Введение в алгоритмы

Лекция 11.

Деревья поиска – 4+: Операции в деревьях по неявному ключу. Отложенные операции. Дерево отрезков.

“Split/merge trees”: достижения прошлой лекции

- На прошлой лекции мы познакомились с двумя деревьями, поддерживающими операции Insert, Delete, Search, а также Split(key) и Merge.
- В обоих случаях, Split(key) разделял дерево на два деревья, в одном из которых находились вершины с ключами, ≤ key, а в другом - >key.
- Также напомним, что Merge (*left, *right) требовал, чтобы все ключи дерева right были не меньше ключей дерева left.

«Split/merge trees»: k-я порядковая статистика

- Структура операций позволяет поддерживать в вершинах деревьев (обоих) параметр size, в котором хранится количество вершин в поддереве.
- С помощью такого параметра, можно реализовать операцию Search (index), которая будет искать k-ю порядковую статистику в дереве.
- Search(index) будет работать аналогично Search(key); отличие будет заключаться лишь в способе понимания, в какое из поддеревьев требуется пойти:

```
struct SplayTree {
    SplayTree *left, *right, *parent;
    int key;
    uint32 size;
};

inline uint32 getSize(SplayTree* root) {
    if (root == NULL)
        return 0;
    return root->size;
}

// index is in 0-indexing; returns new root of tree
SplayTree* searchByIndex(SplayTree* root, uint32 index) {
    if (index > getSize(root) || index == 0)
        throw BadIndexException(index, getSize, root);

    while (true) { // invariant: searching for vertex with given index in subtree of *root
        ui32 leftSize = root->size();
        if (leftSize == index) // now *root is vertex we are searching for
            break;
        else if (leftSize > index)
            root = root->left;
        else {
            index -= leftSize + 1;
            root = root->right;
        }
    }
    splay(root);
    return root;
}
```

Деревья по неявному ключу

- А теперь откажемся от отсортированности key; будем выполнять Search и Split лишь по индексу.
- «Ключом» в таких деревьях оказывается индекс элемента; он хранится неявно, откуда и берется название метода.
- Фактически, мы храним массив элементов, который поддерживает следующие операции:
  - Merge(Tree* left, Tree* right) – так как в данной операции требуется лишь значение приоритетов (декартово дерево) либо ничего (splay – дерево), теперь корректным оказывается слияние деревьев в любом порядке!
  - Insert(key, index) – вставка элемента key в дерево так, чтобы соответствующая вершина оказалась index-ой в порядке inOrder-обхода (0-индексация) (вставка через split);

Деревья по неявному ключу: операции

- Delete(index) – вырезать элемент номер index (два Split-а, удаление и Merge);
- Delete(left, right) – удаление подмассива (полностью аналогично Delete(index);
- CyclicShift(num) – циклический сдвиг на num элементов (вправо/влево; один Split + один Merge);
- CyclicShiftOnSubSegment(num, left, right) – выполнить циклических сдвиг на подотрезке;
- assign(index, x) – присвоить i-му элементу значение x;

- Также в каждой вершине можно хранить и поддерживать сумму всех элементов поддерева, что дает возможность простым образом изменять элементы и вычислять сумму элементов на подотрезке с помощью следующей инфраструктуры:

```
struct SplayTree {
    SplayTree *left, *right, *parent;
    int key, sum;
    uint32 size;
};

inline uint32 getSize(SplayTree* root) {
    return (root ? root->size : 0);
}

inline uint32 getSum(SplayTree* root) {
    return (root ? root->sum : 0);
}

void recalculateSize(SplayTree* root) {
    if (!root)
        return;
    root->size = getSize(root->left) + 1 + getSize(root->right);
}

void recalculateSum(SplayTree* root) {
    if (!root)
        return;
    root->sum = getSum(root->left) + 1 + getSum(root->right);
}
```

```
void recalculateSum(SplayTree* root) {
    if (!root)
        return;
    root->sum = getSum(root->left) + 1 + getSum(root->right);
}

void recalculate(SplayTree* root) {
    recalculateSize(root);
    recalculateSum(root);
}

void disconnect(SplayTree* root, SplayTree* child) {
    bool isRight = (root->right == child);
    (isRight ? root->right : root->left) = NULL;
    if (child)
        child->parent = NULL;
    recalculate(root);
}

void checkIsNull(SplayTree* root) {
    if (!root)
        throw IsNotNullException(root);
}

void checkIsNullAndAssign(SplayTree* &root, SplayTree* child) {
    checkIsNull(root);
    root = child;
}

void connect(SplayTree* root, SplayTree* newChild, bool isRight) {
    checkIsNullAndAssign((isRight ? root->right : root->left), newChild);
    recalculate(root);
}
```

Деревья с неявным ключом: инфраструктура

- Инфраструктура реализована для splay – деревьев; однако реализация для декартовых деревьев практически идентична.
- Устройство split и merge будет отличаться для двух деревьев.
- Покажем, как изменять элемент и находить сумму на отрезке:

```
void assignInVertex(SplayTree* root, int key) {
    checkIsNull(root);
    root->key = key;
    recalculate(root);
}

SplayTree* assign(SplayTree* root, uint32 index, int key) {
    root = searchByIndex(root, index);
    assignInVertex(root, key);
    return root;
}
```

```
bool isSubsegment(uint32 indexLeft, uint32 indexRight, uint32 size) {
    return indexLeft <= indexRight && indexRight < size;
}

void checkIsSubsegment(uint32 indexLeft, uint32 indexRight, uint32 size) {
    if (!isSubsegment)
        throw IsNotSubsegmentException(indexLeft, indexRight, size);
}

typedef std::pair<SplayTree*, SplayTree*> TSplayTreePair;
struct Triple {
    SplayTree *first, *second, *third;
};

Triple subsegmentSplit(SplayTree* root, uint32 indexLeft, ui32 indexRight) {
    checkIsSubsegment(indexLeft, indexRight, getSize(root));
    Triple t;
    TSplayTreePair p = splitByIndex(root, indexLeft);
    t.first = p.first;
    p = splitByIndex(p.second, indexRight - indexLeft + 1);
    t.second = p.first;
    t.third = p.second;
    return t;
}

SplayTree* merge(const Triple &t) {
    return merge(first, merge(second ,third));
}

int calculateSum(SplayTree* &root, uint32 indexLeft, uint32 indexRight) {
    Triple t = subsegmentSplit(root);
    int ans = getSum(t.second);
    root = merge(t)
    return ans;
}
```

Задачи RSQ, RMQ: определение

- RSQ (Range Sum Query): дан массив A = A[0..n-1] из n чисел. Необходимо уметь обрабатывать два запроса:
  - 1) assign(index, newValue): присвоить числу A[index] значение newValue;
  - 2) findSum(l, r): найти сумму A[l] + A[l + 1] + … + A[r] на подотрезке A[l..r].
- RMQ (Range Min Query): дан массив A = A[0..n-1] из n чисел. Необходимо уметь обрабатывать два запроса:
  - 1) assign(index, newValue): присвоить числу A[index] значение newValue;
  - 2) findMin(l, r): найти min(A[l, A[l + 1], … , A[r] на подотрезке A[l..r]).
- Заметим, что с помощью декартовых/splay-деревьев мы научились выполнять предлагаемые операции за O(log n); исследуем и другие методы.

Static RSQ: частичные суммы

- Для начала рассмотрим задачу static RSQ, т.е. задачу, в которой не требуется выполнять присваивание, т.е. менять массив.
- Для успешной реализации static RSQ насчитаем массив частичных сумм:
  - partialSums[i] := A[0] + A[1] + … A[i], $0 \le i < n$.
- Такие суммы могут быть преднасчитаны за O(n) в силу того, что partialSums[i+1] = partialSums[i] + A[i+1].
- Сумму же на подотрезке [l, r] можно найти за O(1) как partialSums[r] – partialSum[l-1] (или partialSums[r], если l = 0).
- Такое решение асимптотически оптимально.

Static RMQ: разреженная таблица

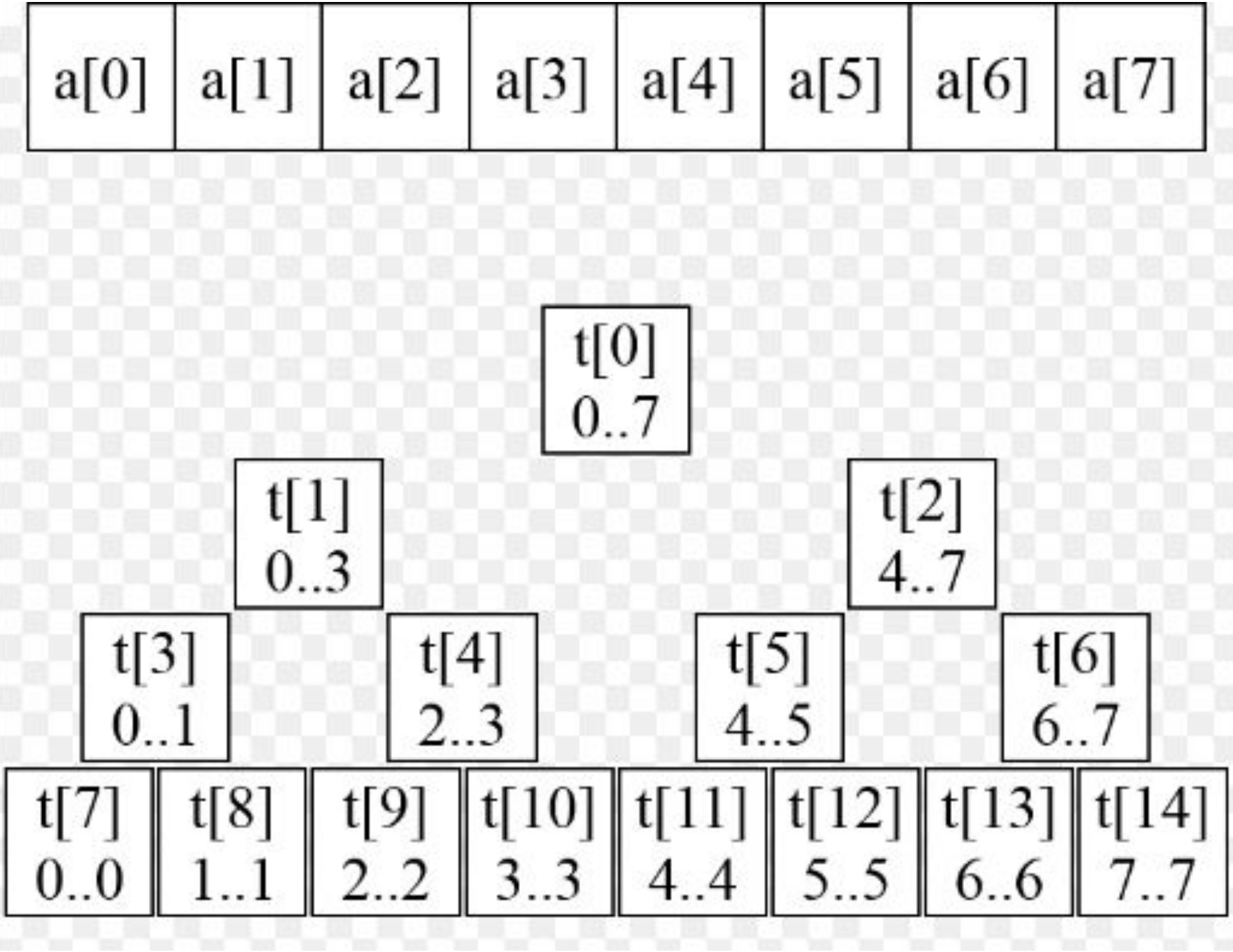
- min не является обратимой операцией, как операция суммы; вследствие этого, реализовать аналог частичных сумм для static RMQ не получится.
- Однако мы воспользуемся *идемпотентностью* операции min, т.е. тем фактом, что min(a, a).
- Насчитаем для массива A[0..n-1] разреженную таблицу (Sparse Table) sp[0..n-1][0..[$\log_2 n$]], где
  - sp[i][j] = min{a[k] | i ≤ k < min(i + 2^k, n)}.
- Это можно сделать за O(n log n), учитывая, что:

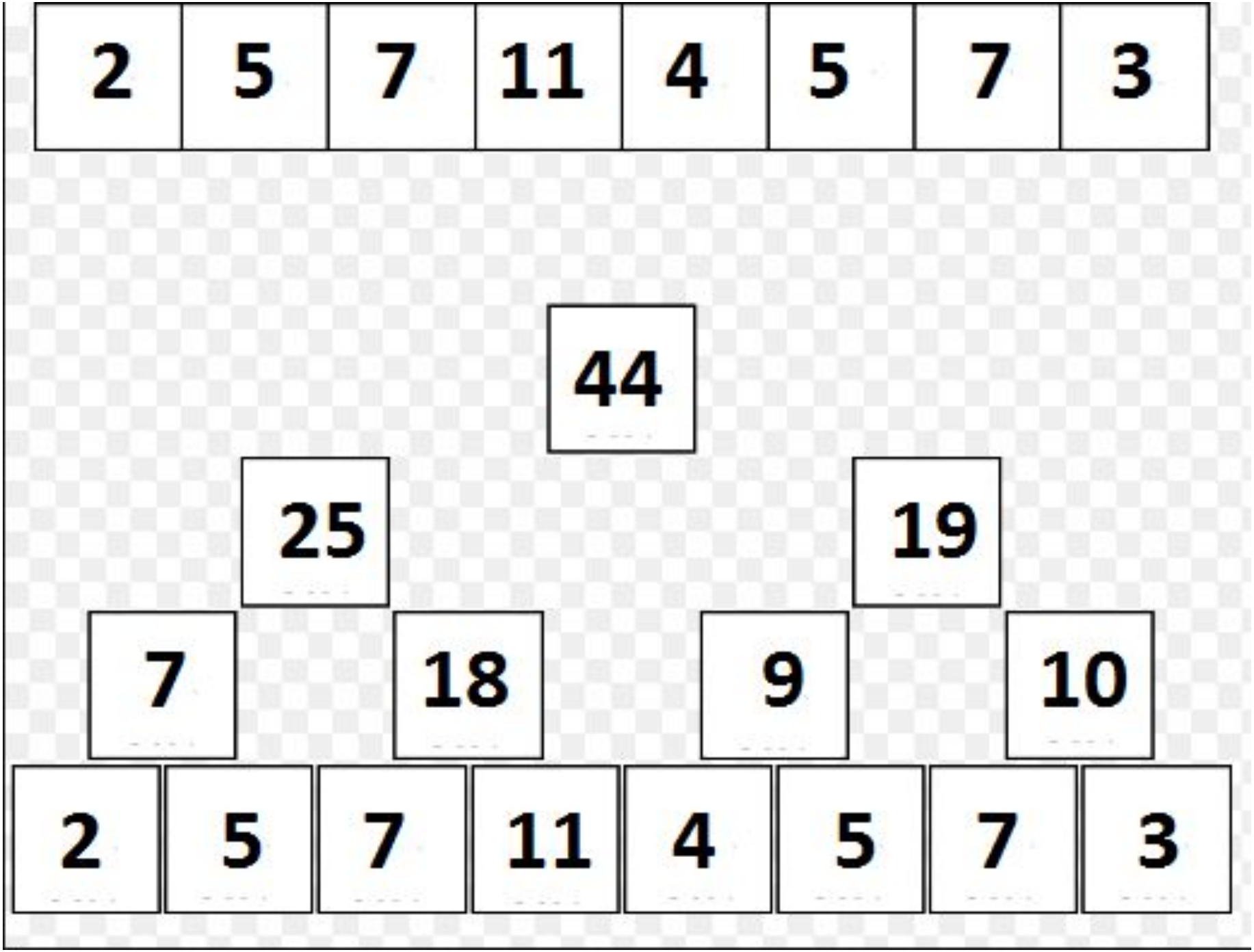
Static RMQ: разреженная таблица

- С помощью такой таблицы можно вычислять минимумы на подотрезке A[l, r] за O(1)!
- А именно:
  - Пусть lv = [$\log_2$ (r-l+1)]; другими словами, lv – это максимальное целое число, т.ч. 2^{lv} <= r-l+1.
  - Тогда min(A[l, r]) = min(minl, minr), где:
    - minl = min(A[l, l + 2^{lv} - 1]) = sp[l][lv];
    - minr = min(A[r-2^{lv}+1, r]) = sp[r-2^{lv}+1][lv];
- Таким образом, единственным нетривиальным действием оказывается вычисление логарифма.
- На практике, значение lv(x) вычисляют при предподсчете для всех x = 1, 2, …, n.
- Но что делать, если массив все-таки изменяется?

«Dynamic RSQ/RMQ»: дерево отрезков

- Чтобы успешно выполнять оба типа запросов, введем следующую структуру данных на отрезке (разберем дерево на примере RSQ, для RMQ структура строится полностью аналогично):
  - Предположим для удобства реализации, что $n = 2^k$;
  - Как и в разреженной таблице для RMQ, выберем некоторые подотрезки и будем хранить сумму в них.
  - Здесь это будут следующие отрезки:
    - [0, 0], [1, 1], …, [n-1, n-1] (n штук);
    - [0, 1], [2, 3], …, [n-2, n-1] (n/2 штук);
    - [0, 3], [4, 7], …, [n-4, n-1] (n/4 штук);
    - …
    - [0..n-1] (1 штука).
  - Всего хранимых отрезков ровно 2n-1.


a[0]
a[1]
a[2]
a[3]
a[4]
a[5]
a[6]
a[7]
t[0]
0..7
t[1]
0..3
t[2]
4..7
t[3]
0..1
t[4]
2..3
t[5]
4..5
t[6]
6..7
t[7]
0..0
t[8]
1..1
t[9]
2..2
t[10]
3..3
t[11]
4..4
t[12]
5..5
t[13]
6..6
t[14]
7..7

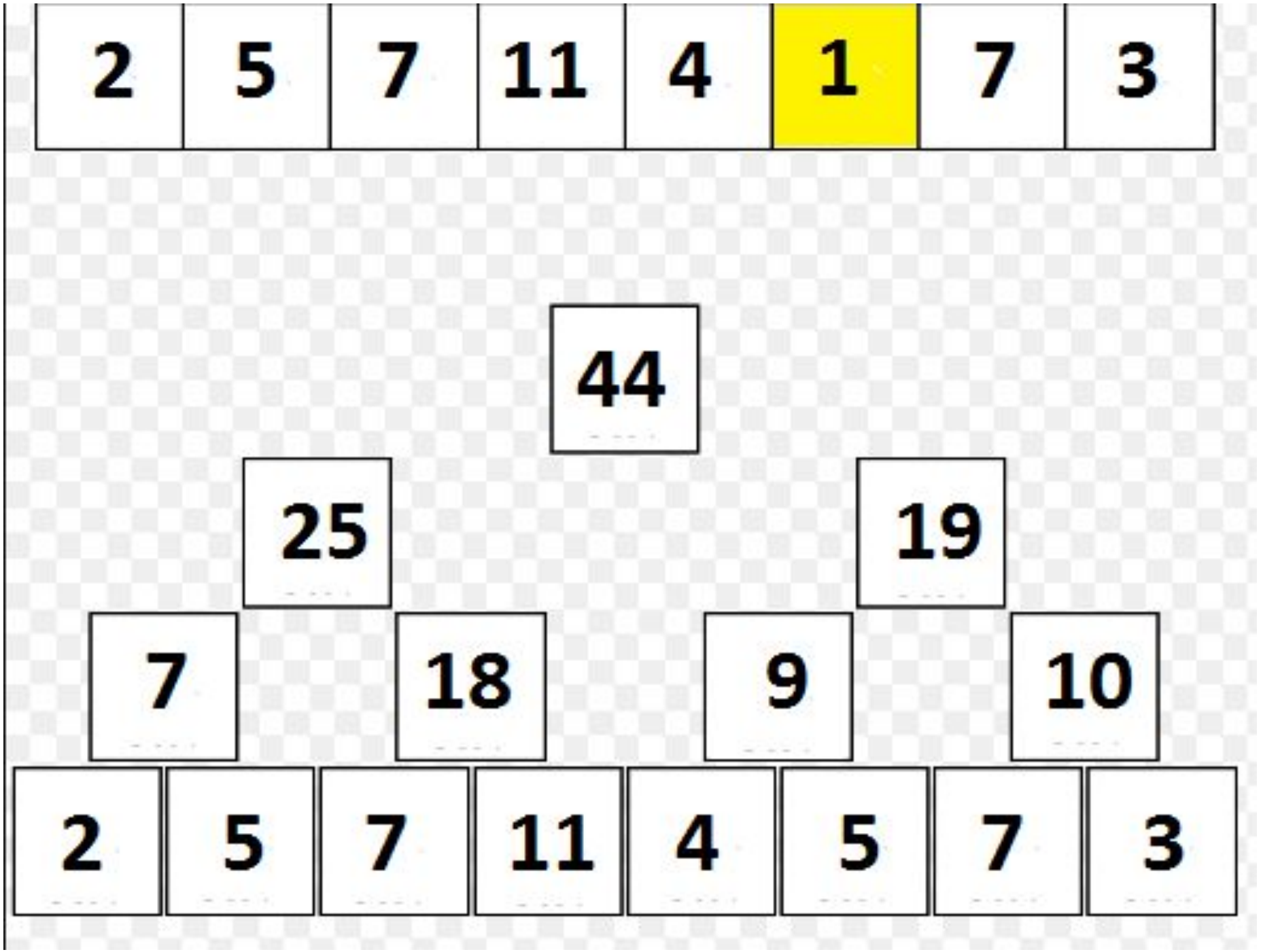

2 5 7 11 4 5 7 3
44
25 19
7 18 9 10
2 5 7 11 4 5 7 3

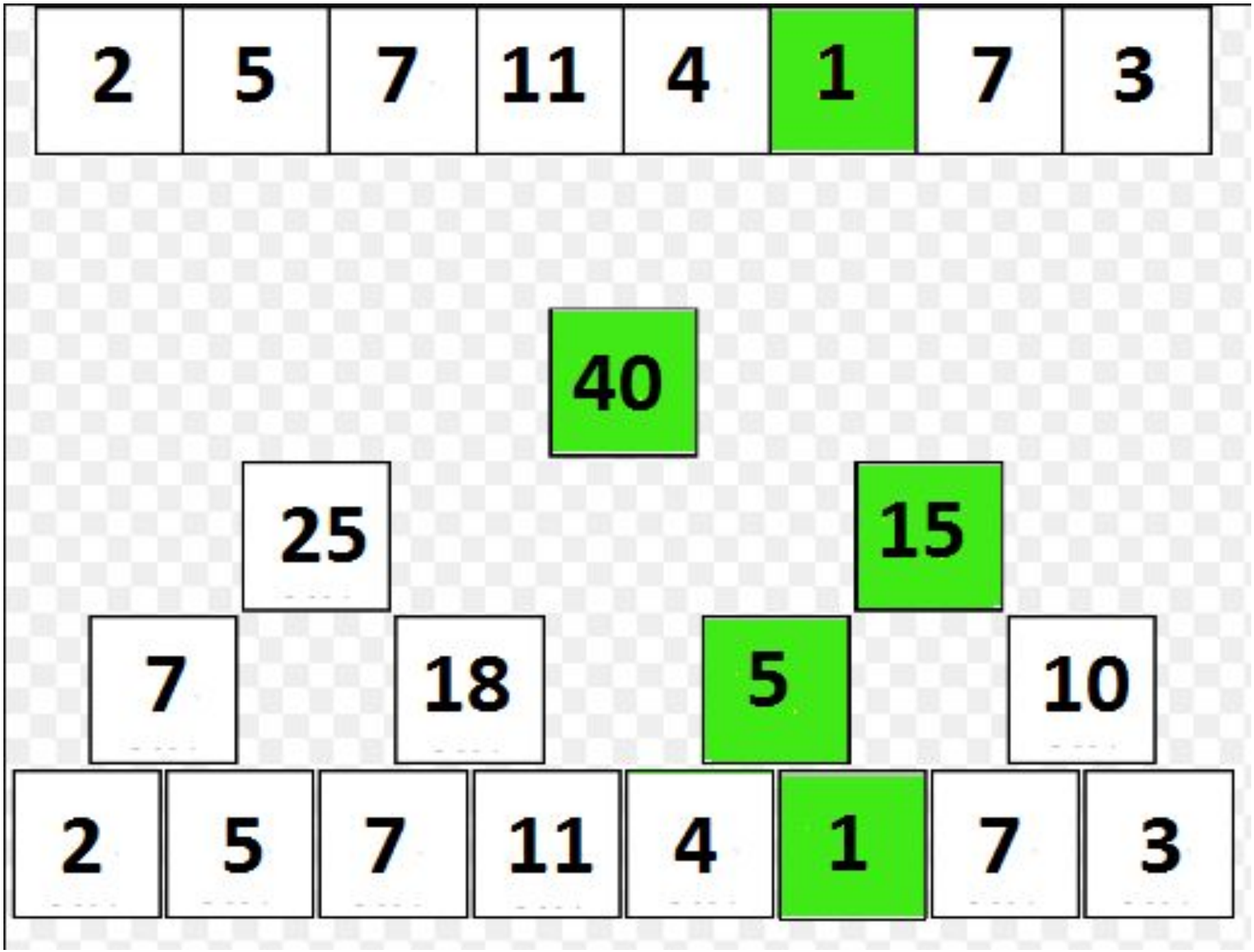
Дерево отрезков: общая структура

- Как видно из рисунка, все подотрезки хранятся в едином массиве t[0..2n-2] так, что:
  - вершине 0 соответствует подотрезок [0..n-1];
  - вершинам n-1, n-2, …, 2n-2 соответствуют подотрезки-элементы массива [0..0], [1..1], …, [n-1..n-1];
  - У вершины с номером v < n-1 есть два потомка 2v+1 и 2v+2, т.ч. если вершине v соответствует подотрезок [l..r], то детям соответствуют подотрезки [l..mid] и [mid+1..r], где mid = [(l+r)/2].
  - У вершины с номером v > 0 есть предок [(v-1)/2].
- Но как же на такой структуре выполнять операции?

Дерево отрезков: присваивание

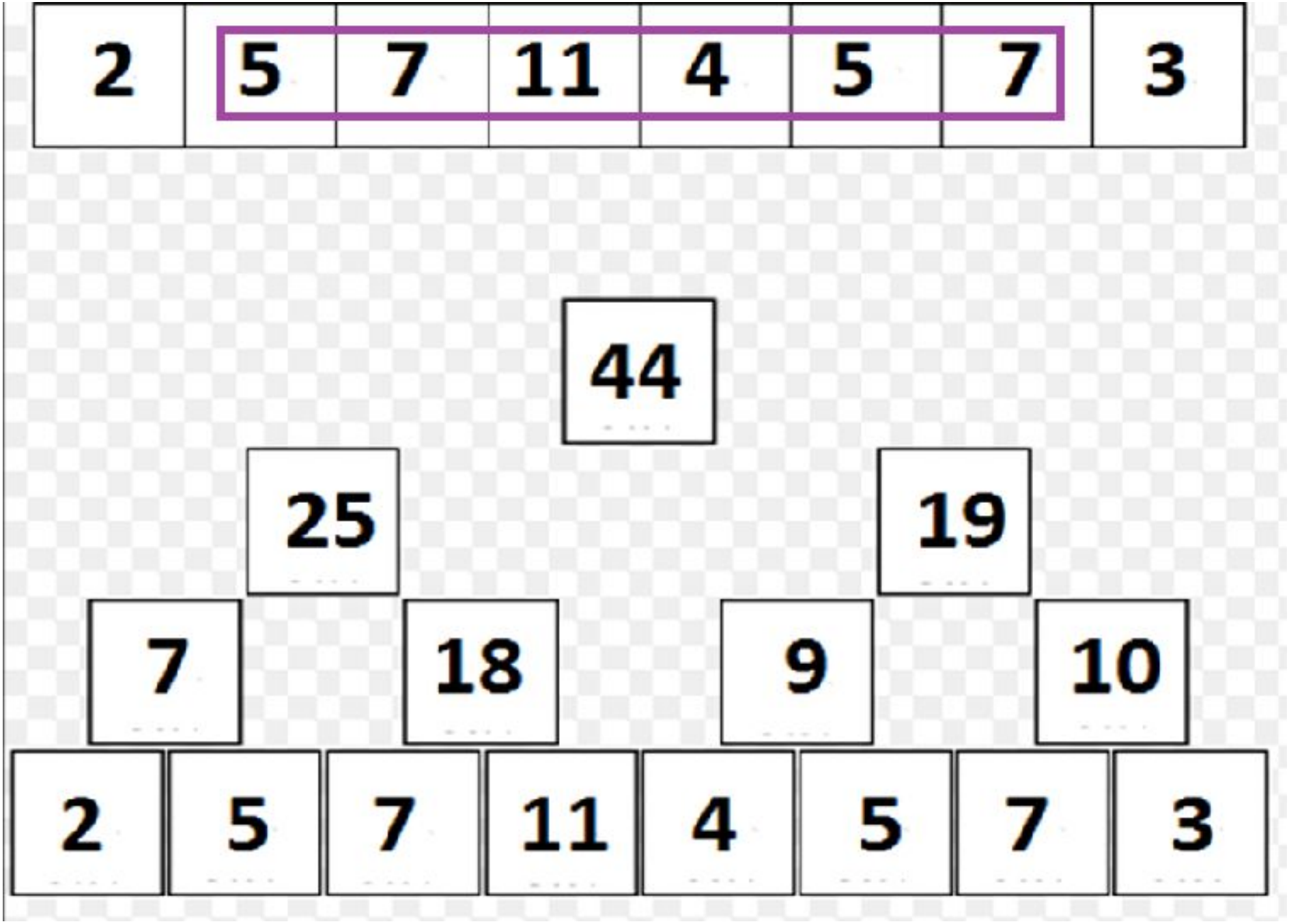
- Пусть пришел запрос на изменение параметра num;
- Тогда в дереве изменятся значения в вершине num + n – 1 и в её предках;
- Таких вершин ровно $[\log_2 n] + 1$, и только в них изменятся значения; следовательно, обновить дерево можно за O(log n):

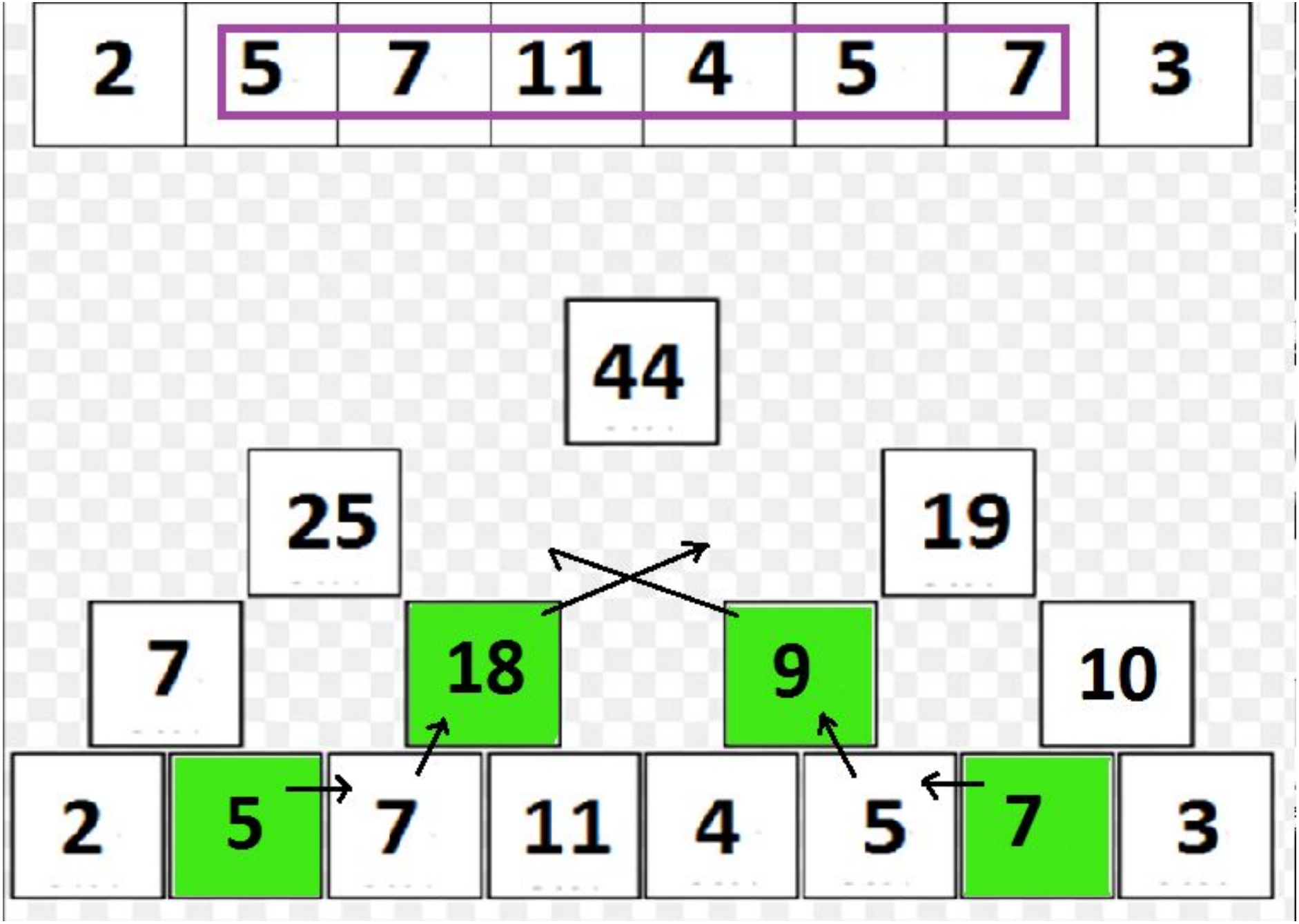

2 5 7 11 4 1 7 3
44
25 19
7 18 9 10
2 5 7 11 4 5 7 3

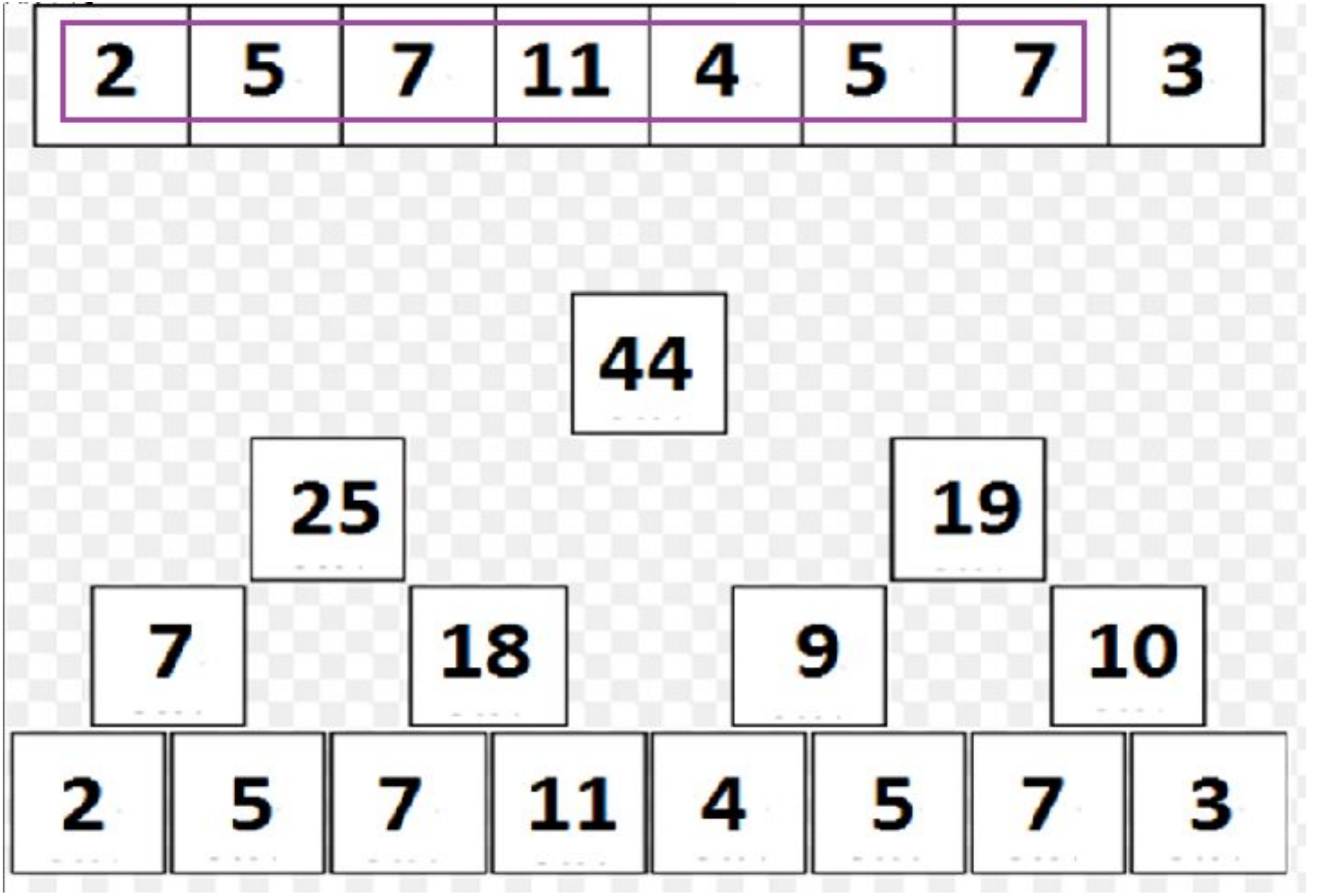

2
5
7
11
4
1
7
3
40
25
15
7
18
5
10
2
5
7
11
4
1
7
3

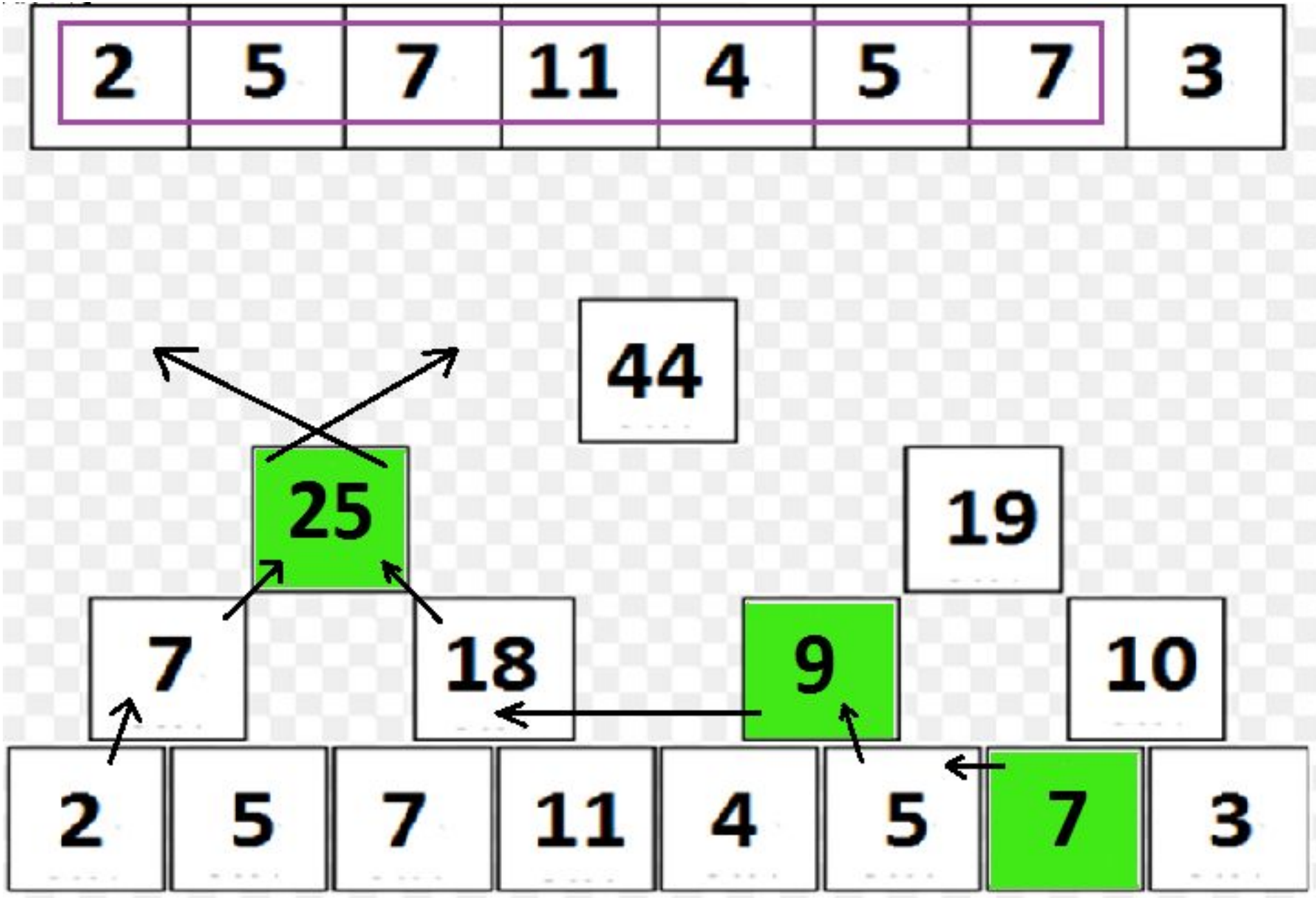
Дерево отрезков: сумма на подотрезке

- Но как же найти сумму?
- Способ «снизу», «нерекурсивный»:
  - Пусть нужно найти sum(A[l, r]), r > l (r = l - очевидно).
  - Заведем переменную ans для хранения текущего ответа; изначально, ans = 0
  - Заметим, что все элементы подотрезка, кроме, возможно, крайних, «покрываются» представленными в структуре отрезками длины 2, являющимися подотрезками отрезка [l, r];
  - Более того, l-ый элемент покрыт таким отрезком, если и только если l % 2 == 1, а r-ый – если и только если r % 2 == 0.
  - Учтя, если нужно, l-ый и r-ый элемент в ans, перейдем на уровень выше, где повторим рассуждения.
  - На каждом уровне выполняется O(1) действий; следовательно, сумму удается вычислить за $O(\log_2 n)$ действий!


2
5
7
11
4
5
7
3
44
25
19
7
18
9
10
2
5
7
11
4
5
7
3


2
5
7
11
4
5
7
3
44
25
19
7
18
9
10
2
5
7
11
4
5
7
3


2 5 7 11 4 5 7 3
44
25 19
7 18 9 10
2 5 7 11 4 5 7 3

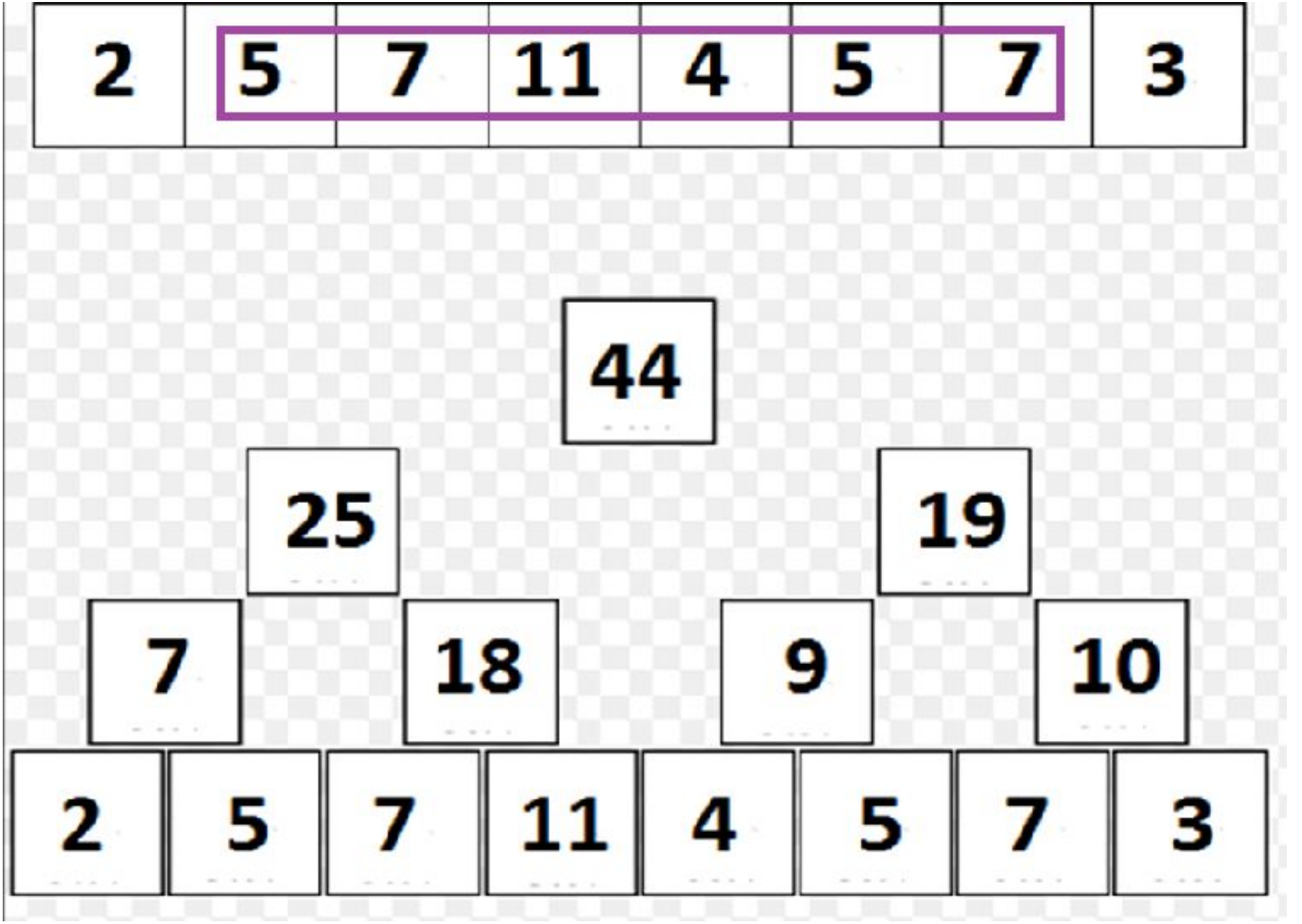

2
5
7
11
4
5
7
3
44
25
19
7
18
9
10
2
5
7
11
4
5
7
3

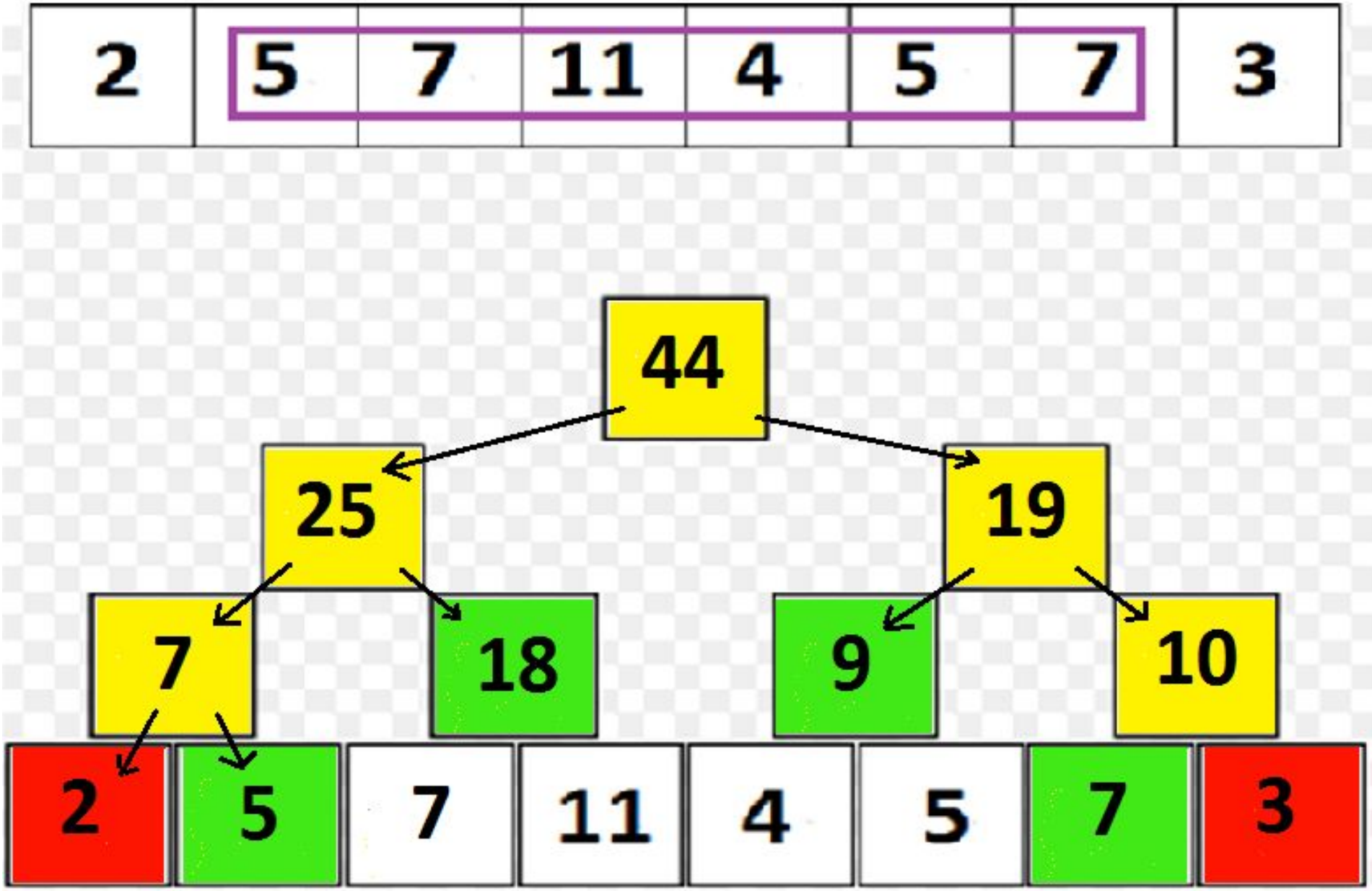
```
uint32 n; // size of array a
uint32 shift; // start of 1-length segment vertices in tree
std::vector<int> a(n), tree;

int getParent(int v) {
    return (v - 1) / 2;
}

int recalculate(int v) {
    tree[v] = tree[2*v+1] + tree[2*v+2];
}

uint32 calculateShift(uint32 n) {
    uint32 shift = 1;
    while (n > shift)
        shift += shift;
    --shift;
    return shift;
}

void buildTree() {
    shift = calculateShift(n);
    tree.resize(2*shift + 1);
    std::copy(a.begin(), a.end(), tree.begin() + shift);
    for (int i = static_cast<int>(shift) - 2; i >= 0; --i)
        recalculate(i);
}
```

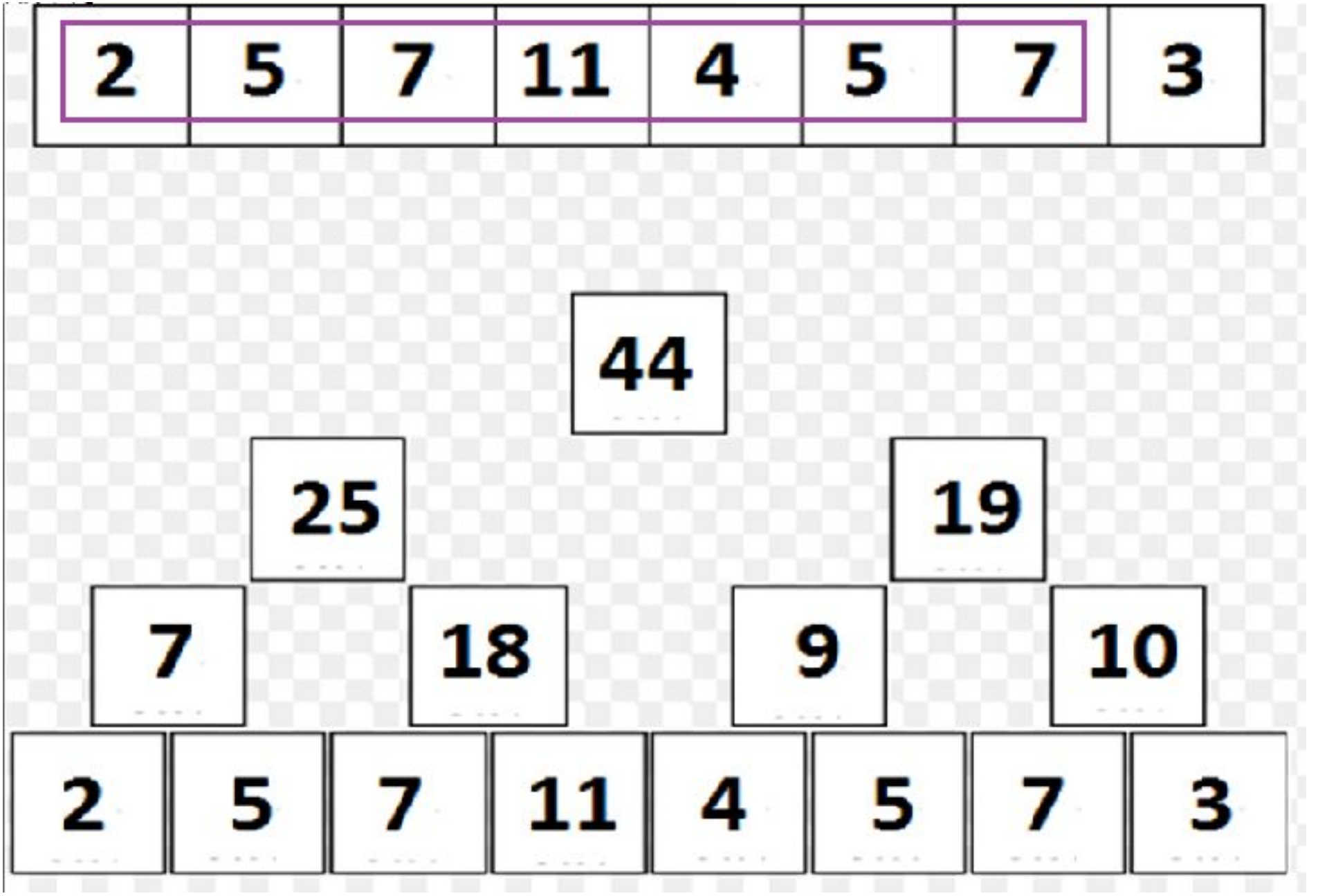
```
void assign(uint32 index, int x) {
    a[index] = x;
    int v = index + step - 1;
    tree[v] = x;
    for (v = getParent(v); v >= 0; v = getParent(v)
        recalculate(v);
}

int findSum(uint32 l, uint32 r) {
    int ans = 0;
    l += shift;
    r += shift;
    while (l <= r) {
        if (!(l&1))
            ans += tree[l++];
        if (r&1)
            ans += tree[r--];
        l = getParent(l);
        r = getParent(r);
    }
    return ans;
}
```

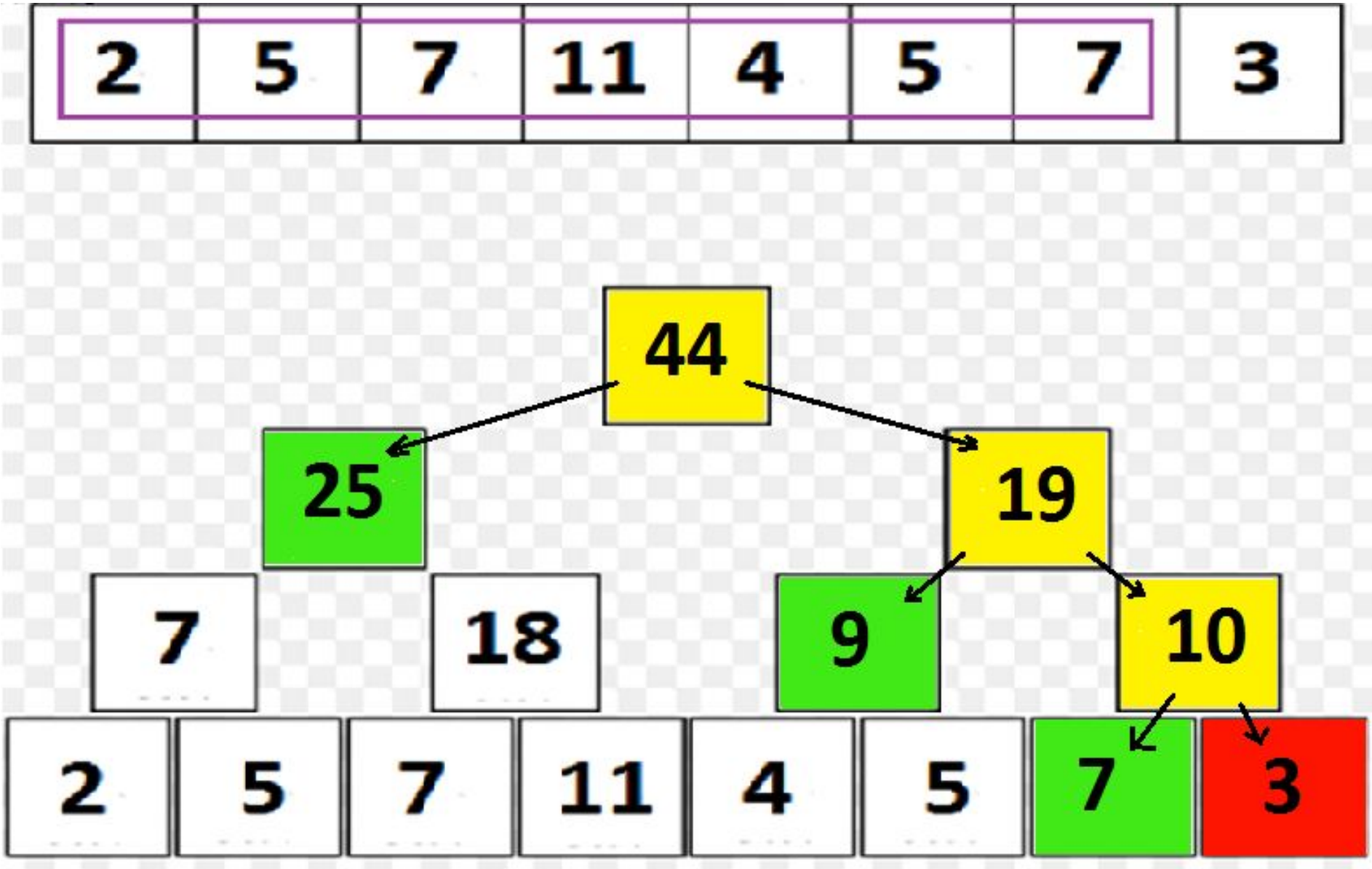
Дерево отрезков: операции «сверху»

- Рассмотрим теперь «рекурсивный» способ найти сумму sum(A[ql..qr]):
  - Будем идти, начиная с корня и запускаясь, если нужно, рекурсивно от детей.
  - Пусть в текущий момент времени мы находимся в вершине v, которой соответствует отрезок [l..r]:
    - Если [l..r] и [ql..qr] не пересекаются, то выходим из функции с суммой 0;
    - Если [l..r] $\subseteq$ [ql..qr], то возвращаем tree[v];
    - Иначе, запускаемся от детей и возвращаем сумму результатов запусков.


2
5
7
11
4
5
7
3
44
25
19
7
18
9
10
2
5
7
11
4
5
7
3


2
5
7
11
4
5
7
3
44
25
19
7
18
9
10
2
5
7
11
4
5
7
3

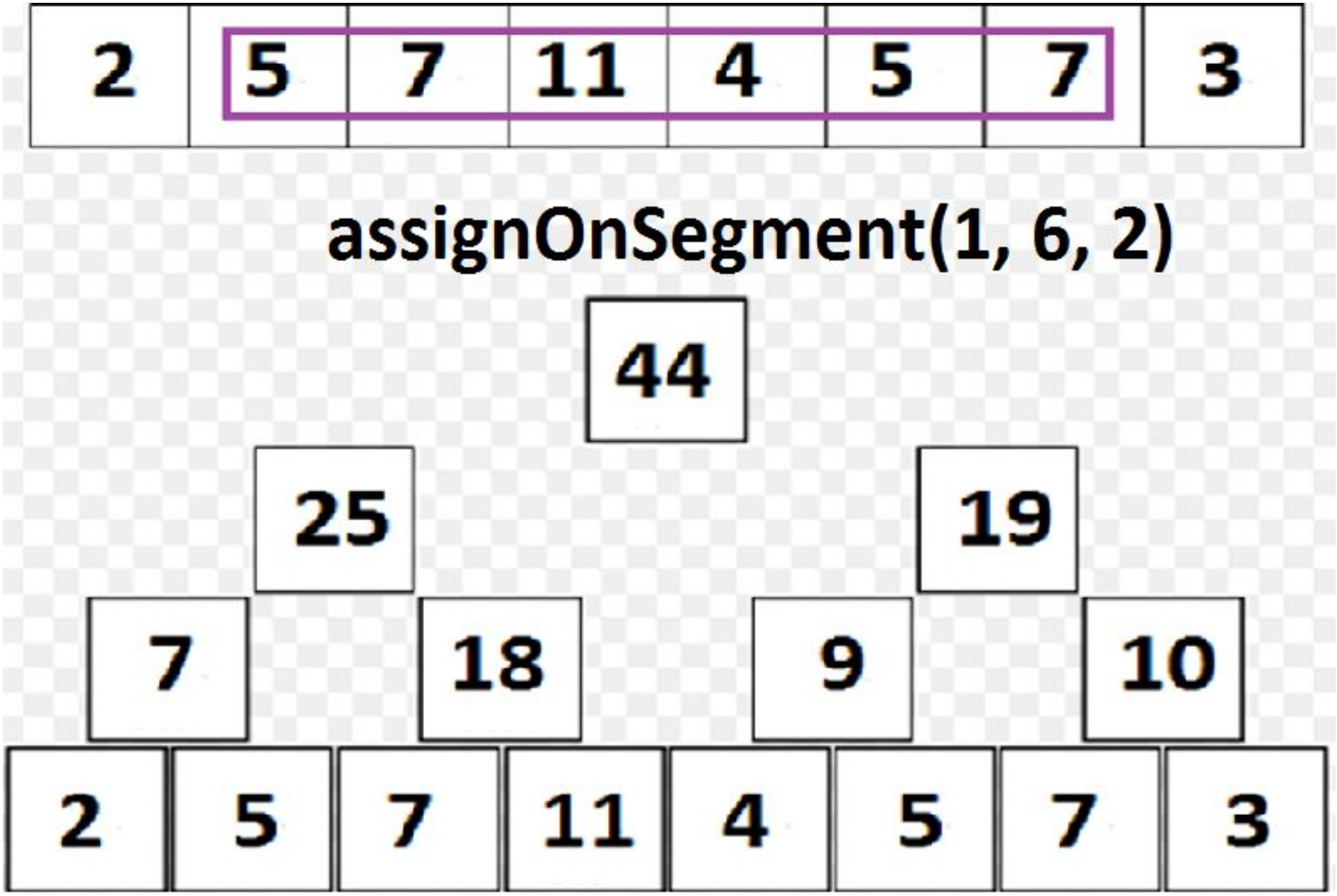

2 5 7 11 4 5 7 3
44
25 19
7 18 9 10
2 5 7 11 4 5 7 3

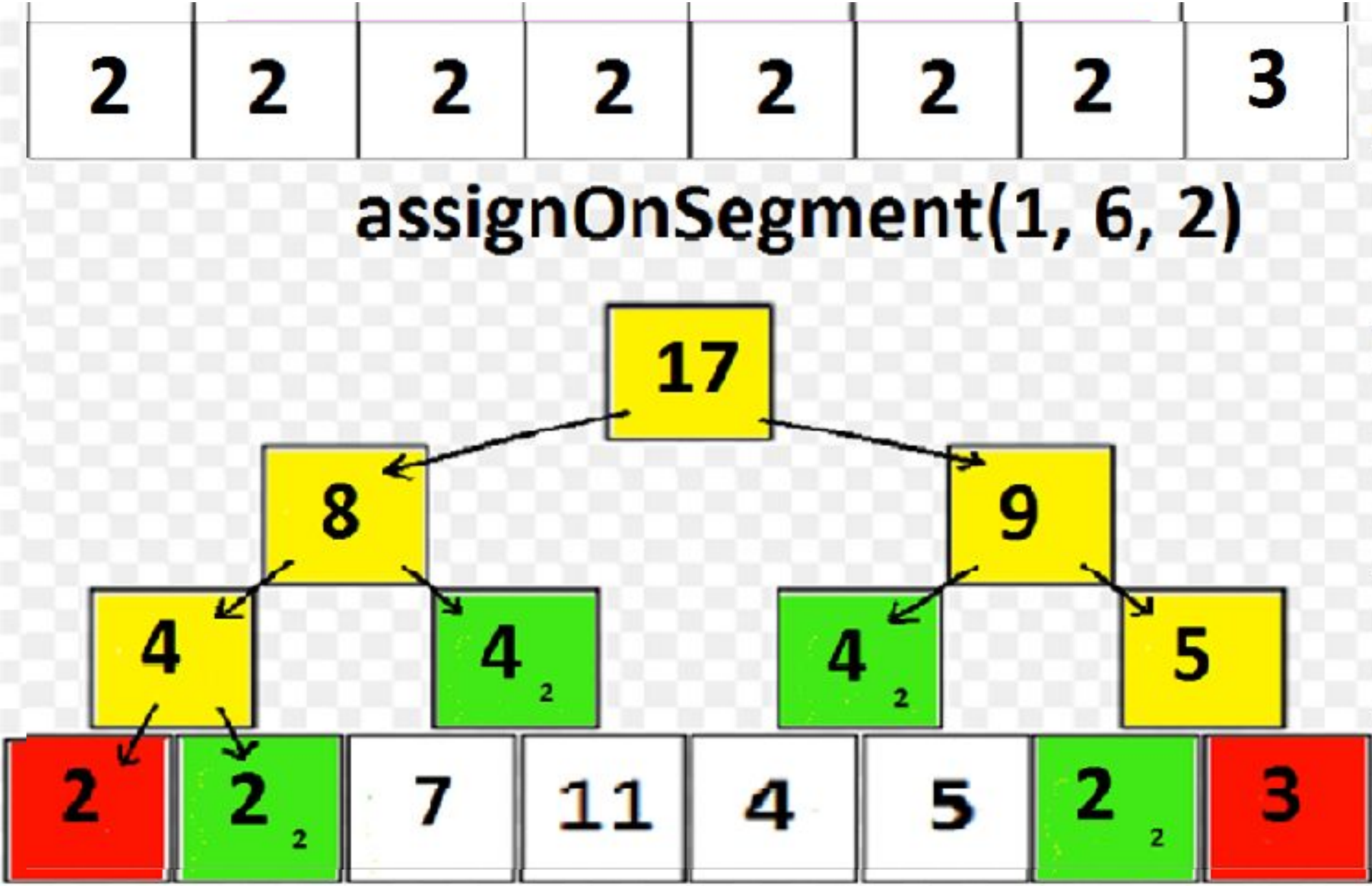

2
5
7
11
4
5
7
3
44
25
19
7
18
9
10
2
5
7
11
4
5
7
3

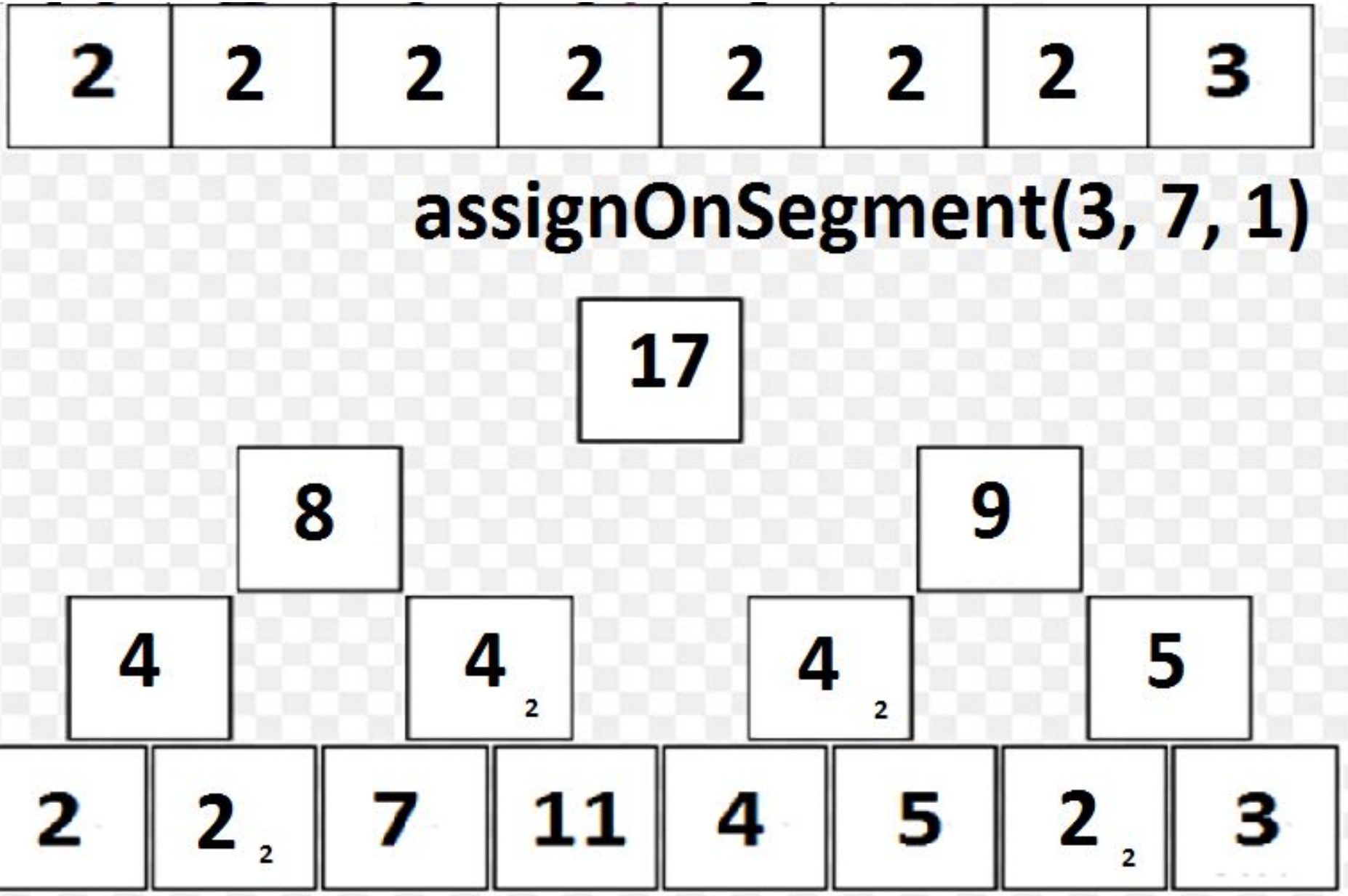
```
void assign(uint32 v, uint32 l, uint32 r, uint32 index, int x) {
    if (r < x || l > x)
        return;
    if (l == r) {
        tree[v] = x;
        return;
    }

    uint32 mid = (l + r) / 2;
    assign(2*v+1, l, mid, ql, qr);
    assign(2*v+2, mid+1, r, ql, qr);
    recalculate(v);
}

int findSum(uint32 v, uint32 l, uint32 r, uint32 ql, uint32 qr) {
    if (ql <= l && l <= r && r <= qr)
        return tree[v];
    if (qr < l || r < ql)
        return 0;
    uint32 mid = (l + r) / 2;
    return findSum(2*v+1, l, mid, ql, qr) + findSum(2*v+2, mid+1, r, ql, qr);
}
```

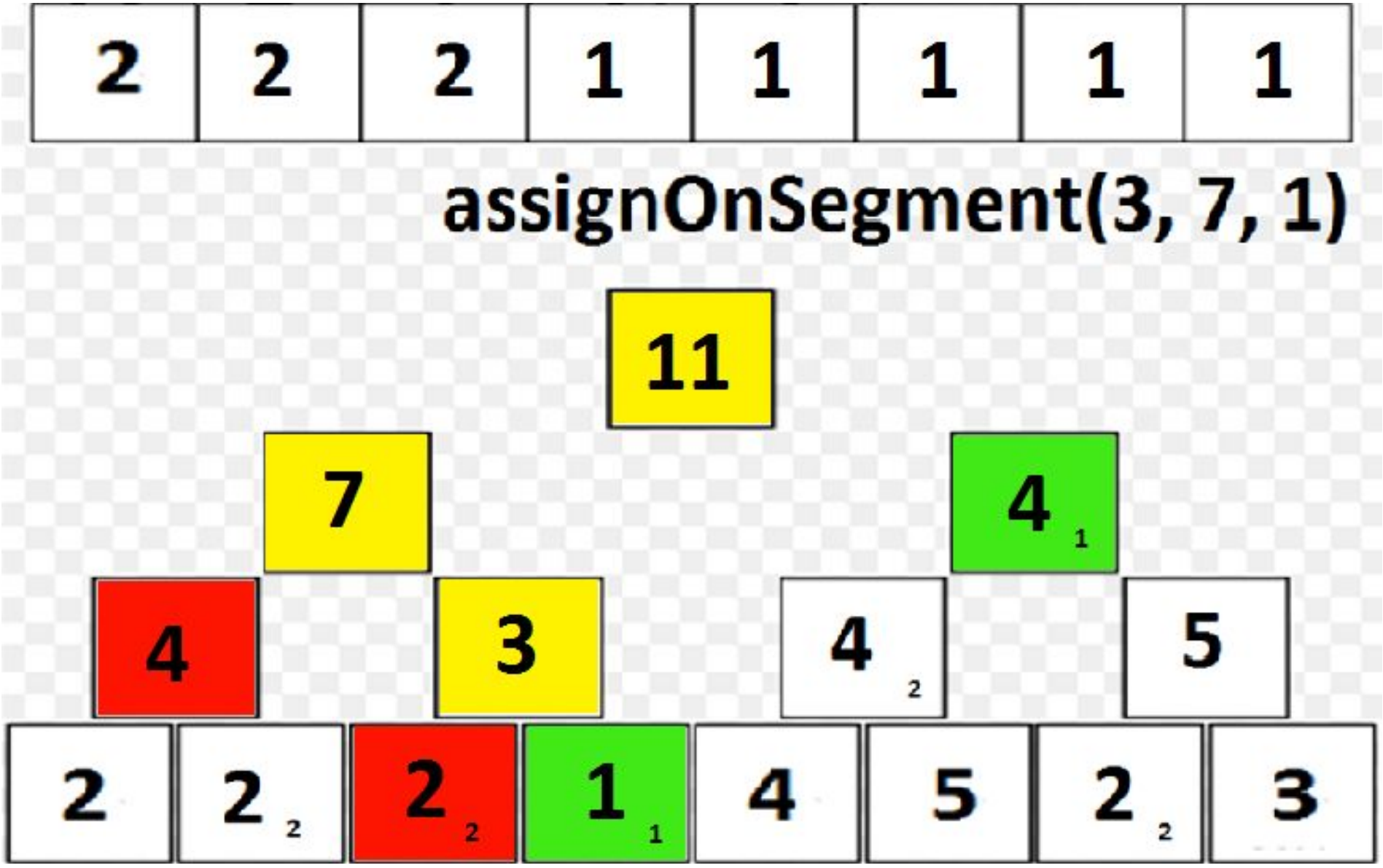
Дерево отрезков: множественные операции

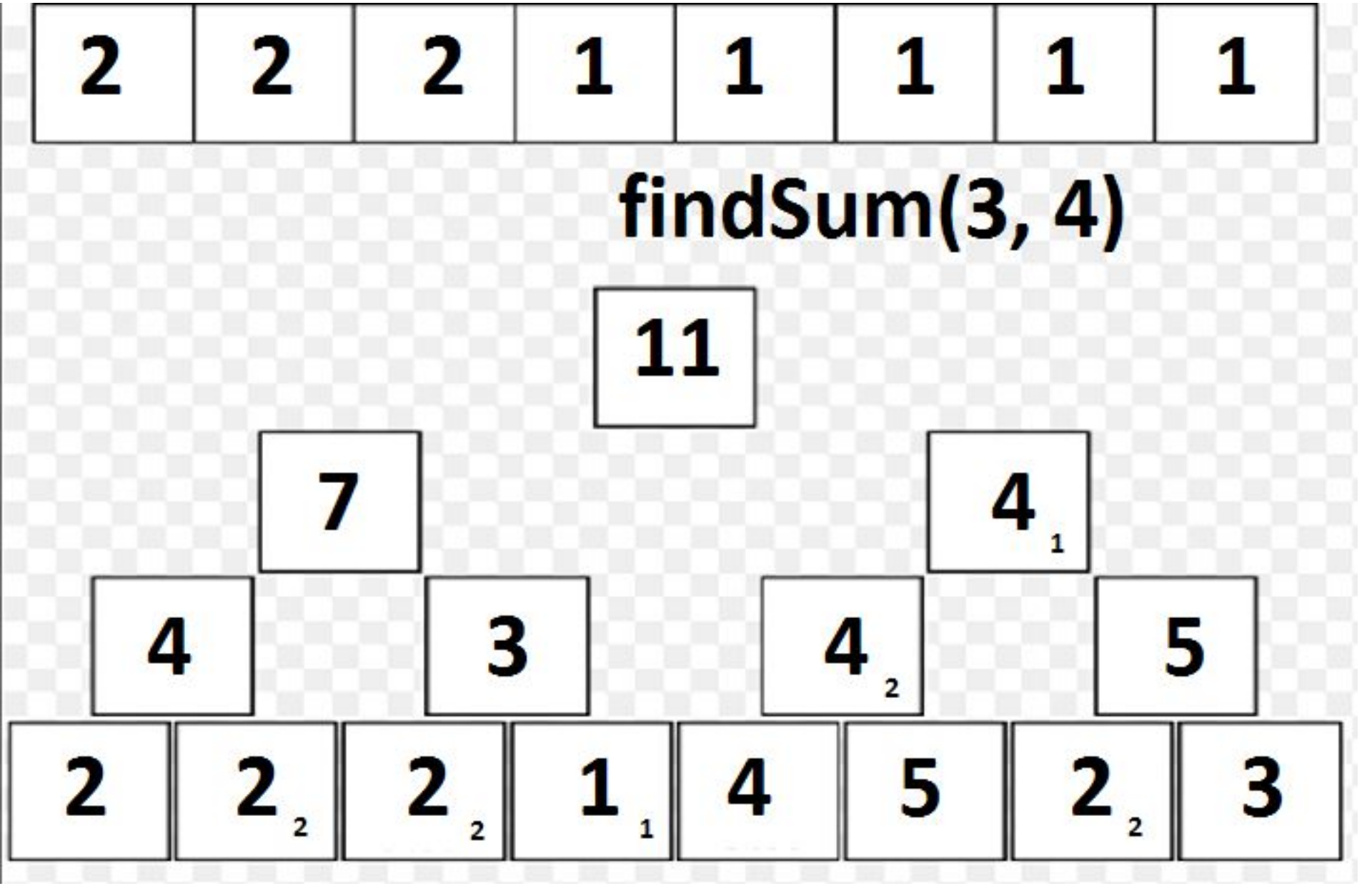
- Добавим к имеющимся двум операцию третью:
  - assignOnSegment(l, r, x), в которой требуется присвоить x элементам массива a[l], a[l+1], …, a[r].
- Чтобы поддерживать дерево отрезков в имеющемся виде, необходимо поменять O(n) вершин, что делает структуру бессмысленной.
- Чтобы сохранить эффективность, сделаем следующее:
  - Будем выполнять все операции «сверху»;
  - В каждой вершине v будем дополнительно хранить bool isAssigned и int valueAssigned;
  - Если isAssigned = false, то valueAssigned значения не имеет;
  - Если isAssigned = true, то valueAssigned несет в себе смысл вида «на самом деле, все значения из соответствующего вершине v отрезка массива равны valueAssigned; однако потомки v об этом еще не узнали.
  - assignOnSegment выполняем аналогично findSum; о присваивании «сообщаем» лишь «зеленым» вершинам;
  - Если в какой-либо операции нужно пройти в потомков v, а tree[v].isAssigned = true, то перед рекурсивным запуском нужно «сообщить» о valueAssigned потомкам, а в самой вершине v стереть упоминание о присваивании (tree[v].isAssigned := false); таким образом, реализуется принцип если мы работаем с вершиной v, то все операции, о которых v должна была узнать, она узнала».

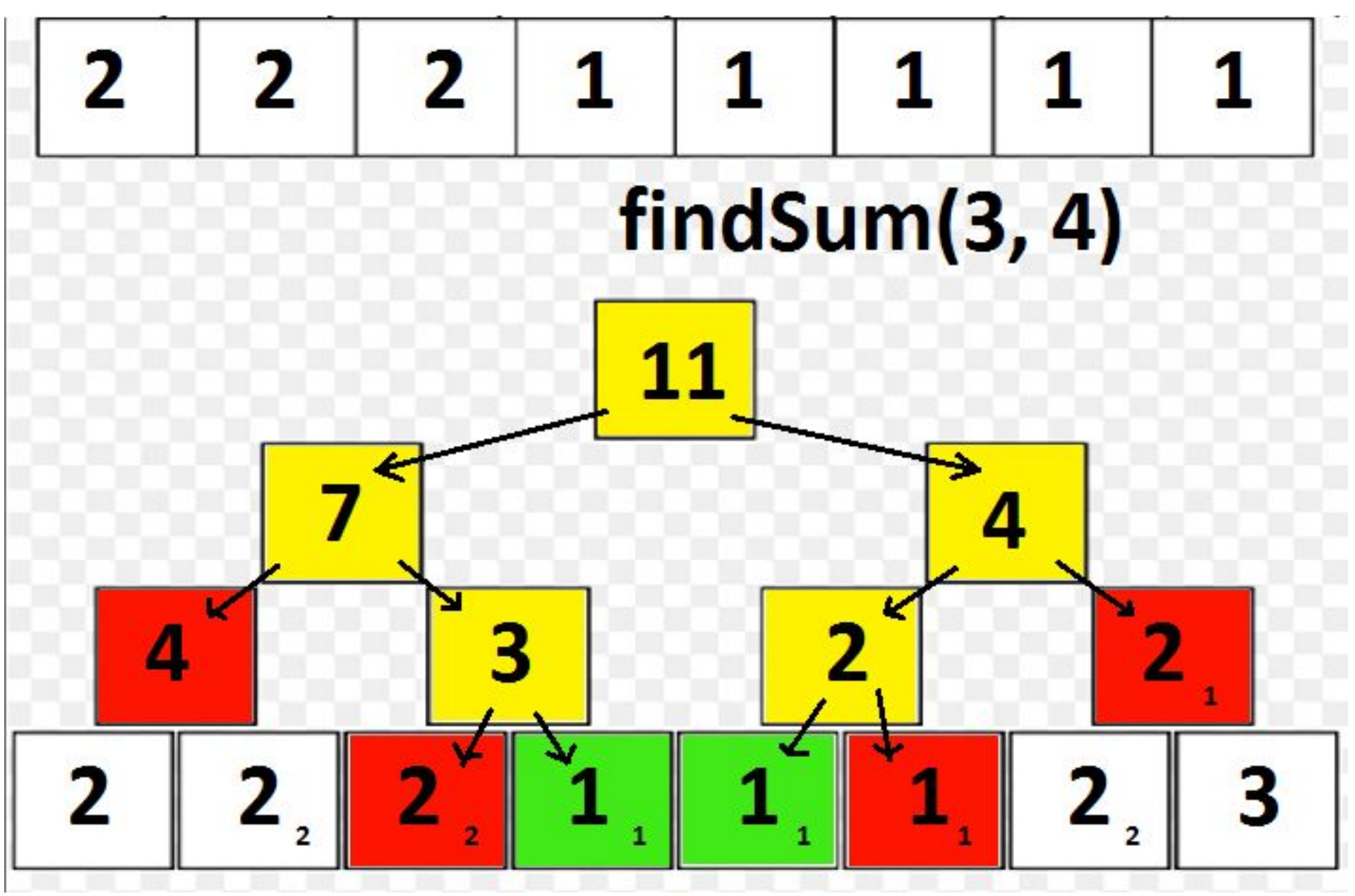

2
5
7
11
4
5
7
3
assignOnSegment(1, 6, 2)
44
25
19
7
18
9
10
2
5
7
11
4
5
7
3


2 2 2 2 2 2 2 3
assignOnSegment(1, 6, 2)
17
8
9
4
4 2
4 2
5
2
2 2
7
11
4
5
2 2
3


2 2 2 2 2 2 2 3
assignOnSegment(3, 7, 1)
17
8
9
4
4 2
4 2
5
2
2 2
7
11
4
5
2 2
3


2 2 2 1 1 1 1 1
assignOnSegment(3, 7, 1)
11
7
4 1
4
3
4 2
5
2 2 2 2 1 1 4 5 2 2 3


2 2 2 1 1 1 1 1
findSum(3, 4)
11
7
4 1
4
3
4 2
5
2 2 2 1 4 5 2 3
2 2 2 1 2 2


2 2 2 1 1 1 1 1
findSum(3, 4)
11
7
4
4
3
2
2 1
2
2 2
2 2
1 1
1 1
1 1
2 2
3

```
struct Node {
    int sum, assignedValue;
    bool isAssigned;
};

uint32 n; // size of array a
uint32 shift; // start of 1-length segment vertices in tree
std::vector<int> a(n);
std::vector<Node> tree;

int getParent(int v) {
    return (v - 1) / 2;
}

int recalculate(int v) {
    tree[v].sum = tree[2*v+1].sum + tree[2*v+2].sum;
}

uint32 calculateShift(uint32 n) {
    uint32 shift = 1;
    while (n > shift)
        shift += shift;
    --shift;
    return shift;
}
```

```
void buildTree() {
    shift = calculateShift(n);
    tree.resize(2*shift + 1);
    for (int i = 0; i < n; ++i)
        Node[shift + i] = Node(a[i], 0, false);

    for (int i = static_cast<int>(shift) - 2; i >= 0; --i) {
        recalculate(i);
        tree[i].isAssigned = false;
    }
}

void assignOnSubtree(uint32 v, uint32 l, uint32 r, int x) {
    tree[v].isAssigned = true;
    tree[v].assignedValue = x;
    tree[v].sum = x * (r - l + 1);
}

void pushDown(uint32 v, uint32 l, uint32 r) {
    if (!tree[v].isAssigned || v >= shift)
        return;
    int mid = (l + r)/2;
    assignOnSubtree(v*2+1, l, mid, tree[v].valueAssigned);
    assignOnSubtree(v*2+2, mid+1, r, tree[v].valueAssigned);
}
```

```
void assignOnSegment(uint32 v, uint32 l, uint32 r, uint32 ql, uint32 qr, int x) {
    if (r < ql || l > qr)
        return;
    if (ql <= l && r <= qr) {
        assignOnSubtree(v, l, r, x);
        return;
    }

    pushDown(v, l, r);
    uint32 mid = (l + r) / 2;
    assignOnSegment(2*v+1, l, mid, ql, qr);
    assignOnSegment(2*v+2, mid+1, r, ql, qr);
    recalculate(v);
}

int findSum(uint32 v, uint32 l, uint32 r, uint32 ql, uint32 qr) {
    if (ql <= l && l <= r && r <= qr)
        return tree[v];
    if (qr < l || r < ql)
        return 0;

    pushDown(v, l, r);
    uint32 mid = (l + r) / 2;
    return findSum(2*v+1, l, mid, ql, qr) + findSum(2*v+2, mid+1, r, ql, qr);
}
```

Дерево отрезков vs. splay/cartesian trees

- Итак, дерево отрезков умеет выполнять присваивание на отрезке и нахождение суммы на подотрезке, причем каждую операцию можно выполнять за O(log n);
- Также можно выполнять операции прибавления на подотрезке, хранить хэш подотрезка и многие другие операции;
- Заметим, что т.к. все операции в декартовых и splay-деревьях выполняются «сверху», то в этих деревьях практически в том же виде может быть применена технология отложенных операций, с тем же временем работы на операцию!

Дерево отрезков vs. splay/cartesian trees

- Преимущества дерева отрезков:
  - Низкая (по крайней мере, по сравнению с декартовыми деревьями) константа;
  - Асимптотика «честная», без вероятностей и потенциалов;
- Преимущества splay/cartesian trees:
  - Поддерживают split/merge/insert/delete (дерево отрезков работает с массивом фиксированной длины);
  - Более гибко: на этих деревьях можно реализовать операцию reverseOnSegment(l, r)!

splay/cartesian trees: reverseOnSegment

- Заметим, что для разворота, например, дерева SplayTree* tree, необходимо:
  - Поменять местами детей tree->left и tree->right;
  - Развернуть поддеревья детей tree->left и tree->right.
- Такую операцию легко сделать отложенной, храня в вершине флаг bool isReversed и проталкивать его!
- Однако будучи просто реализованной в деревьях, такую операцию не удается реализовать в ДО; таким образом, теоретически splay/cartesian tree превосходят

```
struct SplayTree {
    SplayTree *left, *right, *parent;
    int key, sum;
    bool isFlag;
    uint32 size;
};

void reverseOnSubtree(SplayTree* root) {
    if (!root)
        return;
    root->isReversed ^= 1;
}

void pushDown(SplayTree* root) {
    if (!root || !root->isReversed)
        return;
    std::swap(root->left, root->right)
    reverseOnSubtree(root->left);
    reverseOnSubtree(root->right);
    root->isReversed = false;
}
```

```
SplayTree* searchByIndex(SplayTree* root, uint32 index) {
    if (index > getSize(root) || index == 0)
        throw BadIndexException(index, getSize, root);

    while (true) {
        ui32 leftSize = root->size();
        if (leftSize == index)
            break;
        else {
            pushDown(root);
            if (leftSize > index)
                root = root->left;
            else {
                index -= leftSize + 1;
                root = root->right;
            }
        }
    }
    splay(root); // now all vertices on path from real root to
                 //current vertex iare not under delayed operations
    return root;
}

SplayTree* reverseOnSegment(SplayTree* root, uint32 l, uint32 r) {
    Triple t = subsegmentSplit(root);
    reverseOnSubtree(t.second);
    return merge(t);
}
```