Informatik II

Übung 6

Andreas Bärtschi, Andreea Ciuprina, Felix Friedrich, Patrick Gruntz,
Hermann Lehner, Max Rossmannek, Chris Wendler

FS 2018

Program Today

1. Feedback of last exercise
2. Repetition Lectures
3. In-Class-Exercises

Last week: Imperative -> Functional

```
try (FileReader fr = new FileReader("data.csv");
    BufferedReader br = new BufferedReader(fr);) {
    String line = br.readLine();
    while ((line = br.readLine()) != null) {
        Measurement m = new Measurement(line);
        if (zuerich.near(m.getPosition())
        && beginning.compareTo(m.getDateTime()) <= 0
        &&  end.compareTo(m.getDateTime()) >= 0) {
            System.out.print(m);
        }
    }
}
```

Last week: Imperative -> Functional

```
try (Stream<String> stream = Files.lines(Paths.get("data.csv"))) {
    stream
        .skip(1)
        .map(Measurement::new)
        .filter(m -> zuerich.near(m.getPosition()))
        .filter(m -> beginning.compareTo(m.getDateTime()) <= 0)
        .filter(m -> end.compareTo(m.getDateTime()) >= 0)
        .filter(m -> m.getMagnitude() >= lower) // new filter for
        .filter(m -> m.getMagnitude() <= upper) // magnitude interval
        .forEach(System.out::print);
}
```

Repetition: Binary Trees, Inserting a Key

Binary Search Trees

- Search for Key.
- Insert at the reached empty leaf (`null`).

MinHeap

- Insert at the very back of the Array.
- Restore Heap-Condition: `siftUp` (climb successively).

Repetition: Binary Trees, Inserting a Key

Binary Search Trees

- Search for Key.
- Insert at the reached empty leaf (`null`).

MinHeap

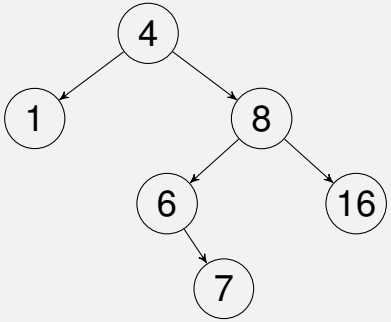
- Insert at the very back of the Array.
- Restore Heap-Condition: `siftUp` (climb successively).

Exercise: Insert $4, 8, 16, 1, 6, 7$ into empty Tree/Heap.

Repetition: Binary Trees, Inserting a Key

Binary Search Trees

- Search for Key.
- Insert at the reached empty leaf (`null`).



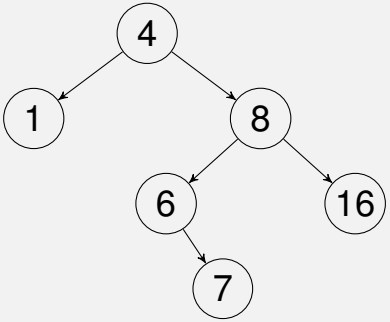
MinHeap

- Insert at the very back of the Array.
- Restore Heap-Condition: `siftUp` (climb successively).

Repetition: Binary Trees, Inserting a Key

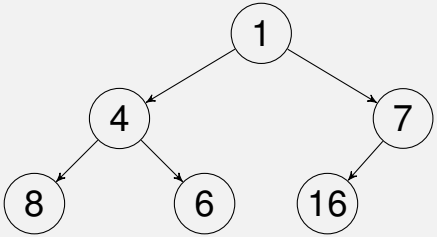
Binary Search Trees

- Search for Key.
- Insert at the reached empty leaf (`null`).



MinHeap

- Insert at the very back of the Array.
- Restore Heap-Condition: `siftUp` (climb successively).



Java: Insertion into Search Tree

```
public SearchNode Insert (int k) {
  if (root == null) { return root = new SearchNode(k); }
  SearchNode t=root;
  while (true) {
    if (k == t.key) { return null; }    // key already exists, abort
    if (k < t.key) {                    // IF new key < t.key
                                        // CHECK LEFT CHILD:
      if (t.left == null)               // – Empty leaf => Insert
           return t.left = new SearchNode(k);
      else  t = t.left;                 // – Not a leaf => Continue
    }
    else {                              // ELSE IF new key > t.key
        ...                             // CHECK RIGHT CHILD
  }
}
```

Java: Insertion into MinHeap

```
class MinHeap
{
    private int size;                   // current number of elements
    private int[] elements;             // currently stored elements
    ...
    // Insert Element
    public void push(int k) {
        size++;                         // increase number of elements
        elements[size−1] = k;           // add key to end of array
        siftUp(size−1);                 // restore heap condition
    }
    // siftUp(index)
    private void siftUp(int x) {
        ...
}
```

Repetition: Binary Trees, Deleting a Key

Binary Search Trees

- Replace key k by symmetric successor n.
- Careful: What about right child of n?

MinHeap

- Replace key by last element of the array.
- Restore Heap-Condition: `siftDown` *or* `siftUp`.

Repetition: Binary Trees, Deleting a Key

Binary Search Trees

- Replace key k by symmetric successor n.
- Careful: What about right child of n?

MinHeap

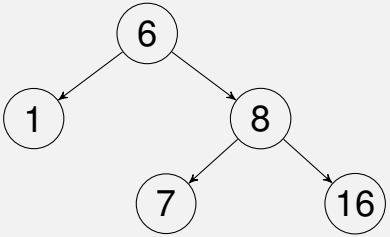
- Replace key by last element of the array.
- Restore Heap-Condition: `siftDown` *or* `siftUp`.

Exercise: Delete 4 from Example Tree/Heap.

Repetition: Binary Trees, Deleting a Key

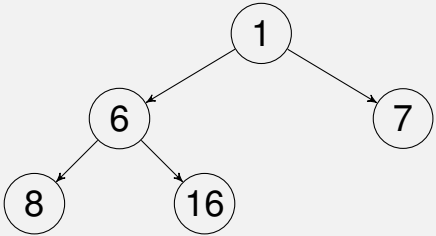
Binary Search Trees

- Replace key k by symmetric successor n.
- Careful: What about right child of n?



MinHeap

- Replace key by last element of the array.
- Restore Heap-Condition: `siftDown` *or* `siftUp`.



Java: Delete from Search Tree I

```
public void Delete (int k) {
  SearchNode n = root;
  if (n != null && n.key == k) {
    root = SymmetricDesc(root);
  } else {
    while (n != null) {
      if (n.left != null && k == n.left.key) {
        n.left = SymmetricDesc(n.left); return; // Replace by symm.
      } else if (n.right != null && k == n.right.key) {
        n.right = SymmetricDesc(n.right); return; // successor
      } else if (k < n.key) { n = n.left;
      } else { n = n.right; }
    }
  }
}
```

Java: Delete from Search Tree II

```
public SearchNode SymmetricDesc(SearchNode node) { // to be replaced
  if (node.left == null) { return node.right; } // ! only 1 child
  if (node.right == null) { return node.left; } // ! no children
  SearchNode n = node;  // n will become symmetric successor of node
  SearchNode parent = null;
  n = n.right;
  while (n.left != null) { parent = n; n = n.left; }
  if (parent != null) {  // Take care of all side effects:
    parent.left = n.right; // -connect child of n to its grandparent
    n.left = node.left;     // -wire children of node to
    n.right = node.right;   //  the symmetric successor n
  } else { // n is both symmetric successor AND child of node
    n.left = node.left; }
  return n;
}
```

Java: Delete from MinHeap

Problem! How to find a key in a MinHeap?

$\Rightarrow$ We only take care of root deletions (Extract-Min).

```
// Return Top element
public int top() { return elements[0]; }
// Extract/Remove Top Element
public void pop() {
    elements[0] = elements[size-1]; // replace by last leaf
    size--;                         // 'delete' last leaf
    siftDown(0);                    // restore heap condition
}
// siftDown (index)
private void siftDown(int x) {
...
```

Recursion in search trees

Exercise:
Write a (recursive) Function `int height(SearchNode n)` to determine the height of a (sub-)tree.

[Start here: `https://codeboard.io/projects/48851`]

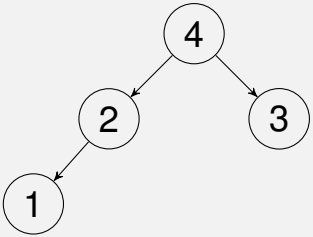
Recursion in search trees

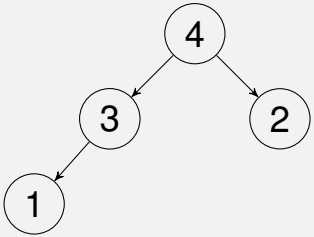
```
public int height(SearchNode n) {
    if (n == null) { return 0; }     // empty tree
    int leftheight = height(n.left);
    int rightheight = height(n.right);
    if (leftheight >= rightheight) { // left subtree is higher
        return leftheight+1;
    } else {
        return rightheight+1;        // right subtree is higher
    }
}
/* // call with
else if(command.equals("height")) {
    int value = scanner.nextInt();
    System.out.println(s.height(s.Search(value)));
} */
```

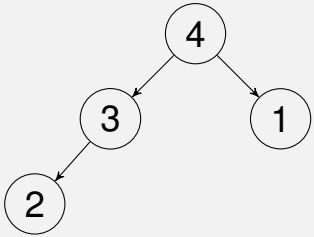
Number of MaxHeaps on n distinct keys

Let $N(n)$ denote the number of distinct Max-Heaps which can be built from all the keys $1, 2, \ldots, n$. For example we have $N(1) = 1,\ N(2) = 1,\ N(3) = 2,\ N(4) = 3$ und $N(5) = 8$.

Find the values $N(6)$ and $N(7)$.

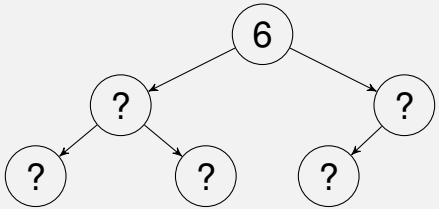






Number of MaxHeaps on n distinct keys

A MaxHeap containing the elements $1, 2, 3, 4, 5, 6$ has the structure:



Number of combinations to choose elements for the left subtree: $\binom{5}{3}$.

$$\Rightarrow N(6) = \binom{5}{3} \cdot N(3) \cdot N(2) = 10 \cdot 2 \cdot 1 = 20.$$

and $N(7) = \binom{6}{3} \cdot N(3) \cdot N(3) = 20 \cdot 2 \cdot 2 = 80.$

Questions / Suggestions?