ARIZONA STATE UNIVERSITY

CSE 310 — **Data Structures and Algorithms** — Spring 2016

Project #2

Available 02/23/2016; milestone due 03/17/2016; complete project due 04/05/2016
See Blackboard for submission *time* requirements for your CSE 310 section

In this project you will implement `myMalloc()` and `myFree()`, functions to allocate and free memory from a block of physical memory, and then use these functions to implement some elementary data structures. The goals of this project include:

1. To gain experience implementing hash tables for the symbol table, and implementing (max) heaps for maintaining the free space.
2. To allocate, initialize, and perform operations on some elementary data structures via your memory management functions; these data structures include integers, strings, and binary search trees.
3. To become proficient in understanding and using pointers, and with the notion of coercion.

Note: This project is to be completed **individually**. Your implementation **must** use C/C++ and your code **must** run on the Linux machine `general.asu.edu`.

As always, any dynamic memory allocation **must** be done by yourself, i.e., using `malloc()` and `free()`, or `new()` and `delete()`. You may not use any external libraries to implement any part of this project.

All input is to be read from standard input (`stdin`). Similarly, all output is to be written to standard output (`stdout`). This allows input and output to be redirected from and to a file, respectively. No file operations are allowed.

See §2 for full project requirements. A script will be used to check the correctness of your program. Therefore, absolutely no changes to these project requirements are permitted.
You should use a version control system as you develop your solution to this project, e.g., Dropbox or GitHub. Your code repository should be private to prevent anyone from plagiarizing your work.

1 Your own Memory Management: `myMalloc()` and `myFree()`

Start by allocating one large physical memory block. This block **must** be allocated using `malloc()` or `new()`. Your job in this project is to:

1. Use `myMalloc()` to allocate elementary data structures from this physical memory block.
2. Use `myFree()` to return allocated memory for re-use; hence, you need to maintain free blocks and be able to do compaction of the free space.
3. Perform operations on elementary data structures allocated via `myMalloc()`.

You **must** use a (max) heap to implement the free space ordered by free block size; see §1.4.

You **must** use a hash table to implement the symbol table using the variable name as the key, i.e., the symbol table stores information about the variables allocated using `myMalloc()`; see §1.4.

1.1 Input Format

The format of the input data is as follows:

- A positive integer k giving the size in $n = 2^k$ bytes of the physical memory block to allocate. Use `malloc()` or `new()` to allocate `(char *)` n bytes and insert information about this block (i.e., size n and offset zero), into the heap representing your free space. Initialize each byte in the memory block to a blank character.
- A positive integer t giving the size of the hash table used for the symbol table; see §1.4.
- A positive integer c giving the number of commands that follow.
  - On each of c lines, a valid command is given; see §1.2 and §2.3.1 for command format for the milestone and final project deadline, respectively, and an explanation.

You may assume the format of the input data is correct.

1.2 Commands for Milestone Deadline

The following seven (7) commands are to be implemented for the milestone deadline. Additional commands are to be implemented for the full project deadline; see §2.3 for details.

- `allocate type var [len] value`, write a function named `myMalloc()` to allocate a variable `var` having type `type` from the physical memory block and initialize its value to `value`. For the project milestone, the only valid types are integers `(INT)` of size `sizeof(int) = 4` bytes, and character strings `(CHAR)` of length $1 \leq$ `len` $\leq n$ bytes.
  - Use the *worst-fit algorithm* to allocate memory for the variable `var` in the free space. This first removes the free block of maximum size s from the heap. This free block is split into two blocks, one to hold the value of the variable, the other is free. That is, the first b bytes at the given offset are to be used to store the value of the variable `var`. The second block, of size $s - b$ bytes is free and must therefore be inserted back into the free space with updated offset.

    If there is insufficient memory to allocate the variable `var`, output `Error: Insufficient memory to allocate variable`.
  - Allocation always occurs on 4 byte boundaries. For integer variables $b = 4$ bytes are allocated so no special processing is required. Character strings, however, are of variable length, and must be rounded up to a length that is a multiple of 4 on allocation. That is, if a `len` of 1 is given, a minimum of 4 bytes are allocated; if `len` is 6, then 8 bytes are allocated, and so on.
  - Insert the variable `var` into the symbol table so that it may be retrieved for use in elementary operations.
  - Initialize the value of `var` in the physical memory block to the value `value`. Assume that for integers, the `value` given is always positive.
- `add var v`, add a value `v` to the integer variable `var`, storing the result in `var`. That is, the effect of this command is: `var = var + v;` The value `v` may be a scalar (i.e., a positive integer value), or it may be the identifier of another integer variable. In the latter case, take `v` as the value of this variable.
  - Search the symbol table to retrieve information about the variable `var`.
  - If `var` is not of type `INT`, output `Error: LHS of add must be type INT`.
  - If `v` is a scalar, retrieve the current value of `var` from your physical memory, add the scalar to it, and store the result back in physical memory allocated for variable `var`.

  - If `v` is not a scalar, interpret `v` as a variable name. Search the symbol table to retrieve information about the variable `v`. If `v` is not of type `INT`, output `Error: RHS of an add command must be type integer`. Otherwise, retrieve the current value of `v` from physical memory, add it to the value of `var`, and save the result back in physical memory allocated for variable `var`.

- `strcat var s`, concatenate the character string `s` to the character string `var`, storing the result in `var` if `var` is large enough. That is, the effect of this command is: `strcat( var, s );` The value `s` may be a string constant (i.e., a sequence of zero or more characters surrounded by double quotes), or it may be the identifier of another character string variable. In the latter case, take `s` as the value of this variable.
  - Search the symbol table to retrieve information about the variable `var`.
  - If `var` is not of type `CHAR`, output `Error: LHS of strcat must be type CHAR`.
  - If `s` is a string constant, compute its length. If `var` is not long enough to hold the concatenated strings output `Error: LHS insufficient length to perform strcat`. Otherwise, retrieve the current value of `var` from your physical memory, concatenate the string constant to it, and store the result back in physical memory allocated for variable `var`.
  - If `s` is not a string constant, interpret `s` as a variable name. Search the symbol table to retrieve information about the variable `s`. If `s` is not type `CHAR`, output `Error: RHS of strcat must be type CHAR`. Otherwise, retrieve the current value of `s` from physical memory. Check that `var` is long enough to hold the concatenated strings. If not, output `Error: LHS insufficient length to perform strcat`. Otherwise, retrieve the current value of `var` from your physical memory, concatenate the value of variable `s` to it, and store the result back in memory allocated for variable `var`.
  - It is usual for strings to be terminated with a '\0' (the string termination character) so that functions such as `printf` can detect the end of the string. You should do this to ease string processing; note that it occupies one additional byte in the string.

- `print var`, prints the value of the variable `var`.
  - Search the symbol table to retrieve information about the variable `var`.
  - Print the value associated with the variable `var`. If the variable is type INT, print its value in decimal. If the variable is type CHAR, print the string (see the last point of the `strcat` command above to ease implementation).

- `free var`, write a function named `myFree()` that returns the block occupied in physical memory by variable `var` to free space.
  - Search the symbol table to retrieve information about the variable `var`.
  - Return the block the variable occupies in physical memory to free space. That is the block at the given `offset` of size `noBytes` (see §1.4) is inserted back into free space.
  - You may "clear" the free space if you wish, i.e., reset the bytes to blank characters (or zeros) if it aids in debugging.
  - Now, delete the variable `var` from the symbol table.

- `compact`, coalesce adjacent free blocks of size s_1 and s_2 into a single free block of size $s_1 + s_2$. Compaction terminates when no free blocks can be coalesced with any others. Report a list of free blocks, giving their size, in order of offset.

- `map`, prints a memory map of the entire $n = 2^k$ bytes of memory, similar to that shown in §1.3. Print 64 bytes per line in a constant width font. You may find this command very useful for debugging. You may also find it useful to print the map in hexadecimal (use, e.g., `printf( "Integer i=%d (decimal), i=0x%x (hexadecimal)\n", i, i );` assuming `i` is a integer), and also the ASCII side-by-side.

In all cases, if a search of the symbol table fails, i.e., the variable has not been allocated by `myMalloc`, output `Error: symbol table lookup failed`.

1.3 Example

Consider the following valid example command sequence. Comments are not part of the input, and are only included here for clarification.

```
5 // Allocate a block of 2^5=32 bytes
11 // Size of hash table; should be a prime number
18 // 18 commands follow
allocate INT a 1 // Allocate INT variable a and initialize its value to 1
allocate INT b 2 // Allocate INT variable b and initialize its value to 2
allocate CHAR s 4 "h" // Allocate CHAR variable s of length 3 and initialize its value to h
allocate CHAR s2 2 "!" //  Allocate CHAR variable s2 of length 2 and initialize its value to !
add a 15 // Add scalar 15 to variable a (16)
add a b // Add the value of variable b to a storing the result in a (18)
print a // Print the value of variable a (18)
strcat s "i" // Concatenate string "i" to s storing the result in s
print s // Print value of variable s ("hi")
strcat s s2 // Concatenate string s2 to s storing the result in s
print s // Print value of variable s ("hi!")
strcat s b // Results in an error as b is of type INT
free s // Return the memory allocated to s to free space
print s // Print the value of s (error)
free b // Return the memory allocated to b to free space
print b // Results in an error as b was freed
allocate INT c 10 // Allocate INT variable c and initialize its value to 10
compact // Run compaction; blocks used by b and s are coalesced into one block
```

First, $n = 2^5 = 32$ bytes are allocated using `malloc()` or `new()`; this is the physical memory block. That is, the free space consists of a single block of size 32 at an offset of zero. (The free space is initialized to blanks to aid in debugging.)

Now, let's step through and consider the effect of each command. The `allocate INT a 1` command requires 4 bytes to store the integer. The largest free block is retrieved from free space (here there is only the original block). This block is split into two blocks, one of size 4 at offset zero, for the variable `a`. The other block has size $32 - 4 = 28$, starting at offset $0 + 4 = 4$; this block is inserted back into free space. The memory allocated for variable `a` is then initialized to one. The memory map looks as follows (bytes for integers are shown in hexadecimal, whereas bytes for characters strings are shown as characters);

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
00	00	00	01												

16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31

The `allocate INT b 2` command also requires 4 bytes to store the integer. Again, the largest free block is retrieved from free space. This block is split into two blocks, one of size 4 at offset 4, for the variable `b`. The other block has size $28 - 4 = 24$, starting at offset $4 + 4 = 8$; this block is inserted back into free space. The memory allocated for variable `b` is then initialized to two. The memory map now looks as follows.

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
00	00	00	01	00	00	00	02								

16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31

The `allocate CHAR s 4 "h"` command also uses 4 bytes to store the character string. Again, the largest free block is retrieved from free space. This block is split into two blocks, one of size 4 at offset 8, for the variable `s`. The other block has size $24 - 4 = 20$, starting at offset $8 + 4 = 12$; this block is inserted back into free space. The memory allocated for variable `s` is then initialized to “h”; this includes appending the string termination character. The memory map is now:

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
00	00	00	01	00	00	00	02	h	\0						

16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31

The `allocate CHAR s2 2 "!"` command requests 2 bytes for the character string. Allocation must always occur on 4 byte boundaries, therefore a minimum of 4 bytes are allocated for string `s2`. As before, the largest free block is retrieved from free space. This block is split into two blocks, one of size 4 at offset 12, for the variable `s2`. The other block has size $20 - 4 = 16$, starting at offset $12 + 4 = 16$; this block is inserted back into free space. The memory allocated for variable `s2` is initialized to “!” (including the string termination character) and the memory map is now:

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
00	00	00	01	00	00	00	02	h	\0			!	\0		

16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31

The `add a 15` command, followed by the `add a b` command results in the following memory map (0x13 in hexadecimal is 18 in decimal):

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
00	00	00	13	00	00	00	02	h	\0			!	\0		

16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31

The `strcat s "i"` command, followed by the `strcat s s2` results in the following memory map (both `s` and `s2` are at maximum length):

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
00	00	00	13	00	00	00	02	h	i	!	\0	!	\0		

16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31

The command `free s` returns the 4 bytes of memory with offset 8 allocated to variable `s` back to free space. The free space now contains two free blocks: one of size 16 at offset 16, and a second of size 4 at offset 8 from freeing `s`. If we then execute command `free b`, now a free block of size 4, this one at offset 4, is added to free space. Free space now consists of 3 blocks, two of size 4, and one of size 16. While the freed bytes are indicated by @ in the memory map, unless freed bytes are reset to blank characters, their contents do not change. The contents of freed memory are completely meaningless as the free bytes are not allocated to any variable.

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
00	00	00	13	@	@	@	@	@	@	@	@	!	\0		

16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31

The `allocate INT c 10` command requires 4 bytes to store the integer variable `c`. According to the *worst-fit* algorithm, the largest free block is retrieved from free space; it has size 16 bytes starting at offset 16. This block is split into two blocks, one of size 4 at offset 16, for the variable `c`. The other block has size $16 - 4 = 12$, starting at offset $16 + 4 = 20$; this block is inserted back into free space. The memory allocated for variable `c` is then initialized to 10. The memory map now looks as follows.

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
00	00	00	13	@	@	@	@	@	@	@	@	!	\0		

16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
00	00	00	0A												

At this point, free space consists of three blocks: one of size 4 at offset 4, another of size 4 at offset 8, and a third of size 12 at offset 20. Compaction coalesces the adjacent free blocks at offsets 4 and 8 into one block of size 8 at offset 4. No further coalescing is possible in this case, hence the free space contains two blocks after compaction.

1.4 Symbol Table and Free Space Management

1.4.1 Symbol Table implemented as a Hash Table

You **must** implement the symbol table in this project as a hash table of size t; t should be a prime number and is the second positive integer given in the input. Upon reading the value for t, allocate an array of type `struct symbolTableEntry` of size t using `malloc()` or `new()`. This array is your symbol table.

The hash function is computed as the sum of the ASCII value of each character in the variable name, modulo the hash table size. For example, if a variable is named `Sky` and the hash table size is 11, then the hash function value is: $(83 + 107 + 121) \bmod 11 = 311 \bmod 11 = 3$, because the ASCII value for `S` is 83, for `k` is 107, and for `y` is 121. Variable names are assumed to be at most `SYMBOL_LENGTH` characters in length.

If there is a collision, the hash table is probed linearly until an empty position is found. With *linear probing*, the positions of the table are simply probed in sequence in a circular fashion. That is for the example, if there was a collision at position 3, then check positions $4, 5, \ldots, 10, 0, \ldots 2$ successively for an empty position. If the hash table is full, then your program should exit gracefully and print `Error: Hash table full, exiting.`

1.4.2 Free Space implemented in a (Max) Heap

Upon reading k, a physical block of memory of size $n = 2^k$ bytes, i.e., `(char *)`, must be allocated using `malloc()` or `new()`. Initialize each byte in the memory block to a blank character (or to zeros).

You **must** implement the free space using a (max) heap; the heap is an array of dynamic size of `struct heapEntry`. Initially, the free space consists of a single block of size $n = 2^k$ bytes with offset zero. The heap only stores the size of the free blocks, and their offsets, keyed on `blockSize`.

1.4.3 `defns.cc` File

A file named `defns.cc` is provided for you containing the following constant and structure definitions to use in this project.

```
#define SYMBOL_LENGTH 20 // Maximum length of a variable name

#define INT 0 // Integer type
#define CHAR 1 // Character string type
#define BST 2 // Binary search tree type

#define STRTERM '\0' // String termination character

struct heapEntry{
   int blockSize; // Size of free block in bytes
   int offset; // Start address of free block given as offset into memory block
};
```

```
struct symbolTableEntry{
   char symbol[ SYMBOL_LENGTH ]; // Variable identifier
   int type; // The type of the variable: one of INT, CHAR, or BST
   int offset; // Start address of variable in memory given as offset
   int length; // Size in bytes of to store variable
};
```

1.5 Coercion

Your physical memory is allocated as `(char *)`, yet your `myMalloc()` function store elementary data structures such as integers, character strings, or binary search trees (see §2.3.1) in this memory.

You will need to perform explicit type conversions on the `(char *)` bytes. These can be forced ("coerced") in any expression with a construct called a *cast*. In the construction:

(type-name) expression

the *expression* is converted to the named type by the conversion rules of the C/C++ programming language. The precise meaning of a cast is in fact as if *expression* were assigned in a variable of the specified type, which is then used in place of the whole construction.

A file named `cast.cc` contains a small program showing how an array of bytes may be overlaid by two different structures, each containing different types. You can use something similar in this project to overlay the appropriate structure on the bytes allocated to a particular type in the physical memory.

2 Program Requirements for Project #2

1. Write a C/C++ program that implements all of the commands described in §1.2 for the project milestone, and adds one commands and extends the functionality of three other commands (see §2.3) for the final project deadline. Input is provided in the format described in §1.1.
2. Provide a `Makefile` that compiles your program into an executable named `p2`. This executable must be able to run commands read from standard input directly, or from a script file redirected from standard input (this should require no change to your program).

Sample input files that adhere to the format described in §1.1 will be provided on Blackboard; use them to test the correctness of your program.

2.1 Submission Instructions

All submissions are electronic. This project has two submission deadline dates. The *time* each is due depends on your CSE 310 section; see Blackboard for details.

1. The milestone deadline date is on Thursday, 03/17/2016.
2. The final project deadline date is on Tuesday, 04/05/2016.

It is your responsibility to submit your project well before the time deadline!!! Late projects will not be accepted. Do not expect the clock on your machine to be synchronized with the one on Blackboard!

Multiple submissions are allowed. The last submission will be graded.

2.2 Requirements for Milestone Deadline

By the milestone deadline, your project must implement the commands described in §1.2.

Submit electronically, on Thursday, 03/17/2016 using the submission link on Blackboard for the Project #2 milestone, a zip[1] file named `yourFirstName-yourLastName.zip` containing the following:

Project State (5%): In a folder (directory) named `State` provide a brief report (.txt, .doc, .docx, .pdf) that addresses the following:

1. Describe any problems encountered in your implementation for this project milestone.
2. Describe any known bugs and/or incomplete command implementation for the project milestone.
3. While this project is to be complete individually, describe any significant interactions with anyone (peers or otherwise) that may have occurred.
4. Cite any external code bases, books, and/or websites used or referenced.

Implementation (50%): In a folder (directory) named `Code` provide:

1. In one or more files, your well documented C/C++ source code implementing the commands required for this project milestone.
2. A `Makefile` that compiles your program to an executable named `p2` on the Linux machine `general.asu.edu`. Our TA will write a script to compile and run all student submissions on `general.asu.edu`; therefore executing the command `make p2` in the `Code` directory must produce the executable `p2` also located in the `Code` directory.

Correctness (45%): The correctness of your program will be determined by running a series of commands in the format described in §1.1. Sample input will be provided to you on Blackboard prior to the deadline for testing purposes. As described in the introduction to this project, your program must take input from standard input directly, or from a file redirected from standard input and write output to standard output. **You must not use file operations to read the input!**

The milestone is worth 30% of the total project grade.

2.3 Requirements for Final Project Deadline

2.3.1 Additional Commands for Final Project Deadline

For the final project deadline, you must support an additional type in addition to `INT` and `CHAR`. This type is a *binary search tree* (BST).

For this project, we use the `struct bstNode` definition below for a node in a BST; it is 12 bytes in size. Different from usual, the "pointers" to the left and right children of a node are given by offsets into your physical memory block.

```
struct bstNode{
   int key; // A node of a binary search tree ordered on this key value
   int left; // Left child
   int right; // Right child
};
```

For the final project deadline, one new command is introduced, and the `allocate`, `free`, and `print` commands of §1.2 are extended to support BSTs on (distinct) integer keys as follows:

- `allocate type var [len] value`, add support in the `myMalloc()` function to allocate a variable `var` having type `BST`. The variable `var` is the root of the BST, and its key value is initialized to `value`.

[1]**Do not** use any other archiving program except `zip`.

  - As before, use the *worst-fit algorithm* to allocate memory for the variable `var` from the free space. A node in a BST is 12 bytes in size. If there is insufficient memory to allocate the variable `var`, output `Error:  Insufficient memory to allocate variable`.
  - As before, allocation always occurs on 4 byte boundaries. For BSTs, $b = 12$ bytes are allocated so no special processing is required.
  - Insert the variable `var` into the symbol table so that it may be retrieved for operations on the BST; `var` is the root node of the BST.
  - Initialize the key field of the root node `var` in the memory block to the (distinct integer) value `value`; initialize the left and right "pointers" to -1. Here, -1 serves the same function as a NIL pointer in the usual implementation of BSTs.

- `insert root v`, implement a new command to insert the (distinct integer) value `v` into the BST rooted at node `root`.
  - Search the symbol table to retrieve information about the root node of the BST `root`. If `root` is not type `BST`, output `Error:  can only insert into BST`.
  - Follow the usual BST insertion operation to insert `v` as a leaf node into the tree rooted at `root`.

- `print var`, add support to the `print` command to print a BST via an inorder traversal of the BST rooted at `var`.
  - Search the symbol table to retrieve information about the root node of the BST `var`.
  - Use it to perform an inorder traversal of the tree; this is most easily accomplished through a recursive function.

- `free var`, add support to the `free` command to free *all* the nodes that form the BST rooted at `var`. This is most easily accomplished through a recursive function; free a leaf node when one is encountered, update its parent and continue.

For example, suppose that physical memory contains $n = 2^6 = 64$ bytes and that the free list consists of this single block. Now consider the following sequence of commands:

```
allocate BST root 5 // Create a BST with key of root node equal to 5
insert root 3 // Insert 3 as left child of tree rooted at root
insert root 7 // Insert 7 as right child tree rooted at root
insert root 6 // Insert 6 as left child of node 7 in tree rooted at root
print root // Perform an inorder traversal of the tree rooted at root, printing keys along the way
```

The command `allocate BST root 5` allocates a root node for the binary search tree of size 12 bytes. The largest free block is retrieved from free space. This block is split into two blocks, one of size 12 at offset zero, for the variable `root`. The other block has size $64 - 12 = 52$ bytes, starting at offset $0 + 12 = 12$; this block is inserted back into free space. The key value for the root node `root` is initialized to 5, and its left and right "pointer" values are initialized to -1 (-1 is 0xFFFFFFFF in hexadecimal). The memory map looks as follows:

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
00	00	00	05	FF	FF	FF	FF	FF	FF	FF	FF				
16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
32	33	34	35	36	37	38	39	40	41	42	43	44	45	46	47
48	49	50	51	52	53	54	55	56	57	58	59	60	61	62	63

The command `insert root 3` first looks up the root node `root`. Since the key to insert is less than the key value of the root, $3 < 5$, insertion should occur in the left subtree of the root. However, the left child of the root node is -1, indicating the insertion occurs as the left child of the root. Therefore, we must allocate a new leaf node. We split the one free block of size 52 into two blocks, one of size 12 at offset 12, for the new leaf node. The other block has size $52 - 12 = 40$ bytes, starting at offset $12 + 12 = 24$; this block is inserted back into free space. The key value for the leaf being inserted is initialized to 3, and its left and right "pointer" values are initialized to -1. We must also link the left child "pointer" of the root to "point" to this newly created leaf node. Here, "pointers" are given by offsets, so the `left` field of the root node is set to 12, the offset where the new leaf was allocated. The resulting memory map is:

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
00	00	00	05	00	00	00	0C	FF	FF	FF	FF	00	00	00	03
16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
FF	FF	FF	FF	FF	FF	FF	FF								
32	33	34	35	36	37	38	39	40	41	42	43	44	45	46	47
48	49	50	51	52	53	54	55	56	57	58	59	60	61	62	63

The command `insert root 7` first looks up the root node `root`. Since the key to insert is greater than the key value of the root, $7 > 5$, insertion should occur in the right subtree of the root. However, the right child of the root node is -1, indicating insertion occurs as the right child of the root. We must allocate a new leaf node. We split the one free block of size 40 into two blocks, one of size 12 at offset 24, for the new leaf node. The other block has size $40 - 12 = 28$ bytes, starting at offset $24 + 12 = 36$; this block is inserted back into free space. The key value for the leaf being inserted is initialized to 7, and its left and right "pointer" values are initialized to -1. We must also link the right child "pointer" of the root to "point" to this newly created leaf node; offset of 24 in decimal is 0x18 in hexadecimal. The resulting memory map is:

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
00	00	00	05	00	00	00	0C	00	00	00	18	00	00	00	03
16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
FF	FF	FF	FF	FF	FF	FF	FF	00	00	00	07	FF	FF	FF	FF
32	33	34	35	36	37	38	39	40	41	42	43	44	45	46	47
FF	FF	FF	FF												
48	49	50	51	52	53	54	55	56	57	58	59	60	61	62	63

Finally, the command `insert root 6` first looks up the root node `node`. The key to insert is greater than the key value of the root, $6 > 5$, so insertion should occur in the right subtree of the root. The "pointer" of the right subtree is not -1, indicating we need to move into the right subtree; hence we fetch the node at the given `right` offset of 0x18 (or index position 24 in decimal). The node at index position 24 has a key value of 7; the value to insert is less than this value $6 < 7$, so insertion should occur in the left subtree of this node. However, the "pointer" of the left subtree equals -1, indicating the left subtree is empty. Hence, a new leaf node should be allocated and inserted here. We split the one free block of size 28 into two blocks, one of size 12 at offset 36, for the new leaf node. The other block has size $28 - 12 = 14$ bytes, starting at offset $36 + 12 = 48$; this block is inserted back into free space. The key value for the leaf being inserted is initialized to 6, and its left and right "pointer" values are initialized to -1. We must also link the left child "pointer" its parent to "point" to this newly created leaf node; offset of 36 in decimal is 0x24 in hexadecimal. The resulting memory map is:

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
00	00	00	05	00	00	00	0C	00	00	00	18	00	00	00	03
16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
FF	FF	FF	FF	FF	FF	FF	FF	00	00	00	07	00	00	00	24
32	33	34	35	36	37	38	39	40	41	42	43	44	45	46	47
FF	FF	FF	FF	00	00	00	06	FF	FF	FF	FF	FF	FF	FF	FF
48	49	50	51	52	53	54	55	56	57	58	59	60	61	62	63

The output of the `print root` command is the sequence of keys: 3 5 6 7.

2.4 Submission Instructions for Final Project Deadline

Submit electronically, on Tuesday, 04/05/2016 using the submission link on Blackboard for the complete Project #2, a zip[2] file named `yourFirstName-yourLastName.zip` containing the following:

Project State (5%): Follow the same instructions for Project State as in §2.2.

Implementation (40%): Follow the same instructions for Implementation as in §2.2.

Correctness (55%): The same instructions for Correctness as in §2.2 apply except that the input files will exercise all commands (and command extensions) rather than a subset of them.

3 Marking Guide

The project milestone is out of 100 marks.

Project State (5%): Summary of project state, use of a zip file, and directory structure required (i.e., a folder/directory named `State` and `Code` is provided).

Implementation (50%): 40% for the quality of implementation in your code including proper memory management, 10% for a correct `Makefile`.

Correctness (45%): 40% for correct output on several files of sample input, 5% for redirection from standard input.

The full project is out of 100 marks.

Project State (5%): Summary of project state, use of a zip file, and directory structure required (i.e., a folder/directory named `State`, `Report`, and `Code` is provided).

Implementation (40%): 30% for the quality of implementation in your code, 10% for a correct `Makefile`.

Correctness (55%): 40% for correct output on several files of sample input, 5% for redirection from standard input.

[2] **Do not** use any other archiving program except `zip`.