



# B-Tree



# Nội dung

- Nhu cầu (Motivation)
- Định nghĩa (Definition)
- Các thao tác (Operations)
  - Searching
  - Insertion
  - Deletion
- Ứng dụng
- Bài tập



# Motivation

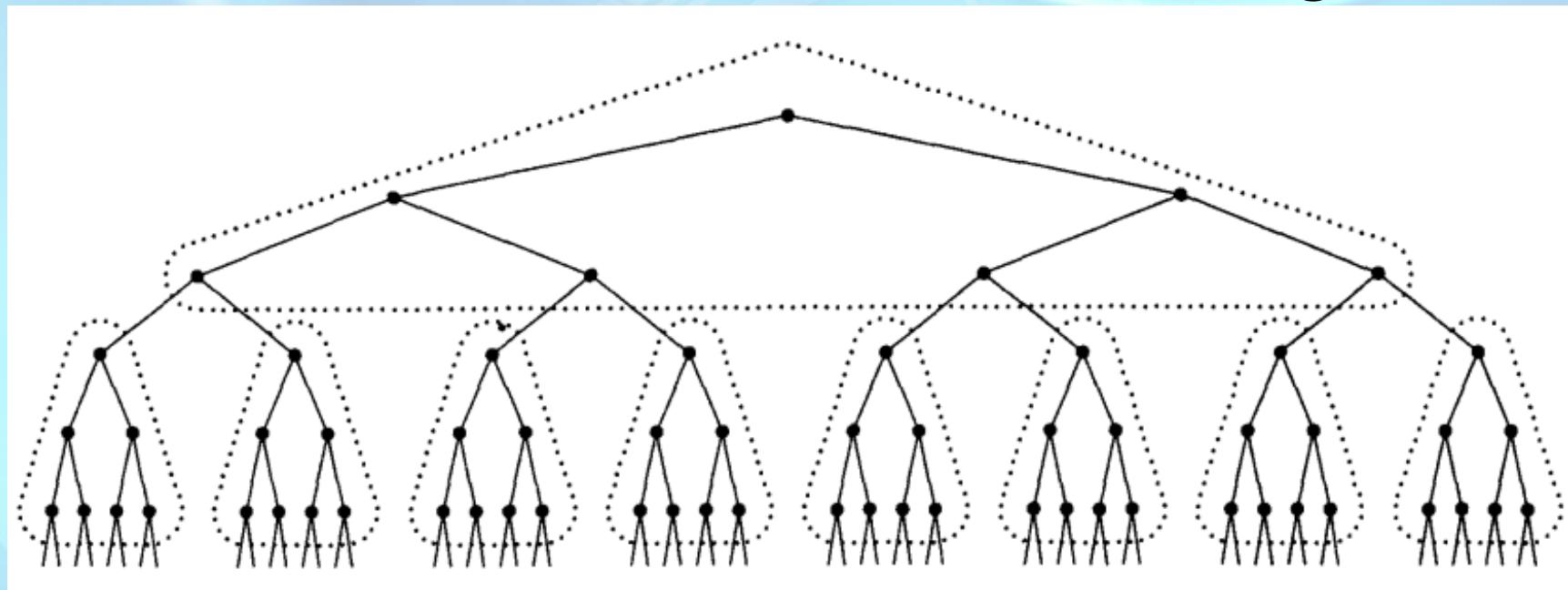
- Độ phức tạp <  $O(\log(n))$  cho 3 thao thác
  - Searching
  - Insertion
  - Deletion
- Tối ưu trên đĩa cứng ?
- → Các nhược điểm của BST?



# Motivation

- Gom nhiều node thành một “super node”

- Knuth, D. E. (2007).  
The Art of Computer Programming: Sorting and searching. Vol. 3,





# Definition

- BAYER, R., & MCCREIGHT, E. (1972). Organization and Maintenance of Large Ordered Indexes. *Acta Informatica*, 1, 173-189.
- Cho số tự nhiên  $k > 0$ , B-Trees bậc m với  $m = 2*k+1$  là một cây thỏa mãn các tính chất:
  - i. Tất cả node lá nằm trên cùng một mức
  - ii. Tất cả các node, trừ node gốc và node lá, có **\*tối thiểu\***  $k+1$  node con.
  - iii. Tất cả các node có **\*tối đa\*** m con
  - iv. Tất cả các node, trừ node gốc, có từ k cho đến  $m - 1$  khóa (keys). Node gốc có từ 1 đến  $m-1$  khóa.
  - v. Một node không phải lá và có  $n$  khóa thì phải có  $n+1$  node con.



# Definition

- Bố trí khóa trong một node:
  - Tất cả các khóa  $k_0, k_1, \dots, k_{n-1}$  trong node được sắp thứ tự tăng dần. Tương ứng với n khóa này là n+1 con trỏ  $p_0, p_1, \dots, p_n$  trỏ đến các node con.
- Gọi  $K(p_i)$  là tập các khóa trong node trỏ tới bởi con trỏ  $p_i$ , B-Trees đảm bảo:

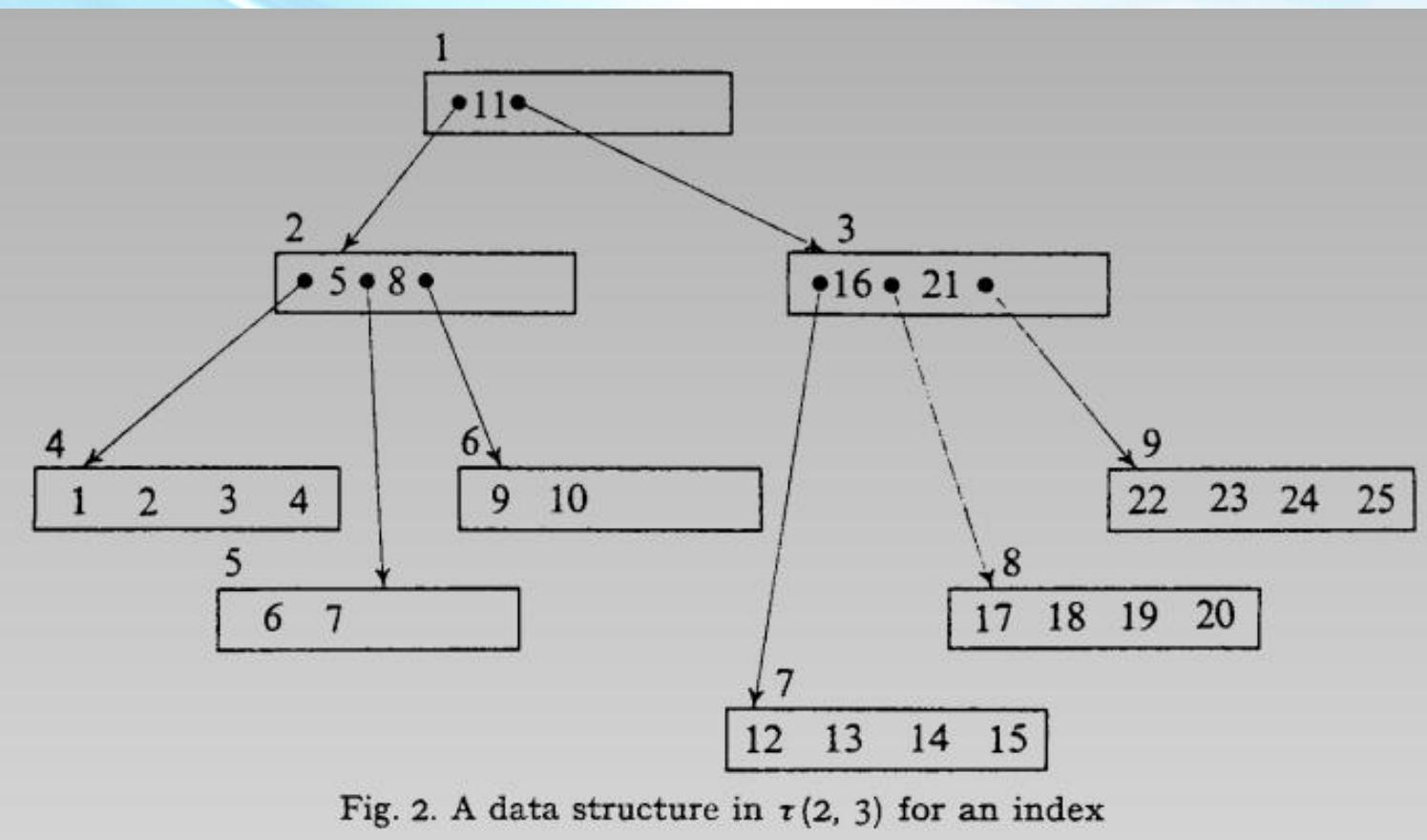
$$\forall y \in K(p_0), y < k_0$$

$$\forall y \in K(p_i), k_{i-1} < y < k_i; i = 1, 2, \dots, n-1$$

$$\forall y \in K(p_n), k_{n-1} < y$$



# Definition



- B-Tree bậc 5, chiều cao là 3 (trong ví dụ của Bayer và McCreight)



# Operations - class definition

- Cấu trúc dữ liệu cho B-Tree

```
class BTTree_node{
    vector<my_data> keys;
    vector<BTTree_node*> children;
    bool is_leaf(){ return children.empty(); }
    friend class BTTree;
};

class BTTree{
public:
    BTTree(int initializing_m){
        if (initializing_m %2 == 0) {
            initializing_m = initializing_m/2 +1;
        }
        this->_m = initializing_m;
        root = NULL;
    }
    bool empty(){
        return root == NULL;
    }
private:
    BTTree_node *root;
    int _m;
```



# Operations - searching

- INPUT

- BTREE\_NODE \*root
  - my\_data x

- OUTPUT

- true nếu tìm thấy khóa x trong B-Tree
  - false nếu không thấy

- Bước 2 cần sử dụng binary search

Nếu root == NULL, return false

Tìm i là số lượng khóa nhỏ hơn x

Nếu i == keys.size() và keys[i] == x, return true;

Nếu root là lá, return false

Root = children[i], quay lại bước (2)



# Operations - searching

- Minh họa cài đặt thao tác tìm kiếm
  - Kết quả trả về của hàm?
  - Hàm std::lower\_bound
- Hàm này phức tạp hơn thuật toán mô tả!

```
pair<int, vector<BTree_node*>>
search(BTree_node*root, my_data key){
    vector<BTree_node*> path;
    auto p = root;
    while(p != NULL){
        path.push_back(p);
        auto i = lower_bound(p->keys.begin(), p-
>keys.end(), key);
        if (i != p->keys.end() && *i == key){
            return {i - p->keys.begin(), path};
            //FOUND
        } else {
            if (p->is_leaf() == 0) break;
            p = p->children[i - p->keys.begin()];
        }
    }
    return {-1, path};
}
```



- Tìm giá trị 7

- Root = {11}, i = 0
- Root = {5,8}, i = 1
- Root = {6,7}, i = 1, khóa keys[1] có giá trị 7 == 7  
==> Tìm thấy

- Tìm giá trị 14.5

- Root = {11}, i = 1
- Root = {16,21}, i = 0
- Root = {12,13,14,15}, i = 3.  
Khóa keys[3] có giá trị 15  
!= 14.5 ==> Không tìm thấy

.....{11}.....  
.....{5,8}.....  
.....{16,21}.....  
{1,2,3,4} {6,7} {9,10} {12,13,14,15}  
{17,18,19,20} {22,25}

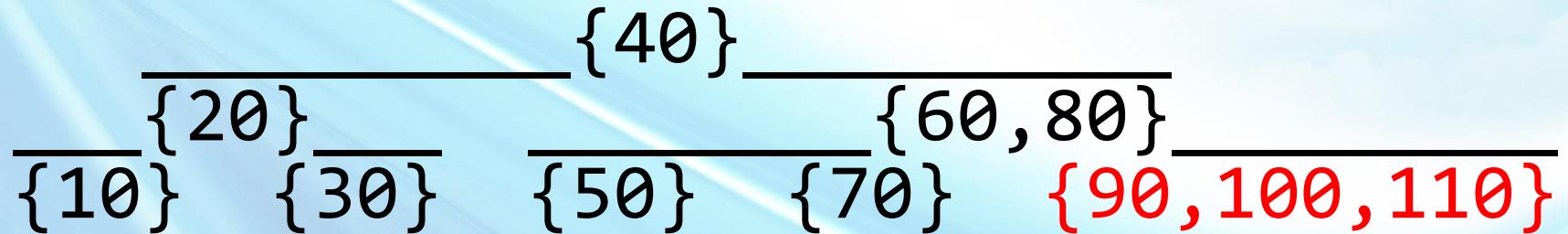


# Operations - node splitting

- Cho node N có cha là P và hiện tại N có **nhiều hơn m-1** khóa
  - **Di chuyển** khóa ở giữa của N lên P
  - Tạo node N' mới, **di chuyển** phân nửa số node và phân nửa số cây con còn lại sang N'
  - Thêm N' vào danh sách cây con của P
- Thao tác splitting có thể lan truyền (propagate) đến nhiều mức phía trên



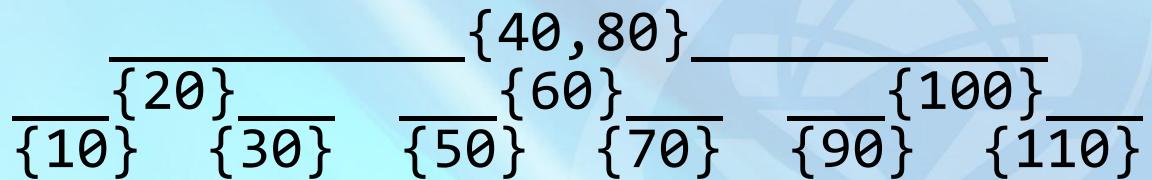
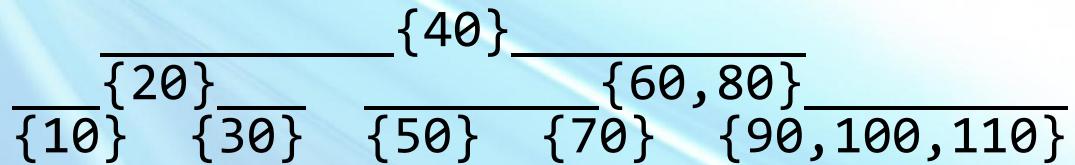
## Operations - node splitting



- Cho B-Tree bậc 3, k = 1.
- Node in đỏ có 3 khóa



# Operation – node splitting



- Sau khi split, 100 được đưa lên node cha.
- Node cha lại có quá nhiều
- 80 được đưa lên node cha
- B-Tree trở về đúng định nghĩa
- Nếu ta split node gốc??



# Operation - node splitting

- Hàm kiểm tra và split node i có cha là p

- Trước ng hợp khô ng cần split?

```
BTree_node* split(BTree_node *i,  
BTree_node*p){  
    if(i->keys.size() < this->_m){//No need  
        to split  
        return NULL;  
    }  
    //Create new BTree_node  
    auto n = new BTree_node;  
    auto move_up = i->keys[i->keys.size()/2];  
    move(i->keys.begin() + i->keys.size()/2 +  
    1, i->keys.end(), back_inserter(n-  
    >keys));  
    if( !i->is_leaf() ) {  
        move(i->children.begin() + i-  
        >keys.size()/2+1, i->children.end(),  
        back_inserter(n->children));  
    }  
    if( !i->is_leaf() ) i->children.resize(i-  
    >keys.size()/2+1);  
}
```



# Operation - node splitting

- INPUT:

- Đường đi từ gốc đến một node

- TASK:

- Tiến hành split node đó và lan truyền ngược nếu cần

```
BTree_node* split(vector<BTree_node*> &path){  
    auto i = path.end() - 1;  
    BTree_node*n;  
    int move_up;  
    while(i != path.begin() - 1){  
        auto p = i - 1;  
        if (i == path.begin()){  
            auto x = split(*i, NULL);  
            if (x == NULL) return path.front();  
            else return x;  
        } else {  
            auto x = split(*i, *p);  
            if (x == NULL) return path.front();  
            else {  
                i = p; // Move up the path  
            }  
        }  
    }  
}
```

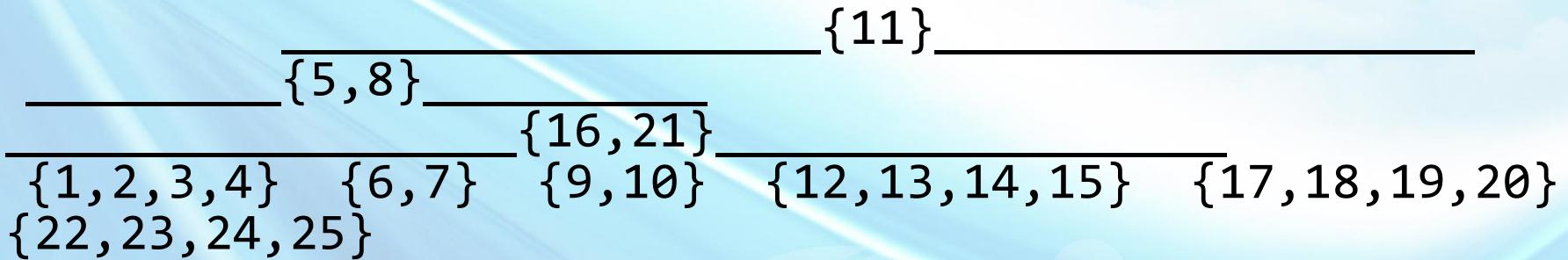


# Operation - insertion

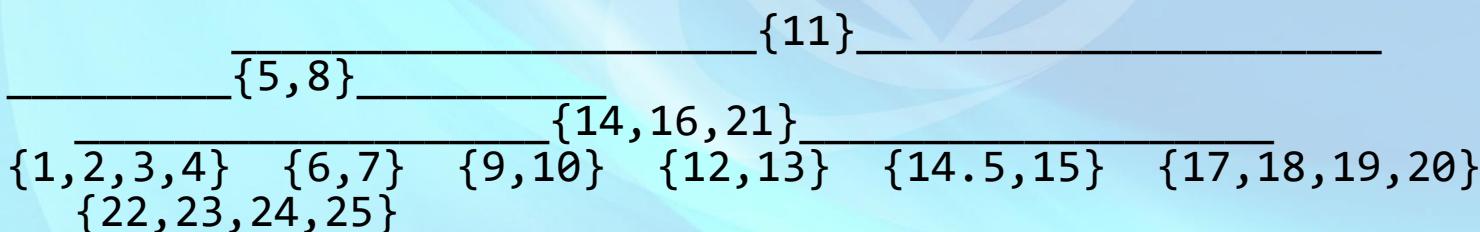
- Thao tác chèn luôn tiến hành tại lá
  - Tại sao?
  - Lá nào sẽ được chèn?
- Sau khi chèn
  - Split node lá nếu cần
  - Lan truyền lên mức trên nếu cần
- Đảm bảo 5 yêu cầu của B-Tree
- Tận dụng các hàm đã viết



# Operation - Insertion



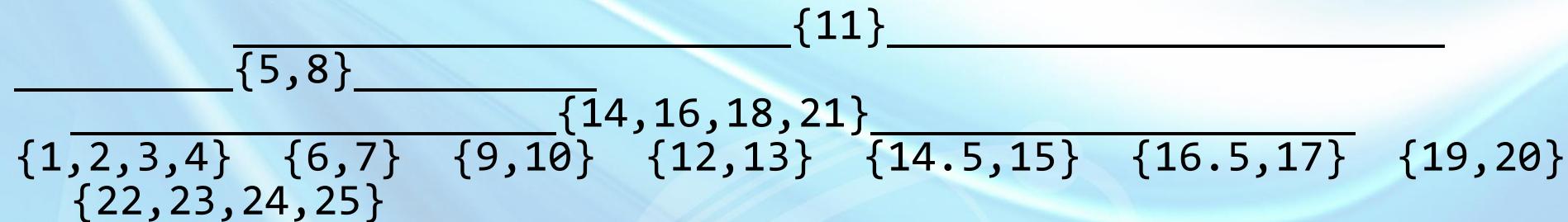
- B-Tree bậc 5
- Chèn thêm khóa 14.5



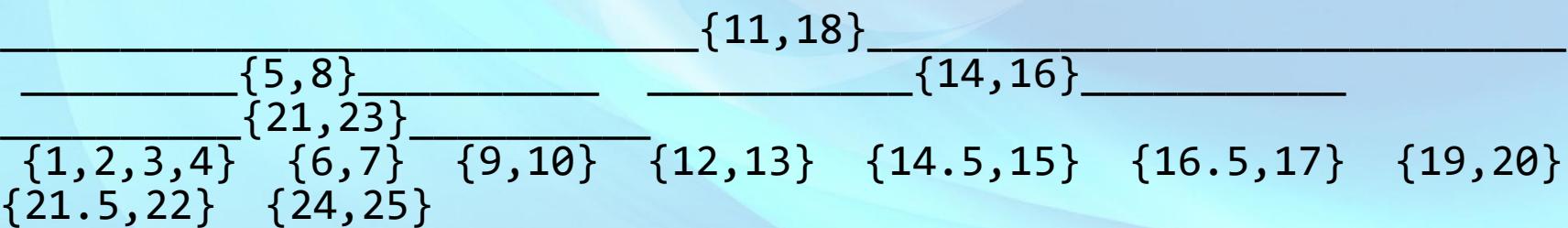


# Operation - Insertion

- Chèn thêm khóa 16.5



- Chèn thêm khóa 21.5





# Operations - proactive splitting

- Proactive splitting - preemptive splitting
  - Khi tìm node để thêm khóa mới, nếu gặp một node có vừa đủ  $m-1$  khóa thì split luôn.
- Triệt tiêu lan truyền ngược
- Không tận dụng được hàm search
- Ưu nhược điểm khác ??



# Operation - Catenation

- Node P có 02 người con N' và N''
  - lần lượt ở vị trí i và i + 1
- N' có n' khóa
- N'' có n'' khóa

N'	$k'_0$	...	$k'_{n-1}$	
P.p <sub>i</sub>	$p'_0$	$p'_1$	...	$p'_n$

P	...	$k_{i-1}$	$k_i$	$k_{i+1}$	...	
	...	$p_{i-1}$	$p_i$	$p_{i+1}$	$p_{i+2}$	...

N''	$k''_0$	...	$k''_{n''-1}$	
P.p <sub>i+1</sub>	$p''_0$	$p''_1$	...	$p''_{n''}$



# Operations - Catenation

- Ta có thể gộp  $N'$  và  $N''$  thành một node
- P mất đi một khóa và một cây con
  - Nếu P là node gốc??
- Các chỉ số được đánh lại sau khi gộp

P	...	$k_{i-1}$	$k_{i+1}$	...		
	...	$p_{i-1}$	$p_i$	$p_{i+2}$	...	

$N'$ ( $p_i$ )	$k'_0$	...	$k'_{n'}$	$k_i$	$k''_0$	...	$k''_{n''}$	
	$p'_0$	$p'_1$	...	$p'_{n'+1}$	$p''_0$	$p''_1$	...	$p''_{n''+1}$



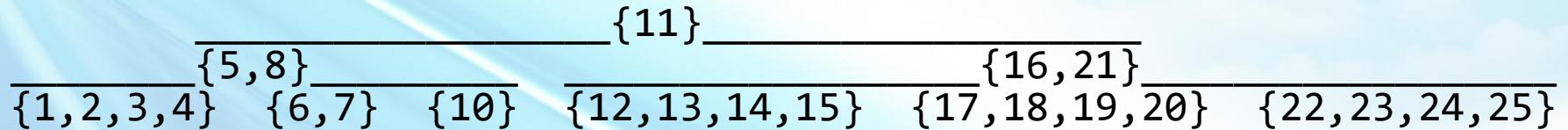
# Operations - Catenation

- Hàm *find\_cat\_partner* tìm node anh em có thể gộp
- Hàm *catenation* gộp node con ở vị trí icat với node con icat+1

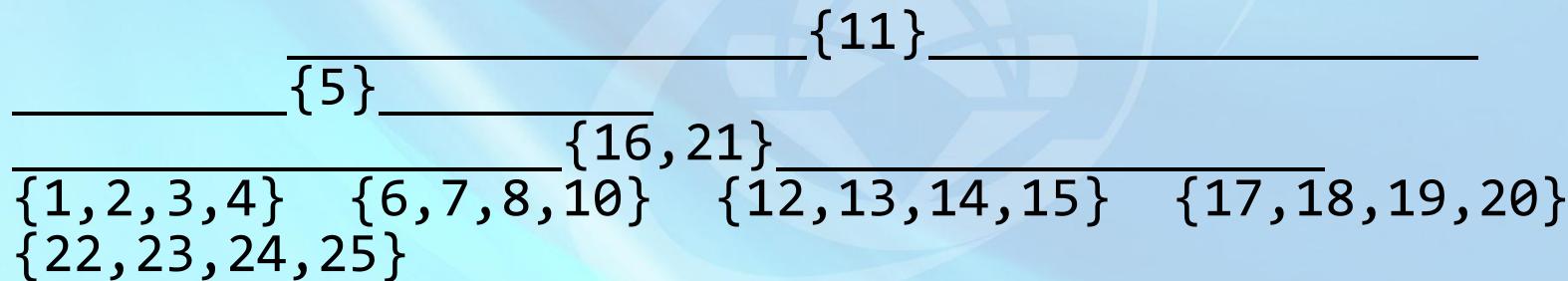
```
int find_cat_partner(BTree_node* p, int ip){  
    int s = p->children[ip]->keys.size();  
    if (ip > 0 && p->children[ip - 1]->keys.size() + s <  
        this->_m - 1)  
        return ip - 1;  
    if (ip < p->children.size()-1 && p->children[ip+1]-  
        >keys.size() + s < this->_m - 1)  
        return ip;  
    return -1;  
}  
my_data catenation(BTree_node* p, int icat){  
    auto moved_key = p->keys[icat];  
    p->children[icat]->keys.emplace_back(p->keys[icat]);  
    move(p->children[icat+1]->keys.begin()  
        , p->children[icat+1]->keys.end()  
        , back_inserter(p->children[icat]->keys));  
    move(p->children[icat+1]->children.begin()  
        , p->children[icat+1]->children.end()  
        , back_inserter(p->children[icat]->children));  
    delete(p->children[icat+1]);  
    p->keys.erase(p->keys.begin() + icat);  
    p->children.erase(p->children.begin() + icat + 1);  
    return moved_key;  
}
```



# Operations - Catenation



- B-Tree bậc 5 có node {10} quá ít khóa
- Gộp node {6,7} và {10} thì node cha bị mất khóa 8



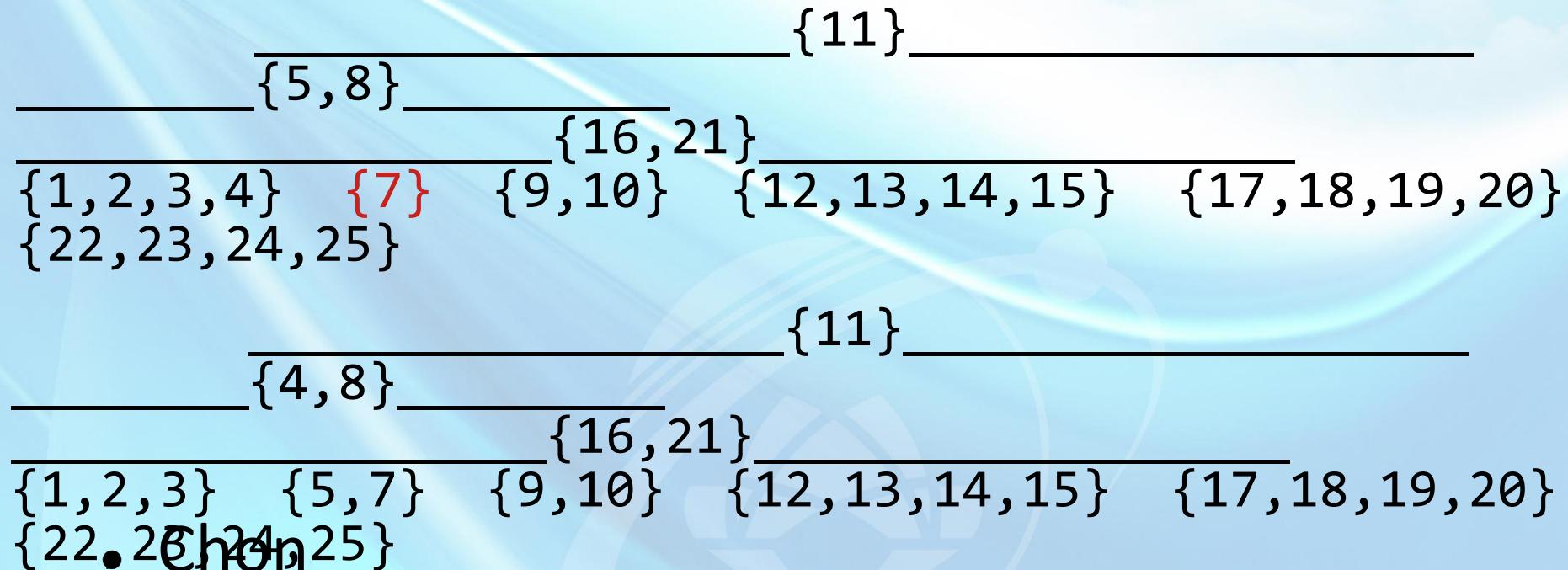


# Operations - Underflow

- P có 02 người con N' và N''
  - Một trong 02 node có ít hơn k khóa
  - Tổng số khóa trong 2 node lại  $> 2*k$
- Node to sẽ nhường vài khóa và con cho node nhỏ => underflow
  - 1) Catenate N' và N'' vào thành một node to
  - 2) Split node to này thành 02 node đều nhau hơn



# Operations - Underflow



- underflow{1,2,3,4},{7} hoặc
- catenate{7}+{9, 10}
- Tại sao underflow thì khóa 4 lại nằm ở node trên?



# Operation - deletion

- Xóa bắt đầu từ lá
- Nếu khóa không nằm ở lá:
  - kiểm một khóa thế mạng ở lá
  - Swap
  - Xóa khóa ở lá
- Kiểm tra nếu cần thực hiện underflow và Catenation.



# Operation - deletion

- Input: Kết quả tìm kiếm node cần xóa
- Tìm phần tử thế mạng và hoán đổi.

```
my_data find_replace_and_swap(int &idx,  
vector<BTree_node*>&path){  
    auto x = path.back();  
    auto i = path.back()->children[idx];  
    while(!i->is_leaf()){  
        path.push_back(i);  
        i = i->children.back();  
    }  
    path.push_back(i);  
    auto back = i->keys.back();  
    swap(x->keys[idx], i->keys.back());  
    idx = i->keys.size() - 1;  
    return back;  
}
```



# Operation - deletion

- Kiểm tra và thực hiện underflow, catenation nếu cần

```
BTree_node*  
check_for_underflow_and_cat(my_data  
remove_key, vector<BTree_node*>  
path){  
    auto i = path.end() - 1;  
    auto p = i - 1;  
    while(i != path.begin()){  
        int ip = lower_bound((*p)-  
        >keys.begin(), (*p)->keys.end(),  
        remove_key)  
        - (*p)->keys.begin();  
        int iu = underflow_check(*p,  
        ip);  
        if (iu != -1){  
            //If we can underflow, we do  
            underflow first  
            underflow(*p, iu); return  
        }  
        i--;  
    }  
}
```



# Ứng dụng

- B-Tree có nhiều biến thể và cải tiến
  - B<sup>+</sup> -Tree
  - B<sup>\*</sup> -Tree
- Quản lý dữ liệu trên đĩa cứng, dữ liệu lớn



# Ứng dụng

- File system - Hệ thống quản lý file trên đĩa cứng
  - Danh sách các block còn trống
  - File x đang nằm ở block số mấy?
- Windows:
  - NTFS, FAT32,...
- MacOS
  - HFS+
- Linux
  - Btrfs, Ext, xFS
- Other:
  - HFS, Reiser4, HAMMER, ...



# Ứng dụng

- Database
  - Indexing - quản lý tập các khóa và dữ liệu
  - Rank, between, v.v...
- MySQL
- MariaDB
- MS-SQL
- MongoDB



## Bài tập (1)

1. Theo định nghĩa thì không thể có B-Tree bậc 1 và  
2. Tại sao Bayer và McCreight không định nghĩa  
loại B-Tree này??
2. Viết hàm thêm node trong B-Tree, không dùng  
chiến lược proactive splitting (tham khảo hàm  
search và split đã cho)
3. Viết hàm xóa node trong B-Tree (tham khảo hàm  
underflow, catenation đã cho)
4. (\*) Tìm công thức tính chiều cao tối đa của B-Tree  
bậc m có N khóa
5. Tìm công thức tính chiều cao tối thiểu của B-Tree  
bậc m có N khóa



## Bài tập (2)

1. Vẽ cây B-Tree bậc 5 khi lần lượt thêm các số từ 1-25.
  1. Không dùng proactive splitting?
  2. Dùng proactive splitting?
2. (\*) Lần lượt thêm các khóa có giá trị 1,2, ..., n vào B-Tree. Tìm công thức tính số node của B-Tree trên theo  $n$
3. Cho một danh sách các khóa. Vẽ B-Tree ít node nhất có thể chứa tất cả các khóa này
4. Cho một danh sách các khóa. Vẽ B-Tree nhiều node nhất có thể chứa tất cả các khóa này
5. Cho sẵn một B-Tree, tìm bậc và thứ tự các khóa



## Bài tập (3)

1. Viết hàm đếm số node trong B-Tree
2. Viết hàm tạo ra một B-Tree bản sao, có cấu trúc y chang một B-Tree cho trước
3. Viết hàm xuất các khóa trong B-Tree theo thứ tự giảm dần
4. Viết hàm tìm khóa có giá trị lớn nhất mà nhỏ hơn x
5. Viết hàm tìm khóa có giá trị gần với x nhất
6. (\*) Viết hàm tìm số lượng khóa nhỏ hơn x (thứ hạng của x)
  1. Muốn thực hiện thao tác tìm thứ hạng với độ phức tạp  $O(\log(n))$  cẩn thêm thông tin vào trong Btree\_node