






SEGMENT TREE

NHÓM 7

- 19522323 - Hồ Hải Thủy
 - 19522363 - Nguyễn Mạnh Toàn
 - 17520475 - Lê Trung Hiếu
- 

Common Data Structure Operations

Data Structure	Time Complexity								Space Complexity
	Average				Worst				Worst
	Access	Search	Insertion	Deletion	Access	Search	Insertion	Deletion	
Array	$\theta(1)$	$\theta(n)$	$\theta(n)$	$\theta(n)$	$\theta(1)$	$\theta(n)$	$\theta(n)$	$\theta(n)$	$\theta(n)$
Stack	$\theta(n)$	$\theta(n)$	$\theta(1)$	$\theta(1)$	$\theta(n)$	$\theta(n)$	$\theta(1)$	$\theta(1)$	$\theta(n)$
Queue	$\theta(n)$	$\theta(n)$	$\theta(1)$	$\theta(1)$	$\theta(n)$	$\theta(n)$	$\theta(1)$	$\theta(1)$	$\theta(n)$
Singly-Linked List	$\theta(n)$	$\theta(n)$	$\theta(1)$	$\theta(1)$	$\theta(n)$	$\theta(n)$	$\theta(1)$	$\theta(1)$	$\theta(n)$
Doubly-Linked List	$\theta(n)$	$\theta(n)$	$\theta(1)$	$\theta(1)$	$\theta(n)$	$\theta(n)$	$\theta(1)$	$\theta(1)$	$\theta(n)$
Skip List	$\theta(\log(n))$	$\theta(\log(n))$	$\theta(\log(n))$	$\theta(\log(n))$	$\theta(n)$	$\theta(n)$	$\theta(n)$	$\theta(n)$	$\theta(n \log(n))$
Hash Table	N/A	$\theta(1)$	$\theta(1)$	$\theta(1)$	N/A	$\theta(n)$	$\theta(n)$	$\theta(n)$	$\theta(n)$
Binary Search Tree	$\theta(\log(n))$	$\theta(\log(n))$	$\theta(\log(n))$	$\theta(\log(n))$	$\theta(n)$	$\theta(n)$	$\theta(n)$	$\theta(n)$	$\theta(n)$
Cartesian Tree	N/A	$\theta(\log(n))$	$\theta(\log(n))$	$\theta(\log(n))$	N/A	$\theta(n)$	$\theta(n)$	$\theta(n)$	$\theta(n)$
B-Tree	$\theta(\log(n))$	$\theta(\log(n))$	$\theta(\log(n))$	$\theta(\log(n))$	$\theta(\log(n))$	$\theta(\log(n))$	$\theta(\log(n))$	$\theta(\log(n))$	$\theta(n)$
Red-Black Tree	$\theta(\log(n))$	$\theta(\log(n))$	$\theta(\log(n))$	$\theta(\log(n))$	$\theta(\log(n))$	$\theta(\log(n))$	$\theta(\log(n))$	$\theta(\log(n))$	$\theta(n)$
Splay Tree	N/A	$\theta(\log(n))$	$\theta(\log(n))$	$\theta(\log(n))$	N/A	$\theta(\log(n))$	$\theta(\log(n))$	$\theta(\log(n))$	$\theta(n)$
AVL Tree	$\theta(\log(n))$	$\theta(\log(n))$	$\theta(\log(n))$	$\theta(\log(n))$	$\theta(\log(n))$	$\theta(\log(n))$	$\theta(\log(n))$	$\theta(\log(n))$	$\theta(n)$
KD Tree	$\theta(\log(n))$	$\theta(\log(n))$	$\theta(\log(n))$	$\theta(\log(n))$	$\theta(n)$	$\theta(n)$	$\theta(n)$	$\theta(n)$	$\theta(n)$

Tổng của dãy số: 1, 9, 6, 8

Index	0	1	2	3
0	1	10	16	24
1		9	15	23
2			6	14
3				8

1. Đặt vấn đề

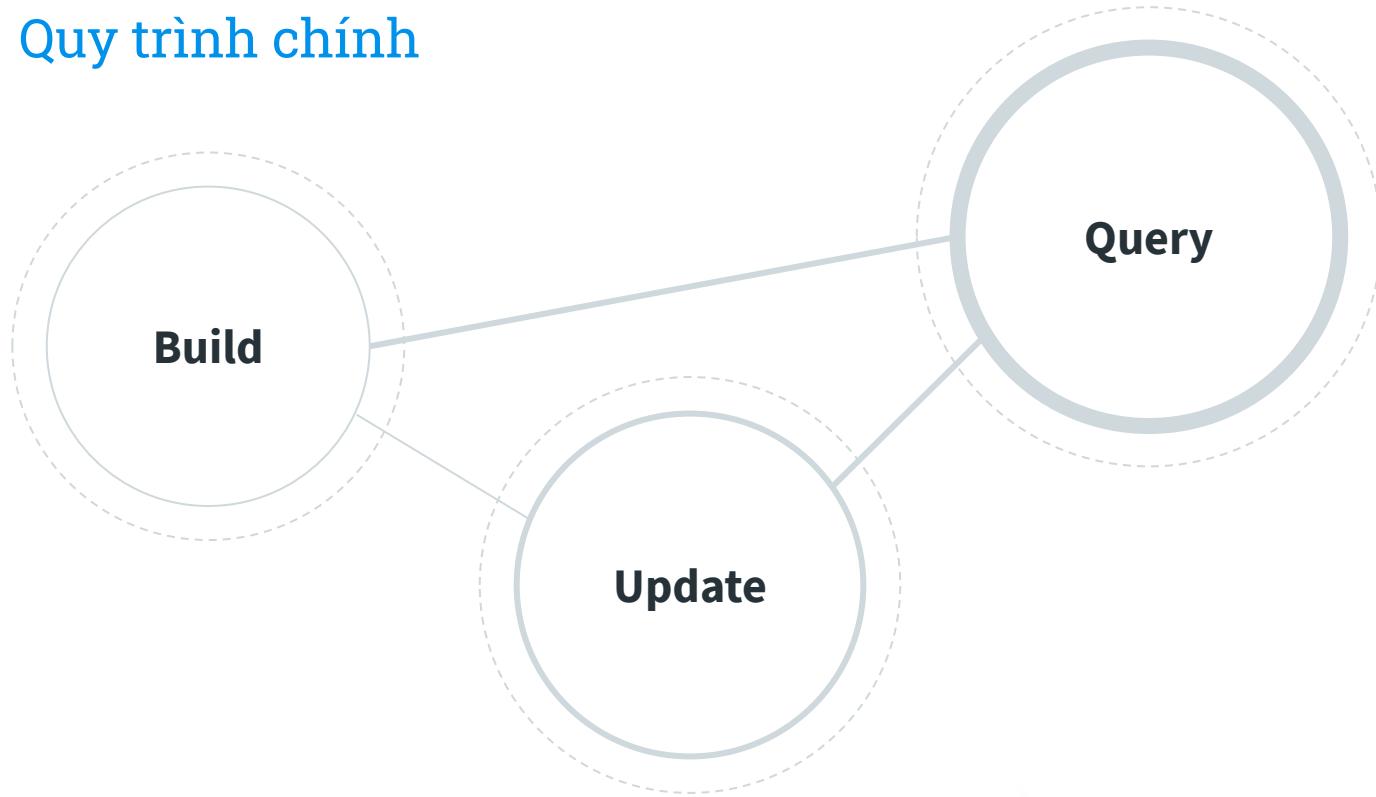
1, 3, 5, 7, 9, 11, 13

- Tính max/min của đoạn $[i, j]$?

I. SEGMENT TREE LÀ GÌ?

- Cây phân đoạn hay cây thống kê, là cấu trúc dữ liệu cây thường dùng để lưu trữ thông tin về khoảng hay phân đoạn.
- Là một cấu trúc dạng tĩnh mà một khi đã xây dựng thì không thể sửa đổi
- Là một cấu trúc dữ liệu cho phép ta trả lời các truy vấn dữ liệu trên mảng một cách hiệu quả nhưng vẫn đủ linh hoạt để cho phép ta sửa đổi mảng

Quy trình chính



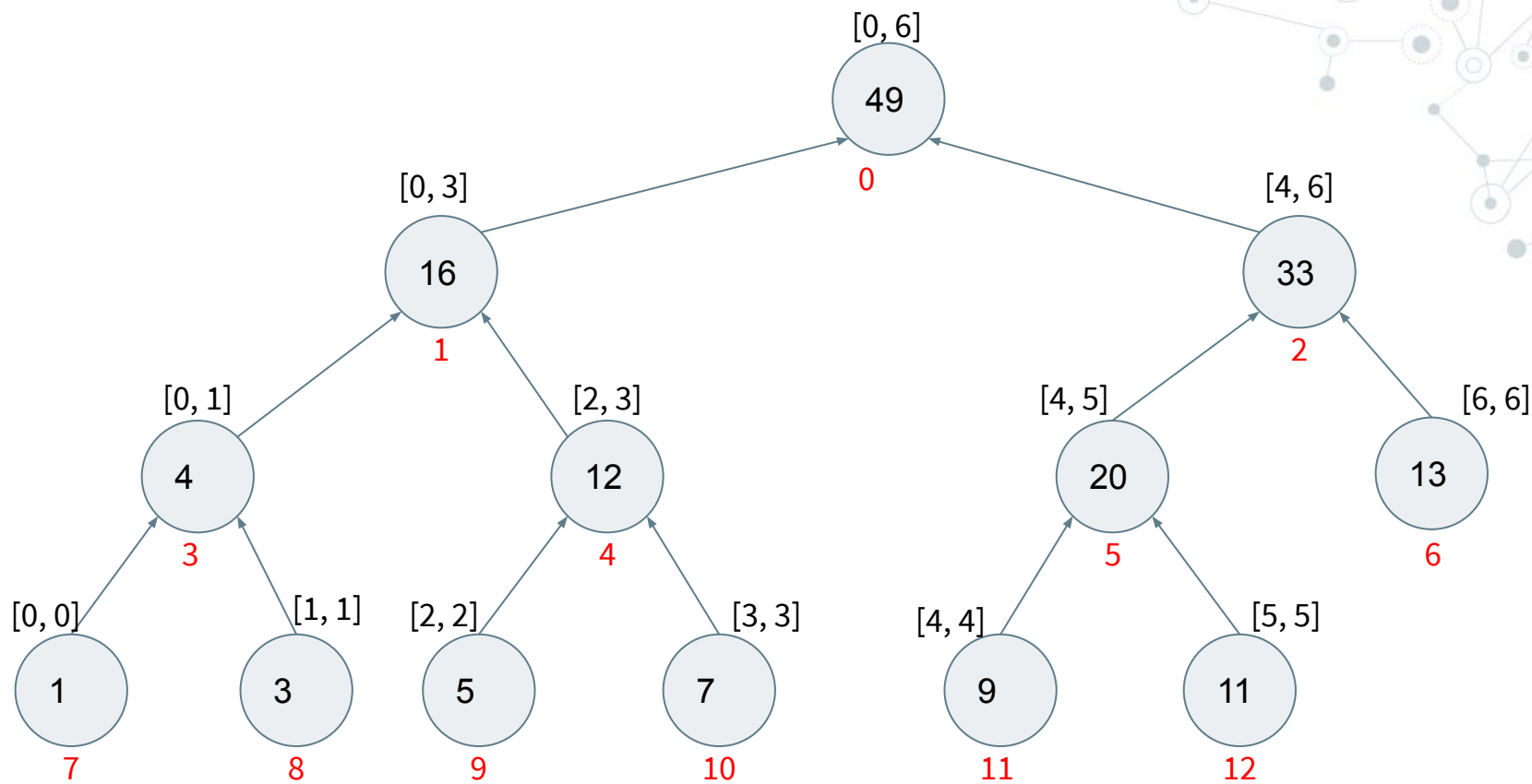
1. KHỞI TẠO

- Cây phân đoạn là một cây nhị phân đầy đủ. Mỗi node trong cây phân đoạn sẽ quản lý thông tin của một đoạn trên dãy số đã cho.
- Tùy thuộc vào loại thao tác truy vấn mà mỗi Node sẽ lưu 1 giá trị khác nhau. Ví dụ: Tính tổng thì sẽ lưu giá trị tổng của đoạn node quản lý, min/max thì node sẽ lưu giá trị min/max của đoạn, ...
- Với 1 dãy số n phần tử, node đầu tiên (node gốc) của cây sẽ quản lý đoạn $[0, n]$, nút thứ 2 là node con bên trái của node gốc sẽ quản lý đoạn $[0, n/2 + 1]$, node thứ 3 là node con bên phải sẽ quản lý đoạn $[n/2 + 2, n]$, cứ tiếp tục vậy cho đến hết.
- Để cài đặt cây phân đoạn, ta có thể dùng mảng 1 chiều. Phần tử 0 của mảng sẽ là node gốc.
- Độ phức tạp bộ nhớ của segment tree là một bộ nhớ tuyến tính với giá trị là $4n$

2. KHỞI TẠO

```
#Construct a segment tree
ST = [0] * n * 4

def build(arr, st, id : int, l : int, r : int):
    if l == r:
        st[id] = arr[l]
        return
    else:
        mid = (l + r) // 2
        build(arr, st, id * 2 + 1, l, mid)
        build(arr, st, id * 2 + 2, mid + 1, r)
        st[id] = st[id * 2 + 2] + st[id * 2 + 1]
```



$A = [1, 3, 5, 7, 9, 11, 13]$

3. TRUY VẤN

“

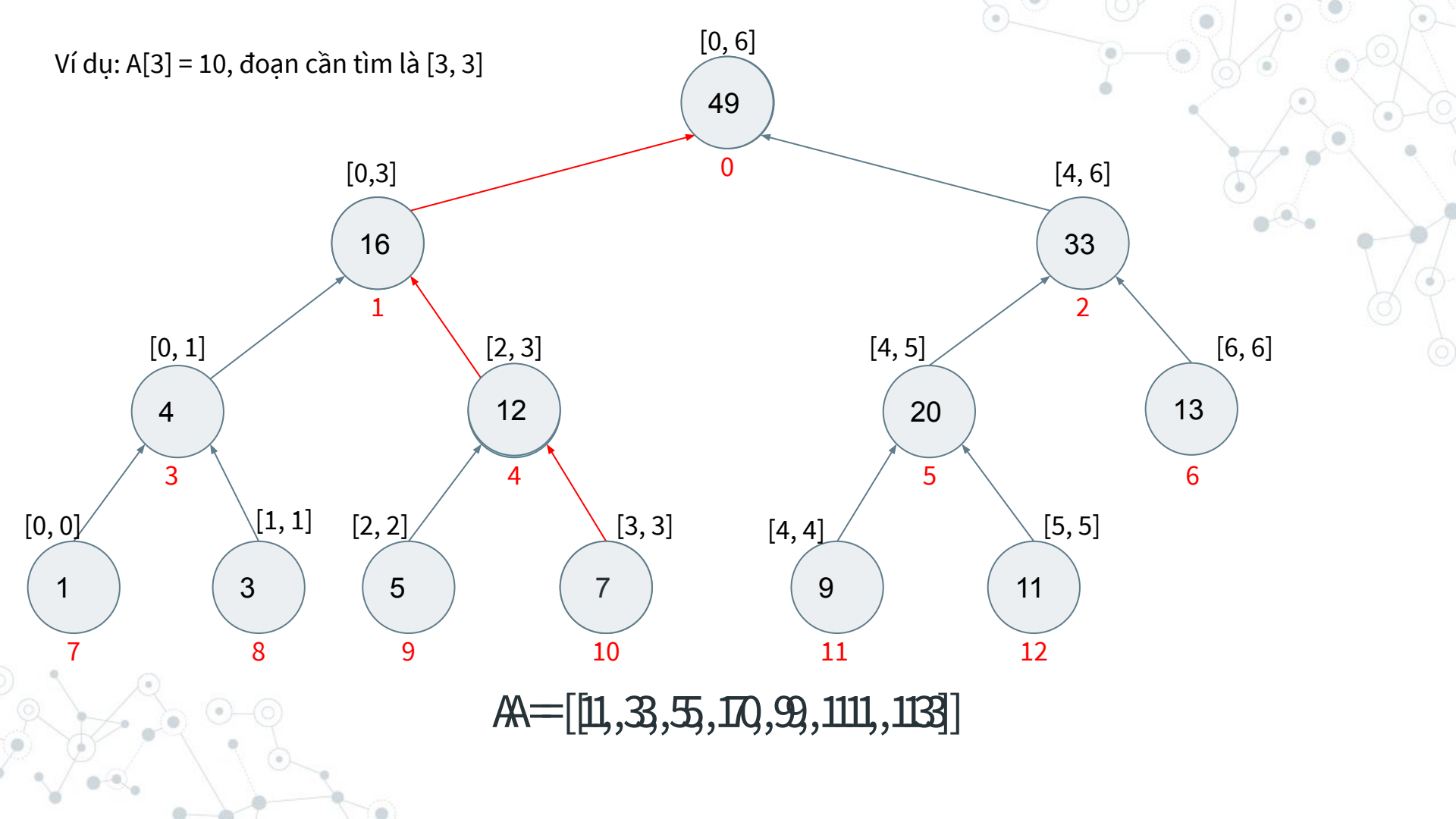
```
def get(st, id : int, l : int, r : int, u : int, v : int):  
    if v < l or r < u:  
        return 0  
    if u <= l and r <= v:  
        return st[id]  
    mid = (l + r) // 2  
    return get(st, id * 2 + 1, l, mid, u, v) + get(st, id * 2 + 2, mid + 1, r, u, v)
```

4. Cập nhật

- Một thao tác cập nhật cơ bản là thực hiện phép gán $A[j] = x$.
- Ý tưởng: Dùng hàm đệ quy theo quy trình sau.
 - Gọi hàm đệ quy từ nút gốc.
 - Truy vấn xuống cho tới khi gặp node lá đại diện cho vị trí phần tử cần cập nhật.
 - Cập nhật giá trị node và kết thúc đệ quy tại node đó.
 - Trở lại cập nhật giá trị các node cha của node vừa cập nhật tùy theo yêu cầu truy vấn. VD: tổng, hoặc min/max đoạn.

Ví dụ: $A[3] = 10$, đoạn cần tìm là $[3, 3]$

$A = [11, 33, 55, 10, 99, 111, 133]$



Ví dụ: $A[3] = 10$, đoạn cần tìm là $[3, 3]$

```
graph TD; 49((49 [0, 6] 0)) --> 16((16 [0, 3] 1)); 49 --> 33((33 [4, 6] 2)); 16 --> 4((4 [0, 1] 3)); 16 --> 12((12 [2, 3] 4)); 33 --> 20((20 [4, 5] 5)); 33 --> 13((13 [6, 6] 6)); 4 --> 1((1 [0, 0] 7)); 4 --> 3((3 [1, 1] 8)); 12 --> 5((5 [2, 2] 9)); 12 --> 7((7 [3, 3] 10)); 20 --> 9((9 [4, 4] 11)); 20 --> 11((11 [5, 5] 12));
```

$A = [11, 33, 55, 10, 99, 111, 133]$

4. Cập nhật

“

```
def update(st, id: int, l: int, r: int, pos: int, val: int):  
    if l == r:  
        st[id] = val  
        return  
    mid = (l + r) // 2  
    if pos <= mid: update(st, id * 2 + 1, l, mid, pos, val)  
    else: update(st, id * 2 + 2, mid + 1, r, pos, val)  
  
    st[id] = st[id * 2 + 1] + st[id * 2 + 2]
```

4. Cập nhật

- $A[j]$ chỉ tham gia vào 1 node ở mỗi bậc của cây. Do đó độ phức tạp khi cập nhật 1 node là $O(\log(n))$.

5. Bộ nhớ

- Số node lá: 2^x
- Tổng số node: $2^{(x)} - 1$
- Chiều cao cây: x
- Độ phức tạp bộ nhớ: $4N$
- Độ phức tạp thời gian truy vấn: $O(\log n)$



A decorative graphic at the top of the slide featuring a network of interconnected nodes and lines. A central node is highlighted with a dashed circle and contains a blue quotation mark.

“

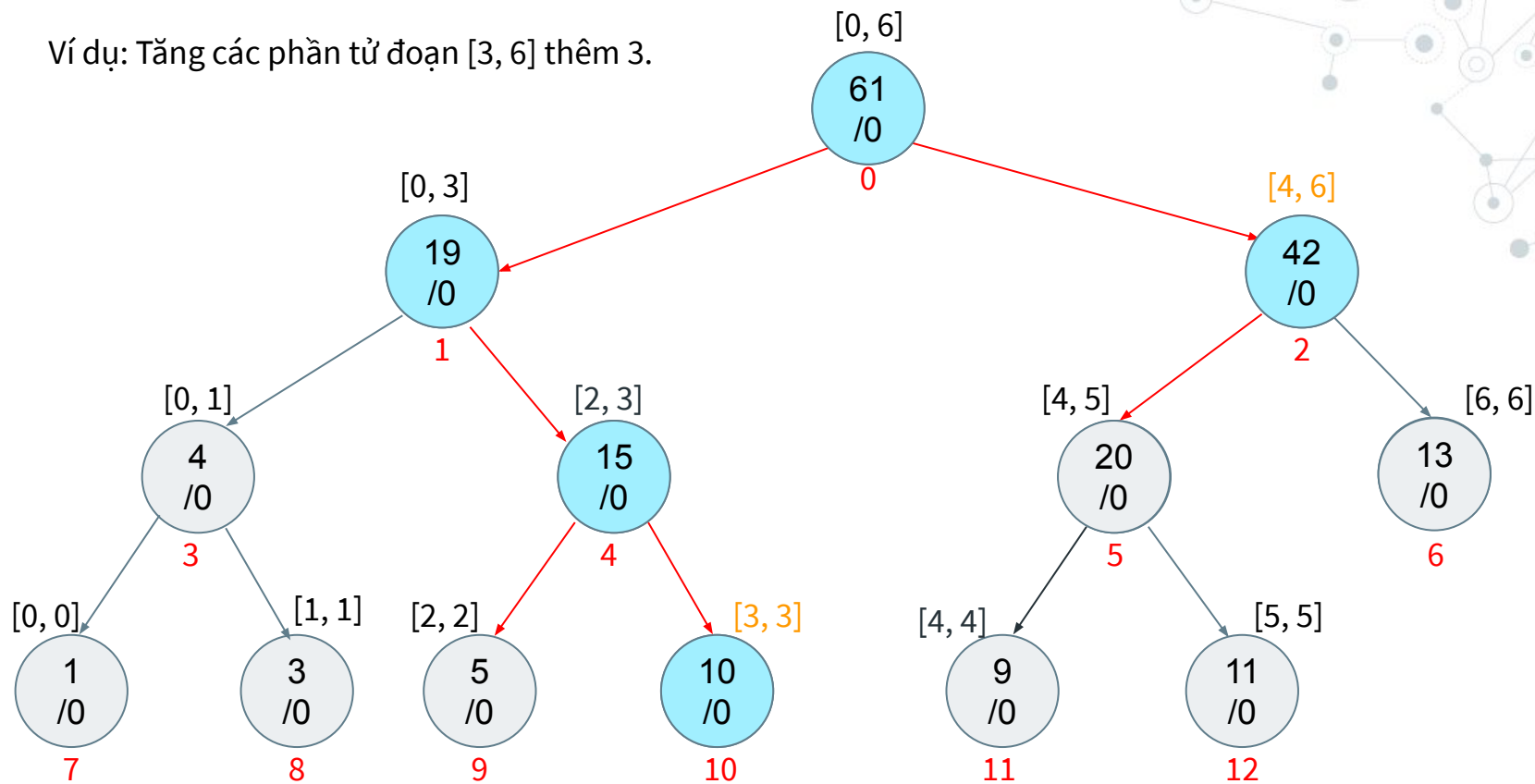
6. MỘT SỐ KỸ THUẬT NÂNG CAO

6.1. Kỹ thuật lan truyền lười biếng

- Là kĩ thuật dùng để giảm độ phức tạp của cây phân đoạn trong các truy vấn cập nhật đoạn.
- Khi cập nhật 1 phần tử, với phương pháp thông thường có độ phức tạp là $O(\log(n))$.
- Vậy khi cập nhật 1 đoạn $[u, v]$ thì độ phức tạp là $O(n \cdot \log(n))$
 \Rightarrow Nó rất là lớn, lãng phí thời gian
- Kỹ thuật này sẽ không cập nhật hết từng phần tử mà chỉ cập nhật đoạn cha quản lý các phần tử đó.

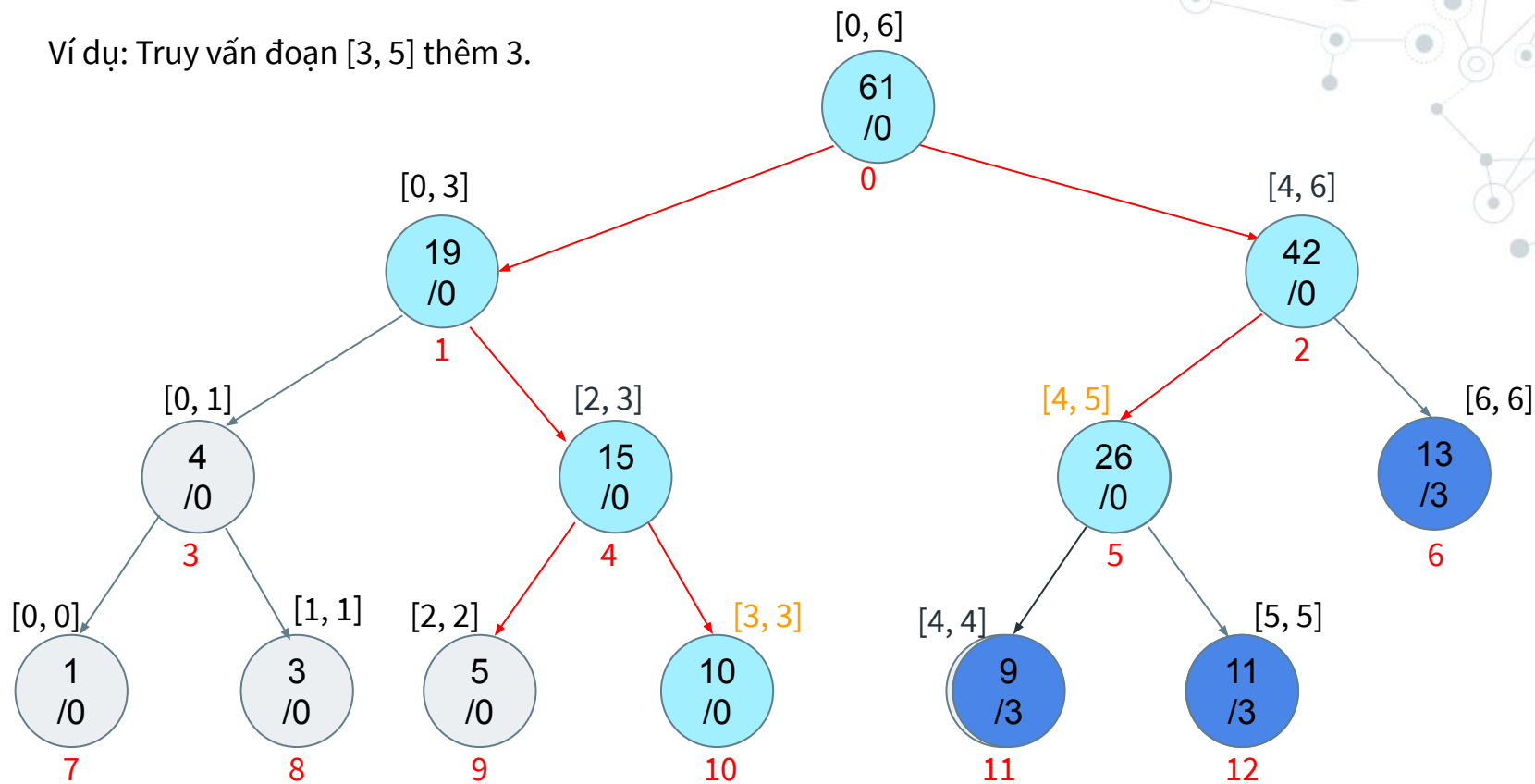


Ví dụ: Tăng các phần tử đoạn $[3, 6]$ thêm 3.



$A = [1, 3, 5, 7, 9, 11, 13]$

Ví dụ: Truy vấn đoạn [3, 5] thêm 3.



$A = [1, 3, 5, 7, 9, 11, 13]$

6.1. Kỹ thuật lan truyền lười biếng



Ý tưởng:

- Cách hoạt động giống như lúc truy vấn. Ta không cập nhật toàn bộ node mà chỉ cập nhật những node quản lý đoạn nằm gọn trong yêu cầu, và node con của nó được bỏ qua(cập nhật sau nếu cần).

Nguyên tắc:

- Nếu node đang duyệt còn có giá trị được thêm vào chưa xét đến (tức là từ các truy vấn cập nhật xảy ra trước đó, mà do lười biếng nên chưa cập nhật hết), thì ta buộc phải cập nhật nó trước khi tiếp tục duyệt sâu hơn.

```
def Update(id: int, l: int, r: int, u: int, v: int, k: int):  
    #[l, r] nằm ngoài [u, v] cần truy vấn  
    if l > v or r < u:  
        return  
    #[l, r] nằm gọn trong [u, v] cần truy vấn  
    if l >= u and r <= v:  
        if l == r:  
            st[id].val += k  
        else:  
            st[id].val += (r - l + 1) * k  
            st[id * 2 + 1].lazy = k  
            st[id * 2 + 2].lazy = k  
        return  
    #[l, r] không nằm gọn trong [u, v] cần truy vấn, xuống node con  
    mid = (l + r) // 2  
    Update(id * 2 + 1, l, mid, u, v, k)  
    Update(id * 2 + 2, mid + 1, r, u, v, k)  
    st[id].val = st[id * 2 + 1].val + st[id * 2 + 2].val
```

6.2. TẬP DỤNG ĐẶC ĐIỂM CỦA KHÓA

- Sử dụng chính thuộc tính của khóa để làm các tác vụ khác
- Thêm vào các khóa bổ sung

Tận dụng khóa chính

- Ví dụ với bài toán như sau: Đếm số lượng số 0 có trong mảng và tìm vị trí của số 0 thứ k

```
def find_kth(st, id : int, l : int, r : int, k : int):  
    if k > t[v]:  
        return -1  
    if l == r:  
        return l  
    mid = (l + r) // 2  
    if t[v * 2] >= k:  
        return find_kth(st, id * 2, l, r, k)  
    else: return find_kth(st, id * 2 + 1, l, r, k - t[v * 2])
```




Thêm vào các khóa bổ sung

- Thêm một số khóa đặc biệt, đặc thù để làm một số công việc cụ thể
- Lưu các khóa dưới cấu trúc dữ liệu khác

Thêm các khóa đặc thù

- Ví dụ với bài toán như sau: Đếm số lượng số 0 có trong mảng và tìm vị trí của số 0 thứ k

```
class Node:
    def __init__(self, optimal = None, open_bracket = None, close_bracket = None):
        self.optimal = optimal
        self.open_bracket = open_bracket
        self.close_bracket = close_bracket
    def __str__(self):
        return str(self.optimal)
```

Lưu khóa dưới dạng cấu trúc khác

- Ví dụ với bài toán như sau: Đếm số lượng số 0 có trong mảng và tìm vị trí của số 0 thứ k

```
def build(arr, st, id : int, l : int, r : int):  
    if (l == r):  
        # Đoạn gồm 1 phần tử. Ta dễ dàng khởi tạo nút trên ST.  
        st[id].append(arr[l])  
        return  
  
    mid = (l + r) // 2  
    build(arr, st, id * 2, l, mid)  
    build(arr, st, id * 2 + 1, mid + 1, r);  
  
    st[id] = mergeSort(st[id * 2], st[id * 2 + 1])
```

A decorative network diagram at the top of the slide, featuring a series of interconnected nodes and lines. A central node is highlighted with a dashed circle and a solid circle, containing a large blue quotation mark.

“

7. SEGMENT TREE 2 CHIỀU

7. SEGMENT TREE 2 CHIỀU

- Một cây phân đoạn có thể được tổng quát hóa với các thứ nguyên cao hơn
- Nếu trong trường hợp một chiều chúng ta chia các chỉ số của mảng thành các phân đoạn, thì trong hai chiều chúng ta tạo ra một cây phân đoạn thông thường với các chỉ số đầu tiên, và cho mỗi phân đoạn chúng ta xây dựng một cây thông thường với các chỉ số thứ hai.

7. SEGMENT TREE 2 CHIỀU

- Trong trường hợp này, segment tree của chúng ta sẽ build theo tuần tự: Đầu tiên là trên trục x, sau đó là đến trục y

```
def build_x(arr, st, int vx, int lx, int rx):  
    if (lx != rx):  
        mx = (lx + rx) // 2  
        build_x(arr, st, vx * 2, lx, mx)  
        build_x(arr, st, vx * 2 + 1, mx + 1, rx)  
  
    build_y(arr, st, vx, lx, rx, 1, 0, m - 1)
```

Build y

```
def build_y(arr, st, vx: int, lx: int, rx : int, vy : int, ly : int, ry : int):  
    if (ly == ry):  
        if (lx == rx):  
            st[vx][vy] = a[lx][ly]  
        else:  
            st[vx][vy] = st[vx * 2][vy] + st[vx * 2 + 1][vy]  
    else:  
        mid = (ly + ry) // 2  
        build_y(arr, st, vx, lx, rx, vy * 2, ly, mid)  
        build_y(arr, st, vx, lx, rx, vy * 2 + 1, mid + 1, ry)  
        st[vx][vy] = st[vx][vy * 2] + st[vx][vy * 2 + 1]
```

Query tính sum

```
✓ def sum_y(st, vx : int, vy : int, tly : int, try_ : int, ly : int, ry : int) :  
✓     if (ly > ry):  
✓         return 0  
✓     if (ly == tly and try_ == ry):  
✓         return st[vx][vy]  
        tmy = (tly + try_) // 2  
✓     return sum_y(st, vx, vy*2, tly, tmy, ly, min(ry, tmy))  
        + sum_y(st, vx, vy*2+1, tmy+1, try_, max(ly, tmy+1), ry)  
  
✓ def sum_x(st, vx : int, tlx : int, trx : int, lx : int, rx : int, ly : int, ry : int) :  
✓     if (lx > rx):  
✓         return 0  
✓     if (lx == tlx && trx == rx):  
✓         return sum_y(st, vx, 1, 0, m-1, ly, ry)  
        tmx = (tlx + trx) // 2  
✓     return sum_x(st, vx*2, tlx, tmx, lx, min(rx, tmx), ly, ry)  
        + sum_x(st, vx*2+1, tmx+1, trx, max(lx, tmx+1), rx, ly, ry)
```


8. CÂY PHÂN ĐOẠN NGẦM

- Thông thường chúng ta muốn khởi tạo một cây segment tree hoàn thiện để quản lý dữ liệu với kích thước cố sẵn và bộ dữ liệu cố sẵn
- Khởi tạo một cây dữ liệu với các giá trị mặc định ban đầu, sau đó chúng ta sẽ thêm dần dần

Hàm khởi tạo

```
class Vertex:
    def __init__(self, left = 0, right = 0):
        self.sum = 0
        self.left = left
        self.right = right
        self.left_child = None
        self.right_child = None
```

Hàm extend và add

```
def extend(self):
    if (not self.left_child) and self.left < self.right:
        mid = (self.left + self.right) // 2
        self.left_child = Vertex(self.left, mid)
        self.right_child = Vertex(mid + 1, self.right)

def add(self, k : int, x : int):
    self.extend()
    self.sum += x
    if self.left_child is not None:
        if k <= self.left_child.right:
            self.left_child.add(k, x)
        else:
            self.right_child.add(k, x)
```

Hàm tính sum

```
def get_sum(self, u : int, v : int):  
    if u <= self.left and v >= self.right:  
        return self.sum  
    if self.right < v and self.left > u:  
        return 0  
    self.extend()  
    sumLeft = 0  
    sumRight = 0  
    if self.left_child : sumLeft = self.left_child.get_sum(u, v)  
    if self.right_child : sumRight = self.right_child.get_sum(u, v)  
  
    return sumLeft + sumRight
```

9. Cây phân đoạn liên tục

- Cây phân đoạn liên tục là cây phân đoạn ghi nhớ trạng thái trước đó của cây cho mỗi lần thực hiện truy vấn sửa đổi.
- Điều này cho phép ta có thể truy cập bất kỳ phiên bản nào của cây này khi cần thiết.

9. Cây phân đoạn liên tục

- Thực tế đối với việc cập nhật cây (cụ thể là cập nhật một node) chỉ cập nhật một lượng node nhất định.

=> Phiên bản mới của cây chỉ chứa n node mới, các node còn lại sẽ giống với phiên bản cũ.

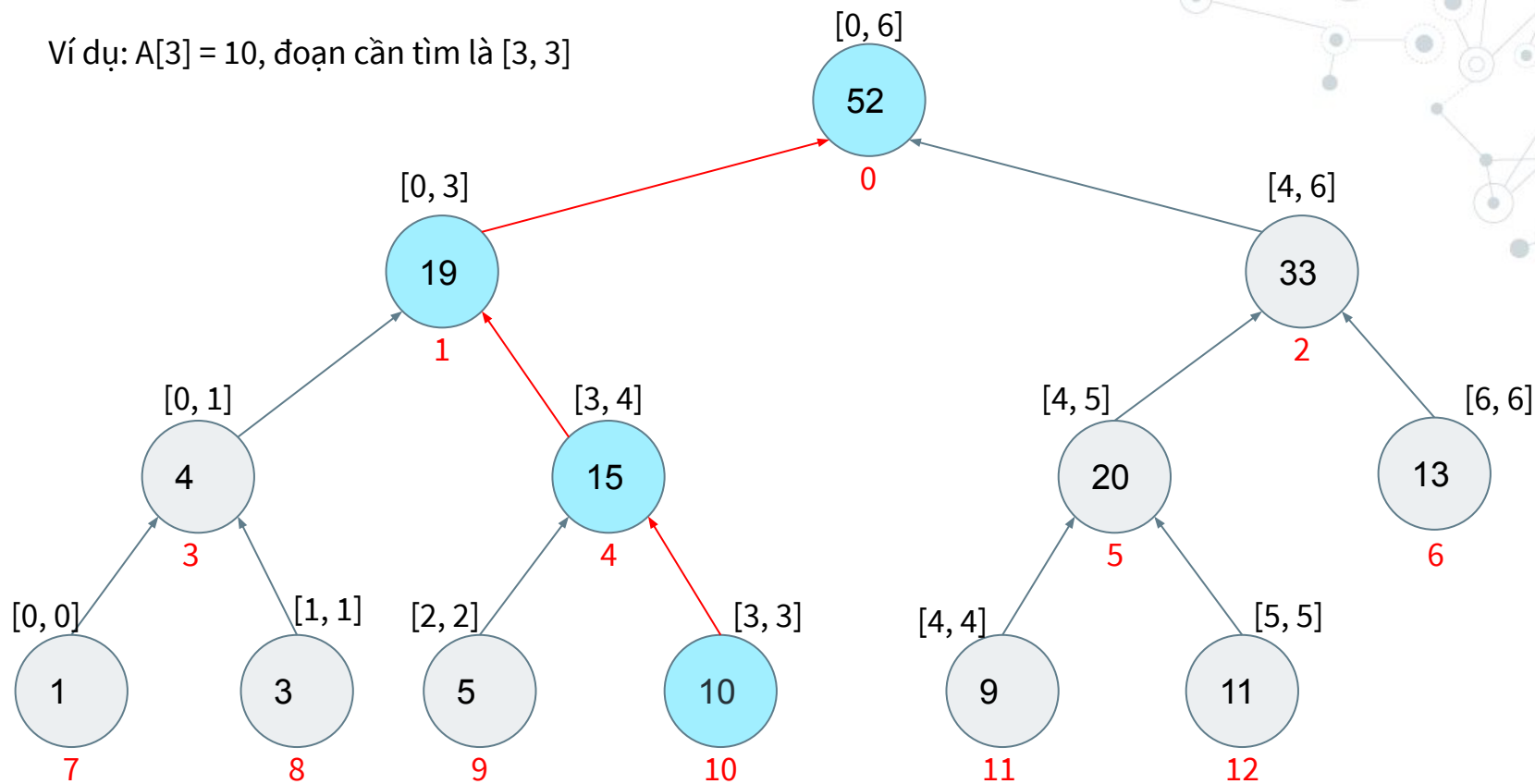
- Áp dụng ý tưởng này, khi tạo 1 phiên bản mới, ta chỉ tạo n node mới này, phần còn lại lấy từ phiên bản trước.

9. Cây phân đoạn liên tục

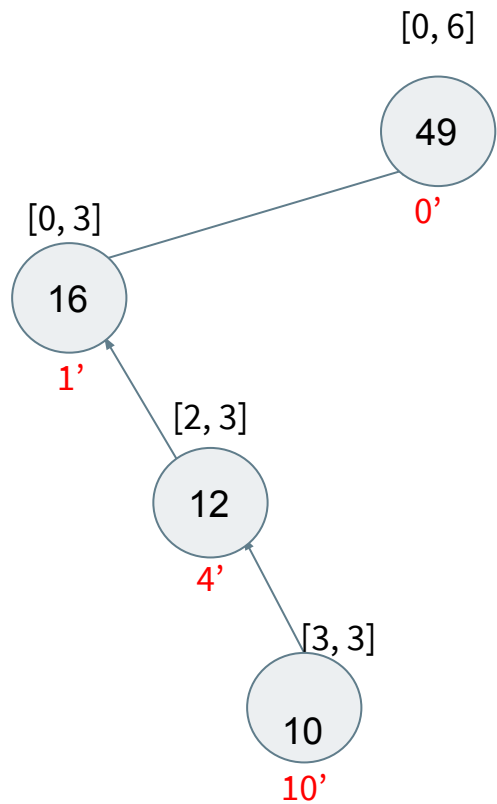
Cài đặt:

- Node sẽ thêm 2 con trỏ để lưu trữ node con trái và phải.
- Để theo dõi tất cả phiên bản, ta chỉ cần theo dõi node gốc đầu tiên của mỗi phiên bản.

Ví dụ: $A[3] = 10$, đoạn cần tìm là $[3, 3]$

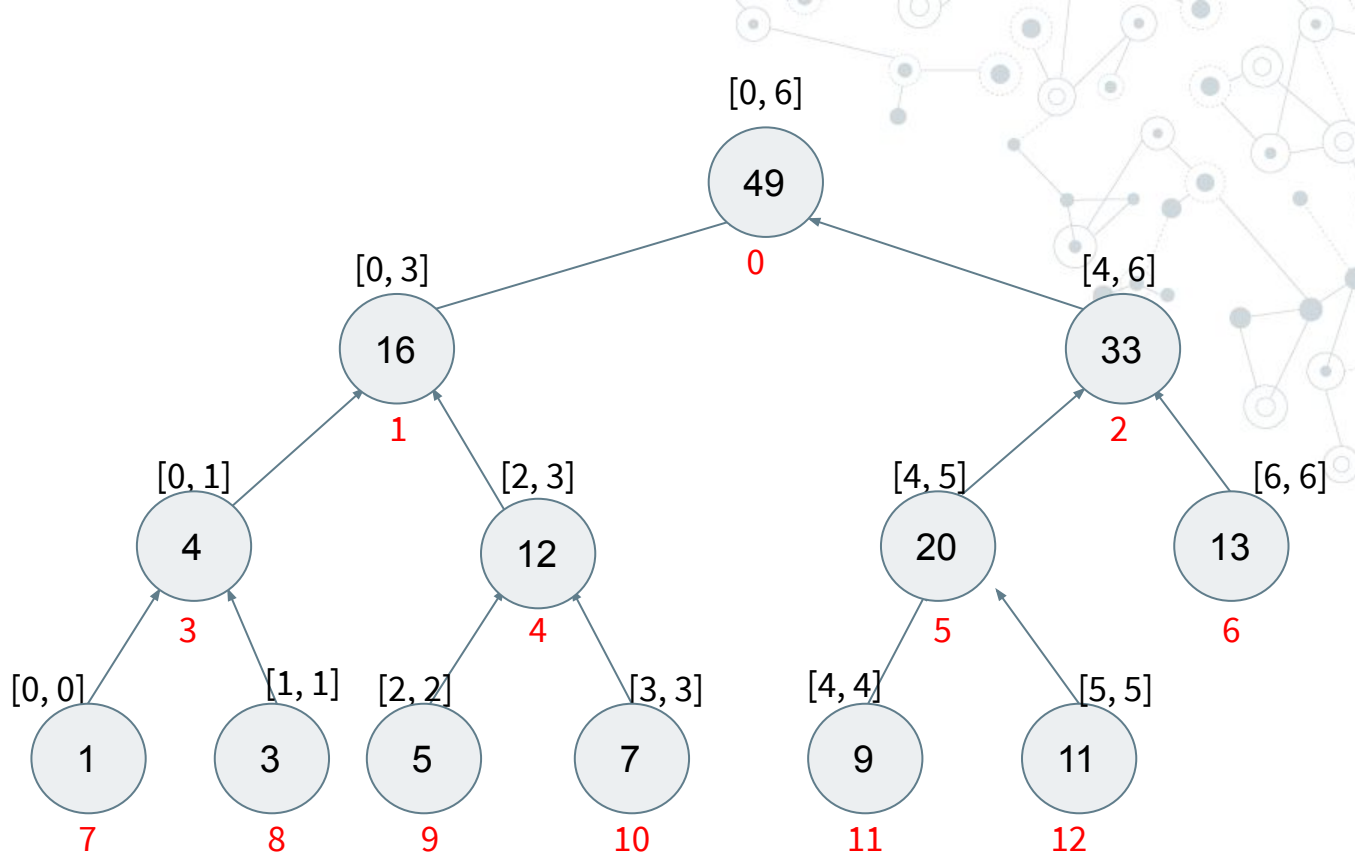


$A = [1, 3, 5, 10, 9, 11, 13]$



$A = [1, 3, 5, 10, 9, 11, 13]$

Ver 1



$A = [1, 3, 5, 7, 9, 11, 13]$

Ver 0

```
struct node
{
    int val;

    node* left, *right;

    node() {}
    node(node* l, node* r, int v)
    {
        left = l;
        right = r;
        val = v;
    }
};
```

```
void build(node* n,int l,int r)
{
    if (l==r)
    {
        n->val = arr[l];
        return;
    }
    int mid = (l+r) / 2;
    n->left = new node(NULL, NULL, 0);
    n->right = new node(NULL, NULL, 0);
    build(n->left, l, mid);
    build(n->right, mid+1, r);
    n->val = n->left->val + n->right->val;
}
```

```
void upgrade(node* prev, node* cur, int l, int r,
             int idx, int value)
{
    if (idx > r or idx < l or l > r)
        return;

    if (l == r)
    {
        // modification in new version
        cur->val = value;
        return;
    }
    int mid = (l+r) / 2;
    if (idx <= mid)
    {
        // link to right child of previous version
        cur->right = prev->right;

        // create new node in current version
        cur->left = new node(NULL, NULL, 0);

        upgrade(prev->left, cur->left, l, mid, idx, value);
    }
    else
    {
        // link to left child of previous version
        cur->left = prev->left;

        // create new node for current version
        cur->right = new node(NULL, NULL, 0);

        upgrade(prev->right, cur->right, mid+1, r, idx, value);
    }
}
```

```
int get(node* n, int l, int r, int u, int v)
{
    if (u > r or v < l or l > r)
        return 0;
    if (u <= l and r <= v)
        return n->val;
    int mid = (l + r) / 2;
    int p1 = get(n->left, l, mid, u, v);
    int p2 = get(n->right, mid+1, r, u, v);
    return p1+p2;
}
```

A decorative network diagram at the top of the slide, featuring a series of interconnected nodes and lines. The nodes are represented by circles of varying sizes, some solid and some dashed, connected by thin lines. A central node is highlighted with a dashed circle and a blue double quote symbol.

“

DEMO

A decorative network diagram at the top of the slide, featuring a series of interconnected nodes and lines. The nodes are represented by circles of varying sizes, some solid and some dashed, connected by thin lines. A central node is highlighted with a dashed circle and a blue double quote symbol.

“

9. TỔNG KẾT

PHÂN TÍCH VỀ SEGMENT TREE

ƯU ĐIỂM

Tốc độ truy vấn nhanh, chỉ $O(\log n)$

Linh hoạt để cho phép ta sửa đổi mảng

S

NHƯỢC ĐIỂM

Tốn bộ nhớ

Code phức tạp nếu cần phù hợp với nhiều loại truy vấn khác nhau

W

Sử dụng trong các bộ dữ liệu lớn để quản lý nhiều kiểu dữ liệu nhờ tốc độ truy vấn nhanh

CƠ HỘI

O

T

Phải build lại từ đầu trong trường hợp cần làm mới bộ dữ liệu

TIỀM ẨN



Thanks!

Any questions?

You can find me at:

19522323@gm.uit.edu.vn

BÀI TẬP VỀ NHÀ

