본 PSet 은 저의 강의 경험과 학생들의 의견 및 Stanford CS106 과 Harvard CS50 같은 강의에서 수집된 자료를 토대로 작성되었습니다. 본 PSet 에 문제가 있거나, 질문 혹은 의견이 있다면, 언제든지 알려 주시면 감사하겠습니다. 강의 개선에 많은 도움이 되겠습니다. idebtor@gmail.com

PSet on BT, BST and AVL Tree

목차

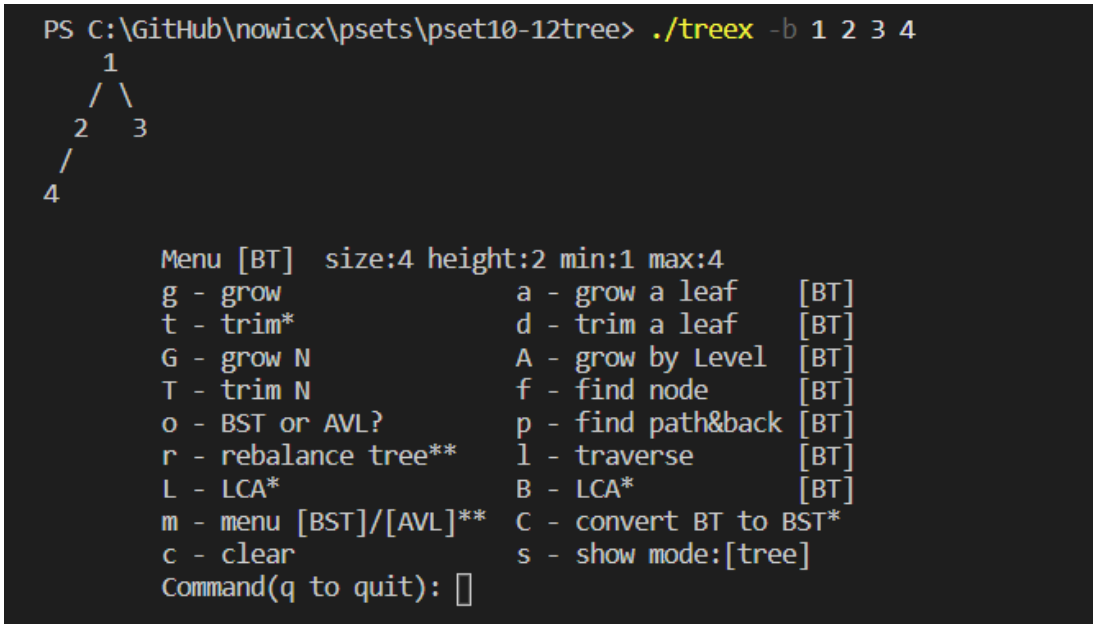
소개

본 Pset 은 서로 연관된 3 개의 Pset 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.



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

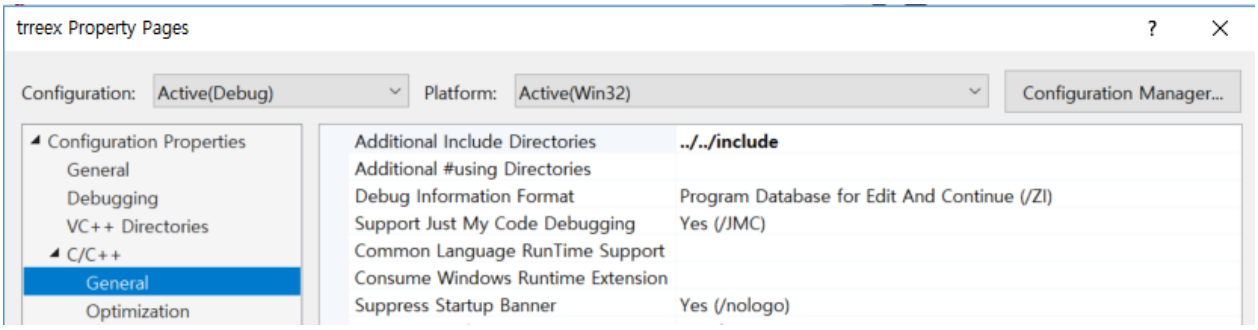
```
./treex -b  1 2 3 4 5 6 7 8 9

./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:



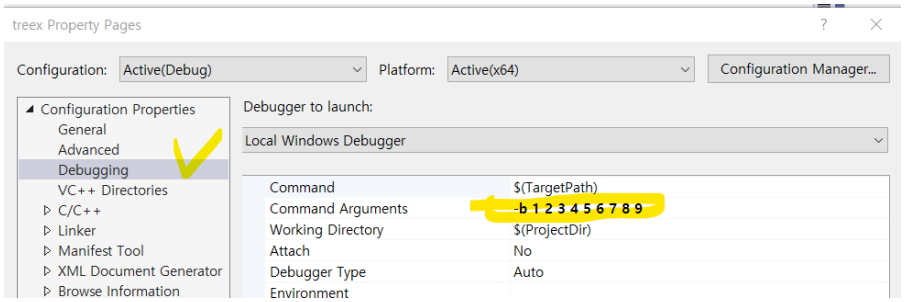
idebtor@gmail.com

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 1.1: BT 기본 작업

tree.cpp 의 일부 함수들은 이미 구현되어 있습니다. 아래 명시된 함수들을 구현하세요. 필요한 경우 (특히 재귀의 경우) 추가적인 도우미 함수를 작성하여 사용하세요. 이상적으로, 이 Pset 에서 모든 코드는 tree.cpp 에 작성합니다.

[BT]가 있는 메뉴 항목은 대개 3 가지 유형의 트리에 모두 적용됩니다 (다소 느리게 작동할 수 있습니다). 예를 들어, find(), findPath(), findPathBack(), maximum(), 그리고 minimum() 등등이 있습니다. 다음 함수들을 테스트하여 올바르게 작동하는지 확인하고, 개발 환경에 익숙해지세요. 한국어로 작성된 재귀 트리에 대한 참고 문헌을 확인해 보세요.

- `clear()`
- `size()`
- `height()`
- `minimumBT(), maximumBT();`
- `containsBT(), findBT()`
- `inorder(), preorder(), postorder()`

Step 1.2: levelorder() - iteration version

이 순회(traversal)는 하위 레벨로 내려가기 전에 해당 레벨에 존재하는 모든 노드를 방문합니다. 이 탐색은 탐색 트리가 다음 깊이로 이동하기 전에 각 깊이에서 최대한으로 넓어지므로 너비 우선 탐색(BFS)이라고 부릅니다. 이 작업에는 지정된 깊이에서 노드의 최대 개수에 비례하는 공간이 필요합니다. 이 크기는 총 노드의 개수 / 2 만큼 커질 수 있습니다.

알고리즘 (Iteration):

- 빈 queue 를 생성하고 루트 노드를 push 합니다.
- Queue 가 비워질 때까지 다음을 반복합니다.
  - Queue 에서 노드를 pop 하고 출력/저장하세요.
  - Pop 한 노드의 왼쪽 자식 노드가 null 이 아니라면 push 하세요.
  - Pop 한 노드의 오른쪽 자식 노드가 null 이 아니라면 push 하세요.

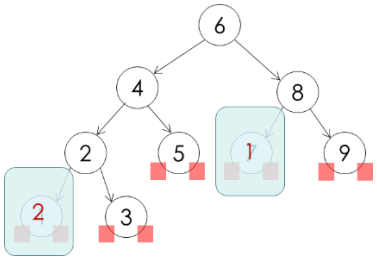
```
// level order traversal of a given binary tree using iteration.
void levelorder(tree root, vector<int>& vec) {
  Visit the root.
  if it is not null, push it to queue.
  while queue is not empty
    queue.front() - get the node from the queue
    visit the node (save the key in vec).
    if its left child is not null, push it to queue.
   if its right child is not null, push it to queue.
   queue.pop() - remove the node in the queue.
  }
}
```

레벨 순회를 이해하고 구현하면 그다음 함수인 growBT()가 levelorder() 함수의 또 다른 변형임을 알 수 있습니다.

Step 1.3: growBT() - 레벨 순서대로 노드 추가하기

이 함수는 key 를 가진 노드를 삽입하고 이진 트리의 루트를 반환합니다.

아래에 나온 트리의 경우, 레벨 순서대로 노드가 추가될 때 처음 추가될 자리는 빈 노드인 8 의 왼쪽 자식 노드입니다. 그다음으로 추가될 자리는 빈 노드인 2 의 왼쪽 자식 노드입니다.



핵심은 queue 를 이용하여 지정된 트리를 반복적으로 레벨 순회하는 것입니다. 필요하다면 다음 알고리즘을 참고하세요.

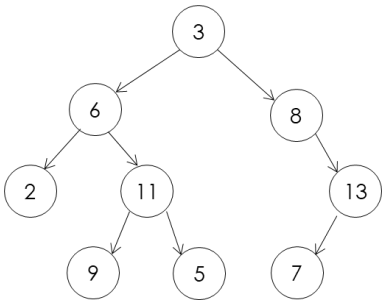
```
First, push the root to the queue.
Then, while the queue is not empty,
    Get the front() node on the queue
    If the left child of the node is empty,
       make new key as left child of the node. - break and return;
    else
       add it to queue to process later since it is not nullptr.
    If the right child is empty,
       make new key as right child of the node. - break and return;
    else
       add it to queue to process later since it is not nullptr.
    Make sure that you pop the queue finished.
    Do this until you find a node whose either left or right is empty.
```

코드의 도입부는 아래와 같이 작성할 수 있습니다.

```
tree growBT(tree root, int key) {
  if (root == nullptr)
    return new TreeNode(key);
  queue<tree> q;
  q.push(root);
  while (!q.empty()) {
    // your code here
  }
  return root; // returns the root node
}
```

Step 1.4: findPath() & findPathBack()

findPath(): 고유 키를 가진 이진 트리가 주어졌을 때, 루트부터 지정된 노드 x 로 가는 경로를 반환합니다. 예를 들어, 노드 2 로 가는 경로는 [3, 6, 2], 노드 9 → [3, 6, 11, 9], 노드 13 → [3, 8, 13]입니다.



Intuition:
현재 노드를 벡터(path)에 push 합니다. 현재 노드가 x 라면 true 를 반환합니다. 트리의 왼쪽과 오른쪽을 재귀적으로 반복해 내려가면서 x 를 찾습니다. x 를 찾으면 true 를 반환하고, 찾지 못하면 현재 노드를 제거합니다. 이 알고리즘은 preorder()의 개념에서 유래되었습니다.

알고리즘:

- **루트 = nullptr** 이면 false 를 반환합니다. [base case]
- 루트의 키를 **벡터**에 push 합니다.
- **루트의 키 = x** 이면 true 를 반환합니다.
- 루트의 왼쪽 또는 오른쪽 하위 트리에서 x 를 재귀적으로 찾습니다.

- 루트의 왼쪽 또는 오른쪽 하위 트리에 **x** 가 존재하면 true 를 반환합니다.
  x 가 존재하지 않으면 루트의 키를 **벡터**에서 제거하고 false 를 반환합니다.

findPathBack(): 유사한 알고리즘을 이용하여 루트로 가는 경로를 찾을 수 있습니다.

Intuition:
현재 노드가 x 이거나 트리의 왼쪽 또는 오른쪽 하위 트리를 재귀적으로 탐색하다가 x 를 찾으면, 현재 노드를 **벡터**에 push 합니다. x 를 찾지 못하면 false 를 반환합니다. 이 알고리즘은 postorder()의 개념에서 유래되었습니다. 노드 x 를 찾은 뒤 루트를 역추적합니다.

알고리즘:

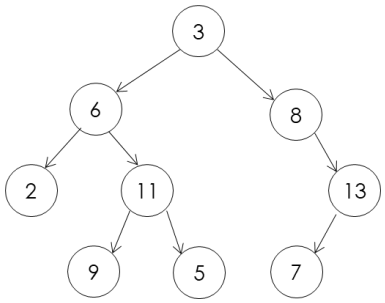
- **루트 = nullptr** 이면 false 를 반환합니다. [base case]
- **루트의 키 = x** 이거나 (재귀적으로 탐색하는 동안) **x** 가 루트의 왼쪽 또는 오른쪽 하위 트리에 존재하면,
  - 루트의 키를 **벡터**에 push 합니다. (이때 재귀적 역추적을 수행합니다)
  - true 를 반환합니다.
- 존재하지 않으면
  - false 를 반환합니다.

노트: 두 함수를 독립적으로 코딩해야 합니다. 한 함수가 다른 함수를 호출하여 reverse 하지 않아야 합니다.

Step 1.5: LCA in BT

이 단계에서는 반복과 재귀 알고리즘을 사용하여 주어진 이진 트리에서 지정된 두 노드의 가장 낮은 공통 상위 노드(LCA)를 찾는 함수 LCA_BT 를 구현합니다.

LCA 는 두 노드 p 와 q 사이에서 p 와 q 를 하위 노드로 가진 T 에서 가장 낮은 노드로 정의합니다 (노드 자기 자신 역시 자신의 하위 노드로 인정합니다). 주어진 두 노드 p 와 q 는 서로 다르고 두 값 모두 이진 트리에 존재합니다.



예를 들어, 두 노드 (2, 8)의 가장 낮은 공통 상위 노드는 3 입니다.
마찬가지로, LCA(2, 5) = 6, LCA(9, 5) = 11, LCA(8, 7) = 8, LCA(9, 3) = 3 입니다.

Intuition (반복): 주먹구구식의 접근법은 트리를 순회하며 노드 p 와 q 의 경로를 구하는 것입니다. 경로를 비교하여 경로에서 마지막으로 일치하는 노드를 반환합니다.

알고리즘 (반복):

- 루트에서 p 까지 가는 경로를 찾고 벡터에 저장합니다.
- 루트에서 q 까지 가는 경로를 찾고 또 다른 벡터에 저장합니다.

- 벡터의 값이 동일할 때까지 두 경로를 모두 순회합니다. 불일치한 값이 나오기 직전의 공통 요소를 반환합니다.

예를 들어, LCA(2, 5)를 찾으려면 findPath() 함수를 사용하여 p 와 q 의 경로를 구합니다. 이 예시에 대한 p 와 q 의 경로는 아래와 같습니다.

- Path to 2: 3 6 2
- Path to 5: 3 6 11 5

따라서 두 경로가 가지고 있는 동일한 시퀀스의 마지막 요소가 가장 낮은 공통 상위 노드가 됩니다. 위 예시에서 3 과 6 이 공통 조상이지만, 6 이 두 노드 (2, 5)에 가장 가까우므로 가장 낮은 공통 상위 노드는 6 이 됩니다.

재귀 알고리즘은 아래와 같습니다.

Intuition (재귀): 깊이 우선 방식으로 트리를 횡단합니다. 노드 p 또는 q 중 하나를 발견하면 해당 노드를 반환합니다. LCA 는 두 하위 트리 재귀(subtree recursion)가 반환하는 NULL 이 아닌 노드입니다. 이는 p 또는 q 자신이 될 수도 있고, 하위 트리 재귀에서 반환하는 특정 노드가 될 수도 있습니다.

알고리즘 (재귀):

- 루트 노드에서 트리 횡단을 시작합니다.
- 현재 노드가 nullptr 이면 nullptr 을 반환합니다. [base case]
- 현재 노드가 p 또는 q 자신이면, 해당 노드를 반환합니다. [base case]
- [recursive case]
  - 왼쪽과 오른쪽을 재귀적으로 탐색합니다.
  - 왼쪽 혹은 오른쪽 하위 트리가 NULL 이 아닌 노드를 반환하면, 이는 p 와 q 두 노드 중 하나가 아래에서 발견된 것입니다. 발견된 NULL 이 아닌 노드를 반환합니다.
  - 횡단 중 어느 지점에서 왼쪽과 오른쪽 하위 트리 모두 어떤 노드를 반환하면, 이는 노드 p 와 q 의 LCA 를 찾은 것입니다.

시간 복잡도: O(n), 공간 복잡도: O(n)

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()

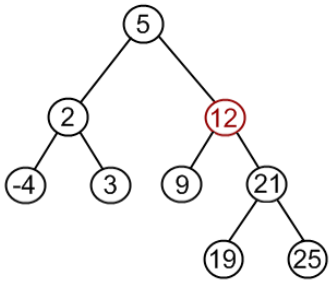
이 단계에서는 기본 함수 grow()와 trim()을 구현합니다. 강의 영상에서 grow()를 다뤘으니 trim() 함수를 한 번 구현해 봅시다. 이진 탐색 트리의 제거(trim) 작업은 추가(grow)와 탐색 작업보다 복잡합니다. 기본적으로, 이 작업은 두 단계로 나눌 수 있습니다.

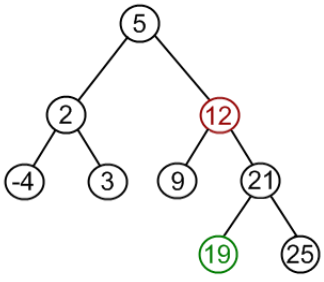
- 제거할 노드를 재귀적으로 탐색합니다. 예를 들어:
  ```
  if (key < node->key)
      node->left = trim(node->left, key);
  ```
- 결과적으로 이 **node→left** 의 값은 **trim(node→left, key)**이 완료되면 그 반환값으로 설정됩니다.
- 노드를 발견하면, 제거(trim) 알고리즘을 **재귀적으로** 실행합니다.

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.

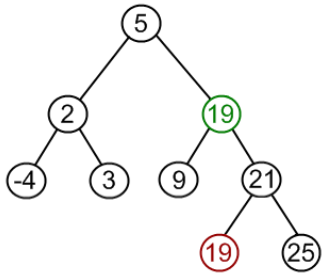
- 제거할 노드가 leaf 인(혹은 자식 노드가 없는) 경우:
  - 단순히 트리에서 노드를 제거합니다. 알고리즘은 (부모 노드의) 해당 링크를 **nullptr** 로 설정하고 노드를 제거합니다. **따라서 nullptr 을 반환합니다.**
- 제거할 노드의 자식 노드가 1 개만 있는 경우:
  - 노드를 임시(temp) 노드에 복사합니다.
  - 노드를 자식 노드로 설정합니다.
  - 임시 노드(혹은 제거할 기존 노드)를 해제합니다.
  - 재귀 trim()은 이 자식 노드를 (하위 노드와 함께) 제거된 노드의 부모 노드에 직접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

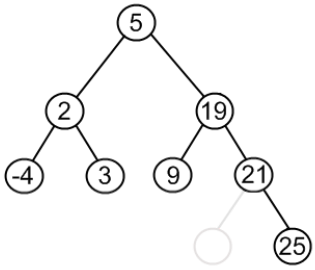
예시: BST 에서 12 를 제거합니다.





1. (오른쪽 하위 트리가 왼쪽 하위 트리보다 높으므로) 제거할 노드의 successor 를 찾습니다. 이 예시에서는 18 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.

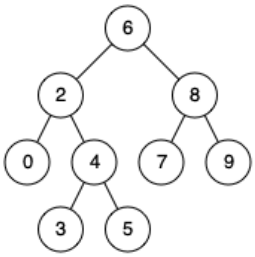
- Step 2:
  1. 배열의 키를 사용하여 순서대로 **trim()**을 **N 번** 호출합니다. for 문에서 **trim()이 트리의 새 루트를 반환할 수 있다**는 점을 기억하세요.

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

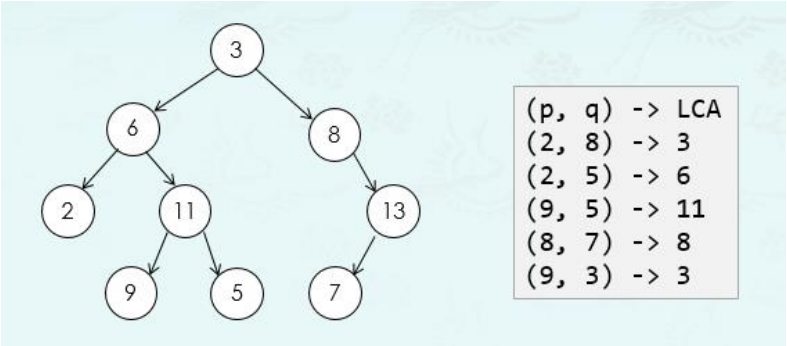
가장 낮은 공통 상위 노드(LCA)는 두 노드 p 와 q 사이에서 p 와 q 를 모두 하위 노드로 가진 T 의 가장 낮은 노드로 정의합니다 (**노드 자신을 자기 자신의 하위 노드**로 볼 수 있습니다).



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:

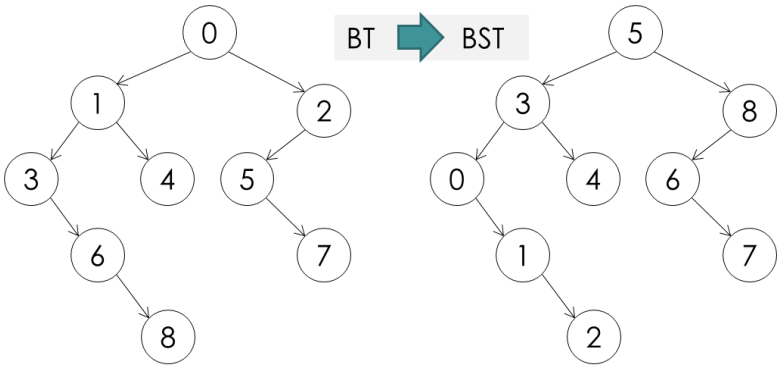
1. 루트 노드부터 트리 순회를 시작합니다.
2. 노드 p 와 q 모두 오른쪽 하위 트리에 위치한 경우, 오른쪽 하위 트리에서 step1 부터 시작하여 탐색을 계속합니다.
3. 노드 p 와 q 모두 왼쪽 하위 트리에 위치한 경, 왼쪽 하위 트리에서 step1 부터 시작하여 탐색을 계속합니다.
4. step2 와 step3 가 모두 참이 아니라면, 이는 노드 p 와 q 의 하위 트리와 공통되는 노드를 찾았다는 의미입니다. 따라서 해당 노드(공통 노드)를 LCA 로 반환합니다.

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.

Step 2.5: 제자리에서 BT 를 BST 로 변환하기

이 step 에서는 트리 구조를 그대로 유지하면서 이진 트리를 이진 탐색 트리로 변환하려고 합니다. 아래에 있는 예시를 확인해 봅시다.



알고리즘: 배운 내용을 활용해 보기 위해 배열이 아닌 **STL 의 벡터 혹은 set 을 사용하세요.**

- Step 1 – 이진 트리의 key 를 컨테이너(STL 의 벡터 혹은 set)에 저장합니다.
- Step 2 – 임의의 정렬 기법을 사용하여 벡터를 정렬합니다. STL 의 set 은 이미 정렬되어 있습니다.
- Step 3 – 이제 트리를 중위(inorder) 순회하고 컨테이너의 요소를 하나씩 트리의 노드에 복사합니다.

Step 3: AVL Tree

이 step 에서는 모든 노드의 왼쪽 하위 트리와 오른쪽 하위 트리의 높이 차이가 1 을 넘길 수 없는 BST(자체 균형 이진 탐색 트리)인 AVL 트리를 완성합니다. 이 높이 차이를 균형 계수(balance factor)라고 합니다. 대부분의 BST 작업(예. find, maximum, minimum, grow, trim 등등)은 h 가 BST 높이를 나타내는 O(h) 시간이 소요됩니다. 이 작업의 연산 비용은 편향된 이진 트리의 경우 O(n)이 될 수 있습니다. 삽입과 제거 작업을 모두 끝낸 후 트리의 높이가 O(Log n)으로 유지되도록 만들면 이러한 작업들에 O(Log n) 상한 범위를 보장할 수 있습니다. AVL 트리의 높이는 항상 n 이 트리의 노드 수를 나타내는 O(Log n)입니다.

AVL 트리를 생성하기 위해 명령행 인자를 -a 로 설정하여 AVL 트리와 함께 treeDriver.cpp 프로그램을 시작할 수 있습니다.

Step 3.1 growAVL() & trimAVL() - 1 점

growAVL()와 **trimAVL()**를 구현하세요.

삽입과 제거 작업을 모두 끝낸 후 주어진 트리를 AVL 로 유지하기 위해 표준 BST grow/trim 연산이 트리의 균형을 재조정하도록 만듭니다. 또한, growAVL()과 trimAVL()에서 호출되는 rebalance() 함수를 구현합니다.

rebalance() 함수는 grow 와 trim(삽입과 제거) 작업이 진행되는 동안 AVL 트리의 균형을 재조정합니다. 노드에서 균형 재조정 계수(rebalance factor)를 확인하고, 필요에 따라 rotateLL(), rotateRR(), rotateLR(), rotateRL() 함수를 호출합니다.

AVL 트리가 올바르게 형성되었는지 확인하는 방법 중 하나는 노드의 수와 트리의 높이를 비교하는 것입니다. 이 방법은 다음 공식을 따릅니다.

$h \leq 1.44\, log_2 n$

예를 들어, n = 1024 인 경우, 트리의 높이가 14.4 보다 작습니다. 노드의 수를 두 배로 만들 거나 n = 2048 로 만들어도 높이는 1 만 증가해야 합니다.

Step 3.2 - reconstruct() - 2 점

제대로 작동하고 있는 것처럼 보이는 AVL 트리의 growN()과 trimN()에 대해 생각해 봅시다.

1. growN() & trimN() 복습

"grow N"과 "Trim N" 옵션에 제공되는 두 함수 growN()과 trimN()은 작은 N 에 대해 잘 작동합니다. 이 두 함수는 아래와 같이 노드를 삽입 또는 제거할 때마다 growAVL 과 trimAVL 함수를 사용합니다. 물론, 이 방법은 비용이 많이 드는 rebalance() 함수를 계속 호출하므로 큰 N 을 가진 AVL 트리에서는 허용되지 않습니다.

```
tree growN(tree root, int N, bool AVLtree) {
   DPRINT(cout << ">growN N=" << N << endl;);
   int start = empty(root) ? 0 : value(maximum(root)) + 1;

   int* arr = new (nothrow) int[N];
   assert(arr != nullptr);
   randomN(arr, N, start);

   // use its own grow() function, respectively. it is too slow for AVL tree.
   if (AVLtree)
   for (int i = 0; i < N; i++) root = growAVL(root, arr[i]); //// UNACCEPTABLE CODE ////
   else
   for (int i = 0; i < N; i++) root = grow(root, arr[i]);

   delete[] arr;
   DPRINT(cout << "<growN size=" << size(root) << endl;);
   return root;
}
```

rebalance() N 번 호출을 피하는 방법은 (AVL 트리 역시 BST 이므로) BST 함수를 사용하여 trim(또는 grow)를 N 번하는 것입니다. trim 또는 grow 를 N 번 완료한 후, 루트에서 reconstruct()를 한 번 호출합니다. 또한, reconstruct()가 아래와 같이 효율적으로 동작해야 합니다.

```
// inserts N numbers of keys in the tree(AVL or BST), based
// on the current menu status.
// If it is empty, the key values to add ranges from 0 to N-1.
// If it is not empty, it ranges from (max+1) to (max+1 + N).
// For AVL tree, use BST grow() and reconstruct() once at root.
tree growN(tree root, int N, bool AVLtree) {
  int start = empty(root) ? 0 : value(maximum(root)) + 1;
  int* arr = new (nothrow) int[N];
  assert(arr != nullptr);
  randomN(arr, N, start);

  for (int i = 0; i < N; i++) root = grow(root, arr[i]);
  if (AVLtree) root = reconstruct(root);

  delete[] arr;
  return root;
}
```

```
// removes randomly N numbers of nodes in the tree(AVL or BST).
// It gets N node keys from the tree, trim one by one randomly.
// For AVL tree, use BST trim() and reconstruct() once at root.
tree trimN(tree root, int N, bool AVLtree) {

// left out

return root;
}
```

trimN()을 구현할 때, 코드가 음수 또는 N 보다 큰 숫자를 처리하는지 확인해야 합니다. 양수가 아닌 숫자를 입력하면 제거할 노드가 없지만, 사용자의 입력값이 N 보다 크거나 같으면 모든 노드를 제거합니다.

여기까지 구현을 완료하면, reconstruct()를 호출하는 'w - switch to AVL or BST'를 **제외**하고 모든 메뉴 항목이 동작해야 합니다.

2. Reconstruct()

우리는 N 의 크기가 큰 작업에 growAVL()과 trimAVL()을 사용하지 않는 대신, BST 함수를 그대로 사용하고 reconstruct()를 한 번 호출하여 트리의 균형을 재조정합니다.

이 step 에서 기존 BST 에서 AVL 트리를 O(n) 시간에 재구성하는 reconstruct()를 구현하는 것이 목적입니다. BST 의 모든 노드를 제자리에서 재조정하는 것보다 빠릅니다. 이를 위한 두 가지 방법을 구현합니다.

노드가 10 보다 작거나 같은 트리의 경우, BST 에서 key 를 가져와 새 AVL 트리를 재생성합니다. 이 방법은 **재생성 방법**(recreation method)입니다.

노드가 10 보다 큰 트리의 경우, 기존 BST 에서 노드를 가져와 ASL 트리로 재배치합니다. 이 방법은 **재활용 방법**(recycling method)입니다.

여러 가지 방법이 존재합니다. 아래에서 3 가지 방법을 제시할 것입니다. 다음은 뼈대 코드입니다.

```
// reconstructs a new AVL tree in O(n), Actually it is O(n) + O(n).
// Use the recreation method if the size is less than or equal to 10
// Use the recycling method if the size is greater than 10.
// recreation method: creates all nodes again from keys
// recycling method: reuses all the nodes, no memory allocation needed
tree reconstruct(tree root) {
   DPRINT(cout << ">reconstruct " << endl;);
   if (empty(root)) return nullptr;

  cout << "your code below" << endl;
   if (size(root) > 10) {    // recycling method
                             // use new inorder() to get nodes sorted
                             // O(n), v.data() - the array of nodes(tree) sorted
                             // root = buildAVL(v.data(), (int)v.size()); // O(n)
   }
   else {                    // recreation method
                             // use indorder() to get keys sorted
                             // O(n), v.data() - the array of keys(int) sorted
                             // clear root
                             // root = buildAVL(v.data(), (int)v.size()); // O(n)
   }
   DPRINT(cout << "<reconstruct " << endl;);
   return root;
}
```

방법 1: 우리가 생각할 수 있는 첫 번째 방법은 루트부터 시작해서 아래로 트리를 순회하는 동안 필요한 경우 재균형(rebalance) 연산을 적용하는 것입니다. 이 방법은 가능한 방법이지만 비용이 많이 드는 rebalance()를 여러 번 호출해야 하므로 비용이 너무 많이 들 수 있습니다. 따라서 이 해결책은 **수용할 수 없습니다**.

방법 2 (재생성 방법): 또 하나의 효율적인 방법은 기존의 BST 알고리즘과 함수의 주요 특징 중 하나를 사용하는 방법입니다. 알고리즘은 다음과 같습니다:

1. **key** 를 **정렬된 배열**로 반환하는 중위 순회(inorder traversal)의 특성을 사용합니다.
2. 해당 배열로부터 균형 트리(balanced tree)를 형성합니다 (배열 중간에서 루트를 선택하고 문제를 재귀적으로 분할하여 구현할 수 있습니다). 균형 트리는 AVL 정의를 충족합니다.

기존 트리의 할당을 해제하고 전체 트리를 재생성합니다. 두 작업 모두 O(n) 시간 내에 쉽게 수행할 수 있습니다. 방법 2 와 3 의 뼈대 코드가 제공됩니다.

다음 inorder()와 벡터의 data() 함수를 사용하여 **key 의 배열**을 얻을 수 있습니다.

```
void inorder(tree t, std::vector<int>& v);    // traverses tree in inorder & returns keys
```

```
// rebuilds an AVL tree with a list of keys sorted.
// v - an array of keys sorted, n - the array size
tree buildAVL (int* v, int n) {
  if (n <= 0) return nullptr;
```

```
  // create a new root and set a TreeNode and initial values, use the [mid] element of v.

  // your code here - recursive buildAVL() calls for left & right
                              // from 0 to mid-1 (or mid number of nodes)
  // from mid+1 to the end (how many nodes?)
   return root;
}
```

방법 3 (재활용 방법): 노드의 key 대신 노드의 배열을 얻는 것을 제외하고는 방법 2 와 동일합니다. 모든 노드를 그대로 활용하고 알고리즘에 따라 노드를 다시 연결합니다. tree.h 에 추가된 inorder()의 함수 프로토타입은 트리의 key 에 의해 정렬된 모든 노드를 가진 vector<tree> type 을 반환합니다. 이 알고리즘은 새 노드를 다시 생성하지 않고 트리의 기존 노드만 사용합니다. 다음과 같은 inorder()과 벡터의 data() 함수를 사용해서 **노드의 배열**을 얻을 수 있습니다.

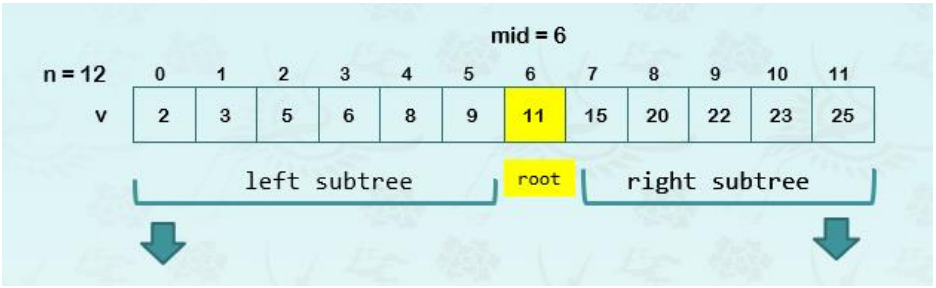
```
void inorder(tree t, std::vector<tree>& v);  // traverses tree in inorder & returns nodes
```

```
// rebuilds an AVL tree using a list of nodes sorted, no memory allocations
// v - an array of nodes sorted, n - the array size
tree buildAVL(tree* v, int n) {
  if (n <= 0) return nullptr;

  // set the mid node as a root.

  // recursive calls for left & right
                              // from 0 to mid-1 (or mid number of nodes)
  // from mid+1 to the end (how many nodes?)

  return root;
}
```

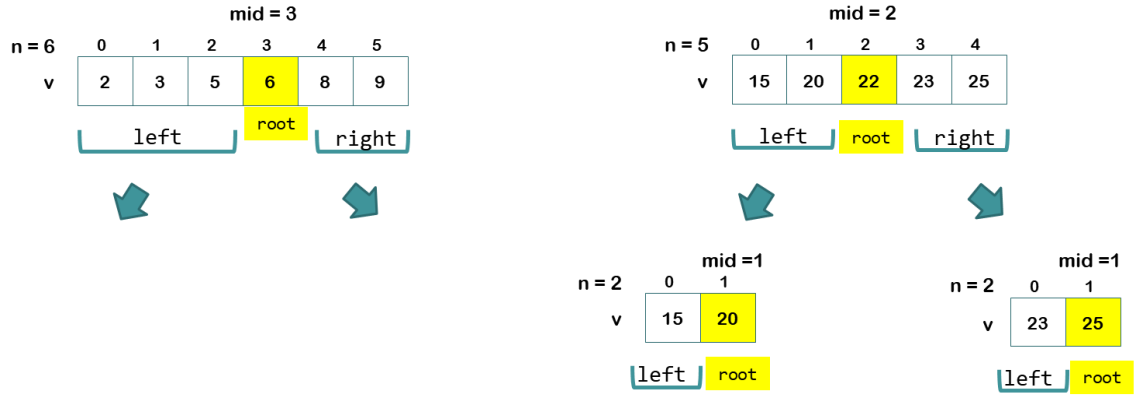
힌트: 두 방법 모두 O(n) 시간 복잡도를 가지고 있지만, 마지막 두 방법의 시간 차이를 확인해 보면 꽤 흥미로울 것입니다.

3. buildAVL() 함수 예시

BST 의 요소 배열인 v 와 v 의 크기인 n 이 있습니다. 배열 v 는 key 또는 노드에 inorder()를 사용하여 가져올 수 있습니다. buildAVL()의 첫 번째 인수로 다음 데이터가 있다고 가정합시다.



위에 표시된 인수들 중, 가운데 요소를 루트로 사용합니다. 배열의 첫 번째 반절은 왼쪽 하위 트리를, 두 번째 반절은 오른쪽 하위 트리를 재귀적으로 형성합니다.



Step 3.4 show 모드 - 1 점

"**show tasty**" 모드를 개선합니다. 트리가 크게 자라면 레벨과 노드가 너무 많아서 다 볼 수 없습니다. 현재는 노드가 많을 경우 처음 3 행과 마지막 3 행만 보여줍니다. 각 레벨에서 **show_n** 개의 노드만 보여주도록 기능을 개선하세요.

treeprint.cpp 의 다음 함수를 **수정**합니다. 또한, 필요한 경우, treeprint.cpp 에 도우미 함수를 생성할 수 있습니다.

```
void treeprint_levelorder_tasty(tree root)
```

편의를 위해 tree.cpp 에 도우미 함수 show_vector()가 제공됩니다.

```
void show_vector(std::vector<int> vec, int show_n = 20);
```

과제 제출

- 소스 파일 상단에 아래와 같이 아너 코드 문장을 적고 서명하세요.
  On my honor, I pledge that I have neither received nor provided improper assistance in the completion of this assignment.
  서명: _____ 분반: _____ 학번: _____
- 제출하기 전에 코드가 제대로 컴파일이 되고 실행되는지 확인하세요. 제출 직전에 급하게 코드를 수정한 후 코드가 제대로 컴파일이 될 거라고 짐작하지 않는 게 좋습니다. "거의" 작동하는 코드도 틀린 것입니다.
- 과제가 컴파일 및 실행된다면, 마감 기한 전까지 과제의 일부만 완성했더라도 제출하기 바랍니다. 컴파일 및 실행되지 않는다면 제출하지 마세요. 마감 시간 이후 24 시간 이내 제출하면, 만점에서 25% 감점하고 채점합니다. 그 이상 늦은 것은 채점하지 않으며, 0 점 처리합니다.
- 제출 후, **마감 기한 전까지** 수정 및 재제출이 가능합니다. 파일 하나만 수정하더라도 해당 파일과 관련된 파일들을 모두 재제출해야 합니다. 재제출 횟수는 제한 없습니다. 마감 기한 전에 **가장 마지막으로** 제출된 파일을 채점할 것입니다.

제출 파일 목록

- **pset10** for BT - tree.cpp
  'clear'을 포함한 BT menus

- **pset11** - tree.cpp
  BT & BST menu 항목이 함께 동작해야 합니다.
- **pset12** - tree.cpp, treeprint.cpp
  BT, BST and AVL 메뉴 항목이 함께 동작해야 합니다.

마감 기한 & 배점

- Binary Tree: Step 1.1 ~ 1.5
- Binary Search Tree: Step 2.2 ~ 2.5
- AVL Tree: Step 3.1 ~ Step 3.4

참고 문헌

1. Recursion :
2. Recursion: