Алгоритми та складність

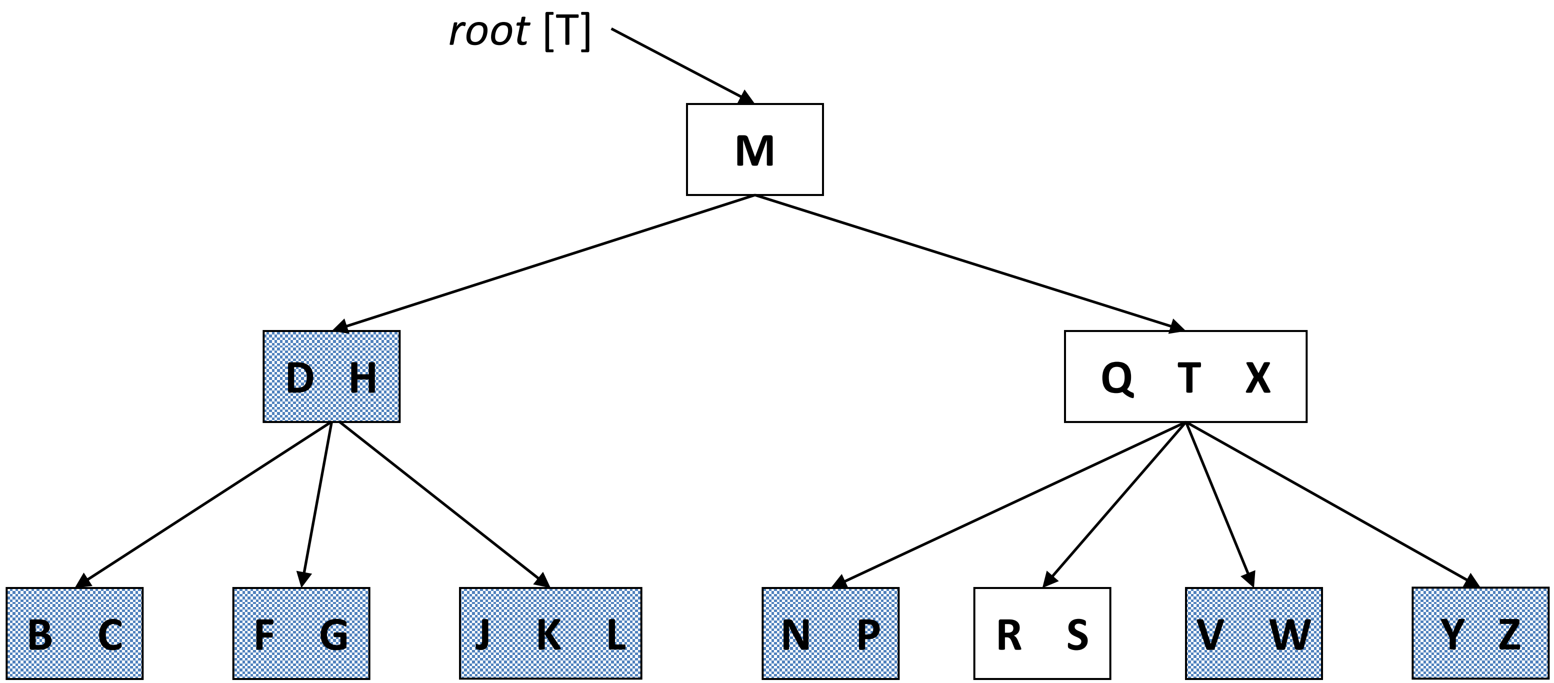
II семестр

Лекція 4

B-дерева

- Робота з великими об’ємами даних, які не поміщаються в оперативну пам’ять і зберігаються на диску (наприклад, СУБД, файлові системи).
- B-дерева – узагальнення бінарних дерев пошуку.
- Висока степінь розгалуження – вузли можуть мати до тисяч потомків.
- Якщо внутрішній вузол містить $n[x]$ ключів, то він має ($n[x]+1$) синів.
- Ключі у вузлі x є роздільниками діапазону ключів на ($n[x]+1$) піддіапазонів.
- При пошуку переходимо до сина з потрібним діапазоном.

B-дерева



B-дерево з латинськими приголосними в якості ключів

Структури даних у вторинній пам’яті

Пам’ять комп’ютера

Оперативна

(primary, main memory)

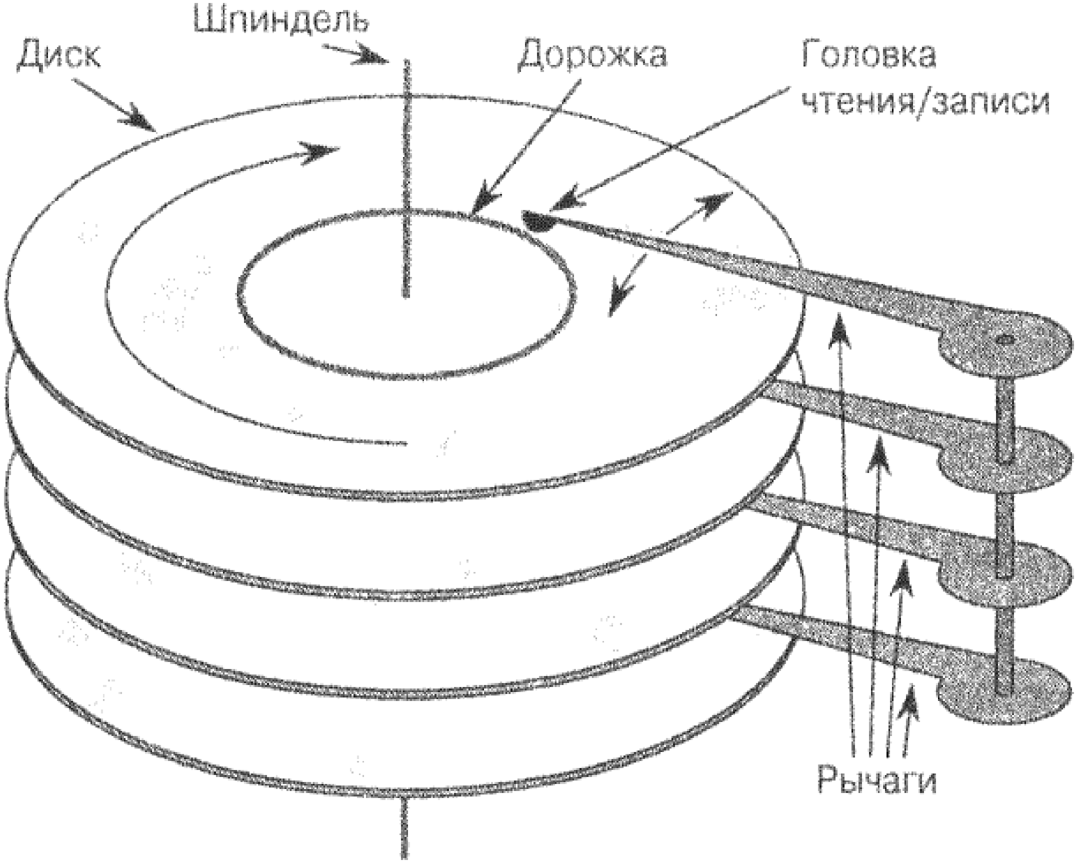
- висока швидкодія
- висока ціна

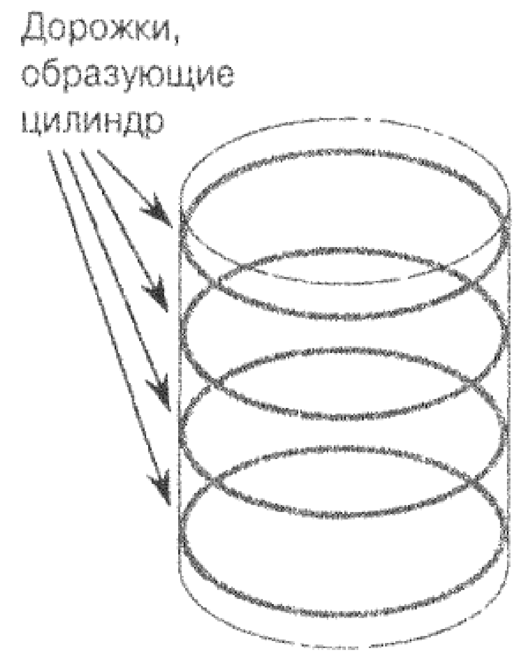
Вторинна

(secondary storage)

- низька швидкодія
- низька ціна

Структури даних у вторинній пам’яті





Вигляд дискового накопичувача

Структури даних у вторинній пам’яті

- Час доступу до оперативної та вторинної пам’яті відрізняється в тисячі разів за рахунок необхідності механічних переміщень:
  - обертання дисків,
  - переміщення голівок по радіусу.
- Накопичувач дозволяє одночасно звертатися до декількох елементів, що на ньому зберігаються.
- Інформація розбивається на сторінки однакового розміру (2^{11}-2^{14} байт), що зберігаються послідовно в межах одного циліндру.
- Відповідно кожна операція читання або запису працює з декількома сторінками одночасно.

В-дерева

- Аналізуючи алгоритми, що враховують роботу над даними у зовнішній пам’яті, доцільно розглядати
  - кількість звернень до диску,
  - час обчислень (процесорний час).
- Кількість звернень до диску вимірюється кількістю зчитаних або записаних сторінок інформації.
- Алгоритми роботи над В-деревами копіюють до оперативної пам’яті тільки необхідні для роботи сторінки і записують назад лише змінені сторінки.
- В довільний момент часу алгоритм працює над деякою постійною кількістю сторінок в оперативній пам’яті – а розмір самого В-дерева при цьому не обмежений.
- Вважаємо, що система сама видаляє з основної пам’яті сторінки, що більше не використовуються.

B-дерева

- Типовий алгоритм роботи з об’єктом x:

  x <= вказівник на деякий об’єкт

  DISK_READ(*x*)

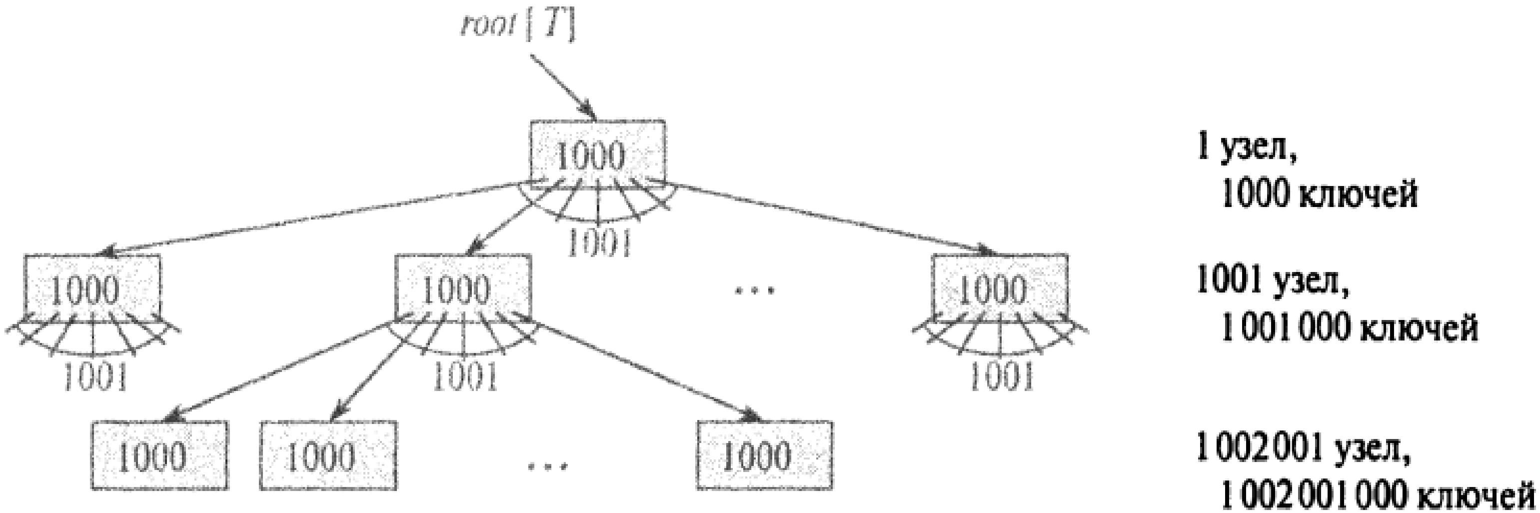
  Операції, що звертаються та/або змінюють поля *x*

  DISK_WRITE(*x*) // не потрібно, якщо поля *x* не змінювалися

  Інші операції, що не змінюють полів *x*

- Бажано мінімізувати кількість звернень до диску.
- Розмір вузла B-дерева звичайно відповідає розміру сторінки на накопичувачі.
- Таким чином кількість потомків вузла обмежена розміром сторінки.
- Зазвичай для великих B-дерев ступінь розгалуження становить від 50 до 2000.
- Чим більше розгалуження, тим менша висота дерева, а отже і кількість звернень до диску.

B-дерева



• Дерево ступеня розгалуження 1001.

• Може зберігати понад мільярд ключів.

• Оскільки корінь може зберігатися в оперативній пам’яті постійно, пошук ключа вимагатиме не більше двох звернень до накопичувача.

B-дерева

<u>Означення B-дерева</u>

Дерево T з коренем *root*[T] та властивостями:

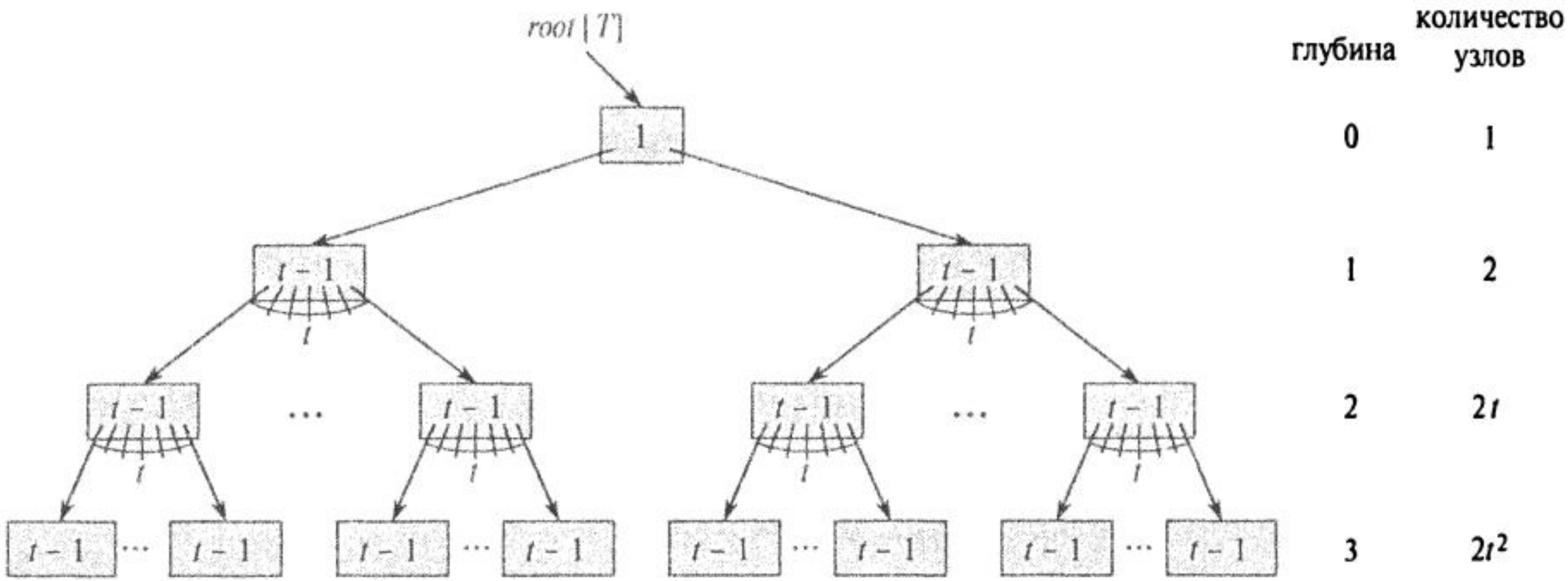
1. Кожен вузол x містить поля:
   - n[x] – поточна кількість ключів вузла x;
   - впорядковано збережені ключі, так що $key_1[x] \le key_2[x] \le \ldots \le key_{n[x]}[x]$;
   - логічне значення *leaf*[x], істинне, якщо x – лист.
2. Кожен внутрішній вузол x містить (n[x]+1) вказівник $c_1[x], \ldots, c_{n[x]+1}[x]$ на дочірні вузли.
3. Ключі $key_i[x]$ розділяють піддіапазони ключів піддерев: якщо k_i – ключ, що зберігається у піддереві з коренем $c_i[x]$, то $k_1 \le key_1[x] \le k_2 \le$ $\le key_2[x] \le \ldots \le key_{n[x]}[x] \le k_{n[x]+1}$.

B-дерева

<u>Означення B-дерева</u>

4. Всі листи розташовані на одній глибині h, що дорівнює висоті дерева. (Тобто B-дерево ідеально збаласоване за висотою.)
5. Мінімальна і максимальна кількість ключів у вузлі регламентовані фіксованим цілим t≥2 (мінімальна степінь, minimum degree):
   – кожен вузол крім кореня містить як мінімум (t–1) ключ, тобто матиме принаймні t синів; непорожнє дерево має в корені хоча б один ключ;
   – кожен вузол містить не більше (2t–1) ключів, тобто матиме максимум 2t синів; вузол вважається *повним*, якщо має рівно (2t–1) ключ.

В-дерева



В-дерево висоти 3 з мінімально можливою кількістю ключів

Висота B-дерева

<u>Теорема</u>. Висота B-дерева T з $n \geq 1$ вузлами та мінімальною степінню $t \geq 2$ не перевищує $\log_t(n+1)/2$.

<u>Доведення</u>. Нехай B-дерево має висоту h.

Корінь містить мінімум 1 ключ, решта вузлів – не менше (t–1) ключа кожен.

Отже на глибині 1 маємо мінімум 2 вузли.

На глибині 2 мінімум 2t вузлів.

На глибині 3 мінімум $2t^2$ вузлів. ………

На глибині h мінімум $2t^{h-1}$ вузол.

Кількість ключів задовольняє нерівність

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

Тобто $t^h \leq (n+1)/2$.

Прологарифмуємо по t: $h \leq \log_t(n+1)/2$.

Основні операції над B-деревами

- Пошук
- Створення нового дерева
- Додавання вузла
- Видалення вузла

Вважаємо, що корінь завжди знаходиться в оперативній пам’яті та всі вузли, що передаються як параметри, вже зчитані.

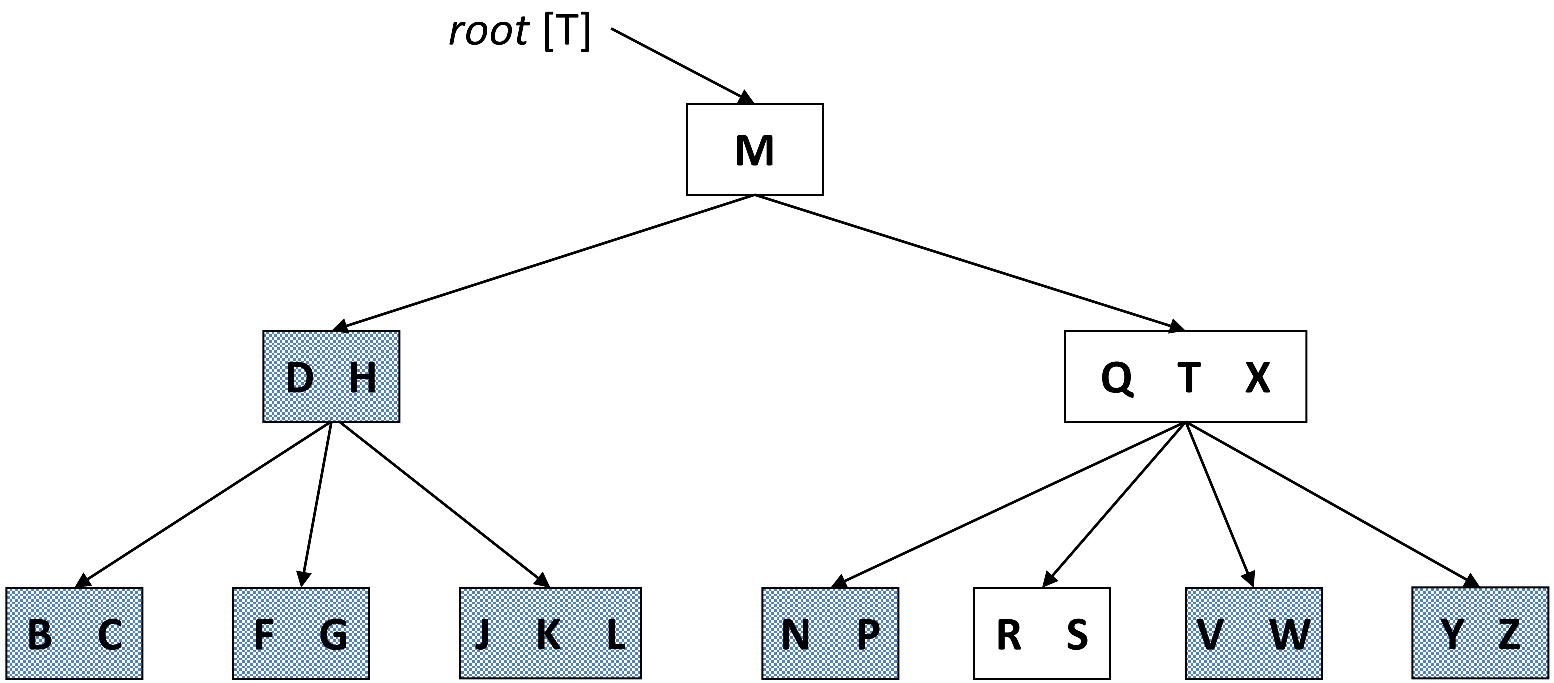
Пошук в B-дереві

АЛГОРИТМ *B_TREE_SEARCH* (x, k)

```
i <= 1
while i ≤ n[x] та k > key_i[x]
    do i <= i + 1
if i ≤ n[x] та 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)
```

- Повертає пару (x,i): $key_i[x]=k$ або NIL.
- Серед ключів вузла виконується лінійний пошук.
- Рекурсивно спускаємось від кореня до листа.
- Пройдемо дискових сторінок $O(h)=O(\log_t n)$, де h – висота B-дерева, n – кількість його вузлів.
- Перегляд ключів займе час O(t), оскільки $n[x]<2t$.
- Загальний час обчислень $O(t \cdot h)=O(t \log_t n)$.

В-дерева



Пошук ключа R в В-дереві

Створення В-дерева

АЛГОРИТМ *B_TREE_CREATE* (*T*)

x <= ALLOCATE_NODE()

leaf[*x*] <= **true**

n[*x*] <= 0

DISK_WRITE(*x*)

root[*T*] <= *x*

- При створенні дерева спочатку створюється порожній кореневий вузол, а потім вносяться нові ключі шляхом вставки вершин.
- Використовується стандартна процедура виділення нової дискової сторінки для створеного вузла ALLOCATE_NODE, яка виконується за константний час.
- Кількість дискових операцій O(1).
- Загальний час виконання O(1).

Вставка ключа у В-дерево

- Замість вставки в потрібне місце листа, як у бінарному дереві пошуку, вставляємо *новий ключ* в існуючий лист.

Яка при цьому може виникнути проблема?

Вставка ключа у B-дерево

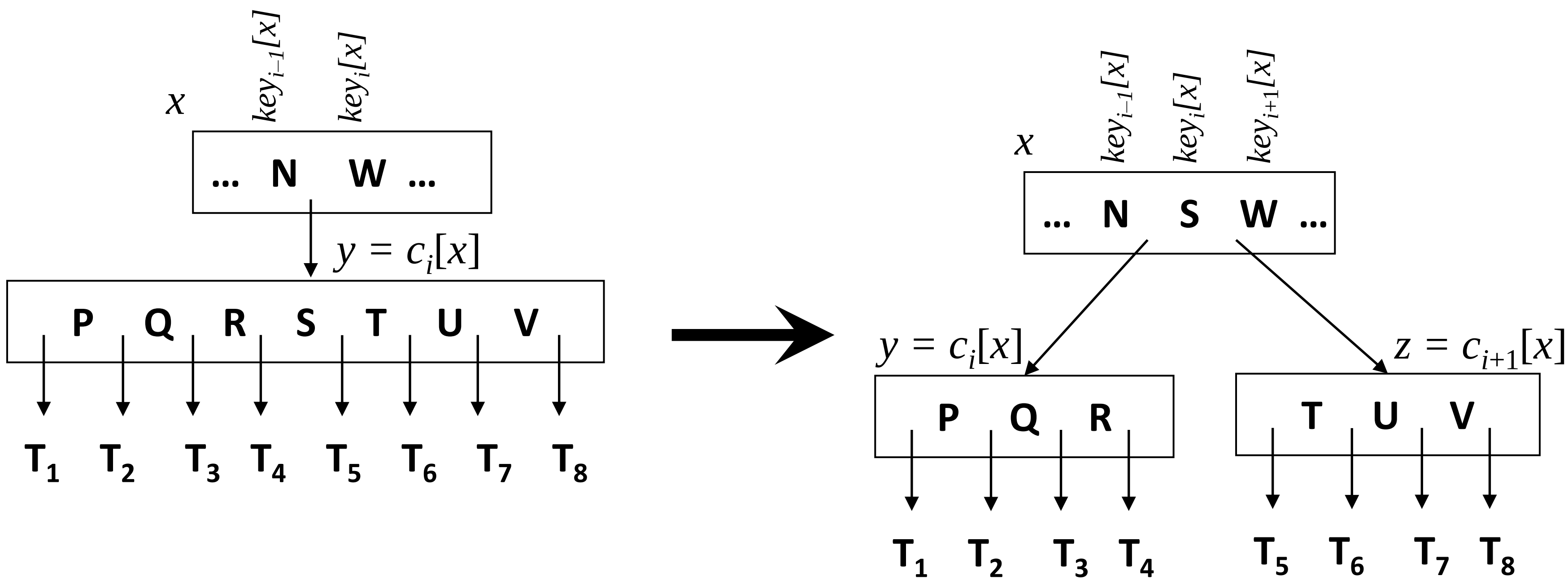
- Замість вставки в потрібне місце листа, як у бінарному дереві пошуку, вставляємо *новий ключ* в існуючий лист.
- Якщо лист вже повний, він розбивається на два, а ключ, по якому відбувається розбиття, вставляється в батьківський вузол.
- Цей вузол також може виявитися повним, тому процес розбиття може "підніматися" до кореня.

Чи можливо зробити вставку за один прохід вниз?

Вставка ключа у В-дерево

- Замість вставки в потрібне місце листа, як у бінарному дереві пошуку, вставляємо *новий ключ* в існуючий лист.
- Якщо лист вже повний, він розбивається на два, а ключ, по якому відбувається розбиття, вставляється в батьківський вузол.
- Цей вузол також може виявитися повним, тому процес розбиття може “підніматися” до кореня.
- Вставку ключа можливо здійснити за один прохід від кореня до листа, якщо по ходу робити розбиття відвіданих заповнених вузлів, включаючи лист.

Розбиття вузла в B-дереві



- Вхідні параметри – незаповнений внутрішній вузол *x*, індекс *i*, та вузол $y=c_i[x]$ – заповнений син вузла x.
- Дочірній вузол розбивається на два з (*t*–1) ключами кожний, а медіана вноситься у батьківський вузол разом з відповідною інформацією, включаючи інформацію про нового сина. Всі ключі *y*, більші за медіану, поміщаються в новий вузол *z*.

Розбиття вузла в B-дереві

АЛГОРИТМ *B_TREE_SPLIT_CHILD* (x, i, y)

1 z <= ALLOCATE_NODE()
2 $leaf[z] <= leaf[y]$
3 $n[z] <= t - 1$
4 **for** $j <= 1$ **to** $t - 1$
5   **do** $key_j[z] <= key_{j+t}[y]$
6 **if not** $leaf[y]$
7   **then for** $j <= 1$ **to** t
8   **do** $c_j[z] <= c_{j+t}[y]$
9 $n[y] <= t - 1$
10 **for** $j <= n[x] + 1$ **downto** $i + 1$
11   **do** $c_{j+1}[x] <= c_j[x]$
12 $c_{i+1}[x] <= z$
13 **for** $j <= n[x]$ **downto** i
14   **do** $key_{j+1}[x] <= key_j[x]$
15 $key_i[x] <= key_t[y]$
16 $n[x] <= n[x] + 1$
17 DISK_WRITE(y)
18 DISK_WRITE(z)
19 DISK_WRITE(x)

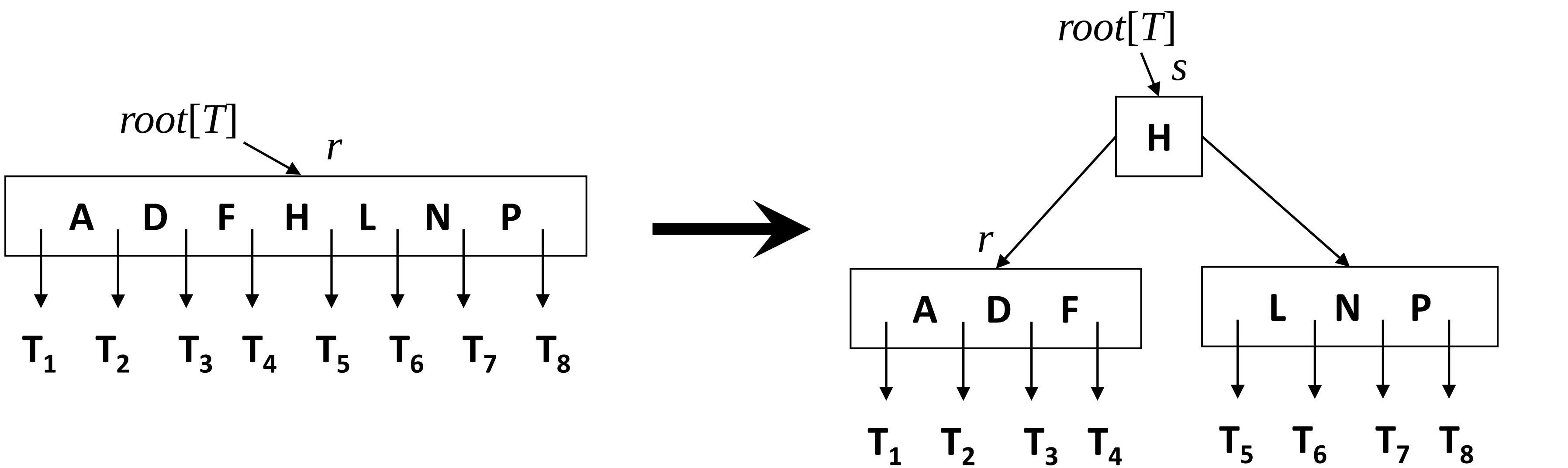
- Рядки 1–8: створення вузла z і перенесення туди (t–1) більших ключів та відповідно t правих дочірніх вузлів y.
- Рядок 9: оновлення кількості ключів в y.
- Рядки 10–16: вставка z як дочірнього вузла x з переносом медіани з y в x для розділення y та z; оновлення кількості ключів в x.
- Рядки 17–19: запис змін на диск.

Розбиття вузла в B-дереві

```
АЛГОРИТМ B_TREE_SPLIT_CHILD (x, i, y)
z <= ALLOCATE_NODE()
leaf[z] <= leaf[y]
n[z] <= t – 1
for j <= 1 to t – 1
    do key_j[z] <= key_{j+t}[y]
if not leaf[y]
    then for j <= 1 to t
             do c_j[z] <= c_{j+t}[y]
n[y] <= t – 1
for j <= n[x] + 1 downto i + 1
    do c_{j+1}[x] <= c_j[x]
c_{i+1}[x] <= z
for j <= n[x] downto i
    do key_{j+1}[x] <= key_j[x]
key_i[x] <= key_t[y]
n[x] <= n[x] +1
DISK_WRITE(y)
DISK_WRITE(z)
DISK_WRITE(x)
```

- Виконується O(1) дискових операцій.
- Процедура перерозподіляє всі ($2t$–1) ключів, тому час її роботи $\Theta(t)$.

Розбиття вузла в B-дереві



- Для розбиття заповненого кореня спочатку робимо його сином нового порожнього кореня.
- При цьому висота дерева збільшиться на одиницю.
- Розбиття – єдиний спосіб збільшення висоти B-дерева. Дерево “підростає” зверху, а не знизу.

Вставка ключа в B-дерево

```
АЛГОРИТМ B_TREE_INSERT (T, k)
1   r <= root[T]
2   if n[r] = 2t – 1
3       then s <= ALLOCATE_NODE()
4            root[T] <= s
5            leaf[s] <= false
6            n[s] <= 0
7            c1[s] <= r
8            B_TREE_SPLIT_CHILD(s, 1, r)
9            B_TREE_INSERT_NONFULL(s, k)
10      else B_TREE_INSERT_NONFULL(r, k)
```

- Рядки 3-7: випадок заповненого кореня.
- Процедура B_TREE_INSERT_NONFULL робить вставку ключа k в дерево з незаповненим коренем.
- Необхідний процесорний час $O(t \cdot h)=O(t \log_t n)$.

Вставка ключа в B-дерево

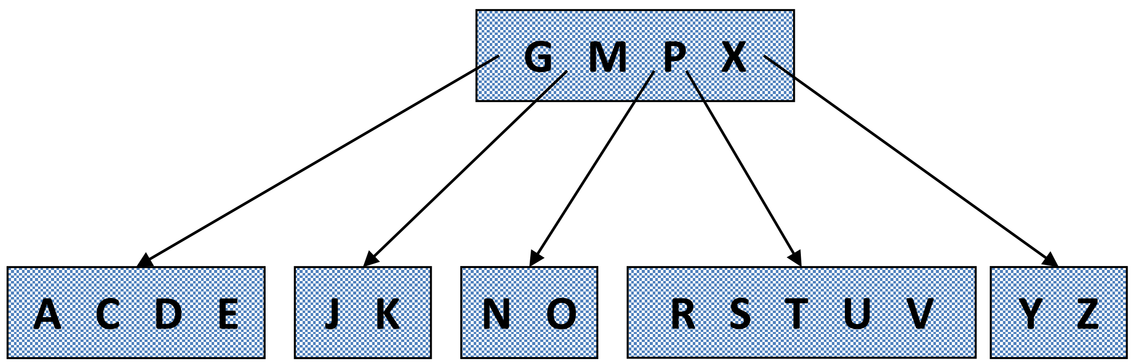
АЛГОРИТМ *B_INSERT_NONFULL* (*x*, *k*)

```
i <= n[x]
if  leaf[x]      //проста вставка ключа в лист
    then  while  i ≥ 1  та  k < key_i[x]
                do  key_{i+1}[x] <= key_i[x]
                    i <= i – 1
          key_{i+1}[x] <= k
          n[x] <= n[x] +1
          DISK_WRITE(x)
    else  while  i ≥ 1  та  k < key_i[x]  //не лист, шукаємо піддерево
                do i <= i – 1                  //для спуску
          i <= i + 1
          DISK_READ(c_i[x])
          if  n[c_i[s]] = 2t – 1               //вузол внизу повний – ділимо
                then B_TREE_SPLIT_CHILD(x, i, c_i[x])
                        if k > key_i[x]        //куди спускатися після поділу
                            then  i <= i + 1
          B_TREE_INSERT_NONFULL(c_i[x], k)
```

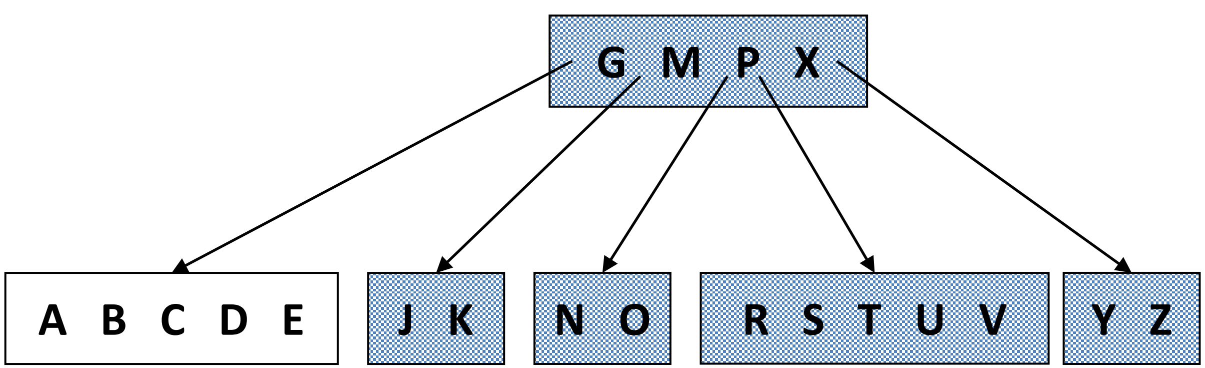
Вставка ключа в B-дерево

B-дерево мінімальної степені 3

а) початкове дерево



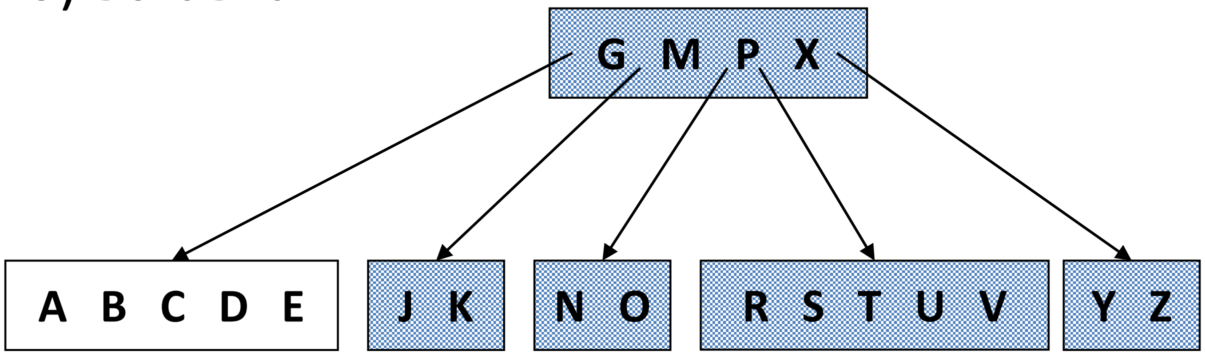
б) вставка B



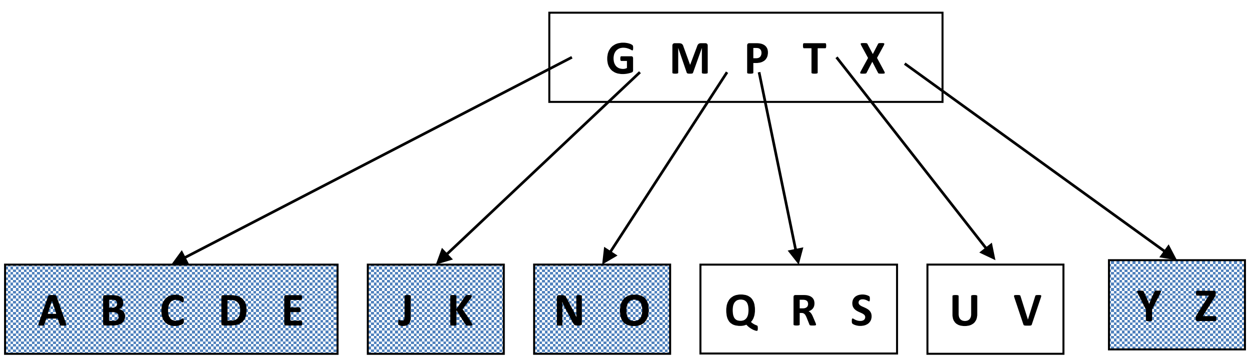
Вставка ключа в B-дерево

B-дерево мінімальної степені 3 (min 2, max 5 ключів)

б) вставка B



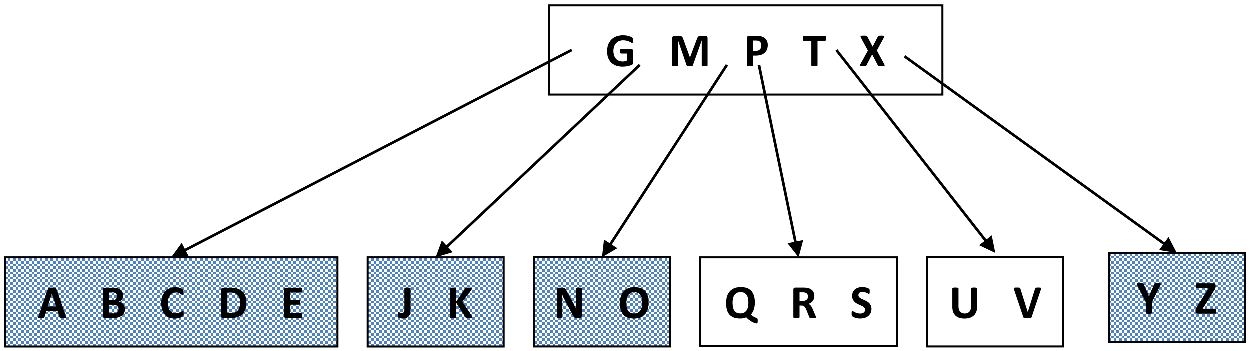
в) вставка Q



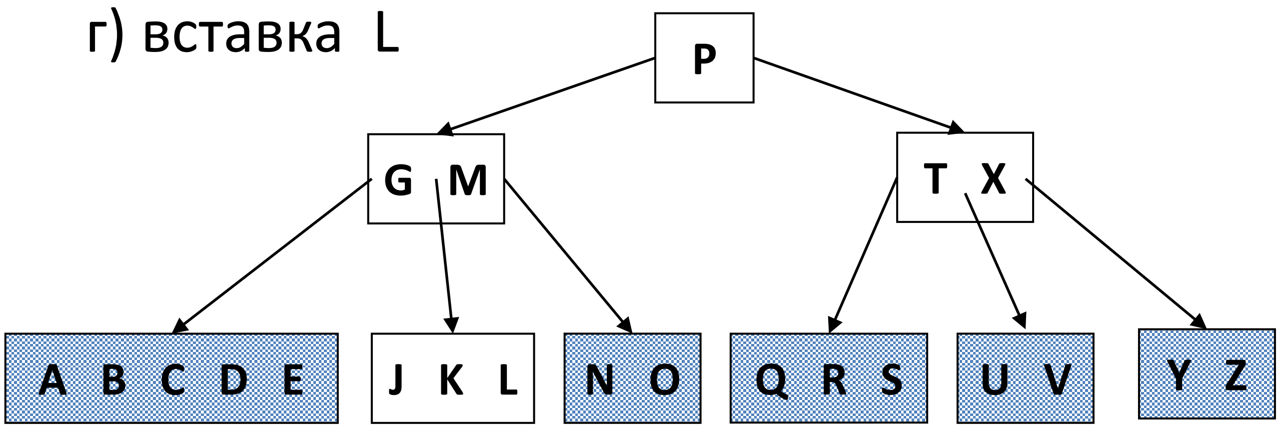
Вставка ключа в В-дерево

В-дерево мінімальної степені 3 (min 2, max 5 ключів)

в) вставка Q



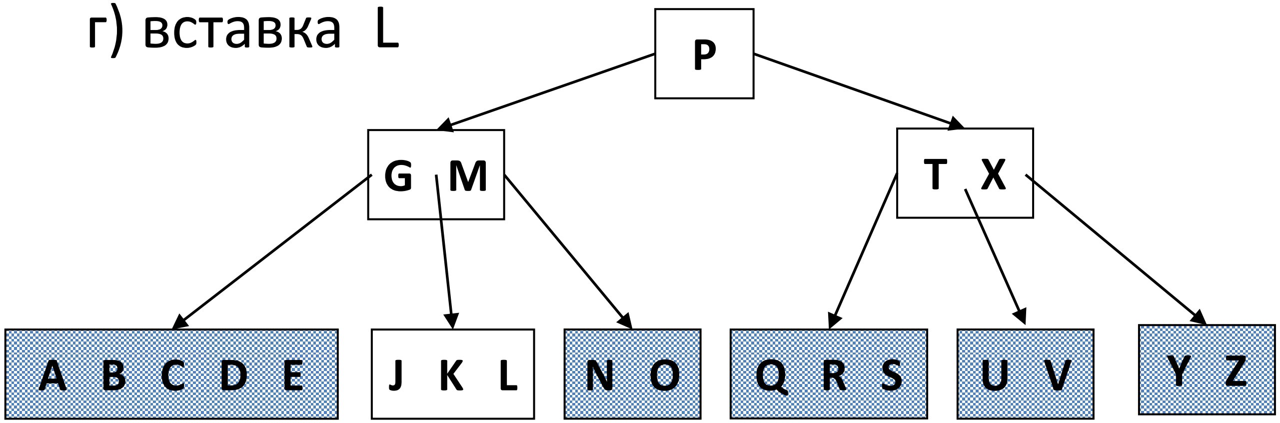
г) вставка L



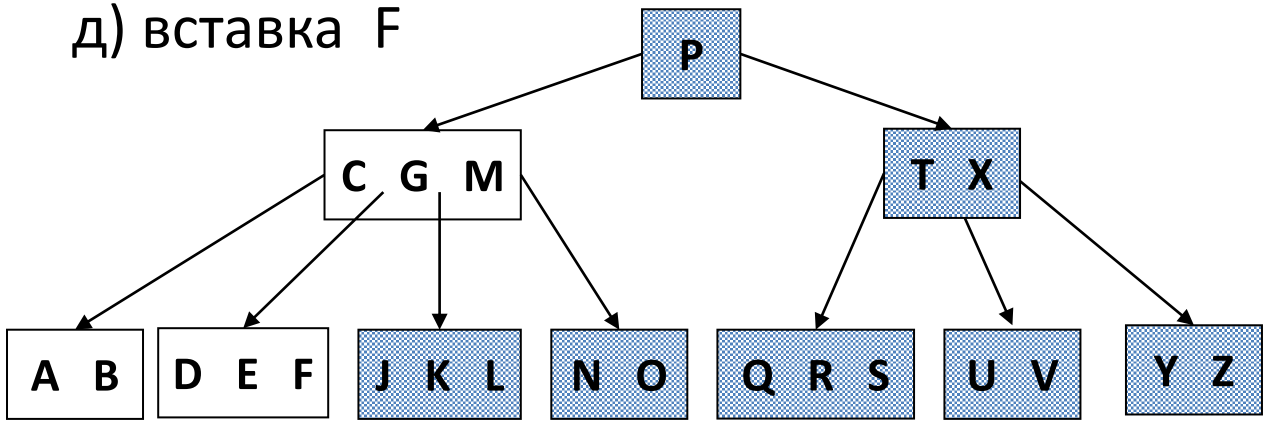
Вставка ключа в В-дерево

В-дерево мінімальної степені 3 (min 2, max 5 ключів)

г) вставка L



д) вставка F



Видалення ключа з B-дерева

- Видалення ключа з B-дерева є складнішим за вставку.

Які при цьому можуть виникати проблеми?

Видалення ключа з В-дерева

- Видалення ключа з В-дерева є складнішим за вставку.
- Ключ може видалятися з довільного вузла, а не тільки з листа. Це призведе до перебудови дочірніх вузлів.
- Тепер слід перевіряти, чи достатньо заповнені вузли (крім кореня).
- Щоб виконувати видалення ключа за один прохід, для роботи процедури потрібна сильніша умова: при виклику для вузла *x* там міститься не менше *t* ключів. Тому перед рекурсивним викликом можливе переміщення ключа в синівський вузол.
- В результаті видалення ключа висота дерева може зменшитися на 1.

Видалення ключа з В-дерева

Процедура B_TREE_DELETE(*k*,*x*). Випадки розгляду.

1. Якщо ключ *k* знаходиться у вузлі *x* та останній є листом – видаляємо *k* з *x*.
2. Якщо ключ *k* знаходиться у вузлі *x* та *x* є внутрішнім вузлом:

   а) якщо дочірній для *x* вузол *y*, що є попередником ключа *k* у вузлі *x*, містить не менше *t* ключів, то знаходимо k_1 – попередника *k* в піддереві з коренем *y*; рекурсивно видаляємо k_1 і замінюємо *k* в *x* ключем k_1 (при цьому пошук і видалення k_1 виконується за один прохід вниз);

   б) симетрична ситуація: дочірній вузол *z* – наступний за ключем *k* і містить не менше *t* ключів; тоді шукаємо наступника *k*;

Видалення ключа з B-дерева

в) інакше, якщо *y* та *z* містять по (*t*–1) ключів, вносимо *k* та всі ключі *z* в *y* (при цьому з *x* видаляється ключ *k* та вказівник на *z*, а *y* після цього міститиме (2*t*–1) ключ); звільняємо пам’ять від *z* та рекурсивно видаляємо *k* з *y*.

3. Якщо ключ *k* відсутній у внутрішньому вузлі *x*, знаходимо корінь $c_i[x]$ піддерева, що має містити *k* (якщо такий ключ існує). Якщо $c_i[x]$ містить лише (*t*–1) ключ, виконуємо крок 3а) або 3б) для гарантії переходу у вузол з мінімум *t* ключами. Потім рекурсивно видаляємо *k* з піддерева $c_i[x]$.

Видалення ключа з B-дерева

а) якщо $c_i[x]$ містить тільки (t–1) ключ, але один з його безпосередніх сусідів (брати справа і зліва, відділені єдиним ключем-роздільником) має хоча б t ключів, передамо $c_i[x]$ відповідний ключ-роздільник, на його місце поставимо крайній ключ сусіда і перенесемо відповідний вказівник від сусіда до $c_i[x]$.

б) якщо $c_i[x]$ з обома прямими сусідами містять по (t–1) ключу, об’єднаємо $c_i[x]$ з одним з цих сусідів (при цьому відповідний ключ-роздільник стане медіаною нового вузла).

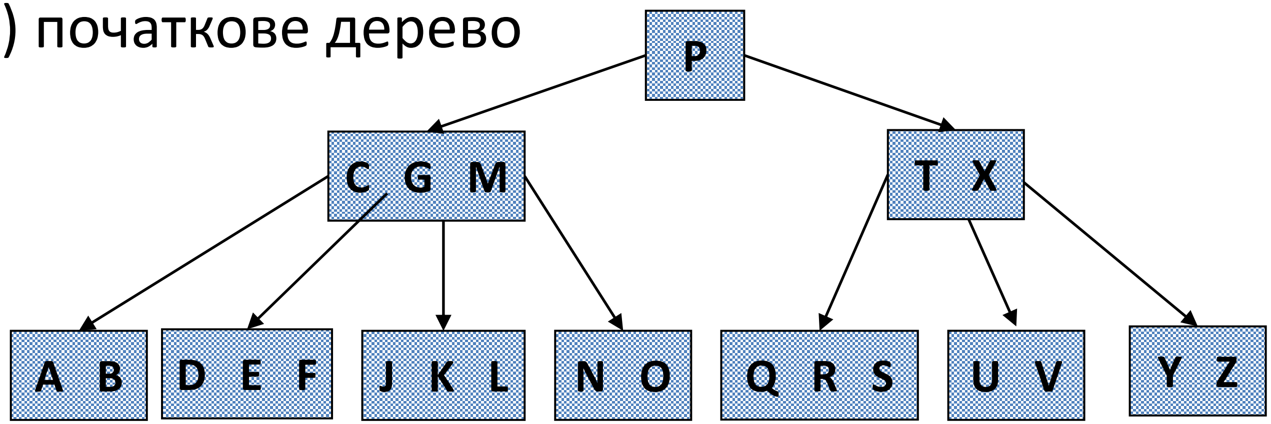
Видалення ключа з В-дерева

- Більшість ключів В-дерева знаходяться в листках, тому на практиці видалення найчастіше з листів і відбуватимуться. В цьому випадку B_TREE_DELETE виконається в один прохід униз.
- Повернення може відбутися при видаленні ключа з внутрішнього вузла (випадки 2а) та 2б)).
- Для В-дерева висоти h потрібно O(h) дискових операцій.
- Загальний процесорний час $O(t \cdot h)=O(t \log_t n)$.

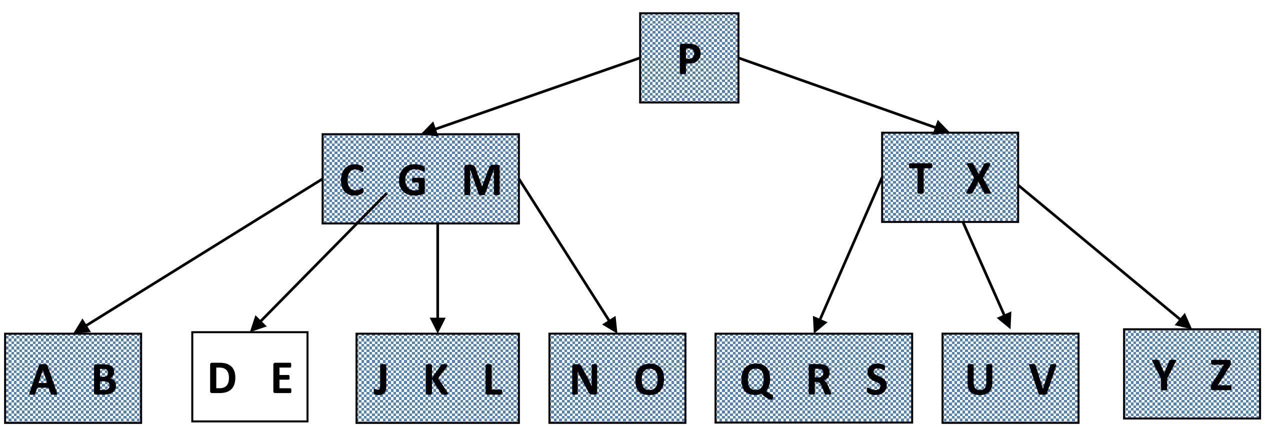
Видалення ключа з B-дерева

B-дерево мінімальної степені 3 (min 2, max 5 ключів)

а) початкове дерево



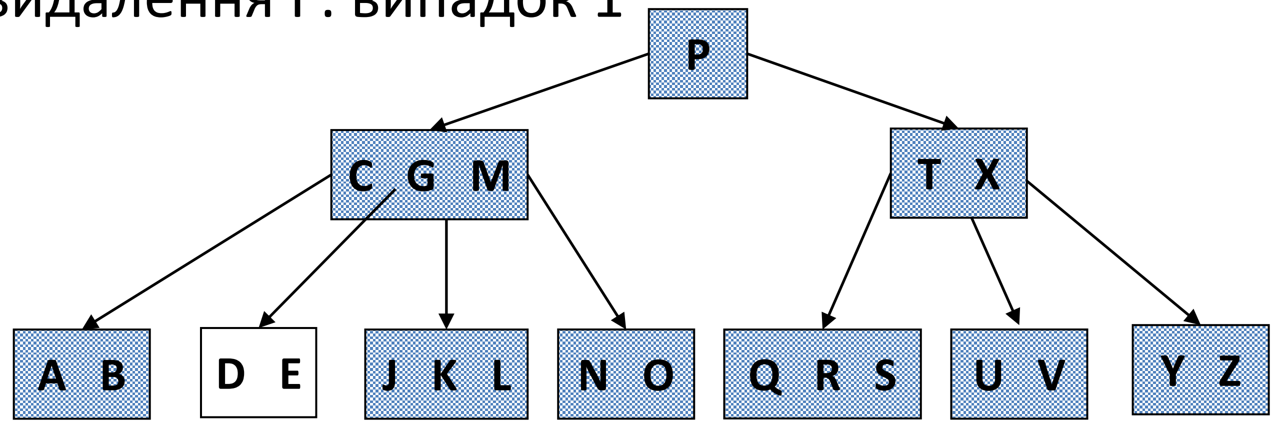
б) видалення F: випадок 1



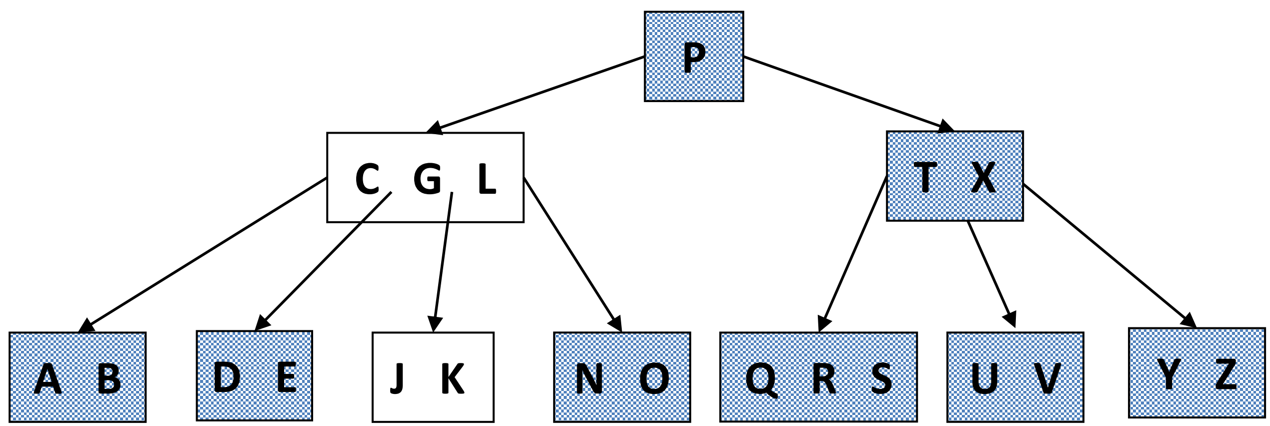
Видалення ключа з В-дерева

В-дерево мінімальної степені 3 (min 2, max 5 ключів)

б) видалення F: випадок 1



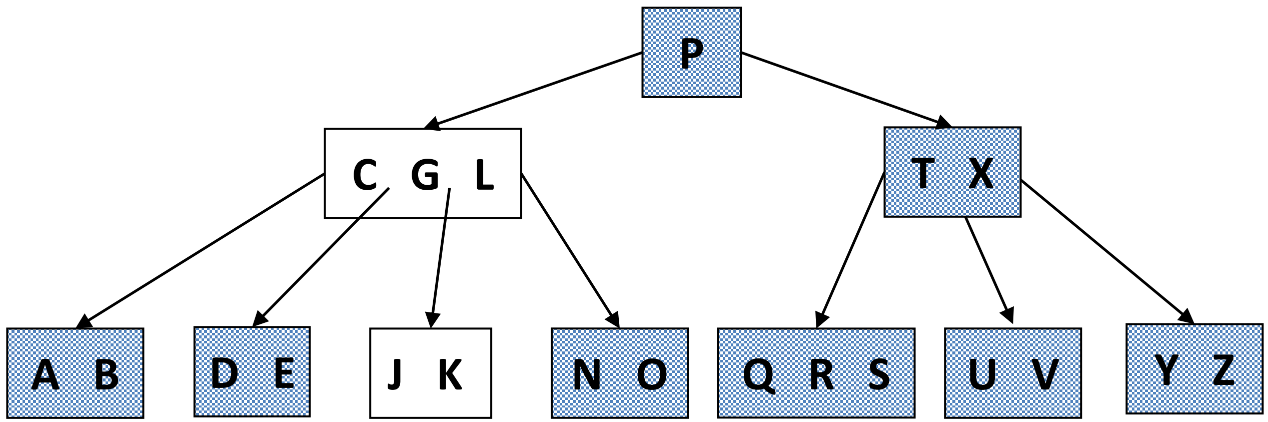
в) видалення M: випадок 2a



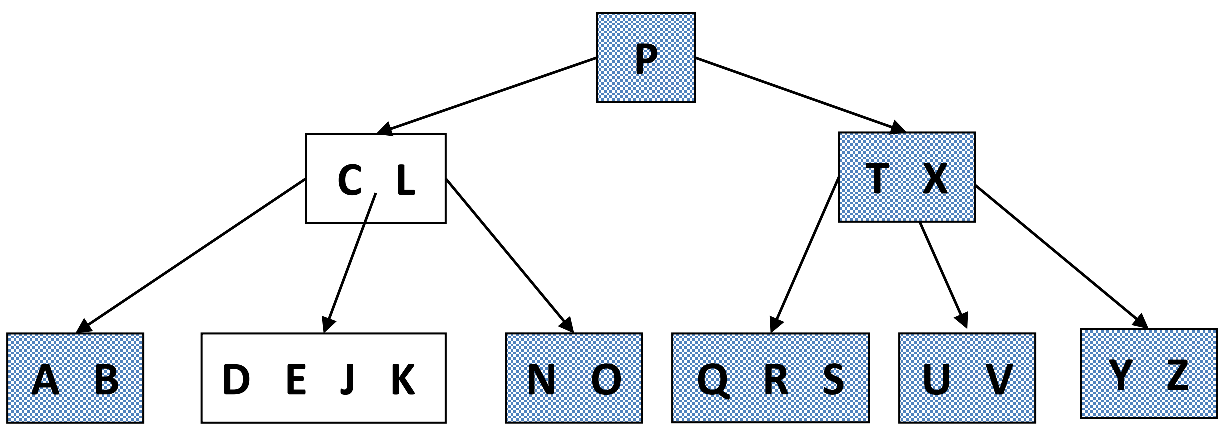
Видалення ключа з B-дерева

B-дерево мінімальної степені 3 (min 2, max 5 ключів)

в) видалення M: випадок 2а



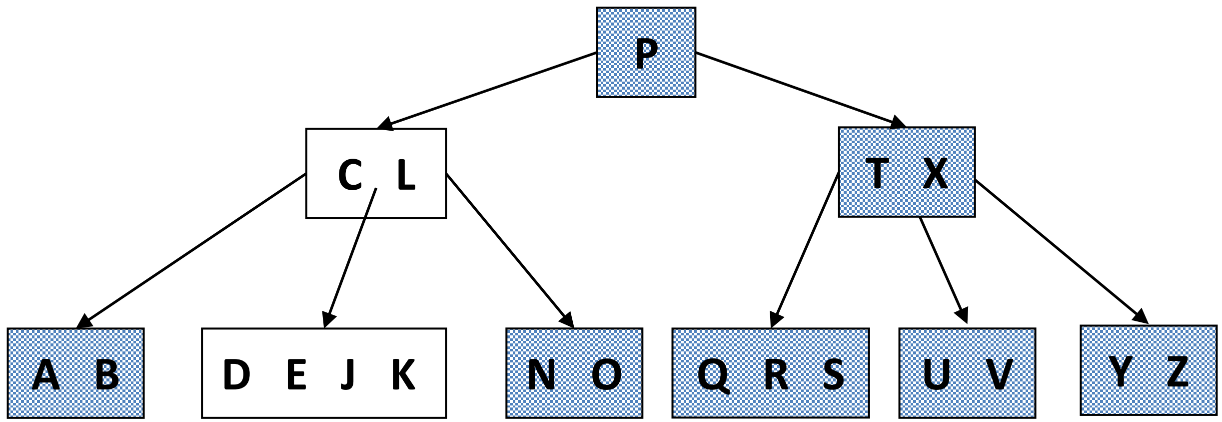
г) видалення G: випадок 2в



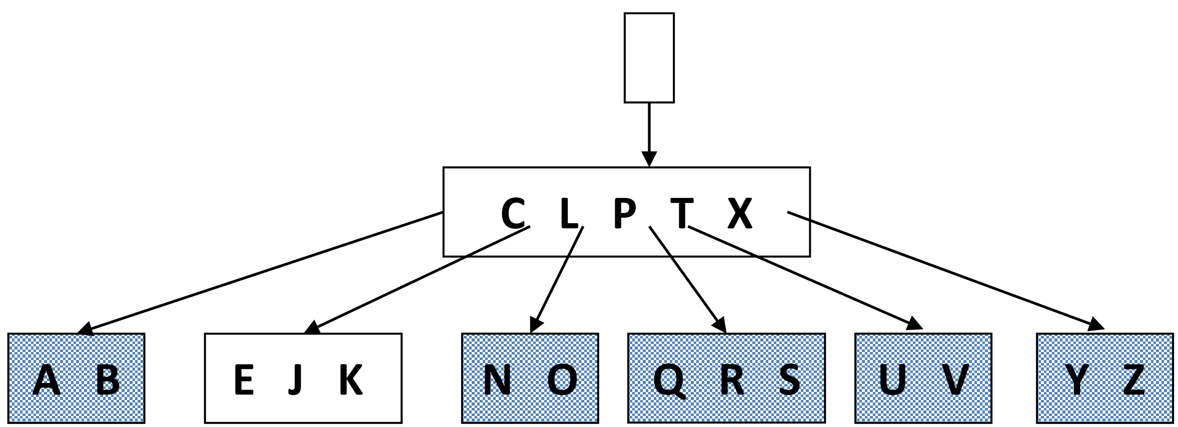
Видалення ключа з В-дерева

В-дерево мінімальної степені 3 (min 2, max 5 ключів)

г) видалення G: випадок 2в



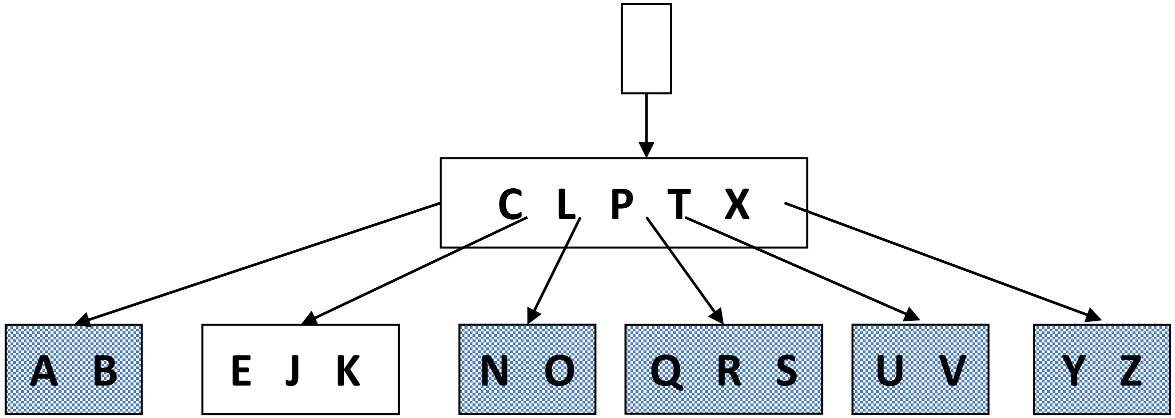
д) видалення D: випадок 3б



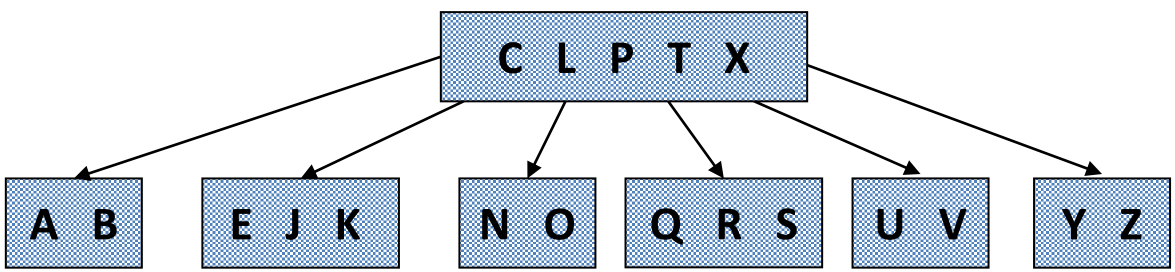
Видалення ключа з B-дерева

B-дерево мінімальної степені 3 (min 2, max 5 ключів)

д) видалення D: випадок 3б



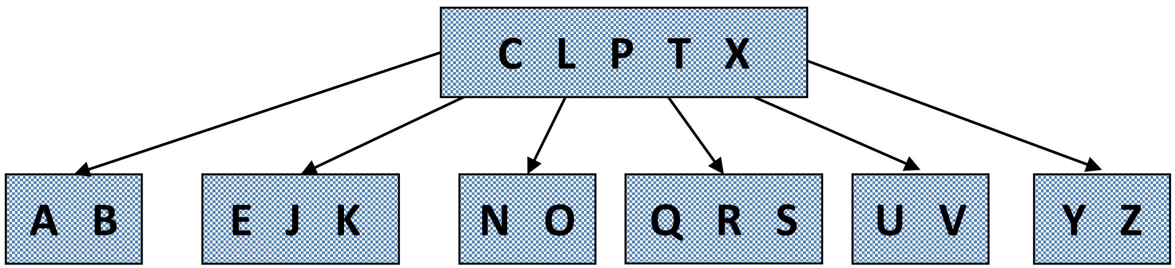
д') зменшення висоти дерева



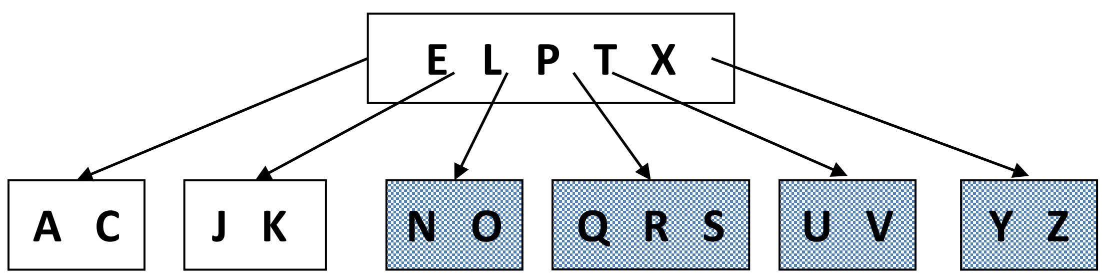
Видалення ключа з B-дерева

B-дерево мінімальної степені 3 (min 2, max 5 ключів)

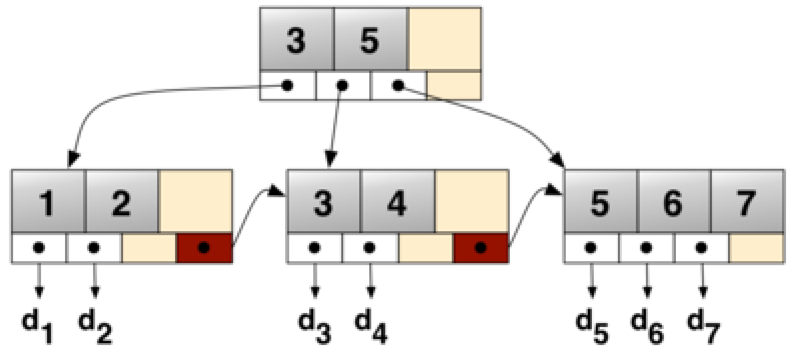
д') зменшення висоти дерева



е) видалення B: випадок 3a



B+-дерева



- Істинні значення ключів містяться тільки в листах, внутрішні вузли містять лише ключі-роздільники діапазонів піддерев.
- Листки додатково зв'язані у список. Це дозволяє швидкий доступ до ключів в порядку зростання.
- Легко реалізується незалежність програми від структури інформаційної запису.
- Пошук обов'язково закінчується в листі. Видалення ключа завжди з листа.
- Вимагають більше пам'яті для представлення, порівняно з B-деревами.

B*-дерева

- Різновид B-дерева, що вимагає заповненості кожного вузла мінімум на 2/3 (а не наполовину).
- Компактніші за звичайні B-дерева.
- Просте розділення вузлів при розбитті вже не працює.
- Замість цього – “переливання” до сусідського вузла.
- Якщо сусід також повний, відбувається поділ ключів приблизно порівну між трьома новими вузлами.
- B*-дерево, що задовольняє умовам B+ дерева, називають B+*-деревом.

2-3-дерева

- Різновид В-дерева: кожен проміжний вузол має або двох нащадків і одне поле (2-вузол), або трьох нащадків і два поля (3-вузол).
- Всі листи знаходяться на одному рівні і містять 1 або 2 поля (власне, вони й містять всю інформацію).
- Значення поля 2-вузла строго більше найбільшого значення в лівому піддереві і не менше найменшого значення в правому піддереві.
- Значення першого поля 3-вузла строго більше найбільшого значення в лівому піддереві і не менше найменшого значення в центральному піддереві. Значення другого поля 3-вузла строго більше найбільшого значення в центральному піддереві і не менше найменшого значення в правому піддереві.
- 2-3 дерева – ізометрія АА-дерев.

2-3-4-дерева

- Різновид B-дерева: мінімальна степінь t=2.
  - 2-вузол містить 1 поле і (якщо не лист) 2 нащадки.
  - 3-вузол містить 2 поля і (якщо не лист) 3 нащадки.
  - 4-вузол містить 3 поля і (якщо не лист) 4 нащадки.
- 2-3-4-дерева – ізометрія червоно-чорних дерев: це еквівалентні структури даних. Кожен чорний вузол можна об’єднати з його червоними потомками, при цьому результуючий вузол матиме ≤ 3 ключів та ≤ 4 нащадків.
- Операції вставки і видалення ключів спричиняють розширення вузлів, розбиття і злиття, що будуть еквівалентними перефарбуванням і обертанням червоно-чорних дерев.
- Ідейно 2-3-4-дерева простіші для розуміння, але червоно-чорні дерева легші в реалізації, тому саме вони і отримали використання.