Aufgabe 1

Ein Binomialbaum vom Rang $r > 0$ entsteht durch das Anfügen eines Binomialbaumes vom Rang $r-1$ an die Wurzel eines Binomialbaumes vom Rang $r-1$. Es gilt also

$$N(r,k) = \begin{cases} 1 & k = 0 \\ 0 & k > 0, r = 0 \\ N(r-1,k) + N(r-1,k-1) & k > 0, r > 0 \end{cases}$$

Wir beweisen die Aussage durch Induktion über r (mit k quantifiziert).

$$\forall k : N(r,k) = \binom{r}{k}$$

Sei also $r = 0$.

$$N(0,k) = \left\{ \begin{matrix} 1 & \text{für } k = 0 \\ 0 & \text{für } k > 0 \end{matrix} \right\} = \binom{0}{k}$$

Nun $r > 0$. Angenommen $k = 0$, dann

$$N(r,0) = 1 = \binom{r}{0}$$

Ansonsten ist $k > 0$ und

$$N(r,k) = N(r-1,k) + N(r-1,k-1) = \binom{r-1}{k} + \binom{r-1}{k-1} = \binom{r}{k}$$

wobei die letzte Gleichung aus dem Pascalschen Dreieck folgt.

Verwandte Eigenschaften

1. Da
$$\sum_{i=0}^{r} \binom{r}{i} = 2^r$$
(die Summe einer Ebene des Pascalschen Dreiecks ist das Doppelte der Vorhergehenden, da jeder Eintrag in zwei Einträge der nächsten Ebene eingeht), besitzt ein Binomialbaums mit Rang r 2^r viele Knoten.

2. Da die Höhe und der Rang eines Binomialbaums identisch sind (Ein Link erhöht Höhe und Rang gleichzeitig um 1), hat ein Binomialbaum der Höhe h, 2^h viele Knoten.

3. Ein Binomialbaum mit n vielen Knoten hat eine Höhe von $\log n$ (folgt direkt aus 2.).

4. Die Darstellung eines Binomial Heaps ist eindeutig durch die Anzahl der Knoten bestimmt.

   Hierfür nehmen wir uns das Array, das für jeden Baum einer Größe eine Position hat, die entweder leer ist falls kein Baum der Größe existiert oder genau einen Baum der Größe beinhaltet. Dieses Array können wir uns nun als eine Binärzahl vorstellen, wobei leere Positionen der 0 und gefüllte der 1 entsprechen. Es reicht nun das Zusammenfügen zweier Binomial Heaps zu betrachten, da sich jede Änderung an einem Binomial Heap dadurch darstellen lässt. Beim Zusammenfügen wird bei dem kleinsten Baum angefangen und der Reihe nach auf beiden Arrays die i-te Position verglichen und in ein neues Array an der i-ten Stelle wie folgt eingefügt:

   - Beide haben an Position i einen Baum:

     Wir setzen die Bäume zu einem Baum der um 1 größer ist zusammen, merken uns diesen als Übertrag für das nächste Feld und fügen an die i-te Position den Übertrag vom letzten Feld ein, falls es einen gibt.
   - Beide haben an Position i keinen Baum:

     Wir fügen, falls vorhanden, den Übertrag vom letzten Feld ein.
   - Genau einer hat an Position i einen Baum:

     Falls wir einen Baum als Übertrag haben behandeln wir diesen Fall wie den ersten, ansonsten fügen wir den Baum einfach an die i-te Stelle ein.

   Dies entspricht genau der Addition der entsprechenden Binärzahlen, weshalb es eine Bijektion zwischen dem Array der Wurzeln und Binärzahlen gibt. Somit ist die Darstellung eindeutig, da die Binärdarstellung eindeutig sind.

Aufgabe 2

Mit Knoten sind hier *full* sowie *hollow* Knoten gemeint. Wir beweisen die Aussage durch Induktion über r.

- Für $r = 0$ haben wir einen Knoten (die Wurzel) und es gilt $F_3 - 1 = 2 - 1 = 1$.
- Für $r = 1$ haben wir zwei Knoten und es gilt $F_4 - 1 = 3 - 1 = 2$.
- Für $r \geq 2$ haben wir mindestens Kinder vom Rang $r - 2$ und $r - 1$, also gibt es mindestens $1 + F_{r+1} - 1 + F_{r+2} - 1 = F_{r+3} - 1$ Knoten.

Aufgabe 3

Sollte IncreaseKey in amortisierter konstanter Laufzeit umsetzbar sein, so könnten wir vergleichsbasiert in $\mathcal{O}(n)$ in sortieren:

```
Sort(A){
 max = FindMax(A);
 hheap = MakeHollowHeap(A);
 sorted = new List;
 for i = 1 to A.size() {
  sorted.add(hheap.min);
  hheap.increaseKey(hheap.min,max+1);
 }
 return sorted;
}
```

Intuitiv ist das Problem, dass beim Erhöhen eines Knoten er nun im Baum nach unten geschoben werden muss. Die Frage ist also welcher seiner Kinder ihn ersetzt. Da wir aber keine Ahnung haben, welcher seiner Kinder jetzt das kleinste ist, müssen wir alle Kinder traversieren (eventuell müssen wir den Knoten noch tiefer schieben).

Aufgabe 4

```
struct BinomialNode {
    int key;
    int rank;
    List<BinomialNode> children;
    BinominalNode* parent;
};

struct BinomialHeap {
    int n;
    List<BinomialNode> trees;
    BinomialNode* minimum;
};
```

```
struct HollowNode {
    int key;
    int rank;
    List<HollowNode> children;
    bool hollow;
};

struct HollowHeap {
    List<HollowNode> trees;
    HollowNode* minimum;
};
```

```
void consolidate(BinomialHeap h) {
  int maxrank = log2(h.n);
  BinomialNode ** ofrank = new BinomialNode*[maxrank + 1];
  for (BinomialNode tree : h.trees){
    while (ofrank[tree.rank] != NULL){
      tree = link(tree, *ofrank[tree.rank]);
      ofrank[tree.rank] = NULL;
    }
    *ofrank[tree.rank] = tree;
  }
  h.trees.clear();
  for (int r = 0; r <= maxrank; r++){
    BinomialNode root = *ofrank[r];
    if (root.key < h.minimum->key)
      h.minimum = &root;
    h.trees.push_back(root);
  }
}
```

Da jeder Schritt in der While-Schleife inklusive der Link Operation konstante Zeit benötigt und gleichzeitig eine Wurzel verschwindet, kann die Laufzeit durch das Guthaben dieser Wurzel ausgeglichen werden. Sei k die Anzahl der Elemente in der Wurzelliste. Jedes Element aus der Liste wird einmal angefasst. Daher ergibt sich eine zusätzliche Laufzeit in $O(k)$. Da am Ende aber nur $\log n$ viele Wurzlen übrig bleiben wird ein Guthaben von $k - \log n$ frei, sodass nach einem Ausgleich Restkosten von $\log n$ verbleiben.

Die zweite Hälfte arbeitet nur auf den verbleibenden Wurzeln und hat daher Laufzeit $\mathcal{O}(logn)$.

Aufgabe 5

Wir kreieren ein Szenario in dem unsere Laufzeiten nicht erreicht werden können:

Hierfür betrachten wir einen Hollow-Heap der aus einem beliebig großen einzelnen Baum ohne leere Knoten besteht. In diesem rufen wir DecreaseKey auf jedes der Kinder der Wurzel auf (logarithmisch viele). Dann fügen wir einen Knoten der größer als das aktuelle Minimum ist neu hinzu und zuletzt rufen wir DeleteMinimum auf.

Nun haben wir zum einem $\log(n)$ viele Bäume durch die DecreseKey Operationen, 1 Baum durch das Einfügen und $\sum_{i=0}^{\log(n)} i$ viele Bäume die bei DeleteMinimum entstanden

sind. Dementsprechend haben wir

$$\log(n) + 1 + \sum_{i=0}^{\log(n)} i = \log(n) + 1 + \frac{\log^2(n) + \log(n)}{2}$$

viele Bäume, also $\Omega(\log^2(n))$. Diese müssen wir nun alle zusammenfügen, dementsprechend eine Laufzeit von mindestens $\Omega(\log^2(n))$ (sollten wir keine entstandenen Bäume zusammenfügen müssen).

Auf der anderen Seite verlangen wir, dass die Operationen DecreaseKey und Insert [amortisierte] Laufzeit $\mathcal{O}(1)$ und DeleteMinimium amortisierte Laufzeit $\mathcal{O}(\log n)$ besitzen. Dementsprechend müsste diese Folge von Operationen eine amortisierte Gesamtlaufzeit von $\mathcal{O}(\log n)$ besitzen.

Da wir durch das Hinzufügen eines Knotens wieder auf die anfängliche Anzahl kommen und keine leeren Knoten mehr vorhanden sind, ist der Baum strukturell identisch zum Anfang. Dadurch können wir diese Folge von Operationen beliebig oft wiederholen und somit jedes erdenkliche Guthaben aufbrauchen. Folglich ist es mit der Änderung von DecreaseKey unmöglich die amortisierten Laufzeiten aufrechtzuerhalten.