Henri Korpela

Toteutus

Kaikissa tietorakenteissa solmut joihin data talletetaan on toteutettu niin että solmu sisältää sekä key kentän ja value kentän. Poikkeuksena lista, jossa key:n ajatellaan olevan indeksi. Kaikki tietorakenteet toteuttavat rajapinnan Set, joka sisältää joitain joukkojen tarkastelun kannalta oleellisia operaatioita. Ohessa on käyty läpi kyseisen rajapinnan tarjoamat metodit ja niiden aikavaativuudet. Nämä metodit ovat ohjelman kannalta keskeisiä, sillä ohjelma vertailee juuri näitä metodeita eri tietorakenteilla.

add:

Add toimii kaikilla puilla logaritmisessa ajassa. Listassa add toimii ajassa O(n). Hash mapissa add toimii keskimäärin vakio ajassa. Hash mapissa ylivuoto lista käydään läpi, jotta tunnistettaisiin onko lisättävä solmu jo olemassa(jos on päivitetään solmun arvo). Jos hash taulun koko ja talettetavan objectin hash funktio ovat järkeviä on ylivuotolistojen koko yleensä pieni.

add_identical:

Add_identical toimii kaikilla tietorakenteilla samassa ajassa kuin add.

get:

Get toimii puilla samassa logaritmisessa ajassa. Listalla get toimii ajassa O(n). Hash mapissa get toimii pahimmassa tapauksissa ajassa O(n). Tämä tapahtuisi käytännössä tilanteissa, jossa hash taulunkoko on yksi (tai hyvin pieni) ja/tai tallennettavan objektin hash funktio on huono. Eli hash mapilla get toimii keskimäärin ajassa O(1).

remove:

Remove toimii puilla ajassa O(log n). Listassa Remove toimii ajassa O(n). Hash mapissa pahimmassa tapauksessa remove kuten get toimii pahimmillaan., sillä tällöin täytyy koko ylivuotolista käydä läpi, jotta poistettava löytyisi. Hash mapilla kuitenkin siis keskimäärin ajassa O(1).

predecessor:

Predecessor toimii puilla ajassa O(log n) (poikeuksena punamustapuu). Listalla ja hash Mapilla ajassa O(n). Punamustan puun tapauksessa puu toteutettiin ilman että solmut tietävät omaa vanhempaansa, jolloin predecessor toteutettiin listan avulla, jolloin kaikki puun solmut joudutaan käymään läpi.

successor:

Successor toimii kaikilla tietorakenteilla samassa ajassa kuin predecessor.

min:

Min toimii puilla ajassa O(log n). Listalla ja Hash mapilla ajassa O(n).

max:

Max toimii puilla ajassa O(log n). Listalla ja Hash mapilla ajassa O(n).

contains:

Contains toimii kaikilla tietorakenteilla ajassa O(n), sillä containsissa tutkitaan onko millään solmulla value:ta, joka vastaa anettua arvoa. Sen sijaan että tutkittaisiin sitä sisältääkö rakenne solmua, jolla on annettu avain.

contains_key:

Contains key toimii puilla ajassa O(log n). Listalla ajassa O(n) ja hash mapilla keskimäärin ajassa O(1) (riippuen hash taulun koosta ja talletettavan objektin hash funktiosta). Contains toimii puilla contains metodia nopeammin, koska contains keyssä tutkitaan sisältääkö puu annettua avainta (ei siis arvoa kuten containsissa) ja tällöin puun hakupuurakennetta voidaan käyttää hyödyksi.

clear:

Clear toimii kaikilla tietorakenteilla vakioajassa.

size:

Size toimii puilla ajassa O(n), listalla ja hash mapilla ajassa O(1). Puiden hitaus johtuu siitä että niissä ei tässä toteutuksessa ole laitettu omaa muuttujaa koolle vaan aina pyydettäessä puun koko lasketaan uudelleen käymällä puu läpi. Puun koko olisi siis voitu saada haluttaessa vakioaikaiseksi. Hash map ja lista sisältävät oman muuttujan koolle jota kasvatetaan lisäyksessä, vähennetään poistettaessa ja nollataan clearin yhteydessä.

Tietorakenteet siis toimivat eri nopeuksilla eri metodien yhteydessä. Puista huomioitavaa on se että punamusta puu on nopein (lukuun ottamatta successor ja predecessor operaatioita. Tämä johtuu kuitenkin toteutuksesta). Vaikka myös AVL-puu on tasapainossa, kuten punamustapuu on punamustan puun add ja remove operaatio kevyempi kuin AVL-puun. Aikavaativuudeltaan molemmat puut ovat kuitenkin yhtä nopeita.