Problem Set 1

Anne Gatchell

Due February 4, 2013

1

For reference: $1day * (24hrs/1day) * (3600s/1hr) * (10^6/1s) = 8.64x10^4\mu s$

A

In this case, Acme should pay professor Flitwick.
$n = 41$
$f(n) = 1.99^n$
$g(n) = n^3$
$t = 17days = 1.4688x10^{12}\mu s$

Without Flitwick, it takes $f(41) = 1.99^{41} = 1.79x10^{12}\mu s$
With Flitwick it takes $t + g(n) = 1.47x10^{12}\mu s + 41^3 = 1.46x10^{12}\mu s$

It will take Flitwick less time to spend 17 days working and then running his program than it will take to just run the program by itself:

Flitwick's alg is $17days, 0.069s$
The other lag will take $20days, 17hours, 21min, 47.45seconds$

B

In this case, Acme should not pay professor Flitwick.
$n = 10^6$
$f(n) = n^{2.00}$
$g(n) = n^{1.99}$
$t = 2days = 2 * 8.64x10^{10}\mu s = 1.728x10^{11}\mu s$

Without Flitwick, it takes $f(10^6) = (10^6)^{2.00} = 10^{12}\mu s =$
$11.574days = 11days, 13hours, 46minutes, 40seconds$

With Flitwick it takes $t + g(n) = 2days + (10^6)^{1.99} = 1.73x10^{11}\mu s + 8.71x10^{11}\mu s = 1.044x10^{12}\mu s$
$1.044x10^{12}/8.64x10^{10}\mu s = 12.08days = 12days, 1hr, 56min, 3.6s$

It is a close call, but it will be faster to just run the original algorithm and not pay Flitwick.

I worked on Problem 1 by myself

2

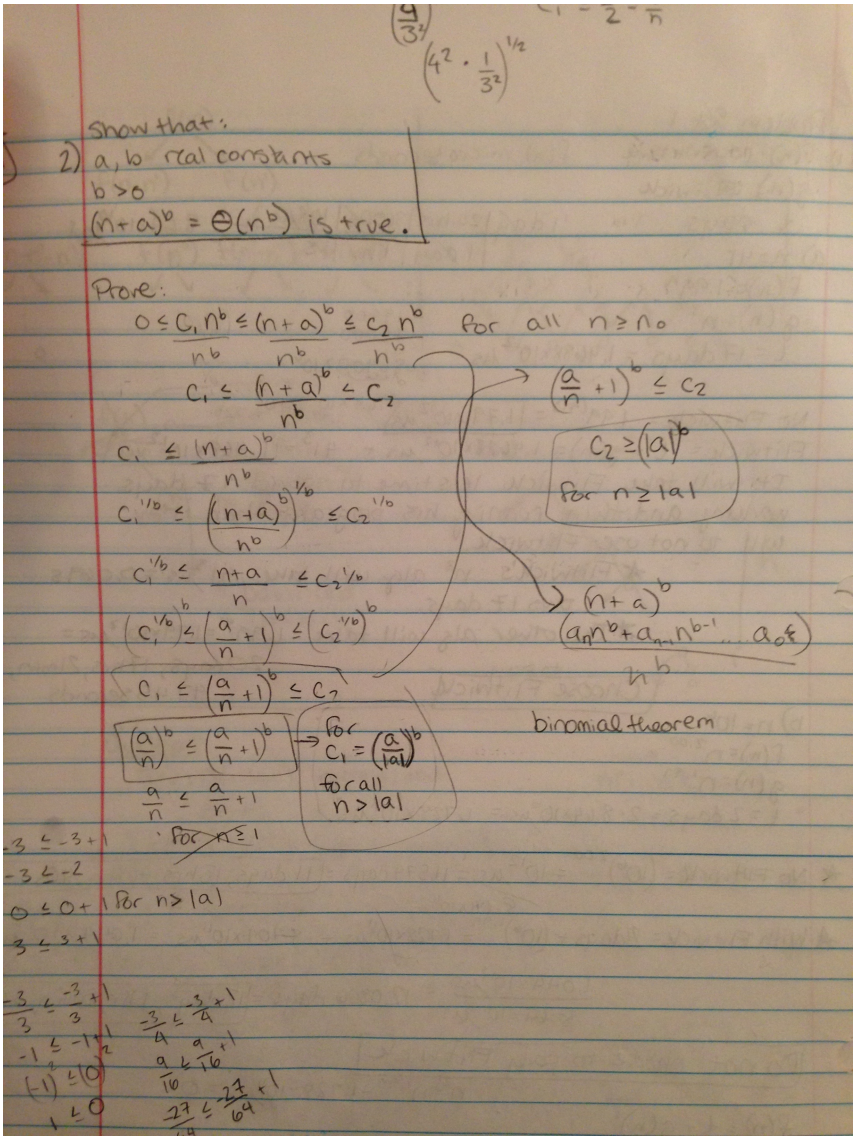

Figure 1: default

Another way to look at this problem is that we have
$0 \le c_1 n^b \le (n+a)^b \le c_2 n^b$ for all $n \ge n^0$
The series expansion of the $(n+a)^b$ numerator (courtesy of Wolram Alpha) at $n = 0$ is

$a^b + bna^{b-1} + 1/2(b-1)bn^2a^{b-2} + 1/6(b-2)(b-1)bn^3a^{b-3} + 1/24(b-3)(b-2)(b-1)bn^4a^{b-4} + O(n^5)$

We can see that if we divide all sides by n^b, each term after the first term will be very small as n goes to infinity, leaving a constant relation like $0 \leq c_1 \leq A \leq c_2$ which we can certainly pick c_1 and c_2 to satisfy. This insight can be seen in any expansion of a polynomial. The largest term will be an n^b term, matching our $\Theta(n^b)$ as we take the limit as n goes to infinity.

I worked on Problem 2 by myself, but I did discuss how we were supposed to prove this with the group Nora Conner, Anshul Kanakia, John Klingner, Andy McEvoy, Bill Casson. We were trying to decide if it makes sense to expand the numerator partially. Since I wasn't sure about that when I did it on my own, and I am also unsure about my other proof, I put both down

3

A

Is there a c for which $0 \leq 2^{nk} \leq c2^n$ for $k > 1$?

Dividing both sides by 2^n:
$(2^n)^{k-1} \leq c$

No. There is no constant c that is greater or equal to $(2^n)^{k-1}$ for sufficiently large n.

B

Is there a c for which $0 \leq 2^{n+k} \leq c2^n$ for $0 \leq k \leq c$ (some positive constant)?

$2^n * 2^k \leq c2^n$
Dividing both sides by 2^n:
$2^k \leq c$for Yes. For $n_o = 0, c \geq 2^k$ where $k \geq 0$.

I worked on Problem 3 by myself

4

A

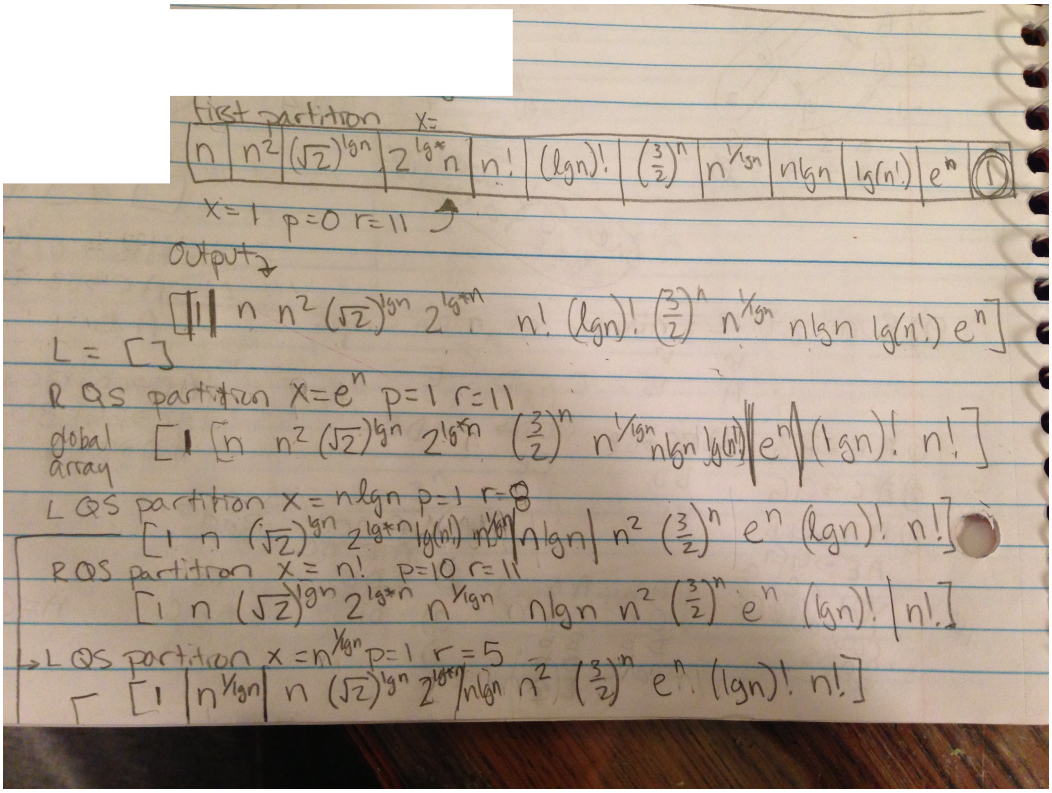

Figure 2: 4a

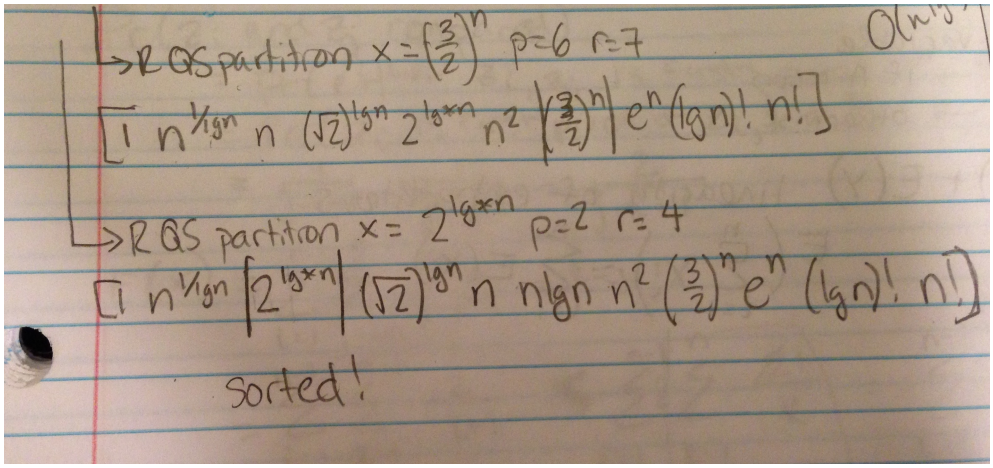

Figure 3: 4a continued

B

Final order: 1, $n^{1/lgn}$, 2^{lg*n}, $(\sqrt{2})^{lgn}$, n, $nlgn$, $lg(n!)$, n^2, $(3/2)^n$, e^n, $(lgn)!$, $n!$

Both $nlgn$, $lg(n!)$ are in the $O(nlgn)$ equivalence class.

I talked about l'hopitals and comparing some of the functions in 4 with John Klingner, Andy McEvor, and possibly Nora Conner, Anshul Kanakia, and Bill Casson but I don't remember if the latter three really were discussing that problem with us

5

A

$f_0 = 3$
$f_1 = 5$
$f(n) = 3 * f_{n-1} - 2 * f_{n-2}$ for $n \geq 2$
The base cases are f(0) and f(1).

B

$F_n = 3 * F_{n-1} - 2 * F_{n-2}$
$F_{n+2} = 3 * F_{n+1} - 2 * F_n$
$F_{n+2} - 3 * F_{n+1} + 2 * F_n = 0$

This corresponds to the characteristic polynomial:
$x^2 - 3x + 2 = 0$
Solving for the roots:
$(x-2)(x-1) = 0$
$x = 2$ and $x = 1$

Using these roots, the expression below solves the recursion:
$Fn = a_1 * 2^n + a_2 * 1^n$
To solve for a_1 and a_2, we use the two base cases to create a system of equations:
$n = 0 : 3 = a_1 * 2^0 + a_2 * 1^0$
$n = 1 : 5 = a_1 * 2^1 + a_2 * 1^1$
———————————— adding the two equations...
$-2 = a_1 - 2 * a_1$
$a_1 = 2$
$a_2 = 3 - 2 = 1$
Therefore:
$F_n = 2^{n+1} + 1$

C

Base cases: $f(0) = 3 = 2^{0+1} + 1 = 3$ Checks out.
$f(1) = 5 = 2^{1+1} + 1 = 5$ Checks out.
Assume true up to n-1.
$F(n) = 3 * F_{n-1} - 2 * F_{n-2} = 3 * (2^{n-1+1} + 1) - 2 * (2^{n-2+1} + 1)$
$F(n) = 3 * (2^n + 1) - 2 * (2^{n-1} + 1)$
$F(n) = 3 * 2^n + 3 - 2 * 2^{n-1} - 2$
$F(n) = 3 * 2^n - 2^n + 1$
$F(n) = 2 * 2^n + 1$
$F(n) = 2^{n+1} + 1$ Checks out

D

$T(n) = \Theta(1)$ if $n \leq 1$ and $T(n-1) + T(n-2) + \Theta(1)$ if $n > 1$

E

The recursion tree of this function will be a height of n and a width of $2^l * c$ where c is some constant and l is the level of the recursion tree. A rough integral of the $2^l * c$ function over levels (l) from 0 to n shows that the running time would be something like $c2^n$. However, the recursion tree is not a dense tree, so it will not actually be reaching the $O(2^n)$ as a tight bound.

F

The recurrence relation in (D) can be written as:
$T_0 = c$
$T_1 = c$
$T(n) = T_{n-1} + T_{n-2} + c$ where the c's are constant times
This corresponds to the characteristic polynomial:
$x^2 - x - 1 = c$
For now, we ignore the constant and solve $x^2 - x - 1 = 0$ for the roots:
$x = (1 \pm \sqrt{5})/2$, or the Golden Ratio, ϕ

The equations below solve the recursion: $T(n) = a_1 * (\phi)^n + a_2 * (1 - \sqrt{5})/2)^n$
Since we are looking for running time, this corresponds to:
$T(n) = O(1.61^n - 0.61^n)$, which explains why $O(2^n)$ was not a tight bound

I worked on problem 5 alone. I did ask about the characteristic polynomials in my study group, but I ended up looking them up and learning them on my own

6

A

I am assuming that T(0) = 0; $T(n) = T(n-1) + n$
$T(n) = T(n-2) + n - 1 + n$
$T(n) = T(n-3) + n - 2 + n - 1 + n$
$T(n) = T(n-4) + n - 3 + n - 2 + n - 1 + n$
.
.
.
$T(n) = T(n-(n-1)) + (n-(n-2)) + ... + n - 3 + n - 2 + n - 1 + n$
$T(n) = T(n-(n-0)) + T(n-(n-1)) + (n-(n-2)) + ... + n - 3 + n - 2 + n - 1 + n$
This leads to: $T(n) = 0 + n - (n-1) + n - (n-2) + ... + n - (3) + n - (2) + n - (1) + n - (0)$
$T(n) = n^2 - ((n-1) + (n-2) + ... + 3 + 2 + 1)$
$T(n) = n^2 - \sum_{k=1}^{n-1} k$
$T(n) = n^2 - (\sum_{k=1}^{n} k - n) = n^2 - (1/2n(n+1) - n)$
$T(n) = n^2 - n^2/2 + n/2 - n$
$T(n) = n^2/2 + n/2$
$T(n) = n(n+1)/2$
$T(n) = \Theta(n^2)$

B

$T(n) = 2T(n/2) + n^3$
$T(n) = aT(n/b) + f(n)$

$f(n) = n^3 = \Omega(n^{log_b a+\epsilon}) = \Omega(n^{log_2 2+\epsilon}) = \Omega(n^{1+\epsilon})$
if $2 * f(n) \leq cf(n)$
$2 * (n/2)^3 \leq c * n^3$ for $c < 1$
$1/4n^3 \leq cn^3$ for $c < 1$

So, $T(n) = \Theta(n^3$

C

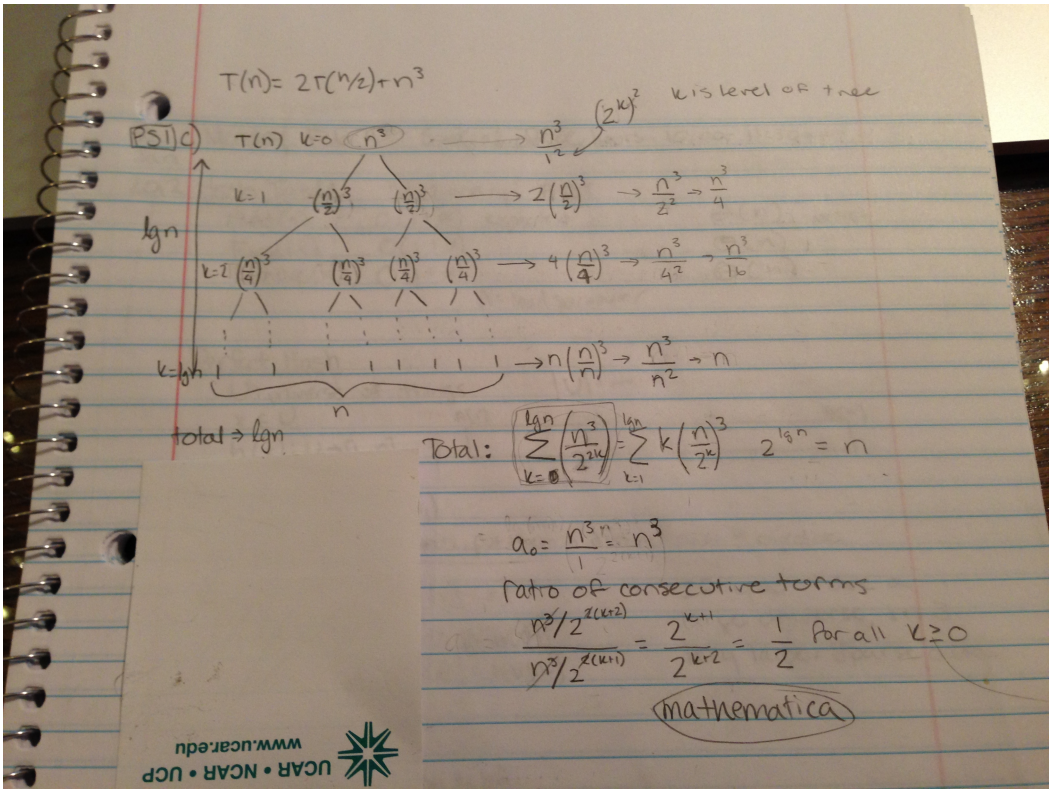

Figure 4: default

Worked alone on problem 6

7

A

If Snape had n/2 or more

B

If there are more than n/2 good chips, then Snape can simply split the n chips in half and compare one set of the chips to the other set. Since over half of them are good, if $nmod2 = 0$, there must be at least one pair of good chips that get compared to each other. Therefore, he then simply must take one of each Good-Good result to put in his pool to test further against each other in the same manner. The reason he only need to take one of each Good-Good pair is because in a Good-Good outcome, it is possible that both chips are good or both chips are bad. So, there is only a need to try one of the pair, since all that matter is finding one good chip from all the bad ones. In the worst case there is an even number of both bad and good chips, and all the bad chips in the bunch get tested against each other, and they all decided to say that they are Good-Good pairs. So, Snape gets n/2 pairs of Good-Good. This is not a problem, though, because Snape only takes one of each pair of Good-Goods, which will still reduce the size to n/2.

If he continues to follow this pattern, he will quickly keep halving his group of chips.

A key part to this technique is that any time a good chip is paired with a Bad chip, the Good chip prevents the result from being Good-Good. So, bad chips continue to be eliminated any time there is an odd number of bad chips (the extra will be compared to a good chip).

There is also no chance that the Good chips could become outnumbered but he bad chips, because even if all Bad chips are compared to goods, resulting in the elimination of many good chips through Good-Bad pairings, the remaining Good-Good pairs would be only good chips, which would end up getting tested together.

C

The recurrence relation is $T(n) = 1 * T(n/2) + O(n)$ because the problem is split into 1 subset of size $n/2$, and the time it takes to go through the n/2 pairs of results to pick up the one of each Good-Good pair is O(n).

Using the Master Method,$f(n) = O(n) = \Omega(n^{log_2(1+0.5)})$ and $1 * (n/2) \leq c * n$ for $1/2 < c < 1$, $T(N) = \Theta(n)$

I worked on Problem 7 a and b with John Klingner, Andy McEvor, Nora Conner, Anshul Kanakia and a small amount of group discussion was held with Bill Casson. After day one of working with the first 4 in the group, we had not come up with a way to solve B, even though we had recognized pretty easily that A is true. We spent time trying to come up with comparison scenarios. I came up with the final insight

on my own after that session, and I discussed it a bit with John Klingner and Anshul Kanakia, since they had said they had solved it too. I hadn't fleshed out the whole solution (ie. picking up only half of the Good-Good pairs) yet, but the hard part was done

8

A smallest possible counterexample is as follows:

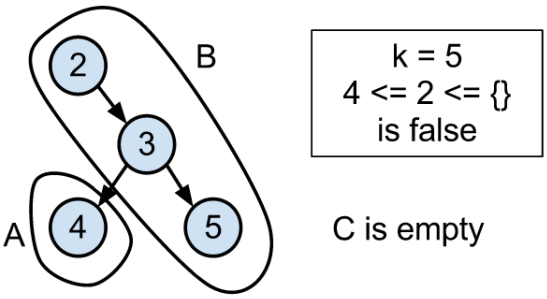

Figure 5: default

I talked about l'hopitals and comparing some of the functions in 4 with John Klingner and Nora Conner, and we looked for examples of binary trees on Wikipedia, which gave good insight. I drew my own tree. Anshul Kanakia, Andy McEvoy, and Bill Casson were working on that problem in the room with the group another day and we did give them a nudge to look for examples. I can't remember if they took it

9

My code is at the end of the file. I am still trying to work out a bug in my insertion code. It is difficult to isolated because it does not happen every time you insert something to the list. I have implemented Insert, Delete, and Search, but there are still issues. Worked alone on problem 9. Did ask Anshul Kanakia if he used arrays or linked lists for the towers and he said that he used linked lists that then were cross-linked

10

Since my SkipList code has bugs for adding things to the list, I was unable to plot the graph. Worked alone on problem 9. Did ask Anshul Kanakia if he used arrays or linked lists for the towers and he said that he used linked lists that then were cross-linked

11 SOURCE CODE

SkipList.java

```
package src.main;

import java.util.Random;

public class SkipList<K extends Comparable<K>, T>{
        private float p; //p nodes with level i pointers also have level i+1 pointers
        private Element<K, T> header;
        private int maxLevel;
        private int level;
        //TODO: Make MaxLevel start off at a resonable level and then increase with numElements
        private int numElements;

        public SkipList(float probability){
                p = probability;
                numElements = 0;
                level = 1;
                maxLevel = 16;
                header = new Element<K, T>(null, null, maxLevel);
        }

        public SkipList(float probability, int powerOfTwoSize){
                p = probability;
                numElements = 0;
                maxLevel = powerOfTwoSize;
                level = 1;
                header = new Element<K, T>(null, null, maxLevel);
        }

        public T search(K searchKey){
                /*Search(list, searchKey)
                        x := list header
                        -- loop invariant: x key < searchKey
                        for i := list level downto 1 do
                                while x forward[i] key < searchKey do
```

```
                        x := x forward[i]
                -- x key < searchKey      x forward[1] key
                x := x forward[1]
                if x key = searchKey then return x value
                        else return failure*/
        header.resetCurrentLevelToRoot();
        Element<K, T> x = header;
        int lev = x.getCurrentLevelNum();
        //Go from top level down to level 1
        while(lev > 0){
                //If the next element at this level is not null and is less than the key, advance
                while(x.getNextElementForCurrentLevel() != null
                        && x.getNextElementKeyForCurrentLevel().compareTo(searchKey) < 0){
                        x = x.getNextElementForCurrentLevel();
                }
                //System.out.println(x.getCurrentLevelNum() + " " + x.getKey());
                // System.out.println(update.getCurrentLinkNum() + " " + update.getCurrentLink());
                //Advance to the next level down in our update list as well as our current element
                if(lev > 1){
                        x.advanceCurrentLevel();
                        lev = x.getCurrentLevelNum();
                }else{
                        lev--;
                }
        }
        //Go to the next element
        x = x.getNextElementForCurrentLevel();
        System.out.println("HERE!!!");
        if(x != null && x.getKey().compareTo(searchKey) == 0){
                return x.getData();
        } else {
                return null;
        }
}
```

```
        public void delete(K searchKey){
                LinkedList<K, T> update = new LinkedList<>(maxLevel);
                Element<K, T> x = header;
                K key;
                header.resetCurrentLevelToRoot();
                int lev = x.getCurrentLevelNum();
                //Go from top level down to level 1
                while(lev > 0){
                        //If the next element at this level is not null and is less than the key, advance
                        while(x.getNextElementForCurrentLevel() != null
                                && x.getNextElementKeyForCurrentLevel().compareTo(searchKey) < 0){
                                x = x.getNextElementForCurrentLevel();
                        }
                        //Store the current element in our update save list
                        update.setCurrentLink(x.getCurrentLevelLink());
                        //Since update is a linked list, we need to set the root if this is the top
                        if(x.getCurrentLevelNum() == maxLevel){
                                update.setRoot(x.getCurrentLevelLink());
                        }
                        // System.out.println(x.getCurrentLevelNum() + " " + x.getCurrentLevelLink());
                        // System.out.println(update.getCurrentLinkNum() + " " + update.getCurrentLink());
                        //Advance to the next level down in our update list as well as our current element
                        if(lev > 1){
                                update.advanceCurrent();
                                x.advanceCurrentLevel();
                                lev = x.getCurrentLevelNum();
                        }
                        else
                                lev--;
                }

                // System.out.println(x.getCurrentLevelNum() + " " + x.getCurrentLevelLink());


/**** Purely code to test *****/
```

```
                // update.resetCurrentLinkToRoot();
                // while(update.getCurrentLinkNum() > 1){
                //       System.out.println(update.getCurrentLinkNum() + " " + update.getCurrentLink());
                //       update.advanceCurrent();
                // }
                // System.out.println(update.getCurrentLinkNum() + " " + update.getCurrentLink());

/**************/
                //Go to the next element
                x = x.getNextElementForCurrentLevel();
                //If key == our key to delete, we can delete it
                if(x != null && x.getKey().compareTo(searchKey) == 0){

                        //Go to the top of the update list, attach the new vector
                        update.resetCurrentLinkToRoot();
                        lev = update.getCurrentLinkNum();
                        while(lev > 0){
                                //System.out.println("lev "+ lev);
                                //When we hit level v, start splicing in the new element x
                                if(update.getCurrentLinkNextElement() == x){
                                        update.setCurrentLinkNextElement(x.getNextElementForCurrentLevel());
                                }
                                if(lev > 1){
                                        update.advanceCurrent();
                                        lev = update.getCurrentLinkNum();
                                }else lev--;
                        }
                        header.resetCurrentLevelToRoot();
                        numElements--;
                }

                        //Reset current pointers
                header.resetCurrentLevelToRoot();
        }
```

```
public Element<K, T> insert(K searchKey, T newValue){
        LinkedList<K, T> update = new LinkedList<>(maxLevel);
        Element<K, T> x = header;
        K key;
        header.resetCurrentLevelToRoot();
        int lev = x.getCurrentLevelNum();
        //Go from top level down to level 1
        while(lev > 0){
                //If the next element at this level is not null and is less than the key, advance
                if(x.getNextElementForCurrentLevel() != null){
                System.out.println("Lev key comparision " + x.getNextElementKeyForCurrentLevel() + " " +
        }
                while(x.getNextElementForCurrentLevel() != null
                                && x.getNextElementKeyForCurrentLevel().compareTo(searchKey) < 0){
                        System.out.println("Went to next x");
                        x = x.getNextElementForCurrentLevel();
                }
                //Store the current element in our update save list
                update.setCurrentLink(x.getCurrentLevelLink());
                //Since update is a linked list, we need to set the root if this is the top
                if(x.getCurrentLevelNum() == maxLevel){
                        update.setRoot(x.getCurrentLevelLink());
                }
                System.out.println(x.getCurrentLevelNum() + " " + x.getCurrentLevelLink());
                System.out.println(update.getCurrentLinkNum() + " " + update.getCurrentLink());
                //Advance to the next level down in our update list as well as our current element
                if(lev > 1){
                        update.advanceCurrent();
                        x.advanceCurrentLevel();
                        lev = x.getCurrentLevelNum();
                }
                else
                        lev--;
        }
```

```
                // System.out.println(x.getCurrentLevelNum() + " " + x.getCurrentLevelLink());


/**** Purely code to test *****/
                // update.resetCurrentLinkToRoot();
                // while(update.getCurrentLinkNum() > 1){
                //      System.out.println(update.getCurrentLinkNum() + " " + update.getCurrentLink());
                //      update.advanceCurrent();
                // }
                // System.out.println(update.getCurrentLinkNum() + " " + update.getCurrentLink());

/**************/
                //Go to the next element
                System.out.println("Pre last move " + x.getKey());
                x = x.getNextElementForCurrentLevel();
                //System.out.println("Post last move " + x.getKey());

                //If key == our key, update the data
                if(x != null && x.getKey().compareTo(searchKey) == 0){
                        x.setData(newValue);
                }else{
                //Otherwise, create a new Element with a level v
                        int v = randomLevel();
                        System.out.println("Level generated: "+v);
                        //Keep track of our list's current level for Search
                        if(v > level){
                                level = v;
                        }
                        x = new Element<K, T>(searchKey, newValue, v);
                        x.resetCurrentLevelToRoot();
                        //Go to the top of the update list, attach the new vector
                        update.resetCurrentLinkToRoot();
                        lev = update.getCurrentLinkNum();
                        while(lev > 0){
                                // System.out.println("lev "+ lev);
```

```
                        //When we hit level v, start splicing in the new element x
                        if(lev <= v){
                                x.setNextElementForCurrentLevel(update.getCurrentLinkNextElement());
                                update.setCurrentLinkNextElement(x);
                                System.out.println("x is "+ x + " "+ x.getNextElementForCurrentLevel() +
                                if(lev > 1)
                                        x.advanceCurrentLevel();
                        }
                        if(lev > 1){
                                update.advanceCurrent();
                                lev = update.getCurrentLinkNum();
                        }else lev--;

                }
                // update.advanceCurrent();

                numElements++;
                //Reset current pointers
                x.resetCurrentLevelToRoot();
                header.resetCurrentLevelToRoot();
        }
        return x;

}

public int randomLevel(){
        int v = 1;
        Random rand = new Random();
        //random value between [0...1)
        while (rand.nextDouble() < p && v < maxLevel){
                v++;
        }
        return v;

}
```

```
public void traverseInOrderAndPrintKeys(){
        int count = 0;
        K k;
        Element<K, T> current = header;
        while(true){
                if(current.getNextElementForCurrentLevel() != null){
                        k = current.getNextElementForCurrentLevel().getKey();
                } else{
                        k = null;
                }
                System.out.println(count + "-"+current.getCurrentLevelNum() + " " + k);
                while(current.getCurrentLevelNum() > 1){
                        current.advanceCurrentLevel();
                        if(current.getNextElementForCurrentLevel() != null){
                                k = current.getNextElementForCurrentLevel().getKey();
                        } else{
                                k = null;
                        }
                        System.out.println(count + "-"+current.getCurrentLevelNum() + " " + k);
                }
                current.advanceToBottomLevel();
                current = current.getNextElementForCurrentLevel();
                if (current == null){
                        break;
                }
                current.resetCurrentLevelToRoot();
                count++;
        }
}

public Element<K, T> getHeader(){
        return header;
}
```

```
        public int getMaxLevel(){
                return maxLevel;
        }

        public int getNumElements(){
                return numElements;
        }

}
```

Element.java

```
package src.main;

public class Element<K extends Comparable<K>, T>{
        private LinkedList<K, T> levels;
        private int height;
        private K key;
        private T data;

        //General constructor for a null list
        public Element(K k, T d, int h){
                key = k;
                data = d;
                height = h;
                levels = new LinkedList<>(h); //null linked list
        }

        //Constructor for copying an array of pointers
        //e.g. when using the update temporary node to insert/delete
        public Element(K k, T d, LinkedList<K, T> lst){
                key = k;
                data = d;
                height = lst.getLength();
                levels = lst;
        }
```

```
// public Element(K k, T d, Element<K, T>[] update, int height){
//       key = k;
//       data = d;
//       height = height;
//       levels = new LinkedList<>(height, update);
// }

public int getCurrentLevelNum(){
        return levels.getCurrentLinkNum();
}

public Element<K, T> getNextElementForCurrentLevel(){
        return levels.getCurrentLinkNextElement();
}

public void setNextElementForCurrentLevel(Element<K, T> elem){
        levels.setCurrentLinkNextElement(elem);
}

public K getNextElementKeyForCurrentLevel(){
        if(levels.getCurrentLinkNextElement() == null){
                return null;
        }
        return levels.getCurrentLinkNextKey();
}

public void resetCurrentLevelToRoot(){
        levels.resetCurrentLinkToRoot();
}

public Link<K, T> advanceCurrentLevel(){
        return levels.advanceCurrent();
}

public Link<K, T> getCurrentLevelLink(){
```

```
        return levels.getCurrentLink();
}

public void advanceToBottomLevel(){
        while(getCurrentLevelNum() > 1){
                levels.advanceCurrent();
        }
}

public void setTopLevel(Link<K, T> tL){
        levels.setRoot(tL);
}

public Link<K, T> getTopLevel(){
        return levels.getRoot();
}

public K getKey(){
        if(key == null){
                return null;
        }
        return key;
}
public void setKey(K k){
        key = k;
}

public void setData(T newData){
        data = newData;
}
public T getData(){
        return data;
}

public int getHeight(){
```

```
                return height;
        }


}
```

LinkedList.java

```
package src.main;

public class LinkedList<K extends Comparable<K>, T>{
        private Link<K, T> root;
        private Link<K, T> current;
        private int length;

        //Make a new linked list from an old one
        public LinkedList(int l, LinkedList<K, T> lL){
                root = lL.getRoot();
                current = root;
                length = l;
        }

        // public LinkedList(int l, Element<K, T> elem){
        //      int elemSize = elem.getHeight();
        //      Link<K, T> temp;
        //      elem.resetCurrentLevelToRoot();
        //      while(elem.getCurrentLevelNum() >= l){
        //              temp = elem.advanceCurrentLevel();
        //      }
        //      root = temp;
        //      current = root;
        //      length = l;
        // }

        // public LinkedList(int l, Element<K, T>[] elems){
        //      length = l;
        //      root = new Link<>(length-1, elems[length-1]);
```

```
//      current = root;
//      for(int i = length - 2; i > -1; i--){
//              current.setNextLevelDown(elems[i]);
//              current = current.getNextLevelDown();
//      }
//      current = root;
// }

//Set up the root of a linked list
// public LinkedList(int l, Link<K, T> lL){
//      root = lL;
//      current = root;
//      length = l;
// }

//Set up the root of a linked list with the element
//TODO: WTF
// public LinkedList(int l, Element<K, T> elem){
//      length = l;
//      Link<K, T> link= new Link<K, T>();
//      root = link;
//      current = root;
// }

//Create a new linked list of nulls
public LinkedList(int l){
        length = l;
        root = new Link<K, T>(l);
        current = root;
        for(int i = 1; i < length; i++){
                current.setNextLevelDown(new Link<K, T>(l-i));
                current = current.getNextLevelDown();
        }
        current = root;
}
```

```
// public void growListByNumElementsAtRoot(int len){
//       //Update length
//       length = length + len;
//       current = root;
//       Link<K, T> beginning = new Link<K, T>(length-1);
//       Link<K, T> temp = beginning;
//       for(int i = 1; i < len; i++){
//               temp.setNextLevelDown(new Link<K, T>(length-1-i));
//               temp = temp.getNextLevelDown();
//       }
//       //Point to the current root
//       temp.setNextLevelDown(root);
//       //Reset root to be the beginning of the list again
//       root = beginning;
//       //Set current to be the root
//       current = root;

// }

public void setNextLink(Link<K, T> l){
        //Should I check if null or not to make sure it will be
        current.setNextLevelDown(l);
}

public Link<K, T> getNextLink(){
        return current.getNextLevelDown();
}

public Link<K, T> getCurrentLink(){
        return current;
}
public void setCurrentLink(Link<K, T> l){
        current = l;
}
```

```
public int getCurrentLinkNum(){
        return current.getNum();
}

public Element<K, T> getCurrentLinkNextElement(){
        return current.getNextElement();
}

public void setCurrentLinkNextElement(Element<K, T> elem){
        current.setNextElement(elem);
}

public K getCurrentLinkNextKey(){
        return current.getNextElementKey();
}

public Link<K, T> advanceCurrent(){
        current = current.getNextLevelDown();
        return current;
}

public void resetCurrentLinkToRoot(){
        current = root;
}

public void setRoot(Link<K, T> l){
        root = l;
}
public Link<K, T> getRoot(){
        return root;
}
public int getLength(){
        return length;
}
```

```
        public static void main(String[] args){
                LinkedList<Integer, String> lNull4 = new LinkedList<>(4);
                Link<Integer, String> initialStartOfList = lNull4.getRoot();
        //Advance two links to see if that will be the root
        lNull4.advanceCurrent(); // should be link 1
        lNull4.advanceCurrent(); // should be link 2, same as old root
        }

}
```

Link.java

```
package src.main;

public class Link<K extends Comparable<K>, T>{
        private Element<K, T> nextElement;
        private Link<K, T> nextLevelDown;
        private int num;

        public Link(int n){
                num = n;
                nextElement = null;
                nextLevelDown = null;
        }

        // public void Link(){
        //      nextElement = null;
        //      nextLevelDown = null;
        // }

        public Link(int n, Element<K, T> nextE){
                nextElement = nextE;
                num = n;
        }

        public Link(Element<K, T> nextE, Link<K, T> nextL, int n){
```

```
                nextElement = nextE;
                nextLevelDown = nextL;
                num = n;
        }

        public void setNextElement(Element<K, T> nextE){
                nextElement = nextE;
        }

        public Element<K, T> getNextElement(){
                return nextElement;
        }

        public K getNextElementKey(){
                return nextElement.getKey();
        }

        public void setNextLevelDown(Link<K, T> l){
                nextLevelDown = l;
        }

        public Link<K, T> getNextLevelDown(){
                return nextLevelDown;
        }

        public int getNum(){
                return num;
        }

        public void setNum(int n){
                num = n;
        }
}
```