COMP 161 - Lecture Notes 14 - Search and Sort

In these notes we apply the basic recursive and iterative design template to the problems of searching and sorting.

Structural Recursion and std::vectors

Structural recursions is about making recursive procedures that recurse on the recursive structure of the data. The most basic form of this structure come from having an empty collection BASE CASE and a non-base case where the data is deconstructed into the first element and all of the rest. The "rest" is a smaller version of the original structure. More generally, we can identify any non-recursive smallest size[1] as a base case and deconstruct the non-base case in any way so long as repetition of the decomposition eventually results in a base case[2]. Such structures exists abstractly for just about any collection type. However, not all collections support recursive decomposition or do so in an inefficient manner. The C++ *std::vector* does not support recursive decomposition directly.

[1] one element, two elements, etc

[2] all but the last + last, left half + right half, etc.

Thankfully, any structure with indexed elements can be managed recursively thought the recursive handling of the set of index values. A vector of size s used the integer interval $[0, s)$ for its indices. When the vector is empty, then the interval $[0, 0)$ is also empty[3]. The first of this interval is 0 and the rest is $[1, s)$. Similarly, the last is $s - 1$ and all but the last is $[0, s - 1)$. We can generalize this for any range of contiguous positive integers $[a, b)$. The interval is empty if $a \geq b$. When $a < b$, the first is a and the rest is $[a + 1, b)$.

[3] $[a, b)$ is empty if $a \geq b$

To recursively process a vector we must write a procedure that takes the index interval bounds *first* and *last*. We don't always need both bounds. Pure functions on vectors can often exclude the last when recursing first to last or last when recursing last to first. Having both, however, provides maximum flexibility. If you need to mutate the vector and doing so changes the size, then you're probably going to need to shift first and last to account for the change in the vector's structure. You also can design with both bounds so that you can defer choosing the exact pattern of recursion to implementation time. Sometimes you don't realize that a pattern won't work until you really start working with the details.

If your goal is to work with a vector in its entirety, then the extra parameters change the basic interface to that problem. To hide them we use a OVERLOADED function where one version takes on the vector and calls the recursive procedure with first equal to zero and last equal to the size of the vector. This will cause the recursive variant of the function to consider all the elements of the vector.

Figure 1 gives the basic template for first + rest structural recursion on vectors with a top-level variant that works on the whole vector by using the recursive variant as a helper. We'll be applying this template to search and sort.

```
/**
 *
 *@param v the vector
 *@param first the smallest index to be processed
 *@param last the excluded upper bound of the interval of indexes
     to be
 * processed
 *@return ...
 *@pre 0 <= first,last < v.size()
 *@post ...
 */
... foo([const] std::vector< ... >& v, int first, int last){

  if( first >= last ){
    // base case
    ....
  }
  else{
    ... v[first] ... foo(v,first+1,last)
  }

}

/**
 *
 *@param v the vector
 *@return
 *@post ...
 */
... foo([const] std::vector< ... >& v){
  ... foo(v,0,v.size()) ...
}
```

Figure 1: Structural Recursion Template for Vectors

Search: The Problem

Search is a fundamental problem in computing. Given a collection[4], find a specific element, typically called the *search key*[5] in that collection. Several variations can occur: find the first occurrence, the last occurrence, and more generally the $k-th$ occurrence. With the *std::vector*, we want to return the index of occurrence. A simpler version of search is the *contains* predicate that returns true if the col-

[4] std::vector for now

[5] or just *key*

lection contains at least one occurrence of the search key. Finally, another search-like procedure is a counting procedure that returns the total number of occurrences.

For these notes we'll look at the version of search that examines the entire vector and returns the index of the first occurrence of the key. The declaration for this function is given in figure 2 with tests in figure 3.

Figure 2: The Basic Search Function

```
/**
 * Compute the location of the first occurrence of the integer key
 * in the vector data.
 * @param data vector of integers
 * @param key search value
 * @return -1 if key is not found, otherwise the index where key
 * is first found
 * @pre none
 * @post none
 */
int search(const std::vector<int>& data,int key);
```

Figure 3: Tests for Basic Search

```
TEST(search,all){

  EXPECT_EQ(-1,search(std::vector<int>({}),1));
  EXPECT_EQ(-1,search(std::vector<int>({2}),1));
  EXPECT_EQ(0,search(std::vector<int>({1}),1));
  EXPECT_EQ(1,search(std::vector<int>({1,3,5}),3));
  EXPECT_EQ(0,search(std::vector<int>({1,3,1}),1));
  EXPECT_EQ(2,search(std::vector<int>({1,3,5}),5));

}
```

This basic interface and the tests should work regardless of the underlying implementation.

Search: The Solutions

The recursive and iterative versions share a lot in common. We'll start with the recursive variant and then look at the iterative version. In both cases it's clear that we only need read-only access to the vector and using *pass-by-const-reference* would be a good idea.

Recursive Search

For the recursive implementation we need the variant of the search that accepts the bounds of the search range in order to recurse along that interval as shown in figure 4. It's wroth noting this function, as declared, doesn't need to be recursive. We could simply view it as a more general version of search: find the key within this sub-section of the vector. The more generic interface to search gives us the space we need to enable recursion.

Figure 4: Recursive-Capable Search

```
/**
 * Compute the location of the first occurence of the integer key
 * in the vector data for the index range [fst,lst).
 * @param data vector of integers
 * @param fst the lower bound of the search range
 * @param lst the excluded upper bound of the search range
 * @param key search value
 * @return -1 if key is not found, otherwise the index where key
 * is first found
 * @pre fst <= lst
 * @post none
 */
int search(const std::vector<int>& data,int fst, int lst, int key);
```

When testing the more generic search we should test it not only for whole vector searches but partial vector searches as we see in figure 5.

Figure 5: Recursive-Capable Search Tests

```
TEST(search,some){

 // search all
 EXPECT_EQ(-1,ln14::search(std::vector<int>({}),0,0,1));
 EXPECT_EQ(1,ln14::search(std::vector<int>({1,2,3,2,1}),0,5,2));
 EXPECT_EQ(-1,ln14::search(std::vector<int>({1,2,3,2,1}),0,5,7));
 // search some
 EXPECT_EQ(3,ln14::search(std::vector<int>({1,2,3,2,1}),2,5,2));
 EXPECT_EQ(-1,ln14::search(std::vector<int>({1,2,3,2,1}),2,3,2));

}
```

The top-level search[6] simply calls the more generic version with the interval for the entire vector as shown in figure 6.

To work out the recursive implementation of the generalized search we start with the base case. When a vector is empty, it cannot contain the key so return -1. With the non-empty case we usual

[6] search the whole vector

```
int search(const std::vector<int>& data,int key){
  return search(data,0,data.size(),key);
}
```

Figure 6: Top-Level Search: Recursive Implementation

think to recurse on the rest[7] then combine that result with something relative to the first[8] and in this case the key. A key observation to make here is that we don't need the result of the recursive call if the first element is the key. This allows us to break out the non-empty case into two cases where one doesn't need or use recursion on the rest.

In figure 7 we see the finished code for the recursive search. The two cases of the non-empty portion of the main conditional have been flattened into the main conditional itself rather than nesting a second conditional in the else of the main conditional.

[7] search(data,fst+1,lst,key)

[8] data[fst]

```
int search(const std::vector<int>& data,int fst, int lst, int key){
  if( fst >= lst ){
    return -1;
  }
  else if( data[fst] == key ){
    return fst;
  }
  else{
    return search(data,fst+1,lst,key);
  }
}
```

Figure 7: Search Some: Recursive Implementation

Iterative Search