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;
        }
}
```