University of Toronto
csc148S—Introduction to Computer Science, Spring 1997

Second Term Test — Horton's section

Sample Solutions!

Duration: 50 minutes

Aids allowed: None

Make sure that your examination booklet has 7 pages (including this one). Write your answers in the spaces provided. Write legibly.

Family Name: ____________________ **Lecturer:** ____________________

First Name: ____________________ **Tutor:** ____________________

Student # : ____________________

1. ________ / 6
2. ________ / 8
3. ________ / 9
4. ________ / 7

Total ________ / 30

Question 1

[6 marks in total]

Consider the following types and variable, useful for creating the Iqaluit data structure from assignment 2:

```
typedef struct neighb {
    int           house;      /* House number of this neighbour */
    int           cost;       /* Cost of a road segment to this neighbour */
    struct neighb *next;      /* Pointer to the next neighbour */
} Neighbour;

typedef struct {
    int       x, y;           /* x and y coordinates of this house */
    Neighbour *neighbourList; /* Linked list of neighbours */
} HouseRec;

HouseRec *HouseList;
```

(a) [2 marks] Assume that appropriate space has been allocated for `HouseList`, and that information about a town with 100 houses has been stored in it. Write a single line of C code that will print out the house number of the first neighbour stored for house 17.

```
printf ("%d\n", HouseList[17].neighbourList -> house);
```

(b) [1 mark] Recall the algorithm for paving Iqaluit. What houses are in A?

`All houses that are adjacent to a house in B, but not in B themselves. In other words, all houses that are not in the paved part of town, but are neighbour to some house that is.`

(c) [3 marks] Recall that R was the set of all road segments selected for paving. A road segment is specified by the numbers of the two houses that it connects.

Consider the following possible data structures for implementing R:

- A linked list of structs, sorted in ascending order by the house number of the first house in the segment, or if equal, by the house number of the second house.
- An unsorted linked list of structs.

Which data structure would be better? Circle one.

Sorted linked list | Unsorted linked list

Explain your answer:

- `It would take extra time to keep the list sorted (inserts would require searching for the right spot rather than just adding to the front of the list).`
- `But we would get no benefit from the sortedness because no operation can take advantage of it. (We only add to R; we never search it, for example.)`

Question 2

[8 marks in total]

(a) [6 marks]

Using the data structure from question 1, complete the following function. Its purpose is to add a new neighbour to the front of a given neighbour list. It should return 1 if it failed because memory could not be allocated for the new neighbour, and 0 otherwise.

```
/* 'p' points to the front pointer for the neighbour list.
 * 'house' and 'cost' give information about the new neighbour. */
int addNeighbour (Neighbour **p, int house, int cost)
```

```
{
    Neighbour *temp = (Neighbour *) malloc(sizeof(Neighbour));
    if (temp == NULL)
        return 1;
    else {
        temp -> house = house;
        temp -> cost = cost;
        temp -> next = *p;
        *p = temp;
        return 0;
    }
}
```

(b) [2 marks]

Two calls to this function can be used to record the fact that house 11 and house 15 are adjacent, and that the road segment between them has cost 4. Write those two calls. You may ignore the return value of the function.

```
addNeighbour (&(HouseList[11].neighbourList), 15, 4);
addNeighbour (&(HouseList[15].neighbourList), 11, 4);
```

Question 3

[9 marks in total]

Below is a function that returns the sum of all elements in $A[i..j]$.

```
int sum (int *A, int i, int j)
{
    /* Preconditions: i <= j
     *                any other preconditions ??
     */
    int answer = 0;
    int pos = i;
    while (pos != (j+1)) {
        /*  Loop invariants:
         *  1) i <= pos <= j
         *  2) answer is the sum of all elements in A[i..(pos-1)]
         */
        answer += A[pos];
        pos ++;
    }
    return answer;
    /*  Postcondition: answer returned was
     *                 the sum of all elements in A[i..j].
     */
}
```

(a) [2 marks]
What preconditions are necessary in order for this function to work correctly (that is, to terminate without crashing, and to make the postcondition true)?

```
i >= 0
j < capacity of array A
all elements in A[i..j] inclusive have values
```

(b) [1 mark]
For *any* loop, if you assume that the loop iterates at least once and does terminate, there are two important things that you know to be true after the loop terminates. What are they?

- `The loop invariants must be true.`
- `The exit condition of the loop must be true.`

(c) [5 marks]
Prove the "partial correctness" of function sum. That is, prove that if the preconditions are true and the code does terminate, then the postconditions will be true.
Hint: Part (b) was intended to remind you of some things you can use in this proof.

`Assume that the preconditions are true and that the code does terminate.`

- `When the code terminates, the loop invariants are true (because, by definition, they are true at the beginning and ending of every iteration, including the last one. And nothing has happened since the end of the last one other than that we left the loop).`
- `This means that 1) i <= pos <= j, and 2) answer is the sum of all elements in A[i..(pos-1)].`
- `Also, The exit condition of the loop must be true (otherwise, we couldn't have gotten out of the loop).`
- `This means that pos = (j+1).`
- `So by substitution, we have that answer is the sum of all elements in A[i..((j+1)-1)] = A[i..j].`
- `This answer is returned.`
- `Thus the postcondition is true.`

`So, if the preconditions are true and the code does terminate, then the postconditions will be true.`

(d) [1 mark]
Which of the following proof techniques did you use in your proof? Circle all that were used.

1. proof by contradiction
2. proof by cases
3. $\boxed{\text{conditional proof}}$
4. proof by induction

Question 4

[7 marks]

The following type can be used to build binary trees of strings:

```
typedef struct treenode_struct {
    char  *s;
    struct treenode_struct *left, *right;
} treeNode;
```

Write a recursive function that takes a pointer to the root of such a tree and returns the number of leaf nodes in that tree (or zero if the tree is empty). Do not assume anything about the shape of the tree. The function header has been written for you.

Note: The next page suggests how to tackle this question and gives you room for some rough work.

```
int numLeaves (treeNode *root)
```

```
{
    if (root == NULL)
        return 0;
    else if ((root -> left == NULL) && (root -> right == NULL))
        return 1;
    else
        return numLeaves(root->left) + numLeaves(root->right);
}
```

Rough work. Use this if it's helpful. It will **not** be marked.

When is the answer so simple that we know it without recursing?

```
If the tree is empty: there are no leaves.
```

Write an if-statement to handle the degenerate case(s).

```
if (root == NULL)
    return 0;
```

For the harder case(s), describe the answer in terms of the answer on smaller inputs.

```
If the tree isn't empty: the number of leaves overall is
      the number on the left side + the number on the right side

With only this plus the one degenerate case above, we can
never get any answer other than zero!  We only ever take
sub-answers of zero and add them together.

We need an additional degenerate case for when the tree
has only one node (a root with no kids).  Then the answer is 1.
Code:
     if ((root -> left == NULL) && (root -> right == NULL))
          return 1;
```

Write code for the recursive case(s).

```
If there is a root node and it has a least one child,
     return numLeaves(root->left) + numLeaves(root->right);
```