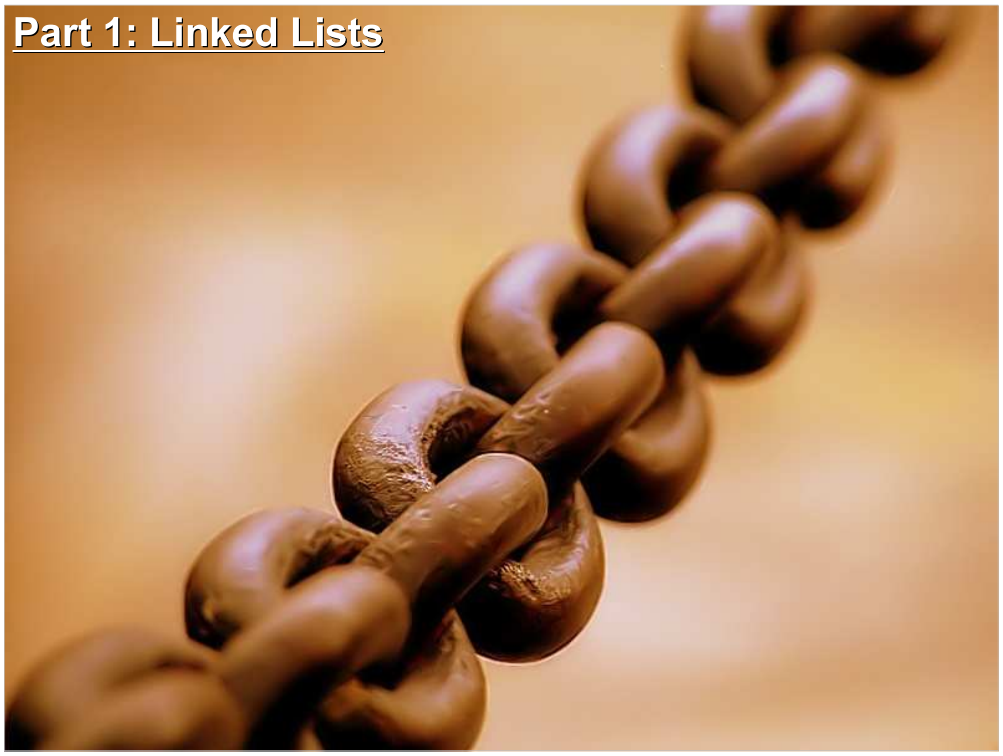
Physics 2660

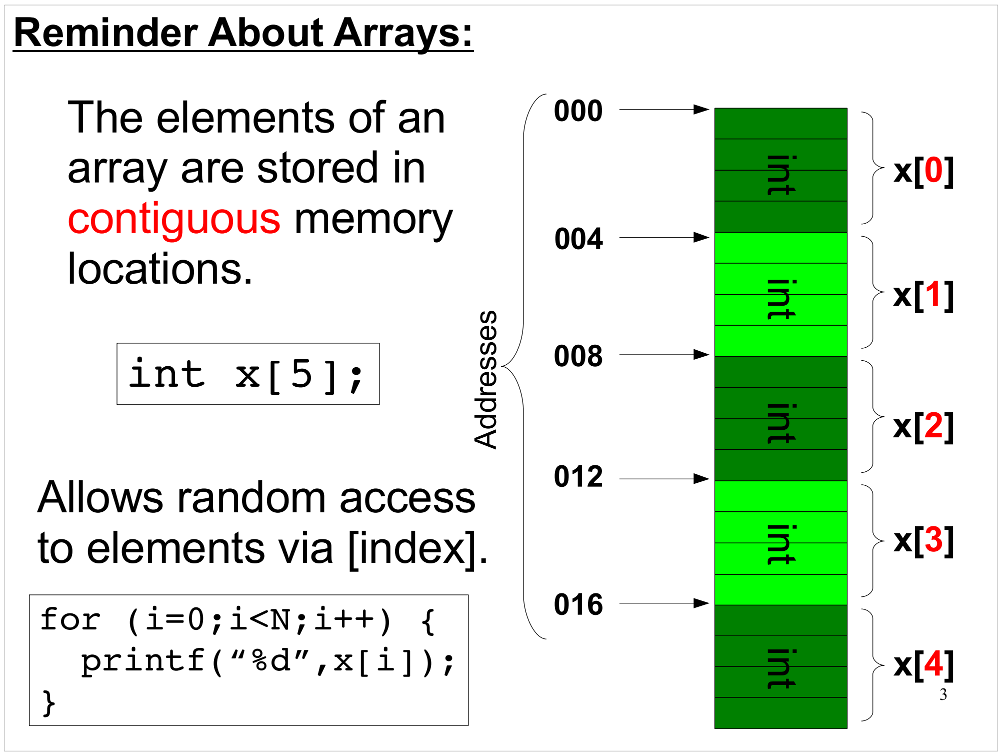
Lecture 12

Today
- **Data Structures**

Part 1: Linked Lists



Data structures are ways of organizing your data in memory.



Reminder About Arrays:

The elements of an array are stored in contiguous memory locations.

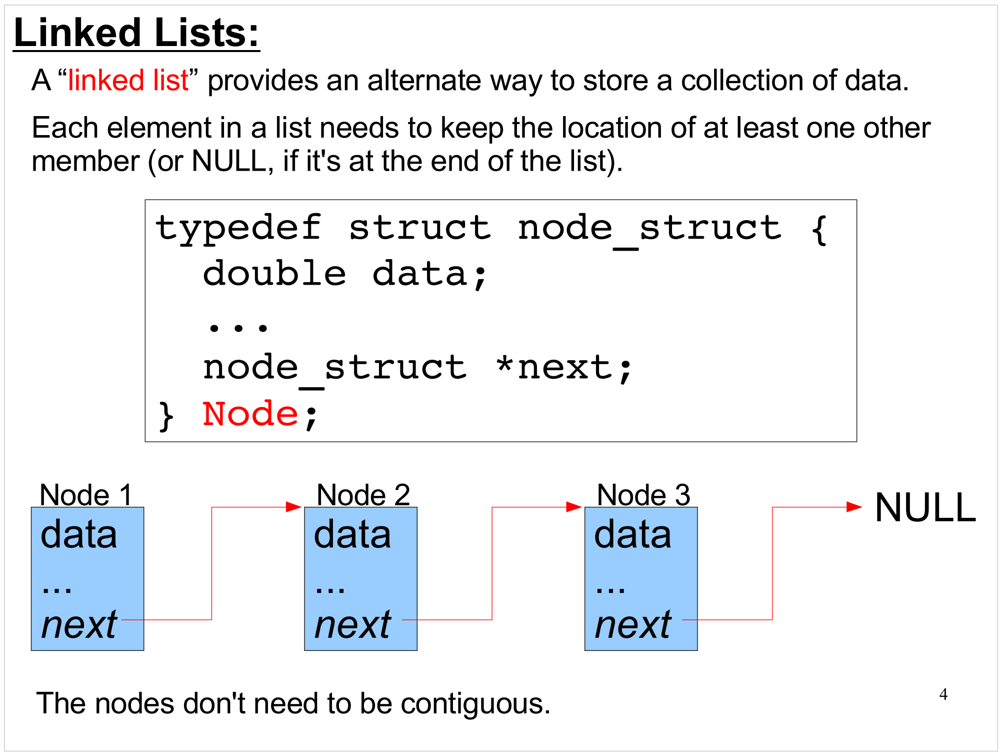
```
int x[5];
```

Allows random access to elements via [index].

```
for (i=0;i<N;i++) {
  printf("%d",x[i]);
}
```

Arrays are common data structures that we've used quite a lot in this course.

When you give the program an array index, the program multipies the index times the size of each element to find the address where a particular element lives. (This is possible because all of the array elements are contiguous.) Then the program jumps directly to that address and retreives the data.

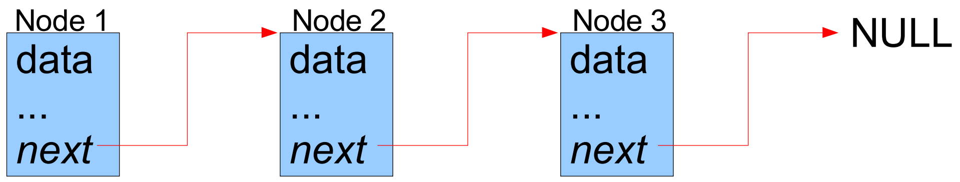


Linked Lists:

A “linked list” provides an alternate way to store a collection of data.

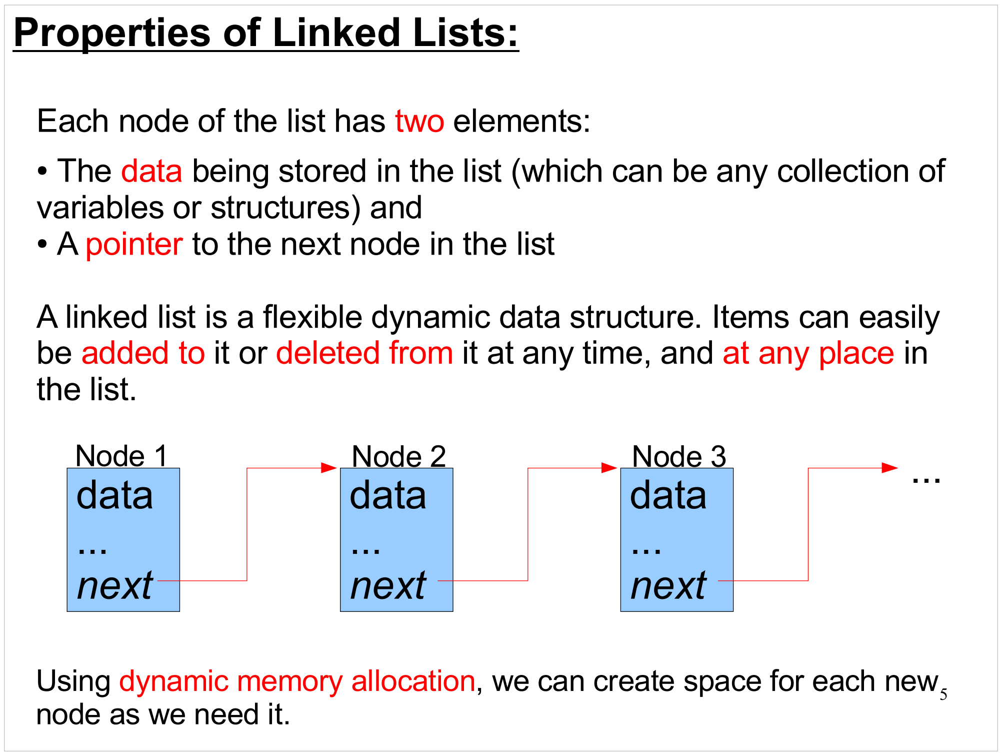
Each element in a list needs to keep the location of at least one other member (or NULL, if it's at the end of the list).

```
typedef struct node_struct {
  double data;
  ...
  node_struct *next;
} Node;
```



The nodes don't need to be contiguous.

Linked lists are another kind of data structure.

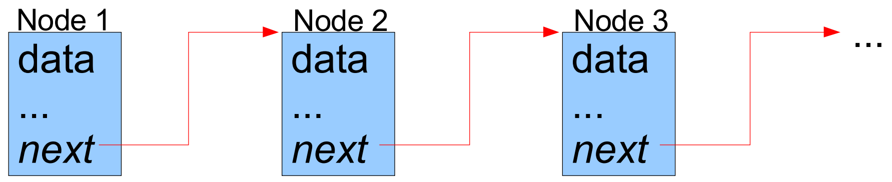


Properties of Linked Lists:

Each node of the list has two elements:

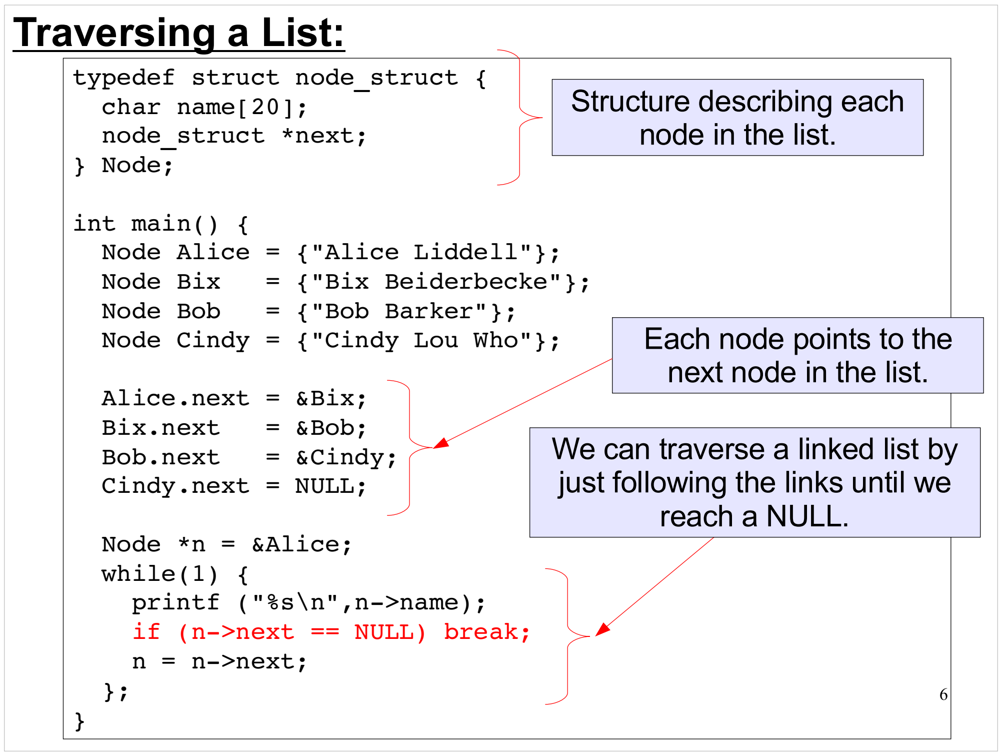
• The data being stored in the list (which can be any collection of variables or structures) and
• A pointer to the next node in the list

A linked list is a flexible dynamic data structure. Items can easily be added to it or deleted from it at any time, and at any place in the list.



Using dynamic memory allocation, we can create space for each new node as we need it.

“Dynamic memory allocation” refers to the tools like malloc, calloc and realloc that we talked about last time. They let your program allocate new space in memory whenever it's needed.



Traversing a List:

```
typedef struct node_struct {
  char name[20];
  node_struct *next;
} Node;

int main() {
  Node Alice = {"Alice Liddell"};
  Node Bix   = {"Bix Beiderbecke"};
  Node Bob   = {"Bob Barker"};
  Node Cindy = {"Cindy Lou Who"};

  Alice.next = &Bix;
  Bix.next   = &Bob;
  Bob.next   = &Cindy;
  Cindy.next = NULL;

  Node *n = &Alice;
  while(1) {
    printf ("%s\n",n->name);
    if (n->next == NULL) break;
    n = n->next;
  };
}
```

Structure describing each node in the list.

Each node points to the next node in the list.

We can traverse a linked list by just following the links until we reach a NULL.

6

Here's a simple program that uses a linked list. The nodes of the list all have the structure at the top. The nodes of a linked list can store any data we want to put there. In this case, we're just storing one thing, a person's name.

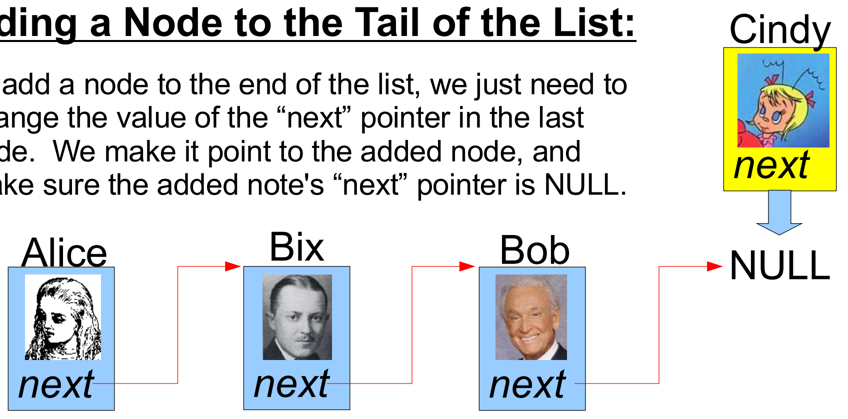
We create four nodes, and initialize them by storing a name string (some data) in each.

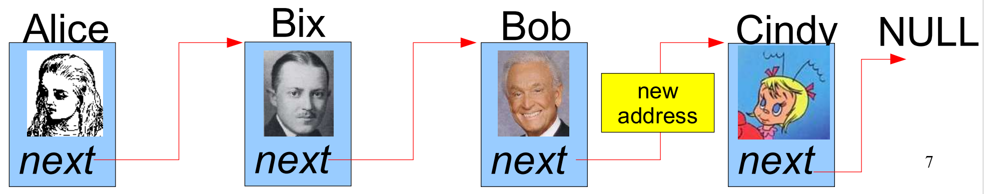
Then we set the nodes' “next” pointers so that Alice points to Bix, which points to Bob, which points to Cindy. Cindy's “next” pointer is NULL, indicating that this node is the end of the list.

Then we just start out with Alice and follow the links through the list, printing out the names.

Adding a Node to the Tail of the List:

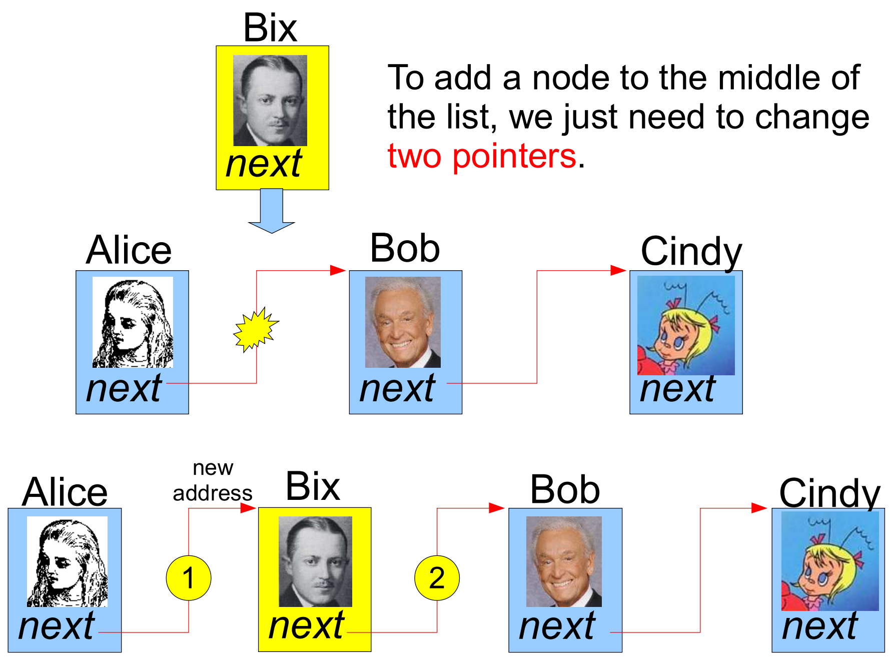
To add a node to the end of the list, we just need to change the value of the "next" pointer in the last node. We make it point to the added node, and make sure the added note's "next" pointer is NULL.





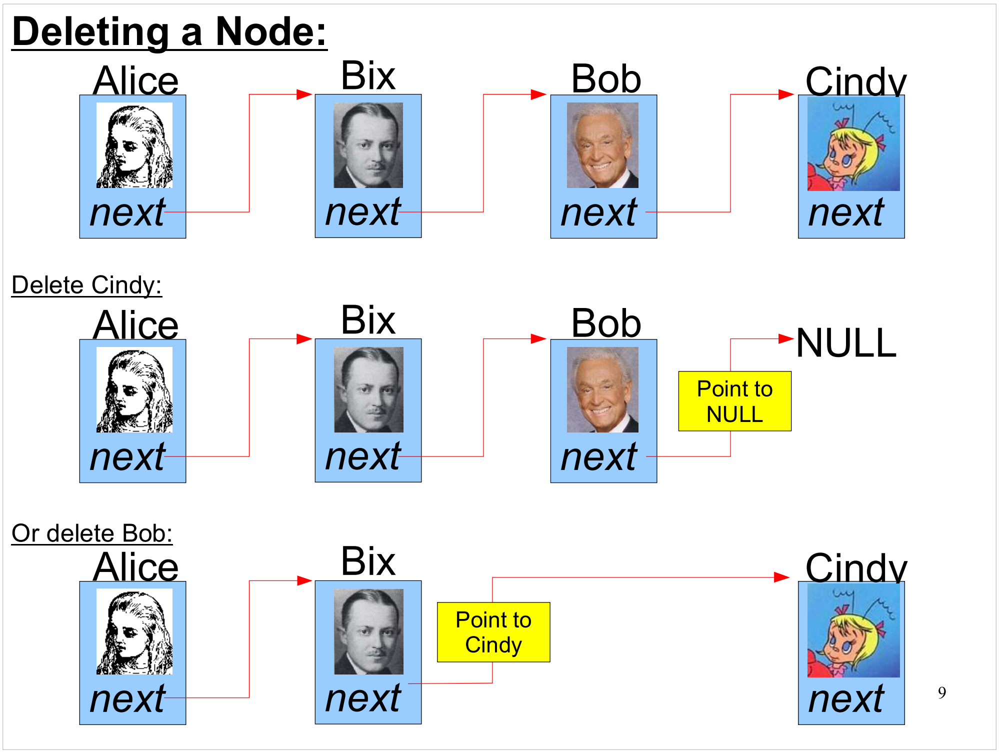
Adding a Node in the Middle:

To add a node to the middle of the list, we just need to change two pointers.



To add an element in the middle of an array, we'd need to move all the later elements down by one slot. With a large array, this could be a lot of work.

But with a singly-linked list like this one, we only need to change two pointers. The original pointer from Alice to Bob is changed so that it now points to Bix. Then the pointer from Bix is changed so that it points to Bob.

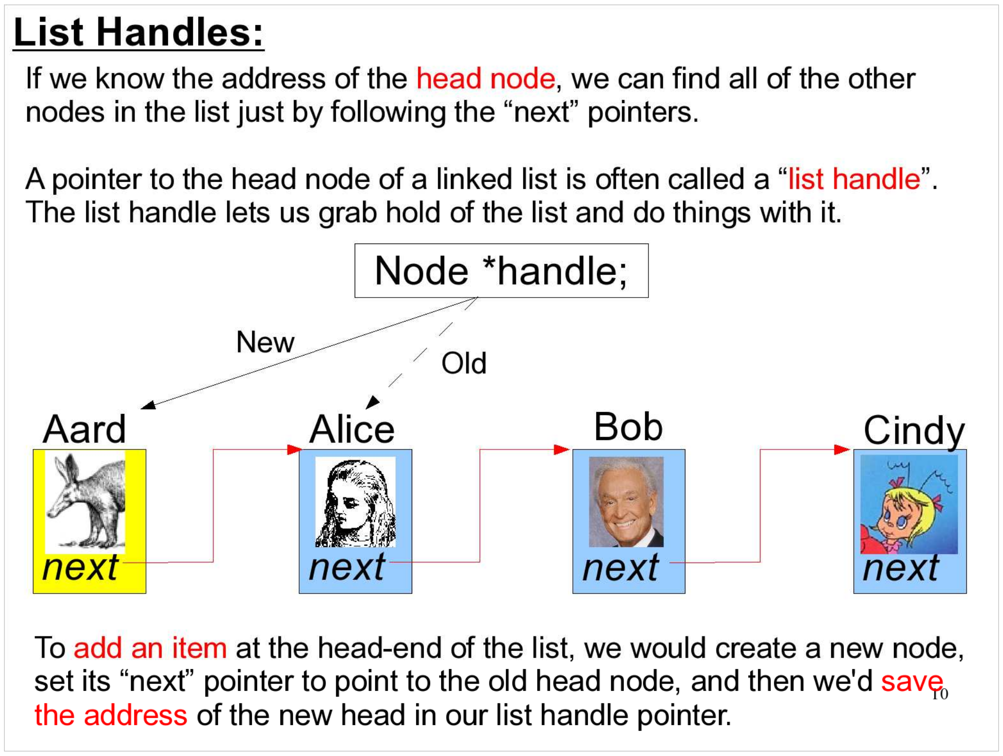


Deleting a Node:

Delete Cindy:

Or delete Bob:

Deleting a node just requires changing one pointer.



List Handles:

If we know the address of the head node, we can find all of the other nodes in the list just by following the “next” pointers.

A pointer to the head node of a linked list is often called a “list handle”. The list handle lets us grab hold of the list and do things with it.

Node *handle;
New
Old
Aard
Alice
Bob
Cindy
next
next
next
next

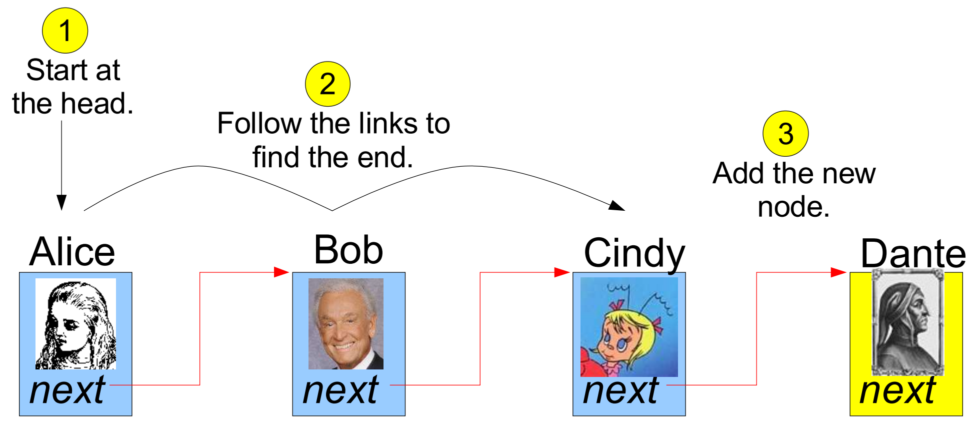
To add an item at the head-end of the list, we would create a new node, set its “next” pointer to point to the old head node, and then we'd save the address of the new head in our list handle pointer.

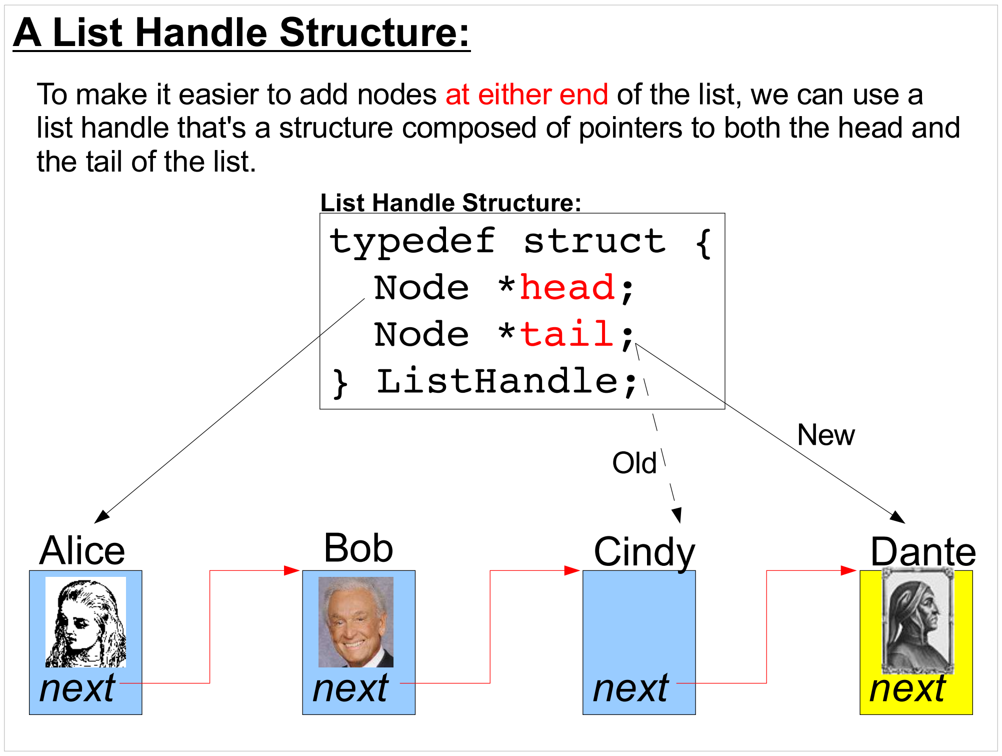
In the preceding examples, Alice has been the “head” node. This is the node we start at when we begin traversing the list.

When using an array, we refer to it by its name. When we're using a linked list, the list doesn't have a name of its own. Instead, we just store the address of the list's head node in a pointer. This is the “list handle”.

Finding the Tail of the List:

Adding a new node at the tail of the list is straightforward, but it may be slow, since we have to follow the links from the head to find where the tail is. This might take a long time if there are many nodes in the list.





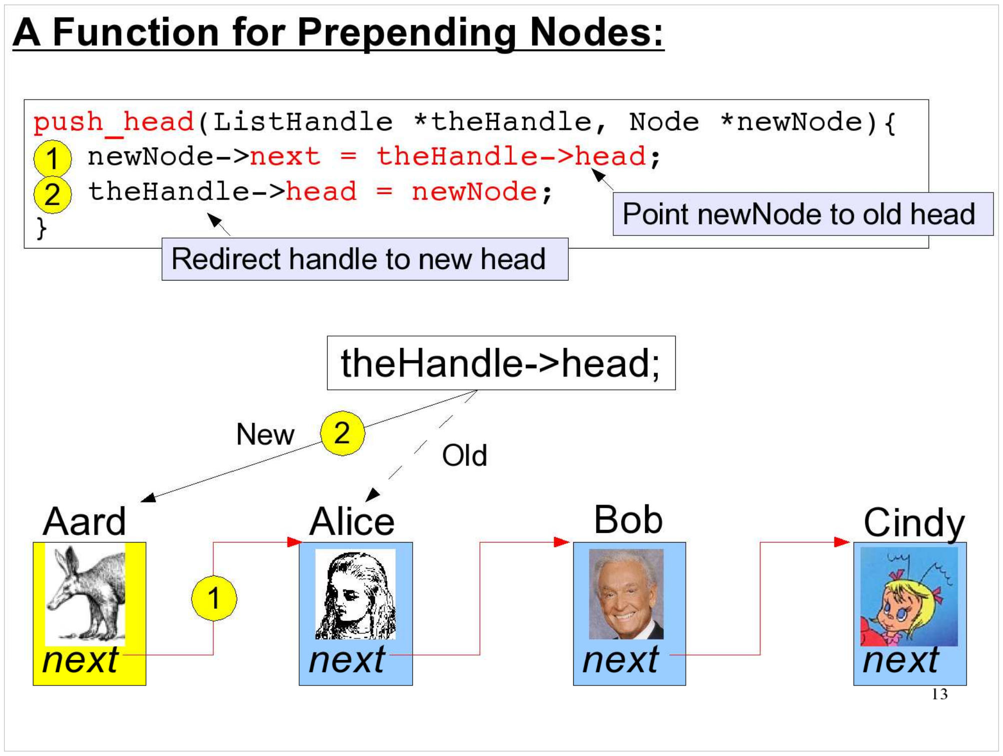
A List Handle Structure:

To make it easier to add nodes at either end of the list, we can use a list handle that's a structure composed of pointers to both the head and the tail of the list.

List Handle Structure:

```
typedef struct {
   Node *head;
   Node *tail;
} ListHandle;
```

Again, the list handle is just the equivalent of an array name. The list handle identifies the list, and holds the information we need to have in order to use the list.

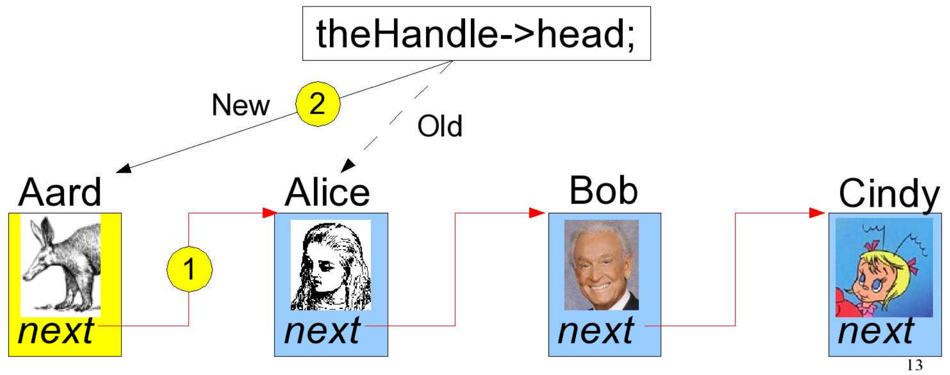


A Function for Prepending Nodes:

```
push_head(ListHandle *theHandle, Node *newNode){
1 newNode->next = theHandle->head;
2 theHandle->head = newNode;
}
```

Point newNode to old head

Redirect handle to new head



13

When talking about linked lists, we use the terms "push" and "pop" to mean "add an item to the list" and "remove an item from the list", respectively.

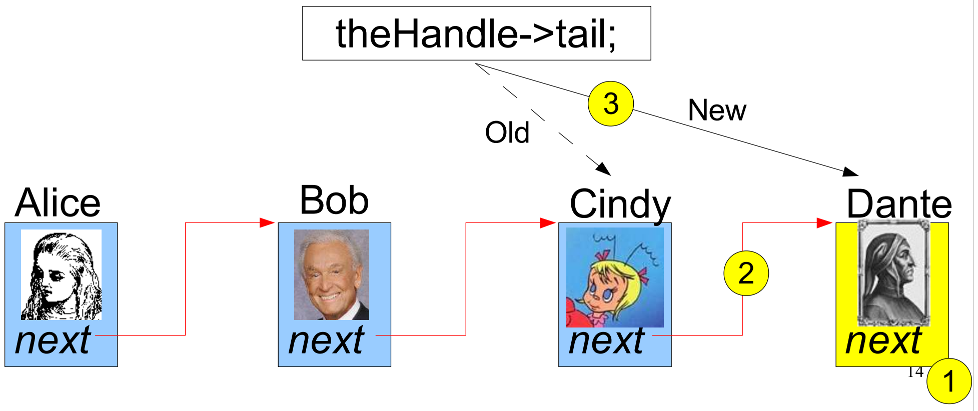
A Function for Appending Nodes:

```
push_tail(ListHandle *theHandle, Node *newNode){
1 newNode->next = NULL;
2 theHandle->tail->next = newNode;
3 theHandle->tail = newNode;
}
```

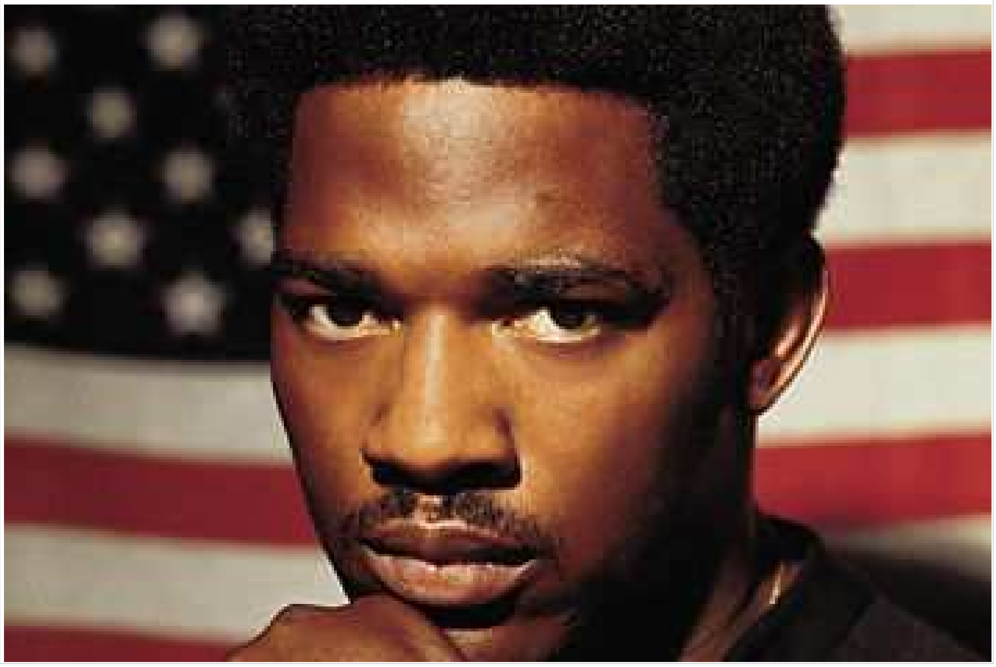
Point newNode to NULL

Point tail to newNode.

Update list handle.



Part 2: What is it Good For?

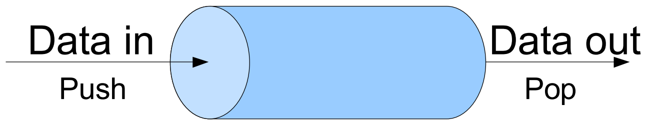


The answer isn't “absolutely nothing”. In fact, linked lists are quite useful. They're building blocks that can be used to construct many helpful data structures.

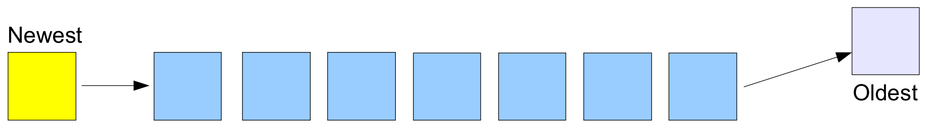
Queues:

Often, we want some data to stand in line and wait its turn to be processed. Consider a stream of audio data sent to a computer's sound card, or a stream of network traffic travelling through a router.

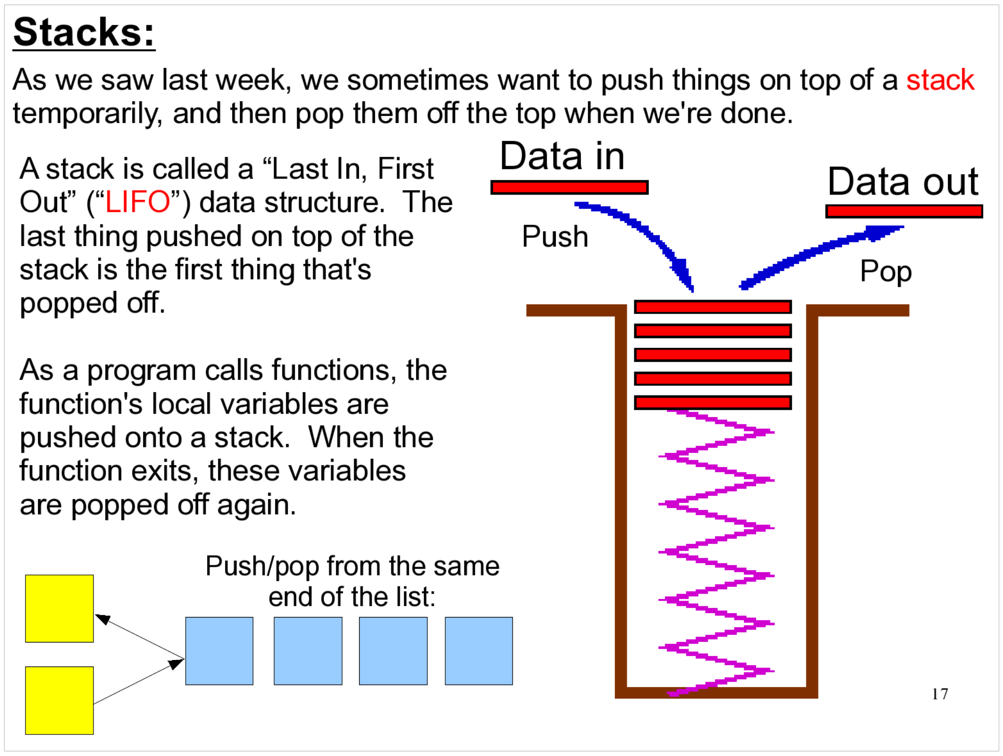
Until the data can be processed, it waits in a “queue” (also called a “buffer”). New data comes in at one side of the queue, and the oldest data comes out on the other.



A queue is called a “First In, First Out” (“FIFO”) structure. A queue is like a pipeline. Data is pushed into one end, and eventually pops out the other.



You can see that it would be easy to use a linked list as a queue.



Stacks:

As we saw last week, we sometimes want to push things on top of a stack temporarily, and then pop them off the top when we're done.

A stack is called a "Last In, First Out" ("LIFO") data structure. The last thing pushed on top of the stack is the first thing that's popped off.

As a program calls functions, the function's local variables are pushed onto a stack. When the function exits, these variables are popped off again.

Push/pop from the same end of the list:

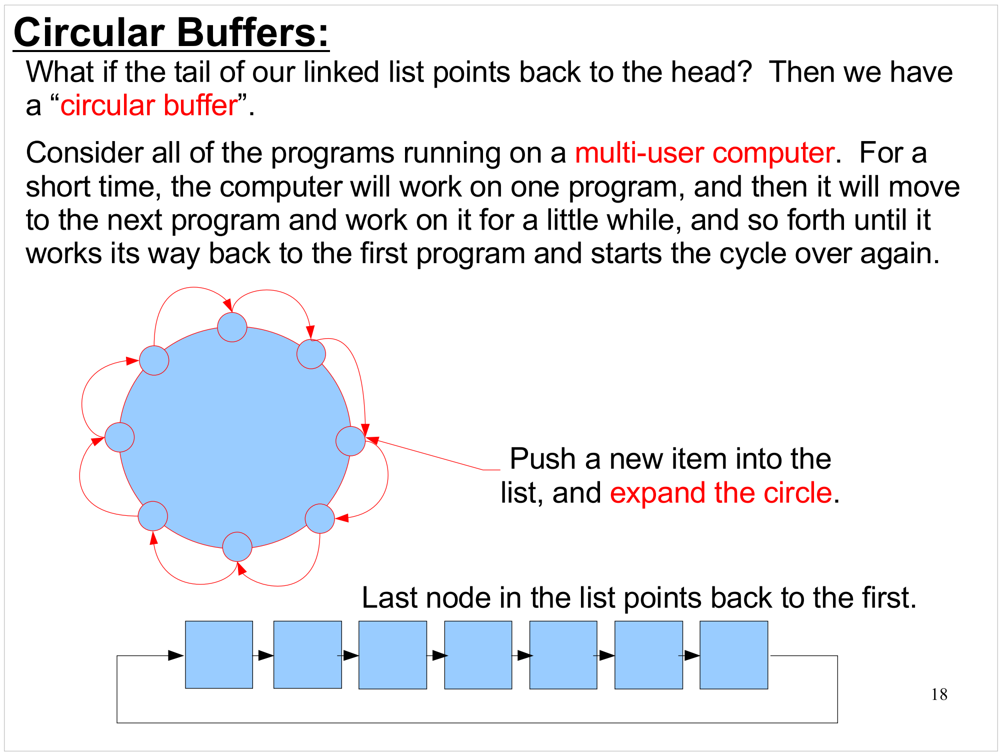
Data in

Push

Data out

Pop

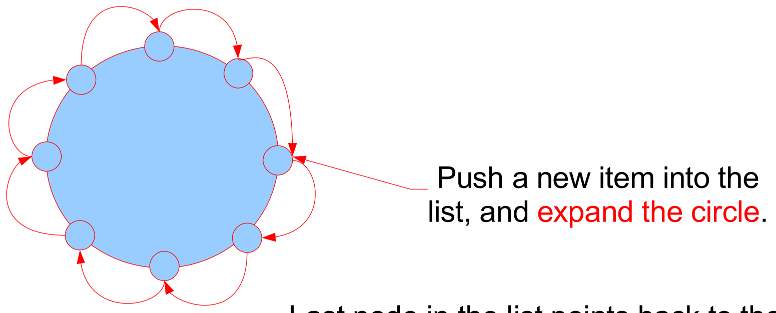
A stack is just analogous to a PEZ dispenser.

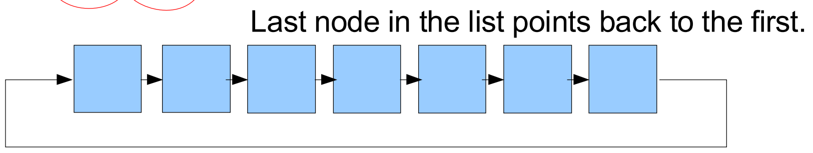


Circular Buffers:

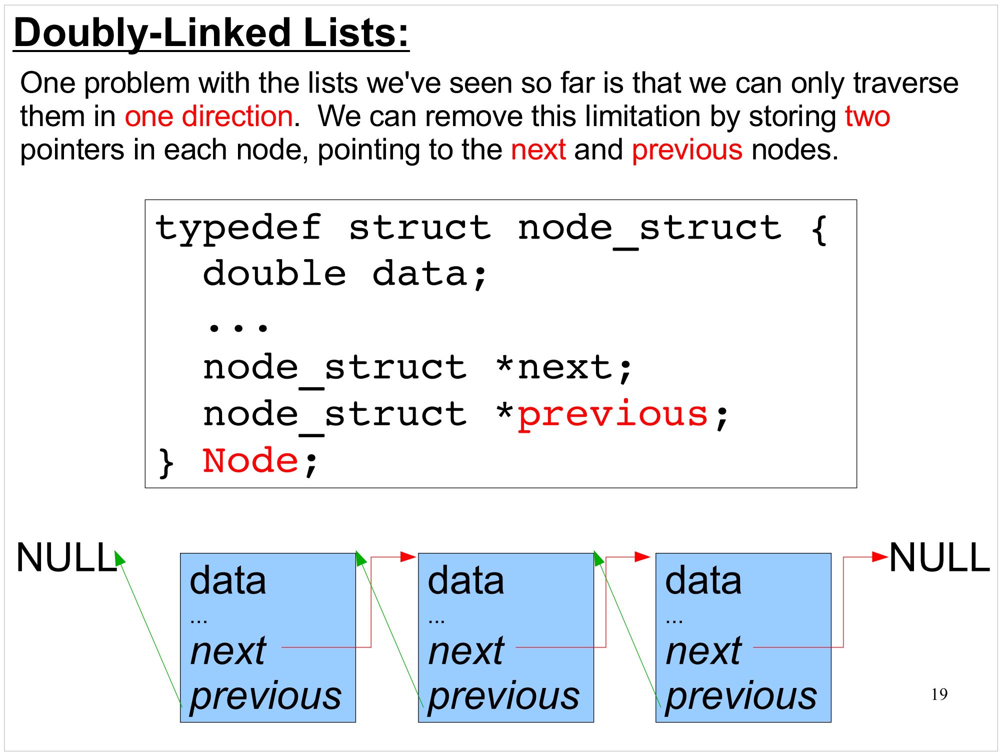
What if the tail of our linked list points back to the head? Then we have a “circular buffer”.

Consider all of the programs running on a multi-user computer. For a short time, the computer will work on one program, and then it will move to the next program and work on it for a little while, and so forth until it works its way back to the first program and starts the cycle over again.





A circular buffer is also useful for holding the last “n” things that happened. For example, we might want to keep a running list of the last n results from our program. We might use this as part of a running average calculation.



Doubly-Linked Lists:

One problem with the lists we've seen so far is that we can only traverse them in one direction. We can remove this limitation by storing two pointers in each node, pointing to the next and previous nodes.

```
typedef struct node_struct {
  double data;
  ...
  node_struct *next;
  node_struct *previous;
} Node;
```

NULL
data
...
next
previous
data
...
next
previous
data
...
next
previous
NULL

19

The “previous” pointer of the head node points to NULL.

Lists versus Arrays:

Whether single or doubly linked, a list is limited to sequential access only. An element can only be found by navigating from its neighbor.

The flexibility of adding data to the list must be weighed against the lack of simple random access to any element.

The answer should be clear based on the use of the data, i.e. will we be doing some sort of sequential processing? Or will we be randomly retrieving records from storage?

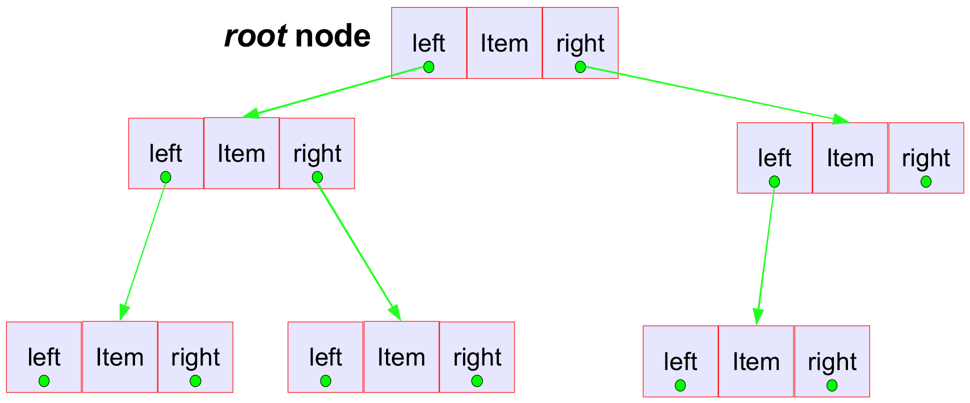
Part 3: Trees

With a doubly-linked list, we use nodes that have two pointers. What if we took these same building blocks and connected them in a different way? We could build trees out of them, for example.

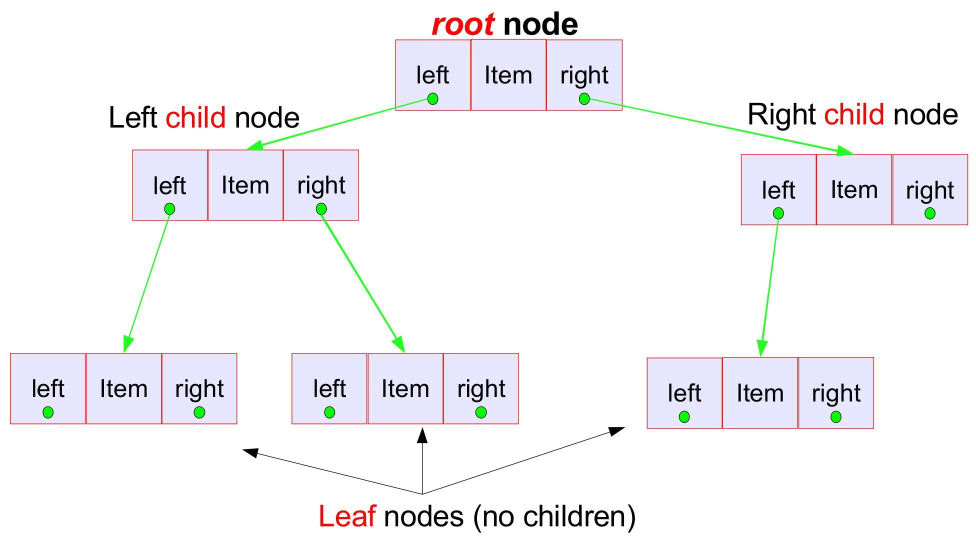
A Binary Tree:

We can create a doubly linked list out of nodes with two links, but we could alternatively use those nodes to make a tree structure.

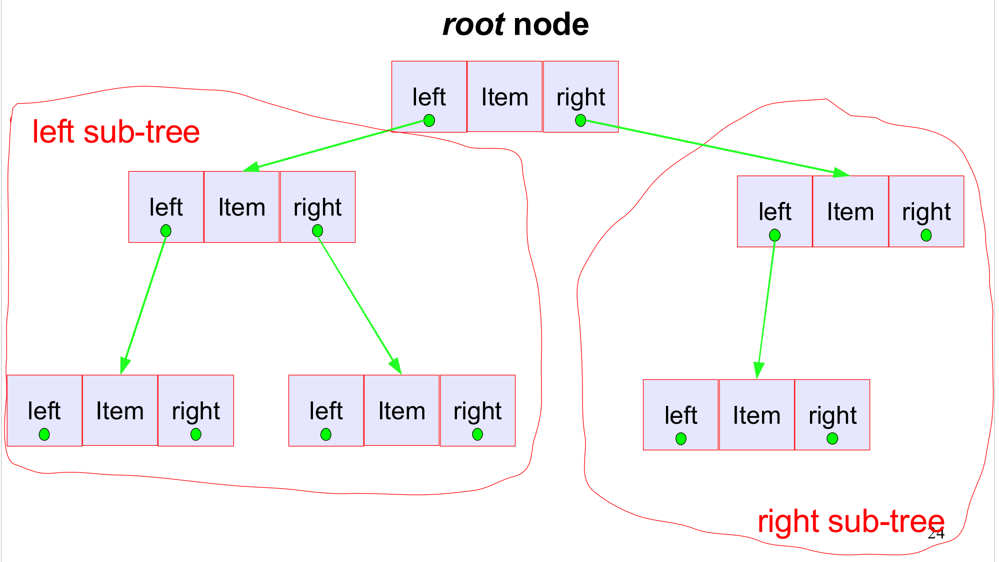
Like a list, a tree has a head node. For trees, this is called the "root" node. If each node of the tree has two links it's called a "binary tree".



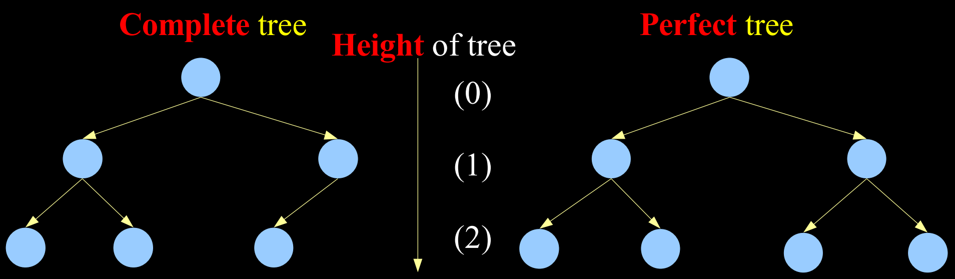
Anatomy of a Binary Tree:



Sub-Trees:



More Tree definitions:



Filled to level h, or h-1,
All nodes as far left as possible.

All levels filled.

Number of possible positions:

$$n = 2^0 + 2^1 + 2^2 + \dots 2^h = 2^{(h+1)}-1$$

Minimum height:

$$h = \text{floor}(\log_2(n))$$

Binary Search Trees:

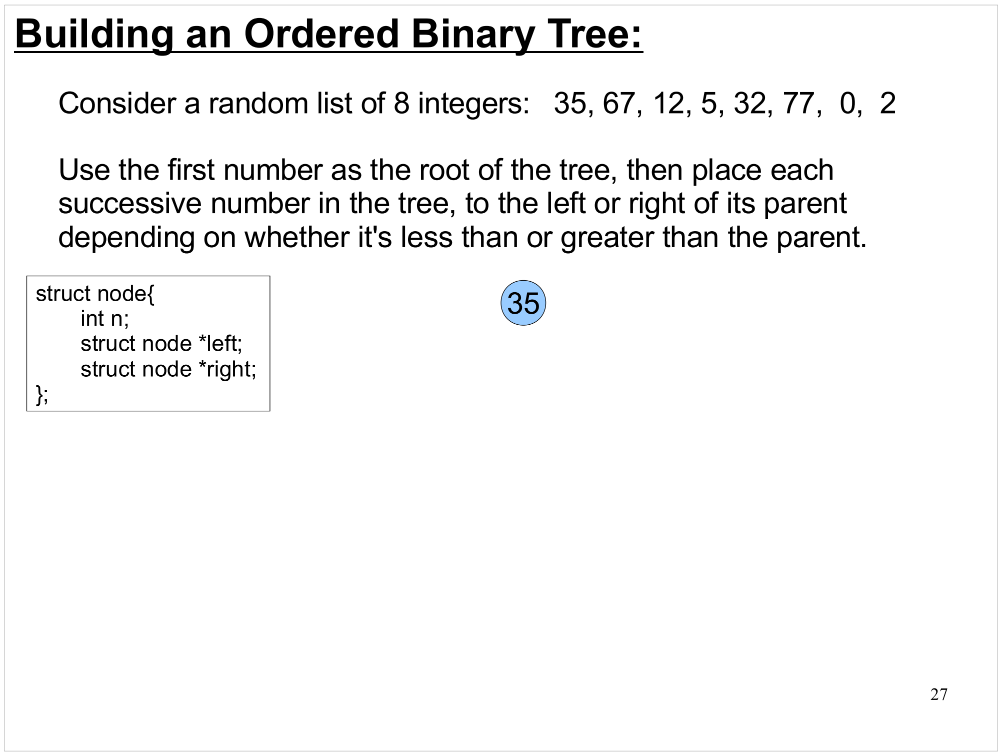
As we mentioned earlier, linked lists can only be searched sequentially. This isn't true for trees, though.

Binary trees are often used for storing data that must be searched quickly.

An ordered binary tree is defined as one for which:

1. values of all the nodes in left sub-tree are less than that of the root,
2. values of all the nodes in right sub-tree are greater than that of the root,
3. the left and right sub-trees are themselves ordered binary trees.

This should sound familiar from our earlier discussion of the Quicksort sorting algorithm.



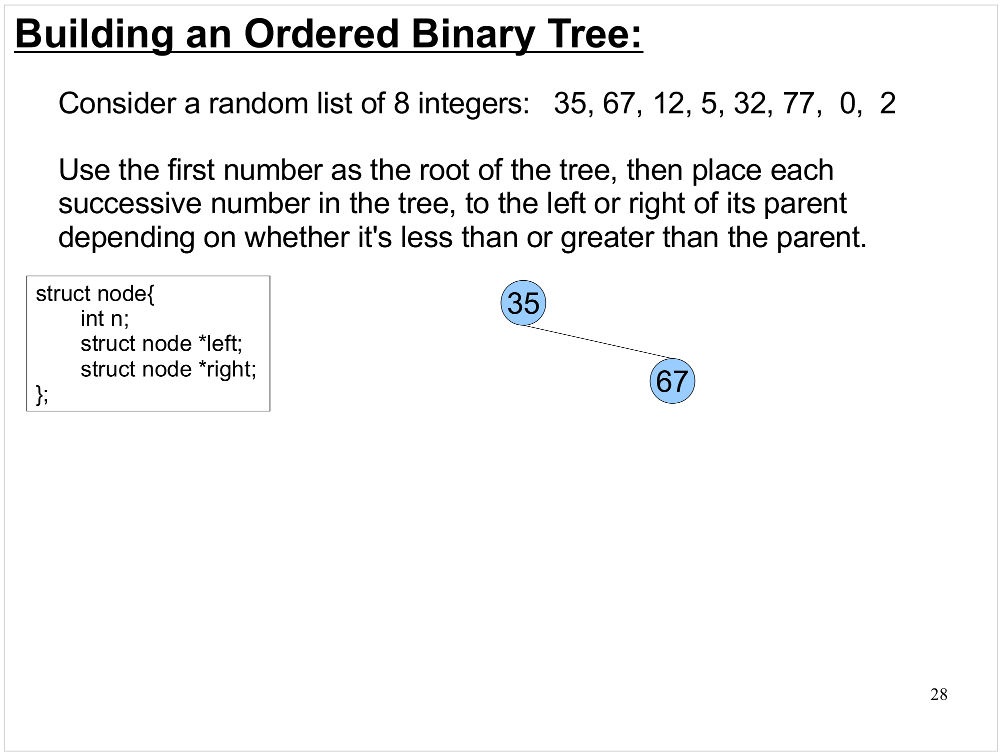
Building an Ordered Binary Tree:

Consider a random list of 8 integers: 35, 67, 12, 5, 32, 77, 0, 2

Use the first number as the root of the tree, then place each successive number in the tree, to the left or right of its parent depending on whether it's less than or greater than the parent.

```
struct node{
      int n;
      struct node *left;
      struct node *right;
};
```

Start by putting the first item, 35, at the root of the tree.

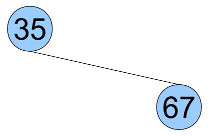


Building an Ordered Binary Tree:

Consider a random list of 8 integers: 35, 67, 12, 5, 32, 77, 0, 2

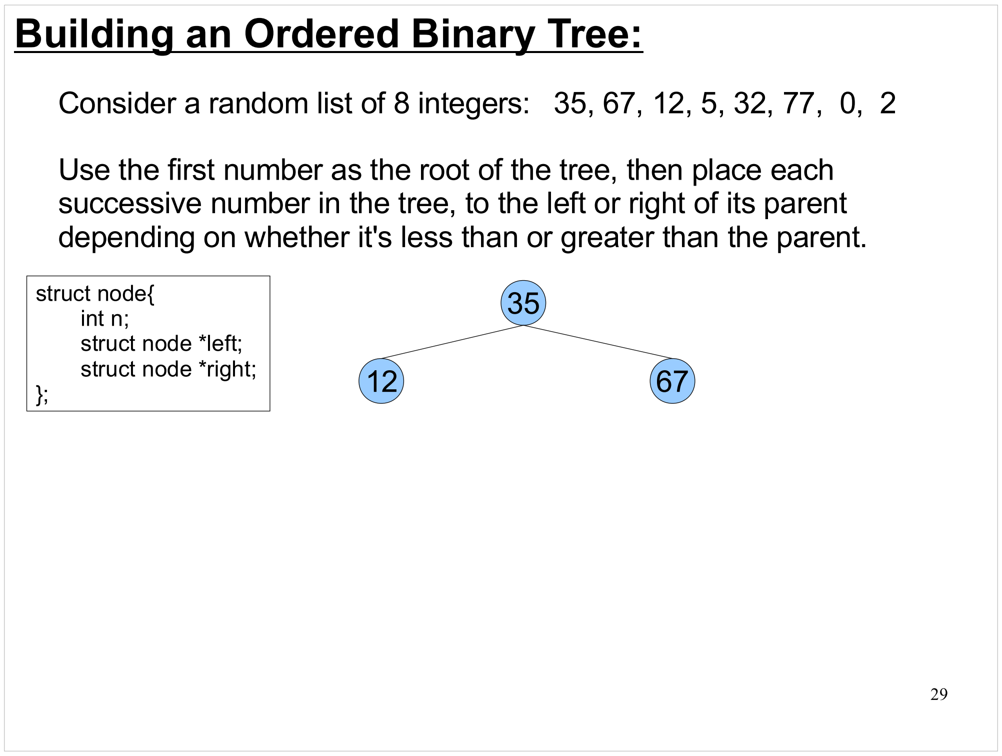
Use the first number as the root of the tree, then place each successive number in the tree, to the left or right of its parent depending on whether it's less than or greater than the parent.

```
struct node{
      int n;
      struct node *left;
      struct node *right;
};
```



28

The next item, 67, is greater than 35, so it goes on the right-hand link.

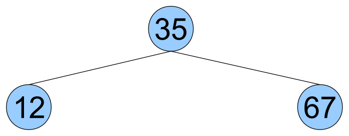


Building an Ordered Binary Tree:

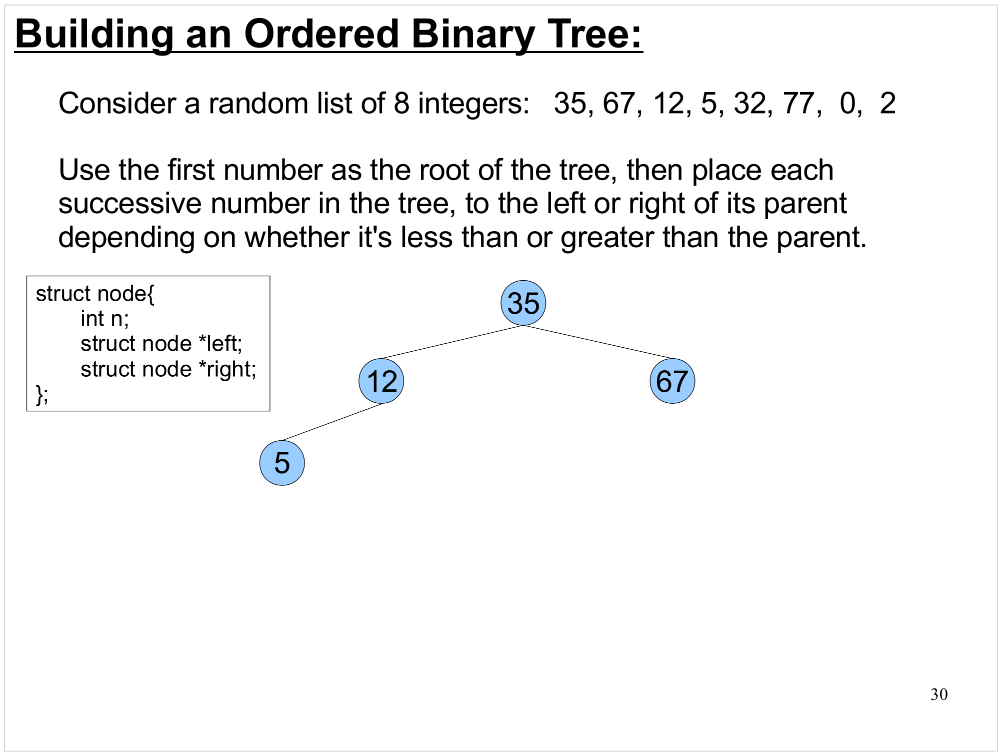
Consider a random list of 8 integers: 35, 67, 12, 5, 32, 77, 0, 2

Use the first number as the root of the tree, then place each successive number in the tree, to the left or right of its parent depending on whether it's less than or greater than the parent.

```
struct node{
      int n;
      struct node *left;
      struct node *right;
};
```



The next number, 12, is less than 35, so it goes on the left-hand link.

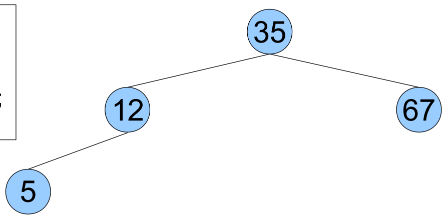


Building an Ordered Binary Tree:

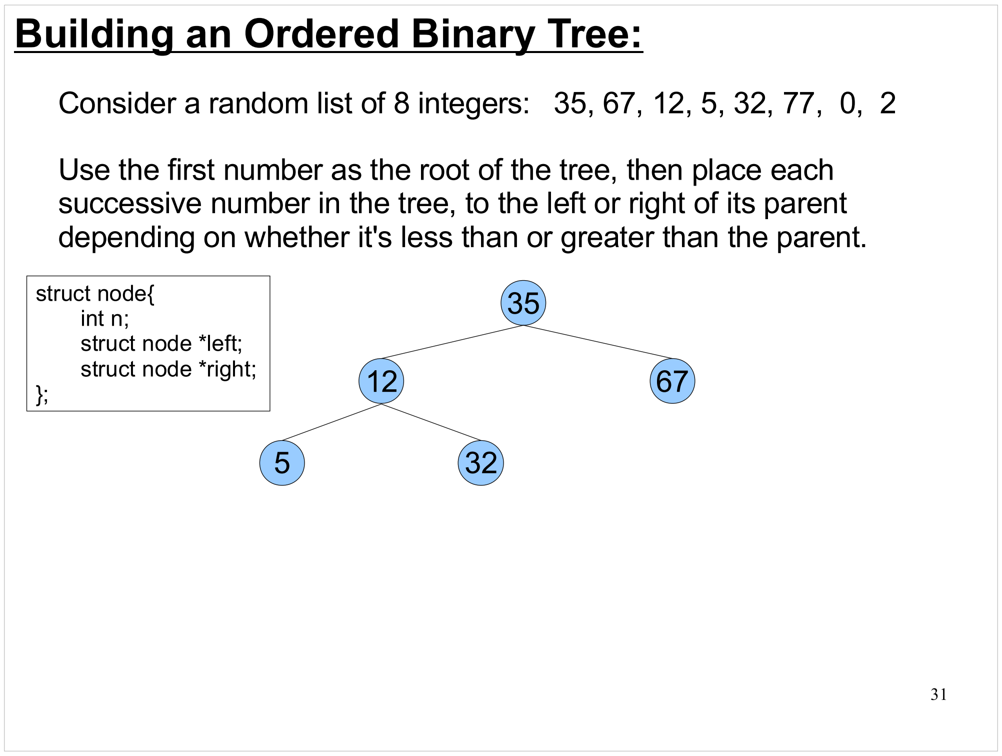
Consider a random list of 8 integers: 35, 67, 12, 5, 32, 77, 0, 2

Use the first number as the root of the tree, then place each successive number in the tree, to the left or right of its parent depending on whether it's less than or greater than the parent.

```
struct node{
      int n;
      struct node *left;
      struct node *right;
};
```



The number 5 is less than 35 and less than 12, so it goes on the left-hand link of the 12 node.

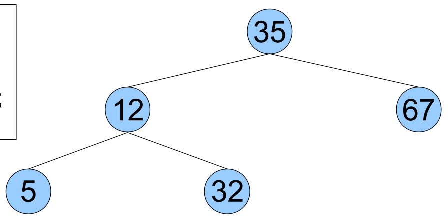


Building an Ordered Binary Tree:

Consider a random list of 8 integers: 35, 67, 12, 5, 32, 77, 0, 2

Use the first number as the root of the tree, then place each successive number in the tree, to the left or right of its parent depending on whether it's less than or greater than the parent.

```
struct node{
      int n;
      struct node *left;
      struct node *right;
};
```



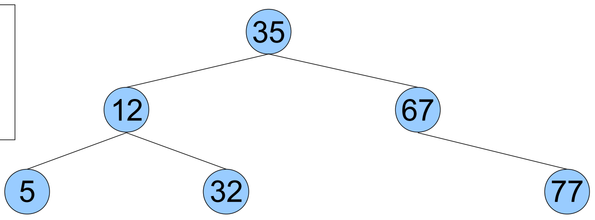
32 is less than 35, but greater than 12.

Building an Ordered Binary Tree:

Consider a random list of 8 integers: 35, 67, 12, 5, 32, 77, 0, 2

Use the first number as the root of the tree, then place each successive number in the tree, to the left or right of its parent depending on whether it's less than or greater than the parent.

```
struct node{
      int n;
      struct node *left;
      struct node *right;
};
```



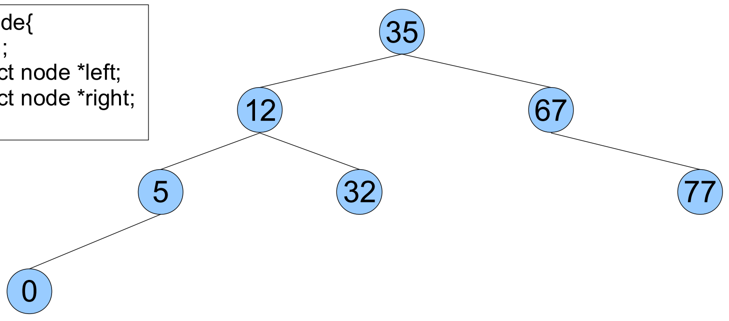
77 is greater than 35 and greater than 67.

Building an Ordered Binary Tree:

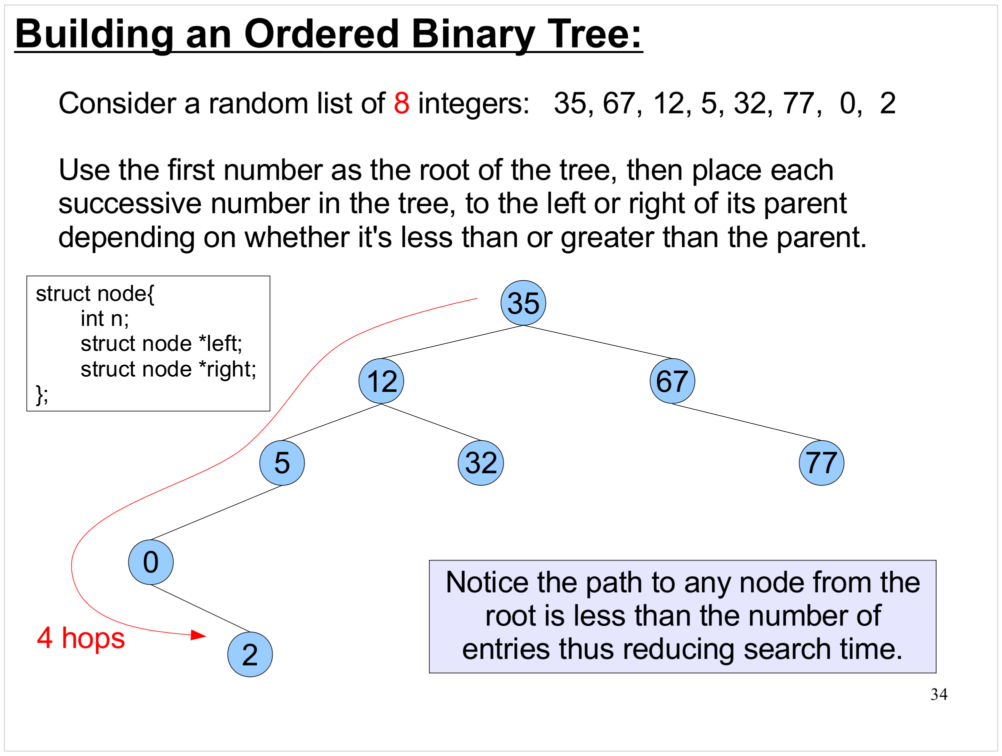
Consider a random list of 8 integers: 35, 67, 12, 5, 32, 77, 0, 2

Use the first number as the root of the tree, then place each successive number in the tree, to the left or right of its parent depending on whether it's less than or greater than the parent.

```
struct node{
      int n;
      struct node *left;
      struct node *right;
};
```



0 is less than 35, 12 and 5.



Building an Ordered Binary Tree:

Consider a random list of 8 integers: 35, 67, 12, 5, 32, 77, 0, 2

Use the first number as the root of the tree, then place each successive number in the tree, to the left or right of its parent depending on whether it's less than or greater than the parent.

```
struct node{
      int n;
      struct node *left;
      struct node *right;
};
```

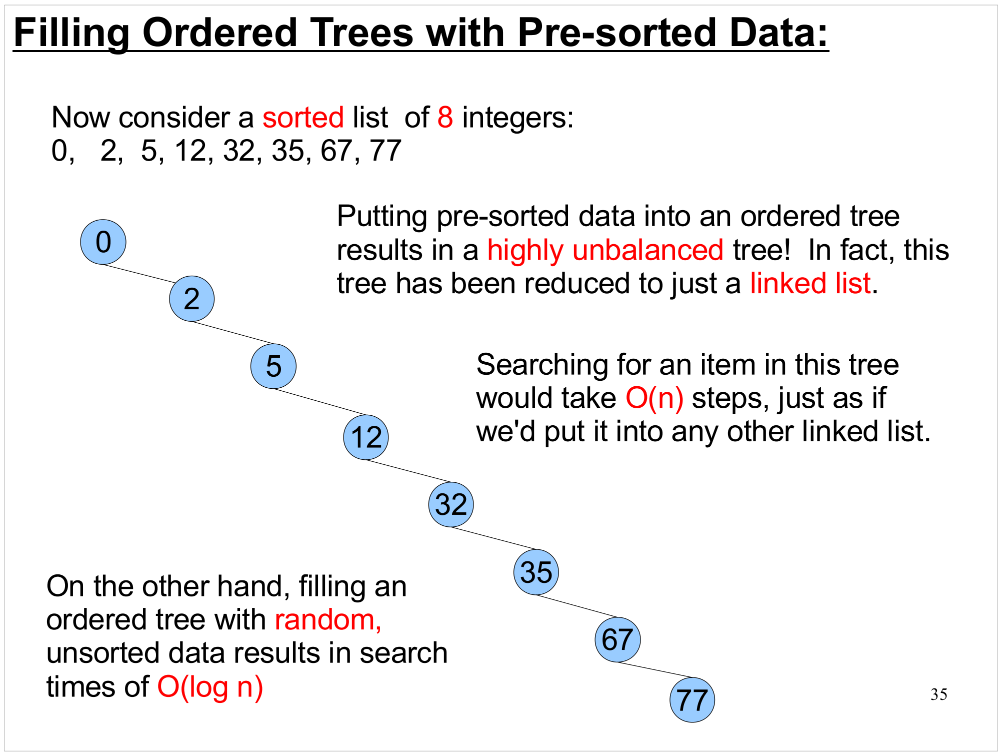
Notice the path to any node from the root is less than the number of entries thus reducing search time.

And finally, 2 is less than 35,12, and 5, but greater than 0.

If we had to search for “2” in the original list of integers, we'd need to go through all eight numbers before we found what we were looking for. The time required to search for a number in a linked list increases in proportion to the number of items.

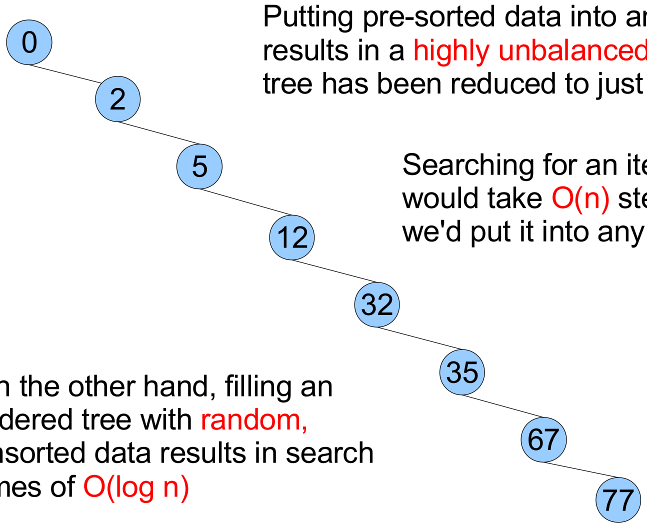
Searching from the root of the ordered binary tree, though, we can find “2” in only four steps. The time required to find an item in a binary tree grows in proportion to the log of the number of items.

If you need to search through many items, a binary tree can be searched much faster than a linked list.



Filling Ordered Trees with Pre-sorted Data:

Now consider a sorted list of 8 integers:
0, 2, 5, 12, 32, 35, 67, 77



Putting pre-sorted data into an ordered tree results in a highly unbalanced tree! In fact, this tree has been reduced to just a linked list.

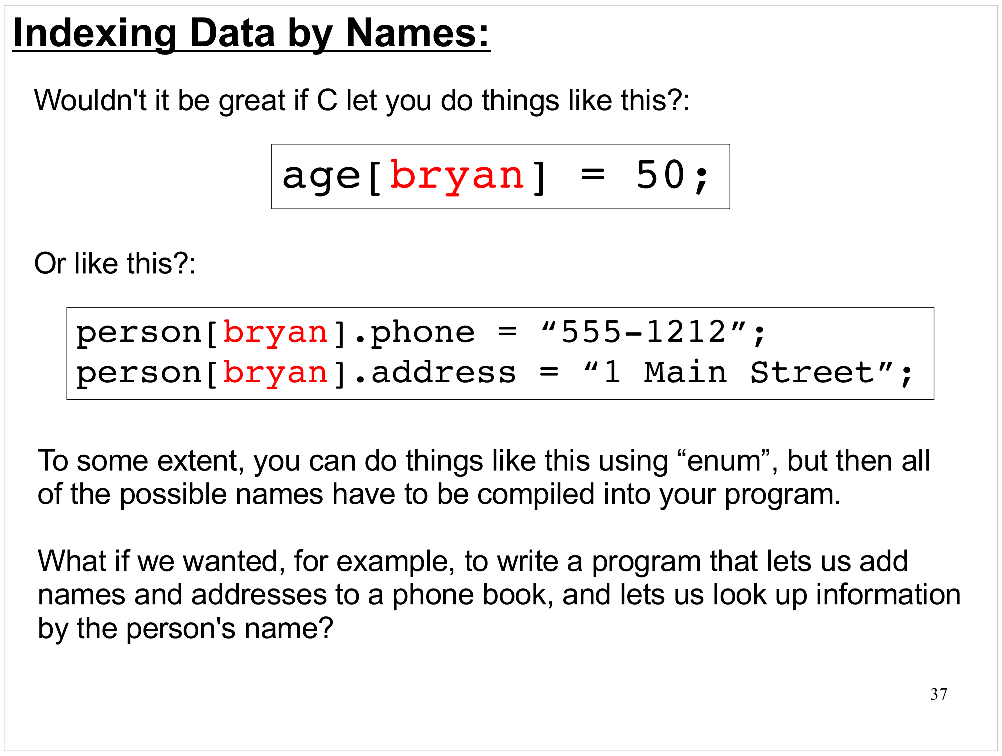
Searching for an item in this tree would take O(n) steps, just as if we'd put it into any other linked list.

On the other hand, filling an ordered tree with random, unsorted data results in search times of O(log n)

If we have a bunch of data that we know is already sorted, we might want to randomize the order of the items before we put them into our binary tree.

Part 4: Hash Tables

Hash tables are another useful data structure.



Indexing Data by Names:

Wouldn't it be great if C let you do things like this?:

```
age[bryan] = 50;
```

Or like this?:

```
person[bryan].phone = “555-1212”;
person[bryan].address = “1 Main Street”;
```

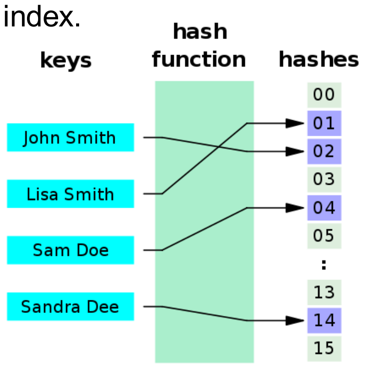
To some extent, you can do things like this using “enum”, but then all of the possible names have to be compiled into your program.

What if we wanted, for example, to write a program that lets us add names and addresses to a phone book, and lets us look up information by the person's name?

Note that the program statements above (sadly) won't really work in C. Some languages do support this kind of thing, though.

Hash Functions:

One way to do this is through a “hash function”. A hash function is something that takes a character string (often called a “key”) and converts it into a number (called a “hash”). We could then use the number as an array index.



With a suitable hash function, we might be able to write statements like this:

```
age[hash(bryan)] = 50;
```

```
int a = age[hash(bryan)];
```

A Simple Hash Function:

Here's a simple hash function that converts any character string into a number between 0 and some specified maximum:

Program stupidhash.cpp:

```
unsigned int stupidhash (char *key, unsigned int imax) {
  int sum = 0;
  for (unsigned int i=0;i<strlen(key);i++){
    sum += (int)key[i];
  }
  unsigned int hash = sum%imax;
  return(hash);
}

int main (int argc, char* argv[]) {
  const int nelements = 100;
  unsigned int hash = stupidhash(argv[1],nelements);
  printf ("hash = %d\n",hash);
}
```

Add up the letters as though they were numbers.

Use % to set the range.

Set max = 100.

```
$ stupidhash bryan
hash = 40
```

Hash Collisions:

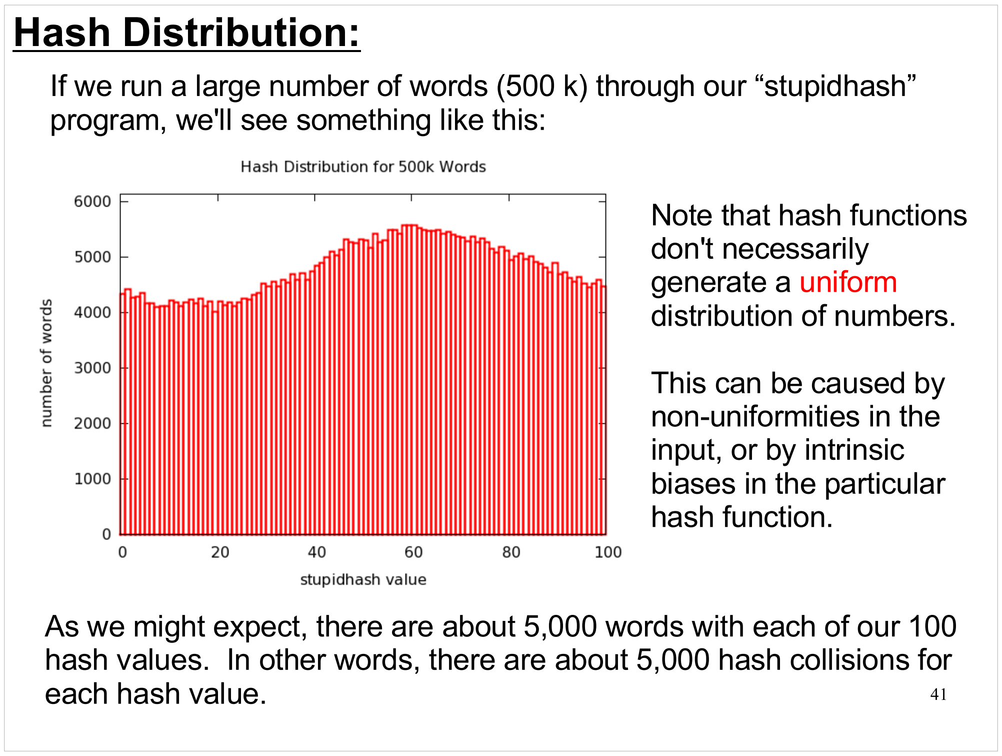
If we try out our hash function on a few names, we'll soon see a problem:

Name:	Hash:
Alice	78
Bob	75
Cindy	3
Dante	92
Babbage	60
Wainwright	60

$\left.\begin{matrix}60\\60\end{matrix}\right\}$ Hash Collision

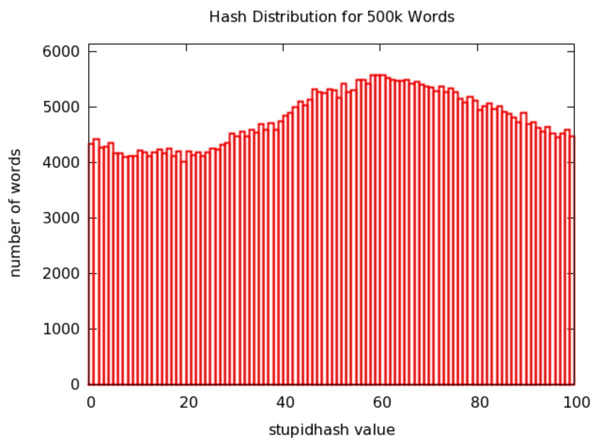
Since our hash function maps all possible strings into a small range of numbers, sooner or later we'll find multiple strings that have the same hash.

This is called a "hash collision", and it doesn't mean that our hash function is useless. It just makes things a little more complicated.



Hash Distribution:

If we run a large number of words (500 k) through our “stupidhash” program, we'll see something like this:

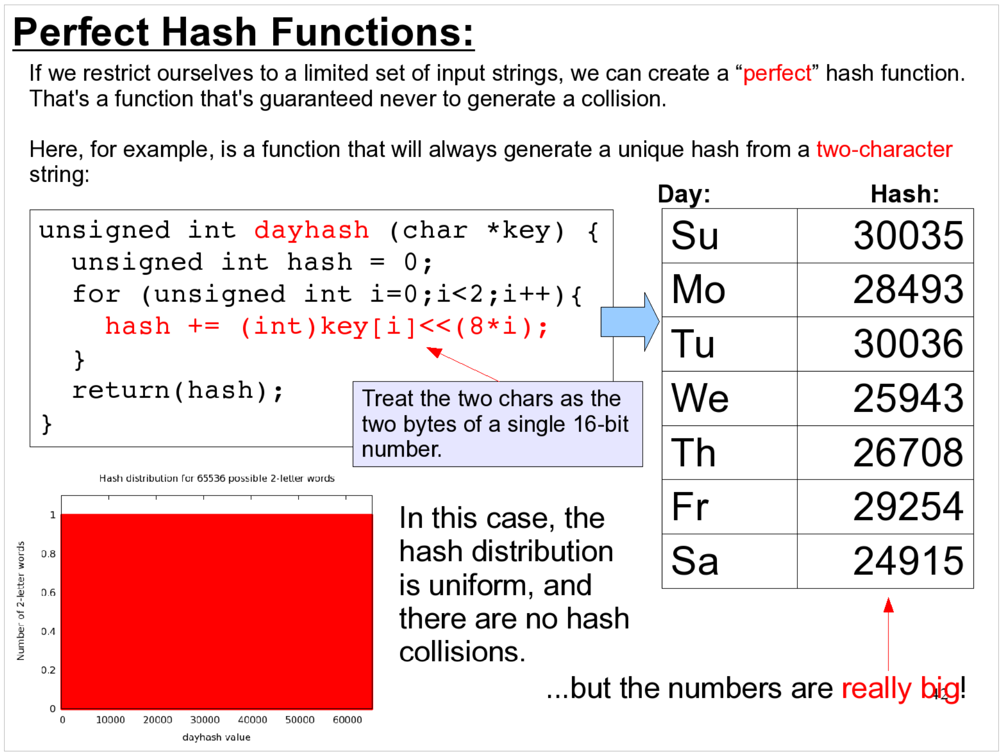


Note that hash functions don't necessarily generate a uniform distribution of numbers.

This can be caused by non-uniformities in the input, or by intrinsic biases in the particular hash function.

As we might expect, there are about 5,000 words with each of our 100 hash values. In other words, there are about 5,000 hash collisions for each hash value.

(The non-uniformities in this case are probably mostly due to the distribution of letters in the dictionary words I fed to the stupidhash function.)



Perfect Hash Functions:

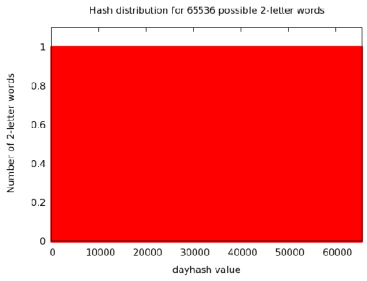
If we restrict ourselves to a limited set of input strings, we can create a “perfect” hash function. That's a function that's guaranteed never to generate a collision.

Here, for example, is a function that will always generate a unique hash from a two-character string:

```
unsigned int dayhash (char *key) {
  unsigned int hash = 0;
  for (unsigned int i=0;i<2;i++){
    hash += (int)key[i]<<(8*i);
  }
  return(hash);
}
```

Treat the two chars as the two bytes of a single 16-bit number.

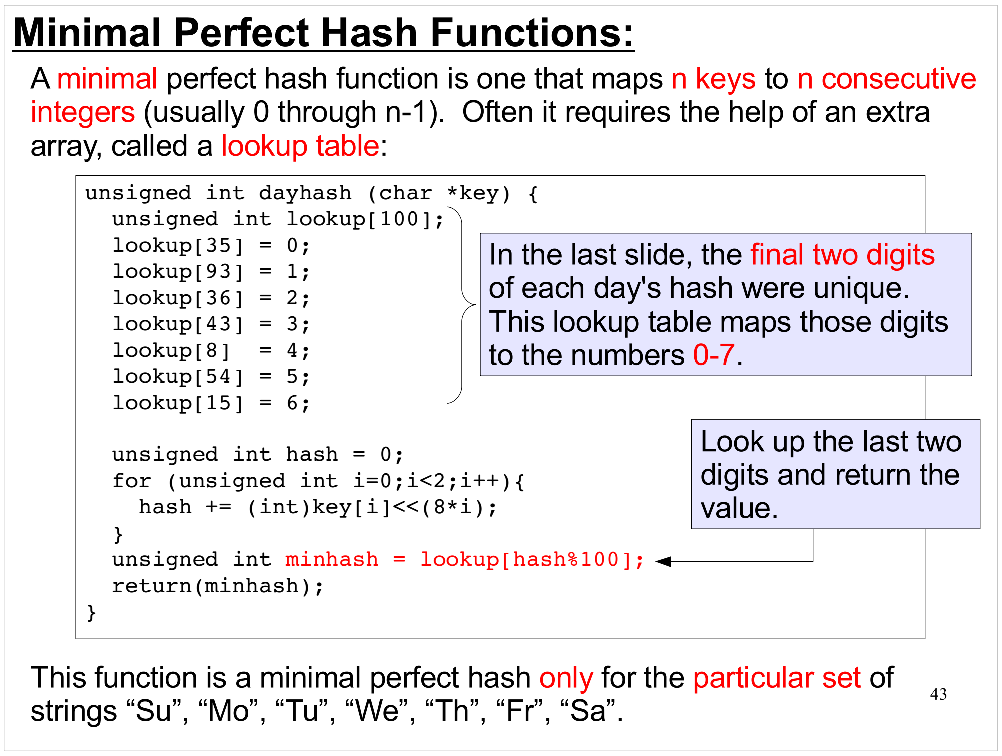
Day:	Hash:
Su	30035
Mo	28493
Tu	30036
We	25943
Th	26708
Fr	29254
Sa	24915



In this case, the hash distribution is uniform, and there are no hash collisions.

...but the numbers are really big!

The same hash function could be used, for example, with two-letter abbreviations for states in the U.S., or with any other 2-letter identifier.



Minimal Perfect Hash Functions:

A minimal perfect hash function is one that maps n keys to n consecutive integers (usually 0 through n-1). Often it requires the help of an extra array, called a lookup table:

```
unsigned int dayhash (char *key) {
  unsigned int lookup[100];
  lookup[35] = 0;
  lookup[93] = 1;
  lookup[36] = 2;
  lookup[43] = 3;
  lookup[8]  = 4;
  lookup[54] = 5;
  lookup[15] = 6;

  unsigned int hash = 0;
  for (unsigned int i=0;i<2;i++){
    hash += (int)key[i]<<(8*i);
  }
  unsigned int minhash = lookup[hash%100];
  return(minhash);
}
```

In the last slide, the final two digits of each day's hash were unique. This lookup table maps those digits to the numbers 0-7.

Look up the last two digits and return the value.

This function is a minimal perfect hash only for the particular set of strings “Su”, “Mo”, “Tu”, “We”, “Th”, “Fr”, “Sa”.

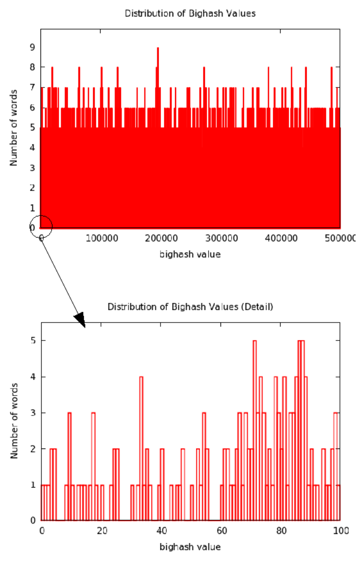
The 100-element lookup table is still larger than the list of values, but it's much smaller than the 65,536-element array we'd need if we used the preceding hash function.

But look how complicated this function is, and how fragile and limited it is. We've gone to a lot of trouble to get a hash function that only works for seven specific inputs.

Sometimes this amount of trouble is justified. For example, if we were writing a parser for a compiler or an interpreter, and needed to associated a known, fixed list of keywords (“if”, “else”, “int”, “double”) with values in an array.

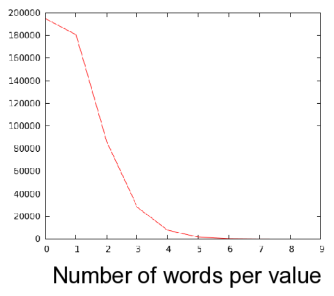
Bigger Arrays:

Maybe we could avoid collisions by just expanding the range of values generated by our “stupidhash” function. For example, with our 500k words, what if our hash function generated 500k possible integers?



We would probably still get hash collisions!

Even though the average number of words per value is 1, there are fluctuations. In some places, the words pile up. In others, there are no words at all.



The number of words per bin is just a Poisson distribution. We could look at a distribution like this and conclude that, for our purposes, the probability of getting more than a certain number of collisions was so low that we could ignore it, or deal with it in some time-expensive way that would only be occasionally invoked.

Dealing with Collisions:

The bottom line is that in many real-world situations we'll just have to accept hash collisions and deal with them.

We could do this by replacing a simple array like:

```
int age[hash]
```

with a more complicated structure. For example, we could have a 2-dimensional array:

```
int age[hash][j]
```

where the second index was used to hold data for each of the multiple keys that could produce this hash value.

Finding the Right Match:

Going back to our original table of stupidhash values, let's say we want to use these to store the ages of a bunch of people.

Name:	Hash:	Age:
Alice	78	12
Bob	75	83
Cindy	3	4
Dante	92	138
Babbage	60	37
Wainwright	60	42

We could use our 2-d array to store the ages for both of our colliding names:

```
age[60][0] = 37;
age[60][1] = 42;
```

But how do we know which is which? Maybe we need to store more information.

An Array of Structs:

Why don't we replace our array of “int”s with an array of “struct”s. Each struct will hold the data we're interested in, as wall as some other information that will help us resolve hash collisions.

```
const unsigned int hash_max = 100;
const unsigned int hash_maxcoll = 10;

typedef struct hashnode_struct {
   int data;
   char word[80];
   int collisions;
} HashNode;

HashNode hash_array[hash_max][hash_maxcoll];
```

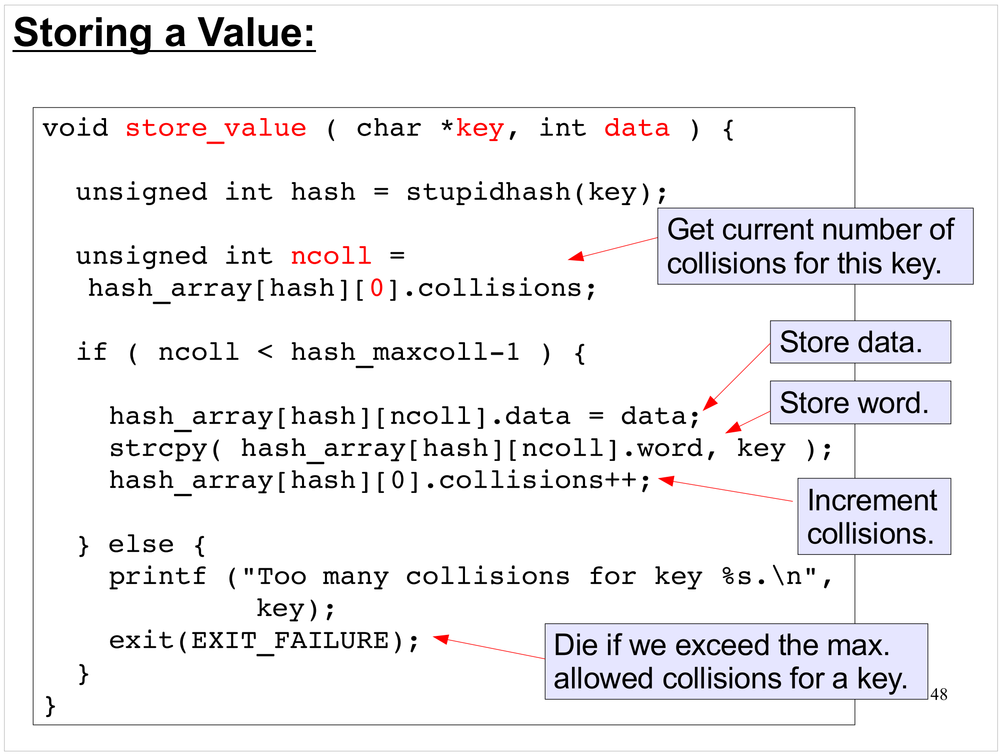
Maximum hash value.

Max. collisions we'll allow.

The data we're storing (e.g., age).

The word that made this hash.

Number of collisions for this hash.



Storing a Value:

```
void store_value ( char *key, int data ) {

  unsigned int hash = stupidhash(key);

  unsigned int ncoll =
   hash_array[hash][0].collisions;

  if ( ncoll < hash_maxcoll-1 ) {

    hash_array[hash][ncoll].data = data;
    strcpy( hash_array[hash][ncoll].word, key );
    hash_array[hash][0].collisions++;

  } else {
    printf ("Too many collisions for key %s.\n",
             key);
    exit(EXIT_FAILURE);
  }
}
```

Get current number of collisions for this key.

Store data.

Store word.

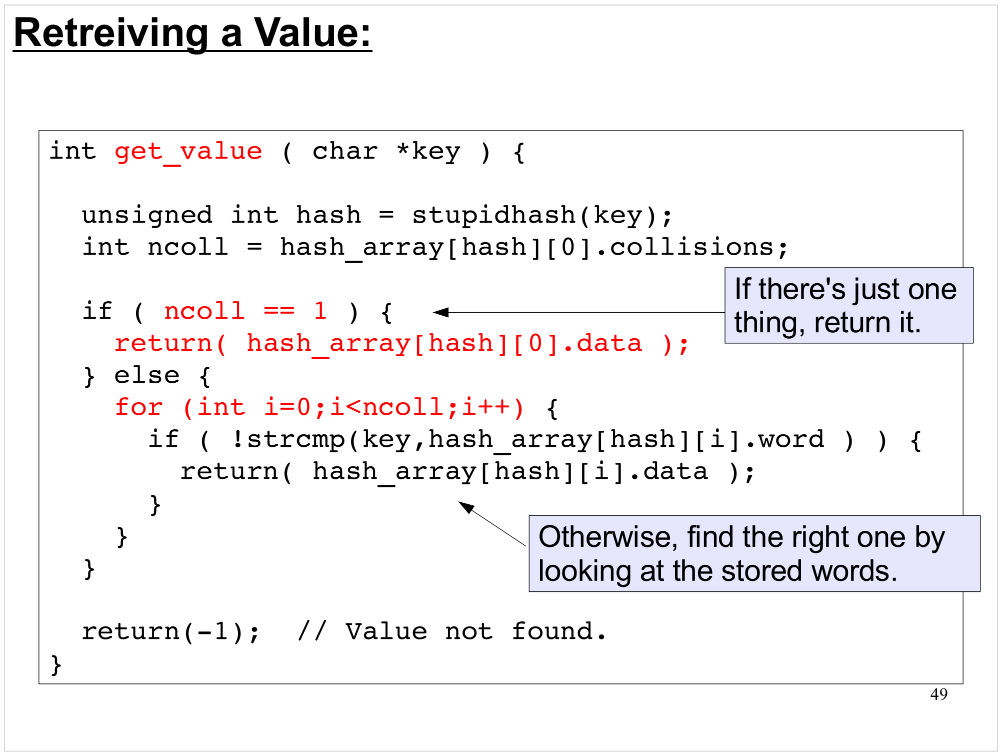
Increment collisions.

Die if we exceed the max. allowed collisions for a key.

48

Note that, in this code, we just store the number of collisions in the [0] element of each row.

Also note that “collisions” is probably a bad choice of name for this variable, since it's really counting the number of words that had that particular hash value. So, if “collisions” = 1, that means that one word is stored in that row.



Retreiving a Value:

```
int get_value ( char *key ) {

  unsigned int hash = stupidhash(key);
  int ncoll = hash_array[hash][0].collisions;

  if ( ncoll == 1 ) {
    return( hash_array[hash][0].data );
  } else {
    for (int i=0;i<ncoll;i++) {
      if ( !strcmp(key,hash_array[hash][i].word ) ) {
        return( hash_array[hash][i].data );
      }
    }
  }

  return(-1);  // Value not found.
}
```

If there's just one thing, return it.

Otherwise, find the right one by looking at the stored words.

The set_value code in the preceding slide chooses to just abort the program if we exceed the maximum number of allowed collisions.

We could do other things:

- Expand the 2-d array, using “realloc”. This is time-expensive, but it would only need to be done infrequently.
- Warn the user about the problem, and throw away that particular word. This is a reasonable response in some situations.
- Use a second, alternative hash function to generate a new hash value. Then, when searching for an entry later we'd need to be sure to check both hash values.

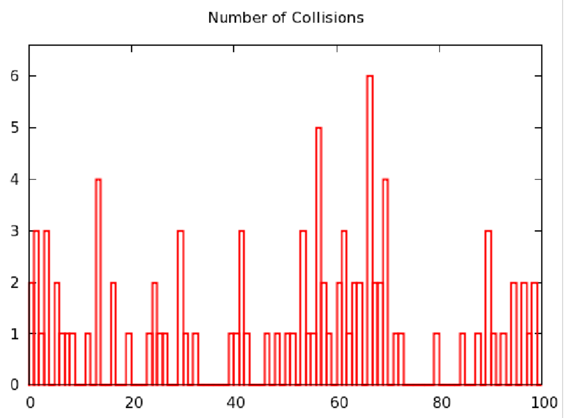
Using the Functions:

We could use these functions to store and retrieve values like this:

```
store_value("bryan",50);
```

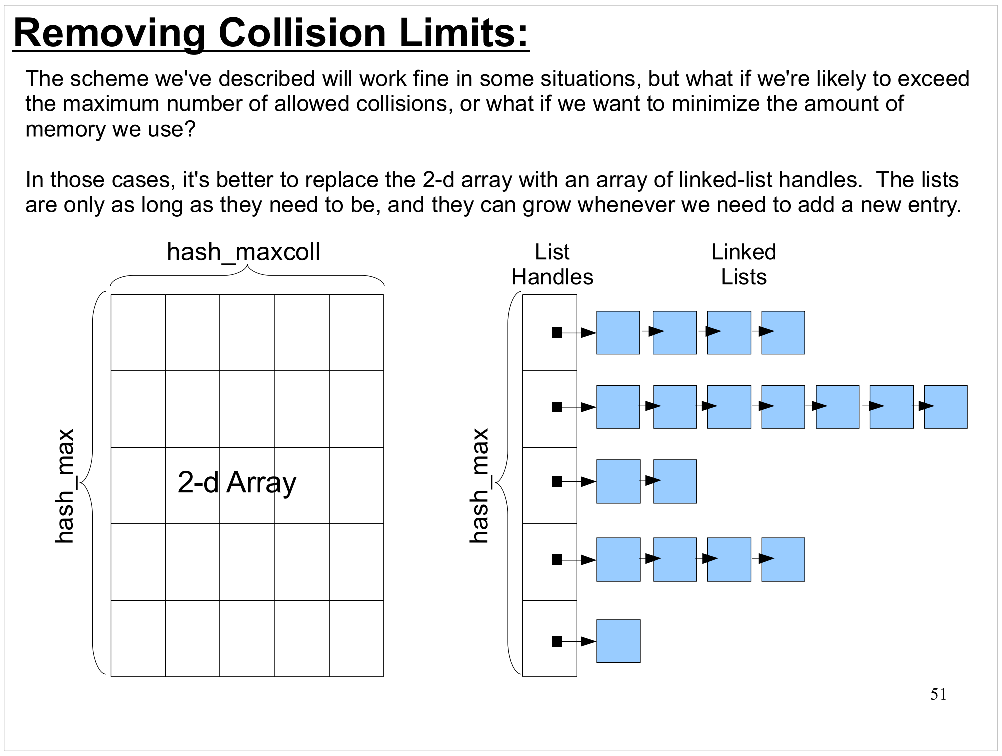
```
int a = get_value("bryan");
```

Number of collisions for 100 random dictionary words put into a 100-element hash array:



As long as the number of collisions for any value doesn't exceed the maximum width of our 2-d array, this will work fine.

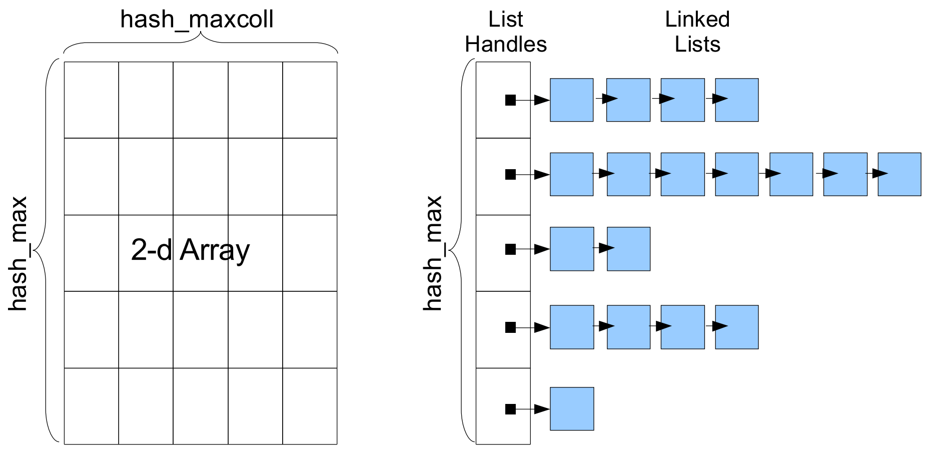
With appropriate choices for the hash array size and maximum allowable number of collisions, we could use these functions in real programs.



Removing Collision Limits:

The scheme we've described will work fine in some situations, but what if we're likely to exceed the maximum number of allowed collisions, or what if we want to minimize the amount of memory we use?

In those cases, it's better to replace the 2-d array with an array of linked-list handles. The lists are only as long as they need to be, and they can grow whenever we need to add a new entry.



In reality, programs that implement hash tables use schemes like the one on the right. This doesn't take up any more memory than necessary, and the number of collisions can grow without bound.

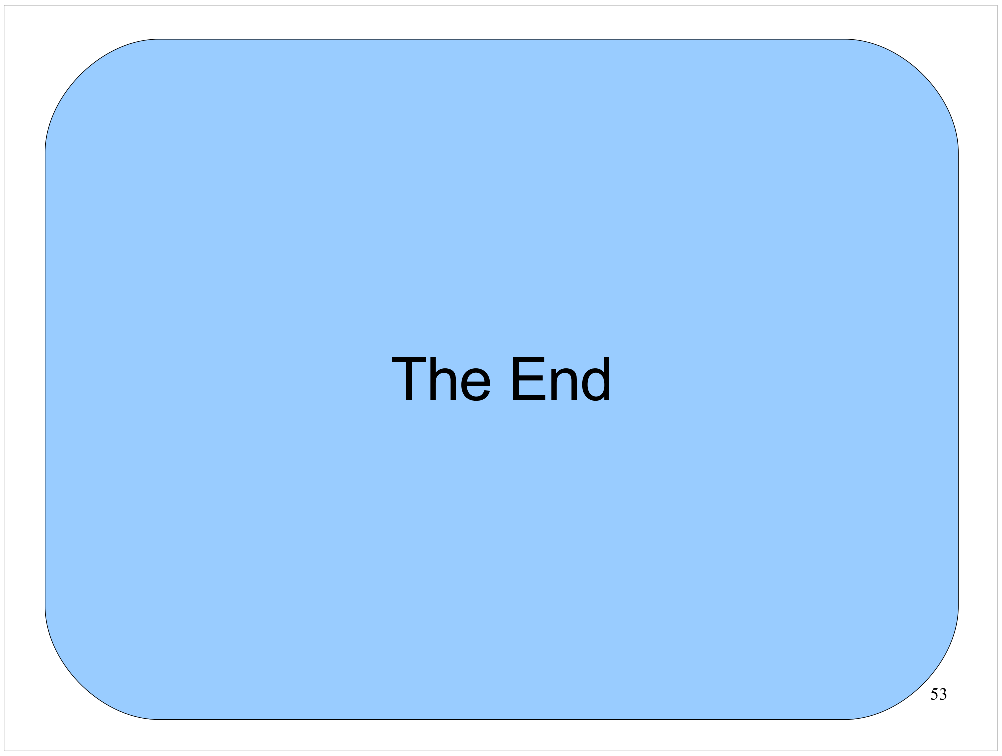
Next Time:

- ???????????????????????????

This week's Lab:

- Linked Lists

Brooks Ch. 1, Ch. 2 sections 1-2.2



The End

Thanks!