Lecture 27 − Sorting, Continued

J. Zarnett
jzarnett@uwaterloo.ca

Department of Electrical and Computer Engineering
University of Waterloo

July 28, 2016

Last time, we looked at the selection sort.
Now we'll look at a similar sorting algorithm: insertion sort.

Recall the scenario: you have been taking notes for a course on loose-leaf paper and have been numbering the pages.

You accidentally knock over the pile of pages and they scatter and fly everywhere in your room, landing all over.

You have an unsorted pile and a (currently-empty) sorted pile.

Insertion Sort

Take the first page off the unsorted pile and put it in the sorted pile.
  It may be any number. Suppose it's number 22.

Grab the next page from the unsorted pile and look at its number.
  Let's say it's number 4.

Add it to the sorted pile in the appropriate order.
  In this case, add number 4 in front of number 22.

Take the next page from the unsorted pile (number 17).
  Put it into the sorted pile in its place (between 4 and 22).

Repeat this procedure until all the pages are sorted again.

This algorithm using extra memory (not in-place) looks like:

For an input array of capacity n, allocate an output array of capacity n. Then, for each entry of the input array:

1. Remove the entry from the input array
2. Find where to put it in the output array
3. Put it in the output array at that position

Insertion Sort can also be done in-place:

Divide the unsorted array (conceptually) into two smaller arrays:
    The 1st contains entries processed and inserted in order.
    The 2nd contains entries that have not yet been processed.

We iterate over all entries of the array, taking entries from the second of the small arrays, processing them, and adding them to the first.

We're finished when the 1st array has grown to the size of the input.
    This means all elements are processed and sorted.

Given an array of *n* entries:

For *index* from 1 to $(n - 1)$:

1. Set *temp* to the value of the array at *index*
2. For *pos* from *index* to 1
   a) Check if *temp* is greater than the value of the array at $(pos - 1)$
      If yes, the exit the inner loop
      If no, swap value of array at *pos* with the value at $(pos - 1)$
3. Set value of array at *pos* to *temp*

Consider sorting {12, 36, 20, -1} using the insertion sort.

After iteration x, the first $x + 1$ values have been sorted.
  Progress is shown by the red highlighting.

Iteration 1: Array Contents = {12, 36, 20, -1}
  Compare 36 to 12
  Exit inner loop due to larger value
  Store 36 at index 1

Iteration 2: Array Contents = {12, 36, 20, - 1}
  Compare 20 to 36
  Store 36 at index 2
  Compare 20 to 12
  Exit inner loop due to larger value
  Store 20 at index 1

Iteration 3: Array Contents = {12, 20, 36, -1}
    Compare -1 to 36
    Store 36 at index 3
    Compare -1 to 20
    Store 20 at index 2
    Compare -1 to 12
    Store 12 at index 1
    Exit inner loop due to start of array
    Store -1 at index 0

The array, after sorting, contains {-1, 12, 20, 36}.

Consider sorting {12, 36, 42, -1, 99, 20, 10, 19, 70}.

{12, 36, 42, -1, 99, 20, 10, 19, 70} // Initial array values
{12, 36, 42, -1, 99, 20, 10, 19, 70} // After iteration 1
{12, 36, 42, -1, 99, 20, 10, 19, 70} // After iteration 2
{-1, 12, 36, 42, 99, 20, 10, 19, 70} // After iteration 3
{-1, 12, 36, 42, 99, 20, 10, 19, 70} // After iteration 4
{-1, 12, 20, 36, 42, 99, 10, 19, 70} // After iteration 5
{-1, 10, 12, 20, 36, 42, 99, 19, 70} // After iteration 6
{-1, 10, 12, 19, 20, 36, 42, 99, 70} // After iteration 7
{-1, 10, 12, 19, 20, 36, 42, 70, 99} // After iteration 8

Note that a total of 8 iterations are required to sort 9 values.

In general, insertion sort requires $n - 1$ iterations to sort n values.

```
void insertion_sort( int[] data, int length )
{
    for( int index = 1; index < data.Length; index++ )
    {
        int pos;
        int temp = data[index];

        for( pos = index; pos > 0; pos-- )
        {
            if( temp > data[pos - 1] )
            {
                break;
            }
            else
            {
                data[pos] = data[(pos - 1)];
            }
        }
        data[pos] = temp;
    }
}
```

The selection sort and the insertion sort work well on arrays, but how can they be used on linked lists?

To use these sorting algorithms, you simply need to keep track of an unsorted list of elements and a sorted list of elements.

Both can be used on linked lists without needing additional storage.
    An element will only exist in one list at a time.
    Therefore, the amount of storage required is constant.

Given a list of n elements:

For each of the n elements:

1 Remove the minimum element from the unsorted list
2 Append that element to the sorted list.

The difficulty: removing the minimum element from the unsorted list.

```
void selection_sort( linked_list* original )
{
    linked_list* sorted = new linked_list;
    initialize( sorted );

    list_entry* tmp = find_min( original );
    while( tmp != 0 )
    {
        append( sorted, tmp->data );
        delete tmp;
        tmp = find_min( original );
    }

    delete original;
    original = sorted;
}
```

The find_min function is too big to put on the slides.

There are a number of special case handling blocks, such as when the list is empty or contains only one element...

[In class demo: the code for this function]

Given a list of *n* elements:

For each of the *n* elements:

1 Remove the head element from the unsorted list
2 Insert that element at the correct position in the sorted list.

The difficulty: adding the element to the sorted list.

```
void insertion_sort( linked_list* original )
{
    linked_list* sorted = new linked_list;
    initialize( sorted );

    list_entry* tmp = remove_first( original );
    while( tmp != null )
    {
        insert_in_order( sorted, tmp->data );
        delete tmp;
        tmp = tmp = remove_first( original );
    }

    delete original;
    original = sorted;
}
```

The insert_in_order function is also too big to put on the slides.

Like find_min, there are a number of special case handling blocks, such as when the list is empty or contains only one element...

[In class demo: the code for this function]

Constructing a sorted linked list is often more efficient than sorting a linked list after creation.

Insertion takes more time, but then sorting is not required.

This minimizes the number of traversals of the linked list.

Given the existence of an `insert_in_order` function, simply use that every time an element is added.

As a general rule, you should always build sorted data structures if you plan to search them.

If you think sorting a linked list is tricky, imagine the difficulty of sorting data stored sequentially on magnetic tape units.

An interesting article on sorting data on magnetic tape units:
`http://portal.acm.org/ft_gateway.cfm?id=808961&type=pdf`

For an estimated 20 000 records of 200 characters, the sorting algorithm described required 270 minutes to complete.

That's 4.5 hours to sort 4 MB of character records.

Sorting Performance Comparison

The performance of selection sort, insertion sort, and sorted lists can be easily compared.

If a list contains 100 000 random integers:

Operation	Time (s)
Linked list creation time	0:00.01
Selection sort time	1:45.07
Insertion sort time	1:09.26
Sorted list creation time	1:10.12

Outside of an academic context, it is rare that you will implement your own search algorithm, but doing so is important for learning.

In practice, when you use the built-in collection types of the language, they have `Sort` functions that will sort the data for you.

For integers and other built-in types, you may use this `Sort` function and the system will be able to sort the data correctly.

But what about programmer-defined types like classes?

To sort programmer-defined types, we need to decide on a sort order.
We might sort students by last name, or student ID number...

And we need to find a way to tell the system what we chose.
The system can't know in advance what types we are going to define...

Soon, we'll examine how to do that.