

Universitatea București
Facultatea de Matematică și Informatică

LUCRARE DE LICENȚĂ

COORDONATOR ȘTIINȚIFIC:
Lect. dr. Marin Letiția

ABSOLVENT
Bălțatu Andrei-Mircea

BUCUREȘTI
2018

Universitatea București
Facultatea de Matematică și Informatică

LUCRARE DE LICENȚĂ

**Algorithms Visualization Library – Aplicație de vizualizare a
algoritmilor și structurilor de date**

COORDONATOR ȘTIINȚIFIC:
Lect. dr. Marin Letiția

ABSOLVENT
Bălțatu Andrei-Mircea

BUCUREȘTI
2018

Cuprins

1. Introducere.....	2
1.1. Motivație.....	2
1.2. Obiectiv.....	2
1.3. Soluții existente	2
1.3.1. <i>PathFinding.js</i>	3
1.3.2. <i>USF Data Structure Vizualizations</i>	4
1.3.3. <i>CsAcademy Applications</i>	5
2. Concepte teoretice și arhitectura aplicației.....	7
2.1. Algoritmi și structuri de date	7
2.2. Arhitectura aplicației	11
2.3.1. Arhitectura pe <i>front-end</i>	11
2.3.2. Arhitectura pe <i>back-end</i>	14
2.3. Simulator de <i>debugger</i>	16
3. Tehnologii utilizate.....	18
3.1. <i>Javascript</i>	18
3.2. <i>Angular</i>	19
3.3. <i>NodeJs</i>	22
3.4. <i>PostgreSQL</i>	23
3.5. Biblioteci.....	23
3.5.1. <i>Vis.js</i>	23
4. Descrierea aplicației	25
5. Concluzii	26
Bibliografie.....	29

1. Introducere

1.1. Motivație

Odată cu intrarea în lumea informaticii, o persoană are de învățat anumiți algoritmi sau structuri de date care îi vor fixa o bază și îl vor ajuta mai departe pentru rezolvarea problemelor pe care le întâmpină. De obicei, premisa care stă la baza învățării acestora este aceea că trebuie înțeles mai întâi modul de rulare a programului (funcții, bucle, recursivitate). Apoi, algoritmi și structurile de date pot fi abstractizate pentru o înțelegere mai ușoară a unui program, prin faptul că nu mai suntem restrictionați de implementare ci mai mult de ideile principale. De aceea, în primă fază, se folosește un pseudocod pentru a explica un algoritm sau o structură de date.

Abstractizarea ne îndepărtează de lucrurile cu care suntem obișnuiți (de exemplu, lucrul cu grupurile algebrice) și ne pune câteodată în situații dificile, în care ne este greu să vizualizăm/imaginăm soluția unei probleme. Astfel, este mai ușor să facem gradual această tranziție, prin exemple sau prezentarea unor cazuri mai simple.

1.2. Obiectiv

Această aplicație are scopul de a ușura și minimiza timpul de învățare a unui algoritm sau unei structuri de date descrise în următoarele pagini. Există situații când, chiar dacă codul scris pare corect, este mai ușor de vizualizat prin imagini și rulare pas cu pas a programului. Diferența dintre modul în care sunt ținute datele în calculator și semnificația pe care le-o dăm sau modul în care vizualizăm pentru a ușura înțelegerea, constituie unul din principalele scopuri cu care a fost scrisă aplicația.

Totodată, se pune la dispoziția utilizatorului un mediu de lucru flexibil, controale de flux ale programului (butoane de începere/pauză, un *timer* pentru fiecare pas al programului), o componentă de vizualizare a acelui algoritm sau structură de date, puncte de merit pentru răspunsuri corecte, pentru transforma învățarea într-o experiență plăcută și provocatoare.

Obiectivul final este ca un utilizator să poată învăța în modul ales de el, mai încet sau mai rapid, și energia depusă să fie cât mai mare pe acel element și nu pe dificultatea utilizării acestei aplicații.

1.3. Soluții existente

Există mai multe soluții de acest gen, gratuite sau nu, iar dintre acestea am ales să discut și compar trei.

Când avem în vedere o soluție pentru problema de învățare a unui algoritm sau structură de date, luăm în considerare următoarele aspecte:

- **flexibilitatea aplicației** (dacă utilizatorul poate modifica intrările, schimba codul, pune *breakpoints* sau comentarii);
- **tip de vizualizare** (dacă se poate da *zoom*, dacă are culori, dacă interfața este statică sau interactivă);
- **controlul de flux** (dacă putem opri programul la un anumit pas, inversa pasul curent, stabili un interval de parcurgere a liniilor de cod);

Toate aceste caracteristici contează pentru un utilizator și pentru modul de învățare pe care îl aplică.

1.3.1. *PathFinding.js*

Aceasta este o aplicație ([6]) care simulează diferiți algoritmi de găsim a drumurilor minime pe o matrice. Totuși, unii din algoritmi prezențați pot fi aplicați și pe grafuri. Câțiva dintre algoritmi care pot fi selectați de utilizator sunt:

- *A**
- *Dijkstra*
- *Breadth-First search*
- *Jump-Point search*

Utilizatorul poate alege pe oricare dintre aceștia și tipul de distanță care se folosește la calcularea costului unui drum (euclidiană, Manhattan). Pe lângă acestea, utilizatorul dispune de o vizualizare interactivă care acceptă comenzi de *mouse-click* sau *drag-and-drop*. Există o matrice de dimensiuni fixe, cu un punct de start reprezentat prin verde închis și un punct de oprire reprezentat prin roșu.

Mai mult, comanda *click* pe o celulă comută o celulă liberă într-una invalidă, și invers.

Aplicația are butoane de începere și oprire a algoritmului, dar rulează foarte rapid pentru o matrice cu dimensiunile ilustrate, astfel că utilizatorul nu are timp să apese butonul de oprire. Totuși, oferă o funcționalitate folositoare în unele situații.

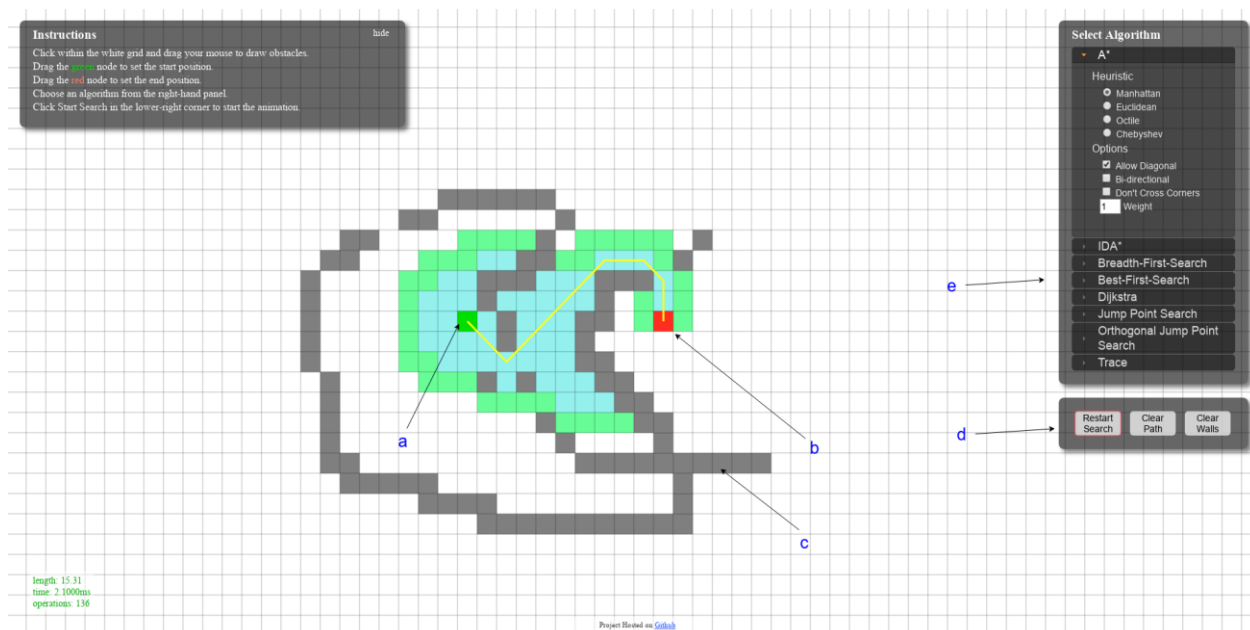


Figura 1-1 *PathFinding.js*

a) celulă start; b) celulă sfârșit; c) celulă invalidă; d) controale flux; e) listă algoritmi

Sursă: <https://qiao.github.io/PathFinding.js/visual/>

Astfel, aplicația este flexibilă, utilizatorul putând schimba aproape tot cu excepția dimensiunilor matricei. Ea oferă o vizualizare care diferențiază diferitele tipuri de celule și care este interactivă (acțiunile facându-se cu ajutorul *mouse*-ului). Totuși partea de control al fluxului oferă un set incomplet, lipsând controlul de timp și altele.

1.3.2. *USF Data Structure Visualizations*

Universitatea din San Francisco, departamentul de Computer Science, a construit o listă de vizualizări pentru diverși algoritmi și structuri de date ([7]):

- sortări (*Radix sort*, *Bucket sort*, *Heap sort*);
- structuri de indexare (*Hash tables*, arbori indexați binar, arbori AVL);
- programare dinamică (numere Fibonacci, problema celui mai lung subșir comun);
- algoritmi de grafuri (*Depth-First search*, *Breadth-First search*, componente conexe, arbore parțial de cost minim);

Având o mulțime vastă de algoritmi și structuri de date, ne vom referi la principalele componente care se găsesc în vizualizări și la avantajele/dezavantajele pe care această aplicație le are.

Ca și controale de flux, aplicația oferă:

- un *slider* pentru viteza animației unei operații din cadrul exercițiului;

- butoane de începere și pauză;
- butoane care oferă posibilitatea parcurgerii doar a câte unui pas în algoritm, înainte sau înapoi;
- butoane speciale care depind de exercițiu și definesc operațiile pe care un utilizator le poate selecta;

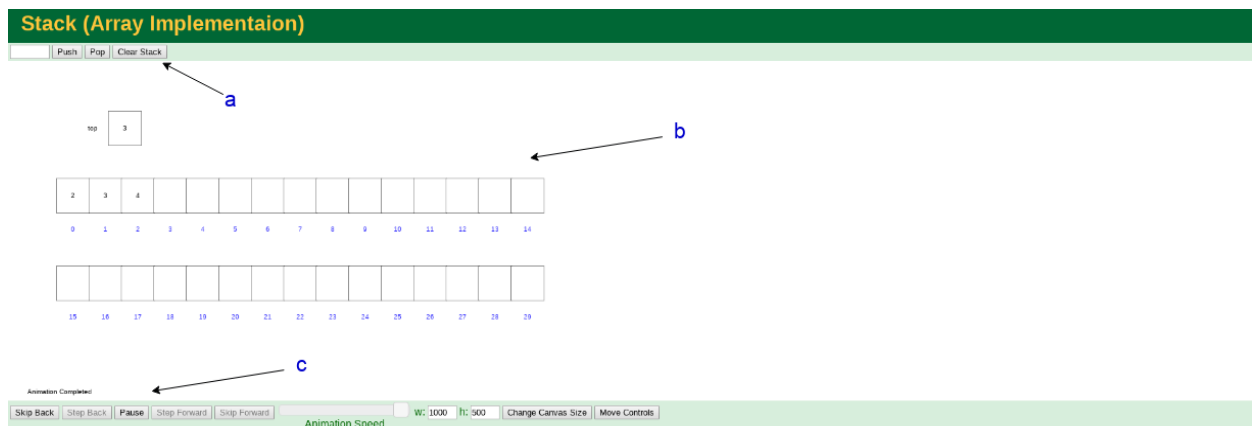


Figura 1-2 *USF - Stack implementation*

a) operații; b) vizualizare; c) controale de flux;

Sursă: <https://www.cs.usfca.edu/~galles/visualization/StackArray.html>

Majoritatea vizualizărilor nu sunt interactive, ci doar operațiile specifice aceluși exercițiu schimbă vizualizarea. Totuși, aplicația are animații destul de interesante și diverse, plăcute utilizatorului. Vizualizarea este conținută într-un bloc *canvas* care se poate redimensiona prin editarea a două valori, *width* și *height*.

Aplicația este relativ flexibilă, putând primi ca date de intrare și ieșire orice valori în formatul descris de exercițiu. Un dezavantaj este faptul că nu se pot introduce mai multe operații deodată. Astfel, când sesiunea a expirat, trebuie reintroduse manual operațiile. De obicei, există mai multe operații care se fac asupra unei structuri de date sau algoritm, și ar fi natural să se poată preciza numărul de operații pe prima linie a datelor de intrare, iar după aceea să se descrie fiecare operație pe următoarele linii.

În concluzie, aplicația dezvoltată de Universitatea din San Francisco oferă o gamă largă de algoritmi și structuri de date și o pagină de simulare destul de completă. Cu toate acestea, există câteva dezavantaje, cum ar fi vizualizările care nu pot fi redimensionate cu ajutorul acțiunilor *mouse*-ului, sau operațiile specifice unei probleme care trebuie introduse una câte una.

1.3.3. *CsAcademy Applications*

Acest *site* ([8]) este folosit în principal pentru concursuri de algoritmică, dar conține și sub-aplicații pe care orice utilizator le poate folosi dacă dorește să vizualizeze o structură de date.

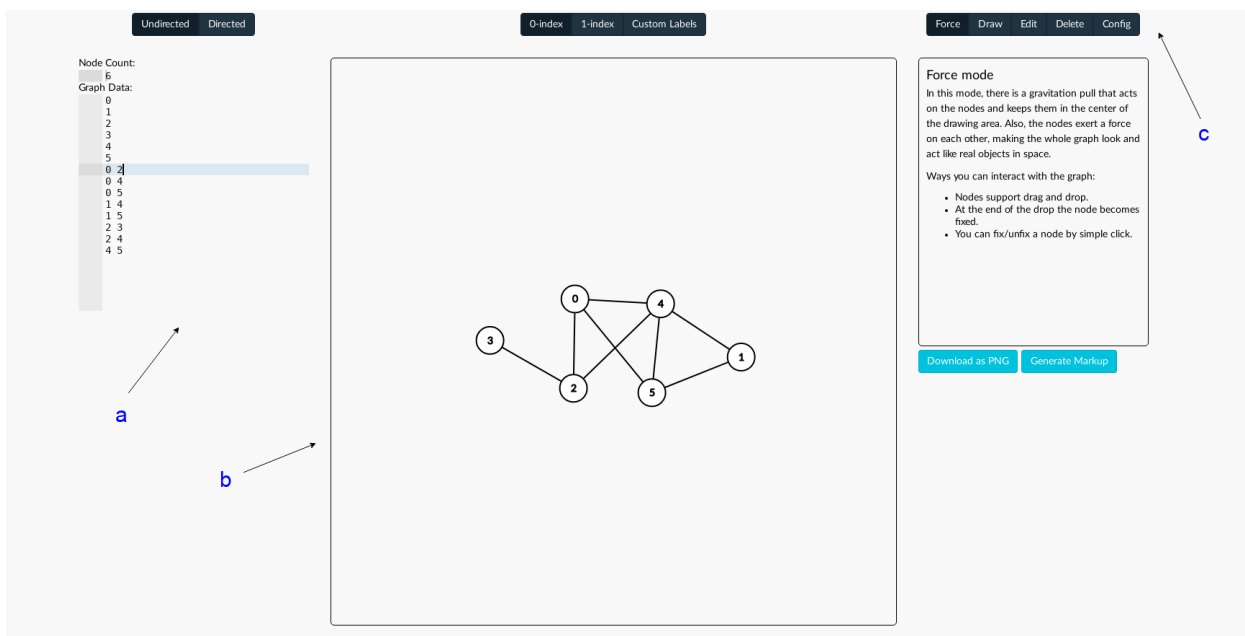


Figura 1-3 CsAcademy - Graph Editor

a) date de intrare; b) vizualizare; c) moduri vizualizare

Sursă: https://csacademy.com/app/graph_editor/

Cele mai importante aplicații sunt „*Graph Editor*” și „*Geometry widget*”. Aceste aplicații construiesc o vizualizare bazată pe date de intrare scrise de utilizator.

Astfel, la partea de control al fluxului aplicațiile nu au nimic utilizabil. În schimb, vizualizările sunt interactive și pot fi declanșate prin utilizarea *mouse*-ului (*scroll*, *drag-and-drop*).

Mai mult, datele de intrare pot fi editate în totalitate de utilizator, fiind restricționate de formatul impus de aplicație. Totodată, există mai multe moduri pentru editarea vizualizării fără ajutorului datelor de intrare (*Draw*, *Edit*, *Delete*).

În concluzie, aplicațiile CsAcademy nu oferă o gamă largă din care utilizatorul să poată alege și nici o simulare a unui program. Vizualizările sunt singura categoria unde aplicațiile oferă funcționalități, chiar mai mult față de celelalte două aplicații descrise mai sus.

2. Concepte teoretice și arhitectura aplicației

2.1. Algoritmi și structuri de date

Proiectul asociat acestei lucrări are scopul de a-i învăța pe utilizatori anumiți algoritmi și structuri de date dintr-o listă predefinită. În cele ce urmează voi descrie lista pusă la dispoziție în cadrul aplicației, prezentând fiecare element și explicând structura/soluția acestuia.

Algoritmii pe care aplicația îi pune la dispoziție sunt:

- algoritmi de sortare (*Bubble Sort*, *Quick Sort*, *Merge Sort*);
- algoritmi de găsim a drumului minim între două puncte din matrice (Algoritmul lui Lee, *A* search algorithm*);
- algoritmi de grafuri (componente conexe, componente biconexe, arbore parțial de cost minim - algoritmul lui Prim).

Structurile de date pe care aplicația le pune la dispoziție sunt:

- structuri de tip *push-pop* (stivă, coadă, *deque*);
- arbori de intervale/arbori indexați binar.

Algoritmi de sortare

Toți algoritmii de sortare menționați mai sus se bazează pe comparații între două elemente, și nu pe indexare ca alți algoritmi precum *Radix Sort*. În cele ce urmează, este realizată o scurtă prezentare a fiecărui algoritm.

- *Bubble sort*
 - Pasul 1: se parcurge șirul de elemente de la început până la sfârșit, iar dacă două elemente consecutive nu păstrează semnul de comparație $<$, se vor interschimba.
 - Pasul 2: dacă la pasul 1 nu s-a realizat nicio interschimbare, atunci algoritmul se oprește, altfel se reia pasul 1.
- *Merge sort*
 - Se creează o funcție care ia ca parametrii capetele intervalului din șirul inițial care se dorește a sorta. Acest interval se înjumătățește, iar jumătățile se vor sorta recursiv, apelând aceeași funcție. Având jumătățile sortate, le interclasăm pentru a sorta intervalul curent.
- *Quick sort*

- Se procedează similar algoritmului de *Merge Sort*, cu excepția faptului că intervalul curent nu se va mai înjumătăți. Se va alege un element drept pivot, care semnifică faptul că intervalul trebuie partiționat astfel încât primele elemente să conțină valori mai mici sau egale cu valoarea pivotului, iar restul să conțină valori mai mari.
- Apoi, având intervalul partiționat în două subintervale, unul cu elemente mai mici sau egale, altul cu elemente mai mari, se va apela aceeași funcție pentru a le sorta.

Algoritmi de *Path-Finding* pe matrice

De obicei, algoritmi de căutare a drumului minim pe o matrice pleacă dintr-o celulă de început și vor să ajungă într-o celulă de sfârșit. Pe lângă acestea, există celule valide (prin care putem trece) și invalide (prin care nu putem trece). Ne vom referi la acest tip clasic de algoritm în cadrul explicațiilor de mai jos.

- Algoritmul lui Lee.
 - Ne vom folosi de o coadă în care vom insera coordonatele unei celule atunci când o vizităm pentru prima dată. Primul element din coadă este celula de start. Se mai reține și o matrice de valori c_{ij} care reprezintă costul minim de la celula de start la celula (i, j) ;
 - Cât timp coada nu este goală, se ia primul element din coadă și se elimină. Ne vom uita la vecinii acestei celule, iar dacă nu au fost puși în coadă niciodată și sunt celule valide, actualizăm costul și îi punem în coadă;
 - La sfârșitul algoritmului vom avea costurile tuturor drumurilor minime de la celula de start la oricare celulă validă.
- *A* search algorithm*
 - Este similar algoritmului lui Lee, doar că de această dată se va reține o coadă de priorități, iar costul drumului între două elemente este dat de suma a două funcții f și g . În implementarea noastră, $f = c[i][j]$ iar g este reprezentat de distanța Manhattan.
 - Astfel, la fiecare pas în care extragem un nou element din coadă, vom lua în considerare și cât de aproape este o celulă față de *target*. Nu dorim să vizităm celule foarte îndepărtate deoarece, cel mai probabil, nu vor contribui la drumul minim final.

Algoritmi de grafuri

- Componente conexe
 - Se creează o funcție care primește ca parametru un indice al unui nod și care se va apela recursiv. Totodată, vom reține un vector de valori boolene care semnifică dacă un nod a fost vizitat sau nu.
 - Se parcurg indicii nodurilor în ordine crescătoare. Dacă nodul cu indicele curent nu este vizitat, atunci se apelează funcția cu parametrul corespunzător acestui nod.
 - Funcția are rolul de a vizita componenta conexă în care se află un nod. La un anumit apel al funcției, se parcurg vecinii nodului, iar dacă aceștia sunt nevizitați, se apelează funcția recursiv pentru acel vecin. Dacă un nod a fost transmis în funcție ca parametru, atunci modificăm starea lui de vizitare.
- Componente biconexe:
 - Se formează arborele *DFS (Depth-first search)* al grafului, împărțind muchiile în muchii *top* (care nu se găsesc în arbore) și muchii *down* (care se găsesc în arbore);
 - Dacă un nod are o muchie de tip *top* spre alt nod, atunci lanțul din arbore dintre ele se află în aceeași componentă biconexă;
 - Se rulează un algoritm tip DFS unde, dacă un nod fiu nu are în subarborele acestuia vreun nod cu muchie *top* mai sus de nodul curent, înseamnă ca fiul face parte din altă componentă biconexă față de nodul curent. Astfel, se elimină din graf componenta biconexă a fiului și se continuă algoritmul.
- Algoritmul lui Prim:
 - Se sortează muchiile grafului crescător după cost;
 - Se parcurg muchiile sortate și se ține o structură de date numită *Păduri de mulțimi disjuncte* pentru a reține dacă două noduri sunt la un anumit pas în aceeași componentă conexă;
 - Când suntem la o anumită poziție, verificăm dacă nodurile muchiei sunt în aceeași componentă conexă. Dacă sunt, se trece la următoarea muchie. În caz contrar, se leagă aceste două noduri și se inserează muchia în structura de date;
 - La sfârșit, muchiile alese formează arborele parțial de cost minim.

Structuri de date

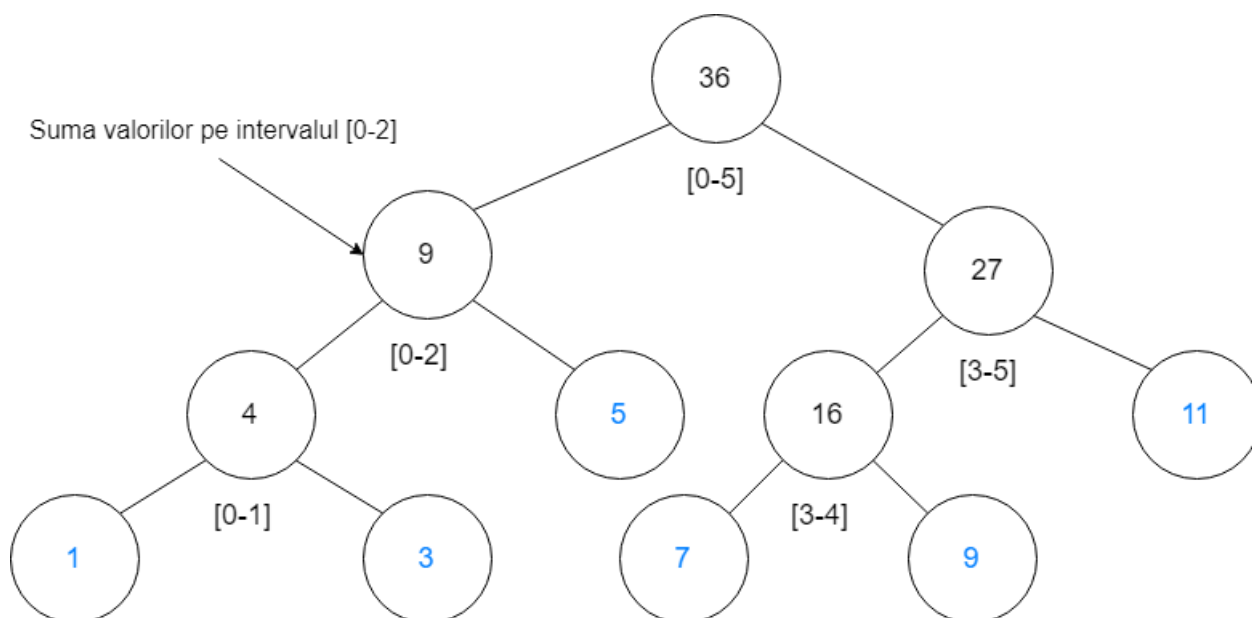
Stivă, coadă, deque

Sunt structuri de date ce reprezintă un șir de obiecte și execută operații de tip *push/pop* în timp constant. Stiva poate executa operații de tip *push-back* și *pop-back*, coada poate executa operații de tip *pop-front* și *push-back*, iar *deque* poate executa operații de tip *push-front*, *push-back*, *pop-front* și *pop-back*.

Arbori de intervale

Această structură de date reprezintă un șir de numere pe care se pot executa operații de tipul:

- *Update(pos, x)* - schimbă valoarea numărului de la poziția *pos* cu *x*;
- *Query(left, right)* - află suma elementelor din intervalul *[left, right]*.



Arbore de intervale asociat șirului {1, 3, 5, 7, 9, 11}

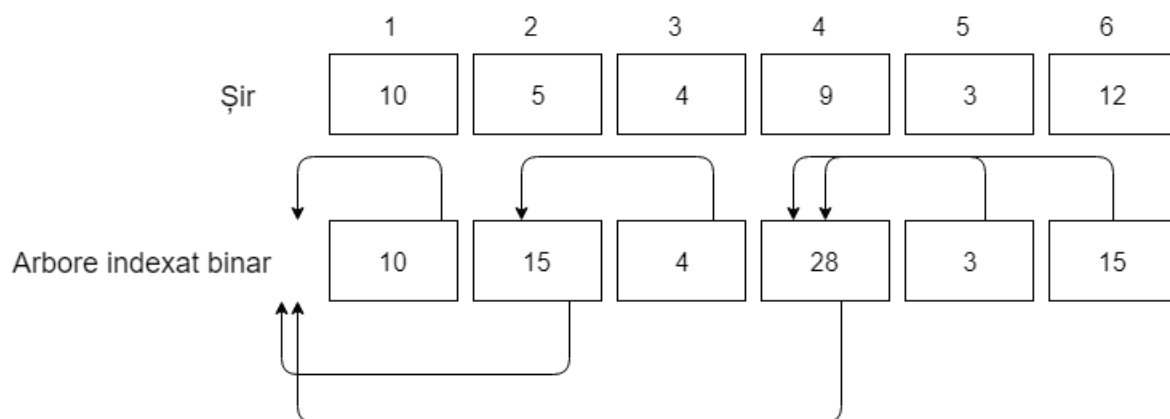
Figura 2-1 Arbore de intervale

Ambele operații se execută în timp logaritmice. Aceste operații sunt ale unui arbore de intervale clasic. În funcție de problema care trebuie rezolvată, aceste operații pot fi extinse. Există și arbori de intervale 2D, în care fiecare nod din arbore ține o altă structură de date, cum ar fi o trie.

Acest tip de arbore ține informația asupra unui interval de obiect. Ca și structură este asemănător unui arbore binar. Primul nod ține informații asupra întregului șir, iar apoi intervalul se înjumătățește și informații despre cele două subintervale se rețin în fii nodului.

Arbori indexați binar

Această structură de date este asemănătoare arborilor de intervale prin faptul că reprezintă un șir de numere și execută operații de tip $Update(pos, x)$ și $Query(left, right)$ în timp logaritm.



Fiecare muchie reprezintă părintele unui nod. Poziția parintelui este dată de $pos - \text{LeastSignificantBit}(pos)$

Figura 2-2 Arbori indexați binar

În practică arborii indexați binar sunt mai rapizi ca arborii de intervale, dar din cauza structurii lor nu pot face așa de multe tipuri de operații în timp eficient. De exemplu, arborii de intervale pot executa operația de *query* cu înmulțire pe un interval în $O(\log n)$, dar arborii indexați binar în $O(\log^2 n)$.

2.2. Arhitectura aplicației

Arhitectura aplicației poate fi împărțită în:

- arhitectura pe *front-end* (specifică ce componente de *Angular* există și cum interacționează una cu alta);
- arhitectura pe *back-end* (conține baza de date și tabelele acestora, rute și cereri);

2.3.1. Arhitectura pe partea de *front-end*

Componenta cea mai importantă și complexă din această aplicație o reprezintă cea care formează *view*-ul specific unui algoritm sau unei structuri de date. Pe lângă aceasta mai există alte componente, cum ar fi cea care se ocupă de pagina principală sau de înregistrarea și autentificarea unui utilizator, dar structura și acțiunile pe care le execută sunt standard.

Această componentă este formată, precum este prezentat în **Figura 2-1**, din patru componente mai mici:

- o componentă editor;
- o componentă de *logging*;

- o componentă de vizualizare;
- o componentă denumită *master*, care se folosește de celelalte trei și le sincronizează;

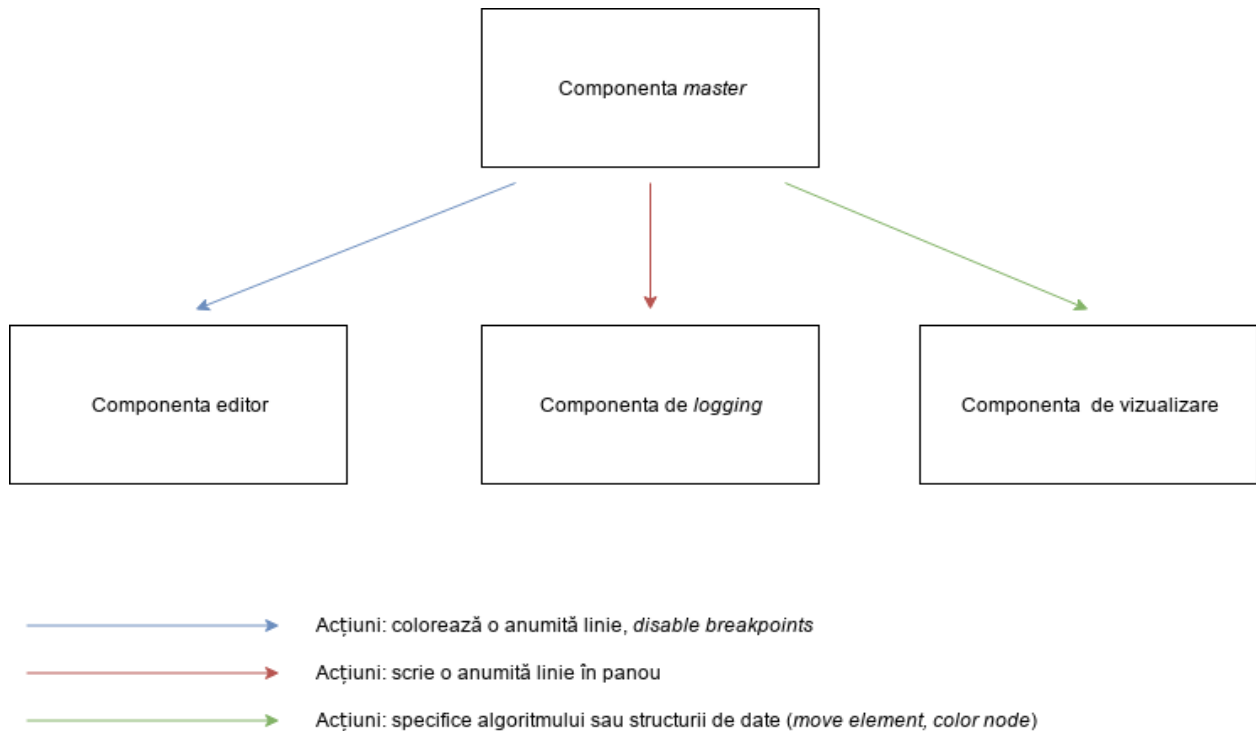


Figura 2-3 Componente Angular

Componenta *master*

Componenta *master* poate comanda celorlalte componente anumite acțiuni, sau poate obține date de la ele (de exemplu, componenta editor conține un *tab* în care utilizatorul poate scrie datele de intrare specifice problemei). Mai mult, această componentă conține și o clasă care rulează codul specific problemei cu datele de intrare.

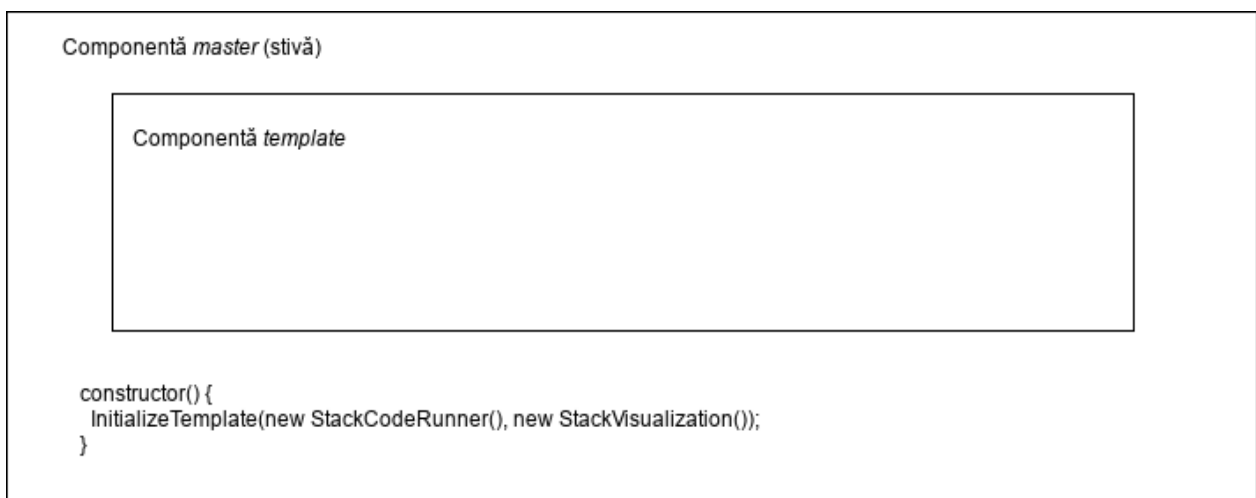


Figura 2-4 Master template

Pentru a ușura implementarea, deoarece există o listă vastă de algoritmi și structuri de date, s-a creat o nouă clasă template. Știm că toate paginile se comportă la fel din punct de

vedere al acțiunilor utilizatorului, singurele schimbări având loc în componenta de vizualizare și în clasa care rulează datele de intrare. Prin urmare, putem separa toată logica comună (cum se comportă butoanele de începere/oprire) și view-ul comun (conținând componenta editor, componenta de logging) într-o clasă template, iar clasa master să o conțină și să o inițializeze conform exercițiului pe care îl prezintă.

Componenta *master* are rolul de a executa anumite funcții ale componentelor copil și de a sincroniza codul cu vizualizarea. Clasa care rulează codul îi va trimite o listă cu liniile de cod care trebuie executate și ordinea lor, astfel știind ce operații să execute la saltul între *breakpoints*.

Componenta editor

Această componentă are rolul de a permite utilizatorului să vizualizeze codul care se execută și să insereze datele de intrare. Totodată, fiecare linie de cod care se execută are în stânga sa un *checkbox* care, dacă este activat, va valida acea poziție ca un *breakpoint*. Acest lucru înseamnă că, dacă codul va fi rulat, componenta va evidenția toate *breakpoint*-urile, inclusiv această poziție.

Breakpoints	Code	Comments
<input type="checkbox"/>	var stack: number[] = [];	<input type="text" value="declare stack"/>
<input type="checkbox"/>	for (var item of input) {	<input type="text"/>
<input type="checkbox"/>	if (item.operation == 'Push')	<input type="text" value="if operation is push"/>
<input type="checkbox"/>	stack.push(item.data);	<input type="text" value="push number to stack"/>
<input type="checkbox"/>	if (item.operation == 'Pop')	<input type="text" value="if operation is pop"/>
<input type="checkbox"/>	stack.pop();	<input type="text" value="pop item from stack"/>
<input type="checkbox"/>	}	<input type="text"/>

Figura 2-5 Componenta editor

Pe lângă un *checkbox*, fiecare linie are atribuită și un comentariu care este afișat în view-ul componentei de *logging*. Comentariul este editabil, utilizatorul putând scrie orice îi oferă o informație folositoare.

Componenta de logging

Această componentă are rolul de a afișa orice valoare de tip *string* existentă în model. Componenta *master* are rolul de a sincroniza componenta editor cu cea de *logging*. Astfel, această componentă determină ca, atunci când este configurat un nou *breakpoint*, comentariul asociat acelei linii de cod să fie adăugat modelului componentei de *logging*.

Scopul principal este de a ușura observarea de către utilizator a liniilor de cod parcurse în timpul execuției.

Componenta de vizualizare

Această componentă are rolul de a executa operații trimise de către componenta *master* și de a ajuta utilizatorul să înțeleagă codul și structura sa prin metode grafice.

Operațiile executate sunt specifice problemei (de exemplu, vizualizarea algoritmului *quick sort* conține operații precum *swap positions* sau *color pivot*). Operațiile se trimit în clasa care rulează codul pe datele de intrare ale utilizatorului. Nu este obligatoriu ca aceste operații să fie specifice unei linii de cod. De exemplu, operația de colorare a unui interval în algoritmul *merge sort* nu se găsește în algoritmul original, fiind o operație adițională pentru a augmenta utilitatea componentei.

2.3.2. Arhitectura pe *back-end*

O aplicație de tip web trebuie să conțină și partea de *back-end*, o parte esențială pentru datele unui utilizator și pentru securitatea aplicației. De obicei, există filtre de securitate și pe *front-end*, dar mereu trebuie să existe și pe *back-end* deoarece nu întotdeauna cererile către server vor veni dintr-o interfață grafică, ci pot veni și din linia de comandă. Astfel, filtrele de pe *front-end* nu vor mai avea folos.

Totodată, toate datele despre un utilizator sunt ținute *remote*, și pot fi accesate doar dacă se trimit credențialele corecte.

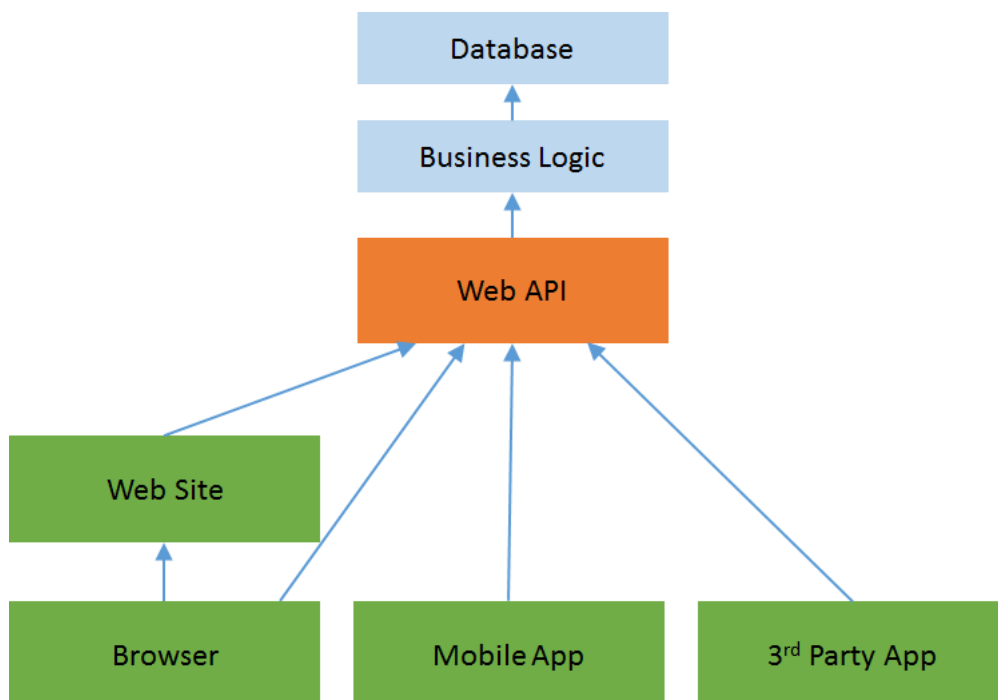


Figura 2-6 API pattern

Sursă: <http://bizcoder.com/the-web-api-business-layer-anti-pattern>

În general, crearea unei aplicații de tip *back-end* se rezumă la crearea unor rute care îndeplinesc anumite funcționalități. Pentru fiecare rută și verb HTTP (GET, POST, PUT,

OPTIONS), există *handler*-e. În funcție de rută, verbul HTTP și alte informații din cerere, aplicația *back-end* va procesa cererea și va întoarce un cod HTTP alături de informațiile cerute

Pentru a defini rute și metodele care se apelează pentru o rută, aplicația folosește *framework*-ul Express ([15]). Acest *framework* asociază unei perechi formate din rută și verb HTTP una sau multe funcții numite *handler*-e. Acest funcții nu trebuie să fie cele care să rezolve filtrarea cererii sau comunicarea cu baza de date, ci se pot crea mai multe nivele, cererea coborând până cel mai de jos nivel (*Database layer*) și apoi urcând până la *handler*-ul rutei, cel care returnează un răspuns.

Astfel, funcționalitățile vor fi modularizate și vor avea o structură mai simplă.

Aplicația asociată lucrării are ca scop oferirea unei identități utilizatorului (nume, profil) și adăugarea de caracteristici care îi vor eficientiza învățarea. Identitatea utilizatorului a fost introdusă deoarece a fost inspirată din realitate și ajută mai departe la diferențierea dintre utilizatori prin diferite caracteristici.

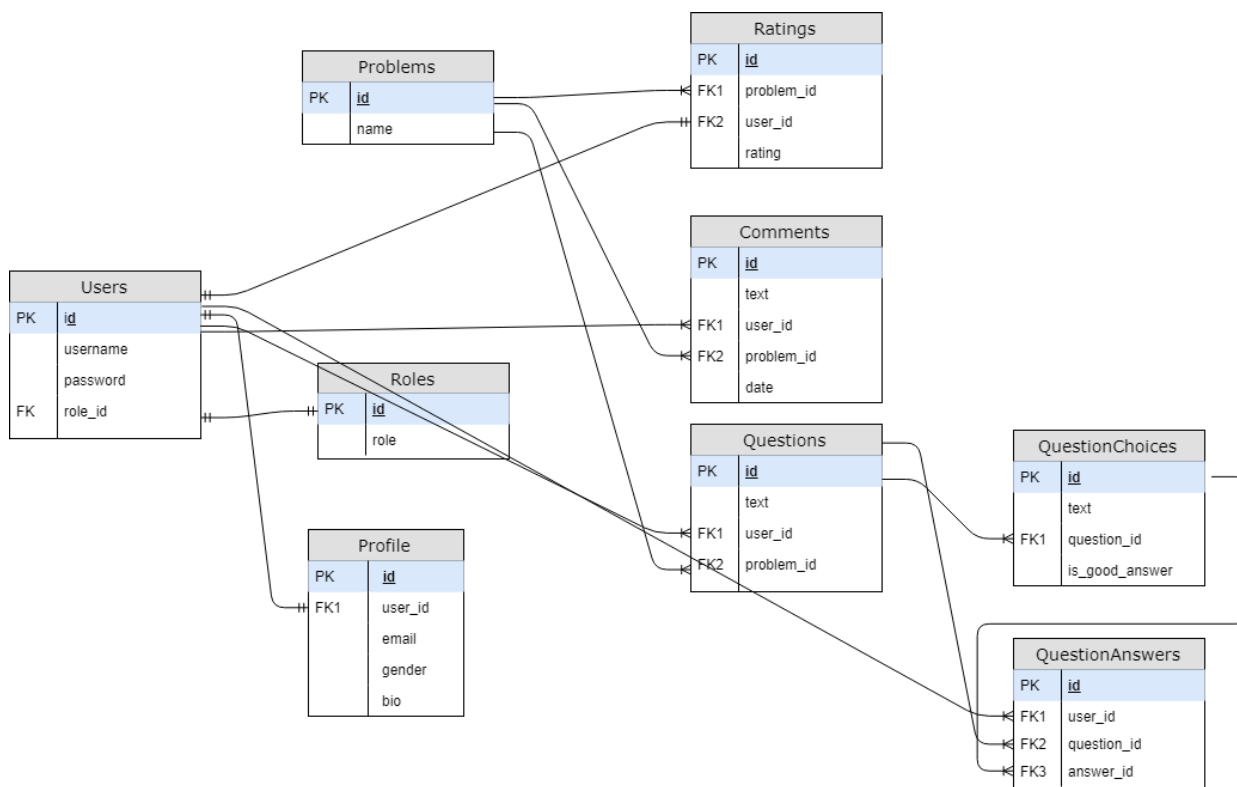


Figura 2-7 Diagrama bază de date

Alte caracteristici adăugate aplicației sunt:

- comentarii asupra unei exerciții;
- note de la 1 la 5 atribuite unui exercițiu;
- roluri pentru utilizatori (elev sau profesor);

- întrebări atribuite unui exercițiu la care utilizatorii cu rolul elev pot răspunde.

Comentariile și notele asociate unui algoritm sau unei structuri de date au scopul de a lăsa un utilizator să exprime utilitatea exercițiului și să ajute alți utilizatori, de exemplu, prin referințe la articole care oferă o mai bună explicație.

La conceperea aplicației, a fost luat în considerare și faptul că ea poate fi folosită la predarea în școli de către profesori. Prin urmare, utilizatorii au fost împărțiți în elevi și profesori.

Profesorii pot crea noi întrebări asociat unui exercițiu cu scopul de a testa un elev asupra cunoștințelor învățate. Întrebările sunt de tip grilă, iar fiecare elev are un număr de puncte de experiență. Dacă un elev va răspunde corect, un număr de puncte îi va fi adăugat experienței sale. Acest lucru va crește competitivitatea între utilizatori, sporind învățarea.

2.3. Simulator de *debugger*

Aplicația, în decursul rulării unui cod, evidențiază linia pe care o execută programul, în mod asemănător unui *debugger*. Pentru a implementa un *debugger* real trebuie interpretat codul și ținut un tabel cu valorile variabelor și o stivă pentru apelurile funcțiilor. Nu este o cerință imposibilă luând în considerare că majoritatea *browser*-elor îl au implementat, dar pentru scopul aplicației, acela de a sincroniza pe cât posibil operațiile unei linii de cod cu vizualizarea corespunzătoare, se poate implementa o versiune mai simplă.

Ca prim pas, se va executa tot codul pe datele de intrare, făcând o listă cu indicii liniilor care au fost parcurse pe acele date, în ordine. Utilizatorul își poate stabili ce linii vrea să fie evidențiate atunci când se va rula codul. Prin urmare, avem o listă cu toate liniile și ordinea în care vor fi parcurse, din care ne interesează doar anumiți indici. Acest lucru înseamnă că putem vizualiza această listă ca pe una de poziții marcate cu o valoare booleană: 0 (poziție în care programul nu se oprește); 1 (poziție în care programul se oprește).

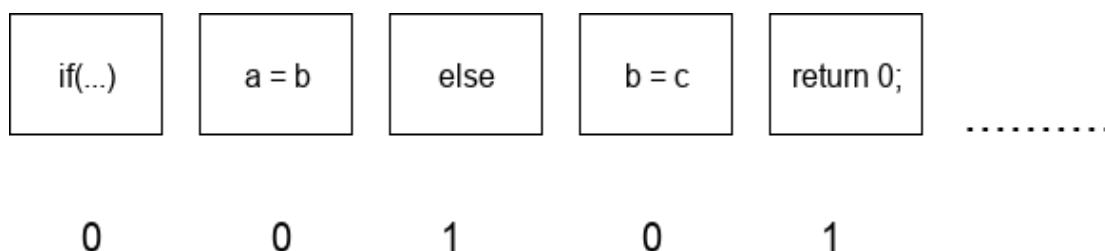


Figura 2-8 Vizualizare tehnică simulator

Al doilea pas este acela de a parcurge codul linie cu linie, dar oprindu-ne X milisecunde într-o linie marcată cu 1, X fiind *timer*-ul setat de utilizator în UI. Acest lucru se poate programa folosind funcția *window.setInterval()*, specifică *Javascript*.

Știind că parcurgerea codului linie cu linie trebuie să permită o sincronizare cu componenta de vizualizare specifică unui algoritm sau unei structuri de date, trebuie ca fiecărei linii să-i fie atribuită una sau mai multe operații. Atunci când se va trece peste acea linie, operațiile sunt trimise către componenta de vizualizare. Prin urmare, există anumite linii **speciale** care vor fi executate, dar care nu au o legătură cu codul inițial și nu au atribuite nicio valoare specială folosită la simularea *debugger*-ului

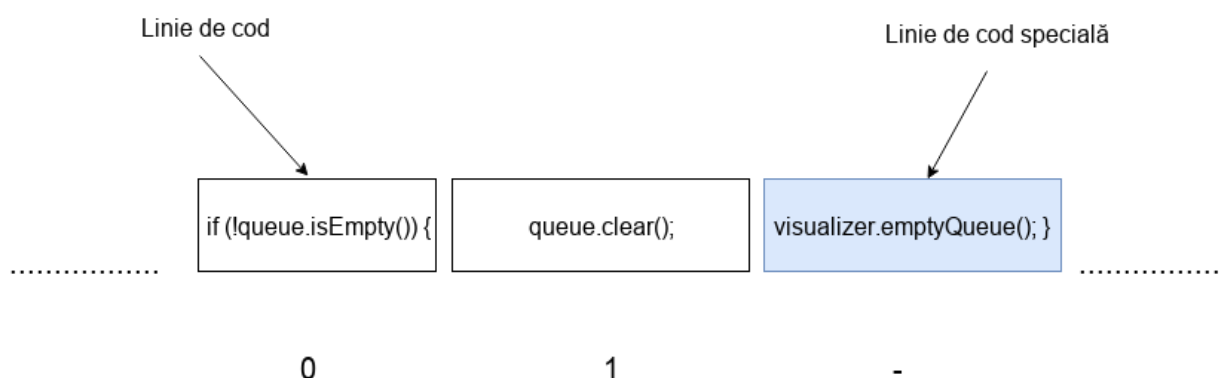


Figura 2-9 Sincronizare vizualizare

Aceeași metodă se aplică și în cadrul comentariilor asociate unei linii de cod. Fiecare linie de cod, reprezentată printr-un indice, o valoare booleană și operații care schimbă vizualizarea, mai are atașată o valoare de tip *string* care reprezintă un comentariu ce trebuie afișat în panoul de *logging*.

Astfel, codul care este interpretat atunci când are loc rularea acestuia cu datele de intrare editate de utilizator nu este același cu cel prezentat în componenta de *editor*. Este mai complex și trebuie să comunice cu alte componente și să facă alte procesări pe lângă executarea codului. Codul prezentat în *editor* are rol educativ și este echivalent cu cel care rulează în realitate, din punct de vedere al acțiunilor principale și operațiilor care schimbă vizualizarea. Utilizatorul nu trebuie indus în eroare de implementarea cu care s-a reușit a fi construit un *debugger* superficial, ci trebuie să vadă codul din execuțiul deschis și să înțeleagă legătura dintre cod și vizualizare.

3. Tehnologii utilizate

3.1. Javascript

Deoarece proiectul asociat acestei lucrări este o aplicație web, limbajul principal folosit, pe lângă HTML și CSS, este *Javascript*.

Javascript este un limbaj de programare orientat pe obiecte bazat pe conceptul prototipurilor. Totodată, este un limbaj dinamic, având următoarele caracteristici:

- obiectele pot dobândi noi proprietăți și funcții la *run-time*;
- se pot crea/rula *script*-uri prin ajutorul funcției *eval()*;
- *object introspection* (se poate examina tipul de date sau proprietățile unui obiect la *run-time*);

Prin urmare, *Javascript* este un limbaj flexibil și este ideal pentru o aplicație unde nu se cunosc datele cu care se va lucra dinainte (fiind o aplicație de tip *client-server*).

Javascript este asemănător limbajelor precum *C++* sau *Java*, având funcții, obiecte, tipuri de date, dar diferă prin alte componente cum ar fi obiectul *document* care este folosit pentru a comunica cu elementele de pe pagina ce formează DOM (*Document Object Model*).

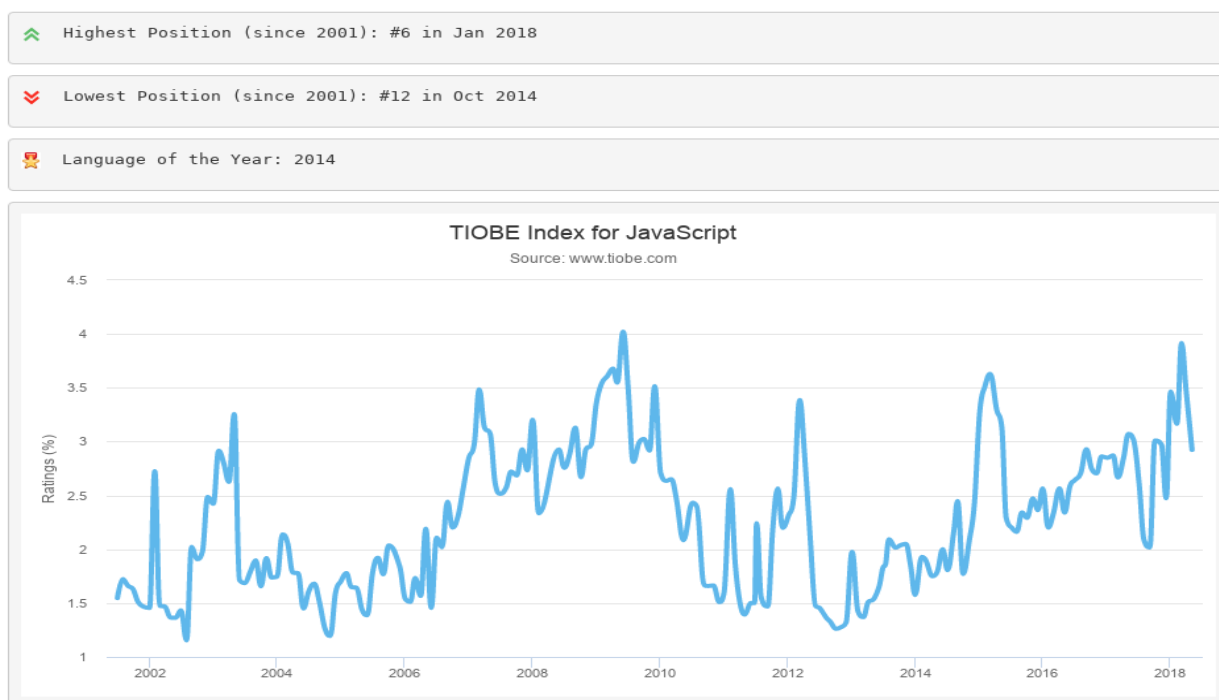


Figura 3-1 Javascript TIOBE index

Sursă: <https://www.tiobe.com/tiobe-index/javascript/>

În istoria web-ului, la început *site*-urile erau statice, utilizatorul având posibilitatea să le vizualizeze dar nu interacționa. Mai târziu, a fost introdus *Javascript* cu scopul de a combate acest dezavantaj.

Javascript rulează pe partea de client a unei aplicații web (în *browser*), care poate fi folosit pentru a programa comportamentul aplicației în cazul unui eveniment creat de utilizator (de exemplu, acțiuni provocate de *mouse* sau tastatură). Mai mult, prin ajutorul *browser*-ului, poate comunica cu alte calculatoare cu scopul transferului de informații, prin ajutorul apelurilor *HTTP* (*Hypertext Transfer Protocol*).

Javascript, în combinație cu HTML/CSS și mediul oferit de un browser (*cookies*, *browser storage*), formează un grup matur și puternic de tehnologii, comunitatea fiind vastă și de ajutor. Acest lucru se poate deduce și din poziția pe care o are *Javascript* față de alte limbaje de programare în utilizarea sa de programatori.

Se observă din graficul obținut de cei de la *TIOBE* că *Javascript*, chiar și după douăzeci de ani de folosire în lumea dezvoltării aplicațiilor web, rămâne în top zece cele mai folosite limbaje de programare de-a lungul globului.

3.2. Angular

Angular ([9]) este succesorul popularului framework *AngularJs*.

AngularJs este un framework bazat pe *Javascript*, menținut în principal de Google, și permite dezvoltarea de *single-page applications*. Principalele obiective de design sunt:

- să decupleze manipularea *DOM*-ului (*Document Object Model*) față de logica aplicației;
- să decupleze *frontend*-ul de *backend* (are rolul de a împărți sarcinile separat, acceptând ideea lucrului în paralel);
- să furnizeze o structură coerentă pe măsură ce se vor dezvolta diferitele module ale aplicației (UI, logica de business, testarea);

AngularJs implementează *pattern*-ul *MVC* (*Model-View-Controller*) pentru a separa prezentarea, datele și logica componentelor. Folosindu-se de *DI* (*Dependency Injection*), aduce tradiționalele servicii *server-side*, cum ar fi *controller*-ele dependente de *view*-ul utilizatorului, în *client-side*. Astfel, reduce traficul de date necesare serverului pentru a injecta serviciile clientului la un anumit moment.

Totodată, *AngularJs* citește codul HTML care formează pagina (*view*) ce poate conține *tag-uri* atribut specifice. Acestea sunt interpretate ca directive pentru a forma un *binding* între elementele de *input* și *output* ale paginii și un model reprezentat de un cod scris în *Javascript* standard. Acest model aparține unui *controller*, o funcție scrisă cu ajutorul bibliotecii oferite de *AngularJs*.

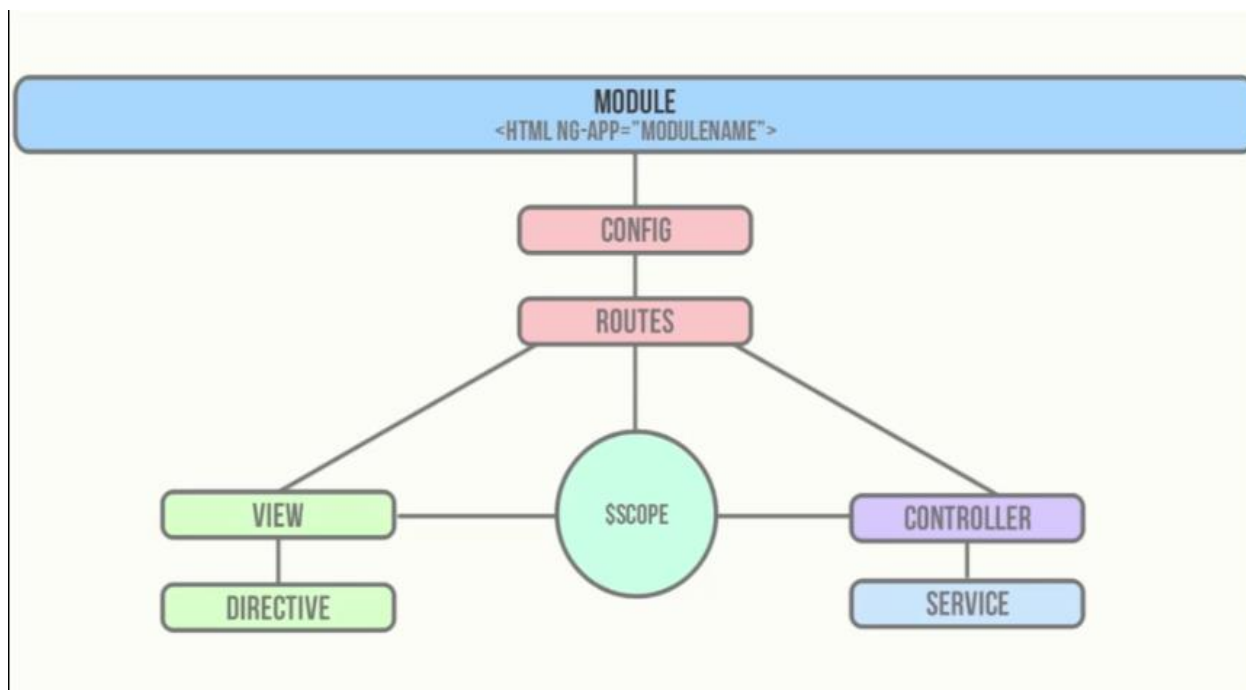


Figura 3-2 Structura *AngularJs*

Sursă: <https://kriyance.wordpress.com/2014/04/05/angular-js-application-structure/>

Directivele menționate mai sus au rolul de a extinde codul HTML și de a ușura interacțiunea dintre *view* și *controller*. *AngularJs* oferă multe directive pe care un programator le poate folosi, câteva dintre acestea fiind:

- *ng-app* (inițializează componenta de bază denumită *root component*);
- *ng-controller* (specifică o clasă de tip *controller* care poate să evalueze expresii HTML);
- *ng-model* (stabilește un *two-way data-binding* între elementul HTML și variabila model);
- *ng-repeat* (instanțiază câte un element pentru fiecare *item* din listă care poate fi folosit în *view*).

O altă componentă de structurare a *view*-ului pe care o creează *AngularJs* este *scope tree*.

Inițializând elementele HTML prin directiva *ng-controller*, se creează un element numit *\$scope*, care este intermediarul comunicării dintre *view* și *controller*.

Scope Tree

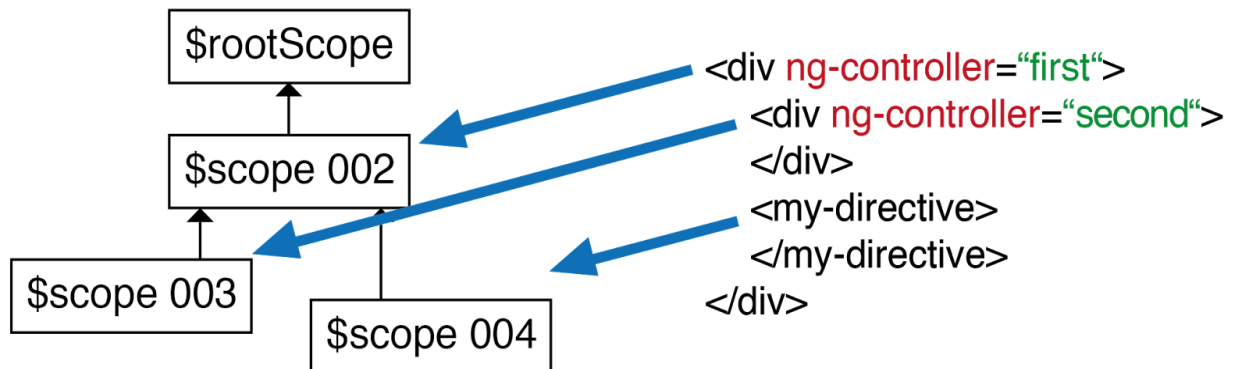


Figura 3-3 *AngularJs scope tree*

Sursă: <http://www.angularjstutorialsneg.com/2016/07/angularjs-scopes.html>

Acest arbore de *scope*-uri se comportă asemănător moștenirii în contextul limbajelor orientate pe obiecte. Dacă un element căruia i s-a atribuit un *scope* este un descendent al unui element DOM cu un *scope* diferit, *scope*-ul descendent moștenește modelul din *scope*-ul strămoș.

Angular se bazează pe caracteristicile și flexibilitatea oferite de *AngularJs*, oferind un pachet complet (*routing*, componente, module, *dependency injection*) și posibilitatea de a crea aplicații pentru platforme mobile, dar scriind *Javascript*. Diferența majoră între cele două *framework*-uri este dată de faptul că *Angular* renunță la *scope tree*, lăsând la îndemâna programatorilor doar posibilitatea de a crea componente cu *scope* izolat. Problema comunicării între componentele cu *scope* izolat se rezolvă prin introducerea unor directive numite decoratori (*@Input*, *@Output*).

Angular este tânăr în comparație cu *AngularJs* (2016 versus 2010) dar s-a înălțat rapid în rândul proiectelor de pe *Github* ([10]), crescând în popularitate an după an.

Majoritatea avantajelor față de *AngularJs* sunt prezentate în cele ce urmează.

- ***Typescript*:** este un limbaj *statically-typed* și este superset al lui *Javascript*. *Angular* impune utilizarea *Typescript* pentru a crea o aplicație web. Unul dintre marile avantaje ale lui *Typescript* este faptul că, fiind tradus în *Javascript*, se poate configura să fie compatibil cu *browser*-e vechi (cum ar fi Internet Explorer 8).
- ***Angular-CLI*:** este un utilitar pentru linia de comandă care ajută la dezvoltarea de aplicații în *Angular*. Are comenzi pentru generare de componente sau directive, pentru testare și are inclus un utilitar, numit *Webpack*, pentru împachetarea diferitelor biblioteci folosite în aplicație.

- **Modularitate:** *Angular* se bazează mult pe modularitate deoarece oferă o logică mai bună și reduce din dimensiunea aplicației. Totodată este ușor de întreținut și oferă extensibilitate.

3.3. NodeJs

NodeJs ([11]) este un mediu de dezvoltare *server-side* pentru *Javascript*, construit cu ajutorul motorului V8 al *browser*-ului *Chrome*.

NodeJs permite dezvoltarea de servere web și aplicații de *networking* folosind *Javascript* și o colecție de module de bază (sistem de fișiere; rețele - DNS, HTTP; criptografie). Pentru instalarea modulelor, se folosește *npm*, un utilitar de administrare a pachetelor *Javascript*, ce conține un registru de pachete a contribuitorilor și toată istoria versiunilor.

NodeJs procesează cererile de computație într-o buclă, procesul fiind numit *Event Loop*.

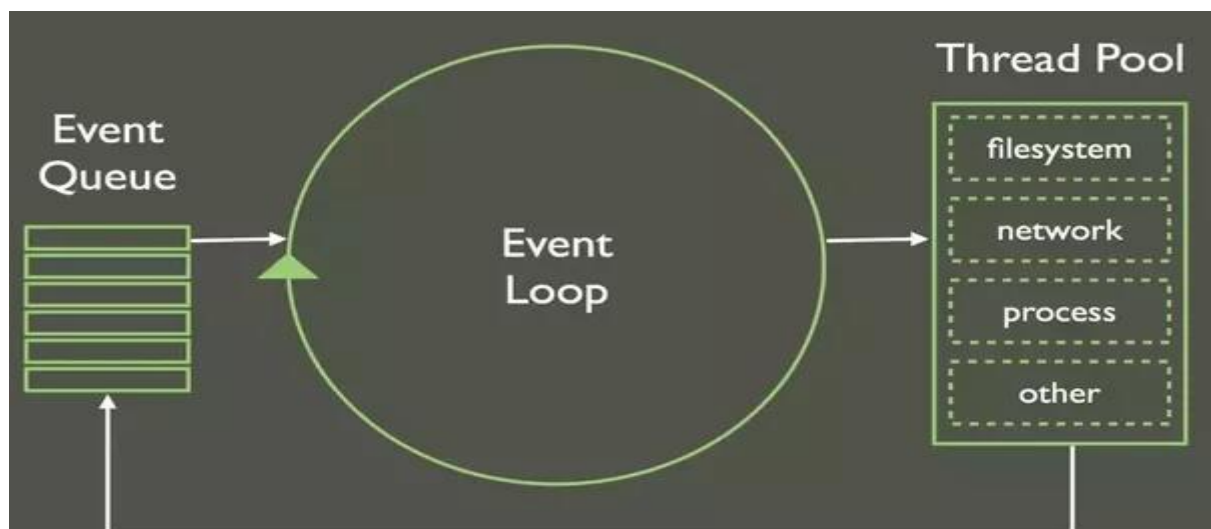


Figura 3-4 Event Loop

Sursă: <https://www.quora.com/In-Node-js-how-does-the-event-loop-work/answer/Jeff-Meyerson>

În principiu, există un singur fir de execuție, folosind operații de I/O (*Input/Output*) non-blocante și permițând suportarea a sute de conexiuni concurente. Totodată, se elimină consumul de computație al *context-switching*-ului al unui mediu în care se folosesc mai multe fire de execuție.

Acest fir de execuție principal are rolul de a administra și repartiza cererile către un *thread-pool*, o mulțime de fire de execuție care se ocupă de operații de I/O asincrone.

Un dezavantaj al acestui model de computație este existența unor evenimente costisitoare care trebuie executate de firul de execuție principal și care blochează coada de evenimente.

NodeJs are câteva avantaje importante pentru alegerea dezvoltării unui server web cu această tehnologie:

- *npm*: inițializarea unui proiect se folosește prin comanda *npm install*, care citește un fișier ce descrie pachetele necesare proiectului și versiunile lor. Astfel, toate bibliotecile și datele necesare sunt aflate în directorul proiectului, fiind izolat și neexistând conflicte cu pachete globale.
- *Event Loop*: în *Javascript* toate operațiile asincrone (citirea datelor de pe *hard disk*, așteptarea datelor aduse de server) se folosesc de *callbacks* pentru a informa firul de execuție principal de terminarea executării. Astfel, nu există *race conditions* sau *deadlocks* ca în cazul mediilor cu mai multe fire de execuție.
- *Javascript*: fiind un limbaj folosit atât pe *front-end* cât și pe *back-end*, va exista o comunicare mai bună între echipele din fiecare categorie, ambele având abilități asemănătoare.

3.4. PostgreSQL

PostgreSQL ([12]) este un sistem de baze de date relaționale. Principalele obiective ale unui astfel de sistem este să păstreze datele securizate și să returneze datele cererilor care vin de la alte aplicații, cu performanțe bune.

PostgreSQL prezintă caracteristici precum:

- replicarea datelor pe mai multe noduri;
- indecși care îmbunătățesc performanțele cererilor;
- tipuri de date (*date/time*, șiruri de caractere, binar) sau tipuri care pot fi create de utilizator.

Mai mult, *PostgreSQL* are numeroase biblioteci pentru diferite limbaje de programare, cu scopul de a ușura munca depusă de programatori pentru a comunica cu acest sistem. Aceste lucruri îl favorizează ca soluție de stocare față de alți competitori.

3.5. Biblioteci

3.5.1. Vis.js

Această bibliotecă ([14]) este proiectată să aibă o utilizare ușoară, să manipuleze multe date dinamice și să permită interacțiuni cu aceste date. Câteva dintre componentele principale prin care se pot crea vizualizări cu ajutorul acestei biblioteci sunt:

- *Network* (grafuri);
- *Timeline* (permite vizualizarea evoluției datelor de-a lungul timpului);
- *Graph2d/Graph3d* (*charts*).

Proiectul este *open-source*, fiind menținut de mulți contributori. Scopul principal al acestei biblioteci este de a ușura crearea de vizualizări *data-drive* și de a le manipula programatic.

Aplicația asociată acestor lucrări folosește doar vizualizarea tip *Network* pentru a exercițiile despre grafuri. Acest tip de vizualizare îți oferă multe funcționalități, precum interacțiuni prin *mouse* cu graful sau funcții de colorare a nodurilor și a muchiilor. Mai mult, oferă și posibilitatea de a apela funcții pentru anumite evenimente executate de utilizator (*drag*, *click*) cu scopul de a crea o experiență interactivă.

4. Descrierea aplicației

5. Concluzii

O aplicație de *e-learning* nu este un lucru simplu de dezvoltat deoarece audiența este diversă, iar unii utilizatori s-ar putea să învețe mai eficient dacă, de exemplu, li se arată direct algoritmul în pseudocod și o descriere succintă, fără alte explicații/imagini. Prin urmare, aplicația trebuie să fie flexibilă și să satisfacă nevoile tuturor clienților.

În decursul creării aplicației asociate acestei lucrări și alegerii caracteristicilor care vor ajuta utilizatori să învețe mai ușor, am întâlnit anumite incompatibilități între caracteristicile alese și am compus idei care sporesc flexibilitatea aplicației, dar adaugă un grad de complexitate și sfidează scopul inițial.

Un prim exemplu de incompatibilitate ar fi posibilitatea de a putea seta un *breakpoint* pentru o anumită linie din codul exercițiului, cu un anumit timp de salt între linii. Știind ca aplicația trebuie să sincronizeze operațiile care se execută în cod și reprezentarea lor în componenta de vizualizare, se pune problema vitezei de animație a diferitelor elemente din vizualizare. Această viteză trebuie să fie încadrată într-un interval, deoarece, dacă este prea mare, utilizatorul nu va putea observa ce se întâmplă, iar dacă este prea mică, nu va oferi o experiență plăcută.

Astfel, trebuie ca viteza de animație să fie fixată la o valoare acceptabilă pentru utilizator, sau să poată fi editabilă pentru a satisface nevoile împărțite. Indiferent de ce valoare se va alege, viteza de salt între *breakpoint*-uri și viteza de animație sunt două lucruri separate, și nu va exista o sincronizare reală între cele două. De exemplu, dacă orice *breakpoint* are o operație care va schimba vizualizarea și viteza de salt este de 100 milisecunde, iar viteza animației este de 300 milisecunde, în cazul în care se vor parcurge 13 *breakpoint*-uri, rularea codului se va termina în 1,3 secunde, dar rularea animației din vizualizare în 3,9 secunde. Aici se presupune că liniile care vor fi parcurse între *breakpoint*-uri nu au operații adiționale, dar nu întotdeauna există acest caz.

Mai mult, nu orice implementare a componentei de vizualizare care primește operații de animat este corectă, luând în considerare faptul ca operațiile trebuie făcute în ordinea primirii și secvențial pentru a le executa corect. De aceea, implementarea aleasă pentru această problemă se folosește de o coadă, o structură de date care este existentă și în lista de structuri de date ale aplicației.

Operațiile care ajung la componenta de vizualizare sunt puse într-o coadă deoarece ar putea exista operații care au ajuns mai devreme dar nu au fost terminate. Există o funcție care verifică la un anumit interval de timp dacă coada este goală sau nu, iar dacă nu este și nu există alta animație în executare, o va executa pe aceasta.

Astfel, animațiile se vor executa în ordinea primirii și nu vor exista doua animații care se execută în același timp.

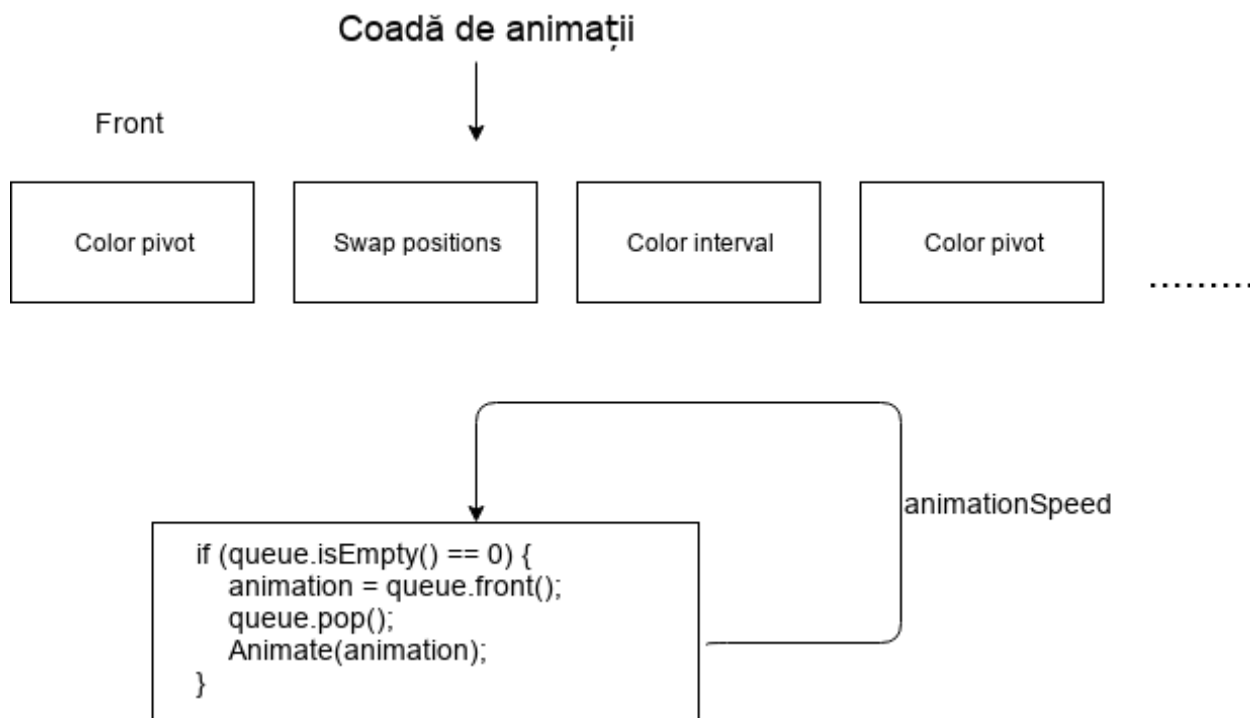


Figura 5-1 Sincronizare animații

O idee care ar putea spori flexibilitatea aplicației ar fi ca utilizatorul să poată scrie cod în componenta editor, în loc să fie executat un cod predefinit. Scopul din spatele acestei idei este de a lăsa utilizatorul să-și dea seama de ce un anumit cod nu execută ceea ce dorește, vizualizând operațiile făcute. Asta ar însemna că aplicația trebuie să expună un *API* (*Application Programming Interface*) utilizatorului, pe care îl poate folosi în codul scris. Acest API reprezintă operațiile care se vor executa în componenta de vizualizare.

Totuși, această funcționalitatea sfidează scopul inițial cu care a fost scrisă aplicația, și anume, de a învăța orice tip de programator, simplu și eficient, anumiți algoritmi și structuri de date. Poate un utilizator nu va știe să programeze sau poate ca operațiile expuse de către API nu îi vor satisface nevoile. În plus, această funcționalitate ar putea deveni cu totul o altă aplicație. De exemplu, o aplicație care să expună funcții de colorare cu elemente și acțiuni folosite des în prezentări și *demo-uri*. Profesorii ar putea folosi aceasta aplicație pentru prezentarea anumitor concepte teoretice prin imagini și cod scris care folosește acel API.

Există multe alte caracteristici care pot fi adăugate aplicației, doar că există incompatibilități între ele sau există idei foarte greu de implementat și poate nefolositoare. De exemplu, opțiunea de a afișa un comentariu pentru fiecare *breakpoint* parcurs de către simulator nu are o utilitate mare pentru un utilizator. Comentariul asociat unei linii nu are niciun *binding*

cu variabilele din program astfel încat nu se poate afla nimic despre starea variabilelor la acel pas. Comentariile sunt statice, afisând același text mereu.

Desigur, există posibilitatea de a face un *binding* și a afișa valorile diferitelor variabile vizibile în acea linie, dar nu este simplu de implementat.

Un set de caracteristici destul de complete și compatibile pentru acest tip de aplicație sunt regăsite în aplicația Visualgo ([13]). Referitor la timpul cu care se rulează codul, există doar un *slider* care reprezintă viteza cu care se rulează liniile de cod pentru un anumit input (asemanător soluției care simulează un *debugger*). Această soluție scapă problema sincronizării prezentată mai sus. Mai mult, comentariile nu sunt editabile, dar au informație folositoare despre valoarea variabilelor la un anumit pas și descriu operațiile făcute la acea linie de cod.

În concluzie, aplicația asociată acestei lucrări reprezintă o încercare în găsirea soluției problemei de învățare a algoritmilor și structurilor de date prin vizualizări. Nu va exista o soluție perfectă din cauza numeroaselor caracteristici care ar putea fi implementate dar care, ori vor crește complexitatea, ori au o utilitate scăzută. Totuși, în această lucrare au fost evidențiate caracteristici importante pentru găsirea unei soluții la această problemă și o aplicație web care reprezintă una dintre posibilele soluții.

Bibliografie

- [1] D. E. Knuth, *The Art of Computer Programming*, Volume 1: Fundamental Algorithms, Third Edition, Addison Wesley, **1997**
- [2] D. E. Knuth, *The Art of Computer Programming*, Volume 3: Sorting and Searching, Second Edition, Addison Wesley, **1998**
- [3] S. Even, *Graph Algorithms*, Second edition, Cambridge University Press, **1979**
- [4] M. Frisbie, *Angular 2 Cookbook*, Packt Publishing, **2017**
- [5] S. Powers, *Learning Node*, O'Reilly Media, **2012**
- [6] „PathFinding.js visualization”. Available: <http://qiao.github.io/PathFinding.js/visual/>
- [7] „Data Structure Visualizations”, University of San Francisco.
Available: <https://www.cs.usfca.edu/~galles/visualization/Algorithms.html>
- [8] „CsAcademy Applications”. Available: <https://csacademy.com/>
- [9] „Angular”, Google. Available: <https://angular.io/docs/>
- [10] „Github”, Github Inc. Available: <https://github.com/>
- [11] „NodeJs”, Joyent. Available: <https://nodejs.org/en/docs/>
- [12] „PostgreSql”, PostgreSQL Global Development Group.
Available: <https://www.postgresql.org/docs/>
- [13] „Visualgo”. Available: <https://visualgo.net/en>
- [14] „Vis.js”. Available: <http://visjs.org/>
- [15] „Express”. Available: <https://expressjs.com/en/4x/api.html>