☺ makes it easier...)

~~int node_addresses[];~~

~~const~~

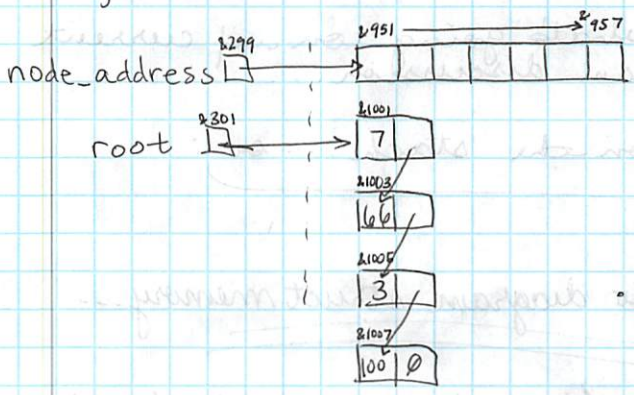
- functions deal w/ variables @ the same levels (apparently) but the extra *'s provide the proper level of push and pop → I suppose it's connected w/ what I'm doing w/ the variables. Anyways, the functions want access to the very end (value of address pointers), while the calling side may provide access to the root address on stack or to some level inbetween... or something...

```
const int SOME_NUM = #;
int node_addresses[SOME_NUM];
//or
int *node_addresses = new int[SOME_NUM];

for (int i = 0; i < SOME_NUM; i++) {
    while (node) {

    node_addresses[i] = (&node->data);
    node = node->next;
    }
}
```

- I'm not convinced I'm not missing something, but I guess it's worth a try...




```
while (root) {
    for (int i = 0; i < 7; i++) {
        node_address[i] = (&root->data);
        root = root->next;
    }
}
```

- first pass
  while (root) (1001)
  for (i = 0)
    node_address[0] = 1001;
    root = 1003;

- second pass
  ~~while (root)~~ (1003) this won't get checked b/c for loop isn't over.
  for (i = 1)
    node_address

```
for (int i=0; i<7; i++) {
    while (root) {
        node_address[i] = (&root->data);
        root = root->next;
    }    if (root==NULL) break;
}
```

- first pass
  for (int i = 0)
  ~~while (root~~
  node_address[0] = 1001;
  root = 1003;
  (root doesn't equal NULL)

- second pass
  for (int i = 1)
  node_address[1] = 1003;
  root = 1005;
  (root doesn't equal NULL)

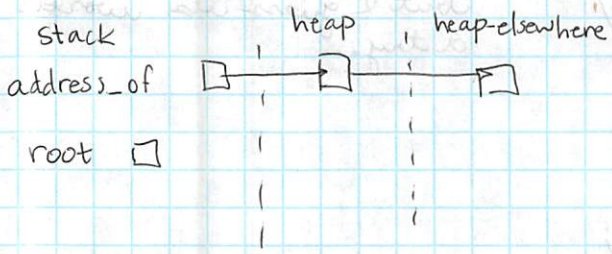
- third pass
  for (int i = 2)
  node_address[2] = 1005;
  root = 1007
  (root doesn't equal NULL)

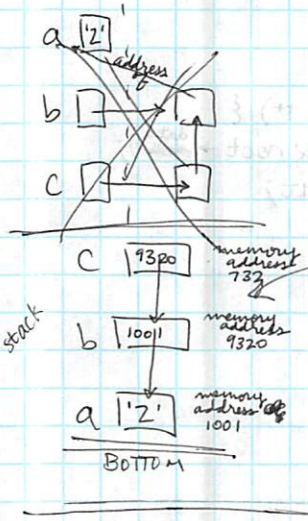
- fourth pass
  for (int i = 3)
  node_address[3] = 1007;
  root = NULL;
  (root does equal NULL)
  break;
  (don't continue along for() loop for any longer)

4 (3 entries)

```
int ** address_of = new int * [SOME_NUM];
node* root = new node;
```
← pointer to pointer

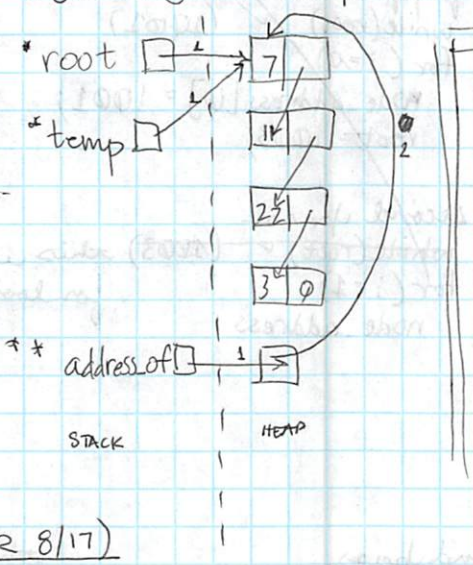
```
struct node {
    int data;
    node * next;
};
```





- I think this is what's going on w/ current blackboard / cplusplus.com discussion...
- but these are all on the stack... so:

* have Jim help me to diagram struct memory...



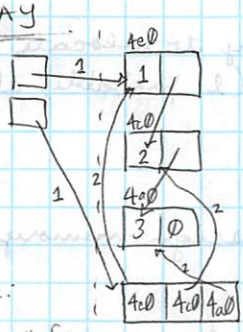
✓ break function out of main and into its own black box?

TOMORROW: (THUR 8/17)

- few! I'm losing light and, perhaps, brainpower. I'd like to put in a solid 7hrs though. For tomorrow:
  - continue w/ linked_list_sort_b.cpp
  - ✓ try to make print_array black box, if possible w/o Jim & w/o wasting too much time (consider diagram some more)
  - if not, move to studying buffers to make my array more dynamic in size (or, create a helper function @ finds the length of finished linked list, and then use @ callback to define length of address_of array in heap memory)
  - perhaps address_of array should be in stack: would @ make things easier?
- Show Jim my code so far, of course.
- And, if I can't make progress on this, I could always move to ~~the~~ home-brewed bubble-sort function! A different project w/ different challenges, to be sure.
- And, there's always the web app!!!

2017/08/17 THURSDAY

node * root
int ** address_of



✓ Jim says this is more or less correct

//linked_list_sort_b.cpp functions:

```
int ** address_of = new int *[SOME_NUM] ⇒//function fills array w/ memory addresses [4c0, 4c0, 4a0]

node_address_to_array(root, address_of, SOME_NUM); //~~decla~~ call function

void node_address_to_array(node *head, int ** array, const int a_length) { //define function
    node * temp = head;
    for (int i = 0; i < a_length; i++) {
        array [i] = & temp→data; //value of array element = address of temp→data
        if (temp→next == NULL) {break;}
        temp = temp→next;
    }
    temp = NULL;
}
```

= = = = //qsort.cpp print function:

```
int* int_array = new int[ARRAY_SIZE]; //plus added code @ populates array: {5,111,-86,84}
~~int int_array[ARRAY_SIZE] = {5, 111, -86, 84}; // on stack~~
print_int_array (int_array, ARRAY_SIZE); // call function

void print_int_array (int *a, const int a_lgth) { //define function
    for (int i = 0; i < a_lgth; i ++) {
        Std::cout << a[i] << "  ";
    }
    Std::cout << std::endl;
}
```

= = = =

//raw ~~da~~ code:

```
Std:: cout << "address_of[i]: " << Std::endl;
for (int i = 0; i < SOME_NUM; i ++) {
    Std::cout << address_of[i] << Std::endl;
}
```

• Shucks → trying similar memory abstraction and manipulation of array in qsort.cpp has the same errors → I guess nice @ I can recreate error, but too bad @ I'm still stumped on ~~for~~ figuring out a solution. Perhaps search the web for tips on printing ~~do~~ pointer to pointer...

= = =

• lesson learned: if there are multiple layers of pointers, but the desired functionality is otherwise the same (such as, for instance, a printer function), the only changed place is in the arguments listed when defining a function. Not on the variables within a function, not on the variables passed when calling a function. Those are all the same regardless of the level of pointers. The arg's passed when defining a function hold the magic key to push or pop to/from memory.

- how would I homecook a dynamic way to allocate memory / size to array?
  - function to find length of linked list (I probably already have one I could copy)
  - ~~use~~ set a length variable to ⓐ length
  - assign ⓑ length to array
  - proceed as I've already got written: assign memory addresses of nodes to array elements, then print.

- home-brewed dynamic array length easy! See if there's anything else I can do on this w/o Jim's code review step.
  - → hmm, I don't think there's much else I can do w/o embarking on a new mini project w/o Jim's intervention. I did identify ⓒ my current code does <u>not</u> match Jim's API, but I don't know what to do.

main/driver → prog.cpp (directly referenced in build command)

object → object.h (referenced indirectly through #include)

object → object.cpp

2017/08/19 SATURDAY

node_address_to_array()

print_address() (not cout << head << endl;)

- I think before I tried std::cout << &head << std::endl;
  - of course ⓐ wouldn't work right: I'm printing the memory address of the pointer (which doesn't change), not the pointer /aka, not the address of/reference to the value pointed at (~~w~~ which does change ~~#~~ b/c ~~I the~~ my look directs head to point to different values as it goes).

maybe temp is useful here, as a placeholder?

insert_new_at_front() } both have unneeded temp variables:
node_address_to_array() } head var is already a kind of temporary variable (or so says Jim).

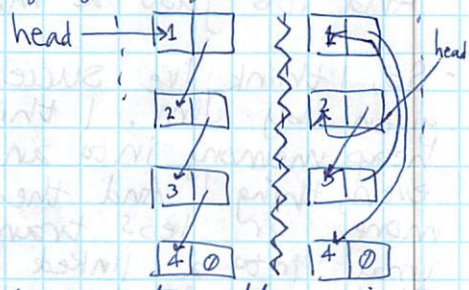
- declaring my array a_address_of_node (need to figure better name) w/in array_ify() is cleaner b/c size_buff and array are both defined and declared w/in array_ify function → more self-contained black boxes, less reliance on one-off code in main. But I still don't understand how node ** array can simply = another array w/o iterating through elements and manually defining them one by one.
  - Perhaps, before implementing array_ify, I should play a bit: create a couple of one-off arrays and define them to each other, and also lookup literature on c++ arrays

- ok, so website readily volunteers ⓑ this is kosher: int foo[] = {10,20,30} → ⓑ the compiler is smart enough to count the length. So a minor edit in Jim's declarations linked w/ array_ify() would match ⓑ setup.

· ha! vindicated at least a little bit! my instincts re array assignments appears to be at least partially true! Now to see if I can find an answer.

· hurrah! figured it out!!! Silly pointers! → OK, not vindicated after all:

Jim's code also contains pointers. But I'm still struggling to figure it out when implementing a function

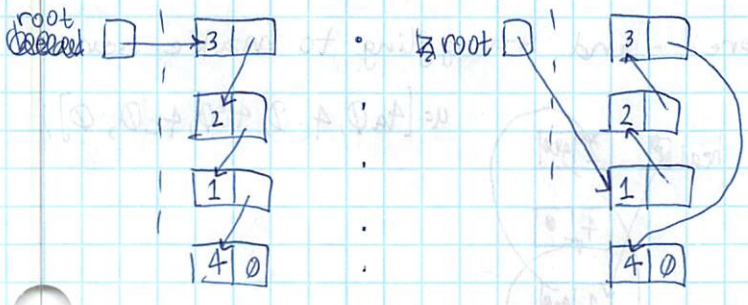


=== === ===

array to linked list

```
node * list_ify(node * * a_of_node_address, int size) {
    // return value assigned to root in main()
}
```

```
node * head = new node;      //declare heap memory node to return after assignment
//or
node* temp = new node;
```



don't change value of data, but change value of next

~~while (temp) {~~
~~for (length of array) {~~
~~node → next = array[i]~~
~~node~~

keep track of one ahead before change?

for (length of array) {
~~temp = node → next~~
one_ahead = node → next;
node → next = array[i];
node = one_ahead;
}

first	second	third
(points to 2)	(points to 1)	(points to 4)
(3 = points to 4)	(2 = points to 3)	(1 = points to 2)
(points to 2)	(points to 1)	(points to 4)

fourth
(points to NULL)
(4 = points to NULL)
(points to NULL)

> also point head to 1!

2017|08|31 THURSDAY

- Oh man, it's painful to dive back into a project that I haven't thought of in a while. Especially b/c now I remember just enough about what I was doing before to know that I'll have to strain and struggle a bit to get into any kind of groove. But that's what it's all about, isn't it? And it's just a matter of getting my hands dirty and diving in!

- So, I think I've successfully completed my first of two assignments given by Jim, I think. It was to 'translate' a linked list heap memory into an array of addresses. I've got array-ify and everything! And the second half of the assignment was to more or less translate the memory addresses back from an array into a linked list.
- Jim's proto-function looked like this:

```
node * list_ify (node ** node_array, int size) {
    // return value can be assigned to head-pointer in
    calling context "it's the new linked list"
}
```

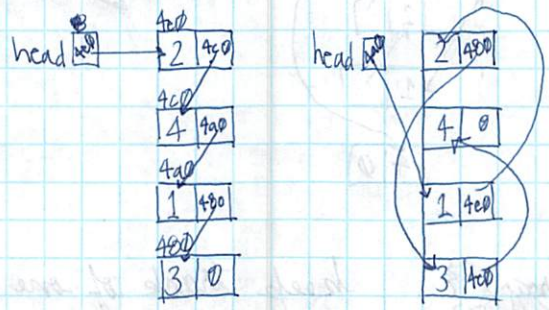
- I remember trying to diagram this before -- and struggling to make sense of it all.

1 2 3 4 2 4 1 3

~~[4a0, 4e0,~~

[480, null, 4e0, 4c0]

head points to 4a0

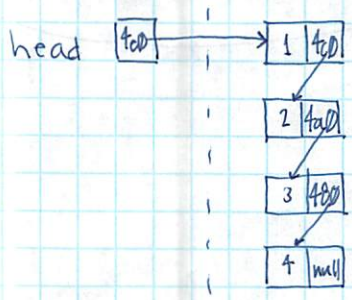


a: [4a0, 4e0, 480, 4c0, 0];

2017|09|01 FRIDAY 9:57

- perhaps the final step of array-ify() should be to pass the value "null" to the final element of array

a: [4e0, 4c0, 4a0, 480, null];



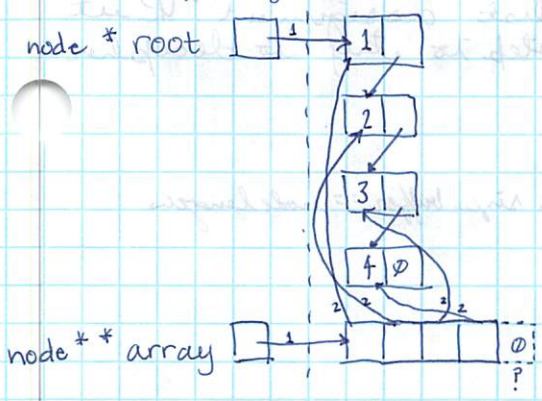
~~a[0]~~ ~~head~~

~~head = a[0];~~
~~head = head -> next;~~
~~head = a[1];~~
~~head = head -> next;~~
~~head = a[2];~~
~~head = head -> next;~~
~~head = a[3];~~
~~head = head -> next;~~
~~hea~~

head = a[0];
head -> next = a[1];
head

- I know how to access the addresses of the node->data, but I haven't figured out how to access node->next. Maybe ☺'s what I should work on first.

reminder of memory layers:



· for some reason I'm scared to run my code. Perhaps b/c I've got this irrational fear that it'll explode everything to smithereens if I try it out. But I should know @ ⓐ won't happen. I suppose I do know ⓐ I need some more supporting infrastructure: at the least, to find the 'size' of the array. I suppose I could quickly write an array-length helper function → even if it's not my final setup, it will likely help me to move forward. Whew! These seem like such basic concepts → I wonder why my mind is struggling so much? ~~A to~~ Regardless of the reason, I'm glad I've got the space to explore my thoughts and calm myself down a little.

2017/09/05 TUESDAY

```
insert_new_at_front (node ** head, int * a, int s) {
    for (int i=0; i<s; i++) {
        node* temp=new ~~da~~ node;
        temp -> data= a[i];
        temp -> next =  *head;
        *head =temp;
    }
}


void print_node_next(node * head) {
    if (head != NULL) {
        std::cout << " root->next: " << std::endl;
        std::cout << head << std::endl;
        while (head ->next != 0) {
            std::cout << head->next << std::endl;
            head = head -> next;
        }
        std::cout << head -> next << std::endl;
    }
}
```

what if I created function ⓐ set head->next values as separate variables? As yet another iteration towards my ideal final goal?

Super chill: give myself a little problem, and then low key low pressure try to solve it. No pressure. No huge strain, just flow and vibe.

- judgment (of myself or others) is its own kind of violence.
- holding myself ~~to~~ back doesn't allow me to live my full potential, ~~to~~ experience life and emotions at their full extent, and thereby gives me a more pale existence and doesn't let the world benefit as much as it could from me. ~~At~~ All this sounds pretty grandeous and intimidating but I'm trying to frame it as small and part of the everyday.

- node_next_to_array: I think this is the backwards version of what I want to do w/ Jim's array-to-linked-list assignment. But maybe (hopefully) it's valuable as a next step to try to decipher what to do to move forward?

```
node ** node_next_to_array(node * head, int s) {
    node ** a_of_node_next = new node * [s+2]; //or size buffer == node length
    a_of_node_next[0] = head;
    for (int i = 1; i < s+2; i++) {
        a_of_node_next[i] = head -> next;
        head = head -> next;
    }
    return a_of_node_next;
}
```

- how would I turn this around? so @ node -> next = array[i]?

```
node * array_to_node_next (node ** a_of_node_next, int s) {
    node * node_next_from_array = new node;
    node_next_from_array = a_of_node_next[0];
    for (int i = 1; i < s+2; i++) {
        node_next_from_array -> next = a_of_node_next[i];
        node_next_from_array = n_n_f_a -> next;
    }
    return ~~n_n_f_a;~~ a_of_node_next[0];
}
```

===

yay, it works! Now to see whether I can convert my code to something like Jim's assignment.

```
node* list_ify (
```

=====

whats next? Show Jim what I've got, have him say what he actually intended, and then go from there.

- If he's not available, what else could I do? I could rearrange array & play w/ @. I could create func @ rearranges array? A couple rando ideas.

- Well, there's your disaster... emergency dripping in kitchen has grabbed all of Jim's attentions. I'll try to plod along as best as I'm able on my own.

- based on the idea of having memory addresses in array, how would I then go on to sort or rearrange the values/addresses?
I imagine somehow working in tandem w/ ~~two~~ data values of node & w/ memory addresses of array. ~~Not to~~ To evaluate data values based on some defined metric, and use the ~~res~~ results of @ to resort array.
  - compare previous node w/ next node. if prev -> data > next -> data return x
  - based on return (x or y, 0 or 1 or 1 or 0), use @ logic to rearrange values of array elements. if prev -> data < next -> data return y

• Should I try a more sophisticated ~~s~~ sort brain than just bubblesort?

~~temp~~ node * temp = a[0];
a[0] = a[size-1];
a[size-1] = temp;

12-5 min

PART III of JIM'S ASSIGNMENT

* SORT USING QSORT

time
• 8:30 3:30 hrs logged 1:30 hrs remain to meet my goal. I could code till 10pm to meet that goal. If I stayed up till 12 that would give me 2 hrs. If I stayed up till 11 that would only give me 1 hr. Do I want to get to bed a little earlier, wake up a little earlier? Does that even matter? Perhaps coding for 1 more hour could be the compromise. 1 more hour to get the ball rolling, but still give me enough time to be chill about it? Perhaps if I poured myself some wine in 30 minutes I could keep going for the full 1.5 hr?
• I think I do like the idea of reintroducing pomodorro. Perhaps trying it w/ my old phone for a while → it's too clunky w/ Toggl, and also wonky to use w/ web browser. Maybe I could download a desktop pomodorro app? Maybe I'll have to look later on. For now, perhaps I'll use Note 11.

DID IT!!!!

• tomorrow: clean up this mess, integrate into ~~r~~ ~~st~~ road-race 200m dash game.

Jim:
reflect on array → meaning, what it is, how use it

2017/09/07 Thursday

~~POMODORO:~~

what do I have?

• array ~~takes~~ contains memory addresses of each node in linked list. array is of type node ** instead of, say, int * b/c it's saving pointers for the whole node, not just, say, ~~the~~ node->data. This distinction is important especially for later use, when qsort is moving around entire nodes instead of just data elements w/in node (also, I assume the compiler's brains would get upset ~~too~~ at some level too).
• Having an array of pointers is helpful to use qsort to sort nodes of a linked list. Qsort only works w/ arrays, so to sort a linked list it needs an array middle step/interpreter/etc. In a way, ~~Q~~ qsort can work directly w/ linked list by ~~too~~ being passed an argument of an →

7/10 mi

outside perimiter of capitol (across-the-street sidewalks): 3685.55 ft (1.12 km)

array ⊕ contains pointers of nodes in linked list. W/ this access, qsort can compare data nodes of ll, determine how they should be sorted, then using ⊕ info to actively sort the elements (pointers) of the array. In many ways it works just like a regular qsort operation, just w/ one extra step. After qsorts changed the order of the pointers in the array, I need to use list-ify (in this case) to translate rearrangement onto the nodes of ll. (Just b/c qsort can access nodes [via array pointers] for comparison reasons doesn't mean ⊕ it can actively rearrange the ll itself → it can only directly rearrange the array elements). List-ify takes the rearranged array and assigns each array element to each node pointer in ll. In this case, it accesses the head pointer and each node->next, manually assigning NULL to the very last one. This final step keeps the existing linked list in place in heap memory, but it reassigns ~~the node~~ new values to the node pointers so ⊕ they point to new nods in ll. No memory should get leaked b/c each node is accounted for in array → it's like a backup copy of linked list's memory location. I could in principle delete the array after I'm done rearranging the linked list b/c ⊕ redundant memory backup's no longer needed (although since in my setup it's stack memory, the memory will deallocate automatically after the program ends.)

= = = ↓TRANSCRIBED ↓ TRANSCRIBED ↓ TRANSCRIBED ↓

What's my next step for 200m dash? I've got code ⊕ sorts ll (which has also given me, without really realizing it, much more experience w/ pointers), which was a big next step for 200m dash game/project.

- look back at old notes to revisit my initial conception of 200m dash game/project, and I had to go all the wayback to 03 August! Woah! It's not even in my current batch of notes, but in the stapled-already-scanned bundle. Time is wonk!
- I've got a program ⊕ creates a linked list of nodes, and allows me to add new players at the beginning, middle, or end of the list. I can also use qsort to sort the linked list of nodes in order of how fast they are.
- Moving forward, I'd want the ability to flip a coin. Maybe to randomly select a node from the linked list? Or to have the user select which player should challenge? Definitely have a way to track 1 node previous from the player chosen, so ⊕ the two of them can race. I suppose those features make up the bulk of the main game. Oh, I'll also want to reassign the time_sec portion of each of the player's nodes.
- What about this thought? Maybe I should reintroduce Ranking? B/c each mini race will depend on who wins to adjust the ranking, not necessarily on what the specific time outcome was. So, actually, the time_sec portion of each node adds more than the bare minimum complexity... Unless I were to have a feature that pitted all the players against the others in one big race?

B/c it's a 200m dash and can expand and contract as much as needed.
· Well, in some ways ~~@~~ a technicality...

(14oz @ $12 = $0.86/oz)(15% off = $0.73)

runners:
~~Bianca 1st place~~
~~Akmal 2nd place~~
~~Maurice 3rd place~~

runners:
1st Bianca
2nd Akmal
3rd Maurice
4th Aziz
5th Susan

before or after discount? 30 oz ×12 → 208.99 total ($0.58/oz) (like a 30% discount off of full-priced jars ≈ $8.12/14oz)

call tomorrow AM

· Select a challenger: _ [Aziz] ← user selection
· OK! 4th Place Aziz challenges 3rd place Maurice!
· Get ready... get set... go!
>>> Race Race Race <<<
· 200 m dash challenge over.
Aziz ran in (27.1) sec.
Maurice ran in (28.7) sec.
Aziz wins! Aziz advances one ranking! New layout:
1st Bianca
2nd Akmal
3rd Aziz
4th Maurice
5th Susan.

start over → · Select a challenger: _

consider ~~to~~ giving the user multiple options. Single challenger (2-person race). Whole group race. View rankings. ~~View~~ Add node datapoint that shows the time sec of the most recent race? And view time stats? (average run time, difference in time btwn first and last runners?)

· Background functions: accept user input. Func @ figures out random run times of challenger. Function @ determines a winner. Resort roster using array-ify, qsort, list-ify. Print user selection and print outs and race progress and race outcomes to the screen. Look through to go back to 'Select a challenger' prompt after a race is completed. Consider Printing out user selection options (runner name, whole group, show stats, quit program, etc?). So many considerations!

· I'm excited about looking to find coin flip functionality in C++, and also in thinking about how to assign a random racing time to each racer (w/ in a pre-defined range: say, 33 sec thru 20 sec).

· any considerations to weigh the outcomes at all, based on previous run times? That seems to like it would complicate things in the wrong direction → I want to play around w/ pointers, not w/ chance-generation.

↑ TRANSCRIBED ↑ TRANSCRIBED ↑ TRANSCRIBED ↑ TRANSCRIBED ↑