02분반

파일처리론

BST (Binary Search Tree)

2142851

김형준

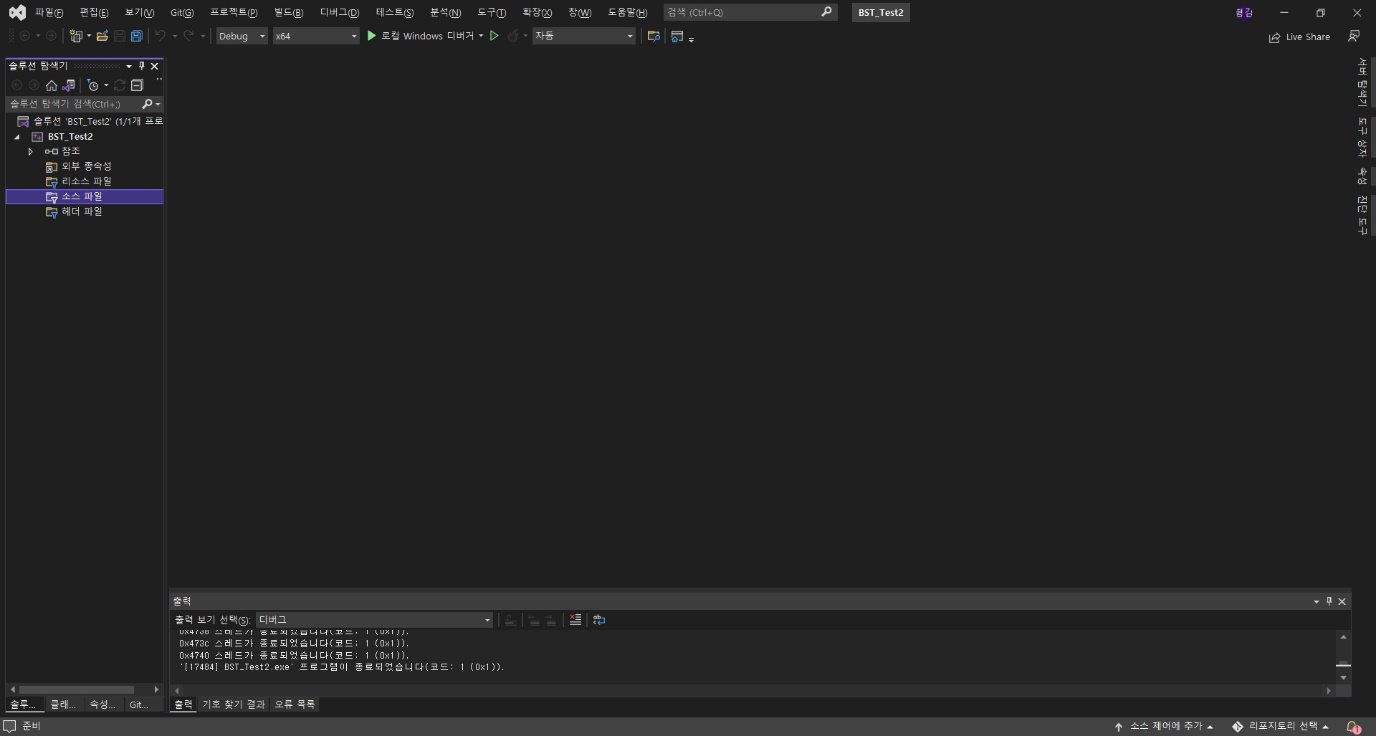
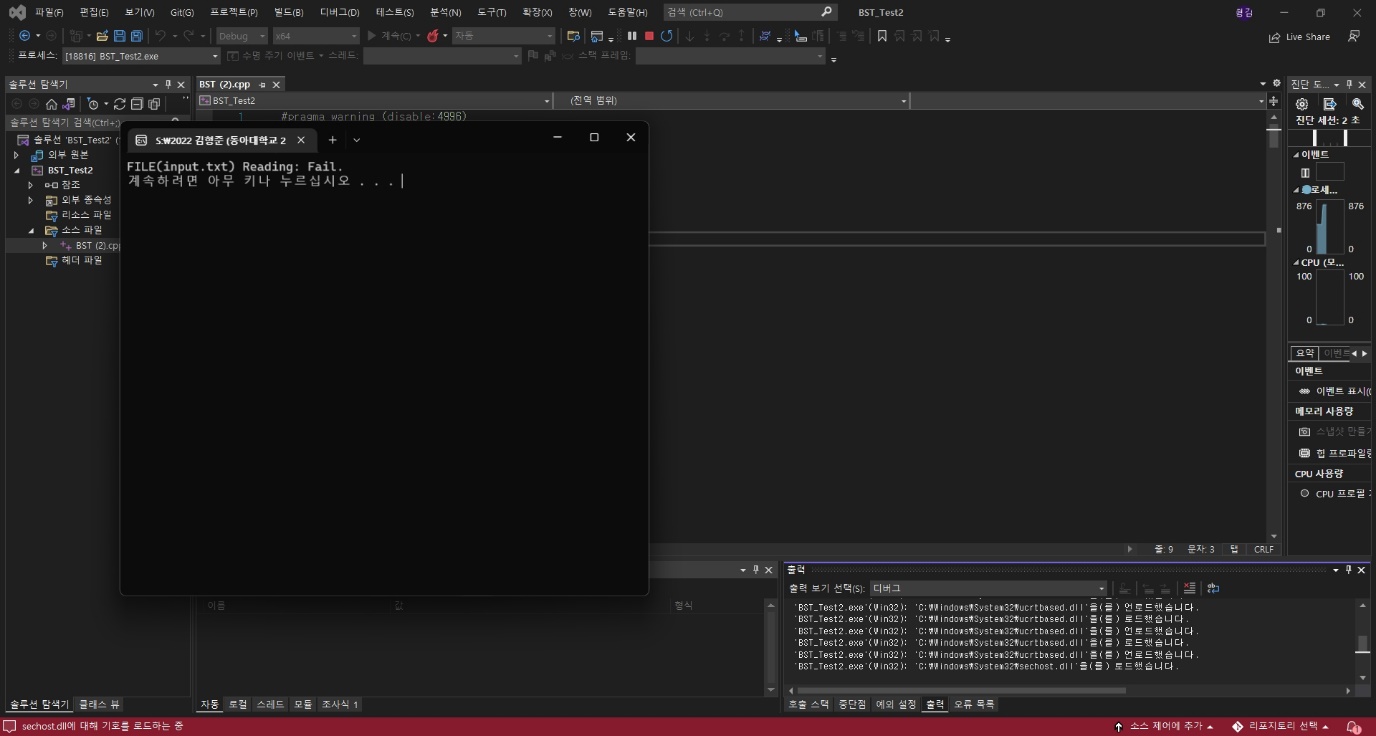
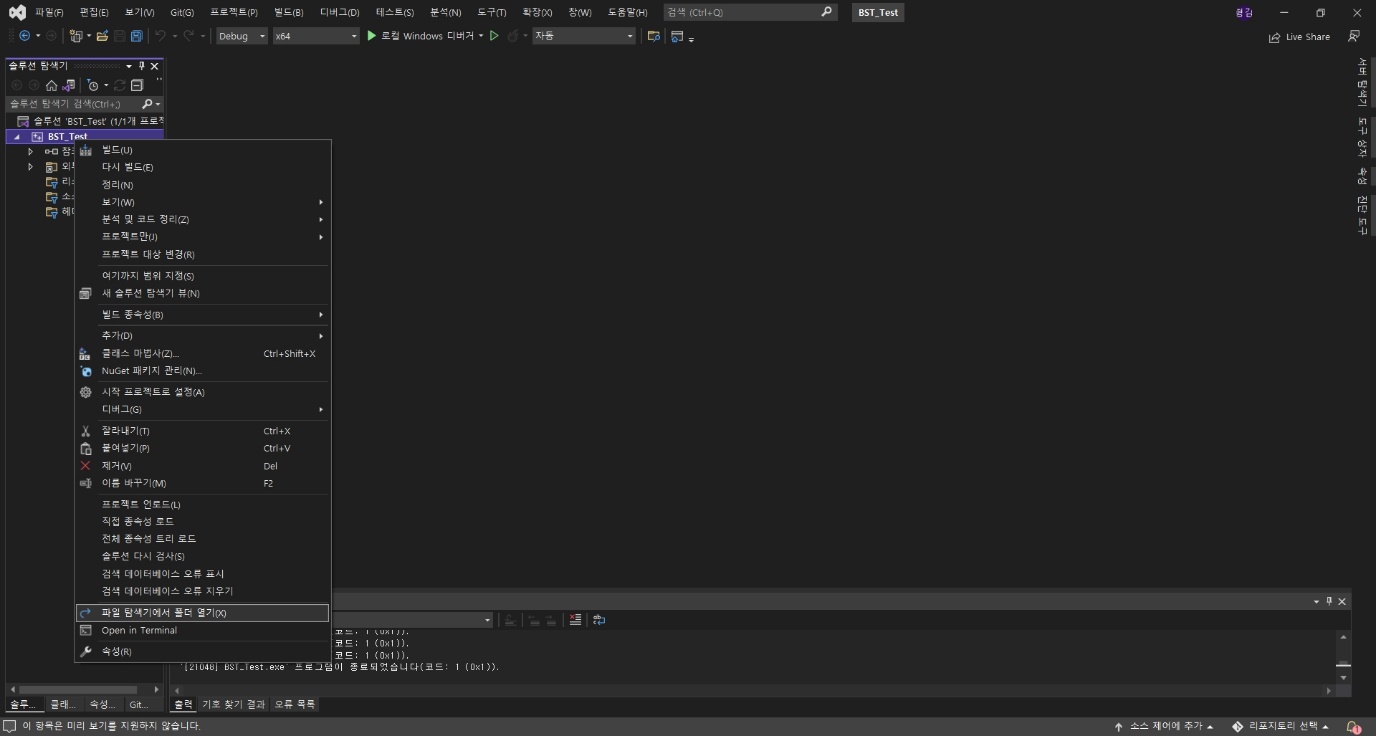
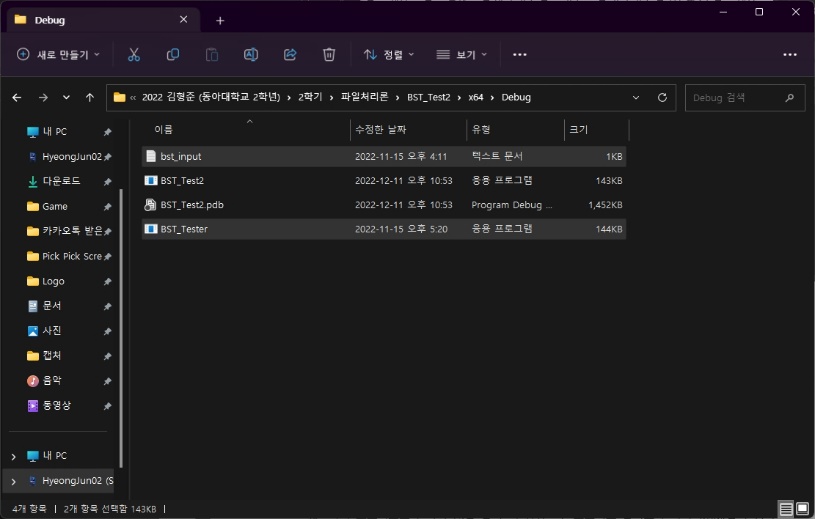
* **구현한 프로그램의 컴파일 방법**
* 사용한 프로그램: Visual Studio 2022
* 사용한 컴파일러명: Visual C++ Debugger

**이전 과제1(Replacement Selection)에서 컴파일 방법을 허술하게 적은 것 같아**

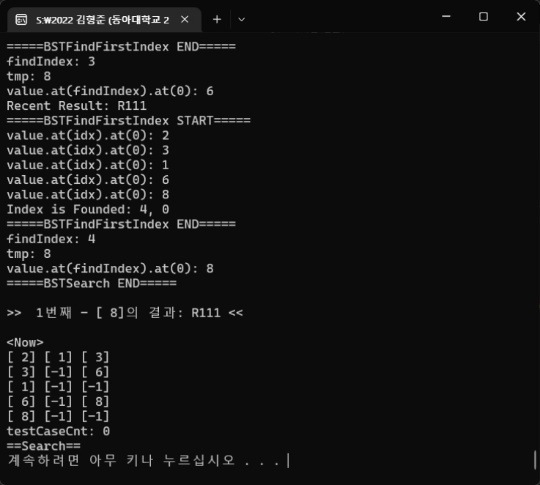
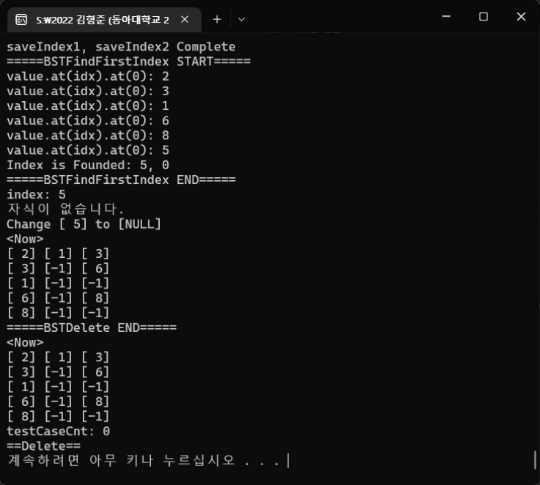
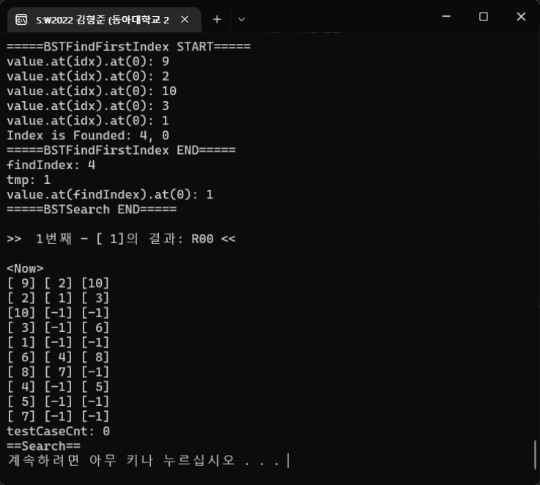
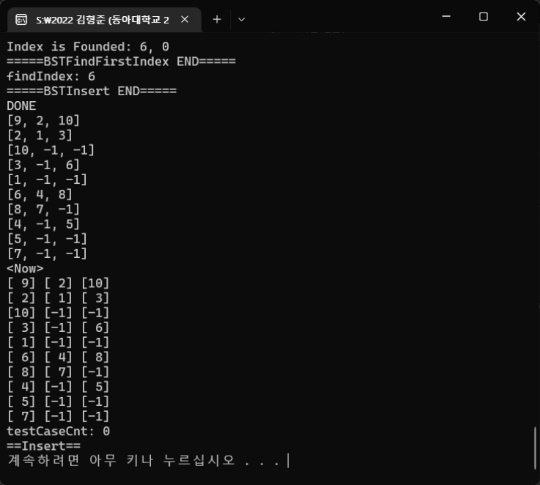
**과제2에서는 좀 더 구체적으로 자세히 적었습니다.**

혹시 과제1이 컴파일이 안된다면 아래 방법을 통해 다시 컴파일 부탁드리겠습니다.

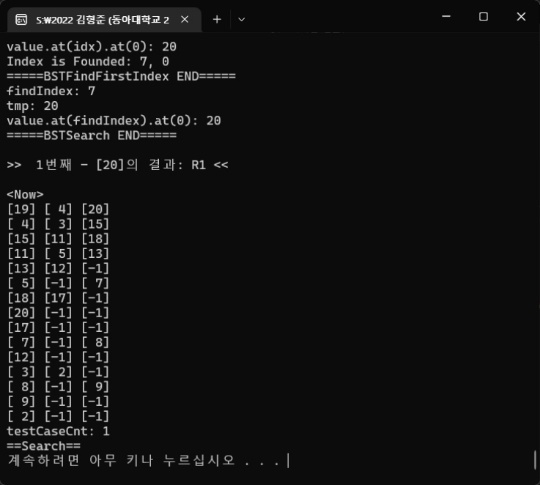
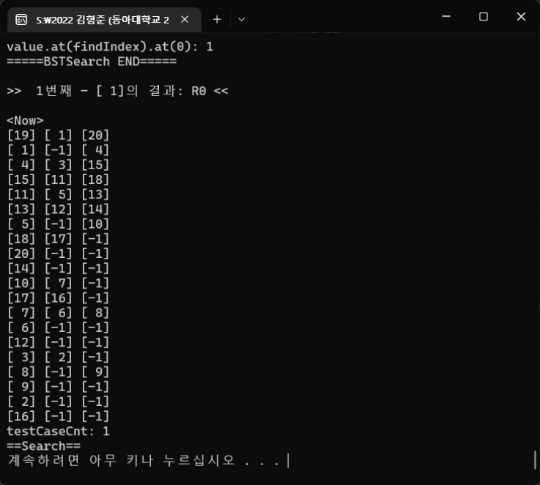
2142851 김형준입니다.

1. BST.cpp 파일을 다운 받습니다.
2. Visual Studio에서 새 프로젝트를 생성합니다.  
   
3. 빈 프로젝트 생성 후(프로젝트명 작성 후 전부 다음 클릭) 아래 사진처럼  
     
   소스 파일 부분으로 다운 받은 BST.cpp 파일을 드래그합니다.
4. 소스 파일이 열리면 F5를 눌러 디버깅을 합니다.  
   
5. 프로젝트를 우클릭하여 파일 탐색기에서 폴더 열기를 클릭합니다.  
   
6. 상위 폴더로 한 칸 올라온 뒤 x64\Debug 경로로 들어가면 작성한 프로젝트명의 .exe 파일이 생성된 것을 확인할 수 있습니다.
7. 이곳에 bst\_input.txt, BST\_Tester.exe 파일(총 2개)을 넣습니다.   
   
8. 6번에서 확인했던 프로젝트명.exe 파일을 실행시킵니다.
9. 이후 bst\_output.txt 파일이 생성되고 BST\_Tester.exe 파일을 실행시킬 시 3개의 Test Case에 대하여 RIGHT가 뜨게 됩니다.

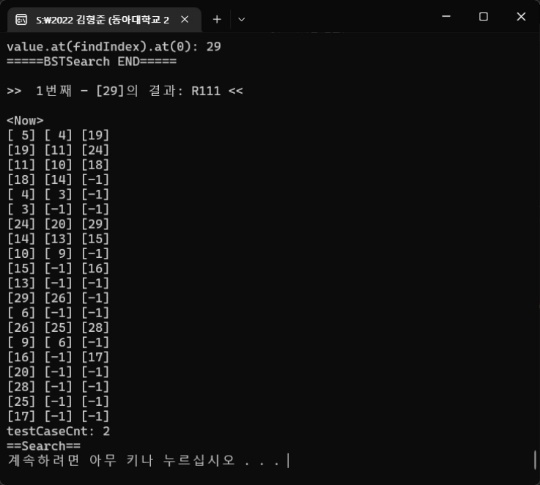
* 4가지 행동(Insert, Search, Delete, Search)을 각각 진행할 때마다 system(“pause”); 를 넣어 중간중간 계속 확인할 수 있게 하였습니다.
* 제출한 BST.cpp 파일은 system(“pause”); 를 제거하여 제출하였습니다.
* **구현한 프로그램의 실행 결과 스크린샷**
* Test Case: 1 (Insert, Search, Delete, Search)



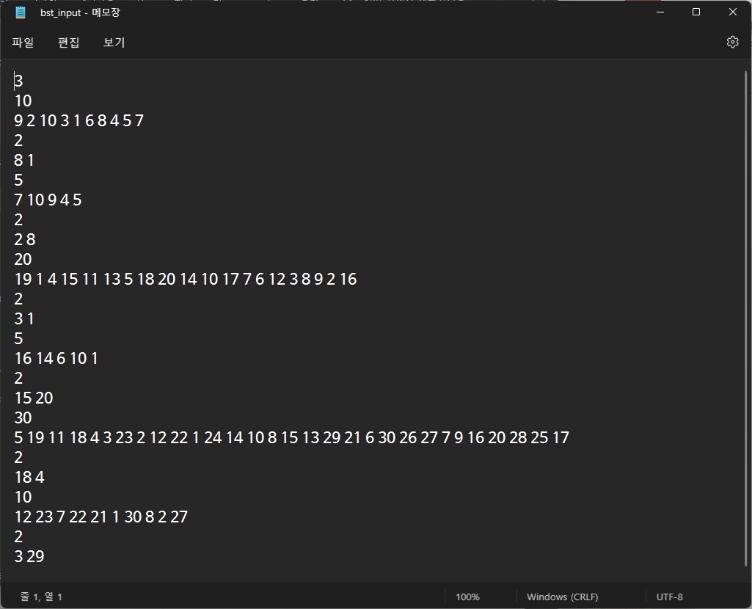
* Test Case: 2 (Insert, Search, Delete, Search)



* Test Case: 3 (Insert, Search, Delete, Search)



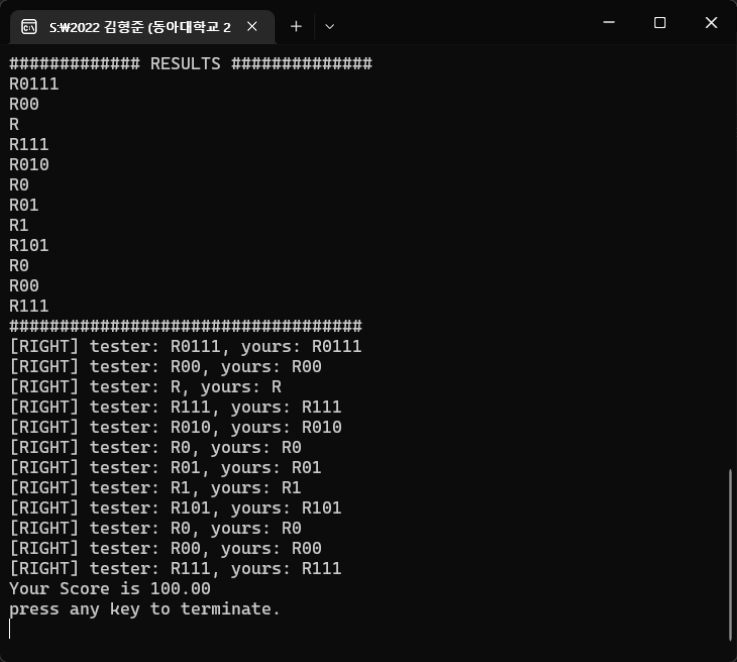
* bst\_input.txt



* bst\_output.txt



* BST\_Tester.exe



* **프로그램 구현에 있어서 어려웠던 점이 무엇이고 어떻게 해결하였는가?**

**<시작하기 전에>**

* 먼저, Insert(BSTInsert), Search(BSTSearch) 함수의 구현에서 어려운 점은 없었습니다.
* Delete(BSTDelete) 부분에 어려운 점이 많았기에 Delete 부분의 어려운 점을 기술하였습니다.
* Tree 속에 10이 있고 10의 왼쪽 자식이 4, 오른쪽 자식이 없다면 2차원 vector를 이용하여 (10, 4, -1)로 저장하였습니다.
* 즉, ((10, 4, -1), (4, 2, 7), (2, -1, -1), (7, -1, -1))과 같이 저장하였습니다.
* BSTFindFirstIndex: 저장된 vector들 사이에서 2차원 속 vector의 at(0)이 10인 index를 찾는다. (자신이 해당 값을 갖는 index를 찾아 반환한다.)
* BSTFindLastIndex: 저장된 vector들 사이에서 2차원 속 vector의 at(1) 또는 at(2)가 10인 index를 찾는다. (자식이 해당 값을 갖는 index를 찾아 반환한다.)

**<어려웠던 점과 해결>**

* 어려웠던 점 1) 자식이 2개 있을 경우 Delete

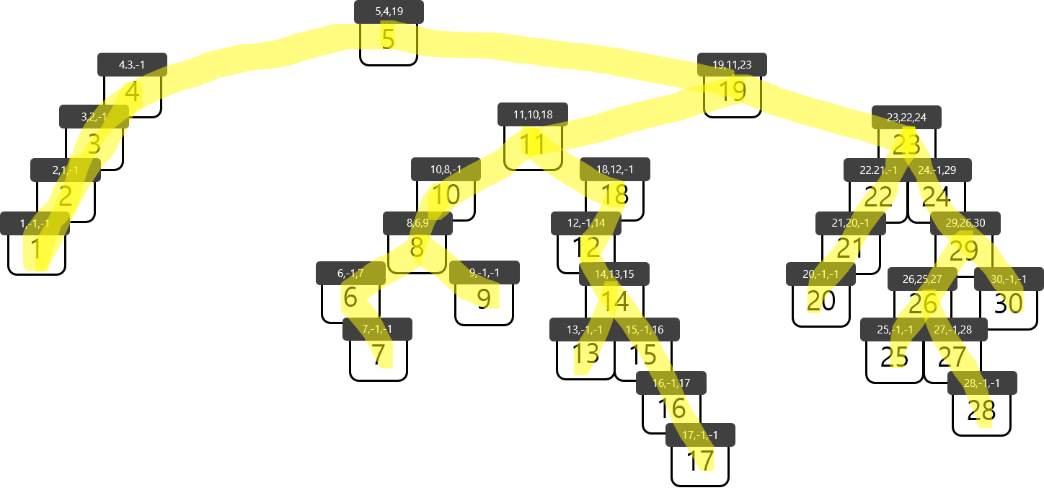
자식이 2개 있을 경우 오른 쪽 자식 중 가장 작은 값을 넣어야 합니다. 처음에는 그 가장 작은 값으로 대체한 후 그 부모의 index1 || index2 중 해당하는 값을 변경해주었습니다. 당연하게도 오류가 발생하였습니다.

그래서 콘솔창에 전부 찍어보았고, 대체하려던 값의 부모의 vector 값이 잘못되었다는 것을 알게 되었습니다. 이 문제를 해결하기 위해 신경 쓸 변수들을 먼저 정리하였습니다.

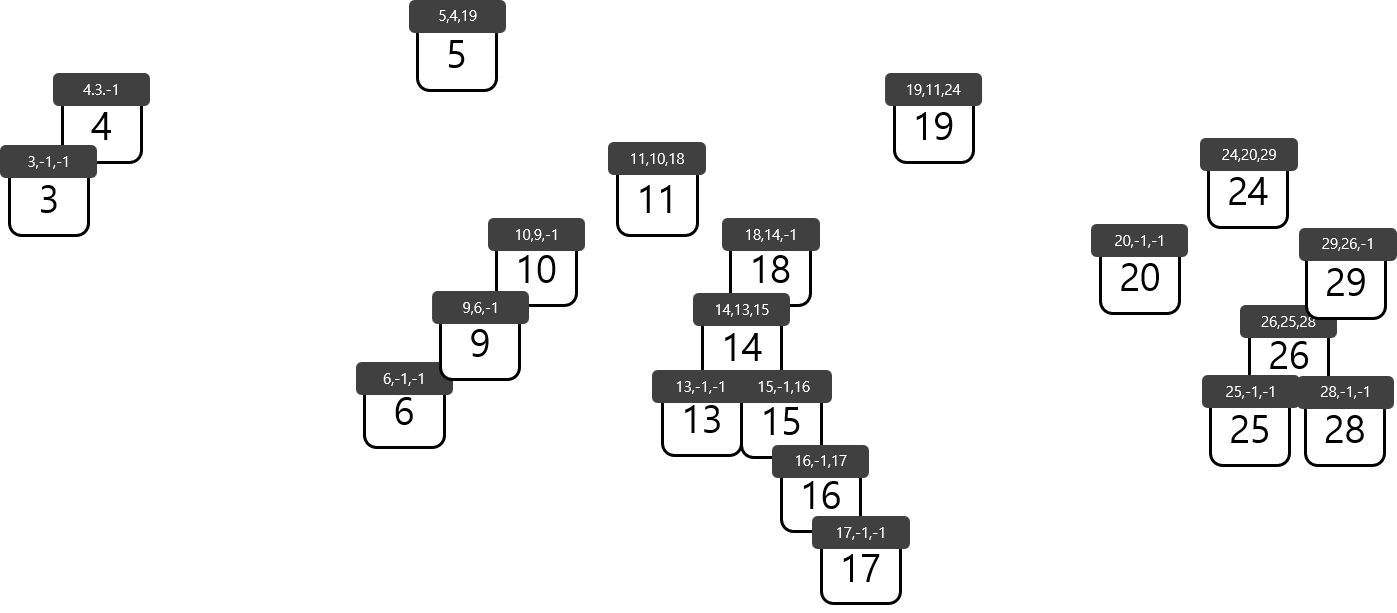
1. 삭제할 node
   1. 삭제 전 idx1, indx2 저장하기
2. 삭제할 node의 부모 node
   1. 삭제할 node 쪽 값 변경
   2. [삭제할 node 값] => [대체할 node 값]
3. 대체할 node
   1. [대체할 node의 idx1] => [삭제할 node의 idx1]
   2. [대체할 node의 idx2] => [삭제할 node의 idx2]
4. 대체할 node의 부모 node
   1. 대체할 node 쪽 값 변경
   2. [대체할 node 값] => [-1]

위에서 언급한 변수들을 신경 쓰며 값을 변경해주었더니 잘 돌아가는 것 같았습니다.

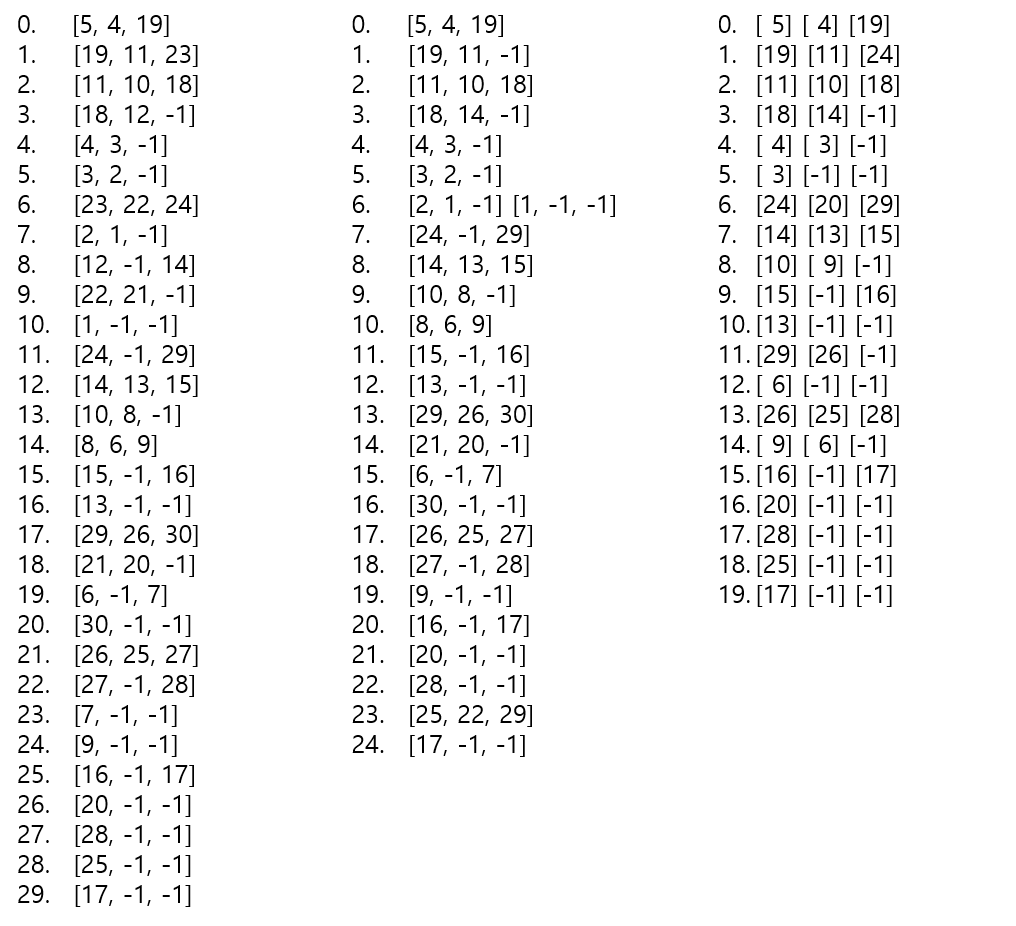
하지만 testCase3에서 무한루프에 빠지게 되었습니다. 무한루프에 빠지게 된 값을 찾는 것은 콘솔 로그를 통해 쉽게 확인할 수 있었으나 왜 문제가 생긴지 몰랐습니다. 그래서 testCase3의 모든 node들을 직접 그려보았습니다.



이 후 delete 되었을 때 그림이 어떻게 바뀌게 되는지도 그려보았습니다.



이 그림들과 더불어 콘솔에 찍었던 address(vector의 값)를 정렬하고, 순차적으로 어떻게 값이 변경되어야 하는지 적어보았습니다.



이렇게 작성하고 나니 문제점이 보이기 시작했습니다.

원래 나와야 할 값과 다른 값이 찍혀 있는 node의 vector에서 예외를 발견하였습니다. 첫번째 예외는 Root에서 Delete를 하는 경우입니다. 두번째로는 대체할 자식이 삭제할 node의 바로 아래 자식일 때입니다. 이 2가지 경우는 아래에 상세히 기술하였습니다.

* 어려웠던 점 2) vector에서 pop과 비슷한 기능을 하는 함수

처음 delete를 시작하였을 때, 삭제할 값이 생기면 그냥 (-1, -1, -1)로 설정하려고 했습니다. 하지만 이렇게 코드를 작성한다면, search를 진행할 때 문제가 생겼습니다. 그래서 특정 값을 지우는 함수를 찾아보았습니다.

특히 index의 크기를 그대로 유지하면서 지우는 함수가 아닌, python의 pop과 비슷하지만 다른 기능을 하는 함수를 찾아보았습니다. 검색해본 결과 v.pop\_back()이 있었습니다. 하지만 이 함수는 가장 마지막 값을 지우는 pop과 아예 같은 함수였습니다. 그래서 v.erase()를 사용하였습니다.

v.erase()는 v.erase(v.begin+index)를 통해 해당하는 index의 vector 값을 지우며 제가 원하던 index의 크기를 변형시키며(줄이며) 값을 제거하는 함수였습니다.

그래서 이 함수를 사용하여 지우고자 하는 vector의 index를 찾아와 v.erase(v.begin+index)로 지웠더니 원하는 결과가 나오게 되었습니다.

* 어려웠던 점 3) Root에서 Delete를 하는 경우

정말 당연한 예외이지만 간과하고 있었던 예외입니다. Root에서 Delete를 하는 경우 Root의 부모는 없기에 부모 node를 찾으려는 search를 진행할 경우 찾지 못하여 무한루프에 빠지게 됩니다.

이는 아주 간단하게 해결하였습니다. Root node의 경우 index는 당연히 0이므로 index가 0일 경우를 예외 처리하여 부모 node를 찾지 않고 지울 수 있도록 하였습니다. 그리고 문제를 해결하였습니다.

* 어려웠던 점 4) 대체할 자식이 삭제할 node의 바로 아래 자식인 경우

어떻게 보면 쉬운 예외이지만, 저에게는 생각보다 긴 시간 생각하게 만든 예외입니다. 어려웠던 점 1에서 언급했듯이 신경 써야 할 변수 4개만을 생각하며 코딩하였습니다. 하지만 이 예외에 걸리게 된다면, 서로 값을 변경하고, 지우며 vector들의 값이 엉키고 사라지게 됩니다.

만약, 대체할 자식이 삭제할 node의 바로 아래 자식인 경우 신경 써야 할 변수는 4개가 아닌 3개로 줄어들게 됩니다. 삭제할 node, 삭제할 node의 부모 node, 대체할 node 이렇게 3개가 됩니다. 대체할 node의 부모 node는 결국 삭제할 node이기 때문에 신경 쓸 필요가 없습니다.

가장 처음 생각했던 문제점은 ‘그럼 대체할 자식이 삭제할 node의 바로 아래 자식인 경우 && 삭제할 node가 Root node일 경우’ 였습니다. 하지만 이것은 일단 지금의 문제를 해결한 후 그 코드를 Root node를 제거할 경우의 코드에 적용하니 쉽게 풀렸습니다.

본론으로 돌아와 이 예외를 처리하기 위해 떠올린 방법은 카운트를 세는 것이었습니다. 만약 search의 진행 카운트가 1일 때 변경이 이루어진다면 이 예외에 부합하는 것이기 때문입니다. 하지만 이 방법은 사용하지 않았습니다. 먼저 오른쪽 자식 중 가장 작은 값을 rightChildMin이라는 변수를 만들어 대입하여 놓았습니다. 그 후 조건문을 사용해 만약 첫 대체할 값의 오른쪽 자식이 rightChildMin일 경우를 예외 처리하였습니다.

이렇게 진행한다면 당연히 성공이지만, 코딩 중 잔 실수를 많이 하였고, 아예 delete를 싹 갈아엎기까지 하는 저에겐 굉장히 큰 어려움이었습니다.

그래도 다행인 건 코드를 중간중간 계속 NOTION에 저장해 두었기 때문에 저장된 코드를 보며 갈아엎을 때 도움이 되었습니다. 이를 통해 정말 git 사용의 장점을 제대로 느끼게 되었습니다.