UNIVERSITY OF SCIENCE

VIETNAM NATIONAL UNIVERSITY - HO CHI MINH CITY

PROJECT REPORT
Skew Heap

DATA STRUCTURE AND ALGORITHM

Theory Lecturer	Nguyen Thanh Phuong	
Instructors	Bui Huy Thong	
	Nguyen Ngoc Thao	
Students	Nguyen Quoc Huy	21127511
	Phu Thanh Nhan	21127382
	Le Hoang Sang	21127158
	Vo Thanh Tu	21127469

Contents

1 Introduction

1.1 Group members and contact information

There are 4 members in our group.

- 21127511 - Nguyen Quoc Huy - nqhuy21@clc.fitus.edu.vn
- 21127382 - Phu Thanh Nhan - ptnhan21@clc.fitus.edu.vn
- 21127158 - Le Hoang Sang - lhsang21@clc.fitus.edu.vn
- 21127469 - Vo Thanh Tu - vttu21@clc.fitus.edu.vn

1.2 Work assignments

Our work assignments are as follows:

- Phu Thanh Nhan in charge of organizing and refining the report
- Le Hoang Sang
- Vo Thanh Tu
- Nguyen Quoc Huy

1.3 Project organization

Main file and subordinates:

- main.cpp
- SkewHeap.cpp
- SkewHeap.h

Library used (Allowed library for the course):

- **iostream** used for manipulating input/output streams

1.4 Programming note

This project uses the C++ 11 standard

We use struct `Node` to implement HeapNode, it includes:

- `int data`: to store data as 32-bit integer.
- `Node* left, right`: pointer to the Node's childrend.
- Constructor with default value.
- `Node* Clone()`: helps get a clone of Node.

We use struct `SkewHeap` to implement Skew Heap data structure. Assume that the top of this SkewHeap is the smallest number in this SkewHeap, it includes:

- `Node* root`: the root of heap.
- `int heap_size`: store heap size, reduce time to get size.
- Constructor with no parameter and with an array as parameter.
- `Size(), Empty()`: using `heap_size` to return the result.
- `Push(int data)`: inseart a node has `data` into this `SkewHeap`.
- `Top()` return the smallest number in SkewHeap.
- `Pop()` remove Top element and doesn't return anythings.
- `Print()` Print out the SkewHeap to the output stream. Data is organized similarly a binary tree, which each line is a level, from top to bottom, using BFS algoritm.
- `Clone`: use `Node.Clone()` to help them clone the Node.
- `PrintPreOrder(), PrintInOrder(), PrintPostOrder()` show the path with three kind of traversal.
- `Clear()`: clear all node, avoid memory-leaking.

We use `Merge()` to merge two node, one of two node become a new root and no nodes are created.

With `MergeTree()`, we clone two SkewHeap and use a new SkewHeap to store everything. With this function, the result do not depend on the old nodes.

2 Overall information

2.1 Idea

The idea for the Skew Heap is based on the ability to perform not just a single operation but a sequence of operations.

2.2 History

The Skew Heap creators, also known as a self-adjusting Heap, are Daniel Dominic Sleator and Robert Endre Tarjan in February 1986. The Skew Heap by itself is a form of the leftist Heap. Many data structures are designed to have the running time per operation for the worst-case scenario, the smallest possible. However, we need to look further into the typical applications of the data structures, which are multiple operations rather than one.

2.3 Applications

Application for Skew Heap: Skew Heap offer faster merge time as they are a particular case of leftist trees. A Skew Heap is not structurally restricted. This can make the height of the tree non-logarithmic. Real-life application: SIM user information storage

2.4 Variants

Up unitl the time of writing this report, we have not found any variants of Skew Heap.

3 Comparisons with other Heap Structures

3.1 Basic Heap vs Skew Heap

3.1.1 Condition satisfaction

Basic Heap:

- **Min Heap**: Node parents are always smaller than their children (descedants)
- **Max Heap**: Node parents are always bigger than their descedants

Skew Heap, if X is a node with L and R are its left and right children, then:

- $X.value \leq L.value$
- $X.value \leq R.value$
- Swapping children at every step.

3.1.2 Basic operations

Basic Heap, there are three primary operations:

- Insertion
- Deletion
- Merge

Skew Heap:

- Adding a value to a Skew Heap is like merging a tree with one node with the original tree
- Removing the first value in a Heap can be accomplished by removing the root and merging its child subtrees.
- In summary, there is only one main operation, merging.

3.1.3 Experimental results and conclusion

Experimental results
Conclusion

- Skew Heaps are advantageous because they merge more quickly than Binary Heap. There are no structural constraints, so there is no need to guarantee that the Tree's height is logarithmic. Only two conditions must be satisfied: The general Heap order must be enforced.
- Binary Heaps must be traversed in a specific orpder: left-node-right, node-left-right, etc when adding a node, you must traverse the Tree to find the right location.
- The main operation in Skew Heaps is Merge.

3.2 Leftist Heap vs Skew Heap

In this segment, the Skew Heap seems to suffer more because the Leftist Heap is an improved version of the Skew Heap, thanks to a more conditional swap than the unconditional Skew Heap.

P/S: We'll not go far on Leftist Heap. Just show the disadvantage and advantages between Skew Heap and its improved version.

3.2.1 Similarities

- The time complexity of both Heaps is $O(\log n)$ in all operations.
- The main operation of both Heaps is merge.
- Due to the no structural constraints, there is no guarantee that the height of the Skew Heap is logarithmic. A Skew Heap is just a self-adjusting form of a Leftist Heap.
- On the other hand, the Leftist Heap may be unbalanced, but every node has an s-value (or rank or distance) which is the distance to the nearest leaf. That makes Leftist Heap seem better.

Here is some example to show the difference between 2 Heap: We have an array with five elements: 10, 20, 30, 40, and 50, and put it into 2 Heaps.

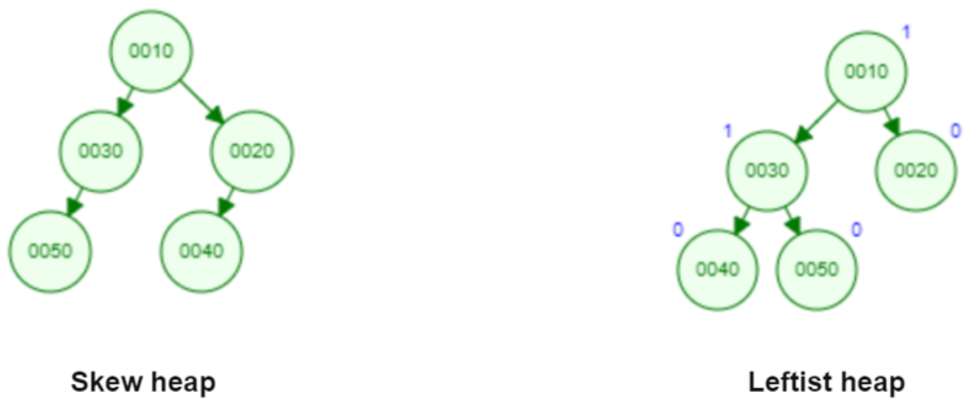


Figure 1: Visualization of Skew Heap vs Leftist Heap

As you can see, the Skew Heap tends to balance itself after each step, while Leftist Heap has an index above each node called null path length to show the distance between that node and the null node.

In that way, the leftist seems to work more efficiently than the Skew Heap.

3.2.2 Experimental results and conclusion

4 Step-by-step descriptions

4.1 A few notes

Conditions when working with Skew Heap:

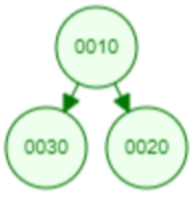
- Heap can be just one node and also a Skew Heap
- Skew Heap is somewhat similar to min-Heap, where the parent nodes are all smaller than the child nodes (The top node is always the smallest node).

4.2 Push

The purpose of Push(Insertion) in Skew Heap is to add a node in the correct root

1. After comparing all the correct nodes in the right branch, put it in the necessary place. If it's the smallest node in the Heap, now its left child would be the Heap.
2. Begin from the adding place, and swap the branch where that node occurs with the left branch.
3. Move to node parent, then continue to swap like step 2.
4. Continue step 3 until there is no more node parent.

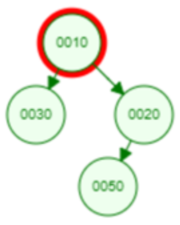
To exemplify, here is an example: We have a Heap with 3 nodes



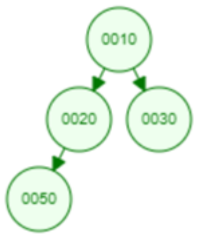
Now when adding node 50 into Skew Heap:

- First, add the node in the correct position of root → right
  - Compare $10 < 50$
  - Move to root = root → right, compare $20 < 50$
  - Nove to root = root → right now. This node is null, so we have to insert node 50 in this place

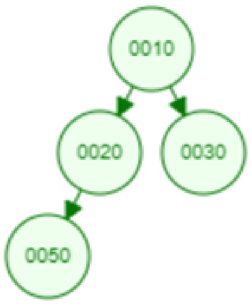
- Second, swap the right and left branches
  - Swap left and right nodes of 20, then move to the parent of node 20, which is now node 10



  - Continue swap left and right nodes of 10
- Continue swapping between 2 branches until there is no more node parent



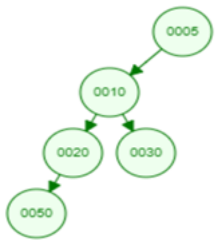
Example of swapping node smallest into Skew Heap, now we get the previous Heap



Add node 5:

- As we know, this is the smallest node in the Heap, so put the Heap at the right branch of node 5.
- Then, swap the right child with the left child(the left child is null before the swap).
- Thus, in simplicity, what we have to do when inserting the node have the smallest key value in the Heap is put the Heap at the left branch of that node, and the correct child is null.

Thus we have the final result:

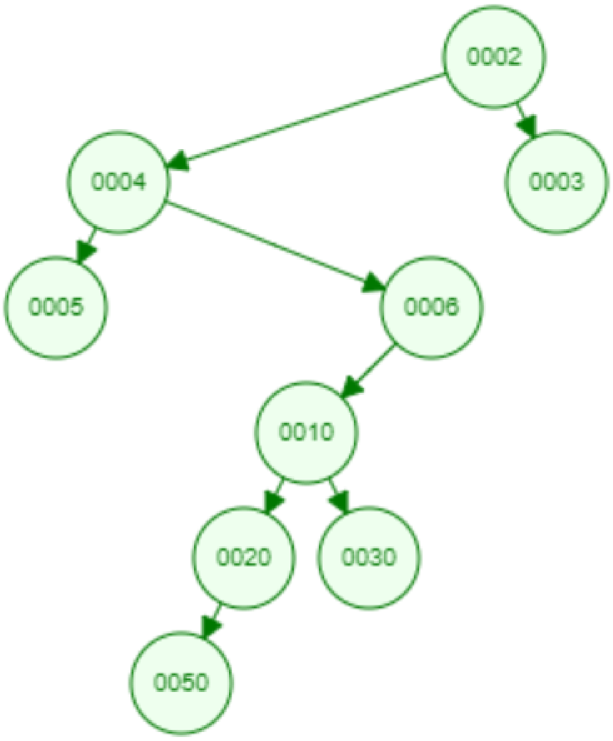


4.3 Pop

The purpose when Pop(deleting) a node in Heap:

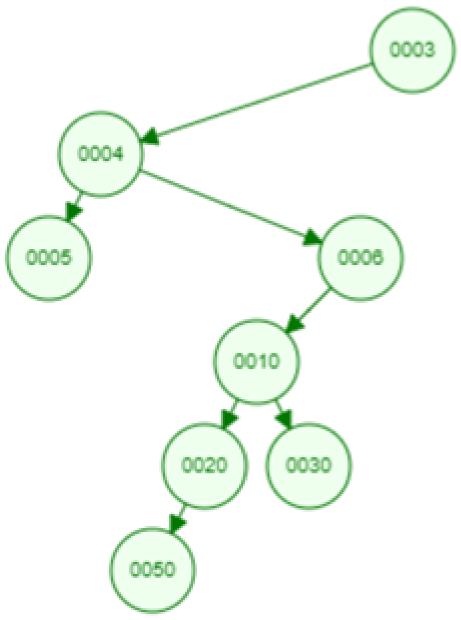
1. The deletion is just for moving the smallest node (the top node).
2. After eliminating the smallest node, there are 2 child nodes of the eliminated node. Compare these 2 nodes. The smaller node is the top node, and its child is decided based on its eliminated parent node. For example, if the smaller node compared is the right child of the parent node, then its child node is placed in the right branch.
3. If the top node contains no more than 1 node, then delete that node, and the new Heap would be the non-null child.
4. If the top node has a left null child after deletion, swap the right branch with the left branch and make sure the left node always exists.

To exemplify, here is an example: Delete the top of this particular Heap:

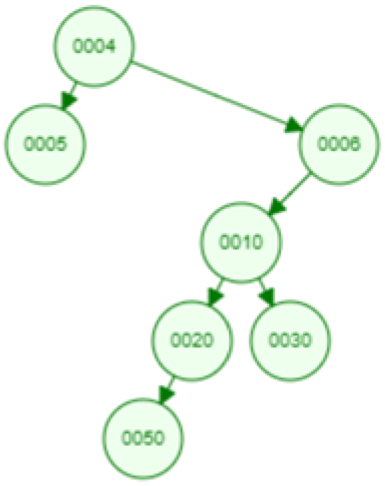


- As we know, the smallest node of the Heap is 2, then eliminate that node
- Then we have 2 child nodes, 4 and 3, compare these nodes, and 3 is the chosen.
- Look again at the original Heap. Node 3 is the RIGHT child of the old top node.
- Now, put the left branch of the old top node (node 4) at the RIGHT child of node 3 (new top node).

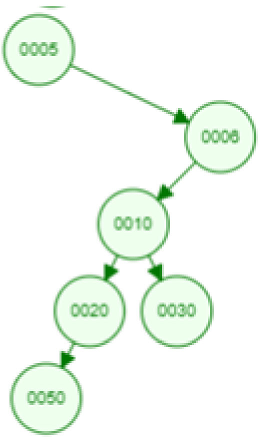
We have the following result:



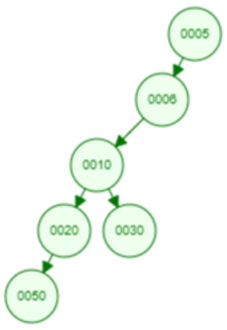
Here are some continued deletions of this Heap:
Removing node 3:



Removing node 4:



We have a situation: The left child of node 5 is now null. Our purpose is to ensure that the left child always contains a value, so we swap the left branch with the right branch. Thus we have the final result:



5 Complexity evaluations

Time complexity:

5.1 Merge

Best-case: $O(1)$ when there is no more left root
Worst-case: $O(n)$ when the root left is not over yet

5.2 Size

Time complexity in all cases: $O(1)$

5.3 IsEmpty

Time complexity in all cases: $O(1)$

5.4 Push

Best-case: $O(1)$ when the Heap is empty
Worst-case: $O(n)$ When the Heap is not empty:

5.5 Pop

Best-case: $O(1)$ when the Heap is empty, or the Heap contains 1 node
Average-case: $O(logn)$

5.6 Top

Time complexity in all cases: $O(1)$

5.7 Print

Time complexity in all cases: $O(n)$

5.8 Clone

Best-case: $O(1)$ when the Heap is empty
Worst-case: When the Heap is not empty: $O(n)$

6 References

Skew Heap function: `https://www.geeksforgeeks.org/skew-Heap/`
Skew Heap theory: `https://en.wikipedia.org/wiki/Skew_Heap`
Authors of Skew Heap: `https://www.cs.cmu.edu/~sleator/papers/Adjusting-Heaps.html`
Leftist Heap: `https://iq.opengenus.org/skew-Heap/`
Operations on Skew Heap: `https://www.wisdomjobs.com/e-university/data-structures-tu`
`skew-Heap-7243.html`
Visualization screenshot in Step-by-step section, Comparison section and Complexity section: `https://www.cs.usfca.edu/~galles/visualization/SkewHeap.html`

We appreciate your reading!