자료구조: 2022년 1학기 [강의]

Dictionary (2)

강지훈
jhkang@cnu.ac.kr
충남대학교 컴퓨터융합학부

Class

“Dictionary<Key, Obj>”

Dictionary<Key,Obj> 의 공개함수

- Dictionary 객체 사용법
  - public Dictionary () ;
  - public boolean isEmpty ( ) ;
  - public boolean isFull () ;
  - public int size() ;
  - public boolean keyDoesExist (Key aKey) ;
  - public Obj objectForKey (Key aKey) ;
  - public boolean addKeyAndObject (Key aKey, Obj anObject) ;
  - public Obj removeObjectForKey (Key aKey) ;
  - public boolean replaceObjectForKey (Key aKey, Obj objectForReplace) ;
  - public void clear ( ) ;

Class

“Dictionary<Key, Obj>”의 구현

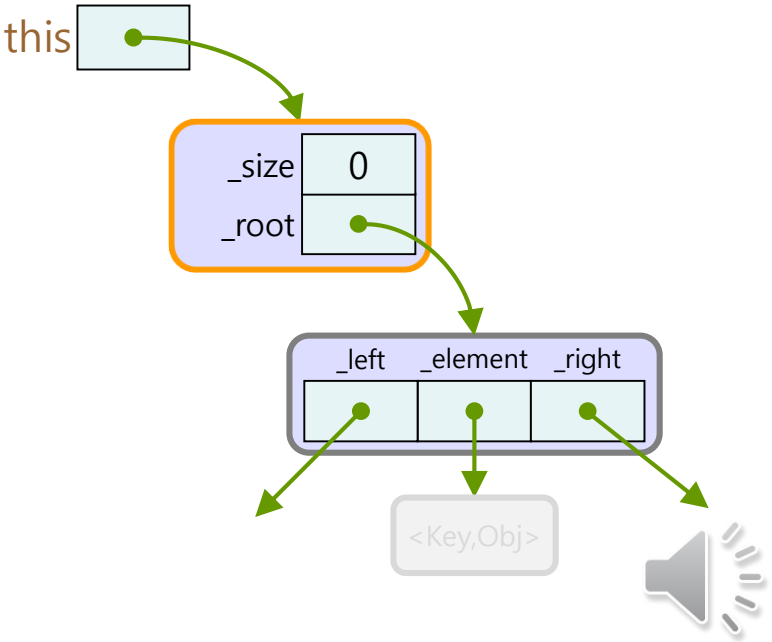
Dictionary: Key extends Comparable<Key>

- Class "Dictionary" 안에서 Key 객체 끼리의 비교가 필요.
- Generic type "Key" 를 대체할 실제 class 에 조건을 부여하고 있다:
  - Interface "Comparable" 을 구현하고 있어야 한다.
  - 즉, "compareTo()" 함수를 override 하여 구현하고 있어야 한다.

```
public class  Dictionary <Key extends Comparable<Key>, Obj >
{
    // 비공개 인스턴스 변수
    private int                                          _size ;
    private BinaryNode<DictionaryElement<Key,Obj>> _root ;
```

Dictionary: 객체 인스턴스

```
public class  Dictionary < Key extends Comparable<Key> , Obj >
{
    // 비공개 인스턴스 변수
    private int                                                  _size ;
    private BinaryNode<DictionaryElement<Key,Obj>> _root ;
```



Dictionary: Getter/Setter

```
public class  Dictionary < Key extends Comparable<Key> , Obj >
{
    // 비공개 인스턴스 변수
    private int                                          _size ;
    private BinaryNode<DictionaryElement<Key,Obj>>       _root ;

    // Getter/Setter
    public int  size () {
        return  this._size ;
    }
    private void  setSize (int newSize) {
        this._size = newSize ;
    }

    private BinaryNode<DictionaryElement<Key,Obj>> root () {
        return  this._root ;
    }
    private void  setRoot (BinaryNode<DictionaryElement<Key,Obj>> newRoot) {
        this._root = newRoot ;
    }
```

Dictionary의 생성자

```
public class  Dictionary<Key extends Comparable<Key>,Obj>
{
    ……

    // 생성자
    public  Dictionary( )
    {
        this.setSize (0) ;
        this.setRoot (null) ;
    }
```

Dictionary : 상태 알아보기

```
public class  Dictionary<Key extends Comparable<Key>, Obj>
{
    ......

    // 상태 알아보기
    public boolean  isEmpty()
    {
        return  (this.root() == null) ; // 또는  return  (this.size() == 0) ;
    }

    public boolean  isFull ()
    {
        return  false ;
    }

    // public int  size()
    // {
    //     return  this._size ;
    // }
```

Dictionary: objectForKey() [Nonrecursive]

```
public Obj  objectForKey (Key aKey)
{
    boolean  found = false ;
    BinaryNode<DictionaryElement<Key,Obj>>  currentRoot = this.root() ;
    while ( (! found) && (currentRoot != null) ) {
        if ( aKey.compareTo(currentRoot.element().key()) == 0 ) {
            found = true ;
        }
        else if ( aKey.compareTo(currentRoot.element().key()) < 0 ) {
            currentRoot = currentRoot.left() ;
        }
        else {
            currentRoot = currentRoot.right() ;
        }
    }
    if ( found ) {
        return  currentRoot.element().object() ;
    }
    else {
        return  null ;
    }
}
```

Dictionary: objectForKey() [Recursive]

```
public Obj  objectForKey (Key aKey)
{
    return  this.objectForKeyRecursively (this.root(), aKey) ;
}

private Obj  objectForKeyRecursively
    (BinaryNode<DictionaryElement<Key,Obj>> currentRoot, Key aKey)
{
    if ( currentRoot == null ) {
        return  null ;
    else {
        if ( aKey.compareTo(currentRoot.element().key()) == 0 ) {
            return  currentRoot.element().object() ;
        }
        else if ( aKey.compareTo(currentRoot.element().key()) < 0 ) {
            return  this.objectForKeyRecursively (currentRoot.left(), aKey) ;
        }
        else {
            return  this.objectForKeyRecursively (currentRoot.right(), aKey) ;
        }
    }
}
```

Dictionary: keyDoesExist() [Nonrecursive]

```
public boolean  keyDoesExist (Key aKey)
{
    boolean  found = false ;
    BinaryNode<DictionaryElement<Key,Obj>>  currentRoot = this.root() ;
    while ( (! found) && (currentRoot != null) ) {
        if ( aKey.compareTo(currentRoot.element().key()) == 0 ) {
            found = true ;
        }
        else if ( aKey.compareTo(currentRoot.element().key()) < 0 ) {
            currentRoot = currentRoot.left() ;
        }
        else {
            currentRoot = currentRoot.right() ;
        }
    }
    return  found ;
}
```

Dictionary: keyDoesExist() [Recursive]

```
public boolean  keyDoesExist (Key aKey)
{
    return  this.keyDoesExistRecursively (this.root(), aKey) ;
}

private boolean  keyDoesExistRecursively
    (BinaryNode<DictionaryElement<Key,Obj>> currentRoot, Key aKey)
{
    if ( currentRoot == null ) {
        return  false ;
    else {
        if ( aKey.compareTo(currentRoot.element().key()) == 0 ) {
            return  true ;
        }
        else if ( aKey.compareTo(currentRoot.element().key()) < 0 ) {
            return  this.keyDoesExistRecursively (currentRoot.left(), aKey) ;
        }
        else {
            return  this.keyDoesExistRecursively (currentRoot.right(), aKey) ;
        }
    }
}
```

Dictionary: keyDoesExist() [objectForKey() 를 사용]

```
public boolean  keyDoesExist (Key aKey)
{
    return  ( this.objectForKey (aKey) != null ) ;
}
```

Dictionary: addKeyAndObject() [recursive]

```
public boolean  addKeyAndObject (Key aKey, Obj anObject)
{
    if ( this.root() == null ) {
        this.setRoot (
            new BinaryNode ((new DictionaryElement(aKey, anObject)), null, null) ) ;
        this.setSize (1) ;
        return  true ;
    }
    else {
        return  this.addKeyAndObjectToSubtree (this.root(), aKey, anObject) ;
    }
}
```

Dictionary: addKeyAndObjectToSubtree()

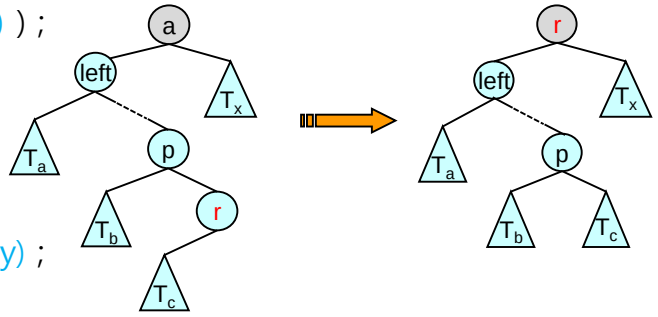
```
private boolean addKeyAndObjectToSubtree
      (BinaryNode<DictionaryElement<Key,Obj>> currentRoot, Key aKey, Obj anObject)
{
      if ( currentRoot.element().key().compareTo(aKey) == 0 ) {
            return  false ; // "aKey" already exists, so we cannot add.
      }
      else if ( aKey.compareTo(currentRoot.element().key()) < 0 ) {
            if ( currentRoot.left() == null ) { // We can add to the left
                  DictionaryElement<Key,Obj> elementForAdd =
                        new DictionaryElement<Key,Obj>(aKey, anObject) ;
                  currentRoot.setLeft (
                        new BinaryNode<DictionaryElement<Key,Obj>>(elementForAdd, null, null) ;
                  this.setSize (this.size()+1) ;
                  return  true ;
            }
            else {
                  return  this.addKeyAndObjectToSubtree(currentRoot.left(), aKey, anObject) ;
            }
      }
      else {
            if ( currentRoot.right() == null ) { // We can add to the right
                  DictionaryElement<Key,Obj> elementForAdd =
                        new DictionaryElement<Key,Obj>(aKey, anObject) ;
                  currentRoot.setRight (
                        new BinaryNode<DictionaryElement<Key,Obj>>(elementForAdd, null, null) ;
                  this.setSize (this.size()+1) ;
                  return  true ;
            }
            else {
                  return  this.addKeyAndObjectToSubtree(currentRoot.right(), aKey, anObject) ;
            }
      }
}
```

Dictionary: addKeyAndObject() [Nonrecursive]

```
public boolean  addKeyAndObject (Key aKey, Obj anObject)
{
        if ( this.root() == null ) {
                DictionaryElement<Key,Obj> elementForAdd =
                        new DictionaryElement<Key,Obj>(aKey, anObject) ;
                this.setRoot (new BinaryNode<DictionaryElement<Key,Obj>> (elementForAdd, null, null) ) ;
                this.setSize (1) ;
                return  true ;
        }
        BinaryNode<DictionaryElement<Key,Obj>>  currentRoot = this.root() ;
        while ( aKey.compareTo(currentRoot.element().key()) != 0 ) {
                if ( aKey.compareTo(currentRoot.element().key()) < 0 ) {
                        // We should go down to the left subtree.
                        if ( currentRoot.left() == null ) { // No left subtree. We can add to the left of "current".
                                DictionaryElement<Key,Obj> elementForAdd =
                                        new DictionaryElement<Key,Obj>(aKey, anObject) ;
                                BinaryNode<DictionaryElement<Key,Obj>> nodeForAdd =
                                        new BinaryNode<DictionaryElement<Key,Obj>> (elementForAdd, null, null) ;
                                currentRoot.setLeft(nodeForAdd) ;
                                this.setSize (this.size()+1) ;
                                return  true ;
                        }
                        currentRoot = currentRoot.left() ;
                }
                else {
                        // We should go down to the right subtree.
                        if ( currentRoot.right() == null ) { // No right subtree. We can add to the right of "current".
                                DictionaryElement<Key,Obj> elementForAdd =
                                        new DictionaryElement<Key,Obj>(aKey, anObject) ;
                                BinaryNode<DictionaryElement<Key,Obj>> nodeForAdd =
                                        new BinaryNode<DictionaryElement<Key,Obj>> (elementForAdd, null, null) ;
                                currentRoot.setRight(nodeForAdd) ;
                                this.setSize (this.size()+1) ;
                                return  true ;
                        }
                        currentRoot = currentRoot.right() ;
                }
        } // end of while
        return  false ; // "aKey" already exists, so we cannot add.
}
```

Dictionary: removeObjectForKey() (ver.1)

```
public Obj  removeObjectForKey (Key aKey)
{
      Obj  removedObject = null ;
      if ( this.root() == null ) {
            return  null ; // Empty tree.
      }
      if ( aKey.compareTo(this.root().element().key()) == 0 ) { // root 를 삭제해야 한다
            removedObject = this.root().element().object() ;
            if ( (this.root().left() == null) && ( this.root().right() == null) ) { // root 만 있는 tree
                  this.setRoot (null) ;
            }
            else if ( this.root().left() == null )  { // root 의 left tree 가 없다
                  this.setRoot (this.root().right()) ;
            } else { // root 의 right tree 가 없다
                  this.setRoot (this.root().left()) ;
            }
            else { // child의 left tree, right tree 가 모두 있다
                  this.root().setElement (
                        this.removeRightMostElementOfLeftTree(this.root()) ) ;
            }
            this.setSize (this.size()-1) ;
            return  removedObject ;
      }
      else {
            return  this.removeObjectForKeyFromSubtree (this.root(), aKey) ;
      }
}
```



Dictionary: removeRightMostElementOfLeftTree() [1]

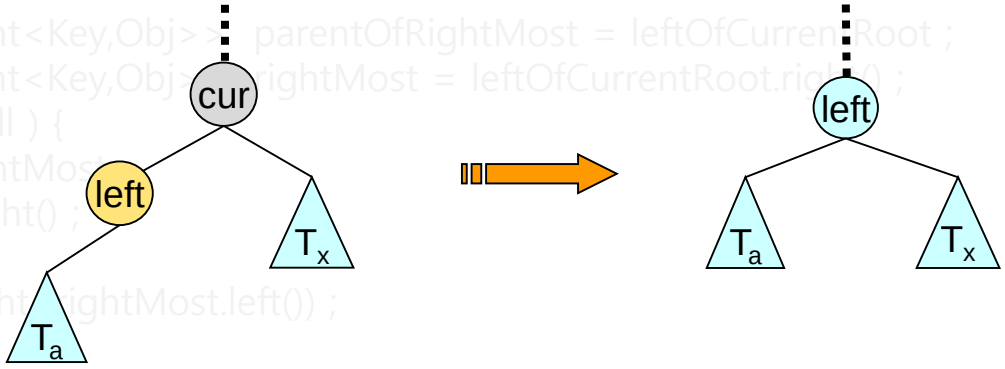
```
private DictionaryElement<Key,Obj>
      removeRightMostElementOfLeftTree (BinaryNode currentRoot)
{
      // 현재의 currentRoot 의 원소를 대체할 원소인,
      // 왼쪽 트리의 가장 오른쪽 노드의 원소를 삭제하여 얻는다.
      // call 되는 시점에, currentRoot.left() 는 null 이 아니다

      BinaryNode<DictionaryElement<Key,Obj>>  leftOfCurrentRoot = currentRoot.left() ;
      if ( leftOfCurrentRoot.right() == null ) {
            currentRoot.setLeft (leftOfCurrentRoot.left()) ;
            return  leftOfCurrentRoot.element() ;
      }
      else {
            BinaryNode<DictionaryElement<Key,Obj>>  parentOfRightMost = leftOfCurrentRoot ;
            BinaryNode<DictionaryElement<Key,Obj>>  rightMost = leftOfCurrentRoot.right() ;
            while ( rightMost.right() != null ) {
                  parentOfRightMost = rightMost ;
                  rightMost = rightMost.right() ;
            }
            this.parentOfRightMost.setRight (rightMost.left()) ;
            return  rightMost.element() ;
      }
}
```

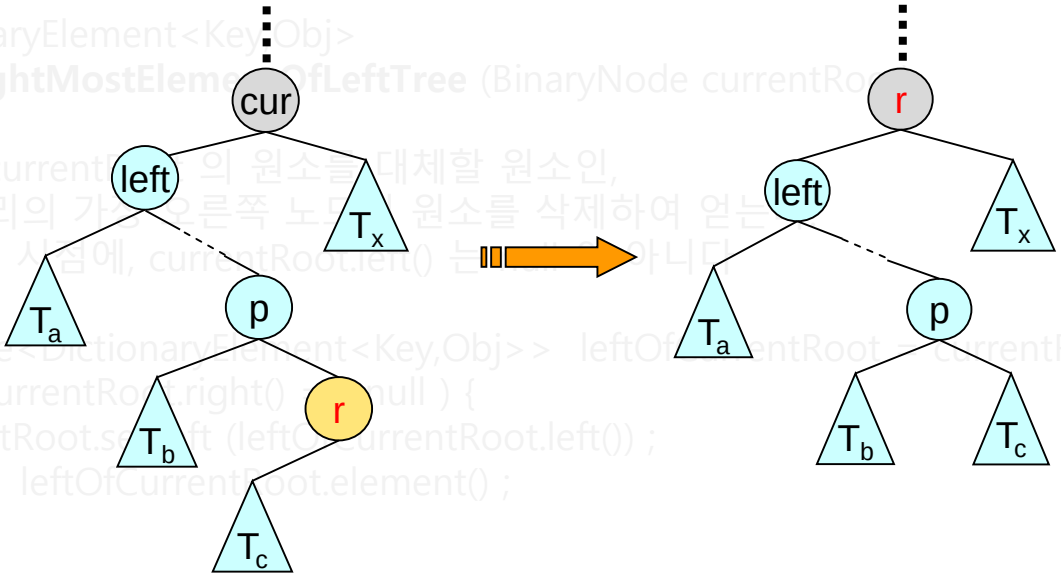
Dictionary: removeRightMostElementOfLeftTree() [2]

```
private DictionaryElement<Key,Obj>
      removeRightMostElementOfLeftTree (BinaryNode currentRoot)
{
      // 현재의 currentRoot 의 원소를 대체할 원소인,
      // 왼쪽 트리의 가장 오른쪽 노드의 원소를 삭제하여 얻는다.
      // call 되는 시점에, currentRoot.left() 는 null 이 아니다

      BinaryNode<DictionaryElement<Key,Obj>>  leftOfCurrentRoot = currentRoot.left() ;
      if ( leftOfCurrentRoot.right() == null ) {
            currentRoot.setLeft (leftOfCurrentRoot.left()) ;
            return  leftOfCurrentRoot.element() ;
      }
      else {
            BinaryNode<DictionaryElement<Key,Obj>> parentOfRightMost = leftOfCurrentRoot ;
            BinaryNode<DictionaryElement<Key,Obj>> rightMost = leftOfCurrentRoot.right() ;
            while ( rightMost.right() != null ) {
                  parentOfRightMost = rightMost ;
                  rightMost = rightMost.right() ;
            }
            this.parentOfRightMost.setRight (rightMost.left()) ;
            return  rightMost.element() ;
      }
}
```



Dictionary: removeRightMostElementOfLeftTree() [3]




```
    else {
        BinaryNode<DictionaryElement<Key,Obj>>  parentOfRightMost = leftOfCurrentRoot ;
        BinaryNode<DictionaryElement<Key,Obj>>  rightMost = leftOfCurrentRoot.right() ;
        while ( rightMost.right() != null ) {
            parentOfRightMost = rightMost ;
            rightMost = rightMost.right() ;
        }
        this.parentOfRightMost.setRight (rightMost.left()) ;
        return  rightMost.element() ;
    }
}
```

Dictionary: removeObjectForKeyFromSubtree() (Recursive) [1]

```
private Obj
        removeObjectForKeyFromSubtree (BinaryNode<DictionaryElement<Key,Obj>> currentRoot, Key aKey)
{
        // 이 시점에, currentRoot 는 null 이 아니고, currentRoot 의 key 는 "aKey" 와 일치하지 않는다.
        if ( aKey.compareTo(currentRoot.element().key()) < 0 ) { // left subtree 에서 삭제해야 한다
                BinaryNode<DictionaryElement<Key,Obj>> child = currentRoot.left () ;
                if ( child == null ) {
                        return  null ;
                }
                else {
                        if ( aKey.compareTo(child.element().key()) == 0 ) {
                                Obj  removedObject = child.element().object() ;
                                if ( child.left() == null && child.right() == null ) { // child가 leaf
                                        currentRoot.setLeft (null) ;
                                }
                                else if ( child.left() == null ) { // child 의 left tree가 없다
                                        currentRoot.setLeft (child.right()) ;
                                }
                                 else if ( child.right() == null ) { // child 의 right tree 가 없다
                                        currentRoot.setLeft (child.left()) ;
                                } else { // child의 left tree, right tree가 모두 있다
                                        currentRoot.setElement (this.removeRightMostElementOfLeftTree(child)) ;
                                }
                                this.setSize (this.size()-1) ;
                                return  removedObject ;
                        }
                        else {
                                return  this.removeObjectForKeyFromSubtree (child, aKey) ;
                        }
                }
        }
        else { // right subtree 에서 삭제해야 한다
```

Dictionary: removeObjectForKeyFromSubtree() (Recursive) [2]

```
private Obj
        removeObjectForKeyFromSubtree (BinaryNode<DictionaryElement<Key,Obj>> currentRoot, Key aKey)
{
        if ( aKey.compareTo(currentRoot.element().key()) < 0 ) {// left subtree 에서 삭제해야 한다
                .....
        }
        else { // right subtree 에서 삭제해야 한다
                BinaryNode<DictionaryElement<Key,Obj>> child = currentRoot.right() ;
                if ( child == null ) {
                        return  null ;
                }
                else {
                        if ( aKey.compareTo(child.element().key()) == 0 ) {
                                Obj  removedObject = child.element().object() ;
                                if ( child.left() == null && child.right() == null ) { // child가 leaf
                                        currentRoot.setRight (null) ;
                                }
                                else if ( child.left() == null ) { // child 의 left tree가 없다
                                        currentRoot.setRight (child.right()) ;
                                }
                                 else if ( child.right() == null ) { // child 의 right tree 가 없다
                                        currentRoot.setRight (child.left()) ;
                                } else { // child의 left tree, right tree가 모두 있다
                                        currentRoot.setElement (this.removeRightMostElementOfLeftTree(child)) ;
                                }
                                this.setSize (this.size()-1) ;
                                return  removedObject ;
                        }
                        else {
                                return  this.removeObjectForKeyFromSubtree (child, aKey) ;
                        }
        }
}
```

Dictionary: removeObjectForKeyFromSubtree() (Nonrecursive) [1]

```
private Obj  removeObjectForKeyFromSubtree
      (BinaryNode<DictionaryElement<Key,Obj>> currentRoot, Key aKey)
{
      // 이 시점에, currentRoot 는 null 이 아니고, currentRoot의 key 는 aKey 와 일치하지 않는다.
      do {
            if ( aKey.compareTo(currentRoot.element().key()) < 0 ) { // left tree에서 삭제 노드를 찾아야 한다
                  BinaryNode<DictionaryElement<Key,Obj>> child = currentRoot.left () ;
                  if ( child == null ) {
                        return  null ;
                  }
                  if ( aKey.compareTo(child.element().key()) == 0 ) { // found
                        Obj  removedObject = child.element().object() ;
                        if ( child.left() == null && child.right() == null ) { // child가 leaf
                              currentRoot.setLeft (null) ;
                        }
                        else if ( child.left() == null ) { // child 의 left tree가 없다
                              currentRoot.setLeft (child.right()) ;
                        }
                         else if ( child.right() == null ) { // child 의 right tree 가 없다
                              currentRoot.setLeft (child.left()) ;
                        } else { // child의 left tree, right tree 가 모두 있다
                              currentRoot.setElement (removeRightMostElementOfLeftTree(child)) ;
                        }
                        this.setSize (this.size()-1) ;
                        return  removedObject ;
                  }
            }
            else { // right subtree에서 삭제 노드를 찾아야 한다
```

Dictionary: removeObjectForKeyFromSubtree() (Nonrecursive) [2]

```
private Obj  removeObjectForKeyFromSubtree (BinaryNode<DictionaryElement<Key,Obj>> currentRoot, Key aKey)
{
      do {
            if ( aKey.compareTo(currentRoot.element().key()) < 0 ) {// left subtree 에서 삭제해야 한다
                  ......
            }
            else { // right subtree 에서 삭제해야 한다
                  BinaryNode<DictionaryElement<Key,Obj>> child = currentRoot.right() ;
                  if ( child == null ) {
                        return null ;
                  }
                  if ( aKey.compareTo(child.element().key()) == 0 ) { // found
                        Obj  removedObject = child.element().object() ;
                        if ( child.left() == null && child.right() == null ) { // child가 leaf
                              currentRoot.setRight (null) ;
                        }
                        else if ( child.left() == null ) { // child 의 left tree가 없다
                              currentRoot.setRight (child.right()) ;
                        }
                         else if ( child.right() == null ) { // child 의 right tree 가 없다
                              currentRoot.setRight (child.left()) ;
                        } else { // child의 left tree, right tree가 모두 있다
                              currentRoot.right().setElement (this.removeRightMostNodeOfLeftTree(child)) ;
                        }
                        this.setSize (this.size()-1) ;
                        return  removedObject ;
                  }
            }
            currentRoot = child ;
      } while (true) ;
} // End of "removeObjectForKeyFromSubtree"
```

Dictionary: replaceObjectForKey()

```
public boolean  replaceObjectForKey (Obj objectForReplace, Key aKey)
{
    boolean  found = false ;
    BinaryNode<DictionaryElement<Key,Obj>>  currentRoot = this.root() ;
    while ( (! found) && (currentRoot != null) ) {
        if ( aKey.compareTo(currentRoot.element().key()) < 0) {
            currentRoot = currentRoot.left () ;
        }
        else if ( aKey.compareTo(currentRoot.element().key()) > 0 ) {
            currentRoot = currentRoot.right () ;
        }
        else {
            found = true ;
        }
    }
    if ( found ) {
        currentRoot.element().setObject (objectForReplace) ;
        return  true ;
    }
    else {
        return  false ;
    }
}
```

Dictionary: clear ( )

```
public void  clear ( )
{
    this.setSize (0) ;
    this.setRoot (null) ;
}
```

중위 탐색에서 Visit 행위를 Call Back 으로 처리하기 위한

Interface "VisitDelegateForTraversal"

Interface VisitDelegateForTraversal

- Class "DictionaryByBinarySearchTree"의 트리 탐색 (traverse)을 실행하는 함수에서 (예를 들어 inorder()), 노드를 방문(visit) 할 때 마다 실행할 CallBack 함수 사용법 (함수의 이름과 매개변수)를 정의하기 위한 Interface.

```
public interface  VisitDelegateForTraversal<Key,Obj> {
    public void  visitForInorder (DictionaryElement<Key,Obj> anElement, int aLevel) ;
    public void  visitForReverseOfInorder
       (DictionaryElement<Key,Obj> anElement, int aLevel) ;
}
```

  - 이진검색트리로 구현된 Class "Dictionary"와 사용자 class 는, CallBack 을 위한 interface "VisitDelegateForTraversal" 을 공유한다.
  - CallBack 함수 ("visitForInorder()" 와 "visitForReverseOfInorder()")의 매개변수:
    - 사용자는 트리를 탐색하는 동안 노드를 방문할 때 마다, 방문 노드의 원소 (anElement) 와 그 노드의 트리 레벨 (aLevel) 정보를 CallBack 함수를 통해서 전달받게 된다.
    - 이 두 정보를 사용하여 사용자는 자신의 목적에 맞는 일을 구현할 수 있다.

중위 탐색에서
Call Back 이 가능하도록
Class "Dictionary" 를 보완

Class "Dictionary" 의 수정 [1]

```
public class  Dictionary<Key extends Comparable<Key>, Obj> {
    .....

    private VisitDelegateForTraversal<Key,Obj> _visitDelegate ;
        // 전달받은 Visit delegate 객체를 저장할 곳

    public VisitDelegateForTraversal<Key,Obj>  visitDelegate () {
        return  this._visitDelegate ;
    }
    public void  setVisitDelegate (VisitDelegateForTraversal<Key,Obj> newVisitDelegate) {
        this._visitDelegate = newVisitDelegate ;
    }
    ......

    private void  inorderRecursively
        ( BinaryNode<DictionaryElement<Key,Obj>> aRootOfSubtree, int  aLevel )
    {
        if ( aRootOfSubtree != null ) {
            this.inorderRecursively (aRootOfSubtree.left(), aLevel+1) ;
            this.visitDelegate().visitForInorder (aRootOfSubtree.element(), aLevel) ;
            this.inorderRecursively (aRootOfSubtree.right(), aLevel+1) ;
        }
    }

    public void  inorder () {
        this. inorderRecursively (this.root(), 1) ;
    }
```

```
public interface  VisitDelegateForTraversal<Key,Obj> {
        public void  visitForInorder
                (DictionaryElement<Key,Obj> anElement, int aLevel) ;
        public void  visitForReverseOfInorder
                (DictionaryElement<Key,Obj> anElement, int aLevel) ;
}
```

Class "Dictionary" 의 수정 [2]

```
public class  Dictionary<Key extends Key, Obj> {
        .....

        private void  inorderRecursively (...) {...}
        public  void  inorder () {...}

        private void  reverseOfInorderRecursively
              (BinaryNode<DictionaryElement<Key,Obj>> aRootOfSubtree, int aLevel)
        {
              if ( aRootOfSubtree != null ) {
                    this.reverseOfInorderRecursively (aRootOfSubtree.right(), aLevel+1) ;
                    this.visitDelegate().visitForInorder (aRootOfSubtree.element(), aLevel) ;
                    this.reverseOfInorderRecursively (aRootOfSubtree.left(), aLevel+1) ;
              }
        }
        public void  reverseOfIorder () {
              this.reverseOfInorderRecursively (this.root(), 1) ;
        }
```

```
public interface  VisitDelegateForTraversal<Key,Obj> {
        public void  visitForInorder
              (DictionaryElement<Key,Obj> anElement, int aLevel) ;
        public void  visitForReverseOfInorder
              (DictionaryElement<Key,Obj> anElement, int aLevel) ;
}
```

사용자 쪽인 Class "AppController" 의 수정

AppController: Delegate 알려주기

```
public class  AppController implements VisitDelegateForTraversal<Integer, Integer>
{
      // Constants
      private static final int
            DEFAULT_DATA_SIZE = 10 ;


      // Private instance variables
      private Dictionary<Integer,Integer> _dictionary ;
      private int[]  _list ;

      ...

      public void  run() {
            this. _list = DataGenerator.randomList (DEFAULT_DATA_SIZE) ;

            AppView.outputLine(...) ;
            this._dictionary = new Dictionary<Integer, Integer>() ;

            this._dictionary.setVisitDelegate (this) ;

            this.addToBinarySearchTreeAndShowShape() ;
            this.showInorderOfBinarySearchTree() ;
            this.removeFromBinarySearchTreeAndShowShape() ;
      }
```

AppController 안에 call back method 를 구현하겠다는 interface 의 선언이다. AppController 객체 스스로를 visit delegate 객체로 선언하고 있는 것이다.

Dictionary 객체에게 Visit delegate 가 어느 객체인지를 알려준다. 여기서는 AppController 객체 자신이므로 this 를 전달한다.

AppController: VisitDelegateForTraversal 의 구현

```
public interface  VisitDelegateForTraversal<Key,Obj> {
        public void  visitForInorder
                (DictionaryElement<Key,Obj> anElement, int aLevel) ;
        public void  visitForReverseOfInorder
                (DictionaryElement<Key,Obj> anElement, int aLevel) ;
}
```

```
public class  AppController  implements VisitDelegateForTraversal<Integer, Integer>
{
        ……
        public void  run() {
                ……
                this._dictionary.setVisitDelegate (this) ; // Dictionary 객체에게 this 가 Delegate 임을 알려준다.
                ……
        }

        @Override
        public void  visitForInorder (DictionaryElement<Integer, Integer> anElement, int aLevel)
        {
                // TODO Auto-generated method stub
                …… // Inorder 탐색을 하는 동안 원소를 방문했을 때 해야 할 일
        }

        @Override
        public void  visitForReverseOfInorder (DictionaryElement<Integer, Integer> anElement, int aLevel)
        {
                // TODO Auto-generated method stub
                …… // Inorder 의 역순으로 탐색하는 동안 원소를 방문했을 때 해야 할 일
        }
}
```

End of

"Dictionary (2)"