CAPITOLUL I

STL (Standard Template Library)

STL este o bibliotecă foarte utilă pentru programatorii de C++ și este, de asemenea, un ajutor prețios pentru participanții la olimpiadele și concursurile de programare.

Avantajele utilizării STL:

* Micşorează timpul de implementare a algoritmilor pentru că se pot folosi structuri de date care au fost implementate în prealabil.
* Sporește lizibilitatea codului sursă și îmbunătățește performanțele lucrului în echipă
* Implementările sunt, în general, eficiente, atât din punctul de vedere al spațiului de memorie, cât și al timpului de execuție (uneori se face o alegere între eficiența timp și eficiența spațiu)

Dezavantajele utilizării STL:

* Numărul de structuri de date este relativ mic
* Structurile de date pot fi mai lente uneori

**Template** - Un template (șablon, tip de date parametrizat) reprezintă o familie de tipuri sau funcţii, parametrizate cu un tip generic. Un template permite rezolvarea unei probleme generale implementând o singură soluție, care este valabilă pentru toate tipurile de date.

* **Avantaje**:

Reutilizarea codului

Permite implementarea de biblioteci cu scopuri generale

* **Sintaxă** :

template <listaDeParametrii> declarație;

unde listaDeParametrii este o listă de parametrii ai templatului, separați prin virgule.

STL: biblioteca standard a limbajului C++ care oferă cele mai uzuale structuri de date și algoritmi fundamentali pentru utilizarea lor. Aceasta este prima bibliotecă generică a C++ folosită pe scară largă de către programatorii de C++. Ea are în componență 3 categorii de obiecte fundamentale: containere, iteratori și algoritmi.

**Container**: un obiect care conține alte obiecte.

Ele sunt implementate ca și clase șablon (template), acest lucru oferindu-le posibilitatea de a manipula orice tip de dată. Containerele implementează structuri de date foarte uzuale și se ocupă atât de stocarea cât și de manipularea obiectelor pe care le conțin.

**Iterator**: Un iterator STL este o generalizare a noțiunii de pointer și permite accesul secvențial la elementele unui container, în acelasi fel, indiferent de structura de date folositã de container.

Pentru clasele iterator sunt supradefiniți operatorii de indirectare (\* - obținerea valorii elementului aflat pe poziția indicate de iterator la momentul curent), comparație (== și !=), incrementare (++), și decrementare (-- numai pentru iteratori bidirectionali). .

Pentru a putea decide, în cunoștință de cauză, ce fel de container este potrivit pentru rezolvarea unei anumite probleme, este nevoie să cunoaștem și detalii legate de implementarea lor și de modul de utilizare a memoriei.

Clasele container se împart în:

* **Clase secvență**:

vector (vector extensibil)

list (listă dublu înlănțuită)

deque ( coadă cu două capete )

array (C++11)

forward\_list (C++11)

Acestea sunt colecții liniare de date în care accesul se face pe baza poziţiei elementului în cadrul containerului.

Toate clasele secvență implementează adăugarea de elemente la sfărșitul secvenței (metoda push\_back) într-un timp constant.

Pentru obiectele de tip vector sau deque nu se recomandă inserări și ștergeri frecvente (metodele insert și erase). Dacă dorim să folosim secvențe cu conținut dinamic, atunci este mai eficientă secvența list (timp de ordinul O(1) pentru orice poziție din listă).

* **Clase adaptor**, pentru modificarea interfeței unor clase container:

stack (Stivă)

queue (Coadă)

priority\_queue (Coadă cu priorități)

Clasele adaptor sunt construite pe baza claselor secventă. Ele folosesc implicit un anumit container, dar se poate specifica explicit altul. Este important de menționat că aceste containere nu pot fi parcurse cu ajutorul iteratorilor.

* **Containere asociative:**

set (Mulțime sortată fără repetiții de elemente)

multiset (Mulțime sortată cu repetiții de elemente)

map (Mulțime de perechi ordonate <cheie, valoare>, în care o cheie identifică în mod unic o valoare )

multimap (Mulțime de perechi ordonate <cheie, valoare>, în care unei chei i se pot asocia mai multe valori )

Clasele container asociativau ca și caracteristici principale faptul că stocare elementelor se face pe baza unor chei, ordonate după valoarea cheii, iar accesul la elemente este direct (după cheie).

* Containere associative neordonate (C++11) :

unordered\_set

unordered\_multiset

unordered\_map

unordered\_multimap

În STL există si clase șablon (template) care permit manipularea mai uşoară a datelor:

**Clasa pereche** ( pair ).

pair< clasa1, clasa2> p;

pair este o clasă template, definită în <utility>, care are două date membru publice accesbile folosind p.first()și respectiv p.second() (unde p este obiectul declarant mai sus).

clasa1 si clasa2 pot fi definite de utilizator sau pot fi predefinite, acestea putând fi orice tip de container sau orice tip de data din C/C++, cu condiția să aibă definit operatorul “=”.

**Observație**:

O pereche poate conține la rândul ei o pereche, iar operatorul make\_pair(element,element), care returnează perechea este învechit și nu funcționează optim. Se recomandă utilizarea operatorului {}, din standardul C++2011 .

Prin urmare, atribuirea unui pair<int,int> p, se va face in modul următor:

p = {1,2};

Pentru pair<int,pair<pair<int,int> ,int> > p;

p = {1,{ { 1,2 } , 3 } };

**Clasa tuple**

Un tuple este un obiect care stocheză o colecţie de elemente de tipuri diferite. Această clasă are predefinit constructorul (). Pentru a vedea elementul de pe poziţia k din tuple, vom folosi funcţia de tip şablon (template) get<k> ( tuple ). Tuple-urile pot fi concatenate cu metoda tuple\_cut(tuple1,tuple2), funcţia returnând un tuple rezultat al cărui tip poate fi detectat folosind specificatorul auto. Putem accesa elementele dintr-un tuple atât prin intermediul funcţiei template tie(val1,val2,val3 ... valn) = tuple, cât si a funcţiei val1 = get<k>(tuple).

Exemplu:

#include <bits/stdc++.h>

using namespace std;

tuple<int,char,string> t;

int main()

{

///t = tuple<int,char,string>(12,'a',"masina");

t = make\_tuple(12,'a',"masina");

cout << get<0>(t) << '\n';

cout << get<1>(t) << '\n';

cout << get<2>(t) << '\n';

auto rez = tuple\_cat(t,t);

get<3>(rez) = 14;

cout << get<3>(rez) << '\n';

cout << get<4>(rez) << '\n';

get<5>(rez) = "camion";

cout << get<5>(rez) << '\n';

int val1;

char val2;

tie(val1,val2,ignore) = t;

cout << val1 << " " << val2 << '\n';

val1 = get<0>(t);

val2 = get<1>(t);

cout << val1 << " " << val2 << '\n';

return 0;

}

**Clasa string**

Șirurile de caractere pot fi manipulate atât cu ajutorul funcțiilor specifice din biblioteca cstring ( string.h ), cât și cu ajutorul metodelor puse la dispoyiție de clasa string din STL. Acesta are implementate câteva funcții de bază prin supradefinirea unor operatori uzuali: append (concatenare) prin operatorii ‘+’ și ‘+=’,  atribuire prin operatorul ‘=’, comparare prin operatorii relaționali, introducere și extragere în/din stream (citire/scriere), cu ajurorul operatorilor ‘<<’ și ‚>>’.

Observație:

String-urile, ca majoritatea claselor secvenţă din STL, utilizează iteratori bidirecționali.

Cuvântul cheie: auto

“auto” este un cuvânt cheie ce se poate referi la o gamă foarte largăde elemente. Poate detecta atât variabile de tipuri existente deja (int, long long, double, etc), dar \*puterea lui specială\* este aceea de a se mula pe orice tip de dată, containere, clase si funcţii lambda pe care i le alăturăm. Acesta foloseşte principiul şabloanelor (template) pentru determinarea tipului de dată al elementului specificat.

**Exemplu:**

struct elev{

int varsta;

char nume[100];

}f;

auto e = f; ///în acest caz, auto se înlocuieşte cu elev

printf(“varsta lui f este %d”, e.varsta);

Mai mult decât faptul că auto suportă atribuirea oricarui alt tip de date definit de utilizator sau existent, acesta poate să accepte și o funcție de tip lambda. Funcțiile lambda sunt cele mai apropiate de functorii folosiți de către compilator, astfel încât ele necesită cele mai mici ajustări pentru a deveni direct compilabile comparativ cu orice alt tip de funcție / clasă.

Clasele secvenţă:

## Vector

Biblioteca: vector

Avantaje:

Inserarea la final şi ştergerea de la final se face în timp constant.

Permite accesul la o gamă largă de funcţii din STL

Căutarea se face in timp logaritmic

Se folosește alocarea dinamică a memoriei (Atunci când se adaugă noi elemente în vector și se constată depășirea spațiului de memorie alocat, este nevoie să se să se aloce spatiu pentru un nou vector în care să se copieze elementele existente, să se realoce mai multă memorie pentru vectorul initial (spațiul de memorie se dublează) și apoi să se copieze elementele în vectorul initial.)

Dezavantaje:

inserarea şi ştergerea aleatoare au complexitate liniară

Tipul de date vector poate lucra cu orice tip de date definit de utilizator sau existent deja

vector<int> v;

sau

struct catel{

int varsta;

char nume[100];

};

vector<catel> v;

Cele mai des utilizate funcții membru ale clasei vector sunt:

* push\_back( element ) – introduce un element la finalul vectorului
* pop\_back() – elimină elementul de la finalul vectorului
* front() – accesează începutul vectorului
* back() – accesează finalul vectorului
* begin() – pointer catre iteratorul de inceput (coincide cu începutul vectorului )
* end() – pointer către iteratorul de final ( NU coincide cu finalul vectorului, fiind următoarea poziţie după ultimul element din vector )
* size() – o funcţie care returnează dimensiunea vectorului

Exemple:

for(int i = 0 ; i < A.size() ; i ++)

cout << A[i] << ' ';

for(auto x : A)

cout << x << " ";;

for(int & x : A)

x += 2;

for(vector<int>::iterator I = A.begin() ; I < A.end() ; I ++)

cout << \*I << " ";

for(vector<int>::reverse\_iterator I = A.rbegin() ; I < A.rend() ; I ++)

cout << \*I << " ";

for(auto I = A.begin() ; I < A.end() ; I ++)

cout << \*I << " ";

Lista

Avantaje:

inserarea şi ştergerea aleatoare sunt efectuate în timp constant

memoria se alocă element cu element făra a aloca în exces

Dezavantaje:

cautarea unui element are o complexitate de ordinul numarului de elemente

este mai înceată decât un vector

Bibliotecă particulară:

#include <list>

Spre deosebire de clasa vector<>, clasa list<> își va aloca memoria în mod diferit. Pe când în vector, se alocă memoria de forma 2^p, astfel încât capacitatea vectorului să fie mai mare sau egală cu numărul de elemente continute, lista alocă memorie doar cât îi este necesară. De observat faptul că lista aloca mai multă memorie decât vectorul, aceasta având nevoie de pointeri auxiliari. O observație foarte importantă este aceea că lista suportă căutare liniară şi inserare sau ştergere în timp constant.

Exemplu de implementare introsort  in N^2

#include <bits/stdc++.h>

using namespace std;

vector<int> v = {1,23,4,5,1,2,3,12,34};

list<int> l;

int main()

{

   for(auto it : v){

       auto pz = lower\_bound(l.begin(),l.end(),it);

       l.insert(pz,it); /// funcția membră a clasei list, insert, primeste parametrul locul și elementul pe care trebuie să il insereze

   }

   for(auto it : l)

       printf("%d ",it);

   return 0;

}

Observăm faptul că deși lower\_bound încearcă să caute binar rezultatul, avansarea iteratorilor prin listă va fi liniară, prin urmare aceștia trebuie să avanseze unitate cu unitate către mijloacele selectate de căutarea binară, de unde se pierde timpul logaritmic, devenind liniar.

Deque

Avantaje:

inserarea în timp constant atât in faţă cât şi în spate

are o structura asemănătoare vectorului

alocă memoria element cu element făra a aloca în exces

Dezavantaje:

este mai încet decât vectorul la inserarile in mijloc si la accesarea elementelor

Bibliotecă aferentă:

#include <deque>

Deque, este o coadă cu două capete, care suportă inserarea de elemente în ambele capete, utilizând push\_back() si push\_front() și scoaterea de elemente din ambele capete, folosind pop\_back() respectiv pop\_front(). Elementele din capetele cozii se accesează utilizând metodele front()și back().

Există implementat şi operatorul []astfel încat elementele cozii cu două capete pot fi accesate identic cu cele ale unui vector.

Operaţiile de inserare şi de ştergere, deq.insert(iterator,numar,valoare) respectiv deq.erase(iterator\_start, iterator\_end) sunt liniare în distanţa până la cel mai apropiat capăt.

Forward\_list

Bibliotecă aferentă:

#include <forward\_list>

Aceasta reprezintă o listă simplu înlăntuită si are toate caracteristicile listei <list> . Aceasta poate fi parcursă unidirecţional (înainte). Memoria folosită este mai puţină deoarece nu se mai reţin adresele elementelor precedente.

Array

Avantaje:

se comporta asemanator vectorului

i se pot aplica algoritmii din STL

Bibliotecă aferentă:

#include <array>

Acesta reprezintă un container cu dimensiune fixa. Nu se reţine memorie suplimentară, acesta memorând doar elementele în ordinea specificată de inserare.

#include <bits/stdc++.h>

using namespace std;

array <int,5> a;

int main()

{

for(int i = 0; i < a.size(); ++i)

cin >> a[i];

for(auto it : a)

cout << it << " ";

return 0;

}

Algoritm de căutare binară pe array:

#include <bits/stdc++.h>

using namespace std;

array <int,5> a;

int main()

{

for(int i = 0; i < a.size(); ++i)

a[i] = i;

auto it = lower\_bound(a.begin(),a.end(),3);

cout << it - a.begin();

return 0;

}

Metode generale pentru vectori, stringuri şi liste:

v.push\_back(element); /// pune un element la capătul vectorului, crescând dimensiunea cu o unitate și alocând până la de doua ori mai multă memorie decât dimensiunea acestuia.

v.pop\_back(); /// scoate ultimul element din vector, fără să redimensioneze vectorul.

v.shrink\_to\_fit(); /// redimensionează capacitatea vectorului astfel încât el să ocupe tot atâta memorie cât este dimensiunea elementelor stocate în el.

v.resize(dimensiune); /// măreste/micsorează vectorul astfel încât dimensiunea sa să devină cea specificată prin parametru.

v.front() si v.back() accesează primul, respectiv ultimul element al vectorului.

v.begin() si v.end() returnează un pointer de tip iterator către începutul și sfârsitul vectorului. Atentie, sfârșitul unui vector nu coincide cu adresa de memorie a ultimului element, pe când începutul coincide.

vector<element>::iterator it este un iterator bidirectional, care prin operatii de tip it + k si it + (- k),  se deplasează în ambele directii, în interiorul vectorului.

stg  = v.begin()+x;

dr = v.begin()+y;

v.erase(stg,dr); /// șterge elementele din vector de pe pozitiile de la stanga la dreapta, stg <= dr;

v.clear(); /// golește vectorul

v.insert(pozitie,element); /// va insera în v la adresa de memorie “pozitie”  un element specificat.

v.empty(); va returna daca vectorul este gol (1) sau nu (0).

v.size(); va returna dimensiunea vectorului

v.find(element) va returna un iterator către elementul căutat în cazul în care este găsit, sau -1 in caz contrar.

v.reverse(v.begin(),v.end()); /// inversează toate elementele din intervalul specificat, spre exemplu, tot vectorul.

Observatie: Este posibilă alocarea de memorie a unui vector utilizând operatorii = si {}:

vector<int> v;

v = {1,2,3,4};

String-ul este de fapt un vector de caractere, comportându-se asemănător și având aceleași functii membre ale clasei.

În plus, funcţia string acceptă urmatoarele functii:

string s,s2;

s.substr(s2), va returna tot stringul începând de la prima aparitie a lui s2 în s.

Relatia dintre string-urile din string.h si cele din biblioteca string se face în felul următor:

string s;

char d[100];

s = string(d);

strcpy(d,s.c\_str());

Funcţia .c\_str(); returnează un string de tipul tablou de caractere (char c[]), care este propriu pentru a fi utilizat de scanf si printf.

Într-o listă, apar următoarele functii care nu sunt membre nici ale vectorilor nici ale stringurilor:

list<element> L,L2;

L.push\_front();

L.pop\_front();

L.splice(L.begin(),L2); ///iterator și lista=> va insera lista L2 intre iteratorul specificat și următorul element de după iterator, membru al listei L.

## Clase adaptor

Coada

Bibliotecă patriculară:

#include <queue>

Coada, asemănătoare listei, este alocată dinamic element cu element, punctul de acces fiind doar capul cozii.  Se inserează elemente utilizând .push()și se elimină utilizând .pop(). Elementul curent din coadă se acceseaz ă prin .front(). Metoda .empty() returnează 1 dacă coada este goală sau 0 în caz contrar.

Stiva

Bibliotecă patriculară:

#include <stack>

Stiva, asemănătoare listei, este alocată dinamic element cu element, punctul de acces fiind doar vârful stivei. Se introduce in stivă prin .push(), se elimină din stivă cu .pop(), iar elementul din vârf se accesează cu .top(). Metoda .empty() returnează 1 dacă stiva este goală sau 0 în caz contrar.

Priority Queue

Bibliotecă patriculară:

#include <queue>

Coada de priorităti este un maxheap, care poate fi ajustat din punct de vedere al tipului de date cu care lucrează, containerului în care va pune acel tip de date (de obicei vector), și al funcției de comparare. Se inserează element cu .push()și se elimină cu .pop(), iar varful se accesează asemenea stivei, utilizând .top().Metoda .empty() returnează 1 dacă coada este goală sau 0 în caz contrar.

priority\_queue< element, vector<element> , cmp > Coada;

Aceasta menține structura de heap utilizând legea de comparație cmp, care poate fi una predefinită, de tip greater<element>, sau less<element> dacă elementul este predefinit de STL ( de exemplu pereche ).

Funcţia cmp ne-o putem crea identică cu cea de la sortarea unui vector, dar cel mai eficient este utilizând un lambda closure utilizând decltype. Aceasta declarare, opreste compilatorul să șteargă lambda-ul după prima compilare.

Mai jos avem exemple de definiri ale operatorului de comparare.

#include <bits/stdc++.h>

using namespace std;

/\*\*

auto comp = [](const pair<int,int> &A, const pair<int,int> &B)->const bool{ return A > B; };

priority\_queue<pair<int,int>, vector<pair<int,int> > , decltype(comp) > Q(comp);

\*/

/// se observa faptul că la coada de prioritati trebuie pus pe dos operatorul

/// observatie generala pentru toate implementarile de priority queue

/\*\*

class cmp{

public:

   bool operator () ( const pair<int,int> & A, const pair<int,int> & B) const{

       return A > B;

   }

};

priority\_queue<pair<int,int>,vector<pair<int,int> >, cmp > Q;

\*/

priority\_queue<pair<int,int>,vector<pair<int,int> >,greater<pair<int,int> > > Q;

int main()

{

   Q.push({2,3});

   Q.push({1,4});

   Q.push({5,9});

   Q.push({5,-9});

   while(!Q.empty()){

       printf("%d %d\n",Q.top().first, Q.top().second);

       Q.pop();

   }

   return 0;

}

## Containere asociative

Set

Bibliotecă particulară:

#include <set>

Tipul de date set, are în spate un un arbore roșu negru (binary search tree echilibrat), care are capacitatea de a stoca în frunze elementele pe care le inserăm. Prin urmare, îl putem folosi ca pe un heap la care nu putem accesa decât valoarea dintr-un nod fără a-i ști tatăl sau fii.

set-ul suportă următoarele operatii:

set<element> S;

S.insert(element); /// va returna o pereche formată din un iterator bidirectional ce indică locul unde a fost inserat elementul, si un bool care indică dacă elementul există deja în set sau nu. Atentie, în set nu există elemente duplicate !!

S.erase(pozitie\_iterator), va șterge elementul aflat la poziţia indicată printr-un iterator.

find(S.begin(),S.end(),element) este o căutare în timp logaritmic într-un arbore de căutare binar. Funcţia returnează un iterator către poziţia din arbore unde se află nodul cu valoarea cautată.

Observatie: Funcţia find va rula la fel de eficient ca funcţia lower\_bound, astfel încât arborele rosu negru nu este format din iteratori random, iar pentru a avansa, va trebui sa meargă pozitie cu pozitie ( asemănător listei ).

S.begin(); -> returnează un iterator către elementul de valoare extremă, relativă la metoda de comparare. Se poate parcurge setul, pornind de la S.begin() până la S.end() in O(Nr\_elemente).

Se poate crea o metodă de comparare pentru set, asemănătoare celei pentru coadă:

#include <bits/stdc++.h>

using namespace std;

auto cmp = [](const int &x1, const int &x2)->bool{

   return x1 > x2;

};

set<int,decltype(cmp)> S(cmp);

int main()

{

   S.insert(1);

   S.insert(12);

   S.insert(21);

   S.insert(31);

   return 0;

}

Observaţie: multisetul este tot un set dar care acceptă chei duplicate.

Map

Bibliotecă patriculară:

#include <map>

Tipul de dată Map are în spate o structură de arbore roșu negru, identică cu cea a setului, astfel încât el este arbore binar de căutare, și este echilibrat (are adâncimea maximă logN, unde N e numărul de noduri). Map-ul este o functie capabilă de hashing. Putem declara map-uri care să atribuie unor elemente sau containeri orice element definit anterior sau existent, inclusiv alti containeri:

priority\_queue<pair<int,int>,vector<pair<int,int> >,greater<pair<int,int> > > Q;

auto comp = [](const priority\_queue<pair<int,int>,vector<pair<int,int> >,greater<pair<int,int> > > &Q1,

              const priority\_queue<pair<int,int>,vector<pair<int,int> >,greater<pair<int,int> > > &Q2)->const bool{

       return Q1.top() < Q2.top();

};

map<priority\_queue<pair<int,int>,vector<pair<int,int>>,greater<pair<int,int>> >,vector<int> ,decltype(comp)> M(comp);

vector<int> v;

Observatie: Având declarate aceste containere, este posibilă următoarea operatie:

Atribuirea cozii Q, vectorul v

M[Q] = v;

Observaţie: multimap-ul este tot un map doar că pot exista chei duplicate (precum la multiset)

-Unordered Map

Unordered map este funcţia de hashing perfectă pe care o putem folosi atunci când vrem să codificam containere sau tipuri de dată. Ea suportă aceleasi operatii ca map-ul din STL doar că nu are structura unui arbore roșu negru, ci de listă.

unordered\_map<string,int> M;

M[“mama”] = 1;

cout << M[“mama”]; /// afiseaza 1

Aceste operatii se realizează in O(1)\*constanta\_hashing amortizat, pe cand în map-ul simplu (cu ordine) operatiile de accesare, inserare si ștergere se desfășoară in O(logN) \* constanta\_hashing.

Map-ul poate fi parcurs in O(N) la fel ca un set cu iteratori.

for(auto it : M)

cout << it.first << “  “ << it.second

**Algoritm**: exista o arie destul de largă de algoritmi prezenti in STL, acestia fiind de mai multe tipuri, cei mai importanti fiind cei de căutare si de modificare. Algoritmi de căutare sunt cei liniari, precum:

find(v.begin(),v.end(),element);

lower\_bound(v.begin(),v.end(),element);

upper\_bound(v.begin(),v.end(),element);

binary\_search(v.begin(),v.end(),element);

nth\_element(v.begin(),v.begin() + n,v.end());

Funcţia swap

Membră a bibliotecii <algorithm>, funcţia swap primește ca parametri două tipuri de dată existente deja (variabile simple, structuri ), dar NU tablouri, și le interschimbă. Această functie are capacitatea de a interschimba containeri din STL care au rol de tablou, interschimbând doar adresele de memorie pentru început si sfârșit (de exemplu swap(vector1,vector2), unde vectorii sunt containeri din STL ).

Exemplu:

long long a, b;

struct elev{

   int varsta;

   char nume[10];

}e1,e2;

vector<pair<int,int> > v1,v2;

int main()

{

   swap(a,b);

   swap(e1,e2);

   swap(v1,v2);

   return 0;

}

**Functorii** se definesc în felul următor:

[variabilele din scope-ul curent/globale ce vor fi modificate, specificate prin referinta] ( variabilele membre ale functiei, folosite in mod util de apeluri ) -> tipul de date returnat, posibil autodetectabil{  … continut functie … };

Exemplul pentru un comparator, utilizand o functie lambda:

[] (const int &a, const int &b) -> const bool { return a > b; };

Această functie lambda pentru a putea fi folosită, trebuie mai întâi salvată într-un auto, ce suportă mutatia.

auto cmp = [] (const int &a, const int &b) -> const bool { return a > b; };

Vectorii suportă operatii cu functii membre bibliotecii <algorithm>. Spre exemplu:

lower\_bound(vector.begin(),vector.end(), element);

va returna un iterator către cea mai de jos apariţie a elementului căutat în cazul în care acesta există, altfel va returna un iterator către primul element mai mare decât cel căutat O(logN).

upper\_bound(vector.begin(),vector.end(),element);

va returna un iterator către primul element mai mare decât cel căutat. O(logN)

find(vector.begin(),vector.end(),element);

va returna un iterator către poziţia unde se află elementul element, în caz că el nu se găseste în vector va returna vector.end().

unique(vector.begin(),vector.end());

va returna un interator către poziţia nouă a sfârsitului vectorului după eliminarea tuturor duplicatelor. Atentie, această functie ar trebui folosită după aplicarea unei sortări asupra vectorului.

sort(vector.begin(),vector.end());

va sorta vectorul crescător după un criteriu existent sau definit de utilizator. Criteriul definit de utilizator va fi declarat sub formă de functie, sau clasă afiliată vectorului, functor sau direct functor direct compilabil de tip lambda.

Exemple:

Sortăm un vector de int-uri crescător

vector<int> v;

sort(v.begin(),v.end());

Sortarea descrescătoare: ( in ordinea crescătoare a eficientei implementării )

bool cmp(int x1,int x2){

return x1 > x2;

}

sort(v.begin(),v.end(),cmp);

sau

struct cmp{

bool operator()( int x1,int x2)

{

return x1 > x2;

}

};

sort(v.begin(),v.end(),cmp());

sau

class cmp{

public:

bool operator()(const int &x1, const int &x2) const{

return x1 > x2;

}

};

sort(v.begin(),v.end(),cmp());

Sau, cel mai eficient mod, direct in limbaj compilabil:

auto cmp=[](const int&a,const int&b)-> bool{return a>b;};

sort(v.begin(),v.end(),cmp);

Funcţia de interclasare a doi vectori sortati crescător:

merge(v1.begin(),v1.end(),v2.begin(),v2.end(),rez);

Funcţia cere specificarea celor doi vectori ce trebuie interclasati și containerul în care se va obtine rezultatul.

Există functii care modifică structura unui vector astfel încât elementele sale să apartină unui heap:

make\_heap(v.begin(),v.end());

sort\_heap(v.begin(),v.end());

v.push\_back();push\_heap(v.begin(),v.end());

pop\_heap(v.begin(),v.end());v.pop\_back();

Un alt algoritm folositor este next\_permutation(v.begin(),v.end()); această funcţie permută șirul v până când toate elementele sale sunt ordonate descrescător. (Se recomandă o sortare crescătoare a elementelor șirului înainte de a aplica funcţia). Observatie: funcţia rulează corect și pe permutări cu repetitie! Aceasta nu va repeta nicio permutare, pentru aab, va genera doar 3 elemente: aab, aba, baa.

Structuri Arborescente

Arbore binar de căutare

Arborele binar de căutare este un arbore binar clasic (fiecare nod are 1,2 sau niciun fiu) care are următoarea proprietate: Valoarea nodului curent este mai mare decât cea a fiului stâng și mai mică decât cea a fiului drept. Prin urmare, acest arbore suportă inserări, ștergeri si căutări:

Pentru inserare, vom folosi următorul algoritm:

Cât timp nodul curent este deja alocat, dacă valoarea nodului curent este mai mare decât cea a nodului pe care dorim să îl inserăm, mergem în fiul stâng, altfel mergem în fiul drept. Prin această deplasare în unul din fii, reducem problema la cea initială, de a introduce un nod nou într-un subarbore. Prin urmare, se poate utiliza o functie recursivă care va înainta recursiv în arbore până la locul inserării.

Pentru ștergere, vom folosi următorul algoritm:

Cât timp nu am ajuns în nodul căutat, înaintăm prin arbore tot cu o functie recursivă. Când ajungem în nodul pe care vrem sa îl ștergem, dacă fiul din dreapta este gol, atunci interschimbăm nodul curent cu cel din dreapta (astfel nu se strică proprietatea că fiul stang e mai mic decât nodul curent, și aplicăm aceeași functie recursivă pe subarborele drept. Când nodul dorit va deveni frunză, acesta poate fi eliminat cu succes.

Pentru căutare, vom folosi algoritmul următor:

Cât timp nu am ajuns în nodul căutat, dacă valoarea din nodul curent este mai mare decât cea căutată, mergem în stânga, altfel mergem în dreapta. Dacă am ajuns într-o frunză, ne oprim si returnăm 0 (faptul că nu am găsit nodul), altfel, dacă am determinat nodul putem să returnăm adresa sa de memorie sau valoarea 1.

Structura de Heap

Heap-ul este un arbore binar cu următoarea proprietate: valoarea dintr-un nod curent este strict mai mare decât cea a fiilor (se observă usor faptul că maximul se află în radăcină).

Treap

Utilizând o combinatie dintre aceste două structuri, putem defini o nouă structură arborescentă, cu denumirea de Treap. Acesta este un arbore binar de căutare, echilibrat, bazându-se pe proprietatea de Heap.

Fiecare nod are următoarele caracteristici: Un fiu stâng, un fiu drept, o informatie și o prioritate.

Cei doi fii sunt la rândul lor doi subarbori de tip Treap.

Informatia este utilizată astfel încât Treap-ul să respecte proprietătile arborelui binar de căutare.

Prioritatea este utilizată ca element comparativ pentru a mentine o structură de Heap. Pentru a putea echilibra arborele, ne folosim de un algoritm probabilistic, utilizând funcţia rând(). Această functie se foloseste de catre ceasul procesorului astfel încât să genereze valori aleatoare într-un mod uniform. Această uniformitate conferă proprietatea de echilibru al arborelui.

Pentru Inserare, ne vom baza pe inserarea în arborele binar de căutare, astfel încât, pentru a insera un nod nou cu o prioritate nouă, vom căuta locul din Treap unde acesta ar trebui sa ajungă utilizând o functie recursivă. La pasul de întoarcere in recursivitate, trebuie să verificăm dacă structura de Heap este mentinută. In cazul în care nu este mentinută structura de heap, vom folosi rotatii.

O rotatie este definită în felul următor: Fie un nod T, care are fs și fd, fiul stang și fiul drept. O rotatie spre stânga, înseamnă ca fd devine noua rădăcină a subarborelui, fiul stâng al lui fd devine fiul drept al nodului T, iar fiul stâng al lui fd devine nodul T. O rotatie spre dreapta înseamnă că fs devine noua rădăcină a subarborelui, fiul drept al lui fs devine fiul stâng al nodului T iar fiul drept al lui fs devine nodul T.

Pentru a echilibra arborele, după finalizarea inserării, la întoarcerea din recursivitate, vom mentine structura de maxheap sau minheap, rotind arborele ca atare, astfel încât valoarea maximă /minimă vine spre radacină.

Pentru ștergere, vom roti arborele astfel încat să coboram nodul pe care vrem să îl ștergem pană intr-o frunză. Ajuns în frunză, acesta poate fi eliminat.

Operatiile de căutare, vor fi identice cu cele aplicate pe un arbore binar de căutare.

Operatia de split, introduce un nod cu o anumită valoare, și cu prioritate maximă/minimă corespunzătoare astfel încât ea sa devină radăcină. După inserarea acestui nod, fiul său stâng va reprezenta un Treap cu toate nodurile mai mici decât valoarea respectivă, si fiul drept va reprezenta un Treap cu toate nodurile mai mari decât valoarea respectivă inserată. Putem deci să îl separăm in două subtreapuri, Trstang și Trdrept.

Operatia de Join, unește două treap-uri în unul singur în felul următor: creăm un nou nod, care să aibă drept valoare infinit, si prioritate maximă/minimă în functie de tipul heapului. Acest nod nou, va avea ca fiu drept și fiu stâng cele două Treap-uri in ordinea corectă pentru a respecta proprietatea de arbore binar de căutare. Apelând funcţia de ștergere pentru rădăcină, vom obtine treap-ul ce reprezintă rezultatul uniunii celor două treapuri.

Cod sursa:

#include <bits/stdc++.h>

#define INF 0x3f3f3f3f

using namespace std;

class Treap{

public:

    int inf;

    int priority;

    Treap \*st,\*dr;

    Treap(){

        st = dr = NULL;

        inf = priority = 0;

    }

    Treap(const int &inf,const int &prio, Treap \*st, Treap \*dr){

        this->inf = inf;

        this->priority = prio;

        this->st = st;

        this->dr = dr;

    }

};

Treap \*Null, \*Root;

void Init(){

    srand((unsigned)time(0));

    Null = new Treap(0,-INF,NULL,NULL);

    Root = Null;

}

void Rotate\_left(Treap \*&T)

{

    Treap \*aux = T->dr;

    T->dr = aux->st;

    aux->st = T;

    T = aux;

}

void Rotate\_right(Treap \*&T)

{

    Treap \*aux = T->st;

    T->st = aux->dr;

    aux->dr = T;

    T = aux;

}

void Equilibrum(Treap \*&T)

{

    if(T->priority < T->st->priority)

            Rotate\_right(T);

    else

        if(T->priority < T->dr->priority)

            Rotate\_left(T);

}

void Insert(Treap \*&T,const int &info,const int &prio){

    if(T == Null){

        T = new Treap(info,prio,Null,Null);

        return;

    }

    if(T->inf > info)

        Insert(T->st,info,prio);

    else

        Insert(T->dr,info,prio);

    Equilibrum(T);

}

Treap\* Find (Treap \*&T, int value)

{

    if(T == Null || T->inf == value)

        return T;

    if(T->inf > value)

        return Find(T->st,value);

    return Find(T->dr,value);

}

void Delete(Treap \*&T, const int &value)

{

    if(T->inf > value)

        Delete(T->st,value);

    else

        if(T->inf < value)

            Delete(T->dr,value);

        else

        {

            if(T->st == Null && T->dr == Null)

            {

                Treap \*aux = T;

                T = Null;

                delete aux;

                return;

            }

            if(T->st->priority < T->dr->priority){

                Rotate\_left(T);

                Delete(T->st,value);

            }else{

                    Rotate\_right(T);

                    Delete(T->dr,value);

                }

        }

}

void Split(Treap \*&T,Treap \*&T1,Treap \*&T2,int value)

{

    Insert(T,value,2\*INF);

    T1 = T->st;

    T2 = T->dr;

    Treap \*aux;

    aux = T;

    T = Null;

    delete aux;

}

void Join(Treap \*&T,Treap \*&T1, Treap \*&T2)

{

    T = new Treap(2\*INF,2\*INF,T1,T2);

    Delete(T,2\*INF);

    Treap \*aux1,\*aux2;

    T1 = T2 = Null;

}

void SRD(Treap \*&T){

    if(T == Null)

        return;

    SRD(T->st);

    printf("%d ",T->inf);

    SRD(T->dr);

}

int main()

{

    Init();

    freopen("algsort.in","r",stdin);

    freopen("algsort.out","w",stdout);

    int N,x,pr;

    scanf("%d",&N);

    for(int i = 1; i <= N; ++i)

    {

        scanf("%d",&x);

        pr = rand();

        Insert(Root,x,pr);

    }

    SRD(Root);

    return 0;

}