Section Solution

This **sortedset** thing should try to conform as much as possible to the pictures. There should be a slab of memory to store **kInitialCapacity** nodes in addition to that admittedly annoying **int** all the way to the left. We need to remember how much space to allocate and how much of that allocated space is in use. We need to know how large client elements are so we can get the node allocations right. We also need to store the comparison function (since only it and not the **sortedset** implementation knows how to compare the mystery bytes so the set functions can travel left or right accordingly.) Once that is done, the **SetNew** implementation can more or less allocate everything and initialize the thing to behave as an empty tree.

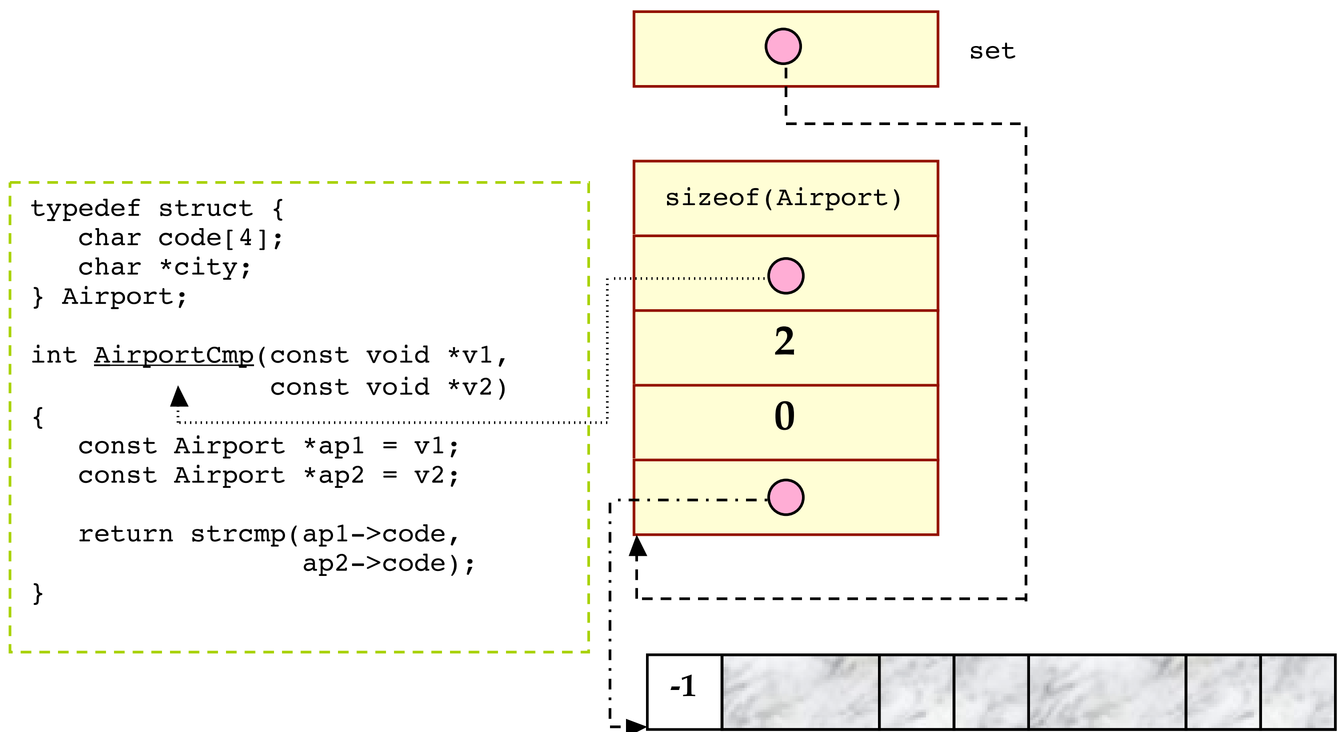
```
typedef struct {
   int *root;          // points to the offset index of the root.
   int logicalSize;    // number of active client elements currently stored.
   int allocatedSize;  // number of elements needed to saturate memory.
   int (*cmp)(const void *, const void *);
   int elemSize;       // client element size.
} sortedset;
```

Clearly there is an array flavor to this implementation. In particular, we need to manage an array (called **root** above) that identifies the location of the slate of bytes that store the nodes and the integers. I could have typed this as a **void *** instead, but I chose **int *** because the smallest entity that resides at this address is the integer that is initially **-1** but becomes **0** the moment that anything is added. Our doubling strategy will often leave us with raw, unused memory, so we'll need to track how many nodes there are and how many are being used. **cmp** is there so we remember how to maintain the BST property, and **elemSize** is there so we know how much storage is wedged in between the integer index pairs.

```
/**
 * Function: SetNew
 * Usage: SetNew(&stringSet, sizeof(char *), StringPtrCompare);
 *        SetNew(&constellations, sizeof(pointT), DistanceCompare);
 * ----------------
 * SetNew allocates the requisite space needed to manage what
 * will initially be an empty sorted set.  More specifically, the
 * routine allocates space to hold up to 'kInitialCapacity' (currently 4)
 * client elements.
 */

#define  NodeSize(clientElem)  ((clientElem) + 2 * sizeof(int))

static const int kInitialCapacity = 4;
void SetNew(sortedset *set, int elemSize,
            int (*cmpfn)(const void *, const void *))
{
   assert(elemSize > 0);
```

```
    assert(cmpfn != NULL);

    set->root = malloc(sizeof(int) + kInitialCapacity * NodeSize(elemSize));
    assert(set->root);

    *set->root = -1;        // set it empty
    set->logicalSize = 0;   // still empty
    set->allocatedSize = kInitialCapacity;
    set->cmp = cmpfn;
    set->elemSize = elemSize;
}
```

The bottom line here is that we build an empty set structure that **behaves** like an empty set. Here's the picture I'm hoping we both have in mind:



A good amount of code is shared by the **`SetSearch`** and the **`SetAdd`** functions. This shared code is consolidated to the **`FindNode`** function.

Motivation: Spelled Out

Both lookup and insertion require the same type of binary search. In the search case, we need to identify the index of the matching element, and in the insertion case, we need to identify where in the BST the element **could** reside if it doesn't reside there already. 90% of **`SetSearch`** and **`SetAdd`** is identical, so **`FindNode`** is an attempt to consolidate the shared code to one helper function.

```
/**
 * Function: FindNode
 * Usage: ip = FindNode(set, elem);
 *        if (*ip == -1) printf("ip points where this element belongs!");
 * ------------------
 * FindNode descends through the underlying binary search tree of the
 * specified set and returns the address of the offset into raw storage
 * where the specified element resides.  If the specified element isn't
 * in the set, FindNode returns the address of the –1 that would be updated
 * to contain the index of the element being sought if it were the
 * element to be inserted——that is, the address of the –1 that ended
 * the search.
 */

static int *FindNode(sortedset *set, const void *elem)
{
   void *curr;
   int comp, *root = set->root;

   while (*root != -1) { // while not addressing a leaf
      curr = (char *) (set->root + 1) + *root * NodeSize(set->elemSize);
      comp = set->cmp(elem, curr);  // compare client element to value at curr
      if (comp == 0) break;
      root = (int *)((char *) curr + set->elemSize);
      if (comp > 0) root++;
   }

   return root;
}
```

Just to be clear about **`FindNode`**'s behavior, you'd benefit from seeing how **`FindNode`** should operate on a tree storing **`double`** values:

set

sizeof(double)

6

4

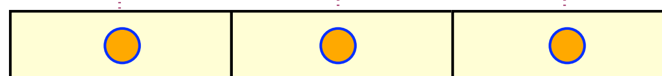
```
int DoubleCmp(const void *v1,
              const void *v2)
{
   const double *dp1 = v1;
   const double *dp2 = v2;
   double diff = *dp1 - *dp2;

   if (diff > 0.0) return 1;
   if (diff < 0.0) return -1;
   return 0;
}
```

0	5.245	3	1	8.214	2	-1	6.001	-1	-1	2.81828	-1	-1

Assume that the **`sortedset *`** variable **`set`** is initialized as above, and assume that the following array of **`double`**s has been declared as well.

```
double testCases[] = { 5.245, 3.141, 6.001 };
int *returnValues[sizeof(testCases)/sizeof(testCases[0])];
int i;

for (i = 0; i < sizeof(testCases)/sizeof(testCases[0]); i++)
   returnValues[i] = FindNode(set, &testCases[i]);
```

At the end of snippet execution, you'd expect to see **`returnValues`** initialized as follows:



Note that each pointer addresses not the **double**, but the **int** that indexes the **double**. If the search element exists, then you expect a pointer to some nonnegative integer. If the search element does not exist, then the pointer points to the integer that would be updated if the element in question were added.

Back To Code

Using **FindNode**, let's provide implementations for **SetSearch** and **SetAdd**. Fortunately, one of them is easy... we can just check to see whether or not a valid **FindNode** call returned the address of an **int** that stores a -1. At this point, we're thrilled that we wrote **FindNode** before we wrote this.

```
/**
 * Function: SetSearch
 * Usage: if (SetSearch(&staffSet, &lecturer) == NULL)
 *            printf("musta been fired");
 * -------------------
 * SetSearch searches for the specified client element according
 * the whatever comparison function was provided at the time the
 * set was created. A pointer to the matching element is returned
 * for successful searches, and NULL is returned to denote failure.
 */

void *SetSearch(sortedset *set, const void *elemPtr)
{
   int *node = FindNode(set, elemPtr);
   if (*node == -1) return NULL;
   return (char *) (set->root + 1) + *node * NodeSize(set->elemSize);
}
```

Actually, **SetAdd** isn't horrendous either. Assessing whether the element was previously added is easy. The tough part is the part that's tough with any type of generic programming. Should the element need to be added, we need to claim the next node in the array (which might require a **realloc** call), copy in the element's bit pattern, and update three indices. Next page, please:

```
/**
 * Function: SetAdd
 * Usage: if (!SetAdd(&friendsSet, &name)) free(name);
 * ----------------
 * Adds the specified element to the set if not already present.  If
 * present, the client element is not copied into the set.  true
 * is returned if and only if the element at address elemPtr
 * was copied into the set.
 */

bool SetAdd(sortedset *set, const void *elemPtr);
{
   int *child;
   void *dest;

   child = FindNode(set, elemPtr);
   if (*child != -1) return false;  // already there.. say we didn't add it.

   if (set->logicalSize == set->allocatedSize) Expand(set);
   *child = set->logicalSize++;

   dest = (char *) (set->root + 1) + (*child) * NodeSize(set->elemSize);
   memcpy(dest, elemPtr, set->elemSize);  // assume (quite reasonably) no overlap
   child = (int *)((char *) dest + set->elemSize);
   child[0] = -1;
   child[1] = -1;
   return true;
}

static void Expand(sortedset *set)
{
   set->allocatedSize *= 2;  // use doubling strategy
   set->root = realloc(set->root,
                sizeof(int) + set->allocatedSize * NodeSize(set->elemSize));
   assert(set->root != NULL);
}
```