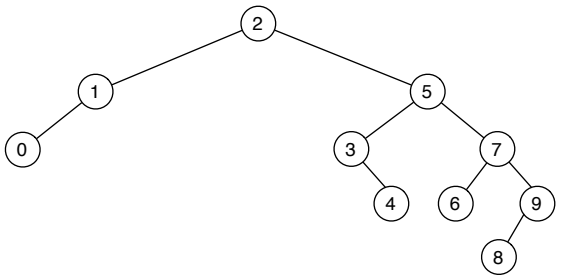
Lösningsförslag

Biinära sökträd

U 1. Binärt sökträd efter insättning av element med nycklarna: 2, 5, 1, 7, 9, 6, 3, 0, 8, 4.



U 2. Preorder: 2, 1, 0, 5, 3, 4, 7, 6, 9, 8
Inorder: 0, 1, 2, 3, 4, 5, 6, 7, 8, 9
Postorder: 0, 1, 4, 3, 6, 8, 9, 7, 5, 2

En traversering i inorder ger elementen i nyckelordning.

U 3. Minsta möjliga höjden för ett binärt träd $\approx \log n$ och största möjliga höjden = n.

U 4. I ett binärt sökträd tillåts inte dubbletter. Därför kan man bara använda ett sådant för att sortera om indata inte innehåller dubbletter. I så fall sätter man in alla element som ska sorteras i trädet. En inordertraversering av trädet ger elementen i växande ordning.

U 5.
```
public String toString() {
    StringBuilder sb = new StringBuilder();
    buildString(root, sb);
    return sb.toString();
}

public void buildString(Node<E> n, StringBuilder sb) {
    if (n != null) {
        buildString(n.left, sb);
        sb.append(n.element.toString());
        sb.append('\n');
        buildString(n.right, sb);
    }
}
```

U 6.
```
/**  Skriver ut innehållet i de noder som är större än eller lika med min
     och mindre än eller lika med max i växande ordning. */
public void printPart(E min, E max) {
    printPart(root, min, max);
}

private void printPart(Node<E> n, E min, E max) {
    if (n != null) {
        if (n.element.compareTo(min) > 0) { // vänster subträd kan ha
                                            // element i rätt intervall
            printPart(n.left, min, max);
        }
```

```
            if (n.element.compareTo(min) >= 0 &&
                n.element.compareTo(max) <= 0) { // min<=n.element<=max, skriv ut
                System.out.println(n.element);
            }
            if (n.element.compareTo(max) < 0) { // höger subträd kan ha
                                                // element i rätt intervall
                printPart(n.right, min, max);
            }
        }
    }
```

U 7. Ett binärt träd är balanserat om det för varje nod i trädet gäller att höjdskillnaden mellan dess båda subträd är högst ett.

Orsaken till att man vill balansera träd är att höjden är proportionell mot $\log n$ och därmed blir tidskomplexiteten för sökning, insättning och borttagning $O(\log n)$.

U 8. a) Programmet går inte att kompilera eftersom klassen `Book` inte implementerar `Comparable<Book>`.

b)
```
public class Book implements Comparable<Book>  {
    ...

    public int compareTo(Book rhs) {
        return isbn.compareTo(rhs.isbn);
    }

    public boolean equals(Object rhs) {
        if (rhs instanceof Book) {
            return compareTo((Book) rhs) == 0;
        } else {
            return false;
        }
    }
}
```