ФАКУЛТЕТ ЗА ИНФОРМАТИЧКИ НАУКИ И КОМПЈУТЕРСКО ИНЖЕНЕРСТВО

Пребарувачки дрва

Алгоритми и податочни структури

Аудиториска вежба 9

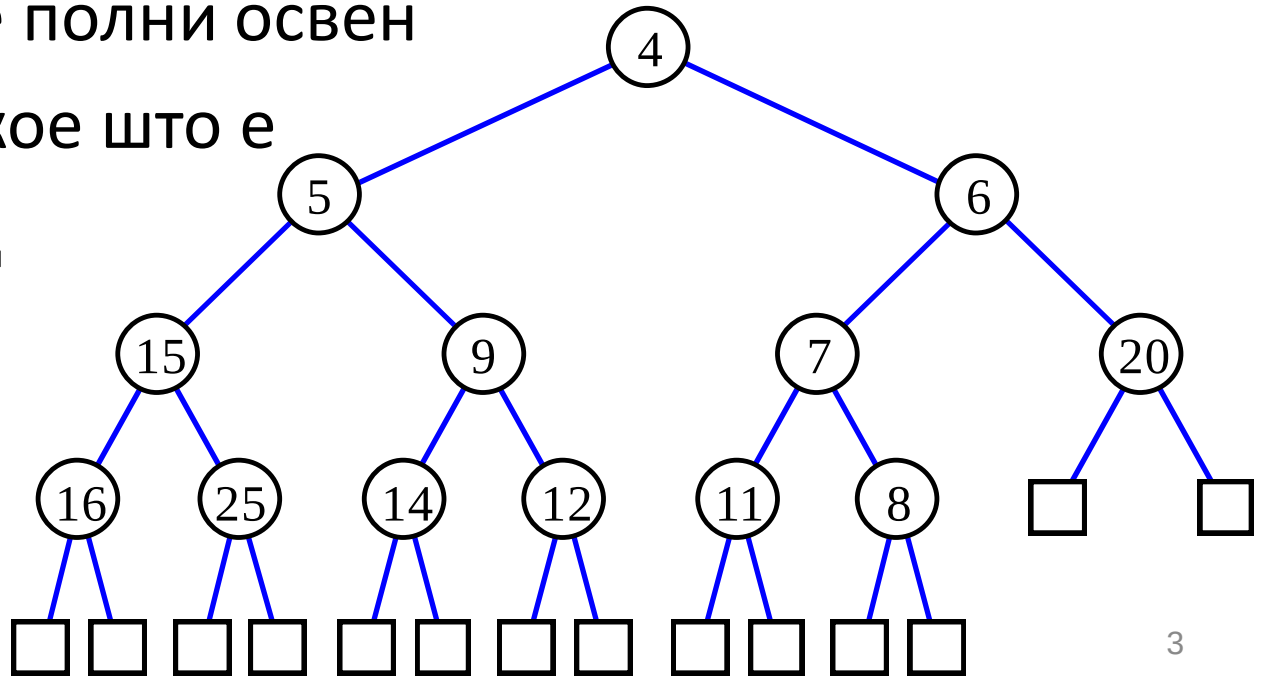
Пребарувачки дрва

- Содржина:
  - Heap дрва
  - Небалансирани бинарни пребарувачки дрва
  - Балансирани бинарни пребарувачки дрва (AVL)

Heap дрва

- Heap дрво е комплетно бинарно дрво со следните heap карактеристики:
- MinHeap: клуч(родител) <= клуч(дете)

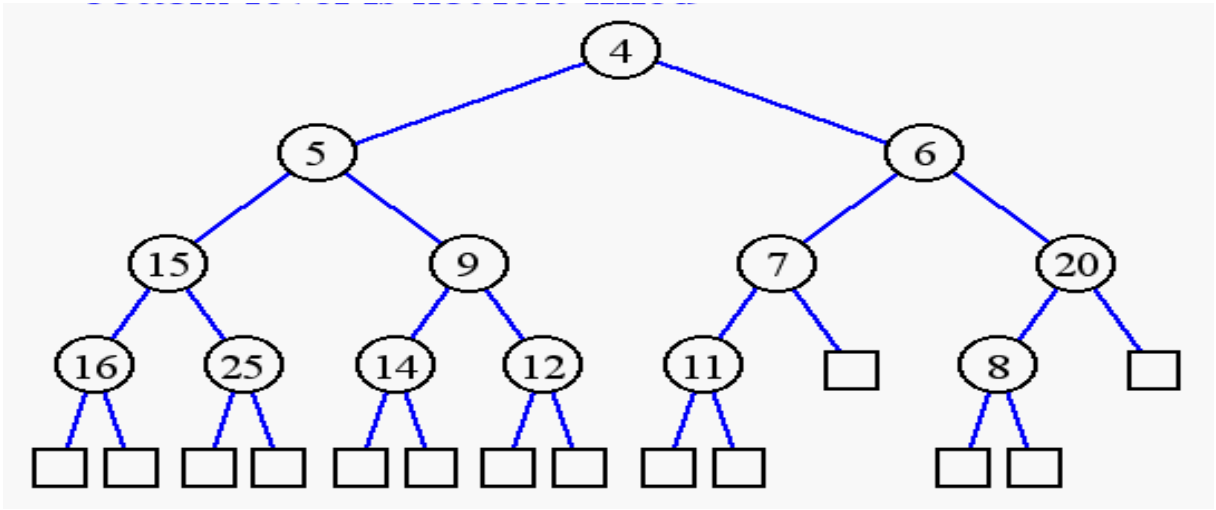
  [или MaxHeap: клуч(родител) >= клуч(дете)]
- Сите нивоа се полни освен последното, кое што е пополнето од левата страна

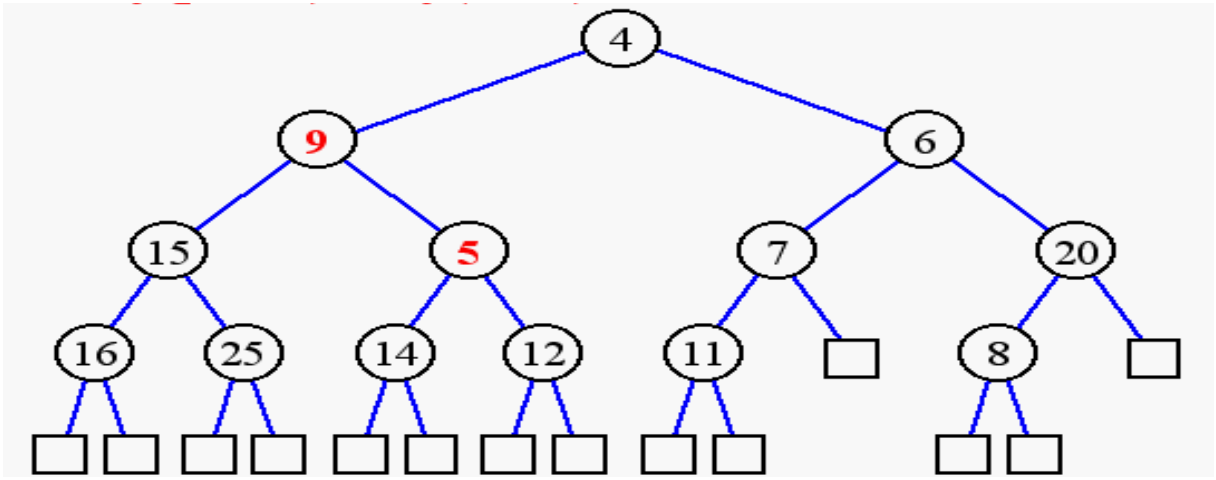


Heap дрва и нивна примена

- За имплементација на приоритетни редици
- Приоритетна редица = ред во кој што сите елементи имаат асоцирано “приоритет”
- Со операцијата бришење во приоритетна редица се отстранува елементот со најмал приоритет
- Операциите кои што можат да се користат кај heap дрвата се:
  - insert
  - removeMin
- Heap дрвата се користат и за сортирање

Heap дрва или не?

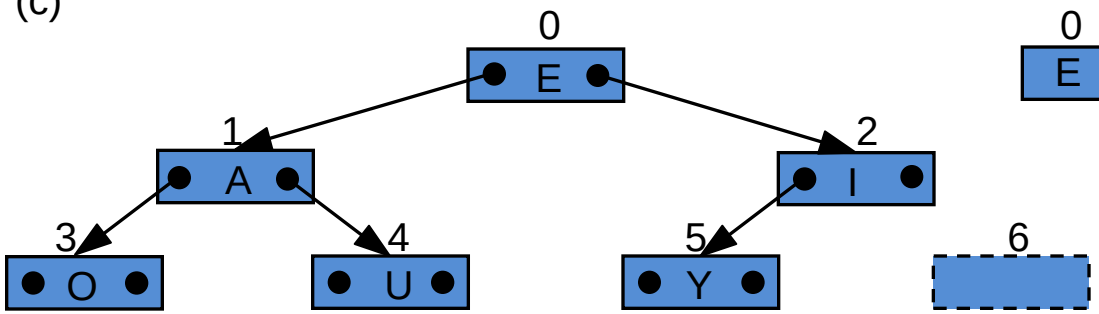


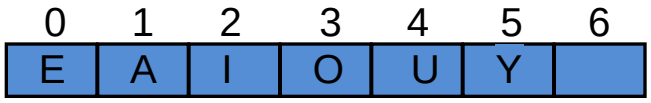


Heap дрва или не?

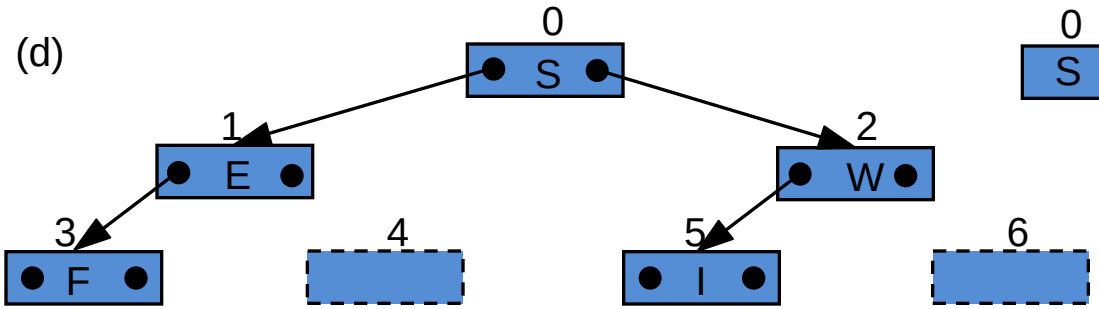
- Примери

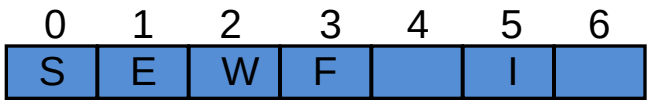
(c)





(d)

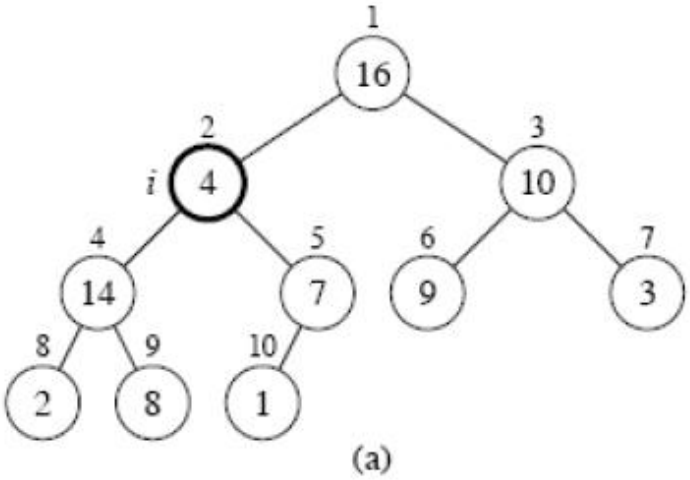




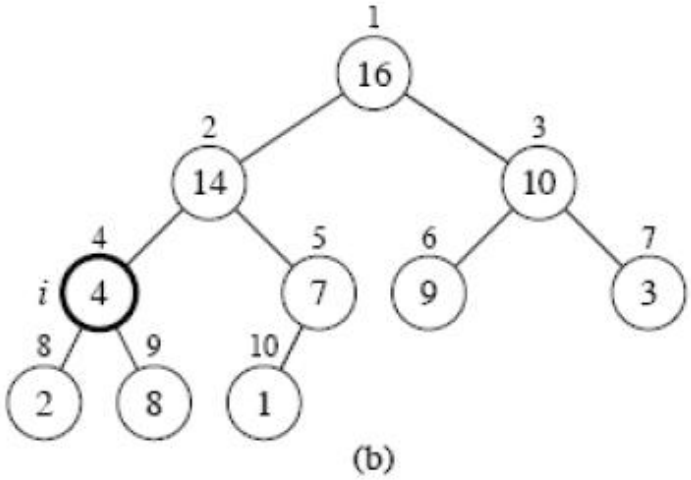
Adjust

- Пред adjust:
  - A[i] може да е помал од неговите деца
  - Претпоставуваме дека левото и десното поддрво на i се max-heap
- По adjust:
  - Поддрвото со корен во i е max-heap

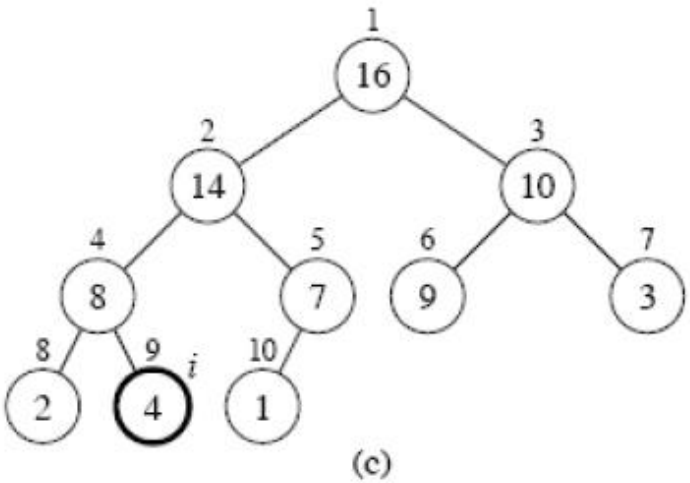
Adjust


1
16
2
i
4
3
10
4
14
5
7
6
9
7
3
8
2
9
8
10
1

(a)


1
16
2
14
3
10
4
i
4
5
7
6
9
7
3
8
2
9
8
10
1

(b)


1
16
2
14
3
10
4
8
5
7
6
9
7
3
8
2
9
i
4
10
1

(c)

Heap дрво - C

- void adjust(heap *h, int i, int n);

  Ги менува позициите на елементот што се наоѓа на позиција i во heap дрвото h (со моментална големина n) и на некое од неговите деца се додека постои нарушување на heap својството

- void buildHeap(heap *h);

  При дадена низа од елементи, ја преуредува низата така што heap дрвото кое што може да се претстави со таа низа ги исполнува heap својствата

Heap дрво - C

```
typedef int info_t;

typedef struct heapType {
    info_t *elements;
    int size;
} heap;

void initialize(heap *h, int size) {
    h->elements = (info_t *)calloc(size, sizeof(info_t));
    h->size = size;
}

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

int getLeft(int i) {
    return (i+1)*2-1;
}

int getRight(int i) {
    return (i+1)*2;
}
```

Heap дрво - C

```
info_t getAt(heap *h, int i) {
    return h->elements[i];
}

void setElement(heap *h, int index, info_t elem) {
    h->elements[index] = elem;
}

void swap(heap *h, int i, int j) {
    info_t tmp = h->elements[i];
    h->elements[i] = h->elements[j];
    h->elements[j] = tmp;
}
```

Heap дрво - C

```
void adjust(heap *h, int i, int n) {
    int left, right, largest;

    while (i < n) {
        left = getLeft(i);
        right = getRight(i);
        largest = i;
        if ((left < n)&&(h->elements[left] > h->elements[largest]))
            largest = left;
        if ((right < n)&&(h->elements[right] > h->elements[largest]))
            largest = right;
        if (largest == i)
            break;
        swap(h, i, largest);
        i = largest;
    }
}

void buildHeap(heap *h) {
    int i;
    for (i=h->size/2-1;i>=0;i--)
        adjust(h, i, h->size);
}
```

Heap дрво - Java

- void adjust(int i, int n);

  Ги менува позициите на елементот што се наоѓа на позиција i во heap дрвото (со моментална големина n) и на некое од неговите деца се додека постои нарушување на heap својството

- void buildHeap();

  При дадена низа од елементи, ја преуредува низата така што heap дрвото кое што може да се претстави со таа низа ги исполнува heap својствата

Heap дрво - Java

```
public class Heap<E extends Comparable<E>> {

    private E elements[];

    private Comparator<? super E> comparator;

    private int compare (E k1, E k2) {
    return (comparator==null ? k1.compareTo(k2) : comparator.compare(k1, k2));
    }

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

    int getLeft(int i) {
        return (i+1)*2-1;
    }

    int getRight(int i) {
        return (i+1)*2;
    }
```

Heap дрво - Java

```
public E getAt(int i) {
    return elements[i];
}

void setElement(int index, E elem) {
    elements[index] = elem;
}

void swap(int i, int j) {
    E tmp = elements[i];
    elements[i] = elements[j];
    elements[j] = tmp;
}
```

Heap дрво - Java

```
void adjust(int i, int n){
    while (i < n) {
        int left = getLeft(i);
        int right = getRight(i);
        int largest = i;

        if ((left < n)&&(elements[left].compareTo(elements[largest]) > 0))
            largest = left;
        if ((right < n)&&(elements[right].compareTo(elements[largest]) > 0))
            largest = right;
        if (largest == i)
            break;

        swap(i, largest);
        i = largest;
    }
}

void buildHeap() {
    int i;
    for (i=elements.length/2-1;i>=0;i--)
        adjust(i, elements.length);
}
```

Задача 1

- Имплементирајте heap sort алгоритам со помош на heap дрво

Задача 1 - C

```
void heapSort(heap *h) {
    int i;
    buildHeap(h);
    for (i=h->size;i>1;i--) {
        swap(h, 0, i-1);
        adjust(h, 0, i-1);
    }
}
```

Задача 1 - Java

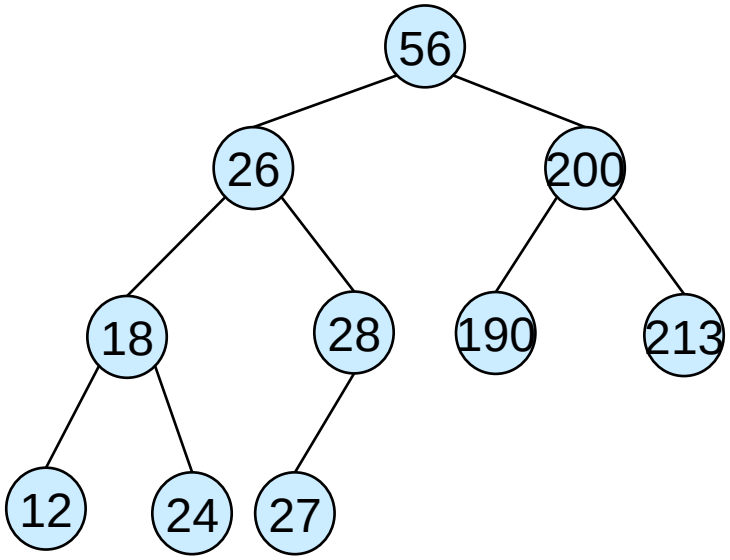
```
public void heapSort() {
    int i;
    buildHeap();
    for (i=elements.length;i>1;i--) {
        swap(0, i-1);
        adjust(0, i-1);
    }
}
```

Бинарни пребарувачки дрва

- Денес се потребни податочни структури кои што можат да подржуваат множество на динамички операции:
  - Пребарување
  - Минимум
  - Максимум
  - Претходник
  - Следбеник
  - Внесување
  - Бришење
- Со помош на овие операции може да се креираат:
  - Речници
  - Приоритетни редици
- Основните операции имаат време на извршување што е пропорционално на висината на дрвото - O(h).

Бинарни пребарувачки дрва

- Основни карактеристики на бинарните пребарувачки дрва се:
  - $\forall$ *y* во левото поддрво на *x*, важи *клуч*[*y*] < *клуч*[*x*]
  - $\forall$ *y* во десното поддрво на *x*, важи *клуч*[*y*] > *клуч*[*x*]
  - нема дупликати на клучевите



Бинарни пребарувачки дрва - C

```
typedef int info_t;

typedef struct bNodeType {
    info_t info;
    struct bNodeType *left;
    struct bNodeType *right;
} bNode;

typedef struct bTreeType {
    bNode *root;
} BSTree;

void initialize(BSTree *t) {
    t->root = NULL;
}
```

Бинарни пребарувачки дрва - C

- void insertT(BSTree *t, info_t x);

  Додади елемент во дрвото tree
- void deleteT(BSTree *tree, info_t x);

  Отстрани го елементот x од дрвото tree
- bNode* findT(BSTree *tree, info_t x);

  Пронајди го јазелот што го содржи инфо полето x
- info_t findMinT(BSTree *tree);

  Врати го најмалиот елемент во дрвото

Бинарни пребарувачки дрва - C

- info_t findMaxT(BSTree *tree);

  Врати го најголемиот елемент во дрвото
- int isEmptyT(BSTree *tree);

  Врати 1 ако дрвото е празно
- void makeEmptyT(BSTree *tree);

  Испразни го дрвото
- void printTreeT(BSTree *tree);

  Испечати го дрвото во сортиран редослед

Бинарни пребарувачки дрва - C

```
bNode* createNewNode(info_t x, bNode *left, bNode *right) {
    bNode *tmp = (bNode*)malloc(sizeof(bNode));
    tmp->info = x;
    tmp->left = left;
    tmp->right = right;
    return tmp;
}

bNode* insert(info_t x, bNode *t) {
    if (t == NULL) {
        t = createNewNode(x, NULL, NULL);
    } else if (x < t->info) {
        t->left = insert(x, t->left);
    } else if (x > t->info) {
        t->right = insert(x, t->right);
    } else; // Duplicate; do nothing
    return t;
}

void insertT(BSTree *t, info_t x) {
    t->root = insert(x, t->root);
}
```

Бинарни пребарувачки дрва - C

```
bNode* findMin(bNode *t) {
    if (t == NULL) {
        return NULL;
    } else if (t->left == NULL) {
        return t;
    }
    return findMin(t->left);
}

info_t findMinT(BSTree *tree) {
    return elementAt(findMin(tree->root));
}

bNode* findMax(bNode *t) {
    if (t == NULL) {
        return NULL;
    } else if (t->right == NULL) {
        return t;
    }
    return findMax(t->right);
}

info_t findMaxT(BSTree *tree) {
    return elementAt(findMax(tree->root));
}
```

Бинарни пребарувачки дрва - C

```
bNode* find(info_t x, bNode *t) {
    if (t == NULL)
        return NULL;

    if (x < t->info) {
        return find(x, t->left);
    } else if (x > t->info) {
        return find(x, t->right);
    } else {
        return t;    // Match
    }
}

bNode* findT(BSTree *tree, info_t x) {
    return find(x, tree->root);
}
```

Бинарни пребарувачки дрва - C

```
bNode* delete(info_t x, bNode *t) {
    if (t == NULL)
        return t;       // Item not found; do nothing

    if (x < t->info) {
        t->left = delete(x, t->left);
    } else if (x > t->info) {
        t->right = delete(x, t->right);
    } else if (t->left != NULL && t->right != NULL) { // Two children
        t->info = findMin(t->right)->info;
        t->right = delete(t->info, t->right);
    } else {
        if (t->left != NULL) {
            tmp = t->left;
            free(t);
            return tmp;
        }
        else {
            tmp = t->right;
            free(t);
            return tmp;
        }
    }
    return t;
}

void deleteT(BSTree *tree, info_t x) {
    tree->root = delete(x, tree->root);
}
```

Бинарни пребарувачки дрва - Java

```
public class BNode<E extends Comparable<E>> {

    public E info;
    public BNode<E> left;
    public BNode<E> right;

    public BNode(E info) {
        this.info = info;
        left = null;
        right = null;
    }

    public BNode(E info, BNode<E> left, BNode<E> right) {
        this.info = info;
        this.left = left;
        this.right = right;
    }
}

public class BinarySearchTree<E extends Comparable<E>> {
    private BNode<E> root;

    public BinarySearchTree() {
        root = null;
    }
}
```

Бинарни пребарувачки дрва - Java

- void insert(E x);

  Додади елемент во дрвото
- void remove(E x);

  Отстрани го елементот x од дрвото
- BNode<E> find(E x);

  Пронајди го јазелот што го содржи инфо полето x
- E findMin();

  Врати го најмалиот елемент во дрвото

Бинарни пребарувачки дрва - Java

- E findMax();

  Врати го најголемиот елемент во дрвото
- boolean isEmpty();

  Врати true ако дрвото е празно
- void makeEmpty();

  Испразни го дрвото
- void printTree();

  Испечати го дрвото во сортиран редослед

Бинарни пребарувачки дрва - Java

```
private BNode<E> insert(E x, BNode<E> t) {
    if (t == null) {
        t = new BNode<E>(x, null, null);
    } else if (x.compareTo(t.info) < 0) {
        t.left = insert(x, t.left);
    } else if (x.compareTo(t.info) > 0) {
        t.right = insert(x, t.right);
    } else;  // Duplicate; do nothing
    return t;
}

public void insert(E x) {
    root = insert(x, root);
}
```

Бинарни пребарувачки дрва - Java

```
private BNode<E> findMin(BNode<E> t) {
    if (t == null) {
        return null;
    } else if (t.left == null) {
        return t;
    }
    return findMin(t.left);
}

public E findMin() {
    return elementAt(findMin(root));
}

private BNode<E> findMax(BNode<E> t) {
    if (t == null) {
        return null;
    } else if (t.right == null) {
        return t;
    }
    return findMax(t.right);
}

public E findMax() {
    return elementAt(findMax(root));
}
```

Бинарни пребарувачки дрва - Java

```
private BNode<E> find(E x, BNode<E> t) {
    if (t == null)
        return null;

    if (x.compareTo(t.info) < 0) {
        return find(x, t.left);
    } else if (x.compareTo(t.info) > 0) {
        return find(x, t.right);
    } else {
        return t;    // Match
    }
}

public BNode<E> find(E x) {
    return find(x, root);
}
```

Бинарни пребарувачки дрва - Java

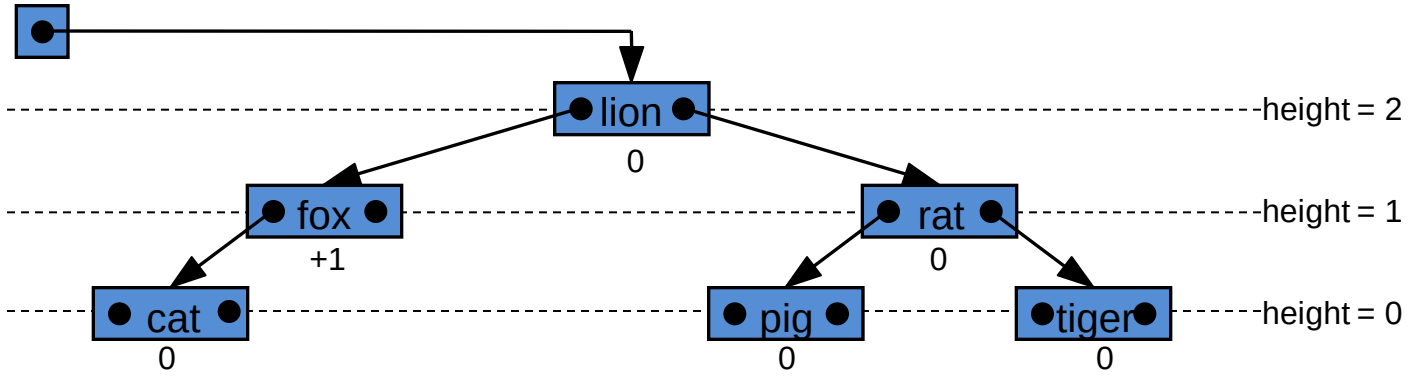
```
private BNode<E> remove(Comparable x, BNode<E> t) {
    if (t == null)
        return t;   // Item not found; do nothing

    if (x.compareTo(t.info) < 0) {
        t.left = remove(x, t.left);
    } else if (x.compareTo(t.info) > 0) {
        t.right = remove(x, t.right);
    } else if (t.left != null && t.right != null) { // Two children
        t.info = findMin(t.right).info;
        t.right = remove(t.info, t.right);
    } else {
        if (t.left != null)
            return t.left;
        else
            return t.right;
    }
    return t;
}

public void remove(E x) {
    root = remove(x, root);
}
```

AVL дрва

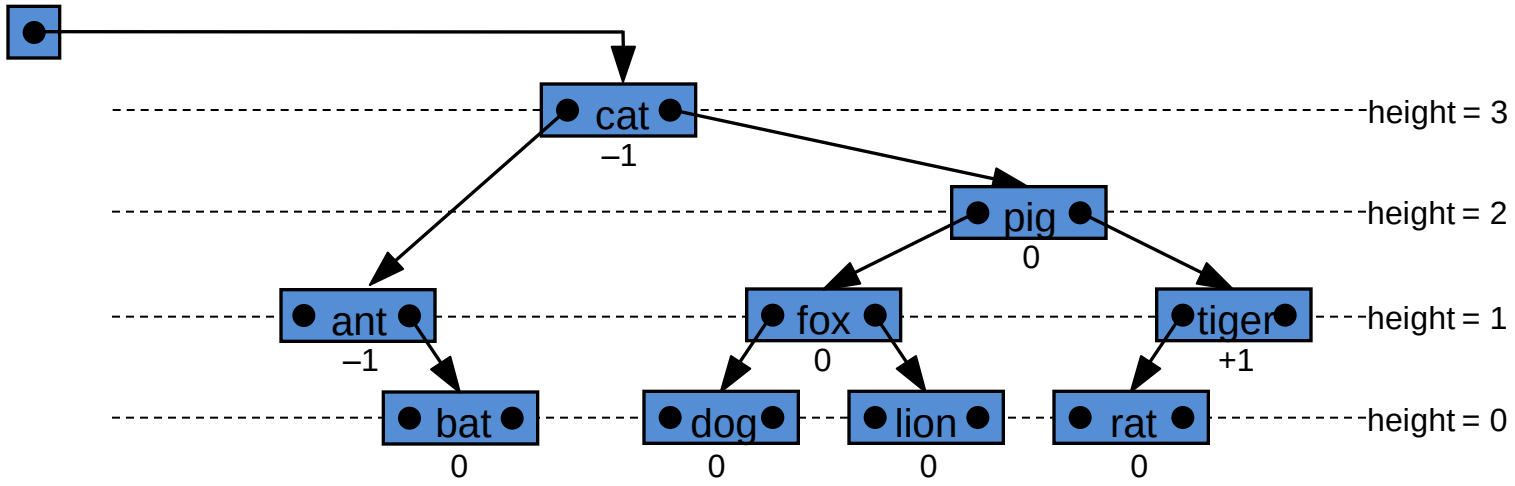
- AVL дрва се балансирани бинарни пребарувачки дрва
- Фактор на балансираност на јазел
  - висина(лево поддрво) - висина(десно поддрво)
- AVL дрвата имаат фактор на балансираност кој се пресметува за секој јазел
  - Кај секој јазел, висините на левото и десното поддрво не смеат да се разликуваат за повеќе од 1
  - Моменталните висини се зачувуваат во секој јазел

Примери

(a)


lion
0
fox
+1
rat
0
cat
0
pig
0
tiger
0
height = 2
height = 1
height = 0

(b)


cat
–1
pig
0
ant
–1
fox
0
tiger
+1
bat
0
dog
0
lion
0
rat
0
height = 3
height = 2
height = 1
height = 0

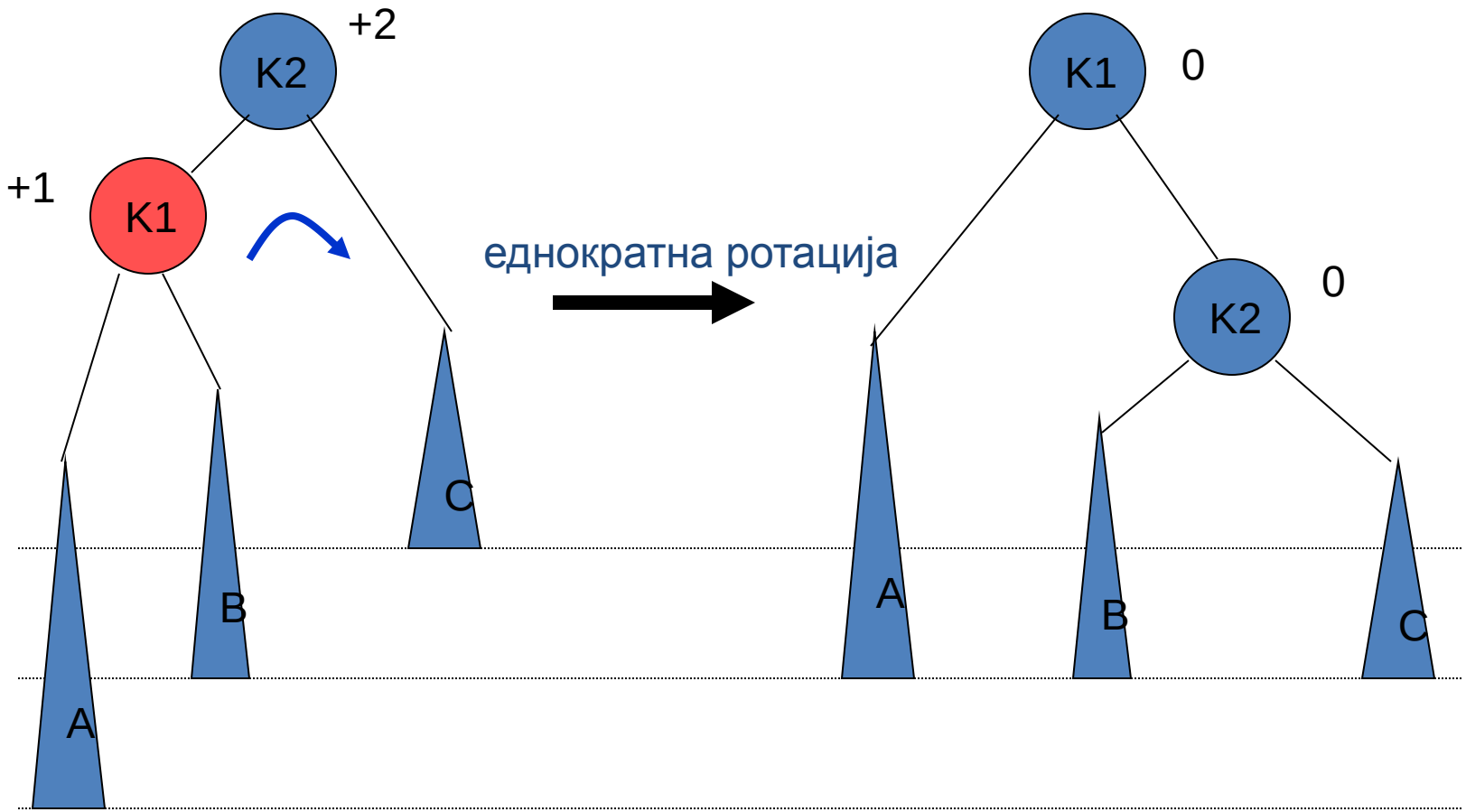
Додавање на јазел

- Операцијата за додавање на нов јазел може да предизвика факторот на балансираност да биде 2 или -2 за некој јазел
- Само јазлите што се наоѓаат на патот од новододаденият јазел до коренот може да имаат промена во висината
- После додавање на елемент, треба да се придвижува нагоре кон коренот (јазел по јазел) и да се ажурираат висините
- Ако новиот фактор на балансираност е 2 или -2, треба да се прилагоди дрвото со ротации околу јазелот

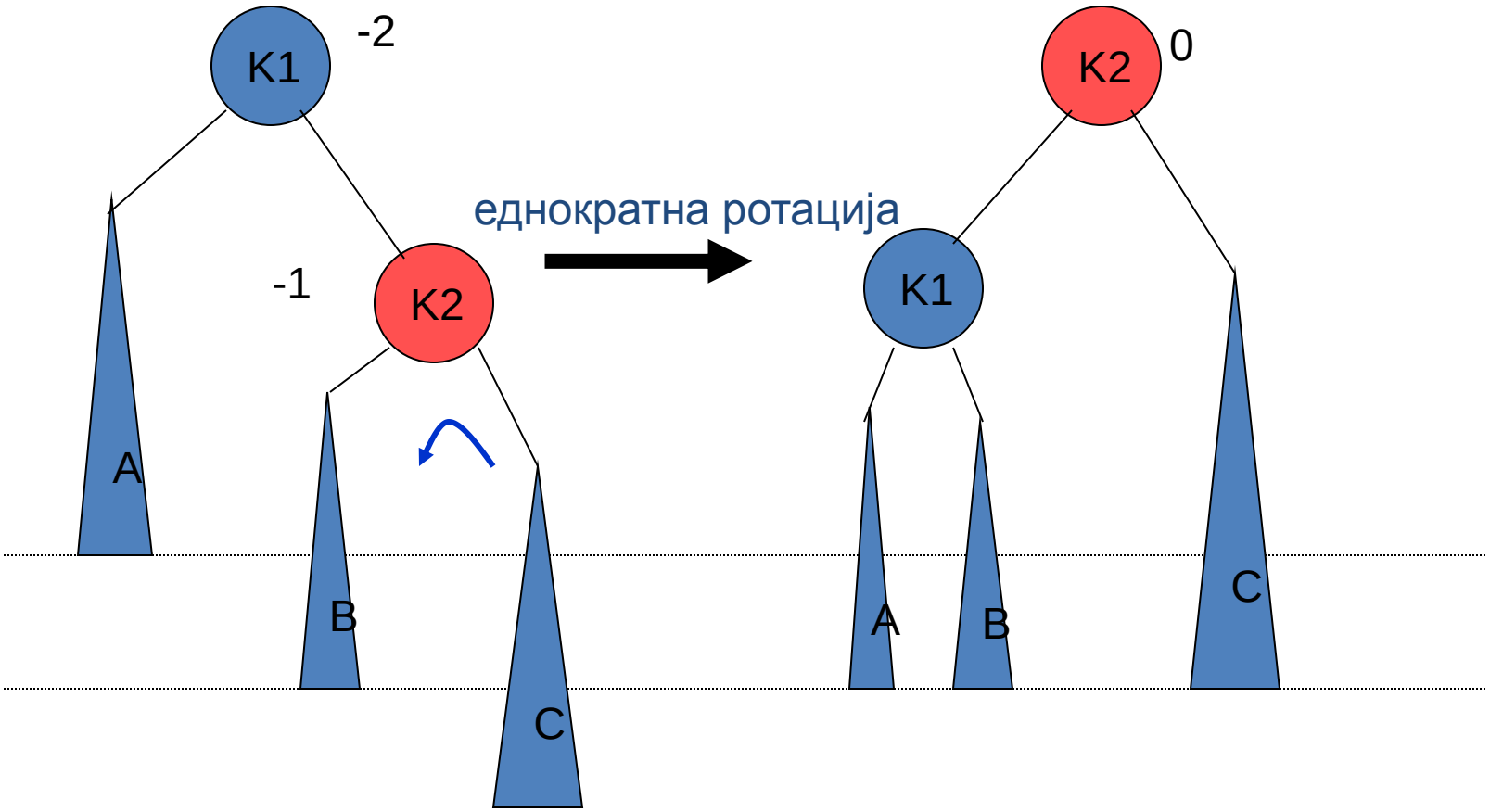
Додавање на јазел

- Нека јазелот што треба да се балансира е A
- Постојат 4 случаи:
  - Надворешни случаи (еднократна ротација):
    1. Додавање во левото поддрво на левото дете на A
    2. Додавање во десното поддрво на десното дете на A
  - Внатрешни случаи (двократна ротација):
    3. Додавање во десното поддрво на левото дете на A
    4. Додавање во левото поддрво на десното дете на A
- Ребалансирањето се прави со 4 посебни алгоритми за ротација

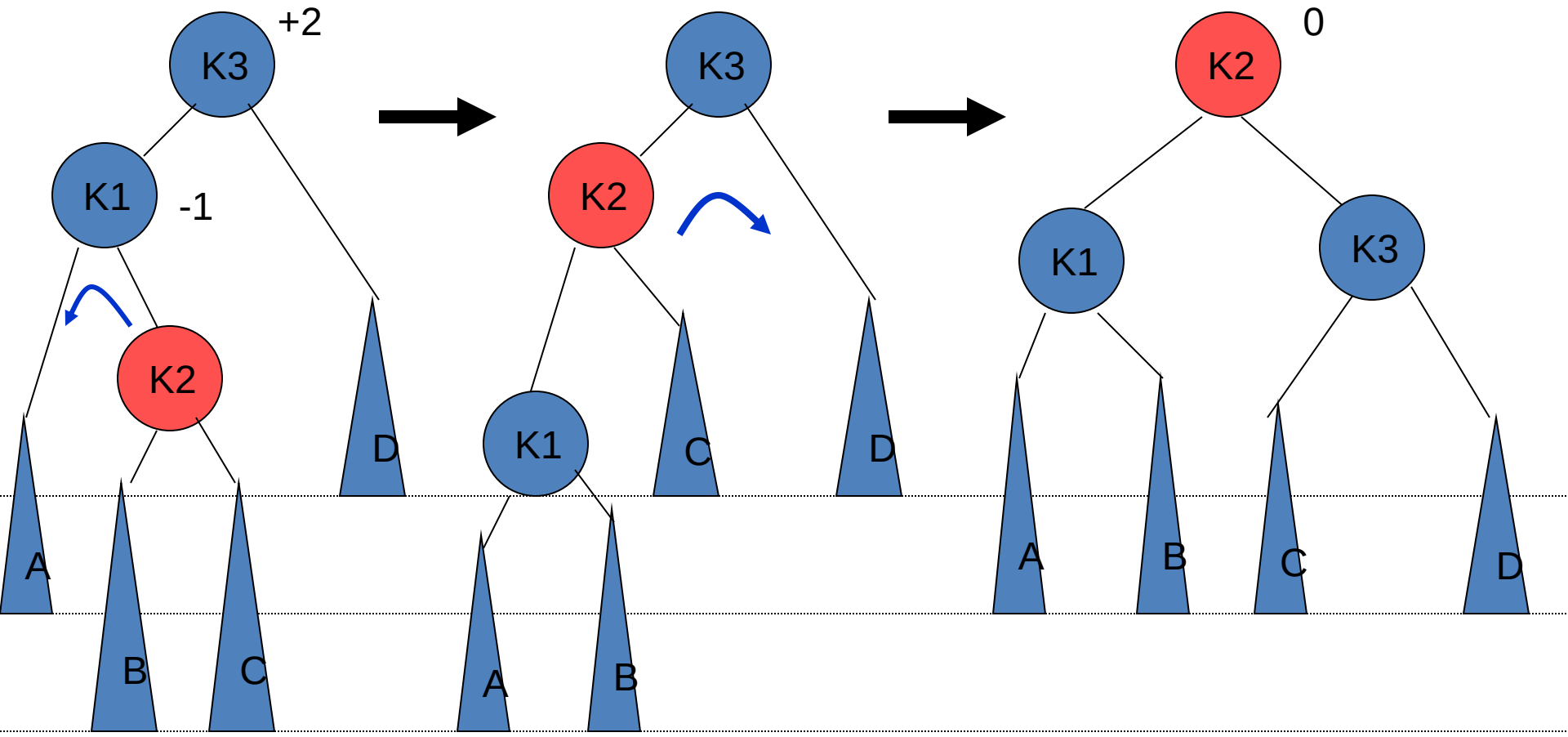
Случај 1: лево-лево ротација



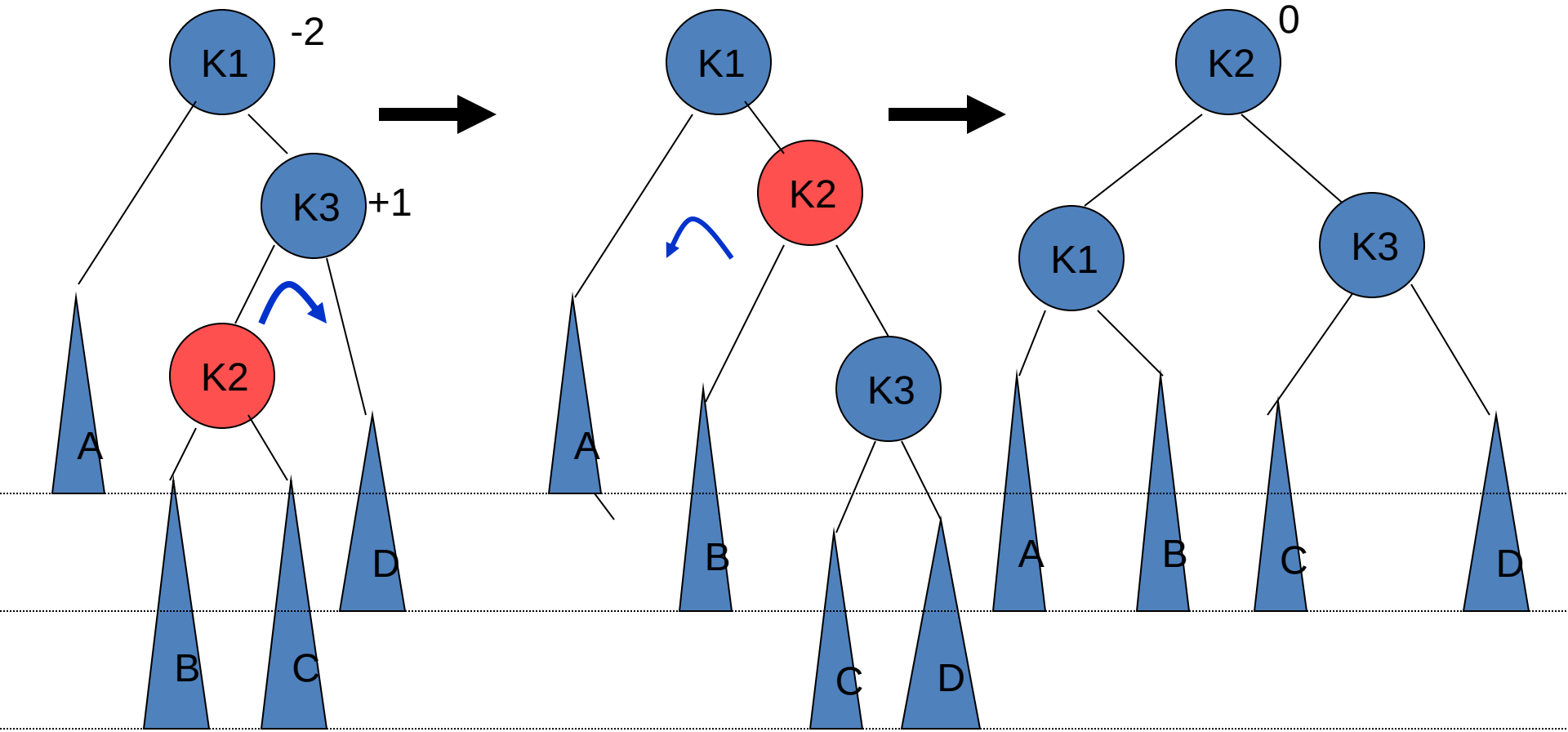
Случај 2: десно-десно ротација



Случај 3: лево-десно ротација



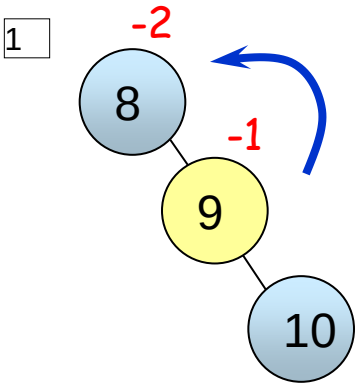
Случај 4: десно-лево ротација



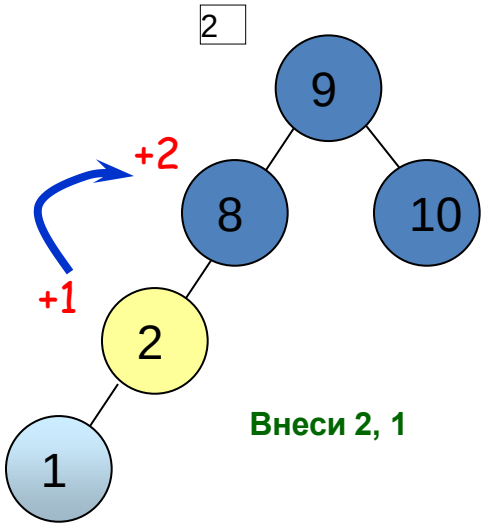
Пример

- Внесете ги следните броеви во празно AVL дрво:
  - 8, 9, 10, 2, 1, 5, 3, 6, 4, 7, 11, 12

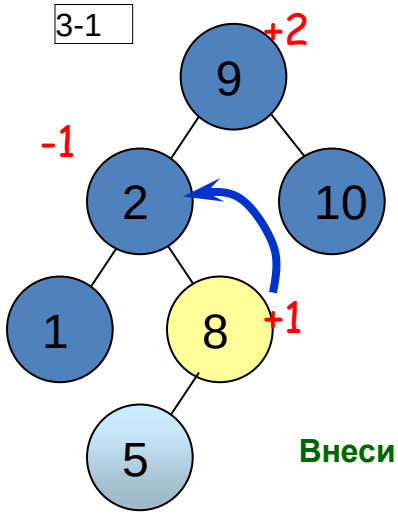
Пример



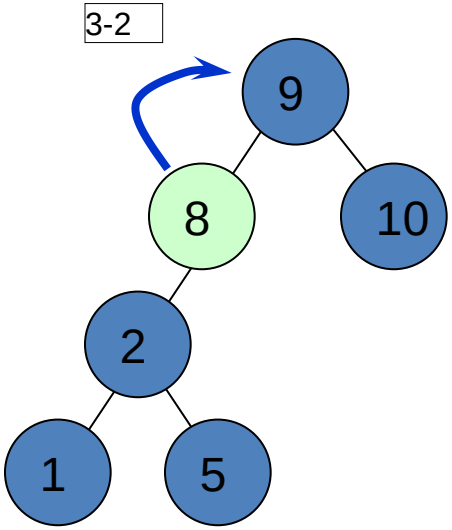
Внеси 8, 9, 10

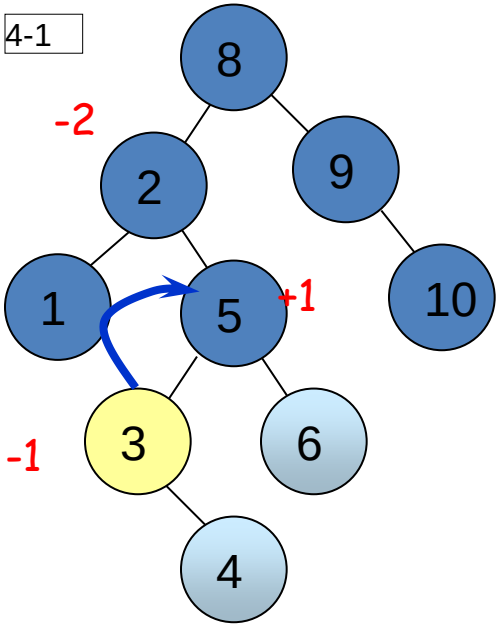


Внеси 2, 1



Внеси 5

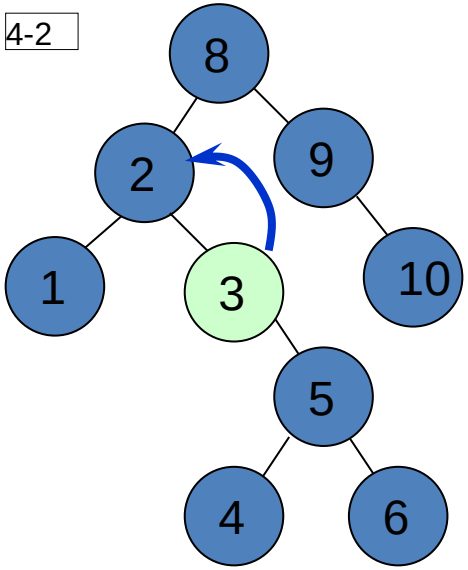




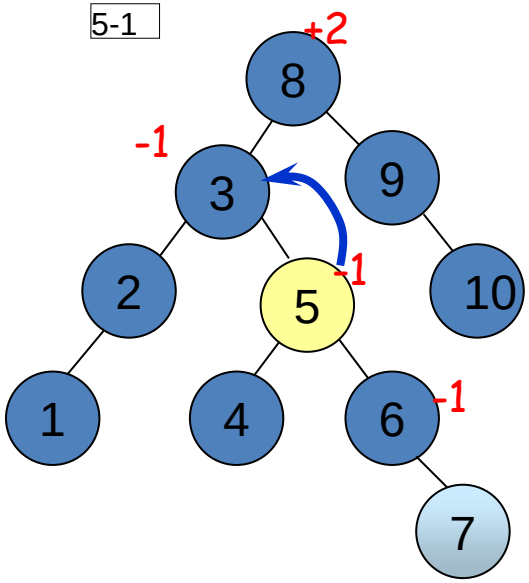
Внеси 3, 6, 4

Пример

4-2

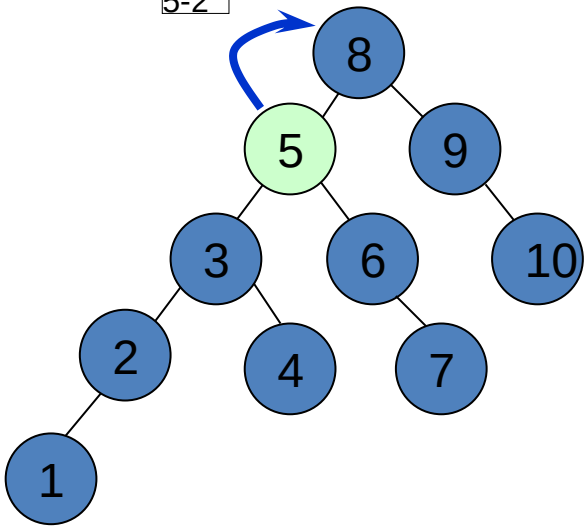


5-1

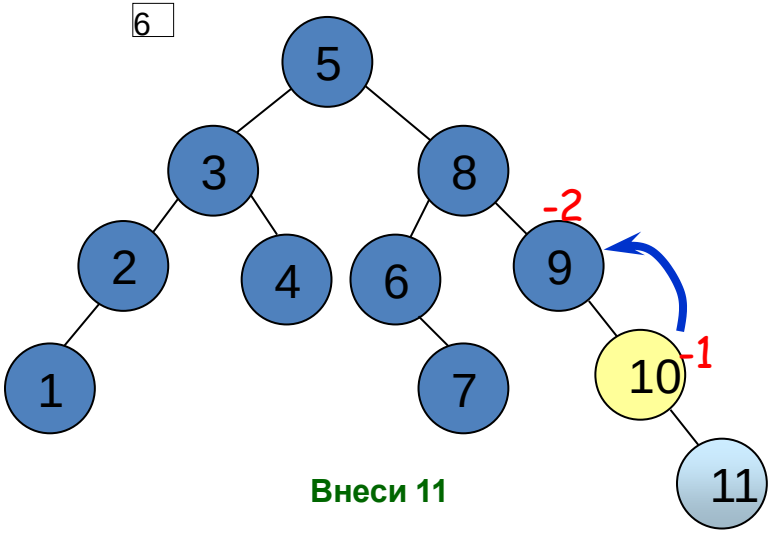


Внеси 7

5-2

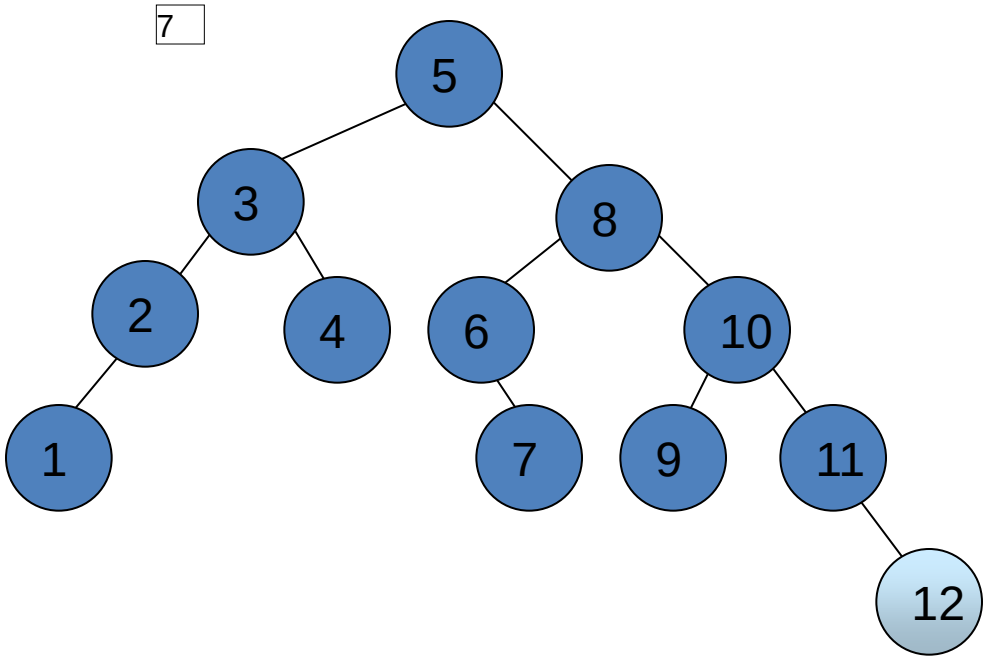


6



Внеси 11

Пример



Внеси 12

Бришење на јазел

- Бришење на јазел се одвива на сличен начин со балансирање на сите јазли кои што имаат нарушена балансираност по патеката до коренот
- Ако новиот фактор на балансираност е 2 или -2, треба да се прилагоди дрвото со еднократни или двократни ротации околу јазелот

AVL дрва - C

```
typedef int info_t;

typedef struct AVLNodeType {
    info_t info;
    struct AVLNodeType *left;
    struct AVLNodeType *right;
    int height;
} AVLNode;

typedef struct AVLTreeType {
    AVLNode *root;
} AVLTree;

void initialize(AVLTree *t) {
    t->root = NULL;
}
```

AVL дрва - C

- void insertT(AVLTree *t, info_t x);

  Додади елемент во дрвото tree
- void deleteT(AVLTree *tree, info_t x);

  Отстрани го елементот x од дрвото tree
- AVLNode* findT(AVLTree *tree, info_t x);

  Пронајди го јазелот што го содржи инфо полето x
- info_t findMinT(AVLTree *tree);

  Врати го најмалиот елемент во дрвото

AVL дрва - C

- info_t findMaxT(AVLTree *tree);

  Врати го најголемиот елемент во дрвото
- int isEmptyT(AVLTree *tree);

  Врати 1 ако дрвото е празно
- void makeEmptyT(AVLTree *tree);

  Испразни го дрвото
- void printTreeT(AVLTree *tree);

  Испечати го дрвото во сортиран редослед

AVL дрва - C

```
int height(AVLNode *t) {
    if (t == NULL)
        return -1;
    else
        return t->height;
}

int max(int lhs, int rhs) {
    if (lhs > rhs)
        return lhs;
    else
        return rhs;
}

// Get Balance factor of node N
int getBalance(AVLNode *n) {
    if (n == NULL)
        return 0;
    return height(n->left) - height(n->right);
}
```

AVL дрва - C

```
AVLNode* rotateWithLeftChild(AVLNode *k2) {
    AVLNode* k1 = k2->left;
    k2->left = k1->right;
    k1->right = k2;
    k2->height = max(height(k2->left), height(k2->right)) + 1;
    k1->height = max(height(k1->left), k2->height) + 1;
    return k1;
}

AVLNode* rotateWithRightChild(AVLNode *k1) {
    AVLNode *k2 = k1->right;
    k1->right = k2->left;
    k2->left = k1;
    k1->height = max(height(k1->left), height(k1->right)) + 1;
    k2->height = max(height(k2->right), k1->height) + 1;
    return k2;
}

AVLNode* doubleWithLeftChild(AVLNode *k3) {
    k3->left = rotateWithRightChild(k3->left);
    return rotateWithLeftChild(k3);
}

AVLNode* doubleWithRightChild(AVLNode *k1) {
    k1->right = rotateWithLeftChild(k1->right);
    return rotateWithRightChild(k1);
}
```

AVL дрва - C

```
AVLNode* insert(info_t x, AVLNode *t) {
    if (t == NULL) {
        t = createNewNode(x, NULL, NULL);
    } else if (x < t->info) {
        t->left = insert(x, t->left);
        if (height(t->left) - height(t->right) == 2) {
             if (x < t->left->info) {
                t = rotateWithLeftChild(t);
            } else {
                t = doubleWithLeftChild(t);
            }
        }
    } else if (x > t->info) {
        t->right = insert(x, t->right);
        if (height(t->right) - height(t->left) == 2) {
            if (x > t->right->info) {
                t = rotateWithRightChild(t);
            } else {
                t = doubleWithRightChild(t);
            }
        }
    } else ;  // Duplicate; do nothing
```

AVL дрва - C

```
    t->height = max(height(t->left), height(t->right)) + 1;
    return t;
}

void insertT(AVLTree *t, info_t x) {
    t->root = insert(x, t->root);
}
```

AVL дрва - Java

```
public class AVLNode<E extends Comparable<E>> {

    public E info;
    public AVLNode<E> left;
    public AVLNode<E> right;
    public int height;

    AVLNode(E theElement) {
        this(theElement, null, null);
    }

    AVLNode(E info, AVLNode<E> left, AVLNode<E> right) {
        this.info = info;
        this.left = left;
        this.right = right;
        height = 0;
    }

}

public class AVLTree<E extends Comparable<E>> {
    private AVLNode<E> root;

    public AVLTree() {
        root = null;
    }
}
```

AVL дрва - Java

- void insert(E x);

  Додади елемент во дрвото
- void remove(E x);

  Отстрани го елементот x од дрвото
- AVLNode<E> find(E x);

  Пронајди го јазелот што го содржи инфо полето x
- E findMin();

  Врати го најмалиот елемент во дрвото

AVL дрва - Java

- E findMax();

  Врати го најголемиот елемент во дрвото
- boolean isEmpty();

  Врати true ако дрвото е празно
- void makeEmpty();

  Испразни го дрвото
- void printTree();

  Испечати го дрвото во сортиран редослед

AVL дрва - Java

```
 private int height(AVLNode<E> t) {
    if (t == null)
        return -1;
    else
        return t.height;
}

private int max(int lhs, int rhs) {
    if (lhs > rhs)
        return lhs;
    else
        return rhs;
}

int getBalance(AVLNode<E> n) {
    if (n == null)
        return 0;
    return height(n.left) - height(n.right);
}
```

AVL дрва - Java

```
 private AVLNode<E> rotateWithLeftChild(AVLNode<E> k2) {
    AVLNode<E> k1 = k2.left;
    k2.left = k1.right;
    k1.right = k2;
    k2.height = max(height(k2.left), height(k2.right)) + 1;
    k1.height = max(height(k1.left), k2.height) + 1;
    return k1;
}

private AVLNode<E> rotateWithRightChild(AVLNode<E> k1) {
    AVLNode<E> k2 = k1.right;
    k1.right = k2.left;
    k2.left = k1;
    k1.height = max(height(k1.left), height(k1.right)) + 1;
    k2.height = max(height(k2.right), k1.height) + 1;
    return k2;
}

private AVLNode<E> doubleWithLeftChild(AVLNode<E> k3) {
    k3.left = rotateWithRightChild(k3.left);
    return rotateWithLeftChild(k3);
}

private AVLNode<E> doubleWithRightChild(AVLNode<E> k1) {
    k1.right = rotateWithLeftChild(k1.right);
    return rotateWithRightChild(k1);
}
```

AVL дрва - Java

```
private AVLNode<E> insert(E x, AVLNode<E> t) {
    if (t == null) {
        t = new AVLNode<E>(x, null, null);
    } else if (x.compareTo(t.info) < 0) {
        t.left = insert(x, t.left);
        if (height(t.left) - height(t.right) == 2) {
            if (x.compareTo(t.left.info) < 0) {
                t = rotateWithLeftChild(t);
            } else {
                t = doubleWithLeftChild(t);
            }
        }
    } else if (x.compareTo(t.info) > 0) {
        t.right = insert(x, t.right);
        if (height(t.right) - height(t.left) == 2) {
            if (x.compareTo(t.right.info) > 0) {
                t = rotateWithRightChild(t);
            } else {
                t = doubleWithRightChild(t);
            }
        }
    } else;  // Duplicate; do nothing
```

AVL дрва - Java

```
t.height = max(height(t.left), height(t.right)) + 1;
    return t;
}

public void insert(E x) {
    root = insert(x, root);
}
```