사전 (Dictionary)

강 지 훈
jhkang@cnu.ac.kr

충남대학교 컴퓨터공학과

사전 (Dictionary)이란?

개요

- 사전이란?
  - (단어, 설명)의 쌍들을 모아놓은 것.
  - 단어가 주어지면 그 단어의 설명을 찾는다.

- 또 다른 예는?
  - (학번, 성적)

- 일반화 하면?
  - (key, object)의 쌍을 모아 놓은 집합.
  - Key 가 주어지면, 그에 해당하는 object 를 찾는다.
  - key 값은 유일하다.

Class "Dictionary"

Dictionary의 공개함수

Dictionary 객체 사용법

- public Dictionary() ;

- public boolean isEmpty ( ) ;
- public booean isFull () ;
- public int size() ;

- public boolean keyDoesExist (Key aKey) ;
- public Object objectForKey (Key aKey) ;

- public boolean addKeyAndObject (Key aKey, Obj anObject) ;
- public Obj removeObjectForKey (Key aKey) ;
- public boolean replaceObjectForKey (Obj aNewObject, Key aKey) ;
- public void clear ( ) ;

이진검색트리
(Binary Search Trees)

이진 검색 트리

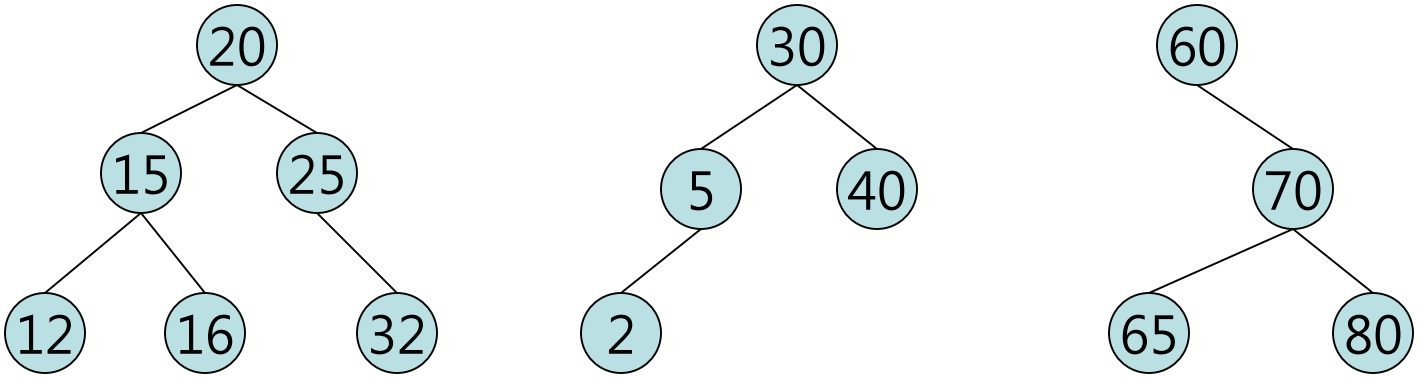
- 임의의 원소의 삽입/삭제/검색

- 시간복잡도 비교

	정렬되지 않은 배열	정렬된 배열	이진검색트리
I삽입	O(1)	O(n)	O(log n)
삭제	O(n)	O(n)	O(log n)
검색	O(n)	O(log n)	O(log n)

이진검색트리 (Binary Search Trees)

- 이진검색트리는 이진트리이다. 비어 있을 수 있으며, 만일 비어 있지 않다면 다음의 조건을 만족해야 한다:
  1. 모든 원소는 키를 가지고 있으며, 어떠한 원소도 동일한 키를 가지고 있지 않다. 즉, 키는 유일(unique)하다.
  2. 트리의 루트의 키는 비어있지 않은 왼쪽 부트리에 있는 키들보다 크다.
  3. 트리의 루트의 키는 비어있지 않은 오른쪽 부트리에 있는 키들보다 작다.
  4. 왼쪽 부트리와 오른쪽 부트리는 또한 이진검색트리이다.



검색 알고리즘: [재귀적으로]

```
BinaryNode search (BinaryNode currentRoot, int aGivenKey)
{
    if ( currentRoot != null ) {
        if ( aGivenKey == currentRoot.element().key() )
            return  currentRoot ;
        else if (key < currentRoot.element().key() )
            return  search (currentRoot.left(), aGivenkey) ;
        else
            return  search (currentRoot.right(), aGivenkey) ;
     }
    else {
        return  null ;
    }
}
```

- 검색 시간 복잡도: O(h)
  - h: 이진검색트리의 높이

```
public class Element {
        private int _key ;
        ..... // 다른 변수들

        public int          kye() {...} ;
        public void setKey() {...} ;
        ..... // 다른 공개함수들
}

public class BinaryNode {
        private Element  _element};
        private BinaryNode      _left;
        private BinaryNode      _right ;

        public Element          element() {...} ;
        public void         setElement() {...} ;
        public BinaryNode       left() {...} ;
        public void         setLeft() {...} ;
        public BinaryNode       right() {...} ;
        public void         setRight() {...} ;
}
```

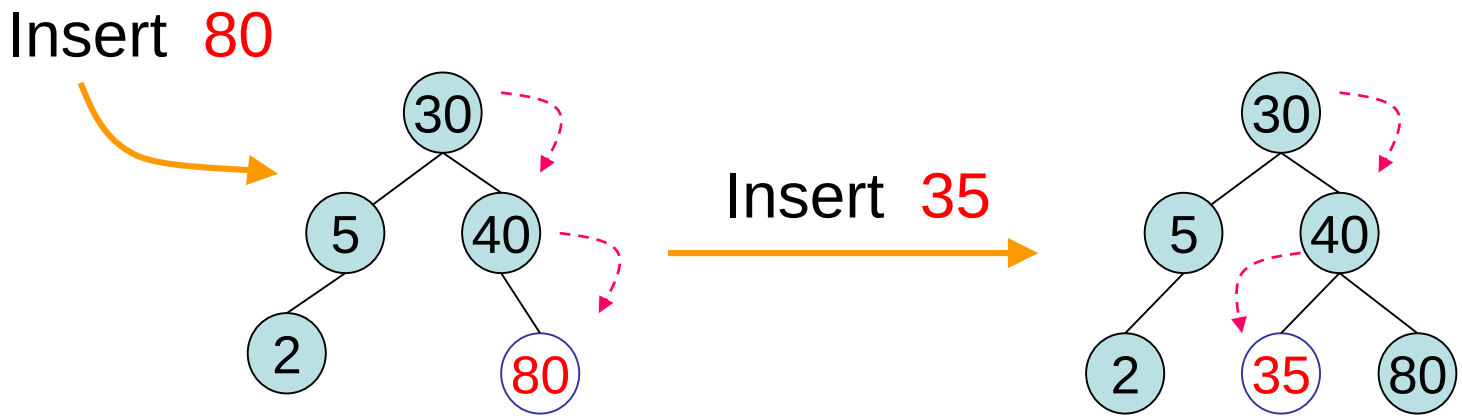
검색 알고리즘: [반복적으로]

```
BinaryNode search (BinaryNode currentRoot, int aGivenKey)
{
    while ( currentRoot != null ) {
        if ( aGivenKey == currentRoot.element().key() )
            return  currentRoot ;
        else if ( aGivenKey < currentRoot.element().key() )
            currentRoot = currentRoot.left() ;
        else
            currentRoot = currentRoot.right() ;
    }
    return null ;
}
```

- 검색 시간 복잡도: O(h)
  - h: 이진검색트리의 높이

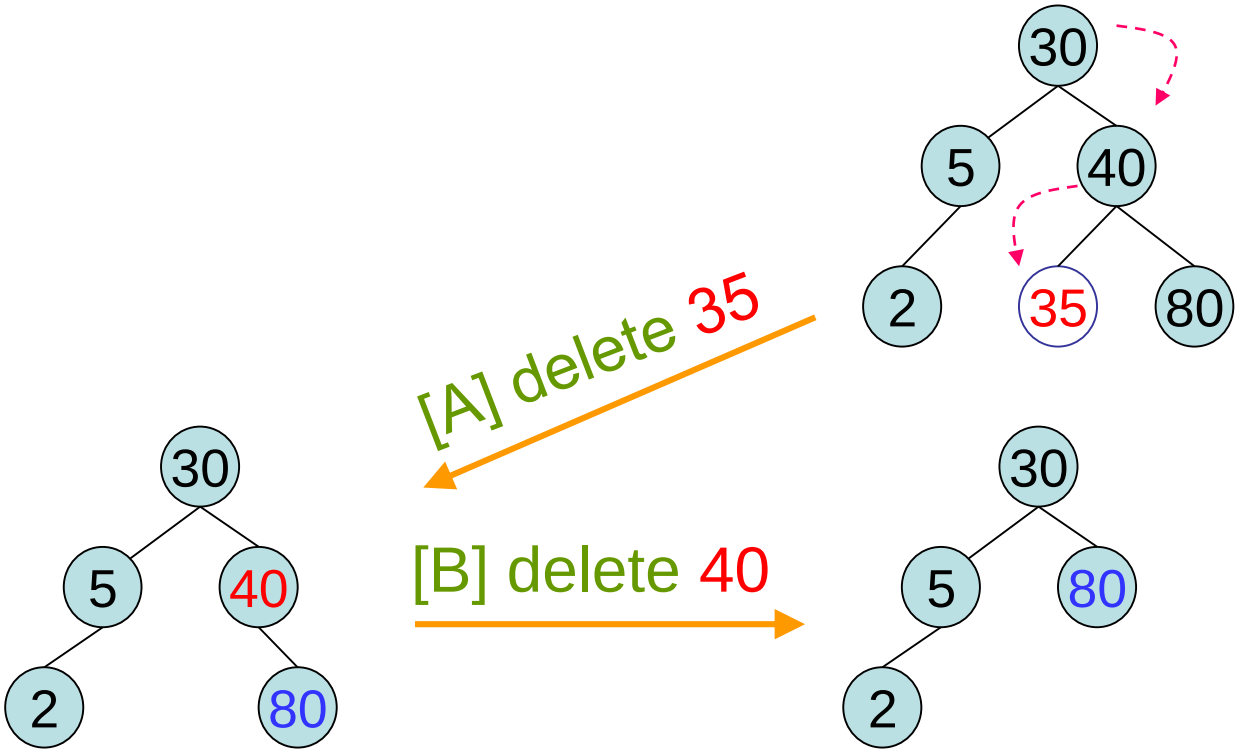
```
public class Element {
        private int _key ;
        ..... // 다른 변수들

        public int          kye() {...} ;
        public void setKey() {...} ;
        ..... // 다른 공개함수들
}

public class BinaryNode {
        private Element  _element};
        private BinaryNode      _left;
        private BinaryNode      _right ;

        public Element          element() {...} ;
        public void         setElement() {...} ;
        public BinaryNode       left() {...} ;
        public void         setLeft() {...} ;
        public BinaryNode       right() {...} ;
        public void         setRight() {...} ;
}
```

삽입



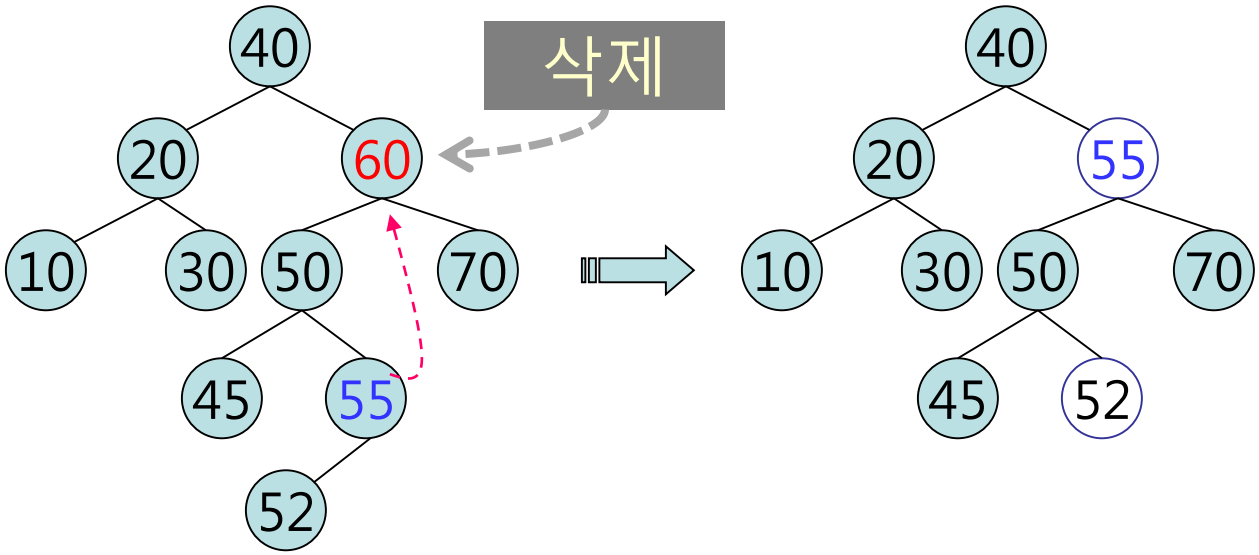
- 삽입의 시간복잡도: O(h)
  - h: 이진검색트리의 높이

삭제



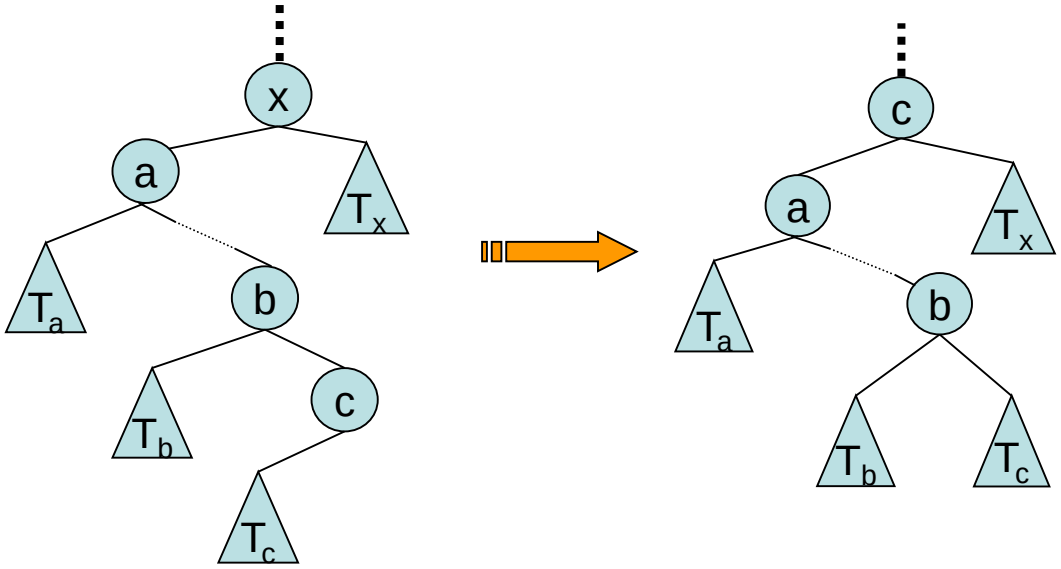
삭제

- 잎 노드: 바로 삭제
- 자식이 하나인 노드: 해당 노드는 삭제하고 그 자리에 자식 노드를 갖다 놓는다
- 자식이 둘인 노드: 잎 노드나 자식이 하나인 노드의 삭제의 문제로 변환



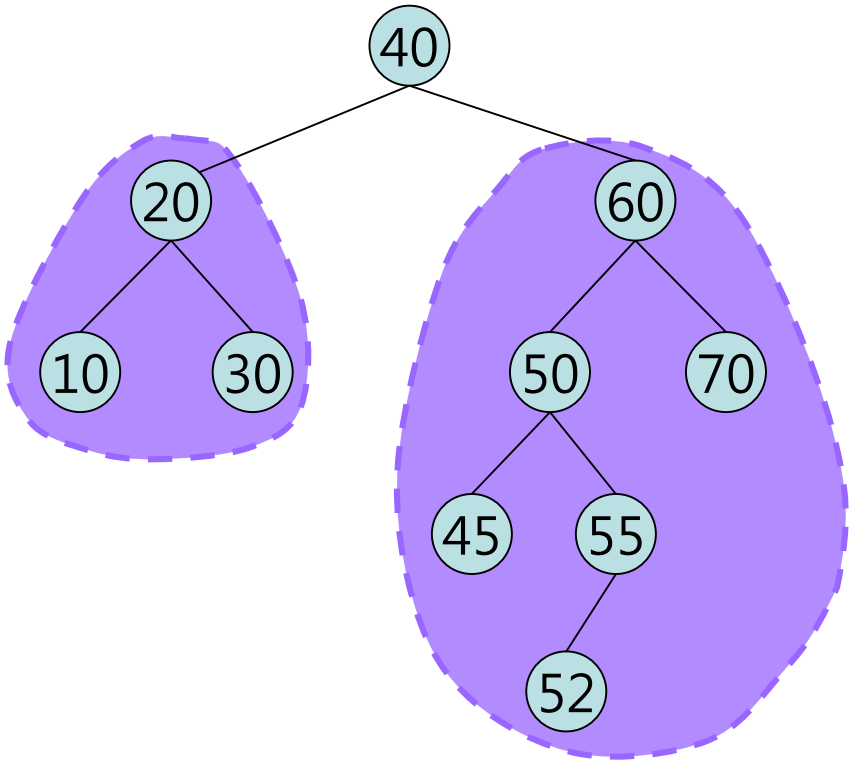
내부 노드 X의 삭제

- X의 왼쪽 부트리에서 가장 큰 값을 갖는 노드로 대체 (또는 X의 오른쪽부트리에서 가장 작은 값을 갖는 노드로 대체)

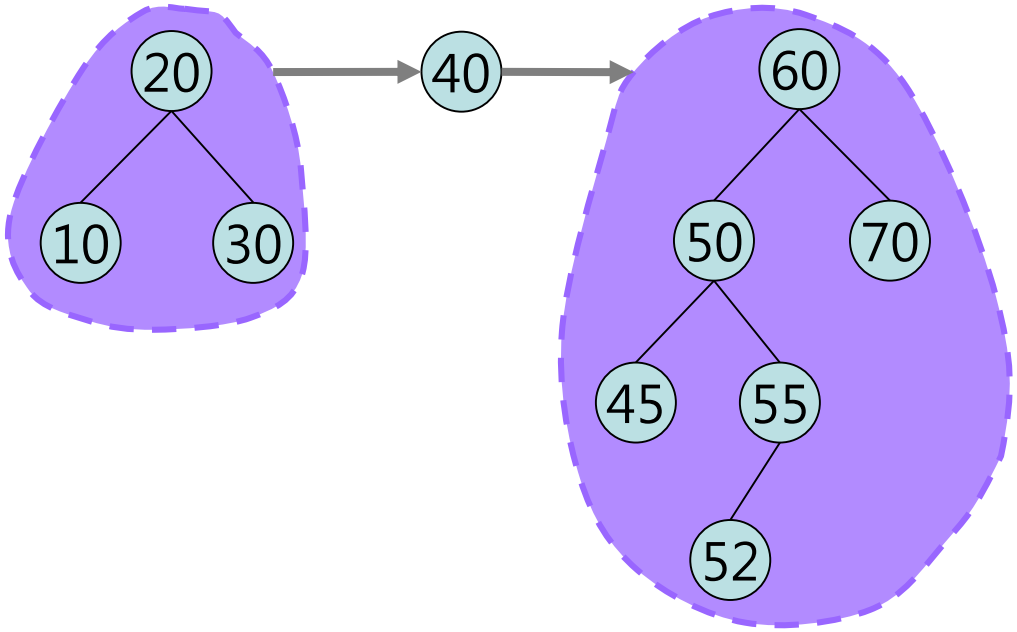


- 삭제의 시간복잡도: O(log n)
  - 트리의 높이 h일 때, O(h)
  - 삽입이나 삭제가 무작위로 발생하면, h = O(log n)

BST에서 중위 탐색을 하면 ?



BST에서 중위 탐색을 하면 ?



사전의 (Key, Object) 쌍을 위한 Class "Element"

Class Element: 공개함수

```
public class Element<Key,Obj>
{
    public Element () ;
    public Element (Key aKey, Obj  anObject) ;

    public Key      key () ;
    public void     setKey (Key aKey) ;

    public Obj      object () ;
    public void     setObject (Obj anObject) ;
}
```

Element의 구현: 비공개 인스턴스 변수

```
public class Element<Key, Obj>
{
    // 비공개 인스턴스 변수
    private Key      _key ;
    private Obj      _object ;
```

Element 구현: 생성자

```
public class Element<Key,Obj>
{
    ......
    public Element ()
    {
        this._key = null ;
        this._object = null ;
    }

    public Element (Key aKey, Obj anObject) ;
    {
        this._key = aKey ;
        this._object = anObject ;
    }
```

Element 구현: 공개함수

```
public class Element<Key,Obj>
{
    ......
    public Key   key () ;
    {
        return   this._key ;
    }

    public void   setKey (Key aKey) ;
    {
         this._key =   aKey ;
    }

    public Obj   object () ;
    {
        return   this._object ;
    }

    public void   setObject (Obj anObject) ;
    {
         this._object =   anObject ;
    }
```

사전 구현에 사용할 이진검색트리의 노드:

Class "BinaryNode"

BinaryNode의 공개함수

```
pubic class BinaryNode<T>
{
    ……

    public BinaryNode () ;
    public BinaryNode (T anElement, BinaryNode aLeft, BinaryNode aRght) ;

    public T              element () ;
    public void           setElement (T anElement) ;

    public BinaryNode     left () ;
    public void           setLeft (BinaryNode aLeft) ;

    public BinaryNode     right () ;
    public void           setRight (BinaryNode aRight) ;
}
```

BinaryNode: 비공개 인스턴스 변수와 생성자

```
public class BinaryNode<T>
{
    // 비공개 인스턴스 변수
    private T                   _element ;
    private BinaryNode          _left ;
    private BinaryNode          _right ;


    public BinaryNode ()
    {
        this._element = null ;
        this._left = null ;
        this._right= null ;
    }

    public BinaryNode (T anElement, BinaryNode aLeft, BinaryNode aRight)
    {
        this._element= anElement ;
        this._left = aLeft ;
        this._right= aRight ;
    }
```

BinaryNode: Setter들과 Getter들의 구현

```
public T    element()
{
      return   this._left ;
}

public void   setElement (T anElement)
{
      this._element= anElement ;
}

public BinaryNode   left ()
{
      return   this._left ;
}

public void   setLeft (BinaryNode aLeft)
{
      this._left = aLeft ;
}


public BinaryNode   right ()
{
      return   this._right ;
}

public void   setRight (BinaryNode aRight)
{
      this._right = aRight ;
}
```

Class "Dictionary"

Dictionary의 공개함수

- Dictionary 객체 사용법
  - public Dictionary() ;
  - public boolean isEmpty ( ) ;
  - public booean 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 (Obj aNewObject, Key aKey) ;
  - public void clear ( ) ;

Dictionary: 비공개 인스턴스 변수

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

Dictionary의 생성자

```
public class Dictionary<Key,Obj>
{
    // 생성자
    public Dictionary( )
    {
        this._size =  0 ;
        this._root = null ;
    }
```

Dictionary : 상태 알아보기

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

    // 상태 알아보기
    public boolean  isEmpty()
    {
        return  (this._root == null) ;
    }

    public boolean  isFull ()
    {
        return  false ;
    }

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

Dictionary: keyDoesExist() [recursive]

```
public boolean     keyDoesExist (Key aKey)
{
    return   this.keyDoesExistInTree (this._root, aKey) ;
}

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

Dictionary: keyDoesExist() [nonrecursive]

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

Dictionary: objectForKey()

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

Dictionary: addKeyAndObject() [recursive]

```
public boolean  addKeyAndObject (Key aKey, Obj anObject)
{
    if (this._root == null) {
        this._root= new BinaryNode((new Element(aKey, anObject)), null, null) ;
        this._size ++ ;
        return true ;
    } else {
        return addKeyAndObjectToSubtree (this._root, aKey, anObject) ;
    }
}
```

Dictionary: addKeyAndObjectToSubtree()

```
private boolean  addKeyAndObjectToSubtree (BinarNode currentRoot, Key aKey, Obj anObject)
{
      BinaryNode  newNode = null ;
      if (currentRoot.element().key().compareTo(givenKey) == 0) {
            return false ;
      }
      else if (currentRoot.element().key().compareTo(givenKey) > 0) {
            if (currentRoot.left() == null) {
                  newNode = new BinaryNode((new Element(aKey, anObject)), null, null) ;
                  currentRoot.setLeft(newNode) ;
                  this._size++ ;
                  return true ;
            }
            else {
                  return  addKeyAndObjectToSubtree(currentRoot.left(), aKey, anObject) ;
            }
      }
      else {
            if (currentRoot.right() == null) {
                  newNode = new BinaryNode((new Element(aKey,anObject)), null, null) ;
                  currentRoot.setRight(newNode) ;
                  this._size++ ;
                  return true ;
            }
            else {
                   return  addKeyAndObjectToSubtree(currentRoot.right(), aKey, anObject) ;
            }
      }
}
```

Dictionary: addKeyAndObject() [nonrecursive]

```
public boolean  addKeyAndObject (Key aKey, Obj anObject)
{
      if (this._root == null) {
            this._root= new BinaryNode((new Element(aKey,anObject)), null, null) ;
            this._size ++ ;
            return true ;
      }
      BinaryNode  current = this._root ;
      BinaryNode  newNode = null ;
      while ( true ) {
            if (current.element().key().compareTo(givenKey) == 0) {
                  return false ;
            }
            else if (current.element().key().compareTo(givenKey) < 0) {
                  if (current.right() == null) {
                        newNode = new BinaryNode((new Element(aKey, anObject)), null, null) ;
                        current.setRight(newNode) ;
                        this._size++ ;
                        return true ;
                  }
                  current = current.right() ;
            }
            else {
                  if (current.right() == null) {
                        newNode = new BinaryNode((new Element(aKey, anObject)), null, null) ;
                        current.setLeft(newNode) ;
                        this._size++ ;
                        return true ;
                  }
                  current = current.left() ;
            }
      }
}
```

Dictionary: removeObjectForKey()

```
public Object  removeObjectForKey (Key givenKey)
{
      Obj  removedObject = null ;
      if ( this._root == null ) {
            return null ;
      } else ( this._root.element().key().compareTo(givenKey) == 0 ) {
            removedObject = this._root.element().object() ;
            if ( (this._root.left() == null) && ( this._root.right() == null) ) { // root만 있는tree
                  this._root = null ;
            }
            else if ( this._root.left() == null )  { // root의 left tree 가 없다
                  this._root = this._root.right() ;
            } else { // root의 right tree 가 없다
                  this._root = this._root.left() ;
            }
            else { // child의 left tree, right tree가 모두 있다
                  BinaryNode  newRoot = removeRightMostOfLeftTree(this._root) ;
                  newRoot.setLeft (this._root.left()) ;
                  newRoot.setRight (this._root.right()) ;
                  this._root = newRoot ;
            }
            this._size -- ;
            return  removedObject ;
      }
      else {
            return  removeObjectForKeyFromSubtree (tjhis._root, givenKey) ;
      }
}
```

Dictionary: removeRightMostOfLeftTree()

```
private  BinaryNode  removeRightMostOfLeftTree (BinaryNode currentRoot)
{
    // 현재의 currentRoot를 대체할 노드인, 왼쪽 트리의 가장 오른쪽 노드를 삭제하여 얻는다.
    // call 하는 시점에, currentRoot.left() 는 null 이 아니다

    BinaryNode  leftOfCurrentRoot = currentRoot.left() ;
    if ( leftOfCurrentRoot== null ) {
        return  null ;
    }
    if ( leftOfCurrentRoot.right() == null ) {
        currentRoot.setLeft (leftOfCurrentRoot.left()) ;
        return  leftOfCurrentRoot ;
    } else {
        BinaryNode  parentOfRightMost = leftOfCurrentRoot ;
        BinaryNode  rightMost = leftOfCurrentRoot.right() ;
        while ( rightMost.right() != null ) {
            parentOfRightMost = rightMost ;
            rightMost = rightMost.right() ;
        }
        parentOfRightMost.setRight (rightMost.left()) ;
        rightMost.setLeft (null) ;
        return  rightMost ;
    }
}
```

Dictionary: removeObjectForKeyFromSubtree() [1]

```
private Obj  removeObjectForKeyFromSubtree (BinaryNode currentRoot, Key givenKey)
{
    // 이 시점에, currentRoot 는 null이 아니고, currentRoot의 key는 givenKey와 일치하지 않는다.
    if ( currentRoot.element().key().compareTo(givenKey) > 0 ) { //left subtree에서 삭제해야 한다
        child = currentRoot.left () ;
        if ( child == null ) {
            return null ;
        }
        else {
            if ( child.element().key().compareTo(givenKey) == 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
                    BinaryNode newChild = removeRightMostOfLeftTree(child) ;
                    newChild.setLeft(child.left()) ;
                    newChild.setRight(child.right()) ;
                    currentRoot.setLeft (newChild) ;
                }
                this._size -- ;
                return  removedObject ;
            }
            else {
                return  removeObjectForKeyFromSubtree (child, givenKey) ;
            }
        }
    }
    else { // right subtree에서 삭제해야 한다
```

Dictionary: removeObjectForKeyFromSubtree() [2]

```
private Obj  removeObjectForKeyFromSubtree (BinaryNode currentRoot, Key givenKey)
{
    if ( currentRoot.element().key().compareTo(givenKey) > 0 ) {// left subtree에서 삭제해야 한다
        ......
    }
    else { // right subtree에서 삭제해야 한다
        child = currentRoot.right() ;
        if ( child == null) {
            return null ;
        }
        else {
            if ( child.element().key().compareTo(givenKey) == 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
                    BinaryNode newChild = removeRightMostNodeOfLeftTree(child) ;
                    newChild.setLeft(child.left()) ;
                    newChild.setRight(child.right()) ;
                    currentRoot.setRight (newChild) ;
                }
                this._size -- ;
                return  removedObject ;
            }
            else {
                return  removeObjectForKeyFromSubtree (child, givenKey) ;
        }
    }
}
```

Dictionary: replaceObjectForKey()

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

Dictionary: clear ( )

```
public void  clear ( )
{
    this._size = 0 ;
    this._root = null ;
}
```

실습:사전 성능 평가

실습: 사전 성능 평가

- 사전을 다섯 가지로 구현하여 성능을 측정한다.
- 구현 방법의 종류
  - Unsorted Array
  - Sorted Array (Increasing order)
  - Unsorted Linked List
  - Sorted Linked List (Increasing order)
  - Binary Search Tree
- 성능 측정 기능
  - 삽입
  - 검색
  - 삭제

실습: 사전 성능 평가

- 입력 :
  - 없음
  - 필요한 데이터는 프로그램에서 생성
- 출력 : 성능 측정 결과
  - 데이터 크기 변화에 따른 성능 측정 결과
- 데이터 크기
  - 2000,4000,6000,8000,10000
- 데이터 생성
  - Random number를 생성하여 사용한다.

“사전” [끝]