1 TreeMap in Java

Based on the information in Java API, we will study TreeMap and related topics so that we can implement MyTreeMap using balanced binary search tree.

TreeMap implements the following interfaces:

Serializable
Cloneable
NavigableMap<K,V>
Map<K,V>
SortedMap<K,V>

and extends
AbstractMap<K,V>

1.1 Map

An object that maps keys to values. A map cannot contain duplicate keys; each key can map to at most one value.

Abstractly a map is an association of two things and these two things are typically called key and value. A map can be viewed as a set of key and value pairs. Another name for a set is a collection, which is the terminology that Java uses. Other views of a map is a collection of keys, a collection of values. Some methods in the map interface could return a collection, which is another interface.

Notice that interface cannot have constructors and therefore cannot specify constructors. An interface provides suggestions for the implementing class about what constructors to have.

Methods in Map<K,V>:

void clear() // optional
boolean containsKey(Object key)
boolean containsValue(Object value)
Set<Map.Entry<K,V>> entrySet()
boolean equals(Object o)
V get(Object key)
int hashCode()
boolean isEmpty()
Set<K> keySet()
V put(K key, V value) // optional
void putAll(Map<? extends K,? extends V> m) // optional
V remove(Object key) // optional
int size()
Collection<V> values()

Note that in the above those methods that modify the map are optional. This could imply the a map object is usually treated as a constant object. Note also within Map there is a interface called Entry<K,V>, which is a static interface as an inner class (see the return type of entrySet() method). Finally note that the classes or interfaces in specifying the return types by above methods: Set<T>, and Collection<T> (see the return types of entrySet(), keySet(), values()).

As it turns out, both Set and Collection are interfaces in Java API. Therefore, we have to understand both interfaces a bit (and get to know which Java classes implement these interfaces if we want to inherit or use some of methods indirectly) if we want to implement all the methods in the Map interface.

API classes that implement this interface include: AbstractMap, ConcurrentHashMap, HashMap, Hashtable, LinkedHashSet, Properties, SimpleBindings, TabularDataSupport, TreeMap.

1.1.1 Map.Entry

Inside the Map<K,V>, a public static interface Map.Entry<K,V> is declared. A map entry is a key-value pair. The Map.entrySet method returns a collection-view of the map, whose elements are of this class. Other methods in TreeMap<K,V> returns a single object of this class. The only way to obtain a reference to a map entry is from the iterator of this collection-view. These Map.Entry objects are valid only for the duration of the iteration; more formally, the behavior of a map entry is undefined if the backing map has been modified after the entry was returned by the iterator, except for the modifications resulted from the setValue operation on the map entry.

Methods in Map.Entry<K,V>:

boolean equals(Object o)
K getKey()
V getValue()
int hashCode()
V setValue(V value) // optional

Note that any class that implements Map should have a inner class, which could be called pair, that implements Map.Entry interface. Then an object of this class or a set of objects of this class could be returned.

1.1.2 Collection

The Collection<E> interface extends Iterable<E> interface. Also it has subinterfaces such as Set, List, Queue, and Deque.
The Collection interface is the root interface in the collection hierarchy in Java API. A collection represents a group of objects, known as its elements. Some collections allow duplicate elements and others do not. Some are ordered and others unordered. The JDK does not provide any direct implementations of this interface: it provides implementations of more specific subinterfaces like Set and List. Note that Set does not allow duplicates and List imposes an “ordering” such as the first element, the second element and associate each element with an index. This interface is typically used to pass collections around and to manipulate them where maximum generality is desired.

Methods in `Collection<E>`:

- `boolean add(E e) // optional`
- `boolean addAll(Collection<? extends E> c) // optional`
- `void clear() // optional`
- `boolean contains(Object o)`
- `boolean containsAll(Collection<?> c)`
- `boolean equals(Object o)`
- `int hashCode()`
- `boolean isEmpty()`
- `Iterator<E> iterator() // from Iterable interface`
- `boolean remove(Object o) // optional`
- `boolean removeAll(Collection<?> c) // optional`
- `boolean retainAll(Collection<?> c) // optional`
- `int size()`
- `Object[] toArray() // array of type Object`
- `<T> T[] toArray(T[] a) // array of type T`

Note that the hashcode method does not define the hashcode of a Collection. This is okay because no class in Java API implements Collection directly. Instead Java API classes implement subinterfaces of the Collection such as Set, where its hashcode is defined as the sum of the hashcodes of its elements. If we want to implement the Collection interface by an inner class of our class (MyTreeMap for instance), we could use the Set definition or our own subject to consistent constraint placed on the hashcode.

API classes ArrayList, LinkedList, Stack, and Vector implement `List<E>`, which extends `Collection<E>`.

### 1.1.3 Set

A Set is a Collection in which no element is duplicated. Set extends Collection, in that Set has exactly the same set of methods as that of Collection. Further stipulations are provided for these methods such hashcode, add, and constructors. For instance, add must make sure the element added is not a duplicate of any element currently in the set.

Set has subinterfaces such as SortedSet and NavigableSet, where elements are ordered by comparisons.

API classes TreeSet and HashSet implement this interface.
1.1.4 Iterator

Iterator<E> is an interface in Java Collection Framework. An iterator over a collection. Iterator takes the place of Enumeration in the Java collections framework. Iterators differ from enumerations in two ways:

- Iterators allow the caller to remove elements from the underlying collection during the iteration with well-defined semantics.
- Method names have been improved.

This interface is a member of the Java Collections Framework.

Methods in Iterator<E>:

- boolean hasNext()
- E next()
- void remove()

API class Scanner implements this interface.

Also any class that implements Iterable<E> interface has a method which returns object of type Iterator<E>. Note Collection interface extends Iterable interface. Therefore, any class (and there are many in the API) that implements Set and List must return an iterator object.

It is very likely that an inner class of these API classes implements Iterator interface and an object of the inner class is returned by the iterator method. It would be nice to obtain an example source code of an API class that implements Collection interface to see if it is really the case.

In general the notion of an iterator is the concept of a “pointer”, through which we can manipulate the underlining data.

Iterator has subinterfaces such as ListIterator, where iteration can be bi-directional and with an index.

1.1.5 List

A List is a Collection in which elements are ordered and allowed to have duplicates. List extends Collection, in that List includes exactly the same set of methods as that of Collection and some methods of its own. Further stipulations are provided for these methods such hashcode, add, and constructors.

Additional methods in List<E>:

- void add(int index, E element)
- boolean addAll(int index, Collection<? extends E> c)
- E get(int index)
- int indexOf(Object o)
- int lastIndexOf(Object o)
- ListIterator<E> listIterator()
- ListIterator<E> listIterator(int index)
- E remove(int index)
- E set(int index, E element)
- List<E> subList(int fromIndex, int toIndex)
API classes ArrayList and LinkedList implement this interface.

1.1.6 Iterable

Implementing this interface allows an object to be the target of the "foreach" statement.

Methods in Iterable<E>:

Iterator<T> iterator()

API classes that implement this interface include: ArrayDeque, ArrayList, ConcurrentHashMap, HashSet, LinkedHashSet, LinkedList, PriorityQueue, Stack, TreeSet, Vector and so on.

1.2 SortedMap

SortedMap<K,V> has a subinterface NavigableMap<K,V> and a superinterface Map<K,V>.

A SortedMap is a Map that further provides a total ordering on its keys. The ordering of the key is determined by the compareTo method if the key object has such method or implements Comparable<E> interface, which is called the natural ordering of its keys, or by a Comparator object typically provided at SortedMap creation time. This order is reflected when iterating over the sorted map’s collection views (returned by the entrySet, keySet and values methods). Several additional operations are provided to take advantage of the ordering. (This interface is the map analogue of SortedSet.)

Methods overridden or introduced in SortedMap<K,V>:

Comparator<? super K> comparator()
Set<Map.Entry<K,V>> entrySet()
K firstKey()
SortedMap<K,V> headMap(K toKey)
Set<K> keySet()
K lastKey()
SortedMap<K,V> subMap(K fromKey, K toKey)
SortedMap<K,V> tailMap(K fromKey)
Collection<V> values()

Notice that the three view methods in Map are overridden in SortedMap. In the overridden methods, the iterators of the Collection (Set or Collection) iterate in ascending key order of entries, keys, and values (the corresponding keys in ascending order).

Note that the new methods include: comparator, headMap (strictly less than toKey), tailMap (keys are greater than or equal to fromKey), and subMap (keys range from fromKey, inclusive, to toKey, exclusive).

API classes that implement this interface include: ConcurrentHashMap, TreeMap.
1.2.1 Comparable

Comparable interface imposes a total ordering on the objects of each class that implements it. This ordering is referred to as the class’s natural ordering, and the class’s compareTo method is referred to as its natural comparison method.

Methods in Comparable<E>:

```
int compareTo(T o)
```

API classes that implement this interface include: BigDecimal, BigInteger, Boolean, Byte, ByteBuffer, Character, CharBuffer, CompositeName, CompoundName, Date, Double, DoubleBuffer, ElementType, Enum, File, Float, FloatBuffer, IntBuffer, Integer, Long, LongBuffer, MemoryType, ObjectName, Short, ShortBuffer, String, Time, TypeKind, URI, UUID, XmlAccessOrder, XmlAccessType and others.

1.2.2 Comparator

A comparison function, which imposes a total ordering on some collection of objects. Comparators can be passed to a sort method (such as Collections.sort or Arrays.sort) to allow precise control over the sort order. Comparators can also be used to control the order of certain data structures (such as sorted sets or sorted maps), or to provide an ordering for collections of objects that don’t have a natural ordering.

Methods in Comparator<E>:

```
int compare(T o1, T o2)
boolean equals(Object obj)
```

API classes that implement this interface include: Collator, RuleBasedCollator.

Note that Object class also has an equals method and every class extends Object. Hence, classes that implement this interface can use the equals method from Object without overriding it.

1.3 NavigableMap

NavigableMap<K,V> has superinterfaces Map<K,V> and SortedMap<K,V> and a subinterface ConcurrentNavigableMap<K,V>. A NavigableMap is a SortedMap that provides descending view of the set of entries, keys, and values. In addition, it also provides methods for returning a Map.Entry object relative to a given key value and methods for returning a key relative a given key value.

Methods overridden or introduced in NavigableMap<K,V>:

```
Map.Entry<K,V> ceilingEntry(K key)
K ceilingKey(K key)
NavigableSet<K> descendingKeySet() // KeySet counterpart
```
NavigableMap<K,V> descendingMap()
Map.Entry<K,V> firstEntry() // SortedMap only has firstKey method
Map.Entry<K,V> floorEntry(K key)
K floorKey(K key)
SortedMap<K,V> headMap(K toKey) // same as that of SortedMap?
// headMap(K toKey) could be implemented by the next method
NavigableMap<K,V> headMap(K toKey, boolean inclusive)
Map.Entry<K,V> higherEntry(K key)
K higherKey(K key)
Map.Entry<K,V> lastEntry() // SortedMap only has lastKey method
Map.Entry<K,V> lowerEntry(K key)
K lowerKey(K key)
NavigableSet<K> navigableKeySet() // diff to KeySet in return type
Map.Entry<K,V> pollFirstEntry() // remove without key
Map.Entry<K,V> pollLastEntry() // remove without key
NavigableMap<K,V> subMap(K fromKey, boolean fromIncl, K toKey, boolean toIncl)
SortedMap<K,V> subMap(K fromKey, K toKey) // same as that of SortedMap,
SortedMap<K,V> tailMap(K fromKey) // same as that of SortedMap
NavigableMap<K,V> tailMap(K fromKey, boolean inclusive)

Notice that SortedMap does not have any method that returns a key and
value entry. In NavigableMap, these methods return an entry: ceilingEntry,
floorEntry, firstEntry, lastEntry, higherEntry, lowerEntry.

Notice that SortedMap has two methods to return a key: firstKey and
lastKey. In NavigableMap, additional methods are provided to return a key:
ceilingKey, floorKey, higherKey, and lowerKey,

Notice that the following methods semantically are basically the same
as that of SortedMap.

SortedMap<K,V> headMap(K toKey)
SortedMap<K,V> subMap(K fromKey, K toKey)
SortedMap<K,V> tailMap(K fromKey)

They have been overridden because they could be developed using the following
methods.

NavigableMap<K,V> headMap(K toKey, boolean inclusive)
NavigableMap<K,V> subMap(K fromKey, boolean fromIncl, K toKey, boolean toIncl)
NavigableMap<K,V> tailMap(K fromKey, boolean inclusive)

In both Map and SortedMap, the only method to delete a key and thus an
entry is removed. In NavigableMap, two new methods are added to remove the
first and last keys: pollFirstEntry() and pollLastEntry().

Finally, descending view of the keys could be obtained via descendingKey-
Set() and descending view of entries and values could be obtained by first using
descendingMap() to get a map and then using entrySet and values methods.

API classes that implement this interface include: ConcurrentSkipListMap
and TreeMap.

7
1.3.1 SortedSet

SortedSet<E> has a subinterface NavigableSet<E> and a superinterface Set<E>.

A SortedSet is a Set that further provides a total ordering on its elements. The ordering of the element is determined by the compareTo method if the element object has such method or implements Comparable<E> interface, which is called the natural ordering of its keys, or by a Comparator object typically provided at SortedSet creation time. This order is reflected when iterating over the elements via an Iterator object. The set’s iterator will traverse the set in ascending element order.

Methods overridden or introduced in SortedSet<E>:

Comparator<? super E> comparator()
E first()
SortedSet<E> headSet(E toElement)
E last()
SortedSet<E> subSet(E fromElement, E toElement)
SortedSet<E> tailSet(E fromElement)

Please note the similarity these methods and that of SortedMap.

1.3.2 NavigableSet

NavigableSet<E> has superinterfaces Set<E> and SortedSet<E> and no subinterface.

A NavigableSet is a SortedSet that provides descending view of the set of elements. In addition, it also provides methods for returning an element object relative to a given element value.

Methods overridden or introduced in NavigableSet<E>:

E ceiling(E e)
Iterator<E> descendingIterator()
NavigableSet<E> descendingSet()
E floor(E e)
SortedSet<E> headSet(E toElement)
NavigableSet<E> headSet(E toElement, boolean inclusive)
E higher(E e)
Iterator<E> iterator()
E lower(E e)
E pollFirst()
E pollLast()
NavigableSet<E> subSet(E fromElement, boolean fromIncl, E toElement, boolean toIncl)
SortedSet<E> subSet(E fromElement, E toElement)
SortedSet<E> tailSet(E fromElement)
NavigableSet<E> tailSet(E fromElement, boolean inclusive)

Notice that SortedSet has two methods to return an element: firstKey and lastKey. In NavigableSet, additional methods are provided to return a key: ceiling, floor, higher, and lower,
Notice that the following methods semantically are basically the same as that of SortedSet.

\[
\text{SortedSet}<E> \text{ headSet}(E \text{ toElement})
\]
\[
\text{SortedSet}<E> \text{ subSet}(E \text{ fromElement}, K \text{ toElement})
\]
\[
\text{SortedSet}<E> \text{ tailSet}(E \text{ fromElement})
\]

They have been overridden because they could be developed using the following methods.

\[
\text{NavigableSet}<E> \text{ headSet}(E \text{ toElement}, \text{ boolean inclusive})
\]
\[
\text{NavigableSet}<E> \text{ subSet}(E \text{ fromElement, boolean fromIncl}, E \text{ toElement, boolean toIncl})
\]
\[
\text{NavigableSet}<E> \text{ tailSet}(E \text{ fromElement, boolean inclusive})
\]

In both Set and SortedSet, the only method to delete an element and thus an element is removed. In NavigableMap, two new methods are added to remove the first and last keys: pollFirst() and pollLast().

Finally, descending view of the elements could be obtained via descendingIterator(), which returns an iterator that iterates in descending order, and descendingSet(), which returns a descending order view via NavigableSet.

API classes that implement this interface include: ConcurrentSkipListSet and TreeSet.

### 1.4 AbstractMap

This class provides a skeletal implementation of the Map interface, to minimize the effort required to implement this interface.

One of the key idea used in the design and implementation is to implement other methods based on the abstract method entrySet. For a map that cannot be modified, that is built or initialized from the constructor, the class that extends this class only needs to get entrySet method implemented. It is a good exercise to read the API and to make sure this design makes sense.

To implement a modifiable map, the programmer must additionally override this class’s put method (which otherwise throws an UnsupportedOperationException), and the iterator returned by entrySet().iterator() must additionally implement its remove method.

It is instructive to learn the design idea used in the class.

#### 1.4.1 AbstractMap.SimpleEntry

Within AbstractMap a public static class AbstractMap.SimpleEntry<K,V> is defined to create object of Map.Entry<K,V>. This class implements Serializable and Map.Entry<K,V> interfaces. It has the following constructors:

\[
\text{AbstractMap.SimpleEntry}(K \text{ key, V value})
\]
\[
\text{AbstractMap.SimpleEntry}(\text{Map.Entry?< extends K,? extends V> entry})
\]

It has the following methods (see how these methods compare to those in Map.Entry and Object):
boolean equals(Object o)  
K getKey()  
V getValue()  
int hashCode() // bitwise XOR two hashcodes  
V setValue(V value)  
String toString()

1.4.2 AbstractMap.SimpleImmutableEntry

This public static class is similar to AbstractMap.SimpleEntry<K,V>, except that AbstractMap.SimpleImmutableEntry<K,V> does not “implement” setValue method. The implementation of this method simply throws UnsupportedOperationException, as this class implements an immutable map entry.

1.5 Java API TreeMap

The TreeMap<K,V> class in Java API guarantees \( \log(n) \) time cost for the containsKey, get, put and remove operations. This requirement implies balanced binary search tree implementation is needed as the underlining data structures. For instance, AVL trees or Red-Black trees.

It has the following constructors:

TreeMap()  
TreeMap(Comparator<? super K> comparator)  
A new, empty tree map, ordered according to the given comparator.  
TreeMap(Map<? extends K,? extends V> m)  
A new tree map containing the same mappings as the given map, natural ordering.

TreeMap(SortedMap<K,? extends V> m)  
A new tree map, using the same ordering as the specified sorted map.

It has the following methods (see how these methods compare to those in NavigableMap interface). Note that methods inherited from Object and AbstractMap are not listed.

Map.Entry<K,V> ceilingEntry(K key)  
K ceilingKey(K key)  
void clear()  
Object clone()  
Comparator<? super K> comparator()  
boolean containsKey(Object key)  
boolean containsValue(Object value)  
NavigableSet<K> descendingKeySet()  
NavigableMap<K,V> descendingMap()  
Set<Map.Entry<K,V>> entrySet()  
Map.Entry<K,V> firstEntry()  
K firstKey()  
Map.Entry<K,V> floorEntry(K key)
K floorKey(K key)
V get(Object key)
SortedMap<K,V> headMap(K toKey)
NavigableMap<K,V> headMap(K toKey, boolean inclusive)
Map.Entry<K,V> higherEntry(K key)
K higherKey(K key)
Set<K> keySet()
Map.Entry<K,V> lastEntry()
K lastKey()
Map.Entry<K,V> lowerEntry(K key)
K lowerKey(K key)
NavigableSet<K> navigableKeySet()
Map.Entry<K,V> pollFirstEntry()
Map.Entry<K,V> pollLastEntry()
V put(K key, V value)
void putAll(Map<? extends K,? extends V> map)
V remove(Object key)
int size()
NavigableMap<K,V> subMap(K fromKey, boolean fromIncl, K toKey, boolean toIncl)
SortedMap<K,V> subMap(K fromKey, K toKey)
SortedMap<K,V> tailMap(K fromKey)
NavigableMap<K,V> tailMap(K fromKey, boolean inclusive)
Collection<V> values()

In the discussion section, we will consider the question of how to implement MyTreeMap (i.e. TreeMap) using the balanced binary search tree of our previous project.

1.6 Object

Class Object is the root of the class hierarchy. Every class has Object as a superclass. All objects, including arrays, implement the methods of this class.

Methods introduced in Object:

protected Object clone()
boolean equals(Object obj)
protected void finalize()
Class<?> getClass()
int hashCode()
void notify()
void notifyAll()
String toString()
void wait()
void wait(long timeout)
void wait(long timeout, int nanos)

Five methods are related to the execution thread: three waits and two notifies.
The clone method is to make a copy, which could be a deep copy or a shallow copy when the method is overridden. Without overriding the method, this method creates a new instance and does a field by field copy via assignment, which is a shallow copy. Any class (which is a subclass of Object) must implement Cloneable interface, an interface that does not have any method, in order to use the Object’s clone method otherwise exception is thrown. The intent for clone is to make a “deep” copy, that is the modification of one should not change the other.

The equals method for testing if this object is the “same” as the argument object. The notion of the same is an equivalence class (reflective, symmetric, and transitive). The equals method for class Object implements the most discriminating possible equivalence relation on objects; that is, for any non-null reference values x and y, this method returns true if and only if x and y refer to the same object (x == y has the value true). The method could be overridden to mean that the two objects have the same value.

The finalize method is called by garbage collector when the object is no longer referenced by any. The finalize method of class Object performs no special action; it simply returns normally. Subclasses of Object may override this definition typically for releasing system resources.

The hashCode method gives the hash code, a numerical value, of the object. The actual numerical returned by class Object is implementation (Java Virtual Machine) dependent. However, it should be always true that if two objects are the same via equals method, the hashCode of the two object must be identical. Ideally, if two objects are not the same via equals method, the hashCodes are different. Note that this semantics is not required and many overridden methods cannot ensure it (see the definition of hash code for Set). Nonetheless, as much as is reasonably practical, the hashCode method defined by class Object does return distinct integers for distinct objects. (This is typically implemented by converting the internal address of the object into an integer, but this implementation technique is not required by the JavaTM programming language.)

The intent of the toString method is to produce natural language text for human to “understand” the information in an object, which could be used for debugging purpose. It is recommended that all subclasses override this method. The toString method for class Object returns a string consisting of the name of the class of which the object is an instance, the at-sign character ‘@’, and the unsigned hexadecimal representation of the hash code of the object. In other words, this method returns a string equal to the value of:

\[
\text{getClass().getName()} + \ '@' + \text{Integer.toHexString(hashCode())}
\]

The getClass method is an interesting method. In Java during run time, classes are loaded. There is a run time class when a class is loaded. This method returns the runtime class of this Object. The returned Class object is the object that is locked by static synchronized methods of the represented class. Here is an example:

```java
Number n = 0;
```
```java
Class<? extends Number> c = n.getClass();
System.out.println(c.getName());
```

The API `Class<T>` has many many methods! One of the method `getName` is used above.

### 1.7 Discussion

The intent of this discussion is to help us connect project two and project three. Even though, due to the limited amount of time we can allocate at the end of the semester to the last project, a small, relative simple subset of the methods in `TreeMap` is required to be implemented in `MyTreeMap` of project 3, the discussion has the goal of implementing all methods in mind. Some of them are more interesting and challenging from a learning viewpoint.

#### 1.7.1 Using the TreeMap Design

As indicated in project 3, the design of `MyTreeMap` must follow that of the `TreeMap` in Java API. Hence, `MyTreeMap` should have the following declaration or definition.

```java
class MyTreeMap<K,V> extends AbstractMap<K,V>
    implements NavigableMap<K,V>{
    // here is our code
}
```

#### 1.7.2 Balanced Binary Search Tree of Project 2

In project two we have developed two kinds of balanced binary search trees: AVL trees and Red-Black trees. Suppose that we have made both classes generic classes, where the data class (type of data) is the generic argument. Let us assume the data class implements `Comparable` interface, that is it has a `compareTo` method. Here are examples of using the balanced binary search trees of project 2:

```java
AVL_tree<Integer> atree = new AVL_tree<Integer>();
Red_Black_tree<String> rbtree = new Red_Black_tree<String>();
atree.insert(16);
rbtree.insert("Hello");
```

In project 2, we have the following public methods:

```java
public search(int key)
public bool insert(int key)
public bool delete(int key)
public void show()
public int height()
public int size()
public bool check()
```
The above are adjusted in project 3 as follows, where T is the generic type of our tree.

public T search(T entry) //return null if not found  
public T insert(T entry)  
public T delete(T entry)  
public void show()  
public int height()  
public int size()  
public bool check()  
// new methods  
public T[] inorder() // return an array of reference to entry in inorder  
public T least() // return the least entry  
public T floor(T entry) // return the largest entry <= the given entry or null  
// more are needed to implement the complete TreeMap or MyTreeMap

1.7.3 Inner Data Structures and Classes
The structure of the MyTreeMap is as follows.

class MyTreeMap<K,V> extends AbstractMap<K,V> implements NavigableMap<K,V>{

    // inner classes
    // 1) MySimpleEntry<K,V>
    // 2) other classes

    // inner data structures
    // 1) AVL_tree<Map.Entry<K,V>>
    // 2) other private data

    // methods in Map
    // here we have a choice to
    // 1) implement entrySet method and let AbstractMap do the rest
    // 2) implement all including entrySet (more work but more efficient?)

    // methods in NavigableMap interface
}

1.7.4 Map.Entry and AbstractMap.SimpleEntry
Map.Entry interface is introduced in Map and a SimpleEntry class is introduced in AbstractMap to implement the interface. We hope to extend the SimpleEntry class so it has a compareTo method, where the compareTo method of the key is used to compare two Map.Entry. Once we have that, we can use the new class as the generic argument for our balanced binary search tree classes.

class MyTreeMap<K,V> extends AbstractMap<K,V>
implements NavigableMap<K,V>{

// an inner class for tree node data part
class MySimpleEntry<K,V> extends AbstractMap.SimpleEntry<K,V>
    implements Comparable<Map.Entry<K,V>>{
    MySimpleEntry(K key, V value){super(key,value);}
    public int compareTo(Map.Entry<K,V> o){
        // can we not cast to Comparable?
        return ( (Comparable)o.getKey() ).compareTo( getKey());
    }
}
// inner data structures
AVL_tree<Map.Entry<K,V>> tree;
}

1.7.5 Simple MyTreeMap methods

With the above code in place, some methods could be implemented as follows. The semantics of the methods are given in Java API TreeMap. It is assumed that you understand the semantics of these methods.

class MyTreeMap<K,V> extends AbstractMap<K,V>
    implements NavigableMap<K,V>{

    // methods examples
    int size(){ return tree.size();}
    public V get (Object key){
        Map.Entry<K,V> x = tree.search(new MySimpleEntry<K, V>((K)key , null));
        if (x == null)
            return null;
        else
            return x.getValue();
    }

    public V put (K key, V value){
        Map.Entry<K, V> y = new MySimpleEntry<K, V>(key , value);
        Map.Entry<K, V> x = tree.search(y);
        if (x == null){ // key is not in
            x = tree.insert(y);
            return x.getValue();
        } else {
            x.setValue(value);
        }
    }
}

1.7.6 View methods

In Map as well as NavigableMap, the public methods provide different “views” of the underlining data structures for the map. For instances, in Map, we have
entrySet and values methods, which provide a `Set` (it is an interface) view of the underlining mapping pairs (entries) and a `Collection` view of the values in the underlining mapping pairs.

To develop those methods, we need to first develop an inner class that implements the view interface then in the method create an object of that class and return. The returned object in effect gives us the “view”. Since some of the views can return an iterator via a method in the view, we then have to develop an inner class, within the view class, that implements the `Iterator` interface. Then in the method that returns an Iterator object, we create an object and return. In a high level, the above are the ideas. The challenges now reduce to how to implement these classes.

The following illustrates how this could be done.

class MyTreeMap<K,V> extends AbstractMap<K,V>{
    implements NavigableMap<K,V>{
        // inner class for entrySet view
        class Return_entrySet<K,V> implements Set<Map.Entry<K,V>>{
            private AVL_tree<Map.Entry<K,V>> t;
            // inner class with the view class for iterator
            class IteratorArray<T> implements Iterator<T>{
                private int i;
                private T[] a;
                private AVL_tree<T> tree;
                IteratorArray(AVL_tree<T> t){
                    a = t.inorder();
                    i=0;
                    tree = t;
                }
                private void reset() { // some code is missing
                    // save the current entry
                    a = tree.inorder();
                    // reset i using saved entry key
                }
                public boolean hasNext(){ return i < a.length;}
                public T next(){ return a[i++];}
                public void remove() {tree.delete(a[i-1]); reset();}
            }
            // constructor for view class
            Return_entrySet (AVL_tree<Map.Entry<K,V>> tree) {
                t = tree;
            }
            // some methods of the view class
            public Iterator<Map.Entry<K,V>> iterator(){
                return new IteratorArray<Map.Entry<K,V>>(tree);
            }
        }
    }
}
public boolean add(Map.Entry<K,V> e){
    boolean flag = false;
    Map.Entry<K,V> v = t.search(e);
    if (v == null){
        t.insert(e);
        flag = true;
    } else if ( !v.getValue().equals( e.getValue() ) ) {
        v.setValue( e.getValue() );
        flag = true;
    }
    return flag;
}

// other methods of the set interface

// inner data for MyTreeMap
AVL_tree<Map.Entry<K,V>> tree;

// constructor
MyTreeMap(){
    tree = new AVL_tree<Map.Entry<K,V>>() ;
}

// AbstractMap uses this method, that is, the iterator
// returned from method of the object returned from this
// method to implement other Map methods!
public Set<Map.Entry<K,V>> entrySet(){
    return new Return_entrySet<K,V>(tree);
}

} Note that in the above example, the IteratorArray class does not implement the Iterator interface in the most efficient way. Note also the add method in the Set interface adds the entry directly into the underlining data structures or the tree! Hence, the Set view is really a view through which the underlining data structures are looked and changed. Finally note that how entrySet method in MyTreeMap (or TreeMap) and iterator method in Return_entrySet are developed.

2 C++ map

In Java the TreeMap class extends and implements other class (AbstractMap) and interface (NavigableMap), which have already defined in Java API. In C++ the map class does not inherit from any class nor implements any interface that has been defined (C++ language does not have the feature of “interface”). So from the class or interface hierarchy aspect, to mimic map class in C++ is less involved than to mimic TreeMap in Java.
Nonetheless the design of map class does use other classes in the standard library and we must understand these classes just as we must understand Java interfaces and classes related to TreeMap.

The classes that map class uses are pair, iterator, less, allocator. Since C++ allows operator overloading, some operators are also overloaded for these classes.

In its implementation in the C++ Standard Template Library, map containers take four template parameters as shown:

```cpp
template < class Key, class T, class Compare = less<Key>,
          class Allocator = allocator<pair<const Key,T> > > class map;
```

The last two parameters have default values. The template parameters have the following meanings:

- **Key**: Type of the key values. Each element in a map is uniquely identified by its key value.
- **T**: Type of the mapped value. Each element in a map is used to store some data as its mapped value.
- **Compare**: Comparison class: A class that takes two arguments of the key type and returns a bool. The expression `comp(a,b)`, where `comp` is an object of this comparison class and `a` and `b` are key values, shall return true if `a` is to be placed at an earlier position than `b` in a strict weak ordering operation. This can either be a class implementing a function call operator or a pointer to a function (see constructor for an example). This defaults to `less<Key>`, which returns the same as applying the less-than operator (`a<b`). The map object uses this expression to determine the position of the elements in the container. All elements in a map container are ordered following this rule at all times.
- **Allocator**: Type of the allocator object used to define the storage allocation model. By default, the allocator class template is used, which defines the simplest memory allocation model and is value-independent.

### 2.1 C++ map class

From the outside looking at C++ map, we see that map provides a group of types we could use. For example,

```cpp
map<char,int>::iterator it;
```

where `iterator` is a type that `map<char,int>` has and we can use it from the outside. In the above, we use it to define a variable `it` of which the type is the `iterator` type defined and declared in `map<char,int>`. The actual class of this type is an inner class of the map and its development is part of the implementation of the map class. In concept, this is similar to what we have in the MyTreeMap discussion where we introduced (developed) `Return_entrySet` class and `IteratorArray` class.

We also see that map provides a group of methods we could use. For example in the following,
// map::begin/end
#include <iostream>
#include <map>
using namespace std;

int main ()
{
    map<char,int> mymap;
    map<char,int>::iterator it;

    mymap['b'] = 100;
    mymap['a'] = 200;
    mymap['c'] = 300;

    // show content:
    for ( it=mymap.begin() ; it != mymap.end(); it++ )
        cout << (*it).first << " => " << (*it).second << endl;

    return 0;
}

four methods of the map are used. They are the default constructor:
    map<char,int> mymap;
the begin() method in the part of the for loop:
    it=mymap.begin() ;
the end() method in the part of the for loop:
    it != mymap.end()
and the overloaded operator (which is in fact a method) [ ] method:
    mymap['b'] = 100;

    Hence, to mimic map, Mymap should also provide an iterator type as well
    as default constructor, begin(), end() and operator [ ] with the same seman-
    tics and syntax. Once we have our template Mymap class, we can replace
    map<char,int> by Mymap<char,int> and the above code should behave ex-
    actly the same as before.

    The example also illustrate another aspect that an iterator class support. For
    instance, we can use ++ and * with iterator as in the for loop. The semantics
    and syntax of the map iterator should support the bidirectional iterator defined
    in the standard library of C++.

2.1.1 map types

The template class map as shown:

template < class Key, class T, class Compare = less<Key>,
        class Allocator = allocator<pair<const Key,T>> > class map;

has the following types:
The definition of a type as an alias is done via `typedef` keyword in C++. For example,

```cpp
typedef Key key_type;
```

is a statement in the map class to make the definition of the first entry in the above.

Note that two template classes are used in the table: `pair` and `reverse_iterator`. These classes are part of the standard library in C++. There are three types for which we have to develop the classes or inner classes for them: `value_compare`, `iterator`, and `const_iterator`.

### 2.1.2 map constructors

The template class map as shown:

```cpp
template < class Key, class T, class Compare = less<Key>,
           class Allocator = allocator<pair<const Key,T> > > class map;
```

has the following three constructors.

```cpp
explicit map ( const Compare& comp = Compare(),
               const Allocator& = Allocator() );
```

The above could be the default constructor (since the two arguments have default values), which constructs an empty map. This constructor cannot be used to convert a single object (of type `Compare`) to a map, because of the keyword `explicit`. Note that this constructor could be treated as a single argument constructor (the second argument using the default). In C++, single argument constructor is used implicitly to convert object of the argument type to object of the type of the class where the constructor belongs unless `explicit` keyword is used.
template <class InputIterator>
map ( InputIterator first, InputIterator last,
     const Compare& comp = Compare(), const Allocator& = Allocator() );

This constructor takes a pair of iterators (whose type is the template argument) to pair and uses them to build the map. This constructor (which is a function) is a template function. In C++, container classes, such as vector and map, provide iterators (a-particular-container::iterator is the iterator type) begin() and end(). If the container contains pairs, then the two iterators could be passed to the constructor to build the map. The implementation of the constructor could use a loop and tree insert method of our project 2.

map ( const map<Key,T,Compare,Allocator>& x );

This is the copy constructor. Notice that the constructor has a single argument and the its type is the class reference (map<&...&>). This is how the compiler and programmer recognize copy constructor. The compiler provides a default copy constructor if no copy constructor is provided within a class. The default copy constructor performs a bit-wise copy of the two objects.

2.2 pair

C++ library (in header <utility>) introduces a pair type or class to combine two pieces of data into a single element. The type is defined as a template class. This class couples together a pair of values, which may be of different types (T1 and T2). The individual values can be accessed through the public members first and second.

The class is defined as:

```
template <class T1, class T2> struct pair
{
    typedef T1 first_type;
    typedef T2 second_type;

    T1 first;
    T2 second;

    pair() : first(T1()), second(T2()) {}
    pair(const T1& x, const T2& y) : first(x), second(y) {}
    template <class U, class V>
    pair (const pair<U,V> &p) : first(p.first), second(p.second) { }
};
```

As can be seen from this definition, the class has three constructors. Since the fields first and second are public members, we may access them directly. Even though the class does not overloaded operators for comparing pairs, the library overloads those global operators in the header <utility>, including the relational operators ==, <, !=, >, >= and <=.
To compare pair objects of the same type directly, the following rules are used:

- Two pair objects are compared equal if the first elements in both objects compare equal to each other and both second elements compare equal to each other - they all have to match.

- In inequality comparisons (<, >), only the first element is compared, except if both first elements compare equal to each other, in this case only the second element is taken into consideration for the comparison operation.

A **make_pair** template function is defined in `<utility>` that can be used to make an object of type pair. This function is defined as:

```cpp
template <class T1, class T2>
pair<T1, T2> make_pair (T1 x, T2 y)
{
    return (pair<T1, T2>(x, y));
}
```

The following is a sample code using `pair` and `make_pair`:

```cpp
#include <iostream>
#include <utility>
#include <string>
using namespace std;

int main () {
    pair<string, double> product1 ("tomatoes", 3.25);
    pair<string, double> product2;
    pair<string, double> product3;

    product2.first = "lightbulbs";       // type of first is string
    product2.second = 0.99;              // type of second is double

    product3 = make_pair ("shoes", 20.0);

    cout << "The price of " << product1.first << " is $" << product1.second << "\n";
    cout << "The price of " << product2.first << " is $" << product2.second << "\n";
    cout << "The price of " << product3.first << " is $" << product3.second << "\n";
    return 0;
}
```

After making the trees in project two as generic, we could use pair type for the trees as shown:

```cpp
template<class Key, 
    class T,
```
class Compare = less<Key>

class Allocator = allocator<pair<const Key, T>> map {

private:
  AVL_tree<pair<const Key, T>> tree;
}

Notice that the meaning of < for pair is not what we would like to compare
the key values. We might need to use first field or redefine < for pair, where
inside the redefined function we use first field and a Compare object.

2.3 less and its base

Notice that one of the default value for the template argument Compare is
less<Key>, which is a template class with template argument Key. In the class
less, the operator () is overloaded. The operator () is the function call
operator. Notice the syntax of calling a function. The object of less is called
functional object, because obj(arg1, agr2) notation is the same as a function
call. Remember that overloaded operator is a notation for calling a member
function or a function. The actual definition of less is as follows:

template <class T> struct less : binary_function <T,T,bool> {
  bool operator() (const T& x, const T& y) const
  {return x<y;}
};

Note that the above assumes that < is overloaded for type T either as a
member function or globally.

The base class binary_function, which is a base class for standard binary
function objects, is defined in the header <functional> and so is less.

Generically, function objects are instances of a class with member function
operator() defined. This member function allows the object to be used with the
same syntax as a regular function call, and therefore it can be used in templates
instead of a pointer to a function.

In the case of binary function objects, this operator() member function takes
two parameters and returns a value. So there are three types in the template.

The binary_function is just a base class, from which specific binary func-
tion objects are derived. It has no operator() member defined (derived classes
are expected to define this) - it simply has three public data members that are
typedefs of the template parameters. It is defined as:

template <class Arg1, class Arg2, class Result>
struct binary_function {
  typedef Arg1 first_argument_type;
  typedef Arg2 second_argument_type;
  typedef Result result_type;
};

23
2.4 iterator

In C++, iterators are pointers. One way to understand iterators is to think of pointers to arrays, where we can dereference it (*ptr) to get the value or object; we can increment it (ptr++) to get to the next element; and so on. On an abstract level, iterators behave like a point, where we can dereference it (*iter) to get the value or object; we can increment it (iter++) to get to the next element; and so on. In actuality, however, unlike pointers to arrays the next element may not be located in the next physical memory. The code or logic behind an iterator needs to know where the next element is to achieve iter++. This aspect in general is of no concern to the user.

Depending on what operators are available for iterators, C++ defines 5 categories of iterators: input, output, forward, bidirectional, and random access. The iterators returned from the map methods are bidirectional iterators, meaning both ++ (forward) and -- (backward) are defined and we may move in either direction.

2.5 reverse_iterator

The begin() method returns an iterator object and rbegin() method returns a reverse_iterator object. The basic idea of a reverse iterator is to traverse the elements in reverse order, but still use ++ to move to the next element (in the reverse order). To completely implement Mymap as map, in addition to an inner iterator class, another inner class for reverse iterator is needed. Even though it is not required for our project.

Here is a sample code using reverse iterator and note that the usage of ++ to move to the next element.

```cpp
// map::rbegin/rend
#include <iostream>
#include <map>
using namespace std;

int main ()
{
    map<char,int> mymap;
    map<char,int>::reverse_iterator rit;
    mymap['x'] = 100;
    mymap['y'] = 200;
    mymap['z'] = 300;

    // show content:
    for ( rit=mymap.rbegin() ; rit != mymap.rend(); rit++ )
        cout << rit->first << " = " << rit->second << endl;

    return 0;
}
```
Here is the output:

\[
\begin{align*}
  z & \Rightarrow 300 \\
  y & \Rightarrow 200 \\
  x & \Rightarrow 100 \\
\end{align*}
\]

2.6 Discussion

The principle ideas in the discussion section related to Java could be used here as well. Please refer to the discussion section for Java (1.7 or page 13).

In the Java discussion, we add a compareTo method to MySimpleEntry, which extends AbstractMap.SimpleEntry to allow minimal modification of the binary tree code. Another way to address the same issue, which could be used in C++, is to view the data part of a tree node having two components key (first or getKey()) and value (second or getValue()). The key part is used in binary tree search, insert, and delete. We need to modify the tree code accordingly. In this approach, a data field always has two components. In our project 2, the value component happens to be empty.

3 Summary

To be added later.