1. Starting with an empty binary heap using a full-tree implementation:
   A. Insert the integers 17, 3, 4, 7, 2, 9, 11, 5, 19, 14, 20, 18 in that order, and after each step, show the heap array.
   B. Do DELETEMINs until the heap is empty, and show the array after each step.

This is a straightforward use of the algorithm presented in class.
2. Suppose that heap data structure using a full binary tree was represented with pointers rather than stored in an array as presented in class; that is, each vertex has a parent pointer, a left child pointer, and a right child pointer. And assume we are still using the same algorithm for INSERT and DELETEMIN, except now, at the low level, movement around the tree is via pointers rather than arithmetic on array indices (and we can no longer simply increment or decrement nextRB). Describe in English and give pseudo-code for how to maintain a pointer to the RB leaf; that is how to find the next or previous leaf in level order when an INSERT or DELETEMIN operation requires insertion or deletion of the RB leaf. Analyze the time used.

Although it may often be easy (e.g., when a leaf to be deleted has a right sibling), in general, to find the next or previous leaf in level order, we can follow parent links up to the lowest common ancestor (which could be the root in the worst case), and walk back down via left-child links.

Here is example pseudo-code to add a new RB leaf; removing the RB leaf works similarly in the other direction. Note that the terms "left" and "right" are being used as the figure is viewed (e.g., the RB leaf attached with the dashed line in the figure is a "left" child).

\[
p = \text{RB leaf} \\
\text{while } p \neq \text{root and } p \text{ is a right child do } p = \text{PARENT}(p) \\
\text{if } p \neq \text{root then begin} \\
\quad p := \text{PARENT}(p) \\
\quad \text{if } \text{RCHILD}(p) \neq \text{nil then } p := \text{RCHILD}(p) \\
\text{end} \\
\text{while } p \text{ has two children do } p := \text{LCHILD}(p) \\
\text{if } \text{LCHILD}(p) := \text{nil} \\
\quad \text{then make the new RB leaf the left child of } p \\
\quad \text{else make the new RB leaf the right child of } p
\]

Since we do at most a walk up to the root and back down to a leaf, a distance of \(O(\log(n))\), the \(O(\log(n))\) time of the INSERT operation is not changed by more than a constant factor.
3. Recall the recursive Merge Sort algorithm presented in class, where the presentation was at a relatively high level that did not address how the two lists were represented. Suppose that the list of \( n \) elements to be sorted was in an array \( A[0] \ldots A[n-1] \), and we have available an additional array \( B[0] \ldots B[n-1] \) that can be used for temporary storage. Describe in English and give pseudo-code to implement Merge Sort so that the input is in \( A \), the algorithm concludes with the sorted data in \( A \), and only \( O(1) \) space is used in addition to the space for \( A \) and \( B \) and the space used "behind the scenes" to implement the recursion. Include an explanation of why the additional space used to implement the recursion is only \( O(\log(n)) \), and hence, at least for large \( n \), the only significant space used in addition to \( A \) is for \( B \).

To sort \( A \), we call \( \text{SORT}(0, n-1) \), which makes use of the \( \text{MERGE} \) procedure.

**Sort:** To sort \( A[i]...A[j] \), \( \text{SORT} \) computes the midpoint \( m \) between \( i \) and \( j \) by doing \( m := \lfloor (i+j)/2 \rfloor \). If \( A[i]...A[j] \) is an odd number of elements (i.e., \( j-i+1 \) is odd), one more element is in the left half \( A[i]...A[m] \) than the right half \( A[m+1]...A[j] \); otherwise, the same number of elements are in the two halves (i.e., \( (j-i+1)/2 \) elements each).

```
procedure \text{SORT}(i,j)
  if \( i\lt j \) then begin
    \( m := \lfloor (i+j)/2 \rfloor \)
    \( \text{SORT}(i,m) \)
    \( \text{SORT}(m+1,j) \)
    \( \text{MERGE}(i,j,m) \)
  end
end
```

**Merge:** To merge \( A[i]...A[m] \) with \( A[m+1]...A[j] \), \( \text{MERGE} \) places items in sorted order into \( B[i]...B[j] \) and then copies back into \( A[i]...A[j] \).

```
procedure \text{MERGE}(i,j,m)
  \( x := i \)
  \( y := m+1 \)
  \( z := i \)
  while \( z \leq j \) do begin
    if \( y \gt j \) then begin
      \( B[z] := A[x]; x := x+1 \) end
    else if \( x \gt m \) then begin
      \( B[z] := A[y]; y := y+1 \) end
    else if \( A[x] \lt A[y] \) then begin
      \( B[z] := A[x]; x := x+1 \) end
    else begin
      \( B[z] := A[y]; y := y+1 \) end
    end
    \( z := z+1 \)
  end
  for \( k := i \) to \( j \) do \( A[k] := B[k] \)
end
```

Since each recursive call is on a range half the size, and the second call happens only after the first completes, the recursion stack can never be more than \( \log(n) \) deep. Beyond that, only \( O(1) \) space and the space for \( B \) is used in addition to the space for \( A \).
4. Consider the following simpler version of \textit{PARTITION} for quick sort:

\begin{verbatim}
function simplePARTITION(i,j)
    x := i+1
    y := j
    while x ≤ y do
        if A[x] ≤ A[i] then x := x+1
        else if A[y] > A[i] then y := y−1
        else exchange A[x] and A[y]
    end
    if i < y then exchange A[i] and A[y]
    return (y−1, y+1)
end
\end{verbatim}

A. Explain why it correctly partitions.

B. Explain why using this version of partitioning does not change the asymptotic running time of partitioning, and discuss how performance would compare in practice.

A. The second else statement can only be executed if the \textit{if} and first \textit{else} fail, so the exchange can only take place if \( x ≤ y \) (the condition of the while loop), \( A[x] > A[i] \), and \( A[y] ≤ A[i] \), which will ensure that items \( i \) through \( x \) are \( ≤ A[i] \) and items \( y \) through \( j \) are \( > A[i] \). After the while loop terminates, \( y = x−1 \) and \( A[y] ≤ A[i] \), and so the final exchange leaves \( A[i] \ldots A[y−1] \leq A[i] \) and \( A[y+1] \ldots A[j] > A[i] \).

B. The quantity \( y−x \) is reduced at least every other iteration of the while loop (since if an exchange occurs, it will not occur on the next iteration), and hence the while loop terminates in linear time (i.e., in \( O(j−i) \) time), and so the time for this version of partition only differs from that of the standard method by a constant factor.

The standard version has more complicated flow control (nested while loops) but this version may perform some comparisons of array elements more than once (the first \textit{if} statement). The number of exchanges is the same in either case (and, in fact, if there are no duplicates, the sequence of exchanges is identical). So in practice, the running time would likely be similar, with the difference depending on machine dependent times such as the time to access an element of \( A \) versus an index like \( x \) or \( y \).