Next week we are starting our first **non-linear** data structure – Graphs!

- Reading for next class is LDC Chapter 19.1 - 19.4

- **Linear search** examines each element one at a time, until it is either found or every element has been examined.
- Elements may be in any order (i.e. **does not assume items are sorted**)
- Easy to implement, not very efficient!

```java
// Search specified array using linear search. Returns null if target not
static Comparable linearSearch (Comparable[] data, Comparable target)
Comparable result = null;
int index = 0;

while(result == null && index < data.length) {
    if(data[index].compareTo(target) == 0)
        result = data[index];

    index++;}

return result;
```
**Assume the elements are sorted**, now we can be more efficient

A *binary search* eliminates large parts of the search pool with each comparison

1. Begin searching at the middle
2. If the target isn’t found, if it exists it is in one half or the other
3. Jump to the middle of that half, and begin again

```java
public static Comparable binarySearch(Comparable[] data, Comparable target)
{
    Comparable result = null;
    int first = 0, last = data.length - 1, mid;

    while(result == null && first <= last)
    {
        mid = (first + last) / 2; //determine midpoint
        if (data[mid].compareTo(target) == 0) //found target
        {
            result = data[mid];
        }
        else
        {
            if (data[mid].compareTo(target) > 0) //in first half
                last = mid - 1;
            else
                first = mid + 1;
        }
    }

    return result;
}
```

**Selection Sort** orders values by repetitively putting a particular value into its final position

The algorithm

1. find the smallest value in the list
2. switch it with the value in the first position
3. find the next smallest value in the list
4. switch it with the value in the second position
5. repeat until all values are in their proper places

```java
public static void selectionSort(Comparable[] data)
{
    int min;

    for(int index = 0; index < data.length - 1; index++)
    {
        min = index;
        for(int scan = index + 1; scan < data.length; scan++)
        {
            if(data[scan].compareTo(data[min]) < 0)
                min = scan;
        }

        swap(data, min, index);
    }
}
```
• **Insertion sort** orders a list of values by repetitively inserting a particular value into a sorted subset of the list.

• The algorithm
  1. consider the first item to be a sorted sublist of length 1
  2. insert the second item in the sorted sublist, shifting the first item if needed
  3. insert the third item into the sorted sublist, shifting the other items as needed
  4. repeat until all values inserted into their proper positions

<table>
<thead>
<tr>
<th>3</th>
<th>9</th>
<th>6</th>
<th>1</th>
<th>2</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>9</td>
<td>6</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>6</td>
<td>9</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>1</td>
<td>3</td>
<td>6</td>
<td>9</td>
<td>2</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td>3</td>
<td>6</td>
<td>9</td>
</tr>
</tbody>
</table>

is sorted sublist. Consider 9.

if nothing, insert 9. (3, 9) are sorted. Consider 6.

if 9, insert 6. (3, 6, 9) are sorted. Consider 1.

if 3, 6, 9, insert 1. (1, 3, 6, 9) sorted. Consider 2.

if 3, 6, 9, insert 2. (1, 2, 3, 6, 9) sorted.

```java
public static void insertionSort(Comparable[] data) {
    for(int index = 1; index < data.length; index++) {
        Comparable key = data[index];
        int position = index;

        //shift larger values to the right
        while (position > 0 && data[position-1].compareTo(key) > 0){
            data[position] = data[position-1];
            position--;
        }
        data[position] = key;
    }
}
```

• **Bubble sort** orders a list of values by repetitively comparing neighboring elements and swapping their positions if necessary.

• The algorithm
  1. scan the list, exchanging adjacent elements if they are not in relative order, bubbles highest value to the top
  2. scan the list again, bubbling up the second highest value
  3. repeat until all elements are in proper order

```java
public static void bubbleSort(Comparable[] data){
    int position, scan;

    for(position = data.length - 1; position >= 0; position--){
        for(scan = 0; scan <= position - 1; scan++){
            //compare adjacent elements
            if(data[scan].compareTo(data[scan+1]) > 0)
                swap (data, scan, scan+1);
        }
    }
}
```
- **Merge sort** orders a list of values by recursively dividing the list in half until each sublist has one element, then recombining.

- The algorithm:
  1. divide the list into two roughly equal parts
  2. recursively divide each part in half, continuing until a part contains only one element
  3. merge the two parts into one sorted list
  4. continue to merge parts as the recursion unfolds

```java
public static void mergeSort(Comparable[] data, int min, int max)
{
    if (min < max) {
        int mid = (min + max) / 2;
        mergeSort(data, min, mid);
        mergeSort(data, mid + 1, max);
        merge(data, min, mid, max);
    }
}
```
public static void merge(Comparable[] data, int first, int mid, int last){
    Comparable[] temp = new Comparable[data.length];

    int first1 = first, last1 = mid; //endpoints for 1st subarray
    int first2 = mid+1, last2 = last; //endpoints for 2nd subarray
    int index = first1; //next index open in temp array

    //copy smaller item from each subarray into temp until one of the
    //ays is exhausted
    while(first1 <= last1 && first2 <= last2){
        if(data[first1].compareTo(data[first2]) < 0) {
            temp[index] = data[first1];
            first1++;
        } else {
            temp[index] = data[first2];
            first2++;
        }
        index++;
    }

    //copy remaining elements from first subarray, if any
    while(first1 <= last1){
        temp[index] = data[first1];
        first1++;
        index++;
    }

    //copy remaining elements from second subarray, if any
    while(first2 <= last2){
        temp[index] = data[first2];
        first2++;
        index++;
    }

    //copy merged data into original array
    for(index = first; index <= last; index++){
        data[index] = temp[index];
    }
}

//copy remaining elements from first subarray, if any
while(first1 <= last1){
    temp[index] = data[first1];
    first1++;
    index++;
}

//copy remaining elements from second subarray, if any
while(first2 <= last2){
    temp[index] = data[first2];
    first2++;
    index++;
}

//copy merged data into original array
for(index = first; index <= last; index++){
    data[index] = temp[index];
}

* Quick sort orders a list of values by partitioning the list around one element (the **pivot**), then sorting each partition

* The algorithm
  1. choose one element in the list to be the partition element
  2. organize the elements so that all elements less than the partition element are to the left and all greater are to the right
  3. apply the quick sort algorithm (recursively) to both partitions

* Nice if the quick sort algorithm divides the list **roughly in half**

* Quick sort has two methods
  * quickSort – performs recursive algorithm
  * partition – rearranges elements into two partitions
public static void quickSort(Comparable[] data, int min, int max) {
    int pivot;
    if(min < max){
        pivot = partition(data, min, max);  //make partitions
        quickSort(data, min, pivot-1);  //sort left partition
        quickSort(data,pivot+1, max);  //sort right partition
    }
}

private static int partition(Comparable[] data, int min, int max) {
    Comparable partitionValue = data[min];
    int left = min;
    int right = max;
    while(left < right){
        //search for an element that is > the partition element
        while(data[left].compareTo(partitionValue) <= 0 && left < right
            left++;
        //search for an element that is < the partition element
        while(data[right].compareTo(partitionValue) > 0)
            right--;
        if(left < right)
            swap(data, left, right);
    }
    //move the partition element to its final position
    swap(data, min, right);
    return right;  //will become the pivot
}