Faculty of Applied Science and Engineering, University of Toronto
CSC181: Introduction to Computer Programming, Fall 2000

Second Term Test
November 22, 2000

Duration: 100 minutes
Aids allowed: None

Do NOT turn this page until you have received the signal to start. (In the meantime, please fill out the identification section below, and read the following instructions carefully.)

This term test consists of 6 questions on 8 pages (including this one). When you receive the signal to start, please make sure that your copy of the test is complete.

Answer each question directly on the test paper, in the space provided, and use the reverse side of the pages for rough work. (If you need more space for one of your solutions, use the reverse side of the page and indicate clearly which part of your work should be marked.)

Be aware that concise, well thought-out answers will be rewarded over long rambling ones. Also, unreadable answers will be given zero (0) so write legibly.

Unless otherwise stated, programming questions will be marked for correctness primarily, with efficiency and style as secondary considerations. Comments are not required, but they should accompany any long or tricky bits of code to aid the marker's understanding.

DON'T PANIC. KEEP COOL.

Family name: ______________________________

Given name: ______________________________

Student number: __ | __ | __ | __ | __ | __ | __ | __ | __

1:	____	/ 5
2:	____	/ 5
3:	____	/ 10
4:	____	/ 10
5:	____	/ 15
6:	____	/ 15
Total:	____	/ 60

Question 0. [2 bonus marks]

Write your student number LEGIBLY on the top of every page of this term test.

Question 1. [5 marks]

Given the `List` typedefs in Appendix A of this test, write a function `listDestroy` with the following prototype:

```
void listDestroy(ListPtr l);
```

This function should free all of the memory allocated to the list pointed to by `l`. Assume that the memory for the nodes in the list, as well as the list itself, has been dynamically allocated using `malloc`.

```
void listDestroy(ListPtr l) {
      ListNodePtr current = l->front;
      while (current != NULL) {
            ListNodePtr temp = current->next;
            free(current);
            current = temp;
      }
      free(l);
}
```

Question 2. [5 marks]

Given the BST typedefs in Appendix B of this test, write a **recursive** function (i.e. one that does not contain any loops) `bstMinimumNode` with the following prototype:

```
BSTNodePtr bstMinimumNode(BSTNodePtr r);
```

This function returns a pointer to the node containing the minimum key in the binary search tree rooted at node `r`, or `NULL` if no such key exists.

```
BSTNodePtr bstMinimumNode(const BSTNodePtr r) {
      if (r == NULL)
            return NULL;
      else if (r->left == NULL)
            return r;
      else
            return bstMinimumNode(r->left);
}
```

Question 3. [10 marks]

Using the string library functions documented in Appendix C, write a function `substringBetweenChars`, which has the following prototype:

`char* substringBetweenChars(char* s, const char* ct, char a, char b)`

This function copies into the array pointed to by s the segment of the string pointed to by `ct` whose first character is the first occurrence of character a in the string and whose last character is the last occurrence of character b in the string. The function appends a null character (‘`\0`’) to this segment, and returns a pointer to the segment.

You cannot use any loops or recursion, but you can use pointer arithmetic.

Examples:

Arguments to `substringBetweenChars`	**Contents of array pointed to by s after `substringBetweenChars`**
`s, "hello", ’h’, ’o’`	`"hello"`
`s, "hello", ’e’, ’l’`	`"ell"`
`s, "hello", ’h’, ’a’`	`""`
`s, "hello", ’a’, ’o’`	`""`
`s, "hello", ’i’, ’j’`	`""`
`s, "hello", ’h’, ’h’`	`"h"`
`s, "hello", ’o’, ’h’`	`""`

```
char* substringBetweenChars(char* s, const char* ct, char a, char b) {
      char *start = strchr(ct, a);
      char *end = strrchr(ct, b);

      if (start != NULL && end != NULL && end >= start) {
            strncpy(s, start, end-start+1);
            *(s+(end-start+1)) = (char)0;
      }
      else {
            *s = (char)0;
      }

      return s;
}
```

Question 4. [10 marks]

Given the BST typedefs in Appendix B of this test, write a function `bstSuccessorNode` with the following prototype:

```
BSTNodePtr bstSuccessorNode(BSTNodePtr x);
```

This function returns a pointer to the node in `x`'s tree containing the lowest key greater than the key of the node pointed to by `x`, or `NULL` if no such node exists.

You may use `bstMinimumNode` from question 2 to answer this question. Assume that it works correctly, even if you didn't answer that question (correctly).

Examples:

<table>
<tr><th>Contents of BST before <code>bstSuccessorNode</code></th><th>Argument to <code>bstSuccessorNode</code></th><th>Result from <code>bstSuccessorNode</code></th></tr>
<tr><td rowspan="4"><pre>
      ___8___
     /   \
    5   10
   / \   / \
  4   6   9   12
       \
        7
</pre></td><td><code>pointer to node with 6</code></td><td><code>pointer to node with 7</code></td></tr>
<tr><td><code>pointer to node with 7</code></td><td><code>pointer to node with 8</code></td></tr>
<tr><td><code>pointer to node with 8</code></td><td><code>pointer to node with 9</code></td></tr>
<tr><td><code>pointer to node with 12</code></td><td><code>NULL</code></td></tr>
</table>

```
BSTNodePtr bstSuccessorNode(const BSTNodePtr x) {
      if (x->right != NULL) {
            return bstMinimumNode(x->right);
      }
      else {
            BSTNodePtr current = x;
            BSTNodePtr parent = current->parent;
            while (parent != NULL && current == parent->right) {
                  current = parent;
                  parent = parent->parent;
            }
            return parent;
      }
}
```

Question 5. [15 marks]

Given the `List` typedefs in Appendix A of this test, write a function `listRemoveDuplicates` with the following prototype:

```
void listRemoveDuplicates(ListPtr l);
```

This function removes duplicate elements from the list pointed to by `l`, and frees any memory used by the nodes containing those elements. Assume that the memory for the nodes in the list has been dynamically allocated using `malloc`.

Examples:

List contents before `listRemoveDuplicates`	List contents after `listRemoveDuplicates`
1, 2, 3, 4, 5, 6	1, 2, 3, 4, 5, 6
1, 1, 2, 2, 3, 3	1, 2, 3
1, 1, 1, 1, 1, 1	1
3, 2, 2, 2, 3, 3	3, 2

```
void listRemoveDuplicates(ListPtr l) {
      ListNodePtr z = l->front;
      ListNodePtr y = NULL;

      while (z != NULL) {
            ListNodePtr x = l->front;
            while (x != z) {
                  if (x->data == z->data) {
                        ListNodePtr t = z->next;
                        free(z);
                        y->next = t;
                        z = y;
                        break;
                  }
                  x = x->next;
            }
            y = z;
            z = z->next;
      }
}
```

Question 6. [15 marks]

Write a **recursive** function (i.e. one that does not contain any loops) `pascalTriangle` with the following prototype:

```
void pascalTriangle(int n);
```

This function prints all of the rows up to the nth row of Pascal's triangle.

You may write (recursive) helper functions for `pascalTriangle`, and are encouraged to do so (perhaps one for generating one number in the triangle and another for generating one row in the triangle). These functions also cannot contain any loops.

You cannot use any global variables or auxiliary data structures (e.g. arrays, linked lists).

Example:

A call to `pascalTriangle(5)` should generate the following output (rows 0 through 5 of Pascal's triangle):

```
row 0:  1
row 1:  1   1
row 2:  1   2   1
row 3:  1   3   3   1
row 4:  1   4   6   4   1
row 5:  1   5   10  10  5   1
```

Note that the numbers on the ends of the rows are all 1. Each other number is the sum of the number directly above it and the number above and to the left.

```
int pascalNumber(int n, int k) {
      return ((k == 0) || (k == n))
            ? 1
            : pascalNumber(n-1, k-1) + pascalNumber(n-1, k);
}

void pascalRow(int n, int k) {
      if (k < 0)
            return;
      pascalRow(n, k-1);
      printf("\t%d", pascalNumber(n, k));
}

void pascalTriangle(int n) {
      if (n < 0)
            return;
      pascalTriangle(n-1);
      printf("row %d:", n);
      pascalRow(n, n);
      printf("\n");
}
```

[END OF TEST]

Appendix A: `List` typedefs

```
/* Represents a node in a singly-linked list. */
struct listnode {

      /* The data stored in this node. */
      int data;

      /* The node following this node. NULL if no node follows this node. */
      struct listnode* next;
};
typedef struct listnode ListNode;
typedef ListNode* ListNodePtr;

/* Represents a list. Implemented using singly-linked nodes. */
struct list {

      /* The node at the front of this list. NULL if this list is empty. */
      ListNode* front;
};
typedef struct list List;
typedef List* ListPtr;
```

Appendix B: BST typedefs

```
/* Represents a node in a linked binary search tree. */
struct bstnode {

      /* The key of this node. */
      int key;

      /* The parent of this node. NULL if this node is the root. */
      struct bstnode* parent;

      /* The left child of this node. NULL if this node has no left
         child. */
      struct bstnode* left;

      /* The right child of this node. NULL if this node has no right
         child.*/
      struct bstnode* right;
};
typedef struct bstnode BSTNode;
typedef BSTNode* BSTNodePtr;

/* Represents a binary search tree. */
struct bst {

      /* The root node of this tree. NULL if this tree is empty. */
      BSTNode* root;
};
typedef struct bst BST;
typedef BST* BSTPtr;
```

Appendix C: String Library Functions

(Source: Appendix D of *C Programming: A Modern Approach* by K.N. King.)

```
char *strcat(char *s1, const char *s2);
```

Appends characters from the string pointed to by `s2` to the string pointed to by `s1`. Returns `s1`.

```
char *strchr(const char *s, int c);
```

Returns a pointer to the first occurrence of the character `c` in the string pointed to by `s`. Returns `NULL` if `c` isn't found.

```
int strcmp(const char *s1, const char *s2);
```

Returns a negative, zero or positive integer, depending on whether the string pointed to by `s1` is less than, equal to, or greater than the string pointed to by `s2`.

```
char *strcpy(char *s1, const char *s2);
```

Copies the string pointed to by `s2` into the array pointed to by `s1`.

```
size_t strlen(const char *s);
```

Returns the length of the string pointed to by `s`, not including the null character.

```
char *strncat(char *s1, const char *s2, size_t n);
```

Appends characters from the array pointed to by `s2` to the string pointed to by `s1`. Copying stops when a null character is encountered or `n` characters have been copied. Returns `s1`.

```
char *strncpy(char *s1, const char *s2, size_t n);
```

Copies the first `n` characters of the array pointed to by `s2` into the array pointed to by `s1`. If it encounters a null character in the array pointed to by `s2`, `strncpy` adds null characters to the array pointed to by `s1` until a total of `n` characters have been written. Returns `s1`.

```
char *strrchr(const char *s, int c);
```

Returns a pointer to the last occurrence of the character `c` in the string pointed to by `s`. Returns `NULL` if `c` isn't found.

```
char *strstr(const char *s1, const char *s2);
```

Returns a pointer to the first occurrence in the string pointed to by `s1` of the characters in the string pointed to by `s2`. Returns `NULL` if no match is found.