Lecture Notes: March 1, 2017

Why dynamic data structures?

The data structures you have used most to hold *collections* of data up until now are arrays: fixed-length structures that store a set amount of data. For example, if you had a bunch of `Student` structures, the following array would let you hold a collection of 100 `Students`, say to track a class roster:

```
Student roster[100];
```

Some times you don't know how many students you need to track until the program starts (maybe you're reading the list of students in from a file, for example), in which case you can't allocate an array to store the students. So instead we use dynamic memory allocation to allocate the array at runtime. Suppose the variable `n` holds the number of students we need:

Student * roster = malloc(n * sizeof(Student));

But in both cases we have a problem: once we decide how many students we want to track, that's all we can track. What happens if we want to add a student to our class? What happens if a student drops the class? In other words, what happens if the data we want to store in our collection *changes* as the program executes? To handle these situations, we use *dynamic data structures*: structures where as we need more space to store additional data we can dynamically allocate more space, and if we ever need *less* space because we are storing less data, we can dynamically free up space.

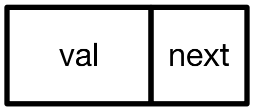
Linked List structure definition

The most basic dynamic data structure is called a *linked list*. The idea of a linked list is to create a linked set of structures: each structure will hold a piece of data (say, an integer), and a *pointer* to the *next* structure in the list, which holds the next piece of data.

To make this happen, we create a *recursive structure definition*. Pointer from *within* structure to another structure *of the same type*. We need to define this carefully so the compiler understands everything (so note that we use "struct Node" when setting the type of the next field).

```
typedef struct Node {
  int val;
  struct Node * next;
} Node;
```

We will use a graphical representation like this to display Node data structures:



We can now use this structure to create a list of integers:

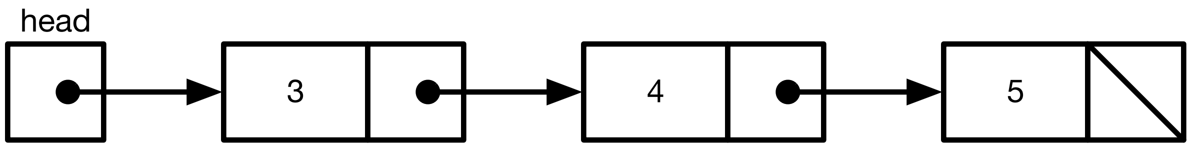
```
Node * head = malloc(sizeof(Node));
head->val = 3;

head->next = malloc(sizeof(Node));
head->next->val = 4;

head->next->next = malloc(sizeof(Node));
head->next->next->val = 5;

head->next->next->next = NULL;
```

Which creates a structure in memory that looks like this:



Note things: First, head is a `Node *` just like the `next` fields of the `Node`s are, so we use the same size box to represent it. `head` will be the pointer we use to refer to the whole linked list, since we can get to other nodes by following `next` pointers starting at `head`. Second, the `next` field of the last `Node` is `NULL`, so we represent that using a slash.

Note: most of the code we develop in these notes uses a particular style centering around using pointers to pointers. This style simplifies the code (we do not need to handle special cases of what to do when inserting or removing nodes at the beginning or end of the linked list), but looks different from the code you might find in a lot of textbooks. It's pretty easy to adapt any linked list code that just uses `Node *`s instead of `Node * *`s by calling the insert/remove/find functions that we develop here on the addresses of the next pointers.

To see the alternate way of writing linked list functions (which require special case handling of the head pointer), see the linked list functions in PA09.

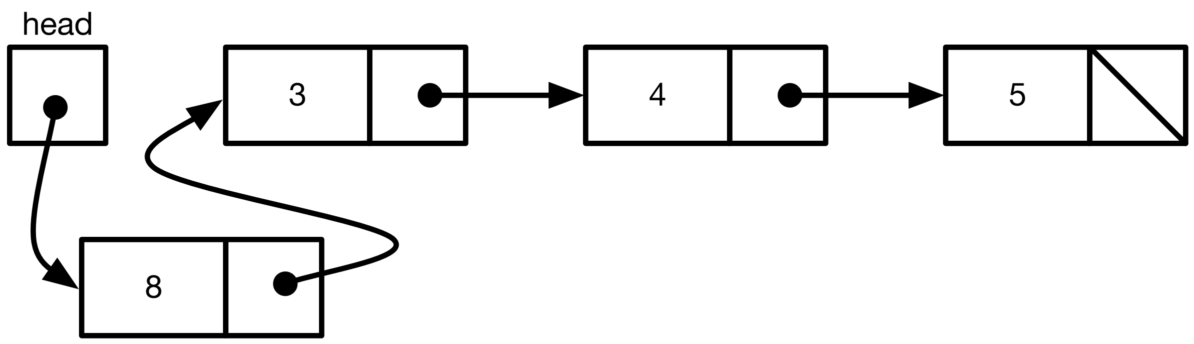
Building a linked list

Now suppose we want to add a new number to our collection. We can insert it right at the beginning of the list very easily:

```
Node * newNode = malloc(sizeof(Node));
newNode->val = 8;
newNode->next = head;
```

```
head = newNode;
```

Which creates a list that looks like this:



Note that what we did here was hook the new node up to the first node in the list using the `next` pointer, then pointed head to the new node, which gave us a new list that now contains `8`.

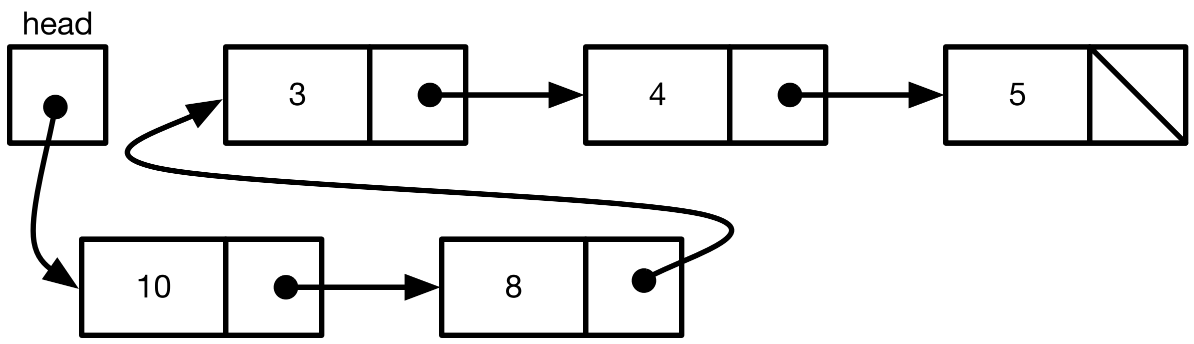
This is code that we can re-use to keep adding new data to the beginning of the linked list, so we would like to lift it out into a separate function. Note, though that because this code *changes* the value of head, we cannot just pass `head` into the function (since that would make a copy of the head pointer). Instead, we have to pass the *address* of `head` into the function, and use * to dereference that address so we can directly manipulate head:

```
void insert(Node * * loc, int val) {
  Node * newNode = malloc(sizeof(Node));
  newNode->val = val;
  newNode->next = * loc;
  * loc = newNode;
}
```

Now calling insert on head will add more nodes to the linked list:

```
insert(&head, 10);
```

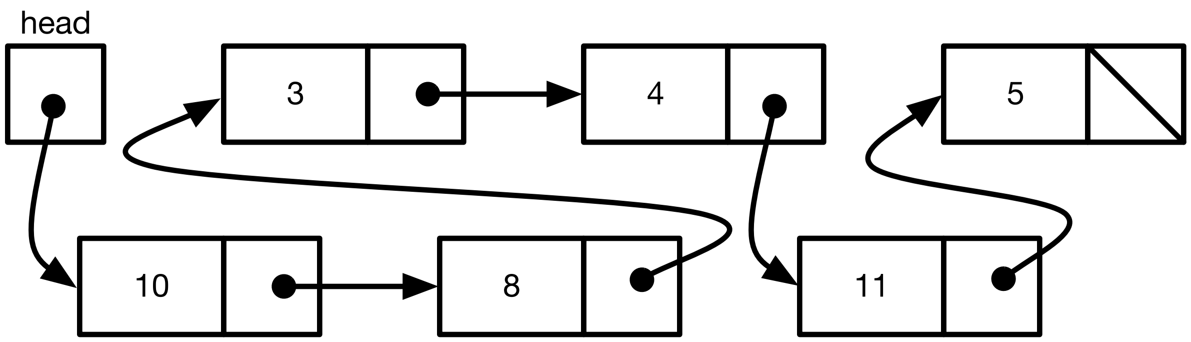
Produces:



But note something interesting about this insert function: all it expects is to be passed the address of a Node * (i.e., the address of a pointer to a Node), and it will insert a new value into the linked list to be pointed to by that pointer. So we can pass in a different pointer to insert into another location in the list. For example, we can do the following:

```
Node * cur = head->next->next->next; //points to 4
insert(&(cur->next), 11); //inserts 11 after 4.
```

Which creates the following linked list:



Removing from a linked list

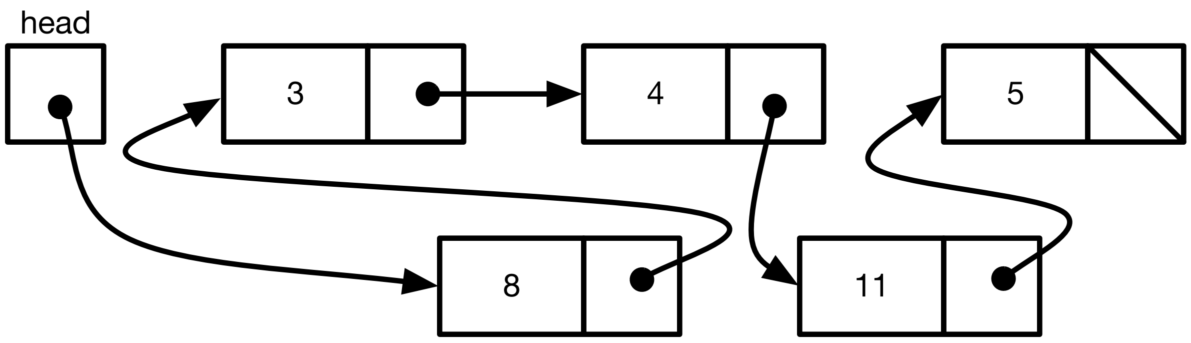
Suppose we want to remove a node from a linked list. We can do that by making head point *past* the node it currently points to:

```
head = head->next;
```

Which will make head point *past* 10 to point to 8 instead. But that causes a problem: the node holding 10 is now “floating around” — it has been allocated, but there is no way to get to it: no pointer points to it, so there is no way for us to access the variable. That’s a memory leak! We need to make sure to *free* the node we’re deleting:

```
Node * toDelete = head;
head = head->next;
free(toDelete);
```

And now we have successfully deleted 10 from the list *and* freed the memory:

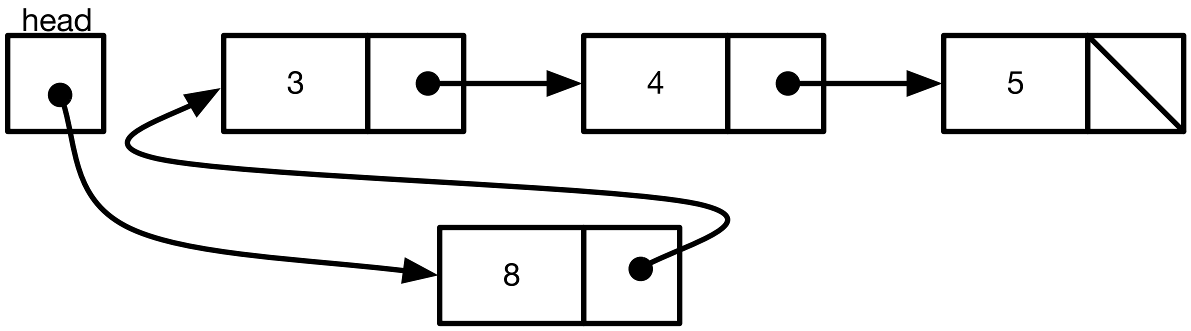


Just like before, we can lift this out into a function:

```
void remove(Node * * loc) {
  Node * toDelete = * loc;
  * loc = (* loc)->next;
  free(toDelete);
}
```

And, just like before, `remove` can now be passed *any* address of a pointer to a `Node` to remove the `Node` that pointer points to. In other words, to remove a node from the linked list, we pass `remove` the address of the next pointer that *points to* that node.

```
Node * del = head->next->next; //points to 4
remove(&(del->next)); //removes 11
```



(Note that remove doesn't check to make sure that * loc isn't NULL. If we passed it the address of 5's next pointer, we would get a segfault. The next pointer doesn't point to anything!

Searching a linked list

What if we want to find out whether a linked list contains a particular value? Here, we will use a standard trick: keeping a "cursor" pointer that points to a particular node in the lined list. We can keep moving the cursor pointer forward until we either find the value we're looking for, or get to the end of the linked list:

```
bool contains(Node * head, int key) {
  Node * cur = head;
  while (cur != NULL) {
    if (cur->val == key) return true;
    cur = cur->next;
  }
  return false;
}
```

`cur` thus steps through the linked list (the key line is `cur = cur->next`). Note that if the key *isn't* in the list (including if the list is empty), `cur` will get to the end of the list and wind up being equal to `NULL`, so we can just return `false` then.

Manipulating specific nodes in a linked list

Can we use this kind of technique to remove a specific node from the linked list? Yes! Suppose we write a slightly different version of contains that does two things:

1. If the value is in the linked list, we return the *address of the next pointer* that points to that node (so we would return the address of `head` if it's the first element in the linked list).
2. If the value is *not* in the linked list, we return the address of the *last* next pointer in the list (i.e., the next pointer that points to `NULL`).

```
Node * * findEq(Node * * loc, int key) {
  while ((* loc) != NULL) {
    if ((* loc)->val == key) return loc;
    loc = &((* loc)->next);
  }
  return loc;
}
```

Note how much this code looks like `contains` (in fact, it has a similar relationship to `contains` that `insert` and `remove` have to their non-function versions). To use this function to remove, say, 8 from the linked list, we could write:

```
Node * * toRemove = findEq(&head, 8);
if ((* toRemove) != NULL)
  remove(toRemove);
```

Note that we had to check to make sure that toRemove wasn't at the end of the list (in case we were trying to remove a number that didn't exist in the list). Note that if we lift this out into a function, that function needs to be called with the *address* of head to make sure that we're not operating on a copy of the head pointer.

Sorted linked lists

We can also use this technique to maintain sorted linked lists (where the items in a linked list always appear in a particular order). To do this, we can write a variant of findEq that instead returns a pointer that *points to the node we want to insert the new value in front of*. If we then call insert on that pointer, we will insert a new value in the right place. So, for example, if we want to keep the list in ascending order (smallest number first), we write a variant of find that returns the address of the first pointer that points to a value *larger* than the number we are looking for:

```
Node * * findGT(Node * * loc, int key) {
  while ((* loc) != NULL) {
    if ((* loc)->val > tmp) return loc;
    loc = &((* loc)->next);
  }
  return loc;
}
```

Note that if the number we are looking for is larger than any number in the list (including if the list is empty), this method does what we want: it returns the address of the *last* Node pointer in the list, which is exactly where we want to insert the new number. Now we can write our addNode function:

```
void addNode(Node * * headPtr, int val) {
  Node * * insertLoc = findGT(headPtr, val);
  insert(insertLoc, val);
}
```

Note that we operate on a `Node * *` here (i.e., we pass in the address of `head`) just in case `head` has to change. If you work through what happens when we're dealing with an empty list, you will see that `insertLoc` will be equal to `&head` in that case, and we will correctly insert the first number right at the beginning of the list.

Note, too, that as long as we *only* use addNode to add new nodes to the list, the list is guaranteed to remain sorted.