CS 2223 D23 Term. Homework 3

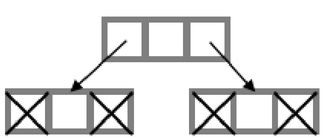
This homework covers material that extends back to before the midterm.

Homework Instructions

- This homework is to be completed individually. If you have any questions as to what constitutes improper behavior, review the examples as I have posted online http://web.cs.wpi.edu/~heineman/html/teaching_/cs2223/d23/#policies.
- Due Date for this assignment is **on Canvas**. Homeworks received after 6PM will receive a 25% penalty if submitted within 48 hours, otherwise zero points.
- Submit your assignments electronically using the canvas site for CS2223. Submit your homework under "HW3". You must submit a single ZIP file that contains all of your code as well as the written answers to the assignment.
- All of your Java classes must be defined in a packager USERID.hw3 where USERID is your CCC user id (i.e., your email address without the @wpi.edu).

Homework Context

This homework introduces students to the Binary Tree data structure. When used as the basis for Binary Search Trees, this structure offers the ability to dynamically insert and remove values, while supporting efficient search and traversals. You will also explore the tradeoffs when implementing symbol tables using the three structures, namely `SeparateChainingHashST`, `LinearProbingST` and `BinaryTree`.

<table>
<tr><td>Heap Array Structure for Priority Queue</td><td>
<table>
<tr><td></td><td>level 0</td><td colspan="2">level 1</td><td colspan="4">level 2</td><td colspan="8">level 3</td><td colspan="3">level 4</td></tr>
<tr><td>-</td><td>15</td><td>13</td><td>14</td><td>12</td><td>11</td><td>12</td><td>14</td><td>8</td><td>9</td><td>1</td><td>10</td><td>8</td><td>6</td><td>9</td><td>7</td><td>4</td><td>5</td><td>2</td></tr>
</table>
</td></tr>
<tr><td>Linear Probing Array for Symbol Table</td><td>
<table>
<tr><td>8</td><td>14</td><td>13</td><td>-</td><td>6</td><td>15</td><td>12</td><td></td><td>4</td><td>5</td><td>1</td><td></td><td>8</td><td></td><td>-</td><td>7</td><td>10</td><td>-</td></tr>
</table>
</td></tr>
<tr><td>Recursive Data Structure</td><td>Binary Tree
</td></tr>
</table>

Copy classes/files into your **`USERID.hw3`**: `BST`, `MaxPQ`, `MergeMaxPQ`, `TrialST`, WrittenQuestions.txt
Classes you do not copy: `Collection`, ID

Q1. Evaluating Symbol Table Structures [20 pts]

When evaluating symbol tables (STs), consider the Figure on p. 487 of the textbook, which summarizes the "asymptotic cost summary" for different ST implementations.

algorithm (data structure)	worst-case cost (after N inserts)		average-case cost (after N random inserts)		key interface	memory (bytes)
	search	insert	search hit	insert		
sequential search (unordered list)	N	N	$N/2$	N	`equals()`	$48\,N$
binary search (ordered array)	$\lg N$	N	$\lg N$	$N/2$	`compareTo()`	$16\,N$
binary tree search (BST)	N	N	$1.39 \lg N$	$1.39 \lg N$	`compareTo()`	$64\,N$
2-3 tree search (red-black BST)	$2 \lg N$	$2 \lg N$	$1.00 \lg N$	$1.00 \lg N$	`compareTo()`	$64\,N$
separate chaining† *(array of lists)*	N	N	$N/(2M)$	N/M	`equals()` `hashCode()`	$48\,N + 32\,M$
linear probing† *(parallel arrays)*	N	N	< 1.50	< 2.50	`equals()` `hashCode()`	between $32\,N$ and $128\,N$

† *under uniform hashing assumption*

Asymptotic cost summary for symbol-table implementations

After inserting N (key, value) pairs into a symbol table, the first two columns provide a measure of the worst case to (a) search for a key; or (b) insert a new (key, value) pair. The third and fourth column suggest an average-case cost of doing same. How do these align with real-world examples?

Use the following domain to evaluate the performance of these different ST implementations in terms of the number of times **`equals()`** is called. Note that **`compareTo()`** also is accounted in this total, but that is not really used within a hash Symbol Table (can you see why?).

College ID numbers

College IDs typically have 9 digits. These can be represented by a 32-bit **int** or a Java String. Also, no ID starts with the digit 0. I have provided an ID class that you can use as the Key for any Symbol Table. One added benefit is that whenever anyone calls **`equals(o)`** or **`compareTo(o)`**, the ID object will increment a counter associated with that ID. In this way, you don't have to instrument the code that implements the symbol table just to determine how many times an ID was compared.

Copy the **algs.hw3.TrialST** class into your **USERID.hw3** package and modify it to produce several tables of statistics that include:

a) # of compares/equals when building the initial structure.
b) average # of compares/equals when processing 1,024 `get(key)` operations where the key is known to exist in the symbol table.
c) average # of compares/equals when processing 1,024 `get(key)` operations where the **key does not exist in the symbol table**.

Modify **TrialST** to create similar **report()** methods as I have already provided. You will evaluate FIVE different Symbol Table implementations:

- `AVLTreeST`
- `BST`
- `LinearProbingHashST`
- `SeparateChainingHashST`
- `SequentialSearchST`

You will need to modify your code to produce five SIMILAR LOOKING tables that are assigned the labels (Table I, Table II, Table III, Table IV, Table V). Now make sure that the label for each table is the one that most closely matches the output on the next page.

For this question, you must complete the **WrittenQuestions.txt** file to identify which Table is generated by which structure. On the next page are the tables you should be able to reproduce.

Once you can generate these tables, make a list (in decreasing order of efficiency) of these five implementations of Symbol Table when considering the average cost for a HIT.

Once again, your output tables may not exactly match the BUILD, HIT, and MISS columns due to the nature of working with randomized trials. STILL, there are clear differences between each table that become evident once N is large enough.

Table I: Symbol Table Structure (FILL IN)

N	BUILD	HIT	MISS
64	1.97	4.06	32.69
128	7.94	16.13	64.62
256	31.88	64.25	128.69
512	127.75	256.50	256.70
1024	511.50	512.50	512.75
2048	2047.00	1040.25	1024.73
4096	8190.00	2194.35	2048.75
8192	32764.00	4066.89	4096.78
16384	131064.00	8450.84	8192.71
32768	524272.00	17230.81	16384.71
65536	2097120.00	34226.50	32768.82

[Note: don't worry about the tabular format]
[it's ok if the columns do not exactly line up]

Table II: Symbol Table Structure (FILL IN)

N	BUILD	HIT	MISS
64	0.30	0.67	3.84
128	0.72	1.57	4.18
256	1.68	3.58	4.77
512	3.90	8.20	5.30
1024	8.74	9.24	5.88
2048	19.64	10.21	6.38
4096	42.97	11.09	6.89
8192	94.69	12.40	7.44
16384	204.71	13.27	7.84
32768	446.06	14.20	8.40
65536	964.26	15.36	9.03

Table III: Symbol Table Structure (FILL IN)

N	BUILD	HIT	MISS
64	0.05	0.16	1.38
128	0.16	0.39	1.23
256	0.32	0.78	1.38
512	0.64	1.60	1.41
1024	1.14	1.53	1.51
2048	2.12	1.45	1.46
4096	4.03	1.48	1.50
8192	8.86	1.57	1.56
16384	15.87	1.48	1.43
32768	33.66	1.52	1.43
65536	72.78	1.53	1.64

Table IV: Symbol Table Structure (FILL IN)

N	BUILD	HIT	MISS
64	0.33	0.78	4.21
128	0.99	2.24	5.54
256	2.15	4.81	5.96
512	5.58	12.17	7.30
1024	11.18	12.18	7.22
2048	23.22	12.53	7.59
4096	54.43	14.51	8.56
8192	125.77	16.83	9.67
16384	254.67	16.84	9.62
32768	560.49	18.23	10.38
65536	1259.95	20.62	11.67

Table V: Symbol Table Structure (FILL IN)

N	BUILD	HIT	MISS
64	1.19	0.68	4.78
128	2.86	1.28	4.57
256	6.44	2.61	4.69
512	13.99	5.10	4.67
1024	27.41	4.96	4.73
2048	55.76	4.94	4.67
4096	111.45	5.22	4.86
8192	223.40	4.95	4.68
16384	446.63	5.14	4.71
32768	897.18	5.01	4.73
65536	1793.04	5.04	4.85

Q2. Working with Heaps [20 pts.]

Given two max heaps of size M and N, devise an algorithm that returns an array of size M + N containing the combined items from M and N in ascending order.

Copy the **algs.hw3.MaxPQ** and **algs.hw3.MergeMaxPQ** classes into your **USERID.hw3** package and complete the methods in **MergeMaxPQ**. Note that this **MaxPQ** class has a special method, **peekMax()**, that returns the maximum value maintained by the **MaxPQ** without removing it.

You will need to instrument your local copy of **MaxPQ** to keep track of the number of "key operations" – which includes both calls to **less()** and to **exch()** – and then complete the implementation of **keyOperations()** and **resetKeyOperationsCount()**.

When you are done, the output of **MergeMaxPQ** should be:

```
[13, 31, 41, 50, 59, 77] should be output
[13, 31, 41, 50, 59, 77]

        1024    2048    4096    8192    16384   32768   65536
     +---------------------------------------------------------
 1024|  50053   81072   149647  298615  621253  1315670 2803452
 2048|  81071   112091  180669  329640  652286  1346721 2834546
 4096|  149646  180669  249247  398218  720864  1415299 2903124
 8192|  298614  329640  398218  547189  869835  1564270 3052095
16384|  621252  652286  720864  869835  1192481 1886916 3374741
32768|  1315669 1346721 1415299 1564270 1886916 2581351 4069176
65536|  2803451 2834545 2903123 3052094 3374740 4069175 5557009
```

[Sorry to have to update this sample table, but MY code had a defect, so I had to fix properly.]

Q3. Working with Binary Search Trees [60 pts.]

There are three kinds of methods you can envision for Binary Search Trees:

- Structural – just inspects .left and .right references, like computing the height of a tree
- Read Only – traverses a tree by inspecting the keys but makes no changes to the structure. Like the `get()` method.
- Modifying – like the `put()` method.

Copy the **`algs.hw3.BST`** class (and **`TestBST`** class) into your **`USERID.hw3`** package and complete the methods at the end of the class. This BST is simplified since it only contains an integer key for each node.

Note that all references below to BST refer to **`USERID.hw3.BST`** which is a streamlined BST that you will be working with

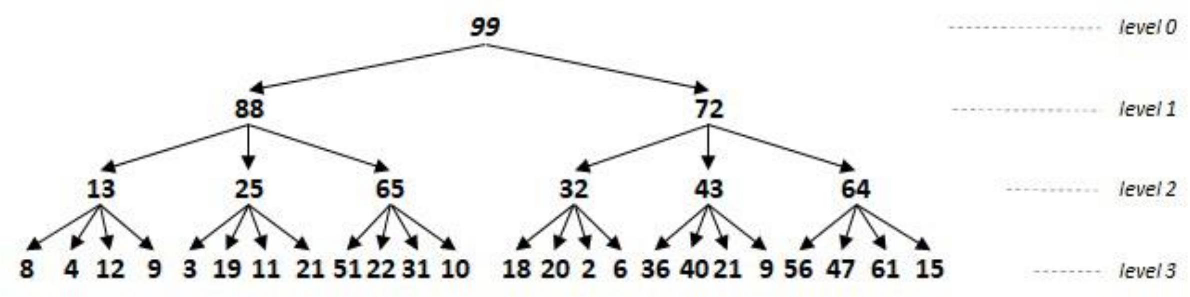
- **[15 pts.]** Return a copy of the BST.
- **[15 pts.]** Return an **`int`** that computes the total number of nodes in the BST whose **`int`** values are even.
- **[15 pts.]** Return a **`Queue<Integer>`** that contains the integer depths for all nodes in the BST.
- **[15 pts.]** Remove all leaf nodes that contain odd key values.

Once you are done, you can execute the `TestBST` class to check any obvious issues with your implementation.

Q4. Bonus Questions (1 pt)

A Max Priority Queue using a heap conforms to (a) the Heap Ordered Property (the value of a node is larger than or equal to either of its children; and (b) the Heap Shape Property, where each level is filled before any nodes appear on any subsequent level.

What if you make a small change to allow children on level k to have k+2 children? Let's call this an **Expanding Max Priority Queue**.



The above is such an **Expanding** Max Priority Queue where, as you can see, each parent node is larger than **any** of its children. It is still possible to store this structure in a contiguous array (though the mapping is a bit more complicated).

Complete the implementation by storing this heap in an array and perform empirical evaluation. Use the worst case as the benchmark, namely, where you insert N integers in ascending order (one at a time) into an empty Max Priority Queue.

Using the same approach as shown in an earlier problem, count the number of key operations (`exch` and `less`) and produce a table that looks like the following.

logN	N	MaxPQ	ExpandMaxPQ
2	4	16	16
3	8	55	52
4	16	168	169
5	32	469	417
6	64	1226	1145
7	128	3055	2619
8	256	7348	6436
9	512	17209	14500
…			

While the number of key operations is smaller for **Expanding** Max PQ, the time to complete is longer, so there is still a tradeoff that our empirical operation counting is missing.

Change Log

Updates are enumerated below:

1. Clarified that the output from Question 1 may be different from the tables that I've produced. Fortunately, each of your tables will likely differ by only a tiny amount from the values in the final rows of each of the tables.