Generikus vector és kupac, alkalmazva rendezésre és dijkstra algoritmusra

Szerkezetek rendelkeznek virtuális függvényekkel a származtatás érdekében. Ezeket V-vel jelölöm.

Vector

- `int` length() V
- resize(size, defaultValue) V
- clear() V
- pushBack(T) V
- T popBack() V
- operator=(Vector) V
- operator={} V
- operator[] V
- operator+(T) V
- operator+(Vector) V
- operator+=(T) V
- operator+=(Vector) V
- Vector()
- Vector(size, defaultValue)
- Vector(Vector)
- Vector({})
- `static` Test()
- kívül
    - operator+(T, Vector)
    - operator<<
        - formátum: "[méret](1.elem, 2.elem, ...)"

BinaryHeap

- clear() V
- insert(T) V
- insert(Vector) V
- T top() V
- T pop() V
- bool empty() V
- int length() V
- BinaryHeap()
- BinaryHeap(Vector)
- operator+(T) V
- operator+(Vector) V
- opartor+=(T) V
- opartor+=(Vector) V
- static Sort(Vector)
- static Test()

- kívül
    - operator(T, BinaryHeap)
    - operator(Vector, BinaryHeap)

Dijkstra

- int length()
- Vector parents
- Vector weights
- printPath(int to) `V`
- Dijkstra(filename, startIndex)

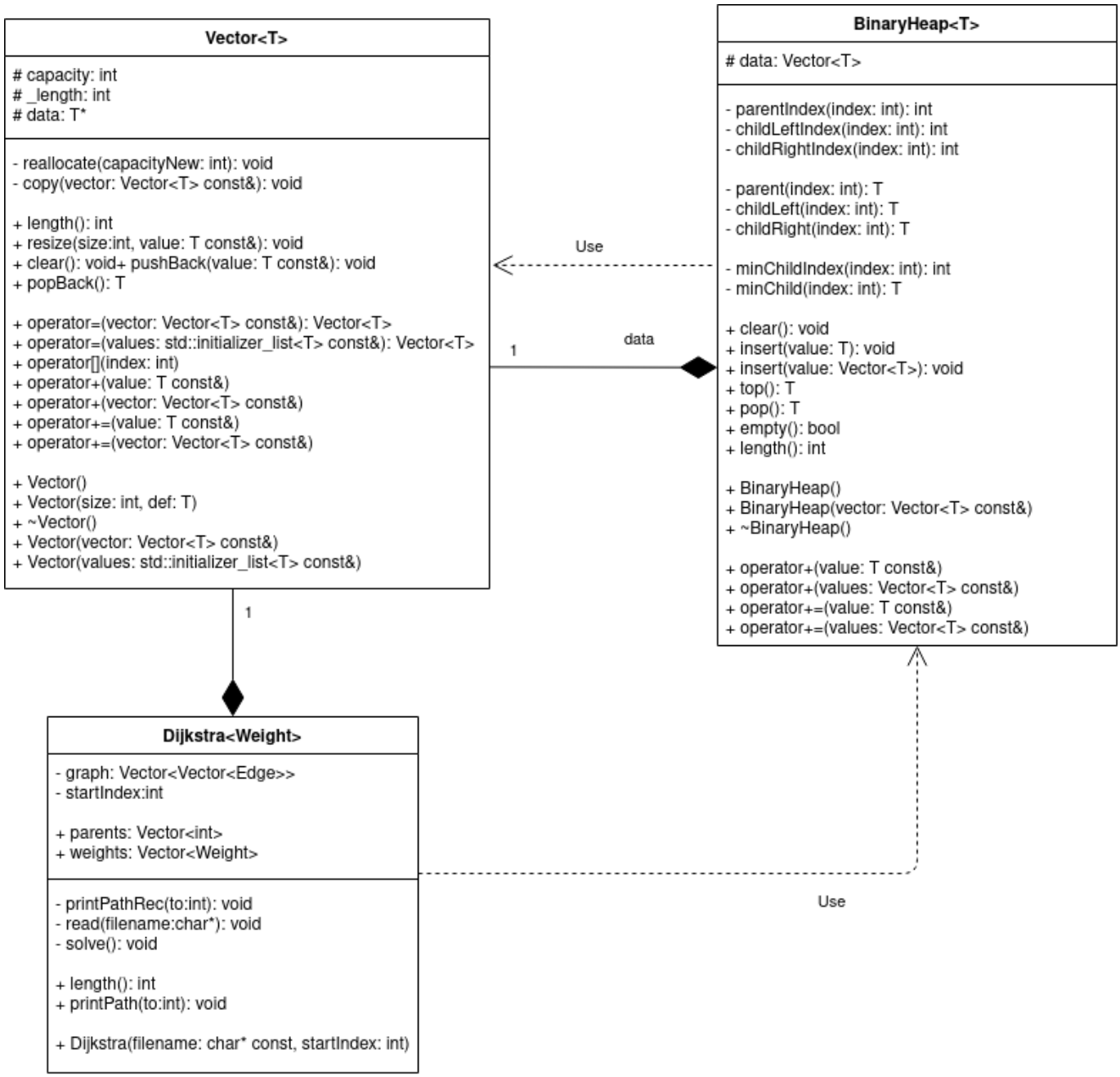
Dijkstra esetében a rejtett `read(filename)` és `printPathRec(int to)` is virtuális a bővíthetőség érdekében.

Az egyes adatszerkezetek ha tartalmaznak `static Test()`-et, akkor annak célja, hogy változatos eseteket, funkcionalitásokat kipróbáljanak és assert-tel leteszteljék azokat.

Ezekhez a szerkezetekhez Exception is tartozik, melyeket az őket tartalmazó namespacebe helyezek (kivétel Dijkstra...), hogy más megegyező funkcionalitású adatszerkezetek is felhasználhassák ezeket.

Feladat szintén egy pl egész számokat tartalmazó fájlt vektorba beolvasni, majd BinaryHeap-pel sortolva kiírni a beépített `operator<<`-ral. Szintén kell egy a Dijkstrát kipróbáló rész, ahol az egyik esetben beolvassa a fájlt (két különböző súlytípusú példa kell), majd kiírja a legrövidebb utakat az összes csúcsba, míg másik esetben detektálja, hogy nem alkalmazható az algoritmus.

UML



Algoritmusok

Vector

- pushBack

> A működés lényege, hogy a lefoglalt terület nem egyezik meg a tárolt adatokkal. Ez azért kell, hogy ne kelljen minden beszúrásnál újrafoglalni területet és másolni. A lefoglalt területet `capacity`-nek nevezem. Ez mindig kettő hatvány lesz, így ha nem lenne elég a capacity a beszúrandó elemnek, akkor megduplázom azt.

- popBack

> Hasonló a pushBack lényegéhez, azonban itt meg azt nem akarjuk, hogy ha pont egy másik kettő hatvány szakaszba esik át, akkor azonnal csökkentsük a capacityt, mivel gyakran ezután szúrunk be adatot, fölöslegessé téve a csökkentést. Így azt a módszert választottam, hogy ha

> már 2-vel alatta lévő 2 hatvány szakaszba esik `capacity == 4 * _length`, akkor dönt úgy hogy csökkenti a területét.

> A többi függvény nem tartalmaz új algoritmust, hanem inkább a meglévő szintén egyszerű függvényekre támaszkodik, ami bár nem a legeffektívebb, de sokkal nagyobb hibamentességet és átláthatóságot garantál.

BinaryHeap

> Az adatstrúktúra egy Vector-t használ fel, hogy egy bináris fát tároljon. Az algoritmusok teljes egészében megegyeznek a közismert algoritmusokkal.

- insert

  > Az adatot beszúrjuk a vector végére, ezzel garantálva, hogy levél legyen, és hogy a kiegyensúlyozott bináris fa tulajdonságot fenntartsuk. Ezután felfele buborékoztatjuk a fán, amíg a szülője nagyobb nála. Valójában csak a szűlőket mozgatjuk lefele a megfelelő helyekre és amikor a ciklus véget ér, akkor szúrjuk be az elemet.

- pop

  > Ilyenkor a legfelső elemet kivesszük, hiszen ez a legkisebb elem. Ezután a két gyerek közül a kisebbet felülre buborékoztatjuk, hiszen ez a másik ág minden eleménél is kisebb lesz. Ezt folytatjuk az üres helyeken, amíg van gyerek, hiszen addig sérül a gráf tulajdonság.

Dijkstra

> Az adatstruktúra Vector-t használ fel a gráf tárolásához, és a legrövidebb út meghatározásához pedig BinaryHeap-et. Az algoritmusok teljes egészében megegyeznek a közismert algoritmusokkal.

- read

  > A megadott fájlból beolvassuk a gráfot a `graph` szomszédsági listába. A fájl először a csúcsok számával kezdődik, majd n sorban először a fokszám található, m, majd m csúcs amihez vezet él.

- solve

  > Először berakunk a heapbe egy élélet, amit a kezdőd csúcsba mutat 0 távval és a -1-es nem létezú forrás csúccsal. Ezután minden iterációban kivesszük a legközelebb lévő elemet a kezdődcsúcstól nézve, lementjük annak távolságát, illetve hogy melyik már ismert távú csúcsból jutott el ide. Végül berakjuk azon szomszédjait, melyek távja még nem ismert. Ez alatt azt értjük, hogy egy élet rakunk a heapbe, ami megegyezik a szomédba mutató éllel, a súly kivételével, ugyanis annak a jelenlegi csúcsba való táv + él súlya súlllyal kell megegyeznie. Teljes indukcióval belátható, hogy az algoritmus helyes, ha a súlyok összeg függvénye monoton növekszik. Ezt a feltételt természetesen ellenőrzi az algoritmus

- printPath

  > Ez egy preorder rekurzív algoritmus. Először megtudjuk az utat a szülőig, majd kiírjuk a jelenlegi csúcsot. Bázis lesz a kezdő csúcs. A formátum: "(út hossza): csúcs1 csúcs2 ...", ahol a csúcs

> felsorolás egy szimpla "-" ha nem vezet oda út. Az utóbbi a `parent` és `weights` tömbökben, amit a csúcsindexszel lehet indexelni, úgy mutatkozik meg, hogy -1 szerepel az adott helyen.