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 8 pages to this exam – once the exam starts, please make sure you have all 8 pages.

Questions are worth different amount of points, from short answer questions worth 4 points each to long answer questions worth 16 points each. The short answer questions 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 on the exam and 1 hour 35 minutes (95 minutes) to take the exam, which means you should spend just under one minute per question point.

If you do not bubble in a page, you will not receive credit for that page! If the page is worth zero points, then you will receive a grade penalty.

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:

____

____

____

____

The Tao that is seen
Is not the true Tao,
until you bring fresh toner.

Trees

1. [4 points] Give a *brief* description of how the `splay()` operation works for splay trees. How does it raise the given element to the top?

2. [6 points] Name one advantage and one disadvantage of each of the three types of self-balancing trees that we have seen: splay, AVL, and red-black.

3. [6 points] You have to implement binary trees in a new programming language (perhaps IBCM). Unfortunately, this programming language does not support pointers or references – only arrays. How would you go about structuring your binary tree to store it in an array? In particular, what we are looking for here is a *brief* description of how the binary tree data type would work with various operations.

---

4. [16 points] An *(a,b)-tree* is a tree data structure where each internal node has between *a* and *b* children – meaning it can't have fewer than *a* children, and it cannot have more than *b* children. Leaf nodes, of course, are exempt from this rule. The only restriction is that $2 \leq a \leq (b+1)/2$ (which implies that *a*<*b*). Thus, a (2,3)-tree is a tree where each node is either a leaf, has two children, or has three children. You are not expected to have heard of this data structure before this exam – that's the point of the question.

   Consider how you would implement each node of a (2,3)-tree. What fields (and what types of those fields) would go into each node of a (2,3)-tree? You can show your answer in either pseudo-code or C++, but be sure to indicate what the field names and their types are.

   The binary search tree property holds for an (a,b)-tree, although it has to be modified in the case of three nodes – in this case, the left child sub-tree is less than the center child sub-tree, which is less than the right child sub-tree. Furthermore, another property of all (a,b)-trees is that all paths from the root to any leaf are of the same length (i.e. it's a *perfect* tree). Given these properties, what would the running time be for a find operation? Briefly, why?

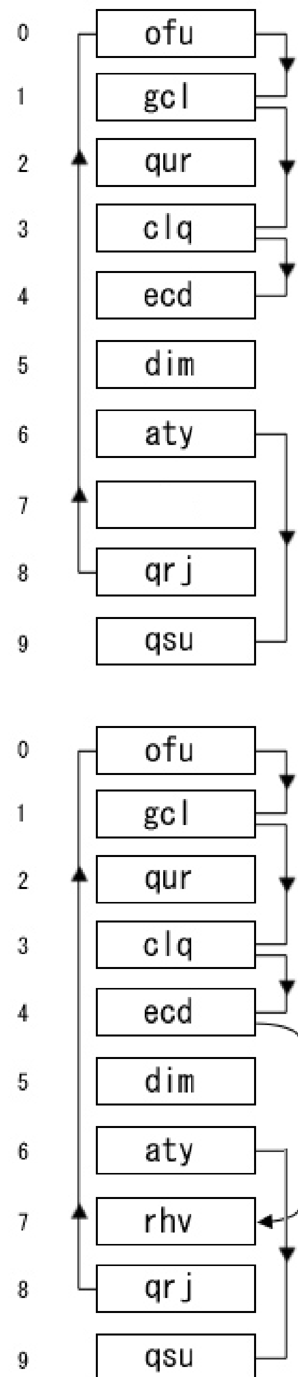
   Given the perfect tree property for (a,b)-trees from above, inserting a new value that is a sub-child of a node that only has *a* children is trivial – the node can hold the new element as a child since *a*<*b*. But if the node is full (i.e. has *b* elements), then the tree needs to be restructured. What is the running time, then, of an insert in this (worst) case? Why? The why part will require a *sketch* (i.e. a *brief* description!) of the algorithm to insert, not a full-blown write-up of the algorithm (if you do that, we'll deduct points).

Hashes

The next few questions on hashing will need the information this page.

We have seen four collision resolution algorithms in this course – separate chaining and three probing strategies (linear, quadratic, and double hashing). There are at least a half dozen more. The one we care about for this question is called *coalesced hashing*.

Coalesced hashing is a hybrid of probing and separate chaining. Each spot in the array can hold a (key, value) pair, as in any hash table with probing. As with the collision resolution techniques presented in lecture, we will only talk about placement of the keys in the hash table. In addition, each spot in the array also holds a pointer. That pointer will point to the next element in the bucket, as in separate chaining. However, the buckets are kept *in the array itself*, rather than in a separate data structure. The top diagram to the right shows an almost full (spot 7 is empty) hash table of random 3-letter strings with coalesced hashing. During an insert, if the spot is empty (such as in spot 7), then the element goes in that spot, as in any probing strategy. If the spot is full, the insert will follow the chain of elements until it ends up at the last element in the chain. If there aren't any pointers leading out of that spot – such as in spots 4 and 5 (among others), for example, then you don't end up going anywhere. The next empty spot in the hash table array (found by iterating down through the hash table array) is then used as the position to insert the value, and the pointer from the current spot is updated to point to the new spot.



For example, we'll insert 'rhv' into the top hash table. Let's assume it hashes to location 1. The pointers are followed until it reaches the end of the chain ('ecd' at spot 4). The next spot in the hash table is then found (at location 7 – the only spot available in this example). The value is inserted into this spot, and the pointer from 'ecd' in location 4 is updated to point to 'rhv' in location 7. The result is the bottom hash table – note that the newly assigned pointer is diagramed as a curved arrow to make it easier to show.

The next five questions (on the next page) deal with a hash table that uses coalesced hashing.

---

5. [4 points] What are the advantages of coalesced hashing over separate chaining? Over the probing strategies (you can group all the probing strategies together for this part of the question)?

6. [4 points] What are the disadvantages of coalesced hashing over separate chaining? Over the probing strategies (you can group all the probing strategies together for this part of the question)?

7. [4 points] What is the running time of `find()` and `insert()` for coalesced hashing? Briefly, why?

8. [4 points] How might you handle the `delete()` function for coalesced hashing?

9. [4 points] Considering all of this, give an example of when you might want to use coalesced hashing.

IBCM

10. [16 points] How would you go about implementing a stack in IBCM, similar to the way the stack works in x86? We are looking for an explanation of how the stack (and stack pointer, etc.) would work. You should include the IBCM code for the `push()` and `pop()` operations of the stack, but leave that code as IBCM opcodes – don't bother translating it to hexadecimal machine code. Note that `push()` and `pop()` do not need to be separate subroutines – they can be a list of IBCM commands that are reproduced to push and pop elements from the stack. We aren't worried about the base pointer or calling conventions for this question – just how the stack operates.

IBCM
halt
I/O
shifts
load
store
add
sub
and
or
xor
not
nop
jmp
jmpe
jmpl
brl

---

x86

11. [16 points] Prior to Apple Macintoshes moving to x86 chips in 2006, they used 32-bit PowerPC chips. Among its other features, the PowerPC chip has 32 registers, named r1 to r32, as well as a stack pointer (sp) and a base pointer (bp).

    You will need to design a C-style calling convention for the PowerPC. Similar to the x86, there is also a stack that operates in the same fashion as what we have seen in lecture. We will assume that all the opcodes we are familiar with in x86 also work on the PowerPC. Your calling convention should include four parts: the prologue and epilogue for the parent (the caller), and the prologue and epilogue for the subroutine that is called (the callee).

    In particular, there are a few things that you need to design into your calling convention:

    - Memory is slow, and we want to utilize as little memory (i.e. the stack) as is feasible.
    - Your calling convention MUST support recursion, of course.
    - This calling convention may be used by many languages, not all of which will have Java-style bounds checking. We want to design a calling convention that minimizes (or eliminates!) the chance for a buffer overflow attack.

UNIX

12. [4 points] What are shell scripts good for? What are they not good for?

13. [6 points] How well do you remember GDB? Let's find out! List all the GDB commands that you know (if you hit a dozen, you're more than fine). For each command, list what the abbreviation means (one or two words here is sufficient).

14. [6 points] Makefiles! Briefly define: target rule, suffix rule, and dependency. If your definition is more than a line or two, you are writing way too much, and we will start to deduct points.