B-TREE

DATA STRUCTURES AND ALGORITHMS

ThS Nguyễn Thị Ngọc Diễm
diemntn@uit.edu.vn

Nội dung

1. Giới thiệu
2. Định nghĩa
   1. Định nghĩa B-Tree using minimum degree
   2. Định nghĩa B-Tree using degree/order
3. Các phép toán cơ bản
   i. B-TREE-TRAVERSE
   ii. B-TREE-SEARCH
   iii. B-TREE-INSERTION
   iv. B-TREE DELETION
4. Ứng dụng
5. Bài tập

Nội dung

1. **Giới thiệu**
2. Định nghĩa
   1. Định nghĩa B-Tree using minimum degree
   2. Định nghĩa B-Tree using degree/order
3. Các phép toán cơ bản
   i. B-TREE-TRAVERSE
   ii. B-TREE-SEARCH
   iii. B-TREE-INSERTION
   iv. B-TREE DELETION
4. Ứng dụng
5. Bài tập

Giới thiệu

- Cây AVL hay cây Red-Black là một cấu trúc từ điển rất phù hợp khi toàn bộ cấu trúc có thể lưu trên bộ nhớ chính.
  - Following or updating a pointer only requires a memory cycle.
- Khi kích cỡ của dữ liệu rất lớn đến nổi không thể lưu trữ trên bộ nhớ chính, cấu trúc AVL không còn phù hợp nữa.
  - Các thao tác xử lý yêu cầu truy xuất đĩa (disk access).
  - Duyệt từ node root đến node lá tốn $\log_2 n$ truy xuất đĩa.

Giới thiệu

Ví dụ: Giả sử cần lưu trữ N= 1,048,576=2^{20} phần tử bằng cây AVL trên bộ nhớ ngoài.

- Chiều cao của cây AVL tương ứng là $\log_2(N) = \log_2 2^{20} = 20$.
- Hay có thể nói số lần truy cập vào bộ nhớ ngoài từ node root đến leaf là 20 lần.
- **Giả sử đĩa nhớ có số vòng quay 7200rpm, sẽ tốn 0.167 giây truy xuất đĩa. 10 lần tìm kiếm tốn hơn 1 giây! Quá chậm.**

=> Không thể cải tiện được độ phức tạp $\log_2 n$ (chặn dưới) cho cấu trúc cây nhị phân.

- Tốc độ truy xuất bộ nhớ ngoài CHẬM, số lượng truy vấn cùng lúc có thể RẤT LỚN, mỗi lần truy xuất được đơn vị nhỏ phần tử sẽ rất tốn chi phí truy xuất dữ liệu.

SOLUTION?

TODO: Đọc thêm phần giới thiệu trong sách của Mark Allen!

Giới thiệu: Duplicate keys

- https://books.google.com.vn/books?id=AiOj6n7-s_UC&pg=PA44&lpg=PA44&dq=b-tree+node+duplicates&source=bl&ots=_PbcEPOx3m&sig=ACfU3U1noHICvAk0iQbzyoIdgXIYS3cvZA&hl=en&sa=X&ved=2ahUKEwie5aTD28bpAhWtzIsBHUZxBBAQ6AEwEXoECAoQAQ#v=onepage&q=b-tree%20node%20duplicates&f=false

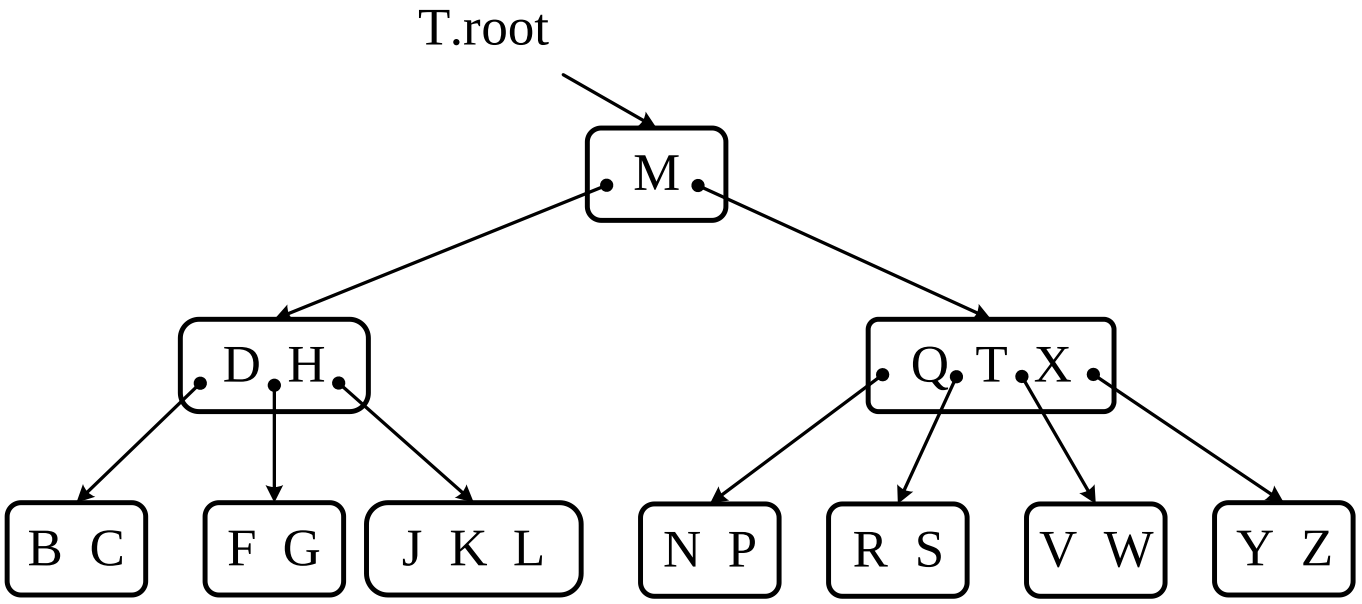
Giới thiệu

- Giải pháp:
  - Sử dụng cây tự cân bằng (self-balanced search tree)
  - Giảm số lần truy cập vào bộ nhớ ngoài => giảm chiều cao cây => tăng số nhánh. Cây có m nhánh, n phần tử => $h=\log_m(N)$, m tăng thì h sẽ giảm.VD: if m=20 then $h=\log_{20}2^{20} = 4.6 < 5$.
  - Giảm chi phí mỗi lần truy xuất đĩa (disk access). => Tăng số lượng phần tử dữ liệu lưu trữ ở mỗi node.

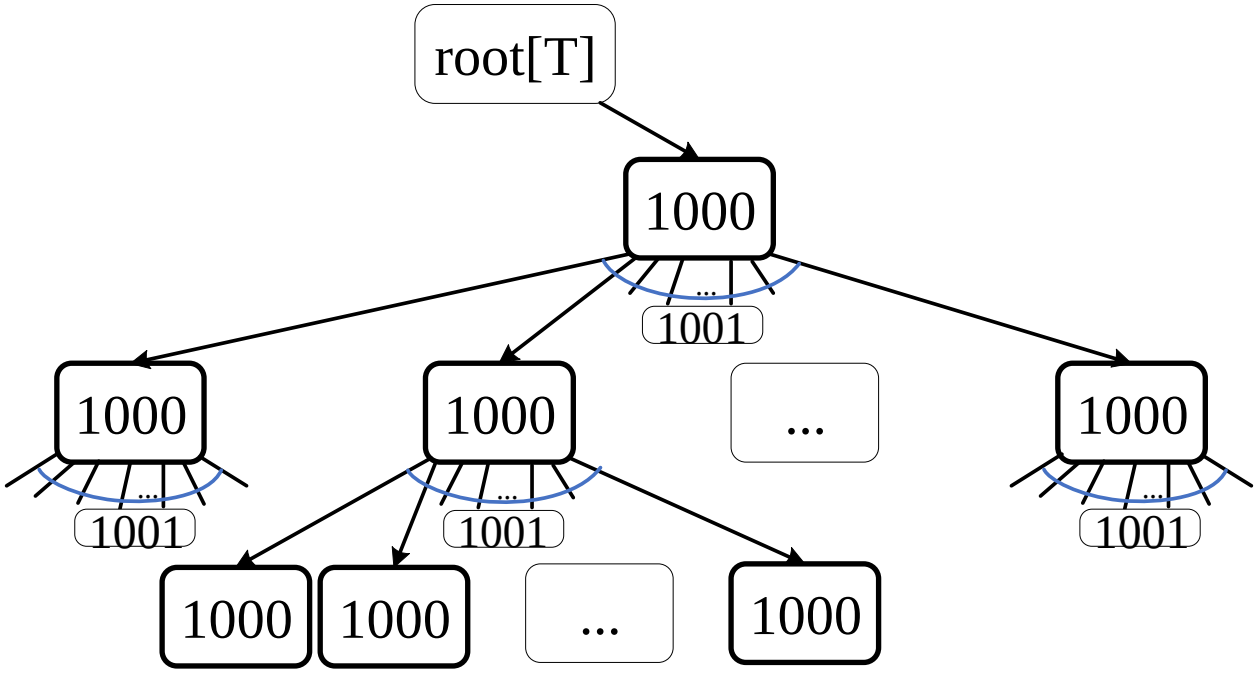
Note: Tính chất cơ bản khi truy xuất dữ liệu trên bộ nhớ ngoài: mỗi lần truy xuất sẽ đọc một đơn vị gồm 512byte hay gọi là 1 sector (đơn vị đọc ghi đĩa)

Giới thiệu

- Ví dụ cây B-tree với các khóa là phụ âm tiếng Anh. Mỗi node trong x chứa x.n khóa và x.n+1 cây con. Tất cả các node lá ở cùng mức. Giả sử muốn tìm khóa R, ta cần đi qua tất cả 3 node (M) (Q T X) (R S).



Giới thiệu



1 node
1,000 keys

1,001 nodes
1,001,000 keys

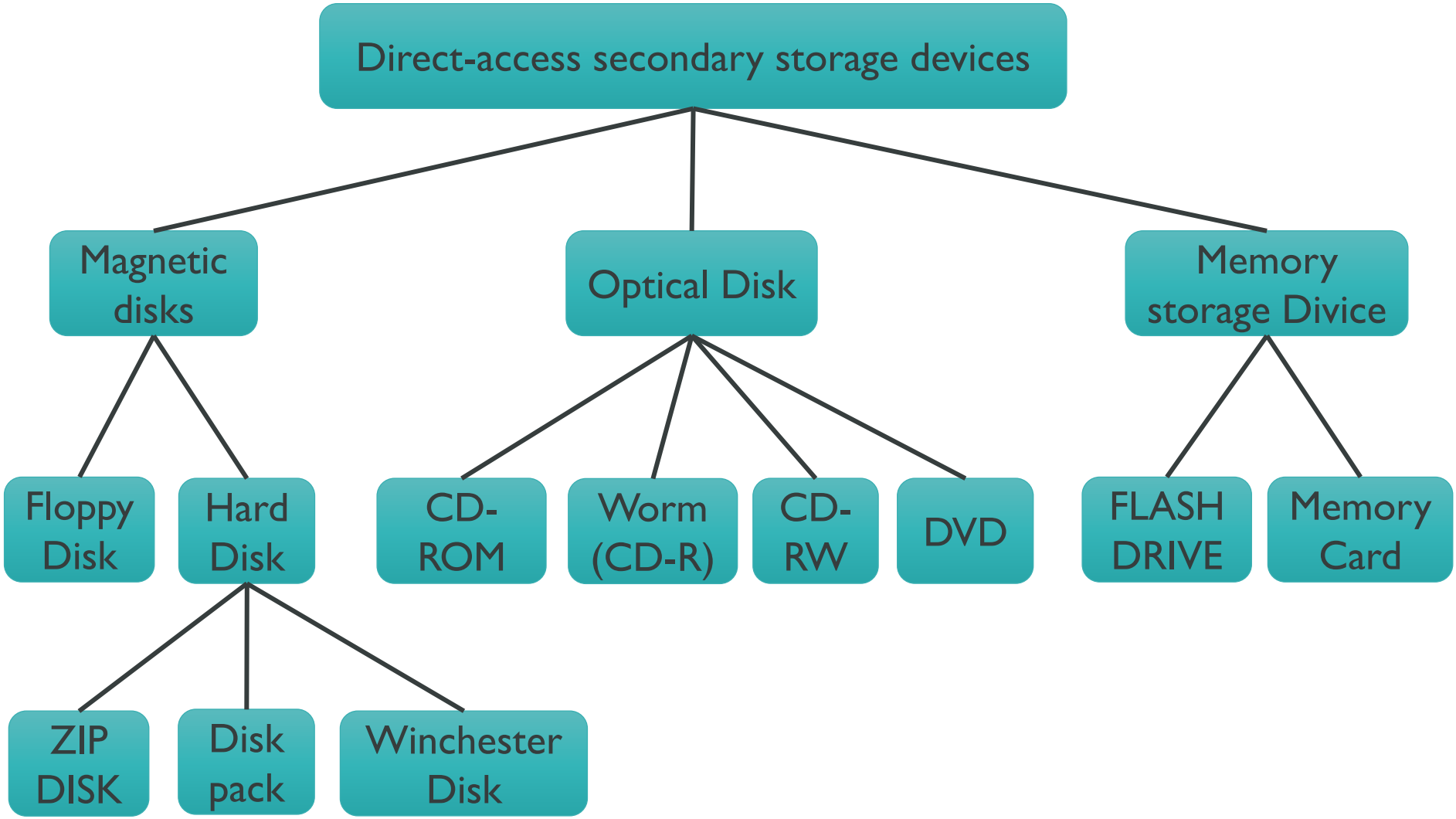
1,002,001 nodes
1,002,001,000 keys

- Ví dụ: Cây B-tree với chiều cao h=2 có hơn 1 tỷ node, với mỗi node có 1,000 khóa. Có 1,001 nodes ở độ sâu thứ nhất và hơn 1 tỷ nodes ở độ sâu thứ 2; nevertheless, since we can keep the root node permanently in main memory, we can find any key in this tree by making at most only two disk accesses.

Giới thiệu

- Cây B-tree do R.Bayer và E.M.McCreight đưa ra năm 1972 trong bài báo: "*Bayer, R.; McCreight, E. (1972), "Organization and Maintenance of Large Ordered Indexes" (PDF), Acta Informatica, 1 (3): 173–189, doi:10.1007/bf00288683*"
- B-tree là cây cân bằng được thiết kế để làm việc tốt trên đĩa từ hay thiết bị lưu trữ thứ cấp truy xuất trực tiếp khác (direct-access secondary storage device).
- Cây B tổng quát hóa cây tìm kiếm nhị phân theo cách tự nhiên.
- Các node của B-tree có thể có nhiều con.
- Mỗi node B-tree có thể quản lý số phần tử rất lớn.
- Tương tự cây Red-Black, mỗi n-node B-tree có chiều cao $O(\log_2 n)$.

Giới thiệu


Direct-access secondary storage devices
Magnetic disks
Optical Disk
Memory storage Divice
Floppy Disk
Hard Disk
CD-ROM
Worm (CD-R)
CD-RW
DVD
FLASH DRIVE
Memory Card
ZIP DISK
Disk pack
Winchester Disk

Giới thiệu

- Một node B-Tree thường lớn bằng 1 disk page (might be 211 to 214 bytes in length), và nó sẽ giới hạn số lượng con của 1 node B-Tree có thể có.
- For a large B-tree stored on a disk, we often see branching factors between 50 and 2000, depending on the size of a key relative to the size of a page. A large branching factor dramatically reduces both the height of the tree and the number of disk accesses required to find any key.

Giới thiệu

- Có 2 quy ước để định nghĩa cây B-tree.
  - Một là định nghĩa sử dụng bậc nhỏ nhất (minimum degree) được trình bày trong sách "*Introduction to Algorithms, 3rd Edition (The MIT Press, page. 487)*".
  - Hai là định nghĩa sử dụng order (bậc của một node) trong sách "*Knuth, Donald (1998), Sorting and Searching, The Art of Computer Programming, Volume 3 (Second ed.), Addison-Wesley, ISBN 0-201-89685-0. Section 6.2.4: Multiway Trees, pp. 481–491. Also, pp. 476–477 of section 6.2.3 (Balanced Trees) discusses 2-3 trees.*".

- Trong slide này sẽ trình bày định nghĩa theo minimum degree.

Nội dung

1. Giới thiệu
2. **Định nghĩa**
    1. **Định nghĩa B-Tree (minimum degree)**
    2. Định nghĩa B-Tree (degree/order)
3. Các phép toán cơ bản
    i. B-TREE-TRAVERSE
    ii. B-TREE-SEARCH
    iii. B-TREE-INSERTION
    iv. B-TREE DELETION
4. Ứng dụng
5. Bài tập

Định nghĩa B-tree using minimun degree

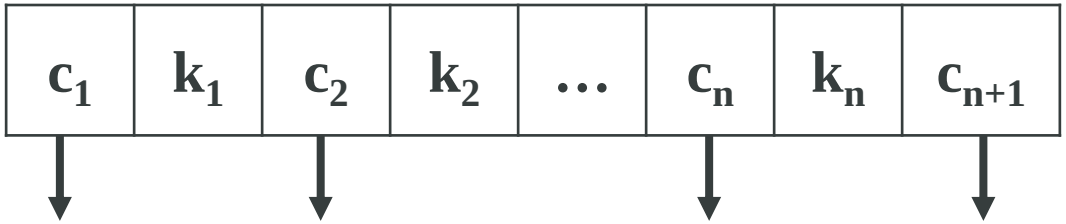
Cây B-tree T (với node root là T) có các tính chất sau:

1. Mỗi node x có các trường sau:
   - **x.n**: số lượng khóa (keys) hiện tại lưu trữ trên node x ,
   - x.n khóa được lưu theo thứ tự không giảm dần: $\mathbf{x.key_1} \leq \mathbf{x.key_2} \leq \ldots \leq \mathbf{x.key_{x.n}}$,
   - **x.leaf:** trả về giá trị boolean, TRUE nếu x là node lá, FALSE nếu x là node trong (internal node)

* *followed page. 487 in Introduction to Algorithms, 3rd Edition (The MIT Press)*

Định nghĩa B-tree

2. Nếu **x là internal node** (node trong cây khác node root và node lá), thì nó chứa **x.n+1** con trỏ **$x.c_1, x.c_2, \ldots, x.c_{x.n+1}$** tới node con của nó. **Node lá không có node con**, vì vậy các trường c_i không được định nghĩa.



3. Các khóa **$x.key_i$** phân chia phạm vi các khóa được lưu trữ trong mỗi cây con: nếu k_i là bất kỳ khóa nào được lưu trữ trong cây con có gốc $c_i[x]$ thì:

$$k_1 \le x.key_1 \le k_2 \le x.key_2 \le \ldots \le x.key_{x.n} \le k_{x.n+1}$$

4. Các node lá có cùng độ sâu h (trong đó h là chiều cao của cây)

Định nghĩa B-tree

5. Số lượng khóa của một node có **chặn trên** và **chặn dưới**. Các giới hạn này được thể hiện dưới dạng một số nguyên cố định t ≥ 2, gọi là **bậc nhỏ nhất (minimum degree)** của cây B-tree.

a. Mỗi node (khác node gốc) chứa ít nhất **t-1** khóa. Mỗi node trong (khác node gốc) có ít nhất **t** cây con. **Nếu cây không rỗng, node gốc phải có ít nhất 1 khóa**.

b. Mỗi node có thể chứa nhiều nhất **2t-1** khóa. Do đó, một node trong có thể có nhiều nhất **2t** cây con. Một node gọi là full nếu chứa đủ 2t-1 khóa.

TÓM LẠI, cho t ≥ 2 là minimum degree của cây B-tree, ta có:

t-1	≤	**số khóa** của mỗi node khác root	≤	2t-1
1	≤	**số khóa** của node root	≤	2t-1
t	≤	**số node con** của mỗi node (khác root và lá)	≤	2t
2	≤	**số node con** của root	≤	2t

Cây B-tree đơn giản nhất được tạo khi t=2. Mỗi internal node có thể có 2, 3 hoặc 4 cây con. Đây là cây 2-3-4 Tree. Tuy nhiên trong thực tế, giá trị của t lớn hơn nhiều và kết quả là B-tree với chiều cao nhỏ hơn.

Chiều cao của B-tree

- Định lý: Nếu n ≥ 1 và minimum degree t ≥ 2, thì với bất kỳ khóa n trên cây B-tree T chiều cao h, ta có:

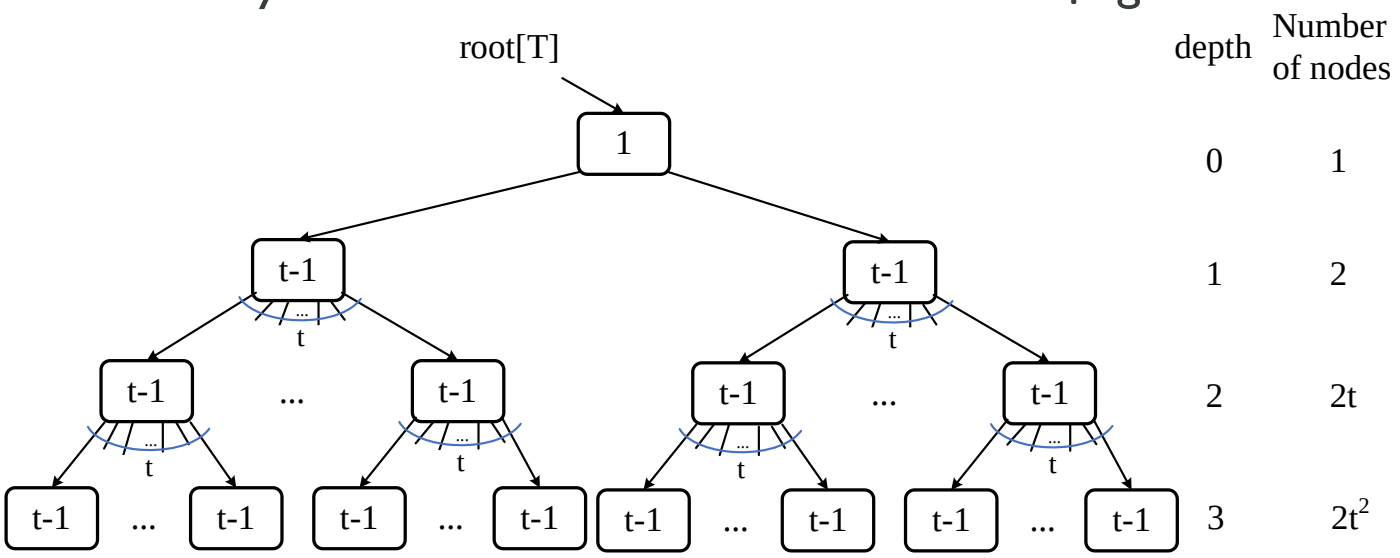
$$h \leq \log_t \frac{n+1}{2}$$

- Chứng minh:

Để chứng minh định lý trên, ta giải bài toán nhỏ sau: Tìm số lượng khóa tối thiểu (N_{min}) trên cây B-tree.

Chiều cao của B-tree

- Hình bên dưới: Cây B-tree có chiều cao h=3 chứa số lượng khóa tối thiểu có thể.



Số lượng node tại chiều cao h là $2t^{h-1}$

$$N_{min} = 1 + (t-1)\left(2 + 2t + 2t^2 + \ldots + 2t^{h-1}\right)$$
$$N_{min} = 1 + 2(t-1)\left(1 + t + t^2 + \ldots + t^{h-1}\right)$$
$$N_{min} = 1 + 2\left((t-1) + (t-1)t^1 + (t-1)t^2 + \cdots + (t-1)t^{h-1}\right)$$
$$N_{min} = 1 + 2(t - 1 + t^2 - t + t^3 - t^2 + \cdots + t^h - t^{h-1})$$
$$N_{min} = 1 + 2(-1 + t^h)$$
$$N_{min} = 1 + 2\left(t^h - 1\right)$$
$$N_{min} = 2t^h - 1$$

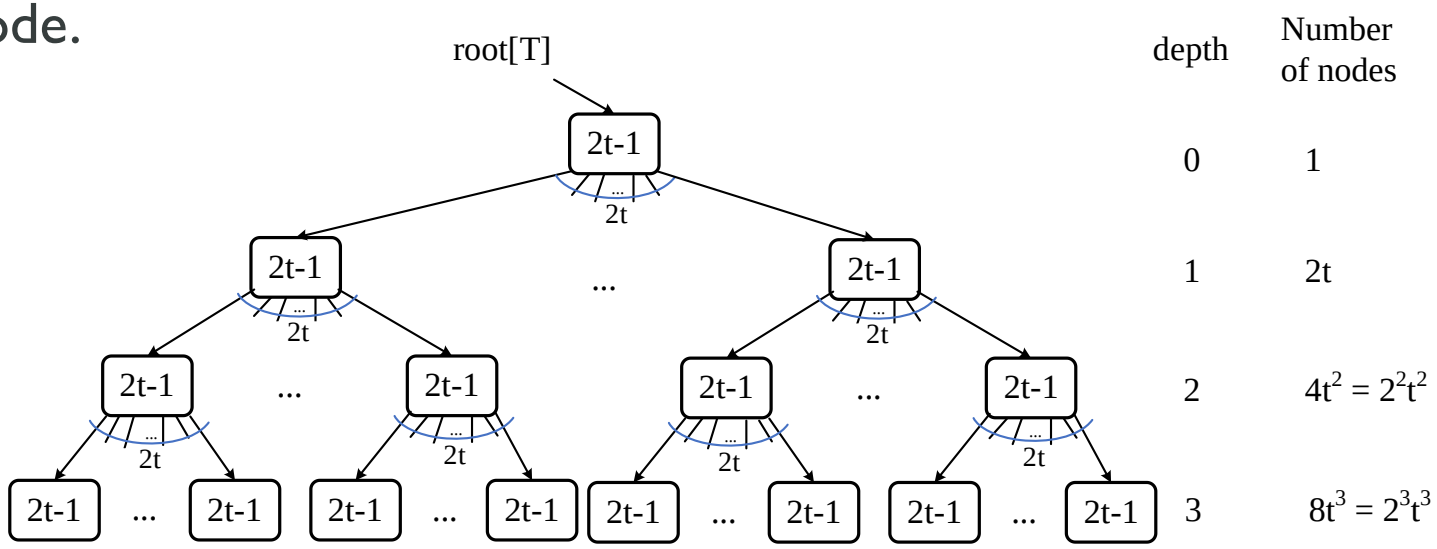
Gọi n là số lượng khóa trên cây B-tree. Ta có:

$$N_{min} \le n \Leftrightarrow 2t^h - 1 \le n \Leftrightarrow \boldsymbol{h \le \log_t \frac{n+1}{2}}$$

Chiều cao của B-tree

? Tìm số lượng khóa tối đa (N_{max}) trên cây B-tree.

Hình bên dưới: Cây B-tree có chiều cao h=3 chứa số lượng khóa tối đa tại mỗi node.



Số lượng node tại chiều cao h là (2^ht^h)

$$N_{max} = (2t-1)\left(1 + 2t + 4t^2 + \ldots + 2^ht^h\right)$$

$$N_{max} = (2t-1) + (2t-1)2t + (2t-1)4t^2 + \cdots + (2t-1)\,2^ht^h$$

$$N_{max} = 2t - 1 + 4t^2 - 2t + 8t^3 - 4t^2 + \cdots + 2^{h+1}t^{h+1} - 2^ht^h$$

$$N_{max} = -1 + 2^{h+1}t^{h+1}$$

$$N_{max} = 2^{h+1}t^{h+1} - 1$$

$$N_{max} = (2t)^{h+1} - 1$$

Nội dung

1. Giới thiệu
2. Định nghĩa
   1. Định nghĩa B-Tree (minimum degree)
   2. **Định nghĩa B-Tree (degree/order)**
3. Các phép toán cơ bản
   i. B-TREE-TRAVERSE
   ii. B-TREE-SEARCH
   iii. B-TREE-INSERTION
   iv. B-TREE DELETION
4. Ứng dụng
5. Bài tập

Định nghĩa B-Tree using degree/order

Định nghĩa

Một B-tree bậc m là cây nhiều nhánh tìm kiếm thỏa các điều kiện sau:

(i) Tất cả các node lá cùng mức.

(ii) Tất cả các node trung gian (trừ node gốc) có nhiều nhất m cây con và có ít nhất m/2 cây con (khác rỗng).

(iii) Mỗi node hoặc là node lá hoặc có k+1 cây con (k là số khoá của node này).

(iv) Node gốc có nhiều nhất m cây con hoặc có thể có 2 cây con (Node gốc có 1 khoá và không phải là node lá) hoặc không chứa cây con nào(node gốc có 1 khoá và cũng là node lá).

Exercises

1. Why don't we allow a minimum degree of $t = 1$?

2. Suppose that we insert the keys f1; 2; : : : ; ng into an empty B-tree with minimum degree 2. How many nodes does the final B-tree have?

3. Show all legal B-trees of minimum degree 2 that represent {1, 2, 3, 4, 5}.

4. Explain how to find the minimum key stored in a B-tree and how to find the predecessor of a given key stored in a B-tree.

5. Derive a tight upper bound on the number of keys that can be stored in a B-tree of height h as a function of the minimum degree t.

6. Describe the data structure that would result if each black node in a red-black tree were to absorb its red children, incorporating their children with its own.

Nội dung

1. Giới thiệu
2. Định nghĩa
3. **Các phép toán cơ bản**
   i. **B-TREE-TRAVERSE**
   ii. B-TREE-SEARCH
   iii. B-TREE-INSERTION
   iv. B-TREE DELETION
4. Ứng dụng
5. Bài tập

B-TREE-TRAVERSE

- Inorder traversal trên B-tree tương tự như Binary Tree. Ta bắt đầu từ con trái cùng, đệ quy để in con trái cùng, sau đó lặp lại quy trình tương tự cho các con còn lại để in các khóa. Bước cuối là in con bên phải cùng.

```
B-TREE-TRAVERSE(T)
1.  for (i = 1; i ≤ n; i++)
2.       if (T.leaf == FALSE)
3.            C[i]->traverse();
4.       PRINT(keys[i]);
5.  if (T.leaf == FALSE)
6.       C[i]->traverse();
```

Disk pages accessed:

CPU time:

Nội dung

1. Giới thiệu
2. Định nghĩa
3. **Các phép toán cơ bản**
   i. B-TREE-TRAVERSE
   ii. **B-TREE-SEARCH**
   iii. B-TREE-INSERTION
   iv. B-TREE DELETION
4. Ứng dụng
5. Bài tập

B-TREE-SEARCH

B-TREE-SEARCH(x, k)

- Input: x: search from this node k: key to be searched
- Return value: (x,i): key k is found at node x's i-th key

B-TREE-SEARCH

```
B-TREE-SEARCH(x, k)
1.  i = 1
2.  while i ≤ x.n and k > x.key_i do
3.      i = i+1
4.  if i ≤ x.n and k = x.key_i then
5.      return (x, i)
6.  elsif x.leaf then
7.      return NULL
8.  else
9.      DISK-READ(x.c_i)
10.     return B-TREE-SEARCH(x.c_i, k)
```

Line 1-3: tìm i nhỏ nhất sao cho $k \le x.key_i$, ngược lại: i = x.n+1.

Lines 4-5: nếu tìm thấy khóa k thì trả về vị trí tìm thấy: node x và chỉ số i, với $k=x.key_i$.

Lines 6-9: Thuật toán kết thúc nếu node đang xét là node lá. Ngược lại: Gọi hàm đệ quy để tiếp tục tìm kiếm trên cây con $x.c_i$ (sau khi thực hiện thao tác đọc dữ liệu trên cây $x.c_i$)

The number of **disk pages accessed** by B-TREE-SEARCH is **O(h) = O($\log_t$n)**, where h is the height of the B-tree and n is the number of keys in the B-tree. Since x.n < 2t, the time taken by the while loop of lines 2-3 within each node is O(t), and the total **CPU time** is **O(t h) = O(t $\log_t$n)**.

Nội dung

1. Giới thiệu
2. Định nghĩa
3. Các phép toán cơ bản
   i. B-TREE-TRAVERSE
   ii. B-TREE-SEARCH
   iii. B-TREE-INSERTION
   iv. B-TREE DELETION
4. Ứng dụng
5. Bài tập

B-TREE-CREATE

- Dùng hàm B-TREE-CREATE để tạo một node rỗng, sau đó gọi hàm B-TREE-INSERT để thêm các node mới vào cây. Cả 2 hàm đều sử dụng hàm ALLOCATE-NODE để cấp phát một disk page sử dụng như một node mới. Chúng ta có thể giả sử rằng một nút được tạo bởi ALLOCATE-NODE không yêu cầu DISK-READ, vì vẫn chưa có thông tin hữu ích được lưu trữ trên đĩa cho node đó:

```
B-TREE-CREATE(T)
1.   x = ALLOCATE-NODE()
2.   x.leaf = TRUE
3.   x.n = 0
4.   DISK-WRITE(X)
5.   T.root = x
```

B-TREE-CREATE requires **O(1) disk operations** and **O(1) CPU time**.

B-TREE-INSERTION

- Chèn 1 khóa trong B-tree phức tạp hơn chèn 1 khóa trong BST.
  - BST: Tìm vị trí thích hợp để tạo node lá mới => chèn khóa mới vào.
  - B-Tree: khóa mới sẽ được chèn vào một node lá có sẵn.
- Vì không thể chèn khóa mới vào một node lá đầy (giả sử gọi là node y), ta phải:
  - Tách node đầy y (2t-1 khóa) xung quanh khóa trung vị (median) thành 2 node (t-1 khóa mỗi node).
  - Khóa trung vị sẽ di chuyển lên trên node cha của y để xác định điểm phân chia giữa hai cây mới.
- Vậy nếu node cha của y cũng là node đầy thì sao?

=> Ta phải split node đó trước khi chèn khóa mới vào, quá trình này có thể lan truyền đến node gốc.

B-TREE-INSERTION

- TUY NHIÊN, ta có thể chèn 1 khóa mới vào B-Tree trong một lần duyệt đi từ gốc đến lá (không lan truyền ngược về gốc).
  - Để làm như vậy, không chờ để tìm xem liệu ta có thực sự cần phải chia một nút đầy đủ để thực hiện việc chèn hay không.
  - Thay vào đó, khi đi xuống cây tìm kiếm vị trí của khóa mới, thực hiện tách từng nút đầy dọc đường (bao gồm cả chính node lá).
  - Do đó, bất cứ khi nào muốn tách một nút y đầy đủ (*full node*) , ta luôn đảm bảo rằng cha của nó không đầy đủ (*nonfull node*) => không lan truyền ngược về gốc.

B-TREE-INSERTION

- Thuật toán chèn ở đây được viết theo sách *Introduction to Algorithms, 3rd Edition (The MIT Press, trang 493-497*): Đây là thuật toán chèn chủ động (proactive insertion algorithm), nghĩa là trước khi di chuyển xuống một node, ta sẽ tách node hiện tại nếu là node đầy.
- Ưu điểm của việc phân tách trước là ta sẽ không phải duyệt một node 2 lần. Nếu ta không tách một node trước khi duyệt xuống và chỉ tách nó khi khóa mới được chèn, ta có thể sẽ phải đi qua tất cả các node một lần nữa từ lá đến gốc. Điều này xảy ra trong trường hợp khi tất cả các node trên đường dẫn từ gốc đến lá đã đầy. Vì vậy, khi đến node lá, ta tách nó và di chuyển một khóa lên. Di chuyển khóa lên sẽ gây ra sự phân chia trong node cha (vì cha đầy). Hiệu ứng này không bao giờ xảy ra trong thuật toán chèn chủ động.
- Tuy nhiên, có một nhược điểm của việc chèn chủ động là có thể thực hiện các phân tách node không cần thiết.

Splitting a node

- Đầu vào hàm B-TREE-SPLIT-CHILD:
  - Con trỏ trong không đầy x (nonfull internal node x)
  - Chỉ số i: để truy cập tới $x.c_i$ là con đầy của x (full child of x), giả sử ta gọi là y

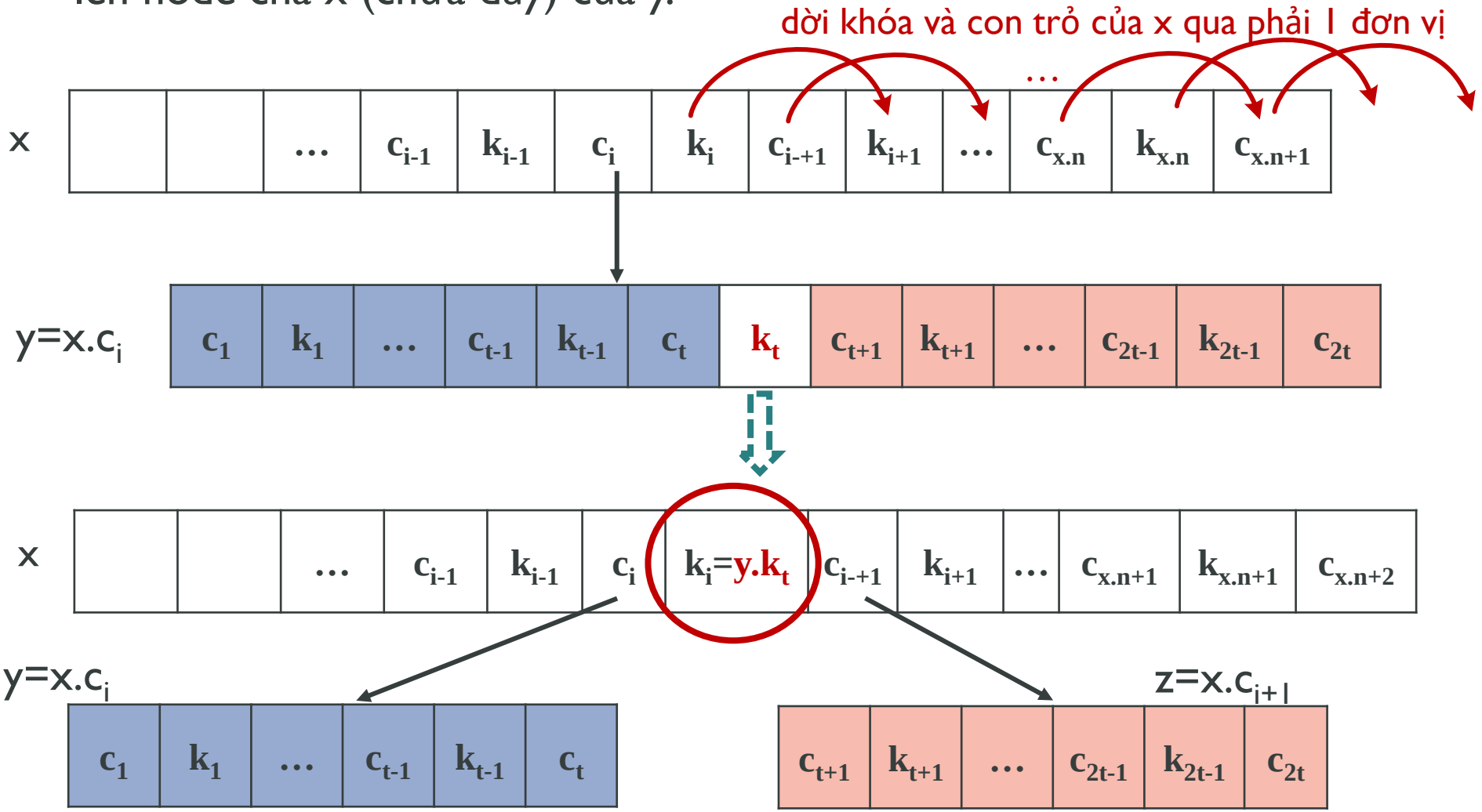
  Giả sử node x và node $x.c_i$ đã được lưu trong main memory.
- Đầu ra: hàm B-TREE-SPLIT-CHILD sẽ tách y thành 2 node y và z, node x sẽ có thêm 1 con là node z và 1 khóa (khóa trung vị của node y ban đầu).
- Để tách 1 gốc đầy (full root):
  - Trước tiên ta sẽ biến root hiện tại thành con của một nút root trống mới, và sau đó ta có thể dùng B-TREE-SPLIT-CHILD.
  - Node gốc bị tách dẫn đến việc tăng trưởng chiều cao của cây. Không giống B-Tree, chiều cao của cây BST tăng do sự phát sinh node lá ở mức mới.

Splitting a node

- B-TREE-SPLIT-CHILD hoạt động khá đơn giản chỉ "cutting and pasting":
  - Node y là con thứ i của x
  - Node y ban đầu có 2t con (2t-1khóa) sẽ được giảm xuống t con (t-1 khóa). Node z sẽ lấy t con lớn nhất (t-1 khóa) từ y và trở thành con thứ i+1 của x.
  - Khóa median của y sẽ di chuyển lên trên và trở thành khóa của x phân chia node y và z.

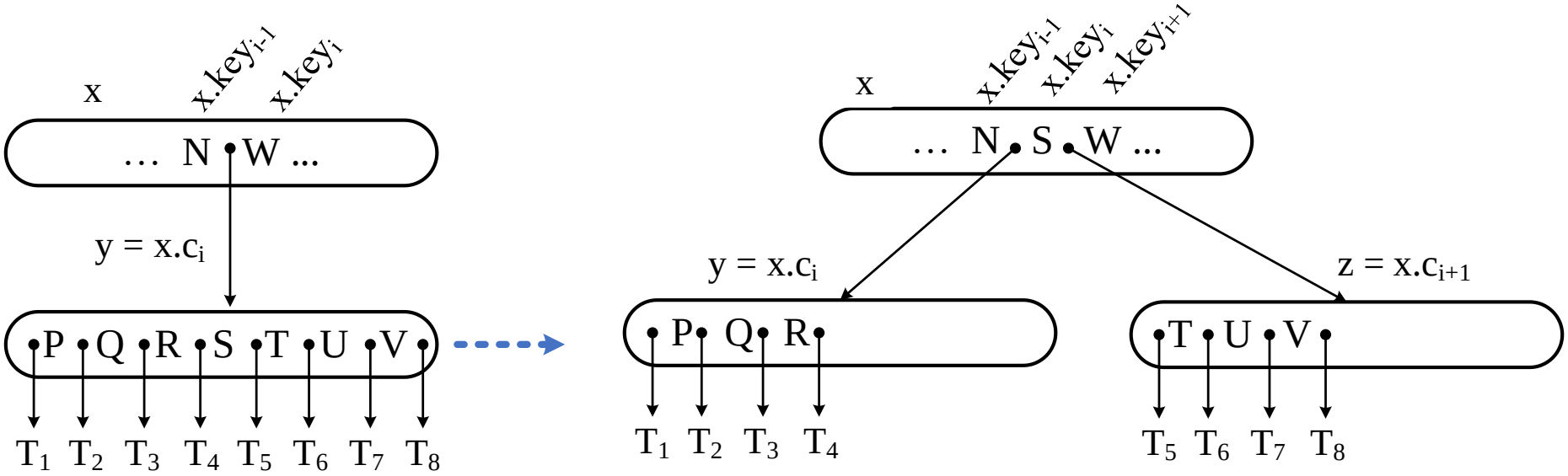
Splitting a node: Minh họa

- Minh họa: split node đầy y thành 2 node y và z, đưa khóa median từ y lên node cha x (chưa đầy) của y.



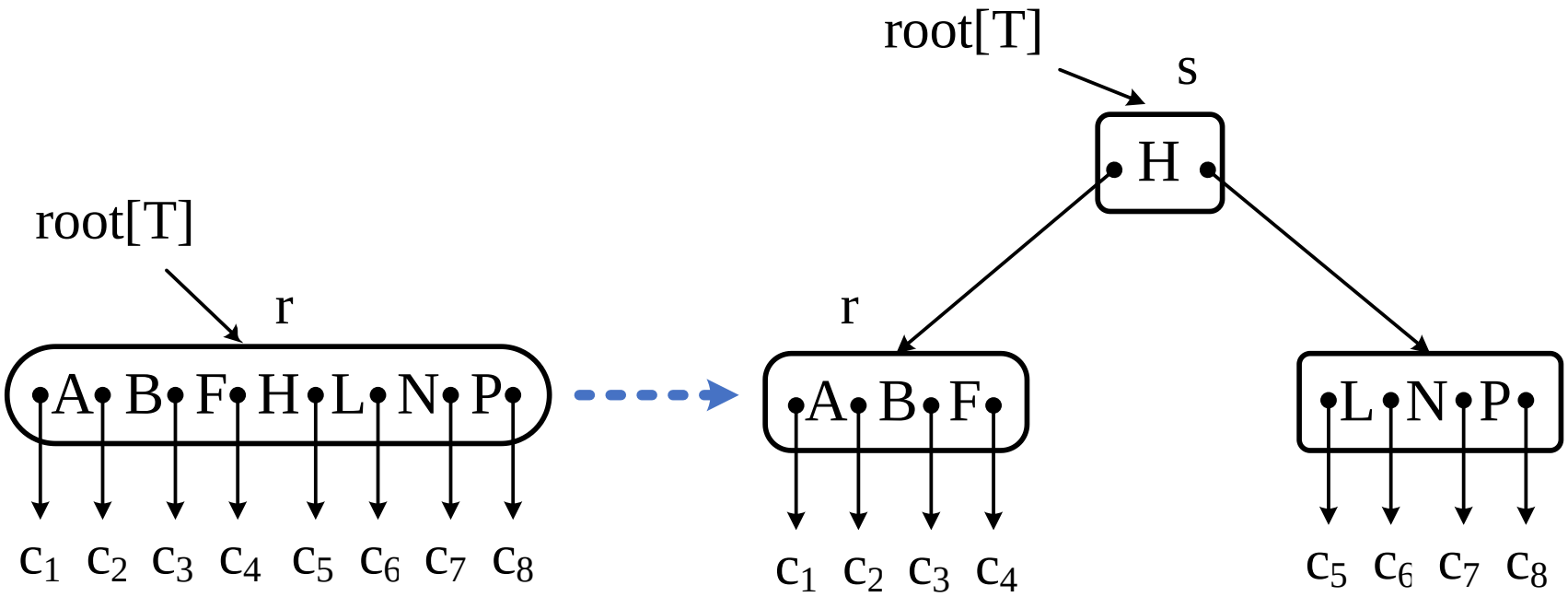
Splitting a node:Ví dụ:

- Ví dụ bên dưới minh họa quá trình split cho cây B-Tree t=4.
  - Ta cần tách node đầy $y=x.c_i$ với S là khóa median.
  - Khóa S sẽ di chuyển lên node cha của y là node x.
  - Tất cả khóa lớn hơn median sẽ di chuyển sang node mới là node z, node z trở thành con của node x.



Splitting a node: Split node Root

- Tách gốc với t=4. Root r được tách làm 2 node. Một root mới s được tạo ra chứa giá trị median của r, và nhận 2 node vừa được tách ra làm con. Cây B-Tree tăng trưởng lên 1 đơn vị chiều cao khi node root được tách.



Splitting a node: Thuật toán

```
B-TREE-SPLIT-CHILD(x, i)
1.    y = x.c_i // con thứ i của x
2.    z = ALLOCATE-NODE()
3.    z.leaf = y.leaf
4.    z.n = t-1 // số khóa của z
5.    for j = 1 to t-1
6.        z.key_j = y.key_j+t
7.    if not y.leaf
8.        for j = 1 to t
9.            z.c_j = y.c_j+t
10.   y.n = t-1
11.   for j=x.n+1 downto i+1
12.       x.c_j+1 = x.c_j
13.   x.c_i+1 = z
14.   for j = x.n downto i
15.       x.key_j+1 = x.key_j
16.   x.key_i = y.key_t
17.   x.n = x.n+1 // số khóa của x
18.   DISK-WRITE(y)
19.   DISK-WRITE(z)
20.   DISK-WRITE(x)
```

Line 10: Điều chỉnh số lượng khóa của node y

Line 2-9: Tạo node z, Chuyển một nửa cuối các khóa và con trỏ của y sang z

Line 11-17: Chèn node z làm con của node x, di chuyển khóa midian từ y lên x nhằm tách y và z, điều chỉnh số lượng khóa của node x.

Lines 18-20: write out all modified disk pages.

- The CPU time used by B-TREE-SPLIT-CHILD is $\Theta(t)$, due to the loops on lines 5-6 and 8-9. (The other loops run for at most t iterations.).
- The procedure performs $O(1)$ disk operations.

B-TREE-INSERTION

- **TODO: CHƯA XONG**
- Inserting a key into a B-tree in a single pass down the tree:
- Quá trình thêm một khoá mới (newkey) vào B-tree có thể được mô tả như sau:
  - Bước 1: Khởi tạo x là root của cây B-Tree.
  - Bước 2: Nếu x là node lá, qua Bước 3. Ngược lại, trong khi x không phải là node lá:

    2.1 Nếu x là node root đầy, thực hiện phân tách node gốc trước khi di chuyển đến cây con thích hợp để chèn newkey.

    2.1 Tìm node con của node x sẽ duyệt tiếp theo. Gọi là node y.

    2.2 Nếu y là node không đầy, gán x trỏ tới y.

    2.3 Ngược lại nếu y là node đầy, thực hiện tách node y thành 2 node con, đưa mediancủa y lên node x. Nếu newkey > khóa median thì gán x trỏ tới phân nửa thứ 2 của node y vừa tách ra. Ngược lại gán x trỏ tới phân nửa thứ 1.
  - Bước 3: Node x sẽ còn chỗ trống để thêm newkey vào vì ta đã tách tất cả các node đầy trong quá trình duyệt node trước đó.

B-TREE-INSERTION

- **TODO: CHƯA XONG**

- Hàm chèn một khóa kết thúc bởi việc gọi hàm chèn một khóa vào node không đầy (B-TREE-INSERT-NONFULL).

- The procedure finishes by calling B-TREE-INSERT-NONFULL to perform the insertion of key k in the tree rooted at the nonfull root node.

- B-TREE-INSERT-NONFULL recurses as necessary down the tree, at all times guaranteeing that the node to which it recurses is not full by calling B-TREE-SPLITCHILD as necessary.

- Inserting a key k into a B-tree T of height h is done in a single pass down the tree, requiring **O(h) disk accesses**. The **CPU time** required is **$O(th) = O(t\log_t n)$**.

B-TREE-INSERTION

```
B-TREE-INSERT(T, k)
1.   r = T.root
2.   if r.n = 2t-1 then
3.        s = ALLOCATE-NODE()
4.        T.root = s
5.        s.leaf = FALSE
6.        s.n = 0
7.        s.c1 = r
8.        B-TREE-SPLIT-CHILD(s, 1)
9.        B-TREE-INSERT-NONFULL(s, k)
10.  else B-TREE-INSERT-NONFULL(r, k)
```

Lines 3-9: xử lý trường hợp trong đó node root r đầy: root bị tách (Line 8) và một node mới s (có hai con) trở thành root.
Line 9: Chèn k vào node không đầy s.

Lines 10: Chèn k vào node root không đầy.

B-TREE-INSERTION

```
B-TREE-INSERT-NONFULL(x, k)
1. i = x.n
2. if x.leaf then
3.     while i ≥ 1 and k < x.key_i
4.         x.key_{i+1} = x.key_i
5.         i = i-1
6.     x.key_{i+1} = k
7.     x.n = x.n + 1
8.     DISK-WRITE(x)
9. else
10.    while i ≥ 1 and k < x.key_i
11.        i = i-1
12.    i = i+1
13.    DISK-READ(x.c_i)
14.    if x.c_i.n = 2t-1 then
15.        B-TREE-SPLIT-CHILD(x, i)
16.        if k > x.key_i then
17.            i = i+1
18.    B-TREE-INSERT-NONFULL(x.c_i, k)
```

B-TREE-INSERTION

```
B-TREE-INSERT-NONFULL(x, k)
1. i = x.n
2. if x.leaf then
3.     while i ≥ 1 and k < x.key_i
4.         x.key_{i+1} = x.key_i
5.         i = i-1
6.     x.key_{i+1} = k
7.     x.n = x.n + 1
8.     DISK-WRITE(x)
9. else
10.    while i ≥ 1 and k < x.key_i
11.        i = i-1
12.    i = i+1
13.    DISK-READ(x.c_i)
14.    if x.c_i.n = 2t-1 then
15.        B-TREE-SPLIT-CHILD(x, i)
16.        if k > x.key_i then
17.            i = i+1
18.    B-TREE-INSERT-NONFULL(x.c_i, k)
```

Lines 10-12: tìm con của x để gọi đệ quy duyệt xuống cây thích hợp chèn k.

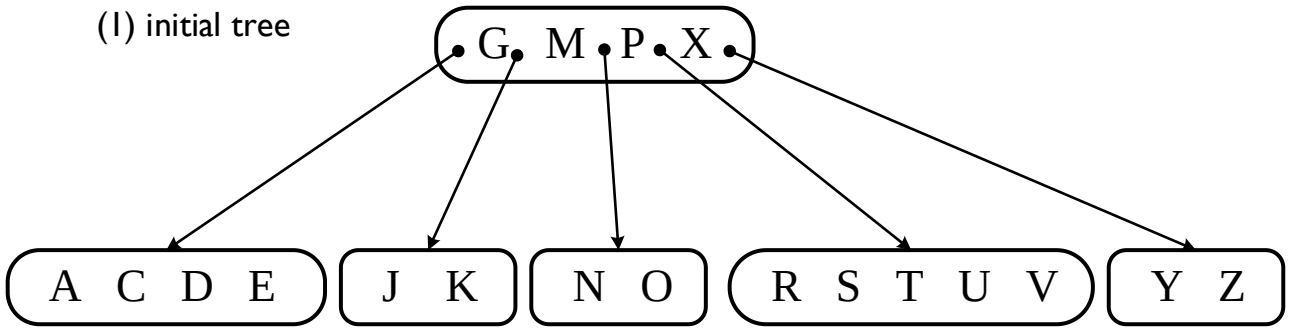
Lines 2-8: x là node lá: chèn khóa k vào vị trí thích hợp trên node lá.

Line 14-17: để đảm bảo rằng hàm chèn không bao giờ gọi đệ quy trên một node đầy => thực hiện tách node ở Line 15. Line 16-17: xác định cây con để gọi đệ quy duyệt xuống chèn k trong hai con vừa được tách ra.

Line 18: gọi đệ quy để chèn k vào cây con thích hợp.

B-TREE-INSERTION: Ví dụ 1

- **Ví dụ 1:** Thêm lần lượt 4 giá trị (B, Q, L, F) vào cây B-tree có t=3 cho sẵn như hình bên:



Bài làm:

Áp dụng vào bài:

Cây B-tree có t=3:

Nhắc lại: Cho $t \geq 2$ là minimum degree của cây B-tree, ta có:

$t-1 \leq$ **số khóa** của mỗi node khác root $\leq 2t-1$

$1 \leq$ **số khóa** của node root $\leq 2t-1$

$t \leq$ **số node con** của mỗi node (khác root và lá) $\leq 2t$

$2 \leq$ **số node con** của root $\leq 2t$

$2 \leq$ **số khóa** của mỗi **node khác root** ≤ 5

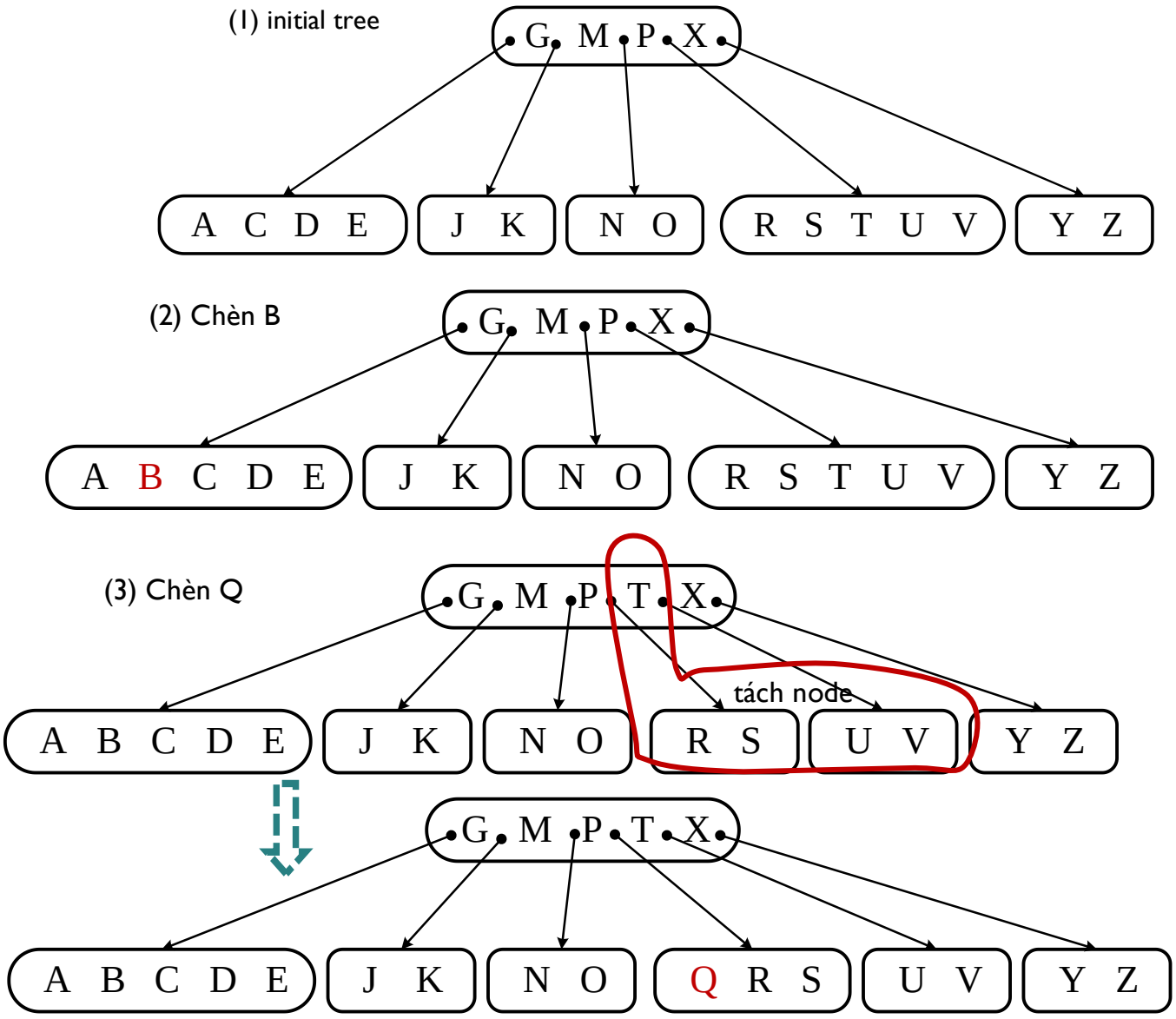
$1 \leq$ **số khóa** của node **root** ≤ 5

$3 \leq$ **số node con** của mỗi **node** (khác root và lá) ≤ 6

$2 \leq$ **số node con** của mỗi node **root** ≤ 6

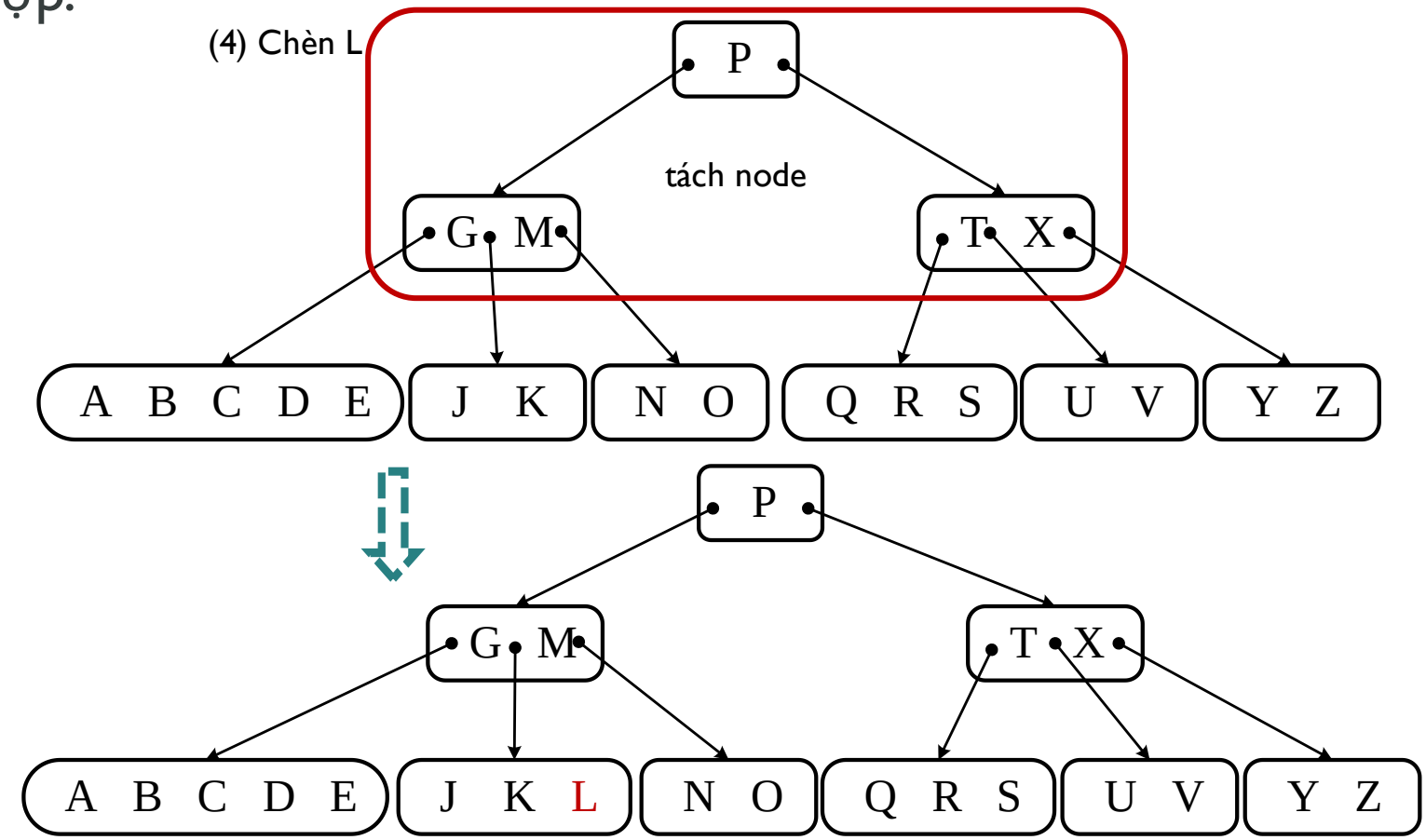
B-TREE-INSERTION: Ví dụ 2

- (1) Cây ban đầu.
- (2) Chèn B vào node lá thích hợp mà không cần thao tác tách nào cả.
- (3) Chèn Q. Việc chèn này sẽ tách node (R, S, T, U, V) thành hai node và khóa T sẽ lên node cha. Sau đó chèn Q vào vị trí thích hợp.



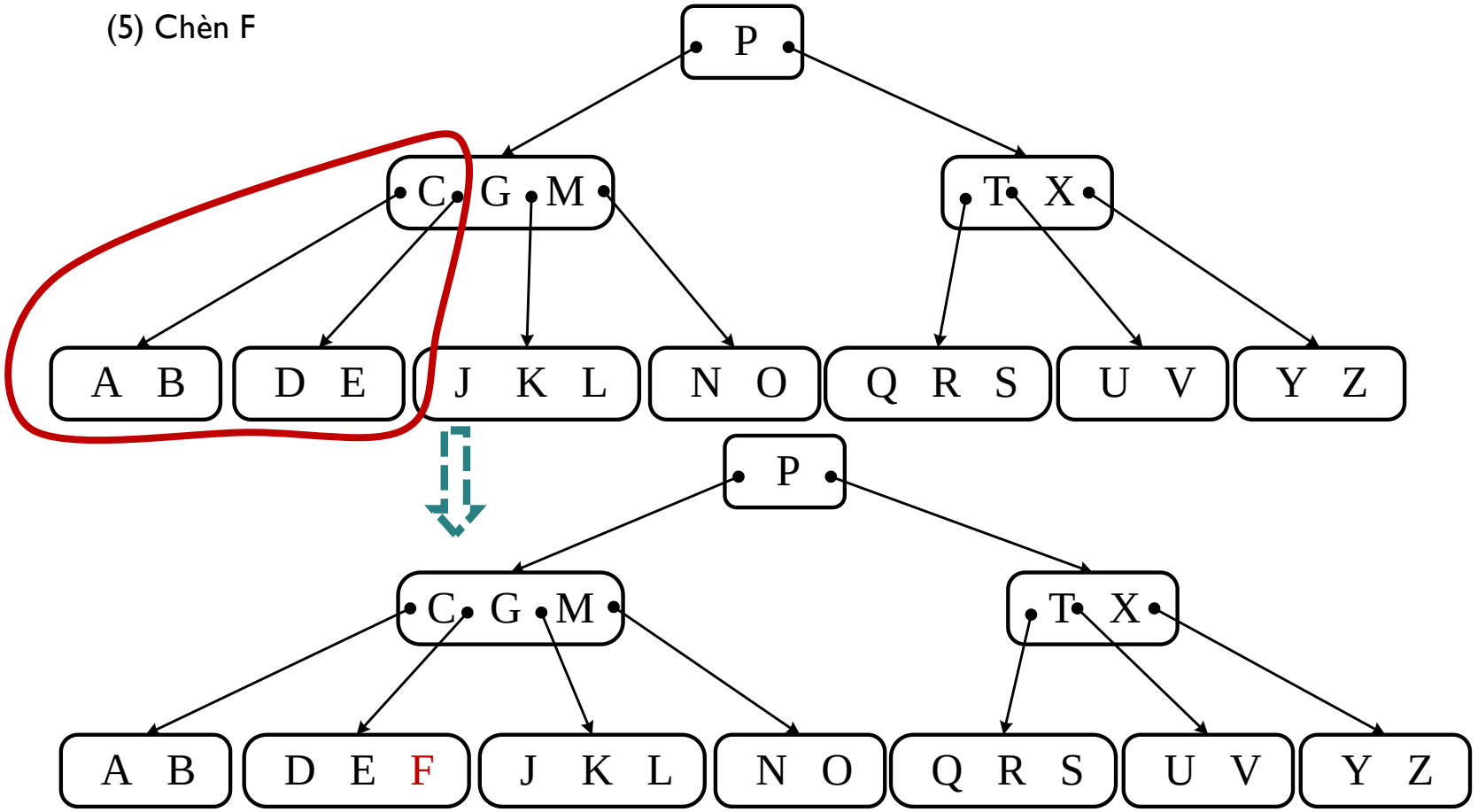
B-TREE-INSERTION: Ví dụ 1

- (4) Chèn L. Vì node gốc đầy, nên đầu tiên sẽ tách làm 2 node với khóa median là P, sau đó tiếp tục đi xuống để chèn L vào cây con thích hợp.



B-TREE-INSERTION: Ví dụ 1

- (5) Chèn F. Việc chèn này sẽ tách node (A, B, C, D, E) thành hai node và khóa C sẽ lên node cha. Sau đó chèn F vào vị trí thích hợp.



B-TREE-INSERTION: Ví dụ 2

Ví dụ 2: Tạo cây B-Tree với minimum degree t=3 từ dãy gồm 9 số nguyên: 5, 10, 15, 20, 25, 30, 35, 40 và 45. Các số được thêm vào theo thứ tự từ trái sang phải.

Bài làm:

Áp dụng vào bài, B-tree t=3 nên ta có:

$2 \leq$ **số khóa** của mỗi **node khác root** ≤ 5

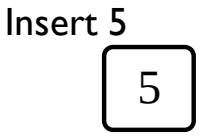
$1 \leq$ **số khóa** của node **root** ≤ 5

$3 \leq$ **số node con** của mỗi **node** (khác root và lá) ≤ 6

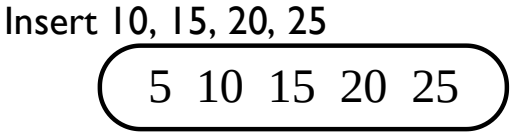
$2 \leq$ **số node con** của mỗi node **root** ≤ 6

B-TREE-INSERTION: Ví dụ 2

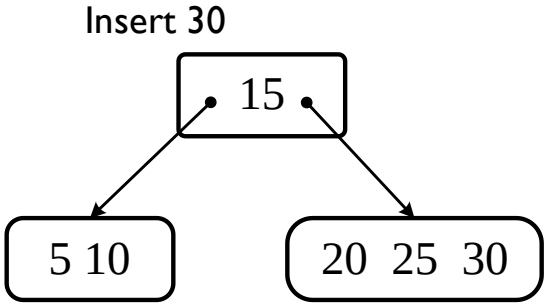
- Tạo root rỗng. Chèn 5 vào đầu tiên.



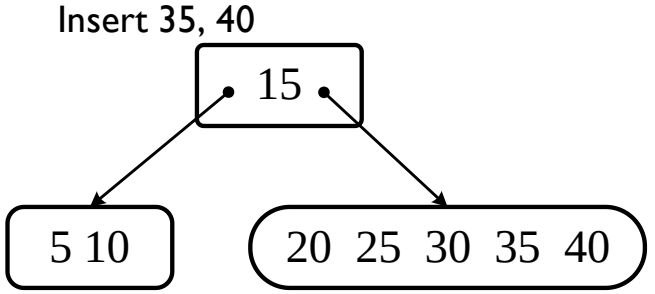
- Chèn 10, 15, 20, 25 vào node root.



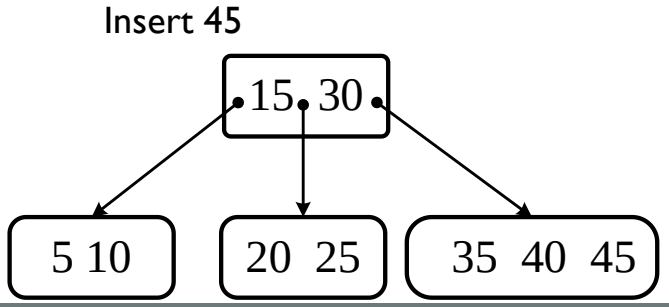
- Chèn 30. Vì node gốc đầy, nên đầu tiên sẽ tách làm 2 node với khóa median là 15, sau đó chèn 30 vào cây con thích hợp



- Chèn 35, 40 vào node lá thích hợp mà không cần thao tác tách nào cả.

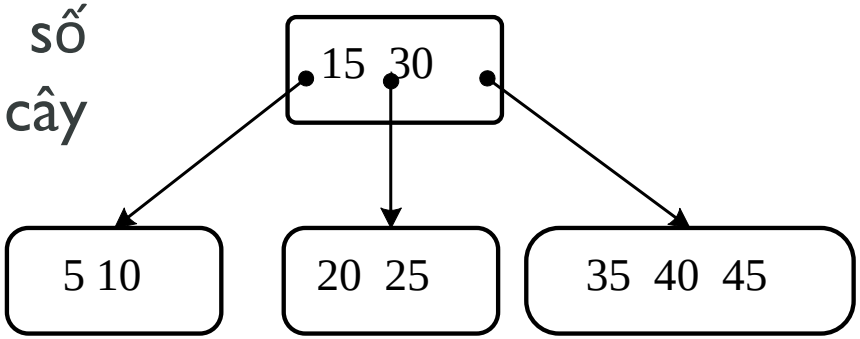


- Chèn 45. Việc chèn này sẽ tách node (20 25 30 35 40) thành hai node và khóa 30 sẽ lên node cha. Sau đó chèn 45 vào vị trí thích hợp.



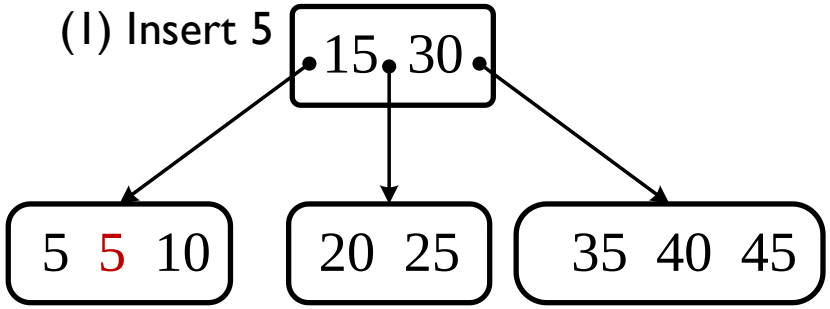
B-TREE-INSERTION: Ví dụ 3 (add duplicate keys)

Ví dụ 3: Thêm lần lượt 7 giá trị số nguyên: (5, 30, 30, 30, 30, 30, 30) vào cây B-tree có t=3 cho sẵn như hình bên:

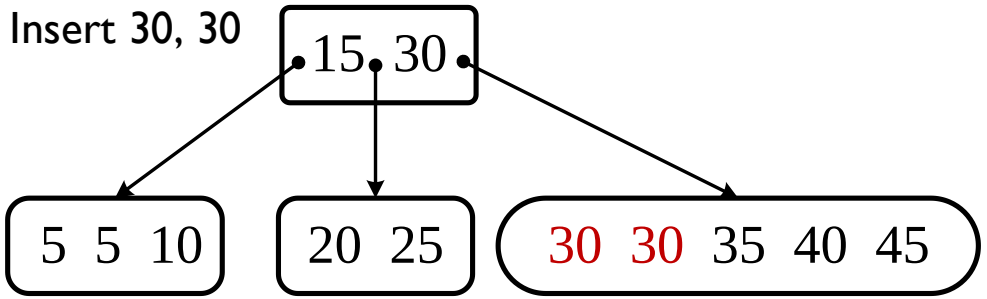


Bài làm:

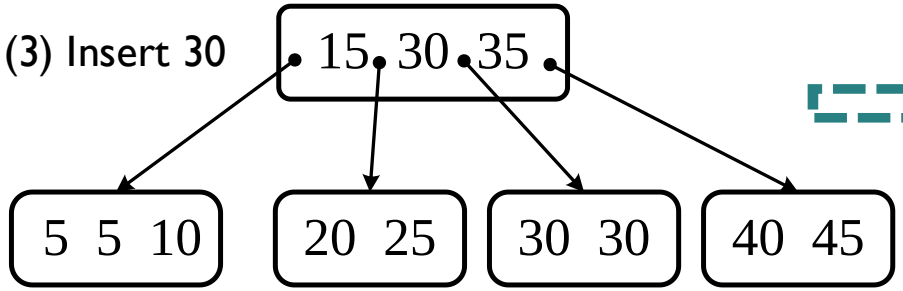
(1) Insert 5

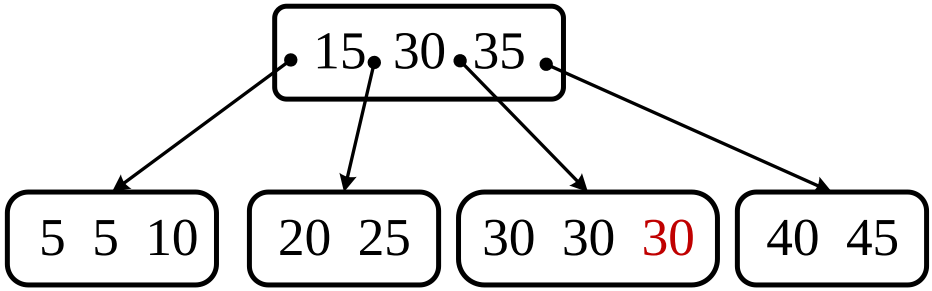


(2) Insert 30, 30

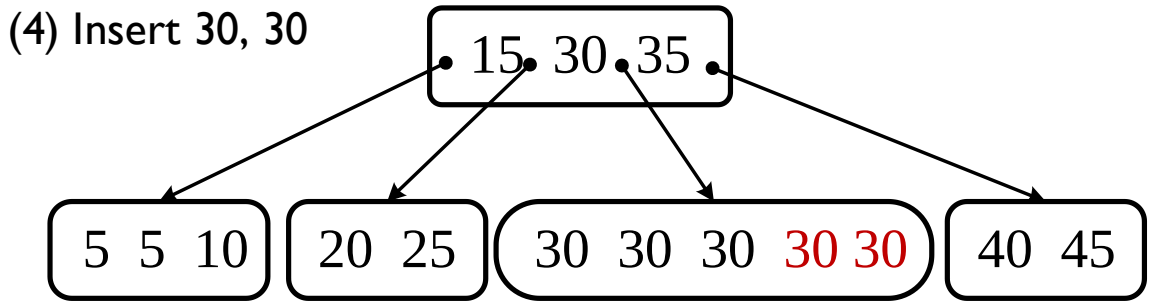


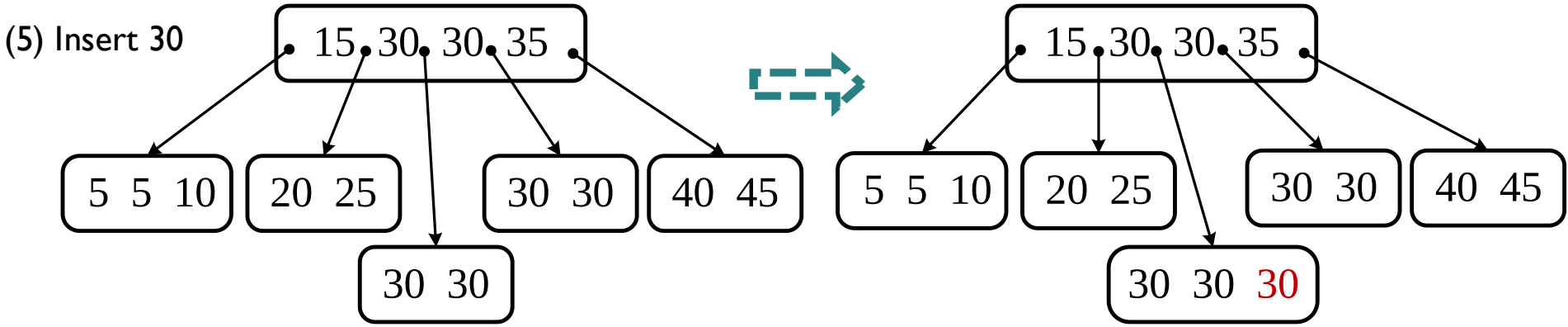
(3) Insert 30





B-TREE-INSERTION: Ví dụ 3 (add duplicate keys)

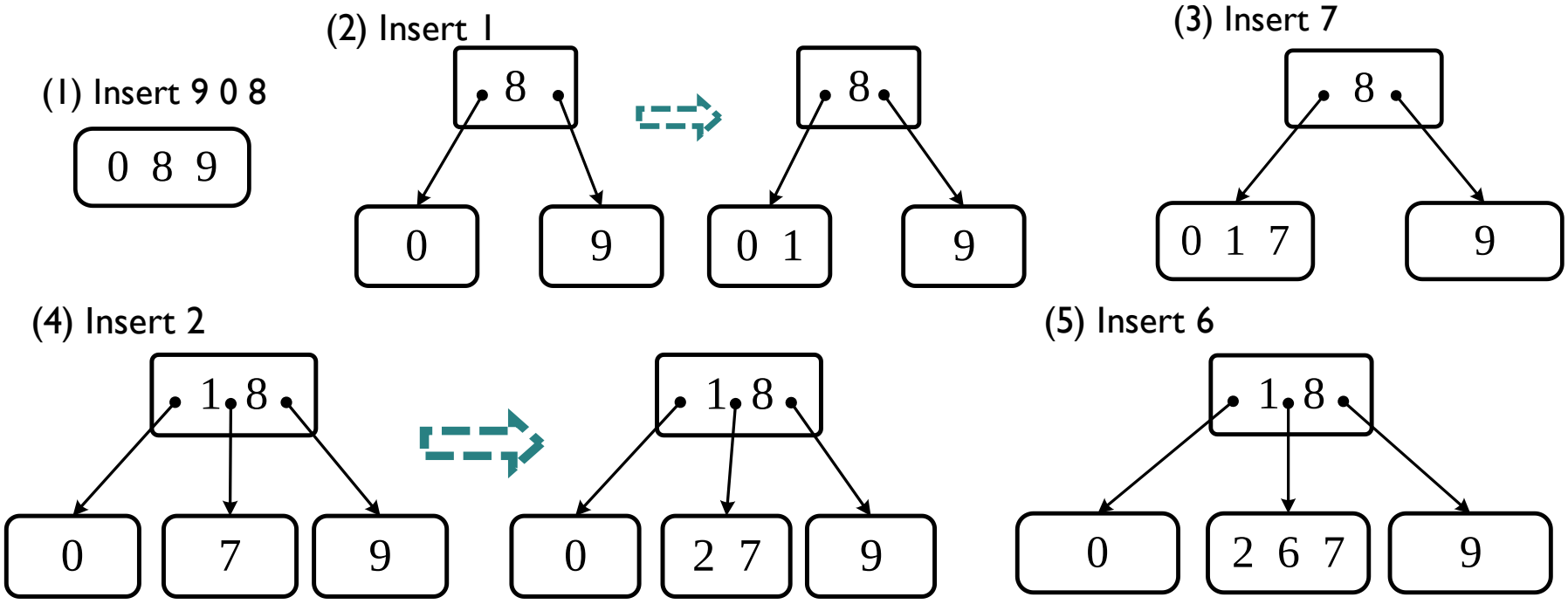




B-TREE-INSERTION:Ví dụ 4

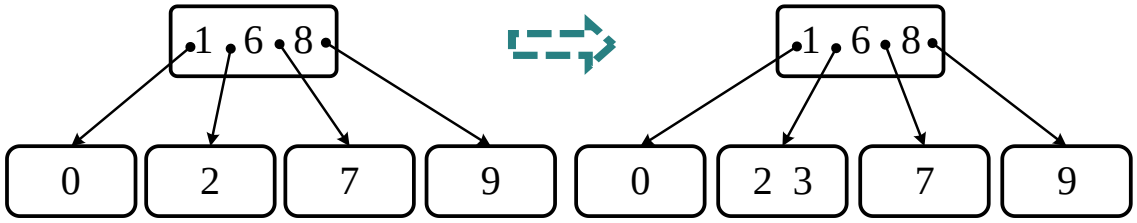
Ví dụ 4: Tạo cây B-Tree với minimum degree t=2 từ dãy gồm 23 số nguyên: 9, 0, 8, 1, 7, 2, 6, 3, 5, 4, 30, 20, 10, 12, 14, 15, 18, 35, 19, 40, 16, 13, 11. Các số được them vào theo thứ tự từ trái sang phải.

Bài làm: *Ghi chú quá trình tạo, số đóng trong dấu ngoặc là cần tách node: 9, 0, 8, (1), 7, (2), 6, (3), (5), (4), 30, 20 (10), 12, (14), (15), (18), 35, 19, 40, (16), (13), (11).*

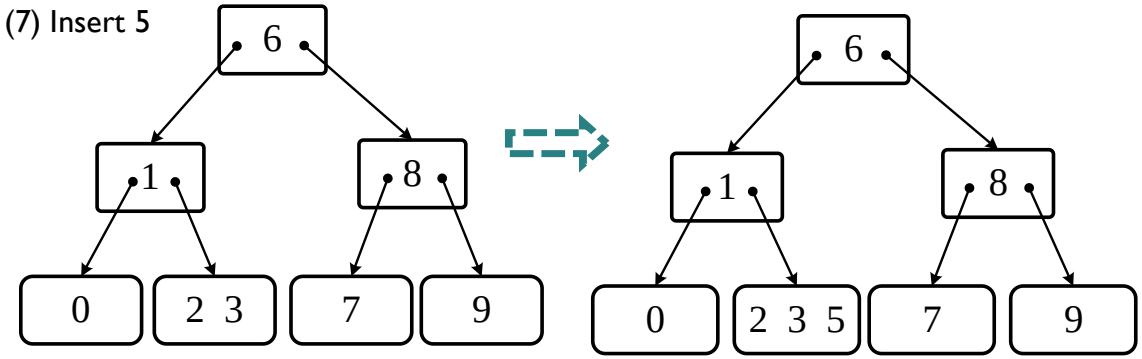


B-TREE-INSERTION:Ví dụ 4

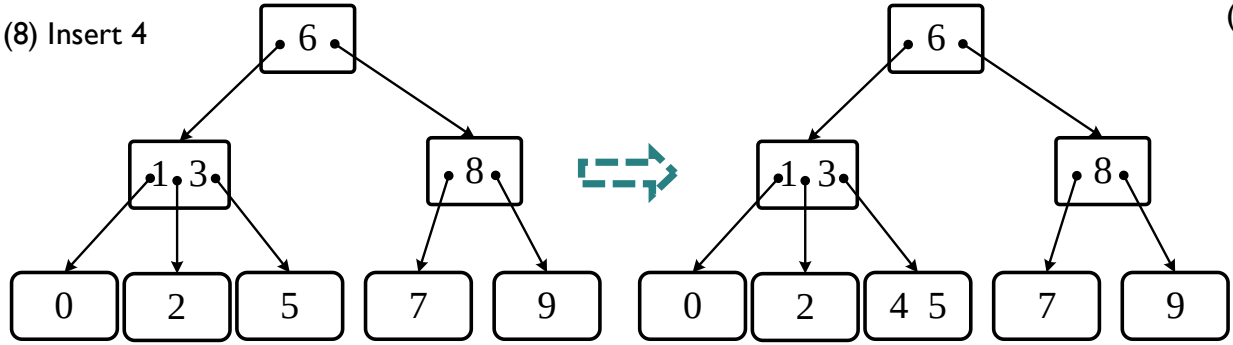
(6) Insert 3



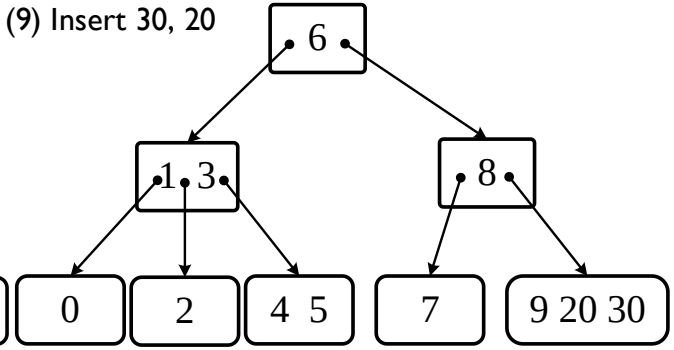
(7) Insert 5



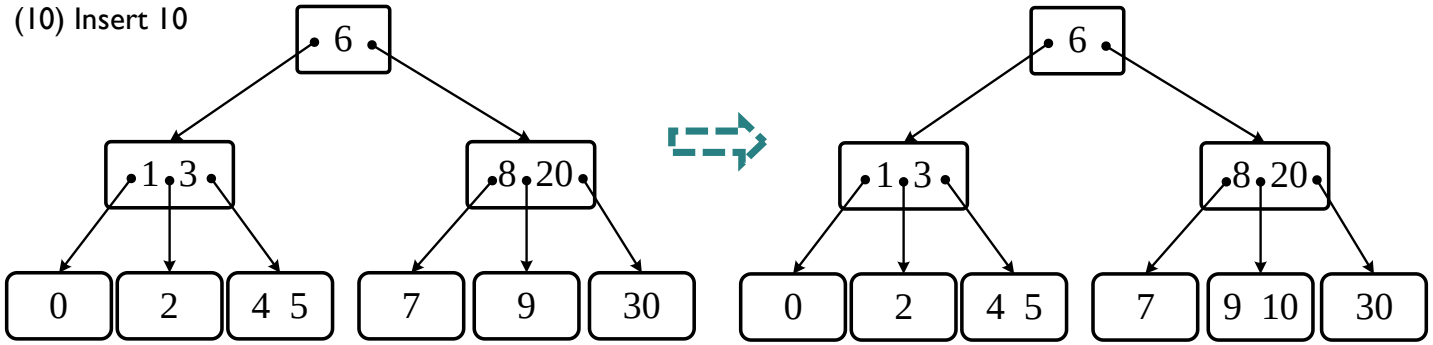
(8) Insert 4

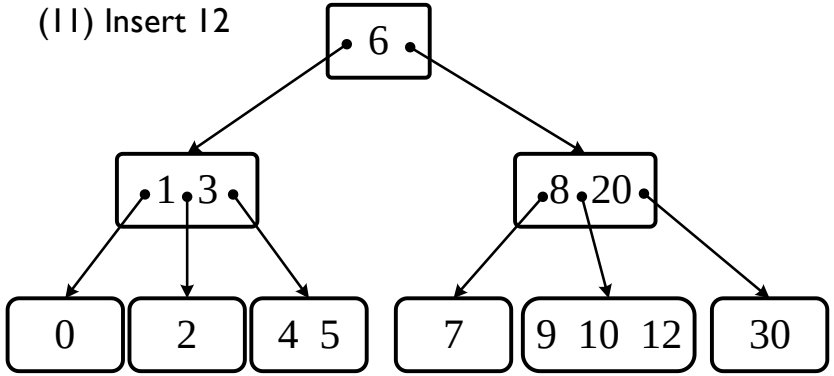


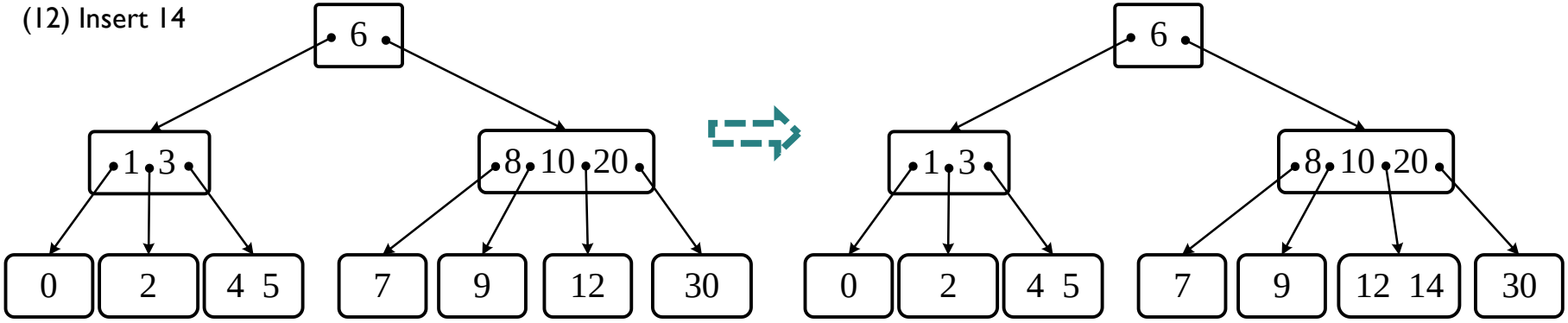
(9) Insert 30, 20



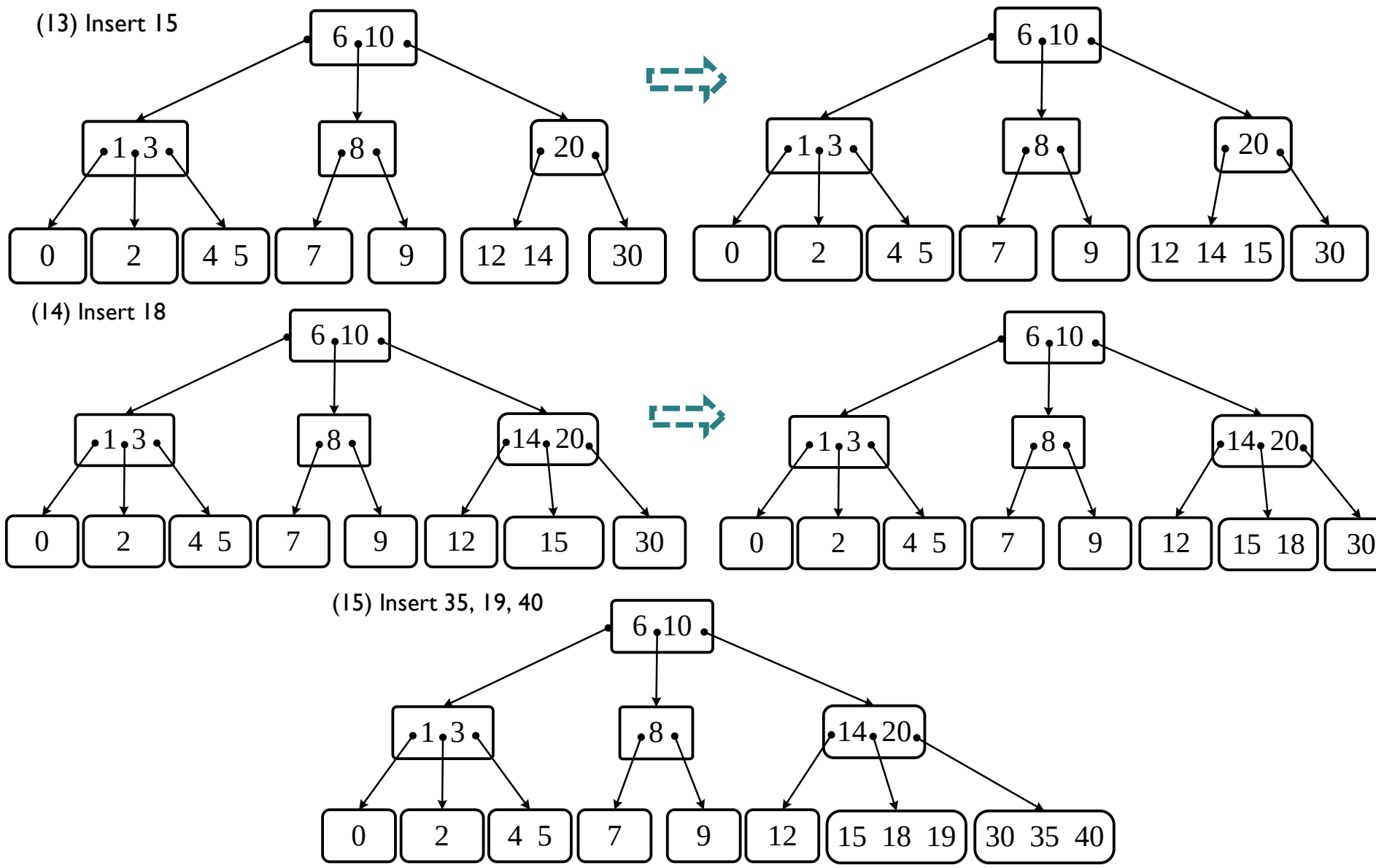
B-TREE-INSERTION:Ví dụ 4



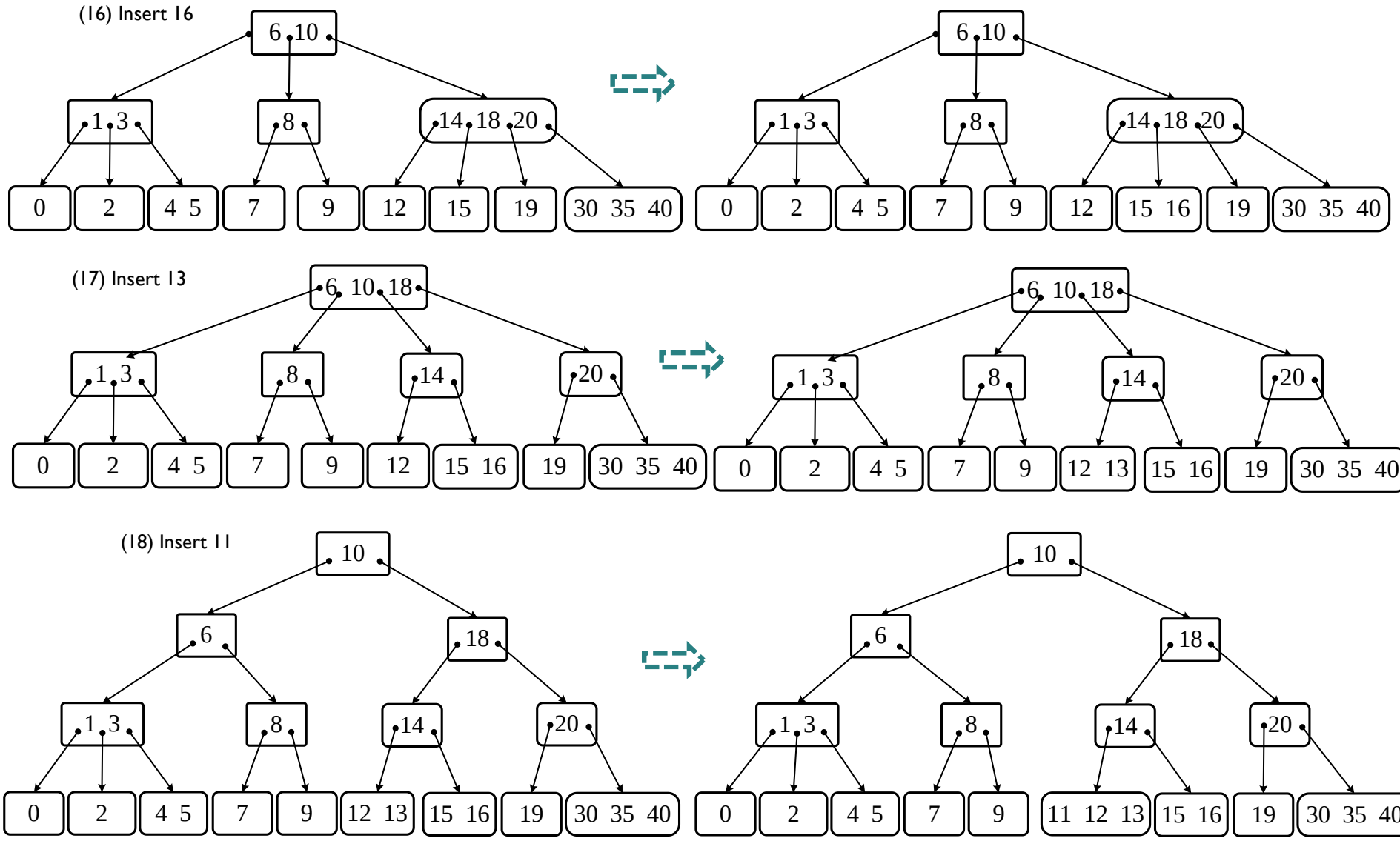




B-TREE-INSERTION:Ví dụ 4



B-TREE-INSERTION:Ví dụ 4



Nội dung

1. Giới thiệu
2. Định nghĩa
3. Các phép toán cơ bản
   i. B-TREE-TRAVERSE
   ii. B-TREE-SEARCH
   iii. B-TREE-CREATE
   iv. B-TREE-INSERTION
   v. B-TREE DELETION
4. Ứng dụng
5. Bài tập

Bài tập

Bài 1: Định nghĩa của cây B-tree.

Bài 2: Show the results of inserting the keys:

F, S, Q, K, C, L, H, T, V, W, M, R, N, P, A, B, X, Y, D, Z, E

in order into an empty B-tree. Only draw the configurations of the tree just before some node must split, and also draw the final configuration.

Bài 3: Cho B-tree bậc 5 gồm các khóa sau (chèn vào theo thứ tự): 3, 7, 9, 23, 45, 1, 5, 14, 25, 24, 13, 11, 8, 19, 4, 31, 35, 56, 2, 6,12. Sau đó thực hiện xóa khóa: 4, 5, 7, 3, 14.

Bài 4: Khởi tạo B Tree bậc 7 với các thao tác Insert: 34, 12, 55, 21, 6, 84, 5, 33, 15, 74, 54, 28, 10, 19. Sau đó thực hiện các xóa lần lượt 11, 15, 6, 98, 34, 5.

Bài 5: Suppose that we were to implement B-TREE-SEARCH to use binary search rather than linear search within each node. Show that this change makes the CPU time required O.lg n/, independently of how t might be chosen as a function of n.

Chúc các em học tốt!