CS211 - Data Structures

Spring 2013

Professor Christopher Andrews

Sorting Algorithms

Insertion sort

- Insertion sort works by conceptually breaking our list into two pieces: the first piece is sorted, the second is not
- We walk through the unsorted items and one by one insert them into the right location in the sorted list.
    - to save ourselves some effort in the insertion, we look for the right location by walking through the sorted list form the end until we find the spot where our value should be placed
    - as we visit each value, we copy it one slot to the right
    - as a result, we have made a slot for our value and extended the length of the sorted list by one

```
// i represents the boundary between the sorted and unsorted list
for i = 1 to length-1
      item  = L[i]
      j = i-1
      // j is our cursor as we walk backwards through the sorted list
      while (j >= 0 and L[j] > item)
            L[j+1] = L[j]
            j--
      // we have found our space for the item
      L[j+1] = item
```

Mergesort

Mergesort is a fairly basic application of divide and conquer
- split the list in half
- sort the two halves
- merge the two halves together by interleaving them in the correct order

```
// this sorts the portion of the list that starts at index p and ends at
index r (inclusive)
mergesort(A, p, r)
      // if p is not less than r, there is no list to sort
      if (p < r)
            m = floor((p+r)/2)
            mergesort(A, p, m)
            mergesort(A,m+1, r)
            merge(A, p, m, r)

// this is where the real work happens. This merges the two sublists of list
A that go from p to m and m+1 to r inclusive. It is assumed that the two sub
lists are sorted and at the end, the sublist from p to r will be sorted
merge(A, p, m, r)
      // merge requires a backup list to copy values into for the merging
      new B[]
      // b is the next open index in temporary list B
      b = 0
      // i is the index of the smallest value in the left sublist that hasn't
been merged
      i = m
      // j is the index of the smallest value in the right sublist that
hasn't been merged
      j = q+1

      // until one of the two sublists runs out of values, pick the smallest
of the two next available values and copy it to B
      while (i <= m AND j <= r)
            if (A[i] <= A[j])
                  B[b++] = A[i]
                  i++
            else
                  B[b++] = A[j]
                  j++

      // if the left sublist is not exhausted, copy it all to B
      for (;i<=m;i++)
            B[b++] =A[i]

      // if the right sublist is not exhausted, copy it all to B
      for (;j<r; j++)
            B[b++] = A[j]

      // copy the sorted sublist out of B and back into A
      for (i = 0; i < b; i++)
            A[p+i] = B[i]
```

Quicksort

- we partition the list into two halves, based on a pivot value
    - everything in one half is less than the pivot, everything in the other is greater (technically, one of those needs to actually include the pivot value if it occurs multiple times in the data
- when then sort the two halves
- we are guaranteed that since the list is partitioned that if the two halves are sorted, the whole list will be sorted when we stick them back together
- the partition step is the important part
    - we consider our list to really be three lists — one with the values less then the pivot, one with values greater, and one with items we haven't looked at yet
    - we then look at the first item of the unpartitioned list
        - if it is larger, leave it where it is and move to the next value (basically puts this at the end of the "larger" list)
        - if it is smaller, swap it with the head of the larger list, and then increase the size of the "smaller" list
    - when we are done, we move the pivot value to the head of the "larger than" list and return this index

```
// this sorts the portion of the list that starts at index p and ends at
index r (inclusive)
quicksort(A, p, r){
      // if p is not less than r, there is no list to sort
      if p < r
            q = partition(A, p, r)
            quicksort(A, p, q−1)
            quicksort(A, q+1, r)

// This partitions the sublist from index p to index r inclusive
// it returns the index of a "pivot" value, such that all values to the
// left of it are less than it and all values to the right are greater than
// or equal to it
partition(A, p,r)
      // pick the last value in the sublist to be our pivot
      pivot = A[r]
      // split indicates the first value not in the "less than" list
      split = p
      // partition the list
      for i=p to i=r−1
            if A[i] < pivot
                  swap(split, i)
                  split++
      // move the pivot to the correct index and return that position
      swap(split, r)
      return split
```