Arbori BINARI de formă ARBITRARĂ

Adrian Vasilescu

Acest articol prezintă modul în care pot fi efectuate eficient diferite operații folosind arborii binari. Vor fi prezentate diverse strategii cu ajutorul cărora se pot îmbunătăți performanțele algoritmilor care folosesc această structură de date și care, se pare, mai au încă multe "secrete".

Introducere

Procesarea eficientă a datelor reprezintă una dintre preocupările majore din domeniul programării si administrării structurilor de date, în mod special în cadrul aplicațiilor în care volumul de informație prelucrată și stocată este ridicat.

În acest context, căutarea si localizarea eficientă de date prezintă un interes aparte, știut fiind faptul că aceste operații sunt costisitoare din punct de vedere computațional și în anumite cazuri chiar prohibite din cauza accesului lent pe disc. Astfel, o căutare de informații într-o structură de date ar putea necesita vizitarea și verificarea identității unui volum foarte mare de informație, această operație devenind cu ușurință o redutabilă sursă de ineficiență computațională.

Procesul de căutare poate fi însă îmbunătățit prin organizarea informației în structura de date, astfel încât lungimea traseelor de căutare să fie drastic redusă și optimizată cât mai mult posibil. Sunt cunoscute diferite strategii de îmbunătățire a performanței de căutare în diferite tipuri de structuri de date, strategii ce speculează în mod eficient particularitățile specifice ale aplicației considerate [1].

În cazul structurilor de date pentru care ordinea de introducere a datelor nu este cunoscută apriori și în care aceste date nu pot fi ordonate în vreun fel, structura care oferă un mecanism de căutare rapidă este arborele binar echilibrat (balansat).

În acest articol sunt prezentate considerentele teoretice si detaliile de programare ale acestui tip de arbore binar de formă arbitrară în cazul cărora inserarea de date se face pe baza unor numere de prioritate obținute de la un generator de numere arbitrare.

Accentul este pus pe implementarea modulară a structurii de ***arbore binar de formă arbitrară*** într-o manieră ce exclude accesul la date bazat pe indexare cât și pe utilizarea blocurilor de memorie continuă și mărime fixă, așa numiții ***arrays*** care nu permit alocarea dinamică a spațiului de memorie.

În încheierea articolului, o analiză a complexității algoritmului va demonstra propietatea arborelui binar de formă arbitrară de a fi echilibrat.

Structura de arbore binar

Arborele binar este o structură de date multinivel, reprezentată de o colecție de noduri, având un nod ***rădăcină*** și doi subarbori, unul ***stâng*** și altul ***drept***, conectați la nodul rădăcină. Astfel, arborele binar reprezentat în figura 1a) este compus din patru noduri interne (reprezentate prin cercuri) și cinci noduri externe (reprezentate prin pătrate), aceștia din urmă corespunzând unor subarbori vizi. În figura 1b), nodurile externe nu au mai fost reprezentate, în general aceste noduri fiind lipsite de o reală importanță practică sunt ocupate de pointerul `NULL`, pentru care nu este alocat spațiu de memorie.

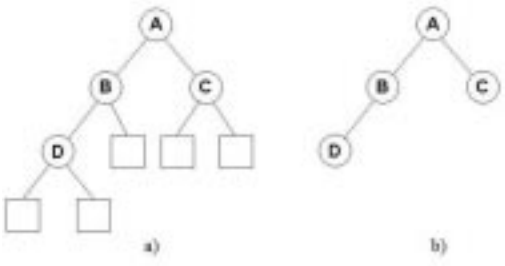

Figura 1. Structura de arbore binar

O clasificare și mai convențională a nodurilor într-un arbore se poate obține pe bază genealogică. Astfel, nodul *A* din figura 1 este ***ancestor*** pentru toate nodurile situate în subarborii săi care sunt ***descendenți*** ai săi. De asemenea, nodul *A* este ***părinte*** pentru nodurile situate pe nivelul ierarhic următor, coborând în arbore, și care sunt legate direct de nodul *A*. În figura 1, nodurile *B* și *C* sunt ***fii*** ai nodului părinte *A*.

Este de notat faptul că traseul de la un nod ancestor n_1 către descendenții săi n_k este unic, traseul fiind determinat de secvența de noduri unică n_1, n_2, ..., n_k în care nodul n_i este părinte al nodului n_{i+1} pentru $i = 1, 2, ..., k - 1$. Prin urmare, lungimea traseului de vizitare este determinată de numărul de legături dintre nodurile aparținând traseului (k - 1 pentru nodul n_k). În figura 1a), traseul de la *A* la *D* are lungimea 2, fiind compus din secvența de noduri *A*-*B*-*D*. Poziționarea unui nod în arbore se face prin definirea a doi scalari complementari denumiți în mod sugestiv ***adâncimea*** și ***înălțimea*** nodului în arbore. Astfel, adâncimea nodului n_k corespunde lungimii traseului de la rădăcina arborelui până la nod care este k - 1, în timp ce înălțimea aceluiași nod n_k este reprezentată de lungimea maximă a traseului având ca rădăcină nodul însuși și terminând la nodul extern cel mai îndepărtat în subarbore.

Căutarea și localizarea datelor

Pentru ca operațiile de căutare și localizare să fie eficiente, datele în arborele binar trebuie aranjate în așa fel încât lungimea traseelor de căutare sa fie cât mai mică posibil, lucru ce poate fi realizat prin gruparea nodurilor pe niveluri ierarhice pe cât posibil mai apropiate de rădăcina arborelui.

În general, inserarea seturilor de date în arbore se face pe baza unui criteriu de comparație binar, astfel încât seturile de date ale căror valori de comparație respectă criteriul impus să fie direcționate în subarborii stângi. Similar, seturile de date ale căror valori de comparație nu satisfac criteriul impus sunt direcționate în subarborii drepți, în final setul de date introdus în arbore ocupând poziția unui nod extern.

Este de remarcat faptul că invocarea criteriului de comparație se face pentru fiecare nod din traseu, aceasta constituind o sursă de calcul intensiv. În acest aranjament al datelor, efortul de căutare și localizare este considerabil redus prin faptul că aceste operațiuni sunt efectuate numai într-un anumit sector al arborelui, în subarborii succesivi stângi sau drepți, după caz.

Totuși, aranjamentul binar descris mai înainte nu este optim, deoarece este dependent de ordinea introducerii datelor în arbore. Astfel, în cazul cel mai defavorabil pentru care seturile de date sunt ordonate după valoarea lor de comparație și introduse în arbore în această ordine, arborele binar obținut degenerează într-o listă liniară în care procesul de căutare devine costisitor, dacă nu impropriu.

Pentru a evita astfel de situații defavorabile operațiilor de căutare ce sunt caracteristice arborilor binari neechilibrați sau nebalansați, se impune ca inserarea seturilor de date să se efectueze în așa fel încât procesul de formare al arborelui să devină independent de ordinea seturilor de date pe baza cărora se construiește arborele.

Prin această strategie se urmărește ca adâncimea arborelui să fie drastic redusă, prin aceasta micșorându-se lungimea traseelor de căutare în arbore și prin urmare a efortului computațional de căutare de date. Arborele binar este considerat echilibrat sau balansat dacă lungimile traseelor de la rădăcină până la nodurile externe sunt aproximativ egale. În cazul ideal, traseele complete au lungimi egale și arborele binar este considerat perfect echilibrat.

Construirea unui arbore binar

O metodă eficientă care asigură formarea arborelui în totală independență de ordinea de introducere a seturilor de date este aceea de a utiliza numere arbitrare ca valori de comparație necesare construcției arborelui. Aceste numere arbitrare constituie prioritățile de intrare atribuite seturilor de date și pot fi obținute de la generatorul de numere arbitrare al compilatorului folosit în prelucrarea codului sursă.

Astfel, fiecărui set de date i se atribuie un număr de prioritate arbitrar pozitiv și subunitar, introducerea datelor în arbore efectuându-se pe baza numerelor de prioritate atribuite în locul valorilor de comparație și folosind același criteriu de comparație binar, astfel încât prioritatea unui set de date să nu fie mai mare decât prioritățile seturilor de date descendente.

În acest mod, regula de bază a arborelui binar rămâne în efect, doar forma sa fiind arbitrară și controlată de numerele de prioritate arbitrare. Din acest motiv, arborele binar descris mai înainte este denumit, în mod sugestiv, ***arbore binar de formă arbitrară***.

Pentru ca acest tip de arbore să fie robust este suficient ca numerele de prioritate arbitrare să fie unice, iar ordinea lor de generare să fie neliniară și aciclică.

Din acest punct de vedere, generatorul de numere `rand()` al compilatorului C++ corespunde scopului arborelui binar de formă arbitrară.

Informația care trebuie stocată este organizată în blocuri de memorie, așa-numitele obiecte care la rândul lor sunt instanțe ale claselor prototip, care nu sunt altceva decât șabloane special proiectate în cadrul aplicației curente după care sunt generate obiectele propriu-zise.

Așadar, o structură de date poate fi concepută ca o colecție de blocuri de memorie pentru stocarea informațiilor și un set de operațiuni special proiectate pentru inserarea, modificarea și accesarea informației stocate.

În acest mod, separarea algoritmului de structura sa de date facilitează un nivel înalt de abstractizare în proiectarea și implementarea eficientă a oricărei operațiuni în cadrul aplicației. Acest proces stă la baza programării modulare, exprimată în termeni de clase prototip și instanțe ale acestora (obiecte), și al cărei scop principal constă în reducerea complexității de exprimare a algoritmilor pe bază de eficiență computațională.

Avantajele programării modulare sunt diverse și incontestabile, conceptul de clasă prototip oferind un înalt grad de reutilizare a lor în cadrul altor aplicații similare cu un efort de programare minim. De asemenea, clasele prototip pot fi extinse sau alterate fără a fi necesară intervenția în structura internă a programului sursă. Astfel, unul din scopurile principale ale activității de producere de *software*, acela de a permite extinderea și menținerea operativă a programelor pe perioade lungi de timp, este îndeplinit.

Primul pas în implementarea arborelui binar de formă arbitrară este de a defini un set abstract de date format din clase prototip cu ajutorul cărora structura de arbore să fie exprimată mai mult la nivel conceptual și mai puțin în termeni referitori la detalii de programare.

Procesul de identificare și formulare a claselor prototip este complex și de obicei repetitiv necesitând câteva cicluri de analiză profundă a detaliilor problemei și formulare ierarhică a algoritmilor.

În același timp, trebuie avut în vedere și aspectul eficienței computaționale care, ignorat, poate duce la dificultăți neașteptate în extinderea și exploatarea economică a programului.

O fragmentare excesivă a codului sursă în obiecte conduce inevitabil la un mare număr de apeluri de funcții și o organizare internă a programului deosebit de complexă, acestea fiind recunoscute ca veritabile surse de ineficiență.

Așadar, procesul de proiectare orientată-obiect și implementare eficientă a modelelor computaționale este unul complex și de durată, necesitând multiple revizuiri și completări.

În ceea ce privește implementarea arborelui binar de formă arbitrară descris anterior, aspectul eficienței computaționale a fost compromis într-o foarte mică măsură numai pentru a oferi o mai bună claritate codului sursă, acesta fiind și principalul scop al actualei versiuni.

O primă entitate abstractă ce poate fi ușor identificată este aceea de ***nod*** al arborelui. Astfel, este creată o clasă denumită `TreeNode`, conținând un element membru denumit `value` și doi pointeri denumiți `_lchild` și `_rchild`, care reprezintă legăturile nodului către cei doi descendenți ai săi.

Prin intermediul pointerilor `_lchild` și `_rchild`, legăturile dintre nodurile arborelui sunt stabilite în mod automat de către constructorul clasei prototip `TreeNode` fără a necesita nici un fel de operații de indexare sau tabelare.

Destructorul `~TreeNode()` este responsabil pentru eliberarea spațiului de memorie ocupat de obiectul nod cât și de descendenții acestuia.

Pentru a asigura utilizarea eficientă a arborelui, clasele prototip vor fi implementate prin intermediul mecanismului `template<>`, astfel încât structura de arbore să devină independentă de tipul de date pe care va trebui să le înmagazineze.

Astfel, tipul datelor `T` care vor fi inserate în arbore este stabilit la inițializarea arborelui prin specificarea sa între parantezele unghiulare ale specificatorului `template<>`.

Acest tip `T` al datelor este la rândul său reprezentat de o clasă prototip special proiectată în cadrul aplicației curente.

Reprezentarea clasei prototip `TreeNode` în cod *C++* cuprinde două părți. Prima este una declarativă și constituie interfața clasei care este implementată într-un fișier purtând denumirea clasei și având extensia **`.H`** sau **`.HPP`**.

```
template <class T> class TreeNode{
  protected:
    T value;
    TreeNode *_lchild;
    TreeNode *_rchild;
  public:
    TreeNode(T);
    virtual ~TreeNode(void);
};
```

Cea de-a doua parte a clasei constă în definirea propriu-zisă a metodelor de operare, această parte fiind implementată într-un fișier purtând denumirea clasei și având extensia **`.C`** sau **`.CPP`**.

Constructorul `TreeNode(T)` creează un arbore binar având un singur nod și două legături către fiii săi, fiecare fiind reprezentat de pointerul `NULL`.

```
template<class T>
TreeNode<T>::TreeNode(T value) :
    value(value),_lchild(NULL),_rchild(NULL){}
```

Destructorul clasei, `~TreeNode(void)` eliberează memoria ocupată de obiectul nod cât și pe cea a fiilor săi, dacă această memorie a fost alocată:

```
template<class T>
TreeNode<T>::~TreeNode(void){
    if (_lchild) delete _lchild;
    if (_rchild) delete _rchild;
}
```

Colecția de noduri aparținând arborelui va fi stocată într-o listă ordonată. Deoarece o altă operație foarte importantă ce trebuie implementată se referă la localizarea succesorului sau a predecesorului unui nod arbitrar din arbore, vom păstra și o listă care conține pointeri la nodurile învecinate din arbore. Astfel, vom introduce o nouă clasă prototip denumită `Node` conținând legăturile la nodurile succesor și predecesor ale nodului curent.

```
class Node{
  protected:
    Node *_next;
    Node *_prev;
  public:
    Node(void);
    virtual ~Node(void);
```

```
    inline Node *next(void) const;
    inline Node *prev(void) const;
    Node *remove(void);
    Node *insert(Node *newNode);
};
```

Constructorul `Node(void)` creează un nou tip de nod corespunzând unui element în listă:

```
Node::Node(void){
  _prev=this;
  _next=this;
}
```

Am folosit pointerul C++ **this** care indică obiectul element al listei. Acest constructor inițializează pointerii `_prev` și `_next`, reprezentând legăturile către nodurile predecesor și succesor din arbore cu pointerul **this**, urmând ca aceste legături să fie setate în mod corespunzător la timpul potrivit.

Destructorul `~Node(void)` este responsabil pentru eliberarea spațiului de memorie ocupat de obiectele tip `Node` după ce durata lor de viață se încheie. Acest destructor este declarat virtual pentru a asigura eliberarea de memorie în mod corect și pentru obiectele derivate de la această clasă prototip.

```
Node::~Node(void){}
```

Funcțiile membru `next()` și `prev()` sunt folosite pentru a obține adresele nodurilor succesor și predecesor.

Aceste două funcții sunt declarate **inline** din motive de eficiență, directiva **inline** sugerând compilatorului să introducă instrucțiunile din corpul funcției în codul sursă fără a mai utiliza mecanismul de activare a funcției, mecanism care este costisitor și nejustificabil în acest caz.

Pe de altă parte, membrii clasei `Node`, `_next` și `_prev`, nu trebuie alterați în nici un fel pe durata de viață a arborelui, motiv pentru care funcțiile `next()` și `prev()` vor fi declarate **const**.

```
inline Node *Node::next(void) const{
  return _next;
}
inline Node *Node::prev(void) const{
  return _prev;
}
```

Funcția membru `insert(Node *n)` inserează pointerul nodului `n` în listă imediat după nodul curent reprezentat de pointerul **this**, refacându-se în același timp legăturile dintre noduri:

```
Node *Node::insert(Node *node){
  Node *new_node = _next;
  node->_next = new_node;
  node->_prev = this;
  _next = node;
  new_node->_prev = node;
  return node;
}
```

Un nod din listă poate fi înlăturat prin intermediul funcției membru `remove(void)` astfel încât legăturile dintre nodurile rămase să poată fi refăcute:

```
Node *Node::remove(void){
  _prev->_next = _next;
  _next->_prev = _prev;
  _next = _prev = this;
  return this;
}
```

Pe baza celor două clase prototip `TreeNode` și `Node` definite mai înainte, este introdusă o nouă clasă cu ajutorul mecanismului de moștenire, clasă numită, în mod sugestiv, `BreidedNode`. Această nouă clasă permite accesul la nodurile arborelui pe baza relației genealogice părinte-descendenți dar și prin parcurgerea listei de noduri prin intermediul listei.

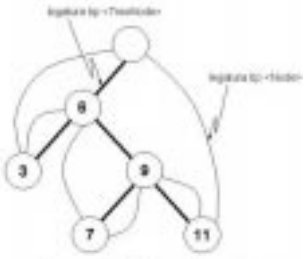

Figura 2. Arbore binar legat

```
template<class T>
class BraidedNode:public Node,
                  public TreeNode<T>{
public:
    BraidedNode( T );
    inline BraidedNode<T> *rchild(void) const;
    inline BraidedNode<T> *lchild(void) const;
    inline BraidedNode<T> *prev(void) const;
    inline BraidedNode<T> *next(void) const;
};
```

Constructorul `BreidedNode(T)` inițializează în mod explicit membrii clasei `TreeNode`:

```
template<class T>
BraidedNode<T>::BraidedNode(T value):
                        TreeNode<T>(value){}
```

Funcțiile membru `rchild()`, `lchild()`, `next()` și `prev()` returnează cele patru legături ale nodului, primele două aparținând structurii de arbore, iar ultimele două structurii de listă.

Aceste funcții sunt declarate **inline** și **const** din aceleași motive prezentate anterior.

```
template<class T>
inline BraidedNode<T> *BraidedNode<T>::
                          rchild(void) const{
  return (BraidedNode<T> *)_rchild;
}

template<class T>
inline BraidedNode<T> *BraidedNode<T>::
                          lchild(void) const{
  return (BraidedNode<T> *)_lchild;
}

template<class T>
inline BraidedNode<T> *BraidedNode<T>::
                            prev(void) const{
  return (BraidedNode<T> *)_prev;
}

template<class T>
inline BraidedNode<T> *BraidedNode<T>::
                            next(void) const{
  return (BraidedNode<T> *)_next;
}
```

Având structura de arbore deja conturată prin intermediul claselor prototip `TreeNode`, `Node` și `BraidedNode`, nu mai rămâne decât fixarea structurii sale funcționale reprezentată în principal de operațiile de inserare, căutare și localizare a datelor.

Procesul de inserare a datelor în arbore are loc în două faze distincte, mai întâi nodul este introdus în arbore prin metoda clasică (înlocuirea unui nod extern), după care se aplică o reordonare a nodurilor în arbore în funcție de numărul de prioritate al fiecăruia.

Trebuie menționat faptul că operația de reordonare necesită introducerea a două noi funcții `rotateRight()` și `rotateLeft()` în vederea poziționării nodurilor în arbore astfel încât nodul părinte să aibă numărul de prioritate mai mic decât al tuturor descendenților săi.

Astfel, dacă `x` este fiul stâng al nodului părinte `y` și are numărul de prioritate mai mic decât numărul de prioritate al nodului părinte `y`, atunci funcția `rotateRight()` va muta nodul `x` pe un nivel ierarhic superior nodului `y` în arbore, conform principiului de bază al arborelui binar de formă arbitrară.

De asemenea, este posibil ca acest principiu să nu fie respectat în cazul noului părinte al nodului `x` în noua sa configurație, caz în care va fi efectuată o nouă rotație a nodului părinte, o rotație de dreapta, dacă `x` este fiu stâng sau o rotație de stânga, dacă `x` este fiu drept.

Acest proces de rotație a nodurilor este continuat până când numărul de prioritate al nodului `x` va fi mai mic decât al tuturor nodurilor sale ascendente.

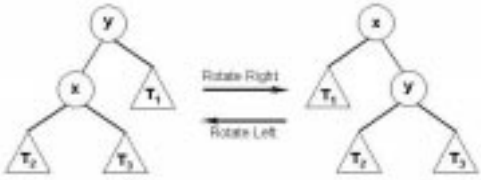

Figura 3. Rotații de noduri

Pentru a implementa funcțiile de rotație, este introdusă o nouă clasă prototip numită `RandomizedNode` al cărei acces la datele clasei `BreidedNode` este permis prin mecanismul de moștenire. Această nouă clasă conține pointerul care indică nodul părinte al nodului curent (`_parent`) cât și numărul său de prioritate (`_priority`).

```
template<class T>
class RandomizedNode: public BraidedNode<T>{
  protected:
    RandomizedNode *_parent;
    double _priority;
    void rotateRight(void);
    void rotateLeft(void);
    void bubbleUp(void);
    void bubbleDown(void);
  public:
    RandomizedNode(T value, int seed = -1 );
    inline RandomizedNode *lchild(void) const;
    inline RandomizedNode *rchild(void) const;
    inline RandomizedNode *prev(void) const;
    inline RandomizedNode *next(void) const;
    inline RandomizedNode *parent(void) const;
    inline double priority(void) const;
    friend class RandomizedSearchTree<T>;
};
```

Constructorul `RandomizedNode(T value, int seed = -1)` atribuie unui nou nod creat un număr de prioritate arbitrar, generat de funcția standard `rand()` a compilatorului C++.

```
template<class T>
RandomizedNode<T>::RandomizedNode(T value,
            int seed):BraidedNode<T>(value){
  if (seed != -1) srand(seed);
  _priority = (rand()) / (32767.0 + 1.0);
  _parent = NULL;
}
```

Funcțiile membru `rotateRight()` și `rotateLeft()` vor schimba forma arborelui prin actualizarea legăturilor nodurilor participante conform principiului reprezentat în figura 3). Astfel, prin rotația de dreapta, legătura nodului `y` la fiul său stâng este conectată la subarborele `T2` iar legătura nodului `x` la fiul său drept este conectată la nodul `y`. De asemenea, trebuie să verificăm dacă părintele nodului `x`

este definit pentru a nu permite nici unui nod să ajungă la nivelul nodului rădăcină, care, prin definiție, este unic și trebuie să rămână unic.

```
template<class T>
void RandomizedNode<T>::rotateRight(void){
  RandomizedNode<T> *y = this;
  RandomizedNode<T> *x = y->lchild();
  RandomizedNode<T> *p = y->parent();
  y->_lchild = x->rchild();
  if (y->lchild() != NULL)
    y->lchild()->_parent = y;
  if (p->rchild() == y) p->_rchild = x;
  else p->_lchild = x;
  x->_parent = p;
  x->_rchild = y;
  y->_parent = x;
  return;
}
```

Funcția `rotateLeft()` efectuează operația de rotație inversă, aceasta fiind exprimată în mod simetric funcției `rotateRight()` după cum urmează:

```
template<class T>
void RandomizedNode<T>::rotateLeft(void){
  RandomizedNode<T> *x = this;
  RandomizedNode<T> *y = x->rchild();
  RandomizedNode<T> *p = x->parent();
  x->_rchild = y->lchild();
  if ( x->rchild() != NULL )
    x->rchild()->_parent = x;
  if ( p->lchild() == x ) p->_lchild = y;
  else p->_rchild = y;
  y->_parent = p;
  y->_lchild = x;
  x->_parent = y;
  return;
}
```

Funcțiile `bubbleUp()` și `bubbleDown()` au scopul de a efectua rotațiile respective pâna când numărul de prioritate al nodului curent satisface principiul de construcție a arborelui binar de formă arbitrară. Operațiile de rotație sunt aplicate nodului părinte.

```
template<class T>
void RandomizedNode<T>::bubbleUp(void){
  RandomizedNode<T> *p = parent();
  if (priority() < p->priority()){
    if ( p->lchild() == this)
      p->rotateRight();
    else p->rotateLeft();
    bubbleUp();
  }
}
```

```
template<class T>
void RandomizedNode<T>::bubbleDown(void){
  double lcPriority = lchild()?
                    lchild()->priority():2.0;
  double rcPriority = rchild()?
                    rchild()->priority():2.0;
  double minPriority = (lcPriority <
           rcPriority)?lcPriority:rcPriority;
  if (priority() <= minPriority)
    return;
  if (lcPriority < rcPriority)
    rotateRight();
  else rotateLeft();
  bubbleDown();
  return;
}
```

Sensul operației de rotație este determinat de relația dintre nodurile fii, dacă fiul stâng are numărul de prioritate mai mic, o rotație de dreapta `rotateRight()` va muta fiul stâng pe un nivel superior în arbore, în timp ce nodul curent va fi mutat pe un nivel inferior, după cum se poate observa în figura 3).

În caz contrar, va fi aplicată o rotație de stânga `rotateLeft()`. Pentru a preveni orice interferență a nodurilor exterioare cu nodurile arborelui, nodurilor exterioare li se atribuie un număr de prioritate `2.0` sau cel puțin un număr din afara intervalului (`0.0`, `1.0`).

Funcțiile `rchild()`, `lchild()`, `next()`, `prev()` și `parent()` au fost implementate în scopul obținerii accesului la nodurile respective, acces foarte util în operațiile cu arbori.

```
template<class T>
inline RandomizedNode<T>
      *RandomizedNode<T>::lchild(void) const{
  return (RandomizedNode<T> *) _lchild;
}
```

```
template<class T>
inline RandomizedNode<T>
      *RandomizedNode<T>::rchild(void) const{
  return (RandomizedNode<T> *) _rchild;
}
```

```
template<class T>
inline RandomizedNode<T>
       *RandomizedNode<T>::prev(void) const{
  return (RandomizedNode<T> *) _prev;
}
```

```
template<class T>
inline RandomizedNode<T>
       *RandomizedNode<T>::next(void) const{
  return (RandomizedNode<T> *) _next;
}
```

```
template<class T>
inline RandomizedNode<T>
      *RandomizedNode<T>::parent(void) const{
  return (RandomizedNode<T> *) _parent;
}

template<class T>
inline double RandomizedNode<T>::
                        priority(void) const{
  return _priority;
}
```

Arborele binar propriu-zis este reprezentat de o nouă clasă prototip denumită `RandomizedSearchTree`. Membrii clasei sunt rădăcina arborelui (`root`) și un cursor (`cursor`).

O metodă foarte importantă a acestei clase este funcția de comparare `comparison(T,T)` pe baza căreia se face introducerea datelor în arbore.

Alte metode ale clasei sunt implementate pentru a facilita folosirea structurii de arbore în mod eficient.

```
template<class T> class RandomizedSearchTree{
  private:
    RandomizedNode<T> *root;
    RandomizedNode<T> *cursor;
    int (*comparison)(T,T);
    void _remove(RandomizedNode<T> *);
  public:
    RandomizedSearchTree(int (*)(T,T),
                         int seed = -1);
    ~RandomizedSearchTree(void);
    inline T next(void);
    inline T prev(void);
    inline T value(void);
    inline bool isFirst(void) const;
    inline bool isLast(void) const;
    inline bool isHead(void) const;
    inline bool isEmpty(void) const;
    inline T findMin(void);
    inline T findMax(void);
    T find(T);
    T locate(T);
    T insert(T);
    T remove(T);
    T removeMin(void);
    T removeMax(void);
    void remove(void);
    void inorder(void (*visit)(T));
};
```

Constructorul clasei, `RandomizedSearchTree(int (*)(T,T), int seed = -1)` inițializează un arbore binar de formă arbitrară care conține un singur nod izolat (`root`) care reprezintă radacina arborelui și al cărui număr de prioritate este `-1.0`.

```
template<class T>
RandomizedSearchTree<T>::RandomizedSearchTree
  (int (*cmp)(T,T),int seed): comparison(cmp){
   cursor = root = new RandomizedNode<T>
                                   (NULL,seed);
   root->_priority = -1.0;
}
```

Destructorul `~RandomizedSearchTree()` este responsabil pentru eliberarea spațiului de memorie a arborelui când durata sa de viață ia sfârșit.

```
template<class T>
RandomizedSearchTree<T>::
                  ~RandomizedSearchTree(void){
  delete root;
}
```

Funcțiile `next()`, `prev()`, `value()`, `isFirst()`, `isLast()`, `isHead()` și `isEmpty()` sunt definite după cum urmează:

```
template<class T>
inline T RandomizedSearchTree<T>::next(void){
  cursor = cursor->next();
  return cursor->value;
}

template<class T>
inline T RandomizedSearchTree<T>::prev(void){
  cursor = cursor->prev();
  return cursor->value;
}

template<class T>
inline T RandomizedSearchTree<T>::value(void){
  return cursor->value;
}

template<class T>
inline bool RandomizedSearchTree<T>::
                          isFirst(void) const{
  return (cursor == root->next()) &&
         (root != root->next());
}

template<class T>
inline bool RandomizedSearchTree<T>::
                           isLast(void) const{
  return (cursor == root->prev()) &&
         (root != root->next());
}

template<class T>
inline bool RandomizedSearchTree<T>::
                           isHead(void) const{
  return cursor == root;
}
```

```
template<class T>
inline bool RandomizedSearchTree<T>::
                          isEmpty(void) const{
  return root == root->next();
}
```

Funcția membru `insert(T avalue)` este responsabilă pentru introducerea datelor în arbore:

```
template<class T>
T RandomizedSearchTree<T>::insert(T avalue){
                        // find external node
  int result = 1;
  RandomizedNode<T> *p = root;
  RandomizedNode<T> *node = root->rchild();
  while (node){
    p = node;
    result = (*comparison)(avalue,p->value);
    if (result < 0)
      node = p->lchild();
    else
      if (result > 0)
        node = p->rchild();
      else
        return NULL;
  }
  cursor = new RandomizedNode<T>(avalue);
  cursor->_parent = p;
  if (result < 0){
     p->_lchild = cursor;
     p->prev()->Node::insert(cursor);
  }
  else{
    p->_rchild = cursor;
    p->Node::insert(cursor);
  }
     // bubble item according to its priority
  cursor->bubbleUp();
  return avalue;
}
```

Mecanismul de inserare a datelor este compus din două faze distincte. Într-o primă etapă, inserarea unei date x_i se efectuează prin plasarea sa în nodul extern corespunzător, făcând abstracție de numărul de prioritate atribuit. În etapa a doua, forma arborelui este ajustată corespunzător regulii priorităților în arbore, astfel încât prioritatea oricărui nod x_i din arbore să fie mai mare sau egală cu prioritatea părintelui său.

Funcția membru `find(T avalue)` este folosită în operația de căutare a datei `avalue` al cărei tip este indicat de variabila `T` care este identică cu cea a datelor stocate în arbore.

```
template<class T>
T RandomizedSearchTree<T>::find(T avalue){
  RandomizedNode<T> *node = root->rchild();
  while (node){
    int result = (*comparison)(avalue,
                                node->value);
    if (result < 0)
      node = node->lchild();
    else if (result > 0)
           node = node->lchild();
         else{
           cursor = node;
           return node->value;
         }
  }
  return NULL;
}
```

Operația de căutare a datei `avalue` se desfășoară de-a lungul unui traseu de căutare unic determinat începând de la rădăcină și coborând în arbore.

Pentru fiecare nod de pe traseu, este folosită funcția de comparație a arborelui pentru compararea datei căutate `avalue` cu cea stocată în nodul curent (`node->value`). Dacă `avalue < node->value`, procesul de căutare este continuat în subarborele stâng prin `node->_lchild`, în caz contrar procesul de căutare fiind continuat în subarborele drept. În cele din urmă, cursorul arborelui este fixat la nodul datei găsite și funcția de căutare se termină prin returnarea datei găsite. Dacă data căutată nu se află în arbore, se returnează pointerul `NULL`.

O altă operație de căutare este reprezentată de funcția membru `locate(T avalue)` care, aplicată argumentului său, returnează argumentul însuși dacă această dată este conținută în arbore. În cazul în care data nu este conținută în arbore, se retunează data cu valoarea imediat următoare în ordine descrescătoare.

```
template<class T>
T RandomizedSearchTree<T>::locate(T avalue){
  RandomizedNode<T> *locate = root;
  RandomizedNode<T> *node = root->rchild();
  while (node){
    int result = (*comparison)(avalue,
                                node->value);
    if (result < 0) node = node->lchild();
    else
      if (result > 0){
        locate = node;
        node = node->rchild();
      }
      else{
        cursor = node;
        return cursor->value;
      }
  }
  cursor = locate;
  return cursor->value;
}
```

Funcțiile membru `findMin()` și `findMax()` sunt responsabile pentru returnarea datelor considerate minime, respectiv maxime potrivit criteriului de comparație impus. Aceste funcții sunt extrem de eficiente având în vedere structura ordonată a arborelui, nici o operație de căutare nefiind necesară; aceste funcții sunt declarate **inline**.

```
template<class T>
inline T RandomizedSearchTree<T>::
                                findMin(void){
  cursor = root->next();
  return cursor->value;
}

template<class T>
inline T RandomizedSearchTree<T>::
                                findMax(void){
  cursor = root->prev();
  return cursor->value;
}
```

Funcția `remove(T avalue)` șterge data `avalue` din arbore dacă aceasta există, cursorul arborelui fiind mutat la locația predecesorului datei `avalue`. Implementarea acestei funcții se bazează pe funcția membru `_return(RandomizedNode<T> *node)` declarată **private** în clasa `RandomizedSearchTree`.

```
template<class T>
T RandomizedSearchTree<T>::remove(T avalue){
  T value = find(avalue);
  if (value){
    remove();
    return value;
  }
  return NULL;
}

template<class T>
void RandomizedSearchTree<T>::
            _remove(RandomizedNode<T> *node){
  node->_priority = 1.5;
  node->bubbleDown();
  RandomizedNode<T> *p = node->parent();
  if (p->lchild() == node) p->_lchild = NULL;
  else p->_rchild = NULL;
  if (cursor == node) cursor = node->prev();
  node->Node::remove();
  delete node;
  return;
}
template<class T>
void RandomizedSearchTree<T>::remove(void){
  if (cursor != root) _remove( cursor );
  return;
}
```

Analiza performanței

În cazul arborelui binar de formă arbitrară, o analiză a performanței se poate efectua numai statistic, dat fiind faptul că forma arborelui este controlată de numerele de prioritate arbitrare asociate seturilor de date. Principalul obiectiv al analizei de față îl constituie determinarea numărului de operații computaționale necesare în procesul de inserare, căutare și localizare de date reprezentate de funcțiile arborelui `insert()`, `find()`, `locate()`, `remove()`.

Pentru aceasta se consideră un arbore de formă arbitrară ale cărui componente sunt $x_1, x_2, ..., x_n$ luate în ordinea lor crescătoare, reprezentând cazul cel mai defavorabil de introducere a datelor în arborele binar clasic. Datorită ordinii de inserare, un nod x_i va ocupa în prima fază un nod extern unic determinat, în cele din urmă ocupând o poziție pe traseul de căutare al nodului imaginar x_k.

Introducerea datelor în arborele inițial vid se face prin ocuparea unuia dintre nodurile sale externe într-o primă fază. Astfel, x_1 va ocupa singurul nod exterior al arborelui, devenind rădăcina acestuia. Prin urmare, probabilitatea ca nodul x_1 să se situeze pe traseul de căutare al nodului imaginar x_k este 1.0, x_1 fiind părinte al acestuia.

Următoarea componentă introdusă în arbore este x_2 care va ocupa unul dintre cele două noduri externe ale arborelui în configurația sa curentă, probabilitatea ca x_2 să se situeze pe traseul de căutare al lui x_k fiind 0.5.

În general, prin introducerea unei date x_i, $i = 1, ..., k - 1$, probabilitatea ca x_i să ocupe un anumit nod extern este aceeași pentru toate nodurile externe. Deoarece numai într-o singură poziție în arbore, x_i se va situa pe traseul de căutare al lui x_k, probabilitatea ca x_i să se situeze pe traseul de căutare al lui x_k este $1/i$.

Prin urmare, adâncimea probabilă a datei x_k poate fi determinată cu ajutorul expresiei:

$$d_{x_k}^{\max} = 1 + \sum_{i=1}^{k-1} \left(\frac{1}{i}\right) \cong O(\log n), \quad k \le n.$$

Deoarece numărul de operații necesare pentru inserare, căutare și localizare este proporțional cu adâncimea în arbore a lui x_k rezultă că aceste procese au loc într-un timp de ordinul $O(\log n)$ pentru acest tip de arbore binar.

Bibliografie

1. **S.B. Lipmann, J. Lajoie**, *C++ Primer*, Addison-Wesley, Third Ed., 1999.
2. **B. Stroustrup**, *The C++ Programming Language*, Addison-Wesley, 1991.
3. **K. Mulmuley**, *Computational Geometry: An introduction through randomized algorithms*, Prentice Hall, Englewood Cliffs, NJ, 1994.
4. **K.E. Gorlen, S.M. Orlow, P.S. Plexico**, *Data Abstraction and Object Oriented Programming in C++*, Willey, Chichester, England, 1990.

Domul Adrian Vasilescu este asistent de programare la Universitatea Carleton din Ottawa, Canada. Poate fi contactat prin e-mail la adresa **avasiles@mrco2.carleton.ca**.