A6: Tree Structures

Due: Sunday, 4/30/2017 01:59am
(this is the usual Saturday night/Sunday morning deadline)

Submit your solutions to all non-coding portions of the assignment as A6.pdf to **A6 Written** on Autolab. Late submissions will receive a 20 point per day penalty (you may submit up to 2 days late). Make sure you submit all 3 parts to their respective places:

- `A6.pdf` submitted to A6 Written: contains the written portion of your work.
- `A6-comp.hpp` submitted to A6 Compare: contains the code to compare two numbers by the number of bits (your solution to 1(g)).
- `A6-Jobs.hpp` submitted to A6 JobQueue: contains the code for the Job and JobQueue classes (your solution to 3).

Total points for the assignment: 100 points.

1. ($4 + 4 + 4 + 4 + 6 + 4 + 10 + 4$ points) Review the code provided in `BST.tar` and answer the following questions:

   1.(a) Consider the following object:

   ```
   BST<CompareGT> bstGT;
   ```

   Draw the tree resulting from calling `bstGT.insert(`i`);` where i ranges over (and inserted in this order) the numbers:

   $$10, 0, 30, 20, 50, 5, 3.$$

   1.(b) Now draw the tree after calling `bstGT.remove(10);`

   1.(c) Now draw the tree after calling `bstGT.remove(30);`

   1.(d) What is the worst case runtime for `remove(`i`)` on a tree of size n? What factor determines the runtime of `remove(`i`)`?

   1.(e) Now consider the following object:

   ```
   BST<CompareLT> bstLT;
   ```

Provide a list of 10 numbers that, when inserted in the given order to `bstLT`, produces to the worst case runtime when calling `remove(`i`)` where i is the root of the tree. Draw the tree resulting from inserting these 10 numbers and then draw the tree resulting from calling remove on the root.

1.(f) Re-order the previously given 10 numbers so they produce a tree where the maximum depth is 3 when inserted in to the following tree:

```
BST<CompareLT> bstLT2;
```

Draw the tree resulting from inserting these 10 numbers in this new order.

1.(g) Write the code for a function object named `CompareBits` that overloads `operator()` such that:

- The method signature is:

  ```
  bool operator() (const int& lhs, const int& rhs)
  ```

- Given an object `CompareBits comp;`
  - `comp(x,y)` returns `true` if the number of bits that are 1 in x is less than the number of bits that are 1 in y.
  - `comp(x,y)` returns `false` if the number of bits that are 1 in x is greater than or equal to the number of bits that are 1 in y.

Submit your code for `class CompareBits` in a file named `A6-comp.hpp` to **A6 Compare** on Autolab. Note that this code will not be graded until after the deadline has passed.

1.(h) Consider the following object:

```
BST<CompareBits> bstBits;
```

Draw the tree resulting from calling `bstBits.insert(`i`);` where i ranges over the numbers (in this order):

$$10, 0, 30, 20, 50, 5, 3.$$

2. (4 + 2 + 4 + 10 points) Consider a tree structure where each node is of type `QuadTreeNode`, which holds a pointer to 4 child nodes.

```
class QuadTreeNode {
public:
  int value;
  QuadTreeNode* child[4];
};
```

2.(a) How many nodes would be in a full tree of QuadTreeNodes with a depth of 1? What about a depth of 2?

2.(b) Provide a general formula for the number of nodes at depth i for a full tree made of QuadTreeNodes.

2.(c) Provide a general formula for the total number of nodes in a full tree made of QuadTreeNodes with max depth i.

2.(d) Update the `operator++` for the TreeIterator code (provided in lecture) to iterate through a tree made of QuadTreeNodes. You can access the child nodes as `node->child[0]`, ..., `node->child[3]` corresponding to the children from left to right.

Clarifications:

- You may assume the constructor works to push the leftmost nodes onto the stack. If you wish to change how this works, provide code for the updated constructor.
- Traverse in post-order.
- Traverse children in order from left to right
  (i.e., visit `node->child[i]` before `node->child[j]` if $i < j$).
- If `node->child[i] == nullptr` you may assume `node->[j] == nullptr` for $i \leq j \leq 3$.

3. (5 + 35 points) Suppose you have to schedule jobs for an operating system. Jobs consist of a priority and a PID, the program ID. Your task is to write the code to store and retrieve the Job object with the highest priority that still needs to be run. You may store them any way you wish. Priority between Jobs is determined as follows:

   - Each job is assigned a priority between 0 and 255.
   - A priority of 0 is highest priority and a priority of 255 is the lowest priority.
   - Given two jobs with the same priority number, the Job with the smaller PID has higher priority.

3.(a) Create `class Job` that stores a PID in a public member variable `_pid` and stores a priority in a public member variable `_priority`.

- `_pid` will be a number between 0 and 2,147,483,647.
- `_priority` will be a number between 0 and 255.

Create a constructor that can be invoked as `Job(pid, priority)` that initializes the appropriate fields.

3.(b) Create `class JobQueue` that implements the following methods:

- `push`: stores a Job in the queue.
- `top`: returns copy of the job with the highest priority.
- `pop`: removes the job with the highest priority from the queue.

You may use any standard libraries you wish. Submit your code for your `class Job` and `JobQueue` in a single file `A6-Jobs.hpp` to **A6 JobQueue** on Autolab.