Prioritetskøer og heaps

Oversikt • Vi fokuserer på prioritetskøer, som er en datastruktur som alltid gir rask tilgang til elementet med høyest (eller lavest) prioritet. • Vi lærer også om binære heaps, som er en effektiv måte å implementere prioritetskøer på. • Før vi går inn i heaps, repeterer vi O-notasjon for å kunne analysere kjøretid.

Prioritetskø – definisjon • En prioritetskø støtter følgende operasjoner: 1. Insert(x) – legg til et element x med en bestemt prioritet 2. FindMax / FindMin – finn elementet med høyest eller lavest prioritet 3. ExtractMax / ExtractMin – fjern elementet med høyest eller lavest prioritet • Eksempel: ○ Task scheduler: alltid kjør jobben med høyest prioritet først ○ Hendelsesstyring i simuleringer: prosesser hendelser i tidsrekkefølge

Binære heaps • Et binært heap er et fullstendig binærtre som tilfredsstiller heap-egenskapen: ○ Max-heap: hver node har verdi ≥ verdiene til sine barn ○ Min-heap: hver node har verdi ≤ verdiene til sine barn • Fullstendig tre: alle nivåer er fylt, bortsett fra det siste, som fylles fra venstre til høyre. • Representasjon: ○ Kan implementeres som et array ○ For node i med indeks i: § Venstre barn: 2i + 1 § Høyre barn: 2i + 2 § Forelder: (i-1)//2

Hvorfor heaps? • Binære heaps gir effektiv implementasjon av prioritetskøer: ○ Insert: O(logn) ○ ExtractMax / ExtractMin: O(logn) ○ FindMax / FindMin: O(1) • De er spesielt effektive fordi de holder treet balansert automatisk ved hjelp av "heapify"-operasjoner.

O-notasjon – handler om vekst Hva O-notasjon prøver å fange • En algoritme tar input av størrelse n og produserer output basert på dette. • Spørsmålet vi ønsker å svare på: ○ Hva er et øvre estimat på hvor lang tid algoritmen bruker? • O-notasjon lar oss abstrahere fra maskinspesifikke detaljer og konstantfaktorer: ○ Konstanten kan inkludere faktorer som CPU-hastighet, kompilatoroptimalisering, eller antall instruksjoner per steg. • Fokus er på veksten av kjøretiden når input blir stor, ikke eksakte tider.

Formell definisjon • La f(n) være kjøretiden til algoritmen for input av størrelse n. • La g(n) være en funksjon som representerer en øvre grense. • Vi sier at: $f(n) \in O(g(n))$hvis det finnes en konstant $c>0$ og et n0, slik at $f(n) \leq c \cdot g(n)$ for alle $n \geq n0$ • Dette betyr: ○ f(n) vokser ikke raskere enn g(n) for store n ○ g(n) fungerer som en øvre skranke for algoritmen

Visuelt eksempel • Anta $f(n)=3n2+5n+2$ • Vi kan finne en konstant c slik at: $f(n) \leq 4 \cdot n2$for alle $n>6$ • Dermed: $f(n) \in O(n2)$ • Dette viser at det ledende leddet dominerer veksten for store input.

Klassiske algoritmeeksempler og deres kjøretidskompleksitet

1. Konstant tid – O(1) 1 Procedure Constant(n) 2 return n * 3 • Utfører alltid én operasjon uavhengig av størrelsen på input n. • Eksempel: Tilordning, enkel aritmetisk operasjon. • Kjøretid: konstant, uavhengig av n.

2. Logaritmisk tid – O(log n) 1 Procedure Log(n) 2 i ← n 3 while i > 0 do 4 Constant(i) 5 i ← ⌊i / 2⌋ • Variabelen halveres hver iterasjon. • Antall steg: log2n iterasjoner. • Typisk for binærsøk, trær, del-og-hersk-algoritmer.

3. Lineær tid – O(n) 1 Procedure Linear(n) 2 for i ← 0 to n-1 do 3 Constant(i) • Algoritmen gjør ett steg per element i input. • Kjøretid vokser proporsjonalt med n. • Eksempel: Sekvensielt søk i array.

4. Lineær-logaritmisk tid – O(n·log n) 1 Procedure Linearithmic(n) 2 for i ← 0 to n-1 do 3 Log(n) • Ytre løkke går n ganger, indre løkke logaritmisk. • Typisk for effektive sorteringsalgoritmer: mergesort, heapsort.

5. Kvadratisk tid – O(n²) 1 Procedure Quadratic(n) 2 for i ← 0 to n-1 do 3 for j ← 0 to n-1 do 4 Constant(i) • Nestede løkker over alle elementer. • Antall steg: n⋅n=n2. • Eksempel: Boblesortering, enkel sammenligning av par.

6. Polynomiell tid – O(n^k) 1 Procedure Polynomial(n) 2 for i1 ← 0 to n-1 do 3 for i2 ← 0 to n-1 do ... 5 for ik ← 0 to n-1 do 6 Constant(i) • Generelt mønster for flere nestede løkker. • Steg vokser som nk. • Kan bli ineffektivt raskt når k øker.

7. Eksponentiell tid – O(2^n) 1 Procedure Exponential(n) 2 if n = 0 then 3 return 1 4 a ← Exponential(n-1) 5 b ← Exponential(n-1) 6 return a + b • Funksjonen kaller seg selv to ganger per nivå. • Antall steg dobles hver gang n øker. • Typisk for naive løsninger på rekursive problemer: Fibonacci uten memoization.

Hvorfor O-notasjon gjør analyse enklere

1. Fokus på veksttrenden • O-notasjon lar oss se hvordan algoritmens kjøretid vokser når input blir stor. • Vi trenger ikke telle hver eneste operasjon, men kan se på hovedtrenden. • Dette gjør det mulig å sammenligne algoritmer på et mer abstrakt nivå.

2. Konstantfaktorer ignoreres • Et uttrykk som 5n+3 skrives enkelt som O(n). • Konstantene 5 og 3 påvirker kjøretiden litt, men ikke vekstmønsteret når n blir stort. • Dette betyr at maskinhastighet, små optimaliseringer og implementasjonsdetaljer ikke endrer O-notasjonen.

3. Kun det største leddet teller • Når vi har flere ledd, tar vi kun hensyn til det som vokser raskest med n. • Eksempel: 3n2+7n+2 → O(n2) • De mindre leddene blir ubetydelige når n blir stort.

4. Vanlige vekstklasser I IN2010 fokuserer vi på følgende typer kompleksitet: Notasjon Veksttype Eksempel på algoritme O(1) Konstant tid Tilordning, enkel beregning O(log n) Logaritmisk tid Binærsøk, balanserte trær O(n) Lineær tid Sekvensielt søk O(n log n) Lineæritmisk tid Mergesort, heapsort O(n²) Kvadratisk tid Boblesortering O(n^k) Polynomiell tid Flere nestede løkker O(2^n) Eksponensiell tid Rekursive løsninger uten memoization

5. Flere input • Hvis algoritmen avhenger av mer enn ett input, må vi vurdere samspillet mellom alle inputene. • O-notasjon gjør det fortsatt mulig å se hovedtrenden, uten å måtte lage kompliserte utregninger for hver kombinasjon.

6. Oppsummert • O-notasjon forenkler analysen av algoritmer dramatisk. • Den fokuserer på det som virkelig betyr noe: hvordan kjøretiden vokser når problemstørrelsen øker. • Dette gir oss en robust metode for å sammenligne effektiviteten til algoritmer uavhengig av maskin eller implementasjon.

Prioritetskøer

1. Definisjon • En prioritetskø er en datastruktur som lagrer en samling elementer, hvor hvert element har en “prioritet”. • Den støtter typisk to hovedoperasjoner: 1. insert(e) – legger til et element i

køen. 2. removeMin() – fjerner og returnerer elementet med minste verdi (høyest prioritet). • I praktisk programmering brukes ofte navnene push(e) og pop().

2. Bruk • Prioritetskøer brukes når man ønsker å behandle elementer i rekkefølge etter prioritet, ikke nødvendigvis i rekkefølge de ble lagt inn. • Eksempler: ○ Oppgaveplanlegging (scheduler i OS) ○ Dijkstra's algoritme for korteste vei ○ Event-håndtering i simuleringer

3. Mulige underliggende datastrukturer Datastruktur insert(e) removeMin() Kommentar Usortert lenket liste O(1) O(n) Lett å sette inn, men man må lete gjennom hele lista for minste element. Sortert lenket liste O(n) O(1) Minste element ligger alltid først, men innsetting krever at man finner riktig posisjon. Balansert binært søketre O(log n) O(log n) Effektiv både for innsetting og fjerning; minste element finnes lengst til venstre. Heap (binær heap) O(log n) O(log n) Vi lærer mer om dette; optimal for prioritetskø-operasjoner.

4. Viktig for implementasjon • For alle implementasjoner må vi kunne sammenligne elementene, slik at vi kan finne hvilket element som har høyest eller lavest prioritet. • I noen tilfeller kan vi ha maks-heap (største element først) i stedet for min-heap.

5. Intuisjon • Valg av datastruktur avhenger av hvilken operasjon som skal være raskest: ○ Mange innsettinger og få fjerninger → usortert liste kan være best. ○ Mange fjerninger → sortert liste, søketre eller heap er mer effektivt.

Totale ordninger

1. Intuisjon • Mange av de tingene vi sorterer daglig, følger en total ordning. • En total ordning sier deg hvordan du kan sammenligne og rangere elementer slik at du alltid kan avgjøre hva som kommer først og sist. • Eksempel: Sortering av personer etter alder. ○ Alder representeres vanligvis med naturlige tall. ○ Tallene kan sammenlignes med ≤. ○ Dette gir en naturlig total ordning på mengden av personer.

2. Formell definisjon • La mengden A være en samling objekter vi ønsker å ordne. • En total ordning er en binær relasjon ≤ på A som oppfyller fire krav for alle x,y,z∈A: 1. Refleksivitet: x\≤x § Et element er alltid likt seg selv. 2. Antisymmetri: Hvis x\≤y og y\≤x, så er x=y § To elementer kan ikke "overstyre" hverandre; da må de være like. 3. Transitivitet: Hvis x\≤y og y\≤z, så er x\≤z § Relasjonen må være konsistent når vi sammenligner flere elementer. 4. Totalitet: Enten x\≤y eller y\≤x § Alle elementer kan sammenlignes med hverandre.

3. Totale ordninger i programmering • Java: ○ En klasse som implementerer Comparable har en total ordning over objektene. ○ Forutsatt at implementasjonen følger refleksivitet, antisymmetri, transitivitet og totalitet. • Python: ○ En klasse som implementerer **lt** (less than) kan brukes til total ordning. ○ Samme forutsetninger gjelder: relasjonen må ikke bryte de fire kravene.

4. Praktisk betydning • Å ha en total ordning er viktig for: ○ Sorteringsalgoritmer (eks. mergesort, quicksort) ○ Prioritetskøer og heaps ○ Binære søketrær (inkludert AVL-trær) • Uten total ordning kan man ikke garantere at algoritmer fungerer korrekt.

Binære heaps

1. Definisjon og egenskaper En binær heap er en spesialisert binærtrestruktur som brukes til effektiv prioritering. Den oppfyller to hovedregler: 1. Heap-egenskap (min-heap): ○ Hver node v som ikke

er roten, har en verdi større enn foreldrenoden. ○ Dette betyr at roten alltid inneholder det minste elementet. ○ (For max-heap er regelen motsatt: foreldrenoden er større enn barna.) 2. Komplett binærtre-egenskap: ○ Treets nivåer fylles helt opp fra venstre mot høyre. ○ Hvis treet har høyde h: § Nivå i (0 ≤ i < h) har 2i noder. § Noder på siste nivå h er plassert så langt til venstre som mulig.

2. Binære heaps vs. balanserte søketrær • Kompleksitet: ○ Innsetting og fjerning av minste element har O(log n), likt som i AVL- og rød-svarte trær. • Fordeler med heaps: ○ Støtter færre operasjoner, men er enklere. ○ Ingen rotasjoner kreves for å opprettholde balanse. ○ Alltid komplette og dermed svært balanserte. ○ Kan implementeres effektivt med arrayer, som gir enkel tilgang til forelder og barn: § Venstre barn: 2i+1 § Høyre barn: 2i+2 § Forelder: ⌊(i−1)/2⌋

3. Eksempel på binær heap Trestruktur: 0 /
   1 7 / \ /
   2 4 11 10 / \ / \ / \ /
   5 10 6 9 29 14 28 21 /
   19 19 15 22 23 • Heap-egenskap: ○ Hver node er større enn sin forelder. ○ Roten (0) er det minste elementet. • Komplett tre: ○ Nivåene fylles helt fra venstre mot høyre.

4. Array-representasjon • Et komplett binærtre kan lagres i et array uten pekere: Indeks 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 Verdi 0 1 7 2 4 11 10 5 10 6 9 29 14 28 21 19 19 15 22 23 • Tilgangsregler: ○ For node i: § Venstre barn = 2i+1 § Høyre barn = 2i+2 § Forelder = ⌊(i−1)/2⌋

Binære heaps – idé bak innsetting Når vi setter inn et nytt element i en min-heap, følger vi en enkel og effektiv strategi: 1. Plassering i treet ○ Legg det nye elementet på neste ledige plass i treet, altså der treet fortsatt vil være komplett. ○ Dette tilsvarer å sette elementet på slutten av arrayet i arrayrepresentasjonen. 2. Opprettholde heap-egenskapen ○ Sjekk om det nye elementet er mindre enn foreldrenoden. ○ Hvis ja, bytt plass med foreldrenoden. ○ Fortsett rekursivt (eller iterativt) oppover til enten roten nås eller heap-egenskapen er tilfredsstilt. Hovedidé: Vi bobler opp elementet til det er på riktig plass.

Binære heaps – idé bak sletting (removeMin) Når vi skal fjerne roten (minste element) fra en min-heap: 1. Bytt med siste node ○ Bytt verdien i roten med verdien i siste noden i treet. ○ Dette gjør at treet fortsatt er komplett når vi fjerner siste node. 2. Opprettholde heap-egenskapen ○ Sjekk om roten er større enn noen av barna. ○ Hvis ja, bytt plass med det minste barnet. ○ Fortsett rekursivt (eller iterativt) nedover til heap-egenskapen er tilfredsstilt. Hovedidé: Vi bobler ned roten til riktig posisjon.

Binære heaps – Tre- vs. arrayimplementasjon • Tre-implementasjon: ○ Hver node trenger: § Elementet § Venstre og høyre barn § Foreldrepointer ○ Må holde styr på siste node for effektiv innsetting/sletting. ○ Kan bli klønete og kreve ekstra pekere. • Array-implementasjon (vanlig): ○ Fullt effektivt fordi treet er komplett. ○ Roten ligger på A[0], og barna ligger på kjente indekser: § Venstre barn = 2i+1 § Høyre barn = 2i+2 § Forelder = ⌊(i−1)/2⌋ ○ Størrelsen på arrayet holder oversikt over siste node. ○ Innsetting/sletting krever O(log n) tid uten behov for pekere.

Binære heaps – Arrayrepresentasjon 1. Innsetting: ○ Legg elementet på A[n], der n er antall elementer i heapen. ○ Boble opp hvis nødvendig for å opprettholde heap-egenskapen. 2. Sletting av minste element: ○ Flytt A[n-1] til roten (A[0]). ○ Boble ned til korrekt posisjon. Fordeler med array: • Ingen ekstra pekere. • Enkel beregning av foreldre/barn med indeksformler. • Effektiv bruk av minne og CPU.

Binære heaps – Hjelpeprosedyrer For å jobbe effektivt med binære heaps implementert som array, trenger vi noen enkle hjelpeprosedyrer som gjør det lett å navigere i treet.

1. Foreldrenoden til et element • La A[i] være elementet på indeks i i arrayet. • Foreldrenoden til A[i] ligger alltid på indeks: ParentOf(i)=⌊(i−1)/2⌋ • Dette gjelder for alle noder unntatt roten (som ikke har forelder). • Eksempel: ○ Hvis i=5, så er foreldrenoden ⌊(5−1)/2⌋=⌊4/2⌋=2. Algoritme: Procedure ParentOf(i) return ⌊(i - 1) / 2⌋

2. Venstre barn til et element • Venstre barn til A[i] ligger alltid på indeks: LeftOf(i)=2⋅i+1 • Eksempel: ○ Hvis i=2, så er venstre barn på 2⋅2+1=5. Algoritme: Procedure LeftOf(i) return 2 * i + 1

3. Høyre barn til et element • Høyre barn til A[i] ligger alltid på indeks: RightOf(i)=2⋅i+2 • Eksempel: ○ Hvis i=2, så er høyre barn på 2⋅2+2=6. Algoritme: Procedure RightOf(i) return 2 * i + 2

Oppsummering • Disse tre hjelpeprosedyrer gjør at vi kan navigere mellom foreldre og barn uten å bruke pekere. • Nøkkelen: Arrayrepresentasjon fungerer fordi heapen alltid er komplett. • Disse prosedyrene brukes ved innsetting (boble opp) og sletting (boble ned) av elementer i heapen.

Binære heaps – Innsetting (implementasjon) Innsetting i en binær heap handler om å legge til et nytt element samtidig som heap-invarianten opprettholdes. Heap-invarianten sier at hver node er større enn foreldrenoden (min-heap) og at treet alltid er komplett.

Hovedidé 1. Legg det nye elementet på neste ledige plass i treet (for å holde treet komplett). 2. Sammenlign elementet med foreldrenoden: ○ Hvis elementet er mindre enn foreldrenoden (i en min-heap), bytt plass. ○ Fortsett prosessen rekursivt oppover til roten (boble opp). 3. Stop når elementet ikke lenger er mindre enn foreldrenoden, eller når du når roten.

Algoritme Procedure Insert(A, x) A[n] ← x // Legg x på neste ledige plass i ← n // Start med den nye noden while 0 < i and A[i] < A[ParentOf(i)] do A[i], A[ParentOf(i)] ← A[ParentOf(i)], A[i] // Bytt med forelder i ← ParentOf(i) // Flytt oppover i treet • Input: Array A som representerer en heap med n elementer, og elementet x. • Output: Oppdatert array som fortsatt oppfyller heap-egenskapene.

Praktiske hensyn ved implementasjon • Array-størrelse: Vi antar at arrayet er stort nok til å legge til et nytt element. ○ Hvis arrayet er fullt, må vi lage et nytt, større array. ○ En vanlig strategi: doble størrelsen på arrayet og kopiere alle elementene. • Dynamiske arrays: Kan bruke f.eks. ArrayList i Java eller list i Python, som håndterer resizing automatisk. • Redusere arrayet: Hvis heapen blir veldig liten i forhold til arrayets kapasitet, kan arrayet gjøres mindre igjen for å spare minne.

Kompleksitet • Hver «boble opp» operasjon tar O(log n) tid i verste tilfelle. • Dette er fordi vi maksimalt må gå fra en løvnoden opp til roten, og høyden på et komplett binærtre med n noder er log(n).

Binære heaps – Fjerning av minste element (implementasjon) I en min-heap er det minste elementet alltid i roten (A[0]). Fjerning av dette elementet må samtidig beholde heap-invarianten og det komplette treet.

Hovedidé 1. Ta vare på rotnoden (A[0]) som skal fjernes – dette er det minste elementet. 2. Flytt siste noden (A[n-1]) til roten. ○ Dette sørger for at treet fortsatt er komplett, siden vi fjernet en node i siste nivå. 3. Bobl ned (heapify) fra roten: ○ Sammenlign noden med barna. ○ Bytt plass med det minste barnet hvis noden er større. ○ Fortsett nedover til noden er mindre enn begge barna eller når du treffer løv.

Algoritme Procedure RemoveMin(A) x ← A[0] // Ta vare på minste element A[0] ← A[n-1] // Flytt siste element til roten i ← 0 // Start fra roten while LeftOf(i) < n−1 do j ← LeftOf(i) // Start med venstre barn if RightOf(i) < n−1 and A[RightOf(i)] < A[j] then j ← RightOf(i) // Velg minste av barna if A[i] ≤ A[j] then return x // Heap-invarianten er oppfylt A[i], A[j] ← A[j], A[i] // Bytt noden med minste barn i ← j // Fortsett nedover return x • Input: Array A med n elementer som representerer en min-heap. • Output: Minste elementet fjernes og returneres; heap-invarianten beholdes.

Hjelpeprosedyrer • LeftOf(i) = 2*i + 1* → *venstre barn av indeks i.* • *RightOf(i) = 2*i + 2 → høyre barn av indeks i. • ParentOf(i) = (i-1) // 2 → foreldrenoden til indeks i.

Kompleksitet • Tid: O(log n) i verste tilfelle, fordi vi maksimalt må flytte noden fra roten til et løv. • Plass: O(1), ingen ekstra array trengs – operasjonen skjer in-place.

Merk • Heap-invarianten må alltid sjekkes under boble-ned-prosessen. • Hvis heapen bare har én node, returneres den og heapen blir tom. • Denne metoden er svært effektiv for implementasjon av prioritetskøer, der insert og removeMin begge har O(log n) kompleksitet.