Tema 2

Arbori de intervale

Responsabil: Dragoş Corlătescu

7 Aprilie 2020

1. Introducere

Arborii binari de căutare, în special cei echilibraţi, sunt folosiţi în marea majoritate a cazurilor pentru a eficientiza căutarea, după cum le spune şi numele. La laborator aţi învăţat cum să îi construiţi şi cum să găsiţi elemente rapid. Pentru această temă vă propun extinderea utilizării arborilor de la a găsi informaţii despre un element la a găsi informaţii despre un interval.

Arborii de intervale sunt utilizaţi pentru a reduce timpul de căutare a unei informaţii într-un interval. Complexitatea temporală în cazul folosirii acestora se reduce de la una liniară (căutare element cu element) la una logaritmică folosindu-ne de proprietăţilor arborilor binari echilibraţi.

2. Arbori de intervale

Un arbore de intervale este un arbore binar echilibrat (diferenţa absolută între adâncimea subarborelui stâng şi cea a subarborelui drept este cel mult 1). În general, informaţiile conţinute de un nod dintr-un astfel de arbore sunt:

- **stânga** - capătul din stânga al intervalului reprezentat de nod
- **dreapta** - capătul din dreapta al intervalului reprezentat de nod
- **info** - informaţia utilă (adiţională) prezentă în nod
- **copil_stânga** - subarborele stâng al nodului
- **copil_dreapta** - subarborele drept al nodului

Intervalul **[stânga, dreapta]** asociat unui nod se numeşte *interval standard*. Intervalul unei frunze din arbore va avea întotdeauna stânga egală cu dreapta şi poartă numele de *interval elementar*.

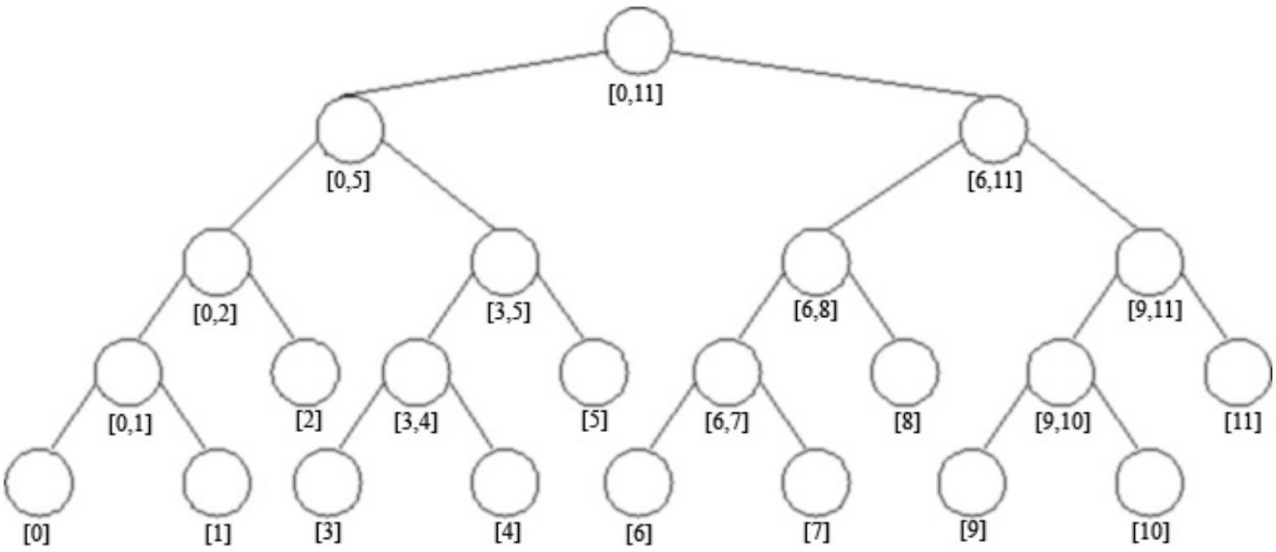
Operaţii arbore de intervale:

1. Creare arbore de intervale dându-se un interval **[st, dr]** se va realiza recursiv în felul următor:

```
funcţie creare_nod(st, dr):
        iniţializare nod cu stânga = st şi dreapta = dr
        dacă st < dr atunci:
                mijloc = (st + dr) / 2
                copilul_stâng_al_nodului = creare_nod(st, mijloc)
                copilul_drept_al_nodului = creare_nod(mijloc + 1, dr)
```

Această funcţie se va aplica începând cu nodul rădăcină al arborelui de intervale care va reprezenta tot intervalul şi se va aplica recursiv până în frunze care nu vor mai respecta condiţia din blocul "dacă".

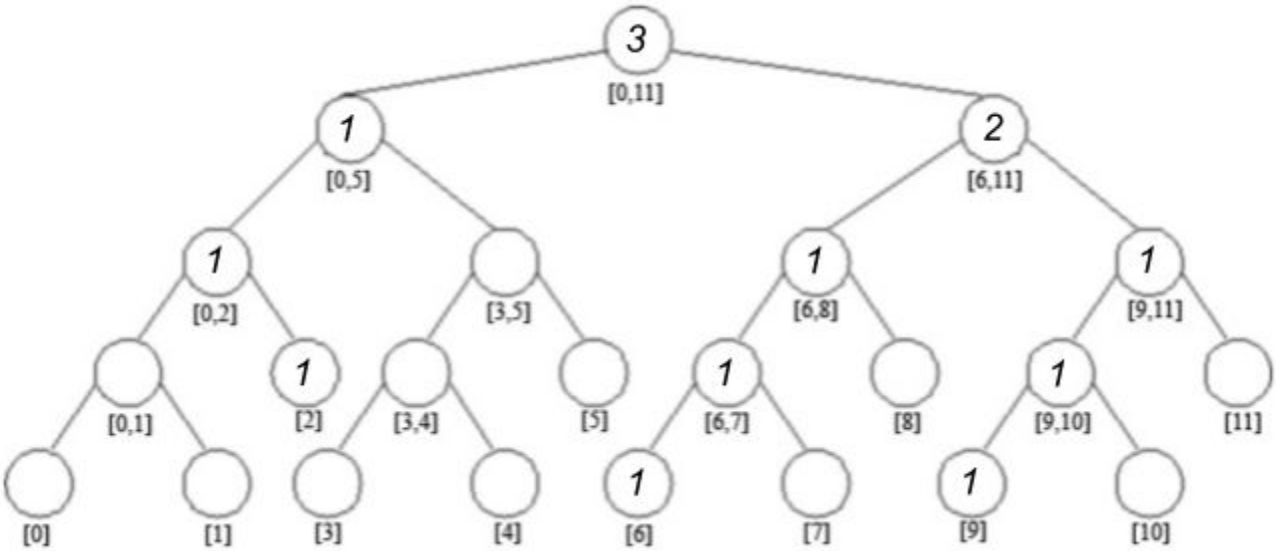
Exemplu - pentru intervalul **[0, 11]** se va genera următorul arbore de intervale (fără partea de informaţie):



2. Actualizarea unui interval dat prin **[a, b]** se va face aplicând următorul pseudocod:

```
funcţie actualizare(nod, [a, b]):
        dacă a <= nod->stanga şi nod->dreapta <= b:
                modifică/actualizează nod->info
        altfel:
                mijloc = (nod->stanga + nod->dreapta) / 2
                dacă a <= mijloc:
                        actualizare(nod->copil_stanga, [a, b])
                dacă mijloc < b:
                        actualizare(nod->copil_dreapta, [a, b])
                modifică/actualizează nod->info
```

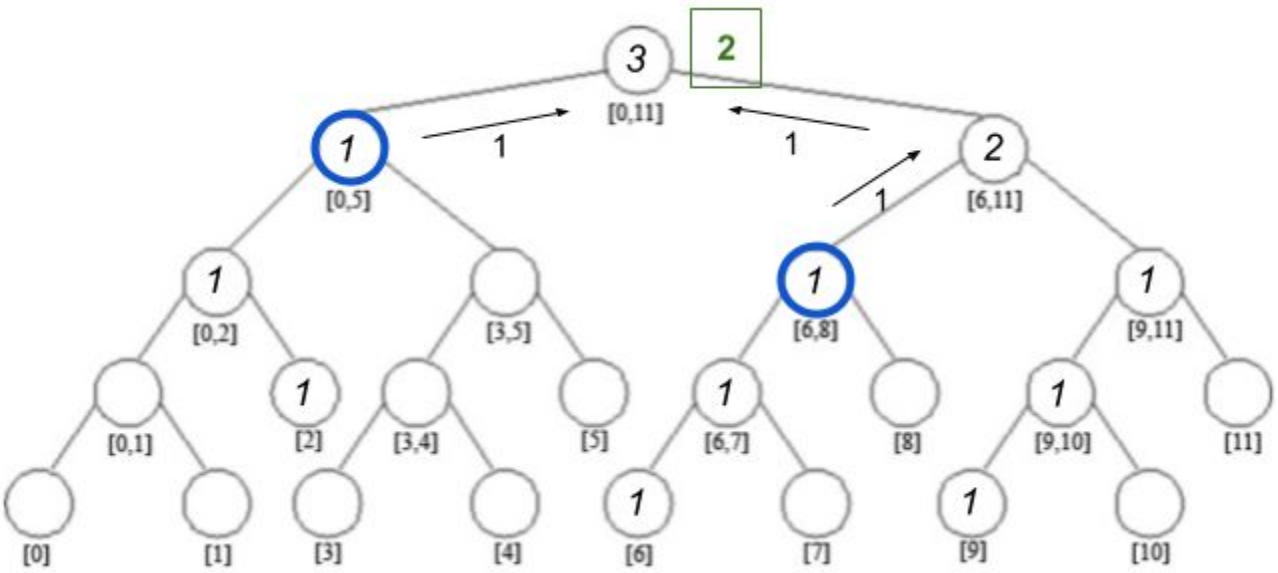
Ca exemplu, dacă presupunem că informaţia utilă din fiecare nod a fost iniţializată cu 0 şi modul prin care modificăm/actualizăm informaţia din nod este să adăugăm 1 la informaţia deja existentă atunci, dacă dorim să actualizăm intervalele [2, 2], [6, 6] şi [9, 9], vom obţine un arbore care arată în felul următor (plecând de la cel prezentat anterior):



3. Interogarea unui interval dat prin **[a, b]** se va face aplicând următorul pseudocod:

```
funcţie interogare(nod, [a, b]):
        dacă a <= nod->stanga şi nod->dreapta <= b:
                întoarce nod->info
        altfel:
                mijloc = (nod->stanga + nod->dreapta) / 2
                dacă a <= mijloc:
                        valoare_stanga = interogare(nod->copil_stanga, [a, b])
                dacă mijloc < b:
                        valoare_dreapta = interogare(nod->copil_dreapta, [a, b])
                întoarce combinaţie_între_valoarea_stangă_şi_valoarea_dreapta
```

De exemplu, dacă considerăm arborele generat anterior şi ca funcţie de combinare între valori alegem suma acestora dacă ele există (aveţi în vedere că una din ele s-ar putea să nu existe), sau doar valoarea care există în caz contrar, atunci pentru intervalul [0,8] interogarea va întoarce 2:



3. Implementare C Arbori de Intervale (50 puncte)

Pentru acest task veţi lucra cu 3 fişiere:

- arboriTest.c - fişier de testare, NU veţi avea de modificat/implementat nimic aici.
- arbori_intervale.h - fişier antet în care sunt definite structurile cu care veţi lucra şi antetul funcţiilor pe care va trebui să le implementaţi pentru acest task.
- arbori_intervale.c - fişier sursă în care veţi implementa funcţiile definite în fişierul antet. Evident că nu trebuie să vă limitaţi la acestea, puteţi să vă implementaţi alte funcţii ajutătoare dacă consideraţi necesar (chiar vă recomand să faceţi acest lucru).
- extra: Makefile care conţine reguli de build şi run pentru toate cerinţele temei, deci inclusiv pentru aceasta.

Definirea structurilor cu care veţi lucra:

- structura Interval - reprezintă un interval simplu care conţine două câmpuri de tip int: capătul din stânga şi capătul din dreapta al intervalului.

```
typedef struct interval {
    int capat_stanga;
    int capat_dreapta;
} Interval;
```

- structura Nod - reprezintă un nod din arborele de intervale. Acesta conţine un int reprezentând informaţia utilă păstrată în nod, un pointer la un Interval reprezentând intervalul standard al nodului şi doi pointeri către copii (stânga şi dreapta).

```
typedef struct nod {
    int info;
    Interval *interval;
    struct nod *copil_stanga;
    struct nod *copil_dreapta;
} Nod;
```

- Structura ArboreDeIntervale - reprezintă chiar arborele de intervale. Acesta conţine o referinţă la un Nod rădăcină şi dimensiunea arborelui. Pe lângă acestea, mai sunt 3 câmpuri pe care le voi detalia în cele ce urmează:

- int valoare_predefinita_raspuns_copil - vă spuneam că la funcţia de interogare pe cazul "altfel" se poate interoga atât subarborele stâng, cât şi subarborele drept, dar există şi cazul în care se va interoga doar unul dintre aceştia. În acest ultim caz, valoarea "întoarsă" de subarborele neexplorat va fi egală cu această *valoare_predefinita_raspuns_copil*. ! Vedeţi că răspunsul este o funcție care depinde de ambele interogări stânga, dreapta.
  *Pentru acest task şi pentru următorul valoarea este **0**. Pentru bonus, va trebui să o iniţializaţi cu altă valoare.*
- void (*f_actualizare)(Nod *nod, int valoare_ce_modifica) - pointer la funcţie care va fi folosită în funcţia de actualizare să modifice info din nod.
  *Pentru acest task şi pentru următorul funcţia este implementată în arboriTest.c, respectiv intersectiiTest.c. Ea adaugă la nod->info valoarea "valoare_ce_modifică". Pentru acest task, "valoare_ce_modifică" va fi egală cu **1**. Pentru bonus, s-ar putea să nu vă trebuiască această valoare (silent hint).*
- int (*f_combinare_raspunsuri_copii)(int raspuns_stanga, int raspuns_dreapta) - pointer la funcţie care va fi folosită în funcţia de interogare când vrem să combinăm cele două răspunsuri.
  Pentru acest task şi pentru următorul funcţia este deja implementată în arboriTest.c. Ea va calcula suma răspunsurilor venite din copii. Pentru bonus, s-ar putea să nu mai fie aceeaşi funcţie.

```
typedef struct adi {
    Nod* radacina;
    int dimensiune;
    int valoare_predifinita_raspuns_copil;
    void (*f_actualizare)(Nod *nod, int valoare_ce_modifica);
    int (*f_combinare_raspunsuri_copii)(int raspuns_stanga, int raspuns_dreapta);
} ArboreDeIntervale;
```

Pentru obţinerea punctajului maxim pentru acest task trebuie să implementaţi funcţiile prezentate mai sus în pseudocod şi să treacă testele. Urmăriţi şi indicaţiile din schelet.

Pentru compilare rulaţi: > make buildArbori

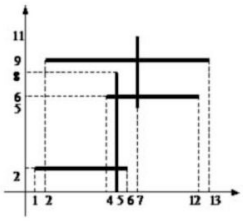
Pentru testare rulaţi: > make runTestArbori

4. Intersecţii de intervale (40 puncte)

Pentru acest task veţi lucra cu 3 fişiere:

- intersectiiTest.c - fişier de testare, NU veţi avea de modificat/implementat nimic aici.
- intersectii.h - fişier antet în care sunt definite structurile cu care veţi lucra şi antetul funcţiilor pe care va trebui să le implementaţi pentru acest task.
- intersectii.c - fişier sursă în care veţi implementa funcţiile definite în fişierul antet. Evident că nu trebuie să vă limitaţi la acestea, puteţi să vă implementaţi alte funcţii ajutătoare dacă consideraţi necesar (chiar vă recomand să faceţi asta).
- extra: Makefile care conţine reguli de build şi run pentru toate cerinţele temei, deci inclusiv pentru aceasta.

Problema: Dându-se o mulţime de segmente în plan care pot fi ori verticale, ori orizontale să se calculeze numărul de total de intersecţii dintre aceste segmente. Exemplu:

segment.in	segment.out	Figura
5 2 9 13 9 4 6 12 6 1 2 6 2 5 0 5 8 7 5 7 11	4	

Rezolvarea acestei probleme se va face în două moduri:

- Varianta trivială (**10 puncte**): veţi verifica interval cu interval dacă acestea se intersectează şi veţi întoarce numărul lor. Această rezolvare are o complexitate de O(n^2) (veţi învăţa mai multe despre complexităţi în anul 2).
- Varianta cu arbori de intervale (**30 puncte**):

Pentru a implementa o rezolvare mai eficientă se vor folosi ideile algoritmilor de “baleiere” (despre care veţi putea citi mai multe în materialele puse la dispoziţie la sfârşitul acestui document). Pe scurt, ne vom plimba cu o dreapta verticală imaginară care v-a intersecta segmentele noastre de la stânga la dreapta şi vom face următoarele acţiuni:

- dacă întâlnim un capăt stânga al unui segment orizontal atunci vom “salva” y-ul acestui interval în structură (spoiler alert: arbore de intervale)

- dacă întâlnim un capăt dreapta al unui segment orizontal atunci vom scoate y-ul acestui interval din structură
- dacă întâlnim un segment vertical atunci vom calcula câte segmente orizontale care sunt deja în structură se intersectează cu acest segment vertical
  Explicaţie suplimentară aici: dacă eu am dat de un segment vertical înseamnă că acest segment are coordonata x în interiorul intervalelor care încă mai sunt în structură. Pentru o găsi intersecţiile mai rămâne să aflu câte din acele intervale pe orizontală au y-ul în interiorul intervalului vertical.

Concret, voi trebuie să implementaţi următorii paşi ţinând cont şi de schelet:

1. Citiţi intervalele din fişierul de intrare.
2. Calculaţi y-ul maxim şi creaţi-vă un arbore de intervale (vezi primul task) cu intervalul iniţial 0, y_maxim. Celelalte functii pentru pointeri le aveti in fişierul intersecţii.c. Acest arbore va reţine câte intervale orizontale sunt într-un interval dat.
3. Sortaţi crescător toate punctele (fie capăt stânga, fie capăt dreapta, nu contează) după coordonata x. Aveţi însă grijă să ştiţi de la ce interval vine un astfel de punct. (Eventual vă mai faceţi o structură care să ţină minte acest lucru).
4. Aplicaţi algoritmul de mai sus parcurgând această listă sortată cu menţiunea că atunci când vreţi să actualizaţi o coordonată “y” veţi actualiza în arborele de intervale intervalul [y, y] cu *v_actualizare* 1 la adăugare şi *v_actualizare* -1 la ştergere. Interogarea intervalului vertical se va face folosind chiar coordonatele y ale acestuia.

Datele de intrare se vor citi dintr-un fişier cu următorul format:
Pe prima linie va fi un număr N reprezentând numărul de intervale.
Pe următoarele N linii vor fi câte 4 numere reprezentând x_punct_stânga, y_punct_stânga, x_punct_dreapta, y_punct_dreapta al unui interval.

Exemplu ca şi mai sus:

segment.in

```
5
2 9 13 9
4 6 12 6
1 2 6 2
5 0 5 8
7 5 7 11
```

Pentru obţinerea punctajului maxim pentru acest task trebuie să implementaţi algoritmul mai sus prezentat şi să treacă testele. Urmăriţi şi indicaţiile din schelet: structuri, funcţii, comentarii.

Pentru compilare rulaţi: > make buildIntersectii

Pentru testare rulaţi: > make runTestIntersectii

5. Poziţie liberă (20 de puncte) - Bonus

Pentru acest task veţi lucra cu 3 fişiere:

- poziţieTest.c - fişier de testare, NU veţi avea de modificat/implementat nimic aici.
- pozitie_libera.h - fişier antet în care sunt definite structurile cu care veţi lucra şi antetul funcţiilor pe care va trebui să le implementaţi pentru acest task.
- pozitie_libera.c - fişier sursă în care veţi implementa funcţiile definite în fişierul antet. Evident că nu trebuie să vă limitaţi la acestea, puteţi să vă implementaţi alte funcţii ajutătoare dacă consideraţi necesar (chiar vă recomand să faceţi asta).
- extra: Makefile care conţine reguli de build şi run pentru toate cerinţele temei, deci inclusiv pentru aceasta.

Problema: Datele de intrare sunt: o listă de numere (să-i spunem S), numărul acestora (N) şi elementul maxim (E_MAX) din această listă. Voi trebuie să completaţi o altă lista (să-i spunem T) având dimensiunea (element maxim + 1) în felul următor:

- Parcurg lista S de la stânga la dreapta element cu element.
- Pe fiecare element voi încerca să-l adaug în lista T astfel:
    - Dacă pe poziţia(indexul) element nu este nimic în lista T, îl voi adăugat pe poziţia respectivă. (adică T[element] = element if T[element] gol)
    - Dacă pe poziţia elementului se află deja ceva atunci voi adăuga elementul pe prima poziţie liberă pornind de la poziţia *element* până la 0. Dacă nu mai există nicio poziţie liberă atunci nu îl voi mai adăuga deloc.

Punctarea se va face în felul următor:

- **5 puncte** soluţia trivială (2 for-uri).
- **15 puncte** soluţia folosind arbori de intervale.

Exemplu date de intrare şi desfăşurarea rezolvării:

Fişierul de intrare va arăta ca mai joc cu E_MAX fiind elementul maxim, N numărul de elemente din listă şi apoi pe următoarele N linii elementele listei:

E_MAX N
n1
n2
..
nn

Exemplu:
4 6
3
4
3
4
1
2

Paşi:

- Iniţializez vectorul T cu o dimensiune E_MAX + 1 (deci 5)

Nimic	Nimic	Nimic	Nimic	Nimic

- Vreau să îl adaug pe 3. Poziţia 3 este liberă deci îl voi adăuga în acel loc.

Nimic	Nimic	Nimic	**3**	Nimic

- Vreau să îl adaug pe 4. Poziţia 4 este liberă deci îl voi adăuga în acel loc.

Nimic	Nimic	Nimic	3	**4**

- Vreau să îl adaug pe 3. Poziţia 3 este ocupată deci voi căuta următoarea (înspre 0) poziţie liberă care este poziţia 2 şi îl voi pune acolo.

Nimic	Nimic	**3**	3	4

- Vreau să îl adaug pe 4. Poziţia 4 este ocupată deci voi căuta următoarea (înspre 0) poziţie liberă care este poziţia 1 şi îl voi pune acolo.

Nimic	**4**	3	3	4

- Vreau să-l adaug pe 1. Poziția 1 este ocupată deci voi căuta următoarea (înspre 0) poziție liberă care este poziția 0 și îl voi pune acolo.

1	4	3	3	4

- Vreau să-l adaug pe 2. Poziția 2 este ocupată deci voi căuta următoarea (înspre 0) poziție liberă. Nu voi mai găsi nimic liber deci nu îl voi adăuga pe 2 nicăieri.

- S-a terminat lista S de la intrare, deci soluția este:

1	4	3	3	4

Pentru obținerea punctajului maxim pentru acest trebuie să implementați algoritmul mai sus prezentat și să treacă testele. Urmăriți și indicațiile din schelet: structuri, funcții, comentarii.
Pentru compilare rulați: > make buildPozitie
Pentru testare rulați: > make runTestPozitie

6. Punctare și trimitere

Tema se va încărca pe moodle sub forma unei arhive cu numele *grupa_nume_prenume_tema2sd.zip* până la data de **27.04.2020 ora 23:55**. Acest deadline este **hard**.

Punctarea se va face pe baza checkerului după cum urmează:

- 50 puncte implementarea arborelui
- 40 puncte intersecții
- 20 puncte poziții (bonus)
- 10 puncte readme, coding style

7. Bibliografie, disclaimer şi materiale adiţionale

Pentru realizarea acestei teme am folosit resurse din mediul online pe care vă recomand să le citiţi şi voi pentru o înţelegere mai profundă a conceptelor explicate în temă. De referinţă este articolul doamnei Dana Lica (care mi-a fost şi mie profesoară în liceu :D) pe care îl puteţi găsi aici:

http://campion.edu.ro/arhiva/www/arhiva_2009/papers/paper14.pdf

sau pe infoarena:

https://infoarena.ro/arbori-de-intervale

O să observaţi că prima problemă explicată în acest material împreună cu informaţiile despre arborii de intervale sunt chiar primele două task-uri ale acestei teme. Există unele modificări la reprezentarea (structura) arborelui, dar conceptual ele se aseamănă.

Alte resurse:

https://sites.google.com/site/centrulinfo1/materiale-video/algoritmi-video/arbori-de-intervale

https://en.wikipedia.org/wiki/Interval_tree