Generikus keresőfák Erlangban

Patai Gergely*

2008. december 1.

Tartalomjegyzék

1 Bináris keresőfák 1

2 Halmazok 1

3 Asszociatív tár 3

4 Függvények általánosítása 5

5 Függvények egységesítése 7

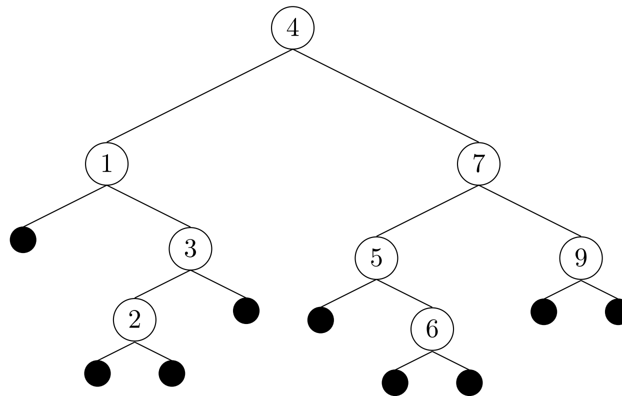
5.1 Halmaz 9

5.2 Asszociatív tár 9

6 Gyakorló feladatok 10

1 Bináris keresőfák

A bináris keresőfa egy olyan fa, amelyben minden belső csomópontnak két gyermeke van, és mindegyikük egy olyan halmaz elemeivel van címkézve, amelyen valamilyen rendezési reláció van értelmezve. A levelek nincsenek címkézve. Azt is feltesszük, hogy a fa véges. Emellett a csomópontok címkéi kielégítik a keresőfa-tulajdonságot: a baloldali gyermekeik minden címkéjénél nagyobbak, és a jobboldali gyermekeik minden címkéjénél kisebbek. Példa egy egész számokkal címkézett bináris keresőfára:



A bináris fákat a következő paraméteres Erlang-típussal fogjuk leírni, ahol Elem a címkék halmazának típusa:

$$\texttt{@type}\ \mathit{tree}\,(\mathsf{Elem}) = \mathit{leaf} \mid \{\mathsf{Elem}, \mathit{tree}\,(\mathsf{Elem}), \mathit{tree}\,(\mathsf{Elem})\}.$$

A típusrendszer nem képes kifejezni a keresőfa-tulajdonságot, így ezt invariánsként kell kikötnünk. Invariáns az olyan tulajdonság, amelyet minden függvény köteles megőrizni: ha a bemenet rendelkezik valamely tulajdonsággal, akkor a kimenetnek is rendelkeznie kell vele – esetünkben a függvény alkalmazása nem ronthatja el az adatstruktúra keresőfa-tulajdonságát.

*Lektorálta, kiegészítette: Hanák Péter

2 Halmazok

A bináris keresőfák egyik legegyszerűbb alkalmazása a halmazok hatékony implementációja. Ha lista helyett fában tároljuk az elemeket, akkor gyorsan el tudjuk dönteni, hogy egy adott elem megtalálható-e egy halmazban. A halmazt tehát bináris fával ábrázoljuk: `@type set() = tree(any())`.

Első lépésként célszerű bevezetni a konverziós műveleteket, amelyekkel listákból halmazokat, illetve halmazokból listákat csinálhatunk. A listák egy egyszerű ciklussal halmazzá alakíthatók: az üres halmazhoz egyenként hozzáadjuk az elemeket.

```
% @spec list_to_set(List::[any()]) -> set().
list_to_set(List) ->
    lists:foldl(fun add_set_elem/2, leaf, List).
```

A másik irányú átalakításhoz be kell járnunk a fát, és egyenként összegyűjtenünk az elemeket. Mivel a listákat egyszerűbb visszafelé gyűjteni, a fát jobbról balra haladva járjuk be. A kapott lista rendezett lesz, ha az adatstruktúránk keresőfa volt.

```
% @spec set_to_list(Set::set()) -> [any()].
set_to_list(Set) ->
    set_to_list(Set, []).
```

```
% @spec set_to_list(Set::set(),List::[any()]) -> [any()].
set_to_list(leaf, List) ->
    List;
set_to_list({Elem, Left, Right}, List) ->
    set_to_list(Left, [Elem | set_to_list(Right, List)]).
```

Üres halmazhoz triviálisan hozzá lehet adni bármilyen elemet. Egyébként meg kell nézni, hogy az új elem hogyan viszonyul a gyökérben található értékhez. Ha egyenlők, akkor a halmaz változatlan marad, és nincs több teendőnk, egyébként a keresőfa-tulajdonságnak köszönhetően el tudjuk dönteni, hogy a bal vagy a jobb részfába kell-e beszúrnunk az új elemet. Véges adatszerkezet esetén a rekurzió során előbb-utóbb eljutunk az üres halmazhoz.

```
% @spec add_set_elem(Elem::any(),Set::set()) -> set().
add_set_elem(Elem, leaf) ->
    {Elem, leaf, leaf};
add_set_elem(Elem, Set = {Elem0, Left, Right}) ->
    if
        Elem0 < Elem -> {Elem0, Left, add_set_elem(Elem, Right)};
        Elem0 > Elem -> {Elem0, add_set_elem(Elem, Left), Right};
        true         -> Set
    end.
```

Egy elem meglétét hasonlóan ellenőrizhetjük. Triviális eset az üres halmaz, és egyértelmű a válasz akkor is, ha a keresett elem egyenlő a gyökérben tárolt értékkel. Egyébként a keresési szabály szerint vagy a bal, vagy a jobb oldali részfában kell folytatnunk a keresést, így előbb-utóbb eljutunk az üres halmazhoz.

```
% @spec is_set_elem(Val::any(),Set::set()) -> bool().
is_set_elem(_Val, leaf) ->
    false;
is_set_elem(Val, {Elem, Left, Right}) ->
    if
        Elem < Val -> is_set_elem(Val, Right);
        Elem > Val -> is_set_elem(Val, Left);
        true       -> true
    end.
```

Ha a szokásos halmazműveleteket szeretnénk bevezetni, szükségünk lesz a halmaz méretét csökkentő műveletekre is. Ezek közül a legegyszerűbb, ha csak egyetlen elemet veszünk ki a halmazból. Üres halmaz esetén nincs semmi

teendőnk, és ha az elem nincs benne a halmazban, akkor a szokásos bejárás során végül ehhez az esethez jutunk el.

Érdekesebb a helyzet, amikor az elem benne van a halmazban. A feladat arra az esetre vezethető vissza, amikor egy fa gyökeréből kell kiszednünk az elemet, mert a teljes fának nem változnak azok a csomópontjai, amelyek nem az adott elem leszármazottai. Ha viszont a gyökeret elvesszük, akkor két különálló részfánk marad, amelyeket valahogy egyesítenünk kell. Ez lényegében az unióképzés művelete, amelyre egy újabb függvényt írunk.

```
% @spec remove_set_elem(Elem::any(),Set::set()) -> set().
remove_set_elem (_Elem, leaf) →
  leaf;
remove_set_elem (Elem, {Elem0, Left, Right}) →
  if
    Elem0 < Elem → {Elem0, Left, remove_set_elem (Elem, Right)};
    Elem0 > Elem → {Elem0, remove_set_elem (Elem, Left), Right};
    true         → union (Left, Right)
  end.
```

Az unió triviális esetei azok, amikor legalább az egyik halmaz üres. Egyébként az egyik halmaz elemeit egyenként kell hozzáadnunk a másik halmazhoz. Ezt megtehetnénk úgy is, hogy előbb az első halmazt listává alakítjuk, de hatékonyabb, ha ezt a lépést kihagyjuk, és a bejárás során rögtön elvégezzük a beszúrásokat. A hatékonyság további növelése érdekében célszerű lenne mindig a kisebb halmaz elemeivel dolgoznunk, de ehhez előbb át kellene alakítanunk az adatstruktúránkat, hogy a méretét hatékonyan lekérdezhessük, így ezzel most nem foglalkozunk.

```
% @spec union(Set1::set(),Set2::set()) -> set().
union (leaf, Set) →
  Set;
union (Set, leaf) →
  Set;
union (Set, {Elem, Left, Right}) →
  union (union (add_set_elem (Elem, Set), Left), Right).
```

Ha a fenti függényben az elemek hozzáadását elemek elvételére cseréljük, és a triviális eseteket is megfelelően módosítjuk, akkor megkapjuk a halmazkülönbség műveletét.

```
% @spec set_diff(Set1::set(),Set2::set()) -> set().
set_diff (leaf, _Set) →
  leaf;
set_diff (Set, leaf) →
  Set;
set_diff (Set, {Elem, Left, Right}) →
  set_diff (set_diff (remove_set_elem (Elem, Set), Left), Right).
```

A halmazkülönbség felhasználásával könnyen kifejezhető a metszetképzés és a tartalmazásvizsgálat is, hiszen $A \cap B = A \setminus (A \setminus B)$, és $A \subseteq B \equiv A \setminus B = \emptyset$.

```
% @spec intersection(Set1::set(),Set2::set()) -> set().
intersection (Set1, Set2) →
  set_diff (Set1, set_diff (Set1, Set2)).

% @spec subset(Set1::set(),Set2::set()) -> bool().
subset (Set1, Set2) →
  set_diff (Set1, Set2) == leaf.
```

A halmazok egyenlősége visszavezethető a kölcsönös tartalmazásra, azaz $A = B \equiv A \subseteq B \wedge B \subseteq A$. A tartalmazásvizsgálat használatakor azonban mindkét halmazt be kellene járnunk, aminél hatékonyabb, ha listává alakítjuk a halmazokat, és a kapott listákat hasonlítjuk össze. Mivel a fent definiált műveletek mind megőrzik a keresőfa-tulajdonságot, ekvivalens fákból azonos listák lesznek.

```
% @spec set_equal(Set1::set(),Set2::set()) -> bool().
set_equal (Set1, Set2) →
  set_to_list (Set1) == set_to_list (Set2).
```

3 Asszociatív tár

Az asszociatív tár, röviden *map*, olyan adatstruktúra, amely képes kulcsokhoz rendelt értékeket tárolni, és a kulcsok alapján visszakeresni, esetleg törölni az adatokat: `@type` *key* () = *any* (), `@type` *val* () = *any* (), @*type* *kv_pair* () = {*key* (), *val* ()}. A kulcsok egyediek, tehát egy kulcs csupán egyszer fordulhat elő a fában (azt a változatot, amely feloldja ezt a megkötést, általában *multimap*-nek nevezik). Ha olyan kulcstípust választunk, amelyen értelmezhető valamilyen rendezési reláció, akkor a *map* megvalósításához keresőfát is használhatunk: `@type` *map* () = *tree* (*kv_pair* ()), hiszen az Erlangban a rendezési reláció univerzális.

A *map* megvalósítását a halmazéhoz hasonlóan célszerű a konverziós műveletekkel kezdeni. Ha ugyanazt a fastruktúrát használjuk, akkor a függvények sem változnak, csak a *map* építésekor értelemszerűen más beszúrófüggvényt kell meghívnunk.

```
% @spec list_to_map(List::[kv_pair()]) -> map().
list_to_map (List) →
  lists : foldl (fun add_map_elem / 2, leaf, List).


% @spec map_to_list(Map::map()) -> [kv_pair()].
map_to_list (Map) →
  map_to_list (Map, []).


% @spec map_to_list(Map::map(),List::[kv_pair()]) -> [kv_pair()].
map_to_list (leaf, List) →
  List;
map_to_list ({Elem, Left, Right}, List) →
  map_to_list (Left, [Elem | map_to_list (Right, List)]).
```

A *map* olyan fa, amelynek a csomópontjai kulcs-érték párokat tárolnak. Ezek a párok biztosan kielégítik a keresőfa-feltételt, ha a kulcs a pár első tagja, mert az ennesek esetén a rendezés lexikografikus az Erlangban, azaz az összehasonlítás balról jobbra halad. Ennek ellenére közvetlenül nem hasonlíthatjuk össze a beszúrandó párt a meglévő párokkal, mert egyező kulcsok esetén hibás eredményt kapnánk. Csak a kulcsokat vehetjük figyelembe a kereséskor, és ha egyezést találunk, az új értéket kell a régi helyébe írnunk a létrehozott fában.

```
% @spec add_map_elem(KeyVal::kv_pair(),Map::map()) -> map().
add_map_elem (KV, leaf) →
  {KV, leaf, leaf};
add_map_elem (KV = {Key, _}, {KV0 = {Key0, _}, Left, Right}) →
  if
    Key0 < Key → {KV0, Left, add_map_elem (KV, Right)};
    Key0 > Key → {KV0, add_map_elem (KV, Left), Right};
    true       → {KV, Left, Right}
  end.
```

Ez abban különbözik a halmaznál látott megoldástól, hogy ott a régi fát adtuk vissza ilyen esetben, hiszen a csomópont értéke nem változott volna meg a felülírás hatására, és csak felesleges munka lett volna új hármast építeni a korábbival egyenlő új érték felhasználásával.

Az egyik legfontosabb művelet természetesen a kulcs alapján történő keresés. Ha a keresett kulcs nincs a fában, akkor a *not_found* atommal jelezzük ezt a tényt, egyébként a *found* atommal párba állítva adjuk eredményül a keresett adatot: `@type` *res* () = *not_found* | {*found*, *val* ()}.

```
% @spec find_map_elem(Key::key(),Map::map()) -> res().
find_map_elem (_Key, leaf) →
  not_found;
find_map_elem (Key, {{Key0, Val0}, Left, Right}) →
  if
    Key0 < Key → find_map_elem (Key, Right);
    Key0 > Key → find_map_elem (Key, Left);
```

```
    true          -> {found, Val0}
  end.
```

Ha a *find_map_elem* függvényt összevetjük az *is_set_elem* függvénnyel, két lényeges különbséget látunk: a) a kereséshez használt érték típusa (kulcs) eltér a csomópont típusától (kulcs-érték pár); b) az eredmény előállításához fel kell használni a megtalált csomópont értékét, nem csak egyszerűen jelezni a létezését.

Egy kulcs hasonló módon törölhető, mint egy halmazelem. A különbség megint csak az, hogy a kulcsot nem a teljes csomópontbeli értékkel hasonlítjuk össze, csak egy részével. A törlés után valamilyen módon össze kell vonni a két részfát, ahogy a halmaznál is láttuk.

```
% @spec remove_map_elem(Key::key(),Map::map()) -> map().
remove_map_elem(_Key, leaf) ->
  leaf;
remove_map_elem(Key, {KV0 = {Key0, _}, Left, Right}) ->
  if
    Key0 < Key -> {KV0, Left, remove_map_elem(Key, Right)};
    Key0 > Key -> {KV0, remove_map_elem(Key, Left), Right};
    true       -> merge_maps(Left, Right)
  end.
```

Két *map* összevonásának eredménye egyértelmű, ha nem tartalmaznak közös kulcsokat. Ez biztosan igaz, ha egy csomópont törlése miatt hívjuk ezt a függvényt, de általánosságban nem jelenthető ki. Az egyező kulcsok konfliktust okoznak, amelyet valahogyan fel kell oldani. Tervezési döntés, hogy ilyen esetben most a második argumentumként átadott fából jövő értéket tartjuk meg.

```
% @spec merge_maps(Map1::map(),Map2::map()) -> map().
merge_maps(leaf, Map) ->
  Map;
merge_maps(Map, leaf) ->
  Map;
merge_maps(Map, {KV, Left, Right}) ->
  merge_maps(merge_maps(add_map_elem(KV, Map), Left), Right).
```

4 Függvények általánosítása

Jól látható a hasonlóság a halmaz és a *map* között, ami nem is meglepő, hiszen mindkét megvalósítás bináris keresőfára épül. A listakonverziós függvények egyesítése a legegyszerűbb. A *list_to_tree* függvény plusz paraméterként megkapja azt a kétargumentumú függvényt, amellyel egy elemet lehet a fához adni.

```
% @type addelem() = fun((any(),tree(any()) -> tree(any())).
% @spec list_to_tree(List::[any()],AddElem::addelem()) -> tree(any()).
list_to_tree(List, AddElem) ->
  lists:foldl(AddElem, leaf, List).
```

A másik irányú konverzió még egyszerűbb, hiszen semmiféle változtatásra nincs szükség. A fa alkalmazástól függetlenül mechanikusan átalakítható listává. A *set_to_list*, *map_to_list* és az új *tree_to_list* függvények teljesen azonosak.

```
% @spec tree_to_list(Tree::tree(any())) -> [any()].
tree_to_list(Tree) ->
  tree_to_list(Tree, []).


% @spec tree_to_list(Tree::tree(any()),List::[any()]) -> [any()].
tree_to_list(leaf, List) ->
  List;
tree_to_list({Elem, Left, Right}, List) ->
  tree_to_list(Left, [Elem | tree_to_list(Right, List)]).
```

Próbáljuk most megírni az *is_set_elem* és a *find_map_elem* közös általánosítását! Mint említettük, két különbség van e két függvény között: a kereséshez szükséges összehasonlítási művelet és a visszatérési érték előállításának módja. Szerencsére mindkét esetben találhatunk közös pontot.

Az összehasonlítás lényegében eldönti, hogy megtaláltuk-e a keresett csomópontot, avagy a bal vagy a jobb részfában kell-e tovább keresnünk. A keresőfáknál ez általában azon múlik, hogy a keresett érték egyenlő, kisebb vagy nagyobb-e, mint az aktuális csomópontban tárolt érték. Ennek leírására definiálunk egy általános összehasonlító függvénysablont, amely egy keresési értékből és egy csomópont értékéből a *less*, *greater* és *equal* atomok valamelyikével tér vissza.

A visszatérési érték előállítható abból a tényből, hogy a keresett csomópont benne van-e a fában, ami mellett igenlő válasz esetén szükség lehet a csomópontban tárolt értékre is. Ehhez egy olyan egyargumentumú segédfüggvényt definiálunk (ld. *get_tree_elem* függvény Getter argumentuma), amely nem létező csomópont esetén a *none* atomot kapja meg, egyébként pedig a csomópont értékét a *some* atommal megjelölt párban.

Először a specifikációt egyszerűsítő típusokat definiáljuk, utána specifikáljuk a segédfüggényt.

```
% @type opt() = none | {some, any()}.
% @type getter() = fun((opt()) -> any()).
% @type rel() = less | equal | greater.
% @type cmp() = fun((any(),any()) -> rel()).


% @spec get_tree_elem(Val::any(),Tree::tree(any()),Getter::getter(),Cmp::cmp() -> any().
get_tree_elem (_Val, leaf, Getter, _Cmp) →
  Getter (none);
get_tree_elem (Val, {Elem, Left, Right}, Getter, Cmp) →
  case Cmp (Elem, Val) of
    less    → get_tree_elem (Val, Right, Getter, Cmp);
    greater → get_tree_elem (Val, Left, Getter, Cmp);
    _       → Getter ({some, Elem})
  end.
```

Lássuk, hogyan valósítható meg a *get_tree_elem* felhasználásával a két specifikus függvény! Halmazok esetén egyszerű a dolgunk, mert a komparáló függvény az Erlang beépített rendezési relációjára épül, a visszatérési érték pedig csak azt jelzi, hogy belső csómóponton vagy levélen állt-e le a rekurzió.

```
% @spec is_set_elem_2(Val::any(),Set::set()) -> rel().
is_set_elem_2 (Val, Set) →
  get_tree_elem (Val, Set, fun (X) → X /= none end,
    fun (E, V) →
      if
        E < V → less;
        E > V → greater;
        true  → equal
      end
    end).
```

A *map* sem sokkal bonyolultabb. A komparálásnál arra kell figyelni, hogy csak a kulcsot vizsgáljuk, a kulcshoz társított értéket ne, a visszatérési érték pedig egyszerűen az atomok cseréjével származtatható a *get_tree_elem* függvényben kapott értékből.

```
% @spec find_map_elem_2(Key::key(),Map::map()) -> res().
find_map_elem_2 (Key, Map) →
  get_tree_elem (Key, Map,
    fun (Elem) →
      case Elem of
        {some, {_, Val}} → {found, Val};
        _                → not_found
      end
    end,
```

```
fun ({K, _}, V) →
  if
    K < V → less;
    K > V → greater;
    true  → equal
  end
end).
```

Hasonló megfontolás alapján a többi műveletet is összevonhatjuk. Ekkor azt fogjuk látni, hogy sokszor ugyanazt a függvényt kell paraméterként átadnunk a különböző műveleteknek. Például a keresés és a törlés esetén ugyanazt a komparáló függvényt kell használnunk a fa bejárására. Erre a problémára a következő szakaszban keresünk megoldást.

5 Függvények egységesítése

Az előzőekben bevezetett általánosítással az első baj az, hogy majdnem ugyanannyi kód kell a paraméteres fakezelő függvények megírásához, mintha közvetlenül implementálnánk a specifikus műveleteket. A másik baj az, hogy ismételnünk kell magunkat, hiszen ahogy újabb műveleteket írunk át, bizonyos függvényeket többször is fel kell használnunk. Ez többletmunka és természetesen újabb hibalehetőség is. Ráadásul külön függvényeket kell definiálnunk csak azért, hogy felparaméterezzük az általános műveleteket, miközben éppen azt várnánk el, hogy az alapműveletek az adattípus megvalósításakor öröklődjenek.

Egy lehetséges megoldás az, hogy a fastruktúrához *mellékeljük* azokat a függvényeket, amelyeket át kell adnunk az általános műveleteket megvalósító függvényeknek. Az eddig említett műveletekhez összesen három ilyen függvényre van szükség:

1. *keycmp*: egy kulcsot és adat kulcsát hasonlítja össze – ez minden keresést igénylő művelethez (adat kinyerése, törlése) kell;
2. *elemcmp*: két adatot hasonlít össze – ez az új csomópont felvétele miatt kell;
3. *getter*: egy adatból előállítja a kért értéket – ezt a visszaolvasásnál használjuk (pl. a halmaz esetén csak egy igen/nem választ ad, míg a *map* esetén kinyeri a kulcshoz tartozó adatot).

Ha elgondolkozunk ezeknek a függvényeknek a típusán, nyilvánvalóvá válik, hogy a generikus fának nem egy, hanem három típusparamétere lesz:

1. Elem: a csomópontokban tárolt érték típusa,
2. Key: a keresési kulcs típusa,
3. Get: a fából visszaolvasott adat típusa.

A generikus fa tehát egy olyan pár lesz, amelynek az első tagja maga a fa, a második tagja pedig a szükséges függvényekből összerakott ennes, esetünkben hármas. Az általános típus így a következő:

```
@type gtree (Elem, Key, Get)  = {tree (Elem), gfuns (Elem, Get, Key)}
@type gtfuns (Elem, Get, Key) = {getter (Elem, Get), keycmp (Elem, Key), elemcmp (Elem)}
@type getter (Elem, Get)      = fun ((maybe (Elem)) → Get)
@type keycmp (Elem, Key)      = fun ((Elem, Key) → ordering ())
@type elemcmp (Elem)          = fun ((Elem, Elem) → ordering ())
```

Bevezettünk két segédtípust is a korábban rögzített konvenciók alapján:

```
@type maybe (X)  = none | {some, X}
@type ordering () = less | greater | equal
```

Ezután már egyszerűen levezethetjük a generikus fából a halmaz és a *map* típusát:

```
@type gset (Elem)      = gtree (Elem, Elem, bool ())
@type gmap (Key, Val) = gtree ({Key, Val}, Key, found (Val))
@type found (X)        = not_found | {found, X}
```

Még azt kell eldöntenünk, hogy pontosan hol definiáljuk a három függvényt. Nyilván célszerű valamilyen konstruktorjellegű műveletbe helyezni őket, de mi legyen ez? Azt láttuk, hogy mind a halmaz, mind a *map* lényegében azonos módon generálható listából, így ezt nem szeretnénk kétszer leírni. Viszont adja magát az ötlet, hogy a kiinduló állapot, az üres fa (halmaz vagy *map*) létrehozásakor tároljuk el a szükséges függvényeket, és ezt az üres struktúrát is adjuk át a *list_to_gtree* függvénynek. Ezek után az általános implementáció ez lehet:

```
% @spec list_to_gtree(Empty::gtree(E,K,G),List::[E]) -> gtree(E,K,G).
list_to_gtree(Empty, List) ->
    lists:foldl(fun update_gtree_elem/2, Empty, List).


% @spec gtree_to_list(Tree::gtree(E,K,G)) -> [E].
gtree_to_list({T, _}) ->
    tree_to_list(T, []).


% @spec update_gtree_elem(Elem::E,Tree::gtree(E,K,G)) -> gtree(E,K,G).
update_gtree_elem(Elem, {Tree, Funs = {_, _, Cmp}}) ->
    {update_gtree_elem(Cmp, Elem, Tree), Funs}.


% @spec update_gtree_elem(Cmp::elemcmp(E),Elem::E,Tree::gtree(E,K,G)) -> tree(E).
update_gtree_elem(_Cmp, Elem, leaf) ->
    {Elem, leaf, leaf};
update_gtree_elem(Cmp, Elem, {Elem0, Left, Right}) ->
    case Cmp(Elem0, Elem) of
        less    -> {Elem0, Left, update_gtree_elem(Cmp, Elem, Right)};
        greater -> {Elem0, update_gtree_elem(Cmp, Elem, Left), Right};
        _       -> {Elem, Left, Right}
    end.


% @spec get_gtree_elem(Key::K,Tree::gtree(E,K,G)) -> G.
get_gtree_elem(Key, {Tree, {Getter, Cmp, _}}) ->
    get_gtree_elem(Getter, Cmp, Key, Tree).


% @spec get_gtree_elem(Get::getter(E,G),Cmp::keycmp(E,K),Key::K,Tree::gtree(E,K,G)) -> G.
get_gtree_elem(Getter, _Cmp, _Key, leaf) ->
    Getter(none);
get_gtree_elem(Getter, Cmp, Key, {Elem, Left, Right}) ->
    case Cmp(Elem, Key) of
        less    -> get_gtree_elem(Getter, Cmp, Key, Right);
        greater -> get_gtree_elem(Getter, Cmp, Key, Left);
        _       -> Getter({some, Elem})
    end.


% @spec remove_gtree_elem(Key::K,Tree::gtree(E,K,G)) -> gtree(E,K,G).
remove_gtree_elem(Key, {Tree, Funs}) ->
    {remove_gtree_elem(Funs, Key, Tree), Funs}.


% @spec remove_gtree_elem(Funs::gtfuns(Elem,Get,Key),Key::K,Tree::tree(E)) -> tree(E).
remove_gtree_elem(_Funs, _Key, leaf) ->
    leaf;
remove_gtree_elem(Funs = {_, Cmp, _}, Key, {Elem, Left, Right}) ->
    case Cmp(Elem, Key) of
        less    -> {Elem, Left, remove_gtree_elem(Funs, Key, Right)};
        greater -> {Elem, remove_gtree_elem(Funs, Key, Left), Right};
```

```
  _       → element (1, merge_gtrees ({Left, Funs}, {Right, Funs}))
  end.
```

```
% @spec merge_gtrees(Tree1::gtree(E,K,G),Tree2::gtree(E,K,G)) -> gtree(E,K,G).
merge_gtrees ({leaf, _}, Tree) →
  Tree;
merge_gtrees (Tree, {leaf, _}) →
  Tree;
merge_gtrees (Tree, {{Elem, Left, Right}, Funs}) →
  merge_gtrees (merge_gtrees (update_gtree_elem (Elem, Tree),
    {Left, Funs}), {Right, Funs}).
```

5.1 Halmaz

A halmaz az egyszerűbbik eset, mert a keresési kulcs és a fában tárolt értékek azonos típusúak. Az uniót leszámítva a halmazműveletekkel most nem foglalkozunk, de a *merge_gtrees* függvény mintájára könnyen írhatnánk egy általános *gtree_diff* függvényt, amelyre a metszet, a tartalmazás és az ekvivalencia is visszavezethető. A konkrét és az általános függvények a következő módon feleltethetők meg egymásnak:

```
list_to_set       → list_to_gtree
set_to_list       → gtree_to_list
add_set_elem      → update_gtree_elem
is_set_elem       → get_gtree_elem
remove_set_elem   → remove_gtree_elem
union             → merge_gtrees
```

```
% @spec empty_set() -> gset(any()).
empty_set () →
  Check = fun (X) → X /= none end,
  Cmp = fun (E, V) →
          if
            E < V → less;
            E > V → greater;
            true  → equal
          end
        end,
  {leaf, {Check, Cmp, Cmp}}.
```

5.2 Asszociatív tár

A halmaz mintájára a *map* is előállítható pusztán a három függvény megadásával, így már tényleg jól látszik, mennyi munkát spórolunk meg az általánosítással. A függvények megfeleltetése magától értetődő:

```
list_to_map       → list_to_gtree
map_to_list       → gtree_to_list
add_map_elem      → update_gtree_elem
find_map_elem     → get_gtree_elem
remove_map_elem   → remove_gtree_elem
merge_maps        → merge_gtrees
```

```
% @spec empty_map() -> gmap(any(),any()).
empty_map () →
  Extract = fun (Elem) →
              case Elem of
```

```
                {some, {_, Val}} -> {found, Val};
                _                -> not_found
            end
        end,
Cmp = fun ({K1, _}, K2) ->
        if
            K1 < K2 -> less;
            K1 > K2 -> greater;
            true    -> equal
        end
    end,
CmpKeys = fun ({K1, _}, {K2, _}) ->
            if
                K1 < K2 -> less;
                K1 > K2 -> greater;
                true    -> equal
            end
        end,
{leaf, {Extract, Cmp, CmpKeys}}.
```

6 Gyakorló feladatok

1. A hiányzó halmazműveletek generikus változatának megírása.
2. A halmazműveletek hatékonyságának növelése úgy, hogy méretinformációt is tárolunk a fában, pl. minden csomópontban a belőle induló részfa csomópontjainak számát.
3. *Multimap* megvalósítása a generikus műveletek felhasználásával. Az elem hozzáadása mindenképpen növelje az adott kulcshoz tartozó találati halmazt (tehát felülírásra nincs lehetőség), törlés esetén pedig az adott kulcs összes előfordulását el kell távolítani.
4. A generikus keresőfák kiegyensúlyozása, azaz AVL fák használata, amellyel egyszerre az összes adatszerkezet minősége javul.