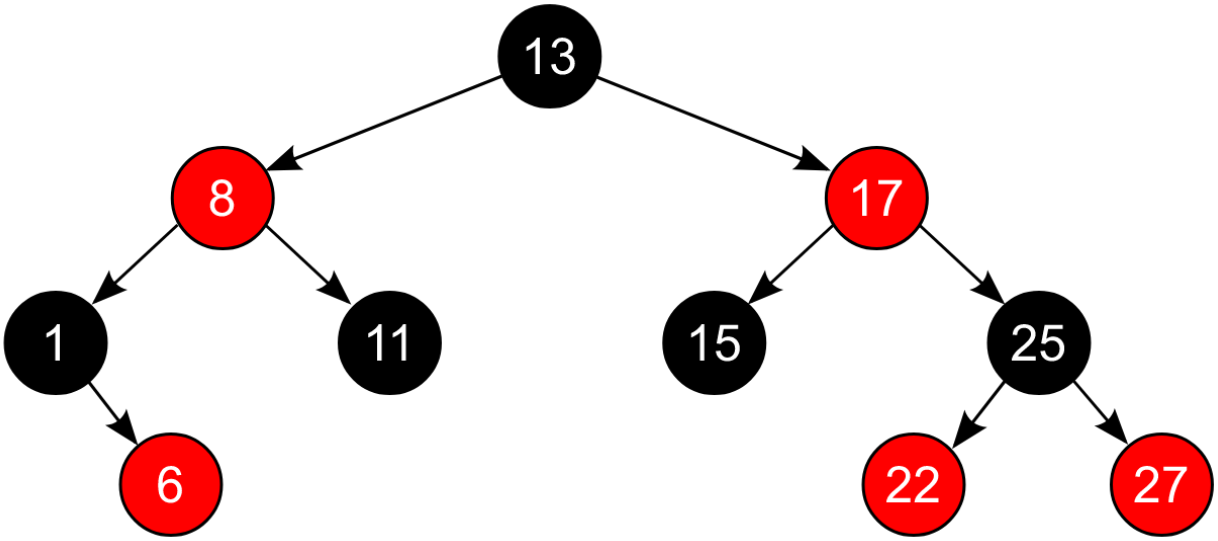
Червено-черно дърво



Червено-черното дърво е подтип на BST. То запазва същите свойства на BST, но освен това е и самобалансиращо се. Червено-черното дърво заема повече памет, защото всеки елемент има един параметър повече – color.

То е измислено в края на миналия век, когато двама студенти решават да създадат по-добър начин за стабилизиране на бинарно дърво. Използвайки червена и черна химикалка за да изобразят идеята си, те измислят червено-черното дърво.

Червено-черното дърво трябва да бъде балансирано спрямо цветовете. То трябва да изпълнява няколко правила:

- Връх в едно червено-черно дърво задължително е или черен или червен.
- Коренът и листата са черни
- Децата на всеки червен връх са задължително черни
- Пътят от върха до което и да е листо от дървото преминава през еднакъв брой черни върхове.

Червено-черното дърво запазва времевата сложност на бинарното дърво:

search, insert, delete -> O (logN);

Всяко червено-черно дърво е BST, но не всяко BST е червено-черно дърво.

Функции:

- Search/Find - използва същия алгоритъм като BST.
- Insert – Винаги добавяме нов елемент с червен цвят. Единственият случай, в който можем да добавим черен връх е ако той е корен. Ако родителят ни е черен добавяме елемента. Ако обаче е червен разглеждаме чичото. Ако чичото на върха, в който се намираме също е червен, оцветяваме чичото и родителя ни в черно, а прародителя ни става червен. Ако чичото не е червен, разглеждаме четири случая:
  1. Елементът, който добавяме е най-лявото листо на лявото поддърво на корена (от корена сме се движили само наляво):
     а) Правим ротация наясно т.е. всички върхове се изместват с една позиция надясно и съответно променят цветовете си.
  2. Ако се намираме отдясно на левия син на прародителя правим лява ротация
  3. Ако се намираме отляво на десния син на прародителя правим дясна ротация.
  4. Ако се намираме отдясно на десния син на прародителя:
     Аналогично на случай 1

Времева сложност: O (logN)

Пространствена сложност: O(N)

- Delete – Разделя се на три функции, всяка от които има подслучаи:
  1. transplant- премества върхове, така че да ни бъдат удобни за изтриване съгласно правилата.
  a. Ако искаме да изтрием корена запомняме неговите деца и правим по-голямото дете новият корен.
  b. Ако искаме да изтрием връх в лявото поддърво на корена, лявото дете на родителя му става дясно дете на върха, който искаме да изтрием

c. Ако искаме да изтрием връх в дясното поддърво на корена извършаме аналогични на b. стъпки

2. delete- премахва съответен връх. Функцията има четири варианта. Един от тривиалните случаи е когато изтриваме елемент без деца. В този случай прекъсваме връзката с родителя и изтриваме елемента.
   a. Елемента има само дясно дете:
      Запомняме цвета на елемента, който искаме да изтрием, извикваме гореобяснената функция transplant, като за нейни параметри подаваме елемента, който искаме да изтрием и неговото дете. Детето застава на мястото на изтрития от нас елемент. Ако цвета на изтрития връх е бил черен, извикваме delete-fixup, а ако е бил червен, това не е необходимо.
   b. Елементът има само ляво дете:

      Аналогично с a.

   c. Ако елементът има две деца:

      Запомняме цвета на елемента, който искаме да изтрием и го запазваме самият елемент. Нека например го обозначим със z. След това избираме или най-малкият елемент отдясно, или най-големият елемент отляво. Нека в случая изберем най-малкия елемент отдясно. Обозначаваме го с y. Означаваме дясното дете на y с x, независимо дали съществува. Извикваме функцията transplant с y и x и възстановяваме връзките: y става корен на дясното поддърво. Извикваме отново transplant с y и z, лявото дете на y става лявото дете на z и изтриваме z. Накрая, ако оригиналния цвят е бил черен, трябва да извикаме delete-fixup.

3. delete-fixup- грижи се за стабилизацията на дървото след изтриване

   Функцията приема един връх. Изпълнява цикъл, който се извършва докато дървото не се балансира и накрая подадения връх става черен. Нека x е подаденият елемент, и нека w е неговият брат/сестра.

   a. Ако w е nullptr, x става черен.
   b. Ако w е червен:
      - w става черен
      - родителя на x става червен

- правим лява ротация спрямо родителя на x
- w отново става брат/сестра на x

c. Ако w е черен и двете му деца са черни
   - w става червен
   - x става неговият родител

d. Ако w е черен и лявото му дете е червено, а дясното е черно
   - Лявото дете на w става черно
   - w става червено
   - правим дясна ротация спрямо w
   - w отново става брат/сестра на x

e. Ако w е черен и дясното му дете е червено, незеависимо от цвета на лявото или дали то съществува
   - w става цвета на родителя на x
   - цвета на родителя става черен
   - дясното дете на w става черно
   - правим лява ротация спрямо родителя
   - x става корена

ПРИМЕРНА ИМПЛЕМЕНТАЦИЯ НА C++ :

https://github.com/Bibeknam/algorithmtutorprograms/blob/master/data-structures/red-black-trees/RedBlackTree.cpp

ПРИМЕРНА ИМПЛЕМЕНТАЦИЯ НА PYTHON:

https://github.com/msambol/dsa/blob/master/trees/red_black_tree.py

Предимства и недостатъци на червено-черното дърво:

Предимствата са, че то със сигурност ни осигурява сложност O (logN), то е самобалансиращо се и е лесно за разбиране.

Недостатъците са, че заема повече памет и е трудно за имплементация.

Изготвено от Чочо Владовски, 4 група

ГЛЕДАЙ ГО (30 MIN MAX) :
https://www.youtube.com/watch?v=qvZGUFHWChY&list=PL9xmBV_5YoZNqDI8qfOZgzbqahCUmUEin

Допълнителна информация:

https://www.geeksforgeeks.org/introduction-to-red-black-tree/

https://www.geeksforgeeks.org/insertion-in-red-black-tree/

https://www.geeksforgeeks.org/deletion-in-red-black-tree/

https://en.wikipedia.org/wiki/Red%E2%80%93black_tree

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

https://bg.wikipedia.org/wiki/%D0%A7%D0%B5%D1%80%D0%B2%D0%B5%D0%BD%D0%BE-%D1%87%D0%B5%D1%80%D0%BD%D0%BE_%D0%B4%D1%8A%D1%80%D0%B2%D0%BE