Hashing – Oversikt

Hashing er en teknikk som lar oss lagre og hente data effektivt i datastrukturer som hashmaps og hashsets. Hovedmålet er å kunne gjøre innsetting, oppslag og sletting av elementer veldig raskt, ofte i O(1) tid i gjennomsnitt. Bruk av arrayer • Underliggende datastruktur for en hash-tabell er en array. • Hver verdi vi ønsker å lagre må først omformes til en indeks i arrayet. Tre sentrale utfordringer i hashing

1. Fra verdi til indeks (hash-funksjon) ○ Vi trenger en funksjon som tar en vilkårlig verdi (for eksempel en streng eller et objekt) og returnerer et tall i arrayets intervall. ○ Funksjonen bør fordele verdier jevnt over arrayet for å unngå klynger.
2. Håndtering av kollisjoner ○ To ulike verdier kan få samme indeks. ○ Vanlige strategier: § Kjeding: Hver indeks peker til en liste av elementer som havner på samme plass. § Åpen adressering: Finn neste ledige plass i arrayet (lineær probing, kvadratisk probing, dobbel hashing).
3. Opprettholde ideell array-størrelse ○ For å unngå for mange kollisjoner må arrayet ha passende størrelse. ○ Når antall elementer blir for stort i forhold til arrayets størrelse (lastfaktor), resizer man tabellen og rehasher alle elementene. Mål • Effektiv innsetting, oppslag og sletting. • Typisk tidskompleksitet: O(1) i gjennomsnitt, men O(n) i verste fall hvis mange kollisjoner skjer. • Hashing er fundamentet bak moderne datastrukturer for ordbøker, sett og cache-mekanismer.

Map – abstrakt datatypen Et map er en datastruktur som kobler nøkler til verdier. • Hver nøkkel har nøyaktig én tilknyttet verdi. • Eksempel: Nøkkel: ♣ ♢ ♠ ♡ Verdi: 0 1 2 3 • Typiske operasjoner for maps: 1. Sett inn: M.put(key, value) – kobler nøkkelen til verdien. 2. Slå opp: M.get(key) – henter verdien knyttet til nøkkelen. 3. Slett: M.remove(key) – fjerner nøkkelen og verdien fra mappen. ○ Dersom nøkkelen ikke finnes, returneres som regel null eller en indikasjon på at nøkkelen ikke eksisterer. Poeng: maps gir rask tilgang til verdier via nøklene, uavhengig av størrelsen på datasettet.

Hashmap – konkret implementasjon av Map Et hashmap er en vanlig måte å implementere et map på. • Vi bruker et array A og en hashfunksjon h. • Hashfunksjonen tar en nøkkel k og konverterer den til et indeksnummer i i arrayet: $i = h(k)$, der $0 \le i < |A|$ • Prosessen kalles å hashe nøkkelen. Kollisjoner • Antall mulige nøkler er ofte mye større enn antall plasser i arrayet. ○ Eksempel: uendelig mange strenger, men arrayet har begrenset størrelse $|A|$. • Kollisjon: To ulike nøkler blir hashet til samme indeks i. • Håndtering av kollisjoner er nødvendig for å sikre korrekt lagring og oppslag. Strategier for kollisjoner

1. Kjeding (chaining): ○ Hver array-indeks peker til en liste (eller lenket liste) av nøkkel-verdi-par. ○ Nye elementer som kolliderer legges i listen.
2. Åpen adressering (open addressing): ○ Hvis plassen er opptatt, søker man etter neste ledige plass i arrayet. ○ Metoder: lineær probing, kvadratisk probing, dobbel hashing. Mål: • Opprettholde rask innsetting, oppslag og sletting, vanligvis O(1) i gjennomsnitt. • Håndtere kollisjoner effektivt slik at tabellen ikke degraderer til O(n) ved for mange kollisjoner.

Distribusjon: strenger og hashfunksjoner Når vi lager et hashmap med strenger som nøkler, må vi konvertere hver streng til et tall som brukes som en indeks i arrayet. Dette gjøres med en hashfunksjon. Dårlig hashfunksjon En enkel idé: summer tallverdiene til bokstavene i strengen. Algoritme (dårlig): Input: streng s, arraystørrelse N Output: heltall h der $0 \le h < N$ Procedure HashStringBad(s, N) $h \leftarrow 0$ for hver bokstav c i s do $h \leftarrow h$ + charToInt(c) return h mod N • charToInt(c) konverterer en bokstav til et tall, f.eks. ASCII-verdi. • Vi summerer alle tallene og tar modulo N for å få en indeks i arrayet. Problemer: • Summen

er kommutativ: a + b = b + a. ○ F.eks. "ab" og "ba" gir samme hash. • Dette fører til kollisjoner for ord med samme bokstavsett. • Distribusjonen blir dårlig for store datasett.

Bedre hashfunksjon Vi kan forbedre distribusjonen ved å introdusere vekt og rekkefølge i bokstavene. Algoritme (god): Input: streng s, arraystørrelse N Output: heltall h der 0 ≤ h < N Procedure HashString(s, N) h ← 0 for hver bokstav c i s do h ← 31·h + charToInt(c) return h mod N • Her multipliseres den tidligere summen med en konstant (31) før vi legger til neste bokstav. • Dette skaper en mer kompleks sammenheng mellom bokstavene i strengen. • Små endringer i strengen gir helt forskjellige hasher, som reduserer kollisjoner. Kommentar: • Dette er samme hashfunksjon som brukes i Java for strenger. • Konstanten 31 er valgt fordi den gir god spredning og effektiv beregning (kan implementeres som (h << 5) - h).

Poeng å huske:

1. En hashfunksjon må være konsekvent: samme input gir alltid samme hash.
2. Den bør fordele nøklene jevnt over arrayet.
3. Den bør minimere kollisjoner.
4. Dårlige funksjoner (som enkel summering) kan fungere på små datasett, men feiler på større eller mønstrede data.

Distribusjon: HashStringBad vs HashString Når vi hasher mange strenger, f.eks. alle ordene i en ordbok (/usr/share/dict/words med 235 886 ord), ser vi tydelig forskjellen mellom en dårlig og en god hashfunksjon.

Eksperiment med HashStringBad • Vi setter arraystørrelse N = 235 886 (samme som antall ord). • Eksempel: "algorithm" hasher til 967. • Antall kollisjoner: 577 andre ord hasher til samme indeks! • Dette viser at enkel summering av bokstavverdier gir veldig mange kollisjoner. • Problemene med HashStringBad blir tydeligere med små N eller ord med like bokstaver i ulik rekkefølge.

Eksperiment med HashString • Samme arraystørrelse N = 235 886. • Eksempel: "algorithm" hasher til 184 369. • Antall kollisjoner: kun 1 annet ord! • Den gode hashfunksjonen sprer nøklene mye jevnere over arrayet. Observasjon: • Selv med stor ordbok og store N, får vi mye færre kollisjoner. • Dette gjør innsetting, oppslag og sletting i hashmappen betydelig mer effektivt.

Visualisering av fordeling • Vi kan visualisere hvordan nøklene fordeles over arrayet: ○ Hver prikk representerer et array-element 0...N-1. ○ Hver gang et ord hasher til en indeks i, øker fargen på prikken i. • For HashStringBad ser vi mange prikker med mørk farge (mange kollisjoner). • For HashString blir prikkene jevnt fordelt, nesten ingen mørke områder. Poeng å huske:

1. God hashfunksjon = jevn fordeling og få kollisjoner.
2. Dårlig hashfunksjon = konsentrasjon av verdier, mange kollisjoner.
3. Selv små endringer i strengen påvirker hashen i god funksjon (diffusjon).
4. Kollisjoner må håndteres med åpen adressering eller lenking, men færre kollisjoner = bedre ytelse.

Kollisjonshåndtering i hashmaps Når vi hasher nøkler til et array med begrenset størrelse, vil kollisjoner oppstå – altså at to ulike nøkler hasher til samme indeks. For å håndtere dette finnes flere strategier:

1. Separate Chaining • Hver plass i arrayet peker til en bøtte. • Bøtten kan være: ○ en lenket liste (vanlig), eller ○ et annet datastruktur, f.eks. et binært søketre. • Innsetting (insert): 1. Gitt nøkkel k som hasher til indeks i, og verdi v. 2. La B ← A[i] være bøtten. 3. Hvis B er tom: opprett ny liste og

legg inn (k, v). 4. Hvis B ikke er tom: § Søk gjennom listen for nøkkel k. § Hvis k finnes: erstatt gammel verdi med v. § Hvis k ikke finnes: legg (k, v) på slutten av listen. • Oppslag (lookup): 1. Gitt nøkkel k som hasher til i. 2. La B ← A[i]. 3. Hvis B er tom, returner null. 4. Ellers, søk i listen etter noden med nøkkel k. • Sletting (delete): 1. Gitt nøkkel k som hasher til i. 2. La B ← A[i]. 3. Hvis B er tom, gjør ingenting. 4. Ellers, fjern noden med nøkkel k fra listen. Fordeler: • Enkelt å implementere. • Fungerer godt når load factor (antall elementer / arraystørrelse) er moderat. Ulemper: • Krever ekstra minne til bøtter. • Oppslag i en lang liste kan bli tregt hvis mange kollisjoner.

2. Linear Probing (åpen adressering) • Vi bruker kun arrayet – ingen ekstra lenkelister. • Ved kollisjon: ○ Vi ser etter neste ledige plass i arrayet (lineært). ○ Fortsett inntil vi finner en tom plass. • Oppslag: ○ Start på hashet indeks. ○ Sjekk hver påfølgende plass til nøkkel k finnes eller en tom plass oppdages. • Sletting: ○ Må håndtere "tombstones" eller merke slettede plasser, ellers kan søk stoppes for tidlig. Fordeler: • Krever mindre minne enn separate chaining. • Enkel å implementere i små, statiske arrays. Ulemper: • Kan gi clustering: mange kollisjoner i samme område, som reduserer effektivitet. • Sletting er mer komplisert enn ved separate chaining.

Merk: • Vi antar at arrayet A har størrelse N og at det alltid inneholder færre enn N elementer, slik at vi unngår at arrayet blir fullt. • Load factor bør være < 1 (for linear probing) eller < 0.75 (for separate chaining) for å opprettholde god ytelse.

Kollisjonshåndtering – Linear Probing Linear probing er en form for åpen adressering i hashing. Her brukes kun arrayet som underliggende datastruktur, og kollisjoner løses ved å søke etter neste ledige plass i arrayet.

Innsetting (Insert) Gitt en nøkkel k som hasher til indeks i, og en verdi v:

1. Hvis A[i] er tom (null), sett (k, v) på plass i og returner.
2. Hvis nøkkelen på A[i] er lik k, oppdater verdien med v og returner.
3. Hvis A[i] er opptatt med en annen nøkkel, gå til neste plass: $i \leftarrow (i+1) \bmod N$ og gjenta prosessen til du finner en ledig plass eller nøkkelen k. Merk: Bruken av modulo N gjør at vi kan «wrap around» arrayet.

Oppslag (Lookup) Gitt en nøkkel k som hasher til i:

1. Hvis A[i] er tom (null), returner null (nøkkelen finnes ikke).
2. Hvis nøkkelen på A[i] er k, returner verdien.
3. Ellers, gå til neste plass: $i \leftarrow (i+1) \bmod N$ og gjenta prosessen til nøkkelen finnes eller en tom plass oppdages.

Sletting (Delete) Gitt en nøkkel k som hasher til i:

1. Hvis A[i] er tom, returner – nøkkelen finnes ikke.
2. Hvis nøkkelen på A[i] er ulik k, gå videre til neste plass (i ← i + 1 mod N) og gjenta.
3. Hvis nøkkelen på A[i] er lik k, fjern elementet (A[i] ← null).
4. Tett hullet for å unngå at søk som skulle ha funnet elementer stopper for tidlig.

Tetting av hull (Hole Filling) Etter fjerning må vi passe på å ikke bryte søkelogikken: • Linear probing-algoritmer terminerer når de treffer en tom plass. • Hvis vi bare setter A[i] ← null, kan elementer som har kollidert tidligere bli utilgjengelige. To strategier for å håndtere dette:

1. Markér som slettet (tombstone) ○ Feltet markeres som slettet, men betraktes fortsatt som «opptatt» ved søk. ○ Ved innsetting kan markerte felter brukes på nytt. ○ Tombstones forsvinner når arrayet rehashes.
2. Flytt elementer for å tette hullet ○ Etter sletting på plass i, sjekk påfølgende plasser. ○ Hvis en nøkkel har en hash som peker til i eller tidligere, flytt den til tomme plass i. ○ Gjenta prosessen for det nye hullet, til alle elementer er korrekt plassert. Fordeler: • Krever ikke ekstra datastruktur som separate chaining. • Kan være effektiv når arrayet ikke er tettpakket. Ulemper: • Kan føre til clustering – mange elementer samler seg i samme område, og søk blir tregere. • Sletting og rehashing er mer komplisert enn i separate chaining.

Effektivitet – Load Factor Load factor er en nøkkelindikator for hvor effektivt et hashmap fungerer. Den sier noe om hvor tett elementene ligger i arrayet. • Definisjon: Load factor $\alpha = \frac{n}{N}$ hvor: ○ n = antall elementer i hashmappen ○ N = størrelse på arrayet • Hva skjer hvis load factor er for høy? ○ Separate chaining: § Flere elementer havner i samme bøtte, som blir en lengre lenket liste. § Oppslag, innsetting og sletting blir lineær tid i antall elementer i bøtten. ○ Linear probing: § Elementene samler seg i lange segmenter uten ledige plasser. § Søkeoperasjoner må gå gjennom mange plasser før riktig element finnes, også lineær tid i arrayets tetthet. • Hva skjer hvis arrayet er for stort? ○ Sløsing med plass. Store deler av arrayet står tomme. ○ Mindre effektiv bruk av minne, selv om operasjonene fortsatt er raske. • Ideell load factor: ○ Eksperimentelt bestemt. ○ Ofte anbefalt mellom 1/2 og 3/4 (0,5–0,75). ○ Det betyr at arrayet bør være litt mer enn halvfullt for best balanse mellom minnebruk og hastighet.

Effektivitet – Rehashing Rehashing brukes for å opprettholde effektiviteten når load factor blir for høy eller for lav:

1. Når: ○ Hvis arrayet blir «for fullt» (høy load factor). ○ Mindre vanlig: Hvis arrayet blir for tomt, kan man krympe arrayet.
2. Hvordan: ○ Lag et nytt, større array (typisk 2× så stort). ○ Beregn hashfunksjonen på nytt for alle eksisterende elementer og sett dem inn i det nye arrayet.
3. Hvorfor: ○ Reduserer lengden på lenkede lister (separate chaining) eller segmenter (linear probing). ○ Holder innsetting, oppslag og sletting nær konstant tid i gjennomsnitt.
4. Merk: ○ Rehashing kan være tidkrevende fordi alle elementene må flyttes. ○ I praksis gjøres dette sjeldent, og ofte velger man en stor nok startstørrelse og øker bare ved behov.

Kort oppsummert: • Hashmaps fungerer best når arrayet ikke er for tettpakket og ikke for tomt. • Load factor gir en enkel måte å måle tettheten. • Rehashing hjelper med å opprettholde effektivitet over tid.

En uformell kjøretidsanalyse Når vi analyserer kjøretiden til algoritmer, ser vi ofte på verste tilfelle: • For et hashmap: ○ I verste tilfelle kan alle nøkler hashe til samme posisjon. ○ Separate chaining oppfører seg som en enkel lenket liste. § Oppslag, innsetting og sletting blir $O(n)$ i antall elementer i bøtten. ○ Linear probing oppfører seg som et uordnet array. § Vi må søke gjennom mange plasser før vi finner elementet, også $O(n)$. • Konklusjon: ○ En tradisjonell O-analyse i verste tilfelle gir et dårlig bilde av hvor effektivt hashmaps fungerer i praksis. ○ Til tross for dette er hashmaps svært raske i gjennomsnitt. • Løsning: ○ Vi bruker forventet amortisert kjøretidsanalyse for å forstå hashmaps' effektivitet. ○ Dette innebærer å vurdere gjennomsnittlig kostnad per operasjon over en sekvens av operasjoner. ○ Merk: Dette er ikke pensum for IN2010, men du bør vite hva det betyr i konteksten av hashing: § Vi ser på hvor raskt hashmaps i gjennomsnitt fungerer, ikke bare i verste tilfelle.

En uformell (forventet) kjøretidsanalyse For å forstå hvorfor hashmaps er effektive i praksis, antar vi at vi har en god hashfunksjon: • Den gir en uniformt tilfeldig fordeling av nøkler over arrayet. •

Sannsynligheten for at en nøkkel k hasher til en gitt posisjon i er: P(k→i)=1N hvor N er størrelsen på arrayet. • For flere nøkler k1,k2,...,km som alle hasher til samme posisjon i, blir sannsynligheten: P(k1,k2,...,km→i)=(1N)m ○ Dette tallet blir forvinnende lite når m vokser. • Konsekvens: ○ Kollisjoner er sjeldne så lenge arrayet har god plass. ○ Selv når arrayet fylles opp, er sannsynligheten for at mange elementer havner på samme posisjon fortsatt lav. • Forventet kjøretid for operasjoner: ○ Oppslag og sletting: forventet O(1) ○ Innsetting: forventet amortisert O(1) § «Amortisert» betyr at enkelte innsettinger kan ta lengre tid (f.eks. ved rehashing), men gjennomsnittet over mange operasjoner er konstant.

Kort oppsummert: • Hashmaps ser lineære ut i verste tilfelle, men er svært raske i praksis. • God hashfunksjon og passende load factor gjør kollisjoner sjeldne. • Vi kan derfor behandle innsetting, oppslag og sletting som nesten konstant tid, selv for store datasett.

En uformell (amortisert) kjøretidsanalyse Når vi bruker hashmaps, vil arrayet til slutt bli fullt, eller load factor blir for høy: • Rehashing: ○ Vi lager et større array og setter inn alle elementene på nytt. ○ Denne operasjonen tar lineær tid, altså O(n), hvor n er antall elementer. ○ Den innsettingen som forårsaker rehashen er dermed et verstefall, men skjer sjelden. Amortisert perspektiv: • I amortisert analyse ser vi ikke på enkeltoperasjonen isolert, men på alle innsettingene som ledet opp til rehashen samlet. • Anta: 1. Vi gjør n innsettinger før rehashen. 2. Hver innsetting før rehashen tar O(1) steg. 3. Rehashen tar c·n steg, der c er en konstant. • Vi kan nå fordele kostnaden av rehashen over alle n innsettinger: ○ Hver innsetting «tar på seg» c ekstra steg. ○ Den totale tiden per innsetting blir dermed O(1 + c) = O(1). • Konklusjon: ○ Selv om en innsetting kan være treg av og til (ved rehashing), bruker hver innsetting i gjennomsnitt konstant tid. ○ Dette kalles forventet amortisert O(1). • Merk: ○ Innsetting kan være treg sjelden, noe som kan være relevant i sanntidsapplikasjoner, men for de fleste praktiske formål er hashmaps ekstremt raske.

Effektivitet oppsummert • Verste tilfelle: alle operasjoner O(n) ○ Teoretisk når alle nøkler havner på samme plass, eller ved rehash. • Forventet amortisert: O(1) per operasjon ○ Krever god hashfunksjon og moderat load factor. • Rehashing: tar O(n), men skjer sjelden og fordeles over mange innsettinger. • Praktisk resultat: hashmaps er svært raske, effektive og anvendelige i de fleste programmeringssammenhenger.