By signing this statement, I hereby certify that the work on this exam is my own and that I have not copied the work of any other student while completing it. I also declare that I will not discuss/share this exam with anybody today. I understand that, if I fail to honor this agreement, I will receive a score of ZERO for this exam and will be subject to possible disciplinary action.

**Signature:**

*You must sign here. Otherwise you will receive a 2-point penalty.*

---

Read the questions carefully.

*You must return all pages before you leave the room. Otherwise, you will receive a 2-point penalty.*

This is an *open-book, open-note* exam. You may use any book, notes, or program printouts. No personal electronic device is allowed. You may not borrow books from other students.

Three learning objectives (recursion, structure, and dynamic structure) are tested in this exam. To pass an objective, you must receive 50% or more points in the corresponding question.

**Questions (Total 15 points)**

1. recursion and binary tree (7 points, learning objectives 1 and 4)
2. structure and binary tree (7 points, learning objectives 1, 3 and 4)
3. comparison function (1 point)
Contents

1 Recursion (7 points) 3
   1.1 Count Shapes (3 points) ................................................... 3
   1.2 Compare Shapes (4 points) ................................................... 4

2 Sparse Arrays (7 points) 7

3 Comparison Function (1 point) 17

Learning Objective 1 (Recursion) Pass Fail
Learning Objective 3 (Structure) Pass Fail
Learning Objective 4 (Dynamic Structure) Pass Fail

Total
1 Recursion (7 points)

Two binary trees have the same shape if the roots have

• the same number of nodes on the left side, and
• the same number of nodes on the right side.

This rule is then applied recursively for the left and the right side of the two trees. The following figure shows different shapes of trees with two and three nodes.

![Figure 1: (a) Two different shapes of binary trees with 2 nodes. (b) Five different shapes of binary trees with 3 nodes.](image)

1.1 Count Shapes (3 points)

For a binary tree with \( n \) nodes \((n > 2)\), how many different shapes are possible? Please write down a recursive formula. Briefly explain your answer.

**Do Not** write a closed form. You will receive no point if you write a closed form with any explanation.

Hint: There are \( k \) nodes on the left side and \( n - k - 1 \) nodes on the right side, here \( k \) can be 0, 1, 2, ..., \( n - 1 \).

The following table is for your reference.

<table>
<thead>
<tr>
<th>( n )</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
</tr>
</thead>
<tbody>
<tr>
<td>number of shapes</td>
<td>1</td>
<td>2</td>
<td>5</td>
<td>14</td>
<td>42</td>
<td>132</td>
<td>429</td>
<td>1430</td>
<td>4862</td>
<td>16796</td>
</tr>
</tbody>
</table>
1.2 Compare Shapes (4 points)

Write a recursive function that compares two trees and returns the constant identifier SAME if they have the same shape, and DIFFERENT if they do not.

```c
#define SAME 1
#define DIFFERENT 0

typedef struct treenode {
    int value;
    struct treenode *left;
    struct treenode *right;
} Tree;

/* Compare two trees to see if they have the same shape.
 * return SAME if the two trees have the same shape.
 * return DIFFERENT if the two tree have different shapes.
 * Do not worry about the values stored in the nodes.
 * Assume the two trees do not share any space on heap memory.
 */
int compareTree(Tree *tree1, Tree *tree2)
{
    // FILL IN CODE HERE
    // if both tree1 and tree2 are NULL, they are the same

    // if tree1 is NULL and tree2 is not NULL, they are different

    // if tree1 is not NULL and tree2 is NULL, they are different
```
// if the left sides of tree1 and tree2 are different, they are
different

// finally, compare the right sides of tree1 and tree2

2 Sparse Arrays (7 points)

Consider a data structure called a sparse array: most elements of a sparse array are 0. This means that most elements do not need to be stored in the array. Only the elements whose values are non-zero are stored. In a C program, a normal array of size $N$ occupies a contiguous piece of memory allocated for indexes from 0 to $N-1$. It would be inefficient to store a sparse array in a normal C array because most space is wasted. Instead, you can use a data structure that stores only non-zero elements.

In this question, you will implement a sparse array using a binary search tree. Each tree node has two integers storing the index and the value. The binary search tree is built based on the indexes:

- If a node's index is smaller than the parent's index, this node is on the left side.
- If a node's index is larger than the parent's index, this node is on the right side.
- The indexes in the whole tree are distinct: no two nodes have the same index.

Fill in the missing code in the listing below to complete the sparse array. The following program has two parts. The first part asks you to fill in code. The second part has some functions that are already implemented for you. You can use those functions.

```
#include <stdio.h>
#include <stdlib.h>

#define NUM_VALUES 6

typedef struct snode {
    int index;  /* Array index (used in binary search tree) */
    int value;  /* Integer value for the index */
    struct snode *left; /* Left subtree (smaller indices) */
    struct snode *right; /* Right subtree (larger indices) */
} SparseNode;

/* This function is already implemented for you. */
SparseNode *SparseNode_create(int index, int value);
/* ----------------------------------------------------------- */

/* Set a particular value into a sparse array on a particular index. */
/* The sparse array uses the index as a key in a binary search tree. */
/* This is a recursive function equivalent to inserting into a binary */
/* search tree. */
/* The function returns the root of the new binary tree. */
```
* If the value is zero, don’t do anything and return the original tree.
* If the index does not exist, create a new node and insert this node into the tree.
* If the index already exists, update the value to the new value.
* Use the index to determine whether to go left or right in the tree (smaller index values than the current one go left, larger ones go right).

```c
SparseNode *SparseArray_set(SparseNode *array, int index, int value) {
    // FILL IN CODE HERE!!! (2 points)
}
```

/* Retrieve the value associated with a specific index in a sparse array. This is a recursive function equivalent to searching in a binary tree. It returns the value associated with the index.
* If the index does not exist in the array, this function returns 0.
* If the given index is smaller than the current node, search left;
* if the given index is larger, search right. */
int SparseArray_get(SparseNode *array, int index)
{
    // FILL IN CODE HERE!!! (2 points)
}

/* Remove a value associated with a particular index from the sparse array. It returns the root of the new sparse array (binary tree root).
HINT: This is a recursive function equivalent to deleting a node from a binary search tree. You will need to isolate several different cases here:
1. If the array is empty (NULL), return NULL.
2. Go left or right if the current node index is different.
3. If the index is found and this node has no child, remove this node.
4. If this node has only child, remove the current node and replace it by the child.
5. If both children exist, you must find the successor of the current node (leftmost of right branch), swap its values with the current node (BOTH index and value), and then delete the index in the right subtree.
*/
SparseNode *SparseArray_remove(SparseNode *array, int index)
{
    // FILL IN CODE HERE!!! (3 points)
/* Create a single instance of a sparse array node with a specific index and value. This is a constructor function that allocates memory, copies the integer values, and sets the subtree pointers to NULL. */

SparseNode * SparseNode_create ( int index, int value ) {
    SparseNode * node = malloc ( sizeof ( SparseNode ));
    node -> index = index ;
    node -> value = value ;
    node -> left = NULL ;
    node -> right = NULL ;
    return node ;
}

/* Destroy an entire sparse array. This is a recursive function */
traversing the binary tree in postorder. Use the
SparseNode_destroy() function to destroy each node by itself.

```
void SparseArray_destroy(SparseNode *array)
{
    if (array == NULL) return;
    if (array->left) SparseArray_destroy(array->left);
    if (array->right) SparseArray_destroy(array->right);
    free(array);
}
```

/* Retrieve the smallest index in the sparse array. This is NOT a
recursive function. */

```
int SparseArray_getMin(SparseNode *array)
{
    if (array == NULL) return 0;
    while (array->left != NULL) array = array->left;
    return array->index;
}
```

/* Retrieve the largest index in the sparse array. This is NOT a
recursive function. */

```
int SparseArray_getMax(SparseNode *array)
{
    if (array == NULL) return 0;
    while (array->right != NULL) array = array->right;
    return array->index;
}
```

```
int main(int argc, char **argv)
{
    int i;
    int indices[NUM_VALUES] = { 0, 5, -3, 9, 7, 6 };
    int values[NUM_VALUES] = { 12, 34, 9, 49, 40, 37 };
    SparseNode *array = NULL;
    for (i = 0; i < NUM_VALUES; i++) {
        array = SparseArray_set(array, indices[i], values[i]);
    }
    for (i = SparseArray_getMin(array); i < SparseArray_getMax(array);
        i++) {
```
printf("%d:%d ", i, SparseArray_get(array, i));
}
printf("\n");
SparseArray_destroy(array);
return EXIT_SUCCESS;
}

Here is the output from this code:

$ ./q2
-3:9 -2:0 -1:0 0:12 1:0 2:0 3:0 4:0 5:34 6:37 7:40 8:0
3 Comparison Function (1 point)

In this question, you will use a comparison function for the C standard library implementation of the Quicksort algorithm (called qsort()). Quicksort is a very efficient sorting algorithm. The UNIX manual page for the qsort() implementation is given at the end of this question.

As you can see from the manual page, qsort() sorts an array of data (pointed to by the void pointer base). It also expects a function pointer called compar to use as a comparison function. The reason for this is that qsort() has been designed to be as generic as possible so that it can handle any data type. By making the comparison function an argument, the programmer who is using qsort() can decide how to compare two elements and determine whether the first element is less than (return negative value), equal to (return 0), or greater than (return positive value) the second argument passed to the comparison function.

Your task is to fill in the missing code for a comparison function called compInt() that takes two integer values and returns -1 if the first argument is smaller than the second, 0 if they are equal, and +1 if the first argument is larger than the second.

```c
#include <stdio.h>
#include <stdlib.h>

#define ARRAY_SIZE 10

int compInt(const void *p1, const void *p2)
{
    // FILL IN CODE HERE!!!
    // The purpose of this question is to test your understanding of
    // types in C. Thus, you will receive no point if your answer
    // has a type error. You will receive no point if your answer
    // has a syntax error or a warning message from gcc
```
```c
int main(int argc, char **argv)
{
    int i;
    int values[ARRAY_SIZE] = { 5, 4, 7, 3, 8, 2, 1, 0, 9, 6 };
    qsort(values, ARRAY_SIZE, sizeof(int), compInt);
    for (i = 0; i < ARRAY_SIZE; i++) {
        printf("%d ", values[i]);
    }
    printf("
");
    return EXIT_SUCCESS;
}
```

Here is the output from running the below program:

```
$ ./q3
0 1 2 3 4 5 6 7 8 9
```

The manual of qsort:

**SYNOPSIS**

```c
void qsort(void *base, size_t nmemb, size_t size,
           int (*compar)(const void *, const void *));
```

**DESCRIPTION**

The qsort() function sorts an array with nmemb elements of size size. The base argument points to the start of the array.

The contents of the array are sorted in ascending order according to a comparison function pointed to by compar, which is called with two arguments that point to the objects being compared.

The comparison function must return an integer less than, equal to, or greater than zero if the first argument is considered to be respectively less than, equal to, or greater than the second. If two members compare as equal, their order in the sorted array is undefined.

**RETURN VALUE**

The qsort() functions return no value.