Binary Search Tree와 AVL Tree의 성능 비교

자료구조 01분반

Assignment 4

2021320122 김정우

1. Binary Tree와 AVL Tree의 Performance 비교

1) 비교 방법

main.cpp의 nElem을 1000, 10000, 100000, 1000000으로 점차 키워가며, Binary Search Tree와 AVL Tree 각각에서, 주어진 insert, find 반복문을 실행한다.

(nElem = 1000000의 경우, Binary Search Tree에서 segmentation fault가 발생하기에 AVL Tree만 동작시켰다.)

데이터의 종류는 두 가지로, Random order와 Skewed order를 사용한다.

- Random ordered data 제작은 다음과 같이 이루어졌다.

  (keyindex배열의 크기는 2*nElem이다. randn변수의 자료형은 int형이다.)

```
for(int i=0; i<nElem*2; i++)
{
    while(1){
        randn = std::rand() % (nElem*2);
        if(keyindex[randn] == 0){
            keyindex[randn]++;
            break;
        }
    }
    key[i] = randn;
    val[i] = (float) std::rand()/RAND_MAX * 20000;
}
delete [] keyindex;
```

  [0 , 2*nElem) 범위의 숫자를, 중복되지 않도록 key배열에 random order로 배치하도록 작성한 프로그램이다.

- Skewed ordered data는 따로 key배열을 갱신하지 않고, 반복문의 반복자

i를 그대로 사용해 insert했다.

```
tm = clock();
for(int i=0; i<nElem; i++)
{
    avl2.insert(i, val[i]);
}
tm = clock() - tm;
```

Random ordered data의 경우, random order에 따라 프로그램의 동작 시간이 다를 수 있으므로, 4회 시행 후 평균치를 계산해 비교 및 분석에 사용하였다.

2) Time Performance 비교, 분석

① **실행 시간 (단위 : 초(s))**

***

실행 시간이 반복적으로 0.015625초가 나오는 경우, 그리고 실행 시간이 정확히 일치하는 경우가 빈번히 존재했다.
조사를 통해 '1. 컴퓨터 자체적으로 프로세스 간 timer interrupt라는 것이 존재' 하며 '2. 이것이 0.015625초 (1/2^64)라는' 사실을 알게 되었다.
이 개념을 정확히 이해하진 못했지만 실행 시간의 단위가 0.015625초이며, 실행 시간이 이보다 작을 경우 time interrupt의 영향으로 엄밀한 실행 시간이 측정되지 않는다는 것으로 받아들였다.
이에 4번 시행 중 2번 이상 나타나지 않는 0.015625초는 0으로 간주하고, 아래 결과를 작성한다. 또한 실행 시간이 정확히 일치하는 현상이 프로그램 자체의 오류가 아님을 밝힌다.

I. **Random ordered data**

RANDOM	**Binary Search Tree**		**AVL Tree**	
nElem	Running time of insert()	Running time of find()	Running time of insert()	Running time of find()
1,000	**0**	**0**	**0**	**0**
10,000	**0**	**0**	**0**	**0**
100,000	**0.09375**	**0.09375**	**0.1875**	**0.0625**
1,000,000	**1.078125**	**1.265625**	**2.390625**	**1.078125**

자료 해석 :

data 자체가 무작위로 배치되므로, Binary Search Tree도 일정 수준의 balance를 보장받을 수 있다. 따라서 AVL Tree의 rebalance함수가 insert와 find함수의 실행 시간을 단축시키는 효과는 미미하다고 보인다.

오히려 rebalance함수의 실행 시간이 AVL Tree에서 전체 실행 시간을 증가시키고 있다. 실제로 Binary Search Tree에서의 실행 시간이 AVL Tree의 실행 시간보다 더 적은 것을 확인할 수 있다.

II. **Skewed ordered data**

SKEWED	**Binary Search Tree**		**AVL Tree**	
nElem	Running time of insert()	Running time of find()	Running time of insert()	Running time of find()
1,000	**0.15625**	**0.15625**	**0**	**0**
10,000	**0.78125**	**1.578125**	**0.15625**	**0**
100,000	**108.859375**	**250.890625**	**0.171875**	**0.03125**
1,000,000	**unable**	**unable**	**1.640625**	**0.28125**

자료 해석 :

Random ordered data 와 비교하여, 확실히 AVL Tree 가 효율적임을 확인할 수 있다.

Skewed – ordered data 에 대해, AVL Tree 의 보장된 Balanced – height 이 insert 와 find 함수의 실행 시간에 대해 큰 효과를 주는 것을 확인할 수 있다.

+) n 개의 Skewed order data 대해 모두 insert 와 find 를 진행할 때,
Binary Search Tree 는 $O(n^2)$의 time complexity 를,
AVL Tree 는 insert 와 find 모두 O(n log n)의 time complexity 를 가진다.

이에 rough 하게 Binary Search Tree 의 연산들에 소모된 시간을 n^2으로 두고, AVL Tree 와 비교해 n log n 형태를 도출해내려 했지만, Big – O notation 의 상수를 고려하지 못할 뿐더러 rough 한 추정치의 한계로 불가능했다.

추가로, Binary Search Tree 에서 insert 과정보다 find 과정이 더 많은 시간을 소모하는데 이는 insert 과정을 시작할 때는 tree 의 크기가 0 인 반면, find 과정을 시작할 때는 tree 가 2*nElem 개의 크기로 완성되어 있기 때문인 것으로 보인다.

2. 분석을 마치며 – 구현에 어려움을 겪었던 점

1) Position과 Node의 혼동

두 가지 Tree를 구현함에 있어, Position(및 Iterator)과 그 내부의 Node 간 관계가 혼동되고, 둘을 조작하는 것이 익숙치 않아 어려 시행착오를 겪었다.

(특히 AVLTree.txx 파일의 restructrue을 구현할 때, node들을 연결하는 작업에서 Tree Position을 node처럼 생각하고 구현하다 오랜 시간을 허비했다.
Ex) a.v.right에 접근해야 하는데 a.right로 작성하는 등)

2) insert함수의 예외처리

expandedExternal 함수의 구조[1]를 간과해 발생한 문제이다.

key가 0인 Entry를 새롭게 insert할 경우, 좌측 최하단 leaf node의 left child까지 탐색하게 되는데, 이 경우 left child의 초기값이 0으로 설정되어 있어서, 'key값이 같은 경우 value만 갱신하는' case로 인식이 되었다. 이 때문에 inserter에서 expandedExternal()함수가 실행되지 않아 segmentation fault가 발생했다.

이를 해결하기 위해, key값이 0일 때 isExternal()함수를 실행해 이를 확인하는 조건문을 하나 추가했다. (하단 사진의 108~112 line)

[1] Leaf node에도 leftchild와 rightchild의 node가 존재하는 구조

```
template <typename E>          // insert utility
typename SearchTree<E>::TPos SearchTree<E>::inserter(const K& k, const V& x) {

  // ToDo
  TPos v = finder(k, root()); // search from virtual root

  if((*v).key() == k){
    if(k == 0 && v.isExternal()){
      T.expandExternal(v); // add new internal node
      (*v).setKey(k); (*v).setValue(x); // set entry
      n++; // one more entry
    }
    else{
      (*v).setValue(x);
    }
  }
  else{
    T.expandExternal(v); // add new internal node
    (*v).setKey(k); (*v).setValue(x); // set entry
    n++; // one more entry
  }

  return v;
}
```