Lab06-Heaps and BST

VE281 - Data Structures and Algorithms, Xiaofeng Gao, TA: Li Ma, Autumn 2019

∗ Please upload your assignment to website. Contact webmaster for any questions.

∗ Name: Sun Yiwen Student ID: 517370910213 Email: sunyw99@sjtu.edu.cn

1. **D-ary Heap.** D-ary heap is similar to binary heap, but (with one possible exception) each non-leaf node of d-ary heap has d children, not just 2 children.

   (a) How to represent a d-ary heap in an array?

   (b) What is the height of the d-ary heap with n elements? Please use n and d to show.

   (c) Please give the implementation of insertion on the min heap of d-ary heap, and show the time complexity with n and d.

```cpp
#ifndef D_ARY_HEAP_H
#define D_ARY_HEAP_H

#include <functional>
#include <vector>
#include <algorithm>

class d_ary_heap {
public:
    d_ary_heap(int d);
    ~d_ary_heap() {}
    void enqueue(int k);
private:
    int d;
    std::vector<int> data;
    void swap(int& a, int& b);
    void perculateUp(int id);
};

d_ary_heap::d_ary_heap(int d) :d(d)
{
}

void d_ary_heap::swap(int& a, int& b) {
    int temp;
    temp = a;
    a = b;
    b = temp;
}

void d_ary_heap::perculateUp(int id) {
    while (id > 0 && data[id] < data[(id - 1) / d]) {
        swap(data[(id - 1) / d], data[id]);
        id = (id - 1) / d;
    }
}
```

```cpp

void d_ary_heap::enqueue(int k) {
    data.push_back(k);
    perculateUp(data.size() - 1);
}

#endif//D_ARY_HEAP_H
```

Solution. (a) Same as binary heap implementation as an array, we can store the elements of a d-ary heap in an array in the order produced by a level order traversal.

(b) The height of a d-ary heap with n elements is $\lceil \log_d(n+1) \rceil - 1$ or $\lfloor \log_d n \rfloor$.

(c) The time complexity is $O(\log_d n)$.

□

2. **Median Maintenance.** Input a sequence of numbers $x_1, x_2..., x_n$, one-by-one. At each time step i, output the median of $x_1, x_2..., x_i$. How to do this with $O(\log i)$ time at each step i? Show the implementation.

```cpp
#ifndef BINARY_HEAP_H
#define BINARY_HEAP_H

#include <functional>
#include <vector>
#include <algorithm>

// OVERVIEW: A specialized version of the 'heap' ADT implemented as
    a binary
//            heap.
template<typename TYPE, typename COMP = std::less<TYPE> >
class binary_heap {
public:
    typedef unsigned size_type;
    ~binary_heap() {}
    binary_heap(COMP comp = COMP());
    void enqueue(const TYPE& val);
    TYPE dequeue_min();
    const TYPE& get_min() const;
    size_type size() const;
    bool empty() const;
private:
    std::vector<TYPE> data;
    COMP compare;
private:
    void swap(TYPE& a, TYPE& b);
    void perculateUp(int id);
    void perculateDown(int id);
};

template<typename TYPE, typename COMP>
```

```
void binary_heap<TYPE, COMP>::swap(TYPE& a, TYPE& b) {
    TYPE temp;
    temp = a;
    a = b;
    b = temp;
}

template<typename TYPE, typename COMP>
void binary_heap<TYPE, COMP>::perculateUp(int id) {
    while (id > 0 && compare(this->data[id], this->data[(id + 1) / 2 - 1])) {
        swap(this->data[(id + 1) / 2 - 1], this->data[id]);
        id = (id + 1) / 2 - 1;
    }
}

template<typename TYPE, typename COMP>
void binary_heap<TYPE, COMP>::perculateDown(int id) {
    unsigned int j;
    for (j = 2 * (id + 1) - 1; j < this->data.size(); j = 2 * (id + 1) - 1) {
        if (j < this->data.size() - 1 && compare(this->data[j + 1], this->data[j])) j++;
        if (!compare(this->data[j], this->data[id])) break;
        swap(this->data[id], this->data[j]);
        id = j;
    }
}

template<typename TYPE, typename COMP>
binary_heap<TYPE, COMP> ::binary_heap(COMP comp) {
    compare = comp;
}

template<typename TYPE, typename COMP>
void binary_heap<TYPE, COMP> ::enqueue(const TYPE& val) {
    this->data.push_back(val);
    perculateUp(this->data.size() - 1);
}

template<typename TYPE, typename COMP>
TYPE binary_heap<TYPE, COMP> ::dequeue_min() {
    TYPE min;
    min = this->data[0];
    swap(this->data[0], this->data[this->data.size() - 1]);
    this->data.pop_back();
    perculateDown(0);
    return min;
}

```

```cpp
template<typename TYPE, typename COMP>
const TYPE& binary_heap<TYPE, COMP>::get_min() const {
    return this->data[0];
}

template<typename TYPE, typename COMP>
bool binary_heap<TYPE, COMP>::empty() const {
    return this->data.empty();
}

template<typename TYPE, typename COMP>
unsigned binary_heap<TYPE, COMP>::size() const {
    return this->data.size();
}

#endif //BINARY_HEAP_H
```

```cpp
#include <iostream>
#include <string>
#include "binary_heap.h"
using namespace std;

struct compare_t
{
    bool operator()(int a, int b) const
    {
        return a > b;
    }
};

int main() {
    string input;
    int newInt, count, current_mean;
    float new_mean;
    binary_heap<int, compare_t>* data1 = new binary_heap<int, compare_t>;
    binary_heap<int>* data2 = new binary_heap<int>;
    cin >> input;
    newInt = stoi(input);
    data1->enqueue(newInt);
    count = 1;
    cout << "mean: " << newInt << " " << endl;
    while (cin >> input) {
        if (input == "q") break;
        newInt = stoi(input);
        if (count % 2 == 1) {
            current_mean = data1->dequeue_min();
            if (newInt < current_mean) {
                data1->enqueue(newInt);
                data2->enqueue(current_mean);
```

```
                new_mean = ((float)current_mean + (float)data1->
                    get_min()) / 2;
            }
            else {
                data2->enqueue(newInt);
                data1->enqueue(current_mean);
                new_mean = ((float)data1->get_min() + (float)data2
                    ->get_min()) / 2;
            }
        }
        else {
            current_mean = data2->dequeue_min();
            if (newInt < current_mean) {
                data1->enqueue(newInt);
                data2->enqueue(current_mean);
                new_mean = data1->get_min();
            }
            else {
                data2->enqueue(newInt);
                data1->enqueue(current_mean);
                new_mean = data1->get_min();
            }
        }
        count++;
        cout << "mean: " << new_mean << " " << endl;
    }
    return 0;
}
```

3. **BST**. Two elements of a binary search tree are swapped by mistake. Recover the tree without changing its structure. Implement with a constant space.

```
/**
 * Definition for binary tree
 * struct TreeNode {
 *     int val;
 *     TreeNode *left;
 *     TreeNode *right;
 *     TreeNode(int x) : val(x), left(NULL), right(NULL) {}
 * };
 */

void inorder(TreeNode* root, TreeNode*& pre, TreeNode*& first,
    TreeNode*& second) {
    if (!root) return;
    inorder(root->left, pre, first, second);
    if (!pre) pre = root;
    else {
        if (pre->val > root->val) {
            if (!first) first = pre;
```

```
            second = root;
        }
        pre = root;
    }
    inorder(root->right, pre, first, second);
}

void recoverTree(TreeNode *root)
{
    TreeNode* pre = NULL, * first = NULL, * second = NULL;
    int temp;
    inorder(root, pre, first, second);
    temp = first->val;
    first->val = second->val;
    second->val = temp;
}
```

4. **BST**. Input an integer array, then determine whether the array is the result of the post-order traversal of a binary search tree. If yes, return Yes; otherwise, return No. Suppose that any two numbers of the input array are different from each other. Show the implementation.

```
// Input: an integer array
// Output: yes or no
bool verifySquenceOfBST(vector<int> sequence)
{
    int root, i, j;
    vector<int> temp;
    vector<int> right;
    if (sequence.size() == 0 || sequence.size() == 1) return true;
    root = sequence[sequence.size() - 1];
    sequence.pop_back();
    i = sequence.size() - 1;
    while (i >= 0 && sequence[i] > root) {
        temp.push_back(sequence[i]);
        sequence.pop_back();
        i--;
    }
    for (j = temp.size() - 1; j >= 0; j--) right.push_back(temp[j]);
    while (i >= 0 && sequence[i] < root) i--;
    if (i != -1) return false;
    else return (verifySquenceOfBST(sequence) && verifySquenceOfBST(right));
}
```