ECE 264: Advanced C Programming

Week5 Lecture Notes 7/8/19 - 7/12/19

1 Topics

This week we discussed more examples of recursive algorithms, enums, unions, complex structures, and dnamic data structures.

- 7/8 and 7/9: Mergesort, analysis of mergesort
- 7/9 and 7/10: Unions, Enums, Brief introduction to shallow copying, Complex structures, 2D Arrays (on heap).
- 7/11 and 7/12: Dynamic data structures: linked lists, stacks, application of stacks (arithmetic expression parsing), queues.

2 Mergesort

Merge sort is an application of divide and conquer recursion to sort an array. The heart of merge sort is the merge operation, which combines two already sorted arrays to produce a new sorted array. To merge two sorted arrays, imagine you have two cursors, which start at the beginnings of the two arrays. Look at the two elements pointed to by the cursor: add whichever element is smaller to the output array, then move that cursor forward by 1 element. (If one of the cursors is already at the end of its array, the other cursor always "wins.")

This merge operation gives us a way of combining the solutions of two smaller problems to solve the larger problem of sorting an array:

1. Divide the array into two pieces
2. Sort the two pieces by recursively calling the same function
3. Use merge to merge the two resulting sorted pieces

So what should the base case be? How do we make sure we don't keep sorting smaller and smaller arrays? Note something simple: an array with only one element is already sorted!

Below is the recursive code skeleton

```
void Mergesort(int* arr, int left, int right) {
        //base case
        if (left >= right)
          return;
        //compute middle index. Left-subarray always >= right sub-array
        int nels = (right ? left + 1) / 2;
        int nelsLeft = (nels + 1) / 2 ;
        int mid = left + nelsLeft - 1;
        //Recursive case: sort the smaller sub-arrays
        Mergesort(arr, left, mid)
        Mergesort(arr, mid+1, right);
        //merge the sorted sub arrays
        Merge(arr, left, mid, right);
}
```

2.1 Analysis

Now, we try to find the approximate time it takes for the mergesort to complete for an input size of `n` elements. A call to `Mergesort` in the recursive case always sorts a subarray roughly equal to half the size of the parent array. As recursion proceeds, the recursive case creates 2 halves in every level of the recursion.

If we assume that `n` is a perfect power of two, we will have $log_2 n$ levels of recursion. Every level creates twice the number of sub arrays as seen in previous level. So starting from level 0, we will have a total of $log_2 n$ +1 levels.

If we assume that the merge operation takes `cn` time for `n` inputs, then at the first level, there are two calls to `merge`. Each of these calls are operating on arrays of size `n/2`. Adding up, we see that the total time consumed by `merge` is `cn`. We observe that at every level, `merge` consumes `cn` time all calls combined.

So, we get an approximate time of `cnlog n+cn`.

2.2 More examples of recursion: depth first search

TBD