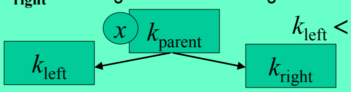
The University of Auckland

Binary Search Tree

- BST converts a static binary search into a dynamic binary search allowing to efficiently insert and delete data items
- Left-to-right ordering in a tree: for every node x, the values of all the keys k_{left} in the left subtree are smaller than the key k_{parent} in x and the values of all the keys k_{right} in the right subtree are larger than the key in x:

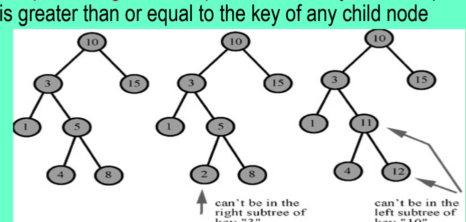
$k_{\text{left}} < k_{\text{parent}} < k_{\text{right}}$



The University of Auckland

Binary Search Tree

Compare the left–right ordering in a binary search tree to the bottom–up ordering in a heap where the key of each parent node is greater than or equal to the key of any child node



The University of Auckland

Binary Search Tree

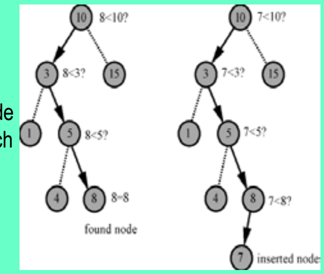
- No duplicates! (attach them all to a single item)
- Basic operations:
  - **find**: find a given search **key** or detect that it is not present in the tree
  - **insert**: insert a node with a given **key** to the tree if it is not found
  - **findMin**: find the minimum **key**
  - **findMax**: find the maximum **key**
  - **remove**: remove a node with a given **key** and restore the tree if necessary

The University of Auckland

BST: find / insert operations

find is the successful binary search

insert creates a new node at the point at which the unsuccessful search stops



The University of Auckland

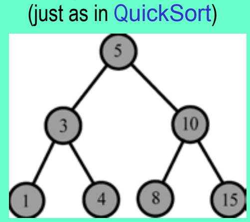
Binary Search Trees: findMin / findMax / sort

- Extremely simple: starting at the root, branch repeatedly left (**findMin**) or right (**findMax**) as long as a corresponding child exists
- The root of the tree plays a role of the pivot in quickSort
- As in QuickSort, the recursive traversal of the tree can sort the items:
  - First visit the left subtree
  - Then visit the root
  - Then visit the right subtree

The University of Auckland

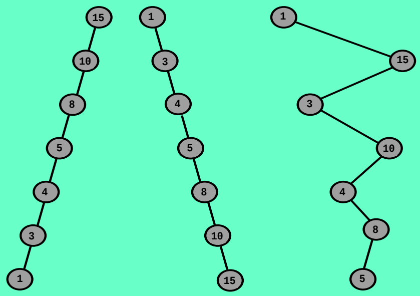
Binary Search Tree: running time

- Running time for **find**, **insert**, **findMin**, **findMax**, **sort**: $O(\log n)$ average-case and $O(n)$ worst-case complexity (just as in QuickSort)



BST of the depth about $\log n$

BST of the depth about n



Binary Search Tree: node removal

- **remove** is the most complex operation:
  - The removal may disconnect parts of the tree
  - The reattachment of the tree must maintain the binary search tree property
  - The reattachment should not make the tree unnecessarily deeper as the depth specifies the running time of the tree operations

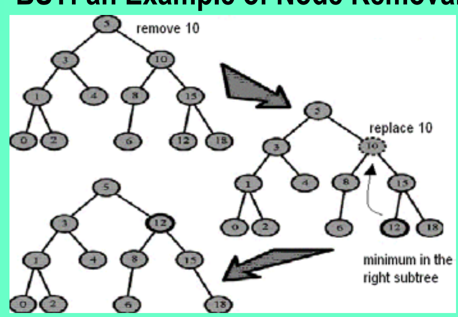
BST: how to remove a node

- If the node k to be removed is a leaf, delete it
- If the node k has only one child, remove it after linking its child to its parent node
- Thus, **removeMin** and **removeMax** are not complex because the affected nodes are either leaves or have only one child

BST: how to remove a node

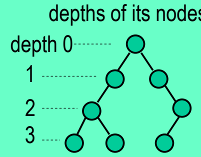
- If the node k to be removed has two children, then replace the item in this node with the item with the **smallest** key in the **right** subtree and remove this latter node from the right subtree (***Exercise:*** if possible, how can the nodes in the left subtree be used instead?)
- The second removal is very simple as the node with the smallest key does not have a left child
- The smallest node is easily found as in **findMin**

BST: an Example of Node Removal



Average-Case Performance of Binary Search Tree Operations

- **Internal path length** of a binary tree is the sum of the depths of its nodes:



IPL = 0 + 1 + 1 + 2 + 2 + 3 + 3 + 3
= 15

- **Average internal path length** $T(n)$ of the binary search trees with n nodes is $O(n \log n)$

Average-Case Performance of Binary Search Tree Operations

- If the n-node tree contains the root, the i-node left subtree, and the $(n–i–1)$-node right subtree, then:
  $$T(n) = n - 1 + T(i) + T(n–i–1)$$
  because the root contributes 1 to the path length of each of the other $n - 1$ nodes
- Averaging over all i; $0 \le i < n$: the same recurrence as for QuickSort:
  $$T(n) = (n-1) + \tfrac{2}{n}\big(T(0) + T(1) + \ldots + T(n-1)\big)$$
  so that $T(n)$ is $O(n \log n)$

Lecture 10 COMPSCI.220.FS.T - 2004 13

Average-Case Performance of Binary Search Tree Operations

- Therefore, the average complexity of **find** or **insert** operations is $T(n)/n = O(\log n)$
- For n^2 pairs of random **insert** / **remove** operations, an expected depth is $O(n^{0.5})$
- In practice, for random input, all operations are about $O(\log n)$ but the worst-case performance can be $O(n)$!

Lecture 10 COMPSCI.220.FS.T - 2004 14

Balanced Trees

- Balancing ensures that the internal path lengths are close to the optimal $n \log n$
- The average-case and the worst-case complexity is about $O(\log n)$ due to their balanced structure
- But, **insert** and **remove** operations take more time on average than for the standard binary search trees
  - **AVL** tree (1962: Adelson-Velskii, Landis)
  - **Red-black** and **AA-tree**
  - **B-tree** (1972: Bayer, McCreight)

Lecture 10 COMPSCI.220.FS.T - 2004 15

AVL Tree

- An AVL tree is a binary search tree with the following additional balance property:
  - for any node in the tree, the height of the left and right subtrees can differ by at most 1
  - the height of an empty subtree is -1
- The AVL-balance guarantees that the AVL tree of height h has at least c^h nodes, $c > 1$, and the maximum depth of an n-item tree is about $\log_c n$

Lecture 10 COMPSCI.220.FS.T - 2004 16

AVL Tree

- Let S_h be the size of the smallest AVL tree of the height h (it is obvious that $S_0 = 1$, $S_1 = 2$)
- This tree has two subtrees of the height $h–1$ and $h–2$, respectively, by the balance condition
- It follows that $S_h = S_{h-1} + S_{h-2} + 1$, or $S_h = F_{h+3} - 1$ where F_i is the i-th Fibonacci number

Lecture 10 COMPSCI.220.FS.T - 2004 17

AVL Tree

- Therefore, for each n-node AVL tree:
  $$n \ge S_h \approx \left(\varphi^{h+3}/\sqrt{5}\right) - 1$$
  where $\varphi = \left(1 + \sqrt{5}\right)/2 \cong 1.618$, or
  $$h \le 1.44 \log_2(n+1) - 1.328$$
- Thus, the worst-case height is at most 44% more than the minimum height of the binary trees

Lecture 10 COMPSCI.220.FS.T - 2004 18