Lab on BST

목차

소개

본 Lab 은 서로 연관된 3 개의 Lab 으로 구성되어 있습니다. 사용자가 이진 탐색 트리를 상호적으로 테스트할 수 있도록 tree.cpp 의 이진 트리(BT), 이진 탐색 트리(BST), 그리고 AVL 트리를 다루는 함수를 완성하세요. 다음 파일들이 제공됩니다.

- **treeDriver.cpp**: tests BT/BST/AVL 트리 구현을 상호적으로 테스트합니다. 수정 금지.
- **tree.cpp** : BST/AVL 트리 구현을 위한 뼈대 코드.
- treenode.h : 기본 트리 구조와 key 자료형 정의
- tree.h : BT, BST, AVL 트리에 대한 ADTs 정의. 수정 금지.
- treeprint.cpp : 콘솔에 트리 그림 출력
- treex.exe : 참고용 실행 답안

자신이 작성한 프로그램이 제공된 treex.exe 와 같이 작동해야 합니다. 자신의 tree.cpp 가 tree.h 와 treeDriver.cpp 와 호환되어야 합니다. 따라서, tree.h 와 tree.cpp 파일의 함수 시그니처와 반환 유형은 수정하지 않아야 합니다.

treeDriver.cpp 의 **build_tree_by_args()**는 명령 인수를 가져와서 위에 나온 것과 같이 **BT, BST** 또는 **AVL** 트리를 빌드합니다. 트리에 대한 인수가 제공되지 않으면 기본적으로 BT 로 시작합니다.

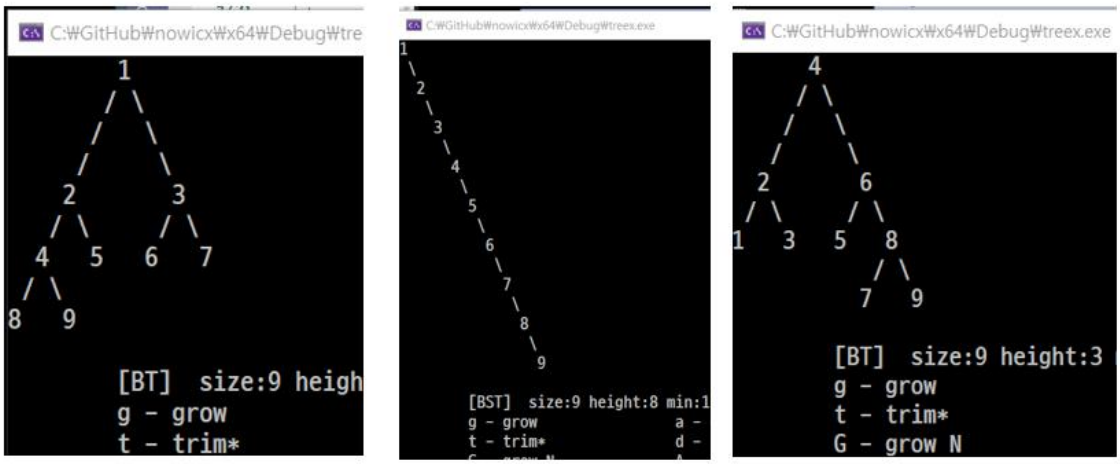
```
PS C:\GitHub\nowicx\psets\pset10-12tree> ./treex -b 1 2 3 4
    1
   / \
  2   3
 /
4

        Menu [BT]  size:4 height:2 min:1 max:4
        g - grow                a - grow a leaf    [BT]
        t - trim*               d - trim a leaf    [BT]
        G - grow N              A - grow by Level  [BT]
        T - trim N              f - find node      [BT]
        o - BST or AVL?         p - find path&back [BT]
        r - rebalance tree**    l - traverse       [BT]
        L - LCA*                B - LCA*           [BT]
        m - menu [BST]/[AVL]**  C - convert BT to BST*
        c - clear               s - show mode:[tree]
        Command(q to quit):
```

다음과 같은 3 가지 옵션을 사용하면 트리 프로그램 실행 시 자동으로 3 개의 다른 트리를 만들 수 있습니다.

./treex -b 1 2 3 4 5 6 7 8 9

./treex -s 1 2 3 4 5 6 7 8 9

./treex -a 1 2 3 4 5 6 7 8 9



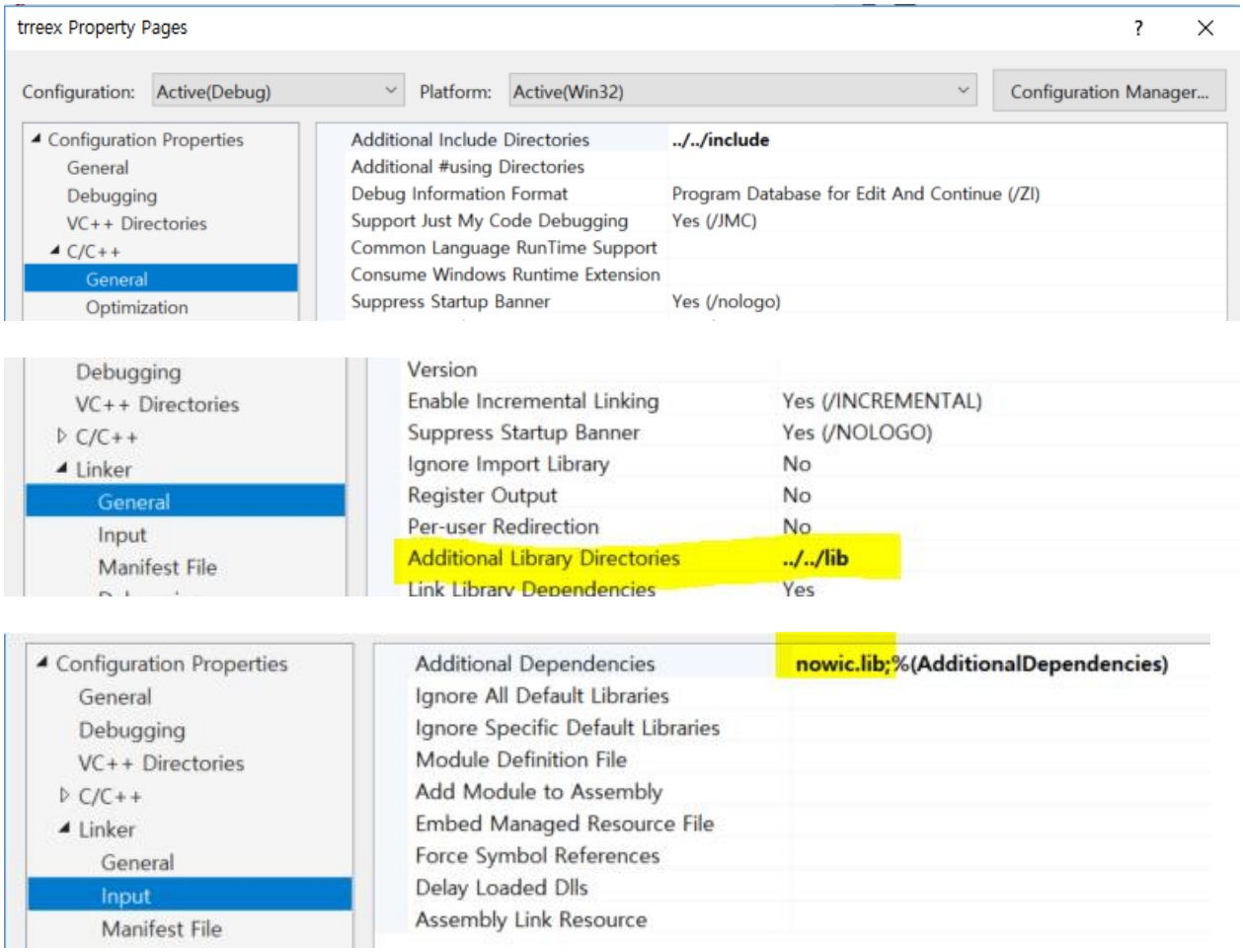
JumpStart

Jump-start 의 경우, 먼저 tree 라는 이름의 프로젝트를 생성합니다. 평소와 같이 다음 작업을 수행합니다.

- Add ~/include at
  - Project Property → C/C++ → General → Additional Include Directories

- Add ~/lib at
  - Project Property → Linker → General → Additional Library Directories
- Add nowic.lib at
  - Project Property → Linker → Input → Additional Dependencies
- Add /D "DEBUG" at
  - Project Property → C/C++ → Command Line

예시:

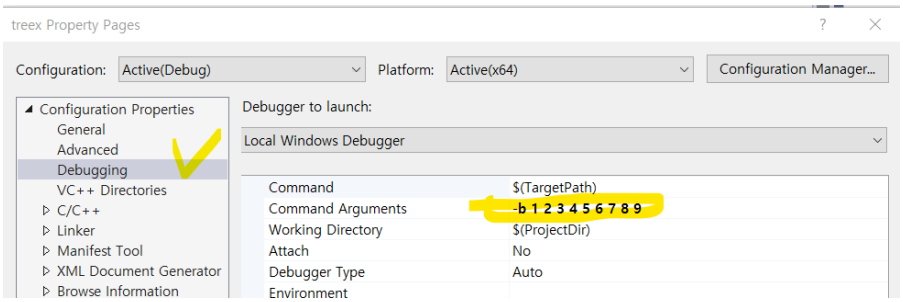


Add ~.h files under 프로젝트 'Header Files' 아래에 ~.h 파일을 추가하고 프로젝트 'Source Files' 아래에 ~.cpp 파일을 추가하면 프로젝트를 빌드할 수 있습니다.

Step 0: 디버깅을 위한 트리를 쉽게 만드는 방법

초반에는 디버깅을 목적으로 동일한 트리를 매번 생성하는 경우가 종종 있습니다. 트리의 초기 key 값을 다음 경로를 통해 지정할 수 있습니다.

Project Properties → Debugging → Command Argument



Step 2.1: 이진 탐색 트리 작업:

BST 에 대한 추가적인 함수가 몇 가지 있습니다. 아래 함수들을 확인해 본 후 작동하지 않는다면 제대로 작동하도록 코드를 작성하세요.

- pred(), succ() - 트리의 predecessor 과 successor 노드를 반환합니다.
- isBST() - 트리가 이진 탐색 트리인 경우 true 를 반환하고, 그렇지 않은 경우 false 를 반환합니다.
- minimum(), maximum()
- contains()
- find()

Step 2.2: grow() & trim()

이 단계에서는 기본 함수 grow()와 trim()을 구현합니다. 강의 영상에서 grow()를 다뤘으니 trim() 함수를 한 번 구현해 봅시다. 이진 탐색 트리의 제거(trim) 작업은 추가(grow)와 탐색 작업보다 복잡합니다. 기본적으로, 이 작업은 두 단계로 나눌 수 있습니다.

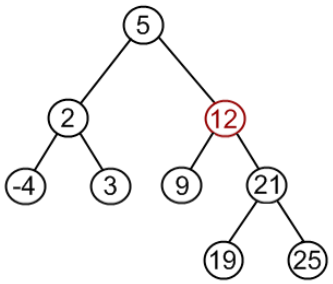
- 제거할 노드를 재귀적으로 탐색합니다. 예를 들어:
  ```
  if (key < node->key)
    node->left = trim(node->left, key);
  ```
- 결과적으로 이 **node→left** 의 값은 **trim(node→left, key)**이 완료되면 그 반환값으로 설정됩니다.
- 노드를 발견하면, 제거(trim) 알고리즘을 **재귀적으로** 실행합니다.

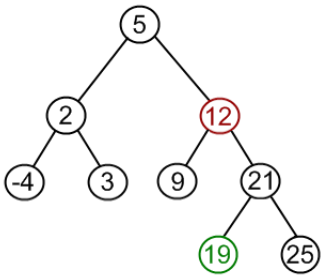
노드를 제거할 때 3 가지 경우를 고려해야 합니다. 노드를 제거하면 반드시 nullptr(leaf 노드인 경우) 또는 교체된 노드를 반환해야 합니다. 반환된 포인터는 결과적으로 **node→left = trim(node→left, key);**와 같이 부모 노드의 **key→left** 혹은 **key→right** 로 설정됩니다.

- 제거할 노드가 leaf 인(혹은 자식 노드가 없는) 경우:
  - 단순히 트리에서 노드를 제거합니다. 알고리즘은 (부모 노드의) 해당 링크를 **nullptr** 로 설정하고 노드를 제거합니다. **따라서 nullptr 을 반환합니다.**
- 제거할 노드의 자식 노드가 1 개만 있는 경우:
  - 노드를 임시(temp) 노드에 복사합니다.
  - 노드를 자식 노드로 설정합니다.
  - 임시 노드(혹은 제거할 기존 노드)를 해제합니다.

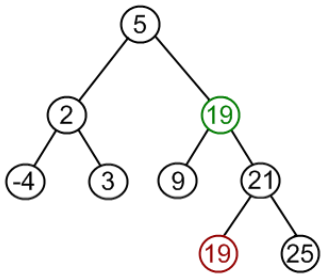
  - 재귀 trim()은 이 자식 노드를 (하위 노드와 함께) 제거된 노드의 부모 노드에 직접 링크합니다. **이 작업은 해당 자식(노드)를 반환하여 수행합니다.**
- 제거할 노드의 자식 노드가 2 개인 경우:
  - 두 하위 노드의 높이를 구해서 predecessor 와 successor 중 무엇을 사용할지 결정합니다. (자세한 내용은 이 섹션 끝에 있는 "노트"를 확인하세요.)
  - **successor 또는 predecessor** 의 key 값을 노드에 복사합니다.
  - successor 를 선택한 경우 **trim("제거할 노드"의 오른쪽 자식, succ()의 key)**를 호출합니다. predecessor 를 선택한 경우 **trim("제거할 노드"의 왼쪽 자식, pred()의 key)**를 호출합니다.

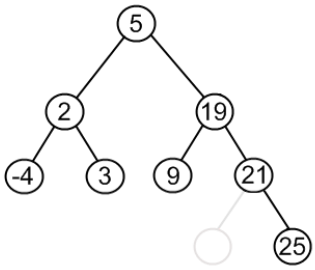
예시: BST 에서 12 를 제거합니다.





1. (오른쪽 하위 트리가 왼쪽 하위 트리보다 높으므로) 제거할 노드의 successor 를 찾습니다. 이 예시에서는 19 입니다.
2. 12 를 19 로 교체합니다. 노드는 교체하지 않고 값만 교체한다는 점을 주의하세요.
   이제 동일한 값을 가진 두 개의 노드가 있습니다.





3. **trim()**을 재귀적으로 호출하여 왼쪽 하위 트리에서 기존 노드 19 를 제거합니다. 노드 19 를 제거하기 위한 입력 매개 변수는 무엇일까요? 당연히 key 는 successor 인 19 입니다.
   그렇다면 전달할 노드는 무엇일까요? node→right 입니다.
   이 단계는 또 다른 trim()을 호출하여 단순히 수행할 수 있습니다: node→right = trim(node→right, 19);

노트: Successor 또는 Predecessor 중 어떤 것을 사용해야 할까?

노드의 successor 을 이용하여 제거 옵션을 성공적으로 구현하면 이제 이 옵션을 개선할 준비가 된 것입니다. successor 를 이용하여 노드를 지속적으로 제거할 경우, 노드를 오른쪽 하위 트리에서 제거하므로 트리가 편향될 수 있습니다.

trim() 함수를 구현할 때 가능하면 트리의 밸런스를 맞추기 위해 왼쪽과 오른쪽 하위 트리의 높이를 확인하고 predecessor 와 successor 중 어떤 것을 사용할지 결정하도록 구현해야 합니다.

Step 2.3: growN() & trimN()

사용자가 지정한 수의 노드를 삽입(grow) 또는 제거(trim)합니다. 참조용으로 제공된 growN() 함수는 사용자가 지정한 수(N)의 노드를 트리에 삽입합니다. 만약 트리가 비어있는 경우, 추가할 key 값의 범위는 0 부터 N-1 까지입니다. 트리에 기존 노드가 있는 경우, 추가할 key 값의 범위는 max + 1 부터 max + 1 + N 까지입니다 (max 는 트리에 있는 key 의 최댓값입니다).

사용자가 지정한 수(N)의 노드를 트리에서 제거하는 **trimN() 함수를 구현하세요.** 제거할 노드는 트리에서 무작위로 선택합니다. key 값이 꼭 연속적인 수라는 보장은 없으므로 트리에서 key 값을 가져와야 합니다. 예를 들어, 트리의 키 값은 5, 6, 10, 20, 3, 30 이 될 수 있습니다.

사용자가 지정한 제거할 노드의 수(N)가 (N 이 아닌) **트리의 크기**보다 작다면, N 개의 노드를 제거합니다. N 이 트리의 크기보다 크다면, N 을 트리의 크기와 같도록 설정합니다. 어떤 경우든 모든 노드를 하나씩, 무작위로 제거해야 합니다. (본인만의 구현 방식도 있겠지만) 코드 구현 시 다음 내용을 참고해도 좋습니다.

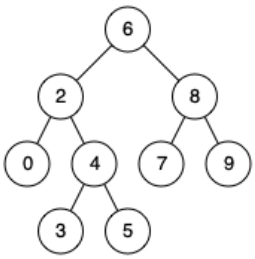
- Step 1:
    1. 트리에 있는 모든 key 의 목록을 가져옵니다.
    2. **inorder()**를 호출하여 벡터를 트리의 key 로 채웁니다.
       (C++의 벡터 객체를 사용하여 **모든 key 를 저장합니다**. 벡터 크기는 필요에 따라 증가합니다.)
    3. size()를 사용하여 **트리의 크기**를 구합니다.
    4. inorder()를 호출하여 구한 벡터의 크기와 트리의 크기를 비교하여 확인합니다.
- Step 2:
    1. 배열의 키를 사용하여 순서대로 **trim()**을 **N 번** 호출합니다. for 문에서 **trim()이 트리의 새 루트를 반환할 수 있다**는 점을 기억하세요.

힌트: 트리의 모든 key 를 가져오는 방법

벡터에 저장된 트리의 key 를 반환하는 트리 순회 함수를 사용하세요. tree.cpp 에 위치한 inorder() 함수를 살펴보세요.

Step 2.4: LCA for BST

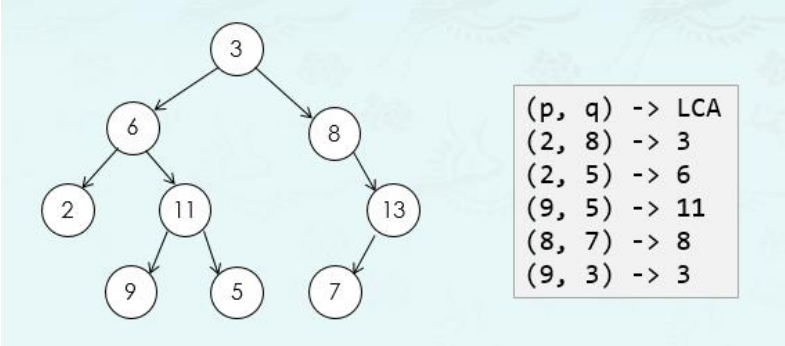
가장 낮은 공통 상위 노드(LCA)는 두 노드 p 와 q 사이에서 p 와 q 를 모두 하위 노드로 가진 T 의 가장 낮은 노드로 정의합니다 (**노드 자신을 자기 자신의 하위 노드**로 볼 수 있습니다).



아래에 나온 이진 트리를 예로 들어봅시다. 노드 2 와 노드 8 의 LCA 는 6 입니다. LCA 정의에 따르면 노드 자신을 자기 자신의 하위 노드로 볼 수 있으므로 노드 2 와 노드 4 의 LCA 는 2 입니다. 이 예시에서 다음 내용을 확인할 수 있습니다.

- 모든 노드의 값은 고유합니다.
- p 와 q 는 서로 다르며 둘 다 BST 에 존재합니다.

Intuition: 두 노드 p 와 q 의 가장 낮은 공통 상위 노드는 두 노드 모두 공통으로 가지고 있는 마지막 상위 노드입니다. 여기서 '마지막'은 노드의 깊이로 정의합니다. 아래 도표가 '가장 낮은'의 의미를 이해하는 데에 도움이 될 것입니다.



노트: p 와 q 중 하나는 LCA 노드의 왼쪽 하위 트리에 위치하고, 다른 하나는 오른쪽 하위 트리에 위치합니다.

알고리즘:

1. 루트 노드부터 트리 순회를 시작합니다.
2. 노드 p 와 q 모두 오른쪽 하위 트리에 위치한 경우, 오른쪽 하위 트리에서 step1 부터 시작하여 탐색을 계속합니다.
3. 노드 p 와 q 모두 왼쪽 하위 트리에 위치한 경, 왼쪽 하위 트리에서 step1 부터 시작하여 탐색을 계속합니다.
4. step2 와 step3 가 모두 참이 아니라면, 이는 노드 p 와 q 의 하위 트리와 공통되는 노드를 찾았다는 의미입니다. 따라서 해당 노드(공통 노드)를 LCA 로 반환합니다.

시간 복잡도: $O(N)$, N 은 BST 의 노드 개수입니다. 최악의 경우, BST 의 모든 노드를 방문할 수도 있습니다.

공간 복잡도: $O(N)$. 편향된 BST 의 높이가 N 이 될 수 있으므로 재귀 스택에 의해 사용되는 공간의 최대 크기는 N 입니다.

제출 파일 목록

- **Lab10** for BST - tree.cpp - 2.1 ~ 2.3

참고 문헌

1. Recursion :
2. Recursion: