KODUTÖÖ NR. 5 ALGORITMID JA ANDMESTRUKTUURID

ÜLESANNE 1 - kood GitHubis

ÜLESANNE 2 KUHJA STRUKTUURI ANALÜÜS

Kuhjasid kasutatakse kiireaarpuu loogikas sõlmede ümberkorraldamisel. Min- ja max kuhi peamine erinevus seisneb selles, kas kiireaarpuu juure on kõige väiksem või kõige suurema väärtusega element.

Min-kuhi:

1. juurväärtus / võti peab olema väiksem või võrdne kõigist kuhjas olevatest elementidest →
2. kõige väiksem element on juurväärtus
3. kuna kõige väiksemad elemendid on kuhjast ehkimisel esipool siis kuitahes uued kõige kiiremini

Max-kuhi:

1. juurväärtus peab olema suurem või võrdne kõigis kuhjas olevatest elementidest, sega...
2. kõige suurem element on juurväärtus.
3. max-kuhja puul on kõige suuremad elemendid eespool otsimise järjekorras, mis tähendab, et nende leidmine võtab kõige vähem aega.

Kuna elementide järjestus mõlemas kuhjas on vastupidine teisele siis oleneb, kus või milline on selle rakendamise eesmärk. Kuna elementide järjestamine ei muuda loogikat kuidas mingit elementi otsitakse siis peab aja ja ruumikeerukus olema sama. Muidugi tuleb arvestada, et kui koodi lisada mist üks siis millised väärtuseid enim võidakse otsida. Ajaline keerukus on siiski nii min kui ka max kuhjal O(log(n)) ja ruumiline O(n). Kuhi jääb ikka sama suureks, vahet pole, kas seda sorteeritakse suuremast väiksemani või vastupidi. Küll on aga oluline kuidas neid valida prioriteedijärjekordade haldamiseks. Min-kuhja oleks mõistlik kasutada olukordades, kus tuleb leida väikseimad väärtused, näiteks otsida veebipoest mingi kategooria odavaimat toodet. Või leida Dijkstra algoritmiga kõige lühem tee punktist A punkti B. Max-kuhja võib samuti kasutada veebipoes aga sellega kõige kallima toote otsimiseks. Mõlemal struktuuril on seega oma eelis, mida tuleks koodi implementeerimisel silmas pidada.

ÜLESANNE 3. BINAARSE OTSINGUPUU ANALÜÜS

Binaarne otsingupuu on andmestruktuur, kus andmed on organiseeritud hierarhiliselt, et neid oleks hiljem lihtne otsida, eemaldada või lisada. Elemente saab hakata lisama ükshaaval ning neid paigutatakse puud moodustavasse struktuuri. Vastavalt sellele, kas lisatava elemendi väärtus on suurem, väiksem või võrdne juurväärtusega võrreldes. Binaarpuu koosneb juurväärtusest/sõlmest ning sõlmedest/elementidest, mida sinna lisatakse. Juhul kui lisatav sõlm on väiksem kui juur, lisatakse see puu vasakusse harusse. Juurest suuremad aga parempoolsesse. Elemendi otsimisel binaarsest otsingupuust tuleb kasutada veel funktsiooni, mis võrdleb elemente omavahel ~~ning~~ tagastab soovitud tulemuse.

Tasakaalustamata puud tekivad kui andmestruktuuri lisatakse järjestikuseid elemente, mis muudavad puu elementide paigutuse tõttu kõrgeks. Tekib olukord, kus binaarne otsingupuu ei täida enam oma efektiivsuse eesmärki ning muutub rohkem lineaarseks listiks. Sellepärast oleks mõistlik kasutada tehnikaid/lisafunktsioone, mis puu struktuuri hoiaks. Näiteks saab kasutada punasemusta puud, mis kontrollib elementide lisamisel reegli, et ühe sõlme all ei tohi olla rohkem kui 2 järjestikust „laps" sõlme. ~~Vastavalt ja~~ Sellisel juhul tehakse ~~vaa~~ pööre ning tagatakse puu tasakaal. Veel kasutatakse ka AVL puud, kus pööramist rakendatakse veel enam, et lineaarset struktuuri jälgida/hoida.

ÜLESANNE 4. PUNASE-MUSTA PUU ÜLEVAADE

Punase-musta andmestruktuuri kasutatakse selleks, et voida binaarset otsingupuud tasakaalus läbi järgmiste omaduste / reeglite:

1. juurväärtus on must
2. iga sõlm millel on 2 laps-sõlme on must
3. punase sõlme laps-sõlmed on mustad
4. igal sõlmel on sama sügavus
5. teekond juursõlmest järgmise musta sõlmeni võtab sama palju musti sõlmi
6. tasakaalu säilitamiseks vahetatakse mustade ja punaste sõlmede asukohti ja värve.

Binaarse otsingupuu kõige suurem miinus oli see, et ta võib muutuda liiga kõrgeks ja otsingu aja-keerukus võib olla siis O(n). Punase-musta puu omadustega saab seda olukorda vältida nii, et ajakeerukus jääb siiski O(log N). Seega punase-musta puu rakendamine teeb andmestruktuuri oluliselt tõhusamaks ja stabiilsemaks.

Värvireeglid aitavad anda sõlmedele mingid lisa-omadused, millega puu tasakaalustamisel arvestada ja programmi kirjutada. See aitab ka paremini binaarpuu loogikast aru saada.

ÜLESANNE 5. AVL PUU VS. ~~PM~~ PUNASE-MUSTA PUU

AVL peamiseks omaduseks on see, et elementide lisamisel struktuuri hoitakse puud tasakaalus. Kummagi haru sügavus on alati 1 või väiksem kui üks, see reegel kehtib ka iga haru parema ja vasaku elemendi tasakaalu kohta. Juhul kui element lisatakse juure all olevasse sõlme, keeratakse puud vastavalt kas paremale või vasakule poole. Samal ajal võrreldakse elemente ka omavahel, vahetatakse nende kohti ning seejärel sooritatakse vastavalt kas parem- või vasakpööre.

Proovisin AVL ja PM puu visualiseerimise lehel järgi leidas kumbki andmestruktuur näitab. Peale kuue sama elemendi lisamist oli näha leidas. AVL puu suutis tänu oma reeglitele (pööramisele, elementide omavahelise järjestuse muutmisele ja maksimaalselt 1 võrra erineva sügavuse hoidmisele harude vahel) paremini tasakaalu säilitada kui PM puu. Teoreetiliselt (ja natuke ka katseliselt) saab hinnata AVL puud tõhusamaks kuna näiteks elemendi otsimise tasakaalustatud puust on oluliselt kiirem kui tasakaalustamata puust. Samas elementide lisamisel AVL puusse võib reeglite pärast rohkem aega minna. Kuid enamasti peaks otsinguid rohkem olema kui lisamisi, siis PM puule oluliselt eelist ei anna.

Kui AVL on parem kui andmestruktuurist elemente pigem otsida kui elemente lisada, kustutada siis seda tasub eelistada PM puule rakendustes, kus on palju elemente. Näiteks sõnaraamatu või EKSS, või sarnaste sõnastike puhul oleks mõistlikum

kasutada pigem AVL puud. Andmed muutuvad suhteliselt vähe aga peamine otstarve on siiski lihtsalt sõnade otsimine. Alguses võib muidugi olla ajakulukas kõike sisestada aga kuna rakenduses ei ole vaja andmeid dünaamiliselt muutumises hoida, tasuks sellises olukorras AVL puud ehitada. Veel võiks ka navigatsiooni- või kaardirakendused samamoodi töötada - palju andmeid, pigem nimede otsimiseks, peaks kiirelt vastuse saama.

Punase-musta puud tasuks ehitada juhul kui rakendus muutub sageli - elemente lisatakse, kustutatakse aga samas juhul kui tekib vajadus, saab andmeid ka lugeda, ehk rakendus ei tohiks olla liiga suur. Näiteks arvutis failide haldamise süsteem. Kuna faile otsitakse, muudetakse, kustutatakse sageli. Lisaks tuleb tagada tõhus ja stabiilne struktuur elementide/failide otsimiseks. Sama võiks kehtida ka meilirakenduste puhul.

BOONUSÜLESANNE

Eelmised küsimused käsitlesid juba AVL ja PM puud ja nende reegleid. Seega enne võrdlust veel ülevaade Splay ja B puust... enne korrektset võrdlust.

Splay puu peamiseks omaduseks on see, et hiljuti lisatud element paigutatakse andmestruktuuris juursõlme lähedale. Elementide paigutus muutub dünaamiliselt vastavalt sellele kui tihti seda otsitakse. Peale elemendi üles otsimist ja ümberpaigutamist kontrollitakse ülejäänud puu tasakaalu nagu samamoodi AVL ja PM puule. pööratakse harus elemente vastavalt vajadusele. Seega antud

andmestruktuuri sobib hästi rakenduse, mis vajab kiirjutisele otsingule optimeerimist.

B-puu peamiseks omaduseks on see, et see suudab hakkama saada suure andmemahuga läbi rohkemale võtmele ühes sõlmes. See tagab tõhusa andmele otsimise ja mälukasutuse. Kuna harusid ühe sõlme all saab olla oluliselt rohkem on sama kuluga andmete hoidmisel B-puul oluliselt väiksem kõrgus võrreldes näiteks PM-puuga.

	MÄLU	LOOKUP	INSERT	DELETE
AVL puu	$O(n)$	$O(\log n)$	$O(\log n)$	$O(\log n)$
Punase-Musta puu	$O(n)$	$O(\log n)$	$O(\log n)$	$O(\log n)$
B-puu	$O(n)$	— " —	— " —	— " —
Splay puu	$O(n)$	~~[illegible]~~ N/A	— " —	— " —

Üks kirjutatud kriteeriumide järgi saaks igat tüüpi puud tabelis/andmestruktuurina kasutada. Paistab, et igale kindlale rakendusele on võimalik parim lahendus valida kui väga täpselt on teada, mis andmeid, kui palju sarnaseid/samu väärtuseid, millised on otsingu kiiruse, elementide lisamise, eemaldamise tõenäosused. Parima lahenduse valimine tähendab väga selget eesmärgi ja andmestruktuuri kasutamise defineerimist. Näiteks võtta veebipood, mis müüb võivaid, saapaid, aksessuaare, igat kategooriat enam-vähem ühepalju. Tehakse AVL puu andmestruktuuri-ga kiire kriteeriumide leidmine, puud pole vaja.

ka regulaarselt tasakaalustada. Sama hästi saaks kasutada ka PH puud - kiire, stabiilne, kliendid saavad kiiresti vastuse kui otsivad nt märksõna "talvejope". Nüüd äkki sama veebipood müüb peamiselt saapaid ja muuseas ka rõivaid, aksessuaare. See tähendab, et otsingule kiirendamiseks saab kasutada Splay puud. Sama ka näiteks kui eelnevalt on teada, et kliendid otsivad "uusi" asju, mis alles süsteemi sisestati. Nüüd kui antud veebipood oleks aga tohutu kaubavalikuga toidupoe e-pood, peaks kasutama B-puud. Sellisel juhul on väga palju eri tooteid, nt. näiteks kümme eri sorti tomateid jms siis peab saama kiiresti vastuse. Edasiarendusteks B-puu korral näiteks ladude andmebaasides - samuti palju andmeid (kõige selle säilitamiseks on vaja mõistlikult serveri ressursse kasutada, mis tähendab, et otstarbekas oleks säilitada andmeid nii, et see võtaks võimalikult vähe mälu).

Erinevaid rakendusi veel:

Instagram - pidevalt lisatakse uusi kasutajaid, kustutatakse vanu, otsitakse kedagi nime järgi. Kõigi kolme on suhteliselt sama palju seega püsiva jõudluse tagamiseks on mõistlik kasutada PH-puud. Splay puu ei anna otsest eelist selles olukorras, sest võib juhtuda et otsing ei anna tulemust (rakendusel läheb liiga kaua tulemuse saamiseks kuni ehk on juba timeout). Samuti võib kasutada ka B-puud kuna see on tõhus suurele andmehulkade puhul ning hoiaks andmemahu mälus kokku.

Ilmateeninduse rakendus – ka lihtne oleks kasutada Splay puud, sest enamasti otsitakse ilmaandmeid käesoleva või järgneva päeva kohta. Andmed ise aga uuendatakse ka kas päev või paar varem. Kuna lisatud element paikneb võrdlemisi lähedal juursõlmele on Splay puu otsing kiirem võrreldes AVL või BM puuga.