**Heres the deal, this is covering a whole lot so I haven’t done much on the new stuff. Feel free to add there. I’d recommend reading the whole thing downloading it and printing what you want. If the formatting gets screwed, i’ll download tonight when I’m done so you can just ask for that. As well since we will be adding to the top the formatting will get a little weird. So I probably won't mess with it until tomorrow around 5 and then once it is finalized i'll make sure all the pictures are in the right area besides what's on here theres a review for midterm 2 and 3 that we all have**

**CSCI 2270**

**Review for final exam. All 3 previous review sheets are also important.**

**Given the graph with vertices A, B, C, D, E, and F, and the following edges:**

**A: F, B**

**B: A, C**

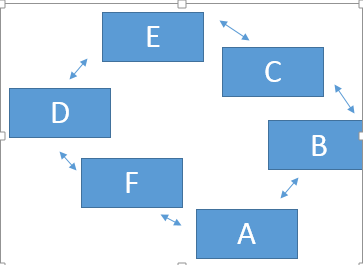
**C: B, D**

**D: C, E**

**E: D, F**

**F: E, A**

**1. Draw the graph as vertices and edges.**

This is how the physically it would look. As you can see from A you can only get to F and B… so on so forth. Each box is a vertex while an arrow indicates an edge. But keep in mind this is not how it appears to a computer. In terms of how the computer sees it, is much more like a tree. Remember heaps? These are all put into heaps and thats how the kept track of. 

// I think b and c are flipped

|  |  |
| --- | --- |
|  |  |
|  |  |

I think these 2 might be backwards...someone else take a stab at them

**2. In a depth first search of the above graph, what vertices will it pop off the stack in a search starting at A and ending at F?**

In depth first search it will pop A off first and then Pop F off.

**3. In a breadth first search of the above graph, what vertices will it pop off the queue in a search starting at C and ending at D?**

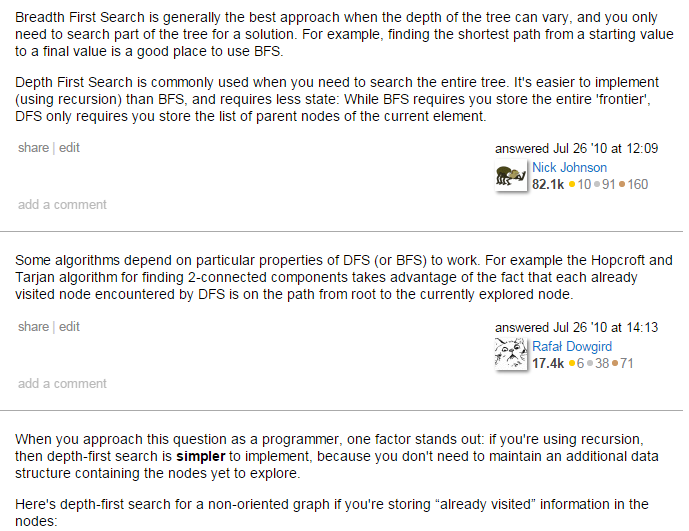
// I think that it should search B then A then F and then D because it searches closest to the origin first.

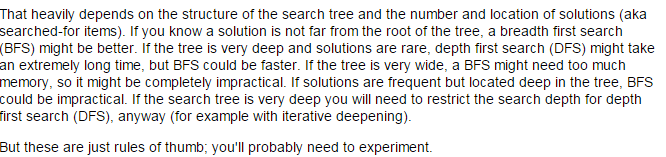
**4. What’s better about breadth first search than depth first search?**

Breadth first search is better at finding the shortest path between two destinations because it hits every vertex. It starts with the locations closest to the origin and expands from there. It takes longer to find a fast way than any way at all.

**5. Which takes longer, breadth first or depth first search?**

Short Answer: Breadth First





**6. Given a hash table of size 17 (this tells you the hash function to use) that uses open addressing**

**plus a search for the next open slot, add the pairs:**

**138, “Frodo”**

**241, “Pippin”**

**070, “Merry”**

**104, “Tom Bombadil”**

**106, “Dick Cheney”**

**Draw the final table when you are done.**

**What problem is getting worse here?**

**How would your answer change if you used double hashing with a second hash function of modulo 5?**

**How would your answer change if you used chained hashing?**

For the first part, figure out which key each thing maps to by doing x % 17.

138 % 17 = 2

241 % 17 = 3

70 % 17 = 2

104 % 17 = 2

106 % 17 = 4

If the hash function produces a collision (like 138, 70, & 104), the slots are examined in a probe sequence until an unused slot is found.

From wikipedia- Well-known probe sequences include:

* Linear probing, in which the interval between probes is fixed (usually 1)
* Quadratic probing, in which the interval between probes is increased by adding the successive outputs of a quadratic polynomial to the starting value given by the original hash computation
* Double hashing, in which the interval between probes is computed by another hash function

When a linear probing scheme is used, a collision is handled by moving down the table until an open slot is available. In searching, first the key it checked and then every entry below it is checked until either you find what you’re looking for, or reach an empty spot.

For this example it’d look like this

0 - x

1 - x

2 - 138 - “Frodo”

3 - 241 - “Pippin”

4 - 70 - “Merry”

5 - 104 - “Tom Bombadil”

6 - 106 - “Dick Cheney”

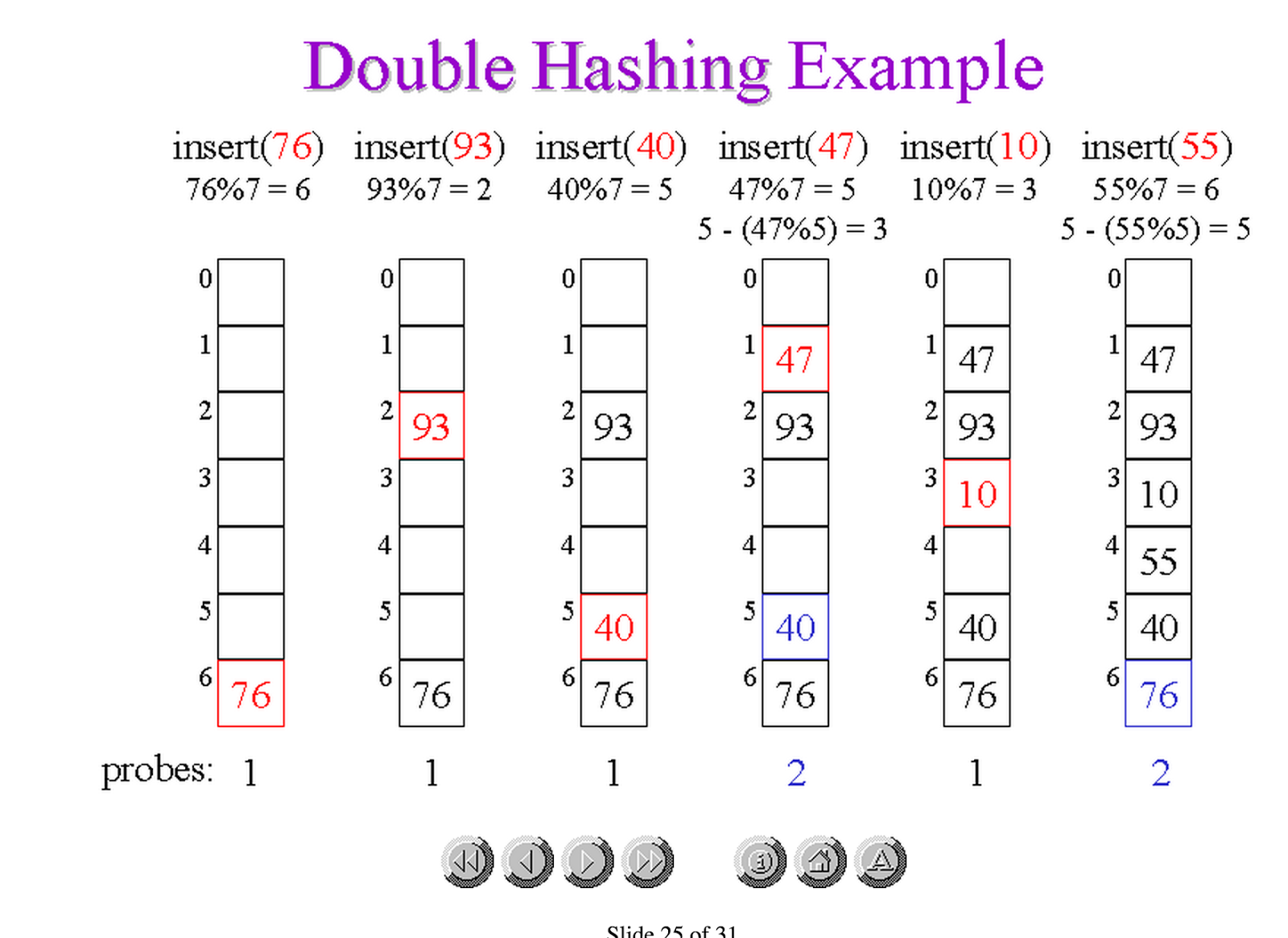
7 - x

…

13 - x

If you searched for “Dick Cheney” it would check slot 4 first (because 106%17=4), then since tom is there instead of cheney, it would check slot 5, then check slot 6.

This example illustrates that a non-uniform hash function can lead to a cluttered hash table that takes a while to find stuff. (By non-uniform hash function, I mean one that maps a lot of the items you’re trying to store to the same key)

**Double hashing**

Looks like when you get a collision you do (new modulo) - (value % new modulo)...

*This is just one potential double hashing function, I don’t remember if she gave us a specific one in class, but this is all I could find online.*

for this example…

138 % 17 = 2

241 % 17 = 3

70 % 17 = 2 -> collision -> 5-(70%5) = 5

104 % 17 = 2 -> collision -> 5-(104%5) = 1

106 % 17 = 4

so you have

0: <empty>

1: 104 - “Tom Bombadil”

2: 138 - “Frodo”

3: 241 - “Pippin”

4: 106 - “Dick Cheney”

5: 70 - “Merry”

I guess you search the table in the same way...

**Chained Hashing**

Each entry in the table is now the head pointer of a linked list. The time it takes to search is constant + O(N) where N is the longest list

for this example …

138 % 17 = 2

241 % 17 = 3

70 % 17 = 2

104 % 17 = 2

106 % 17 = 4

0: <empty> // some websites made it sound like a hashing function should

1: <empty> // never give you zero… this seems weird

2: 138 -> 70 -> 104

3: 241

4: 106

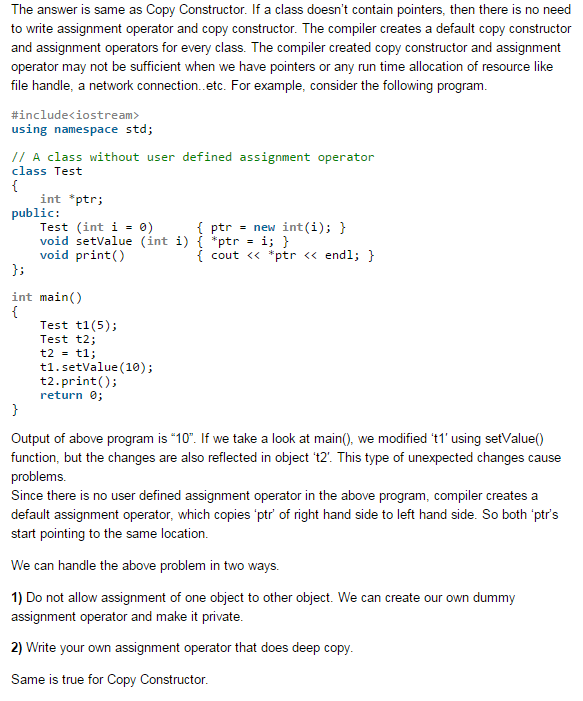
5: <empty>

...

16: <empty>

**7. Explain, in simple English, how a buffer overrun hack works.**

1. In computer security and programming, a buffer overflow, or buffer overrun, is an anomaly where a program, while writing data to a buffer,overruns the buffer's boundary and overwrites adjacent memory. This is a special case of violation of memory safety.

**8. What is the difference between a deep copy and a shallow copy? How can you write a test to tell which one you have? How do pointers and shallow copies relate to each other?** 

**\*\*PRETTY SURE SHES GOING TO PUT A LOT ON THE TEST ABOUT DEEP VS SHALLOW COPIES**

A shallow copy is one that still is pointing to the same address.

Example link to a webpage -> send the link to a friend. They have a shallow copy. Make a new webpage and send it to a friend, now they have a deep copy.  
Pointers point to the original data which is how we access data in a shallow copy

I don’t know how to write a formula for this. but possibly try to change the data and if the original is changed as well we know its a shallow copy

**9. How can you tell if 2 heaps in array form have all of the same elements?**

Start with are they the same number of elements?

**10. Why do big\_numbers benefit from a trim() function? When is such a function useful in HW2?**

The Trim() Function was beneficial to big\_number because it removed all unnecessary leading and trailing zeros. This was useful in HW2 because we were writing each individual digit to a doubly linked list. Removing the zeros saves us a lot of time and memory as well it will make the calculations easier later on because we don’t have to worry about leading zeros

**11. If we didn’t write big\_number’s operator =, but we used the default version that C++ gives us instead, will we leak memory?**

haha obviously or you wouldn’t ask the question

you have to override the assignment operator

**12. Give me an example of the scenario in question 11 causing a crash at runtime.**

**Consider big num a and big num b again. You call a = b;**

**Suppose a is taken out of scope. The destructor for a will be called but we just assigned b to a, meaning the pointer to a's place in memory now points to b's place in memory. When the a-destructor is called, the memory being used for b will be released. Fine, no harm done. BUT, if the program takes b out of scope, the destructor for b will automatically be called however, b has already been destroyed in memory and now we are calling the destructor on a piece of memory that has already deleted which will crash the program at runtime.**

**13. Why do we have the rule that heaps must be complete trees?**

It is defined by the way it is built by our code. a complete tree is one in which there are no gaps between leaves. For instance, a tree with a root node that has only one child must have its child as the left node. More precisely, a complete tree is one that has every level filled in before adding a node to the next level, and one that has the nodes in a given level filled in from left to right, with no breaks.

**14. Given the array 1 4 6 8 3 2 7 5 9 0, show me how quicksort could degrade to quadratic performance in the first 3 partition steps.**

In order to degrade it to n^2 time, you want to pick the pivot value is the least or greatest value in the array.

Now just work out the quicksort

1 4 6 8 3 2 7 5 9 0

0 4 6 8 3 2 7 5 9 1 (first partition)

0 1 6 8 3 2 7 5 9 4 (second partition)

0 1 2 8 3 6 7 5 9 4

0 1 2 3 8 6 7 5 9 4

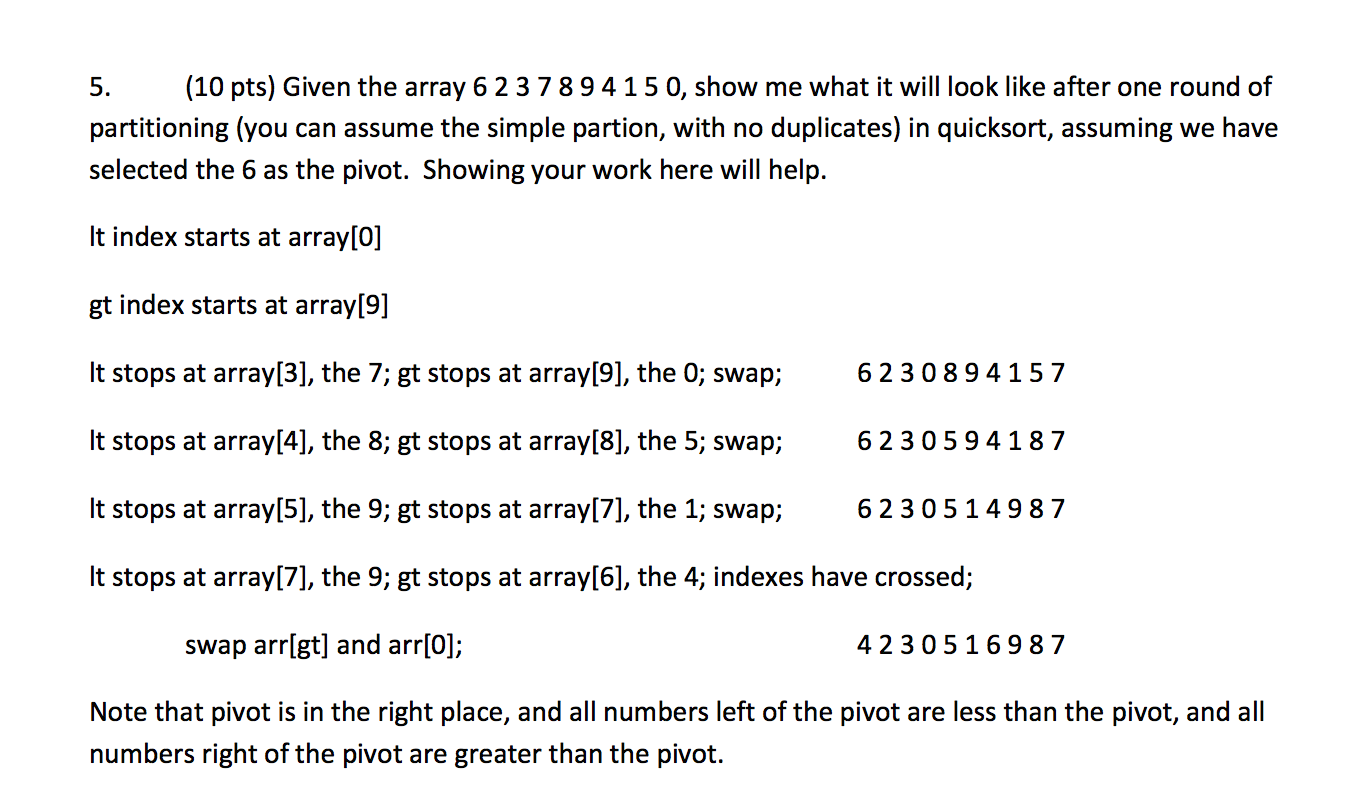
0 1 2 3 4 6 7 5 9 8 (third partition)

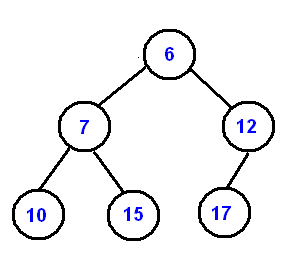
0 1 2 3 4 5 7 6 9 8 (fourth partition)

0 1 2 3 4 5 6 7 9 8 (fifth partition)

0 1 2 3 4 5 6 7 8 9 (six partition)

An example of quicksort from midterm 3



**15. Given a load factor of 25%, what is the general performance (in terms of expected slots checked) of a doubly-hashed hash table?**

The average performance of a doubly-hashed hash table is (–ln(1-a))/a where a is the number of items in table/capacity of table. Normally a decimal percent

So for 25% (-ln(.75)/.25) ~ 1.15

**16. When can a load factor exceed 100%? Why does this happen?**

This can happen with chained hashing tables the average run time is 1 + a/2

Chained hash tables with linked lists are popular because they require only basic data structures with simple algorithms, and can use simple hash functions that are unsuitable for other methods.

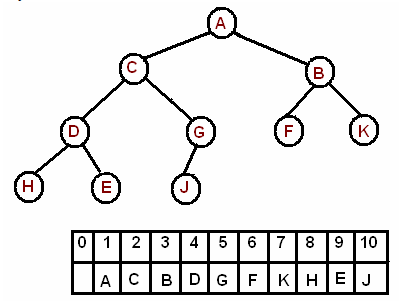
The cost of a table operation is that of scanning the entries of the selected bucket for the desired key. If the distribution of keys is sufficiently uniform, the average cost of a lookup depends only on the average number of keys per bucket—that is, on the load factor.

Chained hash tables remain effective even when the number of table entries n is much higher than the number of slots. Their performance degrades more gracefully (linearly) with the load factor. For example, a chained hash table with 1000 slots and 10,000 stored keys (load factor 10) is five to ten times slower than a 10,000-slot table (load factor 1); but still 1000 times faster than a plain sequential list, and possibly even faster than a balanced search tree.

**INFORMATION ON HEAPS!!**

**WHAT IS THIS HELPFUL FOR?**

I bet we will be asked about this stuff again since our graphing thin used it



A binary heap is a complete binary tree which satisfies the heap ordering property. The ordering can be one of two types:

* the *min-heap property*: the value of each node is greater than or equal to the value of its parent, with the minimum-value element at the root.
* the *max-heap property*: the value of each node is less than or equal to the value of its parent, with the maximum-value element at the root.

In a heap the highest (or lowest) priority element is always stored at the root, hence the name "heap". A heap is not a sorted structure and can be regarded as partially ordered. As you see from the picture, there is no particular relationship among nodes on any given level, even among the siblings.

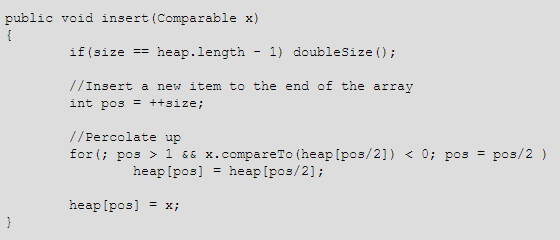
Since a heap is a complete binary tree, it has a smallest possible height - a heap with N nodes always has O(log N) height.

**A heap is useful data structure when you need to remove the object with the highest (or lowest) priority. A common use of a heap is to implement a priority queue.**

The root is the second item in the array. We skip the index zero cell of the array for the convenience of implementation. Consider k-th element of the array, the

* **its left child is located at 2\*k index**
* **its right child is located at 2\*k+1. index**
* **its parent is located at k/2 index**

### Insertmiggity moar.PNG

The new element is initially appended to the end of the heap (as the last element of the array). The heap property is repaired by comparing the added element with its parent and moving the added element up a level (swapping positions with the parent). This process is called "percolation up". The comparison is repeated until the parent is larger than or equal to the percolating element.

**To the PATTERN!**

**pattern(outs, 4, 0);** 2 calls to pattern(2), 7 calls in all

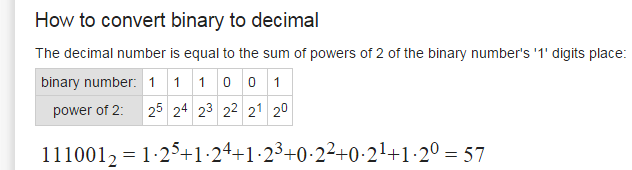
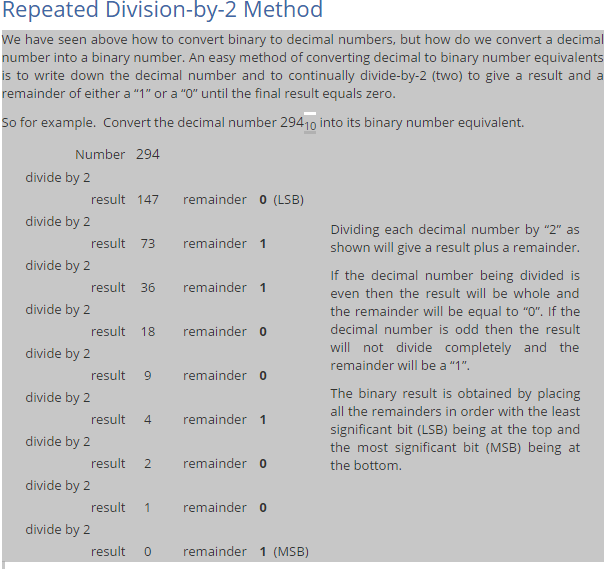
**pattern(outs, 16, 0);** 31 counting pattern(16)

**pattern(outs, 1024, 0);** 2047 counting pattern(1024)

**What formula describes this relationship between the starting n and the number of calls pattern makes?**

# of calls = # of lines = 2\*n-1

**NEXT BINARY TO DEC AND BACK**



**1a. Suppose my algorithm is cubic, O(n3), and I triple the size of its input. How much longer will it**

**take to run?**

**1b. Suppose my algorithm is logarithmic, O(log2n), and I quadruple the size of its input. How much**logn.PNG

**longer will it take to run?**

Let m = 4n. The algorithm is O(log2m) = O(log24n) = O(log24 + log2n) = O(2 + log2n)

**2. Why does the**

**remove\_node function: void remove\_node(node\*& head\_ptr, const int& target)**

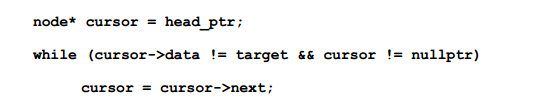
**pass in the head ptr by reference? Give me a scenario where this is needed.**

We need any changes that remove\_node makes to the pointer head\_ptr to stay permanent outside the code. For instance, any removal of the first node in the list requires us to update the head\_ptr. To accomplish this, we add an & so we can pass in the head\_ptr by reference. Any changes to the

head\_ptr’s address will then be updated permanently when the function is done working.

**3. The code below looks for a node containing target in a list, but will crash. Why? And how would**

**you fix this problem?**



That while loop is the problem. C++ compares cursor->data to target before it checks if cursor ==

nullptr. If we reach the end of the list without finding the target, we’ll try to get nullptr->data; this is like

saying (\*nullptr).data, and dereferencing nullptr is always going to be a bad idea.

To fix this, switch the 2 conditions. C++ can then decide that cursor’s nullptr, and thus the whole while

condition must be false, before it tries to grab the data at that nullptr.

**4. How do I copy elements 7-12 of a 30-element array called alice over elements 9-14 of a 45-**

**element array called bobo, using the copy command? You should not need any brackets [] in this**

**answer**

**Pointer arithmetic!**

**Remember that the first element is at index 0, so elements 7-12 are at alice[6] to alice[11].**

**Copy takes the index of the place to start copying from source, alice + 6,**

**the place to stop copying from source, alice + 12, which is one past alice[11]**

**and the place to start copying into destination, which is bobo + 8.**

**copy(alice + 6, alice + 12, bobo + 8);**

**5a. What can an array do faster than a linked list, and why?**

**Can be destroyed faster (one delete)**

**Can be walked faster (fewer dereferences)**

**Can access any element in the array in O(1) time (pointer arithmetic)**

**Can do binary search and find things quickly**

**5b. What can a linked list do faster than an array, and why?**

**It can add nodes between other nodes without needing to move things**

**It can thus resize in constant time**

**6. How can I tell if two arrays are shallow copies of each other when I am writing tests on my**

**code?**

**Shallow copies are copies that aren’t independent of each other, and usually these come from copying pointers. (Note that this question is not asking you if 2 unsorted arrays have the same numbers; both shallow and deep copies will look the same in terms of the contents.)**

**I could check if their addresses were the same (I sometimes call this the underpants test, but that’s not a technical term). If the 2 arrays are crammed into the same memory address, they must be the same array.**

**Or I could change one array and check the other one. If the change appears in the array I didn’t change,**

**I must have a shallow copy.**

**8a. Would this function**

**void bleh(int\* stuff)**

**let us change the data in the array called stuff? Justify your answer.**

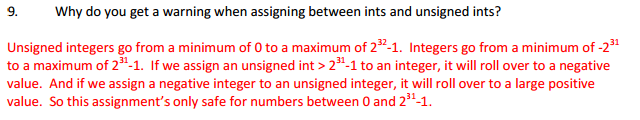
**Yes. We get the location of this array via the int\* pointer, so we can change the data at the location**

**given by that pointer, and it will stay changed after the function runs.**

**8b. Would that same function bleh be able to resize the stuff array? Justify your answer.**

**No. It could try, but any changes we make to the pointer stuff will not persist after the function ends.**

**We’d need to pass the pointer to stuff in as a reference parameter (int\*& stuff) to be able to change the array to a different size.**

**Now to more stuff from the midterm one review**

**Q. Why does the add node function**

**void add\_node(node\*& head\_ptr, const int& payload)**

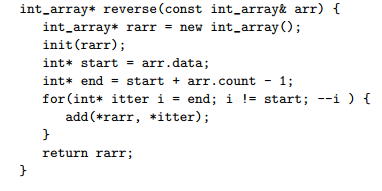
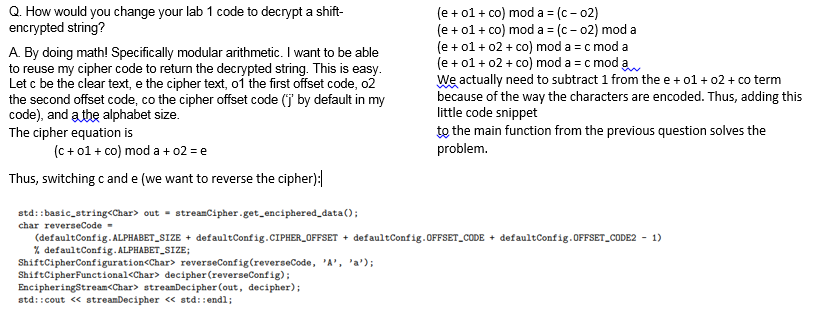
**pass in the head ptr by reference? Give me a scenario where this is needed.**

**A. For the same reason you ever pass an object by reference! You want to access (and perhaps alter) the literal object, not a copy of that object.**

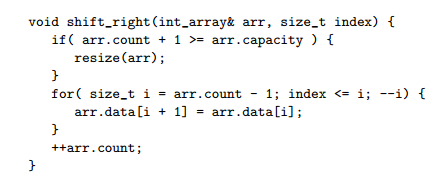
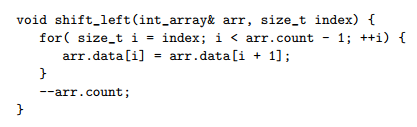
**In this case we want to be able to manipulate the head ptr variable directly. This is obviously necessary if we want to add an element which satisfies**

**payload < head\_ptr->data**

**In this case we want to make the pointer change targets (to a new node). If we were to pass in head ptr by value then we would only be able to change the target of the copy**

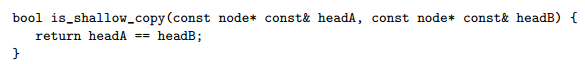
**• Q. Write me a function that reverses the data in an int array. Would this be as easy to do in a singly linked list?**

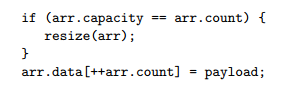
**Given an int array and a slot index, write me a loop that shifts the end elements in the int array data right to open a gap at that index.**

**Given an int array and a slot index, write me a loop that shifts the end elements in the int array data left to overwrite that index.**

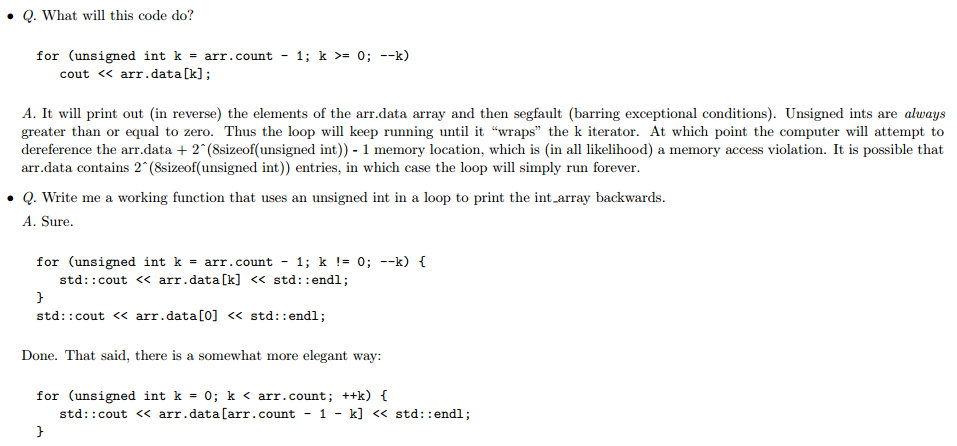
**Q. How can I tell if two linked lists are shallow copies of each other?**

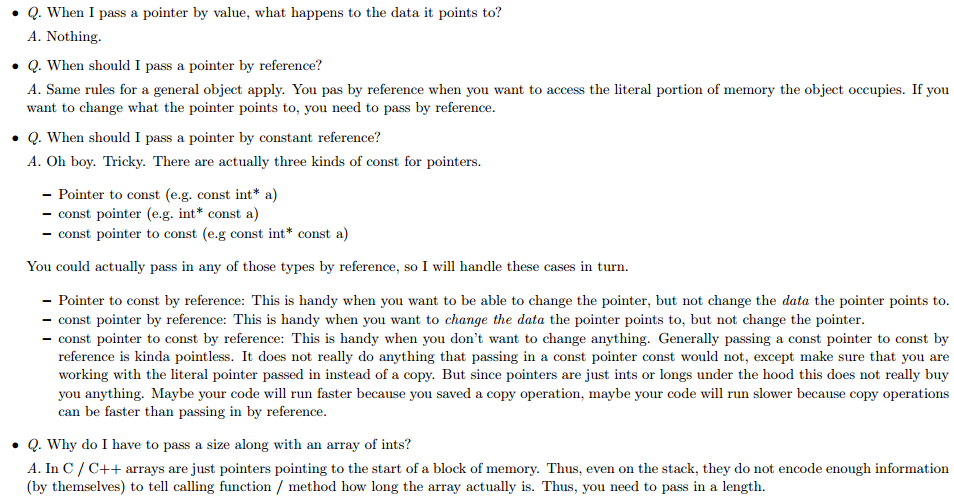
**A. They will have head ptr values pointing to the same location in memory.**



What is wrong with this code for adding to an int array? How could you fix it?

1. It is treating arr.count as an index. It is not an index. You need to subtract one from ++arr.count to get this to behave as intended.

2. It is not accounting for the fact that the array is about to grow by one in its resize check.



**• Q. Will this function**

**void lobo(int& num, unsigned int size)**

**let us change the data for num? Will it let us change the address of num in memory? If so, which of these changes will be permanent after the function runs? Why?**

• A. Yes, lobo will allow us to change num. No, lobo will not allow us to change the address of num in memory. Any changes made to num by lobo will be persistent after lobo runs. This is because num is passed in by reference, thus the literal section of memory that num occupies will be given to lobo for manipulation.

**• Q. Will this function**

**void zobo(int num, unsigned int size)**

**let us change the data for num? Will it let us change the address of num in memory? If so, which of these changes will be permanent after the function runs? Why?**

A. Yes, zobo will allow us to change num. No, zobo will not allow us to change the address of num in memory. Any changes made to num by zobo will not be persistent after zobo runs. This is because num is not passed in by reference, thus it is only a copy of the argument which zobo manipulates.

**• Q.Will this function**

**void bobo(int\* array, unsigned int size)**

**let us change the data in an array? Will it let us change the address of the array in memory. Of these changes will be permanent after the function runs? Why?**

A. Yes, bobo will let you change the data in an array. It will let us change the address of a shallow copy of the array in memory. This copy will cease to exist on function termination. Any changes made to the data pointed to by the array pointer will be persistent. The reason bobo will letyou change the data in an array is simple: the copy of the array pointer points to the array as well. Thus we do have a way to get direct access to the array memory.

**• Q. Will this function**

**void flobo(const int\*& array, unsigned int size)**

**let us change the data in the array? Will it let us change the address of the array? If so, which of these changes will be permanent after the function runs? Why?**

A. No, flobo will not let you change the data in the array. Yes it will let you change the address of the array, and that change will be persistent. You can not change the data because the pointer points to constant data. Thus \*array = whatever is not allowed. On the other hand, array = whatever is allowed, and because the array is passed in by reference then that change will be persistent.

**• Q. Will this function**

**11void globo(int\*& array, unsigned int size)**

**let us change the data in an array? Will it let us change the size of the array? Will it let us change the address? If so, which of these changes will be permanent after the function runs? Why?**

A. Yes, globo will let us change the data in the array. Yes it will let us change the size of the array. Yes it will let you change the address of the array (and indeed this is required if you want to change the size of the array). All changes made to array will be persistent after the function runs. int\*& passes in an array by reference. Arrays are just pointers. Thus you can change the data (by following the pointer to the target and changing the target). You can also run array = whatever to change the location it points to (be careful not to leak memory), and if you run array = new int[someOtherSize] then you will change the size of the array. That said, this function has no way of returning that new size (size is passed in by value), so this is likely a bad idea.

**• Q. What is the main behavioral difference between variables on the local memory (stack) and variables on the heap?**

A. Variables on the heap are preserved after the termination of the scope in which they were defined. Stack variables are not

**• Q. How can I tell if a pointer is bad?**

A. You can tell if a pointer is not pointing to valid memory by comparison to the nullptr keyword. Actually, you can just do this: