Name: ____

USC loginid (e.g., ttrojan):____________________________

CS 455 Final Exam
Spring 2011 [Bono]

May 10, 2011

There are 5 problems on the exam, with 67 points total available. There are 7 pages to the exam, including this one; make sure you have all of them. If you need additional space to write any answers, you may use the backs of exam pages (just direct us to look there).

Remote DEN students only: Do not write on the backs of pages. If additional space is needed, ask proctor for additional blank page(s), put your name on them, and attach them to the exam.

Put your name and USC loginid at the top of the exam. Please read over the whole test before beginning. Good luck!

Types used in problem 5:

```
struct Node {
  int data;
  Node * next;
  Node() { data = 0; next = NULL; }
  Node(int d) { data = d; next = NULL; }
  Node(int d, Node * n) { data = d; next = n; }
};

typedef Node * ListType;
```

---

	value	score
Problem 1	9 pts.	
Problem 2	7 pts.	
Problem 3	6 pts.	
Problem 4	25 pts.	
Problem 5	20 pts.	
TOTAL	67 pts.	

Problem 1 [9 pts.]

Part A. Show the results of inserting the following keys into the empty 9-bucket hash table shown below that uses chaining. The keys are shown along with the their hash codes. Do the insertions in the order the keys are shown (**a-f**).

Key	**hash code**
a. Ashley	5
b. Joe	3
c. Ted	3
d. Dana	8
e. Sam	3
f. Sara	2

0

1

2

3

4

5

6

7

8

Part B. For each of the following lookups from the table resulting from part A, give the sequence of keys that the target key would have to be compared with to do the lookup, or say "none" if it doesn't need to be compared to any keys.

i. lookup *Ted*; Ted hashes to 3

ii. lookup *Anna*; Anna hashes to 8

iii. lookup *Bob*; Bob hashes to 4

Problem 2 [7 pts.]

Consider the static Java method, `yuhu`, below, which has the precondition given in a comment before its header.

```
// PRE: the values in arr are in increasing order and are unique
  public static int yuhu(int [] arr, int x) {
    int low = 0;
    int high = arr.length - 1;
    while (low <= high) {
      int mid = (low + high) / 2;
      if (x == arr[mid]) {
        return mid;
      }
      else if (x < arr[mid]) {
        high = mid - 1;
      }
      else {
        low = mid + 1;
      }
    }  // end while
    return -1;
  }
```

Part A. Write a method comment for `yuhu` in the space below; i.e., one suitable for a caller of the method. Reminder: a method comment describes what a method does, not how it does it, and it should include a description of how the return value relates to the parameters given. (You do not have to restate the precondition in your answer.) No credit will be given for a line-by-line description of the code.

Part B. What is the big-O worst-case time of the method?

Problem 3 [6 pts.]

This problem is about the Queue ADT and about differences in object semantics and parameter passing in Java and C++. In the following programs assume that both Java and C++ have a generic/template Queue class with the operations enQ (short for enqueue), deQ (short for dequeue), front, and isEmpty, with the usual meanings. **Show the output for each of the following two programs that look roughly the "same" in C++ and Java.** Show your answer for a program to the right of that program. Note: the C++ function uses pass by-reference.

```
// Java program
public class MainClass {
  public static void myFunc(Queue<Integer> a, Queue<Integer> b) {
    a.deQ();
    b = a;
  }
  public static void main(String[] args) {
    Queue<Integer> s = new Queue<Integer>();
    s.enQ(3);
    s.enQ(4);

    Queue<Integer> t = new Queue<Integer>();
    t.enQ(9);
    t.enQ(7);
    System.out.println(s.front() + " " + t.front());

    myFunc(s, t);
    System.out.println(s.front() + " " + t.front());
  }
}
```

```
// C++ program – NOTE: uses pass by-reference
void myFunc(Queue<int> & a, Queue<int> & b) {
  a.deQ();
  b = a;
}

int main() {
  Queue<int> s;
  s.enQ(3);
  s.enQ(4);

  Queue<int> t;
  t.enQ(9);
  t.enQ(7);
  cout << s.front() << " " << t.front() << endl;


  myFunc(s, t);
  cout << s.front() << " " << t.front() << endl;
}
```

Problem 4 [25 pts. total]

Suppose we have an array, `arr`, of `arr.length` values that we want to sort into increasing order. Furthermore, we know that all the values are in the range 0..`max`, inclusive. One method to sort them is called bucket sort. In a bucket sort, we will use an auxiliary array, `counts`, to keep track of the number of occurrences of each integer from `arr`. (The elements in `counts` are the "buckets"; there will be `max`+ 1 such buckets.) We sort by first filling the `counts` array, and then finish sorting `arr` by using the data in `counts`. (Note: this is an example of a non comparison-based sort.)

Part A (20). Implement bucket sort in the **Java** method `bsort`, below. Here's an example of possible input and results:

`max` = 5

`arr` before call:	`arr` after `sort(arr, 5)`:
[2, 0, 5, 2, 4, 5, 0, 2]	*[0, 0, 2, 2, 2, 4, 5, 5]*

```
// sorts the arr.length values in arr into increasing order.
// PRE: all the values in arr are between 0 and max, inclusive,
//   and max > 0
static public void bsort(int[] arr, int max) {
```

Problem 4 (cont.)

Part B (2). What’s the big-O worst-case time for the bucket sort algorithm?

Part C (3). Why is bucket sort not useful as a general-purpose sorting algorithm? (Keep your answer to a few sentences; there’s only a lot of space here because of page formatting.)

Problem 5 [20 pts.]

Implement the **C++** function `copyEveryOther` that makes a new list like its parameter, `list`, but that only has every other element from the original list, starting with the first. *`list` is unchanged by this function.* The `Node` type used is given on the front page of the exam.

Examples: (lists shown as space separated list of values surrounded by parentheses)

`list`	**`copyEveryOther(list)`**	**`list`** after call
(6 7 5)	*(6 5)*	*(6 7 5)*
(2 6 7 5)	*(2 7)*	*[. . .for all examples*
(2 6 7 5 3 9)	*(2 7 3)*	*list unchanged by call]*
()	*()*	
(2 6)	*(2)*	
(2)	*(2)*	

```
// Returns a list that has every other element from param. list.
// The original list is unchanged.
// PRE: list is a well-formed linked list
Node * copyEveryOther(Node * list) {
```