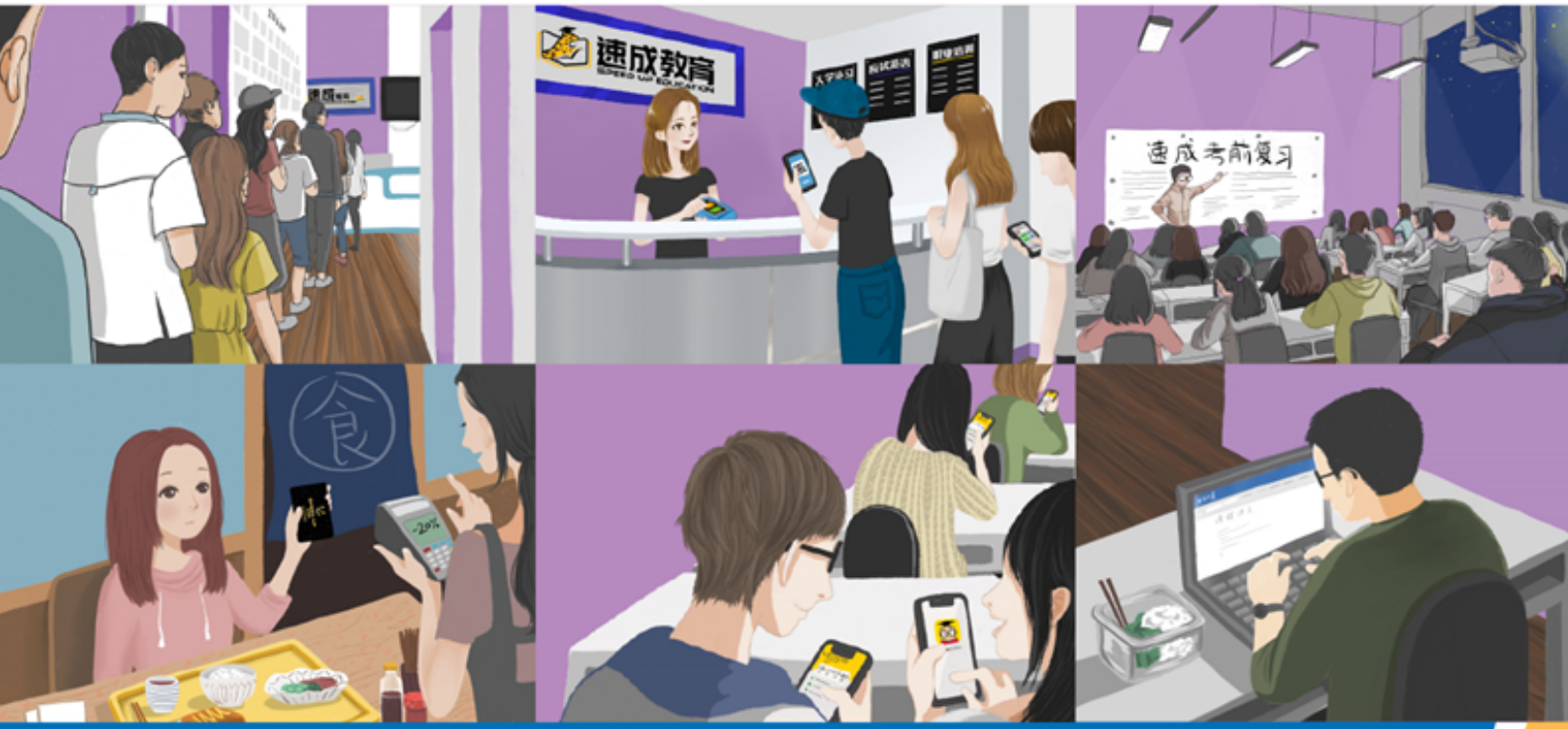
速成教育
SPEED UP EDUCATION



CSCA48

Introduction to Computer Science II

导师： **VC**

UTSC BST Practices | 2025/4/12

Disclaimer

This study handout is provided by Speed Up Education and its affiliated tutors. The study handout seeks to support your study process and should be used as a complement, not substitute to the university course material, lecture notes, problem, sets, past tests and other available resources.

The study handout is distributed for free to the students participating in Speed Up Education classes and is not for sale or other commercial uses whatsoever. We kindly ask you to refrain from copying or selling in part or in whole any information provided in the handout.

Thanks for choosing Speed Up Education. We wish you the best of luck in your exam.

BST 練習

a) All Binary Search Trees' branching factor must be 2. (True / False)

at most 2

b) Given that a binary tree's height is 9. What are the minimum and maximum number of nodes that this tree can have?

min = 9 max = $2^9 - 1$

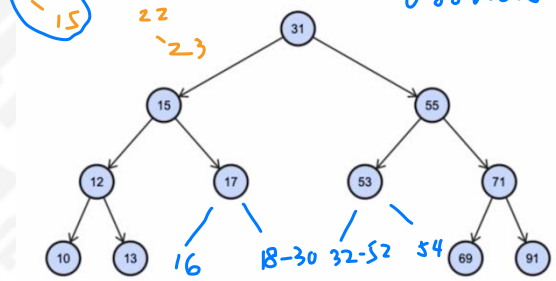
c) The following is the "preorder" traversal of a Binary Search Tree:

18, 16, 13, 15, 19, 24, 22, 23

What is the size (number of nodes) of the subtree whose root is 16?

3

d) Give the range(s) of values that could be added to the tree that would not increase its height.



b) Consider inserting a sequence of 7 values, in the following order:

43, 21, 32, __hidden__, 42, 40, 35

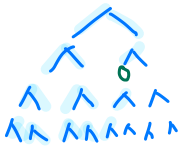
We know that the hidden value is an integer that is "different" from all other values in the sequence, and we lnow that the resulting BST has a height of 7, i.e., the maximum possible height with 7 nodes.

Write down all possible values of the hidden value.
If no possible value exists, write "impossible" as the answer.
If infinitely many values are possible, write "infinite" as the answer.

33, 34

What is the Big O average-case complexity of finding and printing all keys in a BST of integers that are smaller than some value c.

a) O(N) b) O(N Log N) c) O(N^2) d) O(Log N) e) O(Log N^2)



A BST would have the same performance (in terms of average case Big O complexity) for searching in a large collection as this data structure:

a) A linked list
b) An unsorted array
c) A sorted array
d) A sorted linked list
e) An array of size Log N where each entry is a linked list of items

A BST would not be a suitable data structure in which to store (one) of the following (which is it?)

a) Products on an on-line store
b) A library catalog (of books and magazines)
c) Student records for UTSC students
d) Lines of text in a text document
e) Articles in a scientific journal

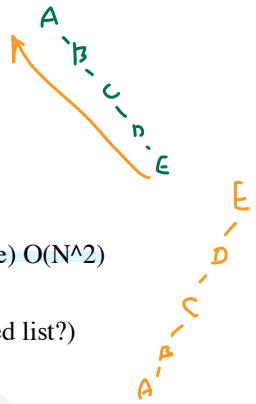
A Linked List would be a better than a BST for one of the applications below, which one is it?

a) Storing library book records
b) Air Traffic Control queue of departing flights
c) An airline's flight reservations system
d) Youtube's music video database
e) Storing the product catalog of a supermarket

We're implementing a product database for an on-line store, and choosing which data field(s) to use for the key. All options below have issues, but one of them is particularly bad for performance. Which one is it?

a) Product Name + Brand
b) Bar-code number
c) Product Name + number
d) Date & Time product added to collection (e.g. Soap 0001)
(e.g. YYYYMMDDHHSS - year, month, day, hours, seconds)
e) Brand + number (e.g. MyCompany 0001)

Given what you know about the complexity of specific operations in BSTs, linked lists, arrays, and sorting, estimate the **worst-case complexity** of the algorithm described by the following pseudocode (**Think carefully about this algorithm before you answer**).



ReshapeBST - input: A BST resulting from inserting N values in sorted order, output: A reshaped BST

* Initialize an empty, output BST, call this BST_out O(1)
* Do a post order traversal on the input BST, inserting the value at each visited node into the output tree BST_out O(h) O(n)

a) O(N) b) O(N Log N + N) c) O(N Log N + N^2) d) O(Log N + N) e) O(N^2)

Is the resulting BST from 'ReshapeBST' a nicely shaped tree? (i.e. is it now no longer shaped like a linked list?)

a) yes b) no c) it's complicated.

Given what you know about the complexity of specific operations in BSTs, linked lists, arrays, and sorting, estimate the **average-case complexity** of the algorithm described by the following pseudocode **(Think carefully about this algorithm before you answer)**.

Tree2SortedArray - input: A set of N randomly ordered numbers, output: A sorted array with these numbers

* Initialize an empty BST O(1) * Insert all random numbers into the BST O(n · log n)
* Carry out an in-order traversal of the BST inserting each entry into a sorted array O(n) O(1)

a) O(N) b) O(N Log N + N) c) O(N Log N + N^2) d) O(Log N + N) e) O(N^2)

TreeDuplicate - input: A BST, output: A duplicate (identical) BST

* Initialize an empty BST O(1)
* Perform a pre-order tree-traversal on the input BST O(n)
  - Make a copy of each node visited, and insert it into the output BST O(log n)

a) O(N) b) O(N Log N + N) c) O(N Log N + N^2) d) O(Log N + N) e) O(N^2)

RandomShuffle - input: A BST with integers, output: A linked list where the numbers are **in random order**

* Initialize output list to empty, length=0 O(1)
* Traverse the BST in order O(n)
  - For each value, choose a random number between 0 and length O(1)
  - Insert the value at the corresponding location in the list O(n)
  - length=length+1 O(1)

a) O(N) b) O(N Log N + N) c) O(N Log N + N^2) d) O(Log N + N) e) O(N^2)

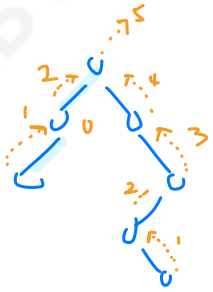
以下三题 不能思考超過 10 秒鐘, 不然不用去考試了

```
BSTNode *maximum(BSTNode *root) {

    if (root == NULL) return NULL;
    if (root->right == NULL) return root;
    return maximum(root->right);


}
```

```
def f1():
    return f2()

def f2():
    return 2

x = f1()
```

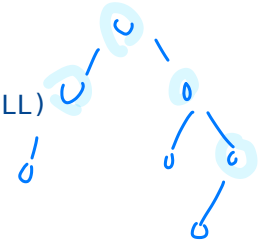
```
int
BSTNode *size(BSTNode *root) {

     if (root == NULL) return 0;
     return 1 + size(root->left) + size(root->right);



}
```

```
int count(BSTNode *root, int x) {

    if (root == NULL) return 0;
    if (root->data == x) return 1 + count(root->right, x);

    if (root->data > x) return count(root->left, x);
    if (root->data < x) return count(root->right, x);


}
```



```
int height(BSTNode *root) {

     if (root == NULL) return 0;
     int lh = height(root->left);
     int rh = height(root->right);
     if (lh > rh)
        return lh + 1;
     else
        return rh + 1;
}
```

```
int count_non_leaf(BSTNode *root) {

    if (root == NULL) return 0;
    if (root->left == NULL && root->right == NULL)
       return 0;


    return 1 + count_non_leaf(root->left) +
               count_non_leaf(root->right);



}
```



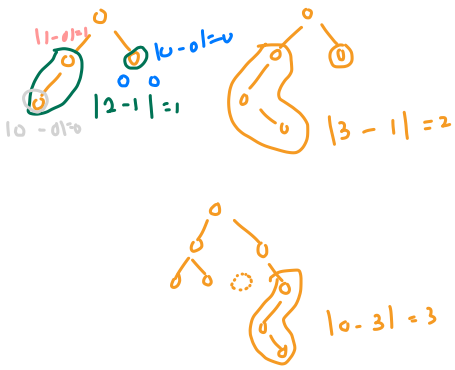
Assume that **`int maximum(Node *root)`** and **`int minimum(Node *root)`** are implemented properly.
Your solution must be recursive.

```
int is_bst(Node *root) {
    // Return whether this binary tree is a binary *search* tree.
    // An empty binary tree satisfies the binary search tree property.
    if (root == NULL) return 1;

    if (root->left && maximum(root->left) > root->val) return 0;

    if (root->right && minimum(root->right) < root->val) return 0;

    if (!is_bst(root->left) || !is_bst(root->right)) return 0;

    return 1;


  }
```

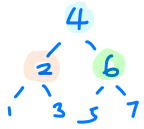
Assume that **`int height(BSTNode *root)`** is implemented properly.
Your solution must be recursive.

左 右 subtree's height 的 diff < 2

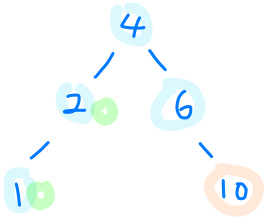
```
int is_balanced(BSTNode *root)
    // Return True iff the Tree is a balanced Tree.
    if (root == NULL) return 1;

    int lh = height(root->left);
    int rh = height(root->right);

    int diff = lh - rh;

    return (diff < 2 && -diff > -2) &&
       is_balanced(root->left) &&
       is_balanced(root->right);
```



```
BSTNode *build_balanced_bst(int lst[1024], int len) {
    // Return a balanced BinarySearchTree containing all values in <lst>
    // Precondition:
    //  - <lst> is SORTED in an increasing order
    //  - len(lst) >= 0
    //  - all values in <lst> are distinct, i.e., no duplicate values

    if (len == 0) return NULL;

    int mid = len // 2;

    BSTNode *root = (BSTNode *) calloc(1, sizeof(BSTNode));
    root->val = lst[mid];

    root->left = build_balanced_bst(lst, mid);

    root->right = build_balanced_bst(&lst[mid+1], mid);
                                      lst+mid

    return root;
```



```
int num_in_range(BSTNode *root, int start, int end) {
    // Return the number of items in the BST which are between <start>
    // and <end>, inclusive.

    if (root == NULL) return 0;

    // 太大了 往左
    if (root->val > end)
       return num_in_range(root->left, start, end)

    if (root->val < start)
       return num_in_range(root->right, start, end);

    // correct range, count
    return 1 + num_in_range(root->left, start, end) +
               num_in_range(root->right, start, end);
```



```
double unright_avg(BSTNode *root) {
    /**
    Return the average of all the numbers in the tree that are not
    stored in a node that is a right child (of another node).
    I.e., that all the nodes that are not right children (including root),
    and return the average of the numbers stored in them.

    Precondition: Tree is not empty

    Example:
            10
           /    \
          5      15          (10 + 5 + 1 + 12 + 6) / 5 = 5.5
        /   \  /   \
       1    8 12   17
           /
          6
    calling this function on the above tree will produce 5.5

    You must use the _unright_helper() defined on the next page.
    **/
    if (root == NULL) return 0.0;

    int sum = 0;
    int numnodes = 0;

    _unright_helper(root, 1, &sum, &numnodes);

    return sum / numnodes;
```

```
void _unright_helper(BSTNode *root, int include_root,
                     int *sum, int *numnodes) {
    /**
    Modify sum and numnodes so that:
    * sum is the sum of all the numbers in the tree that are not stored
      in a node that is a right child, plus the number stored at the root
      if include_root == True, and
    * numnodes is the number of nodes in the tree that are not right
      children, plus one if include_root == True

    Example code:

    BSTNode *root = ...;   // assume bst is the tree above,
    int sum = 0;
    int numnodes = 0;
    _unright_helper(bst, 1, &sum, &numnodes);
    printf("%d %d\n", sum, numnodes); // 22 4   34   5
    _unright_helper(bst, 0, &sum, &numnodes);
    printf("%d %d\n", sum, numnodes); // 12 3   24   4
    **/
    if (root == NULL) return;

    _unright_helper(root->left, 1, sum, numnodes);
    _unright_helper(root->right, 0, sum_numnodes);

    if (include_root) {
       *sum += root->val;
       *numnodes += 1;
    }
```