1. What is the so-called “binary search property” for binary search trees?

Answer: For every node in the tree \( N \), the key data in the left sub-tree is less than or equal to the key data in node \( N \). Also the key data in the right sub-tree is greater than or equal to the key data in node \( N \).

2. Consider the binary search tree in Figure 1. Draw a sequence of diagrams and describe how to delete 30.

![Figure 1: Example Binary Search Tree](image-url)

Answer:

**Step 1**

To delete node here

**Step 2**

Find replacement node

Copy data up
3. Consider the partial Java implementation of a binary search tree class. Complete the `insert()` method.

```java
// TREE NODES
class tree_node {
    int tdata;
    tree_node left;
    tree_node right;

    tree_node(int d) {
        tdata = d;
        left = null;
        right = null;
    }

    // Other methods here as needed.
}
}
```

```java
// BINARY SEARCH TREES
class bst {
    private tree_node root;

    private tree_node recursive_insert(tree_node t, int d) {
        // Your implementation goes here.
    }
}
```
public void insert( int d )
{
    root = recursive_insert( root, d ) ;
}

private tree_node recursive_insert( tree_node t, int d )
{
    if ( t == null ) {
        t = new tree_node( d ) ;
    } else {
        if ( d < t.tdata ) {
            t.left = recursive_insert( t.left, d ) ;
        } else if ( d > t.tdata ) {
            t.right = recursive_insert( t.right, d ) ;
        }
        //
        // Note: If (d == t.tdata) the data is already in the
        // tree. In this version, we do not add duplicates.
    }
    return t ;
}

4. What can go “wrong” in building a binary search tree that motivates height-balanced trees?

Answer: If data items are already in sorted order as they are inserted in the tree, the result
will be a “tree” with only right sub-trees. The left links are never used. In this situation,
searching a tree with \( n \) data items takes \( O(n) \) time instead of the (much faster) \( O(\log n) \) time
when the tree is reasonably balanced.

5. Consider the newly out of balance AVL tree shown in Figure 2. Draw a diagram showing how
to re-balance the tree.

Answer: We re-balance the tree by a single clockwise rotation about the unbalanced node \( x \).
After the rotation, balance is restored. Also, notice that the rotation operation preserves the binary search property.

6. Consider the newly out of balance AVL tree shown in Figure 3. Draw a sequence of diagrams showing how to re-balance the tree.

Figure 3: AVL Tree #2 needing re-balancing

**Answer:** In this case, we need to use a double rotation. First, we expand the sub-tree B.
The double rotation begins with a counter-clockwise rotation about node y. This operation makes the imbalance worse, but it sets up a configuration in which a clockwise rotation about the root will restore balance. We illustrate it in the following diagrams.

First Rotation:

After the counter-clockwise rotation about y, we have:

The next rotation is clockwise about x:
After the second rotation, balance is restored.

Consider the AVL tree shown in figure 4. Draw a sequence of diagrams illustrating how to insert 6 into the tree and restore the HB[1] property. Show the point where the tree is re-balanced. Does it need a single rotation or a double rotation?

Answer: The diagram below (left) shows the tree after inserting 6. The heights of each sub-tree is shown in blue.
Balance is restored by a single rotation about the node containing 19. The re-balanced tree is shown in the diagram below (right).

8. True or false: AVL trees also have the binary search property.

Answer: True.

9. True or false: The binary search tree illustrated in Figure 1 is also an AVL tree.

Answer: False.

10. Refer to the handouts for the depth-first search of a graph and for the breadth-first search of a tree. Design an algorithm to do a breadth-first search of a graph.

Answer: An algorithm for breadth-first search is given below:

Input: Undirected Graph $G = (V, E)$

```
bfs(G)
    for all $v \in V$ do
        visited[v] = false;
    for all $v \in V$ do
        if (not visited[v]) {
            search(G, v)
        }
```
**Input:** Undirected Graph \( G = (V, E) \) and starting node \( v \)

```plaintext
search( G, v )
  Q.enqueue( v ) ;
  while ( Q.not_empty() ) {
    x = Q.dequeue() ;
    visited[x] = true ;
    for each edge \((x, w) \in E\) do { 
      if ( not visited[w] ) {
        Q.enqueue( w ) ;
      } // end if ...
    } // end for ...
  } // end while ...
```

Notice that a visited node is never put onto the queue. Also, every time around the loop, a
previously unvisited node is marked as visited. There are finitely many nodes in the graph,
therefore the queue must eventually be empty and the loop finishes.

11. Describe two data structures to represent an undirected graph \( G = (V, E) \).

**Answer:**

**Adjacency Matrix**

An adjacency matrix, \( A \), for a directed (or undirected) graph, \( G \), with \( n \) nodes is an \( n \times n \) matrix:

\[
A_{i,j} = \begin{cases} 
  1 & \text{if edge } (i, j) \in E \\
  0 & \text{otherwise}
\end{cases}
\]

**Adjacency Lists**

Adjacency lists use less memory for sparse graphs. Given a graph \( G = (V, E) \), we keep a list
of connected vertices for each vertex. The usual structure is to maintain an array of lists. The
length of the array is the number of nodes, \(|V|\).

12. Design an algorithm to detect if a directed graph has a cycle.

**Answer:** This is a tricky problem. We might try to modify a depth-first search and declare
a cycle when we encounter a marked node while searching neighbors. This would be correct
for an undirected graph, but it is not correct for a directed graph. For example:

```
A depth first search will reach \( v_3 \) by the path \( v_0 \rightarrow v_1 \rightarrow v_3 \) and mark it. The depth first search
then "backs up" to \( v_0 \) and encounters \( v_3 \) again via the path \( v_0 \rightarrow v_2 \rightarrow v_3 \). We encounter
node \( v_3 \) twice, but there is no cycle.

Also, a modified breadth-first search does not work with a directed graph.

However, we have one more trick: a directed graph has a cycle if and only if there exists a
back edge. Recall the classification of edges using their pre- and post- numbers. This leads us
to the following easy algorithm.
**Input:** A directed Graph $G = (V, E)$

**Output:** Returns **true** is a cycle exists, **false** otherwise.

**Method:**

```cpp
bool has_cycle( G )
    1. Perform a depth first search of $G$ and assign pre- and post-numbers to each node.
    2. Classify each edge in $E$ using the pre- and post- numbers from step 1.
       If a back edge is found, return **true**.
    3. If no back edge was found in step 2, return **false**.
```

13. Given the directed acyclic graph (DAG) in Figure 5.

![Directed Acyclic Graph](image)

**Figure 5: Example Directed Acyclic Graph**

(a) Perform a depth-first search and assign pre- and post- numbers.

**Answer:**

![Post and Pre Numbers](image)
(b) Give a topological sort of the DAG.

Answer:

\[ v_4, v_3, v_5, v_7, v_1, v_0, v_2, v_6, v_8 \]

(c) Draw a diagram illustrating an adjacency list for the graph in Figure 5.

Answer:

14. Give a “Big O” upper bound for the time taken by the following code segment.

```
t = 0;
for ( i = 1 ; i <= n ; i++ ){
    for ( j = 1 ; j <= i ; j++ ) {
        t = t + i * j;
    }
}
```

Answer: Given in class.

15. Suppose you have a collection of records, each containing name, phone number, date of birth, and additional text information. You can assume that each type of information is stored as a character string. Draw a diagram illustrating a collection of such records. Using binary search tree, include appropriate structures so that the collection of records can be searched either by name, or by phone number.

Answer: We illustrate the two-index structure with example data:

<table>
<thead>
<tr>
<th></th>
<th>Joe</th>
<th>Betty</th>
<th>Bill</th>
<th>Wilma</th>
<th>111-1111</th>
<th>222-2222</th>
<th>333-3333</th>
<th>444-4444</th>
</tr>
</thead>
<tbody>
<tr>
<td>Name</td>
<td>Joe</td>
<td>Betty</td>
<td>Bill</td>
<td>Wilma</td>
<td>111-1111</td>
<td>222-2222</td>
<td>333-3333</td>
<td>444-4444</td>
</tr>
<tr>
<td>Phone</td>
<td>111-1111</td>
<td>222-2222</td>
<td>333-3333</td>
<td>444-4444</td>
<td>111-1111</td>
<td>222-2222</td>
<td>333-3333</td>
<td>444-4444</td>
</tr>
<tr>
<td>Birth</td>
<td>6/1/95</td>
<td>8/12/97</td>
<td>5/19/89</td>
<td>10/3/62</td>
<td>6/1/95</td>
<td>8/12/97</td>
<td>5/19/89</td>
<td>10/3/62</td>
</tr>
<tr>
<td>Notes</td>
<td>Joe's poem</td>
<td>Betty's thesis</td>
<td>Bill's map</td>
<td>Wilma's song</td>
<td>Joe's poem</td>
<td>Betty's thesis</td>
<td>Bill's map</td>
<td>Wilma's song</td>
</tr>
</tbody>
</table>

The names and phone numbers have the following hash values (with table size 5).
16. What is the heap property (for max-heaps)?

**Answer:** The heap property states that the data in every node is greater or equal to the data in both child nodes. This type of heap is often called a max heap. Naturally, we may define a corresponding min heap to be a binary tree in which the data in every node is less than or equal to the data in both child nodes. Both types of heaps are often implemented as an array stored in consecutive memory locations.
17. Consider the heap and its embedding in an array as shown in Figure 6. Given a position $p$ in the array in Figure 6, give an expression for each of the following new positions in the array.

(a) The parent node of $p$
(b) The left child node of $p$
(c) The right child node of $p$

**Answer:**

(a) The parent node of $p$ is in position $\left\lfloor \frac{p-1}{2} \right\rfloor$.
(b) The left child node of $p$ is in position $2p + 1$.
(c) The right child node of $p$ is in position $2p + 2$.

18. Consider the heap shown in problem 17. Draw a sequence of diagrams and give a few sentences of explanation of what happens when we delete the value 90 from the tree.

**Answer:**

1) Copy last data to root

2) Delete last node

3) Exchange with largest child

4) Exchange complete
19. Give pseudocode for the sorting method known as heapsort. Assume a function `heapify( A, n, j )` which performs the heapify operation on array A, with length n, staring in position j.

```c
void heap_sort( int A[], int n )
{
    k = n/2 - 1 ; // Highest index internal node.

    for ( j = k ; j > 0 ; j-- ) { // Build the heap.
        heapify( A, n, j ) ;
    }

    for ( i = n-1 ; i > 0 ; i-- ) {
        heapify( a, 0, i+1 ) ; // Restore the heap.
        temp = a[0] ; // Exchange.
        a[0] = a[i] ;
        a[i] = temp ;
    }
}
```

20. Tries:

(a) Draw a trie for the words: _saw, see, seen, top, tag, tap, tape, tip, _and _top._

   Answer:
(b) How can you quantify the worst-case time taken to search for a word in a trie?

**Answer:** The time to search is proportional to the length of the longest word.

21. Huffman codes:

(a) Construct a Huffman code for the following symbols and probabilities. Draw the Huffman tree as part of your construction.

<table>
<thead>
<tr>
<th>Symbol</th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>E</th>
</tr>
</thead>
<tbody>
<tr>
<td>Probability</td>
<td>0.35</td>
<td>0.22</td>
<td>0.18</td>
<td>0.15</td>
<td>0.10</td>
</tr>
</tbody>
</table>

**Answer:** A Huffman tree is shown below.

The corresponding codes are shown below.

- A 00
- B 10
- C 11
- D 010
- E 011

(b) Compute the expected number of bits to encode one letter using your code from part 21a.

**Answer:**

\[ 2 \times 0.35 + 2 \times 0.22 + 2 \times 0.18 + 3 \times 0.15 + 3 \times 0.10 = 2.25 \]