International Institute of
Information Technology, Hyderabad

Algorithmic Theory

Data Structures, Algorithms and Competitive Programming

College Notes

Maintainers: Animesh Sinha, Gourang Tandon, Bhuvanesh Sridharan, Yogottam Khandelwal, Shreeya Pahune, Aman Kumar Singh.

February 15, 2019

Chapter 1

Heaps

1.1 Question Patterns

1.1.1 Generic Types

- Find the kth Minimum or Maximum. This can also be on arrays or trees with online insertion or deletion.
- Priority Queue questions. Typically greedy algorithms best implemented this way. (eg. Dijkstra)

1.1.2 Specific Illustrations

When not to use Heaps *Question: Find the kth-Maximum sum of all subarrays of any given array.* This question demostrates that when we have **k-sorted elements appended to our queue of k-maximum elements** at one go, it's better to use an actual **Sorted array and the Merge Function for Updates**.

Lazy Deletion Heaps cannot search, so heaps cannot delete. A simple solution for this is to maintain another heap of all deleted elements, and if the actual heap and deleted heap have the same top element, keep popping them out. We only care about the top element, so we can be Lazy in the deletion of the elements lower down in the tree.

1.2 Elementary Theory

1.2.1 Basic Operations

Heapification A **O(n)** algorithm exists for Heapification. It uses the standard sift-down procedure, but starts by correcting the lowest layers first. So start at the end of the array and move back.

Insert A **O(log n)** algorithm is used to insert. We push the element at the end of the array and sift-up or sift-down the element till the element is at a valid locale.

Delete For a **O(log n)** deletion algorithm, we must swap the last element with the element to be deleted, sift-up/down the former last element, and pop-out the last element. This is only possible if we have a refernce to the element, search and delete take O(n) time to perform on heaps.

Minimum/Maximum Minimum on a MinHeap takes **O(1)** time, and so does Maximum on a MaxHeap.

1.2.2 Implementation Code

```
#include "BinaryMinHeap.h"

BinaryMinHeap::BinaryMinHeap(const vector<int> &list
   ) : BinaryTreeArray(list) {
    for (long i = this->data.size() - 1; i >= 0; i
        --) {
        this->siftDown((unsigned) i);
    }
}

BinaryMinHeap::BinaryMinHeap(const BinaryMinHeap &
   heap) : BinaryTreeArray(heap) {
    ;
}

void BinaryMinHeap::siftDown(unsigned long pos) {
    // Handling cases where Element does not exist /
         Does not have either or both it's children.
    if (pos >= this->data.size()) return;
    int lCh = INT32_MAX, rCh = INT32_MAX;
    if (this->lChild(pos) < this->data.size()) lCh =
         this->data[this->lChild(pos)];
    if (this->rChild(pos) < this->data.size()) rCh =
         this->data[this->rChild(pos)];
    // Recursive Sifting down and Heapification
    if (lCh < this->data[pos] || rCh < this->data[
        pos]) {
        if (lCh < rCh) {
            this->swap(pos, this->lChild(pos));
            siftDown(this->lChild(pos));
        } else {
            this->swap(pos, this->rChild(pos));
            siftDown(this->rChild(pos));
        }
    }
}

void BinaryMinHeap::siftUp(unsigned long pos) {
    if (pos == 0) return; // This is the Root
       Element
    if (this->data[this->parent(pos)] > this->data[
       pos]) {
        this->swap(pos, this->parent(pos));
        this->siftUp(this->parent(pos));
    }
}

void BinaryMinHeap::insert(int val) {
    this->data.push_back(val);
    this->siftUp(this->data.size() - 1);
}

void BinaryMinHeap::remove(unsigned long pos) {
    this->swap(this->data.size() - 1, pos);
    this->data.pop_back();
    this->siftUp(pos);
    this->siftDown(pos);
}

vector<int> BinaryMinHeap::sort() {
    vector<int> result;
    BinaryMinHeap copy(*this);
    while(!copy.data.empty()) {
        result.push_back(copy.min());
        copy.remove(0);
    }
    return result;
}

int BinaryMinHeap::min() {
    return this->data[0];
}
```