MORE MERGE SORT

• Java implementation

• Time complexity

• Generic merging and sets
Java Implementation of Merge-Sort

- Interface SortObject

```java
public interface SortObject {

    // sort sequence S in nondecreasing order using comparator c
    public void sort (Sequence S, Comparator c);
}
```
Java Implementation of Merge-Sort(cont.)

```java
public class ListMergeSort implements SortObject {
    public void sort(Sequence S, Comparator c) {
        int n = S.size();
        if (n < 2) return; // a sequence with 0 or 1 element is already sorted.
        // divide
        Sequence S1 = (Sequence) S.newContainer();
        // put the first half of S into S1
        for (int i=1; i <= (n+1)/2; i++) {
            S1.insertLast(S.remove(S.first()));
        }
        Sequence S2 = (Sequence) S.newContainer();
        // put the second half of S into S2
        for (int i=1; i <= n/2; i++) {
            S2.insertLast(S.remove(S.first()));
        }
        sort(S1,c);  // recur
        sort(S2,c);
        merge(S1,S2,c,S);  // conquer
    }
}
```
public void merge(Sequence S1, Sequence S2, Comparator c, Sequence S) {
    while(!S1.isEmpty() && !S2.isEmpty()) {
        if(c.isLessThanOrEqualTo(S1.first().element(), S2.first().element())) {
            // S1’s 1st elt <= S2’s 1st elt
            S.insertLast(S1.remove(S1.first()));
        }
        else { // S2’s 1st elt is the smaller one
            S.insertLast(S2.remove(S2.first()));
        }
    }

    if(S1.isEmpty()) {
        while(!S2.isEmpty()) {
            S.insertLast(S2.remove(S2.first()));
        }
    }

    if(S2.isEmpty()) {
        while(!S1.isEmpty()) {
            S.insertLast(S1.remove(S1.first()));
        }
    }
}
Running Time of Merge-Sort

- **Proposition 1**: The merge-sort tree associated with the execution of a merge-sort on a sequence of $n$ elements has a height of $\lceil \log n \rceil$.

- **Proposition 2**: A merge sort algorithm sorts a sequence of size $n$ in $O(n \log n)$ time.

- We assume only that the input sequence $S$ and each of the sub-sequences created by each recursive call of the algorithm can access, insert to, and delete from the first and last nodes in $O(1)$ time.

- We call the time spent at node $v$ of merge-sort tree $T$ the running time of the recursive call associated with $v$, excluding the recursive calls sent to $v$’s children.
Running Time of Merge-Sort (cont.)

- If we let $i$ represent the depth of node $v$ in the merge-sort tree, the time spent at node $v$ is $O(n/2^i)$ since the size of the sequence associated with $v$ is $n/2^i$.

- Observe that $T$ has exactly $2^i$ nodes at depth $i$. The total time spent at depth $i$ in the tree is then $O(2^i n/2^i)$, which is $O(n)$. We know the tree has height $\lceil \log n \rceil$

  Therefore, the time complexity is $O(n \log n)$
Set ADT

• A Set is a data structure modeled after the mathematical notation of a set. The fundamental set operations are union, intersection, and subtraction.

• A brief aside on mathematical set notation:
  - A ∪ B = { x: x ∈ A or x ∈ B }
  - A ∩ B = { x: x ∈ A and x ∈ B }
  - A − B = { x: x ∈ A and x ∉ B }

• The specific methods for a Set A include the following:
  - union(B):
    Set A equal to A ∪ B.
  - intersect(B):
    Set A equal to A ∩ B.
  - subtract(B):
    Set A equal to A − B.
Generic Merging

**Algorithm** genericMerge($A$, $B$):

- **Input**: Sorted sequences $A$ and $B$
- **Output**: Sorted sequence $C$

let $A'$ be a copy of $A$ { We won’t destroy $A$ and $B$ }
let $B'$ be a copy of $B$

while $A'$ and $B'$ are not empty do
  $a ← A'$.first()
  $b ← B'$.first()
  if $a < b$ then
    aIsLess($a$, $C$)
    $A'$.removeFirst()
  else if $a = b$ then
    bothAreEqual($a$, $b$, $C$)
    $A'$.removeFirst()
    $B'$.removeFirst()
  else
    bIsLess($b$, $C$)
    $B'$.removeFirst()

while $A'$ is not empty do
  $a ← A'$.first()
  aIsLess($a$, $C$)
  $A'$.removeFirst()

while $B'$ is not empty do
  $b ← B'$.first()
  bIsLess($b$, $C$)
  $B'$.removeFirst()
Set Operations

• We can specialize the generic merge algorithm to perform set operations like union, intersection, and subtraction.

• The generic merge algorithm examines and compare the current elements of $A$ and $B$.

• Based upon the outcome of the comparison, it determines if it should copy one or none of the elements $a$ and $b$ into $C$.

• This decision is based upon the particular operation we are performing, i.e. union, intersection or subtraction.

• For example, if our operation is union, we copy the smaller of $a$ and $b$ to $C$ and if $a=b$ then it copies either one (say $a$).

• We define our copy actions in $\text{aIsLess}$, $\text{bothAreEqual}$, and $\text{bIsLess}$.

• Let’s see how this is done ...
Set Operations (cont.)

• For union

```java
public class UnionMerger extends Merger {
    protected void aIsLess(Object a, Object b, Sequence C) {
        C.insertLast(a);
    }
    protected void bothAreEqual(Object a, Object b, Sequence C) {
        C.insertLast(a);
    }
    protected void bIsLess(Object b, Sequence C) {
        C.insertLast(b);
    }
}
```

• For intersect

```java
public class IntersectMerger extends Merger {
    protected void aIsLess(Object a, Object b, Sequence C) {
    }
    protected void bothAreEqual(Object a, Object b, Sequence C) {
        C.insertLast(a);
    }
    protected void bIsLess(Object b, Sequence C) {
    }
}
```
Set Operations (cont.)

• For subtraction

```java
public class SubtractMerger extends Merger {
    protected void aIsLess(Object a, Object b, Sequence C) {
        C.insertLast(a);
    }

    protected void bothAreEqual(Object a, Object b, Sequence C) {
    }

    protected void bIsLess(Object b, Sequence C) {
    }
}
```