6: Søkestrukturer

mandag 21. september 2020

Oppsplitting i videoer

1. Hva er en søkestruktur?
    a. ADT-en Map med alternative navn (Python dictionary)
    b. ADT-en set (mengde)
    c. Undertyper for sorterte Map og mengder
2. Naiv sorted map implementasjon basert på sortert liste (liste som holdes sortert)
    a. Vis kjøretider for ei ikke-sortert liste og vis at den er strengt dårligere. Add må likevel gå gjennom lista for å se etter duplikater!
3. Hashtabell
4. Hash-basert Map implementasjon, basis uten kollisjonshåndtering
5. Basis Kollisjonshåndtering og rehashing
6. Flere teknikker for å håndtere kollisjoner i hashtabeller

Innhold i temaet

- Hva er en søkestruktur
- Abstrakte datatyper for søkestrukturer
- Naiv implementasjon
- Hash-basert implementasjon
- Merk: Tre-basert implementasjon kommer i neste tema, «Trestrukturer»

Hva er en søkestruktur?

- Struktur for å finne objekter i en samling basert på et kriterium
    - Eksempel Python dictionary
- Strukturer for mengder (Python set) er ofte lagd basert på en søkestruktur siden søkestrukturer er flinke til å finne duplikater

Abstrakt datatype: Map

Et map er et objekt som lagrer nøkkel,verdi par, og som kan finne verdien basert på nøkkelen etterpå. Det blir på en måte ei liste som kan ta andre ting enn tall som indekser. Et Python dictionary er et eksempel på denne ADT-en.

- **Get**(nøkkel) -> Verdi. Henter ut verdien som er lagret for en bestemt nøkkel
- **Put**(nøkkel, verdi). Setter en nøkkel til å referere til en verdi. Legger inn en ny verdi hvis nøkkelen ikke fins fra før. Overskriver eksisterende verdi hvis nøkkelen finnes fra før
- **Delete**(nøkkel): Sletter en nøkkel og dens verdi fra samlingen
- **Contains**(nøkkel): Sjekker om samlingen inneholder en gitt nøkkel. Dette er måten **in** operatoren fungerer på i Python for dictionaries.
- **Iterator**: Det skal være mulig å lage en iterator for map-et. Den skal returnere nøkler. Dette er måten iteratorer fungerer på for Python dictionaries.

Merk: Hva som er lovlige nøkler er avhengige av hvordan map-et er implementert. To implementasjoner vil bli vist i dette faget som stiller ulike krav til nøkkelen. Begge krever imidlertid at nøkkelen er et objekt som ikke endrer seg. Grunnen er at hvis nøkkelen endrer seg uten at map-

implementasjonen får vite det, så vil objektet nå ligge på feil sted! Dette er grunnen til at Python lister ikke kan brukes som nøkler. Nøklene må også være unike, et Map tillater ikke duplikater. Setter du inn en ny verdi for en gitt nøkkel i et Python dictionary, erstattes den gamle verdien med den nye. Dette er vanlig oppførsel i Map implementasjoner.

Undertype: Sorted Map

Av og til ønsker man et map som holdes sortert på nøkkelen. Et slikt map har en iterator som gir ut nøklene i sortert rekkefølge. Et slikt map har også følgende operasjoner i tillegg til de et vanlig map har:

- **Next**(nøkkel) -> Nøkkel: Finner den neste nøkkelen i rekkefølgen hvis det er en. Denne metoden kan kalles med en nøkkel som ikke ligger i samlingen. Så Next(«B») kalt på et sorted map av personer med navn som nøkkel vil finne den første personen med navn som starter på B.
- **Previous**(Nøkkel) -> Nøkkel: Finner forrige nøkkel på samme måte som next
- **Between**(Nøkkel, Nøkkel) -> Liste av nøkler. Finner alle nøkler mellom de to oppgitte nøklene. I et sorted map av personer vil Between(«B», «C») finne navnene til alle personene med navn som starter på «B».
- **First**() -> Nøkkel: Finner første nøkkel
- **Last**() -> Nøkkel: Finner siste nøkkel
- **FindKth**(index) -> Nøkkel: Finner k-ende nøkkel, selection problemet

Abstrakt datatype: Set

Representerer en matematisk mengde. En mengde er en samling objekter som ikke kan inneholde duplikater og som støtter mengdeoperasjonene. Et Python set er et eksempel på en mengde.

- **Add**(Nøkkel): Legger inn en nøkkel hvis den ikke allerede er der
- **Delete**(Nøkkel): Sletter en nøkkel fra mengden
- **Get**(nøkkel) -> Nøkkel: Henter ut en nøkkel fra mengden.
- **Contains**(nøkkel): Sjekker om en nøkkel er med i mengden
- **Iterator**: Går gjennom mengden element for element
- **Union**(Mengde) -> Mengde: Lager en ny mengde som inneholder alle elementene som er med i minst én av denne mengden og mengden som er oppgitt som parameter
- **Intersection**(Mengde) -> Mengde: Lager en ny mengde som inneholder alle elementene som er med i både denne mengden og mengden som er oppgitt som parameter
- **Difference**(Mengde) -> Mengde: Lager en ny mengde som inneholder alle elementene som er med i denne mengden men ikke i den oppgitte mengden.
- **Subset**(Mengde) -> Boolean: Er denne mengden en delmengde av den oppgitte mengden?

Undertype: Sorted Set

Som for map kan man ha sorterte mengder. En sortert mengde støtter operasjonene fra sorterte map i tillegg til operasjonene for vanlige mengder.

Naiv Map implementasjon

Naiv Sorted Map implementasjon: array-liste som inneholder tupler med (nøkkel, verdi) par og som holdes i sortert rekkefølge på nøkkelen.

- **Get**(nøkkel) -> Verdi. Bruk binærsøk for å finne nøkkelen, eller stedet nøkkelen burde ha vært hvis den ikke er med. Kjøretid O(log(n)) for binærsøk

- **Put**(nøkkel, verdi). Bruk binærsøk for å finne nøkkelen eller innsettingspunktet. Overskriv hvis nøkkelen er der. Bruk array-liste insert hvis den ikke er der. Kjøretid O(log(n)) for å erstatte verdien til en nøkkel, kjøretid O(n) for å sette inn en ny nøkkel.
- **Delete**(nøkkel): Bruk binærsøk for å finne nøkkelen. Bruk array-liste delete for å fjerne den. Kjøretid O(n)
- **Contains**(nøkkel): Verdi. Bruk binærsøk for å finne nøkkelen, eller stedet nøkkelen burde ha vært hvis den ikke er med. Kjøretid O(log(n)) for binærsøk
- **Iterator**: Går gjennom lista element for element og returnerer elementene sortert på nøkkel. Kjøretid O(n) for å gå gjennom hele lista.
- **Next**(nøkkel) -> Nøkkel: Bruk binærsøk for å finne nøkkelen, og legg eventuelt til 1 for å få neste. Kjøretid O(log(n))
- **Previous**(Nøkkel) -> Nøkkel: Bruk binærsøk for å finne nøkkelen, og trekk eventuelt fra 1 for å få forrige. Kjøretid O(log(n))
- **Between**(Nøkkel, Nøkkel) -> Liste av nøkler. Bruk binærsøk for å finne første. Gå gjennom lista og returner alle nøkler etter den første som er mindre enn den siste.
- **First**() -> Nøkkel: Returner første element i lista, Kjøretid O(1)
- **Last**() -> Nøkkel: Returner siste element i lista, Kjøretid O(1)
- **FindKth**(index) -> Nøkkel: Returner elementet med indeks k i lista, Kjøretid O(1)

Naiv Sorted Set: Bruk en sortert map implemenasjon hvor du bruker nøkkel også som verdi.

Merk at en usortert liste er strengt dårligere enn en sortert liste for dette! Add blir O(n) siden du må leite etter duplikater. Resten av operasjonene blir O(n) eller O(n*k) for ulikhetssøk.

Merk: Lenket liste er dårligere enn array-liste for alle tilfeller her! Her er noen eksempel-kjøretider for ei lenket liste:

- Put: O(n) for å gå fram til innsettingspunktet
- Get: O(n) for å gå til der elementet er. Vinner ingenting på binærsøk i lenkete lister
- Delete: O(n) for å gå fram til elementet.

Hashtabeller

Dette er en enkel og mye brukt metode for å implementere et map eller en mengde.

- Den naive implementasjonen av et map tar O(log(n)) for å finne et element og O(n) for å sette inn et element.
- Kan gjøre det bedre

Jeg kommer til å presentere ideen bak hashtabeller i steg, det første steget er en tabell tilsvarende den tellesortering bruker for å sortere verdier:

Første forsøk: Direkte adresseringstabell

Hvis du har en nøkkel som kan anta n verdier, lag en tabell med n innslag. Du kan da bare bruke nøkkel som indeks i tabellen. En slik tabell støtter operasjonene:

- Get: Θ(1)
- Put: Θ(1)
- Delete: Θ(1)
- Contains: Θ(1)
- Iterator: Må gå gjennom hele tabellen. Θ(tabellstørrelse). Tabellstørrelse er her avhengig av størrelsen til verdiområdet, ikke antall elementer i tabellen!

Dette fungerer bare for nøkler som kan ha relativt få verdier. En nøkkel som er et 2-byte heltall (64000 verdier) går bra. En nøkkel som er en streng går ikke bra selv om en streng teoretisk sett kan konverteres til et tall.

Problem: Tabellen blir lett **veldig** stor. Studentnummer ved UiS er 6-sifrete tall så tabellen må ha 1.000.000 elementer. En streng kan ha mye flere verdier enn det igjen. Hvis antall reelle verdier er mye mindre enn antall mulige verdier kaster du bort en masse plass, samt at iteratoren vil kjøre veldig sakte siden å gå gjennom tabellen er Θ(tabellstørrelse) og ikke Θ(antall elementer).

En bedre idé: Hashing

Bruk en tabell med færre innslag og bruk en funksjon, kalt hashfunksjonen, for å regne ut hvor i tabellen du skal slå opp for å finne en gitt nøkkel.

Hashfunksjon: En funksjon som tar inn en verdi med et stort verdiområdet og returnerer et heltall i et mindre verdiområde. Verdien hashfunksjonen tar inn trenger ikke være et heltall, men kan være flyttall, strenger eller objekter.

Problem: Kollisjoner. Hvis du har to nøkler som er ulike, men hvor hashfunksjonen du bruker gir samme verdi ut. Da havner de to nøklene på samme sted i hashtabellen. Siden de ikke er duplikater (de er ulike) skal en lovlig Map implementasjon finne plass til begge. Dette er et problem som du ikke helt klarer å unngå, men det finnes flere teknikker for å redusere risikoen og håndtere de kollisjonene som fortsatt oppstår.

Løsning del 1: Lag en hashfunksjon som sprer dataene.

Hvis du har nøkler som naturlig klumper seg i en del av verdiområdet, vil du få mange kollisjoner i dette området. Derfor lager man hashfunksjoner som sprer dataene så jevnt utover i hashtabellen som mulig. Slike hashfunksjoner ødelegger ofte rekkefølgen i dataene, men dette er prisen man må betale for å redusere sjansen for kollisjoner.

Eksempler på dårlige hashfunksjoner

- For strenger: Legg sammen verdien til tegnene i strengen uten noe multiplikasjon eller annet. For navn og andre strenger som ofte er rundt samme lengre gir det dårlig spredning, verdiene klumper seg i et område. Noen eksempler:
    - Anders: 65 + 110 + 100 + 101 + 114 + 115 = 605
    - Birger: 66 + 105 + 114 + 103 + 101 + 114 = 603
    - Erlend: 69 + 114 + 108 + 101 + 110 + 100 = 602
    - Roger: 82 + 111 + 103 + 101 + 114 = 511
    - Hans: 72 + 97 + 110 + 115 = 394
    - Abraham: 65 + 98 + 114 + 97 + 104 + 97 + 109 = 684
    - Andre: Daniel 589, Merete 610, Nina 390, Anette 609, Elin 392, Anne 386, Olav 402, Kjersti 732, Trygve 641, Morten 629, Tomas 516, Terje 506
- For tall: Del tallet på en fast verdi. Problem: Nøkler som er tall vil oftere vært i starten enn i slutten av verdiområdet, så på de tidlige plassene i tabellen vil det bli mange kollisjoner.

Eksempler på gode hashfunksjoner

- Den enkleste: modulo: ta «verdi % tabellstørrelse» for heltall. Dette sprer verdiene greit i tabellen siden de minst signifikante sifrene ofte er mer tilfeldig fordelt enn de mest signifikante.

- Det fins mange mer avanserte hashfunksjoner som sprer tallene bedre, men modulo er en enkel og grei funksjon som viser prinsippene bak gode hashfunksjoner. Jeg vil derfor bruke modulo som hashfunksjon for heltall i dette faget.
- Python: Den innebygde funksjonen hash(objekt) lager en hashverdi som er et potensielt stort heltall fra en streng eller et hashbarrt objekt. Bruk modulo funksjonen for å redusere dette heltallet til et som passer hashtabellen.
- Hashing av strenger
    - Konverter første bokstav til et tall gjennom å bruke tallkoden for bokstaven i unicode, og lagre denne tallverdien i variabelen hash_verdi
    - For hver bokstav etter den første
        - gang hash_verdi med 31 eller 37 (ulike kilder bruker ulike tall her)
        - gjør nåværende bokstav til et tall og legg dette tallet til hash_verdi
        - sett hash_verdi lik hash_verdi % ønsket_verdiområde. Slik holder man tallene på brukbar størrelse.

Merk: Kryptografiske hashfunksjoner vs. hashtabell hashfunksjoner. Begge heter hashfunksjoner siden de tar inn verdier fra et stort verdiområde og returnerer verdier i et mindre verdiområde, samt at man bruker funksjoner hvor objekter som er nær hverandre i eget verdiområde likevel kan få svært ulike hashverdier. Kryptografiske hashfunksjoner og hashtabell-hashfunksjoner har imidlertid ulike krav. En hashtabell-hashfunksjon skal primært sett være rask å regne ut. En kryptografisk hashfunksjon kan godt være treg, men har strenge krav til at den ikke skal være reversibel (Gitt en hashverdi skal det være svært vanskelig å finne selv deler av den opprinnelige verdien). **I dette faget vil jeg kun snakke om hashtabell hashfunksjoner! Ikke bruk hashfunksjonene fra dette faget til datasikkerhet!**

Navnet hashfunksjon kommer av «hash», en britisk matrett som tilsvarer lapskaus, siden mange gode hashfunksjoner ødelegger ordnenen i dataene og plasserer objekter som er nært hverandre i virkelige verdier langt unna hverandre i hashtabellen.

Løsning del 2: Håndter de kollisjonene som likevel oppstår

Selv med en god hashfunksjon klarer du ikke alltid å unngå kollisjoner. Det er flere måter å håndtere kollisjoner. Åpen adressering er at man ved en kollisjon prøver andre celler til man finner en ledig en. Åpen adressering inkluderer lineær prøving, kvadratisk prøving, og multippel hashing. Alternativet til åpen adressering kan kalles lukket adressering eller lenket liste hashing siden man da har ei lenket liste for hver celle i hashtabellen siden man da kan lagre flere verdier i hver celle.

Fyllingsgrad (load factor): Tall fra 0.0 (tom) til 1.0 (full) som sier hvor mange av cellene som har en verdi. Kan regnes ut som fyllingsgrad = antall elementer delt på størrelsen til tabellen.

Lineær prøving:

Hvis plassen er opptatt, prøv neste plass og fortsett slik til man finner en ledig plass.

- Sannsynlighet for kollisjon: 1/(1-fyllingsgrad). P(cellen er ledig) = 1-fyllingsgrad
- Ettersom lista blir fylt opp, vil man måtte leite lengre etter ledig plass og det danner seg klynger (tilfeldige verdier vil naturlig klumpe seg og ikke være jevnt fordelt!). Formelen for antall forsøk blir $(1 + 1/(1\text{-fyllingsgrad})^2)/2$, som gir 50 forsøk for en fyllingsgrad på 0.9. Kalles «primary clustering».
    - Worst-case kjøretid for innsetting går fra $\Theta(1)$ til $O(n)$
    - Man må holde fyllingsgraden lav nok til at average case fortsatt holder seg på $\Theta(1)$. Ofte bruker 0.7 som en maksimumsverdi for fyllingsgrad for åpen adressering.

- Kostnad for å finne et element blir worst case lik kostnaden for en innsetting da man må leite helt til man finner en tom plass for å vite at elementet ikke er i lista.
    - Kostnad for mislykket søk = kostnad for innsetting
    - Kostnad for vellykket søk er ofte kortere da den ikke nødvendigvis må prøve så mange ganger. Kostnad er beregnet til (1 + 1/(1-fyllingsgrad))/2 tid
    - Ved fyllingsgrad=0.5:
        - Innsetting 2,5 forsøk
        - Mislykket søk 2,5 forsøk
        - Vellykket søk 1,5 forsøk
- Må implementere sletting annerledes. I stedet for å fysisk slette noe, marker det bare som slettet slik at finn algoritmen fortsatt leiter videre og ikke stopper for tidlig!

Kvadratisk prøving

- For å redusere clustering, bruker vi en litt annen sekvens: K, K + 1^2, K + 2^2, K + 3^2 osv, hvor du «går rundt» for store verdier. Kan beregnes uten å bruke pow-funksjonen med formelen: $K_i = K_{i-1} + 2*i + 1$
- Kan man treffe samme celle flere ganger under prøving? Ikke hvis tabellstørrelsen er et primtall og fyllingsgraden er 0.5 eller lavere. Bevis: Side 724 i min lærebok
- Sekundær clustering: Verdier som kolliderer vil prøve de samme stedene. Regnes som et marginalt problem.

Multippel hashing

Når man får en kollisjon, bruker man en annen hashfunksjon for å finn et godt sted. Man må da ha mange alternative hashfunksjoner. En variant bruker to hashfunksjoner, h_1 og h_2, og formelen $F_i = h_1(\text{nøkkel}) + i*h_2(\text{nøkkel})$, F_i representerer det i-ende forsøket, og i starter på 0. På denne måten påvirker nøkkelen ikke bare startpunktet med også lengden på hvert steg.

Lukket adressering / Liste med verdier for hver hashverdi

Engelsk separate chaining hashing – ofte er lenkete lister brukt her. Man har ei lenket liste for hver celle slik at man kan lagre flere verdier i hver celle

- Sikter ofte mot en fyllingsgrad på 1.0, og denne varianten er ikke begrenset av dette tallet og kan ha fyllingsgrader over 1
- Søketid blir nå tid for hashing + tid for å finne elementet i lista. Med en god hashfunksjon blir listene korte med lengde uavhengig av datamengden.
- Fordel: Mislykkete søk vil ofte støte på ei tom liste og kan returnere med en gang.
- Ofte brukt hvis innslagene i tabellen er store. Ved små innslag er prøving bedre.
- Ulempe: Dårligere cache utnyttelse
- Merk: Kjøretidene for denne varianten blir de samme som får åpen adressering, Θ(1) best og average case, O(n) worst case. Worst case er mange kollisjoner så noen få celler får lenkete lister med O(n) elementer, som man må leite gjennom for Contains, Delete og Get, og må se gjennom for å sjekke for duplikater for Put.

Rehashing

Hvis fyllingsgraden blir for stor må man lage en ny og større hashtabell på samme måte som for en Array-liste. For å kopiere over lista må den rehashes da hashfunksjonen vil være annerledes for den nye tabellen. Rehashing tar O(n) tid. Rehashing endrer rekkefølgen på innslagene!

Gjennom å minst doble størrelsen på hashtabellen for hver utvidelse kan man få O(1) amortized kjøretid for Put på samme måte som for array-lister.

Rehashing er grunnen til at hashtabeller ikke gir noen garantier om rekkefølgen ting kommer ut av iteratoren på.

For kvadratisk prøving må man bruke en metode for å finne neste primtall. Ignorer? Finne metode?

Krav til nøklene i et hashset og hashmap

Nøkkelen må være hashbare, det vil si at Python spesialmetoden __hash__() må returnere et heltall. Denne spesialmetoden kalles av Python sin hash() funksjon hvis den kjøres på objekter av egendefinerte klasser.

Standardimplementasjonen av __hash__ for egendefinerte Python objekter returnerer en hashverdi basert på minneadressen objektet ligger på, med mindre klassen overstyrer __eq__. Hvis klassen overstyrer __eq__ returnerer standardimplementasjonen av __hash__ None. For å lage et objekt som ikke er hashbart, overstyr __eq__ men ikke __hash__. Man kan bruke dette som markør for at objektet ikke bør brukes som nøkkel i en hashtabell siden alt i objektet kan endre seg.

Alle objekter som __eq__ regner som like skal også ha samme hashverdi. Man kan imidlertid ha objekter med samme hashverdi som er ulike ifølge __eq__. Poenget er at like objekter skal ha samme hashverdi slik at hashmap og hashset ser at de er duplikater.

Hvis det som brukes for å regne ut __hash__ endrer seg så ligger objektet plutselig på feil plass og hashtabellen kan ikke lengre finne objektet! For å brukes som en nøkkel i en hashtabell bør objektet derfor være slik at det som brukes for å regne ut __hash__ ikke kan endre seg.

En typisk måte å lage __hash__ på for egendefinerte klasser er å blande sammen hashkodene til de verdiene som brukes i __eq__ for å sammenlikne objektene av klassen.

HashSet som variant av HashMap

- Lagre bare nøkler i stedet for (nøkkel, verdi) par. Ellers likt
- Union: Lag et nytt HashSet med kapasitet lik summen av kapasitetene til de to eksisterende og legg inn verdiene fra begge. Kjøretid O(m + n) hvor m er antall elementer i det første HashSet-et og n er antall elementer i det andre HashSet-et.
- Snitt: Gå gjennom verdiene i det første HashSet-et og sjekk om verdien også er i det andre. Legg verdien inn i resultatet hvis den er med. Kjøretid O(n) hvor n er antall elementer i det første HashSet-et.
- Minus: Gå gjennom verdiene i det første HashSet-et og sjekk om verdien også er i det andre. Legg verdien inn i resultatet hvis den ikke er med. Kjøretid O(n) hvor n er antall elementer i det første HashSet-et.
- Delmengde: Gå gjennom verdiene i det første HashSet-et og sjekk om verdien også er i det andre. Ved første verdi som ikke er med, returner False. Hvis alle er med, returner True. Kjøretid O(n) hvor n er antall elementer i det første HashSet-et.