Programming Problem: Heap Tunable Branching Factor

[12 points for the program, 6 points for the analysis and performance report.]

For this problem, you get to implement your own min-ordered heap, with functions / methods for two of the typical heap operations, removeMin() and insertValue(). Your heap is going to be different, though. Instead of being restricted to just (up to) two children for each heap node, your heap will let the user specify how many children each node can have. This can be any power of two 2^p for $p \geq 1$. With a branching factor that's a power of 2, we can write efficient code for traversing the tree structure of the heap, using bit shifting rather than multiplication and division.

Implementation Language

As with the last assignment, you get to implement your solution in Java, C or C++, whichever language you prefer. The name of your source file will depend on your language:

- heap.java (for java)
- heap.c (for C)
- heap.cpp (for C++)

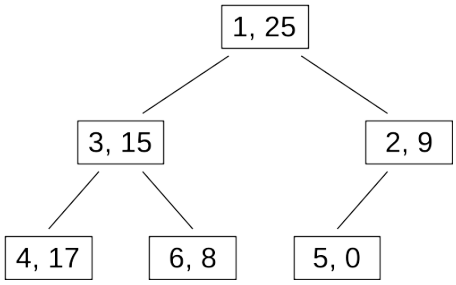
When we test your solutions, we'll compile and run them on an EOS Linux machine using the following commands. Be sure to try these out yourself to make sure you program will compile and work when we try it out. Notice that the program is getting the branching factor for the heap as a command-line argument, but it's reading a sequence of heap operations from standard input. Here, we're telling the heap to use a branching factor of 8 and we're reading input from test input input-4.txt. Of course these commands should work with any valid input file and for any power of 2 as the branching factor.

- For Java
  ```
  javac heap.java
  java heap 8 < input-4.txt
  ```
- For C
  ```
  gcc -Wall -std=c99 -g -O2 -o heap heap.c
  ./heap 8 < input-4.txt
  ```
- For C++
  ```
  g++ -Wall -g -O2 -o heap heap.cpp
  ./heap 8 < input-4.txt
  ```

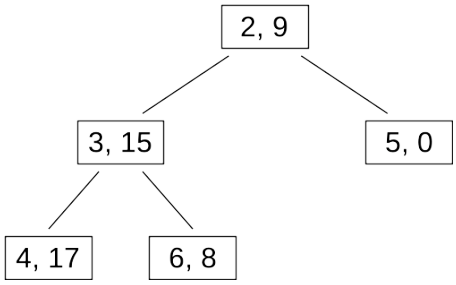
Heap Implementation

Each element of your heap will store a pair of integer values. The first value in each pair will be a key that determines the heap ordering. The second will be an associated value that goes with that key. This is typical of how you might use a heap in practice. The key lets you efficiently choose the minimum (or maximum) among a set of elements, but you often need some other data value that goes with this minimum. For example, if you're using a heap to select the minimum-cost vertex in a graph, then each heap element might need to store a cost (the thing you're minimizing) and the index of the node that goes with that cost.

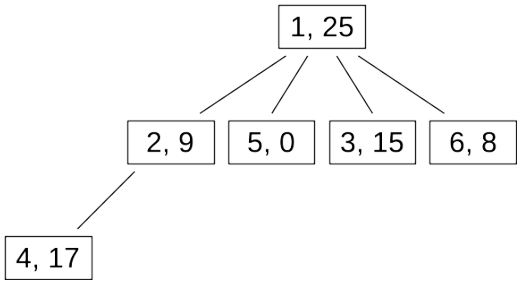
In the binary heap below, the root of the heap is the pair, $1, 25$ (this is the heap you'd expect to get from sample input-1.txt with a branching factor of 2). The $1, 25$ pair is at the top because this is a min-ordered heap, and the 1 is the smallest key. The 25 is just a value that goes with the key, 1.



When the minimum is removed from this heap, removeMin() should return the pair, $1, 25$. The pair, $5, 0$. will be temporarily used as a replacement for the root, but it will be moved down into the heap to restore the heap ordering. Once this is done, the heap should look like:



If the user specifies that the heap should use a branching factor of 4, then this same sample input would build a heap organized like the following instead. As with a binary heap, the ordering constraint requires that the key at a parent is no larger than the key at any of its children. With a larger branching factor of (e.g., 8), the heap would be even shorter (but wider).



Whatever the branching factor, the heap will be organized as an implicit tree, with all the elements stored consecutively in elements of an array. The binary heap shown above would be organized like the following:

1, 25	3, 15	2, 9	4, 17	6, 8	5, 0

The next example, with a branching factor of four would be organized as follows:

1, 25	2, 9	5, 0	3, 15	6, 8	4, 17

You'll traverse the tree structure of the heap by doing math on the index of each element to compute the index of its parent or its children. Here, the choice of a power of two for the branching factor pays off. Instead of using multiplication to compute the index of a child, you'll use a left bit shift (along with a little more arithmetic). Likewise, to compute the index of a parent, you will use a right bit shift (along with a little more arithmetic) instead of division. You get to figure out exactly what math you'll need to traverse the heap's tree structure, but be sure to use bit shifting rather than multiplication and division. In general, try to make these calculations as computationally cheap as possible. In our sample implementation, starting the heap at index 0 (rather than index 1) seemed to make the math work out better.

Your heap should be able to handle duplicate keys and duplicates among the second value in each pair. For example, if the pair, $1, 25$ is in the heap, it's OK if there's also a pair, $1, 10$ or $5, 25$ or even another copy of the pair, $1, 25$. If there are duplicate keys in the heap, the removeMin() can return either pair (when that pair's key is the minimum key).

Your heap will be implemented in two functions / methods. You'll have an insertValue() that puts a new pair of values in the heap. The removeMin() function / method will remove the minimum and return it (as a pair of ints). If you're implementing in Java, returning a pair of integers can be a little bit tricky, since functions can only have one return value. If you want, you can define a little (static) class that holds a pair of ints. Then, you can have the caller pass an instance of this class to the removeMin() method. The method can just fill in the fields of the pair and return void. This way, you won't even have to make a new object every time you call removeMin(). You can just re-use the same instance on every call.

For the representation of your heap, use a resizable array. You don't know in advance how many pairs you're going to have to insert, so you need a representation that can grow as needed. In C, you may have to implement your own resizable array (which is easy), but in C++ you can use std::vector and in Java, you can use ArrayList.

Driver Program

Your heap implementation will be part of a driver program that lets the user specify the heap's branching factor and that reads a sequence of heap operations from standard input. Running the driver will let you test the correctness of the heap implementation and measure its performance.

Command Line Argument

When the user runs the program, they may optionally specify a branching factor as a command-line argument. So, for example, if they run the program like the following, they're specifying that each node of the heap can have 8 children.

```
# In java, the user might run the program like this
java heap 8 < input-3.txt

# In C or C++, it would look like this
./heap 8 < input-3.txt
```

If the user gives command-line arguments (e.g., extra arguments or a value that's not a power of two), your program should print a meaningful error message and exit immediately.

If the user doesn't specify a branching factor on the command line, your program should just use a branching factor of two (i.e., a regular binary heap).

Input Format

The program should read from standard input to determine what heap operations to perform. For example, the following is what's in sample input, input-2.txt.

```
6 7
0 6
3 8
-1
5 5
-1
7 5
8 4
```

```
-1
4 0
-1
1 0
-1
2 2
9 7
-1
-1
-1
-1
-1
```

Each line represents a heap operation. If there are two integer values, that indicates an insertValue() operation, with the key as the first value and the value that goes that key next. For inserts, the program shouldn't print anything.

If a line contains a -1, that indicates a removeMin() operation. For this, the program should remove the minimum from the heap and print that pair to standard output.

You can assume the input is in this format; you don't have to detect or respond to invalid inputs. Also, you can assume that the heap will be non-empty whenever there's a request to do a removeMin().

In my own implementation (C++), I originally used the extraction operator (`>>`) to read input. Later I switched to using scanf() instead. Before I made this change, reading the input had so much overhead that it was hard to see any performance difference between heaps with different branching factors. Switching to scanf() reduced the overhead enough that the branching factor difference was more apparent (but still small). If you're programming in Java, you may encounter a similar problem using Scanner to read input. I'll look around and see if there's a faster way.

Key Comparison Report

Changing the branching factor of the heap will have an effect on its performance. We can measure this by measuring the program's runtime, but we can also track it by counting how many times the heap has to compare two keys. We're just counting comparisons between two heap keys, not other comparisons the heap might have to do (like comparing like comparing the index of an element against the heap size or against zero).

Have your program maintain the count of heap key comparisons as it runs. You can store it as a global variable, or a static field in java. Then, when your program terminates, have it report this count in the format shown in the sample execution below.

Sample Execution

We're providing four sample inputs with this assignment. There are some smaller ones to help you see what the program is supposed to do and to make it easy to check your program is maintaining its heap properly. There are some larger inputs to let you do some additional testing and to let you measure performance on a non-trivial input.

If you run your program on these inputs, here's what you should expect (shown here for a C / C++ solution). Your comparison count may not match the one I get exactly, but it should probably be similar. There may be some places in the heap code where small implementation choices could affect the exact number of key comparisons that are performed.

```
./heap < input-1.txt
1 25
```

```
2 9
3 15
4 17
5 0
6 8
key comparisons: 15

./heap 4 < input-1.txt
1 25
2 9
3 15
4 17
5 0
6 8
key comparisons: 16

./heap 4 < input-2.txt
0 6
3 8
5 5
4 0
1 0
2 2
6 7
7 5
8 4
9 7
key comparisons: 23

69 90
2 6
16 68
29 85
31 21
36 57
1 95
7 55
9 27
12 45
8 14
13 97
6 90
0 73
4 65
key comparisons: 312

./heap 16 < input-4.txt
1999676 291138
1998677 379026
1998656 888671
1998621 962054
1997365 482803
```

```
...
about 2000 lines omitted
...
5823 7924
4823 80360
2603 434887
1905 717911
353 731613
key comparisons: 4435418
```

Analysis and Performance Results

We're going to consider the performance of this heap implementation, both formally and empirically. Put this part in a file called `heap.pdf`.

Briefly analyze the running time for your heap's insertValue() and removeMin() operation. Write up a short paragraph for each. Like last time, you should try to describe your running time so it would make sense to somebody who understood how these two operations were implemented in your solution, but who wasn't necessarily looking at your code as they read your analysis. In your analysis, you can use n as the maximum size of the heap, and you can use b as the user-specified branching factor. Describe the running time of these operations asymptotically, and say anything you can about differences you might expect in the constant factor associated with the running time compared to a regular, binary heap.

Also, run your program as follows to measure its execution time on the largest sample input. Here, we're running with the time command, to measure execution time, and we're sending output to `/dev/null`, so printing the output won't slow down our program too much.

```
# For C / C++
time ./heap 2 < input-4.txt > /dev/null
time ./heap 64 < input-4.txt > /dev/null

# For Java
time java heap 2 < input-4.txt > /dev/null
time java heap 64 < input-4.txt > /dev/null
```

Experiment with different branching factors to see what values give the best performance and how it affects the number of key comparisons. At the end of your `heap.pdf` file, report the timing (user time) for a branching factor of 2 and for a branching factor of 64. Also report the number of key comparisons for these two cases. If you observe anything about what branching factors yield the best performance, briefly describe that also.

The timing results will be noisy, so it may be a little unclear what particular values are best. Running on a private machine (rather than one that's shared) can help with this. Also, running your program multiple times (say, 9 times) and using the median time can help.

Submitting Your Work

In Moodle, you'll see an assignment named `Assignment 2`. Submit the source code to your programs using this link. For your analysis, submit that to a GradeScope assignment with the name `Program 2 Analysis`.