Introduction to Data Structures (430.217, 002)
Seoul National University

Project #2
Due: 24, Nov, 2023, 23:59:59

1 Introduction

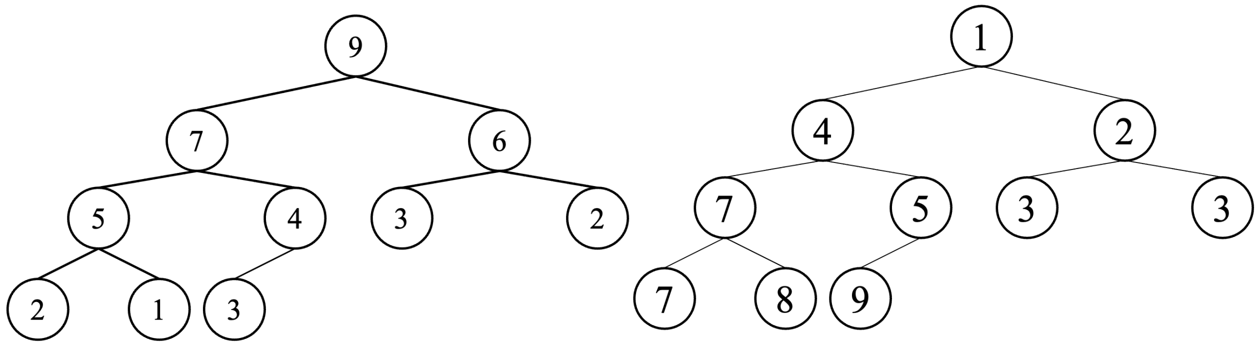
- You are not allowed to copy and paste the code of others (e.g., classmates or website.) Please try to work on your own.
- It is desirable to check if your codes run on the environment we have introduced in the installation guide. The version of **g++** and **c++** should be **11** and **17**, respectively. If it works only on your PC, there may be deduction of the score.
- All kinds of questions are welcome. If you have problem or find out typos in the given skeleton code, please leave the question on the eTL QnA board. It is recommended to make it open to your classmates. However, we are not going to fix your bugs directly.
- You can modify the given header file. It is just one example. You can add a member variable, delete the existing one, or rename functions.
- Grading policy will be as follows:
  - 100 points for 100 test cases.
  - Late submission: 10% deduction per 12 hours.
  - Memory leak or memory error: 30% deduction.
  - Only working on your machine: 20% deduction.
  - Using libraries other than **iostream**, **vector**, **string**: 70% deduction.
  - Not submitting filenames as announced: 10% deduction.
  - Any other non-compliance with given rules will be also deducted.

2 Backgrounds

2.1 Heap

A **heap** is a tree-based data structure that maintains a partial order known as the heap property. The heap properties are as follows:

1. Complete binary tree
   - A **complete binary tree** is a special type of binary tree with two distinct properties:
     (a) All levels are **fully filled** except possibly the last.
       - This means that every level of the tree has the **maximum number** of nodes possible, except possibly for the last level.
     (b) Last level is **left-adjusted**.
       - If the last level is not fully filled, the nodes in that level are filled from **left to right** without any gaps.



(a) Max heap　　(b) Min heap

Figure 1: Two types of heap

  - This ensures there's no missing node in the middle of the last level.

2. Heap-order
   - Internal nodes in tree should satisfy the priority.
     (a) Max heap (Figure 1a)
       - For any given node 'I', the value of 'I' is **greater than or equal** to the values of its children.

     (b) Min heap (Figure 1b)
       - For any given node 'I', the value of 'I' is **less than or equal** to the values of its children.

2.2 Min heap implemented by array

As depicted in Figure 2, Min Heap can be efficiently implemented using arrays, and in such an implementation, the nodes of the tree are accessed through the indexes of the array. Min Heap can be implemented using arrays in the following ways:

1. Array representation
   - Each level of a fully binary tree is filled from left to right, storing elements in an array in the order of the levels in the tree.
   - The first index in the array (index 0) represents the root of the tree, the next index represents the left child of the root, and the next index represents the right child of the root.
2. Index relationships
   - When the index of the parent node is called i, the index of the left child node is $2i + 1$, and the index of the right child node is $2i + 2$.
   - If the index of a child node is j, the index of its parent node is `floor`$((j - 1)/2)$.

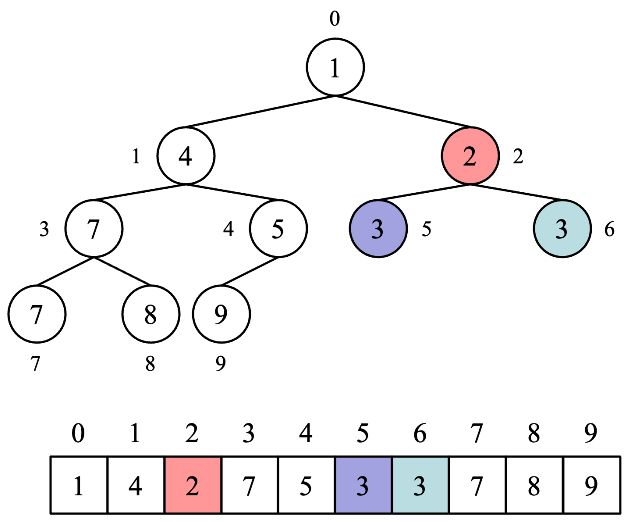


Figure 2: Min heap implemented by array

3. Minimum value

   - In a Min Heap, the minimum value is always located at the **root**, so it is the first element of the array.

2.3 Insert operation in heap

As depicted in Figure 3, whenever we insert/delete a new element into the heap, we need to ensure that the heap properties are preserved. The insert operation on a Min heap proceeds in the following steps:

1. Find Insertion Position.

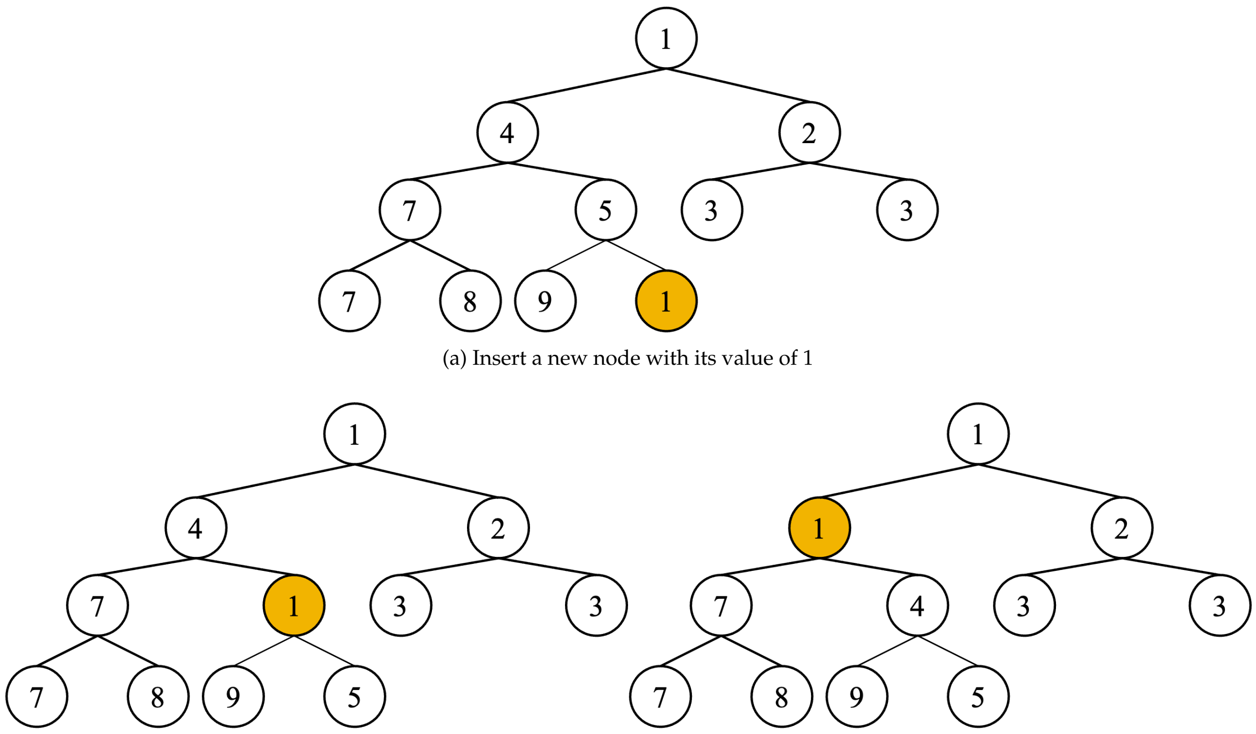
   - The new element is inserted at **the last position** in the tree to maintain the properties of a complete binary tree.
   - It is inserted at the position after the rightmost node of the last level.

2. Insert element.

   - Insert the new element at its position.

3. Maintain Heap property.

   - Since the inserted element may not satisfy the Min Heap property, it is moved upward relative to its parent node and repositioned until it satisfies the Heap property.
   - This process is called "**up-heap**", "**percolate-up**", or "**bubble-up**".



(a) Insert a new node with its value of 1

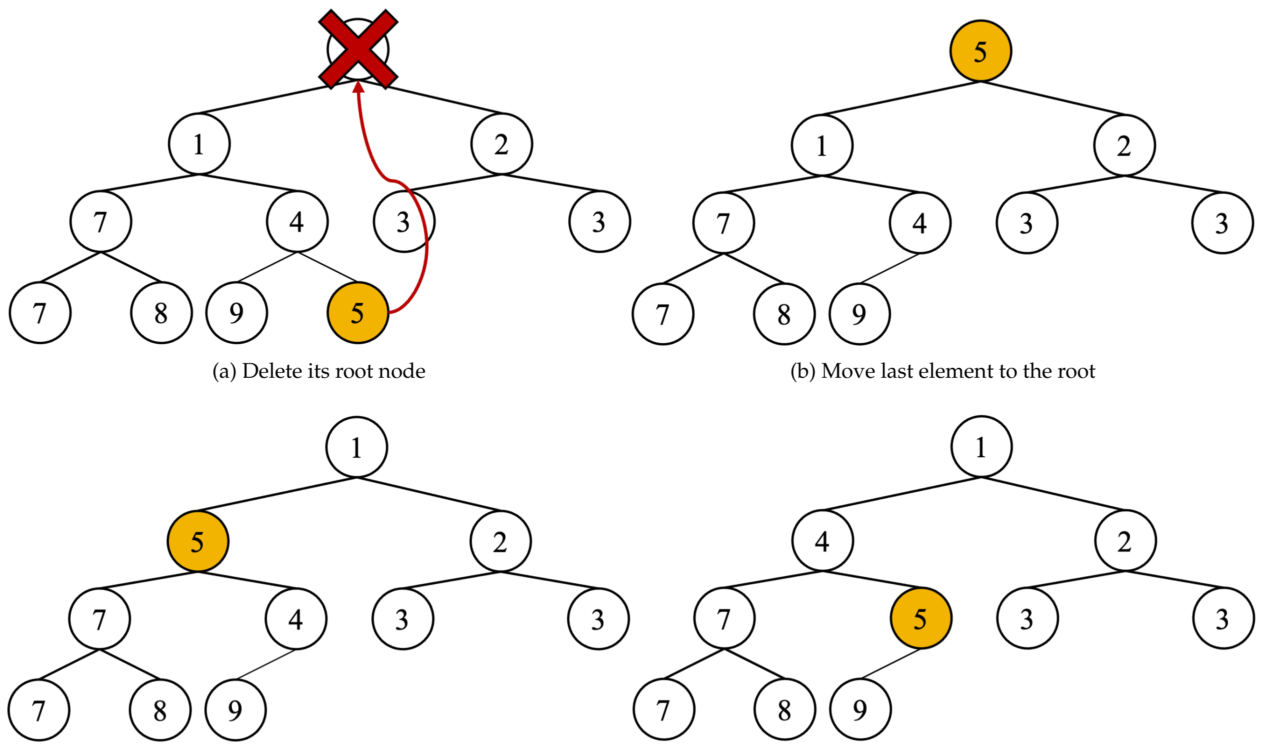
(b) Up-heap: swap '1' and '5'

(c) Up-heap: swap '1' and '4' (Done)

Figure 3: Insert operation

4. Up-Heap process.
   - Compare the inserted node to its parent node.
   - If the inserted node is smaller than its parent, they swap positions.
   - This process is repeated until the new node is no longer smaller than the parent, or until the root node is reached.
5. Done.
   - When the new element reaches the correct position, the Min Heap again satisfies all properties and the insert operation is complete.

The insert operation has a time complexity that is **proportional to the height** of the tree in the worst case, which is $O(\log n)$, where n is the number of elements in the heap, because even in the worst case where the new element needs to travel to the root node, it only needs to travel the height of the tree.



(a) Delete its root node (b) Move last element to the root

(c) Down-heap: swap '5' and '1' (d) Down-heap: swap '5' and '4' (Done)

Figure 4: Delete operation

2.4 Delete operation in heap

As depicted in Figure 4, The delete operation on a Min Heap typically means removing the root node, that is, the node with the smallest value. This happens in the following steps:

1. Remove the element to be deleted.
    - The root node of the Min Heap is deleted, which is the smallest element in the Min Heap.
2. Move last element.
    - The last element in the heap is moved to the position of root. This makes the tree take the form of a fully binary tree again.
3. Restore Heap properties.
    - The last element moved may violate the Min Heap condition, so we move it down to satisfy the Heap properties.
    - This process is called "**down-heap**", "**percolate-down**", or "**bubble-down**".

4. Down-Heap process.
   - Compare the moved node with its children. For a Min Heap, compare it to the child with the smaller value of the two children.
   - If the child is smaller than the moved node, it swaps places with the child with the smaller value.
   - Repeat this process until the node is smaller than the child node, or until a leaf node is reached.
5. Done
   - When the exchange process reaches a position that satisfies the Min Heap property, the delete operation is complete.

The time complexity of this delete operation is also **proportional to the height** of the tree in the worst case, so it is $O(\log n)$, where n is the number of elements in the heap. This is because, in the worst case, the deleted node only needs to travel down the height of the tree until it reaches the leaf node.

2.5 Priority queue

Heap is a great fit for implementing such a Priority Queue because it provides the following features:

1. Priority Sorting
   - With Heap, elements are automatically sorted by priority.
   - When using Min Heap, the lowest priority (the one with the smaller number) is considered high, and the opposite is true when using Max Heap.
2. Efficient insertion & removal
   - When adding new elements, Heap can insert elements with $O(\log n)$ time complexity.
   - Removing the highest priority element (root in Heap) also takes $O(\log n)$ time complexity.
3. Priority lookup
   - In Heap, the element with the highest priority can be looked up immediately.
   - Since the smallest element in Min Heap and the largest element in Max Heap are located at the root, all you need to do is look at the first element of the array.
4. Maintains the properties of a complete binary tree
   - Heap maintains the properties of a complete binary tree.
   - This means that all levels are filled from left to right, which allows for efficient use of memory in implementations using arrays.

3 Implementation

3.1 Part 1: Basic operations of priority queue (40 pts)

Given *"PriorityQueue.h"*, the goal is to complete the skeleton code, *"PriorityQueue.cpp"*, by filling in all the TODOs. You have to all TODOs in *"PriorityQueue.cpp"*. The **PriorityQueue** class consists of some members. In **PriorityQueue** class of *"PriorityQueue.hpp"*:

- Private members:
  - **std::vector** <**int** > heap:
    - Heap (vector) to implement prioity queue.
  - **int** direction:
    - It determines whether it is minHeap or maxHeap.
    - **1** for minHeap, and **-1** for maxHeap.
  - **void swapPQ** (**int** idx1, **int** idx2)
    - Swaps the elements at the specified indices in the heap.
  - **void upHeap** (**int** idx)
    - Adjusts the heap by moving the specified element upwards (towards the root) to maintain the heap property.
  - **void downHeap** (**int** idx)
    - Adjusts the heap by moving the specified element downwards (away from the root) to maintain the heap property.
- Public members:
  - **PriorityQueue** (**int** direction):
    - Constructor of the class.
  - ∼**PriorityQueue** ():
    - Destructor of the class.
  - **void insertHeap** (**int** e):
    - Insert a new entry to the queue.
  - **int popHeap** ():
    - Pop its top entry. if the queue is empty, then print *"Empty queue."*.
    - Note that the priority must be maintained after the entry is popped.
  - **int topHeap** () **const**:

    * Return its top entry. if the queue is empty, then print *"Empty queue."*.

- **int sizeHeap** () **const**:
    * Return size of queue.
- **bool emptyHeap** () **const**:
    * Check if the queue is empty. **true** (i.e., **1**) for empty queue.

3.1.1 Heapify Implementation

void upHeap (**int** idx):
The upHeap operation ensures the heap property by moving an element at index *idx* upwards towards the root until its parent has a higher (or lower, depending on the heap type) priority. This is typically called after an insert operation.

void downHeap(**int** idx):
The downHeap operation ensures the heap property by moving an element at index *idx* downwards until both its children have lower (or higher, depending on the heap type) priorities. This is typically called after a pop operation to restructure the heap.

3.1.2 Insert & pop implementation

void insertHeap (**int** e):
This function inserts a new element, *e*, into the heap. You can refer to Figure 3.

int popHeap ():
This function pops an element with the highest priority of the queue. You can refer to Figure 4.

3.1.3 Other basic operations implementation

void swapPQ (**int** idx1, **int** idx2):
This function swaps the elements located at the indices *idx1* and *idx2* within the heap. This is a utility function that is often used during heap operations to maintain the heap property.

int topHeap () **const**:
This function returns the top element of the heap without removing it. It provides a way to inspect the highest (or lowest, depending on heap type) priority element without altering the heap.

int sizeHeap () **const**:
Returns the number of elements currently stored in the heap.

bool emptyHeap () **const**:
This function checks if the heap is empty, i.e., contains no elements, and returns a boolean value ('1' if empty, '0' otherwise).

3.1.4 Input & output

Input:
The first line contains the direction d that determines minHeap and maxHeap. The second line contains the number of commands, N ($1 \leq N \leq 1,000$). The third line receives N commands. The commands can be one of the following:

Instruction	Your program does
I *p*	Insert a new document with priority *p*
P	Print the top entry (or *"Empty queue."* for empty queue) and pop it
T	Print the top entry (or *"Empty queue."* for empty queue) without removing
S	Print the length of the queue
E	Print '1' if the queue is empty

Output:
For each command, print the answer for each instruction. Figure 5 is examples.

Input	Output
-1	1
20	11
E	15
I 3	6
I 7	15
I 11	11
T	5
I 2	8
I 15	7
I 5	5
T	3
S	2
I 8	Empty queue.
P	
P	
S	
P	
P	
P	
P	
P	
P	

(a) Example 1

Input	Output
-1	1
20	3
E	2
I 3	6
I 7	2
I 11	3
T	5
I 2	5
I 15	7
I 5	8
T	11
S	15
I 8	Empty queue.
P	
P	
S	
P	
P	
P	
P	
P	
P	

(b) Example 2

Figure 5: Example for part 1.

3.2 Part 2: Document printer implementation with priority queue (60 pts)

In this part, you should implement document printer. A company uses a shared document printer. The printer can only print one document at a time, and each document has its importance. The importance of documents is represented by natural numbers, with **lower numbers indicating higher importance**. The printer operates based on the following rules:

1. Take the document from the front of the queue.
2. If there's any other document in the queue with a higher importance, place the taken document at the end of the queue.
3. Otherwise, print the taken document.

For example, if there are four documents in the queue (A, B, C, D) with importance values of 2, 1, 4, and 3 respectively, the printer will print in the order B, A, D, C. Given the queue of documents and their importance, determine the order in which the documents will be printed and find the M-th document to be output.
To do this, **Document** class and **Printer** class should be implemented. **Document** class consists of following members:

- Private members:
  - **std::string** id:
    - ID of the corresponding document. Its length is not longer than 10.
    - It consists of a combination of alphabets and digits.
    - *e.g.*, "IKvOHsldTd", "E0XB776ibk"
  - **std::string** title:
    - The title of the corresponding document.
    - It consists of only alphabets. It is assumed that there is no white space character.
    - *e.g.*, "IntroductionToDataStructures", "DataCommunicationNetworks"
  - **int** priority:
    - The priority of the corresponding document.
- Public members:
  - Document (**std::string** id, **std::string** title, **int** priority):
    - Constructor of the class.
  - Getter functions to access private members:
    - **std::string** getId() **const**

  - **std::string** getTitle() **const**
  - **int** getPriority() **const**

Further, we'll provide **Printer** class. It is very similar to **PriorityQueue** class. Only the different members from original ones are as follows:

- Private members:
  - **std::vector <Document >** docs
    - Queue (vector) to implement document printer.
- Public members:
  - **void** insertDoc(**std::string** id, **std::string** title, **int** priority)
    - Insert a new document.
  - **void** popDoc();
    - Pop its top document.
  - **Document** topDoc() **const**
    - It returns the document with the highest priority in the printer.

3.2.1 Input & output

Input:
The first line contains the number of test cases T ($1 \leq T \leq 10$). For each test case, the first line contains the number of instructions N ($1 \leq N \leq 1,000$). Starting on the second line, an instruction is given on line N. After inputting N instructions, M-th document to be output will be given. M is a natural number less than or equal to 1,000. ID, title, and priority will be given separated by white space when inserting a document. Instructions are as follows:

Instruction	Printer Does
I *id title priority*	Insert a new document with given properties
P	Print ID and title of the top document, and pop it
T	Print ID and title of the top document without removing
S	Print the length of the queue
E	Print '1' if the queue is empty

Output:
For each test case, print the answer for each instruction, and ID, title of the M-th document to be output at the last. If there's no M-th document, *i.e.*, M is larger than printer size, print *"M > printer.size()"*. Figure 6 is an example.:

Input	Output
3	AAA DS
10	0
I AAA DS 1	4
T	AAA DS
I BBB DCN 3	CCC LinearAlgebra
I CCC LinearAlgebra 2	DDD Energy
I DDD Energy 4	back SecondPR
E	M > printer.size()
S	
I EEE Circuits 5	
P	
P	
2	
7	
I abcd Attendance 13	
I bbdie FinalExam 6	
I urgg FirstQuiz 5	
I dbwo4B FirstPR 1	
I go5G SecondQuiz 9	
I back SecondPR 7	
I ffbi11 QnA 2	
5	
1	
I ABC EmptyDoc 6	
2	

(a) Example 1

Figure 6: Example for part 2.

4 Submission

- **Due Date**: 24, Nov, 2023, 23:59:59
- **What to submit**:
  - Zip file with your student number (e.g., ”2023-12345.zip”) containing
  - ”PriorityQueue.hpp”, ”PriorityQueue.cpp”, ”part1_main.cpp”, ”part2_main.cpp”
  - Do not modify the file name!
  - Submit on Project#2 in Assignment tab.