2015/16 Exam

This is just an example of how I would answer this exam – I have no certainty about how this would score. If anyone notices any problems with these solutions, though, please point them out to me, and I'll update them.

Question 2

(i)

A Binary Search Tree is a binary tree in which, for all nodes, the leftchild is less than the parent and the right child is greater than the parent, according to an ordering.

(ii)

Pseudocode

```
if there is a left child
    print left child (with this function)
print this node's element
if there is a right child
    print right child (with this function)
```

Python

```
def __str__():
    if node.leftchild is not None:
        print(node.leftchild)
    print(node.element)
    if node.rightchild is not None:
        print(node.rightchild)

print(root_node)
```

(iii)

(a) – adding a new element

Text

- Start at the root node.
- Repeat this process until you can no longer go the direction you're trying:
    - At each node, compare that node's element to the new element.
    - If the new element is less than the node's element, go to the node's left child.
    - If the new element is greater than the node's element, go to the node's right child.
    - If the new element is equal to the node's element, exit the function.
- Put the new element in a new node
    - If the new element is less than the current node's element, make the new node the left child of the current node.
    - If the new element is greater than the current node's element, make the new node the right child of the current node.

Python

```
def add(node=root, element):
    if element < node.leftchild:
        if node.leftchild is not None:
            add(node.leftchild, element)
        else:
            node.leftchild = DLLNode(element)
    elif element > node.rightchild:
        if node.rightchild is not None:
            add(node.rightchild, element)
        else:
            node.rightchild = DLLNode(element)
```

(b) - removing a node

Text

- Start at the root node, and compare the element to be removed to the root element.
    - If it is less than the root, go to the root's left child.
    - If it's greater than the root, go to the root's rightchild.
- Repeat this until we've found the node containing the element.
- If the node is a leaf node, wipe the node and the parent's appropriate child reference.
- If the node is a semi-leaf, connect the node's child to the the node's parent, and update the parent's appropriate child reference. Then wipe the node.
- If the node is an internal node, find the maximum left-descendant of the node:
    - Go right from the left child until you can't anymore.
- Move this value into our node, then delete this new node as with a semi-leaf above.

(iv)

(a)

- 49 is now the right child of 37
- 57 is gone

(b)

- 48 is now the left child of 92
- 40 is now the right child of 36
- 57 is gone

(v)

No node is unbalanced. A node is considered unbalanced if the heights of its children differ by 2 or more, or if it has only one child, which has a height of 2 or more.

(vi)

(a)

- 37 is now the root
- 16 is the left child of 37
- 18 is the right child of 16
- 43 is the right child of 39

(b)

- 45 is now the root
- 16 is the left child of 45
- 43 is the right child of 16
- 50 is the right child of 45
- 50 has no left child

Question 3

(i)

A Priority Queue is an ordered structure where each item is stored with a key value, to represent its priority. The next item removed from the Priority Queue is always the item (or one item) that has the highest priority, which is defined as the item having the lowest key.

The standard operations are:

- `add(key, value)` – add a new element into the priority queue
- `min()` – return the element with the minimum key
- `remove_min()` – remove and return the element with the minimum key
- `is_empty()` – report whether there are no items in the queue
- `length()` – report the number of elements in the queue

(ii)

- `add(key, value)` – O(n)
    - In the worst case, we'll have to add the new element to the start of our list, and so we'll have to copy the whole list, making this O(n).
- `min()` – O(1)
    - With the sorted list implementation, the minimum element will be at the end of the list, and since list lookup is O(1), this is O(1).
- `remove_min()` – O(1)
    - Since the minimum element is at the end of the list, access is O(1), and removal is O(1) on average (as sometimes Python will shrink the list).
- `is_empty()` – O(1)
    - We do this by checking the length and comparing it to 0, both of which are O(1), so this is O(1).
- `length()` – O(1)
    - In-built `len()` is O(1) for lists (and most Python datatypes), making this O(1).

(iii)

A Binary Heap is a binary tree with the following properties that define its structure:

- Any node is less than its children.
- Every level except the last must be complete.
- The last level must be filled from left to right.
- Every node is either greater than or less than each other node (no nodes are equal to one another).

When a new node is added, it's added to the leftmost empty spot in the last level, to give the correct shape, and the size variable is incremented. The new node is then compared to its parent, and they're swapped if they're ordered incorrectly. This is repeated up the tree until the new element is in the right place.

The value at the root is saved to be returned at the end. The last element in the tree is copied into the root and the last node is removed. The element at the root is then compared to its lowest-ordered child, and they're swapped if the child has lower key – this process is repeated down the tree until no more swaps are needed.

The Binary Heap can be used to implement a Priority Queue by using the key of each node to determine the order – of two nodes, the node with lower key is less than the other node. This is an efficient implementation because:

- Both the add and remove methods are O(log(n)), as we only have to go along one path in the tree, which will be at most log(n) steps.
- Reporting `min()` is also O(1), as we keep a reference to the root.
- We keep a size variable, so reporting the length is O(1).
- For `is_empty()`, we compare the length to 0, which is O(1) because both operations are O(1).

(iv)

In the resulting diagram:

- 14 is where 16 was
- 16 is where 22 was
- 22 is in the new last position, left child of 16

(v)

In the resulting diagram:

- 26 is the root
- 28 is 26's right child
- 40 is 28's left child
- 34 has no right child