Problem 3-4. Design a dictionary data structure in which search, insertion and deletion can each be processed in $O[1]$ time in the worst case. You may assume the set elements are integers drawn from a finite set 1,2,...,n and initialization can take O[n] time.

Solution.
The structure that immediately comes to mind is commonly known as a "hash table". Generally speaking, compilers tend to define the size of an integer to be the same as the size of a register, which is almost always the same as the the size of a memory address. This turns out to be very useful. The algorithm for generating my data structure is as follows (I assume that the set of integers we use as elements are key/value pairs, where the integer is the key with some pointer to some arbitrary associated data as a value):

```
input  : elems: The set of elements
let end ← A pointer to the beginning of a contiguous heap of memory
foreach elem in elems do
    let end + elem.key ← elem.value;
end
```

Algorithm 1: Structure Initialization

Since the algorithm performs a constant number of operations per element, this algorithm completes in $\Theta[n] \subseteq O[n]$ time. In general, if the above condition that the set of keys is not numeric or too large to use as direct modifiers to memory addresses (e.g. if they were strings), a function may be defined that transforms arbitrary unique values into memory addresses, known as the *hash function* for this hash, which adds only a constant amount of work to storing data for each element, thus not affecting time complexity. It can be seen, then, that searching is trivially:

```
input  : key: The key representing the data to fetch
output: A pointer to the data associated with key
return The data stored at end + key;
```

Algorithm 2: Structure Search

insertion is as easy as:

```
input  : key: The key representing the data to store
         value: The pointer to the data associated with key
let end + key ← value;
```

Algorithm 3: Structure Insertion

and deletion can be implemented:

```
input  : key: The key representing the data to remove
Free the memory pointed to by the value stored at end + key;
Free the memory at end + key;
```

Algorithm 4: Structure Deletion

As we can plainly see, each algorithm performs a constant number of atomic operations per search/insertion/deletion, putting their worst case running time as a subset of $O[1]$. (Again, if the keys themselves are in some way unsuitable to use as pointers, any instance of `end + key` can be replaced with `end` + the hash of `key`, without compromising the lower bound of the algorithm's worst-case time complexity).

■

Problem 3-9. A *concatenate* operation takes two sets S_1 and S_2, where every key in S_1 is smaller than any key in S_2, and merges them together. Give an algorithm to concatenate two binary search trees into one binary search tree. The worst case running time should be $O[h]$, where h is the maximal height of the two trees.

Solution.
If we know that our two trees T_1 and T_2 share the special relationship:

$$\forall x \in T_1 : \forall y \in T_2 (x < y)$$

we can trivially deduce that concatenating these trees is equivalent to making T_1 the left sub-tree of the minimum value in T_2. Just in case I'm not allowed to assume we already know the minimum of a tree, I'll give a simple algorithm to find it:

```
input  : tree: A reference to a tree object to find the minimum of
output: A reference to the minimum node of tree
let min ← tree;
while min → left not == NULL do
    min ← (min → left);
end
return min;
```

Algorithm 5: Minimum of a Tree

Note that the time complexity of this in the worst-case (wherein the minimum node occurs at the tree's maximum height) is a subset of $O[h]$ where h is the tree's height. As for concatenation, once we know the minimum of T_2, the algorithm is really quite simple:

```
input  : T_1: A reference to $T_1$
         min: A reference to the minimum node of $T_2$
(min → left)← T_1;
```

Algorithm 6: Concatenation of Trees under Special Conditions

A truly simple "one-liner". The time complexity of this algorithm is a subset of $O[1]$, so the time complexity of sequentially finding the minimum node of T_2 then storing T_1 at its left is a subset of $O[h+1] \subset O[h]$ so overall we can conclude that this solution in full has a time complexity that is a subset of $O[h]$.

■

Problem 3-10. In the *bin packing problem*, we are given n metal objects, each weighing between zero and one kilogram. Our goal is to find the smallest number of bins that will hold the n objects, with each bin holding one kilogram at most.
The *best fit heuristic* for bin packing is as follows. Consider the objects in the order in which they are given. For each object, place it into the partially filled bin with the smallest amount of extra room *after* the object is inserted.. If no such bin exists, start a new bin. Design an algorithm that implements the best-fit heuristic (taking as inputs the n weights $w_1, w_2, \ldots, w_n$ and outputting the number of bins used) in $O[n \log[n]]$ time.

Solution.

```
input  : w: The list of weights
output: The number of bins used

declare bins: a max-heap of bins to store weights in;
append an empty bin to bins;
foreach w_i in w do
    declare smallest- A reference to a bin;
    foreach bin in bins do
        let remain ← 1kg - (bin.weight + w_i);
        if remain ≥ 0 then
            if smallest not exists or remain < 1kg - (w_i + smallest → weight)
            then
                smallest ← a reference to bin;
            end
        end
    end
    /* Because bins is a heap, some sorting is implied in insertion and
       modification.                                                    */
    if smallest not exists then
        insert a bin of weight w_i into bins;
    else
        (smallest → weight)+= w_i
    end
end
return length of bins
```

Algorithm 7: Bin Packing Algorithm

Because the algorithm is contained inside a loop that will execute n times for n weights, (apart from some constant-time work done on the first two and last lines), we can express the worst-case time-complexity $C[n]$ of it as:

$$C[n] \in O\left[\sum_{m=1}^{n}[f[n,m]]\right] \tag{1}$$

for some function of n and m $f[n,m]$ that represents the time-complexity of the set of statements within the loop over the weights. In the worst-case scenario, each bin will be not completely full, but never have enough space for another weight (e.g. if every weight was 0.75 kg). If we ignore the constant-time work for declaring `smallest`, we can express the complexity of these inner statements $f[n]$ as $g[n] + h[n]$, where $g[n]$ is the complexity of the loop over all `bins`, and $h[n]$ is the complexity of the statements that follow this loop. Because of the nature of the relationship between weights and bin capacity in the worst case as discussed above, the loop over `bins` will always execute on the order of the number of entries in `bins`. This yields the complexity as a subset of $O[m]$, where m is indeed the index of summation from Equation 1. In the statements following this loop (the if/else bit) it's irrelevant which block is entered during execution: they each have the same time-complexity. We know that for a max-heap, insertion implies appending the element to the end of an array, and swapping it with its implied parent if said parent is greater in value. This process "bubbles" up the tree that is implicit within the heap's organization until everything is in its proper place. Thus, in the worst-case scenario this covers the height of the tree, which is known to be $O[\log[x]]$ for a heap containing x elements. If we modify the array, we must do the same thing at the point where the modification occurred. In the worst case, this will entail modifying a leaf node of the implicit tree by an amount that will require it to be the new root of the tree, thus it, too must cover the height of said tree, once again: $O[\log[x]]$. In the worst case, the heap should always contain m items at any one time, so our total complexity for this inner block $f[n]$ can be expressed:

$$f[n, m] \in O\left[m + \log[m]\right]$$

Now, putting it all together:

$$C[n] \in O\left[\sum_{m=1}^{n}\left[m + \log[m]\right]\right]$$

Let's try simplifying this, and in particular focus on the logarithmic term:

$$\sum_{m=1}^{n}\left[m + \log[m]\right] = \sum_{m=1}^{n}[m] + \sum_{m=1}^{n}[\log[m]]$$

By properties of logarithms, we can turn this sum of logarithms into a logarithm of a product over the same interval:

$$= \sum_{m=1}^{n}[m] + \log\left[\prod_{m=1}^{n}[m]\right]$$

... which of course gives us an argument to the logarithm that is the exact definition of $n!$, so:

$$= \sum_{m=1}^{n}[m] + \log[n!] = \sum_{m=1}^{n}[m] + n\log[n]$$

Now, the lower bound of the summation term on the left is constant, and the lower bound of $c + n\log[n]$ for some constant c is $n\log[n]$, and at any rate the growth of $n\log[n]$ totally dominates that of n, so even the upper bound shouldn't affect orders of magnitude. Therefore it's safe to say that:

$$C[n] \in O\left[n\log[n]\right]$$

■

Problem 3-21. Write a function to compare whether two binary trees are identical. Identical trees have the same key value at each position and the same structure.

Solution.
The only way to do this is with simultaneous traversal of both trees, most efficiently via Pre-order traversal.

Listing 1: are_identical - A function that tests if two binary trees are equal

```
//Look, Littlefoot: tree*'s!
bool are_identical(tree* a, tree* b) {
  if ( a != null ) {

    //if b is null but a isn't, these trees aren't the same.
    //Also if the data at a isn't equal to that of b
    if ( b == null || a->item != b->item)
      return false;

    //trees are identical if they have identical data values
    //AND their sub-trees are identical
    bool ident = are_identical(a->left, b->left);
    return ident && are_identical(a->right, b->right);
  }

  //if a is null but b isn't, these trees aren't the same.
  else if ( b != null ) {
    return false;
  }

  //if a and b are both null, they're equal
  return true;
}
```

Note that this function is more efficient for differences in trees that occur closer to the top-left of said trees. I don't actually have anything more to say, just pointing it out.

■