1. Write a few lines of Java code to declare and allocate a 3-dimensional array $A$ with $m$ rows, $n$ columns, and $p$ pages. Use any dimension ordering you prefer. What dimension order does your code use? I.e., in the expression $A[x][y][z]$, which dimensions do $x$, $y$, and $z$ refer to?

**Answer:** This example uses the dimension order: pages, columns, rows. **Method #1** is discussed below: In C++ we would use:

```cpp
// Method # 1 -- C++
int *** A ;

// Assume values of m, n, and p are given.
A = new int ** [p] ;
for ( int k = 0 ; k < p ; k++ ) {
    A[k] = new int * [ n ] ;
    for ( int j = 0 ; j < n ; j++ ) {
        A[k][j] = new int [ m ] ;
    }
}
```

In Java we can do the exact same allocation using slightly different syntax:

```java
// Method # 1 -- Java
int [] [] [] A ;
A = new int [p][n][] ;
for ( int k = 0 ; k < p ; k++ ) {
    A[k] = new int [n][] ;
    for ( int j = 0 ; j < n ; j++ ) {
        A[k][j] = new int [m] ;
    }
}
```

This kind of allocation is used so frequently in Java, that there is a short-cut notation:

```java
// Method # 2 -- Java
int [] [] [] A ;
A = new int [p][n][m] ;
```

The expression $A[x][y][z]$, refers to page $x$, column $y$, and row $z$. 
2. Draw a picture illustrating the dimension ordering you used in problem 1.

Answer:

![Dimension Ordering Diagram]

3. The following diagram illustrates a doubly linked list.

(a) Describe (in detail) the operation of removing an item from the doubly linked list. What special cases must be considered?

Answer: Given a position \( p \) on a linked list (i.e., a reference) there are four special cases to consider:

- If the object (reference) \( p \) is only item on the list.
- If the object (reference) \( p \) is first, but not the only.
- If the object (reference) \( p \) is last, but not the only.
- If the object (reference) \( p \) is somewhere in the middle.

Let us assume that we are given a reference \( p \) to the list cell to be removed from the list. If we prefer, we can check for some of the erroneous cases, i.e., where \( p \) does not point to a cell on the list. For example, if \( \text{head} == \text{tail} \), there is only one element on the list. Then, if \( \text{if} (\ p \ != \text{head} \) ), it must be that the input \( p \) is inappropriate for the given list.

Normally, we require that \( p \) is a pointer (reference) to a valid cell on the list. We state this requirement as a pre-condition for the method that removes a cell from the list. Under that pre-condition, we do not check for list corruption.

We consider four cases in detail:
Case I: If `head == tail`, then the list must be of length one. We can delete this single cell using the following statements:

```plaintext
head = null ;
tail = null ;
```

Case II: If `head != tail` and `p.prev == null`, then the list must be longer than length one and `p` must point to the first cell on the list. We can delete the first cell on the list using the following statements:

```plaintext
head = p.next ;
head.prev = null ;
```

Case III: If `head != tail` and `p.next == null`, then the list must be longer than length one and `p` must point to the last cell on the list. We can delete the last cell on the list using the following statements:

```plaintext
tail = p.prev ;
tail->next = null ;
```

Case IV: If `p.prev != null`, and `p.next != null`, then the list must be longer than length one and `p` must point to somewhere in the middle of the list. We use a temporary pointer to update the appropriate links in the previous and next cells.

```plaintext
listcell temp ;
temp = p.prev ; // Get a reference to the previous cell.
temp.next = p.next ;

temp = p->next ; // Get a reference to the next cell.
temp.prev = p->prev ;
```


**Answer:**

Input: A set of `n` observations `S = [s_0, s_1, ... s_{n−1}]` and a positive integer `K`  
Output: A set of `K` clusters which splits `S` into disjoint sets.

Method:

Each observation is randomly assigned to some cluster  
Let `C_0, C_1, ..., C_{K−1}` denote the `K` initial clusters.  
Compute the initial means, denoted `m_0, m_1, ..., m_{K−1}`

```plaintext
do {
    For 0 ≤ i < n  {
        Assign observation `s_i` to cluster `C_j` whenever `s_i` is closer to mean `m_j` than to any other mean  
    }
    For 0 ≤ j < K  {
        Let `m_j` be the average of all observations in cluster `C_j`
    }
}while none of the clusters change.
```
5. Suppose you have three files: `main.java`, `util.java`, `fun.java` containing the classes `main`, `util`, and `fun` respectively. The main program uses objects defined in `util.java` and also in `fun.java`. The methods in `fun.java` use objects defined in `util.java`. Write a Makefile to compile the parts of this program. Include a target named `run` which will run your program.

**Answer:**

```
# Makefile

main.class: main.java util.class fun.class
    javac main.java

util.class: util.java
    javac util.java

fun.class: fun.java util.class
    javac fun.java

clean:
    /bin/rm -f main.class util.class fun.class
```

6. Suppose the following partial implementation of a class for a queue of pairs of integers. Give an implementation of methods `enqueue()` and `dequeue()`. If `enqueue()` is invoked on an a full queue, issue an error message. If `dequeue()` is invoked on an empty queue, then issue an error message and return a null object reference.

**Answer:** (complete implementation)

```java
class pair {
    int a;
    int b;
    pair( int x, int y ) { a = x; b = y; }
}

class queue {
    final int length = 25;
    pair[] Q;
    int first;
    int next;

    // -----------------------------------------------
    queue( ) {
        Q = new pair[ length ];
        first = 0;
        next = 0;
    }
```
```java
// -------------------------------
void enqueue( pair p )
{
    if ( ((next + 1) % length) == first ) {
        System.out.println( "Queue is full." );
    }
    else {
        Q[ next ] = p ;
        next = (next + 1) % length ;
    }
}

// -------------------------------
pair dequeue( )
{
    if ( next == first ) {
        System.out.println( "Queue is empty." );
        return null ;
    }
    else {
        pair temp = Q[ first ] ;
        first = (first + 1) % length ;
        return temp ;
    }
}
} // end class queue

If stored in column-major order in a 1-D array, it would appear as:

<table>
<thead>
<tr>
<th>2</th>
<th>3</th>
<th>5</th>
<th>7</th>
<th>11</th>
</tr>
</thead>
<tbody>
<tr>
<td>13</td>
<td>17</td>
<td>19</td>
<td>23</td>
<td>29</td>
</tr>
<tr>
<td>31</td>
<td>37</td>
<td>41</td>
<td>43</td>
<td>47</td>
</tr>
</tbody>
</table>

(a) Assume an array \( A \) has \( M \) rows and \( N \) columns. Let \( i \) denote a row index and let \( j \) denote a column index. Give an expression using \( i \) and \( j \) which maps (computes) a position in the 1-D array corresponding to row \( i \) and column \( j \) in the 2-D array. Use column major order as illustrated in the example above.

**Answer:** \( j \times M + i \)

(b) Given a position \( p \) in a 1-D array, give expressions which compute the row \( i \) and column \( j \) indices in a 2-D array stored in column major order.

**Answer:** \( i = p \% M ; \quad j = ( p - i )/M ; \)

7. Describe the technique known as “polynomial hashing”.

**Answer:** A character string can be considered as a sequence of ASCII values: \( a = [a_0, a_1, a_2, ..., a_{n-1}] \). We define a polynomial using the \( a_i \) as coefficients. I.e.,

\[
p_a(x) = a_0 + a_1 x + a_2 x^2 + a_3 x^3 + ... + a_{n-1} x^{n-1}
\]

(1)

The polynomial in equation (1) is used to define a hash function as follows:

\[
h(a) = p_a(x) \mod N
\]

(2)

where \( x \) is a small prime number (e.g., 7, 11, 13) and \( N \) is the table size.

8. In the context of open addressing for hash tables, what is **double hashing**?

**Answer:** Double hashing refers to the use of two hash functions \( h_1() \) and \( h_2() \) to define the probe sequence;

\[
( h_1(a) + ih_2(a) ) \mod N
\]

(3)

where \( 0 \leq i < N \) and \( N \) is the table size.

Double hashing is effective because two keys that collide under the primary hash function \( h_1 \) are very unlikely to also collide under the secondary hash function \( h_2 \). Therefore, two keys that collide trace different probe sequences. Tracing different probe sequences avoids clustering.

9. What is the implication of open addressing for the **delete** operation?

**Answer:** A delete operation is impractical when using open addressing. The reason relates to the way in which the **search** operation is performed. We use the hash function to find a starting point for the search. If the target key is not found at the starting point, then we follow the probe sequence until either 1) the target key is found, or, 2) an empty table position is found. If we encounter an empty table position during our search, we conclude that the target key is not in the table.

A natural way to implement delete, is to change the array entry where the target key is found to "empty". The problem with doing that is that we may have created an empty array position along the search path of a different key. In effect, changing the target key’s table entry to “empty” also deletes any additional keys which are “downstream” in the probe sequence.

10. The following is a diagram of a 3-D array in pages-rows-columns order. Suppose there are \( M \) rows, \( N \) columns and \( P \) pages. Give an expression that maps a 3-dimensional index to a 1-D array. Use the index variables \( i, j, \) and \( k \) to indicate row \( i \), column \( j \), and page \( k \).
Answer:

Using the indices $i$, $j$, and $k$ as suggested above, the 3-D structure should be indexed $A[k][i][j]$.

For the 1-D structure, we apply the same principal of counting the number of preceding elements as we used in the 2-D case. Let $L$ denote the 1-D position.

For page $k$, there are $k$ full pages preceding the start of page $k$, with each page containing $M \times N$ elements. In addition, for row $i$ there are $i$ full rows preceding the start of row $i$, where each row contains $N$ elements. Putting these ideas together, we have:

$L = kMN + iN + j$