데이터구조설계

학과 : 컴퓨터정보공학부

학번 : 2017202029

이름 : 전 효 희

1. Introduction

BST(Binary Search Tree)는 작은 값이 왼쪽 자식으로, 큰 값이 오른쪽 자식으로 구분되어 구성되는 binary tree이다. 일반적으로 BST의 각 노드는 노드의 위치를 나타내는 값인 Key값과, 그것이 나타내는 data를 저장하고 있는 element 이 두 가지를 가지고 있다. 따라서 사용자가 지정한 순서에 따라 insert 및 delete가 가능하고, 그 순서에 따라 출력이 가능하다는 장점이 있다. 또한 찾으려는 data를 통한 탐색 또한 가능하다.

이번 과제는 이러한 BST의 insert(입력), delete(삭제), get(순환적탐색과 반복적탐색)을 c++언어로 각각 구현하고, 이 결과를 출력해 자료구조를 이해하는 과제였다. 각 기능을 구현했으며, 탐색 시에 존재하지 않는 노드에 대해서 안내를 출력하는 예외처리를 진행했다.

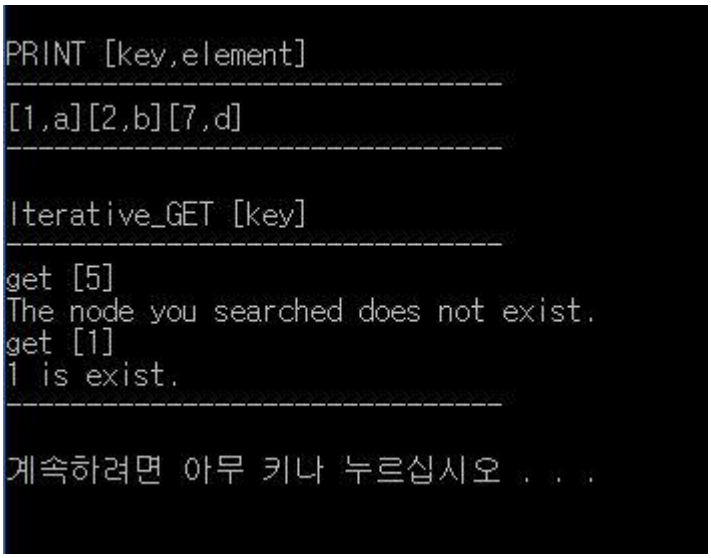
2. Result screen

<그림 1> 전체 결과

- Insert, print(inorder로), delete, get(recursive), get(iterative)을 수행하였다.

<그림 2> insert, print, delete ,get(순회) 결과

- Insert: key는 int값으로, element는 char값으로 순서에 상관없이 입력을 진행했다.
- Print: BST는 inorder로 출력하면 작은 값부터 출력되므로 inorder로 출력을 진행했다.
- Delete: key 5에 대해 삭제를 진행했다.
- Get: 삭제한 5가 존재하는지 확인을 진행하고, 존재하는 노드에 대해서도 탐색을 진행했다.



<그림 3> print 및 get(반복)의 결과

- Print: 5가 삭제된 후 변화한 tree에 따라 inorder 출력의 변화를 확인할 수 있다.
- Get: 반복적 탐색의 경우에도 순환적 탐색과 동일한 결과를 얻을 수 있음을 확인할 수 있다.

3. Source code

```
#pragma once
#include <iostream>;
using namespace std;
template <class K, class E>
class Dictionary
{
public:
    virtual pair<K, E>*Get(const K&) const = 0;
    virtual void Insert(const pair<K, E>&) = 0;
    virtual void Delete(const K&) = 0;

    Dictionary() { ; }
    ~Dictionary() { ; }
};
template <class T>
class TreeNode
{
public:
    T data;
    TreeNode<T> *leftChild;
    TreeNode<T> *rightChild;
    TreeNode(T temp) { data=temp, leftChild=NULL, rightChild=NULL; }

};
```

<그림1>가상 함수의 선언과 BST에 사용될 트리 노드의 선언

```
template <class K, class E>
class BST: public Dictionary <K,E>
{
public:
    BST() {
        root = NULL;
    }
    TreeNode<pair<K, E>> *root;

    pair<K, E>* Get(const K&k) const;
    pair<K, E>* Get(TreeNode<pair<K, E> >*p, const K& k)const;

    pair<K, E>* Iterative_Get(const K& k);
    void Insert(const pair<K, E>&thePair);
    void Delete(const K& k);
    void InOrder(TreeNode<pair<K, E>> *t);
};
```

<그림 2> Dictionary를 상속받은 BST class 선언 및 함수 구성

- Get은 순환적 탐색, Iterative_Get은 반복적 탐색이다.

```
46      template <class K, class E>
47      void BST<K, E>::InOrder(TreeNode<pair<K, E>> *t)
48      {
49          if (t != NULL)
50          { InOrder(t->leftChild);
51            cout<<"["<< t->data.first<<","<<t->data.second<<"]";
52            InOrder(t->rightChild);
53          }
54      }
55
56      template <class K, class E>
57      pair<K, E>* BST<K, E>::Get(const K& k) const
58      {  return Get(root, k); }
59
60      template <class K, class E>
61      pair<K, E>* BST<K, E>::Get(TreeNode<pair<K, E> >*p, const K& k)const
62      {
63          if (p == NULL)
64          {
65              cout << "The node you searched does not exist."<<endl;
66              return NULL; }
67          if(k < p->data.first) return Get(p->leftChild, k);
68          if(k > p->data.first) return Get(p->rightChild, k);
69          return &p->data;
70      }
```

<그림 3> InOrder와 Get 함수 구현

- inorder함수는 루트를 인자로 받아 트리를 순회하도록 구성
- Get함수는 보호되어 되어있음.
- 해당 노드가 없을 경우 존재하지 않는다는 안내문 출력.

```
template <class K, class E>
pair<K, E>* BST<K, E>::Iterative_Get(const K& k)
{
    TreeNode <pair<K, E>> *currentNode = root;
    while (currentNode)
    {
        if (k < currentNode->data.first)
            currentNode = currentNode->leftChild;
        else if (k > currentNode->data.first)
            currentNode = currentNode->rightChild;
        else
            return &currentNode->data;
    }

    cout << "The node you searched does not exist." << endl;
    return 0;
}
```

<그림 4> Iterative Get 구현

- 반복적 탐색의 경우 루트를 가져와 currentNode에 저장하고, currentNode가 key보다 크냐 작냐에 따라 currentNode를 변경하며 탐색.

- 탐색 후 결과가 없을 경우 존재하지 않는다는 안내문 표출.

```
template<class K,class E>
void BST<K, E>::Insert(const pair<K, E> &thePair) //새로운 페어 입력
{
    TreeNode<pair<K, E>> *p = root, *pp = NULL; //pp는 p의 부모
    while (p)
    {
        pp = p;
        if (thePair.first < p->data.first) //새 입력key가 p보다작을때
            p = p->leftChild;//왼쪽자식으로 이동
        else if (thePair.first > p->data.first) //p보다 크면
            p = p->rightChild; //오른쪽자식으로 이동
        else //p와 새 입력의 key가 같으면
        {
        p->data.second = thePair.second; //새 입력으로 element 교체
            return;
        }
    } //loop 다 돌고 나면 pp는 리프노드, p는 null
    p = new TreeNode<pair<K, E>> (thePair); //할당 및 생성
    if (root != NULL)
    {
        if (thePair.first < pp->data.first) //부모보다 작으면
            pp->leftChild = p; //왼쪽
        else
            pp->rightChild = p;
    }
    else root = p;
}
```

<그림 5> insert의 구현

- 현재 값을 pp라는 부모에 저장 후 p를 변경하며 탐색을 진행한다.

- 적정한 위치를 찾으면 새로운 노드를 동적할당한다.

- 이후 부모인 pp의 자식위치 (작으면 왼쪽, 크면 오른쪽)에 배치한다.

```
121  void BST <K, E>::Delete(const K &k)
122  {
123      TreeNode <pair< K, E >> *p = root, *q = 0;
124      while (p && k != p->data.first)//p가 루트가 아니고 key와 p의 key가 다른 동안
125      {
126          q = p; //q는 p 부모
127          if (k < p->data.first) //p의 key가 key보다 크면
128              p = p->leftChild; //왼쪽자식(더작은)이동
129          else p = p->rightChild;
130      }
131      if (p == 0) return;
132      if (p->leftChild == 0 && p->rightChild == 0) //p가 리프노드
133      {
134          if (q == 0) //p가 루트일때
135              root = 0; //삭제할 것이니 루트 지움
136          else if (q->leftChild == p)
137              q->leftChild = 0; //왼쪽자식 지움
138          else
139              q->rightChild = 0; //오른쪽자식 지움
140          delete p; //p 삭제.
141      }
142      if (p->leftChild == 0) //p에게 오른쪽자식만 있을때
143      {
144          if (q == 0)//p가 루트일 때
145              root = p->rightChild;
146          else if (q->leftChild == p) //p가 왼쪽자식이면
147              q->leftChild = p->leftChild; //왼쪽에 p의 왼쪽자식 배치
148          else //p가 오른쪽자식
149              q->rightChild = p->rightChild; //오른쪽에 p의 오른자식 배치
150          delete p; //p삭제
151      }
```

<그림 6> delete 구현 1

- 현재 값을 q라는 부모에 저장 후 p를 변경하며 탐색을 진행한다 .
- P를 찾으면 네 가지 방식에 나눠 삭제를 진행한다.
- 첫 번째는 P가 리프노드인 경우이다. 이 경우 P가 루트라면 루트를 지우고, 그 외의 경우 q의 자식을 지우는 방식으로 삭제한다.
- 두 번째는 p에게 오른쪽 자식만 있다면, p의 오른쪽 자식을 p의 자리에 옮기고 p를 삭제한다.

```
    if (p->rightChild == 0)//p가 왼쪽만 있을때
    {
        if (q == 0)
            root = p->leftChild;
        else if (q->leftChild == p)
            q->leftChild = p->leftChild;
        else
            q->rightChild = p->rightChild;
        delete p;
    }


    //p에게 왼쪽 오른쪽 다있을 때
    TreeNode<pair<K, E>> *prevprev = p, *prev = p->rightChild,
        *curr = p->rightChild->leftChild;

    /*가장 작은 값 찾기 위함.*/
    while (curr)
    {
        prevprev = prev;
        prev = curr; //다 돌고난 후 prev는 p의 오른쪽에서 가장 작은 값.
        curr = curr->leftChild;//다 돌고난 후 curr은 0
    }

    p->data = prev->data;//prev와 p의 데이터 바꿈->이후 prev를 삭제.

    if (prevprev == p) //loop를 돌지 않았으면 prev는 오른쪽 자식이므로.
        prevprev->rightChild = prev->rightChild;
    else //loop를 돌고 난 후 prev는 prevprev의 왼쪽자식이므로.
        prevprev->leftChild = prev->rightChild;
    delete prev; //prev 삭제.
}
```

- 세 번째는 p에게 왼쪽 자식만 있는 경우이다. 이 경우 또한 왼쪽자식에게 p자리를 물려주고 p를 삭제한다.

- 네 번째로 이후 p에게 왼쪽 오른쪽이 모두 있을 경우, p에게 가장 가까운 값을 p로 옮기게 된다. 본 과제에서는 p보다 큰 노드 중 가장 작은 노드를(prev) 선택했다. 선택한 노드를 찾으면 p노드와 데이터를 바꾼 후, 그 노드의 자리를 노드의 자식으로 대체한 후 prev를 삭제하는 방식으로 진행했다.

4. 고찰

과제 해결에 있어서 pair 및 템플릿을 적용하는 과정에서 실수가 잦았다. Pair가 있다면 그 각각을 처리함에 있어서 상세하게 코드를 작성해야 함을 알게 되었다.

또한 main에서 인스턴스 생성시에 생성이 되지 않아 진행이 어려웠었다. 그 이유는 가상함수 때문이었다. Dictionary를 상속받아 BST 클래스를 만들었기 때문에, Dictionary의 가상함수를 정확히 같은 형태로 구현을 했어야 했으나, const 등의 키워드를 빼고 구현하여 인스턴스가 계속 생성되지 않는 문제가 있었다. 가상함수가 있다면 반드시 같은 형태로 구현해야 인스턴스를 만들 수 있음을 알게 되었다.