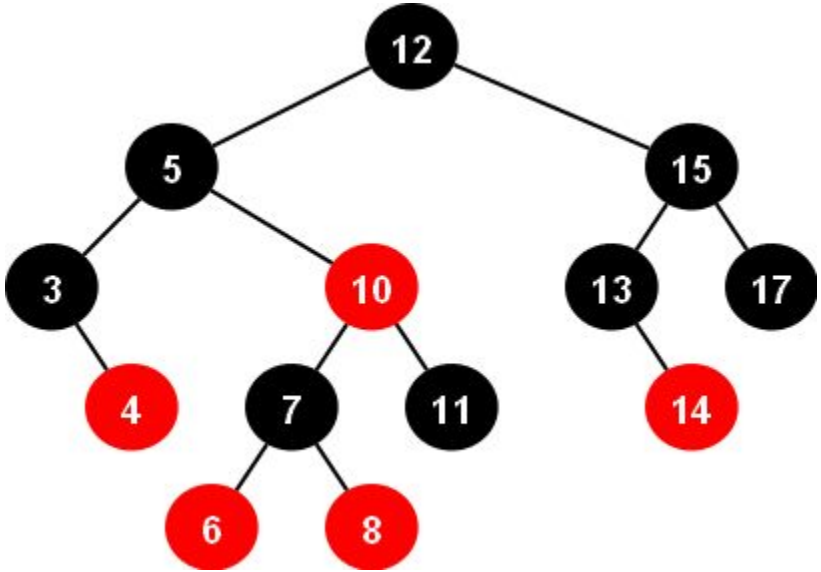
Red-black trees

de Biciușcă Matei-Alexandru



Scurtă introducere Red-black trees (arbori roșii-negri

În informatică, un arbore roșu-negru este un fel de arbore de căutare binar auto-echilibrat. Fiecare nod stochează un spațiu (bool) suplimentar reprezentând „culoarea” („roșu” sau „negru”), folosit pentru a se asigura că arborele rămâne echilibrat în timpul inserărilor și ștergerilor.

Proprietăți:

- Fiecare nod e fie roșu, fie negru
- Rădăcina e mereu neagră
- Nu putem avea două noduri adiacente roșii
- Orice drum de la un nod la un descendent NULL are același număr de noduri negre

Operații/funcții necesare

Ca orice arbore binar de căutare echilibrat, acesta implementează funcțiile clasice, de inserare, ștergere și căutare. În plus, pe arborele am implementat funcții de succesor/predecesor și parcurgere inorder, pentru rezolvarea problemei abce de pe infoarena.

Dat fiind faptul că lucrăm pe un arbore binar de căutare echilibrat, am implementat funcții de left/right rotation, de swap, și de reparare al arborelui, în urma ștergerii/inserării.

Complexitățile în notația big O

Spațiu: O(n)

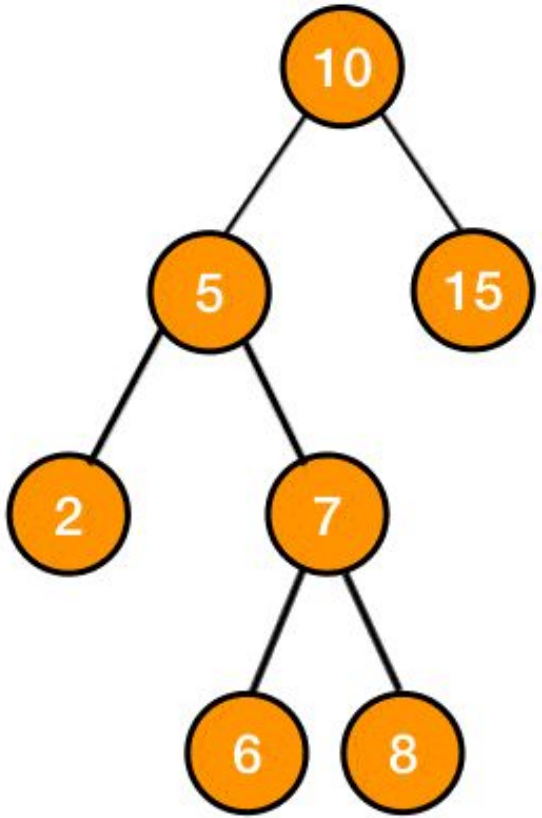
Timp:

Funcție	Amortizat	Worst case
Inserare	O(1)	O(log n)
Căutare	O(log n)	O(log n)
Ștergere	O(1)	O(log n)
Par. Inorder	O(n)	O(n)
Left Rotation	O(1)	O(1)
Right Rotation	O(1)	O(1)
Fix Insert	O(1)	O(log n)
Fix Delete	O(1)	O(log n)

Left rotation

În rotația stângă, presupunem că copilul drept nu este nul.

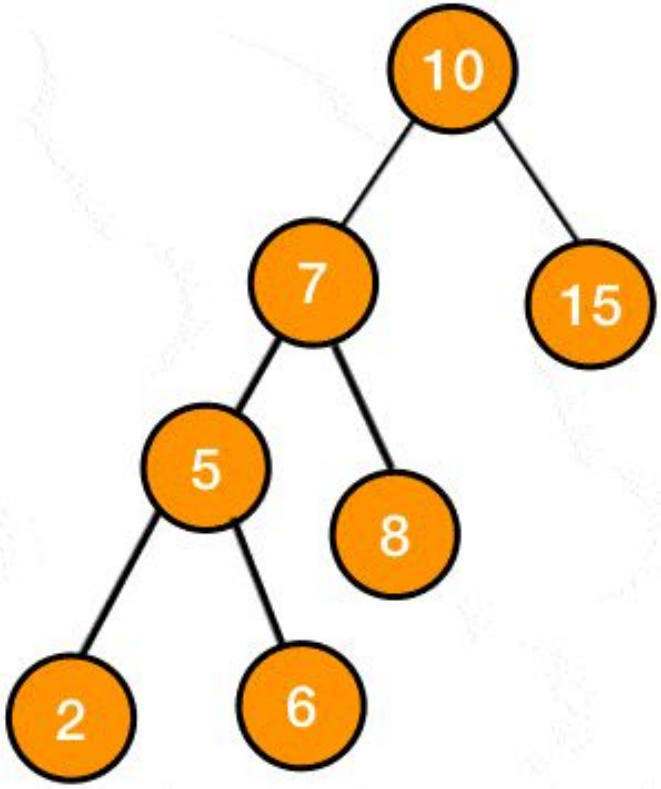
După aplicarea rotației la stânga pe nodul x, nodul y va deveni noua rădăcină a subarborelui și copilul său din stânga va fi x. Și copilul din stânga anterior al lui y va deveni acum copilul din dreapta al lui x.



Right Rotation

În mod similar, în rotația din dreapta, presupunem că copilul stâng nu este nul.

Așadar rotația la dreapta pe nodul y va face din x rădăcina arborelui, y va deveni copilul drept al lui x. Și copilul din dreapta anterior al lui x va deveni acum copilul din stânga al lui y.

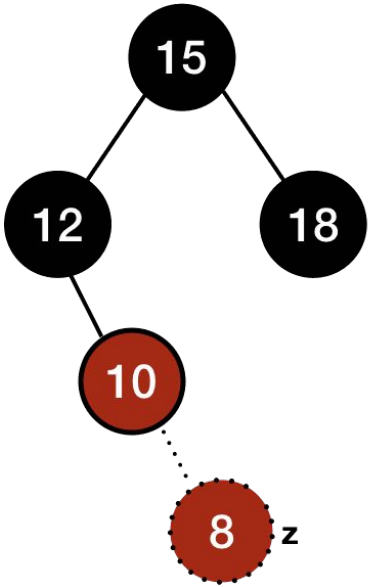


Inserare și fix insert în RB Tree

Există trei faze pentru inserarea unei chei într-un arbore care nu este gol. Operația de inserare a arborelui de căutare binar este efectuată în prima fază. Deoarece un arbore roșu-negru este echilibrat, operația de inserare BST este O (înălțimea arborelui), care este O (log n).

Noul nod este apoi colorat în roșu în a doua etapă. Acest pas este O(1) deoarece implică doar modificarea valorii câmpului de culoare al unui nod. În a treia etapă, restaurăm toate caracteristicile roșu-negru care au fost încălcate.

Schimbarea culorilor nodurilor durează O(1) timp. Cu toate acestea, ar putea fi nevoie să ne ocupăm de o problemă dublu-roșu mai departe de-a lungul traseului de la nodul inserat la rădăcină.



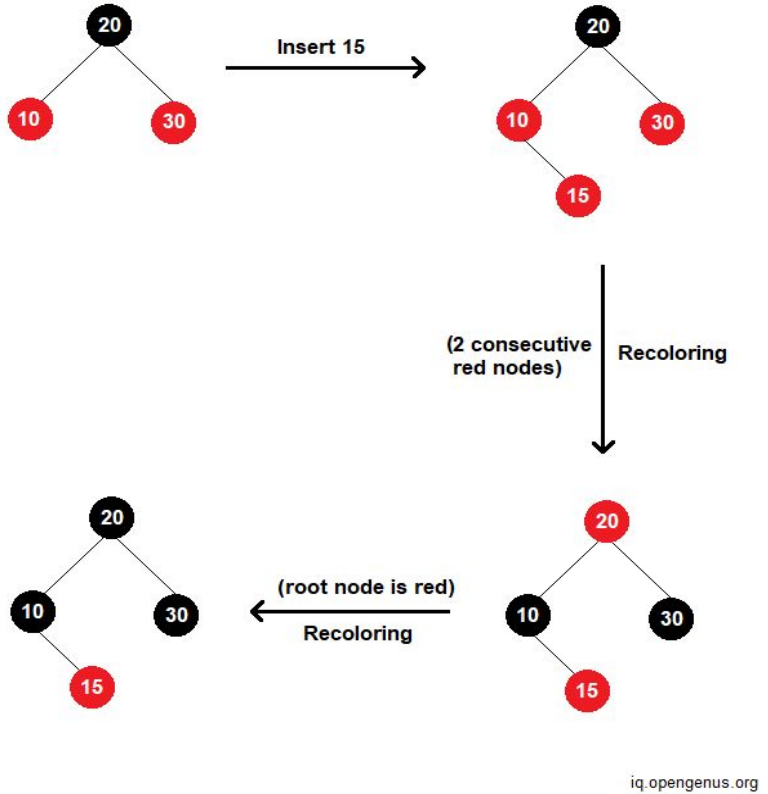
Inserare de valoare, cu apariție de dublu-roșu

Inserare și fix insert în RB Tree

În cel mai rău caz, vom remedia o condiție de dublu roșu pe tot drumul de la nodul inserat la rădăcină. În cel mai rău caz, recolorarea efectuată în timpul inserării este O(log n), adică timpul pentru o recolorare x numărul maxim de recolorări efectuate.

Ca rezultat, restabilirea caracteristicilor roșu-negru necesită O(log n), iar timpul total pentru inserare este O (log n).

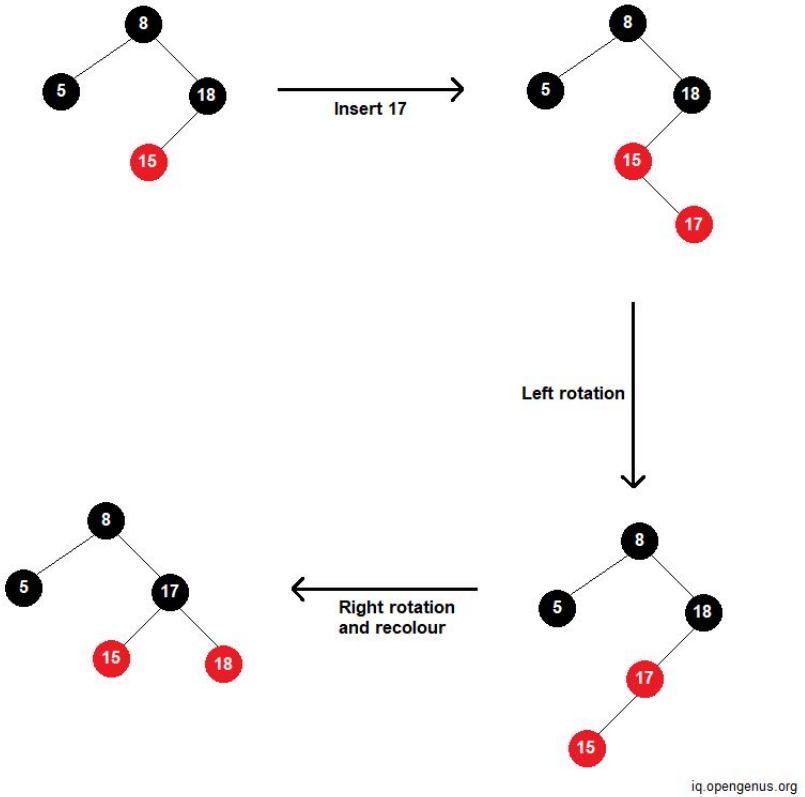
Cel mai bun caz: În cel mai bun caz, nu există rotație. Are loc doar recolorarea. Complexitatea timpului este O(log n).



Inserare și fix insert în RB Tree

Cel mai rău caz: arborii RB necesită un număr constant (cel mult 2 pentru inserare) de rotații. Deci, în cel mai rău caz, vor exista 2 rotații în timpul inserției. Complexitatea timpului este O(log n).

Ca rezultat, restabilirea regulilor arborelui roșu-negru necesită O(log n), iar timpul total pentru inserare este O (log n).

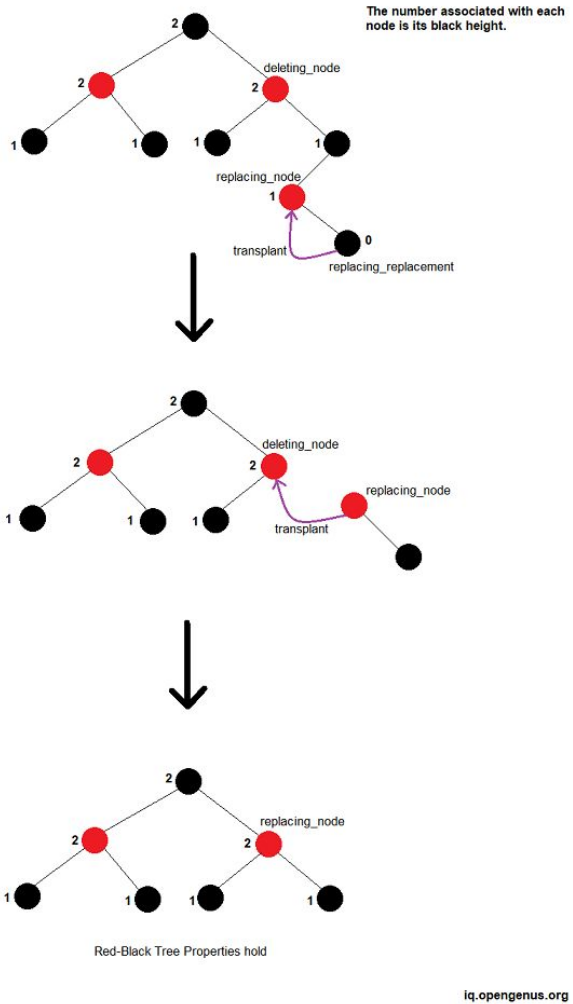


Ștergere și fix delete în RB Tree

Operația de ștergere în arborele roșu-negru este puțin mai complicată decât alți arbori binari. Un lucru de reținut este că un arbore roșu-negru ar trebui să fie în continuare arbore roșu-negru dacă un element este îndepărtat.

Găsirea nodului de ștergere plus succesorul din stânga din dreapta este proporțională cu înălțimea arborelui, deci este O(log n). Schimbarea și ștergerea sunt ambele O(1). Fiecare fix particular (rotație, de exemplu) este O(1).

În cel mai rău caz, un dublu-negru ar putea fi transmis până la rădăcină. Deoarece fiecare rotație durează aceeași perioadă de timp, aceasta este proporțională cu înălțimea arborelui și deci O(log n). Ca rezultat, cel mai rău caz de complexitate a ștergerii este O(log n).

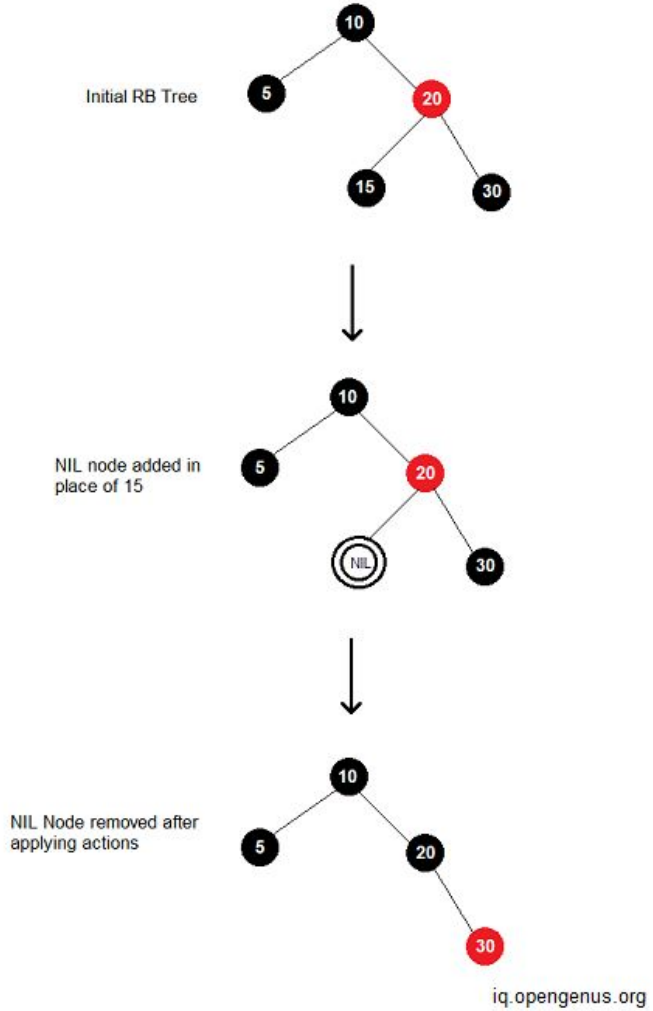


Ștergere și fix delete în RB Tree

Găsirea nodului de ștergere plus succesorul din stânga din dreapta este proporțională cu înălțimea arborelui, deci este O(log n). Schimbarea și ștergerea sunt ambele O(1). Fiecare fix particular (rotație, de exemplu) este O(1). În cel mai rău caz, un dublu-negru ar putea fi transmis până la rădăcină.

Deoarece fiecare rotație durează aceeași perioadă de timp, aceasta este proporțională cu înălțimea arborelui și deci O(log n). Ca rezultat, cel mai rău caz de complexitate a ștergerii este O(log n).

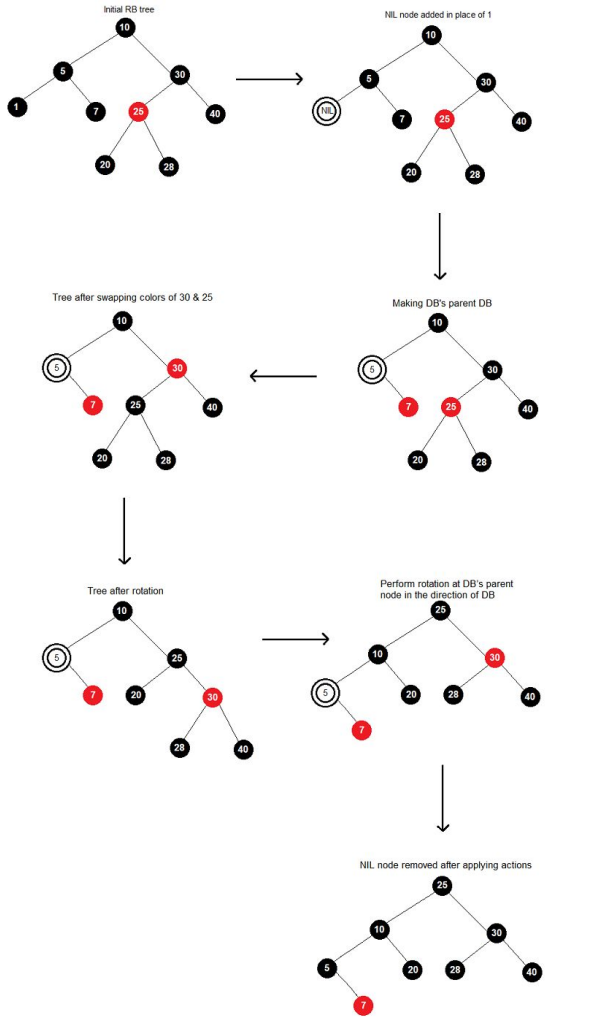
Cel mai bun caz: În cel mai bun caz, nu există rotație. Are loc doar recolorarea. Complexitatea timpului este O(log n).



Ștergere și fix delete în RB Tree

Cel mai rău caz: arborii RB necesită un număr constant (cel mult 3 pentru ștergere) de rotații. Deci, în cel mai rău caz, vor exista 3 rotații în timpul ștergerii. Complexitatea timpului este O(log n).

Cazul mediu: Deoarece cazul mediu este media tuturor cazurilor posibile, complexitatea de timp a ștergerii și în acest caz este O(log n).



Căutarea în RB tree

Operația de căutare în arborele roșu-negru are loc exact ca în oricare BST, fără a folosi vreo funcție specifică RB tree. căutarea unui nod în Red Black Tree

Efectuați o căutare binară asupra înregistrărilor din nodul curent.

Dacă este găsită o înregistrare cu cheia de căutare, atunci returnați acea înregistrare. Dacă nodul curent este un nod frunză și cheia nu este găsită, atunci raportați o căutare nereușită. În caz contrar, urmați ramura potrivită și repetați procesul.

```
void tree_search(int &val) {
      Node* x = root;
      while(x != stop && x -> val != val){
            if(val < x -> val)
                  x = x -> left;
            else
                  x = x -> right;
      }
      if(x != stop)
            fout << "1\n";
      else
            fout << "0\n";
}
///func□ia de căutare în RB tree în C++
```

Predecesor, succesor și afișare sortată

Pe infoarena la problema abce, cerințele 4, 5 și 6 erau mai generale, unde codul pentru găsirea predecesorului, succesorului și afișării inorder pentru valori între x și y era asemănător cu aproape oricare BST, deoarece era nevoie, ca nivel de informație, doar de rădăcină, fii și valoarea din nod.

Predecesor, succesor: O(log n).

Parcurgere inorder: O(n).

Aplicații și concluzii RB tree

Aplicații:

- Scheduler complet corect în kernelul Linux

- Geometrie computațională Structuri de date

- În diferite implementări ale structurilor de date asociative (de exemplu, C++ STL folosește arborele RB intern pentru a implementa Set și Map)

- Începând cu Java 8, HashMap a fost modificat astfel încât, în loc să utilizeze o LinkedList pentru a stoca diferite elemente cu coduri hash care se ciocnesc, este folosit un arbore roșu-negru. Acest lucru are ca rezultat îmbunătățirea complexității în timp a căutării unui astfel de element de la O(m) la O(log m) unde m este numărul de elemente cu coduri hash care se ciocnesc.

Aplicații și concluzii RB tree

Concluzii:

Un anume manush pe nume Robert Sedgewick a prezentat la un curs de-al său cum RBT apare și serialul Missing:

Jess: It was the red door again.

Pollock: I thought the red door was the storage container.

Jess: But it wasn't red anymore, it was black.

Antonio: So red turning to black means what?

Pollock: Budget deficits, red ink, black ink.

Antonio: It could be from a binary search tree. The red–black tree tracks every simple path from a node to a descendant leaf that has the same number of black nodes.

Jess: Does that help you with the ladies?

Sfârșit!

Dacă ți-a plăcut prezentarea, dă un like/follow pe Github.

Dacă nu (most likely), check out this stuff:

https://www.cs.usfca.edu/~galles/visualization/RedBlack.html

Mulțumesc!