Practice 4. Building Binary Search Trees

[CSE2010] Data Structures

Department of Data Science

Practice 3. Queue

- Queue implementation with a linked list.
- First, implement a node.
  - You can use the node implementation in the previous practice.

```
// Practice 3. Queue
#include <cstdlib>
#include <iostream>
#include <fstream>
#include <sstream>
#include <vector>
using namespace std;
const char ENQUEUE = 'E';
const char DEQUEUE = 'D';
const char FRONT = 'F';
const char PUSH = 'U';
const char POP = 'O';
const char TOP = 'T';

class Node {
public:
  int data;
  Node* next;
  Node(int d): data(d), next(nullptr) {}
};
```

Practice 3. Queue

- Queue has head and tail as member variables.
- Constructor and destructor for C++.
- Python implementation would be simpler.
  - You don’t need a destructor in python.

```
class Queue {
  Node* head;
  Node* tail;
public:
  Queue() {
    head = nullptr;
    tail = nullptr;
  }
  ~Queue() {
    Node* currNode = head;
    Node* nextNode = nullptr;
    while (currNode) {
      nextNode = currNode->next;
      delete currNode;
      currNode = nextNode;
    }
  }
  // true if the queue is empty; false otherwise
  bool isEmpty() {
    return head == nullptr;
  }
```

Practice 3. Queue

- Enqueue
  - Create a new node with value d
  - Since this implementation uses a linked list, a new element can be always enqueued, unlike the array implementation.
- Dequeue
  - If a queue is empty, terminate the program with the error message.
  - Otherwise, return the data of head, and update head to head->next.
- Front
  - If a queue is empty, terminate the program with the error message.
  - Otherwise, return head->data.

```
// Enqueue an element to the queue
bool enqueue(int d) {
  Node* newNode = new Node(d);
  if (isEmpty()) {
    head = tail = newNode;
  }
  else {
    tail->next = newNode;
    tail = newNode;
  }
  return true;
}

// Dequeue an element from the queue and return the its value
int dequeue() {
  if (head == nullptr) {
    cout<<"Queue has no element to dequeue"<<endl;
    exit(1);
  }
  int item = head->data;
  head = head->next;
  if (head == nullptr)
    tail = nullptr;
  return item;
}

// Get the front of the queue
int front() {
  if (head == nullptr) {
    cout<<"Queue has no element"<<endl;
    exit(1);
  }
  return head->data;
}
```

Practice 3. StackViaQueues

- A stack can be implemented by using two queue instances.
  - Push: **O(n)** time
  - Pop: O(1) time
  - Peek: O(1) time

```
class StackViaQueues {
public:
  Queue* mainQueue;
  Queue* subQueue;
  StackViaQueues() {
    mainQueue = new Queue();
    subQueue = new Queue();
  }
  ~StackViaQueues() {
    delete mainQueue;
    delete subQueue;
  }
```

```
  bool push(int d) {
    subQueue->enqueue(d);
    while(!mainQueue->isEmpty()) {
      subQueue->enqueue(mainQueue->dequeue());
    }
    Queue* temp = subQueue;
    subQueue = mainQueue;
    mainQueue = temp;
    return true;
  }

  int pop() {
    return mainQueue->dequeue();
  }

  int peek() {
    return mainQueue->front();
  }

  void write(ofstream& outFile) {
    mainQueue->writeReverse(outFile);
  }
};
```

Overview

- Implement a **binary search tree**.
- Functions (where the binary search tree has n nodes)
    1. Given a sequence of integers sorted in the ascending order, build a binary search tree: O(n) time
    2. Find the minimum in the tree: O(logn) time
    3. Find the maximum in the tree: O(logn) time

Input of BST

- Each line represents a single operation.
  1. **B<space>([int]+)**
     B<space>[int]<space>[int]<space>…<space>[int]
     - If a sequence of input integers is not sorted in the ascending order, immediately terminate the program with the error message.
     - Otherwise, build a binary search tree that contains these integers as keys, and **write “B” into output file.**
  2. **m**
     - If finding the minimum fails, immediately terminate the program with the error message.
     - Otherwise, **write the minimum into output file.**
  3. **M**
     - If finding the maximum fails, immediately terminate the program with the error message.
     - Otherwise, **write the maximum into output file.**

Input and Output

- Start from the scratch by using the File I/O practices, or from on the skeleton code.
- Each line represents to the result of the corresponding line of the input file.
- Input File & Output File

Input File	Output File
B 1 2 3 4 5 6 7 8 9 **m** **M**	**B** **1** **9**

```
[hjkim@localhost bst]$ ./practice4 input3.txt output3.txt
                     5

           2                7

     1        3        6        8

                4                   9

[hjkim@localhost bst]$ cat input3.txt
B 1 2 3 4 5 6 7 8 9
m
M
[hjkim@localhost bst]$ cat output3.txt
B
1
9
[hjkim@localhost bst]$
```

- Input File & Output File

Input File	Output File
B 1 2 3 5 4 6 7 8 9 **m** **M**	

```
[hjkim@localhost bst]$ ./practice4 input4.txt output4.txt
BUILD: invalid input
[hjkim@localhost bst]$ cat input4.txt
B 1 2 3 5 4 6 7 8 9
m
M
[hjkim@localhost bst]$ cat output4.txt
[hjkim@localhost bst]$
```

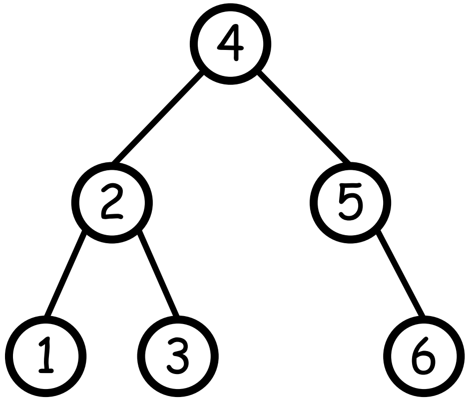
Hints

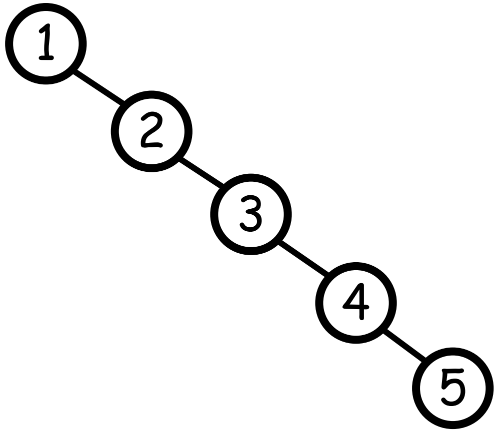
- First, implement a node, which will be a building block of your tree.

```
class TreeNode {
public:
  int key;
  TreeNode* left;
  TreeNode* right;
  TreeNode(int k, TreeNode* l=nullptr, TreeNode* r=nullptr) {
    key = k;
    left = l;
    right = r;
  }
};
```

```
class TreeNode:
  def __init__(self, k, l=None, r=None):
    self.key = k
    self.left = l
    self.right = r
```

- Binary search tree does not need to be complete, or balanced.





Submission Guideline

- Submission: **source code, makefile**
  - Where: Practice4 submission page in LMS
  - Deadline: **23:59, April. 3th (Sunday)**
- Extra points
  - **From April 4th (Monday)**
  - Share your **code, input & output** on Open Board in LMS.
  - Review classmates' code. Give questions or comments on his/her post.
  - Answer others' questions on your post.
  - Title: [Practice4]StudentID
    - e.g., [Practice4]2021000000

Next Practice

- Overview
    - **April 6th (Wednesday)**
    - Traversing a tree with different traversal methods
        - Preorder traversal
        - Inorder traversal
        - Postorder traversal