1. Feladat

Állítsuk csökkenő sorrendbe a standard bemenetről beolvasott egész számokat, és írjuk ki őket a standard kimenetre!

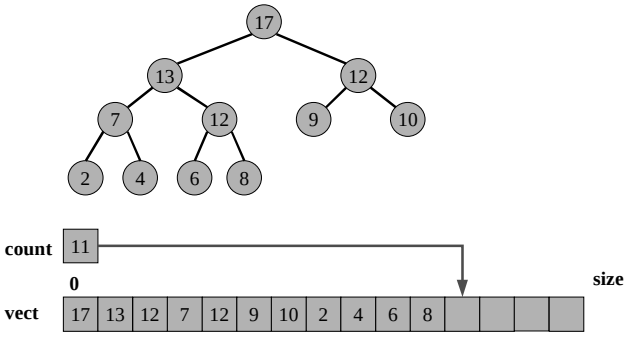
Megoldás

A feladat megoldásához használjunk egy olyan elsőbbségi (prioritásos) sort, amelyet kupac (halom, piramis, heap) adatszerkezettel fogunk ábrázolunk. Egy elsőbbségi sorra az alábbi műveleteket engedjük meg:

- **`Insert`**: új érték behelyezése a sorba
- **`Opt`**: maximális érték kiolvasása a sorból
- **`Extract_Opt`**: maximális érték kivétele a sorból
- **`Empty`** sor ürességének vizsgálata
- **`Full`** tele sor esetén ad igazat

Ennek a típusnak a használatával a feladat megoldása magától értetődik: a beolvasott számokat betesszük a sorba, majd egymás után kivesszük a sor éppen aktuális maximális elemét, amíg a sor nem ürül ki.

A kupac egy olyan balra tömörített bináris fa, amelynek minden csúcsa olyan elemet tartalmaz, amely nagyobb vagy egyenlő az abból leszármazott csúcsokban tárolt elemeknél. Így a maximális elem a fa gyökerében található.



A kupacot sorfolytonosan, egy rögzített hosszú (**`size`**) tömbben (**`vect`**) ábrázoljuk, és tényleges kitöltöttségét a betett elemek számával (**`count`**) jelöljük. A maximális gyökérelem a `vect[0]`. A fa bejárását három függvény segíti:

- **`Parent(i)`**: az `i` indexű csúcs szülőjének indexe: `(i-1)/2`
- **`Left(i)`**: az `i` indexű csúcs balgyerekének indexe: `2*i+1`
- **`Right(i)`**: az `i` indexű csúcs szülőjének indexe: `2*i+2`

Két segédfüggvényt is bevezetünk, amelyeket a kupac tulajdonság helyreállítására használunk:

- **`Up(i)`**: a `vect` `i`-dik elemét sorozatos cserékkel felemeli a kupacban, ha felette nálánál kisebb elemek vannak
- **`Down(i)`**: a `vect` `i`-dik elemét sorozatos cserékkel lesüllyeszti a kupacban, ha alatta nálánál nagyobb elemek vannak

Megoldás C++-ban

Elsőbbségi sor-osztály definíciója

Készítsünk az elsőbbségi sor típusának leírására egy osztálysablont, ahol a sorba helyezhető elemek típusa majd paraméterként adható majd meg.

```
template <class Element>
class Priority_Queue {
  public:
    enum Exceptions{EMPTY_PQ, FULL_PQ};

    Priority_Queue(const int size);

    void  Insert(  const Element& e);
    Element Extract_Opt();

    Element Opt()  const { return vect[0];}
    bool  Empty()  const { return count==0;}
    bool  Full()   const { return count==size;}

    ~Priority_Queue(){ delete[] vect;}

  protected:
    int    size;
    Element* vect;
    int    count;

    int   Left  (int i)const {return (i<<1)+1;}
    int   Right (int i)const {return (i<<1)+2;}
    int   Parent(int i)const {return (i-1)>>1;}

    void  Up  (int i);
    void  Down(int i);
    virtual Element Get(int i) {return vect[i];}
    virtual void  Put(const Element& e, int i) {vect[i] = e;}

    Priority_Queue(const Priority_Queue&);
    Priority_Queue& operator=(const Priority_Queue&);
};
```

A sor destruktora, az `Opt`, `Empty`, `Full` metódusok, valamint három rejtett bejáró függvény (`Parent`, `Right`, `Left`) és a két speciális célú rejtett függvény (`Get`, `Put`) „inline” definíciót tartalmaz. A 2-vel történő osztást és szorzást bit léptetéssel oldottuk meg. A rejtett copy konstruktor és értékadás operátor deklaráció letiltja az elsőbbségi sor-objektumok érték szerinti paraméterátadását és értékadását. A `Get` és `Put` függvény bevezetését ebben a megoldásban semmi sem indokolja, szerepükre később, a következő előadásban térünk majd vissza.

Elsőbbségi sor metódusainak definíciója

Az alábbi kódrészletek – osztálysablonról lévén szó – ugyanabban a header fájlban helyezkednek el, mint a fenti osztálysablon definíció.

A sor konstruktora megadott méretű tömböt foglal le a kupac számára, és a `last` mutatót `-1`-re állítja.

```
template <class Element>
Priority_Queue<Element>::Priority_Queue(const int s)
{
    size  = s;
    vect  = new Element [size];
    count = 0;
}
```

Az `Insert` művelet beírja az új elemet a tömb utolsó eleme után, majd meghívja erre az elemre az `Up` segédfüggvényt miután megnövelte eggyel a `count` értékét.

```
template <class Element>
void Priority_Queue<Element>::Insert(const Element& e)
{
    if( count==size ) throw FULL_PQ;
    Put(e,count);
    count++;
    Up(count-1);
}
```

Az `Extract_Opt` művelet kiolvassa a tömb első elemét, csökkenti a `count` értékét eggyel, majd átmásolja előre az utolsó elemet, végül meghívja az így kapott első elemre a `Down` segédfüggvényt.

```
template <class Element>
Element Priority_Queue<Element>::Extract_Opt()
{
    if( count==0 ) throw EMPTY_PQ;
    Element e = Get(0);
    Put(vect[count-1],0);
    count--;
    Down(0);
    return e;
}
```

A kupacként megvalósított elsőbbségi sor „lelke” az elrejtett felemelő illetve lesüllyesztő segédfüggvény:

```
template <class Element>
void Priority_Queue<Element>::Up( int i )
{
    Element e = vect[i];
    int   j = Parent(i);
    while( j>=0 && e>vect[j] ) {
        Put(vect[j],i);
        i = j;
        j = Parent(i);
    }
    Put(e,i);
}

template <class Element>
void Priority_Queue<Element>::Down( int i )
{
    Element e = vect[i];
    int j = (Right(i)>=count || vect[Left(i)]>vect[Right(i)])
          ? Left(i) : Right(i);
    while(Left(i)<count && vect[j]>e){
        Put(vect[j],i);
        i = j;
        j = (Right(i)>=count || vect[Left(i)]>vect[Right(i)])
          ? Left(i) : Right(i);
    }
    Put(e,i);
}
```

Vegyük észre, hogy `Element` paraméterként csak olyan elemi típus adható meg, amelynek elemein értelmezett a `>` operátor (a copy konstruktoron és értékadás operátoron kívül).

Főprogram

A főprogramban a speciális 9999 érték megjelenéséig olvasunk be egész számokat a standard bemenetről, ezeket a sor-objektumba helyezzük, majd kiürítjük a sor a standard kimenetre.

```
#include <iostream>
#include "priority_queue.h"

using namespace std;

int main()
{
    Priority_Queue h(100);
    int e;
    cin>>e;
    while (e!=9999){
        h.Insert(e);
        cin>>e;
    }

    cout << ”A beolvasott számok csökkenően rendezve:” << endl;
    while(!h.Empty()){
        cout << h.ExtractOpt() << endl;
    }

  char ch;
  cin>>ch;
  return 0;
}
```

2. Feladat

Állítsuk ábécé szerint növekvő sorrendbe a standard bemenetről beolvasott szavakat, és írjuk ki őket a standard kimenetre!

1. Megoldás

Az előbb elkészített elsőbbségi sor paramétereként természetesen a `string` típust is meg lehet adni. Ez az elsőbbségi sor a benne tárolt sztringek közül legnagyobbat tudja visszaadni, hiszen a `string` típus `>` operátorát használja. Első gondolatunk az, hogy a `>` operátort is paraméternek tekinthetnénk, és a példányosításnál a megfelelő rendezési relációt leíró függvényt írhatnánk a helyébe. Sajnos a C++ nem enged meg egy osztálysablon paramétereként függvényt, így nem tudjuk egyszerűen „lecserélni” a `>` operátort. Ha tárolt sztringek közül legkisebbet akarjuk kiolvasni a sorból, akkor olyan `string` típussal kell paraméterezni, amely ellentétesen értelmezi a `>` operátort. Egy ilyen típust a `string` osztályból származtathatunk úgy, hogy felül definiáljuk annak `>` operátorát, de az öröklődés miatt minden más sztringeken értelmezett művelet használható. Ezt a trükköt nemcsak a `string` típusra lehet megcsinálni, hanem bármelyik osztályra egy úgynevezett rendezési relációt megfordító osztálysablon segítségével:

```
template <class Element>
class ClassReversed : public Element {
public:
    ClassReversed(){};
    ClassReversed(Element n): value(n){};
    bool operator>(const ClassReversed& r)
            {return value <r.value;}
    bool operator<(const ClassReversed& r)
            {return value >r.value;}
private:
    Element value;
};
```

Sajnos ez az osztálysablon nem alkalmazható az alap típusokra, így például az `int`-re sem, mert alap típusból nem lehet osztályt származtatni. Ezekhez egy másik rendezési relációt megfordító osztálysablont kell készíteni:

```
template <class Element>
class TypeReversed  {
public:
    TypeReversed(){};
    TypeReversed(const Element& v):value(v){};
    bool operator>(const TypeReversed& r) const
            {return value<r.value;}
    bool operator>(const TypeReversed& r) const
            {return value<r.value;}
    Element GetValue(){return value;}
    void SetValue(Element& e){value=e;}
private:
    Element value;
};
```

Ennek az osztálynak az objektumaira nem öröklődnek az alaptípus műveletei, csak a value adattagjára, amelyet a `GetValue()` függvénnyel tudunk kiolvasni, és a `SetValue()`függvénnyel megváltoztatni .

Végül a főprogram:

```
int main()
{
    Priority_Queue<ClassReversed<string> > h(50);

    ClassReversed<string>  str;
    cin>>e;
    while( e!=”q” ){
        h.Insert(str);
        cin>>e;
    }

    while( !h.Empty() ){
        cout << h.Extract_Opt() << endl;
    }
    cout << endl;

  char ch;
  cin>>ch;
  return 0;
}
```

2. Megoldás

Ez előző megoldás alternatívája az, amikor elkészítjük az elsőbbségi sor azon változatát, amelyikben a „>” reláció helyett a „<” relációt használjuk. Ezt úgy érhetjük el elegánsan, ha a korábbi elsőbbségi sor osztálysablont ellátjuk egy újabb paraméterrel, ami a rendezési relációra utal.
Mivel paraméter csak osztály lehet, célszerű előre elkészíteni az alábbi két összehasonlító osztálysablont, amelyek közül az egyikkel kell majd az elsőbbségi sorunkat példányosítani:

```
template <class Element>
class Greater{
public:
  bool operator()(const Element& a, const Element& b){return a>b;}
};

template <class Element>
class Less{
public:
  bool operator()(const Element& a, const Element& b){return a<b;}
};
```

Ellátjuk az elsőbbségi sor sablonját egy új paraméterrel, amelynek alapértelmezés szerinti értéke „>” összehasonlítást szolgáltató osztály. Felveszünk egy új védett adattagot, amelynek típusa a paraméterként átvett összehasonlító osztály lesz:

```
template <  class Element, class Compare = Greater<Element> >
class Priority_Queue{
public:
    ...
protected:
    int      size;
    Element* vect;
    int      count;
    Compare  c;
    ...
};
```

Ezen kívül csak az `Up` és a `Down` metódusok megvalósítása változik, hiszen csak ezekben kellett elemeket összehasonlítani. Ezt most a „>” operátor helyett az összehasonlító objektum függvényhívás operátorán keresztül végezzük el:

```
template <class Element, class Compare = Greater<Element> >
void Priority_Queue<Element, Compare>::Up( int i )
{
    Element e = vect[i];
    int   j = Parent(i);
    while( j>=0 && c(e,vect[j]) ) {
        Put(vect[j],i);
        i = j;
        j = Parent(i);
    }
    Put(e,i);
}

template <class Element, class Compare = Greater<Element> >
void Priority_Queue<Element, Compare>::Down( int i )
{
    Element e = vect[i];
    int  j;
    bool l = true;
    while( l && Left(i)<count ) {
        j = (Right(i)>=count || c(vect[Left(i)],vect[Right(i)]))
            ? Left(i) : Right(i) ;
         if( l = c(vect[j],e) ){
            Put(vect[j],i);
            i = j;
         };
    }
    Put(e,i);
}
```

A főprogramban az alábbi példányosítás biztosítja a kívánt elsőbbségi sort:

```
Priority_Queue<string, Less<string> > h(20);
```

3. Feladat

Készítsünk olyan elsőbbségi sor osztálysablont, amelyben csak egyedi értékeket lehet tárolni, azaz egy érték nem fordulhat elő többször. Vezessük be azt az új függvényt, amely eldönti, hogy egy érték a sorban van-e.

Megoldás

Az előbbi `Priority_Queue` osztálysablont csak kis mértékben kell módosítanunk. Ezt úgy végezzük el, hogy definiálunk egy új `Unique_Priority_Queue` osztálysablont, amelyet a `Priority_Queue` osztálysablonból származtatunk. Az ősosztálybeli `Insert` műveleten mindössze annyit kell változtatni, hogy annak meg kell tagadni egy már létező kulcsú elem új elemként történő felvételét. Ehhez majd felhasználjuk azt az `In` (egy elem a sorban van-e) műveletet, amellyel a feladat szövege szerint ki kell egészíteni a metódusainkat. Az `In` metódushoz el kell készítenünk azt a lineáris keresést, amely az elsőbbségi sort ábrázoló `vect` tömbben keresi a megadott elemet. Ezt a keresést a `Search` rejtett virtuális metódusba ágyazzuk bele. Ha a `Search` megtalálja a keresett elemet, akkor visszaadja annak `vect`-beli indexét, ha nem, akkor a `–1`-et.

```
template <class Element>
int Unique_Priority_Queue<Element>::Search(const Element& e)
const
{
    int i = count-1;
    while( i>=0 && e!=vect[i] ) i--;
    return i;
}
```

A `Unique_Priority_Queue` osztálysablon definíciójában a konstruktor, az `Insert` és az `In` metódus „inline” definícióval rendelkezik.

```
#include "priority_queue.h"

template <class Element>
class Unique_Priority_Queue : public Priority_Queue<Element> {
public:
    enum Exceptions{EMPTY, FULL, NOTFOUND, EXISTING};

    Unique_Priority_Queue(const int s):
                          Priority_Queue<Element>(s){}
    void Insert(const Element& e)
    {
        if (In(e)) throw EXISTING;
        Priority_Queue<Element>::Insert(e);
    };
    bool In(const Element& e) const {return Search(e)>=0;}

 protected:
     virtual int Search(const Element& e) const;
};
```