Véletlenített bináris keresőfák: Fapac, Treap

Szerző: Horváth Gyula

2025. október 30.

1. Véletlenített bináris keresőfa (FaPac, Treap)

Láttuk, hogy a bináris keresőfák három alapművelete, a `Keres`, `Bovit` és `Torol` mindegyikének futási ideje a fa magasságával arányos. Számos módszer ismert arra, hogy egyensúlyban tartsuk a fa magasságát. Az AVL-fa esetén a magasság legfeljebb $1.44 \log_2(n)$, ha a fának n pontja van.
Ismert azonban lényegesen egyszerűbb algoritmus, ha csak azt követeljük meg, hogy a fa magasságának várható értéke arányos legyen $\log_2(n)$-el n-pontú fa esetén.
Ezeknek a módszernek az alapja az, hogy ha véletlen a beszúrandó elemek rendezettsége, akkor a fa magasságának várható értéke $\log_2(n)$-el lesz arányos.
Tehát a módszer lényege, hogy véletlenítjük a keletkező fa szerkezetét. Ezt úgy érjük el, hogy minden faponthoz hozzárendelünk a létesítéskor egy `prior` véletlen egész számértéket és az új fapontot a prioritási értéke szerinti helyre szúrjuk be.

1.1. Kupac tulajdonság

A fára pedig a keresőfa-tulajdonság mellett teljesülnie kell a **kupac tulajdonságnak** is: minden `p` pontban a `prior` értéke kisebb, mind mindkét fiában lévő `prior` érték.

`p->prior < p->bal->prior` és `p->prior < p->jobb->prior`

Tehát a bináris fa keresőfa és kupac is, ezért az elnevezése fakupac, *Fapac*.

1.2. FaPont adatszerkezet

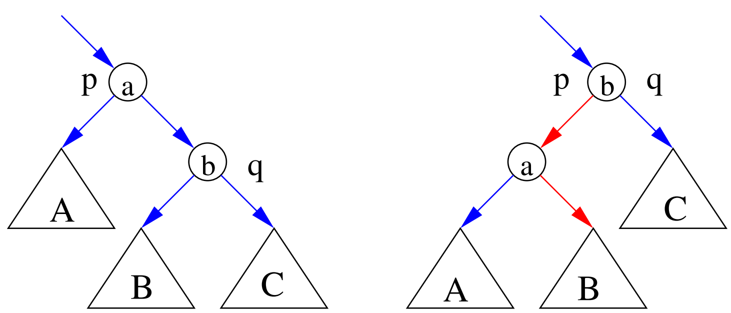
```
const int INF = INT_MAX;
//mt19937 gen(time(nullptr));
mt19937 gen(1234567);//fix sorozat generálása
int randgen(){
  return gen()%INF;
}

typedef int E;
struct BinFaPont;//előre deklarálás NIL miatt
typedef BinFaPont* BinFa;
BinFa NIL;
```

```
struct BinFaPont{
  E adat;    //E típuson értelmezett a < rend. rel.
  int prior;//priororitási érték a minimumos kupachoz
  BinFa bal, jobb;
  BinFaPont(){};
  BinFaPont(E x){//konstruktor
    adat=x;
    bal=NIL,jobb=NIL;
    prior=randgen(); //<INF
  }
};
```

A bővítést úgy végezzük, hogy a standard bővítés után az új fapontot balra-jobbra forgatásokkal visszük felfelé mindaddig, amíg a prioritása kiesebb nem lesz, mint az apja prioritása. Ezzel helyreáll a kupac-tulajdonság.

1.3. Forgatások és a kupac tulajdonság

```
void BForgat(BinFa &p){
  BinFa q=p->jobb;
  p->jobb=q->bal; q->bal=p;
  p=q;
}
```

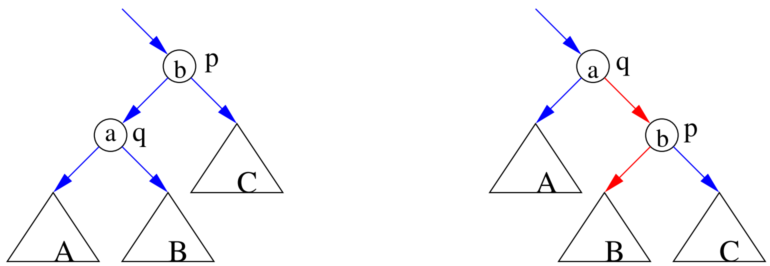


1. ábra. `BForgat` és a kupac tulajdonság

Állítás: Ha a `p`-gyökerű fában csak a `p -> q = p- > jobb` viszonylatban nem teljesül a kupac-tulajdonság, azaz `p- > prior > p- > jobb- > prior`, akkor egy balra forgatás után teljesül a fára a kupac-tulajdonság a `q` gyökerű részfára.

Állítás: Ha a `p`-gyökerű fában teljesül a kupac tulajdonság és `p->bal->prior > p->jobb->prior`, akkor egy balra forgatás után csak a `p` és `p->bal` viszonylatban sérül a kupac tulajdonság a `p`-gyökerű részfában.

```
void JForgat(BinFa &p){
  BinFa q=p->bal;
  p->bal=q->jobb; q->jobb=p;
  p=q;
}
```



2. ábra. `Jforgat` és a kupac tulajdonság

Állítás: Ha a `p`-gyökerű fában csak a `p` $\rightarrow$ `q = p->bal` viszonylatban nem teljesül a kupac-tulajdonság, azaz `p->prior > p->bal->prior`, akkor egy jobbra forgatás után teljesül a fára a kupac-tulajdonság a `q` gyökerű részfára.

Állítás: Ha a `p`-gyökerű fában teljesül a kupac tulajdonság és `p->bal->prior < p->jobb->prior`, akkor egy jobbra forgatás után csak a `p` és `p->jobb` viszonylatban sérül a kupac tulajdonság a `p`-gyökerű részfában.

1.4. Bővítés

```
void Bovit(BinFa &fa, Elemtip x){
    if (fa==NIL){
        fa=new BinFaPont(x);
    }else if (x>fa->adat){
        Bovit(fa->jobb,x);
        //prior helyreállítása
        if(fa->jobb->prior < fa->prior) BForgat(fa);
    }else{
        Bovit(fa->bal,x);
        //prior helyreállítása
        if(fa->bal->prior < fa->prior) JForgat(fa);
    }
}
```

Megjegyzés: fontos, hogy a `NIL` fapont prioritási értéke nagyobb, mint minden más ponté!

1.5. Törlés

Törlés esetén úgy tartható meg a kupac-tulajdonság, hogy a törlendő pontot azzal a forgatással visszük lefelé, amelyik megőrzi a kupac-tulajdonságot, kivéve a törlendő pontot.
Mindkét művelet esetén fontos, hogy az üres fát reprezentáló `NIL` egy létező fapont, amelynek prioritása `INF`, ami nagyobb, mint bármely valódi pont prioritása.

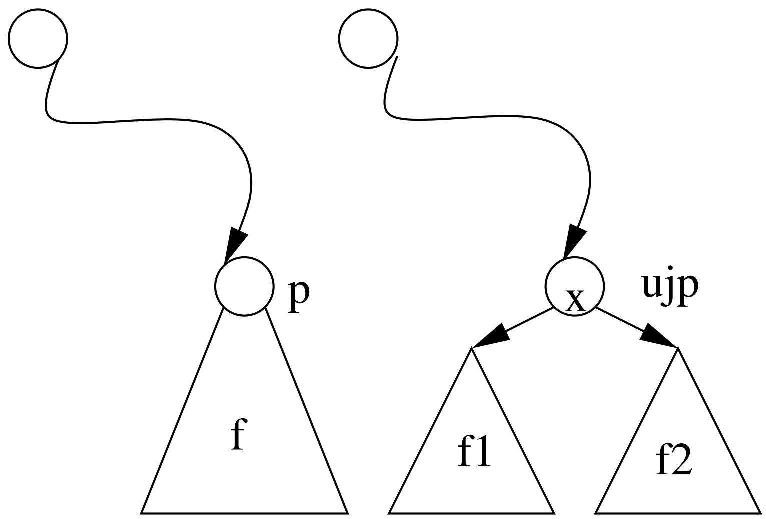
```
BinFa Torol(BinFa &p, Elemtip x){
    if(p==NIL) return NIL;//x nincs a fában
    if(x<p->adat){
        p->bal=Torol(p->bal, x);
    }else if(p->adat<x){
        p->jobb=Torol(p->jobb, x);
    }else{//x==p->adat
        if(p->bal==NIL){
            p=p->jobb;
        }else if(p->jobb==NIL){
            p=p->bal;
        }else if (p->bal->prior < p->jobb->prior){
        //p->bal!=NIL és p->jobb!=NIL
            JForgat(p); p->jobb=Torol(p->jobb,x);
        }else{
            BForgat(p); p->bal=Torol(p->bal,x);
        }
    }
    return p;
}
```

2. Műveletek Vágás és Egyesítés alapján

A bővítés elvégezhető az alábbi módon is.
Hozzunk létre a beszúrandó x adatnak egy új fapontot véletlen prior értékkel. Keressük meg az x-hez tartozó keresőúton az első olyan p pontot, amelyre teljesül, hogy

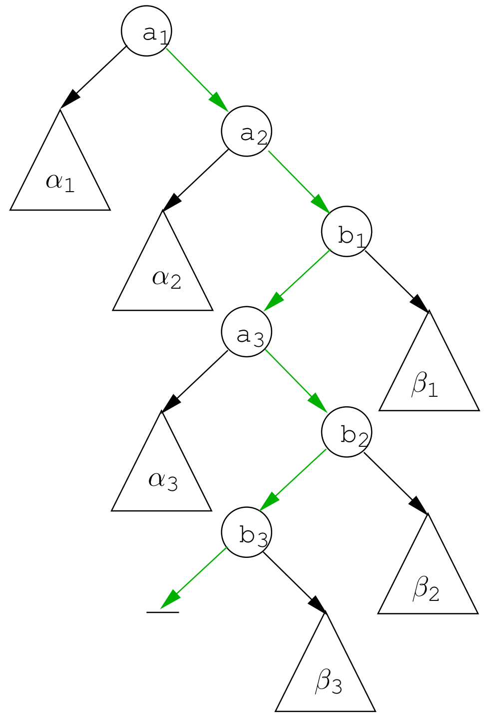
$$\texttt{p}->\texttt{prior} > \texttt{ujp}->\texttt{prior}$$

Ilyen p biztosan létezik, legfeljebb p = NIL, mert NIL prioritása INF. Vágjuk ketté a p-gyökerű f fát olyan f1 és f2 fává, hogy f1 az x-nél kisebb, f2 pedig x-nél nagyobb adatokat tartalmazza és legyen ujp bal-fia f1, jobb-fia f2 és rakjuk f helyébe az ujp gyökerű fát.

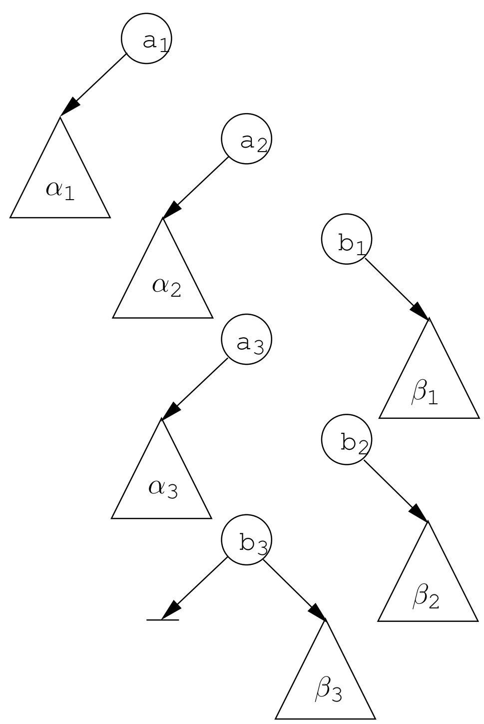


3. ábra

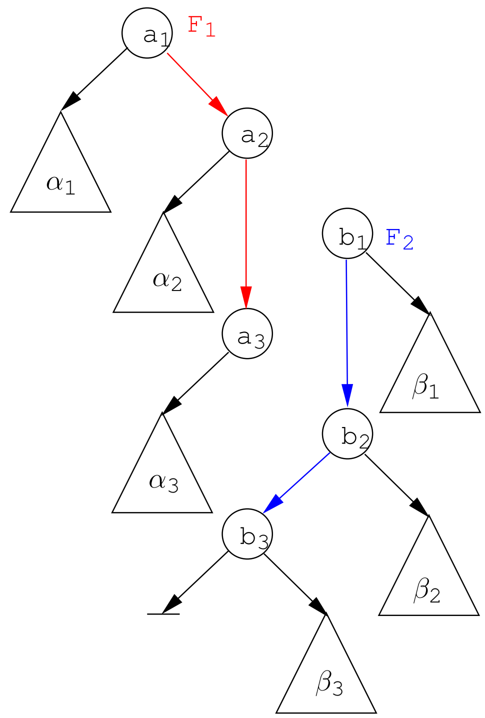
2.1. Vágás



4. ábra. Az x adathoz tartozó bővítőút.

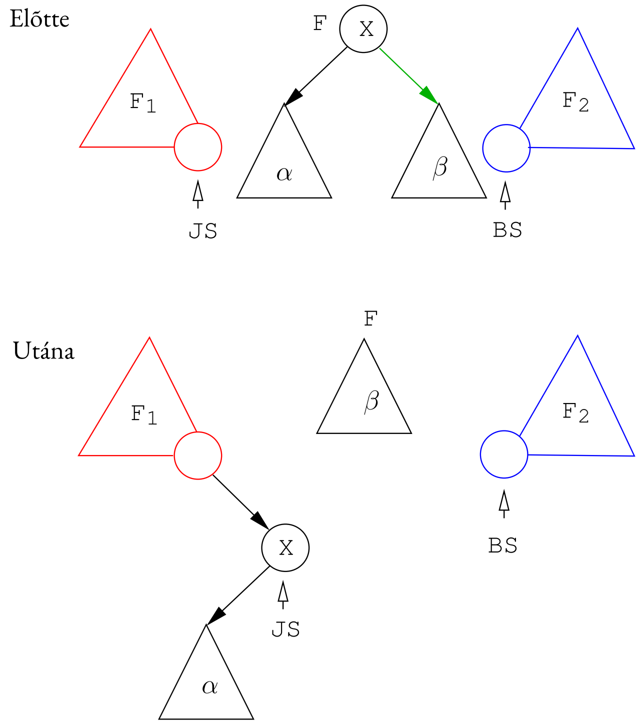


5. ábra. Az F fa x-nél kisebb gyökerű és x-nál nem kisebb gyökerű részfákra bontása.

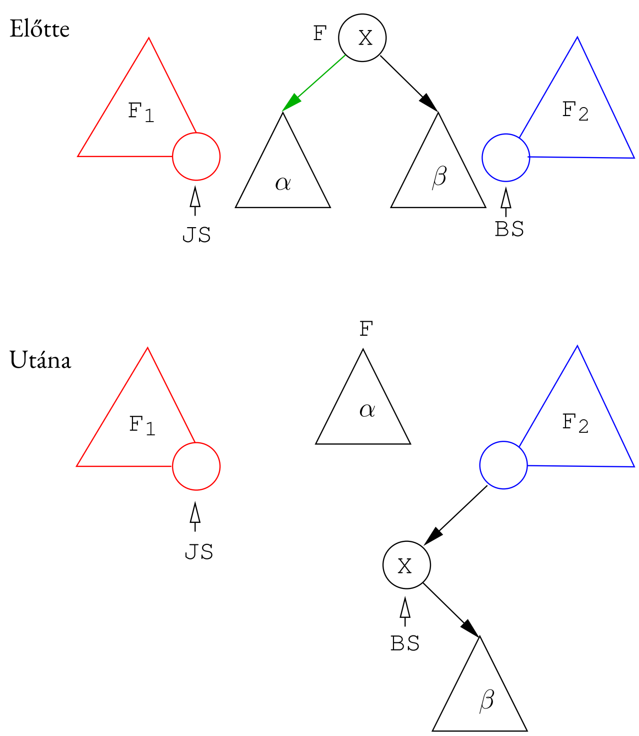


6. ábra. A részfák összekapcsolása F_1 és F_2 fává.

A vágás művelet megvalósítható egy menetben felülről lefelé haladva a következő transzformációs lépésekkel. Minden lépésben a bemeneti fa egy részfáját átkapcsoljuk vagy az F_1 fa jobb sarkához, vagy az F_2 fa bal-sarkához. A lépéseket addig kell ismételni, amíg a bemeneti fa elfogy.



7. ábra. A `T1` vágási transzformáció; feltétel: `X < F-> adat`

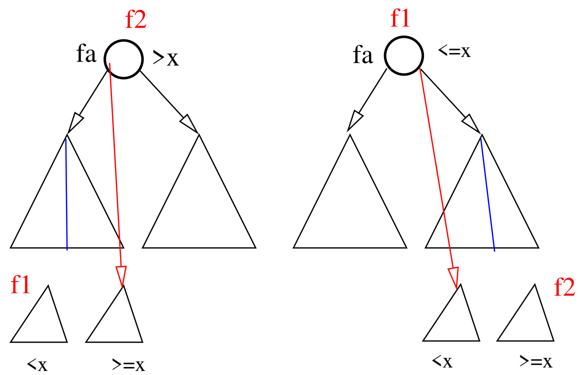


8. ábra. A `T2` : vágási transzformáció; feltétel: `F-> adat < X`

```
void Vag(BinFa fa, E x, BinFa &fa1, BinFa &fa2){
  BinFa Fej=new BinFaPont(); // fiktív gyűjtő cella
  BinFa JS=Fej; // balrész jobb sarka
  BinFa BS=Fej; // jobbrész bal sarka
  while(fa!=NIL){
    if(fa->adat<x){
      JS->jobb=fa;
      JS=fa;
      fa=fa->jobb;
    }else{
      BS->bal=fa;
      BS=fa;
      fa=fa->bal;
    }
  }
  JS->jobb=NIL; BS->bal=NIL;
  fa1=Fej->jobb; fa2=Fej->bal;
  fa=NIL;
}
```

```
BinFa Egyesit(BinFa fa1, BinFa fa2){
  // Feltétel: max(fa1)<=min(fa2)
  if(fa1==NIL) return fa2;
  if(fa2==NIL) return fa1;
  if(fa1->prior < fa2->prior){
    fa1->jobb=Egyesit(fa1->jobb, fa2);
    return fa1;
  }else{
    fa2->bal=Egyesit(fa1, fa2->bal);
    return fa2;
  }
}
```

2.2. Vágás rekurzív megvalósítása



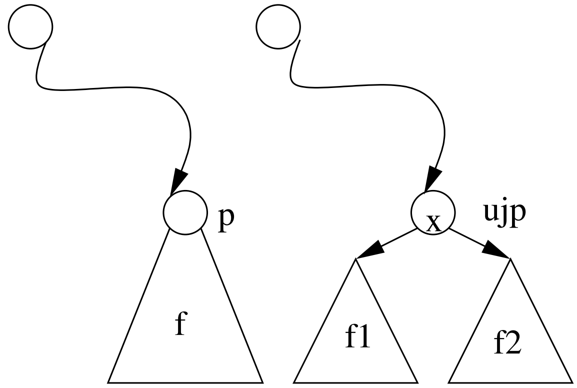
9. ábra. A vágás két esete

```
void Vag(BinFa fa, E x, BinFa &fa1, BinFa &fa2){
   if(fa==NIL){fa1=NIL; fa2=NIL; return;}
   if(x<fa->adat){
      Vag(fa->bal,x, fa1,  fa->bal);
      fa2=fa;
   }else{
      Vag(fa->jobb, x, fa->jobb, fa2);
      fa1=fa;
   }
}
```

2.1. Állítás. *A vágás megőrzi a keresőfa-tulajdonságot és a kupac-tulajdonságot is.*

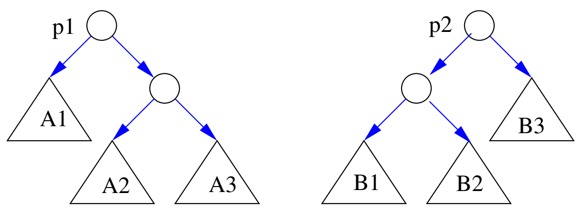
2.3. Bővítés

```
BinFa Bovit(BinFa &fa, BinFa ujp){
   //Bövítés az ujp faponttal
   if(fa==NIL) {fa=ujp; return fa;}
   if(fa->prior > ujp->prior){
      Vag(fa, ujp->adat, ujp->bal, ujp->jobb);
      fa= ujp;
   }else if(ujp->adat<fa->adat){
      fa->bal=Bovit(fa->bal, ujp);
   }else{
      fa->jobb=Bovit(fa->jobb, ujp);
   }
   return fa;
}
```



10. ábra

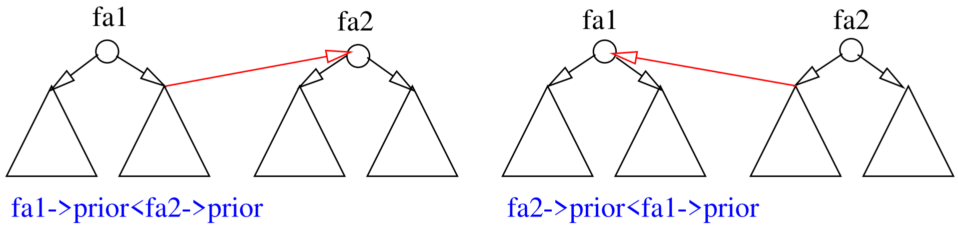
2.4. Egyesítés



11. ábra

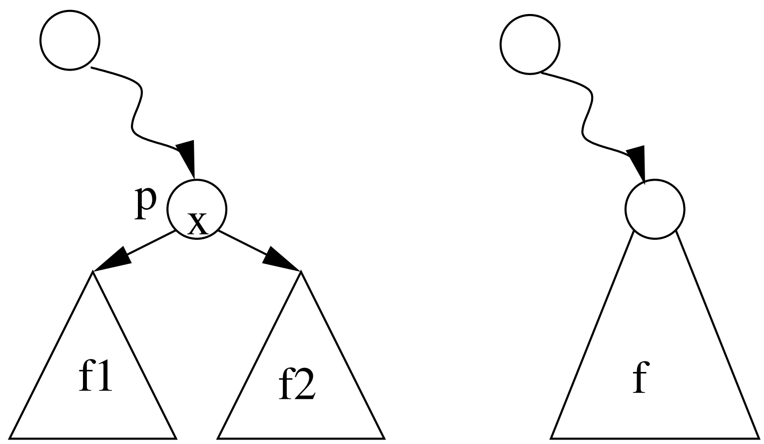
Az egyesítés feltétele, hogy a `p1` gyökerű fában lévő minden adat kisebb, mint a `p2` gyökerűben lévő.
Tehát akár `p1->jobb` gyökerű részfa helyettesíthető `p1->jobb` és `p2` egyesítésével, akár `p2->bal` helyettesíthető `p1` és `p2->bal` egyesítésével, mert mindkét esetben teljesül a keresőfa-tulajdonság. A kupac tulajdonság azonban csak akkor teljesül az egyesítéssel keletkező fára, ha az első esetben `p1->prior < p2->prior`, a második esetben pedig, ha `p2->prior < p1->prior`.

```
BinFa Egyesit(BinFa fa1, BinFa fa2){
  //Feltétel: max(fa1)<=min(fa2)
  if(fa1==NIL) return fa2;
  if(fa2==NIL) return fa1;
  if(fa1->prior < fa2->prior){
    fa1->jobb=Egyesit(fa1->jobb, fa2);
    return fa1;
  }else{
    fa2->bal=Egyesit(fa1, fa2->bal);
    return fa2;
  }
}
```



12. ábra

2.5. Törlés megvalósítása egyesítéssel



13. ábra. A `p` gyökerű részfa helyettesítése az `f = Egyesit(f1, f2)` fával.

```
BinFa Torol(BinFa &p, Elemtip x){
    if(x<p->adat){
        p->bal=Torol(p->bal, x);
    }else if(p->adat<x){
        p->jobb=Torol(p->jobb, x);
    }else{//=x
        p=Egyesit(p->bal, p->jobb);
    }
    return p;
}
```

2.6. Bővít művelet gyors megvalósítása

A Bővít műveletnek van kevésbé hatékony, de rövidebb kóddal történő megvalósítása.

```
void Bovit(BinFa &fa, E x){
   BinFa ujp=new BinFaPont(x);
   if(fa==NIL){ fa=ujp; return;}
   BinFa f1=NIL, f2=NIL;
   Vag(fa, x, f1, f2);
   f2=Egyesit(ujp, f2);
   fa=Egyesit(f1, f2);
}
```

A Fapac, mivel lehetővé teszi a Vág és az Egyesít művelet hatékony megvalósítását, ezért nagyhatású adatszerkezet. Számos olyan művelet hatékonyan megvalósítható vele, amely a sorozat adott `[a, b]` index-tartományába eső részsorozaton végez műveletet. Például maximum, minimum, összeg.
Megjegyezzük, hogy sem az AVL-fák, sem a Piros-fekete fák (általában a magasság kiegyensúlyozó fák) nem támogatják a Vágás és az Egyesítés műveleteket. Megmutatjuk, hogyan valósítható meg az intervallum-maximum művelet.
Követjük az adatszerkezetek bővítésének a stratégiáját:

2.7. A stratégia lépései

1. Az alap adatszerkezet meghatározása.
2. Az alap adatszerkezetben fenntartandó kiegészítő adatok meghatározása.
3. Annak igazolása, hogy a kiegészítő adatok (hatékonyan) fenntarthatók az alap adatszerkezetet módosító műveletek során.
4. Új műveletek kifejlesztése.

Az alap adatszerkezet a Fapac. Kiegészítő adatként tároljuk minden fapontban a `prior` érték mellett az adott pontgyökerű részfa pontjainak `pszam` számát, és a részfában lévő adatok `maxi` maximumát. Ezek az adatok aktualizálhatók a `p` pontban, ha változik a fa, elég csak tudni a két gyerekének a kiegészítő adatait.

```
void frissit(BinFa& fa){
  fa->pszam=fa->bal->pszam + fa->jobb->pszam+1;
  fa->maxi=max(fa->adat,
          max(fa->bal->maxi, fa->jobb->maxi));
}
```

Az alapötlet egyszerű: vágjuk ki a fából azt a részt, amely a kérdéses sorozatot tartalmazza, a gyökerében ott van a kiszámítandó maximum, majd az Egyesít művelettel építsük vissza a fát. A vágást a pontokban lévő `pszam` kiegészítő adat alapján tudjuk elvégezni.

3. A Sorozat Vág és Egyesít és Maximum művelettel

3.1. Műveletek

Elemszam: A sorozat elemszámát adja.

AdatElem(i): A sorozat i-edik elemét adja.

Bovit(i,x): Az i-edik eleme után szúrja be az `x` adatelemet.

Torol(i): Törli a sorozat i-edik elemét.

Modosit(i, x): A sorozat i-edik elemét `x`-re változtatja.

Bejar(Muvel()): A sorozat minden elemére sorrendben végrehajtja a `Muvel` műveletet.

Vag(k, f1, f2): Kettévágja a sorozatot, az f1-be kerülnek azon `i` sorszámú tagjai, amelyekre $\mathtt{i} \leq \mathtt{k}$, a sorozat másik fele f2-be kerül.

Kivag(a, b): Kivágja a sorozatból az `[a, b]` index-intervallumba eső részsorozatát.

Egyesit(f1, f2): Az f1 és sorozat konkatenációját adja.

Maximum(a, b): Az `[a, b]` index-intervallumba eső részsorozat maximumát adja.

3.2. A műveletek megvalósítása Fapac adatszerkezettel

```
const int INF = INT_MAX;
//mt19937 gen(time(nullptr));
mt19937 gen(1234567);//fix sorozat generálása
int randgen(){
  return gen()%INF;
}
typedef int E;
struct BinFaPont;//előre deklarálás NIL miatt
typedef BinFaPont* BinFa;
BinFa NIL;
```

```
struct BinFaPont{
  E adat;   //E típuson értelmezett a < rend. rel.
  int pszam;//pontok száma a p-gyökerű fában
  E maxi;  //a mximális elem a p-gyök fában
  int prior;//priororitási érték a minimumos kupachoz
  BinFa bal, jobb;
  BinFaPont(){};
  BinFaPont(E x){
    adat=x;
    maxi=x;
    pszam=1;
    bal=NIL,jobb=NIL;
    prior=randgen(); //<INF
  }
};
```

```
BinFa Keres(BinFa fa, int i){
  if(i<1 || i>fa->pszam) return NIL;
  while(fa !=NIL && i!=fa->bal->pszam+1){
    if(i<fa->bal->pszam+1)
    fa=fa->bal;
    else{
      i-=(fa->bal->pszam+1);
      fa=fa->jobb;
    }
  }
  return fa;
}
```

```
void frissit(BinFa& fa){
  fa->pszam=fa->bal->pszam + fa->jobb->pszam+1;
  fa->maxi=max(fa->adat,
        max(fa->bal->maxi, fa->jobb->maxi));
}

void Vag(BinFa fa, E k, BinFa &fa1, BinFa &fa2){
  if(fa==NIL) {fa1=NIL; fa2=NIL; return;}
  if(fa->pszam-fa->jobb->pszam<=k){
    Vag(fa->jobb, k-(fa->pszam-fa->jobb->pszam), fa->jobb, fa2);
    fa1=fa;
    frissit(fa1);
  }else{//k<fa->pszam
    Vag(fa->bal, k, fa1, fa->bal);
    fa2=fa;
    frissit(fa2);
  }
}

void Kivag(BinFa& fa, int a, int b){
  //1<=a<=b<=fa->pszam
  BinFa f1,f2;
  Vag(fa, b, fa, f1);
  Vag(fa, a-1, fa, f2);
  fa=Egyesit(fa,f1);
  frissit(fa);
}

BinFa Egyesit(BinFa fa1, BinFa fa2){
  if(fa1==NIL) return fa2;
  if(fa2==NIL) return fa1;
  if(fa1->prior < fa2->prior){
    fa1->jobb=Egyesit(fa1->jobb, fa2);
    frissit(fa1); return fa1;
  }else{
    fa2->bal=Egyesit(fa1, fa2->bal);
    frissit(fa2); return fa2;
  }
}
```

```
void Bovit(BinFa &fa, int i, E x){
  //0<=i<=fa->pszam
  BinFa ujp=new BinFaPont(x);
  if(fa==NIL){fa=ujp; return;}
  BinFa f1=NIL, f2=NIL;
  Vag(fa, i, f1, f2);
  f2=Egyesit(ujp, f2);
  fa=Egyesit(f1, f2);
  frissit(fa);
}

void Torol(BinFa &fa, int i){
  Kivag(fa, i,i);
}

E Elem(BinFa fa, int i){
  fa=Keres(fa, i);
  return fa->adat;
}
void Modosit(BinFa fa, int i, E x){
  fa=Keres(fa, i);
  fa->adat=x;
}
void Bovit(BinFa &fa, int i, E x){
  BinFa ujp=new BinFaPont(x);
  if(fa==NIL) {fa=ujp; return;}
  Bovit(fa, i, ujp);
}

E Maximum(BinFa fa, int a, int b){
  //1<=a<=b<=fa->pszam
  BinFa f1,f2;
  Vag(fa, b, fa, f1);
  Vag(fa, a-1, fa, f2);
  E m=f2->maxi;
  fa=Egyesit(fa,f2);
  fa=Egyesit(fa,f1);
  return m;
}
```

4. A Sorozat műveletek hatékonysága

Nyilvánvaló, hogy az Elemszám kivételével minden művelet futási ideje a Fapac fa magasságával arányos. A Fapac fa magasságának átlagos értéke (várható értéke) $O(\log_2(n))$ ha a fának n pontja van.
Sorozatok más ábrázolásait nézve látható, hogy milyen esetekben célszerű Fapac-ot használni.

$$A = (a_1, \ldots, a_n)$$

Ábrázolás (dinamikus) tömbbel

Ekkor az index-szerinti elérés konstans idejű, de többi esetben a sorozat elemszámával lesz arányos.

Ábrázolás (kétirányú) lánccal

Ekkor, ha ismerjük a sorozat elemének helyét a sorozatban referencia értékkel, akkor a művelet futási ideje konstans idejű. Minden más esetben a sorozat hosszával arányos lesz.

4.1. Szegmensfa vagy Fapac

A Fapac használata hasonlóságot mutat a szegmensfa olyan alkalmazásával, amikor sorozat részsorozatain (intervallumokon) értelmezett függvényt kell kiszámítani. A szegmensfa esetén az ilyen kiszámítás a fa magasságával arányos időben megtehető, ha a kiszámítandó `f` függvény "tördelhető", ami azt jelenti, hogy van olyan konstans időben kiszámítható `f2` függvény, hogy minden $\mathtt{i} \leq \mathtt{k} < \mathtt{j}$ esetén

$$\mathtt{f([i,\ j])} = \mathtt{f2(f([i,\ k]),\ f([k+1,\ j]))}$$

Ez a hatékonyság azzal érhető el, hogy előszámítást végzünk, lineáris számú részsorozatra, amelyek fát alkotnak, és a tördelhetőség miatt minden intervallumra a fa magasságával arányos időben kiszámítható a függvény értéke az előszámított intervallumokból. A Fapac esetén a részek nem intervallumok, hanem részfák. Ezek lesznek az előszámított részek. A "tördelhetőség" azt jelenti, hogy a `p` gyökerű fához tartozó `f` érték konstans időben kiszámítható a fapontban lévő adat és a részfákhoz tartozó `f` értékekből, azaz van olyan konstans időben kiszámítható `f3` függvény, hogy:

$$\mathtt{p->f} = \mathtt{f3(p->adat,\ p->bal->f,\ p->jobb->f)}$$

4.2. Különbségek

A Fapac esetén is a műveletek a fa magasságával arányos futási idejűek.

1. Fapac esetén lehet törölni és beszúrni is.
2. Fapac esetén a sorozat rendezett is lehet, ekkor az elemei elérhetők pozíció szerint és érték szerint is.
3. Szegmensfa esetén nincs Vág és Egyesít művelet.

Számos alkalmazás esetén egy kezdő halmazból, vagy sorozatból kiindulva kell műveleteket végezni. Mivel a sorozatot (halmazt) a bináris fa Inorder bejárásával reprezentáljuk, ezért a kezdeti sorozatot reprezentáló bináris fát elő tudjuk állítani úgy, hogy a fa teljesen kiegyensúlyozott legyen, így a magassága $\lceil \mathtt{log_2(n)} \rceil$ lesz.

```
BinFa FaEpit(int a, int b){
   //Globális: S
   //az S[a..b] sorozatot reprezentáló fát adja
   if(a>b) return NIL;
   int k=(a+b)/2;
   BinFa f=new BinFaPont(S[k]);
   f->bal=FaEpit(a,k-1);
   f->jobb=FaEpit(k+1,b);
   if(f->prior > f->bal->prior)
   swap(f->prior, f->bal->prior);
   if(f->prior > f->jobb->prior)
   swap(f->prior, f->jobb->prior);
   frissit(f);
   return f;
}
```

5. Feladat 1: Mester:Rekurzív adatszerkezetek: Tárgyak sorban.

Sorban egymás után N tárgy helyezkedik el. Egy robot megy végig a soron és minden tárgyra meg kell mondania, hogy előtte a sorban hány olyan tárgy van, amely az adott tárgynál nem magasabb.
Készíts programot, amely megoldja a robot feladatát!

Bemenet

Bemenet A standard bemenet első sorában van a tárgyak `N` száma. A következő `N` sor mindegyike egy tárgy m_i magassági értékét tartalmazza.

Kimenet

Kimenet A standard kimenet első és egyetlen sorába `N` számot kell írni, az i-edik szám azon tárgyak száma legyen, amelyek megelőzik a sorban az i-edik tárgyat és magasságuk legfeljebb akkora, mint az i-edik tárgyé!

Példa

Bemenet	Kimenet
7 9 19 26 28 4 28 71	0 1 2 3 0 5 6

Korlátok

Korlátok $1 < N \leq 100\,000$, $1 \leq m_i \leq 100\,000\,000$
Időlimit: 0.1 másodperc
Memórialimit: 64 MB.

Pontozás

Pontok A pontok 10%-át lehet szerezni olyan bemenetekre, ahol $N \leq 1000$.
A pontok további 40%-át lehet szerezni olyan bemenetekre, ahol $N \leq 10\,000$
A pontok további 50%-át lehet szerezni olyan bemenetekre, ahol nincs egyéb korlátozás.

6. Feladat 2: Mester:Rekurzív adatszerkezetek: Tükrözések.

Adott egy szöveg, amelyen olyan műveleteket kell végrehajtani, amelyek mindegyike adott részsorozatot kicserél a tükörképére. Készíts programot, amely megadja, mi lesz az eredmény az összes művelet elvégzése után!

Bemenet

Bemenet A standard bemenet első sorában lét egész szám van, a bemeneti szöveg hosszának N száma, és a végrehajtandó műveletek M száma. A második sorban van a kezdeti S szöveg, string típusú adatként. A szöveg csak az angol ábécé nagybetűit tartalmazza. A további M sor mindegyike egy tükrözés művelet argumentumát adja meg a b számpár formájában ($1 \leq a \leq b \leq N$), ami azt jelenti, hogy az S szöveg [a, b] zárt index-intervalluma által meghatározott részsorozatot helyettesíteni kell a tükörképével.

Kimenet

Kimenet A standard kimenetre egy sort kell írni, az összes művelet elvégzése után kapott szöveget!

Példa

Bemenet	Kimenet
6 2 ABRAKA 2 5 1 4	RAKABA

Korlátok

Korlátok $1 < N \leq 200\,000$, $1 \leq M \leq 100\,000$
Időlimit: 0.3 másodperc
Memórialimit: 64 MB.

Pontozás

Pontok A pontok 10%-át lehet szerezni olyan bemenetekre, ahol $N \leq 1000$; és $M \leq 1000$.

7. Gyakorló feladatok

Mester: Rekurzív adatszerkezetek: Intervallumok.
https://codeforces.com/gym/102787/problem/A
https://codeforces.com/gym/102787/problem/Z
https://codeforces.com/gym/102787/ X,E,C
http://www.spoj.com/problems/ADACROP/ - Ada and Harvest
http://www.spoj.com/problems/COUNT1IT/ - Ghost Town
http://www.spoj.com/problems/ALLIN1/ - All in One
http://codeforces.com/contest/847/problem/D - Dog Show
http://codeforces.com/contest/863/problem/D - Yet Another Array Q
http://www.spoj.com/problems/MEANARR/ - Mean of Array
http://www.spoj.com/problems/TWIST/ - TWIST
https://www.codechef.com/problems/PRESTIGE - The Prestige
https://codeforces.com/contest/702/problem/F - T-Shirts
https://codeforces.com/problemset/problem/167/D - Wizards and Roads
https://codeforces.com/contest/295/problem/E - Yaroslav and Points