Test 1: CPS 100

Owen Astrachan

February 13, 2003

Name: ________________________________ (1 pt)
Login: ______________
Honor code acknowledgment (signature) ________________________________

<table>
<thead>
<tr>
<th>Problem</th>
<th>value</th>
<th>grade</th>
</tr>
</thead>
<tbody>
<tr>
<td>Problem 1</td>
<td>20 pts.</td>
<td></td>
</tr>
<tr>
<td>Problem 2</td>
<td>20 pts.</td>
<td></td>
</tr>
<tr>
<td>Problem 3</td>
<td>10 pts.</td>
<td></td>
</tr>
<tr>
<td>Problem 4</td>
<td>12 pts.</td>
<td></td>
</tr>
<tr>
<td>Problem 5</td>
<td>14 pts.</td>
<td></td>
</tr>
<tr>
<td>Problem 6</td>
<td>4 pts.</td>
<td></td>
</tr>
<tr>
<td>TOTAL:</td>
<td>80 pts.</td>
<td></td>
</tr>
</tbody>
</table>

This test has 15 pages, be sure your test has them all. Do NOT spend too much time on one question — remember that this class lasts 75 minutes.

In writing code you do not need to worry about specifying the proper \#include header files. Assume that all the header files we’ve discussed are included in any code you write.
The declaration for linked list nodes on this test is:

```c++
struct ListNode
{
    string info;
    ListNode * next;

    ListNode(const string& s, ListNode * ptr)
        : info(s), next(ptr)
    {
    }
};
```

The declaration for binary search tree nodes on this test is:

```c++
struct TreeNode
{
    string info;
    TreeNode * left;
    TreeNode * right;

    TreeNode(const string& s, TreeNode * lt, TreeNode * rt)
        : info(s), left(lt), right(rt)
    {
    }
};
```

Remember:

\[
\sum_{i=1}^{N} i = \frac{N \times (N + 1)}{2}
\]

\[2^{10} = 1024\]
PROBLEM 1:  (O-O-Ohno (20 points))

Part A (4 points)
Consider the function \texttt{addup} below. The call \texttt{addup(40)} returns the value 785. Complete the table indicating the return values for each call shown.

<table>
<thead>
<tr>
<th>call addup(n)</th>
<th>result</th>
</tr>
</thead>
<tbody>
<tr>
<td>addup(40)</td>
<td>785</td>
</tr>
<tr>
<td>addup(5)</td>
<td></td>
</tr>
<tr>
<td>addup(101)</td>
<td></td>
</tr>
</tbody>
</table>

```c
int addup(int n)
// post: return value based on n
{
    int sum = 5;
    for(int k=0; k < n; k++) {
        sum += k;
    }
    return sum;
}
```

Part B (3 points)
What is the complexity (in terms of runtime) of the call \texttt{addup(N)}. Use big-Oh notation and justify your answer briefly.
Part C (6 points)
Consider the function counter below. The call counter(2,1025) returns the value 11. Complete the table indicating the return values for each call shown. The following list of powers of 3 may help.

\[ 3^0 = 1, \ 3^1 = 3, \ 3^2 = 9, \ 3^3 = 27, \ 3^4 = 81, \ 3^5 = 243, \ 3^6 = 729, \ 3^7 = 2187, \ 3^8 = 6561, \ 3^9 = 19683 \]

<table>
<thead>
<tr>
<th>counter(n, limit)</th>
<th>result</th>
</tr>
</thead>
<tbody>
<tr>
<td>counter(2,1025)</td>
<td>11</td>
</tr>
<tr>
<td>counter(2,1023)</td>
<td></td>
</tr>
<tr>
<td>counter(3,-1)</td>
<td></td>
</tr>
<tr>
<td>counter(3,1025)</td>
<td></td>
</tr>
</tbody>
</table>

```c
int counter(int n, int limit)
{
    int count = 0;
    int value = 1;
    while (value <= limit) {
        value *= n;
        count++;
    }
    return count;
}
```

Part D (3 points)
What is the complexity (in terms of runtime) of the call counter(k,N). Use big-Oh notation and justify your answer briefly. Your answer should use k and/or N.
Part E (4 points)
What is the complexity of the call calculate(N) for the function shown below. Use big-Oh notation and justify your answer briefly.

```c
int calculate(int n)
{
    int prod = 1;
    for(int k=1; k <= n; k++)
    {
        prod = prod * prod;
        for(int j=1; j <= k; j = j*2)
        {
            prod *= j;
        }
    }
    return prod;
}
```
PROBLEM 2:  (Subset (20 points))

Part A (6 points)
Write the function `getNth` shown below that returns a pointer to the $n^{th}$ node of the linked list parameter (indexes start with 0). If $n$ is greater than the length of the list, return 0/NULL.

For example, if `list` is diagrammed as follows (the header node isn’t shown)

( "apple", "guava", "cherry" )

getNth(list,0) should return a pointer to the node containing "apple", getNth(list,6) should return 0/NULL, and getNth(list,1) should return a pointer to the node containing "guava"

Note: lists have header nodes, but no pointer to the header node is ever returned. No new nodes should be created.

Write the function `getNth` below:

```c
ListNode * getNth (ListNode * list, int n)
// pre: list has a header node
// post: returns pointer to n-th node in list, NULL/0 if n > # nodes in list
{
```
Part B (8 pts)
Write the function cloneSome that returns a new linked list whose new nodes contain the the first \( k \) values in the linked list parameter in the same order. The list returned is thus a clone/copy of the first \( k \) nodes in the list parameter. If \( k \) is greater than the length of the list, a copy of the entire list should be returned.

**Note:** the list passed in has a header node and the new list returned should have a header node.

For example, if list is diagrammed as follows (header node not shown)

( "apple", "guava", "cherry", "lemon" )

cloneSome(list,4) and cloneSome(list,10) both return a copy of the entire list; cloneSome(list,0) returns only a header node (an empty list); and cloneSome(list,2) should return a new list with two values (in addition to the header node) that are the same as the first two nodes of the original list:

( "apple", "guava" )

Write the function cloneSome below:

```c
ListNode * cloneSome (ListNode * list, int k)
// pre: list has a header node
// post: returns pointer to a list with a header node
// that contains copies of the first k nodes of list
```
**Part C (6 points)**

Write the function `sublist` that returns a new list whose new nodes contain values from consecutive nodes in the linked list parameter. The first node of the list passed in has index 0 and is denoted \( n_0 \). The list returned should consist of copies of nodes \( n_{\text{start}}, n_{\text{start}+1}, n_{\text{start}+2}, \ldots, n_{\text{start}+\text{length}-1} \).

The list returned should contain at most `length` nodes. If there are fewer than `length` nodes starting at index `start` then the list returned should contain as many as possible.

**Note:** the list parameter has a header node, the new list returned should have a header node. For example, if `list` is diagrammed as follows (header node not shown)

```
( "apple", "guava", "cherry" )
```

then `sublist(list,1, 1)` returns a new list with a single node (in addition to the header node):

```
( "guava" )
```

Both `sublist(list, 1, 2)` and `sublist(list, 1, 6)` return a new list with the same values as the last two nodes (plus a header node)

```
( "guava", "cherry" )
```

The call `sublist(list,4,k)` results in undefined behavior since list contains only three nodes, so the only legal values of `start` are 0, 1, and 2.

Write the function `sublist` below. In writing `sublist` assume `cloneSome` and `getNth` work as specified, regardless of what you wrote earlier.

Add code in two places below.

```c
ListNode * sublist (ListNode * list, int start, int length)
// pre: list has a header node, 0 <= start < number of nodes in list
// post: returns linked list with header node whose values are copies
// of list starting at node start and containing at most length nodes.
// it is an error if start isn't a valid index
{
    // starting at first node is a special case, just call cloneSome
    if (start == 0) {
        // fill in call here to cloneSome
        return cloneSome( , );
    }

    // We want to clone the right number of nodes for the sublist. We want to call
    // cloneSome, but it needs a list with header node so we'll fake a header node
    // by calling getNth so we get a pointer to the node before where we're starting to clone
    ListNode * front = getNth(list,start-1); // make header node

    // complete function here
}
```
PROBLEM 3:  (Babies Burp and Flurp (10 points))

Part A (4 pts)
Describe what the function flurp below returns.


Part B (6 pts)
Write the function isSorted below that returns true if its list parameter is sorted in alphabetical order and false otherwise. An empty list is considered sorted and a one-node list is considered sorted. Two strings can be compared in alphabetical order using the < operator, e.g., the code below prints “yes”

```
string a = "zebra", b = "yak";
if (b < a) cout << "yes" << endl;
```

*Your implementation must be recursive.*

bool isSorted (ListNode * list)
// pre: list is singly linked list, possibly empty
// post: returns true if list’s values are sorted in increasing
// alphabetic order, false otherwise
{
}
PROBLEM 4:  
(Compared to What (12 points))

The function isEqual below returns true if the two linked list parameters contain the same values in the same order and false otherwise. It is correctly implemented.

For example, given the lists below the call isEqual(list1,list2) evaluates to true, the call isEqual(list2,list3) evaluates to false (the lists don’t contain the same values) and the call isEqual(list2,list4) evaluates to false (the values aren’t the same).

list1 = ("apple", "banana", "fig", "lemon")
list2 = ("apple", "banana", "fig", "lemon")
list3 = ("apple", "banana", "fig")
list4 = ("apple", "banana", "FIG", "LEMON")

bool isEqual (ListNode * lhs, ListNode * rhs)
// pre: lhs and rhs are singly linked lists, possibly empty
// post: returns true if lhs and rhs have same values in same order
// returns false otherwise
{
    if (lhs == 0 && rhs == 0) return true; // both empty
    if (lhs == 0 && rhs != 0) return false; // lhs shorter, not equal
    if (lhs != 0 && rhs == 0) return false; // rhs shorter, not equal

    return (lhs->info == rhs->info && isEqual(lhs->next, rhs->next));
}

Suppose you want to modify the function to ignore case in determining if two values are equal. For example, you want a new function, isQuasiEqual to work so that the call isQuasiEqual(list2,list4) returns true. You can do this by replacing the last line above with the three lines below.

    string left = LowerString(lhs->info);
    string right = LowerString(rhs->info);
    return (left == right && isEqual(lhs->next, rhs->next));

To avoid re-writing the function and creating a new function with each new kind of comparison, you will implement a generalized version of the function that has a third parameter. The third parameter is used to compare two nodes to determine if they are equal. It is a pointer to an object that does the comparisons. Using inheritance callers can create new Comparator subclasses and use the generalized isEqual function.

// compare two strings for equality, let subclasses define what equals means

class Comparator
{
    public:
        virtual ~Comparator () {}  

        // return true if two strings are the same, false otherwise
        virtual bool isSame (const string& lhs, const string& rhs) const = 0;
};
For example, here’s code that uses the three-parameter, generalized function to do the same thing `isEqual` does above.

```cpp
class Equal : public Comparator
{
    public:
    virtual bool isSame(const string& lhs, const string& rhs) const
    {
        return lhs == rhs;
    }
};

int main()
{
    ListNode * lhs = makeSomeList();
    ListNode * rhs = makeAnotherList();
    Comparator * comp = new Equal();

    if (isEqual(lhs, rhs, comp)) {
        cout << "lists are the same" << endl;
    }
}
```

**Part A (2 pts)**

What is the purpose of the C++ keyword `virtual` in the declaration above?

**Part B (4 points)**

Write a subclass of `Comparator` named `SameLength` that returns true if both string parameters have the same length, and false otherwise. For example, using this Comparator should result in the lists below being compared as “equal” since the strings stored in corresponding nodes have the same lengths.

```
list1 = ("apple", "banana", "fig", "lemon")
list2 = ("lemon", "grapes", "yam", "mango")
```

Write the code below.

```cpp
class SameLength : public Comparator
{
    public:
    // return true if both strings have the same length
    virtual bool isSame (const string& lhs, const string& rhs) const
    {
        // add code here
    }
};
```
Part C (6 points)
Write the generalized version of the function isEqual below that returns true if the corresponding values within the two lists compare as equal according to the comparator parameter and false otherwise. See the call to this generalized function above for how it would be called.

```cpp
def isEqual(ListNode * lhs, ListNode * rhs, Comparator * comp)
    // pre: lhs and rhs are singly linked lists, possibly empty
    // post: returns true if lhs and rhs have nodes that contain equal values
    //       as defined by comp, false otherwise
    {
        if (lhs == 0 && rhs == 0) return true;  // both empty
        if (lhs == 0 && rhs != 0) return false; // lhs shorter
        if (lhs != 0 && rhs == 0) return false; // rhs shorter

        // complete function here
    }
```
PROBLEM 5:  (Birds Do It (14 points))

Consider the binary search tree shown below.

Part A (3 points)
Write the values stored in leaf nodes of the tree.

Part B (3 points)
The height of a tree is the number of nodes on the longest root-to-leaf path, the tree above has a height of three. If the value "rhinoceros" is added to the tree above, what will the height be and why?

Part C (3 points)
If the value "panda" is added to the original search tree above, what will the height be and why?

Part D (5 points)
Consider the code below. Assume that makeAnimalTree returns a pointer to the root-node ("penguin") in the tree diagrammed above. What does the code below print?

```cpp
void tree2vec(TreeNode * t, tvector<string>& v)
{
    if (t != 0) {
        tree2vec(t->left,v);
        v.push_back(t->info);
        tree2vec(t->right,v);
    }
}

int main()
{
    TreeNode * root = makeAnimalTree();
    tvector<string> v;
    tree2vec(root,v);

    for(int k=0; k < v.size(); k++) {
        cout << v[k] << endl;
    }
    return 0;
}
```
The question follows the description of the code/problem.
A student has written the code below in `printAll` as part of the Anagrams assignment. The code doesn’t function as intended. You’ll be asked to reason about why and about one proposed method to fix the problem.

For the list diagrammed as:

("pit", "pots", "part", "stop", "rapt", "opts", "trap", "tip")

The student wants to see the output below, but the order of the words on each line doesn’t matter — there should be three lines of output containing the words shown.

```
tip pit
stop opts pots
rapt trap part
```

Instead, however, the output generated by this fragment is

```
pit tip
pots stop opts
rapt trap part
opts stop
trap rapt
```

```cpp
void printAll(const tvector<Anaword>& list)
{
 for (int k = 0; k < list.size(); k++) {
    int count = 1;
    Anaword current = list[k];
    for (int j = k + 1; j < list.size(); j++) {
       Anaword possible = list[j];
       if (current == possible) {
          cout << possible << " ";
          count++;
       }
    }
    if (count > 1) {
       cout << current << endl;
    }
 }
}
```

`question on next page`
Part A (4 pts)
The proposed fix is to add the line below before the for loops

    Anaword xword("xxx");

and to add the following line after the statement count++.

    list[j] = xword;   // mark as used

This fix doesn’t work. What output is generated for the list.

    ("pit", "pots", "part", "stop", "rapt", "opts", "trap", "tip")

This is extra credit, it is not required

Part B (6 pts)
Describe how to modify the code in printAll so that it works as intended. You should not radically alter the approach used, there should still be the two loops shown, for example. You can either mark up the code shown on the previous page with your proposed modification or describe what to do below.