Kopce Fibonacciego

3 Struktura kopców Fibonacciego

Podobnie jak kopce dwumianowe, kopce Fibonacciego są zbiorami drzew, których wierzchołki pamiętają elementy zgodnie z porządkiem kopcowym. Teraz jednak drzewa nie muszą być drzewami dwumianowymi.

Przyjmujemy taki sam sposób pamiętania drzew i elementu minimalnego, jak w przypadku kopców dwumianowych (wersja lazy). Ponadto w każdym wewnętrznym wierzchołku kopca pamiętamy wartość logiczną, mówiacą czy wierzchołek ten utracił jednego ze swoich synów w wyniku operacji *cut* -

4 Operacje

Operacje *makeheap, insert, findmin* i *meld* wykonujemy w taki sam sposób jak na kopcach dwumianowych.

4.1 Operacja $cut(h, p)$

Operacja ta zastosowana do wierzchołka wewnętrznego (tj. takiego, który nie jest korzeniem) wskazywanego przez p, odcina go od swojego ojca p' i dołącza (operacją *meld* poddrzewo zakorzenione w p do listy drzew kopca. Jeśli p jest pierwszym synem jakiego utracił p', to fakt ten jest zapamiętywany w p'. Jeśli p' wcześniej utracił już jakiegoś syna, to wykonujemy operację $cut(h, p')$. W ten sposób będziemy wędrować w górę drzewa odcinając odpowiednie poddrzewa tak długo, aż napotkamy korzeń lub wierzchołek, który dotąd nie utracił żadnego syna.

4.2 Operacja $decrement(h, p, \Delta)$

Zmniejszamy wartość klucza w wierzchołku wskazywanym przez p. Jeśli nowa wartość klucza zakłóca porządek kopcowy (tzn. jest mniejsza od klucza ojca wierzchołka p), wykonujemy $cut(h, p)$.

4.2.1 Zamortyzowany koszt

Teraz każdy wierzchołek ma swoje konto. Będzie ono niepuste tylko u wierzchołków, które utraciły jednego syna.
Operacji $decrement(h, p, \Delta)$ przydzielamy 4 jednostki kredytu. Jedną jednostką opłacamy koszt instrukcji niskiego poziomu i operację *meld* przyłączenia drzewa o korzeniu w p do kopca. Drugą umieszczamy na koncie tego drzewa (obowiązuje nas w dalszym ciągu niezmiennik kredytowy, mówiący, iż na koncie każdego drzewa kopca znajduje się jedna jednostka). Dwie pozostałe jednostki wykorzystujemy tylko wtedy, gdy wykonujemy $cut(h, p)$ i p jest pierwszym synem odciętym od swojego ojca. Umieszczamy je wówczas na koncie ojca p. Jednostki te są wykorzystywane do opłacenia operacji *cut* wykonanej wskutek tego, że ojciec p straci drugiego syna.

4.3 Operacja $deletemin(h)$

Deletemin wykonujemy w sposób analogiczny jak w przypadku kopców dwumianowych. W szczególności podczas redukcji łączymy drzewa o jednakowym rzędzie (zdefiniowanym jako liczba synów korzenia), otrzymując drzewo o stopniu o jeden wyższym. Jedyna różnica wynika z tego, że teraz drzewa nie są dwumianowe i nie można oczekiwać, że łączone drzewa będą identyczne.

Aby wykazać, że $O(\log n)$ nadal ogranicza czas wykonywania tej operacji musimy dowieść, że stopień wierzchołków drzew występujących w kopcach Fibonacciego jest ograniczony przez $O(\log n)$. Oczywiście będzie to także ograniczeniem na liczbę różnych rzędów drzew.

Lemat 1 *Dla każdego wierzchołka x kopca Fibonacciego o rzędzie k, drzewo zakorzenione w x ma rozmiar wykładniczy względem k.*

DOWÓD: Niech x będzie dowolnym wierzchołkiem kopca i niech $y_1, \ldots, y_k$ będą jego synami uporządkowanymi w kolejności przyłączania ich do x. W momencie przyłączania y_i do x-a, x miał co najmniej $i-1$ synów. Stąd y_i też miał wówczas co najmniej $i-1$ synów, ponieważ przyłączane są tylko drzewa o jednakowym rzędzie. Od tego momentu y_i mógł stracić co najwyżej jednego syna, ponieważ w przeciwnym razie zostałby odcięty od x-a. Tak więc w każdym momencie i-ty syn każdego wierzchołka ma rząd co najmniej $i-2$.
Oznaczmy przez F_i najmniejsze drzewo o rzędzie i, spełniające powyższą zależność. Łatwo sprawdzić, że F_0 jest drzewem jednowierzchołkowym, a F_i składa się z korzenia oraz i poddrzew: $F_0, F_0, F_1, F_2, \ldots, F_{i-2}$. Tak więc liczba $|F_i|$ wierzchołków takiego drzewa jest nie mniejsza niż $1 + \sum_{j=0}^{i-2} |F_j|$, co, jak łatwo pokazać indukcyjnie, jest równe i-tej liczbie Fibonacciego. Stąd liczba wierzchołków w drzewie o rzędzie k jest nie mniejsza niż ϕ^k, gdzie $\phi = (1+\sqrt{5})/2$. □

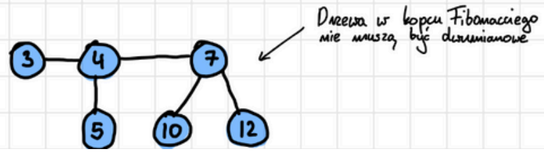
Wniosek 1 *Każdy wierzchołek w n-elementowym kopcu Fibonacciego ma stopień ograniczony przez $O(\log n)$.*

4.3.1 Operacja $delete(h, p)$

Operację $delete(h, p)$ można wykonać najpierw ustanawiając w p minimum kopca (poprzez operację $decrement(h, p, -\infty)$) a następnie usuwając minimum. Zamortyzowany koszt wynosi $O(\log n)$.
UWAGA: W ten sam sposób możemy wykonywać *delete* na kopcach dwumianowych. Oczywiście wówczas *decrement* musi polegać na przesunięciu zmniejszonego elementu do korzenia drzewa.

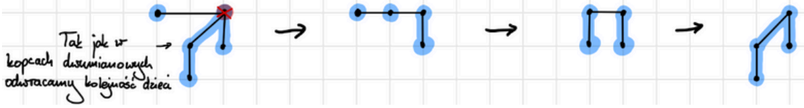
Kopiec Fibonacciego jest podobny do leniwej implementacji kopca dwumianowego, jednakże drzewa w tej strukturze niekoniecznie muszą być dwumianowe.

Przykład

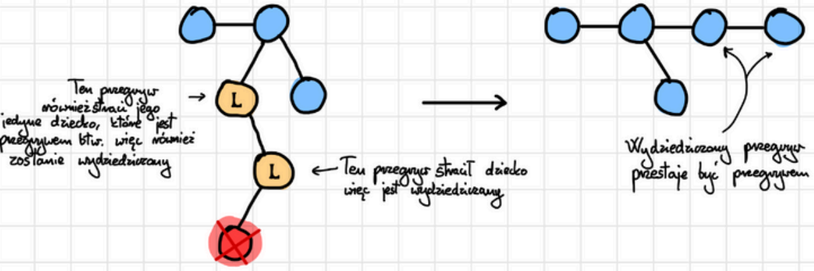


Cechy kopca Fibonacciego

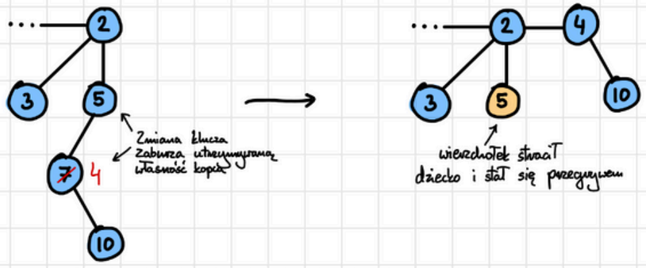
- Dodawanie elementu jest po prostu dodawaniem nowego wierzchołka do listy drzew.
- Delete min - usuwamy wierzchołek, dzieci są teraz nowymi drzewami, a następnie scalamy **wszystkie** drzewa o tym samym stopniu.



- Wierzchołek, którego opuszczono conajmniej jedno dziecko staje się przegrywem L (looser).
- Wierzchołek, który jest przegrywem L straciT następne dziecko - jest wydziedziczony (czyli przeniesiony do listy wiązanej)



- Korzeń nie może być przegrywem
- Decrease key - przenosimy ten węzeł wraz z jego dziećmi do listy wiązanej jako nowe drzewo jeśli jest mniejszy niż jego rodzic:



Implementacja w Dijkstra

W algorytmie Dijkstry wykonujemy |E| razy decrease key na kolejce priorytetowej, czyli sprawdzamy każdą krawędź, która może zmniejszyć odległość do jakiegoś wierzchołka.

Dodatkowo |V| razy wykonujemy delete_min, aby usunąć następny najbliższy wierzchołek po sprawdzeniu wszystkich niesprawdzonych dotychczas krawędzi z odwiedzonych przez nas wierzchołków.

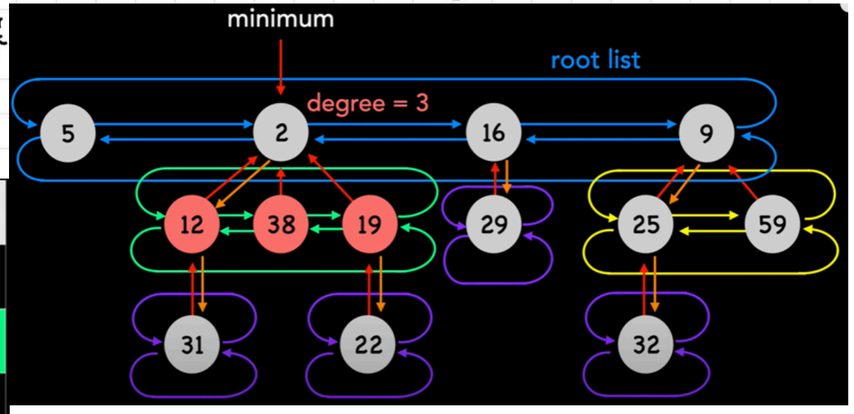
Operation	Binary Heap (worst-case)	Fibonacci Heap (amortized)
make-heap	Θ(1)	Θ(1)
insert	Θ(log n)	Θ(1)
minimum	Θ(1)	Θ(1)
extract-min	Θ(log n)	O(log n)
union	Θ(n)	Θ(1)
decrease-key	Θ(log n)	Θ(1)
delete	Θ(log n)	O(log n)

```
procedure Dijkstra
    X ← {s}
    D(s) ← ∅
    for each u ∈ V \ {s} do D(u) ← l(s,u)
    while X ≠ V do
        Niech u ∈ V \ X o minimalnej wartości D(u)
        X ← X ∪ {u}
        for each ⟨u,v⟩ ∈ E takiej, że v ∈ V \ X do
                D(v) ← min(D(v), D(u) + l(u,v))
```

Dlatego kopce Fibonacciego są tutaj tak wartościowe, ponieważ decrease_key wykonujemy w O(1), a delete_min zajmie nam O(log n).

Zatem złożoność Dijkstry wynosi

$$O(|E| + |V| \log |V|)$$



```python
class FibonacciHeap():
    def __init__(self):
        self.n = 0
        self.min = None
        self.root_list = None
class Node():
    def __init__(self, key):
        self.key = key
        self.degree = 0
        self.mark = False

        # pointers
        self.parent = None
        self.child = None
        self.left = None
        self.right = None
```

fibonacci heap

- data structure that implements
  1. make-heap
  2. insert
  3. minimum
  4. extract-min
  5. union
- additional methods:
  6. decrease-key
  7. delete

```python
def insert(self, key):
    node = Node(key)
    node.left = node
    node.right = node

    self.merge_with_root_list(node)

    if self.min is None or node.key < self.min.key:
        self.min = node

    self.n += 1
    return node
def merge_with_root_list(self, node):
    if self.root_list is None:
        self.root_list = node
    else:
        node.right = self.root_list
        node.left = self.root_list.left
        self.root_list.left.right = node
        self.root_list.left = node
```

(insert w skrócie w skrócie tworzy nowego noda i dodaje go do listy korzeni odpowiednio przepinając wskaźniki)

```python
def minimum(self):
    return self.min
```

```python
def extract_min(self):
    z = self.min
    if z is not None:
        if z.child is not None:
            children = [c for c in self.iterate(z.child)]
            for c in children:
                self.merge_with_root_list(c)
                c.parent = None

        self.remove_from_root_list(z)

        if z == z.right:
            self.min = None
            self.root_list = None
        else:
            self.min = z.right
            self.consolidate()

        self.n -= 1
    return z
```

```python
def union(self, FH2):
    FH = FibonacciHeap()
    FH.root_list = self.root_list

    # set min to lesser of FH1.min and FH2.min
    FH.min = self.min if self.min.key < FH2.min.key else

    # fix pointers to combine root lists
    last = FH2.root_list.left
    FH2.root_list.left = FH.root_list.left
    FH.root_list.left.right = FH2.root_list
    FH.root_list.left = last
    FH.root_list.left.right = FH.root_list

    # update total nodes
    FH.n = self.n + FH2.n

    return FH
```

(union w skrócie łączy ze sobą listy korzeni obu drzew i odpowiednio ustawia wskaźnik min)

extract min w skrócie usuwa najmniejszą wartość w kopcu dodatkowo mergując wszystkie drzewa o tych samych stopniach

```python
def consolidate(self):
    A = [None] * int(math.log(self.n) * 2)
    nodes = [w for w in self.iterate(self.root_list)]
    for w in range(0, len(nodes)):
        x = nodes[w]
        d = x.degree
        while A[d] != None:
            y = A[d]
            if x.key > y.key:
                temp = x
                x, y = y, temp
            self.heap_link(y, x)
            A[d] = None
            d += 1
        A[d] = x
    for i in range(0, len(A)):
        if A[i] is not None:
            if A[i].key < self.min.key:
                self.min = A[i]
```

```python
def decrease_key(self, x, k):
    if k > x.key:
        return None
    x.key = k
    y = x.parent
    if y is not None and x.key < y.key:
        self.cut(x, y)
        self.cascading_cut(y)
    if x.key < self.min_node.key:
        self.min_node = x
```

```python
def cut(self, x, y):
    self.remove_from_child_list(y, x)
    y.degree -= 1
    self.merge_with_root_list(x)
    x.parent = None
    x.mark = False
```

```python
def cascading_cut(self, y):
    z = y.parent
    if z is not None:
        if y.mark is False:
            y.mark = True
        else:
            self.cut(y, z)
            self.cascading_cut(z)
```

decrese_ key w skrócie zmniejsza wartość konkretnego noda do oczekiwanej wartości po czym ucina wszystkie nody które zaburzają strukture kopca bądź mają atrybut mark = True czyli kiedyś już utraciły dziecko (wykonuje się to rekurencyjnie zaczynając odd wskazanego noda idąc do korzenia

```python
def delete(self, x):
    self.decrease_key(x, float('-inf'))
    return self.extract_min()
```

O ile co najwyżej może zwiększyć się liczba drzew w kopcu Fibonacciego w skutek wykonania pojedynczej operacji decreasekey.

▼ Rozwiązanie

O(n) gdy będziemy mieli jedno drzewo w postaci listy, gdzie każdy wierzchołek(korzeń i liść nie muszą spełniać tego warunku) miał już uciętego syna.

Na czym polega opracja kaskadowego odcinania w kopcach Fibonacciego?

Żeby zachować niezmienniki potrzebne do amortyzowanej złożoności, nie pozwalamy w operacji odcięcia, usunięcia więcej niż jednego poddrzewa z jednego wierzchołka. Dlatego zaznaczamy ojca, który utracił jednego syna i przy próbie usunięcia mu kolejnego syna usuwamy całe poddrzewo i dodajemy je do ogólnej listy. Po tym markujemy jego dziadka (jeżeli jest zmarkowany, to znowu usuwamy poddrzewo i rekurencyjnie w gore).

Jakiego rodzaju kopca użyć dla asymptotycznie najefektywniejszego zaimplementowania algorytmu:

a) Prima,

b) Dijkstry

▼ Rozwiązanie

b) Prima - kopiec fibbonaciego, zlozonosc: O(E + V*logV)

b) Dijkstry - kopiec fibbonaciego, zlozonosc: O(E + V*logV)

Ile maksymalnie drzew może znaleźć się w:

- Kopcu dwumianowym(wersja eager)
- Kopcu fibonacciego Zawierającym n elementów

▼ Rozwiązanie

- Kopiec dwumianowy składa się z drzew o wielkościach będących potęgą dwójki. Dla każdej potęgi dwójki może istnieć tylko jedno drzewo, w przeciwnym razie łączymy operacją join dwa drzewa o takiej samej ilości wierzchołków, żeby otrzymać dwa razy większe drzewo. Dodawanie wierzchołków oraz operacja meld w tym kopcu działa na zasadzie dodawnia liczb binarnych. Stąd mając i rząd największego drzewa, największą ilość drzew mamy, gdy istnieje każde mniejsze drzewo od B_i. np. dla n = 15, będzie to B_0 mający 1 wierzchołek, B_1 mający 2 wierzchołki, B_2 mający 4 wierzchołki, B_3 mający 8 wierzchołków. stąd log(n) to największa ilość drzew dwumianowych w wersji eager.
- Dodajemy wierzchołki leniwie, to znaczy dodajemy je do roota kopca. W takim razie nie robiąc żadnej innej operacji poza insert, otrzymujemy kopiec z `n` drzewami.

W jakim czasie działa algorytm Kruskala, jeśli:

- krawędzie podane są w kolejności rosnących wag;
- kolejka priorytetowa zaimplementowana jest przy pomocy kopca Fibonacciego.

Odpowiedź uzasadnij. *Uwaga: Oba te warunki są spełnione jednocześnie.*

(a) Jaką dodatkową operację umożliwiają kopce Fibonacciego względem dwumianowych w wersji Lazy?

(b) Jakie z tego powodu wynikają różnice w budowie tych kopców?

▼ Rozwiązanie

(a)

Operacja `decrement`

(b)

Każdy wierzchołek zawiera dodatkowe pole ze wskaźnikiem na ojca

Komentarz do zadania

To jest odpowiedź poprawna dla kopców poznanych na wykładzie. W wielu źródłach można znaleźć, że kopiec dwumianowy posiada operację `decrement` oraz pole ze wskaźnikiem na ojca.

Podaj definicję rzędu drzew w kopcach Fibonacciego

- Górne ograniczenie na ten rząd
- Ideę dowodu tego ograniczenia

Czy wysokość drzew powstałych w kopcu Fibonacciego o n wierzchołkach da się ograniczyć przez $log^2 n$?

▼ Rozwiązanie

Ładny dowód pokazujący, że log(n) ogranicza wysokość drzew Fibonacciego. A log ^2 (n) jest większe od log (n), zatem tym bardziej też ogranicza.

W którym z kopców pamiętających po n kluczy może być więcej drzew: w kopcu dwumianowym (wersja lazy) czy kopcu Fibonacciego?

▼ Rozwiązanie

W obydwu kopcach może być tyle samo drzew - dokładnie n.