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 10, 2019

Chapter 1

Heaps

1.1 Basic Operations

1.1.1 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.

1.1.2 Sample 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
        while (this->data[this->parent(pos)] < this->
           data[pos]) {
            this->swap(pos, 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];
    }
```

1.2 Binomial Heaps

1.3 Fibonacci Heaps