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.

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 $[a + 1, b)$.

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.

[1] one element, two elements, etc

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

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

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.

Figure 1: Structural Recursion Template for Vectors

```
/**
 *
 *@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()) ...
}
```

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 technique. How you solve the problem does not change the problem itself. In this case, we're interested in searching an entire vector and this function captures that problem succinctly.

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 can, and should, view it as a more general version of search: find the key within this subsection of the vector. The more general problem of searching a part of a vector gives us the flexibility 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, the problem we originally set out to solve, but for partial vector searches as we see in figure 5. Test the procedure as it stands on its own, not just for some subset of its usage.

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.

[6] search the whole vector

To work out the recursive implementation of the generalized search we start with the base case. When a vector is empty, it can-

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

Figure 6: Top-Level Search: Recursive Implementation

not contain the key so return -1. With the non-empty case we work out the problem in terms of the first[7] and the result of recursing on the rest[8]. In the context of search this boils down to the following observation: either the first is the key, the key is in the rest, or the key is not in the vector at all. A complete case analysis reveals four possible situations[9]: the first is the key and the rest contains the key, the first isn't the key and the rest contains the key, the first is the key and the rest doesn't contain the key, or the key is neither the first nor is it in the rest. By recursively calling search on the rest we expect to get the location of key in the rest or a -1 if it's not in the rest. For example, if the vector v contains $\{2, 3, 2, 1\}$ and the search key is 2 then the recursive call *search(v,1,v.size(),2)* should return 2, but because 2 is also at location 0, the first, we should return 0 in favor of 2. If we were searching for the key 4 then the recursive call should return -1 and given that the first, 2 isn't 4, we should return -1. By continuing to work examples like this we see that the recursion is necessary if and only if the first isn't the key. In the case where the first element is the search key, we should just return the first index without recursively searching the rest.

[7] data[fst]

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

[9] two potential locations (first and rest) with two potential states(contains and doesn't contain)

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.

```
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

It's worth stopping to check our implementation against our case analysis. The most important case is the recursive one: the search range of the vector is either empty or it's not. Empty case is handled by the *if* and the non-empty case is caught either by the *else if* or the *else*. When the search range isn't empty we can condense the four cases down to two: the key is the first or its existence and location can be determined through recursively searching the rest. The *else if* case should catch the case where the first is the key. The *else* covers the other case.

Iterative Search

The basic logic of iteration is to traverse the structure and accumulate the solution while you traverse. Vector traversals work the same way as string taversals. You simply count your way through the set of index values[10]. By default we start at zero and count up, but other counting schemes are possible. The solution to this problem is an index value or -1. Developing the recursive solution showed us that we shouldn't need to continue searching once we discover the first occurrence of the search key. In an iterative world this means stopping the traversal and returning the current index. Combine this with the fact that we need to accumulate an index value, we notice that an extra accumulator isn't required, the loop is already accumulating what we need in it's counter. We need to be careful though. Basic accumulator logic dictates that we return the accumluated value when traversal is done. When we don't find the search key the loop's counter will be the size of the vector and we want -1. Similarly, if the vector is empty, the initial counter value is 0 and we need to return -1. The fix here is to recognize that when we find the key, we can return the counter, and if we don't find the key we just return the literal -1.

[10] [0,size)

In figure 8 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.

Figure 8: Search: Iterative Implementation

```
int search(const std::vector<int>& data,int fst, int lst, int key){

  for(unsigned int i{0}; i < data.size() ; ++i){
    if( data[i] == key ){
      return i;
    }
  }

  return -1;
}
```

It's worth your time to really analyze this solution and clearly identify how we tweaked the basic "traverse and accumulate" logic to arrive at this implementation. Imagine we're searching the vector v containing $\{2, 3, 1, 2\}$ for the search key 1. If we're currently looking at index $i = 1$ then we've accumulated the fact that everything before 1 doesn't contain the key[11]. The vector element at 1 isn't the key, so we should leave the accumulated value as -1. If the key were 2 then we should have discovered the key on a previous iteration and the accumulator would be 0, the location of the first occurrence of the key in the previously traversed structure. Finally, consider the case where the key is 2 and the current location is 3. We should have discovered 2 at location 0 on a previous iteration and upon rediscovering 2 and location 3 we should preserve the previous accumulator value because our goal is to find the first occurrence. If you wanted to capture this kind of on the nose iterative thinking in code you'd end up with what you see in figure 9. From this perspective our version is an optimization of the full traversal version[12].

[11] i.e. our accumulator is -1

[12] why keep going once you've found it and why accumulate the location twice, once for the traversal loop and once for the iteration

```
int search(const std::vector<int>& data,int fst, int lst, int key){

  int loc{-1};
   for(unsigned int i{0}; i < data.size() ; ++i){
     if( data[i] == key && loc == -1 ){
        loc = i;
     }
   }

   return loc;
}
```

Figure 9: Search: Iterative Implementation with explicit accumulation and full traversal

Sort: The Problem