Aibės. Medžiai.

Lekt. Darius Matulis, 2022

Sąrašo ADT realizacijų algoritmų sudėtingumai

ktu

- Sąrašo ADT (***ListADT***) gali būti realizuotas įvairiomis duomenų struktūromis: masyvu, įvairiais tiesiniais dinamiškaisiais sąrašais.

- Sąrašo ADT dinamiškosiose realizacijose daugumos pagrindinių operacijų algoritmų sudėtingumas - **O(N)**.

- Sąrašo ADT realizacijoje masyvu kai kurių operacijų algoritmų sudėtingumai:

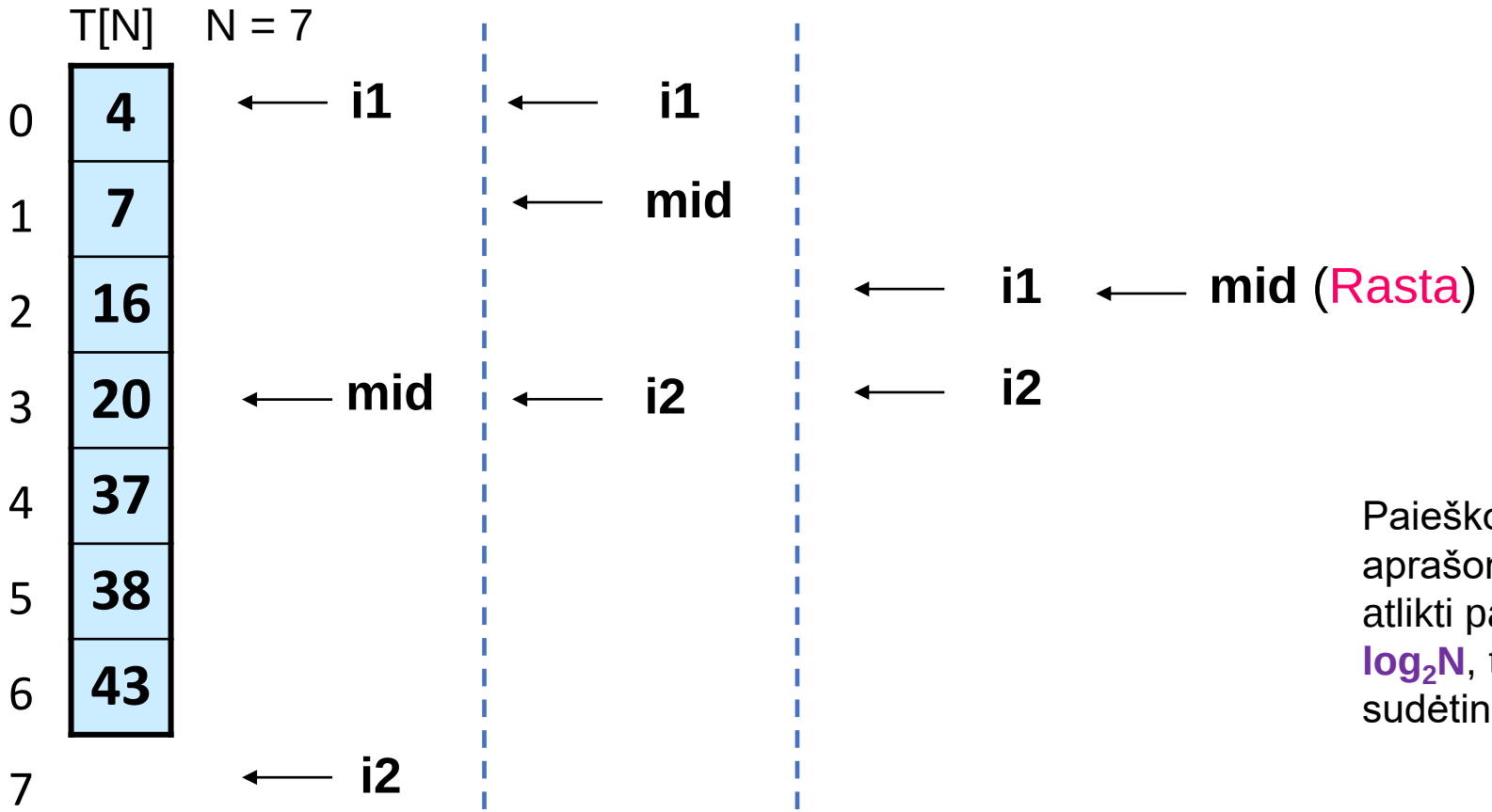
	Netvarkus	Surikiuotas
get(int nr)	**O(1)** (greitas priėjimas prie visų elementų)	
indexOf(E element)	**O(N)** (nuosekli paieška)	**O(log_2N)** (dvejetainė paieška)
add(E element, int nr) ***remove(E element, int nr)***	**O(N)** (greitas priėjimas prie visų elementų + elementų perstūmimas)	

- **O(N)** – labai lėtai. Ar galima pasiekti logaritminį (**O(log_2N)**) ADT (nebūtinai Sarašo ADT) daugelio operacijų algoritmų sudėtingumą? **Pasiaiškinkime kaip.**

Kas tas O(log_2N)?

ktu

Pasinaudokime dvejetainės paieškos algoritmu, kuris taikomas tik surikiuotų elementų masyvui. Ieškosime 16.



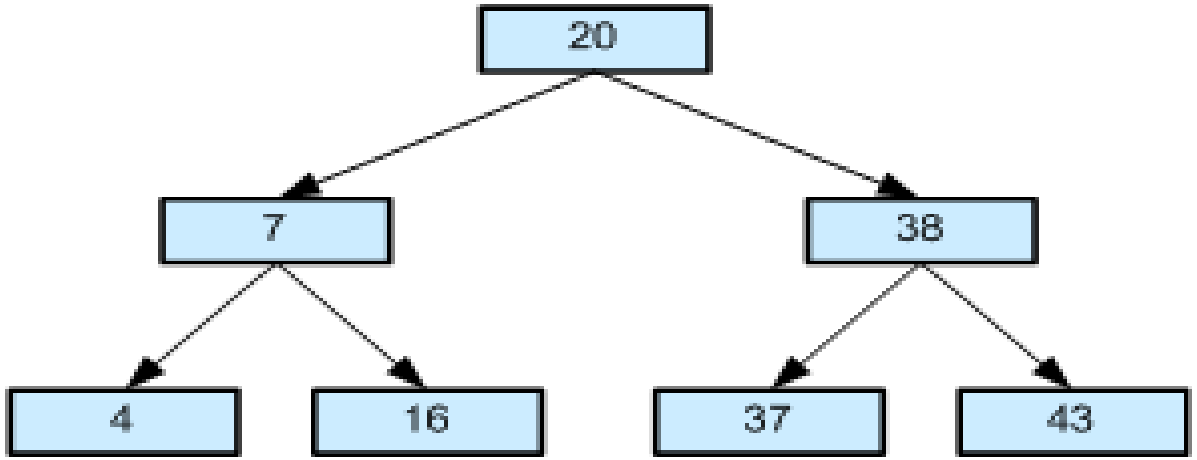
Paieškos erdvės mažėjimas aprašomas $N = 2^i$, kur i – reikiami atlikti palyginimai. Išsireiškus, $i = log_2N$, t.y. algoritmo sudėtingumas - $O(log_2N)$.

Medis

ktu

	T[7]
0	4
1	7
2	16
3	20
4	37
5	38
6	43

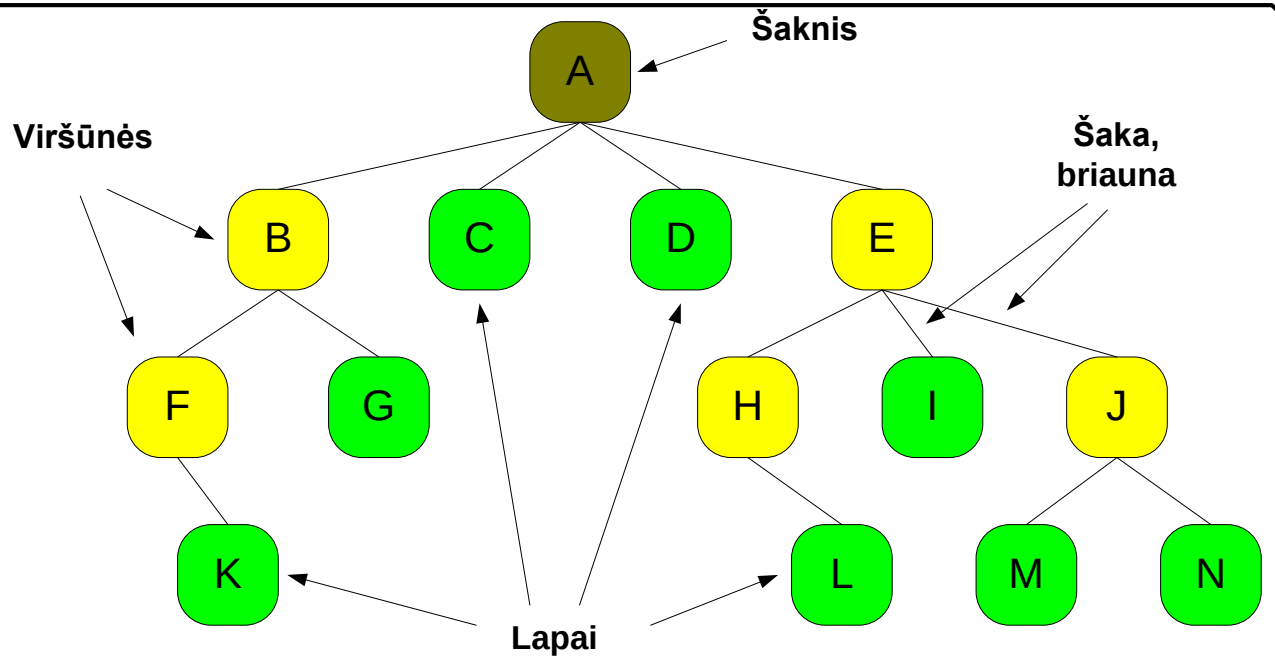
- Dvejetainės paieškos algoritmu galimai vykdomą palyginimų seką galima aprašyti medžiu:



- **Bendrasis Medis (General Tree)** - hierarchinė struktūra, kurioje esybės susiejamos hierarchiniais ryšiais.
- **Medžiai** bendru atveju leidžia atlikti pagrindines ADT operacijas su **$O(\log_L N)$** sudėtingumu, kur N – medžio viršūnių (elementų) skaičius, L – viršūnės vaikų skaičius.

Medžių terminologija (1)

ktu

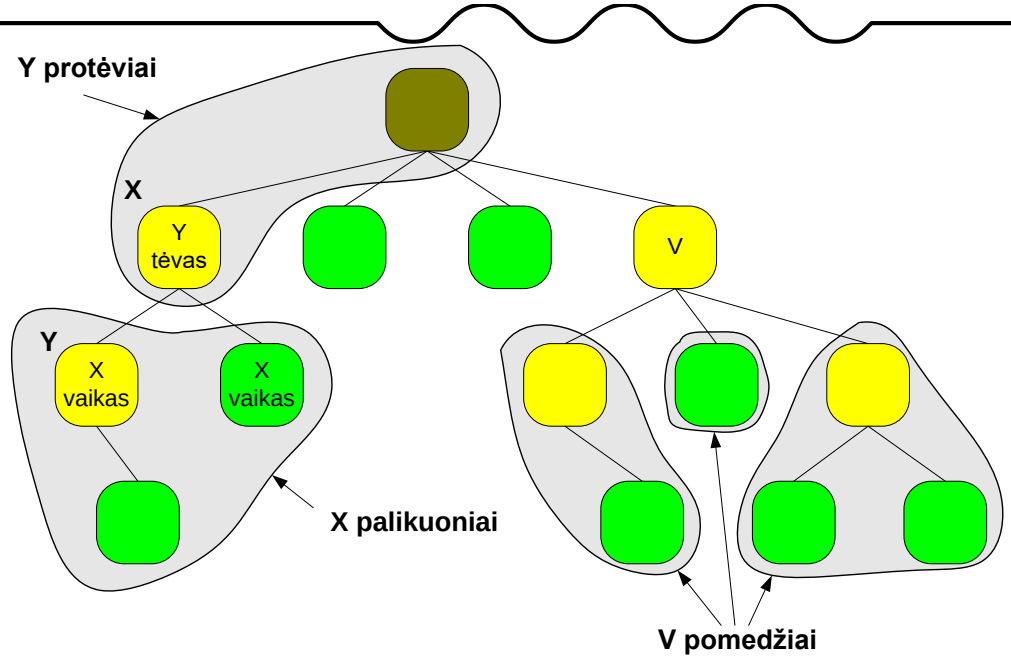


Šaknis: išskirtinė medžio viršūnė, medžio pradžia. Medyje yra tik viena šaknis.
Viršūnės: medžio elementai.
Lapinė viršūnė, *lapas*: viršūnė be vaikų.
Vidinė viršūnė: viršūnė, kuri nėra lapas.
Briauna, *šaka*: sujungimai tarp viršūnių.
Tuščias medis: neturi nei viršūnių nei briaunų.

Medžių terminologija (2)

ktu

Tėvas: viršūnė, esanti hierarchiškai viršuje. Viršūnė gali turėti tik vieną tėvą.
Vaikas: viršūnė, esanti hierarchiškai apačioje.
Viršūnės Y protėviai: Y tėvas, Y tėvo tėvas..
Viršūnės X palikuoniai: X vaikai, X vaikų vaikai..
Viršūnės V pomedis: Vienas V vaikas ir visi jo palikuoniai.



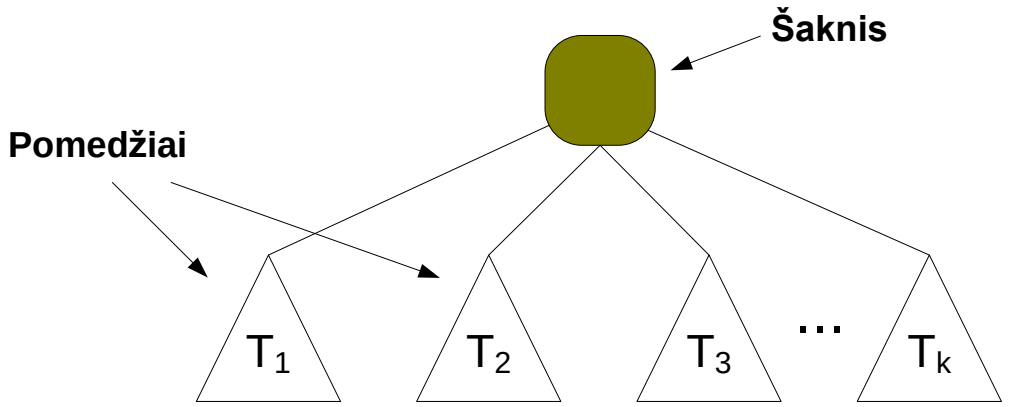
Kelias: viršūnių seka nuo vienos viršūnės į kitą.
Kelio ilgis: viršūnių skaičius kelyje.
Viršūnės gylis (lygis): kelio nuo šaknies iki viršūnės ilgis.
Viršūnės aukštis: ilgiausio kelio nuo viršūnės iki lapo ilgis.
Viršūnės laipsnis: viršūnės vaikų skaičius.
Medžio laipsnis: maksimalus medžio viršūnių laipsnis.
Medžio aukštis (gylis): šaknies aukštis (lapai yra nuliniame lygyje).

Bendrojo medžio apibrėžimas

ktu

Bendrasis medis – tai ciklų neturintis orientuotas grafas, aprašomas kaip briaunomis sujungtų viršūnių aibė:

- Medžio šaknis – medžio pradžios viršūnė.
- Medis yra tuščias, kai jo viršūnių aibė yra tuščia.
- Netuščią medį sudaro medžio šaknis ir bet koks (0..∞) jos pomedžių T_1, T_2, … T_k (k = 1..∞), skaičius.
- Kiekvienas pomedis taip pat yra medis, susidedantis iš viršūnės ir bet kokio jos pomedžių skaičiaus.



Trumpai apie rekursiją (1)

ktu

- Metodas, kuris gali kreiptis į save patį ir dar kartą save įvykdyti nuo pradžios, tik su kitais parametrais, yra vadinamas **rekursiniu**, o pakartotinis metode realizuoto algoritmo vykdymas - **rekursija.**

- Paprastai tariant, rekursiniai algoritmai savo programinės realizacijos kodu iškviečia savo paties kopijas ir sudaro metodų **rekursinę grandinėlę**, todėl rekursija labai primena „veidrodžio atspindį veidrodyje“ ☺.

Trumpai apie rekursiją (2)

ktu

- Kiekviena rekursiškai iškviesto metodo kopija turi nuosavus kintamuosius, tačiau metodo parametrais galima perduoti kintamuosius iš vienos metodo kopijos į kitą.

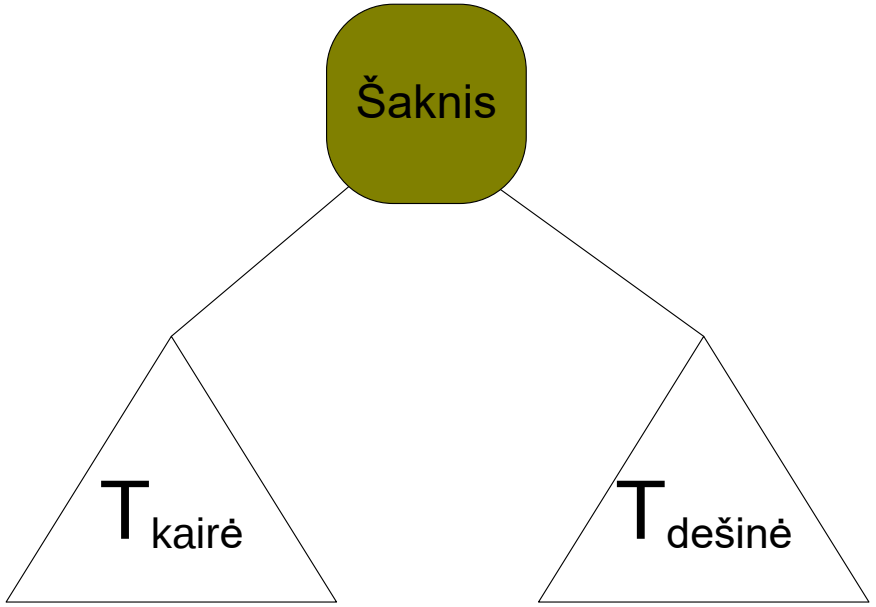
- **Rekursijos gylis** – rekursiškai iškviestų metodo kopijų skaičius. Dažna programuotojo klaida – per didelis iškviečiamų metodo kopijų skaičius, tokiu atveju programos stekas persipildo iki Java VM leidžiamo dydžio, sugeneruojama StackOverFlow situacija ir metodas nutraukiamas.

- Rekursija ypatingai dažnai naudojama medžių duomenų struktūrų apibrėžimuose ir algoritmų realizacijose, kadangi leidžia keliauti ne tik iš tėvo į vaiką, bet ir atgal **nenaudojant papildomos mazgo rodyklės į tėvą** (**tuo dar įsitikinsime**).

Dvejetainis medis

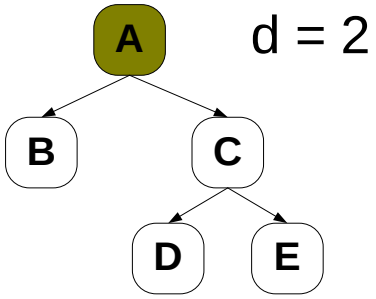
ktu

- Medžių teoriją pradėsime aiškintis nuo paprasčiausio **dvejetainio medžio**.
- **Rekursinis dvejetainio medžio apibrėžimas.** Tai bendrasis medis, kuriame bet kuri viršūnė gali turėti ne daugiau kaip du pomedžius (kairįjį ir dešinįjį), kurie taip pat yra dvejetainiai medžiai.



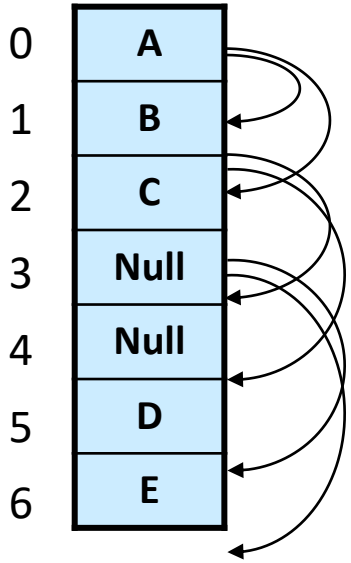
Dvejetainio medžio realizacija masyve

ktu



d = 2

T[i] i =7



Bendrojo dvejetainio medžio viršūnes galime talpinti vienmačiame masyve, nustatant santykius tarp kiekvienos viršūnės ir jos vaikų masyvo indeksų:

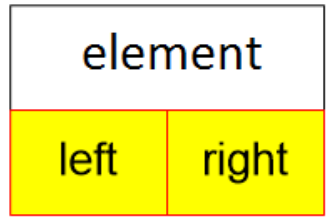
T[i] kairysis vaikas	T[2*i+1]
T[i] dešinysis vaikas	T[2*i+2]
T[i] tėvas	T[(i-1) div 2]

Aukščio d dvejetainiam medžiui reikės 2^{d+1}-1 ilgio masyvo. Tačiau ši realizacija **yra gana reta**, todėl detaliau jos nenagrinėsime.

Dinamiškosios dvejetainio medžio realizacijos mazgas

- Realizuojant dvejetainį medį dinamiškai, reikia sukurti mazgo klasę, kurioje būtų elementas (viršūnė) ir rodyklės į kairįjį ir dešinįjį pomedžius.
- Mazgą geriausiai realizuoti klasės vidine klase, tačiau yra galima realizacija ir atskiroje klasėje.

```
protected class BstNode<N> {

   // Elementas
   protected N element;
   // Rodyklė į kairįjį pomedį
   protected BstNode<N> left;
   // Rodyklė į dešinįjį pomedį
   protected BstNode<N> right;

   protected BstNode() {
   }

   protected BstNode(N element)
{
      this.element = element;
      this.left = null;
      this.right = null;
   }
}
```



Kitos mazgo realizacijos

ktu

Tačiau kai kuriose realizacijose raktas ir elementas mazge gali būti atskiriami, o taip pat atsirasti papildoma rodyklė į tėvą:

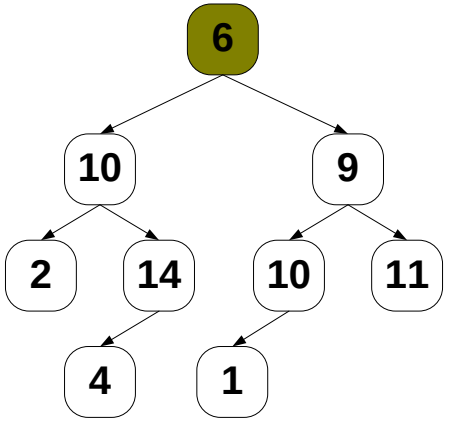
```
protected class BstNode<K, N> {

   // Raktas
   protected K key;
   // Elementas
   protected N element;
   // Rodyklė į kairįjį pomedį
   protected BstNode<K, N> left;
   // Rodyklė į dešinįjį pomedį
   protected BstNode<K, N> right;
   // Rodyklė į tėvą
   protected BstNode<K, N> parent;

   protected BstNode() {
   }

   protected BstNode(K key, N element, BstNode<K,
N> parent) {
      this.key = key;
      this.element = element;
      this.parent = parent;
      this.left = null;
      this.right = null;
   }
}
```

Dvejetainis paieškos medis (DP-medis)

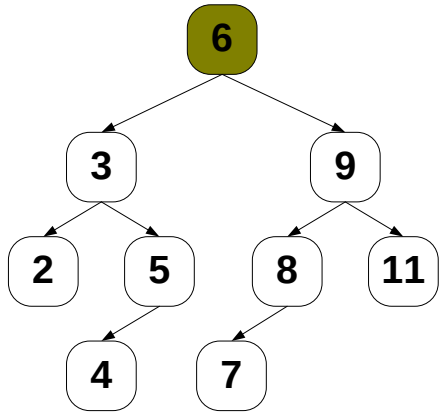
ktu

- **Kokia tvarka organizuoti viršūnes medyje?? Kokie algoritmai?**
- Raktas – viršūnės duomenys, pagal kuriuos viršūnės tarpusavyje palyginamos.
- **DP-medis (angl. Binary Search Tree (BST))** – tai dvejetainis medis, kuriame viršūnės organizuojamos pagal taisyklę: visi viršūnės kairiajame pomedyje esančių viršūnių raktai yra mažesni už viršūnės raktą, o dešiniajame – didesni.
- DP-medyje **negali** būti viršūnių su vienodais raktais.

Dvejetainis medis, bet NE dvejetainis paieškos medis!!



DP-medis (toliau visur naudosime DP-medį)

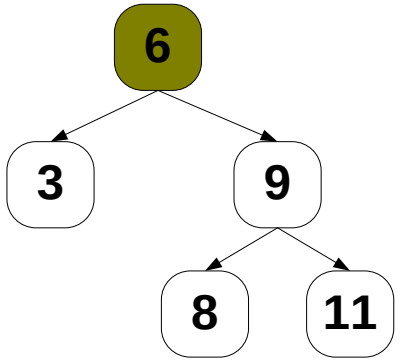


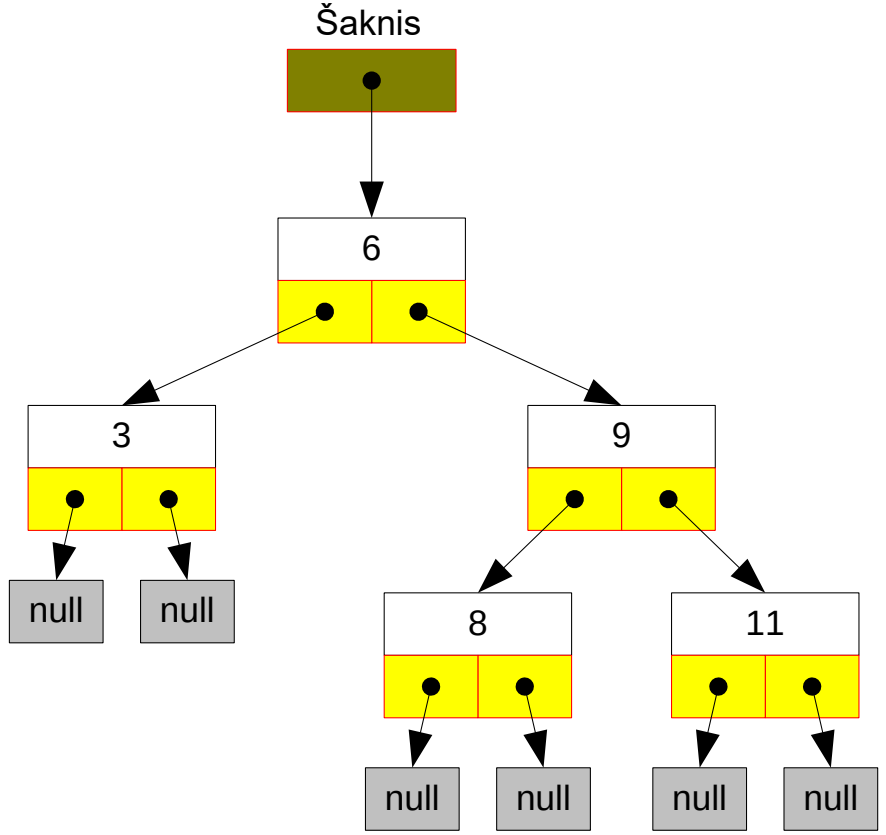
Prisiminkime. Javoje objektai lyginami įdiegiant *Comparable<T> (Natūrali tvarka)*, arba komparatoriais (*Comparator<T>*). Įdiegiant šiuos interfeisus nustatysime raktą.

DP-medžio realizacija dinamiškai

ktu

- Medis prasideda nuo rodyklės į šaknies mazgą.
- Jei mazgas neturi kairiojo arba dešiniojo pomedžio arba yra lapas, juos nurodančios rodyklės lygios *null*.





Aibės ADT (1)

ktu

- Toliau DP-medžio duomenų struktūros algoritmus aiškinsime naudodami Aibės ADT, kuris specifikuoją matematinę aibę.

- Pagrindinės aibės savybės informatikos požiūriu:

  - Aibę sudaro unikalūs elementai (**Prisiminkime**: sąraše gali būti ne tik unikalūs elementai).
  - Aibėse elementai neturi jokios apibrėžtos vietos (**Prisiminkime**: sąraše elementai turi savo vietą).
  - Aibės elementus galima grąžinti sutvarkytus tam tikra tvarka.

- Aibės ADT realizuosime DP-medžiu dinamiškai.

Aibės ADT (2)

ktu

Aibės ADT aprašomas interfeisu ***Set***:

```
public interface Set<E> extends Iterable<E> {

   //Patikrinama ar aibė tuščia.
   boolean isEmpty();

   // Grąžinamas aibėje esančių elementų kiekis.
   int size();

   // Išvaloma aibė.
   void clear();

   // Aibė papildoma nauju elementu.
   void add(E element);

   // Pašalinamas elementas iš aibės.
   void remove(E element);

   // Patikrinama ar elementas egzistuoja aibėje.
   boolean contains(E element);

   // Grąžinamas aibės elementų masyvas.
   Object[] toArray();

   // Gražinamas vizualiai išdėstytas aibės elementų turinys
   String toVisualizedString(String dataCodeDelimiter);
}
```

Aibės ADT (3)

ktu

Rikiuojamos aibės ADT aprašomas interfeisu ***SortedSet***:

```
public interface SortedSet<E> extends Set<E> {

   // Grąžinamas aibės poaibis iki element.
   Set<E> headSet(E element);

   // Grąžinamas aibės poaibis nuo element1 iki element2.
   Set<E> subSet(E element1, E element2);

   // Grąžinamas aibės poaibis nuo element.
   Set<E> tailSet(E element);

   // Grąžinamas atvirkštinis iteratorius.
   Iterator<E> descendingIterator();
}
```

Aibės ADT realizacija DP-medžiu

ktu

- Aibės ADT realizuotas DP-medžiu klasėje ***BstSet***. Klasė įdiegia interfeisą ***Set:***

```
public class BstSet<E extends Comparable<E>> implements SortedSet<E>, Cloneable {
```

Klasės duomenys

```
    // Medžio šaknies mazgas
    protected BstNode<E> root = null;
    // Medžio dydis
    protected int size = 0;
    // Rodyklė į komparatorių
    protected Comparator<? super E> c = null;
```

Klasės konstruktoriai

```
    /**
     * Sukuriamas aibės objektas DP-medžio raktams naudojant Comparable<T>
     */
    public BstSet() {
        this.c = Comparator.naturalOrder();
    }

    /**
     * Sukuriamas aibės objektas DP-medžio raktams naudojant Comparator<T>
     *
     * @param c Komparatorius
     */
    public BstSet(Comparator<? super E> c) {
        this.c = c;
    }
```

Vidinis (Inorder) medžio apėjimas (1)

ktu

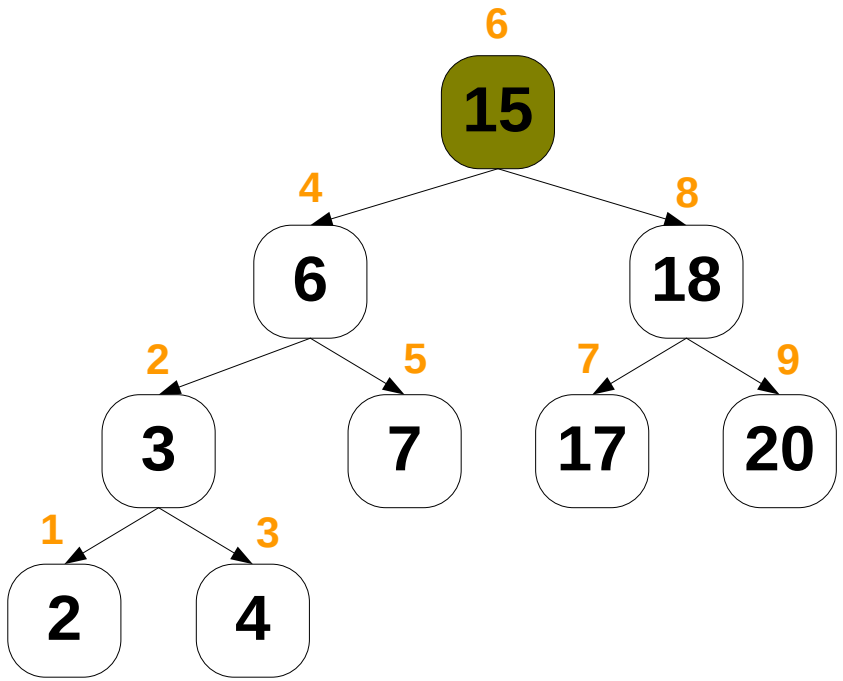
- **Medžio apėjimas** (angl. tree traversal) – operacija, kurios metu visos medžio viršūnės aplankomos vieną kartą.
- Egzistuoja trys pagrindiniai medžio apėjimo būdai: **vidinis, tiesioginis ir atvirkštinis.** Būna ir kitokių.
- Naudosime rekursinius algoritmus. Iteraciniams algoritmams (panaudosime aibės iteratoriuje) reikia panaudoti steką, arba viršūnės mazgas turi turėti papildomą rodyklę į tėvą.
- **Vidiniu apėjimu** medžio viršūnės apeinamos **rakto didėjimo tvarka**: pirmiausiai apeinamas kairysis pomedis, viršūnė, po to dešinysis pomedis.

```
Algoritmas:
FUNCTION Vidinis_apėjimas(Viršūnė)
1.IF (Viršūnė <> Null) THEN
2.   Vidinis_apėjimas(Viršūnės kairysis pomedis);
3.   Println(Viršūnė);
4.   Vidinis_apėjimas(Viršūnės dešinysis pomedis);
5.ENDIF;
END FUNCTION.
```

Vidinis (Inorder) medžio apėjimas (2)

ktu

```
Algoritmas:
FUNCTION Vidinis_apėjimas(Viršūnė)
1.IF (Viršūnė <> Null) THEN
2.  Vidinis_apėjimas(Viršūnės kairysis pomedis);
3.  Println(Viršūnė);
4.  Vidinis_apėjimas(Viršūnės dešinysis pomedis);
5.ENDIF;
END FUNCTION.
```

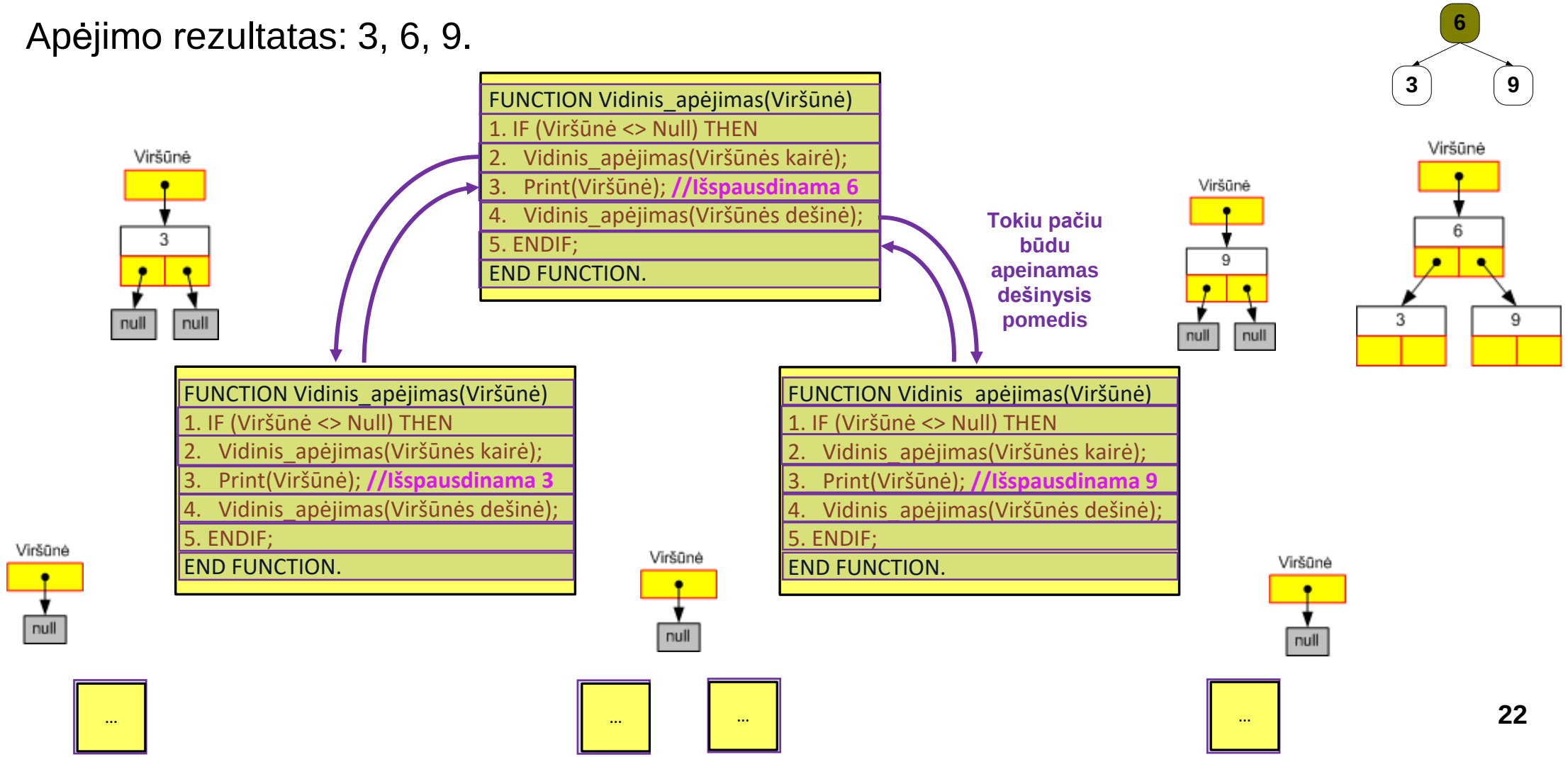


Apėjimo rezultatas:
2, 3, 4, 6, 7, 15, 17, 18, 20

Vidinio (Inorder) medžio apėjimo algoritmo iliustracija pseudokodu

ktu

Apėjimo rezultatas: 3, 6, 9.



Galimas vidinio medžio apėjimo panaudojimas metode *toString()*

ktu

- Klasės metode ***toString()*** vidiniu medžio apėjimu gali būti formuojama String eilutė, susidedanti iš rakto didėjimo tvarka sutvarkytų medžio elementų, atskirtų naujos eilutės simboliu „\n“.

```
@Override
public String toString() {
   return toStringRecursive(root);
}

private String toStringRecursive(BstNode<E> node) {
   if (node == null) {
      return "";
   }

   return toStringRecursive(node.left) +
         node.element.toString() + System.lineSeparator() +
         toStringRecursive(node.right);
}
```

Duomenų modelio klasės, pvz. Car, metodas toString()

Vidinis medžio apėjimas

Klasė Car lab2 projekte

ktu

- Klasę ***Car*** papildome 3 naujais duomenų laukais: kodu ir serijiniu numeriu, kurie yra bendri visiems šios klasės objektams ir individualiu registracijos numeriu:

```
private static final String idCode = "TA";  // ***** nauja
private static int serNr = 100;             // ***** nauja
private final String carRegNr;
```

- Registracijos numerį naudosime DP-medžio raktu. Registracijos numeris turės formą: TA101, TA102, TA103, ir t.t.:

```
carRegNr = idCode + (serNr++)
```

```
@Override
public int compareTo(Car car) {
   return getCarRegNr().compareTo(car.getCarRegNr());
}
```

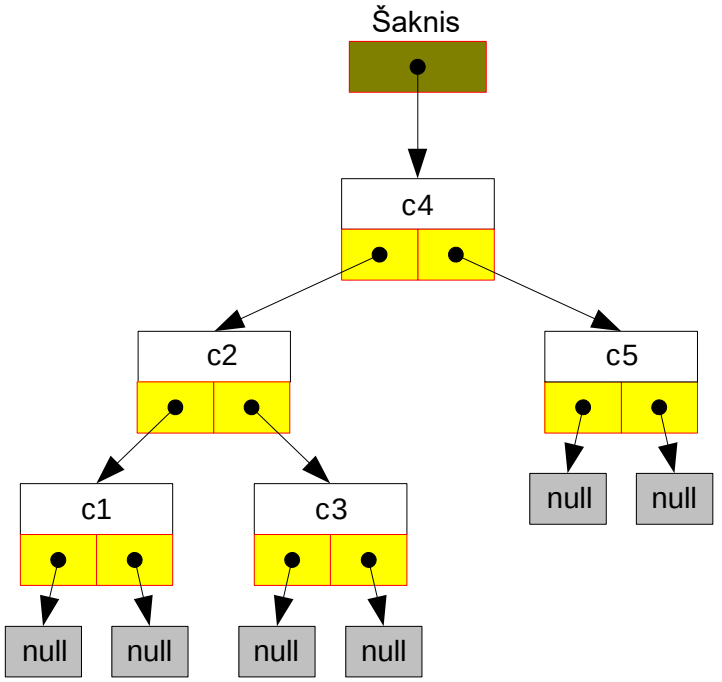
toString() panaudojimas

ktu

- Tarkime, kad suformavome penkis Car klasės objektus:

```
Car c1 = new Car("Renault", "Laguna", 1997, 50000, 1700);
Car c2 = new Car("Renault", "Megane", 2001, 20000, 3500);
Car c3 = new Car("Toyota", "Corolla", 2001, 20000, 3500);
Car c4 = new Car("Renault Laguna 2001 115900 700");
Car c5 = new Car("Renault Megane 1946 365100 9500");
```

- Suformuosime aibės objektą, kuriame saugomi šie objektai. DP-medį papildysime tokia tvarka: c4, c2, c5, c1, c3. Tada aibės objekto klasės ***BstSet*** metodas ***toString()*** suformuos tokią eilutę, kurią galima išspausdinti į ekraną:



Car klasės objektų raktai

Klasės Car metodo toString() rezultatas

```
TA101=Renault_Laguna:1997 50000 1700.0
TA102=Renault_Megane:2001 20000 3500.0
TA103=Toyota_Corolla:2001 20000 3500.0
TA104=Renault_Laguna:2001 115900 700.0
TA105=Renault_Megane:1946 365100 9500.0
```

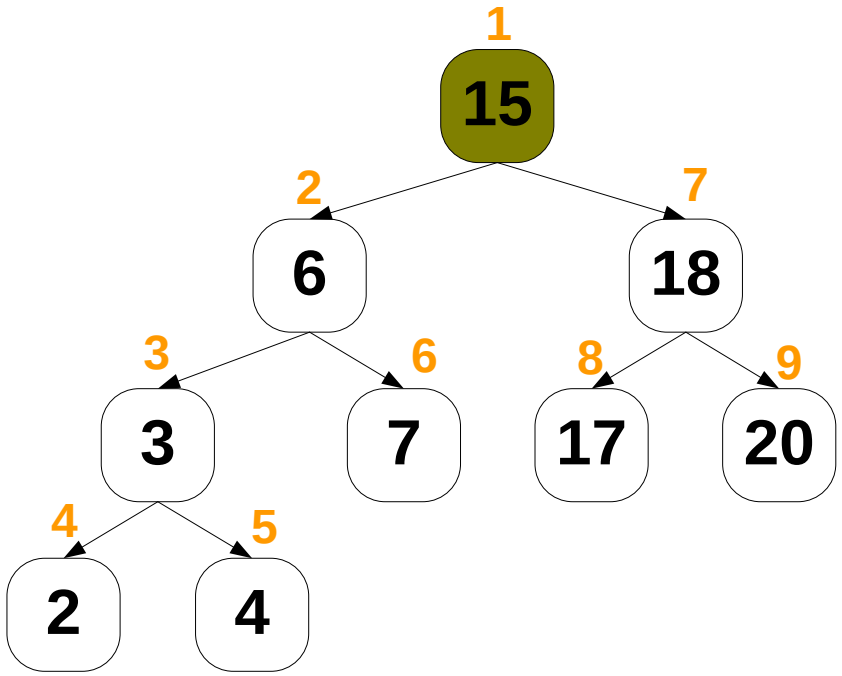
Klasės BstSet metodo toString() rezultatas

Tiesioginis (preorder) medžio apėjimas

ktu

- Tiesioginiu medžio apėjimu pirmiausia aplankoma viršūnė, po to kairysis ir dešinysis pomedžiai.
- Šiuo apėjimu gautas viršūnes **ta pačia tvarka** talpinant kitame medyje, gaunamas identiškas medis pirmajam, todėl tinka panaudojimui ***clone()*** metode.

```
Algoritmas:
FUNCTION Tiesioginis_apėjimas(Viršūnė)
1.IF (Viršūnė <> Null) THEN
2.   Println(Viršūnė);
3.   Tiesioginis_apėjimas(Viršūnės kairysis pomedis);
4.   Tiesioginis_apėjimas(Viršūnės dešinysis pomedis);
5.ENDIF;
END FUNCTION.
```



Apėjimo rezultatas:

15, 6, 3, 2, 4, 7, 18, 17, 20

clone() realizacija

ktu

- Metodo realizacijos principai tokie patys kaip ir klasės ***List*** atveju, išskyrus tai, kad medžio elementai apeinami tiesioginiu medžio apėjimu.
- ***clone()*** metodas sukuria ir grąžina aibės, realizuotos DP-medžiu, kopiją.

```
@Override
public Object clone() throws CloneNotSupportedException {
   BstSet<E> cl = (BstSet<E>) super.clone();
   if (root == null) {
      return cl;
   }
   cl.root = cloneRecursive(root);
   cl.size = this.size;
   return cl;
}

private BstNode<E> cloneRecursive(BstNode<E> node) {
   if (node == null) {
      return null;
   }

   BstNode<E> clone = new BstNode<>(node.element);
   clone.left = cloneRecursive(node.left);
   clone.right = cloneRecursive(node.right);
   return clone;
}
```

Per argumentus perduodamas pomedis

Atliekamas tiesioginis medžio apėjimas

clone() demonstracija (1)

ktu

```
Car c1 = new Car("Renault", "Laguna", 1997, 50000, 1700);
Car c2 = new Car("Renault", "Megane", 2001, 20000, 3500);
Car c3 = new Car("Toyota", "Corolla", 2001, 20000, 3500);
Car c4 = new Car("Renault Laguna 2001 115900 700");
Car c5 = new Car("Renault Megane 1946 365100 9500");
Car c6 = new Car("Honda   Civic  2001  36400 80.3");

Car[] carsArray = {c3, c1, c5, c2};

Ks.oun("Auto Aibė:");
ParsableSortedSet<Car> carsSet = new ParsableBstSet<>(Car::new);
for (Car c : carsArray) {
    carsSet.add(c);
}
Ks.oun(carsSet.toVisualizedString(""));

ParsableSortedSet<Car> carsSetCopy = (ParsableSortedSet<Car>) carsSet.clone();
carsSetCopy.add(c4);
carsSetCopy.add(c6);
Ks.oun("Papildyta autoaibės kopija:");
Ks.oun(carsSetCopy.toVisualizedString(""));
```

Suformuojamas Car klasės objektų masyvas

Sukuriamas Car klasės objektų aibės objektas

Papildomas aibės objektas

Grafinis medžio išvedimas

Sukuriama automobilių aibės objekto kopija

Automobilių aibės objekto kopija papildoma dvejais Automobilis klasės objektais

clone() demonstracija (2)

ktu

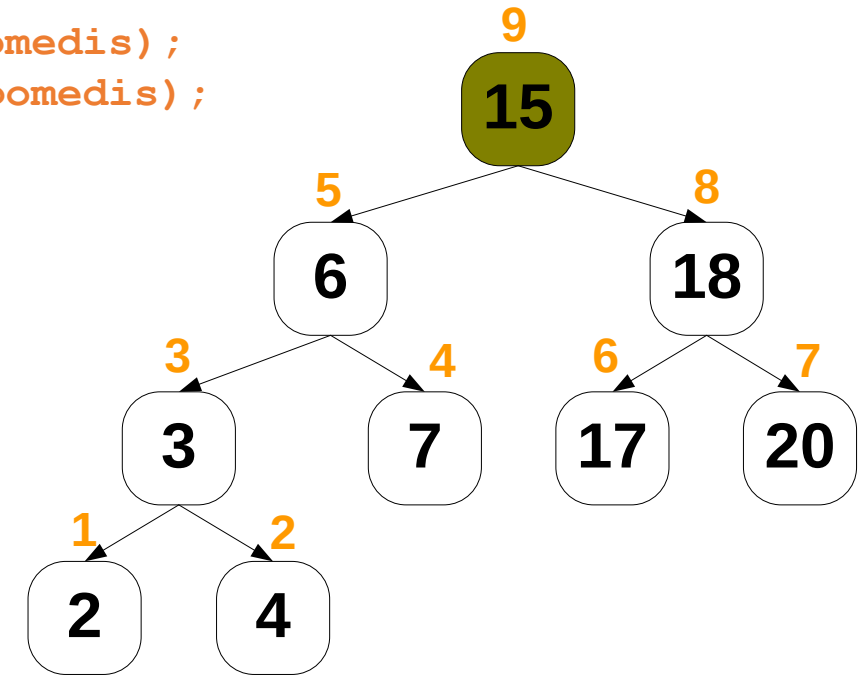
```
1| autoAibė:
2|
      ●TA105=Renault_Megane:1946 365100 9500.0
>     ●TA103=Toyota_Corolla:2001 20000 3500.0
        ●TA102=Renault_Megane:2001 20000 3500.0
        ●TA101=Renault_Laguna:1997 50000 1700.0

3| Papildyta autoaibės kopija:
4|
        ●TA106=Honda_Civic:2001 36400 80.3
      ●TA105=Renault_Megane:1946 365100 9500.0
        ●TA104=Renault_Laguna:2001 115900 700.0
>     ●TA103=Toyota_Corolla:2001 20000 3500.0
        ●TA102=Renault_Megane:2001 20000 3500.0
      ●TA101=Renault_Laguna:1997 50000 1700.0
```

Atvirkštinis (postorder) medžio apėjimas

ktu

- Atvirkštiniu medžio apėjimu pirmiausia apeinamas kairysis, po to dešinysis pomedžiai, ir viršūnė.

```
Algoritmas:
FUNCTION Atvirkštinis_apėjimas(Medis)
1.IF (Medis <> Null) THEN
2.   Atvirkštinis_apėjimas(Medžio kairysis pomedis);
3.   Atvirkštinis_apėjimas(Medžio dešinysis pomedis);
4.   Println(Medis);
5.ENDIF;
END FUNCTION.
```



Apėjimo rezultatas:
2, 4, 3, 7, 6, 17, 20, 18, 15

isEmpty(), *size()* ir *clear()* metodų realizacija

ktu

- Patikrinama ar aibė tuščia:

```
public boolean isEmpty() {
   return root == null;
}
```

- Grąžinamas aibės dydis:

```
public int size() {
   return size;
}
```

- Išvaloma aibė:

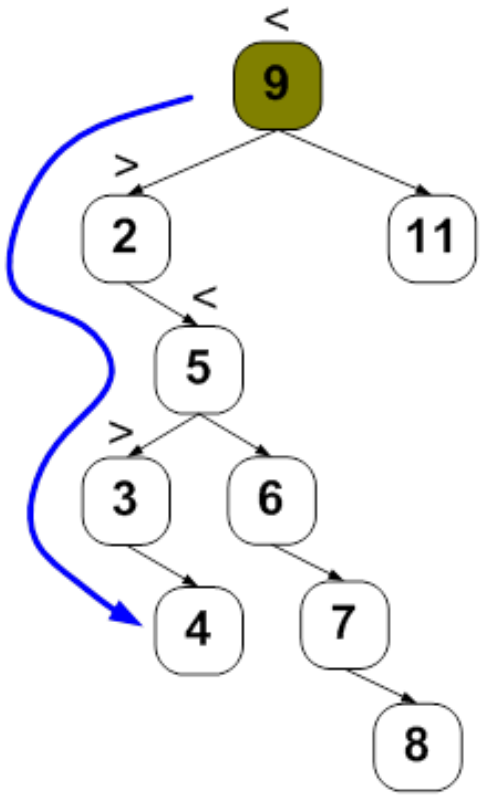
```
public void clear() {
   root = null;
   size = 0;
}
```

Paieška DP-medyje

ktu

```
Iteracinis algoritmas:
FUNCTION Paieška(Šaknis, Viršūnė)
1. Ein_Viršūnė = Šaknis;
2. WHILE (Ein_Viršūnė <> Null) THEN
3.    IF (Viršūnė < Ein_Viršūnė)
4.      Ein_Viršūnė = Ein_Viršūnės kairė
5.    ELSE IF (Viršūnė > Ein_Viršūnė)
6.      Ein_Viršūnė = Ein_Viršūnės dešinė
7.    ELSE
8.     RETURN Ein_Viršūnė;
9.    END IF
10.END WHILE
11.RETURN Null.
END FUNCTION.
```

Ieškome 4:



Paieška DP-medyje (modifikuota)

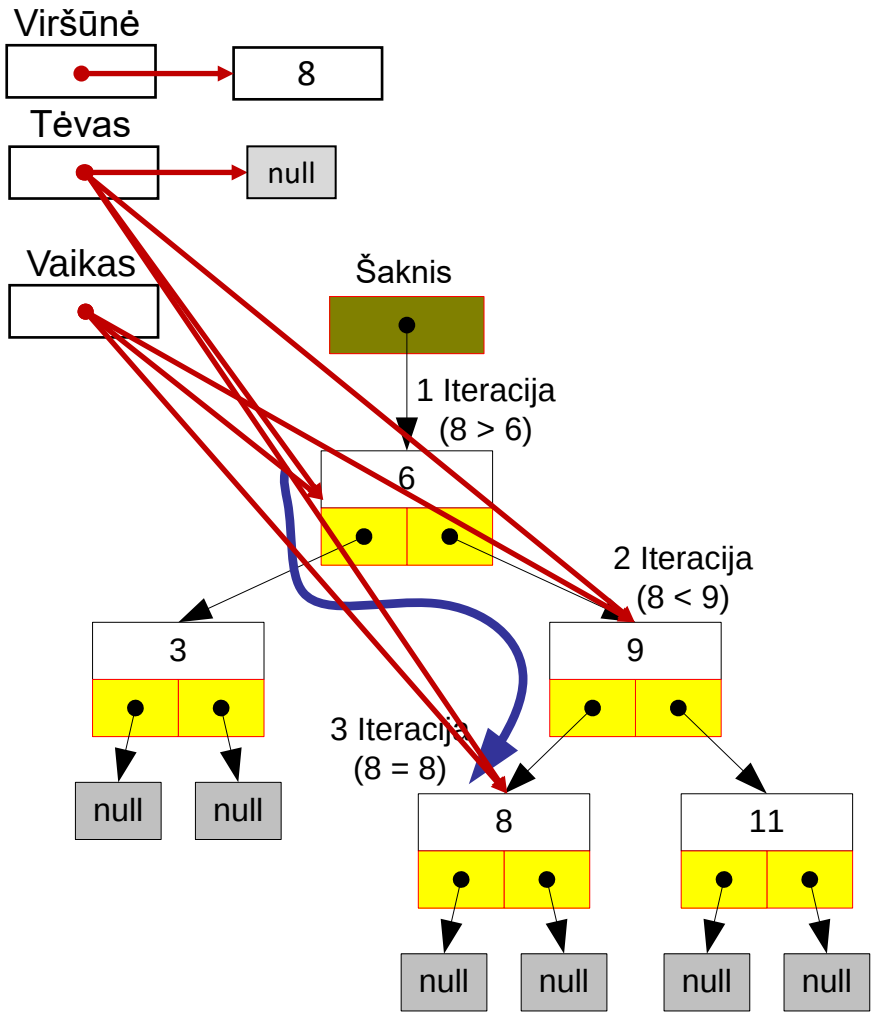
ktu

- Taikydami DP-medžio paieškos algoritmą aibei taip jį modifikuosime, kad neradus elemento, vietoj null būtų grąžinamas paskutinis paieškos metu aplankytas elementas:

Ieškoma 8:

```
FUNCTION Paieška(Šaknis, Viršūnė)
1. Vaikas = Šaknis;
2. Tėvas = Null;
3. WHILE (Vaikas <> Null) THEN
4.    Tėvas = Vaikas;
5.    IF (Viršūnė < Vaikas)
6.      Vaikas = Vaiko kairė
7.    ELSE IF (Viršūnė > Vaikas)
8.      Vaikas = Vaiko dešinė
9.    ELSE
10.     RETURN Vaikas;
11.   END IF
12.END WHILE
13.RETURN Tėvas.
END FUNCTION.
```

Grąžinama 8.



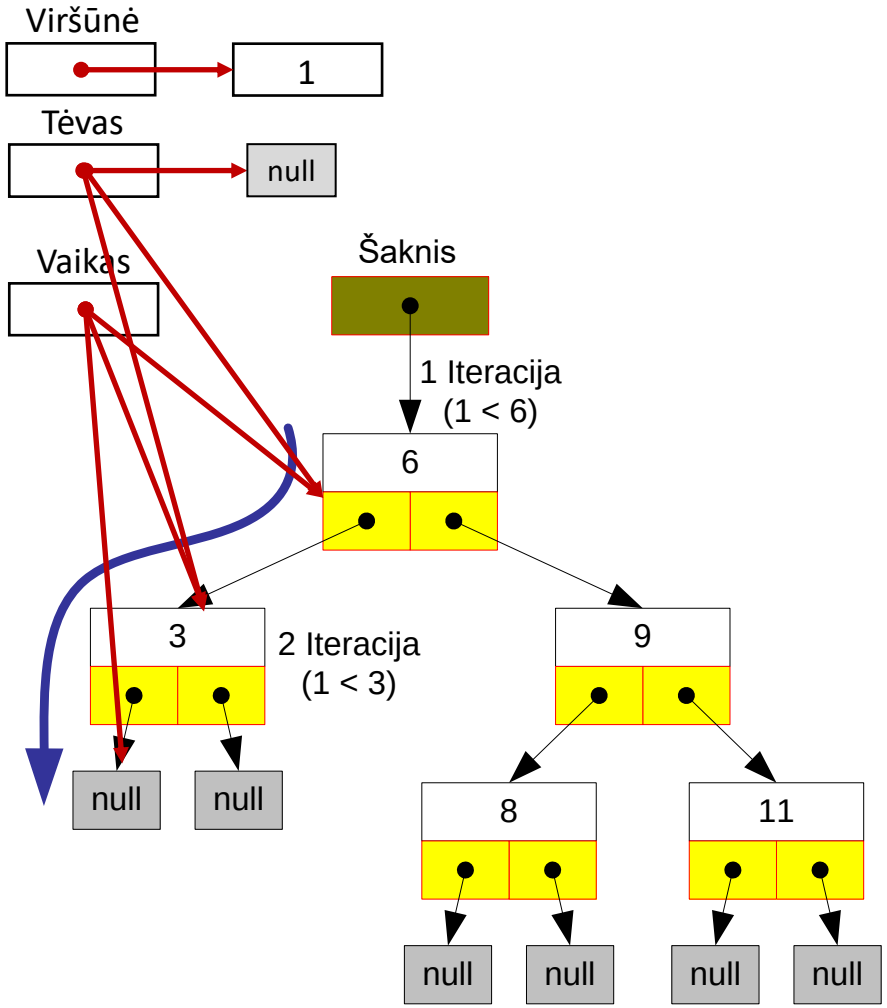
Paieška DP-medyje

ktu

Ieškoma 1:

```
FUNCTION Paieška(Šaknis, Viršūnė)
1. Vaikas = Šaknis;
2. Tėvas = Null;
3. WHILE (Vaikas <> Null) THEN
4.    Tėvas = Vaikas;
5.    IF (Viršūnė < Vaikas)
6.      Vaikas = Vaiko kairė
7.    ELSE IF (Viršūnė > Vaikas)
8.      Vaikas = Vaiko dešinė
9.    ELSE
10.     RETURN Vaikas;
11.   END IF
12.END WHILE
13.RETURN Tėvas;
END FUNCTION.
```

Grąžinama 3.



get realizacija

ktu

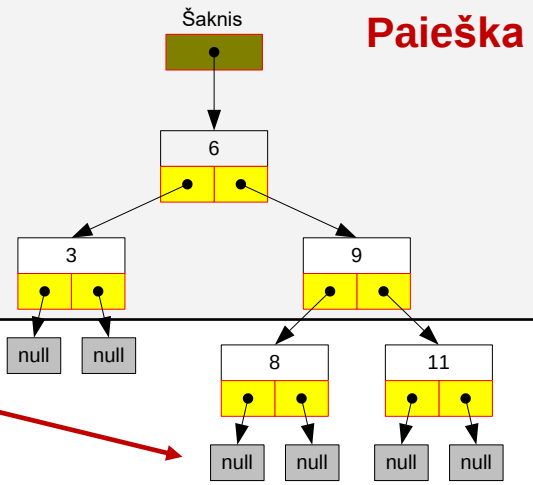
- Iteracinio paieškos DP-medyje algoritmo realizacija:

```
private E get(E element) {
   if (element == null) {
      throw new IllegalArgumentException("Element is null in get(E element)");
   }

   BstNode<E> node = root;
   while (node != null) {
      int cmp = c.compare(element, node.element);

      if (cmp < 0) {
         node = node.left;
      } else if (cmp > 0) {
         node = node.right;
      } else {
         return node.element;
      }
   }

   return null;
}
```

Palyginimas

c - Komparatorius

Jei nieko nerasta, gražinama null



Paieška

Vykdymo laikas.
Proporcingas ieškomo elemento mazgo gyliui medyje.

contains realizacija

ktu

- Metodu patikrinama ar elementas priklauso aibei. Metodo realizacijoje atliekama elemento paieška. Jei paieškos rezultatas = null, gražinama false, kitu atveju – true:

```
public boolean contains(E element) {
    if (element == null) {
        throw new IllegalArgumentException("Element is null in contains(E element)");
    }

    return get(element) != null;
}
```

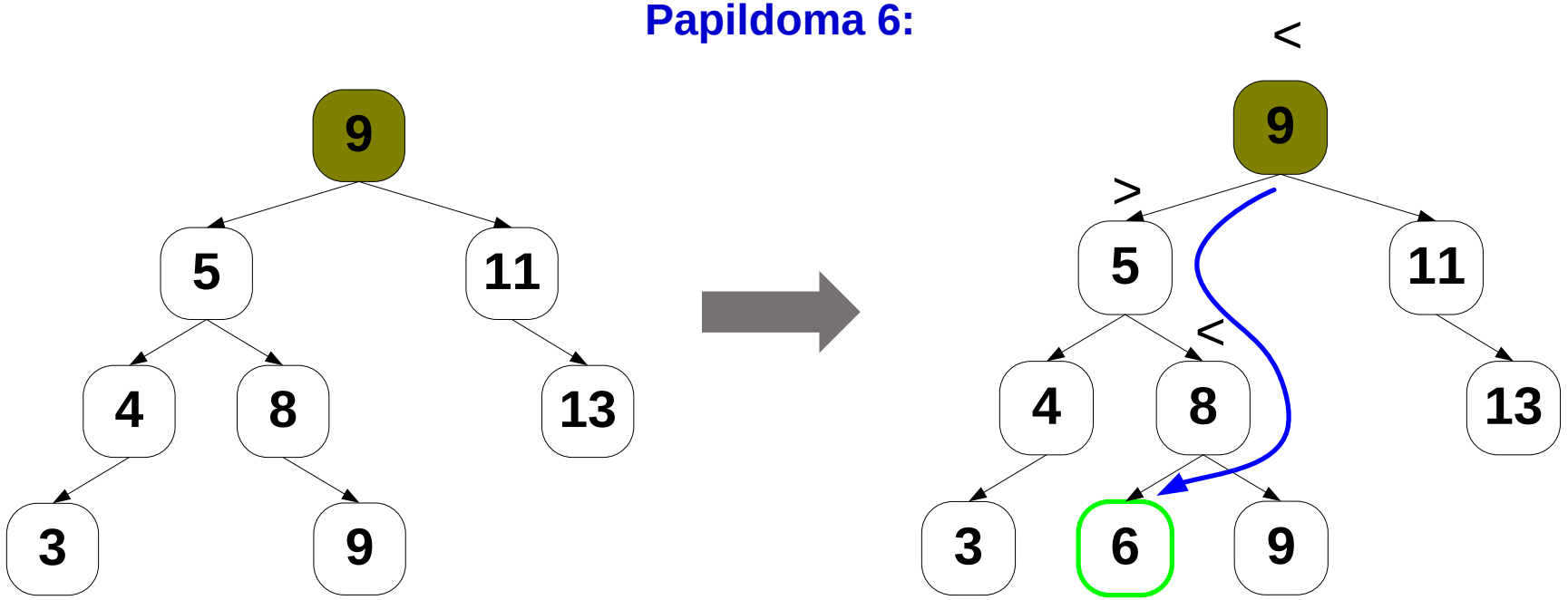
get - Elemento paieška

DP-medžio papildymas (1)

ktu

- DP-medyje viršūnė papildoma **tik** kaip lapas.
- Atliekama viršūnės paieška.
- Jei randama tokia pati viršūnė, atnaujinami su raktu susieti duomenys (raktas išlieka tas pats). Kitu atveju - viršūnė prijungiama prie paieškos kelio paskutinės viršūnės.

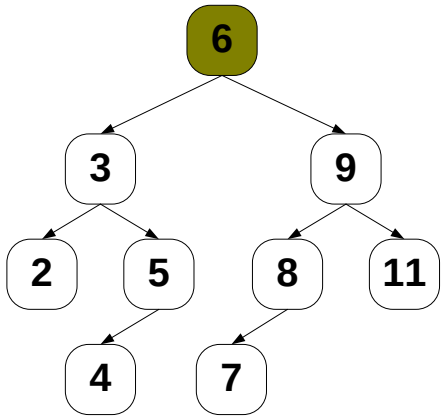
Papildoma 6:



DP-medžio papildymas (2)

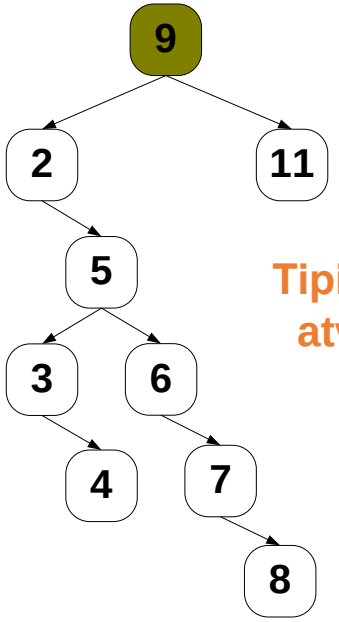
ktu

Tvarka: 6 9 3 2 11 5 8 4 7



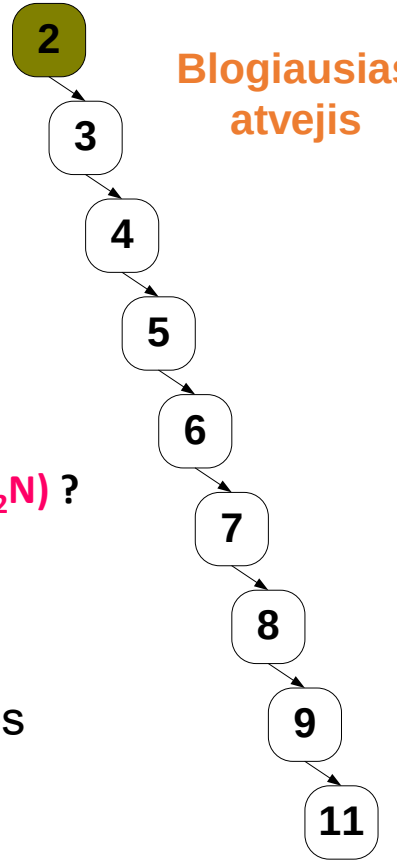
Geriausias atvejis

Tvarka: 9 2 11 5 3 6 4 7 8



Tipiškas atvejis

Tvarka: 2 3 4 5 6 7 8 9 11



Blogiausias atvejis

Kaip su **O($\log_2$N)** ?

Vidutinis viršūnės gylis – **$\log_2$N.**
Maksimalus viršūnės gylis – **(N-1).** Blogiausiu atveju (jei viršūnės prieš įterpimą yra surikiuotos), DP-medis išsigimsta į tiesinį dinamiškąjį sąrašą.

add rekursinė realizacija

ktu

- Metodu aibė papildoma nauju elementu:

```
@Override
public void add(E element) {
    if (element == null) {
        throw new IllegalArgumentException("Element is null in add(E element)");
    }

    root = addRecursive(element, root);
}
```

Elemento papildymui panaudojamas rekursinis metodas

- **Vykdymo laikas.** Proporcingas papildomo elemento mazgo gyliui medyje.

addRecursive realizacija

ktu

```
private BstNode<E> addRecursive(E element, BstNode<E> node) {
   if (node == null) {
      size++;
      return new BstNode<>(element);
   }

   int cmp = c.compare(element, node.element);

   if (cmp < 0) {
      node.left = addRecursive(element, node.left);
   } else if (cmp > 0) {
      node.right = addRecursive(element, node.right);
   }

   return node;
}
```

Papildymas

Palyginimas

Paieška

- Paieškos metu rekursiškai keliaujame į mazgo n kairįjį arba dešinįjį pomedį, kol reikiamas n pomedis = null. Tada prie n prijungiame elemento mazgą.

add iteracinė realizacija

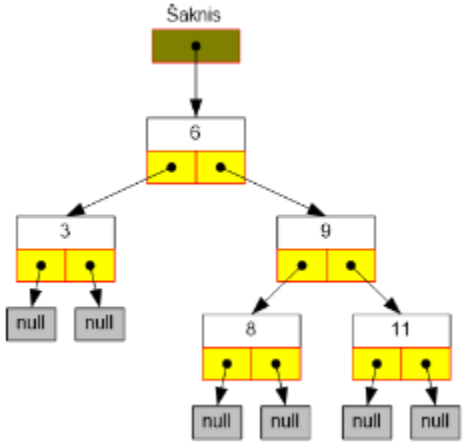
```
public void add(E element) {
   if (element == null) {
      throw new IllegalArgumentException("Element is null in add(E element)");
   }
   if (root == null) {
      root = new BstNode<E>(element);
   } else {
      BstNode<E> current = root;
      BstNode<E> parent = null;
      while (current != null) {
         parent = current;
         int cmp = c.compare(element, current.element);
         if (cmp < 0) {
            current = current.left;
         } else if (cmp > 0) {
            current = current.right;
         } else {
            return;
         }
      }
      int cmp = c.compare(element, parent.element);
      if (cmp < 0) {
         parent.left = new BstNode<E>(element);
      } else if (cmp > 0) {
         parent.right = new BstNode<E>(element);
      }
   }
   size++;
}
```

Jei medis tuščias

Mazgo, prie kurio bus jungiamas elementas, paieška

Kodėl?

Medžio papildymas nauju elementu



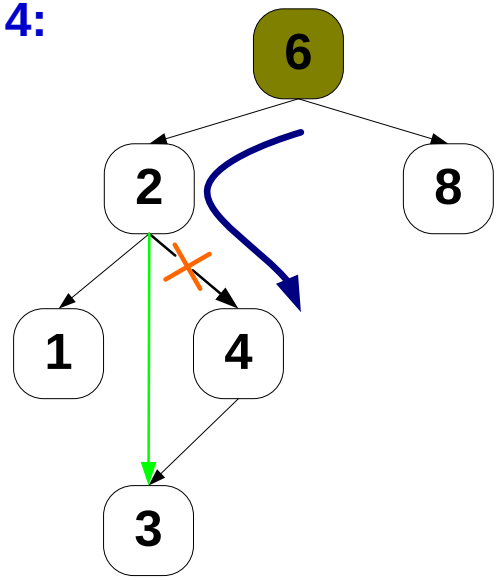
Šalinimas DP-medyje (1)

ktu

- Šalinimo algoritmas pats sudėtingiausias. Iš pradžių atliekama viršūnės paieška pagal raktą, po to ji pašalinama. Pašalinus viršūnę, reikia pasirūpinti jos vaikais, išlaikant DP-medžio savybes. Viršūnės šalinimo atvejai:

1. Viršūnė yra lapas.
   Tėvo atitinkamai rodyklei priskiriame *null*.
2. Viršūnė turi vieną vaiką.
   Viršūnės (4) tėvo (2) rodyklę į vaiką pakeičiama rodykle į viršūnės (4) vaiką (3).

Šalinama 4:



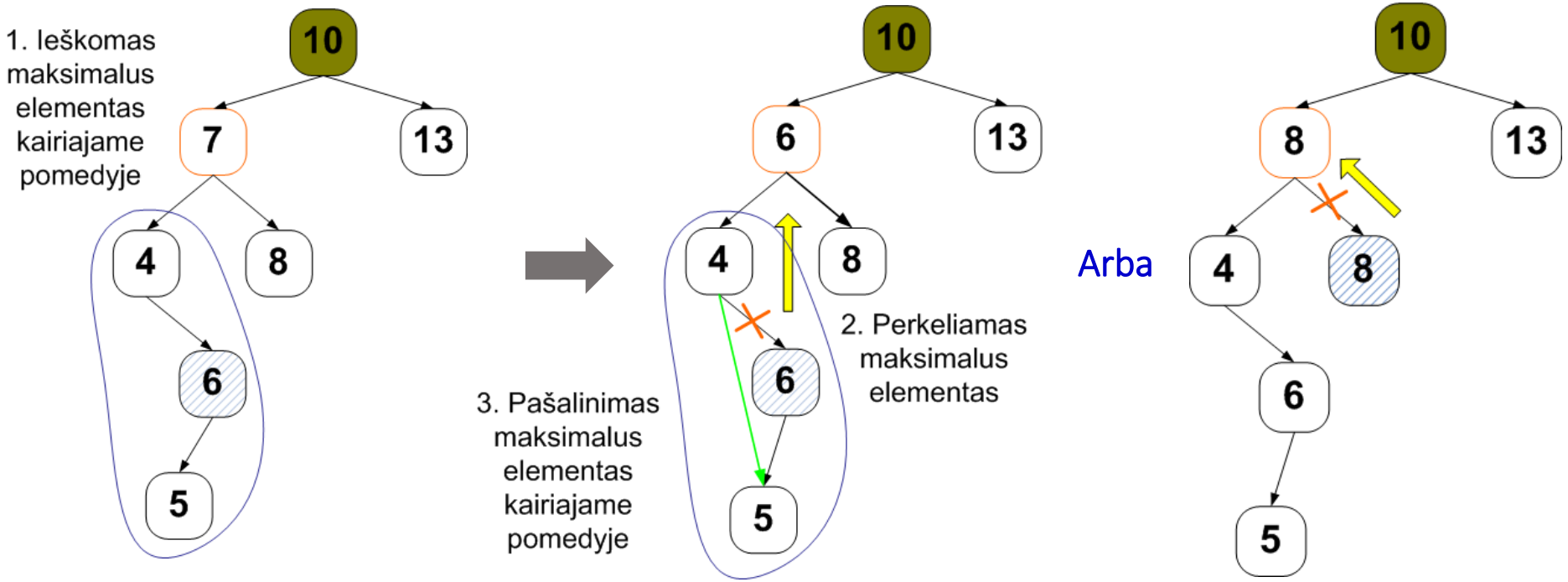
Šalinimas DP-medyje (2)

ktu

3. Viršūnė turi du vaikus.

Viršūnė pakeičiama maksimalaus rakto viršūne iš kairiojo pomedžio (*arba minimalia iš dešiniojo*).
Maksimalaus rakto viršūnė kol kas paliekama.
Po to maksimalaus (*arba minimalaus*) rakto viršūnė visada neturi vaikų arba turi tik vieną kairįjį (*dešinįjį*) vaiką, todėl inicijuojamas 1 arba 2 šalinimo atvejai.

Šalinama 7:



remove realizacija

ktu

- Metodu pašalinamas aibės elementas:

```
@Override
public void remove(E element) {
   if (element == null) {
      throw new IllegalArgumentException("Element is null i remove(E element)");
   }

   root = removeRecursive(element, root);
}
```

Elemento šalinimui panaudojamas rekursinis metodas

- **Vykdymo laikas.** Proporcingas šalinamo elemento mazgo gyliui medyje.

removeRecursive realizacija

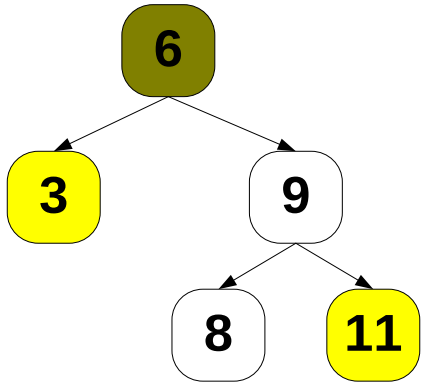
ktu

Tai Lab2 užduotis

removeMax realizacija

ktu

- Pašalinamas didžiausio rakto elementas pomedyje, paiešką pradedant mazgu „node“. Grąžiną tėvą.
- Mažiausio rakto viršūnė DP-medyje paprastai visada yra „kairiausia“, didžiausio – „dešiniausia“.
- Galima atvirkštinė realizacija.

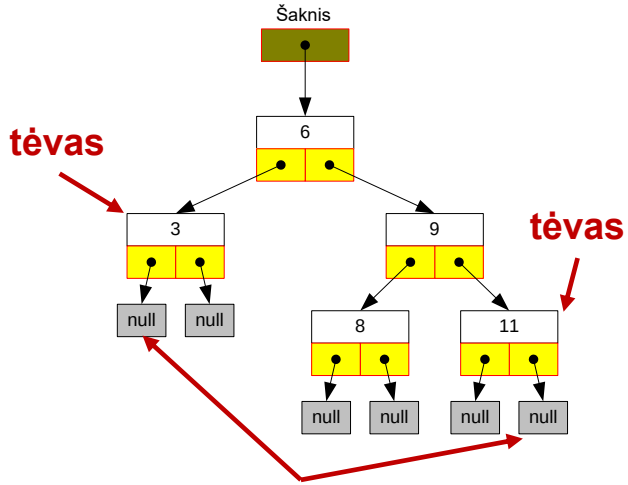



```
BstNode<E> removeMax(BstNode<E> node) {
    if (node == null) {
        return null;
    } else if (node.right != null) {
        node.right = removeMax(node.right);
        return node;
    } else {
        return node.left;
    }
}
```

getMax realizacija

ktu

- Grąžina maksimalaus rakto elementą paiešką pradedant mazgu „node“.
- Galima ir getMin realizacija.




```
BstNode<E> getMax(BstNode<E> node) {
   BstNode<E> parent = null;
   while (node != null) {
      parent = node;
      node = node.right;
   }
   return parent;
}
```

Su kiekviena iteracija atsimenamas tėvas

toArray realizacija

ktu

```
@Override
public Object[] toArray() {
    int i = 0;
    Object[] array = new Object[size];
    for (Object o : this) {
        array[i++] = o;
    }
    return array;
}
```

Aibės realizacijos DP-medžiu iteratorius (1)

ktu

- Aibės realizacijos DP-medžiu iteratorių realizuosime vidine klase ***IteratorBst***, įdiegiančią interfeisą ***Iterator:***

```
private class IteratorBst implements Iterator<E> {

   private Stack<BstNode<E>> stack = new Stack<>();
   // Nurodo iteravimo kolekcija kryptį, true - didėjimo tvarka,
   // false - mažėjimo
   private boolean ascending;
   // Nurodo einamojo medžio elemento tėvą. Reikalingas šalinimui.
   private BstNode<E> parent = root;

   IteratorBst(boolean ascendingOrder) {
      this.ascending = ascendingOrder;
      this.toStack(root);
   }
```

← **Klasės konstruktorius**

- Sukūrus iteratoriaus objektą, konstruktoriuje metodu ***toStack()*** surandamas pomedžio minimalus elementas, o visi paieškos kelyje esantys elementai išsaugomi steke.

Aibės realizacijos DP-medžiu iteratorius (2)

ktu

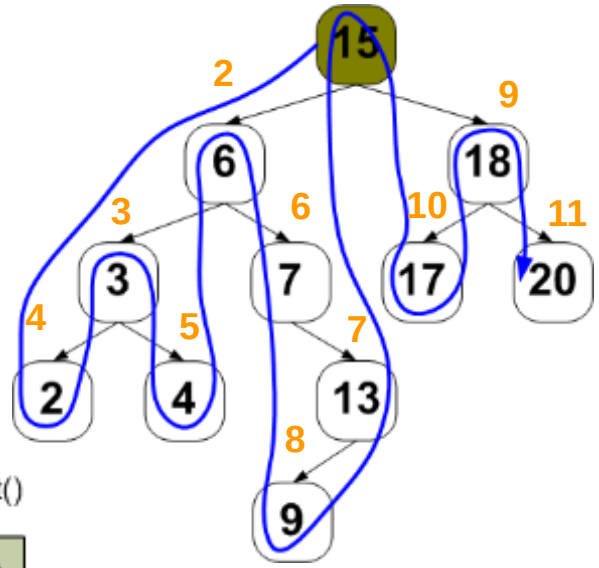
- Metodu ***toStack()*** medyje nuo duoto mazgo keliaujama į kairę, visus kelyje sutiktus elementus saugojant steke:

```
private void toStack(BstNode<E> n) {
   while (n != null) {
      stack.push(n);
      n = ascending ? n.left : n.right;
   }
}
```

Aibės realizacijos DP-medžiu iteratorius (3) – next()

ktu

- Pateiksime aibės iteratoriaus realizacijos pavyzdį, kuriuo elementai su ***next()*** grąžinami surikiuoti, todėl iteravimui pasirenkame vidinį medžio apėjimą.
- ***next()*** operacijos metu visada grąžinamas paskutinis į steką patalpintas elementas.
- Ištraukus elementą iš steko, metodu ***toStack()*** visada ieškoma minimalaus elemento dešiniajame pomedyje, o visi paieškos kelyje esantys elementai taip pat išsaugomi steke.



Sukūrus iteratorių	next()	next()	next()	next()	next()	next()	next()	next()	next()	next()	next()
Rezultatas	-	2	3	4	6	7	9	13	15	17	18	20

Stekas

2											
3	3	4			9						
6	6	6	6	7	13	13		17			
15	15	15	15	15	15	15	15	18	18	20	

Aibės realizacijos DP-medžiu iteratorius (4)

ktu

```
@Override
public boolean hasNext() {
   return !stack.empty();
}

@Override
public E next() {// Jei stekas tuščias
   if (stack.empty()) {
      lastInStack = root;
      last = null;
      return null;
   } else {
      // Grąžinamas paskutinis į steką patalpintas elementas
      BstNode<E> n = stack.pop();
      // Atsimenamas paskutinis grąžintas elementas, o taip pat paskutinis steke esantis elementas.
      // Reikia remove() metodui
      lastInStack = stack.isEmpty() ? root : stack.peek();
      last = n;
      BstNode<E> node = ascending ? n.right : n.left;
      // Dešiniajame n pomedyje ieškoma minimalaus elemento,
      // o visi paieškos kelyje esantys elementai talpinami į steką
      toStack(node);
      return n.element;
   }
}
```

Aibės realizacijos DP-medžiu iteratorius (5)

ktu

- Tačiau Lab2 demo projekte, yra realizuotas ne tik tiesioginis bet atvirkštinis aibės iteratorius, kurie iškviečiami metodais ***iterator()*** ir ***descendingIterator()***.
- Atvirkštinis iteratorius grąžina aibės elementus rakto mažėjimo tvarka.

```
@Override
public Iterator<E> iterator() {
   return new IteratorBst(true);
}

@Override
public Iterator<E> descendingIterator() {
   return new IteratorBst(false);
}
```

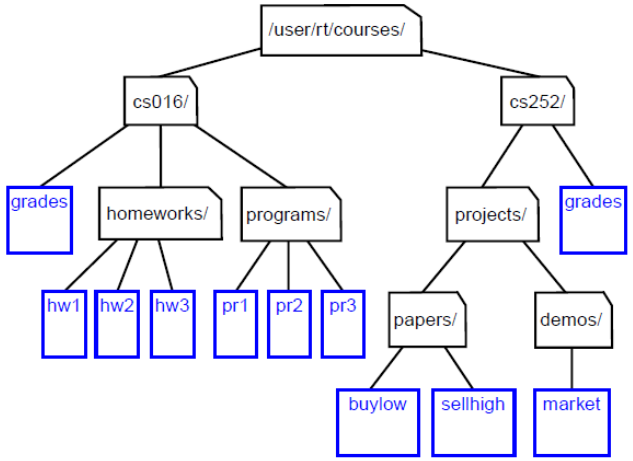
- Metodą ***remove()*** reikia realizuoti patiems, todėl jo realizacijos nepateiksime.

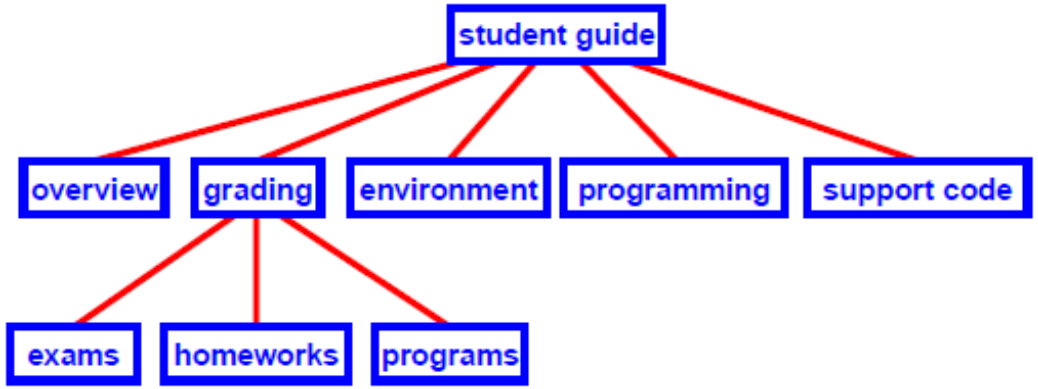
Medžių panaudojimas

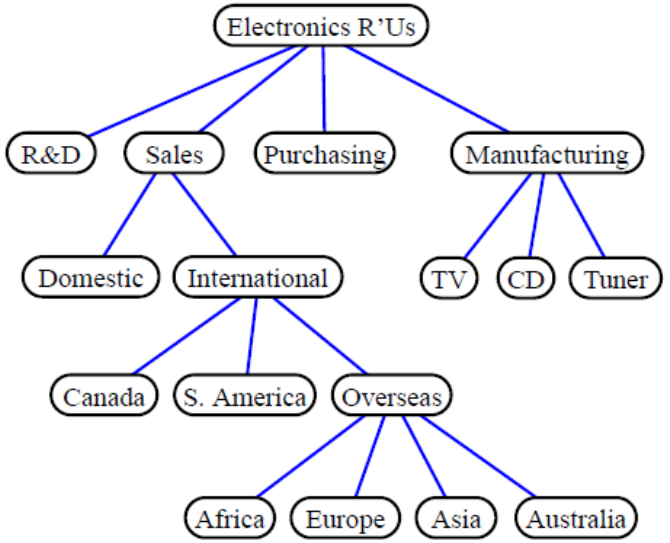
ktu

Realaus gyvenimo pavyzdžiai, kur panaudojami medžiai:

- Knygos ar dokumento turinys.
- Klasių hierarchija Javoje.
- OS failų sistema.
- Sprendimų medžiai.
- Įmonės struktūros medis.



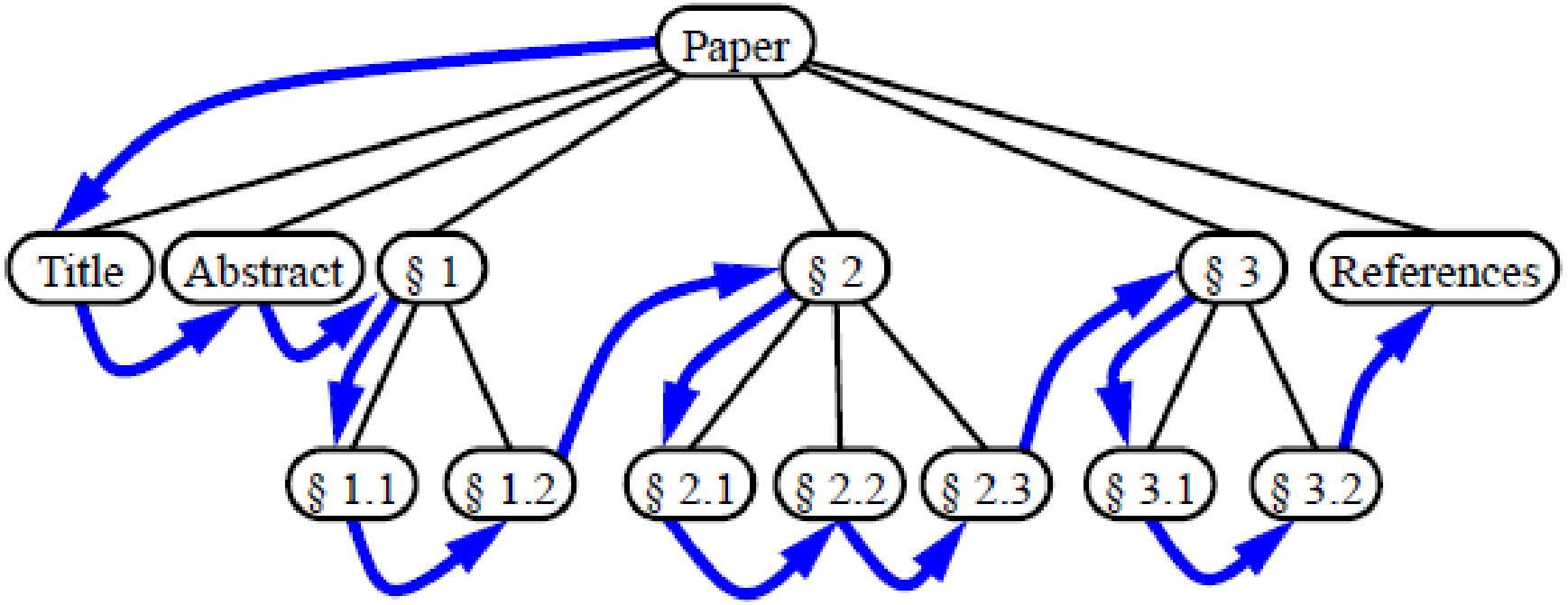




Tiesioginio medžio apėjimo pavyzdys

ktu

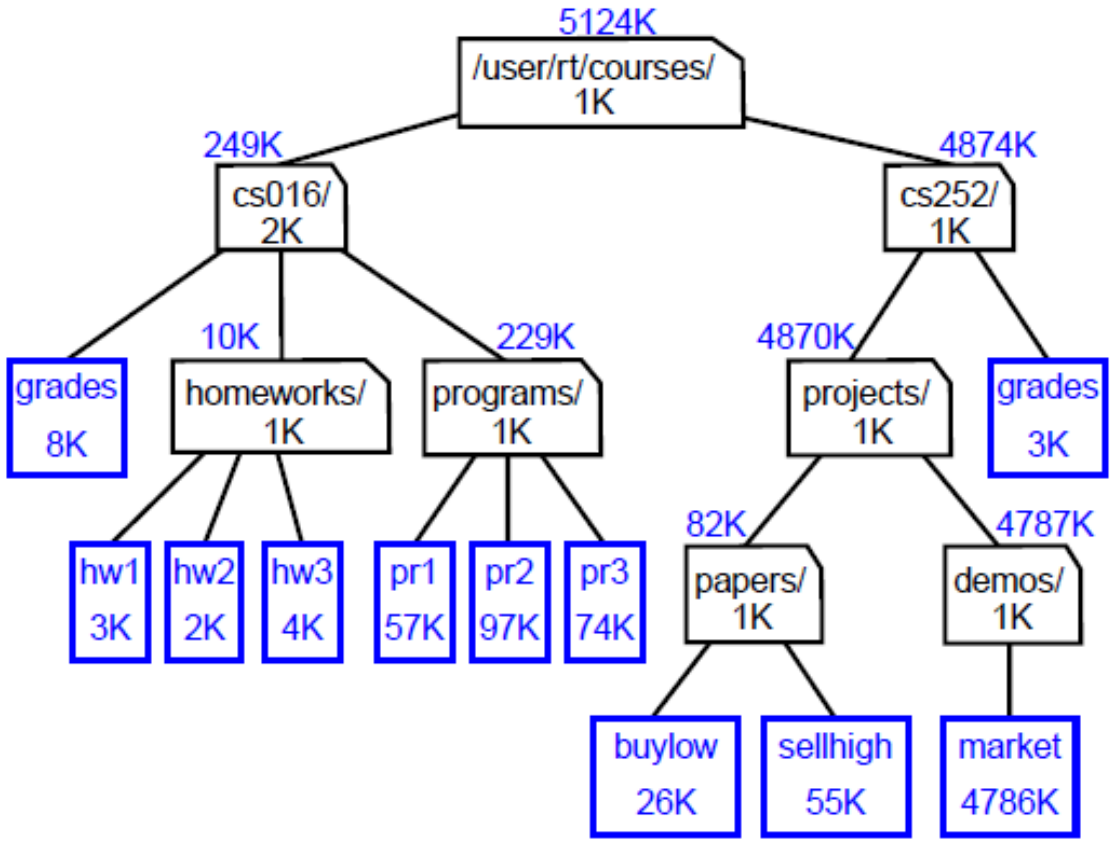
- Dokumento nuskaitymas nuo pradžios iki galo pagal paragrafus atliekamas panaudojant tiesioginį medžio apėjimą.



Atvirkštinio medžio apėjimo pavyzdys

ktu

- Vykdant unix/linux komandą du (disk usage), kuri suskaičiuoja kiek vietos diske užima direktorijos ir failai, naudojamas atvirkštinis medžio apėjimas. Einant per medį sumuojama failų užimama vieta.



Aritmetinės išraiškos

ktu

Tarkime, kad turime aritmetinę išraišką: (a+b*c)+((d*e+f)*g). Aritmetinės išraiškos yra užrašomos trimis formomis:

infix: **(a+b*c)+((d*e+f)*g)**
prefix (prieš dėlinė): **++a*bc*+*defg**
postfix (priesaginė): **abc*+de*f+g*+**

infix – aritmetinis veiksmas užrašomas tarp operandų (t.y. konstantų arba kintamųjų),
prefix – iš pradžių rašomas aritmetinis veiksmas, po to operandas,
postfix – iš pradžių rašomas operandas, po to aritmetinis veiksmas.

prefix ir *postfix* formos labai patogios, kadangi veiksmų eiliškumą apibrėžiame nenaudodami skliaustų.

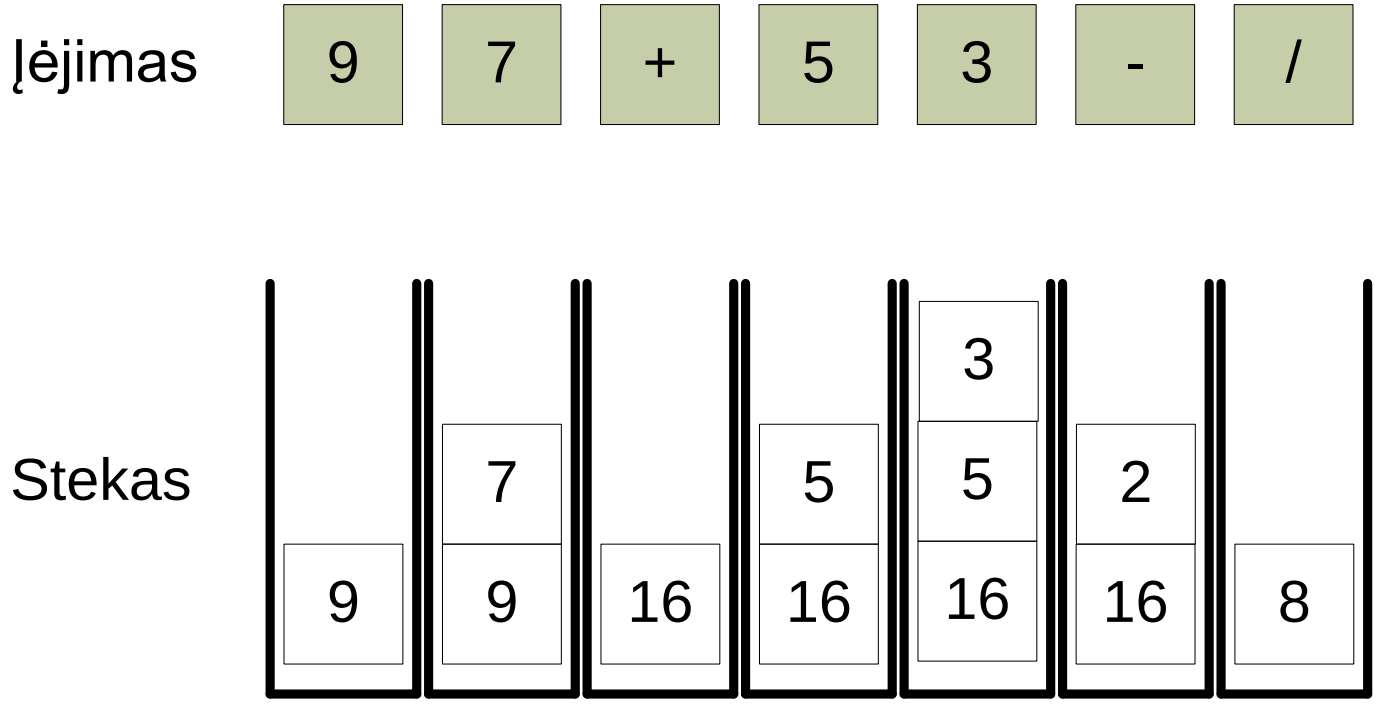
Šiuolaikiniuose kompiuteriuose naudojama *postfix* forma.

Aritmetinės išraiškos skaičiavimo pavyzdys

ktu

Aritmetinės išraiškos skaičiavimas panaudojant steką:

- *infix* išraiškos forma: (9 + 7) / (5 - 3)
- *postfix* išraiškos forma: 97+53-/

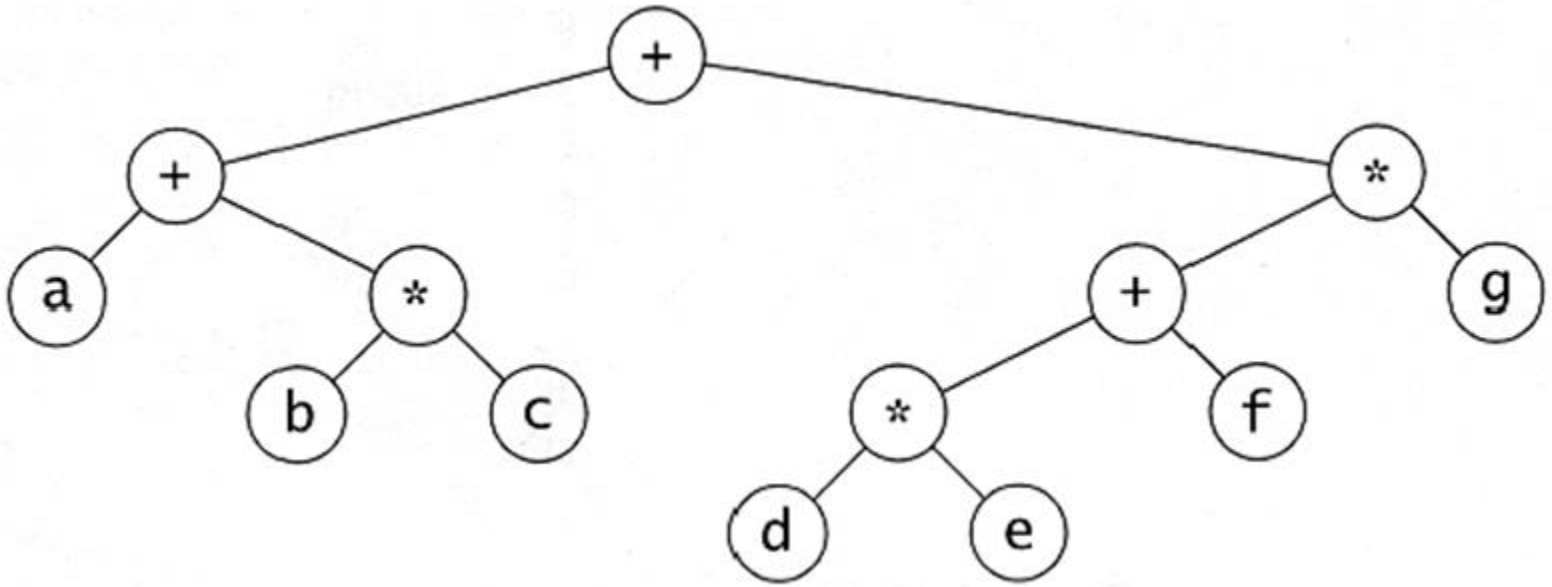


Išraiškų medžiai

ktu

- Informaciją apie aritmetinę išraišką galime saugoti medyje. Aritmetinė išraiška:

```
(a+(b*c))+((d*e+f)*g)
```



- Lapuose saugomi operandai (a,b,...).
- Vidinėse viršūnėse saugomi operatoriai (+ *).
- Medis gali būti nebūtinai dvejetainis, jei išraiškoje būtų ne tik aritmetiniai operatoriai.

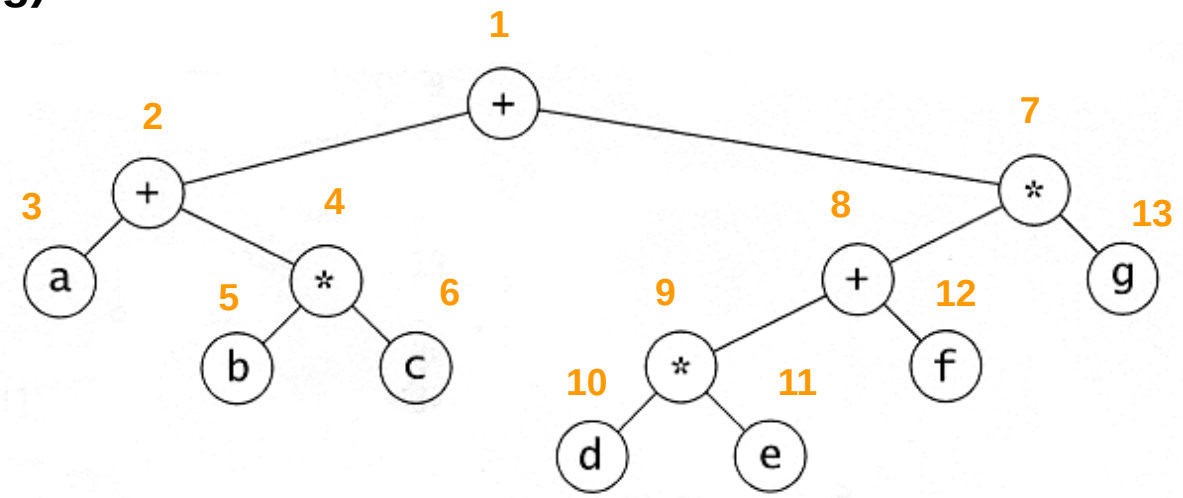
Prefix išraiškos forma

ktu

Tiesioginis medžio apėjimas atitinka prefix išraiškos formą.

infix: **(a+(b*c))+((d*e+f)*g)**

prefix: **++a*bc*+*defg**



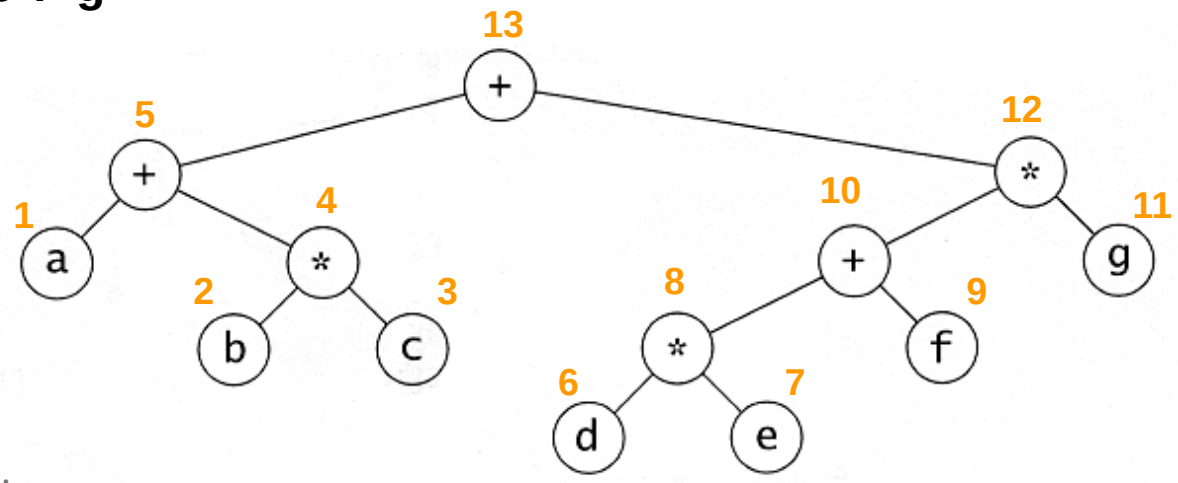

```
Algoritmas:
FUNCTION Tiesioginis_apėjimas(Viršūnė)
1.IF (Viršūnė <> Null) THEN
2.   Println(Viršūnė);
3.   Tiesioginis_apėjimas(Viršūnės kairysis pomedis);
4.   Tiesioginis_apėjimas(Viršūnės dešinysis pomedis);
5.ENDIF;
END FUNCTION.
```

Postfix išraiškos forma

ktu

Atvirkštinis medžio apėjimas atitinka postfix išraiškos formą.

infix: **(a+(b*c))+((d*e+f)*g)**
postfix: **abc*+de*f+g*+**



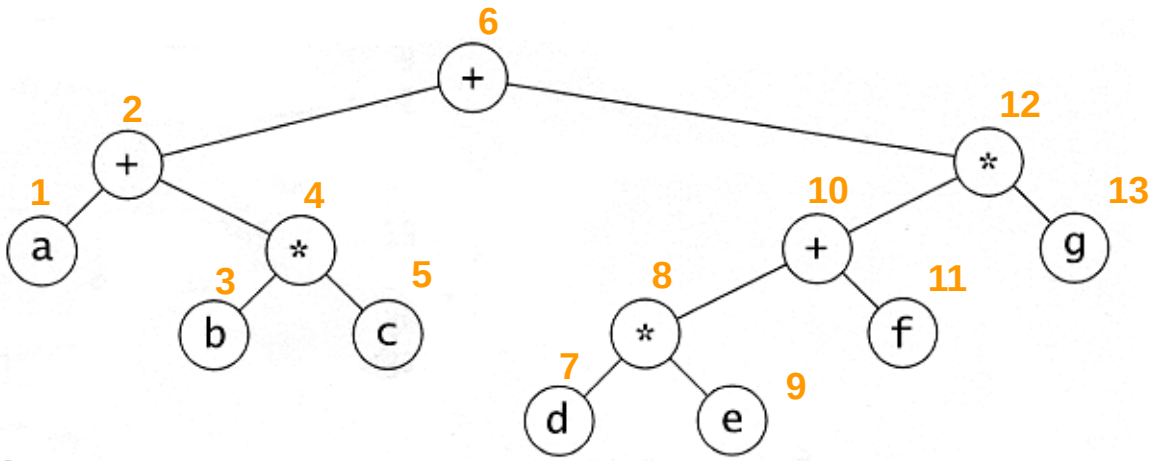

```
Algoritmas:
FUNCTION Atvirkštinis_apėjimas(Viršūnė)
1.IF (Viršūnė <> Null) THEN
2.   Atvirkštinis_apėjimas(Viršūnės kairysis pomedis);
3.   Atvirkštinis_apėjimas(Viršūnės dešinysis pomedis);
4.   Println(Viršūnė);
5.ENDIF;
END FUNCTION.
```

Infix išraiškos forma

ktu

Vidinis medžio apėjimas atitinka infix išraiškos formą.

infix: **(a+(b*c))+((d*e+f)*g)**




```
Algoritmas:
FUNCTION Vidinis_apėjimas(Viršūnė)
1.IF (Viršūnė <> Null) THEN
2.   Vidinis_apėjimas(Viršūnės kairysis pomedis);
3.   Println(Viršūnė);
4.   Vidinis_apėjimas(Viršūnės dešinysis pomedis);
5.ENDIF;
END FUNCTION.
```

Dėkoju už Jūsų dėmesį