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

Pset on AVL Tree

목차

소개

본 PSet 은 앞에서 사용한 파일을 그대로 사용하면서 AVL 트리를 다루는 함수를 완성하는 Problem Set 입니다. 다음 파일들이 제공됩니다. 이미 BT/BST 를 구현한 tree.cpp 가 있다면, 이를 가져와서 사용하고, Step 3.1 부터 시작할 수 있습니다.

- tree-avl.pdf : 이 파일, PSet 에 대한 설명
- **driver.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.

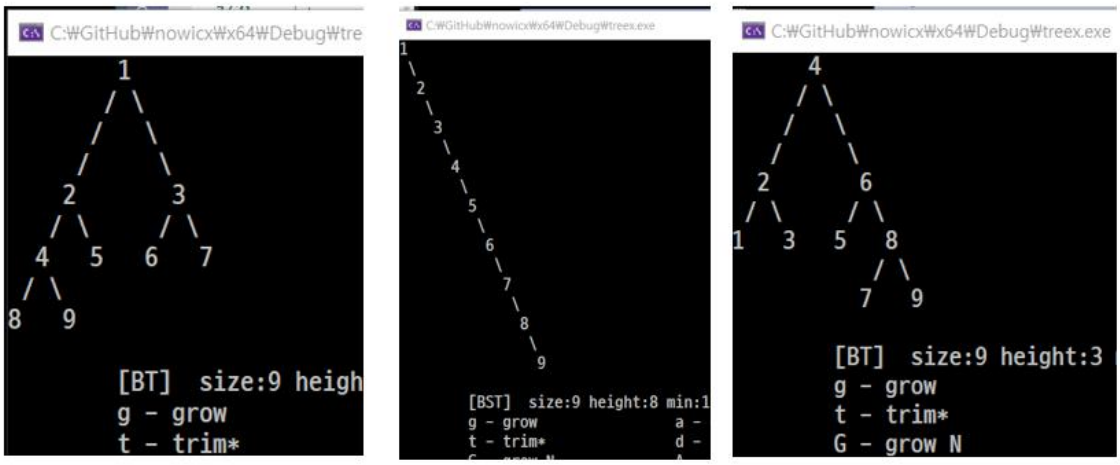
```
PS C:\GitHub\nowicx\psets\pset10-12tree> ./treex -b 1 2 3 4
    1
   / \
  2   3
 /
4

        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

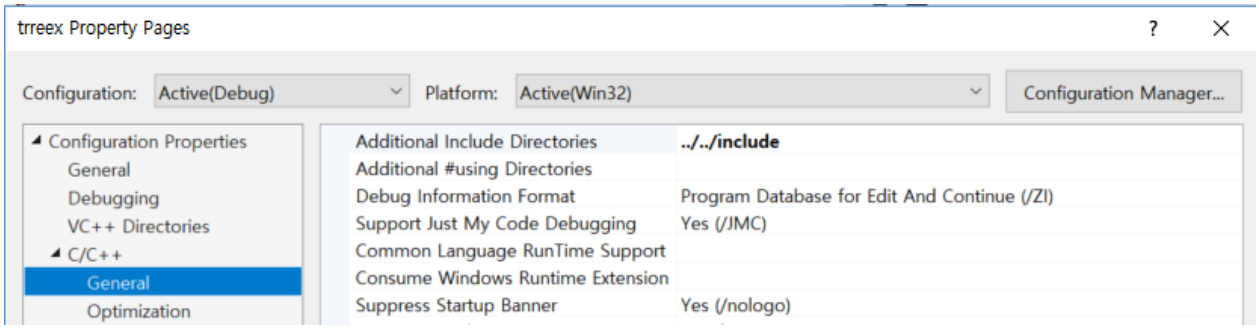
Jump-start 의 경우, 먼저 tree 라는 이름의 프로젝트를 생성합니다. 평소와 같이 다음 작업을 수행합니다.

- 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

idebtor@gmail.com

  - Project Property → Linker → Input → Additional Dependencies
- Add /D "DEBUG" at
  - Project Property → C/C++ → Command Line

예시:

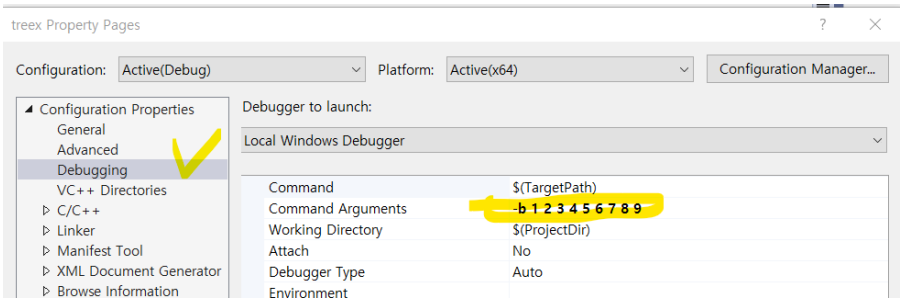


Add ~.h files under 프로젝트 'Header Files' 아래에 ~.h 파일을 추가하고 프로젝트 'Source Files' 아래에 ~.cpp 파일을 추가하면 프로젝트를 빌드할 수 있습니다.

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 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.

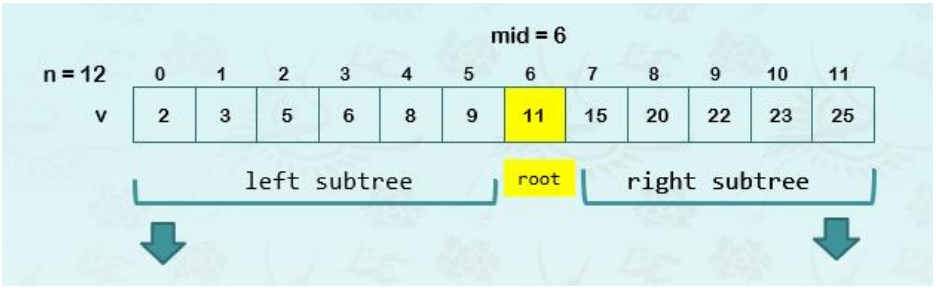
```
void inorder(tree t, std::vector<tree>& v);  // traverses tree in inorder & returns nodes
```

```
// 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;
}
```

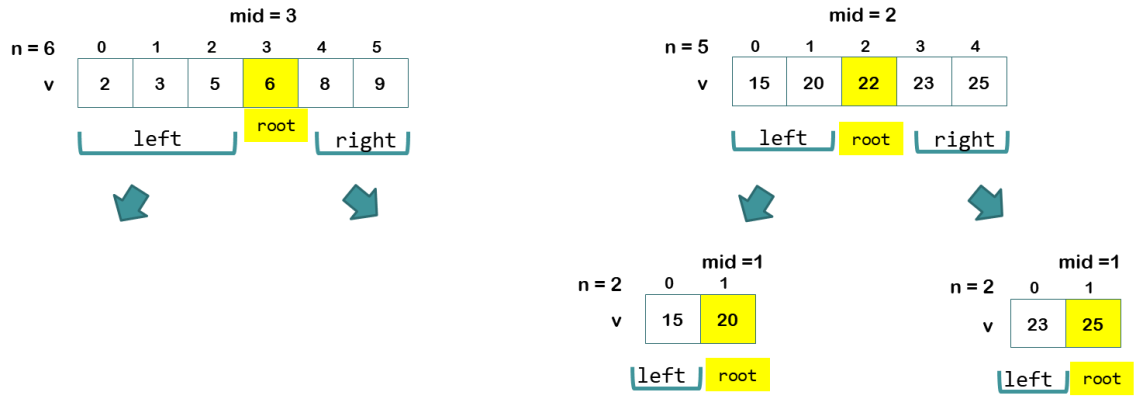
힌트: 두 방법 모두 O(n) 시간 복잡도를 가지고 있지만, 마지막 두 방법의 시간 차이를 확인해 보면 꽤 흥미로울 것입니다.

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 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

- **Pset AVL Tree:** Step 3.1 ~ 3.2, Step 4
  - tree.cpp, treeprint.cpp

참고 문헌

1. Recursion :
2. Recursion: