Algorytmy i Struktury Danych

Kolejka Priorytetowa

(c) Marcin Sydow

Zawartość wykładu

- Definicja kolejki priorytetowej
- proste implementacje (nieefektywne)
- kopiec binarny (najprostsza efektywna implementacja)
- operacje kolejki priorytetowej na kopcu binarnym
- trik: implementacja kopca jako zwykłej tablicy
- szybkie konstruowanie kopca binarnego
- przykłady zastosowań
- rozszerzenia kolejek priorytetowych
- podsumowanie

Definicja Kolejki Priorytetowej

Kolejka Priorytetowa (ang. Priority Queue (PQ)) to abstrakcyjna struktura danych do przetwarzania elementów wraz z przypisanymi **priorytetami** (numerycznymi) zdefiniowana przez następujące operacje (interfejs):

- insert(e, p) // dodaj element e o priorytecie p do kolejki
- findMin() // // zwróć (bez usuwania) element o priorytecie najpilniejszym spośród przechowywanych obecnie w kolejce
- delMin() // //zwróć (z usuwaniem) element j.w.

Każdy element ma przypisany priorytet - liczbę całkowitą. Interpretacja jest następująca: im niższa liczba tym wyższy priorytet

Czasami rozważana jest odwrotna interpretacja: im wyższa liczba tym wyższy priorytet. W takim wypadku kolejkę nazywamy "typu max" a nazwy operacji są zmodyfikowane na `findMax` i `delMax`.

Kolejka vs kolejka priorytetowa?

Czy kolejka priorytetowa jest specjalnym rodzajem kolejki?

Kolejka vs kolejka priorytetowa?

Czy kolejka priorytetowa jest specjalnym rodzajem kolejki?

Nie: bo jej zbiór operacji *nie zawiera* zbioru operacji kolejki (jest to po prostu inny zbiór operacji)

Implementacje Kolejki Priorytetowej

Proponowano wiele, coraz bardziej efektywnych i wyrafinowanych implementacji kolejki priorytetowej:

- "naiwne" (nieefektywne) implementacje (za pomocą tablic lub list dowiązaniowych)
- kopiec binarny (ang. Binary Heap, 1964 Williams; Floyd 1964)
- drzewo vEB* (van Emde Boas, 1975)
- kopiec dwumianowy (ang. Binomial Heap, 1978 Vuillemin)
- Pairing Heap* (1986 Fredman, Sedgewick, Sleator, Tarjan; 2000 Iacono; Pettie 2005)
- kopiec Fibonacciego* (1987 Fredman, Tarjan)
- "Thin Heap","Fat Heap"* (1999 Kaplan, Tarjan), etc.

* - nie omawiane w tym kursie

Implementacje "naiwne"

Jak zwykle, zaczniemy od omównienia bezpośredniej (naiwnej) implementacji kolejki priorytetowej: za pomocą tablic[1]. Są 2 możliwe warianty: trzymanie priorytetów posortowanych i nieposortowanych:

- priorytety nieposortowane:
  insert: O(1), deleteMin: **O(n)**, construct: **O(n)**
- priorytety posortowane:
  insert: **O(n)**, deleteMin: O(1): construct: **O(n log(n))**

Jak widać, w obu wariantach niektóre operacje mają nieakceptowalnie wysoką złożoność.

Potrzeba bardziej pomysłowej implementacji, która zapewni złożoność niższą niż liniowa.

[1] implementacja za pomocą listy dowiązaniowej nie polepsza sytuacji

Kopiec binarny (ang. Binary Heap)

Kopiec binarny jest **binarnym drzewem zupełnym**. Priorytety (wraz z odpowiadającymi im elementami) trzymane są w węzłach drzewa, oraz zachodzi następujący **warunek porządku kopca**:

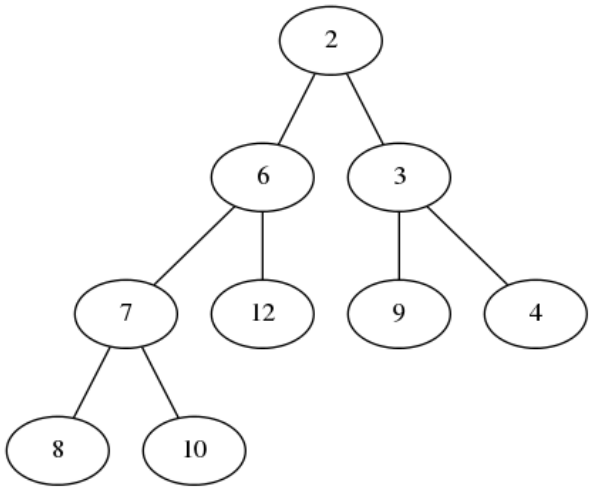
Dla każdego węzła, priorytet w tym węźle jest mniejszy (pilniejszy) lub równy, priorytetom w węzłach-potomkach.

Dodatkowa własność: **zupełność** drzewa oznacza, że kopiec wypełniany jest od góry do dołu i na każdym nowym poziomie od lewej do prawej (tzn. pierwsze wolne miejsca to zawsze skrajnie lewe wolne miejsca na ostatnim poziomie).

Kopiec binarny jest najprostszą znaną konkretną strukturą danych, która pozwala zaimplementować kolejkę priorytetową przy złożoności czasowej operacji nieprzekraczającej logarytmicznej.

Przykład kopca i własności

Przykład kopca (w węzłach pokazano tylko priorytety):



Obserwacje:

- z warunku porządku kopca wynika, że minimalny (najpilniejszy) priorytet jest zawsze w korzeniu kopca
- z w/w warunku wynika też, że priorytety na każdej ścieżce od korzenia do liścia stanowią ciąg niemalejący
- uwaga: priorytety na danym poziomie nie są posortowane
- z warunku zupełności wynika, że wysokość n-elementowego kopca jest zawsze $\Theta(log(n))$

Implementacja operacji kolejki priorytetowej na kopcu binarnym

- insert(e): dodaj nowy element e (wraz z jego priorytetem) w pierwszym od lewej wolnym węźle x ostatniego poziomu, a następnie przywróć porządek kopca począwszy od węzła x w górę (operacja upheap)
- findMin(): zwróć element e znajdujący się w korzeniu kopca
- delMin(): usuń element z korzenia, wstaw do korzenia element ostatni kopca (skrajnie prawy na ostatnim poziomie kopca) i przywróć porządek kopca począwszy od korzenia w dół (operacja downheap)

Przywracanie porządku kopca za pomocą operacji upheap i downheap

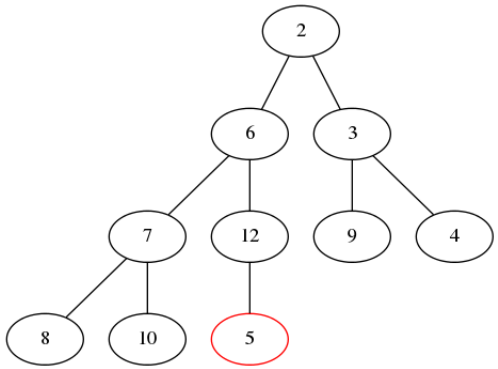
Operacje kolejki priorytetowej zaimplementowane są na kopcu za pomocą 2 pomocniczych operacji wewnętrznych, które mają na celu przywracanie porządku kopca po ewentualnym jego zaburzeniu z powodu modyfikacji w węźle x:

- upheap(x): zaczynając od węzła x, przywróć warunek porządku kopca podążając w górę od x
- downheap(x): zaczynając od węzła x, przywróć warunek porządku kopca podążając w dół od x

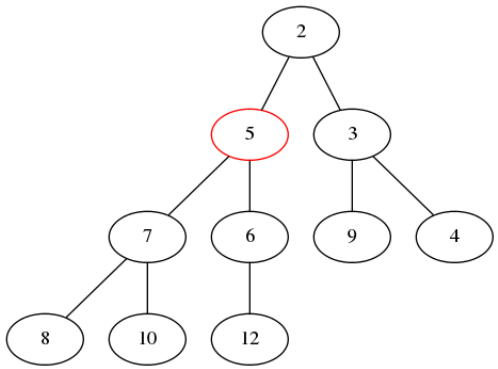
Uwaga: obie operacje zakładają poprawność warunku kopca dla wszystkich priorytetów poza węzłem x, którego priorytet ewentualnie jako jedyny może zakłócać porządek kopca.

Przykład działania operacji insert

Dodajemy do przykładowego kopca nowy element o priorytecie "5":



Przywracamy porządek kopca operacją upheap przesuwając nowo-dodany element w górę do momentu aż jego rodzic nie będzie większy (lub gdy dotrzemy do korzenia)[2]:

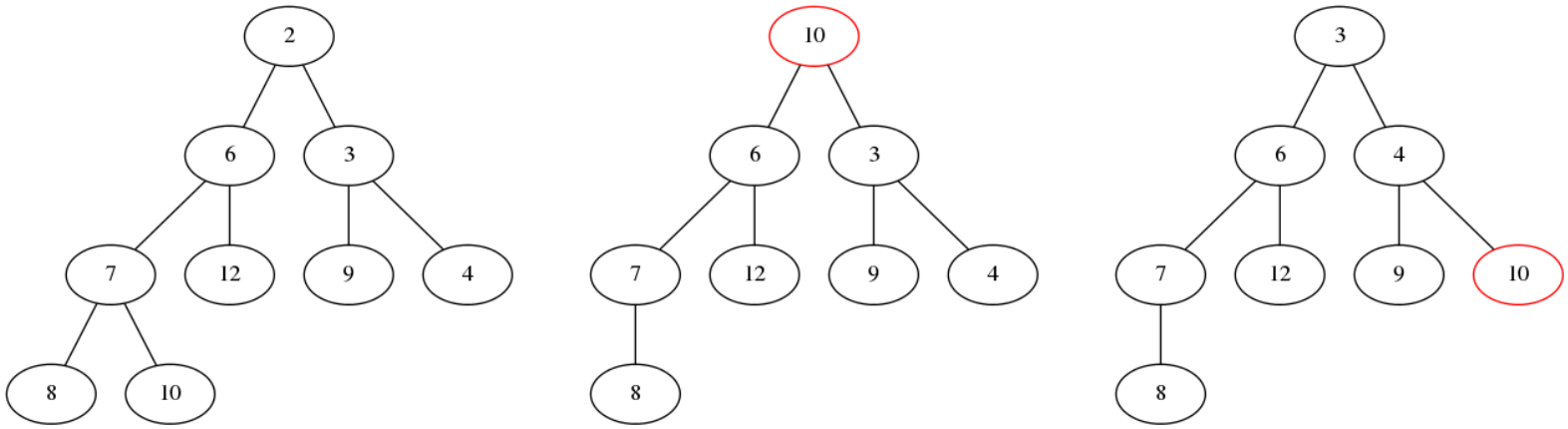


Porządek kopca został przywrócony.

[2] analogia do wstawiania w algorytmie insertionSort

Przykład działania operacji deleteMin

1. oryginalny kopiec (przypomnienie)
2. z oryginalnego kopca usuwamy korzeń i zastępujemy go ostatnim węzłem w kopcu
3. Przywracamy porządek kopca operacją downheap począwszy od korzenia zamieniając nowo-dodany element z minimum z jego obu synów, chyba że obaj synowie mają już niemniejsze priorytety (lub doszliśmy do ostatniego poziomu)



Złożoność czasowa operacji kolejki priorytetowej na kopcu binarnym

rozmiar danych: liczba elementów w kopcu (n)
operacja dominująca: porównanie priorytetów)

Zauważmy, że operacje wykonują co najwyżej 1 (upheap) lub 2 (downheap) porównania na każdym poziomie kopca, a więc łączna liczba porównań jest proporcjonalna co najwyżej do wysokości kopca, która wynosi $O(log(n))$

Wobec tego złożoności na kopcu binarnym są następujące:

- insert(x): dodaj x na dole i wykonaj upheap ($O(log(n))$)
- findMin(): zwróć korzeń ($O(1)$)
- delMin(): zastąp korzeń ostatnim elementem i wykonaj downheap z korzenia ($O(log(n))$)

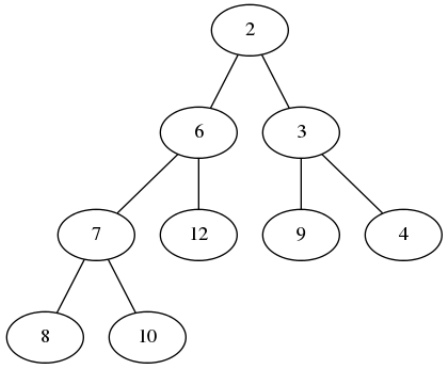
Jak zaimplementować kopiec binarny?

Ogólnie można zaimplementować każde drzewo binarne (a więc i kopiec binarny) jako strukturę dowiązaniową. W przypadku kopca każdy węzeł zawierałby: element, priorytet, oraz wskaźniki (dowiązania) do rodzica i obu synów (NULL jeśli brak).

Można jednak skorzystać z własności zupełności i dzięki temu można "sprytnie" zaimplementować kopiec **za pomocą zwykłej tablicy**, co bardzo upraszcza kod operacji i oszczędza pamięć na wskaźniki.

Implementacja kopca binarnego za pomocą tablicy

Dzięki zupełności, elementy kopca można wpisać do tablicy (pozostawiając nieużywany indeks 0) i idąc w porządku od góry do dołu i od lewej do prawej na każdym poziomie następująco, np:



indeksy:	0	1	**2**	3	4	**5**	6	7	8	9
zaw. tablicy:	n	2	**6**	3	7	**12**	9	4	8	10

Nawigacja w tablicy jest następująca: indeks rodzica (oprócz korzenia) jest 2 razy mniejszy, indeks lewego syna 2 razy większy, a prawego dodatkowo plus 1:

- $parent[i] == i/2$ (dzielenie całkowitoliczbowe: ucinamy ułamek)
- $i.left == 2i$, $i.right == 2i + 1$ (poza liścmi)

np. indeks węzła zawierającego rodzica 12 to $5/2 = 2$.

Pseudokod operacji upheap

(zakładamy reprezentację kopca w tablicy)

```
upheap(i)          // i > 0
  key = heap[i]
  parent = i/2
  while((parent > 0) && (heap[parent] > key))
    heap[i] = heap[parent]
    i = parent
    parent /= 2
  heap[i] = key
```

Pseudokod operacji downheap

(zakładamy reprezentację kopca w tablicy)

```
downheap(i)
  l = 2i           // lewy syn
  r = 2i + 1       // prawy syn
  if l <= n and heap[l] < heap[i]:
    min = l
  else:
    min = i
  if r <= n and heap[r] < heap[min]: // n to rozmiar kopca
    min = r
  if min != i:
    swap(i,min)    // zamiana elementów, nie samych indeksów
    downheap(min) // kontynuuj niżej
```

Uwaga: rekurencja jest tu użyta tylko, aby skrócic prezentację kodu, można to oczywiście zapisać iteracyjnie, unikając rekursji

Szybkie konstruowanie kopca (construct)

Załóżmy, że znamy od razu n elementów, które mają być wstawione do pustej kolejki priorytetowej.

Bezpośrednie wstawianie pojedynczo n razy (w przypadku kopca binarnego) dałoby złożoność $n \times insert$ (czyli w sumie $\Theta(nlog(n))$)

Szybka operacja `construct` pozwala to zrobić efektywniej: Najpierw wstawiamy wszystkie elementy w podanej kolejności do tablicy a następnie wykonujemy następującą procedurę:

```
for(i = n/2; i > 0; i--) downHeap(i)
```

Można udowodnić, że taka "sprytniejsza" procedura daje łączną złożoność czasową zaledwie $O(n)$

Przykładowe zastosowanie: algorytm HeapSort

Oto jak można prosto posortować ciąg używając kolejki priorytetowej:
(zakładamy, że pq jest obiektem reprezentującym kolejkę priorytetową)

```
while(s.hasNext())
  pq.insert(s.next())

while(!pq.isEmpty())
  result.add(pq.delMin())
```

rozmiar danych: liczba elementów (n), op. dom.: porównanie
złożoność czasowa: $\Theta(nlog(n))$

Ilustruje to potęgę używania abstrakcyjnych struktur danych do upraszczania algorytmów.
Uwaga: powyższy algorytm, przy użyciu kopca binarnego, ma nieco wyższą stałą multiplikatywną niż algorytm QuickSort.

Inne przykładowe zastosowania kolejek priorytetowych

Kolejki priorytetowe są standardowo używane w wielu algorytmach (tzw. *zachłannych*) np.:

- obliczanie drzewa optymalnego drzewa kodowania Huffmana
- algorytm Dijkstry obliczania najkrótszych ścieżek w grafie
- algorytm Prima znajdowania minimalnego drzewa rozpinającego
- etc.

Rozszerzenia kolejek priorytetowych

Dynamiczna kolejka priorytetowa (2 dodatkowe operacje):

- construct(sequence<T> s)
- H insert(T e) // zwraca wskaźnik do wstawionego elementu
- T findMin()
- T delMin()
- **decreaseKey(H pointer, T newPriority)** // (zmniejsz priorytet danego elementu)
- **delete (H pointer)** (usuń dany element)

Złączalna kolejka priorytetowa (wszystko powyżej i dodatkowo):

- **merge(PQ priorityQueue1, PQ priorityQueue2)** // złącz podane dwie kolejki w jedną

Powyższe operacje można oczywiście zaimplementować na kopcu binarnym, ale istnieją bardziej wyrafinowane implementacje, które zapewniają lepsze złożoności czasowe

Złożoność czasowa operacji kol. prior. w różnych implementacjach

Jedną z implementacji, która pozwala np. na efektywniejszą implementację operacji `merge` jest tzw. **kopiec dwumianowy**[3]

Poniżej podsumowano porównanie rzędów złożoności operacji na kilku przykładach implementacji kolejki priorytetowej (w tym 2 "naiwnych"):

operacja	nieposort.	posort.	kopiec binarny	**kopiec dwumianowy**
insert	1	n	lg n	lg n
findMin	n	1	1	lg n
delMin	n	1	lg n	lg n
decreaseKey	1	n	lg n	lg n
delete	1	n	lg n	lg n
merge	1	n	**n**	**lg n**

(wszystkie elementy otoczone są symbolem $O(.)$)

[3] ang. binomial heap – nie jest omawiany w tym wykładzie

Podsumowanie/przykładowe zadania:

- definicja kolejki priorytetowej
- deficja kopca binarnego
- dokonać analizy złożoności naiwnych implementacji kol. prior.
- wykonać każdą z operacji kol. prior. na podanym kopcu binarnym i pokazać na rysunku jak wygląda po tej operacji
- zastosować szybką procedurę `construct` do podanego ciągu elementów i narysować jak wygląda skonstruowany kopiec binarny
- reprezentacja tablicowa kopca binarnego
- dokonać analizy złożoności czasowej operacji kol. prior. w przypadku implementacji na kopcu binarnym
- przykłady zastosowań kolejek priorytetowych
- rozszerzenia kolejki priorytetowej
- dokonać analizy złożoności czasowej operacji decreaseKey, delete oraz merge na przypadku kopca binarnego.

Dziękuję za uwagę