Notatki z AiSD. Nr ?. 16 marca 2005

Drzewa samoorganizujące się

IIUWr. II rok informatyki. *Opracował: Krzysztof Loryś*

35 Wprowadzenie

Drzewa samoorganizujące się są kolejnym przykładem struktury danych opartej na binarnych drzewach przeszukiwań. Przymiotnik "samoorganizujące" oznacza, że drzewa te w trakcie wykonywania na nich operacji zmieniają swoją strukturę automatycznie, stosując pewną prostą heurystykę. W przeciwieństwie do drzew zbalansowanych (AVL, czerwono-czarnych) heurystyka ta nie korzysta z żadnych dodatkowych informacji pamiętanych w wierzchołkach. Druga istotna róznica polega na tym, że teraz pojedyncze operacje słownikowe mogą być kosztowne. Jak jednak pokażemy, zamortyzowany koszt ciągu operacji jest niski.

36 Operacje na drzewach samoorganizujących się

Oprócz operacji słownikowych ($find(i, S)$, $insert(i, S)$, $delete(i, S)$, odpowiednio odszukiwania, wstawiania i usuwania klucza i w (do, z) drzewie S) rozważymy realizację następujących operacji:

- $join(S_1, S_2)$ - połącz drzewa S_1 i S_2 w jedno drzewo (przy założeniu, że każdy klucz w drzewie S_1 jest nie większy od każdego klucza z drzewa S_2),
- $split(i, S)$ - rozdziel S na dwa drzewa S_1 i S_2 takie, że każdy klucz w S_1 jest nie większy od i, a każdy klucz w S_2 jest nie mniejszy od i.

37 Implementacja operacji

Podstawową idea drzew samoorganizujących się polega na tym, by wierzchołki drzewa zawierające klucz i (parametr operacji *insert, delete, find, split*) przesuwać serią rotacji do korzenia. Umiejętnie wykonywane rotacje będą powodować "spłaszczenie" drzewa.

Wygodnie jest nam wprowadzić operację *splay*, w terminach której wyrazimy wszystkie interesujące nas operacje.

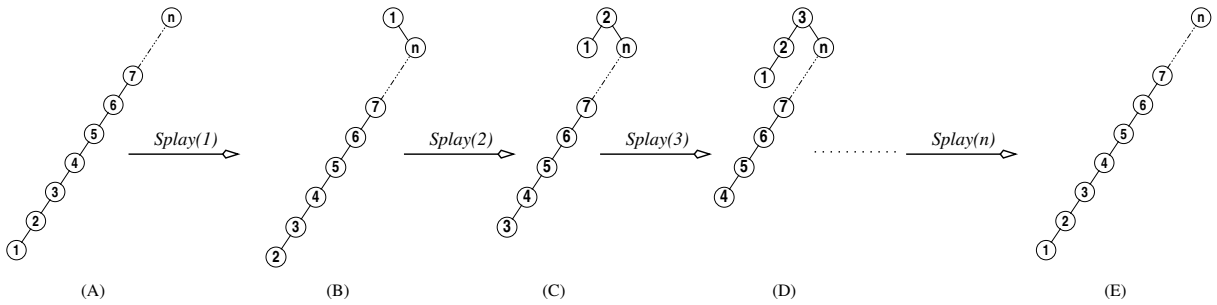
Definicja 17 *$splay(j, S)$ - przeorganizuj S tak, by jego korzeniem stał się wierzchołek zawierający k takie, że w S nie ma elementu leżącego między k i j.*

Tak więc jeśli j znajduje się w S to operacja $splay(j, S)$ przesunie j do korzenia. W przeciwnym razie w korzeniu znajdzie się $k = \min\{x \in S | x > j\}$ lub $k = \max\{x \in S | x < j\}$.

38 Implementacja Splay(x)

Splay łatwo jest zaimplementować przy pomocy rotacji. Jedną z moźliwości jest stosowanie rotacji do elementu x tak długo, aź znajdzie się on w korzeniu. Jak jednak pokazuje poniższy przykład, taka implementacja powoduje, źe niektóre ciągi operacji słownikowych byłyby wykonywane w czasie kwadratowym od długości ciągu.

Przykład 1



Drzewo (A) moźe powstać na skutek wykonania ciągu instrukcji: $insert(1)$, $insert(2)$, ..., $insert(n)$. Kolejne operacje: $splay(1), splay(2), \ldots splay(n-1)$ wykonują odpowiednio: $n-1$, $n-1$, $n-2$, $n-3, \ldots, 1$ rotacji. Po wykonaniu $Splay(n)$ otrzymujemy z powrotem drzewo (A). □

Wobec tego musimy zaproponować inny sposób implementacji. Rozwaźamy 3 przypadki:

(a) x ma ojca, ale nie ma dziadka → $rotate(x)$,

(b) x ma ojca $p(x)$ i ma dziadka; x i $p(x)$ są obydwaj lewymi bąd"x obydwaj prawymi synami swoich ojców → $rotate(y)$; $rotate(x)$,

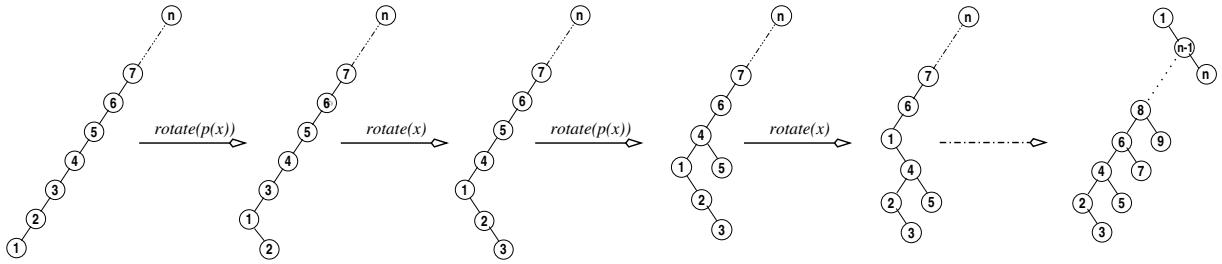
(c) x ma ojca $p(x)$ i ma dziadka; x jest lewym a $p(x)$ prawym synem, bąd"x na odwrót → $rotate(x)$; $rotate(x)$.

Przykład 2

39 Analiza

Stosujemy analizę zamortyzowaną. Każdy wierzchołek drzewa przechowuje pewien depozyt. Operacja wykonywana na drzewie moźe zwiększać depozyty, bad"x teź moźe być opłacana przez kwoty z depozytów.

Oznaczenia



$S(x)$ - poddrzewo o korzeniu w x,

$|S|$ - liczba wierzchołków w drzewie S,

$\mu(S) = \lfloor \log(|S|) \rfloor$,

$\mu(x) = \mu(S(x))$.

Będziemy utrzymywać następujący niezmiennik:

Wierzchołek x ma zawsze co najmniej $\mu(x)$ jednostek na swoim koncie.

Insert daje wierzchołkowi pewien początkoy depozyt.

Lemat 2 *Każda operacja* Splay(x,S) *wymaga nie więcej niż* $3(\mu(S) - \mu(x)) + 1$ *jednostek do wykonania operacji i zachowania niezmiennika kredytowego.*

Dowód: Niech y będzie ojcem x-a, a z - ojcem y-ka (o ile on istnieje). Niech ponadto μ oraz μ' oznaczają odpowiednio depozyty przed i po wykonaniu operacji *splay*.

(a) z nie istnieje. W tym przypadku wykonujemy pojedynczą rotację *rotate(x)*:

rysunek

Jak łatwo widać: $\mu'(x) = \mu(y)$, $\mu'(x) \geq \mu(x)$ oraz $\mu'(y) \leq \mu'(x)$.

Aby utrzymać niezmiennik musimy zapłacić:

$$\mu'(x) + \mu'(y) - \mu(x) - \mu(y) = \mu'(y) - \mu(x) \leq \mu'(x) - \mu(x) \leq 3(\mu'(x) - \mu(x)).$$

Mając do dyspozycji $3(\mu'(x) - \mu(x)) + 1$ jednostek jesteśmy w stanie utrzymać niezmiennik i pozostanie nam jeszcze jedna jednostka na opłacenie operacji niskiego poziomu związanych z wykonaniem *splay* (manipulacje wska"xnikami, porównania,...).

(b) Mamy

rysunek

Pokaźemy, źe *rotate*(y)*; rotate*(x) oraz utrzymanie niezmiennika kosztują nie więcej niź $3(\mu'(x) - \mu(x)$.

Aby utrzymać niezmiennik potrzebujemy:

$$(*) = \mu'(x) + \mu'(y) + \mu'(z) - \mu(x) - \mu(y) - \mu(z)$$

jednostek. Ponieważ

rysunek

mamy $\mu'(x) = \mu(z)$. Stąd

$$(*) = \mu'(y) + \mu'(z) - \mu(x) - \mu(y) = [\mu'(y) - \mu(x)] + [\mu'(z) - \mu(y)] \leq$$

$$\leq [\mu'(x) - \mu(x)] + [\mu'(x) - \mu(y)] \leq 2[\mu'(x) - \mu(x)].$$

Mając $3[\mu'(x)-\mu(x)]$ jednostek do dyspozycji, na opłacenie operacji niskiego poziomu wykonywanych przy tych dwóch rotacjach pozostaje nam $\mu'(x) - \mu(x)$ jednostek. Moźe się jednak okazać, źe $\mu'(x) = \mu(x)$. Pokaźemy, źe wówczas (*) jest ujemna i dlatego w tym przypadku niezmiennik mamy utrzymany bez ponoszenia kosztów a nawet moźemy uszczuplić

(c) Podobnie jak (b).

W trakcie operacji *Splay(x,S)* x zajmuje coraz wyźsze pozycje. Niech $S_1, S_2, \ldots, S_k$ będą drzewami zakorzenionymi w x w momencie gdy x zajmuje te pozycje. Wówczas całkowity koszt *Splay(x,S)* wynosi

$$3(\mu(S_1) - \mu(x)) + 3(\mu(S_2) - \mu(S_1)) + \ldots + 3(\mu(S_k) - \mu(S_{k-1}) + 1 =$$

$$3(\mu(S_k) - \mu(x)) + 1 = 3(\mu(S) - \mu(x)) + 1$$

□

Literatura

[1] D.Sleator, R.E.Tarjan, *Self-adjusting binary trees,* JACM, 32(1985), s. 652-686.

[2] R.E.Tarjan, *Data Structures and Network Algorithms,* SIAM, 1983.

Lecture 12 Splay Trees

A *splay tree* is a data structure invented by Sleator and Tarjan [94, 100] for maintaining a set of elements drawn from a totally ordered set and allowing membership testing, insertions, and deletions (among other operations) at an amortized cost of $O(\log n)$ per operation. The most interesting aspect of the structure is that, unlike balanced tree schemes such as 2-3 trees or AVL trees, it is not necessary to rebalance the tree explicitly after every operation—it happens automatically.

Splay trees are binary trees, but they need not be balanced. The height of a splay tree of n elements can be greater than $\log n$; indeed, it can be as great as $n-1$. Thus individual operations can take as much as linear time. However, as operations are performed on the tree, it tends to rebalance itself, and in the long run the amortized complexity works out to $O(\log n)$ per operation.

Data is represented at all nodes of a splay tree. The data values are distinct and drawn from a totally ordered set. The data items will always be maintained in inorder; that is, for any node x, the elements occupying the left subtree of x are all less than x, and those occupying the right subtree of x are all greater than x.

Splay trees support the following operations:

- **member**(i, S): determine whether element i is in splay tree S
- **insert**(i, S): insert i into S if it is not already there
- **delete**(i, S): delete i from S if it is there

- **join**(S, S'): join S and S' into a single splay tree, assuming that $x < y$ for all $x \in S$ and $y \in S'$
- **split**(i, S): split the splay tree S into two new splay trees S' and S'' such that $x \le i \le y$ for all $x \in S'$ and $y \in S''$.

All these operations are implemented in terms of a single basic operation, called a **splay**:

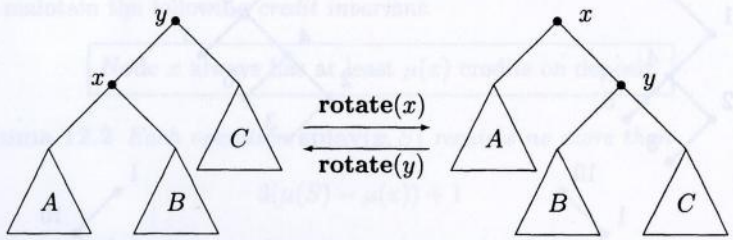
- **splay**(i, S): reorganize the splay tree S so that element i is at the root if $i \in S$, and otherwise the new root is either

$$\max\{k \in S \mid k < i\} \quad \text{or} \quad \min\{k \in S \mid k > i\} .$$

All of the operations mentioned above can be performed with a constant number of **splay**s in addition to a constant number of other low-level operations such as pointer manipulations and comparisons. For example, to do **join**(S, S'), first call **splay**$(+\infty, S)$ to reorganize S so that its largest element is at the root and all other elements are contained in the left subtree of the root; then make S' the right subtree. To do **delete**(i, S), call **splay**(i, S) to bring i to the root if it is there; then remove i and call **join** to merge the left and right subtrees.

12.1 Implementation of Splay

The **splay** operation can be implemented in terms of the even more elementary **rotate** operation. Given a binary tree S and a node x with parent y, the operation **rotate**(x) moves x up and y down and changes a few pointers, according to the following picture:



A very simple but important observation to make at this point is that the **rotate** operation preserves inorder numbering.

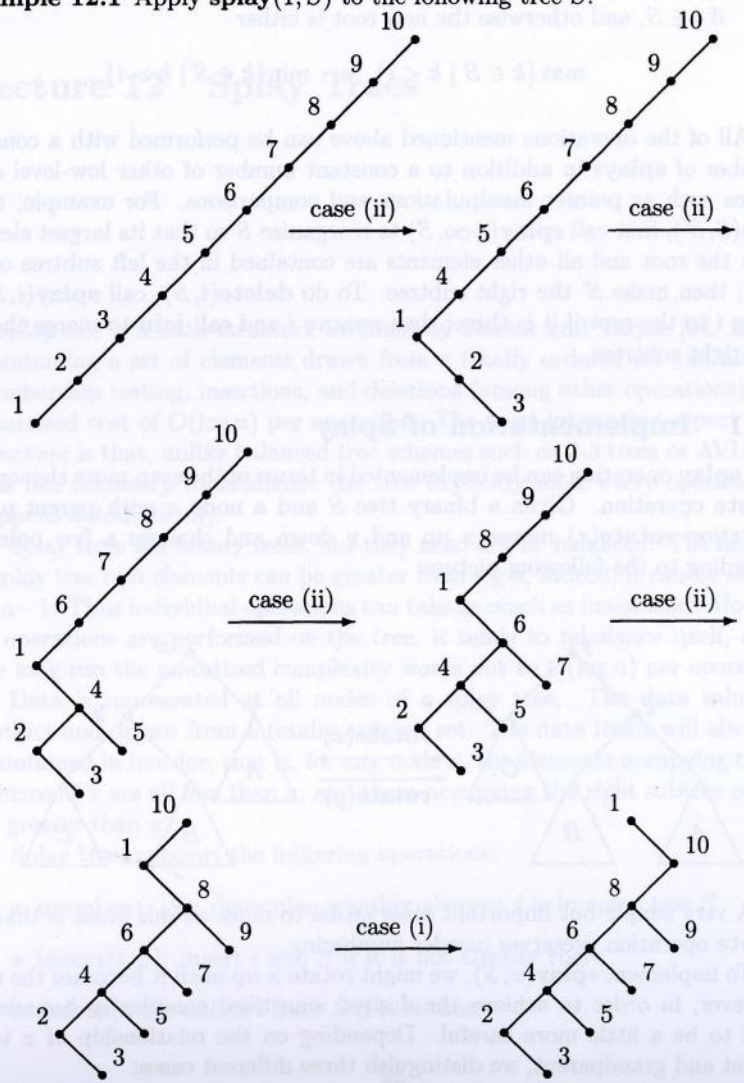
To implement **splay**(x, S), we might rotate x up until it becomes the root. However, in order to achieve the desired amortized complexity bounds, we need to be a little more careful. Depending on the relationship of x to its parent and grandparent, we distinguish three different cases:

(i) if x has a parent but no grandparent, we just **rotate**(x);

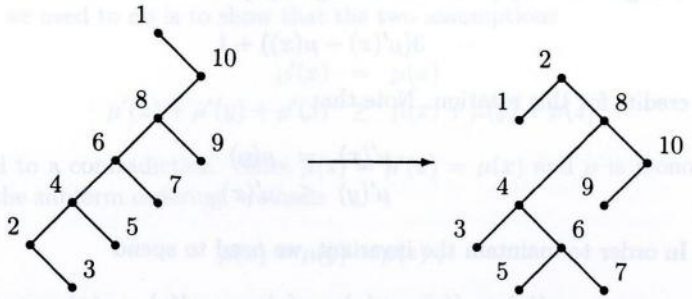
(ii) if x has a parent y and a grandparent, and if x and y are either both left children or both right children, we first **rotate**(y), then **rotate**(x);

(iii) if x has a parent y and a grandparent, and if one of x, y is a left child and the other is a right child, we first **rotate**(x) and then **rotate**(x) again.

Example 12.1 Apply **splay**$(1, S)$ to the following tree S:



Applying **splay** to node 2 of the resulting tree yields:



Note that the tree appears to become more balanced with each **splay**. □

12.2 Analysis

We will now show that the time required to perform m operations on a set of n elements is $O(m \log n)$. To do this, we use a credit accounting scheme similar to the one used in our analysis of Fibonacci heaps. Each node x of the splay tree has a savings account containing a certain number of *credits*. When x is created, some number of credits are charged to the **insert** operation that created x, and these credits are deposited to x's account. These credits can be used later to pay for restructuring operations.

For x a node of a splay tree, let $S(x)$ denote the subtree rooted at x. Let $|S|$ denote the number of nodes in tree S. Define

$$\begin{aligned} \mu(S) &= \lfloor (\log \mid S \mid) \rfloor \\ \mu(x) &= \mu(S(x)) \,. \end{aligned}$$

We maintain the following *credit invariant*:

> Node x always has at least $\mu(x)$ credits on deposit.

Lemma 12.2 *Each operation* **splay**(x, S) *requires no more than*

$$3(\mu(S) - \mu(x)) + 1$$

credits to perform the operation and maintain the credit invariant.

Proof. Let y be the parent of x and z be the parent of y, if it exists. Let μ and μ' be the values of μ before and after the **splay** operation, respectively. We consider three cases:

(i) *Node z does not exist.* This is the last rotation in the **splay**; we perform a single **rotate**(x). We are willing to pay no more than

$$3(\mu'(x) - \mu(x)) + 1$$

credits for this rotation. Note that

$$\begin{aligned} \mu'(x) &= \mu(y) \\ \mu'(y) &\leq \mu'(x) \,. \end{aligned}$$

In order to maintain the invariant, we need to spend

$$\begin{aligned} \mu'(x) + \mu'(y) - \mu(x) - \mu(y) &= \mu'(y) - \mu(x) \\ &\leq \mu'(x) - \mu(x) \\ &\leq 3(\mu'(x) - \mu(x)) \end{aligned}$$

credits. We are left with at least one credit left over to pay for the constant number of low-level operations such as pointer manipulations and comparisons.

(ii) *Node x is the left child of y and y is the left child of z (or both x and y are right children).* In this case we perform a **rotate**(y) followed by a **rotate**(x). We will show that it costs no more than $3(\mu'(x) - \mu(x))$ credits to perform these two **rotate** operations and maintain the credit invariant. Thus if a sequence of these are done to move x up the tree as in the example above, we will get a telescoping sum, so that the total amount spent will be no more than $3(\mu(S) - \mu(x)) + 1$ (the $+1$ comes from the last rotation as discussed in case (i)).
In order to maintain the invariant, we need

$$\mu'(x) + \mu'(y) + \mu'(z) - \mu(x) - \mu(y) - \mu(z) \tag{20}$$

extra credits. Since $\mu'(x) = \mu(z)$, we have

$$\begin{aligned} &\mu'(x) + \mu'(y) + \mu'(z) - \mu(x) - \mu(y) - \mu(z) \\ &= \mu'(y) + \mu'(z) - \mu(x) - \mu(y) \\ &= (\mu'(y) - \mu(x)) + (\mu'(z) - \mu(y)) \\ &\leq (\mu'(x) - \mu(x)) + (\mu'(x) - \mu(x)) \\ &= 2(\mu'(x) - \mu(x)) \,. \end{aligned}$$

We can afford to pay for this and have $\mu'(x) - \mu(x)$ credits left over to pay for the constant number of low-level operations needed to perform these two rotations. Unfortunately, it may turn out that $\mu'(x) = \mu(x)$, in which case we have nothing left over. We show that in this case the quantity (20) is in fact strictly negative, thus the invariant is maintained

for free and we can even afford to spend one of our saved credits to pay for the low-level operations.
All we need to do is to show that the two assumptions

$$\begin{aligned} \mu'(x) &= \mu(x) \\ \mu'(x) + \mu'(y) + \mu'(z) &\geq \mu(x) + \mu(y) + \mu(z) \end{aligned}$$

lead to a contradiction. Since $\mu(z) = \mu'(x) = \mu(x)$ and μ is monotone in the subterm ordering, we have

$$\mu(x) = \mu(y) = \mu(z) \,,$$

therefore

$$\begin{aligned} \mu'(x) + \mu'(y) + \mu'(z) &\geq 3\mu(z) \\ &= 3\mu'(x) \\ \mu'(y) + \mu'(z) &\geq 2\mu'(x) \,. \end{aligned}$$

Because μ' is monotone in the subterm ordering,

$$\begin{aligned} \mu'(y) &\leq \mu'(x) \\ \mu'(z) &\leq \mu'(x) \,. \end{aligned}$$

It follows that

$$\mu'(x) = \mu'(y) = \mu'(z) \,,$$

and since $\mu(z) = \mu'(x)$, we have

$$\mu(x) = \mu(y) = \mu(z) = \mu'(x) = \mu'(y) = \mu'(z) \,. \tag{21}$$

Substituting in for the definition of μ and μ' will quickly show that this situation is untenable. If a is the size of the subtree rooted at x before the operation and b is the size of the subtree rooted at z after the operation, then (21) implies

$$\lfloor \log a \rfloor = \lfloor \log(a + b + 1) \rfloor = \lfloor \log b \rfloor \,. \tag{22}$$

Assuming without loss of generality that $a \leq b$,

$$\begin{aligned} \lfloor \log(a + b + 1) \rfloor &\geq \lfloor \log 2a \rfloor \\ &= 1 + \lfloor \log a \rfloor \\ &> \lfloor \log a \rfloor \,. \end{aligned}$$

This contradicts (22).

(iii) *Node x is a left child of y and y is a right child of z, or vice versa.* Here we do **rotate**(x) followed by **rotate**(x) again, and we are willing to pay no more than $3(\mu'(x) - \mu(x))$ credits for these two rotations. As in the previous case, we need

$$\mu'(x) + \mu'(y) + \mu'(z) - \mu(x) - \mu(y) - \mu(z)$$

credits to maintain the invariant, and this quantity is at most $2(\mu'(x) - \mu(x))$. This leaves at least $\mu'(x) - \mu(x)$ left over to pay for the low-level operations, which suffices unless $\mu'(x) = \mu(x)$. As in case (ii), we prove by contradiction that in this case

$$\mu'(x) + \mu'(y) + \mu'(z) \quad < \quad \mu(x) + \mu(y) + \mu(z) \,,$$

thus the credit invariant is maintained for free and we have at least one extra credit to spend on the low-level operations.

□

Theorem 12.3 *A sequence of m operations involving n* **insert***s takes time $O(m \log n)$.*

Proof. First we note that the maximum value of $\mu(x)$ is $\lfloor \log n \rfloor$. It follows from Lemma 12.2 that at most $3\lfloor \log n \rfloor + 1$ credits are needed for each **splay** operation. Since each of the operations **member**, **insert**, **delete**, **split**, and **join** can be performed using a constant number of **splay**s and a constant number of low-level operations, each of these operations costs $O(\log n)$. Inserting a new item requires at most $O(\log n)$ credits to be deposited to its account for future use; we charge these credits to the **insert** operation. Hence each operation requires at most $O(\log n)$ credits. It follows that the total time required for a sequence of m such operations is $O(m \log n)$. □