EECS 560
Lab 9: Leftist Heap vs Skew Heap

Apoorv Ingle, Prof. Shontz
Fall 2017

1 Lab Details

- Maximum Possible Points: 50
- Lab Timings:
  1. Monday Lab: Oct 30, 9:00 AM–10:50 AM
  2. Wednesday Lab: Nov 1, 12:00 PM–1:50 PM
  3. Friday Lab: Nov 3, 12:00 PM–1:50 PM
- Lab Due:
  1. Monday Lab: Nov 5, 5:00 PM
  2. Wednesday Lab: Nov 7, 5:00 PM
  3. Friday Lab: Nov 9, 5:00 PM

2 Assignment Instructions

In this lab you will implement a priority queue ADT in the form of skew heap with basic operations on it and compare it with leftist heap (from lab 8). The lab kit includes a simulator that will be used to run and test your implementation of the data structure (in this case the leftist heap and skew heap). You will have to fill in the necessary code in function body marked as—\texttt{IMPLEMENT ME}(). You should comment out or delete the line containing \texttt{IMPLEMENT ME}() macros after you are done implementing the function.

Please use the Makefile given with the lab kit to compile and test your code. The sample test data will be read from the file data.txt. \textit{You may not use the standard template library (STL) or standard containers library implementations for this lab.} You may use the data structures you have implemented in previous labs. Please use the principles of object oriented programming to design your program. You are free to add new private data members and private methods in your classes. You may not change the signature of the public methods (pre-declared in \texttt{*.hpp} files), the implementations of the pre-declared public methods will be used to test your code.

In this lab you will simulate a simple \textbf{Scheduler} for a hypothetical operating system. The operating system can run multiple \textbf{Applications} and each application consists of multiple \textbf{Tasks}. Only a few instructions can be executed per time slice by the operating system. Your assignment is to implement a priority queue so that tasks (irrespective of the application) can be executed according to their correct priority order.

\textit{Note: Please ensure your code for leftist-heap is fixed before starting this lab.}

2.1 Class Diagram for the Task Scheduler Simulator

Please refer to fig. \ref{fig:ClassDiagram} for visualizing the relationship between classes for the scheduling simulator system.
3 Simulator Related Classes and Methods

Please refer to the Lab 8 instruction sheet for implementation instructions related to Task, Application and Scheduler.

Figure 1: Simulator system class diagram
4 PriorityQueue and Related Operations

4.1 PriorityQueue Using Leftist Heap

Please refer to the Lab 8 instructions for operations for leftist heap.

4.2 PriorityQueue Using Skew Heap

You may use the child-pointer based implementation design for the skew heap. The priority of the Task is computed by its nice value—the lower the nice value, the higher the priority.

1. Add element: signature void PriorityQueue::addElem(Task*);

This function adds a Task into the PriorityQueue. (Refer fig. 2). Adding a task with nice value, say 2 (and tid 15), will change the queue structure to fig. 3.

Hint: Can you use concat to implement this function?

2. Delete minimum element: signature Task* PriorityQueue::deleteMinElem()

Deletes the minimum element from the queue and restructuring it to maintain the priority queue property. It should return the element deleted from the queue, e.g., executing the delete min function on fig. 4 will result in the queue as shown in fig. 5 and return the highest priority Task.

Hint: Can you use concat to build this function?

3. Concatenation: signature void concat(PriorityQueue* that)

concat method merges 2 priority queues into a single priority queue. Please implement the algorithm as explained in the lecture notes and covered in class by using the concept of ranks.

4. Inorder traversal: signature int PriorityQueue::inorderTraversal(int, int (op*)(Task*))

traverses the queue in an inorder fashion. For example, given the correct implementation of the *op function, the queue structure of fig. 5 should be able to print all the tasks in the priority queue in the following order:

(8, 6) (15, 2) (10, 2) (9, 5) (12, 3).

5. Postorder traversal: signature void PriorityQueue::postorderTraversal(int, int (op*)(Task*))

traverses the queue in a postorder fashion. For example, given the appropriate implementation of the *op function, the queue structure created with fig. 5 should be able to print the tasks in the priority queue in the following order:

(8, 6) (15, 2) (9, 5) (12, 3) (10, 2).
5 Compare and Contrast Performance of Heap Operations

After you are done implementing the data structures and its associated operations, you will have to compare average run time complexities of the following functions for both leftist heap and skew heap. Please submit your graphs with generated timings for all runs (report document).

Tip: To ensure your results are reproducible you may fix your random number generator seed to a large number by using srand(int) function. To ensure fairness, use the same data for leftist heap and skew heap for comparison.

5.1 Performance Analysis of Adding an Element:

```c
void PriorityQueue::addElem(Task*)
```

Generate a PriorityQueue with random Tasks with nice number between 0 and RAND_MAX of the following sizes: 10, 50, 100, 500, 1000, 5000, 10000, 50000, 100000 elements. Generate a random task with nice number between 0 and RAND_MAX. Call the function `void PriorityQueue::addElem(Task*)` to add the randomly generated element into an appropriate position in PriorityQueue and time it.

For each of the sizes mentioned above repeat the experiment at-least 10 times and note the timings for each run. Perform the experiment twice—one for leftist heap and once for skew heap. Plot a graph of the heap size (x-axis) vs. the average time taken in milliseconds (or nanoseconds) (y-axis) for both heaps.

5.2 Performance Analysis of Deleting Minimum Element:

```c
Task* PriorityQueue::deleteMinElem()
```

Generate a PriorityQueue with random Tasks with nice number between 0 and RAND_MAX of the following sizes: 10, 50, 100, 500, 1000, 5000, 10000, 50000, 100000 elements. Generate a random task with nice number between 0 and RAND_MAX. Call the function `Task* PriorityQueue::deleteMinElem()` to remove the highest priority element from the PriorityQueue and time it.

For each of the sizes mentioned above repeat the experiment at-least 10 times and note the timings for each run. Perform the experiment twice—one for leftist heap and once for skew heap. Plot a graph of the heap size (x-axis) vs. the average time taken in milliseconds (or nanoseconds) (y-axis) for both heaps.
5.3 Performance Analysis of Concatenating 2 Queues:

```cpp
template<typename T>
void PriorityQueue::concat(PriorityQueue<T>*)
```

Generate 2 `PriorityQueue`s with random `Task`s with nice number between 0 and RAND_MAX of the following sizes each: 5, 25, 50, 250, 500, 2500, 5000, 25000, 50000 elements. Merge both the priority queues by calling the function `void PriorityQueue::concat(PriorityQueue<T>*)` and time it.

For each of the sizes mentioned above repeat the experiment at least 10 times and note the timings for each run. Perform the experiment twice—once for leftist heap and once for skew heap. Plot a graph of the heap size (x-axis) vs. the average time taken in milliseconds (or nanoseconds) (y-axis) for both heaps.

6 Questions

Please answer the following questions in not more than 5 lines each and submit it with your implemented code in the PDF format. (Writeup document)

1. What is the worst case algorithmic asymptotic complexity i.e. \(O(\cdot)\) of each of the operations that you have implemented. (10 points)
   a. Add element in the skew heap (void PriorityQueue::addElem(Task*)).
   b. Delete min element from the skew heap (void PriorityQueue::deleteMinElem()).
   c. Merge 2 skew heaps (void PriorityQueue::concat()).

2. Is the average case complexity equal to the worst case complexity in each of the 3 cases above? If not, mention the average case complexity for each of the above operations in a., b., and c., and also explain in short the reason for the difference.

7 Grading Scheme

- Your submitted code should compile and run on the EECS unix machines. (Please use cycle2.eecs.ku.edu/cycle3.eecs.ku.edu/EECS lab machines. g++ v6.2.1) There should be no memory leaks, and no compilation warnings. (10 points)
- Your code will be tested against the example flow given at the end of the instruction document. (10 points)
- Hidden test case suite run by the grader on your implementation. (20 points)
- Write-up answering questions in §6 (10 points)

8 Deliverables

1. Code that compiles and runs with `Makefile`.
2. Write up answering questions from §6
3. Report consisting graphs and generated timing values in a tabular format.
4. Code archive and write-up should have correct naming conventions as described in §9

9 Submission and Miscellaneous Hints

1. Please add the grader’s (Dravid Joseph) email id in the To section of the mail (dravidjoseph@ku.edu) and my (Apoorv Ingle) email id in CC (apoorv.ingle@ku.edu)
2. Your subject line for the submission should be of the form [EECS 560] Lab-<Lab #> <Lab Day> <Your KU username> eg. [EECS 560] Lab-9 M j543h898
3. Your reports (if any) should be named as `<your KU username>-leftistH-vs-skewH-lab9-report.pdf` and your writeups (if any) should be named as `<your KU username>-leftistH-vs-skewH-lab9-writeup.pdf`
4. Your code tar archive will be automatically named in correct format by running `make tar`.

5. Expand the tar ball: `$ tar xvf <filename>.tar.gz`

6. Make cheat-sheet:
   - compiling and linking your program: `make clean install`
   - testing your program: `make test`
   - bundle your code in a tar archive: `make tar`

   *Note: Please change `XXXXXXX` in first line of the Makefile into your KU username of the format (j052h567) before running `make tar`*

7. Counting always starts from 0.

8. Please assume all the data input are in correct format and logically coherent. Validation for correctness is not required while data is being read from the input file.

10 Sample Test Cases

*Note: This is just for illustration using the data.txt file as input.*

10.1 Input

5
T0
0
21 init (10, 6, 21, 5) (11, 0, 21, 7) (12, 6, 21, 2) (13, 9, 21, 1)

T1
1
22 App1 (16, 3, 22, 2) (17, 5, 22, 1) (18, 6, 22, 2)

T2
3
23 App2 (110, 1, 23, 5) (111, 5, 23, 7) (112, 10, 23, 2) (113, 7, 23, 1) (114, 3, 23, 2) (115, 3, 23, 1)

T3
2
24 App3 (121, 6, 24, 1)

T4
1
25 App4 (122, 8, 25, 5) (123, 7, 25, 7)

10.2 Output

T0
Tasks executed:
Application added to queue: init

T1
Tasks executed (1): 11(21,0)
Application added to Queue: App1

T2
Tasks executed (3): 16(22,3) 17(22,5) 10(21,6)
Application added to Queue: App2

T3
Tasks executed (2): 110(23,1) 114(23, 3)
Application added to Queue: App3

T4
Tasks executed (1): 115(23,3)
Application added to Queue: App4

T5
Tasks executed: 111(23,5) 12(21,6) 10(21,6) 121(24,6) 113(23,7) 123(25,7) 122(25,8) 13(21,9) 112(23,10)

Done.