Hand in 4

Opgave 1

Using pen and paper, illustrate how a min-heap is built from the array

$$[59,\ 44,\ 79,\ 17,\ 54,\ 32,\ 31,\ 12,\ 7,\ 4,\ 1]$$

Using hepify. You must illustrate all steps of the heapify process. Then illustrate what happens if remove is called on the resulting heap, and then what happens if insert(2) is called on the heap before the remove.

Heapify

Vi skal starte med at finde den mindste interne node, det vil altså sige den sidste node der har mindst ét barn. Dette gør vi ved brug af følgende formel:

$$Sidste\ interne\ node\ =\ \frac{n}{2} - 1$$

Vi kan tælle at vi har 11 noder så vi får:

$$Sidste\ interne\ node\ =\ \frac{11}{2} - 1\ =\ 4,5\ \approx\ 4$$

Derfor er sidste interne node altså index 4 der indeholder værdien 54

Siden vi kører efter min-heap princippet hvor hver forældre er mindre en dets barn skal vi altså sammenligne 54 med sit barn

Vi finder børnene ved brug af formlen:

$$venstre\ barn\ =\ i \times 2 + 1$$
$$højre\ barn\ =\ i \times 2 + 2$$

Ved 4 er det derfor:

$$venstre\ barn\ (i = 4)\ =\ 4 \times 2 + 1 = 9\ (4)$$

$$højre\ barn\ (i\ =\ 4)\ =\ 4 \times 2 + 2\ =\ 10(1)$$

Derfor har vi altså børnene 4 og 1:

Vi sammenligner nu børnene med forældrene, i dette tilfælde vil det mindste tal derfor være 1, og ifølge min-heap så skal vi altså bytte elementerne på index i=4 og i=10, så vi får nu.

$$[59,\ 44,\ 79,\ 17,\ 1,\ 32,\ 31,\ 12,\ 7,\ 4,\ 54]$$

Nu skal vi gå et skridt op i træet og derfor går vi til i = 3 (17).

Her gør vi det samme og kigger på børnene:

$$venstre\ barn\ (i\ =\ 3)\ \ =\ \ 3\ \times\ 2\ +\ 1\ =\ 7(12)$$

$$højre\ barn\ (i\ \ =\ \ 3)\ =\ \ 3\ \times\ 2\ +\ 2\ \ =\ \ 8(7)$$

Vi sammenligner med dets børn og i dette tilfælde indeholder index 8 det mindste element 7, så vi bytter mellem index index 3 og index 7:

Så vi får følgende array:

$$[59,\ 44,\ 79,\ 7,\ 1,\ 32,\ 31,\ 12,\ 17,\ 4,\ 54]$$

Vi gør nu det samme for index 2 (79).

Vi finder børnene:

$$venstre\ barn\ (i\ =\ 2)\ \ =\ \ 2\ \times\ 2\ +\ 1\ =\ 5(32)$$

$$højre\ barn\ (i\ \ =\ \ 2)\ =\ \ 2\ \times\ 2\ +\ 2\ \ =\ \ 6(31)$$

Vi sammenligner og her indeholder index 6 det mindste element 31, så vi bytter elementerne på index 2 og 6 og får:

$$[59,\ 44,\ 31,\ 7,\ 1,\ 32,\ 79,\ 12,\ 17,\ 4,\ 54]$$

Vi gentager for index 1 (44)

$$venstre\ barn\ (i\ =\ 1)\ \ =\ \ 1\ \times\ 2\ +\ 1\ =\ 3(7)$$

$$højre\ barn\ (i\ \ =\ \ 1)\ =\ \ 1\ \times\ 2\ +\ 2\ \ =\ \ 4(1)$$

Vi sammenligner igen og her indeholder index 4 det mindste element 1, så vi bytter elementerne på index 1 og index 4:

$$[59,\ 1,\ 31,\ 7,\ 44,\ 32,\ 79,\ 12,\ 17,\ 4,\ 54]$$

Nu tager vi index 4(44) da

$$venstre\ barn\ (4\ =\ 1)\ \ =\ \ 4\ \times\ 2\ +\ 1\ =\ 9(4)$$

Vi bytter derfor index 4 og 9 og får

$$[59,\ 1,\ 31,\ 7,\ 4,\ 32,\ 79,\ 12,\ 17,\ 44,\ 54]$$

Vi tager nu index 0(59)

Vi sammenligner børnene:

$$venstre\ barn\ (i\ =\ 0)\ \ =\ \ 0\ \times\ 2\ +\ 1\ =\ 1(1)$$

$$højre\ barn\ (i\ \ =\ \ 0)\ =\ \ 0\ \times\ 2\ +\ 2\ \ =\ \ 2(31)$$

Her indeholder index 1 det mindste element 1, så vi bytter index 0 og index 1 og får derfor.

$$[1,\ 59,\ 31,\ 7,\ 4,\ 32,\ 79,\ 12,\ 17,\ 44,\ 54]$$

Vi skal nu fortsætte med det tal vi flyttede 59 som er på index 1

$$venstre\ barn\ (i\ =\ 1)\ \ =\ \ 1\ \times\ 2\ +\ 1\ =\ 3(7)$$

$$højre\ barn\ (i\ \ =\ \ 0)\ =\ \ 1\ \times\ 2\ +\ 2\ \ =\ \ 4(4)$$

vi bytter derfor index 4 og index 1 og får:

$$[1,\ 4,\ 31,\ 7,\ 59,\ 32,\ 79,\ 12,\ 17,\ 44,\ 54]$$

Vi gør det samme for index 4 (59) :

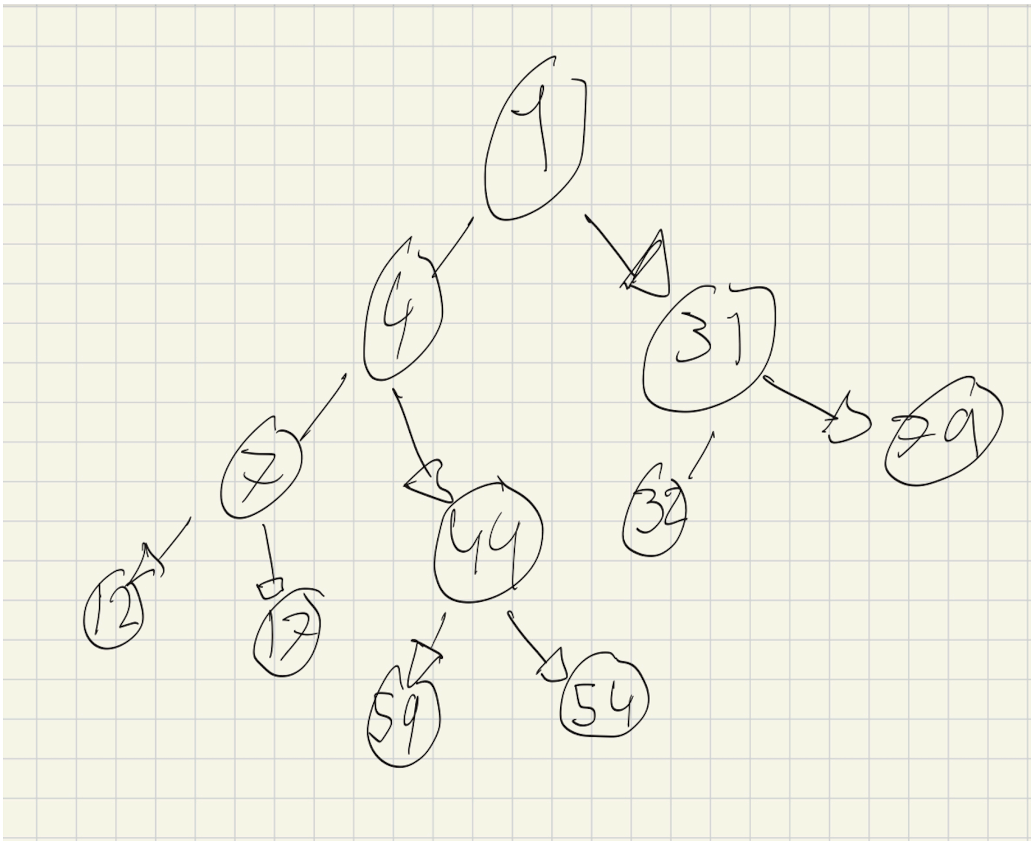
$$venstre\ barn\ (i\ =\ 4)\ \ =\ \ 4\ \times\ 2\ +\ 1\ =\ 9(44)$$

$$højre\ barn\ (i\ \ =\ \ 4)\ =\ \ 4\ \times\ 2\ +\ 2\ \ =\ \ 10(54)$$

Vi bytter derfor index 9 og index 4 og får:

$$[1,\ 4,\ 31,\ 7,\ 44,\ 32,\ 79,\ 12,\ 17,\ 59,\ 54]$$

Nu følger vi altså reglerne for min-heap og har følgende heap:



remove()

Hvis vi kalder remove(), så skal vi altså fjerne den mindste værdi, dette vil altså sige roden for et min-heap. Det første vi skal gøre er altså at kopiere den sidste værdi i arrayet og minimere størrelsen med 1, så vi får.

$$[54,\ 4,\ 31,\ 7,\ 44,\ 32,\ 79,\ 12,\ 17,\ 59]$$

Nu er min-heap strukturen ødelagt, derfor skal vi altså som næste step genoprette strukturen

Derfor skal vi altså sammenligne roden med dens børn:

$$venstre\ barn\ (i = 0) = 0 \times 2 + 1 = 1(4)$$

$$højre\ barn\ (i = 0) = 0 \times 2 + 2 = 2(31)$$

Derfor skal vi altså nu bytte index 1 og index 0(roden):

$$[4,\ 54,\ 31,\ 7,\ 44,\ 32,\ 79,\ 12,\ 17,\ 59]$$

Nu gør vi det samme med index 1:

$$venstre\ barn\ (i = 1) = 1 \times 2 + 1 = 3(7)$$

$$højre\ barn\ (i = 1) = 1 \times 2 + 2 = 4(44)$$

Vi bytter med det mindste element som i dette tilfælde er index 3 og får:

$$[4,\ 7,\ 31,\ 54,\ 44,\ 32,\ 79,\ 12,\ 17,\ 59]$$

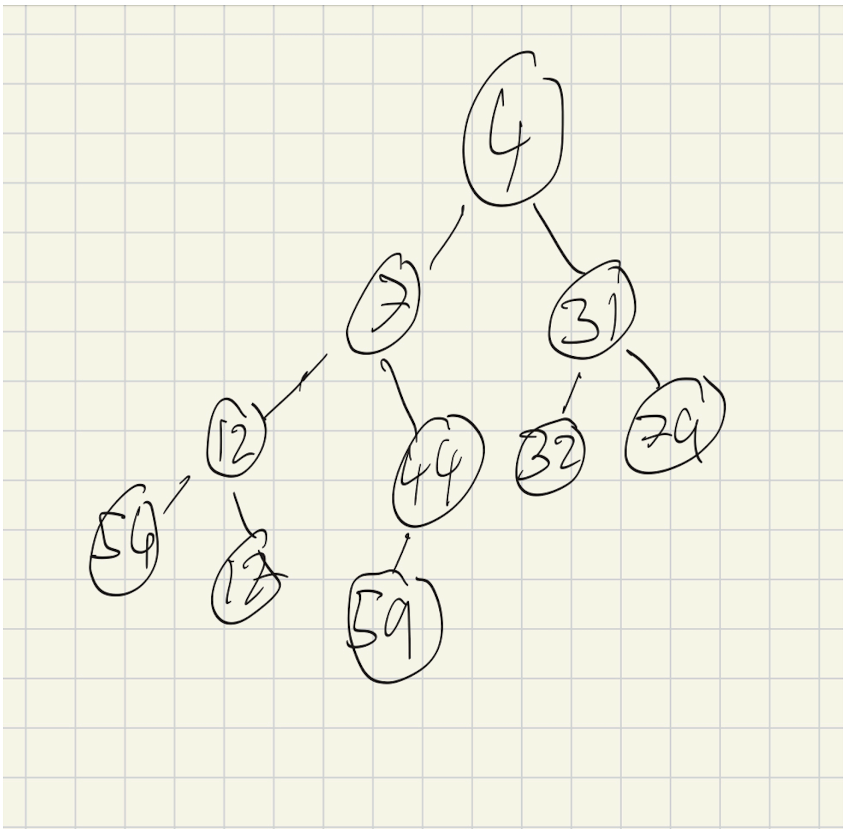
Nu gør vi det samme med index 3:

$$venstre\ barn\ (i = 3) = 3 \times 2 + 1 = 7(12)$$

$$højre\ barn\ (i = 3) = 3 \times 2 + 2 = 8(17)$$

Dette vil altså sige vi bytter med det mindste element som i dette tilfælde er index 7 med en værdi på 12 så vi får:

$$[4,\ 7,\ 31,\ 12,\ 44,\ 32,\ 79,\ 54,\ 17,\ 59]$$



Add(2) and remove()

Nu skal vi gå tilbage til det det første min-heap inden remove.

$$[1,\ 4,\ 31,\ 7,\ 44,\ 32,\ 79,\ 12,\ 17,\ 59,\ 54]$$

Vi skal nu insert(2). Vi indsætter et tal ved at smide det ind bagerst i arrayet. Så vi får altså:

$$[1,\ 4,\ 31,\ 7,\ 44,\ 32,\ 79,\ 12,\ 17,\ 59,\ 54, 2]$$

Vi skal nu sammenligne det nylige indsatte element med dens forældre og fortsætte indtil forældrene er mindre end eller lig med det nye element.

Forældre kan findes med følgende formel:

$$Forældre\ =\ (i\ -\ 1)/2$$

Vi følger nu dette, vi har indsat et element på plads 11 så vi får:

$$Forældre\ =\ (11\ -\ 1)/2\ =\ 5$$

Dette vil altså sige plads/index 5= 32. Dette er mindre end 2, derfor bytter vi og får:

$$[1,\ 4,\ 31,\ 7,\ 44,\ 2,\ 79,\ 12,\ 17,\ 59,\ 54, 32]$$

Vi gør nu det samme igen:

$$Forældre\ =\ (5\ -\ 1)/2\ =\ 2$$

Det vil altså sige plads/index 2 = 31. Det er mindre end 2, derfor bytter vi 2 og 5 og får:

$$[1,\ 4,\ 2,\ 7,\ 44,\ 31,\ 79,\ 12,\ 17,\ 59,\ 54, 32]$$

Vi gør det nu for plads 2:

$$Forældre\ =\ (2\ -\ 1)/2\ =\ 0$$

Vi sammenligner roden med index. Roden er mindre, derfor lader vi det blive som det er.

$$[1,\ 4,\ 2,\ 7,\ 44,\ 31,\ 79,\ 12,\ 17,\ 59,\ 54, 32]$$

Vi skal nu kalde remove og vi skal fjerne roden:

Det første vi gør er at bytte det sidste element i arrayet med roden og minimere arrayet:

$$[32,\ 4,\ 2,\ 7,\ 44,\ 31,\ 79,\ 12,\ 17,\ 59,\ 54]$$

Nu sammenligner vi roden med dets børn :

$$venstre\ barn\ (i\ =\ 0)\ \ =\ \ 0\ \times\ 2\ +\ 1\ =\ 1(4)$$

$$højre\ barn\ (i\ \ =\ \ 0)\ =\ \ 0\ \times\ 2\ +\ 2\ \ =\ \ 2(2)$$

Vi bytter med det mindste element og får som er plads/index 2:

$$[2,\ 4,\ 32,\ 7,\ 44,\ 31,\ 79,\ 12,\ 17,\ 59,\ 54]$$

Nu gør vi det samme for index 2:

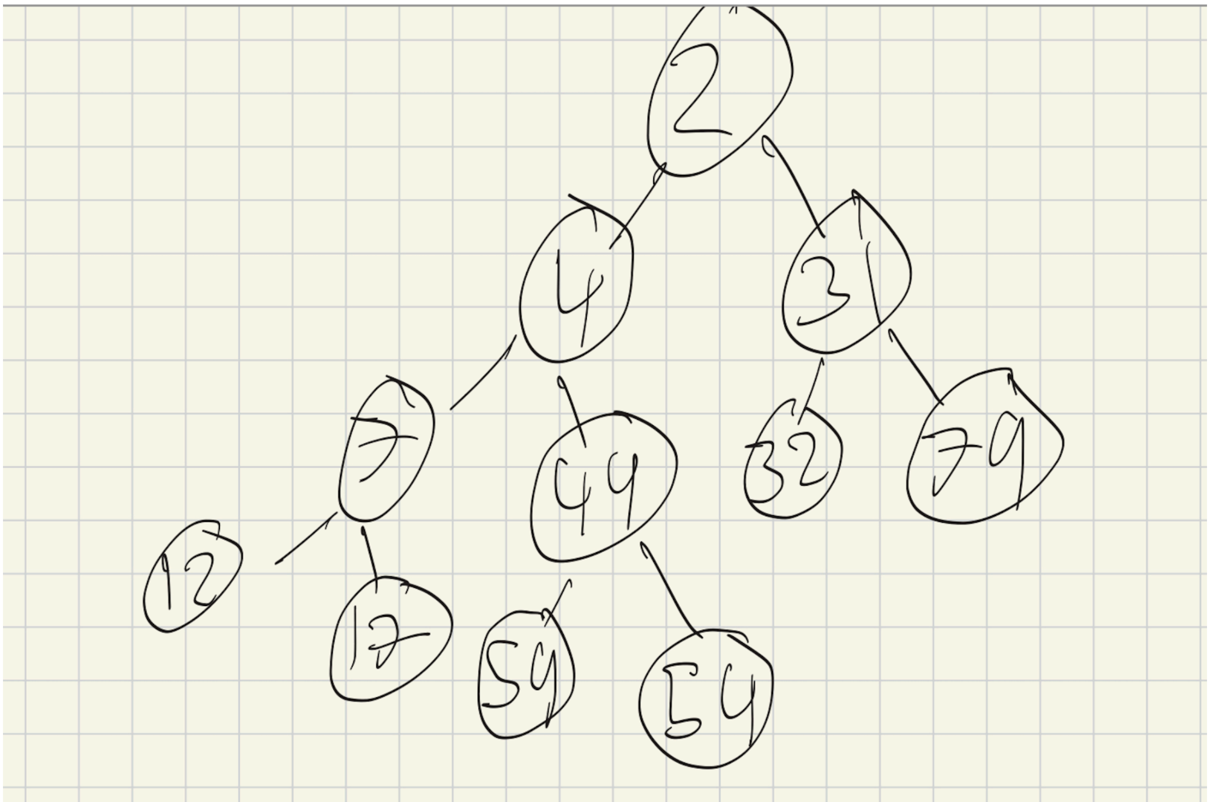
$$venstre\ barn\ (i\ =\ 2)\ \ =\ \ 2\ \times\ 2\ +\ 1\ =\ 5(31)$$

$$højre\ barn\ (i\ \ =\ \ 2)\ =\ \ 2\ \times\ 2\ +\ 2\ \ =\ \ 6(79)$$

Index 5 har element 31, derfor skal vi altså bytte index 2 og 5, og vi får:

$$[2,\ 4,\ 31,\ 7,\ 44,\ 32,\ 79,\ 12,\ 17,\ 59,\ 54]$$

Dette vil være vores endelige min-heap efter insert(2) og remove()



Opgave 2

Based on min_heap.h, implement the min heap functionality based on the algorithm described in the slides, using the binary tree node structure:

```
template <typename T>
struct Node {
    T data;
    Node* parent;
    Node* left;
    Node* right;

    // Constructor
    Node(T value) {
        data = value;
        left = right = nullptr;
    }
};
```

I denne opgave har vi implementeret MinHeap algoritmen i den givne fil *min_heap.h* ved brug af binær tree noder strukturen.

```
template<typename T>
class MinHeap
{
public:
    /**
     * @brief Inserts an element and updates the Heap
     *
     * @param x Element to insert
     */
    void insert(const T& x)
    {
        Node* newnode = new Node(x);
        if (size == 0)
        {
```

```
            root = newnode; // head points on the first node of the
heap
            size++; // increment size
            return;
        }
        queue<Node*> q; // create a queue
        q.push(root); // push the root

        while(!q.empty()) {

            Node* current = q.front(); // return reference to first
element
            q.pop(); // remove front from queue

            if (!current->left) // if no left child
            {
                current->left = newnode;
                newnode->parent = current;
                size++;
                break;
            }
            else if (!current->right) // if no right child
            {
                current->right = newnode;
                newnode->parent = current;
                size++;
                break;
            }

            q.push(current->left); // push the left child
            q.push(current->right); // push the right child

        }

        // Heapify Up: while parent and data les than parent data
        while(newnode->parent && newnode->data < newnode->parent->data)
        {
            std::swap(newnode->data, newnode->parent->data); // swap
values
            newnode = newnode->parent;
        }
    }
```

```cpp
    /**
     * @brief Remove minimum (top) element and update Heap
     *
     */
    void remove()
    {
        if(size == 0){throw std::underflow_error("Heap is empty");}
        if(size == 1)
        {
            delete root; // delete the node
            root = nullptr; // head point at null
            size = 0;  // heap is empty
            return;
        }

        queue<Node*> q; // create queue
        q.push(root); // push the root

        Node* current = nullptr; // current node is null
        while (!q.empty()) // as long as the queue isnt empty
        {
            current = q.front(); // current point on the first element
of the queue
            q.pop(); // remove element from queue
            if(current->left){q.push(current->left);} // if left child
push to the queue
            if(current->right){q.push(current->right);} // if right
child, push to the queue
        }
        // Here current holds the last insterted element in the MinHeap

        std::swap(current->data, root->data); // swap the root and
current data
        if (current->parent->left == current){current->parent->left =
nullptr;} // if current is a left child
        else if (current->parent->right ==
current){current->parent->right = nullptr;} // if current is a right
child
        delete current; // delete the minimum element/root
        size--;

        // Heapify down
        Node* temp = root; // temp points at root
```

```cpp
        while(temp->left || temp->right) // while temp has children
        {

            Node* smallest = temp->left;
            if(temp->right && temp->right->data < temp->left->data)
            {
                smallest = temp->right;
            }

            if(temp->data <= smallest->data)
                break;

            std::swap(temp->data, smallest->data);
            temp = smallest;
        }
    }

    /**
     * @brief Inspect if Heap is empty
     *
     * @return true Heap is empty
     * @return false Heap is not empty
     */
    bool isEmpty() const
    {
        return size == 0;
    }

    /**
     * @brief Access the minimum (top) element of the Heap
     *
     * @return T Minimum element in Heap
     */
    T peek()
    {
        if (size == 0)
            throw std::underflow_error("Heap is empty");

            return root->data;
    }

private:
    int size = 0;
```

```cpp
    struct Node {
        T data;
        Node* parent;
        Node* left;
        Node* right;

        // Constructor
        Node(T value) {
            data = value;
            parent = left = right = nullptr;
        }
    };

    Node* root;
};
```

```cpp
#include <iostream>
#include "min_heap.h"  // Sørg for at navnet matcher din headerfil

using namespace std;

int main()
{
    cout << "===== MinHeap Test =====" << endl;

    MinHeap<int> heap;

    // Test 1: isEmpty på en tom heap
    cout << "Heap tom? " << (heap.isEmpty() ? "Ja" : "Nej") << endl;

    // Test 2: Indsæt nogle elementer
    cout << "\nIndsætter elementer: 5, 2, 8, 1, 7, 3\n";
    heap.insert(5);
    heap.insert(2);
    heap.insert(8);
    heap.insert(1);
    heap.insert(7);
    heap.insert(3);

    cout << "Heap tom? " << (heap.isEmpty() ? "Ja" : "Nej") << endl;
    cout << "Min (peek): " << heap.peek() << endl;
```

```cpp
    // Test 3: Fjern elementer én ad gangen
    cout << "\nFjerner elementer i rækkefølge:\n";
    while (!heap.isEmpty())
    {
        cout << "Peek: " << heap.peek() << "  -->  Fjernes\n";
        heap.remove();

        if (!heap.isEmpty())
            cout << "Nyt min (peek): " << heap.peek() << endl;
        else
            cout << "Heap er nu tom.\n";
    }

    // Test 4: Prøv at fjerne fra en tom heap (for at teste exception)
    cout << "\nTester fjernelse fra tom heap...\n";
    try {
        heap.remove();
    }
    catch (const std::underflow_error& e) {
        cout << "Forventet fejl: " << e.what() << endl;
    }

    // Test 5: Prøv peek() på tom heap
    cout << "\nTester peek på tom heap...\n";
    try {
        cout << heap.peek() << endl;
    }
    catch (const std::underflow_error& e) {
        cout << "Forventet fejl: " << e.what() << endl;
    }

    cout << "\n===== Test afsluttet =====" << endl;
    return 0;
}
```

Opgave 3

Based on priority Queue ADT in priority_queue.h create an implementation, MinHeapPriorityQueue(), that uses your MinHeap implementation. Also, provide testing code.

Vi henviser til koden i opgave 3 i den vedhæftet zip-fil.

```
#pragma once


template<class T>
class PriorityQueue
{
public:
    virtual void push(const T& x) = 0;
    virtual void pop() = 0;
    virtual T top() = 0;
    virtual bool empty() const = 0;
    virtual ~PriorityQueue() {}
};
```

```
#pragma once
#include "priority_queue.h"
#include "min_heap.h"

template<typename T>
class MinHeapPriorityQueue : public PriorityQueue<T>
{

    private:
        MinHeap<T> heap;

    public:
        void push(const T& x) override {heap.insert(x);}
        void pop() override {heap.remove();}
        T top() override {return heap.peek();}
        bool empty() const override {return heap.isEmpty();}

};

#include <iostream>
#include "priority_heap.h"
```

```cpp
using namespace std;

int main()
{
    MinHeapPriorityQueue<int> pq;

    cout << "Indsætter elementer: 5, 2, 8, 1\n";
    pq.push(5);
    pq.push(2);
    pq.push(8);
    pq.push(1);

    cout << "Top (min): " << pq.top() << endl; // 1
    pq.pop();
    cout << "Efter pop, ny top: " << pq.top() << endl; // 2

    cout << "\nTømmer køen:\n";
    while (!pq.empty()) {
        cout << pq.top() << " ";
        pq.pop();
    }
    cout << endl;

    return 0;
}
```

Opgave 4

Using pen an paper, illustrate how an AVL tree is built with the following elements [10, 20, 15, 25, 30, 16, 18, 19]. [I.e. 10 is inserted first then 20…). You must illustrate all rotations etc. Then illustrate what happens when 30 is deleted then 18.

Løsning:

OBS:
Vi har vedhæftet løsning på papir i pdf'en " ny opgave 4 hand in-4.pdf"

Opgave 5

5 (week 8) A node in a binary tree is an only-child if it has a parent node but no sibling node (Note: The root does **not** qualify as an only child). *The loneliness-ratio* (LR) of a given binary tree T is defined as the following ratio:

$$LR(T) = \frac{The\ number\ of\ nodes\ in\ T\ that\ are\ only\ childen}{The\ number\ of\ nodes\ in\ T}$$

a) Argue that for any nonempty AVL tree T, we have $LR(T) \leq 1/2$
b) Is it true for any binary tree T, that if $LR(T) \leq 1/2$ then height(T) = log(n)

Delopgave a)

Et ene barn er altså en node der har:

- En parent
- Ingen sibling
- root tæller ikke

Vi skal basere princippet på et AVL træ, det vil altså sige at vi højest må have en balance faktor på -1 og +1.

Det vil altså sige at til hver node må forskellen på højre og venstre deltræ højest være 1. Dette vil altså sige at hvis en node kun har et barn **skal** dette altså være et blad.

Derudover for at vi skal overholde betingelserne for et AVL skal dette only child have en parent som har en sibling ellers vil det ikke opfyldes.

Vi kan altså derfor skrive:

$$2 \times Antal\ only\ childs \leq antal\ noder$$

Vi kan så omskrive dette til:

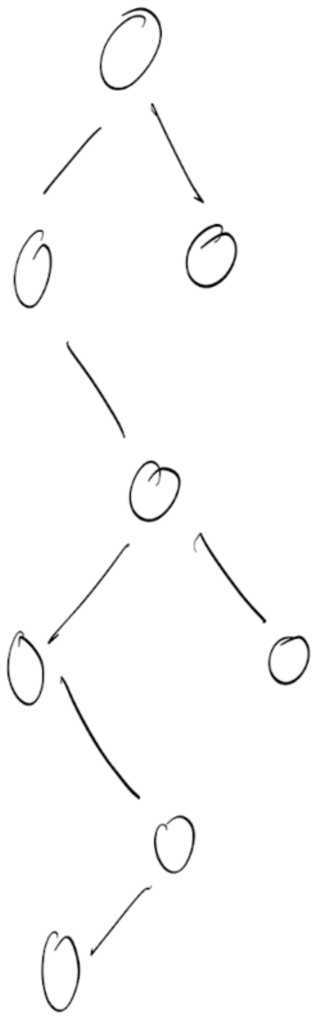
$$2 \times Antal\ only\ childs \leq antal\ noder \rightarrow \frac{Antal\ only\ childs}{Antal\ noder} \leq \frac{1}{2}$$

Vi har altså derfor argumenteret for LR(T) altid vil være lig eller mindre $\frac{1}{2}$

Delopgave b)

Vi skal beviser hvorvidt at $LR(T) \leq \frac{1}{2}$ garenterer at højden af træet følger $log_2(n)$.

Da antagelsen er at det er gældende for et hvert eksempel af et bineært træ kan vi altså lave følgende:



Vi har her som eksempel et binært træ med en højde på 5, her ser vi hver anden node på stammen har to børn, det vil sige at ca halvdelen af børnene vil være såkaldte only children. LR(t) < ½ mens vi har at træets højde vokser lineært med antallet af noder og ikke som log (n). Det er derfor bevist at udsagnet er falsk

Opgave 6

Vi henviser til koden i opgave 6 vedhæftet i zip-filen.

```
        iterator& operator ++() {
            // If the current node is null/end(). return the iterator
            if (node == nullptr){return *this;}

            // if the current node has a right child -> go to the right
child
            if (node->right != nullptr){
                node = node->right;
                // As long as current node has a left child -> go left
                while (node->left != nullptr)
                {
                    node = node->left;
                }
            }
            else{

                BinaryNode* p = node->parent;
                while (p != nullptr && node == p->right) {
                    p = p->parent;
                    node = node->parent;
                }
                node = p;
            }

            return *this;
        }
```

```
#include <vector>
#include "binary_search_tree.h"

using namespace std;

template<typename Comparable>
class OrderedSet {
  private:
```

```
  size_t theSize;
  BinarySearchTree<Comparable> tree;

public:
  OrderedSet() { }

  ~OrderedSet() { clear(); }

  OrderedSet(const OrderedSet& s) {

      // iterate through s and create copy of old set
      for (auto it = s.begin(); it != s.end(); ++it) {
      insert(*it); // insert element
      }
  }

  void clear() {
      // clear and reset size
      tree.makeEmpty();
      theSize = 0;
  }

  size_t size() const {

      // return size
      return theSize;
  }

  bool empty() const {

      // check if empty
      return tree.isEmpty();
  }

  void push(const Comparable& t) {
      insert(t); // insert element
  }


  friend class BinarySearchTree<Comparable>;
  typedef typename BinarySearchTree<Comparable>::iterator iterator;

  iterator begin() const {
```

```cpp
        // return iterator to the smallest element
        return tree.findMin();
    }

    iterator end() const {

        // return iterator at one position after last element
        return iterator(nullptr);
    }

    iterator insert(const Comparable& t) {

        // if the element is not in the set
        if (!tree.contains(t)) {
            // insert and get iterator
            auto it = tree.insert(t);
            ++theSize; // increase size
            return it; // return iterator
        }
        // if element exists -> return iterator to the existing element
        return tree.find(t);    //
    }

    iterator find(const Comparable& t) {

        // If the element exsists
        if (tree.contains(t))
        {
            // return iterator to element
            return tree.find(t);
        }
        // else return nullptr
        return end();
    }

    iterator erase(iterator& itr) {

        // if itr --> nullptr
        if (itr == end())
        {
            return itr; // return iterator
        }
```

```
        // safe position
        iterator next = itr;
        // position after the element
        next++;
        // remove the element
        tree.remove(*itr);
        // decrease size
        --theSize;
        // return position of the next element
        return next;
    }
};
```

```
#include <iostream>
#include "ordered_set.h"

using namespace std;

int main() {
    OrderedSet<int> set;
    set.push(4);
    set.push(0);
    set.push(2);
    set.push(7);
    set.push(0);
    set.push(2);
    set.push(6);
    set.push(4);
    set.push(12);
    set.push(11);
    set.push(8);
    set.push(1);
    set.push(5);
    set.push(0);
    set.push(3);
    set.push(2);
    set.push(7);
    set.push(1);
    set.push(1);
    set.push(1);
    set.push(9);
```

```
    set.push(7);
    set.push(11);
    set.push(1);

    // Test copy constructor
    OrderedSet<int> set2 = set;

    // Insert new smallest element
    OrderedSet<int>::iterator it = set2.insert(-1);
    cout << *it << endl;

    // Use ++ operator to iterate through the ordered set
    if (++it != set2.end())
        cout << *it << endl;

    // Insert duplicate and print succ to prove that no new node is
inserted
    it = set2.insert(4);
    cout << *it << endl;
    if (++it != set2.end())
        cout << *it << endl;

    it = set2.insert(20);
    cout << *it << endl;
    if (++it != set2.end())
        cout << *it << endl;

    it = set2.find(15);
    if (++it != set2.end())
        cout << *it << endl;
    it = set2.find(12);
    cout << *it << endl;
    if (++it != set2.end())
        cout << *it << endl;
    it = set2.erase(it);
    if (it != set2.end())
        cout << *it << endl;

    it = set2.find(4);
    it = set2.erase(it);
    if (it != set2.end())
        cout << *it << endl;
```

```
    for (OrderedSet<int>::iterator it = set2.begin(); it != set2.end();
it++) {
        cout << *it << ' ';
    }
    cout << endl;
}
```

Opgave 7

(7) (Week 8) Design and implement a linear-time algorithm that verifies that the height information in an AVL tree is correct i.e. the balance property is in order for all nodes.

Koden kan også findes i den vedfhæftet zip-fil.

```
#pragma once
#include "avl_tree.h"

template<typename Comparable>
bool verify_avl(AvlTree<Comparable>::AvlNode* node, int &height){


    if (node == nullptr)
    {
        height = -1;
        return true;
    }


    int left_height;
    int right_height;

    bool leftValid = verify_avl(node->left, left_height);
    bool rightValid = verify_avl(node->right, right_height);

    bool balanced = abs(left_height - right_height) <= 1;
    height = max(left_height, right_height) + 1;

    if (node->height != height)
```

```
    {
        return false;
    }


    return leftValid && rightValid && balanced;

}
```