Recap

IN2010: Algoritmer og datastrukturer Pseudokode som formidlingsverktøy • Pseudokode er ikke et læringsmål i seg selv i IN2010. • Hensikten er å kunne formidle egen forståelse av algoritmer på en klar og presis måte. • En pseudokode er god dersom en kvalifisert leser kan forstå hva algoritmen gjør uten å måtte gjette eller tolke. • Man kan tenke på pseudokode som liggende på en skala mellom naturlig språk og et konkret programmeringsspråk som Python/Java: ○ Naturlig språk → lett å lese, men kan være tvetydig ○ Python/Java → presist, men mindre pedagogisk for alle lesere • Målet er å finne en balanse der pseudokoden er forståelig, presis og konsis. • Gode eksempler finnes i forelesningsslidene, i boka, og på Wikipedia.

Pseudokode – Dos & Don'ts Gjør dette: • Bruk innrykk for å tydeliggjøre blokker og kontrollstrukturer. • Bruk kontrollstrukturer som if, while, for, osv. • Bruk fornuftige variabelnavn som gir mening i konteksten. ○ Forklar navnene kort hvis det trengs. • Vær konsekvent i stil, navn og struktur. • Vær presis, slik at pseudokoden ikke er tvetydig. • Vær konsis, unngå unødvendige detaljer. Ikke gjør dette: • Ikke importer biblioteker; pseudokode bør være selvforklarende uten eksterne avhengigheter. • Ikke vær vag; unngå uttrykk som "gør noe" eller "loop til ferdig". • Ikke vær språkspesifikk: unngå Python-spesifikke konstruksjoner som slicing (A[1:3]) eller Java-spesifikke metoder (System.out.println). • Bruk heller generelle beskrivelser og antagelser når du trenger funksjonalitet som ellers krever et spesifikt språk. Kort sagt: Pseudokode skal være et tydelig og språk-uavhengig verktøy for å vise algoritmens logikk, uten å drukne leseren i språkspesifikke detaljer.

Rød-svarte trær (Red-Black Trees) Rød-svarte trær er en type selvbalanserende binært søketre, på samme måte som AVL-trær. Målet med slike trær er å holde treet så balansert som mulig for å sikre at operasjoner som innsetting, sletting og oppslag skjer effektivt. Likheter med AVL-trær: • Begge er selvbalanserende binære søketrær, altså de opprettholder en balanse slik at høyden ikke blir for stor. • Begge har O(log n) tidskompleksitet på innsetting, sletting og oppslag. • Begge bruker rotasjoner (venstre- og høyre-rotasjoner) for å bevare balansen i treet. Forskjeller fra AVL-trær: • Balansekrav: ○ Rød-svarte trær har svakere balansekrav enn AVL-trær. AVL-trær må alltid ha balansefaktor -1, 0 eller 1 for alle noder, mens rød-svarte trær tillater litt mer fleksibilitet. • Minnebruk: ○ Rød-svarte trær bruker litt mindre minne, fordi de ikke trenger å lagre høyden til hver node (kun fargeinformasjon – rød eller svart). • Antall rotasjoner: ○ Rød-svarte trær krever generelt færre rotasjoner ved innsetting og sletting enn AVL-trær, noe som kan gi raskere oppdateringer.

Rød-svarte trær vs AVL-trær – ytelsesaspekt • Innsetting og sletting: ○ Rød-svarte trær er raskere når innsetting og sletting skjer ofte. ○ Færre rotasjoner betyr at treet trenger mindre reorganisering, og operasjonene blir mer effektive. • Oppslag (search): ○ AVL-trær er raskere når oppslag skjer ofte. ○ Fordi AVL-trær er mer strengt balanserte, er høyden på treet ofte lavere, noe som gir kortere vei fra roten til en hvilken som helst node. Oppsummering: • Velg rød-svarte trær når treet oppdateres ofte (innsetting/sletting). • Velg AVL-trær når treet er stort og oppslagene dominerer (lite oppdatering).

Invarianter for rød-svarte trær Rød-svarte trær er selvbalanserende binære søketrær, og deres struktur er definert av fem viktige invarianter som alltid må holdes: 1. Farge: Hver node er enten rød eller svart – dette gir navnet "rød-svart tre". 2. Roten: Roten til treet er alltid svart. 3. Tomme noder: Alle tomme (null-) noder eller blader betraktes som svarte. 4. Rød-regel: En rød node kan ikke ha et rødt barn. Dette

forhindrer for lange kjeder med røde noder og bidrar til balanse. 5. Svart-balansen: Hver gren fra roten til et tomt blad inneholder samme antall svarte noder. Dette sikrer at treet er tilstrekkelig balansert.

Intuisjon bak invariantsystemet • Verste tilfelle for balanse: ○ En gren består kun av svarte noder ○ En annen gren består av rød og svart annenhver • Dette betyr at den lengste grenen kan være maks dobbelt så lang som den korteste. • Selv om treet ikke er perfekt balansert som et AVL-tre, holder denne regelen treets høyde i O(log n). Konsekvens: • Operasjoner som innsetting, sletting og oppslag skjer fortsatt i O(log n) tid, selv i verstefallsscenarier. • Rød-svarte trær kombinerer derfor effektivitet med litt mer fleksibel balanse enn AVL-trær, noe som gjør dem mer effektive for hyppige oppdateringer.

Invarianter for rød-svarte trær Rød-svarte trær er selvbalanserende binære søketrær, og deres struktur styres av fem viktige invarianter: 1. Farge på noder: Hver node er enten rød eller svart – dette gir navnet rød-svart tre. 2. Roten er svart: Roten av treet må alltid være svart. Dette sikrer at balansen starter korrekt fra toppen. 3. Tomme noder er svarte: Alle tomme (null-) noder regnes som svarte. Dette forenkler definisjonen av svartbalansen. 4. Rød-regel: En rød node kan ikke ha et rødt barn. Dette hindrer kjeder av røde noder som kan gjøre treet ubalansert. 5. Svart-balansen: Hver gren fra roten til et tomt blad inneholder samme antall svarte noder. Dette sikrer at alle grener er relativt balanserte.

Intuisjon bak rød-svart balanse • Verste tilfelle for balanse: ○ Én gren består kun av svarte noder. ○ En annen gren består av røde og svarte noder annenhver. • Høydeforskjell: Den lengste grenen kan maksimalt være dobbelt så lang som den korteste. • Dette høres mye ut, men det er ikke kritisk: treets høyde forblir O(log n), så operasjoner som innsetting, sletting og oppslag forblir effektive. Konsekvens: • Rød-svarte trær er ikke like strengt balanserte som AVL-trær, men de opprettholder fortsatt logarithmisk høyde. • De er derfor spesielt nyttige i situasjoner med hyppige innsettinger og slettinger, fordi de krever færre rotasjoner enn AVL-trær.

Heapsort – Idé Heapsort er en effektiv sorteringsalgoritme som kombinerer heap-struktur med array-manipulasjon. • Hovedidé: 1. Bygg en heap fra arrayet. 2. Fjern (pop) elementer fra heapen en etter en og plasser dem på riktig posisjon i arrayet. • Fordi en heap kan representeres som et array, kan vi gjøre hele sorteringen in-place uten ekstra minne. Algoritmisk fremgangsmåte: 1. Transformér arrayet til en max-heap (rot alltid største element). 2. Sett i=n−1, der n er størrelsen på arrayet. 3. Pop fra max-heapen (bytt roten med element på posisjon i) og reduser heap-størrelsen. 4. Decrement i og gjenta steg 3 til i=0. • Resultatet blir et sortert array i stigende rekkefølge. • Heapsort har alltid O(n log n) tidskompleksitet og krever ingen ekstra lagringsplass.

Heapsort – Bygge en max-heap • En max-heap er en heap der hver node er større enn begge barna sine. ○ I motsetning til en min-heap, hvor hver node er mindre enn barna. • Array-representasjon av heap: ○ Roten ligger på posisjon 0. ○ Venstre barn for node i ligger på 2i+1. ○ Høyre barn for node i ligger på 2i+2. • Bygging av max-heap (BuildMaxHeap): ○ Start fra midten av arrayet og gå mot venstre (alle noder etter midten er blader, så de er allerede heap). ○ For hver node, sammenlign med barna: § Hvis foreldrenoden er mindre enn et av barna, bytt plass med det største barnet. § Fortsett rekursivt nedover for å sikre heap-egenskapen. ○ Etter siste iterasjon: § Det største elementet ligger i roten. § Alle foreldrenoder er større enn eller lik begge barnenoder. • Intuisjon: ○ BuildMaxHeap sørger for at arrayet oppfører seg som en komplett binærtrestruktur med heap-egenskapen, som er fundamentet for effektiv heapsort.

Heapsort – Bygge en max-heap (implementasjon) For å implementere heapsort, trenger vi først å kunne bygge en max-heap. Dette gjøres med to hovedprosedyrer: BubbleDown og BuildMaxHeap.

BubbleDown-prosedyre Hensikt: Sikre at noden på posisjon i i heapen oppfyller max-heap-egenskapen, dvs. at den er større enn begge barna sine. Pseudokode: Procedure BubbleDown(A, i, n) largest ← i left ← 2i + 1 right ← 2i + 2 if left < n and A[largest] < A[left] then largest ← left if right < n and A[largest] < A[right] then largest ← right if i ≠ largest then Swap A[i] and A[largest] BubbleDown(A, largest, n) Forklaring: • largest starter som noden i. • Vi sjekker om venstre og høyre barn finnes og er større enn A[i]. • Hvis et barn er større, bytter vi plass og fortsetter nedover rekursivt. • På denne måten "bobler" det største elementet nedover til riktig posisjon i heapen.

BuildMaxHeap-prosedyre Hensikt: Gjør hele arrayet om til en max-heap. Pseudokode: Procedure BuildMaxHeap(A, n) for i ← ⌊n/2⌋ down to 0 do BubbleDown(A, i, n) Forklaring: • Vi starter i midten av arrayet (alle elementer etter midten er blader, og trenger ingen behandling). • Vi går mot venstre, og for hver node utfører vi BubbleDown. • Etter siste iterasjon er rot-noden det største elementet, og heap-egenskapen er oppfylt for alle noder.

Heapsort – Implementasjon Når max-heapen er bygget, kan vi sortere arrayet: Pseudokode: Procedure HeapSort(A) BuildMaxHeap(A, n) for i ← n-1 down to 0 do Swap A[0] and A[i] // Flytt største element til slutt BubbleDown(A, 0, i) // Reparer heapen med redusert størrelse return A Forklaring: 1. Bygg først en max-heap fra arrayet. 2. Bytt roten (største element) med siste element i heapen. 3. Reduser heap-størrelsen (i) og reparer heapen med BubbleDown. 4. Gjenta til hele arrayet er sortert. Egenskaper ved Heapsort: • Tidskompleksitet: O(n log n) i alle tilfeller (beste, gjennomsnitt og verste). • Minnebruk: In-place, krever ikke ekstra array. • Stabilitet: Heapsort er ikke stabil, dvs. like elementer kan endre rekkefølge.

Heapsort – Kjøretidsanalyse (BubbleDown) BubbleDown er nøkkelen til heapsort, og dens kjøretid bestemmer effektiviteten. Observasjoner: 1. Linjene som gjør selve sammenligningen og byttingen (linje 2–13) tar konstant tid. 2. Viktigste faktor er antall rekursive kall. Analyse: • Algoritmen terminerer når i≥n. • Hvert rekursive kall går til enten venstre barn (2i + 1) eller høyre barn (2i + 2). • Altså dobles indeksen for hvert rekursive kall. • En heap med høyde h har mindre enn 2h+1 noder. • Siden heapen er et komplett binærtre, er h∈O(logn). Konklusjon: • Maksimalt antall rekursive kall er h. • Dermed er BubbleDown i O(log n).

Heapsort – Kjøretidsanalyse (BuildMaxHeap) • BuildMaxHeap kaller BubbleDown for omtrent halve nodene i arrayet (n/2 noder). • Siden BubbleDown er O(logn), kan man intuitivt tro at BuildMaxHeap er O(nlogn). • Men det er faktisk O(n)! Intuisjon: • Nær bunnen av heapen finnes mange blader som ikke trenger å bli "bubbled down". • Høyere noder har få barn, så de har korte rekursive kjeder. • Kun roten kan nå verste tilfelle og måtte traversere hele høyden. • Fordelingen av rekursive kall gjør at summen totalt blir lineær i n. Pseudokode: Procedure BuildMaxHeap(A, n) for i ← ⌊n/2⌋ down to 0 do BubbleDown(A, i, n)

Heapsort – Kjøretidsanalyse (HeapSort) Etter at max-heapen er bygget: 1. Vi gjør n iterasjoner, hvor vi: ○ Bytter roten (største element) med siste element i heapen. ○ Reduserer heap-størrelsen med 1. ○ Kaller BubbleDown fra roten. 2. Hver BubbleDown-operasjon tar O(log n). Totalt: • BuildMaxHeap: O(n) • n iterasjoner av BubbleDown: n⋅O(logn)=O(nlogn) Konklusjon: • Heapsort har alltid O(n log n) kjøretid, uansett input. Pseudokode for Heapsort: Procedure HeapSort(A) BuildMaxHeap(A, n) for i ← n-1 down to 0 do Swap A[0] and A[i] BubbleDown(A, 0, i) return A

Oppsummering av tidskompleksitet: Operasjon Kjøretid BubbleDown O(log n) BuildMaxHeap O(n) HeapSort (totalt) O(n log n) Kommentar: • Heapsort er effektiv, in-place, og uavhengig av input-data. • Det er ikke stabilt, men svært godt egnet for store datasett med mange innsettinger og slettinger.

Huffman-koding Hensikt: Huffman-koding brukes for tapløs datakomprimering ved å representere symboler med bitstrenger på en optimal måte. • Input: Et alfabet av symboler, der hvert symbol har en relativ frekvens. • Mål: Representere hvert symbol med en bitstreng slik at den totale lengden av bitstrenger for en tekst blir så liten som mulig. • Terminologi: ○ Enkoding: Mapping fra symboler til bitstrenger. ○ Kodeord: Den enkelte bitstrengen som representerer et symbol.

Huffman-koding – Fast lengde • Hvis vi bruker fast lengde n for alle symboler: ○ Med n bits kan vi representere 2n symboler. ○ For å representere m symboler trenger vi minst ⌈log2(m)⌉ bits. • Kostnad for en streng s med lengde |s| blir: |s|·n bits • Ulempe: Denne metoden tar like mye plass for alle symboler, uavhengig av hvor ofte de forekommer.

Huffman-koding – Variabel lengde • I praksis forekommer noen symboler oftere enn andre. • Huffman-koding gir kortere kodeord til vanlige symboler og lengre kodeord til sjeldne symboler. • Dette reduserer den totale lengden på bitstrengen for en tekst betydelig. • Algoritmen: 1. Lag en prioritetskø med symboler sortert etter frekvens. 2. Bygg et binært tre, hvor de minst hyppige symbolene kombineres først. 3. Hvert symbol får en bitstreng basert på veien fra roten til bladet (0 for venstre, 1 for høyre).

Huffman-koding – Prefiks-egenskap • Når kodeordene har variabel lengde, oppstår problemet med tvetydighet: ○ Hvordan vet man hvor ett symbol slutter og et annet begynner? • Prefiks-regel: Ingen kodeord kan være prefiks av et annet. ○ Eksempel: § Hvis 010 er et kodeord, kan ikke 0101 være et kodeord. § Men 0001 kan være et annet kodeord, fordi det ikke starter med 010. • Denne egenskapen gjør det mulig å dekode sekvenser uten tvetydighet.

Oppsummering: • Huffman-koding gir optimal variabel-lengde enkoding basert på frekvenser. • Prefiks-egenskapen sikrer entydig dekoding. • Kostnadsbesparelse sammenlignet med fast-lengde koding er størst når frekvensene varierer mye.

Huffman-koding – Frekvenstabell For å illustrere Huffman-koding kan vi se på en eksempelsetning: "det er veldig vanskelig å finne på en eksempelsetning" Frekvenstabell: Symbol a d e f g i k l m n p r s t v å Frekvens 8 1 2 10 1 3 4 2 3 1 6 2 1 3 2 2 Fast-lengde koding: • For 17 forskjellige symboler trenger vi minst ⌈log2(17)⌉=5 bits per symbol. • Totalt for setningen med 53 symboler: 53·5=265 bits Huffman-koding: • Optimal variabel-lengde koding basert på frekvenser. • Kortere kodeord for vanlige symboler, lengre for sjeldne. • Totalt for samme setning: 198 bits, betydelig reduksjon fra fast-lengde koding. Dette er den mest effektive koding man kan få med prefiks-egenskapen intakt.

Huffman-koding – Huffman-trær Bygging av Huffman-tre: 1. Hver løvnode representerer et symbol i alfabetet. 2. Hver node har en assosiert frekvens, startende med symbolenes frekvens. 3. Bygging skjer iterativt: ○ Velg de to nodene med lavest frekvens. ○ Lag en ny intern node som får disse to som barn. ○ Frekvensen til den nye noden = sum av barnenes frekvenser. ○ Sett noden tilbake i prioritetskøen. ○ Gjenta til det kun er én node igjen (roten). Kodeord fra treet: • Sti fra rot til løv bestemmer kodeordet: ○ Venstre gren = 0 ○ Høyre gren = 1 • Symboler som forekommer sjeldent vil ofte ligge dypere i treet → lengre kodeord. • Symboler som forekommer ofte ligger nær roten → kortere kodeord. Eksempel: • Hvis symbolet e har høy frekvens, vil koden være kort, f.eks. 10. • Hvis symbolet g har lav frekvens, vil koden være lengre, f.eks. 01101. Intuisjon: • Dette sikrer minst mulig total bitlengde for teksten. • Huffman-trær utnytter effektivt frekvensfordelingen i teksten.

Huffman-koding – Bygge Huffman-trær Hensikt: Å bygge et Huffman-tre er en systematisk måte å lage optimale kodeord for symboler basert på frekvens. Struktur på noder i Huffman-treet: • Element: Selve symbolet (kan være null for interne noder). • Venstre og høyre barn: Referanser til undertrær. • Frekvens

(freq): Summen av frekvensene til alle løvnoder under noden. ○ Nodene ordnes i prioritetskø basert på freq. Algoritme (intuisjon): 1. Opprett en tom prioritetskø. 2. For hvert symbol og tilhørende frekvens i alfabetet: ○ Lag en løvnode (uten barn) med symbolet og frekvensen. ○ Sett noden inn i køen. 3. Så lenge køen har mer enn ett element: ○ Fjern de to nodene med lavest frekvens (v1 og v2). ○ Lag en ny intern node u: § u.freq = v1.freq + v2.freq § v1 og v2 blir barn av u (f.eks. venstre/høyre). ○ Sett u tilbake i køen. 4. Når det bare er én node igjen i køen, er dette roten av Huffman-treet. Denne prosessen sikrer at de minst frekvente symbolene havner lengst fra roten, og de mest frekvente symbolene nærmest roten, noe som gir kortest mulig totale bitstreng for teksten.

Huffman-koding – Implementasjon (Pseudokode) ALGORITHM: BYGGE HUFFMAN TRÆR Input: En mengde C med par (s, f) der s er symbol og f er frekvens Output: Et Huffman-tre 1 Procedure Huffman(C) 2 Q ← new PriorityQueue 3 for (s, f) ∈ C do 4 Insert(Q, new Node(s, f, null, null)) // Lag løvnode 5 while Size(Q) > 1 do 6 v1 ← RemoveMin(Q) // Min frekvens 7 v2 ← RemoveMin(Q) 8 f ← v1.freq + v2.freq // Summen av frekvenser 9 Insert(Q, new Node(null, f, v1, v2)) // Lag intern node 10 return RemoveMin(Q) // Rot av treet Forklaring: • RemoveMin(Q) henter noden med lavest frekvens. • Insert(Q, node) setter noden tilbake i prioritetskøen slik at køen alltid er sortert etter frekvens. • Til slutt står roten igjen, og stier fra rot til løv gir Huffman-kodeordene.

Intuisjon: • Huffman-algoritmen bygger treet nedover fra de minst hyppige symbolene, slik at sjeldne symboler får lengre koder. • Dette gir optimal variabel-lengde kodingsløsning med prefiks-egenskap, som gjør dekoding enkel og entydig.