

Dynamicky vázané metody

Pozdní vazba, virtuální metody

Motivace ...

```
class TBod
{
    protected:
        float x,y;
    public:
        int vrat_pocet_bodu() {return 1; }
};
```

- od třídy TBod odvodíme:

```
class TUsecka: public TBod
{
    protected:
        float x2,y2;
    public:
        int vrat_pocet_bodu() {return 2;}
};
```

- platí: objekt potomka je možné přiřadit do objektu předka, totéž platí i o ukazatelích

```
void main(void)
```

```
{
```

```
    TBod b1, *b;
```

```
    TUsecka u;
```

```
    b = &b1;
```

```
    b->vrat_pocet_bodu();
```

```
    b = &u;
```

```
    b->vrat_pocet_bodu();
```

```
}
```

Otázka:

Jaká se zde vyvolá metoda?

Od třídy TBod nebo TUsecka?

Odpověď:

- v obou případech se vyvolá metoda třídy TBod, protože b je ukazatel na TBod; rozhodnutí, jaká metoda se volá, se provádí *staticky* při překladu podle typu proměnné, resp. typu ukazatele
- určitě by bylo výhodné mít mechanismus, který rozhodne o volání metody *dynamicky*, za běhu programu, podle toho, na jakou instanci ukazatel právě ukazuje

- tento mechanismus objektové jazyky většinou poskytují
- nazývá se mechanismus tzv. *pozdní vazby (late binding)*; příslušné metody se označují jako *dynamicky vázané metody*
- implementace v C++ se provádí pomocí tzv. **virtuálních metod**
 - metody, které chceme vázat dynamicky, označíme klíčovým slovem **virtual**

- pravidlo:
 - „**jednou virtuální, vždy virtuální**“
 - chceme-li mít funkci virtuální, musíme ji označit jako virtuální ve všech třídách (základní i odvozených)
 - klíčové slovo **virtual** píšeme pouze při deklaraci třídy, ne při implementaci metod!
 - lze je naprogramovat jako inline, ale nemá to smysl (Proč?)

```
class TBod {  
    protected:  
        float x,y;  
    public:  
        virtual int vrat_pocet_bodu();  
};
```

```
class TUsecka: public TBod {  
    protected:  
        float x2,y2;  
    public:  
        virtual int vrat_pocet_bodu();  
};
```

```
int TBod::vrat_pocet_bodu()  
{  
    return 1;  
}
```

Při implementaci se již klíčové slovo `virtual` nepíše



```
int TUsecka:: vrat_pocet_bodu()  
{  
    return 2;  
}
```

```
void main(void)
```

```
{
```

```
    TBod b1, *b;
```

```
    TUsecka u;
```

```
    b = &b1;
```

Zde se vyvolá metoda ze třídy TBod

```
    b->vrat_pocet_bodu( );
```

```
    b = &u;
```

Zde se vyvolá metoda ze třídy TUsecka

```
    b->vrat_pocet_bodu( );
```

```
}
```

- funguje to samozřejmě i u objektů vytvořených dynamicky

```
void main(void)
{
    TBod *b;

    b = new TBod;
    b->vrat_pocet_bodu();
    delete b;
    b = new TUsecka;
    b->vrat_pocet_bodu();
    delete b;
}
```

- **POZOR:** volání virtuálních metod funguje "dynamicky" jen přes ukazatele

```
void main(void)
```

```
{
```

```
    TBod b1, b;
```

```
    TUsecka u;
```

```
    b = b1;
```

```
    b.vrat_pocet_bodu();
```

```
    b = u;
```

```
    b.vrat_pocet_bodu();
```

```
}
```

Vždy se vyvolá metoda ze třídy TBod



Jak je to implementováno uvnitř?

- adresy virtuálních metod jsou za běhu programu uloženy v *tabulce virtuálních metod* (VMT – virtual method table)
- při běhu programu se před voláním virtuální metody zjistí, na instanci které třídy ukazatel ukazuje a adresa metody se vyhledá v tabulce
- režie způsobí zpomalení běhu programu

Poznámka:

- příklad je pouze ilustrativní, ale ve skutečnosti by jej asi nikdo takto neřešil
- lepší řešení:
 - ve společném předkovi zavedeme atribut `pocet_rid_bodu`
 - jeho hodnotu nastavíme v konstruktoru každého potomka na správnou hodnotu
 - metoda `vrat_pocet_bodu()` nebude virtuální a bude implementována již v předkovi

Čisté virtuální metody

Abstraktní třídy

- velmi často se v OOP definuje jedna třída, která je obecným a společným předkem několika potomků
- některé virtuální metody v této třídě nemají žádný kód, protože jej **nemá smysl implementovat nebo se implementovat** ve společném předkovi ani nedá
 - kód se implementuje až v jednotlivých potomcích

- metody *bez kódu* ve společném předkovi se označují jako **čisté virtuální metody**; v hlavičce třídy se deklarují takto:

virtual int metoda()=0;

- třída, která má alespoň jednu čistou virtuální metodu, se nazývá **abstraktní třída**
 - **abstraktní třída nemůže mít žádnou instanci**
- v programu se proměnné od abstraktní třídy používají pouze jako **ukazatelé**; ukazují pak na instanci nějakého potomka; virtuální metody se volají dle toho, na jakou instanci ukazují

Příklad - konečný automat

- abstraktní třída TAutomat (předek)
 - čistá virtuální metoda:
 - odezva(int *vst, int *vyst, int n)
 - počítá výstupní posloupnost na vstupní posloupnost délky n
- potomci TMoore a TMealy
 - virtuální metody, které vypočítají pro každý typ automatu ze zadané vstupní posloupnosti výstupní posloupnost stejné délky
 - metodu odezva nelze napsat v předkovi, protože každý z automatů počítá posloupnost jinek

```
main ()
{
    TAutomat *a;
    int typ;
    cout << "Jaky automat (1=Moore, 2=Mealy)";
    cin >> typ;
    if (typ==1) a=new TMoore(); else a=new TMealy();
    a -> nastav_prechod(...);
    a -> nastav_vystup(...);
    a -> odezva(...);

    delete a;
}
```

Kdybychom neměli dynamicky vázané metody ...

```
main ()
{
    TAutomat *a;
    int typ;
    cout << "Jaky automat (1=Moore, 2=Mealy)";
    cin >> typ;
    if (typ==1) a=new TMoore(); else a=new TMealy();
    if (typ==1) (TMoore)a->nastav_prechod(...);
    else (TMealy)a->nastav_prechod();
    atd.

    delete a;
}
```

Bez společného předka

```
main ()
{
    TMoore *a1, TMealy *a2;
    int typ;
    cout << "Jaky automat (1=Moore, 2=Mealy)";
    cin >> typ;
    if (typ==1) a1=new TMoore(); else a2=new TMealy();
    if (typ==1) a1->nastav_prechod(...);
    else a2->nastav_prechod();
    atd.

    if (typ==1) delete a1; else delete a2;
}
```

Další výhoda (společného) předka

- pravidlo z minulé přednášky:
 - do předka lze přiřadit potomka
- uplatnění:
 - pokud je formálním parametrem nějaké metody ukazatel na předka, můžeme jako skutečný parametr uvést libovolného potomka

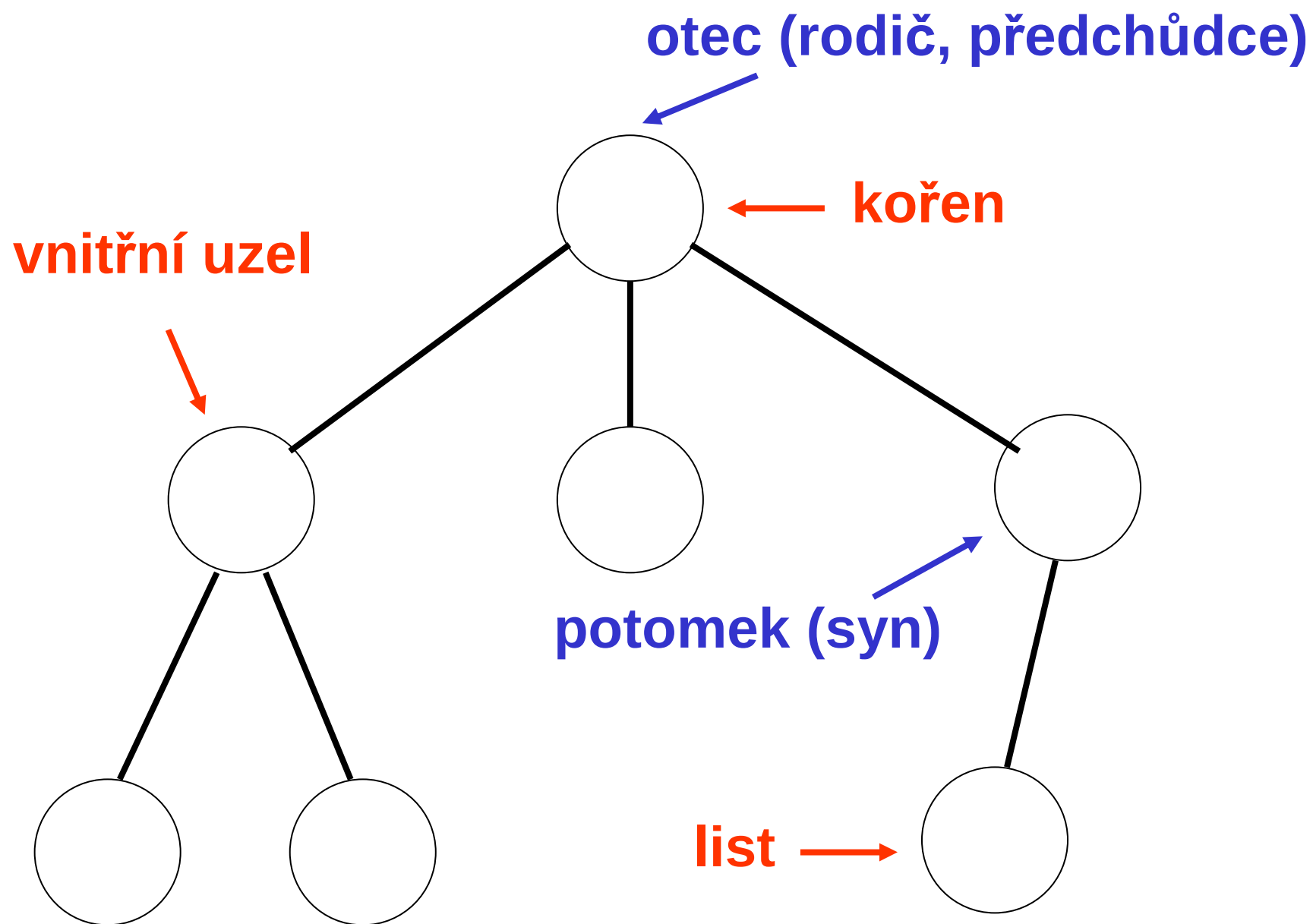
Úkol

- Implementujte abstraktní třídu TUtvar jako společného předka 2D-útvaram. Od třídy odvodte 2 potomky - třídu TKruh a TObdelnik. Naimplementujte čistou virtuální metodu obsah, která vypočítá obsah útvaru. Uchovávejte počet řídicích bodů ve společném předkovi a název útvaru. Vyzkoušejte v hlavním programu ve stylu ukázky s konečným automatem.

Stromy

Strom:

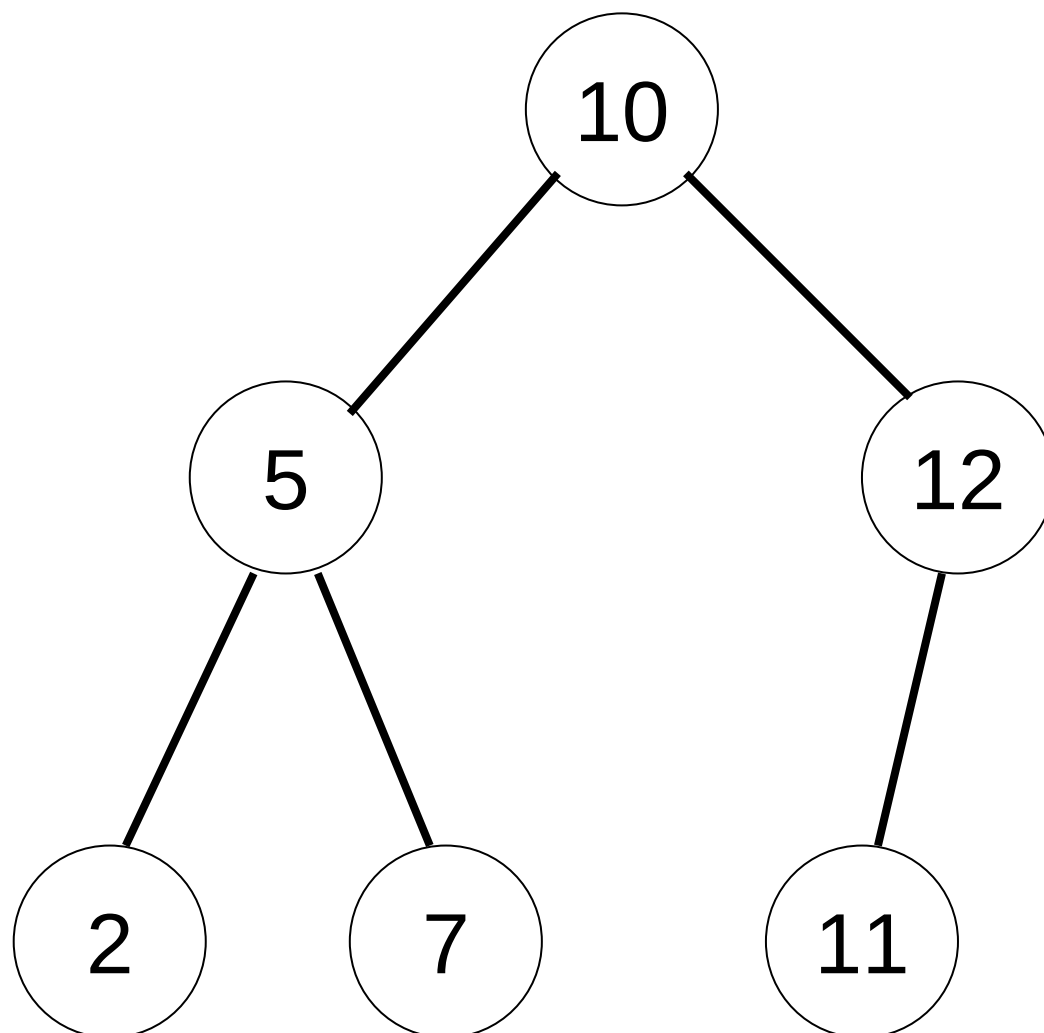
- souvislý graf bez kružnic
- využití:
 - počítačová grafika – seznam objektů
 - efektivní vyhledávání
 - výpočetní stromy (rozhodování,...)



Binární strom

- každý uzel má maximálně dva potomky
- uspořádaný binární strom
 - uzly jsou ohodnoceny prvky (čísla,...)
 - potomek na levé straně má vždy menší hodnotu nebo rovnu než rodič
 - potomek na pravé straně má vždy větší hodnotu
 - uspořádané binární stromy se využívají zejména jako **vyhledávací stromy**; složitost hledání je v průměrném případě $\log_2 n$

Uspořádaný binární strom



- binární strom je nejčastěji reprezentován ukazatelem na kořen (obecně na uzel)
- operace nad stromem jsou rekurzivní

```
class TStrom {  
    class TUzel {  
        public:  
        int hodnota;  
        TUzel *levy;  
        TUzel *pravy;  
        TUzel(int x, TUzel *l, TUzel *p);  
    };  
    TUzel *strom;  
    int najdi(TUzel *u, int x);  
    void pridej(TUzel **u, int x);  
    void zrus(TUzel *u);  
    public:  
    void vloz_prvek(int prvek);  
    int hledej(int prvek);  
    TStrom();  
    ~TStrom();  
};
```

```
TStrom::TUzel::TUzel(int x, TUzel *l,  
    TUzel *p)  
{  
    hodnota = x; levy = l; pravy = p;  
}
```

```
int TStrom::najdi(TUzel *u, int x)  
{  
    if (u==NULL) return 0;  
    if (u->hodnota==x) return 1;  
    if (x < u->hodnota)  
        return najdi(u->levy,x);  
    else  
        return najdi(u->pravy,x);  
}
```

```
int TStrom::hledej(int prvek)
{
    return najdi(strom, prvek);
}
```

```
void TStrom::pridej(TUzel **u, int x)
{
    if (*u == NULL)
        *u = new TUzel(x, NULL, NULL);
    else
        if (x <= (*u)->hodnota)
            pridej(&((*u)->levy), x);
        else pridej(&((*u)->pravy), x);
}
```

```
void TStrom::vloz_prvek(int prvek)
{
    pridej(&strom, prvek);
}
```

```
void TStrom::zrus(TUzel *u)
{
    if (u==NULL) return;
    zrus(u->l); zrus(u->p);
    delete u;
}
```

```
TStrom::TStrom()  
{  
    strom = NULL;  
}
```

```
TStrom::~~TStrom()  
{  
    zrus(strom);  
}
```

Metodu přidej lze napsat i lépe

```
TStrom::TUzel* TStrom::pridej(TUzel *u, int x)
{
    if (u == NULL)
        return new TUzel(x, NULL, NULL);
    else
        if (x <= u->hodnota)
            u->levy = pridej(u->levy, x);
        else
            u->pravy = pridej(u->pravy, x);
    return u;
}
```

Nová metoda vložení prvku do stromu

```
void TStrom::vloz_prvek(int prvek)
{
    strom = pridej(strom,prvek);
}
```

```
void main(void)
{
    TStrom s;
    s.vloz_prvek(10);
    s.vloz_prvek(5);
    s.vloz_prvek(7);
    s.vloz_prvek(2);
    s.vloz_prvek(12);
    s.vloz_prvek(11);

    s.hledej(5);
}
```

Jaký vznikne strom nyní?

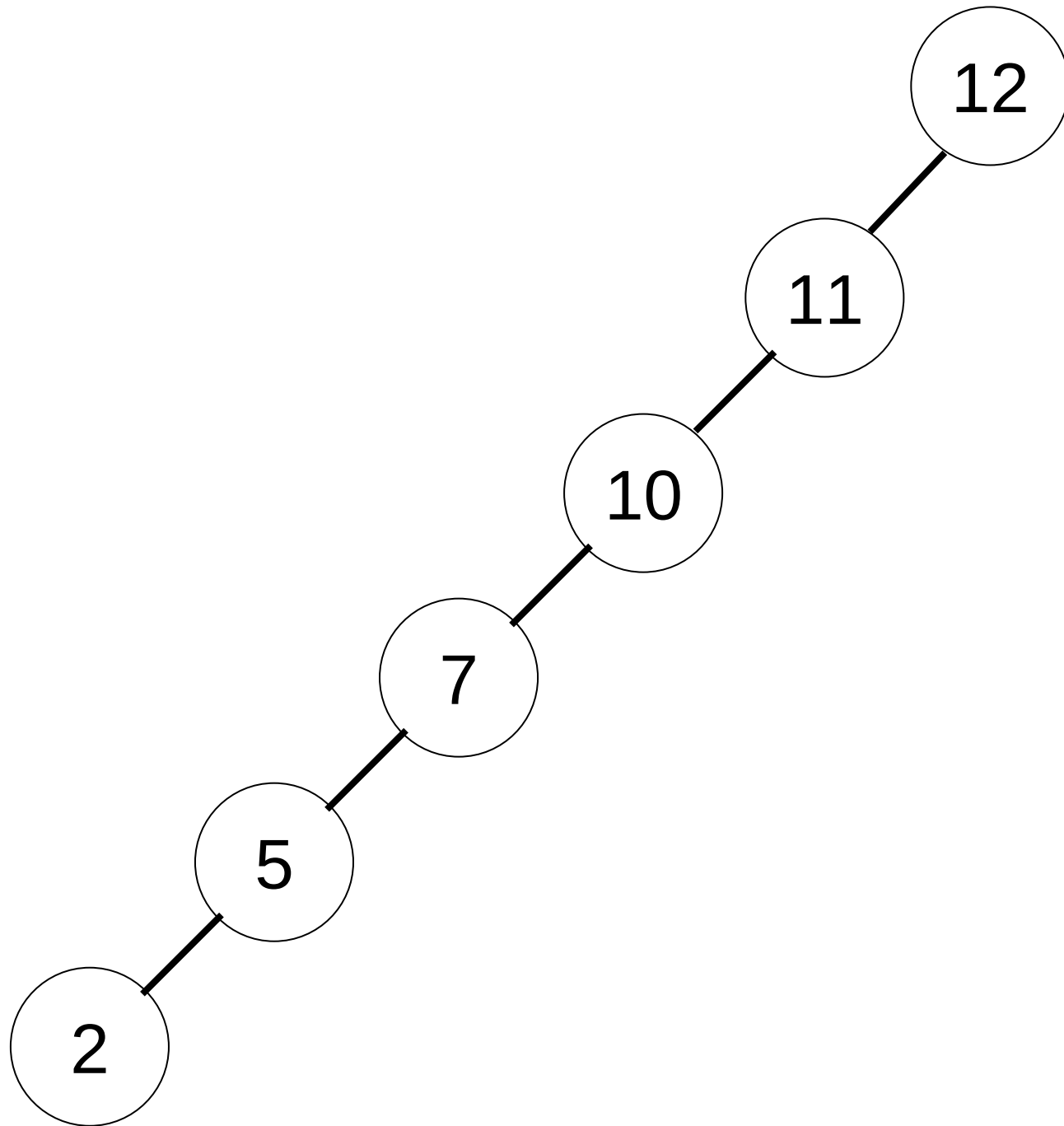
```
void main(void)
{
    TStrom s;
    s.vloz_prvek(10);
    s.vloz_prvek(5);
    s.vloz_prvek(7);
    s.vloz_prvek(2);
    s.vloz_prvek(11);
    s.vloz_prvek(12);

}
```

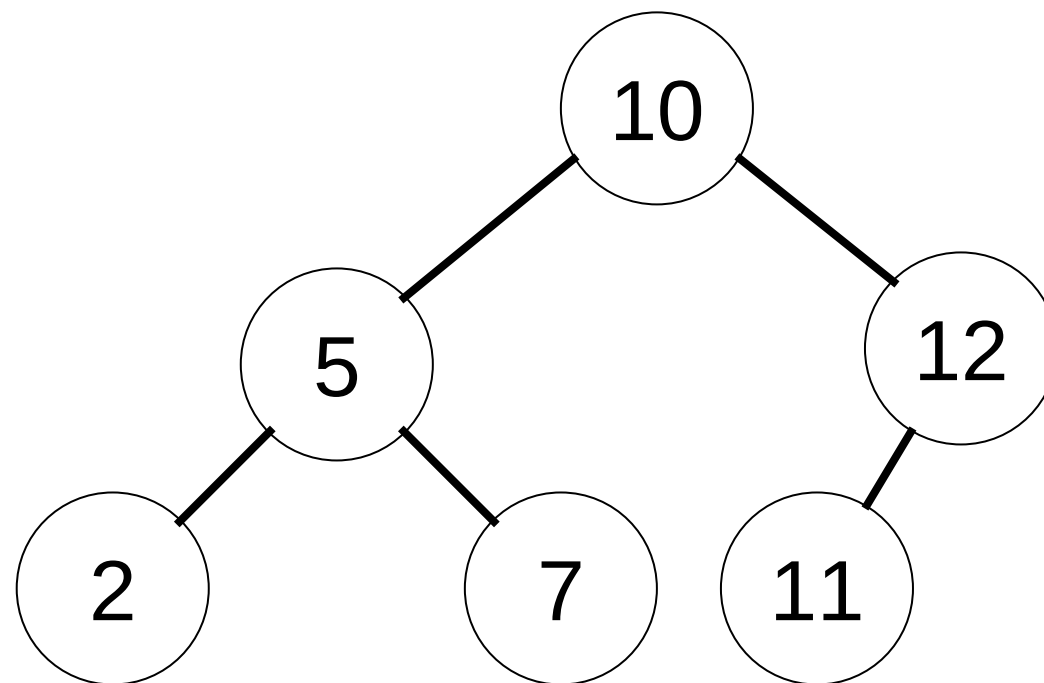
A nyní?

```
void main(void)
{
    TStrom s;
    s.vloz_prvek(12);
    s.vloz_prvek(11);
    s.vloz_prvek(10);
    s.vloz_prvek(7);
    s.vloz_prvek(5);
    s.vloz_prvek(2);

}
```



Snaha je vytvořit strom, kde výška levé a pravé větve se liší maximálně o 1. To musí platit pro libovolný podstrom. Pokud tomu tak není, strom se vyvažuje.



Jak využít dědičnost a virtuální metody u binárního stromu?

- binární strom má tři typy uzlů:
 - uzel se dvěma potomky
 - uzel s levým potomkem
 - uzel s pravým potomkem
 - list
- vytvoříme abstraktní třídu TUzel, od ní odvodíme čtyři třídy pro každý typ uzlu; metoda hledej pak bude virtuální

- takto implementovat jednoduchý binární strom je spíše přístup „s kanónem na vrabce“
- ale poměrně přehledně demonstruje abstraktní třídy a virtuální metody
- příklad: strom2
- náročnější je přidání prvků do stromu
 - zkuste přijít na to, jak to funguje