CS 216 Exam 2

You MUST write your name and e-mail ID on EACH page and bubble in your userid at the bottom of EACH page – including this page.

If you are still writing when "pens down" is called, your exam will be ripped up and not graded – even if you are still writing to fill in the bubble forms. So please do that first. Sorry to have to be strict on this.

Other than bubbling in your userid at the bottom, please do not write in the footer section of each page.

There are 10 pages to this exam – once the exam starts, please make sure you have all 10 pages.

Pages 2-9 are worth 12 points each (for a total of 96 points), and the first and last pages are worth 2 points each. The three point questions on this exam should not take more than a line or two to answer – your answer should not exceed about 20 words. There are 100 points of questions and 1 hour 45 minutes (105 minutes) to take the exam, which means you should spend one minute per question point – the other 5 minutes are to fill out the bubble footers.

If you do not bubble in a page, you will not receive credit for that page!

This exam is CLOSED text book, closed-notes, **closed-calculator**, closed-cell phone, closed-computer, closed-neighbor, etc. Questions are worth different amounts, so be sure to look over all the questions and plan your time accordingly. Please sign the honor pledge here:

____

____

____

____

Out of memory.
We wish to hold the whole sky,
But we never will.

Older Material and Trees

1. [3 points] Why might we want to write a *method* template?

2. [3 points] Why is there so much imprecision in floating point numbers? For example, if we added 0.1 ten times, we would not get 1.0, but instead get 0.9999999999999999. What is the *mathematical* reason for this?

3. [3 points] We say that a splay tree has $\Theta(\log n)$ amortized running time for the major operations. What does the "amortized" portion of that running time mean?

4. [3 points] What is the complete algorithm for removal of a value from a binary search tree? You only need a (short!) paragraph in English to describe this – pseudo code (or regular code) is not needed.

Trees

5. [3 points] Give three possible tree applications that do not use binary trees, but instead use a different type of tree. Rephrased, give three possible tree applications for which a binary tree does not work.

6. [3 points] Consider a trinary tree, which is just like a binary tree, but there are three children per node instead of two children. A trinary tree is still ordered – the left subtree's values are all less than the center subree's values, which are all less than the right subtree's values. The value in a given node is between the left and center (or, depending on the implementation, between the center and the right). What is the running time for such a tree for the primary operations (insert, find, delete)? Briefly, why?

7. [6 points] Prove by induction that the maximum number of nodes in a binary tree of height *h* is 2^{h+1}-1.

Trees

8. [6 points] Consider the three types of self-balancing trees that we have seen: AVL, red-black, and splay. BRIEFLY give one advantage and one disadvantage of each of these trees. If it's not obvious, explain why.

9. [6 points] Show an unbalanced AVL tree (i.e. after an insert, but before rebalancing) that requires a double rotation to fix it – and try to keep the tree as small as possible. Perform the double rotation, and show the intermediate step.

Hash Tables

10. [6 points] Consider the four types of collision resolution that we have seen: separate chaining, linear probing, quadratic probing, and double hashing. BRIEFLY give one advantage and one disadvantage of each of these types of collision resolution.

11. [3 points] What is the big-theta for the insert and find operations on a hash table that uses a probing strategy for collision resolution?

12. [3 points] What properties must a good hash function have?

Hash Tables

Although there is a lot of text on this page, we will give you 12 points on your exam for reading it, as long as you fill in the bubble footer at the bottom of the page.

We have seen three probing strategies for collision resolution – while they are generally faster than separate chaining, they suffer from the drawback that a single operation can take quite some time (worst case $\Theta(n)$) if there is a large cluster of keys. We present a collision resolution algorithm developed in 1986 called *Robin Hood Hashing* (it's real name). The actual algorithm is based off of double hashing, but we will base it off of linear probing to make it simpler for this exam.

Robin Hood hashing operates like a normal linear probing algorithm. However, after a collision, as we are 'probing' for a spot to insert the collided key, we pay attention to the distance that the collided key is from its original hashed spot, as well as the distance that the value in each successively probed spot is from its original hashed spot. Whichever key has the *lower* distance is moved forward. Thus, if the collided key probes a particular spot, and the key in that spot has a lower distance (from its original hashed spot) than the collided key, then the collided key goes into that spot, and the key that was in that spot is moved to probe the next spot to repeat this algorithm. On a tie (of distance to the respective original hashed spots), the collided key is moved forward. The name 'Robin Hood' comes from the fact that the wealth is being shared – here, the distance to the hashed value is being kept as short as possible (the wealth is the shortest distance to the original hashed spot). The purpose of this algorithm is to keep the amount of movement through a cluster (on a find, for example) down to a minimum.

As an example, consider the case to the right where a (partial) hash table has two elements inserted into it at spots 15 and 16. We'll assume, for simplicity, that the key is the spelled-out word at that particular index. Neither of the two elements had a collision when inserted – they hashed directly to the spot that they appear in the table.

…	
15	Fifteen
16	Sixteen
17	
…	

Next, the insertion of "Fifteen-point-one" occurs, and (not surprisingly), it hashes to spot 15, which causes a collision. In a normal linear probing collision resolution, it goes into spot 17. As this is Robin Hood hashing, we check the distance to the original hashed spots (we start at index 15) – both hash to the same spot (also 15), so on a tie, we move the collided key ("Fifteen-point-one") forward. On the next spot (index 16), there is also a collision. When we examine the key in spot 16, we note that the key "Sixteen" is zero spots away from its hash, and Fifteen-point-one is one spot away – thus, we move *Sixteen* to the next spot. Thus, the final hash table looks like the table to the right.

…	
15	Fifteen
16	Fifteen-point-one
17	Sixteen
…	

(the bubble footer is automatically inserted in this space)

13. [12 points] Hash my TAs! I can't keep track of my teaching assistants, so I want to put them into a hash table. Below you will see a hash table of size 9 (nine, not ten!), and a list of the TAs and their hash codes, which have already been computed for a hash table of size 9. Hash the TAs into the hash table following the Robin Hood hashing algorithm described above. You must hash them in the order shown below (i.e., top to bottom)!

TA	**Hash code**
Briana	7
Dan	2
Jeff	4
John	0
Josh	2
Randolph	2
Wil	3

0	
1	
2	
3	
4	
5	
6	
7	
8	

---

14. [3 points] What are the advantages of Robin Hood hashing versus linear probing?

15. [3 points] What are the disadvantages of Robin Hood hashing versus linear probing?

16. [3 points] What is the running time of an insert in Robin Hood hashing? Why?

17. [3 points] What is the running time of a find in Robin Hood hashing? Why?

IBCM

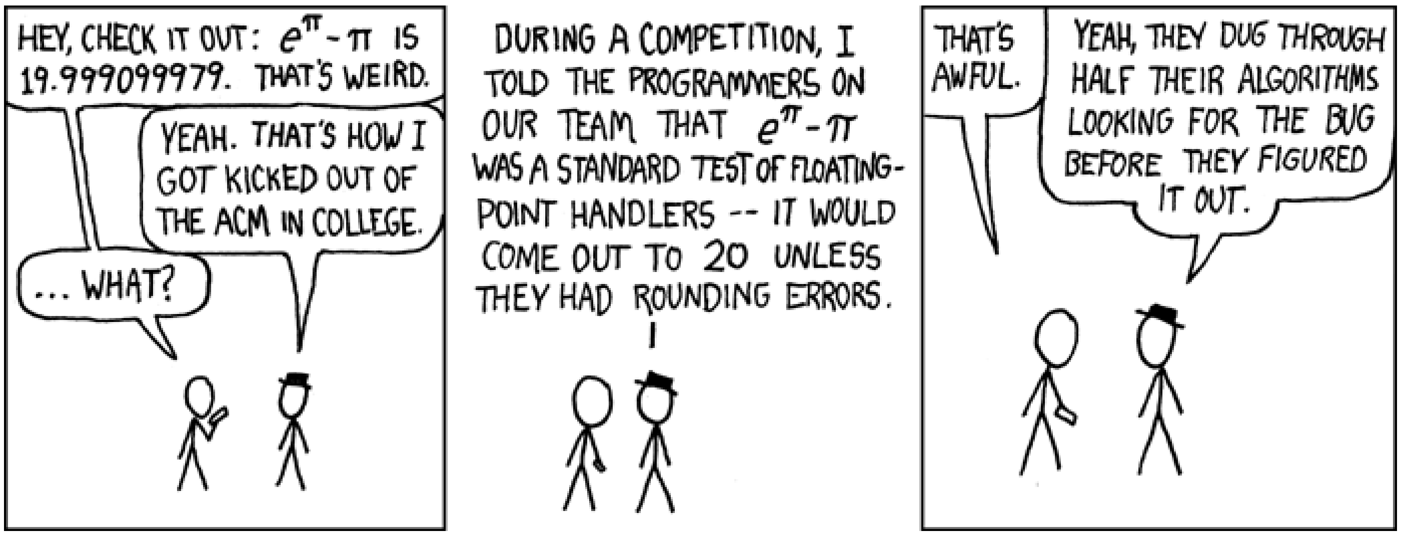
18. [12 points] Write an IBCM program that will read a positive number from the keyboard, convert it into the 2's complement version of its negation (i.e. 17 goes to -17), and print out the result. The only restriction is that you may NOT do this via subtraction (i.e. using the `sub` opcode, adding a negative number, or similar) – you need to implement the program to negate the number by making the necessary binary modifications. As a hint, you will need the `not` opcode. The number that you read in may be either positive or negative (this shouldn't affect your algorithm). Note that the number is read in (and written out) in hex, but the number will still be the hex value for a 2's complement number. Your answer should have both the hexadecimal machine code and the opcodes – you can include just the latter for partial credit.

IBCM

0	halt
1	I/O
2	shifts
3	load
4	store
5	add
6	sub
7	and
8	or
9	xor
A	not
B	nop
C	jmp
D	jmpe
E	jmpl
F	brl

This page unintentionally left blank.

But you get two points for bubbling in the form at the bottom of this page. Woo-hoo!


HEY, CHECK IT OUT: $e^{\pi} - \pi$ IS 19.999099979. THAT'S WEIRD.
YEAH. THAT'S HOW I GOT KICKED OUT OF THE ACM IN COLLEGE.
... WHAT?
DURING A COMPETITION, I TOLD THE PROGRAMMERS ON OUR TEAM THAT $e^{\pi} - \pi$ WAS A STANDARD TEST OF FLOATING-POINT HANDLERS -- IT WOULD COME OUT TO 20 UNLESS THEY HAD ROUNDING ERRORS.
THAT'S AWFUL.
YEAH, THEY DUG THROUGH HALF THEIR ALGORITHMS LOOKING FOR THE BUG BEFORE THEY FIGURED IT OUT.

(the bubble footer is automatically inserted in this space)