Here are exercises from old exams that cover material we’ve encountered recently in CS 61BL.

Hashing

Problem 1
Recall that an earlier lab assignment asked for a hash function for Tic-Tac-Toe boards, defined using a class Board with a private data member myBoard declared as a two-dimensional array of char. ACS 61B student suggests the following code. (There are 19683 possible Tic-Tac-Toe boards.)

```java
public int hashCode () {
    int sum = 0;
    for (int row=0; row<3; row++) {
        for (int col=0; col<3; col++) {
            sum = sum + row*col*myBoard[row][col];
        }
    }
    return sum;
}
```

Is this a good hash function or a bad one? Explain (be specific).

Sorting

Problem 1
Any sorting method that does binary comparisons of key elements requires at least $O(n \log_2 n)$ comparisons in the worst case, where $n$ is the number of elements to be sorted.

True or false? ______  Explanation?

Problem 2
The method TopK on the next page uses a variant of the Quicksort algorithm to return copies of the $k$ largest items in this list. It uses the declarations and definitions given below.

```java
private class ListNode {
    // nested class inside List
    private Comparable myFirst;
    private ListNode myRest;
};

// Precondition:
//  This list is initialized and not circular.
// Return:
//  The number of elements in this list.
public int size ( );
```
// Precondition:
// This list is initialized, nonempty, not circular, and contains no duplicates.
// Postcondition:
// Let the first element of this list be named divider.
// smalls is a list of copies of all elements in this list that are less than divider;
// larges is a list of copies of all elements in this list that are greater than divider.
// This list is unchanged by this method.

private void partition (List smalls, List larges);
// Precondition: l1, l2, and l3 are initialized noncircular linked lists.
// Return:
// A list consisting of the elements of l1 in order, followed by the elements
// of l2 in order, followed by the elements of l3 in order.
// The lists l1, l2, and l3 are unchanged by this method.

public static List concatenate (List l1, List l2, List l3);

Part a

Fill in the blanks in the TopK method, each with a method call (possibly nested) or a single variable name, so that TopK performs as intended.

// Precondition:
// This list is an initialized noncircular list.
// Return:
// A list consisting of the k largest elements in this list.
// The order of the elements within the list returned is unspecified.
// This list is unchanged by this method.

public List topK (int k) {
    if (k == 0) {
        return new List ( );
    } else if (k == size ( )) {
        return new List (this);
    } else {
        List smalls = new List ( );
        List larges = new List ( );
        partition (smalls, larges);
        List divider = new List ( );
        divider.add (myHead.myFirst);
        if (k < larges.size ( )) {
            
        } else if (k == larges.size()) {
            
        } else { // k > larges.size ( )
            
        }
    }
}

Part b

What is missing from TopK’s precondition? List below whatever additions are necessary to ensure that TopK returns a result without crashing.
Problem 3

Background
The station manager/CEO of a radio station using the record album data base program decides to publish a “top 40” listing, once each month, of the forty songs that got the most requests that month, sorted by the number of requests they received. To produce this listing, she decides to store with each song the number of requests it has received so far during the current month. She also adds code to handle the command

request song

to request a particular song; this code searches for the title in the data base, resolves ambiguities if necessary (between two songs with the same title), and increments the request count for the song. Finally, she adds code to examine all the counts at the end of each month to produce the top 40 listing.

For this problem, you will analyze alternatives for examining all the request counts to produce the top 40 listing.

Part a
Analyze the worst-case execution time for the following alternatives for producing the top 40 listing from a data base with $N$ songs. Your analysis should be in terms of $N$ and the number of comparisons, data movements, and similar operations used by each algorithm. Use big-Oh notation only if you are unable to determine the exact number of these quantities. Assume that all operations whose details are not specified are coded to run as quickly as possible.

A. Construct a linked list of all the songs, then use a variant of the Quicksort algorithm similar to what you just programmed for the previous problem to find the forty songs with the most requests. Then sort these forty songs using insertion sort.

B. Construct a linked list of all the songs, then run selection sort for forty iterations to get the top forty songs in order at the front of the list.

C. Form a heap ordered by request count from all the songs, with the top of the heap having the largest value. Then remove the top of the heap forty times, adding each song removed to the end of an initially empty 40-element song array.

Problem 5
Given below are traces of the execution of three sorting routines used to sort the characters AFBECD into alphabetical order. Trace output was produced on entry to the sorting routine, at places where the contents of the array changed, and at other interesting points in the sorting process. Identify each trace with the name of the algorithm used to create it. The methods traced are among the following: bubble sort, insertion sort, selection sort, shell sort (5-3-1), quicksort (pivoting on the first item; otherwise as given in Carrano), and heapsort (as given in Carrano). (Note: Shell sort was not covered in CS 61BL this semester.)
Trees and heaps

Problem 1
List below as many different nonnull binary trees as you can that are both max heaps and binary search trees. “Different” means different in shape as opposed to key values, unlike the two binary search trees below. Assume that the largest value is at the top of the heap and at the rightmost position in a binary search tree.

```
a. AFBECD
   AFBECD
   ADBECF
   ADBCEF
   ACBDEF
   ABCDEF

b. AFBECD
   ABFECD
   ABFEDC
   ABEFCD
   ABCEFD
   ABCDEF

   AFBECD
   AFEDCB
   FAEDCB
   FDEACB
   EDBACF
   DCBAEF
   CABDEF
   BACDEF
   ABCDEF
```

Problem 2
Given on the right is a tree that represents a heap whose largest element is at the top. Circle all the nodes of the tree that could possibly contain the *smallest* value in the heap.

```
3
/   /
1   4

5
/   /
9
/   /
7
/   /
5
/   /
9
```

```
3
/   /
1   4

5
/   /
9
/   /
7
/   /
5
/   /
9
```

```
3
/   /
1   4

5
/   /
9
/   /
7
/   /
5
/   /
9
```
Problem 3
Your lab partner suggests the following method for printing the elements of an array in ascending order. First, make the array into a heap (whose maximum element is the top element). Then, since the top of the heap is the largest element, traverse the heap in postorder (traversing the left subtree before the right), printing each node value as it is encountered in the traversal.

Part a
Assuming the array contains no duplicate values, does this algorithm always print the array elements in ascending order?

Part b
If your answer to part a was yes, give the order of the algorithm’s running time, briefly explaining how you got your answer. If your answer to part a was no, give an array and draw the resulting heap for which the given algorithm does not print the elements in ascending order.

Problem 4
Describe an algorithm that uses a heap of size k+1 to collect the k largest elements in a list of length n in time proportional to n log k. Your description should be in detail sufficient for another CS 61B student to recognize how your solution should be implemented in Java.

Problem 5
Consider a tree of N nodes in which each node has 0 or 2 children, and the value in each node with children is greater than the value in its left child and less than the value in its right child. Finding a particular value in this tree requires, on the average,

a. a constant number of comparisons
b. order log N comparisons
c. order N comparisons
d. order N log N comparisons
e. order N² comparisons

Problem 6
Finding and deleting an item from a heap of N elements (assuming no additional information) requires, on the average,

a. a constant number of comparisons
b. order log N comparisons
c. order N comparisons
d. order N log N comparisons
e. order N² comparisons
Problem 7
Suppose a heap is organized with the smallest item at the top. Finding the \textit{largest} item in the heap requires at worst
\begin{enumerate}[a.]
  \item a constant number of comparisons
  \item order $\log N$ comparisons
  \item order $N$ comparisons
  \item order $N \log N$ comparisons
  \item order $N^2$ comparisons
\end{enumerate}

Problem 8
Suppose we have a hash table of $N$ cells, and all keys that hash to a given cell are stored in a balanced binary search tree for that cell. Suppose also that the table contains 2000 keys. Finding a given item in the table requires, on the average, approximately
\begin{enumerate}[a.]
  \item $\frac{2000}{N}$ comparisons
  \item $\frac{2000}{(\log N)}$ comparisons
  \item $\log \left(\frac{2000}{N}\right)$ comparisons
  \item $\log \left(\frac{N}{2000}\right)$ comparisons
  \item $\log 2000$ comparisons
\end{enumerate}

Problem 9
A programmer on your staff presents you with the following algorithm to create a balanced binary search tree out of a singly linked sorted list of $N$ elements in order $N$ operations. The algorithm assumes that each node in the list contains only the list element and the pointer to the next list element, and that each node in the tree contains the corresponding tree element and two child pointers.

Algorithm
\begin{enumerate}[1.]
  \item Find (in the obvious way) the node of the sorted list that contains the median element, and make it the root of the balanced tree.
  \item Make a balanced tree—using this algorithm—out of the list of elements that precede the median, and make that tree the left subtree of the root.
  \item Make a balanced tree—using this algorithm—out of the list of elements that follow the median, and make that tree the right subtree of the root.
  \item Return a pointer to the root of the tree.
\end{enumerate}

Is the programmer’s claim correct? That is, does the algorithm described above run in linear time, and does it produce a balanced binary search tree? Justify your answer.
Problem 10
Suppose you wished to use a heap to simulate a stack of characters; that is, the pop operation would be implemented by removeTopOfHeap, and push would be implemented by insertIntoHeap. Explain how you would do this, and what the running times are for the push and pop operations.

Problem 11
Write a method threadLevels which, when given a pointer to a binary tree, threads the levels of that tree from left to right within a level. Tree nodes are instances of a nested class declared as follows:

```java
private class TreeNode {
    public Object myItem;
    public TreeNode myLeft;
    public TreeNode myRight;
    public TreeNode levelList;
};
```

Your method may call auxiliary methods. The depth (i.e. the number of levels) of the tree does not exceed MAXDEPTH, a constant, so threadLevels may also use an auxiliary array.

The diagram below provides an example of the structure that threadLevels should create. The fourth (shaded) cell in each node is the levelList pointer. On entry to threadLevels, the value of each levelList pointer is undefined.

Problem 12
Describe, in pseudocode detailed enough to be translated easily to Java code, an algorithm that determines if a given binary tree of integers is a binary search tree whose left subtree contains the smaller values. Your algorithm should take, in the worst case, time proportional to the number of items in the tree.
Problem 13
Consider a binary tree representing an arithmetic expression. Each internal node of the tree represents one of the operators + and *. Each leaf of the tree represents either an integer constant or a variable whose value is indeterminate. Examples appear below.

Such a tree may be implemented in Java with the following declarations:

```java
public class Expression {
    private class TreeNode {
        public String myName;
        public TreeNode myLeft;
        public TreeNode myRight;

        public TreeNode (String s, TreeNode left, TreeNode right) {
            myName = s;
            myLeft = left;
            myRight = right;
        }
    }
}
```

Each internal node of the tree has name equal to “+” or “*”. In each leaf of the tree, either
- name is the string representation of an integer constant; or
- name is some other identifier, not equal to “+” or “*”, representing a variable of indeterminate value.

You are to write a `Expression` method named `simplify` that simplifies expressions where possible. Your method should replace each subexpression whose value can be determined by a constant expression with that value, deleting the replaced `TreeNode` from the tree. For example, your method should convert the sample trees on the previous page to
(no simplification possible)  

\[
\frac{-3}{\times} 5\quad x
\]

\[
\left(3 + \frac{-5}{5}\right)
\]
simplified to 5

// Precondition: the Expression object is correctly constructed.
// Postcondition: the Expression object is simplified as described on the previous page.

```java
public void simplify ( ) ...
```

**Problem 14**

Consider a Tree class whose nodes are instances of a nested class declared as follows:

```java
private class TreeNode {
    public Object myItem;
    public TreeNode myLeft;
    public TreeNode myRight;
    public TreeNode levelList;
};
```

On the next page, write a Tree method named prune that removes from the tree all nodes that have only one child. For each such node, the tree should be restructured as shown in the example below—i.e. the node’s child should become the node’s parent’s child—and the node deleted from the tree. If the root of the tree has only one child, it should be replaced in the same way.

*before pruning*  

*after pruning*  

Note that a node with only one child may be either the left or the right child of its parent—or even the root as just mentioned. Also, such a node may have 0, 1, or 2 grandchildren.

```java
public void prune ( ) ...
```
Problem 15

Part a

The representation in lab assignment 11 used an array to store the integer elements of a max heap. Describe the implementation of a max heap as a linked structure that stores one integer per node; clearly explain the contents of each node, and of whatever auxiliary pointers you use. Your implementation should allow the “insert into the heap” and the “remove the largest heap item” to be performed in time proportional to log(N), where N is the number of items in the heap.

Part b

Given below is the code for heap insertion from lab assignment 11. For each indicated statement or group of statements, describe (in terms suitable for another CS 61B student to understand how it would be translated into code) how it would work with your implementation. Assume for the purposes of this problem that the heap contains at least three elements prior to insertion.

code from lab 11

```java
values[size] = newValue;
size++;

for (int k=size-1; k>1; k=(k-1)/2) {
    if (values[k] < values[(k-1)/2]) {
        break;
    }
    int temp = values[k];
    values[k] = values[(k-1)/2];
    values[(k-1)/2] = temp;
}
```

Data structure design and analysis

Problem 1

Which of the following would argue for using a binary search tree rather than a hash table to store a set of keys? (Answer yes or no in each blank; yes indicates that the condition argues for use of a binary search tree.)

a. the number of keys to be stored is large;
b. memory must be used as efficiently as possible;
c. one of the operations we must support is “find the successor of the given key“;
d. each key has a single abbreviation that must be recognized as identical to the key;
e. the key type is itself a binary tree.

Problem 2

[Note to CS 61BL students: This problem refers to a “directory manager” program that was the subject of a case study.]
The C shell has a “command completion” feature that works as follows: if you type part of a file name and hit the ESCAPE key, the shell searches your working directory for files whose names begins with the characters you’ve typed. If there is exactly one such file, the shell provides the rest of its name, otherwise it beeps.

To add such a feature to the Directory Manager, you decide to provide a method called findMatch with prototype

```java
public FileEntry findMatch (String s);
```

If there is exactly one file in the directory whose name begins with the characters in s, findMatch returns a pointer to its file entry; otherwise findMatch returns null.

Describe how the directory should be structured to make findMatch as well as other directory operations execute quickly for a directory containing up to 10000 files. Your description should be in detail sufficient for another CS 61B student to recognize how your solution should be implemented in Java. Since the Directory Manager is a system program, your structure should not make extravagant use of memory.

**Problem 3**

**Background**

Consider a browser to be used with large files in which the number of different “words” is small. This browser will have four commands:

- `locate word1 word2 word3`
- `locate word1 word2`
- `locate word1`
- `locate`

Search for the next occurrence of the given sequence of words, starting at the position of the most recent selection or at the start of the file if nothing is selected. If the given sequence of words is not found, signal an error; otherwise display a windowful of text that starts with the given sequence of words. There are at most three arguments, words to be matched exactly by words in the file. Thus `locate` is less like the `find` command from homework assignment 10 than like a generalized version of the `search` command. The `locate` command given with no arguments searches for the next occurrence of whatever sequence of words is selected.

- `back locate word1 word2 word3`
- `back locate word1 word2`
- `back locate word1`
- `back locate`

(like `locate` except that it searches backward in the file).

No other commands are to be provided in this browser. Line boundaries in the file are not meaningful; thus in the terminology of homework assignment 10, each file is one large “paragraph”.

---

**CS 61BL (Clancy)  Spring 2007  Sample exam problems**

---

11
Part a
Provide annotated class definitions for a structure in which the words of the file can be stored to allow the locate and back locate commands, particularly those with multiple arguments, to be executed as fast as possible on average. (Note that there are likely to be many occurrences of a given argument sequence.) Don’t worry about memory constraints, or about time to initialize your structure.

Your annotations should provide enough information for us to determine unambiguously how the contents of a given file would be stored and how the structure would be used to process a locate or back locate command. A diagram will probably be appropriate.

Part b
Determine how many bytes of memory will be required in the data structure you just designed for a file containing the following eight words:

```
A  B  A
A  B  B  A  B
```

Show all the computations you make to find the answer. As in homework assignment 10, count 4 bytes for each pointer or integer and 1 byte for each character.

Part c
Give an estimate for how long it should take to process a locate command with three arguments, i.e. the order of the running time of such a command. Make it clear what quantities (for example, the number of words in the file, or the number of occurrences of the argument sequence) your estimate depends on. Show all your work.

Problem 4
In this problem, you compare two structures for storing a list of 25,000 words. All the questions for this problem ask for behavior “on the average”; for the purpose of this problem, you should assume that all the words will be accessed equally often.

Part a
Consider a hash table with $n$ cells, each of which contains a pointer to an unsorted linked list of words that hash to that cell. Suppose 25,000 words are to be added to the hash structure, and the hash function is good enough to distribute them evenly throughout the table (that is, all the $n$ lists contain the same number of words).

How many comparisons, on the average, are needed to access a word in the hash structure? Give your answer in terms of $n$, and show how you got it. The formula

$$1 + 2 + 3 + \ldots + k-1 + k = \frac{k(k+1)}{2}$$

will be useful. For the purposes of the computation, assume that $n$ evenly divides 25,000.
Part b
Another way to store the 25,000 words is in a binary search tree. The internal path length of a binary search tree of minimal height that contains 25,000 entries is around 334,000. How many comparisons, on the average, are needed to access a word in this data structure? An approximate answer is fine; show how you got it.

Part c
How large must \( n \) be for the hash table to allow, on the average, faster access to the 25,000 words than a binary search tree provides? (Faster access means fewer comparisons.) Again, an approximate answer is fine. Show your work.

Problem 5

Background
Consider the problem of writing an index for the words in a file of formatted text (like the kind found in the backs of textbooks). The file of input text contains a sequence of words. Within the text are special characters marking page boundaries. The index produced from the file should be an alphabetized list of lines, each containing a word from the text and the numbers of the pages on which that word appears. (Assume for the purposes of this problem that all words are to be indexed.) The page numbers on a line should be all distinct and in ascending order. An example of three pages and a portion of the resulting index appears below.

```
... backspacing
  boldface ...
...
backspacing ...
... backspacing ...
  page 44

... boldface ...
  ... backspacing
    buffer ...
      ... boldface ...
  page 45

... boldface ...
  page 46

backspacing  44, 45
bit  66
boldface  39, 44, 45, 46, 81
buffer  45
```

Part a
Describe the classes and list their methods and parameters for a solution to the indexing problem. (Make sure the methods are named well enough for us to tell what they do.)
You should treat operations on words (e.g. reading a word and its page number, comparing two words, adding a word to or looking up a word in the index) as primitive operations in your decomposition. (That is, you shouldn’t decompose these operations.) Your decomposition should be flexible enough to allow for different representations for the index data structure, for instance a binary search tree or a hash table.

**Part b**
For each of the methods you listed for part a of this problem, describe what it would do and what other methods it would call if the index data structure were represented as a hash table. As in part a, you don’t need to decompose operations on words.

**Part c**
For each of the methods you listed for part a of this problem, describe what it would do and what other methods it would call if the index data structure were represented as a binary search tree. You need only describe differences from part b.

**Part d**
Assume that you are implementing the index data structure as a hash table. Describe the steps in which you might implement and test the methods you described in parts a and b, and the test data you would use at each step, in order to show most convincingly that your methods work.

**Problem 6**

**Background**
This question is inspired by the game of Tic-Tac-Toe. We wish to store a set of possible configurations of the game. If Tic-Tac-Toe is new to you, a *configuration* is simply a 3x3 array whose cells are either blank, X, or O with the restriction that the number of X’s is either equal to the number of O’s or exceeds it by 1. (We think there are 19683 such configurations.)

**Part a**
Design a data structure for a set of Tic-Tac-Toe configurations that supports the operations `member?` and `insert`, and uses as little *memory* as possible. Speed of operation is only a secondary concern. Provide pseudocode for your implementation of `member?`, `insert`, and the initialization operation for your data structure. Also analyze how much storage you will need to store N configurations and how long it will take in the worst case to access one.

**Part b**
Design a data structure for a set of Tic-Tac-Toe configurations that supports the operations `member?` and `insert`, and retrieves and inserts as quickly as possible. Memory use is only a secondary concern. Provide pseudocode for your implementation of `member?`, `insert`, and the initialization operation for your data structure. Also analyze how much time you will need to access a configuration in the worst case, and how much storage you will need to store N configurations.
Problem 7
Consider a composite data structure to represent an inventory of items, declared as follows.

```java
public class Inventory {
    private LinkedList<Item> sortedByCost;
    private HashSet allNames;
    private class Item {
        public String myName;
        public int myCost;
    }
    ...
}
```

Items are identified uniquely by name; i.e. there shouldn’t be two items with the same name. Elements of the `sortedByCost` list appear in decreasing order by cost, except that if there are more than one item with the same cost, those items appear in the list alphabetically by name. The corresponding items should also all be represented in the hash table.

To aid debugging, one might devise a method that checks that the inventory structure is internally consistent. Such a method would check that the lists are correctly constructed, i.e. that there aren’t any circular pointers and that the size is equal to the number of items in each list. It would also check that `sortedByCost` is correctly sorted. For this problem, you are to describe how to implement two additional consistency checks:

Duplicate check:
    no two items represented in `sortedByCost` should have the same name.

Correspondence between components:
    Every item represented in `sortedByCost` should also be represented in `allNames`, and vice-versa.

Part a
Describe an implementation for performing the operations labeled as “Duplicate check” and “Correspondence between components” on the previous page. Each consistency check should be performed as efficiently as possible on the average. You may use additional structures and should not change the existing contents of the inventory. Your description should specify what additional data structures you use, what they contain and, if your data structures include arrays, how big the arrays are. Your description should be in terms suitable for another CS 61B student to understand immediately how it would be translated into code.

Part b
Give an estimate of the running time needed for each step involved in your answer to part a. Be specific about the quantities your estimates involve.
Problem 8

Describe the implementation of a generalized vector class called GenVector. Its constructor takes as argument a list of the integers to be used as legal subscripts for the GenVector object. For example, passing the constructor a list containing the integers –5, 42, and 363 should set up a three-element GenVector.

Your implementation should allow fast execution of the following operations for large subscript sets:

• retrieving the value associated with a given index (the [ ] operation for regular Vectors);
• replacing the value associated with a given index by a new value (the operation analogous to assignment with elements of regular Vectors)
• detection of whether a given index value is a legal subscript for the GenVector object.

It should also not use an extravagant amount of memory when the subscript set is small.

Your description should specify what data structure is used to implement the generalized vector, what it contains and, if your data structure includes an array, how big the array is. You should also explain how the pseudocode expression a[k] would be evaluated for a given generalized vector a and integer k. Your description should be in terms suitable for another CS 61B student to understand how it would be translated into code.