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: Jintian Ge Student ID: 517021911142 Email: gejintian@sjtu.edu.cn

1. **D-ary Heap.** D-ary heap is similar to binary heapbut (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.

```
// Input: an integer k
// Output: null

void percolateUp(int id){
    while(id > 0 && heap[id] < heap[(id-1)/d]){
        int temp = heap[id];
        heap[id] = heap[(id-1)/d];
        heap[(id-1)/d] = temp;
        id = (id - 1)/d;
    }
}

void enqueue(int k)
{
    heap[size] = k;
    percolateUp(size);
    size ++;
}
```

Solution. (a) Considering the index starting from zero, if the node is in the position n, then its first child is in the position $(n-1)\times d+2$, and its last child is in the position $n\times d+1$. If we consider that the index of the array starts from 0, then for a node with position n, its first child should be in the position $n\times d+1$, its last child should be in the position $(n+1)\times d$.

(b) Assume it is a perfect tree. Then, its height should have the relation:

$$\frac{1-d^{(h+1)}}{1-d}=n$$

Solving this equation, we get:

$$h=\log_d[(d-1)\times n+1]$$

Since the tree is complete but may not be perfect, its height could be found by apply a ceil calculator:

$$h=\lceil\log_d[(d-1)\times n+1]\rceil$$

(c) In the worst case, the inserted number has the smallest key value. In this case, we will need to check each node until the root. So, the 'while-loop' will execute for about $\lceil \log_d[(d-1)\times n+1]\rceil$ times. Each time the 'while-loop' is executed, $O(1)$ times is needed to swap and compare. So, totally the time complexity of the enqueue algorithm should be:

$$T(n) = \lceil \log_d[(d-1)\times n+1]\rceil \times O(n) = O(\log n)$$

□

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.

Solution. Using two binary heaps. One for Max and one for Min. Keeping all elements in Max smaller than those in Min. The implementation is shown below: □

```
#include <iostream>
#include <vector>
using namespace std;

struct compare_t
{
    bool operator()(int a, int b) const
    {
        return a > b;
    }
};
template<typename TYPE, typename COMP = std::less<TYPE> >
class binary_heap{
public:
    typedef unsigned size_type;

    binary_heap(COMP comp = COMP());

    void enqueue(const TYPE &val);

    TYPE dequeue_min();

    const TYPE &get_min() const;

    virtual size_type size() const;

    virtual bool empty() const;

private:
    std::vector<TYPE> data;
    COMP compare;

private:
```

```cpp
    void percolateUp(int id);
    void swap(TYPE &a, TYPE &b);
    void percolateDown(int id);
};

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>::swap(TYPE &a, TYPE &b){
    TYPE temp;
    temp = a;
    a = b;
    b = temp;
}

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

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

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

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

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

template<typename TYPE, typename COMP>
bool binary_heap<TYPE, COMP> :: empty() const {
    if(data.size() == 0) return true;
    else return false;
}

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

int main() {
    binary_heap<float,compare_t> MAX_HEAP;
    binary_heap<float> MIN_HEAP;
    vector<float> data;
    int count = 0;
    while(true){
        int number;
        cout<<"Please input one number: "<<endl;
        cin>>number;
        if(count == 0){
            MAX_HEAP.enqueue(number);
            cout<<"The median is: "<<MAX_HEAP.get_min()<<endl;
        }
        else if(count%2 == 0){
            //count is even.
            if(number <= MIN_HEAP.get_min()) MAX_HEAP.enqueue(
              number);
            else {
                MAX_HEAP.enqueue(MIN_HEAP.dequeue_min());
                MIN_HEAP.enqueue(number);
            }
            cout<<"The median is: "<<MAX_HEAP.get_min()<<endl;
        }
        else{
            //count is odd
            if(number >= MAX_HEAP.get_min()) MIN_HEAP.enqueue(
              number);
            else{
                MIN_HEAP.enqueue(MAX_HEAP.dequeue_min());
                MAX_HEAP.enqueue(number);
            }
            cout<<"The median is : "<<(MAX_HEAP.get_min()+MIN_HEAP.get_min())/2<<endl;
        }
        count++;
    }
    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 recoverTree(TreeNode *root)
{
    TreeNode *first = nullptr;
    TreeNode *second = nullptr;
    TreeNode *current_pointer = root;
    TreeNode *last_pointer = nullptr;
    while(current_pointer != nullptr){
        if(current_pointer->left != nullptr){
            TreeNode *temp = current_pointer->left;
            while(temp->right != nullptr && temp->right != current_pointer) temp = temp->right;
            if(temp->right == nullptr){
                temp->right = current_pointer;//Connect right child to current_pointer
                current_pointer = current_pointer->left;//Go through all the nodes on the left side
                continue;
            }
            else temp->right = nullptr;
        }
        if(last_pointer != nullptr){
            if(last_pointer->val > current_pointer->val){
                if(first == nullptr) first = last_pointer;
                second = current_pointer;
            }
        }
        last_pointer = current_pointer;
        current_pointer = current_pointer->right;
    }
    //Swap first and second
    int temp = first->val;
    first->val = second->val;
    second->val = temp;
}
```

Solution. Basic idea is that we need to go through the BST without calling stacks. This could be achieved by a method found from internet called 'Morris Algorithm'. The basic idea of this Algorithm is that: for each node N in the tree, find the right most node R in its left subtree, and since R is the right most node, its right child must point to $NULL$. Then, connect its right child to N. In this case, when the pointer reached the most right child, it will come back to the middle node above. With this methods, we only need to compare the value of current node and last node, then we will know whether it is in-order. □

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)
{
    if(sequence.size() == 1 || sequence.size() == 0) return true;
    bool if_left = true;
    vector<int>::iterator iterator_root = sequence.end();
    iterator_root --;
    int root = *iterator_root;
    iterator_root --;
    vector<int>::iterator iterator_right = iterator_root;
    vector<int>::iterator iterator_left;
    for(iterator_left = iterator_right; ; iterator_left --){
        if(*iterator_left < root) break;
        if(iterator_left == sequence.begin()) {//No left subtree
            if_left = false;
            break;
        }
    }
    if(if_left){
        for(vector<int>::iterator check = iterator_left;;check--){
            if(*check > root){
                cout<<"Root is "<<root<<" CHECK is "<<*check;
                return false;
            }
            if(check == sequence.begin()) break;
        }
        iterator_left ++;
        vector<int>left_sub(sequence.begin(),iterator_left);
        iterator_right++;
        vector<int>right_sub(iterator_left ,iterator_right);
        return verifySquenceOfST(left_sub) && verifySquenceOfST(
           right_sub);
    }
    else{
        iterator_right++;
        vector<int>right_sub(sequence.begin(),iterator_right);
    }
}
```

Solution. Since in post-order traversal, the end of the vector should be the root node. The element next to the end should be the right child of the root node. And the first element which is less than the root should be the left child of the root node. In this case, between the left child and the right child is all elements in the right subtree and the rest is all the elements in the left subtree. All left subtree and right subtree can be distinguished by this method. We only need to make sure that all elements in the left subtree is smaller than the root and all elements in the right subtree is greater than the root and using recursive function, then we are finished. Implementation is shown above. □