# CS61B, Summer 1996
# Midterm #2

**Question 1 [6 points total]: Heap on the medals, please**

Suppose that a character array
**G**
, to be sorted into alphabetical order via heapsort, initially contains the following sequence of characters:

O L Y M P I C S

In this situation the search keys are characters, and a search key is considered is ">" another search key if it occurs earlier in the alphabet.
**Please circle your answers for each part.**
If you wish, show your work neatly to receive partial credit in case your answer is wrong.

**Part A. (2 pts)**

We can transform an array into a heap by calling
**Rebuild Heap**
(see Carrano 11) on each element in the last array, from last to first. Show how the characters would be arranged in the array after transforming the above array into a heap.

**Part B. (3 pts)**

Suppose that instead of starting with a filled array and then transforming it into a heap, we start with an empty heap and successively insert the characters in "OLYMPICS" from "O" to "S". Show how the characters would be arranged in the array after adding all eight characters into the heap. How many comparisons of search keys are done in total after inserting all eight characters into the heap?

**Part C. (1 pt)**

Using the heap from Part B, show how the characters would be arranged in the array after removing the first two items from the heap.

**Question 2 [7 points total] Let's hash this out**

This problem contains three independent parts that all concern hash table. You should assume for this problem that there are no problems with arithmetic overflows. i.e., that the machine can handle arbitrarily large numerical values.

**Part A. (2 pts)**

Suppose that we implement a hash table using a character array of size 13 to store the keys themselves. Show the contents of the array when the keys
**S I L V E R M E D A L**
are inserted in that order into an empty hash table with linear probing to resolve conflicts. Use
*h(k) = k*
**mod**
13 as the hash function for the
*k*
th letter of the alphabet. (Note: S = 19th letter of alphabet, I = 9th, L = 12th, V = 2nd, E = 5th, R = 19th, M = 13th, E = 5th, D = 4th, A = 1st, L = 12th.) Circle those keys that encountered collisions when they were inserted into the table.

**Part B. (3 pts)**

Suppose you wish to make a hash table of all the computer science students at UC Berkeley using as the key each student's ID number, and suppose you have decided to implement the hash table using an unsigned long integer array of size 499, with quadratic probing to resolve conflicts. (by the way, both 4999 and 997 are prime numbers.)

For each of the following three possible hash functions, say whether it is a good or bad choice, and

explain why in one sentence. Note that
*digit*
1
refers to some integer value between 0 and 9, not to a character in the range between '0' and '9', which have integer values from 48 to 57 on our HP workstations.

1.
*digit*
0
+
*digit*
1
+
*digit*
2
+...
*digit*
7
**mod**
4999:

2. SID number
**mod**
4999:

3. (
*digit*
0
+
*digit*
1
* 997 +
*digit*
2
* 997
2
+...
*digit*
7
* 997
7

)
**mod**
4999:

**Part C. (2 pts)**

Of the different hashing approaches that were discussed in lecture (simple array with linear probing, simple array with quadratic probing, simple array with double hashing, buckets, and separate chaining), which approach would be most appropriate for an application in which many equal keys are likely to be present? In at most two sentences, explain why.

**Question 3 [7 points total] All you negative folks, go over there**

For this problem, your job is to write a function,
**rearrange**
, that takes two arguments:
**A**
, an array of integers, and
**n**
, the size of the array.
**rearrange**
shouldn't return anything, but it should rearrange the integers in
**A**
so that all the negative integers precede the nonnegative intergers. It should require only a constant amount of additional memory to run, regardless of the size of
**A**
(you should not, for instance, allocate space for a complete copy of
**A**
), and it should be as efficient as possible.

As an example,
**rearrange**
might be given
**n**
= 10 and this array
**A**
:

[1 10 -5 14 -9 -3 88 -1000 0 4]

It would rearrange these so that the negative integers come before the nonnegative integers (the order of the integers in each group doesn't matter). For example, it might rearrange
**A**
to be:

[-9 -5 -1000 -3 4 10 0 1 88 14]

**Part A. (6 pts)**

Write your
**rearrange**
function here:

**Part B. (1 pt)**

What is the order of growth of your algorithm?

**Question 4 [7 points total] Sorting out sorting**

AFter taking CS 61B, Amy knows that mergesort is typically faster than bubblesort.

**Part A. (1 pt)**

In one sentence, explain what theoretical evidence she has of this.

**Part B. (2 pts)**

Amy runs mergesort and bubblesort on the same data, and even though both algorithms were properly and efficiently coded in C++, bubblesort ran faster. Give two reasons why this might be the case.

Reason 1:

Reason 2:

**Part C. (4 pts)**

Now Amy turns her attention to mergesort. She writes her own variation of mergesort, which splits a list not into halves but into three parts. Her mergesort algorithm recursively calls itself on the three parts. Then, it merges parts 1 and 2, and then merges the result with part 3. What is the order of growth of this algorithm in the average case and in the worst case? Explain your answers.

**Question 5 [7 points total] At your service, General Oak!**

A
*general tree*
is a tree in which each node can have any number of childern. In homework #3, we saw how any general tree is uniquely described by its preorder and postorder traversals. Your job in that homework was to write a
**makeGeneral**
function that performed the reconsruction, given vectors containing the preorder and postorder traversals.

For this exam problem, your job will be to answer questions about the unique general tree, which we'll refer to as
**Oak**

, that corresponds to the preorder and postorder vectors listed below.
**Please circle your answer for each part.**
the relevant C++ code from the solutions is reproduced on the next page.

preorder = [ 15 9 99 36 66 16 50 81 28 71 42 3 10 7 11 13 12 ]

postorder = [ 99 36 9 16 50 28 71 81 66 42 10 3 11 13 12 7 15 ]

**Part A. (1 pt)**

What is the value at the root node of
**Oak**
?

**Part B. (3 pts)**

What are the values at the child nodes of the root node? You should list them in correct order, from leftmost child to rightmost child.

**Part C. (3 pts)**

One of the nodes in
**Oak**
has value 66. In the space below, draw the subtree of
**Oadk**
that is rooted at this node. You should draw this subtree as a regular general tree. (Don't use the binary tree representation of general trees that is described in homework #3.)

void makeGeneral (binTree &T, Vector<int >&PreorderV, Vector<int >&PostorderV, bool

```
&Succeeded)

{

        if (PreorderV.size() == 0) {

                Succeeded = true;
                return;

        } else {

                // Set the root node. It should be equal to PostOrderV[PostorderV.size()-1]
                T.SetRootDate (PreorderV[0]);

                // search for each child from last to first
                int pre_index = PreorderV.size() -1;
                int post_index = PostorderV.size() - 2;
                while (post_index >=0) {

                        int i;
                        for (i = pre_index; PreorderV[i] != PostorderV[post_index]; --
                        i) {

                                if (i <= 0) {
                                        Succeeded = false;
                                        return;
                                }

                        }

                        //make the subtree and attach it
                        int length = pre_index - i + 1;
                        binTree new_child;
                        int pre_start = pre_index - length + 1;
                        int post_start = post_index - length + 1;
                        Vector< int > pre_subv (length, 0), post_subv(length, 0);
                        for (i = pre_start; i <= pre_index; ++i) {
                                pre_subv[i-pre_start] = PreorderV[i];
                        }
                        for (i = post_start; i <= post_index; ++i) {
                                post_subv[i-post_start] = PostorderV[i];
                        }
                        // recursively call makeGeneral to generate subtree
```

```
                    makeGeneral (new_child, pre_subv, post_subv, Succeeded);
                    if (!Succeeded) {          return;
                    }
                    attachChild (T, new_child);

                    // adjust pre_index and post_index for search
                    // for the next child
                    pre_index = pre_index - length;
                    post_index = post_index - length;

            }
            Succeeded = true;

        }

}


void attachChild (binTree &par, binTree &child)

{

        bool success;
        binTree temp_tree;

        if (par.leftSubtree().IsEmpty()) {
                par.AttachLeftSubtree(child, success);
        } else {
                child.AttachRightSubtree(par.LeftSubtree(), success);
                par.DetachLeftSubtree(tmp_tree, success);
                par.AttachLeftSubtree(child, success);
        }

}
```

**Bonus Question [3 Points total]: Find the median in linear time**

This question is **optional**. Do not work on it unless you are happy with your answers to the required questions on this exam. Litlle

or no credit will be given for incorrect answers to this problem.

For this problem, your job is to write a function,
**median**
, that takes two arguments:
**A**
, an array of integers, and
**n**
, the size of the array.
**median**
should return the value of the median integer in the list,
**and it should run in linear time**
.

As an example,
**median**
might be given
**n**
= 9 and this array
**A**
:

[1 10 -5 14 -9 -3 88 -1000 4]

In this case,
**median**
should return 1 since that is the middle value in the list. If the list has even length, it should return the smaller of the two middle values.

Write your
**median**
function here: