7: Trestrukturer

mandag 28. september 2020

Oppsplitting videoer

1. Hva er et tre? Definisjonen på et tre. Terminologi. Eksempler på trær
2. Implementasjon av trær, Binærtrær
3. Traversering av trær
4. Naive binære søketrær: Tre-basert Map implementasjon
5. Sletting i binære søketrær
6. Analyse av binære søketrær
7. AVL trær: Forbedring av naive binære søketrær. Introduksjon.
8. Vedlikehold av høydeinformasjon i AVL trær
9. Rotasjoner i AVL trær
10. Analyse av AVL trær
11. Bedre løsning på finn-k-ende minste / selection problemet for AVL trær

Innhold

- Hva er datastrukturen «tre»? Definisjon og terminologi
- Eksempler på trær
- Implementasjon av generelle trær
- Traversering av trær
- Binære søketrær: Å bruke trær for å implementere søkestrukturer
- AVL trær: Søketrær som holder seg balanserte
- Hauger og diskret hendelsessimulering kommer som egne tema, men hører på et vis inn her siden hauger er trær.

Definisjon

Rekursiv

- Et tre er enten tomt eller det består av ei rot og null eller flere undertrær T_1 … T_k som er koblet til rota med en kant (rota til hvert undertre er direkte barn av rota)

Ikke-rekursiv

- Et tre består av ei rot og en mengde noder. Hver node unntatt rota har en forelder og null eller flere barn.
- Et tre kan defineres som en graf, og er et spesialtilfelle av grafer. Jeg tar trær før grafer siden trær er enklere å håndtere enn generelle grafer.

Terminologi

- Én **node** er **rotnoden** og er der treet starter.
- Alle andre noder har én foreldernode
- En **sti** er en sekvens av noder forbundet med kanter
- Lengden på stien (antall pekere man må følge) for å komme fra en node til rotnoden er **dybden** (depth) til noden
- **Høyden** til treet som helhet er den maksimale dybden til en node (høyde er «motsatt» av dybde)
- En **bladnode** (engelsk leaf node) er en node som ikke har noen barn

- En **indre node** er en node som ikke er en bladnode
- **Søsken**: Noder som har samme forelder

- **Fullt tre:** Hver interne node har to barn
- **Perfekt tre:** Alle bladnoder er på samme nivå
- **Balansert tre:** Forskjellen i høyde på venstre og høyre subtre er maks 1
- **Komplett tre:** Alle nivåer unntatt det nederste er fullt, det nederste fylles fra venstre mot høyre.

Eksempler på trær

- Filsystemet på en vanlig datamaskin
    - Årsak: Lar brukeren kategorisere filene. En kategori kan ha underkategorier.
    - Kategorisering eksempel: nivå 1: bøker, filmer, musikkfiler, bilder. Nivå 2: sjanger for bøker og filmer, band for musikkfiler, år og sted for bilder.
    - Tegn opp dette som et tre
- Struktur på et program (nøkkelord inni metoder inni klasser)
- Søketrær (mer om de seinere i dette temaet), en mye brukt måte å implementere Sorted Set og Sorted Map ADT-ene på.
    - Idéen er å lage en struktur som holder dataene sortert hele tida slik at man ikke trenger å sortere eksplisitt.
- Matematiske formler kan uttrykkes som trær. Vis et regnestykke i treform.

Implementasjon av et generelt tre

- Hver node har et objekt og ei liste med barn
    - Fordel: raskere navigering
    - Ulempe: ineffektiv lagring da hver node må inneholde ei liste
- Hver node har et objekt, en referanse til første barn, og en referanse til nærmeste søskennode
    - Fordel: Effektiv lagring
    - Ulempe: Tregere navigering (barna blir i praksis ei lenket liste i stedet for en arraylist)
- Bruk av pekere oppover i treet for å forbedre visse algoritmer.

Binære trær

- Ved å begrense antall barn får man en mer effektiv implementasjon
- For vanlige minnebaserte trestrukturer er to et greit tall for antall barn
- Diskbaserte trestrukturer er pensum i DAT220 Databaser
- Vis basis implementasjon av binærtre

Traversering av trær

- Preorder: Besøker noden selv først, deretter venstre subtre, deretter høyre subtre
    - Eksempel: utskriftsmetoden rekursiv_preorder_utskrift
- Postorder: Besøker venstre subtre først, deretter høyre subtre, og til slutt noden selv.
    - Eksempel: utregning av verdien i et matematisk formel-tre.
- Levelorder: Traverserer treet fra topp til bunn, på hvert nivå fra venstre mot høyre.
    - I motsetning til de andre så får du ikke til denne med en enkel rekursiv algoritme. Må bruke iterator-prinsippet.

- Inorder: For binærtrær: Besøker venstre subtre først, deretter noden selv, deretter høyre subtre
    - Eksempel: Utskrift av et formel-tre på vanlig matematisk notasjon
- Rekursive algoritmer for traversering av trær
- Kan simulere rekursjon gjennom bruk av en eksplisitt stabel
- Kan bruke dette til å lage en iterator som simulerer rekursive algoritmer
- Demonstrer iteratorer for binærtre basert på Postorder. Vis hvordan denne kan modifiseres for å gjøre de andre to ordenene. Terje har en stabel-basert iterator for postorder først og deretter varianter for de andre tre. Levelorder er som preorder bortsett fra at den bruker en kø i stedet for en stabel.
- Iterator prinsipp:
    - Lag klasse som teller antall ganger en node er poppet av stabelen
    - Start med å pushe rota til stabelen med teller = 0
    - For hver iterasjon, pop element fra stabelen.
        - Teller = 0: øk teller til 1, push node, push venstre barn hvis den har et venstre barn
        - Teller = 1: øk teller med 1, push node, push høyre barn hvis den har et høyre barn
        - Teller = 2: returner noden selv

Binære søketrær

- Implementasjon av sorterte mengder og sorterte map basert på binærtrær
- Hver node inneholder en verdi. Verdien kan være objekter av enhver klasse som er sammenliknbar (Python: implementerer __lt__)
- Venstre subtre inneholder verdier som er **mindre**
- Høyre subtre inneholder verdier som er **større**
- Operasjoner
    - Finn (likhetssøk): Gå nedover i treet basert på sammenlikninger helt til du når bunnen (det du leiter etter er ikke der) eller du finner det. O(nivåer i treet)
    - Finn første over: O(nivåer i treet)
        - Sjekk verdien i noden
        - Hvis den er høyere, lagre den som midlertidig resultat og gå ned venstre subtre. Hvis det ikke er noe venstre subtre har du funnet svaret, enten her eller tidligere.
        - Hvis den er lavere, gå ned høyre subtre. Hvis noden ikke har et høyre subtre så finnes det ikke noe element over (siden algoritmen starter i rota).
    - Finn neste (høyere), gitt en node N: O(nivåer i treet) for en enkel operasjon. O(1) amortized da man besøker hver kant i treet maksimalt to ganger.
        - Hvis noden N har et høyre subtre, gå til høyre barn og deretter gå ned venstre subtre helt til noden ikke har et venstre barn. Denne noden er svaret.
        - Ellers gå et skritt oppover og sjekk foreldernoden. Hvis N er venstre barn så er foreldernoden svaret. Hvis N er høyre barn må man videre opp i treet helt til noden man kommer fra er venstre barn. Hvis man kommer opp til rota og fortsatt er høyre barn så eksisterer ikke neste høyere.
    - Finn (ulikhetssøk): Bruk «finn første over» for å finne startelementet. Deretter kjør neste helt til du får noe som er utenfor søkerommet.

- Minimum: Gå langs venstre subtre helt til en node ikke har et venstre barn. Da har du minste element. O(nivåer i treet)
- Maksimum: tilsvarer minimum
- Det k-ende minste element: Velg subtre basert på antall elementer i hvert subtre. Må vedlikeholde disse tallene i leggtil og fjern metodene.
- Legg til: Gå nedover i treet basert på sammenlikninger. Legg til elementet første sted det passer og det er en ledig plass i treet. Kast exception ved duplikat. O(nivåer i treet)
- Fjern: Finn elementet O(nivåer i treet) + fjern elementet.
    - Bladnode kan fjernes uten problem
    - Node med bare ett barn kan fjernes og forelderens barnepeker peker på barnet og ikke til noden som skal slettes
    - For rota så må ny rot settes
    - For node med to barn, finn minste node i høyre subtre og erstatt noden som fjernes med denne noden. Fjern så denne noden. Den vil ha maks ett barn.
- Traverser: Lister ut alle verdiene i treet. Inorder traversering gir en sortert liste med elementer.
- Hvor mange nivåer er det i treet?
    - Best case balansert tre O(log(n))
    - Worst case venstre eller høyredypt tre O(n). Dette blir resultatet hvis du setter inn en sortert liste!
    - Average case O(log(n)) ved tilfeldige innsettinger uten sletting for mye samme grunn som for quicksort. Average case inkludert sletting har vist seg å være rundt O(sqrt(n))

Krav til nøklene i et binært søketre

Nøkkelen må være sammenliknbar. Den må implementere __lt__ metoden, og alle nøklene i treet må være sammenliknbare med hverandre.

Dessuten: Hvis det som brukes for å sammenlikne nøklene endrer seg, må man si fra til trestrukturen gjennom å slette nøkkelen og sette den inn på nytt. Ellers vil algoritmene leite på feil sted!

AVL trær

- En type binære søketre som er garantert relativt balansert.
- Krever at for hver node så er forskjellen i dybde på venstre og høyre subtre maksimalt 1.
- Høyden til et AVL tre er O(log(n))
- Å teste høyden på et tre er en O(n) operasjon siden alle greinene må sjekkes. Dette må gjøres for hver innsetting i AVL-treet. For å kunne gjøre innsetting i O(log(n)) tid så må man lagre høydeinformasjon i nodene og vedlikeholde denne informasjonen ved innsetting, sletting og rotasjoner. Å vedlikeholde denne informasjonen tar O(log(n)) tid ved hver innsetting og sletting. Å lagre denne informasjonen tar O(n) plass, ett tall for hver node.
- For å sette inn og slette noder, må algoritmene opprettholde denne betingelsen
- Ved brudd må man rotere, enten single rotation eller double rotation
    - Se figurer for rotasjoner
    - Single rotation: En node A med barn B, venstre subtre B1 og B2, høyre subtre A1. Endre til at B blir forelder med subtre B1 og A, som har subtrær B2 og A1.
    - Double rotation: Figur eksempel eget notat.
    - Algoritme for enkelt venstrerotasjon

        - Kall nodene «forelder», «noden», «vb», «t1», «t2» og «t2»
        - Forelder sitt venstrebarn skal settes lik vb. Hvis forelder er None, sett rot lik vb
        - Sett vb sin forelder lik forelder.
        - Mellomlagre en referanse til t2
        - Sett vb sitt høyrebarn lik noden
        - Sett noden sin forelder lik vb
        - Sett noden sitt venstrebarn lik t2
        - Hvis t2 ikke er None, sett t2 sin forelder lik node
- Gå gjennom innsetting i timene. Sletting er frivillig tilleggsoppgave til øving 6 og blir gjennomgått under gjennomgangen av denne øvingen.

Eksempler på bruk av binære søketrær og andre søkestrukturer

- Lag «database» over kjøretøy med en trebasert implementasjon av Map grensesnittet, Map registreringsnummer -> kjøretøy. Bruker en Map implementasjon og ikke en Set implementasjon.
- Lag et kortspill hvor du ønsker at hånda holdes sortert etter at spilleren har trukket et nytt kort, og hvor det skal være lett å sjekke om spilleren har et gitt kort.
    - Merk at denne vil kreve en TreeSet implementasjon som bruker en eksplisitt komparator i stedet for < operatoren siden ulike kortspill vil sortere kortene ulikt.
    - Andre eksempler på dynamiske strukturer som holdes sortert slik som en telefonkatalog (Map <navn -> telefonnummer> som skal kunne skrives ut)

K-minste verdi problemet / retur til Selection problemet

- Problem: Finn det k-ende minste (eller største) elementet. Eksempel finn det 3. minste elementet.
- Løsning på eksisterende tre: Lag en inorder iterator og gå k steg fram.
    - Kjøretid O(log(n) + k)
- Kan gjøre det bedre ved å lagre størrelsen på hvert subtre i rota til subtreet. Størrelsen må vedlikeholdes i add, remove, removemin og rotasjoner.
- Rekursiv algoritme O(log(n)) kjøretid, start i rota:
    - Sjekk størrelsen på venstre barn, S_V
    - Hvis $S_V = k-1$, så er det noden vi står i som er k-ende minst, returner verdien i denne noden
    - Hvis $S_V > k-1$, så er k-ende minste verdi i venstre subtre. Kall metoden rekursivt på venstre subtre
    - Hvis $S_V < k-1$, så er k-ende minste verdi i høyre subtre. Kall metoden rekursivt på høyre subtre med ny_k = gammel_k $-S_V - 1$
- Vedlikehold av høydeinformasjon
    - Etter innsetting uten rotasjoner, øk størrelsen til forelder og alle forgjengere (forelder til forelder osv. opp til rota) med 1.
    - For enkeltrotasjon: Etter rotasjonen er utført, sett høyden til den originale noden lik høyden til barna den har nå + 1. Sett deretter høyden til det tidligere barnet som nå er forelder lik høyden til dens nåværende barn + 1.
    - For dobbeltrotasjon, sørg for at høydeinformasjonen er vedlikeholdt etter hver enkeltrotasjon.

Red-Black trær og AA trær

Problem med AVL-trær: Innsetting og sletting krever ikke bare et pass nedover i treet, men også et pass oppover igjen.

Red-Black trær og AA trær er alternativer til AVL trær. De er binære søketrær som er garantert relativt balanserte og som har fordelen sammenliknet med AVL-trær at de krever bare ett pass nedover i treet. Prisen for det er at de (spesielt Red-Black trær) er ganske kompliserte å implementere. De er derfor ikke pensum i dette faget.