```
#include <iostream>
#include <vector>
#include <random>
#include <chrono>
#include <future>

using namespace std;

typedef vector<int> Data;
const int N = 1 << 26;

/**
 * @class - Heaper
 * Maintains a heap of vector elements in data and its sum in the vector
intermediate
 * Contains Properties n, data, interior
 * n - length of data
 * data - input vector of int
 * intermediate - vector of int to store pair sum - intermediate nodes of the heap
 */
class Heaper {
    public:
    /**
     * @brief Construct a new Heaper object
     *
     * @param data
     */
    Heaper(const Data *data) : n(data->size()), data(data) {
        interior = new Data(n-1);
    }

    virtual ~Heaper() {
        delete interior;
    }

    protected:
    int n;
    const Data *data;
    Data *interior;

    /**
     * @description Returns the value at node i of heap
     *
     * @param i-  node index
     * @return int - value at node index i
     */
    int value(int i) {
        if(isLeaf(i)) {
            return data->at(i - (n - 1));
        } else {
            return interior->at(i);
        }
    }

    /**
     * @description Returns the index of the parent of the node at given index
     *
     * @param i - current node index
     * @return int - index of parent
     */
    static int parent(int i) {
        return (i - 1) / 2;
    }

    /**
     * @description Returns the index of the left child of the node at given index
     *
     * @param i - current node index
     * @return int - index of left child
     */
    static int left(int i) {
        return (2 * i) + 1;
    }

    /**
     * @description Returns the index of the right child of the node at given index

     *
     * @param i - current node index
     * @return int - index of right child
     */
    static int right(int i) {
        return (2 * i) + 2;
    }

    /**
     * @description Returns the if the node is a leaf node
     *
     * @param i - current node index
     * @return bool - is node a leaf
     */
    static bool isLeaf(int i) {
        return (i >= (N - 1));
    }
};


/**
 * @class - SumHeap
 * Dereived from Heaper
 * Contains Methods calcSum and calcPrefix
 */
class SumHeap : public Heaper {
public:
    /**
     * @brief Construct a new Sum Heap object
     * Asynchronously calls calcSum to fill the intermediate vector with sum
values, thus completing the heap
     *
     * @param data - input vector
     */
    SumHeap(const Data *data) : Heaper(data) {
        auto handle = async(launch::async, &SumHeap::calcSum, this, 0, 1);
    }

    /**
     * @description - returns the value at index node of the heap
     *
     * @param node - index of element
     * @return int - value at the index
     */
    int sum(int node=0) {
        return value(node);
    }

    /**
     * @description - Calculates prefixSum of the input based on LadnerFischer
method
     * Runs asynchronously until level 4 of the heap
     *
     * @param prefixes - output vector
     */
    void prefixSums(Data *prefixes) {
        auto handle = async(launch::async, &SumHeap::prefixSum, this, prefixes,0,
0, 1);
    }
private:
    /**
     * @description - actual private method that implements the ladner fischer
algorithm recursively
     *
     * @param prefixes - output vector
     * @param node - index of the heap to process in the next recursion
     * @param from_left - prefix of the node
     * @param level - level of the recursion
     */
    void prefixSum(Data *prefixes, int node, int from_left, int level) {
        if (isLeaf(node)) {
            prefixes->at(node - (N - 1)) = from_left + value(node);
            return;
        }

        if (level <= 3) {
            auto handle1 = async(launch::async, &SumHeap::prefixSum, this,
prefixes, left(node), from_left, level + 1);
            auto handle2 = async(launch::async, &SumHeap::prefixSum, this,
prefixes, right(node), from_left + sum(left(node)), level + 1);
        } else {
            prefixSum(prefixes, left(node), from_left, level + 1);
            prefixSum(prefixes, right(node), from_left + sum(left(node)), level +
1);
        }

    }

    /**
     * @description - Calculates pairwise sum and fills the intermediate vector
recursively
     *
     * @param i - index of the node to process in the next recursion
     * @param level - level of the recursion
     */
    void calcSum(int i, int level) {
        if(isLeaf(i)) return;

        if(level <= 3) {
            auto handle1 = async(launch::async, &SumHeap::calcSum, this, left(i),
level + 1);
            auto handle2 = async(launch::async, &SumHeap::calcSum, this, right(i),
level + 1);

        } else {
            calcSum(left(i), level + 1);
            calcSum(right(i), level + 1);
        }

        interior->at(i) = value(left(i)) + value(right(i));
    }
};



int main() {
    Data data(N, 1);  // put a 1 in each element of the data array
    data[0] = 10;
    Data prefix(N, 1);

    // start timer
    auto start = chrono::steady_clock::now();

    SumHeap heap(&data);
    heap.prefixSums(&prefix); // Calculate prefix Sums

    // stop timer
    auto end = chrono::steady_clock::now();
    auto elpased = chrono::duration<double,milli>(end-start).count();

    int check = 10;
    for (int elem: prefix)
        if (elem != check++) {
            cout << "FAILED RESULT at " << check-1;
            break;
        }
    cout << "in " << elpased << "ms" << endl;
    return 0;
}
```