Vietnam National University Ho Chi Minh City

University of Science

Faculty of Information Technology

CHALLENGE 1: PRIORITY QUEUE

USING LEFTIST HEAP

Course	**Data structures and algorithms**
Class	**22CLC02**
Lecturers	**Văn Chí Nam** **Lê Thanh Tùng** **Bùi Huy Thông** **Trần Thị Thảo Nhi**
Participants	**22127488 – Trương Thanh Toàn** **22127357 – Phạm Trần Yến Quyên** **22127123 – Lê Hồ Phi Hoàng**

HCMC, 2023

Mục lục

1 Introduction

1.1 Our choice

We chose ***leftist heap*** data structure to implement our priority queue due to its efficiency in operations (`insertion()`, `deletion()`, `merge()` and `peak()`) (visit 2.3.3 for more details).

1.2 Percentage of done works

Initialize medical rooms	100%
Coordinate patients for direct single-patient	100%
Coordinate patients for online registered patient	100%
View patient list from medical rooms	100%
Update medical room's wait list	100%

2 Research

In this section, we are going to implement a priority queue in four ways.

2.1 Array

2.1.1 Representation

- To represent a priority queue using an array, we can use a one-dimensional array and maintain the elements in priority order.
- The idea is to create a structure to store the value and priority of the element and then create an array of that structure to store elements.
- There are two types of priority queue representations of arrays: using othered array than insertion which takes O(n) time complexity and deletions with O(1) time complexity or using unordered arrays with insertion that takes o(1) time complexity and deletion takes o(n) time complexity.

```
struct Node {
    int data; // This data can be a string, a double, or a structure.
    int priority;
};

// Array PQ to store elements of the priority queue.
int maxsize = 100000;
Node *PQ = new Node[100000];
int size = -1; // Pointer to the last index
```

2.1.2 Operations

- **Insertion**: Add a new item into an existing PQ.
  + Input: A priority queue reference pointer PQ implemented using an array following the above convention, an item.

+ Algorithm:

Step 1. Checks if the size is equal to the max size. if equal, copy all elements of PQ into PQ_{tmp} with $maxsize = 2 * maxsize$ and delete array PQ and assign $PQ = PQ_{tmp}$

Step 2. Increase the $size$ by 1. Assign $PQ[size - 1] = value$

+ Example: Giving priority queue represented array is as next and size equal maxsize.Insert item 10 to it.

a_0	a_1	a_2	...	a_{size-1}
8	7	4	...	5

Step 1. Because $size == sizemax$ increase the size of PQ.

PQ:

a_0	a_1	a_2	...	a_{size-1}	a_{size}	...
8	7	4	...	5	–	...

Step 2. $a_{size} = 10$ and $size = size + 1$

a_0	a_1	...	a_{size-2}	a_{size-1}	a_{size}	...
8	7	...	5	10	–	...

– **Deletion**: Remove the highest-priority item from an existing PQ.

+ Algorithm:

Step 0 begin

Step 1. Find the position of the element with the highest priority.

Step 2. Shift the element one index before from the position of the element with the highest priority is found.

Step 3. Decrease the size of the priority queue by one

Step 4. end

+ Example: Giving priority queue represented array is as next, $size = 10$. The highest-priority item is 11.

a_0	a_1	a_2	a_3	...	a_9	a_{10}	...
8	11	4	9	...	5	–	...

Step 1. The position of the highest priority is 1.

Step 2. Copy a_2 to a_1, a_3 to a_2, ... a_9 to a_8.

a_0	a_1	a_2	...	a_8	a_9	a_{10}	...
8	4	9	...	5	5	–	...

Step 3. $size = 9 - 1 = 8$

a_0	a_1	a_2	...	a_8	a_9	a_{10}	...
8	4	9	...	5	–	–	...

– **Peak**: Return the index of the value of the most-priority item without removing it from an existing PQ. If the array is empty, return -1

+ Algorithm:

Step 0. Begin

Step 1. Check the array with empty. If the array is empty, $ind = -1$ and jumps to Step 4. Otherwise go to Step 2

Step 2 Declare $hightestPriority = a[0]$ and $ind = 0$.

Step 3. Loop i from 0 to the size of the array. if priority of $\texttt{i}^{th}$ > `highestPriority` then `highestPriority`= `priority` of $\texttt{i}^{th}$ and `ind` = `i`.

Step 4. return `ind`.

Step 5. end.

+ Example: Giving priority queue represented array is as next with size = 10. The highest-priority item is 11.

a_0	a_1	a_2	a_3	...	a_9	a_{10}	...
8	11	4	9	...	5	–	...

Step 1. the array is not empty; **Step** 2. $highestPriority = a_0 = 8$; **Step** 3. after traversing all items, `ind` = 1.

– **Merge**: Combine two existing PQs into one single PQ.

+ Input: 2 array priority queues PQ_1 and PQ_2 which follow the above convention, $size_1$ and $size_2$.
+ Output: $size$ - the size of merges array priority queue ($MergePQ$) and merged array
+ Algorithm:

Step 0. Begin.
Step 1. Declare $size = size_1 + size_2$ and merged array $MergePQ$ has $size$ item
Step 2. Copy $size_1$ items of PQ1 to the first $size_1$ items of PQ
Step 3. Copy $size_2$ items of PQ2 to the last $size_2$ items of PQ.
Step 4. Return $size$ and PQ
Step 5. End.

+ Example: Giving 2 PQ_1 and PQ_2, that is priority queue represented array are as below, the priority item is us value, the item that has – isn't an item of the priority queue, and $size_1 = 5$ and $size_2 = 3$.

PQ_1:

a_0	a_1	...	a_4	a_5	...
8	11	...	5	–	...

PQ_2:

a_0	a_1	a_2	a_3	...
7	20	17	–	...

Step 1. $size = size_1 + size_2 = 8$.
$MergePQ$:

a_0	...	a_4	...	a_7	a_8	...
0	...	0	...	0	–	...

Step 2.$MergePQ$:

a_0	...	a_4	...	a_7	a_8	...
8	...	5	...	0	–	...

Step 3. $MergePQ$:

a_0	...	a_4	...	a_7	a_8	...
8	...	5	...	17	–	...

Step 4. return $size = 8$ and $MergePQ$

2.1.3 Analysis

– `Delete()` must find the index of the highest priority O(N) and overwrite from that index position until N-1 is O(N-index) = O(N) so the total `delete()` is O(N). `peak()` must traverse all items of array, so the complexity `peak()` `merge()` just copies the items of 2 PQs so it also traverses all so it's O(N). `insert()` only the worst case is O(N) the rest is just O(1)

– The below table shows the time complexity of the above 4 operations.

– The above are the operations with the representation of the unordered array, the other way can be seen in this video Priority Queue Implementation using Array in C++ with example || By Studies Studio

Function	**Complexity**
Insertion	O(1)
Deletion	O(N)
Get the highest priority node	O(N)
Merge	O(N)

2.2 Binary heap

2.2.1 Representation

– According to Geeksforgeeks, a Binary Heap is a Complete Binary Tree that is used to store data efficiently to get the max or min element based on its structure. This property of Binary Heap makes them suitable to be stored in an array. The representation is done as:

+ The root element will be at Arr[0].

+ Below table shows indexes of other nodes for the i^{th} node, i.e., Arr[i]

Arr$[(i-1)/2]$	Returns the parent node
Arr$[(2*i)+1]$	Returns the left child node
Arr$[(2*i)+2]$	Returns the right child node

– Each item in the binary heap represents an item in the priority queue.

– An element with high priority is dequeued before an element with low priority. If two elements have the same priority, they are served according to their order in the queue.

```
struct Node {
    int data; // This data can be string, a double, or a structure...
    int priority;
};
Node *BinaryHeap = new Node[100000];
```

2.2.2 Operations

– **Insertion**: Add a new item into an existing PQ.

+ Algorithm:

Step 0. Begin.

Step 1. Checks if the size is equal to the max size. if equal, copy all elements of PQ into PQ_{tmp} with $maxsize = 2 * maxsize$ and delete array PQ and assign $PQ = PQ_{tmp}$.

Step 2. $PQ[size] = item$ and increase the $size$ by 1, and declare $index = size$.

Step 3. Swap the newly added item with the parent element if its priority is better than its parent and go back step 3, otherwise go to step 4.

Step 4 End.

+ Example: Giving PQ, that is priority queue represented binary max heap is as next and size equal maxsize. Insert item 85 to it.

a_0	a_1	a_2	...	a_5	a_6	...
90	80	70	...	10	–	...

Step 1. Because 10 < maxsize, go to Step 2

Step 2. $PQ[size] = PQ[10] = 85$, $ind = 6$, and $size = 6 + 1 = 7$.

a_0	a_1	a_2	...	a_5	a_6	...
90	80	70	...	10	85	–

Step 3. The parent item a_6 is a_2 and priority of a_2 < priority of a_6. (70 < 85)
Swap a_2 with a_6:

a_0	a_1	a_2	...	a_5	a_6	...
90	80	85	...	10	70	–

Same with a_2 and a_0. Because 85 < 90 r go to step 4 - end:

– **Deletion**: Remove the highest-priority item from an existing PQ.

+ Algorithm:

Step 0. Begin

Step 1. If the array is empty, go to Step 4. Otherwise, go to Step 2.

Step 2. Declare $array[0] = array[size - 1]$ and deincrease the $size$ by 1.

Step 3. Swap the first item with the children's item if children's priority is better than its priority and go back step 3. otherwise go to step 4

Step 4. End.

+ Example: Giving PQ, that is priority queue represented binary max heap is as below and size = 4.

a_0	a_1	a_2	a_3	a_4	...
90	85	70	55	45	–

Step 1. PQ is not empty. Go to step 2.

Step 2. $a_0 = a_9 = 10$ and size = 11 - 1 = 10

a_0	a_1	a_2	a_3	a_4	...
45	85	70	55	–	...

Step 3 *3.1.* Compare between a_1 and a_2 and a_3. Swap a_0 with a_1 as $85 > 45$ and $85 > 70$

a_0	a_1	a_2	a_3	a_4	...
85	45	70	55	–	...

3.2 Then with a_1 and a_3. $45 < 55$, swap a_1 with a_3.

a_0	a_1	a_2	a_3	a_4	...
85	55	70	45	–	...

– **Peak**: Return the value of the most-priority item without removing it from an existing PQ. Otherwise, return -1

+ Algorithm:

Step 0. Begin.

Step 1. If the array is empty, return -1. Otherwise, return 0.

Step 2. End.

+ Example: Giving PQ, that is priority queue represented binary max heap is as below, the priority item is its value, the item that has - isn't an item of the priority queue, $size = 11$.

a_0	a_1	a_2	a_3	a_4	...	a_9	a_{10}	...
90	85	70	40	80	...	55	45	–

Step 1. PQ isn't empty, so return 0.We can get max number by PQ[0] = 90.

– **Merge**: Combine two existing PQs into one single PQ.

+ Algorithm:

Step 0. Begin

Step 1. Create a new array with a size equal to the sum of the sizes of the two initial max heaps.

Step 2. Copy all elements from the two initial max heaps into the new array.

Step 3. Rearrange the new array into a max heap by applying the "Heapify" algorithm from the middle position of the array to the beginning.

Step 4.end

+ Example: Suppose you have two binary max heaps represented as arrays:

PQ_1:

a_0	a_1	a_2	a_3	...
10	8	6	-	...

PQ_2:

a_0	a_1	a_2	a_3	a_4	...
15	12	7	2	-	...

Step 1. Create a new array PQ with a size of $3 + 4 = 7$

Step 2. Copy the elements from the two initial max heaps into the new array:

a_0	a_1	a_2	a_3	a_4	a_5	a_6	a_7	...
10	8	6	15	12	7	2	-	...

Step 3. Using Heapify algorithm.

3.1 Begin from a_2, $7 > 6 > 2$ swap 6 with 7 (swap a_2 with a_5).

a_0	a_1	a_2	a_3	a_4	a_5	a_6	a_7	...
10	8	7	15	12	6	2	-	...

3.2 Then, a_1 with a_3 and a_4. $15 > 12 > 8$. We swap 15 with 8 (a_1 with a_3)

a_0	a_1	a_2	a_3	a_4	a_5	a_6	a_7	...
10	15	7	8	12	6	2	-	...

3.3 Then, a_0 with a_1 and a_2. $15 > 10 > 7$. We swap 15 with 10 (a_1 with a_0)

a_0	a_1	a_2	a_3	a_4	a_5	a_6	a_7	...
15	10	7	8	12	6	2	-	...

3.4 Finally, a_1 with a_3 and a_4.$12 > 10 > 8$. We swap 12 with 10 (a_1 with a_4)

a_0	a_1	a_2	a_3	a_4	a_5	a_6	a_7	...
15	12	7	8	10	6	2	-	...

2.2.3 Analysis

– Same as `insert()` using array only worst case is O(N) the rest is just O($log(N)$). because of the heap's refactoring mechanism, `delete()` is O(log N). `Peak()` only check arry with empty then return 0 or -1, so O(1). `merge()` just be concatenation and heapify, so its complexity is O(M+N)

– The below table shows the time complexity of the above 4 operations.

Function	**Complexity**
Insertion	O(Log N)
Deletion	O(Log N)
Get the highest priority node	O(1)
Merge	O(N + M)

2.3 Leftist heap

2.3.1 Representation

– According to Geeksforgeeks, a leftist heap (leftist tree), is a type of binary heap data structure used for implementing priority queues.

– It is a complete binary tree.

– In contrast to a binary heap, a leftist tree attempts to be very unbalanced. In addition to the heap property, leftist trees are maintained so the right descendant of each node has the lower s-value.

– A Leftist heap has the following characteristics:

+ The null path length (NPL) of a tree node is the length of the shortest path to a node with 0 children or 1 child. The NPL of a leaf is 0. The NPL of a `NULL` pointer is -1.
+ A leftist tree is a binary tree where at each node the null path length of the left child is greater than or equal to the NPL of the right child.
+ The right path of a node (e.g. the root) is obtained by following right children until a `NULL` child is reached.

– Each node in a leftist heap includes 4 components:

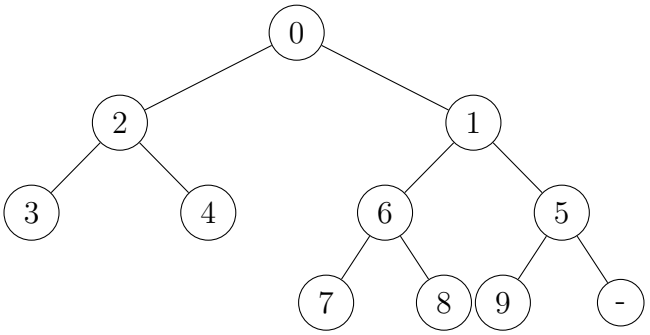
+ The `priority` member contains the priority of the node. in the priority queue.
+ The `data` member contains the **data** that has a priority `priority`.
+ The `left` child and the `right` child which are priority queue implemented using leftist heap data structure.

– In the range of this report, we will follow the convention that the low priority the node has, the small `priority` it contains.

– The implementation of a priority queue implemented using skew heap data structure in `C++` is shown as below:

```
struct Node {
    Data_Type data;
    // it could be an integer, a character, a string
    // or an abstract data type.

    int priority;
};
struct PQ {
    Node* root;
    int NPL;  // update the shortest path to NULL node
    PQ* left;
    PQ* right;
};
```



A leftist tree after inserted with elements from 0 → 9.

2.3.2 Operations

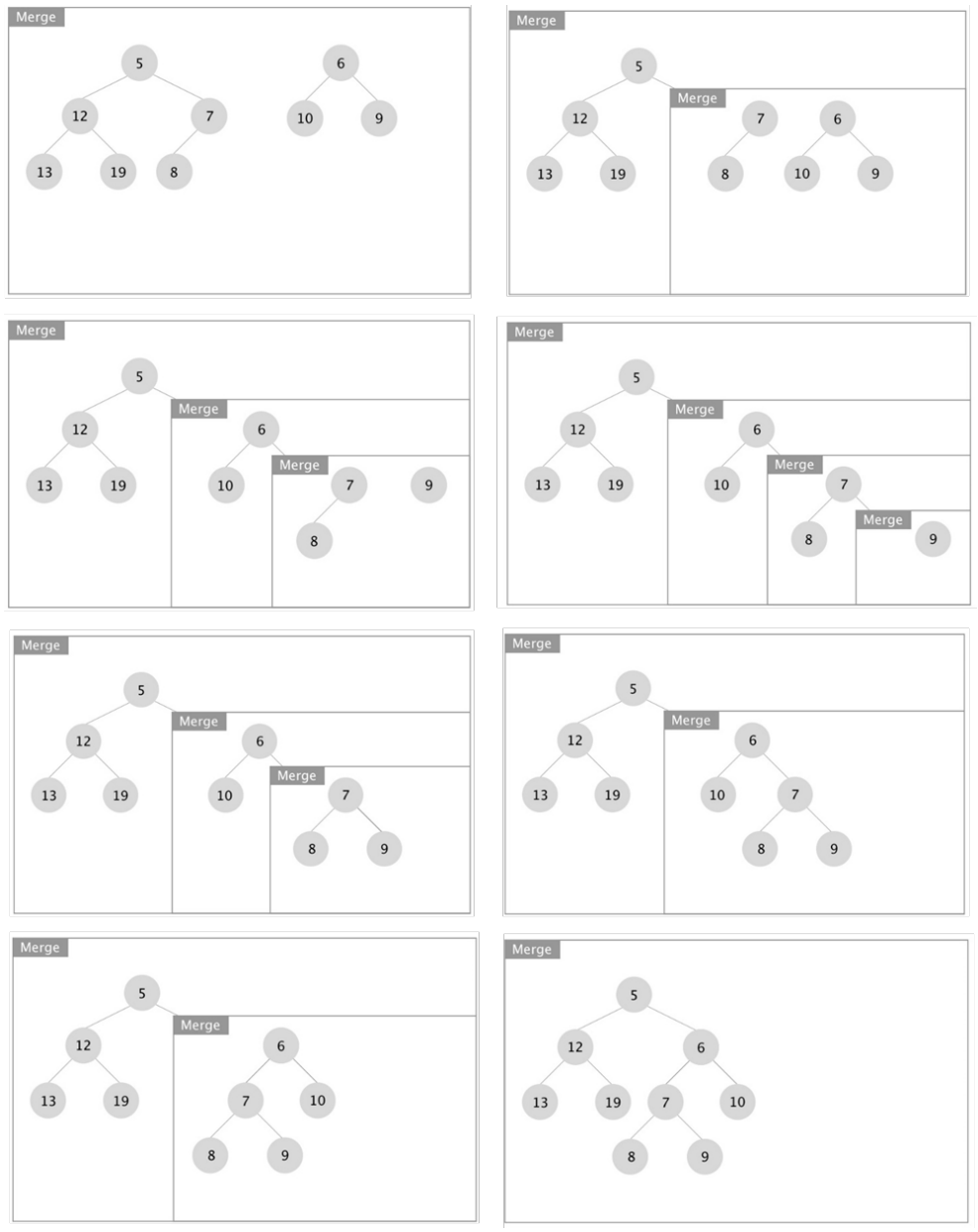
The main operation is `merge()`. `delete()` and `peak()` can be done by removing root and calling `merge()` for left and right subtrees.
`insert()` can be done by creating a leftist tree with that one key as the single node and calling `merge()` for given tree and tree with single node.

– **Merge**: Combine two existing PQs into one single PQ.

Algorithm:

+ Use the smaller root of the two PQs as the new root.
+ Place the smaller root PQ left subtree on the left of the new PQ.
+ Recursively merge right subtree with the other PQ.
+ Before returning from resursion:
  * Update the NPL of merged root.
  * Swap the left and right subtree be-

low the root (if needed) to maintain the original leftist heap characteristics of merged result.



Graphics by Pishigraphy - Own work, CC BY-SA 4.0

2.3.3 Analysis

– Because two operations `insert()` and `delete()` both use `merge()` operation as the main operator in order to complete the task, `insert()` and `delete()` do have the same time complexity to `merge()`, which is $O(\log N)$.

– The below table shows the time complexity:

Function	Complexity
Peak	O(1)
Delete	O(log N)
Insert	O(log N)
Merge	O(log N)

2.4 Skew heap

2.4.1 Representation

– A skew heap* (in this case a **min** skew heap) must meet the following requirements:

+ The root node contains the minimum value comparing to its children.

+ Each node has at most 2 possibly empty children which are also 2 skew heaps.

+ *Convention*: a tree with 0 or 1 node is a skew heap.

– In the range of this report, we will follow the convention that the low priority the node has, the small `priority` it contains.

– The implementation of a priority queue implemented using skew heap data structure in `C++` is shown on the right below:

```
struct PQ {
    int priority;
    Data_Type data;
    // it could be an integer,
    // a character, a string or
    // an abstract data type

    PQ* left;
    PQ* right;
};
```

2.4.2 Operations

– We just need to working on the `merge()` operation only.

+ Input: 2 priority queues `pq1` and `pq2` implemented using skew heap data structures follow the above convention.

+ Algorithm:

Step 1. If one of them is empty, returns the other.

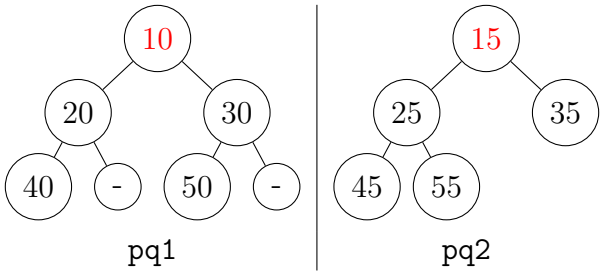
Step 2. If the `priority` of the root node of `pq1` is greater than the `priority` of the root node of `pq1`, swap `pq1` and `pq2`; otherwise, go to step 2.

Step 3. Swap `pq1.left` and `pq1.right` (swap the whole node including `priority`, `data`, `left` and `right`).

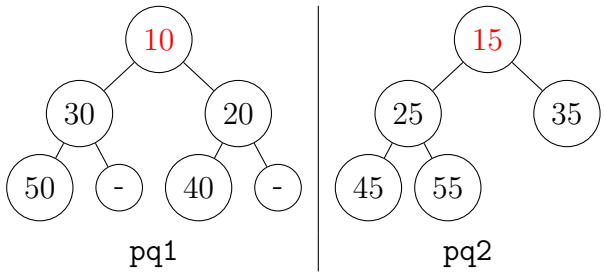
Step 4. We recursively merge `pq2` with `pq1.left` and assign the result to `pq1.left` (these 2 arguments must be passed to the function in this exact order `merge(pq2, q1.left)`).

Step 5. Finally return `q1`.

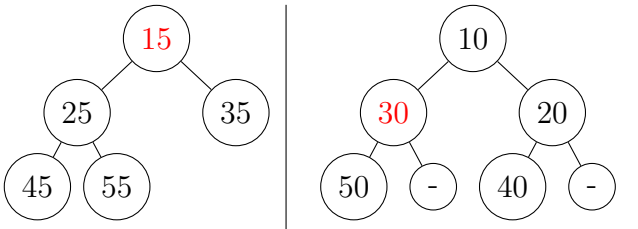
Example: `pq1` and `pq2` are as below, function call `merge(pq1, pq2)`. The tree that is passed to the function has its root colored red. The value shown in the node is the priority of the node in the queue.



`i)` If 1 of them is `NULL`, return the other one.
`ii)` `pq1.root` < `pq2.root`: do not swap these queues.
`iii)` Swap `pq1.left` and `pq1.right`:



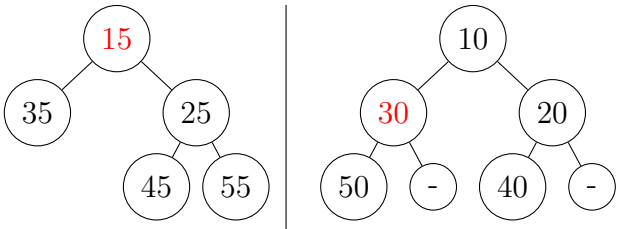
iv) Then call `pq1.left = merge(pq2, pq1.left)`



`pq2` a.k.a new `pq1` | `pq1.left` a.k.a new `pq2`

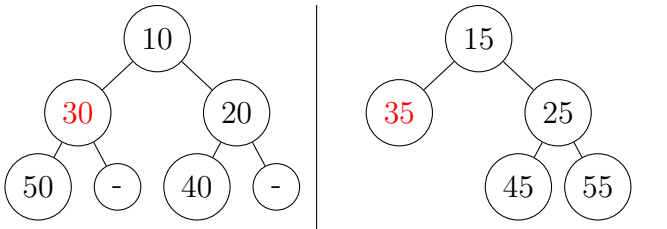
v) `pq1.root` < `pq2.root`: do not swap these queues.

vi) Swap `pq1.left` and `pq1.right`:



`pq2` a.k.a new `pq1` | `pq1.left` a.k.a new `pq2`

vii) Then call `pq1.left = merge(pq2,pq1.left)`.

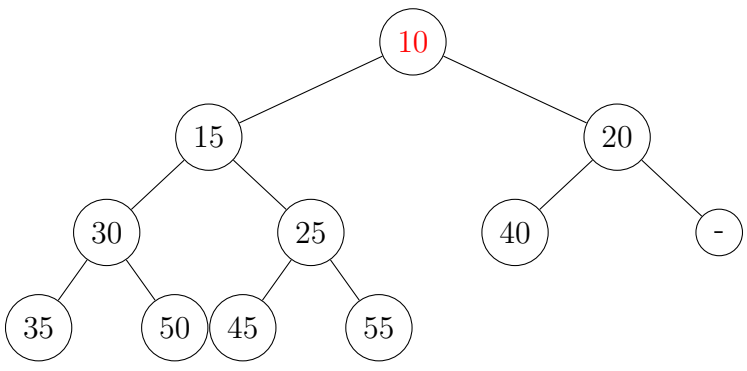


`pq2` a.k.a new `pq1` | `pq1.left` a.k.a new `pq2`

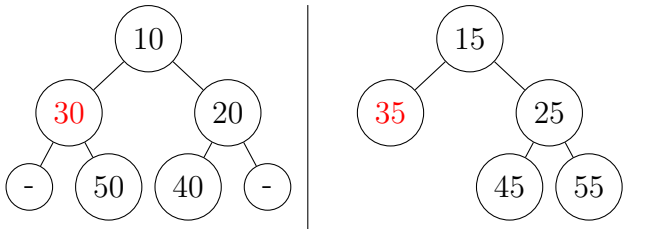
viii) `pq1.root` < `pq2.root`: do not swap these queues.

ix) Swap `pq1.left` and `pq1.right`:

xiii) Then go back to step `iii)`, we have:

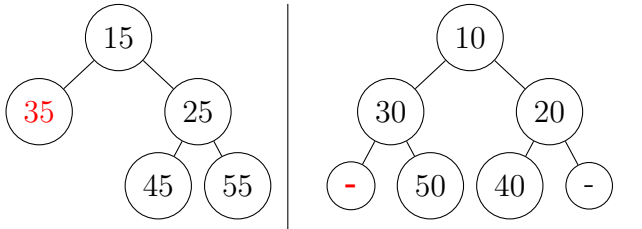


xiv) After all, return `q1` which is the result.



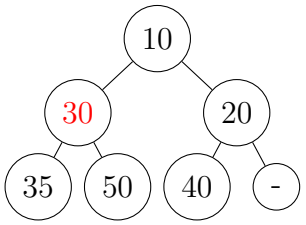
`pq2` a.k.a new `pq1` | `pq1.left` a.k.a new `pq2`

x) Then call `pq1.left = merge(pq2,pq1.left)`.



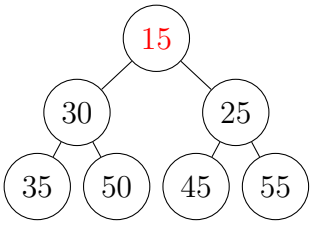
`pq2` a.k.a new `pq1` | `pq1.left` a.k.a new `pq2`

xi) Because the new `pq2` is empty, we return the new `pq1` (which is the node `35`). Then go back to step `ix)`, we have:



assign this tree to `q1.left` *at step* `vii)`.

xii) Then go back to step `vi.`, we have:



assign this tree to `q1.left` *at step* `iv)`.

– To insert a new item `item` to a priority queue `PQ`, we consider that `item` to be a priority queue and merge it with `PQ`.

– To delete an item, we just need to remove it from `PQ` and merge its 2 children together to get a new `PQ`.

– The item with the highest priority is the root node of the tree `PQ`. If we need to get the item with the highest priority, we just need to access to the root node of `PQ`, thus `peak()` function takes $O(1)$.

2.4.3 Analysis

– About `merge()`: time complexity $O(N \log N)$ and space complexity $O(N)$.

– Because two operations `insert()` and `delete()` both use `merge()` operation as the main operator in order to complete the task, `insert()` and `delete()` do have the same time complexity to `merge()`, which is $O(N \log N)$.

– The below table shows the time complexity of the above 4 operations.

Function	**Complexity**
Merge	O(N·logN)
Insertion	O(N·logN)
Deletion	O(N·logN)
Get the highest priority node	O(1)

3 Programming

3.1 Code structure

– The program has **4 sections**, each are divided by a big block of "//". Respectively:

– `structures` contains:

+ All the libraries which were used in the program.
+ 2 namespaces:
  * `ns_priority_queue` includes the struct `PriorityQueue` and its 6 operation functions.
  * `global_variables` includes 2 list of rooms (`regular_room` and `vip_room`).
+ 4 structures:
  * `Patient`: Basic information and specific priority type (Emergency, VIP, Old, Chirden).
  * `Node`: Structured accordingly to a leftist heap node (has an extra `npl` variable).
  * `PriorityQueue`: Holds the top patient and total patients in a `Room`.
  * `Room`: Basic information and amount of specific priority types.
+ 2 preprocessing instruction `#define`:
  * `CURRENT_YEAR`: current year which is 2023.
  * `MAX_EMERGENCY_PATIENT_IN_REGULAR_ROOM`: maximum number of emergency patients in each regular medical room. If all the regular medical rooms reach to this number, all the non-VIP emergency patients after that will become VIP.

– `auxiliary_functions` contains:

+ Prototypes and brief explanation of all the functions used in the program.

- `implementation` contains:
  + All implementations of declared functions in `auxiliary_functions` and `ns_priority_queue`.
  + Thorough comments on each functions indicating: special cases, actual inputs, etc, ...
- `main` contains:
  + Formatted to be able to run with command line arguments (take in command prompts, filenames, etc).

3.2 Librabries

Besides basic librabries `iostream`, `string`, `cstring` and `fstream`, we also use an auxiliary librabry named `<vector>` to store a list of medical rooms of each type separately `vector<Room*> regular_room` and `vector<Room*> vip_room`. Both of them are wrapped in the namespace `global_variables`.

3.3 Priority of patients

- Each patient consists of his/her information and their priority flags:

```
bool prior_ord_vip;
bool prior_ord_emergency;
bool prior_ord_old;
bool prior_ord_children;
bool prior_ord_normal;
int order;
```

- The first 2 conditions will always be checked, whereas the rest will pass through an if-else condition check (if one is met then the rest are skipped).
- If this patient was registered a V.I.P patients, `prior_ord_vip == true`; otherwise, `prior_ord_vip == false`.
- If this patient was registered an emergency patients, `prior_ord_emergency == true`; otherwise, `prior_ord_emergency == false`.

3.4 How does this program work?

- About the function `do_the_task(int argc, char* argv, bool print_new_patient)` which will be called from `main()` and do everything: pass command line arguments to this function to run the task and a boolean value to determine whether the new patient who was added will be displayed (only if the command is `New filename`).
- Run the executable file `hospital.exe`.
- On each turn, the program will read a command line from keyboard and separate them into command line arguments `int argc` and `char* argv[]`. Then these arguments will be passed to function `do_the_task()` and working on a corresponding process to produce a desired output.
- The command line must follow the given convention in order that the program can work precisely; otherwise, it will not do anything.
- If you want to stop the program, just enter `"Exit"`. The data set will be dellocated and the program will stop.
- In case the command is `New filename`: The file named `filename` will be opened.
  + If this file is unable to be opened, a notification will be displayed on the screen and user can enter a new command line.
  + If this file was successfully opened, the program will read each line in the file one-by-one and convert that

line to command line arguments `int file_argc` and `char* file_argv[]`. Then they will be passed to the function `do_the_task()` again with these new command line arguments and a boolean value `true` to display this new patient on the screen. The boolean value which is passed to this function is `true` if and only if the commmand `New filename` occurs; otherwise, it will be `false`.

3.5 How to choose an approriate room for a new patient?

There are 2 cases in choosing room for a patient: **VIP** – **REGULAR**.

1. **REGULAR**: To check whether or not the patient is a priority case is based on:
   (a) Priority cases (Emergency/Old/Children):
      i. Finding:
         (1) The minimum total amount of patients in the rooms.
         (2) The `min` total amount of priority cases in the rooms.
         (3) `%X` of total patients.
      ii. Check whether the room matches 2 of the first requirements (and the 3rd if it is an *Emergency* case):
         <1> Total patients of that room − (1) ≤ (3)
         <2> Total priority cases == (2)
         <3> Total emergency cases of that room +1 ≤ `MAX_EMERGENCY_PATIENT_IN_REGULAR_ROOM`.
         ⇒ { if `YES` : return this room.
           { if `NO` : put the patient into a **VIP** room and go to case **VIP**
   (b) If there are no priority cases:
   ⇒ Find and put the patient in the room with the minimum amount of patients.
2. **VIP**: The same as Regular but without checking the <3> condition in step `a.ii` if it's an Emergency case.

3.6 Update medical room's wait list

– Command line: `> Update␣room_ID␣number_of_finished_patients`

– Process:

+ While the list of patients in the given room is not empty and we haven't deleted enough `number_of_finished_patients`, we delete the highest priority patient in the room.
+ After that, to maintain the balanced number of patients in each room and differences among rooms, we rearrange all the patients in the list of room:
  <1> Put all the patients in a virtual room.
  <2> Then call each of them to choose a new room and put he/she in.

– Time complexity: $\Theta(n)$ (average case) with n is number of patients in the corresponding collection of rooms and $O(N)$ (worst case) where N is number of patients in the hospital.

4 References

- Array:
  + Pradiptamukherjee. 2022 - Priority Queue using array in C++
- Binary Heap:
  + Geekforgeeks - Priority Queue using Binary Heap
- Leftist Heap:
  + Danny Sleator - Parallel and Sequential Data Structures and Algorithms (Fall 2013)
  + Sanfoundry - C++ Program to Implement Leftist Heap
  + Geeksforgeeks - Leftist Tree / Leftist Heap
- Skew Heap:
  + Skew Heap
  + Skew Heap Visualization
  + Skew heap - Chalmers