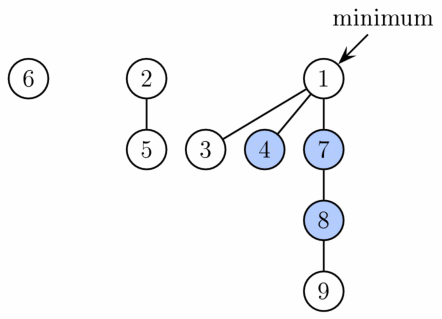
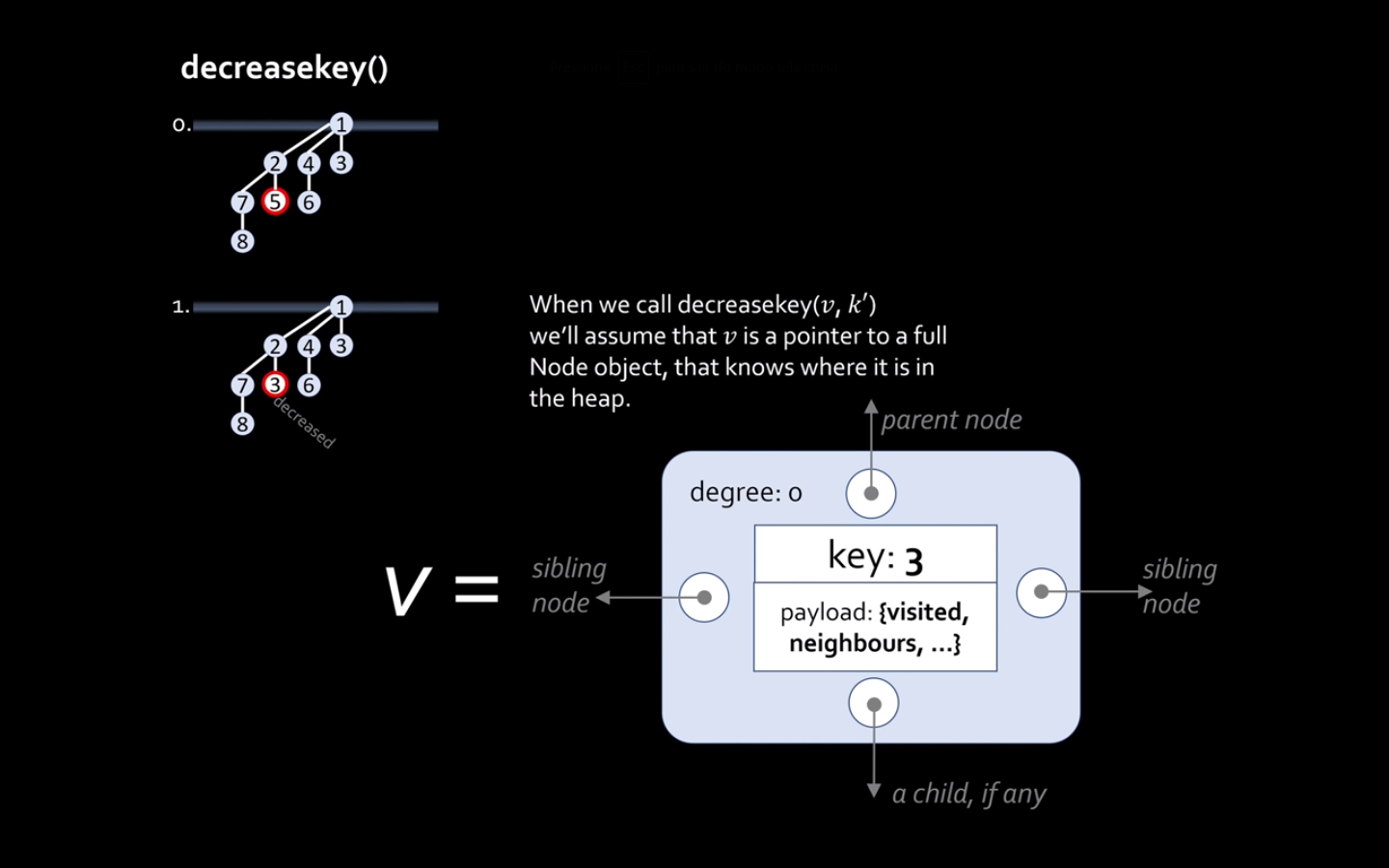
**Fibonacci Heap (1984, J W J Williams)**

Complexidade:

|  |  |
| --- | --- |
| **Algoritmo** | **Custo** |
| Find-min | O(1) |
| Delete-min | O(log n) |
| Insert | O(1) |
| Decrease-key | O(1) |
| Merge | O(1) |



É similar a uma binomial heap, porém sua estrutura é mais relaxada.



**Estrutura**

É um conjunto de árvores que satisfazem a propriedade da heap mínima. Comparadas às árvores Binomiais Heaps, a estrutura é mais flexível. As árvores não têm uma forma determinada e, no caso extremo, a heap pode ter cada elemento em uma árvore diferente. Isso permite a execução preguiçosa de parte das suas operações, adiando o trabalho para operações futuras. O nome vem da quantidade mínima de nós que uma árvore de determinado grau pode ter, que segue a sequência Fibonacci.

**Implementação das operações**

Para permitir exclusão e concatenação rápidas, as raízes são ligadas usando uma circular duplamente encadeada. Os descendentes de cada nó também são conectados assim. Para cada nó, mantemos seu número de descendentes e se o nó já perdeu algum descendente. Também mantemos um ponteiro para a raiz contendo a chave mínima.

A operação **find minimum** é trivial, já que o endereço da chave mínima é mantido. O(1).

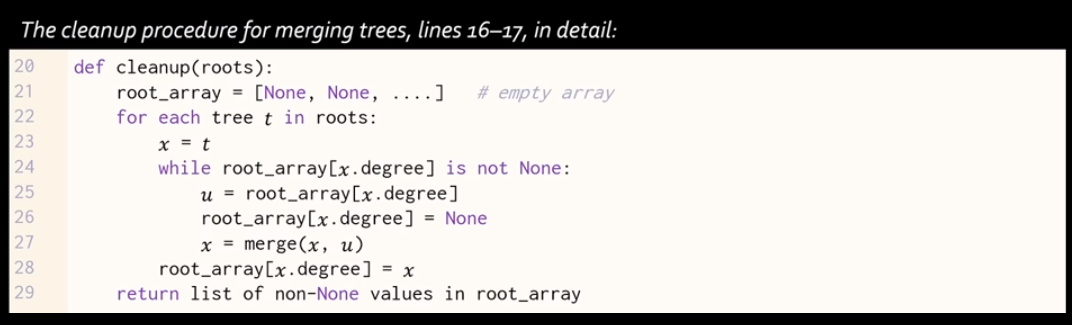
A operação **merge** é implementada pela simples concatenação de duas heaps. O(1).

A operação **insert** funciona criando uma nova heap com um elemento e fazendo o merge com a heap base. O(1)

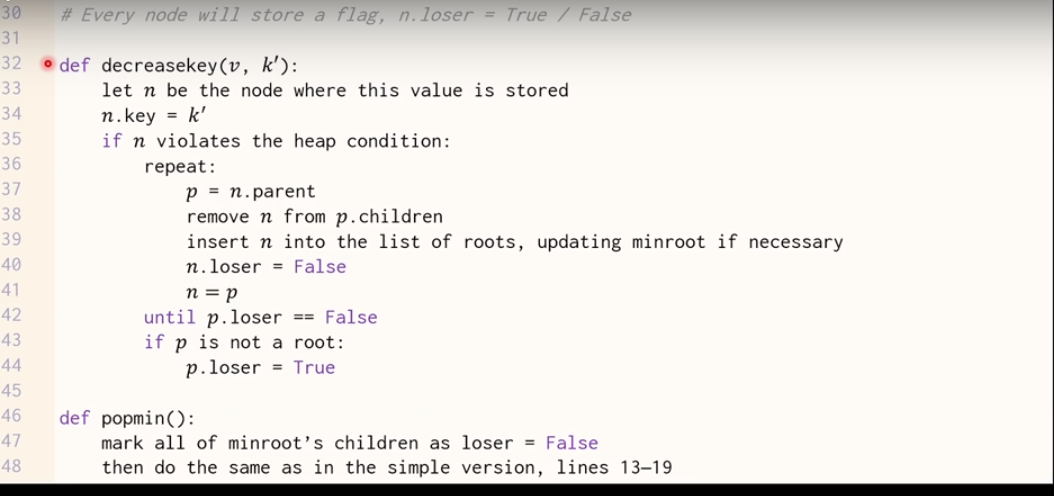
A operação **extract** **minimum** funciona em três etapas:

* Primeiro removemos o menor elemento. E tornamos as suas descendentes raízes de novas árvores.
* Na segunda etapa as árvores de mesmo grau são mescladas de forma a manter a propriedade de heap. Isso é feito fazendo da árvore com a raiz de maior chave a descendente da outra. E aumentando o grau da outra árvore em um. Isso se repete até que todas as raízes tenham grau diferente. Para encontrar as raízes de mesmo grau de forma eficiente, durante a operação de merge, nós adicionamos as raízes de cada árvore a um array à medida que percorremos a heap. Cada elemento do array armazenará o ponteiro para uma raiz com grau igual ao seu índice. Quando tentarmos associar o ponteiro de uma raiz a um índice que já contém um ponteiro, nós mesclamos as duas raízes e movemos a raiz resultante para a posição seguinte. Seguindo esse processo até encontrar uma posição vazia.
* Por último atualizamos o ponteiro para o novo nó de chave mínima.

O tempo de execução amortizado dessa operação é O(log n).



A operação **decrease key** reduzirá o valor da chave de um nó e, se a propriedade de heap for violada, desconectar esse nó do seu ascendente. Se o seu ascendente não for uma raiz ele será marcado como **loser**. Se ele já tiver sido marcado anteriormente também será desconectado da sua árvore e se tornará uma árvore nova adicionada ao fim da heap. E assim sucessivamente até chegarmos a um nó não marcado ou à raiz. O tempo de execução amortizado é O(1).



Por último, a operação **delete** pode ser implementada simplesmente diminuindo-se a chave do elemento a ser deletado para -∞ e em seguida executando-se a operação **extract minimum**. O tempo de execução amortizado é O(log n).