CSE214 Data Structures

Quick Sort, Merge Sort

YoungMin Kwon

Quick Sort

- Divide and conquer pattern
  - Divide: if S has at least two elements
    - Pick an element called pivot and move the elements to
    - L: elements less than pivot
    - E: elements equal to pivot
    - G: elements greater than pivot
  - Conquer: recursively sort L and G
  - Combine: move the elements in L, E, G to S in this order

```
public static <E extends Comparable<E>> void quickSort(ArrayList<E> s) {
    int n = s.size();
    if(n < 2)
        return;

    ArrayList<E> lt = new ArrayList<E>();
    ArrayList<E> eq = new ArrayList<E>();
    ArrayList<E> gt = new ArrayList<E>();
    E pivot = s.get(new Random().nextInt(n)); //randomized quicksort
    for(E e : s) {
        int cmp = e.compareTo(pivot);
        if(cmp == 0)     eq.add(e);
        else if(cmp < 0) lt.add(e);
        else             gt.add(e);
    }

    quickSort(lt);
    quickSort(gt);

    s.clear();
    for(E e : lt) s.add(e);
    for(E e : eq) s.add(e);
    for(E e : gt) s.add(e);
}
```

Quick Sort without Using Extra Space

```
public static <E extends Comparable<E>> void quickSort(E[] a, int l, int r) {
    if(r <= l)
        return;

    E pivot = a[r];
    int i = l, j = r - 1;
    while(true) {
        //TODO:
        // while a[i] <  pivot, i++ (no need to check the bound)
        // while a[j] >= pivot, j-- (check the bound)
        // if i >= j, break
        // otherwise swap a[i] and a[j]
    }
    //TODO:
    // swap a[i] and a[r]
    // call quick sort for a[l .. i-1]
    // call quick sort for a[i+1 .. r]
}

public static <E extends Comparable<E>> void quickSort(E[] a) {
    quickSort(a, 0, a.length-1);
}
```

Merge Sort

- Divide and conquer for sorting
  - Divide: if S has zero or one element, return S; otherwise divide S into
    - S_1 with the first half of S and
    - S_2 with the second half of S
  - Conquer: recursively sort S_1 and S_2
  - Combine: merge S_1 and S_2 into S

Merge Sort

```
protected static <E extends Comparable<E>> void merge(E[] s1, E[] s2, E[] s) {
    int n = s1.length + s2.length;
    for(int i = 0, j = 0; i + j < n; ) {
        if(i < s1.length && j < s2.length)
            s[i+j] = (s1[i].compareTo(s2[j]) < 0) ? s1[i++] : s2[j++];
        else if(i < s1.length)  //j is at the end of s2
            s[i+j] = s1[i++];
        else                    //i is at the end of s1
            s[i+j] = s2[j++];
    }
}

public static <E extends Comparable<E>> void mergeSort(E[] a) {
    int n = a.length;
    if(n < 2)
        return;
    E[] s1 = Arrays.copyOfRange(a, 0, n/2); //s1 has a[i] for 0 <= i < n/2
    E[] s2 = Arrays.copyOfRange(a, n/2, n); //s2 has a[i] for n/2 <= i < n
    mergeSort(s1);
    mergeSort(s2);
    merge(s1, s2, a);
}
```

Merge Sort Using Indexes

```
//merge a[l1..r1) and a[l2..r2) to a[l1...r2)
protected static <E extends Comparable<E>>
void merge(E[] a, int l1, int r1, int l2, int r2) {
    int n = r1 - l1 + r2 - l2;
    E[] m = (E[]) new Comparable[n];
    //TODO:
    //merge a[l1..r1) and a[l2..r2) to m
    //copy m to a[l1..r2)
}

public static <E extends Comparable<E>>
void mergeSort(E[] a, int l/*inclusive*/, int r/*exclusive*/) {
    int n = r - l;
    if(n < 2)
        return;
    //TODO:
    //call mergeSort for the first half and the second half of a[l..)
    //merge the two sorted halves
}

public static <E extends Comparable<E>>
void mergeSort(E[] a) {
    mergeSort(a, 0, a.length);
}
```