Lab 3

SortedLinkedListSet

Contains ser förenklat ut så här:

```
public boolean contains(Comparable x) {
    //some code
    while(nextNode.next != null && !isInSet) {
        if(nextNode.next.elt.equals(x)) {
            isInSet = true;
        }
        nextNode = nextNode.getNext();
    }
    return isInSet;
}
```

Vi ser att det värsta scenariot blir när x inte finns i listan då samtliga element i listan måste jämföras. Antalet operationer begränsas uppåt av listans storlek, som alltså är vårt n och tidskomplexiteten blir då $O(n)$.

Add ser förenklat ut så här:

```
public boolean add(Comparable x) {
    //some code
    Node tempNode = head;
    while(tempNode.getNext() != null){
        if(tempNode.getNext().elt.compareTo(x) == 0){
            return false;
        }
        if(tempNode.getNext().elt.compareTo(x) > 0){
            //some code
            return true;
        }
        tempNode = tempNode.getNext();
    }
    //some code
    return true;
}
```

Även här är det värsta scenariot när hela listan måste jämföras för att hitta rätt plats för elementet, i slutet av listan. Återigen är vårt n listans storlek och tidskomplexiteten $O(n)$.

Remove ser förenklat ut så här:

```
public boolean remove(Comparable x) {
    //some code
    Node tempNode = head;
    while(tempNode.getNext() != null){
        if((tempNode.getNext().elt).equals(x)){
            //some code
            return true;
        }
        tempNode = tempNode.getNext();
    }
    return false;
}
```

Samma sak gäller för remove, om elementet inte finns i listan måste hela listan jämföras, n är listans storlek och tidskomplexiteten $O(n)$.

SplayTreeSet

Contains ser förenklat ut så här:

```
public boolean contains(Comparable x) {
        //some code
        Node next = root;
        boolean keepLooping = true;
        while (keepLooping) {
            int c = next.elt.compareTo(x);
            if (c > 0) {
                if (next.getLeft() == null) {
                    keepLooping = false;
                } else {
                    next = next.getLeft();
                }
            } else if (c < 0) {
                if (next.getRight() == null) {
                    keepLooping = false;
                } else {
                    next = next.getRight();
                }
            } else {
                isInTree = true;
                keepLooping = false;
            }
        }
        splay(next);
        return isInTree;
    }
```

Det värsta scenariot är om elementet inte finns i trädet och antalet operationer begränsas av trädets storlek. Eftersom trädet är ett binärt sökträd är den optimala tidskomplexiteten $O(log\ n)$ och det västa scenariot, om alla element är sorterade när de stoppas in i trädet $O(n)$.

Dock görs i varje anrop av contains ett anrop till splay med trädet som inparameter.

splay ser förenklat ut så här:

```
public void splay(Node nod){
    //some code
    while(nod != root) {
        Node parent = nod.getParent();
        if( parent == root || parent.getParent() == null){ //do Zig step
            //some code
        }else {
            Node grandParent = parent.getParent();
            if(parent.isRight() == nod.isRight()){ //do zigzig
                if(nod.isRight()){ //they are both right children
                    //some code
                } else {
                    //some code
                }
```

```
            }else{ //do zigzag
                if(nod.isRight()){
                    //some code
                }else{
                    //some code
                }
            }
            if(grandParent == root|| nod.getParent() == null){
                //some code
            }
        }
    }
}
```

Vi ser att splay har samma komplexitet som contains men eftersom att vi hela tiden balanserar trädet kommer medelvärdet för tidskomplexiteten att närma sig den optimala.

Medelvärdet för den tidskomplexiteten för contains blir alltså $2 * log\ n\ =\ O(log\ n)$.

Add ser förenklat ut så här:

```
public boolean add(Comparable x) {
        //some code
        Node next = root;
        boolean keepLooping = true;
        while(keepLooping) {
            int c = next.elt.compareTo(x);
            if(c > 0){
                if(next.getLeft() == null){
                    //some code
                    splay(newNode);
                }else{
                    next = next.getLeft();
                }
            }else if (c < 0){
                if(next.getRight() == null){
                    //some code
                    splay(newNode);
                }else{
                    next = next.getRight();
                }
            }else {
                return false;
            }
        }
        return true;
    }
```

Vi ser att samma resonemang gäller för add som för contains, n är trädets storlek, anrop till splay görs så den genomsnittliga tiden för att hitta rätt plats i trädet blir $log\ n$ och splay har fortfarnade samma tidskomplexitet. Den genomsnittliga tidskomplexiteten blir alltså $2 * log\ n\ =\ O(log\ n)$.

Remove ser förenklat ut så här:

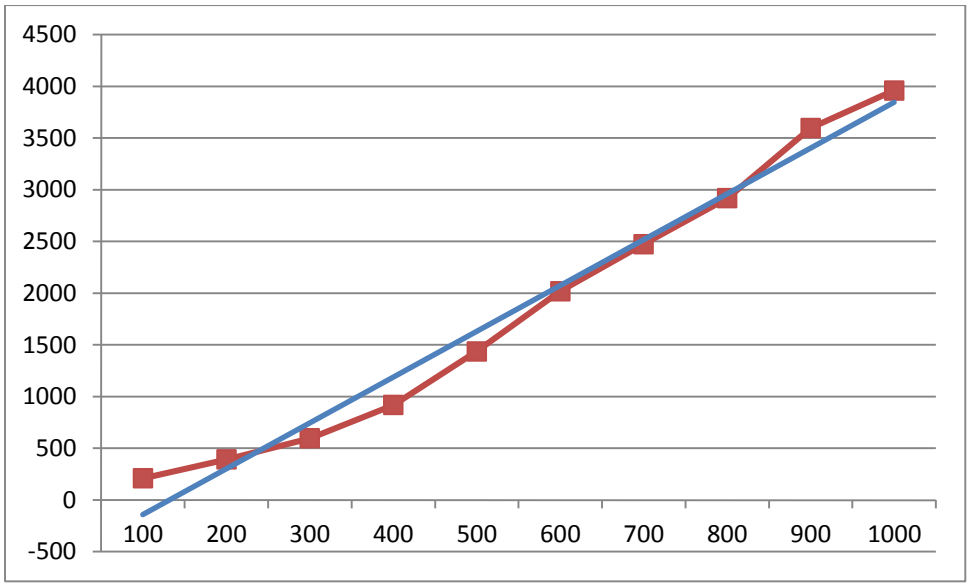
```
    public boolean remove(Comparable x) {
        boolean keepLooping = true;
        boolean doRemove = false;
```

```
        //some code
        Node next = root;
        while(keepLooping){
            int c = next.elt.compareTo(x);
            if(c > 0) {
                if(next.getLeft() == null) {
                    //some code
                }else{
                    next = next.getLeft();
                }
            }else if(c < 0) {
                if(next.getRight() == null) {
                    //some code
                }else{
                    next = next.getRight();
                }
            }else{
                keepLooping = false;
                doRemove = true;
                size--;
            }
        }
        if(doRemove){
            splay(next);
            if(next.left != null && next.right != null) {
                //some code
                while(largestLeft.right != null){
                    //some code
                }
                splay(largestLeft);
                //some code
            }else if (next.left != null && next.right == null ){
               //some code
            }else if(next.left == null && next.right != null){
                //some code
            }else{
                //some code
            }
        }
        return doRemove;
    }
```

Vi ser att remove skiljer sig en aning från add och contains i och med att den i värsta fall har ytterligare ett anrop till splay. I övrigt är resonemanget ovan giltigt och tidskomplexiteten blir $3 * log\ n\ =\ O(log\ n)$.

Digram

Den röda linjen visar uppmätta värden för SortedLinkedListSet och den blå linjen en linjär funktion anpassad efter de upmätta värdena. Linjerna verkar stämma väl överens även om de skiljer sig en del för låga värden på n.



Den röda linjen visar uppmätta värden för SplayTreeSet och den blå linjen en logaritmisk funktion anpassad efter de upmätta värdena. Även här verkar linjerna stämma väl överens.