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

Project #3
Due : 19 Dec 2022, 23:59:59

INSTRUCTIONS

- 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 Linux tutorial. The version of **g++** and **c++** should be **7+** 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.
- 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
  - Any other non-compliance with given rules will be also deducted.

0 Priority Queue

A priority queue is a container adaptor that provides constant time lookup of the largest (by default) element. We define some common interfaces for a priority queue in class *PriorityQueue*. All of them are declared as a virtual function.

The class template *std::optional* manages an optional contained value, i.e. a value that may or may not be present. Function *extract_min()* removes the minimum key from the priority queue and returns the key if it exists. If priority queue is empty, it returns *std::nullopt*. Note that *std::optional* is only available from C++17. In the following, we go into detail about a specific priority queue implementation, Fibonacci heap.

1 Fibonacci Heap (70 points)

Fibonacci heap is one efficient way to implement priority queue. Fibonacci heap consists of multiple trees, where each tree satisfies min-heap property. For the Fibonacci heap, the find-minimum operation takes constant ($\theta(1)$) amortized time. The insert and decrease key operations also work in constant amortized time. Deleting an element (most often used in the special case of deleting the minimum element) works in $O(\log n)$ amortized time, where n is the size of the heap. This means that starting from an empty data structure, any sequence of a insert and decrease key operations and b delete operations would take $O(a + b \log n)$ worst case time, where n is the maximum heap size.

Wikipedia : Fibonacci heap

The roots of all trees are linked using a circular doubly linked list. The children of each node are also linked using such a list. In this assignment, you should implement with C++ smart pointers, *std::shared_ptr* and *std::weak_ptr*. Two of them are useful when representing multiple owners of the object. The difference between *std::shared_ptr* and *std::weak_ptr* is that *std::shared_ptr* affects the reference counter while *std::weak_ptr* doesn't. The links below would be helpful to understand two smart pointers.

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

Project #3
Due : 19 Dec 2022, 23:59:59

---

cppreference.com : std::shared_ptr
cppreference.com : std::weak_ptr

The basic operations of Fibonacci heap is described as follows.

- *get_min()*: We maintain a pointer to the root containing the minimum key. Thus, the operation is trivial.
- *insert()*: It works by creating a new heap with one element and make it link to another roots. The pointer indicating the minimum key **must be kept updated**.

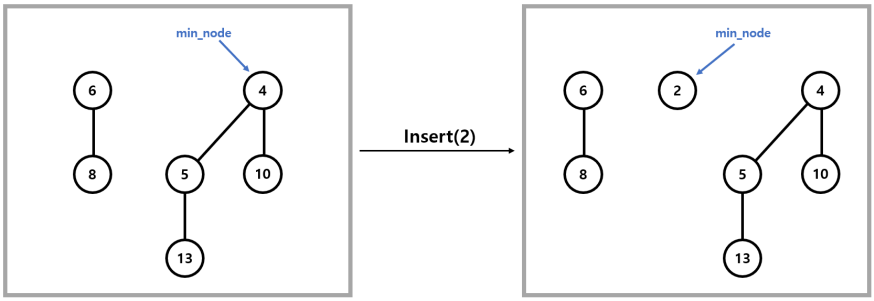

Figure 1: Insertion of new key

- *extract_min()*: Extracting the minimum node is performed with two stages, extracting the node itself and consolidation. First we take the root containing the minimum element and remove it (and return its key). Its children will become roots of new trees. After extracting the minimum node, you have to consolidate the heap that is the work we have delayed in the insertion. The specific procedure will be described in the next part.

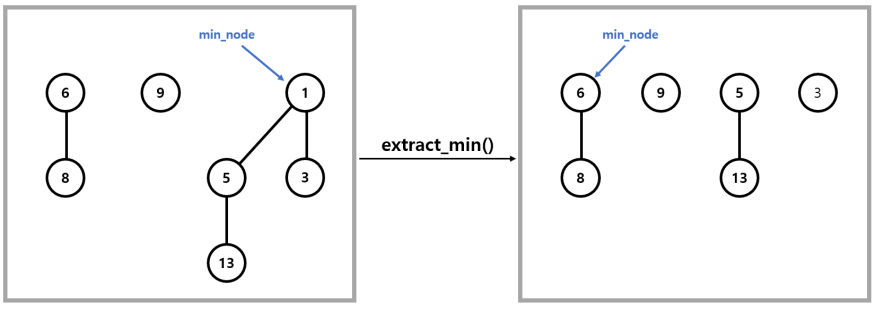

Figure 2: Extraction of the minimum key

- *consolidate()*: Consolidation is performed **only after** extracting the min_node. The purpose of consolidate operation is to make the heap with the trees that have a distinct degree. The degree represents the number of children. If two nodes x and y have the same degree, you have to make one (e.g., y) to the child of x.

To perform, prepare an array with a specific length ($\lfloor \log_\phi n \rfloor + 1$). Here, the constant number ϕ is equal to $\frac{1+\sqrt{5}}{2}$, the golden ratio. Next, traverse the root list starting from min_node. While traversing, make array[d] point to the node if array[d] is empty. Here, let d be the degree (the number of children) of each root node. If the element of the array is already occupied, it means that there are two trees with the same degree. You should make it one as making one node be the child of another(*merge()*). Note that consolidation is included in the operation of *extract_min()*.

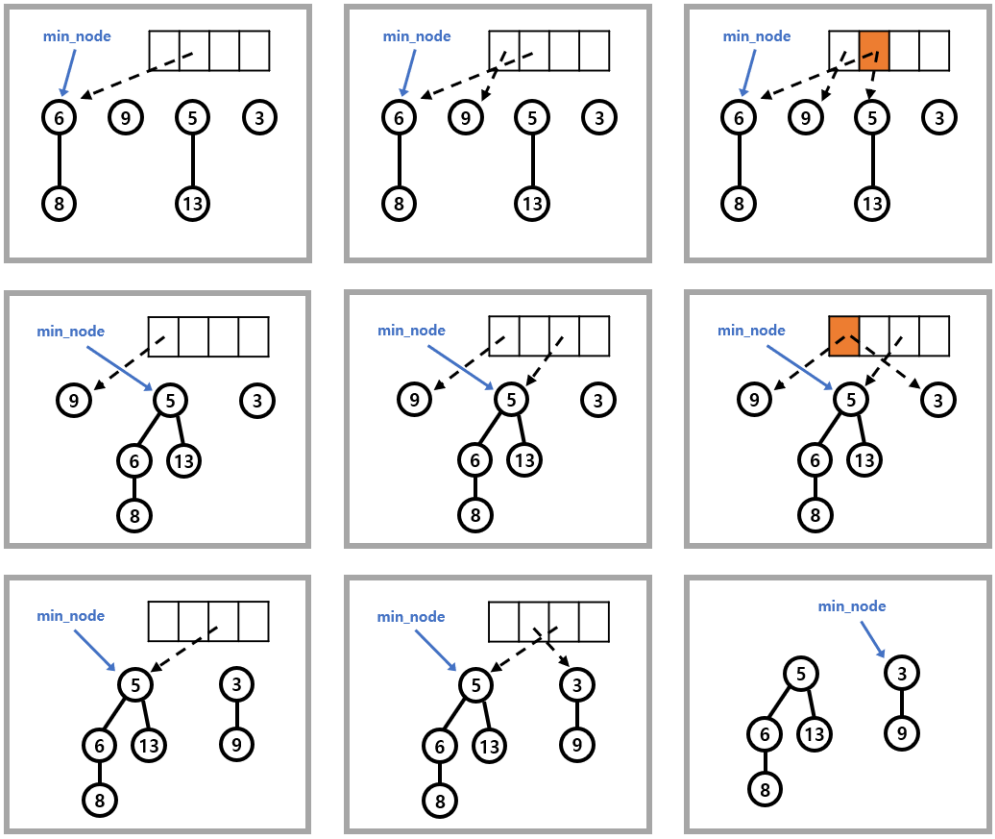

Figure 3: Consolidation of Fibonacci heap

- *merge()*: It takes roots of two trees, and make one root as a child of another. Do not forget to satisfy the **min-heap property**.

- *decrease_key()*: It takes the node, decreases the key and if the heap property becomes violated (the new key is smaller than the key of the parent), the node is cut from its parent. If the parent is not a root, it is marked. If it has been marked already, it is cut as well and its parent is marked. We continue upwards until we reach either the root or an unmarked node. Now we set the minimum pointer to the decreased value if it is the new minimum.

- *cut()*: For easier implementation, cut operation is separated into two. *cut()* removes the designated

node from the current position and add it to the root list. If the node is marked, then mark it as false.

- *recursive_cut()*: If the parent of the node x is not null then follow the following steps.
  1. If x is unmarked, mark the node.
  2. Else, call *cut(x)* and *recursive_cut(parent of x)*.

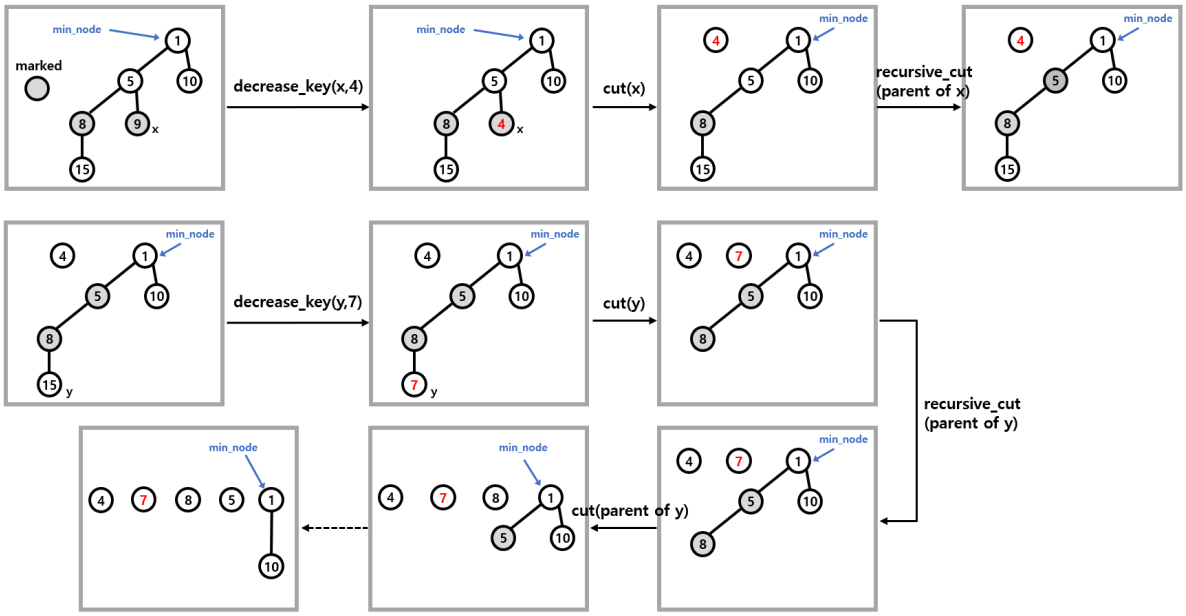

Figure 4: Decreasing the key of the node

- *delete()*: It can be implemented simply by decreasing the key of the element to be deleted to minus infinity (or just less than minimum key), thus turning it into the minimum of the whole heap. Then we call *extract_min()* to remove it.

2 Dijkstra's Algorithm (30 points)

Dijkstra's algorithm is an algorithm for finding the shortest paths between nodes in a graph. Pseudo code of the algorithm is as follows.

Algorithm 1 Dijkstra's Algorithm

Require: Graph G, Source src
Ensure: The shortest paths from s to all other nodes in G

```
1:  dist[src] ← 0
2:  create vertex priority queue Q
3:  for all v ∈ V[G] do
4:     if v is not src then
5:        dist[v] ← +∞
6:        previous[v] ← undefined
7:     end if
8:     Q.insert(v, dist[v])
9:  end for
10: while Q is not empty do
11:    u ← Q.extract_min()
12:    for all edge (u, v) outgoing from u do
13:       if dist[u] + E(u, v) < dist[v] then
14:          dist[v] ← dist[u] + E(u, v)
15:          previous[v] := u
16:          Q.decrease_key(v, dist[u] + E(u, v))
17:       end if
18:    end for
19: end while
```

All you need to do is referring to this code and use **Fibonacci heap** as priority queue to make Dijkstra algorithm work. To do so, first create an array of Fibonacci nodes for the given graph. The return type of function *dijkstra_shortest_path()* is *std::unordered_map* of vertices with corresponding tuple of previous vertex and the distance from the source. Please refer to the link below for details.

cppreference.com : std::unordered_map