Kopce dwumianowe

IIUWr. II rok informatyki. Przygotował: Krzysztof Loryś

29 Definicja

Kopce dwumianowe są strukturą danych umoźliwiającą łatwe wykonywanie zwykłych operacji kopcowych (insert, makeheap, findmin i deletemin) a ponadto operacji *meld* łączenia kopców.

Definicja 14 Drzewa dwumianowe *zdefiniowane są indukcyjnie: i-te drzewo dwumianowe B_i składa się z korzenia oraz i poddrzew: B_0, B_1, ..., B_{i-1}.*

Definicja 15 Kopiec dwumianowy *to zbiór drzew dwumianowych, które pamiętają elementy z uporządkowanego uniwersum zgodnie z porządkiem kopcowym.*

Definicja 16 $\forall_{x-wierzchoka}$ *rząd(x)=liczba dzieci x-a.*
$\forall_{T-drzewa}$ *rząd*(T)=*rząd*(*korzeń*(T)).

Szczegół implementacyjny: Aby umoźliwić szybką realizację operacji na kopcu dwumianowym, będziemy zakładać, źe dzieci każdego wierzchołka zorganizowane są w cykliczną listę dwukierunkową, a ojciec pamięta wska"xnik do jednego z nich (np. do dziecka o najmniejszym rzędzie).

30 Operacje na kopcach dwumianowych

30.1 Łączenie drzew dwumianowych - operacja *join*

Dwa drzewa B_i łączymy ze sobą tak, źe korzeń jednego drzewa staje się synem korzenia drugiego drzewa. W ten sposób otrzymujemy drzewo B_{i+1}.

Uwagi:

(a) Nigdy nie będziemy łączyć drzew o różnych rzędach.

(b) Zawsze podłączamy to drzewo, którego korzeń pamięta mniejszą wartość do tego, którego korzeń pamięta większą wartość.

RYSUNEK - będzie pó"xniej

Koszt: $O(1)$.

30.2 Operacja *makeheap(i)*

Bez komentarza. Koszt - $O(1)$.

30.3 Operacja *findmin*

Z każdym kopcem dwumianowym wiążemy wska"xnik *MIN* wskazujący na minimalny element. Operacja findmin polega na odczytaniu tego elementu. Stąd jej koszt wynosi $O(1)$.

30.4 Operacja *insert(i,h)*

Wykonujemy $meld(h, makeheap(i))$.
Koszt tej operacji zależny jest od kosztu *meld*. Podamy go dalej.

30.5 Operacja *deletemin(h)*

Sposób jej wykonania zależy od realizacji *meld*. Omówimy go dalej.

30.6 Operacja *meld*

Rozważymy dwie metody realizacji operacji *meld*:

(a) wersja "*eager*" - w tej wersji kopiec przybiera docelowy kształt przed wykonaniem następnej po meld operacji;

(b) wersja "*lazy*" - w tej wersji pozwalamy, by kopiec utracił strukturę kopca dwumianowego; zostanie ona mu przywrócona dopiero podczas wykonania operacji deletemin.

30.6.1 Eager meld(h,h')

W tej wersji drzewa kopca dostępne są poprzez tablicę wska"xników (będziemy ją oznaczać tą samą nazwą co kopiec). Każdy kopiec zawiera co najwyżej jedno drzewo każdego rzędu. i-ty wska"xnik jest albo pusty albo wskazuje na drzewo i-tego rzędu.
$Meld(h, h')$ tworzy nowy kopiec H; stare kopce ulegają likwidacji.

Procedure Eagermeld(h,h')
  if $key(MIN_h) < key(MIN_{h'})$ **then** $MIN_H \leftarrow MIN_h$ **else** $MIN_H \leftarrow MIN_{h'}$
  $carry \leftarrow nil$;
  for $i \leftarrow 0$ **to** $maxheapsize$ **do**
    $k \leftarrow$ # wskaźników $\neq nil$ pośród $\{carry, h[i], h'[i]\}$
    case k **of**
      0: $H[i] \leftarrow nil$
      1: $H[i] \leftarrow$ jedyny niepusty wskaźnik pośród $\{carry, h[i], h'[i]\}$
      2: $H[i] \leftarrow nil; carry \leftarrow join(B^1, B^2)$
                gdzie B^1 i B^2 są drzewami wskazywanymi przez
                dwa niepuste wskaźniki spośród $\{carry, h[i], h'[i]\}$)
      3: $H[i] \leftarrow h[i]; carry \leftarrow join(h'[i], carry)$

Koszt: $O(\log n)$. Korzystamy tu z prostego faktu:

Fakt 21 *Kopiec zawierający n elementów składa się z co najwyżej $\log n$ różnych drzew dwumianowych.*

Operacja *deletemin(h).*

Wska"xnik MIN wskazuje na drzewo dwumianowe B, którego korzeń zawiera najmniejszy element. W stałym czasie usuwamy B z h. Następnie usuwamy korzeń z drzewa B otrzymując rodzinę drzew $B_0, B_1, \ldots, B_{\text{rząd}(B)-1}$. Z drzew tych tworzymy kopiec dwumianowy h' i wykonujemy $meld(h, h')$.

Koszt: $O(\log n)$.

Operacja *Insert(i,h)*

Pojedyncza operacja insert może kosztować $\Omega(\log n)$, np. gdy h zawiera drzewa każdego rzędu. Można jednak pokazać, że czas zamortyzowany można ograniczyć do $O(1)$ (ćwiczenie).

30.6.2 Lazy meld

Chcemy, by wszystkie operacje oprócz *deletemin* kosztowały nas $O(1)$ czasu zamortyzowanego.

Zmieniamy reprezentację kopca: zamiast tablicy wska"xników, drzewa dwumianowe danego kopca łączymy w cykliczną listę dwukierunkową.

Procedura $lazymeld(h, h')$ polega na połączeniu list i aktualizacji wska"xnika MIN. Można tego dokonać w czasie $O(1)$. Teraz jednak kopiec może zawierać wiele drzew tego samego rzędu. Dopiero operacja *deletemin* redukuje liczbę drzew.

Operacja *deletemin(h)*

Usuwamy korzeń x drzewa wskazywanego przez *MIN*, dołączamy poddrzewa wierzchołka x do listy drzew kopca, uaktualniamy wska"xnik *MIN*, a następnie redukujemy liczbę drzew w kopcu. W tym celu wystarczy raz przeglądnąć listę drzew kopca (możemy roboczo wykorzystać tablicę wska"xników na wzór tablicy z wersji eager operacji *meld*).

Pojedyncza operacja *deletemin* może być bardzo kosztowna (nawet $O(n)$ - np. wtedy, gdy kopiec składa się z n drzew jednoelementowych). Pokażemy jednak, że czas zamortyzowany można ograniczyć przez $O(\log n)$.

Utrzymujemy następujący niezmiennik kredytowy:

Każde drzewo kopca ma 1 jednostkę kredytu na swoim koncie.

Operacjom przydzielamy następujące kredyty:

makeheap	-	2
insert	-	2
meld	-	1
findmin	-	1
deletemin	-	$2\log n$

Kredyty te wystarczają na wykonanie instrukcji niskiego poziomu, związanych z realizacją operacji oraz na utrzymanie niezmiennika kredytowego:

- Operacje *meld* i *findmin* nie zmieniają liczby drzew w kopcach i wykonują się w stałym czasie.

- Operacje *insert* i *makeheap* także wykonują się w stałym czasie, ale tworzą nowe drzewo. Jedna jednostka kredytu przydzielonego tym operacjom zostaje odłożona na koncie tego drzewa.

- Operacja *deletemin* może dodać co najwyżej $\log n$ drzew do kopca. Z kredytu operacji *deletemin* przekazujemy po jednej jednostce na konta tych drzew. Obliczenie nowej wartości wska"xnika *MIN* można wykonać w czasie $O(\log n)$. Podczas redukcji liczby drzew musimy przeglądnąć listę wszystkich drzew kopca i dokonać pewnej liczby operacji *join*. Koszt operacji *join* możemy pominąć, ponieważ każda taka operacja może być opłacona jednostką kredytu znajdującą się na koncie przyłączanego drzewa. Jednostką tą możemy opłacić także koszt odwiedzenia tego drzewa na liście. Odwiedzenie pozostałych drzew musimy opłacić kredytem przydzielonym operacji *deletemin*. Możemy to zrobić, ponieważ takich drzew (tj. tych, które w czasie *deletemin* nie będą podłączone do innego drzewa) jest nie więcej niż różnych rzędów, a więc $O(\log n)$.

Lecture 8 Binomial Heaps

Binomial heaps were invented in 1978 by J. Vuillemin [106]. They give a data structure for maintaining a collection of elements, each of which has a *value* drawn from an ordered set, such that new elements can be added and the element of minimum value extracted efficiently. They admit the following operations:

makeheap(i)	return a new heap containing only element i
findmin(h)	return a pointer to the element of h of minimum value
insert(h, i)	add element i to heap h
deletemin(h)	delete the element of minimum value from h
meld(h, h')	combine heaps h and h' into one heap

Efficient searching for objects is not supported.

In the next lecture we will extend binomial heaps to *Fibonacci heaps* [35], which allow two additional operations:

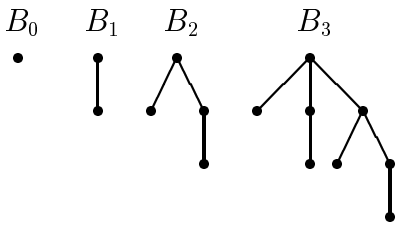
decrement(h, i, Δ)	decrease the value of i by Δ
delete(h, i)	remove i from heap h

We will see that these operations have low *amortized* costs. This means that any particular operation may be expensive, but the costs average out so that over a sequence of operations, the number of steps per operation of each type is small. The amortized cost per operation of each type is given in the following table:

makeheap	$O(1)$
findmin	$O(1)$
insert	$O(1)$
deletemin	$O(\log n)$
meld	$O(1)$ for the lazy version
	$O(\log n)$ for the eager version
decrement	$O(1)$
delete	$O(\log n)$

where n is the number of elements in the heap.

Binomial heaps are collections of *binomial trees*, which are defined inductively: the i^{th} binomial tree B_i consists of a root with i children $B_0, \ldots, B_{i-1}$.



It is easy to prove by induction that $|B_i| = 2^i$.

If data elements are arranged as vertices in a tree, that tree is said to be *heap-ordered* if the minimum value among all vertices of any subtree is found at the root of that subtree. A *binomial heap* is a collection of heap-ordered binomial trees with a pointer **min** to the tree whose root has minimum value. We will assume that all children of any vertex are arranged in a circular doubly-linked list, so that we can link and unlink subtrees in constant time.

Definition 8.1 The *rank* of an element x, denoted rank (x), is the number of children of x. For instance, rank (root of B_i) $= i$. The *rank* of a tree is the rank of its root. □

A basic operation on binomial trees is *linking*. Given two B_i's, we can combine them into a B_{i+1} by making the root of one B_i a child of the root of the other. We always make the B_i with the larger root value the child so as to preserve heap order. We never link two trees of different rank.

8.1 Operations on Binomial Heaps

In the "eager meld" version, the trees of the binomial heap are accessed through an array of pointers, where the i^{th} pointer either points to a B_i or is **nil**. The operation **meld**(h, h'), which creates a new heap by combining h and h', is reminiscent of binary addition. We start with $i = 0$. If either h or h' has a B_0 and the other does not, we let this B_0 be the B_0 of **meld**(h, h'). If neither h nor h' have a B_0, then neither will **meld**(h, h'). If both h and h' have a B_0, then **meld**(h, h') will not; but the two B_0's are linked to form a

B_1, which is treated like a carry. We then move on to the B_1's. At stage i, we may have 0, 1, or 2 B_i's from h and h', plus a possible B_i carried from the previous stage. If there are at least two B_i's, then two of them are linked to give a B_{i+1} which is carried to the next stage; the remaining B_i, if it exists, becomes the B_i of **meld**(h, h'). The entire operation takes $O(\log n)$ time, because the size of the largest tree is exponential in the largest rank. We will modify the algorithm below to obtain a "lazy meld" version, which will take constant amortized time.

The operation **insert**(i, h) is just **meld**$(h,$ **makeheap**$(i))$.

For the operation **deletemin**(h), we examine the **min** pointer to x, the root of some B_k. Removing x creates new trees $B_0, \ldots, B_{k-1}$, the children of x, which are formed into a new heap h'. The tree B_k is removed from the old heap h. Now h and h' are melded to form a new heap. We also scan the new heap to determine the new **min** pointer. All this requires $O(\log n)$ time.

8.2 Amortization

The $O(\log n)$ bound on **meld** and **deletemin** is believable, but how on earth can we do **insert** operations in constant time? Any particular **insert** operation can take as much as $O(\log n)$ time because of the links and carries that must be done. However, intuition tells us that in order for a particular **insert** operation to take a long time, there must be a lot of trees already in the heap that are causing all these carries. We must have spent a lot of time in the past to create all these trees. *We will therefore charge the cost of performing these links and carries to the past operations that created these trees.* To the operations in the past that created the trees, this will appear as a constant extra overhead.

This type of analysis is known as *amortized analysis*, since the cost of a sequence of operations is spread over the entire sequence. Although the cost of any particular operation may be high, over the long run it averages out so that the cost per operation is low.

For our amortized analysis of binomial heaps, we will set up a savings account for each tree in the heap. When a tree is created, we will charge an extra credit to the instruction that created it and deposit that credit to the account of the tree for later use. (Another approach is to use a *potential function*; see [100].) We will maintain the following *credit invariant*:

> Each tree in the heap has one credit in its account.

Each **insert** instruction creates one new singleton tree, so it gets charged one extra credit, and that credit is deposited to the account of the tree that was created. The amount of extra time charged to the **insert** instruction is $O(1)$. The same goes for **makeheap**. The **deletemin** instruction exposes up to $\log n$ new trees (the subtrees of the deleted root), so we charge an extra

$\log n$ credits to this instruction and deposit them to the accounts of these newly exposed trees. The total time charged to the **deletemin** instruction is still $O(\log n)$.

We use these saved credits to pay for linking later on. When we link a tree into another tree, we pay for that operation with the credit associated with the root of the subordinate tree. The **insert** operation might cause a cascade of carries, but the time to perform all these carries is already paid for. We end up with a credit still on deposit for every exposed tree and only $O(1)$ time charged to the **insert** operation itself.

8.3 Lazy Melds

We can also perform **meld** operations in constant time with a slight modification of the data structure. Rather than using an array of pointers to trees, we use a doubly linked circular list. To **meld** two heaps, we just concatenate the two lists into one and update the **min** pointer, certainly an $O(1)$ operation. Then **insert**(h, i) is just **meld**$(h,$ **makeheap**$(i))$.

The problem now is that unlike before, we may have several trees of the same rank. This will not bother us until we need to do a **deletemin**. Since in a **deletemin** we will need $O(\log n)$ time anyway to find the minimum among the deleted vertex's children, we will take this opportunity to clean up the heap so that there will again be at most one tree of each rank. We create an array of empty pointers and go through the list of trees, inserting them one by one into the list, linking and carrying if necessary so as to have at most one tree of each rank. In the process, we search for the minimum.

We perform a constant amount of work for each tree in the list in addition to the linking. Thus if we start with m trees and do k links, then we spend $O(m + k)$ time in all. To pay for this, we have k saved credits from the links, plus an extra $\log n$ credits we can charge to the **deletemin** operation itself, so we will be in good shape provided $m + k$ is $O(k + \log n)$. But each link decreases the number of trees by one, so we end up with $m - k$ trees, and these trees all have distinct ranks, so there are at most $\log n$ of them; thus

$$\begin{aligned} m + k &= 2k + (m - k) \\ &\leq 2k + \log n \\ &= O(k + \log n) \ . \end{aligned}$$