Data Structures and Algorithms

A.Baskar

BITS Pilani, K. K. Birla Goa Campus

16 February 2018

Recap

- Non linear data structures: Trees
- Tree ADT
- Heap Sort

Tree Abstract Data Type

- Tree ADT stores elements at nodes
- element(v) returns the object stored at the node v O(1)
- size(), root(), parent(v)O(1)
- children(v) $O(c_v)$
- isInternal(v), isExternal(v), IsRoot(v) O(1)
- elements(), positions() O(n)
- swapElements(v,w), replaceElements(v,e) O(1)

Outline

- Dictionary ADT
- Binary Search Tree
- AVL Tree

Dictionary Abstract Data Type

- Store items (k,e)
- Search(k), insert (k,e), remove(k)
- size(), isEmpty()

Dictionary Abstract Data Type

- Store items (k,e)
- Search(k), insert (k,e), remove(k)
- size(), isEmpty()
- Ordered Dictionary ADT: maximum(), minimum(), successor(k), predecessor(k)

Implementations

- Logfile, Direct Address table, Hash table
- Either they don't support min, max, succ, pred operations
- Or they are not efficient

Implementations

- Logfile, Direct Address table, Hash table
- Either they don't support min, max, succ, pred operations
- Or they are not efficient
- Solution is to store data in a non-linear data structures
- Store such a way that it can support all these operations

Binary Search Tree

- Store items in the internal nodes of binary tree
- No items in the external nodes

Binary Search Tree

- Store items in the internal nodes of binary tree
- No items in the external nodes
- Key of a node is always less than every keys of the nodes in the left subtree
- Key of a node is always greater than every keys of the nodes in the right subtree

Binary Search Tree

- Store items in the internal nodes of binary tree
- No items in the external nodes
- Key of a node is always less than every keys of the nodes in the left subtree
- Key of a node is always greater than every keys of the nodes in the right subtree
- All keys are distinct

Performance

- search, insert, delete will take $O(h)$ time where h is height of the tree
- But in the worst case h can be $O(n)$

Performance

- search, insert, delete will take $O(h)$ time where h is height of the tree
- But in the worst case h can be $O(n)$
- The is because the tree might be unbalanced
- We have seen one solution: AVL trees

AVL Trees

- Balance the height between left subtree and right subtree
- Height of the tree is $O(\log n)$

AVL Trees

- Balance the height between left subtree and right subtree
- Height of the tree is $O(\log n)$
- All the operations can be done in $O(\log n)$

Height Balance Property

- For every internal node of a binary search tree, the heights of the children of v can differ by at most 1
- It is a property which characterizes the structure of BST

Height Balance Property

- For every internal node of a binary search tree, the heights of the children of v can differ by at most 1
- It is a property which characterizes the structure of BST
- Any BST which satisfies height balance property is an AVL Tree (Adelson-Velskii and Landis)

Height Balance Property

- For every internal node of a binary search tree, the heights of the children of v can differ by at most 1
- It is a property which characterizes the structure of BST
- Any BST which satisfies height balance property is an AVL Tree (Adelson-Velskii and Landis)
- Subtree of an AVL tree is an AVL tree
- Height of AVL tree stroing is n keys is $O(\log n)$