ELEMI ALKALMAZÁSOK FEJLESZTÉSE II.

Elsőbbségi sor kupaccal

Készítette: Gregorics Tibor

Tartalom

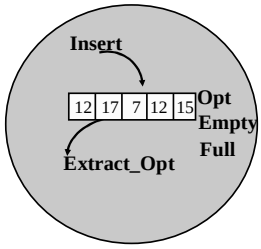
- Elsőbbségi sor kupac adatszerkezettel
- Elsőbbségi sor rendezési relációja
- Elsőbbségi sor egyedi azonosítójú elemekkel

1. Feladat

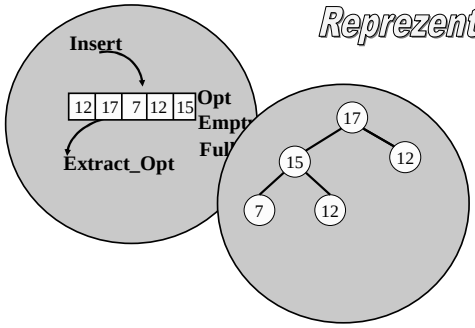
Állítsuk csökkenő sorrendbe a standard bemenetről beolvasott egész számokat, és írjuk ki őket a standard kimenetre!

A feladat megoldásához elkészítünk egy olyan elsőbbségi sor osztálysablont, amely elemi típusa lehet egész szám típus is.
A megvalósításhoz egy tömbben ábrázolt kupac adatszerkezetet használunk.

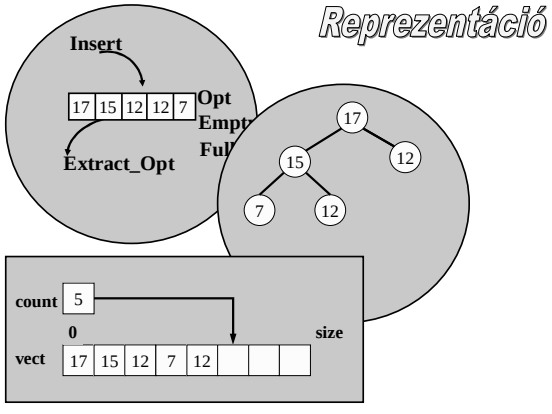
Specifikáció



Reprezentáció



Reprezentáció



Elsőbbségi sor-osztály publikus része

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

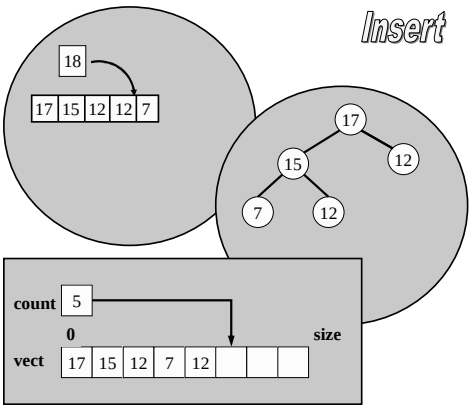
priority_queue.h

Konstruktor

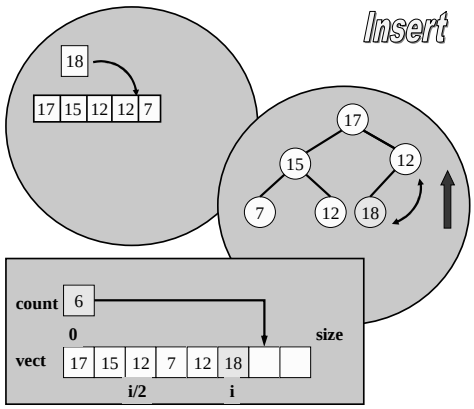
```
template <class Element>
Priority_Queue<Element>::
Priority_Queue(const int s)
{
    size  = s;
    vect  = new Element [size];
    count = 0;
}
```

priority_queue.h

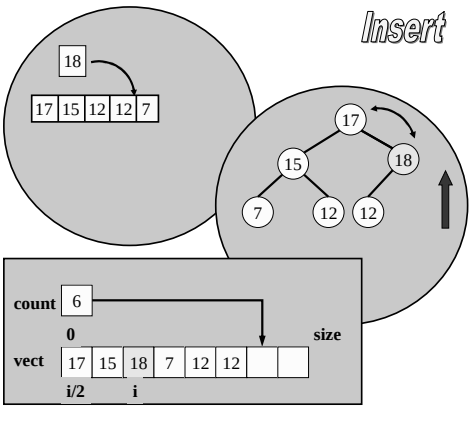
Insert



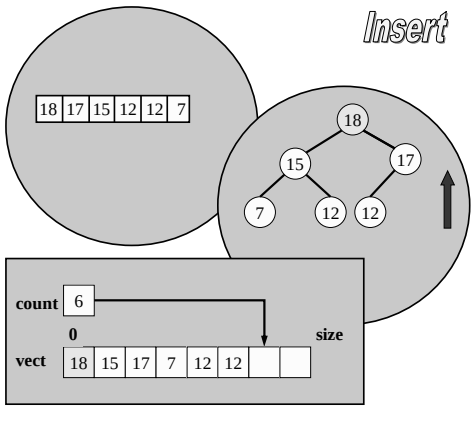
Insert



Insert



Insert



Insert

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


virtual void Put(const Element& e, int i)
                              {vect[i] = e;}
```

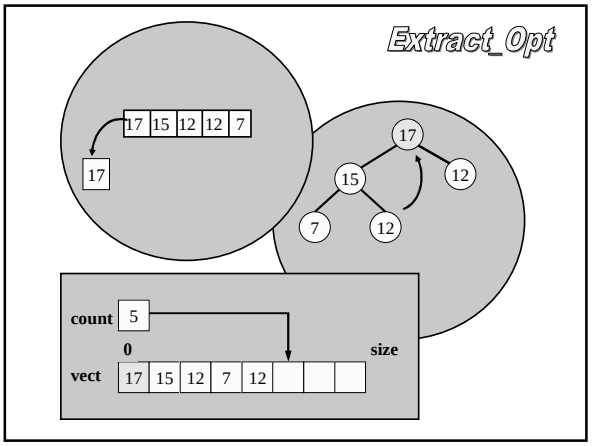
priority_queue.h

Up

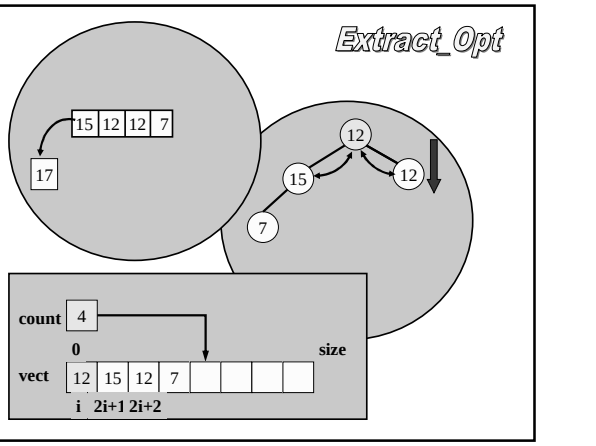
```
template <class Element>
void Priority_Queue<Element>::Up( int i )
{
   Element e = vect[i];
   int j = Parent(i);
   while( j>=0 && e>vect[j] ){
      Put(vect[j],i);
      i = j;
      j = Parent(i);
   }
   Put(e,i);
}


int Parent(const int i){return (i-1)>>1;}
```

priority_queue.h

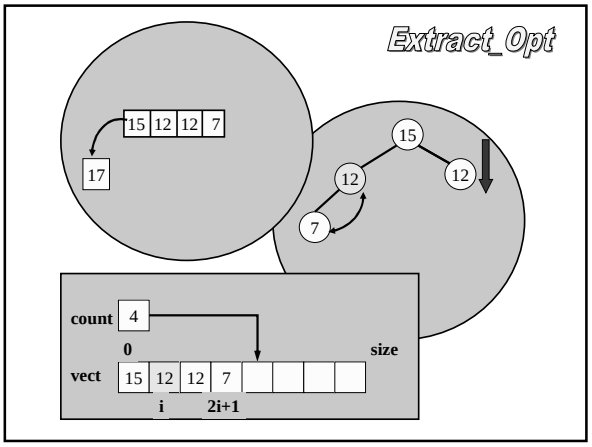
Extract_Opt



Extract_Opt



Extract_Opt



Extract_Opt

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

virtual Element Get(int i){return vect[i];}
```

priority_queue.h

Down

```
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);
}

int Left  (const int i){return (i<<1)+1;}
int Right (const int i){return (i<<1)+2;}
```

priority_queue.h

Elsőbbségi sor-osztály rejtett része

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

  void Up    (int i);
  void Down  (int i);

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

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

priority_queue.h

Főprogram

```
int main()
{
    Priority_Queue<int> h(100);
    int e;
    cin>>e;
    while (e!=9999){
        h.Insert(e);
        cin>>e;
    }
    cout << "Csökkenően rendezve:" << endl;
    while(!h.Empty()){
        cout << h.ExtractOpt() << endl;
    }

  return 0;
}
```

fo.cpp

2. Feladat

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

A korábbi elsőbbségi sor osztálysablont használjuk, csak a rendezési relációt kellene a string típusnál megfordítani.

A string rendezési relációjának megváltoztatása

```
class Reversed_string : public string {
public:
   Reversed_string(){};
   Reversed_string(string s):str(s){};
   bool operator>(const Reversed_string& r)
                        {return str<r.str;}
   bool operator<(const Reversed_string& r)
                        {return str>r.str;}
private:
   string str;
};
```

Rendezési relációt megfordító osztálysablon

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

Rendezési relációt megfordító osztálysablon bázis típusokra

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

Főprogram

```
int main()
{
   Priority_Queue<ClassReversed<string> > h(20);
   ClassReversed<string> e;
   cin>>e;
   while (e!="q"){
      h.Insert(e);
      cin>>e;
   }
   cout << "Növekvően rendezve:" << endl;
   while(!h.Empty()){
      cout << h.ExtractOpt()<< endl;
   }
   cout << endl;

   return 0;
```

fo.cpp

A ">" rendezési reláció osztálya

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

priority_queue.h

A "<" rendezési reláció osztálya

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

priority_queue.h

A rendezési reláció, mint sablonparaméter

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

priority_queue.h

Up

```
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);
}
```

priority_queue.h

Down

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

priority_queue.h

Főprogram

```
Priority_Queue<int> h(100);

Priority_Queue<int, Greater<int> > h(100);

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

fo.cpp

3. Feladat

Készítsünk olyan elsőbbségi sor osztálysablont, amelyben csak egyedi értékeket lehet tárolni, azaz egy érték nem fordulhat elő többször.

Ehhez módosítsuk az Insert metódust, és definiáljuk azt a műveletet is, amely eldönti, hogy egy érték a sorban van-e.

Egyértelmű elsőbbségi sor

```
template <class Element>
class Unique_Priority_Queue :
                  public Priority_Queue<Element> {
public:
   enum Exceptions{EMPTY,FULL,EXISTING};
   Unique_Priority_Queue(const int s):
                     Priority_Queue<Element>(s){}
   void Insert(const Element& e){
        if(In(e)) throw EXISTING;
        Priority_Queue::Insert(e);         }
   bool In    (const Element& e) const
        {return Search(e)>=0;}
protected:
   virtual int Search(const Element& e) const;
};
```

unique_priority_queue.h

Search

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

unique_priority_queue.h