Curs 1

In rezolvarea diverselor probleme sunt necesare sa fie stocate date. Cel mai simplu mod de a stoca date sunt sa fie stocate sub forma de variabile. In general lucrurile sunt mai complexe. Spre exemplu, daca avem 1000 de elemente de tip int, nu o sa definim variabilele

int v1, v2, … , v1000;

Vom folosi un vector:

int v[1000];

In acest caz am stiut de la inceput numarul de variabile necesare, sunt cazuri in care numarul de variabile necesar se va afla la runtime si sa zicem ca va fi n. Atunci vom tine acele date tot intr-un vector, pe care il vom aloca la runtime cand vom afla cat este n.

int \* v = NULL;

…..

//aflam cumva cat este n

v = (int \*)malloc(n\*sizeof(int));

Bun, ce se intampla in cazul in care si n variaza – de exemplu avem o companie in care trebuie sa tinem minte varsa angajatilor. Initial sunt 500, compania creste, ajung 700 si asa mai departe. Abordarea anterioara nu mai este suficienta. O varianta ar fi ca de fiecare data cand ajungem la limita maxima a vectorului alocat sa il realocam dublu si in partea inferioara sa copiem vectorul precedent. Acest lucru s-ar putea dovedi inefficient din doua motive – ar exista timpi destul de mari in cazul operatiilor care ar duce la dublarea vectorului si de asemenea daca am avea sa zicem 1 milion de intrari si am dubla vectorul si apoi am folosi doar 10 000 restul de 990 000 de intrari ar fi nefolosite si memoria respective ar fi folosita ineficienta.

In astfel de situatii o alta abordare e cea de structuri inlantuite(liste, cozi, stive).

typedef struct \_list

{

\_list \*next;

int info;

}list;

Ori de cate ori vom avea nevoie de un element nou il vom aloca si il vom adauga la sfarsitul listei. Se observa ca fata de structura de vector, structura de lista aduce un overhead de memorie de un pointer suplimentar care pointeaza catre elementul urmator din inlantuire. De asemenea memoria nu e contigua, cee ace in cazul vectorului este si reprezinta un plus pentru ca in cazul accesarilor foarte dese foarte probabil elementele fiind in aceleasi pagini de memorie vor fi obtinute mult mai rapid.

Sa analizam cateva operatii care se pot face cu vectori si lista:

* inserare
* cautare
* stergere

Exemple se gasesc in Curs 1\Exemple.

Putem observa asa:

Inserarea

* in cazul inserarii in cazul in care stim vecinul unde vrem sa facem insertia in lista insertia se face in O(1) in vector in O(n) pentru ca e necesara si copierea elementelor care ar urma dupa acel element
* daca inserarea implica si cautarea vecinului din lista unde sa se faca inserarea atunci si inserarea in lista costa O(n)

Cautarea:

* cautarea unui element atat in vector cat si in lista se face in O(n) – atata vreme cat vectorul nu e sortat.

Stergerea

* stergerea unui element din vector va costa O(n), in timp ce stergerea unui element din lista va costa O(1) atata vreme cat stim vecinul de langa care dorim sa stergem sau in cazul in care lista e dublu inlantuita, altfel daca avem si cautarea elementului de sters si in cazul listei stergerea se va face in O(n).

O lista dublu inlantuita arata astfel:

typedef struct \_\_list

{

int elem;

\_\_list \*next;

\_\_list \*prev;

}list;

In Curs1\dublu\_linked\_list\_stergere e un exemplu care demonstreaza folosirea unei liste dublu inlantuite. Se observa avantajul de a putea sterge direct un element pentru ca avem vecinii si putem reface lagaturile in mod direct, dar se observa si dezavantajul ca pentru fiecare element o sa avem nevoie de doi pointeri suplimentari pentru vecini.

**Stive si cozi**

In exemplele de pana acum insertiile facute in liste au fost facute in coada listei. Stiva, o forma de lista are managementul insertiilor si extragerilor din lista un pic diferit. In cazul stivei insertia se face in fata si scoaterea se face tot din fata.

Exemplu – daca avem de inserat elementele 1,2,3 si apoi sa le scoatem vom avea asa:

Insert 1

Insert 2

Insert 3

Pop – va scoate 3

Pop – va scoate 2

Pop – va scoate 1

O prima aplicatie a unei stive poate fi observata din acest exemplu – introducerea unui sir intr-o stiva si scoaterea lui va produce inversarea sirului.

Acest lucru este demonstrat in:

Curs1\Stiva-inserarea

Acelasi exemplu este demonstrate si in

Curs1\Stiva-inserarea-std

Unde exemplu cu stiva este implementat cu o stiva gata implementata din biblioteca std.

Un al 3-lea exemplu pe tema asta este dat in :

Curs1\Stiva-inserarea-stiva-apel

Unde e efectiv folosita stiva apelurilor de functii.

Cozile sunt un element pe care il intalnim si in viata de zi cu zi – daca ne inchipuim de exemplu modul in care sunt procesate comenzile de la amazon sau sunt preluate apelurile telefonice de la o linie de suport client. Un exemplu strict legat de software ar fi un server – in care avem un thread principal care accepta conexiuni si mai multe threaduri worker care preiau acele conexiuni noi.

Exemple de coada:

Curs1\Coada

Curs1\Coada-std

**Arbori**

Sa ne uitam din nou pe operatiile si timpii de raspuns pentru operatiile cu liste si vectori:

* cautarea – in cazul vectorului si a listei se face in O(n)
* inserarea – in cazul listei se face in O(1), iar in cazul vectorului in O(1) daca e sufficient de mare, daca nu in O(n), pentru ca implica realocarea si repozitionarea vectorului
* stergerea – in cazul vectorului si al listei se face in O(n). In cazul listei daca stim elementul se face in O(1), daca nu il stim, se face in O(n).

Exista vreo structura care sa aibe implementarea acestor operatii mai eficient ? Raspunsul este aproape da. Este vorba de arborii binari. Timpii operatiilor respective sunt:

* cautarea – O(lg n);
* inserarea – O(lg n);
* stergerea – O(lg n);

Observam ca doar inserarea este mai putin eficienta, dar restul operatiilor sunt mai eficiente. Un arbore e o structura inlantuita de date care are legatura catre parinte si legaturi catre unul sau mai multi copii. Arborele binar are maxim 2 copii.

Un arbore binar are urmatoarea proprietate – daca sunt in nodul y, atunci orice copil aflat la stanga nodului y va avea informatia stocata mai mica decat valoarea informatiei stocate in y si de asemenea orice copil aflat la dreapta nodului y va avea informatia stocata mai mare decat valoarea informatiei stocate in y.

**Cautarea.**

Pentru a gasi un element cu valoarea val in arbore se procedeaza astfel:

1. Nod\_curent = root
2. Daca nod\_curent->info == val atunci return nod\_curent
3. Altfel:
   1. Daca nod\_curent->info < val
      1. Nod\_curent = nod\_curent->left
   2. Altfel:
      1. Nod\_curent = nod\_curent->right
4. Daca nod\_curent == NULL return NULL
5. Altfel inapoi la pasul 2

Se observa ca timpul de cautare e egal in cel mai rau caz cu inaltimea maxima a arborelui. Daca arborele este unul echilibrat, atunci inaltimea va fi logn, daca nu e echilibrat in cazul cel mai rau o sa ajunga la n.

**Inserarea.**

Orice element nou se va insera ca si frrunza in arbore. Procesul de insertie este descris astfel :

1. New\_elem->left = NULL
2. New\_elem->right = NULL;
3. New\_elem->parent = NULL
4. New\_elem->info = info.
5. Curent\_nod = root;
6. Y = NULL;
7. Atata vreme cat curent\_nod nu e NULL
   1. Y = current\_nod
   2. Daca info < x->info
      1. X = x->left
   3. Altfel
      1. X = x->right
8. Daca y->info > info
   1. Y->left = new\_elem
9. Altfel
   1. Y->right = new\_elem;
10. New\_elem->parent = Y;

Dupa cum se poate observa si inserarea are un timp maxim egal cu inaltimea maxima a arborelui. Daca arborele e echilibrat atunci timpul de insertie va fi O(logn).

**Stergerea**

Stergerea e o operatie mai complicata, pentru ca e necesar sa fie pastrata si structura arborelui in urma stergerii, adica orice element sa aiba in stanga sa elemente mai mici si in dreapa doar elemente mai mari.

Sunt mai multe cazuri in care se poate afla un nod.

1. Are cel putin un copil NULL
2. Are ambii copii not NULL.

Daca are cel putin un copil NULL atunci daca nu are nici un copil se sterge nodul direct si legatura parintelui va fi actualizata la NULL.

Daca are ambii copii not NULL, pentru a pastra structura de arbore binar va trebui sa gasim succesorul nodului ce se doreste sters pentru a inlocui nodul ce dorim sa fie sters. Astfel succesorul nodului de sters va avea in stanga, arborele stanga al nodului de sters si in dreapta va fi restul arborelui din dreapta al nodului de sters. Cum succesorul este mai mare ca nodul de sters tot ce e in stanga va fi mai mic decat el, iar cum succesorul este nodul imediat urmator ca si valoare nodului de sters tot ceea ce ramane in dreapta urmeaza sa fie mai mare decat succesorul, astfel ca proprietatea de arbore binar se pastreaza.

Succesorul unui nod se obtine destul de simplu. Sa notam cu x nodul al carui successor dorim sa il calculam – succesorul lui x, daca exista unul clar trebuie sa fie in partea dreapta pentru ca trebuie sa fie elementul imediat urmator ca valoare fata de nodul x.

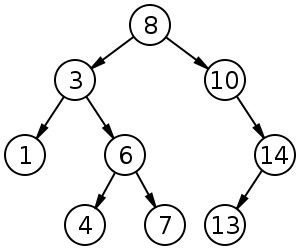
Astfel succesorul nodului x trebuie sa fie minimul din arborele dreapta al lui x. Minimul dintr-un arbore il putem obtine mergand in stanga pana cand nu mai exista copii in stanga.

Implementarea succesorului unui nod si implementarea minimului dintr-un arbore sunt demonstrate in

Curs1\Exemple\tree\

O data ce am gasit succesorul avem urmatoarele cazuri, succesorul e copilul imediat din dreapta, si cum pentru a fi successor trebuie sa nu mai aiba copil in stanga (altfel acela ar fi succesorul) si atunci putem sterge nodul dorit si sa il inlocuim cu succesorul sau, astfel succesorul va avea in stanga copilul din stanga al nodului de sters si in dreapta copilul sau din dreapta.

In cazul in care succesorul nu e copilul imediat din dreapta:



(imagine luata de pe wikipedia <http://en.wikipedia.org/wiki/Binary_search_tree#/media/File:Binary_search_tree.svg>)

Sa zicem ca vrem sa stergem nodul 3, succesorul sau este 4, vom interschimba nodul 4, astfel ca parintele sau sa devina copilul sau in dreapta, in timp ce copilul sau in stanga va fi NULL. Daca nodul 4 ar fi avut un copil in dreapta, atunci acesta lua locul lui 4, iar 4 urma sa aiba ca si copil dreapta pe 6, in timp ce ca si copil stanga urma sa aiba arborele din stanga nodul de sters, 3.

Implementarea poate fi gasita in:

Curs 1\Exemple\Tree

**Heapuri**

O alta structura de date importanta este structura de heap. Sa consideram un vector pe care sa il privim ca un arbore astfel:

Elementul i din vector are ca si copii pe 2\*i si 2\*i+1 (in cazul vectorilor de la 0 vorbim de 2\*i+1 si 2\*i+2). Notam pentru simplitate cu left valoarea pozitiei copilului stanga si cu right valoarea pozitiei copilului dreapta. Daca avem i atunci parintele lui i va fi i/2 in cazul vectorilor de la 1, si i/2 -1 in cazul vectorilor de la 0. Pentru simplitate notam cu parent(i).

Proprietatea de heap este proprietatea:

Pentru orice i de la primul element si pana la ultimul element din vector avem:

A[parent(i)] > A[i].

Daca presupunem ca avem un vector care satisface proprietatea de heap pentru toate elementele mai putin cel de la pozitia pos, pentru a face acel vector sa satisfaca pentru toate elementele proprietatea de heap vom folosi algoritmul – pentru simplitate vom presupune vectori care incep de la pozitia 1:

Do\_heap\_position vector, size\_vector, pos

l = left(pos);

r = right(pos);

Daca l < size\_vector si vector[l] > vector[pos]

Atunci max = l

Altfel

max = pos;

Daca r < size\_vector si vector[r] > vector[max]

Atunci max = r

Daca max <> pos

Schimba vector[max] cu vector[pos]

Do\_heap\_position vector, size\_vector, max

Astfel daca nu este satisfacuta proprietatea de heap pentru pozitia pos, alegem cel mai mare element dintre pos si cei doi copii si il punem pe pozitia pos, continuand recursive algoritmul pentru elementul interschimbat cu pos pentru ca acesta s-ar putea sa nu satisfaca proprietatea de heap.

Daca avem un vector aleator pentru a-l transforma intr-un vector cu proprietatea de heap vom folosi un algoritm bottom-up. Daca incepem de la coada vectorului, acolo unde sunt frunzele putem considera ca fiecare frunza are proprietatea de heap pentru ca nu are copii, astfel ca ne vom duce mai sus in arbore si pentru fiecare element din arbore care are copii vom apela functia do\_heap\_position descrisa mai sus. O data adus vectorul in proprietatea de heap putem sa il sortam foarte usor astfel:

Elementul din pozitia 1 a vectorului va fi cel mai mare pentru ca va fi varful arborelui si toti copiii sai vor fi mai mici decat el. Astfel interschimband elementele 1 cu n pe pozitia n vom avea elementul cel mai mare. Vectorul de n-1 elemente ramas va pastra proprietatea de heap, mai putin pe pozitia 1, unde tocmai a fost pus elementul de pe pozitia n. Astfel, vom apela functia do\_heap\_position pentru pozitia 1, cu vectorul considerat micsoat la n-1 elemente. Reluam acesti pasi pana ajungem cu vectorul de 1 element, in timp ce pe pozitiile n, n-1, … 2 tocmai am pus elementele cele mai mari la fiecare pas, obtinand un vector sortat.

Timpul necesar pentru construirea proprietatii de heap este de (n/2)\*logn. Extragerea celui mai mare element se face in O(1). Ordonarea vectorului aflat in proprietatea de heap se face prin extragerea succesiva a celui mai mare element din vectorul initial, apoi micsorat fara cel mai mare element si asa mai departe, necesitand n extrageri si dupa fiecare extragere necesitand refacerea proprietatii de heap , fapt ce inseamna n extrageri \* logn –timpul necesar refacerii proprietatii de heap. Putem deci concluziona ca timpul necesar sortarii vectorului folosind metoda heapsort este de O(nlogn).

A se vedea Exemple\Heap.

**Hashmapuri**

Hashmapurile sunt o combinatie intre vectori si liste. Putem privi un hashmap ca un cap de tabel care contine capete de liste. Modul in care sunt tinute elementele intr-un hashmap