Section Solutions 7: Polymorphism and Trees

Problem 1 : Polymorphic ADTs

a) We did manage to store distinct element pointers in our stackADT, but every one of those pointers was a pointer into a stack-allocated array, which will be deallocated when the function exits, like all local variables. Thus the stackADT we returned is filled with pointers to storage that will no longer be valid. Trying to later access the contents of that stackADT will have unpredictable results. Any pointer we store in a polymorphicADT must be valid during our entire span of use of the ADT. If we only happen to use the ADT during the function and not afterwards, we could potentially store stack pointers. More likely, we will want to use the ADT contents beyond the lifetime of this particular function call, in which case the pointers we store should be pointing to things in the heap, which are persistent.

b) The problem here is that we forgot that we had saved the entered numbers in their string representations rather than as integers. The compiler is perfectly happy casting what it knows as a `void*` into an `integer*`. (You say it's an integer? Sure, it's an integer!) The code will go ahead and run as if there were integers at the ends of those pointers even though there were really characters there. This of course produces undesirable results.

The take-home lesson here is that we lose something by using `void*` polymorphism—type checking. Normally, the compiler knows what data type is at the end of the pointer and will do the usual type checking when we try to use or operate on it. However, this feature is lost when dealing with `void*`'s. Thus, `void*` polymorphism is hazardous and require extra care and thought to use.

Problem 2: Binary Trees

a) Parents with one or fewer children are considered globally responsible....

```
int GloballyResponsibleParents(Node *t)
{
   int count = 0;

   if (t == NULL) return 0;

   if (t->left == NULL || t-right == NULL)
      count++;

   return count + (GloballyResponsibleParents(t->left) +
                   GloballyResponsibleParents(t->right));
}
```

b)

```
bool IsEqual(Node *t1, Node *t2)
{
   if (t1 == NULL && t2 == NULL) return TRUE;
   if (t1 == NULL || t2 == NULL) return FALSE;
   return (t1->value == t2->value &&
          IsEqual(t1->left, t2->left) && IsEqual(t1->right, t2->right));
}
```

c) Use a queue for the list of waiting nodes. On each iteration, pull off the first node, visit it, and then enqueue its children (add them to the end of the list). Start by enqueueing the root node and continue until the list has been exhausted.

```
void PrintByLevel(Node *root)
{
   queueADT queue;
   Node *cur;

   queue = NewQueue();
   Enqueue(queue, root);
   while (!IsEmpty(queue)) {
      cur = (Node *)Dequeue(queue);  // need cast from (void *)
      if (cur != NULL) {
         printf(" %d", cur->value);
         Enqueue(queue, cur->left);   // our check (cur != NULL) will deal
         Enqueue(queue, cur->right);  // with NULLs we might enqueue here
      }
   }
   FreeQueue(queue);
}
```

d)

```
/*
 * Function TreeDelete
 * Usage: TreeDelete(&tree, stringkey);
 * ----------------------------------------------------
 * This function attempts to find the node with key.  If it reaches a
 * NULL tree, then the node is not in the tree, and it returns.  If it
 * does find the node, it calls DeleteNode to actually delete the node,
 * or it recursively searches the tree for the key
 */
static void TreeDelete(Node **tptr, string key)
{
   Node *t;
   int cmp;

   t = *tptr;
   if (t == NULL) return;
   cmp = StringCompare(key, t->key);
   if (cmp == 0)
          DeleteNode(tptr);
   else if (cmp < 0)
          TreeDelete(&t->left, key);
   else
          TreeDelete(&t->right, key);
}


/* Function DeleteNode
 * Usage: DeleteNode(&tree);
 * ----------------------------------------------------
```

```
 * This function actually deletes the node pointed to by tptr.  If there are
 * no children, the node is deleted. If there is a single child, the current
 * node is removed, and the pointer to the current node is changed to point
 * to the single child of the current node.  If there are two children, the
 * greatest node of the left subtree is found, and it replaces the info for
 * the current node so that the tree maintains its current structure.
 */
static void DeleteNode(Node **tptr)
{
   Node *t, *nodeToCopy;

   t = *tptr;
   if ((t->left == NULL) && (t->right == NULL)) {  // no children
          *tptr = NULL;
          FreeBlock(t);
   } else if (t->left == NULL) {           // just has right child
          *tptr = t->right;
          FreeBlock(t);
   } else if (t->right == NULL) {   // just has left child
          *tptr = t->left;
          FreeBlock(t);
   } else {                                                    // has both
left & right children
          nodeToCopy = FindGreatestAndRemove(&t->left);
          t->key = nodeToCopy->key;
          t->value = nodeToCopy->value;
          FreeBlock(nodeToCopy);
   }
}


/*
 * Function FindGreatestAndRemove
 * Usage: GreatestTree = FindGreatestAndRemove(&tree);
 * ---------------------------------------------------
 * This function takes a pointer to a tree and finds the greatest element
 * in that tree.  Once it has found the greatest element, it sets the
 * pointer to that node to point to the single child of the greatest
 * node. If there are no children of the greatest node, then the pointer
 * to the GreatestNode is set to NULL.  The greatest node is returned.
 */
static Node *FindGreatestAndRemove(Node **tptr)
{
   Node *t;

   t = *tptr;
   if (t == NULL) return (NULL);
   if (t->right == NULL) {
          *tptr = t->left;
          return (t);
   } else
          return (FindGreatestAndRemove(&t->right));
}
```

Problem 3: Binary search tree analysis

a) The height of the tree is 4 (just one node would be height 1). Ross is at depth 2, and the descendants of Morganstern are Hathaway, Ross, and Weaver.

b)

pre-order	**in-order**	**post-order**
Greene	Benton	Benton
Carter	Carter	Carter
Benton	Greene	Hathaway
Morganstern	Hathaway	Weaver
Hathaway	Morganstern	Ross
Ross	Ross	Morganstern
Weaver	Weaver	Greene

c) 'Del Amico' would be the right child of Carter.

d) Greene was the first node inserted (since it's the root).

Problem 4: Symbol Tables

a) The client code would allocate data for the employeeDataT, then pass a pointer to it as the value to be entered into the table. For example:

```
employeeDataT *empDataPtr;
string employeeName;
...
empDataPtr = GetBlock(sizeof employeeDataT);
... code to initialize the structure and name...
Enter(table, employeeName, empDataPtr);
```