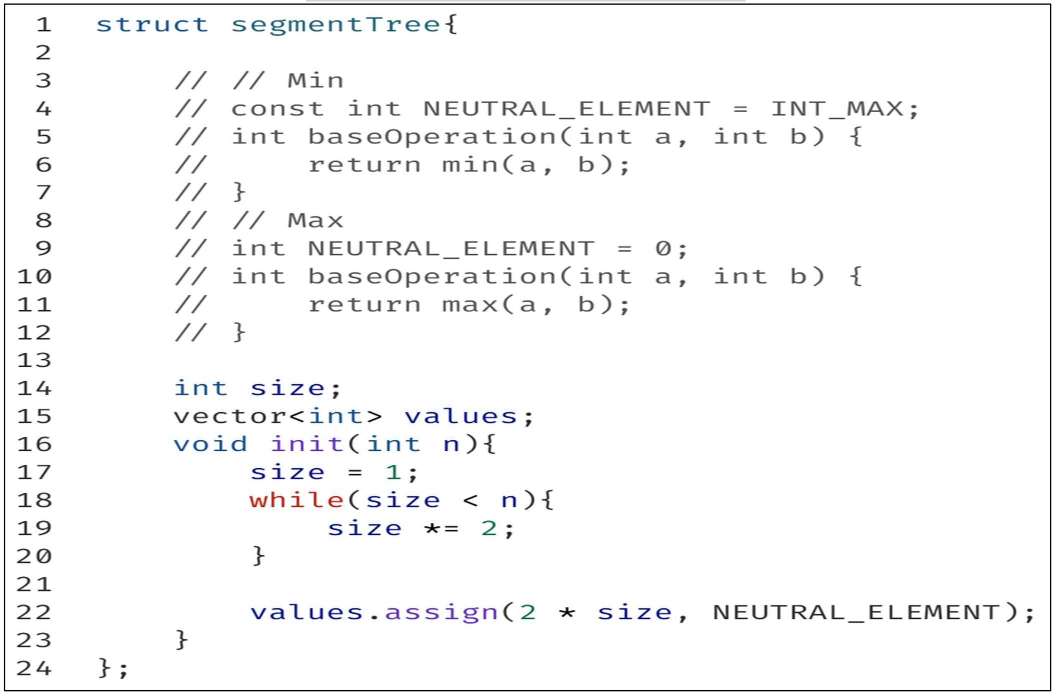
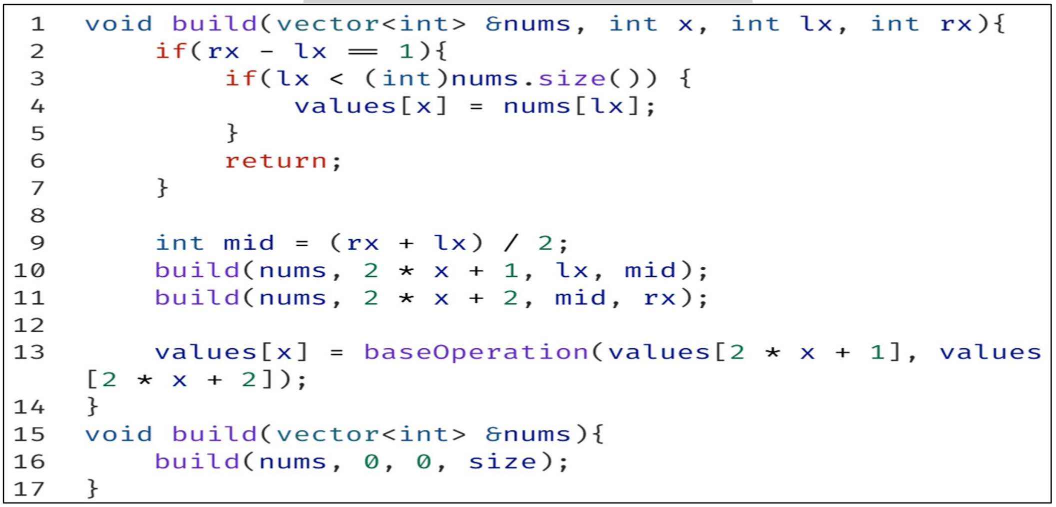
Segment Tree Min-Max

\\   Base code   //



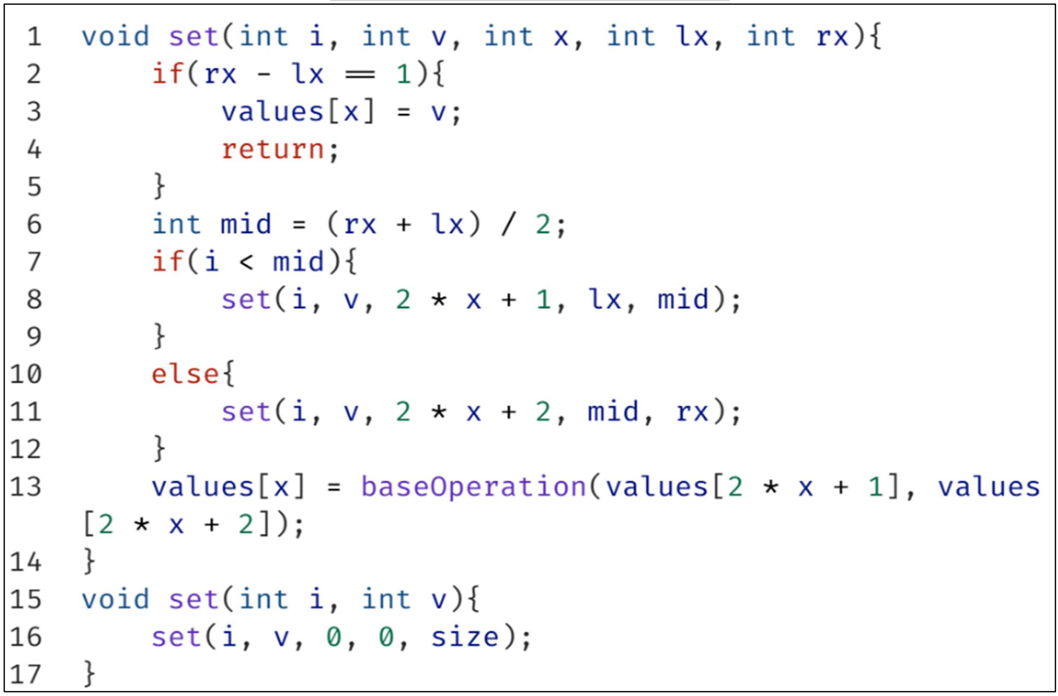
```cpp
struct segmentTree{

    // // Min
    // const int NEUTRAL_ELEMENT = INT_MAX;
    // int baseOperation(int a, int b) {
    //     return min(a, b);
    // }
    // // Max
    // int NEUTRAL_ELEMENT = 0;
    // int baseOperation(int a, int b) {
    //     return max(a, b);
    // }

    int size;
    vector<int> values;
    void init(int n){
        size = 1;
        while(size < n){
            size *= 2;
        }

        values.assign(2 * size, NEUTRAL_ELEMENT);
    }
};
```

\\   Build Code   //



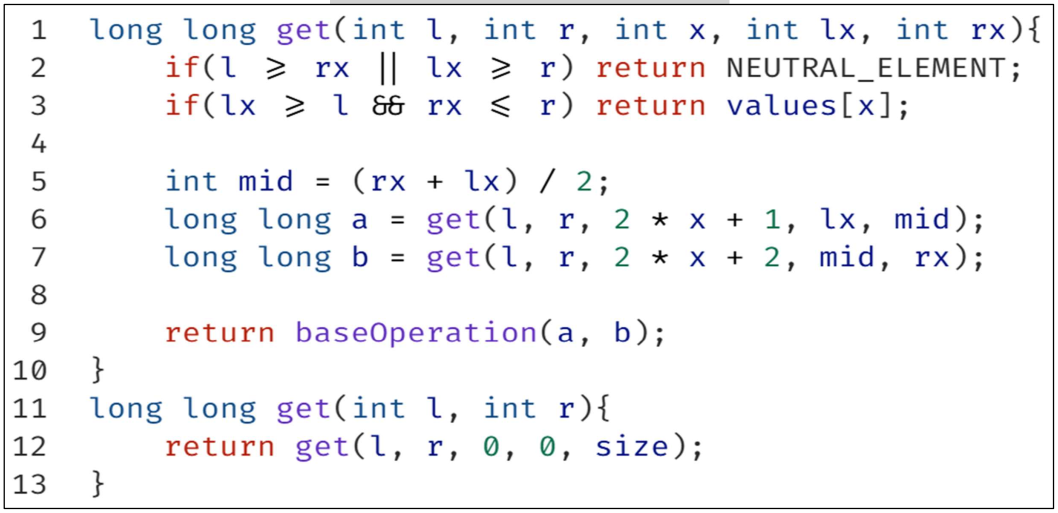
```cpp
void build(vector<int> &nums, int x, int lx, int rx){
    if(rx - lx == 1){
        if(lx < (int)nums.size()) {
            values[x] = nums[lx];
        }
        return;
    }

    int mid = (rx + lx) / 2;
    build(nums, 2 * x + 1, lx, mid);
    build(nums, 2 * x + 2, mid, rx);

    values[x] = baseOperation(values[2 * x + 1], values[2 * x + 2]);
}
void build(vector<int> &nums){
    build(nums, 0, 0, size);
}
```

\\   Set Code   //



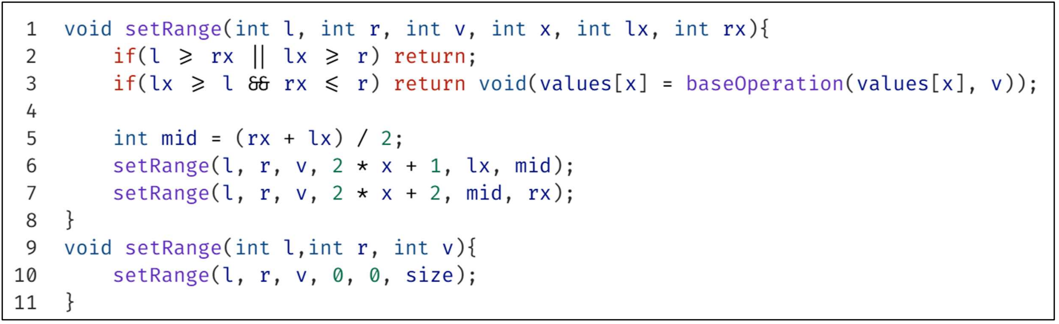
```cpp
void set(int i, int v, int x, int lx, int rx){
    if(rx - lx == 1){
        values[x] = v;
        return;
    }
    int mid = (rx + lx) / 2;
    if(i < mid){
        set(i, v, 2 * x + 1, lx, mid);
    }
    else{
        set(i, v, 2 * x + 2, mid, rx);
    }
    values[x] = baseOperation(values[2 * x + 1], values[2 * x + 2]);
}
void set(int i, int v){
    set(i, v, 0, 0, size);
}
```

\\   Get Code   //



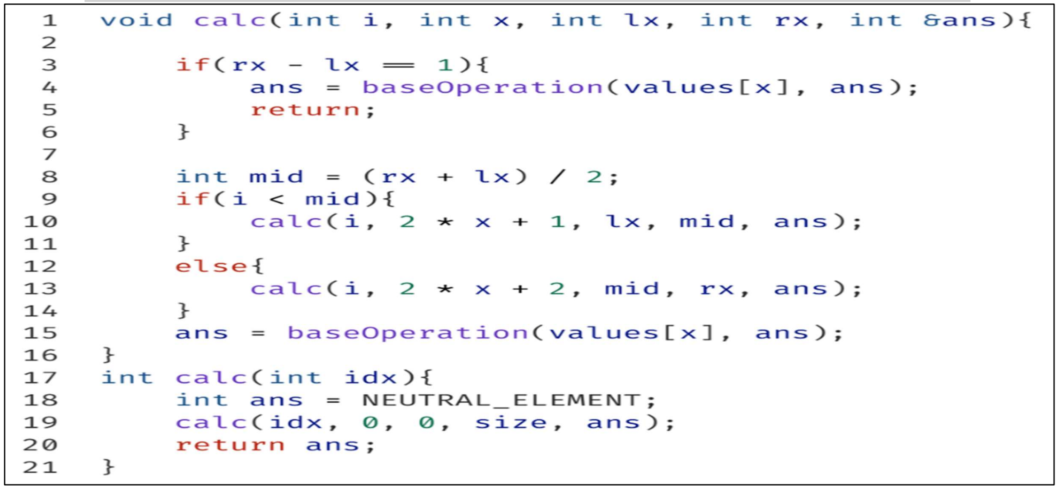
```cpp
long long get(int l, int r, int x, int lx, int rx){
    if(l >= rx || lx >= r) return NEUTRAL_ELEMENT;
    if(lx >= l && rx <= r) return values[x];

    int mid = (rx + lx) / 2;
    long long a = get(l, r, 2 * x + 1, lx, mid);
    long long b = get(l, r, 2 * x + 2, mid, rx);

    return baseOperation(a, b);
}
long long get(int l, int r){
    return get(l, r, 0, 0, size);
}
```

\\ Set Range Code //



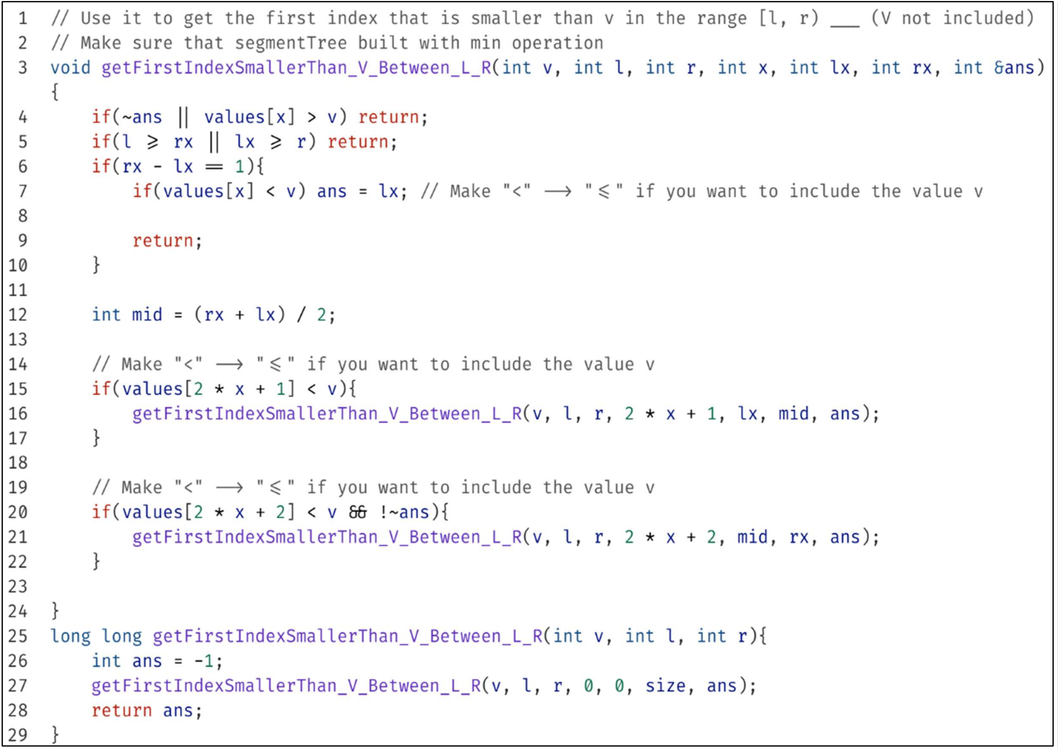
```cpp
void setRange(int l, int r, int v, int x, int lx, int rx){
    if(l >= rx || lx >= r) return;
    if(lx >= l && rx <= r) return void(values[x] = baseOperation(values[x], v));

    int mid = (rx + lx) / 2;
    setRange(l, r, v, 2 * x + 1, lx, mid);
    setRange(l, r, v, 2 * x + 2, mid, rx);
}
void setRange(int l,int r, int v){
    setRange(l, r, v, 0, 0, size);
}
```

\\ Calc Code [get value from setRange()] //



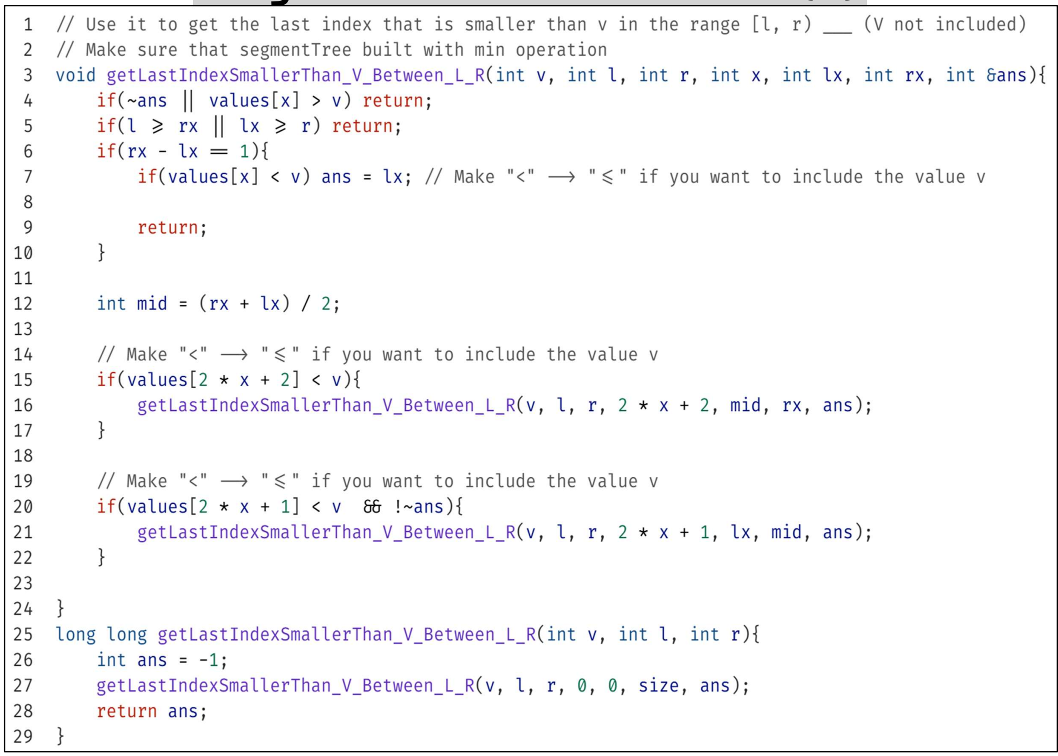
```cpp
void calc(int i, int x, int lx, int rx, int &ans){

    if(rx - lx == 1){
        ans = baseOperation(values[x], ans);
        return;
    }

    int mid = (rx + lx) / 2;
    if(i < mid){
        calc(i, 2 * x + 1, lx, mid, ans);
    }
    else{
        calc(i, 2 * x + 2, mid, rx, ans);
    }
    ans = baseOperation(values[x], ans);
}
int calc(int idx){
    int ans = NEUTRAL_ELEMENT;
    calc(idx, 0, 0, size, ans);
    return ans;
}
```

Get First Index of an element in range less than a value (v)



```cpp
// Use it to get the first index that is smaller than v in the range [l, r) ___ (V not included)
// Make sure that segmentTree built with min operation
void getFirstIndexSmallerThan_V_Between_L_R(int v, int l, int r, int x, int lx, int rx, int &ans){
    if(~ans || values[x] > v) return;
    if(l >= rx || lx >= r) return;
    if(rx - lx == 1){
        if(values[x] < v) ans = lx; // Make "<" -> "<=" if you want to include the value v

        return;
    }

    int mid = (rx + lx) / 2;

    // Make "<" -> "<=" if you want to include the value v
    if(values[2 * x + 1] < v){
        getFirstIndexSmallerThan_V_Between_L_R(v, l, r, 2 * x + 1, lx, mid, ans);
    }

    // Make "<" -> "<=" if you want to include the value v
    if(values[2 * x + 2] < v && !~ans){
        getFirstIndexSmallerThan_V_Between_L_R(v, l, r, 2 * x + 2, mid, rx, ans);
    }

}
long long getFirstIndexSmallerThan_V_Between_L_R(int v, int l, int r){
    int ans = -1;
    getFirstIndexSmallerThan_V_Between_L_R(v, l, r, 0, 0, size, ans);
    return ans;
}
```

Get Last Index of an element in range less than a value (v)



```cpp
// Use it to get the last index that is smaller than v in the range [l, r) ___ (V not included)
// Make sure that segmentTree built with min operation
void getLastIndexSmallerThan_V_Between_L_R(int v, int l, int r, int x, int lx, int rx, int &ans){
    if(~ans || values[x] > v) return;
    if(l >= rx || lx >= r) return;
    if(rx - lx == 1){
        if(values[x] < v) ans = lx; // Make "<" -> "<=" if you want to include the value v

        return;
    }

    int mid = (rx + lx) / 2;

    // Make "<" -> "<=" if you want to include the value v
    if(values[2 * x + 2] < v){
        getLastIndexSmallerThan_V_Between_L_R(v, l, r, 2 * x + 2, mid, rx, ans);
    }

    // Make "<" -> "<=" if you want to include the value v
    if(values[2 * x + 1] < v  && !~ans){
        getLastIndexSmallerThan_V_Between_L_R(v, l, r, 2 * x + 1, lx, mid, ans);
    }

}
long long getLastIndexSmallerThan_V_Between_L_R(int v, int l, int r){
    int ans = -1;
    getLastIndexSmallerThan_V_Between_L_R(v, l, r, 0, 0, size, ans);
    return ans;
}
```

President Segment Tree

```cpp
struct PresidentSegmentTree{
public:
    PresidentSegmentTree(int n){
        sz = n;
        roots.push_back(new node(0));
    }
    void set(int idx , int val , int version = 1){
        version--;
        roots[version] = set(idx , val , roots[version] , 1 , sz);
    }
    void newVersion(int version){
        version--;
        roots.push_back(roots[version]);
    }
    ll query(int l , int r , int version){
        version--;
        return query(l , r , roots[version] , 1 , sz);
    }
private:
    struct node{
        ll val;
        node *l , *r;
        node(ll v){
            val = v;
            l = r = this;
        }
        node(node *_l , node *_r){
            val = 0;
            l = _l;
            r = _r;
            if(l) val+=l->val;
            if(r) val+=r->val;
        }
    };
    vector<node*> roots;
    int sz;
    node* set(int idx , int val , node *cur , int l , int r){
        if(l == r) return new node(val);
        int md = l + (r-l) / 2;
        if(idx <= md) return new node(set(idx , val , cur->l , l , md) , cur->r);
        return new node(cur->l , set(idx , val , cur->r , md+1 , r));
    }
    ll query(int l , int r , node *cur , int lx , int rx){
        if(lx >= l && rx <= r) return cur->val;
        if(lx > r || rx < l) return 0;
        int md = lx + (rx-lx) / 2;
        return query(l , r , cur->l , lx , md) + query(l , r , cur->r , md+1 , rx);
    }
};
```