

Treaps

One data structure to rule them all

Δημήτρης Μπαλάτος, Αγγελική Νταλαπέρα



Παρουσίαση — Κώδικες

<https://github.com/jbalatos/sfhmmy-treaps>



Ορισμοί

TREE T



Ορισμοί

TREE T

BINARY TREE $T = \langle key, L, R \rangle$

- L : Αριστερό παιδί (επίσης Binary Tree)
- R : Δεξί παιδί (επίσης Binary Tree)



Ορισμοί

TREE T

BINARY TREE $T = \langle key, L, R \rangle$

- L : Αριστερό παιδί (επίσης Binary Tree)
- R : Δεξί παιδί (επίσης Binary Tree)

BINARY SEARCH TREE

- L, R επίσης Binary Search Trees
- $L.key \leq key \leq R.key$



Ορισμοί

TREE T

BINARY TREE $T = \langle key, L, R \rangle$

- L : Αριστερό παιδί (επίσης Binary Tree)
- R : Δεξί παιδί (επίσης Binary Tree)

BINARY SEARCH TREE

- L, R επίσης Binary Search Trees
- $L.key \leq key \leq R.key$

BINARY (MAXIMUM) HEAP

- L, R επίσης Binary Heaps
- $key \geq L.key, R.key$ (ή αντίστροφα για Minimum Heaps)



Ορισμοί

TREE T

BINARY TREE $T = \langle key, L, R \rangle$

- L : Αριστερό παιδί (επίσης Binary Tree)
- R : Δεξί παιδί (επίσης Binary Tree)

BINARY SEARCH TREE

- L, R επίσης Binary Search Trees
- $L.key \leq key \leq R.key$

TREAP $T = \langle \underline{key}, \underline{priority}, L, R \rangle$

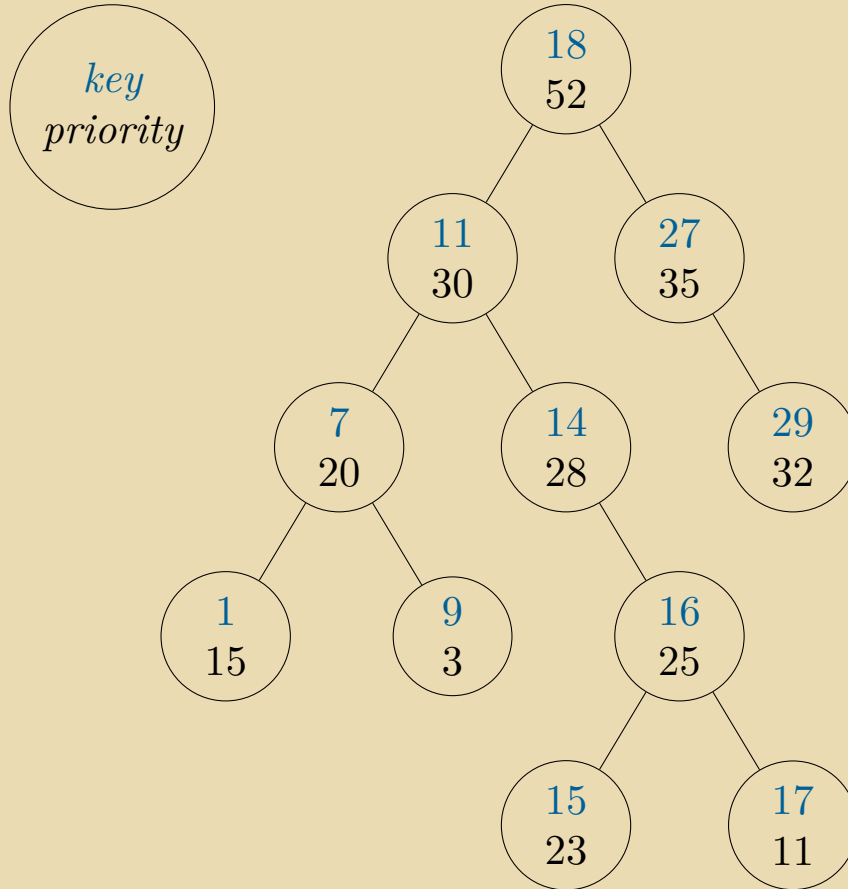
- Binary Search Tree ως προς key
- Δυαδικός Σωρός ως προς $priority$

BINARY (MAXIMUM) HEAP

- L, R επίσης Binary Heaps
- $key \geq L.key, R.key$ (ή αντίστροφα για Minimum Heaps)



Παράδειγμα



Βασικές Συναρτήσεις

SPLIT(T, key) "Σπάει" το T σε 2 treaps
 L, R έτσι ώστε $L \leq key \leq R$.



Βασικές Συναρτήσεις

SPLIT(T, key) "Σπάει" το T σε 2 treaps
 L, R έτσι ώστε $L \leq key \leq R$.

MERGE(L, R) "Ενώνει" τα L, R
δεδομένου ότι $L \leq R$.

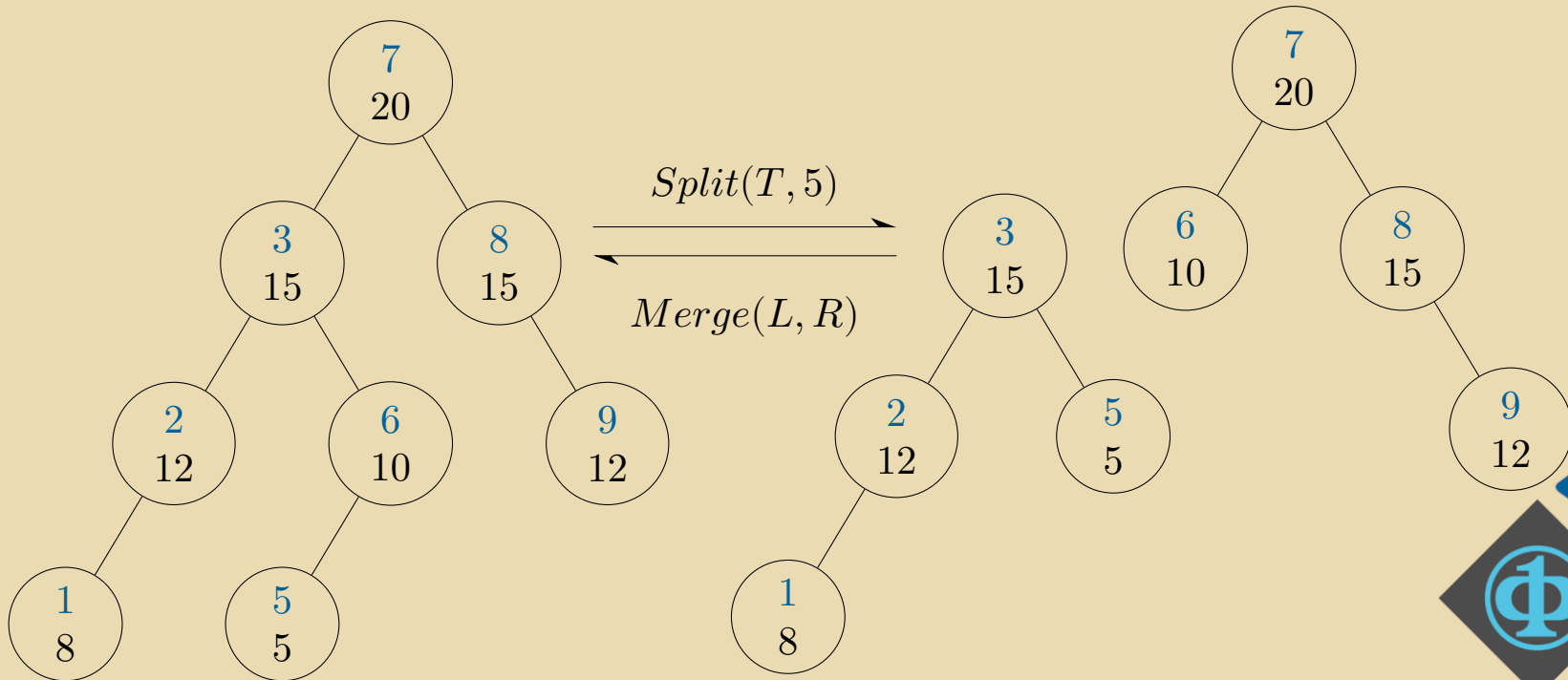


Βασικές Συναρτήσεις

SPLIT(T, key) "Σπάει" το T σε 2 treaps
 L, R έτσι ώστε $L \leq key \leq R$.

MERGE(L, R) "Ενώνει" τα L, R
δεδομένου ότι $L \leq R$.

Ισχύει ότι:

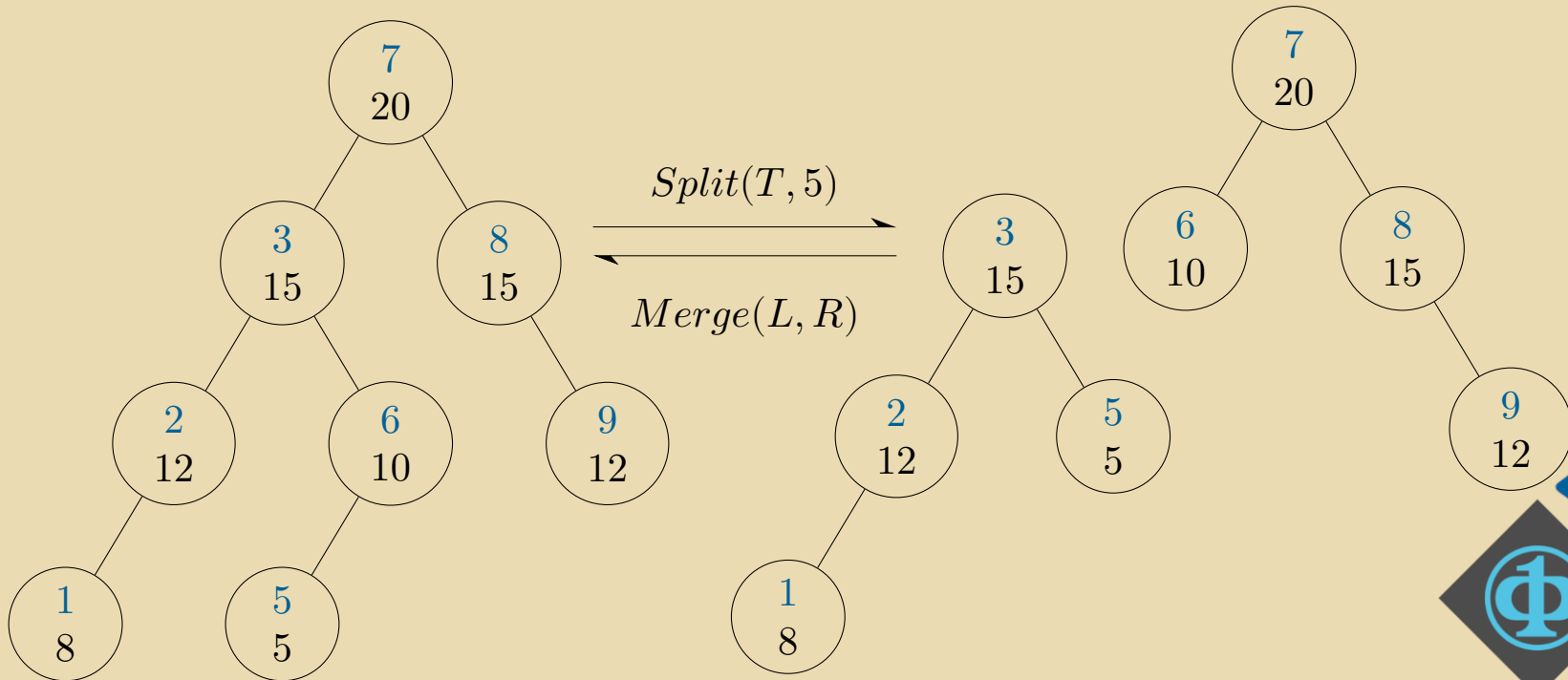


Βασικές Συναρτήσεις

SPLIT(T, key) "Σπάει" το T σε 2 treaps
 L, R έτσι ώστε $L \leq key \leq R$.

MERGE(L, R) "Ενώνει" τα L, R
δεδομένου ότι $L \leq R$.

Ισχύει ότι: $Split(T, key) = \langle L, R \rangle \iff T = Merge(L, R)$



Split

```
function Split( $T, x$ ):  $\langle L, R \rangle$   
  Ensure:  $\max\{L\} \leq x < \min\{R\}$   
  // Επιλογή: Σε ποιο treap θα ανήκει ο κόμβος  $T$ ;
```



Split

```
function Split( $T, x$ ):  $\langle L, R \rangle$   
  Ensure:  $\max\{L\} \leq x < \min\{R\}$   
  // Επιλογή: Σε ποιο treap θα ανήκει ο κόμβος  $T$ ;  
  
  if  $T.key \leq x$  then  
    // (1) στο  $L \rightarrow$  όλο το  $T.R$  θα ανήκει στο  $R$   
     $L, R \leftarrow \text{Split}(T.R, x)$   
     $T.R := L$   
  return  $\langle T, R \rangle$ 
```



Split

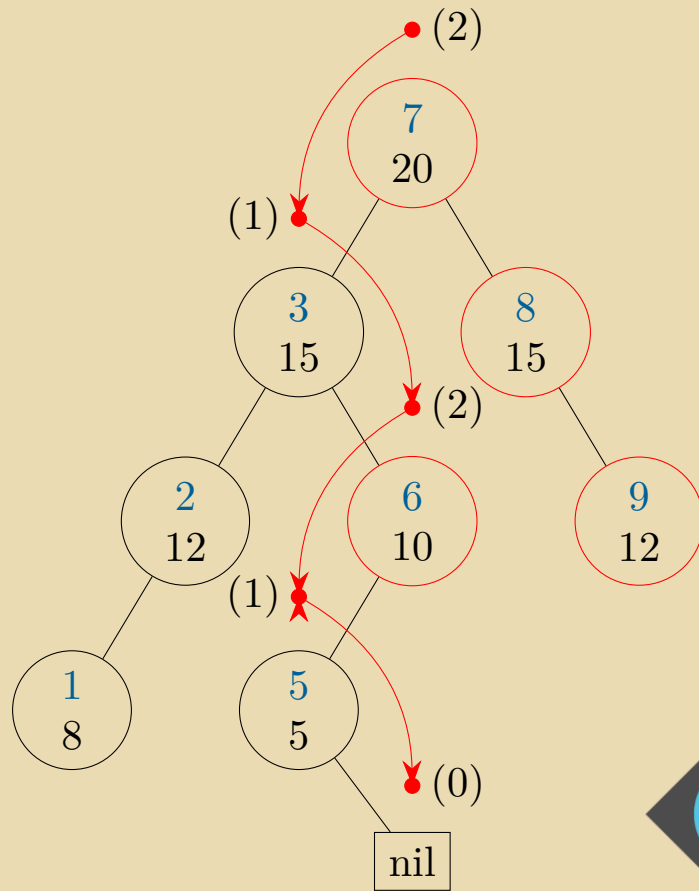
```
function Split( $T, x$ ):  $\langle L, R \rangle$   
  Ensure:  $\max\{L\} \leq x < \min\{R\}$   
  // Επιλογή: Σε ποιο treap θα ανήκει ο κόμβος  $T$ ;  
  
  if  $T.key \leq x$  then  
    // (1) στο  $L \rightarrow$  όλο το  $T.R$  θα ανήκει στο  $R$   
     $L, R \leftarrow \text{Split}(T.R, x)$   
     $T.R := L$   
    return  $\langle T, R \rangle$   
  else  
    // (2) στο  $R \rightarrow$  όλο το  $T.L$  θα ανήκει στο  $L$   
     $L, R \leftarrow \text{Split}(T.L, x)$   
     $T.L := R$   
    return  $\langle L, T \rangle$   
end if
```



Correct Split

```
function Split( $T, x$ ):  $\langle L, R \rangle$   
  Ensure:  $\max\{L\} \leq x < \min\{R\}$   
  // Επιλογή: Σε ποιο treap θα ανήκει ο κόμβος  $T$ ;  
  if  $T = \emptyset$  then  
    // (0)  
    return  $\langle \emptyset, \emptyset \rangle$   
  else if  $T.key \leq x$  then  
    // (1) στο  $L \rightarrow$  όλο το  $T.R$  θα ανήκει στο  $R$   
     $L, R \leftarrow \text{Split}(T.R, x)$   
     $T.R := L$   
    return  $\langle T, R \rangle$   
  else  
    // (2) στο  $R \rightarrow$  όλο το  $T.L$  θα ανήκει στο  $L$   
     $L, R \leftarrow \text{Split}(T.L, x)$   
     $T.L := R$   
    return  $\langle L, T \rangle$   
  end if
```

Σχήμα 1 Split($T, 5$)



Merge



Merge

(Immediately correct)



Merge

(Immediately correct)

```
function Merge( $L, R$ ):  $T$   
  Require:  $\max\{L\} \leq \min\{R\}$   
  // Επιλογή: Ποια ρίζα θα είναι ρίζα του  $T$ ;
```



Merge

(Immediately correct)

```
function Merge( $L, R$ ):  $T$   
  Require:  $\max\{L\} \leq \min\{R\}$   
  // Επιλογή: Ποια ρίζα θα είναι ρίζα του  $T$ ;  
  
  if  $L = \emptyset$  or  $R = \emptyset$  then  
    return non-empty  $L, R$ 
```



Merge

(Immediately correct)

```
function Merge( $L, R$ ):  $T$   
  Require:  $\max\{L\} \leq \min\{R\}$   
  // Επιλογή: Ποια ρίζα θα είναι ρίζα του  $T$ ;  
  
  if  $L = \emptyset$  or  $R = \emptyset$  then  
    return non-empty  $L, R$   
  else if  $L.priority > R.priority$  then  
    // (1) ρίζα είναι το  $L \rightarrow$  ενώνουμε  $L.R$  με  $R$   
     $L.R := \text{Merge}()$   
  return  $L$ 
```



Merge

(Immediately correct)

```
function Merge( $L, R$ ):  $T$ 
  Require:  $\max\{L\} \leq \min\{R\}$ 
  // Επιλογή: Ποια ρίζα θα είναι ρίζα του  $T$ ;

  if  $L = \emptyset$  or  $R = \emptyset$  then
    return non-empty  $L, R$ 
  else if  $L.priority > R.priority$  then
    // (1) ρίζα είναι το  $L \rightarrow$  ενώνουμε  $L.R$  με  $R$ 
     $L.R := \text{Merge}()$ 
    return  $L$ 
  else
    // (2) ρίζα είναι το  $R \rightarrow$  ενώνουμε  $R.L$  με  $L$ 
     $R.L := \text{Merge}(L, R.L)$ 
    return  $R$ 
end if
```



Merge

(Immediately correct)

```
function Merge( $L, R$ ):  $T$ 
  Require:  $\max\{L\} \leq \min\{R\}$ 
  // Επιλογή: Ποια ρίζα θα είναι ρίζα του  $T$ ;

  if  $L = \emptyset$  or  $R = \emptyset$  then
    return non-empty  $L, R$ 
  else if  $L.priority > R.priority$  then
    // (1) ρίζα είναι το  $L \rightarrow$  ενώνουμε  $L.R$  με  $R$ 
     $L.R := \text{Merge}()$ 
    return  $L$ 
  else
    // (2) ρίζα είναι το  $R \rightarrow$  ενώνουμε  $R.L$  με  $L$ 
     $R.L := \text{Merge}(L, R.L)$ 
    return  $R$ 
end if
```

Προσοχή:



Merge

(Immediately correct)

```
function Merge( $L, R$ ):  $T$ 
  Require:  $\max\{L\} \leq \min\{R\}$ 
  // Επιλογή: Ποια ρίζα θα είναι ρίζα του  $T$ ;

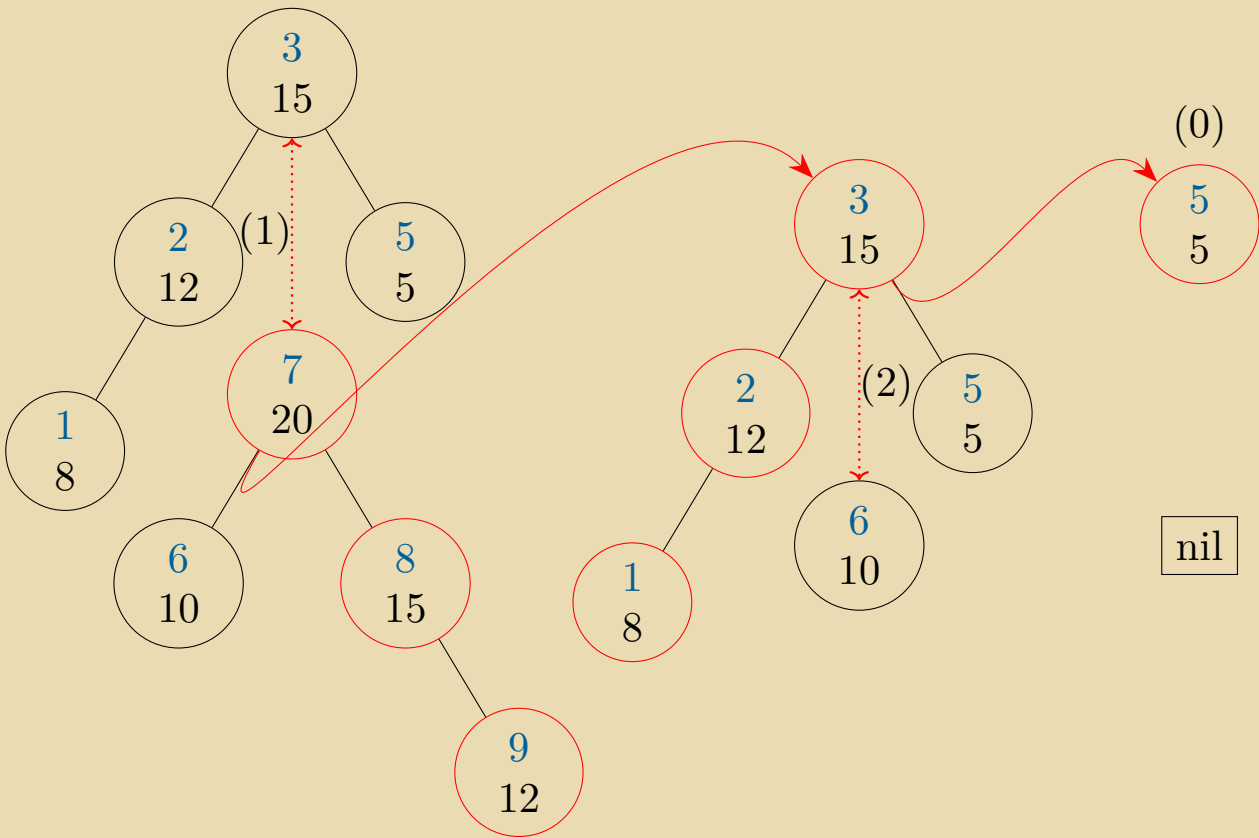
  if  $L = \emptyset$  or  $R = \emptyset$  then
    return non-empty  $L, R$ 
  else if  $L.priority > R.priority$  then
    // (1) ρίζα είναι το  $L \rightarrow$  ενώνουμε  $L.R$  με  $R$ 
     $L.R := \text{Merge}()$ 
    return  $L$ 
  else
    // (2) ρίζα είναι το  $R \rightarrow$  ενώνουμε  $R.L$  με  $L$ 
     $R.L := \text{Merge}(L, R.L)$ 
    return  $R$ 
end if
```

Προσοχή:

Η σειρά των παραμέτρων είναι σημαντική!



Παράδειγμα



Παρατήρηση



Παρατήρηση

- Η `Split()` χρησιμοποιεί **μόνο** το BST — μέρος



Παρατήρηση

- Η `Split()` χρησιμοποιεί **μόνο** το BST — μέρος
- Η `Merge()` χρησιμοποιεί **μόνο** το Heap — μέρος



Υλοποίηση



Υλοποίηση

```
struct treap_node {  
    int64_t key;  
    uint64_t priority;  
    treap_node *left, *right;  
};
```



Υλοποίηση

```
typedef struct treap_node treap_node;  
typedef treap_node *treap;  
struct treap_node {  
    int64_t key;  
    uint64_t priority;  
    treap_node *left, *right;  
};
```



Υλοποίηση

```
typedef struct treap_node treap_node;
typedef treap_node *treap;

struct treap_node {
    int64_t key;
    uint64_t priority;
    treap_node *left, *right;
};

void treap_split_ (treap this, int64_t key,
                  treap *l, treap *r)
{
    if (!this) *l = *r = NULL;
    else if (this->key <= key)
        treap_split_(this->right, key,
                     &(this->right), r), *l = this;
    else
        treap_split_(this->left, key,
                     l, &(this->left)), *r = this;
}
```



Υλοποίηση

```
typedef struct treap_node treap_node;
typedef treap_node *treap;

struct treap_node {
    int64_t key;
    uint64_t priority;
    treap_node *left, *right;
};

void treap_split_ (treap this, int64_t key,
                  treap *l, treap *r)
{
    if (!this) *l = *r = NULL;
    else if (this->key <= key)
        treap_split_(this->right, key,
                     &(this->right), r), *l = this;
    else
        treap_split_(this->left, key,
                     l, &(this->left)), *r = this;
}
```

```
void treap_merge_ (treap *t, treap l, treap r)
{
    if (!l || !r)
        *t = l ? l : r;
    else if (l->priority > r->priority)
        treap_merge_(&(l->right),
                     l->right, r), *t = l;
    else
        treap_merge_(&(r->left),
                     l, r->left), *t = r;
}
```



Υλοποίηση

```
typedef struct treap_node treap_node;
typedef treap_node *treap;

struct treap_node {
    int64_t key;
    uint64_t priority;
    treap_node *left, *right;
};

void treap_split_ (treap this, int64_t key,
                  treap *l, treap *r)
{
    if (!this) *l = *r = NULL;
    else if (this->key <= key)
        treap_split_(this->right, key,
                     &(this->right), r), *l = this;
    else
        treap_split_(this->left, key,
                     l, &(this->left)), *r = this;
}
```

```
void treap_merge_ (treap *t, treap l, treap r)
{
    if (!l || !r)
        *t = l ? l : r;
    else if (l->priority > r->priority)
        treap_merge_(&(l->right),
                     l->right, r), *t = l;
    else
        treap_merge_(&(r->left),
                     l, r->left), *t = r;
}

/* Example */
treap t = {}, l, r;
treap_split_(t, 42, &l, &r);
treap_merge_(&t, l, r);
if (t) { /*...*/ }
```



Και τώρα ... τι;



User Interface – συναρτήσεις



User Interface – συναρτήσεις

FIND(T, key) Ακριβώς όπως και στο BST



User Interface – συναρτήσεις

FIND(T, key) Ακριβώς όπως και στο BST

```
treap treap_find (treap this, int64_t key)
{
    if (!this) return NULL;
    else if (key == this->key)
        return this;
    else if (key < this->key)
        return treap_find(this->left, key);
    else
        return treap_find(this->right, key);
}
```



User Interface – συναρτήσεις (II)

INSERT(T, key) Σχεδόν όπως στο BST:



User Interface – συναρτήσεις (II)

INSERT(T, key) Σχεδόν όπως στο BST:
Search μέχρι **nil** node \rightarrow



User Interface – συναρτήσεις (II)

INSERT(T, key) Σχεδόν όπως στο BST:
Search μέχρι **nil** node \longrightarrow

Search μέχρι χαμηλή προτεραιότητα



User Interface – συναρτήσεις (II)

INSERT(T, key) Σχεδόν όπως στο BST:
Search μέχρι **nil** node \longrightarrow

Search μέχρι χαμηλή προτεραιότητα

\hookrightarrow Όρισε παιδιά μέσω **SPLIT**()



User Interface – συναρτήσεις (II)

INSERT(T, key) Σχεδόν όπως στο BST:
Search μέχρι **nil** node \rightarrow

Search μέχρι χαμηλή προτεραιότητα

\hookrightarrow Όρισε παιδιά μέσω **SPLIT**()

```
treap treap_insert_node (treap this, treap u)
{
    if (!this) return u;
    if (u->priority > this->priority) {
        treap_split(this, u->key, u->left, u->right);
        return u;
    }
    if (u->key < this->key)
        this->left = treap_insert_node(this->left, u);
    else
        this->right = treap_insert_node(this->right, u);
    return this;
}
```



User Interface – συναρτήσεις (III)

ERASE(T, key) Σχεδόν όπως στο BST. Όταν βρεθεί το key :



User Interface – συναρτήσεις (III)

ERASE(T, key) Σχεδόν όπως στο BST. Όταν βρεθεί το key : ένωσε παιδιά μέσω **MERGE**()



User Interface – συναρτήσεις (III)

ERASE(T, key) Σχεδόν όπως στο BST. Όταν βρεθεί το key : ένωσε παιδιά μέσω **MERGE**()

```
treap treap_erase_ (treap this, int64_t key)
{
    treap tmp;

    if (!this) return NULL;
    if (key == this->key) {
        treap_merge(tmp, this->left, this->right);
        free(this);
        return tmp;
    }
    if (key < this->key)
        this->left = treap_erase_(this->left, key);
    else
        this->right = treap_erase_(this->right, key);
    return this;
}
```



Macros to the rescue

```
#define treap_split(t, key, l, r) \  
    treap_split_(t, key, &(l), &(r))  
#define treap_merge(t, l, r)  
treap_merge_(&(t), l, r)
```



Macros to the rescue

```
#define treap_split(t, key, l, r) \  
    treap_split_(t, key, &(l), &(r))  
#define treap_merge(t, l, r)  
treap_merge_(&(t), l, r)
```

```
/* Example */  
treap t = {}, l, r;  
treap_split_(t, 42, &l, &r);  
treap_merge_(&t, l, r);  
if (t) { /*...*/ }
```



Macros to the rescue

```
#define treap_split(t, key, l, r) \  
    treap_split_(t, key, &(l), &(r))  
#define treap_merge(t, l, r)  
treap_merge_(&(t), l, r)
```

```
/* Example */  
treap t = {}, l, r;  
treap_split_(t, 42, &l, &r);  
treap_merge_(&t, l, r);  
if (t) { /*...*/ }
```

```
/* Now becomes */  
  
treap_split(t, 42, l, r);  
treap_merge(t, l, r);
```



Macros to the rescue

```
#define treap_split(t, key, l, r) \  
    treap_split_(t, key, &l, &r))  
#define treap_merge(t, l, r)  
treap_merge_(&(t), l, r)
```

```
/* Example */
```

```
treap t = {}, l, r;  
treap_split_(t, 42, &l, &r);  
treap_merge_(&t, l, r);  
if (t) { /*...*/ }
```

```
#define treap_insert(t, k, p) \  
    (void)((t) = treap_insert_node(t,  
node_create((k), (p))))  
#define treap_erase(t, x) (void)((t) =  
treap_erase_(t, x))  
#define treap_count(t, x) (treap_find(t, x) !=  
NULL)  
//      treap_find(t, x)  
/* Now becomes */
```

```
treap_split(t, 42, l, r);  
treap_merge(t, l, r);
```



Macros to the rescue

```
#define treap_split(t, key, l, r) \  
    treap_split_(t, key, &l, &r))  
#define treap_merge(t, l, r)  
treap_merge_(&(t), l, r)
```

```
/* Example */
```

```
treap t = {}, l, r;  
treap_split_(t, 42, &l, &r);  
treap_merge_(&t, l, r);  
if (t) { /*...*/ }
```

```
#define treap_insert(t, k, p) \  
    (void)((t) = treap_insert_node(t,  
node_create((k), (p))))  
#define treap_erase(t, x) (void)((t) =  
treap_erase_(t, x))  
#define treap_count(t, x) (treap_find(t, x) !=  
NULL)
```

```
//      treap_find(t, x)
```

```
/* Now becomes */
```

```
treap_split(t, 42, l, r);  
treap_merge(t, l, r);
```

```
treap_insert(t, 42, 20);  
if (treap_count(t, 42)) { /*...*/ }  
treap_erase(t, 42);
```



Treaps as Balanced BSTs

BALANCED BINARY TREE



Treaps as Balanced BSTs

BALANCED BINARY TREE

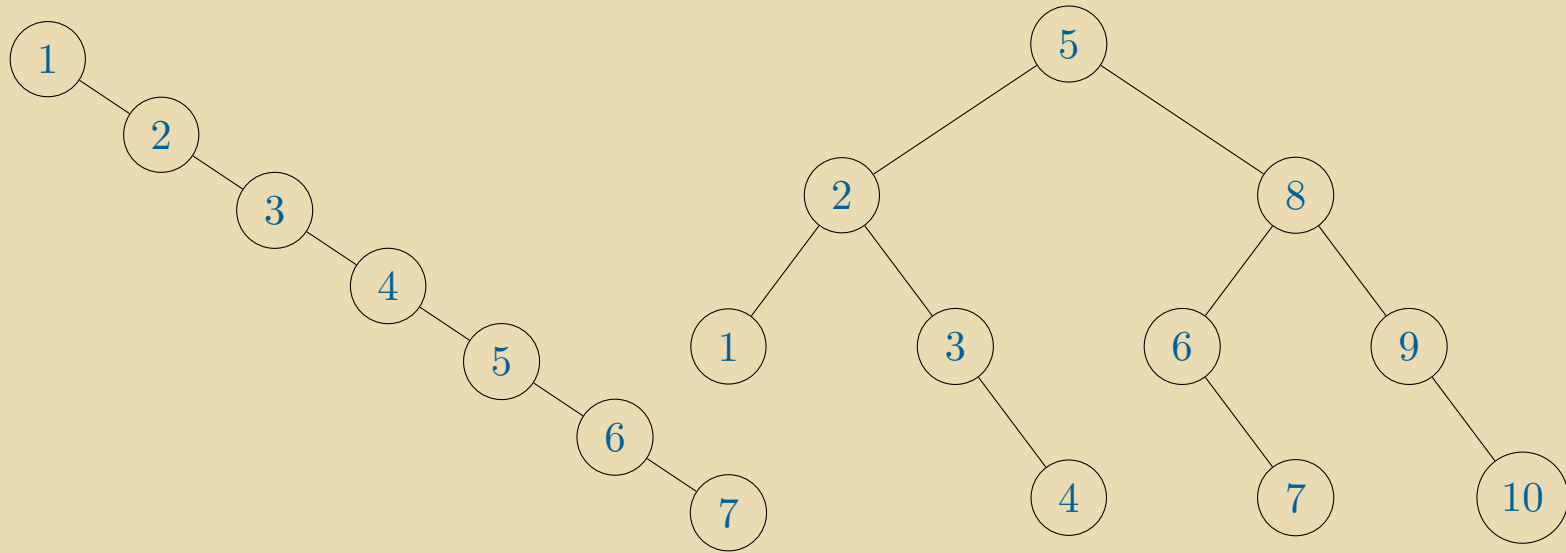
Ύψος $O(\log n)$, n το πλήθος των κόμβων.



Treaps as Balanced BSTs

BALANCED BINARY TREE

Ύψος $O(\log n)$, η το πλήθος των κόμβων.



Σχήμα 2 a) An unbalanced BST b) A balanced BST



Treaps as Balanced BSTs (II)

- Πιο ταξινομημένα στοιχεία \implies πιο μεγάλες αλυσίδες δημιουργούνται.



Treaps as Balanced BSTs (II)

- Πιο ταξινομημένα στοιχεία \implies πιο μεγάλες αλυσίδες δημιουργούνται.
 \hookrightarrow Λύση: Κάνουμε τυχαία ανάμειξη των στοιχείων



Treaps as Balanced BSTs (II)

- Πιο ταξινομημένα στοιχεία \implies πιο μεγάλες αλυσίδες δημιουργούνται.
 \hookrightarrow Λύση: Κάνουμε τυχαία ανάμειξη των στοιχείων
Τι γίνεται όταν δεν έχουμε όλα τα στοιχεία από την αρχή (*online arrival*);



Treaps as Balanced BSTs (II)

- Πιο ταξινομημένα στοιχεία \implies πιο μεγάλες αλυσίδες δημιουργούνται.

\hookrightarrow Λύση: Κάνουμε τυχαία ανάμειξη των στοιχείων

Τι γίνεται όταν δεν έχουμε όλα τα στοιχεία από την αρχή (*online arrival*);

\hookrightarrow Θέλουμε να μπορούμε να εισάγουμε στοιχεία σε υψηλότερα ή χαμηλότερα επίπεδα:

Priorities στο treap



Treaps as Balanced BSTs (II)

- Πιο ταξινομημένα στοιχεία \implies πιο μεγάλες αλυσίδες δημιουργούνται.

\hookrightarrow Λύση: Κάνουμε τυχαία ανάμειξη των στοιχείων

Τι γίνεται όταν δεν έχουμε όλα τα στοιχεία από την αρχή (*online arrival*);

\hookrightarrow Θέλουμε να μπορούμε να εισάγουμε στοιχεία σε υψηλότερα ή χαμηλότερα επίπεδα:

Priorities στο treap

\hookrightarrow Θέλουμε τυχαία σειρά \implies τυχαίο priority



Treaps as Balanced BSTs (II)

- Πιο ταξινομημένα στοιχεία \implies πιο μεγάλες αλυσίδες δημιουργούνται.

\hookrightarrow Λύση: Κάνουμε τυχαία ανάμειξη των στοιχείων

Τι γίνεται όταν δεν έχουμε όλα τα στοιχεία από την αρχή (*online arrival*);

\hookrightarrow Θέλουμε να μπορούμε να εισάγουμε στοιχεία σε υψηλότερα ή χαμηλότερα επίπεδα:

Priorities στο treap

\hookrightarrow Θέλουμε τυχαία σειρά \implies τυχαίο priority

Για 10.000 κόμβους, η πιθανότητα να παραχθεί δένδρο με ύψος μεγαλύτερο από 100 είναι 1 στα 2.5 δισεκατομμύρια



Code Changes

```
#define treap_insert(t, x) (void)((t) = treap_insert_node(t, node_create(x)))
```



Code Changes

```
#define treap_insert(t, x) (void)((t) = treap_insert_node(t, node_create(x)))
treap node_create (int64_t key)
{
    treap_node *ret = malloc(sizeof(*ret));
    ret->key = key;
    ret->priority = ((uint64_t)rand() << 32)
        | rand();

    return ret;
}
```



Treap Augmentation

Γενική ιδέα

- Δικό μας BBST \implies προσθέτουμε ό,τι θέλουμε



Treap Augmentation

Γενική ιδέα

- Δικό μας BBST \implies προσθέτουμε ό,τι θέλουμε
- Τι;



Treap Augmentation

Γενική ιδέα

- Δικό μας BBST \implies προσθέτουμε ό,τι θέλουμε
- Τι;
 1. size
 2. sum
 3. max



Treap Augmentation

Γενική ιδέα

- Δικό μας BBST \implies προσθέτουμε ό,τι θέλουμε
- Τι;
 1. size
 2. sum
 3. max (*not really*)



Treap Augmentation

Γενική ιδέα

- Δικό μας BBST \implies προσθέτουμε ό,τι θέλουμε
- Τι;
 1. size
 2. sum
 3. max (*not really*)

ΑΝΑΔΡΟΜΗ



Treap Augmentation (II)

Υλοποίηση size

```
struct treap_node {  
    int64_t key;  
    uint64_t priority;  
    treap_node *left, *right;  
    size_t size;  
};
```



Treap Augmentation (II)

Υλοποίηση size

```
struct treap_node {
    int64_t key;
    uint64_t priority;
    treap_node *left, *right;
    size_t size;
};

treap node_create (int64_t key)
{
    treap_node *ret = malloc(sizeof(*ret));
    ret->key = key;
    ret->priority = ((uint64_t)rand() << 32)
        | rand();
    ret->left = ret->right = NULL;
    ret->size = 1;
    return ret;
}
```



Treap Augmentation (II)

Υλοποίηση size

```
struct treap_node {  
    int64_t key;  
    uint64_t priority;  
    treap_node *left, *right;  
    size_t size;  
};
```

```
treap_node* create (int64_t key)  
{  
    treap_node *ret = malloc(sizeof(*ret));  
    ret->key = key;  
    ret->priority = ((uint64_t)rand() << 32)  
        | rand();  
    ret->left = ret->right = NULL;  
    ret->size = 1;  
    return ret;  
}
```

```
void treap_update (treap this)  
{  
    if (!this) return;  
    this->size = 1 + treap_size(this->left)  
        + treap_size(this->right);  
}
```



Treap Augmentation (II)

Υλοποίηση size

```
struct treap_node {  
    int64_t key;  
    uint64_t priority;  
    treap_node *left, *right;  
    size_t size;  
};
```

```
treap node_create (int64_t key)  
{  
    treap_node *ret = malloc(sizeof(*ret));  
    ret->key = key;  
    ret->priority = ((uint64_t)rand() << 32)  
        | rand();  
    ret->left = ret->right = NULL;  
    ret->size = 1;  
    return ret;  
}
```

```
void treap_update (treap this)  
{  
    if (!this) return;  
    this->size = 1 + treap_size(this->left)  
        + treap_size(this->right);  
}
```

```
void treap_split_ (treap this, int64_t key, treap  
*l, treap *r)  
{  
    :  
    treap_update(this);  
}  
  
etc.
```



Treap Augmentation (III)

Υλοποίηση size και sum

```
struct treap_node {  
    int64_t key;  
    uint64_t priority;  
    treap_node *left, *right;  
    size_t size;  
    int64_t sum;  
};
```

```
treap_node_create (int64_t key)  
{  
    treap_node *ret = malloc(sizeof(*ret));  
    ret->key = key;  
    ret->priority = ((uint64_t)rand() << 32)  
        | rand();  
    ret->left = ret->right = NULL;  
    ret->size = 1;  
    ret->sum = key;  
    return ret;  
}
```

```
void treap_update (treap this)  
{  
    if (!this) return;  
    this->size = 1 + treap_size(this->left)  
        + treap_size(this->right);  
    this->sum = this->key + treap_sum(this->left)  
        + treap_sum(this->right);  
}
```

Access μέσω:

- `treap_size(t)`
- `treap_sum(t)`



Implicit Treaps



Implicit Treaps

- Αντί για keys \rightarrow θέση στον "πίνακα"



Implicit Treaps

- Αντί για keys \rightarrow θέση στον "πίνακα"
- Δεν αποθηκεύω κλειδιά \rightarrow βρίσκονται (*implied*) από θέση



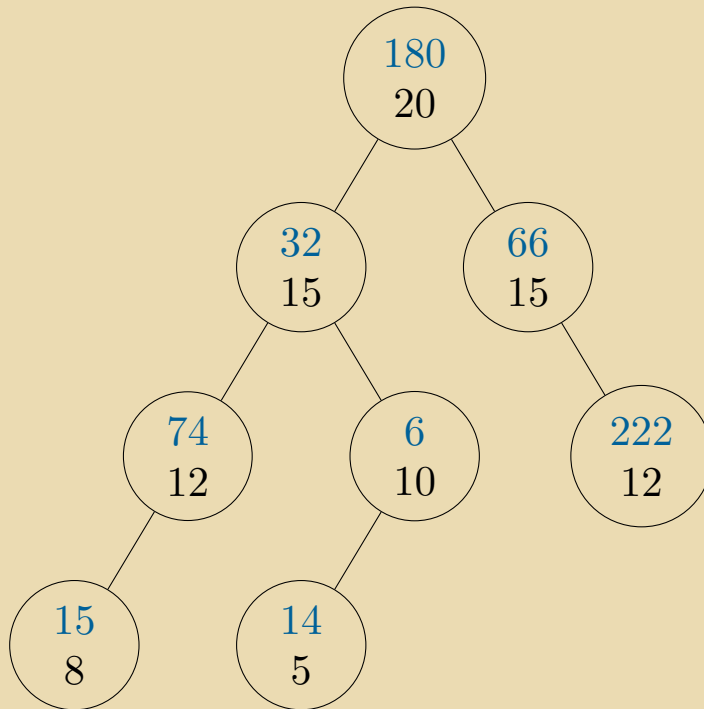
Implicit Treaps

- Αντί για keys \rightarrow θέση στον "πίνακα"
- Δεν αποθηκεύω κλειδιά \rightarrow βρίσκονται (*implied*) από θέση
- Αποθηκεύουμε το value (*Data Structure Augmentation*)



Implicit Treaps

- Αντί για keys \rightarrow θέση στον "πίνακα"
- Δεν αποθηκεύω κλειδιά \rightarrow βρίσκονται (*implied*) από θέση
- Αποθηκεύουμε το value (*Data Structure Augmentation*)



0	1	2	3	4	5	6	7
15	4	32	14	6	180	66	222

Θέση \rightarrow κόμβοι πιο αριστερά
 \rightarrow χρησιμοποιώ size



Implicit Treap Code

```
struct itreap_node {  
    int32_t value;  
    uint64_t priority;  
    itreap_node *left, *right;  
    size_t size;  
};
```



Implicit Treap Code

```
struct itreap_node {  
    int32_t value;  
    uint64_t priority;  
    itreap_node *left, *right;  
    size_t size;  
};
```

```
void itreap_split_ (itreap this, ptrdiff_t key,  
    itreap *l, itreap *r, size_t prv)  
{  
    if (!this) {  
        *l = *r = NULL;  
        return;  
    }  
  
    int cur_key = prv + itreap_size(this->left);  
    if (cur_key <= key)  
        itreap_split_(this->right, key,  
            &(this->right), r, cur_key + 1),  
        *l = this;  
    else  
        itreap_split_(this->left, key, l,  
            &(this->left), prv), *r = this;  
    itreap_update(this);  
}
```



Implicit Treap Code

```
struct itreap_node {
    int32_t value;
    uint64_t priority;
    itreap_node *left, *right;
    size_t size;
};

itreap itreap_set_ (itreap this, size_t pos,
int32_t val)
{
    itreap u = node_create(val),
        l = {}, r = {};
    assert(itreap_size(this) >= pos);

    itreap_split(this, (ptrdiff_t)pos - 1,
        l, r);
    itreap_merge(this, l, u);
    itreap_merge(this, this, r);
    return this;
}
```

```
void itreap_split_ (itreap this, ptrdiff_t key,
    itreap *l, itreap *r, size_t prv)
{
    if (!this) {
        *l = *r = NULL;
        return;
    }
    int cur_key = prv + itreap_size(this->left);
    if (cur_key <= key)
        itreap_split_(this->right, key,
            &(this->right), r, cur_key + 1),
        *l = this;
    else
        itreap_split_(this->left, key, l,
            &(this->left), prv), *r = this;
    itreap_update(this);
}
```



Implicit Treap Code (II)

```
int32_t itreap_get (itreap this, size_t pos)
{
    int32_t ret;
    itreap l = {}, r = {};
    assert(itreap_size(this) > pos);

    itreap_split(this, pos, this, this->right);
    itreap_split(this, pos-1, l, this);
    ret = this->value;
    itreap_merge(this, l, this),
    itreap_merge(this, this, r);
    return ret;
}
```



Implicit Treap Code (II)

```
int32_t itreap_get (itreap this, size_t pos) treap itreap_erase_ (itreap this, size_t pos)
{
    int32_t ret;
    itreap l = {}, r = {};
    assert(itreap_size(this) > pos);
    itreap_split(this, pos, this, this->right);
    itreap_split(this, pos-1, l, this);
    ret = this->value;
    itreap_merge(this, l, this),
    itreap_merge(this, this, r);
    return ret;
}

{
    itreap l, r;
    itreap_split(this, pos, this, r);
    itreap_split(this, (ptrdiff_t)pos - 1, l,
    this);
    itreap_merge(this, l, r);
    return this;
}
```



Implicit Treap Code (II)

```
int32_t itreap_get (itreap this, size_t pos) treap itreap_erase_ (itreap this, size_t pos)
{
    int32_t ret;
    itreap l = {}, r = {};
    assert(itreap_size(this) > pos);

    itreap_split(this, pos, this, this->right);
    itreap_split(this, pos-1, l, this);
    ret = this->value;
    itreap_merge(this, l, this),
    itreap_merge(this, this, r);
    return ret;
}

#define itreap_set(t, pos, val) \
    (void)((t) = itreap_set_(t, pos, val))
#define itreap_erase(t, pos) \
    (void)((t) = itreap_erase_(t, pos))
//      itreap_get(t, pos)
//      itreap_size(t)
```



Implicit Treap Code (II)

```
int32_t itreap_get (itreap this, size_t pos) {
    int32_t ret;
    itreap l = {}, r = {};
    assert(itreap_size(this) > pos);

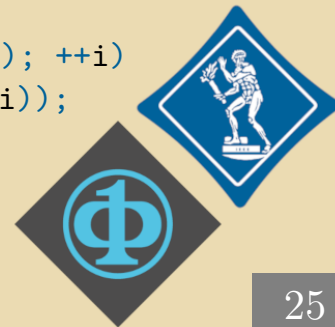
    itreap_split(this, pos, this, this->right);
    itreap_split(this, pos-1, l, this);
    ret = this->value;
    itreap_merge(this, l, this);
    itreap_merge(this, this, r);
    return ret;
}

#define itreap_set(t, pos, val) \
    (void)((t) = itreap_set_(t, pos, val))
#define itreap_erase(t, pos) \
    (void)((t) = itreap_erase_(t, pos))

//      itreap_get(t, pos)
//      itreap_size(t)

itreap itreap_erase_ (itreap this, size_t pos)
{
    itreap l, r;
    itreap_split(this, pos, this, r);
    itreap_split(this, (ptrdiff_t)pos - 1, l,
                  this);
    itreap_merge(this, l, r);
    return this;
}

itreap arr = {};
itreap_set(arr, 0, 42);
itreap_set(arr, 0, 17);
for (size_t i=0; i<itreap_size(arr); ++i)
    printf("%d\n", itreap_get(arr, i));
```



Implicit Treaps - Range Min/Max/Sum Queries



Implicit Treaps - Range Min/Max/Sum Queries

Ίδια ιδέα με τα απλά treaps \rightarrow Implicit κλειδιά άρα εύρος σε πίνακα



Implicit Treaps - Lazy propagation



Implicit Treaps - Lazy propagation

ΠΡΟΒΛΗΜΑ: Θέλουμε να τροποποιήσουμε ένα εύρος του πίνακα



Implicit Treaps - Lazy propagation

ΠΡΟΒΛΗΜΑ: Θέλουμε να τροποποιήσουμε ένα εύρος του πίνακα

- Εύρος \implies ίδια ιδέα με sum range query



Implicit Treaps - Lazy propagation

ΠΡΟΒΛΗΜΑ: Θέλουμε να τροποποιήσουμε ένα εύρος του πίνακα

- Εύρος \implies ίδια ιδέα με sum range query
- Αλλαγή μόνο στον τωρινό κόμβο \rightarrow Υπενθύμιση στα παιδιά



Implicit Treaps - Lazy propagation

ΠΡΟΒΛΗΜΑ: Θέλουμε να τροποποιήσουμε ένα εύρος του πίνακα

- Εύρος \implies ίδια ιδέα με sum range query
- Αλλαγή μόνο στον τωρινό κόμβο \rightarrow Υπενθύμιση στα παιδιά
- Πολυπλοκότητα \rightarrow **μόνο** split/merge + πράξη $\rightarrow O(\log n) + O(\text{πράξης})$



Implicit Treaps - Lazy propagation

ΠΡΟΒΛΗΜΑ: Θέλουμε να τροποποιήσουμε ένα εύρος του πίνακα

- Εύρος \implies ίδια ιδέα με sum range query
- Αλλαγή μόνο στον τωρινό κόμβο \rightarrow Υπενθύμιση στα παιδιά
- Πολυπλοκότητα \rightarrow **μόνο** split/merge + πράξη $\rightarrow O(\log n) + O(\text{πράξης})$

ΠΑΡΑΔΕΙΓΜΑΤΑ



Implicit Treaps - Lazy propagation

ΠΡΟΒΛΗΜΑ: Θέλουμε να τροποποιήσουμε ένα εύρος του πίνακα

- Εύρος \implies ίδια ιδέα με sum range query
- Αλλαγή μόνο στον τωρινό κόμβο \rightarrow Υπενθύμιση στα παιδιά
- Πολυπλοκότητα \rightarrow **μόνο** split/merge + πράξη $\rightarrow O(\log n) + O(\text{πράξης})$

ΠΑΡΑΔΕΙΓΜΑΤΑ

- Πρόσθεση σταθερού αριθμού σε εύρος: $O(\log n) + O(1) = O(\log n)$



Implicit Treaps - Lazy propagation

ΠΡΟΒΛΗΜΑ: Θέλουμε να τροποποιήσουμε ένα εύρος του πίνακα

- Εύρος \implies ίδια ιδέα με sum range query
- Αλλαγή μόνο στον τωρινό κόμβο \rightarrow Υπενθύμιση στα παιδιά
- Πολυπλοκότητα \rightarrow **μόνο** split/merge + πράξη $\rightarrow O(\log n) + O(\text{πράξης})$

ΠΑΡΑΔΕΙΓΜΑΤΑ

- Πρόσθεση σταθερού αριθμού σε εύρος: $O(\log n) + O(1) = O(\log n)$
- Αναστροφή εύρους: $O(\log n) + O(1) = O(\log n)$



Lazy Implicit Treaps

```
struct itreap_node {
    int32_t value;
    uint64_t priority;
    itreap_node *left, *right;
    size_t size;
    int64_t sum; /* auxiliary value */
    int32_t add; /* lazy values */
    bool flip;
};

itreap_node_create (int32_t value)
{
    itreap_node *ret = malloc(sizeof(*ret));
    ret->value = value;
    ret->priority = ((uint64_t)rand() << 32) |
    rand();
    ret->left = ret->right = NULL;
    ret->size = 1;
    ret->sum = value;
    ret->add = ret->flip = 0;
    return ret;
}
```



Lazy Implicit Treaps

```
struct itreap_node {
    int32_t value;
    uint64_t priority;
    itreap_node *left, *right;
    size_t size;
    int64_t sum; /* auxiliary value */
    int32_t add; /* lazy values */
    bool flip;
};

itreap_node_create (int32_t value)
{
    itreap_node *ret = malloc(sizeof(*ret));
    ret->value = value;
    ret->priority = ((uint64_t)rand() << 32) |
rand();
    ret->left = ret->right = NULL;
    ret->size = 1;
    ret->sum = value;
    ret->add = ret->flip = 0;
    return ret;
}
```

```
void itreap_update (itreap this)
{
    if (!this) return;
    this->size = 1 + itreap_size(this->left) +
itreap_size(this->right);
    this->sum = this->value +
itreap_sum(this->left) +
    itreap_sum(this->right);
}
```



Lazy Implicit Treaps (II)

```
void itreap_push (itreap this)
{
    if (this && this->add) {
        if (this->left)
            this->left->sum += this->left->size * this->add,
            this->left->value += this->add,
            this->left->add += this->add;
        if (this->right)
            this->right->sum += this->right->size * this->add,
            this->right->value += this->add,
            this->right->add += this->add;
        this->add = 0;
    }
    if (this && this->flip) {
        SWAP(this->left, this->right);
        if (this->left) this->left->flip = !this->left->flip;
        if (this->right) this->right->flip = !this->right->flip;
        this->flip = false;
    }
}
```



Lazy Implicit Treaps (III)

Add

```
itreap
itreap_add_ (itreap this, size_t from, size_t to, int32_t add)
{
    itreap l = {}, r = {};
    assert(from <= to && itreap_size(this) > to);

    itreap_split(this, to, this, r);
    itreap_split(this, (ptrdiff_t)from - 1, l, this);
    this->sum += this->size * add;
    this->value += add, this->add += add;
    itreap_merge(this, l, this), itreap_merge(this, this, r);
    return this;
}
```



Lazy Implicit Treaps (III)

Add

```
#define itreap_add(t, from, to, x) \  
    (void)((t) = treap_add_(t, from, to, x))  
  
itreap  
itreap_add_ (itreap this, size_t from, size_t to, int32_t add)  
{  
    itreap l = {}, r = {};  
    assert(from <= to && itreap_size(this) > to);  
  
    itreap_split(this, to, this, r);  
    itreap_split(this, (ptrdiff_t)from - 1, l, this);  
    this->sum += this->size * add;  
    this->value += add, this->add += add;  
    itreap_merge(this, l, this), itreap_merge(this, this, r);  
    return this;  
}
```



Lazy Implicit Treaps (IV)

Flip

```
itreap
itreap_flip_ (itreap this, size_t from, size_t to)
{
    itreap l = {}, r = {};
    assert(from <= to && itreap_size(this) > to);

    itreap_split(this, to, this, r);
    itreap_split(this, (ptrdiff_t)from - 1, l, this);
    SWAP(this->left, this->right);
    this->flip ^= 1;
    itreap_merge(this, l, this), itreap_merge(this, this, r);
    return this;
}
```



Lazy Implicit Treaps (IV)

Flip

```
#define itreap_flip(t, from, to) \  
    (void)((t) = treap_flip_(t, from, to))  
  
itreap  
itreap_flip_ (itreap this, size_t from, size_t to)  
{  
    itreap l = {}, r = {};  
    assert(from <= to && itreap_size(this) > to);  
  
    itreap_split(this, to, this, r);  
    itreap_split(this, (ptrdiff_t)from - 1, l, this);  
    SWAP(this->left, this->right);  
    this->flip ^= 1;  
    itreap_merge(this, l, this), itreap_merge(this, this, r);  
    return this;  
}
```

