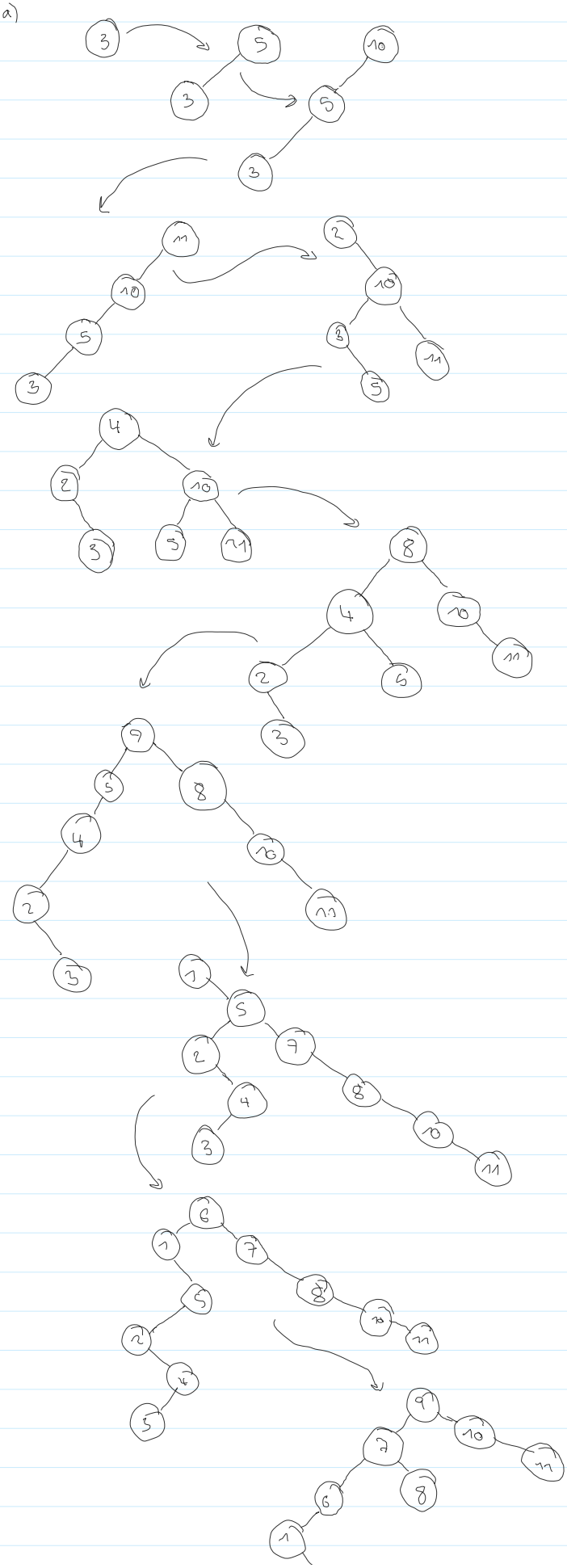
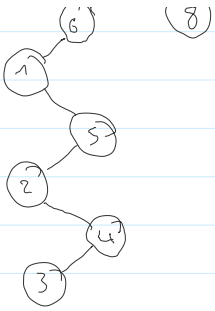
AuD Hausübung 7

Donnerstag, 13. Juni 2019

Jonas Franz, Nicolas Petermann, Julian Imhof

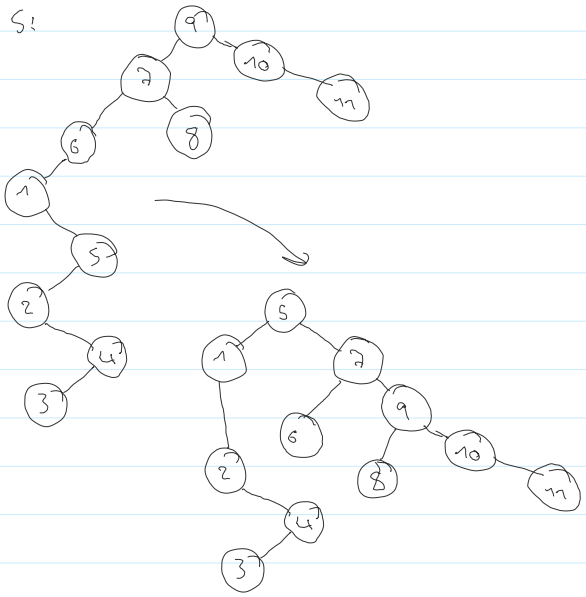
H1) a)



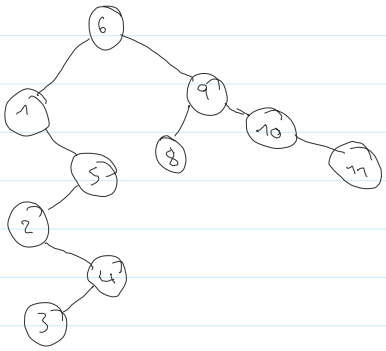


b)

find 5:

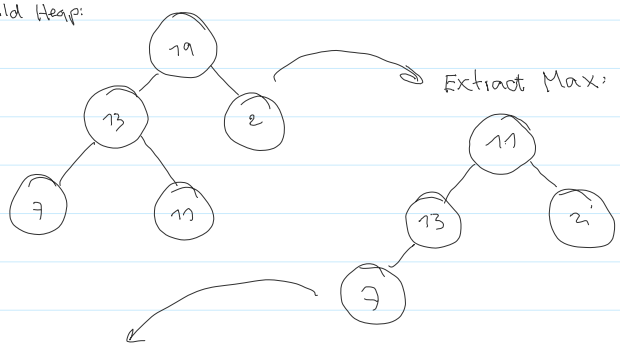


delete 9:



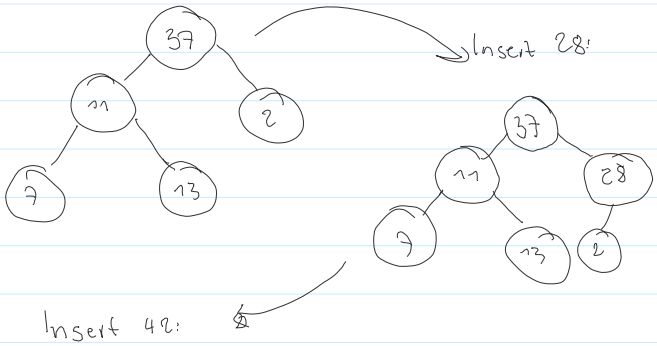
A2) a)

Build Heap:



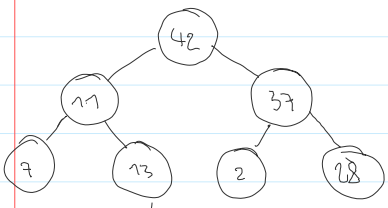
Insert 37:

Insert 37:



Insert 28:

Insert 42:



Heapsort:

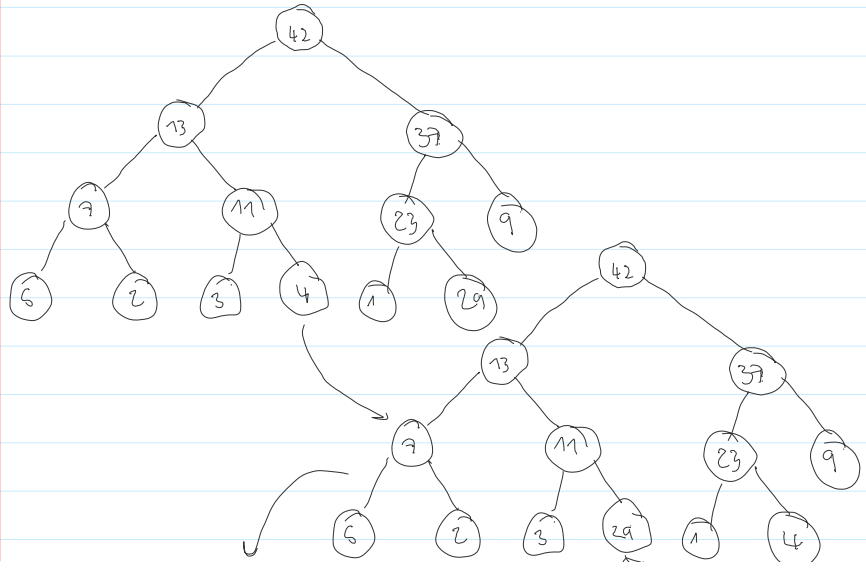
[42, 37, 28, 13, 11, 7, 2]

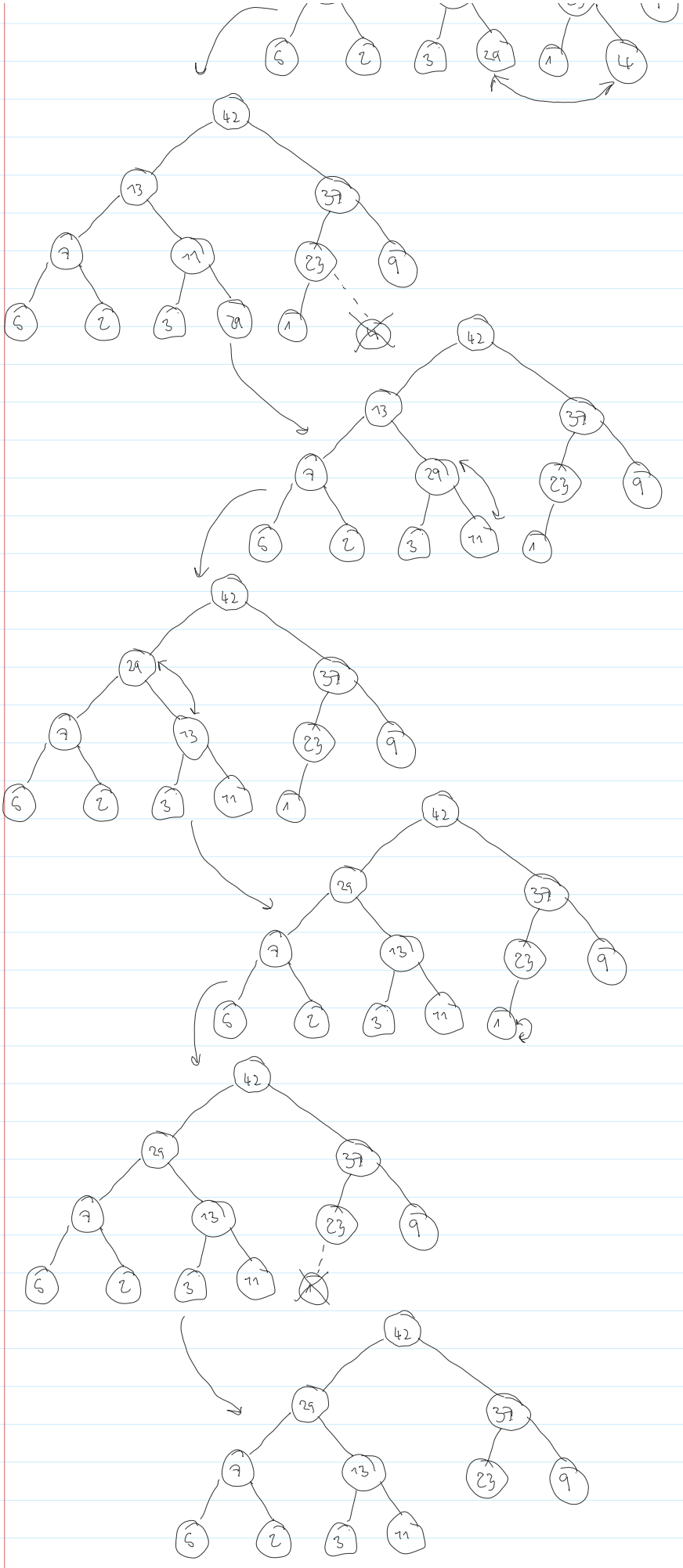
b)

- Extract Max
- Insert 32
- Extract Max
- Insert 8

c) delete (i)

```
swap(H[i], H[H.size - 1])
H[H.size - 1] = nil
if (H[i] ≠ H[H.size - 1])
  while (H[i].parent != nil and H[i].parent < H[i])
    swap H[i], H[i].parent)
```





H3)

a)

```
Public int findPalindromes(String in) {
   // Split into all possible subwords
   List<String> subWords = new ArrayList();
   for (int mask = in.length(); mask > 0; mask--) {
      for (int shift = 0; shift < in.length() - mask + 1; shift++) {
         subWords.add(in.substring(shift, shift + mask));
      }
   }

   // Check for subwords that are palindromes
   List<String> palindromes = new ArrayList();
   for (int i = 0; i < subWords.size(); i++) {
      String s = subWords.get(i);
      if (s.equals("") || palindromes.contains(s))continue;
      boolean isPalindrom = true;
      while (s.length() > 1) {
         if (!s.substring(0, 1).equals(s.substring(s.length() - 1)))isPalindrom = false;
         s = s.substring(1, s.length() - 1);
      }
      if (isPalindrom)palindromes.add(subWords.get(i));
   }

   return findRec(in, palindromes); // Get Result recursiveley
}
```

O(n²) { O(n²) { O(n²) } max O(n³) }

Filter for empty words and duplicates

axxx

a ≠

```
public int findRec(String in, List<String> palindromes) {
   for (int mask = in.length(); mask > 0; mask--) {
      for (int shift = 0; shift < in.length() - mask + 1; shift++) {
         if (palindromes.contains(in.substring(shift, shift + mask))) {
            return 1 +
                  findRec(in.substring(0, shift), palindromes) +                  // Current
                  findRec(in.substring(shift + mask, in.length()), palindromes);  // Left Substring
                                                                                  // Right Substring
         }
      }
   }
   return 0;
}
```

n² { }

$$T(n) = \begin{cases} c & n = 1 \\ 2T(n-1) \cdot cn^2 \end{cases}$$

↳ Max n^2

Der Algorithmus besteht aus zwei Funktionen: Der Vorbereitung und der Rekursiven Ergebnisfindung.

- Vorbereitung: Erstelle zunächst Liste mit allen möglichen Teilwörtern. Filtere dann in dieser Tabelle nach Palindromen.

  ↳ Lookup-Table

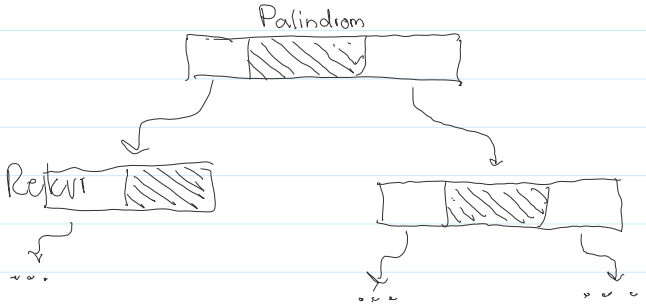
- Rekursion:

  Beginne mit ganzem Wort: Wort ist Palindrom? → Fertig. Ansonsten: Probiere so lange mit allen Teilwörtern der größe nach sortiert bis Treffer in Look-up Tabelle.

  Teile Wort auf, gehe Rekursiv in Linken und rechten Teil.

  Palindrom

und rechten Teil."



sion terminiert wenn Wort leer.

b) Wie schon oben eingezeichnet, besteht das Programm aus drei Teilen mit jeweils $O(n^2)$. Somit befindet sich die Laufzeit im Bereich $O(3n^2) = O(n^2)$.

Korrektheit:

Schleifeninvariante:
Nach n Rekursiven Schritten wurden die n größten Palindrome aus dem Eingabewort erkannt.

Vor der Rekursion:
Es wurden keine Palindrome bisher erkannt, also bei n=0 wurden 0 Palindrome erkannt.

In der Schleife:
Durch die Schleife wird zunächst das gesamte Wort getestet. Bei t äußeren Schleifendurchläufen können nur noch t-lange Palindrome in dem Teilwort sein, ansonsten hätte die Schleife bereits rekursiv die Funktion aufgerufen und Terminiert. Somit wird immer das längste Palindrom erkannt.

Nach der Schleife:
Der Rekursionsanker ist erreicht, sobald das Teilwort leer ist. In jedem Rekursiven Aufruf wird ein Palindrom erkannt oder das Wort ist leer. Hat ein Wort also $p(w) = 3$ Palindrome, ruft sich die Funktion 6-mal rekursiv auf, gibt aber nur bei 3 davon

Palindrome, ruft sich die Funktion 6-mal
rekursiv auf, gibt aber nur bei 3 aufrufen
ein valides Ergebnis zurück.