```
tree_t *bst_create() {
	tree_t* t = (tree_t*)malloc(sizeof(tree_t));
	if (t == NULL) {
	return NULL;
	}
	t->size = 0;
	t->source = NULL;
	return t;
}

treenode_t* create_node(int data) {
	treenode_t* newNode =
(treenode_t*)malloc(sizeof(treenode_t));
	if (newNode == NULL) {
	return NULL;
	}
	newNode->data = data;
	newNode->leftChild = NULL;
	newNode->rightChild = NULL;
	return newNode;
}

// BST Empty
// Check if the BST is empty
// Returns 1 if true (The BST is completely empty)
// Returns 0 if false (the BST has at least one element)
int bst_empty(tree_t *t) {
	if (t == NULL) {
	return 1;
	}
	return t->size == 0;
```

```
}

treenode_t* insertHelper(treenode_t* root, int val) {
        if (root == NULL) {
        root = create_node(val);
        return root;
        }
        if (val <= root->data) {
        root->leftChild = insertHelper(root->leftChild, val);
        } else {
        root->rightChild = insertHelper(root->rightChild, val);
        }
        return root;
}
// Adds a new node containing item to the BST
// The item is added in the correct position in the BST.
//  - If the data is less than or equal to the current node we
traverse left
//  - otherwise we traverse right.
// The bst_function returns '1' upon success
//  - bst_add should increment the 'size' of our BST.
// Returns a -1 if the operation fails.
//      (i.e. the memory allocation for a new node failed).
// Your implementation should should run in O(log(n)) time.
//  - A recursive implementation is suggested.
int bst_add(tree_t *t, int item) {
        if (t == NULL) {
        return -1;
        }
        if (t->size == 0) {
        t->source = create_node(item);
```

```
        if (t->source == NULL) {
        return -1;
        }
        t->size ++;
        return 1;
        }
        treenode_t* res = insertHelper(t->source, item);
        if (res == NULL) {
        return -1;
        }
        t->size ++;
        return 1;
}

void dfsAsc(treenode_t* root) {
        if (root == NULL) {
        return;
        }
        dfsAsc(root->leftChild);
        printf("%d\n", root->data);
        dfsAsc(root->rightChild);
}
void dfsDesc(treenode_t* root) {
        if (root == NULL) {
        return;
        }
        dfsDesc(root->rightChild);
        printf("%d\n", root->data);
        dfsDesc(root->leftChild);
}
```

```
// Prints the tree in ascending order if order = 0, otherwise prints
in descending order.
// For NULL tree (i.e., when t == NULL) -- print "(NULL)".
// It should run in O(n) time.
void bst_print(tree_t *t, int order) {
        if (t == NULL) {
        printf("NULL");
        return;
        }
        if (order == 0) {
        dfsAsc(t->source);
        } else {
        dfsDesc(t->source);
        }
}

int dfsSum(treenode_t* root, int curSum) {
        if (root == NULL) {
        return curSum;
        }
        curSum = dfsSum(root->leftChild, curSum);
        curSum += root->data;
        curSum = dfsSum(root->rightChild, curSum);
        return curSum;
}
// Returns the sum of all the nodes in the bst.
// exits the program for a NULL tree.
// It should run in O(n) time.
int bst_sum(tree_t *t) {
        if (t == NULL) {
        exit(1);
```

```
        }
        return dfsSum(t->source, 0);
}

treenode_t* finder(treenode_t* root, int val) {
        if (root == NULL) {
        return NULL;
        }
        if (val < root->data) {
        return finder(root->leftChild, val);
        } else if(val > root->data) {
        return finder(root->rightChild, val);
        } else {
        return root;
        }
}
// Returns 1 if value is found in the tree, 0 otherwise.
// For NULL tree it exists the program.
// It should run in O(log(n)) time.
int bst_find(tree_t *t, int value) {
        if (t == NULL) {
        exit(1);
        }
        treenode_t* res = finder(t->source, value);
        if (res == NULL) {
        return 0;
        }
        return 1;
}

// Returns the size of the BST
```

```
// A BST that is NULL exits the program.
// (i.e. A NULL BST cannot return the size)
unsigned int bst_size(tree_t *t) {
	if (t == NULL) {
	exit(-1);
	}
	return t->size;
}


void freeHelper(treenode_t* root) {
	if (root == NULL) {
	return;
	}
	freeHelper(root->leftChild);
	freeHelper(root->rightChild);
	free(root);
}
// Free BST
// Removes a BST and ALL of its elements from memory.
// This should be called before the program terminates.
void bst_free(tree_t *t) {
	if (t == NULL) {
	exit(1);
	}
	freeHelper(t->source);
	free(t);
}
```

this is my implementation is it right / cool? how can I improve?

This is a C program that implements a binary search tree (BST) data structure. The BST is a tree structure that has the following properties:

- All nodes in the left subtree of a node have values less than or equal to the node's value.
- All nodes in the right subtree of a node have values greater than the node's value.
- Each subtree is itself a BST.

The program defines two structures: `treenode_t` and `tree_t`. `treenode_t` represents a single node in the tree and has three fields:

- `data` is an integer that represents the value stored in the node.
- `leftChild` is a pointer to the node's left child.
- `rightChild` is a pointer to the node's right child.

`tree_t` represents the entire tree and has two fields:

- `size` is an unsigned integer that represents the number of nodes in the tree.
- `source` is a pointer to the root node of the tree.

The program defines several functions to manipulate the BST:

- `bst_create` creates a new, empty tree and returns a pointer to it.
- `create_node` creates a new node with the specified value and returns a pointer to it.

- `bst_empty` checks if the tree is empty and returns 1 if it is, 0 otherwise.
- `insertHelper` recursively inserts a new node with the specified value into the tree and returns a pointer to the new root of the tree.
- `bst_add` inserts a new node with the specified value into the tree and returns 1 upon success and -1 if the operation fails.
- `dfsAsc` recursively traverses the tree in ascending order and prints each node's value.
- `dfsDesc` recursively traverses the tree in descending order and prints each node's value.
- `bst_print` prints the tree in either ascending or descending order depending on the value of the `order` parameter.
- `dfsSum` recursively calculates the sum of all nodes in the tree.
- `bst_sum` returns the sum of all nodes in the tree.
- `finder` recursively searches the tree for a node with the specified value and returns a pointer to it if found.
- `bst_find` searches the tree for a node with the specified value and returns 1 if found and 0 otherwise.
- `bst_size` returns the size of the tree.
- `freeHelper` recursively frees all memory allocated to the tree.