자료구조 과제3 – IntBalancedSet

**Index**

1. **Specification**
2. **IntBalancedSet.java code**
3. **IntBalancedSetTester.java code**
4. **Example**
5. **Specification**

**public IntBalancedSet()**

비어있는 set으로 초기화한다.

**Postcondition** **:** 이 set는 비어있다

**Throws :** OutOfMemoryError – set를 만드는 동안 불충분한 메모리를 가리킨다.

**public void add(int element)**

이 set에 새 element를 더한다.

**Parameter :** element – 더해지는 새로운 element

**Postcondition :** 만약 element가 이미 이 set에 존재하면, 어떤 변화도 주지 않는다. 반면 element가 이 set에 더해진다.

**Throws :** OutOfMemoryError – set를 만드는 동안 불충분한 메모리를 가리킨다.

**public IntBalancedSet clone()**

이 set의 복사본을 생성한다.

**Returns :** 이 set의 복사본이 반환값이다. Subsequent가 복사본으로 변한 것은 원본에 영향을 주지 않을 것이며 그 역도 영향이 없다. 반환값은 사용되기 전에 IntBalancedSet로 캐스팅되어야 한다.

**Throws :** OutOfMemoryError – set를 만드는 동안 불충분한 메모리를 가리킨다.

**public boolean contains(int target)**

이 set에서 특수한 element인지 정하는 Accessor 메소드

**Parameter :** target – 이 set에서 될지 안될지에 대한 element

**Postcontition :** true(target이 set에 있다면) 또는 false(target이 set에 없다면)

**public boolean remove(int target)**

이 set로부터 구체화된 element를 삭제한다.

**Parameter :** target – 이 set로부터 삭제할 element

**Postcondtion :** 만약 target이 이 set에서 발견되었다면, 그 target을 삭제하고 메소드는 true를 반환한다. 그 반대로, 이 set는 변화하지 않은 상태로 남아있고 메소드는 false를 반환한다.

**private int firstGE(int target)**

**Postcondition :** 반환값 x는 data[x] >= target 처럼 이 루트의 첫째 위치에 존재한다. 만약 어떠한 위치도 없다면, 반환값은 dataCount이다.

**private void looseAdd(int element)**

**Precondition :** B-tree 전체는 유효하다.

**Postcondition :** 만약 element가 이미 이 set에 존재하면 이 set는 변하지 않는다. 반면, element가 이 set에 더해지고, B-Tree전체는 유효하다. 단, 이 set의 루트 내에서의 element 수가 허용된 maximum보다 하나 더 많을 때는 제외한다.

**private void fixExcess(int i)**

**Precondition :** subset[i]가 MAXIMUM+1개의 elements를 가질때를 제외하고, (i<childCount) 와전체적인 B-tree가 유효하다.

**Postcondition :** 이 set의 루트 내에있는 element의 수가 허용된 maximum보다 하나 더 많을 때를제외하고 tree는 정리되어있지 않아서 전체적인 B-tree는 유효하다.

**private boolean looseRemove(int target)**

**Precondition :** B-tree 전체는 유효하다.

**Postcondition :** 만약 target이 이 set내에 있다면, 삭제되고 메소드는 true를 반환한다. 반면, 이 set가 변하지 않았다면 false를 반환한다. 만약 element가 이미 이 set에 존재하면 이 set는 변하지 않는다. 이 set의 루트 내에서의 element 수가 허용된 maximum보다 하나 더 많을 때는 제외하고 B-Tree전체는 유효하다.

**private void fixShortage(int i)**

**Precondition :** subset[i]가 MINIMUM – 1 개의 element를 가진 것을 제외하고 (i<childCount) 이며 전체적인 B-tree는 유효하다.

**Postcondition :** 이 set의 루트 내에있는 element의 수가 허용된 minimum보다 하나 더 적을 때를제외하고 tree는 정리되어있지 않아서 전체적인 B-tree는 유효하다.

**private int removeBiggest()**

**Precondition :** (dataCount > 0)이고 전체적인 B-tree는 유효하다.

**Postcondition :** 가장 큰 element는 삭제되고 반환값은 삭제되었던 그 element이다. 이 set의 루트내에 있는 element 수가 허용된 minimum 보다 하나 더 적을 때를 제외하고 전체적인 B-tree는 유효하다.

**public void print(int indent)**

// 이 메소드는 B-tree의 set를 보여주는 메소드이다.

1. **IntBalancedSet.java code**

**package** BTree;

**public** **class** IntBalancedSet **implements** Cloneable{

**private** **static** **final** **int** ***MINIMUM*** =2;

**private** **static** **final** **int** ***MAXIMUM*** = 2\****MINIMUM***;

**int** dataCount;

**int**[] data;

**int** childCount;

IntBalancedSet [] subset;

/\*\*

\* Initialize an empty set.

\* **@param** - none

\* <dt><b>Postcondition:</b><dd>

\* This set is empty.

\* **@exception** OutOfMemoryError

\* Indicates insufficient memory for adding a new element.

\*\*/

**public** IntBalancedSet(){

data =**new** **int**[***MAXIMUM***+1];

subset=**new** IntBalancedSet[***MAXIMUM*** + 2];

}

/\*\*

\* Add a new element to this set.

\* **@param** <CODE>element</CODE>

\* the new element that is being added

\* <dt><b>Postcondition:</b><dd>

\* If the element was already in this set, then there is no change.

\* Otherwise, the element has been added to this set.

\* **@exception** OutOfMemoryError

\* Indicates insufficient memory for adding a new element.

\*\*/

**public** **void** add(**int** element){

looseAdd(element);

**if**(dataCount>***MAXIMUM***){

IntBalancedSet cpy=copyData();

data =**new** **int**[***MAXIMUM***+1];

dataCount=0;

childCount=dataCount+1;

subset=**new** IntBalancedSet[***MAXIMUM*** + 2];

subset[0]=cpy;

fixExcess(0);

}

}

// Precondition:

// The entire B-tree is valid.

// Postcondition:

// If entry was already in the set, then the set is unchanged. Otherwise,

// entry has been added to the set, and the entire B-tree is still valid

// EXCEPT that the number of entries in the root of this set might be one

// more than the allowed maximum.

**private** **void** looseAdd(**int** element){

**int** i=firstGE(element);

**if**(i==dataCount)

{

**if**(childCount==0)

{

insertEle(element, i);

}

**else**

{

subset[i].looseAdd(element);

fixExcess(i);

}

}

**else**

{

**if**(data[i]==element)

{

**return**;

}

**else**

{

**if**(childCount==0)

{

insertEle(element, i);

}

**else**

{

subset[i].looseAdd(element);

fixExcess(i);

}

}

}

}

**private** **void** insertEle(**int** element,**int** locale){

**for**(**int** i=dataCount;i>locale;i--){

data[i]=data[i-1];

}

data[locale]=element;

dataCount++;

//System.out.println(dataCount+" "+childCount);

}

**private** **void** divoteSub(IntBalancedSet set1,IntBalancedSet set2,**int** i){

System.*arraycopy*(subset[i].data,0, set1.data, 0, ***MINIMUM***);

System.*arraycopy*(subset[i].subset,0, set1.subset, 0, ***MINIMUM***+1);

System.*arraycopy*(subset[i].data, ***MINIMUM***+1, set2.data, 0, ***MINIMUM***);

System.*arraycopy*(subset[i].subset,***MINIMUM***+1, set2.subset, 0, ***MINIMUM***+1);

set1.dataCount=***MINIMUM***;

set1.childCount=subset[i].childCount/2;

set2.dataCount=***MINIMUM***;

set2.childCount=subset[i].childCount/2;

}

// Precondition:

// (i < childCount) and the entire B-tree is valid EXCEPT that

// subset[i] has MAXIMUM + 1 entries. Also, the root is allowed to have

// zero entries and one child.

// Postcondition:

// The tree has been rearranged so that the entire B-tree is valid EXCEPT

// that the number of entries in the root of this set might be one more than

// the allowed maximum.

**private** **void** fixExcess(**int** i){

**if**(subset[i].dataCount>***MAXIMUM***){

IntBalancedSet set1=**new** IntBalancedSet();

IntBalancedSet set2=**new** IntBalancedSet();

insertEle(subset[i].data[***MINIMUM***], i);//在父节点插入该中间元素.

divoteSub(set1, set2, i);

subset[i]=set1;

**for**(**int** j=childCount;j>i+1;j--){

subset[j]=subset[j-1];

}

subset[i+1]=set2;

childCount++;

//System.out.println(data[0]);

}

}

**private** IntBalancedSet copyData(){

IntBalancedSet copy;

copy=**new** IntBalancedSet();

copy.data=data;

copy.dataCount=dataCount;

copy.childCount=childCount;

copy.subset=subset;

**return** copy;

}

/\*\*

\* Generate a copy of this set.

\* **@param** - none

\* **@return**

\* The return value is a copy of this set. Subsequent changes to the

\* copy will not affect the original, nor vice versa. Note that the return

\* value must be type cast to an <CODE>IntBalancedSet</CODE> before it

\* can be used.

\* **@exception** OutOfMemoryError

\* Indicates insufficient memory for creating the clone.

\*\*/

**public** Object clone(){

IntBalancedSet copy;

copy=**new** IntBalancedSet();

copy.data=data.clone();

copy.dataCount=dataCount;

copy.childCount=childCount;

copy.subset=subset.clone();

**return** copy;

}

/\*\*

\* Accessor method to determine whether a particular element is in this set.

\* **@param** <CODE>target</CODE>

\* an element that may or may not be in this set

\* **@return**

\* <CODE>true</CODE> if this set contains <CODE>target</CODE>;

\* otherwise <CODE>false</CODE>

\*\*/

**public** **boolean** contains(**int** target){

**int** i=firstGE(target);

//System.out.println("i:"+i);

**if**(i==dataCount){

**if**(childCount==0)

**return** **false**;

**else**

**return** subset[i].contains(target);

}

**else**{

**if**(target==data[i])

{

**return** **true**;

}

**else**{

**if**(childCount==0)

**return** **false**;

**else**

**return** subset[i].contains(target);

}

}

}

// Postcondition: The return value, x, is the first location in the root

// such that data[x] >= target. If there is no such location, then the

// return value is dataCount.

**private** **int** firstGE(**int** target){

**int** i=0;

**for**(;i<dataCount;i++){

**if**(data[i]>=target)

**return** i;

}

**return** i;

}

/\*\*

\* Remove a specified element from this set.

\* **@param** <CODE>target</CODE>

\* the element to remove from this set

\* <dt><b>Postcondition:</b><dd>

\* if <CODE>target</CODE> was found in this set, then it has been removed

\* and the method returns <CODE>true</CODE>. Otherwise this set remains

\* unchanged and the method returns <CODE>false</CODE>.

\*\*/

**public** **boolean** remove(**int** target){

**boolean** answer = looseRemove(target);

**if**((dataCount==0)&&(childCount==1))

{

dataCount=subset[0].dataCount;

childCount=subset[0].childCount;

data=subset[0].data;

subset=subset[0].subset;

}

**return** answer;

}

// Precondition:

// The entire B-tree is valid.

// Postcondition:

// If target was in the set, then it has been removed from the set and the

// method returns true; otherwise the set is unchanged and the method

// returns false. The entire B-tree is still valid EXCEPT that the

// number of entries in the root of this set might be one less than the

// allowed minimum.

**private** **boolean** looseRemove(**int** target){

**int** i=firstGE(target);

**if**(childCount==0){

**if**(i!=dataCount&&data[i]==target){

cover(i);

**return** **true**;

}**else**{

**return** **false**;

}

}**else**{

**if**(i!=dataCount&&data[i]==target){

data[i]= subset[i].removeBiggest();

**if**(subset[i].dataCount<***MINIMUM***)

fixShortage(i);

**return** **true**;

}

**else**{

**boolean** answer=subset[i].looseRemove(target);

**if**(subset[i].dataCount<***MINIMUM***)

fixShortage(i);

**return** answer;

}

}

}

// Precondition:

// (i < childCount) and the entire B-tree is valid EXCEPT that

// subset[i] has only MINIMUM - 1 entries.

// Postcondition:

// The tree has been rearranged so that the entire B-tree is valid EXCEPT

// that the number of entries in the root of this set might be one less than

// the allowed minimum.

**private** **void** fixShortage(**int** i){

**if**(i!=0&&subset[i-1].dataCount>***MINIMUM***){

subset[i].insertEle(data[i-1], 0);

data[i-1]=subset[i-1].cover(subset[i-1].dataCount-1);

**if**(subset[i-1].childCount!=0){

subset[i].addSubset(subset[i-1].coverSub(subset[i-1].childCount-1),0);

}

**return**;

}

**if**(i!=0&&subset[i-1].dataCount==***MINIMUM***){

subset[i-1].insertEle(data[i-1], subset[i-1].dataCount);

cover(i-1);

combineSub(subset[i-1],subset[i]);

coverSub(i);

**return** ;

}

**if**(i<dataCount&&subset[i+1].dataCount>***MINIMUM***){

subset[i].insertEle(data[i], subset[i].dataCount);

data[i]=subset[i+1].cover(0);

**if**(subset[i+1].childCount!=0){

subset[i].addSubset(subset[i+1].coverSub(0), subset[i].childCount);

}

**return**;

}

**if**(i<dataCount&&subset[i+1].dataCount==***MINIMUM***){

subset[i+1].insertEle(data[i], 0);

cover(i);

combineSub(subset[i],subset[i+1]);

coverSub(i+1);

**return**;

}

}

// Precondition:

// (dataCount > 0) and the entire B-tree is valid.

// Postcondition:

// The largest item in the set has been removed, and the value of this

// item is the return value. The B-tree is still valid EXCEPT

// that the number of entries in the root of this set might be one less than

// the allowed minimum.

**private** **int** removeBiggest(){

**if**(childCount==0){

**return** data[--dataCount];

}**else**{

**int** answer = subset[childCount-1].removeBiggest();

**if**(subset[childCount-1].dataCount<***MINIMUM***){

fixShortage(childCount-1);

}

**return** answer;

}

}

**private** **void** combineSub(IntBalancedSet set1,IntBalancedSet set2){

**for**(**int** i=0;i<set2.dataCount;i++){

//System.out.println(" combineSub "+set1.dataCount+" "+set2.dataCount);

set1.data[set1.dataCount++]=set2.data[i];

}

**for**(**int** i=0;i<set2.childCount;i++){

set1.subset[set1.childCount++]=set2.subset[i];

}

}

**private** IntBalancedSet coverSub(**int** i){

IntBalancedSet answer=subset[i];

**for**(**int** j=i;j<childCount;j++){

subset[j]=subset[j+1];

}

childCount--;

**return** answer;

}

**private** **int** cover(**int** i){

**int** answer=data[i];

**for**(**int** j=i;j<dataCount;j++){

data[j]=data[j+1];

}

dataCount--;

**return** answer;

}

// private void tranferEle(int i,int flag){

// int dl=i+(flag==-1?0:flag);//datalocation

// subset[i].insertEle(data[dl], 0);

// data[dl]=subset[i+flag].data];

// if(subset[i-1].childCount!=0){

// subset[i-1].addSubset(subset[subset[i-1].childCount--],0);

// }

// }

**private** **void** addSubset(IntBalancedSet sub,**int** l){

**for**(**int** i=childCount;i>l;i--){

subset[i]=subset[i-1];

}subset[l]=sub;

childCount++;

}

// Print a representation of this set's B-tree, useful during debugging.

**public** **void** print(**int** indent){

**final** **int** EXTRA\_INDENTATION=4;

**int** i;

**int** space;

**for**(space=0;space<indent;space++){

System.***out***.print(" ");

}

**for**(i=0;i<dataCount;i++){

System.***out***.print(data[i]+",");

}

System.***out***.print("DC:"+dataCount+" CC:"+childCount);

System.***out***.println();

**for**(i=childCount-1;i>=0;i--){

subset[i].print(indent+EXTRA\_INDENTATION);

}

}

}

1. **IntBalancedSetTester.java code**

**package** BTree;

**import** java.util.\*;

**public** **class** IntBalancedSetTester {

**public** **static** **void** main (String [] args) {

IntBalancedSet set = **new** IntBalancedSet();

Random rg = **new** Random();

**for** (**int** x=0; x<10; x++)

{

**int** num = rg.nextInt(10);

System.***out***.println("Adding " + num);

set.add(num);

System.***out***.println("tree so far:");

set.print(4);

}

System.***out***.println("final version of tree");

set.print(4);

**int** num = rg.nextInt(5);

**if** (set.contains(num))

System.***out***.println ("found it!");

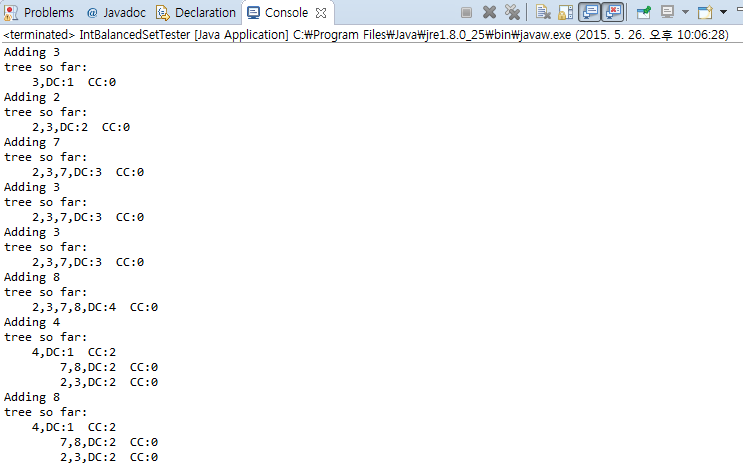
**else**

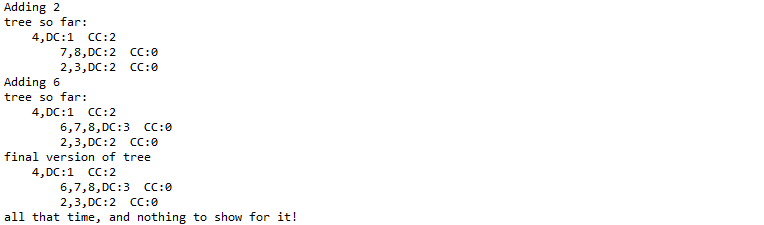
System.***out***.println ("all that time, and nothing to show for it!");

}

}

1. **Example**

****

****