Algorithm Analysis & Design
Mid Term Exam

May 8th, 2025, 10:00 – 12:30 WIB

Instruction

There are two kinds of problem set: A & B. Solve all tasks in problem A will net you at most 0.7. Whereas solve problem B will net full score. Problem A is typical Binary Search Tree API completion given a function body. In contrast problem B will test your instinct in applying binary search tree in real world application. Hence the reason for the full score. Solving both will still net you 1, so choose wisely.

Manual Book

Type in terminal "rustup docs"

Problem A

Implement the following new procedures in Binary Search Tree module.

```
fn add_node(&self, target_node: &BstNodeLink, value: i32)→ bool
```

This function will seek target_node in the current self binary search tree. When the target_node is found, the function will add new node value under the target_node. Since it's possible target_node doesn't exist in the current tree, return status code.

```
fn tree_predecessor(node &BstNodeLink) → Option<BstNodeLink>
```

Seek next immediate predecessor on the current node. Return the predecessor node if found, else None.

```
fn median(&self) → BstNodeLink
```

Get the approximate median node in the whole tree. Using similar but slightly different technique to minimum & maximum. Traverse the tree to obtain median node.

```
fn tree_rebalance(node: &BstNodeLink)  → BstNodeLink
```

This function will process unbalanced binary search tree in node, and generate approximately balanced binary search tree. Using array data structure and sort is strictly forbidden. All processing need to re-traverse the binary search tree. Hint: usage of existing predefined function including the previous functions will assist you in finding of how to rebalanced the tree.

Put all these functions into bst.rs.

Problem B

Tree structure has important usage in database engine. Fast look up for any query is usually depends on an index. An index is simply a tree based structure which hold a mapping of an unique keys. For simplification in the exam, we'll only consider integer index.

Consider the following scenario, a column may hold these values

1234

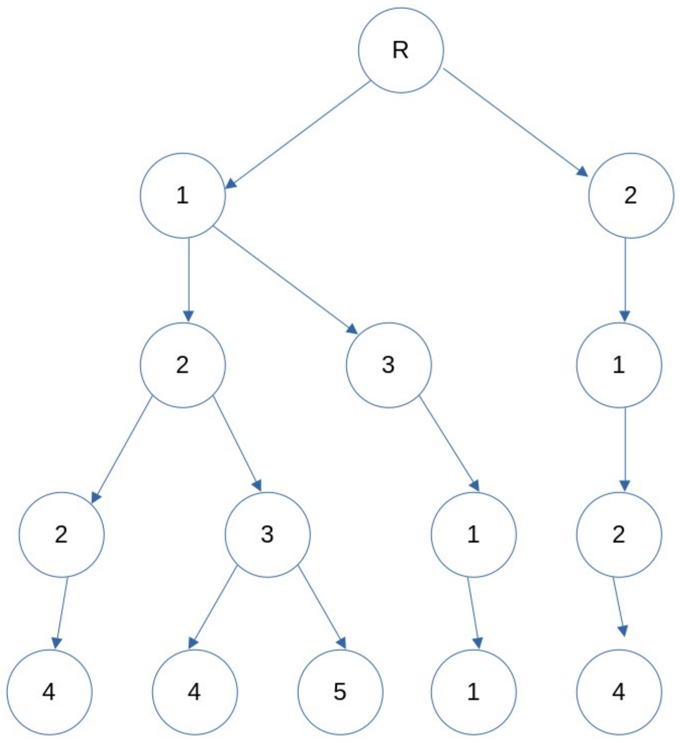
1235

1224

1311

2124

let's say we want to seek value 1311 in the whole rows of certain column. Even in linear time complexity can't afford to wait for web corpus processing that usually hold very large amount of data. Thus, index was invented. Index is simply a shortcut of fast lookup using tree based structure. Using the previous example. The tree that will be stored will looks like the following configuration.



This tree is formally termed as suffix tree, and covers any unicode symbols. Furthermore each node may hold more than two child nodes. Thus in term of structure it's more appropriate to call it, b-tree.

Your first task in problem B is to accommodate data structure for this purpose. Then verify the implementation through data insertion.

For the second task, after the tree has been initiated. Implement the following function

```
fn lookup(node: BTreeNodeLink, keys: Vec<i32>) → bool
```

The function will simply traverse from root seeking any match path that fit the keys digit tracing. If the value match until terminal, return true. Else false.

If your implementation is correct, b-tree utilization will net time complexity of O(lg n) instead of O(n).

For problem B, create btree.rs module under structure subdirectory.