Creating a Generic Data Type in C — Binary Search Tree

For this assignment, you will implement a generic binary search tree in C. Search operations on a binary search tree require that the user data elements stored in the tree be compared. On the one hand, it's desirable to implement a data structure so that the implementation can be used to store whatever type of data the user wants. On the other hand, how can the tree do that without information about the user's data type?

In C, the problem can be solved by employing a *function pointer*. The name of a function is simply a pointer to the first instruction in the implementation of the function, and pointers can be passed as parameters and stored. We will implement a binary search tree as a collection of **`struct`** types (with appropriate associated functions); a first design might look like this:

```
/** Generic binary tree node type.
 *
 *  A BSTNode object is proper iff:
 *    - it is encapsulated in a PayloadWrapper object, and
 *    - lchild is NULL or points to a proper BSTNode object, and
 *    - rchild is NULL or points to a proper BSTNode object
 */
struct _BSTNode {
   struct _BSTNode* lchild;
   struct _BSTNode* rchild;
};
typedef struct _BSTNode BSTNode;

/** Generic binary search tree type.  The implementation depends on three
 *  user-supplied functions:
 *
 *  compare takes pointers to two proper BSTNode objectss; the function
 *            accesses the user payloads, compares them, and returns:
 *              < 0 if user payload on left < user payload on right
 *                0 if user payload on left == user payload on right
 *              > 0 if user payload on left >  user payload on right
 *            (Of course what <, ==, and > mean is entirely up to the user.)
 *
 *  destroy takes a pointer to a proper BSTNode object; the function deallocates all
 *            dynamic memory associated with the payload wrapper and the user's payload
 *
 *  display  takes an open file pointer and a pointer to a proper BSTNode
 *            object and writes a formatted display of the user-defined data object
 *            in the associated wrapper
 *
 *  A BST object is proper iff:
 *    - compare, destroy, and display point to the user-supplied functions
 *      as described, and
 *    - root is NULL, or the user data object in *root is larger than every
 *      user data object in the left subtree of *root, and smaller than every
 *      user data object in the right subtree of *root, and the left and right
 *      subtrees of *root are both proper BST objects
 */
struct _BST {
   // pointer to root node, if any
   struct _BSTNode* root;

   // pointer to the user-supplied functions
   int32_t (*compare)(const BSTNode* const left, const BSTNode* const right);
   void    (*destroy)(BSTNode* pNode);
   void    (*display)(FILE* fp, const BSTNode* const pNode);
};
typedef struct _BST BST;
```

Designing and Implementing a Generic Data Structure

This design requires some explanation. For example, how can user data be stored if the node type makes no provision for it? And, what's the deal with those function pointers?

If you have taken CS 2505 (a prerequisite for this course), you should have been exposed to the design of a generic linked list in C, and parts of the following discussion will be familiar.

Data structure code is usually more complex to implement and test than code for user data types, so we would like to produce an implementation of a binary search tree that could be used, without modification, to store user data of any type whatsoever. In other words, the implementation of the binary search tree should make minimal assumptions about the user's data type.

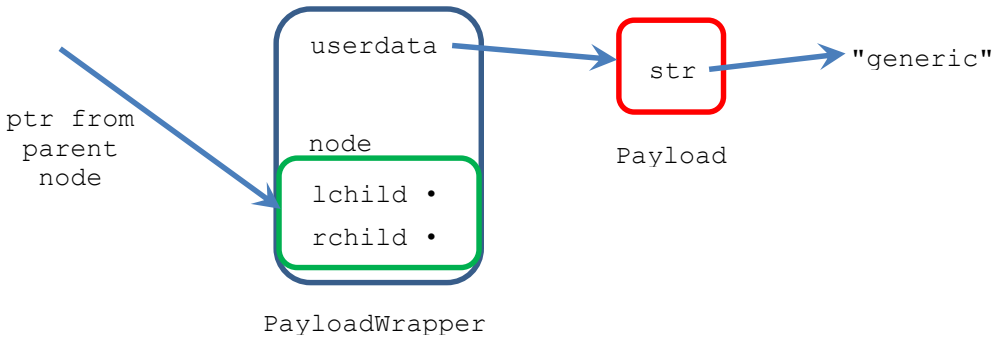
How can a user use this `BST` to store data objects? The answer is actually the key to making the implementation generic: the user data object will actually be stored in a "wrapper" object that encapsulates a user data object and a `BSTNode`. In addition, we'd like to make the wrapper type generic, if possible. We will use the following wrapper design:

```
struct _PayloadWrapper {
   Payload* userdata;   // pointer to a user data object
   BSTNode node;
};
typedef struct _PayloadWrapper PayloadWrapper;
```

A user can make use of the `PayloadWrapper` type by creating a type called `Payload` to hold the desired data; for example, if the user wants to store character strings, the `Payload` type might look like this:

```
struct _Payload {
   char* str;
};
typedef struct _Payload Payload;
```

Then, if the user wants to store the string `"generic"` in the `BST`, the user would create a `Payload` object and a corresponding `PayloadWrapper` object; after insertion, the tree would contain a node, embedded in a wrapper that also contained a pointer to the user's payload object:



Clearly, a user can customize the `Payload` type to support any content that's desired, so this is flexible. But there are issues to solve.

How can the `BST` compare user data objects? Since a binary search tree must compare user data objects, the user must supply some way to do that; that means the user must supply a function that the binary search tree can call to perform the comparisons. Furthermore, when two user data values are compared there are three possible results (smaller, equal, greater). So, the user must provide a function that takes two data values and returns which of the three results occurs. This isn't imposing any severe restriction on the user, since a binary search tree can only store data values that can be compared in this manner.

The user can supply a comparison function as part of the `Payload` type; for our example:

```
int32_t Payload_compare(const Payload* const left, const Payload* const right) {

   return strcmp(left->str, right->str);
}
```

Note this is customized according to the user's design of the contents of a `Payload` object, which is determined by the user. Remember that, when performing a search, the `BST` needs to determine, given two user data objects `A` and `B`, whether `A < B` or `A == B`, or `A > B`. The design of `strcmp()` provides exactly that information:

```
strcmp(A, B) < 0 if A < B (lexicographic ordering)
strcmp(A, B) = 0 if A = B (lexicographic ordering)
strcmp(A, B) > 0 if A > B (lexicographic ordering)
```

If the user chooses different content for a `Payload` object, then the user's comparison function must be designed accordingly, but this should still result in a comparison that behaves line `strcmp()`.

But, `BST` doesn't "know about" the `Payload` interface (that's really the while point of making `BST` generic). Therefore, the `BST` cannot directly call the `Payload` function shown above.

The solution is to have `PayloadWrapper` provide an adaptor, a function that `BST` can call, and that itself calls the comparison function provided by `Payload`:

```
int32_t PayloadWrapper_compare(const BSTNode* const leftNode,
                               const BSTNode* const rightNode);
```

Note how this will work:

- the `BST` can safely call the `PayloadWrapper_compare()` function, since the existence of and the interface of that function have been specified as a requirement by the `BST`
- the BST "receives" the `PayloadWrapper_compare()` function when the `BST` is created, so the function name doesn't even need to match; of course, the return type and the parameters must match
- the `PayloadWrapper_compare()` function only needs to call the function the user provides for the user's `Payload` objects
- the user never needs to make any changes to the `BST` itself

So, the user only needs to implement a data type that supplies an appropriate function (or set of functions) for the `PayloadWrapper` implementation to call. This may require making some adjustments in `PayloadWrapper`, but that isolates changes to the relatively simple code in `PayloadWrapper`. In essence, the `PayloadWrapper` type is an *adaptor* that makes it simple for the user to plug data objects of any type into the existing implementation of the `BST`.

The BST can then call the PayloadWrapper_compare() function just like any function, using the function name from the BST declaration:

```
BSTNode* BST_find(const BST* const pTree, const BSTNode* const userNode) {
    . . .
    return BST_findHelper(pTree->root, userNode, pTree->compare);
}

static BSTNode* BST_findHelper(const BSTNode* pNode, const BSTNode* const userNode,. . .) {

    . . .
    int32_t direction = compare(userNode, pNode);

    if ( direction < 0 ) {
    . . .
```

There are still some issues to work out.

A `BST` has `BSTNode` pointers, so `PayloadWrapper_compare()` must take `BSTNode` pointers as parameters.

But the user-supplied `Payload_compare()` function needs pointers to `Payload` objects, not pointers to `BSTNode` objects.

So, how can `PayloadWrapper_compare()` compute appropriate `Payload` pointers so they can be passed to `PayloadWrapper_compare()`?

This can be accomplished by using pointer arithmetic:

```
/** Given a pointer to a BSTNode object, computes the address of the surrounding
 *  PayloadWrapper object.
 *
 *  Pre:  pNode points to a proper BSTNode, contained in a proper PayloadWrapper
 *
 *  Returns:  a pointer to the surrounding PayloadWrapper
 */
PayloadWrapper* PayloadWrapper_getPtr(const BSTNode* const pNode) {

   return (PayloadWrapper*) ((uint8_t*) pNode - offsetof(PayloadWrapper, node));

}
```

The function uses Standard Library **`offsetof`**`()` macro to determine where the `node` field lies in a `PayloadWrapper` object, and then computes the address of the beginning of the surrounding `PayloadWrapper` object. So, we can use the `Payload_getPtr()` function to implement `PayloadWrapper_compare()`.

Now, some other considerations reveal the need for some additional "external" functions that allow a `BST` to take actions on user data objects:

- deallocate any dynamic memory related to the storage of user data objects
- print a useful display of user data objects (primarily for quick debugging)

We'll discuss the details of this later, but we will require `PayloadWrapper` and `Payload` to implement the following pairs of functions. First, we need a pair of functions to support deallocating dynamic memory:

```
/** Destroys a user payload object
 *
 *  Pre:  pLoad points to a proper Payload object
 *
 *  Post: all dynamic memory associated with *pPayload, and *pLoad have been freed
 */
void Payload_destroy(Payload* const pLoad);

/** Deallocates all memory associated with a PayloadWrapper object.
 *
 *  Pre:  pNode points to a proper BSTNode, contained in a proper PayloadWrapper object
 *
 *  Post: the PayloadWrapper object containing *pNode andthe user data object associated
 *           with that wrapper object have been deallocated
 *
 *  Calls: PayloadWrapper_GetPtr(), Payload_destroy()
 */
void PayloadWrapper_destroy(BSTNode* const pNode);
```

Here, the `PayloadWrapper_destroy()` function simply needs to compute an appropriate `Payload` pointer, from the given `BSTNode` pointer, and then call the user's `Payload_destroy()` function.

Similarly, we need two functions to support displaying the user's data:

```
/** Writes a formatted representation of a Payload object.
 *
 *  Pre:  fp is open on an output device
 *        pLoad points to a proper Payload object
 *
 *  Calls: Payload_display()
 */
void PayloadWrapper_display(FILE* fp, const BSTNode* const pNode);

/** Writes a formatted representation of a Payload object.
 *
 *  Pre:  fp is open on an output device
 *        pLoad points to a proper Payload object
 */
void Payload_display(FILE* fp, const Payload* const pLoad);
```

The user will have designed the actual user data, contained in the user's `Payload` objects, so it's up to the user to:

- determine what elements in the `Payload` objects must be freed
- determine whether the `Payload` objects themselves need to be freed
- determine how to format the contents of a `Payload` object when printing it

The implementation of `PayloadWrapper` will assume the user has implemented these functions correctly.

With this design, it's possible to supply the `BST` with the necessary functionality to compare, destroy, and display the user's data without telling the `BST` any details about that data, simply by plugging the supporting functions provided by `PayloadWrapper` into the `BST`:

```
/** Create a proper, empty binary search tree object.
 *
 *  Pre: compare is the name of a user-defined function satisfying the BST specification
 *       display is the name of a user-defined function satisfying the BST specification
 *       destroy is the name of a user-defined function satisfying the BST specification
 *
 *  Returns: a BST object with NULL root and configured to use the three user-supplied
 *           functions for comparing, destroying and displaying user-defined data objects
 *           stored in the tree
 */
BST BST_create(int32_t (*compare)(const BSTNode* const left, const BSTNode* const right),
               void     (*display)(FILE* fp, const BSTNode* const pD),
               void     (*destroy)(BSTNode* pNode)) {

   BST newTree;
   newTree.root = NULL;
   newTree.compare = compare;
   newTree.display = display;
   newTree.destroy = destroy;
   return newTree;
}
```

A user can then create a new `BST` by calling `BST_create()` as follows:

```
BST T1 = BST_create(PayloadWrapper_compare, PayloadWrapper_display, PayloadWrapper_destroy);
```

`BST_create()` installs the `PayloadWrapper` functions into the new `BST`; this allows a user to create several different binary search trees, each storing different data and using different functions. Note that the user would have to create specialized `Payload` and `PayloadWrapper` types in order to do this. But the underlying BST implementation will not have to change, since it is blissfully unaware of any details of `Payload` and `PayloadWrapper`.

The BST Interface

For this assignment, the `BST` will have the following "public" functions. The first is essentially a constructor that makes a new, empty proper `BST`. An implementation was shown earlier and you are free to use it, but it's not guaranteed to be fully correct, so you should check it carefully.

```
/** Create a proper, empty binary search tree object.
 *
 *  Pre: compare is the name of a user-defined function satisfying the BST specification
 *       display is the name of a user-defined function satisfying the BST specification
 *       destroy is the name of a user-defined function satisfying the BST specification
 *
 *  Returns: a BST object with NULL root and configured to use the three user-supplied
 *           functions for comparing, destroying and displaying user-defined data objects
 *           stored in the tree
 */
BST BST_create(int32_t (*compare)(const BSTNode* const left, const BSTNode* const right),
               void    (*display)(FILE* fp, const BSTNode* const pD),
               void    (*destroy)(BSTNode* pNode));
```

The second function inserts a new element into the tree. Note that `userNode` must point to a `BSTNode` that's embedded in a `PayloadWrapper` object that also contains a pointer to the user's actual data object.

```
/** Inserts user data object into a BST, unless it duplicates an existing object.
 *
 *  Pre:  pTree points to a proper BST object
 *        userNode points to a proper BSTNode object
 *
 *  Returns: true iff the insertion was performed; the implementation will not
 *           insert a new element that is equal to one that's already in the
 *           BST (according to the user-supplied comparison function)
 */
bool  BST_insert(BST* const pTree, const BSTNode* const userNode);
```

The third function searches for a given element in a `BST`. Note that `userNode` must point to a `BSTNode` that's embedded in a `PayloadWrapper` object that also contains a pointer to the user's actual data object. You might wonder why the user would be searching for a data object if the user already has it. In a typical situation, the user's data object will contain multiple fields, only one of which (the *key*) is used for comparisons; in that case, the user will only initialize the key field when creating the wrapper to be used when calling the search function.

```
/** Searches a proper BST for an occurence of a user data object that equals
 *  *pData (according to the user-supplied comparison function).
 *
 *  Pre: pTree points to a proper BST object
 *       pData points to a proper BSTNode object
 *
 *  Returns: pointer to matching user data object; NULL if no match is found
 */
BSTNode* BST_find(const BST* const pTree, const BSTNode* const userNode);
```

The fourth function is a destructor; that is, a function that completely deallocates the dynamic memory for an object:

```
/** Deallocates all dynamic memory associated with a proper BST object.
 *
 *  Pre:   *pTree is a proper BST object
 *  Post:  all the user payloads and payload wrappers associated with *pTree
 *             have been freed
 *         the BST object itself is NOT freed since it may or may not have
 *             been allocated dynamically; that's the responsibility of the caller
 *
 *  Calls: Payload_destroy() to handle destruction of the user's data object
 */
void BST_destroy(BST* const pTree);
```

The final function is a display function, which is supplied as part of the base code you'll be given for the assignment.

```
/** Writes a formatted display of the contents of a proper BST.
 *
 *  Pre: fp is open on an output device
 *       pTree points to a proper BST object
 */
void  BST_display(FILE* fp, const BST* const pTree);
```

You might wonder why no deletion function is listed above; the reason is simply to scale back the scope of the assignment. A complete implementation of a binary search tree would certainly include a deletion function.

The natural way to implement search/insertion/destruction for a tree is to use recursion; that will require that each of those "public" functions be paired with a **`static`** recursive helper function. The design of those is up to you. That said, you should have encountered binary search tree implementations in earlier courses (albeit in a different programming language), and you should feel free to use that knowledge here.

A tar file will be posted containing base code for you to start with, along with testing code that is described later. Be sure to examine what's given closely, since that may give useful hints for how to implement the parts for which you are responsible.

Some Notes on Function Interface Design

Consider the interface for this function:

```
bool  BST_insert(BST* const pTree, const BSTNode* const userNode);
```

In this case, the use of a **`bool`** return type allows the function to indicate whether the operation was completed successfully; an insertion fails if a duplicate value is being inserted, and might fail for other reasons as well.

The function is designed to pass a pointer to a `BST` object, rather than to pass by copy. In this case, it's logically necessary, since the insertion of a new value may modify the original `BST` object. (You should think about why I said "may modify" instead of "will modify".)

The interface also illustrates the complexities of using the **`const`** qualifier in C. The specification of the parameter

```
BST* const pTree
```

implies that the value of the parameter `pTree` cannot be modified within the function, but the value of the pointer's target can be changed via a use of that pointer. This means that if the implementation of the function attempts to change where `pTree` points, there will be a warning (alas, not an error) at compile time.

On the other hand, the specification of the parameter

```
const BSTNode* const userNode
```

prevents the implementation of the function from either changing where `userNode` points or changing the value of the target of `userNode` (again, without a warning).

The use of **`const`** is a good way to avoid careless errors in C code, and is probably underused. But, **`const`** is certainly not foolproof, since you can even eliminate the warnings about violations by writing a suitable typecast.

Testing/Grading Harness

Download the posted tar file, `BSTFiles.tar` from the course website and unpack it on a CentOS 7 system. You should receive the following files:

`README`	brief usage instructions
`dev`	
`c01driver.c`	driver for running all the tests and scoring; see embedded comments
`BST.h`	declaration of `BST` type; do NOT modify this file!
`BST.c`	implementation file for `BST` type; complete the functions as needed
`Payload.h`	declaration of `Payload` type; do NOT modify this file!
`Payload.c`	implementation file for the `Payload` type; complete functions as needed
`PayloadWrapper.h`	declaration of `PayloadWrapper` type; do NOT modify this file!
`PayloadWrapper.c`	implementation file for the `PayloadWrapper` type; complete functions as needed
`Monk.h`	interface for testing/grading code; do NOT modify this file!
`Monk.o`	64-bit CentOS 7 binary for testing/grading code
`runvalgrind.sh`	shell script for testing your solution with Valgrind; see header comment in file
`grading`	
`gradec01.sh`	shell script for performing grading; see embedded comments!
`BSTGrader.tar`	containing the following files used in grading:
`c01driver.c`	
`Payload.h`	
`PayloadWrapper.h`	
`BST.h`	
`Monk.c`	
`Monk.h`	
`Monk.o`	

Unpack the posted tar file, and copy the files from `BSTGrader.tar` into a working directory. You can use the `./dev` subdirectory to work on your solution. Edit the files `BST.c`, `Payload.c`, and `PayloadWrapper.c` as needed to implement your solution. Note that you can enable or disable certain tests by commenting parts of the driver file. You can then compile with the following command:

```
gcc -o c01driver -std=c11 -Wall -W -ggdb3 c01driver.c BST.c Payload.c Payloadwrapper.c Monk.o
```

If you have only implemented some of the specified functions (which would be a consequence of wisely following an incremental development pattern), you should edit `c01driver.c` to comment out the calls to test functions you have not yet completed. You will find it valuable to implement some **`static`** ("private") helper functions. Those must be placed in your `.c` file, and declared there.

Do not modify any of the `.h` files; the original versions will be used in grading your submission, so any changes you have made will be lost, and your submission may not even compile.

If compilation succeeds, you can run tests by executing `c01driver`; examine the various log files that are created for details of the testing. Test data is randomized for each run. There are four components involved in computing your score. The first three are analyzed by running `c01driver`, the fourth will be depend on a Valgrind analysis of your solution. The components will be weighted as follows:

BST creation	10%
BST insertion	40%
BST find	30%
Valgrind	20%

The fourth is based entirely on whether you achieve a clean run on Valgrind. This will depend on the correctness of your implementation of `BST_destroy()`, and the functions it depends on.

When you have tested your implementation enough to have confidence in it, create a tar file containing **only** your `BST.c`, `PayloadWrapper.c`, and `Payload.c` files, and copy that tar file into the same directory as `gradec01.sh` and

`BSTGrader.tar`. We recommend naming your tar file as `PID.tar`, where `PID` is your VT email PID. Execute the grading script:

```
./gradec01.sh <name of your tar file>
```

If all goes well, this will create a file, `PID.txt`, containing complete details of the grading process.

We are supplying the test/grading harness, including the grading script `gradec01.sh`, so that you can be certain that your submission is complete and correct before you make a submission. We will make no allowances for special treatment if you fail to do this.

The requirements related to memory management will be checked by the supplied grading script. Your score for that part will then be assessed manually.

What to Submit

You will submit your completed versions of `BST.c`, `PayloadWrapper.c`, and `Payload.c`, in a flat tar file. The tar file should contain nothing but the specified C files. The `gradec01.sh` file will not work correctly with your tar file if your tar file is incomplete or not structured correctly; that's why we have supplied it.

Your submission will be compiled with the test driver using the command shown above.

Again, we have supplied the testing/grading harness so that you can be sure your solution is complete and correct before you submit it, and we expect you to make use of that harness. We will make no allowances for special treatment if you fail to do this.

The *Student Guide* and other pertinent information, such as the link to the proper submit page, can be found at:

http://www.cs.vt.edu/curator/

Pledge

Each of your program submissions must be pledged to conform to the Honor Code requirements for this course. Specifically, you **must** include the following pledge statement in the submitted file:

```
//  On my honor:
//
//  - I have not discussed the C language code in my program with
//    anyone other than my instructor or the teaching assistants
//    assigned to this course.
//
//  - I have not used C language code obtained from another student,
//    the Internet, or any other unauthorized source, either modified
//    or unmodified.
//
//  - If any C language code or documentation used in my program
//    was obtained from an authorized source, such as a text book or
//    course notes, that has been clearly noted with a proper citation
//    in the comments of my program.
//
//  - I have not designed this program in such a way as to defeat or
//    interfere with the normal operation of the grading code.
//
//    <Student Name>
//    <Student's VT email PID>
```

We reserve the option of assigning a score of zero to any submission that is undocumented or does not contain this statement.

Appendix: Valgrind

Valgrind is a tool for detecting certain memory-related errors, including out of bounds accessed to dynamically-allocated arrays and memory leaks (failure to deallocate memory that was allocated dynamically). A short introduction to Valgrind is posted on the Resources page, and an extensive manual is available at the Valgrind project site (www.valgrind.org).

For best results, you should compile your C program with a debugging switch (`-g` or `-ggdb3`); this allows Valgrind to provide more precise information about the sources of errors it detects. For example, I ran my solution for this project, with one of the test cases, on Valgrind:

```
centosvm BST > valgrind --leak-check=full --show-leak-kinds=all --log-file=vlog.txt
--track-origins=yes -v ./c01driver
```

And, I got good news... there were no detected memory-related issues with my code:

```
==25965== Memcheck, a memory error detector
==25965== Copyright (C) 2002-2017, and GNU GPL'd, by Julian Seward et al.
==25965== Using Valgrind-3.14.0-353a3587bb-20181007X and LibVEX; rerun with -h for copyright info
==25965== Command: ./driver
==25965== Parent PID: 25964
==25965==
. . .
==25965==
==25965== HEAP SUMMARY:
==25965==     in use at exit: 0 bytes in 0 blocks
==25965==   total heap usage: 540 allocs, 540 frees, 10,172 bytes allocated
==25965==
==25965== All heap blocks were freed -- no leaks are possible
==25965==
==25965== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
==25965== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
```

That's the sort of result you want to see when you try your solution with Valgrind.

On the other hand, here's (part of) what I got from an incorrect solution:

```
==26963== Memcheck, a memory error detector
==26963== Copyright (C) 2002-2017, and GNU GPL'd, by Julian Seward et al.
==26963== Using Valgrind-3.14.0-353a3587bb-20181007X and LibVEX; rerun with -h for copyright info
==26963== Command: ./driver
==26963== Parent PID: 26962
==26963==
. . .
==26963==
==26963== HEAP SUMMARY:
==26963==     in use at exit: 4,584 bytes in 191 blocks
==26963==   total heap usage: 540 allocs, 349 frees, 10,172 bytes allocated
==26963==
==26963== Searching for pointers to 191 not-freed blocks
==26963== Checked 69,984 bytes
==26963==
==26963== 24 bytes in 1 blocks are indirectly lost in loss record 1 of 9
==26963==    at 0x4C29EA3: malloc (vg_replace_malloc.c:309)
==26963==    by 0x401143: PayloadWrapper_create (PayloadWrapper.c:7)
==26963==    by 0x4019DE: checkTreeInsertion (Monk.c:186)
==26963==    by 0x4009B5: main (driver.c:28)
==26963==
==26963== 24 bytes in 1 blocks are indirectly lost in loss record 2 of 9
==26963==    at 0x4C29EA3: malloc (vg_replace_malloc.c:309)
==26963==    by 0x401143: PayloadWrapper_create (PayloadWrapper.c:7)
==26963==    by 0x401A4E: checkTreeInsertion (Monk.c:194)
==26963==    by 0x4009B5: main (driver.c:28)
==26963==
==26963== 72 (24 direct, 48 indirect) bytes in 1 blocks are definitely lost in loss record 3 of 9
==26963==    at 0x4C29EA3: malloc (vg_replace_malloc.c:309)
==26963==    by 0x401143: PayloadWrapper_create (PayloadWrapper.c:7)
==26963==    by 0x4016ED: checkTreeInsertion (Monk.c:129)
==26963==    by 0x4009B5: main (driver.c:28)
==26963==
```

```
==26963== 1,176 bytes in 49 blocks are indirectly lost in loss record 4 of 9
==26963==    at 0x4C29EA3: malloc (vg_replace_malloc.c:309)
==26963==    by 0x401143: PayloadWrapper_create (PayloadWrapper.c:7)
==26963==    by 0x4027C4: fillBST (Monk.c:479)
==26963==    by 0x402308: checkTreeFind (Monk.c:363)
==26963==    by 0x400A3E: main (driver.c:39)
==26963==
. . .
==26963==
==26963== LEAK SUMMARY:
==26963==    definitely lost: 96 bytes in 4 blocks
==26963==    indirectly lost: 4,488 bytes in 187 blocks
==26963==      possibly lost: 0 bytes in 0 blocks
==26963==    still reachable: 0 bytes in 0 blocks
==26963==         suppressed: 0 bytes in 0 blocks
==26963==
==26963== ERROR SUMMARY: 4 errors from 4 contexts (suppressed: 0 from 0)
==26963== ERROR SUMMARY: 4 errors from 4 contexts (suppressed: 0 from 0)
```

Do not be deceived by the fact that each of the unfreed allocations was made due to calls from the testing code; the memory leaks are entirely due to errors in the implementation of one or more functions in `BST.c`, or `PayloadWrapper.c`, or `Payload.c`.

Valgrind can also detect out-of-bounds accesses to arrays and uses of uninitialized values; a Valgrind analysis of your solution should not show any of those either.

Change Log Relative to Version 1.00

Version	Posted	Pg	Change
1.00	Jan 21		Base document.