Trær, Binære Søketrær og AVL-trær

Oversikt • Trær er en type hierarkisk datastruktur, der hvert element (node) kan ha flere barn, men kun ett foreldre-element (unntatt roten som ikke har foreldre). • Trær brukes mye i søking, sortering, og organisering av data der hierarki eller relasjoner mellom elementer er viktig. • Binære søketrær (BST – Binary Search Trees) er en spesiell type trær som tillater rask oppslag, innsetting og sletting av elementer.

Binære søketrær (BST) • Hver node har maksimalt to barn, kalt venstre og høyre. • Egenskap for et binært søketre: ○ Venstre barn har verdi mindre enn nodens verdi. ○ Høyre barn har verdi større enn nodens verdi. • Dette gjør det mulig å gjøre logaritmisk søk (i beste tilfelle, når treet er balansert): ○ Start i roten ○ Sammenlign med nodens verdi ○ Hvis mindre → gå til venstre undertre ○ Hvis større → gå til høyre undertre • Operasjoner som søk, innsetting og sletting har O(h) kjøretid, der h er treets høyde. • Ubalanserte trær kan føre til høyde nær n → i verste fall O(n) for søk.

AVL-trær – selvbalanserende binære søketrær • AVL-trær er en balansert variant av binære søketrær utviklet for å sikre at høyden til treet alltid er logaritmisk i antall noder. • Balansefaktor til en node = høyde venstre subtre – høyde høyre subtre ○ Må være –1, 0 eller +1 for at treet skal være balansert. • Etter innsetting eller sletting kan treet bli ubalansert → AVL-trær bruker rotasjoner for å gjenopprette balansen: ○ Enkeltrotasjon (venstre eller høyre) ○ Dobbeltrotasjon (venstre-høyre eller høyre-venstre) • Fordel: Operasjoner (innsetting, sletting, søk) har alltid O(log n) kjøretid.

Oppsummering • Binære søketrær: effektivt for oppslag, men kan bli ubalansert. • AVL-trær: selvbalanserende BST, sikrer logaritmisk høyde og dermed konsekvent effektivitet. • Viktig konsept: balansefaktor og rotasjoner gjør AVL-trær pålitelige for store datasett.

Trær Anvendelser av trær Trær brukes hovedsakelig i to typer situasjoner:

1. Hierarkiske data ○ Dataene har en naturlig hierarkisk struktur, f.eks.: § Fil- og mappestrukturer i operativsystemer § Abstrakte syntakstrær (AST) i kompilatorer § HTML- eller XML-dokumenter
2. Effektiv organisering av datasamlinger ○ Trær brukes for datastrukturer som trenger rask tilgang, innsetting eller sletting: § Mengder (Set) § Ordbøker/Maps (Key-Value lagring)

Lister som spesialtilfelle av trær • En liste kan ses som et enkelt tre: ○ Tom liste: ofte representert med null ○ Node: består av et element + peker til neste node • Alle lister er dermed trær, men ikke alle trær er lister. • Trær er en utvidelse av lister hvor hver node kan ha flere barn/pekere.

Eksempler på trær • Programmering: Kompilatorer omformer kode til abstrakte syntakstrær (AST). • Web: HTML-dokumenter representeres som trær. En nettside er en visualisering av et tre. • Filsystemer: Mapper og filer danner hierarkiske trestrukturer. • Spill: Alle mulige sjakkpartier kan representeres som et enormt tre av trekk.

Definisjon av et tre Et tre er definert rekursivt: • Tomt tre: ofte representert som null. • Node v: består av: ○ Et element/data ○ 0 eller flere pekere til barnenoder ○ Én peker til foreldre (unntatt roten, som ikke har forelder) • Regler: ○ Et tre kan ikke inneholde sykler (fra en node kan du ikke følge pekere tilbake til samme node). ○ Merk: Boken tillater ikke tomme trær.

Terminologi for trær • Node: et element i treet • Rot (root): toppnoden, ingen forelder • Forelder (parent): noden som har pekere til barn • Barn (child): noden som mottar en peker fra en forelder • Søsken (siblings): noder med samme forelder • Løvnoder / eksterne noder (leaf nodes): noder uten barn • Interne noder: noder med minst ett barn • Subtre: et tre bestående av en node og alle dens etterkommere • Forfedre (ancestors): alle noder på veien fra roten til en gitt node • Etterkommere (descendants): alle noder under en gitt node • Sti (path): sekvensen av noder fra en node til en annen via pekere Eksempel: • Treet har roten v1 • Barn av v1: v2, v3, v4 (søsken) • Interne noder: v1, v2, v3... • Løvnoder: v11, v12, ..., v16 • Subtre med roten v3: inkluderer v3, v7, v8, v13, v14 • Sti fra v1 til v14: v1 → v3 → v8 → v14

Trær – Datastruktur • Tomt tre: Representeres ofte med null. • Node v: En node består av flere felter: ○ v.element: dataen lagret i noden ○ v.parent: peker til foreldrenoden (null for roten) ○ v.children: liste/array med barnenoder • Denne strukturen gjør det mulig å representere trær med vilkårlig antall barn per node.

Trær – Dybde • Dybden til en node: antall steg fra roten til noden. ○ Roten har dybde 0 ○ For andre noder: depth(v) = depth(v.parent) + 1 • Tomme trær: tildeles dybde -1 for å skille dem fra reelle noder. Algoritme for dybde: ALGORITHM: FINN DYBDEN AV EN NODE Input: Node v Output: Dybden til noden Procedure Depth(v) if v = null then return -1 return 1 + Depth(v.parent) • Rekursiv løsning: går oppover til roten og teller antall steg.

Trær – Høyde • Høyden av en node: lengden på den lengste stien fra noden til en løvnode (dypeste etterkommer). • Høyden til et tre: høyden til roten. Algoritme for høyde: ALGORITHM: FINN HØYDEN AV EN NODE Input: Node v Output: Høyden til noden Procedure height(v) h ← -1 if v = null then return h for c ∈ v.children do h ← Max(h, height(c)) return 1 + h • Vi bruker en rekursiv tilnærming som går gjennom alle barn og tar maksimumshøyden.

Trær – Traversering • Traversering betyr å besøke alle noder i et tre på en systematisk måte. • Traverseringsrekkefølgen er viktig når operasjoner skal utføres. Typer traversering

1. Preorder (node først): ○ Operasjonen utføres på noden først, deretter på barna. ○ Eksempel: kopiere et tre (du lager noden før du kopierer barna).
2. Postorder (barn først): ○ Operasjonen utføres først på alle barna, så på noden selv. ○ Eksempel: slette et tre (du sletter barna før du sletter forelderen). • Traversering kan implementeres både rekursivt og iterativt med stakk. • Andre typer traversering (for binære trær): inorder (venstre → node → høyre).

Trær – Preorder og Postorder Traversering Traversering brukes for å besøke alle nodene i et tre systematisk, og rekkefølgen vi velger har betydning for hva vi kan gjøre med nodene underveis.

Preorder Traversering • Idé: Besøk noden først, deretter barna. • Typisk bruk: Kopiere et tre, skrive ut nodene i en hierarkisk rekkefølge. • Rekursivt pseudokode-eksempel: ALGORITHM: PREORDER TRAVERSERING Input: Node v (ikke null) Output: Utfør en operasjon på v først, deretter på barna Procedure Preorder(v) if v = null then return Operate on v // f.eks. skriv ut v.element for c ∈ v.children do Preorder(c) • Forklaring: 1. Sjekker om noden er null – base case i rekursjon. 2. Utfører ønsket operasjon på noden. 3. Kaller rekursivt på alle barn, fra venstre til høyre.

Postorder Traversering • Idé: Besøk barna først, deretter noden selv. • Typisk bruk: Slette et tre, beregne aggregert verdi fra barn til forelder. • Rekursivt pseudokode-eksempel: ALGORITHM: POSTORDER TRAVERSERING Input: Node v (ikke null) Output: Utfør operasjon på barna først, deretter v Procedure

Postorder(v) if v = null then return for c ∈ v.children do Postorder(c) Operate on v // f.eks. slett eller summer elementer • Forklaring: 1. Base case: stopper rekursjon når noden er null. 2. Går gjennom alle barn først, rekursivt. 3. Utfører operasjon på noden selv etter at alle barn er behandlet.

Oppsummering av forskjellene Traversering Rekkefølge Typisk bruk Preorder Node → Barn Kopiering, utskrift, hierarkisk prosessering Postorder Barn → Node Sletting, beregning fra barn → forelder • Begge metodene kan også implementeres iterativt ved hjelp av en stakk, men rekursjon er ofte enklere og mer intuitiv for trær.

Binære søketrær (Binary Search Trees – BST) Binære trær • Et binærtre er en trestruktur der hver node kan ha maksimalt to barn. • Vi refererer til barna som venstre og høyre barn. • Hver node v har typisk: ○ v.element → verdien som lagres i noden ○ v.left → venstre barn (null hvis ingen) ○ v.right → høyre barn (null hvis ingen)

Binære søketrær • Et binært søketre (BST) er et binærtre med en spesiell egenskap: ○ Alle elementer i venstre subtre er mindre enn nodens element ○ Alle elementer i høyre subtre er større enn nodens element • Vi kan tillate duplikater ved å bruke ≥ eller ≤ • Elementene må være sammenlignbare (kan ordnes) • Balanserte trær er spesielt effektive, fordi de holder høyden lav og gir raskere operasjoner

Sammenheng med binærsøk • Binærsøk halverer søkeområdet for hver sammenligning → O(log n) tid • Problem: dynamiske strukturer som stadig legger til og fjerner elementer gjør arrays upraktiske • Binære søketrær gir en dynamisk datastruktur som støtter: ○ Effektivt oppslag ○ Effektiv innsetting ○ Effektiv sletting

Innsetting i et BST Pseudokode – rekursiv innsetting Procedure Insert(v, x) if v = null then v ← new Node(x) else if x < v.element then v.left ← Insert(v.left, x) else if x > v.element then v.right ← Insert(v.right, x) return v • Forklaring: 1. Hvis treet er tomt, opprett en ny node. 2. Hvis verdien er mindre enn nodens element → gå venstre. 3. Hvis verdien er større → gå høyre. 4. Returner noden etter at subtreet er oppdatert. • Kompleksitet: ○ O(h) der h er treets høyde ○ I verste tilfelle (skjev trestruktur) → O(n) ○ Hvis treet er balansert → O(log n)

Oppslag i et BST Pseudokode – rekursivt oppslag Procedure Search(v, x) if v = null then return null if v.element = x then return v if x < v.element then return Search(v.left, x) else return Search(v.right, x) • Forklaring: 1. Stopper hvis noden er null (elementet finnes ikke). 2. Hvis elementet matcher → returner noden. 3. Hvis elementet er mindre → søk i venstre subtre. 4. Hvis elementet er større → søk i høyre subtre. • Kompleksitet: Samme som innsetting → O(h)

Notater til eksamen • Høyden på treet (h) er kritisk for effektivitet. • Et balansert BST gir logaritmisk tid for innsetting, sletting og oppslag. • Hvis treet blir skjevt, for eksempel når data allerede er sortert, kan operasjonene bli lineære (O(n)).

Sletting i binære søketrær (BST Deletion) • Sletting i et binærtre er mer komplisert enn innsetting eller oppslag. • Kompleksiteten forblir O(h), der h er høyden på treet. • Hovedmålet ved sletting er å beholde BST-egenskapene og tette eventuelle hull i trestrukturen.

Tre hovedtilfeller ved sletting

1. Noden har ingen barn (løvnode)
2. Noden har ett barn (enten venstre eller høyre)

3. Noden har to barn

4. Sletting – ingen barn (løvnode) • Noden vi vil slette x har ingen barn. • Da er det tilstrekkelig å fjerne pekeren fra foreldrenoden til x. • Eksempel: T0
   x T1 → Etter sletting: T0
   T1 • Dette er det enkleste tilfellet.

5. Sletting – ett barn (venstre) • Noden x har kun venstre barn (T2). • Vi erstatter x med T2, slik at treet fortsatt er sammenhengende. Før: Etter sletting: T0 T0 | | x T2 /
   T2 T1

6. Sletting – ett barn (høyre) • Noden x har kun høyre barn (T2). • Tilsvarende som for venstre barn, vi erstatter x med T2. Før: Etter sletting: T0 T0 | | x T2 \
   T2 T1

7. Sletting – to barn • Noden x har to barn (venstre T1 og høyre T2). • Strategi: 1. Finn det minste elementet y i høyre subtre (in-order successor). 2. Erstatt x.element med y.element. 3. Slett deretter noden y, som nå er et enklere tilfelle (løvnode eller har ett barn). Før: x /
   T1 T2 /
   y T3 Etter: y /
   T1 T2
   T3 • Dette bevarer BST-egenskapene.

Notater til eksamen • Sletting beholder treets struktur og BST-egenskapene. • Kompleksitet: O(h) → O(n) i skjev trestruktur, O(log n) i balansert tre. • To-barn-tilfellet er det mest komplekse, fordi vi må finne in-order successor eller in-order predecessor. • Å forstå de tre tilfellene grafisk hjelper mye under eksamen.

Finne minste element i et binært søketre • Ved sletting trenger vi ofte å finne den minste noden i et subtre, spesielt når vi sletter en node med to barn. • Den minste noden finnes alltid i venstre gren av subtreet. • Algoritmisk: Procedure FindMin(v) 1 if v.left = null then 2 return v 3 else 4 return FindMin(v.left) • Forklaring: ○ Gå helt til venstre fra noden v. ○ Den venstre noden uten venstre barn er den minste noden.

Sletting av en node i et binært søketre • Algoritmen må håndtere alle tre tilfeller: ingen barn, ett barn, to barn. • Pseudokode med kommentarer: Procedure Remove(v, x) 1 if v = null then 2 return null // x finnes ikke 3 if x < v.element then 4 v.left ← Remove(v.left, x) // gå til venstre subtre 5 return v 6 if x > v.element then 7 v.right ← Remove(v.right, x) // gå til høyre subtre 8 return v // x == v.element: slett denne noden 10 if v.left = null then 11 return v.right // kun høyre barn eller ingen barn 12 if v.right = null then 13 return v.left // kun venstre barn // to barn 14 u ← FindMin(v.right) // finn minste i høyre subtre 15 v.element ← u.element // erstatt element 16 v.right ← Remove(v.right, u.element) // slett duplikat 17 return v • Kompleksitet: O(h), hvor h er høyden på treet. • Opprettholder alltid BST-egenskapene.

AVL-trær – introduksjon • Et AVL-tre er et selvbalanserende binært søketre. • Egenskaper: ○ Oppfyller alle vanlige BST-egenskaper. ○ Høydeforskjellen mellom venstre og høyre subtre for hver node ≤ 1. • Må alltid oppdateres ved innsetting og sletting for å bevare balansen. • Oppslag i AVL-trær er uforandret fra vanlige BST-er.

Høyde i AVL-trær • Vi utvider nodene med høydeinformasjon: v.element // lagret data v.left // venstre barn v.right // høyre barn v.height // høyde på noden • Definisjon av høyde: ○ Tomt tre: -1 ○ Node: 1 + høyeste subtre • Hjelpeprosedyre for høyde: Procedure Height(v) 1 if v = null then 2 return -1 3 return v.height •

Prosedyre for å sette høyde: Procedure SetHeight(v) 1 if v = null then 2 return 3 v.height ← 1 + Max(Height(v.left), Height(v.right)) • Viktig: ○ Etter innsetting eller sletting, oppdater høyder fra den endrede noden opp til roten. ○ Dette gjør det mulig å sjekke balanse og rotere treet dersom nødvendig.

AVL-trær – overordnet idé • Vi starter med vanlige innsetting og sletting i binære søketrær. • Etter hver operasjon må vi sjekke balansen fra noden der endringen skjedde opp til roten. • Balansering: ○ Hvis høydeforskjellen mellom venstre og høyre subtre > 1, må noden balanseres. ○ Balansering skjer via rotasjoner: venstre, høyre, eller doble (venstre-høyre eller høyre-venstre). • Viktige observasjoner: ○ Før operasjonen har treet maks høydeforskjell 1. ○ Én innsetting eller sletting kan maksimalt skape en midlertidig høydeforskjell på 2 i ett punkt. • Under rotasjon må vi alltid oppdatere høydene til de berørte nodene.

Venstrerotasjon (LeftRotate) • Situasjon: noden z har fått en høyre-tung ubalanse. • Rotasjonen gjør y = z.right til ny rot, og z blir venstre barn av y. Procedure LeftRotate(z) 1 y ← z.right 2 T1 ← y.left // midlertidig lagring av subtre 3 y.left ← z 4 z.right ← T1 5 SetHeight(z) 6 SetHeight(y) 7 return y • Effekt: balanserer høyre-tungt tre lokalt. • Høydene oppdateres først for z og deretter for y. Illustrasjon: z y / \ / T0 y ---> z T3 / \ / T1 T2 T0 T1

Høyrerotasjon (RightRotate) • Situasjon: noden z har fått en venstre-tung ubalanse. • Rotasjonen gjør y = z.left til ny rot, og z blir høyre barn av y. Procedure RightRotate(z) 1 y ← z.left 2 T2 ← y.right // midlertidig lagring av subtre 3 y.right ← z 4 z.left ← T2 5 SetHeight(z) 6 SetHeight(y) 7 return y • Effekt: balanserer venstre-tungt tre lokalt. Illustrasjon: z y / \ / y T3 ---> x z / \ / x T2 T2 T3 / T0 T1

Doble rotasjoner • Noen ubalanser krever to rotasjoner for å balansere treet: ○ Venstre-høyre rotasjon (Left-Right): først RightRotate på venstre barn, så LeftRotate på roten. ○ Høyre-venstre rotasjon (Right-Left): først LeftRotate på høyre barn, så RightRotate på roten. Pseudokode eksempel (venstre-høyre): Procedure LeftRightRotate(z) 1 z.left ← LeftRotate(z.left) 2 return RightRotate(z) • Dobbel rotasjon kombinerer to enkle rotasjoner for å løse "zig-zag"-strukturer.

Balansefaktor • Definisjon: Balansefaktoren til en node v er forskjellen mellom høyden på venstre og høyre subtre: BalanceFactor(v)=Height(v.left)−Height(v.right) • Tolkning: ○ 0: noden er balansert ○ > 0: noden er venstretung (venstre subtre høyere enn høyre) ○ < 0: noden er høyretung (høyre subtre høyere enn venstre) • Pseudokode: Procedure BalanceFactor(v) 1 if v = null then 2 return 0 3 return Height(v.left) - Height(v.right) • Bruk: Hjelpeprosedyre for å bestemme om en rotasjon er nødvendig.

Balansering av en node • Vi må balansere når balansefaktoren er mindre enn –1 eller større enn 1. • Avhengig av mønsteret i subtrærne, brukes enkel rotasjon eller dobbel rotasjon. Pseudokode: Procedure Balance(v) 1 if BalanceFactor(v) < -1 then // høyretung 2 if BalanceFactor(v.right) > 0 then // høyre-venstre ubalanse 3 v.right ← RightRotate(v.right) 4 return LeftRotate(v) 5 if BalanceFactor(v) > 1 then // venstretung 6 if BalanceFactor(v.left) < 0 then // venstre-høyre ubalanse 7 v.left ← LeftRotate(v.left) 8 return RightRotate(v) 9 return v • Intuisjon: ○ Først håndteres den "indre" ubalansen med en rotasjon på barnet. ○ Så roteres noden selv for å gjenopprette balanse.

Innsetting i et AVL-tre • Vi starter med vanlig binær søketre-innsetting, men legger til: ○ Oppdatering av høyden ○ Balansering etter innsetting Pseudokode: Procedure Insert(v, x) 1 if v = null then 2 v ← new

Node(x) 3 else if x < v.element then 4 v.left ← Insert(v.left, x) 5 else if x > v.element then 6 v.right ← Insert(v.right, x) 7 SetHeight(v) // høyden av noden oppdateres 8 return Balance(v) // balanser noden • Kompleksitet: O(logn) dersom treet er balansert.

Sletting i et AVL-tre • Starter med vanlig binær søketre-sletting, men inkluderer: ○ Oppdatering av høyder ○ Balansering etter sletting Pseudokode: Procedure Remove(v, x) 1 if v = null then 2 return null 3 if x < v.element then 4 v.left ← Remove(v.left, x) 5 else if x > v.element then 6 v.right ← Remove(v.right, x) 7 else if v.left = null then 8 v ← v.right 9 else if v.right = null then 10 v ← v.left 11 else 12 u ← FindMin(v.right) 13 v.element ← u.element 14 v.right ← Remove(v.right, u.element) 15 SetHeight(v) 16 return Balance(v) • Forklaring: ○ Hvis noden har 0 eller 1 barn, fjernes den enkelt. ○ Hvis noden har 2 barn, erstattes noden med den minste noden i høyre subtre. ○ Til slutt oppdateres høyden og noden balanseres. • Kompleksitet: O(logn) dersom treet er balansert.