Let's go back and look at how we implemented our “find” function in a linked list:

```c
int find(int k)
{
    node *n = head;
    while (n != NULL) {
        if (n->data == k) return 1;
        else n = n->next;
    }
    return 0;
}
```

In a linked list of \( n \) elements, how many nodes would we have to look at (in the worst case) to see whether the value is in the list?

In the worst case, the answer would be \( n \): we'd have to look at every single element.

People who study data structures and algorithms ask themselves, “can we do this faster?”

And the answer in this case is, yes!

Rather than using a linked list, we will use a data structure called a **Binary Search Tree**.

A **tree** is a data structure that is similar to a linked list, except that a node can point to multiple other nodes (called its “children”) and instead of a “head” node, there is a “root” that has no parents.

A **binary tree** is a tree in which each node has at most two children.

A **binary search tree** is a binary tree in which the value in its “left” child node (if any) is less than its own value; and the value of its “right” child node (if any) is greater than its own value.

Here’s a binary search tree (BST) in which the root node is the one with value “M”. We say that “A”, “K”, and “V” are **leaf nodes** because they have no children.
Let's define a node for our BST, but using strings instead of ints:

```c
typedef struct Node node;
struct Node {
    char *value;
    node *right, *left;
};

// pointer to root of tree
node *root;
```

Now we'll see how we can search through this data structure. Rather than looking at every single node (as in a linked list), we'll start at the root.

- If the value we're looking for is equal to the value of the root, then we're done!
- If the value we're looking for is less than the value of the root, then we know that, if it's in the tree, that value would be in the root's left subtree
- If the value we're looking for is greater than the value of the root, then we know that, if it's in the tree, that value would be in the root's right subtree

Of course, we can apply this iteratively. For instance, if we're looking for K, we start with the root:

- K is less than M, so we look in M's left subtree, starting with its left child, G
• K is greater than G, so we look in G's right subtree, starting with its right child, K
• K is equal to K, so we found what we're looking for

There are a few ways to implement this but here's a pretty simple one:

```c
int find(char *target) {
  node *n = root;
  while (n != NULL) {
    int compare = strcmp(target, n->value);
    if (compare == 0) return 1; // found it!
    else if (compare < 0) {
      // this means target < n->value
      n = n->left;
    }
    else n = n->right;
  }
  // if we get here, we didn't find it
  return 0;
}
```

On line 3, we start with the root, of course.

If the root is null (line 4), then we know it's not there, so we fall through to line 14 and return 0, indicating that it wasn't found.

Otherwise, we use the `strcmp` function on line 5 to compare `target` (the thing we're looking for) and `n->value` (the value of the node we're considering).

Remember that `strcmp` returns 0 if the two strings are equal. So if that's the case on line 6, we know we found it, and we return 1.

If `strcmp` returns a negative number, that means that the first string is “less than” the second string. So we know that we need to look at the left subtree, starting with the node that is the left child (line 9); we then go back to line 4 and continue looping.

If we get to line 11, that means that `strcmp` returned a positive number, meaning that the first string is “greater than” the second string. So we know that we need to look at the right subtree, starting with the node that is the right child; we then go back to line 4 and continue looping.

If the left or right child (depending on which one we're looking at) is null, then we'll skip from line 4 down to 14, and return 0, because we got to the bottom of the tree and didn't find what we're looking for.
So now we go back to our original question, posed this way: in a BST of $n$ nodes, how many nodes would we have to look at (in the worst case) to see whether the value is in the list?

The answer is not quite as straightforward as it is for a linked list, in which the answer is clearly just $n$.

In a BST, the answer depends on whether the tree is balanced. In a balanced tree, for every node $k$, the number of nodes in $k$'s left subtree is approximately the same as the number of nodes in $k$'s right subtree (how “approximate” is up for debate).

In the picture above, the tree is balanced. The number of children in M's left subtree (3) is approximately the same as the number in its right subtree (2). The number of children in G's left subtree (1) is the same as the number in its right subtree (1). And so on.

Here is an unbalanced tree containing the same nodes:

![Unbalanced tree diagram]

This tree is unbalanced because the number of children in S's left subtree (4) is not “approximately” the same as in its right subtree (1).
In the case of the balanced tree, the answer to the question “how many nodes would we have to look at?” is approximately \(\text{ceiling}(\log_2 n)\), where the “ceiling” function just rounds the number up. So in the case above, in which \(n = 6\), we need to make \(\text{ceiling}(\log_2 6) = 3\) comparisons.

In the case of an unbalanced tree, the answer depends on the maximum depth of any node in the tree, where the depth is the number of ancestors a node has (including itself). In the example above, the node labeled “M” has a depth of 4, so that would be the worst case for all the nodes in that tree.

What would be worst case in general? That would be a “degenerate” tree, in which each node has exactly one child. That, of course, is the same as a linked list, in which case we'd have to look at all \(n\) nodes.

Clearly it's beneficial for a tree to be balanced! The balancing would happen when you add a new value to the tree. We won't look at that here, but there's a data structure called a red-black tree that is “self-balancing” and you might want to take a look at that sometime (but it's outside the scope of what we're doing here).

If we just want to add a node to the tree, and not worry about balancing, we could do something like this:

```c
void add(char *d)
{
  // create a node
  node *new_node = (node *)malloc(sizeof(node));
  new_node->value = d;
  new_node->left = NULL;
  new_node->right = NULL;

  // in case this is the first node
  if (root == NULL) {
    root = new_node;
    return;
  }

  node *n = root;

  while (1) {
    int compare = strcmp(d, n->value);
    if (compare == 0) return; // already in the list
    else if (compare < 0) {
      // try to put it in left subtree
    }
```
if (n->left == NULL) {
    // left child is null, so we're done
    n->left = new_node;
    return;
} else n = n->left;
}
else {
    // it must go in right subtree
    if (n->right == NULL) {
        // right child is null, so we're done
        n->right = new_node;
        return;
    }
    else n = n->right;
}
}
Hashtables

Recall that, in a linked list of $n$ elements, it took $n$ comparisons (in the worst case) to find an element.

And for a binary search tree, we were able to get that down to around $\log_2 n$, assuming the tree was balanced.

Could we do even better than that? Can we get it down to a constant number of comparisons, independent of the value of $n$?

Well, if the answer were “no”, then we'd be done, right? =)

The data structure that allows you to find an element in a constant number of steps – under certain circumstances – is called a **hashtable**. Because it is so fast, it is typically used in a variety of applications, and is very likely to come up in some form on some job interview in the future.

A hashtable can be thought of as an array of linked lists. Each value in the hashtable has an associated **hash code** that is used to determine which linked list (or “bucket”) it belongs in. The idea is that calculating the hash code can be done quickly (in constant time, i.e. independent of the number of elements in the hashtable) and if we're able to ensure that each bucket only has one element, then we'll know that we can find an element with just one comparison.

The hash code is a result of a **hashing function** which takes the value and converts it to an integer. If the value already is an int, then obviously the hashing function is trivial. However, if the value is a string, or a struct, or something like that, then we need a more complex function.

The hashing function must ensure that, for a given value, each time we calculate its hash code, we get the same result (i.e., the hash code cannot be random). The hashing function should also ensure (but doesn't have to) that different values produce different hash codes.

For instance, if we were to calculate the hash code of a string by simply adding up the ASCII values of the characters, then “god” and “dog” would both have the same hash code (specifically, 314). So would “ddr” and “axa” and lots of other strings. That's not necessarily a terrible thing, but it should be avoided, because we want each bucket to only have one value: if two different values have the same hash code, they'd end up in the same bucket.
As a side note, the hashing function that Java uses for Strings is described at http://en.wikipedia.org/wiki/Java_hashCode

As described above, the hash code is used to determine which bucket the value belongs in. So, using the hash codes above, does that mean there are at least 314 buckets?

No, not at all. Rather, the hashtable has a set number of buckets, which is known as the “size” of the hashtable. To determine the bucket that a value should go into, you calculate hashcode % size, and that gives you the bucket number.

Note that, even if all values have different hash codes, they could still end up in the same bucket. For instance, if “cat” had a hash code of 882, and “dog” had a hash code of 3082, and the size of the hashtable was 200, both would end up in bucket #82.

Coming up with good hashing functions and table sizes is a big area of research in Computer Science and is way beyond the scope of this class. For now, we're just interested in how a hashtable works, but it's good to know about these issues.

Let's say we're going to have a hashtable of strings. Here's the definition of a node in our linked list:

```c
typedef struct Node node;
struct Node {
    char *value;
    node *next;
};
```

Now we can define a hashtable as an array of pointers to nodes, i.e. the heads of each of the linked lists:

```c
node *list[SIZE];
```

where SIZE is the size of the hashtable, i.e. the number of buckets (or, the number of linked lists).

To add a new value to the hashtable:
- Calculate its hash code
- Determine which bucket it should be in
- Add it to the linked list for that bucket

Here's what that function may look like:
```c
void add(char *v) {
    // 1. get the hash code
    int hashcode = hash(v);

    // 2. determine the bucket number
    int index = hashcode % SIZE;

    // 3. add to the linked list
    node *head = list[index];
    node *new = malloc(sizeof(node));
    // populate the value field
    new->value = malloc(strlen(v) + 1);
    strcpy(new->value, v);
    // put it at the front of the list
    new->next = head;
    list[index] = new;
}
```

On line 4, we calculate the hash code for this string. Note that, in C, there is no built-in “hash” function, so you'd have to implement that yourself.

On line 7, we figure out which bucket this thing belongs in.

On line 10, we use the index (bucket number) to get the appropriate linked list head out of the array called list. From here, it's just a straightforward “adding to a linked list” exercise (e.g., on line 12 we use malloc to create space for a new node).

However.... lines 15 and 16 are slightly different from what we've seen before. On line 15, we allocate some new space for the string, plus one additional byte for the null-termination. Then, on line 16, we copy from v to that newly allocated space.

Why go through all that trouble? Why not just do this?

```c
    new->value = v;
```

The reason is that we don't know where v points to. If it is a pointer to an immutable string, then that's fine, but if v points to something that can change, and we put that address in new->value, that could cause problems.

For instance, let's say the code that calls this function looks like this:
char input[20];
scanf("%s", input);
add(input);
scanf("%s", input);

Let's say that, the first time (line 3), the user enters “cat”. That means that, on the stack of this function, input points to the letter 'c', followed by 'a' and 't', then null, then 16 spaces of we-don't-know-what. When we call add (line 4), the address of the letter 'c' is passed as the argument, and if we just have new->value = v, then the address of the letter 'c' will be held in the value field of that node.

Which is fine... until we get to line 6. If the user enters “dog”, then that gets written to input, and “cat” gets overwritten on the stack of this function. The problem is, the node we just created has its value field pointing to the first character of input, which has now changed. That's bad.

Note that if you don't use malloc on line 15 of the add function, you'd probably get a segfault because you don't know where new->value is pointing. You need to allocate some space to hold the string.

Note also that you didn't have to worry about this in Homework #7. Why not? Because in the skeleton code we gave you, the node contains a 10-character array, not a pointer. That is, the node already has 10 spaces available for characters; you don't need to allocate any more.

Now that we’ve seen how to put an element into the hashtable, let’s see how we would look through the hashtable to search for a particular value. The steps are:

• Calculate that value’s hash code
• Use the hash code to determine which bucket the value would be in (assuming it’s in the hashtable at all)
• Look through the linked list for that bucket
For example:

```c
int find(char *v) {
  // 1. get the hash code
  int hashcode = hash(v);

  // 2. determine the bucket number
  int index = hashcode % SIZE;

  // 3. look through the linked list
  // starting with the head
  node *n = list[index];

  while (n != NULL) {
    if (strcmp(n->value, v) == 0) {
      // found it!
      return 1;
    }
    else n = n->next;
  }

  // if we made it here, we didn’t find it
  return 0;
}
```

So, now back to the big question: how many comparisons do you have to make in order to find a value if there are $n$ elements in the data structure?

In a linked list, the answer was $n$; for a binary search tree, it was around $\log_2 n$.

What about for a hashtable? If we’re lucky, and each bucket has zero or one elements, then we can find the value in 1 comparison, regardless of the value of $n$. This is the best we could hope for, of course.

But if we’re unlucky, and all elements ended up in the same bucket, then the hashtable is basically just a linked list, and we’d have to look at all $n$ elements. That would be bad.

As mentioned above, there is a lot of research into figuring out good hashing functions and good hashtable sizes, but that’s kind of outside the scope of what we’re doing here.

One last thing: if a hashtable can give really fast lookup times, then why would you every use any other data structure?
Keep in mind that, a data structure may be used for other things besides adding and finding elements. You may, for instance, want to maintain the order in which the elements were added; a linked list would do this, but a hashtable would not. Also, you may want to allow access based on some index, e.g. an element’s “place” in the data structure; again, you could do this with a linked list, but not a hashtable. Last, you may want to sort the elements; a binary search tree would be good for this (why? think about it), whereas a hashtable would not help with that.

So before you go throwing a hashtable at every data structure problem you have, think about the functionality you need, and then choose the right solution.