

## Section Solutions 8

---

### Problem One: Breadth-First Search

```
void breadthFirstSearch(Node* root) {
    Queue<Node*> worklist;
    worklist.enqueue(root);

    while (!worklist.isEmpty()) {
        Node* curr = worklist.dequeue();
        if (curr != NULL) {
            cout << curr->value << endl;

            worklist.enqueue(curr->left);
            worklist.enqueue(curr->right);
        }
    }
}
```

Note that this function does not need to keep track of a set of visited nodes, since there are no cycles in a binary search tree.

Given the tree in the section handout, the function will output the nodes in this order:

f, b, j, a, d, h, k, c, e, g, i

This function will only list off the nodes in a BST in sorted order if the tree is degenerate and each node only has either no children or a right child.

### Problem Two: Functions as Data

```
void breadthFirstSearch(Node* root, void processFn(Node* curr) {
    Queue<Node*> worklist;
    worklist.enqueue(root);

    while (!worklist.isEmpty()) {
        Node* curr = worklist.dequeue();
        if (curr != NULL) {
            processFn(curr);

            worklist.enqueue(curr->left);
            worklist.enqueue(curr->right);
        }
    }
}
```

### Problem Three: Depth-First Search

```
Vector<string> depthFirstSearch(string start, string end,
                                Vector<string> edgeFunction(string nodeName)) {
    Map<string, string> parentMap;
    if (dfsRec(start, start, end, parentMap, edgeFunction)) {
        return flattenPath(parentMap, end);
    }

    /* Otherwise, return an empty Vector. */
    return Vector<string>();
}

bool dfsRec(string curr, string parent, string end,
            Map<string, string>& parentMap,
            Vector<string> edgeFunction(string nodeName)) {
    if (parentMap.containsKey(curr)) return false;
    parentMap[curr] = parent;

    if (curr == end) return true;

    foreach (string child in edgeFunction(curr)) {
        if (dfsRec(child, curr, end, parentMap, edgeFunction)) {
            return true;
        }
    }

    return false;
}

Vector<string> flattenPath(Map<string, string>& parentMap, string endpoint) {
    /* The parent map traces the path back to the starting node, so we have to
     * reverse it before returning it. To do so, we'll throw everything in a Stack
     * before putting it into the Vector.
     */
    Stack<string> reverseResult;
    while (true) {
        reverseResult += endpoint;
        if (parentMap[endpoint] == endpoint) break;

        endpoint = parentMap[endpoint];
    }

    Vector<string> result;
    while (!reverseResult.isEmpty()) {
        result += reverseResult.pop();
    }
    return result;
}
```