CSCI-UA 480.4: APS
Algorithmic Problem Solving

Non-Linear Data Structures

Instructor: Joanna Klukowska

created based on materials for this class by
Bowen Yu and materials shared by the authors of
the textbook Steven and Felix Halim

Binary Trees

Balanced Binary Search Tree

- binary tree
- for each node
  - all the values in the left subtree are smaller than the value in that node
  - all the values in the right subtree are greater than the value in that node
  - (equal values should be stored in one of the subtrees, if allowed)

Balanced Binary Search Tree

- binary tree
- for each node
  - all the values in the left subtree are smaller than the value in that node
  - all the values in the right subtree are greater than the value in that node
  - (equal values should be stored in one of the subtrees, if allowed)
- / / in C++ and / in Java
  - the *map* versions of the classes store pairs
  - the *set* versions of the classes store only keys
  - the (only in C++) allows duplicate keys

Balanced Binary Search Tree

- binary tree
- for each node
  - all the values in the left subtree are smaller than the value in that node
  - all the values in the right subtree are greater than the value in that node
  - (equal values should be stored in one of the subtrees, if allowed)
- / / in C++ and / in Java
  - the *map* versions of the classes store pairs
  - the *set* versions of the classes store only keys
  - the (only in C++) allows duplicate keys
- on
  - add key,
  - remove key,
  - find key ,
  - find min/max,
  - successor/predecessor key

Balanced Binary Search Tree

- binary tree
- for each node
  - all the values in the left subtree are smaller than the value in that node
  - all the values in the right subtree are greater than the value in that node
  - (equal values should be stored in one of the subtrees, if allowed)
- / / in C++ and / in Java
  - the *map* versions of the classes store pairs
  - the *set* versions of the classes store only keys
  - the (only in C++) allows duplicate keys
- on
  - add key,
  - remove key,
  - find key ,
  - find min/max,
  - successor/predecessor key
- AVL tree, Red-Black tree

Binary Heap / Priority Queue

- a complete binary tree (all levels are filled except for the lowest one, the lowest one is filled from left to right)
- for each node the values of its children are
    - smaller than or equal to the one in the node (max-heap)
    - larger than or equal to the one in the node (min-heap)

Binary Heap / Priority Queue

- a complete binary tree (all levels are filled except for the lowest one, the lowest one is filled from left to right)
- for each node the values of its children are
  - smaller than or equal to the one in the node (max-heap)
  - larger than or equal to the one in the node (min-heap)
- in C++, in Java
- usually implemented using compact arrays

Binary Heap / Priority Queue

- a complete binary tree (all levels are filled except for the lowest one, the lowest one is filled from left to right)
- for each node the values of its children are
  - smaller than or equal to the one in the node (max-heap)
  - larger than or equal to the one in the node (min-heap)
- in C++, in Java
- usually implemented using compact arrays
- operations :
  - add
  - remove

  because they require traversal along a single path from root to leaf
- *top* should be

Binary Heap / Priority Queue

- a complete binary tree (all levels are filled except for the lowest one, the lowest one is filled from left to right)
- for each node the values of its children are
  - smaller than or equal to the one in the node (max-heap)
  - larger than or equal to the one in the node (min-heap)
- in C++, in Java
- usually implemented using compact arrays
- operations :
  - add
  - remove

  because they require traversal along a single path from root to leaf
- *top* should be
- building a heap from a collection of values is !!!

  Why?

Hash Table

- mapping from *keys* to *values* not sorted in any particular/predictable way
- lookups by key can be done in (assuming a good hash function, but worst case)
- addition / removal done in (but worst case)

Hash Table

- mapping from *keys* to *values* not sorted in any particular/predictable way
- lookups by key can be done in (assuming a good hash function, but worst case)
- addition / removal done in (but worst case)
- _____ in C++ () - starting in C++11 standard
- , , in Java

Hash Table

- mapping from *keys* to *values* not sorted in any particular/predictable way
- lookups by key can be done in (assuming a good hash function, but worst case)
- addition / removal done in (but worst case)
- _____ in C++ () - starting in C++11 standard
- , , in Java
- challenge: depends on a well designed hash function (and that is often tricky)

Hash Table

- mapping from *keys* to *values* not sorted in any particular/predictable way
- lookups by key can be done in (assuming a good hash function, but worst case)
- addition / removal done in (but worst case)
- _____ in C++ () - starting in C++11 standard
- , , in Java
- challenge: depends on a well designed hash function (and that is often tricky)
- Direct Addressing Table (DAT) is the simplest form of a hash table in which *keys* are the indexes - it is simply an array

Challenge

For each o the following problems 1) come up with tests that could be used for the implementation, 2) come up with an algorithm(s) that can be used to solve the problem, 3) what is the performance of your algorithms?

- Given integers in arbitrary order find largest ones.
- Given a root to a binary tree containing nodes, determine if it is a binary search tree?
- Given a binary search tree containing nodes, output the elements with values in the range [a..b] in ascending order.
- Given a binary search tree containing nodes, output the values in the leaves in ascending order.
- A basic max-heap does only supports additions and removal of the max element. How would you implement the following two operations:
  - - modify the value at a specified index to the new_value
  - - deletes the value at a specified index