CSE 220: Systems Fundamentals I

Homework #4

Fall 2016

Assignment Due: Tuesday, November 22, 2016 by 11:59 pm

Assignment Overview

The focus of this homework assignment is on writing recursive functions and traversing non-linear data structures, i.e., binary trees. This assignment also reinforces MIPS function calling and register conventions.

Please read the assignment completely before implementing any of the functions.

You **MUST** implement all the functions in the assignment as defined. It is OK to implement additional helper functions of your own in the `hw4.asm` file.

⚠ **You MUST follow the efficient MIPS calling and register conventions. Do not brute-force save registers! You WILL lose points.**

⚠ Do **NOT** rely on changes you make in your main files! We will test your functions with our own testing mains. Functions will not be called in the same order and will be called independent of each other.

⚠ Do not submit a file with the functions/labels `main` or `_start` defined. You will obtain a ZERO for the assignment if you do this.

ℹ If you are having difficulties implementing these functions, write out the pseudocode or implement the functions in a higher-level language first. Once you understand the algorithm and what steps to perform, then translate the logic to MIPS.

ℹ When writing your program, try to comment as much as possible. Try to stay consistent with your formatting. It is much easier for your TA and the professor to help you if we can figure out what your code does quickly.

Getting started

Download `hw4.zip` from Piazza in the Homework section of Resources. This file contains `hw4.asm` and 5 different main files, which you need for the assignment.

At the top of your `hw4.asm` program in comments put your name and SBU ID number.

```
# Homework #4
# name: MY_NAME
# sbuid: MY_SBU_ID
```

How to Test Your Functions

To test your functions, simply open the provided `hw4.asm` and one of the `main` files in MARS. Assemble the `main` file and run the file. MARS will take the contents of the file referenced with the `.include hw4.asm` at the bottom of the file and add the contents of your file to the main file before assembling it. Once the contents have been substituted into the file, MARS will then assemble it as normal.

Each of the `main` files provided includes basic tests and data structures to test the functions you will implement for this homework assignment. The files contain multiple sample data structures to test with in the `.data` sections. You will need to modify the main programs to test on the different data structures provided. Comments are included to assist in checking your implementations.

You may modify this file, or create your own files to test your functions independently.

⚠ Your assignment will not be graded using the provided main. Any modifications to `main` files will not be graded. You will only submit your `hw4.asm` file via Sparky. Make sure that all code require for implementing your functions (`.text and .data`) are included in the `hw4.asm` file!

⚠ Make sure to initialize all of your values within your functions! Never assume registers or memory will hold any particular values!

Preliminaries

Binary Trees

In computer science, the **tree** is an important data structure that arises in many applications: database management systems, machine learning, compilers, among others. If you are not familiar with trees, take a moment to read through this linked web page.

In particular, a **binary tree** is a tree in which each node has 0, 1 or 2 children. Every node, except the *root node* has exactly one parent node. A node with no children is called a *leaf node*. A node that has at least one child is called an *internal node*. A child node is either the *left child* node or *right child* node of its parent node. In this assignment we will concern ourselves with binary trees containing at least 1 node (i.e., the *root node*) and at most 255 total nodes. We will also work with a special kind of binary trees, namely **binary search trees**.

Binary Search Trees

A **binary search tree** (BST) is a binary tree which imposes a particular organization of the nodes in the tree. We assume that each node contains a *value* (also known as *key*). In this assignment, this value will be a 16-bit two's complement integer. In a BST, the value in each node must be greater than all values stored in the left sub-tree, and less then or equal to all values in the right sub-tree. Note that this data structure is recursive in nature: the left sub-tree and right sub-tree of a given node are themselves both BSTs.

Representing Binary Trees

There are several ways of representing binary trees in the memory of the computer, but we will confine ourselves to a basic one that uses an array to store all information. In particular, the data and "links" that connect parent nodes with their children will be stored in an array between 1 and 255 words in length (inclusive). Each node consists of 32 bits (1 word). This word is split into 3 parts: `Left Node` index, `Right Node` index, and `Node Value`. The lowest 16 bits contain the two's complement value of the node in the tree. The next 8 bits give the index value in the word array for the right node. The most significant 8 bits are the index value in the word array for the left node.

⚠ NOTE: Nodes of the tree can be stored in ANY array index. They are not stored in the array in sequential order.

Visually, each node looks like this in memory.

`Byte 3`	`Byte 2`	`Byte 1`	`Byte 0`
Left Node	Right Node	Node Value	
8 bits	8 bits	16 bits	

Assume the above node is stored in memory at address 0x400. The node value would take up two bytes (e.g., 0x400 and 0x401). The right node index would be one byte and reside at address 0x402, and the left node index would be one byte and reside at address 0x403.

A binary tree will thus be defined as a word array in memory. Since each node of the tree is a word in length and must be easily accessible, the nodes in the array will be aligned on word boundaries. The left and right node index values are the indexes of the left child node and right child node of this node in the array. As such, the 8-bit `Left Node` and `Right Node` values will be interpreted as unsigned integers. An index value of 255 for either `Left Node` or `Right Node` indicates that this node does not have a left or right child node, respectively. This means that the array can only contain 255 different nodes at once. Thus, index 255 of the node array is unreachable. To index into the nodes array, you use the following formula:

```
offset = index × size_of_element
```

Simply take this offset (in number of bytes) and add it to the base address of the array to obtain the address of the node to fetch:

`addr_of_node = base_addr + offset`

Perform a `lw` of that word and then use bitwise operations to extract the three parts of the node: left node index, right node index and data value.

As an example, assume the base address of the node array is 0x300 and the word value stored at address 0x400 was `0x2E72A823`. The half-word (16 bits) `0xA823` would represent the two's complement node value, -22493$_{10}$. The next byte `0x72` would represent the unsigned, 8-bit, right index 114$_{10}$. This means that this node's right child is stored at index 114 in the node (word) array. The address of the right child node in the array is `0x300 + 114*4 = 0x300 + 0x1C8 = 0x4C8`. The final byte `0x2E` would represent the unsigned, 8-bit, left index 46$_{10}$. This means that this node's left child is stored at index 46$_{10}$ in the node array, address `0x3B8`.

Part 1 - Pre-Order Traversal of Binary Trees

For the first part of the assignment, you will implement a recursive function that will traverse nodes of a binary tree in pre-order fashion. Please watch the following Youtube video to refresh your knowledge of pre-order traversal.

We have given you pseudocode for the function `preorder` below that traverses a binary tree using pre-order traversal. Translate the algorithm into MIPS assembly language. The function is called initially on the root of the tree and then recursively descends the tree in pre-order fashion.

The function `itof` takes a two's complement number and writes it to the file as a string of characters. This is a helper function you will need to write on your own.

⚠ NOTE: `itof` function WILL NOT be independently tested during grading.

```
/**
* This function visits the nodes in a pre-order traversal and writes the
* value of each node to file.
*
* @param currNodeAddr, Address of the node being traversed
* @param nodes, Base address of array of nodes
* @param fd, File descriptor of opened file
*/
public static void preorder(Node currNodeAddr, Node[] nodes, int fd) {

     int nodeValue = currNodeAddr.value; // Get the 16-bit integer value

     itof(fd, nodeValue); // Write the node's value to file
     write(fd, "\n", 1);  // Write a newline to file
```

```
    int nodeIndex, offset;
    nodeIndex = currNodeAddr.left; // Fetch the 8-bit index Left Node

    // Check for left node
    if (nodeIndex != 255) {
         // Determine the address of the left child in node array
         int leftNodeAddr = nodes + leftOffset;
         preorder(leftNodeAddr, nodes, fd);
    }

    nodeIndex = currNodeAddr.right; // fetch the 8-bit index Right Node

    // check for right node
    if (nodeIndex != 255) {
         // Determine the address of the right child in node array
         int rightNodeAddr = nodes + rightOffset;
         preorder(rightNodeAddr, nodes, fd);
    }
    return;
}
```

⚠ Your function may assume the array of nodes will be valid. This means that a node index will not point to an index that is outside the array given or refer to a node that contains invalid data (i.e., garbage).

⚠ Note that the root of the tree can be at any index of the array. We may call your function with the root node of the tree or any other node to traverse and print a sub-tree, but the address of the beginning of the `nodes` array will always be given.

⚠ Since `preorder` is a recursive function, `preorder` **MUST** `jal preorder`.

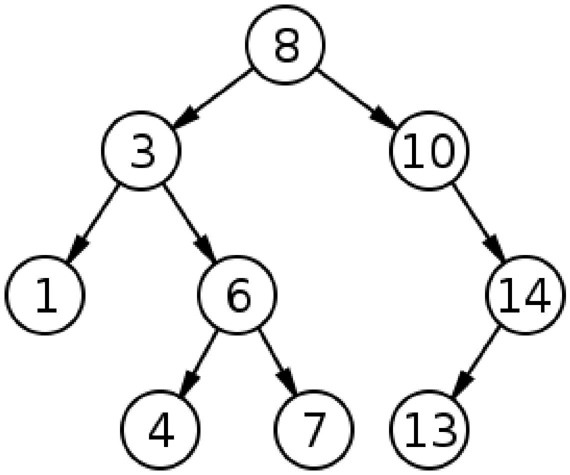
⚠ Your function may assume that the file descriptor is valid, meaning that the file is already open and and writable (for output). Your implementation of the function must **NOT** close the file before returning. Rather, your main program **MUST** close the file prior to exiting main.

⚠ The write function used in the pseduo code is a function available in the C programming language. The first argument is the file descriptor. The second argument is the character string to write to the file. The last argument is the number of bytes to write. You must implement this using syscall 15 in MARS.

For your reference:

Service	Code in $v0	Arguments	Results
open file	13	$a0 = address of null-terminated filename string $a1 = flags $a2 = mode	$v0 contains file descriptor (negative if error). See note below table
write to file	15	$a0 = file descriptor $a1 = address of char buffer to write to file $a2 = maximum num of characters to write	$v0 contains number of characters written (negative if error)
close file	16	$a0 = file descriptor	

Example: If you call `preorder` with the array of nodes with the label `nodes2` given in the sample `main_part1.asm`, you will get the output given below the diagram. The tree looks like the following:



Sample file output:

```
8
3
1
6
4
7
10
14
13
```

Part 2 - Constructing Binary Search Trees

In the remainder of the assignment we will implement functions that work with binary search trees, such as adding and removing nodes in BSTs. Since these functions will change the structure of a BST, we need some way to keep track of which words in the nodes array contain valid data (i.e., valid nodes) and which words are unused (and thus contain garbage).

To help us keep track of which words are available in an array of nodes, we will use a 1 to 255-bit block of memory we will call the "flag array". The flag array will be allocated in a main file's `.data` section as a byte array with a corresponding `maxSize` value in bits. The `maxSize` value denotes the maximum number of nodes possibly stored in the nodes array with which the flag array is paired (i.e., the length of the nodes array). If the `maxSize` value is a number that is not divisible by 8 (meaning that the flag array size is not exactly byte divisible), the flag array will be allocated with the number of bytes required to account for all flag indices. For example,

- If the flag array/nodes array `maxSize == 1`, then a 1-byte flag array is allocated.
- If the flag array/nodes array `maxSize == 10`, then a 2-byte flag array is allocated.
- If the flag array/nodes array `maxSize == 16`, then a 2-byte flag array is allocated.
- If the flag array/nodes array `maxSize == 41`, then a 6-byte flag array is allocated.
- If the flag array/nodes array `maxSize == 255`, then a 32-byte flag array is allocated.

`flags[i] = 0` indicates that `nodes[i]` DOES NOT currently contain valid data and is therefore free to be used to store a new node. `flags[i] = 1` indicates that `nodes[i]` represents a valid node in a binary tree.

Take note that the `maxSize` value indicates that the flag array and nodes array contain indices `0` to `maxSize - 1`.

⚠ The bit indices of the flag array are stored in little endian order, meaning the least significant bit of a flag array corresponds to `i == 0`. Bit 7 (the most-significant bit of the first byte) corresponds to `i == 7`, and so on.

In the previously mentioned examples,

- When `maxSize == 1`, indices 1 through 7 are ignored in the flag array.
- When `maxSize == 10`, indices 10 through 15 are ignored in the flag array.
- When `maxSize == 16`, no indices are ignored in the flag array since the flag array/nodes array size is byte divisible.
- When `maxSize == 41`, indices 41 through 47 are ignored in the flag array.
- When `maxSize == 255`, index 255 is ignored in the flag array.

Below are examples for initializing the flag arrays mentioned. All flag bits are initialized to `0` if the corresponding nodes array does not contain a valid binary tree yet.

```
.align 1
flag_array_size1: .byte 0 # 1 byte
flag_array_size16: .byte 0, 0 # 2 bytes
flag_array_size255: .byte 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0 # 32 bytes
```

⚠ `flags` is only guaranteed to be on a **byte boundary**, not a word boundary like `nodes`.

Any provided nodes array of a specified size will have a provided corresponding VALID flag array. For example, if a nodes array of size 16 has valid binary tree nodes at indices 1, 7, 10, and 14, the flag array provided will be:

```
.align 1
flag_array_size16: .byte 0x82, 0x44
# byte 0: 10000010, byte 1: 01000100
```

When adding a new node to a binary search tree, we must use `flags` to find a free place in `nodes` to store the new node. First, implement a simple linear search that finds the **first** free node, starting the search from index `0` of `flags`.

```
/**
* This function traverses the flag bit-array with a linear search starting
* at flags[0] to find the next free position in the nodes array to add a
* node.
* @param flags, Base address of the maxSize-bit flag array.
* @param maxSize, The size of flags in valid (not ignored) bits.
* @return The index of the first 0 bit in flags (range: [0,maxSize - 1]),
*         or -1 if there is no free index.
*/
public static int linear_search(byte[] flags, int maxSize)
```

⚠ Your function may assume that the given flag array is actually `maxSize` bits in length.

⚠ Your function may assume that `maxSize` is between 1 and 255 (inclusive).

⚠ If `linear_search` is being used in conjunction with `add_node` defined below, meaning `flags` is paired with `nodes` and linear searching is being used to find the next index to add a BST node to, you may assume that `flags` and `nodes` are the same size. If `linear_search` is tested alone, `flags` does not have to correspond to any array of nodes.

We will also need a function mark an entry in the flag array as used or unused. The `set_flag` function will set a 1 to `flags[i]` when we want to use `nodes[i]` to store a new node into the BST. Likewise, we would set a 0 in `flags[i]` when we are deleting `nodes[i]`.

```
/**
* This function sets a specified flag in the flag bit-array with the
* least-significant bit of the value provided.
*
* @param flags, Base address of the flag array.
* @param index, The index position to set in the flag array.
* @param setValue, Use the least-significant bit of setValue to set the
*       flag to either 1 or 0
* @param maxSize, The size of flags in valid (not ignored) bits.
* @return 1 if success, or 0 if failed because the index is not in the
*        range [0,maxSize - 1].
*/
public static int set_flag(byte[] flags, int index, int setValue, int maxSize)
```

Your function may assume that the given flag array is actually `maxSize` bits in length.

NOTE: `setValue` could be any integer value!

Your function may assume that maxSize is between 1 and 255 (inclusive).

When adding a node to a binary search tree, we must take care that the node is inserted in such a way that the binary search tree property is maintained, namely, that a given node's value is greater than the values in its left subtree and less than or equal to the values in its right subtree. Implement the following function, `find_position`, which determines where a node should be inserted into a BST so as to maintain the search property.

Note that this function does not actually add the node to the tree. Insertion will be implemented in a later function.

```
/**
* This function traverses the binary search tree and finds the position
* at which to add the new value. The position will be expressed using both
* the index of the new node's parent node, and a 0 or 1 to indicate
* whether the new node should be the left child (0) or right child (1) of
* its parent.
*
* @param nodes, Base address of the nodes array.
* @param currIndex, The index of the current node.
* @param newValue, Value to be added to the BST.
* @return The index of what would be the parent node of the new node.
* @return 0 If the new node will be its parent's left child, 1 if it will be
*           its parent's right child.
*/
public static int,int find_position(Node[] nodes, int currIndex,
                                   int newValue)

    // Discard most significant 16 bits of newValue (see note)
    newValue = toSignedHalfWord(newValue);

    if (newValue < nodes[currIndex].value ) {
        int leftIndex = nodes[currIndex].left;
        if (leftIndex == 255) {
            return currIndex, 0; // No left child, add here
        } else {
            return find_position(nodes, leftIndex, newValue);
        }
    }
    else {
        int rightIndex = nodes[currIndex].right;
        if (rightIndex == 255) {
            return currIndex, 1; // No right child, add here
        } else {
            return find_position(nodes, rightIndex, newValue);
        }
    }
}
```

⚠ For our first call to your function `find_position`, it is **NOT** guaranteed that `newValue` will be a 16-bit two's complement integer. You must **convert** the original 32-bit `newValue` to a 16-bit two's complement number before using the value to navigate through a binary search tree (as indicated by `toSignedHalfWord`, but `toSignedHalfWord` is not actually a function). Do **NOT** return from the function if the original 32-bit value is out of the range of a 16-bit two's complement number.

ℹ You may assume `nodes` is valid and holds at least one BST node.

⚠ You may assume that we will provide a valid `currIndex in nodes`. For example, if we provided a valid nodes array of size 100, then it is guaranteed that we will provide a `currIndex` between 0 and 99 (inclusive) of a valid starting node in the BST stored there. When you use `find_position` later in the assignment, however, you must incorporate the function in a way that guarantees that `nodes` and `currIndex` are valid.

⚠ Since `find_position` is a recursive function, `find_position` **MUST** `jal find_position`.

Using the `find_position` function, write the function `add_node` to insert a node into the BST.

```
/**
* This function adds a node to a binary search tree with the specified
* two’s complement value.
*
* @param nodes, Base address of the nodes array.
* @param rootIndex, The index position of the root node.
* @param newValue, Value to be added to the BST.
* @param newIndex, The index of the nodes array to add the new BST node.
* @param flags, The base address of the flag bit-array.
* @return 1 if success, or 0 if failure.
*/
public static int add_node(Node[] nodes, int rootIndex, int newValue,
                           int newIndex, byte[] flags, int maxSize) {

    // Discard most significant 24 bits (see note)
    rootIndex = toUnsignedByte(rootIndex);
    newIndex =  toUnsignedByte(newIndex);

    if (rootIndex >= maxSize || newIndex >= maxSize)
        return 0;

    // Discard most significant 16 bits of newValue (see find_position)
    newValue = toSignedHalfWord(newValue);

    // Determine if a root node actually exists at rootIndex
    boolean validRoot = nodeExists(rootIndex);

    if (validRoot) { // if a valid root node already exists
        // Find a valid position in the BST with newValue as the comparison
        // value.
        int parentIndex, leftOrRight = find_position(nodes, rootIndex,
                                                    newValue);

        if (leftOrRight == left) {
            // update parent's Left Node index
            nodes[parentIndex].left = newIndex;
        } else {
            // update parent's Right Node index
            nodes[parentIndex].right = newIndex;
        }
    } else { // we must add newValue as a root node instead
        newIndex = rootIndex;
    }

    nodes[newIndex].left = 255;
    nodes[newIndex].right = 255; // two 255's create a leaf node
    nodes[newIndex] = newValue;

    return set_flag(flags, newIndex, 1, maxSize);
}
```

ℹ It is suggested that you write your own helper function, `nodeExists`, to determine whether an index in `nodes` is a valid BST node. You already have a data structure that will make writing this function very straightforward.

⚠ `add_node` **MUST** call functions `find_position` and `set_flag`.

⚠ You may assume `nodes` is valid, but it is **NOT** guaranteed to currently be holding any BST nodes.

ℹ Your function may assume that the given flag array is actually `maxSize` bits and the given nodes array is actually `maxSize` words in length.

ℹ Your function may assume that maxSize is between 1 and 255 (inclusive).

ℹ Your function may assume that the provided `flags` corresponds to the provided `nodes`, meaning that the flag array will only contain 1's at indices where the nodes array actually has valid BST nodes.

⚠ Your function may assume that the provided `newIndex` will be a free position if `add_node` is tested alone. If `add_node` is called in conjunction with `linear_search` to find a free position to add to, then only a correct linear search function will guarantee that you do not overwrite the BST currently in `nodes`.

⚠ It is **NOT** guaranteed that both `rootIndex` and `newIndex` will be 8-bit unsigned integers. You must **convert** the original 32-bit `rootIndex` and 32-bit `newIndex` to 8-bit unsigned numbers before use (as indicated by `toUnsignedByte`, but `toUnsignedByte` is not actually a function). Do **NOT** return from the function if the original 32-bit values are out of the range of an 8-bit unsigned number.

⚠ It is **NOT** guaranteed that `newValue` will be a 16-bit two's complement integer. You must **convert** the original 32-bit `newValue` to a 16-bit two's complement number before use (as indicated by `toSignedHalfWord`, but `toSignedHalfWord` is not actually a function). Do **NOT** return from the function if the original 32-bit value is out of the range of a 16-bit two's complement number.

⚠ Since you only have 4 argument registers, and this function uses 6 arguments, you **MUST** place the last two arguments onto the call stack before calling `add_node`. By convention, place `flags` onto the stack first, and then `maxSize` on the top of the stack.

At the end of the assignment is an Extra Credit function, `add_random_nodes`, to help test your Part 3 functions together.

Part 3 - Deleting Nodes from Binary Search Trees

In this final part of the assignment you will implement a function to delete nodes from a binary search tree. This can be a little tricky to implement because there are special cases to consider, but fear not because the provided pseudocode will handle all possible cases. The figure below shows some of the cases your code will have to handle. The figure shows examples of three deletions. Each deletion is independent of the others and is performed on the initial tree depicted on the left: (i) deleting a leaf (node #3), (ii) deleting a node with one child (node #6), and (iii) deleting a node with two children (node #4).

```
   2                 2                 2                 2
 /   \             /   \             /   \             /   \
1     7           1     7           1     7           1     7
     / \               / \               / \               / \
    4   8             4   8             4   8             5   8
   / \                 \               / \               / \
  3   6                 6             3   5             3   6
     /                 /
    5                 5

Initial tree     Deleted leaf     Deleted node w/    Deleted node w/
                 node, #3         1 child, #6        2 children, #4
```

To make the various cases easier to deal with, you will implement a few helper functions to minimize the complexity of the `delete_node` function. The first helper function, `get_parent`, performs a recursive search of a BST to find the parent node of a given child node. Since a particular value could appear multiple times in a BST, two values are provided to correctly identify the target child: the child's value and the child's index. The child's value is needed because this is the value that we use to descend and search through the tree. The child's index is provided so that, when we encounter a node with the target value, we can verify that we have found the particular node we are actually searching for.

```
/**
* This function traverses the binary search tree to find the parent
* node of a specified (child) node. The child node will be specified
* by its index in the nodes array and the value at that child node.
* The index of the child is important because
* repeated values in the binary search tree will have different
* indices in the nodes array. This function should be able to
* distinguish between nodes having the same value because they
* have different indices. For convenience, both the child's value
* the child's index are given as parameters. However, should the
* child's index and child's value not correspond (i.e., the node
* at index childIndex does NOT contain the value childValue),
* then the behavior of the function is undefined.
*
* @param nodes, Base address of the nodes array.
* @param currIndex, The index of the current node.
* @param childValue, Value of node childIndex used to navigate the
*                   the binary search tree to find the child node.
* @param childIndex, The index of the child node for which we are
*                   trying to find a parent.
* @Precondition The current node is valid.
* @Precondition childValue and childIndex correspond to the same node.
* @return the parent node index of the specified child node if found,
*         or -1 if the child node is not in the tree or the child
*         node is actually the root from the first call.
* @return 0 if the child node is left of the parent, 1
*         if right, or "don't care" if a parent node was not found
*/
public static int,int get_parent(Node[] nodes, int currIndex,
                                    int childValue, int childIndex)

    // Refer to previous functions for details of these 2 "functions"
    childIndex = toUnsignedByte(childIndex);
    childValue = toSignedHalfWord(childValue);

    if (childValue < nodes[currIndex].value) {
        int leftIndex = nodes[currIndex].left;
        if (leftIndex == 255) {
            return -1;
        } else if (leftIndex == childIndex) {
            return currIndex, 0;
        } else {
            return get_parent(nodes, leftIndex, childValue, childIndex);
        }
    }
    else {
        int rightIndex = nodes[currIndex].right;
        if (rightIndex == 255) {
            return -1;
        } else if (rightIndex == childIndex) {
            return currIndex, 1;
        } else {
            return get_parent(nodes, rightIndex, childValue, childIndex);
        }
    }
}
```

ℹ It is guaranteed that `nodes` is valid and will have at least one BST node. We will provide `currIndex` with a valid BST node index in `nodes`.

⚠ It is guaranteed that the `childValue` and `childIndex` we provide will correspond to the same child node if `childIndex` is actually in the tree. However, you must make this guarantee yourself when using the function later to write `delete_node`.

⚠ It is **NOT** guaranteed that `childIndex` will be an 8-bit unsigned integer. You must **convert** the original 32-bit `childIndex` to an 8-bit unsigned number before use (as indicated by toUnsignedByte, but toUnsignedbyte is not actually a function). Do **NOT** return from the function if the original 32-bit value is out of the range of an 8-bit unsigned number.

⚠ It is **NOT** guaranteed that `childValue` will be a 16-bit two's complement integer. You must **convert** the original 32-bit `childValue` to a 16-bit two's complement number before use (as indicated by `toSignedHalfWord`, but `toSignedHalfWord` is not actually a function). Do **NOT** return from the function if the original 32-bit value is out of the range of a 16-bit two's complement number.

⚠ Since `get_parent` is a recursive function, `get_parent` **MUST** `jal get_parent`.

The next helper function, `find_min`, recursively searches a BST (or subtree of a BST) for the minimum value stored in the tree or subtree. The function will be called initially on the root of the tree/subtree in question.

```
/**
* This function will find the first minimum valued node in a binary
* search tree. Note: isLeaf is not a function, but rather a boolean
* condition.
*
* @param nodes, Base address of the nodes array.
* @param currIndex, The index of the current node.
* @Precondition The current node is valid.
* @return the index of the first minimum valued node in the tree
*         rooted by the current node.
* @return 1 if the first minimum node in the tree rooted by the
*         current node is a leaf, else 0
*/
public static int,int find_min(Node[] nodes, int currIndex) {

    int leftIndex = nodes[currIndex].left;
    if (leftIndex == 255) {
        return currIndex, isLeaf(nodes[currIndex]);
    } else {
        return find_min(nodes, leftIndex);
    }
}
```

It is guaranteed that `nodes` is valid and will have at least one BST node. We will provide `currIndex` with a valid BST node index in `nodes`.

Note: `isLeaf` is not a function, but rather a boolean condition.

Since `find_min` is a recursive function, `find_min` **MUST** `jal find_min`.

With these helper functions complete, you can now implement the function `delete_node`, which will delete a node from the BST. After deleting the node, the function must reorganize the nodes to make sure the tree stays connected while also maintaining the search property of the BST. As stated earlier for `add_node`, it is suggested that you write your own helper function, `nodeExists`, to determine whether an index in `nodes` is a valid BST node.

```
/*
* This function deletes a node specified in the BST.
*
```

```
* @param nodes, Base address of the nodes array.
* @param rootIndex, The index position of the root node.
* @param deleteIndex, The index of the BST node to delete.
* @param flags, The base address of the flag bit-array.
* @param maxSize, The size of nodes in words and size of flags in
*                valid (not ignored) bits
* @return 1 if successful or 0 if a deletion could not be performed since
*         either rootIndex or deleteIndex are not in the range [0,maxSize-1],
*         or rootIndex or deleteIndex are not in the BST.
*/
public static int delete_node(Node[] nodes, int rootIndex, int deleteIndex,
                              byte[] flags, int maxSize) {

    // Refer to previous functions for details of these 2 "functions"
    rootIndex = toUnsignedByte(rootIndex);
    deleteIndex = toUnsignedByte(deleteIndex);

    if (rootIndex >= maxSize || delete >= maxSize)
        return 0;

    // Determine if a root node actually exists at rootIndex and a node
    // actually exists at deleteIndex.
    boolean validRoot = nodeExists(rootIndex);
    boolean validDel = nodeExists(deleteIndex);

    if (!validRoot || !validDel)
        return 0;

    if (isLeaf(nodes[deleteIndex])) {
        set_flag(flags, deleteIndex, 0, maxSize); // remove flag from flags

        if (deleteIndex == rootIndex)
            return 1; // root flag already deleted

        // Get the parent of the node to delete
        int parentIndex, leftOrRight = get_parent(nodes, rootIndex,
                                       nodes[deleteIndex].value, deleteIndex);

        if (leftOrRight == left) //delete left reference from parent node
            nodes[parentIndex].left = 255;
        else // Delete right reference from parent node
```

```
            nodes[parentIndex].right = 255;

        return 1;

    } else if (hasOneChild(nodes[deleteIndex])) {

        // The node to delete has one child subtree
        Node childNode;
        int childIndex;

        if (hasALeftChild(nodes[deleteIndex]))
            childIndex = nodes[deleteIndex].left;
        else // Node has a right child only.
            childIndex = nodes[deleteIndex].right;

        if (deleteIndex == rootIndex) {
            childNode = nodes[childIndex];

            // Overwrite deleteNode’s entire contents with childNode’s
            // entire contents
            nodes[deleteIndex] = childNode;

            // Remove the flag from the child node’s old position
            // since the child node is now the root
            set_flag(flags, childIndex, 0, maxSize);

            return 1;
        }

        // Get the parent of the node to delete
        int parentIndex, leftOrRight = get_parent(nodes, rootIndex,
                                       nodes[deleteIndex].value, deleteIndex);

        // Change parent’s left reference to found child index
        if (leftOrRight == left)
            nodes[parentIndex].left = childIndex;
        else // Change parent’s right reference to found child index
            nodes[parentIndex].right = childIndex;

        // Remove the flag from the deleted node’s position
        set_flag(flags, deleteIndex, 0, maxSize);
        return 1;
    } else {
        // The node to delete has two subtrees: left and right.
        // Find the first instance of the minimum-valued node in deleteNode’s
        // right subtree. This node has the smallest value greater than or
        // equal to the node to delete. We will use this minimum node to to
        // replace the two’s complement value at the node to delete.  In that
        // way,everything in the right subtree will still be greater than or
        // equal to the new parent of that subtree.
        // The min return values will be valid since find_min is called on a
        // valid node.

        int minIndex, minIsLeaf = find_min(nodes, nodes[deleteIndex].right);

        // Get the parent of the minimum node of the right subtree
        int parentIndex, leftOrRight = get_parent(nodes, deleteIndex,
                                       nodes[minIndex].value, minIndex);

        if (minIsLeaf == true) { // The minimum node is a leaf
            if (leftOrRight == left) // Min is left of its parent
                nodes[parentIndex].left = 255;
            else // Min is right of it’s parent
                nodes[parentIndex].right = 255;

        } else { // minimum node has a right subtree
            if (leftOrRight == left)
                // Change parent’s left reference to min’s right child
                nodes[parentIndex].left = nodes[minIndex].right;
            else
                // Change parent’s right reference to min’s right child
                nodes[parentIndex].right = nodes[minIndex].right;
        }

        // Replace deleteNode’s value with minimum from right subtree
        nodes[deleteIndex].value = nodes[minIndex].value;
        // Remove the flag from the min node’s position
        set_flag(flags, minIndex, 0, maxSize);
        return 1;
    }
}
```

⚠ `delete_node` MUST call functions `get_parent` and `find_min`.

ℹ The `isLeaf`, `hasOneChild`, and `hasALeftChild` statements are not actually functions, but rather, boolean conditions.

⚠ You may assume `nodes` is valid, but it is **NOT** guaranteed to currently be holding any BST nodes.

ℹ Your function may assume that the given flag array is actually `maxSize` bits and the given nodes array is actually `maxSize` words in length.

ℹ Your function may assume that `maxSize` is between 1 and 255 (inclusive).

ℹ Your function may assume that the provided `flags` corresponds to the provided `nodes`, meaning that the flag array will only contain 1's at indices where the nodes array actually has valid BST nodes.

ℹ When a node is deleted, do **NOT** clear that element in the `nodes` array. Only implement the algorithm as it is written.

⚠ It is **NOT** guaranteed that both `rootIndex` and `deleteIndex` will be 8-bit unsigned integers. You must **convert** the original 32-bit `rootIndex` and 32-bit `deleteIndex` to 8-bit unsigned numbers before use (as indicated by `toUnsignedByte`, but `toUnsignedbyte` is not actually a function). Do **NOT** return from the function if the original 32-bit values are out of the range of an 8-bit unsigned number.

⚠ Since you only have 4 argument registers, and this function uses 5 arguments, you **MUST** place the last argument, `maxSize`, onto the top of the call stack before calling `delete_node`.

Extra Credit

This function, `add_random_nodes`, uses all the functions in Part 2 together. In this example testing code, we will be testing the capacity of the `nodes` specified, the linearity of your search for an available position to add a node, and creating a pseudo-randomized BST via adding values from a pseudo-random number generator. This pseudo-randomization will allow for, in many instances, the creation of a tree that will not just be a linked-list and allow for you to predict the numbers going into the BST. Therefore, we will bring back our exercise from Homework #1 in creating pseudo-random numbers with syscalls 40 and 42. We have included an abbreviated version of these syscalls for reference.

description	syscall	arguments	result
set seed	40	`$a0` = i.d. of pseudorandom number generator (any int). `$a1` = seed for corresponding pseudorandom number generator.	None
random int range	42	`$a0` = i.d. of pseudorandom number generator (any int). `$a1` = upper bound of range of returned values.	`$a0` next value

Always set the ID of the RNG to **0**, and set the seed of the RNG with the exact value provided by the function argument. Create a while loop to continue drawing numbers from the range [-32768, 32767], the same range as a signed 16-bit number. Details of the syscalls for drawing pseudo-random numbers can be found in greater detail in the MARS documentation, and we suggest you read this documentation to figure out how to get numbers in this range. Draw these pseudo-random numbers and add them to the BST until there is no more room in the nodes array.

```
/**
* This function is a basic test for adding nodes with a linear search.
* As you will see, the nodes should be added linearly to the nodes array,
* but since the node values are pseudo-random, a randomized BST will be
* formed. This is only designed as a preliminary test, and you should
* implement many other tests and test cases for adding nodes to a BST in
* the nodes array.
*
* @param nodes Base address of the nodes array.
* @param maxSize The size of nodes in words and flags in valid
*               (not ignored) bits.
* @param rootIndex The index position of the root node.
* @param flags The base address of the flag bit-array.
* @param seed The seed to the random number generator.
* @param fd File descriptor of a previously opened file.
*/
public static void add_random_nodes(Node[] nodes, int maxSize,
    int rootIndex, byte[] flags, int seed, int fd) {

    if (rootIndex < 0 || rootIndex >= maxSize)
        return;

    // Create the RNG with ID 0 and the given seed
    Random generator = new Random(0, seed);

    int newIndex = linear_search(flags, maxSize);

    int newValue, parentIndex, leftOrRight;
    while (newIndex != -1) {
        newValue = generator.nextInt(-32768,32767); //inclusive range
        parentIndex,leftOrRight = find_position(nodes, rootIndex, newValue);

        // Write the following EXACTLY to file
        write(fd, "New value: ", 11);
        itof(fd, newValue)
        write(fd, "\n", 1);
        write(fd, "Parent index: ", 14);
        itof(fd, parentIndex);
        write(fd, "\n", 1);
        write(fd, "Left (0) or right (1): ", 23);
        itof(fd, leftOrRight);
        write(fd, "\n\n", 2);

        // Add node to next available position with value
        add_node(nodes, rootIndex, newValue, newIndex, flags, maxSize);
        newIndex = linear_search(flags, maxSize);
    }

     // Run pre_order from the rootNode with the same file descriptor above.
     // Do not add anything to file in between the last written output above
     // and the call to preorder.
     preorder(address of nodes[rootIndex], nodes, fd);
}
```

⚠ For this function to work, it must be guaranteed that `rootIndex` refers to valid root. This means that `nodes` will contain at least 1 valid BST node at the start.

⚠ It is guaranteed that `nodes`, `maxSize`, and `flags`, are all valid (in the ways described for previous functions). Do not change the seed value in any way.

ℹ Notice that `newValue` should not ever change value because it is always in the range of a signed half-word. We suggest that you write another test where `newValue` will be truncated to become a signed half-word.

⚠ Since you only have 4 argument registers, and this function uses 6 arguments, you **MUST** place the last two arguments onto the call stack before calling `add_random_nodes`. For consistency, place `seed` onto the stack first, and then `fd` on the top of the stack.

⚠ Your function may assume that the file descriptor is valid, meaning that the file is already open and and writable (for output). Your implementation of the function must **NOT** close the file before returning. Rather, your main program **MUST** close the file prior to exiting main.

⚠ The write function above is a function available in the C programming language. The first argument is the file descriptor. The second argument is the string to write to the file. The last argument is the number of bytes to write. You will implement this using syscall 15 in MARS.

Final Remarks and Guidance

The largest node array possible is 255 words, and a node array represents a space in memory that is allocated to hold binary tree nodes. Note that if this node array is 255 words long, the array can hold anywhere from 0 to 255 valid nodes of a binary tree. It is your job to create a tree in the memory space (node array) you allocate. If the tree stored in your node array is smaller than the length of the nodes array, the unused word segments are junk data. Given a valid root node in a nodes array, your `preorder` function should be able to print the tree stored there to file without navigating to any junk word segments and without printing any junk values. Do not assume that the node array will be zeroed out. We will test your code with arrays that have randomized junk word segments and arrays that have allocated less than 255 words.

If an array is less than 255 words and we give you a tree pre-stored there, you are guaranteed that the tree nodes will not contain references to indices that go outside of bounds of the array. If you are adding your own tree to a node array whose memory space is smaller than 255 words and are using that array in conjunction with `linear_search`, then `linear_search` should be given the correct maximum size of your node array and its corresponding flag bit array so no nodes are added out of bounds.

Your functions for adding and deleting nodes may only change values in the provided `nodes` array argument at the positions specified in the assignment. You may want to write a macro or helper function in your main that will print a `nodes` array in hexadecimal to standard output (i.e., the screen). In this capacity, you can compare the original node array that you defined in your main's data section word-by-word to that same node array after performing add and delete operations on it. Some sample test data is given in the main files provided to show what should be stored in the node array after the provided test cases are run.

Good luck!

Hand-in Instructions

See Sparky Submission Instructions on Piazza for hand-in instructions.

❗ There is no tolerance for homework submission via email. Work must be submitted through Sparky. Please do not wait until the last minute to submit your homework. If you are struggling, stop by office hours for additional help.