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.

March 26, 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 $\mathbf{O(n)}$ 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

```
#ifndef CODE_HEAP_H
#define CODE_HEAP_H

#include <iostream>
#include <vector>
#include <cstdint>
#include <functional>

using namespace std;

template <class Type>
class BinaryHeap {
protected:
    function<bool (Type, Type)> comparator;

    vector<Type> data;
    void siftDown(unsigned long pos) {
        if (pos >= this->data.size()) return;
        bool lChildExists = this->lChild(pos) < this
           ->data.size(), rChildExists = this->
           rChild(pos) < this->data.size();
        if ((lChildExists && comparator(this->data[
           this->lChild(pos)], this->data[pos]))
                || (rChildExists && comparator(this
                   ->data[this->rChild(pos)], this->
                   data[pos]))) {
            if (!rChildExists || (lChildExists &&
                    comparator(this->data[this->
                       lChild(pos)], this->data[this
                       ->rChild(pos)]))) {
                this->swap(pos, this->lChild(pos));
                siftDown(this->lChild(pos));
            } else {
                this->swap(pos, this->rChild(pos));
                siftDown(this->rChild(pos));
            }
        }
    }
    void siftUp(unsigned long pos) {
            if (pos == 0) return; // This is the Root
                Element
            if (comparator(this->data[pos], this->data[
                this->parent(pos)])) {
                 this->swap(pos, this->parent(pos));
                 this->siftUp(this->parent(pos));
            }
        }
        inline unsigned long parent(unsigned long val) {
            return (val - 1) / 2;
        }
        inline unsigned long lChild(unsigned long val) {
            return 2 * val + 1;
        }
        inline unsigned long rChild(unsigned long val) {
            return 2 * val + 2;
        }
        void swap(unsigned long x, unsigned long y) {
            this->data[x] ^= this->data[y];
            this->data[y] ^= this->data[x];
            this->data[x] ^= this->data[y];
        }
    public:
        explicit BinaryHeap(const vector<Type> &list,
            function<Type (Type, Type)> heapComparator =
            less<>()) {
            this->comparator = heapComparator;
            for (auto i : list) this->data.push_back(i);
            for (long i = this->data.size() - 1; i >= 0;
                 i--) {
                 this->siftDown((unsigned) i);
            }
        }
        explicit BinaryHeap(function<Type (Type, Type)>
            heapComparator = less<>()) {
            this->comparator = heapComparator;
            this->data = vector<Type>();
        }
        BinaryHeap(const BinaryHeap &heap) {
            this->comparator = heap.comparator;
            for (auto val : heap.data) this->data.
                push_back(val);
        }
        void insert(int val) {
            this->data.push_back(val);
            this->siftUp(this->data.size() - 1);
        }
        void remove(unsigned long pos) {
            this->swap(this->data.size() - 1, pos);
            this->data.pop_back();
            this->siftUp(pos);
            this->siftDown(pos);
        }
        Type top() {
            return this->data[0];
    }
    vector<Type> sort() {
        vector<Type> result;
        BinaryHeap copy(*this);
        while(!copy.data.empty()) {
            result.push_back(copy.top());
            copy.remove(0);
        }
        return result;
    }
};

#endif //CODE_HEAP_H
```

Chapter 2

Handling Range Queries

2.1 Square Root Decomposition

2.1.1 Motivating Examples

Example 1.1

Question: Given and array, support update and query operation for number of elements less than k in the given range.
Solution: Maintain sorted vector of each block of the square-root decomposition. Search over all blocks and find lowerbound(k), the sum of these counts gives the answer in $O(\sqrt{n}log(n))$. For update, just sort again, using insertion sort you get $O(\sqrt{n})$.

2.2 Mo's Algorithm

Example 2.1

Question: Given an array, find the number of elements distinct elements in range (l, r).
Sort the queries first by the starting block, then by the end position. Maintian a frequency array of all elements currently between right and left pointers. For each of the $O(\sqrt{n})$ blocks, the start pointer moves atmost $O(\sqrt{n})$ times back and forth in the block, adding and deleting elements. For each block the right pointer only goes forward adding in elements, $O(\sqrt{n})$ blocks each taking $O(n)$ time.

Example 2.2

Question: Given an array, find the f(s)*f(s)*s for all distinct s in range (l, r), where f(s) is the frequency of s.
Same Algorithm, and same frequency array as above, just find $f(s)*f(s)*s$ instead of $\delta(f(s))$ as above.

Chapter 3

Dynamic Programming

3.1 Matrix Exponentiation

Example 1.1

Number of ways to construct an array starting in X, and ending in Y, with no two adjacent elements are the same.

$$dp[i] = \begin{bmatrix} dp[i][\text{CLASH}] \\ dp[i][\text{CLEAN}] \end{bmatrix} = \begin{bmatrix} 0 & 1 \\ k-1 & k-2 \end{bmatrix} \times dp[i-1]$$