NAME (print): ____________________________________________

Honor Acknowledgment (signature): __________________________

DO NOT SPEND MORE THAN 10 OR SO MINUTES ON ANY OF THE OTHER QUESTIONS! If you don’t see the solution to a problem right away, move on to another problem and come back to it later.

Before starting, make sure your test contains 8 pages.

If you think there is a syntax error, then ask. You may assume any include statements are provided.

<table>
<thead>
<tr>
<th>Problem</th>
<th>value</th>
<th>grade</th>
</tr>
</thead>
<tbody>
<tr>
<td>Problem 1</td>
<td>16 pts.</td>
<td></td>
</tr>
<tr>
<td>Problem 2</td>
<td>10 pts.</td>
<td></td>
</tr>
<tr>
<td>Problem 3</td>
<td>12 pts.</td>
<td></td>
</tr>
<tr>
<td>Problem 4</td>
<td>16 pts.</td>
<td></td>
</tr>
<tr>
<td>Problem 5</td>
<td>18 pts.</td>
<td></td>
</tr>
<tr>
<td>TOTAL:</td>
<td>72 pts.</td>
<td></td>
</tr>
</tbody>
</table>

Whenever you see references to Node or TNode in this test, check the separate handout for their definition (unless a specific definition is included in the problem statement).
PROBLEM 1:  (Analyses ... (16 points))

Part A (6 points) What is the worst case running time T(N) as expressed by using O(?) (or big O) for the following (crazy) code segment?

```c
int j, k, h;
int sum = 0;
for (k = 1; k <= N; k++) // N has a value from previous code
{
    P = 3;       // this line can be changed; see below
    for (j = 1; j <= P; j++)
    {
        for (h = 1; h <= N; h++)
            sum += h;
    }
}
```

What is the answer for P = 3 as shown?

What is the answer if we change the line setting up P to:
P = N/2;

What is the answer if we change the line setting up P to:
P = floor(log(N));

Part B (3 points) Consider the following (nonsensical) Mystery function, but don’t attempt to decipher what it computes. Instead, give only the recurrence relation describing the running time of Mystery. DO NOT attempt to solve the recurrence relation or to compute big-Oh.

```c
int Mystery(int N)
{
    int tot = 0;
    int k, j;
    if (N == 0)
        return 1;
    for (k = 0; k < 8; k++)
    {
        tot += Mystery(N/3);
        j = k/3;
        tot -= j%k;
    }
    return tot;
}
```
Part C (4 points) Assume you are given a pointer to a singly-linked list of elements in sorted order from smallest to largest. The pointer to the linked list points to the smallest element. The list may contain duplicate items.

Consider the most efficient implementation of each of the following operations, given only a pointer to the front of the list. For example, find the maximum would be implemented as a function with the only parameter a pointer to the linked list. DO NOT modify the data structure. Also, consider only solutions that do not require more than O(1) additional explicit memory, but ignore memory requirements implicit in function call mechanisms, etc.

(1) Find the maximum element. Running time:

(2) Find the minimum element. Running time:

(3) Is X an element in the list. Running time:

(4) Print the element that occurs most often. Running time:

(5) List the elements from largest to smallest. Running time:

Also describe your algorithm for this last one (5).

Part D (3 points) Reconsider the problems in part C above, but assume the same sorted information is stored in a vector and you are given the name of the vector and N, the number of values in the vector.

(1) Find the maximum element. Running time:

(2) Find the minimum element. Running time:

(3) Is X an element in the vector. Running time:

(4) Print the element that occurs most often. Running time:

(5) List the elements from largest to smallest. Running time:
PROBLEM 2:  \textit{(Save a tree... (10 points))}

Part A (6 points)

Draw the single binary search tree from which the following traversals result:

pre:  5 1 0 3 2 4 8 6 7

in:   0 1 2 3 4 5 6 7 8

post: 0 2 4 3 1 7 6 8 5

Part B (4 points)

Give a possible order of insertion that would result in the following binary search tree:

```
       G
      / \  
     /  \  
    /    \ 
   D    O  
  / \  / \ 
 E K  U  
```

And for this one:

```
     A
    / \ 
   M / \ 
  E  O / \ 
 / \ C I 
G K
```
PROBLEM 3:  (Customize (12 points))

Assume that all of the technical specifications, plans, and related documents for a large technical product are stored in a binary tree. Each info field (a pointer to "stuff") points to a document. Assume that this product has many options, so to custom build a particular version of that product, you would only include plans for the parts and options requested by the customer.

We need a routine that can copy the document tree described including only the portions of the tree needed. We will assume that whenever something is not needed, then the whole related subtree is omitted.

Write the function EditCopy that returns a copy of the tree described including only the portions of the tree needed. We will assume that whenever something is not needed, then the whole related subtree is omitted.

In order to decide which subtrees to omit, assume the existence of a boolean function called TreeSelect which, when passed a pointer to a subtree, returns a true if the subtree is to be copied to the new tree and false if that subtree is to be omitted from the copy. The prototype for TreeSelect is shown below, just before the header of EditCopy which you are to complete.

```cpp
bool TreeSelect(TNode<stuff *> * t);  // DO NOT write this

TNode<stuff *> * EditCopy(TNode<stuff *> * t)
// post: returns a copy of t, possibly with some of its subtrees omitted
//       as dictated by the bool function TreeSelect
{
```
PROBLEM 4: (Wash out your mouth with soap... (16 points))

Assume we need to create a data structure to store "naughty" (XXX) words so that we can flag bad language in scripts for TV children’s programs. We want a function to use a vector of pointers to linked lists of words. The first vector entry (element 0) points to a linked list of words starting with the character 'A'. The next element points to words starting with 'B', etc., thru 'Z'/

Don’t worry about the order of words in each list (as long as they start with the right letter). Assume all letters are capitalized. Remember that each word is stored in a node that you create.

Hint: Characters are like short integers: 'A' - 'A' == 0, 'B' - 'A' == 1, 'C' - 'A' == 2, etc.

The diagram shows the data structure after the words AXXX1, AXXX2, AXXX3, BXX1, BXX2, BXX3, CXXX1, and CXXX2 have been read in and processed. However, the words in the file are in no particular order.

Write the function badWords whose header is shown below, so that it takes a text file and reads the words in.

Assume that the function is passed an open file and a 26 element vector.

```c
void badWords(istream & input, Vector<Node<string> *> & baddies)
// pre: input is open istream, baddies is a vector of pointers to nodes,
// all initialized to NULL.
// post: input is at end of file, the words from the file have been
// distributed to linked lists selected by the first letter of
// the words.
{
```
You may have noticed that the nodes used in a doubly linked list are structurally identical to binary tree nodes. Although pointers in list nodes are often named "prev" and "next", they could just as easily be called "left" and "right".

Here our strategy is to NOT create or delete any nodes. Instead, you will take existing nodes that were linked together as a sorted, doubly-linked list and re-arrange them, by changing pointers appropriately, to transform them into a binary search tree.

Write the function `MakeTree` that converts a sorted, doubly-linked list (assume NO header nodes!) into a binary search tree. The middle node of the linked list should become the root of the search tree. Assume the existence of a function `MiddleNode`, that returns a pointer to the middle node of a doubly-linked list (you do NOT need to write `MiddleNode`).

```
TNode * MiddleNode(TNode * list) // do NOT write this
// pre: list is a NULL-terminated, doubly linked list with NO header nodes
// post: returns pointer to middle node of list, returns NULL if
//       list is empty
```

You must use the following algorithm to build the search tree:

1. Find the middle node. It is to be your root node.
2. Unlink the middle node from the list (so the right field of the node before it points to NULL, and the left field of the node after points to NULL) – this step is necessary for the next step (so that `MiddleNode` will work).
3. Make left and right subtrees recursively, using the list to the left of the middle node for the left subtree and the list to the right of the middle node for the right subtree.
4. Return a pointer to your root node.

Complete the function `MakeTree` whose header is given on the next page.
Tnode * MakeTree(Tnode * list)
// pre: list is NULL-terminated, doubly-linked list (with NO header nodes)
// post: Returns a pointer to the root of a binary search tree containing
//        the same nodes as in list.
// post: Returns NULL for an empty list.
{