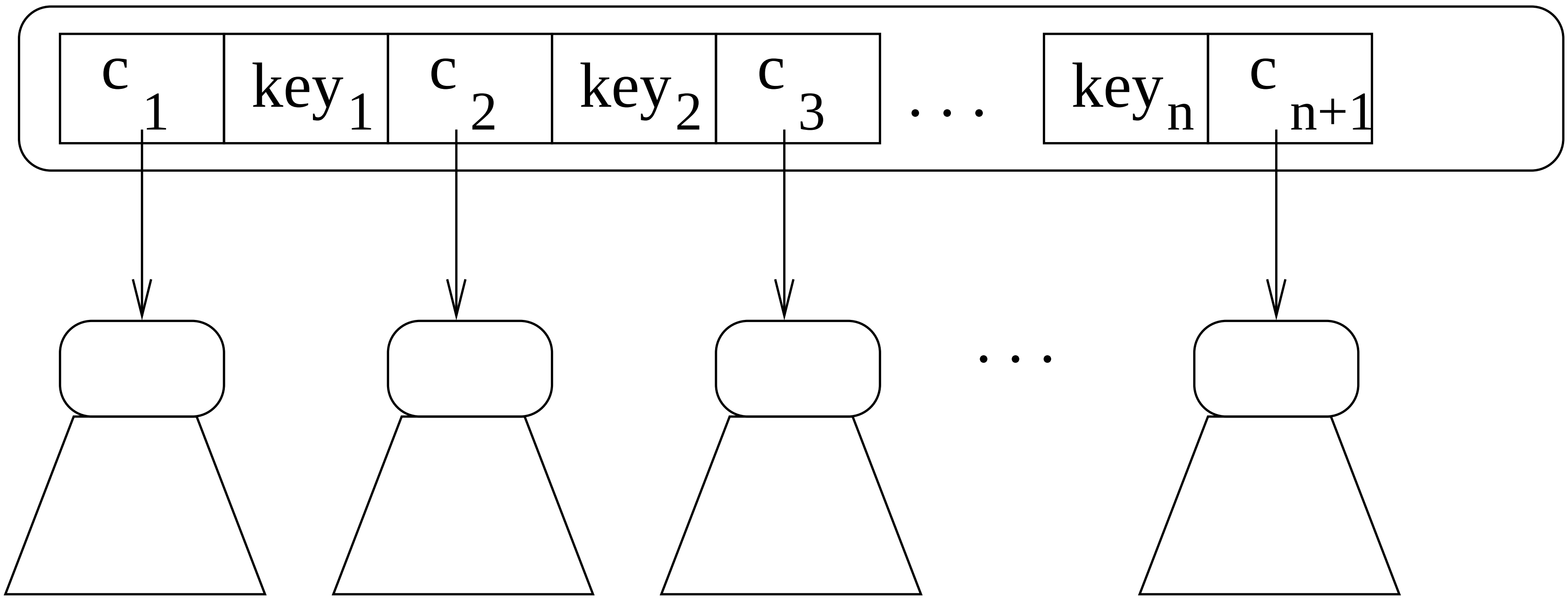
B-drzewo

Zakładamy, że węzły przechowywane jako rekordy na dysku. Każdy węzeł może zawierać dużo (np. 1000) kluczy. Koszt odczytu i zapisu rekordu na dysku dominuje nad operacjami wewnątrz komputera.



B-drzewo

B-Drzewo T, *root*$[T]$ – korzeń. Parametr $t \geq 2$ – *minimalny stopień*.

Pola węzła x:

- *n*$[x]$ – liczba kluczy w x
- *key*$_1[x] \leq \ldots \leq$*key*$_{n[x]}[x]$ – klucze
- *leaf*$[x]$ – pole logiczne: “czy x – liść?”
- *c*$_1[x] \ldots$ *c*$_{n[x]+1}[x]$ – wskaźniki do synów x

Założenia dot. operacji na B-drzewie:

- Korzeń – zawsze w pamięci (nie trzeba wczytywać), ale trzeba zapisywać – kiedykolwiek korzeń się zmienia.
- Każdy węzeł przekazywany jako parametr, musi być wcześniej wczytany z dysku.

B-drzewo

Własności:

- Jeśli, dla $i = 1 \ldots n[x] + 1$, k_i – dowolny klucz z poddrzewa o korzeniu $c_i[x]$, to:
  $k_1 \leq \mathit{key}_1[x] \leq k_2 \leq \mathit{key}_2[x] \leq \ldots \leq \mathit{key}_{n[x]}[x] \leq k_{n[x]+1}[x]$
- Wszystkie liście – na tej samej głębokości
- Dla $x \neq \mathit{root}[T]$, $n[x] \geq t - 1$.
- Dla każdego x, $n[x] \leq 2t - 1$. (x – *pełny*, jeśli $n[x] = 2t - 1$.)

Wysokość B-drzewa

Tw. Dla każdego B-drzewa T o $n \geq 1$ kluczach, wysokości h i minimalnym stopniu t mamy:

$$h \leq \log_t \frac{n+1}{2}$$

D-d. Jeśli T ma wysokość h, to – najmniej węzłów, gdy *n*[*root*$[T]$]$= 1$ oraz, $\forall x \neq$*root*$[T]$, *n*$[x] = t - 1$.
Wtedy: 2 węzły na głębokości 1, $2t$ węzłów na gł. 2, $2t^2$ węzłów na gł. $3, \ldots, 2t^{h-1}$ węzłów na gł. h. Stąd liczba kluczy:

$$n \geq 1 + (t-1)\sum_{i=1}^{h} 2t^{i-1} = 1 + 2(t-1)\left(\frac{t^h-1}{t-1}\right) = 2t^h - 1 \;\square$$

B-Tree-Search

```
B-Tree-Search(x, k)
i ← 1
while i ≤ n[x] and k > key_i[x] do
    i ← i + 1
if i ≤ n[x] and k = key_i[x]
    then return(x, i)
if leaf[x] then
    return NIL
else
    Disk-Read(c_i[x])
    return B-Tree-Search(c_i[x], k)
```

Liczba dostępów do dysku: $O(h) = O(\log_t n)$.

Czas CPU: $O(th)$. (w każdym węźle liczba iteracji $O(t)$.)

(Uwaga: można zastosować wyszukiwanie binarne klucza w węźle: wtedy czas CPU w każdym węźle $O(\lg t)$).

B-Tree-Create

Tworzenie pustego nowego drzewa:

`B-Tree-Create(`T`)`

`1` $x \leftarrow$ `Allocate-Node()`

`2` *leaf*$[x] \leftarrow TRUE$

`3` *n*$[x] \leftarrow 0$

`4 Disk-Write(`x`)`

`5` *root*$[T] \leftarrow x$

Dostępy do dysku: $O(1)$. Czas: $O(1)$.

B-Tree-Split-Child

Rozbijanie pełnego węzła y o niepełnym ojcu x takiego, że $y =$c$_i[x]$. (Z $2t-1$ kluczy y powstaną 2 węzły po $t-1$ kluczy, a środkowy klucz je rozdzieli w x.)

`B-Tree-Split-Child(`x`,`i`,`y`)`

$\triangleright$ tworzymy nowy węzeł z

$z \leftarrow$`Allocate-Node()`

leaf$[z] \leftarrow$*leaf*$[y]$

n$[z] \leftarrow t-1$

$\triangleright$ przenosimy część kluczy i dzieci z y do z

for $j \leftarrow 1$ **to** $t-1$ **do**

    key$_j[z] \leftarrow$*key*$_{j+t}[y]$

if not *leaf*$[y]$ **then**

    for $j \leftarrow 1$ **to** t

        $c_j[z] \leftarrow c_{j+t}[y]$

n$[y] \leftarrow t-1$

...

B-Tree-Split-Child

...

▷ wstawiamy do x wskaźnik na z

▷ i klucz rozdzielający y od z

for $j \leftarrow n[x]+1$ **downto** $i+1$ **do**

    $c_{j+1}[x] \leftarrow c_j[x]$

$c_{i+1}[x] \leftarrow z$

for $j \leftarrow n[x]$ **downto** i **do**

    $key_{j+1}[x] \leftarrow key_j[x]$

$key_i[x] \leftarrow key_t[y]$

$n[x] \leftarrow n[x]+1$

▷ zapisujemy zmiany na dysk

Disk-Write(y)

Disk-Write(z)

Disk-Write(x)

Dostępy do dysku: $O(1)$, czas CPU: $\Theta(t)$.

B-Tree-Insert

B-Tree-Insert(T,k)

$r \leftarrow root[T]$

if $n[r] = 2t - 1$ then

  ▷ wzrasta wysokość T

  $s \leftarrow$ Allocate-Node()

  $root[T] \leftarrow s$

  $leaf[s] \leftarrow FALSE$

  $n[s] \leftarrow 0$

  $c_1[s] \leftarrow r$

  B-Tree-Split-Child(s,1,r)

  B-Tree-Insert-Nonfull(s,k)

else B-Tree-Insert-Nonfull(r,k)

B-Tree-Insert-Nonfull

Wstawianie k do poddrzewa o niepełnym korzeniu x.

B-Tree-Insert-Nonfull(x, k)

$i \leftarrow n[x]$
if $leaf[x]$ then $\triangleright$ x - liść
    while $i \geq 1$ and $k < key_i[x]$ do
        $key_{i+1}[x] \leftarrow key_i[x]$
        $i \leftarrow i - 1$
    $key_{i+1}[x] \leftarrow k$
    $n[x] \leftarrow n[x] + 1$
    Disk-Write(x)
...

B-Tree-Insert-Nonfull

...
else ▷ x - nie liść
    while $i \geq 1$ and $k <$ *key*$_i[x]$ do
        $i \leftarrow i - 1$
    $i \leftarrow i + 1$
    Disk-Read($c_i[x]$)
    if $n[c_i[x]] = 2t - 1$ then
        B-Tree-Split-Child(x, i, $c_i[x]$)
        if $k >$ *key*$_i[x]$ then
            $i \leftarrow i + 1$
    B-Tree-Insert-Nonfull($c_i[x]$, k)

Dostępy do dysku: $O(h) = O(\log_t n)$, czas CPU: $O(th)$.
(Również dotyczy `B-Tree-Insert`.)

Usuwanie klucza z B-drzewa

Usuwanie klucza klucza k z poddrzewa o korzeniu x: `B-Tree-Delete(`x`,`k`)`.
Zachowujemy niezmiennik: Jeśli `B-Tree-Delete(`x`,`k`)` jest wywoływane rekurencyjnie dla $x \neq$*root*$[T]$, to *n*$[x] \geq t$.
Dzięki temu x zawsze może “oddać 1 klucz w dół”.

`B-tree-Delete(`x`,`k`)`

$\triangleright$ $x =$*root*$[T]$ `lub` x `ma` $\geq t$ `kluczy`

Przypadek 1: x *– liść:*

`Jeśli` k `jest w` x `to usuń` k `z` x

$\triangleright$ `Pozostały przypadki:`

$\triangleright$ *Przypadek 2:* k *jest w* x *i* x *– nieliść*`, oraz`

$\triangleright$ *Przypadek 3:* k *nie ma w* x *i* x *– nieliść*

`...`

B-tree-Delete – przypadek 2

...

Przypadek 2: k jest w x i x – nieliść:

a) Jeśli y - syn x poprzedzający k - ma $\geq t$ kluczy, to znajdź w poddrzewie y poprzednika k: k'. Wywołaj B-tree-Delete(y,k') i zastąp k przez k' w węźle x.

B-tree-Delete – przypadek 2

...

Przypadek 2: k *jest w* x *i* x *– nieliść:*

a) Jeśli y - syn x poprzedzający k - ma $\geq t$ kluczy, to znajdź w poddrzewie y poprzednika k: k'. Wywołaj B-tree-Delete(y,k') i zastąp k przez k' w węźle x.

b) W p.p. jeśli z - syn x następujący po k - ma $\geq t$ kluczy, to - symetrycznie do a).

B-tree-Delete – przypadek 2

...

Przypadek 2: k jest w x i x – nieliść:

a) Jeśli y - syn x poprzedzający k - ma $\geq t$ kluczy, to znajdź w poddrzewie y poprzednika k: k'. Wywołaj B-tree-Delete(y,k') i zastąp k przez k' w węźle x.

b) W p.p. jeśli z - syn x następujący po k - ma $\geq t$ kluczy, to - symetrycznie do a).

c) W p.p. (tj. y oraz z mają po $t-1$ kluczy):

- "doklej" do y klucz k i zawartość z
- usuń klucz k i wskaźnik do z z węzła x
- zwolnij pamięć przydzieloną z
- wywołaj B-tree-Delete(y,k)

...

B-tree-Delete – przypadek 3

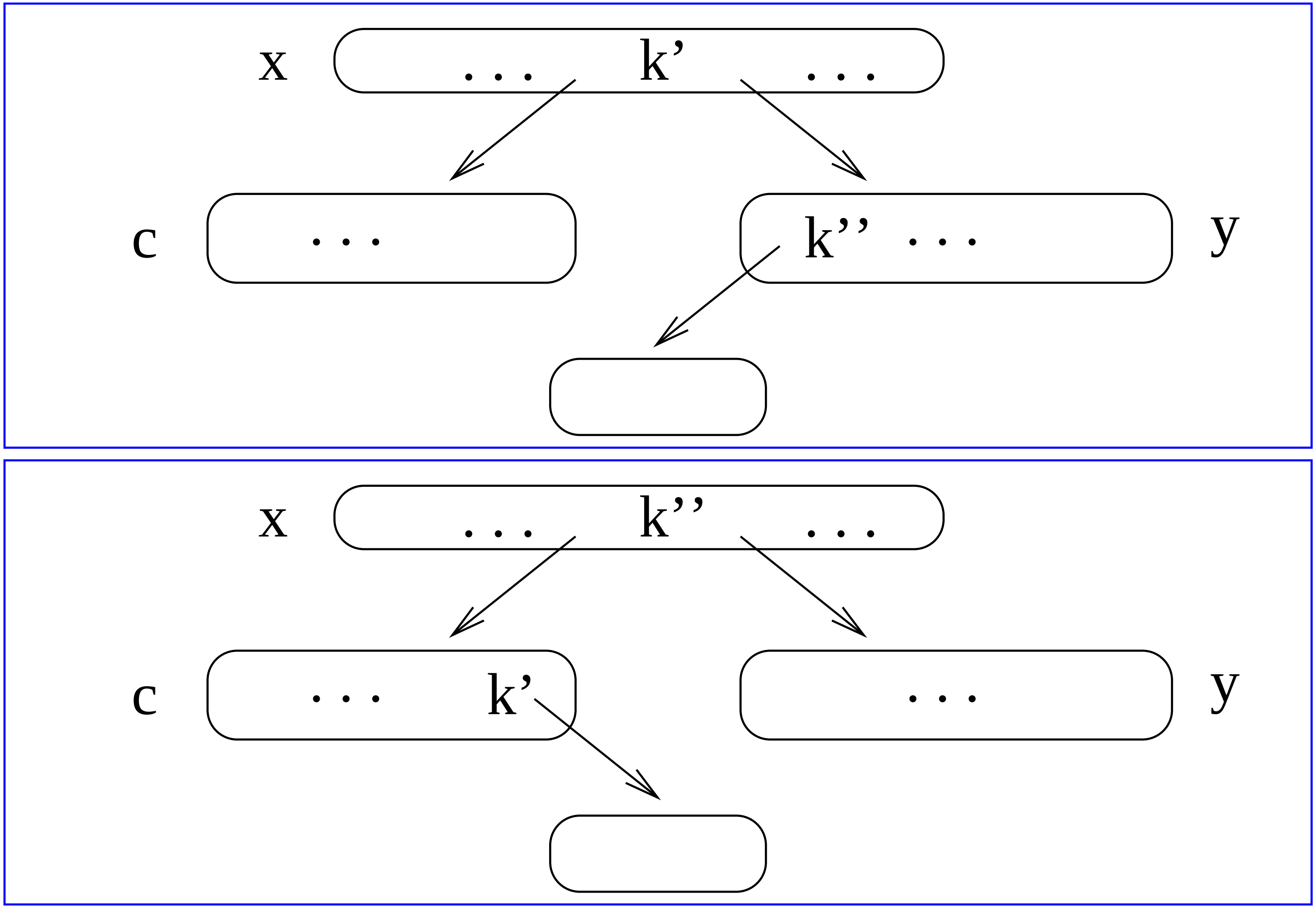
Przypadek 3: k *nie ma w* x *i* x *– nieliść:*

Wyznacz $c = c_i[x]$ - korzeń poddrzewa, w którym należy szukać k. Jeśli c ma $t-1$ kluczy, to:

a) Jeśli y - jeden sąsiednich z braci c ma $\geq t$ kluczy, to:

- przenieś z x do c klucz k' oddzielający c od y
- przenieś w miejsce k' odpowiedni (pierwszy albo ostatni) klucz z y do x jako nowy klucz oddzielający c od y
- przenieś odpowiedniego (pierwszego albo ostatniego) syna y do c jako nowego (ostatniego albo pierwszego) syna c

B-tree-Delete – przypadek 3a



B-tree-Delete – przypadek 3 (c.d.)

b) Jeśli wszyscy sąsiedni bracia c mają po $t-1$ kluczy, to doklej c do y - jednego z sąsiednich braci c - (przenosząc przy tym klucz oddzielający c od y z x do c)

▷ koniec przypadków 3a) i 3b)

▷ - teraz c ma $\geq t$ kluczy

Wywołaj B-Tree-Delete(c,k)

▷ koniec przypadku 3

Jeśli teraz korzeń jest pusty (ma zero kluczy) to go usuwamy a nowym korzeniem zostaje jego (jedyny) syn (jeśli istnieje) lub drzewo staje się puste.

▷ zmniejszenie wysokośći

Analiza B-Tree-Delete

W przypadkach 2a i 2b można wyznaczyć k' i usunąć go z poddrzewa w jednym przejściu w dół drzewa:
W przypadku 2a (odp. 2b) k' jest maksymalnym (odp. minimalnym) elementem w poddrzewie y (odp. z).
Do wykorzystania w przypadku 2a) można np. zaimplementować `B-Tree-Delete-Max(`y`)`, która działa jak `B-Tree-Delete` mimo, że nie zna wartości usuwanego klucza (ale wie gdzie go szukać) i ponadto zwraca wartość usuniętego klucza. W `B-Tree-Delete-Max` nie zachodzą przypadki 2a i 2b, bo maksymalny klucz jest w liściu (skrajnym prawym).
(Podobnie – `B-Tree-Delete-Min` – dla przypadku 2b))
Stąd złożoność `B-Tree-Delete`:
Dostępy do dysku: $O(h)$, czas CPU: $O(th)$.