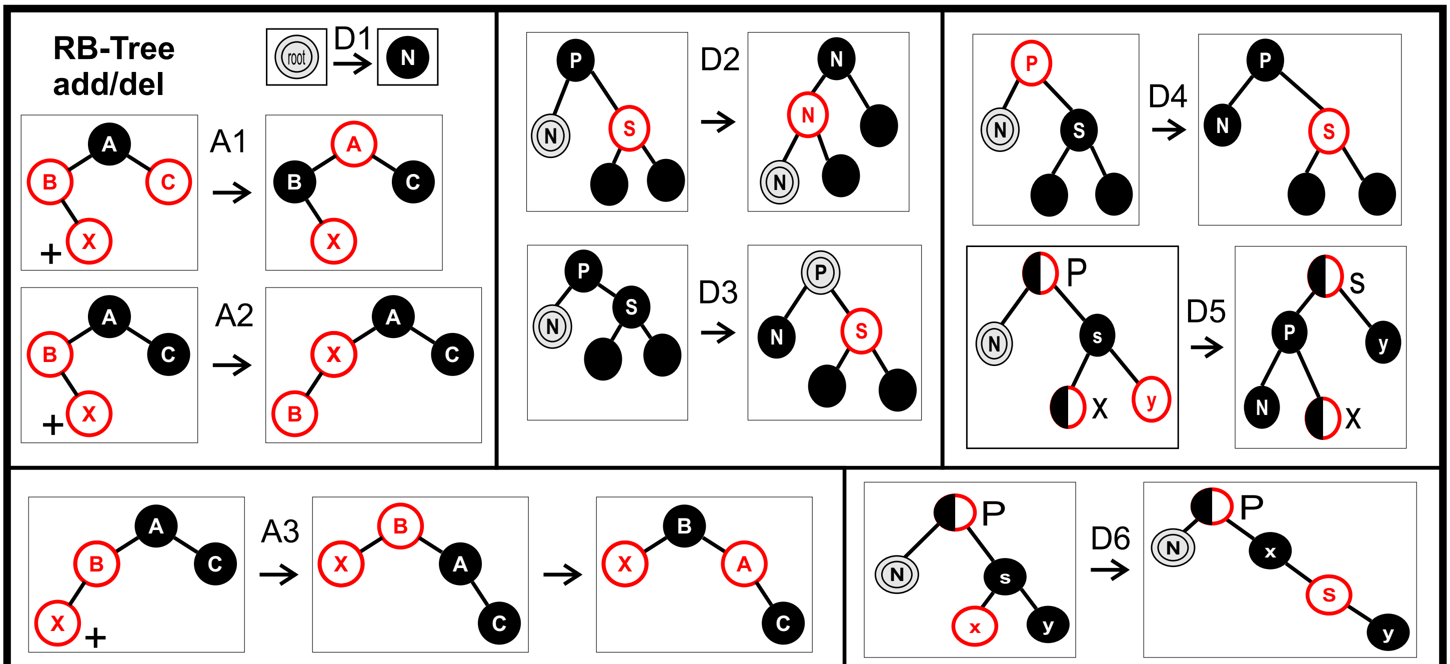
RB-Tree add/del



5 warunków RB-Tree

Każdy węzeł albo czerwony, albo czarny.
Każdy liść (węzeł pusty nil) zawsze czarny.
Korzeń drzewa zawsze czarny.
Jeśli węzeł czerwony, to obaj synowie czarni.
Z węzła do dowolnego liścia potomnego tyle samo czarnych.

Stacje benzynowe

Na początku narysuj graf i powiedz, że to DAG (skier. acykl)
L(i,a) to koszt dojazdu do i-tej stacji z a litrami paliwa w baku.
1. Sprawdź do jakich stacji możesz dojechać na pełnym baku
2. Dojedź do najtańszej stacji
3. Sprawdź do jakich stacji można dojechać na pełnym baku
4. Jeśli na wszystkich cena jest wyższa, zatankuj do pełna i pojedź do kolejnej najtańszej stacji albo zatankuj tyle żeby dojechać do końca
5. Jeśli cena jest gdzieś niższa, zatankuj tyle żeby tam dotrzeć.

$$L(i,a) = min\{L(j,b) + w_j(a - b + (a_i - a_j))\}$$
$$(i,j) \in E,\ b \in \{0, ..., W\}$$

Powtarzać punkty 2-5. Wybrane stacje i ilość zatankowanego paliwa zapisywać do kolekcji. Uogólniając, pierwszy składnik min sprawdzamy W razy dla ilości posiadanego paliwa. Ponawiamy to dla wszystkich n stacji, czyli O(W² * n) = O(n)

Random Select

Dzielimy względem losowego elementu jak QS. Wywołujemy rekurencyjnie algorytm dla tej części, w której znajduje się żądana statystyka pozycyjna, z indeksem i (lewa) lub i-k (prawa).

Select MOM

Dzielimy względem losowego elementu jak QS. Wywołujemy rekurencyjnie algorytm dla tej części, w której znajduje się żądana statystyka pozycyjna, z indeksem i (lewa) lub i-k (prawa).

Tworzenie kopca

```
max-heapify(A,i)
  l = left(i)
  r = right(i)
if l <= A.size() and A[l] > A[i]
  largest = l
else largest = i
if r <= A.size() and A[r] > A[largest]
  largest = r
if largest != i
  swap(A[i], A[largest])
```

Sumy skończone

sum(i*q)=q*(sum(q))

$$\sum_{i=1}^{n} i = \frac{n(n+1)}{2}$$

$$\sum_{i=1}^{n} i^3 = \frac{n^2(n+1)^2}{4}.$$

$$\sum_{i=1}^{n} i^2 = \frac{n(n+1)(2n+1)}{6}$$

$$\sum_{i=0}^{n} c^i = \frac{c^{n+1}-1}{c-1}, \quad c \neq 1,$$

$$\sum_{i=0}^{n-1} i2^i = 2(n-2)2^n$$

$$\sum_{i=0}^{n-1} x^i = \frac{1-x^n}{1-x}$$

$$\sum_{i=0}^{n-1} \frac{i}{2^i} = 2 - \frac{n+1}{2^{n-1}}$$

$$\sum_{i=0}^{n-1} 2^i = 2^n - 1$$

Algorytmy grafowe

```
Prim(G): O(|E|*log|V|), Fibon. O(|E| + |V|*log|V|)
- wybierz wierzchołek start. A
- na kolejce priorytet. dodaj
wierzchołki osiągalne
-wybierz „najtańszą” ścieżkę,
która prowadzi do nowego
wierzchołka i znowu kolejkuj
```

```
topological_sort(G):
- wykonaj DFS w celu obliczenia
czasów przetworzenia v.f dla
każdego wierzchołka
- wstaw każdy wierzchołek v na
początek listy
- return lista wierzchołków
```

```
BFS (G, s)                    O(|V| + |E|)
    let Q be queue
    Q.enqueue( s )
    mark s as visited.
    while ( Q is not empty)
        v  =  Q.dequeue( )
        for all neighbours w of v in Graph G
            if w is not visited
                Q.enqueue( w )
```

naiwnie O(|V|^2) - gęsty
kopcem O(|E|*log|V|) - rzadki
Fibon O(|E| + log|V|)

```
function Dijkstra(Graph, source):
      dist[source] ← 0
      create vertex queue Q
      for each vertex v in Graph:
          if v ≠ source
              dist[v] ← INFINITY
          prev[v] ← NULL
          Q.enqueue(v, dist[v])
     while Q is not empty:
         u ← Q.dequeue()
         for each neighbor v of u:
             alt ← dist[u] + length(u, v)
             if alt < dist[v]
                 dist[v] ← alt
                 prev[v] ← u
                 Q.decrease_priority(v, alt)
return dist, prev
```

```
DFS()                          O(|V| + |E|)
    clock=0
    For each u ∈ G
        u.visited == false
    For each u ∈ G
        if u.visited == false
            DFS(G, u, clock)
```

```
explore(G, u, *clock)
    u.visited = true
    u.pre = *clock++
    for each v ∈ G.Adj[u]
        if v.visited == false
             explore(G,v)
    u.post = *clock++
```

```
stronglyConnectedComponents(G):
1. wykonaj DFS(G), licząc czas u.f
2. oblicz transpozycję GT
2. wykonaj DFS(GT), w kolejności
według u.f malejąco
4. wypisz drzewa z lasu pkt3 jako
oddzielne silnie spójne składowe
```

Sumy do ∞

$$\sum_{i=0}^{\infty} c^i = \frac{1}{1-c},$$

$$\sum_{i=1}^{\infty} c^i = \frac{c}{1-c}, \quad |c| < 1,$$

Sortowania stabilne - zachowują kolejność elementów

	time	mem.	algorytm
bąbelkowe (bubble)	$O(n^2)$	O(1)	porównaj dwa kolejne elementy i zamień je, jeśli zaburzają porządek w tablicy
wstawianie (insertion)	$O(n^2)$	O(1)	do zbioru elementów już posortowanych wstaw element na odpowiednią pozycję
scalanie (merge)	O(n log n)	O(n)	podziel na dwie równe części, posortuj obie przez scalanie (jeśli więcej niż jeden element), połącz sortując
zliczanie (counting)	O(n+k)	O(n)	dla każdego elementu znajdź liczbę elementów od niego mniejszych, z tego oblicz pozycje w gotowej tablicy
kubełkowe (bucket)	$O(n^2)$	O(n)	elementy podziel między kubełki według zakresu, posortuj niepuste kubełki (przez wstawianie), połącz w jeden ciąg
pozycyjne (radix)	O(d(n+k))	O(n)	sortuj kolejno według coraz bardziej znaczących cyfr

Sortowania niestabilne

wybieranie (selection)	$O(n^2)$	O(1)	dla każdego elementu wyszukaj minimalną wartość od i do końca, zamień ten element z A[i]
szybkie (quicksort)	O(n log n) pes. $O(n^2)$	?	rozdziel tablicę na część mniejszą i większą względem pivota, obie posortuj quicksortem
szybkie z 5 (MOM)	O(n log n)	?	jak wyżej, ale z gwarancją dobrego podziału
kopiec (heapsort)	O(n log n)	O(n)	budowanie kopca z tablicy

Szybkie mnożenie Karacuba

```
pomnóż(a,b){            //m-cyfrowe
  rozdziel a na a1 i a2 // L i P
  rozdziel b na b1 i b2 // L i P
  u=pomnóż(a1, b1);
  p=pomnóż(a1+a2, b1+b2);
  z=pomnóż(a2, b2);
  wynik = u*10^2m + (p-u-z)*10^m + z;
    return wynik;
}
```

Największa suma (Kedane)

```
max_local = max_total = A[0]
for x in A[1:]:
 max_local = max(x, max_local + x)
 max_total = max(max_total, max_local)
return max_total
```

Logarytmy

$\log_b(a) = x \Leftrightarrow a^x = b$
$\log_b(mn) = \log_b(m) + \log_b(n)$
$\log_b(m/n) = \log_b(m) - \log_b(n)$
$\log_b(mn) = n*\log_b(m)$
$\log_b(a) = \log_x(a)/\log_x(b)$
$a^{(\log_c b)} = b^{(\log_c a)}$

Stirlinga

$$n! \approx \sqrt[n]{2\pi n}\left(\frac{n}{e}\right)$$

Master Theorem (Cormen)

$$T(n) = \begin{cases} O(n^{\log_b(a)}) & f(n) = O(n^{\log_b(a)-\epsilon}), b>1, \epsilon>1 \\ O(n^{\log_b(a)}\log n) & f(n) = \Theta(n^{\log_b(a)}), b>1 \\ O(f(n)) & f(n) = \Omega(n^{\log_b(a)+\epsilon}), b>1, \epsilon>1 \end{cases}$$

T(n) = a T (n/b) + f(n)

Notacja asymptotyczna + MasterTheorem (W)

$3x^2 = O(x^2)$, $2x = O(x^3)$, $x^2 \not= o(x^2)$, $2x = o(x^2)$ $3x^2 = \Omega(x^2)$, $2x^3 = \omega(x^2)$

f = O(g(x)) -> f ograniczone z góry przez c*g(x) (c - stała)
f = o(g(x)) -> f ostro ograniczone z góry
f = Ω(g(x)) -> f ograniczone z dołu przez g(x)
f = ω(g(x)) -> f ostro ograniczone z dołu
f = Θ(g(x)) -> f ograniczone z góry i z dołu, czyli rzędu O(g(x)) i Ω(g(x))

Skrót:
f = O(g) ⇔ lim f(x)/g(x) < ∞
f = o(g) ⇔ lim f(x)/g(x) = 0
f = Ω(g) ⇔ lim f(x)/g(x) > 0
f = ω(g) ⇔ lim f(x)/g(x) = ∞
f = Θ(g) ⇔ 0 < lim f(x)/g(x) < ∞

$$T(n) = \begin{cases} O(n^d) & d > \log_b a \\ O(n^d \log n) & d = \log_b a \\ O(n^{\log_b a}) & d < \log_b a \end{cases}$$

T(n) = a T (n/b) + f(n^d)

Algorytm Huffmana:

Jeden kod nie może być prefiksem drugiego.
Najkrótszy kod to ten co się pojawia najczęściej.
Liczba bitów konieczna do zakodowania słowa: $\Sigma_i f_i * (\text{byteSize})_i$

```
H-kolejka priorytetowa elementów z alfabetu, gdzie f_i
(częstotliwość występowania) jest priorytetem
  for i=1 to n
    insert(H, (i, f_i))
  for k=n+1 to 2n-1
    i = extract_min(H)
    j = extract_min(H)
    k = nowy element, którego dziećmi są i oraz j
    insert(H, (k, f_i + f_j)
```

Intuicyjnie - określ prawdopodobieństwo występowania każdego elementu, utwórz dla niego drzewo w lesie, w każdym kroku złącz dwa drzewa o najmniejszym prawdopodobieństwem, w korzeniu zsumuj ich prawdopodobieństwa.

Longest Common Subsequence

Znajduje długość najdłuższego wspólnego podciągu ciągów X[1..m], Y[1..n]. Algorytm wykorzystuje rekurencyjne wywołania aby znajdować LCS dla prefixów X[1..i], Y[1..j].

```
mLCS(x, y, i, j):
  if(c[i, j] = null):
    c[i, j] = LCS(x, y, i, j)
  return c[i, j]
```

```
LCS(x, y, i, j):
  if(i = 0 or j = 0):
    c[i, j] = 0
  else if(x[i] = y[j]):
    c[i, j] = 1 + LCS(x, y, i-1, j-1)
  else:
    c[i, j] = max {LCS(x, y, i-1, j), LCS(x, y, i, j-1)}
  return c[i, j]
```

mLCS - standardowy LCS ma złożoność wykładniczą (jak rekurencyjny Fibbonacci). Dla wyeliminowania problemu wielokrotnych wywołań z tymi samymi parametrami można zastosować mLCS, które wykorzysta już wyliczone wartości. Należy zastąpić rekurencyjne wywołania w LCS wywołaniami mLCS. Złożoność obliczeniowa O(n*m).

Algorytm Kruskala (zabrakło miejsca w grafowych)

```
for all u in V
  makeset(u) //Disjoint Sets
X = {} //zbiór krawędzi wynikowego MST
posortuj krawędzie z E niemalejąco względem wag
for all {u, v} in E w porządku niemalejącym
  if find(u) != find(v)
    X = X U {{u, v}}
    union(u, v)
```

Kolejka priorytetowa

łączenie O(n), max O(1), pozostałe O(log n)

`insert(Q,x)` - wstawia element x do zbioru Q
`maximum(Q)` - zwraca element o największym priorytecie
`extract-max(Q)` - jak maximum, ale dodatkowo usuwa
`decrease-key(Q,i)` - zmniejsza priorytet elementu Q[i]
`union(Q1,Q2)` - łączy dwie kolejki `delete(Q,i)` - usuwa Q[i]