

MEMORY MANAGEMENT

6

CS/COE 0449
Introduction to
Systems Software

wilkie

(with content borrowed from Vinicius Petrucci
and Jarrett Billingsley)

Spring 2019/2020

OUR STORY SO FAR

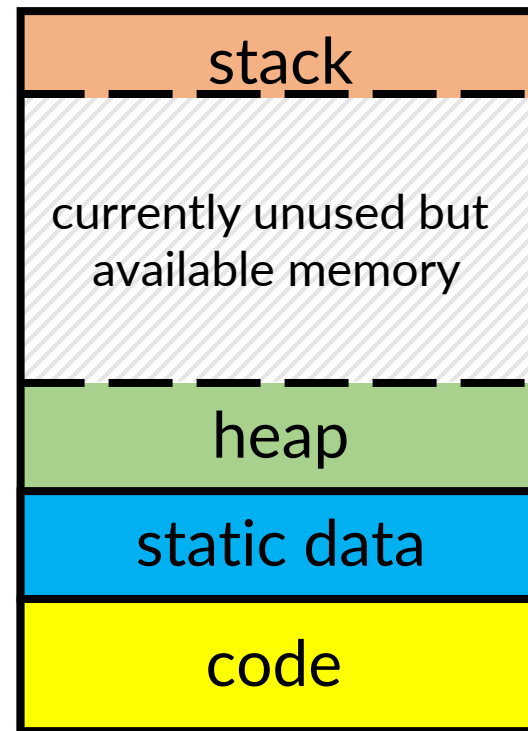
You Hear a Voice Whisper: “The Memory Layout is a Lie”

Reallocating our thoughts

- A program has several sections:
 - Code
 - Static data
 - Stack
 - Heap
- Today, we take a deeper dive at how dynamic memory is allocated in the heap.

Potential Layout
(32-bit addresses)

~ 0xFFFFFFFF

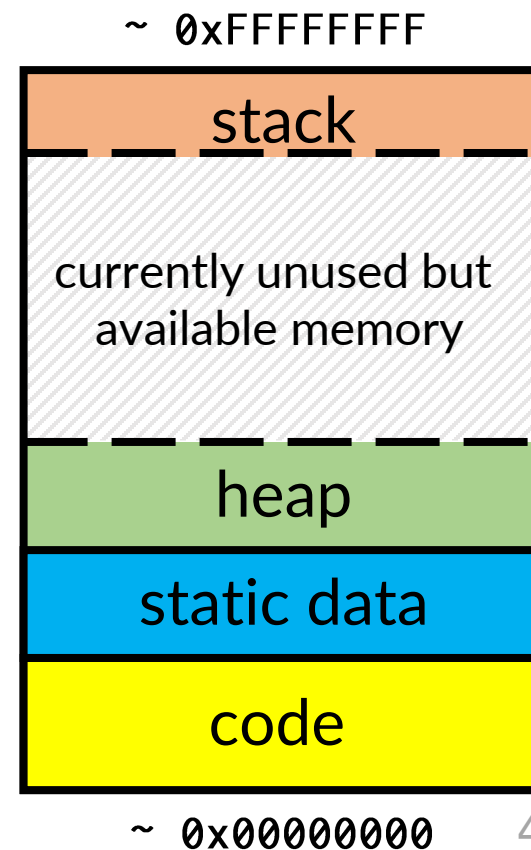


~ 0x00000000

Reallocating our thoughts

- We have looked at `malloc` and `calloc`.
- They stake out space in the heap and return an address.
- Right now, we live in a nice ideal world.
 - No other programs are running.
 - We have access to all of the memory.
 - Muhahahaha!!
- The OS is lying to our program.
 - This memory is... virtual... reality.
 - We will investigate this lie later in the course.

Potential Layout
(32-bit addresses)



THE WORLD OF ALLOCATION

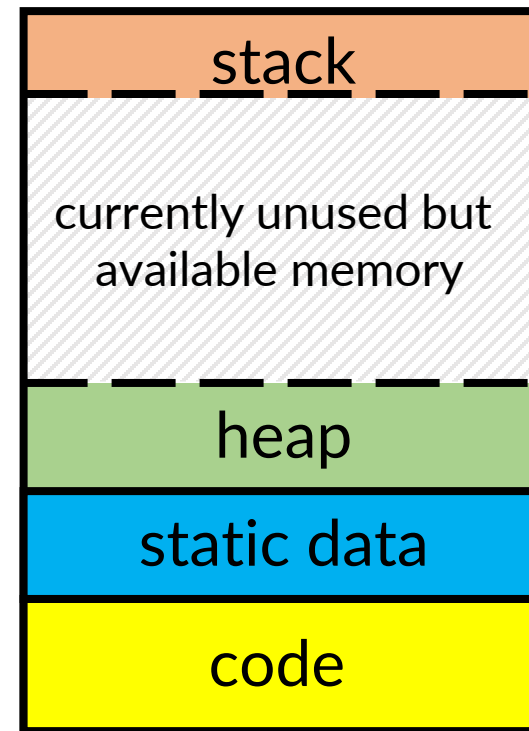
It is a puzzle without any optimal solution. Welcome to computers!

A heap of possibilities

- **Stack access often does not deviate much.**
 - We allocate a little bit at a time.
 - We allocate and free the memory VERY often.
- **Heap allocations have many access patterns that are possible.**
 - You might allocate a lot at a time and keep it around for a long time. Or a short time.
 - You might allocate a lot of small things, instead.
 - Maybe you do a little bit of everything?
- **Often, such patterns are not easy to predict.**
 - Do you get a big file as input? A small file?

Potential Layout
(32-bit addresses)

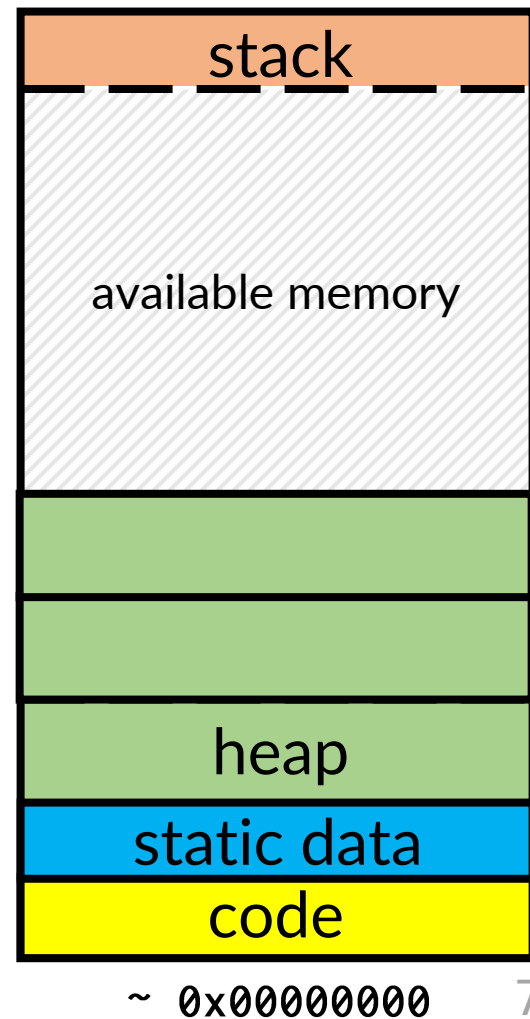
~ 0xFFFFFFFF



~ 0x00000000

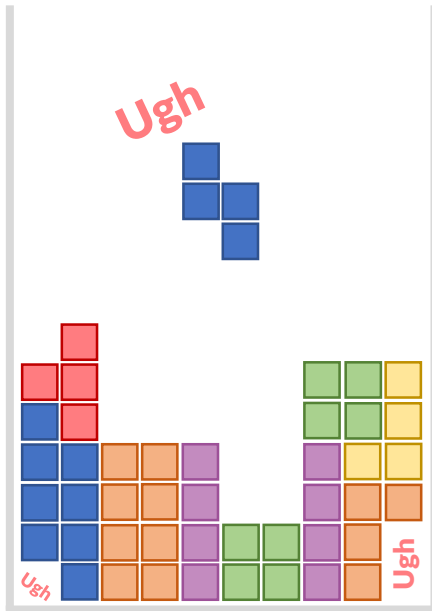
A heaping helping of good luck

- Allocations could happen in a nice order.
- When something is allocated, it can be allocated after everything else.
- When freed, it makes room for new things.
- **IF ONLY.**
 - I mean, it's possible... but like...
 - the heap and stack are different things for a reason.



Digital potholes... as annoying as real ones

- Small allocations interfere with large ones.
- When small gaps interfere with allocation, this is called *fragmentation*.



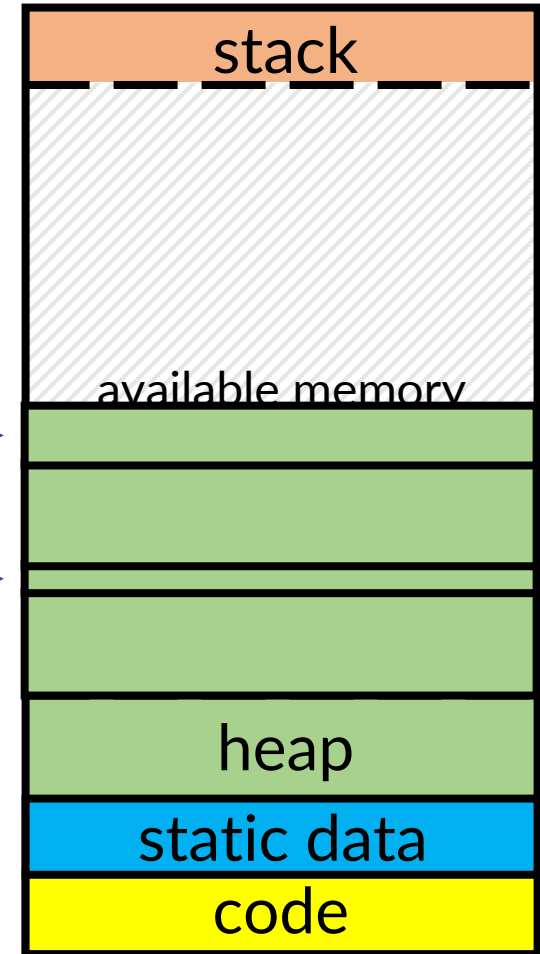
NEXT
ALLOCATION

?

if we had omniscience of future
allocations, we could avoid this...
but we can't know ahead of time!

malloc() →

free() →



~ 0x00000000

The worst case

- When you allocate a lot of small things...
 - Free every other one...
 - And then attempt to allocate a bigger thing...
- Even though there is technically enough memory...
 - There is no continuous space.
 - Therefore, our naïve `malloc` will fail.
- We have to come up with some strategy.

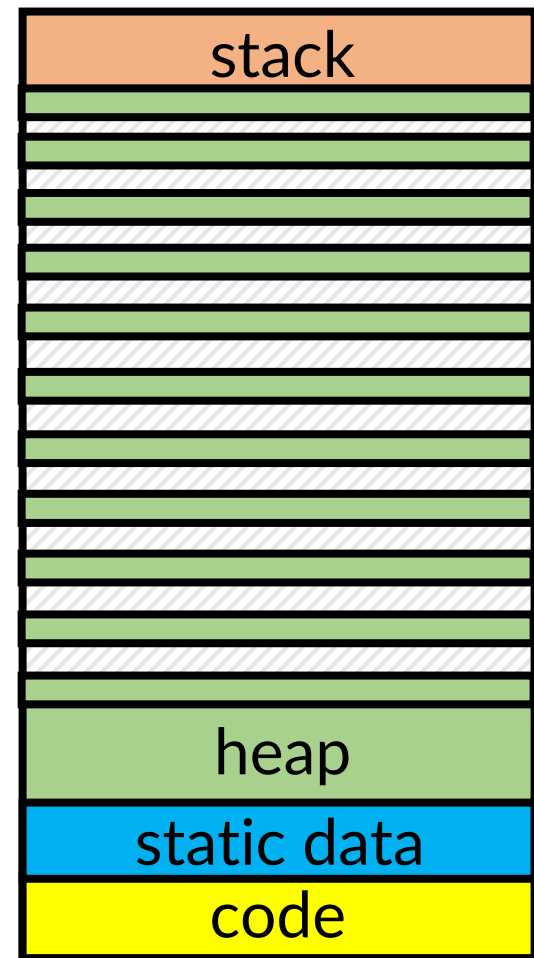
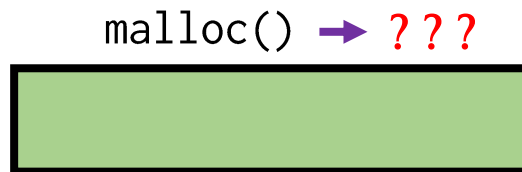
`malloc()` → ???



~ 0x00000000

Moving is never easy

- Why not move things around??
 - A *defragmentation* process/algorithm
- Moving around something in the heap is hard!
 - Any pointers referring to data within a block must be updated.
 - Finding these pointers automatically is effectively as difficult as garbage collection.
- Because of this, moving blocks around is discouraged. (Easier to solve it another way.)



~ 0x00000000 10

Moving is NEVER easy

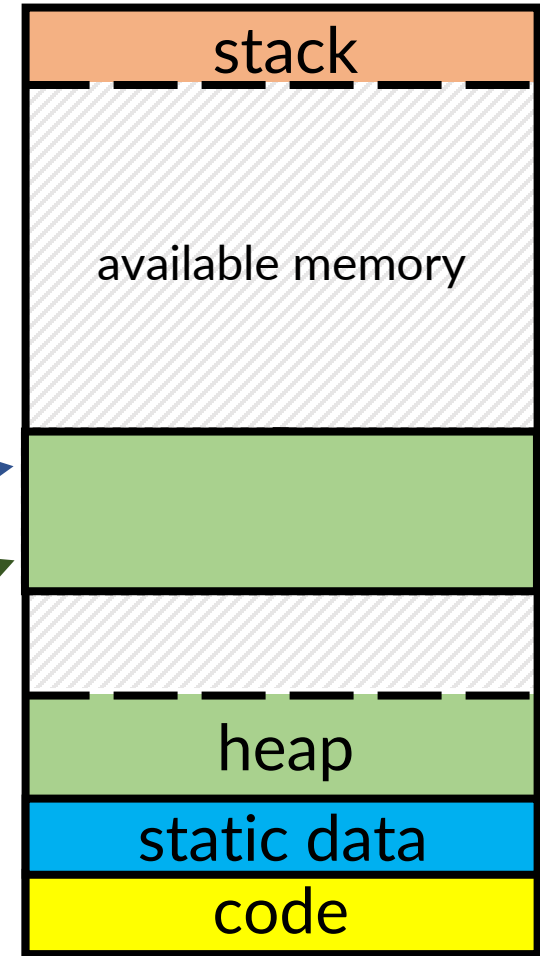
`int* my_int`

?????

`float* my_float`

-1.8e-6

- When blocks move, pointers to anything within them must be updated.
- This is hard to keep track of!
 - C does not check validity of pointers after `free()`

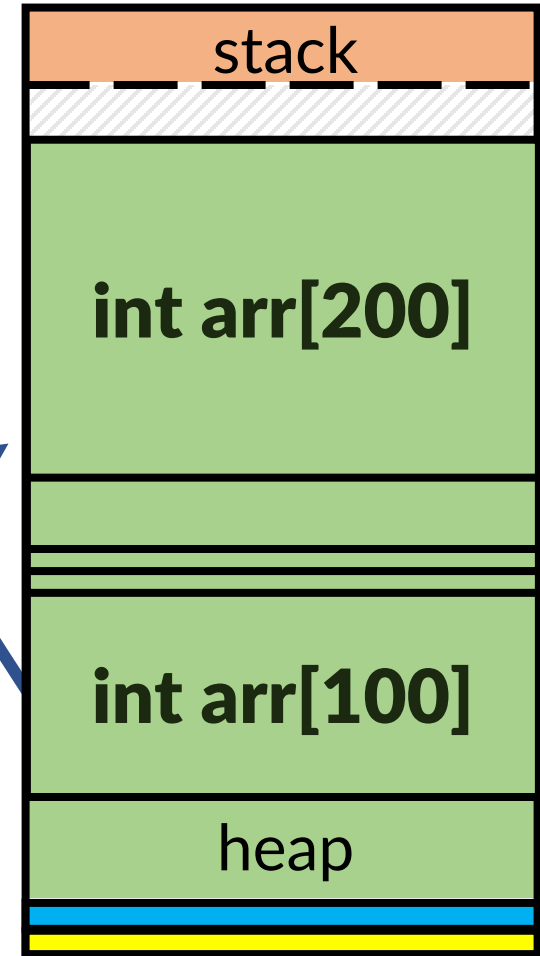


~ 0x00000000 11

Stressing it out

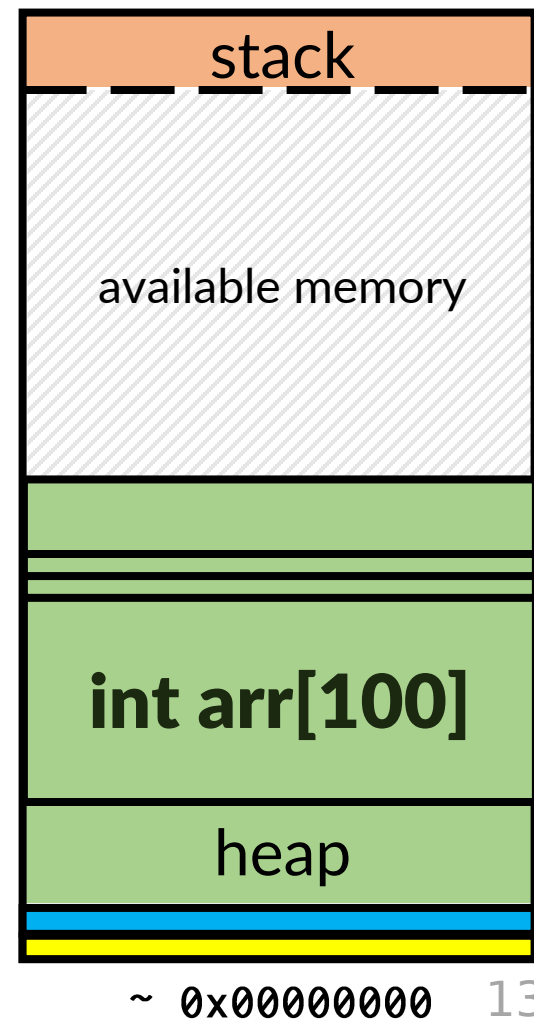
- If we allocate a large array it will be allocated on the heap somewhere.
- Other allocations can also happen, and they go “above” that array.
- What happens when you need to append a 101st element to this array?
 - Uh oh!
- You will need to allocate more space.
 - And then copy the array contents.
 - Free the old array.
 - How long does that take?

fragmentation



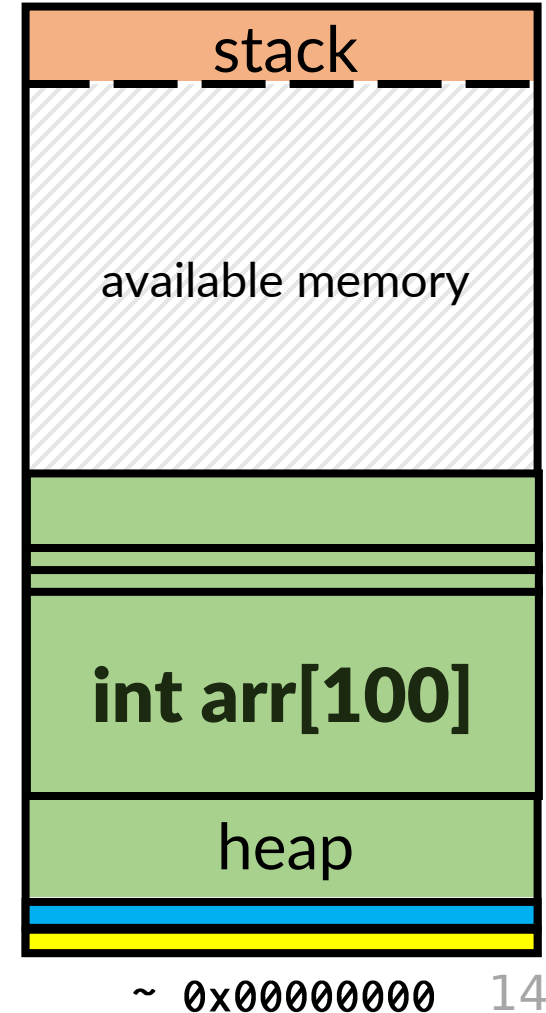
Stressing it out: Big Arrays

- This happens in very practical situations!
 - Reallocating means getting rid of a small thing
 - And replacing it with a larger thing.
 - You could have TiBs of memory and this will be a problem.
- This affects performance: (in terms of writes:)
 - Appending item `arr[0]`: $O(1)$
 - Appending item `arr[1]`: $O(1)$
 - ...
 - Appending item `arr[99]`: $O(1)$
 - Appending item `arr[100]`: $O(n + 1)$ oh no!
- When you would overflow the buffer...
 - You then need to copy all previous values as well.



Stressing it out: Performance Consistency

- Big arrays want to be continuous.
 - Ensuring continuous space is difficult when you do not know how much you will ultimately need.
- This is exactly why **linked lists** exist!
- Since a linked list allocates on every append.
 - Each append takes the same amount of time.
- However, everything is a trade-off.
 - Dang it!!!
 - One cost is extra overhead for metadata.
 - Linked list traversal can stress memory caches.
 - It means traversing the array is slower.
 - However, we will mostly ignore this for now.

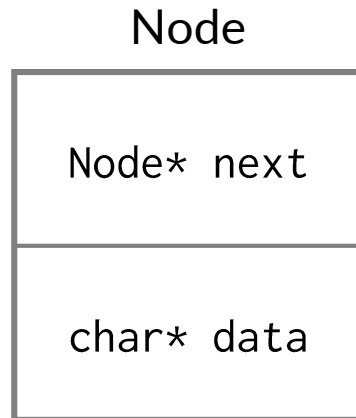


THE LINKED LIST

A story about trade-offs.

What is a linked list?

- A **linked list** is a non-continuous data structure representing an ordered list.
- Each item in the linked list is represented by metadata called a node.
 - This metadata indirectly refers to the actual data.
 - Furthermore, it indirectly refers to at least one other item in the list.



```
typedef struct _Node {  
    struct _Node* next;  
    char* data;  
} Node;
```

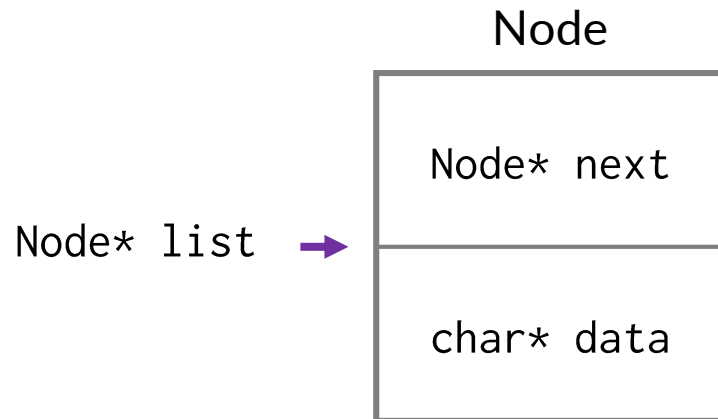
“struct” required since Node is not technically defined until after it is defined!



Keeping ahead of the list.

- Creation of a list occurs when one allocates a single node and tracks it in a pointer. This is the head of our list (first element.)

```
Node* list = (Node*)malloc(sizeof(Node));  
list->next = NULL; // NULL is our end-of-list marker  
list->data = NULL; // Allocate/copy the data you want
```

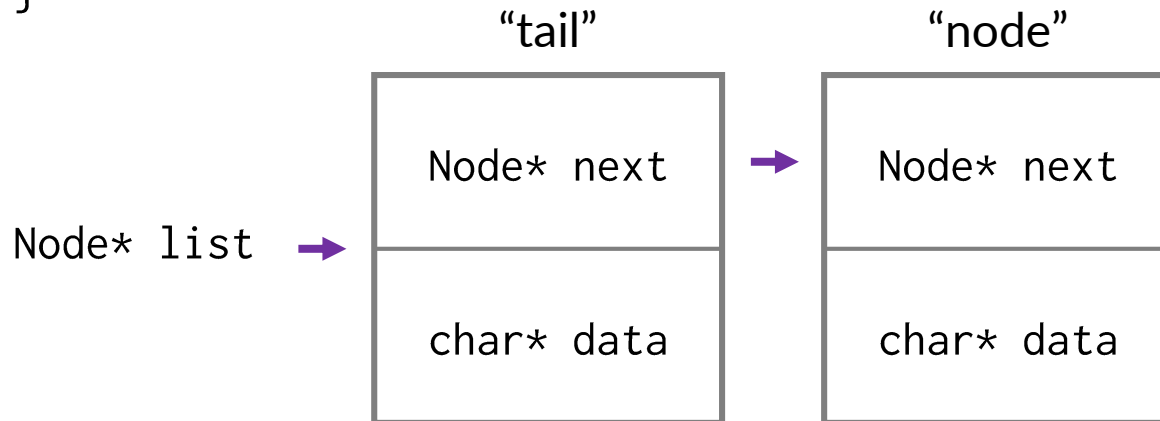


Adding some links to our chain

- If we want to append an item, we can add a node anywhere!

```
void append(Node* tail, const char* value) {  
    Node* node = (Node*)malloc(sizeof(Node));  
    node->next = NULL; // The new end of our list  
    tail->next = node; // We attach this node to the old last node  
    node->data = (char*)malloc(strlen(value, 100) + 1);  
    strncpy(node->data, value, 100);  
}
```

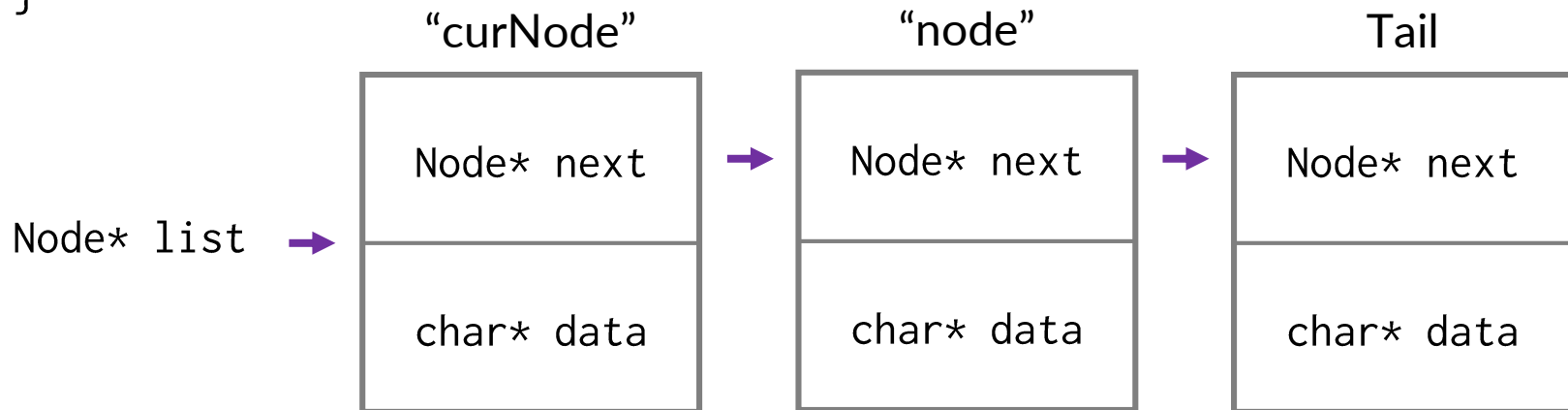
**Remember the
'\0' sentinel!**



We can add them anywhere!!

- Consider what happens if we update our append to take any Node:

```
void linkedListAppend(Node* curNode, const char* value) {  
    Node* node = (Node*)malloc(sizeof(Node));  
    node->next = curNode->next;  
    curNode->next = node;  
    node->data = (char*)malloc(strlen(value, 100) + 1);  
    strncpy(node->data, value, 100);  
}
```



We can add them anywhere!!

- This function has very consistent performance (constant time):

```
void linkedListAppend(Node* curNode, const char* value) {  
    Node* node = (Node*)malloc(sizeof(Node));  
    node->next = curNode->next;  
    curNode->next = node;  
    node->data = (char*)malloc(strlen(value, 100) + 1);  
    strncpy(node->data, value, 100);  
}
```

- The append always allocates the same amount.
- It always copies the same amount.
- Compare to a big array where you may have to copy the entire thing to append something new!




Traversal... on the other hand...

- Accessing an array element is generally very simple.
 - `arr[42]` is the same as `*(arr + 42)` because its location is very well-known!
 - This is because array items are continuous in memory. Not true for linked lists!
- Here is a function that performs the equivalent for linked lists:

```
void linkedListGet(Node* head, size_t index) {  
    Node* curNode = head;  
    while(curNode && index) {  
        index--;  
        curNode = curNode->next;  
    }  
    return curNode;  
}
```

Q: How many times is memory accessed relative to the requested index?

Removing... on the other, other hand!

```
void linkedListDelete(Node* head, size_t index) {
    Node* lastNode = NULL;
    Node* curNode = head;
    while(curNode && index) {
        index--;
        lastNode = curNode;
        curNode = curNode->next;
    }
    if (!curNode)  Can't find item at index.
        return head;
    if (!lastNode)  We are deleting the head.
        head = curNode->next; // New head is next item
    else
        lastNode->next = curNode->next; // Orphans item
    free(curNode->data);
    free(curNode);
    return head;  Returns new head (or old head if unchanged).
}
```

- One nice thing about linked lists is their flexibility to changing shape.
 - I used to be able to bend a lot better, too, when I was in my 20s. Alas.
- Since we don't have a way to go "backward"
 - We first find the node we want to delete (curNode)
 - Keeping track of the node of index - 1 (lastNode)
 - Rewire lastNode to cut out curNode.

Removing... on the other, other hand!

```
void linkedListDelete(Node* head, size_t index) {
    Node* lastNode = NULL;
    Node* curNode = head;
    while(curNode && index) {
        index--;
        lastNode = curNode;
        curNode = curNode->next;
    }
    if (!curNode)
        return head;
    if (!lastNode)
        head = curNode->next; // New head is next item
    else
        lastNode->next = curNode->next; // Orphans item
    free(curNode->data);
    free(curNode);
    return head;
}
```

- This looks complex, but it really is a simple traversal.
 - So it takes $O(n)$ to find the item.
 - And it performs a simple update and deallocation. (quick to do)
- A big array, on the other hand.
 - It can find the element to remove immediately.
 - However, removing it means shifting over every item after it left.
 - That can be an expensive update! (Memory is slow!!)

On your own!

Think about the code you would need to do any of the following:

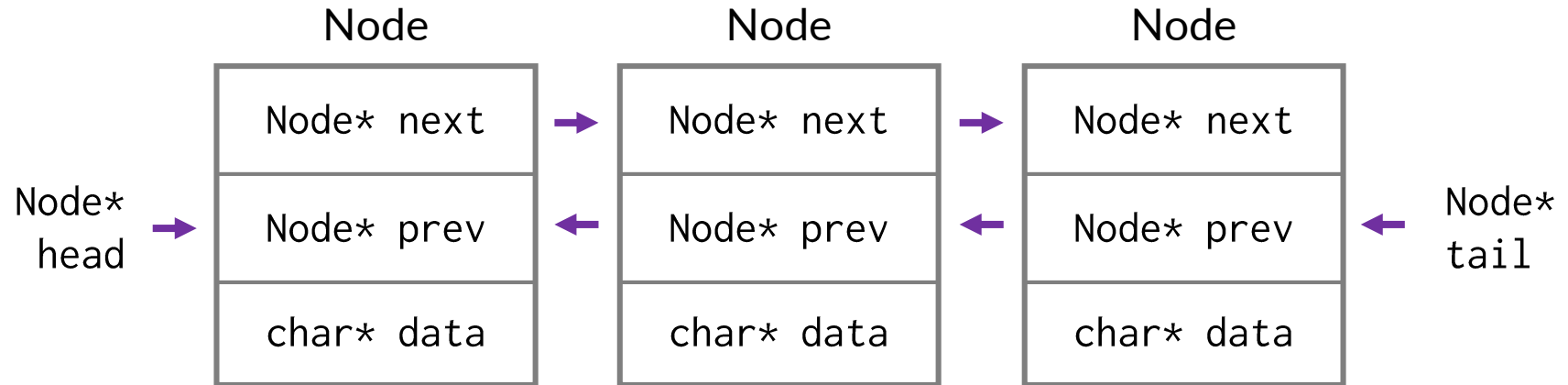
- Delete/free the entire linked list.
- Sort a linked list.
- Append a linked list to an existing one.
- Copy a subset of a linked list to a new list.

Often, operations can be abstracted in such a way that all of these can be written relatively simply.

Consider the performance of these operations compared to an Array.

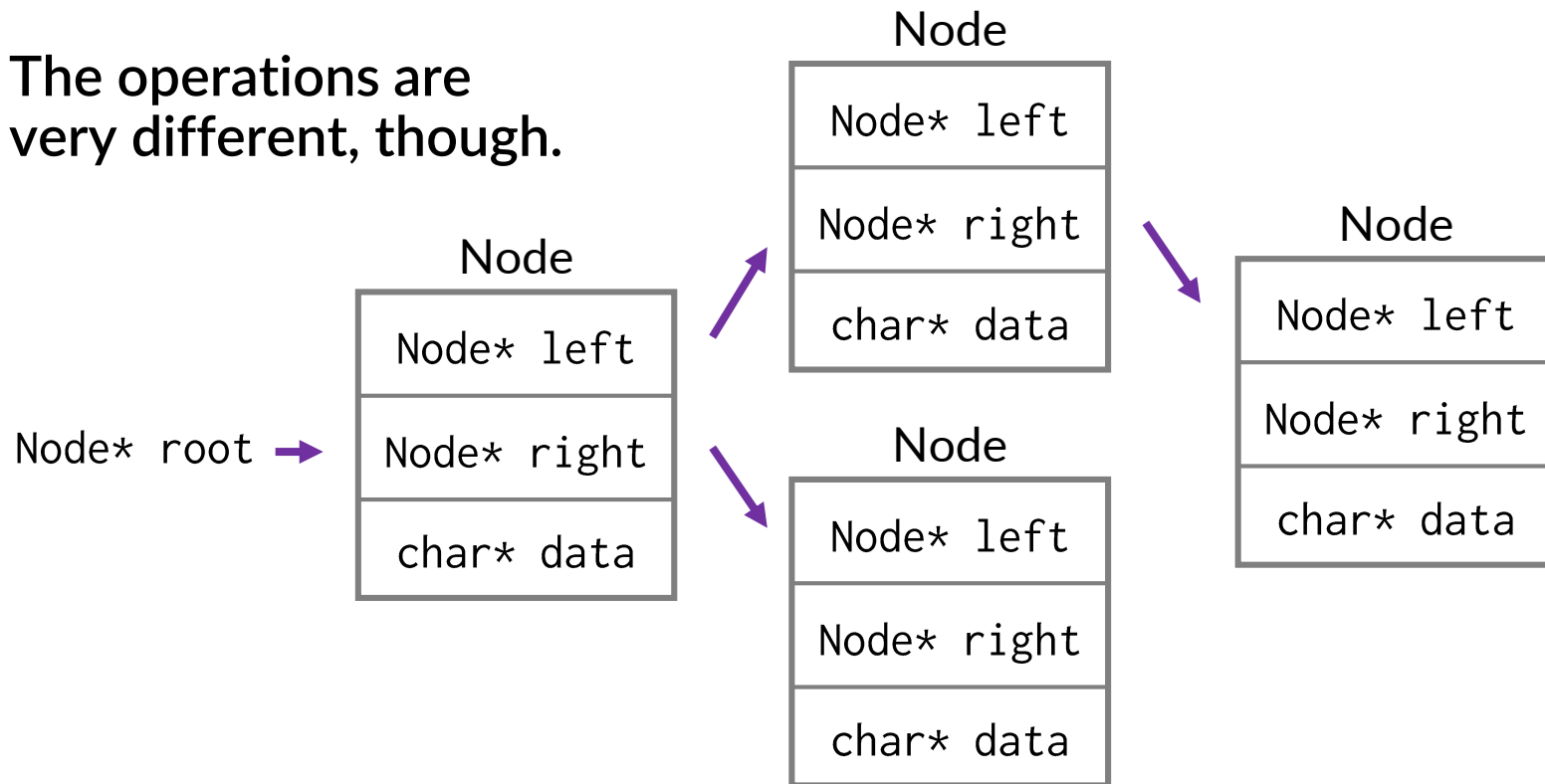
Linked lists ... link you ... to the world!

- Consider how much cleaner you can make certain operations if you tracked the previous node as well.
 - This is a **doubly linked list**.
 - This is typically “double-ended” as well: keeping track of both head and tail.



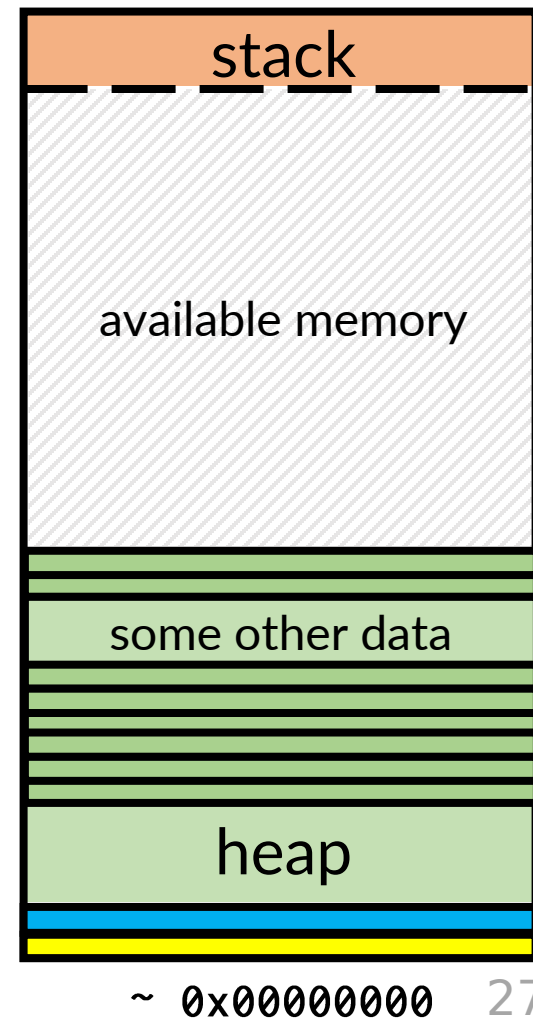
Seeing the trees through the forest

- A **binary tree** can be represented by the same nodes as a linked list.
 - In this case, you have a left and right child node instead of next and prev.
- The operations are very different, though.



De-Stressing it out: Linked Lists

- We know big arrays want to be continuous.
 - However, ensuring continuous space is difficult when you do not know how much you will ultimately need.
- Linked lists allocate very small chunks of metadata.
 - These chunks can be allocated easily on-demand.
 - And then deallocated without creating wide gaps.
- This reduces fragmentation.
 - Deallocating always leaves a small amount of room.
 - It is always the exact amount needed to append!
 - However, it is all at the expense of complexity!
 - And traversal can be expensive (but we can find ways to deal with that.)



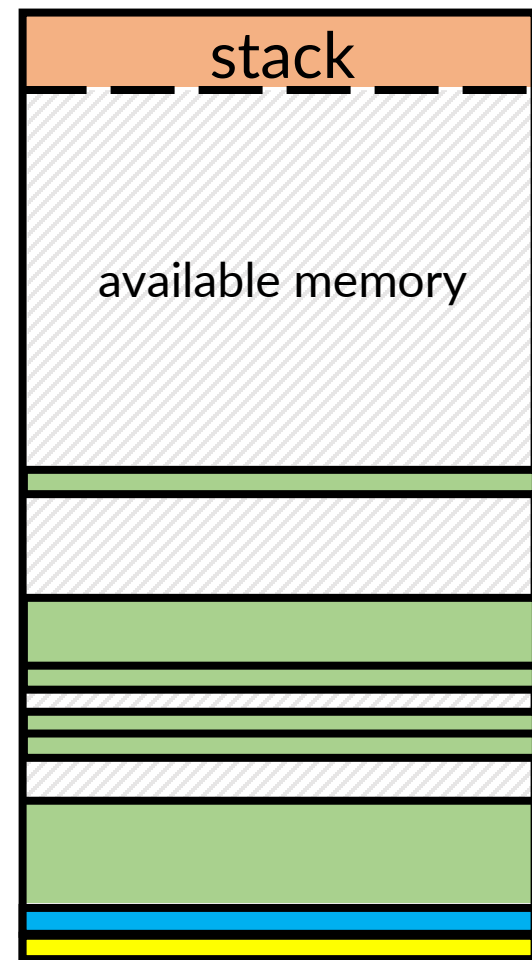
IMPLEMENTING MALLOC

It really sounds like some kind of He-Man or She-Ra villain of the week.

The malloc essentials

- The `malloc(size_t size)` function does the following:
 - Allocates memory of at least `size` bytes.
 - Returns the address to that block of memory (or `NULL` on error)
- Essentially, your program has a potentially large chunk of memory.
 - The `malloc` function tears off a piece of the chunk.
 - Also `free` must then allow that chunk to be reused.
 - The job of `malloc` is to do so in the “best” way to reduce fragmentation.

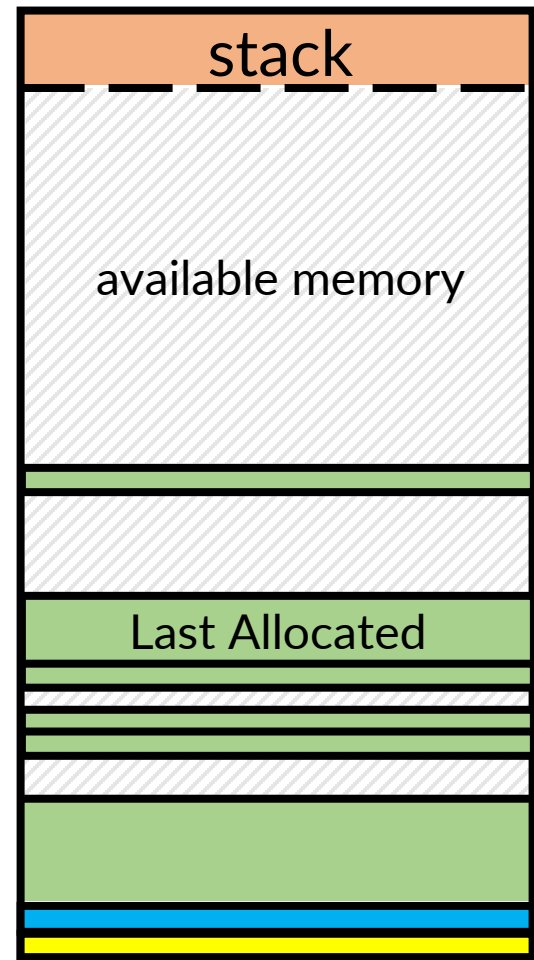
We want to avoid fragmentation ↗



Choosing where to allocate

- Our first problem is, when `malloc` is called, where do we tear off a chunk?
- We can do a few simple things:
 - **First-Fit**: start at lowest address, find first available section.
 - Fast, but small blocks clog up the works.
 - **Next-fit**: Do “First-Fit” but start where we last allocated.
 - Fast and spreads small blocks around a little better.
 - **Best-Fit**: laboriously look for the smallest available section to divide up.
 - Slow, but limits fragmentation.

malloc() → ???



~ 0x00000000 30

Managing that metadata!

- You have a whole section of memory to divide up.
- You need to keep track of what is allocated and what is free.
- One of the least complicated ways of doing so is to use... hmm...
 - A linked list! (or two!) We know how to do this!!
- We can treat each allocated block (and each empty space) as a node in a linked list.
 - Allocating memory is just appending a node to our list.
- The trick is to think about how we want to split up the nodes representing available memory.

Tracking memory: Our fresh new world.

- Let's orient our memory visually horizontally.
- Every `malloc` is responsible for allocating a block of memory.
 - We can have allocation reduce to creating a new node in a linked list.

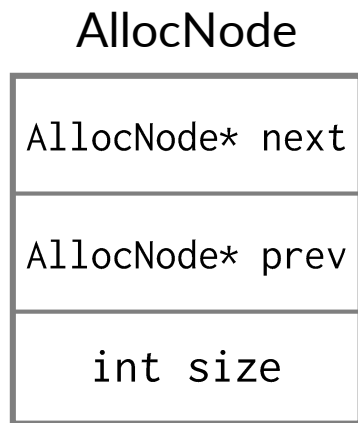


`AllocNode*`
`allocList`

Linked lists are our friend, here

- We will augment our normal doubly linked list to be useful for tracking the size of the block it represents.
- Here, we will maintain a single linked lists of all allocated or free blocks.
 - The size field denotes how big the block is (how much is used/available.)
 - We need to know when a block represents allocated space or if it is free.
 - Hmm... we could use a single bit to denote that. Or... negativity?

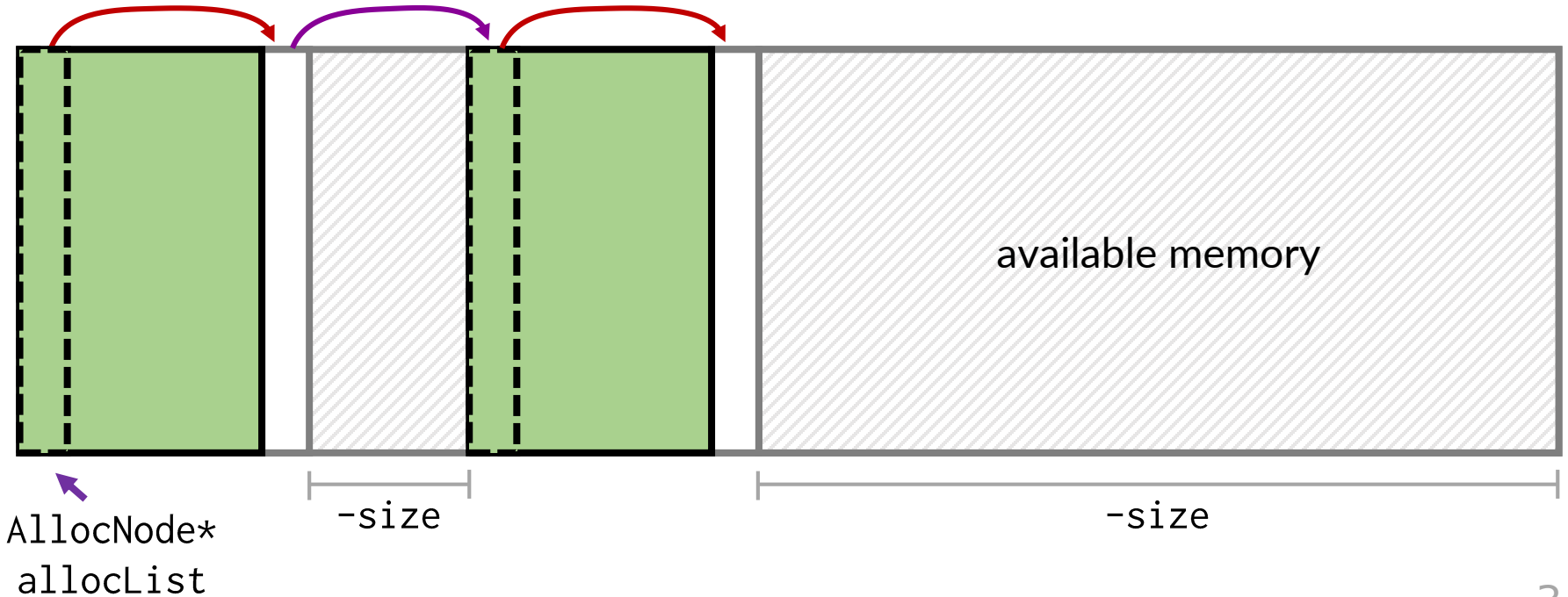
We can make other clever space optimizations, but we will start with this.



← **Negative number means a free block.**

Tracking memory: High level metadata

- We can keep track of used/empty spaces cheaply by having linked list nodes at the beginning of them. The nodes track the size of the space.



Implementing malloc

```
AllocNode* allocList;

void* malloc(size_t size) {
    int wantedSize = -(int)(size + sizeof(AllocNode));
    AllocNode* current = allocList;
    while(current &&
           current->size > wantedSize) {
        current = current->next;
    }
    if (!current)
        return NULL; // No free memory!
    // Split the block
    AllocNode* freeBlock = (AllocNode*)((char*)current + size);
    freeBlock->next = current->next;
    freeBlock->size = current->size - (int)size - sizeof(AllocNode);
    current->next = freeBlock;
    current->size = size;
    return (void*)(current + 1);
}
```

- Allocating means finding a free block big enough.
- Then splitting it into a used block and a smaller free block.
- This is incomplete. (Why?)
 - (you don't always split)

Q: This is first-fit. What should be added to implement next-fit? Best-fit?

Implementing free

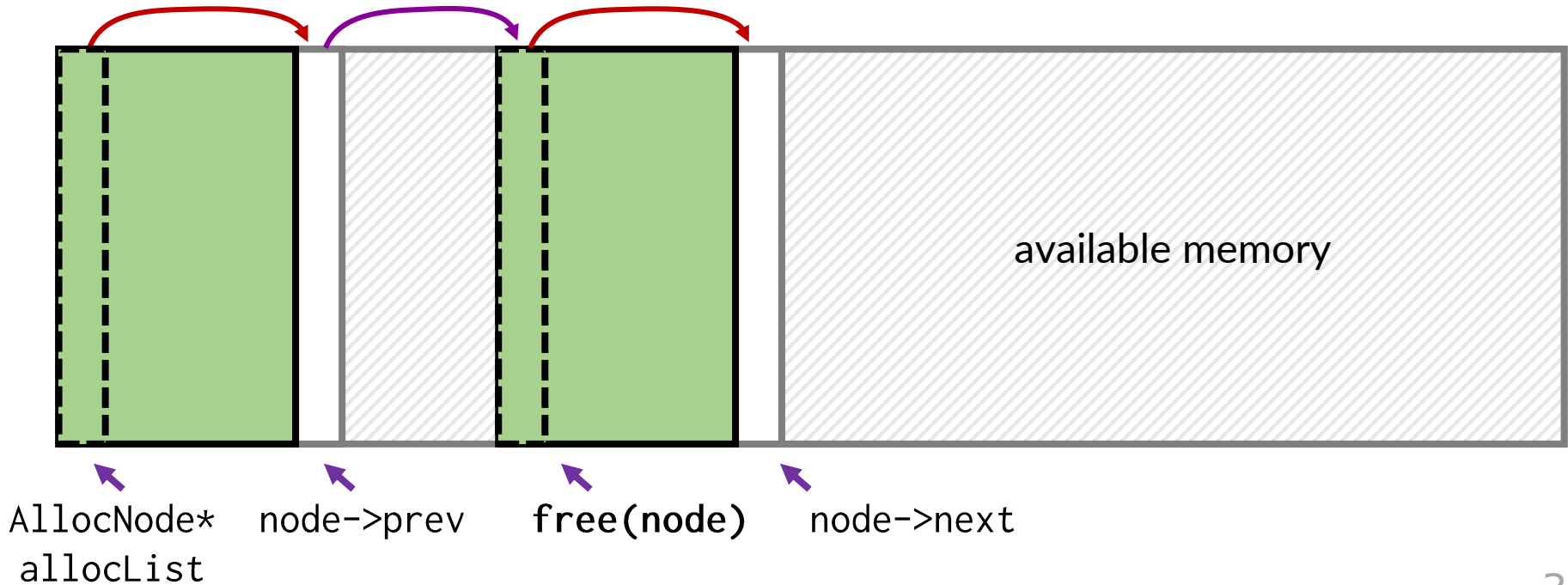
```
AllocNode* allocList;
```

```
void free(void* ptr) {  
    AllocNode* header = ((AllocNode*)ptr) - 1;  
    AllocNode* prev = header->prev;  
    AllocNode* next = header->next;  
    header->size = -header->size;  
    if (prev->size < 0) { // prev is free, coalesce  
        prev->size -= sizeof(AllocNode) + -header->size;  
        prev->next = header->next;  
        header = prev;  
    }  
    if (next->size < 0) { // next is free, coalesce  
        header->size -= sizeof(AllocNode) + -next->size;  
        header->next = next->next;  
    }  
}
```

- Where malloc splits nodes
 - free merges them.
- Whenever a block is freed next to an existing one...
 - It should merge them!
- Consider how much a doubly linked list helped.

Implementing free

- Freeing the middle block will coalesce twice:



Other thoughts

- You don't need to keep the used blocks in the list.
 - More complex to understand but removes implementation complexity.
- The idea is to only keep track of necessary metadata.
 - You only coalesce when free blocks are adjacent.
 - With a list of only free blocks, you can easily tell when that condition is met...
 - just see if `node->next` is the same address as `(char*)(node + 1) + node->size`
- The only other concern is getting from a used block you want to free to its neighboring free block.