15-122 : Principles of Imperative Computation, Fall 2013

Written Homework 11 [Update 2]

Due: Thursday, November 21, 2013 by 10pm

Name: ________________________

Andrew ID: ________________________

Recitation: ________________________

The written portion of this week's homework will give you some practice working with more C programming issues and tries. You can either type up your solutions or write them *neatly* by hand, and you should submit your work in class on the due date just before lecture begins. Please remember to *staple* your written homework before submission.

Question	Points	Score
1	7	
2	8	
Total:	15	

Write your answers *neatly* on this PDF (or fill out the TeX handout), and then submit the stapled printout to the handin box Thursday before lecture or on Thursday afternoon or Friday outside of Tom Cortina's office (GHC 4117).

1. **Typecasting and Function Pointers in C**

   Suppose that we are working with the expected implementation-defined implementation of unsigned and signed (2's compliment) `short` (16 bits, two bytes) and `int` (32 bits, four bytes).

(3) (a) We begin with the following declarations:

```
short w = -15;
unsigned short x = 65521;
int y = -65521;
```

Fill in the table below. In the third column, always use four hex digits to represent a `short`, and eight hex digits to represent an `int`. You might find these numbers useful: $2^{16} = 65536$ and $2^{32} = 4294967296$.

Solution:

C expression	Decimal value	Hexadecimal
`w`	-15	`0xFFF1`
`(unsigned short)w`	65521	`0xFFF1`
`(int)w`	-15	`0xFFFFFFF1`
`x`	65521	
`(int)x`		
`(int)(short)x`		
`y`	-65521	
`(unsigned int)y`		

(2) (b) Consider the following C definition for the factorial function:

```
int factorial(int n)
{
    REQUIRES (n >= 0);
    int result = 1;
    for (int i = 1; i <= n; i++) {
        result *= i;
    }
    return result;
}
```

Use `typedef` to define a C type named `int2int` that represents a function pointer that requires an int as its parameter and returns an int as its return type.

Solution:

```
typedef ____________________________________________________;
```

Let the variable `f` be of type `int2int`. (That is, `f` is a function pointer to a function that has one parameter of type `int` and returns a result of type `int`.) Show how to initialize `f` with the address of the `factorial` function given above using the address-of operator.

Solution:

```
int2int f = ___________________________________________________;
```

Write a C instruction that prints out 10! using the variable `f` defined above. Use an explicit derefencing operation on `f` to get to the factorial function.

Solution:

```
printf( "10! = %d\n", __________________________________________);
```

Suppose we wanted to set `x` equal to 8! using the function above. Is the following valid in C? (Yes or No)

```
int x = f(8);
```

Solution:

(2) (c) Suppose we have a (signed) `char` array of length 4 and we want to store that array in a single (4-byte) `int` by storing the `char` array `{1, 2, 3, 4}`, for example, as `0x01020304`. Remember that `char` is an integer type in C.

Write a C function that takes a length-4 `char` array named `F` and condenses it into a single `int` as outlined above. Do not cast directly between signed and unsigned types of different sizes, and make sure your solution works for `char` arrays containing negative values.

Your solution should be clear and straightforward; convoluted code will not receive full credit.

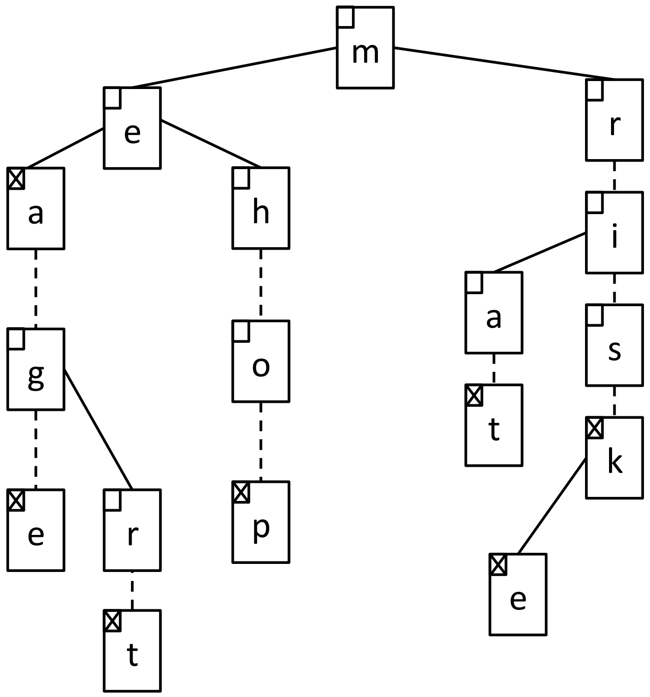
Solution:

```
int condense(char *F) {

}
```

2. **Ternary Search Tries**

Consider the TST shown below.



As in the lecture notes, the dotted lines connect a node to its `middle` child, and solid lines connect a node to its `left` and `right` children. An X in the top left indicates that this node ends a valid word. There could be a link to a corresponding value, like a word definition, for example.

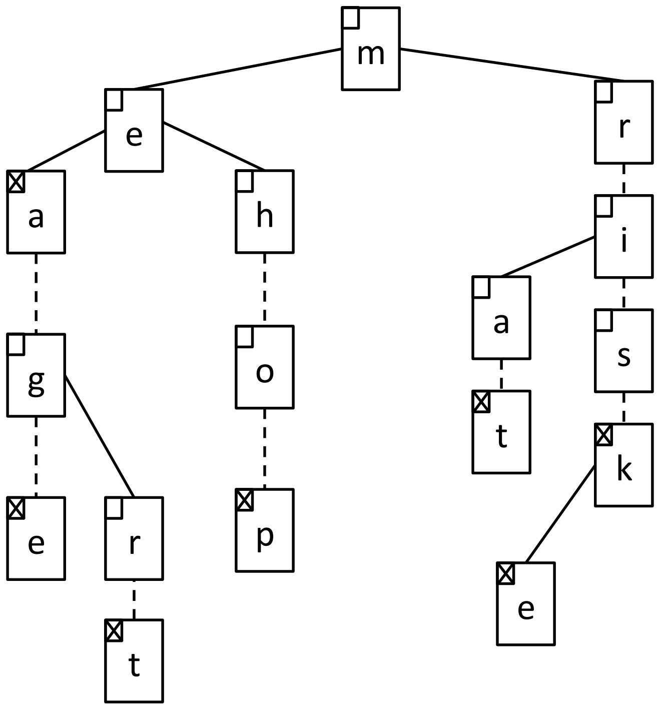
[Update 1, clarification.] The lecture notes and accompanying code describe a desirable invariant of TSTs: that if the middle child is `NULL`, the node has to end a valid word. This TST does not have this property due to the topmost `m` and `e` nodes, but it is not a necessary invariant for safety or correctness. (The insertion and lookup algorithms work even without that invariant.)

(2) (a) List all of the valid words stored in the TST above, in alphabetical order.

Solution:

(3) (b) Add the words `me`, `rake`, `hope`, `hot`, `top`, and `act` to the TST given on the previous page, one at a time, in the order given.

Solution:



(3) (c) For this question, review the published code for tries from lecture.

It is possible to implement `trie_lookup` as an iterative function rather than a recursive one. Fill in the blanks so that the function shown below correctly implements lookup in a TST.

The lines involving the variables `lower` and `upper` are used only to prove that the loop invariant (written in the incorrect location as an assertion in the code below) is preserved. You should not use `lower` or `upper` when filling in the blanks.

Solution:

```
elem trie_lookup(trie TR, char *s) {
    REQUIRES(is_trie(TR));
    REQUIRES(s != NULL);
    tnode *T = TR->root;
    int charmin = 0;
    int charmax = (int)CHAR_MAX + 1;
    int lower = charmin;
    int upper = charmax;

    while (T != ___________________________________) {
       ASSERT(is_tnode(T, lower, upper)); // Loop invariant
       if (*s == T->c) {
           if (*(s+1) == '\0') {

               return ___________________________________________;
           } else {
               lower = charmin;
               upper = charmax;
               s++;

               T = ______________________________________________;
           }
       } else if (_____________________________________________) {
           lower = T->c;

           T = __________________________________________________;

       } else {
           upper = T->c;

           T = __________________________________________________;
       }
    }
    return NULL;
} // Update 2: missing '}' before the 'else if' case
```