Selection Sort:

- Algorithm selectionSort(a, n)
- // Sorts the first n elements of an array a.
  for (index = 0; index < n - 1; index++)
    { indexOfNextSmallest = the index of the smallest value among
      a[index], a[index+1], . . . , a[n-1]
      Interchange the values of a[index] and a[indexOfNextSmallest]
    }

- Iterative method for loop executes n – 1 times
  - For each of n – 1 calls, the indexOfSmallest is invoked, last is n-1, and first
    ranges from 0 to n-2.
  - For each indexOfSmallest, compares last – first times
  - Total operations: (n – 1) + (n – 2) + . . . + 1 = n(n – 1)/2 = O(n²)

- It does not depends on the nature of the data in the array.

Insertion Sort:

- Algorithm insertionSort(a, first, last)
- // Sorts the array elements a[first] through a[last] iteratively.
  for (unsorted = first+1 through last)
    { firstUnsorted = a[unsorted]
      insertInOrder(firstUnsorted, a, first, unsorted-1)
    }

- Algorithm insertInOrder(element, a, begin, end)
  // Inserts element into the sorted array elements a[begin] through a[end].
  index = end
  while ( (index >= begin) and (element < a[index]) )
    { a[index+1] = a[index] // make room
      index - -
    }
  // Assertion: a[index+1] is available.
  a[index+1] = element // insert

- Best time efficiency is O(n) if the array is completely sorted
- Worst time efficiency is O(n²)
- If array is closer to sorted order, less work the insertion sort does, more efficient the sort is
- Insertion sort is acceptable for small array sizes

Merge Sort

- Algorithm mergeSort(a, first, last)
- // Sorts the array elements a[first] through a[last] recursively.
  if (first < last)
    { mid = (first + last)/2
      mergeSort(a, first, mid)
      mergeSort(a, mid+1, last)
Quick Sort

Algorithm quickSort(a, first, last)
// Sorts the array elements a[first] through a[last] recursively.
if (first < last)
    { Choose a pivot
      Partition the array about the pivot
      pivotIndex = index of pivot
      quickSort(a, first, pivotIndex-1) // sort Smaller
      quickSort(a, pivotIndex+1, last) // sort Larger
    }

Quick sort is O(n log n) in the average case
O(n^2) in the worst case due to choice of pivot. Median of three pivot selection can avoid
the worst case.
Worst case can be avoided by careful choice of the pivot
You should know how to do the partition using Median of three pivot selection method

Binary Search

Algorithm binarySearch(a, first, last, desiredItem)
mid = (first + last)/2 // approximate midpoint
if (first > last)
    return false
else if (desiredItem equals a[mid])
    return true
else if (desiredItem < a[mid])
    return binarySearch(a, first, mid-1, desiredItem)
else if (desiredItem > a[mid])
    return binarySearch(a, mid+1, last, desiredItem)

Efficiency of Binary Search

Eliminates about half of the array from consideration after examining the middle element
once.
Counting the maximum number of comparisons under the worst-case
Comparison is made each time the array is divided in half.
First, begin with n items, first division left us with n/2 items, then second division left us
with n/4 items, then third division left with n/8 items, until we get one element.
n/2^k = 1 what is k for the total number of division?
If n is a power of 2, k = log2 n. If not, find a k such that: (k-1)< log2 n <k

Choose a searching method

Use a sequential search to search a chain of linked nodes.
For array of objects, need to know which is applicable.
• Sequential search, must have a method equals
• Binary search,
  • must have a method compareTo
  • and array must be sorted.
<table>
<thead>
<tr>
<th></th>
<th>Best Case</th>
<th>Average Case</th>
<th>Worst Case</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sequential search</td>
<td>O(1)</td>
<td>O(n)</td>
<td>O(n)</td>
</tr>
<tr>
<td>(unsorted data)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Sequential search</td>
<td>O(1)</td>
<td>O(n)</td>
<td>O(n)</td>
</tr>
<tr>
<td>(sorted data)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Binary Search</td>
<td>O(1)</td>
<td>O(log n)</td>
<td>O(log n)</td>
</tr>
<tr>
<td>(sorted array)</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- Iterative search saves time, memory over recursive search.
- Usually, we use iterative version for the sequential search.
- For binary search, coding the binary search recursively is easier than coding it iteratively.
- Since binary search is fast, recursive will not require much additional space for recursive calls.

### Infix-to-Postfix Algorithm

<table>
<thead>
<tr>
<th>Symbol in Infix</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>Operand</td>
<td>Append to end of output expression</td>
</tr>
<tr>
<td>Operator ^</td>
<td>Push ^ onto stack</td>
</tr>
<tr>
<td>Operator +, -, *, or /</td>
<td>Pop operators from stack, append to output expression until stack empty or top has lower precedence than new operator. Then push new operator onto stack</td>
</tr>
<tr>
<td>Open parenthesis</td>
<td>Push ( onto stack, treat it as an operator with the lowest precedence</td>
</tr>
<tr>
<td>Close parenthesis</td>
<td>Pop operators from stack, append to output expression until we pop an open parenthesis. Discard both parentheses.</td>
</tr>
</tbody>
</table>

### Evaluating Postfix Expression

- Save operands until we find the operators that apply to them.
- When see operator, is second operand is the most recently save value. The value saved before it – is the operator’s first operand.
- Push the result into the stack. If we are at the end of expression, the value remains in the stack is the value of the expression.

### A Doubly Linked Implementation of a Deque

- Chain with head reference enables reference of first and then the rest of the nodes
- Tail reference allows reference of last node but not next-to-last
- We need nodes that can reference both
  - Previous node
  - Next node
- For remove action to occur at the end of chain, thus the doubly linked chain

Java code for four operations at both ends of Deque:

```java
public void addToBack( T newEntry) {
    DLNode newNode = new DLNode(lastNode, newEntry, null);
    if( isEmpty())
```

[Incomplete code]

---

```
```
```java
    firstNode = newNode;
    else
        lastNode.setNextNode(newNode);
    lastNode = newNode;
}

public T removeBack() {
    T back = null;
    if(!isEmpty()) {
        back = lastNode.getData();
        lastNode = lastNode.getPreviousNode();
        if(lastNode == null)  
            firstNode = null;
        else
            lastNode.setNextNode(null);
    }
    return back;
}

public void addToFront(T newEntry) {
    DLNode newNode = new DLNode (null, newEntry, firstNode);
    if (isEmpty())
        lastNode = newNode;
    else
        firstNode.setPreviousNode(newNode);
    firstNode = newNode;
}

public T removeFront() {
    T front = null;
    if(!isEmpty()) {
        front = firstNode.getData();
        firstNode = firstNode.getNextNode();
        if(firstNode == null)
            lastNode = null;
        else
            firstNode.setPreviousNode(null);
    }
    return front;
}

What would be the value of queues q1, and q2, and stack s, after the following segment:
s = createStack
q1 = createQueue
q2 = createQueue
enqueue(q1, 5)
enqueue(q1, 6)
enqueue(q1, 9)
enqueue(q1, 0)
enqueue(q1, 7)
enqueue(q1, 5)
enqueue(q1, 0)
enqueue(q1, 2)
enqueue(q1, 6)
1 Loop ( not emptyQueue(q1))
```
1. Dequeue(q1, x)
   2. if(x = 0)
      1. z = 0;
      2. loop(not emptyStack(s))
         1. popStack(s, y)
         2. z = z+y
      3. enqueue (q2, z)
   3. else
      1. pushStack(s, x)

Stack from the top: 6 2
Queue from the head: 20 12

**Keys to Homework 4**

1). Suppose that s and t are empty stacks and a, b, c, and d are objects. What do the stacks contain after the following sequence of operations executes?
   s.push(a);
   s.push(b);
   s.push(c);
   t.push(d);
   t.push(s.pop());
   t.push(s.peek());
   s.push(t.pop());
   t.pop();

The contents of the stacks are shown bottom to top:
   s.push(a); s: a t:
   s.push(b); s: a b t:
   s.push(c); s: a b c t:
   t.push(d); s: a b c t: d
   t.push(s.pop()); s: a b t: d c
   t.push(s.peek()); s: a b t: d c b
   s.push(t.pop()); s: a b b t: d c
   t.pop(); s: a b b t: d

2). Consider the following Java statements:
   ```java
   int n = 4;
   StackInterface<Integer> myStack = new LinkedStack<Integer>();
   while (n > 0)
   {
      myStack.push(n);
      n--;
   } // end while
   int result = 1;
   while (!myStack.isEmpty())
   {
      int integer = myStack.pop();
      result = result * integer;
   } // end while
   System.out.println("result = " + result);
   a. What value is displayed when this code executes?
   b. What mathematical function does the code evaluate?
   a. The value 24 is displayed.
   b. The code evaluates n factorial when n is 4.

3). Convert the infix expression to postfix expression(show your intermediate steps):
   a. a * b / (c - d)
b. \((a - b \cdot c) / (d \cdot e \cdot f + g)\)
c. \(a \div b \cdot (c + (d - e))\)
d. \((a \cdot b \cdot c - d) \cdot e + f \cdot g \cdot h\)

4). Using the algorithm `evaluatePostfix`, given in Segment 21.20, evaluate each of the following postfix expressions. Assume that \(a = 2\), \(b = 3\), \(c = 4\), \(d = 5\), and \(e = 6\).

a. \(a \cdot b \cdot c \cdot d -\)
b. \(a \cdot b \cdot c - d \cdot e \cdot f \cdot g +\)
c. \(a \div b \cdot c \cdot d - e \cdot f \cdot g \cdot h \cdot h\)

d. \(a \cdot b \cdot c \cdot d - e \cdot f \cdot g \cdot h \cdot h\)

5). After the following statements execute, what are the contents of the priority queue?

```
PriorityQueueInterface<String> myPriorityQueue =
    new LinkedPriorityQueue<String>();
myPriorityQueue.add("Jim");
myPriorityQueue.add("Jess");
myPriorityQueue.add("Jill");
myPriorityQueue.add("Jane");
String name = myPriorityQueue.remove();
myPriorityQueue.add(name);
myPriorityQueue.add(myPriorityQueue.peek());
myPriorityQueue.add("Jim");
myPriorityQueue.remove();
```

Initially, `myPriorityQueue` is empty. The following statements affect the priority queue as shown, assuming that priority is interpreted as coming first in a lexicographical ordering (dictionary order):

```
myPriorityQueue.add("Jim"); Jim
myPriorityQueue.add("Jess"); Jess Jim
myPriorityQueue.add("Jill"); Jess Jill Jim
myPriorityQueue.add("Jane"); Jane Jess Jill Jim
String name = myPriorityQueue.remove(); Jane Jill Jim
myPriorityQueue.add(name); Jane Jess Jill Jim
myPriorityQueue.add(myPriorityQueue.peek()); Jane Jane Jess Jill Jim
myPriorityQueue.add("Jim"); Jane Jane Jess Jill Jim Jim
myPriorityQueue.remove(); Jane Jane Jill Jim Jim
```

6). Assume that `customerLine` is an instance of the class `WaitLine`:
The call `customerLine.simulate(15, 0.5, 5)` produces the following random events:
Customer 1 enters the line at time 6 with a transaction time of 3.
Customer 2 enters the line at time 8 with a transaction time of 3.
Customer 3 enters the line at time 10 with a transaction time of 1.
Customer 4 enters the line at time 11 with a transaction time of 5.
During the simulation, how many customers are served, and what is their average waiting time?
Four customers are served. The average waiting time will be \((0 + 1 + 2 + 2) / 4 = 1.25\).

**Keys to homework 3**

.3. Consider the method `quickSort`, that sorts an array of objects into ascending order by using a quick sort.
Suppose that 80 90 70 85 60 40 50 95 represents an array of `Integer` objects.

a. What does the array look like after `quickSort` partitions it for the first time?
b. The pivot is now between two subarrays called Smaller and Larger. Will the position of this particular element change during subsequent steps of the sort? Why or why not?
c. What recursive call to quickSort occurs next?

\[ \begin{align*}
a. & \quad 80 \ 90 \ 70 \ 85 \ 60 \ 40 \ 50 \ 95 \\
   & \quad \text{first} = 0; \ \text{last} = 7; \ \text{mid} = 3 \\
   & \quad \text{sortFirstMiddleLast makes no changes.} \\
   & \quad \text{Swap middle and next to last elements:} \\
   & \quad 80 \ 90 \ 70 \ 50 \ 60 \ 40 \ 85 \ 95 \\
   & \quad \text{Do partitioning steps:} \\
   & \quad 80 \ 40 \ 70 \ 50 \ 60 \ 90 \ 85 \ 95 \\
   & \quad \text{Swap pivot into correct position:} \\
   & \quad 80 \ 40 \ 70 \ 50 \ 60 \ 85 \ 90 \ 95 \\
   & \quad \text{The pivot 85 is at index 5.} \\
\end{align*} \]

b. The position of the pivot will never change during subsequent steps because the recursive calls to quickSort are quickSort(a, first, pivotIndex - 1) and quickSort(a, pivotIndex + 1, last). They do not include the position of the pivot.
c. The next recursive call to quickSort will be quickSort(a, first, pivotIndex - 1).

4. Consider the merge step of the merge sort.
   a. What is the minimum number of comparisons needed to merge two subarrays each of size \( n/2 \)?
   b. Give a recurrence relation that counts the number of comparisons made in the best case.
   c. Make an educated guess at the solution to the recurrence relation.
   a. The minimum number of comparisons occurs when all the values in one half are smaller than all the values in the other half. Only \( n/2 \) comparisons are required to exhaust a subarray. Once that happens, the remaining values can just be merged without doing a comparison.
   b. \( C_B(n) = 2 \ C_B(n / 2) + n / 2 \)
   c. This recurrence has the solution \( C_B(n) = (n / 2) \lg n \)

5. Trace the method binarySearch, when searching for 4 in the following array of values:
\[ 5 \ 8 \ 10 \ 13 \ 15 \ 20 \ 22 \ 26 \ 30 \ 31 \ 34 \ 40 \]
Repeat the trace when searching for 34.
\[ \text{binarySearch(a, 0, 11, 4)} \]
\[ \text{mid} = 11/2 = 5 \]
\[ 4 \text{ is less than 20} \]
\[ \text{binarySearch(a, 0, 4, 4)} \]
\[ \text{mid} = 4/2 = 2 \]
\[ 4 \text{ is less than 10} \]
\[ \text{binarySearch(a, 0, 1, 4)} \]
\[ \text{mid} = 1/2 = 0 \]
\[ 4 \text{ is less than 5} \]
\[ \text{binarySearch(a, 0, -1, 4)} \]
\[ \text{first is not greater than last} \]
\[ \text{return false} \]

\[ \text{binarySearch(a, 0, 11, 34)} \]
\[ \text{mid} = 11/2 = 5 \]
\[ 34 \text{ is greater than 20} \]
\[ \text{binarySearch(a, 6, 11, 34)} \]
\[ \text{mid} = 17/2 = 8 \]
\[ 34 \text{ is greater than 30} \]
\[ \text{binarySearch(a, 9, 11, 34)} \]
\[ \text{mid} = 20/2 = 10 \]
\[ 34 \text{ is equal to 34} \]
\[ \text{return true} \]