MONASH University
Information Technology

22nd and 24th April 2015
Monash University – Sunway Campus, Malaysia

FIT 2004
Algorithms and Data Structures

Tutorial/ Practical 07

Ian Lim
lim.wern.han@monash.edu

Tutorial07 Task01

- Dynamic programming
  - Optimal sub-structure
  - Sub-problems overlap
- Current problem: Longest Common Subsequence (LCS)

Tutorial07 Task01

- Given 2 string
  - String01 to be s1[1…m]
  - String02 to be s2[1…n]
  - Let’s have them inside LCS [i][j] matrix of size M X N

Tutorial07 Task01

- 2 possible situation
- Situation 1: If s1[i] == s2[j]
  - Then we are able to extend LCS[i-1][j-1] the character.
  - LCS[i][j] = LCS[i-1][j-1] + 1
- Situation 2: If s1[i] != s2[j]
  - Then we look for the maximum subsequence of the smaller substring earlier
    - LCS[i][j-1]
    - LCS[i-1][j]
  - LCS[i][j] = max (LCS[i][j-1], LCS[i-1][j])

Tutorial07 Task02

- Binary Search Tree (BST)
  - All of you should be familiar now having done Practical06

Tutorial07 Task02

- Binary Search Tree (BST)
  - All of you should be familiar now having done Practical06
  - Given a tree to be fork(root, left, right) then we establish that Left < Root < Right

Tutorial07 Task02

- Binary Search Tree (BST)
  - All of you should be familiar now having done Practical06
  - Given a tree to be fork(root, left, right) then we establish that Left < Root < Right
  - Thus for this question, one can easily establish the need for a recursive algorithm for the traversal of trees
    - > Traversal would be limited to the condition given to be within lo and hi (inclusive)

Tutorial07 Task02

- Now take some time to come up with the algorithm

Tutorial07 Task02

- Now take some time to come up with the algorithm

Tutorial07 Task02

- Solution

```python
def between(T, lo, hi):
    if (T is None):
        return
    if (lo < T.value) and (T.left is not None):
        between(T.left, lo, hi)
    if (T.value < hi) and (T.right is not None):
        between(T.right, lo, hi)
```

Tutorial07 Task02

- Solution
  - Not complete because?

```python
def between(T, lo, hi):
    if (T is None):
        return
    if (lo < T.value) and (T.left is not None):
        between(T.left, lo, hi)
    if (T.value < hi) and (T.right is not None):
        between(T.right, lo, hi)
```

Tutorial07 Task02

- Solution
  - Now complete

```python
def between(T, lo, hi):
    if (T is None):
        return
    if (lo < T.value) and (T.left is not None):
        between(T.left, lo, hi)
    # adds print here to print in order
    if (lo <= T.value <= hi):
        print(T.value)
    if (T.value < hi) and (T.right is not None):
        between(T.right, lo, hi)
```

Tutorial07 Task03

- Simple problem
- Try work it out yourself first

Tutorial07 Task03

- Simple problem
- Try work it out yourself first

Tutorial07 Task03

- Insert
  - Just add it in the right position
  - Uphold the requirement that left < root < right

Tutorial07 Task03

- Insert
  - Just add it in the right position
  - Uphold the requirement that left < root < right
- Delete
  - Slightly more tricky but relatively the same concept
  - Swap the node with the biggest-left leaf or the smallest-right leaf
  - Maintains the requirement that left < root < right

Tutorial07 Task04

- AVL Trees
  - Slightly more fun and also complex than that of BST in Task03

Tutorial07 Task04

- Requirement of AVL Trees
  - Given a tree fork(root, left, right)

    height(left) – height(right) <= abs(1)

Tutorial07 Task04

- Requirement of AVL Trees
  - Given a tree fork(root, left, right)
    height(left) – height(right) <= abs(1)
  - Done in a recursive manner (bottom up from the tree)

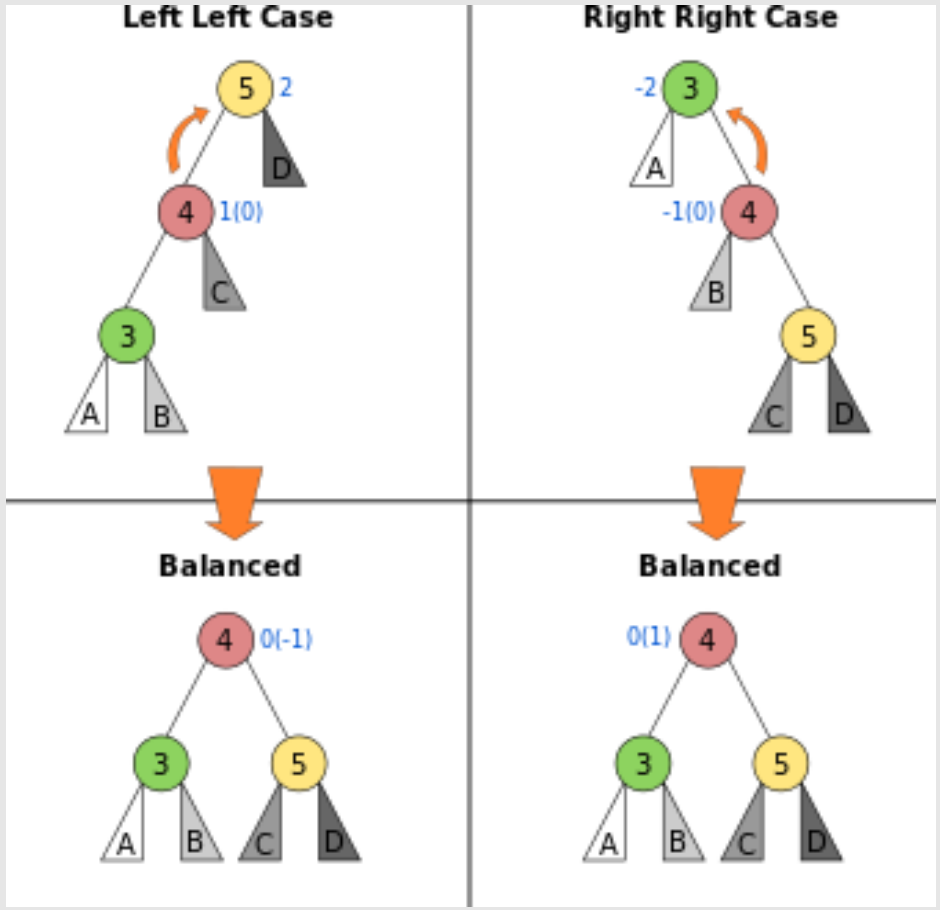
Tutorial07 Task04

- Requirement of AVL Trees
  - Given a tree fork(root, left, right)
    height(left) – height(right) <= abs(1)
  - Performs rotation to maintain the above rule!
    - LL
    - LR
    - RR
    - RL

Tutorial07 Task04

- Requirement of AVL Trees
  - Given a tree fork(root, left, right)
    height(left) – height(right) <= abs(1)
  - Performs rotation to maintain the above rule!
    - LL: Simple
    - LR
    - RR: Simple
    - RL

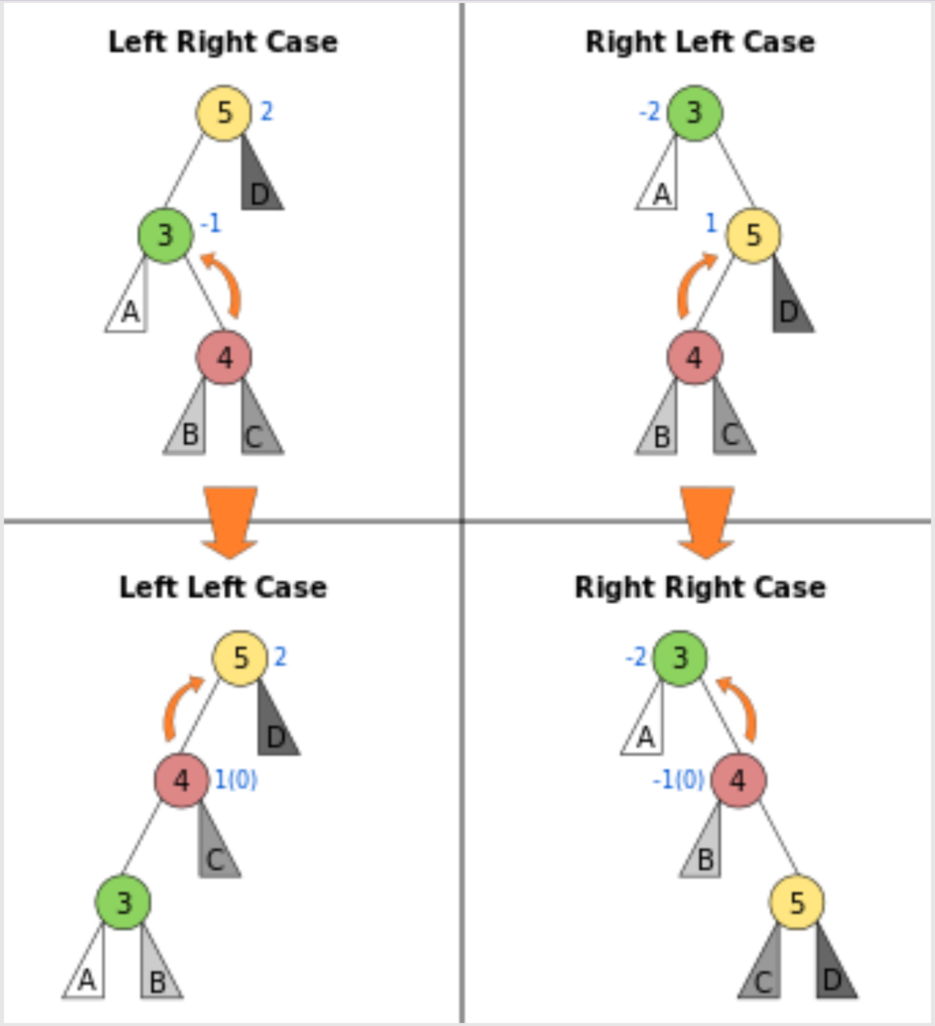
Tutorial07 Task04



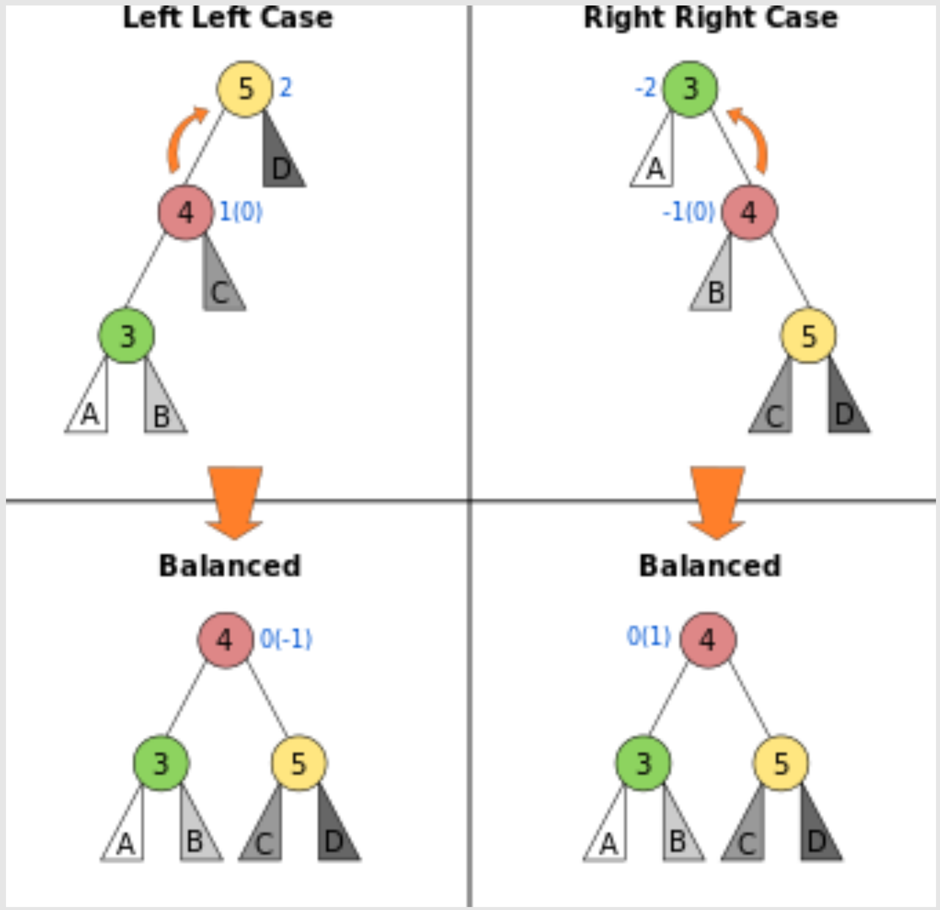
Tutorial07 Task04

- Requirement of AVL Trees
  - Given a tree fork(root, left, right)
    height(left) – height(right) <= abs(1)
  - Performs rotation to maintain the above rule!
    - LL: Simple
    - LR: Convert to LL first
    - RR: Simple
    - RL: Convert to RR first

Tutorial07 Task04



Tutorial07 Task04



Tutorial07 Task05

- Suffix Tries.
- Pretty difficult (for me personally) and very tedious.
- Give it a try now pen and paper
- We would discuss it later

Data Structures

- Abstract Data Type (ADT)
  - Keep this in mind when coding as this is an issue faced by many of you (mainly due to the nature of Python also)
  - Code following the definition of the data structure.
    - Example the HashTable.insert(key, data)

Practical07

- Abstract Data Type (ADT)
  - Keep this in mind when coding as this is an issue faced by many of you (mainly due to the nature of Python also)
  - Code following the definition of the data structure.
    - Example the HashTable.insert(key, data)

Practical07

- Tries
  - Several ways to implement this
  - You can be creative :3

Practical07

- AVL Trees
  - What are the main rules to follow?
    - Height balanced
  - Thus the main components
    - Calculate height
    - Calculate balance
    - Right rotate
    - Left rotate

Practical08

- Anything to clarify or discuss?
- Pretty simple practical
  - So much easier than Practical06 (Prac08 nerfed?)
  - In fact last year's FIT 2004 only have Task02 as their only question so yea, no nerf…

HashTable

- Will do this when I have time.
- There are a lot of variants

HashTable

- Insert(key, data)
  - Insert data into the HashTable, using Key to get the position quickly
    - How?

HashTable

- Insert(key, data)
  - Insert data into the HashTable, using Key to get the position quickly
    - How? By hashing the key
      - Relies on the hash function
    - **Follow this definition! NO OPERATIONS**

HashTable

- Insert(key, data)
  - Insert data into the HashTable, using Key to get the position quickly
    - How? By hashing the key
      - Relies on the hash function
      - Chaining/ probing/ bucketing if required
    - Follow this definition!
  - Roughly O(1) best and worst case
    - Ensure this with resizing
    - When to resize?
      - When it is getting full
      - When the chain is going too long

HashTable

- Insert(key, data)
  - No repeated keys
    - Data with the same key can be added as into the bucket with the Entry having a key and a List<Data>

HashTable

- Lookup(key)
  - Search for the key quickly
    - Via hashing of key
    - With chaining/ probing/ bucketing if required
  - Retrieve
    - Data if the key is found
    - None if key is not found
    - No printout, no return of False etc.
      - Remember, consistent with definition and ADT!

HashTable

- Delete(key)
  - Delete the Entry with the key
    - Lookup via hashing of key
    - With chaining/ probing/ bucketing if required
  - How to delete? With respect to chaining
    - Remove the Entry
      - Need to ensure the chain is maintained by shifting the other elements in the chain over
      - Pretty simple if you keep the chain short
    - Set Entry to a special Delete flag
      - Maintains the chain for probing
      - Remove/ Ignore when Resize

HashTable

- Resize()
  - Easiest one of all!
  - Save all entries into the a temporary list
    - Ignore all None
    - Ignore all with Delete flag
  - Insert(key, data) for all Entry
    - Totally fine since insert have no operation

HashTable

- Bonus: Entry
  - Must contain key and data
  - Can optionally contain other information to ease out operations
    - Hash value without chain H(key,0)
  - Declared as an inner class that encapsulate information you need

MONASH University
Information Technology

22nd and 24th April 2015
Monash University – Sunway Campus, Malaysia

Thank You