CS 635 Advanced Object-Oriented Design & Programming
Fall Semester, 2021
Doc 15 Assignment 2
Nov 16, 2021

NullObject

In Btree

```
private String toStringBuilder() {
        Node currentNode = this;
        StringBuilder stringBuilder = new StringBuilder() ;
        toString(currentNode, stringBuilder);
        return  stringBuilder.toString();
}

private void toString(Node currentNode, StringBuilder stringBuilder) {
         if (currentNode.isNull()>0) {
            toString(currentNode.childrenNode[0], stringBuilder);
           if (currentNode.keyNode[0] != null) {
            stringBuilder.append((E) currentNode.keyNode[0]);
           }
           toString(currentNode.childrenNode[1], stringBuilder);
           if (currentNode.keyNode[1] != null) {
            stringBuilder.append((E) currentNode.keyNode[1]);
           }
           toString(currentNode.childrenNode[2], stringBuilder);
        }
    }
```

In Btree

```
private String toStringBuilder() {
        StringBuilder stringBuilder = new StringBuilder() ;
        root.toString( stringBuilder);
        return  stringBuilder.toString();
}
in Node
private void toString(StringBuilder stringBuilder) {
         if (this.isNull()>0) {
                this.childrenNode[0].toString(stringBuilder);
                if (this.keyNode[0] != null) {
                   stringBuilder.append((E) this.keyNode[0]);
                }
                this.childrenNode[0].toString(stringBuilder);
                if (this.keyNode[1] != null) {
                   stringBuilder.append((E) this.keyNode[1]);
                }
                this.childrenNode[2].toString(stringBuilder);
            }
      }
```

```
private Node getNode(E currentKey, Node traversingNode) {
    if (traversingNode.childrenNode[0] instanceof NullNode)
        return traversingNode;


private Node getNode(E currentKey, Node traversingNode) {
    if (traversingNode.childrenNode[0].isNull())
        return traversingNode;
```

In Btree

```
private String toStringBuilder() {
        StringBuilder stringBuilder = new StringBuilder() ;
        root.toString( stringBuilder);
        return  stringBuilder.toString();
}
```

in NullNode

```
private void toString(StringBuilder stringBuilder) {
}
```

in Node

```
private void toString(StringBuilder stringBuilder) {
    this.childrenNode[0].toString(stringBuilder);
    if (this.keyNode[0] != null) {
          stringBuilder.append((E) this.keyNode[0]);
     }
    this.childrenNode[1].toString(stringBuilder);
    if (this.keyNode[1] != null) {
          stringBuilder.append((E) this.keyNode[1]);
     }
     this.childrenNode[2].toString(stringBuilder);
}
```

In BTree

```
    def __iter__(self):
        """
        return External iterator for searching through the BTree in-order
        """
        for item in self.__root.yieldNext():
            yield item.key
```

In Node

```
    def yieldNext(self):
        """
        Yield's as the Btree is traverse in-order
        """
        for index, child in enumerate(self.children):
            if type(child) != NullNode:
                yield from child.yieldNext()
            if len(self.partitionList) > index: # adding nodes keys to the list
                yield self.partitionList[index]
```

Python

```
class BTree(Mapping):

    def __init__(self):
        self.__root = NullNode()
        self.__length = 0

    def __repr__(self):
        return repr(self.__root)

    def __iter__(self):
        for pair in self.__root:
            yield pair.get_value()
```

```
class BTreeNode:

    def __iter__(self):
        yield from self.__children[0]
        if len(self.__values) > 0:
            yield self.__values[0]
        yield from self.__children[1]
        if len(self.__values) == 2:
            yield self.__values[1]
            yield from self.__children[2]
```

```
class NullNode:

    def __iter__(self):
        return
        yield
```

In BTree

```
private void reverseIterator(Node currentNode, Consumer<? super E> action) {
  for (int index = currentNode.getKeySize() - 1; index >= 0; index--) {
    if (!currentNode.isLeaf()) {
      Node child = currentNode.getChild(index + 1);
      if (!child.isNullNode()) {
        reverseIterator(child, action);
      }
    }
    action.accept(currentNode.getKey(index));
  }
  if (currentNode.getChildrenSize() != 0) {
    reverseIterator(currentNode.getChild(0), action);
  }
}
```

```
// Why not just use
if (child != null)
```

In BTree

```
private void reverseIterator(Node currentNode, Consumer<? super E> action) {
    root.reverseIterator(Consumer<? super E> action);
}
```

In Node

```
private void reverseIterator(Consumer<? super E> action) {
  for (int index = this.getKeySize() - 1; index >= 0; index--) {
    if (!this.isLeaf()) {
      Node child = this.getChild(index + 1);
      if (!child.isNullNode()) {
       child.reverseIterator(action);
      }
    }
    action.accept(this.getKey(index));
  }
  if (this.getChildrenSize() != 0) {
    this.getChild(0).reverseIterator( action);
  }
}
```

```
// Why not just use
if (child != null)
```

Check makes sure that we don't call this on NullNode

This check makes sure we don't call this on a NullNode

In BTree

```
private void traverse(Node current) {
    if (current.isLeaf()) {
        students[++count] = current.getLeftElement();
        if (!current.getRightElement().isNull()) {
            students[++count] = current.getRightElement();
        }
    } else {
        traverse(current.getLeftNode());
        students[++count] = current.getLeftElement();
        traverse(current.getMidNode());

        if (!current.getRightElement().isNull()) {
            if (!current.isLeaf()) {
                students[++count] = current.getRightElement();
            }
            traverse(current.getRightNode());
        }
    }
}
```

II

In BTree

```
public ArrayList<Student> reverseTraversal(Node currentNode, ArrayList<Student> reverseArray) {
    if (currentNode.isNull()) {
        return reverseArray;
    }
    reverseArray = reverseTraversal(currentNode.getChildren()[2], reverseArray);
    if (!currentNode.elementNull(currentNode.getKeys()[1])) {
        reverseArray.add(currentNode.getKeys()[1]);
    }
    reverseArray = reverseTraversal(currentNode.getChildren()[1], reverseArray);
    reverseArray.add(currentNode.getKeys()[0]);
    reverseArray = reverseTraversal(currentNode.getChildren()[0], reverseArray);
    return reverseArray;
}
```

```
@Override
public void forEach(Consumer action) {
    root.forEach(action);
 }


Class Node {
    public void forEach(Consumer action) {
        left.forEach(action);
        action.accept(leftStudent);
        middle.forEach(action);
        if rightStudent != nil {
            action.accept(rightStudent);
        }
        left.forEach(action);
    }
```

```
Node<E> findNode(E element) {
    if(this.childrenCount == 0) {  return this;
    }else {
        int index = 0;
        while(index < this.keysCount && strategy.compare(element,this.keys[index]) > 0 ){
            index++;
        }
        return this.children[index].findNode(element);
    }
}
```

```
Node<E> findNode(E element) {
        int index = 0;
        while(index < this.keysCount && strategy.compare(element,this.keys[index]) > 0 ){
            index++;
        }
        return this.children[index].findNode(element);
    }
}
```

NullNode

```
Node<E> findNode(E element) {
   return this.parent;
}
```

```
public Student[] toArray() {
    students = new Student[studentCount];
    if (!isEmpty())
        traverse(root);
    return students;
}
```

How is students changed?

students is a field that should be a local variable passed to traverse

nodeArray

```
public class BTree<K, V> extends AbstractMap<K, V> implements SortedMap<K, V> {
    private final Comparator<? super K> comparator;
    private final int ORDER = 3;
    private transient Entry<K, V> root;
    /**
     * array to display elements of BTree can be deleted
     */
    private static List<Object> nodeArray = new ArrayList<>();
```

static

Only one for all instances

Just return this from a method

Why store it?

Strategy

```
public interface Strategy <Key extends Comparable<Key>, Value> {
        public int compare(DataNode<Key, Value> dataNode1, DataNode<Key, Value> dataNode2);
}
```

```
    private StudentSortingStrategy sortingStrategyEvaluator(BtreeNode currentNode, String
sortingTechnicSelected) {

        StudentSortingStrategy studentSorting;

        if(sortingTechnicSelected.equalsIgnoreCase("SortByGpa")){
            studentSorting = new StudentSortingByGpa();
        }else {
            studentSorting = new StudentSortingByName();
        }

        return studentSorting;
    }
```

No need for this

```
public class OrderingContext {
    private OrderingStrategy strategy;

    public OrderingContext(OrderingStrategy strategy) {
        this.strategy = strategy;
    }

    public Comparator<Student> getOrderingStrategy(){
        return strategy.order();
    }
}
```

In BTree

```
public void setOrderingStrategy(String orderingStrategy) {
    switch (orderingStrategy) {
        case "lexicographicOrder" -> context = new Context(new LexicographicOrder());
        case "gpaOrder" -> context = new Context(new GpaOrder());
    }
 }
```

A new strategy now requires editing this method in the BTree

Adding a new strategy should not require editing the BTree

Names

private final class Iter implements Iterator<E> {

private final class BTreeIterator implements Iterator<E> {

```
rew name is wrong  you don't return a StringBuilder
  public String toStringBuilder() {
        BTreeNode current = this;
        string = "";
        return toStringBuilder(current);
        }

  public String toStringBuilder(BTreeNode node) {
    for (int i = 0; i < node.getKeysSize(); i++) {
      string += node.keys[i].toString();
    }
    for (int i = 0; i < node.getKeysSize() + 1; i++) {
      toStringBuilder(node.childNodes[i]);
    }
    return string;
  }
```

In BTree Class

```
private void reverseTraverseWithAction(BTreeNode<T> currNode, Consumer<? super T> action) {
        for (int i = currNode.getRecordCount(); i>=0; i--) {
                if (i<currNode.getRecordCount()) {
                        action.accept(currNode.getRecordAt(i));
                }
                reverseTraverseWithAction(currNode.getChildAt(i), action);
        }
}
```

```
def for_each(self, selection_condition):
    '''
    Implements an internal iterator traversing B-Tree in reverse order

    selection_condition: condition that is being applied to the keys' value
    '''
    i = len(self.children)-1
    j = len(self.keys) - 1
    while i >= 0:
        if self._object_comparator.compare(self.children[i].keys[0], self.keys[j]) > 0:
            self.children[i].for_each(selection_condition)
        else:
            break
        i = i - 1
```

```
public Object[] toArray() {
        Iterator<T> iter = this.iterator();
        Object[] objArray = new Object[this.size()];
        int i = 0;
        while(iter.hasNext()) {
                objArray[i] = iter.next();
                i++;
        }
        return objArray;
}
```

```
void TestExternalIterator() {
        Iterator<Student> iter = nameTree.iterator();
        Student prev;
        Student next = iter.next();
        while (iter.hasNext()) {
                prev = next;
                next = iter.next();
                if (prev.getName().compareTo(next.getName()) < 0) { /* pass */ }
                else fail();
        }
        Iterator<Student> iter1 = reverseNameTree.iterator();
        next = iter1.next();
        while (iter1.hasNext()) {
                prev = next;
                next = iter1.next();
                if (prev.getName().compareTo(next.getName()) > 0) { /* pass */ }
                else fail();
        }
        Iterator<Student> iter2 = gpaTree.iterator();
        next = iter2.next();
        while (iter2.hasNext()) {
                prev = next;
                next = iter2.next();
```

```
public List<Object> getTreeElementsInOrder( List<Object> inOrderTraversedElements ) {
    BtreeNode current = this.root;
    return current.getTreeElementsInOrderTraversing(current, inOrderTraversedElements);
}
```

```
public interface StudentSortingStrategy {

    boolean sortStudentObject(BTree.BtreeNode currentNode, Student key, int i);

    void sortTheElementsOfTheKeys(BTree.BtreeNode currentNode);

}
```

Node Class

```
public Boolean getIsNodeFull() {
        return isNodeFull;
}
```

```
 public Boolean isFull() {
         return isNodeFull;
 }
```

Tests

```
public void testFindStudentsWithGPALessThan() {
  Iterator<Student> externalItr = btree.iterator();
  while (externalItr.hasNext()) {
    Student student = externalItr.next();
    if (student.isProbationary()) {
      System.out.println(student.getRedID());
    }
  }
}
```

```
//need to pass root to methods in testing class, hence need to make it public
public BTreeNode getRootNode() {
    return rootNode;
}
```

Weak Test - Is it the Correct Student?

```
void setUp() {

    studentTreeByName.add(new Student("Manny", 825900492, 4.0));
    studentTreeByName.add(new Student("Andy", 825900499, 3.0));
    studentTreeByName.add(new Student("Katrina", 825900497, 2.0));
    studentTreeByName.add(new Student("Ciara", 825900496, 2.8));
    studentTreeByName.add(new Student("Mayer", 825900495, 3.5));
    studentTreeByName.add(new Student("Legend", 825900494, 3.3));
    studentTreeByName.add(new Student("Bill", 825900493, 2.4));
}

@Test
void getIndexStudent() {
    Student student = studentTreeByName.get(3);
    assertTrue(null != student);
}
```

```
class BTreeTest(unittest.TestCase):
    def setUp(self) -> None:
        self.tree = BTree.BTree(3)
        self.tree[1] = 2
        self.tree[12] = 3
        self.tree[4] = 16
        self.tree[20] = 88
        self.tree[6] = 9
        self.tree[16] = 36
        self.tree[19] = 43
        self.tree[50] = 18
        self.tree[25] = 0
        self.tree[17] = 14

    def test_setitem(self):
        self.tree[13] = 28
        self.assertTrue(self.tree[13])              self.assertTrue(28 == self.tree[13])
```

```
void testInternalIterator() {
        List<Student> students = new ArrayList<Student>();
        bTreeSet.forEach(students :: add);
        List<Student> testStudents = Arrays.asList(new Student("F", 2.5f, 825874455),
                        new Student("E", 4.0f, 825874454), new Student("D", 4.0f, 825874453),
                        new Student("C", 2.0f, 825874452), new Student("B", 1.0f, 825874451),
                        new Student("A", 4.0f, 825874450));
        assertIterableEquals(testStudents, students);
}
```

```
@Test
void testToString() {
	assertNotNull(testTreeByName.toString());
}

@Test
void testSize() {
	assertEquals(testTreeByGpa.size(),16);
}
```

```
@Test
void testForEach() {
	testTreeByGpa.forEach(item->assertNotNull(item.toString()));
}

@Test
void testToArray() {
	assertNotNull(testTreeByName.toArray());
}
```

```
@Test
public void testInternalIterator() throws Exception {
    try {
        bTreeObj.forEach((key, val) -> {
            final Student student = key;
            assertNotNull(student);
        });
    } catch (Exception e) {
        // Log exception
    }
}
```

Did it return the right number of students?

Was each student different?

Did the come out in correct order?

```
public boolean isEmpty() {
    return size() > 1 ? false : true;
}

public boolean isEmpty() {
    return size() == 1;
}
```

```
public String toString() {
    return "This is a Null Node Object of BTree";
}
```

Obscure Check

```
def _add_child(self, new_child):
    index = len(self.children) - 1

    while (index >= 0):
        # If not at a null node, find correct place to insert child
        if len(self.children[index]):
```

```
def _add_child(self, new_child):
index = len(self.children) - 1

while (index >= 0):
    # If not at a null node, find correct place to insert child
    if !self.children[index].isNull()):
```

Btree

-5 info hiding

```
    public Node search(Comparable key) {
        if(root == null)
            return null;
        Node current = root;
        while(true) {
            int pos = current.searchInNode(key);
            if(key.equals(current.keys[pos]))
                return current;
            else if(current.isLeaf())
                return null;
            else
                current = current.children[pos];
        }
    }
```

```
public class BTree<K, V> implements Map<K, V> {
    private final Comparator comparator;
    private Node root;
    private final int m;  // num of max children per node


      public Student put(Comparable key, Student student) {
          // If the tree is empty, create a root containing one elem
```

```
public Object get(Comparable key) {
    Iterator iter = new InOrderIterator();
    KeyValuePair keyValuePair;
    while(iter.hasNext()) {
        keyValuePair = (KeyValuePair) iter.next();
        if(keyValuePair.key.equals(key))
            return keyValuePair.value;
    }
    return null;
}

@Override
public V get(Object key) {
    throw new UnsupportedOperationException();
}
```

```
public class BTree <Key extends Comparable<Key>, Value> implements NavigableMap<Key, Value>,
Iterable<Value> {
	//class members
	private Node<Key,Value> root;
	private int order;
	private Strategy<Key,Value> strat;

	//constructors
	public BTree(int order)
	{
		this.setOrder(order);
		this.root = new NullNode<Key,Value>();
		this.strat = new OrderByKey<Key, Value>(true, false);
	}

	public BTree(int order, Strategy<Key, Value> strat)
	{
		this.setOrder(order);
		this.root = new NullNode<Key,Value>();
		this.strat = strat;
	}
```

```
public class DataNode <Key extends Comparable<Key>, Value>{
        private Key key;
        private Value value;
        private Node<Key,Value> leftNode;
        private Node<Key,Value> rightNode;
```

```
public class Node <Key extends Comparable<Key>, Value> {
        private DataNode<Key,Value>[] keys;
        private Node<Key,Value> parentNode;
        private Boolean isNodeFull;
        private int indexOfLastKey;
```

Expensive

```
@Override
public void forEach(Consumer action) {
    Objects.requireNonNull(action);
    StudentSearcher searcher= new StudentSearcher();
    int index = getBTree().size();
    for (int i = index; i > 0; i--) {
        Object student = searcher.getSpecificStudent(i,getBTree());
        action.accept(student);
        }

    }
```

Not needed

```
    public ArrayList<Student> getHighGPAStudents(BTree.BTreeNode node, ArrayList<Student>
students) {
        if (node!= null && !node.isEmpty()) {
            for (BTree.BTreeNode childNode : node.getChildrenNode()) {
                getHighGPAStudents(childNode, students);
            }
            for (Student student : node.getStudents()) {
                if (student!= null && student.getGpa() == 4.0f) {
                    students.add(student);
                }
            }


        }
        // sort students in lexicographical order
        Collections.sort(students, Comparator.comparing(Student::getName));
        // reverse the order
        Collections.reverse(students);
        return students;
    }
```

Use

```
List goodStudents = new ArrayList();
tree.forEach( (student) ->
    {if (student.gpa() > 3.5)
        goodStudents.add(student);})

Collections.sort(goodStudents, Comparator.comparing(Student::getName));
Collections.reverse(goodStudents);
```

Utility Method in BTree

```
static <K, V> Entry<K, V> successor(Entry<K, V> traverseNode) {
        if (traverseNode == null)
                return null;
        else if (!Entry.isNullEntry(traverseNode.rightChild)) {
                Entry<K, V> p = traverseNode.rightChild;
                while (!Entry.isNullEntry(p.leftChild))
                        p = p.leftChild;
                return p;
        } else {
                if (Entry.isNullEntry(traverseNode.rightChild) && traverseNode.next != null) {
                        return traverseNode.next;
                } else {
                        Entry<K, V> p = traverseNode.parent;
                        Entry<K, V> ch = traverseNode;
                        while (p != null && ch == p.rightChild) {
                                ch = p;
                                p = p.parent;
                        }
                        return p;
                }
51      }
```