ZAAWANSOWANE PROGRAMOWANIE KOMPUTEROWE WNE (2021)

KRZYSZTOF ZIEMIAŃSKI

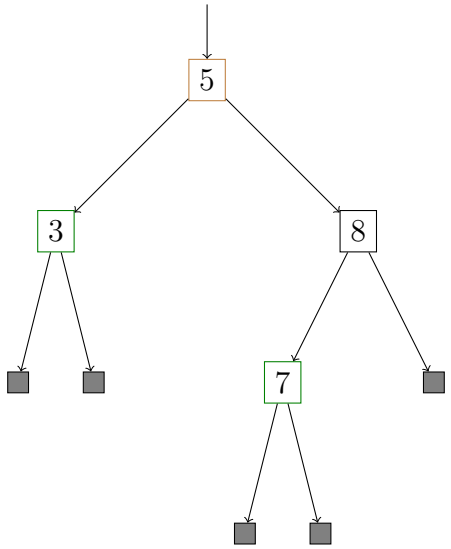
9. Przechowywanie danych - drzewa

W poprzedniej części porównywaliśmy efektywność przechowywania danych przy użyciu różnych struktur. Zaletą list jest możliwość szybkiego usuwania i dodawania danych — nie wymaga ono przepisywania elementów. Zaletą tablic uporządkowanych jest możliwość szybkiego (tj. w czasie logarytmicznym) znajdowania elementów. W tym odcinku poznamy sposób na efektywne przechowywanie danych łączący zalety tablic i list.

9.1. **Drzewa.** Podobnie jak listy, drzewa składają się z elementów. Każdy element zawiera:

- wartość,
- wskaźnik do lewego poddrzewa,
- wskaźnik do prawego poddrzewa.

Przykładowe drzewo wygląda tak:



Dostęp do elementów drzewa uzyskujemy przez wskaźnik do najwyższego elementu, zwanego *korzeniem* (brązowy). Drzewo może być puste; element drzewa taki, że jego oba poddrzewa są puste nazywa się *liśćmi* (zielone). Oczywiście może się zdarzyć, że jedno z poddrzew jest puste, a drugie nie (tak jak element z wartością 8 na przykładzie).

Drzewa są używane do różnych celów; my będziemy ich używać do przechowywania danych. Dlatego też będziemy dbać o to, żeby w naszych drzewach była zachowana następująca zasada:

Wszystkie wartości elementów z lewego poddrzewa są mniejsze, a elementów z prawego poddrzewa większe niż wartość w korzeniu.

Będziemy też dopuszczać tylko jeden element z daną wartością.

9.2. **Implementacja.** Elementy drzewa będą zdefiniowane jako struktury

```
class ElDrzewa {
public:
   ElDrzewa(int _dane, ElDrzewa* _lewy=nullptr, ElDrzewa* _prawy=nullptr)
      : dane(_dane), lewy(_lewy), prawy(_prawy) {}
   int dane;
   ElDrzewa* lewy;
   ElDrzewa* prawy;
};
```

Typ `Drzewo` definujemy jako wskaźnik do `ElDrzewa`. Poniżej przegląd operacji na drzewach:

9.3. **Dodawanie elementów.** Dodawanie elementu z wartością `w` przebiega następująco:

(1) Jeśli drzewo jest puste, zastępujemy je jednoelementowym drzewem w wartością `w`.

(2) Jeśli wartość w korzeniu wynosi `w`, nic nie robimy (element już jest).

(3) Jeśli wartość w korzeniu jest mniejsza niż `w`, dodajemy element z wartością `w` do prawego poddrzewa (rekurencyjnie).

(4) Jeśli wartość w korzeniu jest większa niż `w`, dodajemy element z wartością `w` do lewego poddrzewa.

Ostatecznie jeśli elementu nie ma w drzewie, w pewnym miejscu dodamy nowy liść z odpowiednią wartością. Jest tylko jedno miejsce, które nie psuje porządku w drzewie.

Poniżej funkcja, która dodaje element.

```
void dodaj(Drzewo& d, int wartosc) {
   if(d==nullptr) {
      d = new ElDrzewa(wartosc);
   }
   else {
      if(d->dane < wartosc)
         dodaj(d->prawy, wartosc);
      if(d->dane > wartosc)
         dodaj(d->lewy, wartosc);
      }
   }
}
```

Złożoność tej operacji jest nie większa niż wysokość drzewa, tj. długość ciągu kolejnych poddrzew. W drzewie z przykładu wysokość wynosi 3 (bo najdłuższy ciąg elementów to $5 \to 8 \to 7$).

9.4. **Wyszukiwanie elementów.** Wyszukiwanie elementów bardzo przypomina wyszukiwanie binarne; korzeń jest traktowany jako "środkowy" element. Żeby znaleźć element o wartości `w`, postępujemy tak:

(1) Jeśli drzewo jest puste, nie ma takiego elementu.

(2) Jeśli wartość w korzeniu wynosi `w`, właśnie go znaleźliśmy.

(3) Jeśli wartość w korzeniu jest mniejsza niż `w`, szukamy w prawym poddrzewie.

(4) Jeśli wartość w korzeniu jest większa niż `w`, szukamy w lewym poddrzewie.

Funkcja ta jest pozostawiona jako ćwiczenie. Złożoność tej operacji jest, podobnie jak w przypadku dodawania, proporcjonalna do wysokości drzewa.

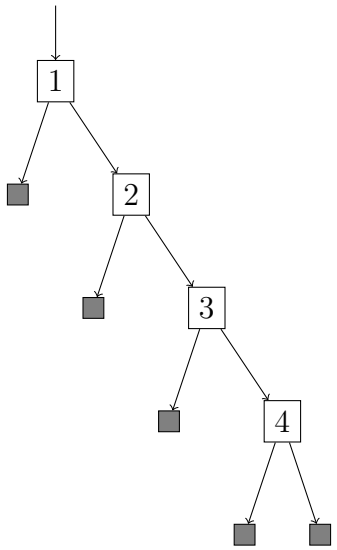
9.5. **Usuwanie elementów.** Żeby usunąć element trzeba go najpierw znaleźć — robimy to jak powyżej. Jeśli odnaleziony element jest liściem, usunąć go łatwo; gorzej, jeśli element, który powinniśmy usunąć ma poddrzewa. Problem ten rozwiązujemy następująco:

(1) Jeśli lewe poddrzewo jest niepuste, usuwamy z niego *największy* element i wpisujemy jego wartość do elementu, który mamy usunąć.

(2) Jeśli lewe poddrzewo jest puste, a prawe niepuste, usuwamy z prawego poddrzewa *najmniejszy* element i nadpisujemy jego wartością wartość, którą chcemy usunąć.

W ten sposób zachowujemy uporządkowanie drzewa. Złożoność jest ponownie proporcjonalna do wysokości drzewa.

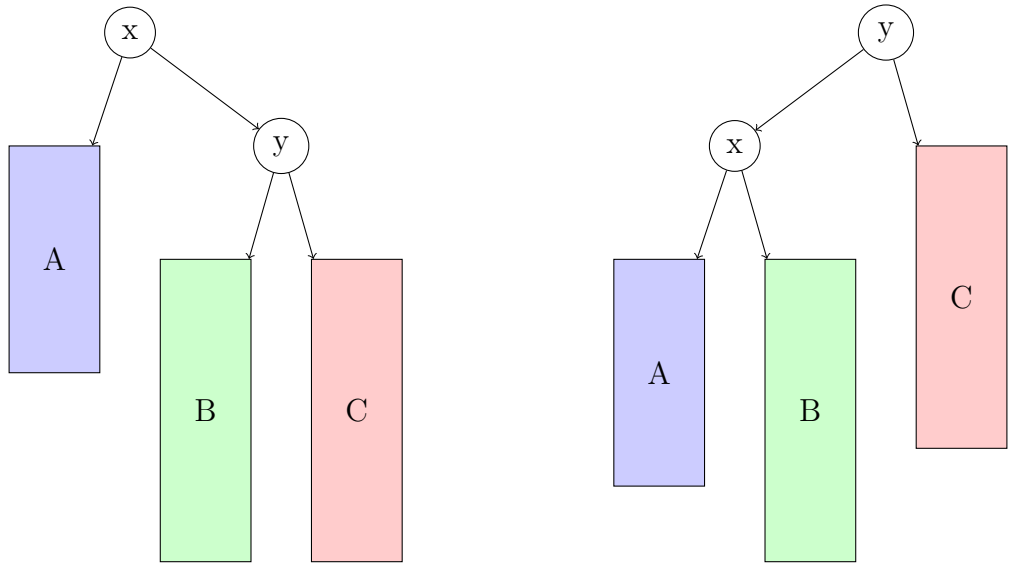
9.6. **Zrównoważenie drzewa.** Złożoność wszystkich operacje na danych jest proporcjonalna do wysokości drzewa. Czy gwarantuje to, że jest ona istotnie mniejsza niż liczba wszystkich elementów drzewa? Niestety, tak nie jest. Łatwo zobaczyć, że jeśli dodajemy do drzewa wartości w kolejności rosnącej (lub malejącej) otrzymujemy coś takiego



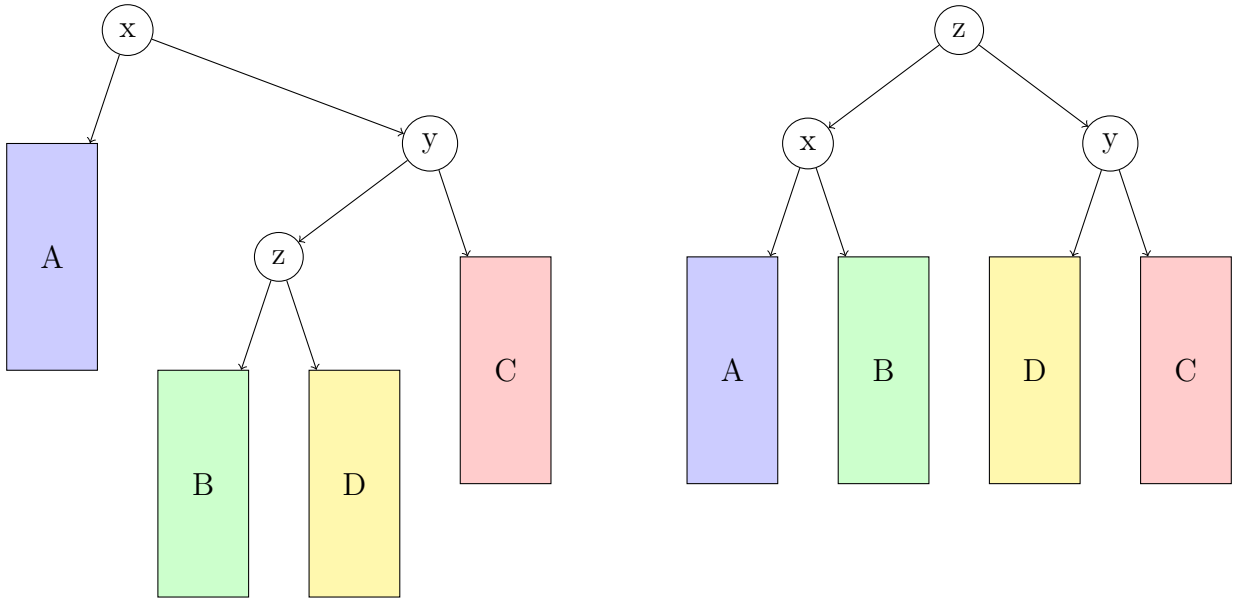
a więc wysokość drzewa jest równa liczbie jego elementów (a samo drzewo przypomina listę jednokierunkową). Musimy więc zadbać o to, aby drzewo było "zrównoważone", tj. żeby lewe

i prawe poddrzewa były podobnej wielkości. Wtedy wysokość drzewa będzie nie większa niż logarytm z liczby elementów (z dokładnością do stałej). Drzewo będziemy uważać za zrównoważone jeśli wysokości lewego i prawego poddrzewa każdego elementu różnią się nie więcej niż o 1.

Załóżmy, że po wykonaniu pewnej operacji drzewo staje się niezrównoważone, tj. wysokość prawego poddrzewa staje się większa o 2 od wysokości lewego poddrzewa (lub na odwrót). Wtedy możemy je zrównoważyć wykonując jedną z poniższych zamian:



Jeśli B nie jest wyższe niż C, dostajemy drzewo zrównoważone. Jeśli B jest wyższe od C, dokonujemy zamiany jak poniżej:



W ten sposób możemy (w czasie nie większym niż wysokość drzewa) zagwarantować, że drzewo pozostaje zrównoważone. Zrównoważone drzewa uporządkowane umożliwiają na przechowywanie danych tak, aby wszystkie operacje były wykonywane w czasie nie większym niż

logarytmiczny ze względu na liczbę elementów. Efektywne równoważenie drzew wymaga pamiętania wysokości każdego poddrzewa.

9.7. **Zadania.**

(1) Napisać funkcję

```
int ile(Drzewo& d);
```

która zwraca liczbę elementów w drzewie.

(2) Napisać funkcję

```
int suma(Drzewo& d);
```

która zwraca sumę wartości zapisanych w drzewie.

(3) Napisać funkcję

```
int wysokosc(Drzewo& d);
```

która zwraca wysokość drzewa.

(4) Napisać funkcję

```
bool czyZawiera(Drzewo& d);
```

która zwraca `true` tylko wtedy, gdy dane drzewo zawiera podaną wartość w jednym z elementów. Zakładamy, że drzewo jest uporządkowane.

(5) Napisać funkcję

```
void lustro(Drzewo& d);
```

która zamienia porządek w drzewie z rosnącego na malejący (lub na odwrót).

(6) (*) Dodać do klasy `ElDrzewa` pole `wysokosc`, które będzie przechowywać wysokość drzewa (tj. 1 jeśli dany element jest liściem, 2 jeśli jego poddrzewa to liście, itd.) Zadbać o to, aby wartość tego pola ustawiała się automatycznie po każdej zmianie w drzewie.

(7) (**) Zmienić funkcje dodające i usuwające elementy tak, aby drzewo po każdej operacji było zrównoważone.