Name_______________________

CS 32
Winter 2010
Final Exam
March 13, 2010

Problem #	Possible Points	Actual Points
1	10	
2	25	
3	20	
4	10	
5	15	
6	20	
TOTAL	100	

STUDENT ID #: _______________________

SIGNATURE: _______________________

INSTRUCTOR (check one):

[ ] Nachenberg (Lecture 1)

[ ] Smallberg (Lecture 2 or 3)

OPEN BOOK, OPEN NOTES
NO ELECTRONIC DEVICES

1. [10 points in all]

Assume the usual precedence and associativity of the arithmetic operators.

a. **[5 points]** Convert the following infix expression to postfix. Examining the expression from left to right, show the complete contents of the operator stack after the processing of each operator, each parenthesis, and the end of the expression. (The stack grows from bottom to top.)

```
A / ( B + C ) - ( D - E + F ) / G
```

Answer:

		+								
		(								
/		/								
after /	after (	after +	after)	after -	after (	after -	after +	after)	after /	after end

b. **[3 points]** Convert the following postfix expression to infix. Do not use any unnecessary parentheses.

```
A B C * D - E F G * / + *
```

Answer:

c. **[2 points]** Evaluate the following *prefix* expression using integer arithmetic. The answer is a single integer.

```
+ / * 2 - 8 2 4 * + 1 0 - 6 2
```

Answer:

2. [25 points in all]

a. [5 points] Here is an algorithm that sorts an `n`-element integer array `a` in non-descending order.

```
for (int j = 1; j < n; j++)
{
    const int v = a[j];     // breakpoint
    int k;
    for (k = j; k > 0  &&  v < a[k-1]; k--)
        a[k] = a[k-1];
    a[k] = v;
}
```

Suppose that after some iterations, execution reaches the line indicated by `breakpoint`, with the array looking like this:

[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]
1	2	3	6	7	5	4	8

What is the largest value `j` could possibly have right now? **Answer:**

b. [5 points] What is the output of the following program segment:

```
stack<int> s;
s.push(10); s.push(20); s.push(30);
queue<int> q;
while (!s.empty())
{
        q.push(s.top());
        s.pop();
}
while (!q.empty())
{
        s.push(q.front());
        q.pop();
}
while (!s.empty())
{
        cout << ' ' << s.top();
        s.pop();
}
```

Answer:

c. [5 points] Suppose there is a tree (not necessarily a binary tree) each of whose nodes is labeled by a letter. A preorder traversal of the tree visits the nodes in the order `A S R T O I E D`, while a postorder traversal visits them in the order `R S O I T D E A`. Draw the tree.

Answer:

d. **[5 points]** We're going to sort a 9-element integer array in *descending* order using the quicksort algorithm. A pivot was selected, the array was partitioned, and the pivot placed in its proper position. Then the algorithm was called recursively to sort the *left* subarray. A pivot was selected, that subarray was partitioned, and the pivot placed in its proper position. At this moment, the array looks like this:

[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]
70	90	80	60	40	50	30	10	20

Which element must have been the first pivot selected? Which must have been the second? (Give us the values of the elements, not their positions.)

Answer: **First pivot selected:**

Second pivot selected:

e. **[5 points]** Sort the characters in the array below in ascending alphabetical order using the heapsort algorithm. The array is initially a maxheap. For each step k in the diagram below, show the complete array after k elements have been placed in their final sorted position with the remainder of the array having been transformed back into a heap. Use one column of the diagram below for each step. You may not need all the columns.

Initial array: `Z O M B I E`

Answer:

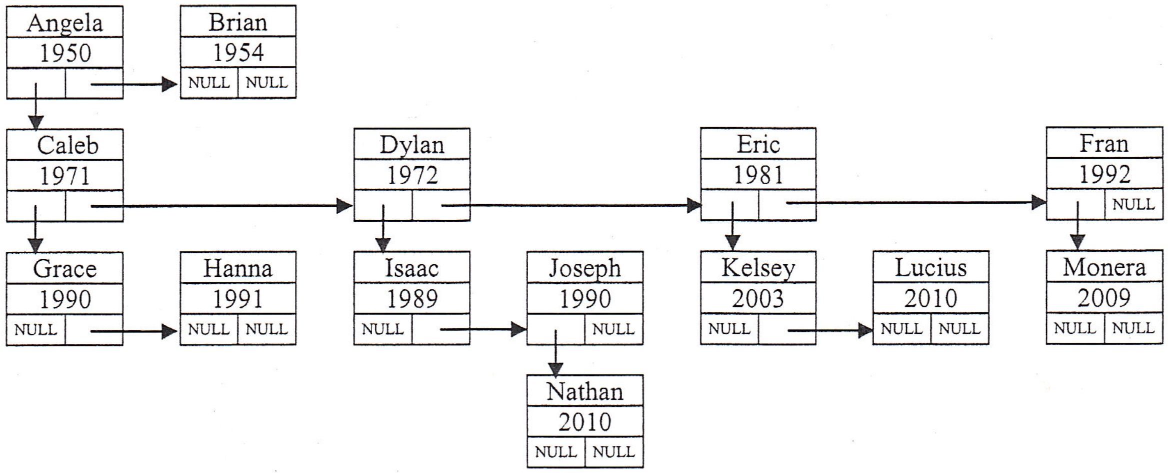
	Initial	step 1	step 2	step 3	step 4	step 5	step 6	step 7	step 8
[0]	Z	O							
[1]	O								
[2]	M			E					
[3]	B			M					
[4]	I								
[5]	E	Z							

3. [20 points in all]

a. **[8 points]** Suppose we store information about a family in a binary tree. The tree consists of nodes of the following type:

```
struct Node
{
    string name;
    int    birthYear;
    Node*  oldestChild;
    Node*  nextYoungerSibling;
};
```

Assume that a `Node*` variable named `myFamily` points to the root of this tree:



This indicates, for example, that Angela has a younger sibling Brian and four children; in decreasing order of age, they are Caleb, Dylan, Eric, and Fran. Eric's oldest child Kelsey has a younger sibling Lucius, but no children.

We say that a tree of `Node`s is *well-formed* if for every node `n`:

- if `n.oldestChild` is not NULL, `n.birthYear < n.oldestChild->birthYear`, and
- if `n.nextYoungerSibling` is not NULL,
  `n.birthYear <= n.nextYoungerSibling->birthYear`

Consider this function:

```
void mystery(const Node* root, int y)
{
   if (root != NULL  &&  y > root->birthYear)
   {
      cout << root->name << endl;
      mystery(root->oldestChild, y);
      mystery(root->nextYoungerSibling, y);
   }
}
```

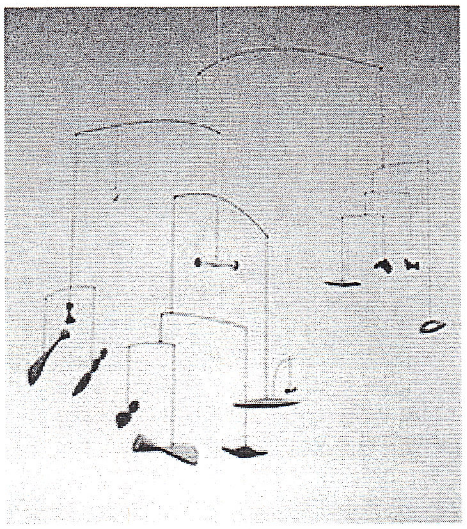
What output is produced by the call `mystery(myFamily, 1991)`?

Answer:

b. [2 points] In a brief sentence, state what the call `mystery(p, yr)` does when `p` points the root of a well-formed tree, and `yr` is an integer. We're not asking for a blow-by-blow account of the function's execution. Just say what its apparent purpose is. (For example, the apparent purpose of the code in problem 2a is to sort an `n`-element integer array `a` in ascending order.)

Answer:

c. **[10 points]** Alexander Calder invented the mobile in 1931.



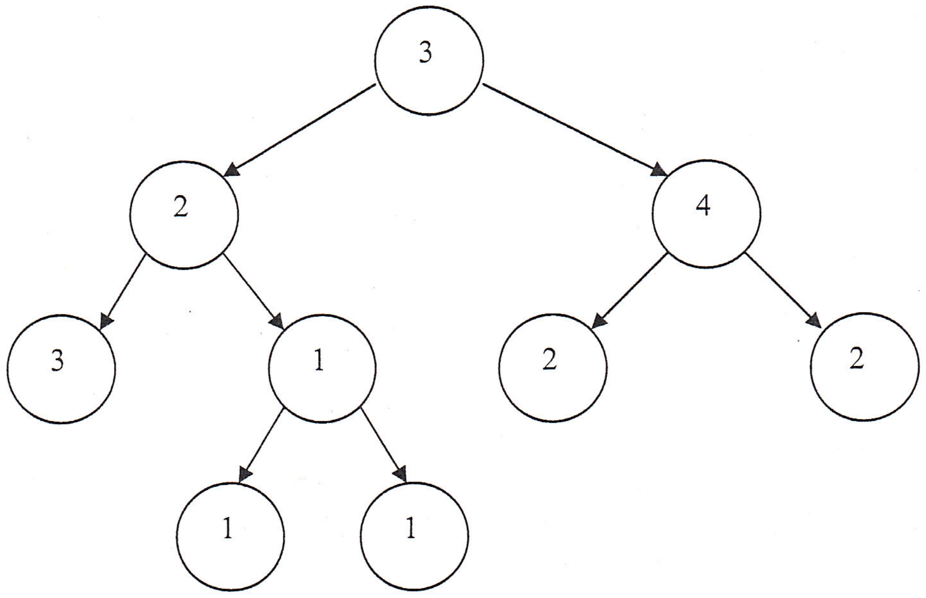
We define a *mobile tree* as a binary tree with nodes of the following type:

```
struct Node
{
   int   weight;   // the weight of this node itself
   Node* left;
   Node* right;
};
```

Define the *weight* of a mobile tree to be the sum of the weights of all the nodes in the tree. (The empty tree therefore has a weight of zero.) We say that a mobile tree is *in balance* if either:

- it is empty, or
- if the weights of its left and right subtrees are equal, and each of those subtrees is itself in balance.

This mobile tree, for example, is in balance:



Write a function named `isInBalance` that takes a pointer to the root node of a mobile tree and returns true if that mobile tree is in balance, or false otherwise. If you wish, you may write an additional function called by `isInBalance`. You must not use any global variables, nor any variables of types other than `bool`, `bool&`, `int`, `int&`, or `Node*`. You must not use the keywords `while`, `for`, `goto`, or `static`. Your solution must not be more than 30 lines of code. To not lose significant credit, your solution must be such that a call to `isInBalance` results in each node in the mobile tree being visited no more than once.

Answer:

```
bool isInBalance(Node* root)
{
```

4. [10 points in all]

You are a programmer for the Committee to Abolish CS 32, which wants to conduct a voter registration drive in West Los Angeles and the Valley. You have a collection of registered voters in those areas, and you have obtained a collection of licensed drivers from there. You want to find all drivers who are not already registered to vote. Both the driver and voter collections are randomly ordered.

Here are two algorithms you are considering:

```
1)  for each licensed driver,
    {   found = false
        for each voter,
                if the driver under consideration == the current voter,
                {       found = true
                        break
                }
        if not found,
                print the driver's name
    }

2)  sort the collection of voters by name, using a good algorithm
    sort the collection of drivers by name, using a good algorithm
    start with the first voter and the first driver
    while not done with the voter collection and not done with the driver collection,
    {   if the current driver's name is alphabetically < the current voter's name,
        {       print the driver's name
                go on to the next driver
        }
        else if the driver's name == the voter's name
        {       go on to the next driver
                go on to the next voter
        }
        else  // the driver > the voter
                go on to the next voter
    }
    print the remaining names, if any, in the driver collection
```

a. **[5 points]** Suppose the driver collection and the voter collection each contain about N names, where N is large. Of the following choices, what is the average case time complexity of the two algorithms? The choices are $O(1)$, $O(\log N)$, $O(N)$, $O(N \log N)$, $O(N^2)$, $O(N^2 \log N)$, $O(N^3)$.

Answer:

Algorithm 1________________ **Algorithm 2**________________

b. **[5 points]** Using a particular computer, you run an algorithm with time complexity $O(N^2)$ and find that it takes a little under 2½ hours to run using the West LA and Valley data. Suppose you wish to run that algorithm for all of California, which has 10 times as many voters and drivers as West LA and the Valley. (California has about 20 million drivers.) About how long would that algorithm take on the California data, using the same computer?

Answer:

5. [15 points]

The following definitions enable us to build a resizable hash table that holds strings (no duplicates allowed):

```
unsigned int hash(string s)
{
   // Omitted is the code that produces a number given a string
   // All you need to know is that this number can range from 0
   // to a large value.  Someone using that number to index into
   // a hash table must reduce it.  The usual approach is to use
   // the number modulo the number of buckets.
}

   // Instead of the hash table containing an array of buckets,
   // it contains a pointer to an array of buckets.  The array is
   // dynamically allocated.  When the average bucket size gets
   // too high, we can call setNumberOfBuckets to cause a new
   // array of buckets to be allocated and the strings to be
   // rehashed into the new array, which will replace the old.
class HashTable
{
  public:
    HashTable(unsigned int initialSize = 97);
    ~HashTable();
    void insert(string s);
    void setNumberOfBuckets(unsigned int newNumberOfBuckets);
    . . .
  private:
    unsigned int   m_numberOfBuckets;
    list<string> * m_buckets;  // ptr to first of many lists
};

HashTable::HashTable(unsigned int initialSize)
 : m_numberOfBuckets(initialSize)
{
   m_buckets = new list<string>[m_numberOfBuckets];
}

HashTable::~HashTable()
{
   delete [] m_buckets;
}

void HashTable::insert(string s)
{
   unsigned int h = hash(s) % m_numberOfBuckets;
     // insert if not already present.  find is a standard
     // library function that does a linear search through
     // a range, returning the range's end iterator if the
     // item is not found.
   if (find(m_buckets[h].begin(), m_buckets[h].end(), s) ==
                                          m_buckets[h].end())
      m_buckets[h].push_front(s);
}
```

Implement `HashTable::setNumberOfBuckets`. You may find this refresher about the standard list class template useful. (This problem can be solved without using any list functions other than these):

```
template<typename T>
class list
{
  . . .
   bool empty() const;        // Return true if the list is empty.
   void push_front(const T& x);  // Insert x at the front.
   const T& front() const;    // Return the item stored at the
                              // front.  (The list must be
                              // non-empty.)
   void pop_front();  // Remove the front item from the list.
                      // (The list must be non-empty.)
  . . .
};
```

Your implementation must not do anything undefined and must not leak memory. Your solution must be no more than 30 lines of code.

Answer:

```
void HashTable::setNumberOfBuckets(unsigned int newNumberOfBuckets)
{
```

6. [20 points]

Spider Man

WebHead, Inc., has hired you to design a web spider component for a search engine. A web spider is a program that crawls the Internet, searching for web pages to index. Usually, a web spider is provided with a small collection of seed URLs by the user. The spider downloads each of the named web pages, and among other things, extracts all of the links to other web pages from that page. Those links are then processed in the same way: The spider downloads the pages they refer to and extracts their links to be processed. This activity continues indefinitely.

WebHead would like you to come up with the data structures and algorithms for a new web spider. For this problem, you must limit your choice of data structures to the following

```
struct Thing                list<T>                 BST_set<T>
{                           stack<T>                BST_map<T,U>
    string m1;              queue<T>                hash_set<T>
    int m2;                 max_heap<T>             hash_map<T,U>
};                          min_heap<T>
```

where T and U can be int, bool, string, Thing, or one of the data structures listed above. Here's what the last four represent:

- A `BST_set<T>` is a collection of objects of type T with no duplicates, organized as a binary search tree. For example, a `BST_set<int>` might be used to represent the ID numbers of students who paid their registration fees.
- A `BST_map<T,U>` is a collection of objects of type U each with a unique key of type T, organized as a binary search tree based on the key. For example, a `BST_map<int,string>` might be used to represent a mapping from ID number to student name. Given an ID number, the binary search tree can be efficiently searched to find the node with that ID number; another field of that node will have the corresponding student name.
- A `hash_set<T>` is a collection of objects of type T with no duplicates, organized as a hash table based on the key. For example, a `hash_set<int>` might be used to represent the ID numbers of students who paid their registration fees.
- A `hash_map<T,U>` is a collection of objects of type U each with a unique key of type T, organized as a hash table based on the key. For example, a `hash_map<int,string>` might be used to represent a mapping from ID number to student name. Given an ID number, the hash table can be efficiently searched to find the node with that ID number; another field of that node will have the corresponding student name.

For `max_heap<T>`, `min_heap<T>`, `BST_set<T>`, and `BST_map<T,U>`, you may assume that there is an ordering relation for type T. For `hash_set<T>` and `hash_map<T,U>`, you may assume that objects of type T can be compared for equality and that there is a good hash function for objects of type T.

Here are four examples of allowable data structures:

```
hash_map<string,int>           BST_map<int,BST_set<Thing> >
stack<list<Thing> >            list<queue<max_heap<int> > >
```

For each answer below, you will select a data structure, what types you would use for T and U, and what those types represent. If the answer includes the `Thing` structure, it will indicate what the members `m1` and `m2` represent. If an answer includes a `max_heap` or a `min_heap`, it will indicate what will determine what is considered the "smallest" item in the heap.

For each problem, you must provide the **most efficient** solution for the given requirements, using as few data structures as possible. Choose the single best answer to each question and write its letter in the table on the last page of the exam.

Here we go!

a. WebHead would like you to record how many web pages you've visited from each domain and look up that number efficiently given the domain. For example, if the spider has crawled to the two web pages `www.yahoo.com/foo` and `www.yahoo.com/bar` in the domain `www.yahoo.com`, then the visit count would be V(`www.yahoo.com`) = 2. WebHead would further like to print, in alphabetical order, a list of all the domains that have been processed up to now, along with their visit counts.

1. **[3 points]** Which data structure would you use to solve this problem? (Remember, you are limited as described on the previous page, and must strive for the most efficient solution.)
   - A. `list<string>`, where the string is the domain name
   - B. `max_heap<Thing>`, where `m1` is the domain name and `m2` is the visit count and the smallest item has the earliest `m1` in alphabetical order
   - C. `min_heap<Thing>`, where `m1` is the domain name and `m2` is the visit count and the smallest item has the earliest `m1` in alphabetical order
   - D. `BST_map<string,int>`, where the string is the domain name and the integer is the visit count
   - E. `hash_map<string,int>`, where the string is the domain name and the integer is the visit count

2. **[1 point]** What is the time complexity for searching for a domain in your data structure if there are D domains?
   - A. O(1)
   - B. O(log D)
   - C. O(D)
   - D. O(D log D)
   - E. none of the above

3. **[1 point]** What is the time complexity for printing all domains in alphabetical order, along with their corresponding visit counts, if there are D domains?
   - A. O(1)
   - B. O(log D)
   - C. O(D)
   - D. O(D log D)
   - E. none of the above

b. WebHead needs to efficiently determine if the crawler has already visited a URL within the last month, so they can avoid revisiting a URL too frequently. They would like to be able to look up a given URL and find the date it was last indexed. They would also like to be able to obtain a list of all URLs that were processed on a given date. You may assume that a date can be encoded as an int.

1. **[2 points]** Which two data structures would you use to solve this problem? [This is the only question for part b.]
   - A. `BST_map<string,int>`, and `BST_map<int,list<string> >` where the strings are the URL and the integers are the date
   - B. `BST_map<string,int>`, and `hash_map<int,list<string> >` where the strings are the URL and the integers are the date
   - C. `hash_map<string,int>`, and `BST_map<int,list<string> >` where the strings are the URL and the integers are the date
   - D. `hash_map<string,int>`, and `hash_map<int,list<string> >` where the strings are the URL and the integers are the date

c. The web spider must have an efficient way to prioritize which link it crawls to next. To do so, WebHead wants you to prioritize crawling based on the popularity of each web page's domain, crawling pages from the more popular domains before pages from less popular domains. There is already a popularity rating function that can instantly tell you how popular a given domain is. The popularity of a domain d, P(d), is an integer from 1 to 100, with smaller numbers representing more popular domains. For example, if the spider has crawled to the two web pages **`www.ucla.edu/cs32`**, with P(**`www.ucla.edu`**) = 4, and **`www.usc.edu/bletch`**, with P(**`www.usc.edu`**) = 67, then the first page would have a higher priority for crawling.

1. **[2 points]** Which data structure would you use to solve this problem?
   - A. **`max_heap<Thing>`**, where `m1` is the domain name and `m2` is the popularity and the smallest item has the smallest `m2` value
   - B. **`min_heap<Thing>`**, where `m1` is the domain name and `m2` is the popularity and the smallest item has the smallest `m2` value
   - C. **`BST_map<int,string>`**, where the integer is the popularity and the string is the domain name
   - D. **`hash_map<int,string>`**, where the integer is the popularity and the string is the domain name

2. **[1 point]** What is the time complexity for inserting a new item into your data structure if there are D domains?
   - A. O(1)
   - B. O(log D)
   - C. O(D)
   - D. O(D log D)
   - E. none of the above

d. WebHead would like to be able to efficiently determine which web pages that have been previously indexed contain links to a particular page. For example, if **`www.a.com/foo`**, **`www.b.com/bar`**, and **`www.c.com/bletch`** each contain a link to the page **`www.ucla.edu/cs32`**, the request L(**`www.ucla.edu/cs32`**) would return a collection of the three links. WebHead would also like to use the same data structure to efficiently determine whether a particular page links to another particular page. For example, L2(**`www.b.com/bar`**, **`www.ucla.edu/cs32`**) would return true.

1. **[3 points]** Which data structure would you use to solve this problem? (In each of these, the first string is the page linked to and the second string is a page linked to it.)
   - A. **`BST_map<string,list<string> >`**
   - B. **`BST_map<string,BST_set<string> >`**
   - C. **`BST_map<string,hash_set<string> >`**
   - D. **`hash_map<string,list<string> >`**
   - E. **`hash_map<string,BST_set<string> >`**
   - F. **`hash_map<string,hash_set<string> >`**

2. **[1 point]** What is the time complexity to determine if a page Q links to a page P, assuming that on average, N pages link to a page?
   - A. O(1)
   - B. O(log N)
   - C. O(N)
   - D. O(N log N)
   - E. O(N^2)
   - F. none of the above

e. Finally, WebHead occasionally locates phishing websites, which are criminal websites that try to steal your personal data. When WebHead locates such a phishing site P, it marks that page as bad, and also marks all pages in the set S1 of pages that link to P as bad, and further marks all pages in the set S2 of pages that link to any page in S1 as bad. For example, assume that `www.phish.com` is a phishing site, and `www.a.com`, `www.b.com`, and `www.c.com` are popular pages that link to `www.phish.com`. Suppose further that the popular pages `www.d.com` and `www.e.com` link to `www.c.com`, and the popular page `www.f.com` links to `www.b.com`. Then WebHead would like to mark all of those pages as bad in addition to `www.phish.com` itself.

1. **[2 points]** Which type of graph traversal algorithm would be most efficient at marking these other pages as bad, with which data structure used by the algorithm?
   - **A.** breadth-first search, using a stack
   - **B.** breadth-first search, using a queue
   - **C.** depth-first search, using a stack
   - **D.** depth-first search, using a queue

2. **[2 points]** If there are T total pages, and on average, N pages link to a page, then given a phishing site P, what is the time complexity to mark all of the appropriate pages as bad?
   - **A.** O(N)
   - **B.** O(T)
   - **C.** O(N log N)
   - **D.** $O(N^2)$
   - **E.** O(NT)

3. **[2 points]** In an alternative version of the algorithm, we want to mark only the *highly popular* pages that link to a phishing site (or that link to a highly popular page that links to a phishing site) as bad. Assume we have a BST_map that maps each of the T pages known to the crawler to its popularity level (low, medium, or high). Assuming again that on average, N pages link to a page, then given a phishing site P, what is the big-oh time complexity to mark all of the appropriate highly popular pages as bad?
   - **A.** O(N log T)
   - **B.** $O(N^2)$
   - **C.** O(NT)
   - **D.** $O(N^2 \log T)$
   - **E.** $O(N^2T)$

Write the letters for your answers in this table:

a1				a2	a3
	b1				
	c1			c2	
d1				d2	
	e1	e2	e3		